告知我们有关下载 PDF 体验的信息。

C# 语言参考

语言参考为初学者和经验丰富的 C# 和 .NET 开发人员提供了有关 C# 语法和习惯用法的非正式参考。

C# 语言参考

概述

C# 语言策略

参考

C# 关键字

C# 运算符和表达式

配置语言版本

C# 语言规范 - C# 8 正在起草期间

新增功能

新变化

C# 13 中的新增功能

C# 12 中的新增功能

C# 11 中的新增功能

参考

C# 编译器中的重大更改

版本兼容性

保持联系

参考

.NET 开发人员社区

YouTube

Twitter

C# 语言版本控制

项目 • 2024/10/31

最新的 C# 编译器根据项目的一个或多个目标框架确定默认语言版本。 Visual Studio 不提供用于更改值的 UI，但可以通过编辑 .csproj 文件来更改值。 此默认选择可确保使用与目标框架兼容的最新语言版本。 你将从访问与项目目标兼容的最新语言功能中受益。 此默认选择还可确保不会使用需要类型或运行时行为在目标框架中不可用的语言。 选择比默认版本更高的语言版本可能导致难以诊断编译时和运行时错误。

C# 13 仅在 .NET 9 及更高版本上受支持。 C# 12 仅在 .NET 8 及更高版本上受到支持。 C# 11 仅在 .NET 7 及更高版本上受支持。

有关 Visual Studio 版本支持的 NET 版本的详细信息，请查看 Visual Studio 平台兼容性页面。 查看 C# 的 Mono 页面，了解 Mono 与 C# 版本的兼容性。

默认值

编译器根据以下规则确定默认值：

展开表

目标	版本	C# 语言版本的默认值
.NET	9.x	C# 13
.NET	8.x	C# 12
.NET	7.x	C# 11
.NET	6.x	C# 10
.NET	5.x	C# 9.0
.NET Core	3.x	C# 8.0
.NET Core	2.x	C# 7.3
.NET Standard	2.1	C# 8.0
.NET Standard	2.0	C# 7.3
.NET Standard	1.x	C# 7.3
.NET Framework	全部	C# 7.3

如果你的项目是以具有相应预览语言版本的 `preview` 框架为目标，那么使用的语言版本是预览语言版本。 你可在任何环境中使用该预览版提供的最新功能，而不会影响面向已发布 .NET Core 版本的项目。

C# 语言版本引用

下表显示当前所有 C# 语言版本。 较旧的编译器可能无法理解每个值。 如果安装了最新的 .NET SDK，则可以访问列出的所有内容。

展开表

值	含义
`preview`	编译器接受最新预览版中的所有有效语言语法。
`latest`	编译器接受最新发布的编译器版本（包括次要版本）中的语法。
`latestMajor` 或 `default`	编译器接受最新发布的编译器主要版本中的语法。
`13.0`	编译器只接受 C# 13 或更低版本中所含的语法。
`12.0`	编译器只接受 C# 12 或更低版本中所含的语法。
`11.0`	编译器只接受 C# 11 或更低版本中包含的语法。
`10.0`	编译器只接受 C# 10 或更低版本中所含的语法。
`9.0`	编译器只接受 C# 9 或更低版本中所含的语法。
`8.0`	编译器只接受 C# 8.0 或更低版本中所含的语法。
`7.3`	编译器只接受 C# 7.3 或更低版本中所含的语法。
`7.2`	编译器只接受 C# 7.2 或更低版本中所含的语法。
`7.1`	编译器只接受 C# 7.1 或更低版本中所含的语法。
`7`	编译器只接受 C# 7.0 或更低版本中所含的语法。
`6`	编译器只接受 C# 6.0 或更低版本中所含的语法。
`5`	编译器只接受 C# 5.0 或更低版本中所含的语法。
`4`	编译器只接受 C# 4.0 或更低版本中所含的语法。
`3`	编译器只接受 C# 3.0 或更低版本中所含的语法。

值	含义
`ISO-2` 或 `2`	编译器只接受 ISO/IEC 23270:2006 C# (2.0) 中所含的语法。
`ISO-1` 或 `1`	编译器只接受 ISO/IEC 23270:2003 C# (1.0/1.2) 中所含的语法。

ⓘ 备注

使用 `default` 值指定 LangVersion 与省略 LangVersion 选项不同。 指定 `default` 会使用编译器支持的语言的最新版本，而不考虑目标框架。 例如，从当前版本的 Visual Studio 版本 2022 生成面向 .NET 6 的项目时，如果未指定 LangVersion，则使用 C# 10，但如果 LangVersion 设置为 `default`，则使用 C# 12。

配置 C# 语言版本

项目 • 2024/12/20

> **⚠ 警告**
>
> 不建议将 `LangVersion` 元素设置为 `latest`。 `latest` 设置意味着已安装的编译器将使用其最新版本。 这可以在计算机之间进行更改，从而使得生成不可靠。 此外，它还支持可能需要当前 SDK 中不包含的运行时或库功能的语言功能。

如果必须明确指定 C# 版本，可以通过以下几种方式实现：

- 手动编辑项目文件。
- 为子目录中的多个项目设置语言版本。
- 配置 LangVersion 编译器选项。

> **提示**
>
> 可以在“项目属性”页查看 Visual Studio 中的语言版本。 在“生成”选项卡下，“高级”窗格将显示所选版本。
>
> 若要了解当前使用的语言版本，请在代码中添加 `#error version`（区分大小写）。 这样做可使编译器报告编译器错误 CS8304，并显示一条消息，其中包含正在使用的编译器版本和当前选择的语言版本。 有关详细信息，请参阅 **#error（C# 参考）**。

为什么无法在 Visual Studio 中选择其他 C# 版本

在 Visual Studio 中，可能会禁用通过 UI 更改语言版本的选项，因为默认版本与项目的目标框架（`TFM`） 保持一致。 此默认配置可确保语言功能和运行时支持之间的兼容性。

例如，将目标 `TFM`（例如，从 .NET 6 更改为 .NET 9）将相应地将语言版本从 C# 10 更新为 C# 13。 此方法可防止运行时兼容性问题，并最大程度地减少由于语言功能不受支持而导致的意外生成错误。

如果需要与自动选择的语言版本不同的特定语言版本，请参阅下面的方法来直接替代项目文件中的默认设置。

编辑项目文件

可在项目文件中设置语言版本。 例如，如果你明确希望访问预览功能，请添加如下元素：

```xml
<PropertyGroup>
   <LangVersion>preview</LangVersion>
</PropertyGroup>
```

值 `preview` 使用编译器支持的最新可用的预览 C# 语言版本。

配置多个项目

若要配置多个项目，可以创建一个包含 元素的 `<LangVersion>` 文件（通常在解决方案目录中）。 将以下设置添加到 *Directory.Build.props* 文件：

```xml
<Project>
 <PropertyGroup>
   <LangVersion>preview</LangVersion>
 </PropertyGroup>
</Project>
```

包含该文件的目录的所有子目录中的版本现在使用 C# 预览版。 有关详细信息，请查看自定义生成。

C# 语言版本引用

> **ⓘ 重要**
>
> 不支持使用比目标 TFM 关联的版本更新的 C# 语言版本。

下表显示当前所有 C# 语言版本。 较旧的编译器可能无法理解每个值。 如果安装了最新的 .NET SDK，则可以访问列出的所有内容。

展开表

值	含义
`preview`	编译器接受最新预览版中的所有有效语言语法。

值	含义
`latest`	编译器接受最新发布的编译器版本（包括次要版本）中的语法。
`latestMajor` 或 `default`	编译器接受最新发布的编译器主要版本中的语法。
`13.0`	编译器只接受 C# 13 或更低版本中所含的语法。
`12.0`	编译器只接受 C# 12 或更低版本中所含的语法。
`11.0`	编译器只接受 C# 11 或更低版本中包含的语法。
`10.0`	编译器只接受 C# 10 或更低版本中所含的语法。
`9.0`	编译器只接受 C# 9 或更低版本中所含的语法。
`8.0`	编译器只接受 C# 8.0 或更低版本中所含的语法。
`7.3`	编译器只接受 C# 7.3 或更低版本中所含的语法。
`7.2`	编译器只接受 C# 7.2 或更低版本中所含的语法。
`7.1`	编译器只接受 C# 7.1 或更低版本中所含的语法。
`7`	编译器只接受 C# 7.0 或更低版本中所含的语法。
`6`	编译器只接受 C# 6.0 或更低版本中所含的语法。
`5`	编译器只接受 C# 5.0 或更低版本中所含的语法。
`4`	编译器只接受 C# 4.0 或更低版本中所含的语法。
`3`	编译器只接受 C# 3.0 或更低版本中所含的语法。
`ISO-2` 或 `2`	编译器只接受 ISO/IEC 23270:2006 C# (2.0) 中所含的语法。
`ISO-1` 或 `1`	编译器只接受 ISO/IEC 23270:2003 C# (1.0/1.2) 中所含的语法。

值类型（C# 参考）

项目 • 2024/10/25

值类型和引用类型是 C# 类型的两个主要类别。 值类型的变量包含类型的实例。 它不同于引用类型的变量，后者包含对类型实例的引用。 默认情况下，在分配中，通过将实参传递给方法并返回方法结果来复制变量值。 对于值类型变量，会复制相应的类型实例。 以下示例演示了该行为：

C#

```
using System;

public struct MutablePoint
{
    public int X;
    public int Y;

    public MutablePoint(int x, int y) => (X, Y) = (x, y);

    public override string ToString() => $"({X}, {Y})";
}

public class Program
{
    public static void Main()
    {
        var p1 = new MutablePoint(1, 2);
        var p2 = p1;
        p2.Y = 200;
        Console.WriteLine($"{nameof(p1)} after {nameof(p2)} is modified:
{p1}");
        Console.WriteLine($"{nameof(p2)}: {p2}");

        MutateAndDisplay(p2);
        Console.WriteLine($"{nameof(p2)} after passing to a method: {p2}");
    }

    private static void MutateAndDisplay(MutablePoint p)
    {
        p.X = 100;
        Console.WriteLine($"Point mutated in a method: {p}");
    }
}
// Expected output:
// p1 after p2 is modified: (1, 2)
// p2: (1, 200)
// Point mutated in a method: (100, 200)
// p2 after passing to a method: (1, 200)
```

如前面的示例所示，对值类型变量的操作只影响存储在变量中的值类型实例。

如果值类型包含引用类型的数据成员，则在复制值类型实例时，只会复制对引用类型实例的引用。 副本和原始值类型实例都具有对同一引用类型实例的访问权限。 以下示例演示了该行为：

C#

```
using System;
using System.Collections.Generic;

public struct TaggedInteger
{
    public int Number;
    private List<string> tags;

    public TaggedInteger(int n)
    {
        Number = n;
        tags = new List<string>();
    }

    public void AddTag(string tag) => tags.Add(tag);

    public override string ToString() => $"{Number} [{string.Join(", ",
tags)}]";
}

public class Program
{
    public static void Main()
    {
        var n1 = new TaggedInteger(0);
        n1.AddTag("A");
        Console.WriteLine(n1);  // output: 0 [A]

        var n2 = n1;
        n2.Number = 7;
        n2.AddTag("B");

        Console.WriteLine(n1);  // output: 0 [A, B]
        Console.WriteLine(n2);  // output: 7 [A, B]
    }
}
```

ⓘ 备注

若要使代码更不易出错、更可靠，请定义并使用不可变的值类型。 本文仅为演示目的使用可变值类型。

值类型的种类以及类型约束

值类型可以是以下种类之一：

- 结构类型，用于封装数据和相关功能
- 枚举类型，由一组命名常数定义，表示一个选择或选择组合

可为 null 值类型 `T?` 表示其基础值类型 `T` 的所有值及额外的 null 值。 不能将 `null` 分配给值类型的变量，除非它是可为 null 的值类型。

你可使用 struct 约束指定类型参数为不可为 null 的值类型。 结构类型和枚举类型都满足 `struct` 约束。 可在基类约束中使用 `System.Enum`（称为枚举约束），以指定类型参数为枚举类型。

内置值类型

C# 提供以下内置值类型，也称为“简单类型”：

- 整型数值类型
- 浮点型数值类型
- bool，表示布尔值
- char，表示 Unicode UTF-16 字符

所有简单值都是结构类型，它们与其他结构类型的不同之处在于，它们允许特定的额外操作：

- 可以使用文字为简单类型提供值。
  例如， `'A'` 是类型的 `char` 文本， `2001` 是类型的 `int` 文本，是类型的文本，是 `12.34m` 该类型的 `decimal` 文本。

- 可以使用 const 关键字声明简单类型的常数。
  例如，可以定义 `const decimal = 12.34m`。
  不能具有其他结构类型的常数。

- 常数表达式的操作数都是简单类型的常数，在编译时进行评估。

值元组是值类型，而不是简单类型。

C# 语言规范

有关更多信息，请参阅 C# 语言规范的以下部分：

- 值类型

- 简单类型
- 变量

另请参阅

- System.ValueType
- 引用类型

整型数值类型（C# 参考）

项目 • 2023/05/10

整型数值类型 表示整数。 所有的整型数值类型均为值类型。 它们还是简单类型，可以使用文本进行初始化。 所有整型数值类型都支持算术、位逻辑、比较和相等运算符。

整型类型的特征

C# 支持以下预定义整型类型：

C# 类型/关键字	范围	大小	.NET 类型
`sbyte`	-128 到 127	8 位带符号整数	System.SByte
`byte`	0 到 255	无符号的 8 位整数	System.Byte
`short`	-32,768 到 32,767	有符号 16 位整数	System.Int16
`ushort`	0 到 65,535	无符号 16 位整数	System.UInt16
`int`	-2,147,483,648 到 2,147,483,647	带符号的 32 位整数	System.Int32
`uint`	0 到 4,294,967,295	无符号的 32 位整数	System.UInt32
`long`	-9,223,372,036,854,775,808 到 9,223,372,036,854,775,807	64 位带符号整数	System.Int64
`ulong`	0 到 18,446,744,073,709,551,615	无符号 64 位整数	System.UInt64
`nint`	取决于（在运行时计算的）平台	带符号的 32 位或 64 位整数	System.IntPtr
`nuint`	取决于（在运行时计算的）平台	无符号的 32 位或 64 位整数	System.UIntPtr

在除最后两行之外的所有表行中，最左侧列中的每个 C# 类型关键字都是相应 .NET 类型的别名。 关键字和 .NET 类型名称是可互换的。 例如，以下声明声明了相同类型的变量：

C#

```
int a = 123;
System.Int32 b = 123;
```

表的最后两行中的 `nint` 和 `nuint` 类型是本机大小的整数。 从 C# 9.0 起，可以使用 `nint` 和 `nuint` 关键字定义本机大小的整数。 在 32 位进程中运行时有 32 位的整数，在 64 位进程中运行时有 64 位的整数。 这些类型可用于互操作方案、低级别的库，可用于在广泛使用整数运算的方案中提高性能。

本机大小的整数类型在内部表示为 .NET 类型 System.IntPtr 和 System.UIntPtr。 从 C# 11 开始，`nint` 和 `nuint` 类型是基础类型的别名。

每个整型类型的默认值都为零 `0`。

每个整型类型都有 `MinValue` 和 `MaxValue` 属性，提供该类型的最小值和最大值。 这些属性是编译时常量，但本机大小类型（`nint` 和 `nuint`）的情况除外。 对于本地大小的类型，`MinValue` 和 `MaxValue` 属性是在运行时计算的。 这些类型的大小取决于进程设置。

System.Numerics.BigInteger 结构用于表示没有上限或下限的带符号整数。

整数文本

整数文本可以是

- 十进制：不使用任何前缀
- 十六进制：使用 `0x` 或 `0X` 前缀
- 二进制：使用 `0b` 或 `0B` 前缀

下面的代码演示每种类型的示例：

```
C#
```

```
var decimalLiteral = 42;
var hexLiteral = 0x2A;
var binaryLiteral = 0b_0010_1010;
```

前面的示例还演示了如何将 `_` 用作数字分隔符。 可以将数字分隔符用于所有类型的数字文本。

整数文本的类型由其后缀确定，如下所示：

- 如果文本没有后缀，则其类型为以下类型中可表示其值的第一个类型：`int`、`uint`、`long`、`ulong`。

> **备注**

文本解释为正值。 例如，文本 `0xFF_FF_FF_FF` 表示 `uint` 类型的数字 `4294967295`，但其位表现形式与 `int` 类型的数字 `-1` 相同。 如果需要特定类型的值，请将文本强制转换为该类型。 如果文本值无法以目标类型表示，请使用运算符 `unchecked`。 示例：`unchecked((int)0xFF_FF_FF_FF)` 生成 `-1`。

- 如果文本以 `U` 或 `u` 为后缀，则其类型为以下类型中可表示其值的第一个类型：`uint`、`ulong`。
- 如果文本以 `L` 或 `l` 为后缀，则其类型为以下类型中可表示其值的第一个类型：`long`、`ulong`。

> **备注**
>
> 可以使用小写字母 `l` 作为后缀。 但是，这会生成一个编译器警告，因为字母 `l` 可能与数字 `1` 混淆。 为清楚起见，请使用 `L`。

- 如果文本的后缀为 `UL`、`Ul`、`uL`、`ul`、`LU`、`Lu`、`lU` 或 `lu`，则其类型为 `ulong`。

如果由整数字面量所表示的值超出了 UInt64.MaxValue，则将出现编译器错误 CS1021。

如果确定的整数文本的类型为 `int`，且文本所表示的值位于目标类型的范围内，则该值可以隐式转换为 `sbyte`、`byte`、`short`、`ushort`、`uint`、`ulong`、`nint` 或 `nuint`：

```
C#
```

```
byte a = 17;
byte b = 300;   // CS0031: Constant value '300' cannot be converted to a
'byte'
```

如前面的示例所示，如果文本的值不在目标类型的范围内，则发生编译器错误 CS0031。

还可以使用强制转换将整数文本所表示的值转换为除确定的文本类型之外的类型：

```
C#
```

```
var signedByte = (sbyte)42;
var longVariable = (long)42;
```

转换

可以将任何整型数值类型转换为其他整数数值类型。 如果目标类型可以存储源类型的所有值，则转换是隐式的。 否则，需要使用强制转换表达式来执行显式转换。 有关详细信息，请参阅内置数值转换。

本机大小的整数

本机大小的整数类型具有特殊行为，因为存储是由目标计算机上的自然整数大小决定的。

- 若要在运行时获取本机大小的整数大小，可以使用 `sizeof()`。 但是，必须在不安全的上下文中编译代码。 例如：

  C#

  ```
  Console.WriteLine($"size of nint = {sizeof(nint)}");
  Console.WriteLine($"size of nuint = {sizeof(nuint)}");

  // output when run in a 64-bit process
  //size of nint = 8
  //size of nuint = 8

  // output when run in a 32-bit process
  //size of nint = 4
  //size of nuint = 4
  ```

  也可以通过静态 IntPtr.Size 和 UIntPtr.Size 属性获得等效的值。

- 若要在运行时获取本机大小的整数的最小值和最大值，请将 `MinValue` 和 `MaxValue` 用作 `nint` 和 `nuint` 关键字的静态属性，如以下示例中所示：

  C#

  ```
  Console.WriteLine($"nint.MinValue = {nint.MinValue}");
  Console.WriteLine($"nint.MaxValue = {nint.MaxValue}");
  Console.WriteLine($"nuint.MinValue = {nuint.MinValue}");
  Console.WriteLine($"nuint.MaxValue = {nuint.MaxValue}");

  // output when run in a 64-bit process
  //nint.MinValue = -9223372036854775808
  //nint.MaxValue = 9223372036854775807
  //nuint.MinValue = 0
  //nuint.MaxValue = 18446744073709551615

  // output when run in a 32-bit process
  //nint.MinValue = -2147483648
  //nint.MaxValue = 2147483647
  //nuint.MinValue = 0
  //nuint.MaxValue = 4294967295
  ```

- 可以使用以下范围内的常量值：
  - 对于 `nint`：Int32.MinValue 到 Int32.MaxValue。
  - 对于 `nuint`：UInt32.MinValue 到 UInt32.MaxValue。

- 编译器可将这些类型隐式和显式转换为其他数值类型。 有关详细信息，请参阅内置数值转换。

- 没有适用于本机大小整数文本的直接语法。 没有后缀可表示文本是本机大小整数，例如 `L` 表示 `long`。 可以改为使用其他整数值的隐式或显式强制转换。 例如：

```
C#
```

```
nint a = 42
nint a = (nint)42;
```

C# 语言规范

有关更多信息，请参阅 C# 语言规范的以下部分：

- 整型类型
- 整数文本
- C# 9 - 本机大小的整型类型
- C# 11 - 数值 IntPtr 和 `UIntPtr

请参阅

- C# 参考
- 值类型
- 浮点类型
- 标准数字格式字符串
- .NET 中的数字

浮点数值类型（C# 引用）

项目 • 2024/04/22

浮点数值类型表示实数。 所有浮点型数值类型均为值类型。 它们还是简单类型，可以使用文本进行初始化。 所有浮点数值类型都支持算术、比较和相等运算符。

浮点类型的特征

C# 支持以下预定义浮点类型：

展开表

C# 类型/关键字	大致范围	精度	大小	.NET 类型
`float`	±1.5 x 10^{-45} 至 ±3.4 x 10^{38}	大约 6-9 位数字	4 个字节	System.Single
`double`	±5.0 × 10^{-324} 到 ±1.7 × 10^{308}	大约 15-17 位数字	8 个字节	System.Double
`decimal`	±1.0 x 10^{-28} 至 ±7.9228 x 10^{28}	28-29 位	16 个字节	System.Decimal

在上表中，最左侧列中的每个 C# 类型关键字都是相应 .NET 类型的别名。 它们是可互换的。 例如，以下声明声明了相同类型的变量：

C#

```
double a = 12.3;
System.Double b = 12.3;
```

每个浮点类型的默认值都为零，`0`。 每个浮点类型都有 `MinValue` 和 `MaxValue` 常量，提供该类型的最小值和最大有限值。 `float` and `double` 类型还提供可表示非数字和无穷大值的常量。 例如，`double` 类型提供以下常量：Double.NaN、Double.NegativeInfinity 和 Double.PositiveInfinity。

当所需的精度由小数点右侧的位数决定时，`decimal` 类型是合适的。 此类数字通常用于财务应用程序、货币金额（例如 $1.00）、利率（例如 2.625%）等。 精确到只有一个小数的偶数用 `decimal` 类型处理会更准确：例如，0.1 可以由 `decimal` 实例精确表示，而没有精确表示 0.1 的 `double` 或 `float` 实例。 由于数值类型存在这种差异，因此当你对十进制数据使用 `double` 或 `float` 时，算术计算可能会出现意外的舍入错误。 当优化性能比确保准确度更重要时，可以使用 `double` 代替 `decimal`。 然而，除了大多数计算密集型

应用程序之外，所有应用程序都不会注意到性能上的任何差异。 避免使用 `decimal` 的另一个可能原因是为了最大限度地降低存储需求。 例如，ML.NET 使用 `float`，因为对于非常大的数据集，4 个字节与 16 个字节之间的差异合乎情理。 有关详细信息，请参阅 System.Decimal。

可在表达式中将整型类型与 `float` 和 `double` 类型混合使用功能。 在这种情况下，整型类型隐式转换为其中一种浮点类型，且必要时，`float` 类型隐式转换为 `double`。 此表达式的计算方式如下：

- 如果表达式中有 `double` 类型，则表达式在关系比较和相等比较中求值得到 `double` 或 bool。
- 如果表达式中没有 `double` 类型，则表达式在关系比较和相等比较中求值得到 `float` 或 `bool`。

你还可在表达式中混合使用整型类型和 `decimal` 类型。 在这种情况下，整型类型隐式转换为 `decimal` 类型，并且表达式在关系比较和相等比较中求值得到 `decimal` 或 `bool`。

不能在表达式中将 `decimal` 类型与 `float` 和 `double` 类型混合使用。 在这种情况下，如果你想要执行算术运算、比较运算或相等运算，则必须将操作数显式转换为 `decimal` 或反向转换，如下例所示：

C#

```
double a = 1.0;
decimal b = 2.1m;
Console.WriteLine(a + (double)b);
Console.WriteLine((decimal)a + b);
```

可以使用标准数字格式字符串或自定义数字格式字符串设置浮点值的格式。

真实文本

真实文本的类型由其后缀确定，如下所示：

- 不带后缀的文本或带有 `d` 或 `D` 后缀的文本的类型为 `double`
- 带有 `f` 或 `F` 后缀的文本的类型为 `float`
- 带有 `m` 或 `M` 后缀的文本的类型为 `decimal`

下面的代码演示每种类型的示例：

C#

```
double d = 3D;
d = 4d;
d = 3.934_001;

float f = 3_000.5F;
f = 5.4f;

decimal myMoney = 3_000.5m;
myMoney = 400.75M;
```

前面的示例还演示了如何将 `_` 用作数字分隔符。 可以将数字分隔符用于所有类型的数字文本。

还可以使用科学记数法，即指定真实文本的指数部分，如以下示例所示：

C#

```
double d = 0.42e2;
Console.WriteLine(d);  // output 42

float f = 134.45E-2f;
Console.WriteLine(f);  // output: 1.3445

decimal m = 1.5E6m;
Console.WriteLine(m);  // output: 1500000
```

转换

浮点数值类型之间只有一种隐式转换：从 `float` 到 `double`。 但是，可以使用显式强制转换将任何浮点类型转换为任何其他浮点类型。 有关详细信息，请参阅内置数值转换。

C# 语言规范

有关更多信息，请参阅 C# 语言规范的以下部分：

- 浮点类型
- 十进制类型
- 真实文本

另请参阅

- 值类型
- 整型类型

- 标准数字格式字符串
- .NET 中的数字
- System.Numerics.Complex

在 GitHub 上与我们协作

可以在 GitHub 上找到此内容的源，还可以在其中创建和查看问题和拉取请求。 有关详细信息，请参阅参与者指南。

.NET 反馈

.NET 是一个开放源代码项目。 选择一个链接以提供反馈：

提出文档问题

提供产品反馈

内置数值转换（C# 参考）

项目 • 2023/04/07

C# 提供了一组整型和浮点数值类型。 任何两种数值类型之间都可以进行隐式或显式转换。 必须使用强制转换表达式来执行显式转换。

隐式数值转换

下表显示内置数值类型之间的预定义隐式转换：

From	到
sbyte	`short`、`int`、`long`、`float`、`double`、`decimal` 或 `nint`。
byte	`short`、`ushort`、`int`、`uint`、`long`、`ulong`、`float`、`double`、`decimal`、`nint` 或 `nuint`
short	`int`、`long`、`float`、`double`、`decimal` 或 `nint`
ushort	`int`、`uint`、`long`、`ulong`、`float`、`double`、`decimal`、`nint` 或 `nuint`
int	`long`、`float`、`double`、`decimal` 或 `nint`
uint	`long`、`ulong`、`float`、`double`、`decimal` 或 `nuint`
long	`float`、`double` 或 `decimal`
ulong	`float`、`double` 或 `decimal`
float	`double`
nint	`long`、`float`、`double` 或 `decimal`
nuint	`ulong`、`float`、`double` 或 `decimal`

> **备注**
>
> 从 `int`、`uint`、`long`、`ulong`、`nint` 或 `nuint` 到 `float` 的隐式转换以及从 `long`、`ulong`、`nint` 或 `nuint` 到 `double` 的隐式转换可能会丢失精准率，但绝不会丢失一个数量级。 其他隐式数值转换不会丢失任何信息。

另请注意

- 任何整型数值类型都可以隐式转换为任何浮点数值类型。

- 不存在针对 `byte` 和 `sbyte` 类型的隐式转换。 不存在从 `double` 和 `decimal` 类型的隐式转换。
- `decimal` 类型和 `float` 或 `double` 类型之间不存在隐式转换。
- 类型 `int` 的常量表达式的值（例如，由整数文本所表示的值）如果在目标类型的范围内，则可隐式转换为 `sbyte`、`byte`、`short`、`ushort`、`uint`、`ulong`、`nint` 或 `nuint`：

  ```
  C#

  byte a = 13;
  byte b = 300;  // CS0031: Constant value '300' cannot be converted to a
  'byte'
  ```

  如前面的示例所示，如果该常量值不在目标类型的范围内，则发生编译器错误 CS0031。

显式数值转换

下表显示不存在隐式转换的内置数值类型之间的预定义显式转换：

From	到
sbyte	`byte`、`ushort`、`uint`、`ulong` 或 `nuint`
byte	`sbyte`
short	`sbyte`、`byte`、`ushort`、`uint`、`ulong` 或 `nuint`
ushort	`sbyte`、`byte` 或 `short`
int	`sbyte`、`byte`、`short`、`ushort`、`uint`、`ulong` 或 `nuint`。
uint	`sbyte`、`byte`、`short`、`ushort`、`int` 或 `nint`
long	`sbyte`、`byte`、`short`、`ushort`、`int`、`uint`、`ulong`、`nint` 或 `nuint`
ulong	`sbyte`、`byte`、`short`、`ushort`、`int`、`uint`、`long`、`nint` 或 `nuint`
float	`sbyte`、`byte`、`short`、`ushort`、`int`、`uint`、`long`、`ulong`、`decimal`、`nint` 或 `nuint`
double	`sbyte`、`byte`、`short`、`ushort`、`int`、`uint`、`long`、`ulong`、`float`、`decimal`、`nint` 或 `nuint`
decimal	`sbyte`、`byte`、`short`、`ushort`、`int`、`uint`、`long`、`ulong`、`float`、`double`、`nint` 或 `nuint`

From	到
nint	`sbyte`、`byte`、`short`、`ushort`、`int`、`uint`、`ulong` 或 `nuint`
nuint	`sbyte`、`byte`、`short`、`ushort`、`int`、`uint`、`long` 或 `nint`

> **备注**
>
> 显式数值转换可能会导致数据丢失或引发异常，通常为 OverflowException。

另请注意：

- 将整数类型的值转换为另一个整数类型时，结果取决于溢出检查上下文。 在已检查的上下文中，如果源值在目标类型的范围内，则转换成功。 否则会引发 OverflowException。 在未检查的上下文中，转换始终成功，并按如下方式进行：
  - 如果源类型大于目标类型，则通过放弃其"额外"最高有效位来截断源值。 结果会被视为目标类型的值。
  - 如果源类型小于目标类型，则源值是符号扩展或零扩展，以使其与目标类型的大小相同。 如果源类型带符号，则是符号扩展；如果源类型是无符号的，则是零扩展。 结果会被视为目标类型的值。
  - 如果源类型与目标类型的大小相同，则源值将被视为目标类型的值。
- 将 `decimal` 值转换为整型类型时，此值会向零舍入到最接近的整数值。 如果生成的整数值处于目标类型的范围之外，则会引发 OverflowException。
- 将 `double` 或 `float` 值转换为整型类型时，此值会向零舍入到最接近的整数值。 如果生成的整数值处于目标类型范围之外，则结果会取决于溢出上下文。 在已检查的上下文中，引发 OverflowException；而在未检查的上下文中，结果是目标类型的未指定值。
- 将 `double` 转换为 `float` 时，`double` 值舍入为最接近的 `float` 值。 如果 `double` 值太小或太大，无法匹配 `float` 类型，结果将为零或无穷大。
- 将 `float` 或 `double` 转换为 `decimal` 时，源值转换为 `decimal` 表示形式，并并五入到第 28 位小数后最接近的数（如有必要）。 根据源值的值，可能出现以下结果之一：
  - 如果源值太小，无法表示为 `decimal`，结果则为零。
  - 如果源值为 NaN（非数值）、无穷大或太大而无法表示为 `decimal`，则引发 OverflowException。

- 将 `decimal` 转换为 `float` 或 `double` 时，源值分别舍入为最接近的 `float` 或 `double` 值。

C# 语言规范

有关更多信息，请参阅 C# 语言规范的以下部分：

- 隐式数值转换
- 显式数值转换

请参阅

- C# 参考
- 强制转换和类型转换

bool（C# 参考）

项目 • 2024/04/22

`bool` 类型关键字是 .NET System.Boolean 结构类型的别名，它表示一个布尔值，可为 `true` 或 `false`。

若要使用 `bool` 类型的值执行逻辑运算，请使用布尔逻辑运算符。 `bool` 类型是 比较和相等运算符的结果类型。 `bool` 表达式可以是 if、do、while 和 for 语句中以及条件运算符 ?: 中的控制条件表达式。

`bool` 类型的默认值为 `false`。

文本

可使用 `true` 和 `false` 文本来初始化 `bool` 变量或传递 `bool` 值：

```
C#
```

```
bool check = true;
Console.WriteLine(check ? "Checked" : "Not checked");  // output: Checked

Console.WriteLine(false ? "Checked" : "Not checked");  // output: Not
checked
```

三值布尔逻辑

如需支持三值逻辑（例如，在使用支持三值布尔类型的数据库时），请使用可为空 `bool?` 类型。 对于 `bool?` 操作数，预定义的 `&` 和 `|` 运算符支持三值逻辑。 有关详细信息，请参阅布尔逻辑运算符一文的可以为 null 的布尔逻辑运算符部分。

有关可为空的值类型的详细信息，请参阅可为空的值类型。

转换

C# 仅提供了两个涉及 `bool` 类型的转换。 它们是对相应的可以为空的 `bool?` 类型的隐式转换以及对 `bool?` 类型的显式转换。 但是，.NET 提供了其他方法可用来转换到 `bool` 类型从或此类型进行转换。 有关详细信息，请参阅 System.Boolean API 参考页的转换为布尔值和从布尔值转换部分。

C# 语言规范

有关详细信息，请参阅 C# 语言规范中的 bool 类型部分。

另请参阅

- 值类型
- true 和 false 运算符

在 GitHub 上与我们协作

可以在 GitHub 上找到此内容的源，还可以在其中创建和查看问题和拉取请求。 有关详细信息，请参阅参与者指南。

.NET

.NET 反馈

.NET 是一个开放源代码项目。 选择一个链接以提供反馈：

提出文档问题

提供产品反馈

char（C# 参考）

项目 • 2024/04/22

`char` 类型关键字是 .NET System.Char 结构类型的别名，它表示 Unicode UTF-16 字符。

展开表

类型	范围	大小	.NET 类型
`char`	U+0000 到 U+FFFF	16 位	System.Char

`char` 类型的默认值为 `\0`，即 U+0000。

`char` 类型支持比较、相等、增量和减量运算符。 此外，对于 `char` 操作数，算数和逻辑位运算符对相应的字符代码执行操作，并得出 `int` 类型的结果。

字符串类型将文本表示为 `char` 值的序列。

文本

可以使用以下命令指定 `char` 值：

- 字符文本。
- Unicode 转义序列，它是 `\u` 后跟字符代码的十六进制表示形式（四个符号）。
- 十六进制转义序列，它是 `\x` 后跟字符代码的十六进制表示形式。

C#

```
var chars = new[]
{
    'j',
    '\u006A',
    '\x006A',
    (char)106,
};
Console.WriteLine(string.Join(" ", chars));  // output: j j j j
```

如前面的示例所示，你还可以将字符代码的值转换为相应的 `char` 值。

> **备注**
>
> 对于 Unicode 转义序列，必须指定全部四位十六进制值。 也就是说，`\u006A` 是一个有效的转义序列，而 `\u06A` 和 `\u6A` 是无效的。

对于十六进制转义序列，可以省略前导零。 也就是说，`\x006A`、`\x06A` 和 `\x6A` 转义序列是有效的，并且对应于同一个字符。

转换

`char` 类型可隐式转换为以下整型类型：`ushort`、`int`、`uint`、`long` 和 `ulong`。 它也可以隐式转换为内置浮点数值类型：`float`、`double` 和 `decimal`。 它可以显式转换为 `sbyte`、`byte` 和 `short` 整型类型。

无法将其他类型隐式转换为 `char` 类型。 但是，任何整型或浮点数值类型都可显式转换为 `char`。

C# 语言规范

有关详细信息，请参阅 C# 语言规范的整型类型部分。

另请参阅

- 值类型
- 字符串
- System.Text.Rune
- .NET 中的字符编码

在 GitHub 上与我们协作

可以在 GitHub 上找到此内容的源，还可以在其中创建和查看问题和拉取请求。 有关详细信息，请参阅参与者指南。

.NET

.NET 反馈

.NET 是一个开放源代码项目。 选择一个链接以提供反馈：

提出文档问题

提供产品反馈

枚举类型（C# 参考）

项目 • 2023/04/07

枚举类型 是由基础整型数值类型的一组命名常量定义的值类型。 若要定义枚举类型，请使用 `enum` 关键字并指定枚举成员 的名称：

```
C#

enum Season
{
    Spring,
    Summer,
    Autumn,
    Winter
}
```

默认情况下，枚举成员的关联常数值为类型 `int`；它们从零开始，并按定义文本顺序递增 1。 可以显式指定任何其他整数数值类型作为枚举类型的基础类型。 还可以显式指定关联的常数值，如下面的示例所示：

```
C#

enum ErrorCode : ushort
{
    None = 0,
    Unknown = 1,
    ConnectionLost = 100,
    OutlierReading = 200
}
```

不能在枚举类型的定义内定义方法。 若要向枚举类型添加功能，请创建扩展方法。

枚举类型 `E` 的默认值是由表达式 `(E)0` 生成的值，即使零没有相应的枚举成员也是如此。

可以使用枚举类型，通过一组互斥值或选项组合来表示选项。 若要表示选项组合，请将枚举类型定义为位标志。

作为位标志的枚举类型

如果希望枚举类型表示选项组合，请为这些选项定义枚举成员，以便单个选项成为位字段。 也就是说，这些枚举成员的关联值应该是 2 的幂。 然后，可以使用按位逻辑运算符 | 或 & 分别合并选项或交叉组合选项。 若要指示枚举类型声明位字段，请对其应用 Flags 属性。 如下面的示例所示，还可以在枚举类型的定义中包含一些典型组合。

C#

```
[Flags]
public enum Days
{
    None      = 0b_0000_0000,  // 0
    Monday    = 0b_0000_0001,  // 1
    Tuesday   = 0b_0000_0010,  // 2
    Wednesday = 0b_0000_0100,  // 4
    Thursday  = 0b_0000_1000,  // 8
    Friday    = 0b_0001_0000,  // 16
    Saturday  = 0b_0010_0000,  // 32
    Sunday    = 0b_0100_0000,  // 64
    Weekend   = Saturday | Sunday
}

public class FlagsEnumExample
{
    public static void Main()
    {
        Days meetingDays = Days.Monday | Days.Wednesday | Days.Friday;
        Console.WriteLine(meetingDays);
        // Output:
        // Monday, Wednesday, Friday

        Days workingFromHomeDays = Days.Thursday | Days.Friday;
        Console.WriteLine($"Join a meeting by phone on {meetingDays &
workingFromHomeDays}");
        // Output:
        // Join a meeting by phone on Friday

        bool isMeetingOnTuesday = (meetingDays & Days.Tuesday) ==
Days.Tuesday;
        Console.WriteLine($"Is there a meeting on Tuesday:
{isMeetingOnTuesday}");
        // Output:
        // Is there a meeting on Tuesday: False

        var a = (Days)37;
        Console.WriteLine(a);
        // Output:
        // Monday, Wednesday, Saturday
    }
}
```

有关详细信息和示例，请参阅 System.FlagsAttribute API 参考页和 System.Enum API 参考页的非独占成员和 Flags 属性部分。

System.Enum 类型和枚举约束

System.Enum 类型是所有枚举类型的抽象基类。 它提供多种方法来获取有关枚举类型及其值的信息。 有关更多信息和示例，请参阅 System.Enum API 参考页。

可在基类约束中使用 `System.Enum`（称为枚举约束），以指定类型参数为枚举类型。 所有枚举类型也都满足 `struct` 约束，此约束用于指定类型参数为不可为 null 的值类型。

转换

对于任何枚举类型，枚举类型与其基础整型类型之间存在显式转换。 如果将枚举值转换为其基础类型，则结果为枚举成员的关联整数值。

C#

```
public enum Season
{
    Spring,
    Summer,
    Autumn,
    Winter
}

public class EnumConversionExample
{
    public static void Main()
    {
        Season a = Season.Autumn;
        Console.WriteLine($"Integral value of {a} is {(int)a}");  // output:
Integral value of Autumn is 2

        var b = (Season)1;
        Console.WriteLine(b);  // output: Summer

        var c = (Season)4;
        Console.WriteLine(c);  // output: 4
    }
}
```

使用 Enum.IsDefined 方法来确定枚举类型是否包含具有特定关联值的枚举成员。

对于任何枚举类型，都存在分别与 System.Enum 类型的装箱和取消装箱相互转换。

C# 语言规范

有关更多信息，请参阅 C# 语言规范的以下部分：

- 枚举
- 枚举值和操作

- 枚举逻辑运算符
- 枚举比较运算符
- 显式枚举转换
- 隐式枚举转换

请参阅

- C# 参考
- 枚举格式字符串
- 设计准则 - 枚举设计
- 设计准则 - 枚举命名约定
- switch表达式
- switch 语句

结构类型（C# 参考）

项目 • 2024/12/31

结构类型（或 *struct type*）是一种可封装数据和相关功能的值类型。 使用 `struct` 关键字定义结构类型：

```C#
public struct Coords
{
    public Coords(double x, double y)
    {
        X = x;
        Y = y;
    }

    public double X { get; }
    public double Y { get; }

    public override string ToString() => $"({X}, {Y})";
}
```

有关 `ref struct` 和 `readonly ref struct` 类型的信息，请参阅参考结构类型一文。

结构类型具有*值语义*。 也就是说，结构类型的变量包含类型的实例。 默认情况下，在分配中，通过将参数传递给方法并返回方法结果来复制变量值。 对于结构类型变量，将复制该类型的实例。 有关更多信息，请参阅值类型。

通常，可以使用结构类型来设计以数据为中心的较小类型，这些类型只有很少的行为或没有行为。 例如，.NET 使用结构类型来表示数字（整数和实数）、布尔值、Unicode 字符以及时间实例。 如果侧重于类型的行为，请考虑定义一个类。 类类型具有*引用语义*。 也就是说，类类型的变量包含的是对类型的实例的引用，而不是实例本身。

由于结构类型具有值语义，因此建议定义*不可变*的结构类型。

`readonly` 结构

可以使用 `readonly` 修饰符来声明结构类型为不可变。 `readonly` 结构的所有数据成员都必须是只读的，如下所示：

- 任何字段声明都必须具有 readonly 修饰符
- 任何属性（包括自动实现的属性）都必须是只读的，或者是 init 只读的。 仅限 init 的资源库从 C# 版本 9 开始可用。

这样可以保证 `readonly` 结构的成员不会修改该结构的状态。 这意味着除构造函数外的其他实例成员是隐式 readonly。

> **备注**
>
> 在 `readonly` 结构中，可变引用类型的数据成员仍可改变其自身的状态。 例如，不能替换 List<T> 实例，但可以向其中添加新元素。

下面的代码使用 init-only 属性资源库定义 `readonly` 结构：

```C#
public readonly struct Coords
{
    public Coords(double x, double y)
    {
        X = x;
        Y = y;
    }

    public double X { get; init; }
    public double Y { get; init; }

    public override string ToString() => $"({X}, {Y})";
}
```

`readonly` 实例成员

还可以使用 `readonly` 修饰符来声明实例成员不会修改结构的状态。 如果不能将整个结构类型声明为 `readonly`，可使用 `readonly` 修饰符标记不会修改结构状态的实例成员。

在 `readonly` 实例成员内，不能分配到结构的实例字段。 但是，`readonly` 成员可以调用非 `readonly` 成员。 在这种情况下，编译器将创建结构实例的副本，并调用该副本上的非 `readonly` 成员。 因此，不会修改原始结构实例。

通常，将 `readonly` 修饰符应用于以下类型的实例成员：

- 方法：

  ```C#
  public readonly double Sum()
  {
  ```

```
    return X + Y;
}
```

还可以将 `readonly` 修饰符应用于可替代在 System.Object 中声明的方法的方法：

```C#
public readonly override string ToString() => $"({X}, {Y})";
```

- 属性和索引器：

```C#
private int counter;
public int Counter
{
    readonly get => counter;
    set => counter = value;
}
```

如果需要将 `readonly` 修饰符应用于属性或索引器的两个访问器，请在属性或索引器的声明中应用它。

> **备注**
>
> 编译器将 `get` 自动实现的属性**的访问器**声明为 `readonly`，无论属性声明中是否存在 `readonly` 修饰符。

可以将 `readonly` 修饰符应用于具有 `init` 访问器的属性或索引器：

```C#
public readonly double X { get; init; }
```

可以将 `readonly` 修饰符应用于结构类型的静态字段，但不能应用于任何其他静态成员，例如属性或方法。

编译器可以使用 `readonly` 修饰符进行性能优化。 有关详细信息，请参阅避免分配。

非破坏性变化

可以使用 with 表达式 生成具有修改指定属性和字段的结构类型实例的副本。 使用对象初始值设定项语法来指定要修改的成员及其新值，如以下示例所示：

C#

```
public readonly struct Coords
{
    public Coords(double x, double y)
    {
        X = x;
        Y = y;
    }

    public double X { get; init; }
    public double Y { get; init; }

    public override string ToString() => $"({X}, {Y})";
}

public static void Main()
{
    var p1 = new Coords(0, 0);
    Console.WriteLine(p1);  // output: (0, 0)

    var p2 = p1 with { X = 3 };
    Console.WriteLine(p2);  // output: (3, 0)

    var p3 = p1 with { X = 1, Y = 4 };
    Console.WriteLine(p3);  // output: (1, 4)
}
```

`record` 结构

可以定义记录结构类型。 记录类型提供用于封装数据的内置功能。 可同时定义 `record struct` 和 `readonly record struct` 类型。 记录结构不能是 ref struct。 有关详细信息和示例，请参阅记录。

内联数组

从 C# 12 开始，可以将*内联数组*声明为 `struct` 类型：

C#

```
[System.Runtime.CompilerServices.InlineArray(10)]
public struct CharBuffer
{
```

```
    private char _firstElement;
}
```

内联数组是包含相同类型的 N 个元素的连续块的结构。 它是一个安全代码，等效于仅在不安全代码中可用的固定缓冲区声明。 内联数组是具有以下特征的 `struct`：

- 它包含单个字段。
- 结构未指定显式布局。

此外，编译器还会验证 System.Runtime.CompilerServices.InlineArrayAttribute 属性：

- 必须大于零 (`> 0`)。
- 目标类型必须是结构。

在大多数情况下，可以像访问数组一样访问内联数组，以读取和写入值。 此外，还可以使用范围和索引运算符。

对内联数组的单个字段的类型有最小的限制。 它不能是指针类型：

C#

```
[System.Runtime.CompilerServices.InlineArray(10)]
public struct CharBufferWithPointer
{
    private unsafe char* _pointerElement;    // CS9184
}
```

但它可以是任何引用类型，也可以是任何值类型：

C#

```
[System.Runtime.CompilerServices.InlineArray(10)]
public struct CharBufferWithReferenceType
{
    private string _referenceElement;
}
```

几乎可以将内联数组与任何 C# 数据结构一起使用。

内联数组是一种高级语言功能。 它们适用于高性能方案，在这些方案中，内联的连续元素块比其他替代数据结构速度更快。 可以从功能规范中了解有关内联数组的详细信息

结构初始化和默认值

`struct` 类型的变量直接包含该 `struct` 类型的数据。 这会让未初始化的 `struct`（具有其默认值）和已初始化的 `struct`（通过构造值来存储一组值）之间存在区别。 例如，考虑下面的代码：

C#

```
public readonly struct Measurement
{
    public Measurement()
    {
        Value = double.NaN;
        Description = "Undefined";
    }

    public Measurement(double value, string description)
    {
        Value = value;
        Description = description;
    }

    public double Value { get; init; }
    public string Description { get; init; }

    public override string ToString() => $"{Value} ({Description})";
}

public static void Main()
{
    var m1 = new Measurement();
    Console.WriteLine(m1);  // output: NaN (Undefined)

    var m2 = default(Measurement);
    Console.WriteLine(m2);  // output: 0 ()

    var ms = new Measurement[2];
    Console.WriteLine(string.Join(", ", ms));  // output: 0 (), 0 ()
}
```

如前面的示例所示，默认值表达式忽略了无参数构造函数，并生成了结构类型的默认值。 结构类型数组实例化还忽略无参数构造函数并生成使用结构类型的默认值填充的数组。

你看到默认值的最常见情况是在数组中或内部存储包含变量块的其他集合中。 以下示例创建了一个由 30 个 `TemperatureRange` 结构组成的数组，每个结构都具有默认值：

C#

```
// All elements have default values of 0:
TemperatureRange[] lastMonth = new TemperatureRange[30];
```

结构的所有成员字段在创建时必须进行*明确指定*，因为 `struct` 类型直接存储其数据。 结构的 `default` 值已将所有字段*明确指定*为 0。 调用构造函数时，必须明确指定所有字段。 可以使用以下机制初始化字段：

- 可以将*字段初始化*表达式添加到任何字段或自动实现的属性。
- 可以在构造函数主体中初始化任何字段或自动属性。

从 C# 11 开始，如果你没有初始化结构中的所有字段，编译器会将代码添加到将这些字段初始化为默认值的构造函数中。 分配给其 `default` 值的结构将初始化为 0 位模式。 使用 `new` 初始化的字符串将初始化为 0 位模式，然后执行任何字段初始值设定项和构造函数。

C#

```
public readonly struct Measurement
{
    public Measurement(double value)
    {
        Value = value;
    }

    public Measurement(double value, string description)
    {
        Value = value;
        Description = description;
    }

    public Measurement(string description)
    {
        Description = description;
    }

    public double Value { get; init; }
    public string Description { get; init; } = "Ordinary measurement";

    public override string ToString() => $"{Value} ({Description})";
}

public static void Main()
{
    var m1 = new Measurement(5);
    Console.WriteLine(m1);  // output: 5 (Ordinary measurement)

    var m2 = new Measurement();
    Console.WriteLine(m2);  // output: 0 ()

    var m3 = default(Measurement);
    Console.WriteLine(m3);  // output: 0 ()
}
```

每个 `struct` 都具有一个 `public` 无参数构造函数。 如果要编写无参数构造函数，它必须是公共构造函数。 如果结构声明了任何字段初始值设定项，就必须显式声明一个构造函数。 该构造函数不必是无参数的。 如果结构声明了字段初始值设定项，但没有构造函数，编译器将报告错误。 任何显式声明的构造函数（有参数或无参数）都会执行该结构的所有字段初始值设定项。 没有字段初始值设定项或构造函数的赋值的所有字段均设置为默认值。 有关详细信息，请参阅无参数结构构造函数功能建议说明。

从 C# 12 开始，`struct` 类型可以将主构造函数定义为其声明的一部分。 主要构造函数为构造函数参数提供了简洁的语法，可在该结构的任何成员声明中的整个 `struct` 正文中使用。

如果结构类型的所有实例字段都是可访问的，则还可以在不使用 `new` 运算符的情况下对其进行实例化。 在这种情况下，在首次使用实例之前必须初始化所有实例字段。 下面的示例演示如何执行此操作：

C#

```
public static class StructWithoutNew
{
    public struct Coords
    {
        public double x;
        public double y;
    }

    public static void Main()
    {
        Coords p;
        p.x = 3;
        p.y = 4;
        Console.WriteLine($"({p.x}, {p.y})");  // output: (3, 4)
    }
}
```

对于内置值类型，请使用相应的文本来指定类型的值。

结构类型的设计限制

结构具有类类型的大部分功能。 存在一些例外情况：

- 结构类型不能从其他类或结构类型继承，也不能作为类的基础类型。 但是，结构类型可以实现接口。
- 不能在结构类型中声明终结器。
- 在 C# 11 之前，结构类型的构造函数必须初始化该类型的所有实例字段。

按引用传递结构类型变量

将结构类型变量作为参数传递给方法或从方法返回结构类型值时，将复制结构类型的整个实例。 通过值传递可能会影响高性能方案中涉及大型结构类型的代码的性能。 通过按引用传递结构类型变量，可以避免值复制操作。 使用 `ref`、`out`、`in` 或 `ref readonly` 方法参数修饰符，指示必须按引用传递某个参数。 使用 ref 返回值按引用返回方法结果。 有关详细信息，请参阅避免分配。

`struct` 约束

你还可在 `struct` 中使用 关键字，来指定类型参数为不可为 null 的值类型。 结构类型和枚举类型都满足 `struct` 约束。

转换

对于任何结构类型（ref struct 类型除外），都存在与 和 System.ValueType 类型之间的 System.Object相互转换。 还存在结构类型和它所实现的任何接口之间的装箱和取消装箱转换。

C# 语言规范

有关详细信息，请参阅 C# 语言规范中的结构部分。

有关 `struct` 功能的详细信息，请参阅以下功能建议说明：

- 只读结构
- 只读实例成员
- 无参数结构构造函数
- 允许对结构 使用 `with` 表达式
- 记录结构
- 自动默认结构

另请参阅

- C# 类型系统
- 设计指南 - 在类和结构之间选择
- 设计指南 - 结构设计
- 解决内联数组声明中的错误和警告

`ref` 结构类型（C# 参考）

项目 • 2024/08/01

你可以在结构类型的声明中使用 `ref` 修饰符。 `ref struct` 类型的实例是在堆栈上分配的，不能转义到托管堆。 为了确保这一点，编译器将 `ref struct` 类型的使用限制如下：

- `ref struct` 不能是数组的元素类型。
- `ref struct` 不能是类或非 `ref struct` 的字段的声明类型。
- `ref struct` 不能被装箱为 System.ValueType 或 System.Object。
- `ref struct` 变量不能在 Lambda 表达式或本地函数中捕获。
- 在 C# 13 之前，`ref struct` 变量无法在 `async` 方法中使用。 从 C# 13 开始，`ref struct` 变量不能在 async 方法中和 await 表达式用于相同的块。 但是，可以在同步方法中使用 `ref struct` 变量，例如，在返回 Task 或 Task<TResult> 的方法中。
- 在 C# 13 之前，`ref struct` 变量不能在迭代器中使用。 从 C# 13 开始，`ref struct` 类型和 `ref` 局部变量可以在迭代器中使用，前提是它们不在代码段中具有 `yield return` 语句。
- 在 C# 13 之前，`ref struct` 无法实现接口。 从 C# 13 开始，`ref` 结构可实现接口，但必须遵循 ref 安全性规则。 例如，由于需要装箱转换，因此无法将 `ref struct` 类型转换为接口类型。
- 在 C# 13 之前，`ref struct` 不能是类型参数。 从 C# 13 开始，`ref struct` 可以是类型参数（当该类型参数在其 `where` 子句中指定 `allows ref struct` 时）。

通常，如果需要一种同时包含 `ref struct` 类型的数据成员的类型，可以定义 `ref struct` 类型：

C#

```
public ref struct CustomRef
{
    public bool IsValid;
    public Span<int> Inputs;
    public Span<int> Outputs;
}
```

若要将 `ref struct` 声明为 `readonly`，请在类型声明中组合使用 `readonly` 修饰符和 `ref` 修饰符（`readonly` 修饰符必须位于 `ref` 修饰符之前）：

C#

```
public readonly ref struct ConversionRequest
{
    public ConversionRequest(double rate, ReadOnlySpan<double> values)
```

```
    {
        Rate = rate;
        Values = values;
    }

    public double Rate { get; }
    public ReadOnlySpan<double> Values { get; }
}
```

在 .NET 中，`ref struct` 的示例分别是 System.Span<T> 和 System.ReadOnlySpan<T>。

`ref` 字段

从 C# 11 开始，可以在 `ref struct` 中声明 `ref` 字段，如以下示例所示：

C#

```
public ref struct RefFieldExample
{
    private ref int number;

    public int GetNumber()
    {
        if (System.Runtime.CompilerServices.Unsafe.IsNullRef(ref number))
        {
            throw new InvalidOperationException("The number ref field is not
initialized.");
        }

        return number;
    }
}
```

`ref` 字段可以具有 `null` 值。 使用 Unsafe.IsNullRef<T>(T) 方法确定 `ref` 字段是否为 `null`。

可通过以下方式将 `readonly` 修饰符应用于 `ref` 字段：

- `readonly ref`：只能在构造函数或 init 访问器中使用 `= ref` 运算符通过 ref 重新赋值此类字段。 可以在字段访问修饰符允许的任何时间点使用 `=` 运算符分配值。
- `ref readonly`：在任何时候，都不能使用 `=` 运算符为此类字段赋值。 但是，可以使用 `= ref` 运算符通过 ref 重新赋值字段。
- `readonly ref readonly`：只能在构造函数或 `init` 访问器中通过 ref 重新赋值此类字段。 在任何时候，都不能为字段赋值。

编译器确保存储在 `ref` 字段中的引用的生存期不会超过其引用。

`ref` 字段功能支持安全实现类型，例如 System.Span<T>：

```C#
public readonly ref struct Span<T>
{
    internal readonly ref T _reference;
    private readonly int _length;

    // Omitted for brevity...
}
```

`Span<T>` 类型存储一个引用，通过该引用访问内存中的连续元素。 通过使用引用，`Span<T>` 实例可以避免复制它所引用的存储。

可释放模式

可以定义一次性的 `ref struct`。 为此，请确保 `ref struct` 符合一次性模式。 也就是说，它有一个实例 `Dispose` 方法，该方法是可访问、无参数的并且具有 `void` 返回类型。 可以将 using 语句或声明与可释放的 `ref struct` 的实例一起使用。

从 C# 13 开始，还可对 `ref struct` 类型上实现 IDisposable。 但是，重载解析更倾向于可释放模式，而不是接口方法。 只有在找不到合适的 `IDisposable.Dispose` 方法时，编译程序才会解析为 `Dispose` 方法。

实现接口的 `ref struct` 类型的限制

这些限制可确保实现接口的 `ref struct` 类型遵守必要的 ref 安全性规则。

- 无法将 `ref struct` 转换为它实现的接口的实例。 在参数为接口类型时，若你使用 `ref struct` 类型作为参数时，则此限制包括隐式转换。 该转换会导致装箱转换，这违反了 ref 安全性。
- 实现接口的 `ref struct` 必须实现所有接口成员。 `ref struct` 必须实现接口包含默认实现的成员。

编译程序强制实施这些限制。 如果编写实现接口的 `ref struct` 类型，则每个新更新都可能包含新的默认接口成员。 在为这些新方法提供实现之前，应用程序不会编译。

> **ⓘ 重要**
>
> 实现接口的 `ref struct` 包括以后进行源中断性和二进制中断性变更的可能性。 如果 `ref struct` 实现在另一个程序集中定义的接口，并且该程序集提供将默认成员添加

到该接口的更新，则会发生中断。

重新编译 `ref struct` 时发生源中断：即使存在默认实现，它也必须实现新成员。

如果升级外部程序集而不重新编译 `ref struct` 类型，并且更新的代码调用新方法的默认实现，则会发生二进制中断。 访问默认成员时，运行时引发异常。

C# 语言规范

有关更多信息，请参阅 C# 语言规范的以下部分：

- 结构：Ref 修饰符
- ref 结构类型的安全上下文约束

有关 `ref` 字段的详细信息，请参阅低级别结构改进建议说明。

另请参阅

- C# 类型系统

在 GitHub 上与我们协作

可以在 GitHub 上找到此内容的源，还可以在其中创建和查看问题和拉取请求。 有关详细信息，请参阅参与者指南。

.NET 反馈

.NET 是一个开放源代码项目。 选择一个链接以提供反馈：

提出文档问题

提供产品反馈

元组类型（C# 参考）

项目 • 2023/06/05

元组功能提供了简洁的语法来将多个数据元素分组成一个轻型数据结构。 下面的示例演示了如何声明元组变量、对它进行初始化并访问其数据成员：

C#

```
(double, int) t1 = (4.5, 3);
Console.WriteLine($"Tuple with elements {t1.Item1} and {t1.Item2}.");
// Output:
// Tuple with elements 4.5 and 3.

(double Sum, int Count) t2 = (4.5, 3);
Console.WriteLine($"Sum of {t2.Count} elements is {t2.Sum}.");
// Output:
// Sum of 3 elements is 4.5.
```

如前面的示例所示，若要定义元组类型，需要指定其所有数据成员的类型，或者，可以指定字段名称。 虽然不能在元组类型中定义方法，但可以使用 .NET 提供的方法，如下面的示例所示：

C#

```
(double, int) t = (4.5, 3);
Console.WriteLine(t.ToString());
Console.WriteLine($"Hash code of {t} is {t.GetHashCode()}.");
// Output:
// (4.5, 3)
// Hash code of (4.5, 3) is 718460086.
```

元组类型支持相等运算符`==`和`!=`。 有关详细信息，请参阅元组相等部分。

元组类型是值类型；元组元素是公共字段。 这使得元组为可变的值类型。

可以使用任意数量的元素定义元组：

C#

```
var t =
(1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
11, 12, 13, 14, 15, 16, 17, 18,
19, 20, 21, 22, 23, 24, 25, 26);
Console.WriteLine(t.Item26);  // output: 26
```

元组的用例

元组最常见的用例之一是作为方法返回类型。 也就是说，你可以将方法结果分组为元组返回类型，而不是定义 out 方法参数，如以下示例所示：

C#

```
var xs = new[] { 4, 7, 9 };
var limits = FindMinMax(xs);
Console.WriteLine($"Limits of [{string.Join(" ", xs)}] are {limits.min} and
{limits.max}");
// Output:
// Limits of [4 7 9] are 4 and 9

var ys = new[] { -9, 0, 67, 100 };
var (minimum, maximum) = FindMinMax(ys);
Console.WriteLine($"Limits of [{string.Join(" ", ys)}] are {minimum} and
{maximum}");
// Output:
// Limits of [-9 0 67 100] are -9 and 100

(int min, int max) FindMinMax(int[] input)
{
    if (input is null || input.Length == 0)
    {
        throw new ArgumentException("Cannot find minimum and maximum of a
null or empty array.");
    }

    // Initialize min to MaxValue so every value in the input
    // is less than this initial value.
    var min = int.MaxValue;
    // Initialize max to MinValue so every value in the input
    // is greater than this initial value.
    var max = int.MinValue;
    foreach (var i in input)
    {
        if (i < min)
        {
            min = i;
        }
        if (i > max)
        {
            max = i;
        }
    }
    return (min, max);
}
```

如前面的示例所示，可以直接使用返回的元组实例，或者可以在单独的变量中析构它。

还可以使用元组类型，而不是匿名类型；例如，在 LINQ 查询中。 有关详细信息，请参阅在匿名类型和元组类型之间选择。

通常，你会使用元组对相关的数据元素进行松散分组。 在公共 API 中，请考虑定义类或结构类型。

元组字段名称

可以在元组初始化表达式中或元组类型的定义中显式指定元组字段名称，如下面的示例所示：

C#

```
var t = (Sum: 4.5, Count: 3);
Console.WriteLine($"Sum of {t.Count} elements is {t.Sum}.");

(double Sum, int Count) d = (4.5, 3);
Console.WriteLine($"Sum of {d.Count} elements is {d.Sum}.");
```

如果未指定字段名称，则可以根据元组初始化表达式中相应变量的名称推断出此名称，如下面的示例所示：

C#

```
var sum = 4.5;
var count = 3;
var t = (sum, count);
Console.WriteLine($"Sum of {t.count} elements is {t.sum}.");
```

这称为元组投影初始值设定项。 在以下情况下，变量名称不会被投影到元组字段名称中：

- 候选名称是元组类型的成员名称，例如 `Item3`、`ToString` 或 `Rest`。
- 候选名称是另一元组的显式或隐式字段名称的重复项。

在前面的示例中，你可以显式指定字段的名称，或按字段的默认名称访问字段。

元组字段的默认名称为 `Item1`、`Item2`、`Item3` 等。 始终可以使用字段的默认名称，即使字段名称是显式指定的或推断出的，如下面的示例所示：

C#

```
var a = 1;
var t = (a, b: 2, 3);
Console.WriteLine($"The 1st element is {t.Item1} (same as {t.a}).");
Console.WriteLine($"The 2nd element is {t.Item2} (same as {t.b}).");
```

```
Console.WriteLine($"The 3rd element is {t.Item3}.");
// Output:
// The 1st element is 1 (same as 1).
// The 2nd element is 2 (same as 2).
// The 3rd element is 3.
```

元组赋值和元组相等比较不会考虑字段名称。

在编译时，编译器会将非默认字段名称替换为相应的默认名称。 因此，显式指定或推断的字段名称在运行时不可用。

> **提示**
>
> 启用 .NET 代码样式规则 **IDE0037** 以设置推理或显式元组字段名称的首选项。

从 C# 12 开始，可以使用 using 指令指定元组类型的别名。 以下示例为元组类型添加 `global using` 别名，其中包含表示允许的 `Min` 值和 `Max` 值的两个整数值：

C#

```
global using BandPass = (int Min, int Max);
```

声明别名后，可以将 `BandPass` 名称用作该元组类型的别名：

C#

```
BandPass bracket = (40, 100);
Console.WriteLine($"The bandpass filter is {bracket.Min} to {bracket.Max}");
```

别名不会引入新类型，而只会为现有类型创建同义词。 可以析构使用 `BandPass` 别名声明的元组，就像析构其基础元组类型一样：

C#

```
(int a , int b) = bracket;
Console.WriteLine($"The bracket is {a} to {b}");
```

与元组赋值或析构一样，元组成员名称不需要进行匹配；类型需要进行匹配。

同样，具有相同 arity 和成员类型的第二个别名可以与原始别名互换使用。 可以声明第二个别名：

C#

```
using Range = (int Minimum, int Maximum);
```

可以将 `Range` 元组分配给 `BandPass` 元组。 与所有元组赋值一样，字段名称不需要进行匹配，只有类型和 arity 需要进行匹配。

C#

```
Range r = bracket;
Console.WriteLine($"The range is {r.Minimum} to {r.Maximum}");
```

使用元组时，元组类型的别名提供更多语义信息。 它不会引入新类型。 若要提供类型安全性，应改为声明位置 record。

元组赋值和析构

C# 支持满足以下两个条件的元组类型之间的赋值：

- 两个元组类型有相同数量的元素
- 对于每个元组位置，右侧元组元素的类型与左侧相应的元组元素的类型相同或可以隐式转换为左侧相应的元组元素的类型

元组元素是按照元组元素的顺序赋值的。 元组字段的名称会被忽略且不会被赋值，如下面的示例所示：

C#

```
(int, double) t1 = (17, 3.14);
(double First, double Second) t2 = (0.0, 1.0);
t2 = t1;
Console.WriteLine($"{nameof(t2)}: {t2.First} and {t2.Second}");
// Output:
// t2: 17 and 3.14

(double A, double B) t3 = (2.0, 3.0);
t3 = t2;
Console.WriteLine($"{nameof(t3)}: {t3.A} and {t3.B}");
// Output:
// t3: 17 and 3.14
```

还可以使用赋值运算符 `=` 在单独的变量中析构元组实例。 可以通过许多方式来进行此操作：

- 在括号外使用 `var` 关键字来声明隐式类型化变量，并让编译器推断其类型：

```C#
var t = ("post office", 3.6);
var (destination, distance) = t;
Console.WriteLine($"Distance to {destination} is {distance}
kilometers.");
// Output:
// Distance to post office is 3.6 kilometers.
```

- 在括号内显式声明每个变量的类型：

```C#
var t = ("post office", 3.6);
(string destination, double distance) = t;
Console.WriteLine($"Distance to {destination} is {distance}
kilometers.");
// Output:
// Distance to post office is 3.6 kilometers.
```

- 在括号内显式声明一些类型，隐式声明其他类型（使用 `var`）：

```C#
var t = ("post office", 3.6);
(var destination, double distance) = t;
Console.WriteLine($"Distance to {destination} is {distance}
kilometers.");
// Output:
// Distance to post office is 3.6 kilometers.
```

- 使用现有变量：

```C#
var destination = string.Empty;
var distance = 0.0;

var t = ("post office", 3.6);
(destination, distance) = t;
Console.WriteLine($"Distance to {destination} is {distance}
kilometers.");
// Output:
// Distance to post office is 3.6 kilometers.
```

析构表达式的目标可以包括现有变量和析构声明中声明的变量。

还可以将析构与模式匹配相结合，以检查元组中字段的特征。 以下示例循环访问多个整数，并输出可被 3 整除的整数。 它析构 Int32.DivRem 的元组结果，并与 `Remainder` 0 进行匹配：

C#

```
for (int i = 4; i < 20;  i++)
{
    if (Math.DivRem(i, 3) is ( Quotient: var q, Remainder: 0 ))
    {
        Console.WriteLine($"{i} is divisible by 3, with quotient {q}");
    }
}
```

有关析构元组和其他类型的详细信息，请参阅析构元组和其他类型。

元组相等

元组类型支持 `==` 和 `!=` 运算符。 这些运算符按照元组元素的顺序将左侧操作数的成员与相应的右侧操作数的成员进行比较。

C#

```
(int a, byte b) left = (5, 10);
(long a, int b) right = (5, 10);
Console.WriteLine(left == right);  // output: True
Console.WriteLine(left != right);  // output: False

var t1 = (A: 5, B: 10);
var t2 = (B: 5, A: 10);
Console.WriteLine(t1 == t2);  // output: True
Console.WriteLine(t1 != t2);  // output: False
```

如前面的示例所示，`==` 和 `!=` 操作不会考虑元组字段名称。

同时满足以下两个条件时，两个元组可比较：

- 两个元组具有相同数量的元素。 例如，如果 `t1` 和 `t2` 具有不同数目的元素，`t1 != t2` 则不会进行编译。
- 对于每个元组位置，可以使用 `==` 和 `!=` 运算符对左右侧元组操作数中的相应元素进行比较。 例如，`(1, (2, 3)) == ((1, 2), 3)` 不会进行编译，因为 `1` 不可与 `(1, 2)` 比较。

`==` 和 `!=` 运算符将以短路方式对元组进行比较。 也就是说，一旦遇见一对不相等的元素或达到元组的末尾，操作将立即停止。 但是，在进行任何比较之前，将对所有元组元素

进行计算，如以下示例所示：

C#

```
Console.WriteLine((Display(1), Display(2)) == (Display(3), Display(4)));

int Display(int s)
{
    Console.WriteLine(s);
    return s;
}
// Output:
// 1
// 2
// 3
// 4
// False
```

元组作为 out 参数

通常，你会将具有 out 参数的方法重构为返回元组的方法。 但是，在某些情况下，`out` 参数可以是元组类型。 下面的示例演示了如何将元组作为 `out` 参数使用：

C#

```
var limitsLookup = new Dictionary<int, (int Min, int Max)>()
{
    [2] = (4, 10),
    [4] = (10, 20),
    [6] = (0, 23)
};

if (limitsLookup.TryGetValue(4, out (int Min, int Max) limits))
{
    Console.WriteLine($"Found limits: min is {limits.Min}, max is
{limits.Max}");
}
// Output:
// Found limits: min is 10, max is 20
```

元组与 `System.Tuple`

System.ValueTuple 类型支持的 C# 元组不同于 System.Tuple 类型表示的元组。 主要区别如下：

- `System.ValueTuple` 类型是值类型。 `System.Tuple` 类型是引用类型。

- `System.ValueTuple` 类型是可变的。 `System.Tuple` 类型是不可变的。
- `System.ValueTuple` 类型的数据成员是字段。 `System.Tuple` 类型的数据成员是属性。

C# 语言规范

有关详细信息，请参阅：

- 元组类型
- 元组相等运算符

另请参阅

- C# 参考
- 值类型
- 在匿名类型和元组类型之间选择
- System.ValueTuple

可为空的值类型（C# 参考）

项目 • 2023/04/07

可为 null 值类型 `T?` 表示其基础值类型 `T` 的所有值及额外的 null 值。 例如，可以将以下三个值中的任意一个指定给 `bool?` 变量：`true`、`false` 或 `null`。 基础值类型 `T` 本身不能是可为空的值类型。

任何可为空的值类型都是泛型 System.Nullable<T> 结构的实例。 可使用以下任何一种可互换形式引用具有基础类型 `T` 的可为空值类型：`Nullable<T>` 或 `T?`。

需要表示基础值类型的未定义值时，通常使用可为空的值类型。 例如，布尔值或 `bool` 变量只能为 `true` 或 `false`。 但是，在某些应用程序中，变量值可能未定义或缺失。 例如，某个数据库字段可能包含 `true` 或 `false`，或者它可能不包含任何值，即 `NULL`。 在这种情况下，可以使用 `bool?` 类型。

声明和赋值

由于值类型可隐式转换为相应的可为空的值类型，因此可以像向其基础值类型赋值一样，向可为空值类型的变量赋值。 还可分配 `null` 值。 例如：

C#

```
double? pi = 3.14;
char? letter = 'a';

int m2 = 10;
int? m = m2;

bool? flag = null;

// An array of a nullable value type:
int?[] arr = new int?[10];
```

可为空值类型的默认值表示 `null`，也就是说，它是其 Nullable<T>.HasValue 属性返回 `false` 的实例。

检查可为空值类型的实例

可以将 is 运算符与类型模式 结合使用，既检查 `null` 的可为空值类型的实例，又检索基础类型的值：

C#

```
int? a = 42;
if (a is int valueOfA)
{
    Console.WriteLine($"a is {valueOfA}");
}
else
{
    Console.WriteLine("a does not have a value");
}
// Output:
// a is 42
```

始终可以使用以下只读属性来检查和获取可为空值类型变量的值：

- Nullable<T>.HasValue 指示可为空值类型的实例是否有基础类型的值。
- 如果 HasValue 为 `true`，则 Nullable<T>.Value 获取基础类型的值。 如果 HasValue 为 `false`，则 Value 属性将引发 InvalidOperationException。

以下示例中的使用 `HasValue` 属性在显示值之前测试变量是否包含该值：

C#

```
int? b = 10;
if (b.HasValue)
{
    Console.WriteLine($"b is {b.Value}");
}
else
{
    Console.WriteLine("b does not have a value");
}
// Output:
// b is 10
```

还可将可为空的值类型的变量与 `null` 进行比较，而不是使用 `HasValue` 属性，如以下示例所示：

C#

```
int? c = 7;
if (c != null)
{
    Console.WriteLine($"c is {c.Value}");
}
else
{
    Console.WriteLine("c does not have a value");
}
```

```
// Output:
// c is 7
```

从可为空的值类型转换为基础类型

如果要将可为空值类型的值分配给不可以为 null 的值类型变量，则可能需要指定要分配的替代 `null` 的值。 使用 Null 合并操作符??执行此操作（也可将 Nullable<T>.GetValueOrDefault(T) 方法用于相同的目的）：

C#

```
int? a = 28;
int b = a ?? -1;
Console.WriteLine($"b is {b}");  // output: b is 28

int? c = null;
int d = c ?? -1;
Console.WriteLine($"d is {d}");  // output: d is -1
```

如果要使用基础值类型的默认值来替代 `null`，请使用 Nullable<T>.GetValueOrDefault() 方法。

还可以将可为空的值类型显式强制转换为不可为 null 的类型，如以下示例所示：

C#

```
int? n = null;

//int m1 = n;    // Doesn't compile
int n2 = (int)n; // Compiles, but throws an exception if n is null
```

在运行时，如果可为空的值类型的值为 `null`，则显式强制转换将抛出 InvalidOperationException。

不可为 null 的值类型 `T` 隐式转换为相应的可为空值类型 `T?`。

提升的运算符

预定义的一元运算符和二元运算符或值类型 `T` 支持的任何重载运算符也受相应的可为空值类型 `T?` 支持。 如果一个或全部两个操作数为 `null` ，则这些运算符（也称为提升的运算符）将生成 `null`；否则，运算符使用其操作数所包含的值来计算结果。 例如：

C#

```
int? a = 10;
int? b = null;
int? c = 10;

a++;        // a is 11
a = a * c;  // a is 110
a = a + b;  // a is null
```

> **ⓘ 备注**
>
> 对于 `bool?` 类型，预定义的 `&` 和 `|` 运算符不遵循此部分中描述的规则：即使其中一个操作数为 `null`，运算符计算结果也可以不为 NULL。 有关详细信息，请参阅**布尔逻辑运算符**一文的**可以为 null 的布尔逻辑运算符**部分。

对于比较运算符`<`、`>`、`<=` 和 `>=`，如果一个或全部两个操作数都为 `null`，则结果为 `false`；否则，将比较操作数的包含值。 请勿作出如下假定：由于某个特定的比较（例如 `<=`）返回 `false`，则相反的比较 (`>`) 返回 `true`。 以下示例显示 10

- 既不大于等于 `null`，
- 也不小于 `null`

C#

```
int? a = 10;
Console.WriteLine($"{a} >= null is {a >= null}");
Console.WriteLine($"{a} < null is {a < null}");
Console.WriteLine($"{a} == null is {a == null}");
// Output:
// 10 >= null is False
// 10 < null is False
// 10 == null is False

int? b = null;
int? c = null;
Console.WriteLine($"null >= null is {b >= c}");
Console.WriteLine($"null == null is {b == c}");
// Output:
// null >= null is False
// null == null is True
```

对于相等运算符`==`，如果两个操作数都为 `null`，则结果为 `true`；如果只有一个操作数为 `null`，则结果为 `false`；否则，将比较操作数的包含值。

对于不等运算符`!=`，如果两个操作数都为 `null`，则结果为 `false`；如果只有一个操作数为 `null`，则结果为 `true`；否则，将比较操作数的包含值。

如果在两个值类型之间存在用户定义的转换，则还可在相应的可为空值类型之间使用同一转换。

装箱和取消装箱

可为空值类型的实例 `T?` 已装箱，如下所示：

- 如果 HasValue 返回 `false`，则生成空引用。
- 如果 HasValue 返回 `true`，则基础值类型 `T` 的对应值将装箱，而不对 Nullable<T> 的实例进行装箱。

可将值类型 `T` 的已装箱值取消装箱到相应的可为空值类型 `T?`，如以下示例所示：

C#

```
int a = 41;
object aBoxed = a;
int? aNullable = (int?)aBoxed;
Console.WriteLine($"Value of aNullable: {aNullable}");

object aNullableBoxed = aNullable;
if (aNullableBoxed is int valueOfA)
{
    Console.WriteLine($"aNullableBoxed is boxed int: {valueOfA}");
}
// Output:
// Value of aNullable: 41
// aNullableBoxed is boxed int: 41
```

如何确定可为空的值类型

下面的示例演示了如何确定 System.Type 实例是否表示已构造的可为空值类型，即，具有指定类型参数 `T` 的 System.Nullable<T> 类型：

C#

```
Console.WriteLine($"int? is {(IsNullable(typeof(int?)) ? "nullable" : "non
nullable")} value type");
Console.WriteLine($"int is {(IsNullable(typeof(int)) ? "nullable" : "non-
nullable")} value type");

bool IsNullable(Type type) => Nullable.GetUnderlyingType(type) != null;

// Output:
// int? is nullable value type
// int is non-nullable value type
```

如示例所示，使用 typeof 运算符来创建 System.Type 实例。

如果要确定实例是否是可为空的值类型，请不要使用 Object.GetType 方法获取要通过前面的代码测试的 Type 实例。 如果对值类型可为空的实例调用 Object.GetType 方法，该实例将装箱到 Object。 由于对可为空的值类型的非 NULL 实例的装箱等同于对基础类型的值的装箱，因此 GetType 会返回表示可为空的值类型的基础类型的 Type 实例：

C#

```
int? a = 17;
Type typeOfA = a.GetType();
Console.WriteLine(typeOfA.FullName);
// Output:
// System.Int32
```

另外，请勿使用 is 运算符来确定实例是否是可为空的值类型。 如以下示例所示，无法使用 `is` 运算符区分可为空值类型实例的类型与其基础类型实例：

C#

```
int? a = 14;
if (a is int)
{
    Console.WriteLine("int? instance is compatible with int");
}

int b = 17;
if (b is int?)
{
    Console.WriteLine("int instance is compatible with int?");
}
// Output:
// int? instance is compatible with int
// int instance is compatible with int?
```

请改为使用第一个示例中的 Nullable.GetUnderlyingType 和 typeof 运算符，以检查实例是否具有可为空的值类型。

> ⓘ **备注**
>
> 此部分中所述的方法不适用于**可为空的引用类型**的情况。

C# 语言规范

有关更多信息，请参阅 C# 语言规范的以下部分：

- 可以为 null 的类型
- 提升的运算符
- 隐式可为空转换
- 显式可为空转换
- 提升的转换运算符

请参阅

- C# 参考
- “提升”的准确含义是什么?
- System.Nullable<T>
- System.Nullable
- Nullable.GetUnderlyingType
- 可为空引用类型

引用类型（C# 参考）

项目 • 2024/04/02

C# 中有两种类型：引用类型和值类型。 引用类型的变量存储对其数据（对象）的引用，而值类型的变量直接包含其数据。 对于引用类型，两种变量可引用同一对象；因此，对一个变量执行的操作会影响另一个变量所引用的对象。 对于值类型，每个变量都具有其自己的数据副本，对一个变量执行的操作不会影响另一个变量（`in`、`ref` 和 `out` 参数变量除外；请参阅 in、ref 和 out 参数修饰符）。

下列关键字用于声明引用类型：

- class
- interface
- delegate
- record

C# 也提供了下列内置引用类型：

- dynamic
- object
- string

另请参阅

- C# 关键字
- 指针类型
- 值类型

在 GitHub 上与我们协作

可以在 GitHub 上找到此内容的源，还可以在其中创建和查看问题和拉取请求。 有关详细信息，请参阅参与者指南。

.NET 反馈

.NET 是一个开放源代码项目。 选择一个链接以提供反馈：

提出文档问题

提供产品反馈

内置引用类型（C# 引用）

项目 • 2023/05/10

C# 有许多内置引用类型。 这些类型包含的关键字或运算符是 .NET 库中的类型的同义词。

对象类型

`object` 类型是 System.Object 在 .NET 中的别名。 在 C# 的统一类型系统中，所有类型（预定义类型、用户定义类型、引用类型和值类型）都是直接或间接从 System.Object 继承的。 可以将任何类型的值赋给 `object` 类型的变量。 可以使用文本 `null` 将任何 `object` 变量赋值给其默认值。 将值类型的变量转换为对象的过程称为装箱。 将 `object` 类型的变量转换为值类型的过程称为取消装箱。 有关详细信息，请参阅装箱和取消装箱。

字符串类型

`string` 类型表示零个或多个 Unicode 字符的序列。 `string` 是 System.String 在 .NET 中的别名。

尽管 `string` 为引用类型，但是定义相等运算符 == 和 != 是为了比较 `string` 对象（而不是引用）的值。 基于值的相等性使得对字符串相等性的测试更为直观。 例如：

C#

```
string a = "hello";
string b = "h";
// Append to contents of 'b'
b += "ello";
Console.WriteLine(a == b);
Console.WriteLine(object.ReferenceEquals(a, b));
```

前面的示例显示“True”，然后显示“False”，因为字符串的内容是相等的，但 `a` 和 `b` 并不指代同一字符串实例。

+ 运算符连接字符串：

C#

```
string a = "good " + "morning";
```

前面的代码会创建一个包含“good morning”的字符串对象。

字符串是不可变的，即：字符串对象在创建后，其内容不可更改。 例如，编写此代码时，编译器实际上会创建一个新的字符串对象来保存新的字符序列，且该新对象将赋给 `b`。 已为 `b` 分配的内存（当它包含字符串“h”时）可用于垃圾回收。

C#

```
string b = "h";
b += "ello";
```

`[]`运算符可用于只读访问字符串的个别字符。 有效索引于 `0` 开始，且必须小于字符串的长度：

C#

```
string str = "test";
char x = str[2];  // x = 's';
```

同样，`[]` 运算符也可用于循环访问字符串中的每个字符：

C#

```
string str = "test";

for (int i = 0; i < str.Length; i++)
{
  Console.Write(str[i] + " ");
}
// Output: t e s t
```

字符串文本

字符串字面量属于 `string` 类型且可以三种形式编写（原始、带引号和逐字）。

原始字符串字面量从 C# 11 开始可用。 字符串字面量可以包含任意文本，而无需转义序列。 字符串字面量可以包括空格和新行、嵌入引号以及其他特殊字符。 原始字符串字面量用至少三个双引号 (""") 括起来：

C#

```
"""
This is a multi-line
    string literal with the second line indented.
"""
```

甚至可以包含三个（或更多）双引号字符序列。 如果文本需要嵌入的引号序列，请根据需要使用更多引号开始和结束原始字符串字面量：

C#

```
"""""
This raw string literal has four """", count them: """" four!
embedded quote characters in a sequence. That's why it starts and ends
with five double quotes.

You could extend this example with as many embedded quotes as needed for
your text.
"""""
```

原始字符串字面量的起始和结束引号序列通常位于与嵌入文本不同的行上。 多行原始字符串字面量支持自带引号的字符串：

C#

```
var message = """
"This is a very important message."
""";
Console.WriteLine(message);
// output: "This is a very important message."
```

当起始引号和结束引号在不同的行上时，则最终内容中不包括起始引号之后和结束引号之前的换行符。 右引号序列指示字符串字面量的最左侧列。 可以缩进原始字符串字面量以匹配整体代码格式：

C#

```
var message = """
    "This is a very important message."
    """;
Console.WriteLine(message);
// output: "This is a very important message."
// The leftmost whitespace is not part of the raw string literal
```

保留结束引号序列右侧的列。 此行为将为 JSON、YAML 或 XML 等数据格式启用原始字符串，如以下示例所示：

C#

```
var json= """
    {
        "prop": 0
```

```
    }
    """;
```

如果任何文本行扩展到右引号序列的左侧，编译器将发出错误。 左引号和右引号序列可以位于同一行上，前提是字符串字面量既不能以引号字符开头，也不能以引号字符结尾：

C#

```
var shortText = """He said "hello!" this morning.""";
```

可以将原始字符串字面量与字符串内插相结合，以在输出字符串中包含引号字符和大括号。

带引号字符串括在双引号 (") 内。

C#

```
"good morning"  // a string literal
```

字符串文本可包含任何字符文本。 包括转义序列。 下面的示例使用转义序列 `\\` 表示反斜杠，使用 `\u0066` 表示字母 f，以及使用 `\n` 表示换行符。

C#

```
string a = "\\\u0066\n F";
Console.WriteLine(a);
// Output:
// \f
//  F
```

> **备注**
>
> 转义码 `\udddd`（其中 `dddd` 是一个四位数字）表示 Unicode 字符 U+`dddd`。 另外，还可识别八位 Unicode 转义码：`\Uddddddddd`。

逐字字符串文本以 `@` 开头，并且也括在双引号内。 例如： 。

C#

```
@"good morning"  // a string literal
```

逐字字符串的优点是不处理转义序列，这样就可轻松编写。 例如，以下文本与完全限定的 Windows 文件名匹配：

C#

```
@"c:\Docs\Source\a.txt"  // rather than "c:\\Docs\\Source\\a.txt"
```

若要在用 @ 引起来的字符串中包含双引号，双倍添加即可：

C#

```
@"""Ahoy!"" cried the captain." // "Ahoy!" cried the captain.
```

UTF-8 字符串字面量

.NET 中的字符串是使用 UTF-16 编码存储的。 UTF-8 是 Web 协议和其他重要库的标准。 从 C# 11 开始，可以将 `u8` 后缀添加到字符串字面量以指定 UTF-8 编码。 UTF-8 字面量存储为 `ReadOnlySpan<byte>` 对象。 UTF-8 字符串字面量的自然类型是 `ReadOnlySpan<byte>`。 使用 UTF-8 字符串字面量创建的声明比声明等效的 System.ReadOnlySpan<T> 更清晰，如以下代码所示：

C#

```
ReadOnlySpan<byte> AuthWithTrailingSpace = new byte[] { 0x41, 0x55, 0x54,
0x48, 0x20 };
ReadOnlySpan<byte> AuthStringLiteral = "AUTH "u8;
```

要将 UTF-8 字符串字面量存储为数组，需使用 ReadOnlySpan<T>.ToArray() 将包含字面量的字节复制到可变数组：

C#

```
byte[] AuthStringLiteral = "AUTH "u8.ToArray();
```

UTF-8 字符串字面量不是编译时常量；而是运行时常量。 因此，不能将其用作可选参数的默认值。 UTF-8 字符串字面量不能与字符串内插结合使用。 不能对同一字符串表达式使用 `$` 令牌和 `u8` 后缀。

委托类型

委托类型的声明与方法签名相似。 它有一个返回值和任意数目任意类型的参数：

C#

```
public delegate void MessageDelegate(string message);
public delegate int AnotherDelegate(MyType m, long num);
```

在 .NET 中，`System.Action` 和 `System.Func` 类型为许多常见委托提供泛型定义。 可能不需要定义新的自定义委托类型。 相反，可以创建提供的泛型类型的实例化。

`delegate` 是一种可用于封装命名方法或匿名方法的引用类型。 委托类似于 C++ 中的函数指针；但是，委托是类型安全和可靠的。 有关委托的应用，请参阅委托和泛型委托。 委托是事件的基础。 通过将委托与命名方法或匿名方法关联，可以实例化委托。

必须使用具有兼容返回类型和输入参数的方法或 lambda 表达式实例化委托。 有关方法签名中允许的差异程度的详细信息，请参阅委托中的变体。 为了与匿名方法一起使用，委托和与之关联的代码必须一起声明。

当运行时涉及的委托类型因变体转换而不同时，委托组合或删除将失败，并出现运行时异常。 以下示例演示了失败的情况：

C#

```
Action<string> stringAction = str => {};
Action<object> objectAction = obj => {};

// Valid due to implicit reference conversion of
// objectAction to Action<string>, but may fail
// at run time.
Action<string> combination = stringAction + objectAction;
```

可以通过创建新的委托对象来创建具有正确运行时类型的委托。 下面的示例演示如何将此解决方法应用于前面的示例。

C#

```
Action<string> stringAction = str => {};
Action<object> objectAction = obj => {};

// Creates a new delegate instance with a runtime type of Action<string>.
Action<string> wrappedObjectAction = new Action<string>(objectAction);

// The two Action<string> delegate instances can now be combined.
Action<string> combination = stringAction + wrappedObjectAction;
```

从 C# 9 开始，可以声明使用类似语法的函数指针。 函数指针使用 `calli` 指令，而不是实例化委托类型和调用虚拟 `Invoke` 方法。

动态类型

`dynamic` 类型表示变量的使用和对其成员的引用绕过编译时类型检查。 改为在运行时解析这些操作。 `dynamic` 类型简化了对 COM API（例如 Office Automation API）、动态 API（例如 IronPython 库）和 HTML 文档对象模型 (DOM) 的访问。

在大多数情况下，`dynamic` 类型与 `object` 类型的行为类似。 具体而言，任何非 Null 表达式都可以转换为 `dynamic` 类型。 `dynamic` 类型与 `object` 的不同之处在于，编译器不会对包含类型 `dynamic` 的表达式的操作进行解析或类型检查。 编译器将有关该操作信息打包在一起，之后这些信息会用于在运行时评估操作。 在此过程中，`dynamic` 类型的变量会编译为 `object` 类型的变量。 因此，`dynamic` 类型只在编译时存在，在运行时则不存在。

下面的示例将 `dynamic` 类型的变量与 `object` 类型的变量进行对比。 若要在编译时验证每个变量的类型，请将鼠标指针放在 `WriteLine` 语句中的 `dyn` 或 `obj` 上。 请将下面的代码复制到可以使用 IntelliSense 的编辑器中。 IntelliSense 对 `dyn` 显示“dynamic”，对 `obj` 显示“object”。

C#

```
class Program
{
    static void Main(string[] args)
    {
        dynamic dyn = 1;
        object obj = 1;

        // Rest the mouse pointer over dyn and obj to see their
        // types at compile time.
        System.Console.WriteLine(dyn.GetType());
        System.Console.WriteLine(obj.GetType());
    }
}
```

WriteLine 语句显示 `dyn` 和 `obj` 的运行时类型。 此时，两者的类型均为整数。 将生成以下输出：

控制台

```
System.Int32
System.Int32
```

若要查看编译时 `dyn` 与 `obj` 之间的区别，请在前面示例的声明和 `WriteLine` 语句之间添加下列两行。

C#

```
dyn = dyn + 3;
obj = obj + 3;
```

尝试在表达式 `obj + 3` 中添加整数和对象时，将报告编译器错误。 但是，对于 `dyn + 3`，不会报告任何错误。 在编译时不会检查包含 `dyn` 的表达式，原因是 `dyn` 的类型为 `dynamic`。

下面的示例在多个声明中使用 `dynamic`。 `Main` 方法也将编译时类型检查与运行时类型检查进行了对比。

C#

```
using System;

namespace DynamicExamples
{
    class Program
    {
        static void Main(string[] args)
        {
            ExampleClass ec = new ExampleClass();
            Console.WriteLine(ec.ExampleMethod(10));
            Console.WriteLine(ec.ExampleMethod("value"));

            // The following line causes a compiler error because
ExampleMethod
            // takes only one argument.
            //Console.WriteLine(ec.ExampleMethod(10, 4));

            dynamic dynamic_ec = new ExampleClass();
            Console.WriteLine(dynamic_ec.ExampleMethod(10));

            // Because dynamic_ec is dynamic, the following call to
ExampleMethod
            // with two arguments does not produce an error at compile time.
            // However, it does cause a run-time error.
            //Console.WriteLine(dynamic_ec.ExampleMethod(10, 4));
        }
    }

    class ExampleClass
    {
        static dynamic _field;
        dynamic Prop { get; set; }

        public dynamic ExampleMethod(dynamic d)
        {
            dynamic local = "Local variable";
            int two = 2;
```

```
            if (d is int)
            {
                return local;
            }
            else
            {
                return two;
            }
        }
    }
}
// Results:
// Local variable
// 2
// Local variable
```

C# 语言规范

有关更多信息，请参阅 C# 语言规范的以下部分：

- §8.2.3 对象类型
- §8.2.4 动态类型
- §8.2.5 字符串类型
- §8.2.8 委托类型
- C# 11 - 原始字符串字面量
- C# 11 - 原始字符串字面量

请参阅

- C# 参考
- C# 关键字
- 事件
- 使用类型 dynamic
- 有关使用字符串的最佳做法
- 基本字符串操作
- 新建字符串
- 类型测试和强制转换运算符
- 如何使用模式匹配以及 is 和 as 运算符安全地进行强制转换
- 演练：创建和使用动态对象
- System.Object
- System.String
- System.Dynamic.DynamicObject

记录（C# 参考）

项目 • 2023/11/22

可使用 `record` 修饰符定义一个引用类型，用来提供用于封装数据的内置功能。 `record class` 语法用作等价符号以标识引用类型，`record struct` 则用于定义一种具有类似功能的 值类型。

在记录上声明主构造函数时，编译器会为主构造函数参数生成公共属性。 记录的主构造函数参数称为位置参数。 编译器创建镜像主构造函数或位置参数的位置属性。 编译器不会在没有 `record` 修饰符的类型上合成主构造函数参数的属性。

以下两个示例演示 `record`（或 `record class`）引用类型：

C#

```
public record Person(string FirstName, string LastName);
```

C#

```
public record Person
{
    public required string FirstName { get; init; }
    public required string LastName { get; init; }
};
```

以下两个示例演示 `record struct` 值类型：

C#

```
public readonly record struct Point(double X, double Y, double Z);
```

C#

```
public record struct Point
{
    public double X { get; init; }
    public double Y { get; init; }
    public double Z { get; init; }
}
```

此外，还可以创建具有可变属性和字段的记录：

C#

```
public record Person
{
    public required string FirstName { get; set; }
    public required string LastName { get; set; }
};
```

记录结构也可以是可变的，包括位置记录结构和没有位置参数的记录结构：

C#

```
public record struct DataMeasurement(DateTime TakenAt, double Measurement);
```

C#

```
public record struct Point
{
    public double X { get; set; }
    public double Y { get; set; }
    public double Z { get; set; }
}
```

虽然记录可以是可变的，但它们主要用于支持不可变的数据模型。 记录类型提供以下功能：

- 用于创建具有不可变属性的引用类型的简明语法
- 内置行为对于以数据为中心的引用类型非常有用：
  - 值相等性
  - 非破坏性变化的简明语法
  - 用于显示的内置格式设置
- 支持继承层次结构

前面的示例展示了引用类型记录和值类型记录之间的一些区别：

- `record` 或 `record class` 声明引用类型。 `class` 关键字是可选项，但可以为读者提高清晰度。 `record struct` 声明值类型。
- 位置属性在 和 `record class` 中不可变。 它们在 中可变。

本文的其余部分将讨论 `record class` 和 `record struct` 类型。 每个部分都详细说明了这些差异。 你应该在 `record class`和 `record struct` 之间作出决定，该过程类似于在 `class` 和 `struct` 之间作出决定。 record 一词用于描述应用于所有记录类型的行为。 `record struct` 或 `record class` 用于描述仅适用于 struct 或 class 类型的行为。

属性定义的位置语法

在创建实例时，可以使用位置参数来声明记录的属性，并初始化属性值：

C#

```
public record Person(string FirstName, string LastName);

public static void Main()
{
    Person person = new("Nancy", "Davolio");
    Console.WriteLine(person);
    // output: Person { FirstName = Nancy, LastName = Davolio }
}
```

当你为属性定义使用位置语法时，编译器将创建以下内容：

- 记录声明中提供的每个位置参数的公共自动实现属性。
  - 对于 `record` 和 `readonly record struct` 类型，为 `record` 属性。
  - 对于 `record struct` 类型，为读写属性。
- 主构造函数，它的参数与记录声明上的位置参数匹配。
- 对于 record struct 类型，则是将每个字段设置为其默认值的无参数构造函数。
- 一个 `Deconstruct` 方法，对记录声明中提供的每个位置参数都有一个 `out` 参数。 此方法解构了使用位置语法定义的属性；它忽略了使用标准属性语法定义的属性。

你可能想要将属性添加到编译器从记录定义中创建的任何元素。 你可以向应用于位置记录属性的任何特性添加目标。 以下示例将向 System.Text.Json.Serialization.JsonPropertyNameAttribute 记录的每个属性应用 `Person`。 `property:` 目标指示该特性应用于编译器生成的属性。 其他值包括将该特性应用于字段的 `field:`，以及将该特性应用于参数的 `param:`。

C#

```
/// <summary>
/// Person record type
/// </summary>
/// <param name="FirstName">First Name</param>
/// <param name="LastName">Last Name</param>
/// <remarks>
/// The person type is a positional record containing the
/// properties for the first and last name. Those properties
/// map to the JSON elements "firstName" and "lastName" when
/// serialized or deserialized.
/// </remarks>
public record Person([property: JsonPropertyName("firstName")] string
FirstName,
    [property: JsonPropertyName("lastName")] string LastName);
```

上面的示例还演示了如何为记录创建 XML 文档注释。 你可以添加 `<param>` 标记，以添加主要构造函数参数的文档。

如果生成的自动生成的属性定义不是所需的属性，则可以定义自己的同名属性。 例如，你可能想要更改可访问性或可变性，或者为 `get` 或 `set` 访问器提供实现。 如果在源中声明属性，则必须从记录的位置参数初始化该属性。 如果属性是自动实现的属性，则必须初始化该属性。 如果在源中添加支持字段，则必须初始化支持字段。 生成的析构函数将使用属性定义。 例如，下面的示例声明了位置记录 `FirstName` 的 `LastName` 和 `public` 属性，但将 `Id` 位置参数限制为 `internal`。 可以对记录和 record struct 类型使用此语法。

C#

```
public record Person(string FirstName, string LastName, string Id)
{
    internal string Id { get; init; } = Id;
}

public static void Main()
{
    Person person = new("Nancy", "Davolio", "12345");
    Console.WriteLine(person.FirstName); //output: Nancy

}
```

记录类型不需要声明任何位置属性。 你可以在没有任何位置属性的情况下声明一个记录，也可以声明其他字段和属性，如以下示例中所示：

C#

```
public record Person(string FirstName, string LastName)
{
    public string[] PhoneNumbers { get; init; } = [];
};
```

如果使用标准属性语法来定义属性，但省略了访问修饰符，这些属性将隐式地 `private`。

不可变性

positional record 和 positional readonly record struct 声明 init-only 属性 。 positional record struct 声明 read-write 属性。 可以替代这些默认值中的任何一个，如上个部分中所示。

如果你需要一个以数据为中心的类型是线程安全的，或者需要使哈希表中的哈希代码保持不变，那么不可变性很有用。 但不可变性并不是适用于所有的数据场景。 例如，Entity Framework Core 就不支持通过不可变实体类型进行更新。

init-only 属性无论是通过位置参数（`record class` 和 `readonly record struct`）创建的，还是通过指定 `init` 访问器创建的，都具有浅的不可变性。 初始化后，将不能更改值型属性的值或引用型属性的引用。 不过，引用型属性引用的数据是可以更改的。 下面的示例展示了引用型不可变属性的内容（本例中是数组）是可变的：

```C#
public record Person(string FirstName, string LastName, string[]
PhoneNumbers);

public static void Main()
{
    Person person = new("Nancy", "Davolio", new string[1] { "555-1234" });
    Console.WriteLine(person.PhoneNumbers[0]); // output: 555-1234

    person.PhoneNumbers[0] = "555-6789";
    Console.WriteLine(person.PhoneNumbers[0]); // output: 555-6789
}
```

记录类型特有的功能是由编译器合成的方法实现的，这些方法都不会通过修改对象状态来影响不可变性。 除非另行指定，否则将为 `record`、`record struct` 和 `readonly record struct` 声明生成综合方法。

值相等性

如果不替代或替换相等性方法，则声明的类型将控制如何定义相等性：

- 对于 `class` 类型，如果两个对象引用内存中的同一对象，则这两个对象相等。
- 对于 `struct` 类型，如果两个对象是相同的类型并且存储相同的值，则这两个对象相等。
- 对于具有 `record` 修饰符（`record class`、`record struct` 和 `readonly record struct`）的类型，如果两个对象是相同的类型并且存储相同的值，则这两个对象相等。

`record struct` 的相等性定义与 `struct` 的相等性定义相同。 不同之处在于，对于 `struct`，实现处于 ValueType.Equals(Object) 中并且依赖反射。 对于记录，实现由编译器合成，并且使用声明的数据成员。

一些数据模型需要引用相等性。 例如，Entity Framework Core 依赖于引用相等性来确保它对概念上是一个实体的实体类型只使用一个实例。 因此，记录和记录结构不适合用作 Entity Framework Core 中的实体类型。

下面的示例说明了记录类型的值相等性：

C#

```
public record Person(string FirstName, string LastName, string[]
PhoneNumbers);

public static void Main()
{
    var phoneNumbers = new string[2];
    Person person1 = new("Nancy", "Davolio", phoneNumbers);
    Person person2 = new("Nancy", "Davolio", phoneNumbers);
    Console.WriteLine(person1 == person2); // output: True

    person1.PhoneNumbers[0] = "555-1234";
    Console.WriteLine(person1 == person2); // output: True

    Console.WriteLine(ReferenceEquals(person1, person2)); // output: False
}
```

为了实现值相等性，编译器合成了几种方法，包括：

- Object.Equals(Object) 的替代。 如果显式声明此替代，则会出现错误。

  当两个参数都为非 NULL 时，此方法被用作 Object.Equals(Object, Object) 静态方法的基础。

- `virtual` 或 `sealed` 的 `Equals(R? other)`，其中 `R` 是记录类型。 此方法实现 IEquatable<T>。 可以显式声明此方法。

- 如果记录类型派生自基础记录类型 `Base`，则为 `Equals(Base? other)`。 如果显式声明此替代，则会出现错误。 如果提供自己的 `Equals(R? other)` 实现，则还应提供 `GetHashCode` 的实现。

- Object.GetHashCode() 的替代。 可以显式声明此方法。

- 运算符 `==` 和 `!=` 的替代。 如果显式声明这些运算符，则会出现错误。

- 如果记录类型派生自基础记录类型，则为 `protected override Type EqualityContract { get; };`。 可以显式声明此属性。 有关详细信息，请参阅继承层次结构中的相等性。

如果记录类型的方法与允许显式声明的合成方法的签名匹配，则编译器不会合成该方法。

非破坏性变化

如果需要复制包含一些修改的实例，可以使用 `with` 表达式来实现非破坏性变化。 `with` 表达式创建一个新的记录实例，该实例是现有记录实例的一个副本，修改了指定属性和字

段。 使用对象初始值设定项语法来指定要更改的值，如以下示例中所示：

C#

```
public record Person(string FirstName, string LastName)
{
    public string[] PhoneNumbers { get; init; }
}

public static void Main()
{
    Person person1 = new("Nancy", "Davolio") { PhoneNumbers = new string[1]
};
    Console.WriteLine(person1);
    // output: Person { FirstName = Nancy, LastName = Davolio, PhoneNumbers
= System.String[] }

    Person person2 = person1 with { FirstName = "John" };
    Console.WriteLine(person2);
    // output: Person { FirstName = John, LastName = Davolio, PhoneNumbers =
System.String[] }
    Console.WriteLine(person1 == person2); // output: False

    person2 = person1 with { PhoneNumbers = new string[1] };
    Console.WriteLine(person2);
    // output: Person { FirstName = Nancy, LastName = Davolio, PhoneNumbers
= System.String[] }
    Console.WriteLine(person1 == person2); // output: False

    person2 = person1 with { };
    Console.WriteLine(person1 == person2); // output: True
}
```

`with` 表达式可以设置位置属性或使用标准属性语法创建的属性。 显式声明属性必须有一个 `init` 或 `set` 访问器才能在 `with` 表达式中进行更改。

`with` 表达式的结果是一个浅的副本，这意味着对于引用属性，只复制对实例的引用。 原始记录和副本最终都具有对同一实例的引用。

为了对 `record class` 类型实现此功能，编译器合成了一个克隆方法和一个复制构造函数。 虚拟克隆方法返回由复制构造函数初始化的新记录。 当使用 `with` 表达式时，编译器将创建调用克隆方法的代码，然后设置 `with` 表达式中指定的属性。

如果你需要不同的复制行为，可以在 `record class` 中编写自己的复制构造函数。 如果这样做，编译器不会合成方法。 如果记录是 `private`，则使构造函数为 `sealed`，否则使其为 `protected`。 编译器不会针对 `record struct` 类型合成复制构造函数。 虽然你可以编写一个方法，但编译器不会为 `with` 表达式生成对它的调用。 `record struct` 的值在分配时进行复制。

你不能替代克隆方法，也不能在任意记录类型中创建名为 `Clone` 的成员。 克隆方法的实际名称是由编译器生成的。

用于显示的内置格式设置

记录类型具有编译器生成的 ToString 方法，可显示公共属性和字段的名称和值。 `ToString` 方法返回一个格式如下的字符串：

> <record type name> { <property name> = <value>, <property name> = <value>, ...}

为 `<value>` 打印的字符串是 ToString() 针对属性类型返回的字符串。 在以下示例中，`ChildNames` 是 System.Array，其中 `ToString` 返回 `System.String[]`：

```
Person { FirstName = Nancy, LastName = Davolio, ChildNames = System.String[]
}
```

为了实现此功能，编译器在 `record class` 类型中合成了一个虚拟 `PrintMembers` 方法和一个 ToString 替代。 此成员在 `record struct` 类型中为 `private`。 `ToString` 替代创建了一个 StringBuilder 对象，它的类型名称后跟一个左括号。 它调用 `PrintMembers` 以添加属性名称和值，然后添加右括号。 下面的示例展示了类似于合成替代中包含的代码：

C#

```
public override string ToString()
{
    StringBuilder stringBuilder = new StringBuilder();
    stringBuilder.Append("Teacher"); // type name
    stringBuilder.Append(" { ");
    if (PrintMembers(stringBuilder))
    {
        stringBuilder.Append(" ");
    }
    stringBuilder.Append("}");
    return stringBuilder.ToString();
}
```

你可以提供自己的 `PrintMembers` 实现或 `ToString` 替代。 本文后面的派生记录中的 PrintMembers 格式设置一节中提供了示例。 您对 `ToString` 的实现可能包括 `sealed` 修饰符，以防止编译器为任何派生记录生成 `ToString` 实现。 可以在 `record` 类型的层次结

构中创建一致的字符串表示形式。 （派生记录仍会为所有派生属性生成一个 `PrintMembers` 方法。）

继承

本部分仅适用于 `record class` 类型。

一条记录可以从另一条记录继承。 但是，记录不能从类继承，类也不能从记录继承。

派生记录类型中的位置参数

派生记录为基本记录主构造函数中的所有参数声明位置参数。 基本记录声明并初始化这些属性。 派生记录不会隐藏它们，而只会创建和初始化未在其基本记录中声明的参数的属性。

下面的示例说明了具有位置属性语法的继承：

C#

```
public abstract record Person(string FirstName, string LastName);
public record Teacher(string FirstName, string LastName, int Grade)
    : Person(FirstName, LastName);
public static void Main()
{
    Person teacher = new Teacher("Nancy", "Davolio", 3);
    Console.WriteLine(teacher);
    // output: Teacher { FirstName = Nancy, LastName = Davolio, Grade = 3 }
}
```

继承层次结构中的相等性

本部分适用于 `record class` 类型，但不适用于 `record struct` 类型。 要使两个记录变量相等，运行时类型必须相等。 包含变量的类型可能不同。 下面的代码示例演示了继承的相等性比较：

C#

```
public abstract record Person(string FirstName, string LastName);
public record Teacher(string FirstName, string LastName, int Grade)
    : Person(FirstName, LastName);
public record Student(string FirstName, string LastName, int Grade)
    : Person(FirstName, LastName);
public static void Main()
{
    Person teacher = new Teacher("Nancy", "Davolio", 3);
```

```
    Person student = new Student("Nancy", "Davolio", 3);
    Console.WriteLine(teacher == student); // output: False

    Student student2 = new Student("Nancy", "Davolio", 3);
    Console.WriteLine(student2 == student); // output: True
}
```

在此示例中，所有变量都声明为 `Person`，即使该实例是 `Student` 或 `Teacher` 的派生类型也是如此。 这些实例都具有相同的属性和相同的属性值。 但 `student == teacher` 返回 `False`（尽管这两个都是 `Person` 型变量），`student == student2` 返回 `True`（尽管一个是 `Person` 变量，而另一个是 `Student` 变量）。 相等性测试依赖于实际对象的运行时类型，而不是声明的变量类型。

为实现此行为，编译器合成了一个 `EqualityContract` 属性，该属性返回一个与记录类型匹配的 Type 对象。 利用 `EqualityContract` 相等性方法在检查相等性时，就可以比较对象的运行时类型。 如果记录的基类型为 `object`，则此属性为 `virtual`。 如果基类型是其他记录类型，则此属性是一个替代。 如果记录类型为 `sealed`，则此属性实际上是 `sealed`，因为类型是 `sealed`。

在代码比较派生类型的两个实例时，合成的相等性方法会检查基类型和派生类型的所有数据成员是否相等。 合成的 `GetHashCode` 方法从基类型和派生的记录类型中声明的所有数据成员中使用 `GetHashCode` 方法。 包含所有声明字段的数据成员 `record`，以及任何自动实现的属性的编译器合成后盾字段。

派生记录中的 `with` 表达式

`with` 表达式结果的运行时间类型与表达式操作数相同： 运行时类型的所有属性都会被复制，但你只能设置编译时类型的属性，如下面的示例所示：

C#

```
public record Point(int X, int Y)
{
    public int Zbase { get; set; }
};
public record NamedPoint(string Name, int X, int Y) : Point(X, Y)
{
    public int Zderived { get; set; }
};

public static void Main()
{
    Point p1 = new NamedPoint("A", 1, 2) { Zbase = 3, Zderived = 4 };

    Point p2 = p1 with { X = 5, Y = 6, Zbase = 7 }; // Can't set Name or
Zderived
```

```
    Console.WriteLine(p2 is NamedPoint);  // output: True
    Console.WriteLine(p2);
    // output: NamedPoint { X = 5, Y = 6, Zbase = 7, Name = A, Zderived = 4
}

    Point p3 = (NamedPoint)p1 with { Name = "B", X = 5, Y = 6, Zbase = 7,
Zderived = 8 };
    Console.WriteLine(p3);
    // output: NamedPoint { X = 5, Y = 6, Zbase = 7, Name = B, Zderived = 8
}
}
```

派生记录中的 `PrintMembers` 格式设置

派生记录类型的合成 `PrintMembers` 方法调用基实现。 结果是派生类型和基类型的所有公共属性和字段都包含在 `ToString` 输出中，如以下示例中所示：

C#

```
public abstract record Person(string FirstName, string LastName);
public record Teacher(string FirstName, string LastName, int Grade)
    : Person(FirstName, LastName);
public record Student(string FirstName, string LastName, int Grade)
    : Person(FirstName, LastName);

public static void Main()
{
    Person teacher = new Teacher("Nancy", "Davolio", 3);
    Console.WriteLine(teacher);
    // output: Teacher { FirstName = Nancy, LastName = Davolio, Grade = 3 }
}
```

你可以提供自己的 `PrintMembers` 方法的实现。 如果这样做，请使用以下签名：

- 对于派生自 `sealed` 的 `object` 记录（不声明基本记录）：使用 `private bool PrintMembers(StringBuilder builder)`；
- 对于从另一个记录派生的 `sealed` 记录（注意封闭类型是 `sealed`，因此该方法实际上是 `sealed`）：`protected override bool PrintMembers(StringBuilder builder)`；
- 对于非 `sealed` 且派生自对象的记录：使用 `protected virtual bool PrintMembers(StringBuilder builder);`
- 对于非 `sealed` 且派生自其他记录的记录：使用 `protected override bool PrintMembers(StringBuilder builder);`

下面的代码示例替换了合成的 `PrintMembers` 方法，一个是派生自对象的记录类型，另一个是派生自另一条记录的记录类型：

C#

```
public abstract record Person(string FirstName, string LastName, string[]
PhoneNumbers)
{
    protected virtual bool PrintMembers(StringBuilder stringBuilder)
    {
        stringBuilder.Append($"FirstName = {FirstName}, LastName =
{LastName}, ");
        stringBuilder.Append($"PhoneNumber1 = {PhoneNumbers[0]},
PhoneNumber2 = {PhoneNumbers[1]}");
        return true;
    }
}

public record Teacher(string FirstName, string LastName, string[]
PhoneNumbers, int Grade)
    : Person(FirstName, LastName, PhoneNumbers)
{
    protected override bool PrintMembers(StringBuilder stringBuilder)
    {
        if (base.PrintMembers(stringBuilder))
        {
            stringBuilder.Append(", ");
        };
        stringBuilder.Append($"Grade = {Grade}");
        return true;
    }
};

public static void Main()
{
    Person teacher = new Teacher("Nancy", "Davolio", new string[2] { "555-
1234", "555-6789" }, 3);
    Console.WriteLine(teacher);
    // output: Teacher { FirstName = Nancy, LastName = Davolio, PhoneNumber1
= 555-1234, PhoneNumber2 = 555-6789, Grade = 3 }
}
```

ⓘ 备注

即使基记录密封了 `ToString` 方法，编译器也会在派生记录中合成 `PrintMembers`。 还可以创建自己的 `PrintMembers` 实现。

派生记录中的解构函数行为

派生记录的 `Deconstruct` 方法返回编译时类型所有位置属性的值。 如果变量类型为基本记录，则只解构基本记录属性，除非该对象强制转换为派生类型。 下面的示例演示了如

何对派生记录调用解构函数。

```C#
public abstract record Person(string FirstName, string LastName);
public record Teacher(string FirstName, string LastName, int Grade)
    : Person(FirstName, LastName);
public record Student(string FirstName, string LastName, int Grade)
    : Person(FirstName, LastName);

public static void Main()
{
    Person teacher = new Teacher("Nancy", "Davolio", 3);
    var (firstName, lastName) = teacher; // Doesn't deconstruct Grade
    Console.WriteLine($"{firstName}, {lastName}");// output: Nancy, Davolio

    var (fName, lName, grade) = (Teacher)teacher;
    Console.WriteLine($"{fName}, {lName}, {grade}");// output: Nancy,
Davolio, 3
}
```

泛型约束

`record` 关键字是 `class` 或 `struct` 类型的修饰符。 添加 `record` 修饰符包括本文前面所述的行为。 没有任何泛型约束要求类型是记录。 `record class` 满足 `class` 约束。 `record struct` 满足 `struct` 约束。 有关详细信息，请参阅类型参数的约束。

C# 语言规范

有关详细信息，请参阅 C# 语言规范中的类部分。

有关这些功能的详细信息，请参阅以下功能建议说明：

- 记录
- init-only 资源库
- 协变返回结果

另请参阅

- 设计指南 - 在类和结构之间选择
- 设计指南 - 结构设计
- C# 类型系统
- with表达式

class（C# 参考）

项目 • 2024/03/13

使用 `class` 关键字声明类，如下例中所示：

```C#
class TestClass
{
    // Methods, properties, fields, events, delegates
    // and nested classes go here.
}
```

备注

在 C# 中仅允许单一继承。 也就是说，一个类仅能从一个基类继承实现。 但是，一个类可实现多个接口。 下表显示类继承和接口实现的一些示例：

展开表

继承	示例
无	`class ClassA { }`
Single	`class DerivedClass : BaseClass { }`
无，实现两个接口	`class ImplClass : IFace1, IFace2 { }`
单一，实现一个接口	`class ImplDerivedClass : BaseClass, IFace1 { }`

直接在命名空间中声明的、未嵌套在其它类中的类，可以是公共或内部。 默认情况下类为 `internal`。

类成员（包括嵌套的类）可以是 public、protected internal、protected、internal、private 或 private protected。 默认情况下成员为 `private`。

有关详细信息，请参阅访问修饰符。

可以声明具有类型参数的泛型类。 有关更多信息，请参见泛型类。

一个类可包含下列成员的声明：

- 构造函数
- 常量

- Fields
- 终结器
- 方法
- 属性
- 索引器
- 运算符
- 事件
- 委托
- 类
- 接口
- 结构类型
- 枚举类型

示例

下面的示例说明如何声明类字段、构造函数和方法。 该示例还说明如何实例化对象及如何打印实例数据。 本例声明了两个类。 第一个类 `Child` 包含两个私有字段（`name` 和 `age`）、两个公共构造函数和一个公共方法。 第二个类 `StringTest` 用于包含 `Main`。

C#

```
class Child
{
    private int age;
    private string name;

    // Default constructor:
    public Child()
    {
        name = "N/A";
    }

    // Constructor:
    public Child(string name, int age)
    {
        this.name = name;
        this.age = age;
    }
```

```
    // Printing method:
    public void PrintChild()
    {
        Console.WriteLine("{0}, {1} years old.", name, age);
    }
}

class StringTest
{
    static void Main()
    {
        // Create objects by using the new operator:
        Child child1 = new Child("Craig", 11);
        Child child2 = new Child("Sally", 10);

        // Create an object using the default constructor:
        Child child3 = new Child();

        // Display results:
        Console.Write("Child #1: ");
        child1.PrintChild();
        Console.Write("Child #2: ");
        child2.PrintChild();
        Console.Write("Child #3: ");
        child3.PrintChild();
    }
}
/* Output:
    Child #1: Craig, 11 years old.
    Child #2: Sally, 10 years old.
    Child #3: N/A, 0 years old.
*/
```

注释

注意：在上例中，私有字段（`name` 和 `age`）只能通过 `Child` 类的公共方法访问。 例如，不能在 `Main` 方法中使用如下语句打印 Child 的名称：

C#

```
Console.Write(child1.name);   // Error
```

只有当 `Main` 是类的成员时，才能从 `Main` 访问 `Child` 的私有成员。

类中不具有访问修饰符的已声明类型默认为 `private`，因此如果已删除关键字，则此示例中的数据成员将仍为 `private`。

最后要注意的是，默认情况下，对于使用无参数构造函数（`child3`）创建的对象，`age` 字段初始化为零。

C# 语言规范

有关详细信息，请参阅 C# 语言规范。 该语言规范是 C# 语法和用法的权威资料。

另请参阅

- C# 参考
- C# 关键字
- 引用类型

在 GitHub 上与我们协作

可以在 GitHub 上找到此内容的源，还可以在其中创建和查看问题和拉取请求。 有关详细信息，请参阅参与者指南。

.NET

.NET 反馈

.NET 是一个开放源代码项目。 选择一个链接以提供反馈：

提出文档问题

提供产品反馈

interface（C#参考）

项目 • 2024/07/31

接口定义协定。 实现该协定的任何 class、record 或 struct 必须提供接口中定义的成员的实现。 接口可为成员定义默认实现。 它还可以定义 static 成员，以便提供常见功能的单个实现。 从 C# 11 开始，接口可以定义 `static abstract` 或 `static virtual` 成员来声明实现类型必须提供声明的成员。 通常，`static virtual` 方法声明实现必须定义一组重载运算符。

在以下示例中，类 `ImplementationClass` 必须实现一个不含参数但返回 `void` 的名为 `SampleMethod` 的方法。

有关详细信息和示例，请参阅接口。

顶级接口（在命名空间中声明但未嵌套在另一种类型的接口中）可以声明为 `public` 或 `internal`。 默认为 `internal`。 嵌套接口声明（在另一类型内部声明的接口）可以使用任何访问修饰符声明。

没有实现的接口成员不能包含访问修饰符。 具有默认实现的成员可以包含任何访问修饰符。

接口示例

C#

```
interface ISampleInterface
{
    void SampleMethod();
}

class ImplementationClass : ISampleInterface
{
    // Explicit interface member implementation:
    void ISampleInterface.SampleMethod()
    {
        // Method implementation.
    }

    static void Main()
    {
        // Declare an interface instance.
        ISampleInterface obj = new ImplementationClass();

        // Call the member.
        obj.SampleMethod();
```

```
    }
}
```

接口可以是命名空间或类的成员。 接口声明可以包含以下成员的声明（没有任何实现的签名）：

- 方法
- 属性
- 索引器
- 事件

默认接口成员

上述成员声明通常不包含主体。 接口成员可以声明主体。 接口中的成员主体是默认实现。 具有主体的成员允许接口为不提供重写实现的类和结构提供“默认”实现。

> ⓘ **重要**
>
> 添加默认接口成员会强制实现接口的任何 `ref struct` 添加该成员的显式声明。

接口可能包括：

- 常量
- 运算符
- 静态构造函数。
- 嵌套类型
- 静态字段、方法、属性、索引和事件
- 使用显式接口实现语法的成员声明。
- 显式访问修饰符（默认访问权限为 public）。

静态抽象成员和虚拟成员

从 C# 11 开始，接口可以声明除字段之外的所有成员类型的 `static abstract` 和 `static virtual` 成员。 接口可以声明实现类型必须定义运算符或其他静态成员。 此功能使泛型算法能够指定类似于数字的行为。 可以在 .NET 运行时的数值类型中看到示例，如 System.Numerics.INumber<TSelf>。 这些接口定义由许多数值类型实现的常见数学运算符。 编译器必须在编译时解析对 `static virtual` 和 `static abstract` 方法的调用。 接口中声明的 `static virtual` 和 `static abstract` 方法没有类似于类中声明的 `virtual` 或 `abstract` 方法的运行时调度机制。 相反，编译器使用编译时可用的类型信息。 因此，`static virtual` 方法几乎完全是在泛型接口中声明的。 此外，声明 `static virtual` 或

`static abstract` 方法的大多数接口都声明了其中一个类型参数必须实现已声明的接口。 例如，`INumber<T>` 接口声明 `T` 必须实现 `INumber<T>`。 编译器使用类型参数解析对接口声明中声明的方法和运算符的调用。 例如，`int` 类型实现 `INumber<int>`。 当类型参数 `T` 表示类型参数 `int` 时，将调用 `int` 上声明的 `static` 成员。 或者，当 `double` 是类型参数时，将调用在 `double` 类型上声明的 `static` 成员。

> **重要**
>
> 使用表达式的编译时类型解析接口中声明的 `static abstract` 和 `static virtual` 方法的方法调度。 如果表达式的运行时类型派生自不同的编译时类型，将调用基（编译时）类型的静态方法。

可以通过使用有关接口中静态抽象成员的教程来尝试此功能。

接口继承

接口不能包含实例状态。 虽然现在允许使用静态字段，但接口中不允许使用实例字段。 接口中不支持实例自动属性，因为它们将隐式声明隐藏的字段。 此规则对属性声明有细微影响。 在接口声明中，以下代码不会像在 `class` 或 `struct` 中一样声明自动实现的属性。 相反，它会声明一个属性，该属性没有默认实现，而必须在该实现接口的任何类型中实现它：

C#

```
public interface INamed
{
  public string Name {get; set;}
}
```

一个接口可从一个或多个基接口继承。 当接口继承自另一个接口时，实现派生接口的类型必须实现基接口中的所有成员以及派生接口中声明的成员，如以下代码所示：

C#

```
public interface I1
{
    void M1();
}

public interface I2 : I1
{
    void M2();
}
```

```
public class C : I2
{
    // implements I1.M1
    public void M1() { }
    // implements I2.M2
    public void M2() { }
}
```

当接口 重写基接口中的方法实现时，必须使用显式接口实现语法。

基类型列表包含基类和接口时，基类必须是列表中的第 1 项。

实现接口的类可以显式实现该接口的成员。 显式实现的成员不能通过类实例访问，而只能通过接口实例访问。 此外，只能通过接口实例访问默认接口成员。

有关显式接口实现的详细信息，请参阅显式接口实现。

接口实现示例

下例演示了接口实现。 在此示例中，接口包含属性声明，类包含实现。 实现 `IPoint` 的类的任何实例都具有整数属性 `x` 和 `y`。

C#

```
interface IPoint
{
    // Property signatures:
    int X { get; set; }

    int Y { get; set; }

    double Distance { get; }
}

class Point : IPoint
{
    // Constructor:
    public Point(int x, int y)
    {
        X = x;
        Y = y;
    }

    // Property implementation:
    public int X { get; set; }

    public int Y { get; set; }

    // Property implementation
    public double Distance =>
```

```
        Math.Sqrt(X * X + Y * Y);
}

class MainClass
{
    static void PrintPoint(IPoint p)
    {
        Console.WriteLine("x={0}, y={1}", p.X, p.Y);
    }

    static void Main()
    {
        IPoint p = new Point(2, 3);
        Console.Write("My Point: ");
        PrintPoint(p);
    }
}
// Output: My Point: x=2, y=3
```

C# 语言规范

有关详细信息，请参阅 C# 语言规范的接口部分、C# 8 - 默认接口成员的功能规范和 C# 11 - 接口中的静态抽象成员的功能规范

另请参阅

- C# 关键字
- 引用类型
- 接口
- 使用属性
- 使用索引器

在 GitHub 上与我们协作

可以在 GitHub 上找到此内容的源，还可以在其中创建和查看问题和拉取请求。 有关详细信息，请参阅参与者指南。

.NET

.NET 反馈

.NET 是一个开放源代码项目。 选择一个链接以提供反馈：

提出文档问题

提供产品反馈

可为 null 的引用类型（C# 引用）

项目 • 2024/12/30

> **ⓘ 备注**
>
> 本文介绍可为 null 的引用类型。 还可以声明**可为 null 的值类型**。

可以为 Null 的引用类型在*可为 null 的感知上下文*中的代码中可用。 可为 null 的引用类型、null 静态分析警告和 null 包容运算符是可选的语言功能。 默认情况下，所有功能都将关闭。 可为 null 的上下文在项目级使用生成设置进行控制，或在代码中使用 pragmas 进行控制 。

> **ⓘ 重要**
>
> 所有项目模板都为项目启用*可为 null 的上下文*。 使用早期模板创建的项目不包含此元素，并且除非在项目文件中启用这些功能或使用 pragma，否则这些功能将关闭。

在可为 null 的感知上下文中：

- 引用类型 `T` 的变量必须用非 null 值进行初始化，并且不能为其分配可能为 `null` 的值。
- 引用类型 `T?` 的变量可以用 `null` 进行初始化，也可以分配 `null`，但在取消引用之前必须对照 `null` 进行检查。
- 类型为 `m` 的变量 `T?` 在应用 null 包容运算符时被认为是非空的，如 `m!` 中所示。

编译器使用上述规则强制实施非空引用类型 `T` 和可空引用类型 `T?` 之间的区别。 `T` 类型的变量和 `T?` 类型的变量是相同的 .NET 类型。 下面的示例声明了一个不可为 null 的字符串和一个可为 null 的字符串，然后使用 null 包容运算符将一个值分配给不可为 null 的字符串：

C#

```
string notNull = "Hello";
string? nullable = default;
notNull = nullable!; // null forgiveness
```

变量 `notNull` 和 `nullable` 都由 String 类型表示。 因为不可为 null 的类型和可为 null 的类型都存储为相同的类型，所以有几个位置不允许使用可为 null 的引用类型。 通常，可为 null 的引用类型不能用作基类或实现的接口。 可为 null 的引用类型不能用于任何对象

创建或类型测试表达式。 可为 null 的引用类型不能是成员访问表达式的类型。 下面的示例说明了这些构造：

C#

```
public MyClass : System.Object? // not allowed
{
}

var nullEmpty = System.String?.Empty; // Not allowed
var maybeObject = new object?(); // Not allowed
try
{
    if (thing is string? nullableString) // not allowed
        Console.WriteLine(nullableString);
} catch (Exception? e) // Not Allowed
{
    Console.WriteLine("error");
}
```

可为 null 的引用和静态分析

上一部分中的示例说明了可为 null 的引用类型的性质。 可为 null 的引用类型不是新的类类型，而是对现有引用类型的注释。 编译器使用这些注释来帮助你查找代码中潜在的 null 引用错误。 不可为 null 的引用类型和可为 null 的引用类型在运行时没有区别。 编译器不会为不可为 null 的引用类型添加任何运行时检查。 这有利于编译时分析。 编译器将生成警告，帮助你查找和修复代码中潜在的 null 错误。 你需声明意向，如果代码违反该意向，编译器会发出警告。

在可为 null 的上下文中，编译器对任何引用类型的变量（可为 null 的和不可为 null 的）执行静态分析。 编译器会跟踪每个引用变量的 null-state，即 not-null 或 maybe-null 。 不可为 Null 的引用的默认状态为 not-null。 可为 Null 的引用的默认状态为 maybe-null。

不可为 Null 的引用类型在取消引用时应该始终是安全的，因为它们的 null-state 是 not-null 。 若强制执行该规则，如果不可为 null 的引用类型没有初始化为非 null 值，编译器将发出警告。 必须在声明的位置分配局部变量。 必须在字段初始化表达式或每个构造函数中为每个字段分配 not-null 值。 如果将不可为 Null 的引用分配给状态为 maybe-null 的引用，编译器会发出警告。 一般情况下，不可为 Null 的引用的状态为 not-null，在取消引用这些变量时不会发出警告。

① 备注

如果将 maybe-null 表达式分配给不可为 Null 的引用类型，编译器会生成警告。 然后，编译器会针对该变量生成警告，直到将该变量分配给 not-null 表达式。

可为 null 的引用类型可以进行初始化或分配给 `null`。 因此，静态分析必须在取消对变量的引用之前确定该变量的状态为 not-null。 如果可为 Null 的引用的状态确定为 maybe-null，将其分配给不可为 Null 的引用变量会生成编译器警告。 以下类显示了这些警告的示例：

C#

```
public class ProductDescription
{
    private string shortDescription;
    private string? detailedDescription;

    public ProductDescription() // Warning! shortDescription not
initialized.
    {
    }

    public ProductDescription(string productDescription) =>
        this.shortDescription = productDescription;

    public void SetDescriptions(string productDescription, string?
details=null)
    {
        shortDescription = productDescription;
        detailedDescription = details;
    }

    public string GetDescription()
    {
        if (detailedDescription.Length == 0) // Warning! dereference
possible null
        {
            return shortDescription;
        }
        else
        {
            return $"{shortDescription}\n{detailedDescription}";
        }
    }

    public string FullDescription()
    {
        if (detailedDescription == null)
        {
            return shortDescription;
        }
        else if (detailedDescription.Length > 0) // OK, detailedDescription
can't be null.
```

```
        {
            return $"{shortDescription}\n{detailedDescription}";
        }
        return shortDescription;
    }
}
```

以下代码段显示了编译器在使用此类时发出警告的位置：

C#

```
string shortDescription = default; // Warning! non-nullable set to null;
var product = new ProductDescription(shortDescription); // Warning! static
analysis knows shortDescription maybe null.

string description = "widget";
var item = new ProductDescription(description);

item.SetDescriptions(description, "These widgets will do everything.");
```

前面的示例演示编译器的静态分析如何确定引用变量的 null-state。 编译器对 null 检查和分配应用语言规则以通知其分析。 编译器无法对方法或属性的语义进行假设。 如果调用执行 null 检查的方法，则编译器无法得知这些方法会影响变量的 null-state。 可以将属性添加到 API，以告知编译器有关参数和返回值的语义。 .NET 库中的许多常见 API 都具有这些属性。 例如，编译器正确地将 IsNullOrEmpty 解释为空值检查。 有关应用于 null-state 静态分析的属性的详细信息，请参阅有关*可为 Null 的属性*的文章。

设置可为 null 的上下文

可以通过两种方式控制可为 null 的上下文。 在项目级别，可以添加 `<Nullable>enable</Nullable>` 项目设置。 在单个 C# 源文件中，可以添加 `#nullable enable` 来启用可为 null 的上下文。 请参阅关于设置可为 null 策略的文章。 在 .NET 6 之前，新项目使用默认值 `<Nullable>disable</Nullable>`。 从 .NET 6 开始，新项目将在项目文件中包含 `<Nullable>enable</Nullable>` 元素。

C# 语言规范

有关详细信息，请参阅 C# 语言规范的以下建议：

- 可为空引用类型
- 可为空引用类型规范草案

另请参阅

- 可以为 null 的值类型

集合

项目 • 2023/09/04

.NET 运行时提供了许多集合类型，用于存储和管理相关对象的组。 一些集合类型（例如 System.Array、System.Span<T> 和 System.Memory<T>）可使用 C# 语言识别。 此外，类似 System.Collections.Generic.IEnumerable<T> 的接口可使用枚举集合元素的语言来识别。

集合提供灵活的方式来使用对象组。 可按以下特征对不同的集合进行分类：

- **元素访问**：可以枚举每个集合以按顺序访问每个元素。 某些集合可通过*索引*（元素在有序集合中的位置）访问元素。 最常见的示例是 System.Collections.Generic.List<T>。 其他集合可按*键*访问元素，其中*值*与单个*键*相关联。 最常见的示例是 System.Collections.Generic.Dictionary<TKey,TValue>。 可根据应用访问元素的方式在这些集合类型之间进行选择。
- **性能配置文件**：每个集合都有不同的性能配置文件，可用于添加元素、查找元素或移除元素等操作。 可以根据应用中最常用的操作选取集合类型。
- **动态增长和收缩**：大多数集合支持动态添加或移除元素。 需要注意的是，Array、System.Span<T> 和 System.Memory<T> 不支持。

除了这些特征之外，运行时还提供专用集合，这些集合可阻止添加或移除元素，或修改集合的元素。 其他专用集合为多线程应用中的并发访问提供安全性。

可以在 .NET API 参考中找到所有集合类型。 有关其他信息，请参阅常用集合类型和选择集合类。

> ⓘ **备注**
>
> 对于本文中的示例，可能需要为 `System.Collections.Generic` 和 `System.Linq` 命名空间添加 **using directives**。

数组由 System.Array 表示，并受 C# 语言语法支持。 此语法为数组变量提供了更简洁的声明。

System.Span<T> 属于 ref struct 类型，它可提供一系列元素的快照，而无需复制这些元素。 编译器强制实施安全规则，以确保在它引用的序列不再存在于作用域内之后无法访问 `Span`。 它用于许多 .NET API 以提高性能。 Memory<T> 可在无法使用 `ref struct` 类型时提供类似的行为。

从 C# 12 开始，可以使用集合表达式初始化所有集合类型。

可索引集合

可索引集合 是一个可以使用其索引访问每个元素的集合。 其*索引*是序列中在它之前的元素数。 因此，按索引 `0` 引用的元素是第一个元素，索引 `1` 则是第二个元素，依此而行。 这些示例使用 List<T> 类。 它是最常见的可索引集合。

以下示例会创建和初始化字符串列表、移除元素并将元素添加到列表末尾。 每次修改后，它会使用 foreach 语句或 `for` 循环来循环访问字符串：

C#

```
// Create a list of strings by using a
// collection initializer.
var salmons = new List<string> { "chinook", "coho", "pink", "sockeye" };

// Iterate through the list.
foreach (var salmon in salmons)
{
    Console.Write(salmon + " ");
}
// Output: chinook coho pink sockeye

// Remove an element from the list by specifying
// the object.
salmons.Remove("coho");


// Iterate using the index:
for (var index = 0; index < salmons.Count; index++)
{
    Console.Write(salmons[index] + " ");
}
// Output: chinook pink sockeye

// Add the removed element
salmons.Add("coho");
// Iterate through the list.
foreach (var salmon in salmons)
{
    Console.Write(salmon + " ");
}
// Output: chinook pink sockeye coho
```

以下示例会从一个泛型列表中按索引移除元素。 它使用以降序进行循环访问的 `for` 语句，而不是 `foreach` 语句。 RemoveAt 方法将导致已移除元素后的元素索引值减小。

C#

```
var numbers = new List<int> { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
```

```
// Remove odd numbers.
for (var index = numbers.Count - 1; index >= 0; index--)
{
    if (numbers[index] % 2 == 1)
    {
        // Remove the element by specifying
        // the zero-based index in the list.
        numbers.RemoveAt(index);
    }
}

// Iterate through the list.
// A lambda expression is placed in the ForEach method
// of the List(T) object.
numbers.ForEach(
    number => Console.Write(number + " "));
// Output: 0 2 4 6 8
```

对于 List<T> 中的元素类型，还可以定义自己的类。 在下面的示例中，由 List<T> 使用的 `Galaxy` 类在代码中定义。

C#

```
private static void IterateThroughList()
{
    var theGalaxies = new List<Galaxy>
    {
        new (){ Name="Tadpole", MegaLightYears=400},
        new (){ Name="Pinwheel", MegaLightYears=25},
        new (){ Name="Milky Way", MegaLightYears=0},
        new (){ Name="Andromeda", MegaLightYears=3}
    };

    foreach (Galaxy theGalaxy in theGalaxies)
    {
        Console.WriteLine(theGalaxy.Name + "  " + theGalaxy.MegaLightYears);
    }

    // Output:
    //  Tadpole  400
    //  Pinwheel  25
    //  Milky Way  0
    //  Andromeda  3
}

public class Galaxy
{
    public string Name { get; set; }
    public int MegaLightYears { get; set; }
}
```

键/值对集合

这些示例使用 Dictionary<TKey,TValue> 类。 这是最常见的字典集合。 使用字典集合，可通过使用每个元素的键访问集合中的元素。 每次对字典的添加都包含一个值和与其关联的键。

以下示例创建 `Dictionary` 集合并通过使用 `foreach` 语句循环访问字典。

C#

```
private static void IterateThruDictionary()
{
    Dictionary<string, Element> elements = BuildDictionary();

    foreach (KeyValuePair<string, Element> kvp in elements)
    {
        Element theElement = kvp.Value;

        Console.WriteLine("key: " + kvp.Key);
        Console.WriteLine("values: " + theElement.Symbol + " " +
            theElement.Name + " " + theElement.AtomicNumber);
    }
}

public class Element
{
    public required string Symbol { get; init; }
    public required string Name { get; init; }
    public required int AtomicNumber { get; init; }
}

private static Dictionary<string, Element> BuildDictionary() =>
    new ()
    {
        {"K",
            new (){ Symbol="K", Name="Potassium", AtomicNumber=19}},
        {"Ca",
            new (){ Symbol="Ca", Name="Calcium", AtomicNumber=20}},
        {"Sc",
            new (){ Symbol="Sc", Name="Scandium", AtomicNumber=21}},
        {"Ti",
            new (){ Symbol="Ti", Name="Titanium", AtomicNumber=22}}
    };
```

以下示例使用 ContainsKey 方法和 `Dictionary` 的 Item[] 属性按键快速查找某个项。 使用 `Item` 属性可通过 C# 中的 `elements[symbol]` 来访问 `elements` 集合中的项。

C#

```
if (elements.ContainsKey(symbol) == false)
{
    Console.WriteLine(symbol + " not found");
}
else
{
    Element theElement = elements[symbol];
    Console.WriteLine("found: " + theElement.Name);
}
```

与之相反，以下示例使用 TryGetValue 方法按键快速查找某个项。

C#

```
if (elements.TryGetValue(symbol, out Element? theElement) == false)
    Console.WriteLine(symbol + " not found");
else
    Console.WriteLine("found: " + theElement.Name);
```

迭代器

迭代器用于对集合执行自定义迭代。 迭代器可以是一种方法，或是一个 `get` 访问器。 迭代器使用 yield return 语句返回集合的每一个元素，每次返回一个元素。

通过使用 foreach 语句调用迭代器。 `foreach` 循环的每次迭代都会调用迭代器。 迭代器中到达 `yield return` 语句时，会返回一个表达式，并保留当前在代码中的位置。 下次调用迭代器时，将从该位置重新开始执行。

有关详细信息，请参阅迭代器 (C#)。

下面的示例使用迭代器方法。 迭代器方法具有位于 `for` 循环中的 `yield return` 语句。 在 `ListEvenNumbers` 方法中，`foreach` 语句体的每次迭代都会创建对迭代器方法的调用，并将继续到下一个 `yield return` 语句。

C#

```
private static void ListEvenNumbers()
{
    foreach (int number in EvenSequence(5, 18))
    {
        Console.Write(number.ToString() + " ");
    }
    Console.WriteLine();
    // Output: 6 8 10 12 14 16 18
}
```

```
private static IEnumerable<int> EvenSequence(
    int firstNumber, int lastNumber)
{
    // Yield even numbers in the range.
    for (var number = firstNumber; number <= lastNumber; number++)
    {
        if (number % 2 == 0)
        {
            yield return number;
        }
    }
}
```

LINQ 和集合

可以使用语言集成查询 (LINQ) 来访问集合。 LINQ 查询提供筛选、排序和分组功能。 有关详细信息，请参阅 C# 中的 LINQ 入门。

以下示例运行一个对泛型 `List` 的 LINQ 查询。 LINQ 查询返回一个包含结果的不同集合。

C#

```
private static void ShowLINQ()
{
    List<Element> elements = BuildList();

    // LINQ Query.
    var subset = from theElement in elements
                 where theElement.AtomicNumber < 22
                 orderby theElement.Name
                 select theElement;

    foreach (Element theElement in subset)
    {
        Console.WriteLine(theElement.Name + " " + theElement.AtomicNumber);
    }

    // Output:
    //  Calcium 20
    //  Potassium 19
    //  Scandium 21
}

private static List<Element> BuildList() => new()
    {
        { new(){ Symbol="K", Name="Potassium", AtomicNumber=19}},
        { new(){ Symbol="Ca", Name="Calcium", AtomicNumber=20}},
        { new(){ Symbol="Sc", Name="Scandium", AtomicNumber=21}},
```

```
        { new(){ Symbol="Ti", Name="Titanium", AtomicNumber=22}}
    };
```

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET feedback

The .NET documentation is open source. Provide feedback here.

Open a documentation issue

Provide product feedback

数组

项目 • 2025/01/08

可以将同一类型的多个变量存储在一个数组数据结构中。 通过指定数组的元素类型来声明数组。 如果希望数组存储任意类型的元素，可将其类型指定为 `object`。 在 C# 的统一类型系统中，所有类型（预定义类型、用户定义类型、引用类型和值类型）都是直接或间接从 Object 继承的。

C#

```
type[] arrayName;
```

数组是引用类型，因此数组可以是可为 null 的引用类型。 元素类型可能是引用类型，因此可以声明数组来保存可为 null 的引用类型。 以下示例声明显示了用于声明数组或元素可为 null 性的不同语法：

C#

```
type?[] arrayName; // non nullable array of nullable element types.
type[]? arrayName; // nullable array of non-nullable element types.
type?[]? arrayName; // nullable array of nullable element types.
```

数组中未初始化的元素会被设置为该类型的默认值。

C#

```
int[] numbers = new int[10]; // All values are 0
string[] messages = new string[10]; // All values are null.
```

> **ⓘ 重要**
>
> 在前面的示例中，即使类型为 `string[]`，这是一种不可为 null 的字符串数组，但每个元素的默认值仍然是 null。 将数组初始化为非 null 值的最佳方法是使用 **集合表达式**。

数组具有以下属性：

- 数组可以是一维、多维或交错的。
- 声明数组变量时设置维度数。 创建数组实例时，将建立每个维度的长度。 这些值在实例的生存期内无法更改。
- 交错数组是数组数组，每个成员数组的默认值为 `null`。

- 数组从零开始编制索引：包含 `n` 元素的数组从 `0` 索引到 `n-1`。
- 数组元素可以是任何类型，其中包括数组类型。
- 数组类型是从抽象的基类型 派生的Array。 所有数组都会实现IList和IEnumerable。 可以使用 foreach 语句循环访问数组。 单维数组还实现了 IList<T> 和 IEnumerable<T>。

创建数组时，可以将数组的元素初始化为已知值。 从 C# 12 开始，可以使用集合表达式初始化所有集合类型。 未初始化的元素设置为默认值。 默认值为 0 位模式。 所有引用类型（包括不可为 null的类型）都具有值`null`。 所有值类型都有 0 位模式。 这意味着Nullable<T>.HasValue属性为`false`且Nullable<T>.Value属性未定义。 在 .NET 实现中，`Value`属性引发异常。

下面的示例创建一维数组、多维数组和交错数组：

C#

```
// Declare a single-dimensional array of 5 integers.
int[] array1 = new int[5];

// Declare and set array element values.
int[] array2 = [1, 2, 3, 4, 5, 6];

// Declare a two dimensional array.
int[,] multiDimensionalArray1 = new int[2, 3];

// Declare and set array element values.
int[,] multiDimensionalArray2 = { { 1, 2, 3 }, { 4, 5, 6 } };

// Declare a jagged array.
int[][] jaggedArray = new int[6][];

// Set the values of the first array in the jagged array structure.
jaggedArray[0] = [1, 2, 3, 4];
```

ⓘ 重要

本文中的许多示例使用**集合表达式**（使用方括号）来初始化数组。 集合表达式最初在 C# 12 中引入，该版本随 .NET 8 发布。 如果尚无法升级到 C# 12，请改用 `{` 和 `}` 初始化数组。

C#

```
// Collection expressions:
int[] array = [1, 2, 3, 4, 5, 6];
```

```
// Alternative syntax:
int[] array2 = {1, 2, 3, 4, 5, 6};
```

一维数组

*一维数组*是一系列相似元素。 可以通过其*索引*访问元素。 *索引*是序列中的顺序位置。 数组中的第一个元素位于索引`0`。 使用 new 运算符创建一维数组，该运算符指定数组元素类型和元素数目。 以下示例声明并初始化一维数组：

C#

```
int[] array = new int[5];
string[] weekDays = ["Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat"];

Console.WriteLine(weekDays[0]);
Console.WriteLine(weekDays[1]);
Console.WriteLine(weekDays[2]);
Console.WriteLine(weekDays[3]);
Console.WriteLine(weekDays[4]);
Console.WriteLine(weekDays[5]);
Console.WriteLine(weekDays[6]);

/*Output:
Sun
Mon
Tue
Wed
Thu
Fri
Sat
*/
```

第一个声明声明五个整数的未初始化数组，从`array[0]`到`array[4]`。 数组元素将初始化为元素类型的默认值，`0`代表整数。 第二个声明声明字符串数组并初始化该数组的所有七个值。 一系列`Console.WriteLine`语句打印`weekDay`数组的所有元素。 对于一维数组，`foreach`语句以递增索引顺序处理元素（从索引 0 开始并以索引`Length - 1`结束）。

将一维数组作为参数传递

可将初始化的一维数组传递给方法。 在下面的示例中，先初始化一个字符串数组，然后将其作为参数传递给字符串的`DisplayArray`方法。 该方法将显示数组的元素。 接下来，`ChangeArray`方法会反转数组元素，然后由`ChangeArrayElements`方法修改该数组的前三个元素。 每个方法返回后，`DisplayArray`方法会显示按值传递数组不会阻止对数组元素的更改。

C#

```
class ArrayExample
{
    static void DisplayArray(string[] arr) =>
Console.WriteLine(string.Join(" ", arr));

    // Change the array by reversing its elements.
    static void ChangeArray(string[] arr) => Array.Reverse(arr);

    static void ChangeArrayElements(string[] arr)
    {
        // Change the value of the first three array elements.
        arr[0] = "Mon";
        arr[1] = "Wed";
        arr[2] = "Fri";
    }

    static void Main()
    {
        // Declare and initialize an array.
        string[] weekDays = ["Sun", "Mon", "Tue", "Wed", "Thu", "Fri",
"Sat"];
        // Display the array elements.
        DisplayArray(weekDays);
        Console.WriteLine();

        // Reverse the array.
        ChangeArray(weekDays);
        // Display the array again to verify that it stays reversed.
        Console.WriteLine("Array weekDays after the call to ChangeArray:");
        DisplayArray(weekDays);
        Console.WriteLine();

        // Assign new values to individual array elements.
        ChangeArrayElements(weekDays);
        // Display the array again to verify that it has changed.
        Console.WriteLine("Array weekDays after the call to
ChangeArrayElements:");
        DisplayArray(weekDays);
    }
}
// The example displays the following output:
//         Sun Mon Tue Wed Thu Fri Sat
//
//       Array weekDays after the call to ChangeArray:
//       Sat Fri Thu Wed Tue Mon Sun
//
//       Array weekDays after the call to ChangeArrayElements:
//       Mon Wed Fri Wed Tue Mon Sun
```

多维数组

数组可具有多个维度。 例如，以下声明创建四个数组。 两个数组具有两个维度。 两个数组具有三个维度。 前两个声明声明每个维度的长度，但不初始化数组的值。 第二个声明使用初始值设定项设置多维数组中每个元素的值。

C#

```
int[,] array2DDeclaration = new int[4, 2];

int[,,] array3DDeclaration = new int[4, 2, 3];

// Two-dimensional array.
int[,] array2DInitialization =  { { 1, 2 }, { 3, 4 }, { 5, 6 }, { 7, 8 } };
// Three-dimensional array.
int[,,] array3D = new int[,,] { { { 1, 2, 3 }, { 4,   5,  6 } },
                                { { 7, 8, 9 }, { 10, 11, 12 } } };

// Accessing array elements.
System.Console.WriteLine(array2DInitialization[0, 0]);
System.Console.WriteLine(array2DInitialization[0, 1]);
System.Console.WriteLine(array2DInitialization[1, 0]);
System.Console.WriteLine(array2DInitialization[1, 1]);

System.Console.WriteLine(array2DInitialization[3, 0]);
System.Console.WriteLine(array2DInitialization[3, 1]);
// Output:
// 1
// 2
// 3
// 4
// 7
// 8

System.Console.WriteLine(array3D[1, 0, 1]);
System.Console.WriteLine(array3D[1, 1, 2]);
// Output:
// 8
// 12

// Getting the total count of elements or the length of a given dimension.
var allLength = array3D.Length;
var total = 1;
for (int i = 0; i < array3D.Rank; i++)
{
    total *= array3D.GetLength(i);
}
System.Console.WriteLine($"{allLength} equals {total}");
// Output:
// 12 equals 12
```

对于多维数组，遍历元素的方式为：首先递增最右边维度的索引，然后是它左边的一个维度，以此类推，向最左的索引遍历元素。 以下示例枚举 2D 和 3D 数组：

C#

```
int[,] numbers2D = { { 9, 99 }, { 3, 33 }, { 5, 55 } };

foreach (int i in numbers2D)
{
    System.Console.Write($"{i} ");
}
// Output: 9 99 3 33 5 55

int[,,] array3D = new int[,,] { { { 1, 2, 3 }, { 4,   5,  6 } },
                          { { 7, 8, 9 }, { 10, 11, 12 } } };
foreach (int i in array3D)
{
    System.Console.Write($"{i} ");
}
// Output: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12
```

在 2D 数组中，可以将左索引视为*行*，将右索引视为*列*。

但对于多维数组，使用嵌套的for循环可以更好地控制处理数组元素的顺序：

C#

```
int[,,] array3D = new int[,,] { { { 1, 2, 3 }, { 4,   5,  6 } },
                          { { 7, 8, 9 }, { 10, 11, 12 } } };

for (int i = 0; i < array3D.GetLength(0); i++)
{
    for (int j = 0; j < array3D.GetLength(1); j++)
    {
        for (int k = 0; k < array3D.GetLength(2); k++)
        {
            System.Console.Write($"{array3D[i, j, k]} ");
        }
        System.Console.WriteLine();
    }
    System.Console.WriteLine();
}
// Output (including blank lines):
// 1 2 3
// 4 5 6
//
// 7 8 9
// 10 11 12
//
```

将多维数组作为参数传递

通过与传递一维数组相同的方式，向方法传递初始化的多维数组。 下列代码演示了 Print 方法的部分声明（该方法接受将二维数组作为其参数）。 可在同一步骤中初始化并传递新数组，如下例所示。 在下列示例中，初始化一个整数的二维数组，并将其传递至 `Print2DArray` 方法。 该方法将显示数组的元素。

C#

```
static void Print2DArray(int[,] arr)
{
    // Display the array elements.
    for (int i = 0; i < arr.GetLength(0); i++)
    {
        for (int j = 0; j < arr.GetLength(1); j++)
        {
            System.Console.WriteLine("Element({0},{1})={2}", i, j, arr[i,
j]);
        }
    }
}
static void ExampleUsage()
{
    // Pass the array as an argument.
    Print2DArray(new int[,] { { 1, 2 }, { 3, 4 }, { 5, 6 }, { 7, 8 } });
}
/* Output:
    Element(0,0)=1
    Element(0,1)=2
    Element(1,0)=3
    Element(1,1)=4
    Element(2,0)=5
    Element(2,1)=6
    Element(3,0)=7
    Element(3,1)=8
*/
```

交错数组

交错数组是一个数组，其元素是数组，大小可能不同。 交错数组有时称为“数组的数组”。其元素是引用类型，初始化为 `null`。 以下示例说明如何声明、初始化和访问交错数组。 第一个示例 `jaggedArray` 在一个语句中声明。 每个包含的数组在后续语句中创建。 第二个示例 `jaggedArray2` 在一个语句中声明和初始化。 可以混合使用交错数组和多维数组。 最终示例 `jaggedArray3` 是一维交错数组声明和初始化，包含大小不同的三个二维数组元素。

C#

```
int[][] jaggedArray = new int[3][];
```

```
jaggedArray[0] = [1, 3, 5, 7, 9];
jaggedArray[1] = [0, 2, 4, 6];
jaggedArray[2] = [11, 22];

int[][] jaggedArray2 =
[
    [1, 3, 5, 7, 9],
    [0, 2, 4, 6],
    [11, 22]
];

// Assign 77 to the second element ([1]) of the first array ([0]):
jaggedArray2[0][1] = 77;

// Assign 88 to the second element ([1]) of the third array ([2]):
jaggedArray2[2][1] = 88;

int[][,] jaggedArray3 =
[
    new int[,] { {1,3}, {5,7} },
    new int[,] { {0,2}, {4,6}, {8,10} },
    new int[,] { {11,22}, {99,88}, {0,9} }
];

Console.Write("{0}", jaggedArray3[0][1, 0]);
Console.WriteLine(jaggedArray3.Length);
```

必须先初始化交错数组的元素，然后才能使用它们。 每个元素本身都是数组。 还可以使用初始值设定项填充数组元素的值。 使用初始值设定项时，无需数组大小。

本例生成一个数组，该数组的元素为数组自身。 每一个数组元素都有不同的大小。

C#

```
// Declare the array of two elements.
int[][] arr = new int[2][];

// Initialize the elements.
arr[0] = [1, 3, 5, 7, 9];
arr[1] = [2, 4, 6, 8];

// Display the array elements.
for (int i = 0; i < arr.Length; i++)
{
    System.Console.Write("Element({0}): ", i);

    for (int j = 0; j < arr[i].Length; j++)
    {
        System.Console.Write("{0}{1}", arr[i][j], j == (arr[i].Length - 1) ?
"" : " ");
    }
    System.Console.WriteLine();
}
```

```
/* Output:
    Element(0): 1 3 5 7 9
    Element(1): 2 4 6 8
*/
```

隐式类型的数组

可以创建隐式类型化的数组，其中数组实例的类型通过数组初始值设定项中指定的元素来推断。 针对隐式类型化变量的任何规则也适用于隐式类型化数组。 有关详细信息，请参阅隐式类型局部变量。

下列示例演示了如何创建隐式类型化数组：

C#

```
int[] a = new[] { 1, 10, 100, 1000 }; // int[]

// Accessing array
Console.WriteLine("First element: " + a[0]);
Console.WriteLine("Second element: " + a[1]);
Console.WriteLine("Third element: " + a[2]);
Console.WriteLine("Fourth element: " + a[3]);
/* Outputs
First element: 1
Second element: 10
Third element: 100
Fourth element: 1000
*/

var b = new[] { "hello", null, "world" }; // string[]

// Accessing elements of an array using 'string.Join' method
Console.WriteLine(string.Join(" ", b));
/* Output
hello  world
*/

// single-dimension jagged array
int[][] c =
[
    [1,2,3,4],
    [5,6,7,8]
];
// Looping through the outer array
for (int k = 0; k < c.Length; k++)
{
    // Looping through each inner array
    for (int j = 0; j < c[k].Length; j++)
    {
        // Accessing each element and printing it to the console
        Console.WriteLine($"Element at c[{k}][{j}] is: {c[k][j]}");
```

```
    }
}
/* Outputs
Element at c[0][0] is: 1
Element at c[0][1] is: 2
Element at c[0][2] is: 3
Element at c[0][3] is: 4
Element at c[1][0] is: 5
Element at c[1][1] is: 6
Element at c[1][2] is: 7
Element at c[1][3] is: 8
*/

// jagged array of strings
string[][] d =
[
    ["Luca", "Mads", "Luke", "Dinesh"],
    ["Karen", "Suma", "Frances"]
];

// Looping through the outer array
int i = 0;
foreach (var subArray in d)
{
    // Looping through each inner array
    int j = 0;
    foreach (var element in subArray)
    {
        // Accessing each element and printing it to the console
        Console.WriteLine($"Element at d[{i}][{j}] is: {element}");
        j++;
    }
    i++;
}
/* Outputs
Element at d[0][0] is: Luca
Element at d[0][1] is: Mads
Element at d[0][2] is: Luke
Element at d[0][3] is: Dinesh
Element at d[1][0] is: Karen
Element at d[1][1] is: Suma
Element at d[1][2] is: Frances
*/
```

在上个示例中，请注意对于隐式类型化数组，初始化语句的左侧没有使用方括号。 此外，和一维数组一样，通过使用`new []`来初始化交错数组。

创建包含数组的匿名类型时，必须在类型的对象初始值设定项中隐式类型化数组。 在下列示例中，`contacts`是匿名类型的隐式类型化数组，每个类型都包含名为`PhoneNumbers`的数组。 不在对象初始值设定项中使用`var`关键字。

```
C#
```

```
var contacts = new[]
{
    new
    {
        Name = "Eugene Zabokritski",
        PhoneNumbers = new[] { "206-555-0108", "425-555-0001" }
    },
    new
    {
        Name = "Hanying Feng",
        PhoneNumbers = new[] { "650-555-0199" }
    }
};
```

void（C# 参考）

项目 • 2024/04/22

可以将 `void` 用作方法（或本地函数）的返回类型来指定该方法不返回值。

C#

```
public static void Display(IEnumerable<int> numbers)
{
    if (numbers is null)
    {
        return;
    }

    Console.WriteLine(string.Join(" ", numbers));
}
```

还可以将 `void` 用作引用类型来声明指向未知类型的指针。 有关详细信息，请参阅指针类型。

不能将 `void` 用作变量的类型。

另请参阅

- System.Void

在 GitHub 上与我们协作

可以在 GitHub 上找到此内容的源，还可以在其中创建和查看问题和拉取请求。 有关详细信息，请参阅参与者指南。

.NET 反馈

.NET 是一个开放源代码项目。 选择一个链接以提供反馈：

提出文档问题

提供产品反馈

内置类型（C# 参考）

项目 • 2023/05/10

下表列出了 C# 内置值类型：

C# 类型关键字	.NET 类型
bool	System.Boolean
byte	System.Byte
sbyte	System.SByte
char	System.Char
decimal	System.Decimal
double	System.Double
float	System.Single
int	System.Int32
uint	System.UInt32
nint	System.IntPtr
nuint	System.UIntPtr
long	System.Int64
ulong	System.UInt64
short	System.Int16
ushort	System.UInt16

下表列出了 C# 内置引用类型：

C# 类型关键字	.NET 类型
object	System.Object
string	System.String
dynamic	System.Object

在上表中，左侧列中的每个 C# 类型关键字（dynamic 除外）都是相应 .NET 类型的别名。 它们是可互换的。 例如，以下声明声明了相同类型的变量：

C#

```
int a = 123;
System.Int32 b = 123;
```

void 关键字表示缺少类型。 将其用作不返回值的方法的返回类型。

请参阅

- 使用语言关键字，而非框架类型名称（样式规则 IDE0049）
- C# 参考
- C# 类型的默认值

非托管类型（C# 参考）

项目 • 2024/04/22

如果某个类型是以下类型之一，则它是非托管类型：

- `sbyte`、`byte`、`short`、`ushort`、`int`、`uint`、`long`、`ulong`、`nint`、`nuint`、`char`、`float`、`double`、`decimal` 或 `bool`
- 任何枚举类型
- 任何指针类型
- 成员都是某个非托管类型的一个元组
- 任何仅包含非托管类型字段的用户定义的结构类型。

可使用 unmanaged 约束指定：类型参数为“非指针、不可为 null 的非托管类型”。

仅包含非托管类型的字段的构造结构类型也是非托管类型，如以下示例所示：

C#

```
using System;

public struct Coords<T>
{
    public T X;
    public T Y;
}

public class UnmanagedTypes
{
    public static void Main()
    {
        DisplaySize<Coords<int>>();
        DisplaySize<Coords<double>>();
    }

    private unsafe static void DisplaySize<T>() where T : unmanaged
    {
        Console.WriteLine($"{typeof(T)} is unmanaged and its size is
{sizeof(T)} bytes");
    }
}
// Output:
// Coords`1[System.Int32] is unmanaged and its size is 8 bytes
// Coords`1[System.Double] is unmanaged and its size is 16 bytes
```

泛型结构可以是非托管类型的源，也可以是托管构造类型的源。 前面的示例定义一个泛型结构 `Coords<T>`，并提供非托管构造类型的示例。 托管类型的示例为

`Coords<object>`。 它具有托管性质，因为它包含具托管性质的 `object` 类型的字段。 如果你希望所有 构造类型都是非托管类型，请在泛型结构的定义中使用 `unmanaged` 约束：

C#

```
public struct Coords<T> where T : unmanaged
{
    public T X;
    public T Y;
}
```

C# 语言规范

有关详细信息，请参阅 C# 语言规范的指针类型部分。

另请参阅

- 指针类型
- 内存和跨度相关类型
- sizeof 运算符
- stackalloc

在 GitHub 上与我们协作

可以在 GitHub 上找到此内容的源，还可以在其中创建和查看问题和拉取请求。 有关详细信息，请参阅参与者指南。

.NET 反馈

.NET 是一个开放源代码项目。 选择一个链接以提供反馈：

提出文档问题

提供产品反馈

C# 类型的默认值（C# 参考）

项目 • 2025/01/02

下表显示了 C# 类型的默认值：

展开表

类型	默认值
任何引用类型	`null`
任何内置整数数值类型	0（零）
任何内置浮点型数值类型	0（零）
布尔	`false`
字符	`'\0'` (U+0000)
枚举	表达式 `(E)0` 生成的值，其中 `E` 是枚举标识符。
结构	通过将所有值类型字段设置为其默认值，并将所有引用类型字段设置为 `null` 生成的值。
任何可以为 null 的值类型	HasValue 属性为 `false` 且 Value 属性未定义的实例。 该默认值也称为可空值类型的 *null* 值。

默认值表达式

使用 default 运算符 生成类型的默认值，如以下示例所示：

```
C#
```

```
int a = default(int);
```

可使用 default 文本来初始化变量，使其具有其类型的默认值：

```
C#
```

```
int a = default;
```

值类型的无参数构造函数

对于值类型，*隐式* 无参数构造函数也生成类型的默认值，如以下示例所示：

```
C#
```

```
var n = new System.Numerics.Complex();
Console.WriteLine(n);  // output: (0, 0)
```

在运行时，如果 System.Type 实例表示值类型，则可以使用 Activator.CreateInstance(Type) 方法调用无参数构造函数来获取该类型的默认值。

> ⓘ **备注**
>
> **结构类型**（即值类型）可能具有 **显式无参数构造函数**，该构造函数可能会生成该类型的非默认值。 因此，我们建议使用 `default` 运算符或 `default` 文本来生成类型的默认值。

C# 语言规范

有关详细信息，请参阅 C# 语言规范的以下部分：

- 默认值
- 默认构造函数
- 无参数结构构造函数
- C# 11 - 自动默认结构

另请参阅

- 构造函数

C# 关键字

项目 • 2024/04/11

关键字是预定义的保留标识符，对编译器有特殊意义。 除非前面有 `@` 前缀，否则不能在程序中用作标识符。 例如，`@if` 是有效标识符，而 `if` 则不是，因为 `if` 是关键字。

此主题中的第一个表列出了是 C# 程序任意部分中的保留标识符的关键字。 此主题中的第二个表列出了 C# 中的上下文关键字。 上下文关键字仅在一部分程序上下文中有特殊含义，可以在相应上下文范围之外用作标识符。 一般来说，C# 语言中新增的关键字会作为上下文关键字添加，以免破坏用旧版语言编写的程序。

abstract
as
base
bool
break
byte
case
catch
char
checked
class
const
continue
decimal
default
delegate
do
double
else
enum

event
explicit
extern
false
finally
fixed
float
for

foreach
goto
if
implicit
in
int
interface
internal
is
lock
long

namespace
new
null
object
operator
out
override
params
private
protected
public
readonly
ref
return
sbyte
sealed
short
sizeof
stackalloc

static
string
struct
switch
this
throw
true
try
typeof

uint
ulong
unchecked
unsafe
ushort
using
virtual
void
volatile
while

上下文关键字

上下文关键字用于在代码中提供特定含义，但它不是 C# 中的保留字。 一些上下文关键字（如 `partial` 和 `where`）在两个或多个上下文中有特殊含义。

add
and
alias
ascending
args
async
await
by
descending
dynamic
equals
from

get
global
group
init
into
join
let
托管（函数指针调用约定）
nameof
nint
not

notnull
nuint
on
or
orderby
partial（类型）
partial（方法）
record
remove
select

set
非托管（函数指针调用约定）
unmanaged（泛型类型约束）
value
var
when（筛选条件）
where（泛型类型约束）
where（查询子句）
with
yield

请参阅

- C# 参考

反馈

此页面是否有帮助? 是 否

提供产品反馈

访问修饰符（C# 参考）

项目 • 2024/10/24

访问修饰符是关键字，用于指定成员或类型已声明的可访问性。 本部分介绍 5 个访问修饰符：

- `public`
- `protected`
- `internal`
- `private`
- `file`

可使用访问修饰符指定以下 7 个可访问性级别：

- public：访问不受限制。
- protected：访问限于包含类或派生自包含类的类型。
- internal：访问限于当前程序集。
- protected internal：访问限于当前程序集或派生自包含类的类型。
- private：访问限于包含类。
- private protected：访问限于包含类或当前程序集中派生自包含类的类型。
- file：已声明的类型仅在当前源文件中可见。 文件范围的类型通常用于源生成器。

本部分还介绍以下概念：

- 可访问性级别：使用访问修饰符，声明可访问性级别。
- 可访问性域：指定可在程序段中的何处引用成员。
- 对使用可访问性级别的限制：概括对使用已声明的可访问性级别的限制。

请参阅

- 添加辅助功能修饰符（样式规则 IDE0040）
- C# 关键字
- 访问修饰符
- 访问关键字
- 修饰符

可访问性级别（C# 参考）

项目 • 2024/04/02

使用访问修饰符 `public`、`protected`、`internal` 或 `private`，为成员指定以下声明的可访问性级别之一。

展开表

声明的可访问性	含义
public	访问不受限制。
protected	访问限于包含类或派生自包含类的类型。
internal	访问限于当前程序集。
protected internal	访问限于当前程序集或派生自包含类的类型。
private	访问限于包含类。
private protected	访问限于包含类或当前程序集中派生自包含类的类型。

除使用 `protected internal` 或`private protected` 组合的情况外，一个成员或类型仅允许一个访问修饰符。

命名空间中不允许出现访问修饰符。 命名空间没有任何访问限制。

根据出现成员声明的上下文，仅允许某些声明的可访问性。 如果未在成员声明中指定访问修饰符，则将使用默认可访问性。

未嵌套在其他类型中的顶级类型只能具有 `internal` 或 `public` 可访问性。 这些类型的默认可访问性为 `internal`。

作为其他类型的成员的嵌套类型可以具有如下表所示的声明的可访问性。

展开表

成员	默认成员可访问性	允许的成员的声明的可访问性
`enum`	`public`	无

`class`	`private`	`public`
		`protected`
		`internal`

成员	默认成员可访问性	允许的成员的声明的可访问性

		`private`
		`protected internal`
		`private protected`

`interface`	`public`	`public`
		`protected`
		`internal`
		`private`*
		`protected internal`
		`private protected`

`struct`	`private`	`public`
		`internal`
		`private`

* 具有 `private` 可访问性的 `interface` 成员必须具有默认的实现。

> **备注**
>
> 如果使用 record 关键字修饰符修改类或结构，则允许相同的访问修饰符。
> 此外，使用 record 修饰符，类和结构的默认成员可访问性仍然为 `private`。

嵌套类型的可访问性依赖于它的可访问域，该域是由已声明的成员可访问性和直接包含类型的可访问域这二者共同确定的。 但是，嵌套类型的可访问域不能超出包含类型的可访问域。

C# 语言规范

有关详细信息，请参阅 C# 语言规范。 该语言规范是 C# 语法和用法的权威资料。

另请参阅

- C# 关键字
- 访问修饰符
- 可访问性域
- 可访问性级别的使用限制
- 访问修饰符
- public
- private
- 受保护
- internal

在 GitHub 上与我们协作

可以在 GitHub 上找到此内容的源，还可以在其中创建和查看问题和拉取请求。 有关详细信息，请参阅参与者指南。

.NET

.NET 反馈

.NET 是一个开放源代码项目。 选择一个链接以提供反馈：

提出文档问题

提供产品反馈

可访问域（C# 参考）

项目 • 2024/04/02

成员的可访问域可指定成员可以引用哪些程序分区。 如果成员嵌套于其他类型中，则该成员的可访问域是由该成员的可访问性级别和直接包含类型的可访问域这二者共同确定的。

顶级类型的可访问域至少是在其中进行声明的项目的程序文本。 也就是说，该域包含此项目的所有源文件。 嵌套类型的可访问域至少是在其中进行声明的类型的程序文本。 也就是说，域是类型的主题，其中包括所有嵌套类型。 嵌套类型的可访问域决不能超出包含类型的可访问域。 下例说明了这些概念。

示例

此示例包含一个顶级类型 `T1` 和两个嵌套类 `M1` 和 `M2`。 这两个类包含的字段具有不同的已声明可访问性。 在 `Main` 方法中，每条语句后跟注释以指示每个成员的可访问域。 请注意，试图引用不可访问成员的语句将被注释掉。如果想要查看通过引用不可访问成员引起的编译器错误，请一次删除一条注释。

C#

```
public class T1
{
    public static int publicInt;
    internal static int internalInt;
    private static int privateInt = 0;

    static T1()
    {
        // T1 can access public or internal members
        // in a public or private (or internal) nested class.
        M1.publicInt = 1;
        M1.internalInt = 2;
        M2.publicInt = 3;
        M2.internalInt = 4;

        // Cannot access the private member privateInt
        // in either class:
        // M1.privateInt = 2; //CS0122
    }

    public class M1
    {
        public static int publicInt;
        internal static int internalInt;
        private static int privateInt = 0;
    }
```

```
    private class M2
    {
        public static int publicInt = 0;
        internal static int internalInt = 0;
        private static int privateInt = 0;
    }
}

class MainClass
{
    static void Main()
    {
        // Access is unlimited.
        T1.publicInt = 1;

        // Accessible only in current assembly.
        T1.internalInt = 2;

        // Error CS0122: inaccessible outside T1.
        // T1.privateInt = 3;

        // Access is unlimited.
        T1.M1.publicInt = 1;

        // Accessible only in current assembly.
        T1.M1.internalInt = 2;

        // Error CS0122: inaccessible outside M1.
        //    T1.M1.privateInt = 3;

        // Error CS0122: inaccessible outside T1.
        //    T1.M2.publicInt = 1;

        // Error CS0122: inaccessible outside T1.
        //    T1.M2.internalInt = 2;

        // Error CS0122: inaccessible outside M2.
        //    T1.M2.privateInt = 3;

        // Keep the console open in debug mode.
        System.Console.WriteLine("Press any key to exit.");
        System.Console.ReadKey();
    }
}
```

C# 语言规范

有关详细信息，请参阅 C# 语言规范。 该语言规范是 C# 语法和用法的权威资料。

另请参阅

- C# 关键字
- 访问修饰符
- 可访问性级别
- 可访问性级别的使用限制
- 访问修饰符
- public
- private
- 受保护
- internal

在 GitHub 上与我们协作

可以在 GitHub 上找到此内容的源，还可以在其中创建和查看问题和拉取请求。 有关详细信息，请参阅参与者指南。

.NET 反馈

.NET 是一个开放源代码项目。 选择一个链接以提供反馈：

提出文档问题

提供产品反馈

可访问性级别的使用限制（C# 参考）

项目 • 2024/04/02

在声明中指定类型时，检查类型的可访问性级别是否依赖于成员或其他类型的可访问性级别。 例如，直接基类必须至少具有与派生类相同的可访问性。 以下声明会导致编译器错误，因为基类 `BaseClass` 的可访问性低于 `MyClass`：

C#

```
class BaseClass {...}
public class MyClass: BaseClass {...} // Error
```

下表汇总了对已声明可访问性级别的限制。

展开表

上下文	备注
类	类类型的直接基类必须至少具有与类类型本身相同的可访问性。
接口	接口类型的显式基接口必须至少具有与接口类型本身相同的可访问性。
委派	委托类型的返回类型和参数类型必须至少具有与委托类型本身相同的可访问性。
常量	常量的类型必须至少具有与常量本身相同的可访问性。
Fields	字段的类型必须至少具有与字段本身相同的可访问性。
方法	方法的返回类型和参数类型必须至少具有与方法本身相同的可访问性。
属性	属性的类型必须至少具有与属性本身相同的可访问性。
事件	事件的类型必须至少具有与事件本身相同的可访问性。
索引器	索引器的类型和参数类型必须至少具有与索引器本身相同的可访问性。
运算符	运算符的类型和参数类型必须至少具有与运算符本身相同的可访问性。
构造函数	构造函数的参数类型必须至少具有与构造函数本身相同的可访问性。

示例

下面的示例包含不同类型的错误声明。 每个声明后的注释指示预期编译器错误。

C#

```
// Restrictions on Using Accessibility Levels
// CS0052 expected as well as CS0053, CS0056, and CS0057
// To make the program work, change access level of both class B
// and MyPrivateMethod() to public.

using System;

// A delegate:
delegate int MyDelegate();

class B
{
    // A private method:
    static int MyPrivateMethod()
    {
        return 0;
    }
}

public class A
{
    // Error: The type B is less accessible than the field A.myField.
    public B myField = new B();

    // Error: The type B is less accessible
    // than the constant A.myConst.
    public readonly B myConst = new B();

    public B MyMethod()
    {
        // Error: The type B is less accessible
        // than the method A.MyMethod.
        return new B();
    }

    // Error: The type B is less accessible than the property A.MyProp
    public B MyProp
    {
        set
        {
        }
    }

    MyDelegate d = new MyDelegate(B.MyPrivateMethod);
    // Even when B is declared public, you still get the error:
    // "The parameter B.MyPrivateMethod is not accessible due to
    // protection level."

    public static B operator +(A m1, B m2)
    {
        // Error: The type B is less accessible
        // than the operator A.operator +(A,B)
        return new B();
    }
```

```
    static void Main()
    {
        Console.Write("Compiled successfully");
    }
}
```

C# 语言规范

有关详细信息，请参阅 C# 语言规范。 该语言规范是 C# 语法和用法的权威资料。

另请参阅

- C# 关键字
- 访问修饰符
- 可访问性域
- 可访问性级别
- 访问修饰符
- public
- private
- 受保护
- internal

在 GitHub 上与我们协作

可以在 GitHub 上找到此内容的源，还可以在其中创建和查看问题和拉取请求。 有关详细信息，请参阅参与者指南。

.NET 反馈

.NET 是一个开放源代码项目。 选择一个链接以提供反馈：

提出文档问题

提供产品反馈

internal（C# 参考）

项目 • 2024/04/07

`internal` 关键字是类型和类型成员的访问修饰符。

> 本页涵盖 `internal` 访问。 `internal` 关键字也是 protected internal 访问修饰符的一部分。

只有在同一程序集的文件中，才能访问内部类型或成员，如以下示例所示：

C#

```
public class BaseClass
{
    // Only accessible within the same assembly.
    internal static int x = 0;
}
```

有关 `internal` 和其他访问修饰符的比较，请参阅可访问性级别和访问修饰符。

`assembly` 是通过编译一个或多个源文件而生成的可执行文件或动态链接库 (DLL)。

有关程序集的详细信息，请参阅 .NET 中的程序集。

内部访问通常用于基于组件的开发，因为它可使一组组件以私有方式进行协作，而不必向应用程序代码的其余部分公开。 例如，用于生成图形用户界面的框架可以提供 `Control` 和 `Form` 类，这两个类通过使用具有内部访问权限的成员进行协作。 由于这些成员是内部的，因此不会向正在使用框架的代码公开。

从定义具有内部访问权限的类型或成员的程序集外部引用该类型或成员是错误的。

示例 1

此示例包含两个文件，即 `Assembly1.cs` 和 `Assembly1_a.cs`。 第一个文件包含内部基类 `BaseClass`。 在第二个文件中，尝试实例化 `BaseClass` 会产生错误。

C#

```
// Assembly1.cs
// Compile with: /target:library
internal class BaseClass
{
```

```
    public static int intM = 0;
}
```

C#

```
// Assembly1_a.cs
// Compile with: /reference:Assembly1.dll
class TestAccess
{
   static void Main()
   {
      var myBase = new BaseClass();   // CS0122
   }
}
```

示例 2

在此示例中，使用在示例 1 中所用的相同文件，并将 `BaseClass` 的可访问性级别更改为 `public`。 另将成员 `intM` 的可访问性级别更改为 `internal`。 在此例中，可以实例化类，但不能访问内部成员。

C#

```
// Assembly2.cs
// Compile with: /target:library
public class BaseClass
{
   internal static int intM = 0;
}
```

C#

```
// Assembly2_a.cs
// Compile with: /reference:Assembly2.dll
public class TestAccess
{
   static void Main()
   {
      var myBase = new BaseClass();   // Ok.
      BaseClass.intM = 444;    // CS0117
   }
}
```

C# 语言规范

有关详细信息，请参阅 C# 语言规范中的声明的可访问性。 该语言规范是 C# 语法和用法的权威资料。

另请参阅

- C# 关键字
- 访问修饰符
- 可访问性级别
- 修饰符
- public
- private
- protected

在 GitHub 上与我们协作

可以在 GitHub 上找到此内容的源，还可以在其中创建和查看问题和拉取请求。 有关详细信息，请参阅参与者指南。

.NET

.NET 反馈

.NET 是一个开放源代码项目。 选择一个链接以提供反馈：

提出文档问题

提供产品反馈

private（C# 参考）

项目 • 2024/04/02

`private` 关键字是一个成员访问修饰符。

> 本页涵盖 `private` 访问。 `private` 关键字也是 private protected 访问修饰符的一部分。

私有访问是允许的最低访问级别。 私有成员只有在声明它们的类和结构体中才是可访问的，如以下示例所示：

```
C#

class Employee
{
    private int _i;
    double _d;   // private access by default
}
```

同一体中的嵌套类型也可以访问那些私有成员。

在声明私有成员的类或结构外引用它会导致编译时错误。

有关 `private` 和其他访问修饰符的比较，请参阅可访问性级别和访问修饰符。

示例

在此示例中， `Employee` 类包含两个私有数据成员 `_name` 和 `_salary`。 作为私有成员，它们只能通过成员方法来访问。 添加名为 `GetName` 和 `Salary` 的公共方法，以便可以对私有成员进行受控的访问。 通过公共方法访问 `_name` 成员，而通过公共只读属性访问 `_salary` 成员。 有关详细信息，请参阅 属性。

```
C#

class Employee2
{
    private readonly string _name = "FirstName, LastName";
    private readonly double _salary = 100.0;

    public string GetName()
    {
        return _name;
    }
```

```
    public double Salary
    {
        get { return _salary; }
    }
}

class PrivateTest
{
    static void Main()
    {
        var e = new Employee2();

        // The data members are inaccessible (private), so
        // they can't be accessed like this:
        //    string n = e._name;
        //    double s = e._salary;

        // '_name' is indirectly accessed via method:
        string n = e.GetName();

        // '_salary' is indirectly accessed via property
        double s = e.Salary;
    }
}
```

C# 语言规范

有关详细信息，请参阅 C# 语言规范中的声明的可访问性。 该语言规范是 C# 语法和用法的权威资料。

另请参阅

- C# 关键字
- 访问修饰符
- 可访问性级别
- 修饰符
- public
- 受保护
- internal

在 GitHub 上与我们协作

可以在 GitHub 上找到此内容的源，还可以在其中创建和查看问

.NET

.NET 反馈

.NET 是一个开放源代码项目。 选择一个链接以提供反馈：

题和拉取请求。 有关详细信息，请参阅参与者指南。

提出文档问题

提供产品反馈

protected（C# 参考）

项目 • 2024/04/02

`protected` 关键字是一个成员访问修饰符。

> **备注**
>
> 本页涵盖 `protected` 访问。 `protected` 关键字也属于 protected internal 和 private protected 访问修饰符。

受保护成员在其所在的类中可由派生类实例访问。

有关 `protected` 和其他访问修饰符的比较，请参阅可访问性级别。

示例 1

只有在通过派生类类型进行访问时，基类的受保护成员在派生类中才是可访问的。 以下面的代码段为例：

C#

```
class A
{
    protected int x = 123;
}

class B : A
{
    static void Main()
    {
        var a = new A();
        var b = new B();

        // Error CS1540, because x can only be accessed by
        // classes derived from A.
        // a.x = 10;

        // OK, because this class derives from A.
        b.x = 10;
    }
}
```

语句 `a.x = 10` 生成错误，因为它是在静态方法 Main 中生成的，而不是类 B 的实例。

无法保护结构成员，因为无法继承结构。

示例 2

在此示例中，`DerivedPoint` 类是从 `Point` 派生的。 因此，可以从派生类直接访问基类的受保护成员。

C#

```
class Point
{
    protected int x;
    protected int y;
}

class DerivedPoint: Point
{
    static void Main()
    {
        var dpoint = new DerivedPoint();

        // Direct access to protected members.
        dpoint.x = 10;
        dpoint.y = 15;
        Console.WriteLine($"x = {dpoint.x}, y = {dpoint.y}");
    }
}
// Output: x = 10, y = 15
```

如果将 `x` 和 `y` 的访问级别更改为 private，编译器将发出错误消息：

```
'Point.y' is inaccessible due to its protection level.
```

```
'Point.x' is inaccessible due to its protection level.
```

C# 语言规范

有关详细信息，请参阅 C# 语言规范中的声明的可访问性。 该语言规范是 C# 语法和用法的权威资料。

另请参阅

- C# 关键字
- 访问修饰符
- 可访问性级别
- 修饰符
- public

- private
- internal
- Internal Virtual 关键字的安全问题

在 GitHub 上与我们协作

可以在 GitHub 上找到此内容的源，还可以在其中创建和查看问题和拉取请求。 有关详细信息，请参阅参与者指南。

.NET 反馈

.NET 是一个开放源代码项目。 选择一个链接以提供反馈：

提出文档问题

提供产品反馈

public（C# 参考）

项目 • 2024/03/13

`public` 关键字是类型和类型成员的访问修饰符。 公共访问是允许的最高访问级别。 对访问公共成员没有限制，如以下示例所示：

C#

```
class SampleClass
{
    public int x; // No access restrictions.
}
```

有关详细信息，请参阅访问修饰符和可访问性级别。

示例

在下面的示例中，声明了两个类：`PointTest` 和 `Program`。 直接从 `Program` 访问 `PointTest` 的公共成员 `x` 和 `y`。

C#

```
class PointTest
{
    public int x;
    public int y;
}

class Program
{
    static void Main()
    {
        var p = new PointTest();
        // Direct access to public members.
        p.x = 10;
        p.y = 15;
        Console.WriteLine($"x = {p.x}, y = {p.y}");
    }
}
// Output: x = 10, y = 15
```

如果将 `public` 访问级别更改为 private 或 protected，则会收到错误消息：

“PointTest.y”不可访问，因为它受保护级别限制。

C# 语言规范

有关详细信息，请参阅 C# 语言规范中的声明的可访问性。 该语言规范是 C# 语法和用法的权威资料。

另请参阅

- C# 参考
- 访问修饰符
- C# 关键字
- 访问修饰符
- 可访问性级别
- 修饰符
- private
- 受保护
- internal

在 GitHub 上与我们协作

可以在 GitHub 上找到此内容的源，还可以在其中创建和查看问题和拉取请求。 有关详细信息，请参阅参与者指南。

.NET

.NET 反馈

.NET 是一个开放源代码项目。 选择一个链接以提供反馈：

提出文档问题

提供产品反馈

受保护的内部成员（C# 参考）

项目 • 2024/03/13

`protected internal` 关键字组合是一种成员访问修饰符。 可从当前程序集或派生自包含类的类型访问受保护的内部成员。 有关 `protected internal` 和其他访问修饰符的比较，请参阅可访问性级别。

示例

可从包含程序集内的任何类型访问基类的受保护的内部成员。 也可在另一程序集中的派生类中访问它，前提是通过派生类类型的变量进行访问。 以下面的代码段为例：

C#

```
// Assembly1.cs
// Compile with: /target:library
public class BaseClass
{
   protected internal int myValue = 0;
}

class TestAccess
{
    void Access()
    {
        var baseObject = new BaseClass();
        baseObject.myValue = 5;
    }
}
```

C#

```
// Assembly2.cs
// Compile with: /reference:Assembly1.dll
class DerivedClass : BaseClass
{
    static void Main()
    {
        var baseObject = new BaseClass();
        var derivedObject = new DerivedClass();

        // Error CS1540, because myValue can only be accessed by
        // classes derived from BaseClass.
        // baseObject.myValue = 10;

        // OK, because this class derives from BaseClass.
        derivedObject.myValue = 10;
```

```
    }
}
```

此示例包含两个文件，即 `Assembly1.cs` 和 `Assembly2.cs`。 第一个文件包含公共基类 `BaseClass` 和另一个类 `TestAccess`。 `BaseClass` 拥有受保护的内部成员 `myValue`，由 `TestAccess` 类型访问。 在第二个文件中，如果尝试通过 `BaseClass` 的实例访问 `myValue`，会生成错误，但如果尝试通过一个派生类的实例来访问此成员，`DerivedClass` 会成功。

结构成员不能为 `protected internal`，因为无法继承结构。

替代受保护的内部成员

替代虚拟成员时，被替代方法的辅助功能修饰符取决于定义派生类的程序集。

在与基类相同的程序集中定义派生类时，所有被替代成员具有 `protected internal` 访问权限。 如果派生类是在基类的不同程序集中定义的，则被替代成员具有 `protected` 访问权限。

C#

```
// Assembly1.cs
// Compile with: /target:library
public class BaseClass
{
    protected internal virtual int GetExampleValue()
    {
        return 5;
    }
}

public class DerivedClassSameAssembly : BaseClass
{
    // Override to return a different example value, accessibility modifiers
remain the same.
    protected internal override int GetExampleValue()
    {
        return 9;
    }
}
```

C#

```
// Assembly2.cs
// Compile with: /reference:Assembly1.dll
class DerivedClassDifferentAssembly : BaseClass
{
```

```
    // Override to return a different example value, since this override
    // method is defined in another assembly, the accessibility modifiers
    // are only protected, instead of protected internal.
    protected override int GetExampleValue()
    {
        return 2;
    }
}
```

C# 语言规范

有关详细信息，请参阅 C# 语言规范。 该语言规范是 C# 语法和用法的权威资料。

另请参阅

- C# 参考
- C# 关键字
- 访问修饰符
- 可访问性级别
- 修饰符
- public
- private
- internal
- Internal Virtual 关键字的安全问题

在 GitHub 上与我们协作

可以在 GitHub 上找到此内容的源，还可以在其中创建和查看问题和拉取请求。 有关详细信息，请参阅参与者指南。

.NET

.NET 反馈

.NET 是一个开放源代码项目。 选择一个链接以提供反馈：

提出文档问题

提供产品反馈

private protected（C# 参考）

项目 • 2024/03/13

`private protected` 关键字组合是一种成员访问修饰符。 仅派生自包含类的类型可访问私有受保护成员，但仅能在其包含程序集中访问。 有关 `private protected` 和其他访问修饰符的比较，请参阅可访问性级别。

> **备注**
>
> `private protected` 访问修饰符在 C# 版本 7.2 及更高版本中有效。

示例

仅当变量的静态类型是派生类类型时，可从派生的类型访问基类的私有受保护成员（在其包含程序集中访问）。 以下面的代码段为例：

C#

```
public class BaseClass
{
    private protected int myValue = 0;
}

public class DerivedClass1 : BaseClass
{
    void Access()
    {
        var baseObject = new BaseClass();

        // Error CS1540, because myValue can only be accessed by
        // classes derived from BaseClass.
        // baseObject.myValue = 5;

        // OK, accessed through the current derived class instance
        myValue = 5;
    }
}
```

C#

```
// Assembly2.cs
// Compile with: /reference:Assembly1.dll
class DerivedClass2 : BaseClass
{
    void Access()
```

```
    {
        // Error CS0122, because myValue can only be
        // accessed by types in Assembly1
        // myValue = 10;
    }
}
```

此示例包含两个文件，即 `Assembly1.cs` 和 `Assembly2.cs`。 第一个文件包含公共基类 `BaseClass` 及其派生的类型 `DerivedClass1`。 `BaseClass` 拥有私有受保护成员 `myValue`，`DerivedClass1` 尝试以两种方式访问该成员。 通过 `BaseClass` 的实例第一次尝试访问 `myValue` 时会产生错误。 但是，如果尝试在 `DerivedClass1` 中将其用作继承的成员，则会成功。

在第二个文件中，如果尝试将 `myValue` 作为 `DerivedClass2` 的继承成员进行访问，会生成错误，因为仅 Assembly1 中的派生类型可以访问它。

如果 `Assembly1.cs` 包含命名为 `Assembly2` 的 InternalsVisibleToAttribute，则派生类 `DerivedClass2` 将可以访问 `BaseClass` 中声明的 `private protected` 成员。 `InternalsVisibleTo` 使 `private protected` 成员对其他程序集中的派生类可见。

结构成员不能为 `private protected`，因为无法继承结构。

C# 语言规范

有关详细信息，请参阅 C# 语言规范。 该语言规范是 C# 语法和用法的权威资料。

另请参阅

- C# 参考
- C# 关键字
- 访问修饰符
- 可访问性级别
- 修饰符
- public
- private
- internal
- Internal Virtual 关键字的安全问题

在 GitHub 上与我们协作

.NET 反馈

可以在 GitHub 上找到此内容的源，还可以在其中创建和查看问题和拉取请求。 有关详细信息，请参阅参与者指南。

.NET 是一个开放源代码项目。 选择一个链接以提供反馈：

提出文档问题

提供产品反馈

文件编辑器

项目 • 2025/01/02

从 C# 11 开始，`file` 上下文关键字是类型修饰符。

`file` 修饰符将顶级类型的可见性限制为声明其的文件。 `file` 修饰符最常应用于源生成器编写的类型。 文件本地类型为源生成器提供了一种方便的方法，以避免在生成的类型之间发生名称冲突。 `file` 修饰符声明文件本地类型，如以下示例所示：

C#

```
file class HiddenWidget
{
    // implementation
}
```

嵌套在文件局部类型中的任何类型，也都只能在其声明所在的文件中可见。 程序集中的其他类型可以使用与文件本地类型相同的名称。 由于文件本地类型仅在声明它的文件中可见，因此这些类型不会造成命名冲突。

文件本地类型不能作为在非文件本地类型中声明的任何成员的返回类型或参数类型。 文件本地类型不能是非文件本地的字段成员。 但是，更可见的类型可以隐式实现文件本地接口类型。 该类型还可以 显式实现 文件本地接口，但显式实现只能在同一文件中使用。

以下示例演示使用文件本地类型提供辅助角色方法的公共类型。 此外，公共类型还隐式实现文件本地接口：

C#

```
// In File1.cs:
file interface IWidget
{
    int ProvideAnswer();
}

file class HiddenWidget
{
    public int Work() => 42;
}

public class Widget : IWidget
{
    public int ProvideAnswer()
    {
        var worker = new HiddenWidget();
        return worker.Work();
```

```
    }
}
```

在另一个源文件中，可以声明与文件本地类型具有相同名称的类型。 文件本地类型不可见：

C#

```
// In File2.cs:
// Doesn't conflict with HiddenWidget
// declared in File1.cs
public class HiddenWidget
{
    public void RunTask()
    {
        // omitted
    }
}
```

成员查找首选在同一文件中声明的文件本地类型，而不是在不同文件中声明的非本地类型。 此规则可确保源生成器可以依赖于解析为文件本地类型的成员查找，而不会与其他类型声明产生歧义。 在前面的示例中，*File1.cs* 中 `HiddenWidget` 的所有用法都解析为在*File1.cs*中声明的文件本地类型。 `HiddenWidget` 的文件本地声明隐藏 *File2.cs*中的公共声明。

C# 语言规范

有关详细信息，请参阅 C# 语言规范中的 声明的可访问性，以及 C# 11 - 文件本地类型功能规范。

另请参阅

- C# 关键字
- 访问修饰符
- 可访问性级别
- 修饰符
- 公共
- 受保护
- 内部

abstract（C# 参考）

项目 • 2023/04/07

`abstract` 修饰符指示被修改内容的实现已丢失或不完整。 abstract 修饰符可用于类、方法、属性、索引和事件。 在类声明中使用 `abstract` 修饰符来指示某个类仅用作其他类的基类，而不用于自行进行实例化。 标记为抽象的成员必须由派生自抽象类的非抽象类来实现。

示例 1

在此示例中，类 `Square` 必须提供 `GetArea` 的实现，因为它派生自 `Shape`：

C#

```
abstract class Shape
{
    public abstract int GetArea();
}

class Square : Shape
{
    private int _side;

    public Square(int n) => _side = n;

    // GetArea method is required to avoid a compile-time error.
    public override int GetArea() => _side * _side;

    static void Main()
    {
        var sq = new Square(12);
        Console.WriteLine($"Area of the square = {sq.GetArea()}");
    }
}
// Output: Area of the square = 144
```

抽象类具有以下功能：

- 抽象类不能实例化。
- 抽象类可能包含抽象方法和访问器。
- 无法使用 sealed 修饰符来修改抽象类，因为两个修饰符的含义相反。 `sealed` 修饰符阻止类被继承，而 `abstract` 修饰符要求类被继承。
- 派生自抽象类的非抽象类，必须包含全部已继承的抽象方法和访问器的实际实现。

在方法或属性声明中使用 `abstract` 修饰符，以指示该方法或属性不包含实现。

抽象方法具有以下功能：

- 抽象方法是隐式的虚拟方法。
- 只有抽象类中才允许抽象方法声明。
- 由于抽象方法声明不提供实际的实现，因此没有方法主体；方法声明仅以分号结尾，且签名后没有大括号 ({ })。 例如：

  C#

  ```
  public abstract void MyMethod();
  ```

  实现由方法 override 提供，它是非抽象类的成员。
- 在抽象方法声明中使用 static 或 virtual 修饰符是错误的。

除了声明和调用语法方面不同外，抽象属性的行为与抽象方法相似。

- 在静态属性上使用 `abstract` 修饰符是错误的。
- 通过包含使用 override 修饰符的属性声明，可在派生类中重写抽象继承属性。

有关抽象类的详细信息，请参阅抽象类、密封类及类成员。

抽象类必须为所有接口成员提供实现。

实现接口的抽象类有可能将接口方法映射到抽象方法上。 例如：

C#

```
interface I
{
    void M();
}

abstract class C : I
{
    public abstract void M();
}
```

示例 2

在此示例中，类 `DerivedClass` 派生自抽象类 `BaseClass`。 抽象类包含抽象方法 `AbstractMethod`，以及两个抽象属性 `X` 和 `Y`。

C#

```
// Abstract class
abstract class BaseClass
{
    protected int _x = 100;
    protected int _y = 150;

    // Abstract method
    public abstract void AbstractMethod();

    // Abstract properties
    public abstract int X { get; }
    public abstract int Y { get; }
}

class DerivedClass : BaseClass
{
    public override void AbstractMethod()
    {
        _x++;
        _y++;
    }

    public override int X   // overriding property
    {
        get
        {
            return _x + 10;
        }
    }

    public override int Y   // overriding property
    {
        get
        {
            return _y + 10;
        }
    }

    static void Main()
    {
        var o = new DerivedClass();
        o.AbstractMethod();
        Console.WriteLine($"x = {o.X}, y = {o.Y}");
    }
}
// Output: x = 111, y = 161
```

在前面的示例中，如果你尝试通过使用如下语句来实例化抽象类：

```
C#
```

```
BaseClass bc = new BaseClass();   // Error
```

将遇到一个错误，告知编译器无法创建抽象类“BaseClass”的实例。

C# 语言规范

有关详细信息，请参阅 C# 语言规范。 该语言规范是 C# 语法和用法的权威资料。

请参阅

- C# 参考
- C# 编程指南
- 修饰符
- virtual
- override
- C# 关键字

async（C# 参考）

项目 • 2023/03/23

使用 `async` 修饰符可将方法、lambda 表达式或匿名方法指定为异步。 如果对方法或表达式使用此修饰符，则其称为异步方法 。 如下示例定义了一个名为 `ExampleMethodAsync` 的异步方法：

```
C#

public async Task<int> ExampleMethodAsync()
{
    //...
}
```

如果不熟悉异步编程，或者不了解异步方法如何在不阻止调用方线程的情况下使用 await 运算符执行可能需要长时间运行的工作，请参阅使用 Async 和 Await 的异步编程中的说明。 如下代码见于一种异步方法中，且调用 HttpClient.GetStringAsync 方法：

```
C#

string contents = await httpClient.GetStringAsync(requestUrl);
```

异步方法同步运行，直至到达其第一个 `await` 表达式，此时会将方法挂起，直到等待的任务完成。 同时，如下节示例中所示，控件将返回到方法的调用方。

如果 `async` 关键字修改的方法不包含 `await` 表达式或语句，则该方法将同步执行。 编译器警告将通知你不包含 `await` 语句的任何异步方法，因为该情况可能表示存在错误。 请参阅编译器警告（等级 1）CS4014。

`async` 关键字是上下文关键字，原因在于只有当它修饰方法、lambda 表达式或匿名方法时，它才是关键字。 在所有其他上下文中，都会将其解释为标识符。

示例

下面的示例展示了异步事件处理程序 `StartButton_Click` 和异步方法 `ExampleMethodAsync` 之间的控制结构和流程。 此异步方法的结果是 Web 页面的字符数。 此代码适用于在 Visual Studio 中创建的 Windows Presentation Foundation (WPF) 应用或 Windows 应用商店应用；请参见有关设置应用的代码注释。

可以在 Visual Studio 中将此代码作为 Windows Presentation Foundation (WPF) 应用或 Windows 应用商店应用运行。 需要一个名为 `StartButton` 的按钮控件和一个名为

`ResultsTextBox` 的文本框控件。 切勿忘记设置名称和处理程序，以便获得类似于以下代码的内容：

XAML

```
<Button Content="Button" HorizontalAlignment="Left" Margin="88,77,0,0"
VerticalAlignment="Top" Width="75"
        Click="StartButton_Click" Name="StartButton"/>
<TextBox HorizontalAlignment="Left" Height="137" Margin="88,140,0,0"
TextWrapping="Wrap"
         Text="&lt;Enter a URL&gt;" VerticalAlignment="Top" Width="310"
Name="ResultsTextBox"/>
```

将代码作为 WPF 应用运行：

- 将此代码粘贴到 MainWindow.xaml.cs 中的 `MainWindow` 类中。
- 添加对 System.Net.Http 的引用。
- 为 System.Net.Http 添加一个 `using` 指令。

将此代码作为 Windows 应用商店应用运行：

- 将此代码粘贴到 MainPage.xaml.cs 中的 `MainPage` 类中。
- 为 System.Net.Http 和 System.Threading.Tasks 添加 using 指令。

C#

```
private async void StartButton_Click(object sender, RoutedEventArgs e)
{
    // ExampleMethodAsync returns a Task<int>, which means that the method
    // eventually produces an int result. However, ExampleMethodAsync
returns
    // the Task<int> value as soon as it reaches an await.
    ResultsTextBox.Text += "\n";

    try
    {
        int length = await ExampleMethodAsync();
        // Note that you could put "await ExampleMethodAsync()" in the next
line where
        // "length" is, but due to when '+=' fetches the value of
ResultsTextBox, you
        // would not see the global side effect of ExampleMethodAsync
setting the text.
        ResultsTextBox.Text += String.Format("Length: {0:N0}\n", length);
    }
    catch (Exception)
    {
        // Process the exception if one occurs.
    }
}
```

```
public async Task<int> ExampleMethodAsync()
{
    var httpClient = new HttpClient();
    int exampleInt = (await
httpClient.GetStringAsync("http://msdn.microsoft.com")).Length;
    ResultsTextBox.Text += "Preparing to finish ExampleMethodAsync.\n";
    // After the following return statement, any method that's awaiting
    // ExampleMethodAsync (in this case, StartButton_Click) can get the
    // integer result.
    return exampleInt;
}
// The example displays the following output:
// Preparing to finish ExampleMethodAsync.
// Length: 53292
```

> ⓘ **重要**
>
> 若要深入了解各项任务以及在等待任务期间所执行的代码，请参阅**使用 Async 和 Await 的异步编程**。 有关使用类似元素的完整控制台示例，请参阅**在异步任务完成时对其进行处理 (C#)**。

返回类型

异步方法可具有以下返回类型：

- Task
- Task<TResult>
- void。 对于除事件处理程序以外的代码，通常不鼓励使用 `async void` 方法，因为调用方不能 `await` 那些方法，并且必须实现不同的机制来报告成功完成或错误条件。
- 任何具有可访问的 `GetAwaiter` 方法的类型。 `System.Threading.Tasks.ValueTask<TResult>` 类型属于此类实现。 它通过添加 NuGet 包 `System.Threading.Tasks.Extensions` 的方式可用。

此异步方法既不能声明任何 in、ref 或 out 参数，也不能具有引用返回值，但它可以调用具有此类参数的方法。

如果异步方法的 语句指定一个 类型的操作数，则应指定 `Task<TResult>` 作为方法的返回类型`TResult`。 如果当方法完成时未返回有意义的值，则应使用 `Task`。 即，对方法的调用将返回一个 `Task`，但是当 `Task` 完成时，任何等待 `await` 的所有 `Task` 表达式的计算结果都为 `void`。

你应主要使用 `void` 返回类型来定义事件处理程序，这些处理程序需要此返回类型。 `void` 返回异步方法的调用方不能等待，并且无法捕获该方法引发的异常。

返回另一个类型（通常为值类型），该类型具有 `GetAwaiter` 方法，可最大程度减少代码段性能关键中的内存分配。

有关详细信息和示例，请参阅异步返回类型。

请参阅

- AsyncStateMachineAttribute
- await
- 使用 Async 和 Await 的异步编程
- 在异步任务完成时对其进行处理
- .NET 博客：async/await 在 C# 中的工作原理

const 关键字

项目 • 2025/01/03

使用 `const` 关键字声明常量字段或本地常量。 常量字段和局部变量不是变量，不能修改。 常量可以是数字、布尔值、字符串或 null 引用。 不要创建一个常量来表示你期望随时更改的信息。 例如，不要使用常量字段来存储服务的价格、产品版本号或公司的品牌名称。 这些值可能会随时间而变化，由于编译器传播常量，使用库编译的其他代码必须重新编译才能查看更改。 另请参阅 readonly 关键字。 例如：

C#

```
const int X = 0;
public const double GravitationalConstant = 6.673e-11;
private const string ProductName = "Visual C#";
```

内插字符串 可以是常量，如果使用的所有表达式也是常量字符串。 此功能可以改进生成常量字符串的代码：

C#

```
const string Language = "C#";
const string Platform = ".NET";
const string FullProductName = $"{Platform} - Language: {Language}";
```

言论

常量声明的类型指定声明引入的成员的类型。 本地常量或常量字段的初始值设定项必须是可隐式转换为目标类型的常量表达式。

常量表达式是在编译时可以完全计算的表达式。 因此，引用类型的常量的唯一可能值为字符串和 null 引用。

常量声明可以声明多个常量，例如：

C#

```
public const double X = 1.0, Y = 2.0, Z = 3.0;
```

常量声明中不允许 `static` 修饰符。

常量可以参与常量表达式，如下所示：

C#

```
public const int C1 = 5;
public const int C2 = C1 + 100;
```

> ⓘ **备注**
>
> readonly 关键字与 `const` 关键字不同。 一个 `const` 字段只能在字段声明时进行初始化。 可以在声明或构造函数中初始化 `readonly` 字段。 因此，`readonly` 字段可以具有不同的值，具体取决于使用的构造函数。 此外，尽管 `const` 字段是编译时常量，但 `readonly` 字段可用于运行时常量，如以下行所示：`public static readonly uint l1 = (uint)DateTime.Now.Ticks;`

例子

C#

```
public class ConstTest
{
    class SampleClass
    {
        public int x;
        public int y;
        public const int C1 = 5;
        public const int C2 = C1 + 5;

        public SampleClass(int p1, int p2)
        {
            x = p1;
            y = p2;
        }
    }

    static void Main()
    {
        var mC = new SampleClass(11, 22);
        Console.WriteLine($"x = {mC.x}, y = {mC.y}");
        Console.WriteLine($"C1 = {SampleClass.C1}, C2 = {SampleClass.C2}");
    }
}
/* Output
    x = 11, y = 22
    C1 = 5, C2 = 10
*/
```

以下示例演示如何声明本地常量：

C#

```
public class SealedTest
{
    static void Main()
    {
        const int C = 707;
        Console.WriteLine($"My local constant = {C}");
    }
}
// Output: My local constant = 707
```

C# 语言规范

有关详细信息，请参阅 C# 语言规范的以下部分：

- 常量
- 常量表达式

另请参阅

- C# 关键字
- 只读

event (C# 参考)

项目 • 2024/09/27

借助***事件***成员，对象可以触发通知。 事件用户可以通过提供***事件处理程序***来附加事件的可执行代码。 `event` 关键字声明***事件***。 该事件属于委托类型。 当对象触发事件时，该事件将调用所有提供的事件处理程序。 事件处理程序是添加到事件的委托实例，并在引发事件时执行。 事件用户可以在事件上添加或删除其事件处理程序。

示例

下面的示例演示如何声明和引发使用 EventHandler 作为基础委托类型的事件。 要查看完整代码示例了解如何使用泛型 EventHandler<TEventArgs> 委托类型以及如何订阅事件并创建事件处理程序方法，请参阅如何发布符合 .NET 准则的事件。

C#

```
public class SampleEventArgs
{
    public SampleEventArgs(string text) { Text = text; }
    public string Text { get; } // readonly
}

public class Publisher
{
    // Declare the delegate (if using non-generic pattern).
    public delegate void SampleEventHandler(object sender, SampleEventArgs
e);

    // Declare the event.
    public event SampleEventHandler SampleEvent;

    // Wrap the event in a protected virtual method
    // to enable derived classes to raise the event.
    protected virtual void RaiseSampleEvent()
    {
        // Raise the event in a thread-safe manner using the ?. operator.
        SampleEvent?.Invoke(this, new SampleEventArgs("Hello"));
    }
}
```

事件是一种特殊的多播委托，仅可以从声明事件的类（或派生类）或结构（发布服务器类）中对其进行调用。 如果其他类或结构订阅该事件，则在发布服务器类引发该事件时，将调用其事件处理程序方法。 有关详细信息和代码示例，请参阅事件和委托。

可以将事件标记为public、private、protected、internal、protected internal 或 private protected。 这些访问修饰符定义该类的用户访问该事件的方式。 有关详细信息，请参阅

访问修饰符。

关键字和事件

下列关键字应用于事件。

展开表

关键字	说明	更多信息
static	使事件可供调用方在任何时候进行调用，即使不存在类的实例。	静态类和静态类成员
virtual	允许派生类使用重写关键字重写事件行为。	继承
sealed	指定对于派生类，它不再是虚拟的。	
abstract	编译器不会生成 `add` 和 `remove` 事件访问器块，因此派生类必须提供其自己的实现。	

可以通过使用静态关键字将事件声明为静态事件。 这可使事件可供调用方在任何时候进行调用，即使不存在类的实例。 有关详细信息，请参阅静态类和静态类成员。

可以通过使用虚拟关键字将事件标记为虚事件。 这可使派生类使用重写关键字重写事件行为。 有关详细信息，请参阅继承。 重写虚拟事件的事件也可以为密封，指定对于派生类，它不再是虚拟的。 最后，可以声明事件为抽象，这意味着编译器将不会生成 `add` 和 `remove` 事件访问器块。 因此，派生类必须提供其自己的实现。

C# 语言规范

有关详细信息，请参阅 C# 语言规范。 该语言规范是 C# 语法和用法的权威资料。

另请参阅

- C# 关键字
- add
- remove
- 修饰符
- 如何合并委托（多播委托）

extern（C# 参考）

项目 • 2024/03/13

`extern` 修饰符用于声明在外部实现的方法。 `extern` 修饰符的常见用法是在使用 Interop 服务调入非托管代码时与 `DllImport` 特性一起使用。 在这种情况下，还必须将方法声明为 `static`，如下面的示例所示：

C#

```
[DllImport("avifil32.dll")]
private static extern void AVIFileInit();
```

`extern` 关键字还可以定义外部程序集别名，使得可以从单个程序集中引用同一组件的不同版本。 有关详细信息，请参阅外部别名。

将 abstract 和 `extern` 修饰符一起使用来修改同一成员是错误的做法。 使用 `extern` 修饰符意味着方法是在 C# 代码的外部实现的，而使用 `abstract` 修饰符意味着类中未提供方法实现。

extern 关键字用于 C# 中时会比用于 C++ 中时受到更多的限制。 若要比较 C# 关键字与 C++ 关键字，请参见 C++ 语言参考中的“使用 extern 指定链接”。

示例 1

在此示例中，程序接收来自用户的字符串并将该字符串显示在消息框中。 程序使用从 User32.dll 库导入的 `MessageBox` 方法。

C#

```
//using System.Runtime.InteropServices;
class ExternTest
{
    [DllImport("User32.dll", CharSet=CharSet.Unicode)]
    public static extern int MessageBox(IntPtr h, string m, string c, int
type);

    static int Main()
    {
        string myString;
        Console.Write("Enter your message: ");
        myString = Console.ReadLine();
        return MessageBox((IntPtr)0, myString, "My Message Box", 0);
    }
}
```

示例 2

此示例阐释了调入 C 库（本机 DLL）的 C# 程序。

1. 创建以下 C 文件并将其命名为 `cmdll.c`：

   C

   ```
   // cmdll.c
   // Compile with: -LD
   int __declspec(dllexport) SampleMethod(int i)
   {
     return i*10;
   }
   ```

2. 从 Visual Studio 安装目录打开 Visual Studio x64（或 x32）本机工具命令提示符窗口，并通过在命令提示符处键入“cl -LD cmdll.c”来编译 `cmdll.c` 文件。

3. 在相同的目录中，创建以下 C# 文件并将其命名为 `cm.cs`：

   C#

   ```
   // cm.cs
   using System;
   using System.Runtime.InteropServices;
   public class MainClass
   {
       [DllImport("Cmdll.dll")]
         public static extern int SampleMethod(int x);

       static void Main()
       {
           Console.WriteLine("SampleMethod() returns {0}.",
   SampleMethod(5));
       }
   }
   ```

4. 从 Visual Studio 安装目录打开一个 Visual Studio x64（或 x32）本机工具命令提示符窗口，并通过键入以下内容来编译 `cm.cs` 文件：

   > “csc cm.cs”（针对 x64 命令提示符）或“csc -platform:x86 cm.cs”（针对 x32 命令提示符）

   这将创建可执行文件 `cm.exe`。

5. 运行 `cm.exe`。 `SampleMethod` 方法将值 5 传递到 DLL 文件，这将返回该值与 10 相乘后的结果。 该程序生成以下输出：

```
输出

SampleMethod() returns 50.
```

C# 语言规范

有关详细信息，请参阅 C# 语言规范。 该语言规范是 C# 语法和用法的权威资料。

另请参阅

- System.Runtime.InteropServices.DllImportAttribute
- C# 参考
- C# 关键字
- 修饰符

在 GitHub 上与我们协作

可以在 GitHub 上找到此内容的源，还可以在其中创建和查看问题和拉取请求。 有关详细信息，请参阅参与者指南。

.NET

.NET 反馈

.NET 是一个开放源代码项目。 选择一个链接以提供反馈：

提出文档问题

提供产品反馈

in（泛型修饰符）（C# 参考）

项目 • 2023/04/07

对于泛型类型参数，`in` 关键字可指定类型参数是逆变的。 可以在泛型接口和委托中使用 `in` 关键字。

逆变使你使用的类型可以比泛型参数指定的类型派生程度更小。 这样可以隐式转换实现协变接口的类以及隐式转换委托类型。 引用类型支持泛型类型参数中的协变和逆变，但值类型不支持它们。

仅在类型定义方法参数的类型，而不是方法返回类型时，类型可以在泛型接口或委托中声明为逆变。 `In`、`ref` 和 `out` 参数必须是固定的，这意味着它们既不是协变的，也不是逆变的。

具有逆变类型参数的接口使其方法接受的参数的类型可以比接口类型参数指定的类型派生程度更小。 例如，在 IComparer<T> 接口中，类型 T 是逆变的，可以将 `IComparer<Person>` 类型的对象分配给 `IComparer<Employee>` 类型的对象，而无需使用任何特殊转换方法（如果 `Employee` 继承 `Person`）。

可以向逆变委托分配相同类型的其他委托，不过要使用派生程度更小的泛型类型参数。

有关详细信息，请参阅协变和逆变。

逆变泛型接口

下面的示例演示如何声明、扩展和实现逆变泛型接口。 它还演示如何对实现此接口的类使用隐式转换。

C#

```
// Contravariant interface.
interface IContravariant<in A> { }

// Extending contravariant interface.
interface IExtContravariant<in A> : IContravariant<A> { }

// Implementing contravariant interface.
class Sample<A> : IContravariant<A> { }

class Program
{
    static void Test()
    {
        IContravariant<Object> iobj = new Sample<Object>();
        IContravariant<String> istr = new Sample<String>();
```

```
        // You can assign iobj to istr because
        // the IContravariant interface is contravariant.
        istr = iobj;
    }
}
```

逆变泛型委托

以下示例演示如何声明、实例化和调用逆变泛型委托。 它还演示如何隐式转换委托类型。

C#

```
// Contravariant delegate.
public delegate void DContravariant<in A>(A argument);

// Methods that match the delegate signature.
public static void SampleControl(Control control)
{ }
public static void SampleButton(Button button)
{ }

public void Test()
{

    // Instantiating the delegates with the methods.
    DContravariant<Control> dControl = SampleControl;
    DContravariant<Button> dButton = SampleButton;

    // You can assign dControl to dButton
    // because the DContravariant delegate is contravariant.
    dButton = dControl;

    // Invoke the delegate.
    dButton(new Button());
}
```

C# 语言规范

有关详细信息，请参阅 C# 语言规范。 该语言规范是 C# 语法和用法的权威资料。

请参阅

- out
- 协变和逆变

- 修饰符

new 修饰符（C# 参考）

项目 • 2023/04/07

在用作声明修饰符时，`new` 关键字可以显式隐藏从基类继承的成员。 隐藏继承的成员时，该成员的派生版本将替换基类版本。 这假定该成员的基类版本是可见的，因为如果它被标记为 `private`，或在某些情况下被标记为 `internal`，则基类版本已隐藏。 虽然不使用 `new` 修饰符即可隐藏 `public` 和 `protected` 成员，但将收到编译器警告。 如果使用 `new` 来显式隐藏成员，将禁止此警告。

`new` 关键字还可用于创建类型的实例或用作泛型类型约束。

若要隐藏继承的成员，请使用相同名称在派生类中声明该成员，并使用 `new` 修饰符对其进行修饰。 例如：

C#

```
public class BaseC
{
    public int x;
    public void Invoke() { }
}
public class DerivedC : BaseC
{
    new public void Invoke() { }
}
```

在此示例中，使用 `DerivedC.Invoke` 隐藏了 `BaseC.Invoke`。 字段 `x` 不受影响，因为未使用类似名称将其隐藏。

通过继承隐藏名称采用下列形式之一：

- 通常，在类或结构中引入的常数、字段、属性或类型会隐藏与其共享名称的所有基类成员。 有三种特殊情况。 例如，如果将名称为 `N` 的新字段声明为不可调用的类型，并且基类型将 `N` 声明为一种方法，则新字段在调用语法中不会隐藏基声明。 有关详细信息，请参阅 C# 语言规范中的成员查找部分。
- 类或结构中引入的方法会隐藏基类中共享该名称的属性、字段和类型。 它还会隐藏具有相同签名的所有基类方法。
- 类或结构中引入的索引器会隐藏具有相同签名的所有基类索引器。

对同一成员同时使用 `new` 和 override 是错误的做法，因为这两个修饰符的含义互斥。 `new` 修饰符会用同样的名称创建一个新成员并使原始成员变为隐藏。 `override` 修饰符会扩展继承成员的实现。

在不隐藏继承成员的声明中使用 `new` 修饰符将会生成警告。

示例

在此示例中，基类 `BaseC` 和派生类 `DerivedC` 使用相同的字段名 `x`，从而隐藏了继承字段的值。 此示例演示 `new` 修饰符的用法。 另外还演示了如何使用完全限定名访问基类的隐藏成员。

C#

```
public class BaseC
{
    public static int x = 55;
    public static int y = 22;
}

public class DerivedC : BaseC
{
    // Hide field 'x'.
    new public static int x = 100;

    static void Main()
    {
        // Display the new value of x:
        Console.WriteLine(x);

        // Display the hidden value of x:
        Console.WriteLine(BaseC.x);

        // Display the unhidden member y:
        Console.WriteLine(y);
    }
}
/*
Output:
100
55
22
*/
```

在此示例中，嵌套类隐藏了基类中同名的类。 此示例演示如何使用 `new` 修饰符来消除警告消息，以及如何使用完全限定名来访问隐藏的类成员。

C#

```
public class BaseC
{
    public class NestedC
    {
```

```
        public int x = 200;
        public int y;
    }
}

public class DerivedC : BaseC
{
    // Nested type hiding the base type members.
    new public class NestedC
    {
        public int x = 100;
        public int y;
        public int z;
    }

    static void Main()
    {
        // Creating an object from the overlapping class:
        NestedC c1  = new NestedC();

        // Creating an object from the hidden class:
        BaseC.NestedC c2 = new BaseC.NestedC();

        Console.WriteLine(c1.x);
        Console.WriteLine(c2.x);
    }
}
/*
Output:
100
200
*/
```

如果移除 `new` 修饰符，程序仍将编译和运行，但你会收到以下警告：

```
text
```

```
The keyword new is required on 'MyDerivedC.x' because it hides inherited
member 'MyBaseC.x'.
```

C# 语言规范

有关详细信息，请参阅 C# 语言规范中的 new 修饰符部分。

请参阅

- C# 参考
- C# 编程指南

- C# 关键字
- 修饰符
- 使用 Override 和 New 关键字进行版本控制
- 了解何时使用 Override 和 New 关键字

out（泛型修饰符）（C# 参考）

项目 • 2023/05/09

对于泛型类型参数，`out` 关键字可指定类型参数是协变的。 可以在泛型接口和委托中使用 `out` 关键字。

协变使你使用的类型可以比泛型参数指定的类型派生程度更大。 这样可以隐式转换实现协变接口的类以及隐式转换委托类型。 引用类型支持协变和逆变，但值类型不支持它们。

具有协变类型参数的接口使其方法返回的类型可以比类型参数指定的类型派生程度更大。 例如，因为在 .NET Framework 4 的 IEnumerable<T> 中，类型 T 是协变的，所以可以将 `IEnumerable(Of String)` 类型的对象分配给 `IEnumerable(Of Object)` 类型的对象，而无需使用任何特殊转换方法。

可以向协变委托分配相同类型的其他委托，不过要使用派生程度更大的泛型类型参数。

有关详细信息，请参阅协变和逆变。

示例 - 协变泛型接口

下面的示例演示如何声明、扩展和实现协变泛型接口。 它还演示如何对实现协变接口的类使用隐式转换。

C#

```
// Covariant interface.
interface ICovariant<out R> { }

// Extending covariant interface.
interface IExtCovariant<out R> : ICovariant<R> { }

// Implementing covariant interface.
class Sample<R> : ICovariant<R> { }

class Program
{
    static void Test()
    {
        ICovariant<Object> iobj = new Sample<Object>();
        ICovariant<String> istr = new Sample<String>();

        // You can assign istr to iobj because
        // the ICovariant interface is covariant.
        iobj = istr;
```

```
    }
}
```

在泛型接口中，如果类型参数满足以下条件，则可以声明为协变：

- 类型参数只用作接口方法的返回类型，而不用作方法参数的类型。

  > **备注**
  >
  > 此规则有一个例外。 如果在协变接口中将逆变泛型委托用作方法参数，则可以将协变类型用作此委托的泛型类型参数。 有关协变和逆变泛型委托的详细信息，请参阅**委托中的变体**和**对 Func 和 Action 泛型委托使用变体**。

- 类型参数不用作接口方法的泛型约束。

示例 - 协变泛型委托

以下示例演示如何声明、实例化和调用协变泛型委托。 它还演示如何隐式转换委托类型。

C#

```
// Covariant delegate.
public delegate R DCovariant<out R>();

// Methods that match the delegate signature.
public static Control SampleControl()
{ return new Control(); }

public static Button SampleButton()
{ return new Button(); }

public void Test()
{
    // Instantiate the delegates with the methods.
    DCovariant<Control> dControl = SampleControl;
    DCovariant<Button> dButton = SampleButton;

    // You can assign dButton to dControl
    // because the DCovariant delegate is covariant.
    dControl = dButton;

    // Invoke the delegate.
    dControl();
}
```

在泛型委托中，如果类型仅用作方法返回类型，而不用于方法参数，则可以声明为协变。

C# 语言规范

有关详细信息，请参阅 C# 语言规范。 该语言规范是 C# 语法和用法的权威资料。

请参阅

- 泛型接口中的变体
- in
- 修饰符

override（C# 参考）

项目 • 2024/04/02

扩展或修改继承的方法、属性、索引器或事件的抽象或虚拟实现需要 `override` 修饰符。

在以下示例中，`Square` 类必须提供 `GetArea` 的重写实现，因为 `GetArea` 继承自抽象 `Shape` 类：

C#

```
abstract class Shape
{
    public abstract int GetArea();
}

class Square : Shape
{
    private int _side;

    public Square(int n) => _side = n;

    // GetArea method is required to avoid a compile-time error.
    public override int GetArea() => _side * _side;

    static void Main()
    {
        var sq = new Square(12);
        Console.WriteLine($"Area of the square = {sq.GetArea()}");
    }
}
// Output: Area of the square = 144
```

`override` 方法提供从基类继承的方法的新实现。 通过 `override` 声明重写的方法称为重写基方法。 `override` 方法必须具有与重写基方法相同的签名。 `override` 方法支持协变返回类型。 具体而言，`override` 方法的返回类型可从相应基方法的返回类型派生。

不能重写非虚方法或静态方法。 重写基方法必须是 `virtual`、`abstract` 或 `override`。

`override` 声明不能更改 `virtual` 方法的可访问性。 `override` 方法和 `virtual` 方法必须具有相同级别访问修饰符。

不能使用 `new`、`static` 或 `virtual` 修饰符修改 `override` 方法。

重写属性声明必须指定与继承的属性完全相同的访问修饰符、类型和名称。 只读重写属性支持协变返回类型。 重写属性必须为 `virtual`、`abstract` 或 `override`。

有关如何使用 `override` 关键字的详细信息，请参阅使用 Override 和 New 关键字进行版本控制和了解何时使用 Override 和 New 关键字。 有关继承的信息，请参阅继承。

示例

此示例定义一个名为 `Employee` 的基类和一个名为 `SalesEmployee` 的派生类。 `SalesEmployee` 类包含一个额外字段 `salesbonus`，并且重写方法 `CalculatePay` 以将它考虑在内。

C#

```
class TestOverride
{
    public class Employee
    {
        public string Name { get; }

        // Basepay is defined as protected, so that it may be
        // accessed only by this class and derived classes.
        protected decimal _basepay;

        // Constructor to set the name and basepay values.
        public Employee(string name, decimal basepay)
        {
            Name = name;
            _basepay = basepay;
        }

        // Declared virtual so it can be overridden.
        public virtual decimal CalculatePay()
        {
            return _basepay;
        }
    }

    // Derive a new class from Employee.
    public class SalesEmployee : Employee
    {
        // New field that will affect the base pay.
        private decimal _salesbonus;

        // The constructor calls the base-class version, and
        // initializes the salesbonus field.
        public SalesEmployee(string name, decimal basepay, decimal
salesbonus)
            : base(name, basepay)
        {
            _salesbonus = salesbonus;
        }

        // Override the CalculatePay method
```

```
        // to take bonus into account.
        public override decimal CalculatePay()
        {
            return _basepay + _salesbonus;
        }
    }

    static void Main()
    {
        // Create some new employees.
        var employee1 = new SalesEmployee("Alice", 1000, 500);
        var employee2 = new Employee("Bob", 1200);

        Console.WriteLine($"Employee1 {employee1.Name} earned:
{employee1.CalculatePay()}");
        Console.WriteLine($"Employee2 {employee2.Name} earned:
{employee2.CalculatePay()}");
    }
}
/*
    Output:
    Employee1 Alice earned: 1500
    Employee2 Bob earned: 1200
*/
```

C# 语言规范

有关详细信息，请参阅 C# 语言规范中的重写方法部分。

有关协变返回类型的详细信息，请参阅功能建议说明。

另请参阅

- 继承
- C# 关键字
- 修饰符
- abstract
- virtual
- new（修饰符）
- 多态性

在 GitHub 上与我们协作

可以在 GitHub 上找到此内容的源，还可以在其中创建和查看问

.NET 反馈

.NET 是一个开放源代码项目。 选择一个链接以提供反馈：

题和拉取请求。 有关详细信息，请参阅参与者指南。

提出文档问题

提供产品反馈

readonly（C# 参考）

项目 • 2024/02/05

`readonly` 关键字是一个可在五个上下文中使用的修饰符：

- 在字段声明中，`readonly` 指示只能在声明期间或在同一个类的构造函数中向字段赋值。 可以在字段声明和构造函数中多次分配和重新分配只读字段。

  构造函数退出后，不能分配 `readonly` 字段。 此规则对于值类型和引用类型具有不同的含义：
  - 由于值类型直接包含数据，因此属于 `readonly` 值类型的字段不可变。
  - 由于引用类型包含对其数据的引用，因此属于 `readonly` 引用类型的字段必须始终引用同一对象。 该对象可能是可变的。 `readonly` 修饰符可防止将字段值替换为引用类型的其他实例。 但是，修饰符不会阻止通过只读字段修改字段的实例数据。

  > **⚠ 警告**
  >
  > 包含属于可变引用类型的外部可见只读字段的外部可见类型可能存在安全漏洞，可能会触发警告 CA2104：“不要声明只读可变引用类型。”

- 在 `readonly struct` 类型定义中，`readonly` 指示结构类型是不可变的。 有关详细信息，请参阅结构类型一文中的 readonly 结构一节。

- 在结构类型内的实例成员声明中，`readonly` 指示实例成员不修改结构的状态。 有关详细信息，请参阅结构类型一文中的 readonly 实例成员部分。

- 在 ref readonly 方法返回中，`readonly` 修饰符指示该方法返回一个引用，且不允许向该引用写入内容。
  - 将 ref readonly 参数声明到某个方法。

Readonly 字段示例

在此示例中，即使在类构造函数中给字段 `year` 赋了值，也无法在方法 `ChangeYear` 中更改其值：

C#

```
class Age
{
    private readonly int _year;
```

```
    Age(int year)
    {
        _year = year;
    }
    void ChangeYear()
    {
        //_year = 1967; // Compile error if uncommented.
    }
}
```

只能在下列上下文中对 `readonly` 字段进行赋值：

- 在声明中初始化变量时，例如：

  C#

  ```
  public readonly int y = 5;
  ```

- 在包含实例字段声明的类的实例构造函数中。
- 在包含静态字段声明的类的静态构造函数中。

只有在这些构造函数上下文中，将 `readonly` 字段作为 out 或 ref 参数传递才有效。

备注

`readonly` 关键字不同于 const 关键字。 `const` 字段只能在该字段的声明中初始化。 可以在字段声明和任何构造函数中多次分配 `readonly` 字段。 因此，根据所使用的构造函数， `readonly` 字段可能具有不同的值。 另外，虽然 `const` 字段是编译时常量，但 `readonly` 字段可用于运行时常量，如下面的示例所示：

C#

```
public static readonly uint timeStamp = (uint)DateTime.Now.Ticks;
```

C#

```
public class SamplePoint
{
    public int x;
    // Initialize a readonly field
    public readonly int y = 25;
    public readonly int z;

    public SamplePoint()
    {
```

```
        // Initialize a readonly instance field
        z = 24;
    }

    public SamplePoint(int p1, int p2, int p3)
    {
        x = p1;
        y = p2;
        z = p3;
    }

    public static void Main()
    {
        SamplePoint p1 = new SamplePoint(11, 21, 32);   // OK
        Console.WriteLine($"p1: x={p1.x}, y={p1.y}, z={p1.z}");
        SamplePoint p2 = new SamplePoint();
        p2.x = 55;   // OK
        Console.WriteLine($"p2: x={p2.x}, y={p2.y}, z={p2.z}");
    }
    /*
     Output:
        p1: x=11, y=21, z=32
        p2: x=55, y=25, z=24
    */
}
```

在前面的示例中，如果使用类似以下示例的语句：

C#

```
p2.y = 66;        // Error
```

收到编译器错误消息：

无法对只读的字段赋值（构造函数或变量初始值指定项中除外）

只读实例成员

还可以使用 `readonly` 修饰符来声明实例成员不会修改结构的状态。

C#

```
public readonly double Sum()
{
    return X + Y;
}
```

> **备注**
>
> 对于读/写属性，可以将 `readonly` 修饰符添加到 `get` 访问器。 某些 `get` 访问器可以执行计算并缓存结果，而不是只返回专用字段的值。 将 `readonly` 修饰符添加到 `get` 访问器可确保 `get` 访问器不会通过缓存任何结果来修改对象的内部状态。

可在结构类型一文中的 readonly 实例成员部分查看更多示例。

Ref readonly 返回示例

`ref return` 上的 `readonly` 修饰符指示返回的引用无法修改。 下面的示例返回了一个对来源的引用。 它使用 `readonly` 修饰符来指示调用方无法修改来源：

C#

```
private static readonly SamplePoint s_origin = new SamplePoint(0, 0, 0);
public static ref readonly SamplePoint Origin => ref s_origin;
```

所返回的类型不需要为 `readonly struct`。 `ref` 能返回的任何类型都能由 `ref readonly` 返回。

只读 Ref readonly return 示例

还可以将 `ref readonly return` 与 `struct` 类型上的 `readonly` 实例成员结合使用：

C#

```
public struct ReadonlyRefReadonlyExample
{
    private int _data;

    public readonly ref readonly int ReadonlyRefReadonly(ref int reference)
    {
        // _data = 1; // Compile error if uncommented.
        return ref reference;
    }
}
```

该方法本质上返回一个 `readonly` 引用，以及状态为 `readonly`（无法修改任何实例字段）的实例成员（在本例中为方法）。

C# 语言规范

有关详细信息，请参阅 C# 语言规范。 该语言规范是 C# 语法和用法的权威资料。

你还可查看语言规范建议：

- readonly ref 和 readonly 结构
- readonly 结构成员

请参阅

- 添加只读修饰符（样式规则 IDE0044）
- C# 参考
- C# 编程指南
- C# 关键字
- 修饰符
- const
- 字段

在 GitHub 上与我们协作

可以在 GitHub 上找到此内容的源，还可以在其中创建和查看问题和拉取请求。 有关详细信息，请参阅参与者指南。

.NET

.NET 反馈

.NET 是一个开放源代码项目。 选择一个链接以提供反馈：

提出文档问题

提供产品反馈

sealed（C# 参考）

项目 • 2025/01/13

应用于类时，`sealed` 修饰符会阻止其他类从该类继承。 在下面的示例中，类 `B` 继承自类 `A`，但类无法从类 `B` 继承。

C#

```
class A {}
sealed class B : A {}
```

还可以对替代基类中的虚方法或属性的方法或属性使用 `sealed` 修饰符。 这使你可以允许类派生自你的类并防止它们替代特定虚方法或属性。

例

在以下示例中，`Z` 继承自 `Y`，但 `Z` 不能替代在 `X` 中声明并密封在 `Y` 中的虚拟函数 `F`。

C#

```
class X
{
    protected virtual void F() { Console.WriteLine("X.F"); }
    protected virtual void F2() { Console.WriteLine("X.F2"); }
}

class Y : X
{
    sealed protected override void F() { Console.WriteLine("Y.F"); }
    protected override void F2() { Console.WriteLine("Y.F2"); }
}

class Z : Y
{
    // Attempting to override F causes compiler error CS0239.
    // protected override void F() { Console.WriteLine("Z.F"); }

    // Overriding F2 is allowed.
    protected override void F2() { Console.WriteLine("Z.F2"); }
}
```

在类中定义新方法或属性时，可以通过不将它们声明为虚拟，来防止派生类替代它们。 重写基类型中声明的 `virtual` 成员时，可以使用密封关键字阻止派生类型重写它们，如以下示例所示：

```
public sealed override string ToString() => Value;
```

将 抽象 修饰符与密封类一起使用是错误的，因为抽象类必须由提供抽象方法或属性实现的类继承。

应用于方法或属性时，`sealed` 修饰符必须始终与 override 结合使用。

由于结构是隐式密封的，因此无法继承这些结构。

有关详细信息，请参阅继承。

有关更多示例，请参阅抽象类、密封类及类成员。

C#

```
sealed class SealedClass
{
    public int x;
    public int y;
}

class SealedTest2
{
    static void Main()
    {
        var sc = new SealedClass();
        sc.x = 110;
        sc.y = 150;
        Console.WriteLine($"x = {sc.x}, y = {sc.y}");
    }
}
// Output: x = 110, y = 150
```

在前面的示例中，可以使用以下语句尝试从密封类继承：

`class MyDerivedC: SealedClass {} // Error`

结果是一条错误消息：

`'MyDerivedC': cannot derive from sealed type 'SealedClass'`

言论

若要确定是否密封类、方法或属性，通常应考虑以下两点：

- 派生类通过可以自定义类而可能获得的潜在好处。

- 派生类可能采用使它们无法再正常工作或按预期工作的方式来修改类的可能性。

C# 语言规范

有关详细信息，请参阅 C# 语言规范。 语言规范是 C# 语法和用法的明确来源。

另请参阅

- C# 关键字
- 静态类和静态类成员
- 抽象类、密封类及类成员
- 访问修饰符
- 修饰符
- 替代
- 虚拟

static（C# 参考）

项目 • 2023/04/07

本页介绍 `static` 修饰符关键字。 `static` 关键字也是 using static 指令的一部分。

使用 `static` 修饰符可声明属于类型本身而不是属于特定对象的静态成员。 `static` 修饰符可用于声明 `static` 类。 在类、接口和结构中，可以将 `static` 修饰符添加到字段、方法、属性、运算符、事件和构造函数。 `static` 修饰符不能用于索引器或终结器。 有关详细信息，请参阅静态类和静态类成员。

可以将 `static` 修饰符添加到本地函数。 静态本地函数无法捕获局部变量或实例状态。

从 C# 9.0 开始，可将 `static` 修饰符添加到 Lambda 表达式或匿名方法。 静态Lambda表达式或匿名方法无法捕获局部变量或实例状态。

示例 - 静态类

下面的类声明为 `static` 并且只含 `static` 方法：

C#

```
static class CompanyEmployee
{
    public static void DoSomething() { /*...*/ }
    public static void DoSomethingElse() { /*...*/  }
}
```

常数或类型声明是隐式的 `static` 成员。 不能通过实例引用 `static` 成员。 然而，可以通过类型名称引用它。 例如，请考虑以下类：

C#

```
public class MyBaseC
{
    public struct MyStruct
    {
        public static int x = 100;
    }
}
```

若要引用 `static` 成员 `x`，除非可从相同范围访问该成员，否则请使用完全限定的名称 `MyBaseC.MyStruct.x`：

C#

```
Console.WriteLine(MyBaseC.MyStruct.x);
```

尽管类的实例包含该类的所有实例字段的单独副本，但每个 `static` 字段只有一个副本。

不可以使用 this 引用 `static` 方法或属性访问器。

如果 `static` 关键字应用于类，则类的所有成员都必须为 `static`。

类、接口和 `static` 类可以具有 `static` 构造函数。 在程序开始和实例化类之间的某个时刻调用 `static` 构造函数。

> **备注**
>
> `static` 关键字比用于 C++ 中时受到的限制更多。 若要与 C++ 关键字进行比较，请参阅 Storage classes (C++)（存储类 (C++)）。

若要演示 `static` 成员，请考虑表示公司员工的类。 假定此类包含计数员工的方法和存储员工人数的字段。 方法和字段均不属于任何一个员工实例。 相反，它们属于全体员工这个类。 应将其声明为该类的 `static` 成员。

示例 - 静态字段和方法

此示例读取新员工的姓名和 ID，员工计数器按 1 递增，并显示新员工信息和新员工人数。 此程序从键盘读取员工的当前人数。

C#

```
public class Employee4
{
    public string id;
    public string name;

    public Employee4()
    {
    }

    public Employee4(string name, string id)
    {
        this.name = name;
        this.id = id;
    }

    public static int employeeCounter;
```

```
    public static int AddEmployee()
    {
        return ++employeeCounter;
    }
}

class MainClass : Employee4
{
    static void Main()
    {
        Console.Write("Enter the employee's name: ");
        string name = Console.ReadLine();
        Console.Write("Enter the employee's ID: ");
        string id = Console.ReadLine();

        // Create and configure the employee object.
        Employee4 e = new Employee4(name, id);
        Console.Write("Enter the current number of employees: ");
        string n = Console.ReadLine();
        Employee4.employeeCounter = Int32.Parse(n);
        Employee4.AddEmployee();

        // Display the new information.
        Console.WriteLine($"Name: {e.name}");
        Console.WriteLine($"ID:   {e.id}");
        Console.WriteLine($"New Number of Employees:
{Employee4.employeeCounter}");
    }
}
/*
Input:
Matthias Berndt
AF643G
15
 *
Sample Output:
Enter the employee's name: Matthias Berndt
Enter the employee's ID: AF643G
Enter the current number of employees: 15
Name: Matthias Berndt
ID:   AF643G
New Number of Employees: 16
*/
```

示例 - 静态初始化

此示例演示了如何使用尚未声明的 `static` 字段来初始化另一个 `static` 字段。 在向 `static` 字段显式赋值之后才会定义结果。

```
C#
```

```
class Test
{
    static int x = y;
    static int y = 5;

    static void Main()
    {
        Console.WriteLine(Test.x);
        Console.WriteLine(Test.y);

        Test.x = 99;
        Console.WriteLine(Test.x);
    }
}
/*
Output:
    0
    5
    99
*/
```

C# 语言规范

有关详细信息，请参阅 C# 语言规范。 该语言规范是 C# 语法和用法的权威资料。

另请参阅

- C# 参考
- C# 编程指南
- C# 关键字
- 修饰符
- using static 指令
- 静态类和静态类成员

unsafe（C# 参考）

项目 • 2023/04/07

`unsafe` 关键字表示不安全上下文，该上下文是任何涉及指针的操作所必需的。 有关详细信息，请参阅不安全代码和指针。

可在类型或成员的声明中使用 `unsafe` 修饰符。 因此，类型或成员的整个正文范围均被视为不安全上下文。 以下面使用 `unsafe` 修饰符声明的方法为例：

C#

```
unsafe static void FastCopy(byte[] src, byte[] dst, int count)
{
    // Unsafe context: can use pointers here.
}
```

不安全上下文的范围从参数列表扩展到方法的结尾，因此也可在以下参数列表中使用指针：

C#

```
unsafe static void FastCopy ( byte* ps, byte* pd, int count ) {...}
```

还可以使用不安全块从而能够使用该块内的不安全代码。 例如：

C#

```
unsafe
{
    // Unsafe context: can use pointers here.
}
```

若要编译不安全代码，必须指定 AllowUnsafeBlocks 编译器选项。 不能通过公共语言运行时验证不安全代码。

示例

C#

```
// compile with: -unsafe
class UnsafeTest
{
    // Unsafe method: takes pointer to int.
```

```
    unsafe static void SquarePtrParam(int* p)
    {
        *p *= *p;
    }

    unsafe static void Main()
    {
        int i = 5;
        // Unsafe method: uses address-of operator (&).
        SquarePtrParam(&i);
        Console.WriteLine(i);
    }
}
// Output: 25
```

C# 语言规范

有关详细信息，请参阅 C# 语言规范中的不安全代码。 该语言规范是 C# 语法和用法的权威资料。

请参阅

- C# 参考
- C# 关键字
- 语句
- 不安全代码、指针类型和函数指针
- 指针相关的运算符

virtual（C# 参考）

项目 • 2024/10/31

`virtual` 关键字用于修改方法、属性、索引器或事件声明，并使它们可以在派生类中被重写。 例如，此方法可被任何继承它的类替代：

C#

```
public virtual double Area()
{
    return x * y;
}
```

虚拟成员的实现可由派生类中的替代成员更改。 有关如何使用 `virtual` 关键字的详细信息，请参阅使用 Override 和 New 关键字进行版本控制和了解何时使用 Override 和 New 关键字。

备注

调用虚拟方法时，将为替代的成员检查该对象的运行时类型。 将调用大部分派生类中的该替代成员，如果没有派生类替代该成员，则它可能是原始成员。

默认情况下，方法是非虚拟的。 不能替代非虚方法。

以下示例显示了虚拟属性：

C#

```
class MyBaseClass
{
    // virtual automatically implemented property. Overrides can only
    // provide specialized behavior if they implement get and set accessors.
    public virtual string Name { get; set; }

    // ordinary virtual property with backing field
    private int _num;
    public virtual int Number
    {
        get { return _num; }
        set { _num = value; }
    }
}

class MyDerivedClass : MyBaseClass
{
    private string _name;
```

```
    // Override automatically implemented property with ordinary property
    // to provide specialized accessor behavior.
    public override string Name
    {
        get
        {
            return _name;
        }
        set
        {
            if (!string.IsNullOrEmpty(value))
            {
                _name = value;
            }
            else
            {
                _name = "Unknown";
            }
        }
    }
}
```

除声明和调用语法不同外，虚拟属性的行为与虚拟方法相似。

- 通过包括使用 `override` 修饰符的属性声明，可在派生类中替代虚拟继承属性。

示例

在该示例中，`Shape` 类包含 `x`、`y` 两个坐标和 `Area()` 虚拟方法。 不同的形状类（如 `Circle`、`Cylinder` 和 `Sphere`）继承 `Shape` 类，并为每个图形计算表面积。 每个派生类都有各自的 `Area()` 替代实现。

请注意，继承的类 `Circle` `Cylinder` 和 `Sphere` 均使用初始化基类的构造函数，如下面的声明中所示。

C#

```
public Cylinder(double r, double h): base(r, h) {}
```

根据与方法关联的对象，下面的程序通过调用 `Area()` 方法的相应实现来计算并显示每个对象的相应区域。

C#

```
class TestClass
{
```

```
public class Shape
{
    public const double PI = Math.PI;
    protected double _x, _y;

    public Shape()
    {
    }

    public Shape(double x, double y)
    {
        _x = x;
        _y = y;
    }

    public virtual double Area()
    {
        return _x * _y;
    }
}

public class Circle : Shape
{
    public Circle(double r) : base(r, 0)
    {
    }

    public override double Area()
    {
        return PI * _x * _x;
    }
}

public class Sphere : Shape
{
    public Sphere(double r) : base(r, 0)
    {
    }

    public override double Area()
    {
        return 4 * PI * _x * _x;
    }
}

public class Cylinder : Shape
{
    public Cylinder(double r, double h) : base(r, h)
    {
    }

    public override double Area()
    {
        return 2 * PI * _x * _x + 2 * PI * _x * _y;
    }
```

```
    }

    static void Main()
    {
        double r = 3.0, h = 5.0;
        Shape c = new Circle(r);
        Shape s = new Sphere(r);
        Shape l = new Cylinder(r, h);
        // Display results.
        Console.WriteLine("Area of Circle   = {0:F2}", c.Area());
        Console.WriteLine("Area of Sphere   = {0:F2}", s.Area());
        Console.WriteLine("Area of Cylinder = {0:F2}", l.Area());
    }
}
/*
Output:
Area of Circle   = 28.27
Area of Sphere   = 113.10
Area of Cylinder = 150.80
*/
```

C# 语言规范

有关详细信息，请参阅 C# 语言规范。 该语言规范是 C# 语法和用法的权威资料。

请参阅

- 多态性
- abstract
- override
- new（修饰符）

volatile（C# 参考）

项目 • 2024/03/13

`volatile` 关键字指示一个字段可以由多个同时执行的线程修改。 出于性能原因，编译器，运行时系统甚至硬件都可能重新排列对存储器位置的读取和写入。 声明为 `volatile` 的字段将从某些类型的优化中排除。 不确保从所有执行线程整体来看时所有易失性写入操作均按执行顺序排序。 有关更多信息，请参见 Volatile 类。

> **备注**
>
> 在多处理器系统上，易失性读取操作不保证获取由任何处理器写入该内存位置的最新值。 同样，易失性写入操作不保证写入的值会立即对其他处理器可见。

`volatile` 关键字可应用于以下类型的字段：

- 引用类型。
- 指针类型（在不安全的上下文中）。 请注意，虽然指针本身可以是可变的，但是它指向的对象不能是可变的。 换句话说，不能声明“指向可变对象的指针”。
- 简单类型，如 `sbyte`、`byte`、`short`、`ushort`、`int`、`uint`、`char`、`float` 和 `bool`。
- 具有以下基本类型之一的 `enum` 类型：`byte`、`sbyte`、`short`、`ushort`、`int` 或 `uint`。
- 已知为引用类型的泛型类型参数。
- IntPtr 和 UIntPtr。

其他类型（包括 `double` 和 `long`）无法标记为 `volatile`，因为对这些类型的字段的读取和写入不能保证是原子的。 若要保护对这些类型字段的多线程访问，请使用 Interlocked 类成员或使用 lock 语句保护访问权限。

`volatile` 关键字只能应用于 `class` 或 `struct` 的字段。 不能将局部变量声明为 `volatile`。

示例

下面的示例说明如何将公共字段变量声明为 `volatile`。

```
C#

class VolatileTest
{
```

```
    public volatile int sharedStorage;

    public void Test(int i)
    {
        sharedStorage = i;
    }
}
```

下面的示例演示如何创建辅助线程，并用它与主线程并行执行处理。 有关多线程处理的详细信息，请参阅托管线程处理。

C#

```
public class Worker
{
    // This method is called when the thread is started.
    public void DoWork()
    {
        bool work = false;
        while (!_shouldStop)
        {
            work = !work; // simulate some work
        }
        Console.WriteLine("Worker thread: terminating gracefully.");
    }
    public void RequestStop()
    {
        _shouldStop = true;
    }
    // Keyword volatile is used as a hint to the compiler that this data
    // member is accessed by multiple threads.
    private volatile bool _shouldStop;
}

public class WorkerThreadExample
{
    public static void Main()
    {
        // Create the worker thread object. This does not start the thread.
        Worker workerObject = new Worker();
        Thread workerThread = new Thread(workerObject.DoWork);

        // Start the worker thread.
        workerThread.Start();
        Console.WriteLine("Main thread: starting worker thread...");

        // Loop until the worker thread activates.
        while (!workerThread.IsAlive)
            ;

        // Put the main thread to sleep for 500 milliseconds to
        // allow the worker thread to do some work.
        Thread.Sleep(500);
```

```
        // Request that the worker thread stop itself.
        workerObject.RequestStop();

        // Use the Thread.Join method to block the current thread
        // until the object's thread terminates.
        workerThread.Join();
        Console.WriteLine("Main thread: worker thread has terminated.");
    }
    // Sample output:
    // Main thread: starting worker thread...
    // Worker thread: terminating gracefully.
    // Main thread: worker thread has terminated.
}
```

将 `volatile` 修饰符添加到 `_shouldStop` 的声明后，将始终获得相同的结果（类似于前面代码中显示的片段）。 但是，如果 `_shouldStop` 成员上没有该修饰符，则行为是不可预测的。 `DoWork` 方法可能会优化成员访问，从而导致读取陈旧数据。 鉴于多线程编程的性质，读取陈旧数据的次数是不可预测的。 不同的程序运行会产生一些不同的结果。

C# 语言规范

有关详细信息，请参阅 C# 语言规范。 该语言规范是 C# 语法和用法的权威资料。

请参阅

- C# 语言规范：可变关键字
- C# 参考
- C# 关键字
- 修饰符
- lock 语句
- Interlocked

在 GitHub 上与我们协作

可以在 GitHub 上找到此内容的源，还可以在其中创建和查看问题和拉取请求。 有关详细信息，请参阅参与者指南。

.NET

.NET 反馈

.NET 是一个开放源代码项目。 选择一个链接以提供反馈：

提出文档问题

提供产品反馈

语句关键字（C# 参考）

项目 • 2024/04/02

语句是程序指令。 除了下表中的引用主题介绍的内容外，语句都是按照顺序执行。 下表列出了 C# 语句关键字。 有关不用任何关键字表示的语句的详细信息，请参阅语句。

展开表

类别	C# 关键字
选择语句	`if`, `switch`
迭代语句	`do`, `for`, `foreach`, `while`
跳转语句	`break`, `continue`, `goto`, `return`
异常处理语句	`throw`, `try-catch`, `try-finally`, `try-catch-finally`
checked 和 unchecked 语句	`checked`, `unchecked`
fixed 语句	`fixed`
lock 语句	`lock`
yield 语句	`yield`

另请参阅

- 语句
- C# 关键字

在 GitHub 上与我们协作

可以在 GitHub 上找到此内容的源，还可以在其中创建和查看问题和拉取请求。 有关详细信息，请参阅参与者指南。

.NET 反馈

.NET 是一个开放源代码项目。 选择一个链接以提供反馈：

提出文档问题

提供产品反馈

方法参数和修饰符

项目 • 2024/12/28

默认情况下，C# 中的参数*按值*传递给函数。 这意味着将变量的副本会传递到方法。 对于值（`struct`）类型，*值*的副本将传递到方法。 对于引用（`class`）类型，*引用*的副本将传递到方法。 参数修饰符可让你*按引用*传递参数。

因为结构是值类型，所以按值将结构传递给方法时，该方法接收结构参数的副本并在其上运行。 该方法无法访问调用方法中的原始结构，因此无法对其进行任何更改。 它只能更改副本。

类实例是引用类型，而非值类型。 当引用类型通过值传递给方法时，该方法将接收对实例的引用的副本。 这两个变量都引用同一对象。 参数是引用的副本。 调用的方法无法在调用方法中重新分配实例。 但是，调用的方法可以使用引用的副本来访问实例成员。 如果调用的方法更改实例成员，调用方法也会看到这些更改，因为它引用同一实例。

按值传递并按引用传递

本节中的所有示例都使用以下两种 `record` 类型来说明 `class` 类型和 `struct` 类型之间的差异：

C#

```
public record struct Point(int X, int Y);
// This doesn't use a primary constructor because the properties implemented
for `record` types are
// readonly in record class types. That would prevent the mutations
necessary for this example.
public record class Point3D
{
    public int X { get; set; }
    public int Y { get; set; }
    public int Z { get; set; }
}
```

以下示例的输出说明了按值传递结构类型与按值传递类类型之间的差异。 这两个 `Mutate` 方法更改其参数的属性值。 当参数是 `struct` 类型时，这些更改是对参数数据的副本进行的。 当参数是 `class` 类型时，这些更改是对参数所引用的实例所做的更改：

C#

```
public class PassTypesByValue
{
    public static void Mutate(Point pt)
```

```
    {
        Console.WriteLine($"\tEnter {nameof(Mutate)}:\t\t{pt}");
        pt.X = 19;
        pt.Y = 23;

        Console.WriteLine($"\tExit {nameof(Mutate)}:\t\t{pt}");
    }
    public static void Mutate(Point3D pt)
    {
        Console.WriteLine($"\tEnter {nameof(Mutate)}:\t\t{pt}");
        pt.X = 19;
        pt.Y = 23;
        pt.Z = 42;

        Console.WriteLine($"\tExit {nameof(Mutate)}:\t\t{pt}");
    }

    public static void TestPassTypesByValue()
    {
        Console.WriteLine("===== Value Types =====");

        var ptStruct = new Point { X = 1, Y = 2 };
        Console.WriteLine($"After initialization:\t\t{ptStruct}");

        Mutate(ptStruct);

        Console.WriteLine($"After called {nameof(Mutate)}:\t\t{ptStruct}");

        Console.WriteLine("===== Reference Types =====");

        var ptClass = new Point3D { X = 1, Y = 2, Z = 3 };

        Console.WriteLine($"After initialization:\t\t{ptClass}");

        Mutate(ptClass);
        Console.WriteLine($"After called {nameof(Mutate)}:\t\t{ptClass}");

        // Output:
        // ===== Value Types =====
        // After initialization:           Point { X = 1, Y = 2 }
        //         Enter Mutate:           Point { X = 1, Y = 2 }
        //         Exit Mutate:            Point { X = 19, Y = 23 }
        // After called Mutate:            Point { X = 1, Y = 2 }
        // ===== Reference Types =====
        // After initialization:           Point3D { X = 1, Y = 2, Z = 3 }
        //         Enter Mutate:           Point3D { X = 1, Y = 2, Z = 3 }
        //         Exit Mutate:            Point3D { X = 19, Y = 23, Z = 42
}
        // After called Mutate:            Point3D { X = 19, Y = 23, Z = 42
}
    }
}
```

修饰符是一种通过引用 将参数 传递给方法的方法。 以下代码遵循前面的示例，但按引用传递参数。 通过引用传递结构时，对 `struct` 类型的修改在调用方法中可见。 引用类型通过引用传递时没有语义变化。

C#

```
public class PassTypesByReference
{
    public static void Mutate(ref Point pt)
    {
        Console.WriteLine($"\tEnter {nameof(Mutate)}:\t\t{pt}");
        pt.X = 19;
        pt.Y = 23;

        Console.WriteLine($"\tExit {nameof(Mutate)}:\t\t{pt}");
    }
    public static void Mutate(ref Point3D pt)
    {
        Console.WriteLine($"\tEnter {nameof(Mutate)}:\t\t{pt}");
        pt.X = 19;
        pt.Y = 23;
        pt.Z = 42;

        Console.WriteLine($"\tExit {nameof(Mutate)}:\t\t{pt}");
    }

    public static void TestPassTypesByReference()
    {
        Console.WriteLine("===== Value Types =====");

        var pStruct = new Point { X = 1, Y = 2 };
        Console.WriteLine($"After initialization:\t\t{pStruct}");

        Mutate(ref pStruct);

        Console.WriteLine($"After called {nameof(Mutate)}:\t\t{pStruct}");

        Console.WriteLine("===== Reference Types =====");

        var pClass = new Point3D { X = 1, Y = 2, Z = 3 };

        Console.WriteLine($"After initialization:\t\t{pClass}");

        Mutate(ref pClass);
        Console.WriteLine($"After called {nameof(Mutate)}:\t\t{pClass}");

        // Output:
        // ===== Value Types =====
        // After initialization:           Point { X = 1, Y = 2 }
        //         Enter Mutate:           Point { X = 1, Y = 2 }
        //         Exit Mutate:            Point { X = 19, Y = 23 }
        // After called Mutate:            Point { X = 19, Y = 23 }
        // ===== Reference Types =====
```

```
        // After initialization:           Point3D { X = 1, Y = 2, Z = 3 }
        //         Enter Mutate:           Point3D { X = 1, Y = 2, Z = 3 }
        //         Exit Mutate:            Point3D { X = 19, Y = 23, Z = 42
}
        // After called Mutate:            Point3D { X = 19, Y = 23, Z = 42
}
    }
}
```

前面的示例修改了参数的属性。 方法还可以将参数重新分配给新值。 对于按值或按引用传递的结构体和类类型，再分配的行为有所不同。 以下示例演示了重新分配值传递的参数时结构类型和类类型的行为方式：

C#

```
public class PassByValueReassignment
{
    public static void Reassign(Point pt)
    {
        Console.WriteLine($"\tEnter {nameof(Reassign)}:\t\t{pt}");
        pt = new Point { X = 13, Y = 29 };

        Console.WriteLine($"\tExit {nameof(Reassign)}:\t\t{pt}");
    }

    public static void Reassign(Point3D pt)
    {
        Console.WriteLine($"\tEnter {nameof(Reassign)}:\t\t{pt}");
        pt = new Point3D { X = 13, Y = 29, Z = -42 };

        Console.WriteLine($"\tExit {nameof(Reassign)}:\t\t{pt}");
    }

    public static void TestPassByValueReassignment()
    {
        Console.WriteLine("===== Value Types =====");

        var ptStruct = new Point { X = 1, Y = 2 };
        Console.WriteLine($"After initialization:\t\t{ptStruct}");

        Reassign(ptStruct);

        Console.WriteLine($"After called
{nameof(Reassign)}:\t\t{ptStruct}");

        Console.WriteLine("===== Reference Types =====");

        var ptClass = new Point3D { X = 1, Y = 2, Z = 3 };

        Console.WriteLine($"After initialization:\t\t{ptClass}");

        Reassign(ptClass);
        Console.WriteLine($"After called {nameof(Reassign)}:\t\t{ptClass}");
```

```
        // Output:
        // ===== Value Types =====
        // After initialization:           Point { X = 1, Y = 2 }
        //         Enter Reassign:         Point { X = 1, Y = 2 }
        //         Exit Reassign:          Point { X = 13, Y = 29 }
        // After called Reassign:          Point { X = 1, Y = 2 }
        // ===== Reference Types =====
        // After initialization:           Point3D { X = 1, Y = 2, Z = 3 }
        //         Enter Reassign:         Point3D { X = 1, Y = 2, Z = 3 }
        //         Exit Reassign:          Point3D { X = 13, Y = 29, Z = -42
}
        // After called Reassign:          Point3D { X = 1, Y = 2, Z = 3 }
    }
}
```

前面的示例显示，将参数重新分配给新值时，无论类型是值类型还是引用类型，该更改都不可见于调用方法。 以下示例显示了重新分配通过引用传递的参数时的行为：

C#

```
public class PassByReferenceReassignment
{
    public static void Reassign(ref Point pt)
    {
        Console.WriteLine($"\tEnter {nameof(Reassign)}:\t\t{pt}");
        pt = new Point { X = 13, Y = 29 };

        Console.WriteLine($"\tExit {nameof(Reassign)}:\t\t{pt}");
    }

    public static void Reassign(ref Point3D pt)
    {
        Console.WriteLine($"\tEnter {nameof(Reassign)}:\t\t{pt}");
        pt = new Point3D { X = 13, Y = 29, Z = -42 };

        Console.WriteLine($"\tExit {nameof(Reassign)}:\t\t{pt}");
    }

    public static void TestPassByReferenceReassignment()
    {
        Console.WriteLine("===== Value Types =====");

        var ptStruct = new Point { X = 1, Y = 2 };
        Console.WriteLine($"After initialization:\t\t{ptStruct}");

        Reassign(ref ptStruct);

        Console.WriteLine($"After called
{nameof(Reassign)}:\t\t{ptStruct}");

        Console.WriteLine("===== Reference Types =====");
```

```
        var ptClass = new Point3D { X = 1, Y = 2, Z = 3 };

        Console.WriteLine($"After initialization:\t\t{ptClass}");

        Reassign(ref ptClass);
        Console.WriteLine($"After called {nameof(Reassign)}:\t\t{ptClass}");

        // Output:
        // ===== Value Types =====
        // After initialization:           Point { X = 1, Y = 2 }
        //         Enter Reassign:         Point { X = 1, Y = 2 }
        //         Exit Reassign:          Point { X = 13, Y = 29 }
        // After called Reassign:          Point { X = 13, Y = 29 }
        // ===== Reference Types =====
        // After initialization:           Point3D { X = 1, Y = 2, Z = 3 }
        //         Enter Reassign:         Point3D { X = 1, Y = 2, Z = 3 }
        //         Exit Reassign:          Point3D { X = 13, Y = 29, Z = -42
}
        // After called Reassign:          Point3D { X = 13, Y = 29, Z = -42
}
    }
}
```

前面的示例演示如何在调用上下文中重新分配由引用传递的参数的值。

引用和值的安全上下文

方法可以将参数的值存储在字段中。 当参数按值传递时，这通常是安全的。 值会进行复制，并且当引用类型存储在字段中时，是可以访问的。 为了安全地按引用传递参数，需要编译器定义何时可以安全地将引用分配给新变量。 对于每个表达式，编译器都会定义*安全上下文*来限制对表达式或变量的访问。 编译器使用两个范围：*safe-context* 和 *ref-safe-context*。

- *safe-context* 定义可以安全地访问任何表达式的范围。
- *ref-safe-context* 定义可以安全地访问或修改对任何表达式的*引用*的范围。

在非正式情况下，可以将这些范围视为机制，以确保代码永远不会访问或修改不再有效的引用。 只要一个引用指向的是有效的对象或结构，它就有效。 *safe-context* 定义何时可以对变量赋值或重新赋值。 *ref-safe-context* 定义何时可以*对 ref 赋值*或*对 ref 重新赋值*。 赋值操作会为变量赋一个新值；ref 赋值操作会为变量赋值以引用其他存储位置。

引用参数

将以下修饰符之一应用于参数声明，以按引用而不是按值传递参数：

- ref：在调用方法之前必须初始化参数。 该方法可以将新值赋给参数，但不需要这样做。
- out：该调用方法在调用方法之前不需要初始化参数。 该方法必须向参数赋值。
- ref readonly：在调用方法之前必须初始化参数。 该方法无法向参数赋新值。
- in：在调用方法之前必须初始化参数。 该方法无法向参数赋新值。 编译器可能会创建一个临时变量来保存 `in` 参数的自变量副本。

由引用传递的参数是 *引用变量*。 它没有自己的价值。 相反，它指的是一个不同的变量，称为其 *引用*。 引用变量可以重新赋值，这会更改它的引用对象。

类的成员不能具有仅在 `ref`、`ref readonly`、`in` 或 `out` 方面不同的签名。 如果类型的两个成员之间的唯一区别在于其中一个具有 `ref` 参数，而另一个具有 `out`、`ref readonly` 或 `in` 参数，则会发生编译器错误。 但是，当一个方法具有 `ref`、`ref readonly`、`in` 或 `out` 参数，另一个方法具有值传递的参数时，则可以重载方法，如下面的示例所示。 在其他要求签名匹配的情况下（如隐藏或重写），`in`、`ref`、`ref readonly` 和 `out` 是签名的一部分，相互之间不匹配。

当某个参数具有上述修饰符之一时，相应的自变量可以具有兼容的修饰符：

- 参数 `ref` 的自变量必须包含 `ref` 修饰符。
- 参数 `out` 的自变量必须包含 `out` 修饰符。
- 参数 `in` 的自变量可以选择性包含 `in` 修饰符。 如果 `ref` 修饰符用于自变量，编译器会发出警告。
- `ref readonly` 参数的自变量应包含 `in` 或 `ref` 修饰符，但不能包含两者。 如果两个修饰符均未包含，编译器会发出警告。

使用这些修饰符时，它们描述如何使用自变量：

- `ref` 表示该方法可以读取或写入自变量的值。
- `out` 表示该方法设置自变量的值。
- `ref readonly` 表示该方法可以读取但无法写入自变量的值。 自变量应按引用传递。
- `in` 表示该方法可以读取但无法写入自变量的值。 自变量将按引用或通过临时变量传递。

不能在以下类型的方法中使用以前的参数修饰符：

- 异步方法，通过使用 async 修饰符定义。
- 迭代器方法，包括 yield return 或 `yield break` 语句。

扩展方法还限制使用以下自变量关键字：

- 不能对扩展方法的第一个参数使用 `out` 关键字。

- 当自变量不是 `ref` 或是不被约束为结构的泛型类型时，不能对扩展方法的第一个自变量使用 `struct` 关键字。
- 除非第一个自变量是 `ref readonly` ，否则无法使用 `in` 和 `struct`关键字。
- 即使约束为结构，也不能对任何泛型类型使用 `ref readonly` 和 `in` 关键字。

属性不是变量。 它们是方法。 属性不能是 `ref` 参数的自变量。

`ref` 参数修饰符

若要使用 `ref` 参数，方法定义和调用方法均必须显式使用 `ref` 关键字，如下面的示例所示。 （除了在进行 COM 调用时，调用方法可忽略 `ref`。）

C#

```
void Method(ref int refArgument)
{
    refArgument = refArgument + 44;
}

int number = 1;
Method(ref number);
Console.WriteLine(number);
// Output: 45
```

传递到 `ref` 参数的自变量必须先经过初始化，然后才能传递。

`out` 参数修饰符

若要使用 `out` 参数，方法定义和调用方法均必须显式使用 `out` 关键字。 例如：

C#

```
int initializeInMethod;
OutArgExample(out initializeInMethod);
Console.WriteLine(initializeInMethod);     // value is now 44

void OutArgExample(out int number)
{
    number = 44;
}
```

作为 `out` 自变量传递的变量在方法调用中传递之前不必进行初始化。 但是，被调用的方法需要在返回之前赋一个值。

析构方法使用 `out` 修饰符声明其参数以返回多个值。 其他方法可以为多个返回值返回值元组。

必须先在单独的语句中声明变量，然后才能将其作为 `out` 参数传递。 还可以在方法调用的参数列表而不是单独的变量声明中声明 `out` 变量。 `out` 变量使代码更简洁可读，还能防止在方法调用之前无意中向该变量赋值。 以下示例在调用 `number` 方法时定义 变量。

C#

```
string numberAsString = "1640";

if (Int32.TryParse(numberAsString, out int number))
    Console.WriteLine($"Converted '{numberAsString}' to {number}");
else
    Console.WriteLine($"Unable to convert '{numberAsString}'");
// The example displays the following output:
//       Converted '1640' to 1640
```

还可以声明隐式类型的局部变量。

`ref readonly` 修饰符

方法声明中必须存在 `ref readonly` 修饰符。 呼叫站点的修饰符是可选的。 可以使用 `in` 或 `ref` 修饰符。 `ref readonly` 修饰符在调用站点上无效。 在调用站点中使用的修饰符有助于描述自变量的特征。 仅当自变量为变量且可写时，才能使用 `ref`。 仅当自变量为变量时，才能使用 `in`。 它可能可写或只读。 如果自变量不是变量，而是表达式，则不能添加任一修饰符。 以下示例显示了这些情况。 以下方法使用 `ref readonly` 修饰符指示，出于性能原因，应按引用传递大型结构：

C#

```
public static void ForceByRef(ref readonly OptionStruct thing)
{
    // elided
}
```

可以使用 `ref` 或 `in` 修饰符调用该方法。 如果省略修饰符，编译器会发出警告。 当自变量是表达式而不是变量时，不能添加 `in` 或 `ref` 修饰符，因此应禁止显示警告：

C#

```
ForceByRef(in options);
ForceByRef(ref options);
ForceByRef(options); // Warning! variable should be passed with `ref` or
`in`
```

```
ForceByRef(new OptionStruct()); // Warning, but an expression, so no
variable to reference
```

如果变量是 `readonly` 变量，则必须使用 `in` 修饰符。 如果改用 `ref` 修饰符，编译器将发出错误。

`ref readonly` 修饰符指示该方法期望自变量是变量，而非不是变量的表达式。 不是变量的表达式示例包括常量、方法返回值和属性。 如果自变量不是变量，编译器会发出警告。

`in` 参数修饰符

方法声明中需要 `in` 修饰符，但在调用站点中不需要。

C#

```
int readonlyArgument = 44;
InArgExample(readonlyArgument);
Console.WriteLine(readonlyArgument);     // value is still 44

void InArgExample(in int number)
{
    // Uncomment the following line to see error CS8331
    //number = 19;
}
```

`in` 修饰符允许编译器为自变量创建一个临时变量，并传递对该自变量的只读引用。 当必须转换自变量、从自变量类型进行隐式转换或自变量为不是变量的值时，编译器始终会创建一个临时变量。 例如，当参数是文本值或从属性访问器返回的值时。 当 API 要求按引用传递参数时，请选择 `ref readonly` 修饰符而不是 `in` 修饰符。

使用 `in` 参数定义的方法可能会获得性能优化。 某些 `struct` 类型参数可能很大，在紧凑的循环或关键代码路径中调用方法时，复制这些结构的成本很高。 方法声明 `in` 参数以指定参数可能按引用安全传递，因为所调用的方法不修改该参数的状态。 按引用传递这些参数可以避免（可能产生的）高昂的复制成本。 在调用站点显式添加 `in` 修饰符以确保参数是按引用传递，而不是按值传递。 显式使用 `in` 有以下两个效果：

- 在调用站点指定 `in` 会强制编译器选择使用匹配的 `in` 参数定义的方法。 否则，如果两种方法唯一的区别在于是否存在 `in`，则按值重载的匹配度会更高。
- 指定 `in` 会声明你想按引用传递自变量。 结合 `in` 使用的参数必须代表一个可以直接引用的位置。 `out` 和 `ref` 自变量的相同常规规则适用：不能使用常数、普通属性或其他生成值的表达式。 否则，在调用站点省略 `in` 就会通知编译器你可以创建临

时变量，并按只读引用传递至方法。 编译器创建临时变量以克服一些 `in` 参数的限制：

- 临时变量允许将编译时常数作为 `in` 参数。
- 临时变量允许使用属性或 `in` 参数的其他表达式。
- 存在从自变量类型到参数类型的隐式转换时，临时变量允许使用自变量。

在前面的所有实例中，编译器创建了临时变量，用于存储常数、属性或其他表达式的值。

以下代码阐释了这些规则：

C#

```
static void Method(in int argument)
{
    // implementation removed
}

Method(5); // OK, temporary variable created.
Method(5L); // CS1503: no implicit conversion from long to int
short s = 0;
Method(s); // OK, temporary int created with the value 0
Method(in s); // CS1503: cannot convert from in short to in int
int i = 42;
Method(i); // passed by readonly reference
Method(in i); // passed by readonly reference, explicitly using `in`
```

现在，假设可以使用另一种使用按值自变量的方法。 结果的变化如以下代码所示：

C#

```
static void Method(int argument)
{
    // implementation removed
}

static void Method(in int argument)
{
    // implementation removed
}

Method(5); // Calls overload passed by value
Method(5L); // CS1503: no implicit conversion from long to int
short s = 0;
Method(s); // Calls overload passed by value.
Method(in s); // CS1503: cannot convert from in short to in int
int i = 42;
Method(i); // Calls overload passed by value
Method(in i); // passed by readonly reference, explicitly using `in`
```

最后一个是按引用传递参数的唯一方法调用。

> **备注**
>
> 为了简化操作，前面的代码将 `int` 用作参数类型。 因为大多数新式计算机中的引用都比 `int` 大，所以将单个 `int` 作为只读引用传递没有任何好处。

`params` 修饰符

在方法声明中的 `params` 关键字之后不允许有任何其他参数，并且在方法声明中只允许有一个 `params` 关键字。

`params` 参数的声明类型必须是集合类型。 识别的集合类型包括：

- 一维数组类型 ，在这种情况下，元素类型为 `T[]`。 `T`
- 范围类型：
  - `System.Span<T>`
  - `System.ReadOnlySpan<T>`
    此处，元素类型为 。 `T`
- 具有可访问创建方法及相应元素类型的类型。 创建方法使用用于*集合表达式*的相同属性进行标识。
- 实现 的结构或类类型，其中：System.Collections.Generic.IEnumerable<T>
  - 类型具有一个构造函数，可以在没有参数的情况下调用，并且该构造函数至少与声明成员一样可访问。
  - 类型具有实例（而不是扩展）方法 ，其中： `Add`
    - 可以使用单个值参数调用该方法。
    - 如果方法是泛型方法，则可以从参数推断类型参数。
    - 该方法至少与声明成员一样可访问。 此处，元素类型是类型的迭代类型。
- 接口类型：
  - System.Collections.Generic.IEnumerable<T>
  - System.Collections.Generic.IReadOnlyCollection<T>
  - System.Collections.Generic.IReadOnlyList<T>
  - System.Collections.Generic.ICollection<T>
  - System.Collections.Generic.IList<T> 此处，元素类型为 。 `T`

在 C# 13 之前，参数必须是一维数组。

使用 `params` 参数调用方法时，可以传入：

- 数组元素类型的参数的逗号分隔列表。
- 指定类型的参数的集合。

- 无参数。 如果未发送任何参数，则 `params` 列表的长度为零。

下面的示例演示可向 `params` 形参发送实参的各种方法。

C#

```
public static void ParamsModifierExample(params int[] list)
{
    for (int i = 0; i < list.Length; i++)
    {
        System.Console.Write(list[i] + " ");
    }
    System.Console.WriteLine();
}

public static void ParamsModifierObjectExample(params object[] list)
{
    for (int i = 0; i < list.Length; i++)
    {
        System.Console.Write(list[i] + " ");
    }
    System.Console.WriteLine();
}

public static void TryParamsCalls()
{
    // You can send a comma-separated list of arguments of the
    // specified type.
    ParamsModifierExample(1, 2, 3, 4);
    ParamsModifierObjectExample(1, 'a', "test");

    // A params parameter accepts zero or more arguments.
    // The following calling statement displays only a blank line.
    ParamsModifierObjectExample();

    // An array argument can be passed, as long as the array
    // type matches the parameter type of the method being called.
    int[] myIntArray = { 5, 6, 7, 8, 9 };
    ParamsModifierExample(myIntArray);

    object[] myObjArray = { 2, 'b', "test", "again" };
    ParamsModifierObjectExample(myObjArray);

    // The following call causes a compiler error because the object
    // array cannot be converted into an integer array.
    //ParamsModifierExample(myObjArray);

    // The following call does not cause an error, but the entire
    // integer array becomes the first element of the params array.
    ParamsModifierObjectExample(myIntArray);
}
/*
Output:
    1 2 3 4
```

```
    1 a test

    5 6 7 8 9
    2 b test again
    System.Int32[]
*/
```

当参数的参数 `params` 为集合类型时，重载解析可能会导致歧义。 参数的集合类型必须可转换为参数的集合类型。 当不同的重载为该参数提供更好的转换时，该方法可能更好。 但是，如果参数的参数 `params` 是离散元素或缺失，则具有不同 `params` 参数类型的所有重载都等于该参数。

有关详细信息，请参阅 C# 语言规范中 参数列表 部分。 该语言规范是 C# 语法和用法的权威资料。

命名空间

项目 • 2024/11/07

`namespace` 关键字用于声明包含一组相关对象的作用域。 可以使用命名空间来组织代码元素并创建全局唯一类型。

C#

```
namespace SampleNamespace
{
    class SampleClass { }

    interface ISampleInterface { }

    struct SampleStruct { }

    enum SampleEnum { a, b }

    delegate void SampleDelegate(int i);

    namespace Nested
    {
        class SampleClass2 { }
    }
}
```

文件范围的命名空间声明使你能够作出以下声明：一个文件中的所有类型都在一个命名空间中。 C# 10 中可使用文件范围的命名空间声明。 以下示例与前面的示例类似，但使用了一个文件范围的命名空间声明：

C#

```
using System;

namespace SampleFileScopedNamespace;

class SampleClass { }

interface ISampleInterface { }

struct SampleStruct { }

enum SampleEnum { a, b }

delegate void SampleDelegate(int i);
```

使用文件作用域命名空间中的语句

使用 *文件范围的命名空间*时，语句的位置 `using` 会影响其在文件中的范围。 文件范围的命名空间低于等效的传统命名空间声明，该声明以文件末尾的右括号结尾。 此行为确定在何处 `using` 应用指令，如下所示：

- 如果语句 `using` 放置在文件范围的命名空间声明之前，则它们被视为不在命名空间之外，并解释为完全限定的命名空间。
- `using`如果在文件范围的命名空间声明之后放置这些语句，则它们的范围在命名空间本身内。

例如：

C#

```
// This using is outside the namespace scope, so it applies globally
using System;

namespace SampleNamespace; // File-scoped namespace declaration

// This using is inside the namespace scope
using System.Text;

public class SampleClass
{
    // Class members...
}
```

在上面的示例中，`System`全局可访问，而`System.Text`仅适用于 .`SampleNamespace`

前面的示例不包括嵌套命名空间。 文件范围的命名空间不能包含其他命名空间声明。 不能声明嵌套的命名空间或第二个文件范围的命名空间：

C#

```
namespace SampleNamespace;

class AnotherSampleClass
{
    public void AnotherSampleMethod()
    {
        System.Console.WriteLine(
            "SampleMethod inside SampleNamespace");
    }
}

namespace AnotherNamespace; // Not allowed!
```

```
namespace ANestedNamespace // Not allowed!
{
   // declarations...
}
```

在命名空间中，可以声明零个或多个以下类型：

- class
- interface
- struct
- enum
- delegate
- 可以声明嵌套的命名空间（文件范围的命名空间声明除外）

编译器会添加默认命名空间。 此未命名的命名空间（有时被称为全局命名空间）存在于每个文件中。 它包含未包含在已声明的命名空间中的声明。 全局命名空间中的任何标识符都可用于已命名的命名空间。

命名空间隐式具有公共访问权限。 有关可以分配给命名空间中元素的访问修饰符的讨论，请参阅访问修饰符。

可以在两个或多个声明中定义一个命名空间。 例如，以下示例将两个类定义为 `MyCompany` 命名空间的一部分：

C#

```
namespace MyCompany.Proj1
{
    class MyClass
    {
    }
}

namespace MyCompany.Proj1
{
    class MyClass1
    {
    }
}
```

以下示例显示如何在嵌套命名空间中调用静态方法。

C#

```
namespace SomeNameSpace
{
    public class MyClass
    {
```

```
        static void Main()
        {
            Nested.NestedNameSpaceClass.SayHello();
        }
    }

    // a nested namespace
    namespace Nested
    {
        public class NestedNameSpaceClass
        {
            public static void SayHello()
            {
                Console.WriteLine("Hello");
            }
        }
    }
}
// Output: Hello
```

C# 语言规范

有关详细信息，请参阅 C# 语言规范中的命名空间部分。 有关文件范围命名空间声明的详细信息，请参阅功能规范。

另请参阅

- N命名空间声明首选项（IDE0160 和 IDE0161）
- C# 关键字
- using
- using static
- 命名空间别名限定符 ::
- 命名空间

`using` 指令

项目 • 2024/11/16

`using` 指令允许使用在命名空间中定义的类型，而无需指定该类型的完全限定命名空间。 `using` 指令以基本形式从单个命名空间导入所有类型，如下例所示：

```
C#

using System.Text;
```

可以将两个修饰符应用于 `using` 指令：

- `global` 修饰符与将同一 `using` 指令添加到项目中的每个源文件具有相同的效果。 此修饰符是在 C# 10 中引入的。
- `static` 修饰符从单个类型中导入该 `static` 成员和嵌套类型，而不是导入命名空间中的所有类型。

可以合并这两个修饰符，将静态成员从类型导入到项目中的所有源文件。

还可以通过 using 别名指令为名称空间或类型创建别名。

```
C#

using Project = PC.MyCompany.Project;
```

可对 using 别名指令使用 `global` 修饰符。

> **备注**
>
> `using` 关键字还用于创建 `using` 语句，此类语句有助于确保正确处理 IDisposable 对象（如文件和字体）。 有关 `using` 语句的详细信息，请参阅 **using 语句**。

没有 `global` 修饰符的 `using` 指令的作用域是显示它的文件。

该 `global using` 指令必须出现在所有命名空间和类型声明之前。 所有全局 using 指令都必须出现在源文件中，然后才能显示任何非全局 `using` 指令。

其他 `using` 指令可以显示：

- 源代码文件的开头，位于任何命名空间或类型声明之前。

- 在任何受阻止范围的命名空间中，但在该命名空间中声明的任何命名空间或类型之前。

否则，将生成编译器错误。

创建 `using` 指令，以便在命名空间中使用类型而不必指定命名空间。 `using` 指令不为你提供对嵌套在指定命名空间中的任何命名空间的访问权限。 命名空间分为两类：用户定义的命名空间和系统定义的命名空间。 用户定义的命名空间是在代码中定义的命名空间。 有关系统定义的命名空间的列表，请参阅 .NET API 浏览器。

修饰 `global` 符

向 `using` 指令添加 `global` 修饰符意味着 using 将应用于编译中的所有文件（通常是一个项目）。 `global using` 指令被添加到 C# 10 中。 其语法为：

```
C#

global using <fully-qualified-namespace>;
```

其中，*完全限定命名空间* 是可以引用其类型而不指定命名空间的命名空间的完全限定名称。

global using 指令可以出现在任何源代码文件的开头。 单个文件中的所有 `global using` 指令必须出现在以下内容之前：

- 没有 `using` 修饰符的所有 `global` 指令。
- 文件中的所有命名空间和类型声明。

可以将指令添加到 `global using` 任何源文件。 通常，你想要将它们保存在单个位置。 `global using` 指令的顺序并不重要，无论是在单个文件中，还是在文件之间。

修饰 `global` 符可以与 `static` 修饰符组合。 `global`修饰符可以应用于 *using alias 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。 以下示例允许在项目中的所有文件中使用 System.Math 中声明的所有方法：

```
C#

global using static System.Math;
```

还可以通过将 `<Using>` 项添加到项目文件（例如 `<Using Include="My.Awesome.Namespace" />`）来全局包含命名空间。 有关详细信息，请参阅 <Using> 项。

如果在不同位置复制 `global` using 指令，分析器会发出诊断。 如果为已引用 using 指令的命名空间或类型`global`添加`using`指令，这些分析器也会通知你。 你可能会发现，通过将它们保存在项目中的一个文件中，可以更轻松地管理 `global` 你的使用。

> **ⓘ 重要**
>
> 适用于 .NET 6 的 C# 模板使用顶级语句。 如果你已升级到 .NET 6，则应用程序可能与本文中的代码不匹配。 有关详细信息，请参阅有关**新 C# 模板生成顶级语句**的文章
>
> .NET 6 SDK 还为使用以下 SDK 的项目添加了一组隐式 `global using` 指令：
>
> - Microsoft.NET.Sdk
> - Microsoft.NET.Sdk.Web
> - Microsoft.NET.Sdk.Worker
>
> 这些隐式 `global using` 指令包含项目类型最常见的命名空间。
>
> 有关详细信息，请参阅有关**隐式 using 指令**的文章

修饰 `static` 符

`using static` 指令命名了一种类型，无需指定类型名称即可访问其静态成员和嵌套类型。 其语法为：

```
C#
```

```
using static <fully-qualified-type-name>;
```

`<fully-qualified-type-name>` 是无需指定类型名称即可访问其静态成员和嵌套类型的类型名称。 如果不提供完全限定的类型名称（完整的命名空间名称以及类型名称），则 C# 将生成编译器错误 CS0246：“无法找到类型或命名空间名称‘type/namespace’（是否缺少 using 指令或程序集引用？）”。

`using static` 指令适用于任何具有静态成员（或嵌套类型）的类型，即使该类型还具有实例成员。 但是，只能通过类型实例来调用实例成员。

你可以访问类型的静态成员，而无需限定使用类型名称进行访问：

```
C#
```

```
using static System.Console;
using static System.Math;
class Program
{
    static void Main()
    {
        WriteLine(Sqrt(3*3 + 4*4));
    }
}
```

通常，调用某个静态成员时，即会提供类型名称以及成员名称。 重复输入相同的类型名称来调用该类型的成员将生成详细的晦涩代码。 例如，`Circle` 类的以下定义引用 Math 类的许多成员。

C#

```
using System;

public class Circle
{
   public Circle(double radius)
   {
      Radius = radius;
   }

   public double Radius { get; set; }

   public double Diameter
   {
      get { return 2 * Radius; }
   }

   public double Circumference
   {
      get { return 2 * Radius * Math.PI; }
   }

   public double Area
   {
      get { return Math.PI * Math.Pow(Radius, 2); }
   }
}
```

通过消除每次引用成员时，显式引用 Math 类的需求，`using static` 指令将生成更简洁的代码：

C#

```
using System;
using static System.Math;

public class Circle
{
   public Circle(double radius)
   {
      Radius = radius;
   }

   public double Radius { get; set; }

   public double Diameter
   {
      get { return 2 * Radius; }
   }

   public double Circumference
   {
      get { return 2 * Radius * PI; }
   }

   public double Area
   {
      get { return PI * Pow(Radius, 2); }
   }
}
```

`using static` 仅导入可访问的静态成员和指定类型中声明的嵌套类型。 不导入继承的成员。 可以从任何带 `using static` 指令的已命名类型导入，包括 Visual Basic 模块。 如果 F# 顶级函数在元数据中显示为一个已命名类型（其名称是有效的 C# 标识符）的静态成员，则可以导入该 F# 函数。

`using static` 使指定类型中声明的扩展方法可用于扩展方法查找。 但是，扩展方法的名称不导入到代码中非限定引用的作用域中。

同一编译单元或命名空间中通过不同 `using static` 命令从不同类型导入的具有相同名称的方法组成一个方法组。 这些方法组内的重载解决方法遵循一般 C# 规则。

以下示例使用 `using static` 指令来提供 Console、Math 和 String 类的静态成员，而无需指定其类型名称。

C#

```
using System;
using static System.Console;
using static System.Math;
using static System.String;
```

```
class Program
{
   static void Main()
   {
      Write("Enter a circle's radius: ");
      var input = ReadLine();
      if (!IsNullOrEmpty(input) && double.TryParse(input, out var radius)) {
         var c = new Circle(radius);

         string s = "\nInformation about the circle:\n";
         s = s + Format("   Radius: {0:N2}\n", c.Radius);
         s = s + Format("   Diameter: {0:N2}\n", c.Diameter);
         s = s + Format("   Circumference: {0:N2}\n", c.Circumference);
         s = s + Format("   Area: {0:N2}\n", c.Area);
         WriteLine(s);
      }
      else {
         WriteLine("Invalid input...");
      }
   }
}

public class Circle
{
   public Circle(double radius)
   {
      Radius = radius;
   }

   public double Radius { get; set; }

   public double Diameter
   {
      get { return 2 * Radius; }
   }

   public double Circumference
   {
      get { return 2 * Radius * PI; }
   }

   public double Area
   {
      get { return PI * Pow(Radius, 2); }
   }
}
// The example displays the following output:
//       Enter a circle's radius: 12.45
//
//       Information about the circle:
//          Radius: 12.45
//          Diameter: 24.90
//          Circumference: 78.23
//          Area: 486.95
```

在此示例中， `using static` 该指令也可以应用于 Double 该类型。 添加该指令使得在未指定类型名称情况下调用 TryParse(String, Double) 方法成为可能。 但是，使用没有类型名称的 `TryParse` 创建的代码可读性较差，因为有必要检查 `using static` 指令，以确定所调用的数值类型的 `TryParse` 方法。

`using static` 也适用于 `enum` 类型。 通过在枚举中添加 `using static`，该类型不再需要使用枚举成员。

C#

```
using static Color;

enum Color
{
    Red,
    Green,
    Blue
}

class Program
{
    public static void Main()
    {
        Color color = Green;
    }
}
```

别名 `using`

创建 `using` 别名指令，以便更易于将标识符限定为命名空间或类型。 在任何 `using` 指令中，都必须使用完全限定的命名空间或类型，而无需考虑它之前的 `using` 指令。 `using` 指令的声明中不能使用 `using` 别名。 例如，以下示例生成一个编译器错误：

C#

```
using s = System.Text;
using s.RegularExpressions; // Generates a compiler error.
```

下面的示例显示如何为命名空间定义和使用 `using` 别名：

C#

```
namespace PC
{
    // Define an alias for the nested namespace.
    using Project = PC.MyCompany.Project;
```

```
    class A
    {
        void M()
        {
            // Use the alias
            var mc = new Project.MyClass();
        }
    }
    namespace MyCompany
    {
        namespace Project
        {
            public class MyClass { }
        }
    }
}
```

using 别名指令的右侧不能有开放式泛型类型。 例如，不能为 `List<T>` 创建 using 别名，但可以为 `List<int>` 创建 using 别名。

下面的示例显示如何为类定义 `using` 指令和 `using` 别名：

C#

```
using System;

// Using alias directive for a class.
using AliasToMyClass = NameSpace1.MyClass;

// Using alias directive for a generic class.
using UsingAlias = NameSpace2.MyClass<int>;

namespace NameSpace1
{
    public class MyClass
    {
        public override string ToString()
        {
            return "You are in NameSpace1.MyClass.";
        }
    }
}

namespace NameSpace2
{
    class MyClass<T>
    {
        public override string ToString()
        {
            return "You are in NameSpace2.MyClass.";
        }
    }
```

```
}

namespace NameSpace3
{
    class MainClass
    {
        static void Main()
        {
            var instance1 = new AliasToMyClass();
            Console.WriteLine(instance1);

            var instance2 = new UsingAlias();
            Console.WriteLine(instance2);
        }
    }
}
// Output:
//    You are in NameSpace1.MyClass.
//    You are in NameSpace2.MyClass.
```

从 C# 12 开始，可为以前受限的类型（包括元组类型、指针类型和其他不安全类型）创建别名。 有关更新的规则的详细信息，请参阅功能规范。

限定的别名成员

命名空间别名限定符提供 `::` 对全局命名空间或其他可能由其他实体隐藏的别名的显式访问。

确保 `global::` 令牌后面的 `::` 命名空间查找相对于全局命名空间。 否则，令牌必须解析为使用别名，并且后面的 `::` 令牌必须解析为该别名命名空间中的类型。 以下示例显示了这两种形式：

C#

```
using S = System.Net.Sockets;

class A
{
    public static int x;
}

class C
{
    public void F(int A, object S)
    {
        // Use global::A.x instead of A.x
        global::A.x += A;

        // Using ::, S must resolve to a namespace alias:
```

```
        S::Socket s = S as S::Socket;

        // In this form, if S were a class, it would be a compile-time
error:
        S.Socket s1 = S as S.Socket;
    }
}
```

C# 语言规范

有关详细信息，请参阅 C# 语言规范中的 Using 指令。 该语言规范是 C# 语法和用法的权威资料。

有关全局 using 修饰符的详细信息，请参阅全局 using 功能规范 - C# 10。

另请参阅

- C# 关键字
- 命名空间
- 样式规则 IDE0005 - 删除不必要的“using”指令
- 样式规则 IDE0065 -“using”指令放置
- using 语句

外部别名（C# 参考）

项目 • 2024/03/13

有时你可能不得不引用具有相同的完全限定类型名称的程序集的两个版本。 例如，可能需要在同一应用程序中使用某程序集的两个或多个版本。 通过使用外部程序集别名，可在别名命名的根级别命名空间内包装每个程序集的命名空间，使其能够在同一文件中使用。

> **① 备注**
>
> **外部**关键字还被用作方法修饰符，用于声明在非托管代码中编写的方法。

若要引用具有相同的完全限定类型名称的两个程序集，必须在命令提示符处指定别名，如下所示：

```
/r:GridV1=grid.dll
```

```
/r:GridV2=grid20.dll
```

这将创建外部别名 `GridV1` 和 `GridV2`。 若要从程序中使用这些别名，请通过使用 `extern` 关键字引用它们。 例如:

```
extern alias GridV1;
```

```
extern alias GridV2;
```

每个外部别名声明都会引入与全局命名空间并行（但不位于其中）的额外根级别命名空间。 因此，可以使用其完全限定的名称（根植于相应的命名空间别名中）无歧义地引用每个程序集的类型。

在上一示例中，`GridV1::Grid` 是 `grid.dll` 中的网格控件，`GridV2::Grid` 是 `grid20.dll` 中的网格控件。

使用 Visual Studio

如果你使用的是 Visual Studio，可以按类似方式提供别名。

在 Visual Studio 中向项目添加 grid.dll 和 grid20.dll 的引用。 打开“属性”选项卡，并将别名从“全局”分别更改为“GridV1”和“GridV2”。

按上述方式使用这些别名

```
```

C#

```
extern alias GridV1;

extern alias GridV2;
```

现在，可以通过使用别名指令为命名空间或类型创建别名。 有关详细信息，请参阅 using 指令。

C#

```
using Class1V1 = GridV1::Namespace.Class1;

using Class1V2 = GridV2::Namespace.Class1;
```

C# 语言规范

有关详细信息，请参阅 C# 语言规范。 该语言规范是 C# 语法和用法的权威资料。

另请参阅

- C# 参考
- C# 关键字
- ::运算符
- 引用（C# 编译器选项）

在 GitHub 上与我们协作

可以在 GitHub 上找到此内容的源，还可以在其中创建和查看问题和拉取请求。 有关详细信息，请参阅参与者指南。

.NET

.NET 反馈

.NET 是一个开放源代码项目。 选择一个链接以提供反馈：

提出文档问题

提供产品反馈

new 约束（C# 参考）

项目 • 2024/04/02

`new` 约束指定泛型类或方法声明中的类型实参必须有公共的无参数构造函数。 若要使用 `new` 约束，则该类型不能为抽象类型。

当泛型类创建类型的新实例时，请将 `new` 约束应用于类型参数，如下面的示例所示：

C#

```
class ItemFactory<T> where T : new()
{
    public T GetNewItem()
    {
        return new T();
    }
}
```

当与其他约束一起使用时，`new()` 约束必须最后指定：

C#

```
public class ItemFactory2<T>
    where T : IComparable, new()
{  }
```

有关详细信息，请参阅类型参数的约束。

`new` 关键字还可用于创建类型的实例或用作成员声明修饰符。

C# 语言规范

有关详细信息，请参阅 C# 语言规范中的类型参数约束部分。

另请参阅

- C# 关键字
- 泛型

在 GitHub 上与我们协作

.NET .NET 反馈

可以在 GitHub 上找到此内容的源，还可以在其中创建和查看问题和拉取请求。 有关详细信息，请参阅参与者指南。

.NET 是一个开放源代码项目。 选择一个链接以提供反馈：

提出文档问题

提供产品反馈

where（泛型类型约束）（C# 参考）

项目 • 2024/08/01

泛型定义中的 `where` 子句指定对用作泛型类型、方法、委托或本地函数中类型参数的参数类型的约束。 约束可指定接口、基类或要求泛型类型为引用、值或非托管类型。 约束声明类型参数必须具有的功能，并且约束必须位于任何声明的基类或实现的接口之后。

例如，可以声明一个泛型类 `AGenericClass`，以使类型参数 `T` 实现 IComparable<T> 接口：

C#

```
public class AGenericClass<T> where T : IComparable<T> { }
```

> **备注**
>
> 有关查询表达式中的 where 子句的详细信息，请参阅 **where 子句**。

`where` 子句还可包括基类约束。 基类约束表明用作该泛型类型的类型参数的类型具有指定的类作为基类（或者是该基类）。 该基类约束一经使用，就必须出现在该类型参数的所有其他约束之前。 某些类型不允许作为基类约束：Object、Array 和 ValueType。 以下示例显示现可指定为基类的类型：

C#

```
public class UsingEnum<T> where T : System.Enum { }

public class UsingDelegate<T> where T : System.Delegate { }

public class Multicaster<T> where T : System.MulticastDelegate { }
```

在可为 null 的上下文中，将强制执行基类类型的为 null 性。 如果基类不可为 null（例如 `Base`），则类型参数必须不可为 null。 如果基类可为 null（例如 `Base?`），则类型参数可以是可为 null 的引用类型，也可以是不可为 null 的引用类型。 当基类不可为 null 时，如果类型参数是可为 null 的引用类型，编译器将发出警告。

`where` 子句可指定类型为 `class` 或 `struct`。 `struct` 约束不再需要指定 `System.ValueType` 的基类约束。 `System.ValueType` 类型不能用作基类约束。 以下示例显示 `class` 和 `struct` 约束：

C#

```
class MyClass<T, U>
    where T : class
    where U : struct
{ }
```

在可为 null 的上下文中，`class` 约束要求类型是不可为 null 的引用类型。 若要允许可为 null 的引用类型，请使用 `class?` 约束，该约束允许可为 null 和不可为 null 的引用类型。

`where` 子句可以包含 `notnull` 约束。 `notnull` 约束将类型参数限制为不可为 null 的类型。 该类型可以是值类型，也可以是不可为 null 的引用类型。 对于在 nullable enable 上下文中编译的代码，可以使用 `notnull` 约束。 与其他约束不同，如果类型参数违反 `notnull` 约束，编译器会生成警告而不是错误。 警告仅在 `nullable enable` 上下文中生成。

添加可为空引用类型可能会在泛型方法的 `T?` 含义中产生歧义。 如果 `T` 是 `struct`，则 `T?` 与 System.Nullable<T> 相同。 但是，如果 `T` 是引用类型，则 `T?` 表示 `null` 是有效值。 出现歧义的原因是，重写方法不能包含约束。 新 `default` 约束解决了这种歧义。 当基类或接口声明一个方法的两个重载（一个指定 `struct` 约束，另一个未应用 `struct` 或 `class` 约束）时，需添加该约束：

C#

```
public abstract class B
{
    public void M<T>(T? item) where T : struct { }
    public abstract void M<T>(T? item);

}
```

使用 `default` 约束来指定派生类在派生类中没有约束的情况下重写方法，或指定显式接口实现。 此做法仅对重写基方法的方法或显式接口实现有效：

C#

```
public class D : B
{
    // Without the "default" constraint, the compiler tries to override the
first method in B
    public override void M<T>(T? item) where T : default { }
}
```

ⓘ 重要

包含 `notnull` 约束的泛型声明可以在可为 null 的不明显上下文中使用，但编译器不会强制执行约束。

C#

```
#nullable enable
    class NotNullContainer<T>
        where T : notnull
    {
    }
#nullable restore
```

`where` 子句还可包括 `unmanaged` 约束。 `unmanaged` 约束将类型参数限制为名为"非托管类型"的类型。 `unmanaged` 约束使得在 C# 中编写低级别的互操作代码变得更容易。 此约束支持跨所有非托管类型的可重用例程。 `unmanaged` 约束不能与 `class` 或 `struct` 约束结合使用。 `unmanaged` 约束强制该类型必须为 `struct`：

C#

```
class UnManagedWrapper<T>
    where T : unmanaged
{ }
```

`where` 子句也可包括构造函数约束 `new()`。 该约束使得能够使用 `new` 运算符创建类型参数的实例。 new() 约束可以让编译器知道：提供的任何类型参数都必须具有可访问的无参数构造函数。 例如：

C#

```
public class MyGenericClass<T> where T : IComparable<T>, new()
{
    // The following line is not possible without new() constraint:
    T item = new T();
}
```

除非后跟 `allows ref struct` 反约束，否则 `new()` 约束会出现在 `where` 子句中的最后。 `new()` 约束不能与 `struct` 或 `unmanaged` 约束结合使用。 所有满足这些约束的类型必须具有可访问的无参数构造函数，这使得 `new()` 约束冗余。

此反约束声明 `T` 的类型参数可以是 `ref struct` 类型。 例如：

C#

```
public class GenericRefStruct<T> where T : allows ref struct
{
    // Scoped is allowed because T might be a ref struct
    public void M(scoped T parm)
    {

    }
}
```

该泛型类型或方法必须遵循 `T` 的任何实例的引用安全规则，因为它可能是 `ref struct`。 `allows ref struct` 子句不能与 `class` 或 `class?` 约束结合使用。 `allows ref struct` 反约束必须遵循该类型参数的所有约束。

对于多个类型参数，每个类型参数都使用一个 `where` 子句，例如：

C#

```
public interface IMyInterface { }

namespace CodeExample
{
    class Dictionary<TKey, TVal>
        where TKey : IComparable<TKey>
        where TVal : IMyInterface
    {
        public void Add(TKey key, TVal val) { }
    }
}
```

还可将约束附加到泛型方法的类型参数，如以下示例所示：

C#

```
public void MyMethod<T>(T t) where T : IMyInterface { }
```

请注意，对于委托和方法两者来说，描述类型参数约束的语法是一样的：

C#

```
delegate T MyDelegate<T>() where T : new();
```

有关泛型委托的信息，请参阅泛型委托。

有关约束的语法和用法的详细信息，请参阅类型参数的约束。

C# 语言规范

有关详细信息，请参阅 C# 语言规范。 该语言规范是 C# 语法和用法的权威资料。

另请参阅

- 泛型介绍
- new 约束
- 类型参数的约束

在 GitHub 上与我们协作

可以在 GitHub 上找到此内容的源，还可以在其中创建和查看问题和拉取请求。 有关详细信息，请参阅参与者指南。

.NET 反馈

.NET 是一个开放源代码项目。 选择一个链接以提供反馈：

提出文档问题

提供产品反馈

base（C# 参考）

项目 • 2024/03/13

`base` 关键字用于从派生类中访问基类的成员。 如果要执行以下操作时使用它：

- 调用基类上已被其他方法重写的方法。
- 指定创建派生类实例时应调用的基类构造函数。

仅允许基类访问在构造函数、实例方法和实例属性访问器中进行。

在静态方法中使用 `base` 关键字将产生错误。

所访问的基类是类声明中指定的基类。 例如，如果指定 `class ClassB : ClassA`，则从 ClassB 访问 ClassA 的成员，而不考虑 ClassA 的基类。

示例 1

在此示例中，基类 `Person` 和派生类 `Employee` 都有一个名为 `GetInfo` 的方法。 通过使用 `base` 关键字，可以从派生类中调用基类的 `GetInfo` 方法。

C#

```
public class Person
{
    protected string ssn = "444-55-6666";
    protected string name = "John L. Malgraine";

    public virtual void GetInfo()
    {
        Console.WriteLine("Name: {0}", name);
        Console.WriteLine("SSN: {0}", ssn);
    }
}
class Employee : Person
{
    public string id = "ABC567EFG";
    public override void GetInfo()
    {
        // Calling the base class GetInfo method:
        base.GetInfo();
        Console.WriteLine("Employee ID: {0}", id);
    }
}

class TestClass
{
    static void Main()
```

```
    {
        Employee E = new Employee();
        E.GetInfo();
    }
}
/*
Output
Name: John L. Malgraine
SSN: 444-55-6666
Employee ID: ABC567EFG
*/
```

有关其他示例，请参阅 new、virtual 和 override。

示例 2

本示例显示如何指定在创建派生类实例时调用的基类构造函数。

C#

```
public class BaseClass
{
    int num;

    public BaseClass()
    {
        Console.WriteLine("in BaseClass()");
    }

    public BaseClass(int i)
    {
        num = i;
        Console.WriteLine("in BaseClass(int i)");
    }

    public int GetNum()
    {
        return num;
    }
}

public class DerivedClass : BaseClass
{
    // This constructor will call BaseClass.BaseClass()
    public DerivedClass() : base()
    {
    }

    // This constructor will call BaseClass.BaseClass(int i)
    public DerivedClass(int i) : base(i)
```

```
    {
    }

    static void Main()
    {
        DerivedClass md = new DerivedClass();
        DerivedClass md1 = new DerivedClass(1);
    }
}
/*
Output:
in BaseClass()
in BaseClass(int i)
*/
```

C# 语言规范

有关详细信息，请参阅 C# 语言规范。 该语言规范是 C# 语法和用法的权威资料。

另请参阅

- C# 参考
- C# 关键字
- this

在 GitHub 上与我们协作

可以在 GitHub 上找到此内容的源，还可以在其中创建和查看问题和拉取请求。 有关详细信息，请参阅参与者指南。

.NET 反馈

.NET 是一个开放源代码项目。 选择一个链接以提供反馈：

提出文档问题

提供产品反馈

this（C# 参考）

项目 • 2024/04/02

`this` 关键字指代类的当前实例，还可用作扩展方法的第一个参数的修饰符。

> **备注**
>
> 本文介绍 `this` 在类实例中的用法。 若要深入了解它在扩展方法中的用法，请参阅**扩展方法**。

以下是 `this` 的常见用法：

- 限定类似名称隐藏的成员，例如：

```
C#

public class Employee
{
    private string alias;
    private string name;

    public Employee(string name, string alias)
    {
        // Use this to qualify the members of the class
        // instead of the constructor parameters.
        this.name = name;
        this.alias = alias;
    }
}
```

- 将对象作为参数传递给方法，例如：

```
C#

CalcTax(this);
```

- 声明索引器，例如：

```
C#

public int this[int param]
{
    get { return array[param]; }
```

```
        set { array[param] = value; }
    }
```

静态成员函数，因为它们存在于类级别且不属于对象，不具有 `this` 指针。 在静态方法中引用 `this` 会生成错误。

示例

在此示例中，`this` 用于限定类似名称隐藏的 `Employee` 类成员、`name` 和 `alias`。 它还用于将某个对象传递给属于其他类的方法 `CalcTax`。

C#

```
class Employee
{
    private string name;
    private string alias;
    private decimal salary = 3000.00m;

    // Constructor:
    public Employee(string name, string alias)
    {
        // Use this to qualify the fields, name and alias:
        this.name = name;
        this.alias = alias;
    }

    // Printing method:
    public void printEmployee()
    {
        Console.WriteLine("Name: {0}\nAlias: {1}", name, alias);
        // Passing the object to the CalcTax method by using this:
        Console.WriteLine("Taxes: {0:C}", Tax.CalcTax(this));
    }

    public decimal Salary
    {
        get { return salary; }
    }
}

class Tax
{
    public static decimal CalcTax(Employee E)
    {
        return 0.08m * E.Salary;
    }
}

class MainClass
```

```
{
    static void Main()
    {
        // Create objects:
        Employee E1 = new Employee("Mingda Pan", "mpan");

        // Display results:
        E1.printEmployee();
    }
}
/*
Output:
    Name: Mingda Pan
    Alias: mpan
    Taxes: $240.00
 */
```

C# 语言规范

有关详细信息，请参阅 C# 语言规范。 该语言规范是 C# 语法和用法的权威资料。

请参阅

- this 代码样式首选项（IDE0003 和 IDE0009）
- C# 关键字
- base
- 方法

在 GitHub 上与我们协作

可以在 GitHub 上找到此内容的源，还可以在其中创建和查看问题和拉取请求。 有关详细信息，请参阅参与者指南。

.NET

.NET 反馈

.NET 是一个开放源代码项目。 选择一个链接以提供反馈：

提出文档问题

提供产品反馈

null（C# 参考）

项目 • 2024/04/02

`null` 关键字是表示不引用任何对象的空引用的文字值。 `null` 是引用类型变量的默认值。 普通值类型不能为 NULL，可为空的值类型除外。

下面的示例演示 `null` 关键字的一些行为：

C#

```
class Program
{
    class MyClass
    {
        public void MyMethod() { }
    }

    static void Main()
    {
        // Set a breakpoint here to see that mc = null.
        // However, the compiler considers it "unassigned."
        // and generates a compiler error if you try to
        // use the variable.
        MyClass mc;

        // Now the variable can be used, but...
        mc = null;

        // ... a method call on a null object raises
        // a run-time NullReferenceException.
        // Uncomment the following line to see for yourself.
        // mc.MyMethod();

        // Now mc has a value.
        mc = new MyClass();

        // You can call its method.
        mc.MyMethod();

        // Set mc to null again. The object it referenced
        // is no longer accessible and can now be garbage-collected.
        mc = null;

        // A null string is not the same as an empty string.
        string s = null;
        string t = String.Empty; // Logically the same as ""

        // Equals applied to any null object returns false.
        Console.WriteLine("t.Equals(s) is {0}", t.Equals(s));

        // Equality operator also returns false when one
```

```
        // operand is null.
        Console.WriteLine("Empty string {0} null string", s == t ? "equals":
"does not equal");

        // Returns true.
        Console.WriteLine("null == null is {0}", null == null);

        // A value type cannot be null
        // int i = null; // Compiler error!

        // Use a nullable value type instead:
        int? i = null;

        // Keep the console window open in debug mode.
    }
}
```

C# 语言规范

有关详细信息，请参阅 C# 语言规范。 该语言规范是 C# 语法和用法的权威资料。

另请参阅

- C# 关键字
- C# 类型的默认值
- Nothing (Visual Basic)

在 GitHub 上与我们协作

可以在 GitHub 上找到此内容的源，还可以在其中创建和查看问题和拉取请求。 有关详细信息，请参阅参与者指南。

.NET 反馈

.NET 是一个开放源代码项目。 选择一个链接以提供反馈：

提出文档问题

提供产品反馈

bool（C# 参考）

项目 • 2024/04/22

`bool` 类型关键字是 .NET System.Boolean 结构类型的别名，它表示一个布尔值，可为 `true` 或 `false`。

若要使用 `bool` 类型的值执行逻辑运算，请使用布尔逻辑运算符。 `bool` 类型是 比较和相等运算符的结果类型。 `bool` 表达式可以是 if、do、while 和 for 语句中以及条件运算符 ?: 中的控制条件表达式。

`bool` 类型的默认值为 `false`。

文本

可使用 `true` 和 `false` 文本来初始化 `bool` 变量或传递 `bool` 值：

```
C#
```

```
bool check = true;
Console.WriteLine(check ? "Checked" : "Not checked");  // output: Checked

Console.WriteLine(false ? "Checked" : "Not checked");  // output: Not
checked
```

三值布尔逻辑

如需支持三值逻辑（例如，在使用支持三值布尔类型的数据库时），请使用可为空 `bool?` 类型。 对于 `bool?` 操作数，预定义的 `&` 和 `|` 运算符支持三值逻辑。 有关详细信息，请参阅布尔逻辑运算符一文的可以为 null 的布尔逻辑运算符部分。

有关可为空的值类型的详细信息，请参阅可为空的值类型。

转换

C# 仅提供了两个涉及 `bool` 类型的转换。 它们是对相应的可以为空的 `bool?` 类型的隐式转换以及对 `bool?` 类型的显式转换。 但是，.NET 提供了其他方法可用来转换到 `bool` 类型从或此类型进行转换。 有关详细信息，请参阅 System.Boolean API 参考页的转换为布尔值和从布尔值转换部分。

C# 语言规范

有关详细信息，请参阅 C# 语言规范中的 bool 类型部分。

另请参阅

- 值类型
- true 和 false 运算符

在 GitHub 上与我们协作

可以在 GitHub 上找到此内容的源，还可以在其中创建和查看问题和拉取请求。 有关详细信息，请参阅参与者指南。

.NET

.NET 反馈

.NET 是一个开放源代码项目。 选择一个链接以提供反馈：

提出文档问题

提供产品反馈

default（C# 参考）

项目 • 2024/04/02

在以下上下文中，可以使用 `default` 关键字：

- 指定 switch 语句中的默认事例。
- 作为 default 默认运算符或文本生成类型的默认值。
- 作为泛型方法重写或显式接口实现上的 default 类型约束。

另请参阅

- C# 关键字

在 GitHub 上与我们协作

可以在 GitHub 上找到此内容的源，还可以在其中创建和查看问题和拉取请求。 有关详细信息，请参阅参与者指南。

.NET

.NET 反馈

.NET 是一个开放源代码项目。 选择一个链接以提供反馈：

提出文档问题

提供产品反馈

add（C# 参考）

项目 • 2023/04/07

`add` 上下文关键字用于定义一个在客户端代码订阅你的事件时调用的自定义事件访问器。 如果提供自定义 `add` 访问器，还必须提供 remove 访问器。

示例

如下示例显示一个具有自定义 `add` 和 remove 访问器的事件。 有关完整示例，请参阅如何实现接口事件。

C#

```
class Events : IDrawingObject
{
    event EventHandler PreDrawEvent;

    event EventHandler IDrawingObject.OnDraw
    {
        add => PreDrawEvent += value;
        remove => PreDrawEvent -= value;
    }
}
```

通常不需要提供自己的自定义事件访问器。 大多数情况下，使用声明事件时由编译器自动生成的访问器就足够了。

请参阅

- 事件

`get` 关键字

项目 • 2024/12/14

`get` 关键字在属性或索引器中定义*访问器*方法，它将返回属性值或索引器元素。 有关详细信息，请参阅 属性、自动实现的属性，以及 索引器。

对于属性 `get` 和 `set` 访问器除了在专用后盾字段中设置或检索值以外的其他操作，可以利用 C# 编译器对自动实现的属性的支持。 以下示例实现 `Hours` 为自动实现的属性。

C#

```
class TimePeriod3
{
    public double Hours { get; set; }
}
```

> **ⓘ 重要**
>
> 接口属性声明**或部分属性的实现声明不允许**自动实现的属性。 编译器将匹配自动实现的属性的语法解释为声明声明，而不是实现声明。

通常，`get` 访问器包含返回一个值的单个语句，如前面的示例所示。 可以将 `get` 访问器作为表达式主体成员实现。 以下示例将 `get` 和 `set` 访问器都作为 expression-bodied 成员实现。

C#

```
class TimePeriod2
{
    private double _seconds;

    public double Seconds
    {
        get => _seconds;
        set => _seconds = value;
    }
}
```

你可能会发现需要实现其中一个访问器主体。 可以使用字段支持的属性让编译器在手动编写另一个访问器时生成一个访问器。 使用 `field` C# 13 中作为预览功能添加的关键字访问编译器合成后盾字段：

C#

```
class TimePeriod4
{
    public double Hours {
        get;
        set => field = (value >= 0)
            ? value
            : throw new ArgumentOutOfRangeException(nameof(value), "The
value must not be negative");
    }
}
```

> **ⓘ 重要**
>
> 关键字 `field` 是 C# 13 中的预览功能。 必须使用 .NET 9 并将元素 `<LangVersion>` 设置为 `preview` 项目文件中，才能使用 `field` 上下文关键字。
>
> 应注意在 `field` 具有名为 `field` 字段的类中使用关键字功能。 新 `field` 关键字将隐藏属性访问器范围中命名 `field` 的字段。 可以更改变量的名称 `field` ，或使用 `@` 令牌将标识符引用 `field` 为 `@field`。 可以通过阅读关键字 `field` 来了解详细信息。

下面的示例为名为 `get` 的属性同时定义 `set` 和 `Seconds` 访问器。 它使用名为 `_seconds` 的私有字段备份属性值。

C#

```
class TimePeriod
{
    private double _seconds;

    public double Seconds
    {
        get { return _seconds; }
        set
        {
            if (value < 0)
            {
                throw new ArgumentOutOfRangeException(nameof(value), "The
value of the time period must be non-negative.");
            }
            _seconds = value;
        }
    }
}
```

C# 语言规范

有关详细信息，请参阅 C# 语言规范。 该语言规范是 C# 语法和用法的权威资料。

另请参阅

- C# 关键字
- 属性

关键字 `init` （C# 参考）

项目 • 2024/11/14

`init` 关键字在属性或索引器中定义访问器方法。 init-only 资源库仅在对象构造期间为属性或索引器元素赋值。 `init` 强制实施不可变性，因此，一旦初始化对象，将无法更改。 `init` 访问器使调用代码能够使用对象初始值设定项来设置初始值。 相比之下，只有 get setter 的自动实现属性必须通过调用构造函数来初始化。 具有 `private set` 访问器的属性可在构造后修改，但只能在类中修改。

以下代码演示 `init` 自动实现的属性中的访问器：

C#

```
class Person_InitExampleAutoProperty
{
    public int YearOfBirth { get; init; }
}
```

可能需要实现其中一个访问器来提供参数验证。 可以使用 C# 13 中作为预览功能引入的 `field` 关键字执行此操作。 关键字 `field` 访问该属性的编译器合成后盾字段。 以下示例显示了一个属性， `init` 其中访问器验证参数的范围 `value` ”

C#

```
class Person_InitExampleFieldProperty
{
    public int YearOfBirth
    {
        get;
        init
        {
            field = (YearOfBirth <= DateTime.Now.Year)
                ? value
                : throw new ArgumentOutOfRangeException(nameof(value), "Year
of birth can't be in the future");
        }
    }
}
```

ⓘ 重要

关键字 `field` 是 C# 13 中的预览功能。 必须使用 .NET 9 并将元素 `preview` 设置为 `<LangVersion>` 项目文件中，才能使用 `field` 上下文关键字。

应注意在 `field` 具有名为 `field` 字段的类中使用关键字功能。 新 `field` 关键字将隐藏属性访问器范围中命名 `field` 的字段。 可以更改变量的名称 `field` ，或使用 `@` 令牌将标识符引用 `field` 为 `@field`。 可以通过阅读关键字**的功能规范** `field` 来了解详细信息。

`init` 访问器可用作表达式主体成员。 示例：

C#

```
class Person_InitExampleExpressionBodied
{
    private int _yearOfBirth;

    public int YearOfBirth
    {
        get => _yearOfBirth;
        init => _yearOfBirth = value;
    }
}
```

以下示例为名为 `YearOfBirth` 的属性同时定义 `get` 和 `init` 访问器。 它使用名为 `_yearOfBirth` 的私有字段备份属性值。

C#

```
class Person_InitExample
{
     private int _yearOfBirth;

     public int YearOfBirth
     {
         get { return _yearOfBirth; }
         init { _yearOfBirth = value; }
     }
}
```

`init` 访问器不强制调用方设置属性。 而它允许调用方使用对象初始值设定项，同时禁止以后的修改。 可以添加 required 修饰符以强制调用方设置属性。 以下示例显示了一个以可为空的值类型作为支持字段的仅 `init` 属性。 如果调用方未初始化 `YearOfBirth` 属性，该属性的默认值 `null` 为：

C#

```
class Person_InitExampleNullability
{
    private int? _yearOfBirth;
```

```
    public int? YearOfBirth
    {
        get => _yearOfBirth;
        init => _yearOfBirth = value;
    }
}
```

若要强制调用方设置初始非 null 值，可添加 `required` 修饰符，如以下示例所示：

C#

```
class Person_InitExampleNonNull
{
    private int _yearOfBirth;

    public required int YearOfBirth
    {
        get => _yearOfBirth;
        init => _yearOfBirth = value;
    }
}
```

以下示例显示了 `private set`、read only 和 `init` 属性之间的区别。 private set 版本和 read only 版本都需要调用方使用添加的构造函数来设置 name 属性。 通过 `private set` 版本，人员可在构造实例后更改其名称。 `init` 版本不需要构造函数。 调用方可使用对象初始值设定项初始化属性：

C#

```
class PersonPrivateSet
{
    public string FirstName { get; private set; }
    public string LastName { get; private set; }
    public PersonPrivateSet(string first, string last) => (FirstName,
LastName) = (first, last);

    public void ChangeName(string first, string last) => (FirstName,
LastName) = (first, last);
}

class PersonReadOnly
{
    public string FirstName { get; }
    public string LastName { get; }
    public PersonReadOnly(string first, string last) => (FirstName,
LastName) = (first, last);
}

class PersonInit
```

```
{
    public string FirstName { get; init; }
    public string LastName { get; init; }
}
```

C#

```
PersonPrivateSet personPrivateSet = new("Bill", "Gates");
PersonReadOnly personReadOnly = new("Bill", "Gates");
PersonInit personInit = new() { FirstName = "Bill", LastName = "Gates" };
```

C# 语言规范

有关详细信息，请参阅 C# 语言规范。 该语言规范是 C# 语法和用法的权威资料。

另请参阅

- 属性

分部类型（C# 参考）

项目 • 2024/08/28

通过分部类型可以定义要拆分到多个定义中的类、结构、接口或记录。 这些多个定义可以位于同一项目内的不同文件中。 一个类型声明只包含分部成员的签名：

C#

```
partial class A
{
    int num = 0;
    void MethodA() { }
    partial void MethodC();
}
```

其他声明包含分部成员的实现：

C#

```
partial class A
{
    void MethodB() { }
    partial void MethodC() { }
}
```

分部类型的声明可能会出现在相同或多个文件中。 通常，这两个声明位于不同文件中。 在处理大型项目或自动生成的代码（如 Windows 窗体设计器或 RegEx 等源生成器提供的代码）时，可以拆分类、结构或接口类型。 分部类型可能包含分部成员。

从 C# 13 开始，你可以定义分部属性和分部索引器。 在 C# 13 之前，只能将方法定义为分部成员。

可以在声明声明或实现声明上提供文档注释。 当文档注释应用于这两种类型声明时，输出 XML 中会包含每个声明中的 XML 元素。 有关分部成员声明的规则，请参阅有关部分成员的文章。

你可以将属性应用于任一声明。 所有属性都是声明（包括重复项），并且合并在编译的输出中。

有关详细信息，请参阅分部类和方法。

C# 语言规范

有关详细信息，请参阅 C# 语言规范。 该语言规范是 C# 语法和用法的权威资料。

另请参阅

- 修饰符
- 泛型介绍

分部（方法）（C# 参考）

项目 • 2024/10/14

分部成员有一个*声明声明*，并且通常有一个*实现声明*。 *声明声明*不包括正文。 *实现声明*提供成员正文。 分部成员使类设计器能够提供可以通过源生成器等工具实施的成员挂钩。 分部类型和成员会为人类开发人员提供一种编写类型部分的方法，而工具则编写该类型的其他部分。 如果开发人员不提供可选的实现声明，则编译器可以在编译时删除声明声明。 以下条件适用于分部成员：

- 声明必须以上下文关键字 partial 开头。
- 分部类型各部分中的签名必须匹配。

不允许对构造函数、终结器、重载运算符或事件声明使用 `partial` 关键字。 在 C# 13 之前，不允许对属性或索引器使用 `partial`。

在以下情况下，分部方法不需要可实现声明：

- 没有任何辅助工具修饰符（包括默认的 private）。
- 它将返回 void。
- 它没有任何 out 参数。
- 它没有任何以下修饰符：virtual、override、sealed、new 或 extern。

任何不符合所有这些限制的成员（例如 `public virtual partial void` 方法）都必须提供实现。 分部属性和索引器必须具有实现。

以下示例演示符合上述限制的分部方法：

C#

```
partial class MyPartialClass
{
    // Declaring definition
    partial void OnSomethingHappened(string s);
}

// This part can be in a separate file.
partial class MyPartialClass
{
    // Comment out this method and the program
    // will still compile.
    partial void OnSomethingHappened(string s) =>
        Console.WriteLine($"Something happened: {s}");
}
```

分部成员还可用于与源生成器结合。 例如，可使用以下模式定义 regex：

C#

```
public partial class RegExSourceGenerator
{
    [GeneratedRegex("cat|dog", RegexOptions.IgnoreCase, "en-US")]
    private static partial Regex CatOrDogGeneratedRegex();

    private static void EvaluateText(string text)
    {
        if (CatOrDogGeneratedRegex().IsMatch(text))
        {
            // Take action with matching text
        }
    }
}
```

上述示例演示了必须具有实现声明的部分方法。 作为生成的一部分，正则表达式源生成器将会创建实现声明。

以下示例演示了类的声明声明和实现声明。 由于方法的返回类型不是`void`（它是`string`），并且其访问权限是`public`，因此，该方法必须具有实现声明：

C#

```
// Declaring declaration
public partial class PartialExamples
{
    /// <summary>
    /// Gets or sets the number of elements that the List can contain.
    /// </summary>
    public partial int Capacity { get; set; }

    /// <summary>
    /// Gets or sets the element at the specified index.
    /// </summary>
    /// <param name="index">The index</param>
    /// <returns>The string stored at that index</returns>
    public partial string this[int index] { get; set; }

    public partial string? TryGetAt(int index);
}

public partial class PartialExamples
{
    private List<string> _items = [
        "one",
        "two",
        "three",
        "four",
        "five"
        ];
```

```
    // Implementing declaration

    /// <summary>
    /// Gets or sets the number of elements that the List can contain.
    /// </summary>
    /// <remarks>
    /// If the value is less than the current capacity, the list will shrink
to the
    /// new value. If the value is negative, the list isn't modified.
    /// </remarks>
    public partial int Capacity
    {
        get => _items.Count;
        set
        {
            if ((value != _items.Count) && (value >= 0))
            {
                _items.Capacity = value;
            }
        }
    }

    public partial string this[int index]
    {
        get => _items[index];
        set => _items[index] = value;
    }

    /// <summary>
    /// Gets the element at the specified index.
    /// </summary>
    /// <param name="index">The index</param>
    /// <returns>The string stored at that index, or null if out of
bounds</returns>
    public partial string? TryGetAt(int index)
    {
        if (index < _items.Count)
        {
            return _items[index];
        }
        return null;
    }
}
```

上述示例演示了关于如何组合这两个声明的规则：

- *签名匹配*：一般情况下，声明声明和实现声明的签名必须匹配。 这包括方法、属性、索引器和单个访问器的辅助功能修饰符。 它包括所有参数的参数类型和 ref 类型修饰符。 返回类型和任何 ref 类型修饰符都必须匹配。 元组成员名称必须匹配。 但是，某些规则很灵活：
  - 声明声明和实现声明可以具有不同的可为 null 的注释设置。 这意味着，一个可以是 *可为 null 的未知声明*，另一个是 *已启用可为 null 的声明*。

    - 不涉及*未知 Null 性*的 Null 性差异会生成警告。
    - 默认参数值不需要匹配。 如果方法或索引器的实现声明声明默认参数值，编译器会发出警告。
    - 当参数名称不匹配时，编译器会发出警告。 发出的 IL 包含声明声明的参数名称。
- *文档注释：* 可以包括任一声明中的文档注释。 如果声明声明和实现声明都包括文档注释，则包括实现声明中的注释。 在上述示例中，文档注释包括：
    - 对于 `Capacity` 属性，注释取自实现声明。 当两个声明都有 `///` 注释时，会使用实现声明注释。
    - 对于索引器，注释取自声明声明。 实现声明不包括任何 `///` 注释。
    - 对于 `TryGetAt`，注释取自实现声明。 声明声明不包括任何 `///` 注释。
    - 生成的 XML 具有所有 `public` 成员的文档注释。
- 大多数*属性声明*会组合在一起。 但是，所有调用方信息属性都使用 `AllowMultiple=false` 来定义。 编译器可识别声明声明上的任何调用方信息属性。 系统会忽略实现声明上的所有调用方信息属性。 如果在实现声明上添加调用方信息属性，则编译器将发出警告。

另请参阅

- 分部类型

remove（C# 参考）

项目 • 2023/04/07

`remove` 上下文关键字用于定义自定义事件访问器，当客户端代码订阅你的事件时将调用该访问器。 如果提供自定义 `remove` 访问器，还必须提供 add 访问器。

示例

以下示例显示具有自定义 add 和 `remove` 访问器的事件。 有关完整示例，请参阅如何实现接口事件。

C#

```
class Events : IDrawingObject
{
    event EventHandler PreDrawEvent;

    event EventHandler IDrawingObject.OnDraw
    {
        add => PreDrawEvent += value;
        remove => PreDrawEvent -= value;
    }
}
```

通常不需要提供自己的自定义事件访问器。 大多数情况下，使用声明事件时由编译器自动生成的访问器就足够了。

请参阅

- 事件

required 修饰符（C# 参考）

项目 • 2023/04/07

`required` 修饰符表示它所应用的字段或属性必须由对象初始值设定项进行初始化。 用于初始化该类型新实例的任何表达式都必须初始化所有必需的成员。 `required` 修饰符从 C# 11 开始可用。 `required` 修饰符使开发人员能够创建必须正确初始化属性或字段的类型，但仍允许使用对象初始值设定项进行初始化。 多个规则可确保此行为：

- `required` 修饰符可应用于在 `struct` 中声明的字段和属性，以及 `class` 类型，包括 `record` 和 `record struct` 类型。 `required` 修饰符不能应用于 `interface` 的成员。
- 显式接口实现不能标记为 `required`。 不能在对象初始值设定项中设置它们。
- 必须初始化必需的成员，但可将其初始化为 `null`。 如果该类型是不可为 null 的引用类型，则在将成员初始化为 `null` 时编译器会发出警告。 如果成员根本没有初始化，编译器会发出错误。
- 必需的成员必须至少与其包含类型一样可见。 例如，`public` 类不能包含 `protected` 的 `required` 字段。 此外，必需的属性必须具有至少与其包含类型一样可见的资源库（`set` 或 `init` 访问器）。 不可访问的成员不能由创建实例的代码设置。
- 派生类不能隐藏在基类中声明的 `required` 成员。 隐藏必需的成员可防止调用方为其使用对象初始值设定项。 此外，重写必需属性的派生类型必须包含 `required` 修饰符。 派生类型无法删除 `required` 状态。 派生类型可以在重写属性时添加 `required` 修饰符。
- 当类型参数包含 `new()` 约束时，不能将具有任何 `required` 成员的类型用作类型参数。 编译器无法强制在泛型代码中初始化所有必需的成员。
- 不允许对记录上的位置参数声明使用 `required` 修饰符。 可以为包含 `required` 修饰符的位置属性添加显式声明。

某些类型（如位置记录）使用主构造函数初始化位置属性。 如果其中任一属性包括 `required` 修饰符，则主构造函数将添加 SetsRequiredMembers 属性。 这表示主构造函数初始化所有必需的成员。 可以使用 System.Diagnostics.CodeAnalysis.SetsRequiredMembersAttribute 特性编写自己的构造函数。 但是，编译器不会验证这些构造函数是否初始化所有必需的成员。 该特性会转而向编译器断言构造函数会初始化所有必需的成员。 `SetsRequiredMembers` 特性会将这些规则添加到构造函数：

- 链接到另一个使用 `SetsRequiredMembers` 属性批注的构造函数（`this()` 或 `base()`）的构造函数还必须包含 `SetsRequiredMembers` 属性。 这可确保调用方能够正确使用所有适当的构造函数。
- 如果任何成员为 `required`，则为 `record` 类型生成的复制构造函数应用 `SetsRequiredMembers` 属性。

> **⚠ 警告**
>
> `SetsRequiredMembers` 禁用编译器检查所有 `required` 成员在创建对象时是否已初始化。 请谨慎使用。

下面的代码显示了对 `FirstName` 和 `LastName` 属性使用 `required` 修饰符的类层次结构：

C#

```
public class Person
{
    public Person() { }

    [SetsRequiredMembers]
    public Person(string firstName, string lastName) =>
        (FirstName, LastName) = (firstName, lastName);

    public required string FirstName { get; init; }
    public required string LastName { get; init; }

    public int? Age { get; set; }
}

public class Student : Person
{
    public Student() : base()
    {
    }

    [SetsRequiredMembers]
    public Student(string firstName, string lastName) :
        base(firstName, lastName)
    {
    }

    public double GPA { get; set; }
}
```

有关所需成员的详细信息，请参阅 C#11 - 必需成员功能规范。

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

Open a documentation issue

more information, see our contributor guide.

Provide product feedback

set 关键字 （C# 参考）

项目 • 2024/11/14

`set` 关键字在属性或索引器中定义*访问器*，它会向属性或索引器元素分配值。 有关详细信息和示例，请参阅 “属性”、“ 自动实现的属性”和 “索引器”。

`get`对于属性和`set`访问器除了在专用后盾字段中设置或检索值以外的其他任何操作，可以使用自动实现的属性。 以下示例实现 `Hours` 为自动实现的属性。

C#

```
class TimePeriod3
{
    public double Hours { get; set; }
}
```

> **ⓘ 重要**
>
> 接口属性声明**或部分属性的实现声明不允许**自动实现的属性。 编译器将匹配自动实现的属性的语法解释为声明声明，而不是实现声明。

你可能会发现需要实现其中一个访问器主体。 在 `field` C# 13 中添加为预览功能的关键字声明字段支持的属性。 可以使用字段支持的属性让编译器在手动编写另一个访问器时生成一个访问器。 使用 `field` 关键字访问编译器合成后盾字段：

C#

```
class TimePeriod4
{
    public double Hours {
        get;
        set => field = (value >= 0)
            ? value
            : throw new ArgumentOutOfRangeException(nameof(value), "The
value must not be negative");
    }
}
```

> **ⓘ 重要**
>
> 关键字 `field` 是 C# 13 中的预览功能。 必须使用 .NET 9 并将元素`preview`设置为`<LangVersion>`项目文件中，才能使用`field`上下文关键字。

应注意在 `field` 具有名为 `field` 字段的类中使用关键字功能。 新 `field` 关键字将隐藏属性访问器范围中命名 `field` 的字段。 可以更改变量的名称 `field` ，或使用 `@` 令牌将标识符引用 `field` 为 `@field`。 可以通过阅读关键字**的功能规范** `field` 来了解详细信息。

通常，`set` 访问器包含分配一个值的单个语句，如前面的示例所示。 可以将 `set` 访问器作为表达式主体成员实现。 下面的示例将 `get` 和 `set` 访问器都作为表达式主体成员实现。

C#

```
class TimePeriod2
{
    private double _seconds;

    public double Seconds
    {
        get => _seconds;
        set => _seconds = value;
    }
}
```

下面的示例为名为 `Seconds` 的属性同时定义 `get` 和 `set` 访问器。 它使用名为 `_seconds` 的私有字段备份属性值。

C#

```
class TimePeriod
{
    private double _seconds;

    public double Seconds
    {
        get { return _seconds; }
        set
        {
            if (value < 0)
            {
                throw new ArgumentOutOfRangeException(nameof(value), "The
value of the time period must be non-negative.");
            }
            _seconds = value;
        }
    }
}
```

C# 语言规范

有关详细信息，请参阅 C# 语言规范。 该语言规范是 C# 语法和用法的权威资料。

另请参阅

- 属性

when（C# 参考）

项目 • 2024/04/02

使用上下文关键字 `when` 在以下上下文中指定筛选条件：

- 在 try-catch 或 try-catch-finally 语句的 catch 子句中。
- 作为 switch 语句中的 case guard。
- 作为 switch 表达式中的 case guard。

catch 子句中的 `when`

可在 catch 子句中使用 `when` 关键字来指定一个条件，此条件必须为 true，才能执行特定异常的处理程序。 语法为：

C#

```
catch (ExceptionType [e]) when (expr)
```

其中，*expr* 是一个表达式，其计算结果为布尔值。 如果该表达式返回 `true`，则执行异常处理程序；如果返回 `false`，则不执行。

以下示例使用 `when` 关键字有条件地执行 HttpRequestException 的处理程序，具体取决于异常消息的文本内容。

C#

```
using System;
using System.Net.Http;
using System.Threading.Tasks;

class Program
{
    static void Main()
    {
        Console.WriteLine(MakeRequest().Result);
    }

    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;
        }
    }
}
```

另请参阅

- C# 关键字

在 GitHub 上与我们协作

可以在 GitHub 上找到此内容的源，还可以在其中创建和查看问题和拉取请求。 有关详细信息，请参阅参与者指南。

.NET

.NET 反馈

.NET 是一个开放源代码项目。 选择一个链接以提供反馈：

提出文档问题

提供产品反馈

`field` - 字段支持的属性声明

项目 • 2024/11/14

> **ⓘ 重要**
>
> 关键字 `field` 是 C# 13 中的预览功能。 必须使用 .NET 9 并将元素 `preview` 设置为 `<LangVersion>` 项目文件中，才能使用 `field` 上下文关键字。
>
> 应注意在 `field` 具有名为 `field` 字段的类中使用关键字功能。 新 `field` 关键字将隐藏属性访问器范围中命名 `field` 的字段。 可以更改变量的名称 `field` ，或使用 `@` 令牌将标识符引用 `field` 为 `@field`。 可以通过阅读关键字**的功能规范** `field` 来了解详细信息。

作为 C# 13 中的预览功能添加的上下文关键字 `field` 可用于属性访问器，以访问属性的编译器合成后盾字段。 使用此语法可以定义或 `set` 访问器的主体 `get`，并允许编译器在自动实现的属性中生成另一个访问器。

添加 `field` 上下文关键字提供了一个流畅的路径，用于向自动实现的属性添加范围检查等优势。 以下示例显示了这种做法：

C#

```
class TimePeriod4
{
    public double Hours {
        get;
        set => field = (value >= 0)
            ? value
            : throw new ArgumentOutOfRangeException(nameof(value), "The
value must not be negative");
    }
}
```

可以将该 `Hours` 属性实现为自动实现的属性。 然后，你发现你想要防范负值。 `field` 在访问器中使用 `set` 并提供范围检查。 无需手动声明后备字段并为访问器提供主体 `get` 。

有关详细信息，请参阅属性和索引器这两篇文章。

C# 语言规范

有关详细信息，请参阅 C# 语言规范。 该语言规范是 C# 语法和用法的权威资料。

`value` 隐式参数

项目 • 2025/01/01

隐式参数用于属性和`value` `set` 它是方法的输入参数。 关键字 `value` 引用客户端代码尝试分配给属性或索引器的值。 在以下示例中，`TimePeriod2` 有一个名为 `Seconds` 的属性，该属性使用 `value` 参数向支持字段 `_seconds` 分配新字符串。 从客户端代码的角度来看，该操作写作一个简单的赋值语句。

C#

```
class TimePeriod2
{
    private double _seconds;

    public double Seconds
    {
        get => _seconds;
        set => _seconds = value;
    }
}
```

有关详细信息，请参阅属性和索引器这两篇文章。

C# 语言规范

有关详细信息，请参阅 C# 语言规范。 该语言规范是 C# 语法和用法的权威资料。

查询关键字（C# 参考）

项目 • 2023/07/20

本部分包含在查询表达式中使用的上下文关键字。

在本节中

子句	说明
from	指定数据源和范围变量（类似于迭代变量）。
where	基于由逻辑 AND 和 OR 运算符（`&&` 或 `||`）分隔的一个或多个布尔表达式筛选源元素。
select	指定执行查询时，所返回序列中元素的类型和形状。
group	根据指定的密钥值对查询结果分组。
into	提供可作为对 join、group 或 select 子句结果引用的标识符。
orderby	根据元素类型的默认比较器对查询结果进行升序或降序排序。
join	基于两个指定匹配条件间的相等比较而联接两个数据源。
let	引入范围变量，在查询表达式中存储子表达式结果。
in	join 子句中的上下文关键字。
on	join 子句中的上下文关键字。
equals	join 子句中的上下文关键字。
by	group 子句中的上下文关键字。
ascending	orderby 子句中的上下文关键字。
descending	orderby 子句中的上下文关键字。

请参阅

- C# 关键字
- C# 中的 LINQ

from 子句（C# 参考）

项目 • 2023/04/07

查询表达式必须以 `from` 子句开头。 此外，查询表达式可包含也以 `from` 子句开头的子查询。 `from` 子句指定下列各项：

- 将在其上运行查询或子查询的数据源。
- 表示源序列中每个元素的本地范围变量。

范围变量和数据源已强类型化。 `from` 子句中引用的数据源必须具有 IEnumerable、IEnumerable<T> 类型之一，或 IQueryable<T> 等派生类型。

在下面的示例中，`numbers` 是数据源，`num` 是范围变量。 请注意，这两个变量都已强类型化，即使使用 var 关键字也是如此。

C#

```
class LowNums
{
    static void Main()
    {
        // A simple data source.
        int[] numbers = { 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 };

        // Create the query.
        // lowNums is an IEnumerable<int>
        var lowNums = from num in numbers
            where num < 5
            select num;

        // Execute the query.
        foreach (int i in lowNums)
        {
            Console.Write(i + " ");
        }
    }
}
// Output: 4 1 3 2 0
```

范围变量

数据源实现 IEnumerable<T> 时，编译器推断范围变量的类型。 例如，如果源具有 `IEnumerable<Customer>` 类型，则范围变量会被推断为 `Customer`。 仅在以下情况下必须

显式指定类型：源是 ArrayList 等非泛型 `IEnumerable` 类型时。 有关详细信息，请参阅如何使用 LINQ 查询 ArrayList。

在以上示例中，`num` 推断为 `int` 类型。 由于强类型化了范围变量，所以可以在其上调用方法，或将其用于其他操作中。 例如，不再编写 `select num`，而编写 `select num.ToString()`，使查询表达式返回字符串序列，而不是整数序列。 或者可以编写 `select num + 10`，使表达式返回序列 14、11、13、12、10。 有关详细信息，请参阅 select 子句。

范围变量与 foreach 语句中的迭代变量类似，一项非常重要的区别除外：范围变量从不实际存储来自源的数据。 它只是一种语法上的便利，让查询能够描述执行查询时将发生的情况。 有关详细信息，请参阅 LINQ 查询简介 (C#)。

复合 from 子句

在某些情况下，源序列中的每个元素可能本身就是一个序列，或者包含一个序列。 例如，数据源可能是 `IEnumerable<Student>`，其中序列中的每个学生对象都包含测试分数的列表。 要访问每个 `Student` 元素的内部列表，可以使用复合 `from` 子句。 这种方法类似于使用嵌套的 foreach 语句。 可以向任一 `from`子句添加 where 或 orderby 子句筛选结果。 下面的示例演示 `Student` 对象的序列，其中每个对象都包含一个整数内部 `List`，表示测验分数。 访问内部列表可使用复合 `from` 子句。 如有必要，可以在这两个 `from` 子句间插入子句。

C#

```
class CompoundFrom
{
    // The element type of the data source.
    public class Student
    {
        public string LastName { get; set; }
        public List<int> Scores {get; set;}
    }

    static void Main()
    {

        // Use a collection initializer to create the data source. Note that
        // each element in the list contains an inner sequence of scores.
        List<Student> students = new List<Student>
        {
           new Student {LastName="Omelchenko", Scores= new List<int> {97,
72, 81, 60}},
           new Student {LastName="O'Donnell", Scores= new List<int> {75, 84,
91, 39}},
           new Student {LastName="Mortensen", Scores= new List<int> {88, 94,
```

```
65, 85}},
            new Student {LastName="Garcia", Scores= new List<int> {97, 89,
85, 82}},
            new Student {LastName="Beebe", Scores= new List<int> {35, 72, 91,
70}}
        };

        // Use a compound from to access the inner sequence within each
element.
        // Note the similarity to a nested foreach statement.
        var scoreQuery = from student in students
                         from score in student.Scores
                            where score > 90
                            select new { Last = student.LastName, score };

        // Execute the queries.
        Console.WriteLine("scoreQuery:");
        // Rest the mouse pointer on scoreQuery in the following line to
        // see its type. The type is IEnumerable<'a>, where 'a is an
        // anonymous type defined as new {string Last, int score}. That is,
        // each instance of this anonymous type has two members, a string
        // (Last) and an int (score).
        foreach (var student in scoreQuery)
        {
            Console.WriteLine("{0} Score: {1}", student.Last,
student.score);
        }

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
/*
scoreQuery:
Omelchenko Score: 97
O'Donnell Score: 91
Mortensen Score: 94
Garcia Score: 97
Beebe Score: 91
*/
```

使用多个 from 子句执行联接

复合 `from` 子句用于访问单个数据源中的内部集合。 但是，查询也可以包含多个 `from` 子句，这些子句从独立的数据源生成补充查询。 通过此方法，可以执行使用 join 子句 无法实现的某类联接操作。

下面的示例演示两个 `from` 子句如何用于形成两个数据源的完全交叉联接。

```
C#
```

```
class CompoundFrom2
{
    static void Main()
    {
        char[] upperCase = { 'A', 'B', 'C' };
        char[] lowerCase = { 'x', 'y', 'z' };

        // The type of joinQuery1 is IEnumerable<'a>, where 'a
        // indicates an anonymous type. This anonymous type has two
        // members, upper and lower, both of type char.
        var joinQuery1 =
            from upper in upperCase
            from lower in lowerCase
            select new { upper, lower };

        // The type of joinQuery2 is IEnumerable<'a>, where 'a
        // indicates an anonymous type. This anonymous type has two
        // members, upper and lower, both of type char.
        var joinQuery2 =
            from lower in lowerCase
            where lower != 'x'
            from upper in upperCase
            select new { lower, upper };

        // Execute the queries.
        Console.WriteLine("Cross join:");
        // Rest the mouse pointer on joinQuery1 to verify its type.
        foreach (var pair in joinQuery1)
        {
            Console.WriteLine("{0} is matched to {1}", pair.upper,
pair.lower);
        }

        Console.WriteLine("Filtered non-equijoin:");
        // Rest the mouse pointer over joinQuery2 to verify its type.
        foreach (var pair in joinQuery2)
        {
            Console.WriteLine("{0} is matched to {1}", pair.lower,
pair.upper);
        }

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
/* Output:
        Cross join:
        A is matched to x
        A is matched to y
        A is matched to z
        B is matched to x
        B is matched to y
        B is matched to z
```

```
        C is matched to x
        C is matched to y
        C is matched to z
        Filtered non-equijoin:
        y is matched to A
        y is matched to B
        y is matched to C
        z is matched to A
        z is matched to B
        z is matched to C
        */
```

若要详细了解使用多个 `from` 子句的联接操作，请参阅执行左外部联结。

另请参阅

- 查询关键字 (LINQ)
- 语言集成查询 (LINQ)

where 子句（C# 参考）

项目 • 2023/07/20

`where` 子句用在查询表达式中，用于指定将在查询表达式中返回数据源中的哪些元素。 它将一个布尔条件（谓词）应用于每个源元素（由范围变量引用），并返回满足指定条件的元素。 一个查询表达式可以包含多个 `where` 子句，一个子句可以包含多个谓词子表达式。

示例 1

在下面的示例中，`where` 子句筛选出除小于五的数字外的所有数字。 如果删除 `where` 子句，则会返回数据源中的所有数字。 表达式 `num < 5` 是应用于每个元素的谓词。

C#

```
class WhereSample
{
    static void Main()
    {
        // Simple data source. Arrays support IEnumerable<T>.
        int[] numbers = { 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 };

        // Simple query with one predicate in where clause.
        var queryLowNums =
            from num in numbers
            where num < 5
            select num;

        // Execute the query.
        foreach (var s in queryLowNums)
        {
            Console.Write(s.ToString() + " ");
        }
    }
}
//Output: 4 1 3 2 0
```

示例 2

在单个 `where` 子句中，可使用 && 和 || 运算符根据需要指定任意多个谓词。 在下面的示例中，查询将指定两个谓词，以便只选择小于五的偶数。

C#

```
class WhereSample2
{
    static void Main()
    {
        // Data source.
        int[] numbers = { 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 };

        // Create the query with two predicates in where clause.
        var queryLowNums2 =
            from num in numbers
            where num < 5 && num % 2 == 0
            select num;

        // Execute the query
        foreach (var s in queryLowNums2)
        {
            Console.Write(s.ToString() + " ");
        }
        Console.WriteLine();

        // Create the query with two where clause.
        var queryLowNums3 =
            from num in numbers
            where num < 5
            where num % 2 == 0
            select num;

        // Execute the query
        foreach (var s in queryLowNums3)
        {
            Console.Write(s.ToString() + " ");
        }
    }
}
// Output:
// 4 2 0
// 4 2 0
```

示例 3

一个 `where` 子句可以包含一个或多个返回布尔值的方法。 在下面的示例中，`where` 子句使用一种方法来确定范围变量的当前值是偶数还是奇数。

C#

```
class WhereSample3
{
    static void Main()
    {
        // Data source
```

```
        int[] numbers = { 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 };

        // Create the query with a method call in the where clause.
        // Note: This won't work in LINQ to SQL unless you have a
        // stored procedure that is mapped to a method by this name.
        var queryEvenNums =
            from num in numbers
            where IsEven(num)
            select num;

         // Execute the query.
        foreach (var s in queryEvenNums)
        {
            Console.Write(s.ToString() + " ");
        }
    }

    // Method may be instance method or static method.
    static bool IsEven(int i)
    {
        return i % 2 == 0;
    }
}
//Output: 4 8 6 2 0
```

备注

`where` 子句是一种筛选机制。 除了不能是第一个或最后一个子句外，它几乎可以放在查询表达式中的任何位置。 `where` 子句可以出现在 group 子句的前面或后面，具体取决于时必须在对源元素进行分组之前还是之后来筛选源元素。

如果指定的谓词对于数据源中的元素无效，则会发生编译时错误。 这是 LINQ 提供的强类型检查的一个优点。

在编译时，`where` 关键字将转换为对 Where 标准查询运算符方法的调用。

另请参阅

- 查询关键字 (LINQ)
- from 子句
- select 子句
- 筛选数据
- C# 中的 LINQ

select 子句（C# 参考）

项目 • 2024/04/02

在查询表达式中，`select` 子句指定在执行查询时产生的值的类型。 根据计算所有以前的子句以及根据 `select` 子句本身的所有表达式得出结果。 查询表达式必须以 `select` 子句或 group 子句结尾。

以下示例演示查询表达式中的简单的 `select` 子句。

C#

```
class SelectSample1
{
    static void Main()
    {
        //Create the data source
        List<int> Scores = [97, 92, 81, 60];

        // Create the query.
        IEnumerable<int> queryHighScores =
            from score in Scores
            where score > 80
            select score;

        // Execute the query.
        foreach (int i in queryHighScores)
        {
            Console.Write(i + " ");
        }
    }
}
//Output: 97 92 81
```

`select` 子句生成的序列的类型确定查询变量 `queryHighScores` 的类型。 在最简单的情况下，`select` 子句仅指定范围变量。 这将导致返回的序列包含与数据源类型相同的元素。 有关详细信息，请参阅 LINQ 查询操作中的类型关系。 但是，`select` 子句还提供了强大的机制，用于将源数据转换（或投影）为新类型。 有关详细信息，请参阅使用 LINQ 进行数据转换 (C#)。

示例

以下示例展示 `select` 子句可能采用的所有不同窗体。 在每个查询中，请注意 `select` 子句和查询变量（`studentQuery1`、`studentQuery2` 等）类型之间的关系。

C#

```
class SelectSample2
{
    // Define some classes
    public class Student
    {
        public required string First { get; init; }
        public required string Last { get; init; }
        public required int ID { get; init; }
        public required List<int> Scores;
        public ContactInfo? GetContactInfo(SelectSample2 app, int id)
        {
            ContactInfo? cInfo =
                (from ci in app.contactList
                where ci.ID == id
                select ci)
                .FirstOrDefault();

            return cInfo;
        }

        public override string ToString() => $"{First} {Last}:{ID}";
    }

    public class ContactInfo
    {
        public required int ID { get; init; }
        public required string Email { get; init; }
        public required string Phone { get; init; }
        public override string ToString() => $"{Email},{Phone}";
    }

    public class ScoreInfo
    {
        public double Average { get; init; }
        public int ID { get; init; }
    }

    // The primary data source
    List<Student> students =
    [
         new Student {First="Svetlana", Last="Omelchenko", ID=111, Scores=
new List<int>() {97, 92, 81, 60}},
         new Student {First="Claire", Last="O'Donnell", ID=112, Scores= new
List<int>() {75, 84, 91, 39}},
         new Student {First="Sven", Last="Mortensen", ID=113, Scores= new
List<int>() {88, 94, 65, 91}},
         new Student {First="Cesar", Last="Garcia", ID=114, Scores= new
List<int>() {97, 89, 85, 82}},
    ];

    // Separate data source for contact info.
    List<ContactInfo> contactList =
    [
        new ContactInfo {ID=111, Email="SvetlanO@Contoso.com", Phone="206-
```

```
555-0108"},
        new ContactInfo {ID=112, Email="ClaireO@Contoso.com", Phone="206-
555-0298"},
        new ContactInfo {ID=113, Email="SvenMort@Contoso.com", Phone="206-
555-1130"},
        new ContactInfo {ID=114, Email="CesarGar@Contoso.com", Phone="206-
555-0521"}
    ];

    static void Main(string[] args)
    {
        SelectSample2 app = new SelectSample2();

        // Produce a filtered sequence of unmodified Students.
        IEnumerable<Student> studentQuery1 =
            from student in app.students
            where student.ID > 111
            select student;

        Console.WriteLine("Query1: select range_variable");
        foreach (Student s in studentQuery1)
        {
            Console.WriteLine(s.ToString());
        }

        // Produce a filtered sequence of elements that contain
        // only one property of each Student.
        IEnumerable<String> studentQuery2 =
            from student in app.students
            where student.ID > 111
            select student.Last;

        Console.WriteLine("\r\n studentQuery2: select
range_variable.Property");
        foreach (string s in studentQuery2)
        {
            Console.WriteLine(s);
        }

        // Produce a filtered sequence of objects created by
        // a method call on each Student.
        IEnumerable<ContactInfo> studentQuery3 =
            from student in app.students
            where student.ID > 111
            select student.GetContactInfo(app, student.ID);

        Console.WriteLine("\r\n studentQuery3: select
range_variable.Method");
        foreach (ContactInfo ci in studentQuery3)
        {
            Console.WriteLine(ci.ToString());
        }

        // Produce a filtered sequence of ints from
        // the internal array inside each Student.
```

```
        IEnumerable<int> studentQuery4 =
            from student in app.students
            where student.ID > 111
            select student.Scores[0];

        Console.WriteLine("\r\n studentQuery4: select
range_variable[index]");
        foreach (int i in studentQuery4)
        {
            Console.WriteLine("First score = {0}", i);
        }

        // Produce a filtered sequence of doubles
        // that are the result of an expression.
        IEnumerable<double> studentQuery5 =
            from student in app.students
            where student.ID > 111
            select student.Scores[0] * 1.1;

        Console.WriteLine("\r\n studentQuery5: select expression");
        foreach (double d in studentQuery5)
        {
            Console.WriteLine("Adjusted first score = {0}", d);
        }

        // Produce a filtered sequence of doubles that are
        // the result of a method call.
        IEnumerable<double> studentQuery6 =
            from student in app.students
            where student.ID > 111
            select student.Scores.Average();

        Console.WriteLine("\r\n studentQuery6: select expression2");
        foreach (double d in studentQuery6)
        {
            Console.WriteLine("Average = {0}", d);
        }

        // Produce a filtered sequence of anonymous types
        // that contain only two properties from each Student.
        var studentQuery7 =
            from student in app.students
            where student.ID > 111
            select new { student.First, student.Last };

        Console.WriteLine("\r\n studentQuery7: select new anonymous type");
        foreach (var item in studentQuery7)
        {
            Console.WriteLine("{0}, {1}", item.Last, item.First);
        }

        // Produce a filtered sequence of named objects that contain
        // a method return value and a property from each Student.
        // Use named types if you need to pass the query variable
        // across a method boundary.
```

```
        IEnumerable<ScoreInfo> studentQuery8 =
            from student in app.students
            where student.ID > 111
            select new ScoreInfo
            {
                Average = student.Scores.Average(),
                ID = student.ID
            };

        Console.WriteLine("\r\n studentQuery8: select new named type");
        foreach (ScoreInfo si in studentQuery8)
        {
            Console.WriteLine("ID = {0}, Average = {1}", si.ID, si.Average);
        }

        // Produce a filtered sequence of students who appear on a contact
list
        // and whose average is greater than 85.
        IEnumerable<ContactInfo> studentQuery9 =
            from student in app.students
            where student.Scores.Average() > 85
            join ci in app.contactList on student.ID equals ci.ID
            select ci;

        Console.WriteLine("\r\n studentQuery9: select result of join
clause");
        foreach (ContactInfo ci in studentQuery9)
        {
            Console.WriteLine("ID = {0}, Email = {1}", ci.ID, ci.Email);
        }
    }
}
/* Output
    Query1: select range_variable
    Claire O'Donnell:112
    Sven Mortensen:113
    Cesar Garcia:114

    studentQuery2: select range_variable.Property
    O'Donnell
    Mortensen
    Garcia

    studentQuery3: select range_variable.Method
    ClaireO@Contoso.com,206-555-0298
    SvenMort@Contoso.com,206-555-1130
    CesarGar@Contoso.com,206-555-0521

    studentQuery4: select range_variable[index]
    First score = 75
    First score = 88
    First score = 97

    studentQuery5: select expression
    Adjusted first score = 82.5
```

```
    Adjusted first score = 96.8
    Adjusted first score = 106.7

    studentQuery6: select expression2
    Average = 72.25
    Average = 84.5
    Average = 88.25

    studentQuery7: select new anonymous type
    O'Donnell, Claire
    Mortensen, Sven
    Garcia, Cesar

    studentQuery8: select new named type
    ID = 112, Average = 72.25
    ID = 113, Average = 84.5
    ID = 114, Average = 88.25

    studentQuery9: select result of join clause
    ID = 114, Email = CesarGar@Contoso.com
*/
```

如前面示例中的 `studentQuery8` 所示，有时可能想要返回序列的元素仅包含一部分源元素属性。 通过让返回序列尽可能变小，可以减少内存需求并提高执行查询的速度。 在 `select` 子句中创建匿名类型并使用对象初始值设定项通过源元素中的相应属性初始化该类型可以完成此操作。 有关如何执行此操作的示例，请参阅对象和集合初始值设定项。

备注

在编译时，`select` 子句被转换为 Select 标准查询运算符的方法调用。

另请参阅

- 查询关键字 (LINQ)
- from 子句
- 分部（方法）（C# 参考）
- 匿名类型
- C# 中的 LINQ

在 GitHub 上与我们协作

可以在 GitHub 上找到此内容的源，还可以在其中创建和查看问

.NET 反馈

.NET 是一个开放源代码项目。 选择一个链接以提供反馈：

题和拉取请求。 有关详细信息，请参阅参与者指南。

提出文档问题

提供产品反馈

group 子句（C# 参考）

项目 • 2024/06/21

`group` 子句返回一个 IGrouping<TKey,TElement> 对象序列，这些对象包含零个或更多与该组的键值匹配的项。 例如，可以按照每个字符串中的第一个字母对字符串序列进行分组。 在这种情况下，第一个字母就是键，类型为 char，并且存储在每个 IGrouping<TKey,TElement> 对象的 `Key` 属性中。 编译器可推断键的类型。

可以用 `group` 子句结束查询表达式，如以下示例所示：

C#

```
// Query variable is an IEnumerable<IGrouping<char, Student>>
var studentQuery1 =
    from student in students
    group student by student.Last[0];
```

如果要对每个组执行附加查询操作，可使用上下文关键字 into 指定一个临时标识符。 使用 `into` 时，必须继续编写该查询，并最终使用一个`select` 语句或另一个 `group` 子句结束该查询，如以下代码摘录所示：

C#

```
// Group students by the first letter of their last name
// Query variable is an IEnumerable<IGrouping<char, Student>>
var studentQuery2 =
    from student in students
    group student by student.Last[0] into g
    orderby g.Key
    select g;
```

对于含有和不含 `into` 的 `group`，本文中的“示例”部分提供有关其用法的更完整示例。

枚举查询分组的结果

由于 `group` 查询产生的 IGrouping<TKey,TElement> 对象实质上是一个由列表组成的列表，因此必须使用嵌套的 foreach 循环来访问每一组中的各个项。 外部循环用于循环访问组键，内部循环用于循环访问组本身包含的每个项。 组可能具有键，但没有元素。 下面的 `foreach` 循环执行上述代码示例中的查询：

C#

```
// Iterate group items with a nested foreach. This IGrouping encapsulates
// a sequence of Student objects, and a Key of type char.
// For convenience, var can also be used in the foreach statement.
foreach (IGrouping<char, Student> studentGroup in studentQuery2)
{
     Console.WriteLine(studentGroup.Key);
     // Explicit type for student could also be used here.
     foreach (var student in studentGroup)
     {
         Console.WriteLine("   {0}, {1}", student.Last, student.First);
     }
 }
```

键类型

组键可以是任何类型，如字符串、内置数值类型、用户定义的命名类型或匿名类型。

按字符串分组

上述代码示例使用 `char`。 可轻松改为指定字符串键，如完整的姓氏：

C#

```
// Same as previous example except we use the entire last name as a key.
// Query variable is an IEnumerable<IGrouping<string, Student>>
var studentQuery3 =
    from student in students
    group student by student.Last;
```

按布尔值分组

下面的示例演示使用布尔值作为键将结果划分成两个组。 请注意，该值由 `group` 子句中的子表达式生成。

C#

```
class GroupSample1
{
    // The element type of the data source.
    public class Student
    {
        public required string First { get; init; }
        public required string Last { get; init; }
        public required int ID { get; init; }
        public required List<int> Scores;
    }
```

```
    public static List<Student> GetStudents()
    {
        // Use a collection initializer to create the data source. Note that
each element
        //  in the list contains an inner sequence of scores.
        List<Student> students =
        [
           new Student {First="Svetlana", Last="Omelchenko", ID=111, Scores=
[97, 72, 81, 60]},
           new Student {First="Claire", Last="O'Donnell", ID=112, Scores=
[75, 84, 91, 39]},
           new Student {First="Sven", Last="Mortensen", ID=113, Scores= [99,
89, 91, 95]},
           new Student {First="Cesar", Last="Garcia", ID=114, Scores= [72,
81, 65, 84]},
           new Student {First="Debra", Last="Garcia", ID=115, Scores= [97,
89, 85, 82]}
        ];

        return students;
    }

    static void Main()
    {
        // Obtain the data source.
        List<Student> students = GetStudents();

        // Group by true or false.
        // Query variable is an IEnumerable<IGrouping<bool, Student>>
        var booleanGroupQuery =
            from student in students
            group student by student.Scores.Average() >= 80; //pass or fail!

        // Execute the query and access items in each group
        foreach (var studentGroup in booleanGroupQuery)
        {
            Console.WriteLine(studentGroup.Key == true ? "High averages" :
"Low averages");
            foreach (var student in studentGroup)
            {
                Console.WriteLine("   {0}, {1}:{2}", student.Last,
student.First, student.Scores.Average());
            }
        }
    }
}
/* Output:
  Low averages
   Omelchenko, Svetlana:77.5
   O'Donnell, Claire:72.25
   Garcia, Cesar:75.5
  High averages
   Mortensen, Sven:93.5
```

```
   Garcia, Debra:88.25
*/
```

按数值范围分组

下一示例使用表达式创建表示百分比范围的数值组键。 请注意，该示例使用 let 作为方法调用结果的方便存储位置，因此无需在 `group` 子句中调用该方法两次。 若要详细了解如何在查询表达式中安全使用方法，请参阅在查询表达式中处理异常。

C#

```
class GroupSample2
{
    // The element type of the data source.
    public class Student
    {
        public required string First { get; init; }
        public required string Last { get; init; }
        public required int ID { get; init; }
        public required List<int> Scores;
    }

    public static List<Student> GetStudents()
    {
        // Use a collection initializer to create the data source. Note that
each element
        //  in the list contains an inner sequence of scores.
        List<Student> students =
        [
           new Student {First="Svetlana", Last="Omelchenko", ID=111, Scores=
[97, 72, 81, 60]},
           new Student {First="Claire", Last="O'Donnell", ID=112, Scores=
[75, 84, 91, 39]},
           new Student {First="Sven", Last="Mortensen", ID=113, Scores= [99,
89, 91, 95]},
           new Student {First="Cesar", Last="Garcia", ID=114, Scores= [72,
81, 65, 84]},
           new Student {First="Debra", Last="Garcia", ID=115, Scores= [97,
89, 85, 82]}
        ];

        return students;
    }

    // This method groups students into percentile ranges based on their
    // grade average. The Average method returns a double, so to produce a
whole
    // number it is necessary to cast to int before dividing by 10.
    static void Main()
    {
        // Obtain the data source.
```

```
        List<Student> students = GetStudents();

        // Write the query.
        var studentQuery =
            from student in students
            let avg = (int)student.Scores.Average()
            group student by (avg / 10) into g
            orderby g.Key
            select g;

        // Execute the query.
        foreach (var studentGroup in studentQuery)
        {
            int temp = studentGroup.Key * 10;
            Console.WriteLine("Students with an average between {0} and
{1}", temp, temp + 10);
            foreach (var student in studentGroup)
            {
                Console.WriteLine("   {0}, {1}:{2}", student.Last,
student.First, student.Scores.Average());
            }
        }
    }
}
/* Output:
     Students with an average between 70 and 80
       Omelchenko, Svetlana:77.5
       O'Donnell, Claire:72.25
       Garcia, Cesar:75.5
     Students with an average between 80 and 90
       Garcia, Debra:88.25
     Students with an average between 90 and 100
       Mortensen, Sven:93.5
 */
```

按复合键分组

希望按照多个键对元素进行分组时，可使用复合键。 使用匿名类型或命名类型来存储键元素，创建复合键。 在下面的示例中，假定已经使用名为 `surname` 和 `city` 的两个成员声明了类 `Person`。 `group` 子句会为每组姓氏和城市相同的人员创建一个单独的组。

C#

```
group person by new {name = person.surname, city = person.city};
```

如果必须将查询变量传递给其他方法，请使用命名类型。 使用键的自动实现的属性创建一个特殊类，然后替代 Equals 和 GetHashCode 方法。 还可以使用结构，在此情况下，并不严格要求替代这些方法。 有关详细信息，请参阅如何使用自动实现的属性实现轻量

类和如何在目录树中查询重复文件。 后文包含的代码示例演示了如何将复合键与命名类型结合使用。

示例 1

下面的示例演示在没有向组应用附加查询逻辑时，将源数据按顺序放入组中的标准模式。 这称为不带延续的分组。 字符串数组中的元素按照它们的首字母进行分组。 查询的结果是 IGrouping<TKey,TElement> 类型（包含一个 `char` 类型的公共 `Key` 属性）和一个 IEnumerable<T> 集合（在分组中包含每个项）。

`group` 子句的结果是由序列组成的序列。 因此，若要访问返回的每个组中的单个元素，请在循环访问组键的循环内使用嵌套的 `foreach` 循环，如以下示例所示。

C#

```
class GroupExample1
{
    static void Main()
    {
        // Create a data source.
        string[] words = ["blueberry", "chimpanzee", "abacus", "banana",
"apple", "cheese"];

        // Create the query.
        var wordGroups =
            from w in words
            group w by w[0];

        // Execute the query.
        foreach (var wordGroup in wordGroups)
        {
            Console.WriteLine("Words that start with the letter '{0}':",
wordGroup.Key);
            foreach (var word in wordGroup)
            {
                Console.WriteLine(word);
            }
        }
    }
}
/* Output:
      Words that start with the letter 'b':
        blueberry
        banana
      Words that start with the letter 'c':
        chimpanzee
        cheese
      Words that start with the letter 'a':
        abacus
```

```
        apple
    */
```

示例 2

此示例演示在创建组之后，如何使用通过 `into` 实现的延续对这些组执行附加逻辑。 有关详细信息，请参阅 into。 下面的示例查询每个组，仅选择键值为元音的元素。

C#

```
class GroupClauseExample2
{
    static void Main()
    {
        // Create the data source.
        string[] words2 = ["blueberry", "chimpanzee", "abacus", "banana",
"apple", "cheese", "elephant", "umbrella", "anteater"];

        // Create the query.
        var wordGroups2 =
            from w in words2
            group w by w[0] into grps
            where (grps.Key == 'a' || grps.Key == 'e' || grps.Key == 'i'
                   || grps.Key == 'o' || grps.Key == 'u')
            select grps;

        // Execute the query.
        foreach (var wordGroup in wordGroups2)
        {
            Console.WriteLine("Groups that start with a vowel: {0}",
wordGroup.Key);
            foreach (var word in wordGroup)
            {
                Console.WriteLine("   {0}", word);
            }
        }
    }
}
/* Output:
    Groups that start with a vowel: a
        abacus
        apple
        anteater
    Groups that start with a vowel: e
        elephant
    Groups that start with a vowel: u
        umbrella
*/
```

备注

在编译时，`group` 子句转换为对 GroupBy 方法的调用。

`group` 子句查询的语法不支持自定义相等比较器。 若要在查询中使用 IEqualityComparer，请显式使用 GroupBy 方法。

另请参阅

- IGrouping<TKey,TElement>
- GroupBy
- ThenBy
- ThenByDescending
- 查询关键字
- 语言集成查询 (LINQ)
- 创建嵌套组
- 对查询结果进行分组
- 对分组操作执行子查询

在 GitHub 上与我们协作

可以在 GitHub 上找到此内容的源，还可以在其中创建和查看问题和拉取请求。 有关详细信息，请参阅参与者指南。

.NET

.NET 反馈

.NET 是一个开放源代码项目。 选择一个链接以提供反馈：

提出文档问题

提供产品反馈

into（C# 参考）

项目 • 2023/04/07

可使用 `into` 上下文关键字创建临时标识符，将 group、join 或 select 子句的结果存储至新标识符。 此标识符本身可以是附加查询命令的生成器。 有时称在 `group` 或 `select` 子句中使用新标识符为“延续”。

示例

下面的示例演示使用 `into` 关键字来启用具有推断类型 `IGrouping` 的临时标识符 `fruitGroup`。 通过使用该标识符，可对每个组调用 Count 方法，并且仅选择那些包含两个或更多个单词的组。

C#

```
class IntoSample1
{
    static void Main()
    {

        // Create a data source.
        string[] words = { "apples", "blueberries", "oranges", "bananas",
"apricots"};

        // Create the query.
        var wordGroups1 =
            from w in words
            group w by w[0] into fruitGroup
            where fruitGroup.Count() >= 2
            select new { FirstLetter = fruitGroup.Key, Words =
fruitGroup.Count() };

        // Execute the query. Note that we only iterate over the groups,
        // not the items in each group
        foreach (var item in wordGroups1)
        {
            Console.WriteLine(" {0} has {1} elements.", item.FirstLetter,
item.Words);
        }

        // Keep the console window open in debug mode
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
/* Output:
   a has 2 elements.
```

```
    b has 2 elements.
*/
```

仅当希望对每个组执行附加查询操作时，才需在 `group` 子句中使用 `into`。 有关详细信息，请参阅 group 子句。

有关在 `join` 子句中使用 `into` 的示例，请参见 join 子句。

另请参阅

- 查询关键字 (LINQ)
- C# 中的 LINQ
- group 子句

orderby 子句（C# 参考）

项目 • 2024/04/02

在查询表达式中，`orderby` 子句可导致返回的序列或子序列（组）以升序或降序排序。 若要执行一个或多个次级排序操作，可以指定多个键。 元素类型的默认比较器执行排序。 默认排序顺序为升序。 还可以指定自定义比较器。 但是，只适用于使用基于方法的语法。 有关详细信息，请参阅对数据进行排序。

示例 1

在以下示例中，第一个查询按字母顺序从 A 开始对字词排序，而第二个查询则按降序对相同的字词排序。 （`ascending` 关键字是默认排序值，可省略。）

C#

```
class OrderbySample1
{
    static void Main()
    {
        // Create a delicious data source.
        string[] fruits = ["cherry", "apple", "blueberry"];

        // Query for ascending sort.
        IEnumerable<string> sortAscendingQuery =
            from fruit in fruits
            orderby fruit //"ascending" is default
            select fruit;

        // Query for descending sort.
        IEnumerable<string> sortDescendingQuery =
            from w in fruits
            orderby w descending
            select w;

        // Execute the query.
        Console.WriteLine("Ascending:");
        foreach (string s in sortAscendingQuery)
        {
            Console.WriteLine(s);
        }

        // Execute the query.
        Console.WriteLine(Environment.NewLine + "Descending:");
        foreach (string s in sortDescendingQuery)
        {
            Console.WriteLine(s);
        }
    }
```

```
}
/* Output:
Ascending:
apple
blueberry
cherry

Descending:
cherry
blueberry
apple
*/
```

示例 2

以下示例对学生的姓氏进行主要排序，然后对其名字进行次要排序。

C#

```
class OrderbySample2
{
    // The element type of the data source.
    public class Student
    {
        public required string First { get; init; }
        public required string Last { get; init; }
        public int ID { get; set; }
    }

    public static List<Student> GetStudents()
    {
        // Use a collection initializer to create the data source. Note that
each element
        //  in the list contains an inner sequence of scores.
        List<Student> students = new List<Student>
        {
           new Student {First="Svetlana", Last="Omelchenko", ID=111},
           new Student {First="Claire", Last="O'Donnell", ID=112},
           new Student {First="Sven", Last="Mortensen", ID=113},
           new Student {First="Cesar", Last="Garcia", ID=114},
           new Student {First="Debra", Last="Garcia", ID=115}
        };

        return students;
    }
    static void Main(string[] args)
    {
        // Create the data source.
        List<Student> students = GetStudents();

        // Create the query.
        IEnumerable<Student> sortedStudents =
```

```
            from student in students
            orderby student.Last ascending, student.First ascending
            select student;

        // Execute the query.
        Console.WriteLine("sortedStudents:");
        foreach (Student student in sortedStudents)
            Console.WriteLine(student.Last + " " + student.First);

        // Now create groups and sort the groups. The query first sorts the
names
        // of all students so that they will be in alphabetical order after
they are
        // grouped. The second orderby sorts the group keys in alpha order.
        var sortedGroups =
            from student in students
            orderby student.Last, student.First
            group student by student.Last[0] into newGroup
            orderby newGroup.Key
            select newGroup;

        // Execute the query.
        Console.WriteLine(Environment.NewLine + "sortedGroups:");
        foreach (var studentGroup in sortedGroups)
        {
            Console.WriteLine(studentGroup.Key);
            foreach (var student in studentGroup)
            {
                Console.WriteLine("   {0}, {1}", student.Last,
student.First);
            }
        }
    }
}
/* Output:
sortedStudents:
Garcia Cesar
Garcia Debra
Mortensen Sven
O'Donnell Claire
Omelchenko Svetlana

sortedGroups:
G
   Garcia, Cesar
   Garcia, Debra
M
   Mortensen, Sven
O
   O'Donnell, Claire
   Omelchenko, Svetlana
*/
```

备注

编译时，`orderby` 子句将转换为对 OrderBy 方法的调用。 `orderby` 子句中的多个关键值将转换为 ThenBy 方法调用。

另请参阅

- 查询关键字 (LINQ)
- C# 中的 LINQ
- group 子句
- 语言集成查询 (LINQ)

在 GitHub 上与我们协作

可以在 GitHub 上找到此内容的源，还可以在其中创建和查看问题和拉取请求。 有关详细信息，请参阅参与者指南。

.NET

.NET 反馈

.NET 是一个开放源代码项目。 选择一个链接以提供反馈：

提出文档问题

提供产品反馈

join 子句（C# 参考）

项目 • 2023/04/07

`join` 子句可用于将来自不同源序列并且在对象模型中没有直接关系的元素相关联。 唯一的要求是每个源中的元素需要共享某个可以进行比较以判断是否相等的值。 例如，食品经销商可能拥有某种产品的供应商列表以及买主列表。 例如，可以使用 `join` 子句创建该产品同一指定地区供应商和买主的列表。

`join` 子句将 2 个源序列作为输入。 每个序列中的元素都必须是可以与其他序列中的相应属性进行比较的属性，或者包含一个这样的属性。 `join` 子句使用特殊 `equals` 关键字比较指定的键是否相等。 `join` 子句执行的所有联接都是同等联接。 `join` 子句的输出形式取决于执行的联接的具体类型。 以下是 3 种最常见的联接类型：

- 内部联接
- 分组联接
- 左外部联接

内部联接

以下示例演示了一个简单的内部同等联接。 此查询生成一个“产品名称/类别”对平面序列。 同一类别字符串将出现在多个元素中。 如果 `categories` 中的某个元素不具有匹配的 `products`，则该类别不会出现在结果中。

C#

```
var innerJoinQuery =
    from category in categories
    join prod in products on category.ID equals prod.CategoryID
    select new { ProductName = prod.Name, Category = category.Name };
//produces flat sequence
```

有关详细信息，请参阅执行内联。

分组联接

含有 `into` 表达式的 `join` 子句称为分组联接。

C#

```
var innerGroupJoinQuery =
    from category in categories
    join prod in products on category.ID equals prod.CategoryID into
prodGroup
    select new { CategoryName = category.Name, Products = prodGroup };
```

分组联接会生成分层的结果序列，该序列将左侧源序列中的元素与右侧源序列中的一个或多个匹配元素相关联。 分组联接没有等效的关系术语；它本质上是一个对象数组序列。

如果在右侧源序列中找不到与左侧源中的元素相匹配的元素，则 `join` 子句会为该项生成一个空数组。 因此，分组联接基本上仍然是一种内部同等联接，区别在于分组联接将结果序列组织为多个组。

如果只选择分组联接的结果，则可访问各项，但无法识别结果所匹配的项。 因此，通常更为有用的做法是：选择分组联接的结果并将其放入一个也包含该项名的新类型中，如上例所示。

当然，还可以将分组联接的结果用作其他子查询的生成器：

C#

```
var innerGroupJoinQuery2 =
    from category in categories
    join prod in products on category.ID equals prod.CategoryID into
prodGroup
    from prod2 in prodGroup
    where prod2.UnitPrice > 2.50M
    select prod2;
```

有关详细信息，请参阅执行分组联接。

左外部联接

在左外部联接中，将返回左侧源序列中的所有元素，即使右侧序列中没有其匹配元素也是如此。 若要在 LINQ 中执行左外部联接，请结合使用 `DefaultIfEmpty` 方法与分组联接，指定要在某个左侧元素不具有匹配元素时生成的默认右侧元素。 可以使用 `null` 作为任何引用类型的默认值，也可以指定用户定义的默认类型。 以下示例演示了用户定义的默认类型：

C#

```
var leftOuterJoinQuery =
    from category in categories
    join prod in products on category.ID equals prod.CategoryID into
prodGroup
```

```
    from item in prodGroup.DefaultIfEmpty(new Product { Name = String.Empty,
CategoryID = 0 })
    select new { CatName = category.Name, ProdName = item.Name };
```

有关详细信息，请参阅执行左外部联接。

等于运算符

`join` 子句执行同等联接。 换言之，只能基于 2 个项之间的相等关系进行匹配。 不支持其他类型的比较，例如“大于”或“不等于”。 为了表明所有联接都是同等联接，`join` 子句使用 `equals` 关键字而不是 `==` 运算符。 `equals` 关键字只能在 `join` 子句中使用，并且其与 `==` 运算符之间存在某些重要差别。 比较字符串时，`equals` 有一个按值比较的重载，而 `==` 运算符使用引用相等性。 如果比较的双方具有相同的字符串变量，`equals` 和 `==` 将得到相同的结果，也就是 true。 这是因为，当程序声明两个或多个等效的字符串变量时，编译器会将它们都存储在同一位置，请参阅引用相等性和字符串驻留了解详细信息。 另一个重要区别是 NULL 比较：`null equals null` 使用 `equals` 运算符的计算结果为 false，而不使用计算结果为 true 的 `==` 运算符。 最后，范围行为不同：对于 `equals`，左键使用外部源序列，而右键使用内部源。 外部源仅在 `equals` 的左侧位于范围内，而内部源序列仅在其右侧位于范围内。

非同等联接

通过使用多个 `from` 子句将新序列单独引入查询，可以执行非同等联接、交叉联接和其他自定义联接操作。 有关详细信息，请参阅执行自定义联接操作。

对象集合联接与关系表

在 LINQ 查询表达式中，联接操作是在对象集合上执行的。 不能使用与 2 个关系表完全相同的方式“联接”对象集合。 在 LINQ 中，仅当 2 个源序列没有通过任何关系相互联系时，才需要使用显式 `join` 子句。 使用 LINQ to SQL 时，外键表在对象模型中表示为主表的属性。 例如，在 Northwind 数据库中，Customer 表与 Orders 表之间具有外键关系。 将这两个表映射到对象模型时，Customer 类具有一个 Orders 属性，其中包含与该 Customer 相关联的 Orders 集合。 实际上，已经为你执行了联接。

若要深入了解如何在 LINQ to SQL 的上下文中跨相关表执行查询，请参阅如何：映射数据库关系。

组合键

可通过使用组合键测试多个值是否相等。 有关详细信息，请参阅使用组合键进行联接。 还可以在 `group` 子句中使用组合键。

示例

以下示例比较了使用相同的匹配键对相同数据源执行内部联接、分组联接和左外部联接的结果。 这些示例中添加了一些额外的代码，以便在控制台显示中阐明结果。

C#

```
class JoinDemonstration
{
    #region Data

    class Product
    {
        public string Name { get; set; }
        public int CategoryID { get; set; }
    }

    class Category
    {
        public string Name { get; set; }
        public int ID { get; set; }
    }

    // Specify the first data source.
    List<Category> categories = new List<Category>()
    {
        new Category {Name="Beverages", ID=001},
        new Category {Name="Condiments", ID=002},
        new Category {Name="Vegetables", ID=003},
        new Category {Name="Grains", ID=004},
        new Category {Name="Fruit", ID=005}
    };

    // Specify the second data source.
    List<Product> products = new List<Product>()
   {
      new Product {Name="Cola",  CategoryID=001},
      new Product {Name="Tea",  CategoryID=001},
      new Product {Name="Mustard", CategoryID=002},
      new Product {Name="Pickles", CategoryID=002},
      new Product {Name="Carrots", CategoryID=003},
      new Product {Name="Bok Choy", CategoryID=003},
      new Product {Name="Peaches", CategoryID=005},
      new Product {Name="Melons", CategoryID=005},
    };
    #endregion

    static void Main(string[] args)
```

```
    {
        JoinDemonstration app = new JoinDemonstration();

        app.InnerJoin();
        app.GroupJoin();
        app.GroupInnerJoin();
        app.GroupJoin3();
        app.LeftOuterJoin();
        app.LeftOuterJoin2();

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }

    void InnerJoin()
    {
        // Create the query that selects
        // a property from each element.
        var innerJoinQuery =
           from category in categories
           join prod in products on category.ID equals prod.CategoryID
           select new { Category = category.ID, Product = prod.Name };

        Console.WriteLine("InnerJoin:");
        // Execute the query. Access results
        // with a simple foreach statement.
        foreach (var item in innerJoinQuery)
        {
            Console.WriteLine("{0,-10}{1}", item.Product, item.Category);
        }
        Console.WriteLine("InnerJoin: {0} items in 1 group.",
innerJoinQuery.Count());
        Console.WriteLine(System.Environment.NewLine);
    }

    void GroupJoin()
    {
        // This is a demonstration query to show the output
        // of a "raw" group join. A more typical group join
        // is shown in the GroupInnerJoin method.
        var groupJoinQuery =
           from category in categories
           join prod in products on category.ID equals prod.CategoryID into
prodGroup
           select prodGroup;

        // Store the count of total items (for demonstration only).
        int totalItems = 0;

        Console.WriteLine("Simple GroupJoin:");

        // A nested foreach statement is required to access group items.
        foreach (var prodGrouping in groupJoinQuery)
        {
```

```
            Console.WriteLine("Group:");
            foreach (var item in prodGrouping)
            {
                totalItems++;
                Console.WriteLine("   {0,-10}{1}", item.Name,
item.CategoryID);
            }
        }
        Console.WriteLine("Unshaped GroupJoin: {0} items in {1} unnamed
groups", totalItems, groupJoinQuery.Count());
        Console.WriteLine(System.Environment.NewLine);
    }

    void GroupInnerJoin()
    {
        var groupJoinQuery2 =
            from category in categories
            orderby category.ID
            join prod in products on category.ID equals prod.CategoryID into
prodGroup
            select new
            {
                Category = category.Name,
                Products = from prod2 in prodGroup
                           orderby prod2.Name
                           select prod2
            };

        //Console.WriteLine("GroupInnerJoin:");
        int totalItems = 0;

        Console.WriteLine("GroupInnerJoin:");
        foreach (var productGroup in groupJoinQuery2)
        {
            Console.WriteLine(productGroup.Category);
            foreach (var prodItem in productGroup.Products)
            {
                totalItems++;
                Console.WriteLine("  {0,-10} {1}", prodItem.Name,
prodItem.CategoryID);
            }
        }
        Console.WriteLine("GroupInnerJoin: {0} items in {1} named groups",
totalItems, groupJoinQuery2.Count());
        Console.WriteLine(System.Environment.NewLine);
    }

    void GroupJoin3()
    {

        var groupJoinQuery3 =
            from category in categories
            join product in products on category.ID equals
product.CategoryID into prodGroup
            from prod in prodGroup
```

```
            orderby prod.CategoryID
            select new { Category = prod.CategoryID, ProductName = prod.Name
};

        //Console.WriteLine("GroupInnerJoin:");
        int totalItems = 0;

        Console.WriteLine("GroupJoin3:");
        foreach (var item in groupJoinQuery3)
        {
            totalItems++;
            Console.WriteLine("   {0}:{1}", item.ProductName,
item.Category);
        }

        Console.WriteLine("GroupJoin3: {0} items in 1 group", totalItems);
        Console.WriteLine(System.Environment.NewLine);
    }

    void LeftOuterJoin()
    {
        // Create the query.
        var leftOuterQuery =
           from category in categories
           join prod in products on category.ID equals prod.CategoryID into
prodGroup
           select prodGroup.DefaultIfEmpty(new Product() { Name =
"Nothing!", CategoryID = category.ID });

        // Store the count of total items (for demonstration only).
        int totalItems = 0;

        Console.WriteLine("Left Outer Join:");

        // A nested foreach statement  is required to access group items
        foreach (var prodGrouping in leftOuterQuery)
        {
            Console.WriteLine("Group:");
            foreach (var item in prodGrouping)
            {
                totalItems++;
                Console.WriteLine("  {0,-10}{1}", item.Name,
item.CategoryID);
            }
        }
        Console.WriteLine("LeftOuterJoin: {0} items in {1} groups",
totalItems, leftOuterQuery.Count());
        Console.WriteLine(System.Environment.NewLine);
    }

    void LeftOuterJoin2()
    {
        // Create the query.
        var leftOuterQuery2 =
           from category in categories
```

```
            join prod in products on category.ID equals prod.CategoryID into
prodGroup
            from item in prodGroup.DefaultIfEmpty()
            select new { Name = item == null ? "Nothing!" : item.Name,
CategoryID = category.ID };

        Console.WriteLine("LeftOuterJoin2: {0} items in 1 group",
leftOuterQuery2.Count());
        // Store the count of total items
        int totalItems = 0;

        Console.WriteLine("Left Outer Join 2:");

        // Groups have been flattened.
        foreach (var item in leftOuterQuery2)
        {
            totalItems++;
            Console.WriteLine("{0,-10}{1}", item.Name, item.CategoryID);
        }
        Console.WriteLine("LeftOuterJoin2: {0} items in 1 group",
totalItems);
    }
}
/*Output:

InnerJoin:
Cola      1
Tea       1
Mustard   2
Pickles   2
Carrots   3
Bok Choy  3
Peaches   5
Melons    5
InnerJoin: 8 items in 1 group.


Unshaped GroupJoin:
Group:
    Cola      1
    Tea       1
Group:
    Mustard   2
    Pickles   2
Group:
    Carrots   3
    Bok Choy  3
Group:
Group:
    Peaches   5
    Melons    5
Unshaped GroupJoin: 8 items in 5 unnamed groups


GroupInnerJoin:
```

```
Beverages
    Cola       1
    Tea        1
Condiments
    Mustard    2
    Pickles    2
Vegetables
    Bok Choy   3
    Carrots    3
Grains
Fruit
    Melons     5
    Peaches    5
GroupInnerJoin: 8 items in 5 named groups


GroupJoin3:
    Cola:1
    Tea:1
    Mustard:2
    Pickles:2
    Carrots:3
    Bok Choy:3
    Peaches:5
    Melons:5
GroupJoin3: 8 items in 1 group


Left Outer Join:
Group:
    Cola      1
    Tea       1
Group:
    Mustard   2
    Pickles   2
Group:
    Carrots   3
    Bok Choy  3
Group:
    Nothing!  4
Group:
    Peaches   5
    Melons    5
LeftOuterJoin: 9 items in 5 groups


LeftOuterJoin2: 9 items in 1 group
Left Outer Join 2:
Cola      1
Tea       1
Mustard   2
Pickles   2
Carrots   3
Bok Choy  3
Nothing!  4
```

```
Peaches   5
Melons    5
LeftOuterJoin2: 9 items in 1 group
Press any key to exit.
*/
```

备注

后面未跟 `into` 的 `join` 子句转换为 Join 方法调用。 后面跟 `into` 的 `join` 子句转换为 GroupJoin 方法调用。

另请参阅

- 查询关键字 (LINQ)
- 语言集成查询 (LINQ)
- 联接运算
- group 子句
- 执行左外部联接
- 执行内部联接
- 执行分组联接
- 对 Join 子句的结果进行排序
- 使用组合键进行联接
- 适用于 Visual Studio 的兼容数据库系统

let 子句（C# 参考）

项目 • 2024/04/07

在查询表达式中，存储子表达式的结果有时很有帮助，可在后续子句中使用。 可以通过 `let` 关键字执行此操作，该关键字创建一个新的范围变量并通过提供的表达式结果初始化该变量。 使用值进行初始化后，范围变量不能用于存储另一个值。 但是，如果范围变量持有可查询类型，则可以查询该变量。

示例

以两种方式使用以下示例 `let`：

1. 创建一个可以查询其自身的可枚举类型。
2. 使查询仅调用一次范围变量 `word` 上的 `ToLower`。 如果不使用 `let`，则不得不调用 `where` 子句中的每个谓词的 `ToLower`。

C#

```
class LetSample1
{
    static void Main()
    {
        string[] strings =
        [
            "A penny saved is a penny earned.",
            "The early bird catches the worm.",
            "The pen is mightier than the sword."
        ];

        // Split the sentence into an array of words
        // and select those whose first letter is a vowel.
        var earlyBirdQuery =
            from sentence in strings
            let words = sentence.Split(' ')
            from word in words
            let w = word.ToLower()
            where w[0] == 'a' || w[0] == 'e'
                || w[0] == 'i' || w[0] == 'o'
                || w[0] == 'u'
            select word;

        // Execute the query.
        foreach (var v in earlyBirdQuery)
        {
            Console.WriteLine("\"{0}\" starts with a vowel", v);
        }
    }
```

```
}
/* Output:
    "A" starts with a vowel
    "is" starts with a vowel
    "a" starts with a vowel
    "earned." starts with a vowel
    "early" starts with a vowel
    "is" starts with a vowel
*/
```

另请参阅

- 查询关键字 (LINQ)
- C# 中的 LINQ
- 语言集成查询 (LINQ)
- 在查询表达式中处理异常

在 GitHub 上与我们协作

可以在 GitHub 上找到此内容的源，还可以在其中创建和查看问题和拉取请求。 有关详细信息，请参阅参与者指南。

.NET 反馈

.NET 是一个开放源代码项目。 选择一个链接以提供反馈：

提出文档问题

提供产品反馈

ascending（C# 参考）

项目 • 2024/04/02

查询表达式中的 orderby 子句 中使用 `ascending` 上下文关键字指定排序顺序为从小到大。 由于 `ascending` 为默认排序顺序，因此无需加以指定。

示例

下面的示例说明 `ascending` 在 orderby 子句中的用法。

C#

```
IEnumerable<string> sortAscendingQuery =
    from vegetable in vegetables
    orderby vegetable ascending
    select vegetable;
```

另请参阅

- C# 中的 LINQ
- descending

在 GitHub 上与我们协作

可以在 GitHub 上找到此内容的源，还可以在其中创建和查看问题和拉取请求。 有关详细信息，请参阅参与者指南。

.NET

.NET 反馈

.NET 是一个开放源代码项目。 选择一个链接以提供反馈：

提出文档问题

提供产品反馈

descending（C# 参考）

项目 • 2024/04/02

查询表达式中的 orderby 子句中使用 `descending` 上下文关键字指定排序顺序为从大到小。

示例

下面的示例说明 `descending` 在 orderby 子句中的用法。

```
C#
```

```
IEnumerable<string> sortDescendingQuery =
    from vegetable in vegetables
    orderby vegetable descending
    select vegetable;
```

另请参阅

- C# 中的 LINQ
- ascending

在 GitHub 上与我们协作

可以在 GitHub 上找到此内容的源，还可以在其中创建和查看问题和拉取请求。 有关详细信息，请参阅参与者指南。

.NET 反馈

.NET 是一个开放源代码项目。 选择一个链接以提供反馈：

提出文档问题

提供产品反馈

on（C# 参考）

项目 • 2024/04/02

`on` 上下文关键字用于在查询表达式的 join 子句中指定联接条件。

示例

下面的示例说明了 `on` 在 `join` 子句中的用法。

```C#
var innerJoinQuery =
    from category in categories
    join prod in products on category.ID equals prod.CategoryID
    select new { ProductName = prod.Name, Category = category.Name };
```

另请参阅

- 语言集成查询 (LINQ)

在 GitHub 上与我们协作

可以在 GitHub 上找到此内容的源，还可以在其中创建和查看问题和拉取请求。 有关详细信息，请参阅参与者指南。

.NET

.NET 反馈

.NET 是一个开放源代码项目。 选择一个链接以提供反馈：

提出文档问题

提供产品反馈

equals（C# 参考）

项目 • 2023/04/07

`equals` 上下文关键字用于在查询表达式的 `join` 子句中比较两个序列的元素。 有关详细信息，请参阅 join 子句。

示例

下面的示例说明 `equals` 关键字在 `join` 子句中的用法。

C#

```
var innerJoinQuery =
    from category in categories
    join prod in products on category.ID equals prod.CategoryID
    select new { ProductName = prod.Name, Category = category.Name };
```

请参阅

- 语言集成查询 (LINQ)

by（C# 参考）

项目 • 2023/04/07

`by` 上下文关键字用于在查询表达式的 `group` 子句中指定应返回项的分组方式。 有关详细信息，请参阅 group 子句。

示例

下面的示例演示在 `group` 字句中使用 `by` 关键字指定应根据每个学生的姓氏首字母对学生分组。

C#

```
var query = from student in students
            group student by student.LastName[0];
```

另请参阅

- C# 中的 LINQ

in（C# 参考）

项目 • 2023/04/07

在以下上下文中，使用了 `in` 关键字：

- 泛型接口和委托中的泛型类型参数。
- 作为参数修饰符，它允许按引用而不是按值向方法传递参数。
- foreach 语句。
- LINQ 查询表达式中的 from 子句。
- LINQ 查询表达式中的 join 子句。

请参阅

- C# 关键字
- C# 参考

C# 运算符和表达式（C# 参考）

项目 • 2024/04/11

C# 提供了许多运算符。 其中许多都受到内置类型的支持，可用于对这些类型的值执行基本操作。 这些运算符包括以下组：

- 算术运算符，将对数值操作数执行算术运算
- 比较运算符，将比较数值操作数
- 布尔逻辑运算符，将对 bool 操作数执行逻辑运算
- 位运算符和移位运算符，将对整数类型的操作数执行位运算或移位运算
- 相等运算符，将检查其操作数是否相等

通常可以重载这些运算符，也就是说，可以为用户定义类型的操作数指定运算符行为。

最简单的 C# 表达式是文本（例如整数和实数）和变量名称。 可以使用运算符将它们组合成复杂的表达式。 运算符优先级和结合性决定了表达式中操作的执行顺序。 可以使用括号更改由运算符优先级和结合性决定的计算顺序。

在下面的代码中，表达式的示例位于赋值的右侧：

C#

```
int a, b, c;
a = 7;
b = a;
c = b++;
b = a + b * c;
c = a >= 100 ? b : c / 10;
a = (int)Math.Sqrt(b * b + c * c);

string s = "String literal";
char l = s[s.Length - 1];

var numbers = new List<int>(new[] { 1, 2, 3 });
b = numbers.FindLast(n => n > 1);
```

通常情况下，表达式会生成结果，并可包含在其他表达式中。 void 方法调用是不生成结果的表达式的示例。 它只能用作语句，如下面的示例所示：

C#

```
Console.WriteLine("Hello, world!");
```

下面是 C# 提供的一些其他类型的表达式：

- 内插字符串表达式，提供创建格式化字符串的便利语法：

```C#
var r = 2.3;
var message = $"The area of a circle with radius {r} is {Math.PI * r * r:F3}.";
Console.WriteLine(message);
// Output:
// The area of a circle with radius 2.3 is 16.619.
```

- Lambda 表达式，可用于创建匿名函数：

```C#
int[] numbers = { 2, 3, 4, 5 };
var maximumSquare = numbers.Max(x => x * x);
Console.WriteLine(maximumSquare);
// Output:
// 25
```

- 查询表达式，可用于直接以 C# 使用查询功能：

```C#
var scores = new[] { 90, 97, 78, 68, 85 };
IEnumerable<int> highScoresQuery =
    from score in scores
    where score > 80
    orderby score descending
    select score;
Console.WriteLine(string.Join(" ", highScoresQuery));
// Output:
// 97 90 85
```

可使用表达式主体定义为方法、构造函数、属性、索引器或终结器提供简洁的定义。

运算符优先级

在包含多个运算符的表达式中，先按优先级较高的运算符计算，再按优先级较低的运算符计算。 在下面的示例中，首先执行乘法，因为其优先级高于加法：

```C#
var a = 2 + 2 * 2;
Console.WriteLine(a); //  output: 6
```

使用括号更改运算符优先级所施加的计算顺序：

C#

```
var a = (2 + 2) * 2;
Console.WriteLine(a); //  output: 8
```

下表按最高优先级到最低优先级的顺序列出 C# 运算符。 每行中运算符的优先级相同。

展开表

运算符	类别或名称
x.y、f(x)、a[i]、x?.y、x?[y]、x++、x--、x!、new、typeof、checked、unchecked、default、nameof、delegate、sizeof、stackalloc、x->y	主要
+x、-x、x、~x、++x、--x、^x、(T)x、await、&&x、*x、true 和 false	一元
x..y	范围
switch、with	`switch` 和 `with` 表达式
x * y、x / y、x % y	乘法
x + y、x – y	加法
x << y、x >> y	Shift
x < y、x > y、x <= y、x >= y、is、as	关系和类型测试
x == y、x != y	相等
`x & y`	布尔逻辑 AND 或按位逻辑 AND
`x ^ y`	布尔逻辑 XOR 或按位逻辑 XOR
`x | y`	布尔逻辑 OR 或按位逻辑 OR
x && y	条件“与”
x || y	条件“或”
x ?? y	Null 合并运算符
c ? t : f	条件运算符

运算符	类别或名称
x = y、x += y、x -= y、x *= y、x /= y、x %= y、x &= y、x |= y、x ^= y、x <<= y、x >>= y、x ??= y、=>	赋值和 lambda 声明

运算符结合性

当运算符的优先级相同，运算符的结合性决定了运算的执行顺序：

- 左结合运算符按从左到右的顺序计算。 除赋值运算符和 null 合并运算符外，所有二元运算符都是左结合运算符。 例如，`a + b - c` 将计算为 `(a + b) - c`。
- 右结合运算符按从右到左的顺序计算。 赋值运算符、null 合并运算符、lambda 和条件运算符?:是右结合运算符。 例如，`x = y = z` 将计算为 `x = (y = z)`。

使用括号更改运算符结合性所施加的计算顺序：

C#

```
int a = 13 / 5 / 2;
int b = 13 / (5 / 2);
Console.WriteLine($"a = {a}, b = {b}");  // output: a = 1, b = 6
```

操作数计算

与运算符的优先级和结合性无关，从左到右计算表达式中的操作数。 以下示例展示了运算符和操作数的计算顺序：

展开表

表达式	计算顺序
`a + b`	a, b, +
`a + b * c`	a, b, c, *, +
`a / b + c * d`	a, b, /, c, d, *, +
`a / (b + c) * d`	a, b, c, +, /, d, *

通常，会计算所有运算符操作数。 但是，某些运算符有条件地计算操作数。 也就是说，此类运算符的最左侧操作数的值定义了是否应计算其他操作数，或计算其他哪些操作数。 这些运算符有条件逻辑 AND (&&) 和 OR (||) 运算符、null 合并运算符 ?? 和 ??=、null 条件运算符 ?. 和 ?[] 以及条件运算符?:。 有关详细信息，请参阅每个运算符的说明。

C# 语言规范

有关更多信息，请参阅 C# 语言规范的以下部分：

- 表达式
- 运算符

请参阅

- C# 参考
- 运算符重载
- 表达式树

反馈

此页面是否有帮助？ 是 否

提供产品反馈

算术运算符（C# 参考）

项目 • 2023/05/10

以下运算符对数值类型的操作数执行算术运算：

- 一元 `++`（增量）、`--`（减量）、`+`（加）和 `-`（减）运算符
- 二元 `*`（乘法）、`/`（除法）、`%`（余数）、`+`（加法）和 `-`（减法）运算符

所有整型和浮点数值类型都支持这些运算符。

对于整型类型，这些运算符（除 `++` 和 `--` 运算符以外）是为 `int`、`uint`、`long` 和 `ulong` 类型定义的。 如果操作数都是其他整型类型（`sbyte`、`byte`、`short`、`ushort` 或 `char`），它们的值将转换为 `int` 类型，这也是一个运算的结果类型。 如果操作数是不同的整型类型或浮点类型，它们的值将转换为最接近的包含类型（如果存在该类型）。 有关详细信息，请参阅 C# 语言规范的数值提升部分。 `++` 和 `--` 运算符是为所有整型和浮点数值类型以及 char 类型定义的。 复合赋值表达式的结果类型是左侧操作数的类型。

增量运算符 ++

一元增量运算符 `++` 按 1 递增其操作数。 操作数必须是变量、属性访问或索引器访问。

增量运算符以两种形式进行支持：后缀增量运算符 `x++` 和前缀增量运算符 `++x`。

后缀递增运算符

`x++` 的结果是此操作前的 `x` 的值，如以下示例所示：

```
C#

int i = 3;
Console.WriteLine(i);   // output: 3
Console.WriteLine(i++); // output: 3
Console.WriteLine(i);   // output: 4
```

前缀增量运算符

`++x` 的结果是此操作后的 `x` 的值，如以下示例所示：

```
C#

double a = 1.5;
Console.WriteLine(a);   // output: 1.5
```

```
Console.WriteLine(++a); // output: 2.5
Console.WriteLine(a);   // output: 2.5
```

减量运算符 --

一元减量运算符 `--` 按 1 递减其操作数。 操作数必须是变量、属性访问或索引器访问。

减量运算符以两种形式进行支持：后缀减量运算符`x--` 和前缀减量运算符 `--x`。

后缀递减运算符

`x--` 的结果是此操作前的 `x`的值，如以下示例所示：

C#

```
int i = 3;
Console.WriteLine(i);   // output: 3
Console.WriteLine(i--); // output: 3
Console.WriteLine(i);   // output: 2
```

前缀减量运算符

`--x` 的结果是此操作后的 `x`的值，如以下示例所示：

C#

```
double a = 1.5;
Console.WriteLine(a);   // output: 1.5
Console.WriteLine(--a); // output: 0.5
Console.WriteLine(a);   // output: 0.5
```

一元加和减运算符

一元 `+` 运算符返回其操作数的值。 一元 `-` 运算符对其操作数的数值取负。

C#

```
Console.WriteLine(+4);     // output: 4

Console.WriteLine(-4);     // output: -4
Console.WriteLine(-(-4));  // output: 4

uint a = 5;
```

```
var b = -a;
Console.WriteLine(b);            // output: -5
Console.WriteLine(b.GetType());  // output: System.Int64

Console.WriteLine(-double.NaN);  // output: NaN
```

ulong 类型不支持一元 `-` 运算符。

乘法运算符 *

乘法运算符 `*` 计算其操作数的乘积：

C#

```
Console.WriteLine(5 * 2);          // output: 10
Console.WriteLine(0.5 * 2.5);      // output: 1.25
Console.WriteLine(0.1m * 23.4m);   // output: 2.34
```

一元 `*` 运算符是指针间接运算符。

除法运算符 /

除法运算符 `/` 用它的左侧操作数除以右侧操作数。

整数除法

对于整数类型的操作数，`/` 运算符的结果为整数类型，并且等于两个操作数之商向零舍入后的结果：

C#

```
Console.WriteLine(13 / 5);    // output: 2
Console.WriteLine(-13 / 5);   // output: -2
Console.WriteLine(13 / -5);   // output: -2
Console.WriteLine(-13 / -5);  // output: 2
```

若要获取浮点数形式的两个操作数之商，请使用 `float`、`double` 或 `decimal` 类型：

C#

```
Console.WriteLine(13 / 5.0);       // output: 2.6

int a = 13;
```

```
int b = 5;
Console.WriteLine((double)a / b);  // output: 2.6
```

浮点除法

对于 `float`、`double` 和 `decimal` 类型，`/` 运算符的结果为两个操作数之商：

```
C#

Console.WriteLine(16.8f / 4.1f);   // output: 4.097561
Console.WriteLine(16.8d / 4.1d);   // output: 4.09756097560976
Console.WriteLine(16.8m / 4.1m);   // output: 4.0975609756097560975609756098
```

如果操作数之一为 `decimal`，那么另一个操作数不得为 `float` 和 `double`，因为 `float` 和 `double` 都无法隐式转换为 `decimal`。 必须将 `float` 或 `double` 操作数显式转换为 `decimal` 类型。 如需详细了解数值类型之间的转换，请参阅内置数值转换。

余数运算符 %

余数运算符 `%` 计算左侧操作数除以右侧操作数后的余数。

整数余数

对于整数类型的操作数，`a % b` 的结果是 `a - (a / b) * b` 得出的值。 非零余数的符号与左侧操作数的符号相同，如下例所示：

```
C#

Console.WriteLine(5 % 4);   // output: 1
Console.WriteLine(5 % -4);  // output: 1
Console.WriteLine(-5 % 4);  // output: -1
Console.WriteLine(-5 % -4); // output: -1
```

使用 Math.DivRem 方法计算整数除法和余数结果。

浮点余数

对于 `float` 和 `double` 操作数，有限的 `x` 和 `y` 的 `x % y` 的结果是值 `z`，因此

- `z`（如果不为零）的符号与 `x` 的符号相同。

- `z` 的绝对值是 `|x| - n * |y|` 得出的值，其中 `n` 是小于或等于 `|x| / |y|` 的最大可能整数，`|x|` 和 `|y|` 分别是 `x` 和 `y` 的绝对值。

> **备注**
>
> 计算余数的此方法类似于用于整数操作数的方法，但与 IEEE 754 规范不同。 如果需要符合 IEEE 754 规范的余数运算，请使用 Math.IEEERemainder 方法。

有关非限定操作数的 `%` 运算符行为的信息，请参阅 C# 语言规范的余数运算符章节。

对于 `decimal` 操作数，余数运算符 `%` 等效于 System.Decimal 类型的余数运算符。

以下示例演示了具有浮动操作数的余数运算符的行为：

```
C#
```

```
Console.WriteLine(-5.2f % 2.0f); // output: -1.2
Console.WriteLine(5.9 % 3.1);    // output: 2.8
Console.WriteLine(5.9m % 3.1m);  // output: 2.8
```

加法运算符 +

加法运算符 `+` 计算其操作数的和。

```
C#
```

```
Console.WriteLine(5 + 4);       // output: 9
Console.WriteLine(5 + 4.3);     // output: 9.3
Console.WriteLine(5.1m + 4.2m); // output: 9.3
```

还可以将 `+` 运算符用于字符串串联和委托组合。 有关详细信息，请参阅 + 和 += 运算符一文。

减法运算符 -

减法运算符 `-` 从其左侧操作数中减去其右侧操作数：

```
C#
```

```
Console.WriteLine(47 - 3);      // output: 44
Console.WriteLine(5 - 4.3);     // output: 0.7
Console.WriteLine(7.5m - 2.3m); // output: 5.2
```

还可以使用 `-` 运算符删除委托。 有关详细信息，请参阅 - 和 -= 运算符一文。

复合赋值

对于二元运算符 `op`，窗体的复合赋值表达式

```
C#

x op= y
```

等效于

```
C#

x = x op y
```

不同的是 `x` 只计算一次。

以下示例使用算术运算符演示了复合赋值的用法：

```
C#

int a = 5;
a += 9;
Console.WriteLine(a);  // output: 14

a -= 4;
Console.WriteLine(a);  // output: 10

a *= 2;
Console.WriteLine(a);  // output: 20

a /= 4;
Console.WriteLine(a);  // output: 5

a %= 3;
Console.WriteLine(a);  // output: 2
```

由于数值提升，`op` 运算的结果可能不会隐式转换为 `x` 的 `T` 类型。 在这种情况下，如果 `op` 是预定义的运算符并且运算的结果可以显式转换为 `x` 的类型 `T`，则形式为 `x op= y` 的复合赋值表达式等效于 `x = (T)(x op y)`，但 `x` 仅计算一次。 以下示例演示了该行为：

```
C#
```

```
byte a = 200;
byte b = 100;

var c = a + b;
Console.WriteLine(c.GetType());  // output: System.Int32
Console.WriteLine(c);  // output: 300

a += b;
Console.WriteLine(a);  // output: 44
```

你还可以使用 `+=` 和 `-=` 运算符分别订阅和取消订阅事件。 有关详细信息，请参阅如何订阅和取消订阅事件。

运算符优先级和关联性

以下列表对优先级由高到低的顺序对算术运算符进行排序：

- 后缀增量 `x++` 和减量 `x--` 运算符
- 前缀增量 `++x` 和减量 `--x` 以及一元 `+` 和 `-` 运算符
- 乘法 `*`、`/` 和 `%` 运算符
- 加法 `+` 和 `-` 运算符

二元算数运算符是左结合运算符。 也就是说，具有相同优先级的运算符按从左至右的顺序计算。

使用括号 `()` 更改由运算符优先级和关联性决定的计算顺序。

C#

```
Console.WriteLine(2 + 2 * 2);   // output: 6
Console.WriteLine((2 + 2) * 2); // output: 8

Console.WriteLine(9 / 5 / 2);   // output: 0
Console.WriteLine(9 / (5 / 2)); // output: 4
```

如需了解按优先级排序的完整 C# 运算符列表，请参阅 C# 运算符一文中的运算符优先级部分。

算术溢出和被零除

当算术运算的结果超出所涉及数值类型的可能有限值范围时，算术运算符的行为取决于其操作数的类型。

整数算术溢出

整数被零除总是引发 DivideByZeroException。

如果发生整数算术溢出，溢出检查上下文（可能已检查或未检查）将控制引发的行为：

- 在已检查的上下文中，如果在常数表达式中发生溢出，则会发生编译时错误。 否则，当在运行时执行此运算时，则引发 OverflowException。
- 在未检查的上下文中，会通过丢弃任何不适应目标类型的高序位来将结果截断。

在使用 checked 和 unchecked 语句时，可以使用 `checked` 和 `unchecked` 运算符来控制溢出检查上下文，在该上下文中将计算一个表达式：

C#

```
int a = int.MaxValue;
int b = 3;

Console.WriteLine(unchecked(a + b));  // output: -2147483646
try
{
    int d = checked(a + b);
}
catch(OverflowException)
{
    Console.WriteLine($"Overflow occurred when adding {a} to {b}.");
}
```

默认情况下，算术运算发生在 *unchecked* 上下文中。

浮点运算溢出

使用 `float` 和 `double` 类型的算术运算永远不会引发异常。 使用这些类型的算术运算的结果可能是表示无穷大和非数字的特殊值之一：

C#

```
double a = 1.0 / 0.0;
Console.WriteLine(a);                    // output: Infinity
Console.WriteLine(double.IsInfinity(a)); // output: True

Console.WriteLine(double.MaxValue + double.MaxValue); // output: Infinity

double b = 0.0 / 0.0;
Console.WriteLine(b);                // output: NaN
Console.WriteLine(double.IsNaN(b));  // output: True
```

对于 `decimal` 类型的操作数，算术溢出始终会引发 OverflowException。 被零除总是引发 DivideByZeroException。

舍入误差

由于实数和浮点运算的浮点表达形式的常规限制，在使用浮点类型的计算中可能会发生舍入误差。 也就是说，表达式得出的结果可能与预期的数学运算结果不同。 以下示例演示了几种此类情况：

C#

```
Console.WriteLine(.41f % .2f); // output: 0.00999999

double a = 0.1;
double b = 3 * a;
Console.WriteLine(b == 0.3);   // output: False
Console.WriteLine(b - 0.3);    // output: 5.55111512312578E-17

decimal c = 1 / 3.0m;
decimal d = 3 * c;
Console.WriteLine(d == 1.0m);  // output: False
Console.WriteLine(d);          // output: 0.99999999999999999999999999
```

有关详细信息，请参阅 System.Double、System.Single 或 System.Decimal 参考页上的注解。

运算符可重载性

用户定义类型可以重载一元（`++`、`--`、`+` 和 `-`）和二元（`*`、`/`、`%`、`+` 和 `-`）算术运算符。 重载二元运算符时，对应的复合赋值运算符也会隐式重载。 用户定义类型不能显式重载复合赋值运算符。

已检查的用户定义的运算符

从 C# 11 开始，重载算术运算符时，可以使用 `checked` 关键字定义该运算符的已检查版本。 下面的示例演示如何执行此操作：

C#

```
public record struct Point(int X, int Y)
{
    public static Point operator checked +(Point left, Point right)
    {
        checked
```

```
        {
            return new Point(left.X + right.X, left.Y + right.Y);
        }
    }

    public static Point operator +(Point left, Point right)
    {
        return new Point(left.X + right.X, left.Y + right.Y);
    }
}
```

定义已检查的运算符时，还必须定义不带 `checked` 修饰符的相应运算符。 checked 运算符在已检查的上下文中调用；不带 `checked` 修饰符的运算符在未检查的上下文中调用。 如果你只提供了不带 `checked` 修饰符的运算符，则会在 `checked` 和 `unchecked` 上下文中调用该运算符。

定义这两个版本的运算符时，预期其行为仅在操作结果过大而无法在结果类型中表示时有所不同，如下所示：

- 已检查的运算符引发 OverflowException。
- 不带 `checked` 修饰符的运算符返回表示截断结果的实例。

有关内置算术运算符行为差异的信息，请参阅算术溢出和除以零部分。

仅当重载以下任意运算符时，才能使用 `checked` 修饰符：

- 一元 `++`、`--` 和 `-` 运算符
- 二元 `*`、`/`、`+` 和 `-` 运算符
- 显式转换运算符

> **备注**
>
> `checked` 修饰符的存在不影响 checked 运算符正文中的溢出检查上下文。 默认上下文由 CheckForOverflowUnderflow 编译器选项的值定义。 使用 **checked 和 unchecked 语句**显式指定溢出检查上下文，如本部分开头的示例所示。

C# 语言规范

有关更多信息，请参阅 C# 语言规范的以下部分：

- 后缀增量和减量运算符
- 前缀增量和减量运算符
- 一元加运算符

- 一元减运算符
- 乘法运算符
- 除法运算符
- 余数运算符
- 加法运算符
- 减法运算符
- 复合赋值
- checked 和 unchecked 运算符
- 数值提升

另请参阅

- C# 参考
- C# 运算符和表达式
- System.Math
- System.MathF
- .NET 中的数字

布尔逻辑运算符 - AND、OR、NOT、XOR

项目 • 2024/01/10

逻辑布尔运算符使用 bool 操作数执行逻辑运算。 运算符包括一元逻辑非 (`!`)、二元逻辑 AND (`&`)、OR (`|`) 以及异或 (`^`)，二元条件逻辑 AND (`&&`) 和 OR (`||`)。

- 一元 !（逻辑非）运算符。
- 二元 &（逻辑与）、|（逻辑或）和 ^（逻辑异或）运算符。 这些运算符始终计算两个操作数。
- 二元 &&（条件逻辑与）和 ||（条件逻辑或）运算符。 这些运算符仅在必要时才计算右侧操作数。

对于整型数值类型的操作数，`&`、`|` 和 `^` 运算符执行位逻辑运算。 有关详细信息，请参阅位运算符和移位运算符。

逻辑非运算符 !

一元前缀 `!` 运算符计算操作数的逻辑非。 也就是说，如果操作数的计算结果为 `false`，它生成 `true`；如果操作数的计算结果为 `true`，它生成 `false`：

```
C#
```

```
bool passed = false;
Console.WriteLine(!passed);  // output: True
Console.WriteLine(!true);    // output: False
```

一元后缀 `!` 运算符为 null 包容运算符。

逻辑“与”运算符 &

`&` 运算符计算操作数的逻辑与。 如果 `x` 和 `y` 的计算结果都为 `true`，则 `x & y` 的结果为 `true`。 否则，结果为 `false`。

`&` 运算符总是计算两个操作数。 当左侧操作数的计算结果为 `false` 时，运算结果为 `false`，而与右侧操作数的值无关。 但是，即使在这种情况下，也会计算右侧操作数。

在下面的示例中，`&` 运算符的右侧操作数是方法调用，无论左侧操作数的值如何，都会执行方法调用：

C#

```
bool SecondOperand()
{
    Console.WriteLine("Second operand is evaluated.");
    return true;
}

bool a = false & SecondOperand();
Console.WriteLine(a);
// Output:
// Second operand is evaluated.
// False

bool b = true & SecondOperand();
Console.WriteLine(b);
// Output:
// Second operand is evaluated.
// True
```

条件逻辑 AND 运算符`&&`也计算操作数的逻辑 AND，但如果左侧操作数的计算结果为`false`，它就不会计算右侧操作数。

对于整型数值类型的操作数，`&`运算符计算其操作数的位逻辑 AND。 一元`&`运算符是 address-of 运算符。

逻辑异或运算符 ^

`^`运算符计算操作数的逻辑异或（亦称为“逻辑 XOR”）。 如果`x`计算结果为`true`且`y`计算结果为`false`，或者`x`计算结果为`false`且`y`计算结果为`true`，那么`x ^ y`的结果为`true`。 否则，结果为`false`。 也就是说，对于`bool`操作数，`^`运算符的计算结果与不等运算符`!=`相同。

C#

```
Console.WriteLine(true ^ true);    // output: False
Console.WriteLine(true ^ false);   // output: True
Console.WriteLine(false ^ true);   // output: True
Console.WriteLine(false ^ false);  // output: False
```

对于整型数值类型的操作数，`^`运算符计算其操作数的位逻辑异或。

逻辑或运算符 |

`|` 运算符计算操作数的逻辑或。 如果 `x` 或 `y` 的计算结果为 `true`，则 `x | y` 的结果为 `true`。 否则，结果为 `false`。

`|` 运算符总是计算两个操作数。 当左侧操作数的计算结果为 `true` 时，运算结果为 `true`，而与右侧操作数的值无关。 但是，即使在这种情况下，也会计算右侧操作数。

在下面的示例中，`|` 运算符的右侧操作数是方法调用，无论左侧操作数的值如何，都会执行方法调用：

C#

```
bool SecondOperand()
{
    Console.WriteLine("Second operand is evaluated.");
    return true;
}

bool a = true | SecondOperand();
Console.WriteLine(a);
// Output:
// Second operand is evaluated.
// True

bool b = false | SecondOperand();
Console.WriteLine(b);
// Output:
// Second operand is evaluated.
// True
```

条件逻辑 OR 运算符 `||` 也计算操作数的逻辑 OR，但如果左侧操作数的计算结果为 `true`，它就不会计算右侧操作数。

对于整型数值类型的操作数，`|` 运算符计算其操作数的位逻辑 OR。

条件逻辑“与”运算符 &&

条件逻辑与运算符 `&&`（亦称为“短路”逻辑与运算符）计算操作数的逻辑与。 如果 `x` 和 `y` 的计算结果都为 `true`，则 `x && y` 的结果为 `true`。 否则，结果为 `false`。 如果 `x` 的计算结果为 `false`，则不计算 `y`。

在下面的示例中，`&&` 运算符的右侧操作数是方法调用，如果左侧操作数的计算结果为 `false`，就不执行方法调用：

C#

```
bool SecondOperand()
{
    Console.WriteLine("Second operand is evaluated.");
    return true;
}

bool a = false && SecondOperand();
Console.WriteLine(a);
// Output:
// False

bool b = true && SecondOperand();
Console.WriteLine(b);
// Output:
// Second operand is evaluated.
// True
```

逻辑与运算符`&`也计算操作数的逻辑与，但始终计算两个操作数。

条件逻辑或运算符 ||

条件逻辑或运算符 `||`（亦称为"短路"逻辑或运算符）计算操作数的逻辑或。 如果 `x` 或 `y` 的计算结果为 `true`，则 `x || y` 的结果为 `true`。 否则，结果为 `false`。 如果 `x` 的计算结果为 `true`，则不计算 `y`。

在下面的示例中，`||` 运算符的右侧操作数是方法调用，如果左侧操作数的计算结果为 `true`，就不执行方法调用：

C#

```
bool SecondOperand()
{
    Console.WriteLine("Second operand is evaluated.");
    return true;
}

bool a = true || SecondOperand();
Console.WriteLine(a);
// Output:
// True

bool b = false || SecondOperand();
Console.WriteLine(b);
// Output:
// Second operand is evaluated.
// True
```

逻辑或运算符 `|` 也计算操作数的逻辑或，但始终计算两个操作数。

可以为 null 的布尔逻辑运算符

对于 `bool?` 操作数，`&`（逻辑与）和 `|`（逻辑或）运算符支持三值逻辑，如下所示：

- 仅当其两个操作数的计算结果都为 `true` 时，`&` 运算符才生成 `true`。 如果 `x` 或 `y` 的计算结果为 `false`，则 `x & y` 将生成 `false`（即使另一个操作数的计算结果为 `null`）。 否则，`x & y` 的结果为 `null`。
- 仅当其两个操作数的计算结果都为 `false` 时，`|` 运算符才生成 `false`。 如果 `x` 或 `y` 的计算结果为 `true`，则 `x | y` 将生成 `true`（即使另一个操作数的计算结果为 `null`）。 否则，`x | y` 的结果为 `null`。

下表显示了该语义：

展开表

x	y	x 和 y	x|y
true	true	true	true
true	false	false	true
true	null	null	true
false	true	false	true
false	false	false	false
false	null	false	null
null	true	null	true
null	false	false	null
Null	Null	Null	null

这些运算符的行为不同于值类型可以为 null 的典型运算符行为。 通常情况下，为值类型的操作数定义的运算符也能与值类型可以为 null 的相应操作数一起使用。 如果其任一操作数的计算结果为 `null`，此类运算符都会生成 `null`。 不过，即使操作数之一的计算结果为 `null`，`&` 和 `|` 运算符也可以生成非 null。 有关值类型可为空的运算符行为的详细信息，请参阅可为空的值类型一文的提升的运算符部分。

此外，你还可以将 `!` 和 `^` 运算符与 `bool?` 操作数结合使用，如下例所示：

C#

```
bool? test = null;
Display(!test);         // output: null
Display(test ^ false);  // output: null
Display(test ^ null);   // output: null
Display(true ^ null);   // output: null

void Display(bool? b) => Console.WriteLine(b is null ? "null" :
b.Value.ToString());
```

条件逻辑运算符 `&&` 和 `||` 不支持 `bool?` 操作数。

复合赋值

对于二元运算符 `op`，窗体的复合赋值表达式

C#

```
x op= y
```

等效于

C#

```
x = x op y
```

不同的是 `x` 只计算一次。

`&`、`|` 和 `^` 运算符支持复合赋值，如下面的示例所示：

C#

```
bool test = true;
test &= false;
Console.WriteLine(test);  // output: False

test |= true;
Console.WriteLine(test);  // output: True

test ^= false;
Console.WriteLine(test);  // output: True
```

ⓘ 备注

条件逻辑运算符 `&&` 和 `||` 不支持复合赋值。

运算符优先级

以下列表按优先级从高到低的顺序对逻辑运算符进行排序：

- 逻辑非运算符 `!`
- 逻辑与运算符 `&`
- 逻辑异或运算符 `^`
- 逻辑或运算符 `|`
- 条件逻辑与运算符 `&&`
- 条件逻辑或运算符 `||`

使用括号 `()` 可以更改运算符优先级决定的计算顺序：

C#

```
Console.WriteLine(true | true & false);   // output: True
Console.WriteLine((true | true) & false); // output: False

bool Operand(string name, bool value)
{
    Console.WriteLine($"Operand {name} is evaluated.");
    return value;
}

var byDefaultPrecedence = Operand("A", true) || Operand("B", true) &&
Operand("C", false);
Console.WriteLine(byDefaultPrecedence);
// Output:
// Operand A is evaluated.
// True

var changedOrder = (Operand("A", true) || Operand("B", true)) &&
Operand("C", false);
Console.WriteLine(changedOrder);
// Output:
// Operand A is evaluated.
// Operand C is evaluated.
// False
```

要了解按优先级排序的完整 C# 运算符列表，请参阅 C# 运算符一文中的运算符优先级部分。

运算符可重载性

用户定义类型可以重载 `!`、`&`、`|` 和 `^` 运算符。 重载二元运算符时，对应的复合赋值运算符也会隐式重载。 用户定义类型不能显式重载复合赋值运算符。

用户定义类型无法重载条件逻辑运算符 `&&` 和 `||`。 不过，如果用户定义类型以某种方式重载 true 和 false 运算符以及 `&` 或 `|` 运算符，可以对相应类型的操作数执行 `&&` 或 `||` 运算。 有关详细信息，请参阅 C# 语言规范的用户定义条件逻辑运算符部分。

C# 语言规范

有关更多信息，请参阅 C# 语言规范的以下部分：

- 逻辑非运算符
- 逻辑运算符
- 条件逻辑运算符
- 复合赋值

请参阅

- C# 参考
- C# 运算符和表达式
- 位运算符和移位运算符

在 GitHub 上与我们协作

可以在 GitHub 上找到此内容的源，还可以在其中创建和查看问题和拉取请求。 有关详细信息，请参阅参与者指南。

.NET 反馈

.NET 是一个开放源代码项目。 选择一个链接以提供反馈：

提出文档问题

提供产品反馈

位运算符和移位运算符（C# 参考）

项目 • 2023/04/03

位运算符和移位运算符包括一元位补、二进制左移和右移、无符号右移、二进制逻辑 AND、OR 和异或运算符。 这些操作数采用整型数值类型或字符型操作数。

- 一元 ~（按位求补）运算符
- 二进制 <<（左移）、>>（右移）和 >>>（无符号右移）运算符
- 二进制 &（逻辑 AND）、|（逻辑 OR）和 ^（逻辑异或）运算符

这些运算符是针对 `int`、`uint`、`long` 和 `ulong` 类型定义的。 如果两个操作数都是其他整数类型（`sbyte`、`byte`、`short`、`ushort` 或 `char`），它们的值将转换为 `int` 类型，这也是一个运算的结果类型。 如果操作数是不同的整数类型，它们的值将转换为最接近的包含整数类型。 有关详细信息，请参阅 C# 语言规范的数值提升部分。 复合运算符（如 `>>=`）不会将其参数转换为 `int`，也不会具有结果类型 `int`。

`&`、`|` 和 `^` 运算符也是为 `bool` 类型的操作数定义的。 有关详细信息，请参阅布尔逻辑运算符。

位运算和移位运算永远不会导致溢出，并且不会在已检查和未检查的上下文中产生相同的结果。

按位求补运算符 ~

`~` 运算符通过反转每个位产生其操作数的按位求补：

```
C#
```

```
uint a = 0b_0000_1111_0000_1111_0000_1111_0000_1100;
uint b = ~a;
Console.WriteLine(Convert.ToString(b, toBase: 2));
// Output:
// 11110000111100001111000011110011
```

也可以使用 `~` 符号来声明终结器。 有关详细信息，请参阅终结器。

左移位运算符 <<

`<<` 运算符将其左侧操作数向左移动右侧操作数定义的位数。 有关右侧操作数如何定义移位计数的信息，请参阅移位运算符的移位计数部分。

左移运算会放弃超出结果类型范围的高阶位，并将低阶空位位置设置为零，如以下示例所示：

C#

```
uint x = 0b_1100_1001_0000_0000_0000_0000_0001_0001;
Console.WriteLine($"Before: {Convert.ToString(x, toBase: 2)}");

uint y = x << 4;
Console.WriteLine($"After:  {Convert.ToString(y, toBase: 2)}");
// Output:
// Before: 11001001000000000000000000010001
// After:  10010000000000000000000100010000
```

由于移位运算符仅针对 `int`、`uint`、`long` 和 `ulong` 类型定义，因此运算的结果始终包含至少 32 位。 如果左侧操作数是其他整数类型（`sbyte`、`byte`、`short`、`ushort` 或 `char`），则其值将转换为 `int` 类型，如以下示例所示：

C#

```
byte a = 0b_1111_0001;

var b = a << 8;
Console.WriteLine(b.GetType());
Console.WriteLine($"Shifted byte: {Convert.ToString(b, toBase: 2)}");
// Output:
// System.Int32
// Shifted byte: 1111000100000000
```

右移位运算符 >>

`>>` 运算符将其左侧操作数向右移动右侧操作数定义的位数。 有关右侧操作数如何定义移位计数的信息，请参阅移位运算符的移位计数部分。

右移位运算会放弃低阶位，如以下示例所示：

C#

```
uint x = 0b_1001;
Console.WriteLine($"Before: {Convert.ToString(x, toBase: 2), 4}");

uint y = x >> 2;
Console.WriteLine($"After:  {Convert.ToString(y, toBase: 2).PadLeft(4, '0'),
4}");
// Output:
// Before: 1001
// After:  0010
```

高顺序空位位置是根据左侧操作数类型设置的，如下所示：

- 如果左侧操作数的类型是 `int` 或 `long`，则右移运算符将执行 算术移位：左侧操作数的最高有效位（符号位）的值将传播到高顺序空位位置。 也就是说，如果左侧操作数为非负，高顺序空位位置设置为零，如果为负，则将该位置设置为 1。

  C#

  ```
  int a = int.MinValue;
  Console.WriteLine($"Before: {Convert.ToString(a, toBase: 2)}");

  int b = a >> 3;
  Console.WriteLine($"After:  {Convert.ToString(b, toBase: 2)}");
  // Output:
  // Before: 10000000000000000000000000000000
  // After:  11110000000000000000000000000000
  ```

- 如果左侧操作数的类型是 `uint` 或 `ulong`，则右移运算符执行逻辑移位：高顺序空位位置始终设置为零。

  C#

  ```
  uint c = 0b_1000_0000_0000_0000_0000_0000_0000_0000;
  Console.WriteLine($"Before: {Convert.ToString(c, toBase: 2), 32}");

  uint d = c >> 3;
  Console.WriteLine($"After:  {Convert.ToString(d, toBase: 2).PadLeft(32,
  '0'), 32}");
  // Output:
  // Before: 10000000000000000000000000000000
  // After:  00010000000000000000000000000000
  ```

> ⓘ **备注**
>
> 使用**无符号右移运算符**可以对带符号整数类型的操作数执行逻辑移位。 这优于将左侧操作数强制转换为无符号类型，然后将移位操作的结果强制转换回带符号类型。

无符号右移运算符 >>>

在 C# 11 及更高版本中可用，`>>>` 运算符将其左侧操作数向右移动其右侧操作数定义的位数。 有关右侧操作数如何定义移位计数的信息，请参阅移位运算符的移位计数部分。

`>>>` 运算符始终执行逻辑移位。 也就是说，无论左侧操作数的类型如何，高顺序空位位置始终设置为零。 如果左侧操作数是带符号类型，`>>` 运算符将执行算术移位（即，最高有效位的值传播到高顺序空位位置）。 以下示例演示了对于负左操作数，`>>` 和 `>>>` 运算符之间的差别：

C#

```
int x = -8;
Console.WriteLine($"Before:    {x,11}, hex: {x,8:x}, binary:
{Convert.ToString(x, toBase: 2), 32}");

int y = x >> 2;
Console.WriteLine($"After  >>: {y,11}, hex: {y,8:x}, binary:
{Convert.ToString(y, toBase: 2), 32}");

int z = x >>> 2;
Console.WriteLine($"After >>>: {z,11}, hex: {z,8:x}, binary:
{Convert.ToString(z, toBase: 2).PadLeft(32, '0'), 32}");
// Output:
// Before:             -8, hex: fffffff8, binary:
11111111111111111111111111111000
// After  >>:          -2, hex: fffffffe, binary:
11111111111111111111111111111110
// After >>>:  1073741822, hex: 3ffffffe, binary:
00111111111111111111111111111110
```

逻辑与运算符 &

`&` 运算符计算其整型操作数的位逻辑 AND：

C#

```
uint a = 0b_1111_1000;
uint b = 0b_1001_1101;
uint c = a & b;
Console.WriteLine(Convert.ToString(c, toBase: 2));
// Output:
// 10011000
```

对于 `bool` 操作数，`&` 运算符对其操作数执行逻辑 AND 运算。 一元 `&` 运算符是 address-of 运算符。

逻辑异或运算符 ^

`^` 运算符计算其整型操作数的位逻辑异或，也称为位逻辑 XOR：

```C#
uint a = 0b_1111_1000;
uint b = 0b_0001_1100;
uint c = a ^ b;
Console.WriteLine(Convert.ToString(c, toBase: 2));
// Output:
// 11100100
```

对于 `bool` 操作数，`^` 运算符对其操作数执行逻辑异或运算。

逻辑或运算符 |

`|` 运算符计算其整型操作数的位逻辑 OR：

```C#
uint a = 0b_1010_0000;
uint b = 0b_1001_0001;
uint c = a | b;
Console.WriteLine(Convert.ToString(c, toBase: 2));
// Output:
// 10110001
```

对于 `bool` 操作数，`|` 运算符对其操作数执行逻辑 OR 运算。

复合赋值

对于二元运算符 `op`，窗体的复合赋值表达式

```C#
x op= y
```

等效于

```C#
x = x op y
```

不同的是 `x` 只计算一次。

以下示例演示了使用位运算符和移位运算符的复合赋值的用法：

```
```

C#

```
uint INITIAL_VALUE = 0b_1111_1000;

uint a = INITIAL_VALUE;
a &= 0b_1001_1101;
Display(a);  // output: 10011000

a = INITIAL_VALUE;
a |= 0b_0011_0001;
Display(a);  // output: 11111001

a = INITIAL_VALUE;
a ^= 0b_1000_0000;
Display(a);  // output: 01111000

a = INITIAL_VALUE;
a <<= 2;
Display(a);  // output: 1111100000

a = INITIAL_VALUE;
a >>= 4;
Display(a);  // output: 00001111

a = INITIAL_VALUE;
a >>>= 4;
Display(a);  // output: 00001111

void Display(uint x) => Console.WriteLine($"{Convert.ToString(x, toBase:
2).PadLeft(8, '0'), 8}");
```

由于数值提升，`op` 运算的结果可能不会隐式转换为 `x` 的 `T` 类型。 在这种情况下，如果 `op` 是预定义的运算符并且运算的结果可以显式转换为 `x` 的类型 `T`，则形式为 `x op= y` 的复合赋值表达式等效于 `x = (T)(x op y)`，但 `x` 仅计算一次。 以下示例演示了该行为：

C#

```
byte x = 0b_1111_0001;

int b = x << 8;
Console.WriteLine($"{Convert.ToString(b, toBase: 2)}");  // output:
1111000100000000

x <<= 8;
Console.WriteLine(x);  // output: 0
```

运算符优先级

以下列表按位运算符和移位运算符从最高优先级到最低优先级排序：

- 按位求补运算符 `~`
- 移位运算符 `<<`、`>>` 和 `>>>`
- 逻辑与运算符 `&`
- 逻辑异或运算符 `^`
- 逻辑或运算符 `|`

使用括号 `()` 可以更改运算符优先级决定的计算顺序：

C#

```
uint a = 0b_1101;
uint b = 0b_1001;
uint c = 0b_1010;

uint d1 = a | b & c;
Display(d1);  // output: 1101

uint d2 = (a | b) & c;
Display(d2);  // output: 1000

void Display(uint x) => Console.WriteLine($"{Convert.ToString(x, toBase: 2),
4}");
```

要了解按优先级排序的完整 C# 运算符列表，请参阅 C# 运算符一文中的运算符优先级部分。

移位运算符的移位计数

对于内置移位运算符 `<<`、`>>` 和 `>>>`，右侧操作数的类型必须为 `int`，或具有预定义隐式数值转换为 `int` 的类型。

对于 `x << count`、`x >> count` 和 `x >>> count` 表达式，实际移位计数取决于 `x` 的类型，如下所示：

- 如果 `x` 的类型为 `int` 或 `uint`，则移位计数由右侧操作数的低阶五位定义。 也就是说，移位计数通过 `count & 0x1F`（或 `count & 0b_1_1111`）计算得出。
- 如果 `x` 的类型为 `long` 或 `ulong`，则移位计数由右侧操作数的低阶六位定义。 也就是说，移位计数通过 `count & 0x3F`（或 `count & 0b_11_1111`）计算得出。

以下示例演示了该行为：

C#

```
int count1 = 0b_0000_0001;
int count2 = 0b_1110_0001;

int a = 0b_0001;
Console.WriteLine($"{a} << {count1} is {a << count1}; {a} << {count2} is {a
<< count2}");
// Output:
// 1 << 1 is 2; 1 << 225 is 2

int b = 0b_0100;
Console.WriteLine($"{b} >> {count1} is {b >> count1}; {b} >> {count2} is {b
>> count2}");
// Output:
// 4 >> 1 is 2; 4 >> 225 is 2

int count = -31;
int c = 0b_0001;
Console.WriteLine($"{c} << {count} is {c << count}");
// Output:
// 1 << -31 is 2
```

备注

如前例所示，即使右侧操作符的值大于左侧操作符中的位数，移位运算的结果也不可为零。

枚举逻辑运算符

所有枚举类型还支持 `~`、`&`、`|` 和 `^` 运算符。 对于相同枚举类型的操作数，对底层整数类型的相应值执行逻辑运算。 例如，对于具有底层类型 `U` 的枚举类型 `T` 的任何 `x` 和 `y`，`x & y` 表达式生成与 `(T)((U)x & (U)y)` 表达式相同的结果。

通常使用具有枚举类型的位逻辑运算符，该枚举类型使用 Flags 特性定义。 有关详细信息，请参阅枚举类型一文的作为位标记的枚举类型部分。

运算符可重载性

用户定义的类型可以重载 `~`、`<<`、`>>`、`>>>`、`&`、`|` 和 `^` 运算符。 重载二元运算符时，对应的复合赋值运算符也会隐式重载。 用户定义类型不能显式重载复合赋值运算符。

如果用户定义类型 `T` 重载了 `<<`、`>>` 或 `>>>` 运算符，则左侧操作数的类型必须为 `T`。 在 C# 10 及更早版本中，右侧操作数的类型必须为 `int`；从 C# 11 开始，重载移位运算符的右侧操作数的类型可以是任意类型。

C# 语言规范

有关更多信息，请参阅 C# 语言规范的以下部分：

- 按位求补运算符
- 移位运算符
- 逻辑运算符
- 复合赋值
- 数值提升
- C# 11 - 宽松的移位要求
- C# 11 - 逻辑右移运算符

另请参阅

- C# 参考
- C# 运算符和表达式
- Boolean 逻辑运算符

集合表达式 - C# 语言参考

项目 • 2024/03/18

可以使用*集合表达式*来创建常见的集合值。 *集合表达式*是一种简洁的语法，在计算时，可以分配给许多不同的集合类型。 集合表达式在 `[` 和 `]` 括号之间包含元素的序列。 以下示例声明 `string` 元素的 System.Span<T>，并将其初始化为星期几：

C#

```
Span<string> weekDays = ["Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat"];
foreach (var day in weekDays)
{
    Console.WriteLine(day);
}
```

*集合表达式*可以转换为许多不同的集合类型。 第一个示例演示了如何使用集合表达式初始化变量。 以下代码显示了可以使用集合表达式的其他许多位置：

C#

```
// Initialize private field:
private static readonly ImmutableArray<string> _months = ["Jan", "Feb",
"Mar", "Apr", "May", "Jun", "Jul", "Aug", "Sep", "Oct", "Nov", "Dec"];

// property with expression body:
public IEnumerable<int> MaxDays =>
    [31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31];

public int Sum(IEnumerable<int> values) =>
    values.Sum();

public void Example()
{
    // As a parameter:
    int sum = Sum([1, 2, 3, 4, 5]);
}
```

不能使用需要编译时常量（例如初始化常量）或方法参数的默认值的集合表达式。

上述两个示例都使用常量作为集合表达式的元素。 还可以对元素使用变量，如以下示例所示：

C#

```
string hydrogen = "H";
string helium = "He";
string lithium = "Li";
```

```
string beryllium = "Be";
string boron = "B";
string carbon = "C";
string nitrogen = "N";
string oxygen = "O";
string fluorine = "F";
string neon = "Ne";
string[] elements = [hydrogen, helium, lithium, beryllium, boron, carbon,
nitrogen, oxygen, fluorine, neon];
foreach (var element in elements)
{
    Console.WriteLine(element);
}
```

分布元素

使用*分布元素* `..` 在集合表达式中使用内联集合值。 以下示例通过将元音集合、同音集合和字母"y"组合在一起，为完整字母创建一个集合，该集合可以是以下任一类型：

C#

```
string[] vowels = ["a", "e", "i", "o", "u"];
string[] consonants = ["b", "c", "d", "f", "g", "h", "j", "k", "l", "m",
                       "n", "p", "q", "r", "s", "t", "v", "w", "x", "z"];
string[] alphabet = [.. vowels, .. consonants, "y"];
```

求值时，分布元素 `..vowels` 产生五个元素：`"a"`、`"e"`、`"i"`、`"o"` 和 `"u"`。 分布元素 `..consonants` 产生 20 个元素，即 `consonants` 数组中的数字。 分布元素中的变量必须使用 foreach 语句进行枚举。 如前面的示例所示，可以将分布元素与集合表达式中的单个元素组合在一起。

转换

*集合表达式*可以转换为不同的集合类型，包括：

- System.Span<T> 和 System.ReadOnlySpan<T>。
- 数组。
- 具有*创建*方法的任何类型，其参数类型为 `ReadOnlySpan<T>`，其中存在从集合表达式类型到 `T` 的隐式转换。
- 支持集合初始值设定项的任何类型，例如 System.Collections.Generic.List<T>。 通常，此要求意味着类型支持 System.Collections.Generic.IEnumerable<T>，并且有一个可访问的 `Add` 方法将项添加到集合中。 必须有从集合表达式元素类型到集合元素类型的隐式转换。 对于分布元素，必须有从分布元素类型到集合元素类型的隐式转换。

- 下列任何接口：
  - System.Collections.Generic.IEnumerable<T>。
  - System.Collections.Generic.IReadOnlyCollection<T>。
  - System.Collections.Generic.IReadOnlyList<T>。
  - System.Collections.Generic.ICollection<T>。
  - System.Collections.Generic.IList<T>。

> **重要**
>
> 无论转换的目标类型如何，集合表达式总是会创建一个包含集合表达式中的所有元素的集合。 例如，当转换的目标为 System.Collections.Generic.IEnumerable<T> 时，生成的代码将计算集合表达式并将结果存储在内存中集合中。
>
> 这种行为不同于 LINQ；在 LINQ 中，在枚举序列之前可能不会将此序列实例化。 不能使用集合表达式生成不会枚举的无限序列。

编译器使用静态分析来确定使用集合表达式声明的集合的最高性能方法。 例如，如果初始化后不会修改目标，则可以将空集合表达式 `[]` 实现为 Array.Empty<T>()。 当目标为 System.Span<T> 或 System.ReadOnlySpan<T> 时，存储可能是堆栈分配的。 集合表达式功能规范指定编译器必须遵循的规则。

许多 API 使用多个集合类型作为参数进行重载。 由于集合表达式可以转换为许多不同的表达式类型，因此这些 API 可能需要对集合表达式进行强制转换以指定正确的转换。 以下转换规则解决了一些歧义：

- 转换为 Span<T>、ReadOnlySpan<T> 或其他 ref struct 类型比转换为非 ref 结构类型更好。
- 转换为非接口类型比转换为接口类型更好。

当集合表达式转换为 `Span` 或 `ReadOnlySpan` 时，范围对象的*安全上下文*取自范围中包含的所有元素*安全上下文*。 有关详细规则，请参阅集合表达式规范。

集合生成器

集合表达式适用于行为良好的任何集合类型。 行为良好的集合具有以下特征：

- 可计数集合上的 `Count` 或 `Length` 值生成与枚举时元素数相同的值。
- System.Collections.Generic 命名空间中的类型假定为无副作用。 因此，编译器可以优化这类类型可能用作中间值，但不会被公开的场景。
- 如果调用集合中某些适用的 `.AddRange(x)` 成员，那么得到的最终值与循环访问 `x` 并使用 `.Add` 将其所有枚举值分别添加到该集合所得到的值相同。

.NET 运行时中的所有集合类型都行为良好。

> ⚠ **警告**
>
> 如果自定义集合类型行为不佳，则未定义将该集合类型与集合表达式一起使用时的行为。

类型通过编写 `Create()` 方法并对集合类型应用 System.Runtime.CompilerServices.CollectionBuilderAttribute 来指示生成器方法，选择使用集合表达式支持。 例如，请考虑使用固定长度缓冲区 80 个字符的应用程序。 该类可能类似于以下代码：

C#

```
public class LineBuffer : IEnumerable<char>
{
    private readonly char[] _buffer = new char[80];

    public LineBuffer(ReadOnlySpan<char> buffer)
    {
        int number = (_buffer.Length < buffer.Length) ? _buffer.Length :
buffer.Length;
        for (int i = 0; i < number; i++)
        {
            _buffer[i] = buffer[i];
        }
    }

    public IEnumerator<char> GetEnumerator() => _buffer.AsEnumerable<char>
().GetEnumerator();
    IEnumerator IEnumerable.GetEnumerator() => _buffer.GetEnumerator();

    // etc
}
```

你想要将其与集合表达式一起使用，如以下示例所示：

C#

```
LineBuffer line = ['H', 'e', 'l', 'l', 'o', ' ', 'W', 'o', 'r', 'l', 'd',
'!'];
```

`LineBuffer` 类型实现 `IEnumerable<char>`，因此编译器将其识别为 `char` 项的集合。 实现的 System.Collections.Generic.IEnumerable<T> 接口的类型参数指示元素类型。 需要向应用程序添加两个内容才能将集合表达式分配给 `LineBuffer` 对象。 首先，需要创建一个包含 `Create` 方法的类：

C#

```
internal static class LineBufferBuilder
{
    internal static LineBuffer Create(ReadOnlySpan<char> values) => new
LineBuffer(values);
}
```

`Create` 方法必须返回 `LineBuffer` 对象，并且必须采用 `ReadOnlySpan<char>` 类型的单个参数。 `ReadOnlySpan` 的类型参数必须与集合的元素类型匹配。 返回泛型集合的生成器方法将泛型 `ReadOnlySpan<T>` 作为其参数。 该方法必须可访问且为 `static`。

最后，必须将 CollectionBuilderAttribute 添加到 `LineBuffer` 类声明：

C#

```
[CollectionBuilder(typeof(LineBufferBuilder), "Create")]
```

第一个参数提供*生成器*类的名称。 第二个特性提供生成器方法的名称。

在 GitHub 上与我们协作

可以在 GitHub 上找到此内容的源，还可以在其中创建和查看问题和拉取请求。 有关详细信息，请参阅参与者指南。

.NET 反馈

.NET 是一个开放源代码项目。 选择一个链接以提供反馈：

提出文档问题

提供产品反馈

相等运算符 - 测试两个对象是否相等

项目 • 2024/10/26

== （相等） 和 != （不等） 运算符检查其操作数是否相等。 当内容相等时，值类型相等。 当两个变量引用同一存储时，引用类型相等。

相等运算符 ==

如果操作数相等，等于运算符 `==` 返回 `true`，否则返回 `false`。

值类型的相等性

如果内置值类型的值相等，则其操作数相等：

```C#
int a = 1 + 2 + 3;
int b = 6;
Console.WriteLine(a == b);  // output: True

char c1 = 'a';
char c2 = 'A';
Console.WriteLine(c1 == c2);  // output: False
Console.WriteLine(c1 == char.ToLower(c2));  // output: True
```

> **备注**
>
> 对于 `==`、`<`、`>`、`<=` 和 `>=` 运算符，如果任何操作数不是数字（Double.NaN 或 Single.NaN），则运算的结果为 `false`。 这意味着 `NaN` 值不大于、小于或等于任何其他 `double`（或 `float`）值，包括 `NaN`。 有关更多信息和示例，请参阅 Double.NaN 或 Single.NaN 参考文章。

如果基本整数类型的相应值相等，则相同枚举类型的两个操作数相等。

用户定义的 struct 类型默认情况下不支持 `==` 运算符。 要支持 `==` 运算符，用户定义的结构必须重载它。

`==` 和 `!=` 运算符由 C# 元组支持。 有关详细信息，请参阅元组类型一文的元组相等部分。

引用类型的相等性

默认情况下，如果两个非记录引用类型操作符引用同一对象，则这两个操作符相等：

C#

```
public class ReferenceTypesEquality
{
    public class MyClass
    {
        private int id;

        public MyClass(int id) => this.id = id;
    }

    public static void Main()
    {
        var a = new MyClass(1);
        var b = new MyClass(1);
        var c = a;
        Console.WriteLine(a == b);  // output: False
        Console.WriteLine(a == c);  // output: True
    }
}
```

如示例所示，默认情况下，用户定义的引用类型支持 `==` 运算符。 但是，引用类型可重载 `==` 运算符。 如果引用类型重载 `==` 运算符，使用 Object.ReferenceEquals 方法来检查该类型的两个引用是否引用同一对象。

记录类型相等性

记录类型支持 `==` 和 `!=` 运算符，这些运算符默认提供值相等性语义。 也就是说，当两个记录操作数都是 `null` 所有字段的对应值或自动实现的属性相等时，两个记录操作数是相等的。

C#

```
public class RecordTypesEquality
{
    public record Point(int X, int Y, string Name);
    public record TaggedNumber(int Number, List<string> Tags);

    public static void Main()
    {
        var p1 = new Point(2, 3, "A");
        var p2 = new Point(1, 3, "B");
        var p3 = new Point(2, 3, "A");

        Console.WriteLine(p1 == p2);  // output: False
        Console.WriteLine(p1 == p3);  // output: True
```

```
        var n1 = new TaggedNumber(2, new List<string>() { "A" });
        var n2 = new TaggedNumber(2, new List<string>() { "A" });
        Console.WriteLine(n1 == n2);  // output: False
    }
}
```

如前面的示例所示，对于非记录引用类型成员，比较的是它们的引用值，而不是所引用的实例。

字符串相等性

如果两个字符串均为 `null` 或者两个字符串实例具有相等长度且在每个字符位置有相同字符，则这两个字符串操作数相等：

C#

```
string s1 = "hello!";
string s2 = "HeLLo!";
Console.WriteLine(s1 == s2.ToLower());  // output: True

string s3 = "Hello!";
Console.WriteLine(s1 == s3);  // output: False
```

字符串相等比较是区分大小写的序号比较。 有关字符串比较的详细信息，请参阅如何在 C# 中比较字符串。

委托相等

若两个运行时间类型相同的委托操作数均为 `null`，或其调用列表长度系统且在每个位置具有相同的条目，则二者相等：

C#

```
Action a = () => Console.WriteLine("a");

Action b = a + a;
Action c = a + a;
Console.WriteLine(object.ReferenceEquals(b, c));  // output: False
Console.WriteLine(b == c);  // output: True
```

有关详细信息，请参阅 C# 语言规范中的委托相等运算符部分。

通过计算语义上相同的 Lambda 表达式生成的委托不相等，如以下示例所示：

C#

```
Action a = () => Console.WriteLine("a");
Action b = () => Console.WriteLine("a");

Console.WriteLine(a == b);  // output: False
Console.WriteLine(a + b == a + b);  // output: True
Console.WriteLine(b + a == a + b);  // output: False
```

不等运算符 !=

如果操作数不相等，不等于运算符 `!=` 返回 `true`，否则返回 `false`。 对于内置类型的操作数，表达式 `x != y` 生成与表达式 `!(x == y)` 相同的结果。 有关类型相等性的更多信息，请参阅相等运算符部分。

下面的示例演示 `!=` 运算符的用法：

C#

```
int a = 1 + 1 + 2 + 3;
int b = 6;
Console.WriteLine(a != b);  // output: True

string s1 = "Hello";
string s2 = "Hello";
Console.WriteLine(s1 != s2);  // output: False

object o1 = 1;
object o2 = 1;
Console.WriteLine(o1 != o2);  // output: True
```

运算符可重载性

用户定义类型可以重载 `==` 和 `!=` 运算符。 如果某类型重载这两个运算符之一，它还必须重载另一个运算符。

记录类型不能显式重载 `==` 和 `!=` 运算符。 如果需要更改记录类型 `T` 的 `==` 和 `!=` 运算符的行为，请使用以下签名实现 IEquatable<T>.Equals 方法：

C#

```
public virtual bool Equals(T? other);
```

C# 语言规范

有关详细信息，请参阅 C# 语言规范中的关系和类型测试运算符部分。

有关记录类型相等性的详细信息，请参阅记录功能建议附注中的相等性成员部分。

另请参阅

- C# 运算符和表达式
- System.IEquatable<T>
- Object.Equals
- Object.ReferenceEquals
- 相等性比较
- 比较运算符

比较运算符（C# 参考）

项目 • 2024/04/02

< （小于）、> （大于）、<= （小于或等于）和 >= （大于或等于）比较（也称为关系）运算符比较其操作数。 所有整型和浮点数值类型都支持这些运算符。

> **备注**
>
> 对于 `==`、`<`、`>`、`<=` 和 `>=` 运算符，如果任何操作数不是数字（Double.NaN 或 Single.NaN），则运算的结果为 `false`。 这意味着 `NaN` 值不大于、小于或等于任何其他 `double`（或 `float`）值，包括 `NaN`。 有关更多信息和示例，请参阅 Double.NaN 或 Single.NaN 参考文章。

char 类型也支持比较运算符。 在使用 `char` 操作数时，将比较对应的字符代码。

枚举类型也支持比较运算符。 对于相同枚举类型的操作数，基础整数类型的相应值会进行比较。

== 和 != 运算符检查其操作数是否相等。

小于运算符 <

如果左侧操作数小于右侧操作数，`<` 运算符返回 `true`，否则返回 `false`：

C#

```
Console.WriteLine(7.0 < 5.1);   // output: False
Console.WriteLine(5.1 < 5.1);   // output: False
Console.WriteLine(0.0 < 5.1);   // output: True

Console.WriteLine(double.NaN < 5.1);   // output: False
Console.WriteLine(double.NaN >= 5.1);  // output: False
```

大于运算符 >

如果左侧操作数大于右侧操作数，`>` 运算符返回 `true`，否则返回 `false`：

C#

```
Console.WriteLine(7.0 > 5.1);   // output: True
Console.WriteLine(5.1 > 5.1);   // output: False
```

```
Console.WriteLine(0.0 > 5.1);   // output: False

Console.WriteLine(double.NaN > 5.1);   // output: False
Console.WriteLine(double.NaN <= 5.1);  // output: False
```

小于或等于运算符 <=

如果左侧操作数小于或等于右侧操作数，`<=` 运算符返回 `true`，否则返回 `false`：

C#

```
Console.WriteLine(7.0 <= 5.1);   // output: False
Console.WriteLine(5.1 <= 5.1);   // output: True
Console.WriteLine(0.0 <= 5.1);   // output: True

Console.WriteLine(double.NaN > 5.1);   // output: False
Console.WriteLine(double.NaN <= 5.1);  // output: False
```

大于或等于运算符 >=

如果左侧操作数大于或等于右侧操作数，`>=` 运算符返回 `true`，否则返回 `false`：

C#

```
Console.WriteLine(7.0 >= 5.1);   // output: True
Console.WriteLine(5.1 >= 5.1);   // output: True
Console.WriteLine(0.0 >= 5.1);   // output: False

Console.WriteLine(double.NaN < 5.1);   // output: False
Console.WriteLine(double.NaN >= 5.1);  // output: False
```

运算符可重载性

用户定义类型可以重载 `<`、`>`、`<=` 和 `>=` 运算符。

如果某类型重载 `<` 或 `>` 运算符之一，它必须同时重载 `<` 和 `>`。 如果某类型重载 `<=` 或 `>=` 运算符之一，它必须同时重载 `<=` 和 `>=`。

C# 语言规范

有关详细信息，请参阅 C# 语言规范中的关系和类型测试运算符部分。

另请参阅

- C# 运算符和表达式
- System.IComparable<T>
- 相等运算符

在 GitHub 上与我们协作

可以在 GitHub 上找到此内容的源，还可以在其中创建和查看问题和拉取请求。 有关详细信息，请参阅参与者指南。

.NET 反馈

.NET 是一个开放源代码项目。 选择一个链接以提供反馈：

提出文档问题

提供产品反馈

成员访问运算符和表达式 - 点、索引器和调用运算符。

项目 • 2024/07/31

使用多个运算符和表达式来访问类型成员。 这些运算符包括成员访问 (`.`)、数组元素或索引器访问 (`[]`)、从末尾开始索引 (`^`)、范围 (`..`)、null 条件运算符 （`?.` 和 `?[]`），以及方法调用 (`()`)。 其中包括 Null 条件成员访问 (`?.`) 和索引器访问 (`?[]`) 运算符。

- . （成员访问）：用于访问命名空间或类型的成员
- [] （数组元素或索引器访问）：用于访问数组元素或类型索引器
- ?. 和 ?[] （null 条件运算符）：仅当操作数为非 null 时才用于执行成员或元素访问运算
- () （调用）：用于调用被访问的方法或调用委托
- ^ （从末尾开始索引）：指示元素位置来自序列的末尾
- .. （范围）：指定可用于获取一系列序列元素的索引范围

成员访问表达式 `.`

可以使用 `.` 标记来访问命名空间或类型的成员，如以下示例所示：

- 使用 `.` 访问命名空间内的嵌套命名空间，如以下 `.` 的示例所示：

```
C#
```

```
using System.Collections.Generic;
```

- 使用 `.` 构成限定名称以访问命名空间中的类型，如下面的代码所示：

```
C#
```

```
System.Collections.Generic.IEnumerable<int> numbers = [1, 2, 3];
```

使用 using 指令来使用可选的限定名称。

- 使用 `.` 访问类型成员（静态和非静态），如下面的代码所示：

```
C#
```

```
List<double> constants =
[
    Math.PI,
```

```
    Math.E
];
Console.WriteLine($"{constants.Count} values to show:");
Console.WriteLine(string.Join(", ", constants));
// Output:
// 2 values to show:
// 3.14159265358979, 2.71828182845905
```

还可以使用 `.` 访问 `.`。

索引器运算符 []

方括号 `[]` 通常用于数组、索引器或指针元素访问。 从 C# 12 开始，`[]` 会将集合表达式括起来。

数组访问

下面的示例演示如何访问数组元素：

C#

```
int[] fib = new int[10];
fib[0] = fib[1] = 1;
for (int i = 2; i < fib.Length; i++)
{
    fib[i] = fib[i - 1] + fib[i - 2];
}
Console.WriteLine(fib[fib.Length - 1]);  // output: 55

double[,] matrix = new double[2,2];
matrix[0,0] = 1.0;
matrix[0,1] = 2.0;
matrix[1,0] = matrix[1,1] = 3.0;
var determinant = matrix[0,0] * matrix[1,1] - matrix[1,0] * matrix[0,1];
Console.WriteLine(determinant);  // output: -3
```

如果数组索引超出数组相应维度的边界，将引发 IndexOutOfRangeException。

如上述示例所示，在声明数组类型或实例化数组实例时，还会使用方括号。

有关数组的详细信息，请参阅数组。

索引器访问

下面的示例使用 .NET Dictionary<TKey,TValue> 类型来演示索引器访问：

C#

```
var dict = new Dictionary<string, double>();
dict["one"] = 1;
dict["pi"] = Math.PI;
Console.WriteLine(dict["one"] + dict["pi"]);  // output: 4.14159265358979
```

使用索引器，可通过类似于编制数组索引的方式对用户定义类型的实例编制索引。 与必须是整数的数组索引不同，可以将索引器参数声明为任何类型。

有关索引器的详细信息，请参阅索引器。

[] 的其他用法

要了解指针元素访问，请参阅与指针相关的运算符一文的指针元素访问运算符 [] 部分。 有关集合表达式的信息，请参阅集合表达式一文。

方括号还用于指定属性：

C#

```
[System.Diagnostics.Conditional("DEBUG")]
void TraceMethod() {}
```

Null 条件运算符 `?.` 和 `?[]`

仅当操作数的计算结果为非 NULL 时，NULL 条件运算符才对其操作数应用成员访问 (`?.`) 或元素访问 (`?[]`) 操作；否则，它会返回 `null`。 换句话说：

- 如果 `a` 的计算结果为 `null`，则 `a?.x` 或 `a?[x]` 的结果为 `null`。
- 如果 `a` 的计算结果为非 null，则 `a?.x` 或 `a?[x]` 的结果将分别与 `a.x` 或 `a[x]` 的结果相同。

> **备注**
>
> 如果 `a.x` 或 `a[x]` 引发异常，则 `a?.x` 或 `a?[x]` 将对非 null `a` 引发相同的异常。 例如，如果 `a` 为非 null 数组实例且 `x` 在 `a` 的边界之外，则 `a?[x]` 将引发 IndexOutOfRangeException。

NULL 条件运算符采用最小化求值策略。 也就是说，如果条件成员或元素访问运算链中的一个运算返回 `null`，则链的其余部分不会执行。 在以下示例中，如果 `A` 的计算结果为

`null`，则不会计算 `B`；如果 `A` 或 `B` 的计算结果为 `null`，则不会计算 `C`：

C#

```
A?.B?.Do(C);
A?.B?[C];
```

如果 `A` 可以为 null，但如果 A 不为 null，`B` 和 `C` 将不为 null，你只需要对 `A` 应用 null 条件运算符：

C#

```
A?.B.C();
```

在上述示例中，如果 `A` 为 null，则不会计算 `B`，也不会调用 `C()`。 但是，如果链接的成员访问被中断，例如被 `(A?.B).C()` 中的括号中断，则不会发生短路。

以下示例演示了 `?.` 和 `?[]` 运算符的用法：

C#

```
double SumNumbers(List<double[]> setsOfNumbers, int indexOfSetToSum)
{
    return setsOfNumbers?[indexOfSetToSum]?.Sum() ?? double.NaN;
}

var sum1 = SumNumbers(null, 0);
Console.WriteLine(sum1);  // output: NaN

List<double[]?> numberSets =
[
    [1.0, 2.0, 3.0],
    null
];

var sum2 = SumNumbers(numberSets, 0);
Console.WriteLine(sum2);  // output: 6

var sum3 = SumNumbers(numberSets, 1);
Console.WriteLine(sum3);  // output: NaN
```

C#

```
namespace MemberAccessOperators2;

public static class NullConditionalShortCircuiting
{
    public static void Main()
```

```
    {
        Person? person = null;
        person?.Name.Write(); // no output: Write() is not called due to
short-circuit.
        try
        {
            (person?.Name).Write();
        }
        catch (NullReferenceException)
        {
            Console.WriteLine("NullReferenceException");
        }; // output: NullReferenceException
    }
}

public class Person
{
    public required FullName Name { get; set; }
}

public class FullName
{
    public required string FirstName { get; set; }
    public required string LastName { get; set; }
    public void Write() => Console.WriteLine($"{FirstName} {LastName}");
}
```

前面两个示例中的第一个也使用 Null 合并运算符 ?? 来指定替代表达式，以便在 null 条件运算的结果为 `null` 时用于计算。

如果 `a.x` 或 `a[x]` 是不可为 null 的值类型 `T`，则 `a?.x` 或 `a?[x]` 属于对应的可为 null 的值类型 `T?`。 如果需要 `T` 类型的表达式，请将 Null 合并操作符 `??` 应用于 null 条件表达式，如下面的示例所示：

C#

```
int GetSumOfFirstTwoOrDefault(int[]? numbers)
{
    if ((numbers?.Length ?? 0) < 2)
    {
        return 0;
    }
    return numbers[0] + numbers[1];
}

Console.WriteLine(GetSumOfFirstTwoOrDefault(null));  // output: 0
Console.WriteLine(GetSumOfFirstTwoOrDefault([]));  // output: 0
Console.WriteLine(GetSumOfFirstTwoOrDefault([3, 4, 5]));  // output: 7
```

在前面的示例中，如果不使用 `??` 运算符，则在 `numbers` 为 `null` 时，`numbers?.Length < 2` 的计算结果为 `false`。

> ⓘ **备注**
>
> `?.` 运算符对其左操作数的计算不超过一次，从而确保在验证为非 null 后，不能将其更改为 `null`。

Null 条件成员访问运算符 `?.` 也称为 Elvis 运算符。

线程安全的委托调用

使用 `?.` 运算符来检查委托是否非 null 并以线程安全的方式调用它（例如，`?.`时），如下面的代码所示：

C#

```
PropertyChanged?.Invoke(…)
```

该代码等同于以下代码：

C#

```
var handler = this.PropertyChanged;
if (handler != null)
{
    handler(…);
}
```

前面的示例是一种线程安全方法，可确保只调用非 null `handler`。 由于委托实例是不可变的，因此，任何线程都不能更改 `handler` 本地变量所引用的对象。 具体而言，如果另一个线程执行的代码从 `PropertyChanged` 事件中取消订阅，并且 `PropertyChanged` 在调用 `handler` 之前变为 `null`，则 `handler` 引用的对象不受影响。

调用表达式 ()

使用括号 `()` 调用`()`或调用委托。

以下示例演示如何在使用或不使用参数的情况下调用方法，以及调用委托：

C#

```
Action<int> display = s => Console.WriteLine(s);

List<int> numbers =
[
    10,
    17
];
display(numbers.Count);   // output: 2

numbers.Clear();
display(numbers.Count);   // output: 0
```

在调用带 new 运算符的构造函数时，还可以使用括号。

() 的其他用法

此外可以使用括号来调整表达式中计算操作的顺序。 有关详细信息，请参阅 C# 运算符。

强制转换表达式，其执行显式类型转换，也可以使用括号。

从末尾运算符 ^ 开始索引

索引和范围运算符可以与可数类型一起使用。 可数类型具有名为 `Count` 或 `Length` 的 `int` 属性以及可访问的 `get` 访问器。 集合表达式还依赖于可数类型。

`^` 运算符指示序列末尾的元素位置。 对于长度为 `length` 的序列，`^n` 指向与序列开头偏移 `length - n` 的元素。 例如，`^1` 指向序列的最后一个元素，`^length` 指向序列的第一个元素。

C#

```
int[] xs = [0, 10, 20, 30, 40];
int last = xs[^1];
Console.WriteLine(last);  // output: 40

List<string> lines = ["one", "two", "three", "four"];
string prelast = lines[^2];
Console.WriteLine(prelast);  // output: three

string word = "Twenty";
Index toFirst = ^word.Length;
char first = word[toFirst];
Console.WriteLine(first);  // output: T
```

如前面的示例所示，表达式 `^e` 属于 System.Index 类型。 在表达式 `^e` 中，`e` 的结果必须隐式转换为 `int`。

还可以将 `^` 运算符与`^`一起使用以创建一个索引范围。 有关详细信息，请参阅索引和范围。

从 C# 13 开始，可在对象初始值设定项中使用结束运算符中的索引。

范围运算符 `..`

`..` 运算符指定某一索引范围的开头和末尾作为其操作数。 左侧操作数是范围的包含性开头。 右侧操作数是范围的不包含性末尾。 任一操作数都可以是序列开头或末尾的索引，如以下示例所示：

C#

```
int[] numbers = [0, 10, 20, 30, 40, 50];
int start = 1;
int amountToTake = 3;
int[] subset = numbers[start..(start + amountToTake)];
Display(subset);  // output: 10 20 30

int margin = 1;
int[] inner = numbers[margin..^margin];
Display(inner);  // output: 10 20 30 40

string line = "one two three";
int amountToTakeFromEnd = 5;
Range endIndices = ^amountToTakeFromEnd..^0;
string end = line[endIndices];
Console.WriteLine(end);  // output: three

void Display<T>(IEnumerable<T> xs) => Console.WriteLine(string.Join(" ",
xs));
```

如前面的示例所示，表达式 `a..b` 属于 System.Range 类型。 在表达式 `a..b` 中，`a` 和 `b` 的结果必须隐式转换为 Int32 或 Index。

> **重要**
>
> 当值为负数时，从 `int` 隐式转换为 `Index` 会引发 ArgumentOutOfRangeException。

可以省略 `..` 运算符的任何操作数来获取无限制范围：

- `a..` 等效于 `a..^0`
- `..b` 等效于 `0..b`
- `..` 等效于 `0..^0`

C#

```
int[] numbers = [0, 10, 20, 30, 40, 50];
int amountToDrop = numbers.Length / 2;

int[] rightHalf = numbers[amountToDrop..];
Display(rightHalf);  // output: 30 40 50

int[] leftHalf = numbers[..^amountToDrop];
Display(leftHalf);  // output: 0 10 20

int[] all = numbers[..];
Display(all);  // output: 0 10 20 30 40 50

void Display<T>(IEnumerable<T> xs) => Console.WriteLine(string.Join(" ",
xs));
```

下表显示了表达集合范围的各种方法：

展开表

范围运算符表达式	说明
`..`	集合中的所有值。
`..end`	从开头到 `end`（不含）的值。
`start..`	从 `start`（含）到结尾的值。
`start..end`	从 `start`（含）到 `end`（不含）的值。
`^start..`	从 `start`（含）到倒计数结尾的值。
`..^end`	从开头到 `end`（不含，倒计数）的值。
`start..^end`	从 `start`（含）到 `end`（不含，倒计数）的值。
`^start..^end`	从 `start`（含）到 `end`（不含），均为倒计数。

以下示例演示了使用上表中所有范围的效果：

C#

```
int[] oneThroughTen =
[
```

```
    1, 2, 3, 4, 5, 6, 7, 8, 9, 10
];

Write(oneThroughTen, ..);
Write(oneThroughTen, ..3);
Write(oneThroughTen, 2..);
Write(oneThroughTen, 3..5);
Write(oneThroughTen, ^2..);
Write(oneThroughTen, ..^3);
Write(oneThroughTen, 3..^4);
Write(oneThroughTen, ^4..^2);

static void Write(int[] values, Range range) =>
    Console.WriteLine($"{range}:\t{string.Join(", ", values[range])}");
// Sample output:
//      0..^0:      1, 2, 3, 4, 5, 6, 7, 8, 9, 10
//      0..3:       1, 2, 3
//      2..^0:      3, 4, 5, 6, 7, 8, 9, 10
//      3..5:       4, 5
//      ^2..^0:     9, 10
//      0..^3:      1, 2, 3, 4, 5, 6, 7
//      3..^4:      4, 5, 6
//      ^4..^2:     7, 8
```

有关详细信息，请参阅索引和范围。

`..` 词元还用作集合表达式中的 spread 元素。

运算符可重载性

`.`、`()`、`^` 和 `..` 运算符无法进行重载。 `[]` 运算符也被视为非可重载运算符。 使用索引器以支持对用户定义的类型编制索引。

C# 语言规范

有关更多信息，请参阅 C# 语言规范的以下部分：

- 成员访问
- 元素访问
- Null 条件成员访问
- 调用表达式

有关索引和范围的详细信息，请参阅功能建议说明。

请参阅

- 使用索引运算符（样式规则 IDE0056）
- 使用范围运算符（样式规则 IDE0057）
- 使用条件委托调用（样式规则 IDE1005）
- C# 运算符和表达式
- ?? （Null 合并操作符）
- :: 运算符

在 GitHub 上与我们协作

可以在 GitHub 上找到此内容的源，还可以在其中创建和查看问题和拉取请求。 有关详细信息，请参阅参与者指南。

.NET 反馈

.NET 是一个开放源代码项目。 选择一个链接以提供反馈：

提出文档问题

提供产品反馈

类型测试运算符和强制转换表达式 - `is`、`as`、`typeof` 和强制转换

项目 • 2023/05/10

这些运算符和表达式执行类型检查或类型转换。 `is`运算符检查表达式结果的运行时类型是否与给定类型兼容。 `as`运算符将表达式显式转换为给定类型（如果其运行时类型与该类型兼容）。 强制转换表达式执行到目标类型的显式转换。 `typeof`运算符获取某个类型的 System.Type 实例。

is 运算符

`is` 运算符检查表达式结果的运行时类型是否与给定类型兼容。 `is` 运算符还会对照某个模式测试表达式结果。

具有类型测试 `is` 运算符的表达式具有以下形式

```
C#
```

```
E is T
```

其中 `E` 是返回一个值的表达式，`T` 是类型或类型参数的名称。 `E` 不得为匿名方法或 Lambda 表达式。

如果表达式结果为非 null 并且满足以下任一条件，则 `is` 运算符将返回 `true`：

- 表达式结果的运行时类型为 `T`。
- 表达式结果的运行时类型派生自类型 `T`、实现接口 `T`，或者存在从其到 `T` 的另一种隐式引用转换。
- 表达式结果的运行时类型是基础类型为 `T` 且 Nullable<T>.HasValue 为 `true` 的可为空值类型。
- 存在从表达式结果的运行时类型到类型 `T` 的装箱或取消装箱转换。

`is` 运算符不会考虑用户定义的转换。

以下示例演示，如果表达式结果的运行时类型派生自给定类型，即类型之间存在引用转换，`is` 运算符将返回 `true`：

```
C#
```

```
public class Base { }

public class Derived : Base { }

public static class IsOperatorExample
{
    public static void Main()
    {
        object b = new Base();
        Console.WriteLine(b is Base);  // output: True
        Console.WriteLine(b is Derived);  // output: False

        object d = new Derived();
        Console.WriteLine(d is Base);  // output: True
        Console.WriteLine(d is Derived); // output: True
    }
}
```

以下示例演示，`is` 运算符将考虑装箱和取消装箱转换，但不会考虑数值转换：

C#

```
int i = 27;
Console.WriteLine(i is System.IFormattable);  // output: True

object iBoxed = i;
Console.WriteLine(iBoxed is int);  // output: True
Console.WriteLine(iBoxed is long);  // output: False
```

有关 C# 转换的信息，请参阅 C# 语言规范的转换一章。

有模式匹配的类型测试

`is` 运算符还会对照某个模式测试表达式结果。 下面的示例演示如何使用声明模式来检查表达式的运行时类型：

C#

```
int i = 23;
object iBoxed = i;
int? jNullable = 7;
if (iBoxed is int a && jNullable is int b)
{
    Console.WriteLine(a + b);  // output 30
}
```

若要了解受支持的模式，请参阅模式。

as 运算符

`as` 运算符将表达式结果显式转换为给定的引用或可以为 null 值的类型。 如果无法进行转换，则 `as` 运算符返回 `null`。 与强制转换表达式 不同，`as` 运算符永远不会引发异常。

形式如下的表达式

```
C#

E as T
```

其中，`E` 为返回值的表达式，`T` 为类型或类型参数的名称，生成相同的结果，

```
C#

E is T ? (T)(E) : (T)null
```

不同的是 `E` 只计算一次。

`as` 运算符仅考虑引用、可以为 null、装箱和取消装箱转换。 不能使用 `as` 运算符执行用户定义的转换。 为此，请使用强制转换表达式。

下面的示例演示 `as` 运算符的用法：

```
C#

IEnumerable<int> numbers = new[] { 10, 20, 30 };
IList<int> indexable = numbers as IList<int>;
if (indexable != null)
{
    Console.WriteLine(indexable[0] + indexable[indexable.Count - 1]);  //
output: 40
}
```

> ⓘ **备注**
>
> 如之前的示例所示，你需要将 `as` 表达式的结果与 `null` 进行比较，以检查转换是否成功。 可以使用 is **运算符**测试转换是否成功，如果成功，则将其结果分配给新变量。

强制转换表达式

形式为 `(T)E` 的强制转换表达式将表达式 `E` 的结果显式转换为类型 `T`。 如果不存在从类型 `E` 到类型 `T` 的显式转换，则发生编译时错误。 在运行时，显式转换可能不会成功，强制转换表达式可能会引发异常。

下面的示例演示显式数值和引用转换：

C#

```
double x = 1234.7;
int a = (int)x;
Console.WriteLine(a);   // output: 1234

IEnumerable<int> numbers = new int[] { 10, 20, 30 };
IList<int> list = (IList<int>)numbers;
Console.WriteLine(list.Count);  // output: 3
Console.WriteLine(list[1]);  // output: 20
```

有关支持的显式转换的信息，请参阅 C# 语言规范的显式转换部分。 有关如何定义自定义显式或隐式类型转换的信息，请参阅用户定义转换运算符。

() 的其他用法

你还可以使用括号调用方法或调用委托。

括号的其他用法是调整表达式中计算操作的顺序。 有关详细信息，请参阅 C# 运算符。

typeof 运算符

`typeof` 运算符用于获取某个类型的 System.Type 实例。 `typeof` 运算符的实参必须是类型或类型形参的名称，如以下示例所示：

C#

```
void PrintType<T>() => Console.WriteLine(typeof(T));

Console.WriteLine(typeof(List<string>));
PrintType<int>();
PrintType<System.Int32>();
PrintType<Dictionary<int, char>>();
// Output:
// System.Collections.Generic.List`1[System.String]
// System.Int32
// System.Int32
// System.Collections.Generic.Dictionary`2[System.Int32,System.Char]
```

参数不能是需要元数据注释的类型。 示例包括以下类型：

- `dynamic`
- `string?`（或任何可为 null 的引用类型）

这些类型不会直接在元数据中表示出来。 这些类型包括描述基础类型的属性。 在这两种情况下，都可以使用基础类型。 可以使用 `object` 来代替 `dynamic`。 可以使用 `string` 来代替 `string?`。

你还可以使用具有未绑定泛型类型的 `typeof` 运算符。 未绑定泛型类型的名称必须包含适当数量的逗号，且此数量小于类型参数的数量。 以下示例演示了具有未绑定泛型类型的 `typeof` 运算符的用法：

C#

```
Console.WriteLine(typeof(Dictionary<,>));
// Output:
// System.Collections.Generic.Dictionary`2[TKey,TValue]
```

表达式不能为 `typeof` 运算符的参数。 若要获取表达式结果的运行时类型的 System.Type 实例，请使用 Object.GetType 方法。

使用 `typeof` 运算符进行类型测试

使用 `typeof` 运算符来检查表达式结果的运行时类型是否与给定的类型完全匹配。 以下示例演示了使用 `typeof` 运算符和 is 运算符执行的类型检查之间的差异：

C#

```
public class Animal { }

public class Giraffe : Animal { }

public static class TypeOfExample
{
    public static void Main()
    {
        object b = new Giraffe();
        Console.WriteLine(b is Animal);  // output: True
        Console.WriteLine(b.GetType() == typeof(Animal));  // output: False

        Console.WriteLine(b is Giraffe);  // output: True
        Console.WriteLine(b.GetType() == typeof(Giraffe));  // output: True
    }
}
```

运算符可重载性

`is`、`as` 和 `typeof` 运算符无法进行重载。

用户定义的类型不能重载 `()` 运算符，但可以定义可由强制转换表达式执行的自定义类型转换。 有关详细信息，请参阅用户定义转换运算符。

C# 语言规范

有关更多信息，请参阅 C# 语言规范的以下部分：

- is 运算符
- as 运算符
- 强制转换表达式
- typeof 运算符

请参阅

- C# 参考
- C# 运算符和表达式
- 如何使用模式匹配以及 is 和 as 运算符安全地进行强制转换
- .NET 中的泛型

用户定义的显式和隐式转换运算符

项目 • 2023/04/07

用户定义类型可以定义从或到另一个类型的自定义隐式或显式转换。 隐式转换无需调用特殊语法，并且可以在各种情况（例如，在赋值和方法调用中）下发生。 预定义的 C# 隐式转换始终成功，且永远不会引发异常。 用户定义隐式转换也应如此。 如果自定义转换可能会引发异常或丢失信息，请将其定义为显式转换。

is 和 as 运算符不考虑使用用户定义转换。 强制转换表达式用于调用用户定义显式转换。

`operator` 和 `implicit` 或 `explicit` 关键字分别用于定义隐式转换或显式转换。 定义转换的类型必须是该转换的源类型或目标类型。 可用两种类型中的任何一种类型来定义两种用户定义类型之间的转换。

下面的示例展示如何定义隐式转换和显式转换：

C#

```
using System;

public readonly struct Digit
{
    private readonly byte digit;

    public Digit(byte digit)
    {
        if (digit > 9)
        {
            throw new ArgumentOutOfRangeException(nameof(digit), "Digit
cannot be greater than nine.");
        }
        this.digit = digit;
    }

    public static implicit operator byte(Digit d) => d.digit;
    public static explicit operator Digit(byte b) => new Digit(b);

    public override string ToString() => $"{digit}";
}

public static class UserDefinedConversions
{
    public static void Main()
    {
        var d = new Digit(7);

        byte number = d;
        Console.WriteLine(number);  // output: 7
```

```
        Digit digit = (Digit)number;
        Console.WriteLine(digit);  // output: 7
    }
}
```

从 C# 11 开始，可以定义 checked 显式转换运算符。 有关详细信息，请参阅算术运算符一文的用户定义的 checked 运算符部分。

`operator` 关键字也可用于重载预定义的 C# 运算符。 有关详细信息，请参阅运算符重载。

C# 语言规范

有关更多信息，请参阅 C# 语言规范的以下部分：

- 转换运算符
- 用户定义的转换
- 隐式转换
- 显式转换

请参阅

- C# 参考
- C# 运算符和表达式
- 运算符重载
- 类型测试和强制转换运算符
- 强制转换和类型转换
- 设计准则 - 转换运算符
- Chained user-defined explicit conversions in C#（C# 中链接在一起的用户定义的显式转换）

指针相关运算符 - 获取变量的地址、取消引用存储位置和访问内存位置

项目 • 2023/04/07

使用指针运算符可以获取变量的地址 (`&`)、取消引用指针 (`*`)、比较指针值以及添加或减去指针和整数。

使用以下运算符来使用指针：

- 一元 & (address-of) 运算符：用于获取变量的地址
- 一元 * （指针间接）运算符：用于获取指针指向的变量
- -> （成员访问）和 [] （元素访问）运算符
- 算术运算符 +、-、++ 和 --
- 比较运算符 ==、!=、<、>、<= 和 >=

有关指针类型的信息，请参阅指针类型。

> **备注**
>
> 任何带指针的运算都需要使用 unsafe 上下文。 必须使用 AllowUnsafeBlocks 编译器选项来编译包含不安全块的代码。

Address-of 运算符 &

一元 `&` 运算符返回其操作数的地址：

C#

```
unsafe
{
    int number = 27;
    int* pointerToNumber = &number;

    Console.WriteLine($"Value of the variable: {number}");
    Console.WriteLine($"Address of the variable:
{(long)pointerToNumber:X}");
}
// Output is similar to:
// Value of the variable: 27
// Address of the variable: 6C1457DBD4
```

`&` 运算符的操作数必须是固定变量。 固定变量是驻留在不受垃圾回收器操作影响的存储位置的变量 。 在上述示例中，局部变量 `number` 是固定变量，因为它驻留在堆栈上。 驻留在可能受垃圾回收器影响的存储位置的变量（如重定位）称为可移动变量 。 对象字段和数组元素是可移动变量的示例。 如果使用 fixed 语句“固定”，则可以获取可移动变量的地址。 获取的地址仅在 `fixed` 语句块中有效。 以下示例显示如何使用 `fixed` 语句和 `&` 运算符：

C#

```
unsafe
{
    byte[] bytes = { 1, 2, 3 };
    fixed (byte* pointerToFirst = &bytes[0])
    {
        // The address stored in pointerToFirst
        // is valid only inside this fixed statement block.
    }
}
```

你也无法获取常量或值的地址。

有关固定变量和可移动变量的详细信息，请参阅 C# 语言规范的固定变量和可移动变量部分。

二进制 `&` 运算符计算其布尔操作数的逻辑 AND 或其整型操作数的位逻辑 AND。

指针间接运算符 *

一元指针间接运算符 `*` 获取其操作数指向的变量。 它也称为取消引用运算符。 `*` 运算符的操作数必须是指针类型。

C#

```
unsafe
{
    char letter = 'A';
    char* pointerToLetter = &letter;
    Console.WriteLine($"Value of the `letter` variable: {letter}");
    Console.WriteLine($"Address of the `letter` variable:
{(long)pointerToLetter:X}");

    *pointerToLetter = 'Z';
    Console.WriteLine($"Value of the `letter` variable after update:
{letter}");
}
// Output is similar to:
// Value of the `letter` variable: A
```

```
// Address of the `letter` variable: DCB977DDF4
// Value of the `letter` variable after update: Z
```

不能将 `*` 运算符应用于类型 `void*` 的表达式。

二进制 `*` 运算符计算其数值操作数的乘积。

指针成员访问运算符 ->

`->` 运算符将指针间接和成员访问合并。 也就是说，如果 `x` 是类型为 `T*` 的指针且 `y` 是类型 `T` 的可访问成员，则形式的表达式

C#

```
x->y
```

等效于

C#

```
(*x).y
```

下面的示例演示 `->` 运算符的用法：

C#

```
public struct Coords
{
    public int X;
    public int Y;
    public override string ToString() => $"({X}, {Y})";
}

public class PointerMemberAccessExample
{
    public static unsafe void Main()
    {
        Coords coords;
        Coords* p = &coords;
        p->X = 3;
        p->Y = 4;
        Console.WriteLine(p->ToString());  // output: (3, 4)
    }
}
```

不能将 `->` 运算符应用于类型 `void*` 的表达式。

指针元素访问运算符 []

对于指针类型的表达式 `p`，`p[n]` 形式的指针元素访问计算方式为 `*(p + n)`，其中 `n` 必须是可隐式转换为 `int`、`uint`、`long` 或 `ulong` 的类型。 有关带有指针的 `+` 运算符的行为的信息，请参阅向指针中增加或从指针中减少整数值部分。

以下示例演示如何使用指针和 `[]` 运算符访问数组元素：

C#

```
unsafe
{
    char* pointerToChars = stackalloc char[123];

    for (int i = 65; i < 123; i++)
    {
        pointerToChars[i] = (char)i;
    }

    Console.Write("Uppercase letters: ");
    for (int i = 65; i < 91; i++)
    {
        Console.Write(pointerToChars[i]);
    }
}
// Output:
// Uppercase letters: ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

在前面的示例中，stackalloc 表达式 在堆栈上分配内存块。

> ⓘ **备注**
>
> 指针元素访问运算符不检查越界错误。

不能将 `[]` 用于带有类型为 `void*` 的表达式的指针元素访问。

还可以将 `[]` 运算符用于数组元素或索引器访问。

指针算术运算符

可以使用指针执行以下算术运算：

- 向指针增加或从指针中减少一个整数值
- 减少两个指针
- 增量或减量指针

不能使用类型为 `void*` 的指针执行这些操作。

有关带数值类型的受支持的算术运算的信息，请参阅算术运算符。

向指针增加或从指针中减少整数值

对于类型为 `T*` 的指针 `p` 和可隐式转换为 `int`、`uint`、`long` 或 `ulong` 的类型的表达式 `n`，加法和减法定义如下：

- `p + n` 和 `n + p` 表达式都生成 `T*` 类型的指针，该指针是将 `n * sizeof(T)` 添加到 `p` 给出的地址得到的。
- `p - n` 表达式生成类型为 `T*` 的指针，该指针是从 `p` 给出的地址中减去 `n * sizeof(T)` 得到的。

sizeof 运算符获取类型的大小（以字节为单位）。

以下示例演示了 `+` 运算符与指针的用法：

C#

```
unsafe
{
    const int Count = 3;
    int[] numbers = new int[Count] { 10, 20, 30 };
    fixed (int* pointerToFirst = &numbers[0])
    {
        int* pointerToLast = pointerToFirst + (Count - 1);

        Console.WriteLine($"Value {*pointerToFirst} at address
{(long)pointerToFirst}");
        Console.WriteLine($"Value {*pointerToLast} at address
{(long)pointerToLast}");
    }
}
// Output is similar to:
// Value 10 at address 1818345918136
// Value 30 at address 1818345918144
```

指针减法

对于类型为 `T*` 的两个指针 `p1` 和 `p2`，表达式 `p1 - p2` 生成 `p1` 和 `p2` 给出的地址之间的差除以 `sizeof(T)` 的值。 结果的类型为 `long`。 即 `p1 - p2` 计算公式为 `((long)(p1) - (long)(p2)) / sizeof(T)`。

以下示例演示了指针减法：

C#

```
unsafe
{
    int* numbers = stackalloc int[] { 0, 1, 2, 3, 4, 5 };
    int* p1 = &numbers[1];
    int* p2 = &numbers[5];
    Console.WriteLine(p2 - p1);  // output: 4
}
```

指针增量和减量

`++` 增量运算符将 1 添加到其指针操作数。 `--` 减量运算符从其指针操作数中减去 1。

两种运算符都支持两种形式：后缀（`p++` 和 `p--`）和前缀（`++p` 和 `--p`）。 `p++` 和 `p--` 的结果是该运算之前 `p` 的值。 `++p` 和 `--p` 的结果是该运算之后 `p` 的值。

以下示例演示了后缀和前缀增量运算符的行为：

C#

```
unsafe
{
    int* numbers = stackalloc int[] { 0, 1, 2 };
    int* p1 = &numbers[0];
    int* p2 = p1;
    Console.WriteLine($"Before operation: p1 - {(long)p1}, p2 -
{(long)p2}");
    Console.WriteLine($"Postfix increment of p1: {(long)(p1++)}");
    Console.WriteLine($"Prefix increment of p2: {(long)(++p2)}");
    Console.WriteLine($"After operation: p1 - {(long)p1}, p2 - {(long)p2}");
}
// Output is similar to
// Before operation: p1 - 816489946512, p2 - 816489946512
// Postfix increment of p1: 816489946512
// Prefix increment of p2: 816489946516
// After operation: p1 - 816489946516, p2 - 816489946516
```

指针比较运算符

可以使用 `==`、`!=`、`<`、`>`、`<=` 和 `>=` 运算符来比较任何指针类型（包括 `void*`）的操作数。 这些运算符将两个操作数给出的地址进行比较，就好像它们是无符号整数一样。

有关这些运算符对其他类型操作数的行为的信息，请参阅等式运算符和比较运算符一文。

运算符优先级

以下列表按优先级从高到低的顺序对指针相关运算符进行排序：

- 后缀增量 `x++` 和减量 `x--` 运算符以及 `->` 和 `[]` 运算符
- 前缀增量 `++x` 和减量 `--x` 运算符以及 `&` 和 `*` 运算符
- 加法 `+` 和 `-` 运算符
- 比较 `<`、`>`、`<=` 和 `>=` 运算符
- 等式 `==` 和 `!=` 运算符

使用括号 `()` 更改运算符优先级所施加的计算顺序。

如需了解按优先级排序的完整 C# 运算符列表，请参阅 C# 运算符一文中的运算符优先级部分。

运算符可重载性

用户定义的类型不能重载与指针相关的运算符 `&`、`*`、`->` 和 `[]`。

C# 语言规范

有关更多信息，请参阅 C# 语言规范的以下部分：

- 固定和可移动变量
- address-of 运算符
- 指针间接
- 指针成员访问
- 指针元素访问
- 指针算术
- 指针增量和减量
- 指针比较

请参阅

- C# 参考
- C# 运算符和表达式
- 不安全代码、指针类型和函数指针
- unsafe 关键字
- fixed 语句
- stackalloc 表达式

- sizeof 运算符

赋值运算符（C# 参考）

项目 • 2024/07/04

赋值运算符 `=` 将其右操作数的值赋给变量、属性或由其左操作数给出的索引器元素。 赋值表达式的结果是分配给左操作数的值。 右操作数类型必须与左操作数类型相同，或可隐式转换为左操作数的类型。

赋值运算符 `=` 为右联运算符，即形式的表达式

```
C#

a = b = c
```

计算结果为

```
C#

a = (b = c)
```

以下示例演示使用局部变量、属性和索引器元素作为其左操作数的赋值运算符的用法：

```
C#

List<double> numbers = [1.0, 2.0, 3.0];

Console.WriteLine(numbers.Capacity);
numbers.Capacity = 100;
Console.WriteLine(numbers.Capacity);
// Output:
// 4
// 100

int newFirstElement;
double originalFirstElement = numbers[0];
newFirstElement = 5;
numbers[0] = newFirstElement;
Console.WriteLine(originalFirstElement);
Console.WriteLine(numbers[0]);
// Output:
// 1
// 5
```

赋值操作的左操作数接收右操作数的值。 当操作数是值类型时，赋值操作将复制右侧操作数的内容。 当操作数为引用类型时，赋值操作会复制对对象的引用。

这称为值分配：分配值。

ref 赋值

Ref 赋值 `= ref` 使其左侧操作数成为右侧操作数的别名，如以下示例所示：

C#

```
void Display(double[] s) => Console.WriteLine(string.Join(" ", s));

double[] arr = { 0.0, 0.0, 0.0 };
Display(arr);

ref double arrayElement = ref arr[0];
arrayElement = 3.0;
Display(arr);

arrayElement = ref arr[arr.Length - 1];
arrayElement = 5.0;
Display(arr);
// Output:
// 0 0 0
// 3 0 0
// 3 0 5
```

在前面的示例中，局部引用变量 `arrayElement` 初始化为第一个数组元素的别名。 然后，向其重新赋值 `ref`，以引用最后一个数组元素。 因为它是别名，所以使用普通赋值运算符 `=` 更新其值时，也会更新相应的数组元素。

`ref` 赋值的左侧操作数可以是局部引用变量、ref 字段和 ref、out 或 in 方法参数。 两个操作数的类型必须相同。

复合赋值

对于二元运算符 `op`，窗体的复合赋值表达式

C#

```
x op= y
```

等效于

C#

```
x = x op y
```

不同的是 `x` 只计算一次。

算术、布尔逻辑以及逻辑位和移位运算符支持复合赋值。

Null 合并赋值

只有在左操作数计算为 `null` 时，才能使用 null 合并赋值运算符 `??=` 将其右操作数的值分配给左操作数。 有关详细信息，请参阅 ?? 和 ??= 运算符一文。

运算符可重载性

用户定义类型不能重载赋值运算符。 但是，用户定义类型可以定义到其他类型的隐式转换。 这样，可以将用户定义类型的值分配给其他类型的变量、属性或索引器元素。 有关详细信息，请参阅用户定义转换运算符。

用户定义类型不能显式重载复合赋值运算符。 但是，如果用户定义类型重载了二元运算符 `op`，则 `op=` 运算符（如果存在）也将被隐式重载。

C# 语言规范

有关详细信息，请参阅 C# 语言规范中的分配运算符部分。

另请参阅

- C# 运算符和表达式
- ref 关键字
- 使用复合赋值（样式规则 IDE0054 和 IDE0074）

在 GitHub 上与我们协作

可以在 GitHub 上找到此内容的源，还可以在其中创建和查看问题和拉取请求。 有关详细信息，请参阅参与者指南。

.NET 反馈

.NET 是一个开放源代码项目。 选择一个链接以提供反馈：

提出文档问题

提供产品反馈

Lambda 表达式和匿名函数

项目 • 2025/01/03

使用 *lambda 表达式* 创建匿名函数。 使用 lambda 声明运算符 `=>` 将 lambda 的参数列表与其正文分开。 lambda 表达式可以是以下两种形式中的任何一种：

- 以表达式为主体的表达式 lambda：

  ```
  C#

  (input-parameters) => expression
  ```

- 语句块作为其主体的语句 lambda：

  ```
  C#

  (input-parameters) => { <sequence-of-statements> }
  ```

若要创建 lambda 表达式，请指定 lambda 运算符左侧的输入参数（如果有）和另一端的表达式或语句块。

任何 Lambda 表达式都可以转换为委托类型。 其参数的类型和返回值定义 lambda 表达式可转换为的委托类型。 如果 lambda 表达式未返回值，则可以将其转换为 `Action` 委托类型之一;否则，它可以转换为 `Func` 委托类型之一。 例如，有 2 个参数且不返回值的 Lambda 表达式可转换为 Action<T1,T2> 委托。 有 1 个参数且不返回值的 Lambda 表达式可转换为 Func<T,TResult> 委托。 以下示例中，Lambda 表达式 `x => x * x`（指定名为 `x` 的参数并返回 `x` 平方值）将分配给委托类型的变量：

```
C#

Func<int, int> square = x => x * x;
Console.WriteLine(square(5));
// Output:
// 25
```

表达式 lambda 还可以转换为表达式树类型，如下面的示例所示：

```
C#

System.Linq.Expressions.Expression<Func<int, int>> e = x => x * x;
Console.WriteLine(e);
// Output:
// x => (x * x)
```

您可以在任何需要委托类型或表达式树实例的代码中使用 lambda 表达式。 一个示例是 Task.Run(Action) 方法的参数，用于传递应在后台执行的代码。 在 C#中编写 LINQ 时，也可以使用 lambda 表达式，如以下示例所示：

C#

```
int[] numbers = { 2, 3, 4, 5 };
var squaredNumbers = numbers.Select(x => x * x);
Console.WriteLine(string.Join(" ", squaredNumbers));
// Output:
// 4 9 16 25
```

使用基于方法的语法调用 System.Linq.Enumerable 类中的 Enumerable.Select 方法（例如 LINQ to Objects 和 LINQ to XML 中）时，参数是委托类型 System.Func<T,TResult>。 在 System.Linq.Queryable 类中调用 Queryable.Select 方法（例如在 LINQ to SQL 中）时，参数类型是表达式树类型 Expression<Func<TSource,TResult>>。 在这两种情况下，都可以使用相同的 lambda 表达式来指定参数值。 尽管通过 Lambda 创建的对象实际具有不同的类型，但其使得 2 个 `Select` 调用看起来类似。

表达式 lambda

表达式位于 `=>` 运算符右侧的 lambda 表达式称为“表达式 lambda”。 表达式 lambda 返回表达式的结果，并采用以下基本形式：

C#

```
(input-parameters) => expression
```

表达式 lambda 的主体可以包含方法调用。 不过，若要创建在 .NET 公共语言运行时 (CLR) 的上下文之外（如在 SQL Server 中）计算的表达式树，则不得在 Lambda 表达式中使用方法调用。 这些方法在 .NET 公共语言运行时（CLR）的上下文之外没有意义。

语句 lambda

语句 lambda 与表达式 lambda 类似，只是语句括在大括号中：

C#

```
(input-parameters) => { <sequence-of-statements> }
```

语句 lambda 的正文可以包含任意数量的语句;但是，在实践中通常不超过两到三个。

C#

```
Action<string> greet = name =>
{
    string greeting = $"Hello {name}!";
    Console.WriteLine(greeting);
};
greet("World");
// Output:
// Hello World!
```

不能使用语句 lambda 来创建表达式树。

lambda 表达式的输入参数

将 lambda 表达式的输入参数括在括号中。 使用空括号指定零个输入参数：

C#

```
Action line = () => Console.WriteLine();
```

如果 lambda 表达式只有一个输入参数，则括号是可选的：

C#

```
Func<double, double> cube = x => x * x * x;
```

两个或多个输入参数用逗号分隔：

C#

```
Func<int, int, bool> testForEquality = (x, y) => x == y;
```

有时编译器无法推断输入参数的类型。 可以显式指定类型，如以下示例所示：

C#

```
Func<int, string, bool> isTooLong = (int x, string s) => s.Length > x;
```

输入参数类型必须为所有显式或全部隐式;否则，会发生 CS0748 编译器错误。

可以使用 忽略 来指定 lambda 表达式中不使用的两个或多个输入参数。

C#

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

使用 Lambda 表达式提供事件处理程序时，Lambda 弃元参数可能很有用。

> **备注**
>
> 为了向后兼容，如果只有单个输入参数命名为 `_`，则在 lambda 表达式中，`_` 被视为该参数的名称。

从 C# 12 开始，可以为 lambda 表达式上的参数提供 *默认值*。 默认参数值的语法和限制与方法和本地函数的语法和限制相同。 以下示例使用默认参数声明 lambda 表达式，然后使用默认值调用一次，并使用两个显式参数调用一次：

C#

```
var IncrementBy = (int source, int increment = 1) => source + increment;

Console.WriteLine(IncrementBy(5)); // 6
Console.WriteLine(IncrementBy(5, 2)); // 7
```

还可以将具有 `params` 数组或集合的 lambda 表达式声明为参数：

C#

```
var sum = (params IEnumerable<int> values) =>
{
    int sum = 0;
    foreach (var value in values)
        sum += value;

    return sum;
};

var empty = sum();
Console.WriteLine(empty); // 0

var sequence = new[] { 1, 2, 3, 4, 5 };
var total = sum(sequence);
Console.WriteLine(total); // 15
```

作为这些更新的一部分，当将具有默认参数的方法组分配给 lambda 表达式时，该 lambda 表达式也具有相同的默认参数。 还可以将具有 `params` 集合参数的方法组分配给 Lambda 表达式。

具有默认参数或 `params` 集合的 Lambda 表达式，因为参数没有与 `Func<>` 或 `Action<>` 类型对应的自然类型。 但是，可以定义包含默认参数值的委托类型：

C#

```
delegate int IncrementByDelegate(int source, int increment = 1);
delegate int SumDelegate(params int[] values);
delegate int SumCollectionDelegate(params IEnumerable<int> values);
```

或者，可以将隐式类型化变量与 `var` 声明一起使用来定义委托类型。 编译器合成正确的委托类型。

有关 lambda 表达式的默认参数的详细信息，请参阅 lambda 表达式默认参数的功能规范。

异步 lambda

通过使用 async 和 await 关键字，你可以轻松创建包含异步处理的 lambda 表达式和语句。 例如，下面的 Windows 窗体示例中有一个事件处理程序，它调用并等待一个异步方法，`ExampleMethodAsync`。

C#

```
public partial class Form1 : Form
{
    public Form1()
    {
        InitializeComponent();
        button1.Click += button1_Click;
    }

    private async void button1_Click(object sender, EventArgs e)
    {
        await ExampleMethodAsync();
        textBox1.Text += "\r\nControl returned to Click event handler.\n";
    }

    private async Task ExampleMethodAsync()
    {
        // The following line simulates a task-returning asynchronous
process.
        await Task.Delay(1000);
    }
}
```

你可以使用异步 lambda 添加同一事件处理程序。 若要添加此处理程序，请在 lambda 参数列表之前添加 `async` 修饰符，如以下示例所示：

C#

```
public partial class Form1 : Form
{
    public Form1()
    {
        InitializeComponent();
        button1.Click += async (sender, e) =>
        {
            await ExampleMethodAsync();
            textBox1.Text += "\r\nControl returned to Click event
handler.\n";
        };
    }

    private async Task ExampleMethodAsync()
    {
        // The following line simulates a task-returning asynchronous
process.
        await Task.Delay(1000);
    }
}
```

有关如何创建和使用异步方法的详细信息，请参阅 使用 async 和 await 的异步编程。

lambda 表达式和元组

C# 语言提供对元组的内置支持。 可以将元组作为 lambda 表达式的参数提供，lambda 表达式也可以返回元组。 在某些情况下，C# 编译器使用类型推理来确定元组组件的类型。

可通过用括号括住用逗号分隔的组件列表来定义元组。 以下示例使用包含三个组件的元组将数字序列传递给 lambda 表达式，该表达式将每个值加倍，并返回包含乘法结果的三个组件的元组。

C#

```
Func<(int, int, int), (int, int, int)> doubleThem = ns => (2 * ns.Item1, 2 *
ns.Item2, 2 * ns.Item3);
var numbers = (2, 3, 4);
var doubledNumbers = doubleThem(numbers);
Console.WriteLine($"The set {numbers} doubled: {doubledNumbers}");
// Output:
// The set (2, 3, 4) doubled: (4, 6, 8)
```

通常，元组字段命名为 `Item1`、`Item2` 等等。 但是，可以定义包含命名组件的元组，如以下示例所示。

C#

```
Func<(int n1, int n2, int n3), (int, int, int)> doubleThem = ns => (2 *
ns.n1, 2 * ns.n2, 2 * ns.n3);
var numbers = (2, 3, 4);
var doubledNumbers = doubleThem(numbers);
Console.WriteLine($"The set {numbers} doubled: {doubledNumbers}");
```

有关 C# 元组的详细信息，请参阅 元组类型。

含标准查询运算符的 lambda

在其他实现中，LINQ to Objects 有一个输入参数，其类型是泛型委托 Func<TResult> 系列中的一种。 这些委托使用类型参数来定义输入参数的数量和类型，以及委托的返回类型。 `Func` 委托对于封装用户定义的表达式很有用，这些表达式将应用于一组源数据中的每个元素。 例如，假设为 Func<T,TResult> 委托类型：

C#

```
public delegate TResult Func<in T, out TResult>(T arg)
```

可以将委托实例化为 `Func<int, bool>` 实例，其中 `int` 是输入参数，`bool` 是返回值。 返回值始终在最后一个类型参数中指定。 例如，`Func<int, string, bool>` 定义包含两个输入参数（`int` 和 `string`）且返回类型为 `bool` 的委托。 以下 `Func` 委托在调用时返回一个布尔值，该布尔值指示输入参数是否等于 5。

C#

```
Func<int, bool> equalsFive = x => x == 5;
bool result = equalsFive(4);
Console.WriteLine(result);   // False
```

也可以在参数类型为 Expression<TDelegate>时提供 lambda 表达式，例如，在 Queryable 类型中定义的标准查询运算符中。 指定 Expression<TDelegate> 参数时，lambda 将编译为表达式树。

以下示例使用 Count 标准查询运算符：

C#

```
int[] numbers = { 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 };
int oddNumbers = numbers.Count(n => n % 2 == 1);
Console.WriteLine($"There are {oddNumbers} odd numbers in {string.Join(" ",
numbers)}");
```

编译器可以推断输入参数的类型，也可以显式指定它。 此特定的 lambda 表达式计算那些整数（`n`），这些整数被 2 除后余数为 1。

以下示例生成一个序列，其中包含位于 9 前面的 `numbers` 数组中的所有元素，因为这是序列中不符合条件的第一个数字：

C#

```
int[] numbers = { 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 };
var firstNumbersLessThanSix = numbers.TakeWhile(n => n < 6);
Console.WriteLine(string.Join(" ", firstNumbersLessThanSix));
// Output:
// 5 4 1 3
```

以下示例通过将多个输入参数括在括号中来指定这些参数。 该方法将返回 `numbers` 数组中的所有元素，直到找到一个数字，其值小于数组中的序号位置：

C#

```
int[] numbers = { 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 };
var firstSmallNumbers = numbers.TakeWhile((n, index) => n >= index);
Console.WriteLine(string.Join(" ", firstSmallNumbers));
// Output:
// 5 4
```

不能直接在查询表达式中使用 lambda 表达式，但你可以在查询表达式的方法调用中使用它们，如以下示例所示：

C#

```
var numberSets = new List<int[]>
{
    new[] { 1, 2, 3, 4, 5 },
    new[] { 0, 0, 0 },
    new[] { 9, 8 },
    new[] { 1, 0, 1, 0, 1, 0, 1, 0 }
};

var setsWithManyPositives =
    from numberSet in numberSets
    where numberSet.Count(n => n > 0) > 3
    select numberSet;

foreach (var numberSet in setsWithManyPositives)
{
    Console.WriteLine(string.Join(" ", numberSet));
}
// Output:
```

```
// 1 2 3 4 5
// 1 0 1 0 1 0 1 0
```

lambda 表达式中的类型推理

编写 lambda 时，通常不必为输入参数指定类型，因为编译器可以根据 lambda 正文、参数类型和 C# 语言规范中所述的其他因素推断类型。 对于大多数标准查询运算符，第一个输入是源序列中元素的类型。 如果要查询 `IEnumerable<Customer>`，则输入变量被推断为 `Customer` 对象，这意味着你有权访问其方法和属性：

C#

```
customers.Where(c => c.City == "London");
```

lambda 类型推理的一般规则如下所示：

- lambda 必须包含与委托类型相同的参数数。
- lambda 中的每个输入参数都必须隐式转换为其相应的委托参数。
- Lambda 的返回值（如果有）必须能够隐式转换为委托的返回类型。

lambda 表达式的自然类型

lambda 表达式本身没有类型，因为通用类型系统没有“lambda 表达式”的内部概念。但是，有时可以方便地非正式地谈论 lambda 表达式的“类型”。 该非正式“类型”是指委托类型或 Lambda 表达式所转换到的 Expression 类型。

lambda 表达式可以具有*自然类型*。 编译器不会强制你为 Lambda 表达式声明委托类型（例如 `Func<...>` 或 `Action<...>`），而是根据 Lambda 表达式推断委托类型。 例如，请考虑以下声明：

C#

```
var parse = (string s) => int.Parse(s);
```

编译器可以将 `parse` 推断为 `Func<string, int>`。 编译器选择可用的 `Func` 或 `Action` 委托（如果存在合适的委托）。 否则，它将合成委托类型。 例如，如果 Lambda 表达式具有 `ref` 参数，则合成委托类型。 如果 lambda 表达式具有自然类型，则可以将其分配给不太明确的类型，例如 System.Object 或 System.Delegate：

C#

```
object parse = (string s) => int.Parse(s);   // Func<string, int>
Delegate parse = (string s) => int.Parse(s); // Func<string, int>
```

只有一个重载的方法组（即没有参数列表的方法名称）具有自然类型：

C#

```
var read = Console.Read; // Just one overload; Func<int> inferred
var write = Console.Write; // ERROR: Multiple overloads, can't choose
```

如果将 lambda 表达式分配给 System.Linq.Expressions.LambdaExpression或 System.Linq.Expressions.Expression，并且 lambda 具有自然委托类型，则表达式的自然类型为 System.Linq.Expressions.Expression<TDelegate>，自然委托类型作为类型参数的参数使用。

C#

```
LambdaExpression parseExpr = (string s) => int.Parse(s); //
Expression<Func<string, int>>
Expression parseExpr = (string s) => int.Parse(s);       //
Expression<Func<string, int>>
```

并非所有 lambda 表达式都具有自然类型。 请考虑以下声明：

C#

```
var parse = s => int.Parse(s); // ERROR: Not enough type info in the lambda
```

编译器无法推断 `s`的参数类型。 当编译器无法推断自然类型时，必须声明类型：

C#

```
Func<string, int> parse = s => int.Parse(s);
```

显式返回类型

通常，lambda 表达式的返回类型是显而易见的，并且是推断的。 对于某些不起作用的表达式：

C#

```
var choose = (bool b) => b ? 1 : "two"; // ERROR: Can't infer return type
```

可以在输入参数之前指定 lambda 表达式的返回类型。 指定显式返回类型时，必须括号化输入参数：

```
C#

var choose = object (bool b) => b ? 1 : "two"; // Func<bool, object>
```

属性

可以将属性添加到 lambda 表达式及其参数。 以下示例演示如何向 lambda 表达式添加属性：

```
C#

Func<string?, int?> parse = [ProvidesNullCheck] (s) => (s is not null) ?
int.Parse(s) : null;
```

还可以将属性添加到输入参数或返回值，如以下示例所示：

```
C#

var concat = ([DisallowNull] string a, [DisallowNull] string b) => a + b;
var inc = [return: NotNullIfNotNull(nameof(s))] (int? s) => s.HasValue ? s++
: null;
```

如前面的示例所示，向 lambda 表达式或其参数添加属性时，必须括号化输入参数。

> **ⓘ 重要**
>
> Lambda 表达式通过基础委托类型调用。 这不同于方法和本地函数。 委托的 `Invoke` 方法不会检查 lambda 表达式的属性。 调用 lambda 表达式时，属性没有任何影响。 lambda 表达式上的属性对于代码分析非常有用，可以通过反射发现。 此决定的一个后果是，无法将 System.Diagnostics.ConditionalAttribute 应用于 lambda 表达式。

捕获 lambda 表达式中的外部变量和变量范围

lambda 可以引用外部变量。 外部变量是在定义 lambda 表达式的方法中或包含 lambda 表达式的类型中的范围内变量。 以这种方式捕获的变量将进行存储以备在 lambda 表达式中使用，即使在其他情况下，这些变量将超出范围并进行垃圾回收。 必须明确地分配外部变量，然后才能在 lambda 表达式中使用该变量。 以下示例演示了以下规则：

C#

```
public static class VariableScopeWithLambdas
{
    public class VariableCaptureGame
    {
        internal Action<int>? updateCapturedLocalVariable;
        internal Func<int, bool>? isEqualToCapturedLocalVariable;

        public void Run(int input)
        {
            int j = 0;

            updateCapturedLocalVariable = x =>
            {
                j = x;
                bool result = j > input;
                Console.WriteLine($"{j} is greater than {input}: {result}");
            };

            isEqualToCapturedLocalVariable = x => x == j;

            Console.WriteLine($"Local variable before lambda invocation:
{j}");
            updateCapturedLocalVariable(10);
            Console.WriteLine($"Local variable after lambda invocation:
{j}");
        }
    }

    public static void Main()
    {
        var game = new VariableCaptureGame();

        int gameInput = 5;
        game.Run(gameInput);

        int jTry = 10;
        bool result = game.isEqualToCapturedLocalVariable!(jTry);
        Console.WriteLine($"Captured local variable is equal to {jTry}:
{result}");

        int anotherJ = 3;
        game.updateCapturedLocalVariable!(anotherJ);

        bool equalToAnother = game.isEqualToCapturedLocalVariable(anotherJ);
        Console.WriteLine($"Another lambda observes a new value of captured
variable: {equalToAnother}");
    }
    // Output:
    // Local variable before lambda invocation: 0
    // 10 is greater than 5: True
    // Local variable after lambda invocation: 10
    // Captured local variable is equal to 10: True
```

```
    // 3 is greater than 5: False
    // Another lambda observes a new value of captured variable: True
}
```

以下规则适用于 lambda 表达式中的变量范围：

- 捕获的变量将不会被作为垃圾回收，直至引用变量的委托符合垃圾回收的条件。
- lambda 表达式中引入的变量在封闭方法中不可见。
- Lambda 表达式无法从封闭方法中直接捕获 in、ref 或 out 参数。
- lambda 表达式中的 return 语句不会导致封闭方法返回。
- lambda 表达式不能包含 goto、break或 continue 语句，如果这些跳转语句的目标在 lambda 表达式块之外。 同样，如果目标在块内部，在 lambda 表达式块外部使用跳转语句也是错误的。

可以将 `static` 修饰符应用于 lambda 表达式，以防止 lambda 无意中捕获局部变量或实例状态：

C#

```
Func<double, double> square = static x => x * x;
```

静态 lambda 无法从封闭范围捕获局部变量或实例状态，但可以引用静态成员和常量定义。

C# 语言规范

有关详细信息，请参阅 C# 语言规范的 Anonymous 函数表达式 部分。

有关这些功能的详细信息，请参阅以下功能建议说明：

- Lambda 弃元参数
- 静态匿名函数
- Lambda 改进

另请参阅

- 使用本地函数而不是 lambda （样式规则IDE0039）
- C# 运算符和表达式
- LINQ（语言集成查询）
- 表达式树
- 本地函数与 lambda 表达式
- LINQ 示例查询

- XQuery 示例
- 101 LINQ 示例

模式匹配 - `is` 和 `switch` 表达式以及运算符 `and`， `or` 以及 `not` 模式

项目 • 2024/11/22

使用表达式、switch 语句和 is将输入表达式与任意数量的特征匹配。 C# 支持多种模式，包括声明、类型、常量、关系、属性、列表、var 和弃元。 可以使用布尔逻辑关键字 `and`、 `or` 和 `not` 组合模式。

以下 C# 表达式和语句支持模式匹配：

- is表达式
- switch 语句
- switch 表达式

在这些构造中，可将输入表达式与以下任一模式进行匹配：

- 声明模式：用于检查表达式的运行时类型，如果匹配成功，则将表达式结果分配给声明的变量。
- 类型模式：用于检查表达式的运行时类型。
- 常量模式：测试表达式结果是否等于指定的常量。
- 关系模式：用于将表达式结果与指定常量进行比较。
- 逻辑模式：测试表达式是否与模式的逻辑组合匹配。
- 属性模式：测试表达式的属性或字段是否与嵌套模式匹配。
- 位置模式：用于解构表达式结果并测试结果值是否与嵌套模式匹配。
- var 模式：用于匹配任何表达式并将其结果分配给声明的变量。
- 弃元模式：用于匹配任何表达式。
- 列表模式：测试元素序列是否与相应的嵌套模式匹配。 在 C# 11 中引入。

逻辑、属性、位置和列表模式都是递归模式。 也就是说，它们可包含嵌套模式。

有关如何使用这些模式来生成数据驱动算法的示例，请参阅教程：使用模式匹配来生成类型驱动的算法和数据驱动的算法。

声明和类型模式

使用声明和类型模式检查表达式的运行时类型是否与给定类型兼容。 借助声明模式，还可声明新的局部变量。 当声明模式与表达式匹配时，将为该变量分配转换后的表达式结果，如以下示例所示：

```
C#
```

```
object greeting = "Hello, World!";
if (greeting is string message)
{
    Console.WriteLine(message.ToLower());  // output: hello, world!
}
```

类型为 的声明模式在表达式结果为非 null 且满足以下任一条件时与表达式匹配：

- 表达式结果的运行时类型为 `T`。
- 表达式结果的运行时类型派生自类型 `T`，实现接口 `T`，或者存在从其到 的另一种 `T`。 下面的示例演示满足此条件时的两种案例：

  C#

  ```
  var numbers = new int[] { 10, 20, 30 };
  Console.WriteLine(GetSourceLabel(numbers));  // output: 1

  var letters = new List<char> { 'a', 'b', 'c', 'd' };
  Console.WriteLine(GetSourceLabel(letters));  // output: 2

  static int GetSourceLabel<T>(IEnumerable<T> source) => source switch
  {
      Array array => 1,
      ICollection<T> collection => 2,
      _ => 3,
  };
  ```

  在上述示例中，在第一次调用 `GetSourceLabel` 方法时，第一种模式与参数值匹配，因为参数的运行时类型 `int[]` 派生自 Array 类型。 在第二次调用 `GetSourceLabel` 方法时，参数的运行时类型 List<T> 并非派生自 Array 类型，但却实现 ICollection<T> 接口。

- 表达式结果的运行时类型是具有基础类型 的`T`。
- 存在从表达式结果的运行时类型到类型 的装箱 或`T`转换。

下面的示例演示最后两个条件：

C#

```
int? xNullable = 7;
int y = 23;
object yBoxed = y;
if (xNullable is int a && yBoxed is int b)
{
```

```
    Console.WriteLine(a + b);  // output: 30
}
```

如果只想检查表达式类型，可使用弃元 `_` 代替变量名，如以下示例所示：

C#

```
public abstract class Vehicle {}
public class Car : Vehicle {}
public class Truck : Vehicle {}

public static class TollCalculator
{
    public static decimal CalculateToll(this Vehicle vehicle) => vehicle
switch
    {
        Car _ => 2.00m,
        Truck _ => 7.50m,
        null => throw new ArgumentNullException(nameof(vehicle)),
        _ => throw new ArgumentException("Unknown type of a vehicle",
nameof(vehicle)),
    };
}
```

可对此使用类型模式，如以下示例所示：

C#

```
public static decimal CalculateToll(this Vehicle vehicle) => vehicle switch
{
    Car => 2.00m,
    Truck => 7.50m,
    null => throw new ArgumentNullException(nameof(vehicle)),
    _ => throw new ArgumentException("Unknown type of a vehicle",
nameof(vehicle)),
};
```

与声明模式一样，当表达式结果为非 null 并且其运行时类型满足上述任何条件时，类型模式与表达式匹配。

若要检查非 null，可使用否定 `null` 常量模式，如以下示例所示：

C#

```
if (input is not null)
{
    // ...
}
```

有关详细信息，请参阅功能方案说明的声明模式和类型模式部分。

常量模式

可使用常量模式来测试表达式结果是否等于指定的常量，如以下示例所示：

C#

```
public static decimal GetGroupTicketPrice(int visitorCount) => visitorCount
switch
{
    1 => 12.0m,
    2 => 20.0m,
    3 => 27.0m,
    4 => 32.0m,
    0 => 0.0m,
    _ => throw new ArgumentException($"Not supported number of visitors:
{visitorCount}", nameof(visitorCount)),
};
```

在常量模式中，可使用任何常量表达式，例如：

- integer 或 floating-point 数值文本
- 字符型
- 字符串字面量。
- 布尔值 `true` 或 `false`
- enum 值
- 声明常量字段或本地的名称
- `null`

表达式类型必须可转换为常量类型，但有一个例外：类型为 `Span<char>` 或 `ReadOnlySpan<char>` 的表达式可以在 C# 11 及更高版本中针对常量字符串进行匹配。

常量模式用于检查 `null`，如以下示例所示：

C#

```
if (input is null)
{
    return;
}
```

编译器保证在计算表达式 `==` 时，不会调用用户重载的相等运算符 `x is null`。

可使用否定`null` 常量模式来检查非 null，如以下示例所示：

C#

```
if (input is not null)
{
    // ...
}
```

有关详细信息，请参阅功能建议说明的常量模式部分。

关系模式

请使用关系模式将表达式结果与常量进行比较，如以下示例所示：

C#

```
Console.WriteLine(Classify(13));  // output: Too high
Console.WriteLine(Classify(double.NaN));  // output: Unknown
Console.WriteLine(Classify(2.4));  // output: Acceptable

static string Classify(double measurement) => measurement switch
{
    < -4.0 => "Too low",
    > 10.0 => "Too high",
    double.NaN => "Unknown",
    _ => "Acceptable",
};
```

在关系模式中，可使用关系运算符 `<`、`>`、`<=` 或 `>=` 中的任何一个。 关系模式的右侧部分必须是常数表达式。 常数表达式可以是 integer、floating-point、char 或 enum 类型。

要检查表达式结果是否在某个范围内，请将其与合取 and 模式匹配，如以下示例所示：

C#

```
Console.WriteLine(GetCalendarSeason(new DateTime(2021, 3, 14)));  // output:
spring
Console.WriteLine(GetCalendarSeason(new DateTime(2021, 7, 19)));  // output:
summer
Console.WriteLine(GetCalendarSeason(new DateTime(2021, 2, 17)));  // output:
winter

static string GetCalendarSeason(DateTime date) => date.Month switch
{
    >= 3 and < 6 => "spring",
    >= 6 and < 9 => "summer",
    >= 9 and < 12 => "autumn",
    12 or (>= 1 and < 3) => "winter",
```

```
    _ => throw new ArgumentOutOfRangeException(nameof(date), $"Date with
unexpected month: {date.Month}."),
};
```

如果表达式结果为 `null` 或未能通过可为空或取消装箱转换转换为常量类型，则关系模式与表达式不匹配。

有关详细信息，请参阅功能建议说明的关系模式部分。

逻辑模式

请使用 `not`、`and` 和 `or` 模式连结符来创建以下逻辑模式：

- 否定 模式在否定模式与表达式不匹配时与表达式匹配`not`。 下面的示例说明如何否定常量`null` 模式来检查表达式是否为非空值：

  C#

  ```
  if (input is not null)
  {
      // ...
  }
  ```

- 合取 模式在两个模式都与表达式匹配时与表达式匹配`and`。 以下示例显示如何组合关系模式来检查值是否在某个范围内：

  C#

  ```
  Console.WriteLine(Classify(13));  // output: High
  Console.WriteLine(Classify(-100));  // output: Too low
  Console.WriteLine(Classify(5.7));  // output: Acceptable

  static string Classify(double measurement) => measurement switch
  {
      < -40.0 => "Too low",
      >= -40.0 and < 0 => "Low",
      >= 0 and < 10.0 => "Acceptable",
      >= 10.0 and < 20.0 => "High",
      >= 20.0 => "Too high",
      double.NaN => "Unknown",
  };
  ```

- 析取`or` 模式在任一模式与表达式匹配时与表达式匹配，如以下示例所示：

  C#

```
Console.WriteLine(GetCalendarSeason(new DateTime(2021, 1, 19)));  //
output: winter
Console.WriteLine(GetCalendarSeason(new DateTime(2021, 10, 9)));  //
output: autumn
Console.WriteLine(GetCalendarSeason(new DateTime(2021, 5, 11)));  //
output: spring

static string GetCalendarSeason(DateTime date) => date.Month switch
{
    3 or 4 or 5 => "spring",
    6 or 7 or 8 => "summer",
    9 or 10 or 11 => "autumn",
    12 or 1 or 2 => "winter",
    _ => throw new ArgumentOutOfRangeException(nameof(date), $"Date
with unexpected month: {date.Month}."),
};
```

如前面的示例所示，可在模式中重复使用模式连结符。

检查的优先级和顺序

模式组合器根据表达式的绑定顺序进行排序，如下所示：

- `not`
- `and`
- `or`

该 `not` 模式首先绑定到其操作数。 模式 `and` 绑定在任何模式表达式绑定之后 `not` 。 所有 `or` 模式后 `not` 绑定，模式 `and` 绑定到操作数。 以下示例尝试匹配不小写字母 `a` - `z` 的所有字符。 出现错误，因为 `not` 模式在模式之前 `and` 绑定：

C#

```
// Incorrect pattern. `not` binds before `and`
static bool IsNotLowerCaseLetter(char c) => c is not >= 'a' and <= 'z';
```

默认绑定表示将分析前面的示例，如以下示例所示：

C#

```
// The default binding without parentheses is shows in this method. `not`
binds before `and`
static bool IsNotLowerCaseLetterDefaultBinding(char c) => c is ((not >= 'a')
and <= 'z');
```

若要修复此问题，必须指定要 `not` 绑定到 `>= 'a' and <= 'z'` 表达式：

C#

```
// Correct pattern. Force `and` before `not`
static bool IsNotLowerCaseLetterParentheses(char c) => c is not (>= 'a' and
<= 'z');
```

添加括号变得更加重要，因为模式变得更加复杂。 一般情况下，使用括号来阐明其他开发人员的模式，如以下示例所示：

C#

```
static bool IsLetter(char c) => c is (>= 'a' and <= 'z') or (>= 'A' and <=
'Z');
```

> **备注**
>
> 检查具有相同绑定顺序的模式的顺序是未定义的。 在运行时，可以先检查多个 `or` 模式和多个 `and` 模式的右侧嵌套模式。

有关详细信息，请参阅功能建议说明的模式连结符部分。

属性模式

可以使用属性模式将表达式的属性或字段与嵌套模式进行匹配，如以下示例所示：

C#

```
static bool IsConferenceDay(DateTime date) => date is { Year: 2020, Month:
5, Day: 19 or 20 or 21 };
```

当表达式结果为非 NULL 且每个嵌套模式都与表达式结果的相应属性或字段匹配时，属性模式将与表达式匹配。

还可将运行时类型检查和变量声明添加到属性模式，如以下示例所示：

C#

```
Console.WriteLine(TakeFive("Hello, world!"));  // output: Hello
Console.WriteLine(TakeFive("Hi!"));  // output: Hi!
Console.WriteLine(TakeFive(new[] { '1', '2', '3', '4', '5', '6', '7' }));
// output: 12345
Console.WriteLine(TakeFive(new[] { 'a', 'b', 'c' }));  // output: abc

static string TakeFive(object input) => input switch
```

```
{
    string { Length: >= 5 } s => s.Substring(0, 5),
    string s => s,

    ICollection<char> { Count: >= 5 } symbols => new
string(symbols.Take(5).ToArray()),
    ICollection<char> symbols => new string(symbols.ToArray()),

    null => throw new ArgumentNullException(nameof(input)),
    _ => throw new ArgumentException("Not supported input type."),
};
```

属性模式是一种递归模式。 也就是说，可以将任何模式用作嵌套模式。 使用属性模式将部分数据与嵌套模式进行匹配，如以下示例所示：

C#

```
public record Point(int X, int Y);
public record Segment(Point Start, Point End);

static bool IsAnyEndOnXAxis(Segment segment) =>
    segment is { Start: { Y: 0 } } or { End: { Y: 0 } };
```

上一示例使用 `or` 模式结合法和记录类型。

可以在属性模式中引用嵌套属性或字段。 该功能称为“扩展属性模式”。 例如，可将上述示例中的方法重构为以下等效代码：

C#

```
static bool IsAnyEndOnXAxis(Segment segment) =>
    segment is { Start.Y: 0 } or { End.Y: 0 };
```

有关详细信息，请参阅功能建议说明的属性模式部分和扩展属性模式功能建议说明。

> **提示**
>
> 可以使用**简化属性模式 (IDE0170)** 样式规则，通过建议使用扩展属性模式的位置来提高代码的可读性。

位置模式

可使用位置模式来解构表达式结果并将结果值与相应的嵌套模式匹配，如以下示例所示：

C#

```
public readonly struct Point
{
    public int X { get; }
    public int Y { get; }

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

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

static string Classify(Point point) => point switch
{
    (0, 0) => "Origin",
    (1, 0) => "positive X basis end",
    (0, 1) => "positive Y basis end",
    _ => "Just a point",
};
```

在前面的示例中，表达式的类型包含 Deconstruct 方法，该方法用于解构表达式结果。

> **ⓘ 重要**
>
> 位置模式中成员的顺序必须与 `Deconstruct` 方法中的参数顺序匹配。 这是因为为位置模式生成的代码调用 `Deconstruct` 方法。

还可将元组类型的表达式与位置模式进行匹配。 这样，就可将多个输入与各种模式进行匹配，如以下示例所示：

C#

```
static decimal GetGroupTicketPriceDiscount(int groupSize, DateTime
visitDate)
    => (groupSize, visitDate.DayOfWeek) switch
    {
        (<= 0, _) => throw new ArgumentException("Group size must be
positive."),
        (_, DayOfWeek.Saturday or DayOfWeek.Sunday) => 0.0m,
        (>= 5 and < 10, DayOfWeek.Monday) => 20.0m,
        (>= 10, DayOfWeek.Monday) => 30.0m,
        (>= 5 and < 10, _) => 12.0m,
        (>= 10, _) => 15.0m,
        _ => 0.0m,
    };
```

上一示例使用关系和逻辑模式。

可在位置模式中使用元组元素的名称和 `Deconstruct` 参数，如以下示例所示：

C#

```
var numbers = new List<int> { 1, 2, 3 };
if (SumAndCount(numbers) is (Sum: var sum, Count: > 0))
{
    Console.WriteLine($"Sum of [{string.Join(" ", numbers)}] is {sum}");  //
output: Sum of [1 2 3] is 6
}

static (double Sum, int Count) SumAndCount(IEnumerable<int> numbers)
{
    int sum = 0;
    int count = 0;
    foreach (int number in numbers)
    {
        sum += number;
        count++;
    }
    return (sum, count);
}
```

还可通过以下任一方式扩展位置模式：

- 添加运行时类型检查和变量声明，如以下示例所示：

  C#

  ```
  public record Point2D(int X, int Y);
  public record Point3D(int X, int Y, int Z);

  static string PrintIfAllCoordinatesArePositive(object point) => point
  switch
  {
      Point2D (> 0, > 0) p => p.ToString(),
      Point3D (> 0, > 0, > 0) p => p.ToString(),
      _ => string.Empty,
  };
  ```

  前面的示例使用隐式提供 方法的`Deconstruct`。

- 在位置模式中使用属性模式，如以下示例所示：

  C#

  ```
  public record WeightedPoint(int X, int Y)
  {
      public double Weight { get; set; }
  }
  ```

```
static bool IsInDomain(WeightedPoint point) => point is (>= 0, >= 0) {
Weight: >= 0.0 };
```

- 结合前面的两种用法，如以下示例所示：

```
C#

if (input is WeightedPoint (> 0, > 0) { Weight: > 0.0 } p)
{
    // ..
}
```

位置模式是一种递归模式。 也就是说，可以将任何模式用作嵌套模式。

有关详细信息，请参阅功能建议说明的位置模式部分。

`var` 模式

可使用 `var` 模式来匹配任何表达式（包括 ），并将其结果分配给新的局部变量，如以下示例所示：

```
C#

static bool IsAcceptable(int id, int absLimit) =>
    SimulateDataFetch(id) is var results
    && results.Min() >= -absLimit
    && results.Max() <= absLimit;

static int[] SimulateDataFetch(int id)
{
    var rand = new Random();
    return Enumerable
                .Range(start: 0, count: 5)
                .Select(s => rand.Next(minValue: -10, maxValue: 11))
                .ToArray();
}
```

需要布尔表达式中的临时变量来保存中间计算的结果时，`var` 模式很有用。 当需要在 `var` 表达式或语句的 `when` 大小写临界子句中执行更多检查时，也可使用 `switch` 模式，如以下示例所示：

```
C#

public record Point(int X, int Y);

static Point Transform(Point point) => point switch
{
```

```
    var (x, y) when x < y => new Point(-x, y),
    var (x, y) when x > y => new Point(x, -y),
    var (x, y) => new Point(x, y),
};

static void TestTransform()
{
    Console.WriteLine(Transform(new Point(1, 2)));  // output: Point { X =
-1, Y = 2 }
    Console.WriteLine(Transform(new Point(5, 2)));  // output: Point { X =
5, Y = -2 }
}
```

在前面的示例中，模式 `var (x, y)` 等效于位置模式 `(var x, var y)`。

在 `var` 模式中，声明变量的类型是与该模式匹配的表达式的编译时类型。

有关详细信息，请参阅功能建议说明的 Var 模式部分。

弃元模式

可使用弃元模式 来匹配任何表达式，包括 `_`，如以下示例所示 `null`：

C#

```
Console.WriteLine(GetDiscountInPercent(DayOfWeek.Friday));  // output: 5.0
Console.WriteLine(GetDiscountInPercent(null));  // output: 0.0
Console.WriteLine(GetDiscountInPercent((DayOfWeek)10));  // output: 0.0

static decimal GetDiscountInPercent(DayOfWeek? dayOfWeek) => dayOfWeek
switch
{
    DayOfWeek.Monday => 0.5m,
    DayOfWeek.Tuesday => 12.5m,
    DayOfWeek.Wednesday => 7.5m,
    DayOfWeek.Thursday => 12.5m,
    DayOfWeek.Friday => 5.0m,
    DayOfWeek.Saturday => 2.5m,
    DayOfWeek.Sunday => 2.0m,
    _ => 0.0m,
};
```

在前面的示例中，弃元模式用于处理 `null` 以及没有相应的 DayOfWeek 枚举成员的任何整数值。 这可保证示例中的 `switch` 表达式可处理所有可能的输入值。 如果没有在 `switch` 表达式中使用弃元模式，并且该表达式的任何模式均与输入不匹配，则运行时会引发异常。 如果 `switch` 表达式未处理所有可能的输入值，则编译器会生成警告。

弃元模式不能是 `is` 表达式或 `switch` 语句中的模式。 在这些案例中，要匹配任何表达式，请使用带有弃元 `var` 的 `var _`。 弃元模式可以是表达式 `switch` 中的模式。

有关详细信息，请参阅功能建议说明的弃元模式部分。

带括号模式

可在任何模式两边加上括号。 通常，这样做是为了强调或更改逻辑模式中的优先级，如以下示例所示：

C#

```
if (input is not (float or double))
{
    return;
}
```

列表模式

从 C# 11 开始，可以将数组或列表与模式的序列进行匹配，如以下示例所示：

C#

```
int[] numbers = { 1, 2, 3 };

Console.WriteLine(numbers is [1, 2, 3]);  // True
Console.WriteLine(numbers is [1, 2, 4]);  // False
Console.WriteLine(numbers is [1, 2, 3, 4]);  // False
Console.WriteLine(numbers is [0 or 1, <= 2, >= 3]);  // True
```

如前面的示例所示，当每个嵌套模式与输入序列的相应元素匹配时，都会匹配列表模式。 可使用列表模式中的任何模式。 若要匹配任何元素，请使用弃元模式，或者，如果还想捕获元素，请使用 var 模式，如以下示例所示：

C#

```
List<int> numbers = new() { 1, 2, 3 };

if (numbers is [var first, _, _])
{
    Console.WriteLine($"The first element of a three-item list is
{first}.");
}
// Output:
// The first element of a three-item list is 1.
```

前面的示例将整个输入序列与列表模式匹配。 若要仅匹配输入序列开头或/和结尾的元素，请使用切片模式，如以下示例所示`..`：

C#

```
Console.WriteLine(new[] { 1, 2, 3, 4, 5 } is [> 0, > 0, ..]);  // True
Console.WriteLine(new[] { 1, 1 } is [_, _, ..]);  // True
Console.WriteLine(new[] { 0, 1, 2, 3, 4 } is [> 0, > 0, ..]);  // False
Console.WriteLine(new[] { 1 } is [1, 2, ..]);  // False

Console.WriteLine(new[] { 1, 2, 3, 4 } is [.., > 0, > 0]);  // True
Console.WriteLine(new[] { 2, 4 } is [.., > 0, 2, 4]);  // False
Console.WriteLine(new[] { 2, 4 } is [.., 2, 4]);  // True

Console.WriteLine(new[] { 1, 2, 3, 4 } is [>= 0, .., 2 or 4]);  // True
Console.WriteLine(new[] { 1, 0, 0, 1 } is [1, 0, .., 0, 1]);  // True
Console.WriteLine(new[] { 1, 0, 1 } is [1, 0, .., 0, 1]);  // False
```

切片模式匹配零个或多个元素。 最多可在列表模式中使用一个切片模式。 切片模式只能显示在列表模式中。

还可以在切片模式中嵌套子模式，如以下示例所示：

C#

```
void MatchMessage(string message)
{
    var result = message is ['a' or 'A', .. var s, 'a' or 'A']
        ? $"Message {message} matches; inner part is {s}."
        : $"Message {message} doesn't match.";
    Console.WriteLine(result);
}

MatchMessage("aBBA");  // output: Message aBBA matches; inner part is BB.
MatchMessage("apron");  // output: Message apron doesn't match.

void Validate(int[] numbers)
{
    var result = numbers is [< 0, .. { Length: 2 or 4 }, > 0] ? "valid" :
"not valid";
    Console.WriteLine(result);
}

Validate(new[] { -1, 0, 1 });  // output: not valid
Validate(new[] { -1, 0, 0, 1 });  // output: valid
```

有关详细信息，请参阅列表模式功能建议说明。

C# 语言规范

有关详细信息，请参阅 C# 语言规范的模式和模式匹配部分。

若要了解 C# 8 及更高版本中添加的功能，请参阅以下功能建议说明：

- 递归模式匹配
- 模式匹配更新
- 扩展属性模式
- C# 11 - 列出模式
- C# 11 - 字符串字面量上的模式匹配 Span<char>

另请参阅

- C# 运算符和表达式
- 模式匹配概述
- 教程：使用模式匹配来构建类型驱动和数据驱动的算法

加法运算符 - `+` 和 `+=`

项目 • 2024/04/02

内置整型和浮点数值类型、字符串类型以及委托类型支持 `+` 和 `+=` 运算符。

有关算术 `+` 运算符的信息，请参阅一元加和减运算符和算术运算符文章的加法运算符 + 部分。

字符串串联

当其中的一个操作数是字符串类型或两个操作数都是字符串类型时，`+` 运算符将其操作数的字符串表示形式（`null` 的字符串表示形式）串联在一起：

C#

```
Console.WriteLine("Forgot" + "white space");
Console.WriteLine("Probably the oldest constant: " + Math.PI);
Console.WriteLine(null + "Nothing to add.");
// Output:
// Forgotwhite space
// Probably the oldest constant: 3.14159265358979
// Nothing to add.
```

字符串内插提供了格式化字符串的更便捷方式：

C#

```
Console.WriteLine($"Probably the oldest constant: {Math.PI:F2}");
// Output:
// Probably the oldest constant: 3.14
```

从 C# 10 开始，当用于占位符的所有表达式也是常量字符串时，可以使用字符串内插来初始化常量字符串。

从 C# 11 开始，`+` 运算符对 UTF-8 文本字符串执行字符串串联。 此运算符串联两个 `ReadOnlySpan<byte>` 对象。

委托组合

对于委托类型相同的操作数，`+` 运算符在调用时返回新的委托实例，调用左侧的操作数，然后调用右侧的操作数。 如果任何操作数均为 `null`，则 `+` 运算符将返回另一个操作数（也可能是 `null`）的值。 下面的示例演示如何组合使用委托和 `+` 运算符：

C#

```
Action a = () => Console.Write("a");
Action b = () => Console.Write("b");
Action ab = a + b;
ab();  // output: ab
```

若要执行委托删除，请使用 - 运算符。

有关委托类型的详细信息，请参阅委托。

加法赋值运算符 +=

使用 `+=` 运算符的表达式，例如

C#

```
x += y
```

等效于

C#

```
x = x + y
```

不同的是 `x` 只计算一次。

下面的示例演示 `+=` 运算符的用法：

C#

```
int i = 5;
i += 9;
Console.WriteLine(i);
// Output: 14

string story = "Start. ";
story += "End.";
Console.WriteLine(story);
// Output: Start. End.

Action printer = () => Console.Write("a");
printer();  // output: a

Console.WriteLine();
```

```
printer += () => Console.Write("b");
printer();  // output: ab
```

在订阅事件时，还可以使用 `+=` 运算符来指定事件处理程序方法。 有关详细信息，请参阅如何：订阅和取消订阅事件。

运算符可重载性

用户定义的类型可以重载 `+` 运算符。 重载二元 `+` 运算符后，也会隐式重载 `+=` 运算符。 用户定义类型不能显式重载 `+=` 运算符。

C# 语言规范

有关详细信息，请参阅C# 语言规范的一元加运算符和加法运算符部分。

另请参阅

- C# 运算符和表达式
- 如何连接多个字符串
- 事件
- 算术运算符
- - 和 -= 运算符

在 GitHub 上与我们协作

可以在 GitHub 上找到此内容的源，还可以在其中创建和查看问题和拉取请求。 有关详细信息，请参阅参与者指南。

.NET 反馈

.NET 是一个开放源代码项目。 选择一个链接以提供反馈：

提出文档问题

提供产品反馈

- 和 -= 运算符 - 减法

项目 • 2024/04/02

内置整型和浮点数字类型以及委托类型支持 `-` 和 `-=` 运算符。

有关算术 `-` 运算符的信息，请参阅一元加和减运算符和算术运算符文章的减法运算符 - 部分。

委托删除

对于委托类型相同的操作数，`-` 运算符返回如下计算的委托实例：

- 如果两个操作数都为非空，并且右侧操作数的调用列表是左侧操作数调用列表的正确连续子列表，则该操作的结果是通过从左侧操作数的调用列表中删除右侧操作数的条目而获得的新调用列表。 如果右侧操作数的列表与左侧操作数列表中的多个连续子列表匹配，则仅删除最右侧的匹配子列表。 如果删除行为导致出现空列表，则结果为 `null`。

  C#

  ```
  Action a = () => Console.Write("a");
  Action b = () => Console.Write("b");

  var abbaab = a + b + b + a + a + b;
  abbaab();  // output: abbaab
  Console.WriteLine();

  var ab = a + b;
  var abba = abbaab - ab;
  abba();  // output: abba
  Console.WriteLine();

  var nihil = abbaab - abbaab;
  Console.WriteLine(nihil is null);  // output: True
  ```

- 如果右侧操作数的调用列表不是左侧操作数调用列表的正确连续子列表，则该操作的结果是左侧操作数。 例如，删除不属于多播委托的委托不会执行任何操作，从而导致不变的多播委托。

  C#

  ```
  Action a = () => Console.Write("a");
  Action b = () => Console.Write("b");

  var abbaab = a + b + b + a + a + b;
  ```

```
var aba = a + b + a;

var first = abbaab - aba;
first();  // output: abbaab
Console.WriteLine();
Console.WriteLine(object.ReferenceEquals(abbaab, first));  // output:
True

Action a2 = () => Console.Write("a");
var changed = aba - a;
changed();  // output: ab
Console.WriteLine();
var unchanged = aba - a2;
unchanged();  // output: aba
Console.WriteLine();
Console.WriteLine(object.ReferenceEquals(aba, unchanged));  // output:
True
```

前面的示例还演示了在删除委托期间对委托实例进行比较。 例如，通过计算相同的 Lambda 表达式生成的委托不相等。 有关委托相等性的详细信息，请参阅 C# 语言规范的委托相等运算符部分。

- 如果左侧操作数为 `null`，则操作结果为 `null`。 如果右侧操作数为 `null`，则操作的结果是左侧操作数。

```
C#

Action a = () => Console.Write("a");

var nothing = null - a;
Console.WriteLine(nothing is null);  // output: True

var first = a - null;
a();  // output: a
Console.WriteLine();
Console.WriteLine(object.ReferenceEquals(first, a));  // output: True
```

若要合并委托，请使用 + 运算符。

有关委托类型的详细信息，请参阅委托。

减法赋值运算符 -=

使用 `-=` 运算符的表达式，例如

```
C#
```

```
x -= y
```

等效于

C#

```
x = x - y
```

不同的是 `x` 只计算一次。

下面的示例演示 `-=` 运算符的用法：

C#

```
int i = 5;
i -= 9;
Console.WriteLine(i);
// Output: -4

Action a = () => Console.Write("a");
Action b = () => Console.Write("b");
var printer = a + b + a;
printer();  // output: aba

Console.WriteLine();
printer -= a;
printer();  // output: ab
```

还可以使用 `-=` 运算符指定在取消订阅事件时要删除的事件处理程序方法。 有关详细信息，请参阅如何订阅和取消订阅事件。

运算符可重载性

用户定义的类型可以重载 `-` 运算符。 重载二元 `-` 运算符后，也会隐式重载 `-=` 运算符。 用户定义类型不能显式重载 `-=` 运算符。

C# 语言规范

有关详细信息，请参阅 C# 语言规范的一元减运算符和减法运算符部分。

另请参阅

- C# 运算符和表达式

- 事件
- 算术运算符
- + 和 += 运算符

在 GitHub 上与我们协作

可以在 GitHub 上找到此内容的源，还可以在其中创建和查看问题和拉取请求。 有关详细信息，请参阅参与者指南。

.NET 反馈

.NET 是一个开放源代码项目。 选择一个链接以提供反馈：

提出文档问题

提供产品反馈

?: 运算符 - 三元条件运算符

项目 • 2024/07/04

条件运算符 (`?:`) 也称为三元条件运算符，用于计算布尔表达式，并根据布尔表达式的计算结果为 `true` 还是 `false` 来返回两个表达式中的一个结果，如以下示例所示：

C#

```
string GetWeatherDisplay(double tempInCelsius) => tempInCelsius < 20.0 ?
"Cold." : "Perfect!";

Console.WriteLine(GetWeatherDisplay(15));  // output: Cold.
Console.WriteLine(GetWeatherDisplay(27));  // output: Perfect!
```

如上述示例所示，条件运算符的语法如下所示：

C#

```
condition ? consequent : alternative
```

`condition` 表达式的计算结果必须为 `true` 或 `false`。 若 `condition` 的计算结果为 `true`，将计算 `consequent`，其结果成为运算结果。 若 `condition` 的计算结果为 `false`，将计算 `alternative`，其结果成为运算结果。 只会计算 `consequent` 或 `alternative`。 条件表达式是目标类型的。 也就是说，如果条件表达式的目标类型是已知的，则 `consequent` 和 `alternative` 的类型必须可隐式转换为目标类型，如以下示例所示：

C#

```
var rand = new Random();
var condition = rand.NextDouble() > 0.5;

int? x = condition ? 12 : null;

IEnumerable<int> xs = x is null ? new List<int>() { 0, 1 } : new int[] { 2,
3 };
```

如果条件表达式的目标类型未知（例如使用 var 关键字时）或 `consequent` 和 `alternative` 的类型必须相同，或者必须存在从一种类型到另一种类型的隐式转换：

C#

```
var rand = new Random();
var condition = rand.NextDouble() > 0.5;
```

```
var x = condition ? 12 : (int?)null;
```

条件运算符为右联运算符，即形式的表达式

C#

```
a ? b : c ? d : e
```

计算结果为

C#

```
a ? b : (c ? d : e)
```

提示

可以使用以下助记键设备记住条件运算符的计算方式：

text

```
is this condition true ? yes : no
```

ref 条件表达式

条件 ref 表达式可有条件地返回变量引用，如以下示例所示：

C#

```
int[] smallArray = {1, 2, 3, 4, 5};
int[] largeArray = {10, 20, 30, 40, 50};

int index = 7;
ref int refValue = ref ((index < 5) ? ref smallArray[index] : ref
largeArray[index - 5]);
refValue = 0;

index = 2;
((index < 5) ? ref smallArray[index] : ref largeArray[index - 5]) = 100;

Console.WriteLine(string.Join(" ", smallArray));
Console.WriteLine(string.Join(" ", largeArray));
// Output:
```

```
// 1 2 100 4 5
// 10 20 0 40 50
```

可以ref 分配条件 ref 表达式的结果，将其用作引用返回，或将其作为 `ref`、`out`、`in` 或 `ref readonly` 方法参数传递。 还可以分配条件 ref 表达式的结果，如前面的示例所示。

ref 条件表达式的语法如下所示：

```
C#

condition ? ref consequent : ref alternative
```

条件 ref 表达式与条件运算符相似，仅计算两个表达式其中之一：`consequent` 或 `alternative`。

在 ref 条件表达式中，`consequent` 和 `alternative` 的类型必须相同。 ref 条件表达式不由目标确定类型。

条件运算符和 `if` 语句

需要根据条件计算值时，使用条件运算符而不是 if 语句可以使代码更简洁。 下面的示例演示了将整数归类为负数或非负数的两种方法：

```
C#

int input = new Random().Next(-5, 5);

string classify;
if (input >= 0)
{
    classify = "nonnegative";
}
else
{
    classify = "negative";
}

classify = (input >= 0) ? "nonnegative" : "negative";
```

运算符可重载性

用户定义类型不能重载条件运算符。

C# 语言规范

有关详细信息，请参阅 C# 语言规范的条件运算符部分。

较新版本功能的规范如下：

- 目标类型的条件表达式

另请参阅

- 简化条件表达式（样式规则 IDE0075）
- C# 运算符和表达式
- if 语句
- ?. 和 ?[] 运算符
- ?? 和 ??= 运算符
- ref 关键字

在 GitHub 上与我们协作

可以在 GitHub 上找到此内容的源，还可以在其中创建和查看问题和拉取请求。 有关详细信息，请参阅参与者指南。

.NET

.NET 反馈

.NET 是一个开放源代码项目。 选择一个链接以提供反馈：

提出文档问题

提供产品反馈

! （null 包容）运算符（C# 参考）

项目 • 2023/04/07

一元后缀 `!` 运算符是 null 包容运算符或 null 抑制运算符。 在已启用的可为空的注释上下文中，使用 null 包容运算符来取消上述表达式的所有可为 null 警告。 一元前缀 `!` 运算符是逻辑非运算符。 null 包容运算符在运行时不起作用。 它仅通过更改表达式的 null 状态来影响编译器的静态流分析。 在运行时，表达式 `x!` 的计算结果为基础表达式 `x` 的结果。

有关可为空引用类型特性的详细信息，请参见可为空引用类型。

示例

null 包容运算符的一个用例是测试参数验证逻辑。 例如，请考虑以下类：

C#

```
#nullable enable
public class Person
{
    public Person(string name) => Name = name ?? throw new
ArgumentNullException(nameof(name));

    public string Name { get; }
}
```

使用 测试框架，可以在构造函数中为验证逻辑创建以下测试：

C#

```
[TestMethod, ExpectedException(typeof(ArgumentNullException))]
public void NullNameShouldThrowTest()
{
    var person = new Person(null!);
}
```

如果不使用 null 包容运算符，编译器将为前面的代码生成以下警告：`Warning CS8625: Cannot convert null literal to non-nullable reference type`。 通过使用 null 包容运算符，可以告知编译器传递 `null` 是预期行为，不应发出警告。

如果你明确知道某个表达式不能为 `null`，但编译器无法识别它，也可以使用 null 包容运算符。 在下面的示例中，如果 `IsValid` 方法返回 `true`，则其参数不是 `null`，可以放心取消对它的引用：

C#

```
public static void Main()
{
    Person? p = Find("John");
    if (IsValid(p))
    {
        Console.WriteLine($"Found {p!.Name}");
    }
}

public static bool IsValid(Person? person)
    => person is not null && person.Name is not null;
```

如果没有 null 包容运算符，编译器将为 `p.Name` 代码生成以下警告：`Warning CS8602: Dereference of a possibly null reference`。

如果可以修改 `IsValid` 方法，则可使用 NotNullWhen 属性告知编译器，当方法返回 `true` 时，`IsValid` 方法的参数不能是 `null`：

C#

```
public static void Main()
{
    Person? p = Find("John");
    if (IsValid(p))
    {
        Console.WriteLine($"Found {p.Name}");
    }
}

public static bool IsValid([NotNullWhen(true)] Person? person)
    => person is not null && person.Name is not null;
```

在前面的例子中，不需要使用 null 包容运算符，因为编译器有足够的信息来发现 `p` 不能是 `if` 语句中的 `null`。 如需深入了解允许你提供有关变量 null 状态的其他信息的属性，请参阅使用属性升级 API 以定义 null 期望值。

C# 语言规范

有关详细信息，请参阅可为空的引用类型规范草案的 null 包容性运算符部分。

请参阅

- 删除不必要的抑制运算符（样式规则 IDE0080）

- C# 参考
- C# 运算符和表达式
- 教程：使用可为空引用类型进行设计

?? 和 ??= 运算符 - Null 合并操作符

项目 • 2023/07/28

如果左操作数的值不为 `null`，则 null 合并运算符 `??` 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 `??` 运算符不会计算其右操作数。 仅当左操作数的计算结果为 `null` 时，Null 合并赋值运算符 `??=` 才会将其右操作数的值赋值给其左操作数。 如果左操作数的计算结果为非 null，则 `??=` 运算符不会计算其右操作数。

C#

```
List<int> numbers = null;
int? a = null;

Console.WriteLine((numbers is null)); // expected: true
// if numbers is null, initialize it. Then, add 5 to numbers
(numbers ??= new List<int>()).Add(5);
Console.WriteLine(string.Join(" ", numbers));  // output: 5
Console.WriteLine((numbers is null)); // expected: false


Console.WriteLine((a is null)); // expected: true
Console.WriteLine((a ?? 3)); // expected: 3 since a is still null
// if a is null then assign 0 to a and add a to the list
numbers.Add(a ??= 0);
Console.WriteLine((a is null)); // expected: false
Console.WriteLine(string.Join(" ", numbers));  // output: 5 0
Console.WriteLine(a);  // output: 0
```

`??=` 运算符的左操作数必须是变量、属性或索引器元素。

`??` 和 `??=` 运算符的左操作数的类型必须是可以为 null 的值类型。 特别是，可以使用具有无约束类型参数的 null 合并运算符：

C#

```
private static void Display<T>(T a, T backup)
{
    Console.WriteLine(a ?? backup);
}
```

null 合并运算符是右结合运算符。 也就是说，是窗体的表达式

C#

```
a ?? b ?? c
d ??= e ??= f
```

会像这样求值

```
C#

a ?? (b ?? c)
d ??= (e ??= f)
```

示例

`??` 和 `??=` 运算符在以下应用场景中很有用：

- 在包含 null 条件运算符 ?. 和 ?[] 的表达式中，当包含 null 条件运算的表达式结果为 `null` 时，可以使用 `??` 运算符来提供替代表达式用于求值：

  ```
  C#

  double SumNumbers(List<double[]> setsOfNumbers, int indexOfSetToSum)
  {
      return setsOfNumbers?[indexOfSetToSum]?.Sum() ?? double.NaN;
  }

  var sum = SumNumbers(null, 0);
  Console.WriteLine(sum);  // output: NaN
  ```

- 当使用可以为 null 值类型并且需要提供基础值类型的值时，可以使用 `??` 运算符指定当可以为 null 的类型的值为 `null` 时要提供的值：

  ```
  C#

  int? a = null;
  int b = a ?? -1;
  Console.WriteLine(b);  // output: -1
  ```

  如果可以为 null 的类型的值为 `null` 时要使用的值应为基础值类型的默认值，请使用 Nullable<T>.GetValueOrDefault() 方法。

- 可以使用 throw 表达式作为 `??` 运算符的右操作数，以使参数检查代码更简洁：

  ```
  C#
  ```

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

前面的示例还演示了如何使用 expression-bodied 成员来定义属性。

- 可以使用 `??=` 运算符替换窗体的代码

```
C#
```

```
if (variable is null)
{
    variable = expression;
}
```

替换为以下代码：

```
C#
```

```
variable ??= expression;
```

运算符可重载性

运算符 `??` 和 `??=` 无法进行重载。

C# 语言规范

有关 `??` 运算符的详细信息，请参阅 C# 语言规范的 null 合并运算符部分。

有关 `??=` 运算符的详细信息，请参阅功能建议说明。

另请参阅

- 可以简化 Null 检查（IDE0029、IDE0030 和 IDE0270）
- C# 参考
- C# 运算符和表达式
- ?. 和 ?[] 运算符
- ?: 运算符

lambda 表达式 (`=>`) 运算符定义 lambda 表达式

项目 • 2023/04/07

`=>` 令牌支持两种形式：作为 lambda 运算符、作为成员名称的分隔符和表达式主体定义中的成员实现。

lambda 运算符

在 lambda 表达式中，lambda 运算符 `=>` 将左侧的输入参数与右侧的 lambda 主体分开。

以下示例使用带有方法语法的 LINQ 功能来演示 lambda 表达式的用法：

C#

```
string[] words = { "bot", "apple", "apricot" };
int minimalLength = words
  .Where(w => w.StartsWith("a"))
  .Min(w => w.Length);
Console.WriteLine(minimalLength);   // output: 5

int[] numbers = { 4, 7, 10 };
int product = numbers.Aggregate(1, (interim, next) => interim * next);
Console.WriteLine(product);   // output: 280
```

lambda 表达式的输入参数在编译时是强类型。 当编译器可以推断输入参数的类型时，如前面的示例所示，可以省略类型声明。 如果需要指定输入参数的类型，则必须对每个参数执行类型声明，如以下示例所示：

C#

```
int[] numbers = { 4, 7, 10 };
int product = numbers.Aggregate(1, (int interim, int next) => interim *
next);
Console.WriteLine(product);   // output: 280
```

以下示例显示如何在没有输入参数的情况下定义 lambda 表达式：

C#

```
Func<string> greet = () => "Hello, World!";
Console.WriteLine(greet());
```

有关详细信息，请参阅 Lambda 表达式。

表达式主体定义

表达式主体定义具有下列常规语法：

```
C#

member => expression;
```

其中 `expression` 是有效的表达式。 `expression` 的返回类型必须可隐式转换为成员的返回类型。 如果成员：

- 具有 `void` 返回类型或
- 是一个：
  - 构造函数
  - 终结器
  - 属性或索引器 `set` 访问器

`expression` 必须是语句表达式。 由于表达式的结果被丢弃，该表达式的返回类型可以是任何类型。

以下示例演示了用于 `Person.ToString` 方法的表达式主体定义：

```
C#

public override string ToString() => $"{fname} {lname}".Trim();
```

它是以下方法定义的简写版：

```
C#

public override string ToString()
{
   return $"{fname} {lname}".Trim();
}
```

可以为方法、运算符、只读属性、构造函数、终结器以及属性和索引器访问器创建表达式正文定义。 有关详细信息，请参阅 “Expression-bodied 成员”。

运算符可重载性

不能重载 `=>` 运算符。

C# 语言规范

有关 lambda 运算符的详细信息，请参阅 C# 语言规范中的匿名函数表达式部分。

请参阅

- C# 参考
- C# 运算符和表达式

:: 运算符 - 命名空间别名运算符

项目 • 2023/05/10

使用命名空间别名限定符 `::` 访问已设置别名的命名空间的成员。 只能使用两个标识符之间的 `::` 限定符。 左侧标识符可以是命名空间别名、外部别名或 `global` 别名之一。 例如：

- 使用 using 别名指令创建的命名空间别名：

  C#

  ```
  using forwinforms = System.Drawing;
  using forwpf = System.Windows;

  public class Converters
  {
      public static forwpf::Point Convert(forwinforms::Point point) =>
  new forwpf::Point(point.X, point.Y);
  }
  ```

- 外部别名。

- `global` 别名，该别名是全局命名空间别名。 全局命名空间是包含未在命名空间中声明的命名空间和类型的命名空间。 与 `::` 限定符一起使用时，`global` 别名始终引用全局命名空间，即使存在用户定义的 `global` 命名空间别名也是如此。

  以下示例使用 `global` 别名访问 .NET System 命名空间，该命名空间是全局命名空间的成员。 如果没有 `global` 别名，则将访问用户定义的 `System` 命名空间（该命名空间是 `MyCompany.MyProduct` 命名空间的成员）：

  C#

  ```
  namespace MyCompany.MyProduct.System
  {
      class Program
      {
          static void Main() => global::System.Console.WriteLine("Using
  global alias");
      }

      class Console
      {
          string Suggestion => "Consider renaming this class";
      }
  }
  ```

> **备注**
>
> 仅当 `global` 关键字是 `::` 限定符的左侧标识符时，该关键字才是全局命名空间别名。

此外，还可以使用 . 令牌来访问设置了别名的命名空间的成员。 不过，`.` 令牌还可用于访问类型成员。 `::` 限定符确保其左侧标识符始终引用命名空间别名，即使存在同名的类型或命名空间也是如此。

C# 语言规范

有关详细信息，请参阅 C# 语言规范中的命名空间别名限定符部分。

请参阅

- C# 参考
- C# 运算符和表达式

await 运算符 - 异步等待任务完成

项目 • 2024/07/01

`await` 运算符暂停对其所属的 async 方法的求值，直到其操作数表示的异步操作完成。 异步操作完成后，`await` 运算符将返回操作的结果（如果有）。 当 `await` 运算符应用到表示已完成操作的操作数时，它将立即返回操作的结果，而不会暂停其所属的方法。 `await` 运算符不会阻止计算异步方法的线程。 当 `await` 运算符暂停其所属的异步方法时，控件将返回到方法的调用方。

在下面的示例中，HttpClient.GetByteArrayAsync 方法返回 `Task<byte[]>` 实例，该实例表示在完成时生成字节数组的异步操作。 在操作完成之前，`await` 运算符将暂停 `DownloadDocsMainPageAsync` 方法。 当 `DownloadDocsMainPageAsync` 暂停时，控件将返回到 `Main` 方法，该方法是 `DownloadDocsMainPageAsync` 的调用方。 `Main` 方法将执行，直至它需要 `DownloadDocsMainPageAsync` 方法执行的异步操作的结果。 当 GetByteArrayAsync 获取所有字节时，将计算 `DownloadDocsMainPageAsync` 方法的其余部分。 之后，将计算 `Main` 方法的其余部分。

C#

```
public class AwaitOperator
{
    public static async Task Main()
    {
        Task<int> downloading = DownloadDocsMainPageAsync();
        Console.WriteLine($"{nameof(Main)}: Launched downloading.");

        int bytesLoaded = await downloading;
        Console.WriteLine($"{nameof(Main)}: Downloaded {bytesLoaded}
bytes.");
    }

    private static async Task<int> DownloadDocsMainPageAsync()
    {
        Console.WriteLine($"{nameof(DownloadDocsMainPageAsync)}: About to
start downloading.");

        var client = new HttpClient();
        byte[] content = await
client.GetByteArrayAsync("https://learn.microsoft.com/en-us/");

        Console.WriteLine($"{nameof(DownloadDocsMainPageAsync)}: Finished
downloading.");
        return content.Length;
    }
}
// Output similar to:
// DownloadDocsMainPageAsync: About to start downloading.
```

```
// Main: Launched downloading.
// DownloadDocsMainPageAsync: Finished downloading.
// Main: Downloaded 27700 bytes.
```

必须提供 `await` 表达式的操作数，才能在任务完成时进行通知。 通常，任务完成时（无论成功还是失败）都会调用委托。 C# 语言规范的 await 部分提供了有关如何实现这些通知的详细信息。

前一个示例使用异步 Main 方法。 有关详细信息，请参阅 Main 方法中的 await 运算符部分。

> **备注**
>
> 有关异步编程的介绍，请参阅**使用 async 和 await 的异步编程**。 利用 `async` 和 `await` 的异步编程遵循**基于任务的异步模式**。

只能在通过 async 关键字修改的方法、lambda 表达式或匿名方法中使用 `await` 运算符。 在异步方法中，不能在同步函数的本地主体、lock 语句块内以及不安全的上下文中使用 `await` 运算符。

`await` 运算符的操作数通常是以下其中一个 .NET 类型：Task、Task<TResult>、ValueTask 或 ValueTask<TResult>。 但是，任何可等待表达式都可以是 `await` 运算符的操作数。 有关详细信息，请参阅 C# 语言规范中的可等待表达式部分。

如果表达式 `t` 的类型为 Task<TResult> 或 ValueTask<TResult>，则表达式 `await t` 的类型为 `TResult`。 如果 `t` 的类型为 Task 或 ValueTask，则 `await t` 的类型为 `void`。 在这两种情况下，如果 `t` 引发异常，则 `await t` 将重新引发异常。

异步流和可释放对象

可使用 `await foreach` 语句来使用异步数据流。 有关详细信息，请参阅迭代语句一文的 foreach 语句部分。

可使用 `await using` 语句来处理异步可释放对象，即其类型可实现 IAsyncDisposable 接口的对象。 有关详细信息，请参阅实现 DisposeAsync 方法一文中的使用异步可释放对象部分。

Main 方法中的 await 运算符

作为应用程序入口点的 Main 方法可以返回 `Task` 或 `Task<int>`，使其成为异步的，以便在其主体中使用 `await` 运算符。 在较早的 C# 版本中，为了确保 `Main` 方法等待异步操作完成，可以检索由相应的异步方法返回的 Task<TResult> 实例的 Task<TResult>.Result 属性值。 对于不生成值的异步操作，可以调用 Task.Wait 方法。 有关如何选择语言版本的信息，请参阅 C# 语言版本管理。

C# 语言规范

有关详细信息，请参阅 C# 语言规范中的 Await 表达式部分。

另请参阅

- C# 运算符和表达式
- async
- 任务异步编程模型
- 异步编程
- 演练：使用 async 和 await 访问 Web
- 教程：生成和使用异步流
- .NET 博客：async/await 在 C# 中的工作原理

在 GitHub 上与我们协作

可以在 GitHub 上找到此内容的源，还可以在其中创建和查看问题和拉取请求。 有关详细信息，请参阅参与者指南。

.NET

.NET 反馈

.NET 是一个开放源代码项目。 选择一个链接以提供反馈：

提出文档问题

提供产品反馈

默认值表达式 - 生成默认值

项目 • 2024/04/02

default value 表达式生成类型的默认值。 有两种类型的 default value 表达式：default 运算符调用和 default 文本。

你还可以将 `default` 关键字用作 switch 语句中的默认用例标签。

default 运算符

`default` 运算符的实参必须是类型或类型形参的名称，如以下示例所示：

C#

```
Console.WriteLine(default(int));  // output: 0
Console.WriteLine(default(object) is null);  // output: True

void DisplayDefaultOf<T>()
{
    var val = default(T);
    Console.WriteLine($"Default value of {typeof(T)} is {(val == null ?
"null" : val.ToString())}.");
}

DisplayDefaultOf<int?>();
DisplayDefaultOf<System.Numerics.Complex>();
DisplayDefaultOf<System.Collections.Generic.List<int>>();
// Output:
// Default value of System.Nullable`1[System.Int32] is null.
// Default value of System.Numerics.Complex is (0, 0).
// Default value of System.Collections.Generic.List`1[System.Int32] is null.
```

default 文本

当编译器可以推断表达式类型时，可以使用 `default` 文本生成类型的默认值。 `default` 文本表达式生成与 `default(T)` 表达式（其中，`T` 是推断的类型）相同的值。 可以在以下任一情况下使用 `default` 文本：

- 对变量进行赋值或初始化时。
- 在声明可选方法参数的默认值时。
- 在方法调用中提供参数值时。
- 在 return 语句中或作为表达式主体成员中的表达式时。

下面的示例演示 `default` 文本的用法：

C#

```
T[] InitializeArray<T>(int length, T initialValue = default)
{
    if (length < 0)
    {
        throw new ArgumentOutOfRangeException(nameof(length), "Array length
must be nonnegative.");
    }

    var array = new T[length];
    for (var i = 0; i < length; i++)
    {
        array[i] = initialValue;
    }
    return array;
}

void Display<T>(T[] values) => Console.WriteLine($"[ {string.Join(", ",
values)} ]");

Display(InitializeArray<int>(3));  // output: [ 0, 0, 0 ]
Display(InitializeArray<bool>(4, default));  // output: [ False, False,
False, False ]

System.Numerics.Complex fillValue = default;
Display(InitializeArray(3, fillValue));  // output: [ (0, 0), (0, 0), (0, 0)
]
```

提示

使用 .NET 样式规则 IDE0034 来指定代码库中使用 `default` 文本的首选项。

C# 语言规范

有关详细信息，请参阅 C# 语言规范中的 默认值表达式部分。

另请参阅

- C# 运算符和表达式
- C# 类型的默认值
- .NET 中的泛型

在 GitHub 上与我们协作

可以在 GitHub 上找到此内容的源，还可以在其中创建和查看问题和拉取请求。 有关详细信息，请参阅参与者指南。

.NET 反馈

.NET 是一个开放源代码项目。 选择一个链接以提供反馈：

提出文档问题

提供产品反馈

delegate 运算符

项目 • 2023/11/20

`delegate` 运算符创建一个可以转换为委托类型的匿名方法。 匿名方法可以转换为 System.Action 和 System.Func<TResult> 等类型，用作许多方法的参数。

C#

```
Func<int, int, int> sum = delegate (int a, int b) { return a + b; };
Console.WriteLine(sum(3, 4));  // output: 7
```

ⓘ 备注

lambda 表达式提供了一种更简洁和富有表现力的方式来创建匿名函数。 使用 => **运算符**构造 Lambda 表达式：

C#

```
Func<int, int, int> sum = (a, b) => a + b;
Console.WriteLine(sum(3, 4));  // output: 7
```

有关 lambda 表达式功能的更多信息（例如，如何捕获外部变量），请参阅 **lambda 表达式**。

使用 `delegate` 运算符时，可以省略参数列表。 如果这样做，可以将创建的匿名方法转换为具有任何参数列表的委托类型，如以下示例所示：

C#

```
Action greet = delegate { Console.WriteLine("Hello!"); };
greet();

Action<int, double> introduce = delegate { Console.WriteLine("This is
world!"); };
introduce(42, 2.7);

// Output:
// Hello!
// This is world!
```

这是 lambda 表达式不支持的匿名方法的唯一功能。 在所有其他情况下，lambda 表达式是编写内联代码的首选方法。 可以使用弃元指定该方法未使用的两个或更多个匿名方法输入参数：

C#

```
Func<int, int, int> constant = delegate (int _, int _) { return 42; };
Console.WriteLine(constant(3, 4));  // output: 42
```

为实现向后兼容性，如果只有一个参数名为 `_`，则将 `_` 视为匿名方法中该参数的名称。

可以在匿名方法的声明中使用 `static` 修饰符：

C#

```
Func<int, int, int> sum = static delegate (int a, int b) { return a + b; };
Console.WriteLine(sum(10, 4));  // output: 14
```

静态匿名方法无法从封闭范围捕获局部变量或实例状态。

还可以使用 `delegate` 关键字声明委托类型。

从 C# 11 开始，编译器可以缓存从方法组转换创建的委托对象。 请考虑以下方法：

C#

```
static void StaticFunction() { }
```

将方法组分配给委托时，编译器将缓存委托：

C#

```
Action a = StaticFunction;
```

在 C# 11 之前，需要使用 lambda 表达式来重复使用单个委托对象：

C#

```
Action a = () => StaticFunction();
```

C# 语言规范

有关详细信息，请参阅 C# 语言规范中的 匿名函数表达式部分。

另请参阅

- C# 参考
- C# 运算符和表达式
- => operator

在 GitHub 上与我们协作

可以在 GitHub 上找到此内容的源，还可以在其中创建和查看问题和拉取请求。 有关详细信息，请参阅参与者指南。

.NET feedback

The .NET documentation is open source. Provide feedback here.

提出文档问题

提供产品反馈

is 运算符（C# 参考）

项目 • 2024/04/02

`is` 运算符检查表达式的结果是否与给定的类型相匹配。 有关类型测试 `is` 运算符的信息，请参阅文章类型测试和强制转换运算符的 is 运算符部分。 还可使用 `is` 运算符将表达式与模式相匹配，如下例所示：

```
C#

static bool IsFirstFridayOfOctober(DateTime date) =>
    date is { Month: 10, Day: <=7, DayOfWeek: DayOfWeek.Friday };
```

在前面的示例中，`is` 运算符将表达式与关系模式和带有嵌套常量的属性模式相匹配。

`is` 运算符在以下应用场景中很有用：

- 检查表达式的运行时类型，如下例所示：

  ```
  C#

  int i = 34;
  object iBoxed = i;
  int? jNullable = 42;
  if (iBoxed is int a && jNullable is int b)
  {
      Console.WriteLine(a + b);  // output 76
  }
  ```

  前面的示例演示声明模式的用法。

- 检查是否为 `null`，如下例所示：

  ```
  C#

  if (input is null)
  {
      return;
  }
  ```

  将表达式与 `null` 匹配时，编译器保证不会调用用户重载的 `==` 或 `!=` 运算符。

- 可使用否定模式 执行非 null 检查，如下例所示：

  ```
  C#
  ```

```
if (result is not null)
{
    Console.WriteLine(result.ToString());
}
```

- 从 C# 11 开始，可以使用列表模式来匹配列表或数组的元素。 以下代码检查数组中处于预期位置的整数值：

```C#
int[] empty = [];
int[] one = [1];
int[] odd = [1, 3, 5];
int[] even = [2, 4, 6];
int[] fib = [1, 1, 2, 3, 5];

Console.WriteLine(odd is [1, _, 2, ..]);   // false
Console.WriteLine(fib is [1, _, 2, ..]);   // true
Console.WriteLine(fib is [_, 1, 2, 3, ..]);     // true
Console.WriteLine(fib is [.., 1, 2, 3, _ ]);     // true
Console.WriteLine(even is [2, _, 6]);     // true
Console.WriteLine(even is [2, .., 6]);    // true
Console.WriteLine(odd is [.., 3, 5]); // true
Console.WriteLine(even is [.., 3, 5]); // false
Console.WriteLine(fib is [.., 3, 5]); // true
```

> **备注**
>
> 有关 `is` 运算符支持的模式的完整列表，请参阅**模式**。

C# 语言规范

有关详细信息，请参阅 C# 语言规范的 is 运算符部分以及模式匹配。

另请参阅

- C# 运算符和表达式
- 模式
- 教程：使用模式匹配来构建类型驱动和数据驱动的算法
- 类型测试和强制转换运算符

在 GitHub 上与我们协作

可以在 GitHub 上找到此内容的源，还可以在其中创建和查看问题和拉取请求。 有关详细信息，请参阅参与者指南。

.NET 反馈

.NET 是一个开放源代码项目。 选择一个链接以提供反馈：

提出文档问题

提供产品反馈

nameof 表达式（C# 参考）

项目 • 2024/11/12

`nameof` 表达式可生成变量、类型或成员的名称作为字符串常量。 `nameof` 表达式在编译时进行求值，在运行时无效。 当操作数是类型或命名空间时，生成的名称不是完全限定的。 以下示例显示了 `nameof` 表达式的用法：

C#

```
Console.WriteLine(nameof(System.Collections.Generic));  // output: Generic
Console.WriteLine(nameof(List<int>));  // output: List
Console.WriteLine(nameof(List<int>.Count));  // output: Count
Console.WriteLine(nameof(List<int>.Add));  // output: Add

List<int> numbers = new List<int>() { 1, 2, 3 };
Console.WriteLine(nameof(numbers));  // output: numbers
Console.WriteLine(nameof(numbers.Count));  // output: Count
Console.WriteLine(nameof(numbers.Add));  // output: Add
```

可以使用 `nameof` 表达式使参数检查代码更易于维护：

C#

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

从 C# 11 开始，可以在方法或其参数的属性 中使用具有方法参数的 `nameof` 表达式。 以下代码演示如何对方法、本地函数和 lambda 表达式的参数执行该操作：

C#

```
[ParameterString(nameof(msg))]
public static void Method(string msg)
{
    [ParameterString(nameof(T))]
    void LocalFunction<T>(T param) { }

    var lambdaExpression = ([ParameterString(nameof(aNumber))] int aNumber)
=> aNumber.ToString();
}
```

在使用可为空的分析属性或 CallerArgumentExpression 属性时，带有参数的 `nameof` 表达式很有用。

当操作数是 逐字标识符时， `@` 该字符不是名称的一部分，如以下示例所示：

C#

```
var @new = 5;
Console.WriteLine(nameof(@new));  // output: new
```

C# 语言规范

有关详细信息，请参阅 C# 语言规范 的 Nameof 表达式 部分和 C# 11 - 扩展 nameof 范围功能规范。

另请参阅

- C# 运算符和表达式
- 将 typeof 转换为 nameof（样式规则 IDE0082）

new 运算符 - `new` 运算符创建类型的新实例

项目 • 2023/11/20

`new` 运算符创建类型的新实例。 `new` 关键字还可用作成员声明修饰符或泛型类型约束。

构造函数调用

要创建类型的新实例，通常使用 `new` 运算符调用该类型的某个构造函数：

C#

```
var dict = new Dictionary<string, int>();
dict["first"] = 10;
dict["second"] = 20;
dict["third"] = 30;

Console.WriteLine(string.Join("; ", dict.Select(entry => $"{entry.Key}:
{entry.Value}")));
// Output:
// first: 10; second: 20; third: 30
```

可以使用带有 `new` 运算符的对象或集合初始值设定项实例化和初始化一个语句中的对象，如下例所示：

C#

```
var dict = new Dictionary<string, int>
{
    ["first"] = 10,
    ["second"] = 20,
    ["third"] = 30
};

Console.WriteLine(string.Join("; ", dict.Select(entry => $"{entry.Key}:
{entry.Value}")));
// Output:
// first: 10; second: 20; third: 30
```

目标类型化 `new`

构造调用表达式由目标确定类型。 也就是说，如果已知表达式的目标类型，则可以省略类型名称，如下面的示例所示：

C#

```
List<int> xs = new();
List<int> ys = new(capacity: 10_000);
List<int> zs = new() { Capacity = 20_000 };

Dictionary<int, List<int>> lookup = new()
{
    [1] = new() { 1, 2, 3 },
    [2] = new() { 5, 8, 3 },
    [5] = new() { 1, 0, 4 }
};
```

如前面的示例所示，在由目标确定类型的 `new` 表达式中始终使用括号。

如果 `new` 表达式的目标类型未知（例如使用 var 关键字时），必须指定类型名称。

数组创建

还可以使用 `new` 运算符创建数组实例，如下例所示：

C#

```
int[] numbers = [10, 20, 30];
Console.WriteLine(string.Join(", ", numbers));
// Output:
// 10, 20, 30
```

使用数组初始化语法创建数组实例，并在一个语句中使用元素填充该实例。 以下示例显示可以执行该操作的各种方法：

C#

```
int[] a = [10, 20, 30];
int[] b = [10, 20, 30];
int[] c = [10, 20, 30];
Console.WriteLine(c.GetType());  // output: System.Int32[]
```

有关数组的详细信息，请参阅数组。

匿名类型的实例化

要创建匿名类型的实例，请使用 `new` 运算符和对象初始值设定项语法：

C#

```
var example = new { Greeting = "Hello", Name = "World" };
Console.WriteLine($"{example.Greeting}, {example.Name}!");
// Output:
// Hello, World!
```

类型实例的析构

无需销毁此前创建的类型实例。 引用和值类型的实例将自动销毁。 包含值类型的上下文销毁后，值类型的实例随之销毁。 在引用类型的最后一次引用被删除后，垃圾回收器会在某个非指定的时间销毁其实例。

对于包含非托管资源的类型实例（例如，文件句柄），建议采用确定性的清理来确保尽快释放其包含的资源。 有关详细信息，请参阅 System.IDisposable API 参考和 using 语句一文。

运算符可重载性

用户定义的类型可以重载 `new` 运算符。

C# 语言规范

有关详细信息，请参阅 C# 语言规范的 new 运算符部分。

有关条件由目标确定类型的 `new` 表达式的详细信息，请参阅功能建议说明。

另请参阅

- C# 参考
- C# 运算符和表达式
- 对象和集合初始值设定项

在 GitHub 上与我们协作

可以在 GitHub 上找到此内容的源，还可以在其中创建和查看问题和拉取请求。 有关详细信息，请参阅参与者指南。

.NET feedback

The .NET documentation is open source. Provide feedback here.

提出文档问题

提供产品反馈

sizeof 运算符 - 确定给定类型的内存需求

项目 • 2024/04/02

`sizeof` 运算符返回给定类型的变量所占用的字节数。 `sizeof` 运算符的参数必须是一个非托管类型的名称，或是一个限定为非托管类型的类型参数。

`sizeof` 运算符需要不安全上下文。 但下表中的表达式在编译时被计算为相应的常数值，并不需要“不安全”的上下文：

展开表

Expression	常量值
`sizeof(sbyte)`	1
`sizeof(byte)`	1
`sizeof(short)`	2
`sizeof(ushort)`	2
`sizeof(int)`	4
`sizeof(uint)`	4
`sizeof(long)`	8
`sizeof(ulong)`	8
`sizeof(char)`	2
`sizeof(float)`	4
`sizeof(double)`	8
`sizeof(decimal)`	16
`sizeof(bool)`	1

下列情况也不需要使用不安全的上下文： `sizeof` 运算符的操作数是枚举类型的名称。

下面的示例演示 `sizeof` 运算符的用法：

C#

```
public struct Point
{
    public Point(byte tag, double x, double y) => (Tag, X, Y) = (tag, x, y);
```

```
    public byte Tag { get; }
    public double X { get; }
    public double Y { get; }
}

public class SizeOfOperator
{
    public static void Main()
    {
        Console.WriteLine(sizeof(byte));  // output: 1
        Console.WriteLine(sizeof(double));  // output: 8

        DisplaySizeOf<Point>();  // output: Size of Point is 24
        DisplaySizeOf<decimal>();  // output: Size of System.Decimal is 16

        unsafe
        {
            Console.WriteLine(sizeof(Point*));  // output: 8
        }
    }

    static unsafe void DisplaySizeOf<T>() where T : unmanaged
    {
        Console.WriteLine($"Size of {typeof(T)} is {sizeof(T)}");
    }
}
```

`sizeof` 运算符返回公共语言运行时将在托管内存中分配的字节数。 对于结构类型，该值包括了填充（如有），如前例所示。 `sizeof` 运算符的结果可能异于 Marshal.SizeOf 方法的结果，该方法返回某个类型在*非托管*内存中的大小。

C# 语言规范

有关详细信息，请参阅 C# 语言规范的 sizeof 运算符部分。

另请参阅

- C# 运算符和表达式
- 指针相关的运算符
- 指针类型
- 内存和跨度相关类型
- .NET 中的泛型

在 GitHub 上与我们协作

可以在 GitHub 上找到此内容的源，还可以在其中创建和查看问题和拉取请求。 有关详细信息，请参阅参与者指南。

.NET 反馈

.NET 是一个开放源代码项目。 选择一个链接以提供反馈：

提出文档问题

提供产品反馈

stackalloc 表达式（C# 参考）

项目 • 2024/12/21

`stackalloc` 表达式在堆栈上分配内存块。 当该方法返回时，将自动丢弃在方法执行期间创建的已分配堆栈内存块。 不能显式释放使用 `stackalloc` 分配的内存。 堆栈中分配的内存块不受垃圾回收的影响，也不必通过 fixed 语句固定。

可以将 `stackalloc` 表达式的结果分配给以下任一类型的变量：

- System.Span<T> 或 System.ReadOnlySpan<T>，如以下示例所示：

  C#

  ```
  int length = 3;
  Span<int> numbers = stackalloc int[length];
  for (var i = 0; i < length; i++)
  {
      numbers[i] = i;
  }
  ```

  将堆栈中分配的内存块分配给 或 Span<T> 变量时，不必使用 ReadOnlySpan<T> 上下文。

  使用这些类型时，可以在`stackalloc`或赋值表达式中使用 表达式，如以下示例所示：

  C#

  ```
  int length = 1000;
  Span<byte> buffer = length <= 1024 ? stackalloc byte[length] : new
  byte[length];
  ```

  只要允许使用 `stackalloc` 或 Span<T> 变量，就可以在其他表达式中使用 ReadOnlySpan<T> 表达式或集合表达式，如下例所示：

  C#

  ```
  Span<int> numbers = stackalloc[] { 1, 2, 3, 4, 5, 6 };
  var ind = numbers.IndexOfAny(stackalloc[] { 2, 4, 6, 8 });
  Console.WriteLine(ind);  // output: 1

  Span<int> numbers2 = [1, 2, 3, 4, 5, 6];
  var ind2 = numbers2.IndexOfAny([2, 4, 6, 8]);
  Console.WriteLine(ind2);  // output: 1
  ```

> **① 备注**
>
> 建议尽可能使用 Span<T> 或 ReadOnlySpan<T> 类型来处理堆栈中分配的内存。

- 指针类型，如以下示例所示：

```
C#

unsafe
{
    int length = 3;
    int* numbers = stackalloc int[length];
    for (var i = 0; i < length; i++)
    {
        numbers[i] = i;
    }
}
```

  如前面的示例所示，在使用指针类型时必须使用 `unsafe` 上下文。

  对于指针类型，只能在局部变量声明中使用 `stackalloc` 表达式来初始化变量。

堆栈上可用的内存量存在限制。 如果在堆栈上分配过多的内存，会引发 StackOverflowException。 为避免这种情况，请遵循以下规则：

- 限制使用 `stackalloc` 分配的内存量。 例如，如果预期的缓冲区大小低于特定限制，则在堆栈上分配内存；否则，使用所需长度的数组，如以下代码所示：

```
C#

const int MaxStackLimit = 1024;
Span<byte> buffer = inputLength <= MaxStackLimit ? stackalloc
byte[MaxStackLimit] : new byte[inputLength];
```

> **① 备注**
>
> 由于堆栈上可用的内存量取决于执行代码的环境，因此在定义实际限值时请持保守态度。

- 避免在循环内使用 `stackalloc`。 在循环外分配内存块，然后在循环内重用它。

新分配的内存的内容未定义。 应使用 `stackalloc` 初始值设定项初始化它，或者使用方法（如 Span<T>.Clear 使用前）。

> **ⓘ 重要**
>
> 不初始化由 `stackalloc` 该运算符分配的 `new` 内存是一个重要区别。 使用运算符分配的 `new` 内存将初始化为 0 位模式。

可以使用数组初始值设定项语法来定义新分配的内存的内容。 下面的示例演示执行此操作的各种方法：

```
C#
```

```
Span<int> first = stackalloc int[3] { 1, 2, 3 };
Span<int> second = stackalloc int[] { 1, 2, 3 };
ReadOnlySpan<int> third = stackalloc[] { 1, 2, 3 };

// Using collection expressions:
Span<int> fourth = [1, 2, 3];
ReadOnlySpan<int> fifth = [1, 2, 3];
```

在表达式 `stackalloc T[E]` 中，`T` 必须是非托管类型，并且 `E` 的计算结果必须为非负 int 值。 使用集合表达式语法来初始化范围时，如果编译器没有违反 ref 安全性，编译器可能会对范围使用堆栈分配的存储。

安全性

使用 `stackalloc` 会自动启用公共语言运行时 (CLR) 中的缓冲区溢出检测功能。 如果检测到缓冲区溢出，则将尽快终止进程，以便将执行恶意代码的可能性降到最低。

C# 语言规范

有关详细信息，请参阅 C# 语言规范中的 堆栈分配 部分。

另请参阅

- C# 运算符和表达式
- 指针相关的运算符
- 指针类型
- 内存和跨度相关类型
- stackalloc 注意事项

switch 表达式 - 使用 `switch` 关键字的模式匹配表达式

项目 • 2023/05/09

可以使用 `switch` 表达式，根据与输入表达式匹配的模式，对候选表达式列表中的单个表达式进行求值。 有关在语句上下文中支持 `switch` 类语义的 `switch` 语句的信息，请参阅选择语句一文的 switch 语句部分。

下面的示例演示了一个 `switch` 表达式，该表达式将在线地图中表示视觉方向的 enum 中的值转换为相应的基本方位：

C#

```
public static class SwitchExample
{
    public enum Direction
    {
        Up,
        Down,
        Right,
        Left
    }

    public enum Orientation
    {
        North,
        South,
        East,
        West
    }

    public static Orientation ToOrientation(Direction direction) =>
direction switch
    {
        Direction.Up    => Orientation.North,
        Direction.Right => Orientation.East,
        Direction.Down  => Orientation.South,
        Direction.Left  => Orientation.West,
        _ => throw new ArgumentOutOfRangeException(nameof(direction), $"Not
expected direction value: {direction}"),
    };

    public static void Main()
    {
        var direction = Direction.Right;
        Console.WriteLine($"Map view direction is {direction}");
        Console.WriteLine($"Cardinal orientation is
{ToOrientation(direction)}");
        // Output:
```

```
        // Map view direction is Right
        // Cardinal orientation is East
    }
}
```

上述示例展示了 `switch` 表达式的基本元素：

- 后跟 `switch` 关键字的表达式。 在上述示例中，这是 `direction` 方法参数。
- `switch` expression arm，用逗号分隔。 每个 `switch` expression arm 都包含一个模式、一个可选的 case guard、`=>` 标记和一个表达式 。

在上述示例中，`switch` 表达式使用以下模式：

- 常数模式：用于处理 `Direction` 枚举的定义值。
- 弃元模式：用于处理没有相应的 `Direction` 枚举成员的任何整数值（例如 `(Direction)10`）。 这会使 `switch` 表达式详尽。

> **ⓘ 重要**
>
> 有关 `switch` 表达式支持的模式的信息及更多示例，请参阅**模式**。

`switch` 表达式的结果是第一个 `switch` expression arm 的表达式的值，该 switch expression arm 的模式与范围表达式匹配，并且它的 case guard（如果存在）求值为 `true`。 `switch` expression arm 按文本顺序求值。

如果无法选择较低的 `switch` expression arm，编译器会发出错误，因为较高的 `switch` expression arm 匹配其所有值。

Case guard

模式或许表现力不够，无法指定用于计算 arm 的表达式的条件。 在这种情况下，可以使用 case guard。 case guard 是一个附加条件，必须与匹配模式同时满足。 case guard 必须是布尔表达式。 可以在模式后面的 `when` 关键字之后指定一个 case guard，如以下示例所示：

C#

```
public readonly struct Point
{
    public Point(int x, int y) => (X, Y) = (x, y);

    public int X { get; }
    public int Y { get; }
```

```
}

static Point Transform(Point point) => point switch
{
    { X: 0, Y: 0 }                    => new Point(0, 0),
    { X: var x, Y: var y } when x < y => new Point(x + y, y),
    { X: var x, Y: var y } when x > y => new Point(x - y, y),
    { X: var x, Y: var y }            => new Point(2 * x, 2 * y),
};
```

上述示例使用带有嵌套 var 模式的属性模式。

非详尽的 switch 表达式

如果 `switch` 表达式的模式均未捕获输入值，则运行时将引发异常。 在 .NET Core 3.0 及更高版本中，异常是 System.Runtime.CompilerServices.SwitchExpressionException。 在 .NET Framework 中，异常是 InvalidOperationException。 大大多数情况下，如果 `switch` 表达式未处理所有可能的输入值，则编译器会生成警告。 如果未处理所有可能的输入，列表模式不会生成警告。

> **提示**
>
> 为了保证 `switch` 表达式处理所有可能的输入值，请为 `switch` expression arm 提供**弃元模式**。

C# 语言规范

有关详细信息，请参阅功能建议说明的 switch 表达式部分。

另请参阅

- 使用 switch 表达式（样式规则 IDE0066）
- 将缺失的事例添加到 switch 表达式（样式规则 IDE0072）
- C# 参考
- C# 运算符和表达式
- 模式
- 教程：使用模式匹配来构建类型驱动和数据驱动的算法
- switch 语句

true 和 false 运算符 - 将对象视为布尔值

项目 • 2024/04/01

`true` 运算符返回 bool 值 `true` 来指示其操作数一定为 true，而 `false` 运算符则返回 `bool` 值 `true` 来指示其操作数一定为 false。
请注意，同时实现 `true` 和 `false` 运算符的类型必须遵循以下语义：

- “此对象是否为 true？”解析为运算符 `true`。 如果对象为 `true`，运算符 `true` 将返回 `true`。 应答为“是，此对象为 true”。
- “此对象是否为 false？”解析为运算符 `false`。 如果对象为 `false`，运算符 `false` 将返回 `true`。 应答为“是，此对象为 false”

无法确保 `true` 和 `false` 运算符互补。 也就是说，`true` 和 `false` 运算符可能同时针对同一个操作数返回 `bool` 值 `false`。 如果某类型定义这两个运算符之一，则其还必须定义另一个运算符。

> **提示**
>
> 如需支持三值逻辑（例如，在使用支持三值布尔类型的数据库时），请使用 `bool?` 类型。 C# 提供 `&` 和 `|` 运算符，它们通过 `bool?` 操作数支持三值逻辑。 有关详细信息，请参阅**布尔逻辑运算符**一文的**可以为 null 的布尔逻辑运算符**部分。

布尔表达式

包含已定义 `true` 运算符的类型可以是 if、do、while 和 for 语句以及条件运算符 ?: 中控制条件表达式的结果的类型。 有关详细信息，请参阅 C# 语言规范中的 Boolean 表达式部分。

用户定义的条件逻辑运算符

如果包含已定义 `true` 和 `false` 运算符的类型以某种方式重载逻辑 OR 运算符 `|` 或逻辑 AND 运算符 `&`，可以对相应类型的操作数分别执行条件逻辑 OR 运算符 `||` 或条件逻辑 AND 运算符 `&&` 运算。 有关详细信息，请参阅 C# 语言规范的用户定义条件逻辑运算符部分。

示例

下面的示例演示了定义 `true` 和 `false` 运算符的类型。 此外，该类型还重载了逻辑 AND 运算符 `&`，因此，也可以对相应类型的操作数计算运算符 `&&`。

C#

```
public struct LaunchStatus
{
    public static readonly LaunchStatus Green = new LaunchStatus(0);
    public static readonly LaunchStatus Yellow = new LaunchStatus(1);
    public static readonly LaunchStatus Red = new LaunchStatus(2);

    private int status;

    private LaunchStatus(int status)
    {
        this.status = status;
    }

    public static bool operator true(LaunchStatus x) => x == Green || x ==
Yellow;
    public static bool operator false(LaunchStatus x) => x == Red;

    public static LaunchStatus operator &(LaunchStatus x, LaunchStatus y)
    {
        if (x == Red || y == Red || (x == Yellow && y == Yellow))
        {
            return Red;
        }

        if (x == Yellow || y == Yellow)
        {
            return Yellow;
        }

        return Green;
    }

    public static bool operator ==(LaunchStatus x, LaunchStatus y) =>
x.status == y.status;
    public static bool operator !=(LaunchStatus x, LaunchStatus y) => !(x ==
y);

    public override bool Equals(object obj) => obj is LaunchStatus other &&
this == other;
    public override int GetHashCode() => status;
}

public class LaunchStatusTest
{
    public static void Main()
    {
        LaunchStatus okToLaunch = GetFuelLaunchStatus() &&
GetNavigationLaunchStatus();
```

```
        Console.WriteLine(okToLaunch ? "Ready to go!" : "Wait!");
    }

    static LaunchStatus GetFuelLaunchStatus()
    {
        Console.WriteLine("Getting fuel launch status...");
        return LaunchStatus.Red;
    }

    static LaunchStatus GetNavigationLaunchStatus()
    {
        Console.WriteLine("Getting navigation launch status...");
        return LaunchStatus.Yellow;
    }
}
```

请注意 `&&` 运算符的短路行为。 当 `GetFuelLaunchStatus` 方法返回 `LaunchStatus.Red` 时，不会进行计算的 `&&` 运算符的右侧操作数。 这是因为 `LaunchStatus.Red` 一定为 false。 然后，逻辑 AND 运算符的结果不依赖右侧操作数的值。 示例的输出如下所示：

控制台

```
Getting fuel launch status...
Wait!
```

另请参阅

- C# 运算符和表达式

在 GitHub 上与我们协作

可以在 GitHub 上找到此内容的源，还可以在其中创建和查看问题和拉取请求。 有关详细信息，请参阅参与者指南。

.NET

.NET 反馈

.NET 是一个开放源代码项目。 选择一个链接以提供反馈：

提出文档问题

提供产品反馈

with 表达式 - 非破坏性突变创建具有修改属性的新对象

项目 • 2025/01/06

`with` 表达式使用修改的特定属性和字段生成其操作数的副本。 使用 对象初始化器 语法来指定要修改的成员以及它们的新值：

C#

```
using System;

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

    public static void Main()
    {
        var p1 = new NamedPoint("A", 0, 0);
        Console.WriteLine($"{nameof(p1)}: {p1}");  // output: p1: NamedPoint
{ Name = A, X = 0, Y = 0 }

        var p2 = p1 with { Name = "B", X = 5 };
        Console.WriteLine($"{nameof(p2)}: {p2}");  // output: p2: NamedPoint
{ Name = B, X = 5, Y = 0 }

        var p3 = p1 with
            {
                Name = "C",
                Y = 4
            };
        Console.WriteLine($"{nameof(p3)}: {p3}");  // output: p3: NamedPoint
{ Name = C, X = 0, Y = 4 }

        Console.WriteLine($"{nameof(p1)}: {p1}");  // output: p1: NamedPoint
{ Name = A, X = 0, Y = 0 }

        var apples = new { Item = "Apples", Price = 1.19m };
        Console.WriteLine($"Original: {apples}");  // output: Original: {
Item = Apples, Price = 1.19 }
        var saleApples = apples with { Price = 0.79m };
        Console.WriteLine($"Sale: {saleApples}");  // output: Sale: { Item =
Apples, Price = 0.79 }
    }
}
```

`with` 表达式的左侧操作数可以是一个记录类型。 `with` 表达式的左侧操作数也可以为结构类型或匿名类型。

`with` 表达式的结果与表达式的操作数具有相同的运行时类型，如以下示例所示：

C#

```
using System;

public class InheritanceExample
{
    public record Point(int X, int Y);
    public record NamedPoint(string Name, int X, int Y) : Point(X, Y);

    public static void Main()
    {
        Point p1 = new NamedPoint("A", 0, 0);
        Point p2 = p1 with { X = 5, Y = 3 };
        Console.WriteLine(p2 is NamedPoint);  // output: True
        Console.WriteLine(p2);  // output: NamedPoint { X = 5, Y = 3, Name =
A }
    }
}
```

对于引用类型成员，在复制操作数时仅复制对成员实例的引用。 复制和原始操作数都有权访问同一引用类型实例。 以下示例演示了该行为：

C#

```
using System;
using System.Collections.Generic;

public class ExampleWithReferenceType
{
    public record TaggedNumber(int Number, List<string> Tags)
    {
        public string PrintTags() => string.Join(", ", Tags);
    }

    public static void Main()
    {
        var original = new TaggedNumber(1, new List<string> { "A", "B" });

        var copy = original with { Number = 2 };
        Console.WriteLine($"Tags of {nameof(copy)}: {copy.PrintTags()}");
        // output: Tags of copy: A, B

        original.Tags.Add("C");
        Console.WriteLine($"Tags of {nameof(copy)}: {copy.PrintTags()}");
        // output: Tags of copy: A, B, C
    }
}
```

自定义复制语义

任何记录类类型都具有复制构造函数。 复制构造函数是一个包含记录类型的单个参数的构造函数。 它将参数的状态复制到新的记录实例。 在计算 `with` 表达式时，将调用复制构造函数，以基于原始记录实例化新记录实例。 之后，新实例会根据指定的修改进行更新。 默认情况下，复制构造函数是隐式的，即编译器生成的。 如果需要自定义记录复制语义，请显式声明具有所需行为的复制构造函数。 以下示例使用显式复制构造函数更新前面的示例。 新的复制方式是在复制记录时，复制列表项而不是列表引用。

C#

```
using System;
using System.Collections.Generic;

public class UserDefinedCopyConstructorExample
{
    public record TaggedNumber(int Number, List<string> Tags)
    {
        protected TaggedNumber(TaggedNumber original)
        {
            Number = original.Number;
            Tags = new List<string>(original.Tags);
        }

        public string PrintTags() => string.Join(", ", Tags);
    }

    public static void Main()
    {
        var original = new TaggedNumber(1, new List<string> { "A", "B" });

        var copy = original with { Number = 2 };
        Console.WriteLine($"Tags of {nameof(copy)}: {copy.PrintTags()}");
        // output: Tags of copy: A, B

        original.Tags.Add("C");
        Console.WriteLine($"Tags of {nameof(copy)}: {copy.PrintTags()}");
        // output: Tags of copy: A, B
    }
}
```

不能自定义结构类型的复制语义。

C# 语言规范

有关详细信息，请参阅 记录功能建议说明的以下部分：

- with 表达式

- 复制和克隆成员

另请参阅

- C# 运算符和表达式
- 记录
- 结构类型

运算符重载 - 预定义的一元运算符、算术运算符、相等运算符和比较运算符

项目 • 2024/04/02

用户定义的类型可重载预定义的 C# 运算符。 也就是说，当一个或两个操作数都是某类型时，此类型可提供操作的自定义实现。 可重载运算符部分介绍了哪些 C# 运算符可重载。

使用 `operator` 关键字来声明运算符。 运算符声明必须符合以下规则：

- 同时包含 `public` 和 `static` 修饰符。
- 一元运算符有一个输入参数。 二元运算符有两个输入参数。 在每种情况下，都至少有一个参数必须具有类型 `T` 或 `T?`，其中 `T` 是包含运算符声明的类型。

下面的示例定义了一个表示有理数的简单结构。 该结构会重载一些算术运算符：

C#

```
public readonly struct Fraction
{
    private readonly int num;
    private readonly int den;

    public Fraction(int numerator, int denominator)
    {
        if (denominator == 0)
        {
            throw new ArgumentException("Denominator cannot be zero.",
nameof(denominator));
        }
        num = numerator;
        den = denominator;
    }

    public static Fraction operator +(Fraction a) => a;
    public static Fraction operator -(Fraction a) => new Fraction(-a.num,
a.den);

    public static Fraction operator +(Fraction a, Fraction b)
        => new Fraction(a.num * b.den + b.num * a.den, a.den * b.den);

    public static Fraction operator -(Fraction a, Fraction b)
        => a + (-b);

    public static Fraction operator *(Fraction a, Fraction b)
        => new Fraction(a.num * b.num, a.den * b.den);

    public static Fraction operator /(Fraction a, Fraction b)
```

```
    {
        if (b.num == 0)
        {
            throw new DivideByZeroException();
        }
        return new Fraction(a.num * b.den, a.den * b.num);
    }

    public override string ToString() => $"{num} / {den}";
}

public static class OperatorOverloading
{
    public static void Main()
    {
        var a = new Fraction(5, 4);
        var b = new Fraction(1, 2);
        Console.WriteLine(-a);   // output: -5 / 4
        Console.WriteLine(a + b);  // output: 14 / 8
        Console.WriteLine(a - b);  // output: 6 / 8
        Console.WriteLine(a * b);  // output: 5 / 8
        Console.WriteLine(a / b);  // output: 10 / 4
    }
}
```

可以通过定义从 int 到 Fraction 的隐式转换来扩展前面的示例。 然后，重载运算符将支持这两种类型的参数。 也就是说，可以将一个整数添加到一个分数中，得到一个分数结果。

还可以使用 `operator` 关键字来定义自定义类型转换。 有关详细信息，请参阅用户定义转换运算符。

可重载运算符

下表显示了可重载的运算符：

展开表

运算符	说明
+x, -x, !x, ~x, ++, --, true, false	`true` 和 `false` 运算符必须一起重载。
x + y, x - y, x * y, x / y, x % y, x & y, x | y, x ^ y, x << y, x >> y, x >>> y	
x == y, x != y, x < y, x > y, x <= y, x >= y	必须按如下方式成对重载：`==` 和 `!=`、`<` 和 `>`、`<=` 和 `>=`。

不可重载运算符

下表显示了无法重载的运算符：

展开表

运算符	备选方法
x && y，x || y	重载 true 和 false 运算符以及 & 或 | 运算符。 有关详细信息，请参阅用户定义的条件逻辑运算符。
a[i]，a?[i]	定义索引器。
(T)x	定义可由强制转换表达式执行的自定义类型转换。 有关详细信息，请参阅用户定义转换运算符。
+=, -=, *=, /=, %=, &=, |=, ^=, <<=, >>=, >>>=	重载相应的二元运算符。 例如，重载二进制 `+` 运算符时，会隐式重载 `+=`。
^x、x = y、x.y、x?.y、c ? t : f、x ?? y、??= y x..y, x->y, =>, f(x), as, await, checked, unchecked, default, delegate, is, nameof, new, sizeof, stackalloc, switch, typeof, with	无。

C# 语言规范

有关更多信息，请参阅 C# 语言规范的以下部分：

- 运算符重载
- 运算符

另请参阅

- C# 运算符和表达式
- 用户定义转换运算符
- 设计准则 - 运算符重载
- 设计准则 - 相等运算符
- 重载运算符为何在 C# 中始终是静态的?

在 GitHub 上与我们协作

可以在 GitHub 上找到此内容的源，还可以在其中创建和查看问题和拉取请求。 有关详细信息，请参阅参与者指南。

.NET 反馈

.NET 是一个开放源代码项目。 选择一个链接以提供反馈：

提出文档问题

提供产品反馈

声明语句

项目 • 2024/03/30

声明语句声明新的局部变量、局部常量或 reference 局部变量。 若要声明局部变量，请指定其类型并提供其名称。 可以在一个语句中声明多个相同类型的变量，如以下示例所示：

C#

```
string greeting;
int a, b, c;
List<double> xs;
```

在声明语句中，还可以使用变量的初始值对该变量进行初始化：

C#

```
string greeting = "Hello";
int a = 3, b = 2, c = a + b;
List<double> xs = new();
```

前面的示例显式指定了变量的类型。 还可以让编译器从其初始化表达式推断出变量的类型。 为此，请使用 `var` 关键字而不是类型的名称。 有关详细信息，请参阅隐式类型的局部变量部分。

若要声明局部常量，请使用 const 关键字，如以下示例所示：

C#

```
const string Greeting = "Hello";
const double MinLimit = -10.0, MaxLimit = -MinLimit;
```

声明局部常量时，还必须对其进行初始化。

有关 reference 局部变量的信息，请参阅 Reference 变量部分。

隐式类型的局部变量

声明局部变量时，可以让编译器从初始化表达式推断出变量的类型。 为此，请使用 `var` 关键字而不是类型的名称：

C#

```
var greeting = "Hello";
Console.WriteLine(greeting.GetType());  // output: System.String

var a = 32;
Console.WriteLine(a.GetType());  // output: System.Int32

var xs = new List<double>();
Console.WriteLine(xs.GetType());  // output:
System.Collections.Generic.List`1[System.Double]
```

如前面的示例所示，隐式类型的局部变量已强类型化。

> ⓘ **备注**
>
> 在已启用的**可为 null 的感知上下文**中使用 `var`，且初始化表达式的类型为引用类型时，即使初始化表达式的类型不可为 null，编译器也始终推断出可为 null 的引用类型。

`var` 的常见用途是用于构造函数调用表达式。 使用 `var` 则不能在变量声明和对象实例化中重复类型名称，如下面的示例所示：

C#

```
var xs = new List<int>();
```

可以使用由目标确定类型的 new 表达式作为替代方法：

C#

```
List<int> xs = new();
List<int>? ys = new();
```

使用匿名类型时，必须使用隐式类型的局部变量。 以下示例显示了一个查询表达式，该表达式使用匿名类型保存客户的姓名和电话号码：

C#

```
var fromPhoenix = from cust in customers
                  where cust.City == "Phoenix"
                  select new { cust.Name, cust.Phone };

foreach (var customer in fromPhoenix)
{
```

```
    Console.WriteLine($"Name={customer.Name}, Phone={customer.Phone}");
}
```

在前面的示例中，无法显式指定 `fromPhoenix` 变量的类型。 类型为 IEnumerable<T>，但在本例中 `T` 为匿名类型，无法提供其名称。 这就是需要使用 `var` 的原因。 出于同一原因，在 `foreach` 语句中声明 `customer` 迭代变量时必须使用 `var`。

有关隐式类型的局部变量的详细信息，请参阅隐式类型的局部变量。

在模式匹配中，在 var 模式中使用 `var` 关键字。

Reference 变量

声明局部变量并在变量类型之前添加 `ref` 关键字时，声明 reference 变量，或 `ref` 局部变量：

C#

```
ref int aliasOfvariable = ref variable;
```

reference 变量是引用另一个变量（称为引用）的变量。 也就是说，reference 变量是其引用的别名。 向 reference 变量赋值时，该值将分配给引用。 读取 reference 变量的值时，将返回引用的值。 以下示例演示了该行为：

C#

```
int a = 1;
ref int aliasOfa = ref a;
Console.WriteLine($"(a, aliasOfa) is ({a}, {aliasOfa})");  // output: (a,
aliasOfa) is (1, 1)

a = 2;
Console.WriteLine($"(a, aliasOfa) is ({a}, {aliasOfa})");  // output: (a,
aliasOfa) is (2, 2)

aliasOfa = 3;
Console.WriteLine($"(a, aliasOfa) is ({a}, {aliasOfa})");  // output: (a,
aliasOfa) is (3, 3)
```

使用 ref 赋值运算符 `= ref` 更改 reference 变量的引用，如以下示例所示：

C#

```
void Display(int[] s) => Console.WriteLine(string.Join(" ", s));

int[] xs = [0, 0, 0];
```

```
Display(xs);

ref int element = ref xs[0];
element = 1;
Display(xs);

element = ref xs[^1];
element = 3;
Display(xs);
// Output:
// 0 0 0
// 1 0 0
// 1 0 3
```

在前面的示例中，`element` reference 变量初始化为第一个数组元素的别名。 然后，`ref` 将被重新分配，以引用最后一个数组元素。

可以定义 `ref readonly` 局部变量。 不能为 `ref readonly` 变量赋值。 但是，可以 `ref` 重新分配这样的 reference 变量，如以下示例所示：

C#

```
int[] xs = [1, 2, 3];

ref readonly int element = ref xs[0];
// element = 100;  error CS0131: The left-hand side of an assignment must be
a variable, property or indexer
Console.WriteLine(element);  // output: 1

element = ref xs[^1];
Console.WriteLine(element);  // output: 3
```

可以将引用返回分配给 reference 变量，如以下示例所示：

C#

```
using System;

public class NumberStore
{
    private readonly int[] numbers = [1, 30, 7, 1557, 381, 63, 1027, 2550,
511, 1023];

    public ref int GetReferenceToMax()
    {
        ref int max = ref numbers[0];
        for (int i = 1; i < numbers.Length; i++)
        {
            if (numbers[i] > max)
            {
                max = ref numbers[i];
```

```
            }
        }
        return ref max;
    }

    public override string ToString() => string.Join(" ", numbers);
}

public static class ReferenceReturnExample
{
    public static void Run()
    {
        var store = new NumberStore();
        Console.WriteLine($"Original sequence: {store.ToString()}");

        ref int max = ref store.GetReferenceToMax();
        max = 0;
        Console.WriteLine($"Updated sequence:  {store.ToString()}");
        // Output:
        // Original sequence: 1 30 7 1557 381 63 1027 2550 511 1023
        // Updated sequence:  1 30 7 1557 381 63 1027 0 511 1023
    }
}
```

在前面的示例中，`GetReferenceToMax` 方法指的是 returns-by-ref 方法。 它不返回最大值本身，而是引用返回，该引用返回是包含最大值的数组元素的别名。 `Run` 方法将引用返回分配给 `max` reference 变量。 然后，通过分配给 `max`，它会更新 `store` 实例的内部存储。 还可以定义 `ref readonly` 方法。 `ref readonly` 方法的调用方无法为其引用返回赋值。

`foreach` 语句的迭代变量可以是引用变量。 有关详细信息，请参阅迭代语句一文的 foreach 语句部分。

在性能关键型方案中，使用 reference 变量和返回可能会避免成本高昂的复制操作，从而提高性能。

编译器确保 reference 变量在时间上不超过其引用，并在其整个生存期内保持有效。 有关详细信息，请参阅 C# 语言规范中的 Ref safe 上下文部分。

有关 `ref` 字段的信息，请参阅 ref 结构类型一文的 ref 字段部分。

scoped ref

上下文关键字 `scoped` 限制值的生存期。 `scoped` 修饰符将 ref-safe-to-escape 或 safe-to-escape 生存期分别限制为当前方法。 实际上，添加 `scoped` 修饰符可确保代码不会延长变量的生存期。

可以将 `scoped` 应用于参数或局部变量。 当类型为 ref struct 时，`scoped` 修饰符可以应用于参数和局部变量。 否则，`scoped` 修饰符只能应用于局部 reference 变量。 这包括使用 `ref` 修饰符声明的局部变量以及使用 `in`、`ref` 或 `out` 修饰符声明的参数。

当类型为 `ref struct` 时，使用 `struct`、`out` 参数和 `ref` 参数声明的方法将 `scoped` 修饰符隐式添加到 `this`。

C# 语言规范

有关更多信息，请参阅 C# 语言规范的以下部分：

- 声明语句
- Reference 变量和返回

有关 `scoped` 修饰符的详细信息，请参阅低级别 struct 改进建议说明。

另请参阅

- 对象和集合初始值设定项
- ref 关键字
- 使用新的 C# 功能减少内存分配
- “var”首选项（样式规则 IDE0007 和 IDE0008）

异常处理语句 - `throw`、`try-catch`、`try-finally` 和 `try-catch-finally`

项目 • 2023/06/05

使用 `throw` 和 `try` 语句来处理异常。 使用 throw 语句引发异常。 使用 try 语句捕获和处理在执行代码块期间可能发生的异常。

`throw` 语句

`throw` 语句引发异常：

C#

```
if (shapeAmount <= 0)
{
    throw new ArgumentOutOfRangeException(nameof(shapeAmount), "Amount of
shapes must be positive.");
}
```

在 `throw e;` 语句中，表达式 `e` 的结果必须隐式转换为 System.Exception。

可以使用内置异常类，例如 ArgumentOutOfRangeException 或 InvalidOperationException。 .NET 还提供了在某些情况下引发异常的帮助程序方法：ArgumentNullException.ThrowIfNull 和 ArgumentException.ThrowIfNullOrEmpty。 还可以定义自己的派生自 System.Exception 的异常类。 有关详细信息，请参阅创建和引发异常。

在 catch 块内，可以使用 `throw;` 语句重新引发由 `catch` 块处理的异常：

C#

```
try
{
    ProcessShapes(shapeAmount);
}
catch (Exception e)
{
    LogError(e, "Shape processing failed.");
    throw;
}
```

备注

`throw;` 保留异常的原始堆栈跟踪，该跟踪存储在 Exception.StackTrace 属性中。 与此相反，`throw e;` 更新 `e` 的 StackTrace 属性。

引发异常时，公共语言运行时 (CLR) 将查找可以处理此异常的 catch 块。 如果当前执行的方法不包含此类 `catch` 块，则 CLR 查看调用了当前方法的方法，并以此类推遍历调用堆栈。 如果未找到 `catch` 块，CLR 将终止正在执行的线程。 有关详细信息，请参阅 C# 语言规范的如何处理异常部分。

`throw` 表达式

还可以将 `throw` 用作表达式。 这在很多情况下可能很方便，包括：

- 条件运算符。 以下示例使用 `throw` 表达式在传递的数组 `args` 为空时引发 ArgumentException：

```
C#

string first = args.Length >= 1
    ? args[0]
    : throw new ArgumentException("Please supply at least one
argument.");
```

- null 合并运算符。 以下示例使用 `throw` 表达式在要分配给属性的字符串为 `null` 时引发 ArgumentNullException：

```
C#

public string Name
{
    get => name;
    set => name = value ??
        throw new ArgumentNullException(paramName: nameof(value),
message: "Name cannot be null");
}
```

- expression-bodied lambda 或方法。 以下示例使用 `throw` 表达式引发 InvalidCastException，以指示不支持转换为 DateTime 值：

```
C#

DateTime ToDateTime(IFormatProvider provider) =>
         throw new InvalidCastException("Conversion to a DateTime is
not supported.");
```

`try` 语句

可以通过以下任何形式使用 `try` 语句：try-catch - 处理在 `try` 块内执行代码期间可能发生的异常，try-finally - 指定在控件离开 `try` 块时执行的代码，以及 try-catch-finally - 作为上述两种形式的组合。

`try-catch` 语句

使用 `try-catch` 语句处理在执行代码块期间可能发生的异常。 将代码置于 `try` 块中可能发生异常的位置。 使用 *catch 子句*指定要在相应的 `catch` 块中处理的异常的基类型：

```
C#

try
{
    var result = Process(-3, 4);
    Console.WriteLine($"Processing succeeded: {result}");
}
catch (ArgumentException e)
{
    Console.WriteLine($"Processing failed: {e.Message}");
}
```

可以提供多个 catch 子句：

```
C#

try
{
    var result = await ProcessAsync(-3, 4, cancellationToken);
    Console.WriteLine($"Processing succeeded: {result}");
}
catch (ArgumentException e)
{
    Console.WriteLine($"Processing failed: {e.Message}");
}
catch (OperationCanceledException)
{
    Console.WriteLine("Processing is cancelled.");
}
```

发生异常时，将从上到下按指定顺序检查 catch 子句。 对于任何引发的异常，最多只执行一个 `catch` 块。 如前面的示例所示，可以省略异常变量的声明，并在 catch 子句中仅指定异常类型。 没有任何指定异常类型的 catch 子句与任何异常匹配，如果存在，则必须是最后一个 catch 子句。

如果要重新引发捕获的异常，请使用 throw 语句，如以下示例所示：

```C#
try
{
    var result = Process(-3, 4);
    Console.WriteLine($"Processing succeeded: {result}");
}
catch (Exception e)
{
    LogError(e, "Processing failed.");
    throw;
}
```

> **备注**
>
> `throw;` 保留异常的原始堆栈跟踪，该跟踪存储在 Exception.StackTrace 属性中。 与此相反，`throw e;` 更新 `e` 的 StackTrace 属性。

`when` 异常筛选器

除了异常类型之外，还可以指定异常筛选器，该筛选器进一步检查异常并确定相应的 `catch` 块是否处理该异常。 异常筛选器是遵循 `when` 关键字的布尔表达式，如以下示例所示：

```C#
try
{
    var result = Process(-3, 4);
    Console.WriteLine($"Processing succeeded: {result}");
}
catch (Exception e) when (e is ArgumentException || e is
DivideByZeroException)
{
    Console.WriteLine($"Processing failed: {e.Message}");
}
```

前面的示例使用异常筛选器提供单个 `catch` 块来处理两个指定类型的异常。

可以为相同异常类型提供若干 `catch` 子句，如果它们通过异常筛选器区分。 其中一个子句可能没有异常筛选器。 如果存在此类子句，则它必须是指定该异常类型的最后一个子句。

如果 `catch` 子句具有异常筛选器，则可以指定与 `catch` 子句之后出现的异常类型相同或小于派生的异常类型。 例如，如果存在异常筛选器，则 `catch (Exception e)` 子句不需要是最后一个子句。

异步和迭代器方法中的异常

如果异步函数中发生异常，则等待函数的结果时，它会传播到函数的调用方，如以下示例所示：

C#

```
public static async Task Run()
{
    try
    {
        Task<int> processing = ProcessAsync(-1);
        Console.WriteLine("Launched processing.");

        int result = await processing;
        Console.WriteLine($"Result: {result}.");
    }
    catch (ArgumentException e)
    {
        Console.WriteLine($"Processing failed: {e.Message}");
    }
    // Output:
    // Launched processing.
    // Processing failed: Input must be non-negative. (Parameter 'input')
}

private static async Task<int> ProcessAsync(int input)
{
    if (input < 0)
    {
        throw new ArgumentOutOfRangeException(nameof(input), "Input must be
non-negative.");
    }

    await Task.Delay(500);
    return input;
}
```

如果迭代器方法中发生异常，则仅当迭代器前进到下一个元素时，它才会传播到调用方。

`try-finally` 语句

在 `try-finally` 语句中，当控件离开 `try` 块时，将执行 `finally` 块。 控件可能会离开 `try` 块，因为

- 正常执行，
- 执行 jump 语句（即 `return`、`break`、`continue` 或 `goto`），或
- 从 `try` 块中传播异常。

以下示例使用 `finally` 块在控件离开方法之前重置对象的状态：

C#

```
public async Task HandleRequest(int itemId, CancellationToken ct)
{
    Busy = true;

    try
    {
        await ProcessAsync(itemId, ct);
    }
    finally
    {
        Busy = false;
    }
}
```

还可以使用 `finally` 块来清理 `try` 块中使用的已分配资源。

> ⓘ **备注**
>
> 当资源类型实现 IDisposable 或 IAsyncDisposable 接口时，请考虑 using **语句。** `using` 语句可确保在控件离开 `using` 语句时释放获取的资源。 编译器将 `using` 语句转换为 `try-finally` 语句。

在几乎所有情况下，都会执行 `finally` 块。 未执行 `finally` 块的唯一情况涉及立即终止程序。 例如，由于 Environment.FailFast 调用或 OverflowException 或 InvalidProgramException 异常，可能会发生此类终止。 大多数操作系统在停止和卸载进程的过程中执行合理的资源清理。

`try-catch-finally` 语句

使用 `try-catch-finally` 语句来处理在执行 `try` 块期间可能发生的异常，并指定在控件离开 `try` 语句时必须执行的代码：

C#

```
public async Task ProcessRequest(int itemId, CancellationToken ct)
{
    Busy = true;
```

```
    try
    {
        await ProcessAsync(itemId, ct);
    }
    catch (Exception e) when (e is not OperationCanceledException)
    {
        LogError(e, $"Failed to process request for item ID {itemId}.");
        throw;
    }
    finally
    {
        Busy = false;
    }

}
```

当 `catch` 块处理异常时，`finally` 块在执行该 `catch` 块后执行（即使执行 `catch` 块期间发生另一个异常）。 有关 `catch` 和 `finally` 块的信息，请分别参阅 try-catch 语句和 try-finally 语句 部分。

C# 语言规范

有关更多信息，请参阅 C# 语言规范的以下部分：

- throw 语句
- try 语句
- 异常

另请参阅

- C# 参考
- 异常和异常处理
- 在 .NET 中处理和引发异常
- 引发首选项（样式规则 IDE0016）
- AppDomain.FirstChanceException
- AppDomain.UnhandledException
- TaskScheduler.UnobservedTaskException

Collaborate with us on GitHub

.NET feedback

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

The .NET documentation is open source. Provide feedback here.

Open a documentation issue

Provide product feedback

迭代语句 - `for`、`foreach`、`do` 和 `while`

项目 • 2023/11/20

此迭代语句重复执行语句或语句块。 for 语句：在指定的布尔表达式的计算结果为 `true` 时会执行其主体。 foreach 语句：枚举集合元素并对集合中的每个元素执行其主体。 do 语句：有条件地执行其主体一次或多次。 while 语句：有条件地执行其主体零次或多次。

在迭代语句体中的任何点，都可以使用 break 语句跳出循环。 可以使用 continue 语句进入循环中的下一个迭代。

`for` 语句

在指定的布尔表达式的计算结果为 `true` 时，`for` 语句会执行一条语句或一个语句块。 以下示例显示了 `for` 语句，该语句在整数计数器小于 3 时执行其主体：

```
C#
```

```
for (int i = 0; i < 3; i++)
{
    Console.Write(i);
}
// Output:
// 012
```

上述示例展示了 `for` 语句的元素：

- “初始化表达式”部分仅在进入循环前执行一次。 通常，在该部分中声明并初始化局部循环变量。 不能从 `for` 语句外部访问声明的变量。

  上例中的“初始化表达式”部分声明并初始化整数计数器变量：

  ```
  C#
  ```

  ```
  int i = 0
  ```

- “条件”部分确定是否应执行循环中的下一个迭代。 如果计算结果为 `true` 或不存在，则执行下一个迭代；否则退出循环。 “条件”部分必须为布尔表达式。

  上例中的“条件”条件部分检查计数器值是否小于 3：

  ```
  C#
  ```

```
i < 3
```

- “迭代器”部分定义循环主体的每次执行后将执行的操作。

  上例中的“迭代器”部分增加计数器：

  C#

  ```
  i++
  ```

- 循环体，必须是一个语句或一个语句块。

“迭代器”部分可包含用逗号分隔的零个或多个以下语句表达式：

- 为 increment 表达式添加前缀或后缀，如 `++i` 或 `i++`
- 为 decrement 表达式添加前缀或后缀，如 `--i` 或 `i--`
- assignment
- 方法的调用
- await表达式
- 通过使用 new 运算符来创建对象

如果未在“初始化表达式”部分中声明循环变量，则还可以在“初始化表达式”部分中使用上述列表中的零个或多个表达式。 下面的示例显示了几种不太常见的“初始化表达式”和“迭代器”部分的使用情况：为“初始化表达式”部分中的外部变量赋值、同时在“初始化表达式”部分和“迭代器”部分中调用一种方法，以及更改“迭代器”部分中的两个变量的值：

C#

```
int i;
int j = 3;
for (i = 0, Console.WriteLine($"Start: i={i}, j={j}"); i < j; i++, j--,
Console.WriteLine($"Step: i={i}, j={j}"))
{
    //...
}
// Output:
// Start: i=0, j=3
// Step: i=1, j=2
// Step: i=2, j=1
```

`for` 语句的所有部分都是可选的。 例如，以下代码定义无限 `for` 循环：

C#

```
for ( ; ; )
{
    //...
}
```

`foreach` 语句

`foreach` 语句为类型实例中实现 System.Collections.IEnumerable 或 System.Collections.Generic.IEnumerable<T> 接口的每个元素执行语句或语句块，如以下示例所示：

C#

```
List<int> fibNumbers = [0, 1, 1, 2, 3, 5, 8, 13];
foreach (int element in fibNumbers)
{
    Console.Write($"{element} ");
}
// Output:
// 0 1 1 2 3 5 8 13
```

`foreach` 语句并不限于这些类型。 可以将其与满足以下条件的任何类型的实例一起使用：

- 类型具有公共无参数 `GetEnumerator` 方法。 `GetEnumerator` 方法可以是类型的扩展方法。
- `GetEnumerator` 方法的返回类型具有公共 `Current` 属性和公共无参数 `MoveNext` 方法（其返回类型为 `bool`）。

下面的示例使用 `foreach` 语句，其中包含 System.Span<T> 类型的实例，该实例不实现任何接口：

C#

```
Span<int> numbers = [3, 14, 15, 92, 6];
foreach (int number in numbers)
{
    Console.Write($"{number} ");
}
// Output:
// 3 14 15 92 6
```

如果枚举器的 `Current` 属性返回引用返回值（`ref T`，其中 `T` 为集合元素类型），就可以使用 `ref` 或 `ref readonly` 修饰符来声明迭代变量，如下面的示例所示：

C#

```
Span<int> storage = stackalloc int[10];
int num = 0;
foreach (ref int item in storage)
{
    item = num++;
}
foreach (ref readonly var item in storage)
{
    Console.Write($"{item} ");
}
// Output:
// 0 1 2 3 4 5 6 7 8 9
```

如果 `foreach` 语句的源集合为空，则 `foreach` 语句的正文不会被执行，而是被跳过。 如果 `foreach` 语句应用为 `null`，则会引发 NullReferenceException。

await foreach

可以使用 `await foreach` 语句来使用异步数据流，即实现 IAsyncEnumerable<T> 接口的集合类型。 异步检索下一个元素时，可能会挂起循环的每次迭代。 下面的示例演示如何使用 `await foreach` 语句：

C#

```
await foreach (var item in GenerateSequenceAsync())
{
    Console.WriteLine(item);
}
```

还可以将 `await foreach` 语句与满足以下条件的任何类型的实例一起使用：

- 类型具有公共无参数 `GetAsyncEnumerator` 方法。 该方法可以是类型的扩展方法。
- `GetAsyncEnumerator` 方法的返回类型具有公共 `Current` 属性和公共无参数 `MoveNextAsync` 方法（其返回类型为 Task<bool>、ValueTask<bool> 或任何其他可等待类型，其 awaiter 的 `GetResult` 方法返回 `bool` 值）。

默认情况下，在捕获的上下文中处理流元素。 如果要禁用上下文捕获，请使用 TaskAsyncEnumerableExtensions.ConfigureAwait 扩展方法。 有关同步上下文并捕获当前上下文的详细信息，请参阅使用基于任务的异步模式。 有关异步流的详细信息，请参阅异步流教程。

迭代变量的类型

可以使用 var 关键字让编译器推断 `foreach` 语句中迭代变量的类型，如以下代码所示：

```
C#

foreach (var item in collection) { }
```

> **备注**
>
> 译器可以将 `var` 的类型推断为可为空的引用类型，具体取决于是否启用**可为空的感知上下文**以及初始化表达式的类型是否为引用类型。 有关详细信息，请参阅**隐式类型本地变量**。

还可以显式指定迭代变量的类型，如以下代码所示：

```
C#

IEnumerable<T> collection = new T[5];
foreach (V item in collection) { }
```

在上述窗体中，集合元素的类型 `T` 必须可隐式或显式地转换为迭代变量的类型 `V`。 如果从 `T` 到 `V` 的显式转换在运行时失败，`foreach` 语句将引发 InvalidCastException。 例如，如果 `T` 是非密封类类型，则 `V` 可以是任何接口类型，甚至可以是 `T` 未实现的接口类型。 在运行时，集合元素的类型可以是从 `T` 派生并且实际实现 `V` 的类型。 如果不是这样，则会引发 InvalidCastException。

`do` 语句

在指定的布尔表达式的计算结果为 `true` 时，`do` 语句会执行一条语句或一个语句块。 由于在每次执行循环之后都会计算此表达式，所以 `do` 循环会执行一次或多次。 `do` 循环不同于 while 循环（该循环执行零次或多次）。

下面的示例演示 `do` 语句的用法：

```
C#

int n = 0;
do
{
    Console.Write(n);
    n++;
} while (n < 5);
```

```
// Output:
// 01234
```

`while` 语句

在指定的布尔表达式的计算结果为 `true` 时，`while` 语句会执行一条语句或一个语句块。由于在每次执行循环之前都会计算此表达式，所以 `while` 循环会执行零次或多次。`while` 循环不同于 do 循环（该循环执行 1 次或多次）。

下面的示例演示 `while` 语句的用法：

C#

```
int n = 0;
while (n < 5)
{
    Console.Write(n);
    n++;
}
// Output:
// 01234
```

C# 语言规范

有关更多信息，请参阅 C# 语言规范的以下部分：

- for 语句
- foreach 语句
- do 语句
- while 语句

有关这些功能的详细信息，请参阅以下功能建议说明：

- 异步流
- 扩展 GetEnumerator 支持 foreach 循环

另请参阅

- C# 参考
- 声明
- 迭代器

在 GitHub 上与我们协作

可以在 GitHub 上找到此内容的源，还可以在其中创建和查看问题和拉取请求。 有关详细信息，请参阅参与者指南。

.NET feedback

The .NET documentation is open source. Provide feedback here.

提出文档问题

提供产品反馈

选择语句 - `if`、`if-else` 和 `switch`

项目 • 2023/08/01

`if`、`if-else` 和 `switch` 语句根据表达式的值从多个可能的语句选择要执行的路径。 仅当提供的布尔表达式的计算结果为 `true` 时，`if`，if 语句才执行语句。 语句if-else允许你根据布尔表达式选择要遵循的两个代码路径中的哪一个。 switch 语句根据与表达式匹配的模式来选择要执行的语句列表。

`if` 语句

`if` 语句可采用以下两种形式中的任一种：

- 包含 `else` 部分的 `if` 语句根据布尔表达式的值选择两个语句中的一个来执行，如以下示例所示：

```
C#

DisplayWeatherReport(15.0);  // Output: Cold.
DisplayWeatherReport(24.0);  // Output: Perfect!

void DisplayWeatherReport(double tempInCelsius)
{
    if (tempInCelsius < 20.0)
    {
        Console.WriteLine("Cold.");
    }
    else
    {
        Console.WriteLine("Perfect!");
    }
}
```

- 不包含 `else` 部分的 `if` 语句仅在布尔表达式计算结果为 `true` 时执行其主体，如以下示例所示：

```
C#

DisplayMeasurement(45);  // Output: The measurement value is 45
DisplayMeasurement(-3);  // Output: Warning: not acceptable value! The
measurement value is -3

void DisplayMeasurement(double value)
{
    if (value < 0 || value > 100)
    {
        Console.Write("Warning: not acceptable value! ");
```

```
    }

    Console.WriteLine($"The measurement value is {value}");
}
```

可嵌套 `if` 语句来检查多个条件，如以下示例所示：

C#

```
DisplayCharacter('f');  // Output: A lowercase letter: f
DisplayCharacter('R');  // Output: An uppercase letter: R
DisplayCharacter('8');  // Output: A digit: 8
DisplayCharacter(',');  // Output: Not alphanumeric character: ,

void DisplayCharacter(char ch)
{
    if (char.IsUpper(ch))
    {
        Console.WriteLine($"An uppercase letter: {ch}");
    }
    else if (char.IsLower(ch))
    {
        Console.WriteLine($"A lowercase letter: {ch}");
    }
    else if (char.IsDigit(ch))
    {
        Console.WriteLine($"A digit: {ch}");
    }
    else
    {
        Console.WriteLine($"Not alphanumeric character: {ch}");
    }
}
```

在表达式上下文中，可使用条件运算符 ?: 根据布尔表达式的值计算两个表达式中的一个。

`switch` 语句

`switch` 语句根据与匹配表达式匹配的模式来选择要执行的语句列表，如以下示例所示：

C#

```
DisplayMeasurement(-4);  // Output: Measured value is -4; too low.
DisplayMeasurement(5);  // Output: Measured value is 5.
DisplayMeasurement(30);  // Output: Measured value is 30; too high.
DisplayMeasurement(double.NaN);  // Output: Failed measurement.

void DisplayMeasurement(double measurement)
```

```
{
    switch (measurement)
    {
        case < 0.0:
            Console.WriteLine($"Measured value is {measurement}; too low.");
            break;

        case > 15.0:
            Console.WriteLine($"Measured value is {measurement}; too
high.");
            break;

        case double.NaN:
            Console.WriteLine("Failed measurement.");
            break;

        default:
            Console.WriteLine($"Measured value is {measurement}.");
            break;
    }
}
```

在上述示例中，`switch` 语句使用以下模式：

- 关系模式（在 C# 9.0 及更高版本中可用）：将表达式结果与常量进行比较。
- 常量模式：测试表达式结果是否等于常量。

> **ⓘ 重要**
>
> 有关 `switch` 语句支持的模式的信息，请参阅**模式**。

上述示例还展示了 `default` case。 `default` case 指定匹配表达式与其他任何 case 模式都不匹配时要执行的语句。 如果匹配表达式与任何 case 模式都不匹配，且没有 `default` case，控制就会贯穿 `switch` 语句。

`switch` 语句执行第一个 switch 部分中的语句列表，其 case 模式与匹配表达式匹配，并且它的 case guard（如果存在）求值为 `true` 。 `switch` 语句按文本顺序从上到下对 case 模式求值。 编译器在 `switch` 语句包含无法访问的 case 时会生成错误。 这种 case 已由大写字母处理或其模式无法匹配。

> **ⓘ 备注**
>
> `default` case 可以在 `switch` 语句的任何位置出现。 无论其位置如何，仅当所有其他事例模式都不匹配或 `goto default;` 语句在其中一个 switch 节中执行时，`default` 才会计算事例。

可以为 `switch` 语句的一部分指定多个 case 模式，如以下示例所示：

C#

```
DisplayMeasurement(-4);  // Output: Measured value is -4; out of an
acceptable range.
DisplayMeasurement(50);  // Output: Measured value is 50.
DisplayMeasurement(132);  // Output: Measured value is 132; out of an
acceptable range.

void DisplayMeasurement(int measurement)
{
    switch (measurement)
    {
        case < 0:
        case > 100:
            Console.WriteLine($"Measured value is {measurement}; out of an
acceptable range.");
            break;

        default:
            Console.WriteLine($"Measured value is {measurement}.");
            break;
    }
}
```

在 `switch` 语句中，控制不能从一个 switch 部分贯穿到下一个 switch 部分。 如本部分中的示例所示，通常使用每个 switch 部分末尾的 `break` 语句将控制从 `switch` 语句传递出去。 还可使用 return 和 throw 语句将控制从 `switch` 语句传递出去。 若要模拟贯穿行为，将控制传递给其他 switch 部分，可使用 goto 语句。

在表达式上下文中，可使用 switch 表达式，根据与表达式匹配的模式，对候选表达式列表中的单个表达式进行求值。

Case guard

case 模式可能表达功能不够，无法指定用于执行 switch 部分的条件。 在这种情况下，可以使用 case guard。 这是一个附加条件，必须与匹配模式同时满足。 case guard 必须是布尔表达式。 可以在模式后面的 `when` 关键字之后指定一个 case guard，如以下示例所示：

C#

```
DisplayMeasurements(3, 4);  // Output: First measurement is 3, second
measurement is 4.
DisplayMeasurements(5, 5);  // Output: Both measurements are valid and equal
to 5.
```

```
void DisplayMeasurements(int a, int b)
{
    switch ((a, b))
    {
        case (> 0, > 0) when a == b:
            Console.WriteLine($"Both measurements are valid and equal to
{a}.");
            break;

        case (> 0, > 0):
            Console.WriteLine($"First measurement is {a}, second measurement
is {b}.");
            break;

        default:
            Console.WriteLine("One or both measurements are not valid.");
            break;
    }
}
```

上述示例使用带有嵌套关系模式的位置模式。

C# 语言规范

有关更多信息，请参阅 C# 语言规范的以下部分：

- if 语句
- switch 语句

有关模式的详细信息，请参阅 C# 语言规范的模式和模式匹配部分。

另请参阅

- C# 参考
- 条件运算符 ?:
- 逻辑运算符
- 模式
- switch表达式
- 将缺失的事例添加到 switch 语句（样式规则 IDE0010）

跳转语句 - `break`、`continue`、`return` 和 `goto`

项目 • 2023/10/30

jump 语句无条件转移控制。 break 语句将终止最接近的封闭迭代语句或 switch 语句。 continue 语句启动最接近的封闭迭代语句的新迭代。 return 语句终止它所在的函数的执行，并将控制权返回给调用方。 goto 语句将控制权转交给带有标签的语句。

有关引发异常并无条件转移控制权的 `throw` 语句的信息，请参阅异常处理语句一文的 throw 语句部分。

`break` 语句

`break` 语句：将终止最接近的封闭迭代语句（即 `for`、`foreach`、`while` 或 `do` 循环）或 switch 语句。 `break` 语句将控制权转交给已终止语句后面的语句（若有）。

C#

```
int[] numbers = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
foreach (int number in numbers)
{
    if (number == 3)
    {
        break;
    }

    Console.Write($"{number} ");
}
Console.WriteLine();
Console.WriteLine("End of the example.");
// Output:
// 0 1 2
// End of the example.
```

在嵌套循环中，`break` 语句仅终止包含它的最内部循环，如以下示例所示：

C#

```
for (int outer = 0; outer < 5; outer++)
{
    for (int inner = 0; inner < 5; inner++)
    {
        if (inner > outer)
        {
            break;
```

```
        }

        Console.Write($"{inner} ");
    }
    Console.WriteLine();
}
// Output:
// 0
// 0 1
// 0 1 2
// 0 1 2 3
// 0 1 2 3 4
```

在循环内使用 `switch` 语句时，switch 节末尾的 `break` 语句仅从 `switch` 语句中转移控制权。 包含 `switch` 语句的循环不受影响，如以下示例所示：

C#

```
double[] measurements = { -4, 5, 30, double.NaN };
foreach (double measurement in measurements)
{
    switch (measurement)
    {
        case < 0.0:
            Console.WriteLine($"Measured value is {measurement}; too low.");
            break;

        case > 15.0:
            Console.WriteLine($"Measured value is {measurement}; too
high.");
            break;

        case double.NaN:
            Console.WriteLine("Failed measurement.");
            break;

        default:
            Console.WriteLine($"Measured value is {measurement}.");
            break;
    }
}
// Output:
// Measured value is -4; too low.
// Measured value is 5.
// Measured value is 30; too high.
// Failed measurement.
```

`continue` 语句

`continue` 语句启动最接近的封闭迭代语句（即 `for`、`foreach`、`while` 或 `do` 循环）的新迭代，如以下示例所示：

C#

```
for (int i = 0; i < 5; i++)
{
    Console.Write($"Iteration {i}: ");

    if (i < 3)
    {
        Console.WriteLine("skip");
        continue;
    }

    Console.WriteLine("done");
}
// Output:
// Iteration 0: skip
// Iteration 1: skip
// Iteration 2: skip
// Iteration 3: done
// Iteration 4: done
```

`return` 语句

`return` 语句终止它所在的函数的执行，并将控制权和函数结果（若有）返回给调用方。

如果函数成员不计算值，则使用不带表达式的 `return` 语句，如以下示例所示：

C#

```
Console.WriteLine("First call:");
DisplayIfNecessary(6);

Console.WriteLine("Second call:");
DisplayIfNecessary(5);

void DisplayIfNecessary(int number)
{
    if (number % 2 == 0)
    {
        return;
    }

    Console.WriteLine(number);
}
// Output:
// First call:
```

```
// Second call:
// 5
```

如前面的示例所示，通常使用不带表达式的 `return` 语句提前终止函数成员。 如果函数成员不包含 `return` 语句，则在执行其最后一个语句后终止。

如果函数成员不计算值，则使用带表达式的 `return` 语句，如以下示例所示：

C#

```
double surfaceArea = CalculateCylinderSurfaceArea(1, 1);
Console.WriteLine($"{surfaceArea:F2}"); // output: 12.57

double CalculateCylinderSurfaceArea(double baseRadius, double height)
{
    double baseArea = Math.PI * baseRadius * baseRadius;
    double sideArea = 2 * Math.PI * baseRadius * height;
    return 2 * baseArea + sideArea;
}
```

如果 `return` 语句具有表达式，该表达式必须可隐式转换为函数成员的返回类型，除非它是异步的。 从 `async` 函数返回的表达式必须隐式转换为 Task<TResult> 或 ValueTask<TResult> 类型参数，以函数的返回类型为准。 如果 `async` 函数的返回类型为 Task 或 ValueTask，则使用不带表达式的 `return` 语句。

引用返回

默认情况下，`return` 语句返回表达式的值。 可以返回对变量的引用。 引用返回值（或 ref 返回值）是由方法按引用向调用方返回的值。 即是说，调用方可以修改方法所返回的值，此更改反映在所调用方法中的对象的状态中。 为此，请使用带 `ref` 关键字的 `return` 语句，如以下示例所示：

C#

```
var xs = new int[] { 10, 20, 30, 40 };
ref int found = ref FindFirst(xs, s => s == 30);
found = 0;
Console.WriteLine(string.Join(" ", xs));  // output: 10 20 0 40

ref int FindFirst(int[] numbers, Func<int, bool> predicate)
{
    for (int i = 0; i < numbers.Length; i++)
    {
        if (predicate(numbers[i]))
        {
            return ref numbers[i];
        }
```

```
    }
    throw new InvalidOperationException("No element satisfies the given
condition.");
}
```

借助引用返回值，方法可以将对变量的引用（而不是值）返回给调用方。 然后，调用方可以选择将返回的变量视为按值返回或按引用返回。 调用方可以新建称为引用本地的变量，其本身就是对返回值的引用。 引用返回值是指，方法返回对某变量的引用（或别名）。 相应变量的作用域必须包括方法。 相应变量的生存期必须超过方法的返回值。 调用方对方法的返回值进行的修改应用于方法返回的变量。

如果声明方法返回引用返回值，表明方法返回变量别名。 设计意图通常是让调用代码通过别名访问此变量（包括修改它）。 方法的引用返回值不得包含返回类型 `void`。

为方便调用方修改对象的状态，引用返回值必须存储在被显式定义为 reference 变量的变量中。

`ref` 返回值是被调用方法范围中另一个变量的别名。 可以将引用返回值的所有使用都解释为，使用它取别名的变量：

- 分配值时，就是将值分配到它取别名的变量。
- 读取值时，就是读取它取别名的变量的值。
- 如果以引用方式返回它，就是返回对相同变量所取的别名。
- 如果以引用方式将它传递到另一个方法，就是传递对它取别名的变量的引用。
- 如果返回引用本地别名，就是返回相同变量的新别名。

引用返回必须是调用方法的 ref-safe-context。 也就是说：

- 返回值的生存期必须长于方法执行时间。 换言之，它不能是返回自身的方法中的本地变量。 它可以是实例或类的静态字段，也可是传递给方法的参数。 尝试返回局部变量将生成编译器错误 CS8168：“无法按引用返回局部 "obj"，因为它不是 ref 局部变量”。
- 返回值不得为文本 `null`。 使用引用返回值的方法可以返回值当前为 `null`（未实例化）或可为空的值类型的变量别名。
- 返回值不得为常量、枚举成员、通过属性的按值返回值或 `class`/`struct` 方法。

此外，禁止对异步方法使用引用返回值。 异步方法可能会在执行尚未完成时就返回值，尽管返回值仍未知。

返回引用返回值的方法必须：

- 在返回类型前面有 ref 关键字。
- 方法主体中的每个 return 语句都在返回实例的名称前面有 ref 关键字。

下面的示例方法满足这些条件，且返回对名为 `p` 的 `Person` 对象的引用：

C#

```
public ref Person GetContactInformation(string fname, string lname)
{
    // ...method implementation...
    return ref p;
}
```

下面是一个更完整的 ref 返回示例，同时显示方法签名和方法主体。

C#

```
public static ref int Find(int[,] matrix, Func<int, bool> predicate)
{
    for (int i = 0; i < matrix.GetLength(0); i++)
        for (int j = 0; j < matrix.GetLength(1); j++)
            if (predicate(matrix[i, j]))
                return ref matrix[i, j];
    throw new InvalidOperationException("Not found");
}
```

所调用方法还可能会将返回值声明为 `ref readonly` 以按引用返回值，并坚持调用代码无法修改返回的值。 调用方法可以通过将返回值存储在局部 `ref readonly` reference 变量中来避免复制该值。

下列示例定义一个具有两个 String 字段（`Title` 和 `Author`）的 `Book` 类。 还定义包含 `Book` 对象的专用数组的 `BookCollection` 类。 通过调用 `GetBookByTitle` 方法，可按引用返回个别 book 对象。

C#

```


public class Book
{
    public string Author;
    public string Title;
}

public class BookCollection
{
    private Book[] books = { new Book { Title = "Call of the Wild, The",
Author = "Jack London" },
                             new Book { Title = "Tale of Two Cities, A", Author =
"Charles Dickens" }
                           };
    private Book nobook = null;

    public ref Book GetBookByTitle(string title)
    {
```

```
        for (int ctr = 0; ctr < books.Length; ctr++)
        {
            if (title == books[ctr].Title)
                return ref books[ctr];
        }
        return ref nobook;
    }

    public void ListBooks()
    {
        foreach (var book in books)
        {
            Console.WriteLine($"{book.Title}, by {book.Author}");
        }
        Console.WriteLine();
    }
}
```

调用方将 `GetBookByTitle` 方法所返回的值存储为 ref 局部变量时，调用方对返回值所做的更改将反映在 `BookCollection` 对象中，如下例所示。

C#

```
var bc = new BookCollection();
bc.ListBooks();

ref var book = ref bc.GetBookByTitle("Call of the Wild, The");
if (book != null)
    book = new Book { Title = "Republic, The", Author = "Plato" };
bc.ListBooks();
// The example displays the following output:
//       Call of the Wild, The, by Jack London
//       Tale of Two Cities, A, by Charles Dickens
//
//       Republic, The, by Plato
//       Tale of Two Cities, A, by Charles Dickens
```

`goto` 语句

`goto` 语句将控制权转交给带有标签的语句，如以下示例所示：

C#

```
var matrices = new Dictionary<string, int[][]>
{
    ["A"] = new[]
    {
        new[] { 1, 2, 3, 4 },
        new[] { 4, 3, 2, 1 }
```

```
    },
    ["B"] = new[]
    {
        new[] { 5, 6, 7, 8 },
        new[] { 8, 7, 6, 5 }
    },
};

CheckMatrices(matrices, 4);

void CheckMatrices(Dictionary<string, int[][]> matrixLookup, int target)
{
    foreach (var (key, matrix) in matrixLookup)
    {
        for (int row = 0; row < matrix.Length; row++)
        {
            for (int col = 0; col < matrix[row].Length; col++)
            {
                if (matrix[row][col] == target)
                {
                    goto Found;
                }
            }
        }
        Console.WriteLine($"Not found {target} in matrix {key}.");
        continue;

    Found:
        Console.WriteLine($"Found {target} in matrix {key}.");
    }
}
// Output:
// Found 4 in matrix A.
// Not found 4 in matrix B.
```

如前面的示例所示，可以使用 `goto` 语句退出嵌套循环。

> **提示**
>
> 使用嵌套循环时，请考虑将单独的循环重构为单独的方法。 这可能会导致没有 `goto` 语句的更简单、更具可读性的代码。

还可使用 switch 语句中的 `goto` 语句将控制权移交到具有常量大小写标签的 switch 节，如以下示例所示：

C#

```
using System;

public enum CoffeeChoice
```

```
{
    Plain,
    WithMilk,
    WithIceCream,
}

public class GotoInSwitchExample
{
    public static void Main()
    {
        Console.WriteLine(CalculatePrice(CoffeeChoice.Plain));  // output:
10.0
        Console.WriteLine(CalculatePrice(CoffeeChoice.WithMilk));  //
output: 15.0
        Console.WriteLine(CalculatePrice(CoffeeChoice.WithIceCream));  //
output: 17.0
    }

    private static decimal CalculatePrice(CoffeeChoice choice)
    {
        decimal price = 0;
        switch (choice)
        {
            case CoffeeChoice.Plain:
                price += 10.0m;
                break;

            case CoffeeChoice.WithMilk:
                price += 5.0m;
                goto case CoffeeChoice.Plain;

            case CoffeeChoice.WithIceCream:
                price += 7.0m;
                goto case CoffeeChoice.Plain;
        }
        return price;
    }
}
```

在 `switch` 语句中，还可使用语句 `goto default;` 将控制权转交给带 `default` 标签的 switch 节。

如果当前函数成员中不存在具有给定名称的标签，或者 `goto` 语句不在标签范围内，则会出现编译时错误。 也就是说，你不能使用 `goto` 语句将控制权从当前函数成员转移到任何嵌套范围。

C# 语言规范

有关更多信息，请参阅 C# 语言规范的以下部分：

- break 语句
- continue 语句
- return 语句
- goto 语句

另请参阅

- C# 参考
- yield 语句

在 GitHub 上与我们协作

可以在 GitHub 上找到此内容的源，还可以在其中创建和查看问题和拉取请求。 有关详细信息，请参阅参与者指南。

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

提出文档问题

提供产品反馈

已选中和未选中的语句（C# 参考）

项目 • 2024/11/02

`checked` 和 `unchecked` 语句指定整型类型算术运算和转换的溢出检查上下文。 当发生整数算术溢出时，溢出检查上下文将定义发生的情况。 在已检查的上下文中，引发 System.OverflowException；如果在常数表达式中发生溢出，则会发生编译时错误。 在未检查的上下文中，会通过丢弃任何不适应目标类型的高序位来将操作结果截断。 例如，加法从最大值换行到最小值。 以下示例显示了已检查和未检查上下文中的相同操作：

C#

```
uint a = uint.MaxValue;

unchecked
{
    Console.WriteLine(a + 3);  // output: 2
}

try
{
    checked
    {
        Console.WriteLine(a + 3);
    }
}
catch (OverflowException e)
{
    Console.WriteLine(e.Message);  // output: Arithmetic operation resulted
in an overflow.
}
```

> **备注**
>
> 用户定义的*运算符和转换的*溢出行为可能与上一段中所述的行为不同。 特别是，**用户定义的 checked 运算符**可能不会在已检查的上下文中引发异常。

有关详细信息，请参阅算术运算符一文的算术溢出和被零除以及用户定义的 checked 运算符部分。

若要为表达式指定溢出检查上下文，还可以使用 `checked` 和 `unchecked` 运算符，如以下示例所示：

C#

```
double a = double.MaxValue;

int b = unchecked((int)a);
Console.WriteLine(b);  // output: -2147483648

try
{
    b = checked((int)a);
}
catch (OverflowException e)
{
    Console.WriteLine(e.Message);  // output: Arithmetic operation resulted
in an overflow.
}
```

`checked` 和 `unchecked` 语句和运算符仅影响以文本形式存在于语句块或运算符括号内的操作的溢出检查上下文，如以下示例所示：

C#

```
int Multiply(int a, int b) => a * b;

int factor = 2;

try
{
    checked
    {
        Console.WriteLine(Multiply(factor, int.MaxValue));  // output: -2
    }
}
catch (OverflowException e)
{
    Console.WriteLine(e.Message);
}

try
{
    checked
    {
        Console.WriteLine(Multiply(factor, factor * int.MaxValue));
    }
}
catch (OverflowException e)
{
    Console.WriteLine(e.Message);  // output: Arithmetic operation resulted
in an overflow.
}
```

在前面的示例中，第一次调用 `Multiply` 本地函数表明，`checked` 语句不会影响 `Multiply` 函数中的溢出检查上下文，因为不会引发任何异常。 在第二次调用 `Multiply` 函数时，计算函数第二个参数的表达式将在已检查的上下文中计算，并导致异常，因为它以文本形式存在于 `checked` 语句的块内。

其行为 `checked` 取决于 `unchecked` 类型和操作。 即使对于整数，操作也总是 `unchecked(x / 0)` 会引发，因为没有合理的行为。 检查类型和操作的行为，以了解代码和`unchecked`关键字如何影响`checked`代码。

数值类型和溢出检查上下文

关键字`checked` `unchecked`主要适用于存在合理溢出行为的整型类型。 换行行为，即 `T.MaxValue + 1` 在两者补充值中变得 `T.MinValue` 合理。 表示的值不正确，因为它不适合该类型的存储。 因此，位代表完整结果的下 n 位。

对于类型（例如`decimal`， `float`和`double` `Half`表示更复杂的值或补充值）而言，换行并不合理。 它不能用于计算更大或更准确的结果，因此 `unchecked` 没有好处。

`float`， `double`并且 `Half` 具有合理的饱和值 `PositiveInfinity` ， `NegativeInfinity`因此可以在上下文中 `unchecked` 检测溢出。 因此 `decimal`，不存在此类限制，饱和 `MaxValue` 可能导致错误或混淆。 在上下文`checked` `unchecked`中使用`decimal`引发的操作。

受溢出检查上下文影响的操作

溢出检查上下文会影响以下操作：

- 以下内置算术运算符：一元 `++`、`--`、`-` 和二元 `+`、`-`、`*` 和 `/` 运算符，当它们的操作数为整型类型（即整数或字符类型）或枚举类型时。
- 整型类型之间或从 float 或 double 到整型类型的显式数字转换。

  > **① 备注**
  >
  > 在将 `decimal` 值转换为整型类型并且结果超出目标类型的范围时，不管溢出检查上下文如何，都始终会引发 OverflowException。

- 从 C# 11 开始，用户定义的 checked 运算符和转换。 有关详细信息，请参阅算术运算符一文的用户定义的 checked 运算符部分。

默认溢出检查上下文

如果未指定溢出检查上下文，CheckForOverflowUnderflow **编译器选项的值**将定义非常量表达式的默认上下文。 默认情况下，该选项的值未设置，并且整型算术运算和转换在未检查的上下文中执行。

常量表达式在检查的上下文中默认计算，溢出会导致编译时错误。 可以使用 `unchecked` 语句或运算符为常数表达式显式指定未检查的上下文。

C# 语言规范

有关更多信息，请参阅 C# 语言规范的以下部分：

- checked 和 unchecked 语句
- checked 和 unchecked 运算符
- 用户定义的已检查和未检查运算符 - C# 11

另请参阅

- CheckForOverflowUnderflow 编译器选项

fixed 语句 - 固定用于指针操作的变量

项目 • 2024/04/01

`fixed` 语句可防止垃圾回收器重新定位可移动变量，并声明指向该变量的指针。 固定变量的地址在语句的持续时间内不会更改。 只能在相应的 `fixed` 语句中使用声明的指针。 声明的指针是只读的，无法修改：

C#

```
unsafe
{
    byte[] bytes = [1, 2, 3];
    fixed (byte* pointerToFirst = bytes)
    {
        Console.WriteLine($"The address of the first array element:
{(long)pointerToFirst:X}.");
        Console.WriteLine($"The value of the first array element:
{*pointerToFirst}.");
    }
}
// Output is similar to:
// The address of the first array element: 2173F80B5C8.
// The value of the first array element: 1.
```

> ⓘ **备注**
>
> 只能在**不安全的**上下文中使用 `fixed` 语句。 必须使用 AllowUnsafeBlocks 编译器选项来编译包含不安全块的代码。

可以按如下所示初始化声明的指针：

- 使用数组，如本文开头的示例所示。 初始化的指针包含第一个数组元素的地址。
- 使用变量的地址。 使用 address-of & 运算符，如以下示例所示：

  C#

  ```
  unsafe
  {
      int[] numbers = [10, 20, 30];
      fixed (int* toFirst = &numbers[0], toLast = &numbers[^1])
      {
          Console.WriteLine(toLast - toFirst);  // output: 2
      }
  }
  ```

对象字段是可以固定的可移动变量的另一个示例。

当初始化的指针包含对象字段或数组元素的地址时，`fixed` 语句保证垃圾回收器在语句主体执行期间不会重新定位或释放包含对象实例。

- 使用实现名为 `GetPinnableReference` 的方法的类型的实例。 该方法必须返回非托管类型的 `ref` 变量。 .NET 类型 System.Span<T> 和 System.ReadOnlySpan<T> 使用此模式。 可以固定跨度实例，如以下示例所示：

  C#

  ```
  unsafe
  {
      int[] numbers = [10, 20, 30, 40, 50];
      Span<int> interior = numbers.AsSpan()[1..^1];
      fixed (int* p = interior)
      {
          for (int i = 0; i < interior.Length; i++)
          {
              Console.Write(p[i]);
          }
          // output: 203040
      }
  }
  ```

  有关详细信息，请参阅 Span<T>.GetPinnableReference() API 参考。

- 使用字符串，如以下示例所示：

  C#

  ```
  unsafe
  {
      var message = "Hello!";
      fixed (char* p = message)
      {
          Console.WriteLine(*p);  // output: H
      }
  }
  ```

- 使用固定大小的缓冲区。

可以在堆栈上分配内存，在这种情况下，内存不受垃圾回收的约束，因此不需要固定。 为此，请使用 stackalloc 表达式。

还可以使用 `fixed` 关键字声明固定大小的缓冲区。

C# 语言规范

有关更多信息，请参阅 C# 语言规范的以下部分：

- fixed 语句
- 固定和可移动变量

另请参阅

- 不安全代码、指针类型和函数指针
- 与指针相关的运算符
- unsafe

在 GitHub 上与我们协作

可以在 GitHub 上找到此内容的源，还可以在其中创建和查看问题和拉取请求。 有关详细信息，请参阅参与者指南。

.NET

.NET 反馈

.NET 是一个开放源代码项目。 选择一个链接以提供反馈：

提出文档问题

提供产品反馈

lock 语句 - 确保对共享资源的独占访问权限

项目 • 2024/05/10

`lock` 语句获取给定对象的互斥 lock，执行语句块，然后释放 lock。 持有 lock 时，持有 lock 的线程可以再次获取并释放 lock。 阻止任何其他线程获取 lock 并等待释放 lock。 `lock` 语句可确保在任何时候最多只有一个线程执行其主体。

`lock` 语句采用以下形式：

C#

```
lock (x)
{
    // Your code...
}
```

变量 `x` 是 System.Threading.Lock 类型或引用类型的表达式。 当 `x` 在编译时已知属于类型 System.Threading.Lock 时，它完全等效于：

C#

```
using (x.EnterScope())
{
    // Your code...
}
```

Lock.EnterScope() 返回的对象是包括一个 `Dispose()` 方法的 ref struct。 生成的 using 语句可确保即使 `lock` 语句正文引发异常，也会释放范围。

否则，该 `lock` 语句完全等效于：

C#

```
object __lockObj = x;
bool __lockWasTaken = false;
try
{
    System.Threading.Monitor.Enter(__lockObj, ref __lockWasTaken);
    // Your code...
}
finally
{
```

```
    if (__lockWasTaken) System.Threading.Monitor.Exit(__lockObj);
}
```

由于该代码使用 try-finally 语句，因此即使在 `lock` 语句的正文中引发异常，也会释放 lock。

在 `lock` 语句的正文中不能使用 await 表达式。

准则

从 .NET 9 和 C# 13 开始，锁定 System.Threading.Lock 类型的专用对象实例以获取最佳性能。 此外，如果已知的 `Lock` 对象被强制转换为另一种类型并锁定，编译器会发出警告。 如果使用旧版的 .NET 和 C#，请锁定不用于其他用途的专用对象实例。 避免对不同的共享资源使用相同的 lock 对象实例，因为这可能导致死锁或锁争用。 具体而言，请避免将以下实例用作 lock 对象：

- `this`，因为调用方也可能锁定 `this`。
- Type 实例（可以通过 typeof 运算符或反射获取）。
- 字符串实例，包括字符串字面量，（这些可能是暂存的）。

尽可能缩短持有锁的时间，以减少锁争用。

示例

以下示例定义了一个 `Account` 类，该类通过锁定专用的 `balanceLock` 实例来同步对其专用 `balance` 字段的访问。 使用同一实例进行锁定可确保两个不同的线程不能同时调用 `Debit` 或 `Credit` 方法更新 `balance` 字段。 此示例使用 C# 13 和新 `Lock` 对象。 如果使用较旧版本的 C# 或较旧的 .NET 库，请锁定 `object` 实例。

C#

```
using System;
using System.Threading.Tasks;

public class Account
{
    // Use `object` in versions earlier than C# 13
    private readonly System.Threading.Lock _balanceLock = new();
    private decimal _balance;

    public Account(decimal initialBalance) => _balance = initialBalance;

    public decimal Debit(decimal amount)
    {
        if (amount < 0)
```

```
        {
            throw new ArgumentOutOfRangeException(nameof(amount), "The debit
amount cannot be negative.");
        }

        decimal appliedAmount = 0;
        lock (_balanceLock)
        {
            if (_balance >= amount)
            {
                _balance -= amount;
                appliedAmount = amount;
            }
        }
        return appliedAmount;
    }

    public void Credit(decimal amount)
    {
        if (amount < 0)
        {
            throw new ArgumentOutOfRangeException(nameof(amount), "The
credit amount cannot be negative.");
        }

        lock (_balanceLock)
        {
            _balance += amount;
        }
    }

    public decimal GetBalance()
    {
        lock (_balanceLock)
        {
            return _balance;
        }
    }
}

class AccountTest
{
    static async Task Main()
    {
        var account = new Account(1000);
        var tasks = new Task[100];
        for (int i = 0; i < tasks.Length; i++)
        {
            tasks[i] = Task.Run(() => Update(account));
        }
        await Task.WhenAll(tasks);
        Console.WriteLine($"Account's balance is {account.GetBalance()}");
        // Output:
        // Account's balance is 2000
    }
```

```
    static void Update(Account account)
    {
        decimal[] amounts = [0, 2, -3, 6, -2, -1, 8, -5, 11, -6];
        foreach (var amount in amounts)
        {
            if (amount >= 0)
            {
                account.Credit(amount);
            }
            else
            {
                account.Debit(Math.Abs(amount));
            }
        }
    }
}
```

C# 语言规范

有关详细信息，请参阅 C# 语言规范中的 lock 语句部分。

另请参阅

- System.Threading.Monitor
- System.Threading.SpinLock
- System.Threading.Interlocked
- 同步基元概述
- System.Threading.Channels 简介

在 GitHub 上与我们协作

可以在 GitHub 上找到此内容的源，还可以在其中创建和查看问题和拉取请求。 有关详细信息，请参阅参与者指南。

.NET 反馈

.NET 是一个开放源代码项目。 选择一个链接以提供反馈：

提出文档问题

提供产品反馈

using 语句 - 确保正确使用可释放对象

项目 • 2023/03/20

`using` 语句可确保正确使用 IDisposable 实例：

C#

```
var numbers = new List<int>();
using (StreamReader reader = File.OpenText("numbers.txt"))
{
    string line;
    while ((line = reader.ReadLine()) is not null)
    {
        if (int.TryParse(line, out int number))
        {
            numbers.Add(number);
        }
    }
}
```

当控件离开 `using` 语句块时，将释放获取的 IDisposable 实例。 `using` 语句可确保即使在 `using` 语句块内发生异常的情况下也会释放可释放实例。 在前面的示例中，打开的文件在处理完所有行后关闭。

使用 `await using` 语句来正确使用 IAsyncDisposable 实例：

C#

```
await using (var resource = new AsyncDisposableExample())
{
    // Use the resource
}
```

有关使用 IAsyncDisposable 实例的详细信息，请参阅实现 DisposeAsync 方法一文中的使用异步可释放对象部分。

还可以使用不需要大括号的 `using` 声明：

C#

```
static IEnumerable<int> LoadNumbers(string filePath)
{
    using StreamReader reader = File.OpenText(filePath);

    var numbers = new List<int>();
    string line;
    while ((line = reader.ReadLine()) is not null)
```

```
    {
        if (int.TryParse(line, out int number))
        {
            numbers.Add(number);
        }
    }
    return numbers;
}
```

在 `using` 声明中进行声明时，局部变量在声明它的作用域末尾释放。 在前面的示例中，会在某个方法的末尾释放对象。

由 `using` 语句或声明进行声明的变量是只读的。 无法重新分配该变量或将其作为 ref 或 out 参数传递。

可以在一个 `using` 语句中声明多个相同类型的实例，如以下示例所示：

C#

```
using (StreamReader numbersFile = File.OpenText("numbers.txt"), wordsFile =
File.OpenText("words.txt"))
{
    // Process both files
}
```

在一个 `using` 语句中声明多个实例时，它们将按声明的相反顺序释放。

还可以将 `using` 语句和声明与适用于可释放模式的 ref 结构的实例一起使用。 也就是说，它有一个实例 `Dispose` 方法，该方法是可访问、无参数的并且具有 `void` 返回类型。

`using` 语句也可以采用以下形式：

C#

```
using (expression)
{
    // ...
}
```

其中 `expression` 会生成可释放实例。 下面的示例演示了这一操作：

C#

```
StreamReader reader = File.OpenText(filePath);

using (reader)
{
```

```
    // Process file content
}
```

> **⚠ 警告**
>
> 在前面的示例中，当控件离开 `using` 语句后，可释放实例会在其释放后仍保留在范围内。 如果进一步使用该实例，可能会遇到异常，例如 **ObjectDisposedException**。 因此，建议在 `using` 语句中声明可释放变量，或者使用 `using` 声明进行声明。

C# 语言规范

有关详细信息，请参阅 C# 语言规范的 using 语句部分以及关于“基于模式的 using”和“using 声明”的建议说明。

另请参阅

- C# 参考
- System.IDisposable
- System.IAsyncDisposable
- 使用实现 IDisposable 的对象
- 实现 Dispose 方法
- 实现 DisposeAsync 方法
- 使用简单的“using”语句（样式规则 IDE0063）
- using 指令

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET feedback

The .NET documentation is open source. Provide feedback here.

Open a documentation issue

Provide product feedback

yield 语句 - 提供下一个元素

项目 • 2024/11/23

在迭代器中使用 `yield` 语句提供下一个值或表示迭代结束。 `yield` 语句有以下两种形式：

- `yield return`：在迭代中提供下一个值，如以下示例所示：

  C#

  ```
  foreach (int i in ProduceEvenNumbers(9))
  {
      Console.Write(i);
      Console.Write(" ");
  }
  // Output: 0 2 4 6 8

  IEnumerable<int> ProduceEvenNumbers(int upto)
  {
      for (int i = 0; i <= upto; i += 2)
      {
          yield return i;
      }
  }
  ```

- `yield break`：显式示迭代结束，如以下示例所示：

  C#

  ```
  Console.WriteLine(string.Join(" ", TakeWhilePositive(new int[] {2, 3,
  4, 5, -1, 3, 4})));
  // Output: 2 3 4 5

  Console.WriteLine(string.Join(" ", TakeWhilePositive(new int[] {9, 8,
  7})));
  // Output: 9 8 7

  IEnumerable<int> TakeWhilePositive(IEnumerable<int> numbers)
  {
      foreach (int n in numbers)
      {
          if (n > 0)
          {
              yield return n;
          }
          else
          {
              yield break;
          }
  ```

```
    }
}
```

当控件到达迭代器的末尾时，迭代也结束。

在前面的示例中，迭代器的返回类型为 IEnumerable<T>（在非泛型情况下，使用 IEnumerable 作为迭代器的返回类型）。 还可以使用 IAsyncEnumerable<T> 作为迭代器的返回类型。 这使得迭代器异步。 使用 await foreach 语句对迭代器的结果进行迭代，如以下示例所示：

C#

```
await foreach (int n in GenerateNumbersAsync(5))
{
    Console.Write(n);
    Console.Write(" ");
}
// Output: 0 2 4 6 8

async IAsyncEnumerable<int> GenerateNumbersAsync(int count)
{
    for (int i = 0; i < count; i++)
    {
        yield return await ProduceNumberAsync(i);
    }
}

async Task<int> ProduceNumberAsync(int seed)
{
    await Task.Delay(1000);
    return 2 * seed;
}
```

迭代器的返回类型可以是 IEnumerator<T> 或 IEnumerator。 在以下方案中实现 `GetEnumerator` 方法时，请使用这些返回类型：

- 设计实现 IEnumerable<T> 或 IEnumerable 接口的类型。
- 添加实例或扩展 `GetEnumerator` 方法来使用 foreach 语句对类型的实例启用迭代，如以下示例所示：

  C#

  ```
  public static void Example()
  {
      var point = new Point(1, 2, 3);
      foreach (int coordinate in point)
      {
          Console.Write(coordinate);
  ```

```
            Console.Write(" ");
        }
        // Output: 1 2 3
    }

    public readonly record struct Point(int X, int Y, int Z)
    {
        public IEnumerator<int> GetEnumerator()
        {
            yield return X;
            yield return Y;
            yield return Z;
        }
    }
```

不能在下列情况中使用 `yield` 语句：

- 带有 in、ref 或 out 参数的方法
- Lambda 表达式和匿名方法
- 不安全块。 在 C# 13 之前，`yield` 在具有 `unsafe` 块的任何方法中都无效。 从 C# 13 开始，可以在包含 `unsafe` 块的方法中使用 `yield`，但不能在 `unsafe` 块中使用。
- `yield return` `yield break` 不能在 catch 和 finally 块中使用，也不能在具有相应 `catch` 块的 try 块中使用。 和 `yield return` `yield break` 语句可以在没有 `catch` 块的块（只有块 `finally`）的块中使用 `try`。

迭代器的执行

迭代器的调用不会立即执行，如以下示例所示：

C#

```
var numbers = ProduceEvenNumbers(5);
Console.WriteLine("Caller: about to iterate.");
foreach (int i in numbers)
{
    Console.WriteLine($"Caller: {i}");
}

IEnumerable<int> ProduceEvenNumbers(int upto)
{
    Console.WriteLine("Iterator: start.");
    for (int i = 0; i <= upto; i += 2)
    {
        Console.WriteLine($"Iterator: about to yield {i}");
        yield return i;
        Console.WriteLine($"Iterator: yielded {i}");
    }
```

```
    Console.WriteLine("Iterator: end.");
}
// Output:
// Caller: about to iterate.
// Iterator: start.
// Iterator: about to yield 0
// Caller: 0
// Iterator: yielded 0
// Iterator: about to yield 2
// Caller: 2
// Iterator: yielded 2
// Iterator: about to yield 4
// Caller: 4
// Iterator: yielded 4
// Iterator: end.
```

如前面的示例所示，当开始对迭代器的结果进行迭代时，迭代器会一直执行，直到到达第一个 `yield return` 语句为止。 然后，迭代器的执行会暂停，调用方会获得第一个迭代值并处理该值。 在后续的每次迭代中，迭代器的执行都会在导致上一次挂起的 `yield return` 语句之后恢复，并继续执行，直到到达下一个 `yield return` 语句为止。 当控件到达迭代器或 `yield break` 语句的末尾时，迭代完成。

C# 语言规范

有关详细信息，请参阅 C# 语言规范中的 yield 语句部分。

另请参阅

- 迭代器
- 循环访问 C# 中的集合
- foreach
- await foreach

C# 特殊字符

项目 • 2023/04/07

特殊字符是预定义的上下文字符，用于修改最前面插入了此类字符的程序元素（文本字符串、标识符或属性名称）。 C# 支持以下特殊字符：

- @：逐字字符串标识符字符。
- $：内插的字符串字符。

本部分仅包括非运算符标记。 如需了解所有运算符，请参阅运算符部分。

请参阅

- C# 参考
- C# 编程指南

代码注释 - `//` 和 `/*`。 `*/`

项目 • 2023/04/07

C# 支持两种不同形式的注释。 单行注释以 `//` 开头，并在该代码行末尾结束。 多行注释以 `/*` 开头，以 `*/` 结尾。 下面的代码示例演示了每种注释：

C#

```
// This is a single line comment.

/* This could be a summary of all the
   code that's in this class.
   You might add multiple paragraphs, or links to pages
   like https://learn.microsoft.com/dotnet/csharp.

   You could even include emojis. This example is 🔥
   Then, when you're done, close with
   */
```

多行注释还可用于在代码行中插入文本。 由于这些注释具有显式结束字符，因此可以在注释后面加上更多可执行代码：

C#

```
public static int Add(int left, int right)
{
    return left /* first operand */ + right /* second operand */;
}
```

单行注释可以出现在同一行上的可执行代码之后。 注释在文本行末尾结束：

C#

```
return source++; // increment the source.
```

某些注释以三个斜杠开头：`///`。 三斜杠注释是 XML 文档注释。 编译器读取这些注释以生成人工文档。 有关 XML 文档注释的详细信息，请参阅三斜杠注释部分。

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

can also create and review issues and pull requests. For more information, see our contributor guide.

Open a documentation issue

Provide product feedback

使用 `$` 的字符串内插

项目 • 2024/08/07

`$` 字符将字符串字面量标识为内插字符串。 内插字符串是可能包含内插表达式的字符串文本 。 将内插字符串解析为结果字符串时，编译程序会将带有内插表达式的项替换为表达式结果的字符串表示形式。

字符串内插为格式化字符串提供了一种可读性和便捷性更高的方式。 它比字符串复合格式设置更容易阅读。 下面的示例使用了这两种功能生成同样的输出结果：

C#

```
var name = "Mark";
var date = DateTime.Now;

// Composite formatting:
Console.WriteLine("Hello, {0}! Today is {1}, it's {2:HH:mm} now.", name,
date.DayOfWeek, date);
// String interpolation:
Console.WriteLine($"Hello, {name}! Today is {date.DayOfWeek}, it's
{date:HH:mm} now.");
// Both calls produce the same output that is similar to:
// Hello, Mark! Today is Wednesday, it's 19:40 now.
```

从 C# 10 开始，可以使用内插字符串来初始化常量字符串。 仅当内插字符串中的所有内插表达式也是常量字符串时，才可以执行此操作。

内插字符串的结构

若要将字符串标识为内插字符串，可在该字符串前面加上 `$` 符号。 字符串字面量开头的 `$` 和 `"` 之间不能有任何空格。

具备内插表达式的项的结构如下所示：

C#

```
{<interpolationExpression>[,<alignment>][:<formatString>]}
```

括号中的元素是可选的。 下表说明了每个元素：

展开表

元素	描述
`interpolationExpression`	生成需要设置格式的结果的表达式。 当表达式为 `null` 时，输出为空字符串 (String.Empty)。
`alignment`	常数表达式，它的值定义表达式结果的字符串表示形式中的最小字符数。 如果值为正，则字符串表示形式为右对齐；如果值为负，则为左对齐。 有关详细信息，请参阅复合格式设置一文的对齐组件部分。
`formatString`	受表达式结果类型支持的格式字符串。 有关详细信息，请参阅复合格式设置一文的格式字符串组件部分。

以下示例使用上表中所描述的可选格式设置组件：

C#

```
Console.WriteLine($"|{"Left",-7}|{"Right",7}|");

const int FieldWidthRightAligned = 20;
Console.WriteLine($"{Math.PI,FieldWidthRightAligned} - default formatting of
the pi number");
Console.WriteLine($"{Math.PI,FieldWidthRightAligned:F3} - display only three
decimal digits of the pi number");
// Output is:
// |Left    |  Right|
//     3.14159265358979 - default formatting of the pi number
//                3.142 - display only three decimal digits of the pi number
```

从 C# 11 开始，可以在内插表达式中使用换行符，以使表达式的代码更具可读性。 下面的示例展示了换行符如何提高涉及模式匹配的表达式的可读性：

C#

```
string message = $"The usage policy for {safetyScore} is {
    safetyScore switch
    {
        > 90 => "Unlimited usage",
        > 80 => "General usage, with daily safety check",
        > 70 => "Issues must be addressed within 1 week",
        > 50 => "Issues must be addressed within 1 day",
        _ => "Issues must be addressed before continued use",
    }
    }";
```

内插原始字符串字面量

从 C# 11 开始，可以使用内插原始字符串字面量，如以下示例所示：

C#

```
int X = 2;
int Y = 3;

var pointMessage = $"""The point "{X}, {Y}" is {Math.Sqrt(X * X + Y * Y):F3}
from the origin""";

Console.WriteLine(pointMessage);
// Output is:
// The point "2, 3" is 3.606 from the origin
```

要在结果字符串中嵌入 `{` 和 `}` 字符，请让内插原始字符串字面量以多个 `$` 字符开头。 执行此操作时，任何长度短于 `$` 字符数的 `{` 或 `}` 字符序列都会嵌入到结果字符串中。 若要将任何内插表达式包含在该字符串中，需要使用与 `$` 字符数相同的大括号数，如以下示例所示：

C#

```
int X = 2;
int Y = 3;

var pointMessage = $$"""{The point {{{X}}, {{Y}}} is {{Math.Sqrt(X * X + Y *
Y):F3}} from the origin}""";
Console.WriteLine(pointMessage);
// Output is:
// {The point {2, 3} is 3.606 from the origin}
```

在前面的示例中，内插原始字符串字面量以两个 `$` 字符开头。 你需要将每个内插表达式放在双大括号（`{{` 和 `}}`）之间。 单个大括号嵌入到结果字符串中。 如果需要将重复的 `{` 或 `}` 字符嵌入结果字符串中，请使用相应增加的 `$` 字符数来指定内插原始字符串字面量。 如果字符串文本的重复大括号数多于 `$` 字符数，则 `{` 和 `}`字符将从内到外分组。 在前面的示例中，文本 `The point {{{X}}, {{Y}}}` 将 `{{X}}` 和 `{{Y}}` 解释为内插表达式。 输出字符串中逐字包含外层的 `{` 和 `}`。

特殊字符

要在内插字符串生成的文本中包含大括号 "{" 或 "}"，请使用两个大括号，即 "{{" 或 "}}"。 有关详细信息，请参阅复合格式设置一文的转义括号部分。

因为冒号（":"）在内插表达式项中具有特殊含义，为了在内插表达式中使用条件运算符，请将该表达式放在括号内。

以下示例演示了如何在结果字符串中包括大括号。 它还演示了如何使用条件运算符：

C#

```
string name = "Horace";
int age = 34;
Console.WriteLine($"He asked, \"Is your name {name}?\", but didn't wait for
a reply :-{{");
Console.WriteLine($"{name} is {age} year{(age == 1 ? "" : "s")} old.");
// Output is:
// He asked, "Is your name Horace?", but didn't wait for a reply :-{
// Horace is 34 years old.
```

内插逐字字符串以 `$` 和 `@` 字符开头。 可以按任意顺序使用 `$` 和 `@`：`$@"..."` 和 `@$"..."` 均为有效的内插逐字字符串。 有关逐字字符串的详细信息，请参阅字符串和逐字标识符文章。

特定于区域性的格式设置

默认情况下，内插字符串将 CultureInfo.CurrentCulture 属性定义的当前区域性用于所有格式设置操作。

要将内插字符串解析为特定于区域性的结果字符串，请使用 String.Create(IFormatProvider, DefaultInterpolatedStringHandler) 方法，该方法从 .NET 6 开始可用。 下面的示例演示如何执行此操作：

C#

```
double speedOfLight = 299792.458;

System.Globalization.CultureInfo.CurrentCulture =
System.Globalization.CultureInfo.GetCultureInfo("nl-NL");
string messageInCurrentCulture = $"The speed of light is {speedOfLight:N3}
km/s.";

var specificCulture = System.Globalization.CultureInfo.GetCultureInfo("en-
IN");
string messageInSpecificCulture = string.Create(
    specificCulture, $"The speed of light is {speedOfLight:N3} km/s.");

string messageInInvariantCulture = string.Create(
    System.Globalization.CultureInfo.InvariantCulture, $"The speed of light
is {speedOfLight:N3} km/s.");

Console.WriteLine($"{System.Globalization.CultureInfo.CurrentCulture,-10}
{messageInCurrentCulture}");
Console.WriteLine($"{specificCulture,-10} {messageInSpecificCulture}");
Console.WriteLine($"{"Invariant",-10} {messageInInvariantCulture}");
// Output is:
// nl-NL      The speed of light is 299.792,458 km/s.
```

```
// en-IN      The speed of light is 2,99,792.458 km/s.
// Invariant  The speed of light is 299,792.458 km/s.
```

在 .NET 5 和更早的 .NET 版本中，请使用从内插字符串到 FormattableString 实例的隐式转换。 然后，可以使用实例 FormattableString.ToString(IFormatProvider) 方法或静态 FormattableString.Invariant 方法来生成特定于区域性的结果字符串。 下面的示例演示如何执行此操作：

C#

```
double speedOfLight = 299792.458;
FormattableString message = $"The speed of light is {speedOfLight:N3}
km/s.";

var specificCulture = System.Globalization.CultureInfo.GetCultureInfo("en-
IN");
string messageInSpecificCulture = message.ToString(specificCulture);
Console.WriteLine(messageInSpecificCulture);
// Output:
// The speed of light is 2,99,792.458 km/s.

string messageInInvariantCulture = FormattableString.Invariant(message);
Console.WriteLine(messageInInvariantCulture);
// Output is:
// The speed of light is 299,792.458 km/s.
```

有关自定义格式设置的详细信息，请参阅在 .NET 中设置类型格式一文中的使用 ICustomFormatter 进行自定义格式设置部分。

其他资源

如果你不熟悉字符串内插，请参阅 C# 中的字符串内插交互式教程。 还可查看另一个 C# 中的字符串内插教程。 该教程演示了如何使用内插字符串生成带格式的字符串。

内插字符串编译

从 C# 10 和 .NET 6 开始，编译器会检查内插字符串是否被分配给满足内插字符串处理程序模式要求的类型。 内插字符串处理程序是一种将内插字符串转换为结果字符串的类型。 当内插字符串的类型为 `string` 时，它由 System.Runtime.CompilerServices.DefaultInterpolatedStringHandler 处理。 有关自定义内插字符串处理程序的示例，请参阅编写自定义字符串内插处理程序教程。 使用内插字符串处理程序是一种高级方案，通常出于性能原因而需要使用。

> **备注**

内插字符串处理程序的一个副作用是，自定义处理程序（包括 System.Runtime.CompilerServices.DefaultInterpolatedStringHandler）可能不会在所有条件下都计算内插字符串中的所有内插表达式。 这意味着这些表达式的副作用可能不会发生。

在 C# 10 之前，如果内插字符串类型为 `string`，则通常将其转换为 String.Format 方法调用。 如果分析的行为等同于串联，则编译器可将 String.Format 替换为 String.Concat。

如果内插字符串类型为 IFormattable 或 FormattableString，则编译器会生成对 FormattableStringFactory.Create 方法的调用。

C# 语言规范

有关详细信息，请参阅 C# 语言规范的内插字符串表达式部分以及以下新功能规范：

- C# 10 - 改进的内插字符串
- C# 11 - 原始字符串字面量
- C# 11 - 字符串内插中的换行符

另请参阅

- C# 特殊字符
- 字符串
- 标准数字格式字符串
- 复合格式设置
- String.Format
- 简化内插（样式规则 IDE0071）
- C# 10 和 .NET 6 中的字符串内插（.NET 博客）

在 GitHub 上与我们协作

可以在 GitHub 上找到此内容的源，还可以在其中创建和查看问题和拉取请求。 有关详细信息，请参阅参与者指南。

.NET

.NET 反馈

.NET 是一个开放源代码项目。 选择一个链接以提供反馈：

提出文档问题

提供产品反馈

逐字文本 - `@` 在变量、属性和字符串文本中

项目 • 2024/04/01

`@` 特殊字符用作原义标识符。 通过以下方式使用它：

1. 指示将原义解释字符串。 `@` 字符在此实例中定义原义标识符 。 简单转义序列（如代表反斜杠的 `"\\"`）、十六进制转义序列（如代表大写字母 A 的 `"\x0041"`）和 Unicode 转义序列（如代表大写字母 A 的 `"\u0041"`）都将按字面解释。 只有引号转义序列 (`""`) 不会按字面解释；因为它生成一个双引号。 此外，如果是逐字内插字符串，大括号转义序列（`{{` 和 `}}`）不按字面解释；它们会生成单个大括号字符。 下面的示例分别使用常规字符串和原义字符串定义两个相同的文件路径。 这是原义字符串的较常见用法之一。

   C#

   ```
   string filename1 = @"c:\documents\files\u0066.txt";
   string filename2 = "c:\\documents\\files\\u0066.txt";

   Console.WriteLine(filename1);
   Console.WriteLine(filename2);
   // The example displays the following output:
   //     c:\documents\files\u0066.txt
   //     c:\documents\files\u0066.txt
   ```

   下面的示例演示定义包含相同字符序列的常规字符串和原义字符串的效果。

   C#

   ```
   string s1 = "He said, \"This is the last \u0063hance\x0021\"";
   string s2 = @"He said, ""This is the last \u0063hance\x0021""";

   Console.WriteLine(s1);
   Console.WriteLine(s2);
   // The example displays the following output:
   //     He said, "This is the last chance!"
   //     He said, "This is the last \u0063hance\x0021"
   ```

2. 使用 C# 关键字作为标识符。 `@` 字符可作为代码元素的前缀，编译器将把此代码元素解释为标识符而非 C# 关键字。 下面的示例使用 `@` 字符定义其在 `for` 循环中使用的名为 `for` 的标识符。

   C#

```
string[] @for = { "John", "James", "Joan", "Jamie" };
for (int ctr = 0; ctr < @for.Length; ctr++)
{
   Console.WriteLine($"Here is your gift, {@for[ctr]}!");
}
// The example displays the following output:
//     Here is your gift, John!
//     Here is your gift, James!
//     Here is your gift, Joan!
//     Here is your gift, Jamie!
```

3. 使编译器在命名冲突的情况下区分两种属性。 属性是派生自 Attribute 的类。 其类型名称通常包含后缀 Attribute，但编译器不会强制进行此转换。 随后可在代码中按其完整类型名称（例如 `[InfoAttribute]`）或短名称（例如 `[Info]`）引用此属性。 但是，如果两个短名称相同，并且一个类型名称包含 Attribute 后缀而另一类型名称不包含，则会出现命名冲突。 例如，由于编译器无法确定将 `Info` 还是 `InfoAttribute` 属性应用于 `Example` 类，因此下面的代码无法编译。 有关详细信息，请参阅 CS1614。

C#

```
using System;

[AttributeUsage(AttributeTargets.Class)]
public class Info : Attribute
{
   private string information;

   public Info(string info)
   {
      information = info;
   }
}

[AttributeUsage(AttributeTargets.Method)]
public class InfoAttribute : Attribute
{
   private string information;

   public InfoAttribute(string info)
   {
      information = info;
   }
}

[Info("A simple executable.")] // Generates compiler error CS1614.
Ambiguous Info and InfoAttribute.
// Prepend '@' to select 'Info' ([@Info("A simple executable.")]).
Specify the full name 'InfoAttribute' to select it.
public class Example
```

```
{
   [InfoAttribute("The entry point.")]
   public static void Main()
   {
   }
}
```

另请参阅

- C# 特殊字符

在 GitHub 上与我们协作

可以在 GitHub 上找到此内容的源，还可以在其中创建和查看问题和拉取请求。 有关详细信息，请参阅参与者指南。

.NET 反馈

.NET 是一个开放源代码项目。 选择一个链接以提供反馈：

提出文档问题

提供产品反馈

原始字符串字面量 - 字符串字面量中的 """

项目 • 2024/07/19

原始字符串字面量以至少三个双引号 (`"`) 字符开始和结束：

C#

```
var singleLine = """This is a "raw string literal". It can contain
characters like \, ' and ".""";
```

原始字符串字面量可以包含多行：

C#

```
var xml = """
        <element attr="content">
            <body>
            </body>
        </element>
        """;
```

以下规则控制多行原始字符串字面量的解释：

- 左引号必须是其各自行上的最后一个非注释标记，右引号必须是其各自行上的第一个非注释标记。
- 右引号左侧的任何空格将从原始字符串字面量的所有行中删除。
- 将忽略同一行中左引号后面的空格。
- 字符串字面量中仅包含左引号后面的空格行。
- 如果在同一行的结束分隔符之前有空白字符（例如空格与制表符），则每行内容的开头必须有确切数量和类型的空白字符。 具体而言，空格与水平制表符不匹配，反之亦然。

可能需要创建包含三个或更多连续双引号字符的原始字符串字面量。 原始字符串字面量可以至少三个双引号字符的序列开始和结束。 如果字符串字面量包含三个连续双引号，则以四个双引号字符开始和结束原始字符串字面量：

C#

```
var moreQuotes = """" As you can see,"""Raw string literals""" can start and
end with more than three double-quotes when needed."""";
```

如果需要以引号字符开始或结束原始字符串字面量，请使用多行格式：

```
C#
```

```
var MultiLineQuotes = """
                """Raw string literals""" can start and end with more than
three double-quotes when needed.
                """;
```

原始字符串字面量还可以与内插字符串结合使用，以在输出字符串中嵌入 `{` 和 `}` 字符。 在内插的原始字符串字面量中使用多个 `$` 字符，以在输出字符串中嵌入 `{` 和 `}` 字符，而无需对这些字符进行转义。

原始字符串字面量的内容不得包含一组长度等于或大于原始字符串字面量分隔符长度的连续 `"` 字符。 例如，字符串 `""" """ """` 和 `""""""" """""" """""" """" """ """""""` 的格式就很标准。 但是，字符串 `""" """ """` 和 `""" """" """` 的格式就不正确

原始字符串字面量是在 C# 11 中引入的。

另请参阅

- C# 特殊字符
- C# 字符串内插
- 原始字符串字面量特征规范

在 GitHub 上与我们协作

可以在 GitHub 上找到此内容的源，还可以在其中创建和查看问题和拉取请求。 有关详细信息，请参阅参与者指南。

.NET

.NET 反馈

.NET 是一个开放源代码项目。 选择一个链接以提供反馈：

提出文档问题

提供产品反馈

C# 编译器解释的程序集级别属性

项目 • 2024/07/31

大多数特性应用于特定语言元素，如类或方法；但是，一些特性是全局特性 - 它们应用于整个程序集或模块。 例如，AssemblyVersionAttribute 属性可用于将版本信息嵌入程序集，如下所示：

C#

```
[assembly: AssemblyVersion("1.0.0.0")]
```

全局特性出现在源代码中任何顶级 `using` 指令之后和任何类型、模块或命名空间声明之前。 全局特性可以出现在多个源文件中，但必须在单个编译过程中编译这些文件。 Visual Studio 将全局特性添加到 .NET Framework 项目中的 AssemblyInfo.cs 文件中。 这些特性不会添加到 .NET Core 项目中。

程序集特性是提供程序集相关信息的值。 它们分为以下几类：

- 程序集标识特性
- 信息性特性
- 程序集清单特性

程序集标识特性

三个特性（与强名称（如果适用））组合起来可以确定程序集的标识：名称、版本和区域性。 这些特性构成程序集的全名，在代码中引用程序集时必需使用。 可使用特性设置程序集的版本和区域性。 但是，创建程序集时，由编译器、程序集信息对话框中的 Visual Studio IDE 或程序集链接器 (Al.exe) 设置名称值。 程序集名称基于程序集清单。 AssemblyFlagsAttribute 属性指定程序集的多个副本是否可以共存。

下表显示标识特性。

展开表

特性	目标
AssemblyVersionAttribute	指定程序集的版本。
AssemblyCultureAttribute	指定程序集支持的区域性。
AssemblyFlagsAttribute	指定描述实时 (JIT) 编译器选项的程序集的标志的按位组合、该程序集是否可重定目标，以及是否具有完整或已标记化的公共密钥。

信息性特性

你可使用信息性特性为程序集提供更多公司或产品信息。 下表显示 System.Reflection 命名空间中定义的信息性属性。

展开表

特性	目标
AssemblyProductAttribute	指定程序集清单的产品名称。
AssemblyTrademarkAttribute	指定程序集清单的商标。
AssemblyInformationalVersionAttribute	为程序集清单指定信息性版本。
AssemblyCompanyAttribute	为程序集清单指定公司名称。
AssemblyCopyrightAttribute	定义为程序集清单指定版权的自定义属性。
AssemblyFileVersionAttribute	设置 Win32 文件版本资源的特定版本号。
CLSCompliantAttribute	指示程序集是否符合公共语言规范 (CLS)。

程序集清单特性

程序集清单特性可用于提供程序集清单中的信息。 特性包括标题、说明、默认别名和配置。 下表显示 System.Reflection 命名空间中定义的程序集清单属性。

展开表

特性	目标
AssemblyTitleAttribute	为程序集清单指定程序集标题。
AssemblyDescriptionAttribute	为程序集清单指定程序集说明。
AssemblyConfigurationAttribute	为程序集清单指定程序集配置（如零售或调试）。
AssemblyDefaultAliasAttribute	定义程序集清单的友好默认别名

在 GitHub 上与我们协作

可以在 GitHub 上找到此内容的源，还可以在其中创建和查看问

.NET

.NET 反馈

.NET 是一个开放源代码项目。 选择一个链接以提供反馈：

题和拉取请求。 有关详细信息，请参阅参与者指南。

提出文档问题

提供产品反馈

使用 C# 编译器解释的属性确定调用方信息

项目 • 2023/04/07

使用信息属性，可以获取有关方法调用方的信息。 可以获取源代码的文件路径、源代码中的行号和调用方的成员名称。 若要获取成员调用方信息，可以使用应用于可选参数的特性。 每个可选参数指定一个默认值。 下表列出在 System.Runtime.CompilerServices 命名空间中定义的调用方信息特性：

展开表

特性	说明	类型
CallerFilePathAttribute	包含调用方的源文件的完整路径。 完整路径是编译时的路径。	`String`
CallerLineNumberAttribute	源文件中调用方法的行号。	`Integer`
CallerMemberNameAttribute	调用方的方法名称或属性名称。	`String`
CallerArgumentExpressionAttribute	参数表达式的字符串表示形式。	`String`

此信息可帮助你进行跟踪和调试，以及创建诊断工具。 下面的示例演示如何使用调用方信息特性。 每次调用 `TraceMessage` 方法时，都会将调用方信息插入到可选参数的变量中。

C#

```
public void DoProcessing()
{
    TraceMessage("Something happened.");
}

public void TraceMessage(string message,
        [CallerMemberName] string memberName = "",
        [CallerFilePath] string sourceFilePath = "",
        [CallerLineNumber] int sourceLineNumber = 0)
{
    Trace.WriteLine("message: " + message);
    Trace.WriteLine("member name: " + memberName);
    Trace.WriteLine("source file path: " + sourceFilePath);
    Trace.WriteLine("source line number: " + sourceLineNumber);
}

// Sample Output:
//  message: Something happened.
//  member name: DoProcessing
```

```
//  source file path: c:\Visual Studio
Projects\CallerInfoCS\CallerInfoCS\Form1.cs
//  source line number: 31
```

为每个可选参数指定显式默认值。 不能将调用方信息特性应用于未指定为可选的参数。 调用方信息特性不会使参数成为可选参数。 相反，它们会在忽略此参数时影响传入的默认值。 在编译时，调用方信息值将作为文本传入中间语言 (IL)。 与异常的 StackTrace 属性的结果不同，这些结果不受模糊处理的影响。 你可显式提供可选参数来控制调用方信息或隐藏调用方信息。

成员名称

可以使用 `CallerMemberName` 特性来避免将成员名称指定为所调用的方法的 `String` 参数。 通过使用这种技术，可以避免“重命名重构”不更改 值的问题。 此好处对于以下任务特别有用：

- 使用跟踪和诊断例程。
- 在绑定数据时实现 INotifyPropertyChanged 接口。 此接口允许对象的属性通知绑定控件该属性已更改。 控件可以显示更新后的信息。 如果没有 `CallerMemberName` 特性，则必须将属性名称指定为文本。

以下图表显示在使用 `CallerMemberName` 特性时返回的成员名称。

展开表

调用发生中	成员名称结果
方法、属性或事件	从中发起调用的方法、属性或事件的名称。
构造函数	字符串“.ctor”
静态构造函数	字符串“.cctor”
终结器	字符串“Finalize”
用户定义的运算符或转换	为成员生成的名称，例如，“op_Addition”。
特性构造函数	要应用特性的方法或属性的名称。 如果该特性是成员中的任何元素（如参数、返回值或泛型参数），则此结果是与该元素关联的成员的名称。
无包含的成员（例如，程序集级别或应用于类型的特性）	可选参数的默认值。

参数表达式

如果想要将表达式作为参数传递，请使用 System.Runtime.CompilerServices.CallerArgumentExpressionAttribute。 诊断库可能需要提供有关传递给参数的表达式的更多详细信息。 通过提供触发诊断的表达式以及参数名称，开发人员可以更加详细地了解触发诊断的条件。 这些额外的信息使修复变得更容易。

以下示例显示如何在参数无效时提供有关参数的详细信息：

C#

```
public static void ValidateArgument(string parameterName, bool condition,
[CallerArgumentExpression("condition")] string? message=null)
{
    if (!condition)
    {
        throw new ArgumentException($"Argument failed validation:
<{message}>", parameterName);
    }
}
```

可以调用参数，如以下示例所示：

C#

```
public void Operation(Action func)
{
    Utilities.ValidateArgument(nameof(func), func is not null);
    func();
}
```

用于 `condition` 的表达式由编译器注入到 `message` 参数中。 当开发者使用 `null` 参数调用 `Operation` 时，以下消息将存储在 `ArgumentException` 中：

text

```
Argument failed validation: <func is not null>
```

此属性让你能够编写可提供更多详细信息的诊断实用工具。 开发人员可以更快地了解需要进行哪些更改。 也可使用 CallerArgumentExpressionAttribute 来确定哪个表达式用作扩展方法的接收方。 以下方法定期对序列进行采样。 如果序列中包含的元素少于频率，则会报错：

C#

```
public static IEnumerable<T> Sample<T>(this IEnumerable<T> sequence, int
frequency,
    [CallerArgumentExpression(nameof(sequence))] string? message = null)
{
    if (sequence.Count() < frequency)
        throw new ArgumentException($"Expression doesn't have enough
elements: {message}", nameof(sequence));
    int i = 0;
    foreach (T item in sequence)
    {
        if (i++ % frequency == 0)
            yield return item;
    }
}
```

上一个示例为 `sequence` 参数使用 nameof 运算符。 此功能在 C# 11 中提供。 在 C# 11 之前，需要将参数的名称作为为字符串键入。 可以按如下所示调用此方法：

C#

```
sample = Enumerable.Range(0, 10).Sample(100);
```

前面的示例将引发一个 ArgumentException，其消息如以下文本所示：

text

```
Expression doesn't have enough elements: Enumerable.Range(0, 10) (Parameter
'sequence')
```

请参阅

- 命名参数和可选参数
- System.Reflection
- Attribute
- 特性

C# 编译器解释的 null 状态静态分析的属性

项目 • 2023/06/05

在启用可为 null 的上下文中，编译器对代码执行静态分析，以确定所有引用类型变量的 null 状态：

- null 状态：静态分析确定变量具有非 null 值。
- 可能为 null：静态分析无法确定是否为变量分配了非 null 值。

这些状态使编译器能够在你可能取消引用 null 值并引发 System.NullReferenceException 时提供警告。 这些属性根据参数和返回值的状态为编译器提供有关参数、返回值和对象成员的 null 状态的语义信息。 当 API 已正确注释此语义信息时，编译器会提供更准确的警告。

本文提供每个可为 null 引用类型特性的简要说明以及它们的使用方法。

我们从一个示例开始。 假设你的库具有以下用于检索资源字符串的 API。 此方法最初是在忽略可为 null 的上下文中编译的：

C#

```
bool TryGetMessage(string key, out string message)
{
    if (_messageMap.ContainsKey(key))
        message = _messageMap[key];
    else
        message = null;
    return message != null;
}
```

前面的示例遵循 .NET 中熟悉的 `Try*` 模式。 此 API 有两个引用参数：`key` 和 `message`。 此 API 具有与这些参数的 null 状态相关的以下规则：

- 调用方不应将 `null` 作为 `key` 的参数传递。
- 调用方可以传递值为 `null` 的变量作为 `message` 的参数。
- 如果 `TryGetMessage` 方法返回 `true`，则 `message` 的值不为 null。 如果返回值是 `false`，则 `message` 的值为 null。

`key` 的规则可以简洁地表示：`key` 应是不可为 null 的引用类型。 `message` 参数更复杂。 它允许将 `null` 变量作为参数，但保证成功时 `out` 参数不是 `null`。 对于这些情况，需要使用更丰富的词汇来描述期望。 下文描述的 `NotNullWhen` 属性描述了用于 `message` 参数的自变量的 null 状态。

> **备注**
>
> 添加这些特性将为编译器提供有关 API 规则的更多信息。 当调用代码在可为 null 的上下文中编译时，编译器将在调用方违反这些规则时发出警告。 这些特性不会启用对实现进行更多检查。

Attribute	Category	含义
AllowNull	Precondition	不可为 null 的参数、字段或属性可以为 null。
DisallowNull	Precondition	可为 null 的参数、字段或属性应永不为 null。
MaybeNull	后置条件	不可为 null 的参数、字段、属性或返回值可能为 null。
NotNull	后置条件	可为 null 的参数、字段、属性或返回值将永不为 null。
MaybeNullWhen	有条件后置条件	当方法返回指定的 `bool` 值时，不可为 null 的参数可以为 null。
NotNullWhen	有条件后置条件	当方法返回指定的 `bool` 值时，可以为 null 的参数不会为 null。
NotNullIfNotNull	有条件后置条件	如果指定参数的自变量不为 null，则返回值、属性或自变量不为 null。
MemberNotNull	方法和属性帮助程序方法	当方法返回时，列出的成员不会为 null。
MemberNotNullWhen	方法和属性帮助程序方法	当方法返回指定的 `bool` 值时，列出的成员不会为 null。
DoesNotReturn	无法访问的代码	方法或属性永远不会返回。 换句话说，它总是引发异常。
DoesNotReturnIf	无法访问的代码	如果关联的 `bool` 参数具有指定值，则此方法或属性永远不会返回。

上述说明是对每个特性的快速参考。 以下各节更详尽地描述了这些属性的行为和含义。

前置条件：`AllowNull` 和 `DisallowNull`

请考虑一个从不返回 `null` 的读/写属性，因为它具有合理的默认值。 调用方在将 `null` 设置为该默认值时将其传递给 set 访问器。 例如，假设一个消息系统要求在聊天室中输入一个屏幕名称。 如果未提供任何内容，系统将生成一个随机名称：

C#

```
public string ScreenName
{
    get => _screenName;
    set => _screenName = value ?? GenerateRandomScreenName();
}
private string _screenName;
```

当你在忽略可为 null 的上下文中编译前面的代码时，一切都是正常的。 启用可为 null 的引用类型后，`ScreenName` 属性将成为不可为 null 的引用。 这对于 `get` 访问器是正确的：它从不返回 `null`。 调用方不需要检查返回的 `null` 属性。 但现在将属性设置为 `null` 将生成警告。 若要支持这种类型的代码，请向属性添加 System.Diagnostics.CodeAnalysis.AllowNullAttribute 特性，如下面的代码所示：

C#

```
[AllowNull]
public string ScreenName
{
    get => _screenName;
    set => _screenName = value ?? GenerateRandomScreenName();
}
private string _screenName = GenerateRandomScreenName();
```

可能需要为 System.Diagnostics.CodeAnalysis 添加一个 `using` 指令才能使用本文中讨论的特性和其他特性。 特性应用于属性，而不是 `set` 访问器。 `AllowNull` 特性指定前置条件，并且仅适用于参数。 `get` 访问器有一个返回值，但没有参数。 因此，`AllowNull` 特性只适用于 `set` 访问器。

前面的示例演示了在参数上添加 `AllowNull` 特性时要查找的内容：

1. 该变量的一般约定是它不应为 `null`，因此需要一个不可为 null 的引用类型。
2. 在某些情况下，调用方将 `null` 作为参数传递，但它们不是最常见的用法。

大多数情况下，属性或 `in`、`out` 和 `ref` 参数需要此特性。 当变量通常为非 null 时，`AllowNull` 属性是最佳选择，但需要允许 `null` 作为前提条件。

将此与使用 `DisallowNull` 的方案相比：使用此特性可以指定可为 null 引用类型的参数不应为 `null`。 请考虑一个特性，其中 `null` 是默认值，但客户端只能将其设置为非 null 值。 考虑下列代码：

C#

```
public string ReviewComment
{
    get => _comment;
    set => _comment = value ?? throw new
ArgumentNullException(nameof(value), "Cannot set to null");
}
string _comment;
```

前面的代码是表达设计的最佳方式，`ReviewComment` 可以是 `null`，但不能设置为 `null`。 代码可识别为 null 后，你就可以使用 System.Diagnostics.CodeAnalysis.DisallowNullAttribute 向调用方更清楚地表达此概念：

C#

```
[DisallowNull]
public string? ReviewComment
{
    get => _comment;
    set => _comment = value ?? throw new
ArgumentNullException(nameof(value), "Cannot set to null");
}
string? _comment;
```

在可为 null 的上下文中，`ReviewComment` `get` 访问器可以返回默认值 `null`。 编译器会警告在访问之前必须进行检查。 此外，它警告调用方，即使它可能是 `null`，调用方也不应显式地将其设置为 `null`。 `DisallowNull` 特性还指定了前置条件，它不影响 `get` 访问器。 当你观察到以下特征时，可以使用 `DisallowNull` 特性：

1. 在核心方案中（通常是在首次实例化时），变量可以是 `null`。
2. 变量不应显式设置为 `null`。

这些情况在忽略 null 的代码中很常见 。 可能是在两个不同的初始化操作中设置了对象属性。 可能只有在某些异步工作完成后才能设置某些属性。

可使用 `AllowNull` 和 `DisallowNull` 特性指定变量上的前置条件可能与这些变量上的可为 null 注释不匹配。 这两个特性提供了关于 API 特征的更多细节。 此附加信息有助于调用方正确使用 API。 请记住，可使用以下特性指定前提条件：

- AllowNull：不可为 null 的参数可以为 null。
- DisallowNull：可为 null 的参数不应为 null。

后置条件：`MaybeNull` 和 `NotNull`

假设你有使用以下签名的方法：

C#

```
public Customer FindCustomer(string lastName, string firstName)
```

你可能已经编写了类似的方法，以便在未找到所查找的名称时返回 `null`。 `null` 清楚地表明未找到记录。 在本例中，你可能会将返回类型从 `Customer` 更改为 `Customer?`。 将返回值声明为可为 null 的引用类型可以清楚地指定此 API 的意图：

C#

```
public Customer? FindCustomer(string lastName, string firstName)
```

由于泛型为 null 性中所述的原因，该技术可能无法生成与 API 匹配的静态分析。 你可能具有遵循类似模式的泛型方法：

C#

```
public T Find<T>(IEnumerable<T> sequence, Func<T, bool> predicate)
```

当找不到所需项时，此方法返回 `null`。 可以通过将 `MaybeNull` 注释添加到方法返回来阐明该方法在未找到项目时返回 `null`：

C#

```
[return: MaybeNull]
public T Find<T>(IEnumerable<T> sequence, Func<T, bool> predicate)
```

前面的代码通知调用方返回值可能实际上为 null。 它还通知编译器该方法可能会返回一个 `null` 表达式，即使该类型不可为 null。 当你有一个返回其类型参数 `T` 的实例的泛型方法时，你可以使用 `NotNull` 属性表示它永远不会返回 `null`。

还可以指定返回值或者参数不为 null，即使该类型是可为 null 的引用类型。 以下方法是一种帮助器方法，如果其第一个参数为 `null`，则引发该方法：

C#

```
public static void ThrowWhenNull(object value, string valueExpression = "")
{
    if (value is null) throw new ArgumentNullException(nameof(value),
valueExpression);
}
```

可以按如下方式调用此例程：

C#

```
public static void LogMessage(string? message)
{
    ThrowWhenNull(message, $"{nameof(message)} must not be null");

    Console.WriteLine(message.Length);
}
```

启用 null 引用类型后，需要确保前面的代码在编译时没有警告。 当方法返回时，`value` 参数保证不为 null。 但是，可以使用 null 引用调用 `ThrowWhenNull`。 可以将 `value` 设为可为 null 的引用类型，并将 `NotNull` 后置条件添加到参数声明中：

C#

```
public static void ThrowWhenNull([NotNull] object? value, string
valueExpression = "")
{
    _ = value ?? throw new ArgumentNullException(nameof(value),
valueExpression);
    // other logic elided
```

前面的代码清楚地表达了现有协定：调用方可以传递具有 `null` 值的变量，但如果该方法在未引发异常的情况下返回，则保证该参数永远不为 null。

可以使用以下特性指定无条件后置条件：

- MaybeNull：不可为 null 的返回值可以为 null。
- NotNull：可为 null 的返回值永远不会为 null。

有条件后置条件：`NotNullWhen`、`MaybeNullWhen` 和 `NotNullIfNotNull`

你可能很熟悉 `string` 方法 String.IsNullOrEmpty(String)。 当参数为 null 或为空字符串时，此方法返回 `true`。 这是一种 null 检查格式：如果方法返回 `false`，调用方不需要 null 检查参数。 若要使这样的方法可识别为 null，需要将参数设置为可为 null 的引用类型，并添加 `NotNullWhen` 特性：

C#

```
bool IsNullOrEmpty([NotNullWhen(false)] string? value)
```

这通知编译器，任何返回值为 `false` 的代码都不需要 null 检查。 添加特性通知编译器的静态分析，`IsNullOrEmpty` 执行必要的 null 检查：当它返回 `false` 时，参数不是 `null`。

C#

```
string? userInput = GetUserInput();
if (!string.IsNullOrEmpty(userInput))
{
    int messageLength = userInput.Length; // no null check needed.
}
// null check needed on userInput here.
```

> ⓘ **备注**
>
> 前面的示例仅适用于 C# 11 及更高版本。 从 C# 11 开始，`[nameof` expression](../operators/nameof.md) 可在用于应用于方法的属性时引用参数和类型参数名称。 在 C# 10 及更早版本中，需要使用字符串字面量而不是 `nameof` 表达式。

对于 .NET Core 3.0，将对 String.IsNullOrEmpty(String) 方法进行注释，如上面所示。 代码库中可能有类似的方法来检查对象的状态是否为 null 值。 编译器无法识别自定义的 null 检查方法，你需要自己添加注释。 添加特性时，编译器的静态分析将知道何时对测试变量进行了 null 检查。

这些特性的另一个用途是 `Try*` 模式。 `ref` 和 `out` 参数的后置条件通过返回值进行通信。 考虑前面显示的这个方法（在一个禁用可为 null 的上下文中）：

C#

```
bool TryGetMessage(string key, out string message)
{
    if (_messageMap.ContainsKey(key))
        message = _messageMap[key];
    else
        message = null;
    return message != null;
}
```

前面的方法遵循典型的 .NET 习惯用法：返回值指示是否将 `message` 设置为已找到的值，或者，如果未找到消息，则为默认值。 如果方法返回 `true`，`message` 的值不为 null；否则，该方法将 `message` 设置为 null。

在启用可为 null 的上下文中，可以使用 `NotNullWhen` 属性传达该习惯用法。 注释可为 null 引用类型的参数时，将 `message` 设为 `string?` 并添加属性：

C#

```
bool TryGetMessage(string key, [NotNullWhen(true)] out string? message)
{
    if (_messageMap.ContainsKey(key))
        message = _messageMap[key];
    else
        message = null;
    return message is not null;
}
```

在前面的示例中，`message` 的值在 `TryGetMessage` 返回 `true` 时不为 null。 你应以相同的方式在代码库中注释类似的方法：参数可以等于 `null`，并且已知在方法返回 `true` 时不为 null。

还可能需要一个最终特性。 有时，返回值的 null 状态取决于一个或多个参数的 null 状态。 只要某些参数不是 `null`，这些方法将返回非 null 值。 若要正确地注释这些方法，可以使用 `NotNullIfNotNull` 特性。 请考虑以下方法：

C#

```
string GetTopLevelDomainFromFullUrl(string url)
```

如果 `url` 参数不为 null，则输出不是 `null`。 启用可为 null 引用后，如果 API 可以接受 null 参数，则需要添加更多注释。 可以注释返回类型，如以下代码所示：

C#

```
string? GetTopLevelDomainFromFullUrl(string? url)
```

这也是可行的，但通常会强制调用方实现额外的 `null` 检查。 协定是，只有当参数 `url` 是 `null` 时，返回值才会是 `null`。 若要表达该协定，你需要注释此方法，如以下代码所示：

C#

```
[return: NotNullIfNotNull(nameof(url))]
string? GetTopLevelDomainFromFullUrl(string? url)
```

上一个示例为 `url` 参数使用 nameof 运算符。 此功能在 C# 11 中提供。 在 C# 11 之前，需要将参数的名称作为为字符串键入。 返回值和参数都用 `?` 进行了注释，这表明两者都可以是 `null`。 该特性进一步阐明了当 `url` 参数不是 `null` 时，返回值不会为 null。

可以使用以下特性指定条件的后置条件：

- MaybeNullWhen：当方法返回指定的 `bool` 值时，不可为 null 的参数可以为 null。
- NotNullWhen：当方法返回指定的 `bool` 值时，可以为 null 的参数不会为 null。
- NotNullIfNotNull：如果指定参数的参数不为 null，则返回值不为 null。

帮助程序方法：`MemberNotNull` 和 `MemberNotNullWhen`

这些特性指定了将构造函数中的公共代码重构为帮助程序方法时的意图。 C# 编译器分析构造函数和字段初始值设定项，以确保在每个构造函数返回之前，所有不可为 null 的引用字段都已初始化。 然而，C# 编译器不会通过所有帮助程序方法跟踪字段赋值。 当字段没有在构造函数中直接初始化，而在帮助程序方法中初始化时，编译器会发出警告 `CS8618`。 可以将 MemberNotNullAttribute 添加到方法声明中，并指定在方法中初始化为非 NULL 值的字段。 例如，考虑以下情况：

C#

```
public class Container
{
    private string _uniqueIdentifier; // must be initialized.
    private string? _optionalMessage;

    public Container()
    {
        Helper();
    }

    public Container(string message)
    {
        Helper();
        _optionalMessage = message;
    }

    [MemberNotNull(nameof(_uniqueIdentifier))]
    private void Helper()
    {
        _uniqueIdentifier = DateTime.Now.Ticks.ToString();
    }
}
```

可以指定多个字段名称作为 `MemberNotNull` 特性构造函数的参数。

MemberNotNullWhenAttribute 有 `bool` 参数。 在帮助程序方法返回指明帮助程序方法是否初始化了字段的 `bool` 的情况下，可以使用 `MemberNotNullWhen`。

调用的方法引发时停止可为 null 分析

某些方法（通常是异常帮助程序或其他实用工具方法）始终通过引发异常来退出。 或者，帮助程序可以基于布尔参数的值引发异常。

在第一种情况下，可以将 DoesNotReturnAttribute 特性添加到方法声明中。 编译器的 null 状态分析不会检查调用带有 `DoesNotReturn` 注释的方法之后的方法中的任何代码。 请考虑此方法：

C#

```
[DoesNotReturn]
private void FailFast()
{
    throw new InvalidOperationException();
}

public void SetState(object containedField)
{
    if (containedField is null)
    {
        FailFast();
    }

    // containedField can't be null:
    _field = containedField;
}
```

调用 `FailFast` 后，编译器不会发出任何警告。

在第二种情况下，需将 System.Diagnostics.CodeAnalysis.DoesNotReturnIfAttribute 特性添加到方法的布尔参数中。 你可以修改前面的示例，如下所示：

C#

```
private void FailFastIf([DoesNotReturnIf(true)] bool isNull)
{
    if (isNull)
    {
        throw new InvalidOperationException();
    }
}

public void SetFieldState(object? containedField)
{
    FailFastIf(containedField == null);
    // No warning: containedField can't be null here:
    _field = containedField;
}
```

当参数的值与 `DoesNotReturnIf` 构造函数的值匹配时，编译器不会在该方法之后执行任何 null 状态分析。

总结

添加可为 null 的引用类型提供了一个初始词汇表，用于描述 API 对可能为 `null` 的变量的期望。 这些特性提供了更丰富的词汇来将变量的 null 状态描述为前置条件和后置条件。 这些特性更清楚地描述了你的期望，并为使用 API 的开发人员提供了更好的体验。

在为可为 null 的上下文中更新库时，添加这些特性可指导用户正确使用 API。 这些特性有助于你全面描述参数和返回值的 null 状态。

- AllowNull：不可为 null 的字段、参数或属性可以为 null。
- DisallowNull：可为 null 的字段、参数或属性应永不为 null。
- MaybeNull：不可为 null 的字段、参数、属性或返回值可以为 null。
- NotNull：可为 null 的字段、参数、属性或返回值将永不为 null。
- MaybeNullWhen：当方法返回指定的 `bool` 值时，不可为 null 的参数可以为 null。
- NotNullWhen：当方法返回指定的 `bool` 值时，可以为 null 的参数不会为 null。
- NotNullIfNotNull：如果指定参数的自变量不为 null，则参数、属性或返回值不为 null。
- DoesNotReturn：方法或属性永远不会返回。 换句话说，它总是引发异常。
- DoesNotReturnIf：如果关联的 `bool` 参数具有指定值，则此方法或属性永远不会返回。

C# 编译器解释的杂项属性

项目 • 2025/01/03

有几个属性可以应用于代码中的元素，这些元素向这些元素添加语义含义：

- Conditional：执行依赖于预处理器标识符的方法。
- Obsolete：标记一个类型或成员，以便将来（可能）删除。
- AttributeUsage：声明可应用属性的语言元素。
- AsyncMethodBuilder：声明异步方法生成器类型。
- InterpolatedStringHandler：为已知应用场景定义内插字符串生成器。
- ModuleInitializer：声明初始化模块的方法。
- SkipLocalsInit：省略将本地变量存储初始化为 0 的代码。
- UnscopedRef：声明通常解释为 `ref` 的 `scoped` 变量应被视为无作用域。
- OverloadResolutionPriority：添加 tiebreaker 属性，以影响可能不明确的重载的重载解析。
- Experimental：将类型或成员标记为实验性。

编译器使用这些语义来更改其输出，并报告使用你的代码的开发人员可能犯的错误。

`Conditional` 特性

`Conditional` 特性使得方法执行依赖于预处理标识符。 `Conditional` 属性是 ConditionalAttribute 的别名，可以应用于方法或特性类。

在以下示例中，`Conditional` 应用于启用或禁用显示特定于程序的诊断信息的方法：

C#

```
#define TRACE_ON
using System.Diagnostics;

namespace AttributeExamples;

public class Trace
{
    [Conditional("TRACE_ON")]
    public static void Msg(string msg)
    {
        Console.WriteLine(msg);
    }
}

public class TraceExample
{
    public static void Main()
```

```
    {
        Trace.Msg("Now in Main...");
        Console.WriteLine("Done.");
    }
}
```

如果未定义 `TRACE_ON` 标识符，则不会显示跟踪输出。 在交互式窗口中自己探索。

`Conditional` 特性通常与 `DEBUG` 标识符一起使用，以启用调试生成（而非发布生成）中的跟踪和日志记录功能，如下例所示：

C#

```
[Conditional("DEBUG")]
static void DebugMethod()
{
}
```

当调用标记为条件的方法时，指定的预处理符号是否存在将决定编译器是包含还是省略对该方法的调用。 如果定义了符号，则将包括调用；否则，将忽略该调用。 条件方法必须是类或结构声明中的方法，而且必须具有 `void` 返回类型。 与将方法封闭在 `Conditional` 块内相比，`#if…#endif` 更简洁且较不容易出错。

如果某个方法具有多个 `Conditional` 特性，则如果定义了一个或多个条件符号（通过使用 OR 运算符将这些符号逻辑链接在一起），编译器会包含对该方法的调用。 在以下示例中，存在 `A` 或 `B` 将导致方法调用：

C#

```
[Conditional("A"), Conditional("B")]
static void DoIfAorB()
{
    // ...
}
```

使用带有特性类的 `Conditional`

`Conditional` 特性还可应用于特性类定义。 在以下示例中，如果定义了 `Documentation`，则自定义特性 `DEBUG` 会向元数据添加信息。

C#

```
[Conditional("DEBUG")]
public class DocumentationAttribute : System.Attribute
{
```

```
    string text;

    public DocumentationAttribute(string text)
    {
        this.text = text;
    }
}

class SampleClass
{
    // This attribute will only be included if DEBUG is defined.
    [Documentation("This method displays an integer.")]
    static void DoWork(int i)
    {
        System.Console.WriteLine(i.ToString());
    }
}
```

`Obsolete` 特性

`Obsolete` 特性将代码元素标记为不再推荐使用。 使用标记为已过时的实体会生成警告或错误。 `Obsolete` 特性是一次性特性，可以应用于任何允许特性的实体。 `Obsolete` 是 ObsoleteAttribute 的别名。

在以下示例中，对类 `Obsolete` 和方法 `A` 应用了 `B.OldMethod` 特性。 因为应用于 `B.OldMethod` 的特性构造函数的第二个参数设置为 `true`，所以此方法会导致编译器错误，而使用类 `A` 只会生成警告。 但是，调用 `B.NewMethod` 不会生成任何警告或错误。 例如，将其与先前的定义一起使用时，以下代码会生成两个警告和一个错误：

C#

```
namespace AttributeExamples
{
    [Obsolete("use class B")]
    public class A
    {
        public void Method() { }
    }

    public class B
    {
        [Obsolete("use NewMethod", true)]
        public void OldMethod() { }

        public void NewMethod() { }
    }

    public static class ObsoleteProgram
```

```
    {
        public static void Main()
        {
            // Generates 2 warnings:
            A a = new A();

            // Generate no errors or warnings:
            B b = new B();
            b.NewMethod();

            // Generates an error, compilation fails.
            // b.OldMethod();
        }
    }
}
```

作为特性构造函数的第一个参数提供的字符串会作为警告或错误的一部分显示。 将生成类 `A` 的两个警告：一个用于声明类引用，另一个用于类构造函数。 `Obsolete` 特性可以在不带参数的情况下使用，但建议说明改为使用哪个项目。 可以使用常量字符串内插和 `nameof` 运算符来确保名称匹配：

C#

```
public class B
{
    [Obsolete($"use {nameof(NewMethod)} instead", true)]
    public void OldMethod() { }

    public void NewMethod() { }
}
```

`Experimental` 特性

从 C# 12 开始，可以使用 System.Diagnostics.CodeAnalysis.ExperimentalAttribute 来标记类型、方法和程序集以指示实验性功能。 如果访问使用 ExperimentalAttribute 注释的方法或类型，编译器将发出警告。 用 `Experimental` 特性标记的程序集或模块中声明的所有类型都是实验性的。 如果访问其中任何一种类型，编译器都会发出警告。 可以禁用这些警告以试用实验性功能。

> **⚠ 警告**
>
> 实验性功能可能会随时更改。 API 可能会更改，或者可能会在未来的更新中被删除。 包括实验性功能是库作者获取有关未来开发的想法和概念反馈的一种方式。 使用标记为实验性的任何功能时，请格外小心。

可以在`Experimental`中阅读有关 属性的更多详细信息。

`SetsRequiredMembers` 特性

`SetsRequiredMembers` 属性通知编译器构造函数设置了该类或结构中的所有 `required` 成员。 编译器假定任何具有 System.Diagnostics.CodeAnalysis.SetsRequiredMembersAttribute 属性的构造函数都会初始化所有 `required` 成员。 调用此类构造函数的任何代码都不需要对象初始值设定项来设置所需的成员。 添加 `SetsRequiredMembers` 特性主要用于位置记录和主要构造函数。

`AttributeUsage` 特性

`AttributeUsage` 特性确定自定义特性类的使用方式。 AttributeUsageAttribute 是应用到自定义特性定义的特性。 `AttributeUsage` 特性帮助控制：

- 可对哪些程序元素应用特性。 除非使用限制，否则特性可能应用到以下任意程序元素：
  - 程序集
  - 模块
  - 字段
  - 事件
  - 方法
  - 参数
  - properties
  - 返回
  - 类型
- 某特性是否可多次应用于单个程序元素。
- 派生类是否继承特性。

显式应用时，默认设置如以下示例所示：

C#

```
[AttributeUsage(AttributeTargets.All,
                  AllowMultiple = false,
                  Inherited = true)]
class NewAttribute : Attribute { }
```

在此示例中，`NewAttribute` 类可应用于任何受支持的程序元素。 但是它对每个实体仅能应用一次。 派生类继承应用于基类的特性。

AllowMultiple 和 Inherited 参数是可选的，因此以下代码具有相同效果：

C#

```
[AttributeUsage(AttributeTargets.All)]
class NewAttribute : Attribute { }
```

第一个 AttributeUsageAttribute 参数必须是 AttributeTargets 枚举的一个或多个元素。 可将多个目标类型与 OR 运算符链接在一起，如下例所示：

C#

```
[AttributeUsage(AttributeTargets.Property | AttributeTargets.Field)]
class NewPropertyOrFieldAttribute : Attribute { }
```

属性可以应用于属性或自动实现属性的后盾字段。 特性应用于属性，除非在特性上指定 `field` 说明符。 都在以下示例中进行了演示：

C#

```
class MyClass
{
    // Attribute attached to property:
    [NewPropertyOrField]
    public string Name { get; set; } = string.Empty;

    // Attribute attached to backing field:
    [field: NewPropertyOrField]
    public string Description { get; set; } = string.Empty;
}
```

如果 AllowMultiple 参数为 `true`，那么结果特性可多次应用于单个实体，如以下示例所示：

C#

```
[AttributeUsage(AttributeTargets.Class, AllowMultiple = true)]
class MultiUse : Attribute { }

[MultiUse]
[MultiUse]
class Class1 { }

[MultiUse, MultiUse]
class Class2 { }
```

在本例中，`MultiUseAttribute` 可重复应用，因为 `AllowMultiple` 设置为 `true`。 所显示的两种用于应用多个特性的格式均有效。

如果 Inherited 是 `false`，则派生类不会从特性化基类继承特性。 例如：

```
C#
```

```
[AttributeUsage(AttributeTargets.Class, Inherited = false)]
class NonInheritedAttribute : Attribute { }

[NonInherited]
class BClass { }

class DClass : BClass { }
```

在本例中，`NonInheritedAttribute` 不会通过继承应用于 `DClass`。

你还可以使用这些关键字来指定应在何处应用特性。 例如，可以使用`field:`说明符将属性添加到自动实现属性的后盾字段中。 或者，可以使用 `field:`、`property:` 或 `param:` 说明符将特性应用于根据位置记录生成的任何元素。 有关示例，请参阅属性定义的位置语法。

`AsyncMethodBuilder` 特性

将 System.Runtime.CompilerServices.AsyncMethodBuilderAttribute 特性添加到可为异步返回类型的类型。 该特性指定会在异步方法返回指定类型时生成异步方法实现的类型。 `AsyncMethodBuilder` 属性可应用于符合以下条件的类型：

- 具有可访问的 `GetAwaiter` 方法。
- `GetAwaiter` 方法返回的对象实现 System.Runtime.CompilerServices.ICriticalNotifyCompletion 接口。

`AsyncMethodBuilder` 特性的构造函数指定关联的生成器的类型。 生成器必须实现以下可访问的成员：

- 一个静态 `Create()` 方法，可返回生成器的类型。
- 一个可读的 `Task` 属性，可返回异步返回类型。
- 一个 `void SetException(Exception)` 方法，可在任务出错时设置异常。
- `void SetResult()` 或 `void SetResult(T result)` 方法，可将任务标记为已完成，并选择性地设置任务的结果
- 一个具有以下 API 签名的 `Start` 方法：

  ```
  C#
  ```

```
void Start<TStateMachine>(ref TStateMachine stateMachine)
          where TStateMachine :
System.Runtime.CompilerServices.IAsyncStateMachine
```

- 具有以下签名的 `AwaitOnCompleted` 方法：

```
C#

public void AwaitOnCompleted<TAwaiter, TStateMachine>(ref TAwaiter
awaiter, ref TStateMachine stateMachine)
    where TAwaiter : System.Runtime.CompilerServices.INotifyCompletion
    where TStateMachine :
System.Runtime.CompilerServices.IAsyncStateMachine
```

- 具有以下签名的 `AwaitUnsafeOnCompleted` 方法：

```
C#

      public void AwaitUnsafeOnCompleted<TAwaiter, TStateMachine>(ref
TAwaiter awaiter, ref TStateMachine stateMachine)
          where TAwaiter :
System.Runtime.CompilerServices.ICriticalNotifyCompletion
          where TStateMachine :
System.Runtime.CompilerServices.IAsyncStateMachine
```

若要了解异步方法生成器，可以阅读有关 .NET 提供的以下生成器的信息：

- System.Runtime.CompilerServices.AsyncTaskMethodBuilder
- System.Runtime.CompilerServices.AsyncTaskMethodBuilder<TResult>
- System.Runtime.CompilerServices.AsyncValueTaskMethodBuilder
- System.Runtime.CompilerServices.AsyncValueTaskMethodBuilder<TResult>

`AsyncMethodBuilder` 属性可以应用于异步方法，以替代该类型的生成器。

`InterpolatedStringHandler` 和 `InterpolatedStringHandlerArguments` 属性

可以使用这些属性指定类型为*内插字符串处理程序*。 在你使用内插字符串作为 System.Runtime.CompilerServices.DefaultInterpolatedStringHandler 参数的自变量的情况下，.NET 6 库已包含 `string`。 你可能还有其他要控制内插字符串处理方式的实例。 你需要将 System.Runtime.CompilerServices.InterpolatedStringHandlerAttribute 应用到实现处理程序的类型。 你需要将

System.Runtime.CompilerServices.InterpolatedStringHandlerArgumentAttribute 应用到类型的构造函数的参数。

若要详细了解如何生成内插字符串处理程序，可以参阅内插字符串改进的功能规范。

`ModuleInitializer` 特性

`ModuleInitializer` 属性标记程序集加载时运行时调用的方法。 `ModuleInitializer` 是 ModuleInitializerAttribute 的别名。

`ModuleInitializer` 属性只能应用于以下方法：

- 静态方法。
- 无参数方法。
- 返回 `void`。
- 能够从包含模块（即 `internal` 或 `public`）访问的方法。
- 不是泛型的方法。
- 没有包含在泛型类中的方法。
- 不是本地函数的方法。

`ModuleInitializer` 属性可应用于多种方法。 在这种情况下，运行时调用它们的顺序是确定的，但未指定。

下面的示例阐释了如何使用多个模块初始化表达式方法。 `Init1` 和 `Init2` 方法在 `Main` 之前运行，并且每种方法都将一个字符串添加到 `Text` 属性。 因此，当 `Main` 运行时，`Text` 属性已具有来自两个初始化表达式方法中的字符串。

C#

```
using System;

internal class ModuleInitializerExampleMain
{
    public static void Main()
    {
        Console.WriteLine(ModuleInitializerExampleModule.Text);
        //output: Hello from Init1! Hello from Init2!
    }
}
```

C#

```
using System.Runtime.CompilerServices;

internal class ModuleInitializerExampleModule
```

```
{
    public static string? Text { get; set; }

    [ModuleInitializer]
    public static void Init1()
    {
        Text += "Hello from Init1! ";
    }

    [ModuleInitializer]
    public static void Init2()
    {
        Text += "Hello from Init2! ";
    }
}
```

源代码生成器有时需要生成初始化代码。 模块初始化表达式为该代码提供了一个标准位置。 在大多数情况下，应编写静态构造函数，而不是模块初始值设定项。

`SkipLocalsInit` 特性

`SkipLocalsInit` 属性可防止编译器在发出到元数据时设置 `.locals init` 标志。 `SkipLocalsInit` 属性是一个单用途属性，可应用于方法、属性、类、结构、接口或模块，但不能应用于程序集。 `SkipLocalsInit` 是 SkipLocalsInitAttribute 的别名。

`.locals init` 标志会导致 CLR 将方法中声明的所有局部变量初始化为其默认值。 由于编译器还可以确保在为变量赋值之前永远不使用变量，因此通常不需要使用 `.locals init`。 但是，在某些情况下，额外的零初始化可能会对性能产生显著影响，例如使用 stackalloc 在堆栈上分配一个数组时。 在这些情况下，可添加 `SkipLocalsInit` 属性。 如果直接应用于方法，该属性会影响该方法及其所有嵌套函数，包括 lambda 和局部函数。 如果应用于类型或模块，则它会影响嵌套在内的所有方法。 此属性不会影响抽象方法，但会影响为实现生成的代码。

此属性需要 AllowUnsafeBlocks 编译器选项。 这一要求表明，在某些情况下，代码可以查看未分配的内存（例如，读取未初始化的堆栈分配的内存）。

下面的示例阐释 `SkipLocalsInit` 属性对使用 `stackalloc` 的方法的影响。 该方法显示分配整数数组后内存中的任何内容。

C#

```
[SkipLocalsInit]
static void ReadUninitializedMemory()
{
    Span<int> numbers = stackalloc int[120];
    for (int i = 0; i < 120; i++)
```

```
    {
        Console.WriteLine(numbers[i]);
    }
}
// output depends on initial contents of memory, for example:
//0
//0
//0
//168
//0
//-1271631451
//32767
//38
//0
//0
//0
//38
// Remaining rows omitted for brevity.
```

若要亲自尝试此代码，请在 .csproj 文件中设置 `AllowUnsafeBlocks` 编译器选项：

XML

```
<PropertyGroup>
  ...
  <AllowUnsafeBlocks>true</AllowUnsafeBlocks>
</PropertyGroup>
```

`UnscopedRef` 特性

`UnscopedRef` 属性将变量声明标记为无作用域，这意味着允许引用进行转义。

添加此属性，其中编译器将 `ref` 视为隐式 `scoped`：

- `this` 实例方法的 `struct` 参数。
- 引用 `ref` 类型的 `ref struct` 参数。
- `out` 参数。

应用 System.Diagnostics.CodeAnalysis.UnscopedRefAttribute 会将元素标记为无作用域。

`OverloadResolutionPriority` 特性

当两个重载可能不明确时，OverloadResolutionPriorityAttribute 使库作者能够选择一个重载而不是另一个重载。 它的主要用例是让库作者编写性能更好的重载，同时仍然支持现有代码而不中断。

例如，可以添加使用 ReadOnlySpan<T> 减少内存分配的新重载：

C#

```
[OverloadResolutionPriority(1)]
public void M(params ReadOnlySpan<int> s) => Console.WriteLine("Span");
// Default overload resolution priority of 0
public void M(params int[] a) => Console.WriteLine("Array");
```

重载解析认为这两种方法同样适用于某些参数类型。 对于 `int[]` 的参数，它首选第一个重载。 若要让编译器首选 `ReadOnlySpan` 版本，你可以增加该重载的优先级。 以下示例显示了添加该属性的效果：

C#

```
var d = new OverloadExample();
int[] arr = [1, 2, 3];
d.M(1, 2, 3, 4); // Prints "Span"
d.M(arr); // Prints "Span" when PriorityAttribute is applied
d.M([1, 2, 3, 4]); // Prints "Span"
d.M(1, 2, 3, 4); // Prints "Span"
```

所有优先级低于最高重载优先级的重载会从适用方法集中删除。 没有此属性的方法的重载优先级设置为默认值零。 库作者在添加更好的新方法重载时，万不得已才应使用此属性。 库作者应深入了解重载解析对选择更好的方法有何影响。 否则，可能会导致意外错误。

另请参阅

- Attribute
- System.Reflection
- 特性
- 反射

不安全代码、指针类型和函数指针

项目 • 2023/05/10

你编写的大部分 C# 代码都是“可验证的安全代码”。可验证的安全代码是指 .NET 工具可验证代码是否安全。 通常，安全代码不会直接使用指针访问内存， 也不会分配原始内存， 而是创建托管对象。

C# 支持 unsafe 上下文，你可在其中编写不可验证的代码。 在 `unsafe` 上下文中，代码可使用指针、分配和释放内存块，以及使用函数指针调用方法。 C# 中的不安全代码不一定是危险的，它只是其安全性不可验证的代码。

不安全代码具有以下属性：

- 可将方法、类型和代码块定义为不安全。
- 在某些情况下，通过移除数组绑定检查，不安全代码可提高应用程序的性能。
- 调用需要指针的本机函数时，需使用不安全代码。
- 使用不安全代码将引发安全风险和稳定性风险。
- 必须使用 AllowUnsafeBlocks 编译器选项来编译包含不安全块的代码。

指针类型

在不安全的上下文中，类型除了是值类型或引用类型外，还可以是指针类型。 指针类型声明采用下列形式之一：

C#

```
type* identifier;
void* identifier; //allowed but not recommended
```

在指针类型中的 `*` 之前指定的类型被称为“referent 类型”。 只有非托管类型可为引用类型。

指针类型不从对象继承，并且指针类型与 之间不存在转换。 此外，装箱和取消装箱不支持指针。 但是，你可在不同的指针类型之间以及指针类型和整型之间进行转换。

当在同一声明中声明多个指针时，星号 (`*`) 仅与基础类型一起写入， 而不是用作每个指针名称的前缀。 例如：

C#

```
int* p1, p2, p3;   // Ok
int *p1, *p2, *p3;   // Invalid in C#
```

指针不能指向引用或包含引用的结构，因为无法对对象引用进行垃圾回收，即使有指针指向它也是如此。 垃圾回收器并不跟踪是否有任何类型的指针指向对象。

`MyType*` 类型的指针变量的值为 `MyType` 类型的变量的地址。 下面是指针类型声明的示例：

- `int* p`： `p` 是指向整数的指针。
- `int** p`： `p` 是指向整数的指针的指针。
- `int*[] p`： `p` 是指向整数的指针的一维数组。
- `char* p`： `p` 是指向字符的指针。
- `void* p`： `p` 是指向未知类型的指针。

指针间接寻址运算符 `*` 可用于访问位于指针变量所指向的位置的内容。 例如，请考虑以下声明：

C#

```
int* myVariable;
```

表达式 `*myVariable` 表示在 `int` 中包含的地址处找到的 `myVariable` 变量。

关于 语句 的文章中有几个指针示例。 下面的示例使用 `unsafe` 关键字和 `fixed` 语句，并显示如何递增内部指针。 你可将此代码粘贴到控制台应用程序的 Main 函数中来运行它。 这些示例必须使用 AllowUnsafeBlocks 编译器选项集进行编译。

C#

```
// Normal pointer to an object.
int[] a = new int[5] { 10, 20, 30, 40, 50 };
// Must be in unsafe code to use interior pointers.
unsafe
{
    // Must pin object on heap so that it doesn't move while using interior
pointers.
    fixed (int* p = &a[0])
    {
        // p is pinned as well as object, so create another pointer to show
incrementing it.
        int* p2 = p;
        Console.WriteLine(*p2);
        // Incrementing p2 bumps the pointer by four bytes due to its type
...
        p2 += 1;
        Console.WriteLine(*p2);
        p2 += 1;
        Console.WriteLine(*p2);
        Console.WriteLine("--------");
        Console.WriteLine(*p);
```

```
        // Dereferencing p and incrementing changes the value of a[0] ...
        *p += 1;
        Console.WriteLine(*p);
        *p += 1;
        Console.WriteLine(*p);
    }
}

Console.WriteLine("--------");
Console.WriteLine(a[0]);

/*
Output:
10
20
30
--------
10
11
12
--------
12
*/
```

你无法对 `void*` 类型的指针应用间接寻址运算符。 但是，你可以使用强制转换将 void 指针转换为任何其他指针类型，反之亦然。

指针可以为 `null`。 将间接寻址运算符应用于 null 指针将导致由实现定义的行为。

在方法之间传递指针会导致未定义的行为。 考虑这种方法，该方法通过 `in`、`out` 或 `ref` 参数或作为函数结果返回一个指向局部变量的指针。 如果已在固定块中设置指针，则它指向的变量不再是固定的。

下表列出了可在不安全的上下文中对指针执行的运算符和语句：

运算符/语句	使用
`*`	执行指针间接寻址。
`->`	通过指针访问结构的成员。
`[]`	为指针建立索引。
`&`	获取变量的地址。
`++` 和 `--`	递增和递减指针。
`+` 和 `-`	执行指针算法。
`==`、`!=`、`<`、`>`、`<=` 和 `>=`	比较指针。

运算符/语句	使用
stackalloc	在堆栈上分配内存。
fixed 语句	临时固定变量以便找到其地址。

要详细了解与指针相关的运算符，请参阅与指针相关的运算符。

任何指针类型都可以隐式转换为 `void*` 类型。 可以为任何指针类型分配值 `null`。 可以使用强制转换表达式将任何指针类型显式转换为任何其他指针类型。 也可以将任何整数类型转换为指针类型，或将任何指针类型转换为整数类型。 这些转换需要显式转换。

以下示例将 `int*` 转换为 `byte*`。 请注意，指针指向变量的最低寻址字节。 如果结果连续递增，直到达到 `int` 的大小（4 字节），可显示变量的其余字节。

C#

```
int number = 1024;

unsafe
{
    // Convert to byte:
    byte* p = (byte*)&number;

    System.Console.Write("The 4 bytes of the integer:");

    // Display the 4 bytes of the int variable:
    for (int i = 0 ; i < sizeof(int) ; ++i)
    {
        System.Console.Write(" {0:X2}", *p);
        // Increment the pointer:
        p++;
    }
    System.Console.WriteLine();
    System.Console.WriteLine("The value of the integer: {0}", number);

    /* Output:
        The 4 bytes of the integer: 00 04 00 00
        The value of the integer: 1024
    */
}
```

固定大小的缓冲区

可以使用 `fixed` 关键字来创建在数据结构中具有固定大小的数组的缓冲区。 当编写与其他语言或平台的数据源进行互操作的方法时，固定大小的缓冲区很有用。 固定大小的缓冲区可以采用允许用于常规结构成员的任何属性或修饰符。 唯一的限制是数组类型必须

为 `bool`、`byte`、`char`、`short`、`int`、`long`、`sbyte`、`ushort`、`uint`、`ulong`、`float` 或 `double`。

C#

```
private fixed char name[30];
```

在安全代码中，包含数组的 C# 结构不包含该数组的元素， 而该结构包含对这些元素的引用。 当在不安全的代码块中使用数组时，可以在结构中嵌入固定大小的数组。

以下 `struct` 的大小不依赖于数组中的元素数，因为 `pathName` 是一个引用：

C#

```
public struct PathArray
{
    public char[] pathName;
    private int reserved;
}
```

结构可以在不安全代码中包含嵌入的数组。 在下面的示例中，`fixedBuffer` 数组具有固定的大小。 使用 fixed 语句获取指向第一个元素的指针。 通过此指针访问数组的元素。 `fixed` 语句将 `fixedBuffer` 实例字段固定到内存中的特定位置。

C#

```
internal unsafe struct Buffer
{
    public fixed char fixedBuffer[128];
}

internal unsafe class Example
{
    public Buffer buffer = default;
}

private static void AccessEmbeddedArray()
{
    var example = new Example();

    unsafe
    {
        // Pin the buffer to a fixed location in memory.
        fixed (char* charPtr = example.buffer.fixedBuffer)
        {
            *charPtr = 'A';
        }
        // Access safely through the index:
        char c = example.buffer.fixedBuffer[0];
```

```
        Console.WriteLine(c);

        // Modify through the index:
        example.buffer.fixedBuffer[0] = 'B';
        Console.WriteLine(example.buffer.fixedBuffer[0]);
    }
}
```

包含 128 个元素的 `char` 数组的大小为 256 个字节。 在固定大小的 char 缓冲区中，每个字符总是占用 2 个字节，不考虑编码。 即使将字符型缓冲区封送到 API 方法或具有 `CharSet = CharSet.Auto` 或 `CharSet = CharSet.Ansi` 的结构，此数组大小也是相同的。 有关更多信息，请参见CharSet。

前面的示例演示访问未固定的 `fixed` 字段。 另一常见的固定大小的数组是 bool 数组。 `bool` 数组中的元素大小始终为 1 个字节。 `bool` 数组不适用于创建位数组或缓冲区。

固定大小的缓冲区使用 System.Runtime.CompilerServices.UnsafeValueTypeAttribute 进行编译，它指示公共语言运行时 (CLR) 某个类型包含可能溢出的非托管数组。 使用 stackalloc 分配的内存还会在 CLR 中自动启用缓冲区溢出检测功能。 前面的示例演示如何在 `unsafe struct` 中存在固定大小的缓冲区。

C#

```
internal unsafe struct Buffer
{
    public fixed char fixedBuffer[128];
}
```

为 `Buffer` 生成 C# 的编译器的特性如下：

C#

```
internal struct Buffer
{
    [StructLayout(LayoutKind.Sequential, Size = 256)]
    [CompilerGenerated]
    [UnsafeValueType]
    public struct <fixedBuffer>e__FixedBuffer
    {
        public char FixedElementField;
    }

    [FixedBuffer(typeof(char), 128)]
    public <fixedBuffer>e__FixedBuffer fixedBuffer;
}
```

固定大小的缓冲区与常规数组的区别体现在以下方面：

- 只能在 `unsafe` 上下文中使用。
- 只能是结构的实例字段。
- 它们始终是矢量或一维数组。
- 声明应包括长度，如 `fixed char id[8]`。 不能使用 `fixed char id[]`。

如何使用指针来复制字节数组

下面的示例使用指针将字节从一个数组复制到另一个数组。

此示例使用 unsafe 关键字，使你可以在 方法中使用指针。 fixed 语句用于声明指向源数组和目标数组的指针。 `fixed` 语句将源数组和目标数组的位置固定在内存中，以便它们不会被垃圾回收所移动。 当完成 `fixed` 块后，将取消固定数组的内存块。 因为此示例中的 `Copy` 方法使用 `unsafe` 关键字，所以必须使用 `Copy` 编译器选项对其进行编译。

此示例使用索引而非第二个非托管的指针访问这两个数组的元素。 `pSource` 和 `pTarget` 指针的声明固定数组。

C#

```
static unsafe void Copy(byte[] source, int sourceOffset, byte[] target,
    int targetOffset, int count)
{
    // If either array is not instantiated, you cannot complete the copy.
    if ((source == null) || (target == null))
    {
        throw new System.ArgumentException("source or target is null");
    }

    // If either offset, or the number of bytes to copy, is negative, you
    // cannot complete the copy.
    if ((sourceOffset < 0) || (targetOffset < 0) || (count < 0))
    {
        throw new System.ArgumentException("offset or bytes to copy is
negative");
    }

    // If the number of bytes from the offset to the end of the array is
    // less than the number of bytes you want to copy, you cannot complete
    // the copy.
    if ((source.Length - sourceOffset < count) ||
        (target.Length - targetOffset < count))
    {
        throw new System.ArgumentException("offset to end of array is less
than bytes to be copied");
    }

    // The following fixed statement pins the location of the source and
    // target objects in memory so that they will not be moved by garbage
    // collection.
```

```
    fixed (byte* pSource = source, pTarget = target)
    {
        // Copy the specified number of bytes from source to target.
        for (int i = 0; i < count; i++)
        {
            pTarget[targetOffset + i] = pSource[sourceOffset + i];
        }
    }
}

static void UnsafeCopyArrays()
{
    // Create two arrays of the same length.
    int length = 100;
    byte[] byteArray1 = new byte[length];
    byte[] byteArray2 = new byte[length];

    // Fill byteArray1 with 0 - 99.
    for (int i = 0; i < length; ++i)
    {
        byteArray1[i] = (byte)i;
    }

    // Display the first 10 elements in byteArray1.
    System.Console.WriteLine("The first 10 elements of the original are:");
    for (int i = 0; i < 10; ++i)
    {
        System.Console.Write(byteArray1[i] + " ");
    }
    System.Console.WriteLine("\n");

    // Copy the contents of byteArray1 to byteArray2.
    Copy(byteArray1, 0, byteArray2, 0, length);

    // Display the first 10 elements in the copy, byteArray2.
    System.Console.WriteLine("The first 10 elements of the copy are:");
    for (int i = 0; i < 10; ++i)
    {
        System.Console.Write(byteArray2[i] + " ");
    }
    System.Console.WriteLine("\n");

    // Copy the contents of the last 10 elements of byteArray1 to the
    // beginning of byteArray2.
    // The offset specifies where the copying begins in the source array.
    int offset = length - 10;
    Copy(byteArray1, offset, byteArray2, 0, length - offset);

    // Display the first 10 elements in the copy, byteArray2.
    System.Console.WriteLine("The first 10 elements of the copy are:");
    for (int i = 0; i < 10; ++i)
    {
        System.Console.Write(byteArray2[i] + " ");
    }
    System.Console.WriteLine("\n");
```

```
    /* Output:
        The first 10 elements of the original are:
        0 1 2 3 4 5 6 7 8 9

        The first 10 elements of the copy are:
        0 1 2 3 4 5 6 7 8 9

        The first 10 elements of the copy are:
        90 91 92 93 94 95 96 97 98 99
    */
}
```

函数指针

C# 提供 delegate 类型来定义安全函数指针对象。 调用委托时，需要实例化从 System.Delegate 派生的类型并对其 `Invoke` 方法进行虚拟方法调用。 该虚拟调用使用 IL 指令 `callvirt`。 在性能关键的代码路径中，使用 IL 指令 `calli` 效率更高。

可以使用 `delegate*` 语法定义函数指针。 编译器将使用 `calli` 指令来调用函数，而不是实例化 `delegate` 对象并调用 `Invoke`。 以下代码声明了两种方法，它们使用 `delegate` 或 `delegate*` 来组合两个类型相同的对象。 第一种方法使用 System.Func<T1,T2,TResult> 委托类型。 第二种方法使用具有相同参数和返回类型的 `delegate*` 声明：

C#

```
public static T Combine<T>(Func<T, T, T> combinator, T left, T right) =>
    combinator(left, right);

public static T UnsafeCombine<T>(delegate*<T, T, T> combinator, T left, T
right) =>
    combinator(left, right);
```

以下代码显示如何声明静态本地函数并使用指向该本地函数的指针调用 `UnsafeCombine` 方法：

C#

```
static int localMultiply(int x, int y) => x * y;
int product = UnsafeCombine(&localMultiply, 3, 4);
```

前面的代码说明了有关作为函数指针使用的函数的几个规则：

- 函数指针只能在 `unsafe` 上下文中声明。
- 只能在 `unsafe` 上下文中调用采用 `delegate*`（或返回 `delegate*`）的方法。

- 只可在 `static` 函数上使用 `&` 运算符获取函数的地址。 （此规则适用于成员函数和本地函数）。

此语法与声明 `delegate` 类型和使用指针具有相似之处。 `delegate` 上的后缀 `*` 表示声明是函数指针。 将方法组分配给函数指针时，`&` 表示操作采用方法的地址。

可以使用关键字 `managed` 和 `unmanaged` 为 `delegate*` 指定调用约定。 另外，对于 `unmanaged` 函数指针，可以指定调用约定。 下面的声明显示每个示例。 第一个声明使用 `managed` 调用约定，这是默认值。 后面四个使用 `unmanaged` 调用约定。 每个声明都指定以下某个 ECMA 335 调用约定：`Cdecl`、`Stdcall`、`Fastcall` 或 `Thiscall`。 最后的声明使用 `unmanaged` 调用约定，指示 CLR 选择平台的默认调用约定。 CLR 将在运行时选择调用约定。

C#

```
public static T ManagedCombine<T>(delegate* managed<T, T, T> combinator, T
left, T right) =>
    combinator(left, right);
public static T CDeclCombine<T>(delegate* unmanaged[Cdecl]<T, T, T>
combinator, T left, T right) =>
    combinator(left, right);
public static T StdcallCombine<T>(delegate* unmanaged[Stdcall]<T, T, T>
combinator, T left, T right) =>
    combinator(left, right);
public static T FastcallCombine<T>(delegate* unmanaged[Fastcall]<T, T, T>
combinator, T left, T right) =>
    combinator(left, right);
public static T ThiscallCombine<T>(delegate* unmanaged[Thiscall]<T, T, T>
combinator, T left, T right) =>
    combinator(left, right);
public static T UnmanagedCombine<T>(delegate* unmanaged<T, T, T> combinator,
T left, T right) =>
    combinator(left, right);
```

可以在 C# 9.0 的函数指针建议中详细了解函数指针。

C# 语言规范

有关详细信息，请参阅 C# 语言规范中的不安全代码部分。

C# 预处理器指令

项目 • 2024/11/12

尽管编译器没有单独的预处理器，但本节中所述指令的处理方式与有预处理器时一样。 可使用这些指令来帮助条件编译。 不同于 C 和 C++ 指令，不能使用这些指令来创建宏。 预处理器指令必须是一行中唯一的说明。

可为空上下文

`#nullable` 预处理器指令将设置可为空注释上下文和可为空警告上下文 。 此指令控制是否可为空注释是否有效，以及是否给出为 Null 性警告。 每个上下文要么处于已禁用状态，要么处于已启用状态 。

通过将 Nullable 元素添加到 `PropertyGroup` 元素中，可在项目级别（C# 源代码之外）指定这两个上下文。 `#nullable` 指令控制注释和警告上下文，并优先于项目级设置。 指令会设置其控制的上下文，直到另一个指令替代它，或直到源文件结束为止。

指令的效果如下所示：

- `#nullable disable`：将可为空注释和警告上下文设置为“已禁用”。
- `#nullable enable`：将可为空注释和警告上下文设置为“已启用”。
- `#nullable restore`：将可为空注释和警告上下文还原为项目设置。
- `#nullable disable annotations`：将可为空注释上下文设置为“已禁用”。
- `#nullable enable annotations`：将可为空注释上下文设置为“已启用”。
- `#nullable restore annotations`：将可为空注释上下文还原为项目设置。
- `#nullable disable warnings`：将可为空警告上下文设置为“已禁用”。
- `#nullable enable warnings`：将可为空警告上下文设置为“已启用”。
- `#nullable restore warnings`：将可为空警告上下文还原为项目设置。

条件编译

使用四个预处理器指令来控制条件编译：

- `#if`：打开条件编译，其中仅在定义了指定的符号时才会编译代码。
- `#elif`：关闭前面的条件编译，并基于是否定义了指定的符号打开一个新的条件编译。
- `#else`：关闭前面的条件编译，如果没有定义前面指定的符号，打开一个新的条件编译。
- `#endif`：关闭前面的条件编译。

仅在定义指定的符号时或者在使用 `!` not 运算符时未定义指定的符号时，C# 编译器才编译 `#if` 指令和 `#endif` 指令之间的代码。 与 C 和 C++ 不同，不能将数字值分配给符号。 C# 中的 `#if` 语句是布尔值，且仅测试是否已定义该符号。 例如，定义 `DEBUG` 时将编译以下代码：

C#

```
#if DEBUG
    Console.WriteLine("Debug version");
#endif
```

未定义 `MYTEST` 时，将编译以下代码：

C#

```
#if !MYTEST
    Console.WriteLine("MYTEST is not defined");
#endif
```

可以使用运算符 `==`（相等）和 `!=`（不相等）来测试 bool 值是 `true` 还是 `false`。 `true` 表示定义该符号。 语句 `#if DEBUG` 具有与 `#if (DEBUG == true)` 相同的含义。 可以使用 &&（and）、||（or）和 !（not）运算符来计算是否已定义多个符号。 还可以用括号对符号和运算符进行分组。

下面是一个复杂指令，允许代码利用较新的 .NET 功能，同时保持向后兼容。 例如，假设你在代码中使用 NuGet 包，但该包仅支持 .NET 6（及更高版本）和 .NET Standard 2.0（及更高版本）：

C#

```
#if (NET6_0_OR_GREATER || NETSTANDARD2_0_OR_GREATER)
    Console.WriteLine("Using .NET 6+ or .NET Standard 2+ code.");
#else
    Console.WriteLine("Using older code that doesn't support the above .NET
versions.");
#endif
```

`#if` 以及 `#else`、`#elif`、`#endif`、`#define` 和 `#undef` 指令，允许基于是否存在一个或多个符号包括或排除代码。 条件编译在编译调试版本的代码或编译特定配置的代码时会很有用。

以 `#if` 指令开头的条件指令必须以 `#endif` 指令显式终止。 `#define` 允许你定义一个符号。 通过将该符号用作传递给 `#if` 指令的表达式，该表达式的计算结果为 `true`。 还可以通过 DefineConstants 编译器选项来定义符号。 可以通过 `#undef` 取消定义符号。 使

用 `#define` 创建的符号的作用域是在其中定义它的文件。 使用 DefineConstants 或 `#define` 定义的符号与具有相同名称的变量不冲突。 也就是说，变量名称不应传递给预处理器指令，且符号仅能由预处理器指令评估。

`#elif` 可以创建复合条件指令。 如果之前的 `#if` 和任何之前的可选 `#elif` 指令表达式的值都不为 `true`，则计算 `#elif` 表达式。 如果 `#elif` 表达式计算结果为 `true`，编译器将计算 `#elif` 和下一条件指令间的所有代码。 例如：

C#

```
#define VC7
//...
#if DEBUG
    Console.WriteLine("Debug build");
#elif VC7
    Console.WriteLine("Visual Studio 7");
#endif
```

`#else` 允许创建复合条件指令，因此，如果先前 `#if` 或（可选） `#elif` 指令中的任何表达式的计算结果都不是 `true`，则编译器将对介于 `#else` 和下一个 `#endif` 之间的所有代码进行求值。 `#endif`(#endif) 必须是 `#else` 之后的下一个预处理器指令。

`#endif` 指定条件指令的末尾，以 `#if` 指令开头。

此外，生成系统还会感知表示 SDK 样式项目中不同目标框架的预定义预处理器符号。 在创建可以面向多个 .NET 版本的应用程序时，这些符号会很有用。

展开表

目标框架	符号	其他符号（在 .NET 5+ SDK 中可用）	平台符号（仅在指定特定于 OS 的 TFM 时可用）
.NET Framework			
.NET Standard			
.NET 5 及更高版本（和 .NET Core）			`[OS][version]`（例如，`IOS15_1`），`[OS][version]_OR_GREATER`（例如，`IOS15_1_OR_GREATER`）

备注

- 无论目标版本是什么，都将定义无版本符号。

- 仅针对目标版本定义特定于版本的符号。
- 为目标版本和所有早期版本定义 `<framework>_OR_GREATER` 符号。 例如，如果针对 .NET Framework 2.0，则会定义以下符号： `NET20`、 `NET20_OR_GREATER`、 `NET11_OR_GREATER` 和 `NET10_OR_GREATER`。
- `NETSTANDARD<x>_<y>_OR_GREATER` 符号仅针对 .NET Standard 目标定义，而不适用于实现 .NET Standard 的目标，例如 .NET Core 和 .NET Framework。
- 它们与 MSBuild TargetFramework 属性和 NuGet 使用的目标框架名字对象 (TFM) 不同。

> **备注**
>
> 对于传统的非 SDK 样式的项目，必须通过项目的属性页面在 Visual Studio 中为不同目标框架手动配置条件编译符号。

其他预定义符号包括 `DEBUG` 和 `TRACE` 常数。 你可以使用 `#define` 替代项目的值集。 例如，会根据生成配置属性（“调试”或者“发布”模式）自动设置 DEBUG 符号。

下例显示如何在文件上定义 `MYTEST` 符号，然后测试 `MYTEST` 和 `DEBUG` 符号的值。 此示例的输出取决于是在“调试”还是“发布”配置模式下生成项目 。

C#

```
#define MYTEST
using System;
public class MyClass
{
    static void Main()
    {
#if (DEBUG && !MYTEST)
        Console.WriteLine("DEBUG is defined");
#elif (!DEBUG && MYTEST)
        Console.WriteLine("MYTEST is defined");
#elif (DEBUG && MYTEST)
        Console.WriteLine("DEBUG and MYTEST are defined");
#else
        Console.WriteLine("DEBUG and MYTEST are not defined");
#endif
    }
}
```

下例显示如何针对不同的目标框架进行测试，以便在可能时使用较新的 API：

C#

```
public class MyClass
{
    static void Main()
    {
#if NET40
        WebClient _client = new WebClient();
#else
        HttpClient _client = new HttpClient();
#endif
    }
    //...
}
```

定义符号

使用以下两个预处理器指令来定义或取消定义条件编译的符号：

- `#define`：定义符号。
- `#undef`：取消定义符号。

使用 `#define` 来定义符号。 将符号用作传递给 `#if` 指令的表达式时，该表达式的计算结果为 `true`，如以下示例所示：

C#

```
#define VERBOSE

#if VERBOSE
   Console.WriteLine("Verbose output version");
#endif
```

> ⓘ **备注**
>
> 在 C# 中，应使用 const 关键字定义基元常量。 `const` 声明会创建一个在运行时无法修改的 `static` 成员。 `#define` 指令不能用于声明常量值，而在 C 和 C++ 中通常可以这样做。 如果具有多个此类常量，请考虑创建一个单独的“常量”类来容纳它们。

符号可用于指定编译的条件。 可通过 `#if` 或 `#elif` 测试符号。 还可以使用 ConditionalAttribute 来执行条件编译。 可以定义符号，但不能为符号分配值。 文件中必须先出现 `#define` 指令，才能使用并非同时也是预处理器指令的任何指示。 还可以通过 DefineConstants 编译器选项来定义符号。 可以通过 `#undef` 取消定义符号。

定义区域

可以使用以下两个预处理器指令来定义可在大纲中折叠的代码区域：

- `#region`：启动区域。
- `#endregion`：结束区域。

利用 `#region`，可以指定在使用代码编辑器的大纲功能时可展开或折叠的代码块。 在较长的代码文件中，折叠或隐藏一个或多个区域十分便利，这样，可将精力集中于当前处理的文件部分。 下面的示例演示如何定义区域：

C#

```
#region MyClass definition
public class MyClass
{
    static void Main()
    {
    }
}
#endregion
```

`#region` 块必须通过 `#endregion` 指令终止。 `#region` 块不能与 `#if` 块重叠。 但是，可以将 `#region` 块嵌套在 `#if` 块内，或将 `#if` 块嵌套在 `#region` 块内。

错误和警告信息

使用以下指令指示编译器生成用户定义的编译器错误和警告，并控制行信息：

- `#error`：使用指定的消息生成编译器错误。
- `#warning`：使用指定的消息生成编译器警告。
- `#line`：更改用编译器消息输出的行号。

`#error` 可从代码中的特定位置生成 CS1029 用户定义的错误。 例如：

C#

```
#error Deprecated code in this method.
```

备注

编译器以特殊的方式处理 `#error version` 并报告编译器错误 CS8304，消息中包含使用的编译器和语言版本。

`#warning` 允许你从代码中的特定位置生成 CS1030 第一级编译器警告。 例如：

C#

```
#warning Deprecated code in this method.
```

借助 `#line`，可修改编译器的行号及（可选）用于错误和警告的文件名输出。

以下示例演示如何报告与行号相关联的两个警告。 `#line 200` 指令将下一行的行号强制设为 200（尽管默认值为 #6）；在执行下一个 `#line` 指令前，文件名都会报告为"特殊"。 `#line default` 指令将行号恢复至默认行号，这会对上一指令重新编号的行进行计数。

C#

```
class MainClass
{
    static void Main()
    {
#line 200 "Special"
        int i;
        int j;
#line default
        char c;
        float f;
#line hidden // numbering not affected
        string s;
        double d;
    }
}
```

编译产生以下输出：

控制台

```
Special(200,13): warning CS0168: The variable 'i' is declared but never used
Special(201,13): warning CS0168: The variable 'j' is declared but never used
MainClass.cs(9,14): warning CS0168: The variable 'c' is declared but never
used
MainClass.cs(10,15): warning CS0168: The variable 'f' is declared but never
used
MainClass.cs(12,16): warning CS0168: The variable 's' is declared but never
used
```

```
MainClass.cs(13,16): warning CS0168: The variable 'd' is declared but never
used
```

可在生成过程的自动、中间步骤中使用 `#line` 指令。 例如，如果已从原始源代码文件中删除行，但仍希望编译器基于文件中的原始行号生成输出，可在删除行后，使用 `#line` 来模拟原始行号。

`#line hidden` 指令能对调试程序隐藏连续行，当开发者逐行执行代码时，介于 `#line hidden` 和下一 `#line` 指令（假设它不是其他 `#line hidden` 指令）间的任何行都将被跳过。 还可通过此选项允许 ASP.NET 区分用户定义和计算机生成的代码。 虽然此功能主要用于 ASP.NET，但可能更多的源生成器会利用此功能。

`#line hidden` 指令不影响错误报告中的文件名或行号。 也就是说，如果编译器在隐藏块中发现错误，编译器将报告错误的当前文件名和行号。

`#line filename` 指令可指定要在编译器输出中显示的文件名。 默认情况下，将使用源代码文件的实际名称。 该文件名必须以双引号 ("") 引起来，且必须位于行号之后。

从 C# 10 开始，可以使用一种新形式的 `#line` 指令：

C#

```
#line (1, 1) - (5, 60) 10 "partial-class.cs"
/*34567*/int b = 0;
```

这种形式的组件包括：

- `(1, 1)`：指令后面行上的第一个字符的起始行和列。 在此示例中，下一行将报告为第 1 行第 1 列。
- `(5, 60)`：标记区域的结束行和列。
- `10`：将使 `#line` 指令生效的列偏移量。 在此示例中，第 10 列将报告为第 1 列。 这就是声明 `int b = 0;` 开始的位置。 此字段可选。 如果省略了该字段，则指令将对第 1 列生效。
- `"partial-class.cs"`：输出文件的名称。

前面的示例将生成以下警告：

.NET CLI

```
partial-class.cs(1,5,1,6): warning CS0219: The variable 'b' is assigned but
its value is never used
```

重新映射后，变量 `b` 位于 `partial-class.cs` 文件第 1 行的第 6 个字符处。

特定于域的语言 (DSL) 通常使用此格式来提供从源文件到生成的 C# 输出的更好的映射。 此扩展 `#line` 指令的最常见用途是将生成的文件中出现的警告或错误重新映射到原始源。 例如，请考虑以下 razor 页：

razor

```
@page "/"
Time: @DateTime.NowAndThen
```

属性 `DateTime.Now` 被错误地键入为 `DateTime.NowAndThen`。 在 `page.g.cs` 中，为此 razor 代码片段生成的 C# 如下所示：

C#

```
  _builder.Add("Time: ");
#line (2, 6) - (2, 27) 15 "page.razor"
  _builder.Add(DateTime.NowAndThen);
```

前面的代码片段的编译器输出如下：

.NET CLI

```
page.razor(2, 2, 2, 27)error CS0117: 'DateTime' does not contain a
definition for 'NowAndThen'
```

`page.razor` 中的第 2 行的第 6 列是文本 `@DateTime.NowAndThen` 的开始位置。 指令中的 `(2, 6)` 已指出这一点。 `@DateTime.NowAndThen` 的结束位置在第 2 行的第 27 列。 指令中的 `(2, 27)` 已指出这一点。 `DateTime.NowAndThen` 的文本从 `page.g.cs` 的第 15 列开始。 指令中的 `15` 已指出这一点。 将所有参数放在一起，编译器将在其在 `page.razor` 中的位置报告错误。 开发人员可直接导航到该错误在源代码（而不是生成的源）中的位置。

若要查看此格式的更多示例，请参阅示例部分中的功能规范。

杂注

`#pragma` 为编译器给出特殊指令以编译它所在的文件。 这些指令必须受编译器支持。 换句话说，不能使用 `#pragma` 创建自定义的预处理指令。

- #pragma warning：启用或禁用警告。
- #pragma checksum：生成校验和。

C#

```
#pragma pragma-name pragma-arguments
```

其中 `pragma-name` 是可识别 pragma 的名称，`pragma-arguments` 是特定于 pragma 的参数。

#pragma warning

`#pragma warning` 可以启用或禁用特定警告。

C#

```
#pragma warning disable warning-list
#pragma warning restore warning-list
```

其中 `warning-list` 是以逗号分隔的警告编号的列表。 “CS”前缀是可选的。 未指定警告编号时，`disable` 会禁用所有警告，`restore` 会启用所有警告。

> **备注**
>
> 若要在 Visual Studio 中查找警告编号，请生成项目，然后在“输出”窗口中查找警告编号。

`disable` 从源文件的下一行开始生效。 警告会在后面的 `restore` 行上还原。 如果文件中没有 `restore`，则在同一编译中任何之后文件的第一行，警告将还原为其默认状态。

C#

```
// pragma_warning.cs
using System;

#pragma warning disable 414, CS3021
[CLSCompliant(false)]
public class C
{
    int i = 1;
    static void Main()
    {
    }
}
#pragma warning restore CS3021
[CLSCompliant(false)]  // CS3021
public class D
{
    int i = 1;
    public static void F()
```

```
    {
    }
}
```

#pragma checksum

生成源文件的校验和以帮助调试 ASP.NET 页面。

C#

```
#pragma checksum "filename" "{guid}" "checksum bytes"
```

其中，`"filename"` 是需要监视更改或更新的文件的名称，`"{guid}"` 是哈希算法的全局唯一标识符 (GUID)，`"checksum_bytes"` 是表示校验和字节的十六进制数字的字符串。 必须是偶数个十六进制数字。 奇数个数字会导致编译时警告出现，且指令遭忽略。

Visual Studio 调试器使用校验和确保它可始终找到正确的源。 编译器为源文件计算校验和，然后将输出发出到程序数据库 (PDB) 文件。 调试器随后使用 PDB 针对它为源文件计算的校验和进行比较。

此解决方案不适用于 ASP.NET 项目，因为计算出的校验和是针对生成的源文件，而不是 .aspx 文件。 为解决此问题，`#pragma checksum` 为 ASP.NET 页面提供校验和支持。

在 Visual C# 中创建 ASP.NET 项目时，生成的源文件包含 .aspx 文件（从该文件生成源）的校验和。 编译器随后将此信息写入 PDB 文件中。

如果编译器在文件中没有找到 `#pragma checksum` 指令，它将计算校验和，并将该值写入 PDB 文件。

C#

```
class TestClass
{
    static int Main()
    {
        #pragma checksum "file.cs" "{406EA660-64CF-4C82-B6F0-42D48172A799}"
"ab007f1d23d9" // New checksum
    }
}
```

C# 编译器选项

项目 • 2024/04/11

本节介绍 C# 编译器解释的选项。 选项根据其控制的内容（例如语言功能、代码生成和输出）分组到不同的文章中。 使用目录来浏览它们。

如何设置选项

可以通过两种不同的方法设置 .NET 项目中的编译器选项：

- **在你的 *.csproj 文件中**

  可以 XML 格式为 *.csproj 文件中的任何编译器选项添加 MSBuild 属性。 属性名称与编译器选项相同。 用属性的值设置编译器选项的值。 例如，下面的项目文件段设置 `LangVersion` 属性。

  ```xml
  <PropertyGroup>
    <LangVersion>preview</LangVersion>
  </PropertyGroup>
  ```

  有关项目文件中设置选项的详细信息，请参阅适用于 .NET SDK 项目的 MSBuild 属性一文。

- ***使用 Visual Studio 属性页***

  Visual Studio 提供了属性页，可用于编辑生成属性。 若要了解有关详细信息，请参阅管理项目和解决方案属性 - Windows 或管理项目和解决方案属性 - Mac。

.NET Framework 项目

> **ⓘ 重要**
>
> 本部分仅适用于 .NET Framework 项目。

除上述机制以外，还可以使用两种附加方法为 .NET Framework 项目设置编译器选项：

- .NET Framework 项目的命令行参数：.NET Framework 项目使用 csc.exe 而不是 `dotnet build` 生成项目。 可以为 .NET Framework 项目指定 csc.exe 的命令行参数。

- 已编译的 ASP.NET 页面：.NET Framework 项目使用 web.config 文件的一部分来编译页面。 对于新的生成系统和 ASP.NET Core 项目，将从项目文件中设置选项。

某些编译器选项的单词从 csc.exe 和 .NET Framework 项目更改为新的 MSBuild 系统。 本部分使用的是新语法。 每个页面顶部同时列出了这两个版本。 对于 csc.exe，所有参数都会在选项和冒号后列出。 例如，`-doc` 选项将为：

控制台

```
-doc:DocFile.xml
```

通过在命令提示符处键入 C# 编译器的可执行文件名称 (csc.exe)，可调用该编译器。

对于 .NET Framework 项目，还可以从命令行运行 csc.exe。 每个编译器选项均有两种形式： **-option** 和 **/option**。 在 .NET Framework Web 项目中，在 web.config 文件中指定用于编译代码隐藏的选项。 有关详细信息，请参阅 <compiler> 元素。

如果使用“Visual Studio 开发人员命令提示”窗口，系统将设置所有必需的环境变量。 有关如何访问此工具的信息，请参阅 Visual Studio 开发人员命令提示。

csc.exe 可执行文件通常位于 Windows 目录下的 Microsoft.NET\Framework\<Version> 文件夹中。 根据每台计算机上的具体配置，此位置可能有所不同。 如果计算机上安装了不止一个版本的 .NET Framework，你将发现此文件的多个版本。 有关此类安装的详细信息，请参阅如何：确定安装的 .NET Framework 版本。

反馈

此页面是否有帮助？ 是 否

提供产品反馈

用于语言功能规则的 C# 编译器选项

项目 • 2024/09/17

以下选项控制编译器如何解释语言功能。 新的 MSBuild 语法以**粗体**显示。 旧的 *csc.exe* 语法以 `code style` 显示。

- **CheckForOverflowUnderflow** / `-checked`：生成溢出检查。
- **AllowUnsafeBlocks** / `-unsafe`：允许“不安全”代码。
- **DefineConstants** / `-define`：定义条件编译符号。
- **LangVersion** / `-langversion`：指定语言版本，如 `default`（最新主版本）或 `latest`（最新版本，包括次要版本）。
- **Nullable** / `-nullable`：启用可为空上下文或可为空警告。

> **备注**
>
> 有关为项目配置这些选项的详细信息，请参阅**编译器选项**。

CheckForOverflowUnderflow

CheckForOverflowUnderflow选项控制在整数算术溢出的情况下定义程序行为的默认溢出检查上下文。

```xml
<CheckForOverflowUnderflow>true</CheckForOverflowUnderflow>
```

CheckForOverflowUnderflow 为 `true` 时，默认上下文为已检查上下文，并且启用了溢出检查；否则，默认上下文为未检查的上下文。 此选项的默认值为 `false`，即禁用溢出检查。

还可以使用 `checked` 和 `unchecked` 语句显式控制代码部分的溢出检查上下文。

有关溢出检查上下文如何影响操作以及哪些操作受到影响的信息，请参阅关于 checked 和 unchecked 语句的文章。

AllowUnsafeBlocks

AllowUnsafeBlocks 编译器选项允许使用 unsafe 关键字的代码进行编译。 此选项的默认值为`false`，表示不允许不安全代码。

XML

```
<AllowUnsafeBlocks>true</AllowUnsafeBlocks>
```

有关不安全代码的详细信息，请参阅不安全代码和指针。

DefineConstants

DefineConstants 选项将定义程序中所有源代码文件的符号。

XML

```
<DefineConstants>name;name2</DefineConstants>
```

此选项指定要定义的一个或多个符号的名称。 **DefineConstants** 选项具有与 #define 预处理器指令相同的效果，只不过编译器选项对项目中的所有文件都有效。 符号在源文件中保持已定义状态，直到源文件中的 #undef 指令删除该定义。 当你使用 `-define` 选项时，一个文件中的 `#undef` 指令不影响项目中的其他源代码文件。 可以将由此选项创建的符号同 #if、#else、#elif 和 #endif 一起使用，对源文件进行条件编译。 C# 编译器本身不定义源代码中使用的符号或宏；所有符号定义必须都是用户定义的。

> **备注**
>
> 同 C++ 等语言一样，C# `#define` 指令不允许为符号赋值。 例如，`#define` 不能用于创建宏或定义常数。 如果需要定义一个常数，请使用 `enum` 变量。 若要创建 C++ 风格的宏，请考虑泛型等替代项。 由于宏非常容易出错，所以 C# 不允许使用宏，但提供了更安全的替代项。

LangVersion

C# 编译器的默认语言版本依赖于应用程序的目标框架以及所安装的 SDK 或 Visual Studio 的版本。 这些规则是在 C# 语言版本控制中定义的。

> **警告**
>
> 不建议将 `LangVersion` 元素设置为 `latest`。 `latest` 设置意味着已安装的编译器将使用其最新版本。 这可以在计算机之间进行更改，从而使得生成不可靠。 此外，它还支持可能需要当前 SDK 中不包含的运行时或库功能的语言功能。

LangVersion 选项使编译器仅接受指定 C# 语言规范中包含的语法，例如：

```xml
<LangVersion>9.0</LangVersion>
```

以下为有效值：

展开表

Value	含义
`preview`	编译器接受最新预览版中的所有有效语言语法。
`latest`	编译器接受最新发布的编译器版本（包括次要版本）中的语法。
`latestMajor` 或 `default`	编译器接受最新发布的编译器主要版本中的语法。
`13.0`	编译器只接受 C# 13 或更低版本中所含的语法。
`12.0`	编译器只接受 C# 12 或更低版本中所含的语法。
`11.0`	编译器只接受 C# 11 或更低版本中包含的语法。
`10.0`	编译器只接受 C# 10 或更低版本中所含的语法。
`9.0`	编译器只接受 C# 9 或更低版本中所含的语法。
`8.0`	编译器只接受 C# 8.0 或更低版本中所含的语法。
`7.3`	编译器只接受 C# 7.3 或更低版本中所含的语法。
`7.2`	编译器只接受 C# 7.2 或更低版本中所含的语法。
`7.1`	编译器只接受 C# 7.1 或更低版本中所含的语法。
`7`	编译器只接受 C# 7.0 或更低版本中所含的语法。
`6`	编译器只接受 C# 6.0 或更低版本中所含的语法。
`5`	编译器只接受 C# 5.0 或更低版本中所含的语法。
`4`	编译器只接受 C# 4.0 或更低版本中所含的语法。
`3`	编译器只接受 C# 3.0 或更低版本中所含的语法。
`ISO-2` 或 `2`	编译器只接受 ISO/IEC 23270:2006 C# (2.0) 中所含的语法。

Value	含义
`ISO-1` 或 `1`	编译器只接受 ISO/IEC 23270:2003 C# (1.0/1.2) 中所含的语法。

注意事项

- 若要确保项目使用为目标框架推荐的默认编译器版本，请不要使用 LangVersion 选项。 可更新目标框架以访问更新的语言功能。
- 使用 `default` 值指定 **LangVersion** 与省略 **LangVersion** 选项不同。 指定 `default` 会使用编译器支持的语言的最新版本，而不考虑目标框架。 例如，如果未指定 LangVersion，则通过 Visual Studio 版本 17.6 生成面向 .NET 6 的项目将使用 C# 10，但如果 LangVersion 设置为 `default`，则使用 C# 11。
- C# 应用程序引用的元数据不受 **LangVersion** 编译器选项约束。
- 每个版本的 C# 编译器都包含语言规范的扩展，因此 **LangVersion** 不提供早期版本编译器的同等功能。
- 虽然 C# 版本更新通常与主要的 .NET 版本一致，但新的语法和功能不一定会绑定到该特定的 Framework 版本。 每项具体功能都有自己的最小 .NET API 或公共语言运行时要求，这些要求通过包括 NuGet 包或其他库允许功能在下层框架上运行。
- 无论使用哪一项 LangVersion 设置，请使用当前版本的公共语言运行时来创建 .exe 或 .dll。 友元程序集和 ModuleAssemblyName 是一个例外，它们在 -langversion:ISO-1 下工作。

若要了解指定 C# 语言版本的其他方式，请参阅 C# 语言版本控制。

有关如何以编程方式设置此编译器选项的信息，请参阅 LanguageVersion。

C# 语言规范

展开表

Version	链接	说明
C# 8.0 和更高版本	下载 PDF	C# 语言规范版本 7：.NET Foundation
C# 7.3	下载 PDF	标准 ECMA-334 第 7 版
C# 6.0	下载 PDF	标准 ECMA-334 第 6 版
C# 5.0	下载 PDF	标准 ECMA-334 第 5 版

Version	链接	说明
C# 3.0	下载 DOC	C# 语言规范版本 3.0：Microsoft Corporation
C# 2.0	下载 PDF	标准 ECMA-334 第 4 版
C# 1.2	下载 DOC	标准 ECMA-334 第 2 版
C# 1.0	下载 DOC	标准 ECMA-334 第 1 版

支持所有语言功能所需的最低 SDK 版本

下表列出了支持相应语言版本的 C# 编译器的 SDK 的最低版本：

展开表

C# 版本	最低 SDK 版本
C# 12	Microsoft Visual Studio/生成工具 2022，版本 17.8 或 .NET 8 SDK
C# 11	Microsoft Visual Studio/生成工具 2022，版本 17.4 或 .NET 7 SDK
C# 10	Microsoft Visual Studio/生成工具 2022 或 .NET 6 SDK
C# 9.0	Microsoft Visual Studio/生成工具 2019 版本 16.8 或 .NET 5 SDK
C# 8.0	Microsoft Visual Studio/生成工具 2019，版本 16.3 或 .NET Core 3.0 SDK
C# 7.3	Microsoft Visual Studio/生成工具 2017，版本 15.7
C# 7.2	Microsoft Visual Studio/生成工具 2017，版本 15.5
C# 7.1	Microsoft Visual Studio/生成工具 2017，版本 15.3
C# 7.0	Microsoft Visual Studio/生成工具 2017
C# 6	Microsoft Visual Studio/生成工具 2015
C# 5	Microsoft Visual Studio/生成工具 2012 或捆绑的 .NET Framework 4.5 编译器
C# 4	Microsoft Visual Studio/生成工具 2010 或捆绑的 .NET Framework 4.0 编译器
C# 3	Microsoft Visual Studio/生成工具 2008 或捆绑的 .NET Framework 3.5 编译器
C# 2	Microsoft Visual Studio/生成工具 2005 或捆绑的 .Net Framework 2.0 编译器
C# 1.0/1.2	Microsoft Visual Studio/生成工具 .NET 2002 或捆绑的 .NET Framework 1.0 编译器

Nullable

使用 **Nullable** 选项可指定可为空上下文。 可以使用 `<Nullable>` 标记在项目的配置中设置它：

```xml
<Nullable>enable</Nullable>
```

参数必须为以下项之一：`enable`、`disable`、`warnings` 或 `annotations`。 `enable` 参数启用可为空上下文。 指定 `disable` 将禁用可为空上下文。 如果指定`warnings`参数，会启用可为空警告上下文。 如果指定`annotations`参数，会启用可为空注释上下文。 有关可为空上下文的文章中对这些值进行了描述和说明。 可以在有关可为空迁移策略的文章中详细了解在现有代码库中启用可为空引用类型所涉及的任务。

> **备注**
>
> 未设置任何值时，将应用默认值 `disable`，但默认提供 .NET 6 模板，其 **Nullable** 值设置为 `enable`。

流分析用于在可执行代码中推断变量的为空性。 推断出的变量的为空性与变量声明的为空性无关。 即使有条件地省略方法调用，也会对其进行分析。 例如，发布模式下的 Debug.Assert。

采用以下特性进行注释的方法调用也将影响流分析：

- 简单的前提条件：AllowNullAttribute 和 DisallowNullAttribute
- 简单的后置条件：MaybeNullAttribute 和 NotNullAttribute
- 有条件后置条件：MaybeNullWhenAttribute 和 NotNullWhenAttribute
- DoesNotReturnIfAttribute（例如 Debug.Assert 的 `DoesNotReturnIf(false)`）和 DoesNotReturnAttribute
- NotNullIfNotNullAttribute
- 成员后置条件：MemberNotNullAttribute(String) 和 MemberNotNullAttribute(String[])

> **重要**
>
> 全局可为空上下文不适用于生成的代码文件。 无论此设置如何，都会针对标记为“已生成”的任何源文件*禁用*可为空上下文。 可采用四种方法将文件标记为“已生成”：
>
> 1. 在 .editorconfig 中，在应用于该文件的部分中指定 `generated_code = true`。
> 2. 将 `<auto-generated>` 或 `<auto-generated/>` 放在文件顶部的注释中。 它可以位于该注释中的任意行上，但注释块必须是该文件中的第一个元素。

3. 文件名以 *TemporaryGeneratedFile_* 开头
4. 文件名用以 *.designer.cs*、*.generated.cs*、*.g.cs* 或 *.g.i.cs* 结尾。

生成器可以选择使用 **#nullable** 预处理器指令。

用于控制编译器输出的 C# 编译器选项

项目 • 2023/05/10

以下选项控制编译器输出生成。

MSBuild	csc.exe	说明
DocumentationFile	`-doc:`	从 `///` 注释生成 XML 文档文件。
OutputAssembly	`-out:`	指定输出程序集文件。
PlatformTarget	`-platform:`	指定目标平台 CPU。
ProduceReferenceAssembly	`-refout:`	生成引用程序集。
TargetType	`-target:`	指定输出程序集的类型。

DocumentationFile

使用 DocumentationFile 选项可以在 XML 文件中放置文档注释。 若要详细了解如何记录代码，请参阅建议的文档注释标记。 该值指定输出 XML 文件的路径。 此 XML 文件包含编译的源代码文件中的注释。

XML

```
<DocumentationFile>path/to/file.xml</DocumentationFile>
```

包含 Main 或顶级语句的源代码文件首先输出到 XML 中。 通常需要将生成的 .xml 文件与 IntelliSense 一起使用。 .xml 文件名必须与程序集名称相同。 .xml 文件必须与程序集位于同一目录中。 在 Visual Studio 项目中引用程序集时，也会找到 .xml 文件。 若要详细了解如何生成代码注释，请参阅提供代码注释。 除非用 <TargetType:Module> 进行编译，否则 `file` 将包含 `<assembly>` 和 `</assembly>` 标记，用于指定包含输出文件的程序集清单的文件名。 例如，请参阅如何使用 XML 文档功能。

> **备注**
>
> DocumentationFile 选项适用于项目中的所有文件。 若要禁用与特定文件或一段代码的文档注释相关的警告，请使用 **#pragma 警告**。

此选项可用于任何 .NET SDK 样式的项目。 有关详细信息，请参阅 DocumentationFile 属性。

OutputAssembly

OutputAssembly 选项指定输出文件的名称。 输出路径指定放置编译器输出的文件夹。

```
XML

<OutputAssembly>folder</OutputAssembly>
```

指定想要创建的文件的完整名称和扩展名。 如果不指定输出文件的名称，MSBuild 将使用项目的名称指定输出程序集的名称。 旧样式的项目使用以下规则：

- .exe 将采用包含 `Main` 方法或顶级语句的源代码文件中的名称。
- .dll 或者 .netmodule 将从第一个源代码文件中获取其名称。

在编译时生成的任何模块都将成为与编译时生成的程序集关联的文件。 使用 ildasm.exe 查看程序集清单，了解关联文件。

为使 exe 成为友元程序集的目标，OutputAssembly 编译器选项是必需的。

PlatformTarget

指定 CLR 的哪个版本可以运行程序集。

```
XML

<PlatformTarget>anycpu</PlatformTarget>
```

- anycpu（默认值）将程序集编译成可在任意平台上运行。 您的应用程序将尽可能作为 64 位进程运行；当只有 32 位模式可用时，才会回退到 32 位。
- anycpu32bitpreferred 将程序集编译成可在任意平台上运行。 在同时支持 64 位和 32 位应用程序的系统上，您的应用程序将以32 位模式运行。 只能为面向 .NET Framework 4.5 或更高版本的项目指定此选项。
- ARM 将程序集编译成可以在具有高级 RISC 计算机 (ARM) 处理器的计算机上运行。
- ARM64 编译程序集以在由 64 位 CLR 在具有支持 A64 指令集的高级 RISC 计算机 (ARM) 处理器的计算机上运行。
- x64 将程序集编译成可由支持 AMD64 或 EM64T 指令集的计算机上的 64 位 CLR 运行。
- x86 将程序集编译成可由 32 位、x86 可兼容 CLR 运行。
- Itanium 将程序集编译成可由配有 Itanium 处理器的计算机上的 64 位 CLR 运行。

在 64 位 Windows 操作系统上：

- 用 x86 编译的程序集将在 WOW64 下运行的 32 位 CLR 上执行。

- 用 anycpu 编译的 DLL 将在加载它的进程所在的同一 CLR 上执行。
- 用 anycpu 编译的可执行文件将在 64 位 CLR 上执行。
- 用 anycpu32bitpreferred 编译的可执行文件将在 32 位 CLR 上执行。

anycpu32bitpreferred 设置只对可执行文件 (.EXE) 有效，并且需要 .NET Framework 4.5 或更高版本。 有关开发 Windows 64 位操作系统上运行的应用程序的详细信息，请参阅 64 位应用程序。

在 Visual Studio 中，可以从项目的“生成”属性页中设置 PlatformTarget 选项。

对于 .NET Core 和 .NET 5 及更高版本，anycpu 的行为有一些额外的细微差别。 设置 anycpu 后，请发布应用并将其与 x86 `dotnet.exe` 或 x64 `dotnet.exe` 一起执行。 对于自包含应用，`dotnet publish` 步骤会将配置 RID 的可执行文件打包。

ProduceReferenceAssembly

ProduceReferenceAssembly 选项控制编译器是否生成引用程序集。

```xml
<ProduceReferenceAssembly>true</ProduceReferenceAssembly>
```

引用程序集是一种特殊类型的程序集，它只包含表示库的公共 API 外围应用所需的最少元数据量。 它们包括在生成工具中引用程序集时所需的所有成员的声明。 引用程序集不包括所有成员实现以及私有成员的声明。 这些成员对其 API 协定没有明显影响。 有关详细信息，请参阅 .NET 指南中的引用程序集。

ProduceReferenceAssembly 和 ProduceOnlyReferenceAssembly 选项是互斥的。

通常不需要直接处理引用程序集文件。 默认情况下，引用程序集是在中间路径的 `ref` 子文件夹中生成的（即 `obj/ref/`）。 若要在输出目录（即 `bin/ref/`）下生成它们，请在你的项目中将 `ProduceReferenceAssemblyInOutDir` 设置为 `true`。

默认情况下，.NET SDK 6.0.200 进行了更改，将引用程序集从输出目录移动到了中间目录。

TargetType

TargetType 编译器选项可指定为以下形式之一：

- library：用于创建代码库。 library 是默认值。
- exe：用于创建 .exe 文件。

- module：用于创建模块。
- winexe：用于创建 Windows 程序。
- winmdobj：用于创建 .winmdobj 中间文件。
- appcontainerexe：用于为 Windows 8.x 应用商店应用创建 .exe 文件。

> **备注**
>
> 对于 .NET Framework 目标，除非指定 module，否则，此选项会导致将 .NET Framework 程序集清单放入输出文件中。 有关详细信息，请参阅 .NET **中的程序集**和**公共属性**。

XML

```
<TargetType>library</TargetType>
```

编译器每次编译只创建一个程序集清单。 关于编译中所有文件的信息都放在程序集清单中。 在命令行生成多个输出文件时，只能创建一个程序集清单，且该清单必须放置在命令行上指定的第一个输出文件中。

如果你创建了一个程序集，则可以用 CLSCompliantAttribute 属性指示全部或部分代码是符合 CLS 的。

库

library 选项会导致编译器创建动态链接库 (DLL) 而不是可执行文件 (EXE)。 将创建具有 .dll 扩展名的 DLL。 除非使用 OutputAssembly 选项指定，否则输出文件的名称采用第一个输入文件的名称。 生成 .dll 文件时，不需要 Main 方法。

exe

exe 选项会导致编译器创建一个可执行的 (EXE) 控制台应用程序。 将创建扩展名为 .exe 的可执行文件。 使用 winexe 创建 Windows 程序可执行文件。 除非使用 OutputAssembly 选项进行指定，否则输出文件的名称将采用包含入口点（Main 方法或顶级语句）的输入文件的名称。 编译为 .exe 文件的源代码文件中只需要一个入口点。 如果代码具有多个附带 `Main` 方法的类，则可使用 StartupObject 编译器选项指定包含 `Main` 方法的类。

name

此选项会导致编译器不生成程序集清单。 默认情况下，使用此选项编译时所创建的输出文件具有扩展名 .netmodule。 .NET 运行时无法加载没有程序集清单的文件。 但是，此类文件可以通过 AddModules 合并到程序集的程序集清单中。 如果在一次编译中创建了多个模块，某个模块中的内部类型将适用于编译中的其他模块。 如果一个模块中的代码引用另一模块中的 `internal` 类型，则两个模块必须通过 AddModules 合并到一个程序集清单中。 Visual Studio 开发环境中不支持创建模块。

winexe

winexe 选项会导致编译器创建可执行的 (EXE) Windows 程序。 将创建扩展名为 .exe 的可执行文件。 Windows 程序是通过 .NET 库或 Windows API 提供用户界面的程序。 使用 exe 创建控制台应用程序。 除非使用 OutputAssembly 选项进行指定，否则输出文件的名称将采用包含 Main 方法的输入文件的名称。 编译为 .exe 文件的源代码文件中只需要一个 `Main` 方法。 如果代码具有多个附带 `Main` 方法的类，则可使用 StartupObject 选项指定包含 `Main` 方法的类。

winmdobj

如果使用 winmdobj 选项，则编译器会创建一个 .winmdobj 中间文件，你可以将此文件转换为 Windows 运行时二进制 (.winmd) 文件。 之后，除了托管语言程序外，JavaScript 和 C++ 程序也可以使用该 .winmd 文件。

winmdobj 设置会向编译器发出信号，表示需要中间模块。 随后，.winmdobj 文件可作为 WinMDExp 导出工具的输入，生成 Windows 元数据 (.winmd) 文件。 .winmd 文件既包含原始库的代码，也包含 JavaScript 或 C++ 以及 Windows 运行时使用的 WinMD 元数据的代码。 使用 winmdobj 编译器选项所编译文件的输出只能用作 WimMDExp 导出工具的输入。 .winmdobj 文件本身并没有被直接引用。 除非使用 OutputAssembly 选项，否则输出文件的名称将采用第一个输入文件的名称。 不需要使用 Main 方法。

appcontainerexe

如果使用 appcontainerexe 编译器选项，则编译器会创建一个 Windows 可执行 (.exe) 文件，该文件必须在应用容器中运行。 此选项与 -target:winexe 等效，但专门用于 Windows 8.x 应用商店应用。

为了要求应用在应用容器中运行，此选项在可移植可执行 (PE) 文件中设置了一个位。 设置该位时，如果 CreateProcess 方法尝试在应用容器外启动该可执行文件，就会发生错误。 除非使用 OutputAssembly 选项，否则输出文件的名称将采用包含 Main 方法的输入文件的名称。

可指定输入的 C# 编译器选项

项目 • 2023/05/10

以下选项控制编译器输入。 新的 MSBuild 语法以粗体显示。 旧的 csc.exe 语法以 `code style` 显示。

- **References** / `-reference` 或 `-references`：引用来自一个或多个指定程序集文件的元数据。
- **AddModules** / `-addmodule`：添加模块（将使用 `target:module` 创建的模块添加到此程序集。）
- **EmbedInteropTypes** / `-link`：嵌入指定互操作程序集文件中的元数据。

参考

References 选项使编译器将指定文件中的 public 类型信息导入当前项目，从而使你可从指定的程序集文件中引用元数据。

XML

```
<Reference Include="filename" />
```

`filename` 是包含程序集清单的文件的名称。 若要导入多个文件，请为每个文件加上一个单独的 Reference 元素。 可以定义一个别名作为 Reference 元素的子元素：

XML

```
<Reference Include="filename.dll">
  <Aliases>LS</Aliases>
</Reference>
```

在上面的示例中，`LS` 是表示根命名空间的有效 C# 标识符，该根命名空间将包含程序集 filename.dll 中的所有命名空间。 导入的文件必须包含一个清单。 使用 AdditionalLibPaths 指定一个或多个程序集引用所在的目录。 AdditionalLibPaths 主题还讨论了编译器在其中搜索程序集的目录。 为使编译器可以识别程序集（而不是模块）中的某个类型，需要强制解析此类型，这可以通过定义此类型的实例来完成。 还可通过其他方法为编译器解析程序集中的类型名称：例如，如果从程序集中继承类型，编译器就能识别类型名称。 有时，需要在一个程序集内引用同一组件的两个不同版本。 为此，请在每个文件的 References 元素上使用 Aliases 元素，以区分这两个文件。 此别名将用作组件名的限定符，并解析为其中一个文件中的组件。

> **备注**
>
> 在 Visual Studio 中，请使用“添加引用”命令。 有关详细信息，请参阅 How to: Add or Remove References By Using the Reference Manager。

AddModules

此选项将添加一个模块，该模块通过将 `<TargetType>module</TargetType>` 切换到当前编译进行创建：

XML

```
<AddModule Include=file1 />
<AddModule Include=file2 />
```

其中 `file`、`file2` 是包含元数据的输出文件。 该文件不能包含程序集清单。 若要导入多个文件，请用逗号或分号将文件名隔开。 通过 AddModules 添加的所有模块在运行时必须位于与输出文件相同的目录中。 也就是说，在编译时可在任何目录中指定模块，但在运行时该模块必须位于应用程序目录中。 如果在运行时该模块不位于应用程序目录中，你将收到 TypeLoadException。 `file` 不能包含程序集。 例如，如果输出文件是使用 module 的 TargetType 创建的，那么它的元数据可通过 AddModules 导入。

如果输出文件是使用 TargetType 选项而不是 module 创建的，则它的元数据不能通过 AddModules 导入，但可以通过 References 选项导入。

EmbedInteropTypes

使编译器让指定程序集中的 COM 类型信息可供当前正在编译的项目使用。

XML

```
<References>
  <EmbedInteropTypes>file1;file2;file3</EmbedInteropTypes>
</References>
```

其中 `file1;file2;file3` 是以分号分隔的程序集文件名的列表。 如果文件名包含空格，则将名称括在引号内。 使用 EmbedInteropTypes 选项可以部署具有嵌入类型信息的应用程序。 应用程序随后可以使用运行时程序集中实现嵌入类型信息的类型，而无需引用运行时程序集。 如果发布了各种版本的运行时程序集，则包含嵌入类型信息的应用程序可

以使用各种版本，而无需重新编译。 有关示例，请参阅演练：嵌入托管程序集中的类型。

当你使用 COM 互操作时，EmbedInteropTypes 选项尤其有用。 可以嵌入 COM 类型，以便应用程序在目标计算机上不再需要主互操作程序集 (PIA)。 EmbedInteropTypes 选项指示编译器将引用的互操作程序集中的 COM 类型信息嵌入到生成的已编译代码中。 COM 类型由 CLSID (GUID) 值进行标识。 因此，应用程序可以在安装了具有相同 CLSID 值的相同 COM 类型的目标计算机上运行。 自动执行 Microsoft Office 的应用程序是一个很好的示例。 由于 Office 等应用程序通常在不同版本间保持相同的 CLSID 值，因此只要在目标计算机上安装了 .NET Framework 4 或更高版本，并且应用程序使用引用的 COM 类型中包含的方法、属性或事件，应用程序便可以使用引用的 COM 类型。 EmbedInteropTypes 选项只嵌入接口、结构和委托。 不支持嵌入 COM 类。

> **备注**
>
> 在代码中创建嵌入 COM 类型的实例时，必须使用适当的接口创建该实例。 尝试使用组件类创建嵌入 COM 类型的实例会导致错误。

与 References 编译器选项一样，EmbedInteropTypes 编译器选项使用 Csc.rsp 响应文件，该文件引用常用的 .NET 程序集。 如果你不希望编译器使用 Csc.rsp 文件，可以使用 NoConfig 编译器选项。

C#

```
// The following code causes an error if ISampleInterface is an embedded
interop type.
ISampleInterface<SampleType> sample;
```

对于具有类型是从互操作程序集嵌入的泛型参数的类型，如果该类型来自外部程序集，则无法使用这种类型。 此限制不适用于接口。 例如，考虑在 Microsoft.Office.Interop.Excel 程序集中定义的 Range 接口。 如果某个库从 Microsoft.Office.Interop.Excel 程序集嵌入互操作类型，并且公开的一个方法返回具有类型是 Range 接口的参数的泛型类型，则该方法必须返回泛型接口，如下面的代码示例所示。

C#

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using Microsoft.Office.Interop.Excel;

public class Utility
{
```

```
    // The following code causes an error when called by a client assembly.
    public List<Range> GetRange1()
    {
        return null;
    }

    // The following code is valid for calls from a client assembly.
    public IList<Range> GetRange2()
    {
        return null;
    }
}
```

在下面的示例中，客户端代码可以调用返回 IList 泛型接口的方法而不会出现错误。

C#

```
public class Client
{
    public void Main()
    {
        Utility util = new Utility();

        // The following code causes an error.
        List<Range> rangeList1 = util.GetRange1();

        // The following code is valid.
        List<Range> rangeList2 = (List<Range>)util.GetRange2();
    }
}
```

用于报告错误和警告的 C# 编译器选项

项目 • 2025/01/03

以下选项控制编译器如何报告错误和警告。

展开表

MSBuild 语法	*csc.exe* 语法	描述
`WarningLevel`	`-warn`	设置警告级别。 更多信息。
`AnalysisLevel`	/	设置可选警告级别。 更多信息。
`TreatWarningsAsErrors`	`-warnaserror`	将所有警告视为错误。 更多信息。
`WarningsAsErrors`	`-warnaserror`	将一个或多个警告视为错误。 更多信息。
`WarningsNotAsErrors`	`-warnnotaserror`	不要将一个或多个警告视为错误。 更多信息。
`NoWarn`	`-nowarn`	设置禁用警告的列表。 更多信息。
`CodeAnalysisRuleSet`	`-ruleset`	指定禁用特定诊断的规则集文件。 更多信息。
`ErrorLog`	`-errorlog`	指定要记录所有编译器和分析器诊断的文件。 更多信息。
`ReportAnalyzer`	`-reportanalyzer`	报告其他分析器信息，如执行时间。 更多信息。

> **备注**
>
> 有关为项目配置这些选项的详细信息，请参阅**编译器选项**。

WarningLevel

WarningLevel 选项指定编译器显示的警告等级。

```
XML

<WarningLevel>3</WarningLevel>
```

此元素值是要为编译显示的警告等级：较低的数字仅显示高严重性警告。 较高的数字显示更多警告。 该值必须是零或正整数：

展开表

警告级别	含义
0	关闭发出所有警告消息。
1	显示严重警告消息。
2	显示等级 1 警告以及某些不太严重的警告，如有关隐藏类成员的警告。
3	显示等级 2 警告以及某些不太严重的警告，如有关经常计算为 `true` 或 `false` 的表达式的警告。
4（默认值）	显示所有等级 3 警告以及信息性警告。

> ⚠ **警告**
>
> 编译器命令行接受大于 4 的值，以启用**警告波警告**。 不过，.NET SDK 设置 *WarningLevel* 以匹配项目文件中的 *AnalysisLevel*。

若要获取有关错误或警告的信息，可以在帮助索引中查找错误代码。 有关获取错误或警告信息的其他方法，请参阅 C# 编译器错误。 使用 TreatWarningsAsErrors 将所有警告视为错误。 使用 DisabledWarnings 禁用某些警告。

分析级别

AnalysisLevel 选项指定要启用的其他警告波和分析器。 警告波警告是改进代码或确保其与即将发布的版本兼容的附加检查。 分析器提供类似 lint 的功能来改进代码。

XML

```
<AnalysisLevel>preview</AnalysisLevel>
```

展开表

分析级别	含义
5	显示所有可选的警告波 5 警告。
6	显示所有可选的警告波 6 警告。
7	显示所有可选的警告波 7 警告。
最新（默认值）	显示所有信息性警告，包括当前版本在内。

分析级别	含义
预览	显示所有信息性警告，包括最新的预览版本在内。
无	关闭所有信息性警告。

有关可选警告的详细信息，请参阅警告波。

若要获取有关错误或警告的信息，可以在帮助索引中查找错误代码。 有关获取错误或警告信息的其他方法，请参阅 C# 编译器错误。 使用 TreatWarningsAsErrors 将所有警告视为错误。 使用 NoWarn 禁用某些警告。

TreatWarningsAsErrors

TreatWarningsAsErrors 选项将所有警告视为错误。 你还可以使用 WarningsAsErrors 仅将部分警告设置为错误。 如果启用 **TreatWarningsAsErrors**，则可以使用 **WarningsNotAsErrors** 列出不应被视为错误的警告。

XML

```
<TreatWarningsAsErrors>true</TreatWarningsAsErrors>
```

而是将所有警告消息报告为错误。 生成过程会停止（不生成输出文件）。 默认情况下，**TreatWarningsAsErrors** 不生效，这意味着警告不会阻止生成输出文件。 （可选）如果希望仅将一些特定警告视为错误，则可以指定视为错误的警告编号的逗号分隔列表。 可以使用 Nullable 的简写形式指定所有为 Null 性警告的集合。 使用 WarningLevel 指定你希望编译器显示的警告等级。 使用 NoWarn 禁用某些警告。

> **ⓘ 重要**
>
> 使用 `<TreatWarningsAsErrors>` 文件中的 元素与使用 `warnaserror` MSBuild 命令行开关之间存在两个细微的差异。 *TreatWarningsAsErrors* 仅影响 C# 编译器，不会影响 *csproj* 文件中的任何其他 MSBuild 任务。 `warnaserror` 命令行开关会影响所有任务。 其次，在使用 *TreatWarningsAsErrors* 时，编译器不会对任何警告生成任何输出。 使用 `warnaserror` 命令行开关时，编译器将生成输出。

WarningsAsErrors 和 WarningsNotAsErrors

WarningsAsErrors 和 **WarningsNotAsErrors** 选项会覆盖警告列表的 **TreatWarningsAsErrors** 选项。 此选项可用于所有 *CS* 警告。 “CS”前缀是可选的。 可以使用数字或“CS”，后跟错误或警告编号。 有关影响警告的其他元素，请参阅 Common

MSBuild 属性。 除了警告 ID 列表之外，还可以指定字符串 `nullable`，它会将与可为 null 性相关的所有警告视为错误。

启用警告 0219、0168 和所有可为空警告作为错误：

```
XML

<WarningsAsErrors>0219,CS0168,nullable</WarningsAsErrors>
```

禁止将相同的警告视为错误：

```
XML

<WarningsNotAsErrors>0219,CS0168,nullable</WarningsNotAsErrors>
```

可以使用 **WarningsAsErrors** 将一组警告配置为错误。 将所有警告都设置为错误时，使用 **WarningsNotAsErrors** 配置一组不应为错误的警告。

NoWarn

NoWarn 选项允许禁止编译器显示一个或多个警告，其中 `warningnumber1` 和 `warningnumber2` 是希望编译器取消显示的警告号。 使用逗号分隔多个警告编号。 可以指定 `nullable` 以禁用与为 Null 性相关的所有警告。

```
XML

<NoWarn>warningnumber1,warningnumber2</NoWarn>
```

仅需指定警告标识符的数值部分。 例如，如果要禁止显示 *CS0028*，可以指定 `<NoWarn>28</NoWarn>`。 编译器会以无提示方式忽略传递给 **NoWarn** 的警告编号，这些编号在之前的版本中有效，但已被移除。 例如，*CS0679* 在 Visual Studio .NET 2002 的编译器中有效，但后来已被移除。

无法通过 NoWarn 选项禁止显示以下警告：

- 编译器警告（等级 1）CS2002
- 编译器警告（等级 1）CS2023
- 编译器警告（等级 1）CS2029

请注意，警告旨在表明代码存在潜在问题，因此应了解禁用任何特定警告的风险。 仅当确定警告为误报且不可能是运行时 bug 时，才使用 NoWarn。

可以使用更有针对性的方法来禁用警告：

- 大多数编译器提供了仅针对特定代码行禁用警告的方法，因此，如果警告发生在同一项目中的其他地方，仍可查看这些警告。 如果要仅在 C# 代码的特定部分中禁止显示警告，请使用 #pragma warning。
- 如果目标是在生成日志中看到更简洁且更集中的输出，则可能需要更改生成日志的详细程度。 有关详细信息，请参阅如何：查看、保存和配置生成日志文件。

若要在不覆盖先前设置的 NoWarn 值的情况下为其添加警告编号，请如下所示引用 `$(NoWarn)`：

XML

```
<NoWarn>$(NoWarn);newwarningnumber3;newwarningnumber4</NoWarn>
```

CodeAnalysisRuleSet

指定可配置特定诊断的规则集文件。

XML

```
<CodeAnalysisRuleSet>MyConfiguration.ruleset</CodeAnalysisRuleSet>
```

其中 `MyConfiguration.ruleset` 是规则集文件的路径。 有关使用规则集的详细信息，请参阅有关规则集的 Visual Studio 文档中的文章。

ErrorLog

指定要记录所有编译器和分析器诊断的文件。

XML

```
<ErrorLog>compiler-diagnostics.sarif</ErrorLog>
```

ErrorLog 选项会导致编译器输出静态分析结果交换格式 (SARIF) 日志。 SARIF 日志通常由分析编译器和分析器诊断结果的工具来读取。

可以使用 `version` 元素的 `ErrorLog` 参数来指定 SARIF 格式：

XML

```
<ErrorLog>logVersion21.json,version=2.1</ErrorLog>
```

分隔符可以是逗号 (`,`)，也可以是分号 (`;`)。 版本的有效值为：“1”、“2”和“2.1”。 默认值为“1”。 “2”和“2.1”均表示 SARIF 版本 2.1.0。

ReportAnalyzer

报告其他分析器信息，如执行时间。

```XML
<ReportAnalyzer>true</ReportAnalyzer>
```

ReportAnalyzer 选项会导致编译器发出额外的 MSBuild 日志信息，这些信息详细说明生成中分析器的性能特征。 它通常由分析器作者在验证分析器时使用。

> **ⓘ 重要**
>
> 只有在使用 `-verbosity:detailed` 命令行选项时，才会生成由此标志生成的额外日志信息。 有关详细信息，请参阅 MSBuild 文档中的**开关**一文。

控制代码生成的 C# 编译器选项

项目 • 2024/08/30

下面的选项控制编译器生成的代码。 新的 MSBuild 语法以**粗体**显示。 旧的 *csc.exe* 语法以 `code style` 显示。

- **DebugType** / `-debug`：发出（或不发出）调试信息。
- **Optimize** / `-optimize`：启用优化。
- **Deterministic** / `-deterministic`：从相同的输入源生成每字节对等的输出。
- **ProduceOnlyReferenceAssembly** / `-refonly`：生成引用程序集作为主输出，而非生成完整程序集。

> **备注**
>
> 有关为项目配置这些选项的详细信息，请参阅**编译器选项**。

DebugType

DebugType 选项将使编译器生成调试信息，并将此信息放置在一个或多个输出文件中。 默认情况下会添加调试信息。

XML

```
<DebugType>pdbonly</DebugType>
```

自 C# 6.0 起，对于所有编译器版本而言，*pdbonly* 与 *full* 之间没有任何区别。 请选择 *pdbonly*。 若要更改 *.pdb* 文件的位置，请参阅 **PdbFile**。

以下为有效值：

展开表

Value	含义
`full`	使用当前平台的默认格式向 *.pdb* 文件发出调试信息： **Windows**：Windows pdb 文件。 **Linux/macOS**：可移植 PDB 文件。
`pdbonly`	与 `full` 相同。 有关详细信息，请参阅下面的注释。
`portable`	使用跨平台可移植 PDB 格式向 .pdb 文件发出调试信息。

Value	含义
`embedded`	使用可移植 PDB 格式向 *.dll/.exe* 自身（未生成 *.pdb* 文件）发出调试信息。

> **重要**
>
> 以下信息仅适用于 C# 6.0 以前的编译器。 此元素的值可以是 `full` 或 `pdbonly`。 *full* 参数（在不指定 *pdbonly* 时生效）允许将调试器附加到正在运行的程序。 指定 *pdbonly* 后，可以在调试器中启动程序时进行源代码调试，但仅在正在运行的程序附加到调试器时才显示汇编程序。 使用此选项创建调试版本。 如果使用 *Full*，请注意，对经过优化的 JIT 代码的速度和大小会存在一定影响，使用 *full* 时对代码质量的影响较小 。 建议使用 *pdbonly* 或不使用 PDB 生成发布代码。 *pdbonly* 和 *full* 之间的一个区别在于，使用 *full*，编译器将发出 DebuggableAttribute，用于告知 JIT 编译器有可用调试信息。 因此，在使用 *full* 时，如果代码包含设置为 false 的 DebuggableAttribute，将出现错误。 有关如何配置应用程序的调试性能的详细信息，请参阅**令映像更易于调试**。

优化

Optimize 选项启用或禁用编译器执行的优化，使输出文件更小、更快、更有效。 默认情况下，为*发布*生成配置启用 *Optimize* 选项。 默认情况下，为*调试*和任何其他生成配置禁用此选项。

XML

```
<Optimize>true</Optimize>
```

在 Visual Studio 中，可以从项目的**生成**属性页中设置 **Optimize** 选项。

Optimize 还指示公共语言运行时在运行时优化代码。 默认情况下，禁用优化。 指定 **Optimize+** 可启用优化。 生成程序集使用的模块时，请使用与程序集所使用的相同 **Optimize** 设置。 可以将 **Optimize** 和 **Debug** 选项组合使用。

具有确定性

如果输入相同，则会导致编译器生成的程序集其逐字节输出在整个编译期间中相同。

XML

```
<Deterministic>true</Deterministic>
```

默认情况下，来自一组给定输入的编译器输出是唯一的，因为编译器添加从随机数生成的时间戳和 MVID（Module.ModuleVersionId。基本上，它是一个 GUID，用于唯一标识模块和版本）。 使用 `<Deterministic>` 选项生成*确定性的程序集*，只要输入保持不变，该程序集的二进制内容在整个编译中都是相同的。 在此类生成中，时间戳和 MVID 字段会被替换为从所有编译输入的哈希派生的值。 编译器会考虑影响确定性的以下输入：

- 命令行参数序列。
- 编译器 .rsp 响应文件的内容。
- 所用编译器的精确版本及其引用的程序集。
- 当前目录路径。
- 直接或间接地显式传递到编译器的所有文件的二进制内容，包括：
  - 源文件
  - 引用的程序集
  - 引用的模块
  - 资源
  - 强名称密钥文件
  - @ 响应文件
  - 分析器
  - 规则集
  - 分析器可能使用的其他文件
- 当前区域性（针对生成诊断和异常消息的语言）。
- 在未指定编码情况下使用的默认编码（或当前代码页）。
- 编译器搜索路径（例如，由 `-lib` 或 `-recurse` 指定）上文件是否存在及其内容。
- 运行编译器的公共语言运行时 (CLR) 平台。
- `%LIBPATH%` 的值，该值会影响分析器的依赖项加载。

可使用确定性编译来确定是否从可信源编译二进制内容。 当源公开可用时，确定性输出会很有用。 它还可以确定生成步骤是否依赖于生成过程中使用的二进制文件的更改。

ProduceOnlyReferenceAssembly

ProduceOnlyReferenceAssembly 选项表示应输出引用程序集（而不是实现程序集）作为主输出。 **ProduceOnlyReferenceAssembly** 参数以无提示方式禁用输出 PDB，因为无法执行引用程序集。

XML

```xml
<ProduceOnlyReferenceAssembly>true</ProduceOnlyReferenceAssembly>
```

引用程序集是一种特殊类型的程序集。 引用程序集只包含表示库的公共 API 外围应用所需的最少元数据量。 它们包括在生成工具中引用程序集时所需的所有成员的声明，但不

包括所有成员实现以及对其 API 协定没有明显影响的私有成员的声明。 有关详细信息，请参阅引用程序集。

ProduceOnlyReferenceAssembly 和 **ProduceReferenceAssembly** 选项是互斥的。

用于安全性的 C# 编译器选项

项目 • 2023/05/10

以下选项可控制编译器安全选项。 新的 MSBuild 语法以粗体显示。 旧的 csc.exe 语法以 `code style` 显示。

- **PublicSign** / `-publicsign`：公开对程序集签名。
- **DelaySign** / `-delaysign`：仅使用强名称密钥的公共部分对程序集进行延迟签名。
- **KeyFile** / `-keyfile`：指定强名称密钥文件。
- **KeyContainer** / `-keycontainer`：指定强名称密钥容器。
- **HighEntropyVA** / `-highentropyva`：启用高熵地址空间布局随机化 (ASLR)

PublicSign

此选项会导致编译器应用公钥，但不会实际对程序集签名。 PublicSign 选项还会在程序集中设置位，以告知运行时该文件已签名。

XML

```
<PublicSign>true</PublicSign>
```

PublicSign 选项需要使用 KeyFile 或 KeyContainer 选项。 KeyFile 和 KeyContainer 选项指定公钥。 PublicSign 和 DelaySign 选项互斥 。 公共签名有时称为“假签名”或“OSS 签名”，它包括输出程序集中的公钥并设置“已签名”标记。 公共签名实际上并不使用私钥对程序集进行签名。 开发人员为开放源代码项目使用公共签名。 当人们无权访问用于对程序集进行签名的私钥时，他们会生成与已发布的“完全签名”程序集兼容的程序集。 由于很少有使用者实际需要检查程序集是否完全签名，因此这些公开生成的程序集几乎适用于每个使用完全签名程序集的方案。

DelaySign

此选项会使编译器在输出文件中保留空间，以便以后添加数字签名。

XML

```
<DelaySign>true</DelaySign>
```

如果需要完全签名的程序集，请使用 DelaySign-。 如果仅需要将公钥置于程序集中，则使用 DelaySign。 除非与 KeyFile 或 KeyContainer 一同使用，否则 DelaySign 选项将不起作用。 KeyContainer 和 PublicSign 选项互斥。 在请求完全签名的程序集时，编译器会对

包含清单（程序集元数据）的文件进行哈希处理，并使用私钥对哈希进行签名。 该操作创建一个数字签名，它存储在包含清单的文件中。 在对程序集延迟签名时，编译器不会计算和存储签名。 相反，编译器会在文件中保留空间，便于稍后添加签名。

使用 DelaySign，测试人员可以将程序集放入全局缓存中。 测试完成后，可使用程序集链接器实用工具将私钥置于程序集中，对程序集进行完全签名。 有关详细信息，请参阅创建和使用具有强名称的程序集和延迟为程序集签名。

KeyFile

指定包含加密密钥的文件名。

XML

```
<KeyFile>filename</KeyFile>
```

`file` 是包含强名称密钥的文件的名称。 使用此选项时，编译器在程序集清单中插入指定字段的公钥，然后使用私钥对最终的程序集进行签名。 若要生成密钥文件，请在命令行键入 `sn -k file`。 如果你使用 -target:module 进行编译，密钥文件的名称将保存在模块中，并在你使用 AddModules 编译程序集时，合并到创建的程序集中。 你也可以使用 KeyContainer 将加密信息传递给编译器。 如果需要部分签名的程序集，请使用 DelaySign。 如果在同一编译中同时指定 KeyFile 和 KeyContainer，则编译器将首先尝试使用密钥容器。 如果成功，则使用密钥容器中的信息对程序集签名。 如果编译器没有找到密钥容器，它将尝试使用通过 KeyFile 指定的文件。 如果成功，则使用密钥文件中的信息对程序集签名，并且将密钥信息安装到密钥容器中。 在下一次编译中，密钥容器将生效。 密钥文件可能仅包含公钥。 有关详细信息，请参阅创建和使用具有强名称的程序集和延迟为程序集签名。

KeyContainer

指定加密密钥容器的名称。

XML

```
<KeyContainer>container</KeyContainer>
```

`container` 是强名称密钥容器的名称。 当使用 KeyContainer 选项时，编译器将创建一个可共享的组件。 编译器在程序集清单中插入指定容器的公钥，然后使用私钥对最终的程序集进行签名。 若要生成密钥文件，请在命令行键入 `sn -k file`。 `sn -i` 将密钥对安装到容器中。 编译器在 CoreCLR 上运行时，不支持此选项。 若要在基于 CoreCLR 生成时对程序集进行签名，请使用 KeyFile 选项。 如果你使用 TargetType 进行编译，那么，当

你使用 AddModules 将此模块编译到程序集时，密钥文件的名称将保存在模块中，而且会并入程序集。 还可以将此选项指定为任何 Microsoft 中间语言 (MSIL) 模块的源代码中的自定义特性 (System.Reflection.AssemblyKeyNameAttribute)。 此外，可使用 KeyFile 将加密信息传递给编译器。 使用 DelaySign 可将公钥添加到程序集清单中，但在程序集被测试之前，会对其进行签名。 有关详细信息，请参阅创建和使用具有强名称的程序集和延迟为程序集签名。

HighEntropyVA

HighEntropyVA 编译器选项可告知 Windows 内核，特定的可执行文件是否支持高熵地址空间布局随机化 (ASLR)。

XML

```
<HighEntropyVA>true</HighEntropyVA>
```

此选项指定 64 位可执行文件或由 PlatformTarget 编译器选项标记的可执行文件支持高熵虚拟地址空间。 默认情况下，所有 .NET Standard 和 .NET Core 版本以及从 .NET Framework 4.5 开始的 .NET Framework 版本都启用该选项。

当随机化进程的地址空间布局包含在 ASLR 中时，HighEntropyVA 选项允许 Windows 内核的兼容版本使用更高程度的熵。 使用更高程度的熵意味着，可向内存区域（例如堆栈或堆）分配更多的地址。 因此，猜测特定内存区域的位置会更加困难。 HighEntropyVA 编译器选项需要目标可执行文件及其依赖的任何模块在作为 64 位进程运行时，能够处理大于 4 GB 的指针值。

指定资源的 C# 编译器选项

项目 • 2023/05/09

以下选项控制 C# 编译器如何创建或导入 Win32 资源。 新的 MSBuild 语法以粗体显示。 旧的 csc.exe 语法以 `code style` 显示。

- **Win32Resource** / `-win32res`：指定 Win32 资源文件 (.res)。
- **Win32Icon** / `-win32icon`：引用来自一个或多个指定程序集文件的元数据。
- **Win32Manifest** / `-win32manifest`：指定 Win32 清单文件 (.xml)。
- **NoWin32Manifest** / `-nowin32manifest`：不包括默认的 Win32 清单。
- **Resources** / `-resource`：嵌入指定的资源（简短形式：/res）。
- **LinkResources** / `-linkresources`：将指定的资源链接到此程序集。

Win32Resource

Win32Resource 选项会在输出文件中插入 Win32 资源。

XML

```
<Win32Resource>filename</Win32Resource>
```

`filename` 是要添加到输出文件的资源文件。 Win32 资源可以包含版本或位图（图标）信息，这些信息有助于在文件资源管理器中标识您的应用程序。 如果不指定此选项，编译器将根据程序集版本生成版本信息。

Win32Icon

Win32Icon 选项在输出文件中插入 .ico 文件，为输出文件提供在文件资源管理器中所需的外观。

XML

```
<Win32Icon>filename</Win32Icon>
```

`filename` 是要添加到输出文件的 .ico 文件。 可以使用资源编译器创建 .ico 文件。 在编译 Visual C++ 程序时会调用资源编译器；.ico 文件是从 .rc 文件创建的。

Win32Manifest

使用 Win32Manifest 选项可以指定要嵌入到项目的可移植可执行 (PE) 文件中的用户定义的 Win32 应用程序清单文件。

```
XML

<Win32Manifest>filename</Win32Manifest>
```

`filename` 是自定义清单文件的名称和位置。 默认情况下，C# 编译器嵌入可指定“asInvoker”的请求执行级别的应用程序清单。 它在生成该可执行文件的同一文件夹中创建清单。 如果要提供自定义清单（例如，指定“highestAvailable”或“requireAdministrator”的请求执行级别的清单），请使用此选项指定文件名。

> **备注**
>
> 此选项和 Win32Resources 选项是互斥的。 如果尝试在同一命令行中同时使用这两个选项，将收到一个生成错误。

如果应用程序没有用于指定请求执行级别的应用程序清单，将受到 Windows 中“用户帐户控制”功能下的文件和注册表虚拟化的影响。 有关详细信息，请参阅用户帐户控制。

如果满足下列任一条件，则应用程序会受到虚拟化的影响：

- 使用 NoWin32Manifest 选项，并且在随后的生成步骤中未提供清单，或者没有通过使用 Win32Resource 选项将其包含在 Windows 资源 (.res) 文件中。
- 提供的自定义清单未指定请求执行级别。

Visual Studio 创建默认 .manifest 文件，并将它与可执行文件一起存储在“调试”和“发布”目录中。 可以用任意文本编辑器创建一个清单，然后将该文件添加到项目中，从而添加自定义清单。 或者，也可以右键单击“解决方案资源管理器”中的“项目”图标，选择“添加新项”，然后选择“应用程序清单文件”。 添加完新的或现有清单文件后，该文件将显示在“清单”下拉列表中。 有关详细信息，请参阅“项目设计器”->“应用程序”页 (C#)。

提供应用程序清单的操作可以作为自定义生成后步骤，也可以通过使用 NoWin32Manifest 选项作为 Win32 资源文件的组成部分。 如果希望应用程序受到 Windows Vista 的文件或注册表虚拟化的影响，请使用该选项。

NoWin32Manifest

使用 NoWin32Manifest 选项可指示编译器不将任何应用程序清单嵌入到可执行文件中。

```
XML
```

```
<NoWin32Manifest />
```

使用此选项时，除非在 Win32 资源文件或以后的生成步骤中提供应用程序清单，否则应用程序会受到 Windows Vista 上虚拟化的影响。

在 Visual Studio 的“应用程序属性”页中，通过在“清单”下拉列表中选择“创建不带清单的应用程序”选项来设置此选项。 有关详细信息，请参阅“项目设计器”->“应用程序”页 (C#)。

资源

将指定资源嵌入输出文件。

XML

```
<Resources Include=filename>
  <LogicalName>identifier</LogicalName>
  <Access>accessibility-modifier</Access>
</Resources>
```

`filename` 是要嵌入到输出文件的 .NET 资源文件。 `identifier`（可选）是资源的逻辑名称；用于加载资源的名称。 默认值是文件的名称。 `accessibility-modifier`（可选）是资源的可访问性：public 或 private。 默认值为 public。 默认情况下，如果使用 C# 编译器创建资源，则这些资源在程序集中是公有的。 若要使资源变为私有，请将 `private` 指定为可访问性修饰符。 不允许使用 `public` 或 `private` 以外的任何其他可访问性。 例如，如果 `filename` 是由 Resgen.exe 创建的或在开发环境中创建的 .NET 资源文件，则可使用 System.Resources 命名空间中的成员来访问它。 有关详细信息，请参阅 System.Resources.ResourceManager。 对于所有其他资源，请使用 Assembly 类中的 `GetManifestResource` 方法在运行时访问资源。 输出文件中资源的顺序由项目文件中所指定的顺序决定。

LinkResources

在输出文件中创建指向 .NET 资源的链接。 不会在输出文件中添加资源文件。 LinkResources 不同于会在输出文件中嵌入资源文件的 Resource 选项。

XML

```
<LinkResources Include=filename>
  <LogicalName>identifier</LogicalName>
  <Access>accessibility-modifier</Access>
</LinkResources>
```

`filename` 是要从程序集链接到的 .NET 资源文件。 `identifier`（可选）是资源的逻辑名称；用于加载资源的名称。 默认值是文件的名称。 `accessibility-modifier`（可选）是资源的可访问性：public 或 private。 默认值为 public。 默认情况下，如果使用 C# 编译器创建链接资源，则这些资源在程序集中是公有的。 若要使资源变为私有，请将 `private` 指定为可访问性修饰符。 不允许使用 `public` 或 `private` 以外的任何其他修饰符。 例如，如果 `filename` 是由 Resgen.exe 创建的或在开发环境中创建的 .NET 资源文件，则可使用 System.Resources 命名空间中的成员来访问它。 有关详细信息，请参阅 System.Resources.ResourceManager。 对于所有其他资源，请使用 Assembly 类中的 `GetManifestResource` 方法在运行时访问资源。 `filename` 中指定的文件可为任何格式。 例如，你可能希望生成程序集的本机 DLL 部分，从而可将它安装到全局程序集缓存中，并且可从该程序集中的托管代码访问它。 可在程序集链接器中执行相同的操作。 有关详细信息，请参阅 Al.exe（程序集链接器）和使用程序集和全局程序集缓存。

其他 C# 编译器选项

项目 • 2023/05/10

下面的选项控制其他编译器行为。 新的 MSBuild 语法以粗体显示。 旧 csc.exe 命令行语法以 `code style` 显示。

- ResponseFiles / *`@CustomOpts.RSP`*：读取指定的响应文件以获取更多选项。
- **NoLogo** / `-nologo`：禁止显示编译器版权消息。
- NoConfig / `-noconfig`：不自动包含 CSC.RSP 文件。

ResponseFiles

通过 ResponseFiles 选项，可以指定包含编译器选项和要编译的源代码文件的文件。

XML

```
<ResponseFiles>response_file</ResponseFiles>
```

`response_file` 指定一个文件来列出编译器选项或要编译的源代码文件。 编译器选项和源代码文件将由编译器处理，就像它们在命令行中被指定一样。 若要在一次编译中指定多个响应文件，请指定多个响应文件选项。 在响应文件中，多个编译器选项和源代码文件可以出现在同一行中。 单个编译器选项的指定必须出现在同一行中（不能跨行）。 响应文件的注释可以 # 符号开始。 从响应文件内指定编译器选项就如同在命令行发出这些命令。 编译器在读取命令选项时会进行处理。 命令行参数可以重写先前在响应文件中列出的选项。 反之，响应文件中的选项也将重写先前在命令行或其他响应文件中列出的选项。 C# 提供 csc.rsp 文件，该文件与 csc.exe 文件位于同一目录。 有关响应文件格式的详细信息，请参阅 NoConfig。 不能在 Visual Studio 开发环境中设置此编译器选项，也不能以编程方式对其进行更改。 以下几行来自示例响应文件：

控制台

```
# build the first output file
-target:exe -out:MyExe.exe source1.cs source2.cs
```

NoLogo

NoLogo 选项可在编译器启动时禁止显示登录版权标志，并在编译期间禁止显示信息性消息。

XML

```
<NoLogo>true</NoLogo>
```

NoConfig

NoConfig 选项会告知编译器不要使用 csc.rsp 文件进行编译。

XML

```
<NoConfig>true</NoConfig>
```

csc.rsp 文件引用 .NET Framework 随附的所有程序集。 Visual Studio .NET 开发环境包括的实际引用取决于项目类型。 可以修改 csc.rsp 文件，并指定每次编译时应包括的其他编译器选项。 如果你不需要编译器查询和使用 csc.rsp 文件中的设置，请指定 NoConfig。 此编译器选项在 Visual Studio 中不可用，并且无法以编程方式更改。

高级 C# 编译器选项

项目 • 2023/07/24

以下选项支持高级方案。 新的 MSBuild 语法以粗体显示。 旧的 `csc.exe` 语法以 `code style` 显示。

- **MainEntryPoint**、**StartupObject** / `-main`：指定包含入口点的类型。
- **PdbFile** / `-pdb`：指定调试信息文件名。
- **PathMap** / `-pathmap`：指定编译器输出的源路径名的映射。
- **ApplicationConfiguration** / `-appconfig`：指定包含程序集绑定设置的应用程序配置文件。
- **AdditionalLibPaths** / `-lib`：指定要在其中搜索引用的其他目录。
- **GenerateFullPaths** / `-fullpath`：编译器生成完全限定的路径。
- **PreferredUILang** / `-preferreduilang`：指定首选输出语言名称。
- **BaseAddress** / `-baseaddress`：指定要生成的库的基址。
- **ChecksumAlgorithm** / `-checksumalgorithm`：指定用于计算 PDB 中存储的源文件校验和的算法。
- **CodePage** / `-codepage`：指定在打开源文件时使用的代码页。
- **Utf8Output** / `-utf8output`：以 UTF-8 编码输出编译器消息。
- **FileAlignment** / `-filealign`：指定用于输出文件节的对齐方式。
- **ErrorEndLocation** / `-errorendlocation`：输出每个错误结尾位置的行和列。
- **NoStandardLib** / `-nostdlib`：不引用标准库 mscorlib.dll。
- **SubsystemVersion** / `-subsystemversion`：指定此程序集的子系统版本。
- **ModuleAssemblyName** / `-moduleassemblyname`：此模块所属程序集的名称。
- **ReportIVT** / `-reportivts`：生成有关 System.Runtime.CompilerServices.InternalsVisibleToAttribute 信息的其他信息。

在 `*.csproj` 文件中的 `<PropertyGroup>` 元素中添加以下任一选项：

XML

```
<PropertyGroup>
    <StartupObject>...</StartupObject>
    ...
</PropertyGroup>
```

MainEntryPoint 或 StartupObject

如果多个类包含 `Main` 方法，此选项将指定包含程序入口点的类。

XML

```
<StartupObject>MyNamespace.Program</StartupObject>
```

或

XML

```
<MainEntryPoint>MyNamespace.Program</MainEntryPoint>
```

其中 `Program` 是包含 `Main` 方法的类型。 提供的类名必须是完全限定类名；它必须包括完整命名空间（包含类），后跟类名。 例如，当 `Main` 方法位于 `MyApplication.Core` 命名空间中的 `Program` 类中时，编译器选项必须为 `-main:MyApplication.Core.Program`。 如果编译包含具有 Main 方法的多个类型，则可以指定哪个类型包含 `Main` 方法。

> ⓘ **备注**
>
> 此选项不能用于包含**顶级语句**的项目，即使该项目包含一个或多个 `Main` 方法也是如此。

PdbFile

PdbFile 编译器选项指定调试符号文件的名称和位置。 `filename` 值指定调试符号文件的名称和位置。

XML

```
<PdbFile>filename</PdbFile>
```

指定 DebugType 后，编译器将在创建输出文件（.exe 或 .dll）的相同目录中创建 .pdb 文件。 .pdb 文件与输出文件的名称具有相同的基文件名。 PdbFile 允许为 .pdb 文件指定非默认的文件名和位置。 不能在 Visual Studio 开发环境中设置此编译器选项，也不能以编程方式对其进行更改。

PathMap

PathMap 编译器选项指定如何将物理路径映射到编译器输出的源路径名称。 此选项将编译器在其上运行的计算机上的每个物理路径映射到应写入输出文件的相应路径。 在下面

的示例中，`path1` 是当前环境中源文件的完整路径，`sourcePath1` 是任何输出文件中替代 `path1` 的源路径。 要指定多个映射的源路径，请用逗号分隔每个路径。

```
XML

<PathMap>path1=sourcePath1,path2=sourcePath2</PathMap>
```

编译器将源路径写入其输出，原因如下：

1. 将 CallerFilePathAttribute 应用于可选参数时，会将源路径替换为参数。
2. PDB 文件中嵌入的源路径。
3. PDB 文件的路径嵌入到 PE（可移植的可执行文件）文件中。

ApplicationConfiguration

ApplicationConfiguration 编译器选项使 C# 应用程序能够在程序集绑定时将程序集的应用程序配置 (app.config) 文件的位置指定为公共语言运行时 (CLR)。

```
XML

<ApplicationConfiguration>file</ApplicationConfiguration>
```

其中 `file` 是包含程序集绑定设置的应用程序配置文件。 ApplicationConfiguration 的一种用途是处理高级方案，在这种方案中，程序集必须同时引用特定引用程序集的 .NET Framework 版本和 .NET Framework for Silverlight 版本。 例如，在 Windows Presentation Foundation (WPF) 中编写的 XAML 设计器可能需要为设计器用户界面引用 WPF 桌面以及随附于 Silverlight 的 WPF 子集。 同一设计器程序集必须访问这两个程序集。 默认情况下，单独引用会导致编译器错误，因为程序集绑定将这两个程序集视为等效。 借助 ApplicationConfiguration 编译器选项，可通过使用 `<supportPortability>` 标记指定禁用默认行为的 app.config 文件的位置，如以下示例所示。

```
XML

<supportPortability PKT="7cec85d7bea7798e" enable="false"/>
```

编译器将文件的位置传递给 CLR 的程序集绑定逻辑。

> **备注**
>
> 若要使用已在项目中设置的 app.config 文件，请将属性标记 `<UseAppConfigForCompiler>` 添加到 .csproj 文件，并将其值设置为 `true`。 若要指定

不同的 app.config 文件，请添加属性标记 `<AppConfigForCompiler>` 并将其值设置为该文件的位置。

以下示例展示一个 app.config 文件，通过使用该文件，应用程序能够同时引用任何 .NET Framework 程序集（同时存在于后述两个实现中）的 .NET Framework 实现和 .NET Framework for Silverlight 实现。 ApplicationConfiguration 编译器选项指定此 app.config 文件的位置。

XML

```
<configuration>
  <runtime>
    <assemblyBinding>
      <supportPortability PKT="7cec85d7bea7798e" enable="false"/>
      <supportPortability PKT="31bf3856ad364e35" enable="false"/>
    </assemblyBinding>
  </runtime>
</configuration>
```

AdditionalLibPaths

AdditionalLibPaths 选项指定通过 References 选项引用的程序集的位置。

XML

```
<AdditionalLibPaths>dir1[,dir2]</AdditionalLibPaths>
```

其中 `dir1` 是在当前工作目录（从中调用编译器的目录）或公共语言运行时的系统目录中未找到引用程序集时，编译器将在其中进行查找的目录。 `dir2` 是要在其中搜索程序集引用的一个或多个其他目录。 用逗号分隔每个目录名称，且中间没有空格。 编译器按以下顺序搜索未完全限定的程序集引用：

1. 当前工作目录。
2. 公共语言运行时系统目录。
3. 由 AdditionalLibPaths 指定的目录。
4. 由 LIB 环境变量指定的目录。

使用 Reference 指定程序集引用。 AdditionalLibPaths 是累加的。 每一次指定的值都追加到以前的值中。 由于程序集清单中未指定依赖程序集的路径，因此应用程序将查找并使用全局程序集缓存中的程序集。 编译器可以引用程序集并不表示公共语言运行时可以在运行时找到并加载程序集。 有关运行时如何搜索引用的程序集的详细信息，请参阅运行时如何定位程序集。

GenerateFullPaths

GenerateFullPaths 选项导致编译器在列出编译错误和警告时指定文件的完整路径。

```Xml
<GenerateFullPaths>true</GenerateFullPaths>
```

默认情况下，由编译所产生的错误和警告指定其中发现错误的文件的名称。 GenerateFullPaths 选项导致编译器指定文件的完整路径。 此编译器选项在 Visual Studio 中不可用，并且无法以编程方式更改。

PreferredUILang

通过使用 PreferredUILang 编译器选项，可指定 C# 编译器用于显示输出（如错误消息）的语言。

```XML
<PreferredUILang>language</PreferredUILang>
```

其中 `language` 是用于编译器输出的语言的语言名称。 可以使用 PreferredUILang 编译器选项指定 C# 编译器用于错误消息和其他命令行输出的语言。 如果未安装针对该语言的语言包，将改用操作系统的语言设置。

BaseAddress

通过 BaseAddress 选项，可指定加载 DLL 的首选基址。 若要深入了解何时且为何要使用此选项，请参阅 Larry Osterman 的网络日志。

```XML
<BaseAddress>address</BaseAddress>
```

其中 `address` 是 DLL 的基址。 可将此地址指定为十进制数、十六进制数或八进制数。 DLL 的默认基址由 .NET 公共语言运行时设置。 此地址中的低序字将被舍入取整。 例如，如果指定 `0x11110001`，它将被舍入为 `0x11110000`。 要完成 DLL 的签名过程，请使用具有 -R 选项的 SN.EXE。

ChecksumAlgorithm

此选项控制用于对 PDB 中的源文件进行编码的校验和算法。

```
XML

<ChecksumAlgorithm>algorithm</ChecksumAlgorithm>
```

`algorithm` 必须是 `SHA1`（默认）或 `SHA256`。

CodePage

如果所需页不是系统的当前默认代码页，则此选项指定编译期间要使用的代码页。

```
XML

<CodePage>id</CodePage>
```

其中 `id` 是要用于编译中所有源代码文件的代码页 ID。 编译器将首先尝试将所有源文件解释为 UTF-8。 如果源代码文件使用除 UTF-8 以外的编码并使用除 7 位 ASCII 字符以外的字符，请使用 CodePage 选项指定应使用的代码页。 CodePage 适用于编译中的所有源代码文件。 有关如何查找系统上支持哪些代码页的信息，请参阅 GetCPInfo。

Utf8Output

Utf8Output 选项使用 UTF-8 编码显示编译器输出。

```
XML

<Utf8Output>true</Utf8Output>
```

在某些国际配置中，编译器输出无法在控制台上正确显示。 使用 Utf8Output 并将编译器输出重定向到文件。

FileAlignment

FileAlignment 选项用于指定输出文件中各节的大小。 有效值为 512、1024、2048、4096 和 8192。 这些值以字节为单位。

```
XML

<FileAlignment>number</FileAlignment>
```

在 Visual Studio 中，可以从项目的“生成”属性的“高级”页中设置 FileAlignment 选项。 每一节都将在是 FileAlignment 值的倍数的边界上对齐。 没有固定的默认值。 如果未指定 FileAlignment，公共语言运行时在编译时会选取一个默认值。 通过指定节的大小，可以影响输出文件的大小。 修改节的大小可能对将在较小设备上运行的程序有用。 使用 DUMPBIN 可查看有关输出文件中各节的信息。

ErrorEndLocation

指示编译器输出每个错误结尾位置的行和列。

XML

```
<ErrorEndLocation>true</ErrorEndLocation>
```

默认情况下，编译器会为所有错误和警告在源代码中写入起始位置。 当此选项设置为 true 时，编译器会为每个错误和警告写入起始和结尾位置。

NoStandardLib

NoStandardLib 可防止导入 mscorlib.dll，后者定义了整个 System 命名空间。

XML

```
<NoStandardLib>true</NoStandardLib>
```

如果你想要定义或创建自己的 System 命名空间和对象，请使用此选项。 如果不指定 NoStandardLib，则 mscorlib.dll 将被导入到程序中（与指定 `<NoStandardLib>false</NoStandardLib>` 相同）。

SubsystemVersion

指定可执行文件可以运行的子系统的最低版本。 大多数情况下，此选项确保该可执行文件可以使用早期 Windows 版本中未提供的安全功能。

> **备注**
>
> 若要指定子系统本身，请使用 TargetType 编译器选项。

XML

```
<SubsystemVersion>major.minor</SubsystemVersion>
```

`major.minor` 指定所需的子系统最低版本，以主版本和次要版本之间使用点标记的方式表示。 例如，你可以指定应用程序不能在 Windows 7 之前的操作系统上运行。 将此选项的值设置为 6.01，如本文后面的表中所述。 将 `major` 和 `minor` 的值指定为整数。 `minor` 版本中的前导零不会更改版本，但尾随零会。 例如，6.1 和 6.01 表示相同的版本，但 6.10 表示另一个版本。 建议次要版本用两位数表示，以免混淆。

下表列出了常见的 Windows 子系统版本。

Windows 版本	**子系统版本**
Windows Server 2003	5.02
Windows Vista	6.00
Windows 7	6.01
Windows Server 2008	6.01
Windows 8	6.02

SubsystemVersion 编译器选项的默认值取决于以下列表中的条件：

- 只要设置了以下列表中的任意编译器选项，则默认值为 6.02：
  - /target:appcontainerexe
  - /target:winmdobj
  - -platform:arm
- 如果使用 MSBuild，面向 .NET Framework 4.5，并且未设置先前在此列表中指定的任何编译器选项，则默认值为 6.00。
- 如果前面的条件均不符合，则默认值为 4.00。

ModuleAssemblyName

指定 .netmodule 可以访问其非公共类型的程序集的名称。

XML

```
<ModuleAssemblyName>assembly_name</ModuleAssemblyName>
```

生成 .netmodule 时，应使用 ModuleAssemblyName 并满足以下条件：

- .netmodule 需要具有访问现有程序集中非公共类型的权限。
- 知道生成后的 .netmodule 所在程序集的名称。

- 现有程序集已经获得友元程序集访问权限，可访问将在其中生成 .netmodule 的程序集。

有关生成 .netmodule 的详细信息，请参阅模块的 TargetType 选项。 有关友元程序集的详细信息，请参阅友元程序集。

ReportIVTs

启用或禁用有关在编译过程中找到的 System.Runtime.CompilerServices.InternalsVisibleToAttribute 的其他诊断信息：

```XML
<ReportIVTs>true</ReportIVTs>
```

如果元素内容为 `true`，则启用诊断；如果元素内容为 `false` 或不存在，则禁用诊断。

ReportIVTs 在启用时会报告以下信息：

1. 任何不可访问的成员诊断包括其源程序集（如果不同于当前程序集）。
2. 编译器会输出正在编译的项目的程序集标识、其程序集名称和公钥。
3. 对于传递给编译器的每个引用，它都会输出；
   a. 引用的程序集标识
   b. 引用是否授予当前项目 `InternalsVisibleTo`
   c. 从此程序集授予 `InternalsVisibleTo` 的任何程序集的名称和所有公钥

XML 文档注释

项目 • 2024/04/11

C# 源文件可以具有结构化注释，这些注释生成这些文件中定义的类型的 API 文档。 C# 编译器会生成一个 XML 文件，其中包含表示注释和 API 签名的结构化数据。 例如，其他工具可以处理该 XML 输出，以网页或 PDF 文件的形式创建人类可读的文档。

此过程为你在代码中添加 API 文档提供了许多好处：

- C# 编译器将 C# 代码的结构与注释文本结合到单个 XML 文档中。
- C# 编译器会验证注释是否与相关标记的 API 签名相匹配。
- 处理 XML 文档文件的工具可定义特定于这些工具的 XML 元素和属性。

Visual Studio 等工具为文档注释中使用的很多常用 XML 元素提供 IntelliSense。

本文将介绍这些主题：

- 文档注释和 XML 文件生成
- 由 C# 编译器和 Visual Studio 验证的标记
- 生成的 XML 文件的格式

创建 XML 文档输出

通过编写由三斜杠指示的特殊注释字段，可以为代码创建文档。 注释字段包含用于描述注释后面的代码块的 XML 元素。 例如：

C#

```
/// <summary>
///  This class performs an important function.
/// </summary>
public class MyClass {}
```

设置 GenerateDocumentationFile 或 DocumentationFile 选项后，编译器将在源代码中找到包含 XML 标记的所有注释字段，并根据这些注释创建 XML 文档文件。 启用此选项后，编译器将为项目中声明的所有公开可见成员生成 CS1591 警告，且没有 XML 文档注释。

XML 注释格式

XML 文档注释需要使用分隔符，这些分隔符指示文档注释开始和结束的位置。 可以将以下分隔符用于 XML 文档标记：

- `///` 单行分隔符：文档示例和 C# 项目模板使用此形式。 如果分隔符后面有空格，则它不会包含在 XML 输出中。

  > **备注**
  >
  > Visual Studio 会自动插入 `<summary>` 和 `</summary>` 标记，你在代码编辑器中键入 `///` 分隔符后，光标将置于这些标记中。 可以在**"选项"对话框**中打开/关闭此功能。

- `/** */` 多行分隔符：`/** */` 分隔符具有以下格式规则：

  - 在包含 `/**` 分隔符的行上，如果行的其余部分为空格，则不将此行作为注释处理。 如果 `/**` 分隔符后面的第一个字符为空格，则忽略此空格字符，并处理行的其余部分。 否则，将 `/**` 分隔符后面的行的所有文本作为注释的一部分进行处理。

  - 在包含 `*/` 分隔符的行中，如果 `*/` 分隔符前面只有空格，此行将被忽略。 否则，将 `*/` 分隔符之前的行的文本作为注释的一部分进行处理。

  - 对于以 `/**` 分隔符开头的行后面的行，编译器在各行的开头寻找共同模式。 此模式可以包含空格和星号 (`*`)，后面跟更多空格。 如果编译器在不以 `/**` 分隔符开头或不以 `*/` 分隔符结尾的各行开头找到共同模式，则忽略此每个行的模式。

  - 以下注释中将被处理的唯一部分是以 `<summary>` 开头的行。 三种标记格式产生的注释相同。

    C#

    ```
    /** <summary>text</summary> */

    /**
    <summary>text</summary>
    */

    /**
    * <summary>text</summary>
    */
    ```

  - 编译器识别出第二和第三行开头的共同模式"*"。 此模式不包括在输出中。

    C#

    ```
    /**
    * <summary>
    ```

```
* text </summary>*/
```

- 编译器在下面的注释中未找到共同模式，因为第三行的第二个字符不是一个星号。 第二和第三行上的所有文本将处理为注释的一部分。

```
C#

/**
* <summary>
   text </summary>
*/
```

- 编译器在以下注释中未找到模式，原因有两个。 首先，星号前的空格数不一致。 其次，第 5 行以制表符开头，这与空格不匹配。 第二到第五行的所有文本都作为注释的一部分进行处理。

```
C#

/**
  * <summary>
  * text
*  text2
    *  </summary>
*/
```

若要引用 XML 元素（例如，你的函数将处理你要在 XML 文档注释中描述的特定 XML 元素），你可使用标准引用机制（`<` 和 `>`）。 若要引用代码引用（`cref`）元素中的通用标识符，可使用转义字符（例如，`cref="List<T>"`）或大括号（`cref="List{T}"`）。 作为特例，编译器会将大括号解析为尖括号以在引用通用标识符时使作者能够更轻松地进行文档注释。

> **备注**
>
> XML 文档注释不是元数据；它们不包括在编译的程序集中，因此无法通过反射对其进行访问。

接受 XML 文档输入的工具

以下工具通过 XML 注释创建输出：

- DocFX：DocFX 是适用于 .NET 的 API 文档生成器，当前支持 C#、Visual Basic 和 F#。 它还支持自定义生成的引用文档。 DocFX 通过源代码和 Markdown 文件生成

静态 HTML 网站。 此外，借助 DocFX 还可以灵活地通过模板自定义网站布局和样式。 也可以创建自定义模板。

- Sandcastle：Sandcastle 工具为包含概念性和 API 参考页面的托管类库创建帮助文件。 Sandcastle 工具是基于命令行的工具，它没有 GUI 前端、项目管理功能或自动构建过程。 Sandcastle 帮助文件生成器提供独立的 GUI 和基于命令行的工具，以自动方式生成帮助文件。 它还可以使用 Visual Studio 集成包，以完全从 Visual Studio 内创建和管理帮助项目。
- Doxygen：*Doxygen* 通过一系列已记录的源文件生成在线文档浏览器（使用 HTML）或离线参考手册（使用 LaTeX）。 此外，还支持生成 RTF (MS Word)、PostScript、hyperlinked PDF、compressed HTML、DocBook 和 Unix 手册页形式的输出。 可以将 Doxygen 配置为从未记录的源文件中提取代码结构。

ID 字符串

每个类型或成员均存储在输出 XML 文件的元素中。 其中每个元素都具有唯一的 ID 字符串用于标识类型或成员。 ID 字符串必须考虑运算符、参数、返回值、泛型类型参数、`ref`、`in` 和 `out` 参数。 若要对所有这些潜在元素进行编码，编译器应遵循明确定义的用于生成 ID 字符串的规则。 处理 XML 文件的程序使用 ID 字符串来标识文档应用于的相应 .NET 元数据或反射项目。

编译器在生成 ID 字符串时应遵循以下规则：

- 字符串不得包含空白符。
- 该字符串的第一部分使用单个字符后跟冒号来标识成员类型。 使用下面的成员类型：

展开表

字符	成员类型	说明
N	namespace	无法将文档注释添加到命名空间中，但可以在支持的情况下对它们进行 cref 引用。
T	type	类型是类、接口、结构、枚举或委托。
F	Field — 字段	
P	property	包括索引器或其他索引属性。
M	method	包括特殊方法，如构造函数和运算符。
E	event	

字符	成员类型	说明
!	错误字符串	字符串的其余部分提供有关错误的信息。 C# 编译器将生成无法解析的链接的错误信息。

- 该字符串的第二部分是项目的完全限定名称，从命名空间的根开始。 用句点分隔项目名称、其封闭类型和命名空间。 如果项目名称本身包含句点，会将其替换为哈希符号 ('#')。 假定没有名称中恰好包含哈希符号的项目。 例如，String 构造函数的完全限定名称是“System.String.#ctor”。

- 对于属性和方法，括号内的参数列表如下。 如果没有任何参数，则不会出现括号。 这些参数以逗号分隔。 每个参数的编码直接遵循它在 .NET 签名中的编码方式（请参阅 Microsoft.VisualStudio.CorDebugInterop.CorElementType，了解以下列表中的所有大写元素的定义）：
  - 基类型。 常规类型（`ELEMENT_TYPE_CLASS` 或 `ELEMENT_TYPE_VALUETYPE`）表示为类型的完全限定名称。
  - 内部类型（例如 `ELEMENT_TYPE_I4`、`ELEMENT_TYPE_OBJECT`、`ELEMENT_TYPE_STRING`、`ELEMENT_TYPE_TYPEDBYREF` 和 `ELEMENT_TYPE_VOID`）表示为相应完整类型的完全限定的名称。 例如，`System.Int32` 或 `System.TypedReference`。
  - `ELEMENT_TYPE_PTR` 表示为修改类型之后的“*”。
  - `ELEMENT_TYPE_BYREF` 表示为修改类型之后的“@”。
  - `ELEMENT_TYPE_CMOD_OPT` 表示为“!”和修饰符类的完全限定名称，前面是修改类型。
  - `ELEMENT_TYPE_SZARRAY` 表示为“[]”，前面是数组的元素类型。
  - `ELEMENT_TYPE_ARRAY` 表示为 [lowerbound:`size`,*lowerbound*:`size`]，其中逗号的数量是秩 - 1，每个维度的下限和大小（如果已知）以十进制形式表示。 如果未指定下限或大小，则将其省略。 如果省略某个特定维度的下限和大小，也会省略“:”。 例如，以 1 作为下限并且未指定大小的二维数组是 [1:,1:]。

- 仅针对转换运算符（`op_Implicit` 和 `op_Explicit`），该方法的返回值被编码为 `~`，后面是返回类型。 例如，`<member name="M:System.Decimal.op_Explicit(System.Decimal arg)~System.Int32">` 是在 `System.Decimal` 类中声明的强制转换运算符 `public static explicit operator int (decimal value);` 的标记。

- 对于泛型类型，类型名称后跟反引号，然后是指示泛型类型参数数量的一个数字。 例如，`<member name="T:SampleClass``2">` 是定义为 `public class SampleClass<T, U>` 的类型的标记。 对于将泛型类型用作参数的方法，泛型类型参数被指定为前面

加上反引号的数字（例如 `0、`1）。 每个数字表示该类型的泛型参数的从零开始的数组表示法。

- `ELEMENT_TYPE_PINNED` 表示为修改类型之后的“^”。 C# 编译器不会生成此编码。
- `ELEMENT_TYPE_CMOD_REQ` 表示为“|”和修饰符类的完全限定名称，前面是修改类型。 C# 编译器不会生成此编码。
- `ELEMENT_TYPE_GENERICARRAY` 表示为“[?]”，前面是数组的元素类型。 C# 编译器不会生成此编码。
- `ELEMENT_TYPE_FNPTR` 表示为“=FUNC:`type`(*signature*)”，其中 `type` 是返回类型，signature 是方法的参数。 如果没有任何自变量，则省略括号。 C# 编译器不会生成此编码。
- 不表示下列签名组件，因为它们不会用于区分重载方法：
  - 调用约定
  - 返回类型
  - `ELEMENT_TYPE_SENTINEL`

下面的示例演示如何为类及其成员生成 ID 字符串：

C#

```
namespace MyNamespace
{
    /// <summary>
    /// Enter description here for class X.
    /// ID string generated is "T:MyNamespace.MyClass".
    /// </summary>
    public unsafe class MyClass
    {
        /// <summary>
        /// Enter description here for the first constructor.
        /// ID string generated is "M:MyNamespace.MyClass.#ctor".
        /// </summary>
        public MyClass() { }

        /// <summary>
        /// Enter description here for the second constructor.
        /// ID string generated is
"M:MyNamespace.MyClass.#ctor(System.Int32)".
        /// </summary>
        /// <param name="i">Describe parameter.</param>
        public MyClass(int i) { }

        /// <summary>
        /// Enter description here for field message.
        /// ID string generated is "F:MyNamespace.MyClass.message".
        /// </summary>
        public string? message;

        /// <summary>
        /// Enter description for constant PI.
```

```
        /// ID string generated is "F:MyNamespace.MyClass.PI".
        /// </summary>
        public const double PI = 3.14;

        /// <summary>
        /// Enter description for method func.
        /// ID string generated is "M:MyNamespace.MyClass.func".
        /// </summary>
        /// <returns>Describe return value.</returns>
        public int func() { return 1; }

        /// <summary>
        /// Enter description for method someMethod.
        /// ID string generated is
"M:MyNamespace.MyClass.someMethod(System.String,System.Int32@,System.Void*)"
.
        /// </summary>
        /// <param name="str">Describe parameter.</param>
        /// <param name="num">Describe parameter.</param>
        /// <param name="ptr">Describe parameter.</param>
        /// <returns>Describe return value.</returns>
        public int someMethod(string str, ref int nm, void* ptr) { return 1;
}

        /// <summary>
        /// Enter description for method anotherMethod.
        /// ID string generated is
"M:MyNamespace.MyClass.anotherMethod(System.Int16[],System.Int32[0:,0:])".
        /// </summary>
        /// <param name="array1">Describe parameter.</param>
        /// <param name="array">Describe parameter.</param>
        /// <returns>Describe return value.</returns>
        public int anotherMethod(short[] array1, int[,] array) { return 0; }

        /// <summary>
        /// Enter description for operator.
        /// ID string generated is
"M:MyNamespace.MyClass.op_Addition(MyNamespace.MyClass,MyNamespace.MyClass)"
.
        /// </summary>
        /// <param name="first">Describe parameter.</param>
        /// <param name="second">Describe parameter.</param>
        /// <returns>Describe return value.</returns>
        public static MyClass operator +(MyClass first, MyClass second) {
return first; }

        /// <summary>
        /// Enter description for property.
        /// ID string generated is "P:MyNamespace.MyClass.prop".
        /// </summary>
        public int prop { get { return 1; } set { } }

        /// <summary>
        /// Enter description for event.
        /// ID string generated is "E:MyNamespace.MyClass.OnHappened".
```

```
        /// </summary>
        public event Del? OnHappened;

        /// <summary>
        /// Enter description for index.
        /// ID string generated is
"P:MyNamespace.MyClass.Item(System.String)".
        /// </summary>
        /// <param name="str">Describe parameter.</param>
        /// <returns></returns>
        public int this[string s] { get { return 1; } }

        /// <summary>
        /// Enter description for class Nested.
        /// ID string generated is "T:MyNamespace.MyClass.Nested".
        /// </summary>
        public class Nested { }

        /// <summary>
        /// Enter description for delegate.
        /// ID string generated is "T:MyNamespace.MyClass.Del".
        /// </summary>
        /// <param name="i">Describe parameter.</param>
        public delegate void Del(int i);

        /// <summary>
        /// Enter description for operator.
        /// ID string generated is
"M:MyNamespace.MyClass.op_Explicit(MyNamespace.MyClass)~System.Int32".
        /// </summary>
        /// <param name="myParameter">Describe parameter.</param>
        /// <returns>Describe return value.</returns>
        public static explicit operator int(MyClass myParameter) { return 1;
}
    }
}
```

C# 语言规范

有关详细信息，请参阅文档注释上的 C# 语言规范附录。

反馈

此页面是否有帮助? 是 否

提供产品反馈

建议用于 C# 文档注释的 XML 标记

项目 • 2024/08/15

C# 文档注释使用 XML 元素来定义输出文档的结构。 此功能的一个结果是可以在文档注释中添加任何有效的 XML。 C# 编译器会将这些元素复制到输出 XML 文件中。 虽然可以在注释中使用任何有效的 XML（包括任何有效的 HTML 元素），但出于许多原因，建议编制代码文档。

接下来将介绍一些建议、常规使用方案，以及在 C# 代码中使用 XML 文档标记时的需知内容。 虽然可以将任何标记放入文档注释中，但本文介绍了建议用于最常用语言构造的标记。 在所有情况下，都应遵循以下建议：

- 为保持一致性，应编制所有公共可见类型及其公共成员的文档。
- 还可使用 XML 注释编制私有成员的文档。 但这会公开库的内部（很可能是机密信息）运作情况。
- 类型及其成员至少应该有一个 `<summary>` 标记。
- 应使用句号结尾的完整句子编写文档文本。
- 完全支持分部类，并且每个类型的文档信息会串联为单个条目。 如果分部成员的两个声明都有文档注释，则有关实现声明的注释将写入输出 XML。

XML 文档以 `///` 开头。 创建新项目时，模板会放置一些以 `///` 开头的行。 处理这些注释时存在一些限制：

- 文档必须是格式正确的 XML。 如果 XML 格式不正确，编译器将生成警告。 文档文件包含一条注释，指出遇到错误。
- 部分建议标记具有特殊含义：
  - `<param>` 标记用于描述参数。 如果已使用，编译器会验证该参数是否存在，以及文档是否描述了所有参数。 如果验证失败，编译器会发出警告。
  - `cref` 属性可以附加到任何标记，以引用代码元素。 编译器验证此代码元素是否存在。 如果验证失败，编译器会发出警告。 编译器在查找 `cref` 属性中描述的类型时会考虑 `using` 指令。
  - `<summary>` 标记由 Visual Studio 中的 IntelliSense 用于显示有关某个类型或成员的附加信息。

    > **备注**
    >
    > XML 文件不提供有关该类型和成员的完整信息（例如，它不包含任何类型信息）。 若要获取有关类型或成员的完整信息，请将文档文件与对实际类型或成员的反射一起使用。

- 开发人员可以随意创建自己的标记集。 编译器会将这些标记复制到输出文件。

部分建议的标记可用于任何语言元素。 其他建议的标记具有更特殊的用法。 最后，某些标记用于设置文档中文本的格式。 本文介绍了按用途整理的建议标记。

编译器验证以下列表中后跟单个 * 的元素的语法。 Visual Studio 为编译器验证的标记以及以下列表中后跟 ** 的所有标记提供 IntelliSense。 除此处列出的标记外，编译器和 Visual Studio 还验证 `<b>`、`<i>`、`<u>`、`<br/>` 和 `<a>` 标记。 编译器还会对已弃用的 `<tt>` 进行验证。

- 用于多个元素的常规标记 - 这些标记是任何 API 的最小集。
  - <summary>：此元素的值显示在 Visual Studio 中的 IntelliSense 中。
  - <remarks> **
- 用于成员的标记 - 记录方法和属性时使用这些标记。
  - <returns>：此元素的值显示在 Visual Studio 中的 IntelliSense 中。
  - <param>*：此元素的值显示在 Visual Studio 中的 IntelliSense 中。
  - <paramref>
  - <exception> *
  - <value>：此元素的值显示在 Visual Studio 中的 IntelliSense 中。
- 设置文档输出格式 - 这些标记为生成文档的工具提供格式设置说明。
  - <para>
  - <list>
  - <c>
  - <code>
  - <example> **
- 重用文档文本 - 这些标记提供的工具使重用 XML 注释变得更加容易。
  - <inheritdoc> **
  - <include> *
- 生成链接和引用 - 这些标记生成指向其他文档的链接。
  - <see> *
  - <seealso> *
  - cref
  - href
- 用于泛型类型和方法的标记 - 这些标记仅用于泛型类型和方法
  - <typeparam> *：此元素的值显示在 Visual Studio 中的 IntelliSense 中。
  - <typeparamref>

> **备注**
>
> 不可对命名空间应用文档注释。

如果希望在文档注释的文本中显示尖括号，请使用 `<` 和 `>` 的 HTML 编码，分别为 `<` 和 `>`。 下面的示例对此编码进行了演示。

C#

```
/// <summary>
/// This property always returns a value &lt; 1.
/// </summary>
```

常规标记

<summary>

XML

```
<summary>description</summary>
```

`<summary>` 标记应当用于描述类型或类型成员。 使用 <remarks> 可针对某个类型说明添加补充信息。 使用 cref 属性可启用文档工具（如 DocFX 和 Sandcastle）来创建指向代码元素的文档页的内部超链接。 `<summary>` 标记的文本显示在 IntelliSense 和对象浏览器窗口中。

<remarks>

XML

```
<remarks>
description
</remarks>
```

`<remarks>` 标记用于添加有关某个类型或某个类型成员的信息，从而补充由 <summary> 指定的信息。 此信息显示在对象浏览器窗口中。 此标记可能包含更冗长的说明。 你可能会发现，将 `CDATA` 部分用于 Markdown 可以更方便地进行编写。 docfx 等工具在 `CDATA` 部分中处理 Markdown 文本。

记录成员

<returns>

XML

```
<returns>description</returns>
```

在方法声明的注释中应使用 `<returns>` 标记来描述返回值。

<param>

```XML
<param name="name">description</param>
```

- `name`：方法参数的名称。 给名称加上引号 (")。 参数的名称必须与 API 签名匹配。 如果未涵盖一个或多个参数，编译器将发出警告。 如果 `name` 的值与方法声明中的正式参数不匹配，编译器也会发出警告。

在方法声明的注释中，应使用 `<param>` 标记来描述方法参数之一。 若要记录多个参数，请使用多个 `<param>` 标记。 `<param>` 标记的文本显示在 IntelliSense、对象浏览器和代码注释 Web 报表中。

<paramref>

```XML
<paramref name="name"/>
```

- `name`：要引用的参数的名称。 给名称加上引号 (")。

`<paramref>` 标记提供一种方式，用于指示 `<summary>` 或 `<remarks>` 块等代码注释中的单词引用某个参数。 可以处理 XML 文件以明显的方式设置此单词的格式，如使用粗体或斜体。

<exception>

```XML
<exception cref="member">description</exception>
```

- cref = "`member`"：对当前编译环境中出现的一个异常的引用。 编译器检查是否存在给定的异常，并将 `member` 转换为输出 XML 中的规范的元素名称。 `member` 必须显示在引号 (") 内。

`<exception>` 标记可用于指定可引发的异常。 此标记可应用于方法、属性、事件和索引器的定义。

<value>

XML

```xml
<value>property-description</value>
```

`<value>` 标记可用于描述属性表示的值。 在 Visual Studio .NET 开发环境中通过代码向导添加属性时，将添加新属性的 <summary> 标记。 可手动添加 `<value>` 标记，以描述属性表示的值。

设置文档输出格式

<para>

XML

```xml
<remarks>
    <para>
        This is an introductory paragraph.
    </para>
    <para>
        This paragraph contains more details.
    </para>
</remarks>
```

`<para>` 标记在标记内使用，例如 <summary>、<remarks> 或 <returns>，并且你可用它来向文本添加结构。 `<para>` 标记创建一个双空格段落。 如果需要单空格段落，请使用 `<br/>` 标记。

<list>

XML

```xml
<list type="bullet|number|table">
    <listheader>
        <term>term</term>
        <description>description</description>
    </listheader>
    <item>
        <term>Assembly</term>
        <description>The library or executable built from a compilation.
</description>
```

```
    </item>
</list>
```

`<listheader>` 块用于定义表或定义列表的标题行。

定义表时：

- 只需为标题中的 `term` 提供条目。
- 列表中的每个项均使用 `<item>` 块指定。 对于每个 `item`，您只需为 `description` 提供一个条目。

创建定义列表时：

- 必须为标题中的 `term` 提供条目。
- 列表中的每个项均使用 `<item>` 块指定。 每个 `item` 必须同时包含 `term` 和 `description`。

列表或表可根据需要具有多个 `<item>` 块。

<c>

XML

```
<c>text</c>
```

使用 `<c>` 标记可以指示应将说明内的文本标记为代码。 使用 <code> 指示作为代码的多行文本。

<code>

XML

```
<code>
    var index = 5;
    index++;
</code>
```

`<code>` 标记用于指示多行代码。 使用 <c> 指示应将说明内的单行文本标记为代码。

<example>

XML

```
<example>
This shows how to increment an integer.
<code>
    var index = 5;
    index++;
</code>
</example>
```

借助 `<example>` 标记，可以指定如何使用方法或其他库成员的示例。 示例通常涉及到使用 <code> 标记。

重用文档文本

<inheritdoc>

XML

```
<inheritdoc [cref=""] [path=""]/>
```

继承基类、接口和类似方法中的 XML 注释。 使用 `inheritdoc` 不必复制和粘贴重复的 XML 注释，并自动保持 XML 注释同步。 将标记 `<inheritdoc>` 添加到类型时，所有成员也将继承注释。

- `cref`：指定要向其继承文档的成员。 当前成员上已定义的标记不会被继承的标记重写。
- `path`：导致显示节点集的 XPath 表达式查询。 可以使用此属性筛选要在继承文档中包含或排除的标记。

在基类或接口中添加 XML 注释，并让 inheritdoc 将注释复制到实现类中。 向同步方法添加 XML 注释，并让 inheritdoc 将注释复制到相同方法的异步版本中。 如果要从特定成员复制注释，可以使用 `cref` 特性来指定成员。

<include>

XML

```
<include file='filename' path='tagpath[@name="id"]' />
```

- `filename`：包含文档的 XML 文件的名称。 可使用相对于源代码文件的路径限定文件名。 使用单引号 (' ') 将 `filename` 括起来。

- `tagpath`： `filename` 中标记的路径，它指向标记 `name`。 使用单引号 (' ') 将路径括起来。
- `name`：标记中的名称说明符（位于注释之前）； `name` 有 `id`。
- `id`：标记的 ID（位于注释之前）。 给 ID 加上引号 (")。

通过 `<include>` 标记，可在其他文件中引用描述源代码中类型和成员的注释。 包含外部文件是对直接在源代码文件中放入文档注释的替代方法。 通过将文档放入不同文件，可以单独从源代码对文档应用源控件。 一人可以签出源代码文件，而其他人可以签出文档文件。 `<include>` 标记使用 XML XPath 语法。 有关如何自定义 `<include>` 用法的信息，请参阅 XPath 文档。

例如，以下源代码使用 `<include>` 标记来包含注解。 文件路径是相对于源的。

C#

```
namespace MyNamespace;

public class MyType
{
    /// <returns>This is the returns text of MyMethod. It comes from triple
slash comments.</returns>
    /// <remarks>This is the remarks text of MyMethod. It comes from triple
slash comments.</remarks>
    /// <include file="MyAssembly.xml"
path="doc/members/member[@name='M:MyNamespace.MyType.MyMethod']/*" />
    public int MyMethod(int p) => p;
}
```

包含文件的 XML 源代码显示在下面的示例中。 它的结构与 C# 编译器生成的 XML 文件相同。 XML 文件可以包含多个方法或类型的文本，只要 XPath 表达式可识别它们。

XML

```
<?xml version="1.0"?>
<doc>
    <members>
        <member name="M:MyNamespace.MyType.MyMethod">
            <param name="p">This is the description of the parameter p of
MyMethod. It comes from the included file.</param>
            <summary>This is the summary of MyMethod. It comes from the
included file.</summary>
        </member>
    </members>
</doc>
```

此方法的 XML 输出如以下示例所示：

XML

```xml
<member name="M:MyNamespace.MyType.MyMethod(System.Int32)">
    <summary>This is the summary of MyMethod. It comes from the included
file.</summary>
    <returns>This is the returns text of MyMethod. It comes from triple
slash comments.</returns>
    <remarks>This is the remarks text of MyMethod. It comes from triple
slash comments.</remarks>
    <param name="p">This is the description of the parameter p of MyMethod.
It comes from the included file.</param>
</member>
```

提示

.NET 运行时团队在文档中广泛使用了 `<include>` 标记。 通过**搜索 dotnet/runtime 存储库**，可以看到许多示例。

生成链接和引用

<see>

XML

```xml
<see cref="member"/>
<!-- or -->
<see cref="member">Link text</see>
<!-- or -->
<see href="link">Link Text</see>
<!-- or -->
<see langword="keyword"/>
```

- `cref="member"`：对可从当前编译环境调用的成员或字段的引用。 编译器检查是否存在给定的码位元素，并将 `member` 传递到输出 XML 中的元素名称。 将成员置于引号 (") 内。 可以使用单独的结束标记为“cref”提供不同的链接文本。
- `href="link"`：指向给定 URL 的可单击链接。 例如，`<see href="https://github.com">GitHub</see>` 生成一个可单击的链接，其中包含文本 GitHub，该文本链接到 `https://github.com`。
- `langword="keyword"`：语言关键字，例如 `true` 或其他有效关键字之一。

`<see>` 标记可用于从文本内指定链接。 使用 <seealso> 指示文本应该放在“另请参阅”部分中。 使用 cref 属性创建指向代码元素的文档页的内部超链接。 包含类型参数以指定对

泛型类型或方法（如 `cref="IDictionary{T, U}"`）的引用。 此外，`href` 还是一个有效属性，用作超链接。

<seealso>

```xml
<seealso cref="member"/>
<!-- or -->
<seealso href="link">Link Text</seealso>
```

- `cref="member"`：对可从当前编译环境调用的成员或字段的引用。 编译器检查是否存在给定的码位元素，并将 `member` 传递到输出 XML 中的元素名称。 `member` 必须显示在引号 (") 内。
- `href="link"`：指向给定 URL 的可单击链接。 例如，`<seealso href="https://github.com">GitHub</seealso>` 生成一个可单击的链接，其中包含文本 GitHub，该文本链接到 `https://github.com`。

使用 `<seealso>` 标记，可以指定想要在**另请参阅**部分中显示的文本。 使用 <see> 从文本内指定链接。 不能将 `seealso` 标记嵌套在 `summary` 标记内。

cref 特性

XML 文档标记中的 `cref` 属性表示“代码引用”。它指定标记的内部文本是代码元素，例如类型、方法或属性。 文档工具（例如 DocFX 和 Sandcastle）使用 `cref` 属性自动生成指向记录类型或成员的页面的超链接。

href 特性

`href` 特性表示对网页的引用。 可以使用它直接引用有关 API 或库的联机文档。

泛型类型和方法

<typeparam>

```xml
<typeparam name="TResult">The type returned from this method</typeparam>
```

- `TResult`：类型参数的名称。 给名称加上引号 (")。

在泛型类型或方法声明的注释中，应使用 `<typeparam>` 标记来描述类型参数。 为泛型类型或方法的每个类型参数添加标记。 `<typeparam>` 标记的文本显示在 IntelliSense 中。

<typeparamref>

```xml
<typeparamref name="TKey"/>
```

- `TKey`：类型参数的名称。 给名称加上引号 (")。

通过此标记，文档文件的使用者可显著设置字体格式，例如采用斜体。

用户定义的标记

本文概述的所有标记均表示由 C# 编译器识别的标记。 但用户可随意定义自己的标记。 Sandcastle 等工具支持其他标记，例如 <event> 和 <note>，甚至支持编制命名空间文档。 自定义或内部文档生成工具也可与标准标记配合使用，并支持 HTML 到 PDF 等多种输出格式。

示例 XML 文档注释

项目 • 2023/03/28

本文包含三个示例，演示如何向大多数 C# 语言元素添加 XML 文档注释。 第一个示例演示如何记录具有不同成员的类。 第二个示例演示如何重用类或接口层次结构的说明。 第三个示例展示用于泛型类和成员的标记。 第二个和第三个示例使用第一个示例中介绍的概念。

记录类、结构或接口

下面的示例展示了公共语言元素，以及可能用于描述这些元素的标记。 文档注释说明标记的使用，而不是类本身。

C#

```
    /// <summary>
    /// Every class and member should have a one sentence
    /// summary describing its purpose.
    /// </summary>
    /// <remarks>
    /// You can expand on that one sentence summary to
    /// provide more information for readers. In this case,
    /// the <c>ExampleClass</c> provides different C#
    /// elements to show how you would add documentation
    ///comments for most elements in a typical class.
    /// <para>
    /// The remarks can add multiple paragraphs, so you can
    /// write detailed information for developers that use
    /// your work. You should add everything needed for
    /// readers to be successful. This class contains
    /// examples for the following:
    /// </para>
    /// <list type="table">
    /// <item>
    /// <term>Summary</term>
    /// <description>
    /// This should provide a one sentence summary of the class or member.
    /// </description>
    /// </item>
    /// <item>
    /// <term>Remarks</term>
    /// <description>
    /// This is typically a more detailed description of the class or member
    /// </description>
    /// </item>
    /// <item>
    /// <term>para</term>
    /// <description>
```

```
    /// The para tag separates a section into multiple paragraphs
    /// </description>
    /// </item>
    /// <item>
    /// <term>list</term>
    /// <description>
    /// Provides a list of terms or elements
    /// </description>
    /// </item>
    /// <item>
    /// <term>returns, param</term>
    /// <description>
    /// Used to describe parameters and return values
    /// </description>
    /// </item>
    /// <item>
    /// <term>value</term>
    /// <description>Used to describe properties</description>
    /// </item>
    /// <item>
    /// <term>exception</term>
    /// <description>
    /// Used to describe exceptions that may be thrown
    /// </description>
    /// </item>
    /// <item>
    /// <term>c, cref, see, seealso</term>
    /// <description>
    /// These provide code style and links to other
    /// documentation elements
    /// </description>
    /// </item>
    /// <item>
    /// <term>example, code</term>
    /// <description>
    /// These are used for code examples
    /// </description>
    /// </item>
    /// </list>
    /// <para>
    /// The list above uses the "table" style. You could
    /// also use the "bullet" or "number" style. Neither
    /// would typically use the "term" element.
    /// <br/>
    /// Note: paragraphs are double spaced. Use the *br*
    /// tag for single spaced lines.
    /// </para>
    /// </remarks>
    public class ExampleClass
    {
        /// <value>
        /// The <c>Label</c> property represents a label
        /// for this instance.
        /// </value>
        /// <remarks>
```

```
        /// The <see cref="Label"/> is a <see langword="string"/>
        /// that you use for a label.
        /// <para>
        /// Note that there isn't a way to provide a "cref" to
        /// each accessor, only to the property itself.
        /// </para>
        /// </remarks>
        public string? Label
        {
            get;
            set;
        }

        /// <summary>
        /// Adds two integers and returns the result.
        /// </summary>
        /// <returns>
        /// The sum of two integers.
        /// </returns>
        /// <param name="left">
        /// The left operand of the addition.
        /// </param>
        /// <param name="right">
        /// The right operand of the addition.
        /// </param>
        /// <example>
        /// <code>
        /// int c = Math.Add(4, 5);
        /// if (c > 10)
        /// {
        ///     Console.WriteLine(c);
        /// }
        /// </code>
        /// </example>
        /// <exception cref="System.OverflowException">
        /// Thrown when one parameter is
        /// <see cref="Int32.MaxValue">MaxValue</see> and the other is
        /// greater than 0.
        /// Note that here you can also use
        /// <see
href="https://learn.microsoft.com/dotnet/api/system.int32.maxvalue"/>
        ///  to point a web page instead.
        /// </exception>
        /// <see cref="ExampleClass"/> for a list of all
        /// the tags in these examples.
        /// <seealso cref="ExampleClass.Label"/>
        public static int Add(int left, int right)
        {
            if ((left == int.MaxValue && right > 0) || (right ==
int.MaxValue && left > 0))
                throw new System.OverflowException();

            return left + right;
        }
    }
```

```
    /// <summary>
    /// This is an example of a positional record.
    /// </summary>
    /// <remarks>
    /// There isn't a way to add XML comments for properties
    /// created for positional records, yet. The language
    /// design team is still considering what tags should
    /// be supported, and where. Currently, you can use
    /// the "param" tag to describe the parameters to the
    /// primary constructor.
    /// </remarks>
    /// <param name="FirstName">
    /// This tag will apply to the primary constructor parameter.
    /// </param>
    /// <param name="LastName">
    /// This tag will apply to the primary constructor parameter.
    /// </param>
    public record Person(string FirstName, string LastName);
}
```

添加文档可使你的源代码与面向库用户的大量注释混杂在一起。 使用 `<Include>` 标记可将 XML 注释与源代码分隔开来。 源代码引用包含 `<Include>` 标记的 XML 文件：

C#

```
/// <include file='xml_include_tag.xml'
path='MyDocs/MyMembers[@name="test"]/*' />
class Test
{
    static void Main()
    {
    }
}

/// <include file='xml_include_tag.xml'
path='MyDocs/MyMembers[@name="test2"]/*' />
class Test2
{
    public void Test()
    {
    }
}
```

第二个文件 xml_include_tag.xml 包含文档注释。

XML

```
<MyDocs>
    <MyMembers name="test">
        <summary>
```

```
        The summary for this type.
        </summary>
    </MyMembers>
    <MyMembers name="test2">
        <summary>
        The summary for this other type.
        </summary>
    </MyMembers>
</MyDocs>
```

记录类和接口的层次结构

`<inheritdoc>` 元素表示类型或成员从基类或接口继承文档注释。 还可将 `<inheritdoc>` 元素与 `cref` 属性配合使用，以从同一类型的成员继承注释。 下面的示例介绍使用此标志的方法。 请注意，将 `inheritdoc` 属性添加到类型时，将继承成员注释。 可对派生类型的成员编写注释来阻止使用继承的注释。 这些注释将优先于继承的注释被选中。

C#

```
/// <summary>
/// A summary about this class.
/// </summary>
/// <remarks>
/// These remarks would explain more about this class.
/// In this example, these comments also explain the
/// general information about the derived class.
/// </remarks>
public class MainClass
{
}

///<inheritdoc/>
public class DerivedClass : MainClass
{
}

/// <summary>
/// This interface would describe all the methods in
/// its contract.
/// </summary>
/// <remarks>
/// While elided for brevity, each method or property
/// in this interface would contain docs that you want
/// to duplicate in each implementing class.
/// </remarks>
public interface ITestInterface
{
    /// <summary>
    /// This method is part of the test interface.
    /// </summary>
```

```
    /// <remarks>
    /// This content would be inherited by classes
    /// that implement this interface when the
    /// implementing class uses "inheritdoc"
    /// </remarks>
    /// <returns>The value of <paramref name="arg" /> </returns>
    /// <param name="arg">The argument to the method</param>
    int Method(int arg);
}

///<inheritdoc cref="ITestInterface"/>
public class ImplementingClass : ITestInterface
{
    // doc comments are inherited here.
    public int Method(int arg) => arg;
}

/// <summary>
/// This class shows hows you can "inherit" the doc
/// comments from one method in another method.
/// </summary>
/// <remarks>
/// You can inherit all comments, or only a specific tag,
/// represented by an xpath expression.
/// </remarks>
public class InheritOnlyReturns
{
    /// <summary>
    /// In this example, this summary is only visible for this method.
    /// </summary>
    /// <returns>A boolean</returns>
    public static bool MyParentMethod(bool x) { return x; }

    /// <inheritdoc cref="MyParentMethod" path="/returns"/>
    public static bool MyChildMethod() { return false; }
}

/// <Summary>
/// This class shows an example ofsharing comments across methods.
/// </Summary>
public class InheritAllButRemarks
{
    /// <summary>
    /// In this example, this summary is visible on all the methods.
    /// </summary>
    /// <remarks>
    /// The remarks can be inherited by other methods
    /// using the xpath expression.
    /// </remarks>
    /// <returns>A boolean</returns>
    public static bool MyParentMethod(bool x) { return x; }

    /// <inheritdoc cref="MyParentMethod" path="//*[not(self::remarks)]"/>
    public static bool MyChildMethod() { return false; }
}
```

泛型类型

使用 `<typeparam>` 标记描述泛型类型和方法的类型参数。 `cref` 属性的值需要新语法才能引用泛型方法或类：

C#

```
/// <summary>
/// This is a generic class.
/// </summary>
/// <remarks>
/// This example shows how to specify the <see cref="GenericClass{T}"/>
/// type as a cref attribute.
/// In generic classes and methods, you'll often want to reference the
/// generic type, or the type parameter.
/// </remarks>
class GenericClass<T>
{
    // Fields and members.
}

/// <Summary>
/// This shows examples of typeparamref and typeparam tags
/// </Summary>
public class ParamsAndParamRefs
{
    /// <summary>
    /// The GetGenericValue method.
    /// </summary>
    /// <remarks>
    /// This sample shows how to specify the <see cref="GetGenericValue"/>
    /// method as a cref attribute.
    /// The parameter and return value are both of an arbitrary type,
    /// <typeparamref name="T"/>
    /// </remarks>
    public static T GetGenericValue<T>(T para)
    {
        return para;
    }
}
```

Math 类示例

下面的代码真实地演示如何将文档注释添加到 math 库。

C#

```
namespace TaggedLibrary
{
    /*
        The main Math class
        Contains all methods for performing basic math functions
    */
        /// <summary>
    /// The main <c>Math</c> class.
    /// Contains all methods for performing basic math functions.
    /// <list type="bullet">
    /// <item>
    /// <term>Add</term>
    /// <description>Addition Operation</description>
    /// </item>
    /// <item>
    /// <term>Subtract</term>
    /// <description>Subtraction Operation</description>
    /// </item>
    /// <item>
    /// <term>Multiply</term>
    /// <description>Multiplication Operation</description>
    /// </item>
    /// <item>
    /// <term>Divide</term>
    /// <description>Division Operation</description>
    /// </item>
    /// </list>
    /// </summary>
    /// <remarks>
    /// <para>
    /// This class can add, subtract, multiply and divide.
    /// </para>
    /// <para>
    /// These operations can be performed on both
    /// integers and doubles.
    /// </para>
    /// </remarks>
    public class Math
    {
        // Adds two integers and returns the result
        /// <summary>
        /// Adds two integers <paramref name="a"/> and <paramref name="b"/>
        ///  and returns the result.
        /// </summary>
        /// <returns>
        /// The sum of two integers.
        /// </returns>
        /// <example>
        /// <code>
        /// int c = Math.Add(4, 5);
        /// if (c > 10)
        /// {
        ///     Console.WriteLine(c);
        /// }
```

```
        /// </code>
        /// </example>
        /// <exception cref="System.OverflowException">
        /// Thrown when one parameter is <see cref="Int32.MaxValue"/> and
the other
        /// is greater than 0.
        /// </exception>
        /// See <see cref="Math.Add(double, double)"/> to add doubles.
        /// <seealso cref="Math.Subtract(int, int)"/>
        /// <seealso cref="Math.Multiply(int, int)"/>
        /// <seealso cref="Math.Divide(int, int)"/>
        /// <param name="a">An integer.</param>
        /// <param name="b">An integer.</param>
        public static int Add(int a, int b)
        {
            // If any parameter is equal to the max value of an integer
            // and the other is greater than zero
            if ((a == int.MaxValue && b > 0) ||
                (b == int.MaxValue && a > 0))
            {
                throw new System.OverflowException();
            }
            return a + b;
        }

        // Adds two doubles and returns the result
        /// <summary>
        /// Adds two doubles <paramref name="a"/> and <paramref name="b"/>
        /// and returns the result.
        /// </summary>
        /// <returns>
        /// The sum of two doubles.
        /// </returns>
        /// <example>
        /// <code>
        /// double c = Math.Add(4.5, 5.4);
        /// if (c > 10)
        /// {
        ///     Console.WriteLine(c);
        /// }
        /// </code>
        /// </example>
        /// <exception cref="System.OverflowException">
        /// Thrown when one parameter is max and the other
        /// is greater than 0.</exception>
        /// See <see cref="Math.Add(int, int)"/> to add integers.
        /// <seealso cref="Math.Subtract(double, double)"/>
        /// <seealso cref="Math.Multiply(double, double)"/>
        /// <seealso cref="Math.Divide(double, double)"/>
        /// <param name="a">A double precision number.</param>
        /// <param name="b">A double precision number.</param>
        public static double Add(double a, double b)
        {
            // If any parameter is equal to the max value of an integer
            // and the other is greater than zero
```

```
            if ((a == double.MaxValue && b > 0)
                || (b == double.MaxValue && a > 0))
            {
                throw new System.OverflowException();
            }

            return a + b;
        }

        // Subtracts an integer from another and returns the result
        /// <summary>
        /// Subtracts <paramref name="b"/> from <paramref name="a"/>
        /// and returns the result.
        /// </summary>
        /// <returns>
        /// The difference between two integers.
        /// </returns>
        /// <example>
        /// <code>
        /// int c = Math.Subtract(4, 5);
        /// if (c > 1)
        /// {
        ///     Console.WriteLine(c);
        /// }
        /// </code>
        /// </example>
        /// See <see cref="Math.Subtract(double, double)"/> to subtract
doubles.
        /// <seealso cref="Math.Add(int, int)"/>
        /// <seealso cref="Math.Multiply(int, int)"/>
        /// <seealso cref="Math.Divide(int, int)"/>
        /// <param name="a">An integer.</param>
        /// <param name="b">An integer.</param>
        public static int Subtract(int a, int b)
        {
            return a - b;
        }

        // Subtracts a double from another and returns the result
        /// <summary>
        /// Subtracts a double <paramref name="b"/> from another
        /// double <paramref name="a"/> and returns the result.
        /// </summary>
        /// <returns>
        /// The difference between two doubles.
        /// </returns>
        /// <example>
        /// <code>
        /// double c = Math.Subtract(4.5, 5.4);
        /// if (c > 1)
        /// {
        ///     Console.WriteLine(c);
        /// }
        /// </code>
        /// </example>
```

```
        /// See <see cref="Math.Subtract(int, int)"/> to subtract integers.
        /// <seealso cref="Math.Add(double, double)"/>
        /// <seealso cref="Math.Multiply(double, double)"/>
        /// <seealso cref="Math.Divide(double, double)"/>
        /// <param name="a">A double precision number.</param>
        /// <param name="b">A double precision number.</param>
        public static double Subtract(double a, double b)
        {
            return a - b;
        }

        // Multiplies two integers and returns the result
        /// <summary>
        /// Multiplies two integers <paramref name="a"/>
        /// and <paramref name="b"/> and returns the result.
        /// </summary>
        /// <returns>
        /// The product of two integers.
        /// </returns>
        /// <example>
        /// <code>
        /// int c = Math.Multiply(4, 5);
        /// if (c > 100)
        /// {
        ///     Console.WriteLine(c);
        /// }
        /// </code>
        /// </example>
        /// See <see cref="Math.Multiply(double, double)"/> to multiply
doubles.
        /// <seealso cref="Math.Add(int, int)"/>
        /// <seealso cref="Math.Subtract(int, int)"/>
        /// <seealso cref="Math.Divide(int, int)"/>
        /// <param name="a">An integer.</param>
        /// <param name="b">An integer.</param>
        public static int Multiply(int a, int b)
        {
            return a * b;
        }

        // Multiplies two doubles and returns the result
        /// <summary>
        /// Multiplies two doubles <paramref name="a"/> and
        /// <paramref name="b"/> and returns the result.
        /// </summary>
        /// <returns>
        /// The product of two doubles.
        /// </returns>
        /// <example>
        /// <code>
        /// double c = Math.Multiply(4.5, 5.4);
        /// if (c > 100.0)
        /// {
        ///     Console.WriteLine(c);
        /// }
```

```
        /// </code>
        /// </example>
        /// See <see cref="Math.Multiply(int, int)"/> to multiply integers.
        /// <seealso cref="Math.Add(double, double)"/>
        /// <seealso cref="Math.Subtract(double, double)"/>
        /// <seealso cref="Math.Divide(double, double)"/>
        /// <param name="a">A double precision number.</param>
        /// <param name="b">A double precision number.</param>
        public static double Multiply(double a, double b)
        {
            return a * b;
        }

        // Divides an integer by another and returns the result
        /// <summary>
        /// Divides an integer <paramref name="a"/> by another
        /// integer <paramref name="b"/> and returns the result.
        /// </summary>
        /// <returns>
        /// The quotient of two integers.
        /// </returns>
        /// <example>
        /// <code>
        /// int c = Math.Divide(4, 5);
        /// if (c > 1)
        /// {
        ///     Console.WriteLine(c);
        /// }
        /// </code>
        /// </example>
        /// <exception cref="System.DivideByZeroException">
        /// Thrown when <paramref name="b"/> is equal to 0.
        /// </exception>
        /// See <see cref="Math.Divide(double, double)"/> to divide doubles.
        /// <seealso cref="Math.Add(int, int)"/>
        /// <seealso cref="Math.Subtract(int, int)"/>
        /// <seealso cref="Math.Multiply(int, int)"/>
        /// <param name="a">An integer dividend.</param>
        /// <param name="b">An integer divisor.</param>
        public static int Divide(int a, int b)
        {
            return a / b;
        }

        // Divides a double by another and returns the result
        /// <summary>
        /// Divides a double <paramref name="a"/> by another double
        /// <paramref name="b"/> and returns the result.
        /// </summary>
        /// <returns>
        /// The quotient of two doubles.
        /// </returns>
        /// <example>
        /// <code>
        /// double c = Math.Divide(4.5, 5.4);
```

```
        /// if (c > 1.0)
        /// {
        ///     Console.WriteLine(c);
        /// }
        /// </code>
        /// </example>
        /// <exception cref="System.DivideByZeroException">
        /// Thrown when <paramref name="b"/> is equal to 0.
        /// </exception>
        /// See <see cref="Math.Divide(int, int)"/> to divide integers.
        /// <seealso cref="Math.Add(double, double)"/>
        /// <seealso cref="Math.Subtract(double, double)"/>
        /// <seealso cref="Math.Multiply(double, double)"/>
        /// <param name="a">A double precision dividend.</param>
        /// <param name="b">A double precision divisor.</param>
        public static double Divide(double a, double b)
        {
            return a / b;
        }
    }
}
```

你可能会发现代码已被所有注释遮盖。 最后一个示例演示如何调整此库以使用 `include` 标记。 将所有文档移到 XML 文件中：

XML

```
<docs>
  <members name="math">
   <Math>
    <summary>
      The main <c>Math</c> class.
      Contains all methods for performing basic math functions.
      </summary>
      <remarks>
      <para>This class can add, subtract, multiply and divide.</para>
      <para>These operations can be performed on both integers and doubles.
</para>
      </remarks>
    </Math>
    <AddInt>
      <summary>
      Adds two integers <paramref name="a"/> and <paramref name="b"/>
      and returns the result.
      </summary>
      <returns>
      The sum of two integers.
      </returns>
      <example>
      <code>
      int c = Math.Add(4, 5);
      if (c > 10)
      {
```

```
        Console.WriteLine(c);
      }
      </code>
      </example>
      <exception cref="System.OverflowException">Thrown when one
      parameter is max
      and the other is greater than 0.</exception>
      See <see cref="Math.Add(double, double)"/> to add doubles.
      <seealso cref="Math.Subtract(int, int)"/>
      <seealso cref="Math.Multiply(int, int)"/>
      <seealso cref="Math.Divide(int, int)"/>
      <param name="a">An integer.</param>
      <param name="b">An integer.</param>
    </AddInt>
    <AddDouble>
      <summary>
      Adds two doubles <paramref name="a"/> and <paramref name="b"/>
      and returns the result.
      </summary>
      <returns>
      The sum of two doubles.
      </returns>
      <example>
      <code>
      double c = Math.Add(4.5, 5.4);
      if (c > 10)
      {
        Console.WriteLine(c);
      }
      </code>
      </example>
      <exception cref="System.OverflowException">Thrown when one parameter
is max
      and the other is greater than 0.</exception>
      See <see cref="Math.Add(int, int)"/> to add integers.
      <seealso cref="Math.Subtract(double, double)"/>
      <seealso cref="Math.Multiply(double, double)"/>
      <seealso cref="Math.Divide(double, double)"/>
      <param name="a">A double precision number.</param>
      <param name="b">A double precision number.</param>
    </AddDouble>
    <SubtractInt>
      <summary>
      Subtracts <paramref name="b"/> from <paramref name="a"/> and
      returns the result.
      </summary>
      <returns>
      The difference between two integers.
      </returns>
      <example>
      <code>
      int c = Math.Subtract(4, 5);
      if (c > 1)
      {
        Console.WriteLine(c);
```

```
    }
    </code>
    </example>
    See <see cref="Math.Subtract(double, double)"/> to subtract doubles.
    <seealso cref="Math.Add(int, int)"/>
    <seealso cref="Math.Multiply(int, int)"/>
    <seealso cref="Math.Divide(int, int)"/>
    <param name="a">An integer.</param>
    <param name="b">An integer.</param>
  </SubtractInt>
  <SubtractDouble>
    <summary>
    Subtracts a double <paramref name="b"/> from another
    double <paramref name="a"/> and returns the result.
    </summary>
    <returns>
    The difference between two doubles.
    </returns>
    <example>
    <code>
    double c = Math.Subtract(4.5, 5.4);
    if (c > 1)
    {
      Console.WriteLine(c);
    }
    </code>
    </example>
    See <see cref="Math.Subtract(int, int)"/> to subtract integers.
    <seealso cref="Math.Add(double, double)"/>
    <seealso cref="Math.Multiply(double, double)"/>
    <seealso cref="Math.Divide(double, double)"/>
    <param name="a">A double precision number.</param>
    <param name="b">A double precision number.</param>
  </SubtractDouble>
  <MultiplyInt>
    <summary>
    Multiplies two integers <paramref name="a"/> and
    <paramref name="b"/> and returns the result.
    </summary>
    <returns>
    The product of two integers.
    </returns>
    <example>
    <code>
    int c = Math.Multiply(4, 5);
    if (c > 100)
    {
      Console.WriteLine(c);
    }
    </code>
    </example>
    See <see cref="Math.Multiply(double, double)"/> to multiply doubles.
    <seealso cref="Math.Add(int, int)"/>
    <seealso cref="Math.Subtract(int, int)"/>
    <seealso cref="Math.Divide(int, int)"/>
```

```
    <param name="a">An integer.</param>
    <param name="b">An integer.</param>
  </MultiplyInt>
  <MultiplyDouble>
    <summary>
    Multiplies two doubles <paramref name="a"/> and
    <paramref name="b"/> and returns the result.
    </summary>
    <returns>
    The product of two doubles.
    </returns>
    <example>
    <code>
    double c = Math.Multiply(4.5, 5.4);
    if (c > 100.0)
    {
      Console.WriteLine(c);
    }
    </code>
    </example>
    See <see cref="Math.Multiply(int, int)"/> to multiply integers.
    <seealso cref="Math.Add(double, double)"/>
    <seealso cref="Math.Subtract(double, double)"/>
    <seealso cref="Math.Divide(double, double)"/>
    <param name="a">A double precision number.</param>
    <param name="b">A double precision number.</param>
  </MultiplyDouble>
  <DivideInt>
    <summary>
    Divides an integer <paramref name="a"/> by another integer
    <paramref name="b"/> and returns the result.
    </summary>
    <returns>
    The quotient of two integers.
    </returns>
    <example>
    <code>
    int c = Math.Divide(4, 5);
    if (c > 1)
    {
      Console.WriteLine(c);
    }
    </code>
    </example>
    <exception cref="System.DivideByZeroException">
    Thrown when <paramref name="b"/> is equal to 0.
    </exception>
    See <see cref="Math.Divide(double, double)"/> to divide doubles.
    <seealso cref="Math.Add(int, int)"/>
    <seealso cref="Math.Subtract(int, int)"/>
    <seealso cref="Math.Multiply(int, int)"/>
    <param name="a">An integer dividend.</param>
    <param name="b">An integer divisor.</param>
  </DivideInt>
  <DivideDouble>
```

```
      <summary>
      Divides a double <paramref name="a"/> by another
      double <paramref name="b"/> and returns the result.
      </summary>
      <returns>
      The quotient of two doubles.
      </returns>
      <example>
      <code>
      double c = Math.Divide(4.5, 5.4);
      if (c > 1.0)
      {
        Console.WriteLine(c);
      }
      </code>
      </example>
      <exception cref="System.DivideByZeroException">Thrown when <paramref
name="b"/> is equal to 0.</exception>
      See <see cref="Math.Divide(int, int)"/> to divide integers.
      <seealso cref="Math.Add(double, double)"/>
      <seealso cref="Math.Subtract(double, double)"/>
      <seealso cref="Math.Multiply(double, double)"/>
      <param name="a">A double precision dividend.</param>
      <param name="b">A double precision divisor.</param>
    </DivideDouble>
  </members>
</docs>
```

在上面的 XML 中，每个成员的文档注释将直接显示在按其作用命名的标记中。 可选择自己的策略。 该代码使用 `<include>` 标记来引用 XML 文件中的相应元素：

C#

```
namespace IncludeTag
{

    /*
        The main Math class
        Contains all methods for performing basic math functions
    */
    /// <include file='include.xml'
path='docs/members[@name="math"]/Math/*'/>
    public class Math
    {
        // Adds two integers and returns the result
        /// <include file='include.xml'
path='docs/members[@name="math"]/AddInt/*'/>
        public static int Add(int a, int b)
        {
            // If any parameter is equal to the max value of an integer
            // and the other is greater than zero
            if ((a == int.MaxValue && b > 0) || (b == int.MaxValue && a >
0))
```

```
                throw new System.OverflowException();

            return a + b;
        }

        // Adds two doubles and returns the result
        /// <include file='include.xml'
path='docs/members[@name="math"]/AddDouble/*'/>
        public static double Add(double a, double b)
        {
            // If any parameter is equal to the max value of an integer
            // and the other is greater than zero
            if ((a == double.MaxValue && b > 0) || (b == double.MaxValue &&
a > 0))
                throw new System.OverflowException();

            return a + b;
        }

        // Subtracts an integer from another and returns the result
        /// <include file='include.xml'
path='docs/members[@name="math"]/SubtractInt/*'/>
        public static int Subtract(int a, int b)
        {
            return a - b;
        }

        // Subtracts a double from another and returns the result
        /// <include file='include.xml'
path='docs/members[@name="math"]/SubtractDouble/*'/>
        public static double Subtract(double a, double b)
        {
            return a - b;
        }

        // Multiplies two integers and returns the result
        /// <include file='include.xml'
path='docs/members[@name="math"]/MultiplyInt/*'/>
        public static int Multiply(int a, int b)
        {
            return a * b;
        }

        // Multiplies two doubles and returns the result
        /// <include file='include.xml'
path='docs/members[@name="math"]/MultiplyDouble/*'/>
        public static double Multiply(double a, double b)
        {
            return a * b;
        }

        // Divides an integer by another and returns the result
        /// <include file='include.xml'
path='docs/members[@name="math"]/DivideInt/*'/>
        public static int Divide(int a, int b)
```

```
        {
            return a / b;
        }

        // Divides a double by another and returns the result
        /// <include file='include.xml'
path='docs/members[@name="math"]/DivideDouble/*'/>
        public static double Divide(double a, double b)
        {
            return a / b;
        }
    }
}
```

- `file` 属性表示包含文档的 XML 文件的名称。
- `path` 属性表示一个 XPath 查询，该查询的对象为指定的 `file` 中的 `tag name`。
- `name` 属性表示位于注释前的标记中的名称说明符。
- 可用于替换 `name` 的 `id` 属性表示位于注释前的标记的 ID。

C# 编译器错误

项目 • 2023/04/07

一些 C# 编译器错误有对应的主题，其中介绍了导致错误生成的原因，在某些情况下还介绍了错误修复方法。 请执行下列操作之一，查看是否有特定错误消息的相关帮助资料。

- 如果使用的是 Visual Studio，请在“输出窗口”中选择错误号（例如 CS0029），然后按 F1 键。
- 在目录的“按标题筛选”框中键入错误号。

如果通过这些步骤无法找到有关错误的信息，请转到此页末尾，然后发送包含错误号或文本的反馈。

若要了解如何在 C# 中配置错误和警告选项，请参阅 C# 编译器选项或 Visual Studio“项目设计器”->“生成”页 (C#)。

> ⓘ **备注**
>
> 以下说明中的某些 Visual Studio 用户界面元素在计算机上出现的名称或位置可能会不同。 这些元素取决于你所使用的 Visual Studio 版本和你所使用的设置。 有关详细信息，请参阅**个性化设置** IDE。

另请参阅

- C# 编译器选项
- “项目设计器”->“生成”页 (C#)
- WarningLevel（C# 编译器选项）
- NoWarn（C# 编译器选项）

解决与语言功能和版本相关的警告

项目 • 2024/11/06

本文介绍以下编译器警告：

- **CS8022、CS8023、CS8024、CS8025、CS8026、CS8059、CS8107、CS8302、CS8320、CS8370、CS8400、CS8773、CS8936、CS9058**：*功能不可用。使用较新的语言版本。*
- **CS8058**：*功能是试验性的。*
- **CS8192**：*提供的语言版本不受支持或无效*
- **CS8303**：*指定的语言版本不能包含前导零*
- **CS8304**：*编译器版本低于语言版本*
- **CS1738**：*在所有固定参数均已指定后，必须显示命名参数规范。*
- **CS8306**：*已推断元组元素名称。*
- **CS8314**：*类型的模式无法处理类型的表达式*
- **CS8371**：*语言版本中不支持自动属性的字段针对特性*
- **CS8401**：*若要对内插逐字字符串使用 `@$` 而不是 `$@`，请使用较新的语言版本。*
- **CS8511**：*类型的模式无法处理类型的表达式。*
- **CS8627**：*可为 null 的类型参数必须已知为值类型或不可为 null 引用类型*
- **CS8630**：*无效的可为 null 的选项。使用较新的语言版本*
- **CS8652**：*修饰符对此项目无效。*
- **CS8704**：*类型不实现接口成员。它无法隐式实现非公共成员。*
- **CS8706**：*类型无法实现接口成员，因为此版本中不提供某个功能。*
- **CS8904**：*方差无效：类型参数必须有效。*
- **CS8912**：*不支持从具有密封的“Object.ToString”的记录继承。*
- CS8919：无法在类型中实现指定的接口成员，因为目标运行时不支持接口中的静态抽象成员
- CS8929：方法无法在类型中实现接口成员，因为目标运行时不支持接口中的静态抽象成员。
- **CS8957**：*条件表达式在语言版本中无效，因为在类型之间找不到通用类型。*
- **CS8967**：*C# 不支持非逐字内插字符串内的换行符*
- **CS9014**：*错误：使用可能未分配的属性。升级到自动默认属性。*
- **CS9015**：*错误：使用可能未分配的字段。升级到自动默认字段。*
- **CS9016**：*警告：使用可能未分配的属性。升级到自动默认属性。*
- **CS9017**：*警告：使用可能未分配的字段。升级到自动默认字段。*
- **CS9064**：*目标运行时不支持 ref 字段。*
- **CS9103**：*模块中的定义具有无法识别的 RefSafetyRulesAttribute 版本，应为“11”。*
- **CS9171**：*目标运行时不支持内联数组类型。*
- **CS9194**：*自变量不可与 `ref` 关键字一起传递。要将 `ref` 实参传递给 `in` 形参，请升级到语言版本 12 或更高版本。*

- **CS9202**：*功能在 C# 12.0 中不可用。请使用较新的语言版本*
- **CS9211**：*“Experimental”属性的 diagnosticId 自变量必须是有效的标识符。*
- **CS9240**：*目标运行时不支持 by-ref-like 泛型。*
- **CS9260**：*功能在 C# 13.0 中不可用。请使用较新的语言版本**

此外，以下错误和警告与最新版本中的结构初始化变更相关：

- CS0171、**CS8881：在控件返回调用方之前，自动实现的属性“name”的支持字段必须完全赋值。**
- CS0188、CS8885：在给“this”对象的所有字段赋值之前，无法使用该对象
- CS0843、CS8880：在控件返回调用方之前，自动实现的属性“name”的支持字段必须完全赋值
- CS8305：*功能仅用于评估目的，在将来的更新中可能会被更改或删除。*
- CS9204：*类型仅用于评估目的，在将来的更新中可能会被更改或删除。取消此诊断以继续。*

导致上述所有错误和警告的原因是安装的编译器支持的 C# 版本比项目选择的版本更高。 C# 编译器可以符合以前的任何版本。 可以针对早期版本的 C# 验证语法，或者因为项目必须支持更低版本的库或运行时。

有两种可能的原因和三种方法来解决这些错误和警告。

更新目标框架

编译器根据以下规则确定默认值：

展开表

目标	版本	C# 语言版本的默认值
.NET	9.x	C# 13
.NET	8.x	C# 12
.NET	7.x	C# 11
.NET	6.x	C# 10
.NET	5.x	C# 9.0
.NET Core	3.x	C# 8.0
.NET Core	2.x	C# 7.3
.NET Standard	2.1	C# 8.0

目标	版本	C# 语言版本的默认值
.NET Standard	2.0	C# 7.3
.NET Standard	1.x	C# 7.3
.NET Framework	全部	C# 7.3

如果所选框架与所需的语言版本不匹配，可以升级目标框架。

选择匹配的语言版本

你可能在项目文件中选择了更低版本的目标框架。 如果从项目文件中删除 `LangVersion` 元素，编译器将使用上一部分中列出的默认值。 下表显示当前所有 C# 语言版本。 还可以指定特定语言版本来启用新功能。

展开表

值	含义
`preview`	编译器接受最新预览版中的所有有效语言语法。
`latest`	编译器接受最新发布的编译器版本（包括次要版本）中的语法。
`latestMajor` 或 `default`	编译器接受最新发布的编译器主要版本中的语法。
`13.0`	编译器只接受 C# 13 或更低版本中所含的语法。
`12.0`	编译器只接受 C# 12 或更低版本中所含的语法。
`11.0`	编译器只接受 C# 11 或更低版本中包含的语法。
`10.0`	编译器只接受 C# 10 或更低版本中所含的语法。
`9.0`	编译器只接受 C# 9 或更低版本中所含的语法。
`8.0`	编译器只接受 C# 8.0 或更低版本中所含的语法。
`7.3`	编译器只接受 C# 7.3 或更低版本中所含的语法。
`7.2`	编译器只接受 C# 7.2 或更低版本中所含的语法。
`7.1`	编译器只接受 C# 7.1 或更低版本中所含的语法。
`7`	编译器只接受 C# 7.0 或更低版本中所含的语法。
`6`	编译器只接受 C# 6.0 或更低版本中所含的语法。

值	含义
`5`	编译器只接受 C# 5.0 或更低版本中所含的语法。
`4`	编译器只接受 C# 4.0 或更低版本中所含的语法。
`3`	编译器只接受 C# 3.0 或更低版本中所含的语法。
`ISO-2` 或 `2`	编译器只接受 ISO/IEC 23270:2006 C# (2.0) 中所含的语法。
`ISO-1` 或 `1`	编译器只接受 ISO/IEC 23270:2003 C# (1.0/1.2) 中所含的语法。

可以在配置语言版本一文中的“语言参考”部分详细了解每个框架版本支持的语言版本。

避免使用更新的功能

如果必须支持更低版本的库或运行时，则可能需要避免使用新功能。

启用实验性功能

可以禁用实验性功能的诊断以使用实验性功能。

> ⚠ **警告**
>
> 实验性功能可能会随时更改。 API 可能会更改，或者可能会在未来的更新中被删除。 包括实验性功能是库作者获取有关未来开发的想法和概念反馈的一种方式。 使用标记为实验性的任何功能时，请格外小心。

还可以使用 System.Diagnostics.CodeAnalysis.ExperimentalAttribute 声明自己的试验性功能。 如果用于试验性功能的标识符不是有效的标识符，则编译器将发出 *CS9211*。

结构初始化的中断性变更

所有这些错误和警告都有助于确保 `struct` 类型在访问其字段之前已正确初始化。 在早期版本的 C# 中，必须在任何构造函数中显式赋值结构中的所有字段。 无参数构造函数将所有字段初始化为其默认值。 在更高版本中，所有构造函数将初始化所有字段。 显式设置字段，在字段初始值设定项中设置，或设置为其默认值。

- CS0171、CS8881：在控件返回调用方之前，自动实现的属性“name”的支持字段必须完全赋值。

- CS0188、CS8885：在给“this”对象的所有字段赋值之前，无法使用该对象
- CS0843、CS8880：在控件返回调用方之前，自动实现的属性“name”的支持字段必须完全赋值

可以通过将语言版本升级到 C# 11 来解决此错误，每个 `struct` 构造函数都会初始化所有字段。 如果无法执行此操作，则必须显式调用默认构造函数，如以下示例所示：

C#

```
struct S
{
    public int AIProp { get; set; }
    public S(int i){} //CS0843
    // Try the following lines instead.
    // public S(int i) : this()
    // {
    //     AIProp = i;
    // }
}

class Test
{
    static int Main()
    {
        return 1;
    }
}
```

解决与程序集引用相关的错误和警告

项目 • 2024/03/04

- CS0012: *在未被引用的程序集中定义了类型“type”。必须添加对程序集“assembly”的引用。*
- CS0234: *命名空间中不存在类型或命名空间名（是否缺少程序集引用？）*
- CS0246: *未能找到类型或命名空间名（是否缺少 using 指令或程序集引用？）*
- CS0400: *未能在全局命名空间中找到类型或命名空间名（是否缺少程序集引用？）*
- CS0735: *指定为 TypeForwardedToAttribute 属性的参数的类型无效。*
- CS1068: *无法在全局命名空间中找到类型名称。此类型已转发到另一个程序集。请考虑添加对该程序集的引用。*
- CS1069: *无法在命名空间中找到类型名称。此类型已转发到另一个程序集。请考虑添加对该程序集的引用。*
- CS1070: *未能找到类型名称。此类型已转发到另一个程序集。请考虑添加对该程序集的引用。*
- CS1704: *已导入具有相同的简单名称的程序集。尝试删除某个引用，或为其签名，以启用并行。*
- CS1714: *此类型的基类或接口未能解析或无效。*
- CS1760: *多个程序集引用相同的元数据，但只有一个是链接引用（使用 /link 选项指定）；请考虑移除其中一个引用。*
- CS7008: *程序集名保留名称，不能在交互会话中用作引用。*
- CS7068: *对类型的引用声明在此程序集中定义了该类型，但源代码或任何添加的模块中并未定义该类型。*
- CS7069: *对类型的引用声明在另一个程序集中定义了该类型，但找不到它。*
- CS7071: *程序集引用无效且无法解析。*
- CS7079: *类型在尚未添加的模块中定义。必须添加该模块。*
- CS8090: *引用的程序集中有错误。*
- CS8203: *程序集名称无效。*

此外，本文还介绍了以下警告：

- CS1683: *对类型“Type Name”的引用声明在此程序集中定义了该类型，但在源代码或任何添加的模块中并未定义该类型。*

缺少引用

以下错误和警告指示缺少程序集引用：

- CS0012: *在未被引用的程序集中定义了类型“type”。必须添加对程序集“assembly”的引用。*

- CS0234: *命名空间中不存在类型或命名空间名（是否缺少程序集引用？）*
- CS0246: *未能找到类型或命名空间名（是否缺少 using 指令或程序集引用？）*

这些编译器错误指示代码中存在以下问题之一：

- 项目未引用所需的程序集。 若要修复此错误，请添加对所需程序集的引用。
- 拼错了某类型的名称。 检查类型的名称。
- 你使用了一个变量名称，其中 System.Type 的名称是预期的，例如 typeof 运算符 或 is 运算符。
- 你在类型不在全局命名空间中时，使用了全局范围运算符(::)。

类型转发

- CS1068: *无法在全局命名空间中找到类型名称。此类型已转发到另一个程序集。请考虑添加对该程序集的引用。*
- CS1069: *无法在命名空间中找到类型名称。此类型已转发到另一个程序集。请考虑添加对该程序集的引用。*
- CS1070: *未能找到类型名称。此类型已转发到另一个程序集。请考虑添加对该程序集的引用。*

这些错误表示引用转发到不同程序集的类型时出错。 若要解决上述任何错误，请添加对错误消息中指示的程序集的引用。

重复引用

以下错误指示重复的程序集引用：

- CS1704: *已导入具有相同的简单名称的程序集。尝试删除某个引用，或为其签名，以启用并行。*
- CS1760: *多个程序集引用相同的元数据，但只有一个是链接引用（使用 /link 选项指定）；请考虑移除其中一个引用。*

若要修复这些错误，必须移除其中一个引用，或解决重复问题。 导致重复的原因包括：

- 多个未签名的程序集具有相同的名称。
- 项目引用同一程序集的多个版本。

程序集引用无效

以下错误指示程序集引用无效：

- CS7008: *程序集名保留名称，不能在交互会话中用作引用。*

- CS7069: *对类型的引用声明在另一个程序集中定义了该类型，但找不到它。*
- CS7071: *程序集引用无效且无法解析。*
- CS7079: *类型在尚未添加的模块中定义。必须添加该模块。*
- CS8090: *引用的程序集中有错误。*
- CS8203: *程序集名称无效。*

以下警告还指示无效的引用程序集：

- CS1683: *对类型“Type Name”的引用声明在此程序集中定义了该类型，但在源代码或任何添加的模块中并未定义该类型*

检查程序集名称拼写是否正确。 引用的程序集文件可能无效。

在 GitHub 上与我们协作

可以在 GitHub 上找到此内容的源，还可以在其中创建和查看问题和拉取请求。 有关详细信息，请参阅参与者指南。

.NET 反馈

.NET 是一个开放源代码项目。 选择一个链接以提供反馈：

提出文档问题

提供产品反馈

解决构造函数声明中的错误和警告

项目 • 2024/12/28

本文介绍以下编译器错误：

- CS0514：*静态构造函数不能有显式的“this”或“base”构造函数调用。*
- CS0515：不允许在静态构造函数中使用 *访问修饰符。*
- CS0516：*构造函数“constructor”无法调用其自身。*
- CS0517：*“class”没有基类，无法调用基构造函数。*
- CS0522：*结构无法调用基类构造函数。*
- CS0526：*接口不能包含构造函数。*
- CS0568：*结构不能包含显式无参数构造函数。*
- CS0710：*静态类不能有实例构造函数。*
- CS0768：*构造函数无法通过另一个构造函数自行调用。*
- CS8054：*枚举不能包含显式无参数构造器。*
- CS8091：*不能是外部的，也不能具有构造函数初始值设定项。*
- CS8861：*意外参数列表。*
- CS8862：*在具有参数列表的类型中声明的构造函数必须具有“this”构造函数初始值设定项。*
- CS8358：*无法使用属性构造函数，因为它具有“in”参数。*
- CS8867：*基类型“{0}”中找不到可访问的复制构造函数。*
- CS8868：*记录中的复制构造函数必须调用基的复制构造函数;如果记录继承自对象，则必须调用无参数对象构造函数。*
- CS8878：*复制构造函数“{0}”必须公开或受保护，因为记录未密封。*
- CS8910：*主构造函数与合成复制构造函数冲突。*
- CS8958：*无参数结构构造函数必须为“public”。*
- CS8982：*在“结构体”中带有参数列表声明的构造函数必须有一个“this”初始化器，用于调用主构造函数或显式声明的其他构造函数。*
- CS8983：*具有字段初始值设定项的“结构”必须包含显式声明的构造函数。*
- CS9105：*不能在此上下文中使用主构造函数参数。*
- CS9106：*在此上下文中，类型和参数之间的标识符不明确。*类也是如此。*
- CS9108：*无法使用在匿名方法、lambda 表达式、查询表达式或本地函数中具有类似 ref 类型的参数。*
- CS9109：*不能在实例成员内使用 `ref`、`out` 或 `in` 主构造函数参数。*
- CS9110：*无法在实例成员中使用具有类似于 ref 类型的主构造函数参数。*
- CS9111：*结构实例成员内的匿名方法、lambda 表达式、查询表达式和本地函数无法访问主构造函数参数。*
- CS9112：*匿名方法、lambda 表达式、查询表达式和结构内的本地函数无法访问实例成员内也使用的主构造函数参数。*

- CS9114:*不能为只读类型的主构造函数参数赋值（在该类型的仅限 init 的资源库中或变量初始值设定项除外）。*
- CS9115：*可写引用无法返回只读类型的主要构造函数参数。*
- CS9116：*无法将只读类型的主构造函数参数用作 ref 或 out 值（在该类型的仅限 init 的资源库中或变量初始值设定项除外）。*
- CS9117:*不能修改只读类型的主构造函数参数的成员（在该类型的仅限 init 的资源库中或变量初始值设定项除外）。*
- CS9118：*只读类型的主构造函数参数的成员不能通过可写引用返回。*
- CS9119：*不能将只读类型的主构造函数参数的成员用作 `ref` 或 `out` 值（在该类型的仅限 init 的资源库中或变量初始值设定项除外）。*
- CS9120：*无法按引用返回主构造函数参数。*
- CS9121：*结构主构造函数参数的类型会导致结构布局中的循环。*
- CS9122：*意外的参数列表。*
- CS9136：*无法在实例成员内使用类型的主构造函数参数。*

此外，本文还介绍了以下警告：

- CS0824：*构造函数“name”标记为外部对象。*
- CS9107：*参数被捕获到封闭类型的状态，其值也传递给基构造函数。也可以由基类捕获该值。*
- CS9113：*参数未读。*
- CS9124：*参数被捕获到封闭类型的状态中，其值还会用于初始化字段、属性或事件。*
- CS9179：*主构造函数参数被基成员隐藏*

静态构造函数

- CS0514：*静态构造函数不能具有显式的“this”或“base”构造函数调用。*
- CS0515：*静态构造函数中不允许出现访问修饰符。*

最多可以编写一个类型的静态构造函数。 静态构造函数的声明必须遵循以下规则：

- 静态构造函数具有 `static` 修饰符，但没有其他修饰符，例如 `public`、`protected`、 `private` 或 `internal`。
- 静态构造函数必须是无参数构造函数。
- 静态构造函数不得调用 `base()` 或 `this()`。 如果基类包含静态构造函数，运行时会自动调用它。

构造函数声明

- CS0526：*接口不能包含构造函数。*

- CS0710：*静态类不能有实例构造函数。*
- CS8054：*枚举不能包含显式的无参数构造函数。*
- CS8358：*无法使用属性构造函数，因为它具有"in"参数。*
- CS8091：*构造函数不能是外部构造函数，并且具有构造函数初始值设定项。*

构造函数仅允许在 `class` 和 `struct` 类型中使用，包括 `record class` 和 `record struct` 类型。 不能在 `enum` 或 `interface` 类型中定义它们。 此外，属性 类类型无法声明 `in` 参数。 而是按值传递参数。

可以声明 `extern` 构造函数，但不能使用 `base()` 或 `this()` 构造函数调用从声明 `extern` 的构造函数调用另一个构造函数。

此外，可以为构造函数声明生成以下警告：

- CS0824：*构造函数标记为外部对象。*

当构造函数标记为 `extern`时，编译器无法保证构造函数存在。 因此，编译器将生成此警告。

结构类型中的构造函数

- CS0568：*结构不能包含显式无参数构造函数。*
- CS8958：*无参数结构构造函数必须为"public"。*
- CS8982：*具有参数列表的"结构"中声明的构造函数必须具有调用主构造函数或显式声明的构造函数的"this"初始值设定项。*
- CS8983：*具有字段初始值设定项的"结构"必须包含显式声明的构造函数。*

C# 中的最近功能消除了对 `struct` 类型的早期限制。 在旧版 C# 中声明无参数实例构造函数时，将生成 CS0568。 在 C# 的较新版本中，可以声明一个无参数的显式实例构造函数。 该显式无参数构造函数必须是`public`。 如果 `struct` 声明了任何字段初始化表达式，你还必须声明显式实例构造函数。 此构造函数可以是具有空正文的无参数构造函数。

当 `struct` 类型声明主构造函数（包括 `record struct` 类型）时，除无参数构造函数以外的所有其他实例构造函数都必须使用 `this()`调用主构造函数或其他显式声明的构造函数。

使用 `base` 和 `this` 进行构造函数调用

- CS0516：*构造函数无法调用其自身。*
- CS0517：*类没有基类，并且无法调用基构造函数。*
- CS0522：*结构无法调用基类构造函数。*

- CS0768：*构造函数无法通过另一个构造函数自行调用。*

可以使用 `base()` 和 `this()` 完成一个构造函数对另一个相同类型或基类型的构造函数的调用。 调用构造函数可以最大程度地减少重复的构造函数逻辑。 在使用 `this()` 或 `base()`调用另一个构造函数时，必须遵循以下规则：

- 构造函数不能通过另一个构造函数直接或间接调用自身。 例如，以下代码是非法的：

```
C#
```

```
public class C
{
  public C() : this() // Error!
  {
  }
}

public class C2
{
  public class C2() : this(10) {}

  public class C2(int capacity) : this()
  {
      _capacity = capacity;
  }

  private int _capacity;
}
``
```

- 结构类型无法调用 `base()`。 System.Object也不能。

记录和复制构造函数

- CS8867：*基类型中找不到可访问的复制构造函数。*
- CS8868：*记录中的复制构造函数必须调用基的复制构造函数;如果记录继承自对象，则必须调用无参数对象构造函数。*
- CS8878：*复制构造函数必须是公共的或受保护的，因为该记录未密封。*
- CS8910：*主构造函数与合成复制构造函数冲突。*

将 `record` 修饰符添加到 `struct` 或 `class` 类型会创建记录。 记录包括编译器合成的复制构造函数。 可以自行编写显式复制构造函数，但它必须遵循以下规则：

- 复制构造函数必须是 `public` 或 `protected`，除非类型为 sealed。

- 复制构造函数必须调用 `base()` 复制构造函数，除非基类为 System.Object。
- 此外，基类型必须具有复制构造函数。 `record` 类型始终具有复制构造函数。

主构造函数声明

当主构造函数违反类和结构的主要构造函数上的一个或多个规则时，编译器将发出以下错误：

- CS8861：*意外的参数列表。*
- CS8862：*在具有参数列表的类型中声明的构造函数必须具有"this"构造函数初始值设定项。*
- CS9105：*不能在此上下文中使用主构造函数参数。*
- CS9106：*标识符在此上下文中的类型和参数之间不明确。*
- CS9108：*无法使用在匿名方法、lambda 表达式、查询表达式或本地函数中具有类似 ref 类型的参数。*
- CS9109：*不能在实例成员内使用 `ref`、`out` 或 `in` 主构造函数参数。*
- CS9110：*无法在实例成员中使用具有 ref 类似类型的主构造函数参数。*
- CS9111：*结构实例成员内的匿名方法、lambda 表达式、查询表达式和本地函数无法访问主构造函数参数。*
- CS9112：*匿名方法、lambda 表达式、查询表达式和结构内的本地函数无法访问实例成员内也使用的主构造函数参数。*
- CS9114：*不能为只读类型的主构造函数参数赋值（在该类型的仅限 init 的资源库中或变量初始值设定项除外）。*
- CS9115：*可写引用无法返回只读类型的主要构造函数参数。*
- CS9116：*无法将只读类型的主构造函数参数用作 `ref` 或 `out` 值（在该类型的仅限 init 的资源库中或变量初始值设定项除外）。*
- CS9117：*不能修改只读类型的主构造函数参数的成员（在该类型的仅限 init 的资源库中或变量初始值设定项除外）。*
- CS9118：*只读类型的主构造函数参数的成员不能通过可写引用返回。*
- CS9119：*不能将只读类型的主构造函数参数的成员用作 `ref` 或 `out` 值（在该类型的仅限 init 的资源库中或变量初始值设定项除外）。*
- CS9120：*无法按引用返回主构造函数参数。*
- CS9121：*类型的结构主构造函数参数导致结构布局中出现循环。*
- CS9122：*意外的参数列表。*
- CS9124：*参数被捕获到封闭类型的状态中，其值也被用于初始化字段、属性或事件。*
- CS9136：*无法在实例成员内使用类型的主构造函数参数。*

主构造函数参数在该类型的主体范围内。 编译器可以合成一个字段，该字段存储用于成员或字段初始值设定项的参数。 由于主构造函数参数可以复制到字段，因此适用以下限制：

- 主要构造函数可以在 `struct` 和 `class` 类型上声明，但不能在 `interface` 类型上声明。
- 主构造函数参数不能在 `base()` 构造函数调用中使用，除非作为主构造函数的一部分。
- 无法在 lambda 表达式、查询表达式或本地函数中访问 `ref struct` 类型的主要构造函数参数。
- 如果类型不是 `ref struct`，则实例成员中无法访问 `ref struct` 参数。
- 在 `ref struct` 类型中，不能在任何实例方法或属性访问器中使用具有 `in`、`ref`或 `out` 修饰符的主要构造函数参数。

结构类型对主要构造函数参数具有以下额外限制：

- 无法在 lambda 表达式、查询表达式或本地函数中捕获主构造函数参数。
- 主构造函数参数不能由引用返回（`ref` 返回或 `readonly ref` 返回）。

仅限只读的结构类型对主要构造函数参数具有以下额外限制：

- 主构造函数参数及其成员无法在 `readonly` 结构中重新分配。
- 不能在 `readonly` 结构中对主构造函数参数及其成员进行 `ref` 返回。
- 主构造函数的参数及其成员不能作为任何方法的 `ref` 或 `out` 参数。

在所有这些情况下，对主要构造函数参数的限制与这些类型中的数据字段的限制一致。 限制是因为主要构造函数参数可能转换为类型中的合成字段。 因此，主构造函数参数必须遵循应用于该合成字段的规则。

派生的主构造函数通过向基构造函数提供参数来调用基本主构造函数。 必须使用派生构造函数声明中的参数名称。

这些警告提供有关捕获或隐藏的主构造函数参数的指导。

- **CS9107**：*参数被捕获到封闭类型的状态，并且其值也传递给基构造函数。 也可以由基类捕获该值。* 此警告指示代码可能分配了主构造函数参数的两个副本。 由于参数传递给基类，因此基类可能会使用它。 由于派生类访问它，因此它可能具有同一参数的第二个副本。 这种额外的存储可能不是预期的。
- **CS9113**：*参数未被读取。* 此警告表明你的类从未引用主构造函数，甚至没有将其传递给基本主构造函数。 可能用不着。
- **CS9124**：*参数捕获到封闭类型的状态，并且其值还用于初始化字段、属性或事件。* 此警告指示嵌套类型的构造函数参数也由封闭类型捕获。 该参数可能存储两次。
- **CS9179**：*主构造函数参数被基成员隐藏*

形参和实参不匹配

项目 • 2023/12/22

如果没有为形参提供自实参，或者实参对该形参无效，编译器将生成以下错误：

- CS0182：特性实参必须是特性形参类型的常量表达式、`typeof` 表达式或数组创建表达式
- CS0591：特性的实参值无效
- CS0599：命名特性实参“argument”的值无效
- CS0617：不是有效的命名特性实参。命名特性实参必须是非只读、非静态、非常数的字段，或者必须是公共的和非静态的读写属性。
- CS0633：特性的实参必须是有效的标识符
- CS0643：命名特性实参重复
- CS0655：不是有效的特性形参类型，因此不是有效的命名特性实参
- CS0839：缺少实参。
- CS1016：应为命名特性实参
- CS1739：相应最佳重载没有相关命名的形参
- CS1740：不能多次指定命名形参
- CS1742：数组访问可能没有命名实参说明符
- CS1744：命名实参指定的形参已被赋予位置参数
- CS1746：委托没有名为“name”的形参
- CS7036：未提供与所需形参对应的实参
- CS7067：特性构造函数形参是可选的，但是未指定默认形参值。
- CS8324：已在动态调用中指定所有固定实参之后，必须出现命名实参规范。
- CS8905：不能使用命名实参调用函数指针。
- CS8943：null 不是有效的形参名称。若要获取对实例方法接收器的访问权限，请使用空字符串作为形参名称。
- CS8944：方法不是实例方法，接收器不能是内插字符串处理程序实参。
- CS8945：不是有效的形参名称。
- CS8948：`InterpolatedStringHandlerArgumentAttribute` 实参不能引用使用该特性的形参。
- CS8949：应用于形参的 `InterpolatedStringHandlerArgumentAttribute` 格式不正确，无法解释。手动构造它的实例。
- CS8950：形参是形参上的内插字符串处理程序转换的实参，但在内插字符串表达式后面指定了相应的实参。请对形参进行重新排序。
- CS8951：未显式提供形参，但它用作形参上的内插字符串处理程序转换的实参。
- CS8964：`CallerArgumentExpressionAttribute` 只能应用于具有默认值的形参
- CS8965：应用于形参的 `CallerArgumentExpressionAttribute` 无效，因为它是自引用的。

- CS8966：`CallerArgumentExpressionAttribute` 将不起作用，因为它应用到的成员在不允许使用可选实参的上下文中使用

缺少实参

当编译器无法匹配所有成员形参的实参时，将发出以下常规错误：

- **CS0839**：缺少实参。
- **CS7036**：未提供与所需形参对应的实参

这些是常规错误：编译器与对方法的所需形参进行的方法调用中给出的实参不匹配。 请检查以下原因：

- 确保包含所有必要的实参。
- 确保实参的顺序正确。
- 确保所有形参的类型都正确。
- 确保重载解析规则选择了所需的方法。

如果编写了重载的本地函数，可能还会看到 CS7036。 无法重载本地函数。 编译器仅识别具有该名称的第一个本地函数。 检查是否要调用其他本地函数。

这些错误通常与其他有助于诊断正确原因的诊断一起出现。

特性的实参

如果特性构造函数的实参不正确，编译器会发出以下错误：

- **CS0182**：特性实参必须是特性形参类型的常量表达式、`typeof` 表达式或数组创建表达式
- **CS0591**：特性的实参值无效
- **CS0599**：命名特性实参“argument”的值无效
- **CS0617**：不是有效的命名特性实参。命名特性实参必须是非只读、非静态、非常数的字段，或者必须是公共的和非静态的读写属性。
- **CS0633**：特性的实参必须是有效的标识符
- **CS0643**：命名特性实参重复
- **CS0655**：不是有效的特性形参类型，因此不是有效的命名特性实参

如果在特性定义上使用 System.AttributeUsageAttribute，请确保允许的值不相互排斥。 检查特性的实参类型和顺序是否正确。 确保字符串实参的文本有效。 对于许多特性，实参必须是有效的 C# 标识符。 特性构造函数的实参必须是编译时常量。 因此，它们仅限于支持文本常量的类型。 此外，禁止将以下允许文本常量的类型作为特性形参：

- sbyte

- ushort
- uint
- ulong
- decimal

不能指定具有相同形参名称的重复命名实参。 只能在初始化特性时设置可访问的属性。 无法设置专用属性。

命名和可选形参和实参

编译器因命名实参和可选实参使用错误而发出以下错误：

- **CS1016**：应为命名特性实参
- **CS1739**：相应最佳重载没有相关命名的形参
- **CS1740**：不能多次指定命名形参
- **CS1742**：数组访问可能没有命名实参说明符
- **CS1744**：命名实参指定的形参已被赋予位置参数
- **CS1746**：委托没有名为“name”的形参
- **CS7067**：特性构造函数形参是可选的，但是未指定默认形参值。
- **CS8324**：已在动态调用中指定所有固定实参之后，必须出现命名实参规范。
- **CS8905**：不能使用命名实参调用函数指针。

请检查是否是以下原因导致了这些错误：

- 命名实参的形参名称不正确。
- 所选重载没有与命名实参匹配的形参。
- 形参名称在多个实参上重复。
- 位置（未命名）实参显示在命名实参的后面。
- 数组索引形参不允许命名实参。

内插字符串处理程序

如果不恰当地指定了内插字符串处理程序，编译器会发出以下错误。

- **CS8943**：null 不是有效的形参名称。若要获取对实例方法接收器的访问权限，请使用空字符串作为形参名称。
- **CS8944**：不是实例方法，接收器不能是内插字符串处理程序参数。
- **CS8945**：不是有效的形参名称。
- **CS8948**：`InterpolatedStringHandlerArgumentAttribute` 实参不能引用使用该特性的形参。
- **CS8949**：应用于形参的 `InterpolatedStringHandlerArgumentAttribute` 格式不正确，无法解释。请手动构造实例。

- **CS8950**：形参是形参上的内插字符串处理程序转换的实参，但在内插字符串表达式后面指定了相应的实参。
- **CS8951**：未显式提供形参，但它用作形参上的内插字符串处理程序转换的实参。

内插字符串处理程序是基于模式的构造。 正确获取模式非常重要。 请参阅功能规范，或者按照有关构建内插字符串处理程序的教程进行操作。

调用方调试信息

如果错误地使用了 System.Runtime.CompilerServices.CallerArgumentExpressionAttribute，编译器会发出以下错误：

- **CS8964**：`CallerArgumentExpressionAttribute` 只能应用于具有默认值的形参

此外，如果错误地使用 `CallerArgumentExpressionAttribute`，编译器会发出以下警告：

- **CS8965**：应用于形参的 `CallerArgumentExpressionAttribute` 无效，因为它是自引用的。
- **CS8966**：`CallerArgumentExpressionAttribute` 将不起作用，因为它应用到的成员在不允许使用可选实参的上下文中使用

使用 `CallerArgumentExpression` 特性批注的任何形参都必须具有默认值。

在 GitHub 上与我们协作

可以在 GitHub 上找到此内容的源，还可以在其中创建和查看问题和拉取请求。 有关详细信息，请参阅参与者指南。

.NET

.NET 反馈

.NET 是一个开放源代码项目。 选择一个链接以提供反馈：

提出文档问题

提供产品反馈

与引用参数、变量和返回关联的错误和警告

项目 • 2024/11/06

使用引用变量时，可能会生成以下错误：

- CS0192：*字段 `readonly` 不能用作 `ref` 或 `out` 值（构造函数除外）*
- CS0199：*字段 `static readonly` 不能用作 `ref` 或 `out` 值（静态构造函数除外）*
- CS0206：*非引用返回属性或索引器不能用作 `out` 或 `ref` 值*
- CS0631：*`ref` 和 `out` 在此上下文中无效*
- CS0767：*无法使用指定的类型参数继承接口，因为它会导致方法包含仅在 `ref` 和 `out` 上存在不同的重载*
- CS1510：*`ref` 或 `out` 值必须是可赋值变量*
- CS1605：*不能将变量用作 `ref` 或 `out` 值，因为它为只读*
- CS1623：*迭代器不能具有 `ref`、`in` 或 `out` 参数*
- CS1649：*字段 `readonly` 的成员不能用作 `ref` 或 `out` 值（构造函数除外）*
- CS1651：*静态只读字段的字段不能用作 `ref` 或 `out` 值（静态构造函数除外）*
- CS1655：*不能将类型字段用作 `ref` 或 `out` 值*
- CS1657：*不能将变量用作 `ref` 或 `out` 值*
- CS1741：*`ref` 或 `out` 参数不能有默认值*
- CS1939：*无法将范围变量作为 `out` 或 `ref` 参数传递*
- CS1988：*异步方法不能具有 `ref`、`in` 或 `out` 参数*
- CS7084：*无法将 Windows 运行时事件作为 `out` 或 `ref` 参数传递。*
- CS8166：*无法通过引用返回参数，因为它不是 `ref` 参数*
- CS8167：*无法通过引用返回参数成员，因为它不是 `ref` 或 `out` 参数*
- CS8168：*无法通过引用返回局部变量，因为它不是 ref 局部变量*
- CS8169：*无法通过引用返回局部变量成员，因为它不是 ref 局部变量*
- CS8196：对隐式类型的 out 变量的引用不允许出现在同一个参数列表中。
- CS8325：*"`await`" 不能在包含 `ref` 条件运算符的表达式中使用*
- CS8326：*这两个条件运算符的值必须都是 ref 值或者都不是 ref 值*
- CS8327：*表达式的类型必须正确，才能匹配备用 ref 值*
- CS8329：*不能将变量用作 `ref` 或 `out` 值，因为它是只读变量*
- CS8330：*变量的成员不能用作 `ref` 或 `out` 值，因为它是只读变量*
- CS8331：*无法赋值给变量或将其用作 `ref` 赋值的右侧，因为它是只读变量*
- CS8332：*无法赋值给变量的成员或将其用作 `ref` 赋值的右侧，因为它是只读变量*
- CS8337：*"`ref`" 扩展方法的第一个参数必须是值类型或受结构约束的泛型类型。*

- CS8338：*扩展方法的第一个 "`in`" 或 "`ref readonly`" 参数必须是具体的（非泛型）值类型。*
- CS8351：`ref` *条件运算符的分支不能引用具有不兼容声明范围的变量*
- CS8373：`ref` *赋值的左侧必须是 ref 变量。*
- CS8374：*无法引用赋值，源的转义范围比目标更窄。*
- CS8388：`out` *变量无法声明为 ref 局部变量*
- CS8977：*无法在具有 "UnmanagedCallersOnly" 特性的方法的签名中使用 "`ref`"、"`in`" 或 "`out`"。*
- CS9072：*析构变量不能声明为 ref 局部变量*
- CS9077：*无法通过 `ref` 参数按引用返回参数，它只能在 return 语句中返回*
- CS9078：*无法通过 `ref` 参数按引用返回参数的成员，它只能在 return 语句中返回*
- CS9079：*无法引用赋值，因为源只能通过 return 语句转义当前方法。*
- CS9096：*无法引用赋值，因为源的值转义范围比目标更广，因此允许通过转义范围比目标更窄的值源进行赋值。*
- CS9101：UnscopedRefAttribute 只能应用于结构或虚拟接口实例方法和属性，不能应用于构造函数或仅初始化成员。
- CS9102：UnscopedRefAttribute 不能应用于接口实现，因为实现的成员没有此属性。
- CS9104：*不能在异步方法或异步 Lambda 表达式中使用类型的 `using` 语句资源。*
- CS9190：`readonly` *修饰符必须在 `ref` 之后指定。*
- CS9199：`ref readonly` *参数不能具有 Out 特性。*

当引用变量被错误地使用时，将生成以下警告：

- CS9085：*这会引用赋值变量，但目标比源的转义范围更窄。*
- CS9086：`ref` *条件运算符的分支引用具有不兼容声明范围的变量*
- CS9087：*这会按引用返回一个参数，但它不是 `ref` 参数*
- CS9089：*这会按引用返回不是 `ref` 或 `out` 参数的参数成员*
- CS9091：*这会按引用返回局部变量，但它不是 ref 局部变量*
- CS9092：*这会按引用返回局部变量的成员，但它不是 ref 局部变量*
- CS9093：*这会引用赋值，但源只能通过 return 语句对当前方法进行转义。*
- CS9094：*这会通过 `ref` 参数按引用返回参数，但它只能在 return 语句中安全返回*
- CS9095：*这会通过 `ref` 参数按引用返回参数的成员，但它只能在 return 语句中安全返回*
- CS9097：*这会引用赋值，但源的值转义范围比目标更广，因此可以通过转义范围比源更窄的值目标进行赋值。*
- CS9191：*对应于 `in` 参数的自变量的 `ref` 修饰符等效于 `in`。请考虑改用 `in`。*
- CS9192：*自变量应随 `ref` 或 `in` 关键字一起传递。*
- CS9193：*自变量应为变量，因为它被传递给 `ref readonly` 参数*
- CS9195：*自变量应随 `in` 关键字一起传递*

- CS9196：*参数的引用类型修饰符与替代或实现的成员中相应的参数不匹配。*
- CS9197：*参数的引用类型修饰符与隐藏的成员中相应的参数不匹配。*
- CS9198：*参数的引用类型修饰符与目标中相应的参数不匹配。*
- CS9200：*为 `ref readonly` 参数指定了默认值，但 `ref readonly` 只应用于引用。请考虑将参数声明为 `in`。*
- CS9201：*使用前应引用赋值 ref 字段。*
- CS9265： *字段永远不会被重新分配给，并且将始终具有其默认值（null 引用）*

这些错误和警告遵循以下主题:

- ***语法不正确***：声明或用法的语法无效。
- ***语言构造，其中 ref 变量无效***：某些 C# 习惯用法不允许变量。 通常，这是因为无法可靠地执行 ref 安全分析。
- ***需要引用变量时使用了值表达式***：用作引用变量的表达式必须是变量，不能是值表达式。
- ***引用只读变量的可写引用变量***：无法通过可写引用传递对只读变量的引用。
- ***违反 ref 安全性***：引用变量不能引用具有更窄上下文的变量。 这意味着引用变量可以引用无效的内存。

本文使用"*引用变量*"一词来作为描述使用 `in`、`ref readonly`、`ref` 或 `out` 修饰符中的一个，或 `ref` 局部变量、`ref struct` 中的 `ref` 字段，或 `ref` 返回声明的参数的一般性术语。 引用变量引用另一个变量，称为"*引用对象*"。

语法错误

这些错误表示你对引用变量使用了不正确的语法：

- CS8373： `ref` *赋值的左侧必须是 ref 变量。*
- CS8388： `out` *变量无法声明为 ref 局部变量。*
- CS9190： `readonly` *修饰符必须在 `ref` 之后指定。*

可以通过以下更改之一更正错误：

- `= ref` 运算符的左操作数必须是引用变量。 有关正确语法的详细信息，请参阅参考变量。
- 参数修饰符 `ref readonly` 必须按该顺序排列。 `readonly ref` 不是合法的参数修饰符。 切换单词的顺序。
- 局部变量不能声明为 `out`。 要声明本地引用变量，请使用 `ref`。

引用变量限制

以下错误表示某个引用变量不能用于你已有一个的位置：

- CS0631：*`ref` 和 `out` 在此上下文中无效*
- CS0767：*无法使用指定的类型参数继承接口，因为它会导致方法包含仅在 `ref` 和 `out` 上存在不同的重载*
- CS1623：*迭代器不能具有 `ref`、`in` 或 `out` 参数*
- CS1741：*`ref` 或 `out` 参数不能有默认值*
- CS1939：*无法将范围变量作为 `out` 或 `ref` 参数传递*
- CS1988：*异步方法不能具有 `ref`、`in` 或 `out` 参数*
- CS7084：*无法将 Windows 运行时事件作为 `out` 或 `ref` 参数传递。*
- CS8196：对隐式类型的 `out` 变量的引用不允许出现在同一个参数列表中。
- CS8325：*"await" 不能在包含 `ref` 条件运算符的表达式中使用*
- CS8326：*这两个条件运算符的值必须都是 ref 值或者都不是 ref 值*
- CS8327：*表达式的类型必须正确，才能匹配备用 ref 值*
- CS8337：*"`ref`" 扩展方法的第一个参数必须是值类型或受结构约束的泛型类型。*
- CS8338：*扩展方法的第一个 "`in`" 或 "`ref readonly`" 参数必须是具体的（非泛型）值类型。*
- CS8977：*无法在具有 "UnmanagedCallersOnly" 特性的方法的签名中使用 "`ref`"、"`in`" 或 "`out`"。*
- CS9072：*析构变量不能声明为 ref 局部变量*
- CS9104：*不能在异步方法或异步 Lambda 表达式中使用类型的 `using` 语句资源。*
- CS9199：*`ref readonly` 参数不能具有 Out 特性。*

以下警告表示不应使用某个引用变量，并且它可能不安全：

- CS9196：*参数的引用类型修饰符与替代或实现的成员中相应的参数不匹配。*
- CS9197：*参数的引用类型修饰符与隐藏的成员中相应的参数不匹配。*
- CS9198：*参数的引用类型修饰符与目标中相应的参数不匹配。*
- CS9200：*为 `ref readonly` 参数指定了默认值，但 `ref readonly` 只应用于引用。请考虑将参数声明为 `in`。*
- CS9201：*使用前应引用赋值 ref 字段。*
- CS9265：*字段永远不会被重新分配给，并且将始终具有其默认值（null 引用）*

要修复此错误，请在不允许引用变量的位置移除它：

- 从索引器、迭代器和异步方法中移除 `in`、`ref` 和 `out` 参数。
- 移除包含 await 的 ref 条件表达式 (`? :`)。
- 从类型不是值类型或约束为值类型的泛型类型的扩展方法的第一个参数中移除 `ref` 修饰符。
- 两个[条件运算符表达式]都必须是或两个都不是 `ref` 变量。 从一个表达式中移除 `ref`，或将其添加到另一个表达式。 如果它是 `ref` 条件表达式，则两个表达式的类

型必须相同。
- `ref` 和 `out` 参数不能有默认值。 移除 `ref` 或 `out` 修饰符，或移除默认值。
- 隐式类型的 `out` 变量声明不能同时出现在同一参数列表中的其他位置。
- 不能在 `async` 方法 Lambda 表达式的 `using` 语句中放置引用变量。
- LINQ 查询表达式中的范围变量不能通过引用传递。
- 不能将对象析构为引用变量。 将引用变量替换为值变量。
- 无法实现方法重载仅在 `ref` 和 `out` 上存在差异的多个接口。 例如，一个接口声明 `void M(ref int i)`，另一个接口声明 `void M(out int i)`。 类无法实现这两个接口，因为方法无法区分。 只能实现其中一个接口。
- 使用 System.Runtime.InteropServices.UnmanagedCallersOnlyAttribute 特性化的方法不能使用引用参数。
- 无法将 Windows 运行时事件作为引用变量传递。
- `ref readonly` 参数不能在远程处理 API 中应用 System.Runtime.InteropServices.OutAttribute。
- `ref`在构造函数中或字段初始值设定项中初始化字段。

`unscoped ref` 限制

某些位置不允许 `ref` 参数的 `unscoped` 限定符：

- CS9101：UnscopedRefAttribute 只能应用于结构实例或虚拟接口方法和属性，不能应用于构造函数或仅初始化成员。
- CS9102：UnscopedRefAttribute 不能应用于接口实现，因为实现的成员没有此属性。

必须移除参数声明上导致错误的 `unscoped` 修饰符。

引用变量需要引用对象

必须向引用参数、引用返回或 ref 本地赋值提供一个变量作为自变量：

- **CS0206**：*非引用返回属性或索引器不能用作 `out` 或 `ref` 值*
- **CS1510**：*`ref` 或 `out` 值必须是可赋值变量*

警告：

- **CS9191**：*对应于 `in` 参数的自变量的 `ref` 修饰符等效于 `in`。请考虑改用 `in`。*
- **CS9192**：*自变量应随 `ref` 或 `in` 关键字一起传递。*
- **CS9193**：*自变量应为变量，因为它被传递给 `ref readonly` 参数*
- **CS9195**：*自变量应随 `in` 关键字一起传递*

当使用计算必须使用变量的值的表达式时，编译器会发出这些错误。 必须将该表达式的结果存储在变量中才能使用它。 例如，属性和索引器返回值，而不是变量。 必须将结果存储在变量中，并传递对该变量的引用。

可写引用变量需要可写引用对象

可写引用变量要求引用对象也可写。 以下错误表示变量不可写：

- CS0192：*字段 `readonly` 不能用作 `ref` 或 `out` 值（构造函数除外）*
- CS0199：*字段 `static readonly` 不能用作 `ref` 或 `out` 值（静态构造函数除外）*
- CS1605：*不能将变量用作 `ref` 或 `out` 值，因为它为只读*
- CS1649：*字段 `readonly` 的成员不能用作 `ref` 或 `out` 值（构造函数除外）*
- CS1651：*字段 `static readonly` 的字段不能用作 `ref` 或 `out` 值（静态构造函数除外）*
- CS1655：*不能将类型字段用作 `ref` 或 `out` 值*
- CS1657：*不能将变量用作 `ref` 或 `out` 值*
- CS8329：*不能将变量用作 `ref` 或 `out` 值，因为它是只读变量*
- CS8330：*变量的成员不能用作 `ref` 或 `out` 值，因为它是只读变量*
- CS8331：*无法赋值给变量或将其用作 `ref` 赋值的右侧，因为它是只读变量*
- CS8332：*无法赋值给变量的成员或将其用作 `ref` 赋值的右侧，因为它是只读变量*

不可写的变量的例子包括：

- 只读字段，包括实例字段和静态字段。
- `readonly` 字段的成员。
- `this` 变量。
- foreach 迭代变量
- using 变量或 fixed 变量。

必须复制该值并传递对副本的引用。

ref 安全违规

编译器会跟踪引用对象和引用变量的安全上下文。 当引用变量引用不再有效的引用对象变量时，编译器会在不安全代码中发出错误或警告。 引用对象必须至少具有与引用变量的 ref 安全上下文一样宽的安全上下文。 违反这些安全检查意味着引用变量会访问随机内存而不是引用对象变量。

- CS8166：*无法通过引用返回参数，因为它不是 `ref` 参数*
- CS8167：*无法通过引用返回参数成员，因为它不是 `ref` 或 `out` 参数*
- CS8168：*无法通过引用返回局部变量，因为它不是 ref 局部变量*

- **CS8169**：*无法通过引用返回局部变量成员，因为它不是 ref 局部变量*
- **CS8345**：*字段或自动实现的属性不能是类型，除非它是 `ref struct` 的实例成员。*
- **CS8351**：*`ref` 条件运算符的分支不能引用具有不兼容声明范围的变量*
- **CS8374**：*无法引用赋值，源的转义范围比目标更窄。*
- **CS9077**：*无法通过 `ref` 参数按引用返回参数，它只能在 return 语句中返回*
- **CS9078**：*无法通过 `ref` 参数按引用返回参数的成员，它只能在 return 语句中返回*
- **CS9079**：*无法将源引用赋值给目标，因为源只能通过 return 语句转义当前方法。*
- **CS9096**：*无法将源引用赋值给目标，因为源的值转义范围比目标更广，因此允许通过转义范围比源更窄的值目标进行赋值。*

警告：

- **CS9085**：*这会将源引用赋值给目标，但源的转义范围比目标要窄。*
- **CS9086**：*ref 条件运算符的分支引用具有不兼容声明范围的变量*
- **CS9087**：*这会按引用返回一个参数，但它不是 `ref` 参数*
- **CS9089**：*这会按引用返回不是 `ref` 或 `out` 参数的参数成员*
- **CS9091**：*这会按引用返回局部变量，但它不是 ref 局部变量*
- **CS9092**：*这会按引用返回局部变量的成员，但它不是 ref 局部变量*
- **CS9093**：*这会将源引用赋值给目标，但源只能通过 return 语句对当前方法进行转义。*
- **CS9094**：*这会通过 `ref` 参数按引用返回参数，但它只能在 return 语句中安全返回*
- **CS9095**：*这会通过 `ref` 参数按引用返回参数的成员，但它只能在 return 语句中安全返回*
- **CS9097**：*这会将源引用赋值给目标，但源的值转义范围比目标更广，因此可以通过转义范围比源更窄的值目标进行赋值。*

编译器使用静态分析来确定引用对象是否在可以使用引用变量的所有地方有效。 你需要重构代码，使引用对象在引用变量可能引用它的所有位置保持有效。 有关 ref 安全规则的详细信息，请参阅 ref 安全上下文中的 C# 标准。

与 `ref struct` 类型相关的错误和警告

项目 • 2024/11/08

- CS8343：`ref structs` 无法实现接口
- CS8344：`foreach` 语句无法对异步或迭代器方法中的枚举器进行操作，因为类型是 `ref struct` 或者是允许 `ref struct` 的类型参数。
- **CS8345**：*字段或自动实现的属性不能是类型，除非它是 `ref struct` 的实例成员。*
- CS9048：`scoped` 修饰符只能用于 refs 和 `ref struct` 值。
- CS9050：`ref` 字段无法引用 `ref struct`。
- CS9059：ref 字段只能在 ref struct 中声明。
- CS9241：已指定 `ref struct`。
- CS9242：`allows` 约束子句必须是指定的最后一个约束。
- CS9243不允许将其他约束已知的类型参数的 ref struct 用作类。
- CS9244：类型不能是 `ref struct` 或允许 ref struct 的类型参数，以便在泛型类型或方法中使用它作为参数。
- CS9245：类型无法实现 `ref struct` 类型的接口成员。
- CS9246：在允许 ref struct 的类型参数上无法访问非虚拟实例接口成员。
- CS9247：foreach 语句无法对类型的枚举器进行操作，因为它是一个允许 ref struct 的类型参数，并且在编译时无法确定它是否实现了 `IDisposable`。
- **CS9267**：*迭代器的元素类型可能不是 ref 结构或允许 ref 结构的类型参数*

ref 安全违规

- **CS8345**：*字段或自动实现的属性不能是类型，除非它是 `ref struct` 的实例成员。*
- CS9048：`scoped` 修饰符只能用于 refs 和 `ref struct` 值。
- CS9050：`ref` 字段无法引用 `ref struct`。
- CS9059：`ref` 字段只能在 `ref struct` 中声明。

ref struct 类型可以包含 `ref` 字段。 其他类型不允许使用 `ref` 字段。 编译程序对 `ref struct` 类型的声明和使用强制执行限制，以对任何 `ref struct` 类型的实例强制执行 ref 安全规则：

- 只有 `ref struct` 类型可以包含自动实现 `ref` 的属性。
- 只有 `ref struct` 类型或 `ref` 变量才能具有 `scoped` 修饰符。
- `ref` 字段只能在 `ref struct` 类型中声明。
- `ref` 字段无法引用 `ref struct` 类型/

违反其中任一规则都会产生所列错误之一。 如果打算使用该语言功能，请将类型转换为 `ref struct`。 否则，请移除不允许的构造。

ref struct 接口实现

- CS8343：`ref structs` 无法实现接口
- CS8344：`foreach` 语句无法对异步或迭代器方法中的枚举器进行操作，因为类型是 `ref struct` 或者是允许 `ref struct` 的类型参数。
- CS9241：已指定 `ref struct`。
- CS9242：`allows` 约束子句必须是指定的最后一个约束。
- CS9243不允许将其他约束已知的类型参数的 ref struct 用作类。
- CS9244：类型不能是 `ref struct` 或允许 ref struct 的类型参数，以便在泛型类型或方法中使用它作为参数。
- CS9245：类型无法实现 `ref struct` 类型的接口成员。
- CS9246：在允许 ref struct 的类型参数上无法访问非虚拟实例接口成员。
- CS9247：foreach 语句无法对类型的枚举器进行操作，因为它是一个允许 ref struct 的类型参数，并且在编译时无法确定它是否实现了 `IDisposable`。
- **CS9267**： *迭代器的元素类型可能不是 ref 结构或允许 ref 结构的类型参数*

在 C# 13 之前，ref struct 类型无法实现接口；编译程序生成 CS8343。 从 C# 13 开始，`ref struct` 类型可以实现接口，但要遵循以下规则：

- 无法将 `ref struct` 转换为它实现的接口的实例。 在参数为接口类型时，若你使用 `ref struct` 类型作为参数时，则此限制包括隐式转换。 该转换会导致装箱转换，这违反了 ref 安全性。
- 实现接口的 `ref struct` 必须实现所有接口成员。 `ref struct` 必须实现接口包含默认实现的成员。

从 C# 13 开始，`ref struct` 可以用作泛型类型参数的类型实参，但前提是泛型类型参数具有 allows ref struct 反约束。 在使用 `allows ref struct` 反约束时，必须遵循以下规则：

- `ref struct` 用作类型实参，类型参数必须具有 `allows ref struct` 反约束。- `allows ref struct` 反约束必须在该参数的 `where` 子句的末尾
- 类型参数必须遵循 ref 安全规则来使用实例。
- `ref struct`可以是类型`ref struct`的类型参数不能用作迭代器方法的元素类型。

与 `yield return` 语句和迭代器方法相关的错误和警告

项目 • 2024/12/21

有许多与 语句和迭代器方法相关的`yield return`：

- CS1622：*无法从迭代器中返回值。请使用 yield return 语句返回值，或生成中断以结束迭代。*
- CS1624：*“accessor”的正文不能为迭代器块，因为“type”不是迭代器接口类型*
- CS1625：*无法在 finally 子句体中生成*
- CS1626：*无法在包含 catch 子句的 Try 块体中生成值*
- CS1627：*yield return 之后应为表达式*
- CS1629：*迭代器中不能出现不安全的代码*
- CS1631：*无法在 catch 子句体中生成值*
- CS1637：*迭代器不能有不安全的参数或 yield 类型*
- CS4013：*类型的实例不能在嵌套函数、查询表达式、迭代器块或异步方法中使用*
- CS8154：*主体不能是迭代器块，因为它按引用返回*
- CS8176：*迭代器不能有按引用局部变量*
- CS9238：*无法在“不安全”的块中使用“yield return”*
- CS9239：*该 `&` 运算符不能用于迭代器方法中的参数或局部变量。*

迭代器方法的结构

迭代器方法必须符合 C# 中的多个规则。 当你的迭代器方法违反其中一个或多个规则时，编译器会发出以下错误：

- CS1622：*无法从迭代器中返回值。请使用 yield return 语句返回值，或生成中断以结束迭代。*
- CS1624：*“accessor”的正文不能为迭代器块，因为“type”不是迭代器接口类型*
- CS1627：*yield return 之后应为表达式*
- CS1637：*迭代器不能有不安全的参数或 yield 类型*
- CS8154：*主体不能是迭代器块，因为它按引用返回*

你的迭代器方法必须遵循以下规则：

- 迭代器方法（使用 `yield return` 和可选的 `yield break`）也不能使用 `return` 语句返回序列。
- 迭代器方法必须将*迭代器接口类型*声明为返回类型。 *迭代器接口类型*为：IEnumerable、IEnumerable<T>、IEnumerator、IEnumerator<T>。

- `yield return` 语句必须包含一个表达式才能作为序列的一部分返回。 `yield return;` 无效。
- 迭代器方法不能将不安全的类型用作参数，如指针。
- 迭代器方法不能 `yield return` 不安全的类型，例如指针。
- 迭代器方法不能通过 `yield return ref`。 你必须按值返回。

迭代器方法的限制

迭代器方法的主体必须符合 `yield return` 语句及其上下文的限制。 当迭代器违反以下一个限制时，编译器会发出以下错误：

- **CS1625**：*无法在 finally 子句体中生成*
- **CS1626**：*无法在包含 catch 子句的 Try 块体中生成值*
- **CS1631**：*无法在 catch 子句体中生成值*
- **CS1629**：*迭代器中不能出现不安全的代码*
- **CS9238**：*无法在"不安全"的块中使用"yield return"*
- **CS9239**：*该 `&` 运算符不能用于迭代器方法中的参数或局部变量。*

这些错误表明你的代码违反了安全规则，因为迭代器返回一个元素并继续生成下一个元素：

- 你不能通过 `yield return` 或 `catch` 子句执行 `finally` 操作。
- 你不能通过 catch 子句从 `yield return` 执行 `try` 操作。
- 你不能从 `yield return` 块中执行 `unsafe` 操作。 迭代器的上下文在封闭的 `safe` 块中创建嵌套的 `unsafe` 块。
- 你不能使用 `&` 运算符在迭代器方法中获取变量的地址。

在 C# 13 之前，迭代器不能包含 `unsafe` 代码 (CS1629)。 从 C# 13 开始，此限制已放宽。 所有 `yield return` 语句都必须位于安全上下文中，但迭代器方法可以包含 `unsafe` 代码。

迭代器方法中的 ref 安全性

迭代器方法具有特殊的 ref 安全性限制。 C# 13 中放宽了这些规则：

- **CS4013**：*类型的实例不能在嵌套函数、查询表达式、迭代器块或异步方法中使用*
- **CS8176**：*迭代器不能有按引用局部变量*

在 C# 13 之前，迭代器无法声明 `ref` 局部变量。 它们无法声明 `ref struct` 类型的任何变量。

从 C# 13 开始，可以在迭代器方法中使用 `ref struct` 类型（如果未跨 `yield return` 语句访问它们）。 从 C# 13 开始，迭代器方法可以声明 `ref` 局部变量。

与 `partial` 类型和 `partial` 成员声明相关的错误和警告

项目 • 2024/11/14

有许多与 `partial` 类型和 `partial` 成员声明相关的*错误*:

- CS0260：*类型为"type"的声明上缺少分部修饰符；存在此类型的其他分部声明*
- CS0261：*"type"的分部声明必须是所有的类、所有的结构或所有的接口*
- CS0262：*"type"的分部声明包含冲突的可访问性修饰符*
- CS0263：*"type"的分部声明必须指定相同的基类*
- CS0264：*"type"的分部声明必须具有顺序相同的相同类型形参名*
- CS0265：*"type"的分部声明对类型参数"type parameter"具有不一致的约束*
- CS0267：*"`partial`"修饰符只能出现在紧靠"`class`"、"`record`"、"`struct`"、"`interface`"或方法或属性返回类型前面的位置。*
- CS0282：*分部类或结构"type"的多个声明中的字段之间没有定义的排序。要指定排序，所有实例字段必须位于同一声明中。*
- CS0501：*"member function"必须声明主体，因为它未标记为 `abstract`、`extern` 或 `partial`*
- CS0750：*分部成员不能有"`abstract`"修饰符。*
- CS0751：*必须在分部 `class` 或分部 `struct` 中声明分部成员*
- CS0754：*分部方法不能显式实现接口方法。*
- CS0755：*两个分部方法声明都必须是扩展方法，或者都不能是扩展方法。*
- CS0756：*分部方法不能有多个定义声明。*
- CS0757：*分部方法不能有多个实现声明。*
- CS0759：*没有为分部方法的实现声明找到定义声明。*
- CS0761：`method<T>` *分部方法的声明具有不一致的类型参数约束。*
- CS0762:*无法通过方法创建委托，因为该方法是没有实现声明的分部方法*
- CS0763：*两个分部方法声明必须都是 `static`，或者两者都不能是 `static`。*
- CS0764：*两个分部方法声明必须都是 `unsafe`，或者两者都不能是 `unsafe`。*
- CS1067：*分部声明必须具有顺序相同的相同类型形参名和差异修饰符。*
- CS8142：*这两个分部成员声明都必须使用相同的元组元素名称。*
- CS8663：*分部成员声明都必须为只读，或者两者都不能是只读*
- CS8796：*分部方法必须具有辅助功能修饰符，因为它具有非空返回类型。*
- CS8795：*分部成员必须有一个实施部分，因为它具有辅助功能修饰符。*
- CS8797：*分部方法必须具有辅助功能修饰符，因为它具有"`out`"参数。*
- CS8798：*分部方法必须具有辅助功能修饰符，因为它具有"`virtual`"、"`override`"、"`sealed`"、"`new`"或"`extern`"修饰符。*
- CS8799：*这两个分部成员声明必须具有相同的辅助功能修饰符。*

- CS8800：*分部成员声明必须具有相同的 `virtual`、`override` 和 `sealed` `new` 修饰符组合。*
- CS8817：*这两个分部方法声明必须具有相同的返回类型。*
- CS8818：*分部成员声明必须具有匹配的 `ref` 返回值。*
- CS8863：*只有一个分部类型声明可以拥有参数列表*
- CS8988：*参数的 `scoped` 修饰符与分部定义不匹配。*
- CS9248：*分部属性必须具有实现部分。*
- CS9249：*分部属性必须具有定义部件。*
- CS9250：*分部属性可能没有多个定义声明，不能是自动属性。*
- CS9251：*分部属性不能有多个实现声明*
- CS9252：*必须实现分部属性访问器，因为在定义部件上声明了它*
- CS9253：*属性访问器不会实现在定义部件上声明的任何访问器*
- CS9254：*属性访问器必须与定义部件匹配*
- CS9255：*两个分部属性声明必须具有相同的类型。*
- CS9256：*分部属性声明有签名差异。*
- CS9257：*必须同时要求这两个分部属性声明，或者都不需要这两个声明*
- CS9258：*在此语言版本中，“`field`”关键字绑定到属性的合成后盾字段。若要避免生成合成后盾字段，并引用现有成员，请改用“”`this.field`或“”。`@field`*
- CS9263： *部分属性不能对定义和实现具有初始值设定项。*

可以为字段支持的属性生成以下警告：

- CS9264： *退出构造函数时，不可为 null 的属性必须包含非 null 值。请考虑添加“必需”修饰符，或将属性声明为可为 null，或添加“`[field: MaybeNull, AllowNull]`”属性。 **
- CS9266： *属性的一个访问器应使用“”`field`，因为另一个访问器正在使用它。*

以下部分介绍了这些错误和警告的原因和修复。

分部类型

- CS0260：*类型为“type”的声明上缺少分部修饰符；存在此类型的其他分部声明*
- CS0261：*“type”的分部声明必须是所有的类、所有的结构或所有的接口*
- CS0262：*“type”的分部声明包含冲突的可访问性修饰符*
- CS0263：*“type”的分部声明必须指定相同的基类*
- CS0264：*“type”的分部声明必须具有顺序相同的相同类型形参名*
- CS0265：*“type”的分部声明对类型参数“type parameter”具有不一致的约束*
- CS0267：*“`partial`”修饰符只能出现在紧靠“`class`”、“`record`”、“`struct`”、“`interface`”或方法或属性返回类型前面的位置。*
- CS8863：*只有一个分部类型声明可以拥有参数列表*

你的分部类型声明可能会导致编译器发出以下警告：

- CS0282：*分部 `class` 或 `struct` "type"的多个声明中的字段之间没有定义的排序。要指定排序，所有实例字段必须位于同一声明中。*

对于任何分部类型，`partial` 关键字必须紧挨着 `class`、`record`、`struct` 或 `interface` 之前。 如果编译器按任何其他顺序显示，则会发出错误。 此外：

- 分部类型的所有声明必须类型匹配（`class`、`struct`、`record class`、`record struct`、`readonly struct` 或 `readonly record struct`）。
- 所有声明都必须包含 `partial` 修饰符。
- 泛型分部类型的声明必须按相同顺序包含相同的类型参数。

在类型的所有声明上，不需要重复声明的某些部分。 但是，如果这些元素在多个 `partial` 声明中重复，那么它们必须匹配：

- 任何访问修饰符，例如 `public`。
- 任何基类或已实现的接口。
- 对类型参数的任何约束。

最多可以在一个分部类型的声明上声明主要构造函数。

如果 `partial struct` 类型的多个文件中声明了多个字段，则编译器会发出警告。 如果布局顺序很重要，则必须在同一文件中声明所有字段。 如果顺序无关紧要，则可以将 System.Runtime.InteropServices.StructLayoutAttribute 与 LayoutKind.Auto 值一起使用。

分部（成员）

- CS0750： *分部成员不能有"`abstract`"修饰符。*
- CS0751：*必须在分部 `class` 或分部 `struct` 中声明分部成员*
- CS0754：*分部方法不能显式实现接口方法。*
- CS0763：*两个分部方法声明必须都是 `static`，或者两者都不能是 `static`。*
- CS0764：*两个分部方法声明必须都是 `unsafe`，或者两者都不能是 `unsafe`*
- CS8142：*这两个分部成员声明都必须使用相同的元组元素名称。*
- CS8663：*分部成员声明都必须为只读，或者两者都不能是只读*
- CS8799：*这两个分部成员声明必须具有相同的辅助功能修饰符。*
- CS8800：*分部成员声明必须具有相同的 `virtual`、`override`、`sealed` 和 `new` 修饰符组合。*
- CS8818：*分部成员声明必须具有匹配的 `ref` 返回值。*
- CS8988：*参数的 `scoped` 修饰符与分部定义不匹配。*

分部成员有两个声明。 没有实现的声明是*声明声明*。 具有实现的声明是 *实现声明*。 只允许在 `partial` 类型中使用分部成员。 分部成员不能为 `abstract`。 分部成员无法显式实现接口。 分部成员的两个声明都必须具有相同的签名。 例如，两个声明都可以或者都不能包含 `static` 或 `unsafe` 修饰符。

分部方法

- CS0501：*“member function”必须声明主体，因为它未标记为 `abstract`、 `extern` 或 `partial`*
- CS0755：*两个分部方法声明都必须是扩展方法，或者都不能是扩展方法。*
- CS0756：*分部方法不能有多个定义声明。*
- CS0757：*分部方法不能有多个实现声明。*
- CS0759：*没有为分部方法的实现声明找到定义声明。*
- CS0761：`method<T>` *分部方法的声明具有不一致的类型参数约束。*
- CS0762：*无法通过方法创建委托，因为该方法是没有实现声明的分部方法*
- CS1067：*分部声明必须具有顺序相同的相同类型形参名和差异修饰符。*
- CS8796：*分部方法必须具有辅助功能修饰符，因为它具有非空返回类型。*
- CS8795： *分部成员必须有一个实施部分，因为它具有辅助功能修饰符。*
- CS8797：*分部方法必须具有辅助功能修饰符，因为它具有“`out`”参数。*
- CS8798：*分部方法必须具有辅助功能修饰符，因为它具有“`virtual`”、“`override`”、“`sealed`”、“`new`”或“`extern`”修饰符。*
- CS8817：*这两个分部方法声明必须具有相同的返回类型。*

某些 `partial` 方法声明不需要*实现声明*。 也就是说，如果成员返回 `void`，则不声明任何访问修饰符（包括默认的 `private` 修饰符），并且不包含任何 `virtual`、 `override`、 `sealed` 或 `new` 修饰符。 否则，任何分部方法都必须包括声明和实现声明。

当分部方法包含实现声明时，这两个声明必须相同。 只能定义一个实现声明。

分部属性

以下错误表示分部属性或索引器声明中的错误：

- CS9248：*分部属性必须具有实现部分。*
- CS9249：*分部属性必须具有定义部件。*
- CS9250：*分部属性可能没有多个定义声明，不能是自动属性。*
- CS9251：*分部属性不能有多个实现声明*
- CS9252：*必须实现分部属性访问器，因为在定义部件上声明了它*
- CS9253：*属性访问器不会实现在定义部件上声明的任何访问器*
- CS9254：*属性访问器必须与定义部件匹配*

- **CS9255**：*两个分部属性声明必须具有相同的类型。*
- **CS9257**：*必须同时要求这两个分部属性声明，或者都不需要这两个声明*

以下警告指示了在分部属性中声明和实现声明的签名差异：

- **CS9256**：*分部属性声明有签名差异。*

分部属性或索引器必须都具有*声明声明*和*实现声明*。 这两个声明的签名必须匹配。 *由于声明声明*使用的语法与自动实现的属性相同，*因此实现声明*不能是自动实现的属性。 访问器必须至少有一个访问器主体。 从 C# 13 开始，可以使用 field 关键字通过简洁的语法声明一个访问器：

```
C#

public partial int ImplementingDeclaration { get => field; set; }
```

字段支持的属性

- **CS9258**：*在此语言版本中，“`field`”关键字绑定到属性的合成后盾字段。若要避免生成合成后盾字段，并引用现有成员，请改用“”`this.field`或“”。`@field`*
- **CS9263**： *部分属性不能对定义和实现具有初始值设定项。*
- **CS9264**： *退出构造函数时，不可为 null 的属性必须包含非 null 值。请考虑添加“必需”修饰符，或将属性声明为可为 null，或添加“`[field: MaybeNull, AllowNull]`”属性。**
- **CS9266**： *属性的一个访问器应使用“”`field`，因为另一个访问器正在使用它。*

> **ⓘ 重要**
>
> 关键字 `field` 是 C# 13 中的预览功能。 必须使用 .NET 9 并将元素`preview`设置为`<LangVersion>`项目文件中，才能使用`field`上下文关键字。
>
> 应注意在 `field` 具有名为 `field`字段的类中使用关键字功能。 新 `field` 关键字将隐藏属性访问器范围中命名 `field` 的字段。 可以更改变量的名称 `field` ，或使用 `@` 令牌将标识符引用 `field` 为 `@field`。 可以通过阅读关键字**的功能规范**`field`来了解详细信息。

从 C# 13 开始，预览功能 `field` 支持的属性允许访问属性的编译器合成后备字段。 **CS9258** 指示你有一个名为的变量，该变量 `field`可由上下文关键字 `field`隐藏。

CS9263 指示声明包括实现。 该实现可能正在访问该属性的编译器合成后盾字段。 **CS9264** 指示在属性声明可为 null 时，使用 `field` 假定不可为 null 的后盾字段。 编译器

假定后备字段和属性具有相同的可为 null 性。 需要将 `[field:MaybeNull, AllowNull]` 属性添加到属性声明，以 `field` 指示值应被视为可为 null。 CS9266 指示其中一个属性访问器使用 `field` 关键字，但另一个使用手动声明的后盾字段。 该警告表明你可能偶然做到了。

与方法参数上的 `params` 修饰符相关的错误和警告

项目 • 2024/06/29

与 `lock` 语句和线程同步相关的一些错误：

- CS0225：params 参数必须是一维数组或具有有效的集合类型
- CS0231：params 参数必须是形参列表中的最后一个参数。
- CS0466：由于“method2”没有 params 参数，因此“method1”也不应当有该参数
- CS0674：请勿使用 `System.ParamArrayAttribute` 或 `System.ParamArrayAttribute`/`System.Runtime.CompilerServices.ParamCollectionAttribute`。请改用 `params` 关键字。
- CS0758：两种分部方法声明必须要么都使用 `params` 参数，要么都不使用 `params` 参数
- CS1104：参数数组不能与 `this` 修饰符一起在扩展方法中使用。
- CS1611：params 参数不能声明为 in `ref` 或 `out`
- CS1670：`params` 在此上下文中无效
- CS1751：无法为参数数组指定默认值。
- CS9218：由于使用了具有动态类型的实参，并且该方法具有非数组 params 集合形参，因此无法从用法中推断出方法的类型参数。请尝试显式指定类型参数。
- CS9223：创建 params 集合会导致构造函数的无限调用链。
- CS9224：方法的可见性不能低于具有 params 集合的成员。
- CS9225：构造函数未初始化所需的成员。
- CS9227：类型不包含适当实例 `Add` 方法的定义。
- CS9228：非数组 params 集合类型必须具有可以不带参数调用的适用构造函数。

方法声明规则

以下错误表明在参数上使用 `params` 修饰符，而该上下文中不允许使用 `params` 修饰符：

- CS0231：params 参数必须是形参列表中的最后一个参数。
- CS1104：参数数组不能与 `this` 修饰符一起在扩展方法中使用。
- CS1611：params 参数不能声明为 in `ref` 或 `out`
- CS1670：`params` 在此上下文中无效
- CS1751：无法为参数数组指定默认值。

编译器对在方法参数上使用 `params` 修饰符强制执行以下规则：

- `params` 修饰符仅可用于形参列表中的最后一个参数。 这包括具有默认值的任何参数。
- 使用 `params` 修饰符时，不能包含形参的默认实参。
- `params` 修饰符不能应用于引用参数。 引用参数是具有 `in`、`ref readonly`、`ref` 或 `out` 修饰符的参数。
- 不能在扩展方法上组合 `params` 修饰符和 `this` 修饰符。
- `params` 修饰符不能用于重载运算符。

在 C# 12 之前的版本中，`params` 修饰符不能用于匿名方法或 lambda 表达式的参数。

形参和实参类型规则

以下错误指示用于 `params` 的参数类型无效：

- CS9218：由于使用了具有动态类型的实参，并且该方法具有非数组 params 集合形参，因此无法从用法中推断出方法的类型参数。请尝试显式指定类型参数。
- CS0225：params 参数必须是一维数组或具有有效的集合类型
- CS9227：类型不包含适当实例 `Add` 方法的定义。
- CS9228：非数组 params 集合类型必须具有可以不带参数调用的适用构造函数。

在 C# 13 之前的版本中，`params` 修饰符仅适用于一维数组。 其他类型均无效。

从 C# 13 开始，可以使用任何有效的集合类型。 但是，仍存在一些限制。 集合类型必须遵循与集合表达式的目标相同的规则。

其他 params 错误

以下错误指示使用 `params` 修饰符的其他问题：

- CS0466：由于“method2”没有 params 参数，因此“method1”也不应当有该参数
- CS0674：请勿使用 `System.ParamArrayAttribute` 或 `System.Runtime.CompilerServices.ParamCollectionAttribute`。请改用 `params` 关键字。
- CS0758：两种分部方法声明必须要么都使用 `params` 参数，要么都不使用 `params` 参数
- CS9223：创建 params 集合会导致构造函数的无限调用链。
- CS9224：方法的可见性不能低于具有 params 集合的成员。
- CS9225：构造函数未初始化所需的成员。

当且仅当接口成员具有 `params` 修饰符时，实现接口的方法才必须包含 `params` 修饰符。 同样，`partial` 方法的两个声明要么都必须包含 `params` 修饰符，要么都不得包含 `params`

修饰符。

必须使用 `params` 修饰符。 不能应用等效属性 System.ParamArrayAttribute 或 System.Runtime.CompilerServices.ParamCollectionAttribute。

当生成的用于创建集合类型的代码无效时，编译器会生成上述列表中的最后三个错误之一：

- 当发出的用于创建集合的代码还包含相同类型的 params 集合时，编译器会发出 CS9223。 通常，`Add` 方法采用同一类型的 `params` 集合。
- 当集合类型的 `Create` 方法的可访问性低于采用集合类型的 `params` 参数的方法时，编译器会发出 CS9224。
- 当集合类型具有必需的成员，而无参数构造函数未初始化该成员，且无参数构造函数上有 System.Diagnostics.CodeAnalysis.SetsRequiredMembersAttribute 时，编译器会发出 CS9225。

另请参阅

- 扩展方法
- 分部类和方法
- 集合表达式
- params

在 GitHub 上与我们协作

可以在 GitHub 上找到此内容的源，还可以在其中创建和查看问题和拉取请求。 有关详细信息，请参阅参与者指南。

.NET

.NET 反馈

.NET 是一个开放源代码项目。 选择一个链接以提供反馈：

提出文档问题

提供产品反馈

解决可以为 null 的警告

项目 • 2023/08/01

本文介绍以下编译器警告：

- CS8597 - *抛出的值可能为 null。*
- CS8600 - *将 null 字面量或可能为 null 的值转换为非 null 类型。*
- CS8601 - *引用类型赋值可能为空。*
- CS8602 - *解引用可能为 null 的引用。*
- CS8603 - *可能返回空引用。*
- CS8604 - *形参可能传入空引用实参。*
- CS8605 - *取消装箱可能为 null 的值。*
- CS8607 - *可能的 null 值不能用于标有 `[NotNull]` 或 `[DisallowNull]` 的类型*
- CS8608 - *类型中引用类型的为 Null 性与重写成员不匹配。*
- CS8609 - *返回类型中引用类型的为 Null 性与重写成员不匹配。*
- CS8610 - *参数类型中引用类型的为 Null 性与重写成员不匹配。*
- CS8611 - *参数类型中引用类型的为 Null 性与分部方法声明不匹配。*
- CS8612 - *类型中引用类型的为 Null 性与隐式实现的成员不匹配。*
- CS8613 - *返回类型中引用类型的为 Null 性与隐式实现的成员不匹配。*
- CS8614 - *参数类型中引用类型的为 Null 性与隐式实现的成员不匹配。*
- CS8615 - *类型中引用类型的为 Null 性与实现的成员不匹配。*
- CS8616 - *返回类型中引用类型的为 Null 性与实现的成员不匹配。*
- CS8617 - *参数类型中引用类型的为 Null 性与实现的成员不匹配。*
- CS8618 - *在退出构造函数时，不可为 null 的变量必须包含非 null 值。请考虑声明为可以为 null。*
- CS8619 - *值中的引用类型的为 Null 性与目标类型不匹配。*
- CS8620 - *由于引用类型的可为 null 性差异，实参不能用于形参。*
- CS8621 - *返回类型中引用类型的为 Null 性与目标委托不匹配（可能是由于为 Null 性特性）。*
- CS8622 - *参数类型中引用类型的为 Null 性与目标委托不匹配（可能是由于为 Null 性特性）。*
- CS8624 - *由于引用类型的可为 null 性差异，实参不能用作输出。*
- CS8625 - *无法将 null 字面量转换为非 null 的引用类型。*
- CS8629 - *可为 null 的值类型可为 null。*
- CS8631 - *类型不能用作泛型类型或方法中的类型参数。类型参数的为 Null 性与约束类型不匹配。*
- CS8633 - *方法的类型参数的约束中的为 Null 性与接口方法的类型参数的约束不匹配。请考虑改用显式接口实现。*
- CS8634 - *类型不能用作泛型类型或方法中的类型参数。类型参数的为 Null 性与“class”约束不匹配。*

- CS8643 - *显式接口说明符中引用类型的 Null 性与该类型实现的接口不匹配。*
- CS8644 - *类型不实现接口成员。接口中基类型实现的引用类型的 Null 性不匹配。*
- CS8645 - *成员已列入类型的接口列表中，其中包含不同引用类型的 Null 性。*
- CS8655 - *Switch 表达式不会处理某些为 null 的输入（它并非详尽无遗）。*
- CS8667 - *分部方法声明在对类型参数的约束中具有不一致的为 Null 性。*
- CS8670 - *对象或集合初始值设定项会隐式解引用可能为 null 的成员。*
- CS8714 - *类型不能用作泛型类型或方法中的类型参数。类型参数的为 Null 性与“notnull”约束不匹配。*
- CS8762 - *退出时，参数必须具有非 null 值。*
- CS8763 - *不能返回标记为 `[DoesNotReturn]` 的方法。*
- CS8764 - *返回类型的为 Null 性与重写成员不匹配（可能是由于为 Null 性特性）。*
- CS8765 - *参数类型的为 Null 性与重写成员不匹配（可能是由于为 Null 性特性）。*
- CS8766 - *返回类型中引用类型的为 Null 性与隐式实现的成员不匹配（可能是由于为 Null 性特性）。*
- CS8767 - *参数类型中引用类型的为 Null 性与隐式实现的成员不匹配（可能是由于为 Null 性特性）。*
- CS8768 - *返回类型中引用类型的为 Null 性与实现的成员不匹配（可能是由于为 Null 性特性）。*
- CS8769 - *参数类型中引用类型的为 Null 性与实现的成员不匹配（可能是由于为 Null 性特性）。*
- CS8770 - *方法缺少 `[DoesNotReturn]` 注释，无法匹配已实现的或被替代的成员。*
- CS8774 - *退出时，成员必须具有非 null 值。*
- CS8776 - *不能在此特性中使用此成员。*
- CS8775 - *退出时，成员必须具有非 null 值。*
- CS8777 - *退出时，参数必须具有非 null 值。*
- CS8819 - *返回类型中引用类型的为 Null 性与分部方法声明不匹配。*
- CS8824 - *退出时参数必须具有非 null 值，因为参数是非 null。*
- CS8825 - *由于参数为非 null，因此返回值必须为非 null。*
- CS8847 - *Switch 表达式不会处理一些 null 输入(它不是穷举)。但是，带有“when”子句的模式可能成功匹配此值。*

可以为 null 的警告的作用是将应用程序在运行时引发 System.NullReferenceException 的机率降到最低。 若要实现此目标，编译器将使用静态分析，并在代码具有可能导致空引用异常的构造时发出警告。 通过应用类型注释和属性为编译器提供其静态分析的信息。 这些注释和特性描述了自变量、参数和类型成员的可为 null 性。 在本文中，你将学习不同的技术来处理编译器从其静态分析生成的可为 null 的警告。 此处所述的方法适用于一般 C# 代码。 通过阅读使用可为 null 的引用类型，了解如何使用可以为 null 的引用类型和实体框架核心。

使用以下四种方法之一可处理几乎所有警告：

- 添加必要的 null 检查。
- 添加 `?` 或 `!` 可为 null 的注释。
- 添加描述 null 语义的特性。
- 正确初始化变量。

可能的取消 null 引用

这一组警告会提醒你正在对一个“null 状态”是“可能为 null”的变量执行取消引用操作。 这些警告如下：

- **CS8602** - *解引用可能为 null 的引用。*
- **CS8670** - *对象或集合初始值设定项会隐式解引用可能为 null 的成员。*

以下代码演示上述每个警告的一个示例：

C#

```
class Container
{
    public List<string>? States { get; set; }
}

internal void PossibleDereferenceNullExamples(string? message)
{
    Console.WriteLine(message.Length); // CS8602

    var c = new Container { States = { "Red", "Yellow", "Green" } }; //
CS8670
}
```

在上例中，警告是因为 `Container` 和 `c` 的 `States` 属性可能具有 null 值。 向可能为 null 的集合分配新状态会导致警告。

若要删除这些警告，需要在取消引用之前添加代码，将该变量的“null 状态”更改为“不为 null”。 集合初始值设定项警告可能更难以发现。 初始化表达式向集合添加元素时，编译器检测到该集合可能为 null。

在许多情况下，可以通过在对变量进行取消引用之前检查变量是否为 null 来消除这些警告。 例如，可以将上面的示例重新编写为：

C#

```
void WriteMessageLength(string? message)
{
    if (message is not null)
    {
```

```
        Console.WriteLine(message.Length);
    }
}
```

其他情况如果收到这些警告，则可能为误报。 你可能有一个专用的实用程序方法，用于测试 null 状态。 编译器不知道此方法提供了 null 检查。 请看下面的示例，该示例使用专用实用程序方法 `IsNotNull`：

C#

```
public void WriteMessage(string? message)
{
    if (IsNotNull(message))
        Console.WriteLine(message.Length);
}
```

编译器会警告你在编写属性 `message.Length` 时可能会取消 null 引用，因为其静态分析确定 `message` 可能是 `null`。 你可能知道 `IsNotNull` 提供了一个 null 检查，当它返回 `true` 时，`message` 的 null 状态应为“不为 null”。 需要告知编译器这些事实。 一种方法是使用 null 包容性运算符 `!`。 可以更改 `WriteLine` 语句，使其与以下代码相匹配：

C#

```
Console.WriteLine(message!.Length);
```

Null 包容性运算符使表达式“不为 null”，即使它曾经“可能为 null”且未应用 `!`。 在此示例中，更好的解决方案是将一个特性添加到 `IsNotNull` 的签名：

C#

```
private static bool IsNotNull([NotNullWhen(true)] object? obj) => obj !=
null;
```

当方法返回 `true` 时，System.Diagnostics.CodeAnalysis.NotNullWhenAttribute 通知编译器为 `obj` 参数使用的参数“不为 null”。 当该方法返回 `false` 时，参数具有在调用方法之前其曾经具有的 null 状态。

> **提示**
>
> 有一组丰富的属性可用于描述方法和属性如何影响 null 状态。 可以在语言参考文章 **可为 null 的静态分析特性**中了解这些属性。

如果要消除一个指示正在取消引用一个可能为 null 的变量的警告，有以下三种方法之一：

- 添加缺少的 null 检查。
- 在 API 上添加 null 分析特性，以影响编译器的 *null 状态* 静态分析。 这些属性会通知编译器调用方法后何时应返回“可能为 null”或“不为 null”的值或参数为。
- 将 null 包容性运算符 `!` 应用到表达式，强制性使状态成为“不为 null”。

向不可为 null 的引用分配了可能为 null 的引用

这一组警告会提醒你正在将一个类型为不可为 null 的变量分配给一个 null 状态为可能为 null 的表达式。 这些警告如下：

- **CS8597** - *抛出的值可能为 null。*
- **CS8600** - *将 null 字面量或可能为 null 的值转换为非 null 类型。*
- **CS8601** - *引用类型赋值可能为 null。*
- **CS8603** - *可能返回 null 引用。*
- **CS8604** - *形参可能传入 null 引用实参。*
- **CS8605** - *取消装箱可能为 null 的值。*
- **CS8625** - *无法将 null 字面量转换为非 null 的引用类型。*
- **CS8629** - *可为 null 的值类型可为 null。*

当你尝试将可能为 null 的表达式分配给不可 null 的变量时，编译器将发出这些警告。 例如：

C#

```
string? TryGetMessage(int id) => "";

string msg = TryGetMessage(42);  // Possible null assignment.
```

各种警告指示提供有关代码的详细信息，例如赋值、取消装箱赋值、返回语句、方法参数和引发表达式。

可以执行三种操作来消除这些警告。 一种是添加 `?` 注释，使变量成为可以为 null 的引用类型。 此更改可能会导致其他警告。 将变量从不可为 null 的引用更改为可为 null 的引用会将其默认的 null 状态从“不为 null”更改为“可能为 null”。 编译器的静态分析可能会找到一些实例，在这些实例中，会取消引用可能为 null 的变量。

另外两种操作会指示编译器，赋值的右侧不为 null。 赋值前，右侧的表达式可为 null，如以下示例中所示：

C#

```
string notNullMsg = TryGetMessage(42) ?? "Unknown message id: 42";
```

前面的示例演示如何向方法的返回值赋值。 可以对方法（或属性）进行注释，以指示方法何时返回不为 null 的值。 当输入参数不为 null 时，通常指定不为 null 的返回值 System.Diagnostics.CodeAnalysis.NotNullIfNotNullAttribute。 另一种方法是将 null 包容性运算符 `!` 添加到右侧：

C#

```
string msg = TryGetMessage(42)!;
```

如果要消除有关将可能为 null 的表达式分配给不为 null 的变量的警告，可以使用以下四种技术之一：

- 将赋值的左侧更改为可以为 null 的类型。 取消引用该变量时，此操作可能会发出新的警告。
- 在赋值之前提供 null 检查。
- 对生成赋值右侧的 API 进行注释。
- 将 null 包容性运算符添加到赋值的右侧。

不可为 null 的引用未进行初始化

这一组警告会提醒你正在将一个类型为不可为 null 的变量分配给一个 null 状态为可能为 null 的表达式。 这些警告如下：

- **CS8618** - *在退出构造函数时，不可为 null 的变量必须包含非 null 值。请考虑声明为可以为 null。*
- **CS8762** - *退出时，参数必须具有非 null 值。*

作为一个例子，请查看以下类：

C#

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

`FirstName` 和 `LastName` 均不保证初始化。 如果此代码为新代码，请考虑更改公共接口。 可以按以下方式更新上面的示例：

C#

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

    public string FirstName { get; set; }
    public string LastName { get; set; }
}
```

如果在设置名称之前需要创建 `Person` 对象，则可以使用默认的不为 null 的值对属性进行初始化：

C#

```
public class Person
{
    public string FirstName { get; set; } = string.Empty;
    public string LastName { get; set; } = string.Empty;
}
```

另一种方法可能是将这些成员更改为可以为 null 的引用类型。 如果应允许为名称使用 `null`，则可以按以下方式定义 `Person` 类：

C#

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

现有代码可能需要进行其他更改，以通知编译器这些成员的 null 语义。 你可能已创建多个构造函数，且你的类可能具有用于初始化一个或多个成员的专用帮助器方法。 可以将初始化代码移动到单个构造函数中，并确保所有构造函数都调用具有通用初始化代码的那个构造函数。 或者，可以使用 System.Diagnostics.CodeAnalysis.MemberNotNullAttribute 和 System.Diagnostics.CodeAnalysis.MemberNotNullWhenAttribute 特性。 这些特性通知编译器，在调用方法后，成员不为 null。 下面的代码就是删除两种空格的示例。 `Person` 类使用由所有其他构造函数调用的通用构造函数。 `Student` 类具有使用 System.Diagnostics.CodeAnalysis.MemberNotNullAttribute 特性进行批注的帮助程序方法：

C#

```
using System.Diagnostics.CodeAnalysis;

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

    public Person(string firstName, string lastName)
    {
        FirstName = firstName;
        LastName = lastName;
    }

    public Person() : this("John", "Doe") { }
}

public class Student : Person
{
    public string Major { get; set; }

    public Student(string firstName, string lastName, string major)
        : base(firstName, lastName)
    {
        SetMajor(major);
    }

    public Student(string firstName, string lastName) :
        base(firstName, lastName)
    {
        SetMajor();
    }

    public Student()
    {
        SetMajor();
    }

    [MemberNotNull(nameof(Major))]
    private void SetMajor(string? major = default)
    {
        Major = major ?? "Undeclared";
    }
}
```

最后，可以使用 null 包容性运算符指示某个成员在其他代码中初始化。 另一个例子是以下表示 Entity Framework Core 模型的类：

C#

```
public class TodoItem
{
    public long Id { get; set; }
    public string? Name { get; set; }
    public bool IsComplete { get; set; }
}

public class TodoContext : DbContext
{
    public TodoContext(DbContextOptions<TodoContext> options)
        : base(options)
    {
    }

    public DbSet<TodoItem> TodoItems { get; set; } = null!;
}
```

将 `DbSet` 属性初始化为 `null!`。 它指示编译器将属性设置成不为 null 的值。 事实上，基 `DbContext` 执行该集的初始化。 编译器的静态分析不会选取这个操作。 有关使用可为 null 的引用类型和 Entity Framework Core 的详细信息，请参阅在 EF Core 中使用为 null 的引用类型一文。

如果要消除一个指示未初始化不可为 null 的成员的警告，可使用以下四种方法之一：

- 更改构造函数或字段初始值设定项，以确保所有不可为 null 的成员已初始化。
- 将一个或多个成员更改为可为 null 的类型。
- 对任何帮助器方法进行注释以指示分配了哪些成员。
- 将初始值设定项添加到 `null!`，以指示该成员在其他代码中初始化。

为 null 性声明中存在不匹配

许多警告指示方法、委托或类型参数的签名之间的为 null 性不匹配。

- CS8608 - *类型中引用类型的为 Null 性与重写成员不匹配。*
- CS8609 - *返回类型中引用类型的为 Null 性与重写成员不匹配。*
- CS8610 - *参数类型中引用类型的为 Null 性与重写成员不匹配。*
- CS8611 - *参数类型中引用类型的为 Null 性与分部方法声明不匹配。*
- CS8612 - *类型中引用类型的为 Null 性与隐式实现的成员不匹配。*
- CS8613 - *返回类型中引用类型的为 Null 性与隐式实现的成员不匹配。*
- CS8614 - *参数类型中引用类型的为 Null 性与隐式实现的成员不匹配。*
- CS8615 - *类型中引用类型的为 Null 性与实现的成员不匹配。*
- CS8616 - *返回类型中引用类型的为 Null 性与实现的成员不匹配。*
- CS8617 - *参数类型中引用类型的为 Null 性与实现的成员不匹配。*
- CS8619 - *值中的引用类型的为 Null 性与目标类型不匹配。*

- CS8620 - *由于引用类型的可为 null 性差异，实参不能用于形参。*
- CS8621 - *返回类型中引用类型的为 Null 性与目标委托不匹配(可能是由于为 Null 性特性)。*
- CS8622 - *参数类型中引用类型的为 Null 性与目标委托不匹配(可能是由于为 Null 性特性)。*
- CS8624 - *由于引用类型的可为 null 性差异，实参不能用作输出。*
- CS8631 - *类型不能用作泛型类型或方法中的类型参数。类型参数的为 Null 性与约束类型不匹配。*
- CS8633 - *方法的类型参数的约束中的为 Null 性与接口方法的类型参数的约束不匹配。请考虑改用显式接口实现。*
- CS8634 - *类型不能用作泛型类型或方法中的类型参数。类型参数的为 Null 性与“class”约束不匹配。*
- CS8643 - *显式接口说明符中引用类型的 Null 性与该类型实现的接口不匹配。*
- CS8644 - *类型不实现接口成员。接口中基类型实现的引用类型的 Null 性不匹配。*
- CS8645 - *成员已列入类型的接口列表中，其中包含不同引用类型的 Null 性。*
- CS8667 - *分部方法声明在对类型参数的约束中具有不一致的为 Null 性。*
- CS8714 - *类型不能用作泛型类型或方法中的类型参数。类型参数的为 Null 性与“notnull”约束不匹配。*
- CS8764 - *返回类型的为 Null 性与重写成员不匹配(可能是由于为 Null 性特性)。*
- CS8765 - *参数类型的为 Null 性与重写成员不匹配(可能是由于为 Null 性特性)。*
- CS8766 - *返回类型中引用类型的为 Null 性与隐式实现的成员不匹配(可能是由于为 Null 性特性)。*
- CS8767 - *参数类型中引用类型的为 Null 性与隐式实现的成员不匹配(可能是由于为 Null 性特性)。*
- CS8768 - *返回类型中引用类型的为 Null 性与实现的成员不匹配(可能是由于为 Null 性特性)。*
- CS8769 - *参数类型中引用类型的为 Null 性与实现的成员不匹配(可能是由于为 Null 性特性)。*
- CS8819 - *返回类型中引用类型的为 Null 性与分部方法声明不匹配。*

以下代码演示了 CS8764：

C#

```
public class B
{
    public virtual string GetMessage(string id) => string.Empty;
}
public class D : B
{
    public override string? GetMessage(string? id) => default;
}
```

前面的示例显示了基类中的 `virtual` 方法和 `override` 具有不同的为 null 性。 基类返回不可为 null 的字符串，但派生的类返回一个可以为 null 的字符串。 如果 `string` 和 `string?` 反转，这是允许的情况，因为派生的类更严格。 同样，参数声明应匹配。 重写方法中的参数可允许 null，即使基类不允许。

其他情况下，可能会产生这些警告。 接口方法声明和该方法的实现中可能存在不匹配的情况。 或者委托类型和该委托的表达式可能不同。 类型参数和类型自变量的为 null 性可能不同。

若要消除这些警告，请更新相应的声明。

代码与特性声明不匹配

前面的部分讨论了如何使用可为 null 的静态分析的特性来通知编译器你的代码的 null 语义。 如果代码不遵循该特性的承诺，编译器会发出警告：

- **CS8607** - *可能的 null 值不能用于标有 `[NotNull]` 或 `[DisallowNull]` 的类型*
- **CS8763** - *不能返回标记为 `[DoesNotReturn]` 的方法。*
- **CS8770** - *方法缺少 `[DoesNotReturn]` 注释，无法匹配已实现的或被替代的成员。*
- **CS8774** - *退出时，成员必须具有非 null 值。*
- **CS8775** - *退出时，成员必须具有非 null 值。*
- **CS8776** - *不能在此特性中使用此成员。*
- **CS8777** - *退出时，参数必须具有非 null 值。*
- **CS8824** - *退出时参数必须具有非 null 值，因为参数是非 null。*
- **CS8825** - *由于参数为非 null，因此返回值必须为非 null。*

请考虑以下方法：

```C#
public bool TryGetMessage(int id, [NotNullWhen(true)] out string? message)
{
    message = null;
    return true;

}
```

编译器生成警告，因为分配了 `message` *并且* `null` 方法返回 `true`。 特性 `NotNullWhen` 指示不应发生这种情况。

若要消除这些警告，请更新代码，使其符合有关已应用的特性的预期。 可以更改特性或算法。

详尽的 switch 表达式

switch 表达式必须是详尽的，这意味着必须处理所有输入值。 即使对于不可为 null 的引用类型，也必须考虑 `null` 值。 未处理 null 值时，编译器会发出警告：

- **CS8655** - *Switch 表达式不会处理某些为 null 的输入(它并非详尽无遗)。*
- **CS8847** - *Switch 表达式不会处理一些 null 输入(它不是穷举)。但是，带有"when"子句的模式可能成功匹配此值。*

以下示例代码演示了此条件：

C#

```
int AsScale(string status) =>
    status switch
    {
        "Red" => 0,
        "Yellow" => 5,
        "Green" => 10,
        { } => -1
    };
```

输入表达式是 `string`，而不是 `string?`。 编译器仍会生成此警告。 `{ }` 模式处理所有非 null 值，但不匹配 `null`。 若要解决这些错误，你可以添加显式 `null` 大小写，也可以将 `{ }` 替换为 `_`（丢弃）模式。 丢弃模式匹配 null 以及其他任意值。

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

Open a documentation issue

Provide product feedback

模式匹配错误和警告

项目 • 2023/12/22

编译器为无效模式匹配表达式生成以下错误：

- CS9134：switch 表达式臂不以“case”关键字开头。
- CS9135：应为类型的常量值

编译器为不完整的模式匹配表达式生成以下警告：

- **CS8509**：switch 表达式不会处理属于其输入类型的所有可能值（它不是穷尽的）。 *例如，未涵盖模式“...”。*

编译器可生成 CS9134 来帮助将 switch 语句转换为 switch 表达式。 switch 表达式中不使用 `case` 关键字。 如果在 switch 表达式中具有 `case` 关键字，则必须将其移除：

C#

```
var answer = x switch
{
    // CS9134: remove the keyword "case" from each switch arm:
    case 0 => false,
    case 1 => true,
}
```

当模式不是常量值时，编译器将生成 CS9135。 无法创建与变量匹配的模式匹配表达式。

以下代码片段生成 CS8509：

C#

```
// CS8509.cs
enum EnumValues
{
    Value1,
    Value2,
    Value3
}

void Method(EnumValues enumValues)
{
    var result = enumValues switch
    {
        EnumValues.Value1 => 1,
        EnumValues.Value2 => 2,
    };
}
```

若要解决此警告，请添加涵盖所有可能输入值的开关臂。 例如：

C#

```
enum EnumValues
{
    Value1,
    Value2,
    Value3
}

void Method(EnumValues enumValues)
{
    var result = enumValues switch
    {
        EnumValues.Value1 => 1,
        EnumValues.Value2 => 2,
        EnumValues.Value3 => 3,
        _ => throw new ArgumentException("Input isn't a valid enum value",
nameof(enumValues)),
        };
}
```

`_` 模式匹配所有剩余值。 `_` 模式的一种方案与无效值匹配，如前面的示例所示。

有关详细信息，请参阅 Switch。

在 GitHub 上与我们协作

可以在 GitHub 上找到此内容的源，还可以在其中创建和查看问题和拉取请求。 有关详细信息，请参阅参与者指南。

.NET

.NET 反馈

.NET 是一个开放源代码项目。 选择一个链接以提供反馈：

提出文档问题

提供产品反馈

解决数组和集合声明及初始化表达式中的错误和警告

项目 • 2024/10/15

本文介绍以下编译器错误：

- CS0022：[] 内索引数错误，应为“number”
- CS0178：级别说明符无效：应为“`,`”或“`]`”
- CS0248：无法创建大小为负值的数组
- CS0270：不能在变量声明中指定数组大小（请尝试使用“`new`”表达式进行初始化）
- CS0611：数组元素不能为 type
- CS0623：数组初始值设定项只能在变量或字段初始化表达式中使用。*请尝试改用新的表达式。*
- CS0650：数组声明符错误：要声明托管数组，级别说明符应位于变量的标识符之前。*若要声明固定大小的缓冲区字段，请在字段类型之前使用 fixed 关键字。*
- CS0719：数组元素不能是静态类型
- CS0747：初始化表达式成员声明符无效。
- CS0820：无法将数组初始值设定项分配给隐式类型化局部变量
- CS0826：找不到隐式类型数组的最佳类型。
- CS0846：应为嵌套数组初始值设定项
- **CS1063**: *与集合初始值设定项元素最匹配的重载 Add 方法已过时。*
- CS1552：数组类型说明符 `[]` 必须出现在参数名称之前
- CS1586：数组创建必须具有数组大小或数组初始值设定项
- CS1920：元素初始值设定项不能为空。
- CS1921：最佳重载的方法匹配具有初始化表达式元素而言错误的签名。可初始化的 `Add` 必须是可访问的实例方法。
- CS1925：无法使用集合初始值设定项初始化类型为“type”的对象。
- **CS1950**：*与集合初始值最佳匹配的重载 Add 方法有一些无效参数。*
- CS1954：不能使用集合初始值设定项元素的最佳重载的方法匹配。集合初始值设定项“`Add`”方法不能有 `ref` 或 `out` 参数。
- CS9174：无法使用集合文本初始化类型，因为该类型不可构造。
- CS9176：集合文本没有目标类型。
- CS9185：CollectionBuilderAttribute 生成器类型必须是非泛型类或结构。
- CS9186：CollectionBuilderAttribute 方法名称无效。
- CS9187：找不到具有预期签名的可访问方法：具有类型为 `ReadOnlySpan<T>` 的单个参数的静态方法；以及正确的返回类型
- CS9188：类型具有 CollectionBuilderAttribute，但没有元素类型。
- CS9203：此类型的集合表达式不能在此上下文中使用，因为它可能在当前范围之外公开。

- CS9210：此版本的 System.Collections.Immutable.ImmutableArray<T> 不能与集合表达式一起使用。

此外，本文还介绍了以下警告：

- **CS1062**：*与集合初始值设定项元素最匹配的重载 Add 方法已过时。*
- **CS1064**：*与集合初始值设定项元素最匹配的重载 Add 方法已过时。*
- CS3007：只是未命名数组类型不同的重载方法"method"不符合 CLS
- CS3016：作为特性参数的数组不符合 CLS
- CS0251：使用负索引编制数组的索引（数组索引始终从零开始）
- CS9208：集合表达式可能会导致意外的堆分配。*请考虑显式创建数组，然后转换为最终类型以将分配变为显式。*
- CS9209：由于使用"`..`"分布，集合表达式可能会导致意外的堆分配。*请考虑显式创建数组，然后转换为最终类型以将分配变为显式。*

可在以下文章中了解有关数组、集合初始值设定项和集合表达式的详细信息：

- 数组
- 对象和集合初始值设定项
- 集合表达式。

集合初始值设定项无效

以下错误指示编译器为集合初始值设定项生成的代码无效：

- CS0747：初始值设定项成员声明符无效。
- **CS1063**：*与集合初始值设定项元素最匹配的重载 Add 方法已过时。*
- CS1920：元素初始值设定项不能为空。
- CS1921：最佳重载的方法匹配具有对初始化表达式元素而言错误的签名。可初始化的 `Add` 必须是可访问的实例方法。
- **CS1950**：*与集合初始值最佳匹配的重载 Add 方法有一些无效参数。*
- CS1954：不能使用集合初始值设定项元素的最佳重载的方法匹配。集合初始值设定项"`Add`"方法不能有 `ref` 或 `out` 参数。
- CS9174：无法使用集合文本初始化类型，因为该类型不可构造。
- CS9176：集合文本没有目标类型。
- CS9203：此类型的集合表达式不能在此上下文中使用，因为它可能在当前范围之外公开。
- CS9210：此版本的 System.Collections.Immutable.ImmutableArray<T> 不能与集合表达式一起使用。

编译器还可能生成以下警告：

- **CS1062**：*与集合初始值设定项元素最匹配的重载 Add 方法已过时。*

- **CS1064**：*与集合初始值设定项元素最匹配的重载 Add 方法已过时。*
- CS9208：集合表达式可能会导致意外的堆分配。*请考虑显式创建数组，然后转换为最终类型以将分配变为显式。*
- CS9209：由于使用“`..`”分布，集合表达式可能会导致意外的堆分配。*请考虑显式创建数组，然后转换为最终类型以将分配变为显式。*

这些错误都表示编译器为集合初始值设定项生成的代码无效。 检查下列项目：

- 集合初始值设定项包含一个元素序列。 不能将在集合实例上设置属性与在同一初始值设定项中添加元素的操作混合使用。
- 包含大括号（`{` 和 `}`）的集合初始值设定项不能为空。
- 该类必须实现 IEnumerable 并具有公共 `Add` 方法。
- 符合条件的 `Add` 方法必须可供访问，并采用与集合元素类型相同的一个参数。 参数不能包含 `ref` 或 `out` 修饰符。
- 重载解析必须选取一个 `Add` 方法作为更好的匹配项。 不可能有多个匹配的方法同样良好。
- 集合表达式可以初始化集合类型的显式类型化变量。 如果变量不是集合或数组类型，或者隐式类型化（使用 `var`），则不能使用集合初始值设定项。
- 不能使用可能违反 ref 安全性的集合表达式初始化类型 `ref struct`（如 System.Span<T>）。
- 集合表达式无法使用当前版本正确初始化 System.Collections.Immutable.ImmutableArray<T>。 使用不同版本的运行时，或更改初始化表达式。

警告指示集合表达式（包括任何分散元素）可能会分配内存。 创建不同的存储并进行转换可能更高效。

数组初始值设定项无效

- CS0623：数组初始值设定项只能在变量或字段初始化表达式中使用。*请尝试改用新的表达式。*
- CS0846：应为嵌套数组初始值设定项
- CS1925：无法使用集合初始值设定项初始化类型为“type”的对象。

这些错误指示创建的初始值设定项无效。 原因可能是围绕一个或多个元素或子数组的大括号 `{` 和 `}` 不均衡。 确保初始化表达式与交错数组初始化中的数组数匹配，并且大括号是均衡的。

无效的数组元素访问

- CS0022：[] 内索引数错误，应为“number”

- CS0251：使用负索引编制数组的索引（数组索引始终从零开始）

通过指定数组中声明的每个轴的索引来访问数组的元素。 索引在数组名称之后的 `[` 和 `]` 之间。 数组索引有两个规则：

1. 必须指定与数组声明中使用的索引数相同。 如果数组有一个维度，则必须指定一个索引。 如果数组有三个维度，则必须指定三个索引。
2. 所有索引必须是非负整数。

数组排名无效

- CS0178：级别说明符无效：应为“`,`”或“`]`”
- CS0650：数组声明符错误：要声明托管数组，级别说明符应位于变量的标识符之前。*若要声明固定大小的缓冲区字段，请在字段类型之前使用 fixed 关键字。*
- CS1552：数组类型说明符 `[]` 必须出现在参数名称之前

数组声明由以下标记组成，顺序如下：

1. 数组元素的类型。 例如，`int`、`string` 或 `SomeClassType`。
2. 数组括号，可以选择包括逗号来表示多维。
3. 变量名称。

当数组初始化指定数组维度时，可以指定以下属性：

- 大括号中的许多元素（`{` 和 `}`）
- 空括号
- 括在括号中的一个或多个逗号

例如，下面是有效的数组声明：

```C#
int[] anArrayOfIntegers;
string[,,] threeDimensionalStringArray;
```

有关详细信息，请参阅 C# 规范（C# 语言规范）中有关数组初始值设定项的部分。

数组长度无效

- CS0248：无法创建大小为负值的数组
- CS0270：不能在变量声明中指定数组大小（请尝试使用“new”表达式进行初始化）
- CS1586：数组创建必须具有数组大小或数组初始值设定项

数组的每个维度的长度必须指定为数组初始化的一部分，而不是其声明。 每个维度的长度必须为正。 可以使用 `new` 表达式分配数组或使用数组初始值设定项来分配所有元素来指定长度。 以下示例演示了这两个机制：

C#

```
int[] fiveIntegers = new int[5];
string[] weekDays = { "Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat" };
```

元素类型无效

- CS0611：数组元素不能为“type”类型
- CS0719：数组元素不能是静态类型
- CS0820：无法将数组初始值设定项分配给隐式类型化局部变量
- CS0826：找不到隐式类型数组的最佳类型。

有些类型不能用作数组类。 这些类型包括 System.TypedReference 和 System.ArgIterator。 数组的类型不能是 `static` 类，因为无法创建 `static` 类的实例。

可以将数组声明为隐式类型本地变量。 必须使用 `new` 表达式初始化数组。 此外，数组初始值设定项中的所有元素都必须具有最佳常见类型。 下列示例演示如何声明隐式类型化数组：

C#

```
var implicitType = new[] { 1, 2, 3 };

char c = 'c';
short s1 = 0;
short s2 = -0;
short s3 = 1;
short s4 = -1;

// common type is "int"
var commonType = new[] { s1, s2, s3, s4, c, 1 };
```

可以使用以下任一技术确保最佳常见类型：

- 为数组指定显式类型。
- 为所有数组元素指定相同的类型。
- 为可能会导致问题的那些元素提供显式强制转换。

集合生成器无效

以下错误指示集合生成器类型中的错误：

- CS9185：`CollectionBuilderAttribute` 生成器类型必须是非泛型类或结构。
- CS9186：`CollectionBuilderAttribute` 方法名称无效。
- CS9187：找不到具有预期签名的可访问方法：具有类型为 `ReadOnlySpan<T>` 的单个参数的静态方法；以及正确的返回类型。
- CS9188：类型具有 `CollectionBuilderAttribute`，但不具有元素类型。

这些错误指示集合生成器类型需要修改。 请记住以下规则：

- 集合类型必须具有迭代类型。 换句话说，可以将 foreach 类型为集合。
- 集合生成器类型不能是泛型类型。
- 在 System.Runtime.CompilerServices.CollectionBuilderAttribute 上指定的方法名称必须是有效的方法名称。 例如，它不能是终结器，也不能是包含无效标识符字符的其他名称。
- 集合生成器方法必须是可访问的静态方法。 它必须返回集合类型，并且必须采用 `ReadOnlySpan<T>` 参数，其中 `T` 与集合的元素类型匹配。

公共语言规范警告

- CS3007：只是未命名数组类型不同的重载方法“method”不符合 CLS
- CS3016：作为特性参数的数组不符合 CLS

如果你有一个接受交错数组的重载方法，并且方法签名的唯一区别是数组的元素类型，则将发生 CS3007 错误。 若要避免此错误，请考虑使用矩形数组而不是交错数组，或者，如果不需要 CLS 合规性，请移除 CLSCompliantAttribute 特性。 有关 CLS 符合性的详细信息，请参阅语言独立性和与语言无关的组件。

CS3016 指示将数组传递给属性不符合公共语言规范 (CLS)。 有关 CLS 符合性的详细信息，请参阅语言独立性和与语言无关的组件。

解决内联数组声明中的错误和警告

项目 • 2024/11/08

本文介绍了以下编译器错误和警告：

- **CS9164**：*无法将表达式转换为 `Span<T>`，因为它不是可赋值变量*
- **CS9165**：*无法将表达式转换为 `ReadOnlySpan<T>`，因为它可能无法通过引用传递或返回*
- CS9166：索引超出内联数组的边界
- CS9167：内联数组长度必须大于 0。
- CS9168：内联数组结构不得具有显式布局。
- CS9169：内联数组结构必须且只能声明一个实例字段，该字段不得为 ref 字段。
- **CS9172**：*只能使用可隐式转换为 `int`、`System.Index` 或 `System.Range` 的单个参数访问内联数组类型的元素。*
- CS9173：内联数组访问可能没有已命名的参数说明符
- **CS9180**：*内联数组元素字段不能声明为必需、只读、易失或固定大小缓冲区。*
- **CS9181**：*内联数组索引器不会用于元素访问表达式。*
- **CS9182**：*内联数组"Slice"方法不会用于元素访问表达式。*
- **CS9183**：*内联数组转换运算符不会用于从声明类型的表达式转换。*
- **CS9184**：*元素字段是"`ref`"字段，或者作为类型参数具有无效类型的内联数组类型不支持"内联数组"语言功能。*
- **CS9189**：不支持类型内联数组中的语句 `foreach`
- **CS9259**：*System.Runtime.CompilerServices.InlineArrayAttribute属性不能应用于记录结构。*

内联数组声明

将内联数组声明为具有单个字段的 `struct` 类型，以及指定数组长度的属性。 编译器为无效的内联数组声明生成以下错误：

- CS9167：内联数组长度必须大于 0。
- CS9168：内联数组结构不得具有显式布局。
- CS9169：内联数组结构必须且只能声明一个实例字段，并且该字段不得为 ref 字段。
- CS9180：内联数组元素字段不能声明为必需、只读、易失或固定大小缓冲区。
- **CS9184**：*元素字段是"`ref`"字段，或者作为类型参数具有无效类型的内联数组类型不支持"内联数组"语言功能。*
- **CS9259**：*System.Runtime.CompilerServices.InlineArrayAttribute属性不能应用于记录结构。*

若要修复这些数组，请确保满足以下条件：

- System.Runtime.CompilerServices.InlineArrayAttribute 的参数是一个正整数。
- 封闭 `struct` 没有指定任何显式布局。
- 封闭 `struct` 具有单个实例字段，且该实例字段不是 `ref` 字段。
- 单个实例字段不是固定大小缓冲区。
- 单个实例字段不包括 `required`、`volatile` 或 `readonly` 修饰符。
- `record`从内联数组声明中删除修饰符。

元素访问

内联数组的元素访问方式与任何数组相同。 编译器会从不正确的元素访问中发出以下错误：

- CS9166：索引超出内联数组的边界
- **CS9172**：*只能使用可隐式转换为 `int`、`System.Index` 或 `System.Range` 的单个参数访问内联数组类型的元素。*
- CS9173：内联数组访问可能没有命名参数说明符
- **CS9189**：不支持类型内联数组中的语句 *`foreach`*

此外，编译器会在声明索引器时发出以下警告：

- CS9181：内联数组索引器不会用于元素访问表达式。

内联缓冲区生成的代码可直接访问缓冲区内存，从而绕过任何声明的索引器。 内联数组不能与 `foreach` 语句一起使用。

索引器的参数必须是：

- 以下三种类型之一：`int`、`System.Index` 或 `System.Range`。
- 不能是命名参数。 编译器生成元素访问器。 参数没有名称，因此无法使用命名参数。
- 在数组的边界内。 与所有 .NET 数组一样，内联数组元素访问也会进行边界检查。 索引必须位于内联数组的边界内。

转换为 Span

通常使用 System.Span<T> 或 System.ReadOnlySpan<T> 来处理内联数组。 编译器针对无效转换生成以下错误：

- **CS9164**：*无法将表达式转换为 `Span<T>`，因为它不是可赋值变量*

- **CS9165**：*无法将表达式转换为 `ReadOnlySpan<T>`，因为它可能无法通过引用传递或返回*

编译器可生成直接访问内联缓冲区内存的代码。 因此，从未调用过某些成员。 如果编写了一个从未调用的成员，编译器将生成以下警告：

- CS9182：内联数组“Slice”方法不会用于元素访问表达式。
- CS9183：内联数组转换运算符将不用于从声明类型的表达式转换。

内联数组可以隐式转换为 `Span<T>` 或 `ReadOnlySpan<T>`，以将其传递给方法。 编译器对这些转换强制实施限制：

- 内联数组必须可写入，以便转换为 `Span<T>`。 如果数组是只读数组，则无法将其转换为可写入的 `Span<T>`。 可以改用 `ReadOnlySpan<T>`。
- 内联数组的*安全上下文*必须至少与 `Span<T>` 或 `ReadOnlySpan<T>` 的*安全上下文*一样宽，才能成功进行转换。 必须限制 Span 的上下文，或扩展内联数组的范围。

此外，编译器永远不会在内联缓冲区中生成对 `Slice` 方法的调用。 不会调用将内联缓冲区转换为 `Span` 或 `ReadOnlySpan` 的转换运算符。 编译器会生成代码，以直接从内存缓冲区创建 System.Span<T> 或 System.ReadOnlySpan<T>。

使用 lambda 表达式和匿名函数时的错误和警告

项目 • 2023/12/25

有几个与声明和使用 lambda 表达式相关的错误：

- CS0748：Lambda 参数的用法不一致；参数类型必须全部为显式或全部为隐式。
- CS1621：不能在匿名方法或 Lambda 表达式内使用 yield 语句。
- CS1628：不能在匿名方法、Lambda 表达式或查询表达式中使用 `in` `ref` 或 `out` 参数。
- CS1632：控制不得离开匿名方法或 Lambda 表达式主体。
- CS1673：结构内部的匿名方法、Lambda 表达式和查询表达式无法访问 'this' 的实例成员。
- CS1686：不能采用局部变量或其成员的地址，也不能在匿名方法或 Lambda 表达式内部使用它们。
- CS1706：表达式不能包含匿名方法或 Lambda 表达式。
- CS8030：转换为 void 返回委托的匿名函数不能返回值。
- CS8175：不能在匿名方法、Lambda 表达式或查询表达式内使用 ref 局部变量。
- CS8916：Lambda 表达式上的属性需要拥有带圆括号的参数列表。
- CS8971：InterpolatedStringHandlerArgument 在应用于 Lambda 参数时不起任何作用，并将在调用站点被忽略。
- CS8975：上下文关键字 `var` 不能用作显式 Lambda 返回类型。
- CS9098：隐式键入的 Lambda 参数“...”不能具有默认值。

此外，还有几个与声明和使用 Lambda 表达式相关的警告：

- CS8971：InterpolatedStringHandlerArgument 在应用于 Lambda 参数时不起任何作用，并将在调用站点被忽略。
- CS9099：默认参数值在目标委托类型中不匹配。
- CS9100：参数在 Lambda 中具有参数修饰符，但在目标委托类型中没有。

Lambda 表达式中的语法限制

Lambda 表达式和匿名方法中禁止某些 C# 语法。 在 Lambda 表达式中使用无效构造会导致以下错误：

- CS1621：不能在匿名方法或 Lambda 表达式内使用 `yield` 语句。
- CS1628：不能在匿名方法、Lambda 表达式或查询表达式中使用 `in`、 `ref` 或 `out` 参数。
- CS1632：控制不得离开匿名方法或 Lambda 表达式主体。

- CS1673：结构内部的匿名方法、Lambda 表达式和查询表达式无法访问 `this` 的实例成员。
- CS1686：不能采用局部变量或其成员的地址，也不能在匿名方法或 Lambda 表达式内部使用它们。
- CS8175：不能在匿名方法、Lambda 表达式或查询表达式内使用 ref 局部变量。

Lambda 表达式中不允许以下所有构造：

- `yield` 语句（`yield return` 或 `yield break`）
- 调用具有 `in`、 `ref` 或 `out` 参数的方法
- `ref` 局部变量
- `break`、`goto` 和 `continue` 语句
- `this` 是 `struct` 类型时的 `this` 访问
- 另一个表达式中的匿名方法或 Lambda 表达式，例如 Attribute 构造函数。

不能在 Lambda 表达式或匿名方法中使用上述任何构造。 本地函数中允许使用多个构造。

此外，当应用于 lambda 参数时，将忽略内插字符串处理程序类型。 如果使用，会看到以下警告：

- CS8971：InterpolatedStringHandlerArgument 在应用于 Lambda 参数时不起任何作用，并将在调用站点被忽略。

Lambda 表达式参数和返回

这些错误表示参数声明存在问题：

- CS0748：Lambda 参数的用法不一致；参数类型必须全部为显式或全部为隐式。
- CS9098：隐式键入的 Lambda 参数“...”不能具有默认值。
- CS8030：转换为 void 返回委托的匿名函数不能返回值。
- CS8916：Lambda 表达式上的属性需要拥有带圆括号的参数列表。
- CS8975：上下文关键字“var”不能用作显式 Lambda 返回类型。

Lambda 表达式参数必须遵循以下规则：

- 当 Lambda 表达式具有多个参数时，所有参数都必须显式类型化，或者所有参数都必须隐式类型化。
- 所有具有默认值的 lambda 参数都必须显式类型化。
- 如果属性应用于任何参数，则参数列表必须括在括号中。

Lambda 表达式的返回类型必须遵循以下规则：

- 返回任何值的 Lambda 表达式不能转换为 `void` 返回委托，例如 `Action`。
- 返回类型为推断类型或显式类型。 不允许使用关键字 `var` 声明的返回类型。

Lambda 表达式委托类型

- CS9099：警告：默认参数值在目标委托类型中不匹配。

声明默认值或使用 lambda 表达式参数添加 `params` 修饰符时，委托类型不是 `Func` 或 `Action` 类型之一。 而是包含默认参数值或 `params` 修饰符的自定义类型。 以下代码生成警告，因为它将具有默认参数的 Lambda 表达式分配给 `Action` 类型：

C#

```
Action<int> a1 = (int i = 2) => { };
Action<string[]> a3 = (params string[] s) => { };
```

若要修复此错误，请删除默认参数或使用委托类型的隐式类型变量：

C#

```
Action<int> a1 = (int i) => { };
var a2 = (int i = 2) => { };
Action<string[]> a3 = (string[] s) => { };
var a4 = (params string[] s) => { };
```


在 GitHub 上与我们协作

可以在 GitHub 上找到此内容的源，还可以在其中创建和查看问题和拉取请求。 有关详细信息，请参阅参与者指南。

.NET

.NET 反馈

.NET 是一个开放源代码项目。 选择一个链接以提供反馈：

提出文档问题

提供产品反馈

解决影响重载解析的错误和警告。

项目 • 2024/10/15

本文介绍以下编译器错误：

- CS0034 - *运算符对于“`type1`”和“`type2`”类型的操作数具有二义性*
- CS0035 - *运算符对于“type”类型的操作数具有二义性*
- CS0111 - *类型已定义了一个名为“name”的具有相同参数类型的成员*
- CS0121 - *以下方法或属性之间的调用不明确*
- CS0457 - *当从“type1”转换为“type2”时，用户定义的转换不明确。*
- CS0663 - *无法定义仅在 `ref` 和 `out` 上存在不同的重载方法。*
- CS1007 - *属性访问器已经定义*
- CS1019 - *应输入可重载的一元运算符*
- CS1020 - *应输入可重载的二元运算符*
- CS1501 - *方法“method”的重载不带有“number”个参数*
- CS1534 - *重载的二元运算符“operator”采用两个参数*
- CS1535 - *重载的一元运算符“operator”采用一个参数*
- CS1928 - *“Type”不包含“method”的定义，并且最佳扩展方法重载“method”的某些参数无效。*
- CS1929 - *“TypeA”不包含“method”的定义，最佳扩展方法重载“TypeB.method”需要“TypeC”类型的接收器*
- CS8315 - *运算符对于操作数具有二义性*
- CS9261 - *不能对替代成员使用“`OverloadResolutionPriorityAttribute`”。*
- CS9262 - *不能对此成员使用“`OverloadResolutionPriorityAttribute`”。*

此外，还有以下编译器警告：

- CS3006 - *仅 `ref` 或 `out` 或者仅数组秩不同的重载方法“method”不符合 CLS*

重载定义不正确

- CS0663 - *无法定义仅在 `ref` 和 `out` 上存在不同的重载方法。*
- CS1019 - *应输入可重载的一元运算符*
- CS1020 - *应输入可重载的二元运算符*
- CS1534 - *重载的二元运算符“operator”采用两个参数*
- CS1535 - *重载的一元运算符“operator”采用一个参数*

此外，还有以下编译器警告：

- CS3006 - *仅 `ref` 或 `out` 或者仅数组秩不同的重载方法“method”不符合 CLS*

在类中创建重载运算符时，签名必须与该运算符所需的参数数目匹配。 运算符定义中的参数数目错误。

此外，重载运算符必须使用定义的运算符名称。 唯一的例外是创建转换运算符时，其中运算符名称与转换的返回类型匹配。

定义的重载重复

- **CS0111** - *类型已定义了一个名为"name"的具有相同参数类型的成员*
- **CS1007** - *属性访问器已经定义*

这些错误指示你创建了不同的重载。 常见原因包括：

- 匹配的参数类型具有标识转换。 例如 `M(dynamic)` 或 `M(object)`，或者 `M2(string)` 和 `M2(string?)`。
- 匹配的参数类型仅在引用修饰符方面不同。 例如，`M(ref int)` 和 `M(out int)`。
- 你同时包含了属性的 `init` 和 `set` 访问器。

未找到重载

- **CS1501** - *方法"method"的重载不带有"number"个参数*
- **CS1928** - *"Type"不包含"method"的定义，并且最佳扩展方法重载"method"的某些参数无效。*
- **CS1929** - *"TypeA"不包含"method"的定义，最佳扩展方法重载"TypeB.method"需要"TypeC"类型的接收器*

代码调用存在此名称的方法，但某些参数不正确，或者使用了错误的参数数量。 如果你要尝试调用扩展方法，则包含类可能位于未导入的命名空间中。

> **备注**
>
> 错误：最新编译器未使用 `CS1928`。 较新的编译器专门使用 `CS1929`。

不明确的重载

- **CS0034** - *运算符对于"`type1`"和"`type2`"类型的操作数具有二义性*
- **CS0035** - *运算符对于"type"类型的操作数具有二义性*
- **CS0121** - *以下方法或属性之间的调用不明确*
- **CS0457** - *当从"type1"转换为"type2"时，用户定义的转换不明确*
- **CS8315** - *运算符对于操作数具有二义性*

这些错误表明没有比其他更好的重载。 这些错误可能因以下多种原因所致：

- 对于每个参数，其他参数更好。 一个可能是完全匹配项，另一个需要转换：

```
C#

  void M(int a, double b) {    }
  void M(double a, int b) {}

  M(1, 1); // ambiguous
```

- 多个用户定义的转换运算符是候选项。 可以在源类型、基类型、目标类型或其基类型中声明转换运算符。 如果在多个项中定义了转换运算符，则编译器无法选取。
- 多个转换运算符可以执行兼容的转换。
- 一个或多个参数使用 `default`（不是 `default(T)`），并且可以选择多个类型。

在大多数情况下，添加显式强制转换可以指定应选择哪个重载。

重载解析优先级

- **CS9261** - *不能对替代成员使用“`OverloadResolutionPriorityAttribute`”。*
- **CS9262** - *不能对此成员使用“`OverloadResolutionPriorityAttribute`”。*

你的代码违反了使用 OverloadResolutionPriorityAttribute 支持一个重载而不是另一个重载的规则。 你不能将 `OverloadResolutionPriorityAttribute` 应用于以下方法类型：

- 非索引器属性
- 属性、索引器或事件访问器
- 转换运算符
- Lambda
- 本地函数
- 终结器
- 静态构造函数

此外，不能将 `OverloadResolutionPriorityAttribute` 应用于 `virtual` 或 `abstract` 成员的 `override`。 编译器使用基类型声明中的值。

解决表达式树中禁止的表达式生成的错误和警告

项目 • 2023/09/11

本文介绍以下编译器错误：

- CS0765 - 不能在表达式树中使用只有定义声明的分部方法或已移除的条件方法。
- CS0831 - 表达式树不能包含基访问。
- CS0832 - 表达式树不能包含赋值运算符。
- CS0834 - 无法将具有语句体的 lambda 表达式转换为表达式树。
- CS0835 - 无法将 lambda 转换为其类型实参“type”不是委托类型的表达式树。
- CS0838 - 表达式树不能包含多维数组初始值设定项。
- CS0845 - 表达式树 lambda 不能包含左侧为 null 或 default 字面量的合并运算符。
- CS0853 - 表达式树不能包含命名参数规范。
- CS0854 - 表达式树不能包含使用可选参数的调用。
- CS0855 - 表达式树不能包含索引属性。
- CS1944 - 表达式树不能包含不安全的指针操作。
- CS1945 - 表达式树不能包含匿名方法表达式。
- CS1946 - 无法将匿名方法表达式转换为表达式树。
- CS1951 - 表达式树 lambda 不能包含 `ref`、`in` 或 `out` 参数。
- CS1952 - 表达式树 lambda 不能包含具有变量参数的方法。
- CS1963 - 表达式树不能包含动态操作。
- CS1989 - 异步 lambda 表达式无法转换为表达式树。
- CS2037 - 表达式树 lambda 不能包含参数中省略 ref 的 COM 调用。
- CS7053 - 表达式树可能不包含“feature”。
- CS8072 - 表达式树 lambda 不能包含空传播运算符。
- CS8074 - 表达式树 lambda 不能包含一个字典初始值设定项。
- CS8075 - 表达式 Lambda 中的集合初始值设定项不支持扩展 `Add` 方法。
- CS8110 - 表达式树不能包含对本地函数的引用。
- CS8122 - 表达式树不能包含“`is`”模式匹配运算符。
- CS8143 - 表达式树不能包含元组字面量。
- CS8144 - 表达式树不能包含元组转换。
- CS8153 - 表达式树 lambda 不能包含对按引用返回的方法、属性或索引器的调用。
- CS8155 - 引用返回的 Lambda 表达式不能转换为表达式树。
- CS8188 - 表达式树不能包含 throw 表达式。
- CS8198 - 表达式树不能包含输出参数变量声明。
- CS8207 - 表达式树不能包含放弃。
- CS8382 - 表达式树不能包含元组 `==` 或 `!=` 运算符。
- CS8514 - 表达式树不能包含 switch 表达式。

- CS8640 - 表达式树不能包含 ref 结构或受限类型的值。
- CS8642 - 表达式树不能包含 null 合并赋值。
- CS8790 - 表达式树不能包含模式 System.Index 或 System.Range 索引器访问。
- CS8791 - 表达式树不能包含 from-end 索引（“`^`”）表达式。
- CS8792 - 表达式树不能包含范围（“`..`”）表达式。
- CS8810 - 不能在表达式树中使用方法组上的“`&`”。
- CS8849 - 表达式树不能包含 `with` 表达式。
- CS8927 - 表达式树不能包含静态虚拟或抽象接口成员的访问权限。
- CS8952 - 表达式树不能包含内插字符串处理程序转换。
- CS8972 - 无法将具有属性的 lambda 表达式转换为表达式树。
- CS9170 - 表达式树不能包含内联数组访问或转换。
- CS9175 - 表达式树不能包含集合表达式。

表达式树限制

前面列表中的所有错误都表明你使用了表达式树中不允许的 C# 表达式类型。 大多数情况下，禁止的表达式代表 C# 3.0 之后引入的语法。 这些表达式被禁止是因为允许它们会在分析表达式树的所有库中造成中断性变更。 如果允许较新的构造，则所有库都需要增强以分析新的 C# 表达式。

禁止使用以下表达式：

- 调用没有实现声明的分部方法。
- 调用已删除的条件方法。
- 调用本地函数。
- 不允许使用 `async` lambda 表达式。
- 使用 base 访问权限直接调用基类中声明的虚拟方法。
- 赋值操作。
- 不允许语句 lambda。
- 多维数组初始值设定项。 必须改为在表达式树外部创建并初始化一个多维数组。
- 不允许 dynamic 操作。
- 不允许模式匹配表达式。
- 不允许使用元组字面量和许多元组操作，例如比较是否相等。
- 不允许 throw 表达式。
- 放弃 (`_`) 声明。
- 不允许使用索引和范围运算符。
- 不允许使用 with 表达式进行非破坏性修改。
- 不能声明或访问内联数组。
- 不能包含集合表达式。
- 不允许使用 null 传播和 null 合并运算符。

- 不允许使用 ref struct 类型，例如 System.Span<T> 和 System.ReadOnlySpan<T>。
- 不允许使用 `in`、`out` 和 `ref` 参数（包括 `out` 变量声明）。
- 不允许 `ref` 返回 。
- 不允许调用由 `ref` 返回的方法。
- 无法访问静态抽象接口成员。

其他限制包括：

- 属性不能应用于 lambda 表达式、其参数或返回。
- lambda 表达式必须可转换为从 System.Linq.Expressions.Expression（其类型参数是委托类型）派生的类型。
- 命名参数和可选参数受到限制。 该表达式无法调用指定命名参数的方法，也不能使用可选参数的默认值。
- 不允许使用字典初始值设定项。 两者都不是扩展 `Add` 方法。
- 目标表达式必须是 Lambda 表达式。 不允许使用常量和变量，但可以使用返回常量或变量的 Lambda 表达式。
- 不允许不安全的指针操作。
- COM 调用必须在参数上包含 `ref`；不能隐含。
- 不允许使用不受支持的 `__arglist` 关键字。

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

Open a documentation issue

Provide product feedback

解决与使用命名空间相关的警告

项目 • 2023/12/25

本文介绍以下编译器错误：

- CS0138：错误：using 命名空间指令只能应用于命名空间；“type”是一个类型，而不是命名空间。
- CS0431：错误：无法将别名“identifier”与“`::`”一起使用，因为该别名引用了类型。请改用 `.`。
- CS0432：错误：未找到别名“identifier”。
- CS0576：错误：命名空间“namespace”包含与别名“identifier”冲突的定义。
- CS0687：错误：命名空间别名限定符 `::` 始终解析为类型或命名空间，因此在这里是非法的。*请考虑改用* `.`。
- CS1529：错误：using 子句必须位于命名空间中定义的所有其他元素之前（外部别名声明除外）。
- CS1537：错误：using 别名“alias”以前在此命名空间中出现过。
- CS7000：错误：意外使用了别名。
- CS7007：错误：`using static` 指令只能应用于类型。请考虑改用 `using namespace` 指令
- CS8083：错误：别名限定名称不是表达式。
- CS8085：错误：“using static”指令不能用于声明别名。
- CS8914：错误：不能在命名空间声明中使用 global using 指令。
- CS8915：错误：global using 指令必须位于所有非 global using 指令之前。
- CS9055：错误：不能在“global using static”指令中使用文件本地类型。
- CS9130：错误：using 别名不能是 `ref` 类型。
- CS9131：错误：只有一个 using 别名可以是 `unsafe`。
- CS9132：错误：using 别名不能是可为 null 的引用类型。
- CS9133：错误：`static` 修饰符必须位于 `unsafe` 修饰符之前。
- CS9162：类型对“using static”无效。只能使用类、结构、接口、枚举、委托或命名空间。

并介绍以下编译器警告：

- CS0105：警告：“namespace”的 using 指令以前在此命名空间中出现过。
- CS0440：警告：由于 `global::` 总是引用全局命名空间而非别名，因此定义一个名为 `global` 的别名是欠妥的。
- CS8019：信息：不必要的 using 指令。
- CS8933：信息：using 指令在以前显示为 global using。

这些错误和警告指示 `using` 指令的格式不正确。 以下部分介绍这些错误及其更正方法。

Using 指令

指令 `using` 必须位于 `namespace` 声明中的任何其他元素之前，或位于文件中的任何 `namespace` 声明之前。 将 `using` 指令放在文件后面会导致编译器生成错误 CS1529：

C#

```
namespace UsingDirective;
public class MyClass
{
}

using System.Text.Json; // CS1529
```

若要解决此问题，请将任何 `using` 声明移动到文件顶部或命名空间顶部：

C#

```
using System.Text.Json;
namespace UsingDirective;
public class MyClass
{
}
```

编译器会针对 `using` 或 `global using` 指令中的重复 `using` 指令生成警告 CS8933、CS0105 或诊断 CS8019。 可以移除任何重复项。

本文稍后会介绍将 `using` 指令与 `using` 指令上的 `static`、 `global` 或 `unsafe` 修饰符错误地结合使用的情况。

Using static 指令

`using static` 指令将一种类型的成员导入到当前命名空间中。 以下示例将 `System.Console` 中的方法（例如 `WriteLine`）导入到当前命名空间中：

C#

```
using static System.Console;
```

如果省略 `static` 修饰符，编译器会生成 CS0138：

C#

```
using System.Console; // CS0138
```

如果包含的是用于导入命名空间而不是类型的 `static` 修饰符，编译器会生成 CS7007：

C#

```
using static System; // CS7007
```

如果符号不是正确的类型之一，编译器将发出 CS9162。

如果将 `static` 修饰符与 `using` 指令中的 `unsafe` 修饰符结合使用， `static` 修饰符必须放在首位：

C#

```
using static unsafe UnsafeExamples.UnsafeType;
```

全局 using 指令

`global using` 指令导入当前项目中所有源文件中的命名空间或类型：

C#

```
global using System.Text;
```

任何 `global using` 指令都必须在该源文件中的任何非全局 `using` 指令之前，并且不得放置在 `namespace` 中。 这样做会分别导致生成 CS8915 和 CS8914。

此外， `static global using` 指令不能引用文件本地类型。

别名限定符

别名限定符 :: 位于命名空间别名之前，或位于 `global` 别名之后。 如果在 `.` 应该用于分隔完全限定名称的元素的情况下使用 `::`，编译器会发出 CS0431、CS0432、CS0687、*CS7000 或 CS8083 之一。

在所有情况下，请将 `::` 替换为 `.` 分隔符。

此外，如果定义名为 `global` 的别名，编译器会发出 CS0440。 `global` 别名始终引用全局命名空间。 为它声明别名不起作用，应选择其他名称作为别名。

别名冲突

可以使用 using 指令声明命名空间或类型的别名：

```
C#

using JSON = System.Text.Json;
using ValueMap = System.Collections.Generic.Dictionary<string, decimal>;
using TimedData = (System.DateTime timeRecorded, decimal value);
```

应尝试为别名创建唯一名称，即前面示例中 `=` 符号左侧的名称。 使用已映射到某个类型（例如 `Object`）或命名空间（`System`）的名称会导致 CS0576 或 CS1537。

有关 using 别名的限制

在 C# 12 之前，该语言对创建类型声明的别名的 `using` 指令施加了以下限制：

- 不能使用 `using static` 指令创建别名：

  ```
  C#

  using static con = System.Console;
  using static unsafe ip = int*;
  ```

从 C# 12 开始，引入了以下限制：

- 不能在 using 别名中使用 `in`、`ref` 或 `out` 修饰符：

  ```
  C#

  // All these are invalid
  using RefInt = ref int;
  using OutInt = out int;
  using InInt = in int;
  ```

- `unsafe` using 指令必须指定别名或 `static using`：

  ```
  C#

  // Elsewhere:
  public namespace UnsafeExamples
  {
      public unsafe static class UnsafeType
      {
          // ...
  ```

```
    }
}

// Using directives:
using unsafe IntPointer = int*;
using static unsafe UnsafeExamples.UnsafeType;
using unsafe UnsafeExamples; // not allowed
```

- 不能为可为 null 的引用类型创建别名：

```
C#

using NullableInt = System.Int32?; // Allowed
using NullableString = System.String?; // Not allowed
```


在 GitHub 上与我们协作

可以在 GitHub 上找到此内容的源，还可以在其中创建和查看问题和拉取请求。 有关详细信息，请参阅参与者指南。

.NET

.NET 反馈

.NET 是一个开放源代码项目。 选择一个链接以提供反馈：

提出文档问题

提供产品反馈

与源生成器和拦截器关联的错误和警告

项目 • 2023/12/25

在编译过程中加载源生成器或拦截器时，可能会生成以下错误:

- CS9137: *未启用"拦截器"实验性功能。将 `<Features>InterceptorsPreview<Features>` 添加到项目。*
- CS9138: *方法不能用作拦截器，因为它或其包含类型具有类型参数。*
- CS9139: *无法拦截: 编译不包含具有路径的文件。*
- CS9140: *无法拦截: 编译不包含具有路径的文件。你是否想使用不同的路径?*
- CS9141: *提供的行数和字符数未引用可拦截的方法名称，而是引用标记。*
- CS9142: *给定文件有 `n` 行，少于提供的行数 `m`。*
- CS9143: *给定行长度为 `c` 字符，少于提供的字符数 `n`。*
- CS9144: *由于签名不匹配，无法使用拦截器 `V` 拦截方法 `M`。*
- CS9145: *无法拦截: 路径未映射。预期应映射路径。*
- CS9146: *拦截器方法必须是普通成员方法。*
- CS9147: *提供的行数和字符数未引用标记开头。你是否想使用行 `n` 和字符 `c`?*
- CS9148: *拦截器必须具有匹配参数的 `this` 参数。*
- CS9149: *由于方法没有 `this` 参数，因此拦截器不得具有 `this` 参数。*
- CS9150: *拦截器不能具有 `null` 文件路径。*
- CS9151: *无法拦截可能的方法名称 `M`，因为它未调用。*
- CS9152: *无法拦截具有此路径的文件中的调用，因为编译中的多个文件都有此路径。*
- CS9153: *多次拦截指示的调用。*
- CS9155: *无法使用 `M` 拦截调用，因为它在 `V` 中不可访问。*
- CS9156: *由于"作用域"修饰符或 `[UnscopedRef]` 属性存在差异，因此无法使用 `V` 拦截对 `M` 的调用。*
- CS9157: *提供给 `InterceptsLocationAttribute` 的行数和字符数必须为正数。*
- CS9160: *无法拦截 nameof 运算符。*
- **CS9161**: *不能使用 `UnmanagedCallersOnlyAttribute` 标记拦截器。*
- CS9177：拦截器必须是非泛型或具有匹配的 arity。
- CS9178：方法必须是非泛型方法才能匹配
- CS9206：无法在全局命名空间中声明拦截器。
- CS9207：无法拦截，因为方法不是对普通成员方法的调用。

在编译过程中加载源生成器或拦截器时，可能会生成以下警告:

- CS9154: *使用拦截器 `V` 拦截对 `M` 的调用，但签名不匹配。*
- CS9158: *返回类型中引用类型的为 Null 性与可拦截方法不匹配。*

- CS9159: *参数类型中引用类型的为 Null 性与可拦截方法不匹配。*

这些错误和警告遵循以下主题:

拦截器是实验性功能

此错误表示你必须启用试验性功能。

- CS9137: *未启用"拦截器"实验性功能。将 `<Features>InterceptorsPreview<Features>` 添加到项目。*

在 C# 12 中，拦截器是实验性功能。 在未来版本中，拦截器可能会进行中断性变更或删除。 因此，不建议将其用于生产或已发布的应用程序。

为了使用拦截器，你必须在项目文件中设置 `<Features>InterceptorsPreview<Features>` 元素。 如果没有此标志，会禁用拦截器，即使已启用其他 C# 12 功能也是如此。

签名不匹配

以下错误指示拦截器方法与可拦截方法不匹配，或者违反有关拦截器方法声明的规则:

- CS9144: *由于签名不匹配，无法使用拦截器 `V` 拦截方法 `M`。*
- CS9148: *拦截器必须具有匹配参数的 `this` 参数。*
- CS9149: *由于方法没有 `this` 参数，因此拦截器不得具有 `this` 参数。*
- CS9155: *无法使用 `M` 拦截调用，因为它在 `V` 中不可访问。*
- CS9156: *由于"作用域"修饰符或 `[UnscopedRef]` 属性存在差异，因此无法使用 `V` 拦截对 `M` 的调用。*
- CS9177：拦截器必须是非泛型或具有匹配的 arity。
- CS9178：方法必须是非泛型方法才能匹配

此外，以下警告指示拦截器和可拦截方法的签名不匹配:

- CS9154: *使用拦截器 `V` 拦截对 `M` 的调用，但签名不匹配。*
- CS9158: *返回类型中引用类型的为 Null 性与可拦截方法不匹配。*
- CS9159: *参数类型中引用类型的为 Null 性与可拦截方法不匹配。*

拦截器方法必须与可拦截方法兼容。 必须遵循以下规则:

- 实例方法可以拦截实例方法，不能拦截静态方法。 同样，静态方法只能拦截静态方法，不能拦截实例方法。
- 拦截器和可拦截方法的方法签名必须匹配: 它们必须具有相同的参数(具有相同的修饰符且顺序相同)。 返回类型也必须匹配。

- ref 安全上下文必须匹配。 换句话说，相应的`ref`参数必须是`scoped`或不是`scoped`。
- 这些方法必须都是非泛型方法，或者都具有相同数量的类型参数。

映射错误

拦截器需有映射可拦截方法和拦截器方法的源映射。 以下错误表明映射存在问题：

- CS9139: *无法拦截: 编译不包含具有路径的文件。*
- CS9140: *无法拦截: 编译不包含具有路径的文件。你是否想使用不同的路径?*
- CS9141: *提供的行数和字符数未引用可拦截的方法名称，而是引用标记。*
- CS9142: *给定文件有 `n` 行，少于提供的行数`m`。*
- CS9143: *给定行长度为 `c` 字符，少于提供的字符数 `n`。*
- CS9145: *无法拦截: 路径未映射。预期应映射路径。*
- CS9147: *提供的行数和字符数未引用标记开头。你是否想使用行 `n` 和字符 `c`?*
- CS9150: *拦截器不能具有 `null` 文件路径。*
- CS9157: *提供给 `InterceptsLocationAttribute` 的行数和字符数必须为正数。*

拦截器指定可拦截方法源代码中的位置。 应用`[InterceptsLocation]`属性以指定位置。 在重新映射的源文件中指定应注入拦截器的行数和列数。 这些错误指示属性或位置中的一些内容与有效可拦截方法的位置不匹配。 有关此属性的格式和值的详细信息，请参阅功能规范。

其他故障

这些错误指示拦截器的其他限制:

- CS9138: *方法不能用作拦截器，因为它或其包含类型具有类型参数。*
- CS9146: *拦截器方法必须是普通成员方法。*
- CS9151: *无法拦截可能的方法名称 `M`，因为它未调用。*
- CS9152: *无法拦截具有此路径的文件中的调用，因为编译中的多个文件都有此路径。*
- CS9153: *多次拦截指示的调用。*
- CS9160: *无法拦截 nameof 运算符。*
- **CS9161**: *不能使用 `UnmanagedCallersOnlyAttribute` 标记拦截器。*
- CS9206：无法在全局命名空间中声明拦截器。
- CS9207：无法拦截，因为方法不是对普通成员方法的调用。

这些错误指示拦截器方法违反了拦截器规则之一:

- 拦截器不能是泛型方法或泛型类的成员。

- 拦截器必须是普通成员。 它们不能是运算符、实例或静态构造函数或终结器。
- 可拦截的方法必须是普通成员。 它们不能是运算符、实例或静态构造函数或终结器，也不能是委托调用。
- 无法拦截从未调用的可拦截方法。
- 可拦截方法最多可以拦截一次。
- 拦截器不能是仅限于非托管调用方的方法。
- 拦截器必须包含在命名空间中。

在 GitHub 上与我们协作

可以在 GitHub 上找到此内容的源，还可以在其中创建和查看问题和拉取请求。 有关详细信息，请参阅参与者指南。

.NET 反馈

.NET 是一个开放源代码项目。 选择一个链接以提供反馈：

提出文档问题

提供产品反馈

静态抽象和虚拟接口成员错误和警告

项目 • 2023/12/06

对于接口中静态抽象或虚拟成员的无效声明，编译器会生成以下错误：

- CS8920：接口不能用作类型参数。静态成员在接口中没有最具体的实现。
- CS8921：一元运算符的参数必须是包含类型或其限制为该类型的类型参数。
- CS8922：`++` 或 `--` 运算符的参数必须是包含类型或其限制为该类型的类型参数。
- CS8923：`++` 或 `--` 运算符的返回类型必须与参数类型匹配，或者必须从参数类型派生，或者必须是包含类型的限制为该类型的类型参数，除非该参数类型是不同的类型参数。
- CS8924：二元运算符的参数之一必须是包含类型或其限制为该类型的类型参数。
- CS8925：重载移位运算符的第一个操作数的类型必须与包含类型或其限制为该类型的类型参数相同
- CS8926：只能在类型参数上访问静态虚拟或抽象接口成员。
- CS8928：类型不实现静态接口成员。该方法无法实现接口成员，因为它不是静态的。
- CS8930：用户定义的运算符的显式实现必须声明为静态
- CS8931：接口中用户定义的转换必须转换为或转换自封闭类型上限制为封闭类型的类型参数
- CS8932：“UnmanagedCallersOnly”方法无法实现类型中的接口成员

这些错误发生在代码中的三个位置：

- 声明具有静态抽象或虚拟成员的接口时，
- 声明一个类型时，该类型实现具有静态抽象或虚拟成员的接口，以及
- 访问接口中声明的静态抽象或虚拟方法时。

接口声明错误

声明具有 `static abstact` 或 `static virtual` 成员的接口时，可能会出现以下错误：

- CS8921：一元运算符的参数必须是包含类型或其限制为该类型的类型参数。
- CS8922：`++` 或 `--` 运算符的参数必须是包含类型或其限制为该类型的类型参数。
- CS8923：`++` 或 `--` 运算符的返回类型必须与参数类型匹配，或者必须从参数类型派生，或者必须是包含类型的限制为该类型的类型参数，除非该参数类型是不同的类型参数。
- CS8924：二元运算符的参数之一必须是包含类型或其限制为该类型的类型参数。
- CS8925：重载移位运算符的第一个操作数的类型必须与包含类型或其限制为该类型的类型参数相同

- CS8931：接口中用户定义的转换必须转换为或转换自封闭类型上限制为封闭类型的类型参数

所有这些规则都是声明重载运算符的规则的扩展。 区别在于，参数可以是接口类型，也可以是接口的类型参数（如果该类型参数限制为实现其类型的接口）。 对于二元运算符，只有一个参数必须满足此规则。

例如，`Inumber<T>` 可以声明 `T operator++(T)`，因为 `T` 限制为实现 `INumber<T>`。

若要修复这些错误，请确保接口中定义的任何运算符的参数都遵循这些规则。 可以在有关接口中的静态抽象成员的语言参考文章或探索静态抽象接口成员的教程中了解更多信息。

实现接口声明的类型中出现错误

定义使用 `static abstract` 或 `static virtual` 方法实现接口的类型时，可能会出现以下错误：

- CS8928：类型不实现静态接口成员。*该方法无法实现接口成员，因为它不是静态的。*
- CS8930：用户定义的运算符的显式实现必须声明为静态
- CS8932："UnmanagedCallersOnly"方法无法实现类型中的接口成员

这些错误都表明错误地声明了实现静态抽象接口成员的方法。 这些成员必须声明为 `static`；它们不能是实例成员。 实现接口成员的方法不能应用 System.Runtime.InteropServices.UnmanagedCallersOnlyAttribute 属性。

调用静态抽象接口成员时出错

尝试调用定义为接口的静态抽象或静态虚拟成员的成员时，可能会出现以下错误：

- CS8920：接口不能用作类型参数。*静态成员在接口中没有最具体的实现。*
- CS8926：只能在类型参数上访问静态虚拟或抽象接口成员。

对声明为 `static abstract` 或 `static virtual` 的接口成员的调用必须在编译时解析。 它们必须解析为在实现该接口的类型中定义的静态成员。 这意味着必须使用实现该接口的具体类型或限制为实现该接口的类型参数来访问这些成员。 要修复这些错误，请更改用于访问静态成员的类型。

在 GitHub 上与我们协作

.NET .NET 反馈

可以在 GitHub 上找到此内容的源，还可以在其中创建和查看问题和拉取请求。 有关详细信息，请参阅参与者指南。

.NET 是一个开放源代码项目。 选择一个链接以提供反馈：

提出文档问题

提供产品反馈

与 `lock` 语句和线程同步相关的错误和警告

项目 • 2024/05/11

与 `lock` 语句和线程同步相关的一些错误：

- CS0185：“type”不是 lock 语句要求的引用类型。
- CS1996：无法在 lock 语句的正文中等待
- CS9217：不能在异步方法或异步 lambda 表达式中针对“System.Threading.Lock”类型的值使用 lock 语句。

此外，编译器可能会生成与 `lock` 语句和线程同步相关的以下警告：

- CS9216：将类型为 `System.Threading.Lock` 的值转换为其他类型时，可能会在 `lock` 语句中使用意外的基于监视器的锁定。

lock 语句错误

- CS0185：“type”不是 lock 语句要求的引用类型。
- CS1996：无法在 lock 语句的正文中等待
- CS9217：不能在异步方法或异步 lambda 表达式中针对“System.Threading.Lock”类型的值使用 lock 语句。

这些错误表明代码违反了有关语句 `lock` 的规则：

- 要 `lock` 的对象必须是引用类型。 不允许使用值类型。
- 不能在 `lock` 语句的作用域中使用 `await` 表达式。
- `lock` 语句不能与 `async` 方法或 lambda 表达式一起使用。 对于此错误，可以将锁定的对象类型替换为其他类型。 `lock` 语句使用 Monitor API。

你必须更新代码，以遵循 `lock` 语句的规则。

lock 警告

- CS9216：将类型为 `System.Threading.Lock` 的值转换为其他类型时，可能会在 `lock` 语句中使用意外的基于监视器的锁定。

从 C# 13 开始，当目标对象是 `System.Threading.Lock` 对象时，lock 将生成专用代码。 当使用的是 `Lock` 对象时，编译器将生成此警告，但代码会将其类型转换为其他类型。 因此，编译器将生成常规锁定代码，而不是特定于 `Lock` 类型的锁定代码。 例如：

C#

```
object lockObject = new System.Threading.Lock();

lock (lockObject) // CS9216
{
    // .. Your code
}
```

你应确保变量或表达式表示 `Lock` 对象的类型。

在 GitHub 上与我们协作

可以在 GitHub 上找到此内容的源，还可以在其中创建和查看问题和拉取请求。 有关详细信息，请参阅参与者指南。

.NET 反馈

.NET 是一个开放源代码项目。 选择一个链接以提供反馈：

提出文档问题

提供产品反馈

C# 警告波

项目 • 2024/07/01

C# 编译器的每个版本中可能会引入新的警告和错误。 如果现有代码上可以报告新的警告，则会在称为警告波的选择加入系统下引入这些警告。 选择加入系统意味着，如果不采取操作启用现有代码，就不应在其中看到新的警告。 使用项目文件中的 AnalysisLevel 元素启用警告波。 指定 `<TreatWarningsAsErrors>true</TreatWarningsAsErrors>` 后，启用的警告波警告将生成错误。 C# 9 中添加了警告波 5 诊断。 C# 10 中添加了警告波 6 诊断。 C# 11 中添加了警告波 7 诊断。 C# 12 中添加了警告波 8 诊断。

CS9123 - 获取异步方法中的局部变量或参数的地址可能会产生 GC 漏洞。

警告波 8

`&` 运算符不应该用于异步方法中的参数或局部变量。 下面的代码生成 CS9123：

C#

```
public static async Task LogValue()
{
    int x = 1;
    unsafe {
        int* y = &x;
        Console.WriteLine(*y);
    }
    await Task.Delay(1000);
}
```

从 C# 13 开始，此代码将生成编译器错误。

CS8981 - 类型名称仅包含小写 ascii 字符。

警告波 7

为 C# 添加的任何新关键字都是小写 ASCII 字符。 此警告可确保任何类型都不会与将来的关键字冲突。 下面的代码生成 CS8981：

C#

```
public class lowercasename
{
```

```
}
```

可以通过重命名类型以包含至少一个非小写 ASCII 字符（例如大写字符、数字或下划线）来解决此警告。

CS8826 - 分部方法声明具有签名差异。

警告波 6

此警告修正了报告分部方法签名之间差异时的一些不一致。 当分部方法签名创建不同的 CLR 签名时，编译器始终报告错误。 现在，当签名为在语法上有差异的 C# 时，编译器会报告 CS8826。 考虑以下分部类：

C#

```
public partial class PartialType
{
    public partial void M1(int x);

    public partial T M2<T>(string s) where T : struct;

    public partial void M3(string s);


    public partial void M4(object o);
    public partial void M5(dynamic o);
    public partial void M6(string? s);
}
```

以下分部类实现生成多个 CS8626 示例：

C#

```
public partial class PartialType
{
    // Different parameter names:
    public partial void M1(int y) { }

    // Different type parameter names:
    public partial TResult M2<TResult>(string s) where TResult : struct =>
default;

    // Relaxed nullability
    public partial void M3(string? s) { }


    // Mixing object and dynamic
    public partial void M4(dynamic o) { }
```

```
    // Mixing object and dynamic
    public partial void M5(object o) { }

    // Note: This generates CS8611 (nullability mismatch) not CS8826
    public partial void M6(string s) { }
}
```

> **备注**
>
> 如果在其他声明接受可为 null 的引用类型时，方法的实现使用不可为 null 的引用类型，则生成 CS8611，而不是 CS8826。

若要修复这些警告的任何实例，请确保两个签名匹配。

CS7023 - 静态类型用于“is”或“as”表达式。

警告波 5

`is` 和 `as` 表达式始终返回 `false` 静态类型，因为你无法创建静态类型的实例。 下面的代码生成 CS7023：

C#

```
static class StaticClass
{
    public static void Thing() { }
}

void M(object o)
{
    // warning: cannot use a static type in 'is' or 'as'
    if (o is StaticClass)
    {
        Console.WriteLine("Can't happen");
    }
    else
    {
        Console.WriteLine("o is not an instance of a static class");
    }
}
```

编译器报告此警告，因为类型测试永远无法成功。 若要修正此警告，请删除测试，并删除仅在测试成功时才执行的任何代码。 在前面的示例中，始终执行 `else` 子句。 可以将该方法主体替换为该单行：

C#

```
Console.WriteLine("o is not an instance of a static class");
```

CS8073 - 表达式的结果始终为“false”（或“true”）。

警告波 5

将 `struct` 类型的实例与 `null` 进行比较时，`==` 和 `!=` 运算符始终返回 `false`（或 `true`）。 下面的代码对此警告进行了演示。 假设 `S` 是一个 `struct`，它定义 `operator ==` 和 `operator !=`：

C#

```
class Program
{
    public static void M(S s)
    {
        if (s == null) { } // CS8073: The result of the expression is always
'false'
        if (s != null) { } // CS8073: The result of the expression is always
'true'
    }
}

struct S
{
    public static bool operator ==(S s1, S s2) => s1.Equals(s2);
    public static bool operator !=(S s1, S s2) => !s1.Equals(s2);
    public override bool Equals(object? other)
    {
        // Implementation elided
        return false;
    }
    public override int GetHashCode() => 0;

    // Other details elided...
}
```

若要修复此错误，请删除在对象为 `null` 时将执行的 null 检查和代码。

CS8848 - 由于优先级，无法在此处使用运算符“from”。 使用括号消除歧义。

警告波 5

下面的示例演示了这一警告。 由于运算符的优先级，表达式绑定不正确。

C#

```
bool b = true;
var source = new Src();
b = true;
source = new Src();
var a = b && from c in source select c;
Console.WriteLine(a);

var indexes = new Src2();
int[] array = { 1, 2, 3, 4, 5, 6, 7 };
var range = array[0..from c in indexes select c];
```

若要修复此错误，请用括号将查询表达式括起来：

C#

```
bool b = true;
var source = new Src();
b = true;
source = new Src();
var a = b && (from c in source select c);
Console.WriteLine(a);

var indexes = new Src2();
int[] array = { 1, 2, 3, 4, 5, 6, 7 };
var range = array[0..(from c in indexes select c)];
```

必须为成员完全赋值。 使用未赋值的变量（CS8880、CS8881、CS8882、CS8883、CS8884、CS8885、CS8886、CS8887）

警告波 5

多个警告改进了导入程序集中声明的 `struct` 类型的明确赋值分析。 当导入程序集中的结构包含引用类型的不可访问的字段时（通常为 `private` 字段），将生成所有这些新警告，如以下示例所示：

C#

```
public struct Struct
{
    private string data = String.Empty;
```

```
    public Struct() { }
}
```

以下示例显示了改进的明确赋值分析中生成的警告：

- CS8880：在控制返回调用方之前，自动实现的属性“Property”必须完全赋值。
- CS8881：在控制返回调用方之前，字段“field”必须完全赋值。
- CS8882：在控件离开当前方法之前，必须对 out 参数“parameter”赋值。
- CS8883：使用了可能未赋值的自动实现的属性“Property”。
- CS8884：使用了可能未赋值的字段“Field”
- CS8885：为所有字段赋值之前，不能使用“this”对象。
- CS8886：使用了未赋值的输出参数“parameterName”。
- CS8887：使用了未赋值的局部变量“variableName”

C#

```
public struct DefiniteAssignmentWarnings
{
    // CS8880
    public Struct Property { get; }
    // CS8881
    private Struct field;

    // CS8882
    public void Method(out Struct s)
    {

    }

    public DefiniteAssignmentWarnings(int dummy)
    {
        // CS8883
        Struct v2 = Property;
        // CS8884
        Struct v3 = field;
        // CS8885:
        DefiniteAssignmentWarnings p2 = this;
    }

    public static void Method2(out Struct s1)
    {
        // CS8886
        var s2 = s1;
        s1 = default;
    }

    public static void UseLocalStruct()
    {
        Struct r1;
        var r2 = r1;
```

```
    }
}
```

可以通过初始化或将导入的结构赋值为默认值来修复上述任何警告：

C#

```
public struct DefiniteAssignmentNoWarnings
{
    // CS8880
    public Struct Property { get; } = default;
    // CS8881
    private Struct field = default;

    // CS8882
    public void Method(out Struct s)
    {
        s = default;
    }

    public DefiniteAssignmentNoWarnings(int dummy)
    {
        // CS8883
        Struct v2 = Property;
        // CS8884
        Struct v3 = field;
        // CS8885:
        DefiniteAssignmentNoWarnings p2 = this;
    }

    public static void Method2(out Struct s1)
    {
        // CS8886
        s1 = default;
        var s2 = s1;
    }

    public static void UseLocalStruct()
    {
        Struct r1 = default;
        var r2 = r1;
    }
}
```

CS8892 - 方法不会用作入口点，因为找到了同步入口点“method”。

警告波 5

如果有多个有效入口点，其中包含一个或多个同步入口点，则将在所有异步入口点候选项上生成此警告。

下面的示例生成 CS8892：

C#

```
public static void Main()
{
    RunProgram();
}

// CS8892
public static async Task Main(string[] args)
{
    await RunProgramAsync();
}
```

> **备注**
>
> 编译器始终使用同步入口点。 如果有多个同步入口点，则会出现编译器错误。

若要修复此警告，请删除或重命名异步入口点。

CS8897 - 静态类型不能用作参数

警告波 5

接口的成员无法声明其类型为静态类的参数。 以下代码演示了 CS8897 和 CS8898：

C#

```
public static class Utilities
{
    // elided
}

public interface IUtility
{
    // CS8897
    public void SetUtility(Utilities u);

    // CS8898
    public Utilities GetUtility();
}
```

若要修复此警告，请更改参数类型或删除方法。

CS8898 - 静态类型不能用作返回类型

警告波 5

接口的成员无法声明静态类的返回类型。 以下代码演示了 CS8897 和 CS8898：

C#

```
public static class Utilities
{
    // elided
}

public interface IUtility
{
    // CS8897
    public void SetUtility(Utilities u);

    // CS8898
    public Utilities GetUtility();
}
```

若要修复此警告，请更改返回类型或删除方法。

在 GitHub 上与我们协作

可以在 GitHub 上找到此内容的源，还可以在其中创建和查看问题和拉取请求。 有关详细信息，请参阅参与者指南。

.NET

.NET 反馈

.NET 是一个开放源代码项目。 选择一个链接以提供反馈：

提出文档问题

提供产品反馈

编译器错误 CS0001

项目 • 2023/04/07

内部编译器错误

尝试确定编译器是否因为无法分析意外的语法而产生故障。 如果反复收到此错误，请联系 Microsoft。

编译器错误 CS0003

项目 • 2023/04/07

内存不足

编译器分配的虚拟内存不足，无法完成编译。 关闭所有不必要的应用程序并重新编译。

你可能想要增加页面文件大小，请确保具有可用的磁盘空间。

如果 Windows SDK 和 C# 编译器的版本不匹配，并且支持 C# 编译器的一个或多个文件损坏，也会出现此错误；请重新安装 Visual Studio。

编译器错误 CS0004

项目 • 2023/04/07

视为错误的警告

你使用了 `/warnaserror` 编译器选项进行编译，这会导致编译器生成作为错误的警告。

编译器错误 CS0005

项目 • 2023/04/07

编译器选项“compiler_option”后面必须有参数

某些编译器选项需要参数。 若未传递编译器选项所需的传递参数，则会生成 CS0005。

有关详细信息，请参阅 C# 编译器选项。

编译器错误 CS0006

项目 • 2023/04/07

未能找到元数据文件“dll_name”

程序已编译并已显式传递了包含元数据的文件的名称，但没有找到 .dll。 有关详细信息，请参阅 References（C# 编译器选项）。

编译器错误 CS0007

项目 • 2023/04/07

意外的公共语言运行时初始化错误 --“description”

如果未能加载运行时，则会发生此错误。 如果计算机上不存在编译器尝试加载的公共语言运行时的版本，或者公共语言运行时的安装或配置已损坏，则可能发生此错误。

如果更改了 csc.exe.config 文件，则可能发生上述情况。 此文件在安装期间配置，不应更改。 如果可能更改了 csc.exe.config 文件，请检查该文件，确保计算机上存在该文件中指定的运行时版本。 如果存在正确版本，它可能已损坏。 重新安装公共语言运行时。

编译器错误 CS0008

项目 • 2023/04/07

从文件“file”读取元数据时发生错误 --“description”

已为检索元数据成功打开 DLL，但它已损坏，因此无法从中读取数据。 有关详细信息，请参阅 References（C# 编译器选项）。

编译器错误 CS0009

项目 • 2023/04/07

无法打开元数据文件“file” - “description”

使用 References 编译器选项指定的文件未包含有效的元数据。

编译器错误 CS0010

项目 • 2023/04/07

意外致命错误 --“error”。

出现会阻止编译的完全意外的内容时，会生成此错误。

更正此错误

再重新编译一个项目。 如果你收到相同错误，请尝试重新安装 Visual Studio。 如果你收到此错误，请向 Microsoft 发送错误报告。

编译器错误 CS0011

项目 • 2023/04/07

未能解析程序集“assembly”中由类型“type”引用的基类或接口“class”

使用 **/reference**从类文件导入的类派生自找不到的类或实现了找不到的接口。 如果没有使用 **/reference**将所需 DLL 也包含在编译中，则可能出现这种情况。

有关详细信息，请参阅“添加引用”对话框和 References（C# 编译器选项）。

示例

C#

```
// CS0011_1.cs
// compile with: /target:library

public class Outer
{
   public class B { }
}
```

第二个文件创建一个 DLL，它定义了派生自上一示例中所创建 `C` 类的 `B` 类。

C#

```
// CS0011_2.cs
// compile with: /target:library /reference:CS0011_1.dll
// post-build command: del /f CS0011_1.dll
public class C : Outer.B {}
```

第三个文件替换第一步创建的 DLL，并忽略内部类 `B`的定义。

C#

```
// CS0011_3.cs
// compile with: /target:library /out:cs0011_1.dll
public class Outer {}
```

最后，第四个文件引用在第二个示例中定义的类 `C` ，该类派生自类 `B`且现已丢失。

以下示例生成 CS0011。

C#

```
// CS0011_4.cs
// compile with: /reference:CS0011_1.dll /reference:CS0011_2.dll
// CS0011 expected

class M
{
   public static void Main()
   {
      C c = new C();
   }
}
```

编译器错误 CS0013

项目 • 2023/05/10

将元数据写入文件“ModelStore.dll”时出现意外错误--没有剩余的逻辑空间来创建更多用户字符串。

.NET 运行时未发出元数据。 检查以确保路径正确无误且磁盘未满。 如果问题仍然存在，你可能需要修复或重新安装 Visual Studio 和/或 .NET。

编译器错误 CS0014

项目 • 2023/04/07

找不到所需的文件 “file”

编译器所需的某个文件不在系统上。 请确保路径正确无误。 如果该文件是 Visual Studio 系统文件，则你可能需要进行修复安装或删除并完全重新安装 Visual Studio。

编译器错误 CS0015

项目 • 2023/04/07

类型“type”的名称太长

用户定义类型的完全限定名必须少于 1024 个字符，包括句点。

下面的示例生成 CS0015：

C#

```
// CS0015.cs

// Remove a C from one of the namespace names or the class name to resolve
the error by
// reducing the number of characters in the qualified class name to fewer
than 1024.

namespace CC
{
    // The following namespace name has 510 characters.
    namespace
CCC510charsCCC
```

编译器错误 CS0016

项目 • 2023/04/07

未能写入输出文件“file”—“reason”

编译器未能写入输出文件。 检查文件路径，确保存在该路径。 如果该位置已存在一个以前生成的文件，请确保该文件可写入，并且当前没有进程将该文件锁定。 例如，确保在尝试构建时可执行文件未在运行。

编译器错误 CS0017

项目 • 2023/04/07

程序"输出文档名称"定义了多个入口点。 请使用 /main 进行编译，以指定包含入口点的类型。

一个程序只能有一个 Main 方法。

若要解决此错误，可在代码中删除所有 Main 方法，仅保留一个，或者可使用 StartupObject 编译器选项来指定想要使用的 Main 方法。

下面的示例生成 CS0017：

C#

```
// CS0017.cs
// compile with: /target:exe
public class clx
{
   static public void Main()
   {
   }
}

public class cly
{
   public static void Main()   // CS0017, delete one Main or use /main
   {
   }
}
```

编译器错误 CS0019

项目 • 2023/04/07

运算符“operator”无法应用于“type”和“type”类型的操作数

对不支持二元运算符的数据类型应用该运算符。 例如，无法在字符串中使用 || 运算符，无法在 bool 变量中使用 +、-、< 或 > 运算符，并且不能配合使用 == 运算符和 `struct` 类型（除非该类型显式重载该运算符）。

可重载运算符，使其支持某些类型的操作数。 有关详细信息，请参阅运算符重载。

示例 1

在下例中，在 3 个位置生成 CS0019，原因是 C# 中的 bool 不能转换为 int。将减法运算符 `-` 应用于字符串时，也会生成 CS0019。 加法运算符 `+` 可与字符串操作数一起使用，因为 `String` 类会重载该运算符以执行字符串串联。

C#

```
static void Main()
{
    bool result = true;
    if (result > 0) //CS0019
    {
        // Do something.
    }

    int i = 1;
    // You cannot compare an integer and a boolean value.
    if (i == true) //CS0019
    {
        //Do something...
    }

    string s = "Just try to subtract me.";
    float f = 100 - s; // CS0019
}
```

示例 2

在以下示例中，必须在 ConditionalAttribute 外指定条件逻辑。 只能向 ConditionalAttribute 传递一个预定义符号。

以下示例生成 CS0019：

C#

```
// CS0019_a.cs
// compile with: /target:library
using System.Diagnostics;

public class MyClass
{
   [ConditionalAttribute("DEBUG" || "TRACE")]   // CS0019
   public void TestMethod() {}

   // OK
   [ConditionalAttribute("DEBUG"), ConditionalAttribute("TRACE")]
   public void TestMethod2() {}
}
```

另请参阅

- C# 运算符

编译器错误 CS0020

项目 • 2023/04/07

被常数零除

表达式在除法运算的分母中使用了文本（不是变量）值零。 未定义除数为零，因此无效。

下面的示例生成 CS0020:

```C#
// CS0020.cs
namespace x
{
   public class b
   {
      public static void Main()
      {
         1 / 0;   // CS0020
      }
   }
}
```

另请参阅

- 算术运算符

编译器错误 CS0021

项目 • 2023/04/07

无法将带 [] 的索引应用于“type”类型的表达式

尝试针对不支持 Indexers的数据类型使用索引器访问值。

在 C++ 程序集中尝试使用索引器时，可能会遇到 CS0021。 在这种情况下，请用 `DefaultMember` 特性修饰 C++ 类，让 C# 编译器能够识别默认索引器。 下面的示例生成 CS0021。

示例

以下 C++ 示例编译为 .dll 文件。 请注意，为了生成错误，DefaultMember 属性会被注释掉。

C++

```
// CPP0021.cpp
// compile with: /clr /LD
using namespace System::Reflection;
// Uncomment the following line to resolve
//[DefaultMember("myItem")]
public ref class MyClassMC
{
        public:
        property int myItem[int]
        {
            int get(int i){  return 5; }
            void set(int i, int value) {}
        }
};
```

以下 C# 示例调用 .dll 文件。 它会尝试通过索引器访问类，但由于没有将任何成员声明为默认索引器，因此产生了此错误。 可通过取消注释上一示例中 .cpp 文件的 `[DefaultMember("myItem")]` 行来解决此错误。

C#

```
// CS0021.cs
// compile with: /reference:CPP0021.dll
public class MyClass
{
    public static void Main()
    {
        MyClassMC myMC = new MyClassMC();
```

```
            int j = myMC[1]; // CS0021
        }
    }
```

编译器错误 CS0023

项目 • 2023/04/07

运算符“operator”无法应用于“type”类型的操作数

尝试将运算符应用于类型没有被设计用于该运算符的变量。 有关详细信息，请参阅类型和 C# 运算符。

下面的示例生成 CS0023：

C#

```
// CS0023.cs
namespace x
{
   public class a
   {
      public static void Main()
      {
         string s = "hello";
         s = -s;   // CS0023, minus operator not allowed on strings
      }
   }
}
```

编译器错误 CS0025

项目 • 2023/04/07

找不到标准库文件“file”

找不到编译器需要的文件。 检查路径是否正确并检查该文件是否存在。

如果该文件是 Visual Studio 系统文件，则你可能需要修复 Visual Studio 安装，或者完全重新安装。

编译器错误 CS0026

项目 • 2023/04/07

关键字 "this" 在静态属性、静态方法或静态字段初始值设定项中无效

this 关键字是指一个对象，其为某类型的实例。 由于静态方法与包含类的任何实例均无关，因此“this”关键字不含任何意义，因此不受允许。 有关详细信息，请参阅静态类和静态类成员以及对象。

示例

下面的示例生成 CS0026：

C#

```
// CS0026.cs
public class A
{
   public static int i = 0;

   public static void Main()
   {
// CS0026
      this.i = this.i + 1;
      // Try the following line instead:
      // i = i + 1;
   }
}
```

编译器错误 CS0027

项目 • 2023/04/07

关键字 "this" 在当前上下文中不可用

在属性、方法或构造函数的外部发现了 this 关键字。

若要修复此错误，请修改语句以消除 `this` 关键字，关键字的使用，并且/或者将语句的一部分或全部移到属性、方法或构造函数的内部。

下面的示例生成 CS0027：

C#

```
using System;
using System.Collections.Generic;
using System.Text;

namespace ConsoleApplication3
{
    class MyClass
    {

        int err1 = this.Fun() + 1;  // CS0027

        public int Fun()
        {
            return 10;
        }

        public void Test()
        {
            // valid use of this
            int err = this.Fun() + 1;
            Console.WriteLine(err);
        }

        public static void Main()
        {
            MyClass c = new MyClass();
            c.Test();
        }
    }
}
```

编译器错误 CS0029

项目 • 2023/05/10

无法将类型“type”隐式转换为“type”

编译器要求显式转换。 例如，可能需要将右值转换成与左值相同的类型。 或者必须提供转换例程以支持某些运算符重载。

在将某个类型的变量赋给其他类型的变量时必须进行转换。 在不同类型的变量之间进行赋值时，编译器必须将赋值运算符右边的类型转换为赋值运算符左边的类型。 以下面的代码为例：

```
C#

int i = 50;
long lng = 100;
i = lng;
```

`i = lng;` 进行赋值运算，但赋值运算符左右两边变量的数据类型不匹配。 进行赋值前，编译器将变量 `lng`（类型为 long）隐式转换为 int。此为隐式转换，因为没有代码显式指示编译器执行此转换。 此代码的问题在于上述转换被视为收缩转换，但由于可能会丢失数据，所以编译器不允许进行隐式收缩转换。

如果转换后的数据类型所占用的内存存储空间比转换前的数据类型所占用的少，则存在收缩转换。 例如，将 long 类型转换为 int 类型可视为收缩转换。 long 类型占用 8 个字节的内存，而 int 类型只占用 4 个字节。 若要查看数据丢失如何发生，请考虑以下示例：

```
C#

int i = 50;
long lng = 3147483647;
i = lng;
```

变量 `lng` 现在包含的值因过大而无法存储在变量 `i` 中。 如果要将该值转换为 int 类型，就会丢失一些数据，并且转换后的值不同于转换前的值。

扩大转换与收缩转换相反。 对于扩大转换，转换后的数据类型占用的内存存储空间比转换前的数据类型占用的多。 下面是一个扩大转换示例：

```
C#

int i = 50;
long lng = 100;
lng = i;
```

请注意此代码示例和第一个示例之间的区别。 此示例中，变量 `lng` 位于赋值运算符的左边，所以它是赋值目标。 在可以进行赋值前，编译器必须将变量 `i`（类型为 int）隐式转换为 long 类型。 这是一个扩大转换，因为是从占用 4 个字节内存的类型 (int) 转换为占用 8 个字节内存的类型 (long)。 由于不会发生数据丢失，所以允许进行隐式扩大转换。 任何可存储于 int 类型的值也可存储于 long 类型。

由于不允许进行隐式收缩转换，因此为了能够编译此代码，需要显式转换数据类型。 使用强制转换完成显式转换。 强制转换是 C# 中用来描述将一种数据类型转换为另一种数据类型的术语。 若要编译代码，需使用以下语法：

C#

```
int i = 50;
long lng = 100;
i = (int) lng;   // Cast to int.
```

第三行代码告知编译器在进行赋值前，将变量 `lng`（类型为 long）显式转换为 int 类型。 切记，使用收缩转换可能会丢失数据。 应小心使用收缩转换，即使可以编译代码，也可能会在运行时获得意外结果。

此讨论只针对值类型。 使用值类型就是直接使用存储在变量中的数据。 但是，.NET 也具有引用类型。 使用引用类型就是使用对变量的引用，而不是使用实际数据。 引用类型的示例包括类、接口和数组。 不能隐式或显式地将一个引用类型转换为其他引用类型，除非编译器允许特定的转换或可以实现相应的转换运算符。

以下示例生成 CS0029：

C#

```
// CS0029.cs
public class MyInt
{
    private int x = 0;

    // Uncomment this conversion routine to resolve CS0029.
    /*
    public static implicit operator int(MyInt i)
    {
        return i.x;
    }
    */

    public static void Main()
    {
        var myInt = new MyInt();
        int i = myInt; // CS0029
```

```
    }
}
```

另请参阅

- 用户定义转换运算符

编译器错误 CS0030

项目 • 2023/04/07

无法将类型“type”转换为“type”

类型之间没有预定义的转换。 可在这些类型之间定义自定义转换。 有关详细信息，请参阅用户定义转换运算符。

下面的示例生成 CS0030：

C#

```
// CS0030.cs
namespace x
{
   public class iii
   {
      /*
      public static implicit operator iii(int aa)
      {
         return null;
      }

      public static implicit operator int(iii aa)
      {
         return 0;
      }
      */

      public static iii operator ++(iii aa)
      {
         return (iii)0;   // CS0030
         // uncomment the conversion routines to resolve CS0030
      }

      public static void Main()
      {
      }
   }
}
```

编译器错误 CS0031

项目 • 2023/04/07

常量值“value”无法转换为“'type”。

尝试对其类型不能存储值的变量进行复制。 有关详细信息，请参阅类型。

在已选中和未选中的上下文中，下面的示例都将生成 CS0031：

C#

```
// CS0031.cs
namespace CS0031
{
    public class Program
    {
        public static void Main()
        {
            int num = (int)2147483648M; //CS0031
            // Try using a larger numeric type instead.
            // long num = (long)2147483648M; //CS0031

            const decimal d = -10M; // Decimal literal
            unchecked
            {
                const byte b = (byte)d; // CS0031
                // For small values try using a signed byte instead.
                // const sbyte b = (sbyte)d;
            }
        }
    }
}
```

另请参阅

- unchecked

编译器错误 CS0036

项目 • 2023/04/07

out 参数不能具有“[In]”特性

目前， **out** 参数上不允许 In 特性。

下面的示例生成 CS0036：

C#

```
// CS0036.cs

using System;
using System.Runtime.InteropServices;

public class MyClass
{
   public static void TestOut([In] out char TestChar)   // CS0036
   // try the following line instead
   // public static void TestOut(out char TestChar)
   {
      TestChar = 'b';
      Console.WriteLine(TestChar);
   }

   public static void Main()
   {
      char i;           //variable to receive the value
      TestOut(out i);   // the arg must be passed as out
      Console.WriteLine(i);
   }
}
```

编译器错误 CS0037

项目 • 2023/04/07

无法将 null 转换成“type”，因为它是不可以为 null 的值类型

编译器无法将 `null` 分配到值类型；仅可将 `null` 分配给引用类型或可以为 null 的类型。 struct 是一个值类型。

下面的示例生成 CS0037：

C#

```
// CS0037.cs
public struct s
{
}

class a
{
   public static void Main()
   {
      int i = null;   // CS0037
      s ss = null;    // CS0037
   }
}
```

编译器错误 CS0038

项目 • 2023/04/07

无法通过嵌套类型“type2”访问外部类型“type1”的非静态成员

类中的字段并不自动可用于嵌套类。 若要可用于嵌套类，该字段必须是静态字段。 否则，必须创建外部类的实例。 有关详细信息，请参阅嵌套类型。

下面的示例生成 CS0038：

C#

```
// CS0038.cs
class OuterClass
{
    public int count;
    // Try the following line instead.
    // public static int count;

    class InnerClass
    {
        void Func()
        {
            // or, create an instance
            // OuterClass class_inst = new OuterClass();
            // int count2 = class_inst.count;
            int count2 = count;   // CS0038
        }
    }

    public static void Main()
    {
    }
}
```

编译器错误 CS0039

项目 • 2023/04/07

无法通过引用转换、装箱转换、取消装箱转换、包装转换或 null 类型转换将类型“type1”转换为“type2”

继承、引用转换和装箱转换允许使用 as 运算符的转换。

示例

以下示例生成 CS0039：

C#

```
using System;

class A { }
class B : A { }
class C : A { }

class Example
{
    static void Main()
    {
        C c;

        // This compiles, because
        // there is an explicit reference conversion from type A to type C.
        A a = new C();
        c = a as C;

        // This generates CS0039, because
        // there is no implicit or explicit reference conversion between B
and C types.
        B b = new B();
        c = b as C;  // CS0039
    }
}
```

编译器错误 CS0040

项目 • 2023/04/07

创建调试信息文件时出现意外错误—“原因”

使用 DebugType 编译器选项时会出现此错误，表示编译器无法写 .pdb 文件。 针对此错误的可能解决方法包括重新安装 Visual Studio、确保编译器具有写入文件或目录的访问权限，或者不使用 /debug 进行编译。

编译器错误 CS0041

项目 • 2024/04/11

写入调试信息时出错 --“{error}”

在 GitHub 上与我们协作

可以在 GitHub 上找到此内容的源，还可以在其中创建和查看问题和拉取请求。 有关详细信息，请参阅参与者指南。

.NET 反馈

.NET 是一个开放源代码项目。 选择一个链接以提供反馈：

提出文档问题

提供产品反馈

编译器错误 CS0042

项目 • 2023/04/07

创建调试信息文件“file”时出现意外错误—“原因”

编译器无法创建调试信息； *原因* 包含错误条件的其他信息。

编译器错误 CS0043

项目 • 2023/04/07

PDB 文件“file”的格式不正确或过时。 请将其删除并重新生成。

删除此编译生成的 .pdb 文件并重新编译。

> **备注**
>
> Roslyn 中不再使用此编译器错误。

编译器错误 CS0050

项目 • 2023/04/07

可访问性不一致：返回类型“type”的可访问性低于方法“method”

方法的返回类型和形参表中引用的每个类型都必须至少具有和方法自身相同的可访问性。有关详细信息，请参阅访问修饰符。

示例

以下示例生成 CS0050，由于没有为 `MyClass` 提供任何可访问性修饰符，因此其可访问性默认为 `private`：

C#

```
// CS0050.cs
class MyClass // Accessibility defaults to private.
// Try the following line instead.
// public class MyClass
{
}

public class MyClass2
{
    public static MyClass MyMethod()   // CS0050
    {
        return new MyClass();
    }

    public static void Main() { }
}
```

编译器错误 CS0051

项目 • 2023/04/07

可访问性不一致：参数类型“type”的可访问性低于方法“method”

方法的返回类型和形参表中引用的每个类型都必须至少具有和方法自身相同的可访问性。 请确保方法签名中使用的类型不会因为省略 `public` 修饰符而意外变为私有类型。 有关详细信息，请参阅访问修饰符。

示例

以下示例生成 CS0051：

C#

```
// CS0051.cs
public class A
{
    // Try making B public since F is public
    // B is implicitly private here.
    class B
    {
    }

    public static void F(B b)  // CS0051
    {
    }

    public static void Main()
    {
    }
}
```

编译器错误 CS0052

项目 • 2023/04/07

可访问性不一致：字段类型“type”的可访问性低于字段“field”

字段类型的可访问性不能低于字段本身的可访问性，因为所有公共构造函数都必须返回一个可公开访问的对象。

示例

下面的示例生成 CS0052：

C#

```
// CS0052.cs
public class MyClass2
{
    // The following line causes an error because the field, M, is declared
    // as public, but the type, MyClass, is private. Therefore the type is
    // less accessible than the field.
    public MyClass M;   // CS0052

    private class MyClass
    {
    }
    // One way to resolve the error is to change the accessibility of the
type
    // to public.
    //public class MyClass
    // Another solution is to change the accessibility of the field to
private.
    // private MyClass M;
}

public class MainClass
{
    public static void Main()
    {
    }
}
```

请参阅

- C# 关键字
- 访问修饰符
- 可访问性级别

- 修饰符

编译器错误 CS0053

项目 • 2023/04/07

可访问性不一致：属性类型“type”的可访问性低于属性“property”

公共构造必须返回可以公开访问的对象。 有关详细信息，请参阅访问修饰符。

下面的示例生成 CS0053：

C#

```
// CS0053.cs
class MyClass //defaults to private accessibility
// try the following line instead
// public class MyClass
{
}

public class MyClass2
{
   public MyClass myProperty   // CS0053
   {
      get
      {
         return new MyClass();
      }
      set
      {
      }
   }
}

public class MyClass3
{
   public static void Main()
   {
   }
}
```

编译器错误 CS0054

项目 • 2023/04/07

可访问性不一致：索引器返回类型“type”的可访问性低于索引器“indexer”

公共构造必须返回可以公开访问的对象。 有关详细信息，请参阅访问修饰符。

以下示例生成 CS0054：

C#

```
// CS0054.cs
class MyClass
// try the following line instead
// public class MyClass
{
}

public class MyClass3
{
   public MyClass this[int i]   // CS0054
   {
      get
      {
         return new MyClass();
      }
   }
}

public class MyClass2
{
   public static void Main()
   {
   }
}
```

编译器错误 CS0055

项目 • 2023/04/07

可访问性不一致：参数类型“type”的可访问性低于索引器“indexer”

公共构造必须返回可以公开访问的对象。 有关详细信息，请参阅访问修饰符。

下面的示例生成 CS0055：

C#

```
// CS0055.cs
class MyClass //defaults to private accessibility
// try the following line instead
// public class MyClass
{
}

public class MyClass2
{
   public int this[MyClass myClass]   // CS0055
   {
      get
      {
         return 0;
      }
   }
}

public class MyClass3
{
   public static void Main()
   {
   }
}
```

编译器错误 CS0056

项目 • 2023/04/07

可访问性不一致：返回类型“type”的可访问性低于运算符“operator”

公共构造必须返回可以公开访问的对象。 有关详细信息，请参阅访问修饰符。

以下示例生成 CS0056：

C#

```
// CS0056.cs
class MyClass
// try the following line instead
// public class MyClass
{
}

public class A
{
   public static implicit operator MyClass(A a)   // CS0056
   {
      return new MyClass();
   }

   public static void Main()
   {
   }
}
```

编译器错误 CS0057

项目 • 2023/04/07

可访问性不一致: 参数类型“type”的可访问性低于运算符“operator”

公共构造必须返回可以公开访问的对象。 有关详细信息，请参阅访问修饰符。

下面的示例生成 CS0057：

C#

```
// CS0057.cs
class MyClass //defaults to private accessibility
// try the following line instead
// public class MyClass
{
}

public class MyClass2
{
   public static implicit operator MyClass2(MyClass iii)   // CS0057
   {
      return new MyClass2();
   }

   public static void Main()
   {
   }
}
```

编译器错误 CS0058

项目 • 2023/04/07

可访问性不一致: 返回类型“type”的可访问性低于委托“delegate”

公共构造必须返回可以公开访问的对象。 有关详细信息，请参阅访问修饰符。

下面的示例生成 CS0058，因为没有访问修饰符应用于 MyClass，因此默认授予它私有可访问性：

C#

```
// CS0058.cs
class MyClass
// try the following line instead
// public class MyClass
{
}

public delegate MyClass MyClassDel();   // CS0058

public class A
{
   public static void Main()
   {
   }
}
```

另请参阅

- private

编译器错误 CS0059

项目 • 2023/04/07

可访问性不一致：参数类型“type”的可访问性低于委托“delegate”

方法的返回类型和形参表中引用的每个类型都必须至少具有和方法自身相同的可访问性。有关详细信息，请参阅访问修饰符。

示例

下面的示例生成 CS0059：

C#

```
// CS0059.cs
class MyClass //defaults to private accessibility
// try the following line instead
// public class MyClass
{
}

public delegate void MyClassDel( MyClass myClass);   // CS0059

public class Program
{
    public static void Main()
    {
    }
}
```

编译器错误 CS0060

项目 • 2023/04/07

可访问性不一致: 基类“class1”的可访问性低于类“class2”

基类和继承类之间的类可访问性应保持一致。

下面的示例生成 CS0060：

C#

```
// CS0060.cs
class MyClass
// try the following line instead
// public class MyClass
{
}

public class MyClass2 : MyClass   // CS0060
{
   public static void Main()
   {
   }
}
```

另请参阅

- 访问修饰符

编译器错误 CS0061

项目 • 2023/04/07

可访问性不一致：基接口“interface 1”的可访问性低于接口“interface 2”

公共 构造必须返回可以公开访问的对象。

派生接口中不能收缩接口的可访问性。 有关详细信息，请参阅接口和访问修饰符。

以下示例生成 CS0061。

C#

```
// CS0061.cs
// compile with: /target:library
internal interface A {}
public interface AA : A {}  // CS0061

// OK
public interface B {}
internal interface BB : B {}

internal interface C {}
internal interface CC : C {}
```

编译器错误 CS0065

项目 • 2023/04/07

“event”：事件属性必须同时具有 add 和 remove 访问器

并不是字段的事件必须有两种访问方法。

以下示例生成 CS0065：

C#

```
// CS0065.cs
using System;
public delegate void EventHandler(object sender, int e);
public class MyClass
{
   public event EventHandler Click   // CS0065,
   {
      // to fix, uncomment the add and remove definitions
      /*
      add
      {
         Click += value;
      }
      remove
      {
         Click -= value;
      }
      */
   }

   public static void Main()
   {
   }
}
```

请参阅

- 事件

编译器错误 CS0066

项目 • 2023/04/07

“event”: 事件必须是委托类型

事件关键字需要 委托 类型。 有关详细信息，请参阅事件和委托。

下面的示例生成 CS0066：

C#

```
// CS0066.cs
using System;

public class EventHandler
{
}

// to fix the error, remove the event declaration and the
// EventHandler class declaration, and uncomment the following line
// public delegate void EventHandler();

public class a
{
   public event EventHandler Click;   // CS0066

   private void TestMethod()
   {
      if (Click != null)
         Click();
   }

   public static void Main()
   {
   }
}
```

编译器错误 CS0068

项目 • 2023/04/07

“event”: 接口中的事件不能有初始值设定项

接口中的事件不能有初始值设定项。 有关详细信息，请参阅接口。

下面的示例生成 CS0068：

C#

```
// CS0068.cs

delegate void MyDelegate();

interface I
{
   event MyDelegate d = new MyDelegate(M.f);   // CS0068
   // try the following line instead
   // event MyDelegate d2;
}

class M
{
   event MyDelegate d = new MyDelegate(M.f);

   public static void f()
   {
   }

   public static void Main()
   {
   }
}
```

编译器错误 CS0069

项目 • 2023/04/07

接口中的事件不能具有 add 或 remove 访问器

不能在 接口中定义某一事件的访问器函数。 有关详细信息，请参阅事件和接口。

以下示例生成 CS0069：

C#

```
// CS0069.cs
// compile with: /target:library

public delegate void EventHandler();

public interface a
{
   event EventHandler Click { remove {} }   // CS0069
   event EventHandler Click2;   // OK
}
```

编译器错误 CS0070

项目 • 2023/04/07

事件“event”只能出现在 += 或 -= 的左边（从类型“type”中使用时除外）

在定义事件的类的外部， event 只能加上或减去引用。 有关详细信息，请参阅事件。

下面的示例生成 CS0070：

C#

```
// CS0070.cs
using System;
public delegate void EventHandler();

public class A
{
   public event EventHandler Click;

   public static void OnClick()
   {
      EventHandler eh;
      A a = new A();
      eh = a.Click;
   }

   public static void Main()
   {
   }
}

public class B
{
   public int Foo ()
   {
      EventHandler eh = new EventHandler(A.OnClick);
      A a = new A();
      eh = a.Click;   // CS0070
      // try the following line instead
      // a.Click += eh;
      return 1;
   }
}
```

编译器错误 CS0071

项目 • 2023/04/07

事件的显式接口实现必须使用事件访问器语法

显式实现在接口中声明的事件时，必须手动提供通常由编译器提供的 `add` 和 `remove` 事件访问器。 访问器代码可将接口事件连接到类中的另一事件（稍后在本主题中介绍），或连接到其自身的委托类型。 有关详细信息，请参阅如何实现接口事件。

示例

下面的示例生成 CS0071。

C#

```
// CS0071.cs
public delegate void MyEvent(object sender);

interface ITest
{
    event MyEvent Clicked;
}

class Test : ITest
{
    event MyEvent ITest.Clicked;  // CS0071

    // Try the following code instead.
    /*
    private MyEvent clicked;

    event MyEvent ITest.Clicked
    {
        add
        {
            clicked += value;
        }
        remove
        {
            clicked -= value;
        }
    }
    */
    public static void Main() { }
}
```

编译器错误 CS0072

项目 • 2023/04/07

“event”：无法重写；“method”不是事件

事件 只能重写另一事件。 有关详细信息，请参阅事件。

示例

下面的示例生成 CS0072：

C#

```
// CS0072.cs
delegate void MyDelegate();

class Test1
{
    public virtual event MyDelegate MyEvent;
    public virtual void VMeth()
    {
    }

    public void FireAway()
    {
        if (MyEvent != null)
            MyEvent();
    }
}

class Test2 : Test1
{
    public override event MyDelegate VMeth   // CS0072
    // Uncomment the following lines to resolve.
    // public override event MyDelegate MyEvent
    {
        add
        {
            VMeth += value;
            // MyEvent += value;
        }
        remove
        {
            VMeth -= value;
            // MyEvent -= value;
        }
   }

   public static void Main()
   {
```

```
    }
}
```

编译器错误 CS0073

项目 • 2023/04/07

add 访问器或 remove 访问器必须有一个主体

事件 定义中的 **add** 或 remove 关键字必须有一个主体。 有关详细信息，请参阅事件。

以下示例生成 CS0073：

C#

```
// CS0073.cs
delegate void del();

class Test
{
   public event del MyEvent
   {
      add;   // CS0073
      // try the following lines instead
      // add
      // {
      //    MyEvent += value;
      // }
      remove
      {
         MyEvent -= value;
      }

   }

   public static void Main()
   {
   }
}
```

编译器错误 CS0074

项目 • 2023/04/07

“event”: 抽象事件不能有初始值设定项

如果 event 被标记为 **抽象**，则无法将其初始化。 有关详细信息，请参阅事件。

下面的示例生成 CS0074：

C#

```
// CS0074.cs
delegate void D();

abstract class Test
{
   public abstract event D e = null;   // CS0074
   // try the following line instead
   // public abstract event D e;

   public static void Main()
   {
   }
}
```

编译器错误 CS0075

项目 • 2023/04/07

若要强制转换某个负值，必须将该值放在括号内

如果使用标识预定义类型的关键字进行强制转换，则不需要括号。 否则，必须用括号括起来，因为 (x) –y 不会被视为强制转换表达式。 摘自 C# 规范的第 7.6.6 节：

从消除歧义规则可以导出下述结论：如果 x 和 y 是标识符，则 (x)y、(x)(y) 和 (x)(-y) 是强制转换表达式，但 (x)-y 不是，即使 x 标识的是类型。 但是，如果 x 是一个标识预定义类型（如 int）的关键字，则所有这四种形式均为强制转换表达式（因为这种关键字本身不可能是表达式）。

以下代码生成 CS0075：

C#

```
// CS0075
namespace MyNamespace
{
    enum MyEnum { }
    public class MyClass
    {
        public static void Main()
        {
            // To fix the error, place the negative
            // values below in parentheses
            int i = (System.Int32) - 4; //CS0075
            MyEnum e = (MyEnum) - 1;    //CS0075
            System.Console.WriteLine(i); //to avoid warning
            System.Console.WriteLine(e); //to avoid warning
        }
    }
}
```

另请参阅

- 强制转换和类型转换

编译器错误 CS0076

项目 • 2023/10/18

枚举器名“value__”是保留字，不能使用

枚举中的项不能有一个名为 **value__** 的标识符，因为它被保留用作枚举的后备字段。

以下示例生成 CS0076：

```
C#
```

```
// CS0076.cs
public enum E
{
    value__   // CS0076
}
```


Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET feedback

The .NET documentation is open source. Provide feedback here.

Open a documentation issue

Provide product feedback

编译器错误 CS0077

项目 • 2024/10/22

as 运算符必须与引用类型或可以为 null 的类型一起使用（“int”是不可以为 null 值的类型）。

向 as 运算符传递了 值类型。 `as` 可返回 null，因此只能向其传递 引用类型 或可以为 null 的值类型。

但是，通过使用 pattern matching 和 is 操作符，我们可以在一个步骤中直接执行类型检查和赋值。

下面的示例生成 CS0077：

C#

```
// CS0077.cs
using System;

class C
{
}

struct S
{
}

class M
{
   public static void Main()
   {
      object o1, o2;
      C c;
      S s;

      o1 = new C();
      o2 = new S();

      s = o2 as S;    // CS0077, S is not a reference type

      // Use pattern matching instead of as
      if (o2 is S sValue)
      {
          s = sValue;
      }
   }
}
```

编译器错误 CS0079

项目 • 2023/04/07

事件“event”只能出现在 += 或 -= 的左边

未正确调用 event 。 有关详细信息，请参阅事件和委托。

下面的示例生成 CS0079：

C#

```
// CS0079.cs
using System;

public delegate void MyEventHandler();

public class Class1
{
   private MyEventHandler _e;

   public event MyEventHandler Pow
   {
      add
      {
         _e += value;
         Console.WriteLine("in add accessor");
      }
      remove
      {
         _e -= value;
         Console.WriteLine("in remove accessor");
      }
   }

   public void Handler()
   {
   }

   public void Fire()
   {
      if (_e != null)
      {
         Pow();   // CS0079
         // try the following line instead
         // _e();
      }
   }

   public static void Main()
   {
      Class1 p = new Class1();
```

```
        p.Pow += new MyEventHandler(p.Handler);
        p._e();
        p.Pow += new MyEventHandler(p.Handler);
        p._e();
        p._e -= new MyEventHandler(p.Handler);
        if (p._e != null)
        {
            p._e();
        }
        p.Pow -= new MyEventHandler(p.Handler);
        if (p._e != null)
        {
            p._e();
        }
    }
}
```

编译器错误 CS0080

项目 • 2023/04/07

在非泛型声明上不允许使用约束

找到的语法只能在泛型声明中用于将约束应用于类型形参。 有关详细信息，请参阅泛型。

下面的示例生成 CS0080，因为 MyClass 不是泛型类，并且 Foo 不是泛型方法。

C#

```
namespace MyNamespace
{
    public class MyClass where MyClass : System.IDisposable // CS0080
//the following line shows an example of correct syntax
    //public class MyClass<T> where T : System.IDisposable
    {
        public void Foo() where Foo : new() // CS0080
        //the following line shows an example of correct syntax
        //public void Foo<U>() where U : struct
        {
        }
    }

    public class Program
    {
        public static void Main()
        {
        }
    }
}
```

编译器错误 CS0081

项目 • 2023/04/07

类型形参声明必须是标识符，不能是类型

声明泛型方法或类型时，请将类型形参指定为标识符，例如“T”或“inputType”。 当客户端代码调用该方法时，它会提供类型，该类型将替换该标识符在该方法或类正文中的每个匹配项。 有关详细信息，请参阅泛型类型参数。

C#

```
// CS0081.cs
class MyClass
{
   public void F<int>() {}   // CS0081
   public void F<T>(T input) {}   // OK

   public static void Main()
   {
      MyClass a = new MyClass();
      a.F<int>(2);
      a.F<double>(.05);
   }
}
```

另请参阅

- 泛型

编译器错误 CS0082

项目 • 2023/04/07

类型“type”已保留了一个名为“name”的具有相同参数类型的成员

属性在编译时被转换为标识符前带有 `get_` 和/或 `set_` 的方法。 如果定义自己的方法，且该方法与方法名称冲突，将生成错误。

示例

下面的示例生成 CS0082：

C#

```
//cs0082.cs
class MyClass
{

    //property
    public int MyProp
    {
        get //CS0082
        {
            return 1;
        }
    }

    //conflicting Get
    public int get_MyProp()
    {
        return 2;
    }

    public static int Main()
    {
        return 1;
    }
}
```

请参阅

- 属性

编译器错误 CS0100

项目 • 2023/04/07

参数名“parameter name”重复

方法声明多次使用相同的参数名。 参数名在方法声明中必须具有唯一性。 有关详细信息，请参阅方法。

下面的示例生成 CS0100::

C#

```
// CS0100.cs
namespace x
{
   public class a
   {
      public static void f(int i, char i)   // CS0100
      // try the following line instead
      // public static void f(int i, char j)
      {
      }

      public static void Main()
      {
      }
   }
}
```

编译器错误 CS0101

项目 • 2023/04/07

命名空间“namespace”已经包含“type”的定义

namespace 具有重复的标识符。 重命名或删除其中一个重复的标识符。 有关详细信息，请参阅命名空间

以下示例生成 CS0101：

C#

```
// CS0101.cs
namespace MyNamespace
{
   public class MyClass
   {
      static public void Main()
      {
      }
   }

   public class MyClass   // CS0101
   {
   }
}
```

当类名与命名空间名称冲突时，也会生成 CS0101。 在使用根类的帮助程序类进行扩展中，会发生这种情况（在基类中，你尝试使命名空间路由保持一致）。 在下例中，UTF8 类显然应该是 String 类的附属类，但如果尝试通过将所述命名空间声明为 Utilities.String 来强制它进入同一个命名空间，则将导致 CS0101 错误：

C#

```
//CS0101-Utilities.String.cs
namespace Utilities
{
   public class String
   {
        public string MyString;
   }
}

//CS0101-Utilities.String.UTF8.cs
namespace Utilities.String  // CS0101
{
   public class UTF8
   {
```

```
        public string MySecondString;
    }
}
```

编译器错误 CS0102

项目 • 2023/04/07

类型“type name”已经包含“identifier”的定义

类包含在同一范围内多个同名标识符的声明。 若要修复此错误，请重命名重复的标识符。

示例

下面的示例生成 CS0102。

C#

```
// CS0102.cs
// compile with: /target:library
namespace MyApp
{
   public class MyClass
   {
      string s = "Hello";
      string s = "Goodbye";   // CS0102

      public void GetString()
      {
         string s = "Hello again";   // method scope, no error
      }
   }
}
```

编译器错误 CS0103

项目 • 2023/04/07

当前上下文中不存在名称“identifier”

尝试了使用类、命名空间或范围中不存在的名称。 检查名称的拼写，并检查 using 指令和程序集引用，确保您尝试使用的名称可用。

如果在循环或者 `try` 或 `if` 块中声明一个变量，然后尝试从封闭代码块或独立代码块访问此变量，通常就会发生此错误，如下面的示例所示：

> **备注**
>
> 如果表达式 lambda 中缺少运算符 `=>` 中的 `greater than` 符号，也可能出现此错误。 有关详细信息，请参阅**表达式** lambda。

C#

```
using System;

class MyClass1
{
    public static void Main()
    {
        try
        {
            // The following declaration is only available inside the try
block.
            var conn = new MyClass1();
        }
        catch (Exception e)
        {
            // The following expression causes error CS0103, because
variable
            // conn only exists in the try block.
            if (conn != null)
                Console.WriteLine("{0}", e);
        }
    }
}
```

下面的示例解决了这一错误：

C#

```
using System;

class MyClass2
{
    public static void Main()
    {
        // To resolve the error in the example, the first step is to
        // move the declaration of conn out of the try block. The following
        // declaration is available throughout the Main method.
        MyClass2 conn = null;
        try
        {
            // Inside the try block, use the conn variable that you declared
            // previously.
            conn = new MyClass2();
        }
        catch (Exception e)
        {
            // The following expression no longer causes an error, because
            // the declaration of conn is in scope.
            if (conn != null)
                Console.WriteLine("{0}", e);
        }
    }
}
```

编译器错误 CS0104

项目 • 2023/04/07

“reference”是“identifier”和“identifier”之间的不明确引用

程序包含两个命名空间的 使用 指令，而代码引用了在两个命名空间中都出现的名称。

以下示例生成 CS0104：

C#

```
// CS0104.cs
using x;
using y;

namespace x
{
   public class Test
   {
   }
}

namespace y
{
   public class Test
   {
   }
}

public class a
{
   public static void Main()
   {
      Test test = new Test();   // CS0104, is Test in x or y namespace?
      // try the following line instead
      // y.Test test = new y.Test();
   }
}
```

编译器错误 CS0106

项目 • 2023/10/07

修饰符“modifier”对该项无效

类或接口的成员标记有无效的访问修饰符。 下面的示例介绍其中某些无效的修饰符：

- 本地函数中不允许使用静态修饰符。 从 C# 8.0 开始支持静态本地函数功能。 尝试使用此功能时，不支持 C# 8.0 的编译器会生成 CS0106。 但是，支持 C# 8.0 但将语言版本设置为 C# 8.0 及更低版本的编译器将生成一个诊断，该诊断建议使用 C# 8.0 或更高版本。
- 显式接口声明上不允许使用 `public` 关键字。 在这种情况下，从显式接口声明删除 `public` 关键字。
- 显式接口声明上不允许具有抽象关键字，因为永远不能重写显式接口实现。
- 本地函数中不允许出现访问修饰符。 本地函数始终为私有函数。
- 类类型中的方法不允许使用 readonly 关键字，`ref readonly` 返回除外（`readonly` 关键字必须出现在 `ref` 关键字之后）。

在 Visual Studio 的先前版本中，类上不允许具有 `static` 修饰符，但允许具有以 Visual Studio 2005 开头的 `static` 类。

有关详细信息，请参阅接口。

示例

下面的示例生成 CS0106：

C#

```
// CS0106.cs
namespace MyNamespace
{
   interface I
   {
      void M1();
      void M2();
   }

   public class MyClass : I
   {
      public readonly int Prop1 { get; set; }   // CS0106
      public int Prop2 { get; readonly set; }   // CS0106
```

```
      public void I.M1() {}   // CS0106
      abstract void I.M2() {}   // CS0106

      public void AccessModifierOnLocalFunction()
      {
         public void LocalFunction() {}   // CS0106
      }

      public readonly void ReadonlyMethod() {}   // CS0106
      // Move the `readonly` keyword after the `ref` keyword
      public readonly ref int ReadonlyBeforeRef(ref int reference)   //
CS0106
      {
         return ref reference;
      }

      public static void Main() {}
   }

   public readonly class ReadonlyClass {}   // CS0106
}
```

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET feedback

The .NET documentation is open source. Provide feedback here.

- Open a documentation issue
- Provide product feedback

编译器错误 CS0107

项目 • 2023/04/07

多个保护修饰符

一个类成员只允许使用一个访问修饰符（public、private、protected 或 internal），protected internal 和 private protected 除外。

下面的示例生成 CS0107：

C#

```
// CS0107.cs
public class C
{
    private internal void F()   // CS0107, delete private or internal
    {
    }
}
```

编译器错误 CS0110

项目 • 2024/03/01

“const declaration”的常量值计算涉及循环定义

const 变量的声明不能引用另一个 const 变量，以免形成循环依赖关系。 这也适用于枚举成员的关联常量值。

示例

`MyClass.Color.Red` 的关联常量值显式设置为 `MyClass.Color.Blue`，但 `MyClass.Color.Blue` 的值取决于以前的枚举成员 (`MyClass.Color.Red`)，因此无法确定这两个常量值。 同样，常量变量 `MyClass.a` 依据 `MyClass.b` 定义，但也依据 `MyClass.a` 来定义。

下面的示例生成 CS0110：

C#

```
// CS0110.cs
// compile with: /target:library
class MyClass
{
    enum Color
    {
        Red = Blue,   // CS0110
        Blue,
    }

    public const int a = b + 1;   // CS0110
    public const int b = a + 1;
}
```

若要解决此错误，请通过修改或删除定义来中断循环引用。

另请参阅

- 常量

在 GitHub 上与我们协作

.NET .NET 反馈

可以在 GitHub 上找到此内容的源，还可以在其中创建和查看问题和拉取请求。 有关详细信息，请参阅参与者指南。

.NET 是一个开放源代码项目。 选择一个链接以提供反馈：

提出文档问题

提供产品反馈

编译器错误 CS0112

项目 • 2023/04/07

静态成员“函数”不能标记为 override、virtual 或 abstract

使用 `override`、 **virtual**或 **abstract** 关键字的任何方法声明也不能使用 **static** 关键字。

有关详细信息，请参阅方法。

下面的示例生成 CS0112:

C#

```
// CS0112.cs
namespace MyNamespace
{
   abstract public class MyClass
   {
      public abstract void Foo();
   }
   public class MyClass2 : MyClass
   {
      override public static void Foo()   // CS0112, remove static keyword
      {
      }
      public static int Main()
      {
         return 0;
      }
   }
}
```

编译器错误 CS0113

项目 • 2023/04/07

标记为 override 的成员“function”不能标记为 new 或 virtual

使用 new 和 override 关键字标记一种方法是互相排斥的。

下面的示例生成 CS0113：

C#

```
// CS0113.cs
namespace MyNamespace
{
   abstract public class MyClass
   {
      public abstract void Foo();
   }

   public class MyClass2 : MyClass
   {
      override new public void Foo()   // CS0113, remove new keyword
      {
      }

      public static int Main()
      {
         return 0;
      }
   }
}
```

编译器错误 CS0115

项目 • 2023/04/07

“function”：未找到合适的重写方法

方法标记为 override，但编译器未找到重写方法。 有关详细信息，请参阅 override 和了解何时使用 Override 和 New 关键字。

示例

下面的示例生成 CS0115。 可通过以下列两种方法之一解决 CS0115 错误：

- 从 `MyClass2` 中的方法删除 `override` 关键字。
- 将 `MyClass1` 用作 `MyClass2` 的基类。

C#

```
// CS0115.cs
namespace MyNamespace
{
    abstract public class MyClass1
    {
        public abstract int f();
    }

    abstract public class MyClass2
    {
        public override int f()   // CS0115
        {
            return 0;
        }

        public static void Main()
        {
        }
    }
}
```

编译器错误 CS0116

项目 • 2023/04/07

命名空间不能直接包含字段或方法之类的成员。

命名空间可以包含其他命名空间、结构和类。 有关详细信息，请参阅 namespace 关键字一文。

示例

以下示例将导致 Visual Studio 将部分代码标记为违反 CS0116。 尝试生成此代码将导致生成失败：

C#

```
// CS0116.cs
namespace x
{
    // A namespace can be placed within another namespace.
    using System;

    // These variables trigger the CS0116 error as they are declared outside
of a struct or class.
    public int latitude;
    public int longitude;
    Coordinate coord;

    // Auto-properties also fall under the definition of this rule.
    public string LocationName { get; set; }

    // This method as well: if it isn't in a class or a struct, it's
violating CS0116.
    public void DisplayLatitude()
    {
        Console.WriteLine($"Lat: {latitude}");
    }

    public struct Coordinate
    {
    }

    public class CoordinatePrinter
    {
        public void DisplayLongitude()
        {
            Console.WriteLine($"Longitude: {longitude}");
        }

        public void DisplayLocation()
```

```
        {
            Console.WriteLine($"Location: {LocationName}");
        }
    }
}
```

请注意，在 C# 中，必须在结构或类中声明和定义方法和变量。 有关 C# 中的程序结构的详细信息，请参阅 C# 程序的通用结构一文。 若要修复此错误，请重写代码，以便所有方法和字段均包含在结构或类中：

C#

```
namespace x
{
    // A namespace can be placed within another namespace.
    using System;

    // These variables are now placed within a struct, so CS0116 is no
longer violated.
    public struct Coordinate
    {
        public int Latitude;
        public int Longitude;
    }

    // The methods and fields are now placed within a class, and the
compiler is satisfied.
    public class CoordinatePrinter
    {
        Coordinate coord;
        public string LocationName { get; set; }

        public void DisplayLatitude()
        {
            Console.WriteLine($"Lat: {coord.Latitude}");
        }

        public void DisplayLongitude()
        {
            Console.WriteLine($"Longitude: {coord.Longitude}");
        }

        public void DisplayLocation()
        {
            Console.WriteLine($"Location: {LocationName}");
        }
    }
}
```

另请参阅

- C# 程序的通用结构
- C# 类型系统
- 命名空间

编译器错误 CS0117

项目 • 2023/04/07

“type”不包含“identifier”的定义

- 当引用的成员对于此数据类型不存在时，某些情况下会出现此错误。

示例

下面的示例生成 CS0117：

C#

```
public class Base { }

public class Derived : Base
{
    public void TestInt()
    {
        int i = base.someMember; // CS0117
    }
}
```

编译器错误 CS0118

项目 • 2023/04/07

“construct1_name”是“construct1”，但如同“construct2”一样使用

编译器检测到以下情况：以错误方式使用了某个构造，或尝试对某个构造执行不允许的操作。 一些常见示例包括以下这些：

- 尝试实例化命名空间（而不是类）
- 尝试调用字段（而不是方法）
- 尝试将类型用作变量
- 尝试将外部别名用作类型。

要解决此错误，请确保所执行的操作对于正在对其执行该操作的类型有效。

示例

下面的示例生成 CS0118。

C#

```
// CS0118.cs
// compile with: /target:library
namespace MyNamespace
{
   class MyClass
   {
      // MyNamespace not a class
      MyNamespace ix = new MyNamespace ();   // CS0118
   }
}
```

编译器错误 CS0119

项目 • 2023/04/07

“construct1_name”是“construct1”，在指定上下文中无效。

编译器检测到了以下意外构造:

- 类构造函数不是条件语句中的有效测试表达式。
- 使用了类名而不是实例名来引用数组元素。
- 将方法标识符视为结构或类使用

示例

下面的示例生成 CS0119：“C.B()”是一个在给定上下文中无效的方法。 可以通过更改方法 `C.B` 的名称，或者使用 `B` 类的完全限定的名称（如 `N2.B`）来修复此错误。

C#

```
namespace N2
{
    public static class B
    {
        public static void X() {}
    }
}

namespace N1
{
    public class C
    {
        void B() {}
        void M() => B.X();   // CS0119
    }
}
```

编译器错误 CS0120

项目 • 2023/04/07

非静态的字段、方法或属性“member”要求对象引用

必须先创建对象实例，才能使用非静态的字段、方法或属性。 有关静态方法的详细信息，请参阅静态类和静态类成员。 有关创建类的实例的详细信息，请参阅实例构造函数。

示例 1

以下示例生成 CS0120：

C#

```
// CS0120_1.cs
public class MyClass
{
    // Non-static field.
    public int i;
    // Non-static method.
    public void f() {}
    // Non-static property.
    int Prop
    {
        get
        {
            return 1;
        }
    }

    public static void Main()
    {
        i = 10;   // CS0120
        f();   // CS0120
        int p = Prop;   // CS0120
    }
}
```

若要更正此错误，请首先创建类的实例：

C#

```
// CS0120_1.cs
public class MyClass
{
    // Non-static field.
```

```
    public int i;
    // Non-static method.
    public void f() { }
    // Non-static property.
    int Prop
    {
        get
        {
            return 1;
        }
    }

    public static void Main()
    {
        var mc = new MyClass();
        mc.i = 10;
        mc.f();
        int p = mc.Prop;
    }
}
```

示例 2

如果从静态方法调用非静态方法，也会生成 CS0120，如下所示：

C#

```
// CS0120_2.cs
// CS0120 expected
using System;

public class MyClass
{
    public static void Main()
    {
        TestCall();   // CS0120
   }

   public void TestCall()
   {
   }
}
```

若要更正此错误，请首先创建类的实例：

C#

```
// CS0120_2.cs
using System;
```

```
public class MyClass
{
    public static void Main()
    {
        var anInstanceofMyClass = new MyClass();
        anInstanceofMyClass.TestCall();
    }

    public void TestCall()
    {
    }
}
```

示例 3

同样，静态方法不能调用实例方法，除非显式给它提供了类的实例，如下所示：

C#

```
// CS0120_3.cs
using System;

public class MyClass
{
   public static void Main()
   {
      DoIt("Hello There");   // CS0120
   }

   private void DoIt(string sText)
   {
      Console.WriteLine(sText);
   }
}
```

若要更正此错误，还可以将关键字 static 添加到方法定义中：

C#

```
// CS0120_3.cs
using System;

public class MyClass
{
   public static void Main()
   {
      DoIt("Hello There");   // CS0120
   }

   private static void DoIt(string sText)
```

```
    {
        Console.WriteLine(sText);
    }
}
```

请参阅

- C# 类型系统

编译器错误 CS0122

项目 • 2023/10/07

“member”不可访问，因为它具有一定的保护级别

类成员的访问修饰符禁止访问该成员。 有关详细信息，请参阅访问修饰符。

扩展方法无法访问它们所扩展的类型的专用成员。

导致出现此错误（未在下面的示例中显示）的一个原因是：在友元程序集的目标上省略了 /out 编译器标志。 有关详细信息，请参阅友元程序集和 OutputAssembly（C# 编译器选项）。

示例

下面的示例生成 CS0122：

C#

```
// CS0122.cs
public class MyClass
{
    private int data;

    void PrivateMethod() {}
    public void PublicMethod() {}
}

public static class MyClassExtensions
{
    public static int GetData(this MyClass myClass)
    {
        return myClass.data;   // CS0122
    }
}

public class Program
{
    public static void Main()
    {
        MyClass a = new MyClass();
        a.PrivateMethod();   // CS0122
        a.PublicMethod();   // OK
    }
}
```

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET feedback

The .NET documentation is open source. Provide feedback here.

Open a documentation issue

Provide product feedback

编译器错误 CS0123

项目 • 2023/04/07

“method”的重载均与委托“delegate”不匹配

尝试创建委托失败，因为未使用正确的签名。 必须使用与委托声明相同的签名声明委托实例。

你可通过调整方法或委托签名来解决此错误。 有关详细信息，请参阅委托。

以下示例生成 CS0123。

C#

```
// CS0123.cs
delegate void D();
delegate void D2(int i);

public class C
{
   public static void f(int i) {}

   public static void Main()
   {
      D d = new D(f);   // CS0123
      D2 d2 = new D2(f);   // OK
   }
}
```

编译器错误 CS0126

项目 • 2023/04/07

需要一个类型可转换为“type”的对象

存在一个 return 语句，但该语句不返回预期类型的值。 有关详细信息，请参阅 属性。

以下示例生成 CS0126：

C#

```
// CS0126.cs
public class a
{
   public int i
   {
      set
      {
      }
      get
      {
         return;   // CS0126, specify a value to return
      }
   }
}
```

编译器错误 CS0127

项目 • 2023/04/07

由于“function”返回 void，返回关键字后面不得有对象表达式

具有 void 返回类型的方法不能返回值。 有关详细信息，请参阅方法。

下面的示例生成 CS0127：

C#

```
// CS0127.cs
namespace MyNamespace
{
   public class MyClass
   {
      public int hiddenMember2
      {
         get
         {
            return 0;
         }
         set   // CS0127, set has an implicit void return type
         {
            return 0;   // remove return statement to resolve this CS0127
         }
      }

      public static void Main()
      {
      }
   }
}
```

编译器错误 CS0128

项目 • 2023/04/07

已在此范围定义了名为“variable”的局部变量

编译器检测到两个同名的局部变量的声明。 有关详细信息，请参阅方法。

下面的示例生成 CS0128：

C#

```
// CS0128.cs
namespace MyNamespace
{
   public class MyClass
   {
      public static void Main()
      {
         char i;
         int i;   // CS0128
      }
   }
}
```

编译器错误 CS0131

项目 • 2023/04/07

赋值号左边必须是变量、属性或索引器

在赋值语句中，右侧的值赋值给了左侧。 左侧必须是变量、属性或索引器。

若要修复此错误，请确保所有运算符均右侧且左侧为变量、属性或索引器。 有关详细信息，请参阅运算符和表达式。

示例 1

以下示例生成 CS0131。

C#

```
// CS0131.cs
public class MyClass
{
    public int i = 0;
    public void MyMethod()
    {
        i++ = 1;   // CS0131
        // try the following line instead
        // i = 1;
    }
    public static void Main() { }
}
```

示例 2

如果你尝试在赋值运算符的左侧执行算术运算，也可能出现此错误，如下例所示。

C#

```
// CS0131b.cs
public class C
{
    public static int Main()
    {
        int a = 1, b = 2, c = 3;
        if (a + b = c) // CS0131
        // try this instead
        // if (a + b == c)
            return 0;
        return 1;
```

```
    }
}
```

编译器错误 CS0132

项目 • 2023/04/07

“constructor”: 静态构造函数必须无参数

不能使用一个或多个参数声明 静态 构造函数。 有关详细信息，请参阅构造函数。

下面的示例生成 CS0132：

C#

```
// CS0132.cs
namespace MyNamespace
{
   public class MyClass
   {
      public MyClass(int i)
      {
      }
   }

   public class MyClass2 : MyClass
   {
      static MyClass2(int i)   // CS0132
      {
      }
   }
}
```

编译器错误 CS0133

项目 • 2023/04/07

指派给“variable”的表达式必须是常量

const 变量不能将非常量表达式作为其值。 有关详细信息，请参阅常量。

下面的示例生成 CS0133：

C#

```
// CS0133.cs
public class MyClass
{
   public const int i = c;   // CS0133, c is not constant
   public static int c = i;
   // try the following line instead
   // public const int i = 6;

   public static void Main()
   {
   }
}
```

编译器错误 CS0134

项目 • 2023/04/07

“variable”的类型为“type”。 只能使用 null 对引用类型（字符串除外）的常量字段进行初始化。

常数表达式是可在编译时完全计算的表达式。 由于应用 new 运算符是创建引用类型的非 null 值的唯一方法，并且常数表达式中不允许使用 new 运算符，因此除字符串外，引用类型常量的唯一可能值为 null。

如果尝试创建 const 字符串数组时遇到此错误，解决方案是使该数组成为只读数组，并在构造函数中将其初始化。

示例

下面的示例生成 CS0134：

C#

```
// CS0134.cs
// compile with: /target:library
class MyTest {}

class MyClass
{
    const MyTest test = new MyTest();   // CS0134

    //OK
    const MyTest test2 = null;
    const System.String test3 = "test";
}
```

编译器错误 CS0135

项目 • 2023/04/07

“declaration1”与声明“declaration2”冲突

编译器不允许隐藏名称，这通常会导致代码中的逻辑错误。

示例

下面的示例生成 CS0135：

C#

```
// CS0135.cs
public class MyClass2
{
   public static int i = 0;

   public static void Main()
   {
      {
         int i = 4;
         i++;
      }
      i = 0;   // CS0135
   }
}
```

来自 C# 语言规范：

局部变量声明空间和嵌套局部变量声明空间不能包含同名的元素。 因此在嵌套声明空间中，不可能在封闭声明空间中声明与局部变量或常量同名的局部变量或常量。 只要声明空间都不包含另一个声明空间，两个声明空间都可包含同名的元素。

编译器错误 CS0136

项目 • 2023/04/07

不能在此范围内声明名为“var”的局部变量，因为这样会使“var”具有不同的含义，而它已在 "parent or current/child" 范围中表示其他内容了

变量声明隐藏了本应在范围内的另一个声明。 请重命名在生成 CS0136 的行上声明的变量。

示例

下面的示例生成 CS0136：

C#

```
// CS0136.cs
namespace MyNamespace
{
   public class MyClass
   {
      public static void Main()
      {
         int i = 0;
         {
            char i = 'a';   // CS0136, hides int i
         }
         i++;
      }
   }
}
```

来自 C# 语言规范：

局部变量声明空间和嵌套局部变量声明空间不能包含同名的元素。 因此在嵌套声明空间中，不可能在封闭声明空间中声明与局部变量或常量同名的局部变量或常量。 只要声明空间都不包含另一个声明空间，两个声明空间都可包含同名的元素。

编译器错误 CS0139

项目 • 2023/04/07

没有要中断或继续的封闭循环

在循环外遇到了一个 break 或 continue 语句。

有关详细信息，请参阅 break 和 continue 语句。

下面的示例生成 CS0139 两次：

C#

```
// CS0139.cs
namespace x
{
   public class a
   {
      public static void Main()
      {
         continue;  // CS0139
         break;     // CS0139
      }
   }
}
```

编译器错误 CS0140

项目 • 2023/04/07

标签“label”重复

具有相同名称的标签出现了两次。 有关详细信息，请参阅 goto。

下面的示例生成 CS0140：

C#

```
// CS0140.cs
namespace MyNamespace
{
   public class MyClass
   {
      public static void Main()
      {
         label1: int i = 0;
         label1: int j = 0;   // CS0140, comment this line to resolve
         goto label1;
      }
   }
}
```

编译器错误 CS0143

项目 • 2023/04/07

类型“class”未定义构造函数

没有合适的构造函数可用。 这是内置数值类型情况，只需通过向它们赋值即可初始化。

下面的示例生成 CS0143：

C#

```
// CS0143.cs
class MyClass
{
   static public void Main ()
   {
      double d = new double(4.5);   // CS0143
      // Try this line instead:
      // double d = 4.5;
   }
}
```

编译器错误 CS0144

项目 • 2023/04/07

无法创建抽象类或接口“interface”的实例

无法创建 abstract 类或 interface的实例。 有关详细信息，请参阅接口。

下面的示例生成 CS0144：

C#

```
// CS0144.cs
interface MyInterface
{
}
public class MyClass
{
   public static void Main()
   {
      MyInterface myInterface = new MyInterface ();   // CS0144
   }
}
```

如何解决冲突

可以通过实现以下两个解决方案之一来解决此问题：

1. 更改类型声明，使它不那么抽象：从类声明中删除 abstract 关键字，或将类型从接口更改为类。

2. 创建一个派生自抽象类或用于实现接口的类型。

编译器错误 CS0145

项目 • 2023/04/07

常量字段要求提供一个值

必须初始化 const 变量。 有关详细信息，请参阅常量。

以下示例生成 CS0145：

C#

```
// CS0145.cs
class MyClass
{
   const int i;   // CS0145
   // try the following line instead
   // const int i = 0;

   public static void Main()
   {
   }
}
```

编译器错误 CS0146

项目 • 2023/04/07

涉及“class1”和“class2”的循环基类依赖项

类的继承列表包括对其自身的直接或间接引用。 类不能从其自身继承。 有关详细信息，请参阅继承。

下面的示例生成 CS0146：

C#

```
// CS0146.cs
namespace MyNamespace
{
   public interface InterfaceA
   {
   }

   public class MyClass : InterfaceA, MyClass2
   {
      public void Main()
      {
      }
   }

   public class MyClass2 : MyClass   // CS0146
   {
   }
}
```

编译器错误 CS0148

项目 • 2023/05/10

委托“delegate”没有有效的构造函数

你导入并使用了用另一编译器创建的托管程序（即使用 .NET 运行时的托管程序）。 该编译器允许了格式错误的 delegate 构造函数。 有关详细信息，请参阅委托。

编译器错误 CS0149

项目 • 2023/04/07

应输入方法名称

在创建 委托时，指定一个方法。 有关详细信息，请参阅委托。

以下示例生成 CS0149：

C#

```
// CS0149.cs
using System;

delegate string MyDelegate(int i);

class MyClass
{
   // class member-field of the declared delegate type
   static MyDelegate dt;

   public static void Main()
   {
      dt = new MyDelegate(17.45);   // CS0149
      // try the following line instead
      // dt = new MyDelegate(Func2);
      F(dt);
   }

   public static string Func2(int j)
   {
      Console.WriteLine(j);
      return j.ToString();
   }

   public static void F(MyDelegate myFunc)
   {
      myFunc(8);
   }
}
```

编译器错误 CS0150

项目 • 2023/04/07

应输入常量值

在应发现常量的位置处发现变量。 有关更多信息，请参见 switch。

以下示例生成 CS0150：

C#

```
// CS0150.cs
namespace MyNamespace
{
   public class MyClass
   {
      public static void Main()
      {
         int i = 0;
         int j = 0;

         switch(i)
         {
            case j:   // CS0150, j is a variable int, not a constant int
            // try the following line instead
            // case 0:
         }
      }
   }
}
```

当数组大小由变量值指定并由数组初始值设定项初始化时，也会产生此错误。 若要删除该错误，在单独语句或多个语句中初始化此数组。

C#

```
// CS0150.cs
    namespace MyNamespace
    {
        public class MyClass
        {
            public static void Main()
            {
                int size = 2;
                double[] nums = new double[size] { 46.9, 89.4 }; //CS0150
                // Try the following lines instead
                // double[] nums = new double[size];
                // nums[0] = 46.9;
                // nums[1] = 89.4;
```

```
            }
        }

    }
```

编译器错误 CS0151

项目 • 2023/04/07

应为整型类型的值

在需要整型数据类型的情况中使用了变量。 有关详细信息，请参阅类型。

不明确转换的示例

在没有转换或者可用的隐式转换导致了不明确的情况时，会出现此错误。 下面的示例生成 CS0151：

C#

```
public class MyClass
{
   public static implicit operator int (MyClass aa)
   {
      return 0;
   }

   public static implicit operator long (MyClass aa)
   {
      return 0;
   }

   public static void Main()
   {
      MyClass a = new MyClass();

      // Compiler cannot choose between int and long.
      switch (a)   // CS0151
      // try the following line instead
      // switch ((int)a)
      {
         case 1:
            break;
      }
   }
}
```

编译器错误 CS0152

项目 • 2023/04/07

标签“label”已经出现在该 switch 语句中

标签在 switch 语句中重复。

以下示例生成 CS0152：

C#

```
// CS0152.cs
namespace MyNamespace
{
   public class MyClass
   {
      public static void Main()
      {
         int i = 0;

         switch (i)
         {
            case 1:
               i++;
               return;

            case 1:   // CS0152, two case 1 statements
               i++;
               return;
         }
      }
   }
}
```

编译器错误 CS0153

项目 • 2023/04/07

goto case 只在 switch 语句中有效

尝试在 `switch` 语句外部使用 switch statement 语法。

下面的示例生成 CS0153：

C#

```
// CS0153.cs
public class a
{
   public static void Main()
   {
      goto case 5;   // CS0153
   }
}
```

编译器错误 CS0154

项目 • 2023/04/07

属性或索引器“property”无法用在此上下文中，因为它缺少 get 访问器

尝试使用 属性 失败，因为该属性中未定义 get 访问器方法。 有关详细信息，请参阅字段。

示例

下面的示例生成 CS0154：

C#

```
// CS0154.cs
public class MyClass2
{
    public int i
    {
        set
        {
        }
        // uncomment the get method to resolve this error
        /*
        get
        {
            return 0;
        }
        */
    }
}

public class MyClass
{
    public static void Main()
    {
        MyClass2 myClass2 = new MyClass2();
        int j = myClass2.i;   // CS0154, no get method
    }
}
```

编译器错误 CS0155

项目 • 2023/04/07

捕获或抛弃的类型必须从 System.Exception 派生

尝试将不是派生自 **System.Exception** 的数据类型传递到 catch 块中。 只有派生自 **System.Exception** 的数据类型才能传递到 **catch** 块中。 有关详细信息，请参阅异常和异常处理。

下面的示例生成 CS0155：

C#

```
// CS0155.cs
using System;

namespace MyNamespace
{
    public class MyClass2
    // try the following line instead
    // public class MyClass2 : Exception
    {
    }
    public class MyClass
    {
        public static void Main()
        {
            try
            {
            }
            catch (MyClass2)   // CS0155, resolves if you derive MyClass2
from Exception
            {
            }
        }
    }
}
```

编译器错误 CS0156

项目 • 2023/04/23

无参数的 throw 语句不允许在 catch 子句之外使用

不带参数的 throw 语句只能出现在不带任何参数的 **catch** 子句中。

有关详细信息，请参阅异常和异常处理。

下面的示例生成 CS0156：

C#

```
// CS0156.cs
using System;

namespace MyNamespace
{
   public class MyClass2 : Exception
   {
   }

   public class MyClass
   {
      public static void Main()
      {
         try
         {
            throw;   // CS0156
         }

         catch(MyClass2)
         {
            throw;   // this throw is valid
         }
      }
   }
}
```

编译器错误 CS0157

项目 • 2023/04/07

控制不能离开 finally 子句主体

finally 子句中的所有语句都必须执行。 有关详细信息，请参阅异常和异常处理。

下面的示例生成 CS0157：

C#

```
// CS0157.cs
using System;
namespace MyNamespace
{
   public class MyClass2 : Exception
   {
   }

   public class MyClass
   {
      public static void Main()
      {
         try
         {
         }
         finally
         {
            return;   // CS0157, cannot leave finally clause
         }
      }
   }
}
```

编译器错误 CS0158

项目 • 2023/04/07

在包含的范围中标签“label”遮盖了具有同样名称的另一个标签

内层作用域中的标签将隐藏外层作用域中具有相同名称的标签。 有关详细信息，请参阅 goto。

以下示例生成 CS0158：

C#

```
// CS0158.cs
namespace MyNamespace
{
   public class MyClass
   {
      public static void Main()
      {
         goto lab1;
         lab1:
         {
            lab1:
            goto lab1;   // CS0158
         }
      }
   }
}
```

编译器错误 CS0159

项目 • 2023/04/07

goto 语句范围内没有“label”这样的标签

在 语句范围内找不到由 goto `goto` 语句引用的标签。

以下示例生成 CS0159：

C#

```
// CS0159.cs
public class Class1
{
   public static void Main()
   {
      int i = 0;

      switch (i)
      {
         case 1:
            goto case 3;   // CS0159, case 3 label does not exist
         case 2:
            break;
      }
      goto NOWHERE;   // CS0159, NOWHERE label does not exist
   }
}
```

编译器错误 CS0160

项目 • 2024/03/04

上一个 catch 子句已经捕获了此类型或超类型（“type”）的所有异常

一系列 catch 子句必须按派生程度降序排列。 例如，派生程度最高的对象必须显示在最前。

有关详细信息，请参阅异常和异常处理。

下面的示例生成 CS0160：

C#

```
// CS0160.cs
public class MyClass2 : System.Exception {}
public class MyClass
{
   public static void Main()
   {
      try {}

      catch(System.Exception) {}   // Second-most derived; should be second
catch
      catch(MyClass2) {}   // CS0160  Most derived; should be first catch
   }
}
```


在 GitHub 上与我们协作

可以在 GitHub 上找到此内容的源，还可以在其中创建和查看问题和拉取请求。 有关详细信息，请参阅参与者指南。

.NET 反馈

.NET 是一个开放源代码项目。 选择一个链接以提供反馈：

提出文档问题

提供产品反馈

编译器错误 CS0161

项目 • 2023/04/07

“method”：并非所有的代码路径都返回值

返回一个值的方法必须在全部代码路径中具有 `return` 语句。 有关详细信息，请参阅方法。

示例

以下示例生成 CS0161：

C#

```
// CS0161.cs
public class Test
{
    public static int Main() // CS0161
    {
        int i = 5;
        if (i < 10)
        {
            return i;
        }
        else
        {
            // Uncomment the following line to resolve.
            // return 1;
        }
    }
}
```

编译器错误 CS0163

项目 • 2023/04/07

控件不能从一个 case 标签（“label”）贯穿到另一个 case 标签

当 switch 语句包含多个开关部分时，必须使用以下关键字之一明确终止每一部分（包括最后一部分）：

- return
- goto
- break
- throw

如果要实现从一部分到下一部分的“贯穿”行为，请使用 `goto case #`。

下面的示例生成 CS0163。

C#

```
// CS0163.cs
public class MyClass
{
    public static void Main()
    {
        int i = 0;

        switch (i)   // CS0163
        {
            // Compiler error CS0163 is reported on the following line.
            case 1:
                i++;
            // To resolve the error, uncomment one of the following example
statements.
            // return;
            // break;
            // goto case 3;

            case 2:
                i++;
                return;

            case 3:
                i = 0;
                return;

            // Compiler error CS0163 is reported on the following line.
            default:
                Console.WriteLine("Default");
                // To resolve the error, uncomment the following line:
```

```
            //break;
        }
    }
```

编译器错误 CS0165

项目 • 2023/04/07

使用了未赋值的局部变量“name”

C# 编译器不允许使用取消初始化的变量。 如果编译器检测到所使用的某个变量可能尚未初始化，就会生成 编译错误 CS0165。 有关详细信息，请参阅字段。 如果编译器遇到可能会导致使用未赋值变量的构造，即使特定代码不使用该变量，也会生成此错误。 这样可以避免对明确赋值使用过度复杂的规则。

有关详细信息，请参阅递归 lambda 为何一定会导致赋值错误？ 。

示例 1

下面的示例生成 CS0165：

C#

```
// CS0165.cs
using System;

class MyClass
{
    public int i;
}

class MyClass2
{
    public static void Main(string[] args)
    {
        // i and j are not initialized.
        int i, j;

        // You can provide a value for args[0] in the 'Command line
arguments'
        // text box on the Debug tab of the project Properties window.
        if (args[0] == "test")
        {
            i = 0;
        }
        // If the following else clause is absent, i might not be
        // initialized.
        //else
        //{
        //    i = 1;
        //}

        // Because i might not have been initialized, the following
```

```
        // line causes CS0165.
        j = i;

        // To resolve the error, uncomment the else clause of the previous
        // if statement, or initialize i when you declare it.

        // The following example causes CS0165 because myInstance is
        // declared but not instantiated.
        MyClass myInstance;
        // The following line causes the error.
        myInstance.i = 0;

        // To resolve the error, replace the previous declaration with
        // the following line.
        //MyClass myInstance = new MyClass();
    }
}
```

示例 2

编译器错误 CS0165 可能发生在递归委托定义中。 若要避免此错误，可在两个语句中定义委托，以使变量在初始化之前不会被投入使用。 下面的示例演示此错误和解决方案。

C#

```
class Program
{
    delegate void Del();
    static void Main(string[] args)
    {
        // The following line causes CS0165 because variable d is used
        // as an argument before it has been initialized.
        Del d = delegate() { System.Console.WriteLine(d); };

        //// To resolve the error, initialize d in a separate statement.
        //Del d = null;
        //// After d is initialized, you can use it as an argument.
        //d = delegate() { System.Console.WriteLine(d); };
        //d();
    }
}
```

Load(String)

编译器错误 CS0167

项目 • 2023/05/09

委托“delegate”缺少 Invoke 方法

你导入并使用了用另一编译器创建的一个托管程序（即使用 .NET 运行时的托管程序）。 此编译器允许了格式错误的 delegate。 因此， `Invoke` 方法不可用。 有关详细信息，请参阅委托。

编译器错误 CS0170

项目 • 2023/04/07

使用了可能未赋值的字段“field”

结构中的字段在使用前未初始化。 若要解决此问题，首先确定哪个字段尚未初始化，然后将其初始化，再尝试访问该字段。 有关初始化结构的详细信息，请参阅结构类型。

下面的示例生成 CS0170：

C#

```
// CS0170.cs
public struct error
{
   public int i;
}

public class MyClass
{
   public static void Main()
   {
      error e;
      // uncomment the next line to resolve this error
      // e.i = 0;
      System.Console.WriteLine( e.i );   // CS0170 because
                                         //e.i was never assigned
   }
}
```

编译器错误 CS0172

项目 • 2023/04/07

无法确定条件表达式的类型，因为“type1”和“type2”可相互隐式转换

在条件语句中，必须能够隐式转换 `:` 标记左右两侧的类型。 此外，不能有相互隐式转换；只需一次转换。

下面的示例生成 CS0172：

C#

```
// CS0172.cs
public class Square
{
   public class Circle
   {
      public static implicit operator Circle(Square aa)
      {
         return null;
      }

      public static implicit operator Square(Circle aa)
      // using explicit resolves this error
      // public static explicit operator Square(Circle aa)
      {
         return null;
      }
   }

   public static void Main()
   {
      Circle aa = new Circle();
      Square ii = new Square();
      object o = (1 == 1) ? aa : ii;   // CS0172
      // the following cast would resolve this error
      // (1 == 1) ? aa : (Circle)ii;
   }
}
```

另请参阅

- 用户定义转换运算符

编译器错误 CS0173

项目 • 2023/06/08

无法确定条件表达式的类型，因为“class1”和“class2”之间没有隐式转换

希望不同类的对象使用相同代码时，类之间的转换非常有用。 但是，协同工作的两个类不能有相互转换和多余转换，也不能有隐式转换。 `class1` 和 `class2` 的类型是独立确定的，选择更全面的类型作为条件表达式的类型。 有关类型确定方式的详细信息，请参阅条件运算符不能隐式转换。

若要解决 CS0173，请确保 `class1` 和 `class2` 之间有且仅有一个隐式转换，而不论向哪个方向进行转换或在哪个类中进行转换。 有关详细信息，请参阅用户定义转换运算符和内置数值转换。

示例

下面的示例生成编译器错误 CS0173：

C#

```
public class C {}

public class A
{
    // The following code defines an implicit conversion operator from
    // type C to type A.
    //public static implicit operator A(C c)
    //{
    //    A a = new A();
    //    a = c;
    //    return a;
    //}
}

public class MyClass
{
    public static void F(bool b)
    {
        A a = new A();
        C c = new C();

        // The following line causes CS0173 because there is no implicit
        // conversion from a to c or from c to a.
        object o = b ? a : c;

        // To resolve the error, you can cast a and c.
        // object o = b ? (object)a : (object)c;
```

```
        // Alternatively, you can add a conversion operator from class C to
        // class A, or from class A to class C, but not both.
    }

   public static void Main()
   {
      F(true);
   }
}
```

C#

```
class M
{
    static int Main ()
    {
        int X = 1;
        // The following line causes CS0173.
        object o = (X == 0) ? null : null;
        return -1;
    }
}
```

编译器错误 CS0174

项目 • 2023/05/10

"base" 引用需要基类

仅当 .NET 运行时编译 `System.Object` 类的源代码，而该类是唯一没有基类的类时，才发生该错误。

编译器错误 CS0175

项目 • 2023/04/07

在此上下文中使用关键字 "base" 无效

必须使用 base 关键字指定基类的特定成员。 有关详细信息，请参阅构造函数。

下面的示例生成 CS0175：

C#

```
// CS0175.cs
using System;
class BaseClass
{
    public int TestInt = 0;
}

class MyClass : BaseClass
{
    public static void Main()
    {
        MyClass aClass = new MyClass();
        aClass.BaseTest();
    }

    public void BaseTest()
    {
        Console.WriteLine(base); // CS0175
        // Try the following line instead:
        // Console.WriteLine(base.TestInt);
        base = 9;   // CS0175

        // Try the following line instead:
        // base.TestInt = 9;
    }
}
```

编译器错误 CS0176

项目 • 2023/04/07

不能使用实例引用访问静态成员“成员”，而是使用类型名称来限定它

仅可使用类名限定 静态 变量；实例名称不能做为限定符。 有关详细信息，请参阅静态类和静态类成员。

下面的示例生成 CS0176：

C#

```
// CS0176.cs
public class MyClass2
{
    public static int num;
}

public class Test
{
    public static void Main()
    {
        MyClass2 mc2 = new MyClass2();
        int i = mc2.num;   // CS0176
        // try the following line instead
        // int i = MyClass2.num;
    }
}
```

编译器错误 CS0177

项目 • 2023/04/07

在控件离开当前方法之前，必须对 out 参数“parameter”赋值

未向标记有 out 关键字的参数分配方法主体中的值。 有关详细信息，请参阅传递参数

下面的示例生成 CS0177：

C#

```
// CS0177.cs
public class MyClass
{
   public static void Foo(out int i)   // CS0177
   {
   // uncomment the following line to resolve this error
   //   i = 0;
   }

   public static void Main()
   {
       int x = -1;
       Foo(out x);
   }
}
```

编译器错误 CS0179

项目 • 2023/04/07

“member”不能是外部的，也无法声明主体

当类成员标记为 外部，这意味着该成员的定义位于另一个文件中。 因此，标记为 **外部** 的类成员不能在类中定义。 删除 `extern` 关键字或删除该定义。 有关详细信息，请参阅 方法。

以下示例生成 CS0179：

C#

```
// CS0179.cs
public class MyClass
{
   public extern int ExternMethod(int aa)   // CS0179
   {
      return 0;
   }
   // try the following line instead
   // public extern int ExternMethod(int aa);

   public static void Main()
   {
   }
}
```

编译器错误 CS0180

项目 • 2023/04/07

“member”不能既是外部的又是抽象的

abstract 和 extern 关键字互相排斥。 `extern` 关键字表示在文件外部定义成员， **abstract** 意味着在派生类中提供了该实现。 有关详细信息，请参阅方法。

以下示例生成 CS0180：

C#

```
// CS0180.cs
namespace MyNamespace
{
   public class MyClass
   {
      public extern abstract int Foo(int a);   // CS0180

      public static void Main()
      {
      }
   }
}
```

编译器错误 CS0181

项目 • 2023/06/08

特性构造函数参数具有类型，这不是有效特性参数类型

示例

以下示例生成 CS0181：

C#

```
// CS0181.cs (12,6)

using System;
using System.Runtime.InteropServices;
[AttributeUsage(AttributeTargets.Method, AllowMultiple = true)]
unsafe class Attr : Attribute
{
    public Attr(delegate*<void> d) {}
}
unsafe class C
{
    [UnmanagedCallersOnly]
    [Attr(&M1)]
    static void M1()
    {
    }
}
```

CLR 当前不允许泛型特性参数类型，因此 C# 不允许泛型特性参数类型。

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

Open a documentation issue

Provide product feedback

编译器错误 CS0186

项目 • 2024/04/11

在此上下文中使用 null 无效

下面的示例生成 CS0186：

C#

```
// CS0186.cs
using System;
using System.Collections;

class MyClass
{
   static void Main()
   {
      // Each of the following lines generates CS0186:
      foreach (int i in null) {}   // CS0186
      foreach (int i in (IEnumerable) null) { };   // CS0186
   }
}
```


在 GitHub 上与我们协作

可以在 GitHub 上找到此内容的源，还可以在其中创建和查看问题和拉取请求。 有关详细信息，请参阅参与者指南。

.NET

.NET 反馈

.NET 是一个开放源代码项目。 选择一个链接以提供反馈：

提出文档问题

提供产品反馈

编译器错误 CS0191

项目 • 2023/04/07

无法对属性或索引器“name”赋值 -- 它是只读的

readonly 字段只能采用构造函数或声明中的一个赋值。 有关详细信息，请参阅构造函数。

如果 `readonly` 字段为 static 并且构造函数没有被标记为 `static`，也会导致 CS0191。

示例

下面的示例生成 CS0191。

C#

```
// CS0191.cs
class MyClass
{
    public readonly int TestInt = 6;  // OK to assign to readonly field in
declaration

    MyClass()
    {
        TestInt = 11; // OK to assign to readonly field in constructor
    }

    public void TestReadOnly()
    {
        TestInt = 19;                  // CS0191
    }

    public static void Main()
    {
    }
}
```

编译器错误 CS0193

项目 • 2023/04/07

* 或 -> 运算符只能应用于指针

* 或 -> 运算符曾用于非指针类型。 有关详细信息，请参阅指针类型。

下面的示例生成 CS0193：

C#

```
// CS0193.cs
using System;

public struct Age
{
   public int AgeYears;
   public int AgeMonths;
   public int AgeDays;
}

public class MyClass
{
   public static void SetAge(ref Age anAge, int years, int months, int days)
   {
      anAge->Months = 3;   // CS0193, anAge is not a pointer
      // try the following line instead
      // anAge.AgeMonths = 3;
   }

   public static void Main()
   {
      Age MyAge = new Age();
      Console.WriteLine(MyAge.AgeMonths);
      SetAge(ref MyAge, 22, 4, 15);
      Console.WriteLine(MyAge.AgeMonths);
   }
}
```

编译器错误 CS0196

项目 • 2023/04/07

指针必须只根据一个值进行索引

指针不能有多个索引。 有关详细信息，请参阅指针类型

下面的示例生成 CS0196：

C#

```
// CS0196.cs
public class MyClass
{
   public static void Main ()
   {
      int *i = null;
      int j = 0;
      j = i[1,2];   // CS0196
      // try the following line instead
      // j = i[1];
   }
}
```

编译器错误 CS0198

项目 • 2023/04/07

无法给静态只读字段“name”的字段赋值（静态构造函数或变量初始值设定项中除外）

readonly 变量必须与构造函数具有相同的 静态 用法，在此构造函数中你要对其进行初始化。 有关详细信息，请参阅静态构造函数。

下面的示例生成 CS0198：

C#

```
// CS0198.cs
class MyClass
{
   public static readonly int TestInt = 6;

   MyClass()
   {
      TestInt = 11;   // CS0198, constructor is not static and readonly
field is
   }

   public static void Main()
   {
   }
}
```

编译器错误 CS0200

项目 • 2024/10/26

无法对属性或索引器“property”赋值 - 它是只读的

尝试给 属性赋值，但该属性没有 set 访问器，或者是在构造函数之外赋值的。 通过添加 set 访问器解决该错误。 有关详细信息，请参阅如何声明和使用读/写属性。

示例

以下示例生成 CS0200：

C#

```
// CS0200.cs
public class Example
{
    private int _mi;
    int I
    {
        get
        {
            return _mi;
        }
        // uncomment the set accessor and declaration for _mi
        /*
        set
        {
            _mi = value;
        }
        */
    }

    public static void Main()
    {
        Example example = new Example();
        example.I = 9;   // CS0200
    }
}
```

以下示例使用 自动实现的属性 和 对象初始值设定项，但仍生成 CS0200：

C#

```
// CS0200.cs
public class Example
{
    int I
```

```
    {
        get;
        // uncomment the set accessor and declaration
        //set;
    }

    public static void Main()
    {
        var example = new Example
        {
            I = 9   // CS0200
        };
    }
}
```

若要对只读的属性或索引器“property”赋值，请添加 set 访问器或在对象的构造函数中赋值。

C#

```
public class Example
{
    int I { get; }

    public Example()
    {
        I = -7;
    }
}
```

编译器错误 CS0201

项目 • 2023/04/07

只有 assignment、call、increment、decrement 和 new 对象表达式可用作语句

编译器在遇到无效语句时会生成错误。 无效语句是以不表示以下各项的分号结尾的任意行或一系列行：赋值 (=)、方法调用 ()、new、-- 或 ++ 运算。 有关详细信息，请参阅语句和运算符和表达式。

示例 1

下面的示例生成 CS0201，因为 2 * 3 是表达式，而不是语句。 若要使代码能够编译，请尝试将表达式的值赋给变量。

C#

```
// CS0201.cs
public class MainClass
{
   public static void Main()
   {
      2 * 3;   // CS0201
      // Try the following line instead.
      //   int i = 2 * 3;
   }
}
```

示例 2

下面的示例生成 CS0201，因为 checked 本身并不是语句，即使已通过递增操作对其进行了参数化。

C#

```
// CS0201_b.cs
// compile with: /target:library
public class MyList<T>
{
   public void Add(T x)
   {
      int i = 0;
      if ( (object)x == null)
      {
         checked(i++);   // CS0201
```

```
            // OK
            checked {
                i++;
            }
        }
    }
}
```

另请参阅

- C# 编译器错误

编译器错误 CS0202

项目 • 2023/04/07

foreach 要求 "type.GetEnumerator()" 的返回类型“type”必须具有适当的公共 MoveNext 方法和公共 Current 属性

用于启用 foreach 语句的函数 GetEnumerator 无法返回指针或数组；它必须返回能够充当枚举器的类的实例。 充当枚举器的适当要求包括一个公共 Current 属性和一个公共 MoveNext 方法。

> **备注**
>
> 在 C# 2.0 中，编译器将自动为你生成 Current 和 MoveNext。 有关详细信息，请参阅**泛型接口**中的代码示例。

下面的示例生成 CS0202：

C#

```
// CS0202.cs

public class C1
{
   public int Current
   {
      get
      {
         return 0;
      }
   }

   public bool MoveNext ()
   {
      return false;
   }

   public static implicit operator C1 (int c1)
   {
      return 0;
   }
}

public class C2
{
   public int Current
   {
      get
      {
```

```
            return 0;
        }
    }

    public bool MoveNext ()
    {
        return false;
    }

    public C1[] GetEnumerator ()
    // try the following line instead
    // public C1 GetEnumerator ()
    {
        return null;
    }
}

public class MainClass
{
    public static void Main ()
    {
        C2 c2 = new C2();

        foreach (C1 x in c2)   // CS0202
        {
            System.Console.WriteLine(x.Current);
        }
    }
}
```

编译器错误 CS0204

项目 • 2024/04/11

只允许 65534 个局部变量

在方法中超过了 65534 个局部变量的限制。

在 GitHub 上与我们协作

可以在 GitHub 上找到此内容的源，还可以在其中创建和查看问题和拉取请求。 有关详细信息，请参阅参与者指南。

.NET

.NET 反馈

.NET 是一个开放源代码项目。 选择一个链接以提供反馈：

提出文档问题

提供产品反馈

编译器错误 CS0205

项目 • 2023/04/07

无法调用抽象基成员：“method”

无法调用 abstract 方法，因为没有方法正文。 有关详细信息，请参阅抽象类、密封类及类成员。

下面的示例生成 CS0205：

C#

```
// CS0205.cs
abstract public class MyClass
{
   abstract public void M();
}

public class MyClass2 : MyClass
{
   public override void M()
   {
      base.M();   // CS0205, delete this line
   }

   public static void Main()
   {
   }
}
```

编译器错误 CS0208

项目 • 2023/04/07

无法获取托管类型（“类型”）的地址和大小，也无法声明指向它的指针

即使使用 unsafe 关键字，也不允许获取托管对象的地址和大小或声明指向托管类型的指针。 托管类型是：

- 任何引用类型
- 包含作为字段或属性的引用类型的任何结构

有关详细信息，请参阅非托管类型。

示例

下面的示例生成 CS0208:

C#

```
// CS0208.cs
// compile with: /unsafe

class myClass
{
    public int a = 98;
}

struct myProblemStruct
{
    string s;
    float f;
}

struct myGoodStruct
{
    int i;
    float f;
}

public class MyClass
{
    unsafe public static void Main()
    {
        // myClass is a class, a managed type.
        myClass s = new myClass();
        myClass* s2 = &s;    // CS0208

        // The struct contains a string, a managed type.
```

```
        int i = sizeof(myProblemStruct); //CS0208

        // The struct contains only value types.
        i = sizeof(myGoodStruct); //OK

    }
}
```

编译器错误 CS0209

项目 • 2023/04/07

fixed 语句中声明的局部变量类型必须是指针类型

fixed 语句 中声明的变量必须是指针。 有关详细信息，请参阅不安全代码和指针。

下面的示例生成 CS0209：

C#

```
// CS0209.cs
// compile with: /unsafe

class Point
{
   public int x, y;
}

public class MyClass
{
   unsafe public static void Main()
   {
      Point pt = new Point();

      fixed (int i)    // CS0209
      {
      }
      // try the following lines instead
      /*
      fixed (int* p = &pt.x)
      {
      }
      fixed (int* q = &pt.y)
      {
      }
      */
   }
}
```

编译器错误 CS0210

项目 • 2024/06/12

必须在 fixed 或 `using` 语句声明中提供初始值设定项

必须声明并初始化在 fixed 语句中的变量。 有关详细信息，请参阅不安全代码和指针。

以下示例生成 CS0210：

C#

```
// CS0210a.cs
// compile with: /unsafe

class Point
{
   public int x, y;
}

public class MyClass
{
   unsafe public static void Main()
   {
      Point pt = new Point();

      fixed (int i)    // CS0210
      {
      }
      // try the following lines instead
      /*
      fixed (int* p = &pt.x)
      {
      }
      fixed (int* q = &pt.y)
      {
      }
      */
   }
}
```

以下示例也生成 CS0210，因为 using 语句没有初始值设定项。

C#

```
// CS0210b.cs

using System.IO;
class Test
{
   static void Main()
```

```
    {
        using (StreamWriter w) // CS0210
        // Try this line instead:
        // using (StreamWriter w = new StreamWriter("TestFile.txt"))
        {
            w.WriteLine("Hello there");
        }
    }
}
```

在 GitHub 上与我们协作

可以在 GitHub 上找到此内容的源，还可以在其中创建和查看问题和拉取请求。 有关详细信息，请参阅参与者指南。

.NET 反馈

.NET 是一个开放源代码项目。 选择一个链接以提供反馈：

提出文档问题

提供产品反馈

编译器错误 CS0211

项目 • 2023/04/07

无法获取给定表达式的地址

可以采用字段、局部变量和指针间接寻址的地址，但不能采用两个局部变量之和的地址等。 有关详细信息，请参阅不安全代码和指针。

下面的示例生成 CS0211：

C#

```
// CS0211.cs
// compile with: /unsafe

public class MyClass
{
   unsafe public void M()
   {
      int a = 0, b = 0;
      int *i = &(a + b);   // CS0211, the addition of two local variables
      // try the following line instead
      // int *i = &a;
   }

   public static void Main()
   {
   }
}
```

编译器错误 CS0212

项目 • 2023/04/07

只能获取 fixed 语句初始值设定项内的未固定表达式的地址

有关详细信息，请参阅不安全代码和指针。

下面的示例演示了如何采用非固定表达式的地址。 下面的示例生成 CS0212。

C#

```
// CS0212a.cs
// compile with: /unsafe /target:library

public class A {
   public int iField = 5;

   unsafe public void M() {
      A a = new A();
      int* ptr = &a.iField;   // CS0212
   }

   // OK
   unsafe public void M2() {
      A a = new A();
      fixed (int* ptr = &a.iField) {}
   }
}
```

下面的示例也生成 CS0212，并演示了如何解决此错误：

C#

```
// CS0212b.cs
// compile with: /unsafe /target:library
using System;

public class MyClass
{
   unsafe public void M()
   {
      // Null-terminated ASCII characters in an sbyte array
      sbyte[] sbArr1 = new sbyte[] { 0x41, 0x42, 0x43, 0x00 };
      sbyte* pAsciiUpper = &sbArr1[0];   // CS0212
      // To resolve this error, delete the previous line and
      // uncomment the following code:
      // fixed (sbyte* pAsciiUpper = sbArr1)
      // {
      //    String szAsciiUpper = new String(pAsciiUpper);
      // }
```

```
    }
}
```

编译器错误 CS0213

项目 • 2023/04/07

不能使用 fixed 语句来获取已固定的表达式的地址

unsafe 方法中的局部变量或某个参数已固定（在堆栈上），因此无法在 fixed 表达式中获取这两个变量的任何一个的地址。 有关详细信息，请参阅不安全代码和指针。

示例

下面的示例生成 CS0213。

C#

```
// CS0213.cs
// compile with: /unsafe
public class MyClass
{
   unsafe public static void Main()
   {
      int i = 45;
      fixed (int *j = &i) { }  // CS0213
      // try the following line instead
      // int* j = &i;

      int[] a = new int[] {1,2,3};
      fixed (int *b = a)
      {
         fixed (int *c = b) { }  // CS0213
         // try the following line instead
         // int *c = b;
      }
   }
}
```

编译器错误 CS0214

项目 • 2023/04/07

指针和固定大小缓冲区只能在不安全的上下文中使用

指针只能通过 unsafe 关键字使用。 有关详细信息，请参阅不安全代码和指针。

下面的示例生成 CS0214：

C#

```
// CS0214.cs
// compile with: /target:library /unsafe
public struct S
{
   public int a;
}

public class MyClass
{
   public static void Test()
   {
      S s = new S();
      S * s2 = &s;    // CS0214
      s2->a = 3;      // CS0214
      s.a = 0;
   }

   // OK
   unsafe public static void Test2()
   {
      S s = new S();
      S * s2 = &s;
      s2->a = 3;
      s.a = 0;
   }
}
```

编译器错误 CS0215

项目 • 2023/04/07

运算符 True 或 False 的返回类型必须是 bool

用户定义的 true 和 false 运算符必须具有 bool的返回类型。

以下示例生成 CS0215：

C#

```
// CS0215.cs
class MyClass
{
   public static int operator true (MyClass MyInt)   // CS0215
   // try the following line instead
   // public static bool operator true (MyClass MyInt)
   {
      return true;
   }

   public static int operator false (MyClass MyInt)   // CS0215
   // try the following line instead
   // public static bool operator false (MyClass MyInt)
   {
      return true;
   }

   public static void Main()
   {
   }
}
```

编译器错误 CS0216

项目 • 2023/04/07

运算符“operator”要求也要定义匹配的运算符“missing_operator”

用户定义的 == 运算符需要用户定义的 != 运算符，反之亦然。
这同样适用于用户定义的 true 运算符和用户定义的 false 运算符。

下面的示例生成 CS0216：

C#

```
// CS0216.cs
class MyClass
{
   public static bool operator == (MyClass MyIntLeft, MyClass MyIntRight)
// CS0216
   {
      return MyIntLeft == MyIntRight;
   }

   // to resolve, uncomment the following operator definition
   /*
   public static bool operator != (MyClass MyIntLeft, MyClass MyIntRight)
   {
      return MyIntLeft != MyIntRight;
   }
   */

   public override bool Equals (object obj)
   {
      return base.Equals (obj);
   }

   public override int GetHashCode()
   {
      return base.GetHashCode();
   }

   public static void Main()
   {
   }
}
```

编译器错误 CS0217

项目 • 2023/04/07

为了可以像短路运算符一样应用，用户定义的逻辑运算符 (“operator”) 的返回类型必须与它的两个参数的类型相同。

如果为用户定义的类型定义运算符，然后尝试将运算符用作短路运算符，则用户定义的运算符必须具有类型相同的参数和返回值。 有关短路运算符的详细信息，请参阅 && 运算符和 || 运算符。 有关用户定义的短路或条件操作符的详细信息，请参阅 C# 语言规范的用户定义条件逻辑运算符部分。

以下示例生成CS0217：

C#

```
// CS0217.cs
using System;

public class MyClass
{
   public static bool operator true (MyClass f)
   {
      return false;
   }

   public static bool operator false (MyClass f)
   {
      return false;
   }

   public static implicit operator int(MyClass x)
   {
      return 0;
   }

   public static int operator & (MyClass f1, MyClass f2)   // CS0217
   // try the following line instead
   // public static MyClass operator & (MyClass f1, MyClass f2)
   {
      return new MyClass();
   }

   public static void Main()
   {
      MyClass f = new MyClass();
      int i = f && f;
   }
}
```

另请参阅

- 运算符重载
- true 和 false 运算符

编译器错误 CS0218

项目 • 2023/04/07

类型（“type”）必须包含运算符 true 和运算符 false 的声明

如果用户定义类型重载 & 运算符或 | 运算符，则它还必须定义 true 和 false 运算符，从而定义短路 && 运算符或 || 运算符。

以下示例生成 CS0218：

C#

```
// CS0218.cs
using System;
public class MyClass
{
   // uncomment these operator declarations to resolve this CS0218
   /*
   public static bool operator true (MyClass f)
   {
      return false;
   }

   public static bool operator false (MyClass f)
   {
      return false;
   }
   */

   public static implicit operator int(MyClass x)
   {
      return 0;
   }

   public static MyClass operator & (MyClass f1, MyClass f2)
   {
      return new MyClass();
   }

   public static void Main()
   {
      MyClass f = new MyClass();
      int i = f && f;   // CS0218, requires operators true and false
   }
}
```

另请参阅

- 运算符重载

编译器错误 CS0220

项目 • 2023/04/07

在 checked 模式下，运算在编译时溢出

checked（这是常数表达式的默认值）检测到一个操作，结果导致数据丢失。 更正赋值的输入或使用 未检查 来解决此错误。 有关详细信息，请参阅 checked 和 unchecked 语句一文。

以下示例生成 CS0220：

C#

```
// CS0220.cs
using System;

class TestClass
{
   const int x = 1000000;
   const int y = 1000000;

   public int MethodCh()
   {
      int z = (x * y);   // CS0220
      return z;
   }

   public int MethodUnCh()
   {
      unchecked
      {
         int z = (x * y);
         return z;
      }
   }

   public static void Main()
   {
      TestClass myObject = new TestClass();
      Console.WriteLine("Checked   : {0}", myObject.MethodCh());
      Console.WriteLine("Unchecked: {0}", myObject.MethodUnCh());
   }
}
```

编译器错误 CS0221

项目 • 2023/04/07

常量值“value”无法转换为“type”(使用“unchecked”语法重写)

checked（默认情况下针对常数表达式打开）检测到会导致数据丢失的赋值操作。 更正赋值或使用 unchecked 解决此错误。 有关详细信息，请参阅 checked 和 unchecked 语句一文。

下面的示例生成 CS0221：

C#

```
// CS0221.cs
public class MyClass
{
   public static void Main()
   {
      // unchecked
      // {
         int a = (int)0xFFFFFFFF;   // CS0221
         a++;
      // }
   }
}
```

编译器错误 CS0226

项目 • 2023/04/07

__arglist 表达式只能出现在调用或新的表达式内部。

不支持的关键字 `__arglist` 只能出现在方法调用或新的表达式中。

示例

以下代码生成 CS0226：

C#

```
// cs0226.cs
using System;

public class C
    {
    public static int Main ()
        {
        __arglist(1,"This is a string"); // CS0226
        return 0;
        }
    }
```

编译器错误 CS0227

项目 • 2023/04/07

不安全代码只会在使用 /unsafe 编译的情况下出现

如果源的代码包含 unsafe 关键字，则还必须使用 AllowUnsafeBlocks 编译器选项。 有关详细信息，请参阅不安全代码和指针。

若要在 Visual Studio 2012 中设置不安全选项，单击主菜单中的"项目"，选择"生成"窗格，然后勾选"允许不安全代码"框。

编译不含 /unsafe 时，下面的示例将生成 CS0227：

C#

```
// CS0227.cs
public class MyClass
{
   unsafe public static void Main()   // CS0227
   {
   }
}
```

另请参阅

- C# 编译器错误

编译器错误 CS0228

项目 • 2023/04/07

“type”不包含“member”的定义，或者它是不可访问的

可能已经替换了 **System.Hashtable**、 **System.String**或 **System.Array**的系统版本，并且此替换不包含 `member`。

否则，你可能需要修复或重新安装 Visual Studio。

编译器错误 CS0229

项目 • 2023/04/07

“member1”和“member2”之间具有二义性

不同接口的成员具有相同的名称。 若要保留相同的名称，必须限定这些名称。 有关详细信息，请参阅接口。

> **备注**
>
> 在某些情况下，通过 using 别名向标识符提供显式前缀，即可解决此二义性问题。

示例

下面的示例生成 CS0229：

C#

```
// CS0229.cs

interface IList
{
    int Count
    {
        get;
        set;
    }

    void Counter();
}

interface ICounter
{
    double Count
    {
        get;
        set;
    }
}

interface IListCounter : IList, ICounter {}

class MyClass
{
    void Test(IListCounter x)
    {
        x.Count = 1;  // CS0229
```

```
        // Try one of the following lines instead:
        // ((IList)x).Count = 1;
        // or
        // ((ICounter)x).Count = 1;
    }

    public static void Main() {}
}
```

编译器错误 CS0230

项目 • 2023/10/23

在 foreach 语句中，类型和标识符都是必需的

foreach 语句格式错误。

下面的示例生成 CS0230：

C#

```
// CS0230.cs
class MyClass
{
   public static void Main()
   {
      int[] myarray = new int[3] {1,2,3};

      foreach (int in myarray)   // CS0230
      {
         Console.WriteLine(x);
      }
   }
}
```

下面的示例提供相同的代码，但没有 CS0230 错误：

C#

```
class MyClass
{
   public static void Main()
   {
      int[] myarray = new int[3] {1,2,3};

      foreach (int x in myarray)  // Both type (int) and indentifier (x) are
specified
      {
         Console.WriteLine(x);
      }
   }
}
```

在 GitHub 上与我们协作

.NET feedback

可以在 GitHub 上找到此内容的源，还可以在其中创建和查看问题和拉取请求。 有关详细信息，请参阅参与者指南。

The .NET documentation is open source. Provide feedback here.

提出文档问题

提供产品反馈

编译器错误 CS0233

项目 • 2023/04/07

“identifier”没有预定义的大小，因此 sizeof 只能在不安全的上下文中使用

如果没有不安全的上下文，sizeof 运算符只能用于大小为编译时常数的类型。 如果发生此错误，请使用不安全的上下文。

下面的示例生成 CS0233：

C#

```
// CS0233.cs
using System;
using System.Runtime.InteropServices;

[StructLayout(LayoutKind.Sequential)]
public struct S
{
    public int a;
}

public class MyClass
{
    public static void Main()
    {
        S myS = new S();
        Console.WriteLine(sizeof(S));   // CS0233
        // Try the following instead:
        // unsafe
        // {
        //     Console.WriteLine(sizeof(S));
        // }
   }
}
```

编译器错误 CS0236

项目 • 2023/04/07

字段初始值设定项无法引用非静态字段、方法或属性“name”。

实例字段不能用于初始化方法之外的其他实例字段。

更正此错误

如果正在尝试初始化方法之外的变量，请考虑在类构造函数内执行此初始化。 有关详细信息，请参阅方法。

示例

下面的示例生成 CS0236，并演示如何修复此错误：

C#

```
public class MyClass
{
    public int i = 5;

    // To fix the error, remove "= i", and uncomment the line in
constructor.
    public int j = i;  // CS0236

    public MyClass()
    {
        // Uncomment the following.
        //j = i;
    }
}
```

编译器错误 CS0238

项目 • 2023/04/07

因为“member”不是重写，所以无法将其密封

对还未使用密封 进行标记的成员使用了 密封。 有关详细信息，请参阅继承。

下面的示例生成 CS0238：

C#

```
// CS0238.cs
abstract class MyClass
{
   public abstract void f();
}

class MyClass2 : MyClass
{
   public static void Main()
   {
   }

   public sealed void f() // CS0238
   // Try the following definition instead:
   // public override sealed void f()
   {
   }
}
```

编译器错误 CS0239

项目 • 2023/04/07

“member”：继承成员“inherited member”是密封的，无法进行重写

成员不能 重写 一个 密封的 继承成员。 有关详细信息，请参阅继承。

下面的示例生成 CS0239：

C#

```
// CS0239.cs
abstract class MyClass
{
   public abstract void f();
}

class MyClass2 : MyClass
{
   public static void Main()
   {
   }

   public override sealed void f()
   {
   }
}

class MyClass3 : MyClass2
{
   public override void f()   // CS0239
   // Try the following definition instead:
   // public new void f()
   {
   }
}
```

编译器错误 CS0241

项目 • 2023/04/07

不允许使用默认参数说明符

方法参数 不能有默认值。 如果要达到同样的效果，请使用方法重载。

示例

以下示例生成 CS0241。 此外，该示例演示如何使用重载模拟具有默认参数的方法。

C#

```
// CS0241.cs
public class A
{
   public void Test(int i = 9) {}   // CS0241
}

public class B
{
   public void Test() { Test(9); }
   public void Test(int i)  {}
}

public class C
{
   public static void Main()
   {
      B x = new B();
      x.Test();
   }
}
```

编译器错误 CS0242

项目 • 2023/04/07

相关操作在 void 指针上未定义

不允许递增 void 指针。 有关详细信息，请参阅不安全代码和指针。

下面的示例生成 CS0242：

C#

```
// CS0242.cs
// compile with: /unsafe
class TestClass
{
   public unsafe void Test()
   {
      void * p = null;
      p++;   // CS0242, incrementing a void pointer not allowed
   }

   public static void Main()
   {
   }
}
```

编译器错误 CS0243

项目 • 2023/04/07

Conditional 特性在“method”上无效，因为它是重写方法

在标记有 override 关键字的方法上不允许使用 ConditionalAttribute 特性。 有关详细信息，请参阅了解何时使用 Override 和 New 关键字。

编译器从不绑定到重写方法。 它只绑定到基方法，而且公共语言运行库在适当的时候调用重写。

以下代码生成 CS0243：

C#

```
// CS0243.cs
// compile with: /target:library
public class MyClass
{
   public virtual void M() {}
}

public class MyClass2 : MyClass
{
   [System.Diagnostics.ConditionalAttribute("MySymbol")]   // CS0243
   // remove Conditional attribute or remove override keyword
   public override void M() {}
}
```

编译器错误 CS0244

项目 • 2023/04/07

"is" 和 "as" 在指针类型上都无效

is 和 as 运算符在指针类型上的使用是无效的。 有关详细信息，请参阅不安全代码和指针。

以下示例生成 CS0244：

C#

```
// CS0244.cs
// compile with: /unsafe

class UnsafeTest
{
   unsafe static void SquarePtrParam (int* p)
   {
      bool b = p is object;   // CS0244 p is pointer
   }

   unsafe public static void Main()
   {
      int i = 5;
      SquarePtrParam (&i);
   }
}
```

编译器错误 CS0245

项目 • 2023/04/07

无法直接调用析构函数和 object.Finalize。 请考虑调用 IDisposable.Dispose（如果可用）。

有关详细信息，请参阅垃圾回收的编程要点和终结器。

下面的示例生成 CS0245：

C#

```
// CS0245.cs
using System;
using System.Collections;

class MyClass // : IDisposable
{
   /*
   public void Dispose()
   {
      // cleanup code goes here
   }
   */

   void m()
   {
      this.Finalize();   // CS0245
      // this.Dispose();
   }

   public static void Main()
   {
   }
}
```

编译器错误 CS0247

项目 • 2023/04/07

无法对 stackalloc 采用负值大小

负数传递到 stackalloc 语句。

下面的示例生成 CS0247：

C#

```
// CS0247.cs
// compile with: /unsafe
public class MyClass
{
   unsafe public static void Main()
   {
      int *p = stackalloc int[-30];   // CS0247
   }
}
```

编译器错误 CS0249

项目 • 2023/04/07

请勿重写 object.Finalize。 相反，请提供析构函数。

使用终结器语法来指定对象被破坏时要执行的指令。

以下示例生成 CS0249：

C#

```
// CS0249.cs
class MyClass
{
   protected override void Finalize()   // CS0249
   // try the following line instead
   // ~MyClass()
   {
   }

   public static void Main()
   {
   }
}
```

编译器错误 CS0250

项目 • 2023/11/08

请勿直接调用基类 Finalize 方法。 将从析构函数自动调用它。

程序不能尝试强制清理基类资源。

有关详细信息，请参阅终结器。 此消息使用“终结器”之前的术语：*析构函数*。

下面的示例生成 CS0250

C#

```
// CS0250.cs
class B
{
}

class C : B
{
   ~C()
   {
      base.Finalize();   // CS0250
   }

   public static void Main()
   {
   }
}
```


在 GitHub 上与我们协作

可以在 GitHub 上找到此内容的源，还可以在其中创建和查看问题和拉取请求。 有关详细信息，请参阅参与者指南。

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

提出文档问题

提供产品反馈

编译器错误 CS0254

项目 • 2023/04/07

fixed 语句赋值的右边不能是强制转换表达式

不能强制转换 固定 表达式的右侧。 有关详细信息，请参阅不安全代码和指针。

下面的示例生成 CS0254：

C#

```
// CS0254.cs
// compile with: /unsafe
class Point
{
   public uint x, y;
}

class FixedTest
{
   unsafe static void SquarePtrParam (int* p)
   {
      *p *= *p;
   }

   unsafe public static void Main()
   {
      Point pt = new Point();
      pt.x = 5;
      pt.y = 6;

      fixed (int* p = (int*)&pt.x)   // CS0254
      // try the following line instead
      // fixed (uint* p = &pt.x)
      {
         SquarePtrParam ((int*)p);
      }
   }
}
```

编译器错误 CS0255

项目 • 2023/04/07

stackalloc 不能用在 catch 或 finally 块中

不能在 catch 或 finally 块中使用 stackalloc 运算符。 有关详细信息，请参阅异常和异常处理。

下面的示例生成 CS0255：

C#

```
// CS0255.cs
// compile with: /unsafe
using System;

public class TestTryFinally
{
   public static unsafe void Test()
   {
      int i = 123;
      string s = "Some string";
      object o = s;

      try
      {
         // Conversion is not valid; o contains a string not an int
         i = (int) o;
      }
      finally
      {
         Console.Write("i = {0}", i);
         int* fib = stackalloc int[100];   // CS0255
      }
   }

   public static void Main()
   {
   }
}
```

编译器错误 CS0266

项目 • 2023/04/07

无法将类型“type1”隐式转换为“type2”。 存在显式转换（是否缺少强制转换？）

当代码尝试在无法进行隐式转换的两个类型之间转换时，就会发生此错误。 有关详细信息，请参阅显式转换和类型转换。

下面的代码演示生成 CS0266 的示例：

C#

```
// CS0266.cs
class MyClass
{
    public static void Main()
    {
        // You cannot implicitly convert a double to an integer.
        double d = 3.2;

        // The following line causes compiler error CS0266.
        int i1 = d;

        // However, you can resolve the error by using an explicit
conversion.
        int i2 = (int)d;

        // You cannot implicitly convert an object to a class type.
        object obj = new MyClass();

        // The following assignment statement causes error CS0266.
        MyClass myClass = obj;

        // You can resolve the error by using an explicit conversion.
        MyClass c = (MyClass)obj;

        // You cannot implicitly convert a base class object to a derived
class type.
        MyClass mc = new MyClass();
        DerivedClass dc = new DerivedClass();

        // The following line causes compiler error CS0266.
        dc = mc;

        // You can resolve the error by using an explicit conversion.
        dc = (DerivedClass)mc;
    }
}

class DerivedClass : MyClass
```

```
{
}
```

编译器错误 CS0268

项目 • 2023/04/07

导入的类型“type”无效。 它包含循环基类依赖项。

如果从其他语言导入的类型具有循环基类依赖项，则将出现此错误。 C# 程序中不能使用这样的类型。 若要解决此错误，请检查任何被引用的程序集或模块中从其他语言导入的类型。

编译器错误 CS0269

项目 • 2023/04/07

使用了未赋值的 out 参数“parameter”

编译器无法验证在使用 out 参数之前是否已给它赋值，它的值在赋值时可能未定义。 在访问值之前，请确保为调用的方法中的 `out` 参数赋值。 如果需要使用传入的变量的值，请改用 `ref` 参数。 有关详细信息，请参阅 Method 参数。

示例 1

下面的示例生成 CS0269：

C#

```
// CS0269.cs
class C
{
    public static void F(out int i)
    // One way to resolve the error is to use a ref parameter instead
    // of an out parameter.
    //public static void F(ref int i)
    {
        // The following line causes a compiler error because no value
        // has been assigned to i.
        int k = i;  // CS0269
        i = 1;
        // The error does not occur if the order of the two previous
        // lines is reversed.
    }

    public static void Main()
    {
        int myInt = 1;
        F(out myInt);
        // If the declaration of method F is changed to require a ref
        // parameter, ref must be specified in the call as well.
        //F(ref myInt);
    }
}
```

示例 2

如果变量的初始化发生在 try 块中，则也会出现此错误，因为编译器无法验证 try 块是否可以成功执行：

C#

```
// CS0269b.cs
class C
{
    public static void F(out int i)
    {
        try
        {
            // Assignment occurs, but compiler can't verify it
            i = 1;
        }
        catch
        {
        }

        int k = i;  // CS0269
        i = 1;
    }

    public static void Main()
    {
        int myInt;
        F(out myInt);
    }
}
```

编译器错误 CS0271

项目 • 2023/04/07

由于 get 访问器不可访问，因此不能在此上下文中使用属性或索引器“property/indexer”

当你尝试访问无法访问的 `get` 访问器时出现此错误。 若要解决此错误，增加访问器的可访问性，或更改调用位置。 有关详细信息，请参阅访问器可访问性和属性。

以下示例生成 CS0271：

C#

```
// CS0271.cs
public class MyClass
{
   public int Property
   {
      private get { return 0; }
      set { }
   }

   public int Property2
   {
      get { return 0; }
      set { }
   }
}

public class Test
{
   public static void Main(string[] args)
   {
      MyClass c = new MyClass();
      int a = c.Property;   // CS0271
      int b = c.Property2;   // OK
   }
}
```

编译器错误 CS0272

项目 • 2023/04/07

由于 set 访问器不可访问，因此不能在此上下文中使用属性或索引器“property/indexer”。

当此程序代码不能访问 `set` 时，发生此错误时。

更正此错误

增加访问器的可访问性，或更改调用位置。 有关详细信息，请参阅限制访问器可访问性。

示例

以下示例生成 CS0272：

C#

```
// CS0272.cs
public class MyClass
{
    public int Property
    {
        get { return 0; }
        private set { }
    }
}

public class Test
{
    static void Main()
    {
        MyClass c = new MyClass();
        c.Property = 10;      // CS0272
        // To resolve, remove the previous line
        // or use an appropriate modifier on the set accessor.
    }
}
```

编译器错误 CS0273

项目 • 2023/04/07

“property_accessor”访问器的可访问性修饰符必须比属性或索引器“property”的限制性更强

Set/get 访问器的可访问性修饰符必须比属性或索引器“property/indexer”的限制性更强

当你声明的访问器的可访问性不比属性或索引器的可访问性的限制性小时，就会发生这个错误。

更正此错误

在属性或访问器上使用适当的访问修饰符。 有关详细信息，请参阅限制访问器可访问性和访问器。

示例

此示例包含具有内部 set 方法的内部属性。 下面的示例生成 CS0273。

C#

```
// CS0273.cs
// compile with: /target:library
public class MyClass
{
   internal int Property
   {
      get { return 0; }
      internal set {}   // CS0273
      // try the following line instead
      // private set {}
   }
}
```

编译器错误 CS0274

项目 • 2023/04/07

不能为属性或索引器“property/indexer”的两个访问器同时指定可访问性修饰符

当你声明一个在两个访问器上都有访问修饰符的属性或索引器时，将出现此错误。 若要解决此错误，请仅在两个访问器的其中一个上使用访问修饰符。 有关详细信息，请参阅访问器可访问性。

下面的示例生成 CS0274：

C#

```
// CS0274.cs
public class MyClass
{
    public int Property   // CS0274
    {
        public get { return 0; }
        protected set { }
    }
}
```

编译器错误 CS0275

项目 • 2023/04/07

“accessor”：可访问性修饰符不能在接口中的访问器上使用

对接口中的属性或索引器的某个访问器使用访问修饰符时，将出现此错误。 要解决此问题，请删除该访问修饰符。

示例

下面的示例生成 CS0275：

C#

```
// CS0275.cs
public interface MyInterface
{
    int Property
    {
        get;
        internal set;   // CS0275
    }
}
```

编译器错误 CS0276

项目 • 2023/04/07

“property/indexer”: 仅当属性或索引器同时具有 get 访问器和 set 访问器时，才能对访问器使用可访问性修饰符

当仅使用一个访问器声明属性或索引器，并对该访问器使用访问修饰符时，会出现此错误。 若要解决此错误，请删除访问修饰符或添加另一个访问器。

示例

下面的示例生成 CS0276：

C#

```
// CS0276.cs
public class MyClass
{
    public int Property
    {
        protected set { }   // CS0276
    }
    public int Property2
    {
        internal get { }   // CS0276
    }
}
```

编译器错误 CS0277

项目 • 2023/04/07

“class”不实现接口成员“accessor”。 “class accessor”不是公共的

如果你尝试实现接口的属性，而类中属性访问器的的实现不是公共的，则将出现此错误。 实现接口成员的方法需要具有公共可访问性。 若要解决此问题，请删除属性访问器上的访问修饰符。

示例

以下示例生成 CS0277：

C#

```
// CS0277.cs
public interface MyInterface
{
    int Property
    {
        get;
        set;
    }
}

public class MyClass : MyInterface   // CS0277
{
    public int Property
    {
        get { return 0; }
        // Try this instead:
        //set { }
        protected set { }
    }
}
```

编译器错误 CS0281

项目 • 2023/04/07

已授予对“AssemblyName1”的友元访问，但输出程序集的名称为“AssemblyName2”。 请尝试添加对“AssemblyName1”的引用，或者更改输出程序集的名称以使其匹配。

友元访问是一种新的公共语言运行时 (CLR) 功能，它可以使一个程序集看到另一个程序集的非公共类型。 当授予友元访问的程序集为被授予程序集指定错误的名称时，将会出现此错误。 有关详细信息，请参阅友元程序集。

示例 1

下面的代码示例序列将生成 CS0281。

按照下面的方式生成用于创建具有强名称的程序集的文件：

- sn -d CS0281.snk
- sn -k CS0281.snk
- sn -i CS0281.snk CS0281.snk
- sn -pc CS0281.snk key.publickey
- sn -tp key.publickey

C#

```
// CS0281.cs
// compile with: /target:library /keyfile:CS0281.snk
public class A {}
```

示例 2

C#

```
// CS0281_b.cs
// compile with: /target:library /keyfile:CS0281.snk /reference:CS0281.dll
[assembly:System.Runtime.CompilerServices.InternalsVisibleTo("CS0281,
PublicKey=0024000004800000940000000602000000240000525341310004000001000100 4b
2d4d56af7c50be2fcbbf97cb880b9e73ad84467a587191fef63aadc118a96cecf9d508cd679c
907b6e20f71684300bdc2c0a851019af0c96b29bf8f1339753276041aefd67db46139e6348b3
a12f29537b4dc6c2c19829df2c9ed6803f3c63c3b84cfa2728849386aea575c543a5f70fa857
```

```
93d2946f15f7fe1ccb0c5e8fe0")]
class B : A {}
```

示例 3

下面的示例生成 CS0281。

请注意，此示例创建的输出文件与第一个示例中创建的输出文件同名。 若要解决此问题，请不要更改组件的程序集特性并添加类 C。

C#

```
// CS0281_c.cs
// compile with: /target:library /out:CS0281.dll /keyfile:CS0281.snk
/reference:CS0281_b.dll
// CS0281 expected
[assembly:System.Reflection.AssemblyVersion("3")]
[assembly:System.Reflection.AssemblyCulture("en-us")]
class C : B {}
public class A {}
```

编译器错误 CS0283

项目 • 2023/04/07

不能将类型“type”声明为 const

常数声明中指定的类型必须是 `byte`、`sbyte`、`ushort`、`short`、`uint`、`int`、`ulong`、`long`、`char`、`float`、`double`、`decimal`、`bool`、`string`、枚举类型，或者赋值为 `null` 的引用类型。 每个常数表达式必须生成一个目标类型的值，或者生成一个可隐式转换为目标类型的类型的值。

示例

以下示例生成 CS0283。

C#

```
// CS0283.cs
struct MyTest
{
}
class MyClass
{
    // To resolve the error but retain the "const-ness",
    // change const to readonly.
    const MyTest test = new MyTest();   // CS0283

    public static int Main() {
        return 1;
    }
}
```

编译器错误 CS0304

项目 • 2023/04/07

变量类型“type”没有 new() 约束，因此无法创建该类型的实例

如果实现泛型类，并且需要使用 `new` 关键字来创建为类型参数 `T` 提供的任何类型的新实例，则必须将 new() constraint 应用到类声明中的 `T`，如下面的示例所示。

C#

```
class C<T> where T : new()
```

`new()` 约束通过保证为 `T` 提供的任何具体类型都具有无参数构造函数来实现类型安全性。 在 `T` 未指定 `new()` 约束时，如果尝试使用类的正文中的 `new` 运算符来创建类型参数 `T` 的实例，则将出现 CS0304。 在客户端，如果代码尝试实例化包含不具有无参数构造函数这一类型的泛型类，该代码将生成编译器错误 CS0310。

下面的示例生成 CS0304。

C#

```
// CS0304.cs
// Compile with: /target:library.
class C<T>
{
    // The following line generates CS0304.
    T t = new T();
}
```

该类的方法中也不允许使用 `new` 运算符。

C#

```
// Compile with: /target:library.
class C<T>
{
    public void ExampleMethod()
    {
        // The following line generates CS0304.
        T t = new T();
    }
}
```

若要避免此错误，请使用 `new()` 约束声明该类，如下面的示例所示。

C#

```
// Compile with: /target:library.
class C<T> where T : new()
{
    T t = new T();

    public void ExampleMethod()
    {
        T t = new T();
    }
}
```

另请参阅

- C# 编译器错误

编译器错误 CS0305

项目 • 2023/04/07

使用泛型类型“generic type”需要“number”类型实参

找不到预期数目的类型实参时，将出现此错误。 若要解决 C0305，请使用所需数目的类型实参。

示例

下面的示例生成 CS0305。

C#

```
// CS0305.cs
public class MyList<T> {}
public class MyClass<T> {}

class MyClass
{
   public static void Main()
   {
      MyList<MyClass, MyClass> list1 = new MyList<MyClass>();   // CS0305
      MyList<MyClass> list2 = new MyList<MyClass>();   // OK
   }
}
```

编译器错误 CS0306

项目 • 2023/04/07

类型“type”不能用作类型参数

不允许用作类型参数的类型。 这可能是因为类型是指针类型。

以下示例生成 CS0306：

C#

```
// CS0306.cs
class C<T>
{
}

class M
{
    // CS0306 – int* not allowed as a type parameter
     C<int*> f;
}
```

编译器错误 CS0307

项目 • 2023/04/07

“construct”“identifier”不是泛型方法。 如果原打算使用表达式列表，请用括号将 < 表达式括起来。

命名的构造不是类型或方法，是唯一能采用泛型参数的构造。 在尖括号中删除类型参数。 如果需要泛型，则将泛型构造声明为泛型类型或方法。

以下示例生成 CS0307：

C#

```
// CS0307.cs
class C
{
   public int P { get { return 1; } }
   public static void Main()
   {
      C c = new C();
      int p = c.P<int>();  // CS0307 – C.P is a property
      // Try this instead
      // int p = c.P;
   }
}
```

编译器错误 CS0308

项目 • 2023/04/07

非泛型类型或方法“identifier”不能与类型参数一起使用。

方法或类型并非泛型，却与类型参数一起使用。 若要避免此错误，删除尖括号和类型参数，或将该方法或类型重新声明为泛型方法或类型。

以下示例生成 CS0308：

C#

```
// CS0308a.cs
class MyClass
{
   public void F() {}
   public static void Main()
   {
      F<int>();  // CS0308 – F is not generic.
      // Try this instead:
      // F();
   }
}
```

以下示例还生成 CS0308。 要解决此错误，请使用"using System.Collections.Generic"指令。

C#

```
// CS0308b.cs
// compile with: /t:library
using System.Collections;
// To resolve, uncomment the following line:
// using System.Collections.Generic;
public class MyStack<T>
{
    // Store the elements of the stack:
    private T[] items = new T[100];
    private int stack_counter = 0;

    // Define the iterator block:
    public IEnumerator<T> GetEnumerator()   // CS0308
    {
        for (int i = stack_counter - 1 ; i >= 0; i--)
        yield return items[i];
    }
}
```

编译器错误 CS0310

项目 • 2023/04/07

类型“typename”必须是具有公共的无参数构造函数的非抽象类型，才能用作泛型类型或方法“generic”中的参数“parameter”

泛型类型或方法在其 `where` 子句中定义了一个 new() 约束，因此任何类型必须具有公共的无参数构造函数，才能用作该泛型类型或方法的类型参数。 若要避免此错误，请确保类型具有正确的构造函数，或修改泛型类型或方法的约束子句。

示例

下面的示例生成 CS0310：

C#

```
// CS0310.cs
using System;

class G<T> where T : new()
{
    T t;

    public G()
    {
        t = new T();
        Console.WriteLine(t);
    }
}

class B
{
    private B() { }
    // Try this instead:
    // public B() { }
}

class CMain
{
    public static void Main()
    {
        G<B> g = new G<B>();   // CS0310
        Console.WriteLine(g.ToString());
    }
}
```

编译器错误 CS0311

项目 • 2024/05/24

不能将类型“type1”用作泛型类型或方法“<name>”中的类型参数“T”。 没有从“type1”到“type2”的隐式引用转换。

对泛型类型参数应用约束时，必须存在从具体参数到约束类型的隐式标识或引用转换。

更正此错误

1. 将正在使用的类型参数更改为满足约束的类型参数。
2. 如果你拥有类，则可以删除该约束，否则需要执行一些操作来启用隐式引用或标识转换。 例如，可以从第一个类型继承第二个类型。

示例

C#

```
// cs0311.cs
class B {}
class C {}
class Test<T> where T : C
{ }

class Program
{
    static void Main()
    {
        Test<B> test = new Test<B>(); //CS0311
    }
}
```

如果在尝试使用值类型参数时发生此错误，请注意，隐式数字转换（例如从 `short` 到 `int`）不符合泛型类型参数。

另请参阅

- 类型参数的约束

在 GitHub 上与我们协作

可以在 GitHub 上找到此内容的源，还可以在其中创建和查看问题和拉取请求。 有关详细信息，请参阅参与者指南。

.NET 反馈

.NET 是一个开放源代码项目。 选择一个链接以提供反馈：

提出文档问题

提供产品反馈

编译器错误 CS0312

项目 • 2023/04/07

不能将类型“type1”用作泛型类型或方法“name”中的类型参数“name”。 可以为 null 的类型“type1”不满足“type2”的约束。

可以为 null 的值类型与其不可为 null 的对应类型不同；它们之间不存在隐式引用转换或标识转换。 可以为 null 的装箱转换不满足泛型类型约束。 在以下示例中，第一个类型形参是 `Nullable<int>` ，第二个类型形参是 `System.Int32`。

更正此错误

1. 删除该约束。
2. 在以下示例中，令第二个类型实参为 `int?` 或 `object`。

示例

以下代码生成 CS0312：

C#

```
// cs0312.cs
class Program
{
    static void MTyVar<T, U>() where T : U { }

    static int Main()
    {
        MTyVar<int?, int>(); // CS0312
        return 1;
    }
}
```

尽管可以为 null 的值类型与不可为 null 的类型不同，但可以为 null 和不可为 null 值之间允许存在各种类型的转换。

另请参阅

- 可以为 null 的值类型

编译器错误 CS0313

项目 • 2023/04/07

在泛型类型或方法“type 2”中不能将类型“type1”用作泛型类型“parameter name”。 可以为 null 的类型“type1”不满足“type2”的约束。 可以为 null 的类型不能满足任何接口约束。

可以为 null 的值类型与其不可为 null 的对应项不相等。 在以下示例中， `ImplStruct` 满足 `BaseInterface` 约束但 `ImplStruct?` 不满足，因为 `Nullable<ImplStruct>` 不实现 `BaseInterface`。

更正此错误

1. 用以下代码作为示例，一种解决方案是在调用 `ImplStruct` 时指定一个普通的 `TestMethod`作为第一个类型。 然后修改 `TestMethod` 以在返回语句中创建一个可为 null 的 `Implstruct` 版本：

   C#

   ```
   return new Nullable<T>(t);
   ```

示例

以下代码生成 CS0313：

C#

```
// cs0313.cs
public interface BaseInterface { }
public struct ImplStruct : BaseInterface { }

public class TestClass
{
    public T? TestMethod<T, U>(T t) where T : struct, U
    {
        return t;
    }
}

public class NullableTest
{
    public static void Run()
    {
```

```
        TestClass tc = new TestClass();
        tc.TestMethod<ImplStruct?, BaseInterface>(new ImplStruct?()); //
CS0313
    }
    public static void Main()
    { }
}
```

另请参阅

- 可以为 null 的值类型

编译器错误 CS0314

项目 • 2023/04/07

不能将类型“type1”用作泛型类型或方法“name”中的类型参数“name”。 没有从“type1”到“type2”的装箱转换或类型参数转换。

当泛型类型使用受约束的类型参数时，新类也必须满足这些相同的约束。

更正此错误

1. 在下面的示例中，将 `where T : ClassConstraint` 添加到类 `B` 中。

示例

下面的代码生成 CS0314：

C#

```
// cs0314.cs
// Compile with: /target:library
public class ClassConstraint { }

public class A<T> where T : ClassConstraint
{ }

public class B<T> : A<T> //CS0314
{ }

// Try using this instead.
public class C<T> : A<T> where T : ClassConstraint
{ }
```

另请参阅

- 类型参数的约束

编译器错误 CS0315

项目 • 2023/04/07

不能将类型“valueType”用作泛型类型或方法“TypeorMethod<T>”中的类型参数“T”。 没有从“valueType”到“referenceType”的装箱转换。

将泛型类型约束为特定类，并尝试使用不能隐式装箱到它的值类型构造该类的实例时，会发生此错误。

更正此错误

1. 一种解决方案是将结构重新定义为类。

示例

下面的示例生成 CS0315：

C#

```
// cs0315.cs
public class ClassConstraint { }
public struct ViolateClassConstraint { }

public class Gen<T> where T : ClassConstraint
{
}
public class Test
{
    public static int Main()
    {
        Gen<ViolateClassConstraint> g = new Gen<ViolateClassConstraint>();
//CS0315
        return 1;
    }
}
```

另请参阅

- 类型参数的约束
- 装箱和取消装箱

编译器错误 CS0316

项目 • 2023/04/07

参数名“name”与某个自动生成的参数名冲突。

保留的字不能用作参数名。 在以下示例中， `value` 是默认属性或索引器访问器上下文中的保留字。

更正此错误

1. 更改参数的名称。

示例

以下代码生成 CS0316：

C#

```
// cs0316.cs
// Compile with: /target:library
public class Test
{
    public int this[int value] // CS0316
    {
        get { return 1; }
        set { }
    }
}
```

另请参阅

- 索引器
- C# 关键字

编译器错误 CS0401

项目 • 2024/08/03

new() 约束必须是指定的最后一个限制性约束

当使用多个约束时，将在 new () 约束之前列出所有其他约束。

示例

下面的示例生成 CS0401。

C#

```
// CS0401.cs
// compile with: /target:library
using System;
 class C<T> where T : new(), IDisposable {}  // CS0401

class D<T> where T : IDisposable
{
   static void F<U>() where U : new(), IDisposable{}   // CS0401
}
```


在 GitHub 上与我们协作

可以在 GitHub 上找到此内容的源，还可以在其中创建和查看问题和拉取请求。 有关详细信息，请参阅参与者指南。

.NET

.NET 反馈

.NET 是一个开放源代码项目。 选择一个链接以提供反馈：

提出文档问题

提供产品反馈

编译器错误 CS0403

项目 • 2023/04/07

无法将 null 转换为类型参数“name”，因为它可能是不可以为 null 的值类型。 请考虑改用 default('T')。

不能将 null 分配给指定的未知类型，因为它可能是不允许进行 null 赋值的值类型。 如果泛型类不打算接受值类型，请使用类类型约束。 如果它可以接受值类型（如内置类型），则你可能能够使用表达式 `default(T)` 替换 null 赋值，如下面的示例所示。

示例

下面的示例生成 CS0403。

C#

```
// CS0403.cs
// compile with: /target:library
class C<T>
{
   public void f()
   {
      T t = null;  // CS0403
      T t2 = default(T);   // OK
    }
}

class D<T> where T : class
{
   public void f()
   {
      T t = null;  // OK
    }
}
```

编译器错误 CS0404

项目 • 2023/04/07

意外的 '<'：特性不能是泛型的

特性中不允许使用泛型类型参数。 删除类型参数和尖括号。

下面的示例生成 CS0404：

C#

```
// CS0404.cs
[MyAttrib<int>]  // CS0404
class C
{
   public static void Main()
   {

   }
}
```

编译器错误 CS0405

项目 • 2023/04/07

类型参数“type parameter”的重复约束“constraint”

泛型声明的两个约束完全相同。 若要消除此错误，请删除重复的约束。

下面的示例生成 CS0405：

C#

```
// CS0405.cs
interface I
{
}

class C<T> where T : I, I  // CS0405.cs
{
}
```

编译器错误 CS0406

项目 • 2023/04/07

类类型约束 “constraint” 必须位于任何其他约束之前

当泛型类型或方法具有类类型约束时，必须先列出该约束。 若要避免此错误，将此类类型约束移动到约束列表的开头。

示例

以下示例生成 CS0406。

C#

```
// CS0406.cs
// compile with: /target:library
interface I {}
class C {}
class D<T> where T : I, C {}   // CS0406
class D2<T> where T : C, I {}   // OK
```

编译器错误 CS0407

项目 • 2023/04/07

“return-type method”的返回类型错误

方法与委托类型不兼容。 参数类型匹配，但返回类型对于该委托来说不是正确的返回类型。 若要避免此错误，使用不同的方法、更改方法的返回类型，或更改该委托的返回类型。

示例

以下示例生成 CS0407：

C#

```
// CS0407.cs
public delegate int MyDelegate();

class C
{
    MyDelegate d;

    public C()
    {
        d = new MyDelegate(F);  // OK: F returns int
        d = new MyDelegate(G);  // CS0407 – G doesn't return int
    }

    public int F()
    {
        return 1;
    }

    public void G()
    {
    }

    public static void Main()
    {
        C c1 = new C();
    }
}
```

编译器错误 CS0409

项目 • 2023/04/07

已为类型参数“type parameter”指定约束子句。 必须在单个 where 子句中指定类型参数的所有约束。

已找到单个类型参数的多个约束子句（where 子句） 删除冗余的 where 子句，或更正 where 子句，以便每个子句中只有唯一类型参数。

C#

```
// CS0409.cs
interface I
{
}

class C<T1, T2> where T1 : I where T1 : I  // CS0409 - T1 used twice
{
}
```

编译器错误 CS0410

项目 • 2023/04/07

“method”的重载均没有正确的参数类型和返回类型

如果尝试使用具有错误参数类型的函数对委托进行实例化，将出现此错误。 委托的参数类型必须与你要分配给该委托的函数相匹配。

示例

下面的示例生成 CS0410：

C#

```
// CS0410.cs
// compile with: /langversion:ISO-1

class Test
{
    delegate void D(double d );
    static void F(int i) { }

    static void Main()
    {
        D d = new D(F);  // CS0410
    }
}
```

备注

Roslyn 中不再使用此编译器错误。 使用 Roslyn 编译时，上一个示例生成 CS0123。

编译器错误 CS0411

项目 • 2023/04/07

无法从用法推断方法“method”的类型参数。 请尝试显式指定类型参数。

如果未显式提供类型参数就调用泛型方法，且编译器无法推断预期类型参数，则会发生此错误。 若要避免此错误，在尖括号中添加预期的类型参数。

示例 1

以下示例生成 CS0411：

C#

```
// CS0411.cs
class C
{
    void G<T>()
    {
    }

    public static void Main()
    {
        G();  // CS0411
        // Try this instead:
        // G<int>();
    }
}
```

示例 2

其他可能发生的错误情况还包括当参数为 `null`（无类型信息）时发生的错误：

C#

```
// CS0411b.cs
class C
{
    public void F<T>(T t) where T : C
    {
    }

    public static void Main()
    {
        C c = new C();
        c.F(null);  // CS0411
```

```
    }
}
```

编译器错误 CS0412

项目 • 2023/04/07

“generic”：参数或局部变量不能与方法类型参数同名

泛型方法的类型参数和此方法的局部变量或其中一项方法的参数之间存在名称冲突。 若要避免此错误，重命名任何存在冲突的参数或局部变量。

示例

以下示例生成 CS0412：

C#

```
// CS0412.cs
using System;

class C
{
    // Parameter name is the same as method type parameter name
    public void G<T>(int T)  // CS0412
    {
    }
    public void F<T>()
    {
        // Method local variable name is the same as method type
        // parameter name
        double T = 0.0;  // CS0412
        Console.WriteLine(T);
    }

    public static void Main()
    {
    }
}
```

编译器错误 CS0413

项目 • 2023/04/07

由于类型参数“type parameter”既没有类类型约束，也没有“class”约束，因此不能与“as”运算符一起使用

如果泛型类型使用 as 运算符，但该泛型类型不具有类类型约束，则发生此错误。 `as` 运算符只允许用于引用和可为 null 的值类型，因此必须约束类型参数，保证它不是值类型。 若要避免此错误，请使用类类型约束或引用类型约束。

这是因为 `as` 运算符能够返回 `null`（它不可能是值类型的值），并且必须将类型参数视为值类型，除非它是类类型约束或引用类型约束。

示例

下面的示例生成 CS0413。

C#

```
// CS0413.cs
// compile with: /target:library
class A {}
class B : A {}

class CMain
{
   A a = null;
   public void G<T>()
   {
      a = new A();
      System.Console.WriteLine (a as T);  // CS0413
   }

   // OK
   public void H<T>() where T : A
   {
      a = new A();
      System.Console.WriteLine (a as T);
   }
}
```

编译器错误 CS0415

项目 • 2023/04/07

“IndexerName”特性仅对不是显式接口成员声明的索引器有效

如果在作为显式接口实现的索引器上使用 IndexerName 特性，将出现此错误。 如有可能，可通过从索引器声明中删除接口名称来避免此错误。 有关详细信息，请参阅 IndexerNameAttribute 类。

以下示例生成 CS0415：

C#

```
// CS0415.cs
using System;
using System.Runtime.CompilerServices;

public interface IA
{
    int this[int index]
    {
        get;
        set;
    }
}

public class A : IA
{
    [IndexerName("Item")]  // CS0415
    int IA.this[int index]
    // Try this line instead:
    // public int this[int index]
    {
        get { return 0; }
        set { }
    }

    static void Main()
    {
    }
}
```

编译器错误 CS0416

项目 • 2023/04/07

“类型形参”：特性实参不能使用类型形参

类型形参用作了特性实参，这是不被允许的。 使用非泛型类型

下面的示例生成 CS0416：

C#

```
// CS0416.cs
public class MyAttribute : System.Attribute
{
   public MyAttribute(System.Type t)
   {
   }
}

class G<T>
{

   [MyAttribute(typeof(G<T>))]  // CS0416
   public void F()
   {
   }

}
```

编译器错误 CS0417

项目 • 2023/04/07

“identifier”：创建变量类型的实例时无法提供变量

如果对类型参数上的 `new` 运算符的调用带有参数，则会发生此错误。 通过对未知参数类型使用 `new`，可以调用的唯一构造函数是不带参数的构造函数。 如果需要调用另一构造函数，请考虑使用类类型约束或接口约束。

示例

下面的示例生成 CS0417：

C#

```
// CS0417
class ExampleClass<T> where T : new()
{
    // The following line causes CS0417.
    T instance1 = new T(1);

    // The following line doesn't cause the error.
    T instance2 = new T();
}
```

另请参阅

- 类型参数的约束

编译器错误 CS0418

项目 • 2023/04/07

“class name”：抽象类不能是密封的或静态的

除非从抽象类继承，否则抽象类不能用于创建对象，所以它成为密封的没有意义。 抽象类成为静态也没有意义；设计抽象类是为了支持将抽象类用作基类的对象层次结构。

示例

下面的示例生成 CS0418：

C#

```
// CS0418.cs
public abstract sealed class C  // CS0418
{
}

sealed static class S  // CS0418
{
}

public class MyClass
{
    public static void Main()
    {
    }
}
```

编译器错误 CS0423

项目 • 2023/04/07

由于“class”具有 ComImport 特性，因此“method”必须是外部的或抽象的

指定 ComImport 特性意味着类的实现将从 COM 模块导入。 不能定义其他方法。

下面的示例生成 CS0423：

C#

```
// CS0423.cs

using System.Runtime.InteropServices;

[
  ComImport,
  Guid("7ab770c7-0e23-4d7a-8aa2-19bfad479829")
]
class ImageProperties
{
  public static void Main()  // CS0423
  {
    ImageProperties i = new ImageProperties();
  }
}
```

编译器错误 CS0424

项目 • 2023/04/07

“class”: 具有 ComImport 特性|的类不能指定基类

指定 ComImportAttribute 特性意味着类的实现将从 COM 模块导入。 不允许将从基类继承的其他方法或字段添加到在 COM 模块中定义的实现中。

下面的示例生成 CS0424：

C#

```
// CS0424.cs
// compile with: /target:library
using System.Runtime.InteropServices;
public class A {}

[ ComImport, Guid("7ab770c7-0e23-4d7a-8aa2-19bfad479829") ]
class B : A {}   // CS0424 error
```

编译器错误 CS0425

项目 • 2023/04/07

方法“method”的类型参数“type parameter”的约束必须与接口方法“method”的类型参数“type parameter”的约束相匹配。 请考虑改用显式接口实现。

如果一个虚泛型方法在派生类中被重写，并且该方法在派生类中的约束与它在基类中的约束不匹配，则会发生此错误。 若要避免此错误，请确保 `where` 子句在两个声明中相同，或者显式实现接口。

示例 1

下面的示例生成 CS0425：

C#

```
// CS0425.cs

class C1
{
}

class C2
{
}

interface IBase
{
    void F<ItemType>(ItemType item) where ItemType : C1;
}

class Derived : IBase
{
    public void F<ItemType>(ItemType item) where ItemType : C2  // CS0425
    {
    }
}

class CMain
{
    public static void Main()
    {
    }
}
```

示例 2

约束不一定要在字面上匹配，只要约束的设置有相同的含义即可。 例如，以下形式是可行的：

C#

```
// CS0425b.cs

interface J<Z>
{
}

interface I<S>
{
    void F<T>(S s, T t) where T: J<S>, J<int>;
}

class C : I<int>
{
    public void F<X>(int s, X x) where X : J<int>
    {
    }

    public static void Main()
    {
    }
}
```

编译器错误 CS0426

项目 • 2023/04/07

类型“type”中不存在类型名称“identifier”

引用嵌套在另一类型中的类型而这一嵌套类型不存在时，会出现此错误。 键错嵌套类型的名称时可能出现此错误。 检查所用名称的拼写，并验证外层类型具有所需成员。

下面的示例生成 CS0426，因为类 C 不具有嵌套类型 A：

C#

```
// CS0426.cs

class C
{
    // No nested types are declared.
}

class D
{
   public static void Main()
   {
       C c = new C();
       // Attempt to reference a nested type A:
       C.A a; // CS0426 because no such type C.A
   }
}
```

请参阅

- C# 类型系统

编译器错误 CS0428

项目 • 2023/04/07

无法将方法组“Identifier”转换为非委托类型“type”。 是否希望调用此方法?

将方法组转换为非委托类型时，或尝试在不使用括号的情况下调用方法时，会发生此错误。

示例

下面的示例生成 CS0428：

C#

```
// CS0428.cs
namespace ConsoleApplication1
{
    class Program
    {
        delegate int Del1();
        delegate object Del2();

        static void Main(string[] args)
        {
            ExampleClass ec = new ExampleClass();

            // The following assignment statement causes compiler error
CS0428.
            // It attempts to assign the address of Method1 to an integer
variable.
            // You can only assign the address to an appropriate delegate
type.
            int i = ec.Method1;

            // Del1 is a delegate type that is appropriate for a method like
            // Method1 that returns an int. The following assignment
statement
            // does not cause an error.
            Del1 d1 = ec.Method1;

            // You can invoke Method1 and assign the int that is returned to
            // integer variable i.
            i = ec.Method1();

            // The following assignment statement causes compiler error
CS0428.
            // It attempts to assign the address of Method2 to an instance
of
            // ExampleClass. You can only assign the address to a delegate
```

```
type.
            ec = ExampleClass.Method2;

            // Del2 is a delegate type that is appropriate for a method like
            // Method2 that returns an instance of a class. The following
assignment
            // statement does not cause an error.
            Del2 d2 = ExampleClass.Method2;

            // Similarly, you can invoke Method2 and assign the result
returned to ec.
            ec = ExampleClass.Method2();
        }
    }

    public class ExampleClass
    {
        public int Method1() { return 1; }
        public static ExampleClass Method2() { return null; }
    }
}
```

编译器错误 CS0430

项目 • 2023/04/07

在 /reference 选项中未指定外部别名“alias”

遇到外部别名、但命令行上未将别名指定为引用时，将发生此错误。 要解决 CS0430，可使用 **/reference**进行编译。

示例 1

C#

```
// CS0430_a.cs
// compile with: /target:library
public class MyClass {}
```

示例 2

使用 **/reference:MyType=cs0430_a.dll** 进行编译以引用上一个示例中创建的 DLL，即可解决此错误。 下面的示例生成 CS0430。

C#

```
// CS0430_b.cs
extern alias MyType;   // CS0430
public class Test
{
   public static void Main() {}
}
```

编译器错误 CS0433

项目 • 2023/06/14

类型 TypeName1 同时存在于 TypeName2 和 TypeName3 中

在应用程序中引用的两个不同程序集包含相同的命名空间和类型，这会造成多义性。

若要修复此错误，请使用 (References) 编译器选项的别名功能，或者不引用其中任何一个程序集。

以下情况下也可能发生此错误：

- 当 `@ Page` 指令应该是 `CodeBehind` 属性时，它具有`CodeFile` 属性。
- 代码置于 App_Code 文件夹中，但此文件夹不应驻留在该位置。

示例

此代码用歧义类型的第一个副本创建 DLL。

C#

```
// CS0433_1.cs in CS0433_1.csproj
// or compile with: /target:library
namespace TypeBindConflicts
{
   public class AggPubImpAggPubImp { }
}
```

此代码用歧义类型的第二个副本创建 DLL。

C#

```
// CS0433_2.cs in CS0433_2.csproj
// or compile with: /target:library
namespace TypeBindConflicts
{
   public class AggPubImpAggPubImp { }
}
```

因此，当在项目中使用这两个库（`CS0433_1.dll` 和 `CS0433_2.dll`）时，使用 `AggPubImpAddPubImp` 类型将是不明确的，并将导致编译器错误 `CS0433`。

XML

```
<!-- CS0433_3.csproj -->
<ProjectReference Include="..\CS0433_1\CS0433_1.csproj" />
<ProjectReference Include="..\CS0433_2\CS0433_2.csproj" />
```

C#

```
// CS0433_3.cs in CS0433_3.csproj
// or compile with: /reference:cs0433_1.dll /reference:cs0433_2.dll
using TypeBindConflicts;

public class Test
{
   public static void Main()
   {
      AggPubImpAggPubImp n6 = new AggPubImpAggPubImp();   // CS0433
   }
}
```

下面的示例演示可如何使用 /reference 编译器选项的别名功能或 `<ProjectReference>` 中的 `<Aliases>` 功能解决此 CS0433 错误。

XML

```
<!-- CS0433_4.csproj -->
<ProjectReference Include="..\CS0433_1\CS0433_1.csproj">
  <Aliases>CustomTypes</Aliases>
</ProjectReference>
<ProjectReference Include="..\CS0433_2\CS0433_2.csproj" />
```

C#

```
// CS0433_4.cs in CS0433_4.csproj
// compile with: /reference:cs0433_1.dll /reference:CustomTypes=cs0433_2.dll
extern alias CustomTypes;
using TypeBindConflicts;

public class Test
{
   public static void Main()
   {
      // AggPubImpAggPubImp taken from CS0433_1.dll
      AggPubImpAggPubImp n6 = new AggPubImpAggPubImp();

      // AggPubImpAggPubImp taken from CS0433_2.dll
      CustomTypes.TypeBindConflicts.AggPubImpAggPubImp n7 =
          new CustomTypes.TypeBindConflicts.AggPubImpAggPubImp();
   }
}
```

编译器错误 CS0434

项目 • 2023/04/07

NamespaceName2 中的命名空间 NamespaceName1 与 NamespaceName3 中的类型 TypeName1 冲突

当导入的类型和导入的嵌套命名空间具有相同的完全限定名称时，将发生此错误。 当引用该名称时，编译器不能区分这两者。 如果可以更改导入的源代码，则可以通过更改类型或命名空间的名称使这二者在程序集中唯一，从而解决该错误。

以下代码生成错误 CS0434。

示例 1

此代码将创建该类型具有相同完全限定名称的第一个副本。

C#

```
// CS0434_1.cs
// compile with: /t:library
namespace TypeBindConflicts
{
    namespace NsImpAggPubImp
    {
        public class X { }
    }
}
```

示例 2

此代码将创建该类型具有相同完全限定名称的第二个副本。

C#

```
// CS0434_2.cs
// compile with: /t:library
namespace TypeBindConflicts {
    // Conflicts with another import (import2.cs).
    public class NsImpAggPubImp { }
    // Try this instead:
    // public class UniqueClassName { }
}
```

示例 3

此代码将引用具有此相同完全限定名称的类型。

C#

```
// CS0434.cs
// compile with: /r:cs0434_1.dll /r:cs0434_2.dll
using TypeBindConflicts;
public class Test
{
    public TypeBindConflicts.NsImpAggPubImp.X n2 = null; // CS0434
}
```

编译器错误 CS0438

项目 • 2023/04/07

“module_1”中的类型“'type”与“module_2”中的命名空间“namespace”冲突。

当源文件中的类型与另一个源文件中的命名空间冲突时，将发生此错误。 这种情况通常发生在其中之一或二者均来自添加的模块时。 若要解决此问题，请对导致冲突的类型或命名空间进行重命名。

下面的示例生成 CS0438：

首先编译以下文件:

C#

```
// CS0438_1.cs
// compile with: /target:module
public class Util
{
   public class A { }
}
```

再编译以下文件：

C#

```
// CS0438_2.cs
// compile with: /target:module
namespace Util
{
   public class A { }
}
```

然后编译以下文件：

C#

```
// CS0438_3.cs
// compile with: /addmodule:CS0438_1.netmodule /addmodule:CS0438_2.netmodule
using System;
public class Test
{
   public static void Main() {
      Console.WriteLine(typeof(Util.A));   // CS0438
   }
}
```

编译器错误 CS0439

项目 • 2023/04/07

外部别名声明必须位于命名空间中定义的所有其他元素之前

当 `extern` 声明出现在同一命名空间中的其他项（如 `using` 声明）后时，将发生此错误。 `extern` 声明必须出现在所有其他命名空间元素之前。

示例

下面的示例生成 CS0439：

C#

```
// CS0439.cs
using System;

extern alias MyType;   // CS0439
// To resolve the error, make the extern alias the first line in the file.

public class Test
{
    public static void Main()
    {
    }
}
```

编译器错误 CS0441

项目 • 2023/04/07

“class”：类不能同时是静态的和密封的

当你声明一个同时是静态和密封的类时，将出现此错误。 静态类的密封是继承性的，所以不需要 sealed 修饰符。 若要解决此问题，请仅使用一个修饰符。

下面的示例生成 CS0441：

C#

```
// CS0441.cs
sealed static class MyClass { }   // CS0441
```

编译器错误 CS0442

项目 • 2023/04/07

“Property”：抽象属性不能具有专用访问器

使用访问修饰符“private”修饰抽象访问器时，将出现此错误。 解决方法是使用其他访问修饰符或将该属性改为非抽象。

示例

下面的示例生成 CS0442：

C#

```
// CS0442.cs
public abstract class MyClass
{
    public abstract int AbstractProperty
    {
        get;
        private set;   // CS0442
        // Try this instead:
        // set;
    }
}
```

编译器错误 CS0443

项目 • 2023/04/07

语法错误，应为值

当引用数组而没有指定数组索引的值时，将发生此错误。

示例

以下代码生成 CS0443：

C#

```
// CS0443.cs
using System;
class MyClass
{
    public static void Main()
    {
        int[,] x = new int[1,5];
        if (x[] == 5) {} // CS0443
        // if (x[0, 0] == 5) {}
    }
}
```

编译器错误 CS0445

项目 • 2023/04/07

无法修改取消装箱转换的结果

取消装箱转换的结果是一个临时变量。 编译器可防止你修改此类变量，因为任何修改都会随临时变量一并消失。 若要修复此错误，请声明新的值类型变量来存储中间表达式，并将取消装箱的转换的结果赋给该新变量。

下面的代码生成 CS0455。

C#

```
// CS0445.CS
class UnboxingTest
{
    public static void Main()
    {
        Point p;
        p.x = 1;
        p.y = 2;
        object obj = p;
        // The following line generates CS0445, because the result
        // of unboxing obj is a temporary variable.
        ((Point)obj).x = 2;

        // The following lines resolve the error.

        // Store the result of the unboxing conversion in p2.
        Point p2;
        p2 = (Point)obj;
        // Then you can modify the unboxed value.
        p2.x = 2;
    }
}

struct Point
{
    public int x, y;
}
```

编译器错误 CS0446

项目 • 2023/04/07

Foreach 不能在“方法或委托”上运行。 是否希望调用“方法或委托”?

指定没有括号的方法或指定在 `foreach` 语句部分（通常在此放入集合类）没有括号的匿名方法会导致此错误。 请注意，如果方法返回集合类，则将该方法调用放置在那个位置是有效的（尽管这不常见）。

示例

下面的代码将生成 CS0446。

C#

```
// CS0446.cs
using System;
class Tester
{
    static void Main()
    {
        int[] intArray = new int[5];
        foreach (int i in M) { } // CS0446
    }
    static void M() { }
}
```

编译器错误 CS0447

项目 • 2023/04/07

特性不能在类型实参上使用，只能在类型形参上使用

当将特性应用于调用语句中出现的类型实参时，将发生此错误。 在如下所示的类或方法声明语句中将特性应用于类型参数是可接受的：

```
C#

class C<[some attribute] T> {…}
```

以下代码行将生成此错误。 假定在之前的代码行中定义的类 `C` 具有一个静态方法 `MyStaticMethod`。

```
C#

C<[some attribute] T>.MyStaticMethod();
```

示例

以下代码生成错误 CS0447：

```
C#

// CS0447.cs
using System;

namespace Test41
{
    public interface I<A>
    {
        void Meth<B>();
    }
    public class B : I<int>
    {
        void I<[Test] int>.Meth<X>() { }  // CS0447
    }
}
```

> **备注**
>
> Roslyn 中不再使用此编译器错误。

编译器错误 CS0448

项目 • 2023/04/07

++ 或 -- 运算符的返回类型必须是包含类型或从包含类型派生的类型

当你重写 `++` 或 `--` 运算符时，它们必须返回与包含类型相同的类型，或返回派生自包含类型的类型。

示例 1

以下示例生成 CS0448。

C#

```
// CS0448.cs
class C5
{
   public static int operator ++(C5 c) { return null; }   // CS0448
   public static C5 operator --(C5 c) { return null; }   // OK
   public static void Main() {}
}
```

示例 2

以下示例生成 CS0448。

C#

```
// CS0448_b.cs
public struct S
{
   public static S? operator ++(S s) { return new S(); }   // CS0448
   public static S? operator --(S s) { return new S(); }   // CS0448
}

public struct T
{
// OK
   public static T operator --(T t) { return new T(); }
   public static T operator ++(T t) { return new T(); }

   public static T? operator --(T? t) { return new T(); }
   public static T? operator ++(T? t) { return new T(); }

   public static void Main() {}
}
```

编译器错误 CS0449

项目 • 2023/04/07

"class" 或 "struct" 约束必须在其他任何约束之前

对泛型类型或方法的类型参数的约束必须按特定顺序发生： `class` 或 `struct` 必须是第一个（如果存在），然后是任何接口约束，最后是任何构造函数约束。 此错误是由于 `class` 或 `struct` 约束没有先出现所致。 若要解决此错误，请重新排列约束子句。

示例

下面的示例生成 CS0449。

C#

```
// CS0449.cs
// compile with: /target:library
interface I {}
public class C4
{
   public void F1<T>() where T : class, struct, I {}   // CS0449
   public void F2<T>() where T : I, struct {}   // CS0449
   public void F3<T>() where T : I, class {}   // CS0449

   // OK
   public void F4<T>() where T : class {}
   public void F5<T>() where T : struct {}
   public void F6<T>() where T : I {}
}
```

编译器错误 CS0450

项目 • 2023/04/07

“Type Parameter Name”：不能既指定约束类又指定“class”或“struct”约束

如果类型形参受结构类型约束的约束，则它不能也受特定类类型的约束：这在逻辑上是矛盾的，因为结构和类是互斥的类别。 如果类型形参受特定类类型约束的约束，则其定义就受类类型约束的约束，因此指定类类型约束是多余的。

示例

C#

```
// CS0450.cs
// compile with: /t:library
public class GenericsErrors
{
    public class B { }
    public class G3<T> where T : struct, B { } // CS0450
// To resolve, use the following line instead:
// public class G3<T> where T : B { }
}
```

编译器错误 CS0451

项目 • 2023/04/07

"new()" 约束不能与 "struct" 约束一起使用

当指定泛型类型上的约束时， `new()` 约束只可与类类型约束、接口类型约束、引用类型约束和类型形参约束一起使用，不可与值类型约束一起使用。

示例

下面的示例生成 CS0451。

C#

```
// CS0451.cs
using System;
public class C4
{
   public void F4<T>() where T : struct, new() {}   // CS0451
}

// OK
public class C5
{
   public void F5<T>() where T : struct {}
}

public class C6
{
   public void F6<T>() where T : new() {}
}
```

编译器错误 CS0452

项目 • 2023/04/07

类型“type name”必须是引用类型，才能用作泛型类型或方法“'identifier of generic”中的参数“parameter name”

当传递诸如 `struct` 或 `int` 的值类型作为具有引用类型约束的泛型类型或方法参数时，将发生此错误。

示例

以下代码生成错误 CS0452。

C#

```
// CS0452.cs
using System;
public class BaseClass<S> where S : class { }
public class Derived1 : BaseClass<int> { } // CS0452
public class Derived2<S> : BaseClass<S> where S : struct { } // CS0452
```

另请参阅

- 类型参数的约束

编译器错误 CS0453

项目 • 2023/08/17

类型“Type Name”必须是不可以为 null 值的类型，才能将其用作泛型类型或方法“Generic Identifier”中的参数“Parameter Name”

如果在对其上具有 **value** 约束的泛型类型或方法进行实例化时使用了一个非值类型的参数，则会出现此错误。 当你使用可以为 null 的值类型参数时，它也会发生。 请参阅以下示例中的最后两行代码。

示例

以下代码生成此错误。

C#

```
// CS0453.cs
using System;
public class HV<S> where S : struct { }

// CS0453: string is not a value type
public class H1 : HV<string> { }

// CS0453: H1 is a class, not a struct
public class H2 : HV<H1> { }

// CS0453: HV is based on a class, not a struct
public class H3<S> : HV<S> where S : class { }
public class H4<S> : HV<S> where S : struct { } // OK

// CS0453: HV accepts a nullable int type
public class H4 : HV<int?> { }

// CS0453: HV is based on Nullable type of int
public class H5 : HV<Nullable<Nullable<int>>> { }
```

请参阅

- 内置引用类型
- 值类型
- 类型参数的约束

编译器错误 CS0454

项目 • 2023/04/07

涉及“Type Parameter 1”和“Type Parameter 2”的循环约束依赖项

由于有时一个类型参数引用另一个，而第二个参数反过来又引用第一个，因此会出现此错误。 要修复此错误，请删除其中一个约束以中断循环依赖项。 请注意，循环约束依赖项可以是间接的。

示例 1

以下代码生成错误 CS0454。

C#

```
// CS0554
using System;
public class GenericsErrors
{
    public class G4<T> where T : T { } // CS0454
}
```

示例 2

下面的示例演示了两个类型约束之间的循环依赖项。

C#

```
public class Gen<T,U> where T : U where U : T  // CS0454
{
}
```

编译器错误 CS0455

项目 • 2023/04/07

类型参数“Type Parameter Name”继承冲突的约束 -“约束名称 1”和“约束名称 2”

获得此错误的两种常见方式是设置约束，使类型参数派生自两个不相关的类，或者使它派生自类类型或引用类型约束和 `struct` 类型或值类型约束。 若要解决此错误，请从继承层次结构中删除冲突。

示例

以下代码生成错误 CS0455。

C#

```
// CS0455.cs
using System;

public class GenericsErrors {
    public class B { }
    public class B2 { }
    public class G6<T> where T : B { public class N<U> where U : B2, T { } }
// CS0455
}
```

编译器错误 CS0456

项目 • 2023/04/07

类型参数“Type Parameter Name 1”具有“struct”约束，因此“Type Parameter Name 1”不能用作“Type Parameter Name 2”的约束

值类型约束是隐式密封的，因此这些约束不能用作第二个类型参数的约束。 这是因为值类型不可重写。 若要解决此错误，直接将一个值类型约束用于第二个类型参数，而不是通过第一个类型参数间接进行。

示例

下面的示例生成 CS0456。

C#

```
// CS0456.cs
// compile with: /target:library
public class GenericsErrors
{
   public class G5<T> where T : struct
   {
      public class N<U> where U : T {}   // CS0456
      public class N2<U> where U : struct {}   // OK
   }
}
```

编译器错误 CS0459

项目 • 2023/04/07

无法获得只读局部变量的地址

C# 语言中有三种可生成只读局部变量的常见方案：`foreach` `using` 和 `fixed`。 在每种情况下，不允许写入只读本地变量，或获取其地址。 编译器发现尝试获取只读局部变量的地址时，会生成此错误。

示例

以下示例当尝试获取 `foreach` 循环和 `fixed` 语句块中的只读局部变量的地址时，将生成 CS0459：

C#

```
// CS0459.cs
// compile with: /unsafe

class Program
{
    public unsafe void M1()
    {
        int[] ints = new int[] { 1, 2, 3 };
        foreach (int i in ints)
        {
            int *j = &i;  // CS0459
        }

        fixed (int *i = &_i)
        {
            int **j = &i;  // CS0459
        }
    }

    private int _i = 0;
}
```

ⓘ 备注

Roslyn 编译器已更新，并从 Visual Studio 2017 版本 15.5 开始删除了此编译器错误，因此，前面的代码将成功编译此版本及更高版本。

编译器错误 CS0460

项目 • 2023/04/07

重写和显式接口实现方法的约束是从基方法继承的，因此不能直接指定这些约束

当属于派生类一部分的泛型方法重写基类中的方法时，不能对重写的方法指定约束。 派生类中的重写方法从基类中的方法继承其约束。

示例

下面的示例生成 CS0460。

C#

```
// CS0460.cs
// compile with: /target:library
class BaseClass
{
   BaseClass() { }
}

interface I
{
   void F1<T>() where T : BaseClass;
   void F2<T>() where T : struct;
   void F3<T>() where T : BaseClass;
}

class ExpImpl : I
{
   void I.F1<T>() where T : BaseClass {}   // CS0460
   void I.F2<T>() where T : class {}  // CS0460
}
```

编译器错误 CS0462

项目 • 2023/04/07

继承的成员“member1”和“member2”在类型“type”中具有相同的签名，因此不能重写这些成员

此错误是由于引入泛型而引起的。 正常情况下，类中的方法不能有两个具有相同签名的版本。 但是对于泛型，如果泛型方法使用某个特定的类型实例化，则可以指定一个可能与另一个方法重复的泛型方法。

示例

当实例化 `C<int>` 时，将为方法 `F` 创建两个具有相同签名的版本，所以类 `D` 中的重写无法确定对哪个方法应用重写。

下面的示例生成 CS0462。

C#

```
// CS0462.cs
// compile with: /target:library
class C<T>
{
   public virtual void F(T t) {}
   public virtual void F(int t) {}
}

class D : C<int>
{
   public override void F(int t) {}   // CS0462
}
```

编译器错误 CS0463

项目 • 2023/04/07

计算十进制常量表达式失败，错误为:“error”

当十进制常量表达式在编译时溢出时会发生此错误。

溢出错误通常在运行时发生。 在本例中，你是这样定义常量表达式的：编译器可以计算结果并且知道可能会发生溢出。

示例

下面的代码生成错误 CS0463。

C#

```
// CS0463.cs
using System;
class MyClass
{
    public static void Main()
    {
        const decimal myDec = 7900000000000000000000000000.0m +
7900000000000000000000000000.0m; // CS0463
        Console.WriteLine(myDec.ToString());
    }
}
```

编译器错误 CS0468

项目 • 2023/04/07

类型“type1”类型“type2”间具有多义性

当正在编译的程序集中的两种类型具有相同的完全限定名时，会生成此错误。 如果这两种类型都在添加的模块中，或者一种在添加的模块中，另一个在源中，也可能出现此错误。

> **备注**
>
> Roslyn 中不再使用此编译器错误。

编译器错误 CS0470

项目 • 2023/04/07

方法“method”不能为“type”类型实现“accessor”接口访问器。 使用显式接口实现。

访问器尝试实现接口时，会生成此错误。 必须使用显式接口实现。

示例

下面的示例生成 CS0470。

C#

```
// CS0470.cs
// compile with: /target:library

interface I
{
   int P { get; }
}

class MyClass : I
{
   public int get_P() { return 0; }   // CS0470
   public int P2 { get { return 0;} }   // OK
}
```

编译器错误 CS0471

项目 • 2023/04/07

“name”方法不是泛型方法。 如果原打算使用表达式列表，请用括号将 < 表达式括起来。

如果代码中包含的表达式列表不带括号，则会生成此错误。

示例

下面的示例生成 CS0471：

C#

```
// CS0471.cs
// compile with: /t:library
class Test
{
    public void F(bool x, bool y) {}
    public void F1()
    {
        int a = 1, b = 2, c = 3;
        F(a<b, c>(3));    // CS0471
        // To resolve, try the following instead:
        // F((a<b), c>(3));
    }
}
```

① 备注

Roslyn 中不再使用此编译器错误。 上一个示例应会成功编译。

编译器错误 CS0473

项目 • 2023/04/07

显式接口实现“method name”与多个接口成员匹配。 实际选择哪个接口成员取决于实现。 请考虑改用非显式实现。

在某些情况下，泛型方法可能获得与非泛型方法相同的签名。 问题是，在公共语言基础结构 (CLI) 元数据系统中，无法明确地说明哪个方法绑定到哪个插槽。 由 CLI 来做出决定。

更正此错误

若要更正此错误，请消除显式实现，并在隐式实现 `public int TestMethod(int)` 中实现这两个接口方法。

示例

以下代码生成 CS0473：

C#

```
public interface ITest<T>
{
    int TestMethod(int i);
    int TestMethod(T i);
}

public class ImplementingClass : ITest<int>
{
    int ITest<int>.TestMethod(int i) // CS0473
    {
        return i + 1;
    }

    public int TestMethod(int i)
    {
        return i - 1;
    }
}

class T
{
    static int Main()
    {
        ImplementingClass a = new ImplementingClass();
        if (a.TestMethod(0) != -1)
```

```
            return -1;

        ITest<int> i_a = a;
        System.Console.WriteLine(i_a.TestMethod(0).ToString());
        if (i_a.TestMethod(0) != 1)
            return -1;

        return 0;
    }
}
```

另请参阅

- 可疑的不明确重载，第二部分

编译器错误 CS0500

项目 • 2023/04/07

“class member”无法声明主体，因为它标记为 abstract

abstract 方法不能包含其实现。

下面的示例生成 CS0500：

C#

```
// CS0500.cs
namespace x
{
   abstract public class clx
   {
      abstract public void f(){}   // CS0500
      // try the following line instead
      // abstract public void f();
   }

   public class cly
   {
      public static int Main()
      {
         return 0;
      }
   }
}
```

编译器错误 CS0502

项目 • 2023/04/07

“member”不能既是抽象的又是密封的

类成员不能既是 抽象的 又是 密封的。

下面的示例生成 CS0502：

C#

```
// CS0502.cs
public class B
{
   abstract public void F();
}

public class C : B
{
   abstract sealed override public void F()   // CS0502
   {
   }
}

public class CMain
{
    public static void Main()
    { }
}
```

编译器错误 CS0503

项目 • 2023/04/07

抽象方法“method”不能标记为 virtual

将成员方法同时标记为 abstract 和 virtual 的做法是多余的，因为 **abstract** 意味着 **virtual**。

下面的示例生成 CS0503：

C#

```
// CS0503.cs
namespace x
{
   abstract public class clx
   {
      abstract virtual public void f();   // CS0503
   }
}
```

编译器错误 CS0504

项目 • 2023/04/07

常数“variable”不能标记为 static

如果变量是 const 变量，则它也是 static 变量。 如果需要一个既是 const 又是 static 的变量，只需将该变量声明为 const 即可；如果只需要 static 变量，只需将其标记为 static。

下面的示例生成 CS0504：

C#

```
// CS0504.cs
namespace x
{
   abstract public class clx
   {
      static const int i = 0;   // CS0504, cannot be both static and const
      abstract public void f();
   }
}
```

编译器错误 CS0505

项目 • 2023/04/07

“member1”: 无法重写，因为“member2”不是函数

类声明已尝试重写基类中的非方法。 重写必须与成员类型相匹配。 如果所需方法的名称要与基类中某方法的名称相同，请在基类的方法声明上使用 new （而不是 override）。

下面的示例生成 CS0505：

C#

```
// CS0505.cs
// compile with: /target:library
public class clx
{
   public int i;
}

public class cly : clx
{
   public override int i() { return 0; }   // CS0505
}
```

编译器错误 CS0506

项目 • 2023/04/07

“function1”：继承成员“function2”未标记为“virtual”、“abstract”或“override”，无法进行重写

已重写的方法未显式标记为 **virtual**、 **abstract**或 `override`。

下面的示例生成 CS0506：

C#

```
// CS0506.cs
namespace MyNameSpace
{
   abstract public class ClassX
   {
      public int i = 0;

      public int f()
      {
         return 0;
      }
      // Try the following definition for f() instead:
      // abstract public int f();
   }

   public class ClassY : ClassX
   {
      public override int f()   // CS0506
      {
         return 0;
      }

      public static int Main()
      {
         return 0;
      }
   }
}
```

编译器错误 CS0507

项目 • 2023/05/10

“function1”：重写“access”继承成员“function2”时不能更改访问修饰符

尝试在方法重写中更改访问规范。

示例 1

下面的示例生成 CS0507。

C#

```
// CS0507.cs
abstract public class clx
{
   virtual protected void f() {}
}

public class cly : clx
{
   public override void f() {}   // CS0507
   public static void Main() {}
}
```

示例 2

如果一个类尝试重写在引用的元数据中定义的标记为 `protected internal` 的方法时，也会发生 CS0507。 在此情况下，重写方法应标记为 `protected`。

C#

```
// CS0507_b.cs
// compile with: /target:library
abstract public class clx
{
   virtual protected internal void f() {}
}
```

下面的示例生成 CS0507。

C#

```
// CS0507_c.cs
// compile with: /reference:cs0507_b.dll
public class cly : clx
{
   protected internal override void f() {}   // CS0507
   // try the following line instead
   // protected override void f() {}   // OK

   public static void Main() {}
}
```

编译器错误 CS0508

项目 • 2023/04/07

“Type1”：返回类型必须为“Type2”才能匹配重写的成员“MemberName”

试图更改方法重写中的返回类型。 若要解决此错误，请确保这两种方法声明相同的返回类型。

示例

下面的示例生成 CS0508。

C#

```
// CS0508.cs
// compile with: /target:library
abstract public class Clx
{
   public int i = 0;
   // Return type is int.
   abstract public int F();
}

public class Cly : Clx
{
   public override double F()
   {
      return 0.0;   // CS0508
   }
}
```

编译器错误 CS0509

项目 • 2023/04/07

“class1”：无法从密封类型“class2”派生

密封 类不能作为 基 类。 默认情况下，结构被密封。

下面的示例生成 CS0509：

C#

```
// CS0509.cs
// compile with: /target:library
sealed public class clx {}
public class cly : clx {}   // CS0509
```

编译器错误 CS0513

项目 • 2023/04/07

“function”是抽象的，但它包含在非抽象类“class”中

方法不能是非抽象类的 abstract 成员。

下面的示例生成 CS0513：

C#

```
// CS0513.cs
namespace x
{
   public class clx
   {
      abstract public void f();   // CS0513, abstract member of nonabstract
class
   }
}
```

编译器错误 CS0518

项目 • 2024/01/12

预定义类型“type”未定义或未导入

引起该问题的主要原因是项目没有导入 mscorlib.dll，该文件定义整个 System 命名空间。 这可能由以下一种原因导致：

- 指定了命令行编译器的 NoStandardLib 选项。 NoStandardLib 选项禁止导入 mscorlib.dll。 如果想要定义或创建用户特定的 System 命名空间，请使用此选项。
- 引用了错误的 mscorlib.dll。
- Visual Studio .NET 或 .NET Framework 公共语言运行时安装存在损坏。
- 早期安装的剩余组件与最新安装不兼容。

若要解决此问题，请执行下列操作之一：

- 不指定命令行编译器的 /nostdlib 选项。
- 确保项目引用正确的 mscorlib.dll。
- 重新安装 .NET Framework 公共语言运行时（若以上解决方案均未能解决此问题）。

可选

- 在 Visual Studio 中重新加载项目。

在 GitHub 上与我们协作

可以在 GitHub 上找到此内容的源，还可以在其中创建和查看问题和拉取请求。 有关详细信息，请参阅参与者指南。

.NET

.NET 反馈

.NET 是一个开放源代码项目。 选择一个链接以提供反馈：

提出文档问题

提供产品反馈

编译器错误 CS0520

项目 • 2024/04/11

预定义类型“name”未正确声明

找不到编译器所需的一个或多个文件。 重新安装 .NET 运行时。

在 GitHub 上与我们协作

可以在 GitHub 上找到此内容的源，还可以在其中创建和查看问题和拉取请求。 有关详细信息，请参阅参与者指南。

.NET 反馈

.NET 是一个开放源代码项目。 选择一个链接以提供反馈：

提出文档问题

提供产品反馈

编译器错误 CS0523

项目 • 2023/11/30

“struct1”类型的结构成员“struct2 field”在结构布局中导致循环

一个或多个结构的定义包括形成循环的递归引用。 此限制仅适用于结构，因为结构是值类型。 若要创建递归引用，请将类型声明为作为引用类型的类。

示例 1

以下示例演示自引用类型如何导致生成 CS0523：

C#

```
// CS0523.cs
// compile with: /target:library
struct SelfReferentialStruct
{
    public SelfReferentialStruct other;   // CS0523
}

class SelfReferentialClass
{
    public SelfReferentialClass other;   // OK
}
```

创建自引用结构类型时，它包含的副本类型与成员类型相同。 但是，该成员随后有另一个副本，该副本以递归方式继续。 循环的结果是无法确定类型的大小并发出 CS0523。

示例 2

以下示例演示类型引用循环如何导致生成 CS0523：

C#

```
// CS0523b.cs
// compile with: /target:library
struct ReferenceCycleStruct1
{
    public ReferenceCycleStruct2 other;   // CS0523
}

struct ReferenceCycleStruct2
{
    public ReferenceCycleStruct3 other;   // CS0523
```

```
}

struct ReferenceCycleStruct3
{
    public ReferenceCycleStruct1 other;    // CS0523
}
```

要解决上述错误，可以调整引用，使循环不再形成，或将至少一种结构类型转换为类。 与上一个示例类似，`ReferenceCycleStruct1` 包含 `ReferenceCycleStruct2`，`ReferenceCycleStruct2` 包含 `ReferenceCycleStruct3`，而 `ReferenceCycleStruct3` 又再次包含 `ReferenceCycleStruct1`。

在 GitHub 上与我们协作

可以在 GitHub 上找到此内容的源，还可以在其中创建和查看问题和拉取请求。 有关详细信息，请参阅参与者指南。

.NET

.NET feedback

.NET is an open source project. Select a link to provide feedback:

提出文档问题

提供产品反馈

编译器错误 CS0524

项目 • 2023/04/07

“type”：接口不能声明类型

接口 不能包含用户定义的类型；它应仅包含方法和属性。

示例

下面的示例生成 CS0524：

C#

```
// CS0524.cs
public interface Clx
{
    public class Cly   // CS0524, delete user-defined type
    {
    }
}
```

编译器错误 CS0525

项目 • 2024/11/21

接口不能包含实例字段

接口可以包含方法和属性字段。

下面的示例生成 CS0525：

C#

```
// CS0525.cs
namespace x
{
   public interface clx
   {
      public int i;   // CS0525
   }
}
```

编译器错误 CS0527

项目 • 2023/04/07

接口列表中的类型“type”不是接口

结构 或 接口 可以继承另一个接口，而无法继承任何其他类型。

以下示例生成 CS0527：

C#

```
// CS0527.cs
// compile with: /target:library
public struct clx : int {}   // CS0527 int not an interface
```

编译器错误 CS0528

项目 • 2023/04/07

“interface”已经在接口列表中列出

接口继承列表包含重复。 接口 只能在继承列表中指定一次。

下面的示例生成 CS0528：

C#

```
// CS0528.cs
namespace x
{
   public interface a
   {
   }

   public class b : a, a   // CS0528
   {
      public void Main()
      {
      }
   }
}
```

编译器错误 CS0529

项目 • 2023/04/07

继承接口“interface1”在“interface2”的接口层次结构中导致一个循环

接口 的继承列表包括对其自身的直接或间接引用。 接口不能从自身继承。

下面的示例生成 CS0529：

C#

```
// CS0529.cs
namespace x
{
   public interface a
   {
   }

   public interface b : a, c
   {
   }

   public interface c : b   // CS0529, b inherits from c
   {
   }
}
```

编译器错误 CS0531

项目 • 2023/04/07

“member”：接口成员不能有定义

在 接口 中声明的方法必须在从接口继承的类而不是接口本身中实现。

下面的示例生成 CS0531：

C#

```
// CS0531.cs
namespace x
{
   public interface clx
   {
      int xclx()   // CS0531, cannot define xclx
      // Try the following declaration instead:
      // int xclx();
      {
         return 0;
      }
   }

   public class cly
   {
      public static void Main()
      {
      }
   }
}
```

编译器错误 CS0533

项目 • 2023/04/07

“derived-class member”隐藏继承的抽象成员“derived-class member”

基 类 方法被隐藏。 检查声明中的语法，查看它是否正确。

有关详细信息，请参阅 base。

以下示例生成 CS0533：

C#

```
// CS0533.cs
namespace x
{
   abstract public class a
   {
      abstract public void f();
   }

   abstract public class b : a
   {
      new abstract public void f();   // CS0533
      // try the following lines instead
      // override public void f()
      // {
      // }

      public static void Main()
      {
      }
   }
}
```

编译器错误 CS0534

项目 • 2023/04/07

“function1”不实现继承的抽象成员“function2”

一个类需要实现基类中的所有 抽象 成员，除非该类也是抽象的。

下面的示例生成 CS0534：

C#

```
// CS0534.cs
namespace x
{
   abstract public class clx
   {
      public abstract void f();
   }

   public class cly : clx   // CS0534, no override for clx::f
   {
      // uncomment the following sample override to resolve CS0534
      // override public void f()
      // {
      // }

      public static int Main()
      {
         return 0;
      }
   }
}
```

编译器错误 CS0535

项目 • 2023/04/07

“class”不实现接口成员“member”

类 派生自 接口，但该类未实现一个或多个接口的成员。 类必须实现其派生自的接口的所有成员，或者被声明为 `abstract`。

示例 1

下面的示例生成 CS0535。

C#

```
// CS0535.cs
public interface A
{
   void F();
}

public class B : A {}   // CS0535 A::F is not implemented

// OK
public class C : A {
   public void F() {}
   public static void Main() {}
}
```

示例 2

下面的示例生成 CS0535。

C#

```
// CS0535_b.cs
using System;
class C : IDisposable {}   // CS0535

// OK
class D : IDisposable {
   void IDisposable.Dispose() {}
   public void Dispose() {}

   static void Main() {
      using (D d = new D()) {}
```

```
    }
}
```

编译器错误 CS0537

项目 • 2023/05/10

类 System.Object 不能有基类也不能实现接口

重新生成 System 类库时，以及在 Object 派生自另一个类的情况下，会发生 CS0537。 遇到此错误的可能性非常小。 如果遇到此错误，请勿从类或接口派生 Object：它是整个 .NET 类层次结构的根，因此不会派生自其他任何对象。

编译器错误 CS0538

项目 • 2023/04/07

显式接口声明中的“name”不是接口

尝试显式声明 接口，但未指定接口。

下面的示例生成 CS0538：

C#

```
// CS0538.cs
interface MyIFace
{
   void F();
}

public class MyClass
{
   public void G()
   {
   }
}

class C: MyIFace
{
   void MyIFace.F()
   {
   }

   void MyClass.G()   // CS0538, MyClass not an interface
   {
   }
}
```

编译器错误 CS0539

项目 • 2023/04/07

显式接口声明中的“member”不是接口成员

尝试显式声明不存在的 接口 成员。 应删除该声明或将其改为引用有效的接口成员。

下面的示例生成 CS0539：

C#

```
// CS0539.cs
namespace x
{
   interface I
   {
      void m();
   }

   public class clx : I
   {
      void I.x()   // CS0539
      {
      }

      public static int Main()
      {
         return 0;
      }
   }
}
```

编译器错误 CS0540

项目 • 2023/04/07

“interface member”：包含类型不实现接口“interface”

你试图在非派生自 接口 的 类中实现接口成员。 应删除接口成员的实现，或将接口添加到类的基类列表。

示例 1

下面的示例生成 CS0540。

C#

```
// CS0540.cs
interface I
{
   void m();
}

public class Clx
{
   void I.m() {}   // CS0540
}

// OK
public class Cly : I
{
   void I.m() {}
   public static void Main() {}
}
```

示例 2

下面的示例生成 CS0540。

C#

```
// CS0540_b.cs
using System;
class C {
   void IDisposable.Dispose() {}   // CS0540
}

class D : IDisposable {
   void IDisposable.Dispose() {}
```

```
    public void Dispose() {}

    static void Main() {
       using (D d = new D()) {}
    }
}
```

编译器错误 CS0541

项目 • 2023/04/07

“declaration”：显式接口声明只能在类或结构中声明

显式 interface 声明在 class 或 struct外找到。

以下示例生成 CS0541：

C#

```
// CS0541.cs
namespace x
{
   interface IFace
   {
      void F();
   }

   interface IFace2 : IFace
   {
      void IFace.F();   // CS0541
   }
}
```

编译器错误 CS0542

项目 • 2023/04/07

“用户定义类型”: 成员名称不能与它们的封闭类型相同

类或结构的成员不能与类或结构同名，除非该成员是一个构造函数。

下面的示例生成 CS0542:

C#

```
// CS0542.cs
class C
{
    public int C;
}
```

如果无意中将返回类型放在构造函数中（实际上会使其成为普通方法），则可能导致此错误。 下面的示例将生成 CS0542，因为 `F` 是一种方法，不是构造函数，因为它具有返回类型：

C#

```
// CS0542.cs
class F
{
   // Remove void from F() to resolve the problem.
   void F()   // CS0542, same name as the class
   {
   }
}

class MyClass
{
   public static void Main()
   {
   }
}
```

编译器错误 CS0543

项目 • 2023/04/07

“enumeration”：枚举器值太大，不能适应它的类型

已赋给 枚举 中某元素的值超出该数据类型。

以下示例生成 CS0543：

C#

```
// CS0543.cs
namespace x
{
   enum I : byte
   {a = 255, b, c}   // CS0543
   public class clx
   {
      public static int Main()
      {
         return 0;
      }
   }
}
```

编译器错误 CS0544

项目 • 2023/04/07

“property overrid”：不能重写，因为“non-property”不是属性

尝试将非属性数据类型重写为 属性，这是不允许的。

以下示例生成 CS0544：

C#

```
// CS0544.cs
// compile with: /target:library
public class a
{
   public int i;
}

public class b : a
{
   public override int i {   // CS0544
   // try the following line instead
   // public new int i {
      get
      {
         return 0;
      }
   }
}
```

编译器错误 CS0545

项目 • 2023/04/07

“function”：无法重写，因为“property”没有可重写的 get 访问器

尝试了定义属性访问器的重写，而此时基类没有这样的定义可供重写。 可通过以下方法解决该错误：

- 在基类中添加 `set` 访问器。
- 从派生类中删除 `set` 访问器。
- 通过向派生类的属性添加 new 关键字来隐藏基类属性。
- 使基类属性为 virtual。

有关详细信息，请参阅使用属性。

示例

下面的示例生成CS0545。

C#

```
// CS0545.cs
// compile with: /target:library
// CS0545
public class a
{
   public virtual int i
   {
      set {}

      // Uncomment the following line to resolve.
      // get { return 0; }
   }
}

public class b : a
{
   public override int i
   {
      get { return 0; }
      set {}   // OK
   }
}
```

编译器错误 CS0546

项目 • 2023/04/07

“accessor”: 无法重写，因为“property”没有可重写的 set 访问器

试图对属性的访问器方法之一进行重写时失败，因为访问器不能重写。 如果存在以下情况，则可能会发生此错误：

- 基类属性未声明为 virtual。
- 基类属性未声明你试图重写的 get 或 set 访问器。

如果不想重写基类属性，可以在派生类的属性前面使用 new 关键字。

有关详细信息，请参阅使用属性。

示例

下面的示例生成 CS0546，因为基类没有为属性声明 set 访问器。

C#

```
// CS0546.cs
// compile with: /target:library
public class a
{
   public virtual int i
   {
      get
      {
         return 0;
      }
   }

   public virtual int i2
   {
      get
      {
         return 0;
      }

      set {}
   }
}

public class b : a
{
   public override int i
```

```
    {
        set {}   // CS0546 error no set
    }

    public override int i2
    {
        set {}   // OK
    }
}
```

请参阅

- 属性

编译器错误 CS0547

项目 • 2023/04/07

“property”：属性或索引器不能具有 void 类型

void 作为属性的返回值无效。

有关详细信息，请参阅 属性。

以下示例生成 CS0547：

C#

```
// CS0547.cs
public class a
{
   public void i   // CS0547
   // Try the following declaration instead:
   // public int i
   {
      get
      {
         return 0;
      }
   }
}

public class b : a
{
   public static void Main()
   {
   }
}
```

编译器错误 CS0548

项目 • 2023/04/07

“property”：属性或索引器必须至少有一个访问器

属性必须至少有一个访问器（“get”或“set”）方法。

有关详细信息，请参阅使用属性。

示例

以下示例生成 CS0548。

C#

```
// CS0548.cs
// compile with: /target:library
public class b
{
   public int MyProp {}   // CS0548

   public int MyProp2   // OK
   {
      get
      {
         return 0;
      }
      set {}
   }
}
```

编译器错误 CS0549

项目 • 2023/04/07

“function”是密封类“class”中的新虚拟成员

密封类不能用作基类。 因此，密封类中具有虚拟方法毫无用处。

下面的示例生成 CS0549：

C#

```
// CS0549.cs
// compile with: /target:library
sealed public class MyClass
{
   virtual public void TestMethod() {}   // CS0549
   public void TestMethod2() {}   // OK
}
```

编译器错误 CS0550

项目 • 2023/04/07

“accessor”添加了未在接口成员“property”中找到的访问器

派生类中属性的实现包含基接口中未指定的访问器。

有关详细信息，请参阅使用属性。

示例

下面的示例生成 CS0550。

C#

```
// CS0550.cs
namespace x
{
   interface ii
   {
      int i
      {
         get;
         // add the following accessor to resolve this CS0550
         // set;
      }
   }

   public class a : ii
   {
      int ii.i
      {
         get
         {
            return 0;
         }
         set {}   // CS0550  no set in interface
      }

      public static void Main() {}
   }
}
```

编译器错误 CS0551

项目 • 2023/04/07

显式接口实现“implementation”缺少访问器“accessor”

显式实现接口属性的类必须实现该接口所定义的所有访问器。

有关详细信息，请参阅使用属性。

示例

以下示例生成 CS0551。

C#

```
// CS0551.cs
// compile with: /target:library
interface ii
{
   int i
   {
      get;
      set;
   }
}

public class a : ii
{
   int ii.i { set {} }   // CS0551

   // OK
   int ii.i
   {
      set {}
      get { return 0; }
   }
}
```

编译器错误 CS0552

项目 • 2023/04/07

“conversion routin”：用户定义的与接口之间的相互转换

不能创建用户定义的与接口之间的相互转换。 如果需要转换例程，通过使接口成为类或者从接口派生一个类，可解决该错误。

下面的示例生成 CS0552：

C#

```
// CS0552.cs
public interface ii
{
}

public class a
{
   // delete the routine to resolve CS0552
   public static implicit operator ii(a aa) // CS0552
   {
      return new ii();
   }

   public static void Main()
   {
   }
}
```

编译器错误 CS0553

项目 • 2023/04/07

“conversion routin”: 从基类或到基类的用户定义的转换

不允许用户定义的到基类值的转换；你不需要这样的运算符。

下面的示例生成 CS0553：

C#

```
// CS0553.cs
namespace x
{
   public class ii
   {
   }

   public class a : ii
   {
      // delete the conversion routine to resolve CS0553
      public static implicit operator ii(a aa) // CS0553
      {
         return new ii();
      }

      public static void Main()
      {
      }
   }
}
```

编译器错误 CS0554

项目 • 2023/04/07

“conversion routin”：用户定义的转换转换为/自派生类

不允许用户定义的转换转换为派生类的值；不需要此类运算符。

有关用户定义的转换的详细信息，请参阅 C# 语言规范中的第 6 章。

以下示例生成 CS0554：

C#

```
// CS0554.cs
namespace x
{
   public class ii
   {
      // delete the conversion routine to resolve CS0554
      public static implicit operator ii(a aa) // CS0554
      {
         return new ii();
      }
   }

   public class a : ii
   {
      public static void Main()
      {
      }
   }
}
```

编译器错误 CS0555

项目 • 2023/04/07

用户定义的运算符不能采用封闭类型的对象，也不能转换成封闭类型的对象

不允许用户定义的到封闭类值的转换；你不需要这样的运算符。

下面的示例生成 CS0555：

C#

```
// CS0555.cs
public class MyClass
{
   // delete the following operator to resolve this CS0555
   public static implicit operator MyClass(MyClass aa)   // CS0555
   {
      return new MyClass();
   }

   public static void Main()
   {
   }
}
```

编译器错误 CS0556

项目 • 2023/04/07

用户定义的转换必须是转换成封闭类型，或者从封闭类型转换

用户定义的转换例程必须转换为或转换自包含该例程的类。

下面的示例生成 CS0556：

C#

```
// CS0556.cs
namespace x
{
   public class ii
   {
      public class iii
      {
         public static implicit operator int(byte aa)   // CS0556
         // try the following line instead
         // public static implicit operator int(iii aa)
         {
            return 0;
         }
      }

      public static void Main()
      {
      }
   }
}
```

编译器错误 CS0557

项目 • 2024/04/11

类型“class”中有重复的用户定义转换

类中不允许重复转换例程。

以下示例生成 CS0557：

C#

```
// CS0557.cs
namespace x
{
    public class ii
    {
        public class iii
        {
        public static implicit operator int(iii aa)
        {
            return 0;
        }

    // CS0557, delete duplicate
        public static explicit operator int(iii aa)
        {
            return 0;
        }
        }

        public static void Main()
        {
        }
    }
}
```


在 GitHub 上与我们协作

可以在 GitHub 上找到此内容的源，还可以在其中创建和查看问题和拉取请求。 有关详细信息，请参阅参与者指南。

.NET 反馈

.NET 是一个开放源代码项目。 选择一个链接以提供反馈：

提出文档问题

提供产品反馈

编译器错误 CS0558

项目 • 2023/04/07

用户定义的运算符“operator”必须声明为 static 和 public

必须在用户定义的运算符上指定 **静态** 和 **公共** 访问 修饰符 。

下面的示例生成 CS0558：

C#

```
// CS0558.cs
namespace x
{
   public class ii
   {
      public class iii
      {
         static implicit operator int(iii aa)   // CS0558, add public
         {
            return 0;
         }
      }

      public static void Main()
      {
      }
   }
}
```

编译器错误 CS0559

项目 • 2023/04/07

++ 或 -- 运算符的参数类型必须是包含类型

运算符重载的方法声明必须遵循特定的准则。 ++ 和 -- 运算符要求参数的类型与重载该运算符的类型相同。

示例 1

下面的示例生成 CS0559：

C#

```
// CS0559.cs
// compile with: /target:library
public class iii
{
   public static implicit operator int(iii x)
   {
      return 0;
   }

   public static implicit operator iii(int x)
   {
      return null;
   }

   public static int operator ++(int aa)   // CS0559
   // try the following line instead
   // public static iii operator ++(iii aa)
   {
      return (iii)0;
   }
}
```

示例 2

下面的示例生成 CS0559。

C#

```
// CS0559_b.cs
// compile with: /target:library
public struct S
{
```

```
   public static S operator ++(S? s) { return new S(); }   // CS0559
   public static S operator --(S? s) { return new S(); }   // CS0559
}

public struct T
{
// OK
   public static T operator --(T t) { return new T(); }
   public static T operator ++(T t) { return new T(); }

   public static T? operator --(T? t) { return new T(); }
   public static T? operator ++(T? t) { return new T(); }
}
```

编译器错误 CS0562

项目 • 2023/04/07

一元运算符的参数必须是包含类型

运算符重载的方法声明必须遵循特定的准则。 有关详细信息，请参阅运算符重载。

下面的示例生成 CS0562：

C#

```
// CS0562.cs
public class iii
{
    public static implicit operator int(iii x)
    {
        return 0;
    }

    public static implicit operator iii(int x)
    {
        return null;
    }

    public static iii operator +(int aa)   // CS0562
    // try the following line instead
    // public static iii operator +(iii aa)
    {
        return (iii)0;
    }

    public static void Main()
    {
    }
}
```

编译器错误 CS0563

项目 • 2023/04/07

二元运算符的参数之一必须是包含类型

运算符重载的方法声明必须遵循特定的准则。

示例

下面的示例生成 CS0563：

C#

```
// CS0563.cs
public class iii
{
    public static implicit operator int(iii x)
    {
        return 0;
    }
    public static implicit operator iii(int x)
    {
        return null;
    }
    public static int operator +(int aa, int bb)   // CS0563
    // Use the following line instead:
    // public static int operator +(int aa, iii bb)
    {
        return 0;
    }
    public static void Main()
    {
    }
}
```

编译器错误 CS0564

项目 • 2023/04/07

重载移位运算符的第一个操作数的类型必须与包含类型相同，第二个操作数的类型必须是 int 类型

你试图使用错误类型的操作数重载移位运算符（<< 或 >>）。 第一个操作数必须此类型，第二个操作数必须是 `int` 类型。

下面的示例生成 CS0564：

C#

```
// CS0564.cs
using System;
class C
{
   public static int operator << (C c1, C c2) // CS0564
// To correct, change second operand to int, like so:
// public static int operator << (C c1, int c2)
   {
      return 0;
   }
   static void Main()
   {
   }
}
```

编译器错误 CS0567

项目 • 2024/04/11

接口不能包含运算符

接口 定义中不允许使用运算符。

下面的示例生成 CS0567：

C#

```
// CS0567.cs
interface IA
{
   int operator +(int aa, int bb);   // CS0567
}

class Sample
{
   public static void Main()
   {
   }
}
```


在 GitHub 上与我们协作

可以在 GitHub 上找到此内容的源，还可以在其中创建和查看问题和拉取请求。 有关详细信息，请参阅参与者指南。

.NET

.NET 反馈

.NET 是一个开放源代码项目。 选择一个链接以提供反馈：

提出文档问题

提供产品反馈

编译器错误 CS0569

项目 • 2023/04/07

“method2”: 无法重写“method1”，因为该语言不支持它

从另一种语言编写的基类派生时以及编译器不支持正试图重写的方法时，将发生此错误。

编译器错误 CS0570

项目 • 2023/04/07

属性、索引器或事件“name”不受现用语言支持；请尝试直接调用访问器方法“name!”

使用导入的元数据（由另一个编译器生成）时，会出现此错误。 代码尝试使用编译器无法处理的类成员。

示例 1

下面的 C++ 程序使用了其他语言不能使用的属性 RequiredAttributeAttribute。

C++

```
// CPP0570.cpp
// compile with: /clr /LD

using namespace System;
using namespace System::Runtime::CompilerServices;

namespace CS0570_Server {
   [RequiredAttributeAttribute(Int32::typeid)]
   public ref struct Scenario1 {
      int intVar;
   };

   public ref struct CS0570Class {
      Scenario1 ^ sc1_field;

      property virtual Scenario1 ^ sc1_prop {
         Scenario1 ^ get() { return sc1_field; }
      }

      Scenario1 ^ sc1_method() { return sc1_field; }
   };
};
```

示例 2

下面的示例生成 CS0570。

C#

```
// CS0570.cs
// compile with: /reference:CPP0570.dll
using System;
```

```
using CS0570_Server;

public class C {
   public static int Main() {
      CS0570Class r = new CS0570Class();
      r.sc1_field = null;   // CS0570
      object o = r.sc1_prop;   // CS0570
      r.sc1_method();   // CS0570
   }
}
```

编译器错误 CS0571

项目 • 2023/04/07

“function”：无法显式调用运算符或访问器

某些运算符具有内部名称。 例如，op_Increment 是 ++ 运算符的内部名称。 不应使用或显式调用这类方法名称。

下面的示例生成 CS0571：

C#

```
// CS0571.cs
public class MyClass
{
   public static MyClass operator ++ (MyClass c)
   {
      return null;
   }

   public static int prop
   {
      get
      {
         return 1;
      }
      set
      {
      }
   }

   public static void Main()
   {
      op_Increment(null);   // CS0571
      // use the increment operator as follows
      // MyClass x = new MyClass();
      // x++;

      set_prop(1);      // CS0571
      // try the following line instead
      // prop = 1;
   }
}
```

编译器错误 CS0572

项目 • 2023/04/07

“type”：无法通过表达式引用类型；请另外尝试“path_to_type”

尝试通过标识符访问类的成员，但这在此情况下不被允许。

以下示例生成 CS0572：

C#

```
// CS0572.cs
using System;
class C
{
   public class Inner
   {
      public static int v = 9;
   }
}

class D : C
{
   public static void Main()
   {
      C cValue = new C();
      Console.WriteLine(cValue.Inner.v);   // CS0572
      // try the following line instead
      // Console.WriteLine(C.Inner.v);
   }
}
```

编译器错误 CS0573

项目 • 2024/12/20

“字段声明”：结构中不能有实例字段初始值

无法初始化 struct 的实例字段。 值类型的字段将初始化为其默认值，引用类型字段将初始化为 `null`。

> **备注**
>
> 可以在其声明中初始化结构的实例字段或属性。 有关详细信息，请参阅 结构类型 一文 部分 结构初始化和默认值。

例

下面的示例生成 CS0573：

C#

```
// CS0573.cs
namespace x
{
    public class clx
    {
        public static void Main()
        {
        }
    }

    public struct cly
    {
        clx a = new clx();   // CS0573
        // clx a;            // OK
        int i = 7;           // CS0573
        // int i;            // OK
    }
}
```

编译器错误 CS0574

项目 • 2023/04/07

析构函数的名称必须与类的名称匹配

终结器的名称必须为前面带波形符 (~) 的类名称。

下面的示例生成 CS0574：

C#

```
// CS0574.cs
namespace x
{
   public class iii
   {
      ~iiii()   // CS0574
      {
      }

      public static void Main()
      {
      }
   }
}
```

编译器错误 CS0575

项目 • 2024/12/17

只有类类型才能包含析构函数

一个结构不能包含终结器。

以下示例生成 CS0575：

C#

```
// CS0575.cs
namespace x
{
   public struct iii
   {
      ~iii()   // CS0575
      {
      }

      public static void Main()
      {
      }
   }
}
```

编译器错误 CS0577

项目 • 2023/12/22

Conditional 特性对“function”无效，因为它是构造函数、析构函数、运算符或显式接口实现

> **备注**
>
> “析构函数”是**终结器**已弃用的一个术语。

`Conditional` 无法应用于指定的方法。

例如，你无法对显式接口定义使用某些特性。 下面的示例生成 CS0577：

C#

```
// CS0577.cs
// compile with: /target:library
interface I
{
   void m();
}

public class MyClass : I
{
   [System.Diagnostics.Conditional("a")]   // CS0577
   void I.m() {}
}
```


在 GitHub 上与我们协作

可以在 GitHub 上找到此内容的源，还可以在其中创建和查看问题和拉取请求。 有关详细信息，请参阅参与者指南。

.NET

.NET 反馈

.NET 是一个开放源代码项目。 选择一个链接以提供反馈：

提出文档问题

提供产品反馈

编译器错误 CS0578

项目 • 2023/04/07

Conditional 特性对“'function!”无效，因为其返回类型不是 void

ConditionalAttribute 不能应用于如下方法：具有除 `void` 以外的返回类型的方法。 这样做的原因是，程序的另一部分可能需要方法的其他返回类型。

示例

下面的示例生成 CS0578。 若要解决此错误，必须删除 ConditionalAttribute，或必须将此方法的返回值更改为 `void`。

C#

```
// CS0578.cs
// compile with: /target:library
public class MyClass
{
    [System.Diagnostics.ConditionalAttribute("a")]   // CS0578
    public int TestMethod()
    {
        return 0;
    }
}
```

编译器错误 CS0579

项目 • 2023/04/07

“attribute”属性重复

不可多次指定相同的属性，除非该属性在其 AttributeUsage 中指定 AllowMultiple=true。

示例

下面的示例生成 CS0579。

C#

```
// CS0579.cs
using System;
public class MyAttribute : Attribute
{
}

[AttributeUsage(AttributeTargets.All,AllowMultiple=true)]
public class MyAttribute2 : Attribute
{
}

public class z
{
    [MyAttribute, MyAttribute]     // CS0579
    public void zz()
    {
    }

    [MyAttribute2, MyAttribute2]   // OK
    public void zzz()
    {
    }

    public static void Main()
    {
    }
}
```

编译器错误 CS0582

项目 • 2023/04/07

Conditional 在接口成员上无效

ConditionalAttribute 在接口成员上无效。

下面的示例生成 CS0582：

C#

```
// CS0582.cs
// compile with: /target:library
using System.Diagnostics;
interface MyIFace
{
   [ConditionalAttribute("DEBUG")]   // CS0582
   void zz();
}
```

编译器错误 CS0583

项目 • 2023/04/07

内部编译器错误。 编译器中发生了内部错误。 若要解决此问题，请尝试简化或更改以上所列位置附近的程序。 列表顶部的位置更接近发生内部错误的点。

> **备注**
>
> Roslyn 中不再使用此编译器错误。

编译器错误 CS0584

项目 • 2023/04/07

内部编译器错误: 步骤“stage”符号“symbol”

尝试确定编译器是否因为无法分析意外的语法而产生故障。 如果不是这种情况，请参阅 Visual Studio 反馈选项。

> **备注**
>
> Roslyn 中不再使用此编译器错误。

编译器错误 CS0585

项目 • 2023/04/07

内部编译器错误: 步骤“stage”

尝试确定编译器是否由于无法分析意外的语法而产生故障。 如果不是这种情况，请参阅 Visual Studio 反馈选项。

> **备注**
>
> Roslyn 中不再使用此编译器错误。

编译器错误 CS0586

项目 • 2023/04/07

内部编译器错误: 步骤“stage”

尝试确定编译器是否由于无法分析意外的语法而产生故障。 如果不是这种情况，请参阅 Visual Studio 反馈选项。

> **备注**
>
> Roslyn 中不再使用此编译器错误。

编译器错误 CS0587

项目 • 2023/04/07

内部编译器错误: 步骤“stage”

尝试确定编译器是否由于无法分析意外的语法而产生故障。 如果不是这种情况，请参阅 Visual Studio 反馈选项。

> **备注**
>
> Roslyn 中不再使用此编译器错误。

编译器错误 CS0588

项目 • 2023/04/07

内部编译器错误: 步骤“LEX”

尝试确定编译器是否因为无法分析意外的语法而产生故障。 如果不是这种情况，请参阅 Visual Studio 反馈选项。

> **备注**
>
> Roslyn 中不再使用此编译器错误。

编译器错误 CS0589

项目 • 2023/04/07

内部编译器错误: 步骤“PARSE”

尝试确定编译器是否由于无法分析意外的语法而产生故障。 如果不是这种情况，请参阅 Visual Studio 反馈选项。

> **备注**
>
> Roslyn 中不再使用此编译器错误。

编译器错误 CS0590

项目 • 2023/04/07

用户定义的运算符不能返回 void

用户定义的运算符的用途是返回对象。

下面的示例生成 CS0590：

C#

```
// CS0590.cs
namespace x
{
   public class a
   {
      public static void operator+(a A1, a A2)   // CS0590
      {
      }

      // try the following user-defined operator
      /*
      public static a operator+(a A1, a A2)
      {
         return A2;
      }
      */

      public static int Main()
      {
         return 1;
      }
   }
}
```

编译器错误 CS0592

项目 • 2023/03/16

属性“attribute”对于此声明类型无效。 仅对“type”声明有效。

定义属性时，通过指定 `AttributeTargets` 值，可以定义该属性可应用于哪些构造。 在下面的示例中，`MyAttribute` 属性仅可应用于接口，因为 `AttributeUsage` 属性指定 `AttributeTargets.Interface`。 产生此错误的原因在于属性应用于类（类 `A`）。

示例

下面的示例生成 CS0592：

C#

```
// CS0592.cs
using System;

[AttributeUsage(AttributeTargets.Interface)]
public class MyAttribute : Attribute
{
}

[MyAttribute]
// Generates CS0592 because MyAttribute is not valid for a class.
public class A
{
    public static void Main()
    {
    }
}
```

请参阅

- 特性

编译器错误 CS0594

项目 • 2023/04/07

浮点常量超出“type”类型的范围

分配给浮点变量的值对于此数据类型的变量而言过大。 请参阅 整型表 以获取有关数据类型中允许的值范围的信息。

以下示例生成 CS0594：

C#

```
// CS0594.cs
namespace MyNamespace
{
   public class MyClass
   {
      public static void Main()
      {
         float f = 6.77777777777E400;   // CS0594, value too large
      }
   }
}
```

编译器错误 CS0596

项目 • 2023/04/07

Guid 特性必须用 ComImport 特性指定

使用 **ComImport** 特性时，必须存在 **Guid** 特性。

下面的示例生成 CS0596：

C#

```
// CS0596.cs
using System.Runtime.InteropServices;

namespace x
{
   [ComImport]   // CS0596
   // try the following line to resolve this CS0596
   // [ComImport, Guid("00000000-0000-0000-0000-000000000001")]
   public class a
   {
   }

   public class b
   {
      public static void Main()
      {
      }
   }
}
```

编译器错误 CS0601

项目 • 2023/04/07

必须在标记为 "static" 和 "extern" 的方法上指定 DllImport 特性

在没有正确的访问关键字的方法上使用 `DllImport` 特性。

以下示例生成 CS0601：

C#

```
// CS0601.cs
using System.Runtime.InteropServices;
using System.Text;

public class C
{
   [DllImport("KERNEL32.DLL")]
   extern int GetCurDirectory(int bufSize, StringBuilder buf);   // CS0601
   // Try the following line instead:
   // static extern int GetCurDirectory(int bufSize, StringBuilder buf);
}

public class MainClass
{
   public static void Main ()
   {
   }
}
```

编译器错误 CS0609

项目 • 2023/04/07

不能在标记为 override 的索引器上设置 IndexerName 特性

名称特性 (**IndexerNameAttribute**) 不能应用于是一个重写的索引属性。 有关详细信息，请参阅 索引器。

以下示例生成 CS0609：

C#

```
// CS0609.cs
using System;
using System.Runtime.CompilerServices;

public class idx
{
   public virtual int this[int iPropIndex]
   {
      get
      {
         return 0;
      }
      set
      {
      }
   }
}

public class MonthDays : idx
{
   [IndexerName("MonthInfoIndexer")]   // CS0609, delete to resolve this
CS0609
   public override int this[int iPropIndex]
   {
      get
      {
         return 0;
      }
      set
      {
      }
   }
}

public class test
{
   public static void Main(string[] args)
   {
```

```
    }
}
```

> **备注**
>
> Roslyn 中不再使用此编译器错误。 前面的代码应会成功编译。

编译器错误 CS0610

项目 • 2023/04/07

字段或属性不能是“type”类型

有一些不能用作字段或属性的类型。 这些类型包括 System.ArgIterator 和 System.TypedReference。

由于使用 System.TypedReference 作为字段，下面的示例生成 CS0610：

C#

```
// CS0610.cs
public class MainClass
{
   System.TypedReference i;   // CS0610
   public static void Main ()
   {
   }

   public static void Test(System.TypedReference i)   // OK
   {
   }
}
```

编译器错误 CS0616

项目 • 2023/04/07

“class”不是特性类

尝试了在属性块中使用非特性类。 所有属性类型都必须从 System.Attribute 继承。

示例 1

下面的示例生成 CS0616。

C#

```
// CS0616.cs
// compile with: /target:library
[CMyClass(i = 5)]   // CS0616
public class CMyClass {}
```

示例 2

下面的示例演示可以如何定义属性：

C#

```
// CreateAttrib.cs
// compile with: /target:library
using System;

[AttributeUsage(AttributeTargets.Class|AttributeTargets.Interface)]
public class MyAttr : Attribute
{
   public int Name = 0;
   public int Count = 0;

   public MyAttr (int iCount, int sName)
   {
      Count = iCount;
      Name = sName;
   }
}

[MyAttr(5, 50)]
class Class1 {}

[MyAttr(6, 60)]
interface Interface1 {}
```

编译器错误 CS0619

项目 • 2023/04/07

“member”已过时：“text”

类成员标记有 Obsolete 特性，这样，当引用类成员时，将发出错误。

示例

以下示例生成 CS0619：

C#

```
using System;

public class C
{
    [Obsolete("Use NewMethod instead", true)] // generates an error on use
    public static void OldMethod()
    {
    }

    // this is the method you should be using
    public static void NewMethod()
    {
    }
}

class MyClass
{
   public static void Main()
   {
      C.OldMethod();   // CS0619
   }
}
```

编译器错误 CS0620

项目 • 2023/04/07

索引器不能有 void 类型

索引器 的返回类型不能是 `void`。 索引器必须返回值。

下面的示例生成 CS0620：

C#

```
// CS0620.cs
class MyClass
{
   public static void Main()
   {
      MyClass test = new MyClass();
      System.Console.WriteLine(test[2]);
   }

   void this [int intI]   // CS0620, return type cannot be void
   {
      get
      {
         // will need to return some value
      }
   }
}
```

另请参阅

- void

编译器错误 CS0621

项目 • 2023/04/07

“member”：虚拟成员或抽象成员不能是私有的

不允许私有的 **虚拟** 或 **抽象** 成员。

下面的示例生成 CS0621:

C#

```
// CS0621.cs
abstract class MyClass
{
   private virtual void DoNothing1()   // CS0621
   {
   }

   private abstract void DoNothing2();   // CS0621

   public static void Main()
   {
   }
}
```

编译器错误 CS0622

项目 • 2023/04/07

只能使用数组初始值设定项表达式为数组类型赋值。 请尝试改用 new 表达式。

非数组声明中使用的是适用于初始化数组的语法。

示例

下面的示例生成 CS0622：

C#

```
// CS0622.cs
using System;

public class Test
{
    public static void Main ()
    {
        Test t = { new Test() };   // CS0622
        // Try the following instead:
        // Test[] t = { new Test() };
    }
}
```

编译器错误 CS0625

项目 • 2023/04/07

“field”：使用 StructLayout(LayoutKind.Explicit) 标记的实例字段类型必须具有 FieldOffset 特性。

使用显式 **StructLayout** 特性标记结构时，该结构中的所有字段都必须具有 FieldOffset 特性。 有关详细信息，请参阅 StructLayoutAttribute 类。

下面的示例生成 CS0625：

C#

```
// CS0625.cs
// compile with: /target:library
using System;
using System.Runtime.InteropServices;

[StructLayout(LayoutKind.Explicit)]
struct A
{
   public int i;   // CS0625 not static; an instance field
}

// OK
[StructLayout(LayoutKind.Explicit)]
struct B
{
   [FieldOffset(5)]
   public int i;
}
```

编译器错误 CS0629

项目 • 2023/04/07

条件成员“member”不能在类型“Type Name”中实现接口成员“base class member”

不能在接口的实现上使用条件特性。

以下示例生成 CS0629：

C#

```
// CS0629.cs
interface MyInterface
{
   void MyMethod();
}

public class MyClass : MyInterface
{
   [System.Diagnostics.Conditional("debug")]
   public void MyMethod()    // CS0629, remove the Conditional attribute
   {
   }

   public static void Main()
   {
   }
}
```

编译器错误 CS0635

项目 • 2023/04/07

“attribute”：System.Interop.UnmanagedType.CustomMarshaller 需要命名参数 ComType 和Marshal。

当封送格式为 **System.Interop.UnmanagedType.CustomMarshaller** 时，必须指定 **ComType** 和 **Marshal**参数。

编译器错误 CS0636

项目 • 2023/04/07

FieldOffset 特性只能放置在标记为 StructLayout(LayoutKind.Explicit) 的类型的成员上

如果结构声明中包含标有 **FieldOffset** 特性的任何成员，则必须在该结构声明中使用 **StructLayout(LayoutKind.Explicit)** 特性。 有关详细信息，请参阅 FieldOffset.

下面的示例生成 CS0636：

C#

```
// CS0636.cs
using System;
using System.Runtime.InteropServices;

// To resolve the error, uncomment the following line:
// [StructLayout(LayoutKind.Explicit)]
struct Worksheet
{
   [FieldOffset(4)]public int i;   // CS0636
}

public class MainClass
{
   public static void Main ()
   {
   }
}
```

编译器错误 CS0637

项目 • 2023/04/07

static 字段或 const 字段上不允许存在 FieldOffset 特性。

FieldOffset 特性不能用于标记为 static 或 const的字段。

以下示例生成 CS0637:

C#

```
// CS0637.cs
using System;
using System.Runtime.InteropServices;

[StructLayout(LayoutKind.Explicit)]
public class MainClass
{
   [FieldOffset(3)]   // CS0637
   public static int i;
   public static void Main ()
   {
   }
}
```

编译器错误 CS0641

项目 • 2023/04/07

“attribute”：仅对从 System.Attribute 派生的类有效

使用仅可用在派生自 **System.Attribute**的类上的特性。

以下示例生成 CS0641：

C#

```
// CS0641.cs
using System;

[AttributeUsage(AttributeTargets.All)]
public class NonAttrClass   // CS0641
// try the following line instead
// public class NonAttrClass : Attribute
{
}

class MyClass
{
   public static void Main()
   {
   }
}
```

编译器错误 CS0644

项目 • 2023/04/07

“class1”无法从特殊类“class2”派生

类不能从任何以下基类中显式继承：

- System.Enum
- System.ValueType
- System.Delegate
- System.Array

这些均用作编译器的隐式基类。 例如， System.ValueType 是结构的隐式基类。

以下示例生成 CS0644：

C#

```
// CS0644.cs
class MyClass : System.ValueType   // CS0644
{
}
```

编译器错误 CS0645

项目 • 2024/04/11

标识符太长

类名称或其他标识符不能超过 512 个字符。

在 GitHub 上与我们协作

可以在 GitHub 上找到此内容的源，还可以在其中创建和查看问题和拉取请求。 有关详细信息，请参阅参与者指南。

.NET

.NET 反馈

.NET 是一个开放源代码项目。 选择一个链接以提供反馈：

提出文档问题

提供产品反馈

编译器错误 CS0646

项目 • 2023/04/07

不能对包含索引器的类型指定 DefaultMember 特性

如果一个类或其他类型指定 **System.Reflection.DefaultMemberAttribute**，则它不能包含索引器。 有关详细信息，请参阅 属性。

下面的示例生成 CS0646：

C#

```
// CS0646.cs
// compile with: /target:library
[System.Reflection.DefaultMemberAttribute("x")]   // CS0646
class MyClass
{
   public int this[int index]   // an indexer
   {
      get
      {
         return 0;
      }
   }

   public int x = 0;
}

// OK
[System.Reflection.DefaultMemberAttribute("x")]
class MyClass2
{
   public int prop
   {
      get
      {
         return 0;
      }
   }

   public int x = 0;
}

class MyClass3
{
   public int this[int index]   // an indexer
   {
      get
      {
         return 0;
      }
```

```
    }

    public int x = 0;
}
```

编译器错误 CS0647

项目 • 2024/04/11

发出“attribute”特性时出错 —“reason”

以下示例生成 CS0647：

C#

```
// CS0647.cs
using System.Runtime.InteropServices;

[Guid("z")]   // CS0647, incorrect uuid format.
// try the following line instead
// [Guid("00000000-0000-0000-0000-000000000001")]
public class MyClass
{
   public static void Main()
   {
   }
}
```


在 GitHub 上与我们协作

可以在 GitHub 上找到此内容的源，还可以在其中创建和查看问题和拉取请求。 有关详细信息，请参阅参与者指南。

.NET

.NET 反馈

.NET 是一个开放源代码项目。 选择一个链接以提供反馈：

提出文档问题

提供产品反馈

编译器错误 CS0648

项目 • 2023/04/07

“type”是该语言不支持的类型

从另一种语言（可能是 C++）生成的元数据包含了未标记为托管的类型。 有关该类型的信息包含在元数据中，但该类型不可用于使用 C# 编写的程序中。

编译器错误 CS0653

项目 • 2023/04/07

无法应用特性类“类”，因为它是抽象的

抽象的 自定义特性类不能用作特性。

下面的示例生成 CS0653:

C#

```
// CS0653.cs
using System;

public abstract class MyAttribute : Attribute
{
}

public class My2Attribute : MyAttribute
{
}

[My]   // CS0653
// try the following line instead
// [My2]
class MyClass
{
   public static void Main()
   {
   }
}
```

编译器错误 CS0656

项目 • 2023/04/07

缺少编译器要求的成员“object.member”

存在以下问题之一：

- 你安装的公共语言运行时已损坏。
- 你具有对某个程序集的引用，该程序集定义同样位于公共语言运行库中的类型。 但是，程序集类型的定义方式与 C# 编译器所期望的不同。

检查引用，确保你使用的是正确版本的公共语言运行时。

编译器错误 CS0662

项目 • 2023/04/07

“method”不能只在 ref 参数上指定 Out 特性。 同时使用 In 或 Out 特性，或者同时弃用。

接口方法具有一个参数，它使用仅带 Out 特性的 ref 。 使用 `ref` Out **特性的** 参数必须使用 In 特性。

以下示例生成 CS0662：

C#

```
// CS0662.cs
using System.Runtime.InteropServices;

interface I
{
   void method([Out] ref int i);   // CS0662
   // try one of the following lines instead
   // void method(ref int i);
   // void method([Out, In]ref int i);
}

class test
{
   public static void Main()
   {
   }
}
```

编译器错误 CS0664

项目 • 2023/04/07

无法将 Double 类型隐式转换为“type”类型；请使用“suffix”后缀创建此类型

赋值无法完成，请使用后缀更正指令。 每个类型的文档确定该类型的相应后缀。 有关转换的详细信息，请参阅强制转换和类型转换。

下面的示例生成 CS0664：

C#

```
// CS0664.cs
class Example
{
    static void Main()
    {
        decimal d1 = 1.0;   // CS0664, because 1.0 is interpreted
                            // as a double.

        // Try the following line instead.
        decimal d2 = 1.0M;  // The M tells the compiler that 1.0 is a
                            // decimal.
        Console.WriteLine(d2);
    }
}
```

请参阅

- 类型转换表

编译器错误 CS0666

项目 • 2023/04/07

“member”：结构中已声明新的保护成员

结构 不能为 抽象 ，并且它始终隐式 密封。 由于结构不支持继承，因此结构中的 保护 成员概念没有意义。 有关详细信息，请参阅继承。

示例

下面的示例生成 CS0666：

C#

```
// CS0666.cs
class M
{
    static void Main()
    {
    }
}

struct S
{
    protected int x;   // CS0666
}
```

编译器错误 CS0667

项目 • 2023/04/07

功能“invalid feature”已弃用。 请改用“valid feature”。

尝试使用的功能现已弃用。 请更新代码，以改用有效的功能。

编译器错误 CS0668

项目 • 2023/04/07

两个索引器的名称不同；在类型中的每个索引器上的 IndexerName 特性都必须使用相同的名称

对于类型中的所有索引器，传递给 **IndexerName** 特性的值必须是相同的。 有关 **IndexerName** 特性的详细信息，请参阅 IndexerNameAttribute 类。

下面的示例生成 CS0668：

C#

```
// CS0668.cs
using System;
using System.Runtime.CompilerServices;

class IndexerClass
{
   [IndexerName("IName1")]
   public int this [int index]   // indexer declaration
   {
      get
      {
         return index;
      }
      set
      {
      }
   }

   [IndexerName("IName2")]
   public int this [string s]    // CS0668, change IName2 to IName1
   {
      get
      {
         return int.Parse(s);
      }
      set
      {
      }
   }

   void Main()
   {
   }
}
```

编译器错误 CS0669

项目 • 2023/04/07

具有 ComImport 特性的类不能有用户定义的构造函数。

公共语言运行时中的 COM 互操作层为 ComImport 类提供构造函数。 因此，COM 对象可以用作运行时中的托管对象。

下面的示例生成 CS0669:

C#

```
// CS0669.cs
using System.Runtime.InteropServices;
[ComImport, Guid("00000000-0000-0000-0000-000000000001")]
class TestClass
{
   TestClass()   // CS0669, delete constructor to resolve
   {
   }

   public static void Main()
   {
   }
}
```

编译器错误 CS0670

项目 • 2024/04/11

字段不能有 void 类型

将字段声明为类型 void。

以下示例生成 CS0670：

C#

```
// CS0670.cs
class C
{
   void f;   // CS0670
   // try the following line instead
   // public int f;

   public static void Main()
   {
      C myc = new C();
      myc.f = 0;
   }
}
```


在 GitHub 上与我们协作

可以在 GitHub 上找到此内容的源，还可以在其中创建和查看问题和拉取请求。 有关详细信息，请参阅参与者指南。

.NET

.NET 反馈

.NET 是一个开放源代码项目。 选择一个链接以提供反馈：

提出文档问题

提供产品反馈

编译器错误 CS0673

项目 • 2023/04/07

无法从 C# 使用 System.Void -- 使用 typeof(void) 获取 void 类型对象。

不能在 C# 中使用**System.Void** 。

以下示例生成 CS0673：

C#

```
// CS0673.cs
class MyClass
{
   public static void Main()
   {
      System.Type t = typeof(System.Void);   // CS0673
      // try the following line instead
      // System.Type t = typeof(void);
   }
}
```

编译器错误 CS0677

项目 • 2023/04/07

“变量”: 可变字段的类型不能是“类型”

使用 `volatile` 关键字声明的字段必须是以下类型之一:

- 任何引用类型
- 任何指针类型（在 `unsafe` 上下文中）
- `sbyte`, **byte**, **short**, `ushort`, `int`, `uint`, `char`, **float**, `bool`
- 基于任何上述类型的枚举类型

下面的示例生成 CS0677:

C#

```
// CS0677.cs
class TestClass
{
   private volatile long i;   // CS0677

   public static void Main()
   {
   }
}
```

编译器错误 CS0678

项目 • 2023/04/07

“variable”: 字段不能既是可变的又是只读的

volatile 和 readonly 关键字的使用互相排斥。

下面的示例生成 CS0678：

C#

```
// CS0678.cs
using System;

class TestClass
{
   private readonly volatile int i;   // CS0678
   // try the following line instead
   // private volatile int i;

   public static void Main()
   {
   }
}
```

编译器错误 CS0681

项目 • 2023/04/07

修饰符“abstract”对于字段无效。 请尝试改用属性。

不能使字段为抽象。 但是，可以有访问该字段的抽象属性。

示例 1

以下示例生成 CS0681：

C#

```
// CS0681.cs
// compile with: /target:library
abstract class C
{
    abstract int num;  // CS0681
}
```

示例 2

改为尝试使用以下代码：

C#

```
// CS0681b.cs
// compile with: /target:library
abstract class C
{
    public abstract int num
    {
       get;
       set;
    }
}
```

编译器错误 CS0682

项目 • 2023/04/07

“type1”无法实现“type2”，因为该语言不支持它

当你试图实现用其他语言编写的接口，而编译器不支持该接口时，将发生此错误。

编译器错误 CS0683

项目 • 2023/04/07

“explicitmethod”显式方法实现无法实现“方法”，因为它是一个访问器

以下示例生成 CS0683:

C#

```
// CS0683.cs
interface IExample
{
   int Test { get; }
}

class CExample : IExample
{
   int IExample.get_Test() { return 0; } // CS0683
   int IExample.Test { get{ return 0; } } // correct
}
```

编译器错误 CS0685

项目 • 2023/04/07

条件成员“member”不能有 out 参数

当在方法上使用 ConditionalAttribute 特性时，该方法可能没有 out 参数。 这是因为在将方法调用编译为 nothing 的情况下，将不会定义用于 out 参数的变量值。 若要避免此错误，从条件方法声明中删除 out 参数或不使用条件特性。

示例

以下示例生成 CS0685：

C#

```
// CS0685.cs
using System.Diagnostics;

class C
{
    [Conditional("DEBUG")]
    void trace(out int i)  // CS0685
    {
        i = 1;
    }
}
```

编译器错误 CS0686

项目 • 2023/04/07

访问器“accessor”不能为“type”类型实现接口成员“member”。 使用显式接口实现。

建议：实现一个接口时，如果其中包含的方法名称和与属性或事件关联的自动生成的方法相冲突，会出现此错误。 属性的 get/set 方法生成为 get_property 和 set_property，事件的 add/remove 方法生成为 add_event 和 remove_event。 如果接口包含以上任一方法，将发生冲突。 若要避免此错误，请使用显式接口实现来实现这些方法。 若要执行此操作，将函数指定为：

```
C#

Interface.get_property() { /* */ }
Interface.set_property() { /* */ }
```

示例 1

下面的示例生成 CS0686：

```
C#

// CS0686.cs
interface I
{
    int get_P();
}

class C : I
{
    public int P
    {
        get { return 1; }  // CS0686
    }
}
// But the following is valid:
class D : I
{
    int I.get_P() { return 1; }
    public static void Main() {}
}
```

示例 2

声明事件时，也会发生此错误。 事件构造自动生成 `add_event` 和 `remove_event` 方法，这两个方法会与接口中同名的方法发生冲突，如以下示例所示：

C#

```
// CS0686b.cs
using System;

interface I
{
    void add_OnMyEvent(EventHandler e);
}

class C : I
{
    public event EventHandler OnMyEvent
    {
        add { }  // CS0686
        remove { }
    }
}

// Correct (using explicit interface implementation):
class D : I
{
    void I.add_OnMyEvent(EventHandler e) {}
    public static void Main() {}
}
```

编译器错误 CS0689

项目 • 2023/04/07

“identifier”是一个类型形参，无法从它进行派生

类型参数无法指定泛型类型的基类或接口。 从特定类或接口派生，或改用从特定泛型类型派生，或包含未知类型作为成员。

以下示例生成 CS0689：

C#

```
// CS0689.cs
class A<T> : T   // CS0689
{
}
```

编译器错误 CS0690

项目 • 2023/04/07

输入文件“file”包含无效的元数据。

可打开元数据文件，但由于某些本地化的问题，元数据已损坏。 这与错误 CS0009相似，只不过可以打开元数据文件。

PE（“进程可执行文件”）是一个可执行文件。

编译器错误 CS0692

项目 • 2023/04/07

重复的类型参数“identifier”

同一名称不能在类型参数列表中使用一次以上。 请重命名或删除重复的类型参数。

示例

下面的示例生成 CS0692：

C#

```
// CS0692.cs
// compile with: /target:library
class C <T, A, T>   // CS0692
{
}

class D <T, T>   // CS0692
{
}
```

编译器错误 CS0694

项目 • 2023/04/07

类型形参“identifier”与包含类型或方法同名

必须为类型形参使用不同名称，因为类型形参不能与包含该类型形参的类型或方法同名。

示例 1

以下示例生成 CS0694。

C#

```
// CS0694.cs
// compile with: /target:library
class C<C> {}   // CS0694
```

示例 2

除了涉及泛型类的以上示例，使用一种方法也可能发生此错误：

C#

```
// CS0694_2.cs
// compile with: /target:library
class A
{
   public void F<F>(F arg);   // CS0694
}
```

编译器错误 CS0695

项目 • 2023/04/07

“generic type”不能同时实现“generic interface”和“generic interface”，原因是它们可以统一以进行某些类型参数替换

当一个泛型类实现同一泛型接口的多个参数化，并且存在一个会使两个接口完全相同的类型参数替换时，将出现此错误。 若要避免此错误，请仅实现一个接口，或者更改类型参数以避免冲突。

下面的示例生成 CS0695：

C#

```
// CS0695.cs
// compile with: /target:library

interface I<T>
{
}

class G<T1, T2> : I<T1>, I<T2>  // CS0695
{
}
```

编译器错误 CS0698

项目 • 2023/04/07

无法从“类”派生泛型类型，因为它是一个特性类

从特性类中派生的任何类都是特性。 特性不能是泛型类型。

下面的示例生成 CS0698:

C#

```
// CS0698.cs
class C<T> : System.Attribute  // CS0698
{
}
```

编译器错误 CS0699

项目 • 2023/04/07

“generic”未定义类型参数“identifier”

类型参数在泛型定义中使用，但该泛型定义未在该泛型的类型参数的声明列表中找到。如果类型参数使用的名称不一致，此情况也可能发生。

以下示例生成 CS0699：

C#

```
// CS0699.cs
class C<T> where U : I   // CS0699 – U is not a valid type parameter
{
}
```

编译器错误 CS0701

项目 • 2023/04/07

“identifier”不是有效的约束。 作为约束使用的类型必须是接口、非密封类或类型形参。

如果密封的类型用作约束，则会发生此错误。 若要解决此错误，请仅使用非密封类型作为约束。

示例

以下示例生成 CS0701。

C#

```
// CS0701.cs
// compile with: /target:library
class C<T> where T : System.String {}   // CS0701
class D<T> where T : System.Attribute {}   // OK
```

编译器错误 CS0702

项目 • 2023/04/07

约束不能为特殊类“identifier”

以下类型不能用作约束：Object、Array 或 ValueType。

示例

下面的示例生成 CS0702：

C#

```
// CS0702.cs
class C<T> where T : System.Array  // CS0702
{
}
```

另请参阅

- 类型参数的约束

编译器错误 CS0703

项目 • 2023/04/07

可访问性不一致：约束类型“identifier”的可访问性比“identifier”低

约束不能强制泛型参数比泛型类本身的可访问性低。 在下面的示例中，虽然将泛型类 C<T> 声明为 public，但约束尝试强制 T 实现内部接口。 即使允许这样做，也只有具有内部访问权限的客户端才能为该类创建参数，因此实际上只有具有内部访问权限的客户端才能使用该类。

若要消除此错误，请确保泛型类的访问级别的限制性不低于边界内出现的任何类或接口。

下面的示例生成 CS0703：

C#

```
// CS0703.cs
internal interface I {}
public class C<T> where T : I  // CS0703 - I is internal; C<T> is public
{
}
```

编译器错误 CS0704

项目 • 2023/04/07

“type”是类型参数，无法在其中执行成员查找

不能通过类型参数指定内部类型。 请尝试显式使用所需的类型。

示例

以下示例生成 CS0704：

C#

```
// CS0704.cs
class B
{
    public class I { }
}

class C<T> where T : B
{
    T.I f;  // CS0704 - member lookup on type parameter
    // Try this instead:
    // B.I f;
}

class CMain
{
    public static void Main() {}
}
```

编译器错误 CS0706

项目 • 2023/04/07

约束类型无效。 作为约束使用的类型必须是接口、非密封类或类型形参。

当在约束子句中使用无效的构造时，将出现此错误。 若要避免此错误，请使用接口或非密封类取代导致此错误的构造。

示例

下面的示例生成 CS0706。

C#

```
// CS0706.cs
// compile with: /target:library
class A {}
class C<T> where T : int[] {}  // CS0706
class D<T> where T : A {}  // OK
```

编译器错误 CS0708

项目 • 2023/04/07

“field”：不能在静态类中声明实例成员

如果在声明静态的类中声明非静态成员，将出现此错误。 不能创建静态类的实例，因此实例变量不会有意义。 **静态** 关键字应应用于静态类的所有成员。

以下示例生成 CS0708：

C#

```
// CS0708.cs
// compile with: /target:library
public static class C
{
   int i;  // CS0708
   static int j;  // OK
}
```

编译器错误 CS0709

项目 • 2023/04/07

“derived class”：无法从静态类“base class”进行派生

无法对静态类进行实例化或派生。 也就是说，静态类不能是任何其他类的基类。

示例

下面的示例生成 CS0709。

C#

```
// CS0709.cs
// compile with: /target:library
public static class Base {}
public class Derived : Base {}   // CS0709
```

编译器错误 CS0711

项目 • 2023/04/07

静态类不能包含析构函数

不能实例化静态类，因此它不需要构造函数或终结器。 若要避免此错误，请从静态类中删除终结器；如果你确实希望构造并销毁实例，请使用非静态类。

下面的示例生成 CS0711：

C#

```
// CS0711.cs
public static class C
{
   ~C()  // CS0711
   {
   }

   public static void Main()
   {
   }
}
```

编译器错误 CS0712

项目 • 2023/04/07

无法创建静态类“static class”的实例

无法创建静态类的实例。 静态类被设计为包含静态字段和方法，但不能实例化。

示例

下面的示例生成 CS0712：

C#

```
// CS0712.cs
public static class SC
{
}

public class CMain
{
    public static void Main()
    {
        SC sc = new SC();  // CS0712
    }
}
```

编译器错误 CS0713

项目 • 2023/04/07

静态类“static type”不能从类型“type”派生。 静态类必须从对象派生。

如果允许这样做，静态类将从基类继承方法和非静态成员，因此它不会是静态的。 因此，不允许这样做。

下面的示例生成 CS0713：

C#

```
// CS0713.cs
public class Base
{
}

public static class Derived : Base  // CS0713
{
}

public class CMain
{
   public static void Main()
   {
   }
}
```

编译器错误 CS0714

项目 • 2023/04/07

“static type”：静态类不能实现接口

接口可能会在对象上定义非静态方法，因此将不能由静态类实现。 若要解决此错误，请确保你的类不尝试实现任何接口。

示例

以下示例生成 CS0714：

C#

```
// CS0714.cs
interface I
{
}

public static class C : I  // CS0714
{
    public static void Main()
    {
    }
}
```

编译器错误 CS0715

项目 • 2023/04/07

“静态类”: 静态类不能包含用户定义的运算符

用户定义的运算符在类的实例上运行。 无法实例化静态类，因此无法创建供运算符执行操作的实例。 因此，用户定义的运算符不可用于静态类。

下面的示例生成 CS0715：

C#

```
// CS0715.cs
public static class C
{
   public static C operator+(C c)  // CS0715
   {
   }

   public static void Main()
   {
   }
}
```

编译器错误 CS0716

项目 • 2023/04/07

无法转换为静态类型“type”

如果你的代码使用强制转换来转换为静态类型，则会发生此错误。 由于对象不能是静态类型的实例，因此强制转换为静态类型并没有任何意义。

示例

下面的示例生成 CS0716：

C#

```
// CS0716.cs

public static class SC
{
    static void F() { }
}

public class Test
{
    public static void Main()
    {
        object o = new object();
        System.Console.WriteLine((SC)o);  // CS0716
    }
}
```

编译器错误 CS0717

项目 • 2023/04/07

“static class”：静态类不能用作约束

不能扩展静态类，因为它们仅包含静态成员，不包含实例成员。 因为它们不能进行扩展，这使它们作为类型参数和约束时无用，因为没有静态类的专用化类型可以存在。

示例

以下示例生成 CS0717：

C#

```
// CS0717.cs

public static class SC
{
    public static void F()
    {
    }
}

public class G<T> where T : SC  // CS0717
{
    public static void Main()
    {
    }
}
```

编译器错误 CS0718

项目 • 2023/04/07

“type”：静态类型不能用作类型参数

因为静态类型不能被实例化，它不能用作泛型参数。 若要解决此错误，请从泛型参数中删除静态类型。

示例

以下示例生成 CS0718：

C#

```
// CS0718.cs
public static class SC
{
    public static void F()
    {
    }
}

public class G<T>
{
}

public class CMain
{
    public static void Main()
    {
        G<SC> gsc = new G<SC>();  // CS0718
    }
}
```

编译器错误 CS0720

项目 • 2023/04/07

“static class”：不能在静态类中声明索引器

索引器在静态类中无意义，因为它们只能与实例一起使用，而你不可能创建静态类型的实例。

示例

以下示例生成 CS0720：

C#

```
// CS0720.cs

public static class Test
{
    public int this[int index]  // CS0720
    {
        get { return 1; }
        set {}
    }

    static void Main() {}
}
```

编译器错误 CS0721

项目 • 2023/04/07

“type”：静态类型不能用作参数

静态类型作为参数没有意义。 由于不能创建静态类型的实例，所以没有实例可以作为参数传递。

下面的示例生成 CS0721：

C#

```
// CS0721.cs
public static class SC
{
}

public class CMain
{
   public void F(SC sc)  // CS0721
   {
   }

   public static void Main()
   {
   }
}
```

编译器错误 CS0722

项目 • 2023/04/07

“type”: 静态类型不能用作返回类型

静态类型用作返回类型时没有意义，因为无法创建静态类型的实例。

下面的示例生成 CS0722：

C#

```
// CS0722.cs
public static class SC
{
}

public class CMain
{
   public SC F()  // CS0722
   {
      return null;
   }

   public static void Main()
   {
   }
}
```

编译器错误 CS0723

项目 • 2024/04/11

无法声明静态类型“type”的变量

无法创建静态类型的实例。

下面的示例生成 CS0723：

C#

```
// CS0723.cs
public static class SC
{
}

public class CMain
{
   public static void Main()
   {
      SC sc = null;  // CS0723
   }
}
```


在 GitHub 上与我们协作

可以在 GitHub 上找到此内容的源，还可以在其中创建和查看问题和拉取请求。 有关详细信息，请参阅参与者指南。

.NET

.NET 反馈

.NET 是一个开放源代码项目。 选择一个链接以提供反馈：

提出文档问题

提供产品反馈

编译器错误 CS0724

项目 • 2023/04/07

在嵌套在最近的封闭 catch 子句内部的 finally 子句内不允许使用不带参数的 throw 语句

示例

下面的示例由于 `finally` 子句块内的 `throw` 语句而生成 CS0724：

C#

```
// CS0724.cs
using System;

class Program
{
    static void Test()
    {
        try
        {
            throw new Exception();
        }
        catch
        {
            try
            {
            }
            finally
            {
                throw; // CS0724
            }
        }
    }

    static void Main()
    {
    }
}
```

编译器错误 CS0726

项目 • 2023/04/07

“format specifier”不是有效的格式说明符

此错误将出现在调试器中。 在某一调试器窗口中键入变量名时，可在其后加一个逗号，再加一个格式说明符。 示例包括： `myInt, h` 或 `myString,nq`。 当编译器无法识别 C# 中的格式说明符时，会出现此错误。

编译器错误 CS0727

项目 • 2023/04/07

无效的格式说明符

此错误将出现在调试器中。 在某一调试器窗口中键入变量名时，可在其后加一个逗号，再加一个格式说明符。 示例有 myInt, h 或 myString, nq。 当编译器完全无法分析你键入的内容时，将出现此错误。 要解决此错误，请重新键入变量名称，可以选择后跟一个逗号和一个 有效的格式说明符。

编译器错误 CS0729

项目 • 2023/04/07

类型“type”是在此程序集中定义的，但又为它指定了一个类型转发器

类型转发器不能用于在同一程序集中定义的类型。

示例

下面的示例生成 CS0729。

C#

```
// CS0729.cs
// compile with: /target:library
using System.Runtime.CompilerServices;
[assembly:TypeForwardedTo(typeof(TestClass))]   // CS0729
class TestClass {}
```

编译器错误 CS0730

项目 • 2023/04/07

类型“type”是“type”的嵌套类型，无法转发

当你尝试转发嵌套的类时，会生成此错误。

示例

以下示例生成 CS0730。 它包括两个源文件。 首先，编译库文件 `CS0730a.cs`，并编译引用库文件的文件 `CS0730.cs` 。

C#

```
// CS0730a.cs
// compile with: /t:library
public class Outer
{
   public class Nested {}
}
```

C#

```
// CS0730.cs
// compile with: /t:library /r:CS0730a.dll
using System.Runtime.CompilerServices;

[assembly:TypeForwardedToAttribute(typeof(Outer.Nested))]   // CS0730

[assembly:TypeForwardedToAttribute(typeof(Outer))]   // OK
```

编译器错误 CS0731

项目 • 2023/04/07

程序集“assembly”中类型“type”的类型转发器导致循环

格式不正确的导入元数据才会发生此错误。 该错误不会在只有 C# 源时发生。

示例

下面的示例将生成 CS0731。 该示例由三个文件组成：

1. Circular.IL
2. Circular2.IL
3. CS0731.cs

首先将 .IL 文件编译为库，然后编译引用这两个文件的 .cs 代码。

il

```
// Circular.il
// compile with: /DLL /out=Circular.dll
.assembly extern circular2
{
  .ver 0:0:0:0
}
.assembly extern circular3
{
  .ver 0:0:0:0
}
.assembly extern mscorlib
{
  .publickeytoken = (B7 7A 5C 56 19 34 E0 89 )   // .z\V.4..
  .ver 2:0:0:0
}
.assembly Circular
{
  .custom instance void
[mscorlib]System.Runtime.CompilerServices.CompilationRelaxationsAttribute::.
ctor(int32) = ( 01 00 08 00 00 00 00 00 )
  .hash algorithm 0x00008004
  .ver 0:0:0:0
}
.class extern forwarder Circular.Referenced.TypeForwarder
{
  .assembly extern circular2
}
.module Circular.dll
```

```
// MVID: {880C2329-C915-42A0-83E9-9D10C3E6DBD0}
.imagebase 0x00400000
.file alignment 0x00000200
.stackreserve 0x00100000
.subsystem 0x0003       // WINDOWS_CUI
.corflags 0x00000001    //  ILONLY
// Image base: 0x04E40000
// ======== CLASS MEMBERS DECLARATION =========
.class public abstract auto ansi sealed beforefieldinit User
       extends [mscorlib]System.Object
{
  .method public hidebysig static class
[circular2]Circular.Referenced.TypeForwarder
          F() cil managed
  {
    .maxstack  1
    newobj     instance void
[circular2]Circular.Referenced.TypeForwarder::.ctor()
    ret
  }
}
```

il

```
// Circular2.il
// compile with: /DLL /out=Circular2.dll
.assembly extern mscorlib
{
  .publickeytoken = (B7 7A 5C 56 19 34 E0 89 )   // .z\V.4..
  .ver 2:0:0:0
}
.assembly extern Circular
{
  .ver 0:0:0:0
}
.assembly circular2
{
  .custom instance void
[mscorlib]System.Runtime.CompilerServices.CompilationRelaxationsAttribute::.
ctor(int32) = ( 01 00 08 00 00 00 00 00 )
  .hash algorithm 0x00008004
  .ver 0:0:0:0
}
.class extern forwarder Circular.Referenced.TypeForwarder
{
  .assembly extern Circular
}
.module circular2.dll
// MVID: {8B3BE5C8-DBE1-49C4-BC72-DF35F0387C21}
.imagebase 0x00400000
.file alignment 0x00000200
.stackreserve 0x00100000
.subsystem 0x0003       //   WINDOWS_CUI
```

```
.corflags 0x00000001    //  ILONLY
// Image base: 0x04E40000
```

C#

```
// CS0731.cs
// compile with: /reference:circular.dll /reference:circular2.dll
// CS0731 expected
class A {
   public static void Main() {
      User.F();
   }
}
```

编译器错误 CS0733

项目 • 2023/04/07

无法转发泛型类型“GenericType<>”

示例

下面的示例生成 CS0733。 将第一个文件编译为一个库，然后在编译第二个文件时引用它。

C#

```
// CS0733a.cs
// compile with: /target:library
public class GenericType<T>
{
}
```

C#

```
// CS0733.cs
// compile with: /target:library /r:CS0733a.dll
[assembly:
System.Runtime.CompilerServices.TypeForwardedTo(typeof(GenericType<int>))]
// CS0733
```

编译器错误 CS0734

项目 • 2023/04/07

只有在生成 "module" 目标类型时才能指定 /moduleassemblyname 选项

编译器选项 ModuleAssemblyName 应仅用于生成 .netmodule 时。 有关详细信息，请参阅 ModuleAssemblyName（C# 编译器选项）。

若要详细了解如何生成 .netmodule，请参阅 TargetType（C# 编译器选项）的 module 元素。

示例

以下示例生成 CS0734。 若要解决此问题，请将 `<TargetType>module</TargetType>` 添加到编译。

C#

```
// CS0734.cs
// compile with: /moduleassemblyname:A
// CS0734 expected
public class Test {}
```

编译器错误 CS0735

项目 • 2024/04/11

指定为 TypeForwardedTo 特性的参数的类型无效

以下示例生成 CS0735。

C#

```
// CS735.cs
// compile with: /target:library
using System.Runtime.CompilerServices;
[assembly:TypeForwardedTo(typeof(int[]))]   // CS0735
[assembly:TypeForwardedTo(typeof(string))]   // OK
```


在 GitHub 上与我们协作

可以在 GitHub 上找到此内容的源，还可以在其中创建和查看问题和拉取请求。 有关详细信息，请参阅参与者指南。

.NET 反馈

.NET 是一个开放源代码项目。 选择一个链接以提供反馈：

提出文档问题

提供产品反馈

编译器错误 CS0736

项目 • 2023/04/07

“type name”不实现接口成员“member name”。 “method name”无法实现接口成员，因为它是静态的。

在将静态方法隐式或显式声明为接口成员的实现时，会生成此错误。

更正此错误

- 从方法声明中删除 static 修饰符。
- 更改接口方法的名称。
- 重新定义包含类型，使其不从接口继承。

示例

下面的代码生成 CS0736，因为 `Program.testMethod` 被声明为静态的：

C#

```
// cs0736.cs
namespace CS0736
{

    interface ITest
    {
        int testMethod(int x);
    }

    class Program : ITest // CS0736
    {
        public static int testMethod(int x) { return 0; }
        // Try the following line instead.
        // public int testMethod(int x) { return 0; }
        public static void Main() { }
    }
}
```

另请参阅

- 接口

编译器错误 CS0737

项目 • 2023/04/07

“type name”不实现接口成员“member name”。 “method name”无法实现接口成员，因为它不是公共的。

实现接口成员的方法必须具有公共可访问性。 所有接口成员均为 `public`。

更正此错误

1. 向该方法添加 public 访问修饰符。

示例

下面的代码生成 CS0737：

C#

```
// cs0737.cs
interface ITest
{
    // Default access of private with no modifier.
    int Return42();
    // Try the following line instead.
    // public int Return42();
}

struct Struct1 : ITest // CS0737
{
    int Return42() { return (42); }
}

public class Test
{
    public static int Main(string[] args)
    {
        Struct1 s1 = new Struct1();

        return (1);
    }

}
```

另请参阅

- 接口

编译器错误 CS0738

项目 • 2023/04/07

“type name”不实现接口成员“member name”。 “method name”不能实现“interface member”，因为它没有“type name”的匹配返回类型。

类中实现方法的返回值必须与它实现的接口成员的返回值匹配。

更正此错误

1. 更改方法的返回类型使其与接口成员的返回类型匹配。

示例

下面的代码生成 CS0738，因为类方法将返回 `void` ，同名的接口成员返回 `int`：

C#

```
using System;

interface ITest
{
    int TestMethod();
}
public class Test: ITest
{
    public void TestMethod() { } // CS0738
    // Try the following line instead.
    // public int TestMethod();
}
```

另请参阅

- 接口

编译器错误 CS0739

项目 • 2023/04/07

“type name”重复 TypeForwardedToAttribute。

程序集不可具有超过一个到外部类型的 TypeForwardedToAttribute 。

更正此错误

1. 找到并删除重复 TypeForwardedToAttribute。

示例

以下代码生成 CS0739：

C#

```
// CS0739.cs
// CS0739
// Assume that a class Test is declared in a separate dll
// with a namespace that is named cs739dll.
[assembly:
System.Runtime.CompilerServices.TypeForwardedTo(typeof(cs739dll.Test))]
[assembly:
System.Runtime.CompilerServices.TypeForwardedTo(typeof(cs739dll.Test))]
namespace cs0739
{
    class Program
    {
        static void Main(string[] args)
        {
        }
    }
}
```

另请参阅

- TypeForwardedToAttribute

编译器错误 CS0742

项目 • 2023/04/07

查询正文必须以 select 或 group 子句结尾

查询表达式必须以不包含继续符的 `select` 子句或 `group` 子句结尾。

更正此错误

1. 向查询中添加一个 select 子句 或 group 子句 。

示例

下面的代码生成 CS0742：

C#

```
// cs0742.cs
using System.Linq;
public class Test
{
    public static int Main()
    {
        int[] array = { 1, 2, 3 };
        var query = from num in array; // CS0742
        return 1;
    }
}
```

如果 `group` 子句使用 into 关键字将分组结果存储到临时标识符中，则它不能用作查询中的最后一个子句。 仍需要使用 `select` 或另一个 `group` 子句。

请参阅

- LINQ 查询表达式

编译器错误 CS0743

项目 • 2023/04/07

应为上下文关键字 "on"

`join` 子句的模式是 `join`...`in`...`on`...`equals`，如本示例所示：

C#

```
var query = from x in array1
            join y in array2 on x equals y
            select x;
```

更正此错误

1. 将 `on` 关键字添加到 `join` 子句中。

示例

下面的代码生成 CS0743：

C#

```
// cs0743.cs
using System;
using System.Linq;

public class C
{
    public static int Main()
    {
        int[] array1 = { 1, 2, 3, 4, 5, 6 };
        int[] array2 = { 5, 6, 7, 8, 9 };
        var c = from x in array1
                join y in array2 x equals y // CS0743
                select x;
        return 1;
    }
}
```

请参阅

- LINQ 查询表达式

- join 子句

编译器错误 CS0744

项目 • 2023/04/07

应为上下文关键字 "equals"

`join` 子句的模式是 `join` ... `in` ... `on` ... `equals`，如本示例所示：

C#

```
var query = from x in array1
            join y in array2 on x equals y
            select x;
```

更正此错误

1. 将 `equals` 关键字添加到 `join` 子句中。

示例

下面的代码生成 CS0744：

C#

```
// cs0744.cs
using System;
using System.Linq;

public class C
{
    public static int Main()
    {
        int[] array1 = { 1, 2, 3 ,4, 5, 6 };
        int[] array2 = { 5, 6, 7, 8, 9 };
        var c = from x in array1
                join y in array2 on x y // CS0744
                select x;
        return 1;
    }
}
```

请参阅

- LINQ 查询表达式

- join 子句

编译器错误 CS0745

项目 • 2023/04/07

应为上下文关键字 "by"

`group` 子句的模式是 `group...by` 跟一个可选 `into`，如以下示例所示：

C#

```
string[] names = { "Bob", "Bill", "Jonetta", "Mary" };

var query = from name in names
            group name by name[0];
```

或

C#

```
var query2 = from name in names
             group name by name[0] into g
             //...additional query clauses
```

更正此错误

1. 将 `by` 关键字添加到 `group` 子句中。

示例

以下代码生成 CS0745：

C#

```
// cs0745.cs
using System;
using System.Linq;

public class C
{
    public static int Main()
    {
        string[] names = { "Bob", "Bill", "Jonetta", "Mary" };

        var query = from name in names
                    group name name[0]; // CS0745
```

```
            return 1;
        }
    }
```

请参阅

- LINQ 查询表达式
- group 子句

编译器错误 CS0746

项目 • 2023/04/07

无效的匿名类型成员声明符。 匿名类型成员必须使用成员赋值、简单名称或成员访问来声明。

匿名类型必须使用成员赋值、简单名称或成员访问来声明。

更正此错误

1. 确保声明仅使用成员赋值、简单名称或成员访问表达式。

示例

下面的代码在使用 `incorrect_1` 和 `incorrect_2`的声明时生成 CS0746。 下面的声明显示了两种用于声明匿名类型的正确方法。

C#

```
// cs0746.cs
public class C
{
    public static int Main()
    {
        int i = 100;
        string s = "Bottles of beer.";

        var incorrect_1 = new { a.b = 1 }; // CS0746
        var incorrect_2 = new {100, "Bottles of beer."}; // CS0746
        var correct_1 = new { i, s }; //OK
        var correct_2 = new {num = i, message = s}; // OK

        return 1;
    }
}
```

另请参阅

- 匿名类型

编译器错误 CS0752

项目 • 2023/04/07

分部方法不能有 out 参数

分部方法不能有 out 参数 不允许使用 Out 参数，因为如果编译器删除分部方法，则不能保证将 out 参数赋值。

更正此错误

1. 从参数中删除 out 修饰符，改用该方法的返回值，或者从该方法声明中删除分部修饰符。

示例

以下代码生成 CS0752：

C#

```
// cs0752.cs
public partial class C
{
    partial void Part(out int num); // CS0752
    // try the following line instead
    // partial void Part(int num);

    public static int Main()
    {
        return 1;
    }
}
```

另请参阅

- 分部类和方法

编译器错误 CS0753

项目 • 2023/04/07

只有方法、类、结构或接口可以是分部形式。

partial 修饰符只能用于类、结构、接口和方法。

更正此错误

1. 从变量或语言构造中删除 `partial` 修饰符。

示例

下面的代码生成 CS0753：

C#

```
// cs0753.cs
using System;

    public partial class C
    {
        partial int num; // CS0753
        public static int Main()
        {
            return 1;
        }
    }
```

另请参阅

- 分部类和方法

编译器错误 CS0766

项目 • 2023/04/07

分部方法必须具有 void 返回类型。

分部方法无法返回值。 其返回类型必须为 void。

更正此错误

1. 为分部方法提供 void 返回类型，或者将该方法转换为常规（非分部）方法。

示例

下面的示例生成 CS0766：

C#

```
// cs0766.cs
using System;

    public partial class C
    {
        partial int Part(); // CS0766

        // Typically the implementing declaration
        // is contained in a separate file.
        partial int Part() //CS0766
        {
        }

        public static int Main()
        {
            return 1;
        }

    }
```

另请参阅

- 分部类和方法

编译器错误 CS0811

项目 • 2023/04/07

“name”的完全限定名对于调试信息太长。 请在不使用“/debug”选项的情况下编译。

调试信息中的变量和类型名称具有大小约束。

更正此错误

1. 如果无法修改名称，则唯一的备用方法是在不使用 DebugType 选项的情况下编译。

示例

下面的代码生成 CS0811：

C#

```
// cs0811.cs
//Compile with: /debug
using System;
using System.Collections.Generic;

namespace TestNamespace
{
    using Long =
List<List<List<List<List<List<List<List<List<List<List<List<List<List

<List<List<List<List<List<List<List<List<List<List<List<List<List<List<
int>>>>>>>>>>>>>>>>>>>>>>>>>>>>>; // CS0811

    class Test
    {
        static int Main()
        {
            return 1;
        }
    }
}
```

编译器错误 CS0815

项目 • 2023/04/07

无法将“expression”赋予隐式类型的局部变量

用作隐式类型变量的初始值设定项的表达式必须具有一个类型。 由于匿名函数表达式、方法组表达式和 null 文本表达式没有类型，因此它们不是适当的初始值设定项。 不能在隐式类型的变量的声明中用 null 值对其进行初始化，但稍后可为其赋予 null 值。 在 C# 版本 10 中，具有自然类型的 Lambda 表达式和方法组可用作 `var` 声明中的初始化表达式。

更正此错误

1. 为变量提供显式类型。
2. 或者，使用 C# 版本 10 及更高版本指定自然类型。

示例

以下代码生成 CS0815：

C#

```
// cs0815.cs
class Test
{
    public static int Main()
    {
        var d = s => -1; // CS0815
        var e = (string s) => 0; // CS0815 for C# versions before 10
        var p = null; // CS0815
        var del = delegate(string a) { return -1; }; // CS0815
        return -1;
    }
}
```

另请参阅

- 隐式类型的局部变量

编译器错误 CS0818

项目 • 2023/04/07

隐式类型的局部变量必须已初始化

在声明它的同时，必须使用值初始化隐式类型的局部变量。

更正此错误

1. 为变量分配一个值，或为其提供显式类型。

示例

下面的代码生成 CS0818：

C#

```
// cs0818.cs
class A
{
    public static int Main()
    {
        var a; // CS0818
        return -1;
    }
}
```

另请参阅

- 隐式类型的局部变量

编译器错误 CS0819

项目 • 2023/04/07

隐式类型化的变量不能有多个声明符。

显式类型声明中允许有多个声明符，而隐式类型化的变量不允许。

更正此错误

有三个选项：

1. 如果变量的类型相同，请使用显式声明。
2. 在单独的行上声明每个隐式类型化的局部变量并为其赋值。
3. 使用元组析构语法声明变量。 **注意**：此选项在 `using` 语句中不起作用，因为 `Tuple` 不会实现 `IDisposable`。

示例 1

下面的代码生成 CS0819：

C#

```
// cs0819.cs
class Program
{
    public static void Main()
    {
        var a = 3, b = 2; // CS0819

        // First correction option.
        //int a = 3, b = 2;

        // Second correction option.
        //var a = 3;
        //var b = 2;

        // Third correction option.
        //var (a, b) = (3, 2);
    }
}
```

示例 2

下面的代码生成 CS0819：

C#

```
// cs0819.cs
class Program
{
    public static void Main()
    {
        using (var font1 = new Font("Arial", 10.0f),
            font2 = new Font("Arial", 10.0f)) // CS0819
        {
        }

        // First correction option.
        //using (Font font1 = new Font("Arial", 10.0f),
        //    font2 = new Font("Arial", 10.0f))
        //{
        //}

        // Second correction option.
        //using (var font1 = new Font("Arial", 10.0f)
        //{
        //    using (var font2 = new Font("Arial", 10.0f)
        //    {
        //    }
        //}
    }
}
```

另请参阅

- 隐式类型的局部变量
- 解构元组和其他类型

编译器错误 CS0821

项目 • 2023/04/07

隐式类型的局部变量不能是固定值

隐式类型的局部变量和匿名类型在 `fixed` 上下文中不受支持。

更正此错误

1. 可以从变量中删除 `fixed` 修饰符或者赋予该变量显式类型。

示例

下面的代码生成 CS0821：

C#

```
class A
{
    static int x;

    public static int Main()
    {
        unsafe
        {
            fixed (var p = &x) { }
        }
        return -1;
    }
}
```

另请参阅

- 隐式类型的局部变量

编译器错误 CS0822

项目 • 2023/04/07

隐式类型化的局部变量不能是常量

仅在存储匿名类型时才需要隐式类型的局部变量。 在所有其他情况下使用此类变量只是为了方便起见。 如果该变量的值从未更改，则只需为其指定显式类型即可。 如果尝试对隐式类型的局部变量使用 `readonly` 修饰符，则会生成 CS0106。

更正此错误

1. 如果要求该变量为常量或 `readonly`，请为其指定显式类型。

示例

下面的代码生成 CS0822：

C#

```
// cs0822.cs
class A
{

    public static int Main()
    {
        const var x = 0; // CS0822.cs
        return -1;
    }
}
```

另请参阅

- 隐式类型的局部变量

编译器错误 CS0825

项目 • 2023/04/07

上下文关键字“var”可能只出现在局部变量声明内。

更正此错误

1. 如果变量属于类范围，请为其提供显式类型。 否则将其移动在方法内待使用的位置。

示例

下面的代码生成 CS0825，因为 `var` 在类字段上使用：

C#

```
// cs0825.cs
class Test
{
    // Both of these declarations trigger CS0825
    private var genreName;
    private var bookTitles = new List<string>();

    static int Main()
    {
        var totalBooks = 42; // var is OK here
        return -1;
    }
}
```

另请参阅

- 隐式类型的局部变量

编译器错误 CS0828

项目 • 2023/04/07

不能将“expression”赋值给匿名类型属性。

不能使用 null 值、不安全类型、方法组或匿名函数初始化匿名类型。

更正此错误

1. 向赋值的左侧添加类型声明，或将右侧的表达式改为具有可接受类型。

示例

下面的代码生成 CS0828，因为不能使用 null 值初始化匿名类型成员。

C#

```
// cs0828.cs
using System;

public class C
{
    public static int Main()
    {
        var a = 1;
        var c = new { p1 = null }; // CS0828
        return 1;
    }
}
```

另请参阅

- 隐式类型的局部变量

编译器错误 CS0833

项目 • 2023/04/07

匿名类型不能有多个同名属性。

匿名类型与其他类型一样，不能有两个同名称的属性。

更正此错误

1. 为类型中的每个属性指定唯一名称。

示例

以下示例生成 CS0833：

C#

```
// cs0833.cs
using System;

public class C
{
    public static int Main()
    {
        var c = new { p1 = 1, p1 = 2 }; // CS0833
        return 1;
    }
}
```

另请参阅

- 匿名类型

编译器错误 CS0836

项目 • 2023/04/07

无法在常量表达式中使用匿名类型。

常量表达式中仅允许使用可合并常量表达式的命名常量、文本和数学表达式。

更正此错误

1. 使匿名类型变为命名类型。

示例

以下示例演示了生成 CS0836 的一种方式：

C#

```
// cs0836.cs
using System;
[AttributeUsage(AttributeTargets.Class, AllowMultiple = true, Inherited =
false)]
public class A : Attribute
{
    public A(object obj)
    {
    }
}

[A(new { })] // CS0836
public class B
{
}

public class Test
{
    public static int Main()
    {
        return 0;
    }
}
```

编译器错误 CS0837

项目 • 2023/04/07

“is”或“as”运算符的第一个操作数不能是 lambda 表达式、匿名方法或方法组。

Lambda 表达式、匿名方法和方法组不能在 is 或 as的左侧使用。

更正此错误

- 如果错误涉及 `is` 运算符，请记住， `is` 会采用一个值和一种类型，并会告诉你是否可以通过引用、装箱或取消装箱转换将该值转换为该类型。 由于 lambda 不是值并且没有引用、装箱或取消装箱转换，因此 lambda 不适用于 `is`。
- 如果代码误用了 `as`，则更正方法可能是将它更改为强制转换。

示例

下面的示例生成 CS0837：

C#

```
// cs0837.cs
namespace TestNamespace
{
    public delegate void Del();

    class Test
    {
        static int Main()
        {
            bool b1 = (() => { }) is Del;   // CS0837
            bool b2 = delegate() { } is Del;// CS0837
            Del d1 = () => { } as Del;      // CS0837
            Del d2 = delegate() { } as Del; // CS0837
            return 1;
        }
    }
}
```

Collaborate with us on GitHub

.NET feedback

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

The .NET documentation is open source. Provide feedback here.

Open a documentation issue

Provide product feedback

编译器错误 CS0840

项目 • 2024/10/26

“Property name”必须声明主体，因为它未标记为 abstract 或 extern。 自动实现的属性必须定义 get 和 set 访问器。

除非常规属性标记为 `abstract` 或 `extern`，或者是 `partial` 类型的成员，否则它必须提供主体。 自动实现的属性不提供访问器主体，但它们必须指定这两个访问器。 若要创建只读自动实现的属性，请将 set 访问器 `private` 设为 。

更正此错误

1. 提供缺少的正文或访问器，否则对其和/或其封闭类型使用 abstract、extern 或 （类型）修饰符。

示例

下面的示例生成 CS0840：

C#

```
// cs0840.cs
// Compile with /target:library
using System;
class Test
{
    public int myProp { get; } // CS0840

    // to create a read-only property
    // try the following line instead
    public int myProp2 { get; private set; }

}
```

另请参阅

- 自动实现的属性

编译器错误 CS0841

项目 • 2023/04/07

局部变量“name”在声明之前无法使用。

必须先声明名称，然后才能使用该变量。

声明前使用的变量示例

以下示例生成 CS0841：

C#

```
// cs0841.cs
using System;

public class Program
{
    public static void Main()
    {
        j = 5; // CS0841
        int j;
    }
}
```

通过在使用之前移动声明来更正错误

将变量声明移动到错误发生的行前。

C#

```
using System;

public class Program
{
    public static void Main()
    {
        int j;
        j = 5;
    }
}
```

隐藏类型的变量示例

在下面的示例中，意图是将 `parameter` 与 `MyEnum.A` 进行比较。 由于之后声明了一个同类型名的局部变量，因此它隐藏类型 `MyEnum`，此方法中的 `MyEnum` 不再引用 `enum`，而是引用声明的局部变量。

C#

```
using System;

public enum MyEnum
{
    A, B, C
}

public class C
{
    public void M(MyEnum parameter)
    {
        // error CS0841: Cannot use local variable 'MyEnum' before it is
declared
        if (parameter == MyEnum.A)
        {
            return;
        }

        // Change the variable to 'myEnum' to avoid shadowing the 'MyEnum'
type.
        // This change also aligns with the C# coding conventions.
        var MyEnum = parameter;
        Console.WriteLine(MyEnum.ToString());
    }
}
```

编译器错误 CS0842

项目 • 2024/10/26

不能在用 StructLayout(LayoutKind.Explicit) 标记的类型内部使用自动实现的属性。

自动实现的属性具有编译器提供的后盾字段，并且源代码无法访问该字段。 因此，它们与 LayoutKind.Explicit不兼容。

更正此错误

1. 使此属性成为一个常规属性，你可通过它提供访问器正文。

示例

下面的示例生成 CS0842：

C#

```
// cs0842.cs
using System;
using System.Runtime.InteropServices;

namespace TestNamespace
{
    [StructLayout(LayoutKind.Explicit)]
    struct Str
    {
        public int Num // CS0842
        {
            get;
            set;
        }

        static int Main()
        {
            return 1;
        }
    }
}
```

编译器错误 CS0844

项目 • 2023/04/07

局部变量“name”在声明之前无法使用。 局部变量的声明隐藏字段“name”。

标识符在给定块中只能有一种含义。 通过为该标识符引入另一种含义，与类字段同名的局部变量可以隐藏字段。 因此，如果在方法中引用类字段，然后使用相同名称声明局部变量，则编译器将生成错误。

更正此错误

- 使用 `this.num` 引用类字段。
- 为局部变量提供一个与类字段名称不同的名称。

示例

下面的代码生成 CS0844：

C#

```
class Test
    {
        int num;
        public void TestMethod()
        {
            num = 5; // CS0844
            int num = 6;
        }
        public static int Main()
        {
            return 1;
        }
    }
```

请通过使用 `this.num` 引用类字段来更正错误

C#

```
class Test
    {
        int num;
        public void TestMethod()
        {
```

```
            this.num = 5; // Error fixed.
            int num = 6;
        }
        public static int Main()
        {
            return 1;
        }
    }
```

请通过向局部变量提供不同于类字段的名称来更正错误

C#

```
class Test
    {
        int num;
        public void TestMethod()
        {
            num = 5; // Error fixed.
            int num2 = 6;
        }
        public static int Main()
        {
            return 1;
        }
    }
```

编译器错误 CS1001

项目 • 2023/04/07

应输入标识符

没有提供标识符。 标识符是所提供的类、结构、命名空间、方法、变量等的名称。

以下示例声明一个简单的类，但没有为类指定名称：

C#

```
public class //CS1001
{
    public int Num { get; set; }
    void MethodA() {}
}
```

下面示例生成 CS1001，因为声明枚举时，必须指定成员：

C#

```
public class Program
{
    enum Colors
    {
        'a', 'b' // CS1001, 'a' is not a valid int identifier
        // The following line shows examples of valid identifiers:
        // Blue, Red, Orange
    };

    public static void Main()
    {
    }
}
```

即使编译器不使用参数名（例如，在接口定义中），也需要参数名。 这些参数是必需的，这样，使用接口的程序员就会知道一些参数的含义。

C#

```
interface IMyTest
{
    void TestFunc1(int, int);  // CS1001
    // Use the following line instead:
    // void TestFunc1(int a, int b);
}

class CMyTest : IMyTest
```

```
{
    void IMyTest.TestFunc1(int a, int b)
    {
    }
}
```

另请参阅

- 运算符和表达式
- 类型

编译器错误 CS1002

项目 • 2023/04/07

应输入 ;

编译器检测到缺少一个分号。 C# 中每个语句的末尾都需要一个分号。 语句可以跨多行。

下面的示例生成 CS1002：

C#

```
// CS1002.cs
namespace x
{
   abstract public class clx
   {
      int i   // CS1002, missing semicolon

      public static int Main()
      {
         return 0;
      }
   }
}
```

请参阅

- 语句

编译器错误 CS1003

项目 • 2023/04/07

语法错误，应为“char”

对于若干错误条件中的任何一项，编译器将生成此错误。 查看你的代码以查找语法错误。

以下示例生成 CS1003：

C#

```
// CS1003.cs
public class b
{
   public static void Main()
   {
      int[] a;
      a();   // CS1003
   }
}
```

编译器错误 CS1004

项目 • 2023/04/07

重复的“modifier”修饰符

检测到重复的修饰符，如 **访问** 修饰符。

下面的示例生成 CS1004：

C#

```
// CS1004.cs
public class clx
{
   public public static void Main()   // CS1004, two public keywords
   {
   }
}
```

编译器错误 CS1008

项目 • 2023/04/07

应输入类型 byte、sbyte、short、ushort、int、uint、long 或 ulong

某些数据类型（如 enums）只可声明为保存指定类型的数据。

下面的示例生成 CS1008：

C#

```
// CS1008.cs
abstract public class clx
{
   enum splitch : char   // CS1008, char not valid type for enums
   {
      x, y, z
   }

   public static void Main()
   {
   }
}
```

编译器错误 CS1009

项目 • 2024/04/01

转义序列无法识别

意外字符遵循转义序列或字符文本的字符串中的反斜杠 (\)。 编译器需要一个有效的转义字符。 有关详细信息，请参阅字符转义。

下面的示例生成 CS1009。

C#

```
// CS1009-a.cs
class MyClass
{
   static void Main()
   {
      // The following escape sequence causes CS1009:
      string a = "\m";
      // Try the following line instead.
      // string a = "\t";

      // The following character literals causes CS1009:
      // CS1009; a lowercase \u-style Unicode escape sequence must have
exactly 4 hex digits
      string unicodeEscapeSequence = '\u061';
      // CS1009; a hex escape sequence must start with lowercase \x
      string hexEscapeSequence = '\X061';
      // CS1009; an uppercase \U-style Unicode escape sequence must have
exactly 8 hex digits
      string uppercaseUnicodeEscape = '\U0061';
   }
}
```

发生该错误的原因通常是在文件名中使用了反斜杠字符，如下面的示例所示。

C#

```
string filename = "c:\myFolder\myFile.txt";
```

若要纠正该错误，请使用“\”或前面带有 @ 且用引号括起的字符串，如下面的示例所示。

C#

```
// CS1009-b.cs
class MyClass
{
   static void Main()
```

```
    {
        // The following line causes CS1009.
        string filename = "c:\myFolder\myFile.txt";
        // Try one of the following lines instead.
        // string filename = "c:\\myFolder\\myFile.txt";
        // string filename = @"c:\myFolder\myFile.txt";
    }
}
```

另请参阅

- string

在 GitHub 上与我们协作

可以在 GitHub 上找到此内容的源，还可以在其中创建和查看问题和拉取请求。 有关详细信息，请参阅参与者指南。

.NET 反馈

.NET 是一个开放源代码项目。 选择一个链接以提供反馈：

提出文档问题

提供产品反馈

编译器错误 CS1010

项目 • 2023/04/07

常量中有换行符

未正确分隔 字符串 。

下面的示例生成 CS1010：

C#

```
// CS1010.cs
class Sample
{
   static void Main()
   {
      string a = "Hello World;   // CS1010
      // Use the following line, with end quote before semicolon:
      string a = "Hello World"; //
   }
}
```

另请参阅

- 字符串（C# 编程指南）

编译器错误 CS1011

项目 • 2023/04/07

空字符

声明了一个 char 并将其初始化为 null。 初始化 `char` 必须指定一个字符。

以下示例生成 CS1011：

C#

```
// CS1011.cs
class Sample
{
   public char CharField = '';   // CS1011
}
```

编译器错误 CS1012

项目 • 2023/04/07

字符文本中的字符太多

尝试用多个字符初始化 “char” 常量。

在执行数据绑定时，也会发生 CS1012。 例如，下面行将发生错误：

```
<%# DataBinder.Eval(Container.DataItem, 'doctitle') %>
```

改为尝试使用以下行：

```
<%# DataBinder.Eval(Container.DataItem, "doctitle") %>
```

以下示例生成 CS1012：

C#

```
// CS1012.cs
class Sample
{
   static void Main()
   {
      char a = 'xx';   // CS1012
      char a2 = 'x';   // OK
      System.Console.WriteLine(a2);
   }
}
```

编译器错误 CS1013

项目 • 2023/04/07

无效数字

编译器检测到一个格式不正确的数字。 有关整型的详细信息，请参阅 Integral Types Table。

示例

下面的示例生成 CS1013：

C#

```
// CS1013.cs
class Sample
{
    static void Main()
    {
        int i = 0x;   // CS1013
    }
}
```

编译器错误 CS1014

项目 • 2023/04/07

应为 get 或 set 访问器

在属性声明中找到了一个方法声明。 在属性中只能声明 `get` 和 `set` 方法。

有关属性的详细信息，请参阅 Using Properties。

示例

下面的示例生成 CS1014。

C#

```
// CS1014.cs
// compile with: /target:library
class Sample
{
   public int TestProperty
   {
      get
      {
         return 0;
      }
      int z;   // CS1014  not get or set
   }
}
```

编译器错误 CS1015

项目 • 2023/04/07

应是对象、字符串或类类型

尝试将预定义的数据类型传入 catch 块。 仅派生自 System.Exception 的数据类型可传入 `catch` 块。 有关异常的详细信息，请参阅异常和异常处理。

示例

下面的示例生成 CS1015：

C#

```
// CS1015.cs
class Sample
{
    static void Main()
    {
        try
        {
        }
        catch(int)   // CS1015, int is not derived from System.Exception
        {
        }
    }
}
```

编译器错误 CS1017

项目 • 2023/04/07

catch 子句不能跟在 try 语句的常规 catch 子句之后

不接受任何参数的 `catch` 块必须是一系列 `catch` 块中的最后一个。 有关异常的详细信息，请参阅异常和异常处理。

示例

下面的示例生成 CS1017：

C#

```
// CS1017.cs
using System;

namespace x
{
    public class b : Exception
    {
    }

    public class a
    {
        public static void Main()
        {
            try
            {
            }

            catch   // CS1017, must be last catch
            {
            }

            catch(b)
            {
                throw;
            }
        }
    }
}
```

编译器错误 CS1018

项目 • 2023/04/07

应输入关键字“this”或“base”

编译器遇到了不完整的构造函数声明。

示例

下面的示例生成 CS1018 并建议了几种解决此错误的方法：

C#

```
// CS1018.cs
public class C
{
}

public class a : C
{
    public a(int i)
    {
    }

    public a () :   // CS1018
    // possible resolutions:
    // public a () resolves by removing the colon
    // public a () : base() calls C's parameterless constructor
    // public a () : this(1) calls the assignment constructor of class a
    {
    }

    public static int Main()
    {
        return 1;
    }
}
```

编译器错误 CS1021

项目 • 2023/04/07

整数常量太大

由整数文本表示的值大于 UInt64.MaxValue。

下面的示例生成 CS1021:

C#

```
// CS1021.cs
class Program
{
    static void Main(string[] args)
    {
        int a = 18_446_744_073_709_552_000;
    }
}
```

以下代码还会生成 CS1021：

C#

```
using System.Numerics;

class Program
{
    static void Main(string[] args)
    {
        var a = new BigInteger(18_446_744_073_709_552_000);
    }
}
```

若要了解如何实例化其值超出内置数值类型范围的 System.Numerics.BigInteger 实例，请查看 BigInteger 参考页面的。实例化 BigInteger 对象。

编译器错误 CS1022

项目 • 2024/04/11

应输入类型、命名空间定义或文件尾

源代码文件不具有一组匹配的大括号。

下面的示例生成 CS1022：

C#

```
// CS1022.cs
namespace x
{
}
}   // CS1022
```


在 GitHub 上与我们协作

可以在 GitHub 上找到此内容的源，还可以在其中创建和查看问题和拉取请求。 有关详细信息，请参阅参与者指南。

.NET

.NET 反馈

.NET 是一个开放源代码项目。 选择一个链接以提供反馈：

提出文档问题

提供产品反馈

编译器错误 CS1023

项目 • 2023/04/07

嵌入的语句不能是声明或标记语句

嵌入语句（如跟在 **if** 语句后面的语句）不能包含声明，也不能包含标记语句。

下面的示例生成 CS1023 两次：

C#

```
// CS1023.cs
public class a
{
   public static void Main()
   {
      if (1)
         int i;      // CS1023, declaration is not valid here

      if (1)
         xx : i++;   // CS1023, labeled statement is not valid here
   }
}
```

编译器错误 CS1024

项目 • 2023/04/07

应输入预处理器指令

行以井号 (#) 开头，但后面的字符串不是有效的 预处理器指令。

下面的示例生成 CS1024：

C#

```
// CS1024.cs
#import System   // CS1024
```

编译器错误 CS1025

项目 • 2023/04/07

应输入单行注释或行尾

带 预处理器指令 的行不能有多行注释。

以下示例生成 CS1025：

C#

```
#if true /* hello
*/   // CS1025
#endif   // this is a good comment
```

如果你尝试某些无效的预处理器指令，也可能发生 CS1025，如下所示：

C#

```
// CS1025.cs
#define a

class Sample
{
   static void Main()
   {
      #if a 1   // CS1025, invalid syntax
         System.Console.WriteLine("Hello, World!");
      #endif
   }
}
```

编译器错误 CS1026

项目 • 2023/05/09

) 应为

找到不完整的语句。

此错误的一个常见原因是在 ASP.NET 页的内联表达式内放置的是一个语句，而不是一个表达式。 例如，以下示例错误：

ASP.NET (C#)

```
<%=new TimeSpan(DateTime.Now.Ticks - new DateTime(2001, 1, 1).Ticks).Days;%>
```

以下示例正确：

ASP.NET (C#)

```
<%=new TimeSpan(DateTime.Now.Ticks - new DateTime(2001, 1, 1).Ticks).Days %>
```

解释如下：

ASP.NET (C#)

```
<% Response.Write(new TimeSpan(DateTime.Now.Ticks - new DateTime(2001, 1,
1).Ticks).Days); %>
```

下面的示例生成 CS1026：

C#

```
// CS1026.cs
#if (a == b   // CS1026, add closing )
#endif

class x
{
   public static void Main()
   {
   }
}
```

编译器错误 CS1027

项目 • 2024/07/04

应输入 #endif 指令

未找到与指定的 `#if` 指令匹配的 `#endif` 预处理器指令。 或者，编译器可能发现了 `#endregion` 指令而 `#region` 块内不存在匹配的 `#if` 指令。

下面的示例生成 CS1027：

C#

```
// CS1027.cs
#if true   // CS1027, uncomment next line to resolve
// #endif

namespace x
{
   public class clx
   {
      public static void Main()
      {
      }
   }
}
```


在 GitHub 上与我们协作

可以在 GitHub 上找到此内容的源，还可以在其中创建和查看问题和拉取请求。 有关详细信息，请参阅参与者指南。

.NET 反馈

.NET 是一个开放源代码项目。 选择一个链接以提供反馈：

提出文档问题

提供产品反馈

编译器错误 CS1028

项目 • 2023/04/07

意外的预处理器指令

已发现一个意外的 预处理器指令 。

例如， `#endif` 前面没有找到 `#if`。

以下示例生成 CS1028：

C#

```
// CS1028.cs
#endif   // CS1028, no matching #if
namespace x
{
   public class clx
   {
      public static void Main()
      {
      }
   }
}
```

编译器错误 CS1029

项目 • 2023/04/07

#error：“文本”

显示使用 #error 指令定义的错误文本。

下面的示例演示如何创建用户定义的错误：

C#

```
// CS1029.cs
class Sample
{
   static void Main()
   {
      #error Let's give an error here   // CS1029
   }
}
```

编译产生以下输出：

控制台

```
example.cs(9,8): error CS1029: #error: 'Let's give an error here   // CS1029
'
```

编译器错误 CS1031

项目 • 2024/04/11

应为类型

应为类型参数。

示例

下面的示例生成 CS1031：

C#

```
// CS1031.cs
namespace x
{
    public class ii
    {
    }

    public class a
    {
        public static operator +(a aa)   // CS1031
        // try the following line instead
        // public static ii operator +(a aa)
        {
            return new ii();
        }

        public static void Main()
        {
            e = new base;   // CS1031, not a type
            e = new this;   // CS1031, not a type
            e = new ();     // CS1031, not a type
        }
    }
}
```


在 GitHub 上与我们协作

可以在 GitHub 上找到此内容的源，还可以在其中创建和查看问题和拉取请求。 有关详细信息，请参阅参与者指南。

.NET

.NET 反馈

.NET 是一个开放源代码项目。 选择一个链接以提供反馈：

提出文档问题

提供产品反馈

编译器错误 CS1032

项目 • 2024/07/04

不能在文件的第一个标记之后定义或取消定义预处理器符号

`#define` 和 `#undef` 预处理器指令必须用在程序的开头，并且在其他任何关键字之前，例如用在命名空间声明中的那些指令之前。

下面的示例生成 CS1032：

C#

```
// CS1032.cs
namespace x
{
   public class clx
   {
      #define a   // CS1032, put before namespace
      public static void Main()
      {
      }
   }
}
```


在 GitHub 上与我们协作

可以在 GitHub 上找到此内容的源，还可以在其中创建和查看问题和拉取请求。 有关详细信息，请参阅参与者指南。

.NET 反馈

.NET 是一个开放源代码项目。 选择一个链接以提供反馈：

提出文档问题

提供产品反馈

编译器错误 CS1033

项目 • 2023/04/07

源文件已超过在 PDB 中可表示的 16,707,565 行的限制；调试信息将不正确

源代码文件超出了编译器可以处理的最大允许行数。 若要解决此错误，请从原始文件创建两个或多个源代码文件。 最大行数为 268,435,454 行。 如果你在使用 **/debug**，则使用超过 16,707,566 次将导致调试信息损坏。

> **备注**
>
> Roslyn 中不再使用此编译器错误。

编译器错误 CS1034

项目 • 2023/04/07

超出编译器限制: 行不能超过“number”个字符

一行中所允许的字符数限制为 16,777,214 个。

编译器错误 CS1035

项目 • 2024/04/11

发现文件尾，应输入 "*/"

注释起始分隔符与结尾分隔符不匹配。

下面的示例生成 CS1035：

C#

```
// CS1035.cs
public class a
{
   public static void Main()
   {
   }
}
/*   // CS1035, needs closing comment
```


在 GitHub 上与我们协作

可以在 GitHub 上找到此内容的源，还可以在其中创建和查看问题和拉取请求。 有关详细信息，请参阅参与者指南。

.NET

.NET 反馈

.NET 是一个开放源代码项目。 选择一个链接以提供反馈：

提出文档问题

提供产品反馈

编译器错误 CS1036

项目 • 2024/04/11

(或 . 应有

DocumentationFile 注释中的 XML 格式错误。

在 GitHub 上与我们协作

可以在 GitHub 上找到此内容的源，还可以在其中创建和查看问题和拉取请求。 有关详细信息，请参阅参与者指南。

.NET

.NET 反馈

.NET 是一个开放源代码项目。 选择一个链接以提供反馈：

提出文档问题

提供产品反馈

编译器错误 CS1037

项目 • 2024/04/11

应输入可重载运算符

使用 DocumentationFile 指定注释时，编译器遇到了无效链接。

在 GitHub 上与我们协作

可以在 GitHub 上找到此内容的源，还可以在其中创建和查看问题和拉取请求。 有关详细信息，请参阅参与者指南。

.NET

.NET 反馈

.NET 是一个开放源代码项目。 选择一个链接以提供反馈：

提出文档问题

提供产品反馈

编译器错误 CS1038

项目 • 2023/04/07

应输入 #endregion 指令

#region 指令没有匹配的 #endregion 指令。

下面的示例生成 CS1038:

C#

```
// CS1038.cs
#region testing

public class clx
{
   public static void Main()
   {
   }
}
// CS1038
// uncomment the next line to resolve
// #endregion
```

编译器错误 CS1039

项目 • 2023/04/07

字符串未终止

编译器检测到格式不正确的 字符串 文本。

示例

下面的示例生成 CS1039。 若要解决此错误，请添加终止引号。

C#

```
// CS1039.cs
public class MyClass
{
    public static void Main()
    {
        string b = @"hello, world;   // CS1039
    }
}
```

编译器错误 CS1040

项目 • 2023/04/07

预处理器指令必须作为一行的第一个非空白字符出现

在行中发现 预处理器指令 ， 而它不是该行中第一个标记。 指令必须是行中第一个标记。

下面的示例生成 CS1040：

C#

```
// CS1040.cs
/* Define a symbol, X */ #define X   // CS1040

// try the following two lines instead
// /* Define a symbol, X */
// #define X

public class MyClass
{
   public static void Main()
   {
   }
}
```

编译器错误 CS1041

项目 • 2023/04/07

应为标识符, “keyword” 是关键字

在应有标识符的位置发现 C# 语言的保留字。 用用户指定的标识符替换 关键字 。

示例 1

以下示例生成 CS1041：

C#

```
// CS1041a.cs
class MyClass
{
    public void f(int long)   // CS1041
    // Try the following instead:
    // public void f(int i)
    {
    }

    public static void Main()
    {
    }
}
```

示例 2

在从另一种不具有同一组保留字的编程语言导入时，你可以用 @ 前缀修改保留的标识符，如下面的示例所示。

具有 `@` 前缀的标识符称为逐字标识符。

C#

```
// CS1041b.cs
class MyClass
{
    public void f(int long)   // CS1041
    // Try the following instead:
    // public void f(int @long)
    {
    }

    public static void Main()
```

```
    {
    }
}
```

编译器错误 CS1043

项目 • 2023/04/07

应为 { 或 ;

未正确声明属性访问器。 有关详细信息，请参阅使用属性。

示例

下面的示例生成 CS1043。

C#

```
// CS1043.cs
// compile with: /target:library
public class MyClass
{
   public int DoSomething
   {
      get return 1;   // CS1043
      set {}
   }

   // OK
   public int DoSomething2
   {
      get { return 1;}
   }
}
```

编译器错误 CS1044

项目 • 2023/04/07

在 for、using、fixed 或声明语句中不能使用多个类型

编译器发现了一个无效的语句。

以下示例生成 CS1044：

C#

```
// CS1044.cs
using System;

public class MyClass : IDisposable
{
   public void Dispose()
   {
      Console.WriteLine("Res1.Dispose()");
   }

   public static void Main()
   {
      using (MyClass mc1 = new MyClass(),
             MyClass mc2 = new MyClass())   // CS1044, remove an
instantiation
      {
      }
   }
}
```

编译器错误 CS1055

项目 • 2023/04/07

应为 add 访问器或 remove 访问器

如果 事件 未声明为字段，则它必须同时定义 **add** 和 **remove** 访问器函数。

下面的示例生成 CS1055：

C#

```
// CS1055.cs
delegate void del();
class Test
{
   public event del MyEvent
   {
      int i;   // CS1055
      // uncomment accessors and delete previous line to resolve
      // add
      // {
      //    MyEvent += value;
      // }
      // remove
      // {
      //    MyEvent -= value;
      // }
   }

   public static void Main()
   {
   }
}
```

编译器错误 CS1056

项目 • 2023/04/07

意外的字符“character”

C# 编译器遇到了意外的字符，并且不能识别当前正在处理的标记。 例如，如果编译器在处理标识符过程中遇到欧洲字符，它将无法将标识符分类，因为欧洲字符仅在字符串内有效，且编译器知道它没在处理字符串。

编译器错误 CS1057

项目 • 2023/04/07

“member”：静态类不能包含受保护的成员

在静态类内部声明受保护成员时，会生成此错误。

示例

下面的示例生成 CS1057。

C#

```
// CS1057.cs
using System;

static class Class1
{
    protected static int x;   // CS1057
    public static void Main()
    {
    }
}
```

编译器错误 CS1059

项目 • 2023/04/07

递增运算符或递减运算符的操作数必须是变量、属性或索引器。

在尝试递增或递减常量值时，会引发此错误。 如果尝试递增表达式（如 `(a+b)++`），也会发生此错误。

更正此错误

- 使变量成为非常量变量。
- 删除递增或递减运算符。
- 将表达式存储在一个变量中，然后递增该变量。

示例

下面的示例生成 CS1059，因为 `i` 是常量而非变量，并且 `E` 为 `Enum` 类型，该类型的元素也是常量值。

C#

```
// CS1059.cs
    class Program
    {
        public enum E : sbyte
        {
            a = 1,
            b = 2
        }

        static void Main(string[] args)
        {
            const int i = 0;
            i++;            // CS1059
            E test = E.a++; // CS1059
        }
    }
```

另请参阅

- 常量

编译器错误 CS1061

项目 • 2023/04/07

“type”未包含“name”的定义，并且找不到可接受第一个“type”类型参数的可访问扩展方法“name”（是否缺少 using 指令或程序集引用？）。

尝试调用方法或访问不存在的类成员时，将出现此错误。

示例

下面的示例将生成 CS1061，因为 `Person` 没有 `DisplayName` 方法。 但它具有称作 `WriteName` 的方法。 或许这就是作者编写此源代码的用意。

C#

```
public class Person
{
    private string _name;

    public Person(string name) => _name = name;

    // Person has one method, called WriteName.
    public void WriteName()
    {
        System.Console.WriteLine(_name);
    }
}

public class Program
{
    public static void Main()
    {
        var p = new Person("PersonName");

        // The following call fails because Person does not have
        // a method called DisplayName.
        p.DisplayName(); // CS1061
    }
}
```

更正此错误

1. 请确保正确输入了成员名。
2. 如果有权修改此类，则可以添加缺少的成员并实现它。
3. 如果没有修改此类的权限，则可以添加扩展方法。

另请参阅

- C# 类型系统
- 扩展方法

编译器错误 CS1065

项目 • 2023/04/07

默认值在此上下文中无效。

声明可选参数时使用默认值。 使用委托运算符声明匿名方法时，不支持可选参数。 委托运算符创建匿名方法。

示例

下面的示例生成 CS1065：

C#

```
// CS1065.cs (5,27)

class A
{
    delegate void D(int x);
    D d1 = delegate(int x = 42) { };
}
```

更正此错误

删除默认值可更正此错误：

C#

```
class A
{
    delegate void D(int x);
    D d1 = delegate(int x) { };
}
```


Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

issues and pull requests. For more information, see our contributor guide.

Open a documentation issue

Provide product feedback

编译器错误 CS1100

项目 • 2023/04/07

方法“name”具有一个不在第一个参数上的“this”参数修饰符。

`this` 修饰符仅允许用于方法的第一个参数，这将向编译器显示该方法是一个扩展方法。

更正此错误

1. 除了此方法的第一个参数外，从其他所有参数删除 `this` 修饰符。

示例

下面的代码生成 CS1100，因为 `this` 参数在修改第二个参数:

C#

```
// cs1100.cs
static class Test
{
    static void ExtMethod(int i, this Test c) // CS1100
    {
    }
}
```

请参阅

- 扩展方法

编译器错误 CS1101

项目 • 2023/10/06

参数修饰符“ref”不能与“this”一起使用。

当 `this` 关键字修改静态方法的第一个参数时，它向编译器发出信号，指示该方法是扩展方法。 对于 C# 版本 7.1 和更低版本，不需要或不允许对扩展方法的第一个参数使用任何其他修饰符。 从 C# 版本 7.2 起允许 `ref` 扩展方法，有关更多详细信息，请查看扩展方法。

示例

下面的示例生成 CS1101：

C#

```
// cs1101.cs
// Compile with: /target:library
public static class Extensions
{
    public static void Test(ref this int i) {} // CS1101
}
```

另请参阅

- this
- ref

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

Open a documentation issue

Provide product feedback

编译器错误 CS1102

项目 • 2023/04/07

参数修饰符“out”不能与“this”一起使用。

当 `this` 关键字修改静态方法的第一个参数时，它向编译器发出信号，指示该方法是扩展方法。 不需要或不允许对扩展方法的第一个参数使用任何其他修饰符。

更正此错误

1. 从第一个参数中删除未经授权的修饰符。

示例

下面的示例生成 CS1102：

C#

```
// cs1102.cs
// Compile with: /target:library.
public static class Extensions
{
    // No type parameters.
        public static void Test(this out int i) {} // CS1102

    //Single type parameter
        public static void Test<T>(this out T t) {}// CS1102

    //Multiple type parameters
        public static void Test<T,U,V>(this out U u) {}// CS1102
}
```

请参阅

- 扩展方法
- this
- out

编译器错误 CS1103

项目 • 2023/04/07

扩展方法的第一个参数的类型不能是“type”。

扩展方法的第一个参数的类型不能是指针类型。

示例

下面的示例生成 CS1103：

C#

```
// cs1103.cs
public static class Extensions
{
    public unsafe static char* Test(this char* charP) { return charP; } //
CS1103
}
```

请参阅

- 扩展方法
- unsafe

编译器错误 CS1105

项目 • 2023/04/07

扩展方法必须是静态的。

扩展方法必须在非泛型静态类中声明为静态方法。

示例

下面的示例由于 `Test` 不是静态的而生成错误 CS1105。

C#

```
// cs1105.cs
// Compile with: /target:library
public class Extensions
{

    // Single type parameter.
        public void Test<T>(this System.String s) {} //CS1105

}
```

请参阅

- 扩展方法
- static

编译器错误 CS1106

项目 • 2023/10/07

扩展方法必须在非泛型静态类中定义。

扩展方法必须在非泛型静态类中定义为静态方法。

示例

下面的示例生成 CS1106：

C#

```
// CS1106.cs
public class NonStaticClass // CS1106
{
    public static void ExtensionMethod1(this int num) {}
}

public static class StaticGenericClass<T> // CS1106
{
    public static void ExtensionMethod2(this int num) {}
}

public static class StaticClass // OK
{
    public static void ExtensionMethod3(this int num) {}
}
```

另请参阅

- 扩展方法
- 泛型类
- static

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

Open a documentation issue

more information, see our contributor guide.

Provide product feedback

编译器错误 CS1107

项目 • 2023/04/07

参数只能具有一个“modifier name”修饰符。

`this`、`ref`、`in` 和 `out` 等参数修饰符在一个参数定义中出现多次时就会出现此错误。

示例

下面的示例生成 CS1107：

C#

```
// cs1107.cs
public static class Test
{
    // Extension methods.
    public static void TestMethod(this this int t) { } // CS1107

}

public class TestTwo
{
    // Regular Instance Method
    public void TestMethod(ref ref int i) { } // CS1107

    // Regular Instance Method
    public void TestMethod(in in double d) { } // CS1107
}
```

编译器错误 CS1108

项目 • 2023/04/07

参数不能具有所有指定的修饰符；该参数上的修饰符太多。

某些参数修饰符组合（例如 `in`、`ref` 和 `out`）是不允许使用的，因为在编译器中它们的含义互斥。

示例

下面的示例生成 CS1108：

C#

```
// cs1108.cs
// Compile with: /target:library
public class Test
{
    // Regular Instance Method.
        public void TestMethod(ref out int i) {} // CS1108

}
```

编译器错误 CS1109

项目 • 2023/04/07

扩展方法必须在顶级静态类中定义，“name”是嵌套类。

不能在嵌套类中定义扩展方法。

示例

下面的示例生成 CS1109，因为类 `Extension` 嵌套在类 `Out` 中：

C#

```
// cs1109.cs
public class Test
{
}
static class Out
{
    static class Extension
    {
        static void ExtMethod(this Test c) // CS1109
        {
        }
    }
}
```

请参阅

- 扩展方法

编译器错误 CS1110

项目 • 2023/05/09

不能在未引用 System.Core.dll 的情况下使用方法声明的第一个参数上的“this”修饰符。 添加对 System.Core.dll 的引用，或从该方法声明中删除“this”修饰符。

.NET Framework 3.5 及更高版本支持扩展方法。 扩展方法生成元数据，该元数据使用特性标记方法。 特性类位于 system.core.dll 中。

更正此错误

1. 如该消息所述，添加对 System.Core.dll 的引用，或从该方法声明中删除 `this` 修饰符。

示例

如果未使用对 System.Core.dll 的引用编译该文件，则以下示例生成 CS1110：

C#

```
// cs1110.cs
// CS1110
// Compile with: /target:library
public static class Extensions
{
    public static bool Test(this bool b) { return b; }
}
```

请参阅

- 扩展方法

编译器错误 CS1112

项目 • 2023/04/07

请勿使用“System.Runtime.CompilerServices.ExtensionAttribute”。 改用“this”关键字。

ExtensionAttribute 用于包含扩展方法的非静态类时，会生成此错误。 如果此属性用于静态类，则可能出现另一个错误，例如 CS0708：“无法声明静态类中的实例成员”。

在 C# 中，必须在静态类中定义扩展方法，并使用 `this` 关键字修改该方法的第一个参数。 在源代码中完全不使用该属性。 有关详细信息，请参阅扩展方法

更正此错误

1. 删除此属性，并将 `this` 修饰符应用到该方法的第一个参数。

示例

下面的示例生成 CS1112：

C#

```
// cs1112.cs
[System.Runtime.CompilerServices.ExtensionAttribute] // CS1112
public class Extensions
{
    public bool A(bool b) { return b; }
}

class A { }
```

编译器错误 CS1113

项目 • 2023/04/07

无法使用值类型“name”上定义的扩展方法“name”来创建委托。

无法使用为类类型定义的扩展方法来创建委托。 无法使用为值类型定义的扩展方法。

更正此错误

1. 将扩展方法与类类型关联。

2. 使该方法成为结构上的常规方法。

示例

以下示例生成 CS1113：

C#

```
// cs1113.cs
using System;
public static class Extensions
{
    public static S ExtMethod(this S s)
    {
        return s;
    }
}

public struct S
{
}

public class Test
{
    static int Main()
    {
        Func<S> f = new S().ExtMethod; // CS1113
        return 1;
    }
}
```

请参阅

- 扩展方法

编译器错误 CS1502

项目 • 2023/04/07

与“name”最匹配的重载方法具有一些无效参数

正传递给方法的参数类型与该方法的参数类型不匹配时，会出现此错误。 如果重载调用的方法，则这些重载版本都不会具有正传递的参数类型相匹配的签名。

若要解决此问题，请执行下列操作之一：

- 仔细检查正在进行传递的参数的类型。 确保它们与正在调用的方法的参数匹配。
- 在适当的情况下，使用 Convert 类转换任何不匹配的参数。
- 在适当的情况下，强制转换任何不匹配的参数，使它们与该方法需要的类型相匹配。
- 在适当的情况下，定义方法的其他重载版本，使其与正在发送的参数类型相匹配。

下面的示例生成 CS1502：

C#

```
// CS1502.cs
namespace x
{
   public class a
   {
      public a(char i)
      // try the following constructor instead
      // public a(int i)
      {
      }

      public static void Main()
      {
         a aa = new a(2222);   // CS1502
      }
   }
}
```

编译器错误 CS1503

项目 • 2023/04/07

参数“number”无法从“TypeA”转换为“TypeB”。

方法中一个参数的类型与实例化类时传递的类型不匹配。 此错误通常与 CS1502 一同出现。 有关如何解决此错误的讨论，请参见 CS1502 。

编译器错误 CS1504

项目 • 2023/04/07

无法打开源文件“file”（“reason”）

无法打开或使用编译器读取源文件。 该文件可能被另一个应用程序锁定，或者可能有一些其他的操作系统问题。 该消息包含编译器无法打开或读取该文件的操作系统的原因。

编译器错误 CS1507

项目 • 2023/04/07

生成模块时，无法链接资源文件“file”

在同一编译中同时使用了 LinkResources **和 TargetType 上的 module 选项 - 不允许这样做。** 例如，下面的选项将生成 CS1507：

控制台

```
csc /linkresource:rf.resource /target:module in.cs
```

不过，允许嵌入资源 (Resources)。

编译器错误 CS1508

项目 • 2023/04/07

此程序集中已使用了资源标识符“identifier”

在编译时，将相同标识符 (identifier) 传递给了两个或更多 Resources 或 LinkResources 编译器选项。

例如，下面的选项生成 CS1508：

控制台

```
/resource:anyfile.bmp,DuplicatIdent /linkresource:a.bmp,DuplicatIdent
```

编译器错误 CS1509

项目 • 2023/04/07

引用的文件“file”不是程序集；请改用“AddModules”选项

在使用 **TargetType** 的 module 元素的编译中（没有程序集清单）生成的输出文件（输出文件 1）被指定为了 References。 因此，不是程序集追加到当前程序的程序集，而是输出文件 1 中的元数据信息将添加到当前程序的程序集。

编译器错误 CS1511

项目 • 2023/04/07

关键字“base”在静态方法中不可用

base 关键字用在 静态 方法中。 只能在实例构造函数、实例方法或实例访问器中调用 `base` 。

示例

以下示例生成 CS1511。

C#

```
// CS1511.cs
// compile with: /target:library
public class A
{
   public int j = 0;
}

class C : A
{
   public void Method()
   {
      base.j = 3;   // base allowed here
   }

   public static int StaticMethod()
   {
      base.j = 3;   // CS1511
      return 1;
   }
}
```

编译器错误 CS1512

项目 • 2023/04/07

关键字“base”在当前上下文中不可用

在方法、属性或构造函数外部使用了 base 关键字。

下面的示例生成 CS1512：

C#

```
// CS1512.cs
using System;

class Base {}

class CMyClass : Base
{
    private String xx = base.ToString();   // CS1512
    // Try putting this initialization in the constructor instead:
    // public CMyClass()
    // {
    //    xx = base.ToString();
    // }

    public static void Main()
    {
        CMyClass z = new CMyClass();
    }
}
```

编译器错误 CS1513

项目 • 2024/04/11

应输入 }

编译器需要未找到的右大括号 (`}`)。

下面的示例生成 CS1513：

```
C#
```

```
// CS1513
namespace y   // CS1513, no close curly brace
{
   class x
   {
      public static void Main()
      {
      }
   }
```


在 GitHub 上与我们协作

可以在 GitHub 上找到此内容的源，还可以在其中创建和查看问题和拉取请求。 有关详细信息，请参阅参与者指南。

.NET

.NET 反馈

.NET 是一个开放源代码项目。 选择一个链接以提供反馈：

提出文档问题

提供产品反馈

编译器错误 CS1514

项目 • 2024/04/11

应为 {

找不到编译器需要的左大括号 (`{`)。

下面的示例生成 CS1514：

C#

```
// CS1514.cs
namespace x
// CS1514, no open curly brace
```


在 GitHub 上与我们协作

可以在 GitHub 上找到此内容的源，还可以在其中创建和查看问题和拉取请求。 有关详细信息，请参阅参与者指南。

.NET 反馈

.NET 是一个开放源代码项目。 选择一个链接以提供反馈：

提出文档问题

提供产品反馈

编译器错误 CS1515

项目 • 2023/04/07

应为 "in"

在 foreach, in 语句中，缺少"in"部分。

示例

以下示例生成 CS1515：

C#

```
using System;

class Driver
{
   static void Main()
   {
      int[] arr = new int[] {1, 2, 3};

      // try the following line instead
      // foreach (int x in arr)
      foreach (int x arr)      // CS1515, "in" is missing
      {
         Console.WriteLine(x);
      }
   }
}
```

编译器错误 CS1517

项目 • 2023/04/07

无效的预处理器表达式

编译器遇到了无效的预处理器表达式。

有关详细信息，请参阅 预处理器指令。

下面的示例显示了一些有效和无效的预处理器表达式：

C#

```
// CS1517.cs
#if symbol      // OK
#endif
#if !symbol     // OK
#endif
#if (symbol)    // OK
#endif
#if true        // OK
#endif
#if false       // OK
#endif
#if 1           // CS1517
#endif
#if ~symbol     // CS1517
#endif
#if *           // CS1517
#endif

class x
{
   public static void Main()
   {
   }
}
```

编译器错误 CS1518

项目 • 2023/04/07

应输入 class、delegate、enum、interface 或 struct

找到在 命名空间中不受支持的声明。 在命名空间中，编译器仅接受类、结构、枚举、接口、命名空间和委托。

示例

下面的示例生成 CS1518：

C#

```
// CS1518.cs
namespace x
{
   sealed class c1 {};      // OK
   namespace f2 {};         // OK
   sealed f3 {};            // CS1518
}
```

编译器错误 CS1519

项目 • 2023/04/07

类、结构或接口成员声明中的令牌“token”无效

令牌的放置位置无效时会生成此错误。 令牌是关键字；标识符（类、结构、方法等的名称）；字符串、字符或数字文本值（例如 108、“Hello”或 A）；运算符或标点符号（例如 `==` 或 `;`）。

在任何类、结构或接口成员声明中，只要在类型前面包含无效修饰符都会生成此错误。 若要修复此错误，请删除无效的修饰符。

下例在五个位置生成 CS1519，因为这些令牌的放置位置无效：

C#

```
// CS1519.cs
// Generates CS1519 because a class name cannot be a number:
class Test 42
{
// Generates CS1519 because of 'j' following 'I'
// with no comma between them:
    int i j;
// Generates CS1519 because of "checked" on void method:
    checked void f4();

// Generates CS1519 because of "num":
    void f5(int a num){}

// Generates CS1519 because of namespace inside class:
    namespace;

}
```

请参阅

- 类
- 结构类型
- 接口
- 方法

编译器错误 CS1520

项目 • 2023/04/07

方法必须具有返回类型

在类、结构或接口中声明的方法必须具有显式返回类型。 在下面的示例中，`IntToString` 方法具有字符串返回值：

C#

```
class Test
{
    string IntToString(int i)
    {
        return i.ToString();
    }
}
```

以下示例生成 CS1520：

C#

```
public class x
{
   // Method declaration missing a return type before the name of MyMethod
   // Note: the method is empty for the purposes of this example so as to
not add confusion.
   MyMethod() { }
}
```

可以通过向方法添加返回类型进行修复，例如在以下示例中添加 `void`：

C#

```
public class x
{
   // MyMethod no longer throws an error, because it has a return type --
"void" in this case.
   void MyMethod() { }
}
```

或者，当构造函数名称的大小写与类或结构声明不同时，则可能遇到此错误，如以下示例所示。 由于该名称与类名称不完全相同，因此编译器将其解释为常规方法而不是构造函数，从而产生此错误：

C#

```
public class Class1
{
   // Constructor should be called Class1, not class1
   public class1()   // CS1520
   {
   }
}
```

请参阅

- 方法
- 构造函数

编译器错误 CS1521

项目 • 2024/04/11

无效的基类型

基 类规范的格式错误。

下面的示例生成 CS1521：

C#

```
// CS1521.cs
class CMyClass
{
   public static void Main()
   {
   }
}

class CMyClass1 : CMyClass     // OK
{
}

class CMyClass2 : CMyClass[]   // CS1521
{
}

class CMyClass3 : CMyClass*    // CS1521
{
}
```


在 GitHub 上与我们协作

可以在 GitHub 上找到此内容的源，还可以在其中创建和查看问题和拉取请求。 有关详细信息，请参阅参与者指南。

.NET

.NET 反馈

.NET 是一个开放源代码项目。 选择一个链接以提供反馈：

提出文档问题

提供产品反馈

编译器错误 CS1524

项目 • 2023/04/07

应输入 catch 或 finally

`try` 块必须后跟 `catch` 或 `finally` 块。

有关异常的详细信息，请参阅异常和异常处理。

示例

下面的示例生成 CS1524：

C#

```
// CS1524.cs
class x
{
    public static void Main()
    {
        try
        {
            // Code here
        }
        catch
        {
        }
        try
        {
            // Code here
        }
        finally
        {
        }
        try
        {
            // Code here
        }
    }     // CS1524, missing catch or finally
}
```

编译器错误 CS1525

项目 • 2024/03/12

无效的表达式项“term”

编译器在表达式中检测到无效项。 此错误可能是由于缺少预期表达式，导致后续标记错误地被分析为表达式，或者表达式中使用无效构造。 常见根本原因包括不匹配的标记、缺少分号或多余的分隔符。

下面的示例生成 CS1525：

C#

```
// CS1525.cs
class MyClass
{
    public static void Method(int number) {}

    public static void Main()
    {
        int i = 0;
        i = i + 'c' + 1 + (2);   // OK
        i = i + void + throw;    // CS1525, these keywords are not valid in
this expression

        Method(123,);            // CS1525, excess trailing comma

        goto EmptyLabel;
        EmptyLabel:              // CS1525, empty label
        // Add something here to resolve the error, for example:
        // System.Console.WriteLine("Hello!");
    }
}
```


在 GitHub 上与我们协作

可以在 GitHub 上找到此内容的源，还可以在其中创建和查看问题和拉取请求。 有关详细信息，请参阅参与者指南。

.NET

.NET 反馈

.NET 是一个开放源代码项目。 选择一个链接以提供反馈：

提出文档问题

提供产品反馈

编译器错误 CS1526

项目 • 2023/04/07

new 表达式要求在类型后有 ()、[] 或 {}

未正确指定用于为对象动态分配内存的 new 运算符。

示例

下面的示例演示如何使用 `new` 为数组和对象分配空间。

C#

```
// CS1526.cs
public class y
{
   public static int i = 0;
   public int myi = 0;
}

public class z
{
   public static void Main()
   {
      y py = new y;   // CS1526
      y[] aoys = new y[10];   // Array of Ys

      for (int i = 0; i < aoys.Length; i++)
         aoys[i] = new y();   // an object of type y
   }
}
```

编译器错误 CS1527

项目 • 2023/04/07

命名空间中定义的元素无法显式声明为 private、protected、protected internal 或 private protected。

命名空间中的类型声明可以具有 公共 或 内部 访问。 如果未指定任何可访问性，则默认设置为 **内部** 。

以下示例生成 CS1527：

C#

```
// CS1527.cs
namespace Sample
{
   private class C1 {}             // CS1527
   protected class C2 {}           // CS1527
   protected internal class C3 {}  // CS1527
   private protected class C4 {}   // CS1527
}
```

以下示例将生成 CS1527，因为当程序代码中没有显式声明命名空间时，所有类型声明均隐式存在于全局命名空间内。

C#

```
//cs1527_2.cs
using System;

protected class C4 {}
private struct S1 {}
```

另请参阅

- 命名空间
- ::运算符
- 可访问性域
- 访问修饰符

编译器错误 CS1528

项目 • 2023/04/07

应输入 ";" 或 "="(无法在声明中指定构造函数参数)

类引用的构造就像正在创建类的对象一样。 例如，尝试将变量传递给构造函数。 使用 new 运算符创建类的对象。

下面的示例生成 CS1528：

C#

```
// CS1528.cs
using System;

public class B
{
   public B(int i)
   {
      _i = i;
   }

   public void PrintB()
   {
      Console.WriteLine(_i);
   }

   private int _i;
}

public class mine
{
   public static void Main()
   {
      B b(3);   // CS1528, reference is not an object
      // try one of the following
      // B b;
      // or
      // B bb = new B(3);
      // bb.PrintB();
   }
}
```

编译器错误 CS1530

项目 • 2023/04/07

不允许在命名空间中定义的元素上使用关键字“new”

无需在 namespace 中的任何构造上均指定 new关键字。

下面的示例生成 CS1530：

C#

```
// CS1530.cs
namespace a
{
   new class i   // CS1530
   {
   }

   // try the following instead
   class ii
   {
      public static void Main()
      {
      }
   }
}
```

编译器错误 CS1536

项目 • 2023/04/07

参数类型 void 无效

不需要指定除 void 指针以外的 void 参数或指定无效。

下面的示例生成 CS1536：

C#

```
// CS1536.cs
class a
{
   public static int x( void )   // CS1536
   // try the following line instead
   // public static int x()
   {
      return 0;
   }

   public static void Main()
   {
   }
}
```

编译器错误 CS1540

项目 • 2023/04/07

无法通过“type1”类型的限定符访问受保护的成员“member”；限定符必须是“type2”类型（或从该类型派生）

派生类无法通过基类的实例访问其基类的受保护成员。 在运行时，派生类中声明的基类的实例可能是另一类型的实例，而后者是由同一基类派生，但与派生类无关的类型。 由于受保护的成员仅可由派生类型访问，因此在运行时对受保护成员可能无效的任何访问尝试，都会被编译器标记为无效。

在以下示例中的 `Employee` 类中，在编译时 `emp2` 和 `emp3` 都具有类型 `Person`，但在运行时 `emp2` 具有类型 `Manager`。 由于 `Employee` 不是派生自 `Manager`，因此它不能通过 `Manager` 类的实例访问基类的受保护成员 `Person`。 编译器无法确定两个 `Person` 对象的运行时类型。 因此，来自 `emp2` 的调用和来自 `emp3` 的调用都会导致编译器错误 CS1540。

C#

```
using System;
using System.Text;

namespace CS1540
{
    class Program1
    {
        static void Main()
        {
            Employee.PreparePayroll();
        }
    }

    class Person
    {
        protected virtual void CalculatePay()
        {
            Console.WriteLine("CalculatePay in Person class.");
        }
    }

    class Manager : Person
    {
        protected override void CalculatePay()
        {
            Console.WriteLine("CalculatePay in Manager class.");

        }
    }
```

```
    class Employee : Person
    {
        public static void PreparePayroll()
        {
            Employee emp1 = new Employee();
            Person emp2 = new Manager();
            Person emp3 = new Employee();
            // The following line calls the method in the Employee base
class,
            // Person.
            emp1.CalculatePay();

            // The following lines cause compiler error CS1540. The compiler
            // cannot determine at compile time what the run-time types of
            // emp2 and emp3 will be.
            //emp2.CalculatePay();
            //emp3.CalculatePay();

        }
    }
}
```

另请参阅

- 继承
- 多态性
- 访问修饰符
- 抽象类、密封类及类成员

编译器错误 CS1541

项目 • 2023/04/07

无效的引用选项：“symbol”-- 无法引用目录

编译器检测到尝试指定一个目录而非特定文件的意图。 例如，当使用 References 编译器选项时，必须指定一个文件，而不能指定一个目录。

例如，将 `/reference:c:\` 传递给编译器会生成 CS1541。

编译器错误 CS1542

项目 • 2023/04/07

“dll”无法添加到此程序集中，因为它已经是程序集；请使用“/R”选项

使用 AddModules 编译器选项引用的文件不是使用 **TargetType** 编译器选项的 module 元素生成的；在此编译中，使用 References 来引用该文件。

编译器错误 CS1545

项目 • 2023/04/07

属性、索引器或事件“property”不受现用语言支持；请尝试直接调用访问器方法“set accessor”或“get accessor”

该代码正在使用具有非默认 索引器 的对象并试图使用索引的语法。 若要解决此错误，请调用该属性的 `get` 或 `set` 访问器方法。

示例 1

C++

```
// CPP1545.cpp
// compile with: /clr /LD
// a Visual C++ program
using namespace System;
public ref struct Employee {
   Employee( String^ s, int d ) {}

   property String^ name {
      String^ get() {
         return nullptr;
      }
   }
};

public ref struct Manager {
   property Employee^ Report [String^] {
      Employee^ get(String^ s) {
         return nullptr;
      }

      void set(String^ s, Employee^ e) {}
   }
};
```

示例 2

下面的示例生成 CS1545：

C#

```
// CS1545.cs
// compile with: /r:CPP1545.dll
```

```
class x {
   public static void Main() {
      Manager Ed = new Manager();
      Employee Bob = new Employee("Bob Smith", 12);
      Ed.Report[ Bob.name ] = Bob;   // CS1545
      Ed.set_Report( Bob.name, Bob);   // OK
   }
}
```

编译器错误 CS1546

项目 • 2023/04/07

属性、索引器或事件“property”不受现用语言支持；请尝试直接调用访问器方法“accessor”

代码正在使用具有默认索引属性的对象并试图使用索引语法。 若要解决此错误，请调用该属性的访问器方法。 有关索引器和属性的详细信息，请参阅索引器。

下面的示例生成 CS1546。

示例 1

此代码示例由一个编译为 .dll 的 .cpp 文件和一个使用该 .dll 的 .cs 文件组成。 下面的代码用于 .dll 文件，并定义要由 .cs 文件中的代码访问的属性。

C++

```
// CPP1546.cpp
// compile with: /clr /LD
using namespace System;
public ref class MCPP
{
public:
    property int Prop [int,int]
    {
        int get( int i, int b )
        {
            return i;
        }
    }
};
```

示例 2

这是 C# 文件。

C#

```
// CS1546.cs
// compile with: /r:CPP1546.dll
using System;
public class Test
{
    public static void Main()
```

```
    {
        int i = 0;
        MCPP mcpp = new MCPP();
        i = mcpp.Prop(1,1); // CS1546
        // Try the following line instead:
        // i = mcpp.get_Prop(1,1);
    }
}
```

编译器错误 CS1547

项目 • 2023/04/07

关键字 "void" 不能在此上下文中使用

编译器检测到无效的 void 关键字使用。

下面的示例生成 CS1547：

C#

```
// CS1547.cs
public class MyClass
{
   void BadMethod()
   {
      void i;   // CS1547, cannot have variables of type void
   }

   public static void Main()
   {
   }
}
```

编译器错误 CS1548

项目 • 2023/04/07

对程序集“assembly”签名时加密失败 —“原因”

程序集签名失败时，发生 CS1548。 这通常是由于无效的密钥文件名称、无效的密钥文件路径或已损坏的密钥文件造成的。

若要对程序集进行完全签名，必须提供包含公钥和私钥信息的有效密钥文件。 若要延迟对程序集的签名，必须选择“仅延迟签名”复选框，并提供包含公钥信息的有效密钥文件。 当程序集为延迟签名时，不需要使用私有密钥。

有关详细信息，请参阅如何：使用强名称对程序集进行签名、KeyFile（C# 编译器选项）和 DelaySign（C# 编译器选项）。

创建程序集时，C# 编译器调入名为 al.exe 的实用工具。 如果程序集创建失败，则由 al.exe 报告失败原因。 请参阅 Al.exe 工具错误和警告，并在该主题中搜索编译器在“原因”中报告的文本。

另请参阅

- 如何：使用强名称为程序集签名

编译器错误 CS1551

项目 • 2024/04/11

索引器必须至少有一个参数

声明的 索引器 不带参数。

下面的示例生成 CS1551：

C#

```
// CS1551.cs
public class MyClass
{
   int intI;

   int this[]   // CS1551
   // try the following line instead
   // int this[int i]
   {
      get
      {
         return intI;
      }
      set
      {
         intI = value;
      }
   }

   public static void Main()
   {
   }
}
```


在 GitHub 上与我们协作

可以在 GitHub 上找到此内容的源，还可以在其中创建和查看问题和拉取请求。 有关详细信息，请参阅参与者指南。

.NET

.NET 反馈

.NET 是一个开放源代码项目。 选择一个链接以提供反馈：

提出文档问题

提供产品反馈

编译器错误 CS1553

项目 • 2023/04/07

声明无效；请改用“修饰符运算符 <dest-type> (...”

转换运算符的返回类型必须紧跟参数列表，并且 *修饰符* 为 `implicit` 或 `explicit`。

下面的示例生成 CS1553：

C#

```
// CS1553.cs
class MyClass
{
   public static int implicit operator (MyClass f)   // CS1553
   // try the following line instead
   // public static implicit operator int (MyClass f)
   {
      return 6;
   }

   public static void Main()
   {
   }
}
```

编译器错误 CS1554

项目 • 2023/04/07

声明无效；请改用“<type> operator op (...”

重载运算符的返回类型必须在 `operator` 关键字的前面。

下面的示例生成 CS1554：

C#

```
// CS1554.cs
class MyClass
{
   public static operator ++ MyClass (MyClass f)    // CS1554
   // try the following line instead
   // public static MyClass operator ++ (MyClass f)
   {
      return new MyClass ();
   }

   public static void Main()
   {
   }
}
```

编译器错误 CS1555

项目 • 2023/04/07

无法找到为“Main”方法指定的“class”

已向 MainEntryPoint 编译器选项指定了一个类，但无法在源代码中找到该类的名称。

编译器错误 CS1556

项目 • 2023/04/07

为 Main 方法指定的“construct”必须是有效的类或结构

MainEntryPoint 编译器选项被传递了不是类名的标识符。

编译器错误 CS1557

项目 • 2023/04/07

无法对 Main 方法使用“class”，因为它在不同的输出文件中

为一个多输出文件编译中的输出文件指定了 MainEntryPoint 编译器选项。 但是，在 /main 编译的源代码中未找到该类；在编译内的一个其他输出文件的源代码文件中找到了该类。

编译器错误 CS1558

项目 • 2023/04/07

“class”没有合适的静态 Main 方法

MainEntryPoint 编译器选项指定了一个在其中查找 Main 方法的类。 但是， Main 方法未正确定义。

以下示例由于无效的返回类型而生成 CS1558。

C#

```
// CS1558.cs
// compile with: /main:MyNamespace.MyClass

namespace MyNamespace
{
   public class MyClass
   {
      public static float Main()
      {
         return 0.0; // CS1558 because the return type is a float.
      }
   }
}
```

编译器错误 CS1559

项目 • 2023/04/07

无法对 Main 方法使用“object”，因为它是被导入的

为 StartupObject 编译器选项指定的类无效；该类不能用作 Main 方法的位置。

编译器错误 CS1560

项目 • 2023/04/07

为预处理器指令指定的文件名无效。 文件名太长或文件名无效

使用 #line 指定的文件名超过了 _MAX_PATH（256 个字符）或其上存在 `#line` 的行超过 2000 个字符。

示例

下面的示例生成 CS1560。

C#

```
// cs1560.cs
using System;
class MyClass
{
   public static void Main()
   {
      Console.WriteLine("Normal line #1.");
      #line 21
"MyFile1234567890MyFile1234567890MyFile1234567890MyFile1234567890MyFile12345
67890MyFile1234567890MyFile1234567890MyFile1234567890MyFile1234567890MyFile1
234567890MyFile1234567890MyFile1234567890MyFile1234567890MyFile1234567890MyF
ile1234567890MyFile1234567890.txt"   // CS1560
    }
}
```

编译器错误 CS1561

项目 • 2023/04/07

输出文件名太长或无效

输出文件名的长度不能超过 256 个字符且不能包含无效字符，如欧元符号、问号或反斜杠等等。

编译器错误 CS1562

项目 • 2023/04/07

必须为没有源的输出指定 /out 选项

编译可以创建输出文件，但没有源代码文件可以作为可从中默示输出文件名的输入。 例如，你可能尝试编译仅元数据文件或仅资源文件。

使用 OutputAssembly 编译器选项指定输出文件的名称。

编译器错误 CS1563

项目 • 2023/04/07

输出“输出文件”没有任何源文件

已指定 OutputAssembly 编译器选项，但后面没有源代码文件。 不得使用 OutputAssembly，或者应指定 OutputAssembly 后面的源代码文件。

编译器错误 CS1564

项目 • 2023/04/07

指定的选项冲突：Win32 资源文件；Win32 清单。

如果使用 /Win32res 编译器选项，资源文件中必须包含自定义 Win32 清单（如果需要）。 不能单独从 Win32 资源文件提供自定义 Win32 清单。 仅当未指定 win32 资源文件时，使用“/win32manifest”选项。

更正此错误

1. 将 win32 清单添加到 win32 资源文件并删除“/win32manifest”编译器选项。

示例

如果下面的示例使用“/Win32res”选项进行编译，但资源文件中不包含清单时，则会生成 CS1564。

C#

```
// cs1564.cs
// Compile with: /Win32res
public class Test
{
    static int Main(string[] args)
    {
        return 1;
    }
}
```

C# 3.0 编译器将默认 win32Manifest 添加到所有二进制文件。

另请参阅

- Win32Manifest（C# 编译器选项）
- Win32Resource（C# 编译器选项）

编译器错误 CS1565

项目 • 2023/04/07

指定的选项冲突: Win32 资源文件；Win32 图标

不可在同一编译中同时指定 Win32Resource 编译器选项和 Win32Icon 编译器选项。

编译器错误 CS1566

项目 • 2023/04/07

读取资源文件“file”时出错 —“reason”

编译器在处理传递给 Resources 编译器选项的文件名时遇到问题。

编译器错误 CS1567

项目 • 2023/04/07

生成 Win32 资源时出错：“file”

编译要么使用了 Win32Icon 编译器选项，要么未使用 Win32Resource，这导致编译器生成包含资源信息的文件，但由于磁盘空间不足或某些其他错误，编译器无法创建该文件。

如果无法解决文件生成问题，可以使用 Win32Resource，这样便不会生成包含资源信息的文件。

编译器错误 CS1569

项目 • 2023/04/07

生成 XML 文档文件“Filename”时出错（“reason”）

在尝试向消息中命名的文件写入 XML 文档时，由于指定的原因出错。 原因可能类似于“找不到网络驱动器”或“访问被拒绝”。通常，原因将指示需要执行哪些操作来更正此错误。 例如，如果该错误指出"访问被拒绝"，应验证你具有在此文件上的写入权限。

示例 1

C#

```
// 1569a.cs
// compile with: /doc:CS1569.xml
// post-build command: attrib +r CS1569.xml
class Test
{
   /// <summary>Test.</summary>
   public static void Main() {}
}
```

示例 2

上面的示例生成 .xml 文件，然后该文件设置为只读。 此示例尝试写入相同的文件。 以下示例生成 CS1569。

C#

```
// CS1569.cs
// compile with: /doc:CS1569.xml
// CS1569 expected
class Test
{
   /// <summary>Test.</summary>
   public static void Main() {}
}
```

编译器错误 CS1575

项目 • 2023/04/07

Stackalloc 表达式要求在类型后有 []

使用 stackalloc所请求的分配的大小必须在方括号中指定。

下面的示例生成 CS1575：

C#

```
// CS1575.cs
// compile with: /unsafe
public class MyClass
{
   unsafe public static void Main()
   {
      int *p = stackalloc int (30);   // CS1575
      // try the following line instead
      // int *p = stackalloc int [30];
   }
}
```

编译器错误 CS1576

项目 • 2023/04/07

为 #line 指令指定的行号缺少或无效

编译器检测到传递给 #line 指令的值有错误。

下面的示例生成 CS1576：

C#

```
// CS1576.cs
public class MyClass
{
   static void Main()
   {
      #line "abc.sc"         // CS1576
      // try the following line instead
      //#line  101 "abc.sc"
      intt i;  // error will be reported on line 101
   }
}
```

编译器错误 CS1577

项目 • 2023/04/07

程序集生成失败 - 原因

在 C# 编译器成功编译此源代码之后，此编译的程序集生成部分失败。 请参阅 ALink 实用程序的错误文档（Al.exe 工具错误和警告），了解详细信息。

另请参阅

- Al.exe（程序集链接器）

编译器错误 CS1578

项目 • 2023/04/07

应输入文件名、单行注释或行尾

#line 指令后面只能出现文件名（双引号括起）或单行注释。

以下示例生成 CS1578：

C#

```
// CS1578.cs
class MyClass
{
   static void Main()
   {
      #line 101 abc.cs   // CS1578
      // try the following line instead
      //#line 101 "abc.cs"
      intt i;          // error will be reported on line 101
   }
}
```

编译器错误 CS1579

项目 • 2023/04/07

foreach 语句不能对“type1”类型的变量进行操作，因为“type2”不包含“identifier”的公共定义

若要使用 foreach 语句循环访问某个集合，该集合必须满足以下要求：

- 它的类型必须包含一个公共无参数 `GetEnumerator` 方法，其返回类型为类、结构或接口类型。
- `GetEnumerator` 方法的返回类型必须包含名为 `Current` 的公共属性和名为 `MoveNext` 的公共无参数方法（其返回类型为 Boolean）。

示例

以下示例生成 CS1579，因为 `MyCollection` 类不包含公共 `GetEnumerator` 方法：

C#

```
// CS1579.cs
using System;
public class MyCollection
{
   int[] items;
   public MyCollection()
   {
      items = new int[5] {12, 44, 33, 2, 50};
   }

   // Delete the following line to resolve.
   MyEnumerator GetEnumerator()

   // Uncomment the following line to resolve:
   // public MyEnumerator GetEnumerator()
   {
      return new MyEnumerator(this);
   }

   // Declare the enumerator class:
   public class MyEnumerator
   {
      int nIndex;
      MyCollection collection;
      public MyEnumerator(MyCollection coll)
      {
         collection = coll;
         nIndex = -1;
      }
```

```
      public bool MoveNext()
      {
         nIndex++;
         return (nIndex < collection.items.Length);
      }

      public int Current => collection.items[nIndex];
   }

   public static void Main()
   {
      MyCollection col = new MyCollection();
      Console.WriteLine("Values in the collection are:");
      foreach (int i in col)   // CS1579
      {
         Console.WriteLine(i);
      }
   }
}
```

编译器错误 CS1583

项目 • 2023/04/07

“file”不是有效的 Win32 资源文件

当你使用 Win32Resource 编译器选项指定一个无效的或未按 Win32 资源文件正确格式设置的文件名时，发生该错误。 在 Visual Studio 中，文件名是在项目设计器的应用程序窗格中指定的。

另请参阅

- “项目设计器”->“应用程序”页 (C#)

编译器错误 CS1585

项目 • 2023/04/07

成员修饰符“keyword”必须位于成员类型和名称之前

方法签名中的访问说明符没有出现在正确的位置。

下面的示例生成 CS1585：

C#

```
// CS1585.cs
public class Class1
{
   public void static Main(string[] args)   // CS1585
   // try the following line instead
   // public static void Main(string[] args)
   {
   }
}
```

编译器错误 CS1588

项目 • 2023/05/10

无法确定公共语言运行时目录 --“reason”

编译器无法检索 .NET 运行时的安装目录。 此错误指示公共语言运行时未正确安装。

编译器错误 CS1593

项目 • 2023/04/07

委托“del”不接受“number”参数

传递给 委托 调用的参数数目与委托声明中的参数数目不一致。

下面的示例生成 CS1593：

C#

```
// CS1593.cs
using System;
delegate string func(int i);   // declare delegate

class a
{
   public static void Main()
   {
      func dt = new func(z);
      x(dt);
   }

   public static string z(int j)
   {
      Console.WriteLine(j);
      return j.ToString();
   }

   public static void x(func hello)
   {
      hello(8, 9);   // CS1593
      // try the following line instead
      // hello(8);
   }
}
```

编译器错误 CS1594

项目 • 2023/04/07

委托“delegate”有一些无效参数

传递给 委托 调用的实参类型与委托声明中的形参类型不符。

以下示例生成 CS1594：

C#

```
// CS1594.cs
using System;
delegate string func(int i);   // declare delegate

class a
{
   public static void Main()
   {
      func dt = new func(z);
      x(dt);
   }

   public static string z(int j)
   {
      Console.WriteLine(j);
      return j.ToString();
   }

   public static void x(func hello)
   {
      hello("8");   // CS1594
      // try the following line instead
      // hello(8);
   }
}
```

编译器错误 CS1597

项目 • 2024/03/01

方法或访问器块后面的分号无效

不需要（或不允许）使用分号结束方法或访问器块。

下面的示例生成 CS1597：

C#

```
// CS1597.cs
class MyClass
{
    public int Prop { get; };   // CS1597

    public static void Main() { };   // CS1597
}
```

若要解决此错误，请删除额外的尾随分号。

在 GitHub 上与我们协作

可以在 GitHub 上找到此内容的源，还可以在其中创建和查看问题和拉取请求。 有关详细信息，请参阅参与者指南。

.NET

.NET 反馈

.NET 是一个开放源代码项目。 选择一个链接以提供反馈：

提出文档问题

提供产品反馈

编译器错误 CS1599

项目 • 2023/05/09

方法或委托不能返回“type”类型

.NET 类库中的某些类型（例如 TypedReference、RuntimeArgumentHandle 和 ArgIterator）不能用作返回类型，因为它们可能被用于执行不安全的操作。

下面的示例生成 CS1599：

C#

```
// CS1599.cs
using System;

class MyClass
{
   public static void Main()
   {
   }

   public TypedReference Test1()   // CS1599
   {
      return null;
   }

   public ArgIterator Test2()   // CS1599
   {
      return null;
   }
}
```

编译器错误 CS1600

项目 • 2024/04/11

编译被用户取消

在使用 Visual Studio IDE 时，取消了使用 C# 编译器的编译。

在 GitHub 上与我们协作

可以在 GitHub 上找到此内容的源，还可以在其中创建和查看问题和拉取请求。 有关详细信息，请参阅参与者指南。

.NET

.NET 反馈

.NET 是一个开放源代码项目。 选择一个链接以提供反馈：

提出文档问题

提供产品反馈

编译器错误 CS1601

项目 • 2023/05/09

方法或委托参数不能是“type”类型

.NET Framework 类库中的一些类型，例如 TypedReference、RuntimeArgumentHandle 和 ArgIterator 不能用作 in、ref 或 out 参数，因为它们有可能会被用来执行不安全的操作。

下面的示例生成 CS1601：

C#

```
// CS1601.cs
using System;

class MyClass
{
    public void Test1(in TypedReference t)   // CS1601
    {
    }

    public void Test2(ref TypedReference t)   // CS1601
    {
    }

    public void Test3(out ArgIterator t)   // CS1601
    {
        t = default;
    }
}
```

编译器错误 CS1604

项目 • 2023/04/07

无法对“variable”进行赋值，因为它是只读的

为只读变量进行了赋值。 若要避免此错误，请勿在此上下文中对此变量进行赋值或修改。

编译器错误 CS1606

项目 • 2023/04/07

程序集签名失败；不能对输出签名 -- reason

生成了程序集，但当编译器试图完成对它的签名时出现失败。

编译器错误 CS1608

项目 • 2023/04/07

C# 类型上不允许有 Required 特性

在 C# 代码中定义的类型上不允许RequiredAttributeAttribute 。

编译器错误 CS1609

项目 • 2023/04/07

修饰符不能放置在事件访问器声明上

修饰符仅能放置在事件声明上，而不能放在事件访问器声明上。 有关详细信息，请参阅使用属性。

示例

以下示例生成 CS1609。

C#

```
// CS1609.cs
// compile with: /target:library
delegate int Del();
class A
{
   public event Del MyEvent
   {
      private add {}   // CS1609
      // try the following line instead
      // add {}
      remove {}
   }
}
```

编译器错误 CS1612

项目 • 2023/03/30

无法修改“expression”的返回值，因为它不是变量

尝试修改某一值类型，该值类型作为中间表达式的结果生成但不存储于变量中。 如果尝试直接修改泛型集合中的结构，则会发生此错误，如下例所示：

C#

```
List<MyStruct> list = {…};
list[0].Name = "MyStruct42"; //CS1612
```

若要修改结构，请首先将其赋给局部变量，修改该变量，然后将变量赋回集合中的项。

C#

```
List<MyStruct> list = {…};
MyStruct ms = list[0];
ms.Name = "MyStruct42";
list[0] = ms;
```

之所以会发生此错误，原因是在赋值时复制了值类型。 在从属性或索引器中检索值类型时，所获取的是对象的副本，而不是对于对象本身的引用。 返回的副本不由属性或索引器存储，因为它们实际上是方法，而不是存储位置（变量）。 必须将副本存储在声明的变量中，然后才能对其进行修改。

引用类型不会发生此错误，因为在这种情况下，属性或索引器将返回对现有对象的引用，该对象是存储位置。

如果是在定义类或结构，则修改属性声明，提供对结构成员的访问，即可解决此错误。 如果是在编写客户端代码，则创建你自己的结构实例，修改其字段，然后将整个结构赋回给该属性，即可解决此错误。 第三种替代方式是，可以将结构更改为类。

示例

尝试通过将返回整个结构的封闭类中的属性访问结构成员时，也会出现 CS1612，如下面的示例中所示：

C#

```
// CS1612.cs
using System;
```

```
public struct MyStruct
{
    public int Width;
}

public class ListView
{
    public MyStruct Size { get; set; }
}

public class MyClass
{
    public MyClass()
    {
        ListView lvi;
        lvi = new ListView();
        lvi.Size.Width = 5; // CS1612

        // You can use the following lines instead.
        // MyStruct ms;
        // ms.Width = 5;
        // lvi.Size = ms;
    }

    public static void Main()
    {
        MyClass mc = new MyClass();
        // Keep the console open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
```

请参阅

- 结构类型
- 值类型
- 引用类型

编译器错误 CS1613

项目 • 2023/04/07

找不到用于接口 “interface” coclass 包装类 “class” （是否缺少程序集引用？）

已尝试实例化来自接口的 COM 对象。 此接口具有 **ComImport** 和 `CoClass` 特性，但编译器找不到为 `CoClass` 特性给定的类型。

要解决此错误，你可以尝试以下项之一：

- 向具有 coclass 的程序集添加一个引用（大多数情况下，接口和 coclass 应位于同一程序集中） 有关信息，请参阅 References 或“添加引用”对话框。
- 修复接口上的 `CoClass` 特性。

以下示例演示如何正确使用 **CoClassAttribute**：

C#

```
// CS1613.cs
using System;
using System.Runtime.InteropServices;

[Guid("1FFD7840-E82D-4268-875C-80A160C23296")]
[ComImport()]
[CoClass(typeof(A))]
public interface IA{}
public class A : IA {}

public class AA
{
   public static void Main()
   {
      IA i;
      i = new IA(); // This is equivalent to new A().
                    // because of the CoClass attribute on IA
   }
}
```

编译器错误 CS1614

项目 • 2023/04/07

“name”的含义不明确，有可能表示“name”，也有可能表示“nameAttribute”；使用“@name”或“nameAttribute”。

编译器遇到了不明确的属性指定。

为方便起见，C# 编译器允许将“ExampleAttribute”指定为 `[Example]`。 但是，如果名为 `Example` 的属性类与“ExampleAttribute”同时存在，则会引起歧义，因为编译器无法确定 `[Example]` 是指 `Example` 属性还是“ExampleAttribute”属性。 为阐释清楚，将 `[@Example]` 用作 `Example` 属性，将 `[ExampleAttribute]` 用作“ExampleAttribute”。

下面的示例生成 CS1614：

C#

```
// CS1614.cs
using System;

// Both of the following classes are valid attributes with valid
// names (MySpecial and MySpecialAttribute). However, because the lookup
// rules for attributes involves auto-appending the 'Attribute' suffix
// to the identifier, these two attributes become ambiguous; that is,
// if you specify MySpecial, the compiler can't tell if you want
// MySpecial or MySpecialAttribute.

public class MySpecial : Attribute {
   public MySpecial() {}
}

public class MySpecialAttribute : Attribute {
   public MySpecialAttribute() {}
}

class MakeAWarning {
   [MySpecial()] // CS1614
                 // Ambiguous: MySpecial or MySpecialAttribute?
   public static void Main() {
   }

   [@MySpecial()] // This isn't ambiguous, it binds to the first attribute
above.
   public static void NoWarning() {
   }

   [MySpecialAttribute()] // This isn't ambiguous, it binds to the second
attribute above.
   public static void NoWarning2() {
```

```
    }

    [@MySpecialAttribute()] // This is also legal.
    public static void NoWarning3() {
    }
}
```

编译器错误 CS1615

项目 • 2023/04/07

参数“number”不应与关键字“keyword”一起传递

使用了 `ref` 或 **out** 中的其中一个关键字，但函数未采用该实参的 `ref` 或 **out** 形参。 若要解决此错误，请删除不正确的关键字，并使用正确的关键字来匹配函数声明（如果存在这种关键字）。

下面的示例生成 CS1615：

C#

```
// CS1615.cs
class C
{
   public void f(int i) {}
   public static void Main()
   {
      int i = 1;
      f(ref i);  // CS1615
   }
}
```

编译器错误 CS1617

项目 • 2023/05/10

LangVersion 的选项"option"无效。 使用 `<LangVersion>?</LangVersion>` 列出受支持的值。

如果使用 LangVersion 命令行开关或项目设置，但未指定有效的语言选项，则会发生此错误。 若要解决此错误，请检查命令行语法或项目设置并将其更改为其中一个列出的选项。

例如，用 `csc -langversion:ISO` 编译将生成错误 CS1617。

-langversion 的有效值

语言版本的有效值取决于你正在使用的 .NET 版本。 若要详细了解哪些语言版本可用于哪个 .NET 版本，请参阅语言版本规则。 如果在尝试使用较新的语言版本时收到此错误，请降级到较低的语言版本，或者将 .NET SDK 更新为支持该语言版本的版本。

下表指定 `-langversion` 的当前有效值：

值	含义
`preview`	编译器接受最新预览版中的所有有效语言语法。
`latest`	编译器接受最新发布的编译器版本（包括次要版本）中的语法。
`latestMajor` 或 `default`	编译器接受最新发布的编译器主要版本中的语法。
`11.0`	编译器仅接受 C# 11 或更低版本中包含的语法。
`10.0`	编译器只接受 C# 10 或更低版本中所含的语法。
`9.0`	编译器只接受 C# 9 或更低版本中所含的语法。
`8.0`	编译器只接受 C# 8.0 或更低版本中所含的语法。
`7.3`	编译器只接受 C# 7.3 或更低版本中所含的语法。
`7.2`	编译器只接受 C# 7.2 或更低版本中所含的语法。
`7.1`	编译器只接受 C# 7.1 或更低版本中所含的语法。
`7`	编译器只接受 C# 7.0 或更低版本中所含的语法。
`6`	编译器只接受 C# 6.0 或更低版本中所含的语法。

值	含义
`5`	编译器只接受 C# 5.0 或更低版本中所含的语法。
`4`	编译器只接受 C# 4.0 或更低版本中所含的语法。
`3`	编译器只接受 C# 3.0 或更低版本中所含的语法。
`ISO-2` 或 `2`	编译器只接受 ISO/IEC 23270:2006 C# (2.0) 中所含的语法。
`ISO-1` 或 `1`	编译器只接受 ISO/IEC 23270:2003 C# (1.0/1.2) 中所含的语法。

编译器错误 CS1618

项目 • 2023/04/07

无法用“method”创建委托，因为它具有 Conditional 特性

无法用 conditional 方法创建委托，因为该方法可能不存在于某些生成中。

下面的示例生成 CS1618：

C#

```
// CS1618.cs
using System;
using System.Diagnostics;

delegate void del();

class MakeAnError {
   public static void Main() {
      del d = new del(ConditionalMethod);   // CS1618
      // Invalid because on builds where DEBUG is not set,
      // there will be no "ConditionalMethod".
   }
   // To fix the error, remove the next line:
   [Conditional("DEBUG")]
   public static void ConditionalMethod()
   {
      Console.WriteLine("Do something only in debug");
   }
}
```

编译器错误 CS1619

项目 • 2023/04/07

无法创建临时文件“filename”- 原因

由于给定的原因（例如磁盘已满），编译器无法创建临时文件。

编译器错误 CS1620

项目 • 2023/04/07

参数“number”必须使用关键字“keyword”传递

如果将参数传递到采用 ref 或 out 参数的函数，并且在调用时不包含 `ref` 或 `out` 关键字，或包括了错误的关键字，则都会发生此错误。 错误文本指示要使用的相应关键字，以及导致失败的参数。

下面的示例生成 CS1620：

C#

```
// CS1620.cs
class C
{
    void f(ref int i) {}
    public static void Main()
    {
        int x = 1;
        f(out x);  // CS1620 - f takes a ref parameter, not an out parameter
        // Try this line instead:
        // f(ref x);
    }
}
```

编译器错误 CS1630

项目 • 2023/04/07

ErrorReport 的选项“option”无效；必须是 prompt、send、queue 或 none

命令行选项 ErrorReport 后面必须跟 ， 以在发生内部编译器错误时指定想要执行的操作。

> {!NOTE} 不再支持此编译器选项。

与 `yield return` 语句和迭代器方法相关的错误和警告

项目 • 2024/12/21

有许多与 语句和迭代器方法相关的`yield return`：

- CS1622：*无法从迭代器中返回值。请使用 yield return 语句返回值，或生成中断以结束迭代。*
- CS1624：*“accessor”的正文不能为迭代器块，因为“type”不是迭代器接口类型*
- CS1625：*无法在 finally 子句体中生成*
- CS1626：*无法在包含 catch 子句的 Try 块体中生成值*
- CS1627：*yield return 之后应为表达式*
- CS1629：*迭代器中不能出现不安全的代码*
- CS1631：*无法在 catch 子句体中生成值*
- CS1637：*迭代器不能有不安全的参数或 yield 类型*
- CS4013：*类型的实例不能在嵌套函数、查询表达式、迭代器块或异步方法中使用*
- CS8154：*主体不能是迭代器块，因为它按引用返回*
- CS8176：*迭代器不能有按引用局部变量*
- CS9238：*无法在“不安全”的块中使用“yield return”*
- CS9239：*该 `&` 运算符不能用于迭代器方法中的参数或局部变量。*

迭代器方法的结构

迭代器方法必须符合 C# 中的多个规则。 当你的迭代器方法违反其中一个或多个规则时，编译器会发出以下错误：

- CS1622：*无法从迭代器中返回值。请使用 yield return 语句返回值，或生成中断以结束迭代。*
- CS1624：*“accessor”的正文不能为迭代器块，因为“type”不是迭代器接口类型*
- CS1627：*yield return 之后应为表达式*
- CS1637：*迭代器不能有不安全的参数或 yield 类型*
- CS8154：*主体不能是迭代器块，因为它按引用返回*

你的迭代器方法必须遵循以下规则：

- 迭代器方法（使用 `yield return` 和可选的 `yield break`）也不能使用 `return` 语句返回序列。
- 迭代器方法必须将*迭代器接口类型*声明为返回类型。 *迭代器接口类型*为：IEnumerable、IEnumerable<T>、IEnumerator、IEnumerator<T>。

- `yield return` 语句必须包含一个表达式才能作为序列的一部分返回。 `yield return;` 无效。
- 迭代器方法不能将不安全的类型用作参数，如指针。
- 迭代器方法不能 `yield return` 不安全的类型，例如指针。
- 迭代器方法不能通过 `yield return ref`。 你必须按值返回。

迭代器方法的限制

迭代器方法的主体必须符合 `yield return` 语句及其上下文的限制。 当迭代器违反以下一个限制时，编译器会发出以下错误：

- **CS1625**：*无法在 finally 子句体中生成*
- **CS1626**：*无法在包含 catch 子句的 Try 块体中生成值*
- **CS1631**：*无法在 catch 子句体中生成值*
- **CS1629**：*迭代器中不能出现不安全的代码*
- **CS9238**：*无法在"不安全"的块中使用"yield return"*
- **CS9239**：*该 `&` 运算符不能用于迭代器方法中的参数或局部变量。*

这些错误表明你的代码违反了安全规则，因为迭代器返回一个元素并继续生成下一个元素：

- 你不能通过 `yield return` 或 `catch` 子句执行 `finally` 操作。
- 你不能通过 catch 子句从 `yield return` 执行 `try` 操作。
- 你不能从 `yield return` 块中执行 `unsafe` 操作。 迭代器的上下文在封闭的 `safe` 块中创建嵌套的 `unsafe` 块。
- 你不能使用 `&` 运算符在迭代器方法中获取变量的地址。

在 C# 13 之前，迭代器不能包含 `unsafe` 代码 (CS1629)。 从 C# 13 开始，此限制已放宽。 所有 `yield return` 语句都必须位于安全上下文中，但迭代器方法可以包含 `unsafe` 代码。

迭代器方法中的 ref 安全性

迭代器方法具有特殊的 ref 安全性限制。 C# 13 中放宽了这些规则：

- **CS4013**：*类型的实例不能在嵌套函数、查询表达式、迭代器块或异步方法中使用*
- **CS8176**：*迭代器不能有按引用局部变量*

在 C# 13 之前，迭代器无法声明 `ref` 局部变量。 它们无法声明 `ref struct` 类型的任何变量。

从 C# 13 开始，可以在迭代器方法中使用 `ref struct` 类型（如果未跨 `yield return` 语句访问它们）。 从 C# 13 开始，迭代器方法可以声明 `ref` 局部变量。

编译器错误 CS1638

项目 • 2023/06/08

“identifier”是保留的标识符，并且不能在使用 ISO 语言版本模式时使用

当 **/langversion** 编译器开关指定了 ISO 语言兼容性选项时，在标识符中任意位置具有双下划线的任何标识符都将造成此错误。 若要避免此错误，请消除任何带双下划线的标识符或不使用 ISO-1 语言版本选项。

示例

下面的示例生成 CS1638：

C#

```
// CS1638.cs
// compile with: /langversion:ISO-1
class bad__identifier // CS1638 (double underscores are not ISO compliant)
{
}

// Try this instead:
//class GoodIdentifier
//{
//}

class CMain
{
    public static void Main() { }
}
```

请参阅

- LangVersion（C# 编译器选项）

编译器错误 CS1639

项目 • 2023/05/10

接口“interface”的托管组件类包装器类签名“signature”不是有效的类名签名

如果该接口的元数据无效，将发生此错误。 如果使用 C# 或受支持的 .NET 语言生成接口，则该错误永远不会发生。 与该接口的供应商一起进行检查，或验证已成功生成的接口程序集。

编译器错误 CS1640

项目 • 2023/04/07

foreach 语句不能对“type”类型的变量进行操作，因为它实现“interface”的多个实例化，请尝试强制转换为特定的接口实例化

此类型从 IEnumerator<T> 的两个或更多的实例继承，这意味着不存在 `foreach` 可以使用的类型唯一枚举。 指定 IEnumerator<T> 的类型，或使用其他循环构造。

示例

下面的示例生成 CS1640：

C#

```
// CS1640.cs

using System;
using System.Collections;
using System.Collections.Generic;

public class C : IEnumerable, IEnumerable<int>, IEnumerable<string>
{
    IEnumerator<int> IEnumerable<int>.GetEnumerator()
    {
        yield break;
    }

    IEnumerator<string> IEnumerable<string>.GetEnumerator()
    {
        yield break;
    }

    IEnumerator IEnumerable.GetEnumerator()
    {
        return (IEnumerator)((IEnumerable<string>)this).GetEnumerator();
    }
}

public class Test
{
    public static int Main()
    {
        foreach (int i in new C()){}    // CS1640

        // Try specifying the type of IEnumerable<T>
        // foreach (int i in (IEnumerable<int>)new C()){}
        return 1;
```

```
    }
}
```

编译器错误 CS1641

项目 • 2023/04/07

固定大小缓冲区字段的字段名称后必须带有数组大小说明符

与常规数组不同，固定大小的缓冲区需要在声明点中指定常量大小。 若要解决此错误，请添加一个正整数字符或正整数常数，并在标识符后面加上方括号

下面的示例生成 CS1641：

C#

```
// CS1641.cs
// compile with: /unsafe /target:library
unsafe struct S {
   fixed int [] a;  // CS1641

   // OK
   fixed int b [10];
   const int c = 10;
   fixed int d [c];
}
```

编译器错误 CS1642

项目 • 2023/04/07

固定大小的缓冲区字段只能是结构的成员。

如果你使用 `class`而不是 `struct`中固定大小的缓冲区字段，则会发生此错误。 若要解决此错误，请将 `class` 更改为 `struct` ，或将字段声明为一个普通数组。

示例

下面的示例生成 CS1642。

C#

```
// CS1642.cs
// compile with: /unsafe /target:library
unsafe class C
{
   fixed int a[10];   // CS1642
}

unsafe struct D
{
    fixed int a[10];
}

unsafe class E
{
   public int[] a = null;
}
```

编译器错误 CS1643

项目 • 2023/04/07

并非所有代码路径都会返回类型为“type!”的方法中的值

如果一个委托体没有返回语句，或者它的返回语句无法由编译器访问，则会出现此错误。在下面的示例中，编译器不尝试预测分支条件的结果以验证匿名方法块是否始终返回一个值。

示例

下面的示例生成 CS1643：

C#

```
// CS1643.cs
delegate int MyDelegate();

class C
{
    static void Main()
    {
        MyDelegate d = delegate
        {                   // CS1643
            int i = 0;
            if (i == 0)
                return 1;
        };
    }
}
```

编译器错误 CS1644

项目 • 2023/04/07

功能“功能”不是标准化 ISO C# 语言规范的一部分，其他编译器可能不接受它

如果已指定 LangVersion 选项 ISO-1，并且正在编译的代码使用不属于 ISO 1.0 标准的功能，则会发生此错误。 若要解决此错误，请勿使用任何 C# 2.0 编译器功能，例如具有 ISO-1 兼容性选项的泛型或匿名方法。

下面的示例生成 CS1644：

C#

```
// CS1644.cs
// compile with: /langversion:ISO-1 /target:library
class C<T> {}   // CS1644
```

编译器错误 CS1646

项目 • 2023/04/07

原意说明符 @ 后面应是关键字、标识符或字符串。

有关原意说明符“@”的用法，请参阅字符串参数。 仅允许原意说明符位于字符串、关键字或标识符之前。 若要解决此错误，从任何不适当的位置删除 @ 符号，或添加需要的字符串、关键字或标识符。

下面的示例生成 CS1646：

C#

```
// CS1646
class C
{
   int i = @5;  // CS1646
   // Try this line instead:
   // int i = 5;
}
```

编译器错误 CS1647

项目 • 2023/06/08

表达式太长或者过于复杂，无法在“code”附近编译

处理你的代码的编译器中有堆栈溢出。 若要解决此错误，请简化你的代码。 如果代码有效，请在开发者社区网站或 StackOverflow 中获取帮助。

编译器错误 CS1648

项目 • 2023/04/07

无法修改只读字段“identifier”的成员(构造函数或变量初始值设定项中除外)

当你试图修改只读字段中不允许修改的成员时，将发生此错误。 若要解决此错误，请将只读字段的赋值限定为构造函数或变量初始值设定项，或从该字段的声明中删除 readonly 关键字。

示例

下面的示例生成 CS1648：

C#

```
// CS1648.cs
public struct Inner
{
    public int i;
}

class Outer
{
    public readonly Inner inner = new Inner();
}

class D
{
    static void Main()
    {
        var outer = new Outer();
        outer.inner.i = 1;  // CS1648
    }
}
```

编译器错误 CS1650

项目 • 2023/04/07

无法给静态只读字段“identifier”的字段赋值（在静态构造函数或变量初始值设定项中除外）

当你试图修改静态只读字段中不允许修改的成员时，将发生此错误。 若要解决此错误，请将只读字段的赋值限定为构造函数或变量初始值设定项，或从该字段的声明中删除 `readonly` 关键字。

C#

```
// CS1650.cs
public struct Inner
{
    public int i;
}

class Outer
{
    public static readonly Inner inner = new Inner();
}

class D
{
    static void Main()
    {
        Outer.inner.i = 1;  // CS1650
    }
}
```

编译器错误 CS1654

项目 • 2023/04/07

无法修改“variable”的成员，因为它是只读变量类型

当你尝试修改因在特殊构造中而只读的变量的成员时，将出现此错误。

发生这种情况的一个常见区域是在 foreach 循环内部。 它是修改集合元素的值时导致的编译时错误。 因此，你不能对 值类型（其中包括 结构）的元素做任何修改。 在其元素为 引用类型的集合中，你可以修改每个元素的成员，但尝试添加、删除或更换全部元素将生成 Compiler Error CS1656。

示例

下面的示例生成错误 CS1654，因为 `Book` 是 `struct`。 若要修复此错误，请将 `struct` 更改为 类。

C#

```
using System.Collections.Generic;
using System.Text;

namespace CS1654
{

    struct Book
    {
        public string Title;
        public string Author;
        public double Price;
        public Book(string t, string a, double p)
        {
            Title=t;
            Author=a;
            Price=p;

        }
    }

    class Program
    {
        List<Book> list;
        static void Main(string[] args)
        {
             //Use a collection initializer to initialize the list
            Program prog = new Program();
            prog.list = new List<Book>();
            prog.list.Add(new Book ("The C# Programming Language",
```

```
                                    "Hejlsberg, Wiltamuth, Golde",
                                     29.95));
            prog.list.Add(new Book ("The C++ Programming Language",
                                    "Stroustrup",
                                     29.95));
            prog.list.Add(new Book ("The C Programming Language",
                                    "Kernighan, Ritchie",
                                    29.95));
            foreach(Book b in prog.list)
            {
                //Compile error if Book is a struct
                //Make Book a class to modify its members
                b.Price +=9.95; // CS1654
            }

        }
    }
}
```

编译器错误 CS1656

项目 • 2023/03/14

无法给“variable”赋值，因为它是“只读变量类型”

对变量赋值发生在只读上下文中时，会出现此错误。 只读上下文包括 foreach 迭代变量、using 变量和 fixed 变量。 若要解决此错误，应避免给 `using` 块、`foreach` 语句和 `fixed` 语句中的语句变量赋值。

示例 1

下面的示例生成错误 CS1656，因为它尝试替换 `foreach` 循环内集合的全部元素。 解决该错误的一种方法是将 `foreach` 循环更改为 for 循环。 另一个方法（此处未显示）是修改现有元素的成员；该方法对类可能有用，但对结构不起作用。

C#

```
using System;
using System.Collections;
using System.Collections.Generic;
using System.Text;

namespace CS1656_2
{

    class Book
    {
        public string Title;
        public string Author;
        public double Price;
        public Book(string t, string a, double p)
        {
            Title=t;
            Author=a;
            Price=p;

        }
    }

    class Program
    {
        private List<Book> list;
        static void Main(string[] args)
        {
            Program prog = new Program();
            prog.list = new List<Book>();
            prog.list.Add(new Book ("The C# Programming Language",
                                    "Hejlsberg, Wiltamuth, Golde",
```

```
                                    29.95));
            prog.list.Add(new Book ("The C++ Programming Language",
                                    "Stroustrup",
                                    29.95));
            prog.list.Add(new Book ("The C Programming Language",
                                    "Kernighan, Ritchie",
                                    29.95));
            foreach(Book b in prog.list)
            {
                // Cannot modify an entire element in a foreach loop
                // even with reference types.
                // Use a for or while loop instead
                if (b.Title == "The C Programming Language")
                    // Cannot assign to 'b' because it is a 'foreach
                    // iteration variable'
                    b = new Book("Programming Windows, 5th Ed.", "Petzold",
29.95); //CS1656
            }

            //With a for loop you can modify elements
            //for(int x = 0; x < prog.list.Count; x++)
            //{
            //    if(prog.list[x].Title== "The C Programming Language")
            //        prog.list[x] = new Book("Programming Windows, 5th
Ed.", "Petzold", 29.95);
            //}
            //foreach(Book b in prog.list)
            //    Console.WriteLine(b.Title);

        }
    }
}
```

示例 2

下面的示例演示如何在除 `foreach` 循环以外的其他上下文中生成 CS1656：

C#

```
// CS1656.cs
// compile with: /unsafe
using System;

class C : IDisposable
{
    public void Dispose() { }
}

class CMain
{
    unsafe public static void Main()
```

```
    {
        using (C c = new C())
        {
            // Cannot assign to 'c' because it is a 'using variable'
            c = new C(); // CS1656
        }

        int[] ary = new int[] { 1, 2, 3, 4 };
        fixed (int* p = ary)
        {
            // Cannot assign to 'p' because it is a 'fixed variable'
            p = null; // CS1656
        }
    }
}
```

编译器错误 CS1660

项目 • 2023/04/07

不能将匿名方法块转换为类型“type”，因为它不是委托类型

如果你尝试赋值或以其他方式将匿名方法块转换为不是委托类型的类型，则将发生此错误。

下面的示例生成 CS1660：

C#

```
// CS1660.cs
delegate int MyDelegate();
class C {
   static void Main()
   {
     int i = delegate { return 1; };  // CS1660
     // Try this instead:
     // MyDelegate myDelegate = delegate { return 1; };
     // int i = myDelegate();
   }
}
```

编译器错误 CS1661

项目 • 2023/04/07

不能将匿名方法块转换为委托类型“delegate type”，因为指定块的参数类型与委托参数类型不匹配

如果在匿名方法定义中，匿名方法的参数类型不匹配委托参数类型，会发生此错误。 检查参数数目、参数类型和所有 ref 或 out 参数并验证是否完全匹配。

以下示例生成 CS1661：

C#

```
// CS1661.cs

delegate void MyDelegate(int i);

class C
{
    public static void Main()
    {
        MyDelegate d = delegate(string s) { };  // CS1661
    }
}
```

编译器错误 CS1662

项目 • 2023/04/07

无法将匿名方法块转换为委托类型“delegate type”，原因是块中的某些返回类型不能隐式转换为委托返回类型

如果匿名方法块的 return 语句具有不可隐式转换为委托返回类型的类型，则会发生此错误。

下面的示例生成 CS1662：

C#

```
// CS1662.cs

delegate int MyDelegate(int i);

class C
{

  public static void Main()
  {
     MyDelegate d = delegate(int i) { return 1.0; };  // CS1662
     // Try this instead:
     // MyDelegate d = delegate(int i) { return (int)1.0; };
  }
}
```

编译器错误 CS1663

项目 • 2023/04/07

固定大小的缓冲区类型必须为下列类型之一: bool、byte、short、int、long、char、sbyte、ushort、uint、ulong、float 或 double

固定大小的缓冲区可能不是所列类型之外的任何类型。 若要避免此错误，请使用另一种类型或不使用固定数组。

示例

下面的示例生成 CS1663。

C#

```
// CS1663.cs
// compile with: /unsafe /target:library

unsafe struct C
{
   fixed string ab[10];   // CS1663
}
```

编译器错误 CS1664

项目 • 2024/03/20

长度为“length”、类型为“type”的固定大小缓冲区太大

固定大小缓冲区的最大大小（由长度乘以元素大小决定）为 2^31 = 2147483648。

在 GitHub 上与我们协作

可以在 GitHub 上找到此内容的源，还可以在其中创建和查看问题和拉取请求。 有关详细信息，请参阅参与者指南。

.NET

.NET 反馈

.NET 是一个开放源代码项目。 选择一个链接以提供反馈：

提出文档问题

提供产品反馈

编译器错误 CS1665

项目 • 2023/04/07

固定大小缓冲区的长度必须大于零

如果将固定大小的缓冲区声明为零或负数，则会发生此错误。 固定大小的缓冲区的长度必须是正整数。

示例

下面的示例生成 CS1665。

C#

```
// CS1665.cs
// compile with: /unsafe /target:library
struct S
{
   public unsafe fixed int A[0];   // CS1665
}
```

编译器错误 CS1666

项目 • 2023/04/23

不能使用非固定表达式中包含的固定大小缓冲区。 请尝试使用 fixed 语句。

如果在包含本身不固定实例的表达式中使用固定大小的缓冲区，将出现此错误。 运行时可以自由移动非固定实例以优化内存访问，但使用固定大小缓冲区时，这可能会导致错误。 若要避免此错误，请使用 fixed 语句。

示例

下面的示例生成 CS1666。

C#

```
// CS1666.cs
// compile with: /unsafe /target:library
unsafe struct S
{
   public fixed int buffer[1];
}

unsafe class Test
{
   S field = new S();

   private bool example1()
   {
      return (field.buffer[0] == 0);   // CS1666 error
   }

   private bool example2()
   {
      // OK
      fixed (S* p = &field)
      {
         return (p->buffer[0] == 0);
      }
   }

   private bool example3()
   {
      S local = new S();
      return (local.buffer[0] == 0);
   }
}
```

编译器错误 CS1667

项目 • 2023/04/07

特性“attribute”对属性或事件访问器无效。 仅对“declaration type”声明有效。

如果当特性本身应在属性或事件上时，对该属性或事件访问器使用该特性，会发生此错误。 使用 CLSCompliantAttribute、 ConditionalAttribute和 ObsoleteAttribute特性时会发生此错误。

示例

以下示例生成 CS1670：

C#

```
// CS1667.cs
using System;

public class C
{
    private int i;

    //Try this instead:
    //[Obsolete]
    public int ObsoleteProperty
    {
        [Obsolete]  // CS1667
        get { return i; }
        set { i = value; }
    }

    public static void Main()
    {
    }
}
```

编译器错误 CS1671

项目 • 2023/04/07

命名空间声明不能有修饰符或特性

修饰符在应用于命名空间时没有意义，因此不允许使用修饰符。

下面的示例生成 CS1671：

C#

```
// CS1671.cs
public namespace NS // CS1671
{

}
```

编译器错误 CS1672

项目 • 2023/04/07

选项“option”对 /platform 无效；必须是 anycpu、x86、Itanium 或 x64

选项指定处理器类型；使用列出的其中一个窗体。

编译器错误 CS1674

项目 • 2023/03/20

“T”：using 语句中使用的类型必须可以隐式转换为“System.IDisposable”

using 语句旨在用于确保在 `using` 块的末尾释放对象，因此只有可释放的类型才能在这类语句中使用。 例如，值类型不是可释放的，并且不能将没有约束为类的类型参数假定为可释放。

示例 1

下面的示例生成 CS1674。

C#

```
// CS1674.cs
class C
{
   public static void Main()
   {
      int a = 0;
      a++;

      using (a) {}   // CS1674
   }
}
```

示例 2

下面的示例生成 CS1674。

C#

```
// CS1674_b.cs
using System;
class C {
   public void Test() {
      using (C c = new C()) {}   // CS1674
   }
}

// OK
class D : IDisposable {
   void IDisposable.Dispose() {}
   public void Dispose() {}
```

```
    public static void Main() {
        using (D d = new D()) {}
    }
}
```

示例 3

下面的示例说明要确保可释放未知类型参数，需要一个类类型约束。 下面的示例生成 CS1674。

C#

```
// CS1674_c.cs
// compile with: /target:library
using System;
public class C<T>
// Add a class type constraint that specifies a disposable class.
// Uncomment the following line to resolve.
// public class C<T> where T : IDisposable
{
    public void F(T t)
    {
        using (t) {}   // CS1674
    }
}
```

编译器错误 CS1675

项目 • 2023/04/07

枚举不能有类型参数

若要解决此错误，请删除 `enum` 声明中的类型形参。

示例

下面的示例生成 CS1675：

C#

```
// CS1675.cs
enum E<T>  // CS1675
{
}

class CMain
{
    public static void Main()
    {
    }
}
```

编译器错误 CS1676

项目 • 2023/04/07

参数“number”必须使用“keyword”关键字进行声明

当匿名方法中的参数类型修饰符不同于你要将方法转换为的委托的声明中所用的参数类型修饰符时，将出现此错误。

以下示例生成 CS1676：

C#

```
// CS1676.cs
delegate void E(ref int i);
class Errors
{
   static void Main()
   {
      E e = delegate(out int i) { };   // CS1676
      // To resolve, use the following line instead:
      // E e = delegate(ref int i) { };
   }
}
```

编译器错误 CS1677

项目 • 2023/04/07

参数“number”不应使用“keyword”关键字进行声明

当匿名方法中的参数类型修饰符不同于你在要将该方法强制转换为的委托的声明中所用的参数类型修饰符时，将出现此错误。

示例

以下示例生成 CS1677：

C#

```
// CS1677.cs
delegate void D(int i);
class Errors
{
    static void Main()
    {
        D d = delegate(out int i) { };   // CS1677
        // To resolve, use the following line instead:
        // D d = delegate(int i) { };

        D d = delegate(ref int j){}; // CS1677
        // To resolve, use the following line instead:
        // D d = delegate(int j){};
    }
}
```

编译器错误 CS1678

项目 • 2023/04/07

参数“number”被声明为类型“type1”，而它应为“type2”

当匿名方法中的参数类型不同于你要将方法转换为的委托的声明时，将出现此错误。

下面的示例生成 CS1678：

C#

```
// CS1678
delegate void D(int i);
class Errors
{
   static void Main()
   {
      D d = delegate(string s) { };   // CS1678
      // To resolve, use the following line instead:
      // D d = delegate(int s) { };
   }
}
```

编译器错误 CS1679

项目 • 2023/04/07

无效的“/reference”外部别名；“identifier”不是有效的标识符

使用 **/reference** 选项的外部程序集别名功能时，根据 C# 语言规范，位于 **/reference:** 之后、“=”之前的文本必须为有效的 C# 标识符或关键字。

若要更正此错误，将“=”之前的文本更改为有效的 C# 标识符或关键字。

示例

以下示例生成 CS1679。

C#

```
// CS1679.cs
// compile with: /reference:123$BadIdentifier%=System.dll
class TestClass {
    static void Main()
    {
    }
}
```

编译器错误 CS1680

项目 • 2023/04/07

无效的引用别名选项:“alias=”-- 缺少文件名。

将 `alias` 功能与 **/reference** 编译器选项结合使用而未指定有效文件名时，会发生此错误。

下面的示例生成 CS1680。

C#

```
// CS1680.cs
// compile with: /reference:alias=
// CS1680 expected
// To resolve, specify the name of a file with an assembly manifest
class MyClass {}
```

编译器错误 CS1681

项目 • 2023/04/07

不能重新定义全局外部别名

全局别名已定义为包括所有非别名引用，因此无法重新定义。

示例

下面的示例生成 CS1681。

C#

```
// CS1681.cs
// compile with: /reference:global=System.dll
// CS1681 expected

// try this instead: /reference:System.dll
class A
{
   static void Main() {}
}
```

编译器错误 CS1688

项目 • 2023/04/07

无法将不含参数列表的匿名方法块转换为委托类型“delegate”，原因是该方法块具有一个或多个 out 参数

大多数情况下，编译器允许省略匿名方法块中的参数。 如果匿名方法块不具有参数列表，而委托具有 `out` 参数，则将出现此错误。 由于编译器需要忽略 `out` 参数的存在（而这不太可能是正确行为），因此它不允许这种情况。

示例

以下代码生成错误 CS1688。

C#

```
// CS1688.cs
using System;
delegate void OutParam(out int i);
class ErrorCS1676
{
    static void Main()
    {
        OutParam o;
        o = delegate  // CS1688
        // Try this instead:
        // o = delegate(out int i)
        {
            Console.WriteLine("");
        };
    }
}
```

编译器错误 CS1689

项目 • 2023/04/07

特性“Attribute Name”仅对方法或特性类有效

此错误仅出现在 **ConditionalAttribute** 特性中。 如消息所称，此特性仅用于方法或特性类。 例如，尝试将此特性应用于类将生成此错误。

示例

下面的示例生成 CS1689：

C#

```
// CS1689.cs
// compile with: /target:library
[System.Diagnostics.Conditional("A")]   // CS1689
class MyClass {}
```

编译器错误 CS1703

项目 • 2023/04/07

已导入具有相同的简单名称“name”的程序集。 尝试删除某个引用，或者为引用签名，从而启用并行。

编译器删除具有相同的路径和文件名的引用，但很可能相同的文件存在于两个位置中，或者你忘记更改版本号。 此错误指出两个引用具有相同的程序集标识，因此编译器无法在元数据中区分它们。 请删除其中一个冗余的引用，或通过某种方法（例如递增程序集版本号）使引用唯一。

下面的代码生成错误 CS1703。

示例 1

此代码在 .\bin1 目录创建程序集 A。

将此示例保存在一个名为 CS1703a1.cs 的文件中，并使用以下标记进行编译：`/t:library /out:.\bin1\cs1703.dll /keyfile:key.snk`

```
C#

using System;
public class A { }
```

示例 2

此代码在 .\bin2 目录创建程序集 A 的副本。

将此示例保存在一个名为 CS1703a2.cs 的文件中，并使用以下标记进行编译：`/t:library /out:.\bin2\cs1703.dll /keyfile:key.snk`

```
C#

using System;
public class A { }
```

示例 3

此代码引用两个之前的模块中的程序集 A。

将此示例保存在一个名为 CS1703ref.cs 的文件中，并使用以下标记进行编译：`/t:library /r:A2=.\bin2\cs1703.dll /r:A1=.\bin1\cs1703.dll`

C#

```
extern alias A1;
extern alias A2;
```

编译器错误 CS1705

项目 • 2023/04/07

程序集“AssemblyName1”使用的“TypeName”版本高于引用的程序集“AssemblyName2”使用的版本

您当前访问的类型的版本号高于所引用的程序集的版本号。 此错误通常由意外使用相同程序集的两个版本导致。

例如，假设您有两个程序集，Asmb1 和 Asmb2。 程序集 Asmb1 引用程序集 Asmb2 1.0 版。 程序集 Asmb1 还使用已添加到程序集 Asmb2 2.0 版的类 `MyClass`。 编译器拥有统一的引用绑定规则，对 2.0 版中的 `MyClass` 的引用无法由 1.0 版满足。

示例

下面更为详细的示例由四个代码模块构成：

- 两个 DLL，它们除版本属性外，其他方面都相同。
- 引用前两个 DLL 的第三个 DLL。
- 仅引用相同 DLL 的 1.0 版但访问 2.0 版中的类的客户端。

下面的代码创建这两个相同 DLL 中的第一个。 有关如何生成密钥文件的信息，请参阅 KeyFile（C# 编译器选项）。

C#

```
// CS1705a.cs

// Compile by using the following command:
//      csc /target:library /out:C:\\CS1705.dll /keyfile:mykey.snk
CS1705a.cs

// The DLL is created in the C:\ directory.

// The AssemblyVersion attribute specifies version 1.0 for this DLL.

[assembly:System.Reflection.AssemblyVersion("1.0")]
public class Class1
{
   public void Method1() {}
}
```

下面的代码定义程序集 2.0 版，如特性 AssemblyVersionAttribute 所指定。

C#

```
// CS1705b.cs

// Compile by using the following command:
//     csc /target:library /out:CS1705.dll /keyfile:mykey.snk CS1705b.cs

// The DLL is created in the current directory.

// The AssemblyVersion attribute specifies version 2.0 for this DLL.

[assembly:System.Reflection.AssemblyVersion("2.0")]
public class Class1
{
    public void Method1() { }
}
```

下面的代码引用前面的代码中定义的两个 DLL 版本。 `AssemblyA` 引用由 CS1705a.cs（1.0 版）创建的 DLL。 `AssemblyB` 引用由 CS1705b.cs（2.0 版）创建的 DLL。 在 `ClassC` 中定义两个方法。 第一个方法 `Return1A` 返回 `Class1A` 类型的对象，该对象是 DLL 1.0 版的中 `Class1` 的别名。 第二个方法 `Return1B` 返回 `Class1B` 类型的对象，该对象是 DLL 2.0 版中的 `Class1` 的别名。 `Return1A` 的定义对版本 1.0 形成依赖关系；`Return1B` 的定义对版本 2.0 形成依赖关系。

C#

```
// CS1705c.cs

// Compile by using the following command. AssemblyA refers to the DLL
created by
// CS1705a.cs (version 1.0). AssemblyB refers to the DLL created by
CS1705b.cs
// (version 2.0).
//    csc /target:library /r:AssemblyA=C:\\CS1705.dll
/r:AssemblyB=CS1705.dll CS1705c.cs

extern alias AssemblyA;
extern alias AssemblyB;

// Class1A is an alias for type Class1 from VS1705a.cs, which is in version
1.0
// of the assembly. Class1B is an alias for type Class1 from CS1705b.cs,
which
// is in version 2.0 of the assembly.

using Class1A = AssemblyA::Class1;
using Class1B = AssemblyB::Class1;

// Method Return1A in ClassC returns an object of type Class1A, which is
// Class1 from version 1.0 of the DLL. Method Return1B returns an object
// of type Class1B, which is Class1 from version 2.0 of the DLL.
```

```
public class ClassC
{
    // The following line creates a dependency on version 1.0 of CS1705.dll.
    // This is not the source of the problem when ClassC is accessed from
    // CS1705d.cs because CS1705d.cs references version 1.0 of the DLL.
    // Therefore, type Class1A and the assembly have the same version.
    public static Class1A Return1A() { return new Class1A(); }

    // The following line creates a dependency on version 2.0 of CS1705.dll.
    // This causes compiler error CS1705 when ClassC is accessed from
    // CS1705d.cs, because CS1705d.cs does not reference version 2.0 of the
    // DLL. Class1B is the alias for Class1 in version 2.0, and CS1705d.cs
    // references version 1.0.
    public static Class1B Return1B() { return new Class1B(); }
}
```

下面的代码生成编译器错误 CS1705。 它引用由 CS1705a.cs（1.0 版）创建的 DLL。 但是，在 `Main` 方法中，代码访问 CS1705c.cs 中的 `ClassC`。 `ClassC` 使用在 CS1705b.cs（2.0 版）中定义的类型。 这导致编译器错误 CS1705，因为类型中的 CS1705.dll 的版本号高于所引用的 CS1705.dll 的版本号。

C#

```
// CS1705d.cs

// Compile by using the following command:
//     csc /reference:C:\\CS1705.dll /reference:CS1705c.dll CS1705d.cs

// C:\\CS1705.dll is version 1.0 of the assembly.

class Tester
{
    static void Main()
    {
        // Return1A returns a type defined in version 1.0.
        ClassC.Return1A().Method1();
        // Return1B returns a type defined in version 2.0.
        ClassC.Return1B().Method1();
    }
}
```

可以通过以下方式之一来解决这个错误：

- 更新代码，以便所有文件使用相同版本的 DLL。
- 通过使用以下命令进行编译，将对 DLL 2.0 版的引用添加到 CS1705d.cs 中：

  `csc /reference:C:\\CS1705.dll /reference:CS1705.dll /reference:CS1705c.dll CS1705d.cs`

使用此命令时，虽然程序在进行编译，但它并未运行。 若要使程序运行，可提供应用程序配置文件并在其中包含使用 <assemblyIdentity> 和 <codeBase> 子元素的 <dependentAssembly> 元素，指定 DLL 1.0 版的位置。 有关配置文件的详细信息，请参阅配置应用。

另请参阅

- 外部别名
- ::运算符

编译器错误 CS1708

项目 • 2023/04/07

固定大小缓冲区只能通过局部变量或字段进行访问

C# 2.0 中有一项新功能，可在 `struct` 内定义内嵌数组。 此类数组只能通过局部变量或字段进行访问，并且不能作为表达式左侧的中间值进行引用。 此外，不能通过 `static` 或 `readonly` 字段访问这些数组。

若要解决此错误，请定义一个数组变量，并将内嵌数组分配给该变量。 或者，从内嵌数组的字段中删除 `static` 或 `readonly` 修饰符。

示例

下面的示例生成 CS1708。

C#

```
// CS1708.cs
// compile with: /unsafe
using System;

unsafe public struct S
{
    public fixed char name[10];
}

public unsafe class C
{
    public S UnsafeMethod()
    {
        S myS = new S();
        return myS;
    }

    static void Main()
    {
        C myC = new C();
        myC.UnsafeMethod().name[3] = 'a';  // CS1708
        // Uncomment the following 2 lines to resolve:
        // S myS = myC.UnsafeMethod();
        // myS.name[3] = 'a';

        // The field cannot be static.
        C._s1.name[3] = 'a';  // CS1708

        // The field cannot be readonly.
        myC._s2.name[3] = 'a';  // CS1708
```

```
    }

    static readonly S _s1;
    public readonly S _s2;
}
```

编译器错误 CS1713

项目 • 2023/04/07

为类型 Typename1 生成元数据名称时发生错误 --“Reason”

此错误通常是由内部编译器错误引起的。 对代码进行一些小更改（如缩短名称长度），然后进行重新编译，可能会解决此问题。

编译器错误 CS1715

项目 • 2023/04/07

“Type1”：类型必须为“Type2”才能匹配重写的成员“MemberName”

此错误与 Compiler Error CS0508相同，区别在于 CS0508 目前仅适用于具有返回类型的方法，而 CS1715 适用于仅具有“类型”而非“返回类型”的属性和索引器。

示例

下面的代码生成 CS1715。

C#

```
// CS1715.cs
abstract public class Base
{
    abstract public int myProperty
    {
        get;
        set;
    }
}

public class Derived : Base
{
    int myField;
    public override double myProperty  // CS1715
    // try the following line instead
    // public override int myProperty
    {
        get { return myField; }
        set { myField;= value; }
    }

    public static void Main()
    {
        Derived d = new Derived();
        d.myProperty = 5;
    }
}
```

编译器错误 CS1716

项目 • 2023/04/07

请勿使用“System.Runtime.CompilerServices.FixedBuffer”属性。 请改用“fixed”字段修饰符。

在不安全的代码段（包含类似字段声明的固定大小的数组声明）中，会出现此错误。 不要使用此特性。 请改用关键字 `fixed`。

示例

下面的示例生成 CS1716。

C#

```
// CS1716.cs
// compile with: /unsafe
using System;
using System.Runtime.CompilerServices;

public struct UnsafeStruct
{
    [FixedBuffer(typeof(int), 4)]  // CS1716
    unsafe public int aField;
    // Use this single line instead of the above two lines.
    // unsafe public fixed int aField[4];
}

public class TestUnsafe
{
    static int Main()
    {
        UnsafeStruct us = new UnsafeStruct();
        unsafe
        {
            if (us.aField[0] == 0)
                return us.aField[1];
            else
                return us.aField[2];
        }
    }
}
```

编译器错误 CS1719

项目 • 2023/04/07

读取 Win32 资源文件“File Name”时出错 --“reason”

因为错误中所给原因（通常类似于“找不到文件”或“拒绝访问”），尝试读取 Win32 资源文件失败。解决此错误的方式是更正原因所描述的问题。

编译器错误 CS1721

项目 • 2023/04/07

类“class”不能具有多个基类：“class_1”和“class_2”

此错误消息最常见的原因是尝试使用多重继承。 C# 中的类只能直接从一个类继承。 但是，一个类可以实现任何数量的接口。

示例

下面的示例演示可能生成 CS1721 的一种方法：

C#

```
// CS1721.cs
public class A {}
public class B {}
public class MyClass : A, B {}   // CS1721
```

更正此错误

使用以下不同方法，可更正此错误：

- 使类 `B` 继承自 `A`，`MyClass` 继承自 `B`：

  C#

  ```
  public class A {}
  public class B : A {}
  public class MyClass : B {}
  ```

- 将 `B` 声明为接口。 使 `MyClass` 继承自接口 `B` 和类 `A`：

  C#

  ```
  public class A {}
  public interface B {}
  public class MyClass : A, B {}
  ```

请参阅

- 多态性
- 接口

编译器错误 CS1722

项目 • 2023/04/07

基类“class”必须在任何接口之前

指定从中继承的类和要实现的接口时，必须首先指定类名。

示例

下面的示例生成 CS1722。

C#

```
// CS1722.cs
// compile with: /target:library
public class A {}
interface I {}

public class MyClass : I, A {}   // CS1722
public class MyClass2 : A, I {}   // OK
```

编译器错误 CS1724

项目 • 2023/04/07

为“System.Runtime.InteropServices.DefaultCharSetAttribute”的参数指定的值无效

此错误由 DefaultCharSetAttribute 类的无效参数生成。

示例

以下示例生成 CS1724。

C#

```
// CS1724.cs
using System.Runtime.InteropServices;
// To resolve, replace 42 with a valid CharSet value.
 [module:DefaultCharSetAttribute((CharSet)42)]   // CS1724
class C {

[DllImport("F.Dll")]
extern static void FW1Named();

static void Main() {}
}
```

编译器错误 CS1725

项目 • 2023/04/07

友元程序集引用“reference”无效。 不能为 InternalsVisibleTo 声明指定版本、区域性、公钥标记或处理器结构。

不能在友元程序集引用中添加版本区域性。 分部类应对友元程序集可见。

示例

下面的示例生成 CS1725。

C#

```
// CS1725.cs
// compile with: /target:library
using System.Runtime.CompilerServices;
[assembly:InternalsVisibleTo("partial01,version=1.1.0.0")]   // CS1725
// try the following line instead
// [assembly:InternalsVisibleTo("partial01")]

partial class TestClass
{
   public static string strBar = "my string";
}
```

另请参阅

- 如何：创建已签名的友元程序集

编译器错误 CS1726

项目 • 2023/04/07

友元程序集引用“reference”无效。 用强名称签名的程序集必须在他们的 InternalsVisibleTo 声明中指定一个公钥。

用强名称签名的程序集可将由 InternalsVisibleToAttribute 制作的友元程序集访问权限授予其他强名称签名程序集。

若要解决 CS1726，可将程序集签名到（将强名称赋予）要/不授予友元程序集访问权限的程序集。

有关详细信息，请参阅友元程序集。

示例

下面的示例将生成 CS1726。

C#

```
// Save this code as CS1726.cs

// Run the following command to create CS1726.key:
//      sn -k CS1726.key

// Then compile by using the following command:
//      csc /keyfile:CS1726.key /target:library CS1726.cs

using System.Runtime.CompilerServices;

// The following line causes compiler error CS1726.
[assembly: InternalsVisibleTo("UnsignedAssembly")]

// To get rid of the error, try the following line instead.
//[assembly: InternalsVisibleTo("SignedAssembly,
PublicKey=0024000004800000940000000602000000240000525341310004000001000100031d7b6f3abc16c7de526fd67ec2926fe68ed2f9901afbc5f1b6b428bf6cd9086021a0b38b76bc340dc6ab27b65e4a593fa0e60689ac98dd71a12248ca025751d135df7b98c5f9d09172f7b62dabdd302b2a1ae688731ff3fc7a6ab9e8cf39fb73c60667e1b071ef7da5838dc009ae0119a9cbff2c581fc0f2d966b77114b2c4")]

class A { }
```

另请参阅

- 如何：创建已签名的友元程序集

编译器错误 CS1727

项目 • 2023/04/07

未经授权不能自动发送错误报告。 请访问 '' 以授权发送错误报告。

错误文本中列出的网站介绍了如何为 Visual Studio 2005 命令行工具启用自动错误报告。

示例

下面的示例生成 CS1727。

C#

```
// CS1727.cs
// compile with: /errorreport:send
// CS1727 expected
class Test
{
    static void Main(){}
}
```

编译器错误 CS1728

项目 • 2023/04/07

不能将委托绑定到 “member”，因为它是 “type” 的成员

不能将委托绑定到 `Nullable` 值类型的成员。

示例

以下示例生成 CS1728：

C#

```
// CS1728.cs
class Test
{
   delegate T GetT<T>();
   delegate T GetT1<T>(T t);

   delegate bool E(object o);
   delegate int I();
   delegate string S();

   static void Main()
   {
      int? x = null;
      int? y = 5;

      GetT<int> d1 = x.GetValueOrDefault;   // CS1728
      GetT<int> d2 = y.GetValueOrDefault;   // CS1728
      GetT1<int> d3 = x.GetValueOrDefault;   // CS1728
      GetT1<int> d4 = y.GetValueOrDefault;   // CS1728
   }
}
```

编译器错误 CS1729

项目 • 2023/04/07

“type”不包含带有“number”个参数的构造函数。

当直接或间接调用类的构造函数，但编译器找不到任何具有相同数量参数的构造函数时，将发生此错误。 在下面的示例中，`test` 类没有任何构造函数带有任何参数。 因此，该类只有一个不带任何参数的无参数构造函数。 由于在生成该错误的第二行中，派生类没有声明其自己的任何构造函数，因此编译器将提供无参数构造函数。 而该构造函数会调用基类中的无参数构造函数。 但由于基类中没有此构造函数，因此会生成 CS1729。

更正此错误

1. 调整对构造函数的调用中的参数数量。
2. 修改该类，提供具有必须调用的参数的构造函数。
3. 在基类中提供一个无参数构造函数。

示例

下面的示例会生成 CS1729：

C#

```
// cs1729.cs
class Test
{
    static int Main()
    {
        // Class Test has only a parameterless constructor, which takes no
arguments.
        Test test1 = new Test(2); // CS1729
        // The following line resolves the error.
        Test test2 = new Test();

        // Class Parent has only one constructor, which takes two int
parameters.
        Parent exampleParent1 = new Parent(10); // CS1729
        // The following line resolves the error.
        Parent exampleParent2 = new Parent(10, 1);

        return 1;
    }
}
```

```
public class Parent
{
    // The only constructor for this class has two parameters.
    public Parent(int i, int j) { }
}

// The following declaration causes a compiler error because class Parent
// does not have a constructor that takes no arguments. The declaration of
// class Child2 shows how to resolve this error.
public class Child : Parent { } // CS1729

public class Child2 : Parent
{
    // The constructor for Child2 has only one parameter. To access the
    // constructor in Parent, and prevent this compiler error, you must
provide
    // a value for the second parameter of Parent. The following example
provides 0.
    public Child2(int k)
        : base(k, 0)
    {
        // Add the body of the constructor here.
    }
}
```

另请参阅

- 继承
- 构造函数

编译器错误 CS1730

项目 • 2023/03/16

程序集和模块特性必须位于文件中定义的所有其他元素之前（using 子句和外部别名声明除外）。

应用在程序集级别的特性不能出现在任何类型定义之后。

更正此错误

1. 将特性移动到文件顶部，但在 `using` 指令和 `extern` 别名声明之下。

示例

以下代码生成 CS1730：

C#

```
// cs1730.cs
class Test
{
}
[assembly: System.Attribute] // CS1730
```

请参阅

- 特性

编译器错误 CS1731

项目 • 2023/04/07

无法将“表达式”转换为委托，因为块中的某些返回类型不可显式转换为委托返回类型。

当 lambda 表达式或匿名方法有与委托的返回类型不兼容的返回类型时，会生成此错误。

更正此错误

1. 更改委托或表达式的返回类型。

示例

下面的代码生成 CS1731：

C#

```
class CS1731
{
    delegate double D();
    D d = () => { return "Who knows the real sword of Gryffindor?"; };
}
```

编译器错误 CS1732

项目 • 2023/04/07

预期参数。

当 lambda 表达式包含一个跟在输入参数后的逗号而未指定以下参数时，则会产生此错误。

更正此错误

1. 请删除此逗号，或添加编译器预期逗号后应有的输入参数。

示例

下面的示例生成 CS1732：

C#

```
// cs1732.cs
// compile with: /target:library
class Test
    {
        delegate void D(int x, int y);
        static void Main()
        {
        D d = (x,) => { }; // CS1732
      }
    }
```

编译器错误 CS1733

项目 • 2023/04/07

应为表达式。

只要编译器在发生错误的行上需要表达式时，就会生成此错误。 在下面的示例中，在初始值设定项中的尾随逗号指示另一个表达式将遵循的编译器。

更正此错误

- 提供缺少的表达式。
- 删除导致编译器需要表达式的标记。

示例

由于尾随逗号，以下示例将生成 CS1733：

C#

```
// cs1733.cs
using System.Collections.Generic;
public class Test
{
    public static void Main()
    {
        List<int> list = new List<int>() {{ 1, 2, }};// CS1733
    }
}
```

编译器错误 CS1736

项目 • 2023/06/13

默认参数值必须是编译时常量

示例

下面的示例生成 CS1736：

```
C#
```

```
// CS1736.cs

public unsafe class C
{
    static void F(int i = G())
    {
        // ...
    }
    static int G() => 0;
```

在调用方法时计算默认参数值。 除非该值在编译时为常量，否则无法在声明时预先确定最终调用该方法时的值。

更正此错误

如果需要动态计算值，请考虑使用编译时常量作为标记值，然后在运行时对其进行检查：

```
C#
```

```
    static void F(int i = -1)
    {
        if(i == -1) i = G();
        //...
    }
```


Collaborate with us on GitHub

The source for this content can be found on GitHub, where you

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

can also create and review issues and pull requests. For more information, see our contributor guide.

Open a documentation issue

Provide product feedback

编译器错误 CS1737

项目 • 2023/04/07

可选参数必须出现在所有必需参数之后

编译器不支持在声明必需参数之前声明可选参数。 所有可选参数必须出现在所有必需参数之后。

示例

下面的示例生成 CS1737：

C#

```
// CS1737.cs (7,45)
class C
{
    static void F(object? x)
    {
        G(y: x);
    }
    static void G(object? x = null, object y)
    {
    }
}
```

更正此错误

可以更改此方法的签名，同时不影响调用该方法的现有代码，因为尚未使用可选参数的值。 例如：

C#

```
// CS1737.cs (7,45)
class C
{
    static void F(object? x)
    {
        G(y: x);
    }
    static void G(object y, object? x = null)
    {
    }
}
```

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET feedback

The .NET documentation is open source. Provide feedback here.

Open a documentation issue

Provide product feedback

编译器错误 CS1750

项目 • 2023/09/11

不能将类型的值用作默认参数，因为没有到类型的标准转换

示例

下面的示例生成 CS1750：

```
C#
```

```
public struct S
{
    public override string ToString() { return "S::ToString"; }
}
public class A
{
    public static S Goo(S p = 42) { return p; }
}
```

`int` 和新声明的结构 `S` 之间不存在标准转换。 使用 `int` 编译时常量初始化结构 `S` 实例会导致 CS1750。 添加用户定义的转换运算符（例如，`public static implicit operator S(int n) => ...`）将不会更正此错误，因为此操作不会添加*标准转换*。

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

Open a documentation issue

Provide product feedback

编译器错误 CS1763

项目 • 2023/04/07

只能用 Null 对引用类型（字符串除外）的默认参数值进行初始化

示例

下面的示例生成 CS1763：

C#

```
// CS1763.cs (0,0)
class Program
{
    public void Goo<T, U>(T t = default(U)) where U : T
    {
    }
    static void Main(string[] args)
    {

    }
}
```

此示例生成 CS1763，因为尽管有 `U` 派生自基类 `T` 的限制，但当参数的类型为 `T` 时，`Goo<T,U>` 参数使用 `default(U)` 的默认值进行声明。

更正此错误

将 `default(U)` 更改为使用相应的类型参数可更正此错误：

C#

```
    public void Goo<T, U>(T t = default(T)) where U : T
    {
    }
```


Collaborate with us on GitHub

The source for this content can be found on GitHub, where you

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

can also create and review issues and pull requests. For more information, see our contributor guide.

Open a documentation issue

Provide product feedback

编译器错误 CS1900

项目 • 2023/11/20

警告等级必须在 0-4 的范围内

WarningLevel 编译器选项只能带五个可能的值（0、1、2、3 或 4）之一。 传递给 **/warn** 的其他任何值均将导致 CS1900。

下面的示例生成 CS1900：

C#

```
// CS1900.cs
// compile with: /W:5
// CS1900 expected
class x
{
   public static void Main()
   {
   }
}
```

备注

编译器不再生成此错误。 大于 4 的值表示“警告波”并有效，即使尚未为此波定义警告。

在 GitHub 上与我们协作

可以在 GitHub 上找到此内容的源，还可以在其中创建和查看问题和拉取请求。 有关详细信息，请参阅参与者指南。

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

提出文档问题

提供产品反馈

编译器错误 CS1902

项目 • 2023/04/07

/debug 的选项“option”无效；必须是 full 或 pdbonly

向 DebugType 编译器选项传递了无效的选项。

下面的示例生成 CS1902：

C#

```
// CS1902.cs
// compile with: /debug:x
// CS1902 expected
class x
{
   public static void Main()
   {
   }
}
```

编译器错误 CS1906

项目 • 2023/05/09

选项“option”无效；资源可见性必须是“public”或“private”

此错误表示无效的 Resources（将资源文件嵌入到输出文件）或 LinkResources（链接到.NET Framework 资源）命令行选项。 检查 **/resource** 或 **/linkresource** 命令行选项的语法，确保使用的可访问性修饰符为 **public** 或 `private`。

编译器错误 CS1908

项目 • 2023/10/08

DefaultParameterValue 属性的参数类型必须与以下参数类型匹配

将错误类型的值传递给 DefaultParameterValueAttribute时，将生成此错误。 确保属性参数的类型与目标参数的类型匹配。

示例

以下示例会生成 CS1908：

C#

```
// CS1908.cs
// compile with: /target:library
using System.Runtime.InteropServices;

public interface ISomeInterface
{
    void Bad([DefaultParameterValue("true")] bool b);   // CS1908
    void Good([DefaultParameterValue(true)] bool b);   // OK
}
```


Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

Open a documentation issue

Provide product feedback

编译器错误 CS1909

项目 • 2023/10/06

DefaultParameterValue 特性不适用于“type”类型的参数

当你使用不适用于参数类型的 DefaultParameterValueAttribute 特性时，会生成 CS1909。 此错误消息不再发出，有关详细信息，请参阅 CS1910。

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

Open a documentation issue

Provide product feedback

编译器错误 CS1910

项目 • 2023/10/06

“type”类型的参数不适用于 DefaultParameterValue 特性

对于类型为对象的形参， DefaultParameterValueAttribute 的实参必须是 `null`、整型、浮点型、 `bool`、 `string`、 `enum`或 `char`。 参数不能为 Type 类型或任何数组类型。

示例

下面的示例会生成 CS1910：

C#

```
// CS1910.cs
// compile with: /target:library
using System.Runtime.InteropServices;

public interface ISomeInterface
{
    void Bad1([DefaultParameterValue(typeof(object))] object o);   // CS1910
    void Bad2([DefaultParameterValue(new int[] { 1, 2 })] int[] arr);   //
CS1910
}
```


Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

Open a documentation issue

Provide product feedback

编译器错误 CS1912

项目 • 2023/04/07

成员“name”的初始化重复。

一个对象初始值设定项仅能对每个成员进行一次初始化。

更正此错误

1. 删除对象初始值设定项中对该成员的第二次初始化。

示例

下面的代码生成 CS1912，因为 `memberA` 被初始化了两次：

C#

```
// cs1912.cs
using System.Linq;

public class TestClass
{
    public int memberA { get; set; }
    public int memberB { get; set; }
}

public class Test
{
    static void Main()
    {
        TestClass tc = new TestClass() { memberA = 5, memberA = 6, memberB =
2}; // CS1912
    }
}
```

请参阅

- 对象和集合初始值设定项

编译器错误 CS1913

项目 • 2023/04/07

无法初始化成员“name”。 不是字段或属性。

对象初始值设定项仅用于初始化可访问的字段或属性。

更正此错误

1. 初始化正则构造函数或其他初始化方法中的类成员。

示例

下面的示例生成 CS1913：

C#

```
// cs1912.cs
class A
{
    public delegate void D();
    public event D myEvent;
}

public class Test
{
    static void Main()
    {

        A a = new A() {myEvent = M}; // CS1913
    }

    public void M(){}
}
```

请参阅

- C# 类型系统

编译器错误 CS1914

项目 • 2023/04/07

无法在对象初始值设定项中为静态字段“name”赋值

根据定义，对象初始值设定项初始化类的对象或实例。 它们不能用于初始化类型的 `static` 字段。 无论创建多少个类的实例， `static` 字段仅有一个副本。

更正此错误

1. 将字段更改为类型中的实例字段，或放弃从对象初始值设定项中初始化它的尝试。

示例

以下代码生成 CS1914，因为初始值设定项尝试初始化为 `TestClass.Number` 的 `static`字段：

C#

```
// cs1914.cs
using System.Linq;
public class TestClass
{
    public string Message { get; set; }
    public static int Number { get; set; }
}
class Test
{
    static void Main()
    {
        TestClass b = new TestClass() { Message = "Hello", Number = "555-
1212" }; // CS1914

    }
}
```

编译器错误 CS1917

项目 • 2023/04/07

无法使用对象初始值设定项为类型为“结构名”的只读字段“名称”的成员赋值，因为它是值类型

在构造函数中只能为值类型的只读字段赋值。

更正此错误

- 将结构更改为类类型。
- 用构造函数初始化结构。

示例

以下代码生成 CS1917:

C#

```
// cs1917.cs
class CS1917
{
    public struct TestStruct
    {
        public int i;
    }
    public class C
    {
        public readonly TestStruct str = new TestStruct();
        public static int Main()
        {
            C c = new C { str = { i = 1 } }; // CS1917
            return 0;
        }
    }
}
```

编译器错误 CS1918

项目 • 2023/04/07

无法使用对象初始值设定项为类型为“type”的属性“name”的成员赋值，因为它是值类型。

尝试使用对象初始值设定项初始化某种结构类型的属性，而该结构类型本身是所初始化的类的属性时，会发生此错误。

更正此错误

1. 如果必须在对象初始值设定项中完全初始化属性的字段，请将结构更改为类类型。否则，在使用对象初始值设定项创建对象之后，在单独方法调用中初始化结构成员。

示例

下面的示例生成 CS1918：

C#

```
// cs1918.cs
public struct MyStruct
{
    public int i;

}
public class Test
{
    private MyStruct str = new MyStruct();
    public MyStruct Str
    {
        get
        {
            return str;
        }
    }
    public static int Main()
    {
        Test t = new Test { Str = { i = 1 } }; // CS1918
        return 0;
    }
}
```

请参阅

- 对象和集合初始值设定项

编译器错误 CS1919

项目 • 2023/06/08

创建对象时不能使用不安全的类型“type name”。

`new` 运算符仅在托管堆上创建对象。 但可以通过使用语言的互操作性功能调用返回指针的本机方法，间接地在非托管内存中创建对象。

更正此错误

1. 在新的对象创建表达式中使用安全类型。 例如，使用 `char` 或 `int`，而不是使用 `char*` 或 `int*`。

2. 如果必须在非托管内存中创建对象，请使用 Win32 或 COM 方法，或者用 C 或 C++ 编写自己的函数并从 C# 调用该函数。

示例

下面的示例生成 CS1919，因为指针类型是不安全的：

C#

```
// cs1919.cs
// Compile with: /unsafe
unsafe public class C
{
    public static int Main()
    {
        var col1 = new int* { }; // CS1919
        var col2 = new char* { }; // CS1919
        return 1;
    }
}
```

另请参阅

- 互操作性

编译器错误 CS1922

项目 • 2023/04/07

集合初始值设定项需要其类型“type”实现 System.Collections.IEnumerable。

要通过类型集合初始值设定项，该类型必须实现 `IEnumerable`。 当你本想使用对象初始值设定项却意外使用集合初始值设定项语法时，可能出现此错误。

更正此错误

- 如果该类型不表示集合，请使用对象初始值设定项语法而不使用集合初始值设定项语法。
- 如果该类型确实表示集合，则先将其修改为实现 `IEnumerable` 才能使用集合初始值设定项来初始化该类型的对象。
- 如果该类型表示集合而你无权访问源代码，使用其元素的类构造函数或其他初始化方法对其元素进行初始化即可。

示例

下列代码生成 CS1922：

C#

```
// cs1922.cs
public class Test
{
    public static void Main()
    {
        // Collection initializer.
        var tc = new TestClass  {1,"hello"} ; // CS1922

        // Object initializer.
        var tc2 = new TestClass { memberA = 1, memberB = "hello" }; // OK
    }
}

public class TestClass
{
    public int memberA { get; set; }
    public string memberB { get; set; }
}
```

请参阅

- 对象和集合初始值设定项

编译器错误 CS1926

项目 • 2023/04/07

读取 Win32 清单文件“filename”时出错 --“error”。

如果满足以下条件，则会生成此错误：

1. 在命名行上或通过以下方法指定了 /win32manifest 选项：右击“解决方案资源管理器”中的“项目”图标，指向“添加”，单击“新建项”，然后单击“应用程序清单文件”。

2. 该文件损坏或缺失。

更正此错误

1. 删除该选项。

2. 替换、修复或重新生成该文件。

示例

使用损坏或缺失的 win32 清单文件编译下面的示例时，会生成 CS1926：

C#

```
// cs1926.cs
// Compile with: /win32manifest: ../../app.manifest
// CS1926
class Test
{
    public static int Main()
    {
        return 1;
    }
}
```

另请参阅

- Win32Manifest（C# 编译器选项）

编译器错误 CS1930

项目 • 2023/04/07

已声明范围变量“name”

查询表达式中的范围变量会一直位于范围内，直到该查询表达式终止。 因此，该变量必须具有唯一标识符。

更正此错误

1. 为在查询表达式中引入的每个范围变量指定一个唯一名称。

示例

下面的示例生成 CS1930，因为同时对 `num` 子句中的范围变量和 `from` 子句所引入的范围变量使用了标识符 `let` 。

C#

```
// cs1930.cs
using System.Linq;
class Program
{
    static void Main()
    {
        int[] nums = { 0, 1, 2, 3, 4, 5 };
        var query = from num in nums
                    let num = 3 // CS1930
                    select num;
    }
}
```

请参阅

- LINQ 查询表达式

编译器错误 CS1931

项目 • 2023/04/07

范围变量“variable”与“variable”以前的声明存在冲突。

正如其他各个声明一样，范围变量的声明必须具有在变量声明空间中唯一的标识符。

更正此错误

1. 为范围变量提供唯一名称。

示例

下面的代码生成 CS1931，因为该标识符 `x` 既用作 `Main` 中的本地变量，也用作查询表达式中的范围变量：

C#

```
// cs1931.cs
class Test
    {
        static void Main()
        {
            int x = 1;
            var y = from x in Enumerable.Range(1, 100) // CS1931
                    select x;
        }
    }
```

请参阅

- LINQ 查询表达式

编译器错误 CS1932

项目 • 2023/04/07

无法将“expression”赋值给范围变量

编译器必须能够推断出的某个范围变量的类型，不管其是在 `from` 子句中引入还是在 `let` 子句中引入。 其不能为 null，因为 null 不是一种类型，而且不能使用不安全类型的表达式对其赋值。

更正此错误

- 删除无效的赋值。
- 将此表达式显式转换为允许的类型

示例

下面的代码会生成 CS1932，因为不能推断范围变量的类型。 将值转换为预期类型以修复此错误，如下面的示例所示。

C#

```
// CS1932.cs
using System.Linq;
class Test
{
    static void Main()
    {

        var x = from i in Enumerable.Range(1, 100)
                let k = null // CS1932
                // Try the following line instead.
                let k = (string) null
                select i;
    }
}
```

请参阅

- LINQ 查询表达式

编译器错误 CS1933

项目 • 2023/04/07

表达式不能包含查询表达式

某些变量无法使用查询表达式进行初始化。 常数无法使用查询表达式进行初始化，因为常数只能使用文本、命名常数和数学运算符的某种组合进行初始化。

更正此错误

1. 从查询变量中删除修饰符。

示例

下面的示例生成 CS1933：

C#

```
// cs1933.cs
using System.Linq;
using System.Collections;

class Program
{
    const IEnumerable e = from x in new[] { 1, 2, 3 } select x; // CS1933
    static int Main()
    {
        return 1;
    }
}
```

备注

Roslyn 中不再使用此编译器错误。 使用 Roslyn 编译时，上一个示例生成 CS0133。

另请参阅

- C# 中的 LINQ

编译器错误 CS1934

项目 • 2023/04/07

未能找到源类型“type”的查询模式的实现。 找不到“method”。 请考虑显式指定范围变量“name”的类型。

如果查询表达式指定了一个数据源，而没有标准查询运算符对其进行实现，则会生成此错误。 指定了 `ArrayList` ，但没有为该范围变量给定显式类型，也会产生此错误。

更正此错误

1. 在以下示例中，该解决方案将只指定范围变量的类型：

```
C#
```

```
var q = from int x in list
```

示例

以下示例演示了生成 CS1934 的一种方式：

```
C#
```

```
// cs1934.cs
using System.Linq;
using System.Collections;
static class Test
{
    public static void Main()
    {
        var list = new ArrayList { 0, 1, 2, 3, 4, 5 };
        var q = from x in list // CS1934
                select x + 1;
    }
}
```

另请参阅

- 如何使用 LINQ 查询 ArrayList

编译器错误 CS1935

项目 • 2023/04/07

未能找到源类型“type”的查询模式的实现。 找不到“method”。 “System.Linq”是否缺少 using 指令?

在查询中的源类型必须是 `IEnumerable`、 `IEnumerable<T>`或派生的类型，或者你或其他人已为其实现了标准查询运算符的类型。 如果源类型为 `IEnumerable` 或 `IEnumerable<T>`，必须添加一个用于 `System.Linq` 命名空间的 `using` 指令，以便将标准查询运算符扩展方法引入作用域。 必须以同样方式，使用 `using` 指令，以及（如有必要）使用对程序集的引用将标准查询运算符的自定义实现引入作用域。

更正此错误

添加所需的 using 指令和对项目的引用。

示例

以下代码生成 CS1935，因为用于 System.Linq 的 `using` 指令被注释掉：

C#

```
// cs1935.cs
// CS1935
using System;
using System.Collections.Generic;
// using System.Linq;

class Test
{
    static int Main()
    {
        int[] nums = { 0,1,2,3,4,5 };
        IEnumerable<int> e = from n in nums
                             where n > 3
                             select n;
        return 0;
    }
}
```

另请参阅

- 标准查询运算符概述

编译器错误 CS1936

项目 • 2023/04/07

未能找到源类型“type”的查询模式的实现。 找不到“method”。

若要查询源类型，该类型必须实现你要在查询中调用的标准查询运算符方法。 实现可以是类成员的形式，也可以是使用相应的 `using` 指令纳入作用域的扩展方法。

更正此错误

- 请确保要查询的是对象集合，而不是单个对象。
- 请确保已指定必需的 `using` 指令。

示例

下面的示例生成 CS1936：

C#

```
// cs1936.cs
using System.Collections;
using System.Linq;
class Test
{
    static int Main()
    {
        object obj;
        IEnumerable e = from x in obj // CS1936
                        select x;
        return 0;
    }
}
```

无意中尝试查询某种类型的对象（而不是这些对象的集合）时，经常发生此错误。

另请参阅

- 标准查询运算符概述

编译器错误 CS1937

项目 • 2023/04/07

名称“name”不在“等号”左侧的作用域中。 请考虑交换位于“等号”任意一侧的表达式。

`equals` 关键字是一种特殊运算符，其在 `join` 子句中用于确定两个表达式之间是否相等。 左侧源序列的范围变量在等号左侧的作用域中，右侧源的范围变量仅存在于等号左侧的作用域中。 可以通过以下代码示例中的 IntelliSense 进行实验，以对此进行验证。

更正此错误

1. 交换两个范围变量的位置，如以下示例中的注释行所示：

示例

以下示例生成 CS1937。

C#

```
// cs1937.cs
using System.Linq;
class Test
{
    static void Main()
    {
        int[] sourceA = { 1, 2, 3, 4, 5 };
        int[] sourceB = { 3, 4, 5, 6, 7 };

        var query = from a in sourceA
                    join b in sourceB on b equals a // CS1937
                    // Try the following line instead.
                    //join b in sourceB on a equals b
                    select new { a, b };
    }
}
```

左侧通常称为“外部”端，而右侧通常称为“内部”端。

另请参阅

- join 子句

编译器错误 CS1938

项目 • 2023/04/07

名称“name”不在“等号”右侧的作用域中。 请考虑交换位于“等号”任意一侧的表达式。

`equals` 关键字是一种特殊运算符，其在 `join` 子句中用于确定两个表达式之间是否相等。 左侧源序列的范围变量在等号左侧的作用域中，右侧源的范围变量仅存在于等号左侧的作用域中。 可以通过以下代码示例中的 IntelliSense 进行实验，以对此进行验证。

更正此错误

1. 交换两个范围变量的位置，如以下示例中的注释行所示：

示例

以下代码生成 CS1938：

C#

```
// cs1938.cs
using System.Linq;
class Test
{
    static void Main()
    {
        int[] sourceA = { 1, 2, 3, 4, 5 };
        int[] sourceB = { 3, 4, 5, 6, 7 };

        var query = from a in sourceA
                    join b in sourceB on b equals a // CS1938
                    // Try the following line instead.
                    // join b in sourceB on a equals b
                    select new { a, b };
    }
}
```

请参阅

- join 子句

编译器错误 CS1940

项目 • 2023/04/07

找到源类型“type”的多个查询模式实现。 对“method”的调用不明确。

当定义了查询方法的多个实现，而编译器无法确定最适合的查询时，则会生成此错误。 在下面的示例中， `Select` 的两个版本具有相同的签名，因为它们都接受同一 `int` 作为输入参数，并且返回值均为 `int` 。

更正此错误

1. 仅为每个方法提供一个实现。

示例

以下代码生成 CS1940：

C#

```
// cs1940.cs
using System; //must include explicitly for types defined in 3.5
class Test
{
    public delegate int Dele(int x);
    int num = 0;
    public int Select(Func<int, int> d)
    {
        return d(this.num);
    }
    public int Select(Dele d) // CS1940
    {
        return d(this.num) + 1;
    }
    public static void Main()
    {
        var q = from x in new Test()
        select x;
    }
}
```

另请参阅

- 标准查询运算符概述

编译器错误 CS1941

项目 • 2023/04/07

“clause”子句中的某个表达式的类型不正确。 对“method”的调用中，类型推理失败。

查询表达式中的类型推理源于数据源中的元素类型。

更正此错误

1. 如果发生此错误的原因不十分明显，请仔细检查查询，并跟踪从数据源到错误发生点的每个子句的结果的类型。

示例

下面的代码生成 CS1941，因为将 `int` 与 `string` 比较时需要 `equals` 运算符。

C#

```
// cs1941.cs
using System.Collections;
using System.Linq;
class Test
{
    static int Main()
    {
        var nums = new[] { 1, 2, 3, 4, 5, 6 };
        var words = new string[] { "lake", "mountain", "sky" };
        IEnumerable e = from n in nums
                        join w in words on n equals w // CS1941
                        select w;
        return 0;
    }
}
```

类型推理失败的方法是系统在编译时将查询子句转换为的方法。

另请参阅

- C# 中的 LINQ
- LINQ 查询操作中的类型关系

编译器错误 CS1942

项目 • 2023/04/07

“clause”子句中表达式的类型不正确。 对“method”的调用中，类型推理失败。

如果为范围变量指定了不正确的显式类型，通常会生成此错误。

更正此错误

1. 如果范围变量是显式类型的，请确保该类型与变量循环访问的集合中的元素类型相同，或者可从后者隐式转换。 如果范围变量前面带有 `var` 关键字，请删除 `var`。

示例

下面的代码生成 CS1942：

C#

```
// cs1942.cs
class Program
    {
        static void Main(string[] args)
        {
            var x = from var i in Enumerable.Range(1, 100) // CS1949
                    select i; //CS1942
        }
    }
```

CS1942 与 CS1949 相关，因为对范围变量使用 `var` 会导致因 `var` 不是类型而使基础 `Cast<T>` 运算失败。

另请参阅

- var
- C# 中的 LINQ

编译器错误 CS1943

项目 • 2023/04/07

“type”类型的表达式不允许存在于“type”源类型的查询表达式中的后续 from 子句中。 对“method”的调用中，类型推理失败。

所有范围变量都必须表示可查询类型。

更正此错误

1. 请确保该类型是实现 `IEnumerable`、 `IEnumerable<T>` 或派生接口的可查询类型，或具有为之定义的查询模式的任何其他类型。
2. 如果类型是非泛型 `IEnumerable`，则提供范围变量的显式类型。

示例

下面的代码生成 CS1943：

C#

```
// cs1943.cs
using System.Linq;
class Test
{
    class TestClass
    { }
    static void Main()
    {
        int[] nums = { 0, 1, 2, 3, 4, 5 };
        TestClass tc = new TestClass();

        var x = from n in nums
                from s in tc // CS1943
                select n + s;
    }
}
```

编译器错误 CS1947

项目 • 2023/04/07

无法对范围变量“变量名称”赋值 -- 它是只读的。

在 `foreach` 语句中，范围变量类似于迭代变量。 查询表达式中不能对其赋值。

更正此错误

1. 删除对范围变量的赋值。

2. 如有必要，可通过使用 `let` 子句引入新的范围变量并用其存储值。

示例

以下代码生成 CS1947：

C#

```
// cs1947.cs
using System.Linq;
class Test
{
    static void Main()
    {
        int[] array = new int[] { 1, 2, 3, 4, 5 };
        var x = from i in array
                let k = i
                select i = 5; // CS1947
        x.ToList();
    }
}
```

请参阅

- LINQ 查询表达式

编译器错误 CS1948

项目 • 2023/04/07

范围变量“name”不能与方法类型参数同名

同一声明空间不能包含对同一标识符的两个声明。

更正此错误

1. 更改范围变量或类型参数的名称。

示例

下面的示例将生成 CS1948，因为同时对方法 `T` 的范围变量和类型参数使用了标识符 `TestMethod`：

C#

```
// cs1948.cs
using System.Linq;
class Test
{
    public void TestMethod<T>(T t)
    {
        var x = from T in Enumerable.Range(1, 100) // CS1948
                select T;
    }
}
```

编译器错误 CS1949

项目 • 2023/04/07

不能在范围变量声明中使用上下文关键字“var”。

编译器隐式类型化范围变量。 无需通过范围变量使用 var 。

更正此错误

1. 删除范围变量前面的 `var` 关键字。

示例

下面的示例生成 CS1949：

C#

```
// cs1949.cs
using System;
using System.Linq;
class Test
{
    static void Main()
    {
        var x = from var i in Enumerable.Range(1, 100) // CS1949
        select i;
    }
}
```

请参阅

- LINQ 查询表达式
- LINQ 查询简介 (C#)

编译器错误 CS1955

项目 • 2023/04/07

不可调用的成员“name”不能像方法一样使用。

只能调用方法和委托。 如果尝试使用空括号调用除方法或委托之外的其他内容，则将生成此错误。

更正此错误

1. 从表达式中删除括号。

示例

下面的代码生成 CS1955，因为该代码尝试使用调用表达式 () 来调用字段和属性。 不能调用字段或属性。 使用成员访问表达式 . 访问其存储的值。

C#

```
// cs1955.cs
class A
{
    public int x = 0;
    public int X
    {
        get { return x; }
        set { x = value; }
    }
}

class Test
{
    static int Main()
    {
        A a = new A();
        a.x(); // CS1955
        a.X(); // CS1955
        // Try this line instead:
        // int num = a.x;
    }
}
```

编译器错误 CS1958

项目 • 2023/04/07

对象和集合初始值设定项表达式不能应用于委托创建表达式。

与类或结构不同，委托不具有任何成员，因此对象初始值设定项没有要进行初始化的内容 如果遇到此错误，则可能是因为委托创建表达式后面有大括号。 只需删除大括号，此错误就会消失。

更正此错误

1. 删除大括号。

示例

下面的代码生成 CS1958：

C#

```
// cs1958.cs
public class MemberInitializerTest
{
    delegate void D<T>();
    public static void GenericMethod<T>() { }
    public static void Run()
    {
        D<int> genD = new D<int>(GenericMethod<int>) { }; // CS1958
       // Try the following line instead
      // D<int> genD = new D<int>(GenericMethod<int>);
    }
}
```

编译器错误 CS1959

项目 • 2023/04/07

“name”属于类型“type”。 在常量声明中指定的类型必须为 sbyte、byte、short、ushort、int、uint、long、ulong、char、float、double、decimal、bool、string、枚举类型或引用类型。

常数声明中允许的类型仅限于此消息中描述的类型。

更正此错误

1. 声明具有允许类型的常量。

示例

以下代码会产生 CS1959，因为 `null` 不是类型。

C#

```
// cs1959.cs
class Program
    {
        static void Test<T>() where T : class
        {
            const T x = null; // CS1959
        }
    }
```

另请参阅

- 常量
- null

编译器错误 CS1983

项目 • 2023/12/06

异步方法的返回类型必须为 void、Task、Task<T>、类似任务的类型（类似任务的类型是正如此处所述，符合 C# 规范要求的模式的类型）、IAsyncEnumerable<T>，或 IAsyncEnumerator<T>

示例

下面的示例生成 CS1983：

C#

```
// CS1983.cs (4,62)
using System.Collections.Generic;

class C
{
    static async IEnumerable<int> M()
    {
        yield return await Task.FromResult(1);
    }
}
```

由于 `IEnumerable.GetEnumerator` 返回一个 `IEnumerator<T>`，其 `MoveNext` 方法不返回值 `Task<T>`，因此它与 `async` 方法不兼容。

更正此错误

使用导致调用返回类型 `Task<T>` 的方法的接口，例如 `IAsyncEnumerable<T>`：

C#

```
using System.Collections.Generic;

class C
{
    static async IAsyncEnumerable<int> M()
    {
        yield return await Task.FromResult(1);
    }
}
```

在 GitHub 上与我们协作

可以在 GitHub 上找到此内容的源，还可以在其中创建和查看问题和拉取请求。 有关详细信息，请参阅参与者指南。

.NET 反馈

.NET 是一个开放源代码项目。 选择一个链接以提供反馈：

提出文档问题

提供产品反馈

编译器错误 CS1986

项目 • 2023/06/08

“await”要求类型包含适当的“GetAwaiter”方法

示例

下面的示例生成 CS1986：

C#

```
using System.Runtime.CompilerServices;
using System;
using System.Threading.Tasks;

class Program
{
    static async Task M(MyTask<int> x)
    {
        var z = await x;
        System.Console.WriteLine(z);
    }
}

public class MyTask<TResult>
{
    readonly MyTaskAwaiter<TResult> awaiter;
    public MyTask(TResult value)
    {
        this.awaiter = new MyTaskAwaiter<TResult>(value);
    }
    public static MyTaskAwaiter<TResult> GetAwaiter() => throw new
NotImplementedException();
}

public class MyTaskAwaiter<TResult> : INotifyCompletion
{
    TResult value;
    public MyTaskAwaiter(TResult value)
    {
        this.value = value;
    }
    public bool IsCompleted { get => true; }
    public TResult GetResult() => value;
    public void OnCompleted(Action continuation) => throw new
NotImplementedException();
}
```

`GetAwaiter` 方法必须是一个名为 `GetAwaiter` 的非静态方法，并返回实现 `INotifyCompletion` 的对象实例。

GetAwaiter 需要实现 `INotifyCompletion` 接口（以及可选的 `ICriticalNotifyCompletion` 接口），并返回一个本身公开三个成员的类型 [1]：

C#

```
bool IsCompleted { get; }
void OnCompleted(Action continuation);
TResult GetResult(); // TResult can also be void
```

更正此错误

示例中引发 CS1986 的原因是 `GetAwaiter` 方法是 `static`。 若要更正此错误，请删除 `static` 修饰符（并正确实现该方法）：

C#

```
public class MyTask<TResult>
{
    readonly MyTaskAwaiter<TResult> awaiter;
    public MyTask(TResult value)
    {
        this.awaiter = new MyTaskAwaiter<TResult>(value);
    }
    public MyTaskAwaiter<TResult> GetAwaiter() => awaiter;
}
```

另请参阅

[1] await anything;（等待任何内容；）

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

Open a documentation issue

Provide product feedback

more information, see our contributor guide.

编译器错误 CS1994

项目 • 2023/04/07

只能在具有主体的方法中使用“async”修饰符。

示例

下面的示例生成 CS1994：

C#

```
interface IInterface
{
    async void F();
}
```

更正此错误

interface 声明中的非具体方法声明没有方法主体。 为了支持 `async` 修饰符，编译器将方法主体逻辑归入状态机。 如果没有方法主体，编译器将无法发出此状态机。 此外，方法主体的逻辑必须包含 await 运算符，以表示状态机必须管理的延续。 如果没有 `await` 运算符，状态机就没有可管理的内容。

对于非具体方法，如果将方法主体的实现延迟到实现 interface 的类，则只需删除 async 修饰符即可更正错误：

C#

```
interface IInterface
{
    void F();
}
```

或者，可以在 interface 中声明具体的默认方法（在 C#8.0 中引入）：

C#

```
interface IInterface
{
    async void F()
    {
        await Task.Run(() =>
        {
```

```
            /* do something useful*/
        });
    }
}
```

编译器错误 CS1996

项目 • 2024/07/12

无法在 lock 语句体中等待

示例

下面的示例生成 CS1996：

```
C#
```

```
public class C
{
    private readonly Dictionary<string, string> keyValuePairs = new();

    public async Task<string> ReplaceValueAsync(string key, HttpClient
httpClient)
    {
        lock (keyValuePairs)
        {
            var newValue = await httpClient.GetStringAsync(string.Empty);
            if (keyValuePairs.ContainsKey(key)) keyValuePairs[key] =
newValue;
            else keyValuePairs.Add(key, newValue);
            return newValue;
        }
    }
}
```

前面的代码生成与 C# 13 相同的错误，因为 `await` 位于 `lock` 语句块中。

更正此错误

`lock` 语句块中的异步代码很难可靠地实现，而且在一般意义上更难实现。 C# 编译器不支持执行此操作，以避免发出容易出现死锁的代码。 从 `lock` 语句块中提取异步代码可更正此错误。 例如：

```
C#
```

```
public class C
{
    private readonly Dictionary<string, string> keyValuePairs = new();

    public async Task<string> ReplaceValueAsync(string key, HttpClient
httpClient)
```

```
    {
        var newValue = await httpClient.GetStringAsync(string.Empty);
        lock (keyValuePairs)
        {
            if (keyValuePairs.ContainsKey(key)) keyValuePairs[key] =
newValue;
            else keyValuePairs.Add(key, newValue);
            return newValue;
        }
    }
}
```

在 GitHub 上与我们协作

可以在 GitHub 上找到此内容的源，还可以在其中创建和查看问题和拉取请求。 有关详细信息，请参阅参与者指南。

.NET 反馈

.NET 是一个开放源代码项目。 选择一个链接以提供反馈：

- 提出文档问题
- 提供产品反馈

编译器错误 CS1997

项目 • 2024/08/19

由于 `method` 是返回 `Task` 的异步方法，因此 return 关键字后面不能跟对象表达式。 是否打算返回 `Task<T>`？

示例

下面的示例生成 CS1997：

C#

```
using System.Threading.Tasks;
class C
{
    public static async Task F1()
    {
        return await Task.Factory.StartNew(() => 1);
    }
}
```

更正此错误

`async` 方法中的 `return` 语句返回可等待语句的结果。 如果可等到语句没有结果，则编译器发出的状态机封装返回非泛型 `Task`，无需语句 `return`。 遇到错误 CS1997 意味着引用的代码包含与 `async` 修饰符和方法的 `return` 类型冲突的 `return` 语句。 该错误指示当前方法的实现与其初始意向不一致。 更正错误的最简单方法是删除 `return` 语句：

C#

```
    public static async Task F1()
    {
        await Task.Factory.StartNew(() => 1);
    }
```

但是，生成的实现不再需要 `async` 修饰符或 `await` 运算符。 更正此错误的一种更准确方法是不删除 `return` 语句，而是删除 `async` 修饰符和 `await` 运算符：

C#

```
    public static Task F1()
    {
```

```
        return Task.Factory.StartNew(() => 1);
    }
```

编译器错误 CS2001

项目 • 2024/04/11

找不到源文件“file”

向编译器传递了源文件名，但未能找到该文件。 检查文件名的拼写和文件的位置。

在 GitHub 上与我们协作

可以在 GitHub 上找到此内容的源，还可以在其中创建和查看问题和拉取请求。 有关详细信息，请参阅参与者指南。

.NET

.NET 反馈

.NET 是一个开放源代码项目。 选择一个链接以提供反馈：

提出文档问题

提供产品反馈

编译器错误 CS2003

项目 • 2023/04/07

已多次包含响应文件“file”

已多次将响应文件传递到编译器。 每个输出文件只能将响应文件传递到编译器一次。

有关响应文件的详细信息，请参阅 ResponseFiles（指定响应文件）。

编译器错误 CS2005

项目 • 2023/04/07

“选项”选项缺少文件规范

只指定了部分 编译器选项 。

例如，使用 `-recurse` 时，必须指定要搜索的文件：/recurse:filename.cs。

示例

下面的示例生成 CS2005。

C#

```
// CS2005.cs
// compile with: /recurse:
// CS2005 expected
class x
{
   public static void Main() {}
}
```

编译器错误 CS2006

项目 • 2023/04/07

命令行语法错误：“option”选项缺少“text”

option 的语法需要额外的文本。 有关信息，请参阅 编译器选项。

编译器错误 CS2007

项目 • 2023/04/07

无法识别的命令行选项:“option”

向编译器传递了不是 编译器选项的字符串，即使它以正斜杠 (/) 开头。

下面的示例生成 CS2007：

C#

```
// CS2007.cs
// compile with: /recur
// CS2007 expected
class x
{
   public static void Main() {}
}
```

编译器错误 CS2008

项目 • 2024/04/11

未指定输入

已调用编译器并且已指定编译器选项，但未传递任何源代码文件。

在 GitHub 上与我们协作

可以在 GitHub 上找到此内容的源，还可以在其中创建和查看问题和拉取请求。 有关详细信息，请参阅参与者指南。

.NET 反馈

.NET 是一个开放源代码项目。 选择一个链接以提供反馈：

提出文档问题

提供产品反馈

编译器错误 CS2011

项目 • 2023/04/07

无法打开响应文件“file”

已在编译中指定响应文件，但编译器无法找到和打开此文件。

有关响应文件的详细信息，请参阅 ResponseFiles（指定响应文件）。

编译器错误 CS2012

项目 • 2024/02/20

无法打开“file”进行写入

当使用 `-bugreport:`、`file` 编译器选项时，未能打开文件进行写入。 请确保指定的文件名有效，并且该文件不是只读文件。 此错误的一个可能原因是文件在另一个进程（例如调试器或防病毒程序）中使用。

> **重要**
>
> 不再支持 `bugreport` 选项。

在 GitHub 上与我们协作

可以在 GitHub 上找到此内容的源，还可以在其中创建和查看问题和拉取请求。 有关详细信息，请参阅参与者指南。

.NET 反馈

.NET 是一个开放源代码项目。 选择一个链接以提供反馈：

提出文档问题

提供产品反馈

编译器错误 CS2013

项目 • 2023/04/07

图像基数“value”无效

向 BaseAddress 编译器选项传递了无效的值（非数字）。

以下示例生成 CS2013：

C#

```
// CS2013.cs
// compile with: /target:library /baseaddress:x
// CS2013 expected
class MyClass
{
}
```

编译器错误 CS2015

项目 • 2023/04/07

“file”是二进制文件而非文本文件

已将文件传递到属于二进制文件的编译器。 编译器需要一个源代码文件。

编译器错误 CS2016

项目 • 2023/04/07

代码页“codepage”无效或未安装

向 CodePage 编译器选项传递了无效值。

下面的示例生成 CS2016:

C#

```
// CS2016.cs
// compile with: /codepage:x
// CS2016 expected
class MyClass
{
   public static void Main()
   {
   }
}
```

编译器错误 CS2017

项目 • 2024/07/04

如果生成模块或库，则无法指定 /main

在生成 **library** TargetType 时，不能指定主入口点。

以下示例生成 CS2017：

C#

```
// CS2017.cs
// compile with: /main:MyClass /target:library
// CS2017 expected
class MyClass
{
   public static void Main()
   {
   }
}
```


在 GitHub 上与我们协作

可以在 GitHub 上找到此内容的源，还可以在其中创建和查看问题和拉取请求。 有关详细信息，请参阅参与者指南。

.NET

.NET 反馈

.NET 是一个开放源代码项目。 选择一个链接以提供反馈：

提出文档问题

提供产品反馈

编译器错误 CS2018

项目 • 2023/04/07

无法找到消息文件“cscmsgs.dll”

未找到包含编译器的错误和警告消息的 .dll 文件。 此文件必须位于与另一个编译器支持文件相同的目录中。

编译器错误 CS2019

项目 • 2023/04/07

/target 的目标类型无效: 必须指定“exe”、“winexe”、“library”或“module”

/TargetType 编译器选项已使用，但传递的是无效参数。 若要解决此错误，请使用适合输出文件的 **/target** 选项的形式重新编译程序。

以下示例生成 CS2017：

C#

```
// CS2019.cs
// compile with: /target:libra
// CS2019 expected
class MyClass
{
}
```

编译器错误 CS2020

项目 • 2023/04/07

只有第一组输入文件能生成非“模块”的目标

在多输出编译中，第一个输出文件必须使用 -target:exe、-target:winexe 或 -target:library 生成。 所有后续输出文件必须使用 -target:module 生成。

编译器错误 CS2021

项目 • 2023/04/07

文件名“file”太长或无效

传递给 C# 编译器的所有文件名的长度均不得超过 `_MAX_PATH` （在 Windows 头文件中定义）。 在以下情况下，编译器将产生此错误：

- 文件名（包括路径）的长度超过 `_MAX_PATH`。
- 文件名包含无效字符。
- 文件名包含此位置不允许的通配符（例如，在资源文件的名称中）。

编译器错误 CS2022

项目 • 2023/04/07

选项“/out”和“/target”必须出现在源文件名之前

当在命令行中指定 -out（设置输出文件名）和 -target（指定输出文件格式）编译器选项时，它们必须位于源代码文件之前。

编译器错误 CS2024

项目 • 2023/04/07

文件节的对齐编号“#”无效

向 FileAlignment 编译器选项传递的值无效。

示例

下面的示例生成 CS2024：

C#

```
// CS2024.cs
// compile with: /filealign:ex
// CS2024 expected
class MyClass
{
    public static void Main()
    {
    }
}
```

编译器错误 CS2032

项目 • 2023/04/07

在命令行上或响应文件中不允许有字符“character”

csc.exe 的响应文件和命令行选项不允许包含 0-31 范围内的 ASCII 码控制字符或竖线 (`|`) 字符。

编译器错误 CS2032 难以从命令行演示，因为命令行处理器和集成开发环境 (IDE) 将筛选出无效的字符。 下面的过程使用响应文件生成错误。

生成此错误的方法

1. 在 My Documents 文件夹中，创建一个名为 CS2032.rsp 的文本文件，然后在其中输入以下编译器选项：

   控制台

   ```
   /target:exe /out:cs|2032.exe cs2032.cs
   ```

2. 在“My Documents”文件夹中，创建一个名为“cs2032.cs”的文本文件，然后在其中输入所需的任何内容。

3. 打开 Visual Studio 开发人员命令提示或 Visual Studio 开发人员 PowerShell。

4. 将当前目录更改为 `C:\Users\\<YourUsername>\Documents`。

5. 从命令提示符处运行以下命令： `csc @cs2032.rsp`

6. 由于响应文件 CS2032.rsp 包含竖线字符，因此会出现 CS2032 错误消息。

编译器错误 CS2033

项目 • 2023/04/07

包含短文件名“filename”的长文件名已存在，无法创建同名短文件名

编译名称超过八个字符的任何 C# 文件。 然后，编译另一个具有前面的文件名称的较短形式的文件，例如名称的前六个字符再加上“~1”。第二个编译将生成此错误。

若要解决此错误，请将较短文件名重命名为一个与长文件名不冲突的文件名。

编译器错误 CS2034

项目 • 2023/04/07

声明外部别名的 /reference 选项只能有一个文件名。 若要指定多个别名或文件名，请使用多个 /reference 选项。

若要指定两个别名和/或文件名，使用两个 **/reference** 选项，如下所示：

示例

下面的代码将生成错误 CS2034。

C#

```
// CS2034.cs
// compile with: /r:A1=cs2034a1.dll;A2=cs2034a2.dll
// to fix, compile with: /r:A1=cs2034a1.dll /r:A2=cs2034a2.dll
// CS2034
extern alias A1;
extern alias A2;
using System;
```

编译器错误 CS2035

项目 • 2023/04/07

命令行语法错误：“compiler_option”选项缺少“:<number>”

某些编译器选项需要一个值。

示例

下面的示例生成 CS2035。

C#

```
// CS2035.cs
// compile with: /baseaddress
// CS2035 expected
```

编译器错误 CS2036

项目 • 2023/04/07

要使用 /pdb 选项，必须同时也使用 /debug 选项。

仅为调试版本生成程序数据库文件。 因此， **/pdb** 选项在零售版本中没有意义。

更正此错误

- 添加 **/debug** 编译器选项。
- 删除 **/pdb** 编译器选项。

示例

当使用 **/pdb** 选项但不使用 /debug 选项对其进行编译时，下面的示例生成 CS2036：

C#

```
// cs2036.cs
// Compile with: /pdb
// CS2036
class Test
{
    public static int Main()
    {
        return 1;
    }
}
```

请参阅

- PdbFile（C# 编译器选项）
- DebugType（C# 编译器选项）

编译器错误 CS4004

项目 • 2024/07/04

无法在不安全的上下文中等待

示例

下面的示例生成 CS4004：

C#

```
using System.Threading.Tasks;

public static class C
{
    public static unsafe async Task<string> ReverseTextAsync(string text)
    {
        return await Task.Run(() =>
        {
            if (string.IsNullOrEmpty(text))
            {
                return text;
            }

            fixed (char* pText = text)
            {
                char* pStart = pText;
                char* pEnd = pText + text.Length - 1;
                for (int i = text.Length / 2; i >= 0; i--)
                {
                    char temp = *pStart;
                    *pStart++ = *pEnd;
                    *pEnd-- = temp;
                }

                return text;
            }
        });
    }
}
```

此代码在 C# 13 中会产生错误，因为 `await` 位于 `unsafe` 块中。

`ReverseText` 方法天真地使用后台任务按给定字符串的反向顺序异步创建新字符串。

更正此错误

将不安全代码与可等待代码分开会更正此错误。 一种分离技术是为不安全代码创建新方法，然后从可等待代码中进行调用。 例如：

C#

```
public static class C
{
    public static async Task<string> ReverseTextAsync(string text)
    {
        return await Task.Run(() => ReverseTextUnsafe(text));
    }

    private static unsafe string ReverseTextUnsafe(string text)
    {
        if (string.IsNullOrEmpty(text))
        {
            return text;
        }

        fixed (char* pText = text)
        {
            char* pStart = pText;
            char* pEnd = pText + text.Length - 1;
            for (int i = text.Length / 2; i >= 0; i--)
            {
                char temp = *pStart;
                *pStart++ = *pEnd;
                *pEnd-- = temp;
            }

            return text;
        }
    }
}
```


在 GitHub 上与我们协作

可以在 GitHub 上找到此内容的源，还可以在其中创建和查看问题和拉取请求。 有关详细信息，请参阅参与者指南。

.NET

.NET 反馈

.NET 是一个开放源代码项目。 选择一个链接以提供反馈：

提出文档问题

提供产品反馈

编译器错误 CS4008

项目 • 2023/06/08

无法等待“void”

示例

下面的示例生成 CS4008：

```C#
// CS4008.cs (7,33)

using System.Threading.Tasks;

class Test
{
    public async void goo()
    {
        await Task.Factory.StartNew(() => { });
    }

    public async void bar()
    {
        await goo();
    }

    public static void Main() { }
}
```

更正此错误

尽管可以通过更改 `goo` 的签名来更正此错误：

```C#
    public async Task goo()
    {
        await Task.Factory.StartNew(() => { });
    }
```

只需将 `Task` 添加到方法的签名中，就可以在不需要时永久保存编译器创建的状态机。 `goo` 方法不需要 `await`，也不需要是异步的。 相反，考虑仅返回由 `Task.Factory` 创建的 `Task`：

C#

```
    public Task goo()
    {
        return Task.Factory.StartNew(() => { });
    }
```

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET feedback

The .NET documentation is open source. Provide feedback here.

Open a documentation issue

Provide product feedback

编译器错误 CS4009

项目 • 2023/04/06

"Type.Method"：入口点不能用 `async` 修饰符来标记。

不能在应用程序入口点中（通常是 `Main` 方法）使用 `async` 关键字。

> **ⓘ 重要**
>
> 从 C# 7.1 开始，`Main` 方法可以具有 `async` 修饰符。 有关详细信息，请参阅**异步主返回值**。 有关如何选择 C# 语言版本的信息，请参阅**选择 C# 语言版本**一文。

示例

下面的示例生成 CS4009：

C#

```
using System;
using System.Threading.Tasks;

public class Example
{
    public static async void Main()
    {
        Console.WriteLine("About to wait two seconds");
        await WaitTwoSeconds();
        Console.WriteLine("About to exit the program");
    }

    private static async Task WaitTwoSeconds()
    {
        await Task.Delay(2000);
        Console.WriteLine("Returning from an asynchronous method");
    }
}
```

更正此错误

将项目使用的 C# 语言版本更新为 7.1 或更高版本。

如果使用 C# 7.0 或更低版本，请从应用程序入口点的签名中删除 `async` 关键字。 还应删除用于在应用程序入口点中等待异步方法的任何 `await` 关键字。

然而，在入口点继续执行之前，仍需要等待异步方法完成。 否则，编译会生成编译器警告 CS4014，且应用程序将在异步操作完成前终止。 以下示例展示了此问题：

C#

```
using System;
using System.Threading.Tasks;

public class Example
{
   public static void Main()
   {
       Console.WriteLine("About to wait two seconds");
       WaitTwoSeconds();
       Console.WriteLine("About to exit the program");
   }

   private static async Task WaitTwoSeconds()
   {
      await Task.Delay(2000);
      Console.WriteLine("Returning from an asynchronous method");
   }
}
// The example displays the following output:
//       About to wait two seconds
//       About to exit the program
```

若要等待返回 Task 的方法，请调用其 Wait 方法，如以下示例所示：

C#

```
using System;
using System.Threading.Tasks;

public class Example
{
   public static void Main()
   {
       Console.WriteLine("About to wait two seconds");
       WaitTwoSeconds().Wait();
       Console.WriteLine("About to exit the program");
   }

   private static async Task WaitTwoSeconds()
   {
      await Task.Delay(2000);
      Console.WriteLine("Returning from an asynchronous method");
   }
}
// The example displays the following output:
//       About to wait two seconds
```

```
//       Returning from an asynchronous method
//       About to exit the program
```

若要等待返回 Task<TResult> 的方法，请检索其 Result 属性的值，如以下示例所示：

C#

```
using System;
using System.Threading.Tasks;

public class Example
{
   public static void Main()
   {
       Console.WriteLine("About to wait two seconds");
       int value = WaitTwoSeconds().Result;
       Console.WriteLine($"Value returned from the async operation:
{value}");
       Console.WriteLine("About to exit the program");
   }

   private static async Task<int> WaitTwoSeconds()
   {
      await Task.Delay(2000);
      Console.WriteLine("Returning from an asynchronous method");
      return 100;
   }
}
// The example displays the following output:
//       About to wait two seconds
//       Returning from an asynchronous method
//       Value returned from the async operation: 100
//       About to exit the program
```

请参阅

- 关键字
- await 运算符

编译器错误 CS4032

项目 • 2023/04/07

“await”运算符只能用于异步方法中。 请考虑用“async”修饰符标记此方法，并将其返回类型更改为“Task<T>”。

示例

下面的示例生成 CS4032：

C#

```
// CS4032.cs (7,9)
using System.Collections.Generic;
using System.Threading.Tasks;

class C
{
    static IAsyncEnumerator<int> M(int value)
    {
        yield return value;
        await Task.CompletedTask;
    }
}
```

更正此错误

若要更正此错误，请更改方法 `M` 的签名以使其异步：

C#

```
using System.Collections.Generic;
using System.Threading.Tasks;

class C
{
    static async IAsyncEnumerator<int> M(int value)
    {
        yield return value;
        await Task.CompletedTask;
    }
}
```

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET feedback

The .NET documentation is open source. Provide feedback here.

Open a documentation issue

Provide product feedback

编译器错误 CS4033

项目 • 2023/04/07

“await”运算符只能用于异步方法中。 请考虑用“async”修饰符标记此方法，并将其返回类型更改为“Task”。

示例

下面的示例生成 CS4033：

C#

```
// CS4033.cs (7,9)

using System.Collections.Generic;
class C
{
    void M(IAsyncEnumerable<int> collection)
    {
        await foreach (var i in collection)
        {
        }
    }
}
```

更正此错误

若要更正此错误，请更改方法 `M` 的签名以使其异步：

C#

```
using System.Collections.Generic;
class C
{
    async void M(IAsyncEnumerable<int> collection)
    {
        await foreach (var i in collection)
        {
        }
    }
}
```

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET feedback

The .NET documentation is open source. Provide feedback here.

Open a documentation issue

Provide product feedback

编译器错误 CS5001

项目 • 2023/05/10

程序不包含适合于入口点的静态“Main”方法

当用于生成可执行文件的代码内未找到具有正确签名的静态 `Main` 方法时，将出现此错误。 如果用错误的大小写（例如小写的 `main`）定义了入口点函数 `Main`，则也将出现此错误。 有关应用于 `Main` 方法的规则的信息，请参阅 Main() 和命令行参数。

如果 `Main` 方法有一个 `async` 修饰符，请确保所选的 C# 语言版本是 7.1 或更高，并使用 `Task` 或 `Task<int>` 作为返回类型。

只有在编译可执行文件时才需要 `Main` 方法，即在指定了 TargetType 编译器选项中的 exe 或 winexe 元素时。 默认情况下，Visual Studio 项目类型指定以下选项之一：

- 控制台应用程序
- ASP.NET Core 应用程序
- WPF 应用程序
- Windows 窗体应用程序

示例

下面的示例生成 CS5001：

C#

```
// CS5001.cs
// CS5001 expected when compiled with -target:exe or -target:winexe
public class Program
{
   // Uncomment the following line to resolve.
   // static void Main() {}
}
```

编译器错误 CS7003

项目 • 2023/09/25

意外使用了未绑定的通用名称

如果使用一个泛型类型，该类型需要一个参数类型，但不在尖括号之间传递任何泛型参数类型名称，则会发生此错误。 这种用法可能用于变量声明或对象实例化。

示例

下面的示例生成 CS7003：

C#

```
// CS7003.cs
using System.Collections.Generic;

class Program
{
    static void Main(string[] args)
    {
        var myDictionary = new Dictionary< , >();  //CS7003

        List<> var2;                               //CS7003
    }
}
```

更正此错误

使用泛型类型时，请在尖括号中提供预期的参数类型名称，并用逗号分隔。

可按照如下方式修复前面的示例：

C#

```
// CS7003-fixed.cs
using System.Collections.Generic;

class Program
{
    static void Main(string[] args)
    {
        var myDictionary = new Dictionary<int, string>();

        List<string> var2;
```

```
    }
}
```

另请参阅

- 泛型

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

Open a documentation issue

Provide product feedback

编译器错误 CS8124

项目 • 2023/04/07

元组必须包含至少两个元素。

示例

下面的示例生成 CS8124：

C#

```
// CS8124.cs (4,20)

class C
{
    void M(int x, () y, (int a) z) { }
}
```

更正此错误

有效的元组声明包含两个或多个元素，确保元组声明具有两个或更多个元素可更正此错误：

C#

```
class C
{
    void M(int x, (int, int) y, (int a, int b) z) { }
}
```


Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

Open a documentation issue

Provide product feedback

编译器错误 CS8125

项目 • 2023/04/07

只允许在位置使用元组元素名称。

示例

下面的示例生成 CS8125：

C#

```
// CS8125.cs (2,15)

public class C
{
    public void Method()
    {
        var tuple3 = (Item2: 2, Item1: 1);
    }
}
```

更正此错误

如果使用元组元素名称 `Item1`、`Item2` 等，确保顺序正确即可更正此错误：

C#

```
    public void Method()
    {
        var tuple3 = (Item1: 2, Item2: 1);
    }
```

编译器错误 CS8127

项目 • 2023/04/07

元组元素名称必须是唯一的。

示例

以下示例生成 CS8127：

```
C#
```

```
// CS8127.cs (7,0)

internal struct NewStruct
{
    public int a;
    public int b;

    public static implicit operator (int a, int a)(NewStruct value)
    {
        return (value.a, value.b);
    }
}
```

更正此错误

请确保元组声明中的元素名称是唯一的，这可以消除此错误：

```
C#
```

```
    public static implicit operator (int a, int b)(NewStruct value)
    {
        return (value.a, value.b);
    }
```


Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

Open a documentation issue

more information, see our contributor guide.

Provide product feedback

编译器错误 CS8129

项目 • 2023/04/07

找不到类型适用的 Deconstruct 实例或扩展方法，参数为 `out`，返回类型为 `void`。

示例

下面的示例生成 CS8129：

C#

```
// CS8129.cs (11,52)

class C
{
    static void Main()
    {
        long x;
        string y;
        (x, y) = new C();
    }

    public int Deconstruct(out int a, out string b)
    {
        a = 1;
        b = "hello";
        return 42;
    }
}
```

更正此错误

有效的 `Deconstruct` 方法会返回 `void`，并具有两个或更多与要析构的元组类型匹配的 `out` 参数。 实现有效的 `Deconstruct` 方法可更正此错误：

C#

```
    public void Deconstruct(out int a, out string b)
    {
        a = 1;
        b = "hello";
    }
```

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET feedback

The .NET documentation is open source. Provide feedback here.

Open a documentation issue

Provide product feedback

编译器错误 CS8130

项目 • 2023/04/07

无法推断隐式类型的析构变量的类型。

示例

下面的示例生成 CS8130：

C#

```
// CS8130.cs (5,14)
class Program
{
    static void Main()
    {
        var (x2, y2) = () => { };
    }
}
```

编译器无法将委托（`Action`）转换为二元素元组，因此无法推断元组中每个元素的类型。

更正此错误

若要为元组分配值，确保右侧表达式元组与左侧表达式元组具有相同数量的元素，这可以更正此错误：

C#

```
    static void Main()
    {
        var (x2, y2) = (1, 2);
    }
```

编译器错误 CS8131

项目 • 2023/04/07

析构任务要求表达式属于右侧的某个类型。

示例

下面的示例生成 CS8131：

C#

```
// CS8131.cs (5,24)
class Program
{
    static void Main()
    {
        var (x2, y2) = () => { };
    }
}
```

编译器无法将委托 (`Action`) 转换为二元素元组，从而导致 CS8131。

更正此错误

若要将值分配给元组，请确保右侧表达式的元组类型与左侧表达式的元组类型相同：

C#

```
    static void Main()
    {
        var (x2, y2) = (1, 2);
    }
```


Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

Open a documentation issue

Provide product feedback

more information, see our contributor guide.

编译器错误 CS8132

项目 • 2023/04/07

无法将元素的元组析构为变量。

示例

下面的示例生成 CS8132：

C#

```
// CS8132.cs (5,9)
class Program
{
    static void F(object x, object y, object? z)
    {
        (object? a, object? b) = (x, y, z = 3);
        Console.WriteLine(a);
        Console.WriteLine(b);
        Console.WriteLine(z);
    }
}
```

更正此错误

若要解构为元素数量较少的元组，使用弃元变量可更正此错误：

C#

```
class Program
{
    static void F(object x, object y, object? z)
    {
        (object? a, object? b, object _) = (x, y, z = 3);
        Console.WriteLine(a);
        Console.WriteLine(b);
        Console.WriteLine(z);
    }
}
```

Collaborate with us on GitHub

.NET feedback

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

The .NET documentation is open source. Provide feedback here.

Open a documentation issue

Provide product feedback

编译器错误 CS8139

项目 • 2023/04/07

替代继承成员时无法更改元组元素名称

示例

下面的示例生成 CS8139：

C#

```
// CS8139.cs (9,38)

public class Base
{
    public virtual (object a, object b) M((object c, object d) x) { return
x; }
}

class C : Base
{
    public override (object, object) M((object c, object d) y) { return y; }
}
```

更正此错误

确保替代成员中的元组元素名称与虚拟成员中的元组元素名称匹配，即可更正此错误：

C#

```
class C : Base
{
    public override (object a, object b) M((object c, object d) y) { return
y; }
}
```


Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

Open a documentation issue

issues and pull requests. For more information, see our contributor guide.

Provide product feedback

编译器错误 CS8140

项目 • 2023/04/07

已列入类型的接口列表中，其中包含不同的元组元素名称。

示例

下面的示例生成 CS8140：

C#

```
// CS8140.cs (11,7)

interface I<T>
{
    T GetValue();
}

interface I2 : I<(int c, int d)>
{
}

class C : I<(int a, int b)>, I2
{
    public (int c, int d) GetValue()
    {
        return (1, 2);
    }
}
```

更正此错误

重命名元组元素名称以匹配接口声明可更正此错误：

C#

```
interface I<T>
{
    T GetValue();
}

interface I2 : I<(int c, int d)>
{
}

class C : I<(int c, int d)>, I2
{
```

```
    public (int c, int d) GetValue()
    {
        return (1, 2);
    }
}
```

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET feedback

The .NET documentation is open source. Provide feedback here.

- Open a documentation issue
- Provide product feedback

编译器错误 CS8141

项目 • 2023/04/07

方法的签名中的元组元素名称必须与接口方法的元组元素名称匹配（包括返回类型）。

示例

下面的示例生成 CS8141：

C#

```
// CS8141.cs (10,27)
using System.Collections;

public interface IGrabber<out T>
{
    T GetOne();
}

class SomeGrabber : IGrabber<(int, int)>
{
    public (int a, int b) GetOne()
    {
        return (1, 2);
    }
}
```

更正此错误

更改 `GetOne` 方法的签名以返回未命名元组（与接口中的未命名元组匹配）将更正此错误：

C#

```
    public (int, int) GetOne()
    {
        return (1, 2);
    }
```

Collaborate with us on GitHub

.NET feedback

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

The .NET documentation is open source. Provide feedback here.

Open a documentation issue

Provide product feedback

编译器错误 CS8145

项目 • 2024/10/26

自动实现的属性无法通过引用返回

自动实现的属性不能保证具有可以引用的成员或变量，因此不支持按引用返回。

示例

下面的示例生成 CS8145：

C#

```
// CS8145.cs (4,13)

public class C
{
    public ref int Property1 { get; }
}
```

更正此错误

如果可以通过支持字段实现该属性，则重构为使用支持字段并 `ref`-返回该字段可更正此错误：

C#

```
public class C
{
    private int property1;

    public ref int Property1 => ref property1;
}
```

如果无法通过支持字段实现该属性，则从该属性中删除 `ref` 修饰符可更正此错误：

C#

```
public class C
{
    public int Property1 { get; }
}
```

编译器错误 CS8146

项目 • 2023/04/07

按引用返回的属性必须有 get 访问器

示例

下面的示例生成 CS8146：

C#

```
// CS8146.cs (5,18)

public class C
{
    private ref int number;

    private ref int Number { init => number = value; }
}
```

更正此错误

确保通过引用返回的属性具有 get 访问器将更正此错误：

C#

```
public class C
{
    private ref int number;

    private ref int Number => ref number;
}
```


Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

Open a documentation issue

Provide product feedback

more information, see our contributor guide.

编译器错误 CS8147

项目 • 2023/04/07

通过引用返回的属性不能有 set 访问器

示例

下面的示例生成 CS8147：

C#

```
// CS8147.cs (6,44)

public class C
{
    private ref int number;

    ref int Number { get => ref number; init => number = value; }
}
```

更正此错误

删除通过引用返回的属性的任何 set 访问器将更正此错误：

C#

```
public class C
{
    private ref int number;

    ref int Number => ref number;
}
```


Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

Open a documentation issue

Provide product feedback

more information, see our contributor guide.

编译器错误 CS8148

项目 • 2023/04/07

必须按引用匹配重写成员的返回

示例

下面的示例生成 CS8148：

C#

```
// CS8148.cs (11,29)

public class Base
{
    public virtual int GetNumber() { return 0; }
}

public class Derived : Base
{
    private int number;

    public override ref int GetNumber() { return ref number; }
}
```

更正此错误

确保按值返回的基本成员由不按引用返回的成员实现，即可更正此错误：

C#

```
public class Base
{
    public virtual int GetNumber() { return 0; }
}

public class Derived : Base
{
    private int number;

    public override int GetNumber() { return number; }
}
```

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET feedback

The .NET documentation is open source. Provide feedback here.

Open a documentation issue

Provide product feedback

编译器错误 CS8149

项目 • 2023/04/07

按引用返回只能在按引用返回的方法中使用

示例

下面的示例生成 CS8149：

C#

```
// CS8149.cs (9,33)

delegate int E();

class C
{
    static int i;
    static void M()
    {
        var e = new E(() => ref i);
    }
}
```

更正此错误

确保声明为按值返回的方法和委托不会按引用返回，即可更正此错误：

C#

```
delegate int E();

class C
{
    static int i;
    static void M()
    {
        var e = new E(() => i);
    }
}
```

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET feedback

The .NET documentation is open source. Provide feedback here.

Open a documentation issue

Provide product feedback

编译器错误 CS8150

项目 • 2023/04/07

按值返回只能在按值返回的方法中使用

示例

下面的示例生成 CS8150：

C#

```
// CS8150.cs (6,9)

class C
{
    ref int M()
    {
        return new();
    }
}
```

更正此错误

若要按值返回，请将方法从按引用进行重构。 例如：

C#

```
// CS8150.cs (6,9)

class C
{
    int M()
    {
        return new();
    }
}
```

Collaborate with us on GitHub

.NET feedback

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

The .NET documentation is open source. Provide feedback here.

Open a documentation issue

Provide product feedback

编译器错误 CS8151

项目 • 2023/04/07

返回表达式必须为类型，因为此方法按引用返回

示例

以下示例生成 CS8151：

C#

```
// CS8151.cs (6,20)

class Program
{
    ref int M(ref long i)
    {
        return ref i;
    }
}
```

更正此错误

确保 `ref` 返回与引用变量的类型匹配（忽略任何隐式转换），可更正此错误：

C#

```
class Program
{
    ref long M(ref long i)
    {
        return ref i;
    }
}
```


Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For

.NET feedback

The .NET documentation is open source. Provide feedback here.

Open a documentation issue

more information, see our contributor guide.

Provide product feedback

编译器错误 CS8152

项目 • 2023/04/07

类型不实现接口成员。 无法实现，因为它没有按引用的匹配返回。

示例

下面的示例生成 CS8152：

若要使用按引用返回的方法实现接口，该方法的实现还必须按引用而不是按值返回。

C#

```
// CS8152.cs (6,21)

public interface ITest
{
    ref readonly int M();
}
public class Test : ITest
{
    public int M() => 0;
}
```

更正此错误

确保按引用返回的接口方法不按值返回。 例如：

C#

```
public class Test : ITest
{
    int m;
    public ref readonly int M() => ref m;
}
```

编译器错误 CS8156

项目 • 2023/04/07

不能在此上下文中使用表达式，因为表达式无法通过引用传递或返回

示例

以下示例生成 CS8156：

C#

```
// CS8156.cs (7,27)

class Test
{
    delegate ref int D1();

    void Test1()
    {
        D1 d1 = () => ref 2 + 2;
    }
}
```

更正此错误

如果不使用可引用变量，重构以按值返回可更正此错误：

C#

```
class Test
{
    delegate int D1();

    void Test1()
    {
        D1 d1 = () => 2 + 2;
    }
}
```

Collaborate with us on GitHub

.NET feedback

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

The .NET documentation is open source. Provide feedback here.

Open a documentation issue

Provide product feedback

编译器错误 CS8157

项目 • 2023/04/07

已初始化为不能通过引用返回的值，因此无法通过引用返回

示例

下面的示例生成 CS8157：

C#

```
// CS8157.cs (8,21)

class C
{
    ref int M()
    {
        int x = 0;
        ref int rx = ref x;
        return ref (rx = ref (new int[1])[0]);
    }
}
```

更正此错误

若要返回一个不能按引用返回的值，重构为按值返回可以更正此错误：

C#

```
class C
{
    int M()
    {
        int x = 0;
        ref int rx = ref x;
        return rx = ref (new int[1])[0];
    }
}
```

Collaborate with us on GitHub

.NET feedback

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

The .NET documentation is open source. Provide feedback here.

Open a documentation issue

Provide product feedback

编译器错误 CS8158

项目 • 2023/04/07

已初始化为不能按引用返回的值，因此无法按引用返回成员

示例

下面的示例生成 CS8158：

C#

```
// CS8158.cs (11,14)

public class Test
{
    public struct S1
    {
        public char x;
    }

    ref char Test1(char arg1, S1 arg2)
    {
        ref S1 r = ref arg2;
        return ref r.x;
    }
}
```

更正此错误

要返回已初始化为不能按引用返回的值的成员，重构为按值返回可更正此错误：

C#

```
    char Test1(char arg1, S1 arg2)
    {
        ref S1 r = ref arg2;
        return r.x;
    }
```

Collaborate with us on GitHub

.NET feedback

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

The .NET documentation is open source. Provide feedback here.

Open a documentation issue

Provide product feedback

编译器错误 CS8159

项目 • 2023/04/07

无法通过引用返回范围变量

示例

以下示例生成 CS8159：

C#

```
// CS8159.cs (7,74)

using System.Linq;
class TestClass
{
    delegate ref char RefCharDelegate();
    void TestMethod()
    {
        var x = from c in "TestValue" select (RefCharDelegate)(() => ref c);
    }
}
```

更正此错误

若要返回范围变量，请重构为按值返回以更正此错误：

C#

```
using System.Linq;
class TestClass
{
    delegate char RefCharDelegate();
    void TestMethod()
    {
        var x = from c in "TestValue" select (RefCharDelegate)(() => c);
    }
}
```

Collaborate with us on GitHub

.NET feedback

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

The .NET documentation is open source. Provide feedback here.

Open a documentation issue

Provide product feedback

编译器错误 CS8160

项目 • 2023/10/06

只读字段无法按可写引用返回

示例

下面的示例生成 CS8160：

```C#
// CS8160.cs (8,20)

class Program
{
    readonly int i = 0;

    ref int M()
    {
        return ref i;
    }
}
```

更正此错误

要返回 `readonly` 字段，请重构为按值返回：

```C#
class Program
{
    readonly int i = 0;

    int M()
    {
        return i;
    }
}
```

Collaborate with us on GitHub

.NET feedback

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

The .NET documentation is open source. Provide feedback here.

Open a documentation issue

Provide product feedback

编译器错误 CS8161

项目 • 2023/04/07

静态只读字段无法按可写引用返回

示例

下面的示例生成 CS8161：

C#

```
// CS8161.cs (12,14)
public class Test
{
    public struct S1
    {
        public char x;
    }

    public static readonly char s1;

    ref char Test2()
    {
        return ref s1;
    }
}
```

更正此错误

要返回 `static readonly` 字段的值，请重构为按值返回：

C#

```
public class Test
{
    public struct S1
    {
        public char x;
    }

    public static readonly char s1;

    char Test2()
    {
        return s1;
```

```
    }
}
```

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET feedback

The .NET documentation is open source. Provide feedback here.

- Open a documentation issue
- Provide product feedback

编译器错误 CS8162

项目 • 2023/04/07

只读字段的成员无法通过可写引用返回

示例

下面的示例生成 CS8162：

C#

```
// CS8162.cs (12,14)
public class Test
{
    public struct S1
    {
        public char x;
    }

    public readonly S1 i2;

    ref char Test1()
    {
        return ref i2.x;
    }
}
```

更正此错误

若要返回 `readonly` 字段的值，重构为按值返回可更正此错误：

C#

```
public class Test
{
    public struct S1
    {
        public char x;
    }

    public readonly S1 i2;
    char Test1()
    {
        return i2.x;
```

```
    }
}
```

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET feedback

The .NET documentation is open source. Provide feedback here.

- Open a documentation issue
- Provide product feedback

编译器错误 CS8163

项目 • 2023/04/07

静态只读字段的字段无法通过可写引用返回

示例

以下示例生成 CS8163：

C#

```
// CS8163.cs (12,14)
public class Test
{
    public struct S1
    {
        public char x;
    }

    public static readonly S1 s2;

    char Test2()
    {
        return s2.x;
    }
}
```

更正此错误

若要返回 `static readonly` 字段的值，重构为按值返回可更正此错误：

C#

```
public class Test
{
    public struct S1
    {
        public char x;
    }

    public static readonly S1 s2;

    char Test2()
    {
        return s2.x;
```

```
    }
}
```

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET feedback

The .NET documentation is open source. Provide feedback here.

- Open a documentation issue
- Provide product feedback

编译器错误 CS8170

项目 • 2023/04/07

结构成员无法按引用返回“this”或其他实例成员

示例

下面的示例生成 CS8170：

值类型（即 `struct`）最常在堆栈上分配。 在堆栈上分配的值类型离开声明它的范围后将失效。 编译器通过生成此错误来避免对失效变量的引用离开范围。

C#

```
// CS8170.cs (8,14)

struct Program
{
    public int d;

    public ref int M()
    {
        return ref d;
    }
}

public class Other
{
    public void Method()
    {
        var p = new Program();
        var d = p.M();
    }
}
```

更正此错误

将方法更改为 not `ref`-return 可更正此错误：

C#

```
delegate void D();

struct Program
{
    public event D d;
```

```
    public D M()
    {
        return d;
    }
}
```

如果需要对成员的引用，请考虑扩展值的范围。 例如：

C#

```
public struct Program
{
    public int d;
}

public static class Extensions
{
    public static ref readonly int RefD(this in Program program)
    {
        return ref program.d;
    }
}

public class Other
{
    public void Method()
    {
        var p = new Program();
        var d = p.RefD();
    }
}
```


Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

Open a documentation issue

Provide product feedback

编译器错误 CS8171

项目 • 2023/04/07

无法使用引用初始化按值变量

示例

下面的示例生成 CS8171：

C#

```
// CS8171.cs (8,13)

class Test
{
    void A()
    {
        int a = 123;
        ref int x = ref a;
        var y = ref x;
    }
}
```

请记住，`var y = ref x` 通过隐式方式 `int y = ref x`，其中 `int y` 是按值变量。

更正此错误

从分配右侧删除 `ref` 修饰符可更正此错误：

C#

```
class Test
{
    void A()
    {
        int a = 123;
        ref int x = ref a;
        var y = x;
    }
}
```

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET feedback

The .NET documentation is open source. Provide feedback here.

Open a documentation issue

Provide product feedback

编译器错误 CS8172

项目 • 2023/04/07

无法使用值初始化按引用变量

示例

下面的示例生成 CS8172：

C#

```
// CS8172.cs (10,17)

class C
{
    void M()
    {
        ref readonly int L() => ref (new int[1])[0];

        ref readonly int x = ref L();
        ref int y = x;
    }
}
```

更正此错误

将对变量的引用分配给一个按引用变量可更正此错误：

C#

```
class C
{
    void M()
    {
        ref readonly int L() => ref (new int[1])[0];

        ref readonly int x = ref L();
        ref int y = ref x;
    }
}
```

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET feedback

The .NET documentation is open source. Provide feedback here.

Open a documentation issue

Provide product feedback

编译器错误 CS8173

项目 • 2023/04/07

表达式必须为类型，因为它按引用赋值

向变量分配引用时，变量的类型必须匹配才能引用。

示例

下面的示例生成 CS8173：

C#

```
// CS8173.cs (12,18)

class C
{
    void M()
    {
        string s = "s";
        object o = s;
        ref string rs = ref s;
        ref object ro = ref o;

        ro = ref s;
    }
}
```

更正此错误

将引用分配给正确类型的变量将更改此错误：

C#

```
        rs = ref s;
```


Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review

.NET feedback

The .NET documentation is open source. Provide feedback here.

Open a documentation issue

issues and pull requests. For more information, see our contributor guide.

Provide product feedback

编译器错误 CS8174

项目 • 2023/04/07

by-reference 变量的声明必须有初始化表达式

示例

下面的示例生成 CS8174：

C#

```
// CS8174.cs (7,22)

class C
{
    void M()
    {
        int i = 0;
        for (ref int rx; i < 5; i++) { }
    }
}
```

更正此错误

通过对变量的引用初始化 by-reference 变量将更正此错误：

C#

```
class C
{
    void M()
    {
        int i = 0;
        for (ref int rx = ref i; i < 5; i++) { }
    }
}
```


Collaborate with us on GitHub

The source for this content can be found on GitHub, where you

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

can also create and review issues and pull requests. For more information, see our contributor guide.

Open a documentation issue

Provide product feedback

编译器错误 CS8177

项目 • 2024/07/04

异步方法不能有按引用的局部变量

为了管理异步状态，`async` 方法使用状态机，捕获在编译器生成的类和属性中实现的封闭中的变量状态。 无法在堆中的类实例中捕获（堆栈上的）局部变量引用，因此编译器会发出错误。

示例

在 C# 13 之前，下面的示例会生成 CS8177：

C#

```
// CS8177.cs (20,26)

using System.Threading.Tasks;

class E
{
    public class Enumerator
    {
        public ref int Current => throw new
System.NotImplementedException();
        public bool MoveNext() => throw new
System.NotImplementedException();
    }

    public Enumerator GetEnumerator() => new Enumerator();
}

class C
{
    public async static Task Test()
    {
        await Task.CompletedTask;

        foreach (ref int x in new E())
        {
            System.Console.Write(x);
        }
    }
}
```

更正此错误

删除该 `ref` 修饰符。 或者，可以升级到随 .NET 9 一起提供的 C# 13。

C#

```
class C
{
    public async static Task Test()
    {
        await Task.CompletedTask;

        foreach (int x in new E())
        {
            System.Console.Write(x);
        }
    }
}
```

在 GitHub 上与我们协作

可以在 GitHub 上找到此内容的源，还可以在其中创建和查看问题和拉取请求。 有关详细信息，请参阅参与者指南。

.NET 反馈

.NET 是一个开放源代码项目。 选择一个链接以提供反馈：

提出文档问题

提供产品反馈

编译器错误 CS8178

项目 • 2023/04/07

不能在包含调用的表达式中使用“await”，因为它按引用返回

示例

下面的示例生成 CS8178：

C#

```
using System;
using System.Threading.Tasks;

class TestClass
{
    int x;
    ref int Save(int y)
    {
        x = y;
        return ref x;
    }

    async Task TestMethod()
    {
        Save(1) = await Task.FromResult(0);
    }
}
```

更正此错误

将按引用返回更改为同步可更正此错误：

C#

```
    async Task TestMethod()
    {
        var x = await Task.FromResult(0);
        Save(1) = x;
    }
```

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET feedback

The .NET documentation is open source. Provide feedback here.

Open a documentation issue

Provide product feedback

编译器错误 CS8210

项目 • 2023/04/07

元组不能包含类型为 "void" 的值。

示例

下面的示例生成 CS8210：

```C#
// CS8210.cs
void Method()
{
}

void Test()
{
    var x = ("something", Method());
}
```

方法返回 `void`，这样该方法不返回值。 不能使用为元组的数据成员返回 `void` 的方法。

请参阅

- Tuple
- Void

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

Open a documentation issue

Provide product feedback

编译器错误 CS8333

项目 • 2023/04/07

无法按可写引用返回，因为它是只读变量

示例

下面的示例生成 CS8333：

```
C#
```

```
// CS8333.cs (3,36)
class Program
{
    static ref T F4<T>(in T t) => ref t;
}
```

更正此错误

要返回对只读变量的引用，重构为返回 `ref readonly` 可更正此错误：

```
C#
```

```
class Program
{
    static ref readonly T F4<T>(in T t) => ref t;
}
```


Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

Open a documentation issue

Provide product feedback

编译器错误 CS8334

项目 • 2023/04/07

不能通过可写引用返回的成员，因为这是只读变量

示例

下面的示例生成 CS8334：

C#

```
// CS8334.cs (5,14)

class Program
{
    static ref int M(in int arg1, in (int Alice, int Bob) arg2)
    {
        return ref arg2.Alice;
    }
}
```

更正此错误

若要返回对只读成员的引用，重构以返回 `ref readonly`，从而更正此错误：

C#

```
class Program
{
    static ref readonly int M(in int arg1, in (int Alice, int Bob) arg2)
    {
        return ref arg2.Alice;
    }
}
```


Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For

.NET feedback

The .NET documentation is open source. Provide feedback here.

Open a documentation issue

more information, see our contributor guide.

Provide product feedback

编译器错误 CS8352

项目 • 2023/12/11

不能在此上下文中使用变量“variable”，因为它可能会在其声明范围以外公开所引用的变量。

此错误指示当存在访问堆栈分配的变量的风险时（在从堆栈中删除该变量后）可能的未定义行为。

更正此错误

通过确保变量的范围至少与引用它变量的范围一样宽来修复 `CS8352`。 可通过在堆上分配存储或在调用上下文的堆栈中分配存储来执行此操作。

示例

以下示例生成 CS8352：

C#

```
class Program
{
    static public Span<int> CreateSpanWithValue(int size, int value)
    {
        Span<int> localSpan = stackalloc int[size];
        localSpan[0] = value;
        return localSpan;    // CS8352
    }                        // localSpan refers to stack memory that's been
reclaimed.

    static public void Main()
    {
        Span<int> localSpan = CreateSpanWithValue(5, 10);
        localSpan[2] = 14;
        foreach(var item in localSpan)
        {
            Console.WriteLine(item.ToString());  // Reading from deleted
object
        }
    }
    // Example of output:
    // 284945320
    // 149
    // 149
    // 1369466325
```

```
    // 284945320
}
```

以下示例移动堆栈分配，使其范围与 `Span` 匹配。
此示例解决方案可行，因为 Span 引用的内存在堆栈上分配，而不是在调用上下文中分配：

C#

```
class Program
{
    static public void FillSpanWithValue(Span<int> span, int value)
    {
        span[0] = value;
    }

    static public void Main()
    {
        Span<int> localSpan = stackalloc int[5];
        FillSpanWithValue(localSpan, 10);
        localSpan[2] = 14;
        foreach(var item in localSpan)
        {
            Console.WriteLine(item.ToString());
        }
    }
    // Output:
    // 10
    // 0
    // 14
    // 0
    // 0
}
```

以下示例使用 `new` 关键字在堆（而不是堆栈）上创建对象。
此示例解决方案可行，因为在从 `CreateSpanWithValue` 方法返回后，Span 引用的内存仍然有效，因为它在堆上分配，而不是堆栈上分配：

C#

```
class Program
{
    static public Span<int> CreateSpanWithValue(int size, int value)
    {
        Span<int> localSpan = new int[size];
        localSpan[1] = value;
        return localSpan;
    }

    static public void Main()
```

```
    {
        var span = CreateSpanWithValue(5, 10);
        span[2] = 14;
        foreach(var item in span)
        {
            Console.WriteLine(item.ToString());
        }
    }
    // Output:
    //0
    //10
    //14
    //0
    //0
}
```

另请参阅

- in 参数修饰符
- new 运算符
- stackalloc表达式

在 GitHub 上与我们协作

可以在 GitHub 上找到此内容的源，还可以在其中创建和查看问题和拉取请求。 有关详细信息，请参阅参与者指南。

.NET 反馈

.NET 是一个开放源代码项目。 选择一个链接以提供反馈：

提出文档问题

提供产品反馈

编译器错误 CS8354

项目 • 2023/04/07

无法通过引用来返回 "this"。

`ref` 参数必须是可分配的变量或数组元素。 `this` 不可分配，且不能通过引用返回。

示例

以下示例生成 CS8354：

C#

```
// CS8354.cs (6,20)

class Program
{
    ref Program M()
    {
        return ref this;
    }
}
```


Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

Open a documentation issue

Provide product feedback

编译器错误 CS8355

项目 • 2023/12/18

in 参数不能具有 Out 特性。

不能将方法的参数声明为只读，同时通知互操作封送器将此参数视为本机代码可写。 OutAttribute 在封送 时将会与 `in` 关键字相矛盾。

示例

下面的示例会生成 CS8355：

```
C#

using System.Runtime.InteropServices;

class Program
{
    public static void Example([Out] in int value)    // CS8355
    {
    }
}
```

另请参阅

- 默认封送处理行为
- 引用参数

在 GitHub 上与我们协作

可以在 GitHub 上找到此内容的源，还可以在其中创建和查看问题和拉取请求。 有关详细信息，请参阅参与者指南。

.NET 反馈

.NET 是一个开放源代码项目。 选择一个链接以提供反馈：

提出文档问题

提供产品反馈

编译器错误 CS8403

项目 • 2023/04/07

带有迭代器块的方法“method”必须为“异步”，才能返回“{IAsyncEnumerable<T>|IAsyncEnumerator<T>}”

更正此错误

请使用 `async` 修饰符标记方法。

另请参阅

- async（C# 参考）

编译器错误 CS8410

项目 • 2023/04/07

“类型”：异步 using 语句中使用的类型必须可隐式转换为“System.IAsyncDisposable”或实现适用的“DisposeAsync”方法。

`await using` 语句中的表达式必须具有 `DisposeAsync` 方法。

更正此错误

删除 `await using` 关键字，或实现适当的 `DisposeAsync` 方法。

示例

C#

```
using System.Threading.Tasks;

class Program
{
    static async Task Main()
    {
        // error CS8410: 'Example': type used in an asynchronous using
statement
        // must be implicitly convertible to 'System.IAsyncDisposable' or
implement
        // a suitable 'DisposeAsync' method.
        await using var example = new Example();
    }
}

class Example
{
}
```

另请参阅

- 实现 DisposeAsync 方法

编译器错误 CS8411

项目 • 2023/04/07

异步 foreach 语句无法对类型“type”的变量进行操作，因为“type”不包含适合“GetAsyncEnumerator”的公共实例定义

`await foreach` 语句仅对定义为 `GetAsyncEnumerator` 的类型（例如 `IAsyncEnumerable<T>`）起作用。

更正此错误

将 `await foreach` 替换为 `foreach`。

示例

C#

```
using System.Collections.Generic;
using System.Threading.Tasks;

class Program
{
    async Task Example(IAsyncEnumerator<int> enumerator)
    {
        // error CS8411: Asynchronous foreach statement cannot operate on
variables
        // of type 'IAsyncEnumerator<T>' because 'IAsyncEnumerator<T>' does
not
        // contain a suitable public instance definition for
'GetAsyncEnumerator'
        await foreach (int i in enumerator)
        {
        }
    }
}
```

编译器错误 CS8422

项目 • 2023/12/20

静态本地函数不能包含对“this”或“base”的引用。

本地函数上的 static 关键字可防止本地函数访问父实例方法或实例字段的状态。
此错误指示本地函数访问包含类型的实例成员。

示例

下面的示例会生成 CS8422：

C#

```
public class C
{
    private int counter = 1;
    public void IncreaseCounter()
    {
        static void localFunc(int addition)
        {
            this.counter += addition;   // CS8422 due to reference to 'this'
            base.ToString();            // CS8422 due to reference to 'base'

            // Also for implicit 'this' or 'base' references:
            counter += addition;   // CS8422 due to implicit reference to
'this'
            ToString();            // CS8422 due to implicit reference to
'base'
        }
        localFunc(10);
        Console.WriteLine(this.counter);
    }
}
```

更正此错误

如果有本地函数用于捕获和修改父函数的状态，则不应将其声明为静态。 还可以删除对任何实例成员的访问权限。

C#

```
public class C
{
    private int counter = 1;
    public void IncreaseCounter()
```

```
    {
        void localFunc(int addition)
        {
            counter += addition;
            ToString();
        }
        localFunc(10);
        Console.WriteLine(this.counter);
    }
}

public class Program
{
    public void Main()
    {
        C c = new();
        c.IncreaseCounter();
    }
}

// Output:
// 11
```

另请参阅

- 本地函数
- CS0120

在 GitHub 上与我们协作

可以在 GitHub 上找到此内容的源，还可以在其中创建和查看问题和拉取请求。 有关详细信息，请参阅参与者指南。

.NET 反馈

.NET 是一个开放源代码项目。 选择一个链接以提供反馈：

提出文档问题

提供产品反馈

编译器错误 CS8515

项目 • 2024/07/04

需要用括号将 switch 控制表达式括起来。

示例

以下代码片段生成 CS8515：

缺少括号：`(` `)`：

```C#
// CS8515.cs
int x = 5;
switch x
{
    case 5:
    break;
}
```

缺少大括号：`{` `}`：

```C#
// CS8515.cs
int x = 5;
switch (x)
    case 5:
    break;
```

正确的格式

```C#
int x = 5;
switch (x)
{
    case 5:
    break;
}
```

有关详细信息，请参阅 Switch。

在 GitHub 上与我们协作

可以在 GitHub 上找到此内容的源，还可以在其中创建和查看问题和拉取请求。 有关详细信息，请参阅参与者指南。

.NET 反馈

.NET 是一个开放源代码项目。 选择一个链接以提供反馈：

提出文档问题

提供产品反馈

Compiler Error CS8802

Article • 11/27/2024

Only one compilation unit can have top-level statements.

This error indicates that there are two or more top-level statements in a single compilation unit (single project or a single group of files compiled into a single binary file).

Example

The following sample of single compilation unit generates CS8802:

XML

```xml
<!-- SingleCompilationUnit.csproj -->
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <TargetFramework>net8.0</TargetFramework>
  </PropertyGroup>
</Project>
```

C#

```
// EntryFile.cs

int a = 0;
```

C#

```
// SecondaryEntryFile.cs

int b = 1;    // CS8802: The top level statement already exists in
EntryFile.cs
```

To correct this error

Use only one top-level statement in the project.
Top-level statements acts as an entry point to the program, so only one file may have top-level statement. All other statements must be defined as members of classes or structs.

See also

- Top-level statements
- CS8803

编译器错误 CS8803

项目 • 2023/04/07

顶级语句必须位于命名空间和类型声明之前。

示例

下面的示例生成 CS8803：

C#

```
// CS8803.cs (0,0)

public record Person
{
    public string? GivenName { get; set; }
    public string? FamilyName { get; set; }
}

int i = 0;
```

在具有顶级语句的文件中，顶级语句必须在任何类型声明之前发生。

更正此错误

将代码移到命名空间声明之前：

C#

```


int i = 0;

public record Person
{
    public string? GivenName { get; set; }
    public string? FamilyName { get; set; }
}
```

通常，类型在各自的文件中声明，这也将通过将类型声明与顶级语句分开来更正此错误。

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET feedback

The .NET documentation is open source. Provide feedback here.

Open a documentation issue

Provide product feedback

编译器错误 CS8812

项目 • 2024/02/01

无法将 `&Method` 组转换为非函数指针类型。

表达式的地址（例如 `&Method`）没有类型，因此无法分配给非函数指针变量。

`&` 运算符是 address-of 运算符，用来返回其操作数的地址。

示例

下面的示例生成 CS8812：

C#

```
// CS8812.cs (6,22)

unsafe class C
{
    static void Method()
    {
        void* ptr1 = &Method;
    }
}
```

更正此错误

将表达式显式转换为所需的类型（例如 `void delegate`）：

C#

```
unsafe class C
{
    static void Method()
    {
        void* ptr1 = (delegate*<void>)&Method;
    }
}
```

另请参阅

- Shouldn't method addresses be implicitly convertible to void* and thus allowing direct casts to function pointers with mismatched signatures? （方法地址不应隐

式转换为 void* 从而允许直接强制转换为签名不匹配的函数指针吗？）

- Casting function pointer directly to void and then a function pointer type causes crash.（将函数指针直接强制转换为 void，然后函数指针类型会导致崩溃。）
- Address-of 运算符 &

在 GitHub 上与我们协作

可以在 GitHub 上找到此内容的源，还可以在其中创建和查看问题和拉取请求。 有关详细信息，请参阅参与者指南。

.NET 反馈

.NET 是一个开放源代码项目。 选择一个链接以提供反馈：

提出文档问题

提供产品反馈

编译器错误 CS8515

项目 • 2024/07/04

需要用括号将 switch 控制表达式括起来。

示例

以下代码片段生成 CS8515：

缺少括号：`(` `)`：

```C#
// CS8515.cs
int x = 5;
switch x
{
    case 5:
    break;
}
```

缺少大括号：`{` `}`：

```C#
// CS8515.cs
int x = 5;
switch (x)
    case 5:
    break;
```

正确的格式

```C#
int x = 5;
switch (x)
{
    case 5:
    break;
}
```

有关详细信息，请参阅 Switch。

在 GitHub 上与我们协作

可以在 GitHub 上找到此内容的源，还可以在其中创建和查看问题和拉取请求。 有关详细信息，请参阅参与者指南。

.NET 反馈

.NET 是一个开放源代码项目。 选择一个链接以提供反馈：

提出文档问题

提供产品反馈

编译器错误 CS9043

项目 • 2023/04/07

无法要求引用返回属性。

`required` 修饰符指定需要在对象初始化期间设置成员（即通过对象初始值设置）。对于要在对象初始值设定项中设置的属性，它必须具有 `set` 访问器（资源库）。 `ref` 返回属性不能有资源库，因此也不能包含 `required` 修饰符。

示例

以下示例生成 CS9043：

C#

```
// CS9043.cs (5,29)

class C
{
    private int i;
    public required ref readonly int Number => ref i;
}
```

更正此错误

若要具有 `required` 属性，重构属性以按值返回可更正此错误：

C#

```
    public required int Number
    {
        get
        {
            return i;
        }
        set
        {
            i = value;
        }
    }
```

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET feedback

The .NET documentation is open source. Provide feedback here.

Open a documentation issue

Provide product feedback

编译器警告（等级 1）CS0183

项目 • 2023/04/07

给定表达式始终为所提供的（“类型”）类型

如果条件语句的计算结果始终为 **true**，则不需要条件语句。 当尝试使用 **is** 运算符评估类型时，会出现此警告。 如果计算结果是值类型，则不需要进行检查。

下面的示例生成 CS0183:

C#

```
// CS0183.cs
// compile with: /W:1
using System;
public class Test
{
   public static void F(Int32 i32, String str)
   {
      if (str is Object)          // OK
         Console.WriteLine( "str is an object" );
      else
         Console.WriteLine( "str is not an object" );

      if (i32 is Object)   // CS0183
         Console.WriteLine( "i32 is an object" );
      else
         Console.WriteLine( "i32 is not an object" ); // never reached
   }

   public static void Main()
   {

      F(0, "CS0183");
      F(120, null);
   }
}
```

编译器警告（等级 1）CS0184

项目 • 2023/04/07

给定表达式始终不是所提供的（“type”）类型

该表达式绝不会是 true，因为你测试的变量既不声明为“类型”，也不派生自“类型”。

下面的示例生成 CS0184：

C#

```
// CS0184.cs
// compile with: /W:1
class MyClass
{
   public static void Main()
   {
      int i = 0;
      if (i is string)   // CS0184
         i++;
   }
}
```

编译器警告（等级 1）CS0197

项目 • 2023/10/29

由于“argument”是引用封送类的字段，因此，将它作为 ref 或 out 参数传递或获取它的地址可能导致运行时异常

从 MarshalByRefObject 直接或间接派生的任何类都是引用封送类。 这样的类可以跨进程和计算机边界引用封送。 因此，此类的实例可以是远程对象的代理。 不能将代理对象的字段作为 ref 或 out传递。因此，不能传递 `ref` 或 `out` 这类字段，除非实例是 this，它不能是代理对象。

示例

下面的示例生成 CS0197。

C#

```
// CS0197.cs
// compile with: /W:1
class X : System.MarshalByRefObject
{
   public int i;
}

class M
{
   public int i;
   static void AddSeventeen(ref int i)
   {
      i += 17;
   }

   static void Main()
   {
      X x = new X();
      x.i = 12;
      AddSeventeen(ref x.i);   // CS0197

      // OK
      M m = new M();
      m.i = 12;
      AddSeventeen(ref m.i);
   }
}
```

在 GitHub 上与我们协作

可以在 GitHub 上找到此内容的源，还可以在其中创建和查看问题和拉取请求。 有关详细信息，请参阅参与者指南。

.NET feedback

The .NET documentation is open source. Provide feedback here.

提出文档问题

提供产品反馈

编译器警告（等级 1）CS0420

项目 • 2023/04/07

“identifier”：对 volatile 字段的引用不会视为 volatile

volatile 字段通常不应使用 `ref` 或 out 参数传递，因为在函数作用域内，它不会视为 volatile。 也有例外，如调用互锁 API 时。 对于任何警告，在有意将 volatile 字段用作引用参数的少数情况下，可以使用 #pragma warning 禁用此警告。

下面的示例生成 CS0420：

C#

```
// CS0420.cs
// compile with: /W:1
using System;

class TestClass
{
   private volatile int i;

   public void TestVolatile(ref int ii)
   {
   }

   public static void Main()
   {
      TestClass x = new TestClass();
      x.TestVolatile(ref x.i);   // CS0420
   }
}
```

编译器警告（等级 1）CS0465

项目 • 2023/04/07

引入“Finalize”方法会妨碍析构函数调用。 是否打算声明析构函数?

使用签名为 `public virtual void Finalize` 的方法创建类时，将出现此警告。

如果将这种类用作基类并且派生类定义终结器，则该终结器将重写基类的 `Finalize` 方法而不是 Finalize。

示例

下面的示例生成 CS0465。

C#

```
// CS0465.cs
// compile with: /target:library
class A
{
   public virtual void Finalize() {}   // CS0465
}

// OK
class B
{
   ~B() {}
}
```

编译器警告（等级 1）CS0602

项目 • 2023/04/07

功能“old_feature”已弃用。 请改为使用“new_feature”

虽然目前仍然支持 (*old_feature*) 代码中使用的语言功能，但在以后的版本中可能会删除该支持。 应转而使用建议的语法 (*new_feature*)。

> **备注**
>
> Roslyn 中不再使用此编译器警告。

编译器警告（等级 1）CS0612

项目 • 2023/04/07

“member” 已过时

类设计器使用 Obsolete 特性标记成员。 这意味着该类的未来版本中可能不支持该成员。

下面的示例显示访问过时的成员如何生成 CS0612：

C#

```
// CS0612.cs
// compile with: /W:1
using System;

class MyClass
{
   [Obsolete]
   public static void ObsoleteMethod()
   {
   }

   [Obsolete]
   public static int ObsoleteField;
}

class MainClass
{
   static public void Main()
   {
      MyClass.ObsoleteMethod();    // CS0612 here: method is deprecated
      MyClass.ObsoleteField = 0;   // CS0612 here: field is deprecated
   }
}
```

编译器警告（等级 1）CS0626

项目 • 2023/04/07

方法、运算符或访问器“method”被标记为 external，且其上不具有特性。 请考虑添加 DllImport 特性来指定外部实现。

标记为 `extern` 的方法也应使用特性（例如 DllImport 特性）进行标记。

该特性用于指定实现方法的位置。 在运行时，程序将需要此信息。

下面的示例生成 CS0626：

C#

```
// CS0626.cs
// compile with: /warnaserror
using System.Runtime.InteropServices;

public class MyClass
{
   static extern public void M(); // CS0626
   // try the following line
   // [DllImport("mydll.dll")] static extern public void M();

   public static void Main()
   {
   }
}
```

编译器警告（等级 1）CS0657

项目 • 2023/03/16

“attribute modifier”不是此声明的有效特性位置。 此声明的有效特性位置是“locations”。 此块中的所有特性都将被忽略。

编译器在无效的位置找到了特性修饰符。 有关详细信息，请参见 特性目标 。

下面的示例生成 CS0657：

C#

```
// CS0657.cs
// compile with: /target:library
public class TestAttribute : System.Attribute {}
[return: Test]   // CS0657 return not valid on a class
class Class1 {}
```

编译器警告（等级 1）CS0658

项目 • 2023/03/16

“特性修饰符”不是可识别的特性位置。 此块中的所有特性都将被忽略。

指定了无效的特性修饰符。 有关详细信息，请参见 特性目标 。

以下示例生成 CS0658:

C#

```
// CS0658.cs
using System;
public class TestAttribute : Attribute {}
[badAttributeLocation: Test]   // CS0658, badAttributeLocation is invalid
class ClassTest
{
   public static void Main()
   {
   }
}
```

编译器警告（等级 1）CS0672

项目 • 2023/04/07

成员“member1”重写过时的成员“member2”。 将 Obsolete 特性添加到“member1”

编译器找到方法标记为 `override` 的 `obsolete`。 但是，重写方法本身未标记为过时。 如果调用，则重写方法仍将生成 CS0612。

查看你的方法声明并显式指示方法（及其所有重写）是否应标记 `obsolete`。

下面的示例生成 CS0672：

C#

```
// CS0672.cs
// compile with: /W:1
class MyClass
{
   [System.Obsolete]
   public virtual void ObsoleteMethod()
   {
   }
}

class MyClass2 : MyClass
{
   public override void ObsoleteMethod()   // CS0672
   {
   }
}

class MainClass
{
   static public void Main()
   {
   }
}
```

编译器警告（等级 1）CS0684

项目 • 2023/04/07

“interface”接口标记为“CoClassAttribute”而不是“ComImportAttribute”

如果在接口上指定 **CoClassAttribute**， 则还必须指定 **ComImportAttribute**。

下面的示例生成 CS0684：

C#

```
// CS0684.cs
// compile with: /W:1
using System;
using System.Runtime.InteropServices;

[CoClass(typeof(C))] // CS0684
// try the following line instead
// [CoClass(typeof(C)), ComImport]
interface I
{
}

class C
{
   static void Main() {}
}
```

编译器警告（等级 1）CS0688

项目 • 2023/04/07

“method1”具有链接要求，但会重写或实现不具有链接要求“method2”。 可能存在安全漏洞。

可通过调用基类方法轻松绕过对派生类方法设置的此链接要求。 要填补此安全漏洞，基类方法需要同样使用此链接要求。 有关详细信息，请参阅 要求与LinkDemand。

示例

以下示例生成 CS0688。 若要解决此警告而不修改基类，请从重写方法中删除安全特性。 这不会解决安全问题。

C#

```
// CS0688.cs
// compile with: /W:1
using System;
using System.Security.Permissions;

class Base
{
    //Uncomment the following line to close the security hole
    //[FileIOPermission(SecurityAction.LinkDemand, All=@"C:\\")]
    public virtual void DoScaryFileStuff()
    {
    }
}

class Derived: Base
{
    [FileIOPermission(SecurityAction.LinkDemand, All=@"C:\\")] // CS0688
    public override void DoScaryFileStuff()
    {
    }
    static void Main()
    {
    }
}
```

编译器警告（等级 1）CS0809

项目 • 2024/08/15

过时成员“memberA”重写未过时成员“memberB”。

通常情况下，标记为过时的成员不应重写未标记为过时的成员。 此警告是在 Visual Studio 2008 中生成的，不是在 Visual Studio 2005 中生成的。

更正此错误

从重写成员删除 `Obsolete` 特性，或将其添加到基类成员。

示例

C#

```
// CS0809.cs
public class Base
{
    public virtual void Test1()
    {
    }
}
public class C : Base
{
    [System.Obsolete()]
    public override void Test1() // CS0809
    {
    }
}
```

被认定为过时的方法

请注意，当实际调用过时的方法时，编译器警告 `CS0809` 将导致没有 CS0618 警告。 在以下示例中，编译器不会警告方法 `Test` 已过时，因为编译器在调用时识别的声明位于基类 `Base` 中，而不是派生类 `Derived` 中：

C#

```
class Base
{
    public virtual void Test() {}
}
```

```
class Derived : Base
{
    [System.Obsolete()]
    public override void Test() { }
}

static class Program
{
    public static void Main()
    {
        Derived derived = new();
        b.Foo();    // No CS0618
    }
}
```

为了解决这个问题，还可以将 `Obsolete` 属性添加到基类中。

另请参阅

- CS0618
- CS0672

6 在 GitHub 上与我们协作

可以在 GitHub 上找到此内容的源，还可以在其中创建和查看问题和拉取请求。 有关详细信息，请参阅参与者指南。

.NET 反馈

.NET 是一个开放源代码项目。 选择一个链接以提供反馈：

- 提出文档问题
- 提供产品反馈

编译器警告（等级 1）CS1030

项目 • 2023/04/07

#warning：“text”

显示使用 #warning 指令定义的警告文本。

下面的示例演示如何创建用户定义的警告：

C#

```
// CS1030.cs
class Sample
{
   static void Main()
   {
      #warning Let's give a warning here
   }
}
```

编译产生以下输出：

控制台

```
example.cs(6,16): warning CS1030: #warning: 'Let's give a warning here'
```

编译器警告（等级 1）CS1058

项目 • 2023/04/07

前一 catch 子句已捕获了所有异常。 引发的所有异常都将包装在 System.Runtime.CompilerServices.RuntimeWrappedException 中

如果 `catch()` 程序块在 `catch (System.Exception e)` 程序块后没有指定的异常类型，此属性会产生 CS1058。 该警告建议 `catch()` 程序块将不捕获任何异常。

如果 AssemblyInfo.cs 文件中 `RuntimeCompatibilityAttribute` 设置为 false，则 `catch (System.Exception e)` 程序块后的 `catch()` 程序块可捕获非 CLS 异常：`[assembly: RuntimeCompatibilityAttribute(WrapNonExceptionThrows = false)]`。 如果此属性未显式设置为 false，则引发的所有非 CLS 异常都作为异常包装，并且 `catch (System.Exception e)` 程序块捕获它们。 有关详细信息，请参阅如何捕捉非 CLS 异常。

示例

下面的示例生成 CS1058。

C#

```
// CS1058.cs
// CS1058 expected
using System.Runtime.CompilerServices;

// the following attribute is set to true by default in the C# compiler
// set to false in your source code to resolve CS1058
[assembly: RuntimeCompatibilityAttribute(WrapNonExceptionThrows = true)]

class TestClass
{
   static void Main()
   {
      try {}

      catch (System.Exception e) {
         System. Console.WriteLine("Caught exception {0}", e);
      }

      catch {}   // CS1058. This line will never be reached.
   }
}
```

编译器警告（等级 1）CS1060

项目 • 2023/04/07

使用可能未赋值的字段“name”。 如果未对结构赋值，则结构实例变量最初未赋值。

如果未显式初始化结构成员，则将它们初始化为其默认值。 类类型（和其他引用类型）的默认值为 null。 如果尝试访问类之前，还未初始化类，则将在运行时引发 `NullReferenceException`。 编译器无法明确地确定是否会初始化类成员，因此 CS1060 是警告而不是错误。

更正此错误

1. 提供 `struct` 的构造函数，该构造函数初始化其所有成员。

示例

下面的代码生成 CS1060，因为类的类型 `U` 是 `struct S` 的成员，但从未进行过初始化。

C#

```
// cs1060.cs
namespace CS1060
{
    public class U
    {
        public int i;
    }

    public struct S
    {
        public U u;
        // Add constructor to correct the error.
        //public S(int val)
        //{
        //    u = new U() { i = val };
        //}
    }
    public class Test
    {
        static void Main()
        {
            S s;
            s.u.i = 5;  // CS1060

            //Try these lines instead, and uncomment the constructor in S
            // S s = new S(0);
```

```
            // s.u.i = 5;

        }
    }
}
```

请参阅

- 结构类型

编译器警告（等级 1）CS1200

项目 • 2023/04/07

功能“invalid feature”已弃用。 请改用“valid feature”。

尝试使用的功能现已弃用。 请更新代码，以改用有效的功能。

编译器警告（等级 1）CS1201

项目 • 2023/04/07

功能“invalid feature”已弃用。 请使用“valid feature”。

尝试使用的功能现已弃用。 请更新代码，以改用有效的功能。

编译器警告（等级 1）CS1202

项目 • 2023/04/07

功能“invalid feature”已弃用。 请改用“valid feature”。

尝试使用的功能现已弃用。 请更新代码，以改用有效的功能。

编译器警告（等级 1）CS1203

项目 • 2023/04/07

功能“feature”已弃用。 请改用“feature”。

功能“invalid feature”已弃用。 请改用“valid feature”。

尝试使用的功能现已弃用。 请更新代码，以改用有效的功能。

编译器警告（等级 1）CS1522

项目 • 2023/04/07

空的 switch 块

编译器检测到 switch 块没有 **case** 或 **default** 语句。 `switch` 块必须具有一个或多个 **case** 或 **default** 语句。

下面的示例生成 CS1522：

C#

```
// CS1522.cs
// compile with: /W:1
using System;
class x
{
   public static void Main()
   {
      int i = 6;

      switch(i)   // CS1522
      {
         // add something to the switch block, for example:
         /*
         case (5):
            Console.WriteLine("5");
            return;
         default:
            Console.WriteLine("not 5");
            return;
         */
      }
   }
}
```

编译器警告（等级 1）CS1570

项目 • 2023/04/07

“construct”上的 XML 注释出现 XML 格式错误 —“reason”

当使用 DocumentationFile 时，源代码中的任何注释必须为 XML 格式。 XML 标记的任何错误将生成 CS1570。 例如：

- 如果要传递一个字符串到 cref，例如，在<异常>标记中，字符串必须括在双引号内。
- 如果正在使用没有结尾标记的标记（如<另请参阅>），必须在右尖括号的前面指定一个正斜杠。
- 如果你需要在说明文本使用大于或小于符号，你需要用 `>` 或 `<` 来表示它们。或者，你可使用 CDATA。
- <包括>标记上的文件或路径特性缺失或格式不正确。

以下示例生成 CS1570：

C#

```
public static class CompareFive
{
    // the following line generates CS1570
    /// <summary> returns true if < 5 </summary>
    // try one of the following instead
    // /// <summary> returns true if &lt; 5 </summary>
    // /// <summary><![CDATA[ returns true if < 5 ]]></summary>
    public static bool LessThanFive(int x) => x < 5;
}
```

编译器警告（等级 1）CS1574

项目 • 2023/04/07

“construct”的 XML 注释中有语法错误的 cref 特性“name”

传递给 cref 标记的字符串（例如在 <exception> 标记中）引用了在当前生成环境中不可用的成员。 传递给 cref 标记的字符串必须是语法正确的成员或字段名称。

有关详细信息，请参阅 建议的文档注释标记。

下面的示例生成 CS1574：

C#

```
// CS1574.cs
// compile with: /W:1 /doc:x.xml
using System;

/// <exception cref="System.Console.WriteLin">An exception class.
</exception>   // CS1574
// instead, uncomment and try the following line
// /// <exception cref="System.Console.WriteLine">An exception class.
</exception>
class EClass : Exception
{
}

class TestClass
{
   public static void Main()
   {
      try
      {
      }
      catch(EClass)
      {
      }
   }
}
```

编译器警告（等级 1）CS1580

项目 • 2023/04/07

XML 注释 cref 特性中参数“parameter number”的类型无效

当试图引用某个方法的重载格式时，编译器检测到了语法错误。 通常情况下，这指示指定了参数名称，而不是类型。 在生成的 XML 文件中将显示格式不正确的行。

下面的示例生成 CS1580：

C#

```
// CS1580.cs
// compile with: /W:1 /doc:x.xml
using System;

/// <seealso cref="Test(i)"/>   // CS1580
// try the following line instead
// /// <seealso cref="Test(int)"/>
public class MyClass
{
   /// <summary>help text</summary>
   public static void Main()
   {
   }

   /// <summary>help text</summary>
   public void Test(int i)
   {
   }

   /// <summary>help text</summary>
   public void Test(char i)
   {
   }
}
```

编译器警告（等级 1）CS1581

项目 • 2023/04/07

XML 注释的 cref 特性中的返回类型无效

当试图引用方法时，由于返回类型无效，编译器检测到错误。

示例

下面的示例生成 CS1581：

C#

```
// CS1581.cs
// compile with: /W:1 /doc:x.xml

/// <summary>help text</summary>
public class MyClass
{
    /// <summary>help text</summary>
    public static void Main()
    {
    }

    /// <summary>help text</summary>
    public static explicit operator int(MyClass f)
    {
        return 0;
    }
}

/// <seealso cref="MyClass.explicit operator intt(MyClass)"/>  // CS1581
// try the following line instead
// /// <seealso cref="MyClass.explicit operator int(MyClass)"/>
public class MyClass2
{
}
```

编译器警告（等级 1）CS1584

项目 • 2023/04/07

“member”的 XML 注释中出现语法错误的 cref 特性“invalid_syntax”

传递给文档注释标记的其中一个参数的语法无效。 有关详细信息，请参阅 建议的文档注释标记。

示例

下面的示例生成 CS1584。

C#

```
// CS1584.cs
// compile with: /W:1 /doc:CS1584.xml
/// <remarks>Test class</remarks>
public class Test
{
   /// <remarks>Called in <see cref="Test.Mai()n"/>.</remarks>   // CS1584
   // try the following line instead
   // /// <remarks>Called in <see cref="Test.Main()"/>.</remarks>
   public static void Test2() {}

   /// <remarks>Main method</remarks>
   public static void Main() {}
}
```

编译器警告（等级 1）CS1589

项目 • 2023/04/07

无法包含文件“file”的 XML 片段“fragment” -- 原因

出于特定原因，<include> 标记的语法（段落）（引用了一个文件 (`file`)）不正确。

将在生成的 XML 文件中放置格式不正确的行。

下面的示例生成 CS1589：

C#

```
// CS1589.cs
// compile with: /W:1 /doc:CS1589_out.xml

/// <include file='CS1589.doc' path='MyDocs/MyMembers[@name="test"]/' />
// CS1589
// try the following line instead
// /// <include file='CS1589.doc' path='MyDocs/MyMembers[@name="test"]/*' />
class Test
{
   public static void Main()
   {
   }
}
```

编译器警告（等级 1）CS1590

项目 • 2023/04/07

XML 包含元素无效 -- 缺少文件特性

缺少传递给 <include> 标记的 path 或 doc 特性，或特性不完整。

下面的示例生成 CS1590：

C#

```
// CS1590.cs
// compile with: /W:1 /doc:x.xml

/// <include path='MyDocs/MyMembers[@name="test"]/*' />   // CS1590
// try the following line instead
// /// <include file='x.doc' path='MyDocs/MyMembers[@name="test"]/*' />
class Test
{
   public static void Main()
   {
   }
}
```

编译器警告（等级 1）CS1592

项目 • 2023/04/07

所包含的注释文件中有格式错误的 XML --“reason”

在由 <include> 标记指定的文件中发现了报告为原因的问题。

编译器警告（等级 1）CS1598

项目 • 2023/04/07

无法加载 XML 分析程序，原因如下：“reason”。 不会生成 XML 文档文件“file”。

已指定 DocumentationFile 选项，但编译器无法找到并加载 msxml3.dll。 请确保已安装并注册文件 msxml3.dll。

编译器警告（等级 1）CS1607

项目 • 2023/06/13

程序集生成 - 原因

在编译的程序集创建阶段生成了一个警告。

若要在 32 位操作系统上生成 64 位应用程序，必须确保在目标操作系统上安装了所有引用的程序集的 64 位版。

所有特定于 x86 的公共语言运行时 (CLR) 程序集都具有 64 位对应项（每个 CLR 程序集都将存在于所有操作系统上）。 因此，可以安全地忽略 CLR 程序集的 CS1607。

如果在创建 AssemblyInformationalVersionAttribute 时遇到此警告，则可以忽略此警告。 信息性版本是一个字符串，它向一个程序集附加额外的版本信息；此信息不在运行时使用。 虽然可以指定任意文本，但是如果字符串的格式不是程序集版本号使用的格式，或者虽然是这种格式但包含通配符，则在编译时会显示一条警告消息。 此警告无碍。

有关详细信息，请参阅 Al.exe 工具错误和警告。

编译器警告（等级 1）CS1616

项目 • 2023/04/07

选项“option”重写源文件或添加的模块中给出的属性“attribute”

如果源中的程序集属性 AssemblyKeyFileAttribute 或 AssemblyKeyNameAttribute 与项目属性中指定的 **KeyFile** 或 **KeyContainer** 命令行选项或密钥文件名或密钥容器冲突，则会出现此警告。

对于下面的示例，假定有一个名为 `cs1616.snk` 的密钥文件。 使用如下命令行生成此文件：

控制台

```
sn -k CS1616.snk
```

下面的示例生成 CS1616：

C#

```
// CS1616.cs
// compile with: /keyfile:cs1616.snk
using System.Reflection;

// To fix the error, remove the next line
[assembly: AssemblyKeyFile("cs1616b.snk")]  // CS1616

class C
{
  public static void Main()
  {
  }
}
```

编译器警告（等级 1）CS1633

项目 • 2023/04/07

无法识别的 #pragma 指令

使用的杂注不是 C# 编译器支持的已知杂注之一。 若要解决此错误，请仅使用受支持的杂注。

以下示例生成 CS1633：

C#

```
// CS1633.cs
// compile with: /W:1
#pragma unknown  // CS1633

class C
{
   public static void Main()
   {
   }
}
```

编译器警告（等级 1）CS1634

项目 • 2023/04/07

应为 disable 或 restore

如果 #pragma 警告子句格式有误，例如忽略了禁用或还原，将出现此错误。 有关详细信息，请参阅 #pragma warning 主题。

示例

下面的示例生成 CS1634：

C#

```
// CS1634.cs
// compile with: /W:1

#pragma warning   // CS1634
// Try this instead:
// #pragma warning disable 0219

class MyClass
{
  public static void Main()
  {
  }
}
```

编译器警告（等级 1）CS1635

项目 • 2023/04/07

“警告代码”警告已全局禁用，无法还原

如果你使用 **/nowarn** 命令行选项或项目设置在整个编译单元中禁用某一警告，而又使用 `#pragma warning restore` 尝试还原该警告，则会出现此警告。 若要解决此错误，对于通过命令行或项目设置禁用的所有警告，请删除其 /nowarn 命令行选项或项目设置，或删除 `#pragma warning restore` 。 有关详细信息，请参阅 #pragma warning。

下面的示例生成 CS1635：

C#

```
// CS1635.cs
// compile with: /w:1 /nowarn:162

enum MyEnum {one=1,two=2,three=3};

class MyClass
{
    public static void Main()
    {
#pragma warning disable 162

    if (MyEnum.three == MyEnum.two)
        System.Console.WriteLine("Duplicate");

#pragma warning restore 162
    }
}
```

编译器警告（等级 1）CS1645

项目 • 2023/04/07

功能“功能”不是标准化 ISO C# 语言规范的一部分，其他编译器可能不接受它

正在使用的功能不是 ISO 标准的一部分。 使用此功能的代码可能不会在其他编译器编译。

C#

```
// CS1645.cs
// compile with: /W:1 /t:module /langversion:ISO-1
[assembly:System.CLSCompliant(false)]
// To suppress the warning use the switch: /nowarn:1645
[module:System.CLSCompliant(false)]   // CS1645
class Test
{
}
```

编译器警告（等级 1）CS1658

项目 • 2023/04/07

“warning text”。 另请参阅错误“error code”

编译器使用警告替代错误时，会发出此警告。 有关该问题的信息，请参阅所述错误。 若要从 Visual Studio IDE 中查找相应的错误，请使用索引。 例如，如果上面的文本为“另请参阅错误‘CS1037’”，则在索引中查找 CS1037。

示例

下面的示例生成 CS1658。

C#

```
// CS1658.cs
// compile with: /doc:x.xml
// CS1584 expected
/// <summary>
/// </summary>
public class C
{
    /// <see cref="C.F(params object[])"/>  // CS1658
    public static void M()
    {
    }

    /// <summary>
    /// </summary>
    public void F(params object[] o)
    {
    }

    static void Main()
    {
    }
}
```

编译器警告（等级 1）CS1682

项目 • 2023/04/07

对类型“type”的引用声称该类型嵌套在“nested type”中，但并未找到该类型

当导入的引用与其他引用或者编写的代码不一致时，会发生此错误。 发生此错误通常是因为编写了在元数据中引用类的代码，而此后删除了该类或修改了该类的定义。

示例 1

C#

```
// CS1682.cs
// compile with: /target:library /keyfile:mykey.snk
public class A {
   public class N1 {}
}
```

示例 2

C#

```
// CS1682_b.cs
// compile with: /target:library /reference:CS1682.dll
using System;
public class Ref {

   public static A A1() {
      return new A();
   }

   public static A.N1 N1() {
      return new A.N1();
   }
}
```

示例 3

C#

```
// CS1682_c.cs
// compile with: /target:library /keyfile:mykey.snk /out:CS1682.dll
public class A {
```

```
    public void M1() {}
}
```

示例 4

下面的示例生成 CS1682。

C#

```
// CS1682_d.cs
// compile with: /reference:CS1682.dll /reference:CS1682_b.dll /W:1
// CS1682 expected
class Tester {
   static void Main()
   {
      Ref.A1().M1();
   }
}
```

编译器警告（等级 1）CS1684

项目 • 2023/04/07

对类型“Type Name”的引用声称该类型是在“Namespace”中定义的，但未能找到它

在一个命名空间内引用据称位于另一命名空间内而实际不存在的类型可能导致此错误。 例如，mydll.dll 称类型 `A` 存在于 yourdll.dll 中，但 yourdll.dll 中不存在这样的类型。 该错误的一个可能原因是你使用的 yourdll.dll 版本过旧而尚未定义 `A` 。

下面的示例生成 CS1684。

示例 1

C#

```
// CS1684_a.cs
// compile with: /target:library /keyfile:CS1684.key
public class A {
   public void Test() {}
}

public class C2 {}
```

示例 2

C#

```
// CS1684_b.cs
// compile with: /target:library /r:cs1684_a.dll
// post-build command: del /f CS1684_a.dll
using System;
public class Ref
{
   public static A GetA() { return new A(); }
   public static C2 GetC() { return new C2(); }
}
```

示例 3

我们现在重新生成第一个程序集，在重新编译中不再定义类 C2。

C#

```
// CS1684_c.cs
// compile with: /target:library /keyfile:CS1684.key /out:CS1684_a.dll
public class A {
   public void Test() {}
}
```

示例 4

此模块通过使用标识符 `Ref` 引用第二个模块。 但是，第二个模块包含对类 `C2` 的引用，而由于上一步骤中的编译该类已不再存在，因此此模块的编译返回了 CS1684 错误消息。

C#

```
// CS1684_d.cs
// compile with: /reference:cs1684_a.dll /reference:cs1684_b.dll
// CS1684 expected
class Tester
{
   public static void Main()
   {
      Ref.GetA().Test();
   }
}
```

编译器警告（等级 1）CS1685

项目 • 2023/05/10

预定义类型“System.type name”是在全局别名的多个程序集中定义的；请使用“文件名”中的定义

在两个程序集中都找到了 System.int32 等预定义系统类型时，会出现此错误。 发生这种情况的方法之一是正在从两个不同的位置引用 mscorlib，例如尝试并行运行 .NET Framework 1.0 版和 1.1 版。

编译器将使用仅来自其中一个程序集的定义。 编译器仅搜索全局别名，不搜索定义 /reference 的库。 如果已指定 /nostdlib，则编译器将搜索 Object，并且将来在找到 Object 的文件中启动针对预定义类型的所有搜索。

编译器警告（等级 1）CS1687

项目 • 2023/04/07

源文件已超过在 PDB 中可表示的 16,707,565 行的限制；调试信息将不正确

PDB 和调试器有一些关于文件大小的限制。 如果源文件太大，在超过该限制的情况下，调试器将不能正常工作。 用户不应发出该文件的调试信息（可能通过使用 `#line hidden`），或者应找到收缩文件的方法（可能通过将该文件拆分为多个文件）。 他们可能想要使用 `partial` 关键字来拆分大类。

编译器警告（等级 1）CS1690

项目 • 2023/04/07

访问“member”上的成员会导致运行时异常，因为它是引用封送类的字段

尝试调用派生自 MarshalByRefObject 的类的成员上的方法、属性或索引器且成员是值类型时，会出现此警告。 在应用程序域中，继承自 `MarshalByRefObject` 的对象通常打算以引用方式封送。 如果任何代码曾尝试在应用程序域中直接访问此类对象的值类型成员，会发生运行时 InvalidOperationException。 要解决此警告，首先将该成员复制到本地变量中，然后在该变量上调用方法。

下面的示例生成 CS1690：

C#

```
// CS1690.cs
using System;

class WarningCS1690 : MarshalByRefObject
{
    int i = 5;

    public static void Main()
    {
        AppDomain domain = AppDomain.CreateDomain("MyDomain");
        Type t = typeof(WarningCS1690);
        WarningCS1690 e =
(WarningCS1690)domain.CreateInstanceAndUnwrap(t.Assembly.FullName,t.FullName
);

        e.i.ToString(); // CS1690

        // OK
        int i = e.i;
        i.ToString();
        e.i = i;
    }
}
```

编译器警告（等级 1）CS1691

项目 • 2023/04/07

“number”是无效的警告编号

传递到 #pragma warning 预处理器指令的数字不是有效的警告编号。 验证该数字表示警告，而不是错误或其他字符序列。

示例

下面的示例生成 CS1691。

C#

```
// CS1691.cs
public class C
{
    int i = 1;
    public static void Main()
    {
        C myC = new C();
#pragma warning disable 151  // CS1691
// Try the following line instead:
// #pragma warning disable 1645
        myC.i++;
#pragma warning restore 151  // CS1691
// Try the following line instead:
// #pragma warning restore 1645
    }
}
```

编译器警告（等级 1）CS1692

项目 • 2023/04/07

无效数字

多项预处理器指令（如 `#pragma` 和 `#line`）使用数字作为参数。 其中一个数字无效，原因是过大、格式错误、包含非法字符等。 若要更正此错误，请更正该数字。

示例

以下示例生成 CS1692。

C#

```
// CS1692.cs

#pragma warning disable a  // CS1692
// Try this instead:
// #pragma warning disable 1691

class A
{
    static void Main()
    {
    }
}
```

编译器警告（等级 1）CS1694

项目 • 2023/04/07

为预处理器指令指定的文件名无效。 文件名太长或文件名无效。

使用 `#pragma checksum` 预处理器指令时会出现此警告。 指定文件名的长度超过 256 个字符。 要解决此警告，请使用较短的文件名。

示例

下面的示例生成 CS1694。

C#

```
// cs1694.cs
#pragma checksum
"MyFile1234567890MyFile1234567890MyFile1234567890MyFile1234567890MyFile12345
67890MyFile1234567890MyFile1234567890MyFile1234567890MyFile1234567890MyFile1
234567890MyFile1234567890MyFile1234567890MyFile1234567890MyFile1234567890MyF
ile1234567890MyFile1234567890.txt" {00 01 02 03 04 05 06 07 08 09 0A 0B 0C
0D 0E 0F}   // CS1694
class MyClass {}
```

编译器警告（等级 1）CS1695

项目 • 2023/04/07

无效的 #pragma checksum 语法；应为 #pragma checksum "filename" "{XXXXXXXX-XXXX-XXXX-XXXX-XXXXXXXXXXXX}" "XXXX..."

你应该很少遇到此错误，因为如果通过 Code Dom API 生成代码，校验和通常在运行时插入。

但是，如果要在此 `#pragma` 语句中键入内容，并且错误地键入了 GUID 或校验和，则会遇到此错误。 编译器的语法检查并不验证你是否键入了正确的 GUID，但它会检查位数和分隔符数量是否正确，以及这些数字是否为十六进制。 同样，它还验证校验和是否包含偶数位数，以及这些数字是否为十六进制。

示例

下面的示例生成 CS1695。

C#

```
// CS1695.cs

#pragma checksum "12345"  // CS1695

public class Test
{
    static void Main()
    {
    }
}
```

编译器警告（等级 1）CS1696

项目 • 2023/04/07

应输入单行注释或行尾

编译器需要预处理器指令后跟行尾终止符或者单行注释。 编译器已完成有效预处理器指令的处理，并遇到了违反此语法约束的问题。

示例

下面的示例生成 CS1696。

C#

```
// CS1696.cs
class Test
{
   public static void Main()
   {
      #pragma warning disable 1030;219   // CS1696
      #pragma warning disable 1030   // OK
   }
}
```

编译器警告（等级 1）CS1697

项目 • 2023/04/07

为“file name”提供了不同的校验和值

为给定文件指定了一个以上的校验和。 当一个项目中有一个以上的同名文件时，调试器会使用校验和值确定调试哪个文件。 大多数用户不会遇到此错误，但是如果你正在编写生成代码的应用程序，则可能会偶然遇到此错误。 若要解决此错误，请确保对任何给定代码文件仅生成一次校验和。

编译器警告（等级 1）CS1699

项目 • 2023/04/07

使用命令行选项“compiler_option”或适当的项目设置代替“attribute_name”

若要为程序集签名，需要指定密钥文件。 在 C# 2.0 之前，在源代码中使用 CLR 属性指定密钥文件。 这些属性现已弃用。

从 C# 2.0 开始，应使用“项目设计器”的“签名页”或程序集链接器来指定密钥文件。

“项目设计器”的“签名”页是首选方法；有关详细信息，请参阅“项目设计器”->“签名”页和管理程序集和清单签名。

如何：使用强名称为程序集签名使用以下编译器选项：

- KeyFile（C# 编译器选项）而不是 AssemblyKeyFileAttribute 属性。
- KeyContainer（C# 编译器选项）而不是 AssemblyKeyNameAttribute。
- DelaySign（C# 编译器选项）而不是 AssemblyDelaySignAttribute。

这些属性由于以下原因已启用：

- 由于属性嵌入在由编译器生成的二进制文件中，因此存在安全问题。 拥有你的二进制文件的所有人也会拥有在其中存储的密钥。
- 由于属性中指定的路径相对于当前工作目录（可能会在集成开发环境 (IDE) 中更改）或输出目录，因此存在可用性问题。 因此，大多数情况下密钥文件有可能是 ..\\..\\mykey.snk。 属性也使项目系统更加难以为附属程序集正确签名。 使用编译器选项而不是这些属性时，可以使用密钥的完全限定路径和文件名而不会在输出文件中嵌入任何内容；项目系统和源代码管理系统可以在项目四处移动时正确操作该完整路径；项目系统可以维护密钥文件的项目相对路径，并仍然将完整路径传递给编译器；其他生成程序可以通过直接向编译器传递正确路径而不是生成具有正确属性的源文件，更轻松地为输出签名。
- 将属性与友元程序集结合使用可能会影响编译器效率。 使用属性时，编译器在必须决定是否授予友元关系时不知道该密钥是什么，因此必须进行猜测。 编译结束时，编译器一旦最终知道密钥，便能够验证猜测。 使用编译器选项指定密钥文件时，编译器可以立即决定是否授予友元关系。

示例

以下示例生成 CS1699。 若要解决该错误，请移除属性，并使用 **/delaysign** 进行编译。

C#

```
// CS1699.cs
// compile with: /target:library
[assembly:System.Reflection.AssemblyDelaySign(true)]   // CS1699
```

另请参阅

- “项目设计器”->“签名”页
- 管理程序集签名和清单签名
- 如何：使用强名称为程序集签名

编译器警告（等级 1）CS1707

项目 • 2023/04/07

由于新的语言规则，委托“DelegateName”绑定到“MethodName1”而非“MethodName2”

C# 2.0 实现用于将委托绑定到方法的新规则。 现在将考虑以前看不到的其他信息。 此警告指示委托现在被绑定到与之前不同的方法重载。 你可能希望验证该委托实际上应绑定到“MethodName1”而不是“MethodName2”。

有关编译器如何确定委托将绑定到的方法的描述，请参阅在委托中使用变体。

编译器警告（等级 1）CS1709

项目 • 2023/04/07

为预处理器指令指定的文件名为空

你指定了包含文件名的预处理器指令，但该文件为空。 若要解决此警告，请将所需内容放入该文件。

示例

下面的示例生成 CS1709。

C#

```
// CS1709.cs
class Test
{
    static void Main()
    {
        #pragma checksum "" "{406EA660-64CF-4C82-B6F0-42D48172A799}" ""  //
CS1709
    }
}
```

编译器警告（等级 1）CS1720

项目 • 2023/04/07

由于“泛型类型”的默认值为 null，因此表达式总会导致 System.NullReferenceException

如果编写的表达式涉及作为引用类型（例如类）的泛型类型变量的默认值，则会发生此错误。 考虑下面的表达式：

C#

```
default(T).ToString()
```

由于 `T` 是引用类型，因此其默认值是 null，并且因此尝试将 ToString 方法应用于它会引发 NullReferenceException。

示例

类型 `T` 上的类引用约束可确保 `T` 是引用类型。

下面的示例生成 CS1720。

C#

```
// CS1720.cs
using System;
public class Tester
{
    public static void GenericClass<T>(T t1) where T : class
    {
        Console.WriteLine(default(T).ToString());  // CS1720
    }
    public static void Main() {}
}
```

编译器警告（等级 1）CS1723

项目 • 2023/03/23

XML 注释中有引用类型参数的 cref 特性“attribute”

如果在代码中使用含有对类型参数而非现有类型（无论是用户定义的，还是内置的）的交叉引用 (cref) 的 <see/> 标记，则会为 XML 注释生成此错误。 无法链接到泛型类型的“attribute”，因为在创建文档时，未来指定为“attribute”的类型尚不得而知。

若要解决此问题，应使用 <typeparamref/> 标记。

示例

以下示例包含生成 CS1723 的注释以及可以正确链接的引用。

C#

```
public class Point
{
}

// compile with: /t:library /doc:filename.XML
///<summary>A generic list class.</summary>
///uses <see cref="T" />      // CS1723
///and <see cref="Point" />   // No warning
public class List<T, Point>
{
}
```

此示例展示了如何正确链接泛型类型 `T` 和已知的用户定义的 `Point`

C#

```
public class Point
{
}

// compile with: /t:library /doc:filename.XML
///<summary>A generic list class.</summary>
///uses <typeparamref name="T" />  // No warning
///and <see cref="Point" />        // No warning
public class List<T, Point>
{
}
```

编译器警告（等级 1）CS1762

项目 • 2023/04/07

创建了对嵌入的互操作程序集“<assembly1>”的引用，因为程序集“<assembly2>”间接引用了该程序集。 请考虑更改任一程序集上的“Embed Interop Types”属性。

已添加对 `Embed Interop Types` 属性设置为 `True` 的程序集 (assembly1) 的引用。 这指示编译器嵌入来自该程序集的互操作类型信息。 但是，编译器不能嵌入来自该程序集的互操作类型信息，因为引用的另一个程序集 (assembly2) 也引用该程序集 (assembly1)，并将 `Embed Interop Types` 属性设置为 `False`。

> **备注**
>
> 将程序集引用的 `Embed Interop Types` 属性设置为 `True`，其效果等同于使用命令行编译器的 `/link` 选项引用该程序集。

解决此警告

- 要嵌入这两个程序集的互操作类型信息，请将对 assembly1 的所有引用的 `Embed Interop Types` 属性设置为 `True`。 有关如何设置该属性的详细信息，请参阅演练：嵌入托管程序集中的类型。
- 若要删除警告，可以将 assembly1 的 `Embed Interop Types` 属性设置为 `False`。 在这种情况下，互操作类型信息由主互操作程序集 (PIA) 提供。

请参阅

- EmbedInteropAssembly（C# 编译器选项）
- 与非托管代码交互操作

编译器警告（等级 1）CS1911

项目 • 2023/04/07

通过“base”关键字从匿名方法、lambda 表达式、查询表达式或迭代器访问成员“name”会导致无法验证的代码。 请考虑在包含类型上移入一个帮助程序方法的访问权限。

在迭代器或匿名方法的方法主体内使用 `base` 关键字调用虚函数将导致无法验证的代码。 无法验证的代码将无法在部分信任环境中运行。

CS1911 的一个解决方法是将虚函数调用移到帮助程序函数中。

示例

下面的示例生成 CS1911。

C#

```
// CS1911.cs
// compile with: /W:1
using System;

delegate void D();
delegate D RetD();

class B {
   protected virtual void M() {
      Console.WriteLine("B.M");
   }
}

class Der : B {
   protected override void M() {
      Console.WriteLine("D.M");
   }

   void Test() { base.M(); }
   D Test2() { return new D(base.M); }

   public D CallBaseM() {
      return delegate () { base.M(); };   // CS1911

      // try the following line instead
      // return delegate () { Test(); };
   }

   public RetD DelToBaseM() {
      return delegate () { return new D(base.M); };   // CS1911

      // try the following line instead
```

```
        // return delegate () { return Test2(); };
    }
}

class Program {
    public static void Main() {
        Der der = new Der();
        D d = der.CallBaseM();
        d();
        RetD rd = der.DelToBaseM();
        rd()();
    }
}
```

编译器警告（等级 1）CS1956

项目 • 2023/04/07

成员“name”实现类型“type”中的接口成员“name”。 接口成员在运行时具有多个匹配项。 其实现取决于将要调用的方法。

如果两个接口方法的唯一区别在于某个特定参数由 `ref` 还是 `out` 标记，则可生成此警告。 最好更改代码以避免此警告，因为在运行时调用哪个方法并不十分明显，或者无法保证在运行时调用哪个方法。

尽管 C# 可以区分 `out` 和 `ref`，但 CLR 却将它们视为相同的。 确定实现该接口的方法时，CLR 仅选择一个。

避免此警告

1. 为编译器提供某种方式来区分这些方法。 例如，可以为这两个方法提供不同的名称，或者为其中一个方法提供附加参数。

示例

下面的代码将生成 CS1956，因为 `Base` 中这两个 `Test` 方法的唯一区别在于它们的第一个参数上的 `ref`/`out` 修饰符。

C#

```
// cs1956.cs
class Base<T, S>
{
    // This is the method that should be called.
    public virtual int Test(out T x) // CS1956
    {
        x = default(T);
        return 0;
    }

    // This is the "last" method and is the one that ends up being called
    public virtual int Test(ref S x)
    {
        return 1;
    }
}

interface IFace
{
    int Test(out int x);
```

```
}

class Derived : Base<int, int>, IFace
{
    static int Main()
    {
        IFace x = new Derived();
        int y;
        return x.Test(out y);
    }
}
```

编译器警告（等级 1）CS1957

项目 • 2023/04/07

成员“name”重写“method”。 在运行时有多个重写候选项。 其实现取决于将要调用的方法。

仅因是 `ref` 还是 `out` 而异的方法参数在运行时无法区分。

避免此警告

1. 为方法之一给定其他名称或其他数目的参数。

示例

下面的代码生成 CS1957：

C#

```
// cs1957.cs
class Base<T, S>
{
    public virtual string Test(out T x) // CS1957
    {
        x = default(T);
        return "Base.Test";
    }
    public virtual void Test(ref S x) { }
}

class Derived : Base<int, int>
{
    public override string Test(out int x)
    {
        x = 0;
        return "Derived.Test";
    }

    static int Main()
    {
        int x;
        if (new Derived().Test(out x) == "Derived.Test")
            return 0;
        return 1;
    }
}
```

编译器警告（等级 1）CS2002

项目 • 2023/04/07

多次指定源文件“file”

已多次将源文件名称传递到编译器。 你只能对编译器指定一次文件来生成输出文件。

-nowarn 选项不能禁止显示此警告。

以下示例生成 CS2002：

C#

```
// CS2002.cs
// compile with: CS2002.cs
public class A
{
public static void Main(){}
}
```

若要生成该错误，请使用命令行编译示例：

控制台

```
csc CS2002.cs CS2002.cs
```

编译器警告（等级 1）CS2014

项目 • 2023/04/07

编译器选项“old option”已过时，请使用“new option”

编译器选项的格式被弃用。 有关详细信息，请参阅 C# 编译器选项 。

编译器警告（等级 1）CS2023

项目 • 2023/04/07

/noconfig 选项是在响应文件中指定的，因此被忽略

已在响应文件中指定 -noconfig 编译器选项，这是不允许的。

NoWarn 选项不能禁止显示此警告。

编译器警告（等级 1）CS2029

项目 • 2023/04/07

“/define”的值无效；“identifier”不是有效的标识符

如果用在 DefineConstants 选项中的值存在一些无效字符，将出现此警告。

NoWarn 选项不能禁止显示此警告。

编译器警告（等级 1）CS3000

项目 • 2023/04/07

带有变量参数的方法不符合 CLS

此方法中使用的参数公开了不属于公共语言规范 (CLS) 的功能。 有关 CLS 符合性的详细信息，请参阅语言独立性和与语言无关的组件。

下面的示例生成警告 CS3000。

C#

```
// CS3000.cs
// compile with: /target:library
// CS3000 expected
[assembly:System.CLSCompliant(true)]

public class Test
{
   public void AddABunchOfInts( __arglist ) {}   // CS3000
}
```

编译器警告（等级 1）CS3001

项目 • 2023/04/07

参数类型“type”不符合 CLS

public、 protected、 or `protected internal` 方法必须接受其类型符合公共语言规范 (CLS) 的参数。 有关 CLS 符合性的详细信息，请参阅语言独立性和与语言无关的组件。

示例

下面的示例生成 CS3001:

C#

```
// CS3001.cs

[assembly:System.CLSCompliant(true)]
public class a
{
    public void bad(ushort i)   // CS3001
    {
    }

    private void OK(ushort i)   // OK, private method
    {
    }

    public static void Main()
    {
    }
}
```

编译器警告（等级 1）CS3002

项目 • 2023/04/07

“method”的返回类型不符合 CLS

public、 protected、 or `protected internal` 方法返回的值的类型必须符合公共语言规范 (CLS)。 有关 CLS 符合性的详细信息，请参阅语言独立性和与语言无关的组件。

示例

下面的示例生成 CS3002：

C#

```
// CS3002.cs

[assembly:System.CLSCompliant(true)]
public class a
{
    public ushort bad()   // CS3002, public method
    {
        ushort a;
        a = ushort.MaxValue;
        return a;
    }

    private ushort OK()   // OK, private method
    {
        ushort a;
        a = ushort.MaxValue;
        return a;
    }

    public static void Main()
    {
    }
}
```

编译器警告（等级 1）CS3003

项目 • 2023/04/07

“变量”的类型不符合 CLS

public、protected 或 `protected internal` 变量的类型必须符合公共语言规范 (CLS)。 有关 CLS 符合性的详细信息，请参阅语言独立性和与语言无关的组件。

示例

下面的示例生成 CS3003：

C#

```
// CS3003.cs

[assembly:System.CLSCompliant(true)]
public class a
{
    public ushort a1;   // CS3003, public variable
    public static void Main()
    {
    }
}
```

编译器警告（等级 1）CS3004

项目 • 2023/04/07

混合的和分解的 Unicode 字符不符合 CLS

public、 protected、 or `protected internal` 标识符中只可使用组合的 UNICODE 字符，以便符合公共语言规范 (CLS)。 有关 CLS 符合性的详细信息，请参阅语言独立性和与语言无关的组件。

编译器警告（等级 1）CS3005

项目 • 2023/04/07

仅大小写不同的标识符“identifier”不符合 CLS

公共、受保护或 `protected internal` 标识符仅一个或多个字母与另一个 `public`、`protected` 或 `protected internal` 标识符不同，不符合公共语言规范 (CLS)。 有关 CLS 符合性的详细信息，请参阅语言独立性和与语言无关的组件。

示例

下面的示例生成 CS3003：

C#

```
// CS3005.cs

using System;

[assembly:CLSCompliant(true)]
public class a
{
    public static int a1 = 0;
    public static int A1 = 1;   // CS3005

    public static void Main()
    {
        Console.WriteLine(a1);
        Console.WriteLine(A1);
    }
}
```

编译器警告（等级 1）CS3008

项目 • 2023/04/07

仅大小写不同的标识符“identifier”不符合 CLS

如果以下划线字符 (_) 开头，则一个公共、受保护或 `protected internal` 标识符违反公共语言规范 (CLS)。 有关 CLS 符合性的详细信息，请参阅语言独立性和与语言无关的组件。

示例

以下示例生成 CS3008：

C#

```
// CS3008.cs

using System;

[assembly:CLSCompliant(true)]
public class a
{
    public static int _a = 0;  // CS3008
    // OK, private
    // private static int _a1 = 0;

    public static void Main()
    {
    }
}
```

编译器警告（等级 1）CS3009

项目 • 2023/04/07

“type”：基类型“type”不符合 CLS

在标记为符合 CLS 的程序集中，基类型被标记为不必符合公共语言规范 (CLS)。 删除指定程序集符合 CLS 的属性，或者删除表示类型不符合 CLS 的属性。 有关 CLS 符合性的详细信息，请参阅语言独立性和与语言无关的组件。

示例

下面的示例生成 CS3009：

C#

```
// CS3009.cs

using System;

[assembly:CLSCompliant(true)]
[CLSCompliant(false)]
public class B
{
}

public class C : B   // CS3009
{
    public static void Main () {}
}
```

编译器警告（等级 1）CS3010

项目 • 2023/04/07

“member”：符合 CLS 的接口必须仅有符合 CLS 的成员

在具有 `[assembly:CLSCompliant(true)]`标记的程序集中，接口包含具有 `[CLSCompliant(false)]`标记的成员。 删除其中一个共用语言规范 (CLS) 遵从性特性。 有关 CLS 合规性的详细信息，请参阅语言独立性和与语言无关的组件。

示例

下面的示例生成 CS3010：

C#

```
// CS3010.cs

using System;

[assembly:CLSCompliant(true)]
public interface I
{
    [CLSCompliant(false)]
    int M();   // CS3010
}

public class C : I
{
    public int M()
    {
        return 1;
    }

    public static void Main()
    {
    }
}
```

编译器警告（等级 1）CS3011

项目 • 2023/04/07

“member”：只有符合 CLS 的成员才能是抽象的

类成员不能既 抽象 又不遵从公共语言规范 (CLS)。 CLS 指定应实现的所有类成员。 有关 CLS 合规性的详细信息，请参阅语言独立性和与语言无关的组件。

示例

下面的示例生成 CS3011：

C#

```
// CS3011.cs

using System;

[assembly:CLSCompliant(true)]
public abstract class I
{
    [CLSCompliant(false)]
    public abstract int M();   // CS3011

    // OK
    [CLSCompliant(false)]
    public void M2()
    {
    }
}

public class C : I
{
    public override int M()
    {
        return 1;
    }

    public static void Main()
    {
    }
}
```

编译器警告（等级 1）CS3012

项目 • 2023/04/07

不能在模块上指定与程序集的 CLSCompliant 特性不同的 CLSCompliant 特性

为了通过 `[module:System.CLSCompliant(true)]` 使模块符合公共语言规范 (CLS)，此模块必须使用 TargetType 编译器选项生成。 有关 CLS 的详细信息，请参阅语言独立性和与语言无关的组件。

示例

当没有使用 `/target:module` 生成时，以下示例将生成 CS3012：

C#

```
// CS3012.cs
// compile with: /W:1

[module:System.CLSCompliant(true)]   // CS3012
public class C
{
    public static void Main()
    {
    }
}
```

编译器警告（等级 1）CS3013

项目 • 2023/04/07

添加的模块必须用 CLSCompliant 特性标记才能与程序集匹配

已使用 AddModule 将使用 TargetType 编译器选项的 module 元素编译的模块添加到编译中。 但是，该模块对公共语言规范 (CLS) 的遵从性与当前编译的 CLS 状态不一致。

CLS 遵从性由模块特性表示。 例如， `[module:CLSCompliant(true)]` 表示模块符合 CLS，`[module:CLSCompliant(false)]` 表示模块不符合 CLS。 默认值为 `[module:CLSCompliant(false)]`。 有关 CLS 的详细信息，请参阅语言独立性和与语言无关的组件。

编译器警告（等级 1）CS3014

项目 • 2023/04/07

由于程序集没有 CLSCompliant 特性，因此“成员”不需要 CLSCompliant 特性

在没有指定符合公共语言规范 (CLS) 的源代码文件中，该文件中的构造将被标记为符合 CLS。 这是不允许的。 若要解决此警告，请将程序集级符合 CLS 的特性添加到文件中（在以下示例中，取消注释包含程序集级特性的行）。 有关 CLS 合规性的详细信息，请参阅语言独立性和与语言无关的组件。

示例

以下示例生成 CS3014:

C#

```
// CS3014.cs

using System;

// [assembly:CLSCompliant(true)]
public class I
{
    [CLSCompliant(true)]   // CS3014
    public void M()
    {
    }

    public static void Main()
    {
    }
}
```

编译器警告（等级 1）CS3015

项目 • 2023/04/07

“method signature”没有只使用符合 CLS 类型的可访问的构造函数

若要符合公共语言规范 (CLS)，特性类的参数列表不可包括数组。 有关 CLS 符合性的详细信息，请参阅语言独立性和与语言无关的组件。

示例

以下示例生成 C3015。

C#

```
// CS3015.cs
// compile with: /target:library
using System;

[assembly:CLSCompliant(true)]
public class MyAttribute : Attribute
{
   public MyAttribute(int[] ai) {}   // CS3015
   // try the following line instead
   // public MyAttribute(int ai) {}
}
```

编译器警告（等级 1）CS3017

项目 • 2023/04/07

不能在模块上指定与程序集的 CLSCompliant 特性不同的 CLSCompliant 特性

如果程序集 CLSCompliant 特性与模块 CLSCompliant 特性冲突，则会出现此警告。 符合 CLS 的程序集不能包含不符合 CLS 的模块。 若要解决此警告，请确保程序集和模块 CLSCompliant 特性同时为 true 或同时为 false，或者删除其中一个特性。 有关 CLS 合规性的详细信息，请参阅语言独立性和与语言无关的组件。

示例

以下示例生成 CS3017：

C#

```
// CS3017.cs
// compile with: /target:module

using System;

[module: CLSCompliant(true)]
[assembly: CLSCompliant(false)]  // CS3017
// Try this line instead:
// [assembly: CLSCompliant(true)]
class C
{
    static void Main() {}
}
```

编译器警告（等级 1）CS3018

项目 • 2023/04/07

“type”是不符合 CLS 的类型“type”的成员，因此不能将其标记为符合 CLS

如果将 CLSCompliant 特性设置为 `true` 的某一嵌套类声明为通过将 CLSCompliant 特性设置为 `false`来声明的类的成员，将发生此警告。 这是不允许的，因为如果嵌套类是不符合 CLS 的某一外部类的成员，它将无法符合 CLS。 若要解决此警告，请从嵌套类中删除 CLSCompliant 特性，或将其从 `true` 更改为 `false`。 有关 CLS 符合性的详细信息，请参阅语言独立性和与语言无关的组件。

示例

下面的示例生成 CS3018。

C#

```
// CS3018.cs
// compile with: /target:library
using System;

[assembly: CLSCompliant(true)]
[CLSCompliant(false)]
public class Outer
{
   [CLSCompliant(true)]   // CS3018
   public class Nested {}

   // OK
   public class Nested2 {}

   [CLSCompliant(false)]
   public class Nested3 {}
}
```

编译器警告（等级 1）CS3022

项目 • 2023/04/07

CLSCompliant 特性在应用于参数时无意义。 请改为尝试将其用于此方法。

由于 CLS 符合性规则将应用于方法和类型声明中，因此未检查方法参数的 CLS 符合性。

示例

下面的示例生成 CS3022：

C#

```
// CS3022.cs
// compile with: /W:1

using System;

[assembly: CLSCompliant(true)]
[CLSCompliant(true)]
public class C
{
    public void F([CLSCompliant(true)] int i)
    {
    }

    public static void Main()
    {
    }
}
```

编译器警告（等级 1）CS3023

项目 • 2023/04/07

CLSCompliant 特性在应用于返回类型时无意义。 请改为尝试将其用于此方法。

由于 CLS 符合性规则将应用于方法和类型声明中，因此未检查返回类型的 CLS 符合性。

示例

下面的示例生成警告 CS3023：

C#

```
// C3023.cs

[assembly:System.CLSCompliant(true)]
public class Test
{
    [return:System.CLSCompliant(true)]  // CS3023
    // Try this instead:
    // [method:System.CLSCompliant(true)]
    public static int Main()
    {
        return 0;
    }
}
```

编译器警告（级别 1）CS3024

项目 • 2023/04/07

约束类型“'type”不符合 CLS。

编译器发出此警告是因为：将不符合 CLS 的类型用作泛型类型约束会导致用某些语言编写的代码无法使用泛型类。

消除此警告

1. 对类型约束使用符合 CLS 的类型。

示例

下面的示例在几个位置生成 CS3024：

C#

```
// cs3024.cs
// Compile with: /target:library
 [assembly: System.CLSCompliant(true)]

[type: System.CLSCompliant(false)]
public class TestClass // CS3024
{
    public ushort us;
}
[type: System.CLSCompliant(false)]
public interface ITest // CS3024
{}
public interface I<T> where T : TestClass
{}
public class TestClass_2<T> where T : ITest
{}
public class TestClass_3<T> : I<T> where T : TestClass
{}
public class TestClass_4<T> : TestClass_2<T> where T : ITest
{}
public class Test
{
    public static int Main()
    {
        return 0;
    }
}
```

另请参阅

- 类型参数的约束

编译器警告（等级 1）CS3026

项目 • 2023/04/07

符合 CLS 的字段“field”不能是可变字段

可变变量不应符合 CLS。

示例

下面的示例生成 CS3026。

C#

```
// CS3026.cs
[assembly:System.CLSCompliant(true)]
public class Test
{
    public volatile int v0 =0;   // CS3026
    // To resolve remove the CLS-Compliant attribute.
    public static void Main() { }
}
```

编译器警告（等级 1）CS3027

项目 • 2023/04/07

“type_1”类型不符合 CLS，因为基接口“type_2”不符合 CLS

不符合 CLS 的类型不能是符合 CLS 的类型的基类型。

示例 1

下面的示例包含的接口中有一个方法，该方法在其签名中使用不符合 CLS 的类型，致使类型不符合 CLS。

C#

```
// CS3027.cs
// compile with: /target:library
public interface IBase
{
   void IMethod(uint i);
}
```

示例 2

下面的示例生成 CS3027。

C#

```
// CS3027_b.cs
// compile with: /reference:CS3027.dll /target:library /W:1
[assembly:System.CLSCompliant(true)]
public interface IDerived : IBase {}
```

编译器警告（等级 1）CS4014

项目 • 2023/04/07

在调用完成之前，会继续执行当前方法，原因是此调用不处于等待状态。 请考虑向调用结果应用 `await` 运算符。

当前方法调用返回 Task 或 Task<TResult> 的异步方法，而不对结果应用 await 运算符。 对异步方法的调用会启动异步任务。 但是，由于没有应用 `await` 运算符，程序将继续运行，而不必等待任务完成。 在大多数情况下，此行为并不是您所期望的。 方法调用的其他方面通常取决于调用的结果，而调用的方法至少应在从包含调用的方法返回之前完成。

对于被调用的异步方法中引发的异常，如何进行处理同样是一个重要的问题。 在返回 Task 或 Task<TResult> 的方法中引发的异常将存储在返回的任务中。 如果不等待任务或不显式检查异常，该异常会丢失。 如果等待任务，将重新引发异常。

作为最佳做法，应始终等待调用。

只有当您确定不需要等待异步调用完成并且调用的方法不会引发任何异常时，才应考虑禁止显示警告。 在此情况下，可以通过将调用的任务结果分配给变量来禁止显示警告。

下面的示例演示如何生成警告，如何禁止显示警告以及如何等待调用。

C#

```
static async Task CallingMethodAsync(int millisecondsDelay)
{
    Console.WriteLine("  Entering calling method.");

    // Call #1.
    // Call an async method. Because you don't await it, its completion
    // isn't coordinated with the current method, CallingMethodAsync.
    // The following line causes warning CS4014.
    CalledMethodAsync(millisecondsDelay);

    // Call #2.
    // To suppress the warning without awaiting, you can assign the
    // returned task to a variable. The assignment doesn't change how
    // the program runs. However, recommended practice is always to
    // await a call to an async method.

    // Replace Call #1 with the following line.
    // Task delayTask = CalledMethodAsync(millisecondsDelay);

    // Call #3
    // To contrast with an awaited call, replace the unawaited call
    // (Call #1 or Call #2) with the following awaited call. Best
    // practice is to await the call.
```

```
    // await CalledMethodAsync(millisecondsDelay);

    Console.WriteLine("  Returning from calling method.");
}

static async Task CalledMethodAsync(int millisecondsDelay)
{
    Console.WriteLine("    Entering called method, starting and awaiting
Task.Delay.");

    await Task.Delay(millisecondsDelay);

    Console.WriteLine("    Task.Delay is finished--returning from called
method.");
}
```

在此示例中，如果选择调用 #1 或调用 #2，未等待的异步方法 `CalledMethodAsync` 将在调用方 `CallingMethodAsync` 和调用方的调用方完成后完成。 以下输出的最后一行演示了调用的方法何时完成。 输出内容标示了整个示例中何时进入和退出调用 `CallingMethodAsync` 的事件处理程序。

控制台

```
Entering the Click event handler.
  Entering calling method.
    Entering called method, starting and awaiting Task.Delay.
  Returning from calling method.
Exiting the Click event handler.
    Task.Delay is finished--returning from called method.
```

通过使用 #pragma warning 指令，也可以禁止显示编译器警告。

示例

以下控制台应用程序包含上一示例中的方法。 下列步骤用于设置该应用程序。

1. 创建控制台应用程序并将其命名为 `AsyncWarning`。
2. 在 Visual Studio Code 编辑器中，选择“Program.cs”文件。
3. 将 Program.cs 中的代码替换为以下代码。

C#

```
using System;
using System.Threading.Tasks;

namespace AsyncWarning
```

```
{
    class Program
    {
        static async Task Main()
        {
            Console.WriteLine("Entering Main() application entry
point.");

            int millisecondsDelay = 2000;
            await CallingMethodAsync(millisecondsDelay);

            Console.WriteLine("Exiting Main() application entry
point.");

            await Task.Delay(millisecondsDelay + 500);
        }

        static async Task CallingMethodAsync(int millisecondsDelay)
        {
            Console.WriteLine("  Entering calling method.");

            // Call #1.
            // Call an async method. Because you don't await it, its
completion
            // isn't coordinated with the current method,
CallingMethodAsync.
            // The following line causes warning CS4014.
            // CalledMethodAsync(millisecondsDelay);

            // Call #2.
            // To suppress the warning without awaiting, you can assign
the
            // returned task to a variable. The assignment doesn't
change how
            // the program runs. However, recommended practice is
always to
            // await a call to an async method.

            // Replace Call #1 with the following line.
            //Task delayTask = CalledMethodAsync(millisecondsDelay);

            // Call #3
            // To contrast with an awaited call, replace the unawaited
call
            // (Call #1 or Call #2) with the following awaited call.
Best
            // practice is to await the call.

            // await CalledMethodAsync(millisecondsDelay);

            Console.WriteLine("  Returning from calling method.");
        }

        static async Task CalledMethodAsync(int millisecondsDelay)
        {
```

```
            Console.WriteLine("    Entering called method, starting and
awaiting Task.Delay.");

            await Task.Delay(millisecondsDelay);

            Console.WriteLine("    Task.Delay is finished--returning
from called method.");
        }
    }

    // Output with Call #1 or Call #2. (Wait for the last line to
appear.)

    // Entering Main() application entry point.
    //   Entering calling method.
    //     Entering called method, starting and awaiting Task.Delay.
    //   Returning from calling method.
    // Exiting Main() application entry point.
    //     Task.Delay is finished--returning from called method.

    // Output with Call #3, which awaits the call to CalledMethodAsync.

    // Entering Main() application entry point.
    //   Entering calling method.
    //     Entering called method, starting and awaiting Task.Delay.
    //     Task.Delay is finished--returning from called method.
    //   Returning from calling method.
    // Exiting Main() application entry point.
}
```

4. 选择 F5 键以运行该项目。

预期的输出显示在代码末尾。

请参阅

- await
- 使用 async 和 await 的异步编程

编译器警告（等级 1）CS5000

项目 • 2023/04/07

未知编译器选项“/option”

指定了无效的 编辑器选项 。

> **备注**
>
> Roslyn 中不再使用此编译器错误。

编译器警告（等级 2）CS0108

项目 • 2024/03/01

“member1”会隐藏继承的成员“member2”。 如果打算隐藏，请使用 new 关键字。

声明成员所使用的名称与基类中的成员名称相同。 但没有使用新修饰符。

以下示例生成 CS0108。 可通过以下列两种方法之一解决 CS0108 错误：

- 如果不打算成员隐藏，请重命名派生类中的成员。
- 使用 `new` 修饰符声明基成员的派生成员隐藏是故意的。

C#

```
// CS0108.cs
// compile with: /W:2
using System;

namespace MyNamespace;

public class BaseClass
{
    public int i = 1;
}

public class DerivedClass : BaseClass
{
    public static int i = 2;   // CS0108
    // Use the following line instead:
    // public static new int i = 2;

    public static void Main()
    {
        Console.WriteLine(i);
    }
}
```

另请参阅

- new 修饰符

在 GitHub 上与我们协作

.NET .NET 反馈

可以在 GitHub 上找到此内容的源，还可以在其中创建和查看问题和拉取请求。 有关详细信息，请参阅参与者指南。

.NET 是一个开放源代码项目。 选择一个链接以提供反馈：

提出文档问题

提供产品反馈

编译器警告（等级 2）CS0114

项目 • 2023/04/07

“function1”隐藏继承的成员“function2”。 要使当前方法重写该实现，请添加 override 关键字。 否则请添加 new 关键字。

类中的声明与基类中的声明冲突，从而使基类成员隐藏。

有关详细信息，请参阅 base。

下面的示例生成 CS0114：

C#

```
// CS0114.cs
// compile with: /W:2 /warnaserror
abstract public class clx
{
   public abstract void f();
}

public class cly : clx
{
   public void f() // CS0114, hides base class member
   // try the following line instead
   // override public void f()
   {
   }

   public static void Main()
   {
   }
}
```

编译器警告（等级 2）CS0162

项目 • 2023/04/07

检测到无法访问的代码

编译器检测到永远不执行的代码。

示例

下面的示例生成 CS0162：

C#

```
// CS0162.cs
// compile with: /W:2
public class Program
{
    public static void Main()
    {
        goto lab1;
        {
            // The following statements cannot be reached:
            int i = 9;   // CS0162
            i++;
        }
    lab1:
        {
        }
    }
}
```

生成此错误的另一个常见示例如下所示：

C#

```
public static class Class1
{
    public static string Method1()
    {
        string x = "a";
        switch (x)
        {
            case "a":
                return "a";
                break;          // CS0162
        }
        return "";
```

```
    }
}
```

无法到达 `break` 语句，因为它出现在 `return` 语句之后。 `return` 语句结束封闭 `case` 分支。

编译器警告（等级 2）CS0164

项目 • 2024/04/11

这个标签尚未被引用

声明了标签，但从未使用过。

下面的示例生成 CS0164:

C#

```
// CS0164.cs
// compile with: /W:2
public class Program
{
    public static void Main()
    {
        int i = 0;
    l1: // CS0164
        i++;
        // Uncomment the following lines to resolve this warning.
        // if (i < 10)
        //     goto l1;
    }
}
```


在 GitHub 上与我们协作

可以在 GitHub 上找到此内容的源，还可以在其中创建和查看问题和拉取请求。 有关详细信息，请参阅参与者指南。

.NET

.NET 反馈

.NET 是一个开放源代码项目。 选择一个链接以提供反馈：

提出文档问题

提供产品反馈

编译器警告（等级 2）CS0252

项目 • 2023/04/07

可能非有意的引用比较；若要获取值比较，请将左边强制转换为类型“type”

编译器正在执行引用比较。 如果想要比较字符串值，请将表达式右侧强制转换为 `type`。

下面的示例生成 CS0252：

C#

```
// CS0252.cs
// compile with: /W:2
using System;

class MyClass
{
   public static void Main()
   {
      string s = "11";
      object o = s + s;

      bool b = o == s;   // CS0252
      // try the following line instead
      // bool b = (string)o == s;
   }
}
```

编译器警告（等级 2）CS0253

项目 • 2023/04/07

可能是无意的引用比较；若要获取值比较，请将右边转换为类型“type”

编译器正在执行引用比较。 如果你想比较字符串的值，请将表达式右侧转换为 `type`。

下面的示例生成 CS0253：

C#

```
// CS0253.cs
// compile with: /W:2
using System;
class MyClass
{
   public static void Main()
   {
      string s = "11";
      object o = s + s;

      bool c = s == o;   // CS0253
      // try the following line instead
      // bool c = s == (string)o;
   }
}
```

编译器警告（等级 2）CS0278

项目 • 2023/04/07

“type”不实现“pattern name”模式。 “method name”具有“method name”歧义。

C# 中的多条语句依赖于定义的模式，例如 `foreach` 和 `using`。 例如，foreach 语句依赖于实现“enumerable”模式的集合类。

如果编译器由于歧义而无法完成匹配，则会出现 CS0278。 例如，“enumerable”模式需要名为 `MoveNext` 的方法，而你的代码可能包含两个名为 `MoveNext` 的方法。 编译器将尝试查找一个要使用的接口，但建议你确定并解决歧义的起因。

示例

下面的示例生成 CS0278：

C#

```
// CS0278.cs
using System.Collections.Generic;
public class myTest
{
   public static void TestForeach<W>(W w)
      where W: IEnumerable<int>, IEnumerable<string>
   {
      foreach (int i in w) {}   // CS0278
   }
}
```

编译器警告（等级 2）CS0279

项目 • 2023/04/07

“type name”不实现“pattern name”模式。 “method name”是静态的或非公共的。

C# 中的多条语句依赖于定义的模式，例如 `foreach` 和 `using`。 例如， `foreach` 依赖于实现可枚举模式的集合类。 由于被声明为 `static` 或未被声明为 `public`的方法，编译器不能建立匹配时，将发生该错误。 模式中的方法必须是类的实例，并且必须是公共的。

示例

下面的示例生成 CS0279：

C#

```
// CS0279.cs

using System;
using System.Collections;

public class myTest : IEnumerable
{
    IEnumerator IEnumerable.GetEnumerator()
    {
        yield return 0;
    }

    internal IEnumerator GetEnumerator()
    {
        yield return 0;
    }

    public static void Main()
    {
        foreach (int i in new myTest()) {}  // CS0279
    }
}
```

编译器警告（等级 2）CS0280

项目 • 2023/04/07

“type”不实现“pattern name”模式。 “method name”具有错误的签名。

C# 中的两个语句 **foreach** 和 **using**分别依赖于预定义模式“集合”和“资源”。 当编译器由于方法的签名不正确，而无法将这两个语句中的某个语句与其模式匹配时，会出现此警告。 例如，“集合”模式要求存在一个不采用任何参数并返回 MoveNext 的名为 `boolean`的方法。 你的代码可能包含具有参数或可能返回对象的 MoveNext 方法。

“资源”模式和 `using` 是另一个示例。 “资源”模式需要 Dispose 方法；如果定义了同名属性，则你会收到此警告。

要解决此警告，请确保类型中的方法签名与模式中对应方法的签名匹配，并确保你没有与模式所需方法同名的属性。

示例

下面的示例生成 CS0280。

C#

```
// CS0280.cs
using System;
using System.Collections;

public class ValidBase: IEnumerable
{
   IEnumerator IEnumerable.GetEnumerator()
   {
      yield return 0;
   }

   internal IEnumerator GetEnumerator()
   {
      yield return 0;
   }
}

class Derived : ValidBase
{
   // field, not method
   new public int GetEnumerator;
}

public class Test
{
```

```
    public static void Main()
    {
        foreach (int i in new Derived()) {}   // CS0280
    }
}
```

编译器警告（等级 2）CS0435

项目 • 2023/04/07

“assembly”中的命名空间“namespace”与“assembly”中的导入类型“type”冲突。 请使用“assembly”中定义的命名空间。

源文件 (file_2) 中的命名空间与 file_1 中的导入类型冲突时，会发出此警告。 编译器使用源文件中的命名空间。

下面的示例生成 CS0435：

首先编译以下文件:

C#

```
// CS0435_1.cs
// compile with: /t:library
public class Util
{
   public class A { }
}
```

然后编译以下文件:

C#

```
// CS0435_2.cs
// compile with: /r:CS0435_1.dll

using System;

namespace Util
{
   public class A { }
}

public class Test
{
   public static void Main()
   {
      Console.WriteLine(typeof(Util.A)); // CS0435
   }
}
```

编译器警告（等级 2）CS0436

项目 • 2023/06/21

“assembly”中的类型“type”与“assembly”中导入的类型“type2”冲突。 请使用“assembly”中定义的类型。

源文件 (file_2) 中的类型与 file_1 中某一导入的类型冲突时会发出此警告。 编译器使用源文件中的命名空间。

示例 1

C#

```
// CS0436_a.cs
// compile with: /target:library
public class A {
   public void Test() {
      System.Console.WriteLine("CS0436_a");
   }
}
```

示例 2

以下示例生成 CS0436。

C#

```
// CS0436_b.cs
// compile with: /reference:CS0436_a.dll
// CS0436 expected
public class A {
   public void Test() {
      System.Console.WriteLine("CS0436_b");
   }
}

public class Test
{
   public static void Main()
   {
      A x = new A();
      x.Test();
   }
}
```

编译产生以下输出：

控制台

```
CS0436_b
```

编译器警告（等级 2）CS0437

项目 • 2023/04/07

“assembly2”中的类型“type”与“fassembly1”中的导入命名空间“namespace”冲突。 请使用“assembly”中定义的类型。

源文件 file_2 中的类型与 file_1 中的导入命名空间冲突时，会发出此警告。 编译器使用源文件中的类型。

示例 1

C#

```
// CS0437_a.cs
// compile with: /target:library
namespace Util
{
   public class A {
      public void Test() {
         System.Console.WriteLine("CS0437_a.cs");
      }
   }
}
```

示例 2

下面的示例生成 CS0437。

C#

```
// CS0437_b.cs
// compile with: /reference:CS0437_a.dll /W:2
// CS0437 expected
class Util
{
   public class A {
      public void Test() {
         System.Console.WriteLine("CS0437_b.cs");
      }
   }
}

public class Test
{
   public static void Main()
   {
```

```
        Util.A x = new Util.A();
        x.Test();
    }
}
```

编译器警告（等级 2）CS0444

项目 • 2023/05/10

在“System namespace 1”中找不到预定义的类型“type name 1”，但是可在“System namespace 2”中找到。

未在编译器应找到预定义对象（例如 Int32 ）的位置找到它，但在“System namespace 2”中找到了它。

错误可能表示 .NET 安装不正确。 若要解决此问题，请重新安装 .NET。

如果你正在编写自己的基类库，也可能会遇到此错误。 在这种情况下，若要解决此错误，重新生成 mscorlib。

> **备注**
>
> Roslyn 中不再使用此编译器警告。

编译器警告（等级 2）CS0458

项目 • 2023/04/07

表达式的结果总是“type name”类型的 "null"

此警告由始终导致 `null` 的可以为 null 的值类型表达式所致。

下面的代码生成警告 CS0458。

示例

此示例阐释了一些因可以为 null 的值类型导致此错误的不同运算。

C#

```
// CS0458.cs
using System;
public  class Test
{
    public static void Main()
    {
        int a = 5;
        int? b = a + null;    // CS0458
        int? qa = 15;
        b = qa + null;        // CS0458
        b -= null;            // CS0458
        int? qa2 = null;
        b = qa2 + null;       // CS0458
        qa2 -= null;          // CS0458
    }
}
```

编译器警告（等级 2）CS0464

项目 • 2023/04/07

与“type”类型的 null 进行比较始终产生“false”

当你比较可以为 null 的值类型变量与 null 时，如果比较不为 `==` 或 `!=`，将出现此警告。 若要解决此错误，请验证你是否确实要检查值是否为 `null`。 类似 `i == null` 这样的比较结果既可以为 true，也可以为 false。 类似 `i > null` 这样的比较结果始终为 false。

示例

下面的示例生成 CS0464。

C#

```
// CS0464.cs
class MyClass
{
   public static void Main()
   {
      int? i = 0;
      if (i < null) ;   // CS0464

      i++;
   }
}
```

编译器警告（等级 2）CS0467

项目 • 2023/04/07

方法“method”和非方法“non-method”之间存在多义性。 使用方法组。

如果继承的成员具有相同的签名，但来自不同的接口，则会导致多义性错误。

示例

下面的示例生成 CS0467。

C#

```
// CS0467.cs
interface IList
{
    int Count { get; set; }
}

interface ICounter
{
    void Count(int i);
}

interface IListCounter : IList, ICounter {}

class Driver
{
    void Test(IListCounter x)
    {
        // The following line causes the warning. The assignment also
        // causes an error because you can't assign a value to a method.
        x.Count = 1;
        x.Count(3);
        // To resolve the warning, you can change the name of the method or
        // the property.

        // You can also disambiguate by specifying IList or ICounter.
        ((IList)x).Count = 1;
        ((ICounter)x).Count(3);
    }

    static void Main()
    {
    }
}
```

编译器警告（等级 2）CS0469

项目 • 2023/04/07

“goto case”值不可隐式转换为类型“type”

使用 `goto case`时，必须将 goto case 的值隐式转换为 switch 类型。

示例

下面的示例生成 CS0469。

C#

```
// CS0469.cs
// compile with: /W:2
class Test
{
   static void Main()
   {
      char c = (char)180;
      switch (c)
      {
         case (char)127:
            break;

         case (char)180:
            goto case 127;   // CS0469
            // try the following line instead
            // goto case (char) 127;
      }
   }
}
```

编译器警告（等级 2）CS0472

项目 • 2023/04/07

由于类型“value2”的值永不等于类型“value3”的“null”，因此该表达式的结果始终为“value1”

如果你使用具有始终为 null 值的运算符，则编译器会发出警告。

示例

下面的示例生成 CS0472。

C#

```
public class Test
{
    public static int Main()
    {
        int i = 5;
        int counter = 0;

        // Comparison:
        if (i == null)  // CS0472
        // To resolve, use a valid value for i.
            counter++;
        return counter;
    }
}
```

编译器警告（等级 2）CS0618

项目 • 2023/06/05

“member”已过时：“text”

类成员标记有 `Obsolete` 属性，这样，当引用该类成员时，将发出警告。 有关详细信息，请参阅程序集级别属性。

下面的示例生成 CS0618：

C#

```
// CS0618.cs
// compile with: /W:2
using System;

public class C
{
   [Obsolete("Use newMethod instead", false)]   // warn if referenced
   public static void m2()
   {
   }

   public static void newMethod()
   {
   }
}

class MyClass
{
   public static void Main()
   {
      C.m2();  // CS0618
   }
}
```

编译器警告（等级 2）CS0652

项目 • 2023/04/07

与整数常量比较无意义；该常量不在“type”类型的范围之内

编译器检测到常量和变量之间的比较，其中常量不在变量范围内。

以下示例生成 CS0652：

C#

```
// CS0652.cs
// compile with: /W:2
public class Class1
{
   private static byte i = 0;
   public static void Main()
   {
      short j = 256;
      if (i == 256)   // CS0652, 256 is out of range for byte
         i = 0;
   }
}
```

编译器警告（等级 2）CS0728

项目 • 2023/04/07

对局部“variable”的赋值可能不正确，此变量是 using 或 lock 语句的参数。 Dispose 调用或解锁将发生在局部变量的原始值上。

有几种 `using` 或 `lock` 会导致资源临时泄漏的情况。 其中一个示例如下：

`thisType f = null;`

`using (f)`

`{`

`f = new thisType();`

`...`

`}`

在此情况下，将在 `thisType` 块执行完成后释放变量 `using` 的原始值（如 null），但不会释放块中创建的 `thisType` 对象（虽然其最终会进行垃圾回收）。

若要解决此错误，请使用以下形式：

`using (thisType f = new thisType())`

`{`

`...`

`}`

在此情况下，将释放新分配的 `thisType` 对象。

示例

下面的代码将生成错误 CS0728。

C#

```
// CS0728.cs

using System;
public class ValidBase : IDisposable
```

```
{
    public void Dispose() {  }
}

public class Logger
{
    public static void dummy()
    {
        ValidBase vb = null;
        using (vb)
        {
            vb = null;  // CS0728
        }
        vb = null;
    }
    public static void Main() { }
}
```

编译器警告（等级 2）CS1571

项目 • 2023/04/07

“构造”上的 XML 注释中有重复的“参数”param 标记

当使用 DocumentationFile 编译器选项时，找到了同一方法参数的多个注释。 删除其中一个重复行。

下面的示例生成 CS1571：

C#

```
// CS1571.cs
// compile with: /W:2 /doc:x.xml

/// <summary>help text</summary>
public class MyClass
{
   /// <param name='Int1'>Used to indicate status.</param>
   /// <param name='Char1'>An initial.</param>
   /// <param name='Int1'>Used to indicate status.</param> // CS1571
   public static void MyMethod(int Int1, char Char1)
   {
   }

   /// <summary>help text</summary>
   public static void Main ()
   {
   }
}
```

编译器警告（等级 2）CS1572

项目 • 2023/04/07

“构造”上的 XML 注释中有“参数”的 param 标记，但是不存在具有该名称的参数

使用 DocumentationFile 编译器选项时，为方法不存在的参数指定了注释。 更改传递给名称特性的值或者删除其中一个注释行。

下面的示例生成 CS1572：

C#

```
// CS1572.cs
// compile with: /W:2 /doc:x.xml

/// <summary>help text</summary>
public class MyClass
{
   /// <param name='Int1'>Used to indicate status.</param>
   /// <param name='Char1'>Used to indicate status.</param>
   /// <param name='Char2'>???</param> // CS1572
   public static void MyMethod(int Int1, char Char1)
   {
   }

   /// <summary>help text</summary>
   public static void Main ()
   {
   }
}
```

编译器警告（等级 2）CS1587

项目 • 2023/04/07

XML 注释没有放在有效语言元素上

建议用于文档注释的标记不能用于所有语言元素。 例如，命名空间上不允许标记。 有关 XML 注释的详细信息，请参阅 Recommended Tags for Documentation Comments。

示例

下面的示例生成 CS1587：

C#

```
// CS1587.cs
// compile with: /W:2 /doc:x.xml

/// <summary>test</summary>   // CS1587, tag not allowed on namespace
namespace MySpace
{
    class MyClass
    {
        public static void Main()
        {
        }
    }
}
```

编译器警告（等级 2）CS1668

项目 • 2023/04/07

在“path string”中指定的路径“path”无效 - “系统错误消息”

提供给命令行的 AdditionalLibPaths 的路径无效，或者 LIB 环境变量中的路径无效。 检查所用的路径以验证它是否存在且可供访问。 单引号中的错误消息是从操作系统返回的错误。

另请参阅

- C# 编译器选项
- C# 编译器错误

编译器警告（等级 2）CS1698

项目 • 2023/04/07

循环的程序集引用“AssemblyName1”与输出程序集名称“AssemblyName2”不匹配。 请尝试添加对“AssemblyName1”的引用，或者更改输出程序集的名称以使其匹配。

程序集引用不正确时，会发生 CS1698。 如果重新编译引用的程序集，也可能发生此错误。 要解决此问题，请不要替换本身是所引用程序集的依赖项的程序集。

示例 1

C#

```
// CS1698_a.cs
// compile with: /target:library /keyfile:mykey.snk
[assembly:System.Reflection.AssemblyVersion("2")]
public class CS1698_a {}
```

示例 2

C#

```
// CS1698_b.cs
// compile with: /target:library /reference:CS1698_a.dll /keyfile:mykey.snk
public class CS1698_b : CS1698_a {}
```

示例 3

下面的示例生成 CS1698。

C#

```
// CS1698_c.cs
// compile with: /target:library /out:cs1698_a.dll /reference:cs1698_b.dll
/keyfile:mykey.snk
// CS1698 expected
[assembly:System.Reflection.AssemblyVersion("3")]
public class CS1698_c : CS1698_b {}
public class CS1698_a {}
```

编译器警告（等级 2）CS1701

项目 • 2023/04/07

如果程序集引用“Assembly Name #1”与“Assembly Name #2”一致，则可能需要提供运行时策略

这两个程序集的发行号和/或版本号不同。 要实现统一，必须在应用程序的 .config 文件中指定指令，并且必须提供程序集正确的强名称，如下面的示例代码所示。

示例 1

下面的多文件示例使用两个不同的外部别名引用一个程序集。 第一个示例生成较旧版本的代码，该代码创建程序集 CS1701_d。

C#

```
// CS1701_a.cs
// compile with: /target:library /out:cs1701_d.dll /keyfile:mykey.snk
using System.Reflection;
[assembly:AssemblyVersion("1.0")]
public class A {
   public void M1() {}
}

public class C1 {}
```

示例 2

这是用于创建较新版本的程序集 CS1701_d 的代码。 注意，由于这两个版本的输出文件同名，因此它需要将较新版本编译到不同于较早版本的目录中。

C#

```
// CS1701_b.cs
// compile with: /target:library /out:c:\\cs1701_d.dll /keyfile:mykey.snk
using System.Reflection;
[assembly:AssemblyVersion("2.0")]
public class A {
   public void M2() {}
   public void M1() {}
}

public class C2 {}
public class C1 {}
```

示例 3

此示例设置外部别名 A1 和 A2。

C#

```
// CS1701_c.cs
// compile with: /target:library /reference:A2=c:\\cs1701_d.dll
/reference:A1=cs1701_d.dll

extern alias A1;
extern alias A2;
// using System;
using a1 = A1::A;
using a2 = A2::A;

public class Ref {
   public static a1 A1() { return new a1(); }
   public static a2 A2() { return new a2(); }

   public static A1::C1 M1() { return new A1::C1(); }
   public static A2::C2 M2() { return new A2::C2(); }
}
```

示例 4

此示例调用的方法使用 A 的两个不同的别名。下面的示例生成 C1701。

C#

```
// CS1701_d.cs
// compile with: /reference:c:\\CS1701_d.dll /reference:CS1701_c.dll
// CS1701 expected
class Tester {
   public static void Main() {
      Ref.A1().M1();
      Ref.A2().M2();
   }
}
```

编译器警告（等级 2）CS1710

项目 • 2023/04/07

“类型”上的 XML 注释中有重复的“参数”typeparam 标记

泛型类型的文档包含重复的类型形参标记。

示例

下面的代码将导致出现警告 CS1710。

C#

```
// CS1710.cs
// compile with: /doc:cs1710.xml
// To resolve this warning, delete one of the duplicate <typeparam>'s.
using System;
class Stack<ItemType>
{
}

/// <typeparam name="MyType">can be an int</typeparam>
/// <typeparam name="MyType">can be an int</typeparam>
class MyStackWrapper<MyType>
{
    // Open constructed type Stack<MyType>.
    Stack<MyType> stack;
    public MyStackWrapper(Stack<MyType> s)
    {
        stack = s;
    }
}

class CMain
{
    public static void Main()
    {
        // Closed constructed type Stack<int>.
        Stack<int> stackInt = new Stack<int>();
        MyStackWrapper<int> MyStackWrapperInt =
            new MyStackWrapper<int>(stackInt);
    }
}
```

编译器警告（等级 2）CS1711

项目 • 2023/04/07

“类型”上的 XML 注释中有“参数”的 typeparam 标记，但是不存在具有该名称的类型形参

泛型类型的文档包含名称错误的类型形参的标记。

示例

下面的代码生成 CS1711。

C#

```
// cs1711.cs
// compile with: /doc:cs1711.xml
// CS1711 expected
using System;
///<typeparam name="WrongName">can be an int</typeparam>
class CMain
{
    public static void Main() { }
}
```

编译器警告（等级 2）CS1927

项目 • 2023/04/07

对模块忽略 /win32manifest，因为它仅应用于程序集。

win32 清单仅应用于程序集级别。 模块将进行编译，但不具有清单。

更正此错误

1. 删除 **/win32manifest option**。
2. 将代码编译为程序集。

示例

下面的示例在同时用 **/target:module** 和 **/win32manifest** 编译器选项进行编译时生成 CS1927。

C#

```
// cs1927.cs
// Compile with: /target:module /win32manifest
using System;

class ManifestWithModule
{
    static int Main()
    {
        return 1;
    }
}
```

另请参阅

- Win32Manifest（C# 编译器选项）
- **TargetType** 模块（C# 编译器选项）

编译器警告（等级 2）CS3019

项目 • 2023/04/07

“type”在此程序集外部不可见，因此不会对它执行 CLS 遵从性检查。

当具有 CLSCompliantAttribute 特性的类型或成员在另一个程序集中不可见时，则会出现此警告。 若要解决此错误，删除在另外一个程序集中不可见的任何类或成员的特性，或使该类型或成员变为可见。 有关 CLS 合规性的详细信息，请参阅语言独立性和与语言无关的组件。

示例

以下示例生成 CS3019：

C#

```
// CS3019.cs
// compile with: /W:2

using System;

[assembly: CLSCompliant(true)]

// To fix the error, remove the next line
[CLSCompliant(true)]  // CS3019
class C
{
    [CLSCompliant(false)]  // CS3019
    void Foo()
    {
    }

    static void Main()
    {
    }
}
```

另请参阅

- 语言独立性和与语言无关的组件

编译器警告（等级 2）CS3021

项目 • 2023/04/07

由于程序集没有 CLSCompliant 特性，因此“type”不需要 CLSCompliant 特性

如果 `[CLSCompliant(false)]` 出现在类上，而该类所在的程序集不具有设置为 true 的程序集级 CLSCompliant 特性（即 `[assembly: CLSCompliant(true)]` 行），则会出现此警告。 由于程序集为将其自身声明为符合 CLS，所以该程序集内的任何项都不必将其自身声明为不符合，因为已将其假定为不符合。 有关 CLS 符合性的详细信息，请参阅语言独立性和与语言无关的组件。

若要消除此警告，请删除该特性或添加程序集级特性。

示例

以下示例生成 CS3021：

C#

```
// CS3021.cs
using System;
// Uncomment the following line to declare the assembly CLS Compliant,
// and avoid the warning without removing the attribute on the class.
//[assembly: CLSCompliant(true)]

// Remove the next line to avoid the warning.
[CLSCompliant(false)]               // CS3021
public class C
{
    public static void Main()
    {
    }
}
```

另请参阅

- 语言独立性和与语言无关的组件

编译器警告（等级 3）CS0067

项目 • 2023/04/07

永远不会使用该事件“event”

已声明 event ，但其永远不会用在已声明的类中使用。

下面的示例生成 CS0067：

C#

```
// CS0067.cs
// compile with: /W:3
using System;
delegate void MyDelegate();

class MyClass
{
   public event MyDelegate evt;   // CS0067
   // uncomment TestMethod to resolve this CS0067
/*
   private void TestMethod()
   {
      if (evt != null)
         evt();
   }
*/
   public static void Main()
   {
   }
}
```

如果有意不使用事件，例如，当它是接口实现的一部分时，则可以避免发出不必要的字段，如下所示：

C#

```
using System;

public interface IThing
{
   event Action? E;
}

public class Thing : IThing
{
   // no CS0067 though the event is left unused
   public event Action? E { add { } remove { } }
}
```

编译器警告（等级 3）CS0168

项目 • 2023/04/07

声明了变量“var”，但从未使用过

当你声明一个变量却不使用该变量时，编译器会发出一个 3 级警告。

下面的示例生成一个 CS0168 警告：

C#

```
// CS0168.cs
// compile with: /W:3
public class clx
{
    public int i;
}

public class clz
{
    public static void Main() {
        clx a; // CS0168, the variable 'a' is declared but never used
        // try the following line instead
        // clx a = new clx();  // this does not generate a warning because
the clx constructor must execute.
    }
}
```

编译器警告（等级 3）CS0169

项目 • 2023/04/07

从未使用过“类成员”私有字段

私有变量已声明但从未引用过。 声明一个类的私有成员而不使用它通常会生成此警告。

下面的示例生成 CS0169:

C#

```
// compile with: /W:3
using System;
public class ClassX
{
   int i;   // CS0169, i is not used anywhere
   // Remove the above variable declaration or uncomment TestMethod to clear
warning CS0169
   /*
   public void TestMethod()
   {
       i = 5;
       System.Console.WriteLine(i);
   }
   */

   public static void Main()
   {
   }
}
```

编译器警告（等级 3） CS0219

项目 • 2023/04/07

变量 “variable” 已赋值，但其值从未使用过

当你声明并给一个变量赋值却不使用该变量时，编译器会发出一个 3 级警告。

> **备注**
>
> 只有当变量值是编译时常量时，编译器才会生成此警告。 通过将非常量表达式或方法结果分配给局部变量，可更轻松地在调试程序中观察这些表达式。 它还使结果可访问，防止在变量可访问时进行垃圾回收。

下面的示例演示了生成警告时和不生成警告时的情况：

C#

```
// CS0219.cs
// compile with: /W:3
public class MyClass
{
   public static void Main()
   {
      var interpolated = "Interpolated";
      var a = 0; // CS0219
      int b = GetZero(); // Doesn't generate a warning.
      var c = "Regular string"; // CS0219
      var d = $"Constant interpolated string"; // Doesn't generate a
warning.
      var e = $"{interpolated} string"; // Doesn't generate a warning.
   }

   private static int GetZero()
   {
       return 0;
   }
}
```

编译器警告（等级 3）CS0414

项目 • 2023/04/07

私有字段已被赋值，但从未使用过其值

在编译器可以验证是否从未引用过变量的几种情况下，会出现此警告：

- 私有字段被赋予一个常数值，但随后从未读取。 不必要的赋值会影响性能。 可考虑删除字段。
- 私有或内部静态字段仅在初始值设定项中被赋予常量值。 请考虑将字段更改为常量。
- 私有或内部字段被赋予常量值，并且仅用于 #ifdef 指令排除的块。 请考虑在 #ifdef 块中放置字段。
- 在多个位置为私有或内部字段赋予常量值，但从未访问过。 如果不需要该字段，请考虑将其删除。 否则，以某种适当的方式使用。

在其他情况下，或建议的解决方法不可接受，请使用 #pragma 0414。

下面的示例演示可能生成 CS0414 的一种方法：

C#

```
// CS0414
// compile with: /W3
class C
{
   private int i = 1;  // CS0414

   public static void Main()
   { }
}
```

备注

如果变量 `i` 被声明为 `protected` 或 `public`，则不会生成任何错误，因为编译器无法知道派生的类是不是会使用它，或者某些其他客户端代码是否会实例化类并引用该变量

另请参阅

- C# 编译器错误
- C# 编译器选项

编译器警告（等级 3）CS0419

项目 • 2023/04/07

cref 特性中有不明确的引用：“Method Name1”。 假设“Method Name2”，但也可能已匹配包括“Method Name3”在内的其他重载。

在代码中的 XML 文档注释中，无法解析某一引用。 如果该方法重载，或者如果找到两个具有相同名称的不同标识符，则可能发生此问题。 若要解决此警告，请使用限定的名称来消除该引用的歧义，或在括号中插入特定的重载。

下面的示例生成 CS0419。

C#

```
// cs0419.cs
// compile with: /doc:x.xml /W:3
interface I
{
   /// text for F(void)
   void F();
   /// text for F(int)
   void F(int i);
}
/// text for class MyClass
public class MyClass
{
   /// <see cref="I.F"/>
   public static void MyMethod(int i)
   {
   }
/* Try this instead:
   /// <see cref="I.F(int)"/>
   public static void MyMethod(int i)
   {
   }
*/
   /// text for Main
   public static void Main ()
   {
   }
}
```

编译器警告（等级 3）CS0642

项目 • 2023/04/07

空语句可能有错误

conditional 语句后面的分号可能导致代码不按预期执行。

可使用 NoWarn 编译器选项或 `#pragmas warning` 来禁用此警告；有关详细信息，请参阅 NoWarn（C# 编译器选项）或 #pragma 警告。

下面的示例生成 CS0642：

C#

```
// CS0642.cs
// compile with: /W:3
class MyClass
{
   public static void Main()
   {
      int i;

      for (i = 0; i < 10; i += 1);   // CS0642 semicolon intentional?
      {
         System.Console.WriteLine (i);
      }
   }
}
```

编译器警告（等级 3）CS0659

项目 • 2023/04/07

“class”重写 Object.Equals(object o) 但不重写 Object.GetHashCode()

编译器检测到 Object.Equals 方法的替代，但未检测到 Object.GetHashCode 方法的替代。 替代 Equals 表示你还需要替代 GetHashCode。

下面的代码生成 CS0659：

C#

```
// CS0659.cs
// compile with: /W:3 /target:library
class Test
{
   public override bool Equals(object o) { return true; }   // CS0659
}

// OK
class Test2
{
   public override bool Equals(object o) { return true; }
   public override int GetHashCode() { return 0; }
}
```

请参阅

- Object.Equals
- Object.GetHashCode
- 相等运算符
- 实现 Equals 方法

编译器警告（等级 3）CS0660

项目 • 2023/04/07

“class”定义运算符 == 或运算符 !=，但不重写 Object.Equals(object o)

编译器检测到用户定义的相等运算符或不相等运算符，但是不重写 Object.Equals 方法。 用户定义的相等或不相等运算符意味着你也想要重写 Equals 方法。 有关详细信息，请参阅如何为类型定义值相等性。

下面的示例生成 CS0660：

C#

```
// CS0660.cs
// compile with: /W:3 /warnaserror
class Test   // CS0660
{
   public static bool operator == (object o, Test t)
   {
      return true;
   }

   // uncomment the Equals function to resolve
   // public override bool Equals(object o)
   // {
   //    return true;
   // }

   public override int GetHashCode()
   {
      return 0;
   }

   public static void Main()
   {
   }
}
```

编译器警告（等级 3）CS0661

项目 • 2023/04/07

“class”定义运算符 == 或运算符 !=，但不重写 Object.GetHashCode()

编译器检测到用户定义的相等运算符或不相等运算符，但没有对 **GetHashCode** 函数的重写。 用户定义的相等或不相等运算符意味着你也想要重写 **GetHashCode** 函数。

以下示例生成 CS0661：

C#

```
// CS0661.cs
// compile with: /W:3
class Test   // CS0661
{
   public static bool operator == (object o, Test t)
   {
      return true;
   }

   public static bool operator != (object o, Test t)
   {
      return true;
   }

   public override bool Equals(object o)
   {
      return true;
   }

   // uncomment the GetHashCode function to resolve
   // public override int GetHashCode()
   // {
   //    return 0;
   // }

   public static void Main()
   {
   }
}
```

编译器警告（等级 3）CS0665

项目 • 2023/04/07

条件表达式中的赋值总是常量；是否希望使用 "==" 而非 "="?

条件表达式使用 = 运算符 而不是 == 运算符。

下面的示例生成CS0665：

C#

```
// CS0665.cs
// compile with: /W:3
class Test
{
   public static void Main()
   {
      bool i = false;

      if (i = true)   // CS0665
      // try the following line instead
      // if (i == true)
      {
      }

      System.Console.WriteLine(i);
   }
}
```

编译器警告（等级 3）CS0675

项目 • 2023/04/07

在经符号扩展的操作数上使用了按位“或”运算符；请考虑首先强制转换为较小的无符号类型

编译器对某个变量进行了隐式拓展和带符号扩展，然后在按位或运算中使用生成的值。这可能会导致意外行为。

下面的示例生成 CS0675：

C#

```
// CS0675.cs
// compile with: /W:3
using System;

public class sign
{
   public static void Main()
   {
      int hi = 1;
      int lo = -1;
      long value = (((long)hi) << 32) | lo;              // CS0675, value
contains -1 (0xffffffff_ffffffff)
      // try the following line instead
      // long value = (((long)hi) << 32) | ((uint)lo);   // correct, value
contains 8589934591 (0x00000001_ffffffff)
   }
}
```

编译器警告（等级 3）CS0693

项目 • 2023/04/07

类型参数“type parameter”与外部类型“type”中的类型参数同名

当你具有一个泛型成员（例如，泛型类中的方法）时，将发生此错误。 由于该方法的类型参数不一定与类的类型参数相同，你无法为它们赋予相同的名称。 有关详细信息，请参阅泛型方法。

若要避免这种情况，对其中一个类型参数采用不同的名称。

示例

以下示例生成 CS0693。

C#

```
// CS0693.cs
// compile with: /W:3 /target:library
class Outer<T>
{
   class Inner<T> {}   // CS0693
   // try the following line instead
   // class Inner<U> {}
}
```

编译器警告（等级 3）CS1700

项目 • 2023/04/07

程序集引用“Assembly Name”无效且无法解析

此警告表示未正确指定 InternalsVisibleToAttribute 等属性。

有关详细信息，请参阅友元程序集。

示例

下面的示例生成 CS1700。

C#

```
// CS1700.cs
// compile with: /target:library
using System.Runtime.CompilerServices;
[assembly:InternalsVisibleTo("app2, Retargetable=f")]   // CS1700
[assembly:InternalsVisibleTo("app2")]   // OK
```

编译器警告（等级 3）CS1702

项目 • 2024/06/10

如果程序集引用“Assembly Name #1”与“Assembly Name #2”一致，则可能需要提供运行时策略

两个程序集引用具有不同的生成和/或修订号，因此不会自动统一。 你可能需要使用应用程序 .config 文件中的 `using` 指令提供运行时策略，以强制执行统一。

在 GitHub 上与我们协作

可以在 GitHub 上找到此内容的源，还可以在其中创建和查看问题和拉取请求。 有关详细信息，请参阅参与者指南。

.NET 反馈

.NET 是一个开放源代码项目。 选择一个链接以提供反馈：

提出文档问题

提供产品反馈

编译器警告（等级 3）CS1717

项目 • 2023/04/07

对同一变量进行赋值；是否希望对其他变量赋值?

当你将一个变量分配到其自身（例如 `a = a`）时，会出现此警告。

几个常见的错误将会生成此警告：

- 将 `a = a` 编写为 **if** 语句的条件，如 `if (a = a)`）时，会出现此警告。 你可能意思是说 `if (a == a)`，它始终为 true，所以你可以将其编写地更加简明，如 `if (true)`。
- 键入错误。 你可能意思是说 `a = b`。
- 在其中参数与该字段具有相同名称的构造函数中，不要使用 **this** 关键字：你可能意思是说 `this.a = a`）时，会出现此警告。

示例

以下示例生成 CS1717。

C#

```
// CS1717.cs
// compile with: /W:3
public class Test
{
   public static void Main()
   {
      int x = 0;
      x = x;   // CS1717
   }
}
```

编译器警告（等级 3）CS1718

项目 • 2023/04/07

对同一变量进行比较；是否希望比较其他变量?

如果要与其他变量进行比较，则更正语句。

但另一种可能是你要测试 True 或 False，并通过如 `if (a == a) (true)` 或 `if (a < a) (false)` 这样的语句来完成此操作。 由于以下两个原因，最好使用 `if (true)` 或 `if (false)`：

- 它更简单：总是更清楚地表达你的意思。
- 它有助于避免混淆：C# 2.0 的一个新功能是可以为 Null 的值类型，这类似于 T-SQL（SQL Server 使用的编程语言）中的值 `null`。 由于 T-SQL 中使用了三元逻辑，熟悉 T-SQL 的开发人员可能会关心可以为 Null 值的值类型在类似于 `if (a == a)` 的表达式上的效果。 如果你使用 `true` 或 `false`，就可以避免这类可能出现的混。

示例

下面的代码生成警告 CS1718。

C#

```
// CS1718.cs
using System;
public class Tester
{
    public static void Main()
    {
        int i = 0;
        if (i == i) { // CS1718.cs
        //if (true) {
            i++;
        }
    }
}
```

编译器警告（等级 4）CS0028

项目 • 2023/04/07

“function declaration”的签名错误，不能作为入口点

`Main` 的方法声明无效：声明它时使用的签名无效。 `Main` 必须声明为静态，且它必须返回 int 或 void。 有关详细信息，请参阅 Main() 和命令行参数。

下面的示例生成 CS0028：

C#

```
// CS0028.cs
// compile with: /W:4 /warnaserror
public class a
{
    public static double Main(int i)   // CS0028
    // Try the following line instead:
    // public static void Main()
    {
    }
}
```

编译器警告（等级 4）CS0078

项目 • 2023/04/07

“l”后缀容易与数字“1”混淆；为清楚起见，请使用“L”

编译器会在检测到使用小写 l 而不是大写 L 的 long 文本时发出警告。

以下示例生成 CS0078：

C#

```
class Program
{
   public static void TestL(long i)
   {
   }

   public static void Main()
   {
      TestL(25l);   // CS0078
      // try the following line instead
      // TestL(25L);
   }
}
```

编译器警告（等级 4）CS0109

项目 • 2023/04/07

成员“member”不会隐藏继承的成员。 不需要 new 关键字

类声明包括 new 关键字，即使该声明不重写基类中的现有声明。 你可以删除 **new** 关键字。

下面的示例生成 CS0109：

C#

```
// CS0109.cs
// compile with: /W:4
namespace x
{
   public class a
   {
      public int i;
   }

   public class b : a
   {
      public new int i;
      public new int j;   // CS0109
      public static void Main()
      {
      }
   }
}
```

编译器警告（等级 4）CS0402

项目 • 2023/04/07

“identifier”：入口点不能是泛型或属于泛型类型

在泛型类型中找到了入口点。 若要删除此警告，实现非泛型的类或结构中的 Main。

C#

```
// CS0402.cs
// compile with: /W:4
class C<T>
{
   public static void Main()  // CS0402
   {

   }
}

class CMain
{
   public static void Main() {}
}
```

编译器警告（等级 4）CS0422

项目 • 2023/04/07

不再支持 /incremental 选项

C# 2.0 中不支持增量编译（/incr 或 /incremental）。

> **备注**
>
> Roslyn 中不再使用此编译器警告。

编译器警告（等级 4）CS0429

项目 • 2023/04/07

检测到无法访问的表达式代码

每当无法访问代码中某个表达式的一部分时，便会发生此错误。 在下面的示例中，条件 `false && myTest()` 满足此条件，由于 `&&` 运算的左侧始终为 false 这一事实，将永不计算 `myTest()` 方法。 只要 `&&` 运算符将 `false` 语句的结果计算为 false，便会停止计算，并且将永不计算右侧。

示例

下面的代码生成 CS0429。

C#

```
// CS0429.cs
public class cs0429
{
    public static void Main()
    {
        if (false && myTest())  // CS0429
        // Try the following line instead:
        // if (true && myTest())
        {
        }
        else
        {
            int i = 0;
            i++;
        }
    }

    static bool myTest() { return true; }
}
```

编译器警告（等级 4）CS0628

项目 • 2023/04/07

“member”：在密封类中声明了新的保护成员

密封 类不能引入 保护 成员，因为其他类不能继承 `sealed` 类并使用 `protected` 成员。

下面的示例生成 CS0628：

C#

```
// CS0628.cs
// compile with: /W:4
sealed class C
{
   protected int i;   // CS0628
}

class MyClass
{
   public static void Main()
   {
   }
}
```

编译器警告（等级 4）CS0649

项目 • 2023/04/07

从未对字段“field”赋值，字段将一直保持其默认值“value”

编译器检测到未初始化且从未被赋值的私有字段声明或内部字段声明。

下面的示例生成 CS0649：

C#

```
// CS0649.cs
// compile with: /W:4
using System.Collections;

class MyClass
{
   Hashtable table;  // CS0649
   // You may have intended to initialize the variable to null
   // Hashtable table = null;

   // Or you may have meant to create an object here
   // Hashtable table = new Hashtable();

   public void Func(object o, string p)
   {
      // Or here
      // table = new Hashtable();
      table[p] = o;
   }

   public static void Main()
   {
   }
}
```

编译器警告（等级 4）CS1573

项目 • 2023/04/07

参数“parameter”在用于“parameter”的 XML 注释中没有匹配的 param 标记（但其他参数有）

使用 DocumentationFile 编译器选项时，只为某些而非全部方法中的参数指定了注释。你可能忘记了为这些参数输入注释。

下面的示例生成 CS1573：

C#

```
// CS1573.cs
// compile with: /W:4
public class MyClass
{
   /// <param name='Int1'>Used to indicate status.</param>
   // enter a comment for Char1?
   public static void MyMethod(int Int1, char Char1)
   {
   }

   public static void Main ()
   {
   }
}
```

编译器警告（等级 4）CS1591

项目 • 2023/04/07

缺少用于公开可见类型或成员“Type_or_Member”的 XML 注释

已指定 DocumentationFile 编译器选项，但一个或多个构造没有注释。

下面的示例生成 CS1591：

C#

```
// CS1591.cs
// compile with: /W:4 /doc:x.xml

/// text
public class Test
{
   // /// text
   public static void Main()   // Try the following instead: remove "//"
from previous line (`// /// text` about the comments)
                               //or add #pragma warning disable 1591
                               //a good practice is add the following
comments:
                               // /// <summary>
                               // ///
                               // /// </summary>
                               // this can be easily achieve at Visual
Studio in a simple way adding triple forward slashes (`///`)
   {
   }
}
```

编译器警告（等级 4）CS1610

项目 • 2023/04/07

无法删除用于默认 Win32 资源的临时文件“file”-- resource

使用 Win32Resource 编译器选项时，以及当 %TEMP% 目录不具有删除权限时，此警告表示编译器无法删除其创建的临时文件。

请确保具有 %TEMP% 目录的读/写/删除权限。

如有必要，可以手动删除这些文件，这不会对 C# 或任何程序造成损害。

编译器警告（等级 4）CS1712

项目 • 2023/04/07

类型参数“type parameter”在“type”的 XML 注释中没有匹配的 typeparam 标记（但其他类型参数有）

泛型类型的文档缺少 **typeparam** 标记。 有关详细信息，请参阅 <typeparam>。

示例

下面的代码生成 CS1712。 若要解决此错误，请为类型参数 S 添加 **typeparam** 标记。

C#

```
// CS1712.cs
// compile with: /doc:cs1712.xml
using System;
class Test
{
    public static void Main() {}
    /// <param name="j"> This is the j parameter.</param>
    /// <typeparam name="T"> This is the T type parameter.</typeparam>
    public void bar<T,S>(int j) {}  // CS1712
}
```

C# 文档

了解如何在 .NET 平台上使用 C# 编程语言编写任何应用程序。

了解如何使用 C# 编程

开始使用

了解 C# | 教程、课程、视频等

VIDEO

C# 初学者视频系列

教程

自引导式教程

浏览器内教程

参考

Q&A 上的 C#

.NET 技术社区论坛上的语言

Stack Overflow 上的 C#

Discord 上的 C#

C# 基础知识

概述

C# 教程

C# 程序内部探究

C# 突出显示视频系列

概念

类型系统

面向对象的编程

功能技术

异常

编码样式

教程

显示命令行

类简介

面向对象的 C#

转换类型

模式匹配

使用 LINQ 查询数据

新增功能

新变化

C# 13 中的新增功能

C# 12 中的新增功能

C# 11 中的新增功能

C# 10 中的新增功能

教程

浏览记录类型

探索顶级语句

探索新模式

编写自定义字符串内插处理程序

参考

C# 编译器中的重大更改

版本兼容性

关键概念

概述

C# 语言策略

编程概念

概念

语言集成查询 (LINQ)

异步编程

高级概念

参考

反射和属性

表达式树

本机互操作性

性能工程

.NET 编译器平台 SDK

保持联系

参考

.NET 开发人员社区

YouTube

Twitter

规范

阅读有关 C# 语言的详细规范和有关最新功能的详细规范。

ECMA 规范和最新功能

概述

规范流程

详细的 ECMA 规范内容

参考

最新功能规范

C# ECMA 规范草稿 - 介绍性材料

参考

前言

介绍

参考

范围

规范引用

术语和定义

常规说明

一致性

C# ECMA 规范草稿 - 语言规范

参考

词法结构

基本概念

类型

变量

转换

模式

表达式

语句

参考

命名空间

类

结构

数组

接口

枚举

委托

异常

特性

不安全代码

C# ECMA 规范草稿 - 附录

参考

语法

可移植性问题

标准库

文档注释

参考文献