C#8有什么新特性?

C# 8.0 向 C# 语言添加了以下功能和增强功能：

Readonly 成员

可将 `readonly` 修饰符应用于结构的成员。 它指示该成员不会修改状态。 这比将 `readonly` 修饰符应用于 `struct` 声明更精细。 请考虑以下可变结构:

```
public struct Point
{
    public double X { get; set; }
    public double Y { get; set; }
    public double Distance => Math.Sqrt(X * X + Y * Y);

    public override string ToString() =>
        $"({X}, {Y}) is {Distance} from the origin";
}
```

与大多数结构一样，`ToString()` 方法不会修改状态。 可以通过将 `readonly` 修饰符添加到 `ToString()` 的声明来对此进行指示:

```
public readonly override string ToString() =>
    $"({X}, {Y}) is {Distance} from the origin";
```

上述更改会生成编译器警告，因为 `ToString` 访问未标记为 `readonly` 的 `Distance` 属性:

```
warning CS8656: Call to non-readonly member 'Point.Distance.get' from a 'readonly' member resu
```

需要创建防御性副本时，编译器会发出警告。 `Distance` 属性不会更改状态，因此可以通过将 `readonly` 修饰符添加到声明来修复此警告:

```
public readonly double Distance => Math.Sqrt(X * X + Y * Y);
```

注意，`readonly` 修饰符对于只读属性是必需的。 编译器会假设 `get` 访问器可以修改状态；必须显式声明 `readonly`。 自动实现的属性是一个例外；编译器会将所有自动实现的 Getter 视为 `readonly`，因此，此处无需向 `X` 和 `Y` 属性添加 `readonly` 修饰符。

编译器确实会强制执行 `readonly` 成员不修改状态的规则。 除非删除 `readonly` 修饰符，否则不会编译以下方法:

```
public readonly void Translate(int xOffset, int yOffset)
{
    X += xOffset;
    Y += yOffset;
}
```

通过此功能，可以指定设计意图，使编译器可以强制执行该意图，并基于该意图进行优化。

默认接口方法

现在可以将成员添加到接口，并为这些成员提供实现。 借助此语言功能，API 作者可以将方法添加到以后版本的接口中，而不会破坏与该接口当前实现的源或二进制文件兼容性。 现有的实现继承默认实现。 此功能使 C# 与面向 Android 或 Swift 的 API 进行互操作，此类 API 支持类似功能。 默认接口方法还支持类似于"特征"语言功能的方案。

在更多位置中使用更多模式

模式匹配提供了在相关但不同类型的数据中提供形状相关功能的工具。 C# 7.0 通过使用 `is` 表达式和 `switch` 语句引入了类型模式和常量模式的语法。 这些功能代表了支持数据和功能分离的编程范例的初步尝试。 随着行业转向更多微服务和其他基于云的体系结构，还需要其他语言工具。

C# 8.0 扩展了此词汇表，这样就可以在代码中的更多位置使用更多模式表达式。 当数据和功能分离时，请考虑使用这些功能。 当算法依赖于对象运行时类型以外的事实时，请考虑使用模式匹配。 这些技术提供了另一种表达设计的方式。

除了可以在新位置使用新模式之外，C# 8.0 还添加了"递归模式"。 任何模式表达式的结果都是一个表达式。 递归模式只是应用于另一个模式表达式输出的模式表达式。

switch 表达式

通常情况下，switch语句在其每个 `case` 块中生成一个值。 借助 Switch 表达式，可以使用更简洁的表达式语法。 只有些许重复的 `case` 和 `break` 关键字和大括号。 以下面列出彩虹颜色的枚举为例:

```
public enum Rainbow
{
    Red,
    Orange,
    Yellow,
    Green,
    Blue,
    Indigo,
    Violet
}
```

如果应用定义了通过 `R`、`G` 和 `B` 组件构造而成的 `RGBColor` 类型，可使用以下包含 switch 表达式的方法，将 `Rainbow` 转换为 RGB 值:

```
public static RGBColor FromRainbow(Rainbow colorBand) =>
    colorBand switch
    {
        Rainbow.Red    => new RGBColor(0xFF, 0x00, 0x00),
        Rainbow.Orange => new RGBColor(0xFF, 0x7F, 0x00),
        Rainbow.Yellow => new RGBColor(0xFF, 0xFF, 0x00),
        Rainbow.Green  => new RGBColor(0x00, 0xFF, 0x00),
        Rainbow.Blue   => new RGBColor(0x00, 0x00, 0xFF),
        Rainbow.Indigo => new RGBColor(0x4B, 0x00, 0x82),
        Rainbow.Violet => new RGBColor(0x94, 0x00, 0xD3),
        _              => throw new ArgumentException(message: "invalid enum value", paramName
    };
```

这里有几个语法改进:

- 变量位于 `switch` 关键字之前。 不同的顺序使得在视觉上可以很轻松地区分 switch 表达式和 switch 语句。
- 将 `case` 和 `:` 元素替换为 `=>`。 它更简洁，更直观。
- 将 `default` 事例替换为 `_` 弃元。
- 正文是表达式，不是语句。

将其与使用经典 `switch` 语句的等效代码进行对比:

```
public static RGBColor FromRainbowClassic(Rainbow colorBand)
{
    switch (colorBand)
    {
        case Rainbow.Red:
            return new RGBColor(0xFF, 0x00, 0x00);
        case Rainbow.Orange:
            return new RGBColor(0xFF, 0x7F, 0x00);
        case Rainbow.Yellow:
            return new RGBColor(0xFF, 0xFF, 0x00);
        case Rainbow.Green:
            return new RGBColor(0x00, 0xFF, 0x00);
        case Rainbow.Blue:
            return new RGBColor(0x00, 0x00, 0xFF);
        case Rainbow.Indigo:
            return new RGBColor(0x4B, 0x00, 0x82);
        case Rainbow.Violet:
            return new RGBColor(0x94, 0x00, 0xD3);
        default:
            throw new ArgumentException(message: "invalid enum value", paramName: nameof(colorBand));
    };
}
```

属性模式

借助属性模式，可以匹配所检查的对象的属性。 请看一个电子商务网站的示例，该网站必须根据买家地址计算销售税。 这种计算不是 `Address` 类的核心职责。 它会随时间变化，可能比地址格式的更改更频繁。 销售税的金额取决于地址的 `State` 属性。 下面的方法使用属性模式从地址和价格计算销售税:

```
public static decimal ComputeSalesTax(Address location, decimal salePrice) =>
    location switch
    {
        { State: "WA" } => salePrice * 0.06M,
        { State: "MN" } => salePrice * 0.075M,
        { State: "MI" } => salePrice * 0.05M,
        // other cases removed for brevity...
        _ => 0M
    };
```

模式匹配为表达此算法创建了简洁的语法。

元组模式

一些算法依赖于多个输入。 使用元组模式，可根据表示为元组的多个值进行切换。 以下代码显示了游戏"rock, paper, scissors（石头剪刀布）"的切换表达式：:

```
public static string RockPaperScissors(string first, string second)
    => (first, second) switch
    {
        ("rock", "paper") => "rock is covered by paper. Paper wins.",
        ("rock", "scissors") => "rock breaks scissors. Rock wins.",
        ("paper", "rock") => "paper covers rock. Paper wins.",
        ("paper", "scissors") => "paper is cut by scissors. Scissors wins.",
        ("scissors", "rock") => "scissors is broken by rock. Rock wins.",
        ("scissors", "paper") => "scissors cuts paper. Scissors wins.",
        (_, _) => "tie"
    };
```

消息指示获胜者。 弃元表示平局(石头剪刀布游戏)的三种组合或其他文本输入。

位置模式

某些类型包含 `Deconstruct` 方法，该方法将其属性解构为离散变量。 如果可以访问 `Deconstruct` 方法，就可以使用位置模式检查对象的属性并将这些属性用于模式。 考虑以下 `Point` 类，其中包含用于为 `X` 和 `Y` 创建离散变量的 `Deconstruct` 方法:

```
public class 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);
}
```

此外，请考虑以下表示象限的各种位置的枚举:

```
public enum Quadrant
{
    Unknown,
    Origin,
    One,
    Two,
    Three,
    Four,
    OnBorder
}
```

下面的方法使用位置模式来提取 `x` 和 `y` 的值。 然后，它使用 `when` 子句来确定该点的 `Quadrant`:

```
static Quadrant GetQuadrant(Point point) => point switch
{
    (0, 0) => Quadrant.Origin,
    var (x, y) when x > 0 && y > 0 => Quadrant.One,
    var (x, y) when x < 0 && y > 0 => Quadrant.Two,
    var (x, y) when x < 0 && y < 0 => Quadrant.Three,
    var (x, y) when x > 0 && y < 0 => Quadrant.Four,
    var (_, _) => Quadrant.OnBorder,
    _ => Quadrant.Unknown
};
```

当 `x` 或 `y` 为 0（但不是两者同时为 0）时，前一个开关中的弃元模式匹配。 Switch 表达式必须要么生成值，要么引发异常。 如果这些情况都不匹配，则 switch 表达式将引发异常。 如果没有在 switch 表达式中涵盖所有可能的情况，编译器将生成一个警告。

using 声明

using 声明是前面带 `using` 关键字的变量声明。 它指示编译器声明的变量应在封闭范围的末尾进行处理。 以下面编写文本文件的代码为例:

```
static int WriteLinesToFile(IEnumerable<string> lines)
{
    using var file = new System.IO.StreamWriter("WriteLines2.txt");
    int skippedLines = 0;
    foreach (string line in lines)
    {
        if (!line.Contains("Second"))
        {
            file.WriteLine(line);
        }
        else
        {
            skippedLines++;
        }
    }
    // Notice how skippedLines is in scope here.
    return skippedLines;
    // file is disposed here
}
```

在前面的示例中，当到达方法的右括号时，将对该文件进行处理。 这是声明 `file` 的范围的末尾。 前面的代码相当于下面使用经典using语句的代码:

```
static int WriteLinesToFile(IEnumerable<string> lines)
{
    using (var file = new System.IO.StreamWriter("WriteLines2.txt"))
    {
        int skippedLines = 0;
        foreach (string line in lines)
        {
            if (!line.Contains("Second"))
            {
                file.WriteLine(line);
            }
            else
            {
                skippedLines++;
            }
        }
        return skippedLines;
    } // file is disposed here
}
```

在前面的示例中，当到达与 `using` 语句关联的右括号时，将对该文件进行处理。

在这两种情况下，编译器将生成对 `Dispose()` 的调用。 如果 `using` 语句中的表达式不可用，编译器将生成一个错误。

静态本地函数

现在可以向本地函数添加 `static` 修饰符，以确保本地函数不会从封闭范围捕获（引用）任何变量。 这样做会生成 `CS8421`，"静态本地函数不能包含对 <variable> 的引用。"

考虑下列代码。 本地函数 `LocalFunction` 访问在封闭范围（方法 `M`）中声明的变量 `y`。 因此，不能用 `static` 修饰符来声明 `LocalFunction`:

```
int M()
{
    int y;
    LocalFunction();
    return y;

    void LocalFunction() => y = 0;
}
```

下面的代码包含一个静态本地函数。 它可以是静态的，因为它不访问封闭范围中的任何变量:

```
int M()
{
    int y = 5;
    int x = 7;
    return Add(x, y);

    static int Add(int left, int right) => left + right;
}
```

可处置的 ref 结构

用 `ref` 修饰符声明的 `struct` 可能无法实现任何接口，因此无法实现 IDisposable 因此，要能够处理 `ref struct`，它必须有一个可访问的 `void Dispose()` 方法。 此功能同样适用于 `readonly ref struct` 声明。

可为空引用类型

在可为空注释上下文中，引用类型的任何变量都被视为不可为空引用类型。 若要指示一个变量可能为 null，必须在类型名称后面附加 `?`，以将该变量声明为可为空引用类型。

对于不可为空引用类型，编译器使用流分析来确保在声明时将本地变量初始化为非 Null 值。 字段必须在构造过程中初始化。 如果没有通过调用任何可用的构造函数或通过初始化表达式来设置变量，编译器将生成警告。 此外，不能向不可为空引用类型分配一个可以为 Null 的值。

不对可为空引用类型进行检查以确保它们没有被赋予 Null 值或初始化为 Null。 不过，编译器使用流分析来确保可为空引用类型的任何变量在被访问或分配给不可为空引用类型之前，都会对其 Null 性进行检查。

异步流

从 C# 8.0 开始，可以创建并以异步方式使用流。 返回异步流的方法有三个属性:

1. 它是用 `async` 修饰符声明的。
2. 它将返回 IAsyncEnumerable<T>。
3. 该方法包含用于在异步流中返回连续元素的 `yield return` 语句。

使用异步流需要在枚举流元素时在 `foreach` 关键字前面添加 `await` 关键字。 添加 `await` 关键字需要枚举异步流的方法，以使用 `async` 修饰符进行声明并返回 `async` 方法允许的类型。 通常这意味着返回 Task 或 Task<TResult>。 也可以为 ValueTask 或 ValueTask<TResult>。 方法既可以使用异步流，也可以生成异步流，这意味着它将返回 IAsyncEnumerable<T>。 下面的代码生成一个从 0 到 19 的序列，在生成每个数字之间等待 100 毫秒:

```
public static async System.Collections.Generic.IAsyncEnumerable<int> GenerateSequence()
{
    for (int i = 0; i < 20; i++)
    {
        await Task.Delay(100);
        yield return i;
    }
}
```

可以使用 `await foreach` 语句来枚举序列:

```
await foreach (var number in GenerateSequence())
{
    Console.WriteLine(number);
}
```

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。

异步可释放

从 C# 8.0 开始，语言支持实现 System.IAsyncDisposable接口的异步可释放类型。 可使用 `await using` 语句来处理异步可释放对象。

索引和范围

索引和范围为访问序列中的单个元素或范围提供了简洁的语法。

此语言支持依赖于两个新类型和两个新运算符:

- `System.Index` 表示一个序列索引。
- 来自末尾运算符 `^` 的索引，指定一个索引与序列末尾相关。
- `System.Range` 表示序列的子范围。
- 范围运算符 `..`，用于指定范围的开始和末尾，就像操作数一样。

让我们从索引规则开始。 请考虑数组 `sequence`。 `0` 索引与 `sequence[0]` 相同。 `^0` 索引与 `sequence[sequence.Length]` 相同。 请注意，`sequence[^0]` 不会引发异常，就像 `sequence[sequence.Length]` 一样。 对于任何数字 `n`，索引 `^n` 与 `sequence.Length - n` 相同。

范围指定范围的开始和末尾。 包括此范围的开始，但不包括此范围的末尾，这表示此范围包含开始但不包含末尾。 范围 `[0..^0]` 表示整个范围，就像 `[0..sequence.Length]` 表示整个范围。

请看以下几个示例。 请考虑以下数组，用其顺数索引和倒数索引进行注释:

```
var words = new string[]
{
                // index from start    index from end
    "The",      // 0                   ^9
    "quick",    // 1                   ^8
    "brown",    // 2                   ^7
    "fox",      // 3                   ^6
    "jumped",   // 4                   ^5
    "over",     // 5                   ^4
    "the",      // 6                   ^3
    "lazy",     // 7                   ^2
    "dog"       // 8                   ^1
};              // 9 (or words.Length) ^0
```

可以使用 `^1` 索引检索最后一个词:

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

以下代码创建了一个包含单词"quick"、"brown"和"fox"的子范围。 它包括 `words[1]` 到 `words[3]`。 元素 `words[4]` 不在该范围内。

```
var quickBrownFox = words[1..4];
```

以下代码使用"lazy"和"dog"创建一个子范围。 它包括 `words[^2]` 和 `words[^1]`。 末尾索引 `words[^0]` 不包括在内:

```
var lazyDog = words[^2..^0];
```

下面的示例为开始和/或结束创建了开放范围:

```
var allWords = words[..]; // contains "The" through "dog".
var firstPhrase = words[..4]; // contains "The" through "fox"
var lastPhrase = words[6..]; // contains "the", "lazy" and "dog"
```

此外可以将范围声明为变量:

```
Range phrase = 1..4;
```

然后可以在 `[` 和 `]` 字符中使用该范围:

```
var text = words[phrase];
```

不仅数组支持索引和范围。 还可以将索引和范围用于 string、Span<T> 或 ReadOnlySpan<T>。

Null 合并赋值

C# 8.0 引入了 null 合并赋值运算符 `??=`。 仅当左操作数计算为 `null` 时，才能使用运算符 `??=` 将其右操作数的值分配给左操作数。

```
List<int> numbers = null;
int? i = null;

numbers ??= new List<int>();
numbers.Add(i ??= 17);
numbers.Add(i ??= 20);

Console.WriteLine(string.Join(" ", numbers));  // output: 17 17
Console.WriteLine(i);  // output: 17
```

非托管构造类型

在 C# 7.3 及更低版本中，构造类型（包含至少一个类型参数的类型）不能为非托管类型。 从 C# 8.0 开始，如果构造的值类型仅包含非托管类型的字段，则该类型不受管理。

例如，假设泛型 Coords<T> 类型有以下定义

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

Coords<int> 类型为 C# 8.0 及更高版本中的非托管类型。 与任何非托管类型一样，可以创建指向此类型的变量的指针，或针对此类型的实例在堆栈上分配内存块:

```
Span<Coords<int>> coordinates = stackalloc[]
{
    new Coords<int> { X = 0, Y = 0 },
    new Coords<int> { X = 0, Y = 3 },
    new Coords<int> { X = 4, Y = 0 }
};
```

嵌套表达式中的 stackalloc

从 C# 8.0 开始，如果 stackalloc 表达式的结果为 System.Span<T> 或 System.ReadOnlySpan<T> 类型，则可以在其他表达式中使用 `stackalloc` 表达式:

```
Span<int> numbers = stackalloc[] { 1, 2, 3, 4, 5, 6 };
var ind = numbers.IndexOfAny(stackalloc[] { 2, 4, 6, 8 });
Console.WriteLine(ind);  // output: 1
```

内插逐字字符串的增强功能

内插逐字字符串中 `$` 和 `@` 标记的顺序可以任意安排: `$@"..."` 和 `@$"..."` 均为有效的内插逐字字符串。 在早期 C# 版本中，`$` 标记必须出现在 `@` 标记之前。