C#7.3 新增功能

系列目录 【已更新最新开发文章，点击查看详细】

C# 7.3 版本有两个主要主题。 第一个主题提供使安全代码的性能与不安全代码的性能一样好的功能。 第二个主题提供对现有功能的增量改进。 此外，在此版本中添加了新的编译器选项。

以下新增功能支持使安全代码获得更好的性能的主题：

- 无需固定即可访问固定的字段。
- 可以重新分配 `ref` 本地变量。
- 可以使用 `stackalloc` 数组上的初始值设定项。
- 可以对支持模式的任何类型使用 `fixed` 语句。
- 可以使用其他泛型约束。

对现有功能进行了以下增强：

- 可以使用元组类型测试 `==` 和 `!=`。
- 可以在多个位置使用表达式变量。
- 可以将属性附加到自动实现的属性的支持字段。
- 由 `in` 区分的参数的方法解析得到了改进。
- 重载解析的多义情况现在变得更少。

新的编译器选项为：

- `-publicsign`，用于启用程序集的开放源代码软件 (OSS) 签名。
- `-pathmap` 用于提供源目录的映射。

01 启用更高效的安全代码

你应能够安全地编写性能与不安全代码一样好的 C# 代码。 安全代码可避免错误类，例如缓冲区溢出、杂散指针和其他内存访问错误。 这些新功能扩展了可验证安全代码的功能。努力使用安全结构编写更多代码。 这些功能使其更容易实现。

1.1 索引 fixed 字段不需要进行固定

定义一个结构体

```
unsafe struct S
{
    public fixed int myFixedField[10];
}
```

在早期版本的 C# 中，需要固定变量才能访问属于 `myFixedField` 的整数之一。 现在，以下代码进行编译，而不将变量 `p` 固定到单独的 `fixed` 语句中：

```
class C
{
    static S s = new S();

    unsafe public void M()
    {
        int p = s.myFixedField[5];
    }
}
```

变量 `p` 访问 `myFixedField` 中的一个元素。 无需声明单独的 `int*` 变量。 请注意，你仍然需要 `unsafe` 上下文。 在早期版本的 C# 中，需要声明第二个固定的指针：

```
class C
{
    static S s = new S();

    unsafe public void M()
    {
        fixed (int* ptr = s.myFixedField)
        {
            int p = ptr[5];
        }
    }
}
```

有关详细信息，请参阅有关 `fixed` 语句 的文章。

1.2 可能会重新分配 ref 局部变量

现在，在对 `ref` 局部变量进行初始化后，可能会对其重新分配，以引用不同的实例。 以下代码现在编译：

```
ref VeryLargeStruct refLocal = ref veryLargeStruct; // 初始化
refLocal = ref anotherVeryLargeStruct;              // 重新分配后，反射引用不同的存储。
```

有关详细信息，请参阅有关 `ref` 返回和 `ref` 局部变量 以及 `foreach` 的文章。

1.3 stackalloc 数组支持初始值设定项

当你对数组中的元素的值进行初始值设定时，你已能够指定该值：

```
var arr =  new int[3] {1, 2, 3};
var arr2 = new int[] {1, 2, 3};
```

现在，可向使用 `stackalloc` 进行声明的数组应用同一语法：

```
int* pArr = stackalloc int[3] {1, 2, 3};
int* pArr2 = stackalloc int[] {1, 2, 3};
Span<int> arr = stackalloc [] {1, 2, 3};
```

有关详细信息，请参阅 `stackalloc` 运算符 一文。

1.4 更多类型支持 fixed 语句

`fixed` 语句支持有限的一组类型。 从 C# 7.3 开始，任何包含返回 `ref T` 或 `ref readonly T` 的 `GetPinnableReference()` 方法的类型均有可能为 `fixed`。 添加此功能意味着 `fixed` 可与 System.Span<T> 和相关类型配合使用。

有关详细信息，请参阅语言参考中的 `fixed` 语句 一文。

1.5 增强的泛型约束

现在，可以将类型 System.Enum 或 System.Delegate 指定为类型参数的基类约束。

现在也可以使用新的 `unmanaged` 约束来指定类型参数必须为“非托管类型”。 “非托管类型”不是引用类型，且在任何嵌套级别都不包含任何引用类型。

有关详细信息，请参阅有关 `where` 泛型约束 和 类型参数的约束 的文章。

将这些约束添加到现有类型是 不兼容的更改。 封闭式泛型类型可能不再满足这些新约束的要求。

02 提升了现有功能

以下功能提供了对语言中的功能的改进。 这些功能提升了在编写 C# 时的效率。

2.1 元组支持 == 和 !=

C# 元组类型现在支持 `==` 和 `!=`。 有关详细信息，请参阅有关 元组 一文中的转换 等式 部分。

2.2 将特性添加到自动实现的属性的支持字段

现在支持此语法：

```
[field: SomeThingAboutFieldAttribute]
public int SomeProperty { get; set; }
```

属性 `SomeThingAboutFieldAttribute` 应用于编译器生成的 `SomeProperty` 的支持字段。 有关详细信息，请参阅 C# 编程指南中的 属性。

2.3 in 方法重载解析决胜属性

在添加 `in` 参数修饰符时，这两个方法将导致多义性：

```
static void M(S arg);
static void M(in S arg);
```

现在，通过值（前面示例中的第一个）的重载比通过只读引用版本的重载更好。 若要使用只读引用参数调用版本，必须在调用方法前添加 `in` 修饰符。

有关详细信息，请参阅有关 `in` 参数修饰符 的文章。

2.4 扩展初始值设定项中的表达式变量

已对在 C# 7.0 中添加的允许 `out` 变量声明的语法进行了扩展，以包含字段初始值设定项、属性初始值设定项、构造函数初始值设定项和查询子句。 它允许使用如以下示例中所示的代码：

```
public class B
{
   public B(int i, out int j)
   {
      j = i;
   }
}

public class D : B
{
   public D(int i) : base(i, out var j)
   {
      Console.WriteLine($"The value of 'j' is {j}");
   }
}
```

2.5 改进了重载候选项

在每个版本中，对重载解析规则进行了更新，以解决多义方法调用具有“明显”选择的情况。此版本添加了三个新规则，以帮助编译器选取明显的选择：

1. 当方法组同时包含实例和静态成员时，如果方法在不含实例接收器或上下文的情况下被调用，则编译器将丢弃实例成员。 如果方法在含有实例接收器的情况下被调用，则编译器将丢弃静态成员。 在没有接收器时，编译器将仅添加静态上下文中的静态成员，否则，将同时添加静态成员和实例成员。 当接收器是不明确的实例或类型时，编译器将同时添加两者。 静态上下文（其中隐式 `this` 实例接收器无法使用）包含未定义 `this` 的成员的正文（例如，静态成员），以及不能使用 `this` 的位置（例如，字段初始值设定项和构造函数初始值设定项）。
2. 当一个方法组包含类型参数不满足其约束的某些泛型方法时，这些成员将从候选集中移除。
3. 对于方法组转换，返回类型与委托的返回类型不匹配的候选方法将从集中移除。

你将注意到此更改，因为当你确定哪个方法更好时，你将发现多义方法重载具有更少的编译器错误。

03 新的编译器选项

新的编译器选项支持 C# 程序的新版本和 DevOps 方案。

3.1 公共或开放源代码签名

`-publicsign` 编译器选项指示编译器使用公钥对程序集进行签名。 程序集被标记为已签名，但签名取自公钥。 此选项使你能够使用公钥在开放源代码项目中构建签名的程序集。

有关详细信息，请参阅 -publicsign 编译器选项 一文。

3.2 pathmap

`-pathmap` 编译器选项指示编译器将生成环境中的源路径替换为映射的源路径。 `-pathmap` 选项控制由编译器编写入 PDB 文件或为 CallerFilePathAttribute 编写的源路径。

有关详细信息，请参阅 -pathmap 编译器选项 一文。