告知我们有关下载 PDF 体验的信息。

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

C# 语言介绍

项目 • 2024/05/09

C# 语言是适用于 .NET 平台（免费的跨平台开源开发环境）的最流行语言。 C# 程序可以在许多不同的设备上运行，从物联网 (IoT) 设备到云以及介于两者之间的任何设备。 可为手机、台式机、笔记本电脑和服务器编写应用。

C# 是一种跨平台的通用语言，可以让开发人员在编写高性能代码时提高工作效率。 C# 是数百万开发人员中最受欢迎的 .NET 语言。 C# 在生态系统和所有 .NET 工作负载中具有广泛的支持。 基于面向对象的原则，它融合了其他范例中的许多功能，尤其是函数编程。 低级功能支持高效方案，无需编写不安全的代码。 大多数 .NET 运行时和库都是用 C# 编写的，C# 的进步通常会使所有 .NET 开发人员受益。

Hello world

“Hello, World”程序历来都用于介绍编程语言。 下面展示了此程序的 C# 代码：

C#

```
// This line prints "Hello, World"
Console.WriteLine("Hello, World");
```

以 `//` 开头的行是单行注释。 C# 单行注释以 `//` 开头，持续到当前行的末尾。 C# 还支持多行注释。 多行注释以 `/*` 开头，以 `*/` 结尾。 `System` 命名空间中的 `Console` 类的 `WriteLine` 方法生成程序的输出。 此类由标准类库提供，默认情况下，每个 C# 程序中会自动引用这些库。

以上示例显示了“Hello，World”程序的一个窗体，其中使用了顶级语句。 早期版本的 C# 要求在方法中定义程序的入口点。 此格式仍然有效，你将在许多现有 C# 示例中看到它。 你也应该熟悉此窗体，如以下示例所示：

C#

```
using System;

class Hello
{
    static void Main()
    {
        // This line prints "Hello, World"
        Console.WriteLine("Hello, World");
    }
}
```

此版本显示了在程序中使用的构建基块。 “Hello, World”程序始于引用 `System` 命名空间的 `using` 指令。 命名空间提供了一种用于组织 C# 程序和库的分层方法。 命名空间包含类型和其他命名空间。例如，`System` 命名空间包含许多类型（如程序中引用的 `Console` 类）和其他许多命名空间（如 `IO` 和 `Collections`）。 借助引用给定命名空间的 `using` 指令，可以非限定的方式使用作为相应命名空间成员的类型。 由于使用 `using` 指令，因此程序可以使用 `Console.WriteLine` 作为 `System.Console.WriteLine` 的简写。 在前面的示例中，该命名空间是隐式包含的。

“Hello, World”程序声明的 `Hello` 类只有一个成员，即 `Main` 方法。 `Main` 方法使用 `static` 修饰符进行声明。 实例方法可以使用关键字 `this` 引用特定的封闭对象实例，而静态方法则可以在不引用特定对象的情况下运行。 按照惯例，当没有顶级语句时，名为 `Main` 的静态方法将充当 C# 程序的入口点。

这两个入口点窗体生成等效的代码。 使用顶级语句时，编译器会合成程序入口点的包含类和方法。

> **提示**
>
> 本文中的示例帮助你初步了解 C# 代码。 某些示例可能会显示你不熟悉的 C# 元素。 当准备好学习 C# 时，请从我们的**初学者教程**开始，或通过每个部分中的链接学习深度知识。 如果你拥有 Java、JavaScript、TypeScript 或 Python 方面的经验，请阅读我们的提示，其中提供了快速学习 C# 所需的信息。

熟悉的 C# 功能

C# 对于初学者而言很容易上手，但同时也为经验丰富的专业应用程序开发人员提供了高级功能。 你很快就能提高工作效率。 你可以根据应用程序的需要学习更专业的技术。

C# 应用受益于 .NET 运行时的自动内存管理。 C# 应用还可以使用 .NET SDK 提供的丰富运行时库。 有些组件独立于平台，例如文件系统库、数据集合与数学库。 还有一些组件特定于单个工作负载，例如 ASP.NET Core Web 库或 .NET MAUI UI 库。 NuGet 的丰富开源生态系统增强了作为运行时一部分的库。 这些库提供更多可用的组件。

C# 属于 C 语言家族。 如果你使用过 C、C++、JavaScript 或 Java，那么也会熟悉 C# 语法。 与 C 语言家族中的所有语言一样，分号 (`;`) 定义语句的结束。 C# 标识符区分大小写。 C# 同样使用大括号（`{` 和 `}`）、控制语句（例如 `if`、`else` 和 `switch`）以及循环结构（例如 `for` 和 `while`）。 C# 还具有适用于任何集合类型的 `foreach` 语句。

C# 是一种强类型语言。 声明的每个变量都有一个在编译时已知的类型。 编译器或编辑工具会告诉你是否错误地使用了该类型。 可以在运行程序之前修复这些错误。 以下基础

数据类型内置于语言和运行时中：值类型（例如 `int`、`double`、`char`）、引用类型（例如 `string`）、数组和其他集合。 编写程序时，你会创建自己的类型。 这些类型可以是值的 `struct` 类型，也可以是定义面向对象的行为的 `class` 类型。 可以将 `record` 修饰符添加到 `struct` 或 `class` 类型，以便编译器合成用于执行相等性比较的代码。 还可以创建 `interface` 定义，用于定义实现该接口的类型必须提供的协定或一组成员。 还可以定义泛型类型和方法。 泛型使用类型参数为使用的实际类型提供占位符。

编写代码时，可以将函数（也称为方法）定义为 `struct` 和 `class` 类型的成员。 这些方法定义类型的行为。 可以使用不同数量或类型的参数来重载方法。 方法可以选择性地返回一个值。 除了方法之外，C# 类型还可以带有属性，即由称作访问器的函数支持的数据元素。 C# 类型可以定义事件，从而允许类型向订阅者通知重要操作。 C# 支持面向对象的技术，例如 `class` 类型的继承和多形性。

C# 应用使用异常来报告和处理错误。 如果你使用过 C++ 或 Java，则也会熟悉这种做法。 当无法执行预期的操作时，代码会引发异常。 其他代码（无论位于调用堆栈上面的多少个级别）可以选择性地使用 `try` - `catch` 块进行恢复。

独特的 C# 功能

你可能不太熟悉 C# 的某些元素。 语言集成查询 (LINQ) 提供一种基于模式的通用语法来查询或转换任何数据集合。 LINQ 统一了查询内存中集合、结构化数据（例如 XML 或 JSON）、数据库存储，甚至基于云的数据 API 的语法。 你只需学习一套语法即可搜索和操作数据，无论其存储在何处。 以下查询查找平均学分大于 3.5 的所有学生：

C#

```
var honorRoll = from student in Students
                where student.GPA > 3.5
                select student;
```

上面的查询适用于 `Students` 表示的许多存储类型。 它可以是对象的集合、数据库表、云存储 Blob 或 XML 结构。 相同的查询语法适用于所有存储类型。

使用基于任务的异步编程模型，可以编写看起来像是同步运行的代码，即使它是异步运行的。 它利用 `async` 和 `await` 关键字来描述异步方法，以及表达式何时进行异步计算。 以下示例等待异步 Web 请求。 异步操作完成后，该方法返回响应的长度：

C#

```
public static async Task<int> GetPageLengthAsync(string endpoint)
{
    var client = new HttpClient();
    var uri = new Uri(endpoint);
```

```
    byte[] content = await client.GetByteArrayAsync(uri);
    return content.Length;
}
```

C# 还支持使用 `await foreach` 语句来迭代由异步操作支持的集合，例如 GraphQL 分页 API。 以下示例以块的形式读取数据，并返回一个迭代器，该迭代器提供对每个可用元素的访问：

C#

```
public static async IAsyncEnumerable<int> ReadSequence()
{
    int index = 0;
    while (index < 100)
    {
        int[] nextChunk = await GetNextChunk(index);
        if (nextChunk.Length == 0)
        {
            yield break;
        }
        foreach (var item in nextChunk)
        {
            yield return item;
        }
        index++;
    }
}
```

调用方可以使用 `await foreach` 语句迭代该集合：

C#

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

C# 提供模式匹配。 这些表达式使你能够检查数据并根据其特征做出决策。 模式匹配为基于数据的控制流提供了极好的语法。 以下代码演示如何使用模式匹配语法来表达布尔 and、or 和 xor 运算的方法：

C#

```
public static bool Or(bool left, bool right) =>
    (left, right) switch
    {
        (true, true) => true,
        (true, false) => true,
        (false, true) => true,
```

```
        (false, false) => false,
    };

public static bool And(bool left, bool right) =>
    (left, right) switch
    {
        (true, true) => true,
        (true, false) => false,
        (false, true) => false,
        (false, false) => false,
    };
public static bool Xor(bool left, bool right) =>
    (left, right) switch
    {
        (true, true) => false,
        (true, false) => true,
        (false, true) => true,
        (false, false) => false,
    };
```

可以通过对任何值统一使用 `_` 来简化模式匹配表达式。 以下示例演示如何简化 and 方法：

C#

```
public static bool ReducedAnd(bool left, bool right) =>
    (left, right) switch
    {
        (true, true) => true,
        (_, _) => false,
    };
```

最后，作为 .NET 生态系统的一部分，你可以将 Visual Studio 或 Visual Studio Code 与 C# DevKit 配合使用。 这些工具可以全方位地理解 C# 语言，包括你编写的代码。 它们还提供调试功能。

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

Java 开发者学习 C# 的路线图

项目 • 2024/04/16

C# 和 Java 有许多相似之处。 学习 C# 时，可以应用许多已经掌握的 Java 编程知识：

1. 类似的语法：Java 和 C# 都属于 C 语言系列。 这种相似性意味着你已经可以阅读并理解 C#。 虽然存在一些差异，但大部分语法与 Java 和 C 相同。大括号和分号的用法类似。 `if`、`else`、`switch` 等控制语句相同。 循环语句 `for`、`while` 和 `do...while` 相同。 两种语言中 `class`、`struct` 和 `interface` 的关键字相同。 访问修饰符（包括 `public` 和 `private`）相同。 甚至许多内置类型都使用相同的关键字：`int`、`string` 和 `double`。
2. 面向对象的范例：Java 和 C# 都是面向对象的语言。 多形性、抽象和封装的概念在这两种语言中都适用。 两种语言都添加了新构造，但核心功能仍然相关。
3. 强类型化：Java 和 C# 都是强类型化语言。 可以显式或隐式声明变量的数据类型。 编译器会强制执行类型安全性。 在运行代码之前，编译器会捕获代码中与类型相关的错误。
4. 跨平台：Java 和 C# 都是跨平台语言。 你可以在喜欢的平台上运行开发工具。 应用程序可以在多个平台上运行。 开发平台不需要与目标平台匹配。
5. 异常处理：Java 和 C# 都通过引发异常来指示错误。 两者都使用 `try` - `catch` - `finally` 块来处理异常。 异常类具有类似的名称和继承层次结构。 一个区别是，C# 没有“已检查的异常”的概念。 任何方法（在理论上）都可能会引发任何异常。
6. 标准库：.NET 运行时和 Java 标准库 (JSL) 支持常见任务。 两者都有适用于其他开源包的广泛生态系统。 在 C# 中，包管理器是 NuGet。 它类似于 Maven。
7. 垃圾回收：这两种语言都通过垃圾回收来应用自动内存管理功能。 运行时从未引用的对象中回收内存。 一个区别在于，C# 支持创建值类型，就像 `struct` 类型一样。

由于这些相似性，几乎可以立即使用 C# 高效工作。 随着学习的深入，你应该会学到 Java 中没有的 C# 功能和习惯用法：

1. 模式匹配：模式匹配可以根据复杂数据结构的形状提供简洁的条件语句和表达式。 is 语句检查变量“是否”为某种模式。 基于模式的 switch 表达式提供了丰富的语法来检查变量并根据其特征做出决策。
2. 字符串插值和原始字符串字面量：字符串插值使你能够在字符串中插入已评估的表达式，而不是使用位置标识符。 原始字符串字面量可用于最小化文本中的转义序列。
3. 可以为 null 的类型和不可为 null 的类型：C# 通过向类型附加 `?` 后缀来支持可以为 null 的值类型和可以为 null 的引用类型。 对于可以为 null 的类型，如果在取消引用表达式之前不检查是否有 `null`，编译器会发出警告。 对于不可为 null 的类型，如果向该变量分配 `null` 值，编译器会发出警告。 不可为 null 的引用类型可最大程度减少引发 System.NullReferenceException 的编程错误。

4. 扩展方法：在 C# 中，可以创建扩展类或接口的方法。 扩展方法可扩展库中某个类型的行为或实现给定接口的所有类型的行为。
5. LINQ：语言集成查询 (LINQ) 提供了一种通用语法来查询和转换数据，无论其存储方式如何。
6. 本地函数：在 C# 中，可以在方法或其他本地函数内嵌套函数。 本地函数提供另一层封装。

C# 中还有一些 Java 中没有的功能。 你可看到 async 和 await 等功能，以及用于自动释放非内存资源的 using 语句。

C# 和 Java 之间还有一些类似的功能存在细微但重要的差异：

1. 属性和索引器：属性和索引器（将类视为数组或字典）都具有语言支持。 在 Java 中，它们是以 `get` 和 `set` 开头的方法命名约定。
2. 记录：在 C# 中，记录可以是 `class`（引用）类型，也可以是 `struct`（值）类型。 C# 记录可以是不可变的，但并非必须是不可变的。
3. 元组在 C# 和 Java 中具有不同的语法。
4. 属性类似于 Java 注释。

最后，有一些 Java 语言功能在 C# 中不可用：

1. 已检查的异常：在 C# 中，理论上任何方法都可能引发任何异常。
2. 已检查的数组协变：在 C# 中，数组不是安全协变的。 如果需要协变结构，则应使用泛型集合类和接口。

总的来说，有 Java 经验的开发者学习 C# 应该会很顺利。 你可发现非常熟悉的习惯用法，从而快速实现高效工作，并且很快就可学会新的习惯用法。

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 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

JavaScript 和 TypeScript 开发者学习 C# 的路线图

项目 • 2024/04/24

C#、TypeScript 和 JavaScript 都属于 C 系列语言。 这些语言之间的相似之处可在你使用 C# 时帮助你快速提高工作效率。

1. ***类似的语法***：JavaScript、TypeScript 和 C# 均属于 C 系列语言。 这种相似性意味着你已经可以阅读并理解 C#。 虽然存在一些差异，但大部分语法与 JavaScript 和 C 相同。大括号和分号的用法类似。 `if`、`else`、`switch` 等控制语句相同。 循环语句 `for`、`while` 和 `do`...`while` 相同。 C# 和 TypeScript 中 `class` 和 `interface` 的关键字相同。 TypeScript 和 C# 中的访问修饰符（包括 `public` 和 `private`）相同。
2. `=>` ***标记***：所有语言都支持轻型函数定义。 在 C# 中，它们被称为 *lambda 表达式*，在 JavaScript 中，它们通常被称为*箭头函数*。
3. ***函数层次结构***：这三种语言都支持局部函数，这些函数是其他函数中定义的函数。
4. ***Async / Await***：这三种语言使用相同的关键字（`async` 和 `await`）来进行异步编程。
5. ***垃圾回收***：这三种语言都依赖垃圾回收器进行自动内存管理。
6. ***事件模型***：C# 的 event 语法类似于 JavaScript 的文档对象模型 (DOM) 事件模型。
7. ***包管理器***：NuGet 是 C# 和 .NET 最常见的包管理器，类似于 JavaScript 应用程序的 npm。 系统会以程序集形式提供 C# 库。

继续学习 C# 时，你将了解不属于 JavaScript 的概念。 如果你使用 TypeScript，你可能会熟悉其中一些概念：

1. ***C# 类型系统***：C# 是一种强类型语言。 每个变量都有一个类型，并且该类型无法更改。 定义 `class` 或 `struct` 类型。 可以定义 interface 定义来定义其他类型实现的行为。 TypeScript 包括其中许多概念，但由于 TypeScript 是基于 JavaScript 构建而成，因此类型系统并不那么严格。
2. ***模式匹配***：模式匹配可以根据复杂数据结构的形状提供简洁的条件语句和表达式。 is 表达式检查变量“是否”为某种模式。 基于模式的 switch 表达式提供了丰富的语法来检查变量并根据其特征做出决策。
3. 字符串插值和原始字符串字面量：字符串插值使你能够在字符串中插入已评估的表达式，而不是使用位置标识符。 原始字符串字面量可用于最小化文本中的转义序列。
4. 可以为 null 的类型和不可为 null 的类型：C# 通过向类型附加 `?` 后缀来支持可以为 null 的值类型和可以为 null 的引用类型。 对于可以为 null 的类型，如果在取消引用表达式之前不检查是否有 `null`，编译器会发出警告。 对于不可为 null 的类型，如果向该变量分配 `null` 值，编译器会发出警告。 这些功能可以最大程度地避免应用

程序引发 System.NullReferenceException。 语法与在 TypeScript 中对可选属性使用 `?` 类似。

5. ***LINQ***：语言集成查询 (LINQ) 提供了一种通用语法来查询和转换数据，无论其存储方式如何。

随着了解的加深，其他差异就会变得明显，但其中许多差异的范围较小。

JavaScript 和 TypeScript 中的一些熟悉的功能和习惯用法不适用于 C#：

1. ***动态类型***：C# 使用静态类型。 变量声明包括该类型，并且该类型无法更改。 C# 中有一种提供运行时绑定的 dynamic 类型。
2. ***原型继承***：C# 继承是类型声明的一部分。 C# 中的 `class` 声明用于表示任意基类。 在 JavaScript 中，可以设置 `__proto__` 属性以在任意实例上设置基类型。
3. ***解释语言***：必须先编译 C# 代码，然后才能运行。 JavaScript 代码可以直接在浏览器中运行。

此外，还有一些 TypeScript 功能在 C# 中不可用：

1. ***联合类型***：C# 不支持联合类型。 不过，设计方案正在进行中。
2. ***修饰器***：C# 没有修饰器。 一些常见的修饰器（如 `@sealed`）是 C# 中的保留关键字。 其他常见修饰器可能具有相应的属性。 对于其他修饰器，你可以创建自己的属性。
3. ***更宽容的语法***：C# 编译器对代码的解析比 JavaScript 要求的更严格。

如果要生成 Web 应用程序，应考虑使用 Blazor 来构建应用程序。 Blazor 是为 .NET 和 C# 构建的全堆栈 Web 框架。 Blazor 组件可以作为 .NET 程序集或在使用 WebAssembly 的客户端上运行。 Blazor 支持与你偏好的 JavaScript 或 TypeScript 库的互操作。

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

Python 开发者学习 C# 的路线图

项目 • 2024/04/12

C# 和 Python 具备类似的概念。 如果你已经了解 Python，这些熟悉的狗仔可以帮助你学习 C#。

1. ***面向对象***：Python 和 C# 都是面向对象的语言。 Python 中类的所有概念都适用于 C#，即使语法不同也适用。
2. ***跨平台***：Python 和 C# 都是跨平台语言。 使用这两种语言中的任意一种编写的应用都可以在许多平台上运行。
3. ***垃圾回收***：这两种语言都通过垃圾收集来应用自动内存管理功能。 运行时从未引用的对象中回收内存。
4. ***强类型化***：Python 和 C# 都是强类型化语言。 类型强制不会隐式发生。 后面会介绍一些差异，因为 C# 是静态类型，而 Python 是动态类型。
5. ***Async / Await***：Python 的 `async` 和 `await` 功能直接受到 C# 的 `async` 和 `await` 支持的启发。
6. ***模式匹配***：Python 的 `match` 表达式和模式匹配类似于 C# 的模式匹配 `switch` 表达式。 可以使用这些功能来检查复杂数据表达式，以确定它是否与模式匹配。
7. ***语句关键字***：Python 和 C# 有许多相同关键字，例如 `if`、`else`、`while`、`for` 等。 虽然并非所有语法都相同，但它们十分相似，如果你了解 Python，就可以阅读 C#。

开始学习 C# 时，你将学习以下重要概念，这些概念在 C# 与 Python 中的含义不同：

1. ***缩进与标记***：在 Python 中，换行和缩进是一流的语法元素。 在 C# 中，空格并不重要。 标记，例如 `;` 用于分隔语句，其他标记如 `{` 和 `}` 用于控制 `if` 和其他块语句的块范围。 但是，为了便于阅读，大多数编码样式（包括这些文档中使用的样式）都使用缩进来强化 `{` 和 `}` 声明的块范围。
2. ***静态类型语言***：在 C# 中，变量声明包括其类型。 将变量重新分配给不同类型的对象将生成编译器错误。 在 python 中，在重新分配时，类型可能会发生更改。
3. ***可以为 null 的类型***：C# 变量*可为空*或*不可为空*。 不可为 null 的类型是不能为 null（或不包含任何类型）的类型。 它始终需要引用有效的对象。 相比之下，可以为 null 的类型可以引用有效对象或 null。
4. ***LINQ***：组成语言集成查询 (LINQ) 的查询表达式关键字与 Python 中的关键字不同。 但是，Python 库（如 `itertools`、`more-itertools` 和 `py-linq`）具备类似的功能。
5. ***泛型***：C# 泛型使用 C# 静态类型语言对类型参数提供的参数进行断言。 泛型算法可能需要指定参数类型必须满足的约束。

最后，有一些 Python 功能在 C# 中不可用：

1. ***结构化（鸭子）语言类型***：在 C# 中，类型具有名称和声明。 除元组外，具有相同结构的类型不可互换。
2. ***REPL***：C# 不具备读取-评估-打印循环 (REPL)，无法快速构建解决方案原型。
3. ***必不可少的空格***：需要正确使用大括号 `{` 和 `}` 来标准块范围。

如果你了解 Python，那么 C# 的学习过程会很顺利。 这些语言具有相似的概念和相似的习语。

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

带批注的 C# 策略

项目 • 2023/05/09

我们将不断改进 C# 以满足开发人员不断变化的需求，并保持最先进的编程语言。 我们将与负责 .NET 库、开发人员工具和工作负载支持的团队一起热切而广泛地进行创新，同时小心翼翼地遵循语言精神。 认识到使用 C# 的域的多样性，我们将首选能够让所有或大多数开发人员受益的语言和性能改进，并保持对向后兼容性的高度承诺。 我们将继续为更广泛的 .NET 生态系统提供支持，扩大其在 C# 未来中的作用，同时保持对设计决策的管理。

策略如何指导 C#

C# 策略指导我们做出有关 C# 演进的决策，这些批注提供了我们对关键语句的看法的见解。

> “我们将热切而广泛地进行创新”

C# 社区不断发展，C# 语言不断演进以满足社区的需求和期望。 我们从各种来源中汲取灵感，选择使大量 C# 开发人员受益的功能，在工作效率、可读性和性能方面提供一致改进的功能。

> “小心翼翼地遵循语言精神”

我们用 C# 语言的精神和历史来评估新想法。 我们优先考虑对大多数现有 C# 开发人员有意义的创新。

> “使所有或大多数开发人员受益的改进”

开发人员在所有 .NET 工作负载（例如 Web 前端和后端、云原生开发、桌面开发和构建跨平台应用程序）中使用 C#。 我们专注于直接或通过授权改进通用库而产生最大影响的新功能。 语言功能开发包括集成到我们的开发人员工具和学习资源中。

> “高度致力于向后兼容性”

我们尊重目前有大量的 C# 代码在使用这一事实。 任何可能的中断性更改都会根据 C# 社区中断的规模和产生的影响进行仔细考虑。

> “维护管理”

C# 语言设计 在社区参与的情况下公开进行。 任何人都可以在我们的 GitHub 存储库 中提出新的 C# 功能。 语言设计团队 在权衡社区意见后做出最终决定。

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

C# 简介

项目 • 2025/01/16

欢迎学习 C# 教程简介。 这些课程从可以在浏览器中运行的交互式代码开始。 在开始这些交互式课程之前，可以从 C# for Beginners 视频系列 中了解 C# 的基础知识。
https://www.youtube-nocookie.com/embed/9THmGiSPjBQ?si=3kUKFtOMLpEzeq7J

第一课介绍使用小代码片段的 C# 概念。 你将了解 C# 语法的基础知识，以及如何使用字符串、数字和布尔值等数据类型。 这一切都是交互式的，你将在几分钟内编写并运行代码。 这些第一课假定事先不知道编程或 C# 语言。

可以在不同的环境中试用这些教程。 你将学习的概念是相同的。 区别在于你喜欢的体验：

- 在浏览器中的文档平台上：此体验在文档页中嵌入可运行的 C# 代码窗口。 在浏览器中编写和执行 C# 代码。
- 在 Microsoft Learn 培训体验中。 此学习路径包含几个模块，这些模块教授 C# 的基础知识。
- 在本地计算机上。 联机浏览后，可在计算机上下载 .NET SDK 和生成程序。

可以使用联机浏览器体验或在自己的本地开发环境中学习 Hello World 课程之后的入门教程。 在每个教程结束时，你决定是要继续学习下一课在线还是在计算机上。 你可以使用链接来帮助设置你的环境，并在计算机上继续学习下一个教程。

你好，世界

在 Hello world 教程中，你将创建最基本的 C# 程序。 读者将探索 `string` 类型以及如何使用文本。 还可以使用 Microsoft Learn 培训上的路径。

C# 中的数字

在 C# 教程中的 数字中，你将了解计算机如何存储数字以及如何使用不同的数字类型执行计算。 你将了解舍入的基础知识，以及如何使用 C# 执行数学计算。 本教程也可下载到计算机本地进行学习。

本教程假定你已完成 Hello world 课程。

分支和循环

分支和循环 教程介绍了基于变量中存储的值选择不同代码执行路径的基础知识。 你将了解控制流的基础知识，这是程序如何做出决策并选择不同操作的基础。 本教程也可下载到计算机本地进行学习。

该教程假定你已完成 Hello World 和 C# 中的数字课程。

列表集合

列表集合课程将介绍存储一系列数据的列表集合类型。 你将了解如何添加和删除项目、搜索项目以及对列表进行排序。 你将浏览不同类型的列表。 本教程也可下载到计算机本地进行学习。

本教程假定你已完成上面列出的课程。

设置本地环境

项目 • 2023/05/10

在计算机上演练教程的第一步是设置开发环境。

- 建议使用 Visual Studio for Windows 或 Mac。 可以从 Visual Studio 下载页面 下载免费版本。 Visual Studio 包括 .NET SDK。
- 还可以使用 Visual Studio Code 编辑器。 需要单独安装最新的 .NET Compiler Platform SDK。
- 如果更想要使用其他编辑器，则需要安装最新的 .NET SDK。

基本的应用程序开发流

若要更好地理解这些教程中的说明，请使用 .NET CLI 来创建、生成和运行应用程序。 你将使用以下命令：

- dotnet new 创建应用程序。 此命令生成应用程序所需的文件和资产。 C# 简介的所有教程都使用 `console` 应用程序类型。 了解基础知识后，可以扩展到其他应用程序类型。
- dotnet build 生成可执行文件。
- dotnet run 运行可执行文件。

如果你在这些教程中使用 Visual Studio 2019，则当教程指导你运行这些 CLI 命令之一时，你将选择一个 Visual Studio 菜单选项：

- “文件”>“新建”>“项目”：创建应用程序。
  - 建议使用 `Console Application` 项目模板。
  - 系统会提供指定目标框架的选项。 如果面向的是 .NET 5 或更高版本，下面的教程更适用。
- “生成”>“生成解决方案”：生成可执行文件。
- “调试”>“启动但不调试”：运行可执行文件。

选择教程

可以从下列任一教程入手：

C# 中的数字

在 C# 中的数字教程中，将了解计算机如何存储数字，以及如何对不同类型的数字执行计算。 读者将学习四舍五入的基础知识，以及如何使用 C# 执行数学运算。

若要更好地学习本教程，需要已完成 Hello world 课程。

分支和循环

在分支和循环教程中，将了解根据变量中存储的值选择不同代码执行路径的基础知识。 读者将学习控制流的基础知识，这是程序决定选择不同操作的基本依据。

若要更好地学习本教程，需要先完成 Hello World 和 C# 中的数字课程。

列表集合

列表集合课程将介绍存储一系列数据的列表集合类型。 读者将学习如何添加和删除项、如何搜索项，以及如何对列表进行排序。 读者将探索各种列表。

若要更好地学习本教程，需要已完成上面列出的课程。

如何使用 C# 中的整数和浮点数

项目 • 2023/05/10

本教程介绍 C# 中的数字类型。 你将编写少量的代码，然后编译并运行这些代码。 本教程包含一系列课程，介绍了 C# 中的数字和数学运算。 这些课程介绍了 C# 语言的基础知识。

> **提示**
>
> 若要在焦点模式下粘贴代码片段，应使用键盘快捷方式（`Ctrl` + `v` 或 `cmd` + `v`）。

先决条件

本教程要求安装一台虚拟机，以用于本地开发。 请参阅设置本地环境，了解 .NET 的安装说明和应用程序开发概述。

如果不想设置本地环境，请参阅本教程的交互式浏览器版本。

探索整数数学运算

创建名为 numbers-quickstart 的目录。 将它设为当前目录并运行以下命令：

.NET CLI

```
dotnet new console -n NumbersInCSharp -o .
```

> **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 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。

在常用编辑器中，打开 Program.cs，并将文件内容替换为以下代码：

C#

```
int a = 18;
int b = 6;
int c = a + b;
Console.WriteLine(c);
```

通过在命令窗口中键入 `dotnet run` 运行此代码。

上面是基本的整数数学运算之一。 `int` 类型表示整数（零、正整数或负整数）。 使用 `+` 符号执行加法运算。 其他常见的整数数学运算包括：

- `-`：减法运算
- `*`：乘法运算
- `/`：除法运算

首先，探索这些不同类型的运算。 在写入 `c` 值的行之后添加以下行：

C#

```
// subtraction
c = a - b;
Console.WriteLine(c);

// multiplication
c = a * b;
Console.WriteLine(c);

// division
c = a / b;
Console.WriteLine(c);
```

通过在命令窗口中键入 `dotnet run` 运行此代码。

如果愿意，还可以通过在同一行中编写多个数学运算来进行试验。 例如，请尝试 `c = a + b - 12 * 17;`。 允许混合使用变量和常数。

提示

在探索 C#（或任何编程语言）的过程中，可能会在编写代码时犯错。 编译器 会发现并报告这些错误。 如果输出中包含错误消息，请仔细比对示例代码和你的窗口中

的代码，看看哪些需要纠正。 这样做有助于了解 C# 代码结构。

你已完成第一步。 开始进入下一部分前，先将当前代码移到单独的方法中。 方法是组合在一起并赋予名称的一系列语句。 通过在方法名称后面编写 `()`，可以调用一个方法。 将代码组织为方法，可以更轻松地开始使用新示例。 完成后，代码应如下所示：

C#

```
WorkWithIntegers();

void WorkWithIntegers()
{
    int a = 18;
    int b = 6;
    int c = a + b;
    Console.WriteLine(c);


    // subtraction
    c = a - b;
    Console.WriteLine(c);

    // multiplication
    c = a * b;
    Console.WriteLine(c);

    // division
    c = a / b;
    Console.WriteLine(c);
}
```

行 `WorkWithIntegers();` 调用方法。 下面的代码声明方法并对其进行定义。

探索运算顺序

注释掉对 `WorkingWithIntegers()` 的调用。 在本节中，它会在你工作时使输出变得不那么杂乱：

C#

```
//WorkWithIntegers();
```

`//` 在 C# 中启动 注释。 注释是你想要保留在源代码中但不能作为代码执行的任何文本。 编译器不会从注释中生成任何可执行代码。 因为 `WorkWithIntegers()` 是一种方法，所以只需注释掉一行。

C# 语言使用与数学运算规则一致的规则，定义不同数学运算的优先级。 乘法和除法的优先级高于加法和减法。 通过在调用 `WorkWithIntegers()` 后添加以下代码并执行 `dotnet run` 来探索：

C#

```
int a = 5;
int b = 4;
int c = 2;
int d = a + b * c;
Console.WriteLine(d);
```

输出结果表明，乘法先于加法执行。

可以在要优先执行的一个或多个运算前后添加括号，从而强制改变运算顺序。 添加以下行并再次运行：

C#

```
d = (a + b) * c;
Console.WriteLine(d);
```

通过组合多个不同的运算，进一步探索运算顺序。 添加以下几行内容。 重试 `dotnet run`。

C#

```
d = (a + b) - 6 * c + (12 * 4) / 3 + 12;
Console.WriteLine(d);
```

可能已注意到，整数有一个非常有趣的行为。 整数除法始终生成整数结果，即使预期结果有小数或分数部分也是如此。

如果还没有见过这种行为，请尝试运行以下代码：

C#

```
int e = 7;
int f = 4;
int g = 3;
int h = (e + f) / g;
Console.WriteLine(h);
```

再次键入 `dotnet run`，看看结果如何。

继续之前，让我们获取你在本节编写的所有代码并放在新的方法中。 调用新方法 `OrderPrecedence`。 代码应如下所示：

C#

```
// WorkWithIntegers();
OrderPrecedence();

void WorkWithIntegers()
{
    int a = 18;
    int b = 6;
    int c = a + b;
    Console.WriteLine(c);


    // subtraction
    c = a - b;
    Console.WriteLine(c);

    // multiplication
    c = a * b;
    Console.WriteLine(c);

    // division
    c = a / b;
    Console.WriteLine(c);
}

void OrderPrecedence()
{
    int a = 5;
    int b = 4;
    int c = 2;
    int d = a + b * c;
    Console.WriteLine(d);

    d = (a + b) * c;
    Console.WriteLine(d);

    d = (a + b) - 6 * c + (12 * 4) / 3 + 12;
    Console.WriteLine(d);

    int e = 7;
    int f = 4;
    int g = 3;
    int h = (e + f) / g;
    Console.WriteLine(h);
}
```

探索整数运算精度和限值

在上一个示例中，整数除法截断了结果。 可以使用取模运算符（即 字符）计算余数。 对 `OrderPrecedence()` 进行方法调用之后，尝试添加以下代码：

C#

```
int a = 7;
int b = 4;
int c = 3;
int d = (a + b) / c;
int e = (a + b) % c;
Console.WriteLine($"quotient: {d}");
Console.WriteLine($"remainder: {e}");
```

C# 整数类型不同于数学上的整数的另一点是，`int` 类型有最小限值和最大限值。 添加以下代码以查看这些限制：

C#

```
int max = int.MaxValue;
int min = int.MinValue;
Console.WriteLine($"The range of integers is {min} to {max}");
```

如果运算生成的值超过这些限值，则会出现下溢 或溢出 的情况。 答案似乎是从一个限值覆盖到另一个限值的范围。 添加以下两行以查看示例：

C#

```
int what = max + 3;
Console.WriteLine($"An example of overflow: {what}");
```

可以看到，答案非常接近最小（负）整数。 与 `min + 2` 相同。 加法运算会让整数溢出允许的值。 答案是一个非常大的负数，因为溢出从最大整数值覆盖回最小整数值。

如果 `int` 类型无法满足需求，还会用到限值和精度不同的其他数字类型。 接下来，让我们来研究一下其他类型。 在开始下一节内容之前，将你在本节编写的代码移到一个单独的方法中。 将其命名为 `TestLimits`。

使用双精度类型

`double` 数字类型表示双精度浮点数。 这些词可能是第一次听说。 浮点 数可用于表示数量级可能非常大或非常小的非整数。 双精度 是一个相对术语，描述用于存储值的二进制数位数。 双精度 数字的二进制数位数是单精度 的两倍。 在新式计算机上，使用双精度数字比使用单精度数字更为常见。 单精度数字是使用 关键字声明的。 接下来，将探索双精度类型。 添加以下代码并查看结果：

```C#
double a = 5;
double b = 4;
double c = 2;
double d = (a + b) / c;
Console.WriteLine(d);
```

可以看到，答案商包含小数部分。 试试对双精度类型使用更复杂一点的表达式：

```C#
double e = 19;
double f = 23;
double g = 8;
double h = (e + f) / g;
Console.WriteLine(h);
```

双精度值的范围远大于整数值。 在当前已编写的代码下方试运行以下代码：

```C#
double max = double.MaxValue;
double min = double.MinValue;
Console.WriteLine($"The range of double is {min} to {max}");
```

打印的这些值用科学记数法表示。 `E` 左侧为有效数字。 右侧为是 10 的 n 次幂。 与数学上的十进制数字一样，C# 中的双精度值可能会有四舍五入误差。 试运行以下代码：

```C#
double third = 1.0 / 3.0;
Console.WriteLine(third);
```

你知道 `0.3` 重复有限次并不完全与 `1/3` 相同。

挑战

尝试使用 `double` 类型执行其他的大小数、乘法和除法运算。 尝试执行更复杂的运算。 花了一些时间应对挑战之后，获取已编写的代码并放在一个新方法中。 将新方法命名为 `WorkWithDoubles`。

使用十进制类型

大家已学习了 C# 中的基本数字类型，即整数和双精度。 还需要学习另一种类型，即 `decimal` 类型。 `decimal` 类型的范围较小，但精度高于 `double`。 让我们来实际操作一下：

C#

```
decimal min = decimal.MinValue;
decimal max = decimal.MaxValue;
Console.WriteLine($"The range of the decimal type is {min} to {max}");
```

可以看到，范围小于 `double` 类型。 通过试运行以下代码，可以看到十进制类型的精度更高：

C#

```
double a = 1.0;
double b = 3.0;
Console.WriteLine(a / b);

decimal c = 1.0M;
decimal d = 3.0M;
Console.WriteLine(c / d);
```

数字中的 `M` 后缀指明了常数应如何使用 `decimal` 类型。 否则，编译器假定为 `double` 类型。

> **备注**
>
> 字母 `M` 被选为 `double` 与 `decimal` 关键字之间最明显不同的字母。

可以看到，使用十进制类型执行数学运算时，十进制小数点右侧的数字更多。

挑战

至此，大家已了解不同的数字类型。请编写代码来计算圆面积（其中，半径为 2.50 厘米）。 请注意，圆面积是用半径的平方乘以 PI 进行计算。 小提示：.NET 包含 PI 常数 Math.PI，可用于相应的值计算。 与 `System.Math` 命名空间中声明的所有常量一样，Math.PI 也是 `double` 值。 因此，应使用 `double`（而不是 `decimal`）值来完成这项挑战。

你应获得 19 和 20 之间的答案。 要查看你的答案，可以查看 GitHub 上的完成示例代码。

如果需要，可以试用一些其他公式。

已完成“C# 中的数字”快速入门教程。 可以在自己的开发环境中继续学习分支和循环快速入门教程。

若要详细了解 C# 中的数字，可以参阅下面的文章：

- 整型数值类型
- 浮点型数值类型
- 内置数值转换

C# `if` 语句和循环 - 条件逻辑教程

项目 • 2023/04/08

本教程介绍了如何编写 C# 代码，从而检查变量，并根据这些变量更改执行路径。 读者可以编写 C# 代码并查看代码编译和运行结果。 本教程包含一系列课程，介绍了 C# 中的分支和循环构造。 这些课程介绍了 C# 语言的基础知识。

> **提示**
>
> 若要在焦点模式下粘贴代码片段，应使用键盘快捷方式（`Ctrl` + `v` 或 `cmd` + `v`）。

先决条件

本教程要求安装一台虚拟机，以用于本地开发。 请参阅设置本地环境，了解 .NET 的安装说明和应用程序开发概述。

如果希望在不设置本地环境的情况下运行代码，请参阅本教程的交互式浏览器版本。

使用 `if` 语句做出决定

创建名为 branches-tutorial 的目录。 将其设为当前目录并运行以下命令：

.NET CLI

```
dotnet new console -n BranchesAndLoops -o .
```

> **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 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。

此命令在当前目录中新建一个 .NET 控制台应用程序。 在常用编辑器中，打开 Program.cs，并将内容替换为以下代码：

C#

```
int a = 5;
int b = 6;
if (a + b > 10)
    Console.WriteLine("The answer is greater than 10.");
```

通过在控制台窗口键入 `dotnet run` 试运行此代码。 你应会看到控制台中输出的“答案大于 10”消息。 修改 `b` 的声明，让总和小于 10：

C#

```
int b = 3;
```

再次键入 `dotnet run`。 由于答案小于 10，因此什么也没有打印出来。 要测试的条件为 false。 没有任何可供执行的代码，因为仅为 `if` 语句编写了一个可能分支，即 true 分支。

提示

在探索 C#（或任何编程语言）的过程中，可能会在编写代码时犯错。 编译器会发现并报告这些错误。 仔细查看错误输出和生成错误的代码。 编译器错误通常可帮助你找出问题。

第一个示例展示了 `if` 和布尔类型的用途。 布尔变量可以包含下列两个值之一：`true` 或 `false`。 C# 为布尔变量定义了特殊类型 `bool`。 `if` 语句检查 `bool` 的值。 如果值为 `true`，执行 `if` 后面的语句。 否则，跳过这些语句。 这种检查条件并根据条件执行语句的过程非常强大。

让 if 和 else 完美配合

若要执行 true 和 false 分支中的不同代码，请创建在条件为 false 时执行的 `else` 分支。 尝试使用 `else` 分支。 添加以下代码中的最后两行（你应该已经有前四行）：

C#

```
int a = 5;
int b = 3;
if (a + b > 10)
    Console.WriteLine("The answer is greater than 10");
else
    Console.WriteLine("The answer is not greater than 10");
```

只有当条件的测试结果为 `false` 时，才执行 `else` 关键字后面的语句。 将 `if`、`else` 与布尔条件相结合，可以满足处理 `true` 和 `false` 所需的一切要求。

> **ⓘ 重要**
>
> `if` 和 `else` 语句下的缩进是为了方便读者阅读。 C# 语言忽略缩进或空格。 `if` 或 `else` 关键字后面的语句根据条件决定是否执行。 本教程中的所有示例都遵循了常见做法，根据语句的控制流缩进代码行。

由于缩进会被忽略，因此需要使用 `{` 和 `}` 指明何时要在根据条件决定是否执行的代码块中添加多个语句。 C# 程序员通常会对所有 `if` 和 `else` 子句使用这些大括号。 以下示例与你创建的示例相同。 修改上面的代码以匹配下面的代码：

C#

```
int a = 5;
int b = 3;
if (a + b > 10)
{
    Console.WriteLine("The answer is greater than 10");
}
else
{
    Console.WriteLine("The answer is not greater than 10");
}
```

> **提示**
>
> 在本教程的其余部分中，代码示例全都遵循公认做法，添加了大括号。

可以测试更复杂的条件。 在当前已编写的代码之后添加以下代码：

C#

```
int c = 4;
if ((a + b + c > 10) && (a == b))
{
```

```
    Console.WriteLine("The answer is greater than 10");
    Console.WriteLine("And the first number is equal to the second");
}
else
{
    Console.WriteLine("The answer is not greater than 10");
    Console.WriteLine("Or the first number is not equal to the second");
}
```

`==` 符号执行相等测试。 使用 `==` 将相等测试与赋值测试区分开来，如在 `a = 5` 中所见。

`&&` 表示“且”。 也就是说，两个条件必须都为 true，才能执行 true 分支中的语句。 这些示例还表明，可以在每个条件分支中添加多个语句，前提是将它们用 `{` 和 `}` 括住。 还可以使用 `||` 表示“或”。 在当前已编写的代码之后添加以下代码：

C#

```
if ((a + b + c > 10) || (a == b))
{
    Console.WriteLine("The answer is greater than 10");
    Console.WriteLine("Or the first number is equal to the second");
}
else
{
    Console.WriteLine("The answer is not greater than 10");
    Console.WriteLine("And the first number is not equal to the second");
}
```

修改 `a`、`b` 和 `c` 的值，并在 `&&` 和 `||` 之间切换浏览。 你将进一步了解 `&&` 和 `||` 运算符的工作原理。

你已完成第一步。 开始进入下一部分前，先将当前代码移到单独的方法中。 这样一来，可以更轻松地开始处理新示例。 将现有代码放入名为 `ExploreIf()` 的方法中。 从程序顶部调用它。 完成这些更改后，代码看起来应类似下面这样：

C#

```
ExploreIf();

void ExploreIf()
{
    int a = 5;
    int b = 3;
    if (a + b > 10)
    {
        Console.WriteLine("The answer is greater than 10");
    }
    else
    {
```

```
        Console.WriteLine("The answer is not greater than 10");
    }

    int c = 4;
    if ((a + b + c > 10) && (a > b))
    {
        Console.WriteLine("The answer is greater than 10");
        Console.WriteLine("And the first number is greater than the
second");
    }
    else
    {
        Console.WriteLine("The answer is not greater than 10");
        Console.WriteLine("Or the first number is not greater than the
second");
    }

    if ((a + b + c > 10) || (a > b))
    {
        Console.WriteLine("The answer is greater than 10");
        Console.WriteLine("Or the first number is greater than the second");
    }
    else
    {
        Console.WriteLine("The answer is not greater than 10");
        Console.WriteLine("And the first number is not greater than the
second");
    }
}
```

注释掉对 `ExploreIf()` 的调用。 在本节中，它会在你工作时使输出变得不那么杂乱：

C#

```
//ExploreIf();
```

`//` 在 C# 中启动 注释。 注释是你想要保留在源代码中但不能作为代码执行的任何文本。 编译器不会从注释中生成任何可执行代码。

使用循环重复执行运算

在本节使用循环以重复执行语句。 在调用 `ExploreIf` 后，添加以下代码：

C#

```
int counter = 0;
while (counter < 10)
{
    Console.WriteLine($"Hello World! The counter is {counter}");
```

```
    counter++;
}
```

`while` 语句会检查条件，并执行 `while` 后面的语句或语句块。 除非条件为 false，否则它会重复检查条件并执行这些语句。

此示例新引入了另外一个运算符。 `counter` 变量后面的 `++` 是增量运算符。 它将 `counter` 值加 1，并将计算后的值存储在 `counter` 变量中。

> **ⓘ 重要**
>
> 请确保 `while` 循环条件在你执行代码时更改为 false。 否则，创建的就是无限循环，即程序永不结束。 本示例中没有演示上述情况，因为你必须使用 CTRL-C 或其他方法强制退出程序。

`while` 循环先测试条件，然后再执行 `while` 后面的代码。 `do` ... `while` 循环先执行代码，然后再检查条件。 下面的代码对 do while 循环进行了演示：

C#

```
int counter = 0;
do
{
    Console.WriteLine($"Hello World! The counter is {counter}");
    counter++;
} while (counter < 10);
```

此 `do` 循环和前面的 `while` 循环生成的输出结果相同。

使用 for 循环

For 循环通常用在 C# 中。 试运行以下代码：

C#

```
for (int index = 0; index < 10; index++)
{
    Console.WriteLine($"Hello World! The index is {index}");
}
```

上述代码的工作原理与已用过的 `while` 循环和 `do` 循环相同。 `for` 语句包含三个控制具体工作方式的部分。

第一部分是 for 初始值设定项：`int index = 0;` 声明 `index` 是循环变量，并将它的初始值设置为 `0`。

中间部分是 for 条件：`index < 10` 声明只要计数器值小于 10，此 `for` 循环就会继续执行。

最后一部分是 for 迭代器：`index++` 指定在执行 `for` 语句后面的代码块后，如何修改循环变量。 在此示例中，它指定 `index` 应在代码块每次执行时递增 1。

请自行试验。 尝试以下每种变化：

- 将初始值设定项更改为其他初始值。
- 将结束条件设定项更改为其他值。

完成后，继续利用所学知识，试着自己编写一些代码。

本教程中未介绍的另一个循环语句：`foreach` 语句。 `foreach` 语句为项序列中的每一项重复其语句。 它最常用于集合，因此将在下一教程中介绍。

已创建嵌套循环

`while`、`do` 或 `for` 循环可以嵌套在另一个循环内，以使用外部循环中的各项与内部循环中的各项组合来创建矩阵。 我们来生成一组字母数字对以表示行和列。

一个 `for` 循环可以生成行：

C#

```
for (int row = 1; row < 11; row++)
{
    Console.WriteLine($"The row is {row}");
}
```

另一个循环可以生成列：

C#

```
for (char column = 'a'; column < 'k'; column++)
{
    Console.WriteLine($"The column is {column}");
}
```

可以将一个循环嵌套在另一个循环中以形成各个对：

C#

```
for (int row = 1; row < 11; row++)
{
    for (char column = 'a'; column < 'k'; column++)
    {
        Console.WriteLine($"The cell is ({row}, {column})");
    }
}
```

可以看到，对于内部循环的每次完整运行，外部循环会递增一次。 反转行和列嵌套，自行查看变化。 完成后，将此部分中的代码放入名为 `ExploreLoops()` 的方法中。

结合使用分支和循环

支持，大家已了解 C# 语言中的 `if` 语句和循环构造。看看能否编写 C# 代码，计算 1 到 20 中所有可被 3 整除的整数的总和。 下面提供了一些提示：

- `%` 运算符可用于获取除法运算的余数。
- `if` 语句中的条件可用于判断是否应将数字计入总和。
- `for` 循环有助于对 1 到 20 中的所有数字重复执行一系列步骤。

亲自试一试吧。 然后，看看自己是怎么做到的。 你应获取的答案为 63。 通过在 GitHub 上查看已完成的代码，你可以看到一个可能的答案。

已完成“分支和循环”教程。

可以在自己的开发环境中继续学习数组和集合教程。

若要详细了解这些概念，请参阅下列文章：

- 选择语句
- 迭代语句

了解如何在 C# 中使用 List<T> 管理数据集合

项目 • 2023/05/10

本介绍性教程介绍了 C# 语言和 List<T> 类的基础知识。

先决条件

本教程要求安装一台虚拟机，以用于本地开发。 请参阅设置本地环境，了解 .NET 的安装说明和应用程序开发概述。

如果希望在不设置本地环境的情况下运行代码，请参阅本教程的交互式浏览器版本。

基本列表示例

创建名为 list-tutorial 的目录。 将新建的目录设为当前目录，并运行 `dotnet new console`。

> **ⓘ 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 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。

在常用编辑器中，打开 Program.cs，并将现有代码替换为以下代码：

C#

```
var names = new List<string> { "<name>", "Ana", "Felipe" };
foreach (var name in names)
{
    Console.WriteLine($"Hello {name.ToUpper()}!");
}
```

将 `<name>` 替换为自己的名称。 保存 Program.cs。 在控制台窗口中键入 `dotnet run`，试运行看看。

你已创建了一个字符串列表，在该列表中添加了三个名称，并打印了所有大写的名称。 循环读取整个列表需要用到在前面的教程中学到的概念。

用于显示名称的代码使用字符串内插功能。 如果 `string` 前面有 `$`符号，可以在字符串声明中嵌入 C# 代码。 实际字符串使用自己生成的值替换该 C# 代码。 在此示例中，`{name.ToUpper()}` 被替换为各个转换为大写字母的名称，因为调用了 ToUpper 方法。

接下来将进一步探索。

修改列表内容

创建的集合使用 List<T> 类型。 此类型存储一系列元素。 元素类型是在尖括号内指定。

List<T> 类型的一个重要方面是，既可以扩大，也可以收缩，方便用户添加或删除元素。 在程序末尾添加以下代码：

C#

```
Console.WriteLine();
names.Add("Maria");
names.Add("Bill");
names.Remove("Ana");
foreach (var name in names)
{
    Console.WriteLine($"Hello {name.ToUpper()}!");
}
```

又向列表的末尾添加了两个名称。 同时，也删除了一个名称。 保存此文件，并键入 `dotnet run`，试运行看看。

借助 List<T>，还可以按索引引用各项。 索引位于列表名称后面的 `[` 和 `]` 令牌之间。 C# 对第一个索引使用 0。 将以下代码添加到刚才添加的代码的正下方，并试运行看看：

C#

```
Console.WriteLine($"My name is {names[0]}");
Console.WriteLine($"I've added {names[2]} and {names[3]} to the list");
```

不得访问超出列表末尾的索引。 请注意，索引是从 0 开始编制，因此最大有效索引是用列表项数减 1 计算得出。 可以使用 Count 属性确定列表长度。 在程序的末尾添加以下代

码：

```
C#
```

```
Console.WriteLine($"The list has {names.Count} people in it");
```

保存此文件，并再次键入 `dotnet run` 看看结果如何。

搜索列表并进行排序

我们的示例使用的列表较小，但大家的应用程序创建的列表通常可能会包含更多元素，有时可能会包含数以千计的元素。 若要在更大的集合中查找元素，需要在列表中搜索不同的项。 IndexOf 方法可搜索项，并返回此项的索引。 如果项不在列表中，`IndexOf` 将返回 `-1`。 在程序底部添加以下代码：

```
C#
```

```
var index = names.IndexOf("Felipe");
if (index == -1)
{
    Console.WriteLine($"When an item is not found, IndexOf returns
{index}");
}
else
{
    Console.WriteLine($"The name {names[index]} is at index {index}");
}

index = names.IndexOf("Not Found");
if (index == -1)
{
    Console.WriteLine($"When an item is not found, IndexOf returns
{index}");
}
else
{
    Console.WriteLine($"The name {names[index]} is at index {index}");

}
```

还可以对列表中的项进行排序。 Sort 方法按正常顺序（如果是字符串则按字母顺序）对列表中的所有项进行排序。 在程序底部添加以下代码：

```
C#
```

```
names.Sort();
foreach (var name in names)
```

```
{
    Console.WriteLine($"Hello {name.ToUpper()}!");
}
```

保存此文件，并键入 `dotnet run`，试运行此最新版程序。

开始进入下一部分前，先将当前代码移到单独的方法中。 这样一来，可以更轻松地开始处理新示例。 将你编写的所有代码放在一个名为 `WorkWithStrings()` 的新方法中。 在程序的顶部调用该方法。 完成后，代码应如下所示：

C#

```
WorkWithStrings();

void WorkWithStrings()
{
    var names = new List<string> { "<name>", "Ana", "Felipe" };
    foreach (var name in names)
    {
        Console.WriteLine($"Hello {name.ToUpper()}!");
    }

    Console.WriteLine();
    names.Add("Maria");
    names.Add("Bill");
    names.Remove("Ana");
    foreach (var name in names)
    {
        Console.WriteLine($"Hello {name.ToUpper()}!");
    }

    Console.WriteLine($"My name is {names[0]}");
    Console.WriteLine($"I've added {names[2]} and {names[3]} to the list");

    Console.WriteLine($"The list has {names.Count} people in it");

    var index = names.IndexOf("Felipe");
    if (index == -1)
    {
        Console.WriteLine($"When an item is not found, IndexOf returns
{index}");
    }
    else
    {
        Console.WriteLine($"The name {names[index]} is at index {index}");
    }

    index = names.IndexOf("Not Found");
    if (index == -1)
    {
        Console.WriteLine($"When an item is not found, IndexOf returns
{index}");
```

```
    }
    else
    {
        Console.WriteLine($"The name {names[index]} is at index {index}");

    }

    names.Sort();
    foreach (var name in names)
    {
        Console.WriteLine($"Hello {name.ToUpper()}!");
    }
}
```

其他类型的列表

到目前为止，大家一直在列表中使用 `string` 类型。 接下来，将让 List<T> 使用其他类型。 那就生成一组数字吧。

在调用 `WorkWithStrings()` 后，在程序中添加以下内容：

```
C#

var fibonacciNumbers = new List<int> {1, 1};
```

这会创建一个整数列表，并将头两个整数设置为值 1。 这些是斐波那契数列（一系列数字）的头两个值。 斐波那契数列中的每个数字都是前两个数字之和。 添加以下代码：

```
C#

var previous = fibonacciNumbers[fibonacciNumbers.Count - 1];
var previous2 = fibonacciNumbers[fibonacciNumbers.Count - 2];

fibonacciNumbers.Add(previous + previous2);

foreach (var item in fibonacciNumbers)
{
    Console.WriteLine(item);
}
```

保存此文件，并键入 `dotnet run` 看看结果如何。

> **提示**
>
> 为了能够集中精力探究此部分，可以注释掉调用 `WorkWithStrings();` 的代码。 只需在此调用前添加两个 `/` 字符即可，如 `// WorkWithStrings();`。

挑战

看看能不能将本课程中的一些概念与前面的课程融会贯通。 使用斐波那契数列，扩展当前生成的程序。 试着编写代码，生成此序列中的前 20 个数字。 （作为提示，第 20 个斐波纳契数是 6765。）

完成挑战

可以查看 GitHub 上的完成示例代码，了解示例解决方案。

在循环的每次迭代中，取此列表中的最后两个整数进行求和，并将计算出的总和值添加到列表中。 循环会一直重复运行到列表中有 20 个项为止。

恭喜！已完成“列表集合”教程。 你可以继续在自己的开发环境中学习更多教程。

若要详细了解如何使用 `List` 类型，可参阅有关集合的 .NET 基础知识文章。 还可以了解其他许多集合类型。

C# 程序的通用结构

项目 • 2023/04/07

C# 程序由一个或多个文件组成。 每个文件均包含零个或多个命名空间。 一个命名空间包含类、结构、接口、枚举、委托等类型或其他命名空间。 以下示例是包含所有这些元素的 C# 程序主干。

C#

```
// A skeleton of a C# program
using System;

// Your program starts here:
Console.WriteLine("Hello world!");

namespace YourNamespace
{
    class YourClass
    {
    }

    struct YourStruct
    {
    }

    interface IYourInterface
    {
    }

    delegate int YourDelegate();

    enum YourEnum
    {
    }

    namespace YourNestedNamespace
    {
        struct YourStruct
        {
        }
    }
}
```

前面的示例使用顶级语句作为程序的入口点。 C# 9 中添加了此功能。 在 C# 9 之前，入口点是名为 `Main` 的静态方法，如以下示例所示：

C#

```
// A skeleton of a C# program
using System;
namespace YourNamespace
{
    class YourClass
    {
    }

    struct YourStruct
    {
    }

    interface IYourInterface
    {
    }

    delegate int YourDelegate();

    enum YourEnum
    {
    }

    namespace YourNestedNamespace
    {
        struct YourStruct
        {
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            //Your program starts here...
            Console.WriteLine("Hello world!");
        }
    }
}
```

相关章节

在基础指南的类型部分中了解这些程序元素：

- 类
- 结构
- 命名空间
- 接口
- 枚举
- 委托

C# 语言规范

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

Main() 和命令行参数

项目 • 2024/06/24

`Main` 方法是 C# 应用程序的入口点。 `Main` 方法是应用程序启动后调用的第一个方法。

C# 程序中只能有一个入口点。 如果多个类包含 `Main` 方法，必须使用 StartupObject 编译器选项来编译程序，以指定将哪个 `Main` 方法用作入口点。 有关详细信息，请参阅 StartupObject（C# 编译器选项）。

C#

```
class TestClass
{
    static void Main(string[] args)
    {
        // Display the number of command line arguments.
        Console.WriteLine(args.Length);
    }
}
```

还可以在一个文件中使用顶级语句作为应用程序的入口点。 与 `Main` 方法一样，顶级语句还可以返回值和访问命令行参数。 有关详细信息，请参阅顶级语句。

C#

```
using System.Text;

StringBuilder builder = new();
builder.AppendLine("The following arguments are passed:");

// Display the command line arguments using the args variable.
foreach (var arg in args)
{
    builder.AppendLine($"Argument={arg}");
}

Console.WriteLine(builder.ToString());

// Return a success code.
return 0;
```

概述

- `Main` 方法是可执行程序的入口点，也是程序控制开始和结束的位置。
- `Main` 必须在类或结构中进行声明。 封闭 `class` 可以是 `static`。

- `Main` 必须为 static。
- `Main` 可以具有任何访问修饰符（`file` 除外）。
- `Main` 的返回类型可以是 `void`、`int`、`Task` 或 `Task<int>`。
- 当且仅当 `Main` 返回 `Task` 或 `Task<int>` 时，`Main` 的声明可包括 async 修饰符。 这明确排除了 `async void Main` 方法。
- 使用或不使用包含命令行自变量的 `string[]` 参数声明 `Main` 方法都行。 使用 Visual Studio 创建 Windows 应用程序时，可以手动添加此形参，也可以使用 GetCommandLineArgs() 方法来获取命令行实参。 参数被读取为从零开始编制索引的命令行自变量。 与 C 和 C++ 不同，程序的名称不被视为 `args` 数组中的第一个命令行实参，但它是 GetCommandLineArgs() 方法中的第一个元素。

以下列表显示了最常见的 `Main` 声明：

C#

```
static void Main() { }
static int Main() { }
static void Main(string[] args) { }
static int Main(string[] args) { }
static async Task Main() { }
static async Task<int> Main() { }
static async Task Main(string[] args) { }
static async Task<int> Main(string[] args) { }
```

前面的示例未指定访问修饰符，因此默认为隐式 `private`。 这是典型的，但可以指定任何显式访问修饰符。

> **提示**
>
> 添加 `async`、`Task` 和 `Task<int>` 返回类型可简化控制台应用程序需要启动时的程序代码，以及 `Main` 中的 `await` 异步操作。

Main() 返回值

可以通过以下方式之一定义方法，以从 `Main` 方法返回 `int`：

展开表

`Main` 声明	`Main` 方法代码
`static int Main()`	不使用 `args` 或 `await`

`Main` 声明	`Main` 方法代码
`static int Main(string[] args)`	使用 `args`，不使用 `await`
`static async Task<int> Main()`	不使用 `args`，使用 `await`
`static async Task<int> Main(string[] args)`	使用 `args` 和 `await`

如果不使用 `Main` 的返回值，则返回 `void` 或 `Task` 可使代码变得略微简单。

展开表

`Main` 声明	`Main` 方法代码
`static void Main()`	不使用 `args` 或 `await`
`static void Main(string[] args)`	使用 `args`，不使用 `await`
`static async Task Main()`	不使用 `args`，使用 `await`
`static async Task Main(string[] args)`	使用 `args` 和 `await`

但是，返回 `int` 或 `Task<int>` 可使程序将状态信息传递给调用可执行文件的其他程序或脚本。

下面的示例演示了如何访问进程的退出代码。

此示例使用 .NET Core 命令行工具。 如果不熟悉 .NET Core 命令行工具，可通过本入门文章进行了解。

通过运行 `dotnet new console` 创建新的应用程序。 修改 Program.cs 中的 `Main` 方法，如下所示：

C#

```
// Save this program as MainReturnValTest.cs.
class MainReturnValTest
{
    static int Main()
    {
        //...
        return 0;
    }
}
```

在 Windows 中执行程序时，从 `Main` 函数返回的任何值都存储在环境变量中。 可使用批处理文件中的 `ERRORLEVEL` 或 PowerShell 中的 `$LastExitCode` 来检索此环境变量。

可使用 dotnet CLI `dotnet build` 命令构建应用程序。

接下来，创建一个 PowerShell 脚本来运行应用程序并显示结果。 将以下代码粘贴到文本文件中，并在包含该项目的文件夹中将其另存为 `test.ps1`。 可通过在 PowerShell 提示符下键入 `test.ps1` 来运行 PowerShell 脚本。

因为代码返回零，所以批处理文件将报告成功。 但是，如果将 MainReturnValTest.cs 更改为返回非零值，然后重新编译程序，则 PowerShell 脚本的后续执行将报告为失败。

PowerShell

```
dotnet run
if ($LastExitCode -eq 0) {
    Write-Host "Execution succeeded"
} else
{
    Write-Host "Execution Failed"
}
Write-Host "Return value = " $LastExitCode
```

输出

```
Execution succeeded
Return value = 0
```

Async Main 返回值

声明 `Main` 的 `async` 返回值时，编译器会生成样本代码，用于调用 `Main` 中的异步方法。 如果未指定 `async` 关键字，则需要自行编写该代码，如以下示例所示。 示例中的代码可确保程序一直运行，直到异步操作完成：

C#

```
class AsyncMainReturnValTest
{
    public static int Main()
    {
        return AsyncConsoleWork().GetAwaiter().GetResult();
    }

    private static async Task<int> AsyncConsoleWork()
    {
        // Main body here
        return 0;
    }
}
```

该样本代码可替换为：

```C#
class Program
{
    static async Task<int> Main(string[] args)
    {
        return await AsyncConsoleWork();
    }

    private static async Task<int> AsyncConsoleWork()
    {
        // main body here
        return 0;
    }
}
```

将 `Main` 声明为 `async` 的优点是，编译器始终生成正确的代码。

当应用程序入口点返回 `Task` 或 `Task<int>` 时，编译器生成一个新的入口点，该入口点调用应用程序代码中声明的入口点方法。 假设此入口点名为 `$GeneratedMain`，编译器将为这些入口点生成以下代码：

- `static Task Main()` 导致编译器发出 `private static void $GeneratedMain() => Main().GetAwaiter().GetResult();` 的等效项
- `static Task Main(string[])` 导致编译器发出 `private static void $GeneratedMain(string[] args) => Main(args).GetAwaiter().GetResult();` 的等效项
- `static Task<int> Main()` 导致编译器发出 `private static int $GeneratedMain() => Main().GetAwaiter().GetResult();` 的等效项
- `static Task<int> Main(string[])` 导致编译器发出 `private static int $GeneratedMain(string[] args) => Main(args).GetAwaiter().GetResult();` 的等效项

> **备注**
>
> 如果示例在 `Main` 方法上使用 `async` 修饰符，则编译器将生成相同的代码。

命令行自变量

可以通过以下方式之一定义方法来将自变量发送到 `Main` 方法：

展开表

`Main` 声明	`Main` 方法代码
`static void Main(string[] args)`	无返回值，不使用 `await`
`static int Main(string[] args)`	返回值，不使用 `await`
`static async Task Main(string[] args)`	无返回值，使用 `await`
`static async Task<int> Main(string[] args)`	返回值，使用 `await`

如果不使用参数，可以从方法声明中省略 `args`，使代码更为简单：

展开表

`Main` 声明	`Main` 方法代码
`static void Main()`	无返回值，不使用 `await`
`static int Main()`	返回值，不使用 `await`
`static async Task Main()`	无返回值，使用 `await`
`static async Task<int> Main()`	返回值，使用 `await`

备注

还可使用 Environment.CommandLine 或 Environment.GetCommandLineArgs 从控制台或 Windows 窗体应用程序的任意位置访问命令行参数。 若要在 Windows 窗体应用程序的 `Main` 方法中启用命令行参数，必须手动修改 `Main` 的声明。 Windows 窗体设计器生成的代码创建没有输入参数的 `Main`。

`Main` 方法的参数是一个表示命令行参数的 String 数组。 通常，通过测试 `Length` 属性来确定参数是否存在，例如：

C#

```
if (args.Length == 0)
{
    System.Console.WriteLine("Please enter a numeric argument.");
    return 1;
}
```

提示

`args` 数组不能为 null。 因此，无需进行 null 检查即可放心地访问 `Length` 属性。

还可以使用 Convert 类或 `Parse` 方法将字符串参数转换为数字类型。 例如，以下语句使用 Parse 方法将 `string` 转换为 `long` 数字：

```
C#

long num = Int64.Parse(args[0]);
```

也可以使用 C# 类型 `long`，其别名为 `Int64`：

```
C#

long num = long.Parse(args[0]);
```

还可以使用 `Convert` 类方法 `ToInt64` 来执行同样的操作：

```
C#

long num = Convert.ToInt64(s);
```

有关详细信息，请参阅 Parse 和 Convert。

> **提示**
>
> 分析命令行参数可能比较复杂。 请考虑使用 System.CommandLine 库（目前为 beta 版）来简化该过程。

以下示例演示如何在控制台应用程序中使用命令行参数。 应用程序在运行时获取一个参数，将该参数转换为整数，并计算数字的阶乘。 如果未提供任何参数，则应用程序会发出一条消息，说明程序的正确用法。

若要在命令提示符下编译并运行该应用程序，请按照下列步骤操作：

1. 将以下代码粘贴到任何文本编辑器，然后将该文件保存为名为“Factorial.cs”的文本文件。

   ```
   C#

   public class Functions
   {
       public static long Factorial(int n)
       {
   ```

```
        // Test for invalid input.
        if ((n < 0) || (n > 20))
        {
            return -1;
        }

        // Calculate the factorial iteratively rather than recursively.
        long tempResult = 1;
        for (int i = 1; i <= n; i++)
        {
            tempResult *= i;
        }
        return tempResult;
    }
}

class MainClass
{
    static int Main(string[] args)
    {
        // Test if input arguments were supplied.
        if (args.Length == 0)
        {
            Console.WriteLine("Please enter a numeric argument.");
            Console.WriteLine("Usage: Factorial <num>");
            return 1;
        }

        // Try to convert the input arguments to numbers. This will
throw
        // an exception if the argument is not a number.
        // num = int.Parse(args[0]);
        int num;
        bool test = int.TryParse(args[0], out num);
        if (!test)
        {
            Console.WriteLine("Please enter a numeric argument.");
            Console.WriteLine("Usage: Factorial <num>");
            return 1;
        }

        // Calculate factorial.
        long result = Functions.Factorial(num);

        // Print result.
        if (result == -1)
            Console.WriteLine("Input must be >= 0 and <= 20.");
        else
            Console.WriteLine($"The Factorial of {num} is {result}.");

        return 0;
    }
}
// If 3 is entered on command line, the
// output reads: The factorial of 3 is 6.
```

2. 从“开始”屏幕或“开始”菜单中，打开 Visual Studio“开发人员命令提示”窗口，然后导航到包含你创建的文件的文件夹。

3. 输入以下命令以编译应用程序。

   ```
   dotnet build
   ```

   如果应用程序不存在编译错误，则会创建一个名为“Factorial.exe”的可执行文件。

4. 输入以下命令以计算 3 的阶乘：

   ```
   dotnet run -- 3
   ```

5. 该命令将生成以下输出：`The factorial of 3 is 6.`

> **备注**
>
> 在 Visual Studio 中运行应用程序时，可在“**项目设计器**”->“**调试**”页中指定命令行参数。

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

- System.Environment
- 如何显示命令行参数

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

顶级语句 - 不使用 `Main` 方法的程序

项目 • 2024/03/04

无需在控制台应用程序项目中显式包含 `Main` 方法。 相反，可以使用顶级语句功能最大程度地减少必须编写的代码。

使用顶级语句可直接在文件的根目录中编写可执行代码，而无需在类或方法中包装代码。 这意味着无需使用 `Program` 类和 `Main` 方法即可创建程序。 在这种情况下，编译器将使用入口点方法为应用程序生成 `Program` 类。 生成方法的名称不是 `Main`，而是你的代码无法直接引用的实现详细信息。

下面是一个 Program.cs 文件看，它是 C# 10 中的一个完整 C# 程序：

```
C#

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

借助顶级语句，可以为小实用程序（如 Azure Functions 和 GitHub Actions）编写简单的程序。 它们还使初次接触 C# 的程序员能够更轻松地开始学习和编写代码。

以下各节介绍了可对顶级语句执行和不能执行的操作的规则。

仅能有一个顶级文件

一个应用程序只能有一个入口点。 一个项目只能有一个包含顶级语句的文件。 在项目中的多个文件中放置顶级语句会导致以下编译器错误：

> CS8802：只有一个编译单元可具有顶级语句。

一个项目可具有任意数量的其他源代码文件，这些文件不包含顶级语句。

没有其他入口点

可以显式编写 `Main` 方法，但它不能作为入口点。 编译器将发出以下警告：

> CS7022：程序的入口点是全局代码；忽略“Main()”入口点。

在具有顶级语句的项目中，不能使用 -main 编译器选项来选择入口点，即使该项目具有一个或多个 `Main` 方法。

`using` 指令

如果包含 using 指令，则它们必须首先出现在文件中，如以下示例中所示：

C#

```
using System.Text;

StringBuilder builder = new();
builder.AppendLine("Hello");
builder.AppendLine("World!");

Console.WriteLine(builder.ToString());
```

全局命名空间

顶级语句隐式位于全局命名空间中。

命名空间和类型定义

具有顶级语句的文件还可以包含命名空间和类型定义，但它们必须位于顶级语句之后。例如：

C#

```
MyClass.TestMethod();
MyNamespace.MyClass.MyMethod();

public class MyClass
{
    public static void TestMethod()
    {
        Console.WriteLine("Hello World!");
    }
}

namespace MyNamespace
{
    class MyClass
    {
        public static void MyMethod()
        {
            Console.WriteLine("Hello World from
MyNamespace.MyClass.MyMethod!");
        }
    }
}
```

`args`

顶级语句可以引用 `args` 变量来访问输入的任何命令行参数。 `args` 变量永远不会为 null，但如果未提供任何命令行参数，则其 `Length` 将为零。 例如：

C#

```
if (args.Length > 0)
{
    foreach (var arg in args)
    {
        Console.WriteLine($"Argument={arg}");
    }
}
else
{
    Console.WriteLine("No arguments");
}
```

`await`

可以通过使用 `await` 来调用异步方法。 例如：

C#

```
Console.Write("Hello ");
await Task.Delay(5000);
Console.WriteLine("World!");
```

进程的退出代码

若要在应用程序结束时返回 `int` 值，请像在 `Main` 方法中返回 `int` 那样使用 `return` 语句。 例如：

C#

```
string? s = Console.ReadLine();

int returnValue = int.Parse(s ?? "-1");
return returnValue;
```

隐式入口点方法

编译器会生成一个方法，作为具有顶级语句的项目的程序入口点。 方法的签名取决于顶级语句是包含 `await` 关键字还是 `return` 语句。 下表显示了方法签名的外观，为了方便起见，在表中使用了方法名称 `Main`。

展开表

顶级代码包含	隐式 `Main` 签名
`await` 和 `return`	`static async Task<int> Main(string[] args)`
`await`	`static async Task Main(string[] args)`
`return`	`static int Main(string[] args)`
否 `await` 或 `return`	`static void Main(string[] args)`

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。

功能规范 - 顶级语句

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 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# 类型系统

项目 • 2024/04/11

C# 是一种强类型语言。 每个变量和常量都有一个类型，每个求值的表达式也是如此。 每个方法声明都为每个输入参数和返回值指定名称、类型和种类（值、引用或输出）。 .NET 类库定义了内置数值类型和表示各种构造的复杂类型。 其中包括文件系统、网络连接、对象的集合和数组以及日期。 典型的 C# 程序使用类库中的类型，以及对程序问题域的专属概念进行建模的用户定义类型。

类型中可存储的信息包括以下项：

- 类型变量所需的存储空间。
- 可以表示的最大值和最小值。
- 包含的成员（方法、字段、事件等）。
- 继承自的基类型。
- 它实现的接口。
- 允许执行的运算种类。

编译器使用类型信息来确保在代码中执行的所有操作都是类型安全的。 例如，如果声明 int 类型的变量，那么编译器允许在加法和减法运算中使用此变量。 如果尝试对 bool 类型的变量执行这些相同操作，则编译器将生成错误，如以下示例所示：

C#

```
int a = 5;
int b = a + 2; //OK

bool test = true;

// Error. Operator '+' cannot be applied to operands of type 'int' and
'bool'.
int c = a + test;
```

> **备注**
>
> C 和 C++ 开发人员请注意，在 C# 中，`bool` 不能转换为 `int`。

编译器将类型信息作为元数据嵌入可执行文件中。 公共语言运行时 (CLR) 在运行时使用元数据，以在分配和回收内存时进一步保证类型安全性。

在变量声明中指定类型

当在程序中声明变量或常量时，必须指定其类型或使用 var 关键字让编译器推断类型。 以下示例显示了一些使用内置数值类型和复杂用户定义类型的变量声明：

C#

```
// Declaration only:
float temperature;
string name;
MyClass myClass;

// Declaration with initializers (four examples):
char firstLetter = 'C';
var limit = 3;
int[] source = { 0, 1, 2, 3, 4, 5 };
var query = from item in source
            where item <= limit
            select item;
```

方法声明指定方法参数的类型和返回值。 以下签名显示了需要 `int` 作为输入参数并返回字符串的方法：

C#

```
public string GetName(int ID)
{
    if (ID < names.Length)
        return names[ID];
    else
        return String.Empty;
}
private string[] names = { "Spencer", "Sally", "Doug" };
```

声明变量后，不能使用新类型重新声明该变量，并且不能分配与其声明的类型不兼容的值。 例如，不能声明 `int` 后再向它分配 `true` 的布尔值。 不过，可以将值转换成其他类型。例如，在将值分配给新变量或作为方法自变量传递时。 编译器会自动执行不会导致数据丢失的类型转换。 如果类型转换可能会导致数据丢失，必须在源代码中进行*显式转换*。

有关详细信息，请参阅显式转换和类型转换。

内置类型

C# 提供了一组标准的内置类型。 这些类型表示整数、浮点值、布尔表达式、文本字符、十进制值和其他数据类型。 还有内置的 `string` 和 `object` 类型。 这些类型可供在任何 C# 程序中使用。 有关内置类型的完整列表，请参阅内置类型。

自定义类型

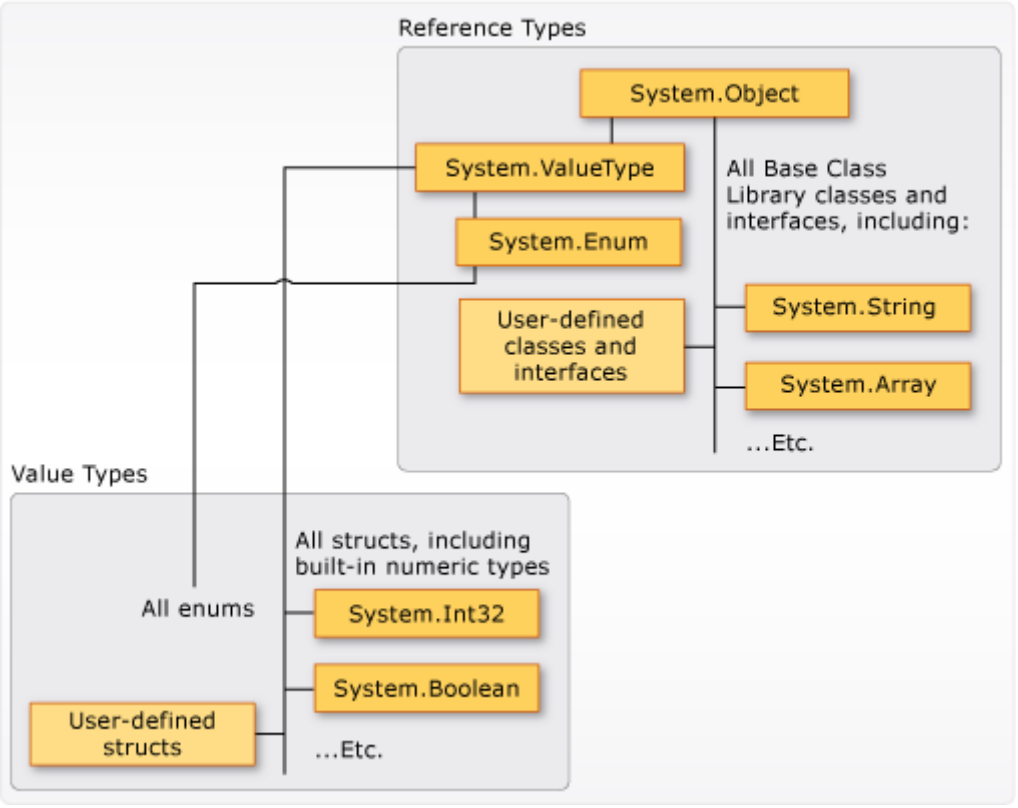
可以使用 struct、class、interface、enum 和 record 构造来创建自己的自定义类型。 .NET 类库本身是一组自定义类型，以供你在自己的应用程序中使用。 默认情况下，类库中最常用的类型在任何 C# 程序中均可用。 其他类型只有在显式添加对定义这些类型的程序集的项目引用时才可用。 编译器引用程序集之后，你可以声明在源代码的此程序集中声明的类型的变量（和常量）。 有关详细信息，请参阅 .NET 类库。

通用类型系统

对于 .NET 中的类型系统，请务必了解以下两个基本要点：

- 它支持继承原则。 类型可以派生自其他类型（称为*基类型*）。 派生类型继承（有一些限制）基类型的方法、属性和其他成员。 基类型可以继而从某种其他类型派生，在这种情况下，派生类型继承其继承层次结构中的两种基类型的成员。 所有类型（包括 System.Int32 (C# keyword: `int`) 等内置数值类型）最终都派生自单个基类型，即 System.Object (C# keyword: object)。 这样的统一类型层次结构称为通用类型系统 (CTS)。 若要详细了解 C# 中的继承，请参阅继承。
- CTS 中的每种类型被定义为值类型或引用类型。 这些类型包括 .NET 类库中的所有自定义类型以及你自己的用户定义类型。 使用 `struct` 关键字定义的类型是值类型；所有内置数值类型都是 `structs`。 使用 `class` 或 `record` 关键字定义的类型是引用类型。 引用类型和值类型遵循不同的编译时规则和运行时行为。

下图展示了 CTS 中值类型和引用类型之间的关系。



> **备注**
>
> 你可能会发现，最常用的类型全都被整理到了 System 命名空间中。 不过，包含类型的命名空间与类型是值类型还是引用类型没有关系。

类和结构是 .NET 通用类型系统的两种基本构造。 C# 9 添加记录，记录是一种类。 每种本质上都是一种数据结构，其中封装了同属一个逻辑单元的一组数据和行为。 数据和行为是类、结构或记录的成员。 这些行为包括方法、属性和事件等，本文稍后将具体列举。

类、结构或记录声明类似于一张蓝图，用于在运行时创建实例或对象。 如果定义名为 `Person` 的类、结构或记录，则 `Person` 是类型的名称。 如果声明和初始化 `Person` 类型的变量 `p`，那么 `p` 就是所谓的 `Person` 对象或实例。 可以创建同一 `Person` 类型的多个实例，每个实例都可以有不同的属性和字段值。

类是引用类型。 创建类型的对象后，向其分配对象的变量仅保留对相应内存的引用。 将对象引用分配给新变量后，新变量会引用原始对象。 通过一个变量所做的更改将反映在另一个变量中，因为它们引用相同的数据。

结构是值类型。 创建结构时，向其分配结构的变量保留结构的实际数据。 将结构分配给新变量时，会复制结构。 因此，新变量和原始变量包含相同数据的副本（共两个）。 对一个副本所做的更改不会影响另一个副本。

记录类型可以是引用类型（`record class`）或值类型（`record struct`）。

一般来说，类用于对更复杂的行为建模。 类通常存储计划在创建类对象后进行修改的数据。 结构最适用于小型数据结构。 结构通常存储不打算在创建结构后修改的数据。 记录类型是具有附加编译器合成成员的数据结构。 记录通常存储不打算在创建对象后修改的数据。

值类型

值类型派生自System.ValueType（派生自 System.Object）。 派生自 System.ValueType 的类型在 CLR 中具有特殊行为。 值类型变量直接包含其值。 结构的内存在声明变量的任何上下文中进行内联分配。 对于值类型变量，没有单独的堆分配或垃圾回收开销。 可以声明属于值类型的 `record struct` 类型，并包括记录的合成成员。

值类型分为两类：`struct`和`enum`。

内置的数值类型是结构，它们具有可访问的字段和方法：

```
C#
```

```
// constant field on type byte.
byte b = byte.MaxValue;
```

但可将这些类型视为简单的非聚合类型，为其声明并赋值：

C#

```
byte num = 0xA;
int i = 5;
char c = 'Z';
```

值类型已密封。 不能从任何值类型（例如 System.Int32）派生类型。 不能将结构定义为从任何用户定义的类或结构继承，因为结构只能从 System.ValueType 继承。 但是，一个结构可以实现一个或多个接口。 可将结构类型强制转换为其实现的任何接口类型。 这将导致“装箱”操作，以将结构包装在托管堆上的引用类型对象内。 当你将值类型传递给使用 System.Object 或任何接口类型作为输入参数的方法时，就会发生装箱操作。 有关详细信息，请参阅装箱和取消装箱。

使用 struct 关键字可以创建你自己的自定义值类型。 结构通常用作一小组相关变量的容器，如以下示例所示：

C#

```
public struct Coords
{
    public int x, y;

    public Coords(int p1, int p2)
    {
        x = p1;
        y = p2;
    }
}
```

有关结构的详细信息，请参阅结构类型。 有关值类型的详细信息，请参阅值类型。

另一种值类型是 `enum`。 枚举定义的是一组已命名的整型常量。 例如，.NET 类库中的 System.IO.FileMode 枚举包含一组已命名的常量整数，用于指定打开文件应采用的方式。 下面的示例展示了具体定义：

C#

```
public enum FileMode
{
    CreateNew = 1,
    Create = 2,
```

```
    Open = 3,
    OpenOrCreate = 4,
    Truncate = 5,
    Append = 6,
}
```

System.IO.FileMode.Create 常量的值为 2。 不过，名称对于阅读源代码的人来说更有意义，因此，最好使用枚举，而不是常量数字文本。 有关详细信息，请参阅 System.IO.FileMode。

所有枚举从 System.Enum（继承自 System.ValueType）继承。 适用于结构的所有规则也适用于枚举。 有关枚举的详细信息，请参阅枚举类型。

引用类型

定义为 `class`、`record`、delegate、数组或 interface 的类型是 reference type。

在声明变量 reference type 时，它将包含值 null，直到你将其分配给该类型的实例，或者使用 new 运算符创建一个。 下面的示例演示了如何创建和分配类：

C#

```
MyClass myClass = new MyClass();
MyClass myClass2 = myClass;
```

无法使用 new 运算符直接实例化 interface。 而是创建并分配实现接口的类实例。 请考虑以下示例：

C#

```
MyClass myClass = new MyClass();

// Declare and assign using an existing value.
IMyInterface myInterface = myClass;

// Or create and assign a value in a single statement.
IMyInterface myInterface2 = new MyClass();
```

创建对象时，会在托管堆上分配内存。 变量只保留对对象位置的引用。 对于托管堆上的类型，在分配内存和回收内存时都会产生开销。 “垃圾回收”是 CLR 的自动内存管理功能，用于执行回收。 但是，垃圾回收已是高度优化，并且在大多数情况下，不会产生性能问题。 有关垃圾回收的详细信息，请参阅自动内存管理。

所有数组都是引用类型，即使元素是值类型，也不例外。 数组隐式派生自 System.Array 类。 可以使用 C# 提供的简化语法声明和使用数组，如以下示例所示：

C#

```
// Declare and initialize an array of integers.
int[] nums = { 1, 2, 3, 4, 5 };

// Access an instance property of System.Array.
int len = nums.Length;
```

引用类型完全支持继承。 创建类时，可以从其他任何未定义为密封的接口或类继承。 其他类可以从你的类继承并替代虚拟方法。 若要详细了解如何创建你自己的类，请参阅类、结构和记录。 有关继承和虚方法的详细信息，请参阅继承。

文本值的类型

在 C# 中，文本值从编译器接收类型。 可以通过在数字末尾追加一个字母来指定数字文本应采用的类型。 例如，若要指定应按 `float` 来处理值 `4.56`，则在该数字后追加一个“f”或“F”，即 `4.56f`。 如果没有追加字母，那么编译器就会推断文本值的类型。 若要详细了解可以使用字母后缀指定哪些类型，请参阅整型数值类型和浮点数值类型。

由于文本已类型化，且所有类型最终都是从 System.Object 派生，因此可以编写和编译如下所示的代码：

C#

```
string s = "The answer is " + 5.ToString();
// Outputs: "The answer is 5"
Console.WriteLine(s);

Type type = 12345.GetType();
// Outputs: "System.Int32"
Console.WriteLine(type);
```

泛型类型

可使用一个或多个类型参数声明的类型，用作实际类型（具体类型）的占位符 。 客户端代码在创建类型实例时提供具体类型。 这种类型称为泛型类型。 例如，.NET 类型 System.Collections.Generic.List<T> 具有一个类型参数，它按照惯例被命名为 `T`。 当创建类型的实例时，指定列表将包含的对象的类型，例如 `string`：

C#

```
List<string> stringList = new List<string>();
stringList.Add("String example");
```

```
// compile time error adding a type other than a string:
stringList.Add(4);
```

通过使用类型参数，可重新使用相同类以保存任意类型的元素，且无需将每个元素转换为对象。 泛型集合类称为强类型集合，因为编译器知道集合元素的具体类型，并能在编译时引发错误，例如当尝试向上面示例中的 `stringList` 对象添加整数时。 有关详细信息，请参阅泛型。

隐式类型、匿名类型和可以为 null 的值类型

你可以使用 var 关键字隐式键入一个局部变量（但不是类成员）。 变量仍可在编译时获取类型，但类型是由编译器提供。 有关详细信息，请参阅隐式类型局部变量。

不方便为不打算存储或传递外部方法边界的简单相关值集合创建命名类型。 因此，可以创建*匿名类型*。 有关详细信息，请参阅匿名类型。

普通值类型不能具有 null 值。 不过，可以在类型后面追加 `?`，创建可为空的值类型。 例如，`int?` 是还可以包含值 null 的 `int` 类型。 可以为 null 的值类型是泛型结构类型 System.Nullable<T> 的实例。 在将数据传入和传出数据库（数值可能为 `null`）时，可为空的值类型特别有用。 有关详细信息，请参阅可以为 null 的值类型。

编译时类型和运行时类型

变量可以具有不同的编译时和运行时类型。 编译时类型是源代码中变量的声明或推断类型。 运行时类型是该变量所引用的实例的类型。 这两种类型通常是相同的，如以下示例中所示：

C#

```
string message = "This is a string of characters";
```

在其他情况下，编译时类型是不同的，如以下两个示例所示：

C#

```
object anotherMessage = "This is another string of characters";
IEnumerable<char> someCharacters = "abcdefghijklmnopqrstuvwxyz";
```

在上述两个示例中，运行时类型为 `string`。 编译时类型在第一行中为 `object`，在第二行中为 `IEnumerable<char>`。

如果变量的这两种类型不同，请务必了解编译时类型和运行时类型的应用情况。 编译时类型确定编译器执行的所有操作。 这些编译器操作包括方法调用解析、重载决策以及可用的隐式和显式强制转换。 运行时类型确定在运行时解析的所有操作。 这些运行时操作包括调度虚拟方法调用、计算 `is` 和 `switch` 表达式以及其他类型的测试 API。 为了更好地了解代码如何与类型进行交互，请识别哪个操作应用于哪种类型。

相关章节

有关详细信息，请参阅以下文章：

- Builtin 类型
- 值类型
- 引用类型

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。

反馈

此页面是否有帮助? 是 否

提供产品反馈

声明命名空间来整理类型

项目 • 2024/12/18

在 C# 编程中，命名空间在两个方面被大量使用。 首先，.NET 使用命名空间来组织它的许多类，如下所示：

```
C#

System.Console.WriteLine("Hello World!");
```

System 是一个命名空间，Console 是该命名空间中的一个类。 可使用 `using` 关键字，这样就不必使用完整的名称，如下例所示：

```
C#

using System;
```

```
C#

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

有关详细信息，请参阅 using 指令。

> **ⓘ 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 指令 `global using`：
>
> - 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**隐式 using 指令**一文

其次，在较大的编程项目中，声明自己的命名空间可以帮助控制类和方法名称的范围。 使用 namespace 关键字可声明命名空间，如下例所示：

C#

```
namespace SampleNamespace
{
    class SampleClass
    {
        public void SampleMethod()
        {
            System.Console.WriteLine(
                "SampleMethod inside SampleNamespace");
        }
    }
}
```

命名空间的名称必须是有效的 C# 标识符名称。

可以为该文件中定义的所有类型声明命名空间，如以下示例所示：

C#

```
namespace SampleNamespace;

class AnotherSampleClass
{
    public void AnotherSampleMethod()
    {
        System.Console.WriteLine(
            "SampleMethod inside SampleNamespace");
    }
}
```

这种新语法的优点是更简单，这节省了水平空间且不必使用大括号。 这使得你的代码易于阅读。

命名空间概述

命名空间具有以下属性：

- 它们组织大型代码项目。
- 通过使用 `.` 运算符分隔它们。
- `using` 指令可免去为每个类指定命名空间的名称。
- `global` 命名空间是“根”命名空间：`global::System` 始终引用 .NET System 命名空间。

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。

类简介

项目 • 2024/08/28

引用类型

定义为 class 的类型是*引用类型*。 在运行时，如果声明引用类型的变量，此变量就会一直包含值 null，直到使用 new 运算符显式创建类实例，或直到为此变量分配已在其他位置创建的兼容类型，如下面的示例所示：

C#

```
//Declaring an object of type MyClass.
MyClass mc = new MyClass();

//Declaring another object of the same type, assigning it the value of the
first object.
MyClass mc2 = mc;
```

创建对象时，在该托管堆上为该特定对象分足够的内存，并且该变量仅保存对所述对象位置的引用。 对象使用的内存由 CLR 的自动内存管理功能（称为*垃圾回收*）回收。 有关垃圾回收的详细信息，请参阅自动内存管理和垃圾回收。

声明类

使用后跟唯一标识符的 `class` 关键字可以声明类，如下例所示：

C#

```
//[access modifier] - [class] - [identifier]
public class Customer
{
   // Fields, properties, methods and events go here...
}
```

可选访问修饰符位于 `class` 关键字前面。 `class` 类型的默认访问权限为 `internal`。 此例中使用的是 public，因此任何人都可创建此类的实例。 类的名称遵循 `class` 关键字。 类名称必须是有效的 C# 标识符名称。 定义的其余部分是类的主体，其中定义了行为和数据。 类上的字段、属性、方法和事件统称为*类成员*。

创建对象

虽然它们有时可以互换使用，但类和对象是不同的概念。 类定义对象类型，但不是对象本身。 对象是基于类的具体实体，有时称为类的实例。

可通过使用 `new` 关键字，后跟类的名称来创建对象，如下所示：

```
C#

Customer object1 = new Customer();
```

创建类的实例后，会将一个该对象的引用传递回程序员。 在上一示例中，`object1` 是对基于 `Customer` 的对象的引用。 该引用指向新对象，但不包含对象数据本身。 事实上，可以创建对象引用，而完全无需创建对象本身：

```
C#

Customer object2;
```

不建议创建不引用对象的对象引用，因为尝试通过这类引用访问对象会在运行时失败。 但是，但实际上引用可以引用某个对象，方法是创建新对象，或者将其分配给现有对象，例如：

```
C#

Customer object3 = new Customer();
Customer object4 = object3;
```

此代码创建指向同一对象的两个对象引用。 因此，通过 `object3` 对对象做出的任何更改都会在后续使用 `object4` 时反映出来。 由于基于类的对象是通过引用来实现其引用的，因此类被称为引用类型。

构造函数和初始化

前面的部分介绍了声明类类型并创建该类型的实例的语法。 创建类型的实例时，需要确保其字段和属性已初始化为有用的值。 可通过多种方式初始化值：

- 接受默认值
- 字段初始化表达式
- 构造函数参数
- 对象初始值设定项

每个 .NET 类型都有一个默认值。 通常，对于数字类型，该值为 0，对于所有引用类型，该值为 `null`。 如果默认值在应用中是合理的，则可以依赖于该默认值。

当 .NET 默认值不是正确的值时，可以使用*字段初始化表达式*设置初始值：

C#

```
public class Container
{
    // Initialize capacity field to a default value of 10:
    private int _capacity = 10;
}
```

可以通过定义负责设置初始值的*构造函数*来要求调用方提供初始值：

C#

```
public class Container
{
    private int _capacity;

    public Container(int capacity) => _capacity = capacity;
}
```

从 C# 12 开始，可以将*主构造函数*定义为类声明的一部分：

C#

```
public class Container(int capacity)
{
    private int _capacity = capacity;
}
```

向类名添加参数可定义*主构造函数*。 这些参数在包含其成员的类正文中可用。 可以将其用于初始化字段或需要它们的任何其他位置。

还可以对某个属性使用 `required` 修饰符，并允许调用方使用*对象初始值设定项*来设置该属性的初始值：

C#

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

添加 `required` 关键字要求调用方必须将这些属性设置为 `new` 表达式的一部分：

C#

```
var p1 = new Person(); // Error! Required properties not set
var p2 = new Person() { FirstName = "Grace", LastName = "Hopper" };
```

类继承

类完全支持*继承*，这是面向对象的编程的基本特点。 创建类时，可以从其他任何未定义为 sealed 的类继承。 其他类可以从你的类继承并替代类虚拟方法。 此外，你可以实现一个或多个接口。

继承是通过使用*派生*来完成的，这意味着类是通过使用其数据和行为所派生自的*基类*来声明的。 基类通过在派生的类名称后面追加冒号和基类名称来指定，如：

C#

```
public class Manager : Employee
{
    // Employee fields, properties, methods and events are inherited
    // New Manager fields, properties, methods and events go here...
}
```

类声明包括基类时，它会继承基类除构造函数外的所有成员。 有关详细信息，请参阅继承。

C# 中的类只能直接从基类继承。 但是，因为基类本身可能继承自其他类，因此类可能间接继承多个基类。 此外，类可以支持实现一个或多个接口。 有关详细信息，请参阅接口。

类可以声明为 abstract。 抽象类包含抽象方法，抽象方法包含签名定义但不包含实现。 抽象类不能实例化。 只能通过可实现抽象方法的派生类来使用该类。 与此相反，密封类不允许其他类继承。 有关详细信息，请参阅抽象类、密封类和类成员。

类定义可以在不同的源文件之间分割。 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。

C# 中的记录类型简介

项目 • 2024/05/21

C# 中的记录是一个类或结构，它为使用数据模型提供特定的语法和行为。 `record` 修饰符指示编译器合成对主要角色存储数据的类型有用的成员。 这些成员包括支持值相等的 ToString() 和成员的重载。

何时使用记录

在下列情况下，请考虑使用记录而不是类或结构：

- 你想要定义依赖值相等性的数据模型。
- 你想要定义对象不可变的类型。

值相等性

对记录来说，值相等性是指如果记录类型的两个变量类型相匹配，且所有属性和字段值都相同，那么记录类型的两个变量是相等的。 对于其他引用类型（例如类），相等性默认指引用相等性，除非执行了值相等性。 也就是说，如果类的两个变量引用同一个对象，则这两个变量是相等的。 确定两个记录实例的相等性的方法和运算符使用值相等性。

并非所有数据模型都适合使用值相等性。 例如，Entity Framework Core 依赖引用相等性，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，记录类型不适合用作 Entity Framework Core 中的实体类型。

不可变性

不可变类型会阻止你在对象实例化后更改该对象的任何属性或字段值。 如果你需要一个类型是线程安全的，或者需要哈希代码在哈希表中能保持不变，那么不可变性很有用。 记录为创建和使用不可变类型提供了简洁的语法。

不可变性并不适用于所有数据方案。 例如，Entity Framework Core 不支持通过不可变实体类型进行更新。

记录与类和结构的区别

声明和实例化类或结构时使用的语法与操作记录时的相同。 只是将 `class` 关键字替换为 `record`，或者使用 `record struct` 而不是 `struct`。 同样地，记录类支持相同的表示继承关系的语法。 记录与类的区别如下所示：

- 可在主构造函数中使用位置参数来创建和实例化具有不可变属性的类型。
- 在类中指示引用相等性或不相等的方法和运算符（例如 Object.Equals(Object) 和 `==`）在记录中指示值相等性或不相等。
- 可使用 with 表达式对不可变对象创建在所选属性中具有新值的副本。
- 记录的 `ToString` 方法会创建一个格式字符串，它显示对象的类型名称及其所有公共属性的名称和值。
- 记录可从另一个记录继承。 但记录不可从类继承，类也不可从记录继承。

记录结构与结构的不同之处是，编译器合成了方法来确定相等性和 `ToString`。 编译器为位置记录结构合成 `Deconstruct` 方法。

编译器为 `record class` 中的每个主构造函数参数合成一个公共仅初始化属性。 在 `record struct` 中，编译器合成公共读写属性。 编译器不会在不包含 `record` 修饰符的 `class` 和 `struct` 类型中创建主构造函数参数的属性。

示例

下面的示例定义了一个公共记录，它使用位置参数来声明和实例化记录。 然后，它会输出类型名称和属性值：

C#

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

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

}
```

下面的示例演示了记录中的值相等性：

C#

```
public record Person(string FirstName, string LastName, string[]
PhoneNumbers);
public static class Program
{
    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
    }
}
```

下面的示例演示如何使用 `with` 表达式来复制不可变对象和更改其中的一个属性：

C#

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

public class Program
{
    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
    }
}
```

有关详细信息，请查看记录（C# 参考）。

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。

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

接口 - 定义多种类型的行为

项目 • 2023/03/18

接口包含非抽象 class 或 struct 必须实现的一组相关功能的定义。 接口可以定义 `static` 方法，此类方法必须具有实现。 接口可为成员定义默认实现。 接口不能声明实例数据，如字段、自动实现的属性或类似属性的事件。

例如，使用接口可以在类中包括来自多个源的行为。 该功能在 C# 中十分重要，因为该语言不支持类的多重继承。 此外，如果要模拟结构的继承，也必须使用接口，因为它们无法实际从另一个结构或类继承。

可使用 interface 关键字定义接口，如以下示例所示。

C#

```
interface IEquatable<T>
{
    bool Equals(T obj);
}
```

接口名称必须是有效的 C# 标识符名称。 按照约定，接口名称以大写字母 `I` 开头。

实现 IEquatable<T> 接口的任何类或结构都必须包含与该接口指定的签名匹配的 Equals 方法的定义。 因此，可以依靠实现 `IEquatable<T>` 的类来包含 `Equals` 方法，类的实例可以通过该方法确定它是否等于相同类的另一个实例。

`IEquatable<T>` 的定义不为 `Equals` 提供实现。 类或结构可以实现多个接口，但是类只能从单个类继承。

有关抽象类的详细信息，请参阅抽象类、密封类及类成员。

接口可以包含实例方法、属性、事件、索引器或这四种成员类型的任意组合。 接口可以包含静态构造函数、字段、常量或运算符。 从 C# 11 开始，非字段接口成员可以是 `static abstract`。 接口不能包含实例字段、实例构造函数或终结器。 接口成员默认是公共的，可以显式指定可访问性修饰符（如 `public`、`protected`、`internal`、`private`、`protected internal` 或 `private protected`）。 `private` 成员必须有默认实现。

若要实现接口成员，实现类的对应成员必须是公共、非静态，并且具有与接口成员相同的名称和签名。

> **备注**

当接口声明静态成员时，实现该接口的类型也可能声明具有相同签名的静态成员。 它们是不同的，并且由声明成员的类型唯一标识。 在类型中声明的静态成员不会覆盖接口中声明的静态成员。

实现接口的类或结构必须为所有已声明的成员提供实现，而非接口提供的默认实现。 但是，如果基类实现接口，则从基类派生的任何类都会继承该实现。

下面的示例演示 IEquatable<T> 接口的实现。 实现类 `Car` 必须提供 Equals 方法的实现。

C#

```
public class Car : IEquatable<Car>
{
    public string? Make { get; set; }
    public string? Model { get; set; }
    public string? Year { get; set; }

    // Implementation of IEquatable<T> interface
    public bool Equals(Car? car)
    {
        return (this.Make, this.Model, this.Year) ==
            (car?.Make, car?.Model, car?.Year);
    }
}
```

类的属性和索引器可以为接口中定义的属性或索引器定义额外的访问器。 例如，接口可能会声明包含 get 取值函数的属性。 实现此接口的类可以声明包含 `get` 和 `get` 取值函数的同一属性。 但是，如果属性或索引器使用显式实现，则访问器必须匹配。 有关显式实现的详细信息，请参阅显式接口实现和接口属性。

接口可从一个或多个接口继承。 派生接口从其基接口继承成员。 实现派生接口的类必须实现派生接口中的所有成员，包括派生接口的基接口的所有成员。 该类可能会隐式转换为派生接口或任何其基接口。 类可能通过它继承的基类或通过其他接口继承的接口来多次包含某个接口。 但是，类只能提供接口的实现一次，并且仅当类将接口作为类定义的一部分（`class ClassName : InterfaceName`）进行声明时才能提供。 如果由于继承实现接口的基类而继承了接口，则基类会提供接口的成员的实现。 但是，派生类可以重新实现任何虚拟接口成员，而不是使用继承的实现。 当接口声明方法的默认实现时，实现该接口的任何类都会继承该实现（你需要将类实例强制转换为接口类型，才能访问接口成员上的默认实现）。

基类还可以使用虚拟成员实现接口成员。 在这种情况下，派生类可以通过重写虚拟成员来更改接口行为。 有关虚拟成员的详细信息，请参阅多态性。

接口摘要

接口具有以下属性：

- 在 8.0 以前的 C# 版本中，接口类似于只有抽象成员的抽象基类。 实现接口的类或结构必须实现其所有成员。
- 从 C# 8.0 开始，接口可以定义其部分或全部成员的默认实现。 实现接口的类或结构不一定要实现具有默认实现的成员。 有关详细信息，请参阅默认接口方法。
- 接口无法直接进行实例化。 其成员由实现接口的任何类或结构来实现。
- 一个类或结构可以实现多个接口。 一个类可以继承一个基类，还可实现一个或多个接口。

泛型类和方法

项目 • 2024/03/20

泛型向 .NET 引入了类型参数的概念。 泛型支持设计类和方法，你可在在代码中使用该类或方法时，再定义一个或多个类型参数的规范。 例如，通过使用泛型类型参数 `T`，可以编写其他客户端代码能够使用的单个类，而不会产生运行时转换或装箱操作的成本或风险，如下所示：

C#

```
// Declare the generic class.
public class GenericList<T>
{
    public void Add(T input) { }
}
class TestGenericList
{
    private class ExampleClass { }
    static void Main()
    {
        // Declare a list of type int.
        GenericList<int> list1 = new GenericList<int>();
        list1.Add(1);

        // Declare a list of type string.
        GenericList<string> list2 = new GenericList<string>();
        list2.Add("");

        // Declare a list of type ExampleClass.
        GenericList<ExampleClass> list3 = new GenericList<ExampleClass>();
        list3.Add(new ExampleClass());
    }
}
```

泛型类和泛型方法兼具可重用性、类型安全性和效率，这是非泛型类和非泛型方法无法实现的。 在编译过程中将泛型类型参数替换为类型参数。 在前面的示例中，编译器会使用 `int` 替换 `T`。 泛型通常与集合以及作用于集合的方法一起使用。 System.Collections.Generic 命名空间包含几个基于泛型的集合类。 不建议使用非泛型集合（如 ArrayList），并且仅出于兼容性目的而维护非泛型集合。 有关详细信息，请参阅 .NET 中的泛型。

你也可创建自定义泛型类型和泛型方法，以提供自己的通用解决方案，设计类型安全的高效模式。 以下代码示例演示了出于演示目的的简单泛型链接列表类。 （大多数情况下，应使用 .NET 提供的 List<T> 类，而不是自行创建类。）在通常使用具体类型来指示列表中所存储项的类型的情况下，可使用类型参数 `T`：

- 在 `AddHead` 方法中作为方法参数的类型。
- 在 `Node` 嵌套类中作为 `Data` 属性的返回类型。
- 在嵌套类中作为私有成员 `data` 的类型。

`T` 可用于 `Node` 嵌套类。 如果使用具体类型实例化 `GenericList<T>`（例如，作为 `GenericList<int>`），则出现的所有 `T` 都将替换为 `int`。

C#

```
// type parameter T in angle brackets
public class GenericList<T>
{
    // The nested class is also generic on T.
    private class Node
    {
        // T used in non-generic constructor.
        public Node(T t)
        {
            next = null;
            data = t;
        }

        private Node? next;
        public Node? Next
        {
            get { return next; }
            set { next = value; }
        }

        // T as private member data type.
        private T data;

        // T as return type of property.
        public T Data
        {
            get { return data; }
            set { data = value; }
        }
    }

    private Node? head;

    // constructor
    public GenericList()
    {
        head = null;
    }

    // T as method parameter type:
    public void AddHead(T t)
    {
        Node n = new Node(t);
```

```
        n.Next = head;
        head = n;
    }

    public IEnumerator<T> GetEnumerator()
    {
        Node? current = head;

        while (current != null)
        {
            yield return current.Data;
            current = current.Next;
        }
    }
}
```

以下代码示例演示了客户端代码如何使用泛型 `GenericList<T>` 类来创建整数列表。 如果更改类型参数，以下代码将创建字符串列表或任何其他自定义类型：

C#

```
class TestGenericList
{
    static void Main()
    {
        // int is the type argument
        GenericList<int> list = new GenericList<int>();

        for (int x = 0; x < 10; x++)
        {
            list.AddHead(x);
        }

        foreach (int i in list)
        {
            System.Console.Write(i + " ");
        }
        System.Console.WriteLine("\nDone");
    }
}
```

备注

泛型类型不限于类。 前面的示例使用了 `class` 类型，但你可以定义泛型 `interface` 和 `struct` 类型，包括 `record` 类型。

泛型概述

- 使用泛型类型可以最大限度地重用代码、保护类型安全性以及提高性能。
- 泛型最常见的用途是创建集合类。
- .NET 类库在 System.Collections.Generic 命名空间中包含几个新的泛型集合类。 应尽可能使用泛型集合来代替某些类，如 System.Collections 命名空间中的 ArrayList。
- 可以创建自己的泛型接口、泛型类、泛型方法、泛型事件和泛型委托。
- 可以对泛型类进行约束以访问特定数据类型的方法。
- 可以使用反射在运行时获取有关泛型数据类型中使用的类型的信息。

C# 语言规范

有关详细信息，请参阅 C# 语言规范。

请参阅

- .NET 中的泛型
- System.Collections.Generic

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

匿名类型

项目 • 2024/11/21

匿名类型提供了一种方便的方法，可用来将一组只读属性封装到单个对象中，而无需首先显式定义一个类型。 类型名由编译器生成，并且不能在源代码级使用。 每个属性的类型由编译器推断。

可结合使用 new 运算符和对象初始值设定项创建匿名类型。 有关对象初始值设定项的详细信息，请参阅对象和集合初始值设定项。

以下示例显示了用两个名为 `Amount` 和 `Message` 的属性进行初始化的匿名类型。

C#

```
var v = new { Amount = 108, Message = "Hello" };

// Rest the mouse pointer over v.Amount and v.Message in the following
// statement to verify that their inferred types are int and string.
Console.WriteLine(v.Amount + v.Message);
```

匿名类型通常用在查询表达式的 select 子句中，以便返回源序列中每个对象的属性子集。 有关查询的详细信息，请参阅C# 中的 LINQ。

匿名类型包含一个或多个公共只读属性。 包含其他种类的类成员（如方法或事件）为无效。 用来初始化属性的表达式不能为 `null`、匿名函数或指针类型。

最常见的方案是用其他类型的属性初始化匿名类型。 在下面的示例中，假定名为 `Product` 的类存在。 类 `Product` 包括 `Color` 和 `Price` 属性，以及你不感兴趣的其他属性：

C#

```
class Product
{
    public string? Color {get;set;}
    public  decimal Price {get;set;}
    public string? Name {get;set;}
    public string? Category {get;set;}
    public string? Size {get;set;}
}
```

匿名类型声明以 `new` 关键字开始。 声明初始化了一个只使用 `Product` 的两个属性的新类型。 使用匿名类型会导致在查询中返回的数据量变少。

如果你没有在匿名类型中指定成员名称，编译器会为匿名类型成员指定与用于初始化这些成员的属性相同的名称。 需要为使用表达式初始化的属性提供名称，如下面的示例所示。 在下面示例中，匿名类型的属性名称都为 `Price` `Color` 和 。 这些实例是类型集合中的 `products` `Product` 项：

C#

```
var productQuery =
    from prod in products
    select new { prod.Color, prod.Price };

foreach (var v in productQuery)
{
    Console.WriteLine("Color={0}, Price={1}", v.Color, v.Price);
}
```

提示

可以使用 .NET 样式规则 IDE0037 强制执行是首选推断成员名称还是显式成员名称。

还可以按另一种类型（类、结构或另一个匿名类型）的对象定义字段。 它通过使用保存此对象的变量来完成，如以下示例中所示，其中两个匿名类型是使用已实例化的用户定义类型创建的。 在这两种情况下， `product` 匿名类型的 `shipment` 字段都将 `shipmentWithBonus` `Product` 是包含其每个字段的默认值的类型。 `bonus` 字段将是编译器创建的匿名类型。

C#

```
var product = new Product();
var bonus = new { note = "You won!" };
var shipment = new { address = "Nowhere St.", product };
var shipmentWithBonus = new { address = "Somewhere St.", product, bonus };
```

通常，当使用匿名类型来初始化变量时，可以通过使用 var 将变量作为隐式键入的本地变量来进行声明。 类型名称无法在变量声明中给出，因为只有编译器能访问匿名类型的基础名称。 有关 `var` 的详细信息，请参阅隐式类型本地变量。

可通过将隐式键入的本地变量与隐式键入的数组相结合创建匿名键入的元素的数组，如下面的示例所示。

C#

```
var anonArray = new[] { new { name = "apple", diam = 4 }, new { name =
"grape", diam = 1 }};
```

匿名类型是 class 类型，它们直接派生自 object，并且无法强制转换为除 object 外的任何类型。 虽然你的应用程序不能访问它，编译器还是提供了每一个匿名类型的名称。 从公共语言运行时的角度来看，匿名类型与任何其他引用类型没有什么不同。

如果程序集中的两个或多个匿名对象初始值指定了属性序列，这些属性采用相同顺序且具有相同的名称和类型，则编译器将对象视为相同类型的实例。 它们共享同一编译器生成的类型信息。

匿名类型支持采用 with 表达式形式的非破坏性修改。 这使你能够创建匿名类型的新实例，其中一个或多个属性具有新值：

C#

```
var apple = new { Item = "apples", Price = 1.35 };
var onSale = apple with { Price = 0.79 };
Console.WriteLine(apple);
Console.WriteLine(onSale);
```

无法将字段、属性、时间或方法的返回类型声明为具有匿名类型。 同样，你不能将方法、属性、构造函数或索引器的形参声明为具有匿名类型。 要将匿名类型或包含匿名类型的集合作为参数传递给某一方法，可将参数作为类型 `object` 进行声明。 但是，对匿名类型使用 `object` 违背了强类型的目的。 如果必须存储查询结果或者必须将查询结果传递到方法边界外部，请考虑使用普通的命名结构或类而不是匿名类型。

由于匿名类型上的 Equals 和 GetHashCode 方法是根据方法属性的 `Equals` 和 `GetHashCode` 定义的，因此仅当同一匿名类型的两个实例的所有属性都相等时，这两个实例才相等。

> **备注**
>
> 匿名类型的**辅助功能级别**为 `internal`，因此在不同程序集中定义的两种匿名类型并非同一类型。 因此，当在不同的程序集中进行定义时，匿名类型的实例不能彼此相等，即使其所有属性都相等。

匿名类型确实会重写 ToString 方法，将用大括号括起来的每个属性的名称和 `ToString` 输出连接起来。

```
var v = new { Title = "Hello", Age = 24 };

Console.WriteLine(v.ToString()); // "{ Title = Hello, Age = 24 }"
```

C# 中的类、结构和记录概述

项目 • 2024/04/11

在 C# 中，某个类型（类、结构或记录）的定义的作用类似于蓝图，指定该类型可以进行哪些操作。 从本质上说，对象是按照此蓝图分配和配置的内存块。 本文概述了这些蓝图及其功能。 本系列的下一篇文章介绍对象。

封装

*封装*有时称为面向对象的编程的第一支柱或原则。 类或结构可以指定自己的每个成员对外部代码的可访问性。 可以隐藏不得在类或程序集外部使用的方法和变量，以限制编码错误或恶意攻击发生的可能性。 有关详细信息，请参阅面向对象的编程教程。

成员

类型的成员包括所有方法、字段、常量、属性和事件。 C# 没有全局变量或方法，这一点其他某些语言不同。 即使是编程的入口点（`Main` 方法），也必须在类或结构中声明（使用顶级语句时，隐式声明）。

下面列出了所有可以在类、结构或记录中声明的各种成员。

- 字段
- 常量
- 属性
- 方法
- 构造函数
- 事件
- 终结器
- 索引器
- 运算符
- 嵌套类型

有关详细信息，请参见成员。

可访问性

一些方法和属性可供类或结构外部的代码（称为“*客户端代码*”）调用或访问。 另一些方法和属性只能在类或结构本身中使用。 请务必限制代码的可访问性，仅供预期的客户端代码进行访问。 需要使用以下访问修饰符指定类型及其成员对客户端代码的可访问性：

- public
- 受保护
- internal
- protected internal
- private
- 专用受保护。

可访问性的默认值为 `private`。

继承

类（而非结构）支持继承的概念。 派生自另一个类（称为基类）的类自动包含基类的所有公共、受保护和内部成员（其构造函数和终结器除外）。

可以将类声明为 abstract，即一个或多个方法没有实现代码。 尽管抽象类无法直接实例化，但可以作为提供缺少实现代码的其他类的基类。 类还可以声明为 sealed，以阻止其他类继承。

有关详细信息，请参阅继承和多态性。

界面

类、结构和记录可以实现多个接口。 从接口实现意味着类型实现接口中定义的所有方法。 有关详细信息，请参阅接口。

泛型类型

类、结构和记录可以使用一个或多个类型参数进行定义。 客户端代码在创建类型实例时提供类型。 例如，System.Collections.Generic 命名空间中的 List<T> 类就是用一个类型参数定义的。 客户端代码创建 `List<string>` 或 `List<int>` 的实例来指定列表将包含的类型。 有关详细信息，请参阅泛型。

静态类型

类（而非结构或记录）可以声明为`static`。 静态类只能包含静态成员，不能使用 `new` 关键字进行实例化。 在程序加载时，类的一个副本会加载到内存中，而其成员则可通过类名进行访问。 类、结构和记录可以包含静态成员。 有关详细信息，请参阅静态类和静态类成员。

嵌套类型

类、结构和记录可以嵌套在其他类、结构和记录中。 有关详细信息，请参阅嵌套类型。

分部类型

可以在一个代码文件中定义类、结构或方法的一部分，并在其他代码文件中定义另一部分。 有关详细信息，请参阅分部类和方法。

对象初始值设定项

可以通过将值分配给属性来实例化和初始化类或结构对象以及对象集合。 有关详细信息，请参阅如何使用对象初始值设定项初始化对象。

匿名类型

在不方便或不需要创建命名类的情况下，可以使用匿名类型。 匿名类型由其命名数据成员定义。 有关详细信息，请参阅匿名类型。

扩展方法

可以通过创建单独的类型来“扩展”类，而无需创建派生类。 该类型包含可以调用的方法，就像它们属于原始类型一样。 有关详细信息，请参阅扩展方法。

隐式类型的局部变量

在类或结构方法中，可以使用隐式类型指示编译器在编译时确定变量类型。 有关详细信息，请参阅 var（C# 参考）。

记录

C# 9 引入了 `record` 类型，可创建此引用类型而不创建类或结构。 记录是带有内置行为的类，用于将数据封装在不可变类型中。 C# 10 引入了 `record struct` 值类型。 记录（`record class` 或 `record struct`）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。
- 值相等性。 两个记录类型的变量在它们的类型和两个记录中每个字段的值都相同时，它们是相等的。 类使用引用相等性，即：如果类类型的两个变量引用同一对象，则这两个变量是相等的。

- 非破坏性变化的简明语法。 使用 `with` 表达式，可以创建作为现有实例副本的新记录实例，但更改了指定的属性值。
- 显示的内置格式设置。 `ToString` 方法输出记录类型名称以及公共属性的名称和值。
- 支持记录类中的继承层次结构。 记录类支持继承。 记录结构不支持继承。

有关详细信息，请参阅记录。

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。

反馈

此页面是否有帮助? 是 否

提供产品反馈

对象 - 创建类型的实例

项目 • 2023/05/10

类或结构定义的作用类似于蓝图，指定该类型可以进行哪些操作。 从本质上说，对象是按照此蓝图分配和配置的内存块。 程序可以创建同一个类的多个对象。 对象也称为实例，可以存储在命名变量中，也可以存储在数组或集合中。 使用这些变量来调用对象方法及访问对象公共属性的代码称为客户端代码。 在 C# 等面向对象的语言中，典型的程序由动态交互的多个对象组成。

> **备注**
>
> 静态类型的行为与此处介绍的不同。 有关详细信息，请参阅**静态类和静态类成员**。

结构实例与类实例

由于类是引用类型，因此类对象的变量引用该对象在托管堆上的地址。 如果将同一类型的第二个变量分配给第一个变量，则两个变量都引用该地址的对象。 本文稍后部分将更详细地讨论这一点。

类的实例是使用 new 运算符创建的。 在下面的示例中，`Person` 为类型，`person1` 和 `person2` 为该类型的实例（即对象）。

C#

```
public class Person
{
    public string Name { get; set; }
    public int Age { get; set; }
    public Person(string name, int age)
    {
        Name = name;
        Age = age;
    }
    // Other properties, methods, events...
}

class Program
{
    static void Main()
    {
        Person person1 = new Person("Leopold", 6);
        Console.WriteLine("person1 Name = {0} Age = {1}", person1.Name,
person1.Age);

        // Declare new person, assign person1 to it.
```

```
        Person person2 = person1;

        // Change the name of person2, and person1 also changes.
        person2.Name = "Molly";
        person2.Age = 16;

        Console.WriteLine("person2 Name = {0} Age = {1}", person2.Name,
person2.Age);
        Console.WriteLine("person1 Name = {0} Age = {1}", person1.Name,
person1.Age);
    }
}
/*
    Output:
    person1 Name = Leopold Age = 6
    person2 Name = Molly Age = 16
    person1 Name = Molly Age = 16
*/
```

由于结构是值类型，因此结构对象的变量具有整个对象的副本。 结构的实例也可使用 `new` 运算符来创建，但这不是必需的，如下面的示例所示：

C#

```
namespace Example;

public struct Person
{
    public string Name;
    public int Age;
    public Person(string name, int age)
    {
        Name = name;
        Age = age;
    }
}

public class Application
{
    static void Main()
    {
        // Create  struct instance and initialize by using "new".
        // Memory is allocated on thread stack.
        Person p1 = new Person("Alex", 9);
        Console.WriteLine("p1 Name = {0} Age = {1}", p1.Name, p1.Age);

        // Create  new struct object. Note that  struct can be initialized
        // without using "new".
        Person p2 = p1;

        // Assign values to p2 members.
        p2.Name = "Spencer";
        p2.Age = 7;
```

```
        Console.WriteLine("p2 Name = {0} Age = {1}", p2.Name, p2.Age);

        // p1 values remain unchanged because p2 is  copy.
        Console.WriteLine("p1 Name = {0} Age = {1}", p1.Name, p1.Age);
    }
}
/*
    Output:
    p1 Name = Alex Age = 9
    p2 Name = Spencer Age = 7
    p1 Name = Alex Age = 9
*/
```

`p1` 和 `p2` 的内存在线程堆栈上进行分配。 该内存随声明它的类型或方法一起回收。 这就是在赋值时复制结构的一个原因。 相比之下，当对类实例对象的所有引用都超出范围时，为该类实例分配的内存将由公共语言运行时自动回收（垃圾回收）。 无法像在 C++ 中那样明确地销毁类对象。 有关 .NET 中的垃圾回收的详细信息，请参阅垃圾回收。

> **备注**
>
> 公共语言运行时中高度优化了托管堆上内存的分配和释放。 在大多数情况下，在堆上分配类实例与在堆栈上分配结构实例在性能成本上没有显著的差别。

对象标识与值相等性

在比较两个对象是否相等时，首先必须明确是想知道两个变量是否表示内存中的同一对象，还是想知道这两个对象的一个或多个字段的值是否相等。 如果要对值进行比较，则必须考虑这两个对象是值类型（结构）的实例，还是引用类型（类、委托、数组）的实例。

- 若要确定两个类实例是否引用内存中的同一位置（这意味着它们具有相同的标识），可使用静态 Object.Equals 方法。 （System.Object 是所有值类型和引用类型的隐式基类，其中包括用户定义的结构和类。）

- 若要确定两个结构实例中的实例字段是否具有相同的值，可使用 ValueType.Equals 方法。 由于所有结构都隐式继承自 System.ValueType，因此可以直接在对象上调用该方法，如以下示例所示：

  C#

  ```
  // Person is defined in the previous example.

  //public struct Person
  //{
  //    public string Name;
  ```

```
//    public int Age;
//    public Person(string name, int age)
//    {
//        Name = name;
//        Age = age;
//    }
//}

Person p1 = new Person("Wallace", 75);
Person p2 = new Person("", 42);
p2.Name = "Wallace";
p2.Age = 75;

if (p2.Equals(p1))
    Console.WriteLine("p2 and p1 have the same values.");

// Output: p2 and p1 have the same values.
```

`Equals` 的 System.ValueType 实现在某些情况下使用装箱和反射。 若要了解如何提供特定于类型的高效相等性算法，请参阅如何为类型定义值相等性。 记录是使用值语义实现相等性的引用类型。

- 若要确定两个类实例中字段的值是否相等，可以使用 Equals 方法或 == 运算符。 但是，只有类通过重写或重载提供关于那种类型对象的“相等”含义的自定义时，才能使用它们。 类也可能实现 IEquatable<T> 接口或 IEqualityComparer<T> 接口。 这两个接口都提供可用于测试值相等性的方法。 设计好替代 `Equals` 的类后，请务必遵循如何为类型定义值相等性和 Object.Equals(Object) 中介绍的准则。

相关章节

更多相关信息：

- 类
- 构造函数
- 终结器
- 事件
- object
- 继承
- class
- 结构类型
- new 运算符
- 常规类型系统

继承 - 派生用于创建更具体的行为的类型

项目 • 2023/04/07

继承（以及封装和多态性）是面向对象的编程的三个主要特征之一。 通过继承，可以创建新类，以便重用、扩展和修改在其他类中定义的行为。 其成员被继承的类称为“基类”，继承这些成员的类称为“派生类”。 派生类只能有一个直接基类。 但是，继承是可传递的。 如果 `ClassC` 派生自 `ClassB`，并且 `ClassB` 派生自 `ClassA`，则 `ClassC` 将继承在 `ClassB` 和 `ClassA` 中声明的成员。

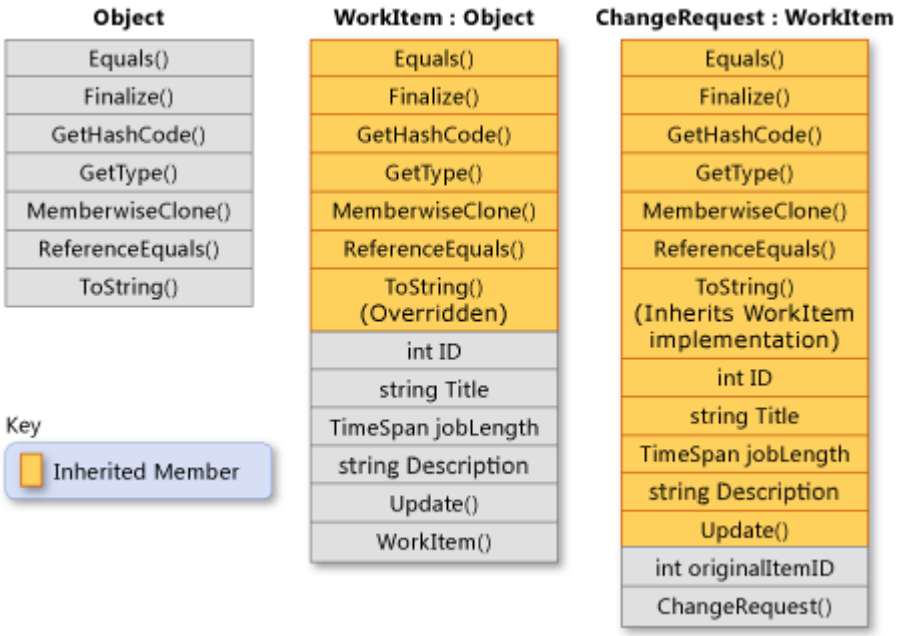
> **备注**
>
> 结构不支持继承，但它们可以实现接口。

从概念上讲，派生类是基类的专门化。 例如，如果有一个基类 `Animal`，则可以有一个名为 `Mammal` 的派生类，以及另一个名为 `Reptile` 的派生类。 `Mammal` 是 `Animal`，`Reptile` 也是 `Animal`，但每个派生类表示基类的不同专门化。

接口声明可以为其成员定义默认实现。 这些实现通过派生接口和实现这些接口的类来继承。 有关默认接口方法的详细信息，请参阅关于接口的文章。

定义要从其他类派生的类时，派生类会隐式获得基类的所有成员（除了其构造函数和终结器）。 派生类可以重用基类中的代码，而无需重新实现。 可以在派生类中添加更多成员。 派生类扩展了基类的功能。

下图显示一个类 `WorkItem`，它表示某个业务流程中的工作项。 像所有类一样，它派生自 System.Object 且继承其所有方法。 `WorkItem` 会添加其自己的六个成员。 这些成员中包括一个构造函数，因为不会继承构造函数。 类 `ChangeRequest` 继承自 `WorkItem`，表示特定类型的工作项。 `ChangeRequest` 将另外两个成员添加到它从 `WorkItem` 和 Object 继承的成员中。 它必须添加自己的构造函数，并且还添加了 `originalItemID`。 属性 `originalItemID` 使 `ChangeRequest` 实例可以与向其应用更改请求的原始 `WorkItem` 相关联。



下面的示例演示如何在 C# 中表示前面图中所示的类关系。 该示例还演示了 `WorkItem` 替代虚方法 Object.ToString 的方式，以及 `ChangeRequest` 类继承该方法的 `WorkItem` 的实现方式。 第一个块定义类：

C#

```
// WorkItem implicitly inherits from the Object class.
public class WorkItem
{
    // Static field currentID stores the job ID of the last WorkItem that
    // has been created.
    private static int currentID;

    //Properties.
    protected int ID { get; set; }
    protected string Title { get; set; }
    protected string Description { get; set; }
    protected TimeSpan jobLength { get; set; }

    // Default constructor. If a derived class does not invoke a base-
    // class constructor explicitly, the default constructor is called
    // implicitly.
    public WorkItem()
    {
        ID = 0;
        Title = "Default title";
        Description = "Default description.";
        jobLength = new TimeSpan();
    }

    // Instance constructor that has three parameters.
    public WorkItem(string title, string desc, TimeSpan joblen)
    {
        this.ID = GetNextID();
        this.Title = title;
```

```
        this.Description = desc;
        this.jobLength = joblen;
    }

    // Static constructor to initialize the static member, currentID. This
    // constructor is called one time, automatically, before any instance
    // of WorkItem or ChangeRequest is created, or currentID is referenced.
    static WorkItem() => currentID = 0;

    // currentID is a static field. It is incremented each time a new
    // instance of WorkItem is created.
    protected int GetNextID() => ++currentID;

    // Method Update enables you to update the title and job length of an
    // existing WorkItem object.
    public void Update(string title, TimeSpan joblen)
    {
        this.Title = title;
        this.jobLength = joblen;
    }

    // Virtual method override of the ToString method that is inherited
    // from System.Object.
    public override string ToString() =>
        $"{this.ID} - {this.Title}";
}

// ChangeRequest derives from WorkItem and adds a property (originalItemID)
// and two constructors.
public class ChangeRequest : WorkItem
{
    protected int originalItemID { get; set; }

    // Constructors. Because neither constructor calls a base-class
    // constructor explicitly, the default constructor in the base class
    // is called implicitly. The base class must contain a default
    // constructor.

    // Default constructor for the derived class.
    public ChangeRequest() { }

    // Instance constructor that has four parameters.
    public ChangeRequest(string title, string desc, TimeSpan jobLen,
                         int originalID)
    {
        // The following properties and the GetNexID method are inherited
        // from WorkItem.
        this.ID = GetNextID();
        this.Title = title;
        this.Description = desc;
        this.jobLength = jobLen;

        // Property originalItemID is a member of ChangeRequest, but not
        // of WorkItem.
        this.originalItemID = originalID;
```

```
    }
}
```

下一个块显示如何使用基类和派生类：

C#

```
// Create an instance of WorkItem by using the constructor in the
// base class that takes three arguments.
WorkItem item = new WorkItem("Fix Bugs",
                             "Fix all bugs in my code branch",
                             new TimeSpan(3, 4, 0, 0));

// Create an instance of ChangeRequest by using the constructor in
// the derived class that takes four arguments.
ChangeRequest change = new ChangeRequest("Change Base Class Design",
                                         "Add members to the class",
                                         new TimeSpan(4, 0, 0),
                                         1);

// Use the ToString method defined in WorkItem.
Console.WriteLine(item.ToString());

// Use the inherited Update method to change the title of the
// ChangeRequest object.
change.Update("Change the Design of the Base Class",
    new TimeSpan(4, 0, 0));

// ChangeRequest inherits WorkItem's override of ToString.
Console.WriteLine(change.ToString());
/* Output:
    1 - Fix Bugs
    2 - Change the Design of the Base Class
*/
```

抽象方法和虚方法

基类将方法声明为 virtual 时，派生类可以使用其自己的实现override该方法。 如果基类将成员声明为 abstract，则必须在直接继承自该类的任何非抽象类中重写该方法。 如果派生类本身是抽象的，则它会继承抽象成员而不会实现它们。 抽象和虚拟成员是多形性（面向对象的编程的第二个主要特征）的基础。 有关详细信息，请参阅多态性。

抽象基类

如果要通过使用 new 运算符来防止直接实例化，则可以将类声明为抽象。 只有当一个新类派生自该类时，才能使用抽象类。 抽象类可以包含一个或多个本身声明为抽象的方法签名。 这些签名指定参数和返回值，但没有任何实现（方法体）。 抽象类不必包含抽象

成员；但是，如果类包含抽象成员，则类本身必须声明为抽象。 本身不抽象的派生类必须为来自抽象基类的任何抽象方法提供实现。

接口

接口是定义一组成员的引用类型。 实现该接口的所有类和结构都必须实现这组成员。 接口可以为其中任何成员或全部成员定义默认实现。 类可以实现多个接口，即使它只能派生自单个直接基类。

接口用于为类定义特定功能，这些功能不一定具有“is a (是)”关系。 例如，System.IEquatable<T> 接口可由任何类或结构实现，以确定该类型的两个对象是否等效（但是由该类型定义等效性）。 IEquatable<T> 不表示基类和派生类之间存在的同一种“是”关系（例如，`Mammal` 是 `Animal`）。 有关详细信息，请参阅接口。

防止进一步派生

类可以通过将自己或成员声明为 sealed，来防止其他类继承自它或继承自其任何成员。

基类成员的派生类隐藏

派生类可以通过使用相同名称和签名声明成员来隐藏基类成员。 new 修饰符可以用于显式指示成员不应作为基类成员的重写。 使用 new 不是必需的，但如果未使用 new，则会产生编译器警告。 有关详细信息，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。

多态性

项目 • 2023/04/07

多态性常被视为自封装和继承之后，面向对象的编程的第三个支柱。 Polymorphism（多态性）是一个希腊词，指“多种形态”，多态性具有两个截然不同的方面：

- 在运行时，在方法参数和集合或数组等位置，派生类的对象可以作为基类的对象处理。 在出现此多形性时，该对象的声明类型不再与运行时类型相同。
- 基类可以定义并实现虚方法，派生类可以重写这些方法，即派生类提供自己的定义和实现。 在运行时，客户端代码调用该方法，CLR 查找对象的运行时类型，并调用虚方法的重写方法。 你可以在源代码中调用基类的方法，执行该方法的派生类版本。

虚方法允许你以统一方式处理多组相关的对象。 例如，假定你有一个绘图应用程序，允许用户在绘图图面上创建各种形状。 你在编译时不知道用户将创建哪些特定类型的形状。 但应用程序必须跟踪创建的所有类型的形状，并且必须更新这些形状以响应用户鼠标操作。 你可以使用多态性通过两个基本步骤解决这一问题：

1. 创建一个类层次结构，其中每个特定形状类均派生自一个公共基类。
2. 使用虚方法通过对基类方法的单个调用来调用任何派生类上的相应方法。

首先，创建一个名为 `Rectangle` `Shape` 的基类，并创建一些派生类，例如 `Triangle` `Circle`、 和 。 为 `Shape` 类提供一个名为 `Draw` 的虚拟方法，并在每个派生类中重写该方法以绘制该类表示的特定形状。 创建 `List<Shape>` 对象，并向其添加 `Circle`、 `Triangle` 和 `Rectangle`。

C#

```
public class Shape
{
    // A few example members
    public int X { get; private set; }
    public int Y { get; private set; }
    public int Height { get; set; }
    public int Width { get; set; }

    // Virtual method
    public virtual void Draw()
    {
        Console.WriteLine("Performing base class drawing tasks");
    }
}

public class Circle : Shape
{
    public override void Draw()
```

```
    {
        // Code to draw a circle...
        Console.WriteLine("Drawing a circle");
        base.Draw();
    }
}
public class Rectangle : Shape
{
    public override void Draw()
    {
        // Code to draw a rectangle...
        Console.WriteLine("Drawing a rectangle");
        base.Draw();
    }
}
public class Triangle : Shape
{
    public override void Draw()
    {
        // Code to draw a triangle...
        Console.WriteLine("Drawing a triangle");
        base.Draw();
    }
}
```

若要更新绘图图面，请使用 foreach 循环对该列表进行循环访问，并对其中的每个 `Shape` 对象调用 `Draw` 方法。 虽然列表中的每个对象都具有声明类型 `Shape`，但调用的将是运行时类型（该方法在每个派生类中的重写版本）。

C#

```
// Polymorphism at work #1: a Rectangle, Triangle and Circle
// can all be used wherever a Shape is expected. No cast is
// required because an implicit conversion exists from a derived
// class to its base class.
var shapes = new List<Shape>
{
    new Rectangle(),
    new Triangle(),
    new Circle()
};

// Polymorphism at work #2: the virtual method Draw is
// invoked on each of the derived classes, not the base class.
foreach (var shape in shapes)
{
    shape.Draw();
}
/* Output:
    Drawing a rectangle
    Performing base class drawing tasks
    Drawing a triangle
```

```
    Performing base class drawing tasks
    Drawing a circle
    Performing base class drawing tasks
*/
```

在 C# 中，每个类型都是多态的，因为包括用户定义类型在内的所有类型都继承自 Object。

多形性概述

虚拟成员

当派生类从基类继承时，它包括基类的所有成员。 基类中声明的所有行为都是派生类的一部分。 这使派生类的对象能够被视为基类的对象。 访问修饰符（`public`、`protected`、`private` 等）确定是否可以从派生类实现访问这些成员。 通过虚拟方法，设计器可以选择不同的派生类行为：

- 派生类可以重写基类中的虚拟成员，并定义新行为。
- 派生类可能会继承最接近的基类方法而不重写方法，同时保留现有的行为，但允许进一步派生的类重写方法。
- 派生类可以定义隐藏基类实现的成员的新非虚实现。

仅当基类成员声明为 virtual 或 abstract 时，派生类才能重写基类成员。 派生成员必须使用 override 关键字显式指示该方法将参与虚调用。 以下代码提供了一个示例：

C#

```
public class BaseClass
{
    public virtual void DoWork() { }
    public virtual int WorkProperty
    {
        get { return 0; }
    }
}
public class DerivedClass : BaseClass
{
    public override void DoWork() { }
    public override int WorkProperty
    {
        get { return 0; }
    }
}
```

字段不能是虚拟的，只有方法、属性、事件和索引器才可以是虚拟的。 当派生类重写某个虚拟成员时，即使该派生类的实例被当作基类的实例访问，也会调用该成员。 以下代码提供了一个示例：

C#

```
DerivedClass B = new DerivedClass();
B.DoWork();  // Calls the new method.

BaseClass A = B;
A.DoWork();  // Also calls the new method.
```

虚方法和属性允许派生类扩展基类，而无需使用方法的基类实现。 有关详细信息，请参阅使用 Override 和 New 关键字进行版本控制。 接口提供另一种方式来定义将实现留给派生类的方法或方法集。

使用新成员隐藏基类成员

如果希望派生类具有与基类中的成员同名的成员，则可以使用 new 关键字隐藏基类成员。 `new` 关键字放置在要替换的类成员的返回类型之前。 以下代码提供了一个示例：

C#

```
public class BaseClass
{
    public void DoWork() { WorkField++; }
    public int WorkField;
    public int WorkProperty
    {
        get { return 0; }
    }
}

public class DerivedClass : BaseClass
{
    public new void DoWork() { WorkField++; }
    public new int WorkField;
    public new int WorkProperty
    {
        get { return 0; }
    }
}
```

通过将派生类的实例强制转换为基类的实例，可以从客户端代码访问隐藏的基类成员。 例如：

C#

```
DerivedClass B = new DerivedClass();
B.DoWork();  // Calls the new method.

BaseClass A = (BaseClass)B;
A.DoWork();  // Calls the old method.
```

阻止派生类重写虚拟成员

无论在虚拟成员和最初声明虚拟成员的类之间已声明了多少个类，虚拟成员都是虚拟的。 如果类 `A` 声明了一个虚拟成员，类 `B` 从 `A` 派生，类 `C` 从类 `B` 派生，则不管类 `B` 是否为虚拟成员声明了重写，类 `C` 都会继承该虚拟成员，并可以重写它。 以下代码提供了一个示例：

C#

```
public class A
{
    public virtual void DoWork() { }
}
public class B : A
{
    public override void DoWork() { }
}
```

派生类可以通过将重写声明为 sealed 来停止虚拟继承。 停止继承需要在类成员声明中的 `override` 关键字前面放置 `sealed` 关键字。 以下代码提供了一个示例：

C#

```
public class C : B
{
    public sealed override void DoWork() { }
}
```

在上一个示例中，方法 `DoWork` 对从 `C` 派生的任何类都不再是虚拟方法。 即使它们转换为类型 `B` 或类型 `A`，它对于 `C` 的实例仍然是虚拟的。 通过使用 `new` 关键字，密封的方法可以由派生类替换，如下面的示例所示：

C#

```
public class D : C
{
    public new void DoWork() { }
}
```

在此情况下，如果在 `D` 中使用类型为 `D` 的变量调用 `DoWork`，被调用的将是新的 `DoWork`。 如果使用类型为 `C`、`B` 或 `A` 的变量访问 `D` 的实例，对 `DoWork` 的调用将遵循虚拟继承的规则，即把这些调用传送到类 `C` 的 `DoWork` 实现。

从派生类访问基类虚拟成员

已替换或重写某个方法或属性的派生类仍然可以使用 `base` 关键字访问基类的该方法或属性。 以下代码提供了一个示例：

C#

```
public class Base
{
    public virtual void DoWork() {/*...*/ }
}
public class Derived : Base
{
    public override void DoWork()
    {
        //Perform Derived's work here
        //...
        // Call DoWork on base class
        base.DoWork();
    }
}
```

有关详细信息，请参阅 base。

> **备注**
>
> 建议虚拟成员在它们自己的实现中使用 `base` 来调用该成员的基类实现。 允许基类行为发生使得派生类能够集中精力实现特定于派生类的行为。 未调用基类实现时，由派生类负责使它们的行为与基类的行为兼容。

模式匹配概述

项目 • 2024/03/21

“模式匹配”是一种测试表达式是否具有特定特征的方法。 C# 模式匹配提供更简洁的语法，用于测试表达式并在表达式匹配时采取措施。 “is 表达式”目前支持通过模式匹配测试表达式并有条件地声明该表达式结果。 “switch 表达式”允许你根据表达式的首次匹配模式执行操作。 这两个表达式支持丰富的模式词汇。

本文概述了可以使用模式匹配的方案。 这些方法可以提高代码的可读性和正确性。 有关可以应用的所有模式的完整讨论，请参阅语言参考中有关模式的文章。

Null 检查

模式匹配最常见的方案之一是确保值不是 `null`。 使用以下示例进行 `null` 测试时，可以测试可为 null 的值类型并将其转换为其基础类型：

C#

```
int? maybe = 12;

if (maybe is int number)
{
    Console.WriteLine($"The nullable int 'maybe' has the value {number}");
}
else
{
    Console.WriteLine("The nullable int 'maybe' doesn't hold a value");
}
```

上述代码是声明模式，用于测试变量类型并将其分配给新变量。 语言规则使此方法比其他方法更安全。 变量 `number` 仅在 `if` 子句的 true 部分可供访问和分配。 如果尝试在 `else` 子句或 `if` 程序块后等其他位置访问，编译器将出错。 其次，由于不使用 `==` 运算符，因此当类型重载 `==` 运算符时，此模式有效。 这使该方法成为检查空引用值的理想方法，可以添加 `not` 模式：

C#

```
string? message = ReadMessageOrDefault();

if (message is not null)
{
    Console.WriteLine(message);
}
```

前面的示例使用常数模式将变量与 `null` 进行比较。 `not` 为一种逻辑模式，在否定模式不匹配时与该模式匹配。

类型测试

模式匹配的另一种常见用途是测试变量是否与给定类型匹配。 例如，以下代码测试变量是否为非 null 并实现 System.Collections.Generic.IList<T> 接口。 如果是，它将使用该列表中的 ICollection<T>.Count 属性来查找中间索引。 不管变量的编译时类型如何，声明模式均与 `null` 值不匹配。 除了防范未实现 `IList` 的类型之外，以下代码还可防范 `null`。

C#

```
public static T MidPoint<T>(IEnumerable<T> sequence)
{
    if (sequence is IList<T> list)
    {
        return list[list.Count / 2];
    }
    else if (sequence is null)
    {
        throw new ArgumentNullException(nameof(sequence), "Sequence can't be
null.");
    }
    else
    {
        int halfLength = sequence.Count() / 2 - 1;
        if (halfLength < 0) halfLength = 0;
        return sequence.Skip(halfLength).First();
    }
}
```

可在 `switch` 表达式中应用相同测试，用以测试多种不同类型的变量。 你可以根据特定运行时类型使用这些信息创建更好的算法。

比较离散值

你还可以通过测试变量找到特定值的匹配项。 在以下代码演示的示例中，你针对枚举中声明的所有可能值进行数值测试：

C#

```
public State PerformOperation(Operation command) =>
   command switch
   {
       Operation.SystemTest => RunDiagnostics(),
```

```
        Operation.Start => StartSystem(),
        Operation.Stop => StopSystem(),
        Operation.Reset => ResetToReady(),
        _ => throw new ArgumentException("Invalid enum value for command",
nameof(command)),
    };
```

前一个示例演示了基于枚举值的方法调度。 最终 `_` 案例为与所有数值匹配的弃元模式。 它处理值与定义的 `enum` 值之一不匹配的任何错误条件。 如果省略该开关分支，编译器会发出警告，指示模式表达式不处理所有可能的输入值。 在运行时，如果检查的对象与任何 switch 分支均不匹配，则 `switch` 表达式会引发异常。 可以使用数值常量代替枚举值集。 你还可以将这种类似的方法用于表示命令的常量字符串值：

C#

```
public State PerformOperation(string command) =>
    command switch
    {
        "SystemTest" => RunDiagnostics(),
        "Start" => StartSystem(),
        "Stop" => StopSystem(),
        "Reset" => ResetToReady(),
        _ => throw new ArgumentException("Invalid string value for command",
nameof(command)),
    };
```

前面的示例显示相同的算法，但使用字符串值代替枚举。 如果应用程序响应文本命令而不是常规数据格式，则可以使用此方案。 从 C# 11 开始，还可以使用 `Span<char>` 或 `ReadOnlySpan<char>` 来测试常量字符串值，如以下示例所示：

C#

```
public State PerformOperation(ReadOnlySpan<char> command) =>
    command switch
    {
        "SystemTest" => RunDiagnostics(),
        "Start" => StartSystem(),
        "Stop" => StopSystem(),
        "Reset" => ResetToReady(),
        _ => throw new ArgumentException("Invalid string value for command",
nameof(command)),
    };
```

在所有这些示例中，“弃元模式”可确保处理每个输入。 编译器可确保处理每个可能的输入值，为你提供帮助。

关系模式

你可以使用关系模式测试如何将数值与常量进行比较。 例如，以下代码基于华氏温度返回水源状态：

C#

```
string WaterState(int tempInFahrenheit) =>
    tempInFahrenheit switch
    {
        (> 32) and (< 212) => "liquid",
        < 32 => "solid",
        > 212 => "gas",
        32 => "solid/liquid transition",
        212 => "liquid / gas transition",
    };
```

上述代码还演示了联合 `and` 逻辑模式，用于检查两种关系模式是否匹配。 你还可以使用析取 `or` 模式检查模式匹配。 这两种关系模式括在括号中，可以在任何模式下用于清晰表述。 最后两个 switch 分支用于处理熔点和沸点的案例。 如果没有这两个分支，编译器将警告你的逻辑未涵盖每个可能的输入。

上述代码还说明了编译器为模式匹配表达式提供的另一项重要功能：如果没有处理每个输入值，编译器会发出警告。 如果一个开关分支的模式被前一模式覆盖，编译器也会发出警告。 这使你能够随意重构和重新排列 switch 表达式。 编写同一表达式的另一种方法是：

C#

```
string WaterState2(int tempInFahrenheit) =>
    tempInFahrenheit switch
    {
        < 32 => "solid",
        32 => "solid/liquid transition",
        < 212 => "liquid",
        212 => "liquid / gas transition",
        _ => "gas",
};
```

前面的示例以及任何其他重构或重新排序的重点是编译器会验证代码是否处理所有可能的输入。

多个输入

到目前为止涉及的所有模式都检查了一个输入。 可以写入检查一个对象的多个属性的模式。 请考虑以下 `Order` 记录：

C#

```
public record Order(int Items, decimal Cost);
```

前面的位置记录类型在显式位置声明两个成员。 首先出现 `Items`，然后是订单的 `Cost`。 有关详细信息，请参阅记录。

以下代码检查项数和订单值以计算折扣价：

C#

```
public decimal CalculateDiscount(Order order) =>
    order switch
    {
        { Items: > 10, Cost: > 1000.00m } => 0.10m,
        { Items: > 5, Cost: > 500.00m } => 0.05m,
        { Cost: > 250.00m } => 0.02m,
        null => throw new ArgumentNullException(nameof(order), "Can't
calculate discount on null order"),
        var someObject => 0m,
    };
```

前两个分支检查 `Order` 的两个属性。 第三个仅检查成本。 下一个检查 `null`，最后一个与其他任何值匹配。 如果 `Order` 类型定义了适当的 Deconstruct 方法，则可以省略模式的属性名称，并使用析构检查属性：

C#

```
public decimal CalculateDiscount(Order order) =>
    order switch
    {
        ( > 10,  > 1000.00m) => 0.10m,
        ( > 5, > 50.00m) => 0.05m,
        { Cost: > 250.00m } => 0.02m,
        null => throw new ArgumentNullException(nameof(order), "Can't
calculate discount on null order"),
        var someObject => 0m,
    };
```

上述代码演示了位置模式，其中表达式的属性已析构。

列表模式

可以使用列表模式检查列表或数组中的元素。 列表模式提供了一种方法，将模式应用于序列的任何元素。 此外，还可以应用弃元模式 (_) 来匹配任何元素，或者应用切片模式来匹配零个或多个元素。

当数据不遵循常规结构时，列表模式是一个有价值的工具。 可以使用模式匹配来测试数据的形状和值，而不是将其转换为一组对象。

看看下面的内容，它摘录自一个包含银行交易信息的文本文件：

输出

```
04-01-2020, DEPOSIT,    Initial deposit,            2250.00
04-15-2020, DEPOSIT,    Refund,                      125.65
04-18-2020, DEPOSIT,    Paycheck,                    825.65
04-22-2020, WITHDRAWAL, Debit,           Groceries,  255.73
05-01-2020, WITHDRAWAL, #1102,           Rent, apt, 2100.00
05-02-2020, INTEREST,                                  0.65
05-07-2020, WITHDRAWAL, Debit,           Movies,      12.57
04-15-2020, FEE,                                       5.55
```

它是 CSV 格式，但某些行的列数比其他行要多。 对处理来说更糟糕的是，`WITHDRAWAL` 类型中的一列包含用户生成的文本，并且可以在文本中包含逗号。 一个包含弃元模式、常量模式和 var 模式的列表模式用于捕获这种格式的值处理数据：

C#

```
decimal balance = 0m;
foreach (string[] transaction in ReadRecords())
{
    balance += transaction switch
    {
        [_, "DEPOSIT", _, var amount]     => decimal.Parse(amount),
        [_, "WITHDRAWAL", .., var amount] => -decimal.Parse(amount),
        [_, "INTEREST", var amount]       => decimal.Parse(amount),
        [_, "FEE", var fee]               => -decimal.Parse(fee),
        _                                 => throw new
InvalidOperationException($"Record {string.Join(", ", transaction)} is not
in the expected format!"),
    };
    Console.WriteLine($"Record: {string.Join(", ", transaction)}, New
balance: {balance:C}");
}
```

前面的示例采用了字符串数组，其中每个元素都是行中的一个字段。 第二个字段的 `switch` 表达式键，用于确定交易的类型和剩余列数。 每一行都确保数据的格式正确。 弃元模式 (_) 跳过第一个字段，以及交易的日期。 第二个字段与交易的类型匹配。 其余元

素匹配跳过包含金额的字段。 最终匹配使用 var 模式来捕获金额的字符串表示形式。 表达式计算要从余额中加上或减去的金额。

*列表模式*可以在数据元素序列的形状上进行匹配。 使用弃元模式和切片模式来匹配元素的位置。 使用其他模式来匹配各个元素的特征。

本文介绍了可以使用 C# 中的模式匹配写入的代码类型。 下面的文章显示了在方案中使用模式的更多示例，以及可供使用的完整模式词汇。

另请参阅

- 使用模式匹配来避免后跟强制转换的“is”检查（样式规则 IDE0020 和 IDE0038）
- 探索：使用模式匹配生成类行为以获得更好的代码
- 教程：使用模式匹配来构建类型驱动和数据驱动的算法
- 引用：模式匹配

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

弃元 - C# 基础知识

项目 • 2023/11/14

弃元是一种在应用程序代码中人为取消使用的占位符变量。 弃元相当于未赋值的变量；它们没有值。 弃元将意图传达给编译器和其他读取代码的文件：你打算忽略表达式的结果。 你可能需要忽略表达式的结果、元组表达式的一个或多个成员、方法的 `out` 参数或模式匹配表达式的目标。

弃元使代码意图更加明确。 弃元指示代码永远不会使用变量。 它们可以增强其可读性和可维护性。

通过将下划线 (`_`) 赋给一个变量作为其变量名，指示该变量为一个占位符变量。 例如，以下方法调用返回一个元组，其中第一个值和第二个值为弃元。 `area` 是以前声明的变量，设置为由 `GetCityInformation` 返回的第三个组件：

```
C#

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

可以使用弃元来指定 Lambda 表达式中未使用的输入参数。 有关详细信息，请参阅 Lambda 表达式一文中的 Lambda 表达式的输入参数一节。

当 `_` 是有效弃元时，尝试检索其值或在赋值操作中使用它时，会生成编译器错误 CS0103：“当前上下文中不存在名称‘_’”。 出现此错误是因为 `_` 未赋值，甚至可能未分配存储位置。 如果它是一个实际变量，则不能像之前的示例那样对多个值使用弃元。

元组和对象析构

如果应用程序代码使用某些元组元素，但忽略其他元素，这时使用弃元来处理元组就会很有用。 例如，以下 `QueryCityDataForYears` 方法返回一个元组，包含城市名称、城市面积、一个年份、该年份的城市人口、另一个年份及该年份的城市人口。 该示例显示了两个年份之间人口的变化。 对于元组提供的数据，我们不关注城市面积，并在一开始就知道城市名称和两个日期。 因此，我们只关注存储在元组中的两个人口数量值，可将其余值作为占位符处理。

```
C#

var (_, _, _, pop1, _, pop2) = QueryCityDataForYears("New York City", 1960,
2010);

Console.WriteLine($"Population change, 1960 to 2010: {pop2 - pop1:N0}");
```

```
static (string, double, int, int, int, int) QueryCityDataForYears(string
name, int year1, int year2)
{
    int population1 = 0, population2 = 0;
    double area = 0;

    if (name == "New York City")
    {
        area = 468.48;
        if (year1 == 1960)
        {
            population1 = 7781984;
        }
        if (year2 == 2010)
        {
            population2 = 8175133;
        }
        return (name, area, year1, population1, year2, population2);
    }

    return ("", 0, 0, 0, 0, 0);
}
// The example displays the following output:
//      Population change, 1960 to 2010: 393,149
```

有关使用占位符析构元组的详细信息，请参阅 析构元组和其他类型。

类、结构或接口的 `Deconstruct` 方法还允许从对象中检索和析构一组特定的数据。 如果想只使用析构值的一个子集，可使用弃元。 以下示例将 `Person` 对象析构为四个字符串（名字、姓氏、城市和省/市/自治区），但舍弃姓氏和省/市/自治区。

C#

```
using System;

namespace Discards
{
    public class Person
    {
        public string FirstName { get; set; }
        public string MiddleName { get; set; }
        public string LastName { get; set; }
        public string City { get; set; }
        public string State { get; set; }

        public Person(string fname, string mname, string lname,
                      string cityName, string stateName)
        {
            FirstName = fname;
            MiddleName = mname;
            LastName = lname;
            City = cityName;
```

```
            State = stateName;
        }

        // Return the first and last name.
        public void Deconstruct(out string fname, out string lname)
        {
            fname = FirstName;
            lname = LastName;
        }

        public void Deconstruct(out string fname, out string mname, out
string lname)
        {
            fname = FirstName;
            mname = MiddleName;
            lname = LastName;
        }

        public void Deconstruct(out string fname, out string lname,
                                out string city, out string state)
        {
            fname = FirstName;
            lname = LastName;
            city = City;
            state = State;
        }
    }
    class Example
    {
        public static void Main()
        {
            var p = new Person("John", "Quincy", "Adams", "Boston", "MA");

            // Deconstruct the person object.
            var (fName, _, city, _) = p;
            Console.WriteLine($"Hello {fName} of {city}!");
            // The example displays the following output:
            //      Hello John of Boston!
        }
    }
}
```

有关使用弃元析构用户定义的类型的详细信息，请参阅 析构元组和其他类型。

利用 `switch` 的模式匹配

弃元模式可通过 switch 表达式用于模式匹配。 每个表达式（包括 `null`）都始终匹配弃元模式。

以下示例定义了一个 `ProvidesFormatInfo` 方法，它使用 `switch` 表达式来确定对象是否提供 IFormatProvider 实现并测试对象是否为 `null`。 它还使用占位符模式来处理任何其他

类型的非 null 对象。

C#

```
object?[] objects = [CultureInfo.CurrentCulture,
                     CultureInfo.CurrentCulture.DateTimeFormat,
                     CultureInfo.CurrentCulture.NumberFormat,
                     new ArgumentException(), null];
foreach (var obj in objects)
    ProvidesFormatInfo(obj);

static void ProvidesFormatInfo(object? obj) =>
    Console.WriteLine(obj switch
    {
        IFormatProvider fmt => $"{fmt.GetType()} object",
        null => "A null object reference: Its use could result in a
NullReferenceException",
        _ => "Some object type without format information"
    });
// The example displays the following output:
//    System.Globalization.CultureInfo object
//    System.Globalization.DateTimeFormatInfo object
//    System.Globalization.NumberFormatInfo object
//    Some object type without format information
//    A null object reference: Its use could result in a
NullReferenceException
```

对具有 `out` 参数的方法的调用

当调用 `Deconstruct` 方法来析构用户定义类型（类、结构或接口的实例）时，可使用占位符表示单个 `out` 参数的值。 但当使用 `out` 参数调用任何方法时，也可使用弃元表示 `out` 参数的值。

以下示例调用 DateTime.TryParse(String, out DateTime) 方法来确定日期的字符串表示形式在当前区域性中是否有效。 因为该示例侧重验证日期字符串，而不是解析它来提取日期，所以方法的 `out` 参数为占位符。

C#

```
string[] dateStrings = ["05/01/2018 14:57:32.8", "2018-05-01 14:57:32.8",
                        "2018-05-01T14:57:32.8375298-04:00", "5/01/2018",
                        "5/01/2018 14:57:32.80 -07:00",
                        "1 May 2018 2:57:32.8 PM", "16-05-2018 1:00:32 PM",
                        "Fri, 15 May 2018 20:10:57 GMT"];
foreach (string dateString in dateStrings)
{
    if (DateTime.TryParse(dateString, out _))
        Console.WriteLine($"'{dateString}': valid");
    else
```

```
        Console.WriteLine($"'{dateString}': invalid");
}
// The example displays output like the following:
//       '05/01/2018 14:57:32.8': valid
//       '2018-05-01 14:57:32.8': valid
//       '2018-05-01T14:57:32.8375298-04:00': valid
//       '5/01/2018': valid
//       '5/01/2018 14:57:32.80 -07:00': valid
//       '1 May 2018 2:57:32.8 PM': valid
//       '16-05-2018 1:00:32 PM': invalid
//       'Fri, 15 May 2018 20:10:57 GMT': invalid
```

独立弃元

可使用独立弃元来指示要忽略的任何变量。 一种典型的用法是使用赋值来确保一个参数不为 null。 下面的代码使用弃元来强制赋值。 赋值的右侧使用 Null 合并操作符，用于在参数为 `null` 时引发 System.ArgumentNullException。 此代码不需要赋值结果，因此将对其使用弃元。 该表达式强制执行 null 检查。 弃元说明你的意图：不需要或不使用赋值结果。

C#

```
public static void Method(string arg)
{
    _ = arg ?? throw new ArgumentNullException(paramName: nameof(arg),
message: "arg can't be null");

    // Do work with arg.
}
```

以下示例使用独立占位符来忽略异步操作返回的 Task 对象。 分配任务的效果等同于抑制操作即将完成时所引发的异常。 这使你的意图更加明确：你需要对 `Task` 使用弃元，并忽略该异步操作生成的任何错误。

C#

```
private static async Task ExecuteAsyncMethods()
{
    Console.WriteLine("About to launch a task...");
    _ = Task.Run(() =>
    {
        var iterations = 0;
        for (int ctr = 0; ctr < int.MaxValue; ctr++)
            iterations++;
        Console.WriteLine("Completed looping operation...");
        throw new InvalidOperationException();
    });
    await Task.Delay(5000);
```

```
    Console.WriteLine("Exiting after 5 second delay");
}
// The example displays output like the following:
//       About to launch a task...
//       Completed looping operation...
//       Exiting after 5 second delay
```

如果不将任务分配给弃元，则以下代码会生成编译器警告：

C#

```
private static async Task ExecuteAsyncMethods()
{
    Console.WriteLine("About to launch a task...");
    // CS4014: Because this call is not awaited, execution of the current
method continues before the call is completed.
    // Consider applying the 'await' operator to the result of the call.
    Task.Run(() =>
    {
        var iterations = 0;
        for (int ctr = 0; ctr < int.MaxValue; ctr++)
            iterations++;
        Console.WriteLine("Completed looping operation...");
        throw new InvalidOperationException();
    });
    await Task.Delay(5000);
    Console.WriteLine("Exiting after 5 second delay");
```

备注

如果使用调试器运行前面两个示例中的任意一个，则在引发异常时，调试器将停止该程序。 在没有附加调试器的情况下，这两种情况下的异常都会被以静默方式忽略。

`_` 也是有效标识符。 当在支持的上下文之外使用时，`_` 不视为占位符，而视为有效变量。 如果名为 `_` 的标识符已在范围内，则使用 `_` 作为独立占位符可能导致：

- 将预期的占位符的值赋给范围内 `_` 变量，会导致该变量的值被意外修改。 例如：

  C#

  ```
  private static void ShowValue(int _)
  {
     byte[] arr = [0, 0, 1, 2];
     _ = BitConverter.ToInt32(arr, 0);
     Console.WriteLine(_);
  }
  ```

```
// The example displays the following output:
//       33619968
```

- 因违反类型安全而发生的编译器错误。 例如：

```
C#

private static bool RoundTrips(int _)
{
   string value = _.ToString();
   int newValue = 0;
   _ = Int32.TryParse(value, out newValue);
   return _ == newValue;
}
// The example displays the following compiler error:
//      error CS0029: Cannot implicitly convert type 'bool' to 'int'
```

- 编译器错误 CS0136：“无法在此范围中声明名为 "_" 的局部变量或参数，因为该名称用于在封闭的局部范围中定义局部变量或参数”。例如：

```
C#

 public void DoSomething(int _)
{
 var _ = GetValue(); // Error: cannot declare local _ when one is
already in scope
}
// The example displays the following compiler error:
// error CS0136:
//       A local or parameter named '_' cannot be declared in this
scope
//       because that name is used in an enclosing local scope
//       to define a local or parameter
```

另请参阅

- 删除不必要的表达式值（样式规则 IDE0058）
- 删除不必要的赋值（样式规则 IDE0059）
- 删除未使用的参数（样式规则 IDE0060）
- 析构元组和其他类型
- is 运算符
- switch表达式

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

析构元组和其他类型

项目 • 2024/12/31

元组提供一种从方法调用中检索多个值的轻量级方法。 但是，一旦检索到元组，就必须处理它的各个元素。 按元素逐个操作比较麻烦，如下例所示。 `QueryCityData` 方法返回一个三元组，并通过单独的操作将其每个元素分配给一个变量。

C#

```
public class Example
{
    public static void Main()
    {
        var result = QueryCityData("New York City");

        var city = result.Item1;
        var pop = result.Item2;
        var size = result.Item3;

         // Do something with the data.
    }

    private static (string, int, double) QueryCityData(string name)
    {
        if (name == "New York City")
            return (name, 8175133, 468.48);

        return ("", 0, 0);
    }
}
```

从对象检索多个字段值和属性值可能同样麻烦：必须按成员逐个将字段值或属性值赋给一个变量。

可从元组中检索多个元素，或者在单个析构操作中从对象检索多个字段值、属性值和计算值。 若要析构元组，请将其元素分配给各个变量。 析构对象时，将选定值分配给各个变量。

元组

C# 提供内置的元组析构支持，可在单个操作中解包一个元组中的所有项。 用于析构元组的常规语法与用于定义元组的语法相似：将要向其分配元素的变量放在赋值语句左侧的括号中。 例如，以下语句将四元组的元素分配给 4 个单独的变量：

C#

```
var (name, address, city, zip) = contact.GetAddressInfo();
```

有三种方法可用于析构元组：

- 可以在括号内显式声明每个字段的类型。 以下示例使用此方法来析构由 `QueryCityData` 方法返回的三元组。

  C#

  ```
  public static void Main()
  {
      (string city, int population, double area) = QueryCityData("New
  York City");

      // Do something with the data.
  }
  ```

- 可使用 `var` 关键字，以便 C# 推断每个变量的类型。 将 `var` 关键字放在括号外。 以下示例在析构由 `QueryCityData` 方法返回的三元组时使用类型推理。

  C#

  ```
  public static void Main()
  {
      var (city, population, area) = QueryCityData("New York City");

      // Do something with the data.
  }
  ```

  还可在括号内将 `var` 关键字单独与任一或全部变量声明结合使用。

  C#

  ```
  public static void Main()
  {
      (string city, var population, var area) = QueryCityData("New York
  City");

      // Do something with the data.
  }
  ```

  前面的示例很繁琐，不建议这样做。

- 最后，可以将元组解构为已声明的变量。

  C#

```
public static void Main()
{
    string city = "Raleigh";
    int population = 458880;
    double area = 144.8;

    (city, population, area) = QueryCityData("New York City");

    // Do something with the data.
}
```

- 可以在解构中混合变量声明和赋值。

C#

```
public static void Main()
{
    string city = "Raleigh";
    int population = 458880;

    (city, population, double area) = QueryCityData("New York City");

    // Do something with the data.
}
```

即使元组中的每个字段都具有相同的类型，也不能在括号外使用特定类型。 这样做会生成编译器错误 CS8136：“析构 'var (...)' 形式不允许对 'var' 使用特定类型。”

必须将元组的每个元素分配给一个变量。 如果省略任何元素，编译器将生成错误 CS8132：“无法将 "x" 元素的元组析构为 "y" 变量”。

使用弃元的元组元素

析构元组时，通常只需要关注某些元素的值。 可利用 C# 对*弃元*的支持，弃元是一种仅能写入的变量，且其值将被忽略。 在赋值中使用下划线字符（“_”）声明一个弃元。 您可以丢弃任意数量的值，一个丢弃标记 `_` 表示所有已丢弃的值。

以下示例演示了对元组使用弃元时的用法。 `QueryCityDataForYears` 方法返回一个六元组，包含城市名称、城市面积、一个年份、该年份的城市人口、另一个年份及该年份的城市人口。 该示例显示了两个年份之间人口的变化。 对于元组提供的数据，我们不关注城市面积，并在一开始就知道城市名称和两个日期。 因此，我们只关注存储在元组中的两个人口数量值，可将其余值作为占位符处理。

C#

```
using System;

public class ExampleDiscard
{
    public static void Main()
    {
        var (_, _, _, pop1, _, pop2) = QueryCityDataForYears("New York
City", 1960, 2010);

        Console.WriteLine($"Population change, 1960 to 2010: {pop2 -
pop1:N0}");
    }

    private static (string, double, int, int, int, int)
QueryCityDataForYears(string name, int year1, int year2)
    {
        int population1 = 0, population2 = 0;
        double area = 0;

        if (name == "New York City")
        {
            area = 468.48;
            if (year1 == 1960)
            {
                population1 = 7781984;
            }
            if (year2 == 2010)
            {
                population2 = 8175133;
            }
            return (name, area, year1, population1, year2, population2);
        }

        return ("", 0, 0, 0, 0, 0);
    }
}
// The example displays the following output:
//      Population change, 1960 to 2010: 393,149
```

用户定义类型

C# 提供对解构元组类型、record 和 DictionaryEntry 类型的内置支持。 但是，用户作为类、结构或接口的创建者，可通过实现一个或多个 `Deconstruct` 方法来析构该类型的实例。 该方法返回“void”。 方法签名中的 out 参数表示要解构的每个值。 例如，`Person` 类的以下 `Deconstruct` 方法返回名字、中间名和姓氏：

```
C#
```

```
public void Deconstruct(out string fname, out string mname, out string
lname)
```

然后，可使用与下列代码类似的赋值来析构名为 `Person` 的 `p` 类的实例：

C#

```
var (fName, mName, lName) = p;
```

以下示例重载 `Deconstruct` 方法以返回 `Person` 对象的各种属性组合。 单个重载返回：

- 名字和姓氏。
- 名字、中间名和姓氏。
- 名字、姓氏、城市名称和州名称。

C#

```
using System;

public class Person
{
    public string FirstName { get; set; }
    public string MiddleName { get; set; }
    public string LastName { get; set; }
    public string City { get; set; }
    public string State { get; set; }

    public Person(string fname, string mname, string lname,
                  string cityName, string stateName)
    {
        FirstName = fname;
        MiddleName = mname;
        LastName = lname;
        City = cityName;
        State = stateName;
    }

    // Return the first and last name.
    public void Deconstruct(out string fname, out string lname)
    {
        fname = FirstName;
        lname = LastName;
    }

    public void Deconstruct(out string fname, out string mname, out string
lname)
    {
        fname = FirstName;
        mname = MiddleName;
        lname = LastName;
```

```
    }

    public void Deconstruct(out string fname, out string lname,
                            out string city, out string state)
    {
        fname = FirstName;
        lname = LastName;
        city = City;
        state = State;
    }
}

public class ExampleClassDeconstruction
{
    public static void Main()
    {
        var p = new Person("John", "Quincy", "Adams", "Boston", "MA");

        // Deconstruct the person object.
        var (fName, lName, city, state) = p;
        Console.WriteLine($"Hello {fName} {lName} of {city}, {state}!");
    }
}
// The example displays the following output:
//    Hello John Adams of Boston, MA!
```

具有相同数量参数的多个 `Deconstruct` 方法是不明确的。 在定义 `Deconstruct` 方法时，必须小心使用不同数量的参数或“arity”。 在重载解析过程中，不能区分具有相同数量参数的 `Deconstruct` 方法。

使用弃元的用户定义类型

就像使用元组一样，可使用弃元来忽略 `Deconstruct` 方法返回的选定项。 名为“_”的变量表示放弃。 单个解构操作可以包含多个弃元。

以下示例将 `Person` 对象解构为四个字符串（名字、姓氏、城市和州），但丢弃了姓氏和州。

C#

```
// Deconstruct the person object.
var (fName, _, city, _) = p;
Console.WriteLine($"Hello {fName} of {city}!");
// The example displays the following output:
//      Hello John of Boston!
```

解构扩展方法

如果没有创建类、结构或接口，仍可通过实现一个或多个 `Deconstruct` 扩展方法来析构该类型的对象，以返回所需值。

以下示例为 `Deconstruct` 类定义了两个 System.Reflection.PropertyInfo 扩展方法。 第一个返回一组值，该值指示属性的特征。 第二个方法指示属性的可访问性。 布尔值指示属性是否具有单独的 get 和 set 访问器或不同的辅助功能。 如果只有一个访问器，或者 get 和 set 访问器具有相同的可访问性，则 `access` 变量指示整个属性的可访问性。 否则，get 和 set 访问器的可访问性由 `getAccess` 和 `setAccess` 变量指示。

C#

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

public static class ReflectionExtensions
{
    public static void Deconstruct(this PropertyInfo p, out bool isStatic,
                                   out bool isReadOnly, out bool isIndexed,
                                   out Type propertyType)
    {
        var getter = p.GetMethod;

        // Is the property read-only?
        isReadOnly = ! p.CanWrite;

        // Is the property instance or static?
        isStatic = getter.IsStatic;

        // Is the property indexed?
        isIndexed = p.GetIndexParameters().Length > 0;

        // Get the property type.
        propertyType = p.PropertyType;
    }

    public static void Deconstruct(this PropertyInfo p, out bool
hasGetAndSet,
                                   out bool sameAccess, out string access,
                                   out string getAccess, out string
setAccess)
    {
        hasGetAndSet = sameAccess = false;
        string getAccessTemp = null;
        string setAccessTemp = null;

        MethodInfo getter = null;
        if (p.CanRead)
            getter = p.GetMethod;

        MethodInfo setter = null;
        if (p.CanWrite)
```

```
            setter = p.SetMethod;

        if (setter != null && getter != null)
            hasGetAndSet = true;

        if (getter != null)
        {
            if (getter.IsPublic)
                getAccessTemp = "public";
            else if (getter.IsPrivate)
                getAccessTemp = "private";
            else if (getter.IsAssembly)
                getAccessTemp = "internal";
            else if (getter.IsFamily)
                getAccessTemp = "protected";
            else if (getter.IsFamilyOrAssembly)
                getAccessTemp = "protected internal";
        }

        if (setter != null)
        {
            if (setter.IsPublic)
                setAccessTemp = "public";
            else if (setter.IsPrivate)
                setAccessTemp = "private";
            else if (setter.IsAssembly)
                setAccessTemp = "internal";
            else if (setter.IsFamily)
                setAccessTemp = "protected";
            else if (setter.IsFamilyOrAssembly)
                setAccessTemp = "protected internal";
        }

        // Are the accessibility of the getter and setter the same?
        if (setAccessTemp == getAccessTemp)
        {
            sameAccess = true;
            access = getAccessTemp;
            getAccess = setAccess = String.Empty;
        }
        else
        {
            access = null;
            getAccess = getAccessTemp;
            setAccess = setAccessTemp;
        }
    }
}

public class ExampleExtension
{
    public static void Main()
    {
        Type dateType = typeof(DateTime);
        PropertyInfo prop = dateType.GetProperty("Now");
```

```
        var (isStatic, isRO, isIndexed, propType) = prop;
        Console.WriteLine($"\nThe {dateType.FullName}.{prop.Name}
property:");
        Console.WriteLine($"   PropertyType: {propType.Name}");
        Console.WriteLine($"   Static:       {isStatic}");
        Console.WriteLine($"   Read-only:    {isRO}");
        Console.WriteLine($"   Indexed:      {isIndexed}");

        Type listType = typeof(List<>);
        prop = listType.GetProperty("Item",
                                    BindingFlags.Public |
BindingFlags.NonPublic | BindingFlags.Instance | BindingFlags.Static);
        var (hasGetAndSet, sameAccess, accessibility, getAccessibility,
setAccessibility) = prop;
        Console.Write($"\nAccessibility of the {listType.FullName}.
{prop.Name} property: ");

        if (!hasGetAndSet | sameAccess)
        {
            Console.WriteLine(accessibility);
        }
        else
        {
            Console.WriteLine($"\n   The get accessor: {getAccessibility}");
            Console.WriteLine($"   The set accessor: {setAccessibility}");
        }
    }
}
// The example displays the following output:
//       The System.DateTime.Now property:
//          PropertyType: DateTime
//          Static:       True
//          Read-only:    True
//          Indexed:      False
//
//       Accessibility of the System.Collections.Generic.List`1.Item
property: public
```

系统类型的扩展方法

为了方便起见，某些系统类型提供 `Deconstruct` 方法。 例如，System.Collections.Generic.KeyValuePair<TKey,TValue> 类型提供此功能。 循环访问 System.Collections.Generic.Dictionary<TKey,TValue> 时，每个元素都是 `KeyValuePair<TKey, TValue>`，并且可以析构。 请考虑以下示例：

C#

```
Dictionary<string, int> snapshotCommitMap =
new(StringComparer.OrdinalIgnoreCase)
{
```

```
    ["https://github.com/dotnet/docs"] = 16_465,
    ["https://github.com/dotnet/runtime"] = 114_223,
    ["https://github.com/dotnet/installer"] = 22_436,
    ["https://github.com/dotnet/roslyn"] = 79_484,
    ["https://github.com/dotnet/aspnetcore"] = 48_386
};

foreach (var (repo, commitCount) in snapshotCommitMap)
{
    Console.WriteLine(
        $"The {repo} repository had {commitCount:N0} commits as of November
10th, 2021.");
}
```

`record` 类型

使用两个或多个位置参数声明记录类型时，编译器将为 `Deconstruct` 声明中的每个位置参数创建一个带有 `out` 参数的 `record` 方法。 有关详细信息，请参阅属性定义的位置语法和派生记录中的解构函数行为。

请参阅

- 析构变量声明（样式规则 IDE0042）
- 弃元
- 元组类型

异常和异常处理

项目 • 2024/04/11

C# 语言的异常处理功能有助于处理在程序运行期间发生的任何意外或异常情况。 异常处理功能使用 `try`、`catch` 和 `finally` 关键字来尝试执行可能失败的操作、在你确定合理的情况下处理故障，以及在事后清除资源。 公共语言运行时 (CLR)、.NET/第三方库或应用程序代码都可生成异常。 异常是使用 `throw` 关键字创建而成。

在许多情况下，异常并不是由代码直接调用的方法抛出，而是由调用堆栈中再往下的另一方法抛出。 如果发生这种异常，CLR 会展开堆栈，同时针对特定异常类型查找包含 `catch` 代码块的方法，并执行它找到的首个此类 `catch` 代码块。 如果在调用堆栈中找不到相应的 `catch` 代码块，将会终止进程并向用户显示消息。

在以下示例中，方法用于测试除数是否为零，并捕获相应的错误。 如果没有异常处理功能，此程序将终止，并显示 **DivideByZeroException was unhandled** 错误。

C#

```
public class ExceptionTest
{
    static double SafeDivision(double x, double y)
    {
        if (y == 0)
            throw new DivideByZeroException();
        return x / y;
    }

    public static void Main()
    {
        // Input for test purposes. Change the values to see
        // exception handling behavior.
        double a = 98, b = 0;
        double result;

        try
        {
            result = SafeDivision(a, b);
            Console.WriteLine("{0} divided by {1} = {2}", a, b, result);
        }
        catch (DivideByZeroException)
        {
            Console.WriteLine("Attempted divide by zero.");
        }
    }
}
```

异常概述

异常具有以下属性：

- 异常是最终全都派生自 `System.Exception` 的类型。
- 在可能抛出异常的语句周围使用 `try` 代码块。
- 在 `try` 代码块中出现异常后，控制流会跳转到调用堆栈中任意位置上的首个相关异常处理程序。 在 C# 中，`catch` 关键字用于定义异常处理程序。
- 如果给定的异常没有对应的异常处理程序，那么程序会停止执行，并显示错误消息。
- 除非可以处理异常并让应用程序一直处于已知状态，否则不捕获异常。 如果捕获 `System.Exception`，使用 `catch` 代码块末尾的 `throw` 关键字重新抛出异常。
- 如果 `catch` 代码块定义异常变量，可以用它来详细了解所发生的异常类型。
- 使用 `throw` 关键字，程序可以显式生成异常。
- 异常对象包含错误详细信息，如调用堆栈的状态和错误的文本说明。
- 即使引发异常，`finally` 代码块中的代码仍会执行。 使用 `finally` 代码块可释放资源。例如，关闭在 `try` 代码块中打开的任何流或文件。
- .NET 中的托管异常在 Win32 结构化异常处理机制的基础之上实现。 有关详细信息，请参阅结构化异常处理 (C/C++) 和速成教程：深入了解 Win32 结构化异常处理。

C# 语言规范

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

请参阅

- SystemException
- C# 关键字
- throw
- try-catch
- try-finally
- try-catch-finally
- 异常

反馈

此页面是否有帮助? 是 否

提供产品反馈

使用异常

项目 • 2023/05/10

在 C# 中，程序中的运行时错误通过使用一种称为“异常”的机制在程序中传播。 异常由遇到错误的代码引发，由能够更正错误的代码捕捉。 异常可由 .NET 运行时或由程序中的代码引发。 一旦引发了一个异常，此异常会在调用堆栈中传播，直到找到针对它的 `catch` 语句。 未捕获的异常由系统提供的通用异常处理程序处理，该处理程序会显示一个对话框。

异常由从 Exception 派生的类表示。 此类标识异常的类型，并包含详细描述异常的属性。 引发异常涉及创建异常派生类的实例，配置异常的属性（可选），然后使用 `throw` 关键字引发该对象。 例如：

C#

```
class CustomException : Exception
{
    public CustomException(string message)
    {
    }
}
private static void TestThrow()
{
    throw new CustomException("Custom exception in TestThrow()");
}
```

引发异常后，运行时将检查当前语句，以确定它是否在 `try` 块内。 如果在，则将检查与 `try` 块关联的所有 `catch` 块，以确定它们是否可以捕获该异常。 `Catch` 块通常会指定异常类型；如果该 `catch` 块的类型与异常或异常的基类的类型相同，则该 `catch` 块可处理该方法。 例如：

C#

```
try
{
    TestThrow();
}
catch (CustomException ex)
{
    System.Console.WriteLine(ex.ToString());
}
```

如果引发异常的语句不在 `try` 块内或者包含该语句的 `try` 块没有匹配的 `catch` 块，则运行时将检查调用方法中是否有 `try` 语句和 `catch` 块。 运行时将继续调用堆栈，搜索兼容

的 `catch` 块。 在找到并执行 `catch` 块之后，控制权将传递给 `catch` 块之后的下一个语句。

一个 `try` 语句可包含多个 `catch` 块。 将执行第一个能够处理该异常的 `catch` 语句；将忽略任何后续的 `catch` 语句，即使它们是兼容的也是如此。 按从最具有针对性（或派生程度最高）到最不具有针对性的顺序对 catch 块排列。 例如：

C#

```
using System;
using System.IO;

namespace Exceptions
{
    public class CatchOrder
    {
        public static void Main()
        {
            try
            {
                using (var sw = new StreamWriter("./test.txt"))
                {
                    sw.WriteLine("Hello");
                }
            }
            // Put the more specific exceptions first.
            catch (DirectoryNotFoundException ex)
            {
                Console.WriteLine(ex);
            }
            catch (FileNotFoundException ex)
            {
                Console.WriteLine(ex);
            }
            // Put the least specific exception last.
            catch (IOException ex)
            {
                Console.WriteLine(ex);
            }
            Console.WriteLine("Done");
        }
    }
}
```

执行 `catch` 块之前，运行时会检查 `finally` 块。 `Finally` 块使程序员可以清除中止的 `try` 块可能遗留下的任何模糊状态，或者释放任何外部资源（例如图形句柄、数据库连接或文件流），而无需等待垃圾回收器在运行时完成这些对象。 例如：

C#

```
static void TestFinally()
{
    FileStream? file = null;
    //Change the path to something that works on your machine.
    FileInfo fileInfo = new System.IO.FileInfo("./file.txt");

    try
    {
        file = fileInfo.OpenWrite();
        file.WriteByte(0xF);
    }
    finally
    {
        // Closing the file allows you to reopen it immediately - otherwise
IOException is thrown.
        file?.Close();
    }

    try
    {
        file = fileInfo.OpenWrite();
        Console.WriteLine("OpenWrite() succeeded");
    }
    catch (IOException)
    {
        Console.WriteLine("OpenWrite() failed");
    }
}
```

如果 `WriteByte()` 引发了异常并且未调用 `file.Close()`，则第二个 `try` 块中尝试重新打开文件的代码将会失败，并且文件将保持锁定状态。 由于即使引发异常也会执行 `finally` 块，前一示例中的 `finally` 块可使文件正确关闭，从而有助于避免错误。

如果引发异常之后没有在调用堆栈上找到兼容的 `catch` 块，则会出现以下三种情况之一：

- 如果异常存在于终结器内，将中止终结器，并调用基类终结器（如果有）。
- 如果调用堆栈包含静态构造函数或静态字段初始值设定项，将引发 TypeInitializationException，同时将原始异常分配给新异常的 InnerException 属性。
- 如果到达线程的开头，则终止线程。

异常处理（C# 编程指南）

项目 • 2024/03/04

C# 程序员使用 try 块来对可能受异常影响的代码进行分区。 关联的 catch 块用于处理生成的任何异常。 finally 块包含无论 `try` 块中是否引发异常都会运行的代码，如发布 `try` 块中分配的资源。 `try` 块需要一个或多个关联的 `catch` 块或一个 `finally` 块，或两者皆之。

下面的示例演示 `try-catch` 语句、 `try-finally` 语句和 `try-catch-finally` 语句。

C#

```
try
{
    // Code to try goes here.
}
catch (SomeSpecificException ex)
{
    // Code to handle the exception goes here.
    // Only catch exceptions that you know how to handle.
    // Never catch base class System.Exception without
    // rethrowing it at the end of the catch block.
}
```

C#

```
try
{
    // Code to try goes here.
}
finally
{
    // Code to execute after the try block goes here.
}
```

C#

```
try
{
    // Code to try goes here.
}
catch (SomeSpecificException ex)
{
    // Code to handle the exception goes here.
}
finally
{
```

```
    // Code to execute after the try (and possibly catch) blocks
    // goes here.
}
```

一个不具有 `catch` 或 `finally` 块的 `try` 块会导致编译器错误。

catch 块

`catch` 块可以指定要捕获的异常的类型。 该类型规范称为异常筛选器。 异常类型应派生自 Exception。 一般情况下，不要将 Exception 指定为异常筛选器，除非了解如何处理可能在 `try` 块中引发的所有异常，或者已在 `catch` 块的末尾处包括了 throw 语句。

可将具有不同异常类的多个 `catch` 块链接在一起。 代码中 `catch` 块的计算顺序为从上到下，但针对引发的每个异常，仅执行一个 `catch` 块。 将执行指定所引发的异常的确切类型或基类的第一个 `catch` 块。 如果没有 `catch` 块指定匹配的异常类，则将选择不具有类型的 `catch` 块（如果语句中存在）。 务必首先定位具有最具体的（即，最底层派生的）异常类的 `catch` 块。

当以下条件为 true 时，捕获异常：

- 能够很好地理解可能会引发异常的原因，并且可以实现特定的恢复，例如捕获 FileNotFoundException 对象时提示用户输入新文件名。
- 可以创建和引发一个新的、更具体的异常。

  C#

  ```
  int GetInt(int[] array, int index)
  {
      try
      {
          return array[index];
      }
      catch (IndexOutOfRangeException e)
      {
          throw new ArgumentOutOfRangeException(
              "Parameter index is out of range.", e);
      }
  }
  ```

- 想要先对异常进行部分处理，然后再将其传递以进行更多处理。 在下面的示例中，`catch` 块用于在重新引发异常之前将条目添加到错误日志。

  C#

  ```
  try
  {
  ```

```
        // Try to access a resource.
    }
    catch (UnauthorizedAccessException e)
    {
        // Call a custom error logging procedure.
        LogError(e);
        // Re-throw the error.
        throw;
    }
```

还可以指定异常筛选器，以向 catch 子句添加布尔表达式。 异常筛选器表明仅当条件为 true 时，特定 catch 子句才匹配。 在以下示例中，两个 catch 子句均使用相同的异常类，但是会检查其他条件以创建不同的错误消息：

C#

```
int GetInt(int[] array, int index)
{
    try
    {
        return array[index];
    }
    catch (IndexOutOfRangeException e) when (index < 0)
    {
        throw new ArgumentOutOfRangeException(
            "Parameter index cannot be negative.", e);
    }
    catch (IndexOutOfRangeException e)
    {
        throw new ArgumentOutOfRangeException(
            "Parameter index cannot be greater than the array size.", e);
    }
}
```

始终返回 `false` 的异常筛选器可用于检查所有异常，但不可用于处理异常。 典型用途是记录异常：

C#

```
public class ExceptionFilter
{
    public static void Main()
    {
        try
        {
            string? s = null;
            Console.WriteLine(s.Length);
        }
        catch (Exception e) when (LogException(e))
        {
```

```
        }
        Console.WriteLine("Exception must have been handled");
    }

    private static bool LogException(Exception e)
    {
        Console.WriteLine($"\tIn the log routine. Caught {e.GetType()}");
        Console.WriteLine($"\tMessage: {e.Message}");
        return false;
    }
}
```

`LogException` 方法始终返回 `false`，使用此异常筛选器的 `catch` 子句均不匹配。 catch 子句可以是通用的，使用 System.Exception后面的子句可以处理更具体的异常类。

Finally 块

`finally` 块让你可以清理在 `try` 块中所执行的操作。 如果存在 `finally` 块，将在执行 `try` 块和任何匹配的 `catch` 块之后，最后执行它。 无论是否会引发异常或找到匹配异常类型的 `catch` 块， `finally` 块都将始终运行。

`finally` 块可用于发布资源（如文件流、数据库连接和图形句柄）而无需等待运行时中的垃圾回收器来完成对象。

在下面的示例中， `finally` 块用于关闭在 `try` 块中打开的文件。 请注意，在关闭文件之前，将检查文件句柄的状态。 如果 `try` 块不能打开文件，则文件句柄仍将具有值 `null` 且 `finally` 块不会尝试将其关闭。 或者，如果在 `try` 块中成功打开文件，则 `finally` 块将关闭打开的文件。

C#

```
FileStream? file = null;
FileInfo fileinfo = new System.IO.FileInfo("./file.txt");
try
{
    file = fileinfo.OpenWrite();
    file.WriteByte(0xF);
}
finally
{
    // Check for null because OpenWrite might have failed.
    file?.Close();
}
```

C# 语言规范

有关详细信息，请参阅 C# 语言规范中的异常和 try 语句。 该语言规范是 C# 语法和用法的权威资料。

另请参阅

- C# 参考
- 异常处理语句
- using 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 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

创建和引发异常

项目 • 2023/12/22

异常用于指示在运行程序时发生了错误。 此时将创建一个描述错误的异常对象，然后使用 throw 语句或表达式*引发*。 然后，运行时搜索最兼容的异常处理程序。

当存在下列一种或多种情况时，程序员应引发异常：

- 方法无法完成其定义的功能。 例如，如果一种方法的参数具有无效的值：

  C#

  ```
  static void CopyObject(SampleClass original)
  {
      _ = original ?? throw new ArgumentException("Parameter cannot be
  null", nameof(original));
  }
  ```

- 根据对象的状态，对某个对象进行不适当的调用。 一个示例可能是尝试写入只读文件。 在对象状态不允许操作的情况下，引发 InvalidOperationException 的实例或基于此类的派生的对象。 以下代码是引发 InvalidOperationException 对象的方法示例：

  C#

  ```
  public class ProgramLog
  {
      FileStream logFile = null!;
      public void OpenLog(FileInfo fileName, FileMode mode) { }

      public void WriteLog()
      {
          if (!logFile.CanWrite)
          {
              throw new InvalidOperationException("Logfile cannot be
  read-only");
          }
          // Else write data to the log and return.
      }
  }
  ```

- 方法的参数引发了异常。 在这种情况下，应捕获原始异常，并创建 ArgumentException 实例。 应将原始异常作为 InnerException 参数传递给 ArgumentException 的构造函数：

  C#

```
static int GetValueFromArray(int[] array, int index)
{
    try
    {
        return array[index];
    }
    catch (IndexOutOfRangeException e)
    {
        throw new ArgumentOutOfRangeException(
            "Parameter index is out of range.", e);
    }
}
```

> **备注**
>
> 前面的示例演示了如何使用 `InnerException` 属性。 这是有意简化的。 在实践中，应先检查索引是否在范围内，然后再使用它。 当参数成员引发在调用成员之前无法预料到的异常时，可以使用此方法来包装异常。

异常包含一个名为 StackTrace 的属性。 此字符串包含当前调用堆栈上的方法的名称，以及为每个方法引发异常的位置（文件名和行号）。 StackTrace 对象由公共语言运行时 (CLR) 从 `throw` 语句的位置点自动创建，因此必须从堆栈跟踪的开始点引发异常。

所有异常都包含一个名为 Message 的属性。 应设置此字符串来解释发生异常的原因。 不应将安全敏感的信息放在消息文本中。 除 Message 以外，ArgumentException 也包含一个名为 ParamName 的属性，应将该属性设置为导致引发异常的参数的名称。 在属性资源库中，ParamName 应设置为 `value`。

公共的受保护方法在无法完成其预期功能时将引发异常。 引发的异常类是符合错误条件的最具体的可用异常。 这些异常应编写为类功能的一部分，并且原始类的派生类或更新应保留相同的行为以实现后向兼容性。

引发异常时应避免的情况

以下列表标识了引发异常时要避免的做法：

- 不要使用异常在正常执行过程中更改程序的流。 使用异常来报告和处理错误条件。
- 只能引发异常，而不能作为返回值或参数返回异常。
- 请勿有意从自己的源代码中引发 System.Exception、System.SystemException、System.NullReferenceException 或 System.IndexOutOfRangeException。
- 不要创建可在调试模式下引发，但不会在发布模式下引发的异常。 若要在开发阶段确定运行时错误，请改用调试断言。

任务返回方法中的异常

使用 `async` 修饰符声明的方法在出现异常时，有一些特殊的注意事项。 方法 `async` 中引发的异常会存储在返回的任务中，直到任务即将出现时才会出现。 有关存储的异常的详细信息，请参阅异步异常。

建议在输入方法的异步部分之前验证参数并引发任何相应的异常，例如 ArgumentException 和 ArgumentNullException。 也就是说，在开始工作之前，这些验证异常应同步出现。 以下代码片段演示了一个示例，其中，如果引发异常，ArgumentException 个异常将同步出现，而 InvalidOperationException 个将存储在返回的任务中。

C#

```
// Non-async, task-returning method.
// Within this method (but outside of the local function),
// any thrown exceptions emerge synchronously.
public static Task<Toast> ToastBreadAsync(int slices, int toastTime)
{
    if (slices is < 1 or > 4)
    {
        throw new ArgumentException(
            "You must specify between 1 and 4 slices of bread.",
            nameof(slices));
    }

    if (toastTime < 1)
    {
        throw new ArgumentException(
            "Toast time is too short.", nameof(toastTime));
    }

    return ToastBreadAsyncCore(slices, toastTime);

    // Local async function.
    // Within this function, any thrown exceptions are stored in the task.
    static async Task<Toast> ToastBreadAsyncCore(int slices, int time)
    {
        for (int slice = 0; slice < slices; slice++)
        {
            Console.WriteLine("Putting a slice of bread in the toaster");
        }
        // Start toasting.
        await Task.Delay(time);

        if (time > 2_000)
        {
            throw new InvalidOperationException("The toaster is on fire!");
        }

        Console.WriteLine("Toast is ready!");
```

```
        return new Toast();
    }
}
```

定义异常的类别

程序可以引发 System 命名空间中的预定义异常类（前面提到的情况除外），或通过从 Exception 派生来创建其自己的异常类。 派生类应该至少定义三个构造函数：一个无参数构造函数、一个用于设置消息属性，还有一个用于设置 Message 和 InnerException 属性。 例如：

C#

```
[Serializable]
public class InvalidDepartmentException : Exception
{
    public InvalidDepartmentException() : base() { }
    public InvalidDepartmentException(string message) : base(message) { }
    public InvalidDepartmentException(string message, Exception inner) :
base(message, inner) { }
}
```

当新属性提供的数据有助于解决异常时，将新属性添加到异常类中。 如果将新属性添加到派生异常类中，则应替代 `ToString()` 以返回添加的信息。

C# 语言规范

有关详细信息，请参阅 C# 语言规范中的异常和 throw 语句。 该语言规范是 C# 语法和用法的权威资料。

另请参阅

- 异常层次结构

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

编译器生成的异常

项目 • 2023/06/05

当基本操作失败时，.NET 运行时会自动引发一些异常。 这些异常及其错误条件在下表中列出。

例外	描述
ArithmeticException	算术运算期间出现的异常的基类，例如 DivideByZeroException 和 OverflowException。
ArrayTypeMismatchException	由于元素的实际类型与数组的实际类型不兼容而导致数组无法存储给定元素时引发。
DivideByZeroException	尝试将整数值除以零时引发。
IndexOutOfRangeException	索引小于零或超出数组边界时，尝试对数组编制索引时引发。
InvalidCastException	从基类型显式转换为接口或派生类型在运行时失败时引发。
NullReferenceException	尝试引用值为 null 的对象时引发。
OutOfMemoryException	尝试使用新运算符分配内存失败时引发。 此异常表示可用于公共语言运行时的内存已用尽。
OverflowException	`checked` 上下文中的算术运算溢出时引发。
StackOverflowException	执行堆栈由于有过多挂起的方法调用而用尽时引发；通常表示非常深的递归或无限递归。
TypeInitializationException	静态构造函数引发异常并且没有兼容的 `catch` 子句来捕获异常时引发。

请参阅

- 异常处理语句

C# 标识符命名规则和约定

项目 • 2024/03/19

标识符是分配给类型（类、接口、结构、委托或枚举）、成员、变量或命名空间的名称。

命名规则

有效标识符必须遵循以下规则。 C# 编译器针对不遵循以下规则的任何标识符生成错误：

- 标识符必须以字母或下划线 (`_`) 开头。
- 标识符可以包含 Unicode 字母字符、十进制数字字符、Unicode 连接字符、Unicode 组合字符或 Unicode 格式字符。 有关 Unicode 类别的详细信息，请参阅 Unicode 类别数据库。

可以在标识符上使用 `@` 前缀来声明与 C# 关键字匹配的标识符。 `@` 不是标识符名称的一部分。 例如，`@if` 声明名为 `if` 的标识符。 这些逐字标识符主要用于与使用其他语言声明的标识符的互操作性。

有关有效标识符的完整定义，请参阅 C# 语言规范中的标识符一文。

> **重要**
>
> C# 语言规范仅允许字母（Lu、Ll、Lt、Lm、Lo 或 Nl）、数字 (Nd)、连接 (Pc)、组合 (Mn 或 Mc) 和格式 (Cf) 类别。 除此之外的任何内容都会自动使用 `_` 替换。 这可能会影响某些 Unicode 字符。

命名约定

除了规则之外，标识符名称的约定也在整个 .NET API 中使用。 这些约定为名称提供一致性，但编译器不会强制执行它们。 可以在项目中使用不同的约定。

按照约定，C# 程序对类型名称、命名空间和所有公共成员使用 `PascalCase`。 此外，`dotnet/docs` 团队使用从 .NET Runtime 团队的编码风格中吸收的以下约定：

- 接口名称以大写字母 `I` 开头。
- 属性类型以单词 `Attribute` 结尾。
- 枚举类型对非标记使用单数名词，对标记使用复数名词。

- 标识符不应包含两个连续的下划线 (`_`) 字符。 这些名称保留给编译器生成的标识符。
- 对变量、方法和类使用有意义的描述性名称。
- 清晰胜于简洁。。
- 将 PascalCase 用于类名和方法名称。
- 对方法参数和局部变量使用驼峰式大小写。
- 将 PascalCase 用于常量名，包括字段和局部常量。
- 专用实例字段以下划线 (`_`) 开头，其余文本为驼峰式大小写。
- 静态字段以 `s_` 开头。 此约定不是默认的 Visual Studio 行为，也不是框架设计准则的一部分，但在 editorconfig 中可配置。
- 避免在名称中使用缩写或首字母缩略词，但广为人知和广泛接受的缩写除外。
- 使用遵循反向域名表示法的有意义的描述性命名空间。
- 选择表示程序集主要用途的程序集名称。
- 避免使用单字母名称，但简单循环计数器除外。 此外，描述 C# 构造的语法示例通常使用与 C# 语言规范中使用的约定相匹配的以下单字母名称。 语法示例是规则的例外。
  - 将 `S` 用于结构，将 `C` 用于类。
  - 将 `M` 用于方法。
  - 将 `v` 用于变量，将 `p` 用于参数。
  - 将 `r` 用于 `ref` 参数。

> **提示**
>
> 可以使用**代码样式命名规则**强制实施涉及大写、前缀、后缀和单词分隔符的命名约定。

在下面的示例中，与标记为 `public` 的元素相关的指导也适用于使用 `protected` 和 `protected internal` 元素的情况 - 所有这些元素旨在对外部调用方可见。

Pascal 大小写

在命名 `class`、`interface`、`struct` 或 `delegate` 类型时，使用 Pascal 大小写（“PascalCasing”）。

C#

```
public class DataService
{
}
```

C#

```
public record PhysicalAddress(
    string Street,
    string City,
    string StateOrProvince,
    string ZipCode);
```

C#

```
public struct ValueCoordinate
{
}
```

C#

```
public delegate void DelegateType(string message);
```

命名 `interface` 时，使用 pascal 大小写并在名称前面加上前缀 `I`。 此前缀可以清楚地向使用者表明这是 `interface`。

C#

```
public interface IWorkerQueue
{
}
```

在命名字段、属性和事件等类型的 `public` 成员时，使用 pascal 大小写。 此外，对所有方法和本地函数使用 pascal 大小写。

C#

```
public class ExampleEvents
{
    // A public field, these should be used sparingly
    public bool IsValid;

    // An init-only property
    public IWorkerQueue WorkerQueue { get; init; }
```

```
    // An event
    public event Action EventProcessing;

    // Method
    public void StartEventProcessing()
    {
        // Local function
        static int CountQueueItems() => WorkerQueue.Count;
        // ...
    }
}
```

编写位置记录时，对参数使用 pascal 大小写，因为它们是记录的公共属性。

C#

```
public record PhysicalAddress(
    string Street,
    string City,
    string StateOrProvince,
    string ZipCode);
```

有关位置记录的详细信息，请参阅属性定义的位置语法。

驼峰式大小写

在命名 `private` 或 `internal` 字段时，使用驼峰式大小写（“camelCasing”），并对它们添加 `_` 作为前缀。 命名局部变量（包括委托类型的实例）时，请使用驼峰式大小写。

C#

```
public class DataService
{
    private IWorkerQueue _workerQueue;
}
```

提示

在支持语句完成的 IDE 中编辑遵循这些命名约定的 C# 代码时，键入 `_` 将显示所有对象范围的成员。

使用为 `private` 或 `internal` 的`static` 字段时 请使用 `s_` 前缀，对于线程静态，请使用 `t_`。

C#

```
public class DataService
{
    private static IWorkerQueue s_workerQueue;

    [ThreadStatic]
    private static TimeSpan t_timeSpan;
}
```

编写方法参数时，请使用驼峰式大小写。

C#

```
public T SomeMethod<T>(int someNumber, bool isValid)
{
}
```

有关 C# 命名约定的详细信息，请参阅 .NET Runtime 团队的编码样式。

类型参数命名指南

以下准则适用于泛型类型参数上的类型参数。 类型参数是泛型类型或泛型方法中参数的占位符。 可以在 C# 编程指南中详细了解 泛型类型参数。

- 请使用描述性名称命名泛型类型参数，除非单个字母名称完全具有自我说明性且描述性名称不会增加任何作用。

  ./snippets/coding-conventions

  ```
  public interface ISessionChannel<TSession> { /*...*/ }
  public delegate TOutput Converter<TInput, TOutput>(TInput from);
  public class List<T> { /*...*/ }
  ```

- 对具有单个字母类型参数的类型，**考虑**使用 `T` 作为类型参数名称。

  ./snippets/coding-conventions

  ```
  public int IComparer<T>() { return 0; }
  public delegate bool Predicate<T>(T item);
  public struct Nullable<T> where T : struct { /*...*/ }
  ```

- 在类型参数描述性名称前添加前缀 "T"。

  ./snippets/coding-conventions

```
public interface ISessionChannel<TSession>
{
    TSession Session { get; }
}
```

- 请考虑在参数名称中指示出类型参数的约束。 例如，约束为 `ISession` 的参数可命名为 `TSession`。

可以使用代码分析规则 CA1715 确保恰当地命名类型参数。

额外的命名约定

- 在不包括 using 指令的示例中，使用命名空间限定。 如果你知道命名空间默认导入项目中，则不必完全限定来自该命名空间的名称。 如果对于单行来说过长，则可以在点 (.) 后中断限定名称，如下面的示例所示。

```
C#
var currentPerformanceCounterCategory = new System.Diagnostics.
    PerformanceCounterCategory();
```

- 你不必更改使用 Visual Studio 设计器工具创建的对象的名称以使它们适合其他准则。

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

常见 C# 代码约定

项目 • 2024/01/10

代码标准对于在开发团队中维护代码可读性、一致性和协作至关重要。 遵循行业实践和既定准则的代码更易于理解、维护和扩展。 大多数项目通过代码约定强制要求样式一致。 dotnet/docs 和 dotnet/samples 项目并不例外。 在本系列文章中，你将了解我们的编码约定和用于强制实施这些约定的工具。 你可以按原样采用我们的约定，或修改它们以满足团队的需求。

我们对约定的选择基于以下目标：

1. *正确性*：我们的示例将会复制并粘贴到你的应用程序中。 我们希望如此，因此我们需要代码具有复原能力且正确无误，即使在多次编辑之后也是如此。
2. *教学*：示例的目的是教授 .NET 和 C# 的全部内容。 因此，我们不会对任何语言功能或 API 施加限制。 相反，这些示例会告知某个功能在何时会是良好的选择。
3. *一致性*：读者期望我们的内容提供一致的体验。 所有示例应遵循相同的样式。
4. *采用*：我们积极更新示例以使用新的语言功能。 这种做法提高了对新功能的认识，并且提高了所有 C# 开发人员对这些功能的熟悉程度。

> **ⓘ 重要**
>
> Microsoft 会使用这些准则来开发示例和文档。 它们摘自 **.NET 运行时、C# 编码样式** 和 **C# 编译器 (roslyn)** 准则。 我们选择这些准则是因为它们已经经过了多年开放源代码开发的测试。 他们帮助社区成员参与运行时和编译器项目。 它们是常见 C# 约定的示例，而不是权威列表（有关此内容，请参阅**框架设计指南**）。
>
> *教学*和*采用*目标是文档编码约定不同于运行时和编译器约定的原因。 运行时和编译器对热路径具有严格的性能指标。 许多其他应用程序则并非如此。 我们的*教学*目标要求我们不会禁止任何构造。 相反，示例显示了何时应使用构造。 与大多数生产应用程序相比，我们在更新示例方面更加积极。 我们的*采用*目标要求我们显示你目前应该编写的代码，即使去年编写的代码无需更改。

本文将对我们的准则进行说明。 这些准则已随时间推移发生变化，因此，你会发现并不遵循准则的示例。 我们欢迎推动这些示例合规的 PR，或促使我们关注应更新的示例的问题。 我们的准则是开放源代码的，因此我们欢迎 PR 和问题。 但如果你的提交将更改这些建议，请先提出一个问题以供讨论。 欢迎使用我们的准则，或根据你的需求对其进行调整。

工具和分析器

工具可帮助团队强制实施标准。 可以启用代码分析来强制实施你偏好的规则。 还可以创建 editorconfig，以便 Visual Studio 可自动强制实施样式准则。 作为起点，可以复制 dotnet/docs 存储库的文件 以使用我们的样式。

借助这些工具，团队可以更轻松地采用首选的准则。 Visual Studio 将在范围中的所有 `.editorconfig` 文件中应用规则，以设置代码的格式。 可以使用多个配置来强制实施企业范围的标准、团队标准，甚至精细的项目标准。

启用的规则被违反时，代码分析会生成警告和诊断。 可以配置想要应用于项目的规则。 然后，每个 CI 生成会在违反任何规则时通知开发人员。

诊断 ID

- 生成自己的分析器时选择适当的诊断 ID

语言准则

以下部分介绍了 .NET 文档团队在准备代码示例和示例时遵循的做法。 一般情况下，请遵循以下做法：

- 尽可能利用新式语言功能和 C# 版本。
- 避免陈旧或过时的语言构造。
- 仅捕获可以正确处理的异常；避免捕获泛型异常。
- 使用特定的异常类型提供有意义的错误消息。
- 使用 LINQ 查询和方法进行集合操作，以提高代码可读性。
- 将异步编程与异步和等待用于 I/O 绑定操作。
- 请谨慎处理死锁，并在适当时使用 Task.ConfigureAwait。
- 对数据类型而不是运行时类型使用语言关键字。 例如，使用 `string` 而不是 System.String，或使用 `int` 而不是 System.Int32。
- 使用 `int` 而不是无符号类型。 `int` 的使用在整个 C# 中很常见，并且当你使用 `int` 时，更易于与其他库交互。 特定于无符号数据类型的文档例外。。
- 仅当读者可以从表达式推断类型时使用 `var`。 读者可在文档平台上查看我们的示例。 它们没有悬停或显示变量类型的工具提示。
- 以简洁明晰的方式编写代码。
- 避免过于复杂和费解的代码逻辑。

遵循更具体的准则。

字符串数据

- 使用字符串内插来连接短字符串，如下面的代码所示。

C#

```
string displayName = $"{nameList[n].LastName},
{nameList[n].FirstName}";
```

- 若要在循环中追加字符串，尤其是在使用大量文本时，请使用 System.Text.StringBuilder 对象。

C#

```
var phrase =
"lalalalalalalalalalalalalalalalalalalalalalalalalalalalalala";
var manyPhrases = new StringBuilder();
for (var i = 0; i < 10000; i++)
{
    manyPhrases.Append(phrase);
}
//Console.WriteLine("tra" + manyPhrases);
```

数组

- 当在声明行上初始化数组时，请使用简洁的语法。 在以下示例中，不能使用 `var` 替代 `string[]`。

C#

```
string[] vowels1 = { "a", "e", "i", "o", "u" };
```

- 如果使用显式实例化，则可以使用 `var`。

C#

```
var vowels2 = new string[] { "a", "e", "i", "o", "u" };
```

委托

- 使用 Func<> 和 Action<>，而不是定义委托类型。 在类中，定义委托方法。

C#

```
Action<string> actionExample1 = x => Console.WriteLine($"x is: {x}");

Action<string, string> actionExample2 = (x, y) =>
    Console.WriteLine($"x is: {x}, y is {y}");
```

```
Func<string, int> funcExample1 = x => Convert.ToInt32(x);

Func<int, int, int> funcExample2 = (x, y) => x + y;
```

- 使用 `Func<>` 或 `Action<>` 委托定义的签名来调用方法。

C#

```
actionExample1("string for x");

actionExample2("string for x", "string for y");

Console.WriteLine($"The value is {funcExample1("1")}");

Console.WriteLine($"The sum is {funcExample2(1, 2)}");
```

- 如果创建委托类型的实例，请使用简洁的语法。 在类中，定义委托类型和具有匹配签名的方法。

  C#

  ```
  public delegate void Del(string message);

  public static void DelMethod(string str)
  {
      Console.WriteLine("DelMethod argument: {0}", str);
  }
  ```

- 创建委托类型的实例，然后调用该实例。 以下声明显示了紧缩的语法。

  C#

  ```
  Del exampleDel2 = DelMethod;
  exampleDel2("Hey");
  ```

- 以下声明使用了完整的语法。

  C#

  ```
  Del exampleDel1 = new Del(DelMethod);
  exampleDel1("Hey");
  ```

`try-catch` 和 `using` 语句正在异常处理中

- 对大多数异常处理使用 try-catch 语句。

C#

```
static double ComputeDistance(double x1, double y1, double x2, double
y2)
{
    try
    {
        return Math.Sqrt((x1 - x2) * (x1 - x2) + (y1 - y2) * (y1 -
y2));
    }
    catch (System.ArithmeticException ex)
    {
        Console.WriteLine($"Arithmetic overflow or underflow: {ex}");
        throw;
    }
}
```

- 通过使用 C# using 语句简化你的代码。 如果具有 try-finally 语句（该语句中 `finally` 块的唯一代码是对 Dispose 方法的调用），请使用 `using` 语句代替。

  在以下示例中，`try-finally` 语句仅在 `finally` 块中调用 `Dispose`。

  C#

  ```
  Font bodyStyle = new Font("Arial", 10.0f);
  try
  {
      byte charset = bodyStyle.GdiCharSet;
  }
  finally
  {
      if (bodyStyle != null)
      {
          ((IDisposable)bodyStyle).Dispose();
      }
  }
  ```

  可以使用 `using` 语句执行相同的操作。

  C#

  ```
  using (Font arial = new Font("Arial", 10.0f))
  {
      byte charset2 = arial.GdiCharSet;
  }
  ```

  使用不需要大括号的新 using 语法：

  C#

```
using Font normalStyle = new Font("Arial", 10.0f);
byte charset3 = normalStyle.GdiCharSet;
```

`&&` 和 `||` 运算符

- 在执行比较时，使用 && 而不是 &，使用 || 而不是 |，如以下示例所示。

```
C#

Console.Write("Enter a dividend: ");
int dividend = Convert.ToInt32(Console.ReadLine());

Console.Write("Enter a divisor: ");
int divisor = Convert.ToInt32(Console.ReadLine());

if ((divisor != 0) && (dividend / divisor) is var result)
{
    Console.WriteLine("Quotient: {0}", result);
}
else
{
    Console.WriteLine("Attempted division by 0 ends up here.");
}
```

如果除数为 0，则 `if` 语句中的第二个子句将导致运行时错误。 但是，当第一个表达式为 false 时，&& 运算符将发生短路。 也就是说，它并不评估第二个表达式。 如果 `divisor` 为 0，则 & 运算符将同时计算这两个表达式，从而导致运行时错误。

`new` 运算符

- 使用对象实例化的简洁形式之一，如以下声明中所示。

```
C#

var firstExample = new ExampleClass();
```

```
C#

ExampleClass instance2 = new();
```

前面的声明等效于下面的声明。

```
C#
```

```
ExampleClass secondExample = new ExampleClass();
```

- 使用对象初始值设定项简化对象创建，如以下示例中所示。

```
C#

var thirdExample = new ExampleClass { Name = "Desktop", ID = 37414,
    Location = "Redmond", Age = 2.3 };
```

下面的示例设置了与前面的示例相同的属性，但未使用初始值设定项。

```
C#

var fourthExample = new ExampleClass();
fourthExample.Name = "Desktop";
fourthExample.ID = 37414;
fourthExample.Location = "Redmond";
fourthExample.Age = 2.3;
```

事件处理

- 使用 lambda 表达式定义稍后无需移除的事件处理程序：

```
C#

public Form2()
{
    this.Click += (s, e) =>
        {
            MessageBox.Show(
                ((MouseEventArgs)e).Location.ToString());
        };
}
```

Lambda 表达式缩短了以下传统定义。

```
C#

public Form1()
{
    this.Click += new EventHandler(Form1_Click);
}

void Form1_Click(object? sender, EventArgs e)
{
```

```
    MessageBox.Show(((MouseEventArgs)e).Location.ToString());
}
```

静态成员

使用类名调用 static 成员：ClassName.StaticMember。 这种做法通过明确静态访问使代码更易于阅读。 请勿使用派生类的名称来限定基类中定义的静态成员。 编译该代码时，代码可读性具有误导性，如果向派生类添加具有相同名称的静态成员，代码可能会被破坏。

LINQ 查询

- 对查询变量使用有意义的名称。 下面的示例为位于西雅图的客户使用 `seattleCustomers`。

  C#

  ```
  var seattleCustomers = from customer in customers
                         where customer.City == "Seattle"
                         select customer.Name;
  ```

- 使用别名确保匿名类型的属性名称都使用 Pascal 大小写格式正确大写。

  C#

  ```
  var localDistributors =
      from customer in customers
      join distributor in distributors on customer.City equals
  distributor.City
      select new { Customer = customer, Distributor = distributor };
  ```

- 如果结果中的属性名称模棱两可，请对属性重命名。 例如，如果你的查询返回客户名称和分销商 ID，而不是在结果中将它们保留为 `Name` 和 `ID`，请对它们进行重命名以明确 `Name` 是客户的名称，`ID` 是分销商的 ID。

  C#

  ```
  var localDistributors2 =
      from customer in customers
      join distributor in distributors on customer.City equals
  distributor.City
      select new { CustomerName = customer.Name, DistributorID =
  distributor.ID };
  ```

- 在查询变量和范围变量的声明中使用隐式类型化。 有关 LINQ 查询中隐式类型的本指导会替代适用于隐式类型本地变量的一般规则。 LINQ 查询通常使用创建匿名类型的投影。 其他查询表达式使用嵌套泛型类型创建结果。 隐式类型变量通常更具可读性。

```
C#

var seattleCustomers = from customer in customers
                       where customer.City == "Seattle"
                       select customer.Name;
```

- 对齐 from 子句下的查询子句，如上面的示例所示。

- 在其他查询子句前面使用 where 子句，确保后面的查询子句作用于经过缩减和筛选的一组数据。

```
C#

var seattleCustomers2 = from customer in customers
                        where customer.City == "Seattle"
                        orderby customer.Name
                        select customer;
```

- 使用多行 `from` 子句代替 join 子句来访问内部集合。 例如，`Student` 对象的集合可能包含测验分数的集合。 当执行以下查询时，它返回高于 90 的分数，并返回得到该分数的学生的姓氏。

```
C#

var scoreQuery = from student in students
                 from score in student.Scores!
                 where score > 90
                 select new { Last = student.LastName, score };
```

隐式类型本地变量

- 当变量的类型在赋值右侧比较明显时，对局部变量使用隐式类型。

```
C#

var message = "This is clearly a string.";
var currentTemperature = 27;
```

- 当类型在赋值右侧不明显时，请勿使用 var。 请勿假设类型明显来自方法名称。 如果变量类型是 `new` 运算符、对文本值的显式强制转换或赋值，则将其视为明确的变量类型。

```
C#

int numberOfIterations = Convert.ToInt32(Console.ReadLine());
int currentMaximum = ExampleClass.ResultSoFar();
```

- 不要使用变量名称指定变量的类型。 它可能不正确。 请改用类型来指定类型，并使用变量名称来指示变量的语义信息。 以下示例应对类型使用 `string`，并使用类似 `iterations` 的内容指示从控制台读取的信息的含义。

```
C#

var inputInt = Console.ReadLine();
Console.WriteLine(inputInt);
```

- 避免使用 `var` 来代替 dynamic。 如果想要进行运行时类型推理，请使用 `dynamic`。 有关详细信息，请参阅使用类型 dynamic（C# 编程指南）。

- 在 for 循环中对循环变量使用隐式类型。

  下面的示例在 `for` 语句中使用隐式类型化。

```
C#

var phrase =
"lalalalalalalalalalalalalalalalalalalalalalalalalalalalalalala";
var manyPhrases = new StringBuilder();
for (var i = 0; i < 10000; i++)
{
    manyPhrases.Append(phrase);
}
//Console.WriteLine("tra" + manyPhrases);
```

- 不要使用隐式类型化来确定 foreach 循环中循环变量的类型。 在大多数情况下，集合中的元素类型并不明显。 不应仅依靠集合的名称来推断其元素的类型。

  下面的示例在 `foreach` 语句中使用显式类型化。

```
C#

foreach (char ch in laugh)
{
    if (ch == 'h')
```

```
            Console.Write("H");
        else
            Console.Write(ch);
    }
    Console.WriteLine();
```

- 对 LINQ 查询中的结果序列使用隐式类型。 关于 LINQ 的部分说明了许多 LINQ 查询会导致必须使用隐式类型的匿名类型。 其他查询则会产生嵌套泛型类型，其中 `var` 的可读性更高。

> **① 备注**
>
> 注意不要意外更改可迭代集合的元素类型。 例如，在 `foreach` 语句中从 System.Linq.IQueryable 切换到 System.Collections.IEnumerable 很容易，这会更改查询的执行。

我们的一些示例解释了表达式的*自然类型*。 这些示例必须使用 `var`，以便编译器选取自然类型。 即使这些示例不太明显，但示例必须使用 `var`。 文本应解释该行为。

将 using 指令放在命名空间声明之外

当 `using` 指令位于命名空间声明之外时，该导入的命名空间是其完全限定的名称。 完全限定的名称更加清晰。 如果 `using` 指令位于命名空间内部，则它可以是相对于该命名空间的，也可以是它的完全限定名称。

C#

```
using Azure;

namespace CoolStuff.AwesomeFeature
{
    public class Awesome
    {
        public void Stuff()
        {
            WaitUntil wait = WaitUntil.Completed;
            // ...
        }
    }
}
```

假设存在对 WaitUntil 类的引用（直接或间接）。

现在，让我们稍作改动：

C#

```
namespace CoolStuff.AwesomeFeature
{
    using Azure;

    public class Awesome
    {
        public void Stuff()
        {
            WaitUntil wait = WaitUntil.Completed;
            // ...
        }
    }
}
```

今天的编译成功了。 明天的也没问题。 但在下周的某个时候，前面（未改动）的代码失败，并出现两个错误：

控制台

```
- error CS0246: The type or namespace name 'WaitUntil' could not be found
(are you missing a using directive or an assembly reference?)
- error CS0103: The name 'WaitUntil' does not exist in the current context
```

其中一个依赖项已在命名空间中引入了此类，然后以 `.Azure` 结尾：

C#

```
namespace CoolStuff.Azure
{
    public class SecretsManagement
    {
        public string FetchFromKeyVault(string vaultId, string secretId) {
return null; }
    }
}
```

放置在命名空间中的 `using` 指令与上下文相关，使名称解析复杂化。 在此示例中，它是它找到的第一个命名空间。

- `CoolStuff.AwesomeFeature.Azure`
- `CoolStuff.Azure`
- `Azure`

添加匹配 `CoolStuff.Azure` 或 `CoolStuff.AwesomeFeature.Azure` 的新命名空间将在全局 `Azure` 命名空间前匹配。 可以通过向 `using` 声明添加 `global::` 修饰符来解决此问题。

但是，改为将 `using` 声明放在命名空间之外更容易。

C#

```
namespace CoolStuff.AwesomeFeature
{
    using global::Azure;

    public class Awesome
    {
        public void Stuff()
        {
            WaitUntil wait = WaitUntil.Completed;
            // ...
        }
    }
}
```

样式指南

一般情况下，对代码示例使用以下格式：

- 使用四个空格缩进。 不要使用选项卡。
- 一致地对齐代码以提高可读性。
- 将行限制为 65 个字符，以增强文档上的代码可读性，尤其是在移动屏幕上。
- 将长语句分解为多行以提高清晰度。
- 对大括号使用“Allman”样式：左和右大括号另起一行。 大括号与当前缩进级别对齐。
- 如有必要，应在二进制运算符之前换行。

注释样式

- 使用单行注释（`//`）以进行简要说明。
- 避免使用多行注释（`/* */`）来进行较长的解释。 注释不进行本地化处理。 相反，配套文章中提供了较长的解释。
- 若要描述方法、类、字段和所有公共成员，请使用 XML 注释。
- 将注释放在单独的行上，而非代码行的末尾。
- 以大写字母开始注释文本。
- 以句点结束注释文本。

- 在注释分隔符 (`//`) 与注释文本之间插入一个空格，如下面的示例所示。

```
C#

// The following declaration creates a query. It does not run
// the query.
```

布局约定

好的布局利用格式设置来强调代码的结构并使代码更便于阅读。 Microsoft 示例和样本符合以下约定：

- 使用默认的代码编辑器设置（智能缩进、4 字符缩进、制表符保存为空格）。 有关详细信息，请参阅选项、文本编辑器、C#、格式设置。
- 每行只写一条语句。
- 每行只写一个声明。
- 如果连续行未自动缩进，请将它们缩进一个制表符位（四个空格）。
- 在方法定义与属性定义之间添加至少一个空白行。
- 使用括号突出表达式中的子句，如下面的代码所示。

```
C#

if ((startX > endX) && (startX > previousX))
{
    // Take appropriate action.
}
```

例外情况出现在示例解释运算符或表达式优先级时。

安全性

请遵循安全编码准则中的准则。

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

如何显示命令行参数

项目 • 2023/06/08

可通过顶级语句或 `Main` 的可选参数来访问在命令行处提供给可执行文件的参数。 参数以字符串数组的形式提供。 数组的每个元素都包含 1 个参数。 删除参数之间的空格。 例如，下面是对虚构可执行文件的命令行调用：

命令行上的输入	传递给 Main 的字符串数组
executable.exe a b c	"a" "b" “c”
executable.exe one two	"one" "two"
executable.exe "one two" three	"one two" "three"

> **备注**
>
> 在 Visual Studio 中运行应用程序时，可在“**项目设计器**”->“**调试**”页中指定命令行参数。

示例

本示例显示了传递给命令行应用程序的命令行参数。 显示的输出对应于上表中的第一项。

C#

```
// The Length property provides the number of array elements.
Console.WriteLine($"parameter count = {args.Length}");

for (int i = 0; i < args.Length; i++)
{
    Console.WriteLine($"Arg[{i}] = [{args[i]}]");
}

/* Output (assumes 3 cmd line args):
```

```
    parameter count = 3
    Arg[0] = [a]
    Arg[1] = [b]
    Arg[2] = [c]
*/
```

另请参阅

- System.CommandLine 概述
- 教程：System.CommandLine 入门

使用类和对象探索面向对象的编程

项目 • 2023/06/05

在本教程中，你将生成一个控制台应用程序，并了解 C# 语言中的面向对象的基本功能。

先决条件

- 建议使用 Visual Studio for Windows 或 Mac。 可以从 Visual Studio 下载页面 下载免费版本。 Visual Studio 包括 .NET SDK。
- 还可以使用 Visual Studio Code 编辑器。 需要单独安装最新的 .NET Compiler Platform SDK。
- 如果更想要使用其他编辑器，则需要安装最新的 .NET SDK。

创建应用程序

使用终端窗口，创建名为 classes 的目录。 可以在其中生成应用程序。 将此目录更改为当前目录，并在控制台窗口中键入 `dotnet new console`。 此命令可创建应用程序。 打开 Program.cs。 应如下所示：

```
C#

// See https://aka.ms/new-console-template for more information
Console.WriteLine("Hello, World!");
```

在本教程中，将要新建表示银行帐户的类型。 通常情况下，开发者都会在不同的文本文件中定义每个类。 这样可以更轻松地管理不断增大的程序。 在 Classes 目录中，新建名为 BankAccount.cs 的文件。

此文件包含"银行帐户"定义。 面向对象的编程通过创建类形式的类型来整理代码。 这些类包含表示特定实体的代码。 `BankAccount` 类表示银行帐户。 代码通过方法和属性实现特定操作。 在本教程中，银行帐户支持以下行为：

1. 用一个 10 位数唯一标识银行帐户。
2. 用字符串存储一个或多个所有者名称。
3. 可以检索余额。
4. 接受存款。
5. 接受取款。
6. 初始余额必须是正数。
7. 取款后的余额不能是负数。

定义银行帐户类型

首先，创建定义此行为的类的基本设置。 使用 File:New 命令创建新文件。 将其命名为“BankAccount.cs”。 将以下代码添加到 BankAccount.cs 文件：

C#

```
namespace Classes;

public class BankAccount
{
    public string Number { get; }
    public string Owner { get; set; }
    public decimal Balance { get; }

    public void MakeDeposit(decimal amount, DateTime date, string note)
    {
    }

    public void MakeWithdrawal(decimal amount, DateTime date, string note)
    {
    }
}
```

继续操作前，先来看看已经生成的内容。 借助 `namespace` 声明，可以按逻辑组织代码。 由于本教程的篇幅较小，因此所有代码都将添加到一个命名空间中。

`public class BankAccount` 定义要创建的类或类型。 类声明后面 `{` 和 `}` 中的所有内容定义了类的状态和行为。 类有 5 个成员。 前三个成员是属性。 属性是数据元素，可以包含强制执行验证或其他规则的代码。 最后两个是方法。 方法是执行一个函数的代码块。 读取每个成员的名称应该能够为自己或其他开发者提供了解类用途的足够信息。

打开新帐户

要实现的第一个功能是打开银行帐户。 打开帐户时，客户必须提供初始余额，以及此帐户的一个或多个所有者的相关信息。

新建 `BankAccount` 类型的对象意味着定义构造函数来赋值。 `BankAccount` 构造函数是与类同名的成员。 用于初始化相应类类型的对象。 将以下构造函数添加到 `BankAccount` 类型。 将下面的代码放在 `MakeDeposit` 声明的上方：

C#

```
public BankAccount(string name, decimal initialBalance)
{
    this.Owner = name;
```

```
    this.Balance = initialBalance;
}
```

前面的代码通过包括 `this` 限定符标识所构造对象的属性。 该限定符通常可选且会被省略。 还可以编写：

C#

```
public BankAccount(string name, decimal initialBalance)
{
    Owner = name;
    Balance = initialBalance;
}
```

仅当局部变量或参数具有与该字段或属性相同的名称时，才需要限定符 `this`。 除非有必要，否则在本文的其余部分中省略限定符 `this`。

构造函数是在使用 new 创建对象时进行调用。 将 Program.cs 中的代码行 `Console.WriteLine("Hello World!");` 替换为以下代码行（将 `<name>` 替换为自己的名称）：

C#

```
using Classes;

var account = new BankAccount("<name>", 1000);
Console.WriteLine($"Account {account.Number} was created for {account.Owner}
with {account.Balance} initial balance.");
```

我们来运行到目前为止已构建的内容。 如果使用的是 Visual Studio，请在“调试”菜单中选择“启动而不调试”。 如果使用的是命令行，请在创建项目的目录中键入 `dotnet run`。

有没有注意到帐号为空？ 是时候解决这个问题了。 帐号应在构造对象时分配。 但不得由调用方负责创建。 `BankAccount` 类代码应了解如何分配新帐号。 一种简单的方法是从一个 10 位数开始。 帐号随每个新建的帐户而递增。 最后，在构造对象时，存储当前的帐号。

将成员声明添加到 `BankAccount` 类。 将以下代码行放在 `BankAccount` 类开头的左括号 `{` 后面：

C#

```
private static int accountNumberSeed = 1234567890;
```

`accountNumberSeed` 为数据成员。 它是 `private`，这意味着只能通过 `BankAccount` 类中的代码访问它。 这是一种分离公共责任（如拥有帐号）与私有实现（如何生成帐号）的方法。 它也是 `static`，这意味着它由所有 `BankAccount` 对象共享。 非静态变量的值对于 `BankAccount` 对象的每个实例是唯一的。 将下面两行代码添加到构造函数，以分配帐号。 将它们放在 `this.Balance = initialBalance` 行后面：

C#

```
this.Number = accountNumberSeed.ToString();
accountNumberSeed++;
```

键入 `dotnet run` 看看结果如何。

创建存款和取款

银行帐户类必须接受存款和取款，才能正常运行。 接下来，将为银行帐户创建每笔交易日记，实现存款和取款。 与仅更新每笔交易余额相比，跟踪每一笔交易都有一些优点。 历史记录可用于审核所有交易，并管理每日余额。 在需要时根据所有交易的历史记录计算余额，从而确保单笔交易中修正的任何错误将会在下次计算余额时得到正确体现。

接下来，先新建表示交易的类型。 此交易是一个没有任何责任的简单类型。 但需要有多个属性。 新建名为 Transaction.cs 的文件。 向新文件添加以下代码：

C#

```
namespace Classes;

public class Transaction
{
    public decimal Amount { get; }
    public DateTime Date { get; }
    public string Notes { get; }

    public Transaction(decimal amount, DateTime date, string note)
    {
        Amount = amount;
        Date = date;
        Notes = note;
    }
}
```

现在，将 `Transaction` 对象的 List<T> 添加到 `BankAccount` 类中。 将以下声明放在 BankAccount.cs 文件中的构造函数后面：

C#

```
private List<Transaction> allTransactions = new List<Transaction>();
```

现在，让我们来正确计算 `Balance`。 可以通过对所有交易的值进行求和来计算当前余额。 由于当前代码，你只能计算出帐户的初始余额，因此必须更新 `Balance` 属性。 将 BankAccount.cs 中的 `public decimal Balance { get; }` 行替换为以下代码：

C#

```
public decimal Balance
{
    get
    {
        decimal balance = 0;
        foreach (var item in allTransactions)
        {
            balance += item.Amount;
        }

        return balance;
    }
}
```

此示例反映了属性的一个重要方面。 现在，可以在其他程序员要求获取余额时计算值。 计算会枚举所有交易，总和即为当前余额。

接下来，实现 `MakeDeposit` 和 `MakeWithdrawal` 方法。 这些方法将强制执行最后两条规则：初始余额必须为正数，且取款后的余额不能是负数。

这些规则引入了异常的概念。 指明方法无法成功完成工作的标准方式是引发异常。 异常类型及其关联消息描述了错误。 在此示例中，如果存款金额为负数，`MakeDeposit` 方法会引发异常。 如果取款金额为负数，或者取款后的余额为负数，`MakeWithdrawal` 方法会引发异常。 将以下代码添加到 `allTransactions` 列表的声明后面：

C#

```
public void MakeDeposit(decimal amount, DateTime date, string note)
{
    if (amount <= 0)
    {
        throw new ArgumentOutOfRangeException(nameof(amount), "Amount of
deposit must be positive");
    }
    var deposit = new Transaction(amount, date, note);
    allTransactions.Add(deposit);
}

public void MakeWithdrawal(decimal amount, DateTime date, string note)
{
```

```
    if (amount <= 0)
    {
        throw new ArgumentOutOfRangeException(nameof(amount), "Amount of
withdrawal must be positive");
    }
    if (Balance - amount < 0)
    {
        throw new InvalidOperationException("Not sufficient funds for this
withdrawal");
    }
    var withdrawal = new Transaction(-amount, date, note);
    allTransactions.Add(withdrawal);
}
```

throw 语句**引发**异常。 当前块执行结束，将控制权移交给在调用堆栈中发现的第一个匹配的 `catch` 块。 添加 `catch` 块可以稍后再测试一下此代码。

构造函数应进行一处更改，更改为添加初始交易，而不是直接更新余额。 由于已编写 `MakeDeposit` 方法，因此通过构造函数调用它。 完成的构造函数应如下所示：

C#

```
public BankAccount(string name, decimal initialBalance)
{
    Number = accountNumberSeed.ToString();
    accountNumberSeed++;

    Owner = name;
    MakeDeposit(initialBalance, DateTime.Now, "Initial balance");
}
```

DateTime.Now 是返回当前日期和时间的属性。 在创建新 `BankAccount` 的代码后面，在 `Main` 方法中添加几个存款和取款，对此代码进行测试：

C#

```
account.MakeWithdrawal(500, DateTime.Now, "Rent payment");
Console.WriteLine(account.Balance);
account.MakeDeposit(100, DateTime.Now, "Friend paid me back");
Console.WriteLine(account.Balance);
```

接下来，尝试创建初始余额为负数的帐户，测试能否捕获到错误条件。 在刚刚添加的上述代码后面，添加以下代码：

C#

```
// Test that the initial balances must be positive.
BankAccount invalidAccount;
```

```
try
{
    invalidAccount = new BankAccount("invalid", -55);
}
catch (ArgumentOutOfRangeException e)
{
    Console.WriteLine("Exception caught creating account with negative
balance");
    Console.WriteLine(e.ToString());
    return;
}
```

使用 try-catch 语句标记可能会引发异常的代码块，并捕获预期错误。 可以使用相同的技术，测试代码能否在取款后余额为负数时引发异常。 在 `Main` 方法中的 `invalidAccount` 声明之前添加以下代码：

C#

```
// Test for a negative balance.
try
{
    account.MakeWithdrawal(750, DateTime.Now, "Attempt to overdraw");
}
catch (InvalidOperationException e)
{
    Console.WriteLine("Exception caught trying to overdraw");
    Console.WriteLine(e.ToString());
}
```

保存此文件，并键入 `dotnet run`，试运行看看。

挑战 - 记录所有交易

为了完成本教程，可以编写 `GetAccountHistory` 方法，为交易历史记录创建 `string`。 将此方法添加到 `BankAccount` 类型中：

C#

```
public string GetAccountHistory()
{
    var report = new System.Text.StringBuilder();

    decimal balance = 0;
    report.AppendLine("Date\t\tAmount\tBalance\tNote");
    foreach (var item in allTransactions)
    {
        balance += item.Amount;
        report.AppendLine($"
```

```
{item.Date.ToShortDateString()}\t{item.Amount}\t{balance}\t{item.Notes}");
    }

    return report.ToString();
}
```

历史记录使用 StringBuilder 类，设置包含每个交易行的字符串的格式。 在前面的教程中，也遇到过字符串格式设置代码。 新增的一个字符为 `\t`。 这用于插入选项卡，从而设置输出格式。

添加以下代码行，在 Program.cs 中对它进行测试：

C#

```
Console.WriteLine(account.GetAccountHistory());
```

运行程序以查看结果。

后续步骤

如果遇到问题，可以在 GitHub 存储库 中查看本教程的源代码。

可继续学习面向对象的编程教程。

若要详细了解这些概念，请参阅下列文章：

- 选择语句
- 迭代语句

面向对象的编程 (C#)

项目 • 2023/06/08

C# 是面向对象的编程语言。 面向对象编程的四项基本原则为：

- 抽象：将实体的相关特性和交互建模为类，以定义系统的抽象表示。
- 封装：隐藏对象的内部状态和功能，并仅允许通过一组公共函数进行访问。
- 继承：根据现有抽象创建新抽象的能力。
- 多形性：跨多个抽象以不同方式实现继承属性或方法的能力。

在前面的类简介 教程中，我们介绍了抽象和封装。 `BankAccount` 类提供银行帐户这一概念的抽象。 你可以修改其实现，而不影响使用 `BankAccount` 类的任何代码。 `BankAccount` 和 `Transaction` 类都提供在代码中描述这些概念所需组件的封装。

在本教程中，你将扩展该应用程序以利用继承和多形性来添加新功能。 你还将利用在上一教程中学到的抽象和封装技术，向 `BankAccount` 类添加功能。

创建不同类型的帐户

生成此程序后，你将获取向其添加功能的请求。 在只有一种银行帐户类型的情况下，其效果良好。 随着时间的推移，需求会发生变化，因而请求了相关的帐户类型：

- 在每个月的月末获得利息的红利帐户。
- 余额可以为负，但存在余额时会产生每月利息的信用帐户。
- 以单笔存款开户且只能用于支付的预付礼品卡帐户。 可在每月初重新充值一次。

所有这些帐户都与前面的教程中定义的 `BankAccount` 类类似。 你可以复制该代码、重命名类并进行修改。 这种方法在短期内有效，但随着时间的推移，其效果会更为明显。 所有更改都将复制到所有受影响的类。

相反，你可以创建新的银行帐户类型，使其从上一教程中创建的 `BankAccount` 类继承方法和数据。 这些新类可以用每种类型所需的特定行为来扩展 `BankAccount` 类：

C#

```
public class InterestEarningAccount : BankAccount
{
}

public class LineOfCreditAccount : BankAccount
{
}

public class GiftCardAccount : BankAccount
```

```
{
}
```

其中的每个类都从其共享的基类（`BankAccount` 类）继承共享的行为。 为的每个派生类中的新增和不同功能编写实现。 这些派生类已具有 `BankAccount` 类中定义的所有行为。

最好在不同的源文件中创建每个新类。 在 Visual Studio 中，可以右键单击项目，然后选择“添加类”以在新文件中添加新类。 在 Visual Studio Code 中，选择“文件”，然后选择“新建”以创建新的源文件。 在任一工具中，将文件命名为与类匹配：InterestEarningAccount.cs、LineOfCreditAccount.cs 和 GiftCardAccount.cs。

如前一示例中所示创建类时，你会发现派生类都没有编译。 构造函数负责初始化对象。 派生类构造函数必须初始化派生类，并提供有关如何初始化派生类中所包含基类对象的说明。 一般无需任何额外的代码即可正常初始化。 `BankAccount` 类声明一个具有以下签名的公共构造函数：

C#

```
public BankAccount(string name, decimal initialBalance)
```

当你自行定义构造函数时，编译器不会生成默认构造函数。 这意味着每个派生类必须显式调用此构造函数。 声明可将自变量传递到基类构造函数的构造函数。 下面的代码显示了 `InterestEarningAccount` 的构造函数：

C#

```
public InterestEarningAccount(string name, decimal initialBalance) :
base(name, initialBalance)
{
}
```

此新构造函数的参数与基类构造函数的参数类型和名称匹配。 使用 `: base()` 语法来指示对基类构造函数的调用。 某些类定义多个构造函数，此语法让你可以选取调用的基类构造函数。 更新构造函数后，可以为每个派生类开发代码。 可以这样描述对新类的要求：

- 红利帐户：
  - 将获得月末余额 2% 的额度。
- 信用帐户：
  - 余额可以为负，但不能大于信用限额的绝对值。
  - 如果月末余额不为 0，每个月都会产生利息。
  - 将在超过信用限额的每次提款后收取费用。
- 礼品卡帐户：
  - 每月最后一天，可以充值一次指定的金额。

可以看到，这三种帐户类型都有一个在每月月末发生的操作。 但是，每种帐户类型负责不同的任务。 可以使用多态性来实现此代码。 在 `BankAccount` 类中创建单个 `virtual` 方法：

C#

```
public virtual void PerformMonthEndTransactions() { }
```

前面的代码演示如何使用 `virtual` 关键字在基类中声明一个方法，让派生类可以为该方法提供不同的实现。 在 `virtual` 方法种，任何派生类都可以选择重新实现。 派生类使用 `override` 关键字定义新的实现。 通常将其称为“重写基类实现”。 `virtual` 关键字指定派生类可以重写此行为。 还可以声明 `abstract` 方法，让派生类必须在其中重写此行为。 基类不提供 `abstract` 方法的实现。 接下来，需要为你创建的两个新类定义实现。 从 `InterestEarningAccount` 开始：

C#

```
public override void PerformMonthEndTransactions()
{
    if (Balance > 500m)
    {
        decimal interest = Balance * 0.05m;
        MakeDeposit(interest, DateTime.Now, "apply monthly interest");
    }
}
```

将以下代码添加到 `LineOfCreditAccount`。 此代码会取余额的相反数，计算从该帐户提取的正利息：

C#

```
public override void PerformMonthEndTransactions()
{
    if (Balance < 0)
    {
        // Negate the balance to get a positive interest charge:
        decimal interest = -Balance * 0.07m;
        MakeWithdrawal(interest, DateTime.Now, "Charge monthly interest");
    }
}
```

`GiftCardAccount` 类需要通过两项更改来实现其月末功能。 首先，将构造函数修改为包含每个月要充值的可选金额：

C#

```
private readonly decimal _monthlyDeposit = 0m;

public GiftCardAccount(string name, decimal initialBalance, decimal
monthlyDeposit = 0) : base(name, initialBalance)
    => _monthlyDeposit = monthlyDeposit;
```

构造函数为 `monthlyDeposit` 值提供默认值，因此调用方可以省略 `0` 以表示不进行每月存款。 接下来，如果已在构造函数中将 `PerformMonthEndTransactions` 方法设置为非零值，则重写该方法以添加每月存款：

C#

```
public override void PerformMonthEndTransactions()
{
    if (_monthlyDeposit != 0)
    {
        MakeDeposit(_monthlyDeposit, DateTime.Now, "Add monthly deposit");
    }
}
```

重写将在构造函数中应用每月存款设置。 将以下代码添加到 `Main` 方法，以便为 `GiftCardAccount` 和 `InterestEarningAccount` 测试这些更改：

C#

```
var giftCard = new GiftCardAccount("gift card", 100, 50);
giftCard.MakeWithdrawal(20, DateTime.Now, "get expensive coffee");
giftCard.MakeWithdrawal(50, DateTime.Now, "buy groceries");
giftCard.PerformMonthEndTransactions();
// can make additional deposits:
giftCard.MakeDeposit(27.50m, DateTime.Now, "add some additional spending
money");
Console.WriteLine(giftCard.GetAccountHistory());

var savings = new InterestEarningAccount("savings account", 10000);
savings.MakeDeposit(750, DateTime.Now, "save some money");
savings.MakeDeposit(1250, DateTime.Now, "Add more savings");
savings.MakeWithdrawal(250, DateTime.Now, "Needed to pay monthly bills");
savings.PerformMonthEndTransactions();
Console.WriteLine(savings.GetAccountHistory());
```

验证结果。 现在，为 `LineOfCreditAccount` 添加一组类似的测试代码：

C#

```
var lineOfCredit = new LineOfCreditAccount("line of credit", 0);
// How much is too much to borrow?
lineOfCredit.MakeWithdrawal(1000m, DateTime.Now, "Take out monthly
```

```
advance");
lineOfCredit.MakeDeposit(50m, DateTime.Now, "Pay back small amount");
lineOfCredit.MakeWithdrawal(5000m, DateTime.Now, "Emergency funds for
repairs");
lineOfCredit.MakeDeposit(150m, DateTime.Now, "Partial restoration on
repairs");
lineOfCredit.PerformMonthEndTransactions();
Console.WriteLine(lineOfCredit.GetAccountHistory());
```

添加前面的代码并运行程序时，你将看到类似于以下错误的内容：

控制台

```
Unhandled exception. System.ArgumentOutOfRangeException: Amount of deposit
must be positive (Parameter 'amount')
   at OOProgramming.BankAccount.MakeDeposit(Decimal amount, DateTime date,
String note) in BankAccount.cs:line 42
   at OOProgramming.BankAccount..ctor(String name, Decimal initialBalance)
in BankAccount.cs:line 31
   at OOProgramming.LineOfCreditAccount..ctor(String name, Decimal
initialBalance) in LineOfCreditAccount.cs:line 9
   at OOProgramming.Program.Main(String[] args) in Program.cs:line 29
```

> ⓘ **备注**
>
> 实际输出包含项目文件夹的完整路径。 为简洁起见，省略了文件夹名称。 此外，根据你的代码格式，行号可能略有不同。

此代码会失败，因为 `BankAccount` 假设初始余额必须大于 0。 `BankAccount` 类固有的另一假设是余额不能为负。 相反，将拒绝透支帐户的任何提款。 这两个假设都需要更改。 信用帐户从 0 开始，一般情况下余额为负。 此外，如果客户借款量过大，将产生费用。 会接受该交易，只是会增加费用。 可以通过向 `BankAccount` 构造函数添加用于指定最小余额的可选参数来实现第一条规则。 默认为 `0`。 第二条规则需要允许派生类修改默认算法的机制。 在某种意义上，基类会“询问”派生类型在透支时应该怎么做。 默认行为是通过引发异常来拒绝交易。

首先，让我们添加包含可选 `minimumBalance` 参数的第二个构造函数。 此新构造函数会执行现有构造函数完成的所有操作。 此外，它还会设置最小余额属性。 可以复制现有构造函数的正文，但这意味着将来需要更改两个位置。 相反，可以使用构造函数链接，让一个构造函数调用另一个构造函数。 下面的代码显示了两个构造函数和新的附加字段：

C#

```
private readonly decimal _minimumBalance;
```

```
public BankAccount(string name, decimal initialBalance) : this(name,
initialBalance, 0) { }

public BankAccount(string name, decimal initialBalance, decimal
minimumBalance)
{
    Number = s_accountNumberSeed.ToString();
    s_accountNumberSeed++;

    Owner = name;
    _minimumBalance = minimumBalance;
    if (initialBalance > 0)
        MakeDeposit(initialBalance, DateTime.Now, "Initial balance");
}
```

前面的代码演示了两种新方法。 首先，`minimumBalance` 字段被标记为 `readonly`。 这意味着构造对象之后不能更改值。 创建 `BankAccount` 后，`minimumBalance` 不可更改。 其次，采用两个参数的构造函数使用 `: this(name, initialBalance, 0) { }` 作为其实现。 `: this()` 表达式调用另一个构造函数（它具有三个参数）。 即使客户端代码可以从多个构造函数中进行选择，你也可以使用单个实现来初始化对象。

仅当初始余额大于 `0` 时，此实现才会调用 `MakeDeposit`。 这将保留存款必须为正的规则，同时允许信用帐户以 `0` 余额开户。

既然 `BankAccount` 类具有用于规定最小余额的只读字段，最终更改就是在 `MakeWithdrawal` 方法中将硬编码的 `0` 更改为 `minimumBalance`：

C#

```
if (Balance - amount < minimumBalance)
```

扩展 `BankAccount` 类后，可以修改 `LineOfCreditAccount` 构造函数以调用基构造函数，如以下代码中所示：

C#

```
public LineOfCreditAccount(string name, decimal initialBalance, decimal
creditLimit) : base(name, initialBalance, -creditLimit)
{
}
```

请注意，`LineOfCreditAccount` 构造函数会更改 `creditLimit` 参数的标志，使其与 `minimumBalance` 参数的意义匹配。

不同的透支规则

要添加的最后一项功能让 `LineOfCreditAccount` 可以对超过限额的取款进行收费，而不是拒绝交易。

一种方法是定义一个虚函数，并在其中实现所需的行为。 `BankAccount` 类将 `MakeWithdrawal` 方法重构为两个方法。 当取款使余额低于最小值时，新方法将执行指定的操作。 现有的 `MakeWithdrawal` 方法具有以下代码：

C#

```
public void MakeWithdrawal(decimal amount, DateTime date, string note)
{
    if (amount <= 0)
    {
        throw new ArgumentOutOfRangeException(nameof(amount), "Amount of
withdrawal must be positive");
    }
    if (Balance - amount < _minimumBalance)
    {
        throw new InvalidOperationException("Not sufficient funds for this
withdrawal");
    }
    var withdrawal = new Transaction(-amount, date, note);
    allTransactions.Add(withdrawal);
}
```

将其替换为以下代码：

C#

```
public void MakeWithdrawal(decimal amount, DateTime date, string note)
{
    if (amount <= 0)
    {
        throw new ArgumentOutOfRangeException(nameof(amount), "Amount of
withdrawal must be positive");
    }
    Transaction? overdraftTransaction = CheckWithdrawalLimit(Balance -
amount < _minimumBalance);
    Transaction? withdrawal = new(-amount, date, note);
    _allTransactions.Add(withdrawal);
    if (overdraftTransaction != null)
        _allTransactions.Add(overdraftTransaction);
}

protected virtual Transaction? CheckWithdrawalLimit(bool isOverdrawn)
{
    if (isOverdrawn)
    {
        throw new InvalidOperationException("Not sufficient funds for this
withdrawal");
    }
```

```
    else
    {
        return default;
    }
}
```

添加的方法是 `protected`，这意味着只能从派生类中调用它。 该声明会阻止其他客户端调用该方法。 它还是 `virtual` 的，因此派生类可以更改行为。 返回类型为 `Transaction?`。 `?` 批注指示该方法可能返回 `null`。 在 `LineOfCreditAccount` 中添加以下实现，以在超过取款限额时收取费用：

C#

```
protected override Transaction? CheckWithdrawalLimit(bool isOverdrawn) =>
    isOverdrawn
    ? new Transaction(-20, DateTime.Now, "Apply overdraft fee")
    : default;
```

当帐户透支时，该重写将返回费用交易。 如果取款未超出限额，则该方法将返回 `null` 交易。 这表明不收取任何费用。 通过将以下代码添加到 `Program` 类中的 `Main` 方法来测试这些更改：

C#

```
var lineOfCredit = new LineOfCreditAccount("line of credit", 0, 2000);
// How much is too much to borrow?
lineOfCredit.MakeWithdrawal(1000m, DateTime.Now, "Take out monthly
advance");
lineOfCredit.MakeDeposit(50m, DateTime.Now, "Pay back small amount");
lineOfCredit.MakeWithdrawal(5000m, DateTime.Now, "Emergency funds for
repairs");
lineOfCredit.MakeDeposit(150m, DateTime.Now, "Partial restoration on
repairs");
lineOfCredit.PerformMonthEndTransactions();
Console.WriteLine(lineOfCredit.GetAccountHistory());
```

运行该程序，并检查结果。

摘要

如果遇到问题，可以在 GitHub 存储库 中查看本教程的源代码。

本教程演示了面向对象的编程中使用的多种技术：

- 为每个不同的帐户类型定义了类时，你使用了抽象。 这些类描述了该帐户类型的行为。

- 在每个类中将许多详细信息保留为 `private` 时，你使用了封装。
- 使用已在 `BankAccount` 类中创建的实现来保存代码时，你使用了继承。
- 创建 `virtual` 方法，派生类可以重写它们来创建该帐户类型的特定行为时，你使用了多形性。

C# 和 .NET 中的继承

项目 • 2023/04/08

此教程将介绍 C# 中的继承。 继承是面向对象的编程语言的一项功能，可方便你定义提供特定功能（数据和行为）的基类，并定义继承或重写此功能的派生类。

先决条件

- 建议使用 Visual Studio for Windows 或 Mac。 可以从 Visual Studio 下载页面 下载免费版本。 Visual Studio 包括 .NET SDK。
- 还可以使用 Visual Studio Code 编辑器。 需要单独安装最新的 .NET Compiler Platform SDK。
- 如果更想要使用其他编辑器，则需要安装最新的 .NET SDK。

运行示例

若要创建并运行此教程中的示例，请通过命令行使用 dotnet 实用工具。 对于每个示例，请按照以下步骤操作：

1. 创建用于存储示例的目录。
2. 在命令提示符处，输入 dotnet new console 命令，新建 .NET Core 项目。
3. 将示例中的代码复制并粘贴到代码编辑器中。
4. 在命令行处输入 dotnet restore 命令，加载或还原项目的依赖项。

   无需运行 dotnet restore，因为它由所有需要还原的命令隐式运行，如 `dotnet new`、`dotnet build`、`dotnet run`、`dotnet test`、`dotnet publish` 和 `dotnet pack`。 若要禁用隐式还原，请使用 `--no-restore` 选项。

   在执行显式还原有意义的某些情况下，例如 `dotnet restore` 中，或在需要显式控制还原发生时间的生成系统中，`dotnet restore` 命令仍然有用。

   有关如何使用 NuGet 源的信息，请参阅 文档。
5. 输入 dotnet run 命令，编译并执行示例。

背景：什么是继承?

*继承*是面向对象的编程的一种基本特性。 借助继承，能够定义可重用（继承）、扩展或修改父类行为的子类。 成员被继承的类称为*基类*。 继承基类成员的类称为*派生类*。

C# 和 .NET 只支持*单一继承*。 也就是说，类只能继承自一个类。 不过，继承是可传递的。这样一来，就可以为一组类型定义继承层次结构。 换言之，类型 `D` 可继承自类型 `C`，其中类型 `C` 继承自类型 `B`，类型 `B` 又继承自基类类型 `A`。 由于继承是可传递的，因此类型 `D` 继承了类型 `A` 的成员。

并非所有基类成员都可供派生类继承。 以下成员无法继承：

- 静态构造函数：用于初始化类的静态数据。
- 实例构造函数：在创建类的新实例时调用。 每个类都必须定义自己的构造函数。
- 终结器：由运行时的垃圾回收器调用，用于销毁类实例。

虽然基类的其他所有成员都可供派生类继承，但这些成员是否可见取决于它们的可访问性。 成员的可访问性决定了其是否在派生类中可见，如下所述：

- 只有在基类中嵌套的派生类中，私有成员才可见。 否则，此类成员在派生类中不可见。 在以下示例中，`A.B` 是派生自 `A` 的嵌套类，而 `C` 则派生自 `A`。 私有 `A._value` 字段在 A.B 中可见。不过，如果从 `C.GetValue` 方法中删除注释并尝试编译示例，则会生成编译器错误 CS0122：“"A._value" 不可访问，因为它具有一定的保护级别。”

  C#

```
public class A
{
    private int _value = 10;

    public class B : A
    {
        public int GetValue()
        {
            return _value;
        }
    }
}

public class C : A
{
    //    public int GetValue()
    //    {
    //        return _value;
    //    }
}

public class AccessExample
```

```
{
    public static void Main(string[] args)
    {
        var b = new A.B();
        Console.WriteLine(b.GetValue());
    }
}
// The example displays the following output:
//       10
```

- 受保护成员仅在派生类中可见。
- 内部成员仅在与基类同属一个程序集的派生类中可见， 在与基类属于不同程序集的派生类中不可见。
- 公共成员在派生类中可见，并且属于派生类的公共接口。 可以调用继承的公共成员，就像它们是在派生类中定义一样。 在以下示例中，类 `A` 定义 `Method1` 方法，类 `B` 继承自类 `A`。 然后，以下示例调用 `Method1`，就像它是 `B` 中的实例方法一样。

C#

```
public class A
{
    public void Method1()
    {
        // Method implementation.
    }
}

public class B : A
{ }

public class Example
{
    public static void Main()
    {
        B b = new ();
        b.Method1();
    }
}
```

派生类还可以通过提供重写实现代码来*重写*继承的成员。 基类成员必须标记有 virtual 关键字，才能重写继承的成员。 默认情况下，基类成员没有 `virtual` 标记，因此无法被重写。 尝试重写非虚拟成员（如以下示例所示）将生成编译器错误 CS0506：“<member>无法重写继承的成员成员<>，因为它未标记为虚拟、抽象或重写。”

C#

```
```

```
public class A
{
    public void Method1()
    {
        // Do something.
    }
}

public class B : A
{
    public override void Method1() // Generates CS0506.
    {
        // Do something else.
    }
}
```

在某些情况下，派生类*必须*重写基类实现代码。 标记有 abstract 关键字的基类成员要求派生类必须重写它们。 如果尝试编译以下示例，则会生成编译器错误 CS0534：“<class>不实现继承的抽象成员 <member>”，因为类 `B` 没有提供 `A.Method1` 的实现代码。

C#

```
public abstract class A
{
    public abstract void Method1();
}

public class B : A // Generates CS0534.
{
    public void Method3()
    {
        // Do something.
    }
}
```

继承仅适用于类和接口。 其他各种类型（结构、委托和枚举）均不支持继承。 由于这些规则，尝试编译类似以下示例的代码会产生编译器错误 CS0527：“接口列表中的类型 "ValueType" 不是一个接口。”该错误消息指示，尽管可定义结构所实现的接口，但不支持继承。

C#

```
public struct ValueStructure : ValueType // Generates CS0527.
{
}
```

隐式继承

.NET 类型系统中的所有类型除了可以通过单一继承进行继承之外，还可以隐式继承自 Object 或其派生的类型。 Object 的常用功能可用于任何类型。

为了说明隐式继承的具体含义，让我们来定义一个新类 `SimpleClass`，这只是一个空类定义：

C#

```
public class SimpleClass
{ }
```

然后可以使用反射（便于检查类型的元数据，从而获取此类型的相关信息），获取 `SimpleClass` 类型的成员列表。 尽管没有在 `SimpleClass` 类中定义任何成员，但示例输出表明它实际上有九个成员。 这些成员的其中之一是由 C# 编译器自动为 `SimpleClass` 类型提供的无参数（或默认）构造函数。 剩余八个是 Object（.NET 类型系统中的所有类和接口最终隐式继承自的类型）的成员。

C#

```
using System.Reflection;

public class SimpleClassExample
{
    public static void Main()
    {
        Type t = typeof(SimpleClass);
        BindingFlags flags = BindingFlags.Instance | BindingFlags.Static |
BindingFlags.Public |
                             BindingFlags.NonPublic |
BindingFlags.FlattenHierarchy;
        MemberInfo[] members = t.GetMembers(flags);
        Console.WriteLine($"Type {t.Name} has {members.Length} members: ");
        foreach (MemberInfo member in members)
        {
            string access = "";
            string stat = "";
            var method = member as MethodBase;
            if (method != null)
            {
                if (method.IsPublic)
                    access = " Public";
                else if (method.IsPrivate)
                    access = " Private";
                else if (method.IsFamily)
                    access = " Protected";
                else if (method.IsAssembly)
                    access = " Internal";
                else if (method.IsFamilyOrAssembly)
                    access = " Protected Internal ";
                if (method.IsStatic)
```

```
                    stat = " Static";
            }
            string output = $"{member.Name} ({member.MemberType}): {access}
{stat}, Declared by {member.DeclaringType}";
            Console.WriteLine(output);
        }
    }
}
// The example displays the following output:
//  Type SimpleClass has 9 members:
//  ToString (Method):  Public, Declared by System.Object
//  Equals (Method):  Public, Declared by System.Object
//  Equals (Method):  Public Static, Declared by System.Object
//  ReferenceEquals (Method):  Public Static, Declared by System.Object
//  GetHashCode (Method):  Public, Declared by System.Object
//  GetType (Method):  Public, Declared by System.Object
//  Finalize (Method):  Internal, Declared by System.Object
//  MemberwiseClone (Method):  Internal, Declared by System.Object
//  .ctor (Constructor):  Public, Declared by SimpleClass
```

由于隐式继承自 Object 类，因此 `SimpleClass` 类可以使用下面这些方法：

- 公共 `ToString` 方法将 `SimpleClass` 对象转换为字符串表示形式，返回完全限定的类型名称。 在这种情况下，`ToString` 方法返回字符串“SimpleClass”。
- 三个用于测试两个对象是否相等的方法：公共实例 `Equals(Object)` 方法、公共静态 `Equals(Object, Object)` 方法和公共静态 `ReferenceEquals(Object, Object)` 方法。默认情况下，这三个方法测试的是引用相等性；也就是说，两个对象变量必须引用同一个对象，才算相等。
- 公共 `GetHashCode` 方法：计算允许在经哈希处理的集合中使用类型实例的值。
- 公共 `GetType` 方法：返回表示 `SimpleClass` 类型的 Type 对象。
- 受保护 Finalize 方法：用于在垃圾回收器回收对象的内存之前释放非托管资源。
- 受保护 MemberwiseClone 方法：创建当前对象的浅表复制。

由于是隐式继承，因此可以调用 `SimpleClass` 对象中任何继承的成员，就像它实际上是 `SimpleClass` 类中定义的成员一样。 例如，下面的示例调用 `SimpleClass` 从 Object 继承而来的 `SimpleClass.ToString` 方法。

C#

```
public class EmptyClass
{ }

public class ClassNameExample
{
```

```
    public static void Main()
    {
        EmptyClass sc = new();
        Console.WriteLine(sc.ToString());
    }
}
// The example displays the following output:
//        EmptyClass
```

下表列出了可以在 C# 中创建的各种类型及其隐式继承自的类型。 每个基类型通过继承向隐式派生的类型提供一组不同的成员。

类型类别	隐式继承自
class	Object
struct	ValueType, Object
enum	Enum, ValueType, Object
delegate	MulticastDelegate, Delegate, Object

继承和“is a”关系

通常情况下，继承用于表示基类和一个或多个派生类之间的“is a”关系，其中派生类是基类的特定版本；派生类是基类的具体类型。 例如，`Publication` 类表示任何类型的出版物，`Book` 和 `Magazine` 类表示出版物的具体类型。

> **备注**
>
> 一个类或结构可以实现一个或多个接口。 虽然接口实现代码通常用作单一继承的解决方法或对结构使用继承的方法，但它旨在表示接口与其实现类型之间的不同关系（即“can do”关系），而不是继承关系。 接口定义了其向实现类型提供的一部分功能（如测试相等性、比较或排序对象，或支持区域性敏感的分析和格式设置）。

请注意，“is a”还表示类型与其特定实例化之间的关系。 在以下示例中，`Automobile` 类包含三个唯一只读属性：`Make`（汽车制造商）、`Model`（汽车型号）和 `Year`（汽车出厂年份）。 `Automobile` 类还有一个自变量被分配给属性值的构造函数，并将 Object.ToString 方法重写为生成唯一标识 `Automobile` 实例（而不是 `Automobile` 类）的字符串。

C#

```
public class Automobile
{
```

```
    public Automobile(string make, string model, int year)
    {
        if (make == null)
            throw new ArgumentNullException(nameof(make), "The make cannot
be null.");
        else if (string.IsNullOrWhiteSpace(make))
            throw new ArgumentException("make cannot be an empty string or
have space characters only.");
        Make = make;

        if (model == null)
            throw new ArgumentNullException(nameof(model), "The model cannot
be null.");
        else if (string.IsNullOrWhiteSpace(model))
            throw new ArgumentException("model cannot be an empty string or
have space characters only.");
        Model = model;

        if (year < 1857 || year > DateTime.Now.Year + 2)
            throw new ArgumentException("The year is out of range.");
        Year = year;
    }

    public string Make { get; }

    public string Model { get; }

    public int Year { get; }

    public override string ToString() => $"{Year} {Make} {Model}";
}
```

在这种情况下，不得依赖继承来表示特定汽车品牌和型号。 例如，不需要定义 `Packard` 类型来表示帕卡德制造的汽车。 相反，可以通过创建将相应值传递给其类构造函数的 `Automobile` 对象来进行表示，如以下示例所示。

C#

```
using System;

public class Example
{
    public static void Main()
    {
        var packard = new Automobile("Packard", "Custom Eight", 1948);
        Console.WriteLine(packard);
    }
}
// The example displays the following output:
//       1948 Packard Custom Eight
```

基于继承的"is a"关系最适用于基类和向基类添加附加成员或需要基类没有的其他功能的派生类。

设计基类及其派生类

让我们来看看如何设计基类及其派生类。 在此部分中，将定义一个基类 `Publication`，用于表示任何类型的出版物，如书籍、杂志、报纸、期刊、文章等。还将定义一个从 `Publication` 派生的 `Book` 类。 可以将示例轻松扩展为定义其他派生类，如 `Magazine`、`Journal`、 `Newspaper` 和 `Article`。

Publication 基类

设计 `Publication` 类时，需要做出下面几项设计决策：

- 要在 `Publication` 基类中添加哪些成员、 `Publication` 成员是否提供方法实现或 `Publication` 是否是用作派生类模板的抽象基类。

  在此示例中， `Publication` 类提供方法实现代码。 设计抽象基类及其派生类部分中的示例就使用抽象基类定义派生类必须重写的方法。 派生类可以随时提供适合派生类型的任意实现代码。

  能够重用代码（即多个派生类共用基类方法的声明和实现代码，无需重写它们）是非抽象基类的优势所在。 因此，如果代码可能由某些或大多数特定 `Publication` 类型共用，则应向 `Publication` 添加成员。 如果无法有效地提供基类实现，则最终将不得不在派生类中提供基本相同的成员实现代码，而不是共用基类中的同一实现代码。 如果需要在多个位置保留重复的代码，可能会导致 bug 出现。

  为了最大限度地提高代码重用性并创建合乎逻辑的直观继承层次结构，需要确保在 `Publication` 类中只添加所有或大多数出版物通用的数据和功能。 然后，派生类可以实现所表示的特定出版物种类的唯一成员。

- 类层次结构的扩展空间大小。 是否要开发包含三个或更多类的层次结构，而不是仅包含一个基类和一个或多个派生类？ 例如， `Publication` 可以是 `Periodical` 的基类，而后者又是 `Magazine`、 `Journal` 和 `Newspaper` 的基类。

  在示例中，将使用包含 `Publication` 类和一个派生类 `Book` 的小型层次结构。 可以轻松扩展此示例，使其可以创建其他许多派生自 `Publication` 的类，如 `Magazine` 和 `Article`。

- 能否实例化基类。 如果不可以，则应向类应用 abstract 关键字。 否则，可通过调用类构造函数来实例化 `Publication` 类。 如果尝试通过直接调用类构造函数来实例化

标记有 `abstract` 关键字的类，则 C# 编译器会生成错误 CS0144：“无法创建抽象类或接口的实例。”如果尝试使用反射进行类实例化，那么反射方法会引发 MemberAccessException。

默认情况下，可以通过调用类构造函数来实例化基类。 无需显式定义类构造函数。 如果基类的源代码中没有类构造函数，C# 编译器会自动提供默认的（无参数）构造函数。

在此示例中，将把 `Publication` 类标记为 `Publication`，使其无法实例化。 一个不具备任何 `abstract` 方法的 `abstract` 类表示该类代表一个在几个具体类（例如 `Book` 和 `Journal`）之间共享的抽象概念。

- 派生类是否必须继承特定成员的基类实现代码、是否能选择重写基类实现代码或者是否必须提供实现代码。 使用 abstract 关键字来强制派生类提供实现代码。 使用 virtual 关键字来允许派生类重写基类方法。 默认情况下，不可重写基类中定义的方法。

  `Publication` 类不具备任何 `abstract` 方法，不过类本身是 `abstract`。

- 派生类是否表示继承层次结构中的最终类，且本身不能用作其他派生类的基类。 默认情况下，任何类都可以用作基类。 可以应用 sealed 关键字来指明类不能用作其他任何类的基类。 尝试派生自密封类生成的编译器错误 CS0509，“无法从密封类型 <typeName> 派生”。

  在此示例中，将把派生类标记为 `sealed`。

以下示例展示了 `Publication` 类的源代码，以及 `Publication.PublicationType` 属性返回的 `PublicationType` 枚举。 除了继承自 Object 的成员之外，`Publication` 类还定义了以下唯一成员和成员重写：

C#

```
public enum PublicationType { Misc, Book, Magazine, Article };

public abstract class Publication
{
    private bool _published = false;
    private DateTime _datePublished;
    private int _totalPages;

    public Publication(string title, string publisher, PublicationType type)
    {
        if (string.IsNullOrWhiteSpace(publisher))
            throw new ArgumentException("The publisher is required.");
        Publisher = publisher;
```

```
        if (string.IsNullOrWhiteSpace(title))
            throw new ArgumentException("The title is required.");
        Title = title;

        Type = type;
    }

    public string Publisher { get; }

    public string Title { get; }

    public PublicationType Type { get; }

    public string? CopyrightName { get; private set; }

    public int CopyrightDate { get; private set; }

    public int Pages
    {
        get { return _totalPages; }
        set
        {
            if (value <= 0)
                throw new ArgumentOutOfRangeException(nameof(value), "The
number of pages cannot be zero or negative.");
            _totalPages = value;
        }
    }

    public string GetPublicationDate()
    {
        if (!_published)
            return "NYP";
        else
            return _datePublished.ToString("d");
    }

    public void Publish(DateTime datePublished)
    {
        _published = true;
        _datePublished = datePublished;
    }

    public void Copyright(string copyrightName, int copyrightDate)
    {
        if (string.IsNullOrWhiteSpace(copyrightName))
            throw new ArgumentException("The name of the copyright holder is
required.");
        CopyrightName = copyrightName;

        int currentYear = DateTime.Now.Year;
        if (copyrightDate < currentYear - 10 || copyrightDate > currentYear
+ 2)
            throw new ArgumentOutOfRangeException($"The copyright year must
be between {currentYear - 10} and {currentYear + 1}");
```

```
        CopyrightDate = copyrightDate;
    }

    public override string ToString() => Title;
}
```

- 构造函数

  由于 `Publication` 类标记有 `abstract`，因此无法直接通过以下代码进行实例化：

  C#

  ```
  var publication = new Publication("Tiddlywinks for Experts", "Fun and
  Games",
                                    PublicationType.Book);
  ```

  不过，它的实例构造函数可以直接通过派生类构造函数进行调用，如 `Book` 类的源代码所示。

- 两个与出版物相关的属性

  `Title` 是只读 String 属性，其值通过调用 `Publication` 构造函数提供。

  `Pages` 是读写 Int32 属性，用于指明出版物的总页数。 值存储在 `totalPages` 私有字段中。 值必须为正数，否则会抛出 ArgumentOutOfRangeException。

- 与出版商相关的成员

  两个只读属性：`Publisher` 和 `Type`。 值最初是通过调用 `Publication` 类构造函数来提供。

- 与出版相关的成员

  两个方法（`Publish` 和 `GetPublicationDate`）用于设置并返回发布日期。 调用时，`Publish` 方法会将 `published` 标志设置为 `true`，并将传递给它的日期作为自变量分配给 `datePublished` 私有字段。 如果 `published` 标志为 `false`，`GetPublicationDate` 方法会返回字符串“NYP”；如果为 `true`，则会返回 `datePublished` 字段的值。

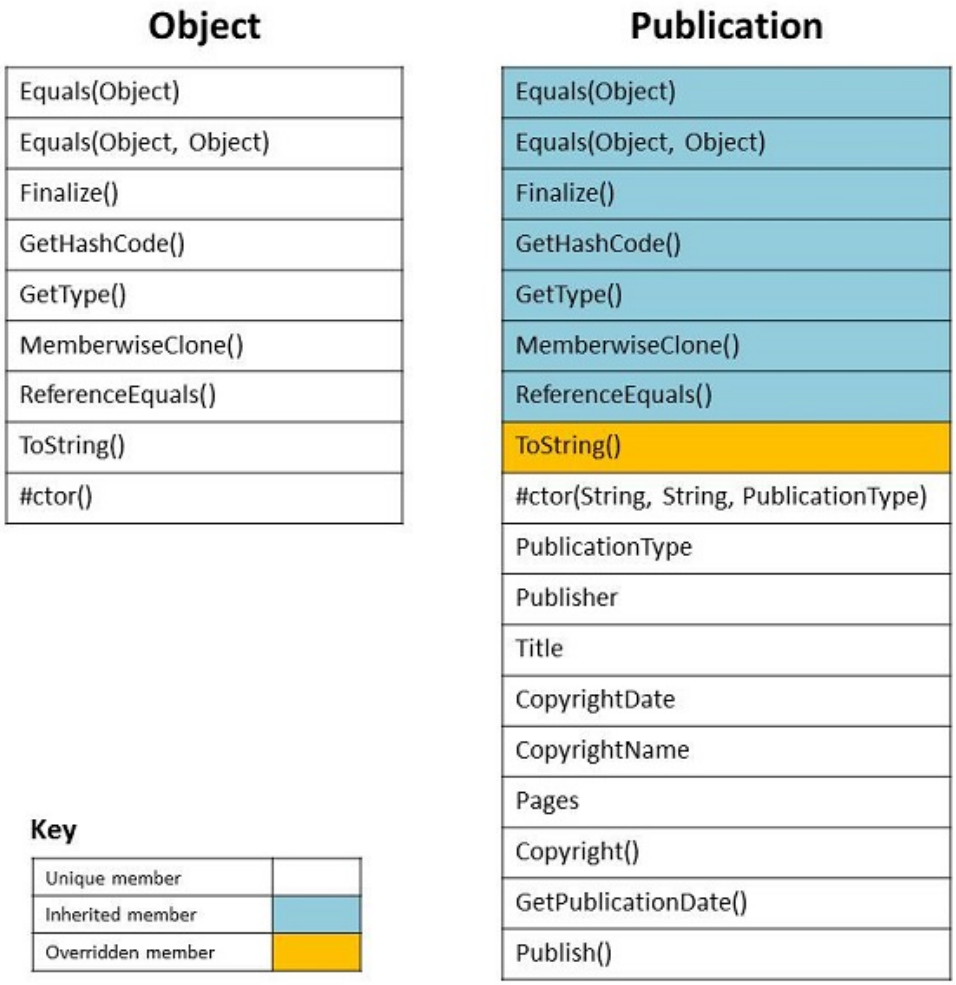
- 与版权相关的成员

  `Copyright` 方法需要将版权所有者的姓名和版权授予年份用作参数，并将它们分配给属性 `CopyrightName` 和 `CopyrightDate`。

- 重写 `ToString` 方法

如果类型不重写 Object.ToString 方法，则返回类型的完全限定的名称，这对于区分实例没什么用。 `Publication` 类将 Object.ToString 重写为返回 `Title` 属性值。

下图展示了基类 `Publication` 及其隐式继承类 Object 之间的关系。



`Book` 类

`Book` 类表示作为一种特定类型出版物的书籍。 下面的示例展示了 `Book` 类的源代码。

C#

```
using System;

public sealed class Book : Publication
{
    public Book(string title, string author, string publisher) :
           this(title, string.Empty, author, publisher)
    { }

    public Book(string title, string isbn, string author, string publisher)
: base(title, publisher, PublicationType.Book)
    {
```

```
        // isbn argument must be a 10- or 13-character numeric string
without "-" characters.
        // We could also determine whether the ISBN is valid by comparing
its checksum digit
        // with a computed checksum.
        //
        if (!string.IsNullOrEmpty(isbn))
        {
            // Determine if ISBN length is correct.
            if (!(isbn.Length == 10 | isbn.Length == 13))
                throw new ArgumentException("The ISBN must be a 10- or 13-
character numeric string.");
            if (!ulong.TryParse(isbn, out _))
                throw new ArgumentException("The ISBN can consist of numeric
characters only.");
        }
        ISBN = isbn;

        Author = author;
    }

    public string ISBN { get; }

    public string Author { get; }

    public decimal Price { get; private set; }

    // A three-digit ISO currency symbol.
    public string? Currency { get; private set; }

    // Returns the old price, and sets a new price.
    public decimal SetPrice(decimal price, string currency)
    {
        if (price < 0)
            throw new ArgumentOutOfRangeException(nameof(price), "The price
cannot be negative.");
        decimal oldValue = Price;
        Price = price;

        if (currency.Length != 3)
            throw new ArgumentException("The ISO currency symbol is a 3-
character string.");
        Currency = currency;

        return oldValue;
    }

    public override bool Equals(object? obj)
    {
        if (obj is not Book book)
            return false;
        else
            return ISBN == book.ISBN;
    }
```

```
    public override int GetHashCode() => ISBN.GetHashCode();

    public override string ToString() => $"{(string.IsNullOrEmpty(Author) ?
"" : Author + ", ")}{Title}";
}
```

除了继承自 `Publication` 的成员之外，`Book` 类还定义了以下唯一成员和成员重写：

- 两个构造函数

  两个 `Book` 构造函数共用三个常见参数。 其中两个参数（*title* 和 *publisher*）对应于构造函数的相应参数。 第三个参数是 *author*，存储在不可变的 属性中。 其中一个构造函数包含存储在 自动属性中的 isbn 参数。

  第一个构造函数使用 this 关键字来调用另一个构造函数。 构造函数链是常见的构造函数定义模式。 调用参数最多的构造函数时，由参数较少的构造函数提供默认值。

  第二个构造函数使用 base 关键字，将标题和出版商名称传递给基类构造函数。 如果没有在源代码中显式调用基类构造函数，那么 C# 编译器会自动提供对基类的默认或无参数构造函数的调用。

- 只读 `ISBN` 属性，用于返回 `Book` 对象的国际标准书号，即 10 位或 13 位的专属编号。 ISBN 作为参数提供给 `Book` 构造函数之一。 ISBN 存储在私有支持字段中，由编译器自动生成。

- 只读 `Author` 属性。 作者姓名作为参数提供给两个 `Book` 构造函数，并存储在属性中。

- 两个与价格相关的只读属性（`Price` 和 `Currency`）。 值作为自变量提供给调用的 `SetPrice` 方法。 `Currency` 属性是三位的 ISO 货币符号（例如，USD 表示美元）。 可以从 ISOCurrencySymbol 属性检索 ISO 货币符号。 这两个属性均为外部只读，但均可在 `Book` 类中由代码设置。

- `SetPrice` 方法，用于设置 `Price` 和 `Currency` 属性的值。 这些值由那些相同属性返回。

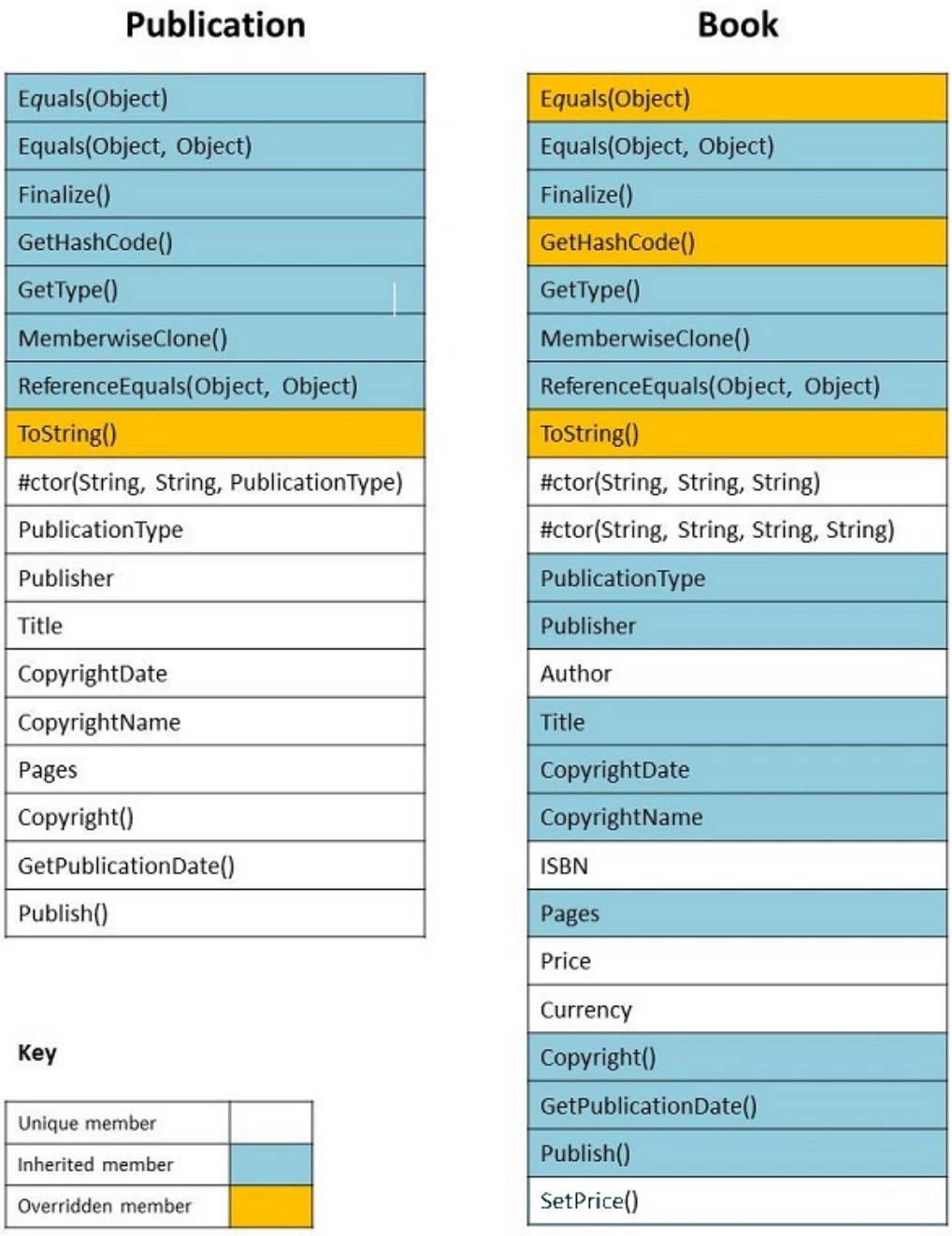
- 重写 `ToString` 方法（继承自 `Publication`）、Object.Equals(Object) 和 GetHashCode 方法（继承自 Object）。

  除非重写，否则 Object.Equals(Object) 方法测试的是引用相等性。 也就是说，两个对象变量必须引用同一个对象，才算相等。 相比之下，在 `Book` 类中，两个 `Book` 对象必须包含相同的 ISBN，才算相等。

  重写 Object.Equals(Object) 方法时，还必须重写 GetHashCode 方法，此方法返回运行时为了实现高效检索，在经哈希处理的集合中存储项所使用的值。 哈希代码应

返回与测试相等性一致的值。 由于已将 Object.Equals(Object) 重写为在两个 `Book` 对象的 ISBN 属性相等时返回 `true`，因此返回的哈希代码是通过调用 `ISBN` 属性返回的字符串的 GetHashCode 方法计算得出。

下图展示了 `Book` 类及其基类 `Publication` 之间的关系。



现在可以实例化 `Book` 对象，调用其唯一成员和继承的成员，并将其作为自变量传递给需要 `Publication` 或 `Book` 类型参数的方法，如以下示例所示。

```
C#
```

```
public class ClassExample
{
    public static void Main()
    {
        var book = new Book("The Tempest", "0971655819", "Shakespeare,
William",
                            "Public Domain Press");
        ShowPublicationInfo(book);
        book.Publish(new DateTime(2016, 8, 18));
        ShowPublicationInfo(book);

        var book2 = new Book("The Tempest", "Classic Works Press",
"Shakespeare, William");
        Console.Write($"{book.Title} and {book2.Title} are the same
publication: " +
                $"{((Publication)book).Equals(book2)}");
    }

    public static void ShowPublicationInfo(Publication pub)
    {
        string pubDate = pub.GetPublicationDate();
        Console.WriteLine($"{pub.Title}, " +
                  $"{(pubDate == "NYP" ? "Not Yet Published" : "published on
" + pubDate):d} by {pub.Publisher}");
    }
}
// The example displays the following output:
//        The Tempest, Not Yet Published by Public Domain Press
//        The Tempest, published on 8/18/2016 by Public Domain Press
//        The Tempest and The Tempest are the same publication: False
```

设计抽象基类及其派生类

在上面的示例中定义了一个基类，它提供了许多方法的实现代码，以便派生类可以共用代码。 然而，在许多情况下，我们并不希望基类提供实现代码。 相反，基类是声明抽象方法的抽象类，用作定义每个派生类必须实现的成员的模板 。 通常情况下，在抽象基类中，每个派生类型的实现代码都是相应类型的专属代码。 尽管该类提供了出版物通用的功能的实现代码，但由于实例化 `Publication` 对象毫无意义，因此，使用 abstract 关键字来标记该类。

例如，每个封闭的二维几何形状都包含两个属性：面积（即形状的内部空间）和周长（或沿形状一周的长度）。 然而，这两个属性的计算方式完全取决于具体的形状。 例如，圆和正方形的周长计算公式就有所不同。 `Shape` 类是一个包含 `abstract` 方法的 `abstract` 类。 这表示派生类共享相同的功能，但这些派生类以不同的方式实现该功能。

以下示例定义了 `Shape` 抽象基类，此基类又定义了两个属性：`Area` 和 `Perimeter`。 除了用 abstract 关键字标记类之外，还需要用 abstract 关键字标记每个实例成员。 在此示例

中，`Shape` 还将 Object.ToString 方法重写为返回类型的名称，而不是其完全限定的名称。 基类还定义了两个静态成员（`GetArea` 和 `GetPerimeter`），以便调用方可以轻松检索任何派生类实例的面积和周长。 将派生类实例传递给两个方法中的任意一个时，运行时调用的是派生类重写的方法。

C#

```
public abstract class Shape
{
    public abstract double Area { get; }

    public abstract double Perimeter { get; }

    public override string ToString() => GetType().Name;

    public static double GetArea(Shape shape) => shape.Area;

    public static double GetPerimeter(Shape shape) => shape.Perimeter;
}
```

然后可以从表示特定形状的 `Shape` 派生一些类。 以下示例定义了三个类：`Square`、`Rectangle` 和 `Circle`。 每个类都使用特定形状的专属公式来计算面积和周长。 一些派生类还定义所表示形状的专属属性（如 `Rectangle.Diagonal` 和 `Circle.Diameter`）。

C#

```
using System;

public class Square : Shape
{
    public Square(double length)
    {
        Side = length;
    }

    public double Side { get; }

    public override double Area => Math.Pow(Side, 2);

    public override double Perimeter => Side * 4;

    public double Diagonal => Math.Round(Math.Sqrt(2) * Side, 2);
}

public class Rectangle : Shape
{
    public Rectangle(double length, double width)
    {
        Length = length;
        Width = width;
```

```
    }

    public double Length { get; }

    public double Width { get; }

    public override double Area => Length * Width;

    public override double Perimeter => 2 * Length + 2 * Width;

    public bool IsSquare() => Length == Width;

    public double Diagonal => Math.Round(Math.Sqrt(Math.Pow(Length, 2) +
Math.Pow(Width, 2)), 2);
}

public class Circle : Shape
{
    public Circle(double radius)
    {
        Radius = radius;
    }

    public override double Area => Math.Round(Math.PI * Math.Pow(Radius, 2),
2);

    public override double Perimeter => Math.Round(Math.PI * 2 * Radius, 2);

    // Define a circumference, since it's the more familiar term.
    public double Circumference => Perimeter;

    public double Radius { get; }

    public double Diameter => Radius * 2;
}
```

以下示例使用派生自 `Shape` 的对象。 它实例化派生自 `Shape` 的一组对象，然后调用 `Shape` 类的静态方法，用于包装返回的 `Shape` 属性值。 运行时从派生类型的重写属性检索值。 以下示例还将数组中的每个 `Shape` 对象显式转换成其派生类型；如果显式转换成功，则检索 `Shape` 的特定子类的属性。

C#

```
using System;

public class Example
{
    public static void Main()
    {
        Shape[] shapes = { new Rectangle(10, 12), new Square(5),
                           new Circle(3) };
        foreach (Shape shape in shapes)
```

```
        {
            Console.WriteLine($"{shape}: area, {Shape.GetArea(shape)}; " +
                              $"perimeter, {Shape.GetPerimeter(shape)}");
            if (shape is Rectangle rect)
            {
                Console.WriteLine($"   Is Square: {rect.IsSquare()},
Diagonal: {rect.Diagonal}");
                continue;
            }
            if (shape is Square sq)
            {
                Console.WriteLine($"   Diagonal: {sq.Diagonal}");
                continue;
            }
        }
    }
}
// The example displays the following output:
//         Rectangle: area, 120; perimeter, 44
//            Is Square: False, Diagonal: 15.62
//         Square: area, 25; perimeter, 20
//            Diagonal: 7.07
//         Circle: area, 28.27; perimeter, 18.85
```

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

项目 • 2023/04/07

由于是多态对象，基类类型的变量可以保存派生类型。 要访问派生类型的实例成员，必须将值强制转换回派生类型。 但是，强制转换会引发 InvalidCastException 风险。 C# 提供模式匹配语句，该语句只有在成功时才会有条件地执行强制转换。 C# 还提供 is 和 as 运算符来测试值是否属于特定类型。

下面的示例演示如何使用模式匹配 `is` 语句：

C#

```
var g = new Giraffe();
var a = new Animal();
FeedMammals(g);
FeedMammals(a);
// Output:
// Eating.
// Animal is not a Mammal

SuperNova sn = new SuperNova();
TestForMammals(g);
TestForMammals(sn);

static void FeedMammals(Animal a)
{
    if (a is Mammal m)
    {
        m.Eat();
    }
    else
    {
        // variable 'm' is not in scope here, and can't be used.
        Console.WriteLine($"{a.GetType().Name} is not a Mammal");
    }
}

static void TestForMammals(object o)
{
    // You also can use the as operator and test for null
    // before referencing the variable.
    var m = o as Mammal;
    if (m != null)
    {
        Console.WriteLine(m.ToString());
    }
    else
    {
```

```
        Console.WriteLine($"{o.GetType().Name} is not a Mammal");
    }
}
// Output:
// I am an animal.
// SuperNova is not a Mammal

class Animal
{
    public void Eat() { Console.WriteLine("Eating."); }
    public override string ToString()
    {
        return "I am an animal.";
    }
}
class Mammal : Animal { }
class Giraffe : Mammal { }

class SuperNova { }
```

前面的示例演示了模式匹配语法的一些功能。 `if (a is Mammal m)` 语句将测试与初始化赋值相结合。 只有在测试成功时才会进行赋值。 变量 `m` 仅在已赋值的嵌入式 `if` 语句的范围内。 之后无法在同一方法中访问 `m`。 前面的示例还演示了如何使用 as 运算符将对象转换为指定类型。

也可以使用同一语法来测试可为 null 的值类型是否具有值，如以下示例所示：

C#

```
int i = 5;
PatternMatchingNullable(i);

int? j = null;
PatternMatchingNullable(j);

double d = 9.78654;
PatternMatchingNullable(d);

PatternMatchingSwitch(i);
PatternMatchingSwitch(j);
PatternMatchingSwitch(d);

static void PatternMatchingNullable(ValueType? val)
{
    if (val is int j) // Nullable types are not allowed in patterns
    {
        Console.WriteLine(j);
    }
    else if (val is null) // If val is a nullable type with no value, this
expression is true
    {
        Console.WriteLine("val is a nullable type with the null value");
```

```
    }
    else
    {
        Console.WriteLine("Could not convert " + val.ToString());
    }
}

static void PatternMatchingSwitch(ValueType? val)
{
    switch (val)
    {
        case int number:
            Console.WriteLine(number);
            break;
        case long number:
            Console.WriteLine(number);
            break;
        case decimal number:
            Console.WriteLine(number);
            break;
        case float number:
            Console.WriteLine(number);
            break;
        case double number:
            Console.WriteLine(number);
            break;
        case null:
            Console.WriteLine("val is a nullable type with the null value");
            break;
        default:
            Console.WriteLine("Could not convert " + val.ToString());
            break;
    }
}
```

前面的示例演示了模式匹配用于转换的其他功能。 可以通过专门检查 `null` 值来测试 NULL 模式的变量。 当变量的运行时值为 `null` 时，用于检查类型的 `is` 语句始终返回 `false`。 模式匹配 `is` 语句不允许可以为 null 值的类型，如 `int?` 或 `Nullable<int>`，但你可以测试任何其他值类型。 上述示例中的 `is` 模式不局限于可为空的值类型。 也可以使用这些模式测试引用类型的变量具有值还是为 `null`。

前面的示例还演示如何在变量为其他类型的 `switch` 语句中使用类型模式。

如果需要测试变量是否为给定类型，但不将其分配给新变量，则可以对引用类型和可以为 null 的值类型使用 `is` 和 `as` 运算符。 以下代码演示如何在引入模式匹配以测试变量是否为给定类型前，使用 C# 语言中的 `is` 和 `as` 语句：

```
C#
```

```
// Use the is operator to verify the type.
// before performing a cast.
Giraffe g = new();
UseIsOperator(g);

// Use the as operator and test for null
// before referencing the variable.
UseAsOperator(g);

// Use pattern matching to test for null
// before referencing the variable
UsePatternMatchingIs(g);

// Use the as operator to test
// an incompatible type.
SuperNova sn = new();
UseAsOperator(sn);

// Use the as operator with a value type.
// Note the implicit conversion to int? in
// the method body.
int i = 5;
UseAsWithNullable(i);

double d = 9.78654;
UseAsWithNullable(d);

static void UseIsOperator(Animal a)
{
    if (a is Mammal)
    {
        Mammal m = (Mammal)a;
        m.Eat();
    }
}

static void UsePatternMatchingIs(Animal a)
{
    if (a is Mammal m)
    {
        m.Eat();
    }
}

static void UseAsOperator(object o)
{
    Mammal? m = o as Mammal;
    if (m is not null)
    {
        Console.WriteLine(m.ToString());
    }
    else
    {
        Console.WriteLine($"{o.GetType().Name} is not a Mammal");
```

```
    }
}

static void UseAsWithNullable(System.ValueType val)
{
    int? j = val as int?;
    if (j is not null)
    {
        Console.WriteLine(j);
    }
    else
    {
        Console.WriteLine("Could not convert " + val.ToString());
    }
}
class Animal
{
    public void Eat() => Console.WriteLine("Eating.");
    public override string ToString() => "I am an animal.";
}
class Mammal : Animal { }
class Giraffe : Mammal { }

class SuperNova { }
```

正如你所看到的，将此代码与模式匹配代码进行比较，模式匹配语法通过在单个语句中结合测试和赋值来提供更强大的功能。 尽量使用模式匹配语法。

教程：使用模式匹配来生成类型驱动和数据驱动的算法

项目 • 2023/05/10

可以编写行为类似于扩展其他库中可能有的类型的功能。 模式的另一个用途是，创建应用程序需要的功能，但此功能不是要扩展的类型的基本功能。

本教程介绍以下操作：

- ✓ 识别应使用模式匹配的情况。
- ✓ 使用模式匹配表达式来实现基于类型和属性值的行为。
- ✓ 结合使用模式匹配和其他方法来创建完整算法。

先决条件

- 建议使用 Visual Studio for Windows 或 Mac。 可以从 Visual Studio 下载页面 下载免费版本。 Visual Studio 包括 .NET SDK。
- 还可以使用 Visual Studio Code 编辑器。 需要单独安装最新的 .NET Compiler Platform SDK。
- 如果更想要使用其他编辑器，则需要安装最新的 .NET SDK。

本教程假设你熟悉 C# 和 .NET，包括 Visual Studio 或 .NET CLI。

模式匹配方案

新式开发通常包括从多个源集成数据，并在一个整体应用程序中呈现以相应数据为依据的信息和见解。 你和你的团队无法控制或访问表示传入数据的所有类型。

面向对象的经典设计要求，在应用程序中创建表示多个数据源中的所有数据类型的类型。 然后，应用程序便能处理这些新类型、生成继承层次结构、创建虚拟方法并实现抽象。 这些技术起作用，而且有时候是最佳工具。 不过，在其他时候，你可以编写更少的代码。 可使用将数据与管理相应数据的操作分离开来的技术，编写更明确的代码。

在本教程中，你将创建并探索从一个方案的多个外部源中提取传入数据的应用程序。 你将看到，模式匹配如何通过原始系统中没有的方式高效地使用和处理相应数据。

假设某大都市区通过通行费和高峰时段定价来管理交通。 你编写的应用程序根据车辆类型来计算车辆通行费。 后续增强功能包括，定价因车内乘客数而异。 进一步增强功能包括，定价因时间和周几而异。

通过上述简要说明，你可能已快速勾勒出用于对此系统进行建模的对象层次结构。 不过，数据来自多个源，如其他车辆注册管理系统。 这些系统提供不同的类来对相应数据进行建模，而你连可使用的一个对象模型都没有。 在本教程中，你将使用这些简化后的类，对这些外部系统中的车辆数据进行建模，如下面的代码所示：

C#

```
namespace ConsumerVehicleRegistration
{
    public class Car
    {
        public int Passengers { get; set; }
    }
}

namespace CommercialRegistration
{
    public class DeliveryTruck
    {
        public int GrossWeightClass { get; set; }
    }
}

namespace LiveryRegistration
{
    public class Taxi
    {
        public int Fares { get; set; }
    }

    public class Bus
    {
        public int Capacity { get; set; }
        public int Riders { get; set; }
    }
}
```

若要下载起始代码，可以访问 dotnet/samples GitHub 存储库。 可以看到，车辆类来自不同的系统，且位于不同的命名空间中。 没有常见基类，可使用的 `System.Object` 除外。

模式匹配设计

本教程中使用的方案重点介绍了非常适合适用模式匹配解决的问题类型：

- 需要使用的对象不在匹配目标的对象层次结构中。 可能要使用属于不相关系统的类。

- 要添加的功能不属于这些类的核心抽象。 车辆通行费因不同车辆类型而异，但通行费不是车辆的核心功能。

如果不一起描述数据形状和对相应数据执行的操作，C# 中的模式匹配功能可以简化这一切。

实现基本通行费计算

最基本的通行费计算仅依赖车辆类型：

- `Car` 的通行费为 2.00 美元。
- `Taxi` 的通行费为 3.50 美元。
- `Bus` 的通行费为 5.00 美元。
- `DeliveryTruck` 的通行费为 10.00 美元

新建 `TollCalculator` 类，并对车辆类型实现模式匹配，以获取通行费金额。 以下代码显示了 `TollCalculator` 的初始实现。

C#

```
using System;
using CommercialRegistration;
using ConsumerVehicleRegistration;
using LiveryRegistration;

namespace Calculators;

public class TollCalculator
{
    public decimal CalculateToll(object vehicle) =>
        vehicle switch
    {
        Car c           => 2.00m,
        Taxi t          => 3.50m,
        Bus b           => 5.00m,
        DeliveryTruck t => 10.00m,
        { }             => throw new ArgumentException(message: "Not a known
vehicle type", paramName: nameof(vehicle)),
        null            => throw new ArgumentNullException(nameof(vehicle))
    };
}
```

上面的代码使用测试声明模式的 switch 表达式（与 switch 语句不同）。 switch 表达式以变量（上面代码中的 `vehicle`）开头，后跟 `switch` 关键字。 接下来是大括号内的所有 switch 臂。 `switch` 表达式对 `switch` 语句周围的语法进行了其他优化。 不仅省略了 `case` 关键字，还让每个臂的结果成为表达式。 最后两个臂展示了一种新语言功能。 `{ }`

子句匹配与之前的臂不匹配的任何非 null 对象。 此臂捕获传递到这个方法的所有不正确类型。 `{ }` 事例必须遵循每种车辆类型的情况。 如果订单被撤销，则 `{ }` 事例优先。 最后，`null`常量模式检测何时将 `null` 传递给此方法。 `null` 模式可以是最后一个，因为其他模式仅匹配正确类型的非 NULL 对象。

可使用 `Program.cs` 中的以下代码来测试上面的代码：

C#

```
using System;
using CommercialRegistration;
using ConsumerVehicleRegistration;
using LiveryRegistration;

using toll_calculator;

var tollCalc = new TollCalculator();

var car = new Car();
var taxi = new Taxi();
var bus = new Bus();
var truck = new DeliveryTruck();

Console.WriteLine($"The toll for a car is {tollCalc.CalculateToll(car)}");
Console.WriteLine($"The toll for a taxi is {tollCalc.CalculateToll(taxi)}");
Console.WriteLine($"The toll for a bus is {tollCalc.CalculateToll(bus)}");
Console.WriteLine($"The toll for a truck is
{tollCalc.CalculateToll(truck)}");

try
{
    tollCalc.CalculateToll("this will fail");
}
catch (ArgumentException e)
{
    Console.WriteLine("Caught an argument exception when using the wrong
type");
}
try
{
    tollCalc.CalculateToll(null!);
}
catch (ArgumentNullException e)
{
    Console.WriteLine("Caught an argument exception when using null");
}
```

此代码虽然包含在初学者项目中，但已被注释掉。删除注释即可测试已编写的代码。

你正开始了解，模式如何有助于创建将代码和数据分离开来的算法。 `switch` 表达式测试类型，并根据结果生成不同的值。 这仅仅是开始。

添加因乘客数而异的定价

通行费收取机构希望鼓励车辆以最大载客量出行。 他们已决定，对乘客数较少的车辆收取更多费用，并通过更低定价来鼓励车辆乘客满员：

- 没有乘客的汽车和出租车需额外支付 0.50 美元。
- 载有两名乘客的汽车和出租车可享受 0.50 美元折扣。
- 载有三名或更多乘客的汽车和出租车可享受 1.00 美元折扣。
- 乘客数不到满载量 50% 的巴士需额外支付 2.00 美元。
- 乘客数超过满载量 90% 的巴士可享受 1.00 美元折扣。

可使用属性模式在同一 switch 表达式中实现这些规则。 属性模式将属性值与常数值进行比较。 属性模式在类型已确定后检查对象的属性。 `Car` 的一个子句扩展为四个不同的子句：

C#

```
vehicle switch
{
    Car {Passengers: 0} => 2.00m + 0.50m,
    Car {Passengers: 1} => 2.0m,
    Car {Passengers: 2} => 2.0m - 0.50m,
    Car                 => 2.00m - 1.0m,

    // ...
};
```

前三个子句测试类型 `Car`，然后检查 `Passengers` 属性的值。 如果两个条件都匹配，系统便会计算并返回相应表达式。

还可以类似方式扩展出租车的子句：

C#

```
vehicle switch
{
    // ...

    Taxi {Fares: 0}  => 3.50m + 1.00m,
    Taxi {Fares: 1}  => 3.50m,
    Taxi {Fares: 2}  => 3.50m - 0.50m,
    Taxi             => 3.50m - 1.00m,

    // ...
};
```

接下来，通过扩展巴士的子句来实现载客量规则，如下面的示例所示：

C#

```
vehicle switch
{
    // ...

    Bus b when ((double)b.Riders / (double)b.Capacity) < 0.50 => 5.00m +
2.00m,
    Bus b when ((double)b.Riders / (double)b.Capacity) > 0.90 => 5.00m -
1.00m,
    Bus => 5.00m,

    // ...
};
```

通行费收取机构并不关注运货卡车中的乘客数。 相反，它们根据卡车的重量级别调整通行费金额，如下所示：

- 超过 5000 磅的运货卡车需额外支付 5.00 美元。
- 3000 磅以下的轻型卡车可享受 2.00 美元折扣。

此规则通过以下代码实现：

C#

```
vehicle switch
{
    // ...

    DeliveryTruck t when (t.GrossWeightClass > 5000) => 10.00m + 5.00m,
    DeliveryTruck t when (t.GrossWeightClass < 3000) => 10.00m - 2.00m,
    DeliveryTruck => 10.00m,
};
```

以上代码展示了 switch 臂的 `when` 子句。 `when` 子句用于对属性测试条件（相等性除外）。 完成后的方法如以下代码所示：

C#

```
vehicle switch
{
    Car {Passengers: 0}        => 2.00m + 0.50m,
    Car {Passengers: 1}        => 2.0m,
    Car {Passengers: 2}        => 2.0m - 0.50m,
    Car                        => 2.00m - 1.0m,

    Taxi {Fares: 0}  => 3.50m + 1.00m,
    Taxi {Fares: 1}  => 3.50m,
    Taxi {Fares: 2}  => 3.50m - 0.50m,
    Taxi             => 3.50m - 1.00m,
```

```
    Bus b when ((double)b.Riders / (double)b.Capacity) < 0.50 => 5.00m +
2.00m,
    Bus b when ((double)b.Riders / (double)b.Capacity) > 0.90 => 5.00m -
1.00m,
    Bus => 5.00m,

    DeliveryTruck t when (t.GrossWeightClass > 5000) => 10.00m + 5.00m,
    DeliveryTruck t when (t.GrossWeightClass < 3000) => 10.00m - 2.00m,
    DeliveryTruck => 10.00m,

    { }     => throw new ArgumentException(message: "Not a known vehicle
type", paramName: nameof(vehicle)),
    null    => throw new ArgumentNullException(nameof(vehicle))
};
```

其中许多 switch 臂都是递归模式示例。 例如，`Car { Passengers: 1}` 表明属性模式内有常量模式。

可使用嵌套的 switch 来减少此代码中重复的地方。 在上面的示例中，`Car` 和 `Taxi` 都有四个不同的臂。 在这两种案例中，都可创建向常量模式馈送数据的声明模式。 下面的代码展示了这项技术：

C#

```
public decimal CalculateToll(object vehicle) =>
    vehicle switch
    {
        Car c => c.Passengers switch
        {
            0 => 2.00m + 0.5m,
            1 => 2.0m,
            2 => 2.0m - 0.5m,
            _ => 2.00m - 1.0m
        },

        Taxi t => t.Fares switch
        {
            0 => 3.50m + 1.00m,
            1 => 3.50m,
            2 => 3.50m - 0.50m,
            _ => 3.50m - 1.00m
        },

        Bus b when ((double)b.Riders / (double)b.Capacity) < 0.50 => 5.00m +
2.00m,
        Bus b when ((double)b.Riders / (double)b.Capacity) > 0.90 => 5.00m -
1.00m,
        Bus b => 5.00m,

        DeliveryTruck t when (t.GrossWeightClass > 5000) => 10.00m + 5.00m,
        DeliveryTruck t when (t.GrossWeightClass < 3000) => 10.00m - 2.00m,
```

```
        DeliveryTruck t => 10.00m,

        { }  => throw new ArgumentException(message: "Not a known vehicle
type", paramName: nameof(vehicle)),
        null => throw new ArgumentNullException(nameof(vehicle))
    };
```

在上面的示例中，使用递归表达式意味着不用重复包含测试属性值的子臂的 `Car` 和 `Taxi` 臂。 此技术不适用于 `Bus` 和 `DeliveryTruck` 臂，因为这些臂测试的是属性范围，而不是离散值。

添加高峰时段定价

对于最后一个功能，通行费收取机构希望添加有时效性的高峰时段定价。 在早晚高峰时段，通行费翻倍。 此规则只影响一个方向的交通：早高峰时段入城和晚高峰时段出城。 在工作日的其他时段，通行费增加 50%。 在深夜和清晨，通行费减少 25%。 在周末，无论什么时间，都按正常费率收费。 你可以借助以下代码使用 `if` 和 `else` 语句系列表达此内容：

C#

```
public decimal PeakTimePremiumIfElse(DateTime timeOfToll, bool inbound)
{
    if ((timeOfToll.DayOfWeek == DayOfWeek.Saturday) ||
        (timeOfToll.DayOfWeek == DayOfWeek.Sunday))
    {
        return 1.0m;
    }
    else
    {
        int hour = timeOfToll.Hour;
        if (hour < 6)
        {
            return 0.75m;
        }
        else if (hour < 10)
        {
            if (inbound)
            {
                return 2.0m;
            }
            else
            {
                return 1.0m;
            }
        }
        else if (hour < 16)
        {
            return 1.5m;
```

```
        }
        else if (hour < 20)
        {
            if (inbound)
            {
                return 1.0m;
            }
            else
            {
                return 2.0m;
            }
        }
        else // Overnight
        {
            return 0.75m;
        }
    }
}
```

前面的代码可以正常工作，但无法读取。 必须链接所有输入事例和嵌套的 `if` 语句，才能对代码进行推理。 相反，虽然将对此功能使用模式匹配，但要将它与其他技术集成。 可以生成一个模式匹配表达式，将方向、周几和时间所有这一切都考虑在内。 生成的结果是一个复杂的表达式。 它既难读取，也难理解。 这就加大了确保正确性的难度。 请改为将这些方法合并为生成值元组，用于简要描述所有这些状态。 然后，使用模式匹配来计算通行费乘数。 元组包含以下三个离散条件：

- 是工作日，还是周末。
- 收取通行费时所处的时间带区。
- 方向是入城，还是出城

下表展示了输入值和高峰时段定价乘数的组合：

日期	时间	方向	高级
星期	早高峰	入城	x 2.00
星期	早高峰	出城	x 1.00
星期	日间	入城	x 1.50
星期	日间	出城	x 1.50
星期	晚高峰	入城	x 1.00
星期	晚高峰	出城	x 2.00
星期	夜间	入城	x 0.75
星期	夜间	出城	x 0.75

日期	时间	方向	高级
周末	早高峰	入城	x 1.00
周末	早高峰	出城	x 1.00
周末	日间	入城	x 1.00
周末	日间	出城	x 1.00
周末	晚高峰	入城	x 1.00
周末	晚高峰	出城	x 1.00
周末	夜间	入城	x 1.00
周末	夜间	出城	x 1.00

三个变量有 16 种不同的组合。 通过结合某些条件，将能简化最终的 switch 表达式。

通行费收取系统在收取通行费时对时间使用 DateTime 结构。 生成根据上表创建变量的成员方法。 以下函数用作模式匹配 switch 表达式，以表示 DateTime 是表示周末，还是表示工作日：

C#

```
private static bool IsWeekDay(DateTime timeOfToll) =>
    timeOfToll.DayOfWeek switch
    {
        DayOfWeek.Monday    => true,
        DayOfWeek.Tuesday   => true,
        DayOfWeek.Wednesday => true,
        DayOfWeek.Thursday  => true,
        DayOfWeek.Friday    => true,
        DayOfWeek.Saturday  => false,
        DayOfWeek.Sunday    => false
    };
```

此方法虽然正确，但是具有重复性。 可以简化它，如下面的代码所示：

C#

```
private static bool IsWeekDay(DateTime timeOfToll) =>
    timeOfToll.DayOfWeek switch
    {
        DayOfWeek.Saturday => false,
        DayOfWeek.Sunday => false,
        _ => true
    };
```

接下来，添加将时间分类为块的类似函数：

C#

```
private enum TimeBand
{
    MorningRush,
    Daytime,
    EveningRush,
    Overnight
}

private static TimeBand GetTimeBand(DateTime timeOfToll) =>
    timeOfToll.Hour switch
    {
        < 6 or > 19 => TimeBand.Overnight,
        < 10 => TimeBand.MorningRush,
        < 16 => TimeBand.Daytime,
        _ => TimeBand.EveningRush,
    };
```

添加将每个时间范围转换为离散值的专用 `enum`。 然后，`GetTimeBand` 方法使用关系模式和合取 or 模式，两者都已添加到 C# 9.0 中。 通过关系模式，可使用 `<`、`>`、`<=` 或 `>=` 来测试数值。 `or` 模式测试表达式是否与一个或多个模式匹配。 还可以使用 `and` 模式来确保表达式匹配两个不同的模式，并使用 `not` 模式来测试表达式是否与模式不匹配。

创建这些方法后，可以结合使用另一个 `switch` 表达式和元组模式，以计算定价附加费。 可以生成包含所有 16 个臂的 `switch` 表达式：

C#

```
public decimal PeakTimePremiumFull(DateTime timeOfToll, bool inbound) =>
    (IsWeekDay(timeOfToll), GetTimeBand(timeOfToll), inbound) switch
    {
        (true, TimeBand.MorningRush, true) => 2.00m,
        (true, TimeBand.MorningRush, false) => 1.00m,
        (true, TimeBand.Daytime, true) => 1.50m,
        (true, TimeBand.Daytime, false) => 1.50m,
        (true, TimeBand.EveningRush, true) => 1.00m,
        (true, TimeBand.EveningRush, false) => 2.00m,
        (true, TimeBand.Overnight, true) => 0.75m,
        (true, TimeBand.Overnight, false) => 0.75m,
        (false, TimeBand.MorningRush, true) => 1.00m,
        (false, TimeBand.MorningRush, false) => 1.00m,
        (false, TimeBand.Daytime, true) => 1.00m,
        (false, TimeBand.Daytime, false) => 1.00m,
        (false, TimeBand.EveningRush, true) => 1.00m,
        (false, TimeBand.EveningRush, false) => 1.00m,
        (false, TimeBand.Overnight, true) => 1.00m,
```

```
        (false, TimeBand.Overnight, false) => 1.00m,
    };
```

上面的代码虽起作用，但可以进行简化。 周末对应的所有八个组合的通行费都相同。 可以将所有八个组合都替换为下面的代码：

C#

```
(false, _, _) => 1.0m,
```

入城和出城交通乘数在工作日日间和夜间时段都相同。 可以将四个 switch 臂替换为以下两行代码：

C#

```
(true, TimeBand.Overnight, _) => 0.75m,
(true, TimeBand.Daytime, _)   => 1.5m,
```

执行这两项更改后，代码应如下所示：

C#

```
public decimal PeakTimePremium(DateTime timeOfToll, bool inbound) =>
    (IsWeekDay(timeOfToll), GetTimeBand(timeOfToll), inbound) switch
    {
        (true, TimeBand.MorningRush, true)  => 2.00m,
        (true, TimeBand.MorningRush, false) => 1.00m,
        (true, TimeBand.Daytime,     _)     => 1.50m,
        (true, TimeBand.EveningRush, true)  => 1.00m,
        (true, TimeBand.EveningRush, false) => 2.00m,
        (true, TimeBand.Overnight,   _)     => 0.75m,
        (false, _,                   _)     => 1.00m,
    };
```

最后，可以删除通行费正常的两个高峰时段。 删除这些臂后，可以在最后的 switch 臂中将 `false` 替换为弃元 (`_`)。 完成的方法如下所示：

C#

```
public decimal PeakTimePremium(DateTime timeOfToll, bool inbound) =>
    (IsWeekDay(timeOfToll), GetTimeBand(timeOfToll), inbound) switch
    {
        (true, TimeBand.Overnight, _) => 0.75m,
        (true, TimeBand.Daytime, _) => 1.5m,
        (true, TimeBand.MorningRush, true) => 2.0m,
        (true, TimeBand.EveningRush, false) => 2.0m,
```

```
        _ => 1.0m,
    };
```

此示例突出了模式匹配的一个优点：模式分支是依序计算的。 如果将它们重排为更早的分支处理后续事例之一，编译器便会提示你无法访问的代码。 借助这些语言规则，可以更轻松地执行前面的简化，同时确信代码未更改。

模式匹配使某些类型的代码更具可读性，并且当你无法向类添加代码时，它会提供面向对象技术的替代方法。 云会导致数据和功能分离。 数据形状和对相应数据执行的操作不一定在一起进行描述。 在本教程中，你通过与原始功能完全不同的方法使用了现有数据。 使用模式匹配，可以覆盖这些类型编写功能，即使无法扩展类型，也不例外。

后续步骤

若要下载完成后的代码，可以访问 dotnet/samples GitHub 存储库。 请自行探索模式，并将此技术纳入你的常规编码活动。 学些这些技术，你可以通过其他方式来处理问题和新建功能。

另请参阅

- 模式
- switch表达式

如何使用 try/catch 处理异常

项目 • 2023/04/07

try-catch 块的用途是捕获并处理工作代码产生的异常。 某些异常可以在 `catch` 块中进行处理，问题得以解决并不再出现异常；但是，大多数情况下你唯一可做的是确保引发的异常是合理异常。

示例

在此示例中，IndexOutOfRangeException 不是最合理的异常：ArgumentOutOfRangeException 对于此方法来说更有意义，因为此错误是由调用方传递的 `index` 参数引起的。

C#

```
static int GetInt(int[] array, int index)
{
    try
    {
        return array[index];
    }
    catch (IndexOutOfRangeException e)  // CS0168
    {
        Console.WriteLine(e.Message);
        // Set IndexOutOfRangeException to the new exception's
InnerException.
        throw new ArgumentOutOfRangeException("index parameter is out of
range.", e);
    }
}
```

注释

引发异常的代码包含在 `try` 块中。 在此块后面紧挨着添加 `catch` 语句以处理 `IndexOutOfRangeException` 异常（如果发生此异常）。 `catch` 块处理 `IndexOutOfRangeException` 并改为引发更合理的 `ArgumentOutOfRangeException`。 为了向调用方提供尽可能多的信息，请考虑将原始异常指定为新异常的 InnerException。 因为 InnerException 属性为 read-only，所以必须在新异常的构造函数中指定此属性。

如何使用 finally 执行清理代码

项目 • 2024/03/03

`finally` 语句的用途是确保立即进行对象（通常是容纳外部资源的对象）的必要清理（即使引发异常）。 这类清理的一个示例是在使用之后立即对 FileStream 调用 Close（而不是等待公共语言运行时对对象进行垃圾回收），如下所示：

C#

```
static void CodeWithoutCleanup()
{
    FileStream? file = null;
    FileInfo fileInfo = new FileInfo("./file.txt");

    file = fileInfo.OpenWrite();
    file.WriteByte(0xF);

    file.Close();
}
```

示例

若要将上面的代码转换为 `try-catch-finally` 语句，可将清理代码与工作代码分开，如下所示。

C#

```
static void CodeWithCleanup()
{
    FileStream? file = null;
    FileInfo? fileInfo = null;

    try
    {
        fileInfo = new FileInfo("./file.txt");

        file = fileInfo.OpenWrite();
        file.WriteByte(0xF);
    }
    catch (UnauthorizedAccessException e)
    {
        Console.WriteLine(e.Message);
    }
    finally
    {
        file?.Close();
```

```
    }
}
```

因为可能会在 `try` 块中进行 `OpenWrite()` 调用之前的任何时候发生异常，或 `OpenWrite()` 调用本身可能会失败，所以我们不保证在尝试关闭文件时文件处于打开状态。 `finally` 块添加了一个检查，以便在调用 Close 方法之前确保 FileStream 对象不是 `null`。 如果不进行 `null` 检查，则 `finally` 块可能会引发自己的 NullReferenceException，但是在可能时应避免在 `finally` 块中引发异常。

数据库连接是在 `finally` 块中进行关闭的另一个很好的候选项。 因为与数据库服务器之间的允许连接数有时会受到限制，所以应尽快关闭数据库连接。 如果在可以关闭连接之前引发异常，则使用 `finally` 块比等待垃圾回收更好。

另请参阅

- using 语句
- 异常处理语句

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 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# 13 中的新增功能

项目 • 2024/12/31

C# 13 包括以下新功能。 可以使用最新的 Visual Studio 2022 版本或 .NET 9 SDK 来尝试这些功能：

- params 集合
- 新的 lock 类型和语义。
- 新的转义序列 - \e。
- 方法组自然类型改进
- 对象初始值设定项中的隐式索引器访问
- 在迭代器和异步方法中启用 ref 局部变量和 unsafe 上下文
- 启用 ref struct 类型来实现接口。
- 允许在泛型中将 ref 结构类型 作为类型参数的参数。
- 分部属性和索引器 现在可在 `partial` 类型中使用。
- 重载解析优先级 允许库作者将一个重载指定为比其他重载更好。

从 Visual Studio 17.12 开始，C# 13 将 field 上下文关键字作为预览功能包含在内。

C# 13 支持在 **.NET 9**上。 有关详细信息，请参阅 C# 语言版本控制。

可以从 .NET 下载页 下载最新的 .NET 9 SDK。 还可以下载 Visual Studio 2022，其中包括 .NET 9 SDK。

在公共预览版中提供新功能时，新功能将添加到“C# 中的新增功能”页。 roslyn 功能状态页 的工作集 部分跟踪即将推出的功能何时合并到主分支中。

可以在我们关于重大更改的文章中找到 C# 13 中引入的任何重大更改。

> **备注**
>
> 我们对这些功能的反馈感兴趣。 如果发现这些新功能存在问题，请在 dotnet/roslyn 存储库中创建**新问题**。

`params` 集合

`params` 修饰符不限于数组类型。 现在可以将 `params` 用于任何已识别的集合类型，包括 System.Span<T>、System.ReadOnlySpan<T>，以及那些实现 System.Collections.Generic.IEnumerable<T> 并具有 `Add` 方法的类型。 除了具体类型外，还可以使用接口 System.Collections.Generic.IEnumerable<T>、System.Collections.Generic.IReadOnlyCollection<T>、

System.Collections.Generic.IReadOnlyList<T>、System.Collections.Generic.ICollection<T>和 System.Collections.Generic.IList<T>。

使用接口类型时，编译器会合成所提供的参数的存储。 可以在 params 集合的功能规范中了解更多信息。

例如，方法声明可以将范围声明为 `params` 参数：

```C#
public void Concat<T>(params ReadOnlySpan<T> items)
{
    for (int i = 0; i < items.Length; i++)
    {
        Console.Write(items[i]);
        Console.Write(" ");
    }
    Console.WriteLine();
}
```

新建锁定对象

.NET 9 运行时包括线程同步的新类型，即 System.Threading.Lock 类型。 此类型通过其 API 提供更好的线程同步。 Lock.EnterScope() 方法进入独占范围。 从中返回的 `ref struct` 支持退出独占范围的 `Dispose()` 模式。

C# lock 语句可识别锁的目标是否为 `Lock` 对象。 如果是，则它使用更新的 API，而不是使用 System.Threading.Monitor的传统 API。 编译器还可以识别是否将 `Lock` 对象转换为另一种类型，并且将生成基于 `Monitor` 的代码。 可以在 新锁对象的功能规范中了解详细信息。

通过此功能，您可以通过更改对象 `lock`的类型来获得新库类型的优势。 无需更改其他代码。

新的转义序列

你可以使用 `\e` 作为 `ESCAPE` 字符 Unicode `U+001B` 的字符文本转义序列。 以前，你使用了 `\u001b` 或 `\x1b`。 不建议使用 `\x1b`，因为如果 `1b` 后面的下一个字符是有效的十六进制数字，则那些字符会成为转义序列的一部分。

方法组自然类型

此功能对涉及方法组的重载解析进行了小型优化。 *方法组*是一个方法，并且所有重载都具有相同的名称。 以前的行为是编译器为方法组构造完整的候选方法集。 如果需要自然类型，则自然类型是根据整套候选方法确定的。

新行为是在每个作用域内削减候选方法集，移除那些不适用的候选方法。 通常，被删除的方法是具有错误参数或不满足约束的泛型方法。 只有在没有找到候选方法的情况下，该过程才会继续到下一个外部作用域。 此过程更紧密地遵循重载解析的一般算法。 如果在给定范围内找到的所有候选方法不匹配，则方法组没有自然类型。

您可以在 提案规范中查看更改的详细信息。

隐式索引访问

现在在对象初始化表达式中允许使用隐式"从末尾"索引运算符`^`。 例如，现在可以在对象初始值设定项中初始化数组，如以下代码所示：

C#

```
public class TimerRemaining
{
    public int[] buffer { get; set; } = new int[10];
}

var countdown = new TimerRemaining()
{
    buffer =
    {
        [^1] = 0,
        [^2] = 1,
        [^3] = 2,
        [^4] = 3,
        [^5] = 4,
        [^6] = 5,
        [^7] = 6,
        [^8] = 7,
        [^9] = 8,
        [^10] = 9
    }
};
```

`TimerRemaining` 类包括初始化为长度为 10 的 `buffer` 数组。 前面的示例使用"from the end"索引运算符（`^`）将值赋给此数组，从而有效地创建从 9 到 0 的数组。

在 C# 13 之前的版本中，不能在对象初始值设定项中使用 `^` 运算符。 需要从前面为元素编制索引。

迭代器和 `async` 方法中的 `ref` 和 `unsafe`

此功能和以下两项功能使 `ref struct` 类型能够使用新构造。 除非你编写自己的 `ref struct` 类型，否则你不会使用这些类型。 你更可能会看到间接的好处，因为 System.Span<T> 和 System.ReadOnlySpan<T> 获得了更多的功能。

在 C# 13 之前，迭代器方法（使用 `yield return` 的方法）和 `async` 方法无法声明局部 `ref` 变量，也不能具有 `unsafe` 上下文。

在 C# 13 中，`async` 方法可以声明 `ref` 局部变量或 `ref struct` 类型的局部变量。 但是，无法跨 `await` 边界访问这些变量。 也不能跨越 `yield return` 边界访问它们。

这种宽松的限制使编译器能够在更多位置安全地使用 `ref` 局部变量和 `ref struct` 类型。 可以在这些方法中安全地使用 System.ReadOnlySpan<T> 等类型。 编译器会告诉你你是否违反了安全规则。

同样，C# 13 允许迭代器方法中的 `unsafe` 上下文。 但是，所有 `yield return` 和 `yield break` 语句都必须处于安全上下文中。

`allows ref struct`

在 C# 13 之前，无法将 `ref struct` 类型声明为泛型类型或方法的类型参数。 现在，泛型类型声明可以添加反约束，`allows ref struct`。 此反约束声明为该类型参数提供的类型参数可以是 `ref struct` 类型。 编译器对该类型参数的所有实例强制实施 ref 安全规则。

例如，可以声明泛型类型，如以下代码：

C#

```
public class C<T> where T : allows ref struct
{
    // Use T as a ref struct:
    public void M(scoped T p)
    {
        // The parameter p must follow ref safety rules
    }
}
```

这样，System.Span<T> 和 System.ReadOnlySpan<T> 等类型就可以与泛型算法一起使用（如果适用）。 可以在有关 where 的更新和有关 泛型约束的编程指南文章中了解详细信息。

`ref struct` 接口

在 C# 13 之前，不允许 `ref struct` 类型实现接口。 从 C# 13 开始，这些功能可以实现。 你可以声明 `ref struct` 类型实现某个接口。 但是，为了确保 ref 安全规则，无法将 `ref struct` 类型转换为接口类型。 这种转换是装箱转换，可能会违反 ref 安全性。 根据该规则，`ref struct` 类型无法声明用于显式实现接口方法的方法。 此外，`ref struct` 类型必须实现接口中声明的所有方法，包括具有默认实现的方法。

在 ref struct 类型的更新中了解详细信息。

更多部分成员

可以在 C# 13 中声明 `partial` 属性和 `partial` 索引器。 部分属性和索引器通常遵循与 `partial` 方法相同的规则：创建一个 *声明声明*，一个*实现声明*。 两个声明的签名必须匹配。 一个限制是，不能使用自动属性声明来*实现*部分属性。 未声明正文的属性被视为*声明声明*。

C#

```
public partial class C
{
    // Declaring declaration
    public partial string Name { get; set; }
}

public partial class C
{
    // implementation declaration:
    private string _name;
    public partial string Name
    {
        get => _name;
        set => _name = value;
    }
}
```

可以在有关部分成员的文章中了解详细信息。

重载解析优先级

在 C# 13 中，编译器识别 OverloadResolutionPriorityAttribute，以便优先选择一个重载而不是另一个。 库作者可以使用此属性来确保首选新的更好的重载而不是现有重载。 例如，可以添加性能更高的新重载。 你不希望中断使用库的现有代码，但希望用户在重新

编译时更新到新版本。 可以使用 重载解析优先级 来指示编译器应该优先选择哪个重载。 优先考虑优先级最高的超载。

此功能适用于库作者，以避免添加新重载时出现歧义。 库作者应谨慎处理此属性以避免混淆。

`field` 关键字

field 上下文关键字在 C# 13 中作为预览特性提供。 令牌 `field` 访问属性访问器中的编译器合成支持字段。 它使你可以编写访问器的主体，而无需在类型声明中显式声明备份字段。 可以为字段支持的属性的一个或两个访问器声明一个主体。

`field` 功能作为预览功能发布。 我们希望从你的使用经验中学习。 在还包含名为 `field` 的字段的类型中读取代码时，可能会出现破坏性更改或混淆。 可以使用 `@field` 或 `this.field` 来消除 `field` 关键字和标识符之间的歧义。

> **ⓘ 重要**
>
> `field` 关键字是 C# 13 中的预览功能。 必须使用 .NET 9 并将 `<LangVersion>` 元素设置为项目文件中 `preview`，才能使用 `field` 上下文关键字。
>
> 应小心在具有名为 `field` 的字段的类中使用 `field` 关键字功能。 新的 `field` 关键字在属性访问器的范围内隐藏一个名为 `field` 的字段。 可以更改 `field` 变量的名称，也可以使用 `@` 令牌将 `field` 标识符引用为 `@field`。 可以通过阅读 **field关键字**的功能规范来了解详细信息。

如果试用了此功能并有反馈，请将其添加到 `csharplang` 存储库中的 功能问题 。

另请参阅

- .NET 9 中的新增功能

C# 12 中的新增功能

项目 • 2024/03/28

C# 12 包括以下新增功能。 可以使用最新的 Visual Studio 2022 版本或 .NET 8 SDK 尝试这些功能。

- 主构造函数 - 在 Visual Studio 2022 版本 17.6 预览版 2 中引入。
- 集合表达式 - 在 Visual Studio 2022 版本 17.7 预览版 5 中引入。
- 内联数组 - 在 Visual Studio 2022 版本 17.7 预览版 3 中引入。
- Lambda 表达式中的可选参数 - 在 Visual Studio 2022 版本 17.5 预览版 2 中引入。
- ref readonly 参数 - 在 Visual Studio 2022 版本 17.8 预览版 2 中引入。
- 任何类型的别名 - 在 Visual Studio 2022 版本 17.6 预览版 3 中引入。
- 实验属性 - 已在 Visual Studio 2022 版本 17.7 预览版 3 中引入。
- 拦截器 - *预览功能* 在 Visual Studio 2022 版本 17.7 预览版 3 中引入。

.NET 8 支持 C# 12。 有关详细信息，请参阅 C# 语言版本控制。

可以通过 .NET 下载页下载最新的 .NET 8 SDK。 还可以下载 Visual Studio 2022，其中包括 .NET 8 SDK。

> **备注**
>
> 我们有兴趣了解你对这些功能的反馈。 如果发现这些新功能存在问题，请在 dotnet/roslyn 存储库中创建**新问题**。

主构造函数

现在可以在任何 `class` 和 `struct` 中创建主构造函数。 主构造函数不再局限于 `record` 类型。 主构造函数参数都在类的整个主体的范围内。 为了确保显式分配所有主构造函数参数，所有显式声明的构造函数都必须使用 `this()` 语法调用主构造函数。 将主构造函数添加到 `class` 可防止编译器声明隐式无参数构造函数。 在 `struct` 中，隐式无参数构造函数初始化所有字段，包括 0 位模式的主构造函数参数。

编译器仅在 `record` 类型（`record class` 或 `record struct` 类型）中为主构造函数参数生成公共属性。 对于主构造函数参数，非记录类和结构可能并不总是需要此行为。

有关主构造函数的详细信息，请参阅探索主构造函数的教程和实例构造函数的相关文章。

集合表达式

集合表达式引入了新的 terse 语法来创建常见的集合值。 可以使用展开运算符 `..` 将其他集合内联到这些值中。

可以创建多个类似集合的类型，而无需使用外部 BCL 支持。 这些类型包括：

- 数组类型，例如 `int[]`。
- System.Span<T> 和 System.ReadOnlySpan<T>。
- 支持集合初始值设定项的类型，例如 System.Collections.Generic.List<T>。

以下示例演示了集合表达式的使用：

C#

```
// Create an array:
int[] a = [1, 2, 3, 4, 5, 6, 7, 8];

// Create a list:
List<string> b = ["one", "two", "three"];

// Create a span
Span<char> c  = ['a', 'b', 'c', 'd', 'e', 'f', 'h', 'i'];

// Create a jagged 2D array:
int[][] twoD = [[1, 2, 3], [4, 5, 6], [7, 8, 9]];

// Create a jagged 2D array from variables:
int[] row0 = [1, 2, 3];
int[] row1 = [4, 5, 6];
int[] row2 = [7, 8, 9];
int[][] twoDFromVariables = [row0, row1, row2];
```

展开运算符（集合表达式中的 `..`）可将其参数替换为该集合中的元素。 参数必须是集合类型。 以下示例演示了展开运算符的工作原理：

C#

```
int[] row0 = [1, 2, 3];
int[] row1 = [4, 5, 6];
int[] row2 = [7, 8, 9];
int[] single = [.. row0, .. row1, .. row2];
foreach (var element in single)
{
    Console.Write($"{element}, ");
}
```

```
// output:
// 1, 2, 3, 4, 5, 6, 7, 8, 9,
```

展开运算符的操作数是可以枚举的表达式。 展开运算符可计算枚举表达式的每个元素。

可以在需要元素集合的任何位置使用集合表达式。 它们可以指定集合的初始值，也可以作为参数传递给采用集合类型的方法。 可以在有关集合表达式的语言参考文章或功能规范中详细了解集合表达式。

`ref readonly` 参数

C# 添加了 `in` 参数作为传递只读引用的方法。 `in` 参数既允许变量和值，并且无需对参数进行任何注释即可使用。

添加 `ref readonly` 参数可以更清楚地了解可能使用 `ref` 参数或 `in` 参数的 API：

- 即使参数未修改，在 `in` 引入之前创建的 API 也可能会使用 `ref`。 可以使用 `ref readonly` 更新这些 API。 对于调用方来说，这不会是一项中断性变更，就像参数 `ref` 更改为 `in` 那样。 示例为 System.Runtime.InteropServices.Marshal.QueryInterface。
- 采用 `in` 参数但逻辑上需要变量的 API。 值表达式不起作用。 示例为 System.ReadOnlySpan<T>.ReadOnlySpan<T>(T)。
- 使用 `ref` 的 API，因为它们需要变量，但不改变该变量。 示例为 System.Runtime.CompilerServices.Unsafe.IsNullRef。

若要了解有关 `ref readonly` 参数的详细信息，请参阅有关语言参考中的参数修饰符的文章，或参考只读参数功能规范。

默认 Lambda 参数

现在可以为 Lambda 表达式的参数定义默认值。 语法和规则与将参数的默认值添加到任何方法或本地函数相同。

可以在有关 Lambda 表达式的文章中详细了解 Lambda 表达式上的默认参数。

任何类型的别名

可以使用 `using` 别名指令创建任何类型的别名，而不仅仅是命名类型。 这意味着可以为元组类型、数组类型、指针类型或其他不安全类型创建语义别名。 有关详细信息，请参阅功能规范。

内联数组

运行时团队和其他库作者使用内联数组来提高应用的性能。 内联数组使开发人员能够创建固定大小的 `struct` 类型数组。 具有内联缓冲区的结构应提供类似于不安全的固定大小缓冲区的性能特征。 你可能不会声明自己的内联数组，但当它们从运行时 API 作为 System.Span<T> 或 System.ReadOnlySpan<T> 对象公开时，你将透明地使用这些数组。

*内联数组*的声明类似于以下 `struct`：

C#

```
[System.Runtime.CompilerServices.InlineArray(10)]
public struct Buffer
{
    private int _element0;
}
```

它们的用法与任何其他数组类似：

C#

```
var buffer = new Buffer();
for (int i = 0; i < 10; i++)
{
    buffer[i] = i;
}

foreach (var i in buffer)
{
    Console.WriteLine(i);
}
```

区别在于编译器可以利用有关内联数组的已知信息。 你可能会像使用任何其他数组一样使用内联数组。 有关如何声明内联数组的详细信息，请参阅有关 struct 类型的语言参考。

Experimental 属性

可以使用 System.Diagnostics.CodeAnalysis.ExperimentalAttribute 来标记类型、方法或程序集，以指示实验性特征。 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` 特性标记的程序集中包含的所有类型都是实验性的。 可以在有关编译器读取的常规属性的文章或功能规范中阅读详细信息。

拦截器

> **⚠ 警告**
>
> 拦截器是一项试验性功能，在 C# 12 的预览模式下提供。 在将来的版本中，该功能可能会发生中断性变更或被删除。 因此，不建议将其用于生产或已发布的应用程序。
>
> 若要使用拦截器，用户项目必须指定属性 `<InterceptorsPreviewNamespaces>`。 这是允许包含拦截器的命名空间的列表。
>
> 例如：
> `<InterceptorsPreviewNamespaces>$(InterceptorsPreviewNamespaces);Microsoft.AspNetCore.Http.Generated;MyLibrary.Generated</InterceptorsPreviewNamespaces>`

*拦截器*是一种方法，该方法可以在编译时以声明方式将对*可拦截*方法的调用替换为对其自身的调用。 通过让拦截器声明所拦截调用的源位置，可以进行这种替换。 拦截器可以向编译中（例如在源生成器中）添加新代码，从而提供更改现有代码语义的有限能力。

在源生成器中使用*拦截器*修改现有编译的代码，而非向其中添加代码。 源生成器将对可拦截方法的调用替换为对拦截器方法的调用。

如果你有兴趣尝试拦截器，可以阅读功能规范 来了解详细信息。 如果使用该功能，请确保随时了解此实验功能的功能规范中的任何更改。 最终确定功能后，我们将在本站点上添加更多指导。

另请参阅

- .NET 8 的新增功能

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# 11 中的新增功能

项目 • 2024/03/15

C# 11 中增加了以下功能：

- 原始字符串字面量
- 泛型数学支持
- 泛型属性
- UTF-8 字符串字面量
- 字符串内插表达式中的换行符
- 列表模式
- 文件本地类型
- 必需的成员
- 自动默认结构
- 常量 string 上的模式匹配 Span<char>
- 扩展的 nameof 范围
- 数值 IntPtr
- ref 字段和 scoped ref
- 改进了方法组向委托的转换
- 警告波 7

.NET 7 支持 C# 11。 有关详细信息，请参阅 C# 语言版本控制。

可以通过 .NET 下载页下载最新 .NET 7 SDK。 还可以下载 Visual Studio 2022，其中包括 .NET 7 SDK。

> **备注**
>
> 我们有兴趣了解你对这些功能的反馈。 如果发现这些新功能存在问题，请在 dotnet/roslyn 存储库中创建**新问题**。

泛型属性

可以声明基类为 System.Attribute 的泛型类。 此功能为需要 System.Type 参数的属性提供了更方便的语法。 以前需要创建一个属性，该属性将 `Type` 作为其构造函数参数：

C#

```
// Before C# 11:
public class TypeAttribute : Attribute
{
```

```
    public TypeAttribute(Type t) => ParamType = t;

    public Type ParamType { get; }
}
```

并且为了应用该属性，需要使用 typeof 运算符：

C#

```
[TypeAttribute(typeof(string))]
public string Method() => default;
```

使用此新功能，可以改为创建泛型属性：

C#

```
// C# 11 feature:
public class GenericAttribute<T> : Attribute { }
```

然后指定类型参数以使用该属性：

C#

```
[GenericAttribute<string>()]
public string Method() => default;
```

应用属性时，必须提供所有类型参数。 换句话说，泛型类型必须完全构造。 在上面的示例中，可以省略空括号 (`(` 和 `)`)，因为特性没有任何参数。

C#

```
public class GenericType<T>
{
    [GenericAttribute<T>()] // Not allowed! generic attributes must be fully
constructed types.
    public string Method() => default;
}
```

类型参数必须满足与 typeof 运算符相同的限制。 不允许使用需要元数据注释的类型。 例如，不允许将以下类型用作类型参数：

- `dynamic`
- `string?`（或任何可为 null 的引用类型）
- `(int X, int Y)`（或使用 C# 元组语法的任何其他元组类型）。

这些类型不会直接在元数据中表示出来。 它们包括描述该类型的注释。 在所有情况下，都可以改为使用基础类型：

- `object`（对于 `dynamic`）。
- `string`，而不是 `string?`。
- `ValueTuple<int, int>`，而不是 `(int X, int Y)`。

泛型数学支持

有几个语言功能支持泛型数学支持：

- `static virtual` 接口中的成员
- 已检查的用户定义的运算符
- 宽松移位运算符
- 无符号右移运算符

可以在接口中添加 `static abstract` 或 `static virtual` 成员，以定义包含可重载运算符、其他静态成员和静态属性的接口。 此功能的主要场景是在泛型类型中使用数学运算符。 例如，可以在实现 `operator +` 的类型中实现 `System.IAdditionOperators<TSelf, TOther, TResult>` 接口。 其他接口定义其他数学运算或明确定义的值。 可以在有关接口的文章中了解新语法。 包含 `static virtual` 方法的接口通常是泛型接口。 此外，大部分将声明一个约束，即类型参数实现声明接口。

可以在探索静态抽象接口成员教程或 .NET 6 中的预览功能 - 泛型数学博客文章中了解详细信息并亲自尝试该功能。

泛型数学对语言创建了其他要求。

- 无符号右移运算符：在 C# 11 之前，若要强制无符号右移，需要将任何带符号整数类型强制转换为无符号类型，执行移动，然后将结果强制转换回带符号类型。 从 C# 11 开始，可以使用 `>>>` *无符号移动运算符*。
- 宽松的移动运算符要求：C# 11 删除了第二个操作数必须是 `int` 或隐式转换为 `int` 的要求。 此更改允许在这些位置使用实现泛型数学接口的类型。
- 已检查和未检查的用户定义运算符：开发人员现在可以定义 `checked` 和 `unchecked` 算术运算符。 编译器根据当前上下文生成对正确变体的调用。 有关 `checked` 运算符的详细信息，可以阅读有关算术运算符的文章。

数值 `IntPtr` 和 `UIntPtr`

现在 `nint` 和 `nuint` 类型的别名分别为 System.IntPtr 和 System.UIntPtr。

字符串内插中的换行符

字符串内插的 `{` 和 `}` 字符内的文本现在可以跨多个行。 `{` 和 `}` 标记之间的文本分析为 C#。 允许任何合法 C#（包括换行符）。 使用此功能可以更轻松地读取使用较长 C# 表达式的字符串内插，例如模式匹配 `switch` 表达式或 LINQ 查询。

可以在语言参考的字符串内插文章中了解有关换行符功能的详细信息。

列表模式

列表模式扩展了模式匹配，以匹配列表或数组中的元素序列。 例如，当 `sequence` 为数组或三个整数（1、2 和 3）的列表时，`sequence is [1, 2, 3]` 为 `true`。 可以使用任何模式（包括常量、类型、属性和关系模式）来匹配元素。 弃元模式 (`_`) 匹配任何单个元素，新的范围模式 (`..`) 匹配零个或多个元素的任何序列。

可以在语言参考的模式匹配文章中了解有关列表模式的更多详细信息。

改进了方法组向委托的转换

方法组转换上的 C# 标准现在包含以下项：

> - 允许转换（但不是必需的）以使用已包含这些引用的现有委托实例。

以前版本的标准禁止了编译器重用为方法组转换而创建的委托对象。 C# 11 编译器缓存从方法组转换创建的委托对象，并重用该单个委托对象。 此功能最初在 Visual Studio 2022 版本 17.2 中作为预览功能提供，在 .NET 7 预览版 2 中首次提供。

原始字符串文本

原始字符串字面量是字符串字面量的一种新格式。 原始字符串字面量可以包含任意文本，包括空格、新行、嵌入引号和其他特殊字符，无需转义序列。 原始字符串字面量以至少三个双引号 (`"""`) 字符开头。 它以相同数量的双引号字符结尾。 通常，原始字符串字面量在单个行上使用三个双引号来开始字符串，在另一行上用三个双引号来结束字符串。 左引号之后、右引号之前的换行符不包括在最终内容中：

C#

```
string longMessage = """
    This is a long message.
    It has several lines.
        Some are indented
```

```
                more than others.
    Some should start at the first column.
    Some have "quoted text" in them.
    """;
```

右双引号左侧的任何空格都将从字符串字面量中删除。 原始字符串字面量可以与字符串内插结合使用，以在输出文本中包含大括号。 多个 `$` 字符表示有多少个连续的大括号开始和结束内插：

C#

```
var location = $$"""
   You are at {{{Longitude}}, {{Latitude}}}
   """;
```

前面的示例指定了两个大括号开始和结束内插。 第三个重复的左大括号和右大括号包括在输出字符串中。

可以在编程指南中关于字符串的文章中，以及关于字符串字面量和内插字符串的语言参考文章中详细了解原始字符串字面量。

自动默认结构

C# 11 编译器可以确保在执行构造函数的过程中，将 `struct` 类型的所有字段初始化为其默认值。 此更改意味着，任何未由构造函数初始化的字段或自动属性都由编译器自动初始化。 现在，构造函数未明确分配所有字段的结构可以进行编译，并且未显式初始化的任何字段都设置为其默认值。 有关此更改如何影响结构初始化的详细信息，请阅读有关结构的文章。

常量 `string` 上的模式匹配 `Span<char>` 或 `ReadOnlySpan<char>`

你已经能够在几个版本中使用模式匹配来测试 `string` 是否有某个特定的常量值。 现在，可以将同一模式匹配逻辑用于 `Span<char>` 或 `ReadOnlySpan<char>` 的变量。

扩展的 nameof 范围

在该方法的属性声明中的 `nameof` 表达式中使用时，类型参数名称和参数名称现在处于范围内。 此功能意味着可以使用 `nameof` 运算符在方法或参数声明的属性中指定方法参数的名称。 此功能最常用于为可为空分析添加属性。

UTF-8 字符串字面量

可以对字符串字面量指定 `u8` 后缀来指定 UTF-8 字符编码。 如果应用程序需要 UTF-8 字符串，则对于 HTTP 字符串常量或类似的文本协议来说，可以使用此功能来更轻松地创建 UTF-8 字符串。

有关 UTF-8 字符串字面量的详细信息，请参阅关于内置引用类型的文章的字符串字面量部分。

必需的成员

可以将 required 修饰符添加到属性和字段，以强制构造函数和调用方初始化这些值。 可以将 System.Diagnostics.CodeAnalysis.SetsRequiredMembersAttribute 添加到构造函数，以通知编译器构造函数将初始化所有必需的成员。

有关所需成员的详细信息，请参阅属性文章的仅限初始化部分。

`ref` 字段和 `ref scoped` 变量

可以在 ref struct 中声明 `ref` 字段。 这支持没有特殊特性或隐藏的内部类型的 System.Span<T> 等类型。

可向任意 `ref` 声明添加 scoped 修饰符。 这限制了可将引用转义到的范围。

文件本地类型

从 C# 11 开始，可以使用 `file` 访问修饰符创建其可见性范围限定为其声明时所在的源文件的类型。 此功能可帮助源生成器创建者避免命名冲突。 可以在语言参考中有关文件范围类型的文章中详细了解此功能。

另请参阅

- .NET 7 的新增功能

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

了解 C# 编译器中的所有重大更改

项目 • 2023/05/10

你可以在此处找到自 C# 10 版本以来的重大更改。

Roslyn 团队保留 C# 和 Visual Basic 编译器中的重大更改列表。 可以在 GitHub 存储库上的以下链接中找到有关这些更改的信息：

- C# 10.0/.NET 6 中，Roslyn 中的中断性变更
- .NET 5 之后 Roslyn 中的中断性变更
- VS2019 版本 16.8 中的中断性变更引入了 .NET 5 和 C# 9.0
- 与 VS2019 相比，VS2019 Update 1 及更高版本中的重大更改
- 自 VS2017 以来的重大更改 (C# 7)
- 自 Roslyn 2.* (VS2017) 以来 Roslyn 3.0 (VS2019) 中的重大更改
- 自 Roslyn 1.* (VS2015) 和本机 C# 编译器（VS2013 及更低版本）以来 Roslyn 2.0 (VS2017) 中的重大更改。
- 自本机 C# 编译器（VS2013 及更低版本）以来 Roslyn 1.0 (VS2015) 中的重大更改。
- C# 6 中的 Unicode 版本更改

C# 发展历史

项目 • 2025/01/14

本页介绍了 C# 语言每个主要版本的发展历史。 C# 团队将继续创新，以添加新功能。 可以在 GitHub 上的 dotnet/roslyn 存储库 上找到详细的语言功能状态，包括考虑在即将发布的版本中添加的功能。 若要查找何时将特定功能添加到语言，请在 GitHub 上的 dotnet/csharplang 存储库中查阅 C# 版本历史记录 文件。

> **ⓘ 重要**
>
> 为了提供一些功能，C# 语言依赖 C# 规范定义为*标准库*所用的类型和方法。 .NET 平台通过许多包交付这些类型和方法。 例如，异常处理。 为了确保引发的对象派生自 `throw`，将会检查每个 **Exception** 语句或表达式。 同样，还会检查每个 `catch`，以确保捕获的类型派生自 **Exception**。 每个版本都可能会新增要求。 若要在旧版环境中使用最新语言功能，可能需要安装特定库。 每个特定版本的页面中记录了这些依赖项。 若要了解此依赖项的背景信息，可以详细了解**语言与库的关系**。

C# 版本 13

发布日期：2024 年 11 月

C# 13 包含以下新功能：

- `params` 集合： `params` 修饰符不限于数组类型。 现在，可以将 `params` 与任何已识别的集合类型（包括 `Span<T>` 和接口类型）配合使用。
- 新的 `lock` 类型和语义：如果 `lock` 语句的目标为 System.Threading.Lock，编译器将生成代码以使用 Lock.EnterScope() 方法进入独占范围。 从中返回的 `ref struct` 支持退出独占范围的 `Dispose()` 模式。
- 新转义序列 - `\e`：可以将 `\e` 用作 `ESCAPE` 字符 (Unicode `U+001B`) 的字符字面值转义序列。
- 涉及方法组的重载解析的小型优化。
- 对象初始值设定项中的隐式索引器访问：现在允许在对象初始值设定项表达式中使用隐式的“from The end”索引运算符 `^`。
- 可以在迭代器和异步方法中使用 `ref` 局部变量和 `unsafe` 上下文。
- 可以使用 `ref struct` 类型来实现接口。
- 可以将 `ref struct` 类型作为泛型中类型参数的参数。
- 现在， `partial` 类型中允许部分属性和索引器。
- 重载解析优先级允许库作者将一个重载指定为比其他重载更好。

此外，用于访问自动实现属性中编译器生成的支持字段的 `field` 上下文关键字现已作为预览功能发布。

C# 版本 12

发布时间：2023 年 11 月

C# 12 中增加了以下功能：

- 主构造函数 - 可以创建任意 `class` 或 `struct` 类型的主构造函数。
- 集合表达式 - 一种用于指定集合表达式的新语法，包括 spread 元素 (`..e`)，可展开任何集合。
- 内联数组 - 使用内联数组，你可以创建固定大小的 `struct` 类型数组。
- Lambda 表达式中的可选参数 - 可以为 Lambda 表达式的参数定义默认值。
- ref readonly 参数 - `ref readonly` 参数可以让可能使用 `ref` 参数或 `in` 参数的 API 更清晰。
- 创建任何类型的别名 - 可以使用 `using` 别名指令创建任何类型的别名，而不仅仅是命名类型。
- 试验性属性 - 指示试验性功能。

拦截器 - 已作为预览功能发布。

总体而言，C# 12 提供的新功能可让你更高效地编写 C# 代码。 你已经知道的语法可以在更多地方使用。 其他语法可实现相关概念的一致性。

C# 版本 11

发布时间：2022 年 11 月

C# 11 中增加了以下功能：

- 原始字符串字面量
- 泛型数学支持
- 泛型属性
- UTF-8 字符串字面量
- 字符串内插表达式中的换行符
- 列表模式
- 文件本地类型
- 必需的成员
- 自动默认结构
- `Span<char>` 上的模式匹配 `string`
- 扩展的 nameof 范围

- 数值 IntPtr
- ref 字段和 scoped ref
- 改进了方法组向委托的转换
- 警告波 7

C# 11 引入了*泛型数学*以及支持该目标的几个功能。 可以为所有数字类型编写一次数值算法。 有更多功能可简化 `struct` 类型处理，例如所需成员和自动默认结构。 使用原始字符串文本、字符串内插中的换行符和 UTF-8 字符串文本可以更轻松地处理字符串。 文件本地类型等功能使源生成器更简单。 最后，列表模式添加了对模式匹配的更多支持。

C# 10 版

发布时间：2021 年 11 月

C# 10 向 C# 语言添加了以下功能和增强功能：

- 记录结构
- 结构类型的改进
- 内插字符串处理程序
- global using 指令
- 文件范围的命名空间声明
- 扩展属性模式
- Lambda 表达式可以具有自然类型，这使编译器可从 Lambda 表达式或方法组推断委托类型。
- 如果编译器无法推断返回类型，Lambda 表达式会声明该类型。
- 属性可应用于 Lambda 表达式。
- 在 C# 10 中，如果所有占位符本身均为常量字符串，可以使用字符串内插来初始化 `const` 字符串。
- 在 C# 10 中，在记录类型中重写 `ToString` 时可以添加 `sealed` 修饰符。
- 明确赋值和空状态分析的警告更准确。
- 允许在同一解构中同时进行赋值和声明。
- 可在方法上使用 AsyncMethodBuilder 属性
- CallerArgumentExpression 属性
- C# 10 支持 `#line` pragma 的新格式。

*预览*模式下提供了更多功能。 为了使用这些功能，需要在项目中将 <LangVersion> 设置为 Preview：

- 本文后面的泛型属性。
- 接口中的静态抽象成员。

C# 10 继续致力于删除不必要的模式、将数据与算法分离以及提高 .NET 运行时的性能等主题。

许多功能意味着可以通过键入更少的代码来表达相同的概念。 *记录结构*合并了许多*记录类*所使用的相同方法。 结构和匿名类型支持 *with 表达式*。 *全局 using 指令*和*文件范围的命名空间声明*意味着可以更清楚地表达依赖项和命名空间组织。 进行 Lambda 改进后，就可以在使用 Lambda 表达式时更容易地声明它们。 新的属性模式和析构改进可创建更简洁的代码。

新的内插字符串处理程序和 `AsyncMethodBuilder` 行为可提高性能。 在 .NET 运行时中应用了这些语言功能来实现 .NET 6 中的性能改进。

C# 10 还标志着每年 .NET 发布节奏的更多转变。 因为不是每项功能都可以在一年内完成，因此你可以尝试 C# 10 中的几项“预览”功能。 泛型属性和接口中的静态抽象成员均可使用，但这些预览功能在最终发布之前可能会发生更改。

C# 9 版

发布时间：2020 年 11 月

C# 9 随 .NET 5 一起发布。 它是面向 .NET 5 版本的任何程序集的默认语言版本。 它包含以下新功能和增强功能：

- 记录
- 仅限 Init 的资源库
- 顶级语句
- 模式匹配增强：关系模式和逻辑模式
- 性能和互操作性
  - 本机大小的整数
  - 函数指针
  - 禁止发出 localsinit 标志
  - 模块初始值设定项
  - 分部方法的新功能
- 调整和完成功能
  - 目标类型的 new 表达式
  - static 匿名函数
  - 目标类型的条件表达式
  - 协变返回类型
  - 扩展 GetEnumerator 支持 foreach 循环
  - Lambda 弃元参数
  - 本地函数的属性

C# 9 继续以前版本中的三大主题：删除不必要的模式、将数据与算法分离，以及在更多位置提供更多模式。

顶级语句意味着主程序将更易于读取。 减少了不必要的模式：命名空间、`Program` 类和 `static void Main()` 都是不必要的。

records 的引入为遵循值语义的引用类型提供了简洁的语法，以实现相等性。 使用这些类型来定义那些通常定义最小行为的数据容器。 仅限 init 的资源库提供了在记录中进行非破坏性修改的功能（`with` 表达式）。 C# 9 还添加了协变返回类型，以便派生记录可以重写虚拟方法，并返回从基方法的返回类型派生的类型。

模式匹配功能以多种方式进行了扩展。 数值类型现在支持*范围模式*。 可以使用 `and`、`or` 和 `not` 模式组合模式。 可以通过添加括号来阐明更复杂的模式：

C# 9 包括新的模式匹配改进：

- ***类型模式***匹配一个与特定类型匹配的对象
- 带圆括号的模式强制或强调模式组合的优先级
- ***联合 `and` 模式***要求两个模式都匹配
- ***析取 `or` 模式***要求任一模式匹配
- ***否定 `not` 模式要求模式不匹配***
- 关系模式要求输入小于、大于、小于等于或大于等于某个给定常数

这些模式丰富了模式的语法。 最常见的用途之一是用于 NULL 检查的新语法：

```C#
if (e is not null)
{
    // ...
}
```

后面模式中的任何一种都可在允许使用模式的任何上下文中使用：`is` 模式表达式、`switch` 表达式、嵌套模式以及 `switch` 语句的 `case` 标签的模式。

另一组功能支持 C# 中的高性能计算：

- `nint` 和 `nuint` 类型对目标 CPU 上的本机大小整数类型进行建模。
- 函数指针提供类似委托的功能，同时避免创建委托对象所需的分配。
- `localsinit` 指令可以省略以保存指令。

性能和互操作性

另一组改进支持*代码生成器*添加功能的场景：

- 模块初始化表达式是程序集加载时运行时调用的方法。
- 分部方法支持新的可访问修饰符和非 void 返回类型。 在这些情况下，必须提供一个实现。

调整和完成功能

C# 9 添加了许多其他小功能，提高了开发人员的工作效率，包括编写和读取代码：

- 目标类型 `new` 表达式
- `static` 匿名函数
- 目标类型的条件表达式
- 扩展 `GetEnumerator()` 支持 `foreach` 循环
- Lambda 表达式可以声明弃元参数
- 特性可应用于本地函数

C# 9 版本继续致力于让 C# 成为一种新式通用编程语言。 功能继续支持新式工作负载和应用程序类型。

C# 8.0 版

发布时间：2019 年 9 月

C# 8.0 版是专门面向 .NET C# Core 的第一个主要 C# 版本。 一些功能依赖于新的公共语言运行时 (CLR) 功能，而其他功能则依赖于仅在 .NET Core 中添加的库类型。 C# 8.0 向 C# 语言添加了以下功能和增强功能：

- Readonly 成员
- 默认接口方法
- 模式匹配增强功能：
  - switch 表达式
  - 属性模式
  - 元组模式
  - 位置模式
- Using 声明
- 静态本地函数
- 可处置的 ref 结构
- 可为空引用类型
- 异步流
- 索引和范围
- Null 合并赋值
- 非托管构造类型

- 嵌套表达式中的 Stackalloc
- 内插逐字字符串的增强功能

默认接口成员需要 CLR 中的增强功能。 这些功能已添加到 .NET Core 3.0 的 CLR 中。 范围和索引以及异步流需要 .NET Core 3.0 库中的新类型。 在编译器中实现时，可为 null 的引用类型在批注库时更有用，因为它可以提供有关参数和返回值的 null 状态的语义信息。 这些批注将添加到 .NET Core 库中。

C# 7.3 版

发布时间：2018 年 5 月

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，此版本中还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。
- 可以使用更多泛型约束。

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属性的后盾字段。
- 改进了通过 `in` 区分参数时进行的方法解析。
- 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。

C# 7.2 版

发布时间：2017 年 11 月

C# 7.2 版添加了几个小型语言功能：

- `stackalloc` 数组上的初始值设定项。
- 对支持模式的任何类型使用 `fixed` 语句。
- 无需固定即可访问固定的字段。
- 重新分配 `ref` 本地变量。
- 声明 `readonly struct` 类型，以指示结构不可变且应作为 `in` 参数传递到其成员方法。
- 在实参上添加 `in` 修饰符，以指定形参通过引用传递，但不通过调用方法修改。
- 对方法返回项使用 `ref readonly` 修饰符，以指示方法通过引用返回其值，但不允许写入该对象。
- 声明 `ref struct` 类型，以指示结构类型直接访问托管的内存，且必须始终分配有堆栈。
- 使用更多泛型约束。
- 非尾随命名参数：
  - 位置参数可以跟在命名参数后面。
- 数值文本中的前导下划线：
  - 数值文字现可在任何打印数字前放置前导下划线。
- private protected 访问修饰符：
  - `private protected` 访问修饰符允许访问同一程序集中的派生类。
- 条件 `ref` 表达式：
  - 现在可以引用条件表达式 (`?:`) 的结果。

C# 7.1 版

发布时间：2017 年 8 月

C# 已开始随 C# 7.1 发布*单点发行*。 此版本增加了语言版本选择配置元素、三个新的语言功能和新的编译器行为。

此版本中新增的语言功能包括：

- asyncMain 方法
  - 应用程序的入口点可以含有 `async` 修饰符。
- default 文本表达式
  - 在可以推断目标类型的情况下，可在默认值表达式中使用默认文本表达式。
- 推断元组元素名称
  - 在许多情况下，可通过元组初始化来推断元组元素的名称。
- 泛型类型参数的模式匹配
  - 可以对类型为泛型类型参数的变量使用模式匹配表达式。

最后，编译器有 -refout 和 -refonly 两个选项，可用于控制引用程序集生成。

C# 7.0 版

发布时间：2017 年 3 月

C# 7.0 版已与 Visual Studio 2017 一起发布。 此版本继承和发展了 C# 6.0。 以下介绍了部分新增功能：

- out 变量
- 元组和析构函数
- 模式匹配
- 本地函数
- 已扩展 expression bodied 成员
- ref 局部变量
- 引用返回

其他功能包括：

- 弃元
- 二进制文本和数字分隔符
- 引发表达式

所有这些功能都为开发者提供了新功能，帮助编写比以往任何时候都简洁的代码。 重点是缩减了使用 `out` 关键字的变量声明，并通过元组实现了多个返回值。 .NET Core 现在面向所有操作系统，着眼于云和可移植性。 语言设计者除了推出新功能外，也会在这些新功能方面付出时间和精力。

C# 6.0 版

发布时间：2015 年 7 月

版本 6.0 随 Visual Studio 2015 一起发布，发布了很多使得 C# 编程更有效率的小功能。 以下介绍了部分功能：

- 静态导入
- 异常筛选器
- 自动属性初始化表达式
- Expression bodied 成员
- Null 传播器
- 字符串内插
- nameof 运算符

其他新功能包括：

- 索引初始化表达式
- Catch/Finally 块中的 Await
- 仅限 getter 属性的默认值

如果整体看待这些功能，你会发现一个有趣的模式。 在此版本中，C# 开始消除语言样本，让代码更简洁且更具可读性。 所以对喜欢简洁代码的用户来说，此语言版本非常成功。

除了发布此版本，他们还做了另一件事，虽然这件事本身与传统的语言功能无关。 他们发布了 Roslyn 编译器即服务。 C# 编译器现在是用 C# 编写的，你可以使用编译器作为编程工作的一部分。

C# 5.0 版

发布时间：2012 年 8 月

C# 版本 5.0 随 Visual Studio 2012 一起发布，是该语言有针对性的一个版本。 对此版本中所做的几乎所有工作都归入另一个突破性语言概念：适用于异步编程的 `async` 和 `await` 模型。 下面是主要功能列表：

- 异步成员
- 调用方信息特性
- 代码工程：C# 5.0 中的调用方信息属性

调用方信息特性让你可以轻松检索上下文的信息，不需要采用大量样本反射代码。 这在诊断和日志记录任务中也很有用。

但是 `async` 和 `await` 才是此版本真正的主角。 C# 在 2012 年推出这些功能时，将异步引入语言作为最重要的组成部分，另现状大为改观。

C# 4.0 版

发布时间：2010 年 4 月

C# 4.0 版随 Visual Studio 2010 一起发布，引入了一些有趣的新功能：

- 动态绑定
- 命名参数/可选参数
- 泛型协变和逆变
- 嵌入的互操作类型

嵌入式互操作类型缓解了为应用程序创建 COM 互操作程序集的部署难题。 泛型协变和逆变提供了更强的功能来使用泛型，但风格比较偏学术，应该最受框架和库创建者的喜

爱。 命名参数和可选参数帮助消除了很多方法重载，让使用更方便。 但是这些功能都没有完全改变模式。

主要功能是引入 `dynamic` 关键字。 在 C# 4.0 版中引入 `dynamic` 关键字让用户可以替代编译时类型上的编译器。 通过使用 dynamic 关键字，可以创建和动态类型语言（例如 JavaScript）类似的构造。 可以创建 `dynamic x = "a string"` 再向它添加六个，然后让运行时理清下一步操作。

动态绑定存在出错的可能性，不过同时也为你提供了强大的语言功能。

C# 3.0 版

发布时间：2007 年 11 月

C# 3.0 版和 Visual Studio 2008 一起发布于 2007 年下半年，但完整的语言功能是在 .NET Framework 3.5 版中发布的。 此版本标示着 C# 发展过程中的重大更改。 C# 成为了真正强大的编程语言。 我们来看看此版本中的一些主要功能：

- 自动实现的属性
- 匿名类型
- 查询表达式
- Lambda 表达式
- 表达式树
- 扩展方法
- 隐式类型本地变量
- 分部方法
- 对象和集合初始值设定项

回顾过去，这些功能中大多数似乎都是不可或缺，难以分割的。 它们的组合都是经过巧妙布局。 此 C# 版本的杀手锏功能是查询表达式，也就是语言集成查询 (LINQ)。

LINQ 的构造可以建立在更细微的视图检查表达式树、Lambda 表达式以及匿名类型的基础上。 不过无论如何 C# 3.0 都提出了革命性的概念。 C# 3.0 开始为 C# 转变为面向对象/函数式混合语言奠定基础。

具体来说，你现在可以编写 SQL 样式的声明性查询对集合以及其他项目执行操作。 无需再编写 `for` 循环来计算整数列表的平均值，现在可改用简单的 `list.Average()` 方法。 组合使用查询表达式和扩展方法让各种数字变得智能多了。

C# 2.0 版

发布时间：2005 年 11 月

让我们看看 C# 2.0（2005 年发布）和 Visual Studio 2005 中的一些主要功能：

- 泛型
- 分部类型
- 匿名方法
- 可以为 null 的值类型
- 迭代器
- 协变和逆变

除现有功能以外的其他 C# 2.0 功能：

- getter/setter 单独可访问性
- 方法组转换（委托）
- 静态类
- 委托推断

虽然 C# 一开始是通用的面向对象 (OO) 语言，但 C# 2.0 版很快改变了这一点。 通过泛型，类型和方法可以操作任意类型，同时保持类型安全性。 例如，通过 List<T>，将获得 `List<string>` 或 `List<int>` 并且可以对这些字符串或整数执行类型安全操作，同时对其进行循环访问。 使用泛型优于创建派生自 `ListInt` 的 `ArrayList` 类型，也优于从每个操作的 `Object` 强制转换。

C# 2.0 版引入了迭代器。 简单来说，迭代器允许使用 `List` 循环来检查 `foreach`（或其他可枚举类型）中的所有项。 拥有迭代器是该语言最重要的一部分，显著提升了语言的可读性以及人们推出代码的能力。

C# 版本 1.2

发布时间：2003 年 4 月

随 Visual Studio .NET 2003 一起提供的 C# 版本 1.2。 它对语言做了一些小改进。 最值得注意的是，从此版本开始，当 `foreach` 实现 Dispose 时，IEnumerator 循环中生成的代码会在 IEnumerator 上调用 IDisposable。

C# 1.0 版

发布时间：2002 年 1 月

回想起来，和 Visual Studio .NET 2002 一起发布的 C# 版本 1.0 非常像 Java。 作为 ECMA 既定设计目标的一部分，其目标是成为一种“简单、现代、通用的面向对象的语言”。当时，看起来像 Java 意味着它实现了早期的设计目标。

不过如果现在回顾 C# 1.0，你会觉得有点晕。 它没有习以为常的内置异步功能和以泛型为中心的巧妙功能。 其实它完全不具备泛型。 那 LINQ 呢？ 尚不可用。 这些新增内容需要几年才能推出。

与现在的 C# 相比，C# 1.0 版少了很多功能。 你会发现自己的代码很冗长。 不过凡事总要有个开始。 在 Windows 平台上，C# 1.0 版是 Java 的一个可行的替代之选。

C# 1.0 的主要功能包括：

- 类
- 结构
- 接口
- 事件
- 属性
- 委托
- 运算符和表达式
- 语句
- 特性

文章最初发布在 NDepend 博客上，由 Erik Dietrich 和 Patrick Smacchia 提供。

语言功能与库类型之间的关系

项目 • 2023/11/30

C# 语言定义要求标准库拥有某些类型以及这些类型的特定可访问成员。 编译器针对多种不同语言功能生成使用这些必需类型和成员的代码。 因此，C# 版本仅支持相应的 .NET 版本和更新版本。 这可确保正确的运行时行为以及所有必需类型和成员的可用性。

此标准库功能的依赖项自其第一个版本起就是 C# 语言的一部分。 在该版本中，相关示例包括：

- Exception - 用于编译器生成的所有异常。
- String - `string` 的同义词。
- Int32 - `int` 的同义词。

第一个版本很简单：编译器和标准库一起提供，且各自都只有一个版本。

后续版本的 C# 偶尔会向依赖项添加新类型或成员。 相关示例包括：INotifyCompletion、CallerFilePathAttribute 和 CallerMemberNameAttribute。 C# 7.0 在 ValueTuple 上添加了一个依赖项来实现元组语言功能。 C# 8 对于范围和索引需要 System.Index 和 System.Range 以及其他功能。 每个新版本可能会添加其他要求。

语言设计团队致力于最小化符合标准的标准库所需的类型和成员的外围应用。 该目标针对新库功能无缝集成到语言的简洁设计进行了平衡。 未来版本的 C# 中还会包括需要标准库中的新类型和成员的新功能。 C# 编译器工具现在从支持的平台上 .NET 库的发布周期分离。

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

.NET is an open source project.
Select a link to provide feedback:

提出文档问题

提供产品反馈

面向 C# 开发人员的版本和更新注意事项

项目 • 2023/07/03

随着新功能被添加到 C# 语言中，兼容性成为一个重要的目标。 在几乎所有情况下，都可以使用新的编译器版本重新编译现有代码，不会出现任何问题。 .NET 运行时团队还有一个目标，即确保更新库的兼容性。 在几乎所有情况下，从具有更新库的更新运行时启动应用时，其行为与以前版本的行为完全相同。

用于编译应用的语言版本通常与项目中引用的运行时目标框架名字对象 (TFM) 匹配。 若要详细了解如何更改默认语言版本，请参阅标题为配置语言版本的文章。 此默认行为可确保最大的兼容性。

引入中断性变更时，它们被归为以下几类：

- 二进制中断性变更：使用新运行时启动时，二进制中断性变更会导致应用程序或库中出现不同的行为，包括可能发生故障。 必须重新编译应用才能合并这些更改。 现有二进制文件无法正常工作。
- 源中断性变更：源中断性变更会更改源代码的含义。 在使用最新语言版本编译应用程序之前，需要编辑源代码。 现有二进制文件将在较新的主机和运行时中正常运行。 请注意，对于语言语法，源中断性变更也是行为变更，如运行时中断性变更中所定义。

如果二进制中断性变更影响应用，就必须重新编译应用，但无需编辑任何源代码。 如果源中断性变更影响应用，现有二进制文件仍可在具有更新的运行时和库的环境中正常运行。 但是，必须进行源更改才能使用新的语言版本和运行时重新编译。 如果更改既是源中断性变更又是二进制中断性变更，则必须使用最新版本重新编译应用程序并更新源。

由于 C# 语言团队和运行时团队的目标是避免中断性变更，因此更新应用程序通常是更新 TFM 和重新生成应用的问题。 但对于公开分发的库，应仔细评估策略是否支持 TFM 和支持的语言版本。 你可能要使用最新版本中提供的功能创建一个新库，并且需要确保使用以前版本的编译器生成的应用可以使用它。 或者，你可能要升级现有库，但许多用户可能还没有升级版本。

在库中引入中断性变更

在库的公共 API 中采用新语言功能时，应评估采用该功能是否为库用户引入了二进制或源中断性变更。 在 `public` 或 `protected` 接口中未出现的针对内部实现的任何更改都是兼容的。

> **备注**

如果使用 System.Runtime.CompilerServices.InternalsVisibleToAttribute 使类型能够看到内部成员，则内部成员可能会引入中断性变更。

*二进制中断性变更*要求用户重新编译其代码才能使用新版本。 例如，请考虑以下公共方法：

```C#
public double CalculateSquare(double value) => value * value;
```

如果将 `in` 修饰符添加到方法，则这是二进制中断性变更：

```C#
public double CalculateSquare(in double value) => value * value;
```

用户必须重新编译使用 `CalculateSquare` 方法的任何应用程序才能使新库正常工作。

源中断性变更要求用户在重新编译之前更改其代码。 例如，请考虑以下类型：

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

    public Person(string firstName, string lastName) => (FirstName,
LastName) = (firstName, lastName);

    // other details omitted
}
```

在较新版本中，你想要利用为 `record` 类型生成的合成成员。 进行下列更改：

```C#
public record class Person(string FirstName, string LastName);
```

前面的更改要求对派生自 `Person` 的任何类型进行更改。 所有这些声明都必须将 `record` 修饰符添加到其声明中。

中断性变更的影响

向库添加二进制中断性变更时，将强制所有使用库的项目重新编译。 但这些项目中的源代码都不需要更改。 因此，中断性变更对每个项目的影响都相当小。

对库进行源中断性变更时，需要所有项目进行源更改才能使用新库。 如果必要的更改需要新的语言功能，则强制这些项目升级到你现在使用的语言版本和 TFM。 你需要为用户执行更多工作，并且他们还可能被迫升级。

所做的任何中断性变更的影响取决于依赖于库的项目数。 如果一些应用程序在内部使用库，你可以对所有受影响项目中的任何中断性变更做出响应。 但如果库是公开下载的，则应评估潜在影响并考虑替代方法：

- 可添加与现有 API 并行的新 API。
- 可考虑对不同的 TFM 使用并行生成。
- 可以考虑使用多目标。

教程：探索主构造函数

项目 • 2023/06/29

C# 12 引入了主构造函数，这是一种简明的语法，用于声明一些构造函数，它们的参数在类型主体中的任何位置可用。

本教程介绍如何执行下列操作：

- ✓ 何时对类型声明主构造函数
- ✓ 如何从其他构造函数调用主构造函数
- ✓ 如何在类型的成员中使用主构造函数参数
- ✓ 将主构造函数参数存储在哪里

先决条件

需要将计算机设置为运行 .NET 8 或更高版本，包括 C# 12 或更高版本编译器。 自 Visual Studio 2022 版本 17.7 或 .NET 8 SDK 起，开始随附 C# 12 编译器。

主构造函数

可以将参数添加到 `struct` 或 `class` 声明中，用于创建主构造函数。 主构造函数参数在整个类定义范围内。 请务必将主构造函数参数视为参数，即使它们在整个类定义范围内也是如此。 有几个规则阐明了它们是参数：

1. 如果不需要主构造函数参数，可能不会存储它们。
2. 主构造函数参数不是类的成员。 例如，名为 `param` 的主构造函数参数不能作为 `this.param` 访问。
3. 可以对主构造函数参数进行分配。
4. 主构造函数参数不会成为属性，record 类型除外。

这些规则与任何方法的参数相同，包括其他构造函数声明。

主构造函数参数的最常见用途包括：

1. 作为 `base()` 构造函数调用的参数。
2. 初始化成员字段或属性。
3. 引用实例成员中的构造函数参数。

类的所有其他构造函数都必须通过 `this()` 构造函数调用直接或间接调用主构造函数。 该规则可确保在类型的主体中的任意位置分配主构造函数参数。

初始化属性

以下代码初始化从主构造函数参数计算的两个只读属性：

C#

```
public readonly struct Distance(double dx, double dy)
{
    public readonly double Magnitude = Math.Sqrt(dx * dx + dy * dy);
    public readonly double Direction = Math.Atan2(dy, dx);
}
```

前面的代码演示了用于初始化计算的只读属性的主构造函数。 `Direction` 和 `Magnitude` 的字段初始值设定项使用主构造函数参数。 主构造函数参数不会在结构中的其他任何位置使用。 前面的结构就像编写了以下代码一样：

C#

```
public readonly struct Distance
{
    public readonly double Magnitude { get; }

    public readonly double Direction { get; }

    public Distance(double dx, double dy)
    {
        Magnitude = Math.Sqrt(dx * dx + dy * dy);
        Direction = Math.Atan2(dy, dx);
    }
}
```

当需要参数来初始化字段或属性时，利用新功能可以更轻松地使用字段初始值设定项。

创建可变状态

前面的示例使用主构造函数参数来初始化只读属性。 如果属性不是只读的，你还可以使用主构造函数。 考虑下列代码：

C#

```
public struct Distance(double dx, double dy)
{
    public readonly double Magnitude => Math.Sqrt(dx * dx + dy * dy);
    public readonly double Direction => Math.Atan2(dy, dx);

    public void Translate(double deltaX, double deltaY)
    {
```

```
        dx += deltaX;
        dy += deltaY;
    }

    public Distance() : this(0,0) { }
}
```

在前面的示例中，`Translate` 方法了更改 `dx` 和 `dy` 组件。 这就需要在访问时计算 `Magnitude` 和 `Direction` 属性。 `=>` 运算符指定一个以表达式为主体的 `get` 访问器，而 `=` 运算符指定一个初始值设定项。 此版本将无参数构造函数添加到结构。 无参数构造函数必须调用主构造函数，以便初始化所有主构造函数参数。

在前面的示例中，主构造函数属性是在方法中访问的。 因此，编译器会创建隐藏的字段来表示每个参数。 下面的代码大致展示了编译器生成的内容。 实际字段名称是有效的 MSIL 标识符，但不是有效的 C# 标识符。

C#

```
public struct Distance
{
    private double __unspeakable_dx;
    private double __unspeakable_dy;

    public readonly double Magnitude => Math.Sqrt(__unspeakable_dx *
__unspeakable_dx + __unspeakable_dy * __unspeakable_dy);
    public readonly double Direction => Math.Atan2(__unspeakable_dy,
__unspeakable_dx);

    public void Translate(double deltaX, double deltaY)
    {
        __unspeakable_dx += deltaX;
        __unspeakable_dy += deltaY;
    }

    public Distance(double dx, double dy)
    {
        __unspeakable_dx = dx;
        __unspeakable_dy = dy;
    }
    public Distance() : this(0, 0) { }
}
```

请务必了解，第一个示例不需要编译器创建字段来存储主构造函数参数的值。 第二个示例在方法内使用了主构造函数参数，因此需要编译器为它们创建存储。 只有在类型成员的主体中访问该参数时，编译器才会为任何主构造函数创建存储。 否则，主构造函数参数不会存储在对象中。

依赖关系注入

主构造函数的另一个常见用途是指定依赖项注入的参数。 下面的代码创建了一个简单的控制器，使用时需要有一个服务接口：

C#

```
public interface IService
{
    Distance GetDistance();
}

public class ExampleController(IService service) : ControllerBase
{
    [HttpGet]
    public ActionResult<Distance> Get()
    {
        return service.GetDistance();
    }
}
```

主构造函数清楚地指明了类中所需的参数。 使用主构造函数参数就像使用类中的任何其他变量一样。

初始化基类

可以从派生类的主构造函数调用基类的主构造函数。 这是编写必须调用基类中主构造函数的派生类的最简单方法。 例如，假设有一个类的层次结构，将不同的帐户类型表示为一个银行。 基类类似于以下代码：

C#

```
public class BankAccount(string accountID, string owner)
{
    public string AccountID { get; } = accountID;
    public string Owner { get; } = owner;

    public override string ToString() => $"Account ID: {AccountID}, Owner:
{Owner}";
}
```

所有银行帐户（无论是什么类型）都具有帐号和所有者的属性。 在完成的应用程序中，其他常见功能将添加到基类中。

许多类型都需要对构造函数参数进行更具体的验证。 例如，`BankAccount` 对 `owner` 和 `accountID` 参数有特定的要求：`owner` 不得为 `null` 或空格，并且 `accountID` 必须是包含

10 位数字的字符串。 在分配相应的属性时，可以添加此验证：

C#

```
public class BankAccount(string accountID, string owner)
{
    public string AccountID { get; } = ValidAccountNumber(accountID)
        ? accountID
        : throw new ArgumentException("Invalid account number",
nameof(accountID));

    public string Owner { get; } = string.IsNullOrWhiteSpace(owner)
        ? throw new ArgumentException("Owner name cannot be empty",
nameof(owner))
        : owner;

    public override string ToString() => $"Account ID: {AccountID}, Owner:
{Owner}";

    public static bool ValidAccountNumber(string accountID) =>
    accountID?.Length == 10 && accountID.All(c => char.IsDigit(c));
}
```

前面的示例展示了如何在将构造函数参数分配给属性之前对其进行验证。 可以使用内置的方法（如 String.IsNullOrWhiteSpace(String)），也可以使用你自己的验证方法（如 `ValidAccountNumber`）。 在前面的示例中，当构造函数调用初始值设定项时，将引发任何异常。 如果未使用构造函数参数来分配字段，在首次访问构造函数参数时将引发任何异常。

一个派生类将呈现一个支票帐户：

C#

```
public class CheckAccount(string accountID, string owner, decimal
overdraftLimit = 0) : BankAccount(accountID, owner)
{
    public decimal CurrentBalance { get; private set; } = 0;

    public void Deposit(decimal amount)
    {
        if (amount < 0)
        {
            throw new ArgumentOutOfRangeException(nameof(amount), "Deposit
amount must be positive");
        }
        CurrentBalance += amount;
    }

    public void Withdrawal(decimal amount)
    {
        if (amount < 0)
```

```
        {
            throw new ArgumentOutOfRangeException(nameof(amount),
"Withdrawal amount must be positive");
        }
        if (CurrentBalance - amount < -overdraftLimit)
        {
            throw new InvalidOperationException("Insufficient funds for
withdrawal");
        }
        CurrentBalance -= amount;
    }

    public override string ToString() => $"Account ID: {AccountID}, Owner:
{Owner}, Balance: {CurrentBalance}";
}
```

派生的 `CheckingAccount` 类具有一个主构造函数，采用基类中所需的所有参数，以及另一个具有默认值的参数。 主构造函数使用 `: BankAccount(accountID, owner)` 语法调用基构造函数。 此表达式同时指定基类的类型和主构造函数的参数。

派生类不需要使用主构造函数。 可以在派生类中创建一个构造函数，用于调用基类的主构造函数，如以下示例所示：

C#

```
public class LineOfCreditAccount : BankAccount
{
    private readonly decimal _creditLimit;
    public LineOfCreditAccount(string accountID, string owner, decimal
creditLimit) : base(accountID, owner)
    {
        _creditLimit = creditLimit;
    }
    public decimal CurrentBalance { get; private set; } = 0;

    public void Deposit(decimal amount)
    {
        if (amount < 0)
        {
            throw new ArgumentOutOfRangeException(nameof(amount), "Deposit
amount must be positive");
        }
        CurrentBalance += amount;
    }

    public void Withdrawal(decimal amount)
    {
        if (amount < 0)
        {
            throw new ArgumentOutOfRangeException(nameof(amount),
"Withdrawal amount must be positive");
        }
```

```
        if (CurrentBalance - amount < -_creditLimit)
        {
            throw new InvalidOperationException("Insufficient funds for
withdrawal");
        }
        CurrentBalance -= amount;
    }

    public override string ToString() => $"{base.ToString()}, Balance:
{CurrentBalance}";
}
```

类层次结构和主构造函数有一个潜在的问题：在派生类和基类中使用主构造函数参数时，可以创建主构造函数参数的多个副本。 下面的代码示例为每个 `owner` 和 `accountID` 字段创建两个副本：

C#

```
public class SavingsAccount(string accountID, string owner, decimal
interestRate) : BankAccount(accountID, owner)
{
    public SavingsAccount() : this("default", "default", 0.01m) { }
    public decimal CurrentBalance { get; private set; } = 0;

    public void Deposit(decimal amount)
    {
        if (amount < 0)
        {
            throw new ArgumentOutOfRangeException(nameof(amount), "Deposit
amount must be positive");
        }
        CurrentBalance += amount;
    }

    public void Withdrawal(decimal amount)
    {
        if (amount < 0)
        {
            throw new ArgumentOutOfRangeException(nameof(amount),
"Withdrawal amount must be positive");
        }
        if (CurrentBalance - amount < 0)
        {
            throw new InvalidOperationException("Insufficient funds for
withdrawal");
        }
        CurrentBalance -= amount;
    }

    public void ApplyInterest()
    {
        CurrentBalance *= 1 + interestRate;
```

```
    }

    public override string ToString() => $"Account ID: {accountID}, Owner:
{owner}, Balance: {CurrentBalance}";
}
```

突出显示的行显示 `ToString` 方法使用主构造函数参数（`owner` 和 `accountID`），而不是基类属性（`Owner` 和 `AccountID`）。 结果是派生类 `SavingsAccount` 为这些副本创建存储。 派生类中的副本与基类中的属性不同。 如果可以修改基类属性，派生类的实例将看不到该修改。 编译器对派生类中使用的、传递给基类构造函数的主构造函数参数发出警告。 在此实例中，解决方法是使用基类接口中的属性。

总结

可以使用最适合设计的主构造函数。 对于类和结构，主构造函数参数是必须调用的构造函数的参数。 可以使用它们来初始化属性。 可以初始化字段。 这些属性或字段可以是不可变的，也可以是可变的。 可以在方法中使用它们。 它们都是参数，可以用最适合设计的方式来使用它们。 有关主构造函数的详细信息，请参阅有关实例构造函数的 C# 编程指南文章和建议的主构造函数规范。

教程：探索 C# 11 功能 - 接口中的静态虚拟成员

项目 • 2024/07/04

C# 11 和 .NET 7 包括接口中的静态虚拟成员。 使用此功能，可定义包含 重载运算符或其他静态成员的接口。 使用静态成员定义接口后，可使用这些接口作为约束来创建使用运算符或其他静态方法的泛型类型。 即使不使用重载运算符创建接口，你也可能会受益于此功能和语言更新启用的泛型数学类。

本教程介绍以下操作：

- ✓ 定义具有静态成员的接口。
- ✓ 使用接口定义实现接口的类，这些接口定义了运算符。
- ✓ 创建依赖于静态接口方法的泛型算法。

先决条件

需要将计算机设置为运行 .NET 7，该软件支持 C# 11。 自 Visual Studio 2022 版本 17.3 或 .NET 7 SDK 起，开始随附 C# 11 编译器。

静态抽象接口方法

我们从一个示例开始。 以下方法返回两个 `double` 数字的中点：

C#

```
public static double MidPoint(double left, double right) =>
    (left + right) / (2.0);
```

该逻辑适用于任何数值类型：`int`、`short`、`long`、`float`、`decimal` 或表示数字的任何类型。 你需要有一种方法来使用 `+` 运算符和 `/` 运算符，并为 `2` 指定一个值。 可使用 System.Numerics.INumber<TSelf> 接口，将上述方法编写为以下泛型方法：

C#

```
public static T MidPoint<T>(T left, T right)
    where T : INumber<T> => (left + right) / T.CreateChecked(2);  // note:
the addition of left and right may overflow here; it's just for
demonstration purposes
```

实现 INumber<TSelf> 接口的任何类型都必须包含 `operator +` 和 `operator /` 的定义。 分母由 `T.CreateChecked(2)` 定义，从而为任何数值类型创建 `2` 值，这将强制要求分母与两个参数的类型相同。 INumberBase<TSelf>.CreateChecked<TOther>(TOther) 根据指定值创建类型的实例，如果值超出可表示范围，则引发 OverflowException。 （如果 `left` 和 `right` 都是足够大的值，则此实现可能会发生溢出。有一些替代算法可避免这个潜在问题。）

使用熟悉的语法定义接口中的静态抽象成员：将 `static` 和 `abstract` 修饰符添加到不提供实现的任何静态成员。 下面的示例定义了一个 `IGetNext<T>` 接口，该接口可应用于替代 `operator ++` 的任何类型：

C#

```
public interface IGetNext<T> where T : IGetNext<T>
{
    static abstract T operator ++(T other);
}
```

要求类型参数 `T` 实现 `IGetNext<T>` 的约束可确保运算符的签名中具有包含类型或其类型参数。 许多运算符都强制要求其参数必须与类型匹配，或者是按照约束要实现包含类型的类型参数。 如果没有此约束，则不能在 `IGetNext<T>` 接口中定义 `++` 运算符。

你可以创建一个结构，该结构创建由“A”字符组成的字符串，其中每个增量使用以下代码向字符串另外添加一个字符：

C#

```
public struct RepeatSequence : IGetNext<RepeatSequence>
{
    private const char Ch = 'A';
    public string Text = new string(Ch, 1);

    public RepeatSequence() {}

    public static RepeatSequence operator ++(RepeatSequence other)
        => other with { Text = other.Text + Ch };

    public override string ToString() => Text;
}
```

更常见的情况是，你可生成任何算法；在该算法中，你可能需要定义 `++` 来表示“生成此类型的下一个值”。 使用此接口将生成清晰的代码和结果：

C#

```
var str = new RepeatSequence();

for (int i = 0; i < 10; i++)
    Console.WriteLine(str++);
```

上述示例得到以下输出：

PowerShell

```
A
AA
AAA
AAAA
AAAAA
AAAAAA
AAAAAAA
AAAAAAAA
AAAAAAAAA
AAAAAAAAAA
```

这个很小的示例演示了此功能的动机。 可以对运算符、常量值和其他静态运算使用自然语法。 在创建依赖静态成员的多个类型（包括重载运算符）时，可探索这些方法。 定义与类型功能匹配的接口，然后声明这些类型对新接口的支持。

泛型数学

要促使在接口中允许使用静态方法（包括运算符），需要支持泛型数据算法。 .NET 7 基类库中具有许多算术运算符的接口定义，还具有在一个 `INumber<T>` 接口中组合许多算术运算符的派生接口。 让我们应用这些类型来生成可对 `T` 使用任何数值类型的 `Point<T>` 记录。 可使用 `+` 运算符按一定的 `XOffset` 和 `YOffset` 来移动点。

首先使用 `dotnet new` 或 Visual Studio 创建一个新的控制台应用程序。

`Translation<T>` 和 `Point<T>` 的公共接口应类似于以下代码：

C#

```
// Note: Not complete. This won't compile yet.
public record Translation<T>(T XOffset, T YOffset);

public record Point<T>(T X, T Y)
{
    public static Point<T> operator +(Point<T> left, Translation<T> right);
}
```

同时对 `Translation<T>` 和 `Point<T>` 类型使用 `record` 类型：这两种类型都存储两个值，它们表示数据存储而不是复杂的行为。 `operator +` 的实现将类似于以下代码：

C#

```
public static Point<T> operator +(Point<T> left, Translation<T> right) =>
    left with { X = left.X + right.XOffset, Y = left.Y + right.YOffset };
```

为了使上述代码进行编译，你需要声明 `T` 支持 `IAdditionOperators<TSelf, TOther, TResult>` 接口。 该接口包含 `operator +` 静态方法。 它声明 3 个类型参数：一个用于左操作数，一个用于右操作数，一个用于结果。 某些类型对不同的操作数和结果类型实现 `+`。 添加要求类型参数 `T` 实现 `IAdditionOperators<T, T, T>` 的声明：

C#

```
public record Point<T>(T X, T Y) where T : IAdditionOperators<T, T, T>
```

添加该约束后，`Point<T>` 类可对其加法运算符使用 `+`。 对 `Translation<T>` 声明添加同一约束：

C#

```
public record Translation<T>(T XOffset, T YOffset) where T :
IAdditionOperators<T, T, T>;
```

`IAdditionOperators<T, T, T>` 约束可防止开发人员使用你的类通过类型创建 `Translation`，其中该类型不满足点加法的约束。 已将必需的约束添加到 `Translation<T>` 和 `Point<T>` 的类型参数，所以此代码可正常工作。 可通过在 Program.cs 文件中的 `Translation` 和 `Point` 的声明上面添加如下所示的代码进行测试：

C#

```
var pt = new Point<int>(3, 4);

var translate = new Translation<int>(5, 10);

var final = pt + translate;

Console.WriteLine(pt);
Console.WriteLine(translate);
Console.WriteLine(final);
```

可通过声明这些类型实现适当的算术接口，使此代码更易于重用。 首先要做的更改是声明 `Point<T, T>` 实现 `IAdditionOperators<Point<T>, Translation<T>, Point<T>>` 接口。

`Point` 类型对操作数和结果使用不同的类型。 `Point` 类型已使用该签名实现了 `operator +`，因此只需要将接口添加到声明中：

```
C#
```

```
public record Point<T>(T X, T Y) : IAdditionOperators<Point<T>,
Translation<T>, Point<T>>
    where T : IAdditionOperators<T, T, T>
```

最后，在执行加法时，具有定义该类型的加法恒等元值的属性非常有用。 此功能有一个新的接口：IAdditiveIdentity<TSelf,TResult>。 `{0, 0}` 的转换是加法恒等元：所得的点与左操作数相同。 `IAdditiveIdentity<TSelf, TResult>` 接口定义一个只读属性 `AdditiveIdentity`，它返回恒等元值。 `Translation<T>` 需要进行一些更改来实现此接口：

```
C#
```

```
using System.Numerics;

public record Translation<T>(T XOffset, T YOffset) :
IAdditiveIdentity<Translation<T>, Translation<T>>
    where T : IAdditionOperators<T, T, T>, IAdditiveIdentity<T, T>
{
    public static Translation<T> AdditiveIdentity =>
        new Translation<T>(XOffset: T.AdditiveIdentity, YOffset:
T.AdditiveIdentity);
}
```

这里有一些更改，接下来让我们逐一介绍。 首先，声明 `Translation` 类型实现 `IAdditiveIdentity` 接口：

```
C#
```

```
public record Translation<T>(T XOffset, T YOffset) :
IAdditiveIdentity<Translation<T>, Translation<T>>
```

接下来，可尝试实现接口成员，如下面的代码所示：

```
C#
```

```
public static Translation<T> AdditiveIdentity =>
    new Translation<T>(XOffset: 0, YOffset: 0);
```

前面的代码不会进行编译，因为 `0` 取决于类型。 答案：对 `0` 使用 `IAdditiveIdentity<T>.AdditiveIdentity`。 此更改意味着约束现在必须要求 `T` 实现

`IAdditiveIdentity<T>`。 这会导致以下实现：

```
C#

public static Translation<T> AdditiveIdentity =>
    new Translation<T>(XOffset: T.AdditiveIdentity, YOffset:
T.AdditiveIdentity);
```

现在，你已将该约束添加到 `Translation<T>`，接下来需要将它添加到 `Point<T>`：

```
C#

using System.Numerics;

public record Point<T>(T X, T Y) : IAdditionOperators<Point<T>,
Translation<T>, Point<T>>
    where T : IAdditionOperators<T, T, T>, IAdditiveIdentity<T, T>
{
    public static Point<T> operator +(Point<T> left, Translation<T> right)
=>
        left with { X = left.X + right.XOffset, Y = left.Y + right.YOffset
};
}
```

本示例向你介绍了如何编写泛型数学接口。 你已了解如何：

- ✓ 编写一个方法，该方法依赖于 `INumber<T>` 接口，以便可将该方法与任何数值类型一起使用。
- ✓ 生成一个类型，该类型依赖于添加接口来实现仅支持一种数学运算的类型。 该类型声明它对这些接口的支持，因此可通过其他方式编写它。 算法使用最自然的数学运算符语法进行编写。

试用这些功能并提出反馈。 可使用 Visual Studio 中的“发送反馈”菜单项，或者 GitHub 上的 roslyn 存储库中创建新问题。 生成适用于任何数值类型的泛型算法。 使用这些接口生成算法，其中类型参数只能实现一部分类似于数字的功能。 即使不生成使用这些功能的新接口，也可尝试在算法中使用它们。

请参阅

- 泛型数学

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

创建记录类型

项目 • 2025/01/03

*记录*是使用*基于值的相等性*的类型。 可以将记录定义为引用类型或值类型。 如果记录类型定义相同，并且每个字段的值在两个记录中都相等，那么这两个记录类型的变量是相等的。 如果引用的对象是相同的类类型，并且变量引用同一对象，则类类型的两个变量相等。 基于值的相等性意味着可能需要的记录类型中的其他功能。 声明 `record` 而不是 `class`时，编译器会生成其中许多成员。 编译器为 `record struct` 类型生成相同的方法。

您将在本教程中学习如何：

- ✔ 确定是否将 `record` 修饰符添加到 `class` 类型。
- ✔ 声明记录类型和位置记录类型。
- ✔ 在记录中将你的方法替换为编译器生成的方法。

先决条件

需要将计算机设置为运行 .NET 6 或更高版本。 Visual Studio 2022 或 .NET SDK 中提供了 C# 编译器。

记录的特征

要定义 *记录*，需要先使用 `record` 关键字来声明一种类型，然后修改 `class` 或 `struct` 声明。 （可选）可以省略 `class` 关键字来创建 `record class`。 记录遵循基于值的相等语义。 为了强制实施值语义，编译器会为记录类型生成多种方法（适用于 `record class` 类型和 `record struct` 类型）：

- Object.Equals(Object) 的替代。
- 参数为记录类型的虚拟 `Equals` 方法。
- Object.GetHashCode() 的替代。
- 用于 `operator ==` 和 `operator !=` 的方法。
- 记录类型实现 System.IEquatable<T>。

记录还提供了 Object.ToString() 的重写。 编译器使用 Object.ToString()生成用于显示记录的方法。 在编写本教程的代码时，你将浏览这些成员。 记录支持 `with` 表达式，以启用记录的非破坏性修改。

你还可以使用更简洁的语法声明 *位置记录*。 声明位置记录时，编译器会为你合成更多方法：

- 一个主构造函数，其参数与记录声明上的位置参数匹配。
- 主构造函数的每个参数的公共属性。 对于 `record class` 和 `readonly record struct` 类型，这些属性为 init-only。 对于 `record struct` 类型，它们是可读写的。
- 用于从记录中提取属性的 `Deconstruct` 方法。

生成温度数据

数据和统计信息是需要使用记录的场景之一。 在本教程中，你将构建一个用于计算度日数的应用程序，以用于不同用途。 *度日数*是反映几天、几周或几个月内采暖（或采暖不足）的度量。 度日数可跟踪和预测能源使用情况。 更热的日子意味着更多的空调，更冷的日子意味着更多的炉子使用。 度日数有助于管理植物种群，并且随着季节的变化，与植物的生长密切相关。 度日数有助于跟踪动物为适应气候而进行的物种迁徙。

公式基于给定日的平均温度和基线温度。 若要计算一段时间内的度日数，需要这段时间的每日最高温度和最低温度。 首先创建一个新应用程序。 创建新的控制台应用程序。 在名为“DailyTemperature.cs”的新文件中创建新记录类型：

C#

```
public readonly record struct DailyTemperature(double HighTemp, double
LowTemp);
```

前面的代码定义一个 *位置记录*。 由于不打算从 `DailyTemperature` 记录继承并且该记录应该不可变，因此该记录为 `readonly record struct`。 `HighTemp` 和 `LowTemp` 属性是 init-only 属性，这意味着可在构造函数中设置它们，或使用属性初始化表达式设置它们。 如果希望位置参数是可读可写的，则应该声明 `record struct` 而不是 `readonly record struct`。 `DailyTemperature` 类型还具有一个 *主构造函数*，该构造函数具有两个与两个属性匹配的参数。 使用该主构造函数初始化 `DailyTemperature` 记录。 以下代码创建并初始化多个 `DailyTemperature` 记录。 第一个使用命名参数来阐明 `HighTemp` 和 `LowTemp`。 剩余的初始值设定项使用位置参数来初始化 `HighTemp` 和 `LowTemp`：

C#

```
private static DailyTemperature[] data = [
    new DailyTemperature(HighTemp: 57, LowTemp: 30),
    new DailyTemperature(60, 35),
    new DailyTemperature(63, 33),
    new DailyTemperature(68, 29),
    new DailyTemperature(72, 47),
    new DailyTemperature(75, 55),
    new DailyTemperature(77, 55),
    new DailyTemperature(72, 58),
    new DailyTemperature(70, 47),
    new DailyTemperature(77, 59),
```

```
    new DailyTemperature(85, 65),
    new DailyTemperature(87, 65),
    new DailyTemperature(85, 72),
    new DailyTemperature(83, 68),
    new DailyTemperature(77, 65),
    new DailyTemperature(72, 58),
    new DailyTemperature(77, 55),
    new DailyTemperature(76, 53),
    new DailyTemperature(80, 60),
    new DailyTemperature(85, 66)
];
```

可以将自己的属性或方法添加到记录，包括位置记录。 你需要计算每天的平均温度。 可以将该属性添加到 `DailyTemperature` 记录：

C#

```
public readonly record struct DailyTemperature(double HighTemp, double
LowTemp)
{
    public double Mean => (HighTemp + LowTemp) / 2.0;
}
```

让我们一起确保你可以使用这些数据。 将以下代码添加到 `Main` 方法：

C#

```
foreach (var item in data)
    Console.WriteLine(item);
```

运行应用程序，你会看到类似于以下显示的输出（删除了几行空间）：

.NET CLI

```
DailyTemperature { HighTemp = 57, LowTemp = 30, Mean = 43.5 }
DailyTemperature { HighTemp = 60, LowTemp = 35, Mean = 47.5 }


DailyTemperature { HighTemp = 80, LowTemp = 60, Mean = 70 }
DailyTemperature { HighTemp = 85, LowTemp = 66, Mean = 75.5 }
```

上述代码显示了由编译器合成的 `ToString` 的替代输出。 如果你更喜欢不同的文本，可以编写自己的 `ToString` 版本，以防止编译器为你合成版本。

计算度日数

若要计算度日数，需要获得给定的某一天的基准温度和平均温度之间的差额。 若要测量随时间推移的热量，请丢弃平均温度低于基线的任何日期。 若要测量一段时间内的冷度，请放弃平均温度高于基线的任何日期。 例如，美国将 65 华氏度作为供暖和制冷度日计算的基准。 这是不需要加热或冷却的温度。 如果一天的平均温度为 70 F，那天是五度冷却日和零加热度日。 相反，如果平均温度为 55 华氏度，那么那一天是 10 个采暖度日和 0 个制冷度日。

可将这些公式表示为记录类型的小型层次结构：一种抽象度日数类型以及两种具体的采暖度日数和制冷度日数类型。 这些类型也可以是位置记录。 它们采用基线温度和每日温度记录序列作为主要构造函数的参数：

C#

```
public abstract record DegreeDays(double BaseTemperature,
IEnumerable<DailyTemperature> TempRecords);

public sealed record HeatingDegreeDays(double BaseTemperature,
IEnumerable<DailyTemperature> TempRecords)
    : DegreeDays(BaseTemperature, TempRecords)
{
    public double DegreeDays => TempRecords.Where(s => s.Mean <
BaseTemperature).Sum(s => BaseTemperature - s.Mean);
}

public sealed record CoolingDegreeDays(double BaseTemperature,
IEnumerable<DailyTemperature> TempRecords)
    : DegreeDays(BaseTemperature, TempRecords)
{
    public double DegreeDays => TempRecords.Where(s => s.Mean >
BaseTemperature).Sum(s => s.Mean - BaseTemperature);
}
```

抽象 `DegreeDays` 记录是 `HeatingDegreeDays` 和 `CoolingDegreeDays` 记录的共享基类。 派生记录上的主要构造函数声明显示如何管理基本记录初始化。 派生记录为基本记录主构造函数中的所有参数声明参数。 基记录声明并初始化这些属性。 派生记录不会隐藏它们，但只会为其基记录中未声明的参数创建和初始化属性。 在此示例中，派生记录不会添加新的主构造函数参数。 将以下代码添加到 `Main` 方法中以对代码进行测试：

C#

```
var heatingDegreeDays = new HeatingDegreeDays(65, data);
Console.WriteLine(heatingDegreeDays);

var coolingDegreeDays = new CoolingDegreeDays(65, data);
Console.WriteLine(coolingDegreeDays);
```

您将看到如下所示的输出：

.NET CLI

```
HeatingDegreeDays { BaseTemperature = 65, TempRecords =
record_types.DailyTemperature[], DegreeDays = 85 }
CoolingDegreeDays { BaseTemperature = 65, TempRecords =
record_types.DailyTemperature[], DegreeDays = 71.5 }
```

定义由编译器自动生成的方法

代码将计算该时间段内正确的取暖度和冷却度天数。 但此示例展示了为何需要替换记录的某些合成方法。 可以在记录类型中声明自己的任何编译器合成方法版本，但克隆方法除外。 克隆方法具有编译器生成的名称，不能提供不同的实现。 这些合成方法包括复制构造函数、System.IEquatable<T> 接口的成员、相等性和不相等测试以及 GetHashCode()。 为此，可以合成 `PrintMembers`。 你还可以声明自己的 `ToString`，但 `PrintMembers` 为继承方案提供了更好的选择。 若要提供自己的合成方法版本，签名必须与合成方法匹配。

控制台输出中的 `TempRecords` 元素不起作用。 它显示类型，但不显示其他任何东西。 可以通过提供您自己的合成 `PrintMembers` 方法的实现来更改此行为。 签名取决于应用于 `record` 声明的修饰符：

- 如果记录类型为 `sealed` 或 `record struct`，则签名 `private bool PrintMembers(StringBuilder builder);`
- 如果记录类型不是 `sealed` 并且从 `object` 派生（即，它不声明基记录），那么签名是 `protected virtual bool PrintMembers(StringBuilder builder);`
- 如果记录类型不是 `sealed` 且派生自另一条记录，则签名为 `protected override bool PrintMembers(StringBuilder builder);`

通过了解 `PrintMembers` 的目的，可以更容易理解这些规则。 `PrintMembers` 向字符串添加记录类型中每个属性的相关信息。 该协定要求基本记录添加其要显示的成员，并假设派生成员将添加其成员。 每个记录类型都会合成一个 `ToString` 替代，与下面的 `HeatingDegreeDays` 示例类似：

C#

```
public override string ToString()
{
    StringBuilder stringBuilder = new StringBuilder();
    stringBuilder.Append("HeatingDegreeDays");
    stringBuilder.Append(" { ");
    if (PrintMembers(stringBuilder))
    {
        stringBuilder.Append(" ");
```

```
    }
    stringBuilder.Append("}");
    return stringBuilder.ToString();
}
```

在不打印集合类型的 `DegreeDays` 记录中声明 `PrintMembers` 方法：

C#

```
protected virtual bool PrintMembers(StringBuilder stringBuilder)
{
    stringBuilder.Append($"BaseTemperature = {BaseTemperature}");
    return true;
}
```

签名声明 `virtual protected` 方法以匹配编译器的版本。 如果访问器出错，请不要担心；语言会强制执行正确的签名。 如果忘记了任何合成方法的正确修饰符，编译器会发出警告或错误，帮助你获取正确的签名。

可以在记录类型中将 `ToString` 方法声明为 `sealed`。 这会阻止派生记录提供新的实现。 派生记录将仍包含 `PrintMembers` 替代。 如果你不希望 `ToString` 显示记录的运行时类型，你就会将其密封。 在前面的示例中，你会丢失有关记录测量取暖度日数或降温度日数的位置信息。

非破坏性突变

位置记录类中的合成成员不会修改记录的状态。 目标是更轻松地创建不可变记录。 请记住，声明 `readonly record struct` 以创建不可变记录结构。 请再次查看前面的关于 `HeatingDegreeDays` 和 `CoolingDegreeDays` 的声明。 添加的成员对记录的值执行计算，但不会改变状态。 通过位置记录，可以更轻松地创建不可变引用类型。

创建不可变引用类型意味着要使用非破坏性突变。 使用 with 表达式创建与现有记录实例类似的新记录实例。 这些表达式是通过复制构造来创建的，并通过额外的赋值操作来修改该副本。 结果是一个新的记录实例，其中每个属性都从现有记录复制并选择性地修改。 原始记录保持不变。

让我们向程序添加一些演示 `with` 表达式的功能。 首先，创建一条新记录，使用相同数据计算增长的度日数。 增长的度日数通常使用 41 F 作为基准，并测量超出基准的温度。 若要使用相同的数据，可以创建类似于 `coolingDegreeDays` 的新记录，但具有不同的基本温度：

C#

```
// Growing degree days measure warming to determine plant growing rates
var growingDegreeDays = coolingDegreeDays with { BaseTemperature = 41 };
Console.WriteLine(growingDegreeDays);
```

可以将计算的度数与使用较高基线温度生成的数字进行比较。 请记住，记录是引用类型，这些副本是浅表副本。 不会复制数据的数组，但两条记录都引用相同的数据。 这一事实在另一个情况下是一个优势。 对于温度增长的日数，记录前 5 天的总度数非常有用。 可以使用 `with` 表达式创建具有不同源数据的新记录。 以下代码生成这些累积的集合，然后显示值：

C#

```
// showing moving accumulation of 5 days using range syntax
List<CoolingDegreeDays> movingAccumulation = new();
int rangeSize = (data.Length > 5) ? 5 : data.Length;
for (int start = 0; start < data.Length - rangeSize; start++)
{
    var fiveDayTotal = growingDegreeDays with { TempRecords = data[start..
(start + rangeSize)] };
    movingAccumulation.Add(fiveDayTotal);
}
Console.WriteLine();
Console.WriteLine("Total degree days in the last five days");
foreach(var item in movingAccumulation)
{
    Console.WriteLine(item);
}
```

还可以使用 `with` 表达式来创建记录的副本。 不要在 `with` 表达式的大括号之间指定任何属性。 这意味着创建副本，并且不更改任何属性：

C#

```
var growingDegreeDaysCopy = growingDegreeDays with { };
```

运行已完成的应用程序以查看结果。

总结

本教程介绍了记录的几个方面。 记录为存储数据的基本用途的类型提供简洁的语法。 对于面向对象的类，基本用途是定义责任。 本教程重点介绍 *位置记录*，你可以在其中使用简洁的语法来声明记录的属性。 编译器合成记录的多个成员，用于复制和比较记录。 你可针对记录类型添加所需的任何其他成员。 你可以创建不可变的记录类型，知道编译器生成的成员都不会改变状态。 `with` 表达式可以轻松支持非破坏性突变。

记录添加另一种方法来定义类型。 使用 `class` 定义来创建面向对象的层次结构，这些层次结构侧重于对象的责任和行为。 为存储数据且足够小的数据结构创建 `struct` 类型，以便高效复制。 当想要基于值的相等性和比较、不想复制值以及想要使用引用变量时，可以创建 `record` 类型。 当希望某个类型的记录功能足够小，可以高效复制时，可以创建 `record struct` 类型。

要详细了解记录，请访问 记录类型的 C# 语言参考文章、建议的记录类型规范和记录结构规范。

教程：在学习过程中，探索使用顶级语句生成代码的想法

项目 • 2023/11/15

本教程介绍以下操作：

- ✔ 了解控制顶层语句使用的规则。
- ✔ 使用顶级语句了解算法。
- ✔ 重构对可重用组件的探索。

先决条件

需要将计算机设置为运行 .NET 6，其中包括 C# 10 编译器。 自 Visual Studio 2022 或 .NET 6 SDK 起，开始提供 C# 10 编译器。

本教程假设你熟悉 C# 和 .NET，包括 Visual Studio 或 .NET CLI。

开始探索

借助顶级语句，你可以将程序的入口点置于类的静态方法中，以避免额外的工作。 新控制台应用程序的典型起点类似于以下代码：

C#

```
using System;

namespace Application
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Hello World!");
        }
    }
}
```

上面的代码是运行 `dotnet new console` 命令并创建新控制台应用程序的结果。 这 11 行只包含一行可执行代码。 可以通过新的顶级语句功能简化该程序。 由此，你可以删除此程序中除两行以外的所有行：

C#

```
// See https://aka.ms/new-console-template for more information
Console.WriteLine("Hello, World!");
```

> **ⓘ 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 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

此功能简化了开始探索新想法所需的操作。 你可以将顶级语句用于脚本编写场景，或用于探索。 掌握基础知识后，便可以开始重构代码，并为生成的可重用组件创建方法、类或其他程序集。 顶级语句支持快速试验和初学者教程。 它们还提供一条从试验到完整程序的平滑路径。

顶级语句按照其在文件中出现的顺序执行。 顶级语句只能用在应用程序的一个源文件中。 如果将其用于多个文件，编译器将生成错误。

构建神奇的 .NET 应答机

对于本教程，让我们构建一个控制台应用程序，该应用程序使用随机答案回答“是”或“否”问题。 你将逐步生成功能。 你可以专注于你的任务，而不是典型程序结构所需的工作。 然后，当你满意该功能后，可根据自己的情况重构应用程序。

将问题写回控制台是一个良好的起点。 首先，可以编写以下代码：

```
C#

Console.WriteLine(args);
```

不声明 `args` 变量。 对于包含顶级语句的单个源文件，编译器会将 `args` 识别为表示命令行参数。 参数的类型是一个 `string[]`，就像在所有 C# 程序中一样。

你可以运行以下 `dotnet run` 命令对代码进行测试：

.NET CLI

```
dotnet run -- Should I use top level statements in all my programs?
```

命令行上 `--` 后面的参数将传递给程序。 你可以查看 `args` 变量的类型，因为该类型会输出到控制台：

控制台

```
System.String[]
```

若要将问题写入控制台，需要枚举参数，并使用空格分隔参数。 将 `WriteLine` 调用替换为以下代码：

C#

```
Console.WriteLine();
foreach(var s in args)
{
    Console.Write(s);
    Console.Write(' ');
}
Console.WriteLine();
```

现在，运行该程序时，它会将问题正确地显示为参数字符串。

使用随机答案响应

在回显问题后，你可以添加代码以生成随机答案。 首先添加可能的答案的数组：

C#

```
string[] answers =
[
    "It is certain.",       "Reply hazy, try again.",     "Don't count on
it.",
    "It is decidedly so.",  "Ask again later.",           "My reply is no.",
    "Without a doubt.",     "Better not tell you now.",   "My sources say
no.",
    "Yes - definitely.",    "Cannot predict now.",        "Outlook not so
good.",
    "You may rely on it.",  "Concentrate and ask again.", "Very doubtful.",
    "As I see it, yes.",
    "Most likely.",
```

```
    "Outlook good.",
    "Yes.",
    "Signs point to yes.",
];
```

此数组有 10 个肯定答案，5 个态度不明确的答案，以及 5 个否定答案。 接下来，添加以下代码以生成并显示来自数组的随机答案：

C#

```
var index = new Random().Next(answers.Length - 1);
Console.WriteLine(answers[index]);
```

你可以再次运行该应用程序以查看结果。 显示的内容应与以下输出类似：

.NET CLI

```
dotnet run -- Should I use top level statements in all my programs?

Should I use top level statements in all my programs?
Better not tell you now.
```

此代码回答了这些问题，但我们再添加一个功能。 你想让你的问题应用模拟对答案的思考。 可添加一小段 ASCII 动画并在工作时暂停，以实现此目的。 在回显问题的行后添加以下代码：

C#

```
for (int i = 0; i < 20; i++)
{
    Console.Write("| -");
    await Task.Delay(50);
    Console.Write("\b\b\b");
    Console.Write("/ \\");
    await Task.Delay(50);
    Console.Write("\b\b\b");
    Console.Write("- |");
    await Task.Delay(50);
    Console.Write("\b\b\b");
    Console.Write("\\ /");
    await Task.Delay(50);
    Console.Write("\b\b\b");
}
Console.WriteLine();
```

还需要将 `using` 语句添加到源文件顶部：

C#

```
using System.Threading.Tasks;
```

`using` 语句必须位于文件中的任何其他语句之前。 否则，会出现编译器错误。 可以再次运行该程序，并查看动画。 这样可以获得更好的体验。 试验一下延迟的时长，使其与你的喜好匹配。

上面的代码创建一组由空格分隔的旋转行。 添加 `await` 关键字可指示编译器生成程序入口点作为具有 `async` 修饰符的方法，并返回 System.Threading.Tasks.Task。 此程序不返回值，因此程序入口点返回 `Task`。 如果程序返回整数值，你可将 return 语句添加到顶级语句的末尾。 该 return 语句将指定要返回的整数值。 如果顶级语句包含 `await` 表达式，返回类型将变为 System.Threading.Tasks.Task<TResult>。

为满足未来需要而重构

程序应类似于以下代码：

C#

```
Console.WriteLine();
foreach(var s in args)
{
    Console.Write(s);
    Console.Write(' ');
}
Console.WriteLine();

for (int i = 0; i < 20; i++)
{
    Console.Write("| -");
    await Task.Delay(50);
    Console.Write("\b\b\b");
    Console.Write("/ \\");
    await Task.Delay(50);
    Console.Write("\b\b\b");
    Console.Write("- |");
    await Task.Delay(50);
    Console.Write("\b\b\b");
    Console.Write("\\ /");
    await Task.Delay(50);
    Console.Write("\b\b\b");
}
Console.WriteLine();

string[] answers =
[
    "It is certain.",       "Reply hazy, try again.",     "Don't count on
```

```
it.",
    "It is decidedly so.",  "Ask again later.",           "My reply is no.",
    "Without a doubt.",     "Better not tell you now.",   "My sources say
no.",
    "Yes - definitely.",    "Cannot predict now.",        "Outlook not so
good.",
    "You may rely on it.",  "Concentrate and ask again.", "Very doubtful.",
    "As I see it, yes.",
    "Most likely.",
    "Outlook good.",
    "Yes.",
    "Signs point to yes.",
];

var index = new Random().Next(answers.Length - 1);
Console.WriteLine(answers[index]);
```

前面的代码合理。 有效。 但它无法重用。 现在，应用程序正在运行，可以提取可重用的部分。

一个候选项是显示等待动画的代码。 该代码片段可以成为一种方法：

首先，可以在文件中创建本地函数。 将当前动画替换为以下代码：

C#

```
await ShowConsoleAnimation();

static async Task ShowConsoleAnimation()
{
    for (int i = 0; i < 20; i++)
    {
        Console.Write("| -");
        await Task.Delay(50);
        Console.Write("\b\b\b");
        Console.Write("/ \\");
        await Task.Delay(50);
        Console.Write("\b\b\b");
        Console.Write("- |");
        await Task.Delay(50);
        Console.Write("\b\b\b");
        Console.Write("\\ /");
        await Task.Delay(50);
        Console.Write("\b\b\b");
    }
    Console.WriteLine();
}
```

前面的代码在 main 方法中创建本地函数。 这仍不可重用。 因此，将该代码提取到类中。 创建名为 utilities.cs 的新文件，并添加以下代码：

C#

```
namespace MyNamespace
{
    public static class Utilities
    {
        public static async Task ShowConsoleAnimation()
        {
            for (int i = 0; i < 20; i++)
            {
                Console.Write("| -");
                await Task.Delay(50);
                Console.Write("\b\b\b");
                Console.Write("/ \\");
                await Task.Delay(50);
                Console.Write("\b\b\b");
                Console.Write("- |");
                await Task.Delay(50);
                Console.Write("\b\b\b");
                Console.Write("\\ /");
                await Task.Delay(50);
                Console.Write("\b\b\b");
            }
            Console.WriteLine();
        }
    }
}
```

具有顶级语句的文件还可以在顶级语句后，在文件末尾包含命名空间和类型。 但对于本教程，请将动画方法放在一个单独的文件中，使其更易于重复使用。

最后，你可以清理动画代码以删除一些重复项：

C#

```
foreach (string s in animations)
{
    Console.Write(s);
    await Task.Delay(50);
    Console.Write("\b\b\b");
}
```

现在你有了一个完整的应用程序，并且已经重构了可重用部分供以后使用。 可以从顶级语句中调用新的实用工具方法，如下面主程序的最终版本所示：

C#

```
using MyNamespace;

Console.WriteLine();
```

```
foreach(var s in args)
{
    Console.Write(s);
    Console.Write(' ');
}
Console.WriteLine();

await Utilities.ShowConsoleAnimation();

string[] answers =
[
    "It is certain.",       "Reply hazy, try again.",     "Don't count on
it.",
    "It is decidedly so.",  "Ask again later.",           "My reply is no.",
    "Without a doubt.",     "Better not tell you now.",   "My sources say
no.",
    "Yes - definitely.",    "Cannot predict now.",        "Outlook not so
good.",
    "You may rely on it.",  "Concentrate and ask again.", "Very doubtful.",
    "As I see it, yes.",
    "Most likely.",
    "Outlook good.",
    "Yes.",
    "Signs point to yes.",
];

var index = new Random().Next(answers.Length - 1);
Console.WriteLine(answers[index]);
```

上述示例增加了对 `Utilities.ShowConsoleAnimation` 的调用，并增加了一条 `using` 语句。

摘要

使用顶级语句，可以更轻松地创建简单的程序来探索新的算法。 可以尝试使用不同的代码片段来试验算法。 了解了哪些可用后，可以重构代码，使其更易于维护。

顶级语句可简化基于控制台应用程序的程序。 其中包括 Azure Functions、GitHub 操作和其他小实用工具。 有关详细信息，请参阅顶级语句（C# 编程指南）。

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

索引和范围

项目 • 2023/11/14

范围和索引为访问序列中的单个元素或范围提供了简洁的语法。

在本教程中，你将了解：

- ✓ 对某个序列中的范围使用该语法。
- ✓ 隐式定义 Range。
- ✓ 了解每个序列开头和末尾的设计决策。
- ✓ 了解 Index 和 Range 类型的应用场景。

对索引和范围的语言支持

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 ..，用于将范围的开头和末尾指定为其操作数。

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。 表达式 `sequence[^0]` 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。

C#

```
string[] words = [
                 // index from start    index from end
    "The",       // 0                   ^9
    "quick",     // 1                   ^8
    "brown",     // 2                   ^7
    "fox",       // 3                   ^6
    "jumps",     // 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。 在初始化下面添加以下代码：

C#

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

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。

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。

C#

```
string[] quickBrownFox = words[1..4];
foreach (var word in quickBrownFox)
    Console.Write($"< {word} >");
Console.WriteLine();
```

以下代码使用“lazy”和“dog”返回范围。 它包括 `words[^2]` 和 `words[^1]`。 结束索引 `words[^0]` 不包括在内。 同样添加以下代码：

C#

```
string[] lazyDog = words[^2..^0];
foreach (var word in lazyDog)
    Console.Write($"< {word} >");
Console.WriteLine();
```

下面的示例为开始和/或结束创建了开放范围：

C#

```
string[] allWords = words[..]; // contains "The" through "dog".
string[] firstPhrase = words[..4]; // contains "The" through "fox"
string[] lastPhrase = words[6..]; // contains "the", "lazy" and "dog"
foreach (var word in allWords)
    Console.Write($"< {word} >");
Console.WriteLine();
foreach (var word in firstPhrase)
    Console.Write($"< {word} >");
Console.WriteLine();
foreach (var word in lastPhrase)
    Console.Write($"< {word} >");
Console.WriteLine();
```

还可以将范围或索引声明为变量。 然后可以在 `[` 和 `]` 字符中使用该变量：

C#

```
Index the = ^3;
Console.WriteLine(words[the]);
Range phrase = 1..4;
string[] text = words[phrase];
foreach (var word in text)
    Console.Write($"< {word} >");
Console.WriteLine();
```

下面的示例展示了使用这些选项的多种原因。 请修改 `x`、`y` 和 `z` 以尝试不同的组合。 在进行实验时，请使用 `x` 小于 `y` 且 `y` 小于 `z` 的有效组合值。 在新方法中添加以下代码。 尝试不同的组合：

C#

```
int[] numbers = [..Enumerable.Range(0, 100)];
int x = 12;
int y = 25;
int z = 36;

Console.WriteLine($"{numbers[^x]} is the same as {numbers[numbers.Length -
x]}");
Console.WriteLine($"{numbers[x..y].Length} is the same as {y - x}");

Console.WriteLine("numbers[x..y] and numbers[y..z] are consecutive and
disjoint:");
Span<int> x_y = numbers[x..y];
Span<int> y_z = numbers[y..z];
Console.WriteLine($"\tnumbers[x..y] is {x_y[0]} through {x_y[^1]},
numbers[y..z] is {y_z[0]} through {y_z[^1]}");

Console.WriteLine("numbers[x..^x] removes x elements at each end:");
Span<int> x_x = numbers[x..^x];
Console.WriteLine($"\tnumbers[x..^x] starts with {x_x[0]} and ends with
{x_x[^1]}");

Console.WriteLine("numbers[..x] means numbers[0..x] and numbers[x..] means
numbers[x..^0]");
Span<int> start_x = numbers[..x];
Span<int> zero_x = numbers[0..x];
Console.WriteLine($"\t{start_x[0]}..{start_x[^1]} is the same as
{zero_x[0]}..{zero_x[^1]}");
Span<int> z_end = numbers[z..];
Span<int> z_zero = numbers[z..^0];
Console.WriteLine($"\t{z_end[0]}..{z_end[^1]} is the same as {z_zero[0]}..
{z_zero[^1]}");
```

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>。

隐式范围运算符表达式转换

使用范围运算符表达式语法时，编译器会将开始值和结束值隐式转换为 Index，并根据这些值创建新的 Range 实例。 以下代码显示了范围运算符表达式语法的隐式转换示例及其对应的显式替代方法：

C#

```
Range implicitRange = 3..^5;

Range explicitRange = new(
    start: new Index(value: 3, fromEnd: false),
    end: new Index(value: 5, fromEnd: true));

if (implicitRange.Equals(explicitRange))
{
    Console.WriteLine(
        $"The implicit range '{implicitRange}' equals the explicit range
'{explicitRange}'");
}
// Sample output:
//     The implicit range '3..^5' equals the explicit range '3..^5'
```

> **重要**
>
> 当值为负数时，从 Int32 隐式转换为 Index 会引发 ArgumentOutOfRangeException。 同样，当 `value` 参数为负时，`Index` 构造函数会引发 `ArgumentOutOfRangeException`。

索引和范围的类型支持

索引和范围提供清晰、简洁的语法来访问序列中的单个元素或元素的范围。 索引表达式通常返回序列元素的类型。 范围表达式通常返回与源序列相同的序列类型。

若任何类型提供带 Index 或 Range 参数的索引器，则该类型可分别显式支持索引或范围。 采用单个 Range 参数的索引器可能会返回不同的序列类型，如 System.Span<T>。

> **重要**
>
> 使用范围运算符的代码的性能取决于序列操作数的类型。
>
> 范围运算符的时间复杂度取决于序列类型。 例如，如果序列是 `string` 或数组，则结果是输入中指定部分的副本，因此，时间复杂度为 O(N)（其中 N 是范围的长

度）。 另一方面，如果它是 System.Span<T> 或 System.Memory<T>，则结果引用相同的后备存储，这意味着没有副本且操作为 O(1)。

除了时间复杂度外，这还会产生额外的分配和副本，从而影响性能。 在性能敏感的代码中，考虑使用 `Span<T>` 或 `Memory<T>` 作为序列类型，因为不会为其分配范围运算符。

若类型包含名称为 `Length` 或 `Count` 的属性，属性有可访问的 Getter 并且其返回类型为 `int`，则此类型为可计数类型。 不显式支持索引或范围的可计数类型可能为它们提供隐式支持。 有关详细信息，请参阅功能建议说明的隐式索引支持和隐式范围支持部分。 使用隐式范围支持的范围将返回与源序列相同的序列类型。

例如，以下 .NET 类型同时支持索引和范围：String、Span<T> 和 ReadOnlySpan<T>。 List<T> 支持索引，但不支持范围。

Array 具有更多的微妙行为。 单个维度数组同时支持索引和范围。 多维数组不支持索引器或范围。 多维数组的索引器具有多个参数，而不是一个参数。 交错数组（也称为数组的数组）同时支持范围和索引器。 下面的示例演示如何循环访问交错数组的矩形子节。 它循环访问位于中心的节，不包括前三行和后三行，以及每个选定行中的前两列和后两列：

C#

```
int[][] jagged =
[
   [0, 1, 2, 3, 4, 5, 6, 7, 8, 9],
   [10,11,12,13,14,15,16,17,18,19],
   [20,21,22,23,24,25,26,27,28,29],
   [30,31,32,33,34,35,36,37,38,39],
   [40,41,42,43,44,45,46,47,48,49],
   [50,51,52,53,54,55,56,57,58,59],
   [60,61,62,63,64,65,66,67,68,69],
   [70,71,72,73,74,75,76,77,78,79],
   [80,81,82,83,84,85,86,87,88,89],
   [90,91,92,93,94,95,96,97,98,99],
];

var selectedRows = jagged[3..^3];

foreach (var row in selectedRows)
{
    var selectedColumns = row[2..^2];
    foreach (var cell in selectedColumns)
    {
        Console.Write($"{cell}, ");
    }
```

```
    Console.WriteLine();
}
```

在所有情况下，Array 的范围运算符都会分配一个数组来存储返回的元素。

索引和范围的应用场景

要分析较大序列的一部分时，通常会使用范围和索引。 在准确读取所涉及的序列部分这一方面，新语法更清晰。 本地函数 `MovingAverage` 以 Range 为参数。 然后，该方法在计算最小值、最大值和平均值时仅枚举该范围。 在项目中尝试以下代码：

C#

```
int[] sequence = Sequence(1000);

for(int start = 0; start < sequence.Length; start += 100)
{
    Range r = start..(start+10);
    var (min, max, average) = MovingAverage(sequence, r);
    Console.WriteLine($"From {r.Start} to {r.End}:    \tMin: {min},\tMax:
{max},\tAverage: {average}");
}

for (int start = 0; start < sequence.Length; start += 100)
{
    Range r = ^(start + 10)..^start;
    var (min, max, average) = MovingAverage(sequence, r);
    Console.WriteLine($"From {r.Start} to {r.End}:  \tMin: {min},\tMax:
{max},\tAverage: {average}");
}

(int min, int max, double average) MovingAverage(int[] subSequence, Range
range) =>
    (
        subSequence[range].Min(),
        subSequence[range].Max(),
        subSequence[range].Average()
    );

int[] Sequence(int count) => [..Enumerable.Range(0, count).Select(x => (int)
(Math.Sqrt(x) * 100))];
```

关于范围索引和数组的说明

从数组中获取范围时，结果是从初始数组复制的数组，而不是引用的数组。 修改生成的数组中的值不会更改初始数组中的值。

例如：

```C#
var arrayOfFiveItems = new[] { 1, 2, 3, 4, 5 };

var firstThreeItems = arrayOfFiveItems[..3]; // contains 1,2,3
firstThreeItems[0] =  11; // now contains 11,2,3

Console.WriteLine(string.Join(",", firstThreeItems));
Console.WriteLine(string.Join(",", arrayOfFiveItems));

// output:
// 11,2,3
// 1,2,3,4,5
```

另请参阅

- 成员访问运算符和表达式

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 feedback

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

提出文档问题

提供产品反馈

教程：使用可为空和不可为空引用类型更清晰地表达设计意图

项目 • 2023/05/10

可为空引用类型采用与可为空值类型补充值类型相同的方式补充引用类型。 通过将 `?` 追加到此类型，你可以将变量声明为 可为空引用类型。 例如，`string?` 表示可为空的 `string`。 可以使用这些新类型更清楚地表达你的设计意图：某些变量 必须始终具有值，其他变量可以缺少值 。

在本教程中，你将了解：

- ✓ 将可为空和不可为空引用类型合并到你的设计中
- ✓ 在整个代码中启用可为空引用类型检查。
- ✓ 编写编译器强制执行这些设计决策的代码。
- ✓ 在自己的设计中使用可为空引用功能

先决条件

需要将计算机设置为运行 .NET，包括 C# 编译器。 Visual Studio 2022 或 .NET SDK 随附 C# 编译器。

本教程假设你熟悉 C# 和 .NET，包括 Visual Studio 或 .NET CLI。

将可为空引用类型合并到你的设计中

在本教程中，你将构建一个模拟运行调查的库。 该代码使用可为空引用类型和不可为可空引用类型来表示实际概念。 调查问题决不会为 NULL。 回应者可能不愿回答某个问题。 在这种情况下响应可能为 `null`。

你为此示例编写的代码表示该意向，并且编译器强制执行该意向。

创建应用程序并启用可为空引用类型

在 Visual Studio 中或使用 `dotnet new console` 从命令行创建新的控制台应用程序。 命名应用程序 `NullableIntroduction`。 创建应用程序后，需要指定整个项目都在启用的“可为空注释上下文”中进行编译 。 打开 .csproj 文件，并向 `PropertyGroup` 元素添加 `Nullable` 元素 。 将其值设置为 `enable`。 在 C# 11 之前的项目中，必须选择“可为空引用类型”功能。 这是因为，一旦启用该功能，现有的引用变量声明将成为不可为空引用类型 。 尽管

该决定将有助于发现现有代码可能不具有适当的 NULL 检查的问题，但它可能无法准确反映你的原始设计意图：

```
XML

<Nullable>enable</Nullable>
```

在 .NET 6 之前，新项目不包含 `Nullable` 元素。 从 .NET 6 开始，新项目将在项目文件中包含 `<Nullable>enable</Nullable>` 元素。

设计应用程序的类型

此调查应用程序需要创建许多类：

- 建模问题列表的类。
- 建模为调查所联系的人员列表的类。
- 建模来自参加调查人员的答案的类。

这些类型将使用可为空和不可为空引用类型来表示哪些成员是必需的，哪些成员是可选的。 可为空引用类型清楚地传达了设计意图：

- 调查中的问题不可为 null：提出空问题没有任何意义。
- 回应者永远不能为 NULL。 你需要跟踪所联系的人员，即便回应者拒绝参与也是如此。
- 对某个问题的任何响应都可能为 NULL。 回应者可拒绝回答部分或全部问题。

如果你使用 C# 编程，则可能已经习惯于允许 `null` 值的引用类型，但你可能错过了其他声明不可为空实例的机会：

- 问题集合应不可为空。
- 回应者集合应不可为空。

在编写代码时，你将看到不可为空引用类型作为引用的默认值，可避免可能导致 NullReferenceException 的常见错误。 从本教程得出的一个经验是，你应决定哪些变量可为或不可为 `null`。 该语言未提供表达这些决定的语法。 现在它可提供此项功能。

你构建的应用程序将执行以下步骤：

1. 创建调查并向其添加问题。
2. 为调查创建一组伪随机回应者。
3. 联系回应者，直到已完成的调查规模达到目标数量。
4. 写出有关调查响应的重要统计数据。

使用可为 null 和不可为 null 引用类型构建调查

你将编写的第一个代码创建调查。 你将编写类来为调查问题和调查运行建模。 调查有三种类型的问题，通过答案格式进行区分：答案为“是”/“否”、答案为数字以及答案为文本。 创建 `public SurveyQuestion` 类：

C#

```
namespace NullableIntroduction
{
    public class SurveyQuestion
    {
    }
}
```

编译器将在启用的可为空的注释上下文中的代码的每个引用类型变量声明解释为“不可为空”引用类型 。 你可以通过添加问题文本的属性和问题类型来查看第一个警告，如以下代码所示：

C#

```
namespace NullableIntroduction
{
    public enum QuestionType
    {
        YesNo,
        Number,
        Text
    }

    public class SurveyQuestion
    {
        public string QuestionText { get; }
        public QuestionType TypeOfQuestion { get; }
    }
}
```

因为尚未初始化 `QuestionText`，所以编译器会发出警告，指出尚未初始化不可为空属性。 设计要求问题文本为非空，因此需要添加构造函数来初始化它以及 `QuestionType` 值。 已完成类定义类似于以下代码：

C#

```
namespace NullableIntroduction;

public enum QuestionType
{
    YesNo,
```

```
    Number,
    Text
}

public class SurveyQuestion
{
    public string QuestionText { get; }
    public QuestionType TypeOfQuestion { get; }

    public SurveyQuestion(QuestionType typeOfQuestion, string text) =>
        (TypeOfQuestion, QuestionText) = (typeOfQuestion, text);
}
```

添加构造函数会删除警告。 构造函数参数也是不可为空引用类型，因此编译器不会发出任何警告。

接下来，创建一个名为 `SurveyRun` 的 `public` 类。 此类包含 `SurveyQuestion` 对象的列表以及向调查添加问题的方法，如以下代码所示：

C#

```
using System.Collections.Generic;

namespace NullableIntroduction
{
    public class SurveyRun
    {
        private List<SurveyQuestion> surveyQuestions = new
List<SurveyQuestion>();

        public void AddQuestion(QuestionType type, string question) =>
            AddQuestion(new SurveyQuestion(type, question));
        public void AddQuestion(SurveyQuestion surveyQuestion) =>
surveyQuestions.Add(surveyQuestion);
    }
}
```

和以前一样，你必须将列表对象初始化为非空值，否则编译器会发出警告。 在 `AddQuestion` 的第二次重载中没有 NULL 检查，因为不需要进行二次检查：已声明该变量不可为空。 其值不可为 `null`。

切换到编辑器中的 Program.cs ，并使用以下代码行替换 `Main` 的内容：

C#

```
var surveyRun = new SurveyRun();
surveyRun.AddQuestion(QuestionType.YesNo, "Has your code ever thrown a
NullReferenceException?");
surveyRun.AddQuestion(new SurveyQuestion(QuestionType.Number, "How many
```

```
times (to the nearest 100) has that happened?"));
surveyRun.AddQuestion(QuestionType.Text, "What is your favorite color?");
```

由于整个项目处于启用的可为空的注释上下文中，因此将 `null` 传递给任何应为不可为空引用类型的方法时，将收到警告。 通过将以下行添加到 `Main` 进行尝试：

C#

```
surveyRun.AddQuestion(QuestionType.Text, default);
```

创建回应者并获取调查答案

接下来，编写生成调查答案的代码。 此过程涉及到多个小型任务：

1. 构建一个生成回应者对象的方法。 这些对象表示要求填写调查的人员。
2. 生成逻辑以模拟向回应者询问问题并收集答案，或者注意到回应者没有回答。
3. 重复以上过程，直到有足够的回应者回答此调查。

你需要一个表示调查响应的类，所以现在就添加它。 启用可为空支持。 添加初始化它的 `Id` 属性和构造函数，如以下代码所示：

C#

```
namespace NullableIntroduction
{
    public class SurveyResponse
    {
        public int Id { get; }

        public SurveyResponse(int id) => Id = id;
    }
}
```

接下来，通过生成随机 ID 添加 `static` 方法来创建新参与者：

C#

```
private static readonly Random randomGenerator = new Random();
public static SurveyResponse GetRandomId() => new
SurveyResponse(randomGenerator.Next());
```

该类的主要职责是为调查中问题的参与者生成问题答案。 实现此职责有几个步骤：

1. 要求参加这项调查。 如果此人不同意，则返回缺失（或 NULL）响应。
2. 询问每个问题并记录答案。 每个答案也可能会缺失（或 NULL）。

将以下代码添加到 `SurveyResponse` 类：

C#

```
private Dictionary<int, string>? surveyResponses;
public bool AnswerSurvey(IEnumerable<SurveyQuestion> questions)
{
    if (ConsentToSurvey())
    {
        surveyResponses = new Dictionary<int, string>();
        int index = 0;
        foreach (var question in questions)
        {
            var answer = GenerateAnswer(question);
            if (answer != null)
            {
                surveyResponses.Add(index, answer);
            }
            index++;
        }
    }
    return surveyResponses != null;
}

private bool ConsentToSurvey() => randomGenerator.Next(0, 2) == 1;

private string? GenerateAnswer(SurveyQuestion question)
{
    switch (question.TypeOfQuestion)
    {
        case QuestionType.YesNo:
            int n = randomGenerator.Next(-1, 2);
            return (n == -1) ? default : (n == 0) ? "No" : "Yes";
        case QuestionType.Number:
            n = randomGenerator.Next(-30, 101);
            return (n < 0) ? default : n.ToString();
        case QuestionType.Text:
        default:
            switch (randomGenerator.Next(0, 5))
            {
                case 0:
                    return default;
                case 1:
                    return "Red";
                case 2:
                    return "Green";
                case 3:
                    return "Blue";
            }
            return "Red. No, Green. Wait.. Blue... AAARGGGGGHHH!";
    }
}
```

调查答案的存储空间为 `Dictionary<int, string>?`，表示它可能为 NULL。 你正在使用新的语言功能向编译器和稍后阅读你的代码的任何人声明你的设计意图。 如果在不首先检查是否为 `null` 值的情况下取消引用 `surveyResponses`，则会收到编译器警告。 你没有在 `AnswerSurvey` 方法中收到警告，因为编译器可以确定 `surveyResponses` 变量已设置为上述非空值。

对缺少的答案使用 `null` 强调了处理可为空引用类型的一个关键点：目标不是从程序中删除所有 `null` 值。 而是确保编写的代码表达设计意图。 缺失值是在代码中进行表达的一个必需概念。 `null` 值是表示这些缺失值的一种明确方法。 尝试删除所有 `null` 值只会导致定义一些其他方法来在没有 `null` 的情况下表示缺失值。

接下来，你需要在 `SurveyRun` 类中编写 `PerformSurvey` 方法。 将下面的代码添加到 `SurveyRun` 类中：

C#

```
private List<SurveyResponse>? respondents;
public void PerformSurvey(int numberOfRespondents)
{
    int respondentsConsenting = 0;
    respondents = new List<SurveyResponse>();
    while (respondentsConsenting < numberOfRespondents)
    {
        var respondent = SurveyResponse.GetRandomId();
        if (respondent.AnswerSurvey(surveyQuestions))
            respondentsConsenting++;
        respondents.Add(respondent);
    }
}
```

同样，你选择的可为空 `List<SurveyResponse>?` 指示响应可能为 NULL。 这表明尚未向任何回应者提供调查。 请注意，在同意参与调查的回应者未达到足够数量之前，不会添加回应者。

运行调查的最后一步是添加一个调用，从而可在 `Main` 方法结束时执行此调查：

C#

```
surveyRun.PerformSurvey(50);
```

检查调查响应

最后一步是显示调查结果。 将代码添加到你所编写的诸多类。 此代码演示了区分可为空和不可为空引用类型的值。 首先将以下两个表达式形式成员添加到 `SurveyResponse` 类：

C#

```
public bool AnsweredSurvey => surveyResponses != null;
public string Answer(int index) => surveyResponses?.GetValueOrDefault(index)
?? "No answer";
```

因为 `surveyResponses` 是一个不可为空引用，所以在取消引用之前不需要输入任何检查。`Answer` 方法返回不可为空的字符串，因此我们必须使用 null 合并运算符来涵盖缺少答案的情况。

接下来，将这三个表达式形式成员添加到 `SurveyRun` 类：

C#

```
public IEnumerable<SurveyResponse> AllParticipants => (respondents ??
Enumerable.Empty<SurveyResponse>());
public ICollection<SurveyQuestion> Questions => surveyQuestions;
public SurveyQuestion GetQuestion(int index) => surveyQuestions[index];
```

`AllParticipants` 成员必须考虑 `respondents` 变量可能为 NULL 但返回值不能为 NULL 的情况。 如果通过删除后面的 `??` 和空序列来更改该表达式，则编译器会警告你方法可能返回 `null` 并且其返回签名返回不可为空类型。

最后，在 `Main` 方法的底部添加以下循环：

C#

```
foreach (var participant in surveyRun.AllParticipants)
{
    Console.WriteLine($"Participant: {participant.Id}:");
    if (participant.AnsweredSurvey)
    {
        for (int i = 0; i < surveyRun.Questions.Count; i++)
        {
            var answer = participant.Answer(i);
            Console.WriteLine($"\t{surveyRun.GetQuestion(i).QuestionText} :
{answer}");
        }
    }
    else
    {
        Console.WriteLine("\tNo responses");
    }
}
```

你不需要在此代码中执行任何 `null` 检查，因为你已设计了基础接口，这样它们均返回不可为空引用类型。

获取代码

你可以从 csharp/NullableIntroduction 文件夹中的示例存储库获取已完成教程的代码。

通过更改可为空和不可为空引用类型之间的类型声明进行试验。 了解如何生成不同的警告以确保不会意外取消引用 `null`。

后续步骤

了解如何在使用实体框架时使用可为空引用类型：

Entity Framework Core 基础知识：使用可为空引用类型

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

使用字符串内插构造格式化字符串

项目 • 2023/04/08

本教程介绍了如何使用 C# 字符串内插将值插入单个结果字符串中。 读者可以编写 C# 代码并查看代码编译和运行结果。 本教程包含一系列课程，介绍了如何将值插入字符串，以及用不同方式设置这些值的格式。

本教程要求你有一台可用于开发的计算机。 .NET 教程 Hello World 10 分钟入门 介绍了如何在 Windows、Linux 或 macOS 上设置本地开发环境。 另外，还可在浏览器中完成本教程的交互式版本。

创建内插字符串

创建名为 interpolated 的目录 。 将其设置为当前目录并从控制台窗口运行以下命令：

.NET CLI

```
dotnet new console
```

此命令会在当前目录中创建一个新的 .NET Core 控制台应用程序。

在常用编辑器中，打开 Program.cs，将行 `Console.WriteLine("Hello World!");` 替换为以下代码，并将 `<name>` 替换为你的姓名 ：

C#

```
var name = "<name>";
Console.WriteLine($"Hello, {name}. It's a pleasure to meet you!");
```

通过在控制台窗口键入 `dotnet run` 试运行此代码。 当运行该程序时，它会在问候语中显示一个包含你的姓名的字符串。 WriteLine 方法调用中包含的字符串是一个内插字符串表达式 。 这是一种模板，可让你用包含嵌入代码的字符串构造单个字符串（称为结果字符串 ）。 内插字符串特别适用于将值插入字符串或连接字符串（将字符串联在一起）。

该简单示例包含了每个内插字符串必须具有的两个元素：

- 字符串文本以 `$` 字符开头，后接左双引号字符。 `$` 符号和引号字符之间不能有空格。 （如果希望看到包含空格会发生什么情况，请在 `$` 字符后面插入一个空格并保存该文件，然后在控制台窗口中键入 `dotnet run` 再次运行该程序。C# 编译器显示错误消息“错误 CS1056: 意外的字符 '$'”。）

- 一个或多个内插表达式 。 左大括号和右大括号（`{` 和 `}`）指示内插表达式。 可将任何返回值的 C# 表达式置于大括号内（包括 `null`）。

下面再尝试一些其他数据类型的字符串内插示例。

包含不同的数据类型

上一节使用了字符串内插将一个字符串插入到了另一字符串中。 不过，内插字符串表达式的结果可以是任何数据类型。 下面让我们在内插字符串中添加多种数据类型的值。

在以下示例中，首先定义一个具有 `Name`属性和 `ToString`方法的类数据类型 `Vegetable`，它可以替代Object.ToString() 方法的行为。 public 访问修饰符使该方法可用于任何客户端代码以获取 `Vegetable` 实例的字符串表示形式。 在本示例中，`Vegetable.ToString` 方法返回在 `Vegetable`构造函数处初始化的 `Name` 属性的值：

C#

```
public Vegetable(string name) => Name = name;
```

然后，通过使用 new 运算符并为构造函数 `Vegetable` 提供一个名称来创建名为 `item` 的 `Vegetable` 类的实例：

C#

```
var item = new Vegetable("eggplant");
```

最后，将 `item` 变量添加到同样包含 DateTime 值、Decimal 值和 `Unit`枚举值的内插字符串中。 将编辑器中的所有 C# 代码替换为以下代码，然后使用 `dotnet run` 命令运行：

C#

```
using System;

public class Vegetable
{
   public Vegetable(string name) => Name = name;

   public string Name { get; }

   public override string ToString() => Name;
}

public class Program
{
   public enum Unit { item, kilogram, gram, dozen };
```

```
   public static void Main()
   {
      var item = new Vegetable("eggplant");
      var date = DateTime.Now;
      var price = 1.99m;
      var unit = Unit.item;
      Console.WriteLine($"On {date}, the price of {item} was {price} per
{unit}.");
   }
}
```

注意，内插字符串中的内插表达式 `item` 会解析为结果字符串中的“eggplant”文本。 这是因为，当表达式结果的类型不是字符串时，会按照以下方式将其解析为字符串：

- 如果内插表达式的计算结果为 `null`，则会使用一个空字符串（"" 或 String.Empty）。
- 如果内插表达式的计算结果不是 `null`，通常会调用结果类型的 `ToString` 方法。 可以通过更新 `Vegetable.ToString` 方法的实现来进行测试。 你甚至不用实现 `ToString` 方法，因为每个类型都有一些此方法的实现。 可通过注释掉示例中 `Vegetable.ToString` 方法的定义（在它前面添加注释符号 `//` 即可）来进行测试。 在输出中，字符串“eggplant”被替换为完全限定的类型名称（本示例中为“Vegetable”），这是 Object.ToString() 方法的默认行为。 对于枚举值的 `ToString` 方法，其默认行为是返回该值的字符串表示形式。

在此示例的输出中，日期过于精确（eggplant 的价格不会以秒为单位变化），且价格值没有标明货币单位。 下一节将介绍如何通过控制表达式结果的字符串表示形式来解决这些问题。

控制内插表达式的格式

上一节将两个格式不正确的字符串插入到了结果字符串中。 一个是日期和时间值，只有日期是合适的。 第二个是没有标明货币单位的价格。 这两个问题都很容易解决。 通过字符串内插，可以指定用于控制特定类型格式的格式字符串 。 将前面示例中的调用修改为 `Console.WriteLine`，从而包含日期和价格表达式的格式字符串，如以下行所示：

C#

```
Console.WriteLine($"On {date:d}, the price of {item} was {price:C2} per
{unit}.");
```

可通过在内插表达式后接冒号（“:”）和格式字符串来指定格式字符串。 “d”是标准日期和时间格式字符串，表示短日期格式。 “C2”是标准数值格式字符串，用数字表示货币值

（精确到小数点后两位）。

.NET 库中的许多类型支持一组预定义的格式字符串。 这些格式字符串包括所有数值类型以及日期和时间类型。 有关支持格式字符串的完整类型列表，请参阅 .NET 中的格式化类型文章中的格式字符串和. NET 类库类型。

尝试在文本编辑器中修改格式字符串，并在每次更改时重新运行该程序，查看更改如何影响日期和时间以及数值的格式。 将 `{date:d}` 中的“d”更改为“t”（显示短时间格式）、“y”（显示年份和月份）和“yyyy”（显示四位数年份）。 将 `{price:C2}` 中的“C2”更改为“e”（用于指数计数法）和“F3”（使数值在小数点后保持三位数字）。

除了控制格式之外，还可以控制结果字符串中包含的格式字符串的字段宽度和对齐方式。 下一节会介绍如何执行此操作。

控制内插表达式的字段宽度和对齐方式

通常，当内插表达式的结果格式化为字符串时，结果字符串中会包含该字符串，但没有前导或尾随空格。 特别是对于使用一组数据的情况，控制字段宽度和对齐方式有助于增强输出的可读性。 为此，用下方代码替换文本编辑器中的所有代码，然后键入 `dotnet run` 来执行程序：

C#

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

public class Example
{
   public static void Main()
   {
      var titles = new Dictionary<string, string>()
      {
          ["Doyle, Arthur Conan"] = "Hound of the Baskervilles, The",
          ["London, Jack"] = "Call of the Wild, The",
          ["Shakespeare, William"] = "Tempest, The"
      };

      Console.WriteLine("Author and Title List");
      Console.WriteLine();
      Console.WriteLine($"|{"Author",-25}|{"Title",30}|");
      foreach (var title in titles)
         Console.WriteLine($"|{title.Key,-25}|{title.Value,30}|");
   }
}
```

创建者姓名采用左对齐方式，其所写标题采用右对齐方式。 通过在内插表达式后面添加一个逗号（“,”）并指定“最小” 字段宽度来指定对齐方式。 如果指定的值是正数，则该字

段为右对齐。 如果它为负数，则该字段为左对齐。

尝试删除 `{"Author",-25}` 和 `{title.Key,-25}` 代码中的负号，然后再次运行该示例，如以下代码所示：

C#

```
Console.WriteLine($"|{"Author",25}|{"Title",30}|");
foreach (var title in titles)
   Console.WriteLine($"|{title.Key,25}|{title.Value,30}|");
```

此时，创建者信息为右对齐。

可合并单个内插表达式中的对齐说明符和格式字符串。 为此，请先指定对齐方式，然后是冒号和格式字符串。 将 `Main` 方法中的所有代码替换为以下代码，该代码用定义的字段宽度显示格式化字符串。 然后输入 `dotnet run` 命令来运行程序。

C#

```
Console.WriteLine($"[{DateTime.Now,-20:d}] Hour [{DateTime.Now,-10:HH}]
[{1063.342,15:N2}] feet");
```

输出类似于以下内容：

控制台

```
[04/14/2018          ] Hour [16        ] [       1,063.34] feet
```

你已完成“字符串内插”教程。

有关详细信息，请参阅字符串内插主题和 C# 中的字符串内插教程。

Use pattern matching to build your class behavior for better code

Article • 12/04/2024

The pattern matching features in C# provide syntax to express your algorithms. You can use these techniques to implement the behavior in your classes. You can combine object-oriented class design with a data-oriented implementation to provide concise code while modeling real-world objects.

In this tutorial, you learn how to:

- ✓ Express your object oriented classes using data patterns.
- ✓ Implement those patterns using C#'s pattern matching features.
- ✓ Leverage compiler diagnostics to validate your implementation.

Prerequisites

You need to set up your machine to run .NET. Download Visual Studio 2022 or the .NET SDK.

Build a simulation of a canal lock

In this tutorial, you build a C# class that simulates a canal lock. Briefly, a canal lock is a device that raises and lowers boats as they travel between two stretches of water at different levels. A lock has two gates and some mechanism to change the water level.

In its normal operation, a boat enters one of the gates while the water level in the lock matches the water level on the side the boat enters. Once in the lock, the water level is changed to match the water level where the boat leaves the lock. Once the water level matches that side, the gate on the exit side opens. Safety measures make sure an operator can't create a dangerous situation in the canal. The water level can be changed only when both gates are closed. At most one gate can be open. To open a gate, the water level in the lock must match the water level outside the gate being opened.

You can build a C# class to model this behavior. A `CanalLock` class would support commands to open or close either gate. It would have other commands to raise or lower the water. The class should also support properties to read the current state of both gates and the water level. Your methods implement the safety measures.

Define a class

You build a console application to test your `CanalLock` class. Create a new console project for .NET 5 using either Visual Studio or the .NET CLI. Then, add a new class and name it `CanalLock`. Next, design your public API, but leave the methods not implemented:

C#

```
public enum WaterLevel
{
    Low,
    High
}
public class CanalLock
{
    // Query canal lock state:
    public WaterLevel CanalLockWaterLevel { get; private set; } =
WaterLevel.Low;
    public bool HighWaterGateOpen { get; private set; } = false;
    public bool LowWaterGateOpen { get; private set; } = false;

    // Change the upper gate.
    public void SetHighGate(bool open)
    {
        throw new NotImplementedException();
    }

    // Change the lower gate.
    public void SetLowGate(bool open)
    {
        throw new NotImplementedException();
    }

    // Change water level.
    public void SetWaterLevel(WaterLevel newLevel)
    {
        throw new NotImplementedException();
    }

    public override string ToString() =>
        $"The lower gate is {(LowWaterGateOpen ? "Open" : "Closed")}. " +
        $"The upper gate is {(HighWaterGateOpen ? "Open" : "Closed")}. " +
        $"The water level is {CanalLockWaterLevel}.";
}
```

The preceding code initializes the object so both gates are closed, and the water level is low. Next, write the following test code in your `Main` method to guide you as you create a first implementation of the class:

C#

```
// Create a new canal lock:
var canalGate = new CanalLock();

// State should be doors closed, water level low:
Console.WriteLine(canalGate);

canalGate.SetLowGate(open: true);
Console.WriteLine($"Open the lower gate:  {canalGate}");

Console.WriteLine("Boat enters lock from lower gate");

canalGate.SetLowGate(open: false);
Console.WriteLine($"Close the lower gate:  {canalGate}");

canalGate.SetWaterLevel(WaterLevel.High);
Console.WriteLine($"Raise the water level: {canalGate}");

canalGate.SetHighGate(open: true);
Console.WriteLine($"Open the higher gate:  {canalGate}");

Console.WriteLine("Boat exits lock at upper gate");
Console.WriteLine("Boat enters lock from upper gate");

canalGate.SetHighGate(open: false);
Console.WriteLine($"Close the higher gate: {canalGate}");

canalGate.SetWaterLevel(WaterLevel.Low);
Console.WriteLine($"Lower the water level: {canalGate}");

canalGate.SetLowGate(open: true);
Console.WriteLine($"Open the lower gate:  {canalGate}");

Console.WriteLine("Boat exits lock at upper gate");

canalGate.SetLowGate(open: false);
Console.WriteLine($"Close the lower gate:  {canalGate}");
```

Next, add a first implementation of each method in the `CanalLock` class. The following code implements the methods of the class without concern to the safety rules. You add safety tests later:

C#

```
// Change the upper gate.
public void SetHighGate(bool open)
{
    HighWaterGateOpen = open;
}

// Change the lower gate.
```

```
public void SetLowGate(bool open)
{
    LowWaterGateOpen = open;
}

// Change water level.
public void SetWaterLevel(WaterLevel newLevel)
{
    CanalLockWaterLevel = newLevel;
}
```

The tests you wrote so far pass. You implemented the basics. Now, write a test for the first failure condition. At the end of the previous tests, both gates are closed, and the water level is set to low. Add a test to try opening the upper gate:

C#

```
Console.WriteLine("=========================================");
Console.WriteLine("     Test invalid commands");
// Open "wrong" gate (2 tests)
try
{
    canalGate = new CanalLock();
    canalGate.SetHighGate(open: true);
}
catch (InvalidOperationException)
{
    Console.WriteLine("Invalid operation: Can't open the high gate. Water is
low.");
}
Console.WriteLine($"Try to open upper gate: {canalGate}");
```

This test fails because the gate opens. As a first implementation, you could fix it with the following code:

C#

```
// Change the upper gate.
public void SetHighGate(bool open)
{
    if (open && (CanalLockWaterLevel == WaterLevel.High))
        HighWaterGateOpen = true;
    else if (open && (CanalLockWaterLevel == WaterLevel.Low))
        throw new InvalidOperationException("Cannot open high gate when the
water is low");
}
```

Your tests pass. But, as you add more tests, you add more `if` clauses and test different properties. Soon, these methods get too complicated as you add more conditionals.

Implement the commands with patterns

A better way is to use *patterns* to determine if the object is in a valid state to execute a command. You can express if a command is allowed as a function of three variables: the state of the gate, the level of the water, and the new setting:

Expand table

New setting	Gate state	Water Level	Result
Closed	Closed	High	Closed
Closed	Closed	Low	Closed
Closed	Open	High	Closed
~~Closed~~	~~Open~~	~~Low~~	~~Closed~~
Open	Closed	High	Open
Open	Closed	Low	Closed (Error)
Open	Open	High	Open
~~Open~~	~~Open~~	~~Low~~	~~Closed (Error)~~

The fourth and last rows in the table have strike through text because they're invalid. The code you're adding now should make sure the high water gate is never opened when the water is low. Those states can be coded as a single switch expression (remember that `false` indicates "Closed"):

C#

```
HighWaterGateOpen = (open, HighWaterGateOpen, CanalLockWaterLevel) switch
{
    (false, false, WaterLevel.High) => false,
    (false, false, WaterLevel.Low) => false,
    (false, true, WaterLevel.High) => false,
    (false, true, WaterLevel.Low) => false, // should never happen
    (true, false, WaterLevel.High) => true,
    (true, false, WaterLevel.Low) => throw new
InvalidOperationException("Cannot open high gate when the water is low"),
    (true, true, WaterLevel.High) => true,
    (true, true, WaterLevel.Low) => false, // should never happen
};
```

Try this version. Your tests pass, validating the code. The full table shows the possible combinations of inputs and results. That means you and other developers can quickly

look at the table and see that you covered all the possible inputs. Even easier, the compiler can help as well. After you add the previous code, you can see that the compiler generates a warning: *CS8524* indicates the switch expression doesn't cover all possible inputs. The reason for that warning is that one of the inputs is an `enum` type. The compiler interprets "all possible inputs" as all inputs from the underlying type, typically an `int`. This `switch` expression only checks the values declared in the `enum`. To remove the warning, you can add a catch-all discard pattern for the last arm of the expression. This condition throws an exception, because it indicates invalid input:

```
C#
```

```
_  => throw new InvalidOperationException("Invalid internal state"),
```

The preceding switch arm must be last in your `switch` expression because it matches all inputs. Experiment by moving it earlier in the order. That causes a compiler error *CS8510* for unreachable code in a pattern. The natural structure of switch expressions enables the compiler to generate errors and warnings for possible mistakes. The compiler "safety net" makes it easier for you to create correct code in fewer iterations, and the freedom to combine switch arms with wildcards. The compiler issues errors if your combination results in unreachable arms you didn't expect, and warnings if you remove a needed arm.

The first change is to combine all the arms where the command is to close the gate; that's always allowed. Add the following code as the first arm in your switch expression:

```
C#
```

```
(false, _, _) => false,
```

After you add the previous switch arm, you'll get four compiler errors, one on each of the arms where the command is `false`. Those arms are already covered by the newly added arm. You can safely remove those four lines. You intended this new switch arm to replace those conditions.

Next, you can simplify the four arms where the command is to open the gate. In both cases where the water level is high, the gate can be opened. (In one, it's already open.) One case where the water level is low throws an exception, and the other shouldn't happen. It should be safe to throw the same exception if the water lock is already in an invalid state. You can make the following simplifications for those arms:

```
C#
```

```
(true, _, WaterLevel.High) => true,
(true, false, WaterLevel.Low) => throw new InvalidOperationException("Cannot
open high gate when the water is low"),
_ => throw new InvalidOperationException("Invalid internal state"),
```

Run your tests again, and they pass. Here's the final version of the `SetHighGate` method:

C#

```
// Change the upper gate.
public void SetHighGate(bool open)
{
    HighWaterGateOpen = (open, HighWaterGateOpen, CanalLockWaterLevel)
switch
    {
        (false, _,     _)               => false,
        (true, _,      WaterLevel.High) => true,
        (true, false, WaterLevel.Low)   => throw new
InvalidOperationException("Cannot open high gate when the water is low"),
        _                               => throw new
InvalidOperationException("Invalid internal state"),
    };
}
```

Implement patterns yourself

Now that you've seen the technique, fill in the `SetLowGate` and `SetWaterLevel` methods yourself. Start by adding the following code to test invalid operations on those methods:

C#

```
Console.WriteLine();
Console.WriteLine();
try
{
    canalGate = new CanalLock();
    canalGate.SetWaterLevel(WaterLevel.High);
    canalGate.SetLowGate(open: true);
}
catch (InvalidOperationException)
{
    Console.WriteLine("invalid operation: Can't open the lower gate. Water
is high.");
}
Console.WriteLine($"Try to open lower gate: {canalGate}");
// change water level with gate open (2 tests)
Console.WriteLine();
Console.WriteLine();
try
```

```
{
    canalGate = new CanalLock();
    canalGate.SetLowGate(open: true);
    canalGate.SetWaterLevel(WaterLevel.High);
}
catch (InvalidOperationException)
{
    Console.WriteLine("invalid operation: Can't raise water when the lower
gate is open.");
}
Console.WriteLine($"Try to raise water with lower gate open: {canalGate}");
Console.WriteLine();
Console.WriteLine();
try
{
    canalGate = new CanalLock();
    canalGate.SetWaterLevel(WaterLevel.High);
    canalGate.SetHighGate(open: true);
    canalGate.SetWaterLevel(WaterLevel.Low);
}
catch (InvalidOperationException)
{
    Console.WriteLine("invalid operation: Can't lower water when the high
gate is open.");
}
Console.WriteLine($"Try to lower water with high gate open: {canalGate}");
```

Run your application again. You can see the new tests fail, and the canal lock gets into an invalid state. Try to implement the remaining methods yourself. The method to set the lower gate should be similar to the method to set the upper gate. The method that changes the water level has different checks, but should follow a similar structure. You might find it helpful to use the same process for the method that sets the water level. Start with all four inputs: The state of both gates, the current state of the water level, and the requested new water level. The switch expression should start with:

C#

```
CanalLockWaterLevel = (newLevel, CanalLockWaterLevel, LowWaterGateOpen,
HighWaterGateOpen) switch
{
    // elided
};
```

You have 16 total switch arms to fill in. Then, test and simplify.

Did you make methods something like this?

C#

```
// Change the lower gate.
public void SetLowGate(bool open)
{
    LowWaterGateOpen = (open, LowWaterGateOpen, CanalLockWaterLevel) switch
    {
        (false, _, _) => false,
        (true, _, WaterLevel.Low) => true,
        (true, false, WaterLevel.High) => throw new
InvalidOperationException("Cannot open low gate when the water is high"),
        _ => throw new InvalidOperationException("Invalid internal state"),
    };
}

// Change water level.
public void SetWaterLevel(WaterLevel newLevel)
{
    CanalLockWaterLevel = (newLevel, CanalLockWaterLevel, LowWaterGateOpen,
HighWaterGateOpen) switch
    {
        (WaterLevel.Low, WaterLevel.Low, true, false) => WaterLevel.Low,
        (WaterLevel.High, WaterLevel.High, false, true) => WaterLevel.High,
        (WaterLevel.Low, _, false, false) => WaterLevel.Low,
        (WaterLevel.High, _, false, false) => WaterLevel.High,
        (WaterLevel.Low, WaterLevel.High, false, true) => throw new
InvalidOperationException("Cannot lower water when the high gate is open"),
        (WaterLevel.High, WaterLevel.Low, true, false) => throw new
InvalidOperationException("Cannot raise water when the low gate is open"),
        _ => throw new InvalidOperationException("Invalid internal state"),
    };
}
```

Your tests should pass, and the canal lock should operate safely.

Summary

In this tutorial, you learned to use pattern matching to check the internal state of an object before applying any changes to that state. You can check combinations of properties. Once you built tables for any of those transitions, you test your code, then simplify for readability and maintainability. These initial refactorings might suggest further refactorings that validate internal state or manage other API changes. This tutorial combined classes and objects with a more data-oriented, pattern-based approach to implement those classes.

C# 中的字符串内插

项目 • 2023/09/02

本教程演示如何使用字符串插值设置表达式结果的格式并将其包含仅结果字符串中。 以下示例假设阅读者熟悉基础 C# 概念和 .NET 类型格式设置。 如果不熟悉字符串插值或 .NET 类型格式设置，请先参阅交互式字符串内插教程。 若要详细了解如何在 .NET 中设置类型的格式，请参阅设置 .NET 中类型的格式。

介绍

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。 可嵌入任何会在内插字符串中返回值的有效 C# 表达式。 在以下示例中，对某个表达式执行计算后，其结果立即转换为一个字符串并包含到结果字符串中：

C#

```
double a = 3;
double b = 4;
Console.WriteLine($"Area of the right triangle with legs of {a} and {b} is
{0.5 * a * b}");
Console.WriteLine($"Length of the hypotenuse of the right triangle with legs
of {a} and {b} is {CalculateHypotenuse(a, b)}");
double CalculateHypotenuse(double leg1, double leg2) => Math.Sqrt(leg1 *
leg1 + leg2 * leg2);
// Output:
// Area of the right triangle with legs of 3 and 4 is 6
// Length of the hypotenuse of the right triangle with legs of 3 and 4 is 5
```

如示例所示，通过将表达式用大括号括起来，可将表达式包含到内插字符串中：

C#

```
{<interpolationExpression>}
```

内插字符串支持字符串复合格式设置功能的所有功能。 这使得它们成为 String.Format 方法的更具可读性的替代选项。

如何为内插表达式指定格式字符串

若要指定受表达式结果类型支持的格式字符串，请在内插表达式后面添加冒号（":"）和格式字符串：

C#

```
{<interpolationExpression>:<formatString>}
```

以下示例演示如何为生成日期和时间或数值结果的表达式指定标准和自定义格式字符串：

C#

```
var date = new DateTime(1731, 11, 25);
Console.WriteLine($"On {date:dddd, MMMM dd, yyyy} L. Euler introduced the
letter e to denote {Math.E:F5}.");
// Output:
// On Sunday, November 25, 1731 L. Euler introduced the letter e to denote
2.71828.
```

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。

如何控制设置了格式的内插表达式的字段宽度和对齐方式

若要指定设置了格式的表达式结果的最小字段宽度和对齐方式，请在内插表达式后添加逗号（“,”）和常数表达式：

C#

```
{<interpolationExpression>,<alignment>}
```

如果对齐方式值为正，则设置了格式的表达式结果为右对齐，如果为负，则为左对齐。

如果需要同时指定对齐方式和格式字符串，则先从对齐方式组件开始：

C#

```
{<interpolationExpression>,<alignment>:<formatString>}
```

以下示例演示如何指定对齐方式并使用管道字符 ("|") 分隔文本字段：

C#

```
const int NameAlignment = -9;
const int ValueAlignment = 7;
double a = 3;
double b = 4;
Console.WriteLine($"Three classical Pythagorean means of {a} and {b}:");
Console.WriteLine($"|{"Arithmetic",NameAlignment}|{0.5 * (a +
```

```
b),ValueAlignment:F3}|");
Console.WriteLine($"|{"Geometric",NameAlignment}|{Math.Sqrt(a *
b),ValueAlignment:F3}|");
Console.WriteLine($"|{"Harmonic",NameAlignment}|{2 / (1 / a + 1 /
b),ValueAlignment:F3}|");
// Output:
// Three classical Pythagorean means of 3 and 4:
// |Arithmetic|  3.500|
// |Geometric|  3.464|
// |Harmonic |  3.429|
```

如示例输出所示，如果已设置格式的表达式结果长度超出指定字段宽度，则忽略对齐方式值。

有关详细信息，请参阅复合格式设置一文的对齐组件部分。

如何在内插字符串中使用转义序列

内插字符串支持所有可在普通字符串文本中使用的转义序列。 有关详细信息，请参阅字符串转义序列。

若要逐字解释转义序列，可使用逐字字符串文本。 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。

若要在结果字符串中包含大括号 "{" 或 "}"，请使用两个大括号 "{{" 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。

以下示例演示如何在结果字符串中包含大括号并构造逐字内插字符串：

C#

```
var xs = new int[] { 1, 2, 7, 9 };
var ys = new int[] { 7, 9, 12 };
Console.WriteLine($"Find the intersection of the {{{string.Join(", ",xs)}}}
and {{{string.Join(", ",ys)}}} sets.");
// Output:
// Find the intersection of the {1, 2, 7, 9} and {7, 9, 12} sets.

var userName = "Jane";
var stringWithEscapes = $"C:\\Users\\{userName}\\Documents";
var verbatimInterpolated = $@"C:\Users\{userName}\Documents";
Console.WriteLine(stringWithEscapes);
Console.WriteLine(verbatimInterpolated);
// Output:
// C:\Users\Jane\Documents
// C:\Users\Jane\Documents
```

从 C# 11 开始，可以使用内插的原始字符串文本。

如何在内插表达式中使用三元条件运算符 `?:`

因为冒号 (:) 在具有内插表达式的项中具有特殊含义，为了在表达式中使用条件运算符，请将表达式放在括号内，如下例所示：

C#

```
var rand = new Random();
for (int i = 0; i < 7; i++)
{
    Console.WriteLine($"Coin flip: {(rand.NextDouble() < 0.5 ? "heads" :
"tails")}");
}
```

如何使用字符串插值创建区域性特定的结果字符串

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。

从 .NET 6 开始，可以使用 String.Create(IFormatProvider, DefaultInterpolatedStringHandler) 方法将内插字符串解析为特定于区域性的结果字符串，如以下示例所示：

C#

```
var cultures = new System.Globalization.CultureInfo[]
{
    System.Globalization.CultureInfo.GetCultureInfo("en-US"),
    System.Globalization.CultureInfo.GetCultureInfo("en-GB"),
    System.Globalization.CultureInfo.GetCultureInfo("nl-NL"),
    System.Globalization.CultureInfo.InvariantCulture
};
var date = DateTime.Now;
var number = 31_415_926.536;
foreach (var culture in cultures)
{
    var cultureSpecificMessage = string.Create(culture, $"{date,23}
{number,20:N3}");
    Console.WriteLine($"{culture.Name,-10}{cultureSpecificMessage}");
}
// Output is similar to:
// en-US       8/27/2023 12:35:31 PM      31,415,926.536
// en-GB         27/08/2023 12:35:31      31,415,926.536
// nl-NL         27-08-2023 12:35:31      31.415.926,536
//               08/27/2023 12:35:31      31,415,926.536
```

在更早的 .NET 版本中，请使用从内插的字符串到 System.FormattableString 实例的隐式转换，并调用它的 ToString(IFormatProvider) 方法来创建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#

```
var cultures = new System.Globalization.CultureInfo[]
{
    System.Globalization.CultureInfo.GetCultureInfo("en-US"),
    System.Globalization.CultureInfo.GetCultureInfo("en-GB"),
    System.Globalization.CultureInfo.GetCultureInfo("nl-NL"),
    System.Globalization.CultureInfo.InvariantCulture
};
var date = DateTime.Now;
var number = 31_415_926.536;
FormattableString message = $"{date,23}{number,20:N3}";
foreach (var culture in cultures)
{
    var cultureSpecificMessage = message.ToString(culture);
    Console.WriteLine($"{culture.Name,-10}{cultureSpecificMessage}");
}
// Output is similar to:
// en-US       8/27/2023 12:35:31 PM      31,415,926.536
// en-GB         27/08/2023 12:35:31      31,415,926.536
// nl-NL         27-08-2023 12:35:31      31.415.926,536
//               08/27/2023 12:35:31      31,415,926.536
```

如示例所示，可使用某个 FormattableString 实例为各种区域性生成多个结果字符串。

如何使用固定区域性创建结果字符串

从 .NET 6 开始，请使用 String.Create(IFormatProvider, DefaultInterpolatedStringHandler) 方法将内插字符串解析为 InvariantCulture 的结果字符串，如以下示例所示：

C#

```
string message = string.Create(CultureInfo.InvariantCulture, $"Date and time
in invariant culture: {DateTime.Now}");
Console.WriteLine(message);
// Output is similar to:
// Date and time in invariant culture: 05/17/2018 15:46:24
```

在更早版本的 .NET 以及 FormattableString.ToString(IFormatProvider) 方法中，可以使用静态 FormattableString.Invariant 方法，如以下示例所示：

C#

```
string message = FormattableString.Invariant($"Date and time in invariant
culture: {DateTime.Now}");
Console.WriteLine(message);
// Output is similar to:
// Date and time in invariant culture: 05/17/2018 15:46:24
```

结论

本教程介绍字符串插值用法的常见方案。 有关字符串内插的详细信息，请参阅字符串内插。 若要详细了解如何在 .NET 中设置类型的格式，请参阅这两篇文章：设置 .NET 中类型的格式和复合格式设置。

另请参阅

- String.Format
- System.FormattableString
- System.IFormattable
- 字符串

控制台应用

项目 • 2023/03/14

本教程将介绍 .NET 和 C# 语言的许多功能。 学习内容：

- .NET CLI 的基础知识
- C# 控制台应用程序的结构
- 控制台 I/O
- .NET 中文件 I/O API 的基础知识
- .NET 中基于任务的异步编程基础知识

你将生成一个应用程序，用于读取文本文件，然后将文本文件的内容回显到控制台。 按配速大声朗读控制台输出。 可以按“<”（小于）或“>”（大于）键加速或减速显示。 可以在 Windows、Linux、macOS 或 Docker 容器中运行此应用程序。

此教程将介绍许多功能。 我们将逐个生成这些功能。

先决条件

- .NET 6 SDK.
- 代码编辑器。

创建应用

第一步是新建应用程序。 打开命令提示符，然后新建应用程序的目录。 将新建的目录设为当前目录。 在命令提示符处，键入命令 `dotnet new console`。 这将为基本的“Hello World”应用程序创建起始文件。

在开始进行修改之前，我们先运行一个简单的 Hello World 应用程序。 创建应用程序之后，在命令提示符处键入 `dotnet run`。 此命令运行 NuGet 包还原过程，创建应用程序可执行文件，并运行该可执行文件。

简单的 Hello World 应用程序代码全都在 Program.cs 中。 使用常用文本编辑器打开此文件。 将 Program.cs 中的代码替换为以下代码：

C#

```
namespace TeleprompterConsole;

internal class Program
{
    static void Main(string[] args)
    {
```

```
        Console.WriteLine("Hello World!");
    }
}
```

该文件顶部将出现 `namespace` 语句。 与你可能用过的其他面向对象的语言一样，C# 也使用命名空间来整理类型。 此 Hello World 程序也一样。 你可以看到，该程序位于名为 `TeleprompterConsole` 的命名空间中。

读取和回显文件

要添加的第一项功能是读取文本文件，然后在控制台中显示全部文本。 首先，让我们来添加文本文件。 将此示例 的 GitHub 存储库中的 sampleQuotes.txt 文件复制到项目目录中。 这将用作应用程序脚本。 有关如何下载本教程示例应用的信息，请参阅示例和教程中的说明。

接下来，在 `Program` 类中添加以下方法（即 `Main` 方法的下方）：

C#

```
static IEnumerable<string> ReadFrom(string file)
{
    string? line;
    using (var reader = File.OpenText(file))
    {
        while ((line = reader.ReadLine()) != null)
        {
            yield return line;
        }
    }
}
```

这是一种称为“iterator 方法”的特殊类型 C# 方法。 迭代器方法返回延迟计算的序列。 也就是说，序列中的每一项是在使用序列的代码提出请求时生成。 迭代器方法包含一个或多个 yield return 语句。 `ReadFrom` 方法返回的对象包含用于生成序列中所有项的代码。 在此示例中，这涉及读取源文件中的下一行文本，然后返回相应的字符串。 每当调用代码请求生成序列中的下一项时，代码就会读取并返回文件中的下一行文本。 读取完整个文件时，序列会指示没有其他项。

有两个 C# 语法元素你可能是刚开始接触。 此方法中的 using 语句用于管理资源清除。 `using` 语句中初始化的变量（在此示例中，为 `reader`）必须实现 IDisposable 接口。 该接口定义一个方法（`Dispose`），应在释放资源时调用此方法。 当快执行到 `using` 语句的右大括号时，编译器会生成此调用。 编译器生成的代码可确保资源得到释放，即使代码块中用 using 语句定义的代码抛出异常，也不例外。

`reader` 变量是使用 `var` 关键字进行定义。 var 定义的是*隐式类型局部变量*。 也就是说，变量的类型是由分配给变量的对象的编译时类型决定的。 此处，它为 OpenText(String) 方法的返回值，即 StreamReader 对象。

现在，让我们在 `Main` 方法中填充用于读取文件的代码：

C#

```
var lines = ReadFrom("sampleQuotes.txt");
foreach (var line in lines)
{
    Console.WriteLine(line);
}
```

使用 `dotnet run` 运行程序，可以看到控制台中打印输出所有文本行。

添加延迟和设置输出格式

现在的问题是，输出显示过快，无法大声朗读。 此时，需要为输出添加延迟。 首先，将生成一些可实现异步处理的核心代码。 不过，在执行这些初始步骤时，将遵循一些反面模式。 反面模式会在你添加代码时在注释中指出，代码将在后面的步骤中进行更新。

这部分包含两步操作。 首先，将迭代器方法更新为返回单个字词，而不是整行文本。 为此，执行下面这些修改。 用以下代码替换 `yield return line;` 语句：

C#

```
var words = line.Split(' ');
foreach (var word in words)
{
    yield return word + " ";
}
yield return Environment.NewLine;
```

接下来，需要修改对文件行的使用方式，并在写入每个字词后添加延迟。 用以下代码块替换 `Main` 方法中的 `Console.WriteLine(line)` 的语句：

C#

```
Console.Write(line);
if (!string.IsNullOrWhiteSpace(line))
{
    var pause = Task.Delay(200);
    // Synchronously waiting on a task is an
    // anti-pattern. This will get fixed in later
    // steps.
```

```
    pause.Wait();
}
```

运行此示例并检查输出。 现在，每打印输出一个字词后，就会有 200 毫秒的延迟。 不过，显示的输出反映出一些问题，因为源文本文件有好几行都超过 80 个字符，且没有换行符。 很难滚动读取这些文本。 此问题很容易解决。 只需跟踪每行长度，然后在行长度达到特定阈值时生成新的一行即可。 在 `ReadFrom` 方法中声明 `words` 后声明一个局部变量，用于保存行长度：

C#

```
var lineLength = 0;
```

然后，在 `yield return word + " ";` 语句后（在右大括号前）添加以下代码：

C#

```
lineLength += word.Length + 1;
if (lineLength > 70)
{
    yield return Environment.NewLine;
    lineLength = 0;
}
```

运行此示例，将能够按预配速大声朗读文本。

异步任务

最后一步将是添加代码，以便在一个任务中异步编写输出，同时运行另一任务来读取用户输入（如果用户想要加快或减慢文本显示速度，或完全停止文本显示的话）。 此过程分为几步操作，最后将完成所需的全部更新。 第一步是创建异步 Task 返回方法，用于表示已创建的用于读取和显示文件的代码。

将以下方法（截取自 `Main` 方法主体）添加到 `Program` 类中：

C#

```
private static async Task ShowTeleprompter()
{
    var words = ReadFrom("sampleQuotes.txt");
    foreach (var word in words)
    {
        Console.Write(word);
        if (!string.IsNullOrWhiteSpace(word))
        {
```

```
            await Task.Delay(200);
        }
    }
}
```

你会注意到两处更改。 首先，此版本在方法主体中使用 `await` 关键字，而不是调用 Wait() 同步等待任务完成。 为此，需要将 `async` 修饰符添加到方法签名中。 此方法返回 `Task`。 请注意，没有用于返回 `Task` 对象的返回语句。 相反，`Task` 对象由编译器在你使用 `await` 运算符时生成的代码进行创建。 可以想象，此方法在到达 `await` 时返回。 返回的 `Task` 指示工作未完成。 在等待的任务完成时，此方法继续执行。 执行完后，返回的 `Task` 会指示已完成。 调用代码可以通过监视返回的 `Task` 来确定完成时间。

在调用 `ShowTeleprompter` 之前添加 `await` 关键字：

C#

```
await ShowTeleprompter();
```

这要求将 `Main` 方法签名更改为：

C#

```
static async Task Main(string[] args)
```

在基础知识部分详细了解 async Main 方法。

接下来，需要编写第二个异步方法，从控制台读取键，并监视“<”（小于）、“>”（大于）和“X”或“x”键。 下面是为此任务添加的方法：

C#

```
private static async Task GetInput()
{
    var delay = 200;
    Action work = () =>
    {
        do {
            var key = Console.ReadKey(true);
            if (key.KeyChar == '>')
            {
                delay -= 10;
            }
            else if (key.KeyChar == '<')
            {
                delay += 10;
            }
            else if (key.KeyChar == 'X' || key.KeyChar == 'x')
```

```
            {
                break;
            }
        } while (true);
    };
    await Task.Run(work);
}
```

这创建了一个表示 Action 委托的 lambda 表达式，用于在用户按“<”（小于）或“>”（大于）键时，从控制台读取键，并修改表示延迟的局部变量。 当用户按下“X”或“x”键时，委托方法结束，允许用户随时停止文本显示。 此方法使用 ReadKey() 来阻止并等待用户按键。

若要完成这项功能，需要新建 `async Task` 返回方法，用于启动这两项任务（`GetInput` 和 `ShowTeleprompter`），并管理这两项任务之间共享的数据。

是时候创建一个类来处理这两项任务之间共享的数据了。 此类包含两个公共属性，即延迟和指示已读取完整个文件的标志 `Done`：

C#

```
namespace TeleprompterConsole;

internal class TelePrompterConfig
{
    public int DelayInMilliseconds { get; private set; } = 200;
    public void UpdateDelay(int increment) // negative to speed up
    {
        var newDelay = Min(DelayInMilliseconds + increment, 1000);
        newDelay = Max(newDelay, 20);
        DelayInMilliseconds = newDelay;
    }
    public bool Done { get; private set; }
    public void SetDone()
    {
        Done = true;
    }
}
```

将该类放入一个新文件中，并将该类包含在 `TeleprompterConsole` 命名空间中，如下所示。 还需在文件顶部添加 `using static` 语句，以便可以引用 `Min` 和 `Max` 方法，而无需使用封闭类或命名空间名称。 using static 语句从一个类导入方法。 该语句不同于没有 `static` 的 `using` 语句，后者从命名空间导入所有类。

C#

```
using static System.Math;
```

接下来，需要将 `ShowTeleprompter` 和 `GetInput` 方法更新为使用新的 `config` 对象。 编写最后一个 `Task` 返回 `async` 方法，用于启动这两项任务，并在第一项任务完成时退出：

C#

```
private static async Task RunTeleprompter()
{
    var config = new TelePrompterConfig();
    var displayTask = ShowTeleprompter(config);

    var speedTask = GetInput(config);
    await Task.WhenAny(displayTask, speedTask);
}
```

此处的一种新方法是 WhenAny(Task[]) 调用。 这会创建 `Task`，只要自变量列表中的任意一项任务完成，它就会完成。

接下来，需要同时将 `ShowTeleprompter` 和 `GetInput` 方法更新为对延迟使用 `config` 对象：

C#

```
private static async Task ShowTeleprompter(TelePrompterConfig config)
{
    var words = ReadFrom("sampleQuotes.txt");
    foreach (var word in words)
    {
        Console.Write(word);
        if (!string.IsNullOrWhiteSpace(word))
        {
            await Task.Delay(config.DelayInMilliseconds);
        }
    }
    config.SetDone();
}

private static async Task GetInput(TelePrompterConfig config)
{
    Action work = () =>
    {
        do {
            var key = Console.ReadKey(true);
            if (key.KeyChar == '>')
                config.UpdateDelay(-10);
            else if (key.KeyChar == '<')
                config.UpdateDelay(10);
            else if (key.KeyChar == 'X' || key.KeyChar == 'x')
                config.SetDone();
        } while (!config.Done);
    };
```

```
    await Task.Run(work);
}
```

这个新版 `ShowTeleprompter` 在 `TelepromptConfig` 类中调用新方法。 现在，需要将 `Main` 更新为调用 `RunTeleprompter`（而不是 `ShowTeleprompter`）：

```C#
await RunTeleprompter();
```

结束语

此教程介绍了与处理控制台应用程序相关的许多 C# 语言和 .NET Core 库功能。 可以在此教程的基础上进一步探索语言和本文介绍的类。 你已了解文件和控制台 I/O 的基础知识、基于任务的异步编程的阻止性和非阻止性用途、C# 语言介绍、C# 程序的组织结构，以及 .NET CLI。

有关文件 I/O 的详细信息，请参阅文件和流 I/O。 有关本教程中使用的异步编程模型的详细信息，请参阅基于任务的异步编程和异步编程。

教程：在 .NET 控制台应用程序使用 C# 发出 HTTP 请求

项目 • 2023/05/10

本教程将生成应用，用于向 GitHub 上的 REST 服务发出 HTTP 请求。 该应用读取 JSON 格式的信息并将 JSON 转换为 C# 对象。 JSON 转换为 C# 对象称为“反序列化”。

该教程演示如何：

- ✓ 发送 HTTP 请求。
- ✓ 反序列化 JSON 响应。
- ✓ 配置具有特性的反序列化。

如果想要按照本教程的最终示例操作，你可以下载它。 有关下载说明，请参阅示例和教程。

先决条件

- .NET SDK 6.0 或更高版本
- 代码编辑器如 [Visual Studio Code（开源、跨平台编辑器）。 可以在 Windows、Linux、macOS 或 Docker 容器中运行此示例应用。

创建客户端应用

1. 打开命令提示符并为应用新建目录。 将新建的目录设为当前目录。
2. 在控制台窗口中输入以下命令：

   .NET CLI

   ```
   dotnet new console --name WebAPIClient
   ```

   该命令将为“Hello World”基本应用创建入门文件。 项目名称为“WebAPIClient”。
3. 导航到“WebAPIClient”目录并运行应用。

   .NET CLI

   ```
   cd WebAPIClient
   ```

   .NET CLI

```
dotnet run
```

dotnet run 自动运行 dotnet restore 还原应用需要的依赖项。 还会按需运行 dotnet build。 你应该会看到应用输出 `"Hello, World!"`。 在终端中，按 `Ctrl`+`C` 可停止应用。

发出 HTTP 请求

此应用调用 GitHub API 以获取 .NET Foundation 伞下的项目相关信息。 终结点为 https://api.github.com/orgs/dotnet/repos。 若要检索信息，它会发出 HTTP GET 请求。 此外，浏览器也发出 HTTP GET 请求，以便你可以将相应的 URL 粘贴到浏览器地址栏，查看将要接收和处理的信息。

使用 HttpClient 类发出 HTTP 请求。 HttpClient 仅支持其长时间运行 API 的异步方法。 因此，采取下列步骤创建异步方法，并从 Main 方法中调用该方法。

1. 在项目目录中打开 `Program.cs` 文件，并将其内容替换为以下内容：

```C#
await ProcessRepositoriesAsync();

static async Task ProcessRepositoriesAsync(HttpClient client)
{
}
```

此代码：

- 将 `Console.WriteLine` 语句替换为调用使用 `await` 关键字的 `ProcessRepositoriesAsync`。
- 定义空 `ProcessRepositoriesAsync` 方法。

2. 在 `Program` 类中，使用 HttpClient 将内容替换为以下 C# 来处理请求和响应。

```C#
using System.Net.Http.Headers;

using HttpClient client = new();
client.DefaultRequestHeaders.Accept.Clear();
client.DefaultRequestHeaders.Accept.Add(
    new
MediaTypeWithQualityHeaderValue("application/vnd.github.v3+json"));
client.DefaultRequestHeaders.Add("User-Agent", ".NET Foundation
Repository Reporter");
```

```
await ProcessRepositoriesAsync(client);

static async Task ProcessRepositoriesAsync(HttpClient client)
{
}
```

此代码：

- 为所有请求设置 HTTP 标头：
  - 接受 JSON 响应的 Accept 标头
  - 一个 User-Agent 标头。 标头均由 GitHub 服务器代码进行检查，需使用标头检索 GitHub 中的信息。

3. 在 `ProcessRepositoriesAsync` 方法中调用 GitHub 终结点，该终结点返回 .NET foundation 组织下的所有存储库列表：

C#

```
static async Task ProcessRepositoriesAsync(HttpClient client)
{
    var json = await client.GetStringAsync(
        "https://api.github.com/orgs/dotnet/repos");

    Console.Write(json);
}
```

此代码：

- 等待从调用 HttpClient.GetStringAsync(String) 方法返回的任务。 此方法将 HTTP GET 请求发送到指定的 URI。 响应正文以 String 形式返回，任务结束时可用。
- 响应字符串 `json` 将输出到控制台。

4. 生成并运行应用。

.NET CLI

```
dotnet run
```

因为 `ProcessRepositoriesAsync` 目前含有一个 `await` 运算符，所以没有生成警告。 输出 JSON 文本的长显示。

反序列化 JSON 结果

以下步骤将 JSON 响应转换为 C# 对象。 使用 System.Text.Json.JsonSerializer 类将 JSON 反序列化为对象。

1. 创建名为“Repository.cs”的文件并添加以下代码：

   ```C#
   public record class Repository(string name);
   ```

   先前的代码定义了一个类，用于表示从 GitHub API 返回的 JSON 对象。 你将使用此类来显示存储库名称列表。

   存储库对象的 JSON 包含数十个属性，但仅对 `name` 属性进行反序列化。 序列化程序自动忽略目标类中没有匹配项的 JSON 属性。 借助此功能，可更轻松地创建仅适用于 JSON 数据包中一个子集字段的类型。

   C# 约定是属性名称首字母大写，但此处的 `name` 属性首字母小写，因为这与 JSON 中的内容完全匹配。 稍后你将了解如何使用与 JSON 属性名称不匹配的 C# 属性名称。

2. 使用序列化程序将 JSON 转换成 C# 对象。 使用以下行替换 `ProcessRepositoriesAsync` 方法中 GetStringAsync(String) 的调用：

   ```C#
   await using Stream stream =
       await
   client.GetStreamAsync("https://api.github.com/orgs/dotnet/repos");
   var repositories =
       await JsonSerializer.DeserializeAsync<List<Repository>>(stream);
   ```

   更新的代码将 GetStringAsync(String) 替换为 GetStreamAsync(String)。 序列化程序方法使用流代替字符串作为其源。

   JsonSerializer.DeserializeAsync<TValue>(Stream, JsonSerializerOptions, CancellationToken) 的第一个自变量是 `await` 表达式。 尽管到目前为止，你只在赋值语句中见过，但 `await` 表达式可以出现在代码中的几乎任何位置。 其他两个参数 `JsonSerializerOptions` 与 `CancellationToken` 均可选，并在代码片段中省略。

   `DeserializeAsync` 方法为泛型，这意味着必须为应为从 JSON 文本创建的对象类型提供类型参数。 在此示例中，你要反序列化到 `List<Repository>`，即另一个泛型对象 System.Collections.Generic.List<T>。 `List<T>` 类存储对象的集合。 类型参数声明存储在 `List<T>` 中的对象的类型。 类型参数是 `Repository` 记录，因为 JSON 文本表示存储库对象的集合。

3. 添加代码以显示每个存储库的名称。 将以下代码行：

```
C#

Console.Write(json);
```

使用以下代码：

```
C#

foreach (var repo in repositories ?? Enumerable.Empty<Repository>())
    Console.Write(repo.name);
```

4. 文件顶部应存在以下 `using` 指令：

```
C#

using System.Net.Http.Headers;
using System.Text.Json;
```

5. 运行应用。

```
.NET CLI

dotnet run
```

输出是 .NET Foundation 中的存储库名称列表。

配置反序列化

1. 在 Repository.cs 中，将文件内容替换为以下 C#。

```
C#

using System.Text.Json.Serialization;

public record class Repository(
    [property: JsonPropertyName("name")] string Name);
```

此代码：

- 将 `name` 属性的名称更改为 `Name`。
- 添加 JsonPropertyNameAttribute 以指定此属性在 JSON 中的显示方式。

2. 在“Program.cs”中，更新代码以使用首字母大写的 `Name` 属性：

```
C#

foreach (var repo in repositories)
   Console.Write(repo.Name);
```

3. 运行应用。

   输出相同。

重构代码

`ProcessRepositoriesAsync` 方法可以执行异步工作，并返回一组存储库。 更改该方法以返回 `Task<List<Repository>>`，并将写入到控制台的代码移动到其调用方附近的控制台。

1. 更改 `ProcessRepositoriesAsync` 的签名，以返回可生成 `Repository` 对象列表的任务：

```
C#

static async Task<List<Repository>> ProcessRepositoriesAsync()
```

2. 处理 JSON 响应后返回存储库：

```
C#

await using Stream stream =
    await
client.GetStreamAsync("https://api.github.com/orgs/dotnet/repos");
var repositories =
    await JsonSerializer.DeserializeAsync<List<Repository>>(stream);
return repositories ?? new();
```

   编译器生成返回值的 `Task<T>` 对象，因为你已将此方法标记为 `async`。

3. 修改 Program.cs 文件，将对 `ProcessRepositoriesAsync` 的调用替换为以下内容以捕获结果并将每个存储库名称写入控制台。

```
C#

var repositories = await ProcessRepositoriesAsync(client);

foreach (var repo in repositories)
    Console.Write(repo.Name);
```

4. 运行应用。

   输出相同。

反序列化更多属性

以下步骤添加代码来处理收到的 JSON 数据包中的多个属性。 你可能不希望处理每个属性，但却希望另外添加一些属性演示 C# 的其他功能。

1. 将 `Repository` 类的内容替换为以下 `record` 订阅：

   C#

   ```
   using System.Text.Json.Serialization;

   public record class Repository(
       [property: JsonPropertyName("name")] string Name,
       [property: JsonPropertyName("description")] string Description,
       [property: JsonPropertyName("html_url")] Uri GitHubHomeUrl,
       [property: JsonPropertyName("homepage")] Uri Homepage,
       [property: JsonPropertyName("watchers")] int Watchers);
   ```

   Uri 和 `int` 类型具有转换字符串表示形式的内置功能。 无需额外代码即可从 JSON 字符串格式反序列化为这些目标类型。 如果 JSON 数据包包含不会转换为目标类型的数据，则序列化操作将引发异常。

2. 在 Program.cs 文件中更新 `foreach` 循环以显示属性值：

   C#

   ```
   foreach (var repo in repositories)
   {
       Console.WriteLine($"Name: {repo.Name}");
       Console.WriteLine($"Homepage: {repo.Homepage}");
       Console.WriteLine($"GitHub: {repo.GitHubHomeUrl}");
       Console.WriteLine($"Description: {repo.Description}");
       Console.WriteLine($"Watchers: {repo.Watchers:#,0}");
       Console.WriteLine();
   }
   ```

3. 运行应用。

   现在列表包含其他属性。

添加日期属性

在 JSON 响应中以此方式设置上次推送操作的日期格式：

```
JSON

2016-02-08T21:27:00Z
```

此格式适用于协调世界时 (UTC)，因此反序列化的结果是 DateTime 值，其 Kind 属性为 Utc。

若要获取你所在时区表示的日期和时间，必须写入自定义转换方法。

1. 在“Repository.cs”中添加日期和时间的 UTC 表示形式的属性和只读 `LastPush` 属性（该属性返回转换为当地时间的日期），文件应如下所示：

   ```
   C#

   using System.Text.Json.Serialization;

   public record class Repository(
       [property: JsonPropertyName("name")] string Name,
       [property: JsonPropertyName("description")] string Description,
       [property: JsonPropertyName("html_url")] Uri GitHubHomeUrl,
       [property: JsonPropertyName("homepage")] Uri Homepage,
       [property: JsonPropertyName("watchers")] int Watchers,
       [property: JsonPropertyName("pushed_at")] DateTime LastPushUtc)
   {
       public DateTime LastPush => LastPushUtc.ToLocalTime();
   }
   ```

   `LastPush` 属性使用 `get` 访问器的 expression-bodied member 进行定义。 不存在 `set` 访问器。 通过省略 `set` 访问器，可采用 C# 语言定义“只读”属性。 （是的，可以在 C# 中创建*只写*属性，但属性值受限。）

2. 在“Program.cs”中再次添加另一个输出语句：

   ```
   C#

   Console.WriteLine($"Last push: {repo.LastPush}");
   ```

3. 完整的应用应类似于以下 Program.cs 文件：

   ```
   C#

   using System.Net.Http.Headers;
   using System.Text.Json;

   using HttpClient client = new();
   ```

```
client.DefaultRequestHeaders.Accept.Clear();
client.DefaultRequestHeaders.Accept.Add(
    new
MediaTypeWithQualityHeaderValue("application/vnd.github.v3+json"));
client.DefaultRequestHeaders.Add("User-Agent", ".NET Foundation
Repository Reporter");

var repositories = await ProcessRepositoriesAsync(client);

foreach (var repo in repositories)
{
    Console.WriteLine($"Name: {repo.Name}");
    Console.WriteLine($"Homepage: {repo.Homepage}");
    Console.WriteLine($"GitHub: {repo.GitHubHomeUrl}");
    Console.WriteLine($"Description: {repo.Description}");
    Console.WriteLine($"Watchers: {repo.Watchers:#,0}");
    Console.WriteLine($"{repo.LastPush}");
    Console.WriteLine();
}

static async Task<List<Repository>> ProcessRepositoriesAsync(HttpClient
client)
{
    await using Stream stream =
        await
client.GetStreamAsync("https://api.github.com/orgs/dotnet/repos");
    var repositories =
        await JsonSerializer.DeserializeAsync<List<Repository>>
(stream);
    return repositories ?? new();
}
```

4. 运行应用。

   输出包括上次推送到每个存储库的日期和时间。

后续步骤

在本教程中，你创建了一个能够发出 web 请求并分析结果的应用。 你的应用版本现在应与已完成的示例匹配。

若要详细了解如何在如何在 .NET 中序列化和反序列化（封送和拆收）JSON 中配置 JSON 序列化。

使用语言集成查询 (LINQ)

项目 • 2023/05/10

简介

本教程将介绍 .NET Core 和 C# 语言的功能。 你将了解如何执行以下操作：

- 使用 LINQ 生成序列。
- 编写可轻松用于 LINQ 查询的方法。
- 区分及早计算和惰性计算。

你将通过生成应用程序来了解如何执行这些操作，应用程序体现了所有魔术师都具备的一项基本技能，即完美洗牌。 简而言之，完美洗牌这项技能是指，将一副纸牌分成两半，然后两手各拿一半交错洗牌（一张隔一张），以便重新生成原来的一副纸牌。

魔术师之所以要掌握这项技能是因为，每次洗牌后每张纸牌的位置已知，且顺序为重复模式。

考虑到你的目的，数据序列控制起来就会非常轻松。 生成的应用程序会构造一副纸牌，然后执行一系列洗牌操作，每次都会输出序列。 还可以将更新后的顺序与原始顺序进行比较。

在此教程中，将执行多步操作。 执行每步操作后，都可以运行应用程序，并查看进度。 还可参阅 dotnet/samples GitHub 存储库中的完整示例。 有关下载说明，请参阅示例和教程。

先决条件

必须将计算机设置为运行 .Net Core。 有关安装说明，请访问 .NET Core 下载页。 可以在 Windows、Ubuntu Linux、OS X 或 Docker 容器中运行此应用程序。 必须安装常用的代码编辑器。 在以下说明中，我们使用的是开放源代码跨平台编辑器 Visual Studio Code。 不过，你可以使用习惯使用的任意工具。

创建应用程序

第一步是新建应用程序。 打开命令提示符，然后新建应用程序的目录。 将新建的目录设为当前目录。 在命令提示符处，键入命令 `dotnet new console`。 这将为基本的“Hello World”应用程序创建起始文件。

如果之前从未用过 C#，请参阅这篇教程，其中介绍了 C# 程序的结构。 可以阅读相应的内容，然后回到此教程详细了解 LINQ。

创建数据集

开始前，请确保下列行在 `dotnet new console` 生成的 `Program.cs` 文件的顶部：

C#

```
// Program.cs
using System;
using System.Collections.Generic;
using System.Linq;
```

如果这三行（`using` 语句）未在该文件的顶部，程序将无法编译。

现在已具备所需的所有引用，接下来可以考虑一副扑克牌是由什么构成的。 通常一副扑克牌包含四种花色，每种花色包含 13 个值。 通常情况下，你可能会立即考虑创建一个 `Card` 类，然后手动填充一组 `Card` 对象。 相对于通常的方式，使用 LINQ 创建一副扑克牌更加简捷。 可以创建两个序列来分别表示花色和点数，而非创建 `Card` 类。 创建两个非常简单的迭代器方法，用于将级别和花色生成为 IEnumerable<T> 字符串：

C#

```
// Program.cs
// The Main() method

static IEnumerable<string> Suits()
{
    yield return "clubs";
    yield return "diamonds";
    yield return "hearts";
    yield return "spades";
}

static IEnumerable<string> Ranks()
{
    yield return "two";
    yield return "three";
    yield return "four";
    yield return "five";
    yield return "six";
    yield return "seven";
    yield return "eight";
    yield return "nine";
    yield return "ten";
    yield return "jack";
    yield return "queen";
```

```
    yield return "king";
    yield return "ace";
}
```

将它们置于 `Main` 文件的 `Program.cs` 方法的下面。 这两种方法都利用 `yield return` 语法在运行时生成序列。 编译器会生成对象来实现 IEnumerable<T>，并在有请求时生成字符串序列。

现在使用这些迭代器创建一副扑克牌。 将 LINQ 查询置于 `Main` 方法中。 具体如下所示：

C#

```
// Program.cs
static void Main(string[] args)
{
    var startingDeck = from s in Suits()
                       from r in Ranks()
                       select new { Suit = s, Rank = r };

    // Display each card that we've generated and placed in startingDeck in
the console
    foreach (var card in startingDeck)
    {
        Console.WriteLine(card);
    }
}
```

多个 `from` 子句生成 SelectMany，用于将第一个和第二个序列中的所有元素合并成一个序列。 考虑到我们的目的，顺序非常重要。 第一个源序列（花色）中的首个元素与第二个序列（等级）中的每个元素结合使用。 这就生成了第一个花色的所有十三张纸牌。 对第一个序列（花色）中的每个元素重复此过程。 最后生成按花色排序（后跟值）的一副纸牌。

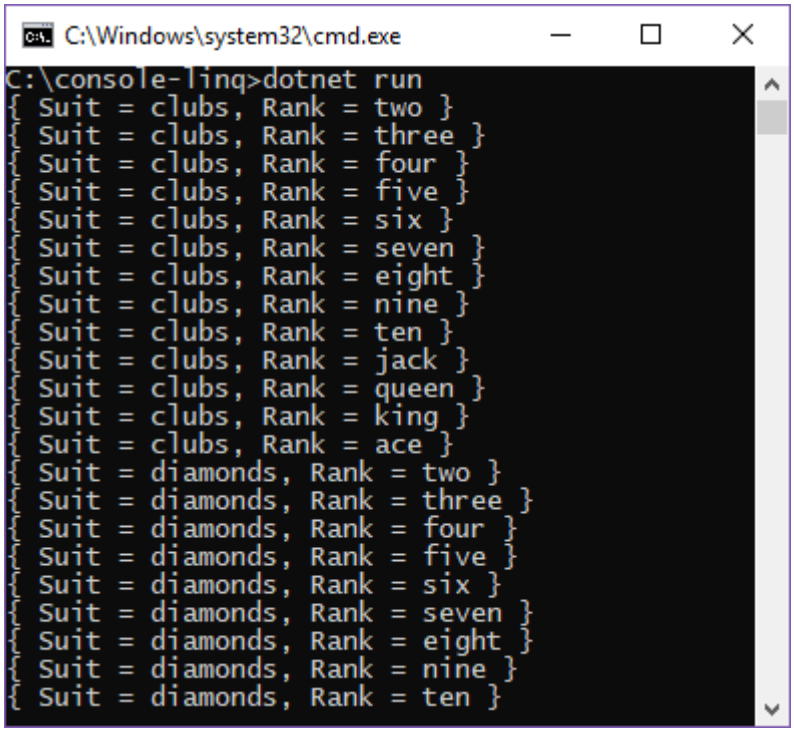
务必注意，无论是选择使用上文所用的查询语法编写 LINQ，还是使用方法语法，始终都可以从一种语法形式转至另一种。 可使用方法语法编写使用查询语法编写的上述查询，如下所示：

C#

```
var startingDeck = Suits().SelectMany(suit => Ranks().Select(rank => new {
Suit = suit, Rank = rank }));
```

编译器会将使用查询语法编写的 LINQ 语句转换为等效的方法调用语法。 因此无论选择哪种语法，两种查询版本生成的结果相同。 选择最适合你的情况的语法：例如，若所在的工作团队中的某些成员不擅长方法语法，则尽量首选使用查询语法。

此时，运行已生成的示例。 将显示一副纸牌中的所有 52 张纸牌。 在调试器模式下运行此示例来观察 `Suits()` 和 `Ranks()` 方法的执行情况，你可能会觉得非常有用。 可以清楚地看到，每个序列中的所有字符串仅在需要时生成。



操作顺序

接下来重点介绍如何洗牌。 正确洗牌的第一步是将一副扑克牌分成两半。 LINQ API 包含的 Take 和 Skip 方法提供了这种功能。 将它们置于 `foreach` 循环的下面：

C#

```
// Program.cs
public static void Main(string[] args)
{
    var startingDeck = from s in Suits()
                       from r in Ranks()
                       select new { Suit = s, Rank = r };

    foreach (var c in startingDeck)
    {
        Console.WriteLine(c);
    }

    // 52 cards in a deck, so 52 / 2 = 26
    var top = startingDeck.Take(26);
    var bottom = startingDeck.Skip(26);
}
```

但标准库中没有可供使用的洗牌方法，因此必须自行编写。 将要创建的洗牌方法体现了要对基于 LINQ 的程序执行的几种操作，因此我们将逐步介绍此过程的每个部分。

需要编写几种特殊的方法，我们称之为IEnumerable<T>扩展方法，来添加一些功能，以便于与 LINQ 查询返回的 交互。 简而言之，扩展方法是具有特殊用途的静态方法，借助它无需修改你想要为其添加功能的已有原始类型，即可向其添加功能。

向程序添加新的静态类文件（名称为 `Extensions.cs`），以用于存放扩展方法，然后开始生成第一个扩展方法：

C#

```
// Extensions.cs
using System;
using System.Collections.Generic;
using System.Linq;

namespace LinqFaroShuffle
{
    public static class Extensions
    {
        public static IEnumerable<T> InterleaveSequenceWith<T>(this
IEnumerable<T> first, IEnumerable<T> second)
        {
            // Your implementation will go here soon enough
        }
    }
}
```

仔细观察方法签名，尤其是参数：

C#

```
public static IEnumerable<T> InterleaveSequenceWith<T> (this IEnumerable<T>
first, IEnumerable<T> second)
```

可以发现，在扩展方法的第一个自变量中添加了 `this` 修饰符。 也就是说，调用扩展方法，就像是第一个自变量类型的成员方法一样。 此方法声明还遵循标准惯用做法，其中输入和输出类型为 `IEnumerable<T>`。 遵循这种做法，可以将 LINQ 方法链在一起，从而执行更复杂的查询。

正常情况下，将扑克牌分为两半后，需要将这两半合并在一起。 在代码中，这意味着一次性地枚举通过 Take 和 Skip 获得的两个序列，*`interleaving`* 元素，并创建一个序列：即现在洗牌后的扑克牌。 必须了解 IEnumerable<T> 的工作原理，才能编写处理两个序列的 LINQ 方法。

IEnumerable<T> 接口有一个方法 (GetEnumerator)。 GetEnumerator 返回的对象包含用于移动到下一个元素的方法，以及用于检索序列中当前元素的属性。 将使用这两个成员来枚举集合并返回元素。 由于此交错方法是迭代器方法，因此将使用上面的 `yield return` 语法，而不用生成并返回集合。

下面是此方法的实现代码：

C#

```
public static IEnumerable<T> InterleaveSequenceWith<T>
    (this IEnumerable<T> first, IEnumerable<T> second)
{
    var firstIter = first.GetEnumerator();
    var secondIter = second.GetEnumerator();

    while (firstIter.MoveNext() && secondIter.MoveNext())
    {
        yield return firstIter.Current;
        yield return secondIter.Current;
    }
}
```

至此，你已编写好这个方法，请回到 `Main` 方法，并进行一次洗牌：

C#

```
// Program.cs
public static void Main(string[] args)
{
    var startingDeck = from s in Suits()
                       from r in Ranks()
                       select new { Suit = s, Rank = r };

    foreach (var c in startingDeck)
    {
        Console.WriteLine(c);
    }

    var top = startingDeck.Take(26);
    var bottom = startingDeck.Skip(26);
    var shuffle = top.InterleaveSequenceWith(bottom);

    foreach (var c in shuffle)
    {
        Console.WriteLine(c);
    }
}
```

比较

进行多少次洗牌才能恢复一副纸牌的原始顺序？ 若要解决这个问题，需要编写用于确定两个序列是否相等的方法。 编写好此方法后，需要循环执行用于洗牌的代码，看看一副纸牌何时才能恢复原始顺序。

编写用于确定两个序列是否相等的方法应该很简单。 此方法的结构与你编写的洗牌方法类似。 不同之处在于，这一次将比较每个序列的匹配元素，而不是 `yield return` 每个元素。 枚举整个序列后，如果每个元素都一致，那么序列就是相同的：

C#

```
public static bool SequenceEquals<T>
    (this IEnumerable<T> first, IEnumerable<T> second)
{
    var firstIter = first.GetEnumerator();
    var secondIter = second.GetEnumerator();

    while ((firstIter?.MoveNext() == true) && secondIter.MoveNext())
    {
        if ((firstIter.Current is not null) &&
!firstIter.Current.Equals(secondIter.Current))
        {
            return false;
        }
    }

    return true;
}
```

这反映了另一种 LINQ 惯用做法，即终端方法。 此类方法需要将序列（或在此示例中，为两个序列）用作输入，并返回一个标量值。 使用终端方法时，它们始终是 LINQ 查询方法链中最后的方法，因此其名称为“终端”。

使用此类方法来确定一副纸牌何时恢复原始顺序时，就可以了解实际效果。 循环执行洗牌代码，通过应用 `SequenceEquals()` 方法确定序列已恢复原始顺序时停止执行。 你会发现，此类方法始终是任何查询中的最后一个方法，因为返回的是一个值，而不是一个序列：

C#

```
// Program.cs
static void Main(string[] args)
{
    // Query for building the deck

    // Shuffling using InterleaveSequenceWith<T>();
```

```
    var times = 0;
    // We can re-use the shuffle variable from earlier, or you can make a
new one
    shuffle = startingDeck;
    do
    {
        shuffle = shuffle.Take(26).InterleaveSequenceWith(shuffle.Skip(26));

        foreach (var card in shuffle)
        {
            Console.WriteLine(card);
        }
        Console.WriteLine();
        times++;

    } while (!startingDeck.SequenceEquals(shuffle));

    Console.WriteLine(times);
}
```

运行现有的代码，并记录每次洗牌时扑克牌的重写排列方式。 进行 8 次洗牌后（迭代 do-while 循环），扑克牌恢复从最初的 LINQ 查询首次创建它时的原始配置。

优化

到目前为止，你已生成的示例执行的是向外洗牌，即每次洗牌时第一张和最后一张纸牌保持不变。 让我们来做一点改变，改为使用向内洗牌，改变全部 52 张纸牌的位置。 向内洗牌是指，交错一副纸牌时，将后一半中的第一张纸牌变成一副纸牌中的第一张纸牌。 也就是说，上半部分中的最后一张纸牌变成一副纸牌中的最后一张纸牌。 这是对单行代码的简单更改。 交换 Take 和 Skip 的位置，来更新当前洗牌查询。 这会更改一副纸牌的上半部分和下半部分的顺序：

C#

```
shuffle = shuffle.Skip(26).InterleaveSequenceWith(shuffle.Take(26));
```

再次运行程序，你会发现需要进行 52 次迭代才能恢复一副纸牌的原始顺序。 随着程序的继续运行，你还会开始注意到一些非常严重的性能下降问题发生。

导致这种情况出现的原因有很多。 可以解决导致性能下降的主要原因之一：延迟计算的使用效率低下。

简单来说，延迟计算是指直至需要语句的值时才会执行语句计算。 LINQ 查询属于延迟计算的语句。 仅当有元素请求时才生成序列。 通常情况下，这是 LINQ 的主要优势所在。 不过，在诸如此程序之类的用例中，这就会导致执行时间指数式增长。

应该记得原来的一副纸牌是使用 LINQ 查询生成的。 每次洗牌操作是通过对上一副纸牌执行三次 LINQ 查询进行。 所有这些操作均采用惰性执行方式。 也就是说，每当有序列请求时，才会再次执行这些操作。 执行到第 52 次迭代时，就已经重新生成很多很多次原来的一副纸牌。 让我们来编写一个日志方法来阐明此行为。 然后，你就可以解决此问题。

在 `Extensions.cs` 文件中输入或复制下面的方法。 此扩展方法会在项目目录中新建一个名称为 `debug.log` 的文件，并将当前正在执行的查询记录到日志文件中。 可以将此扩展方法追加到任何查询中，以标记查询已执行。

C#

```
public static IEnumerable<T> LogQuery<T>
    (this IEnumerable<T> sequence, string tag)
{
    // File.AppendText creates a new file if the file doesn't exist.
    using (var writer = File.AppendText("debug.log"))
    {
        writer.WriteLine($"Executing Query {tag}");
    }

    return sequence;
}
```

你将看到 `File` 下显示红色波浪线，这表示它不存在。 它将不编译，因为编译器无法识别 `File`。 要解决此问题，请务必在 `Extensions.cs` 中第一行的下面添加以下代码行：

C#

```
using System.IO;
```

这应可解决问题，随后红色错误将消失。

接下来，使用日志消息来检测每个查询的定义：

C#

```
// Program.cs
public static void Main(string[] args)
{
    var startingDeck = (from s in Suits().LogQuery("Suit Generation")
                        from r in Ranks().LogQuery("Rank Generation")
                        select new { Suit = s, Rank = r
}).LogQuery("Starting Deck");

    foreach (var c in startingDeck)
    {
        Console.WriteLine(c);
```

```
    }

    Console.WriteLine();
    var times = 0;
    var shuffle = startingDeck;

    do
    {
        // Out shuffle
        /*
        shuffle = shuffle.Take(26)
            .LogQuery("Top Half")
            .InterleaveSequenceWith(shuffle.Skip(26)
            .LogQuery("Bottom Half"))
            .LogQuery("Shuffle");
        */

        // In shuffle
        shuffle = shuffle.Skip(26).LogQuery("Bottom Half")
                .InterleaveSequenceWith(shuffle.Take(26).LogQuery("Top
Half"))
                .LogQuery("Shuffle");

        foreach (var c in shuffle)
        {
            Console.WriteLine(c);
        }

        times++;
        Console.WriteLine(times);
    } while (!startingDeck.SequenceEquals(shuffle));

    Console.WriteLine(times);
}
```

请注意，不是每次访问查询都会生成日志。 只有在创建原始查询时，才会生成日志。 程序的运行时间仍然很长，但现在知道原因了。 如果对在启用日志记录的情况下运行向内洗牌失去了耐心，请切换回向外洗牌。 但仍会看到惰性计算效果。 在一次运行中，共执行 2592 次查询，包括生成所有值和花色。

可以提高此处的代码性能，以减少执行次数。 一个简单的修复方法是缓存构造扑克牌的原始 LINQ 查询的结果。 目前，每当 do-while 循环进行迭代时，需要反复执行查询，每次都要重新构造扑克牌并进行洗牌。 可以利用 LINQ 方法 ToArray 和 ToList 来缓存扑克牌；将这两个方法追加到查询时，它们将执行你已告知它们要执行的同种操作，而现在它们会将结果存储在数组或列表中，具体取决于选择调用的方法。 将 LINQ 方法 ToArray 追加到两个查询中，并再次运行程序：

```
C#
```

```
public static void Main(string[] args)
{
    IEnumerable<Suit>? suits = Suits();
    IEnumerable<Rank>? ranks = Ranks();

    if ((suits is null) || (ranks is null))
        return;

    var startingDeck = (from s in suits.LogQuery("Suit Generation")
                        from r in ranks.LogQuery("Value Generation")
                        select new { Suit = s, Rank = r })
                        .LogQuery("Starting Deck")
                        .ToArray();

    foreach (var c in startingDeck)
    {
        Console.WriteLine(c);
    }

    Console.WriteLine();

    var times = 0;
    var shuffle = startingDeck;

    do
    {
        /*
        shuffle = shuffle.Take(26)
            .LogQuery("Top Half")
            .InterleaveSequenceWith(shuffle.Skip(26).LogQuery("Bottom
Half"))
            .LogQuery("Shuffle")
            .ToArray();
        */

        shuffle = shuffle.Skip(26)
            .LogQuery("Bottom Half")
            .InterleaveSequenceWith(shuffle.Take(26).LogQuery("Top Half"))
            .LogQuery("Shuffle")
            .ToArray();

        foreach (var c in shuffle)
        {
            Console.WriteLine(c);
        }

        times++;
        Console.WriteLine(times);
    } while (!startingDeck.SequenceEquals(shuffle));

    Console.WriteLine(times);
}
```

现在向外洗牌下降到 30 次查询。 再次运行向内洗牌程序，改善情况类似：现在它执行 162 次查询。

请注意，此示例旨在突出显示延迟计算可能会导致性能下降的用例。 了解延迟计算会在何处影响代码性能至关重要，但了解并非所有查询应及早运行也同等重要。 未使用 ToArray 会导致性能损失，这是因为每次重新排列一副纸牌都要以先前的排列为依据。 使用惰性计算意味着一副纸牌的每个新配置都以原来的一副纸牌为依据，甚至在执行生成 `startingDeck` 的代码时，也不例外。 这会导致大量额外的工作。

在实践中，一些算法使用及早计算的效果较好，另一些算法使用延迟计算的效果较好。 对于日常使用，如果数据源为单独进程（如数据库引擎），通常更好的选择是使用延迟计算。 对于数据库，使用延迟计算，更复杂的查询可以只对数据库进程执行一次往返，然后返回至剩余的代码。 无论选择使用延迟计算还是及早计算，LINQ 均可以灵活处理，因此请衡量自己的进程，然后选择可为你提供最佳性能的计算种类。

结束语

在此项目中介绍了下列内容：

- 使用 LINQ 查询，来将数据聚合到有意义的序列中
- 编写扩展方法，来将自己的自定义功能添加到 LINQ 查询
- 查找 LINQ 查询可能会在其中遇到性能问题（如速度下降）的代码区域
- 与 LINQ 查询相关的延迟计算和及早计算，以及它们对查询性能的影响

除 LINQ 外，还简单介绍了魔术师用于扑克牌魔术的一个技术。 魔术师之所以采用完美洗牌是因为，可以控制每张纸牌在一副纸牌中的移动。 现在你了解了，也不要告诉其他人以免破坏他们的兴致！

有关 LINQ 的更多信息，请访问：

- 语言集成查询 (LINQ)
- LINQ 介绍
- 基本 LINQ 查询操作 (C#)
- 使用 LINQ 进行数据转换 (C#)
- LINQ 中的查询语法和方法语法 (C#)
- 支持 LINQ 的 C# 功能

语言集成查询 (LINQ)

项目 • 2024/04/11

语言集成查询 (LINQ) 是一系列直接将查询功能集成到 C# 语言的技术统称。 数据查询历来都表示为简单的字符串，没有编译时类型检查或 IntelliSense 支持。 此外，需要针对每种类型的数据源了解不同的查询语言：SQL 数据库、XML 文档、各种 Web 服务等。 借助 LINQ，查询成为了最高级的语言构造，就像类、方法和事件一样。

对于编写查询的开发者来说，LINQ 最明显的“语言集成”部分就是查询表达式。 查询表达式采用声明性*查询语法*编写而成。 使用查询语法，可以用最少的代码对数据源执行筛选、排序和分组操作。 可使用相同的基本查询表达式模式来查询和转换 SQL 数据库、ADO .NET 数据集、XML 文档和流以及 .NET 集合中的数据。

下面的示例展示了完整的查询操作。 完整的操作包括创建数据源、定义查询表达式和在 `foreach` 语句中执行查询。

C#

```
// Specify the data source.
int[] scores = { 97, 92, 81, 60 };

// Define the query expression.
IEnumerable<int> scoreQuery =
    from score in scores
    where score > 80
    select score;

// Execute the query.
foreach (int i in scoreQuery)
{
    Console.Write(i + " ");
}

// Output: 97 92 81
```

查询表达式概述

- 查询表达式可用于查询并转换所有启用了 LINQ 的数据源中的数据。 例如，通过一个查询即可检索 SQL 数据库中的数据，并生成 XML 流作为输出。
- 查询表达式易于掌握，因为它们使用了许多熟悉的 C# 语言构造。
- 查询表达式中的变量全都是强类型，尽管在许多情况下，无需显式提供类型，因为编译器可以推断出。 有关详细信息，请参阅 LINQ 查询操作中的类型关系。

- 只有在循环访问查询变量后，才会执行查询（例如，在 `foreach` 语句中）。 有关详细信息，请参阅 LINQ 查询简介。
- 在编译时，查询表达式根据 C# 规范规则转换成标准查询运算符方法调用。 可使用查询语法表示的任何查询都可以使用方法语法进行表示。 不过，在大多数情况下，查询语法的可读性更高，也更为简洁。 有关详细信息，请参阅 C# 语言规范和标准查询运算符概述。
- 通常，我们建议在编写 LINQ 查询时尽量使用查询语法，并在必要时尽可能使用方法语法。 这两种不同的形式在语义或性能上毫无差异。 查询表达式通常比使用方法语法编写的等同表达式更具可读性。
- 一些查询操作（如 Count 或 Max）没有等效的查询表达式子句，因此必须表示为方法调用。 可以各种方式结合使用方法语法和查询语法。 有关详细信息，请参阅 LINQ 中的查询语法和方法语法。
- 查询表达式可被编译成表达式树或委托，具体视应用查询的类型而定。 IEnumerable<T> 查询编译为委托。 IQueryable 和 IQueryable<T> 查询编译为表达式树。 有关详细信息，请参阅表达式树。

后续步骤

若要详细了解 LINQ，请先自行熟悉查询表达式基础知识中的一些基本概念，然后再阅读感兴趣的 LINQ 技术的相关文档：

- XML 文档：LINQ to XML
- ADO.NET 实体框架：LINQ to Entities
- .NET 集合、文件、字符串等：LINQ to objects

若要更深入地全面了解 LINQ，请参阅 C# 中的 LINQ。

若要开始在 C# 中使用 LINQ，请参阅教程使用 LINQ。

反馈

此页面是否有帮助? 是 否

提供产品反馈

C# 中的 LINQ 查询简介

项目 • 2024/04/30

*查询*是一种从数据源检索数据的表达式。 不同的数据源具有不同的原生查询语言，例如，用于关系数据库的 SQL 和用于 XML 的 XQuery。 开发人员对于他们必须支持的每种数据源或数据格式，都必须学习一种新的查询语言。 LINQ 通过为各种数据源和数据格式提供一致的 C# 语言模型，简化了这一情况。 在 LINQ 查询中，你始终使用 C# 对象。 当有 LINQ 提供程序可用时，你可以使用相同的基本编码模式来查询和转换 XML 文档、SQL 数据库、.NET 集合中的数据以及任何其他格式的数据。

查询操作的三个部分

所有 LINQ 查询操作都由以下三个不同的操作组成：

1. 获取数据源。
2. 创建查询。
3. 执行查询。

下面的示例演示如何用源代码表示查询操作的三个部分。 为方便起见，此示例将一个整数数组用作数据源；但其中涉及的概念同样适用于其他数据源。 本文的其余部分也引用了此示例。

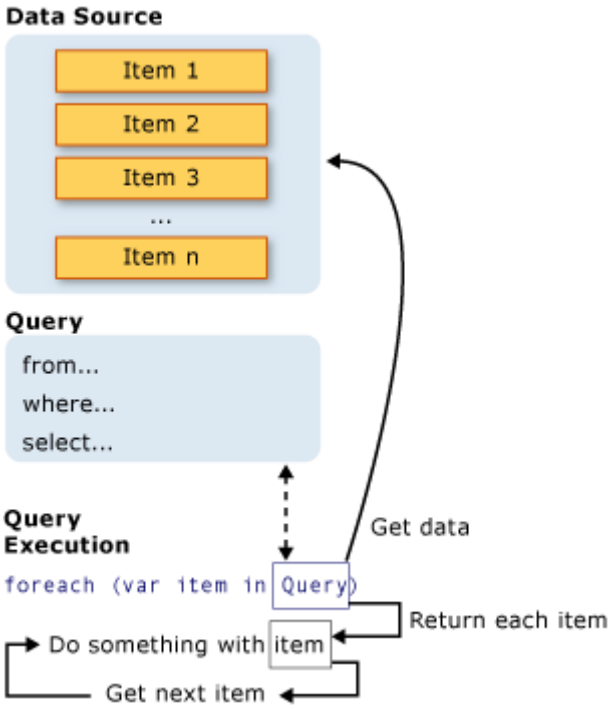
C#

```
// The Three Parts of a LINQ Query:
// 1. Data source.
int[] numbers = [ 0, 1, 2, 3, 4, 5, 6 ];

// 2. Query creation.
// numQuery is an IEnumerable<int>
var numQuery =
    from num in numbers
    where (num % 2) == 0
    select num;

// 3. Query execution.
foreach (int num in numQuery)
{
    Console.Write("{0,1} ", num);
}
```

下图演示完整的查询操作。 在 LINQ 中，查询的执行不同于查询本身。 换句话说，你不会通过创建查询变量来检索任何数据。



数据源

上例中的数据源是一个数组，它支持泛型 IEnumerable<T> 接口。 这一事实意味着该数据源可以用 LINQ 进行查询。 查询在 `foreach` 语句中执行，且 `foreach` 需要 IEnumerable 或 IEnumerable<T>。 支持 IEnumerable<T> 或派生接口（如泛型 IQueryable<T>）的类型称为可查询类型 。

可查询类型不需要进行修改或特殊处理就可以用作 LINQ 数据源。 如果源数据还没有作为可查询类型出现在内存中，则 LINQ 提供程序必须以此方式表示源数据。 例如，LINQ to XML 将 XML 文档加载到可查询的 XElement 类型中：

C#

```
// Create a data source from an XML document.
// using System.Xml.Linq;
XElement contacts = XElement.Load(@"c:\myContactList.xml");
```

使用 EntityFramework，你在 C# 类与数据库架构之间创建对象关系映射。 你针对这些对象编写查询，然后 EntityFramework 在运行时处理与数据库的通信。 下例中，`Customers` 表示数据库中的特定表，而查询结果的类型 IQueryable<T> 派生自 IEnumerable<T>。

C#

```
Northwnd db = new Northwnd(@"c:\northwnd.mdf");

// Query for customers in London.
IQueryable<Customer> custQuery =
    from cust in db.Customers
```

```
    where cust.City == "London"
    select cust;
```

有关如何创建特定类型的数据源的详细信息，请参阅各种 LINQ 提供程序的文档。 但基本规则很简单：LINQ 数据源是支持泛型 IEnumerable<T> 接口（或者是继承该接口的接口，通常表示为 IQueryable<T>）的任何对象。

> ⓘ **备注**
>
> 支持非泛型 IEnumerable 接口的类型（如 ArrayList）还可用作 LINQ 数据源。 有关详细信息，请参阅**如何使用 LINQ 查询 ArrayList (C#)**。

查询

查询指定要从数据源中检索的信息。 查询还可以指定在返回这些信息之前如何对其进行排序、分组和结构化。 查询存储在查询变量中，并用查询表达式进行初始化。 你使用 C# 查询语法来编写查询。

上一个示例中的查询从整数数组中返回所有偶数。 该查询表达式包含三个子句：`from`、`where` 和 `select`。 （如果你熟悉 SQL，你会注意到这些子句的顺序与 SQL 中的顺序相反。）`from` 子句指定数据源，`where` 子句应用筛选器，`select` 子句指定返回的元素的类型。 本部分详细介绍了所有查询子句。 目前需要注意的是，在 LINQ 中，查询变量本身不执行任何操作并且不返回任何数据。 它只是存储在以后某个时刻执行查询时为生成结果而必需的信息。 有关如何构造查询的详细信息，请参阅标准查询运算符概述 (C#)。

> ⓘ **备注**
>
> 还可以使用方法语法来表示查询。 有关详细信息，请参阅 **LINQ 中的查询语法和方法语法**。

标准查询运算符按执行方式的分类

标准查询运算符方法的 LINQ to Objects 实现主要通过两种方法之一执行：立即执行和延迟执行。 使用延迟执行的查询运算符可以进一步分为两种类别：流式处理和非流式处理。

即时

立即执行指的是读取数据源并执行一次运算。 返回标量结果的所有标准查询运算符都立即执行。 `Count`、`Max`、`Average` 和 `First` 就属于此类查询。 由于查询本身必须使用 `foreach` 来返回结果，因此这些方法在执行时不使用显式 `foreach` 语句。 这些查询返回单个值，而不是 `IEnumerable` 集合。 可以使用 Enumerable.ToList 或 Enumerable.ToArray 方法强制任何查询立即执行。 立即执行可重用查询结果，而不是查询声明。 结果被检索一次，然后存储以供将来使用。 下面的查询返回源数组中偶数的计数：

C#

```
var evenNumQuery =
    from num in numbers
    where (num % 2) == 0
    select num;

int evenNumCount = evenNumQuery.Count();
```

要强制立即执行任何查询并缓存其结果，可调用 ToList 或 ToArray 方法。

C#

```
List<int> numQuery2 =
    (from num in numbers
        where (num % 2) == 0
        select num).ToList();

// or like this:
// numQuery3 is still an int[]

var numQuery3 =
    (from num in numbers
        where (num % 2) == 0
        select num).ToArray();
```

此外，还可以通过在紧跟查询表达式之后的位置放置一个 `foreach` 循环来强制执行查询。 但是，通过调用 `ToList` 或 `ToArray`，也可以将所有数据缓存在单个集合对象中。

已推迟

延迟执行指的是不在代码中声明查询的位置执行运算。 仅当对查询变量进行枚举时才执行运算，例如通过使用 `foreach` 语句执行。 查询的执行结果取决于执行查询而非定义查询时的数据源内容。 如果多次枚举查询变量，则每次结果可能都不同。 几乎所有返回类型为 IEnumerable<T> 或 IOrderedEnumerable<TElement> 的标准查询运算符皆以延迟方式执行。 延迟执行提供了查询重用功能，因为在每次循环访问查询结果时，查询都会从数据源中提取更新的数据。 以下代码演示了延迟执行的示例：

C#

```
foreach (int num in numQuery)
{
    Console.Write("{0,1} ", num);
}
```

`foreach` 语句也是检索查询结果的地方。 例如，在上一个查询中，迭代变量 `num` 保存了返回的序列中的每个值（一次保存一个值）。

由于查询变量本身从不保存查询结果，因此你可以重复执行它来检索更新的数据。 例如，单独的应用程序可能会不断更新数据库。 在应用程序中，你可以创建一个检索最新数据的查询，并可以不时地执行该查询以便检索更新的结果。

使用延迟执行的查询运算符可以进一步分类为流式处理和非流式处理。

流式处理

流式处理运算符不需要在生成元素前读取所有源数据。 在执行时，流式处理运算符一边读取每个源元素，一边对该源元素执行运算，并在可行时生成元素。 流式处理运算符将持续读取源元素直到可以生成结果元素。 这意味着可能要读取多个源元素才能生成一个结果元素。

非流式处理

非流式处理运算符必须先读取所有源数据，然后才能生成结果元素。 排序或分组等运算均属于此类别。 在执行时，非流式处理查询运算符读取所有源数据，将其放入数据结构，执行运算，然后生成结果元素。

分类表

下表按照执行方法对每个标准查询运算符方法进行了分类。

> **备注**
>
> 如果某个运算符被标入两个列中，则表示在运算中涉及两个输入序列，每个序列的计算方式不同。 在此类情况下，参数列表中的第一个序列始终以延迟流式处理方式来执行计算。

展开表

标准查询运算符	返回类型	立即执行	延迟的流式处理执行	延迟非流式处理执行
Aggregate	`TSource`	X		
All	Boolean	X		
Any	Boolean	X		
AsEnumerable	IEnumerable<T>		X	
Average	单个数值	x		
Cast	IEnumerable<T>		X	
Concat	IEnumerable<T>		X	
Contains	Boolean	X		
Count	Int32	X		
DefaultIfEmpty	IEnumerable<T>		X	
Distinct	IEnumerable<T>		X	
ElementAt	`TSource`	X		
ElementAtOrDefault	`TSource?`	X		
Empty	IEnumerable<T>	X		
Except	IEnumerable<T>		X	X
First	`TSource`	X		
FirstOrDefault	`TSource?`	X		
GroupBy	IEnumerable<T>			X
GroupJoin	IEnumerable<T>		X	X
Intersect	IEnumerable<T>		X	X
Join	IEnumerable<T>		X	X
Last	`TSource`	X		
LastOrDefault	`TSource?`	X		
LongCount	Int64	X		
Max	单个数值 `TSource` 或 `TResult?`	X		

标准查询运算符	返回类型	立即执行	延迟的流式处理执行	延迟非流式处理执行
Min	单个数值 `TSource` 或 `TResult?`	X		
OfType	IEnumerable<T>		X	
OrderBy	IOrderedEnumerable<TElement>			X
OrderByDescending	IOrderedEnumerable<TElement>			X
Range	IEnumerable<T>		X	
Repeat	IEnumerable<T>		X	
Reverse	IEnumerable<T>			X
Select	IEnumerable<T>		X	
SelectMany	IEnumerable<T>		X	
SequenceEqual	Boolean	X		
Single	`TSource`	X		
SingleOrDefault	`TSource?`	X		
Skip	IEnumerable<T>		X	
SkipWhile	IEnumerable<T>		X	
Sum	单个数值	x		
Take	IEnumerable<T>		X	
TakeWhile	IEnumerable<T>		X	
ThenBy	IOrderedEnumerable<TElement>			X
ThenByDescending	IOrderedEnumerable<TElement>			X
ToArray	`TSource[]` 数组	X		
ToDictionary	Dictionary<TKey,TValue>	X		
ToList	IList<T>	X		
ToLookup	ILookup<TKey,TElement>	X		
Union	IEnumerable<T>		X	
Where	IEnumerable<T>		X	

LINQ to objects

“LINQ to Objects”是指将 LINQ 查询直接用于任何 IEnumerable 或 IEnumerable<T> 集合。 可以使用 LINQ 来查询任何可枚举的集合，例如 List<T>、Array 或 Dictionary<TKey,TValue>。 该集合可以是用户定义的集合，也可以是由 .NET API 返回的类型。 而采用 LINQ 方法，只需编写描述要检索的内容的声明性代码。 LINQ to Objects 很好地介绍了如何使用 LINQ 进行编程。

LINQ 查询与传统 `foreach` 循环相比具有三大优势：

- 它们更简明、更易读，尤其在筛选多个条件时。
- 它们使用最少的应用程序代码提供强大的筛选、排序和分组功能。
- 无需修改或只需做很小的修改即可将它们移植到其他数据源。

要对数据执行的操作越复杂，就越能体会到 LINQ 相较于传统迭代技术的优势。

在内存中存储查询结果

查询基本上是针对如何检索和组织数据的一套说明。 当请求结果中的每个后续项目时，查询将延迟执行。 使用 `foreach` 循环访问结果时，项将在受到访问时返回。 若要在不执行 `foreach` 循环的情况下评估查询并存储其结果，只需调用查询变量上的以下方法之一：

- ToList
- ToArray
- ToDictionary
- ToLookup

在存储查询结果时，应将返回的集合对象分配给一个新变量，如下面的示例所示：

C#

```
List<int> numbers = [1, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20];

IEnumerable<int> queryFactorsOfFour =
    from num in numbers
    where num % 4 == 0
    select num;

// Store the results in a new variable
// without executing a foreach loop.
var factorsofFourList = queryFactorsOfFour.ToList();

// Read and write from the newly created list to demonstrate that it holds
data.
Console.WriteLine(factorsofFourList[2]);
```

```
factorsofFourList[2] = 0;
Console.WriteLine(factorsofFourList[2]);
```

另请参阅

- 演练：用 C# 编写查询
- foreach, in
- 查询关键字 (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

查询表达式基础

项目 • 2024/03/06

本文介绍与 C# 中的查询表达式相关的基本概念。

查询是什么及其作用是什么?

查询是一组指令，描述要从给定数据源（或源）检索的数据以及返回的数据应具有的形状和组织。 查询与它生成的结果不同。

通常情况下，源数据按逻辑方式组织为相同类型的元素的序列。 例如，SQL 数据库表包含行的序列。 在 XML 文件中，存在 XML 元素的“序列”（尽管 XML 元素在树结构按层次结构进行组织）。 内存中集合包含对象的序列。

从应用程序的角度来看，原始源数据的特定类型和结构并不重要。 应用程序始终将源数据视为 IEnumerable<T> 或 IQueryable<T> 集合。 例如，在 LINQ to XML 中，源数据显示为 `IEnumerable`<XElement>。

对于此源序列，查询可能会执行三种操作之一：

- 检索元素的子集以生成新序列，而不修改各个元素。 然后，查询可能以各种方式对返回的序列进行排序或分组，如下面的示例所示（假定 `scores` 是 `int[]`）：

  C#

  ```
  IEnumerable<int> highScoresQuery =
      from score in scores
      where score > 80
      orderby score descending
      select score;
  ```

- 如前面的示例所示检索元素的序列，但是将它们转换为新类型的对象。 例如，查询可能只从数据源中的某些客户记录检索姓氏。 或者可以检索完整记录，然后用于构造其他内存中对象类型甚至是 XML 数据，再生成最终的结果序列。 下面的示例演示从 `int` 到 `string` 的投影。 请注意 `highScoresQuery` 的新类型。

  C#

  ```
  IEnumerable<string> highScoresQuery2 =
      from score in scores
      where score > 80
      orderby score descending
      select $"The score is {score}";
  ```

- 检索有关源数据的单独值，如：
  - 与特定条件匹配的元素数。
  - 具有最大或最小值的元素。
  - 与某个条件匹配的第一个元素，或指定元素集中特定值的总和。 例如，下面的查询从 `scores` 整数数组返回大于 80 的分数的数量：

    C#

    ```
    var highScoreCount = (
        from score in scores
        where score > 80
        select score
    ).Count();
    ```

    在前面的示例中，请注意在调用 Enumerable.Count 方法之前，在查询表达式两边使用了括号。 也可以通过使用新变量存储具体结果。

    C#

    ```
    IEnumerable<int> highScoresQuery3 =
        from score in scores
        where score > 80
        select score;

    var scoreCount = highScoresQuery3.Count();
    ```

在上面的示例中，查询在 `Count` 调用中执行，因为 `Count` 必须循环访问结果才能确定 `highScoresQuery` 返回的元素数。

查询表达式是什么?

查询表达式是以查询语法表示的查询。 查询表达式是一流的语言构造。 它如同任何其他表达式一样，可以在 C# 表达式有效的任何上下文中使用。 查询表达式由一组用类似于 SQL 或 XQuery 的声明性语法所编写的子句组成。 每个子句又包含一个或多个 C# 表达式，而这些表达式可能本身是查询表达式或包含查询表达式。

查询表达式必须以 from 子句开头，且必须以 select 或 group 子句结尾。 在第一个 `from` 子句与最后一个 `select` 或 `group` 子句之间，可以包含以下这些可选子句中的一个或多个：where、orderby、join、let，甚至是其他 from 子句。 还可以使用 into 关键字，使 `join` 或 `group` 子句的结果可以充当相同查询表达式中的更多查询子句的源。

查询变量

在 LINQ 中，查询变量是存储查询而不是查询结果的任何变量。 更具体地说，查询变量始终是可枚举类型，在 `foreach` 语句或对其 IEnumerator.MoveNext() 方法的直接调用中循环访问时会生成元素序列。

> **备注**
>
> 本文中的示例使用以下数据源和示例数据。

C#

```
record City(string Name, long Population);
record Country(string Name, double Area, long Population, List<City>
Cities);
record Product(string Name, string Category);
```

C#

```
static readonly City[] cities = [
    new City("Tokyo", 37_833_000),
    new City("Delhi", 30_290_000),
    new City("Shanghai", 27_110_000),
    new City("São Paulo", 22_043_000),
    new City("Mumbai", 20_412_000),
    new City("Beijing", 20_384_000),
    new City("Cairo", 18_772_000),
    new City("Dhaka", 17_598_000),
    new City("Osaka", 19_281_000),
    new City("New York-Newark", 18_604_000),
    new City("Karachi", 16_094_000),
    new City("Chongqing", 15_872_000),
    new City("Istanbul", 15_029_000),
    new City("Buenos Aires", 15_024_000),
    new City("Kolkata", 14_850_000),
    new City("Lagos", 14_368_000),
    new City("Kinshasa", 14_342_000),
    new City("Manila", 13_923_000),
    new City("Rio de Janeiro", 13_374_000),
    new City("Tianjin", 13_215_000)
];

static readonly Country[] countries = [
    new Country ("Vatican City", 0.44, 526, [new City("Vatican City",
826)]),
    new Country ("Monaco", 2.02, 38_000, [new City("Monte Carlo", 38_000)]),
    new Country ("Nauru", 21, 10_900, [new City("Yaren", 1_100)]),
    new Country ("Tuvalu", 26, 11_600, [new City("Funafuti", 6_200)]),
    new Country ("San Marino", 61, 33_900, [new City("San Marino", 4_500)]),
```

```
    new Country ("Liechtenstein", 160, 38_000, [new City("Vaduz", 5_200)]),
    new Country ("Marshall Islands", 181, 58_000, [new City("Majuro",
28_000)]),
    new Country ("Saint Kitts & Nevis", 261, 53_000, [new City("Basseterre",
13_000)])
];
```

下面的代码示例演示一个简单查询表达式，它具有一个数据源、一个筛选子句、一个排序子句并且不转换源元素。 该查询以 `select` 子句结尾。

C#

```
// Data source.
int[] scores = [90, 71, 82, 93, 75, 82];

// Query Expression.
IEnumerable<int> scoreQuery = //query variable
    from score in scores //required
    where score > 80 // optional
    orderby score descending // optional
    select score; //must end with select or group

// Execute the query to produce the results
foreach (var testScore in scoreQuery)
{
    Console.WriteLine(testScore);
}

// Output: 93 90 82 82
```

在上面的示例中，`scoreQuery` 是查询变量，它有时仅仅称为查询。 查询变量不存储在 `foreach` 循环生成中的任何实际结果数据。 并且当 `foreach` 语句执行时，查询结果不会通过查询变量 `scoreQuery` 返回。 而是通过迭代变量 `testScore` 返回。 `scoreQuery` 变量可以在另一个 `foreach` 循环中进行循环访问。 只要既没有修改它，也没有修改数据源，便会生成相同结果。

查询变量可以存储采用查询语法、方法语法或是两者的组合进行表示的查询。 在以下示例中，`queryMajorCities` 和 `queryMajorCities2` 都是查询变量：

C#

```
City[] cities = [
    new City("Tokyo", 37_833_000),
    new City("Delhi", 30_290_000),
    new City("Shanghai", 27_110_000),
    new City("São Paulo", 22_043_000)
];

//Query syntax
```

```
IEnumerable<City> queryMajorCities =
    from city in cities
    where city.Population > 100000
    select city;

// Execute the query to produce the results
foreach (City city in queryMajorCities)
{
    Console.WriteLine(city);
}

// Output:
// City { Population = 120000 }
// City { Population = 112000 }
// City { Population = 150340 }

// Method-based syntax
IEnumerable<City> queryMajorCities2 = cities.Where(c => c.Population >
100000);
```

另一方面，以下两个示例演示不是查询变量的变量（即使各自使用查询进行初始化）。它们不是查询变量，因为它们存储结果：

C#

```
var highestScore = (
    from score in scores
    select score
).Max();

// or split the expression
IEnumerable<int> scoreQuery =
    from score in scores
    select score;

var highScore = scoreQuery.Max();
// the following returns the same result
highScore = scores.Max();
```

C#

```
var largeCitiesList = (
    from country in countries
    from city in country.Cities
    where city.Population > 10000
    select city
).ToList();

// or split the expression
IEnumerable<City> largeCitiesQuery =
    from country in countries
    from city in country.Cities
```

```
    where city.Population > 10000
    select city;
var largeCitiesList2 = largeCitiesQuery.ToList();
```

查询变量的显式和隐式类型化

本文档通常提供查询变量的显式类型以便显示查询变量与 select 子句之间的类型关系。 但是，还可以使用 var 关键字指示编译器在编译时推断查询变量（或任何其他局部变量）的类型。 例如，本文前面演示的查询示例也可以使用隐式类型化进行表示：

C#

```
var queryCities =
    from city in cities
    where city.Population > 100000
    select city;
```

在前面的示例中，var 的使用是可选的。 `queryCities` 是隐式或显式类型的 `IEnumerable<City>`。

开始查询表达式

查询表达式必须以 `from` 子句开头。 它指定数据源以及范围变量。 范围变量表示遍历源序列时，源序列中的每个连续元素。 范围变量基于数据源中元素的类型进行强类型化。 在下面的示例中，因为 `countries` 是 `Country` 对象的数组，所以范围变量也类型化为 `Country`。 因为范围变量是强类型，所以可以使用点运算符访问该类型的任何可用成员。

C#

```
IEnumerable<Country> countryAreaQuery =
    from country in countries
    where country.Area > 500000 //sq km
    select country;
```

范围变量一直处于范围中，直到查询使用分号或 continuation 子句退出。

查询表达式可能会包含多个 `from` 子句。 在源序列中的每个元素本身是集合或包含集合时，可使用更多 `from` 子句。 例如，假设具有 `Country` 对象的集合，其中每个对象都包含名为 `Cities` 的 `City` 对象集合。 若要查询每个 `Country` 中的 `City` 对象，请使用两个 `from` 子句，如下所示：

C#

```
IEnumerable<City> cityQuery =
    from country in countries
    from city in country.Cities
    where city.Population > 10000
    select city;
```

有关详细信息，请参阅 from 子句。

结束查询表达式

查询表达式必须以 `group` 子句或 `select` 子句结尾。

group 子句

使用 `group` 子句可生成按指定键组织的组的序列。 键可以是任何数据类型。 例如，以下查询会创建包含一个或多个 `Country` 对象，并且其关键值是数值为国家/地区名称首字母的 `char` 类型。

C#

```
var queryCountryGroups =
    from country in countries
    group country by country.Name[0];
```

有关分组的详细信息，请参阅 group 子句。

select 子句

使用 `select` 子句可生成所有其他类型的序列。 简单 `select` 子句只生成类型与数据源中包含的对象相同的对象的序列。 在此示例中，数据源包含 `Country` 对象。 `orderby` 子句只按新顺序对元素进行排序，而 `select` 子句生成重新排序的 `Country` 对象的序列。

C#

```
IEnumerable<Country> sortedQuery =
    from country in countries
    orderby country.Area
    select country;
```

`select` 子句可以用于将源数据转换为新类型的序列。 此转换也称为投影。 在下面的示例中，`select` 子句对只包含原始元素中的字段子集的匿名类型序列进行投影。 新对象使用对象初始值设定项进行初始化。

C#

```
var queryNameAndPop =
    from country in countries
    select new
    {
        Name = country.Name,
        Pop = country.Population
    };
```

因此，在此示例中，var 是必需的，因为查询会生成匿名类型。

有关可以使用 `select` 子句转换源数据的所有方法的详细信息，请参阅 select 子句。

使用“into”延续

可以在 `select` 或 `group` 子句中使用 `into` 关键字创建存储查询的临时标识符。 如果在分组或选择操作之后必须对查询执行额外查询操作，则可以使用 `into` 子句。 在下面的示例中，`countries` 按 1000 万范围，根据人口进行分组。 创建这些组之后，更多子句会筛选出一些组，然后按升序对组进行排序。 若要执行这些额外操作，需要由 `countryGroup` 表示的延续。

C#

```
// percentileQuery is an IEnumerable<IGrouping<int, Country>>
var percentileQuery =
    from country in countries
    let percentile = (int)country.Population / 10_000_000
    group country by percentile into countryGroup
    where countryGroup.Key >= 20
    orderby countryGroup.Key
    select countryGroup;

// grouping is an IGrouping<int, Country>
foreach (var grouping in percentileQuery)
{
    Console.WriteLine(grouping.Key);
    foreach (var country in grouping)
    {
        Console.WriteLine(country.Name + ":" + country.Population);
    }
}
```

有关详细信息，请参阅 into。

筛选、排序和联接

在开头 `from` 子句与结尾 `select` 或 `group` 子句之间，所有其他子句（`where`、`join`、`orderby`、`from`、`let`）都是可选的。 任何可选子句都可以在查询正文中使用零次或多次。

where 子句

使用 `where` 子句可基于一个或多个谓词表达式，从源数据中筛选出元素。 以下示例中的 `where` 子句具有一个谓词及两个条件。

C#

```
IEnumerable<City> queryCityPop =
    from city in cities
    where city.Population is < 200000 and > 100000
    select city;
```

有关详细信息，请参阅 where 子句。

orderby 子句

使用 `orderby` 子句可按升序或降序对结果进行排序。 还可以指定次要排序顺序。 下面的示例使用 `Area` 属性对 `country` 对象执行主要排序。 然后使用 `Population` 属性执行次要排序。

C#

```
IEnumerable<Country> querySortedCountries =
    from country in countries
    orderby country.Area, country.Population descending
    select country;
```

`ascending` 关键字是可选的；如果未指定任何顺序，则它是默认排序顺序。 有关详细信息，请参阅 orderby 子句。

join 子句

使用 `join` 子句可基于每个元素中指定的键之间的相等比较，将一个数据源中的元素与另一个数据源中的元素进行关联和/或合并。 在 LINQ 中，联接操作是对元素属于不同类型的对象序列执行。 联接了两个序列之后，必须使用 `select` 或 `group` 语句指定要存储在输出序列中的元素。 还可以使用匿名类型将每组关联元素中的属性合并到输出序列的新类型中。 下面的示例关联其 `Category` 属性与 `categories` 字符串数组中一个类别匹配的

`prod` 对象。 筛选出其 `Category` 不与 `categories` 中的任何字符串匹配的产品。 `select` 语句会投影其属性取自 `cat` 和 `prod` 的新类型。

C#

```
var categoryQuery =
    from cat in categories
    join prod in products on cat equals prod.Category
    select new
    {
        Category = cat,
        Name = prod.Name
    };
```

还可以通过使用 into 关键字将 `join` 操作的结果存储到临时变量中来执行分组联接。 有关详细信息，请参阅 join 子句。

let 子句

使用 `let` 子句可将表达式（如方法调用）的结果存储在新范围变量中。 在下面的示例中，范围变量 `firstName` 存储 `Split` 返回的字符串数组的第一个元素。

C#

```
string[] names = ["Svetlana Omelchenko", "Claire O'Donnell", "Sven
Mortensen", "Cesar Garcia"];
IEnumerable<string> queryFirstNames =
    from name in names
    let firstName = name.Split(' ')[0]
    select firstName;

foreach (var s in queryFirstNames)
{
    Console.Write(s + " ");
}

//Output: Svetlana Claire Sven Cesar
```

有关详细信息，请参阅 let 子句。

查询表达式中的子查询

查询子句本身可能包含查询表达式，这有时称为子查询。 每个子查询都以自己的 `from` 子句开头，该子句不一定指向第一个 `from` 子句中的相同数据源。 例如，下面的查询演示在 select 语句用于检索分组操作结果的查询表达式。

C#

```
var queryGroupMax =
    from student in students
    group student by student.Year into studentGroup
    select new
    {
        Level = studentGroup.Key,
        HighestScore = (
            from student2 in studentGroup
            select student2.ExamScores.Average()
        ).Max()
    };
```

有关详细信息，请参阅对分组操作执行子查询。

另请参阅

- 查询关键字 (LINQ)
- 标准查询运算符概述

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

编写 C# LINQ 查询以查询数据

项目 • 2024/04/26

介绍性的语言集成查询 (LINQ) 文档中的大多数查询是使用 LINQ 声明性查询语法编写的。 但是在编译代码时，查询语法必须转换为针对 .NET 公共语言运行时 (CLR) 的方法调用。 这些方法调用会调用标准查询运算符（名称为 `Where`、`Select`、`GroupBy`、`Join`、`Max` 和 `Average` 等）。 可以使用方法语法（而不查询语法）来直接调用它们。

查询语法和方法语法在语义上是相同的，但是查询语法通常更简单且更易于阅读。 某些查询必须表示为方法调用。 例如，必须使用方法调用表示检索与指定条件匹配的元素数的查询。 还必须对检索源序列中具有最大值的元素的查询使用方法调用。 System.Linq 命名空间中的标准查询运算符的参考文档通常使用方法语法。 你应该熟悉如何在查询和查询表达式本身中使用方法语法。

标准查询运算符扩展方法

下面的示例演示一个简单查询表达式以及编写为基于方法的查询的语义上等效的查询。

C#

```
int[] numbers = [ 5, 10, 8, 3, 6, 12 ];

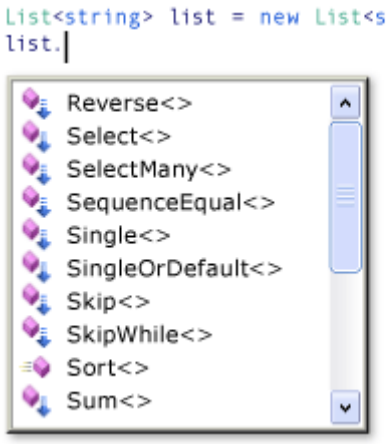
//Query syntax:
IEnumerable<int> numQuery1 =
    from num in numbers
    where num % 2 == 0
    orderby num
    select num;

//Method syntax:
IEnumerable<int> numQuery2 = numbers.Where(num => num % 2 == 0).OrderBy(n =>
n);

foreach (int i in numQuery1)
{
    Console.Write(i + " ");
}
Console.WriteLine(System.Environment.NewLine);
foreach (int i in numQuery2)
{
    Console.Write(i + " ");
}
```

这两个示例的输出是相同的。 可以看到查询变量的类型在两种形式中是相同的：IEnumerable<T>。

为了了解基于方法的查询，我们来仔细讨论它。 在表达式右侧，请注意，`where` 子句现在表示为 `numbers` 对象上的实例方法，它具有类型 `IEnumerable<int>`。 如果熟悉泛型 IEnumerable<T> 接口，则会知道它没有 `Where` 方法。 但是，如果在 Visual Studio IDE 中调用 IntelliSense 完成列表，则不仅会看到 `Where` 方法，还会看到许多其他方法（如 `Select`、`SelectMany`、`Join` 和 `Orderby`）。 这些方法实现标准查询运算符。



虽然看起来好像 IEnumerable<T> 包括其他方法，但它没有。 标准查询运算符作为*扩展方法*来实现。 扩展方法可"扩展"现有类型；它们可以如同类型上的实例方法一样进行调用。 标准查询运算符扩展了 IEnumerable<T>，因此可以写入 `numbers.Where(...)`。

若要使用扩展方法，请使用 `using` 指令将它们引入范围。 从应用程序的角度来看，扩展方法与常规实例方法是相同的。

有关扩展方法的详细信息，请参阅扩展方法。 有关标准查询运算符的详细信息，请参阅标准查询运算符概述 (C#)。 某些 LINQ 提供程序（如 实体框架和 LINQ to XML），会实现自己的标准查询运算符，并为 IEnumerable<T> 之外的其他类型实现扩展方法。

Lambda 表达式

在上面的示例中，请注意，条件表达式 (`num % 2 == 0`) 作为内联参数传递给 Enumerable.Where 方法：`Where(num => num % 2 == 0).` 此内联表达式为 lambda 表达式。 编写代码是一种方便的方法，否则必须以更繁琐的形式编写代码。 运算符左侧的 `num` 是输入变量，它与查询表达式中的 `num` 对应。 编译器可以推断出 `num` 的类型，因为它知道 `numbers` 是泛型 IEnumerable<T> 类型。 Lambda 的主体与查询语法中或任何其他 C# 表达式或语句中的表达式完全相同。 它可以包含方法调用和其他复杂逻辑。 返回值就是表达式结果。 某些查询只能采用方法语法进行表示，而其中一些查询需要 lambda 表达式。 Lambda 表达式是 LINQ 工具箱中的一个强大且灵活的工具。

查询的可组合性

在前面的代码示例中，Enumerable.OrderBy 方法通过对 `Where` 调用使用点运算符来调用。 `Where` 生成筛选序列，然后 `Orderby` 对 `Where` 所生成的序列进行排序。 由于查询返回 `IEnumerable`，因此可通过将方法调用链接在一起在方法语法中撰写查询。 使用查询语法编写查询时，编译器会执行此组合。 因为查询变量不存储查询的结果，所以可以随时修改它或将它用作新查询的基础（即使在执行它之后）。

下面的示例演示使用前面列出的每种方法的一些简单 LINQ 查询。

> **备注**
>
> 这些查询对简单的内存中集合进行操作；但是，基本语法等同于在 LINQ to Entities 和 LINQ to XML 中使用的语法。

示例 - 查询语法

使用*查询语法*编写大多数查询来创建*查询表达式*。 下面的示例演示三个查询表达式。 第一个查询表达式演示如何通过应用包含 `where` 子句的条件来筛选或限制结果。 它返回源序列中值大于 7 或小于 3 的所有元素。 第二个表达式演示如何对返回的结果进行排序。 第三个表达式演示如何根据某个键对结果进行分组。 此查询基于单词的第一个字母返回两个组。

C#

```
List<int> numbers = [5, 4, 1, 3, 9, 8, 6, 7, 2, 0];

// The query variables can also be implicitly typed by using var

// Query #1.
IEnumerable<int> filteringQuery =
    from num in numbers
    where num is < 3 or > 7
    select num;

// Query #2.
IEnumerable<int> orderingQuery =
    from num in numbers
    where num is < 3 or > 7
    orderby num ascending
    select num;

// Query #3.
string[] groupingQuery = ["carrots", "cabbage", "broccoli", "beans",
"barley"];
IEnumerable<IGrouping<char, string>> queryFoodGroups =
```

```
    from item in groupingQuery
    group item by item[0];
```

查询的类型为 IEnumerable<T>。 可以使用 var 编写所有这些查询，如下面的示例所示：

```
var query = from num in numbers...
```

在前面的每个示例中，在 foreach 语句或其他语句中循环访问查询变量之前，查询不会实际执行。

示例 - 方法语法

某些查询操作必须表示为方法调用。 最常见的此类方法是可返回单一数值的方法，例如 Sum、Max、Min、Average 等。 这些方法在任何查询中都必须始终最后一个调用，因为它们返回单个值，不能用作额外查询操作的源。 下面的示例演示查询表达式中的方法调用：

C#

```
List<int> numbers1 = [5, 4, 1, 3, 9, 8, 6, 7, 2, 0];
List<int> numbers2 = [15, 14, 11, 13, 19, 18, 16, 17, 12, 10];

// Query #4.
double average = numbers1.Average();

// Query #5.
IEnumerable<int> concatenationQuery = numbers1.Concat(numbers2);
```

如果方法具有 System.Action 或 System.Func<TResult> 参数，则这些参数以 lambda 表达式的形式提供，如下面的示例所示：

C#

```
// Query #6.
IEnumerable<int> largeNumbersQuery = numbers2.Where(c => c > 15);
```

在前面的查询中，只有查询 #4 立即执行，因为它返回单个值，而不是泛型 IEnumerable<T> 集合。 方法本身使用 `foreach` 或类似的代码来计算其值。

上面的每个查询可以通过 `var`` 使用隐式类型化进行编写，如下面的示例所示：

C#

```
// var is used for convenience in these queries
double average = numbers1.Average();
```

```
var concatenationQuery = numbers1.Concat(numbers2);
var largeNumbersQuery = numbers2.Where(c => c > 15);
```

示例 - 混合查询和方法语法

此示例演示如何对查询子句的结果使用方法语法。 只需将查询表达式括在括号中，然后应用点运算符并调用方法。 在下面的示例中，查询 #7 返回对值介于 3 与 7 之间的数字进行的计数。 但是通常情况下，最好使用另一个变量存储方法调用的结果。 采用此方法时，查询不太可能与查询的结果相混淆。

C#

```
// Query #7.

// Using a query expression with method syntax
var numCount1 = (
    from num in numbers1
    where num is > 3 and < 7
    select num
).Count();

// Better: Create a new variable to store
// the method call result
IEnumerable<int> numbersQuery =
    from num in numbers1
    where num is > 3 and < 7
    select num;

var numCount2 = numbersQuery.Count();
```

由于查询 #7 返回单个值而不是集合，因此查询立即执行。

前面的查询可以通过 `var` 使用隐式类型化进行编写，如下所示：

C#

```
var numCount = (from num in numbers...
```

它可以采用方法语法进行编写，如下所示：

C#

```
var numCount = numbers.Count(n => n is > 3 and < 7);
```

它可以使用显式类型化进行编写，如下所示：

C#

```
int numCount = numbers.Count(n => n is > 3 and < 7);
```

在运行时动态指定谓词筛选器

在某些情况下，在运行时之前你不知道必须将多少个谓词应用于 `where` 子句中的源元素。 动态指定多个谓词筛选器的方法之一是使用 Contains 方法，如以下示例中所示。 查询将根据执行查询时的 `id` 值返回不同的结果。

C#

```
int[] ids = [111, 114, 112];

var queryNames =
    from student in students
    where ids.Contains(student.ID)
    select new
    {
        student.LastName,
        student.ID
    };

foreach (var name in queryNames)
{
    Console.WriteLine($"{name.LastName}: {name.ID}");
}

/* Output:
    Garcia: 114
    O'Donnell: 112
    Omelchenko: 111
 */

// Change the ids.
ids = [122, 117, 120, 115];

// The query will now return different results
foreach (var name in queryNames)
{
    Console.WriteLine($"{name.LastName}: {name.ID}");
}

/* Output:
    Adams: 120
    Feng: 117
    Garcia: 115
    Tucker: 122
 */
```

可以使用控制流语句（如 `if... else` 或 `switch`）在预确定的替代查询之间进行选择。在下面的示例中，`studentQuery` 使用其他 `where` 子句，如果 `oddYear` 的运行时值为 `true` 或 `false`。

C#

```
void FilterByYearType(bool oddYear)
{
    IEnumerable<Student> studentQuery = oddYear
        ? (from student in students
           where student.Year is GradeLevel.FirstYear or
GradeLevel.ThirdYear
           select student)
        : (from student in students
           where student.Year is GradeLevel.SecondYear or
GradeLevel.FourthYear
           select student);
    var descr = oddYear ? "odd" : "even";
    Console.WriteLine($"The following students are at an {descr} year
level:");
    foreach (Student name in studentQuery)
    {
        Console.WriteLine($"{name.LastName}: {name.ID}");
    }
}

FilterByYearType(true);

/* Output:
    The following students are at an odd year level:
    Fakhouri: 116
    Feng: 117
    Garcia: 115
    Mortensen: 113
    Tucker: 119
    Tucker: 122
 */

FilterByYearType(false);

/* Output:
    The following students are at an even year level:
    Adams: 120
    Garcia: 114
    Garcia: 118
    O'Donnell: 112
    Omelchenko: 111
    Zabokritski: 121
 */
```

在查询表达式中处理 null 值

此示例显示如何在源集合中处理可能的 null 值。 IEnumerable<T> 等对象集合可包含值为 null 的元素。 如果源集合为 `null` 或包含值为 `null` 的元素，并且查询不处理 `null` 值，则在执行查询时将引发 NullReferenceException。

可采用防御方式进行编码，以避免空引用异常，如以下示例所示：

C#

```
var query1 =
    from c in categories
    where c != null
    join p in products on c.ID equals p?.CategoryID
    select new
    {
        Category = c.Name,
        Name = p.Name
    };
```

在前面的示例中，`where` 子句筛选出类别序列中的所有 null 元素。 此方法独立于 join 子句中的 null 检查。 在此示例中，带有 null 的条件表达式有效，因为 `Products.CategoryID` 的类型为 `int?`，这是 `Nullable<int>` 的速记形式。

在 join 子句中，如果只有一个比较键是可以为 null 的值类型，则可以在查询表达式中将另一个比较键转换为可以为 null 的值类型。 在以下示例中，假定 `EmployeeID` 是包含 `int?` 类型的值列：

C#

```
var query =
    from o in db.Orders
    join e in db.Employees
        on o.EmployeeID equals (int?)e.EmployeeID
    select new { o.OrderID, e.FirstName };
```

每个示例中都使用 `equals` 查询关键字。 还可使用模式匹配，其中包括 `is null` 和 `is not null` 的模式。 不建议在 LINQ 查询中使用这些模式，因为查询提供程序可能无法正确解读新的 C# 语法。 查询提供程序是一个库，用于将 C# 查询表达式转换为本机数据格式，例如 Entity Framework Core。 查询提供程序实现 System.Linq.IQueryProvider 接口，以创建实现 System.Linq.IQueryable<T> 接口的数据源。

在查询表达式中处理异常

在查询表达式的上下文中可以调用任何方法。 请勿在查询表达式中调用任何会产生副作用（如修改数据源内容或引发异常）的方法。 此示例演示在查询表达式中调用方法时如何避免引发异常，而不违反有关异常处理的常规 .NET 指南。 这些指南阐明，当你理解在给定上下文中为何会引发异常时，捕获到该特定异常是可以接受的。 有关详细信息，请参阅异常的最佳做法。

最后的示例演示了在执行查询期间必须引发异常时，该如何处理这种情况。

以下示例演示如何将异常处理代码移到查询表达式外。 只有当方法不取决于查询的任何本地变量时，才可以执行重构。 在查询表达式之外处理异常更容易。

C#

```
// A data source that is very likely to throw an exception!
IEnumerable<int> GetData() => throw new InvalidOperationException();

// DO THIS with a datasource that might
// throw an exception.
IEnumerable<int>? dataSource = null;
try
{
    dataSource = GetData();
}
catch (InvalidOperationException)
{
    Console.WriteLine("Invalid operation");
}

if (dataSource is not null)
{
    // If we get here, it is safe to proceed.
    var query =
        from i in dataSource
        select i * i;

    foreach (var i in query)
    {
        Console.WriteLine(i.ToString());
    }
}
```

在上述示例的 `catch (InvalidOperationException)` 块中，请以适合你的应用程序的方式处理（或不处理）异常。

在某些情况下，针对由查询内部引发的异常的最佳措施可能是立即停止执行查询。 下面的示例演示如何处理可能在查询正文内部引发的异常。 假定 `SomeMethodThatMightThrow` 可能导致要求停止执行查询的异常。

`try` 块封装 `foreach` 循环，且不对自身进行查询。 `foreach` 循环是实际执行查询时的点。 执行查询时，会引发运行时异常。 因此，必须在 `foreach` 循环中处理这些异常。

C#

```
// Not very useful as a general purpose method.
string SomeMethodThatMightThrow(string s) =>
    s[4] == 'C' ?
        throw new InvalidOperationException() :
        @"C:\newFolder\" + s;

// Data source.
string[] files = ["fileA.txt", "fileB.txt", "fileC.txt"];

// Demonstration query that throws.
var exceptionDemoQuery =
    from file in files
    let n = SomeMethodThatMightThrow(file)
    select n;

try
{
    foreach (var item in exceptionDemoQuery)
    {
        Console.WriteLine($"Processing {item}");
    }
}
catch (InvalidOperationException e)
{
    Console.WriteLine(e.Message);
}

/* Output:
    Processing C:\newFolder\fileA.txt
    Processing C:\newFolder\fileB.txt
    Operation is not valid due to the current state of the object.
 */
```

请记得捕获预期引发的任何异常，并且/或者在 `finally` 块中执行任何必要的清理。

另请参阅

- 演练：用 C# 编写查询
- where 子句
- 基于运行时状态进行查询
- Nullable<T>
- 可以为 null 的值类型

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

LINQ 查询操作中的类型关系 (C#)

项目 • 2023/12/18

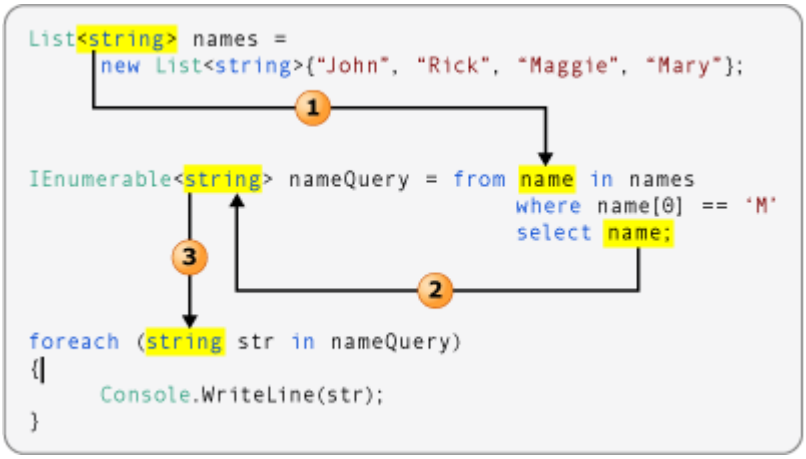
若要有效编写查询，应了解完整的查询操作中的变量类型是如何全部彼此关联的。 如果了解这些关系，就能够更容易地理解文档中的 LINQ 示例和代码示例。 另外，还能了解在使用 var 隐式对变量进行类型化时的操作。

LINQ 查询操作在数据源、查询本身及查询执行中是强类型的。 查询中变量的类型必须与数据源中元素的类型和 foreach 语句中迭代变量的类型兼容。 此强类型保证在编译时捕获类型错误，以便可以在用户遇到这些错误之前更正它们。

为了演示这些类型关系，下面的大多数示例对所有变量使用显式类型。 最后一个示例演示在利用使用 `var` 的隐式类型时，如何应用相同的原则。

不转换源数据的查询

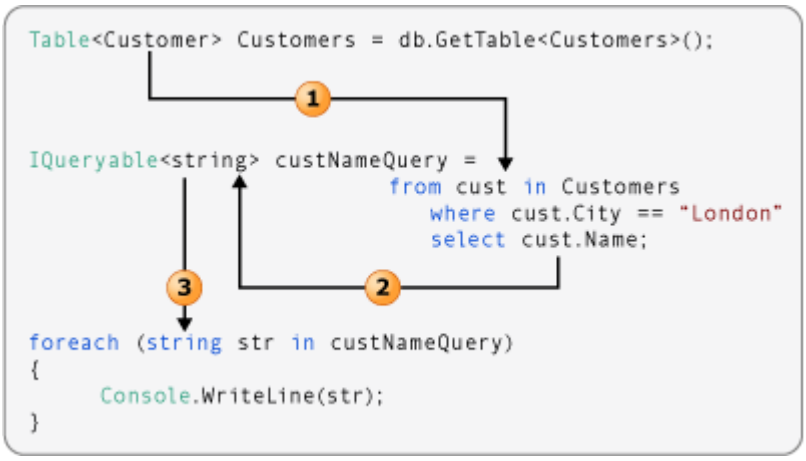
下图演示不对数据执行转换的 LINQ to Objects 查询操作。 源包含一个字符串序列，查询输出也是一个字符串序列。



1. 数据源的类型参数决定范围变量的类型。
2. 所选对象的类型决定查询变量的类型。 此处的 `name` 是一个字符串。 因此，查询变量是一个 `IEnumerable<string>`。
3. 在 `foreach` 语句中循环访问查询变量。 因为查询变量是一个字符串序列，所以迭代变量也是一个字符串。

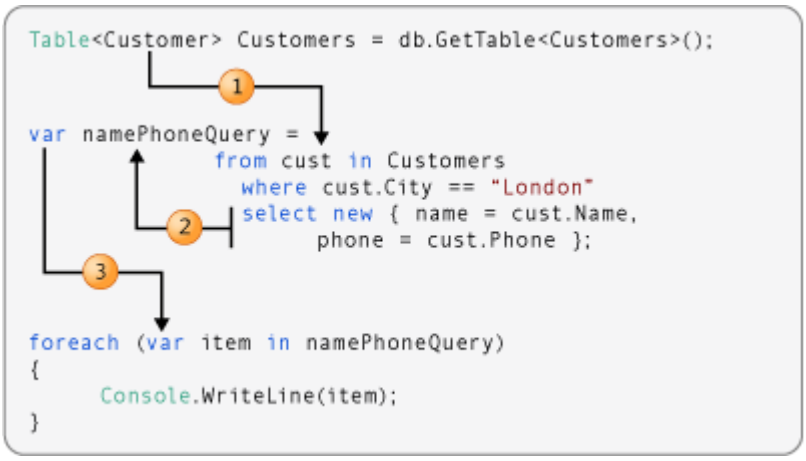
转换源数据的查询

下图演示对数据执行简单转换的 LINQ to SQL 查询操作。 查询将一个 `Customer` 对象序列用作输入，并只选择结果中的 `Name` 属性。 因为 `Name` 是一个字符串，所以查询生成一个字符串序列作为输出。



1. 数据源的类型参数决定范围变量的类型。
2. `select` 语句返回 `Name` 属性，而非完整的 `Customer` 对象。 因为 `Name` 是一个字符串，所以 `custNameQuery` 的类型参数是 `string`，而非 `Customer`。
3. 因为 `custNameQuery` 是一个字符串序列，所以 `foreach` 循环的迭代变量也必须是 `string`。

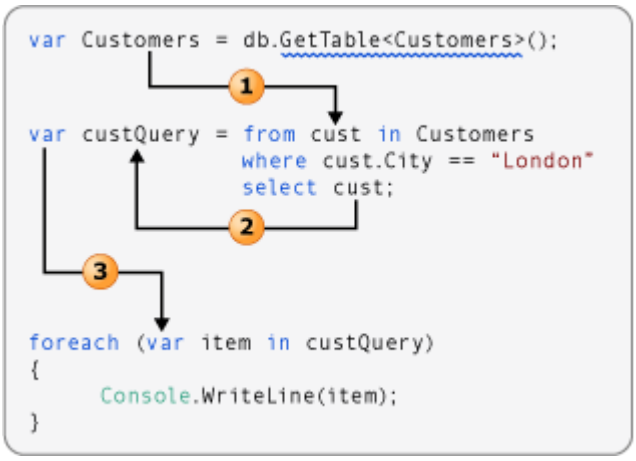
下图演示稍微复杂的转换。 `select` 语句返回只捕获原始 `Customer` 对象的两个成员的匿名类型。



1. 数据源的类型参数始终为查询中范围变量的类型。
2. 因为 `select` 语句生成匿名类型，所以必须使用 `var` 隐式类型化查询变量。
3. 因为查询变量的类型是隐式的，所以 `foreach` 循环中的迭代变量也必须是隐式的。

让编译器推断类型信息

虽然需要了解查询操作中的类型关系，但是也可以选择让编译器执行全部工作。 关键字 var 可用于查询操作中的任何本地变量。 下图与前面讨论的第二个示例相似。 但是，编译器为查询操作中的各个变量提供强类型。



LINQ 和泛型类型 (C#)

LINQ 查询基于泛型类型。 无需深入了解泛型即可开始编写查询。 但是，可能需要了解 2 个基本概念：

1. 创建泛型集合类（如 List<T>）的实例时，需将“T”替换为列表将包含的对象类型。 例如，字符串列表表示为 `List<string>`，`Customer` 对象列表表示为 `List<Customer>`。 泛型列表属于强类型，与将其元素存储为 Object 的集合相比，泛型列表具备更多优势。 如果尝试将 `Customer` 添加到 `List<string>`，则会在编译时收到错误。 泛型集合易于使用的原因是不必执行运行时类型转换。
2. IEnumerable<T> 是一个接口，通过该接口，可以使用 `foreach` 语句来枚举泛型集合类。 泛型集合类支持 IEnumerable<T>，正如非泛型集合类（如 ArrayList）支持 IEnumerable。

有关泛型的详细信息，请参阅泛型。

LINQ 查询中的 IEnumerable<T> 变量

LINQ 查询变量被类型化为 IEnumerable<T> 或派生类型（如 IQueryable<T>）。 看到类型化为 `IEnumerable<Customer>` 的查询变量时，这只意味着执行查询时，该查询将生成包含零个或多个 `Customer` 对象的序列。

C#

```
IEnumerable<Customer> customerQuery =
    from cust in customers
    where cust.City == "London"
    select cust;

foreach (Customer customer in customerQuery)
{
```

```
    Console.WriteLine($"{customer.LastName}, {customer.FirstName}");
}
```

让编译器处理泛型类型声明

如果愿意，可以使用 var 关键字来避免使用泛型语法。 `var` 关键字指示编译器通过查看在 `from` 子句中指定的数据源来推断查询变量的类型。 以下示例生成与上例相同的编译代码：

C#

```
var customerQuery2 =
    from cust in customers
    where cust.City == "London"
    select cust;

foreach(var customer in customerQuery2)
{
    Console.WriteLine($"{customer.LastName}, {customer.FirstName}");
}
```

变量的类型明显或显式指定嵌套泛型类型（如由组查询生成的那些类型）并不重要时，`var` 关键字很有用。 通常，我们建议如果使用 `var`，应意识到这可能使他人更难以理解代码。 有关详细信息，请参阅隐式类型局部变量。

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

支持 LINQ 的 C# 功能

项目 • 2024/04/27

查询表达式

查询表达式使用类似于 SQL 或 XQuery 的声明性语法来查询 System.Collections.Generic.IEnumerable<T> 集合。 在编译时，查询语法转换为对 LINQ 提供程序的标准查询方法实现的方法调用。 应用程序通过使用 using 指令指定适当的命名空间来控制范围内的标准查询运算符。 下面的查询表达式获取一个字符串数组，按字符串中的第一个字符对字符串进行分组，然后对各组进行排序。

C#

```
var query = from str in stringArray
            group str by str[0] into stringGroup
            orderby stringGroup.Key
            select stringGroup;
```

隐式类型化变量 (var)

可以使用 var 修饰符来指示编译器推断和分配类型，如下所示：

C#

```
var number = 5;
var name = "Virginia";
var query = from str in stringArray
            where str[0] == 'm'
            select str;
```

声明为 `var` 的变量与显式指定其类型的变量一样都是强类型。 通过使用 `var`，可以创建匿名类型，但只能用于本地变量。 有关详细信息，请参阅隐式类型局部变量。

对象和集合初始值设定项

通过对象和集合初始值设定项，初始化对象时无需为对象显式调用构造函数。 初始值设定项通常用在将源数据投影到新数据类型的查询表达式中。 假定一个类名为 `Customer`，具有公共 `Name` 和 `Phone` 属性，可以按下列代码中所示使用对象初始值设定项：

C#

```
var cust = new Customer { Name = "Mike", Phone = "555-1212" };
```

继续 `Customer` 类，假设有一个名为 `IncomingOrders` 的数据源，并且每个订单具有一个较大的 `OrderSize`，你希望基于该订单创建新的 `Customer`。 可以在此数据源上执行 LINQ 查询，并使用对象初始化来填充集合：

C#

```
var newLargeOrderCustomers = from o in IncomingOrders
                             where o.OrderSize > 5
                             select new Customer { Name = o.Name, Phone =
o.Phone };
```

数据源可能具有比 `Customer` 类（例如 `OrderSize`）定义的更多的属性，但执行对象初始化后，从查询返回的数据被定型为所需的数据类型；选择与你的类相关的数据。 因此，你现在有填充了想要的多个新 `Customer` 的 System.Collections.Generic.IEnumerable<T>。 前面的示例还可以使用 LINQ 的方法语法编写：

C#

```
var newLargeOrderCustomers = IncomingOrders.Where(x => x.OrderSize >
5).Select(y => new Customer { Name = y.Name, Phone = y.Phone });
```

从 C# 12 开始，可以使用集合表达式来初始化集合。

有关详细信息，请参阅：

- 对象和集合初始值设定项
- 标准查询运算符的查询表达式语法

匿名类型

编译器构造匿名类型。 该类型名称仅适用于编译器。 匿名类型提供一种在查询结果中对一组属性临时分组的简便方法，无需定义单独的命名类型。 使用新的表达式和对象初始值设定项初始化匿名类型，如下所示：

C#

```
select new {name = cust.Name, phone = cust.Phone};
```

从 C# 7 开始，可以使用元组创建未命名的类型。

扩展方法

扩展方法是一种可与类型关联的静态方法，因此可以像实例方法那样对类型调用它。 实际上，利用此功能，可以将新方法"添加"到现有类型，而不会实际修改它们。 标准查询运算符是一组扩展方法，它们为实现 IEnumerable<T> 的任何类型提供 LINQ 查询功能。

Lambda 表达式

Lambda 表达式是一种内联函数，该函数使用 `=>` 运算符将输入参数与函数体分离，并且可以在编译时转换为委托或表达式树。 在 LINQ 编程中，在对标准查询运算符进行直接方法调用时，会遇到 lambda 表达式。

作为数据的表达式

查询对象可编写，这意味着你可以从方法中返回查询。 表示查询的对象不会存储生成的集合，但会根据需要存储生成结果的步骤。 从方法中返回查询对象的好处是可以进一步编写或修改这些对象。 因此，返回查询的方法的任何返回值或 `out` 输出参数也必须具有该类型。 如果某个方法可将查询具体化为具体的 List<T> 或 Array 类型，则它会返回查询结果，而不是查询本身。 仍然能够编写或修改从方法中返回的查询变量。

在下面的示例中，第一个方法 `QueryMethod1` 返回了一个查询作为返回值，第二个方法 `QueryMethod2` 返回了一个查询作为 `out` 参数（在本例中为 `returnQ`）。 在这两种情况下，返回的都是查询，而不是查询结果。

C#

```
IEnumerable<string> QueryMethod1(int[] ints) =>
    from i in ints
    where i > 4
    select i.ToString();

void QueryMethod2(int[] ints, out IEnumerable<string> returnQ) =>
    returnQ =
        from i in ints
        where i < 4
        select i.ToString();

int[] nums = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9];

var myQuery1 = QueryMethod1(nums);
```

查询 `myQuery1` 在以下 foreach 循环中执行。

C#

```
foreach (var s in myQuery1)
{
    Console.WriteLine(s);
}
```

将鼠标指针悬停在 `myQuery1` 上方以查看其类型。

还可以直接执行从 `QueryMethod1` 返回的查询，而无需使用 `myQuery1`。

C#

```
foreach (var s in QueryMethod1(nums))
{
    Console.WriteLine(s);
}
```

将鼠标指针悬停在对 `QueryMethod1` 的调用上以查看其返回类型。

`QueryMethod2` 会返回一个查询作为其 `out` 参数的值：

C#

```
QueryMethod2(nums, out IEnumerable<string> myQuery2);

// Execute the returned query.
foreach (var s in myQuery2)
{
    Console.WriteLine(s);
}
```

可以使用查询组合来修改查询。 在这种情况下，前一个查询对象用于创建新的查询对象。 此新对象会返回与原始查询对象不同的结果。

C#

```
myQuery1 =
    from item in myQuery1
    orderby item descending
    select item;

// Execute the modified query.
Console.WriteLine("\nResults of executing modified myQuery1:");
foreach (var s in myQuery1)
{
    Console.WriteLine(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 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

Tutorial: Write queries in C# using language integrated query (LINQ)

Article • 04/25/2024

In this tutorial, you create a data source and write several LINQ queries. You can experiment with the query expressions and see the differences in the results. This walkthrough demonstrates the C# language features that are used to write LINQ query expressions. You can follow along and build the app and experiment with the queries yourself. This article assumes you've installed the latest .NET SDK. If not, go to the .NET Downloads page and install the latest version on your machine.

First, create the application. From the console, type the following command:

.NET CLI

```
dotnet new console -o WalkthroughWritingLinqQueries
```

Or, if you prefer Visual Studio, create a new console application named *WalkthroughWritingLinqQueries*.

Create an in-memory data source

The first step is to create a data source for your queries. The data source for the queries is a simple list of `Student` records. Each `Student` record has a first name, family name, and an array of integers that represents their test scores in the class. Add a new file named *students.cs*, and copy the following code into that file:

C#

```
namespace WalkthroughWritingLinqQueries;

public record Student(string First, string Last, int ID, int[] Scores);
```

Note the following characteristics:

- The `Student` record consists of automatically implemented properties.
- Each student in the list is initialized with the primary constructor.
- The sequence of scores for each student is initialized with a primary constructor.

Next, create a sequence of `Student` records that serves as the source of this query. Open *Program.cs*, and remove the following boilerplate code:

C#

```
// See https://aka.ms/new-console-template for more information
Console.WriteLine("Hello, World!");
```

Replace it with the following code that creates a sequence of `Student` records:

C#

```
using WalkthroughWritingLinqQueries;

// Create a data source by using a collection initializer.
IEnumerable<Student> students =
[
    new Student(First: "Svetlana", Last: "Omelchenko", ID: 111, Scores: [97,
92, 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: [88,
94, 65, 91]),
    new Student(First: "Cesar",    Last: "Garcia",     ID: 114, Scores: [97,
89, 85, 82]),
    new Student(First: "Debra",    Last: "Garcia",     ID: 115, Scores: [35,
72, 91, 70]),
    new Student(First: "Fadi",     Last: "Fakhouri",   ID: 116, Scores: [99,
86, 90, 94]),
    new Student(First: "Hanying",  Last: "Feng",       ID: 117, Scores: [93,
92, 80, 87]),
    new Student(First: "Hugo",     Last: "Garcia",     ID: 118, Scores: [92,
90, 83, 78]),

    new Student("Lance",   "Tucker",      119, [68, 79, 88, 92]),
    new Student("Terry",   "Adams",       120, [99, 82, 81, 79]),
    new Student("Eugene",  "Zabokritski", 121, [96, 85, 91, 60]),
    new Student("Michael", "Tucker",      122, [94, 92, 91, 91])
];
```

- The sequence of students is initialized with a collection expression.
- The `Student` record type holds the static list of all students.
- Some of the constructor calls use named arguments to clarify which argument matches which constructor parameter.

Try adding a few more students with different test scores to the list of students to get more familiar with the code so far.

Create the query

Next, you create your first query. Your query, when you execute it, produces a list of all students whose score on the first test was greater than 90. Because the whole `Student` object is selected, the type of the query is `IEnumerable<Student>`. Although the code could also use implicit typing by using the var keyword, explicit typing is used to clearly illustrate results. (For more information about `var`, see Implicitly Typed Local Variables.) Add the following code to *Program.cs*, after the code that creates the sequence of students:

C#

```
// Create the query.
// The first line could also be written as "var studentQuery ="
IEnumerable<Student> studentQuery =
    from student in students
    where student.Scores[0] > 90
    select student;
```

The query's range variable, `student`, serves as a reference to each `Student` in the source, providing member access for each object.

Run the query

Now write the `foreach` loop that causes the query to execute. Each element in the returned sequence is accessed through the iteration variable in the `foreach` loop. The type of this variable is `Student`, and the type of the query variable is compatible, `IEnumerable<Student>`. After you added the following code, build and run the application to see the results in the **Console** window.

C#

```
// Execute the query.
// var could be used here also.
foreach (Student student in studentQuery)
{
    Console.WriteLine($"{student.Last}, {student.First}");
}

// Output:
// Omelchenko, Svetlana
// Garcia, Cesar
// Fakhouri, Fadi
// Feng, Hanying
// Garcia, Hugo
// Adams, Terry
```

```
// Zabokritski, Eugene
// Tucker, Michael
```

To further refine the query, you can combine multiple Boolean conditions in the `where` clause. The following code adds a condition so that the query returns those students whose first score was over 90 and whose last score was less than 80. The `where` clause should resemble the following code.

C#

```
where student.Scores[0] > 90 && student.Scores[3] < 80
```

Try the preceding `where` clause, or experiment yourself with other filter conditions. For more information, see where clause.

Order the query results

It's easier to scan the results if they are in some kind of order. You can order the returned sequence by any accessible field in the source elements. For example, the following `orderby` clause orders the results in alphabetical order from A to Z according to the family name of each student. Add the following `orderby` clause to your query, right after the `where` statement and before the `select` statement:

C#

```
orderby student.Last ascending
```

Now change the `orderby` clause so that it orders the results in reverse order according to the score on the first test, from the highest score to the lowest score.

C#

```
orderby student.Scores[0] descending
```

Change the `WriteLine` format string so that you can see the scores:

C#

```
Console.WriteLine($"{student.Last}, {student.First} {student.Scores[0]}");
```

For more information, see orderby clause.

Group the results

Grouping is a powerful capability in query expressions. A query with a group clause produces a sequence of groups, and each group itself contains a `Key` and a sequence that consists of all the members of that group. The following new query groups the students by using the first letter of their family name as the key.

C#

```
IEnumerable<IGrouping<char, Student>> studentQuery =
    from student in students
    group student by student.Last[0];
```

The type of the query changed. It now produces a sequence of groups that have a `char` type as a key, and a sequence of `Student` objects. The code in the `foreach` execution loop also must change:

C#

```
foreach (IGrouping<char, Student> studentGroup in studentQuery)
{
    Console.WriteLine(studentGroup.Key);
    foreach (Student student in studentGroup)
    {
        Console.WriteLine($"   {student.Last}, {student.First}");
    }
}
// Output:
// O
//   Omelchenko, Svetlana
//   O'Donnell, Claire
// M
//   Mortensen, Sven
// G
//   Garcia, Cesar
//   Garcia, Debra
//   Garcia, Hugo
// F
//   Fakhouri, Fadi
//   Feng, Hanying
// T
//   Tucker, Lance
//   Tucker, Michael
// A
//   Adams, Terry
// Z
//   Zabokritski, Eugene
```

Run the application and view the results in the **Console** window. For more information, see group clause.

Explicitly coding `IEnumerables` of `IGroupings` can quickly become tedious. Write the same query and `foreach` loop much more conveniently by using `var`. The `var` keyword doesn't change the types of your objects; it just instructs the compiler to infer the types. Change the type of `studentQuery` and the iteration variable `group` to `var` and rerun the query. In the inner `foreach` loop, the iteration variable is still typed as `Student`, and the query works as before. Change the `student` iteration variable to `var` and run the query again. You see that you get exactly the same results.

C#

```
IEnumerable<IGrouping<char, Student>> studentQuery =
    from student in students
    group student by student.Last[0];

foreach (IGrouping<char, Student> studentGroup in studentQuery)
{
    Console.WriteLine(studentGroup.Key);
    foreach (Student student in studentGroup)
    {
        Console.WriteLine($"   {student.Last}, {student.First}");
    }
}
```

For more information about `var`, see Implicitly Typed Local Variables.

Order the groups by their key value

The groups in the previous query aren't in alphabetical order. You can provide an `orderby` clause after the `group` clause. But to use an `orderby` clause, you first need an identifier that serves as a reference to the groups created by the `group` clause. You provide the identifier by using the `into` keyword, as follows:

C#

```
var studentQuery4 =
    from student in students
    group student by student.Last[0] into studentGroup
    orderby studentGroup.Key
    select studentGroup;

foreach (var groupOfStudents in studentQuery4)
{
    Console.WriteLine(groupOfStudents.Key);
```

```
    foreach (var student in groupOfStudents)
    {
        Console.WriteLine($"   {student.Last}, {student.First}");
    }
}

// Output:
//A
//   Adams, Terry
//F
//   Fakhouri, Fadi
//   Feng, Hanying
//G
//   Garcia, Cesar
//   Garcia, Debra
//   Garcia, Hugo
//M
//   Mortensen, Sven
//O
//   Omelchenko, Svetlana
//   O'Donnell, Claire
//T
//   Tucker, Lance
//   Tucker, Michael
//Z
//   Zabokritski, Eugene
```

Run this query, and the groups are now sorted in alphabetical order.

You can use the `let` keyword to introduce an identifier for any expression result in the query expression. This identifier can be a convenience, as in the following example. It can also enhance performance by storing the results of an expression so that it doesn't have to be calculated multiple times.

C#

```
// This query returns those students whose
// first test score was higher than their
// average score.
var studentQuery5 =
    from student in students
    let totalScore = student.Scores[0] + student.Scores[1] +
        student.Scores[2] + student.Scores[3]
    where totalScore / 4 < student.Scores[0]
    select $"{student.Last}, {student.First}";

foreach (string s in studentQuery5)
{
    Console.WriteLine(s);
}

// Output:
```

```
// Omelchenko, Svetlana
// O'Donnell, Claire
// Mortensen, Sven
// Garcia, Cesar
// Fakhouri, Fadi
// Feng, Hanying
// Garcia, Hugo
// Adams, Terry
// Zabokritski, Eugene
// Tucker, Michael
```

For more information, see the article on the let clause.

Use method syntax in a query expression

As described in Query Syntax and Method Syntax in LINQ, some query operations can only be expressed by using method syntax. The following code calculates the total score for each `Student` in the source sequence, and then calls the `Average()` method on the results of that query to calculate the average score of the class.

C#

```
var studentQuery =
    from student in students
    let totalScore = student.Scores[0] + student.Scores[1] +
        student.Scores[2] + student.Scores[3]
    select totalScore;

double averageScore = studentQuery.Average();
Console.WriteLine("Class average score = {0}", averageScore);

// Output:
// Class average score = 334.166666666667
```

To transform or project in the select clause

It's common for a query to produce a sequence whose elements differ from the elements in the source sequences. Delete or comment out your previous query and execution loop, and replace it with the following code. The query returns a sequence of strings (not `Students`), and this fact is reflected in the `foreach` loop.

C#

```
IEnumerable<string> studentQuery =
    from student in students
    where student.Last == "Garcia"
```

```
    select student.First;

Console.WriteLine("The Garcias in the class are:");
foreach (string s in studentQuery)
{
    Console.WriteLine(s);
}

// Output:
// The Garcias in the class are:
// Cesar
// Debra
// Hugo
```

Code earlier in this walkthrough indicated that the average class score is approximately 334. To produce a sequence of `Students` whose total score is greater than the class average, together with their `Student ID`, you can use an anonymous type in the `select` statement:

C#

```
var aboveAverageQuery =
    from student in students
    let x = student.Scores[0] + student.Scores[1] +
        student.Scores[2] + student.Scores[3]
    where x > averageScore
    select new { id = student.ID, score = x };

foreach (var item in aboveAverageQuery)
{
    Console.WriteLine("Student ID: {0}, Score: {1}", item.id, item.score);
}

// Output:
// Student ID: 113, Score: 338
// Student ID: 114, Score: 353
// Student ID: 116, Score: 369
// Student ID: 117, Score: 352
// Student ID: 118, Score: 343
// Student ID: 120, Score: 341
// Student ID: 122, Score: 368
```

Standard Query Operators Overview

Article • 05/31/2024

The *standard query operators* are the keywords and methods that form the LINQ pattern. The C# language defines LINQ query keywords that you use for the most common query expression. The compiler translates expressions using these keywords to the equivalent method calls. The two forms are synonymous. Other methods that are part of the System.Linq namespace don't have equivalent query keywords. In those cases, you must use the method syntax. This section covers all the query operator keywords. The runtime and other NuGet packages add more methods designed to work with LINQ queries each release. The most common methods, including those that have query keyword equivalents are covered in this section. For the full list of query methods supported by the .NET Runtime, see the System.Linq.Enumerable API documentation. In addition to the methods covered here, this class contains methods for concatenating data sources, computing a single value from a data source, such as a sum, average, or other value.

> **ⓘ Important**
>
> These samples use an **System.Collections.Generic.IEnumerable<T>** data source. Data sources based on **System.Linq.IQueryProvider** use **System.Linq.IQueryable<T>** data sources and **expression trees**. Expression trees have **limitations** on the allowed C# syntax. Furthermore, each `IQueryProvider` data source, such as **EF Core** may impose more restrictions. Check the documentation for your data source.

Most of these methods operate on sequences, where a sequence is an object whose type implements the IEnumerable<T> interface or the IQueryable<T> interface. The standard query operators provide query capabilities including filtering, projection, aggregation, sorting and more. The methods that make up each set are static members of the Enumerable and Queryable classes, respectively. They're defined as *extension methods* of the type that they operate on.

The distinction between IEnumerable<T> and IQueryable<T> sequences determines how the query is executed at runtime.

For `IEnumerable<T>`, the returned enumerable object captures the arguments that were passed to the method. When that object is enumerated, the logic of the query operator is employed and the query results are returned.

For `IQueryable<T>`, the query is translated into an expression tree. The expression tree can be translated to a native query when the data source can optimize the query. Libraries such as Entity Framework translate LINQ queries into native SQL queries that execute at the database.

The following code example demonstrates how the standard query operators can be used to obtain information about a sequence.

C#

```
string sentence = "the quick brown fox jumps over the lazy dog";
// Split the string into individual words to create a collection.
string[] words = sentence.Split(' ');

// Using query expression syntax.
var query = from word in words
            group word.ToUpper() by word.Length into gr
            orderby gr.Key
            select new { Length = gr.Key, Words = gr };

// Using method-based query syntax.
var query2 = words.
    GroupBy(w => w.Length, w => w.ToUpper()).
    Select(g => new { Length = g.Key, Words = g }).
    OrderBy(o => o.Length);

foreach (var obj in query)
{
    Console.WriteLine("Words of length {0}:", obj.Length);
    foreach (string word in obj.Words)
        Console.WriteLine(word);
}

// This code example produces the following output:
//
// Words of length 3:
// THE
// FOX
// THE
// DOG
// Words of length 4:
// OVER
// LAZY
// Words of length 5:
// QUICK
// BROWN
// JUMPS
```

Where possible, the queries in this section use a sequence of words or numbers as the input source. For queries where more complicated relationships between objects are used, the following sources that model a school are used:

C#

```
public enum GradeLevel
{
    FirstYear = 1,
    SecondYear,
    ThirdYear,
    FourthYear
};

public class Student
{
    public required string FirstName { get; init; }
    public required string LastName { get; init; }
    public required int ID { get; init; }

    public required GradeLevel Year { get; init; }
    public required List<int> Scores { get; init; }

    public required int DepartmentID { get; init; }
}

public class Teacher
{
    public required string First { get; init; }
    public required string Last { get; init; }
    public required int ID { get; init; }
    public required string City { get; init; }
}

public class Department
{
    public required string Name { get; init; }
    public int ID { get; init; }

    public required int TeacherID { get; init; }
}
```

Each `Student` has a grade level, a primary department, and a series of scores. A `Teacher` also has a `City` property that identifies the campus where the teacher holds classes. A `Department` has a name, and a reference to a `Teacher` who serves as the department head.

You can find the data set in the source repo.

Types of query operators

The standard query operators differ in the timing of their execution, depending on whether they return a singleton value or a sequence of values. Those methods that

return a singleton value (such as Average and Sum) execute immediately. Methods that return a sequence defer the query execution and return an enumerable object. You can use the output sequence of one query as the input sequence to another query. Calls to query methods can be chained together in one query, which enables queries to become arbitrarily complex.

Query operators

In a LINQ query, the first step is to specify the data source. In a LINQ query, the `from` clause comes first in order to introduce the data source (`students`) and the *range variable* (`student`).

C#

```
//queryAllStudents is an IEnumerable<Student>
var queryAllStudents = from student in students
                       select student;
```

The range variable is like the iteration variable in a `foreach` loop except that no actual iteration occurs in a query expression. When the query is executed, the range variable serves as a reference to each successive element in `students`. Because the compiler can infer the type of `student`, you don't have to specify it explicitly. You can introduce more range variables in a `let` clause. For more information, see let clause.

> **Note**
>
> For non-generic data sources such as ArrayList, the range variable must be explicitly typed. For more information, see How to query an ArrayList with LINQ (C#) and from clause.

Once you obtain a data source, you can perform any number of operations on that data source:

- Filter data using the `where` keyword.
- Order data using the `orderby` and optionally `descending` keywords.
- Group data using the `group` and optionally `into` keywords.
- Join data using the `join` keyword.
- Project data using the `select` keyword.

Query Expression Syntax Table

The following table lists the standard query operators that have equivalent query expression clauses.

Expand table

Method	C# query expression syntax
Cast	Use an explicitly typed range variable: `from int i in numbers` (For more information, see from clause.)

GroupBy	`group … by`
	
	-or-
	
	`group … by … into …`
	
	(For more information, see group clause.)

GroupJoin<TOuter,TInner,TKey,TResult>(IEnumerable<TOuter>, IEnumerable<TInner>, Func<TOuter,TKey>, Func<TInner,TKey>, Func<TOuter,IEnumerable<TInner>, TResult>)	`join … in … on … equals … into …` (For more information, see join clause.)
Join<TOuter,TInner,TKey,TResult>(IEnumerable<TOuter>, IEnumerable<TInner>, Func<TOuter,TKey>, Func<TInner,TKey>, Func<TOuter,TInner,TResult>)	`join … in … on … equals …` (For more information, see join clause.)
OrderBy<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>)	`orderby` (For more information, see orderby clause.)

Method	C# query expression syntax
OrderByDescending<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>)	`orderby … descending` (For more information, see orderby clause.)
Select	`select` (For more information, see select clause.)
SelectMany	Multiple `from` clauses. (For more information, see from clause.)
ThenBy<TSource,TKey>(IOrderedEnumerable<TSource>, Func<TSource,TKey>)	`orderby …, …` (For more information, see orderby clause.)
ThenByDescending<TSource,TKey>(IOrderedEnumerable<TSource>, Func<TSource,TKey>)	`orderby …, … descending` (For more information, see orderby clause.)
Where	`where` (For more information, see where clause.)

Data Transformations with LINQ

Language-Integrated Query (LINQ) isn't only about retrieving data. It's also a powerful tool for transforming data. By using a LINQ query, you can use a source sequence as input and modify it in many ways to create a new output sequence. You can modify the sequence itself without modifying the elements themselves by sorting and grouping.

But perhaps the most powerful feature of LINQ queries is the ability to create new types. The select clause creates an output element from an input element. You use it to transform an input element into an output element:

- Merge multiple input sequences into a single output sequence that has a new type.
- Create output sequences whose elements consist of only one or several properties of each element in the source sequence.
- Create output sequences whose elements consist of the results of operations performed on the source data.
- Create output sequences in a different format. For example, you can transform data from SQL rows or text files into XML.

These transformations can be combined in various ways in the same query. Furthermore, the output sequence of one query can be used as the input sequence for a new query. The following example transforms objects in an in-memory data structure into XML elements.

C#

```
// Create the query.
var studentsToXML = new XElement("Root",
    from student in students
    let scores = string.Join(",", student.Scores)
    select new XElement("student",
                new XElement("First", student.FirstName),
                new XElement("Last", student.LastName),
                new XElement("Scores", scores)
            ) // end "student"
        ); // end "Root"

// Execute the query.
Console.WriteLine(studentsToXML);
```

The code produces the following XML output:

XML

```
<Root>
  <student>
    <First>Svetlana</First>
    <Last>Omelchenko</Last>
    <Scores>97,90,73,54</Scores>
  </student>
  <student>
    <First>Claire</First>
    <Last>O'Donnell</Last>
```

```
    <Scores>56,78,95,95</Scores>
  </student>
  ...
  <student>
    <First>Max</First>
    <Last>Lindgren</Last>
    <Scores>86,88,96,63</Scores>
  </student>
  <student>
    <First>Arina</First>
    <Last>Ivanova</Last>
    <Scores>93,63,70,80</Scores>
  </student>
</Root>
```

For more information, see Creating XML Trees in C# (LINQ to XML).

You can use the results of one query as the data source for a subsequent query. This example shows how to order the results of a join operation. This query creates a group join, and then sorts the groups based on the category element, which is still in scope. Inside the anonymous type initializer, a subquery orders all the matching elements from the products sequence.

C#

```
var orderedQuery = from department in departments
                   join student in students on department.ID equals
student.DepartmentID into studentGroup
                   orderby department.Name
                   select new
                   {
                       DepartmentName = department.Name,
                       Students = from student in studentGroup
                                  orderby student.LastName
                                    select student
                   };

foreach (var departmentList in orderedQuery)
{
    Console.WriteLine(departmentList.DepartmentName);
    foreach (var student in departmentList.Students)
    {
        Console.WriteLine($"  {student.LastName,-10}
{student.FirstName,-10}");
    }
}
/* Output:
Chemistry
  Balzan     Josephine
  Fakhouri   Fadi
  Popov      Innocenty
  Seleznyova Sofiya
```

```
  Vella      Carmen
Economics
  Adams      Terry
  Adaobi     Izuchukwu
  Berggren   Jeanette
  Garcia     Cesar
  Ifeoma     Nwanneka
  Jamuike    Ifeanacho
  Larsson    Naima
  Svensson   Noel
  Ugomma     Ifunanya
Engineering
  Axelsson   Erik
  Berg       Veronika
  Engström   Nancy
  Hicks      Cassie
  Keever     Bruce
  Micallef   Nicholas
  Mortensen  Sven
  Nilsson    Erna
  Tucker     Michael
  Yermolayeva Anna
English
  Andersson  Sarah
  Feng       Hanying
  Ivanova    Arina
  Jakobsson  Jesper
  Jensen     Christiane
  Johansson  Mark
  Kolpakova  Nadezhda
  Omelchenko Svetlana
  Urquhart   Donald
Mathematics
  Frost      Gaby
  Garcia     Hugo
  Hedlund    Anna
  Kovaleva   Katerina
  Lindgren   Max
  Maslova    Evgeniya
  Olsson     Ruth
  Sammut     Maria
  Sazonova   Anastasiya
Physics
  Åkesson    Sami
  Edwards    Amy E.
  Falzon     John
  Garcia     Debra
  Hansson    Sanna
  Mattsson   Martina
  Richardson Don
  Zabokritski Eugene
*/
```

The equivalent query using method syntax is shown in the following code:

C#

```
var orderedQuery = departments
    .GroupJoin(students, department => department.ID, student =>
student.DepartmentID,
    (department, studentGroup) => new
    {
        DepartmentName = department.Name,
        Students = studentGroup.OrderBy(student => student.LastName)
    })
    .OrderBy(department => department.DepartmentName);


foreach (var departmentList in orderedQuery)
{
    Console.WriteLine(departmentList.DepartmentName);
    foreach (var student in departmentList.Students)
    {
        Console.WriteLine($"  {student.LastName,-10}
{student.FirstName,-10}");
    }
}
```

Although you can use an `orderby` clause with one or more of the source sequences before the join, generally we don't recommend it. Some LINQ providers might not preserve that ordering after the join. For more information, see join clause.

See also

- Enumerable
- Queryable
- select clause
- Extension Methods
- Query Keywords (LINQ)
- Anonymous Types

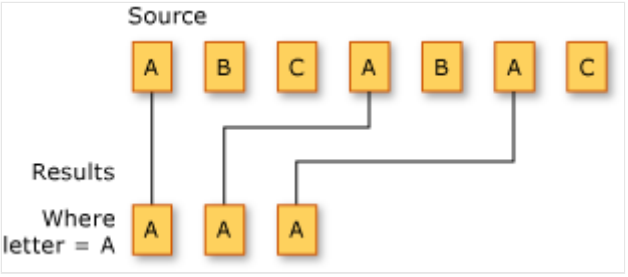
使用 LINQ 筛选 C# 中的数据

项目 • 2024/06/06

筛选是指将结果集限制为仅包含满足指定条件的元素的操作。 它也称为“选择”与指定条件匹配的元素。

> **ⓘ 重要**
>
> 这些示例使用 System.Collections.Generic.IEnumerable<T> 数据源。 基于 System.Linq.IQueryProvider 的数据源使用 System.Linq.IQueryable<T> 数据源和**表达式树**。 表达式树对允许的 C# 语法有**限制**。 此外，每个 `IQueryProvider` 数据源（如 EF Core）可能会施加更多限制。 查看数据源的文档。

下图演示了对字符序列进行筛选的结果。 筛选操作的谓词指定字符必须为“A”。



下表中列出了执行选择的标准查询运算符方法：

展开表

方法名	描述	C# 查询表达式语法	详细信息
OfType	根据其转换为特定类型的能力选择值。	不适用。	Enumerable.OfType Queryable.OfType
Where	选择基于谓词函数的值。	`where`	Enumerable.Where Queryable.Where

以下示例使用 `where` 子句从数组中筛选具有特定长度的字符串。

C#

```
string[] words = ["the", "quick", "brown", "fox", "jumps"];

IEnumerable<string> query = from word in words
                            where word.Length == 3
```

```
                          select word;

foreach (string str in query)
{
    Console.WriteLine(str);
}

/* This code produces the following output:

    the
    fox
*/
```

以下代码显示了使用方法语法的等效查询：

C#

```
string[] words = ["the", "quick", "brown", "fox", "jumps"];

IEnumerable<string> query =
    words.Where(word => word.Length == 3);

foreach (string str in query)
{
    Console.WriteLine(str);
}

/* This code produces the following output:

    the
    fox
*/
```

另请参阅

- System.Linq
- where 子句
- 如何使用反射查询程序集的元数据 (LINQ) (C#)
- 如何查询具有指定特性或名称的文件 (C#)
- 如何按任意词或字段对文本数据进行排序或筛选 (LINQ) (C#)

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

投影运算 (C#)

项目 • 2024/06/06

投影是指将对象转换为一种新形式的操作，该形式通常只包含那些将随后使用的属性。 通过使用投影，您可以构造从每个对象生成的新类型。 可以投影属性，并对该属性执行数学函数。 还可以在不更改原始对象的情况下投影该对象。

> **ⓘ 重要**
>
> 这些示例使用 System.Collections.Generic.IEnumerable<T> 数据源。 基于 System.Linq.IQueryProvider 的数据源使用 System.Linq.IQueryable<T> 数据源和**表达式树**。 表达式树对允许的 C# 语法有**限制**。 此外，每个 `IQueryProvider` 数据源（如 EF Core）可能会施加更多限制。 查看数据源的文档。

下面一节列出了执行投影的标准查询运算符方法。

方法

展开表

方法名称	说明	C# 查询表达式语法	详细信息
选择	投影基于转换函数的值。	`select`	Enumerable.Select Queryable.Select
SelectMany	投影基于转换函数的值序列，然后将它们展平为一个序列。	使用多个 `from` 子句	Enumerable.SelectMany Queryable.SelectMany
Zip	使用 2-3 个指定序列中的元素生成元组序列。	不适用。	Enumerable.Zip Queryable.Zip

`Select`

下面的示例使用 `select` 子句来投影字符串列表中每个字符串的第一个字母。

C#

```
List<string> words = ["an", "apple", "a", "day"];

var query = from word in words
            select word.Substring(0, 1);
```

```
foreach (string s in query)
{
    Console.WriteLine(s);
}

/* This code produces the following output:

    a
    a
    a
    d
*/
```

以下代码显示了使用方法语法的等效查询：

C#

```
List<string> words = ["an", "apple", "a", "day"];

var query = words.Select(word => word.Substring(0, 1));

foreach (string s in query)
{
    Console.WriteLine(s);
}

/* This code produces the following output:

    a
    a
    a
    d
*/
```

SelectMany

下面的示例使用多个 `from` 子句来投影字符串列表中每个字符串中的每个单词。

C#

```
List<string> phrases = ["an apple a day", "the quick brown fox"];

var query = from phrase in phrases
            from word in phrase.Split(' ')
            select word;

foreach (string s in query)
{
    Console.WriteLine(s);
```

```
}

/* This code produces the following output:

    an
    apple
    a
    day
    the
    quick
    brown
    fox
*/
```

以下代码显示了使用方法语法的等效查询：

C#

```
List<string> phrases = ["an apple a day", "the quick brown fox"];

var query = phrases.SelectMany(phrases => phrases.Split(' '));

foreach (string s in query)
{
    Console.WriteLine(s);
}

/* This code produces the following output:

    an
    apple
    a
    day
    the
    quick
    brown
    fox
*/
```

该方法 `SelectMany` 还可以形成匹配第一个序列中的每个项与第二个序列中的每个项的组合：

C#

```
var query = from number in numbers
            from letter in letters
            select (number, letter);

foreach (var item in query)
{
```

```
    Console.WriteLine(item);
}
```

以下代码显示了使用方法语法的等效查询：

C#

```
var method = numbers
    .SelectMany(number => letters,
    (number, letter) => (number, letter));

foreach (var item in method)
{
    Console.WriteLine(item);
}
```

Zip

`Zip` 投影运算符有多个重载。 所有 `Zip` 方法都处理两个或更多可能是异构类型的序列。 前两个重载返回元组，具有来自给定序列的相应位置类型。

请考虑下列集合：

C#

```
// An int array with 7 elements.
IEnumerable<int> numbers = [1, 2, 3, 4, 5, 6, 7];
// A char array with 6 elements.
IEnumerable<char> letters = ['A', 'B', 'C', 'D', 'E', 'F'];
```

若要将这些序列一起投影，请使用 Enumerable.Zip<TFirst,TSecond>(IEnumerable<TFirst>, IEnumerable<TSecond>) 运算符：

C#

```
foreach ((int number, char letter) in numbers.Zip(letters))
{
    Console.WriteLine($"Number: {number} zipped with letter: '{letter}'");
}
// This code produces the following output:
//     Number: 1 zipped with letter: 'A'
//     Number: 2 zipped with letter: 'B'
//     Number: 3 zipped with letter: 'C'
//     Number: 4 zipped with letter: 'D'
//     Number: 5 zipped with letter: 'E'
//     Number: 6 zipped with letter: 'F'
```

> **ⓘ 重要**
>
> zip 操作生成的序列的长度永远不会长于最短序列。 `numbers` 和 `letters` 集合的长度不同，生成的序列将省略 `numbers` 集合中的最后一个元素，因为它没有任何要压缩的内容。

第二个重载接受 `third` 序列。 让我们创建另一个集合，即 `emoji`：

C#

```
// A string array with 8 elements.
IEnumerable<string> emoji = [ "😳", "🔥", "🎉", "👀", "⭐", "💜", "✓",
"💯"];
```

若要将这些序列一起投影，请使用 Enumerable.Zip<TFirst,TSecond,TThird>(IEnumerable<TFirst>, IEnumerable<TSecond>, IEnumerable<TThird>) 运算符：

C#

```
foreach ((int number, char letter, string em) in numbers.Zip(letters,
emoji))
{
    Console.WriteLine(
        $"Number: {number} is zipped with letter: '{letter}' and emoji:
{em}");
}
// This code produces the following output:
//     Number: 1 is zipped with letter: 'A' and emoji: 😳
//     Number: 2 is zipped with letter: 'B' and emoji: 🔥
//     Number: 3 is zipped with letter: 'C' and emoji: 🎉
//     Number: 4 is zipped with letter: 'D' and emoji: 👀
//     Number: 5 is zipped with letter: 'E' and emoji: ⭐
//     Number: 6 is zipped with letter: 'F' and emoji: 💜
```

与前面的重载非常相似，`Zip` 方法投影一个元组，但这次包含三个元素。

第三个重载接受用作结果选择器的 `Func<TFirst, TSecond, TResult>` 参数。 可以从压缩的序列中投影新的生成序列。

C#

```
foreach (string result in
    numbers.Zip(letters, (number, letter) => $"{number} = {letter}
({(int)letter})"))
{
    Console.WriteLine(result);
}
```

```
// This code produces the following output:
//     1 = A (65)
//     2 = B (66)
//     3 = C (67)
//     4 = D (68)
//     5 = E (69)
//     6 = F (70)
```

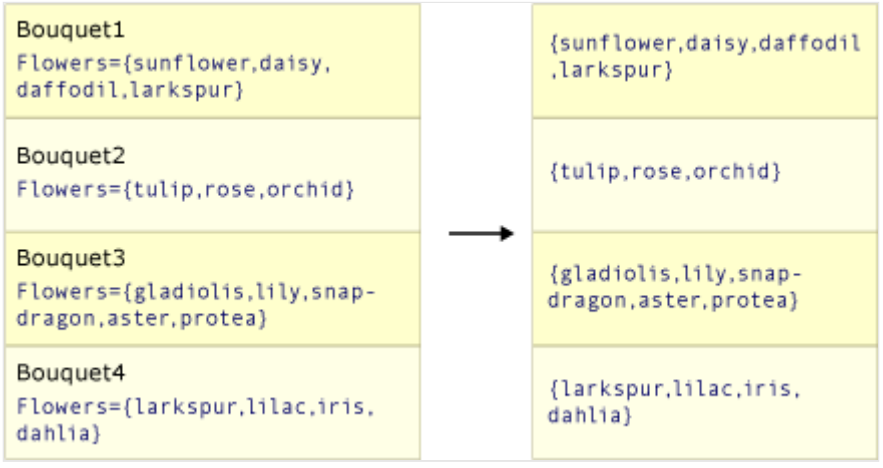
使用前面的 `Zip` 重载，指定的函数应用于相应的元素 `numbers` 和 `letter`，生成 `string` 结果的序列。

`Select` 与 `SelectMany`

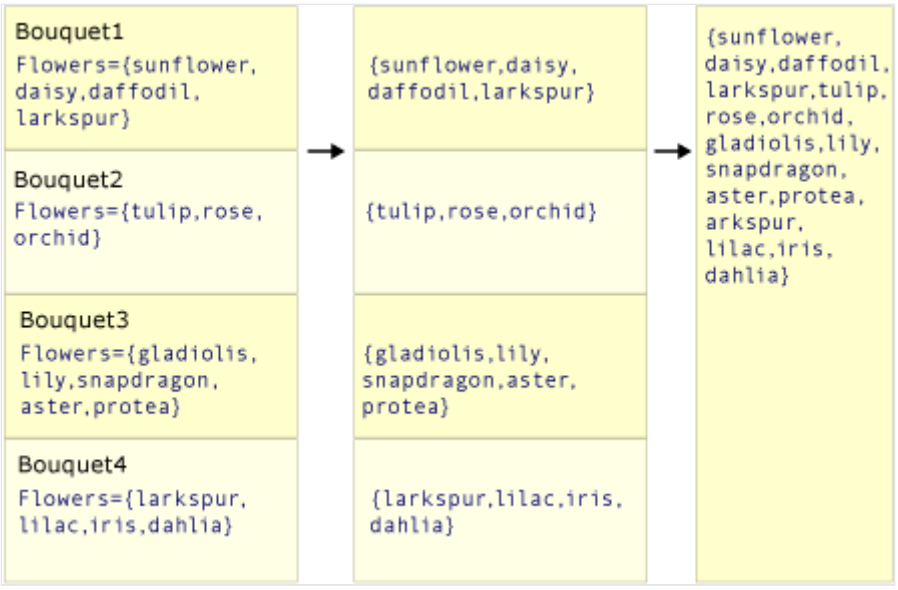
`Select` 和 `SelectMany` 的工作都是依据源值生成一个或多个结果值。 `Select` 为每个源值生成一个结果值。 因此，总体结果是一个与源集合具有相同元素数目的集合。 与之相反，`SelectMany` 生成单个总体结果，其中包含来自每个源值的串联子集合。 作为参数传递到 `SelectMany` 的转换函数必须为每个源值返回一个可枚举值序列。 `SelectMany` 连接这些可枚举序列，以创建一个大的序列。

下面两个插图演示了这两个方法的操作之间的概念性区别。 在每种情况下，假定选择器（转换）函数从每个源值中选择一个由花卉数据组成的数组。

下图描述 `Select` 如何返回一个与源集合具有相同元素数目的集合。



下图描述 `SelectMany` 如何将中间数组序列串联为一个最终结果值，其中包含每个中间数组中的每个值。



代码示例

下面的示例比较 `Select` 和 `SelectMany` 的行为。 代码通过从源集合的每个花卉名称列表中提取项来创建一个“花束”。 在以下示例中，转换函数 Select<TSource,TResult>(IEnumerable<TSource>, Func<TSource,TResult>) 使用的“单值”是值的集合。 此示例需要额外的 `foreach` 循环，以便枚举每个子序列中的每个字符串。

C#

```
class Bouquet
{
    public required List<string> Flowers { get; init; }
}

static void SelectVsSelectMany()
{
    List<Bouquet> bouquets =
    [
        new Bouquet { Flowers = ["sunflower", "daisy", "daffodil",
"larkspur"] },
        new Bouquet { Flowers = ["tulip", "rose", "orchid"] },
        new Bouquet { Flowers = ["gladiolis", "lily", "snapdragon", "aster",
"protea"] },
        new Bouquet { Flowers = ["larkspur", "lilac", "iris", "dahlia"] }
    ];

    IEnumerable<List<string>> query1 = bouquets.Select(bq => bq.Flowers);

    IEnumerable<string> query2 = bouquets.SelectMany(bq => bq.Flowers);

    Console.WriteLine("Results by using Select():");
    // Note the extra foreach loop here.
    foreach (IEnumerable<string> collection in query1)
    {
        foreach (string item in collection)
```

```
        {
            Console.WriteLine(item);
        }
    }

    Console.WriteLine("\nResults by using SelectMany():");
    foreach (string item in query2)
    {
        Console.WriteLine(item);
    }
}
```

另请参阅

- System.Linq
- select 子句
- 如何从多个源填充对象集合 (LINQ) (C#)
- 如何使用组将一个文件拆分成多个文件 (LINQ) (C#)

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

设置操作 (C#)

项目 • 2024/10/31

LINQ 中的集运算是指根据相同或单独集合中是否存在等效元素来生成结果集的查询运算。

> **重要**
>
> 这些示例使用 System.Collections.Generic.IEnumerable<T> 数据源。 基于 System.Linq.IQueryProvider 的数据源使用 System.Linq.IQueryable<T> 数据源和**表达式树**。 表达式树对允许的 C# 语法有**限制**。 此外，每个 `IQueryProvider` 数据源（如 EF Core）可能会施加更多限制。 查看数据源的文档。

展开表

方法名称	说明	C# 查询表达式语法	详细信息
`Distinct` 或 `DistinctBy`	删除集合中的重复值。	不适用。	Enumerable.Distinct Enumerable.DistinctBy Queryable.Distinct Queryable.DistinctBy
`Except` 或 `ExceptBy`	返回差集，差集指位于一个集合但不位于另一个集合的元素。	不适用。	Enumerable.Except Enumerable.ExceptBy Queryable.Except Queryable.ExceptBy
`Intersect` 或 `IntersectBy`	返回交集，交集指同时出现在两个集合中的元素。	不适用。	Enumerable.Intersect Enumerable.IntersectBy Queryable.Intersect Queryable.IntersectBy
`Union` 或 `UnionBy`	返回并集，并集指位于两个集合中任一集合的唯一的元素。	不适用。	Enumerable.Union Enumerable.UnionBy Queryable.Union Queryable.UnionBy

`Distinct` 和 `DistinctBy`

以下示例演示字符串序列上 Enumerable.Distinct 方法的行为。 返回的序列包含输入序列的唯一元素。



C#

```
string[] words = ["the", "quick", "brown", "fox", "jumped", "over", "the",
"lazy", "dog"];

IEnumerable<string> query = from word in words.Distinct()
                            select word;

foreach (var str in query)
{
    Console.WriteLine(str);
}

/* This code produces the following output:
 *
 * the
 * quick
 * brown
 * fox
 * jumped
 * over
 * lazy
 * dog
 */
```

DistinctBy 是 `Distinct` 的替代方法，它采用 `keySelector`。 `keySelector` 用作源类型的比较鉴别器。 在以下代码中，单词根据其 `Length` 进行区分，并且显示每个长度的第一个单词：

C#

```
string[] words = ["the", "quick", "brown", "fox", "jumped", "over", "the",
"lazy", "dog"];

foreach (string word in words.DistinctBy(p => p.Length))
{
    Console.WriteLine(word);
}

// This code produces the following output:
//     the
//     quick
//     jumped
//     over
```

`Except` 和 `ExceptBy`

以下示例演示 Enumerable.Except 的行为。 返回的序列只包含位于第一个输入序列但不位于第二个输入序列的元素。



> **① 备注**
>
> 本文中的以下示例使用此区域的常见数据源。
> 每个 `Student` 都有年级、主要院系和一系列分数。 `Teacher` 还有一个 `City` 属性，用于标识教师的授课校区。 `Department` 有一个名称，以及对担任院系主任的 `Teacher` 的引用。
> 可以在源存储库**中找到**示例数据集。

C#

```
public enum GradeLevel
{
    FirstYear = 1,
    SecondYear,
    ThirdYear,
    FourthYear
};

public class Student
{
    public required string FirstName { get; init; }
    public required string LastName { get; init; }
    public required int ID { get; init; }

    public required GradeLevel Year { get; init; }
    public required List<int> Scores { get; init; }

    public required int DepartmentID { get; init; }
}

public class Teacher
{
    public required string First { get; init; }
    public required string Last { get; init; }
    public required int ID { get; init; }
    public required string City { get; init; }
}

public class Department
{
    public required string Name { get; init; }
```

```
    public int ID { get; init; }

    public required int TeacherID { get; init; }
}
```

C#

```
string[] words1 = ["the", "quick", "brown", "fox"];
string[] words2 = ["jumped", "over", "the", "lazy", "dog"];

IEnumerable<string> query = from word in words1.Except(words2)
                            select word;

foreach (var str in query)
{
    Console.WriteLine(str);
}

/* This code produces the following output:
 *
 * quick
 * brown
 * fox
 */
```

ExceptBy 方法是 `Except` 的替代方法，它采用可能是异构类型的两个序列和一个 `keySelector`。 `keySelector` 的类型与第一个集合的类型相同。 请考虑以下要排除的 `Teacher` 数组和教师 ID。 若要查找第一个集合中没有出现在第二个集合中的教师，可以将教师的 ID 投影到第二个集合中：

C#

```
int[] teachersToExclude =
[
    901,    // English
    965,    // Mathematics
    932,    // Engineering
    945,    // Economics
    987,    // Physics
    901     // Chemistry
];

foreach (Teacher teacher in
    teachers.ExceptBy(
        teachersToExclude, teacher => teacher.ID))
{
    Console.WriteLine($"{teacher.First} {teacher.Last}");
}
```

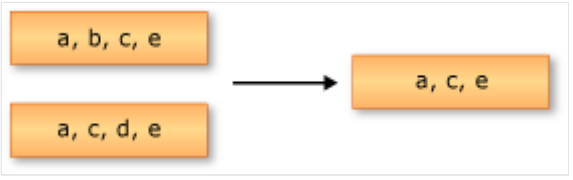
在前述 C# 代码中：

- 已筛选 `teachers` 数组以仅包含不在 `teachersToExclude` 数组中的那些教师。
- `teachersToExclude` 数组包含所有部门负责人的 `ID` 值。
- 对 `ExceptBy` 的调用会生成一个新的值集，这些值被写入到控制台。

新的值集的类型为 `Teacher`，这是第一个集合的类型。 对于 `teachers` 数组中的每个 `teacher`，如果在 `teachersToExclude` 数组中没有相应 ID 值，则会写入到控制台。

`Intersect` 和 `IntersectBy`

以下示例演示 Enumerable.Intersect 的行为。 返回的序列包含两个输入序列共有的元素。



C#

```
string[] words1 = ["the", "quick", "brown", "fox"];
string[] words2 = ["jumped", "over", "the", "lazy", "dog"];

IEnumerable<string> query = from word in words1.Intersect(words2)
                            select word;

foreach (var str in query)
{
    Console.WriteLine(str);
}

/* This code produces the following output:
 *
 * the
 */
```

IntersectBy 方法是 `Intersect` 的替代方法，它采用可能是异构类型的两个序列和一个 `keySelector`。 `keySelector` 用作第二个集合类型的比较鉴别器。 请考虑以下学生和教师数组。 查询按名称匹配每个序列中的项，以查找那些也是教师的学生：

C#

```
foreach (Student person in
    students.IntersectBy(
        teachers.Select(t => (t.First, t.Last)), s => (s.FirstName,
s.LastName)))
```

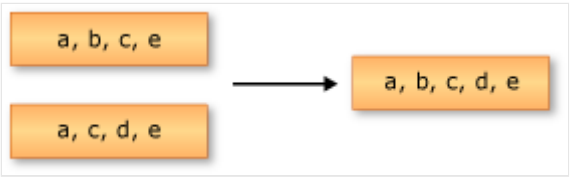
```
{
    Console.WriteLine($"{person.FirstName} {person.LastName}");
}
```

在前述 C# 代码中：

- 该查询通过比较名称生成 `Teacher` 和 `Student` 的交集。
- 只有在这两个阵列中都找到的人员才会出现在结果序列中。
- 将生成的 `Student` 实例写入控制台。

`Union` 和 `UnionBy`

以下示例演示对两个字符串序列执行的联合操作。 返回的序列包含两个输入序列的唯一元素。



C#

```
string[] words1 = ["the", "quick", "brown", "fox"];
string[] words2 = ["jumped", "over", "the", "lazy", "dog"];

IEnumerable<string> query = from word in words1.Union(words2)
                            select word;

foreach (var str in query)
{
    Console.WriteLine(str);
}

/* This code produces the following output:
 *
 * the
 * quick
 * brown
 * fox
 * jumped
 * over
 * lazy
 * dog
*/
```

UnionBy 方法是 `Union` 的替代方法，它采用相同类型的两个序列和一个 `keySelector`。 `keySelector` 用作源类型的比较鉴别器。 以下查询生成所有人员（学生或教师）的列

表。 同时也是教师的学生只会被添加到并集中一次：

```C#
foreach (var person in
    students.Select(s => (s.FirstName, s.LastName)).UnionBy(
        teachers.Select(t => (FirstName: t.First, LastName: t.Last)), s =>
(s.FirstName, s.LastName)))
{
    Console.WriteLine($"{person.FirstName} {person.LastName}");
}
```

在前述 C# 代码中：

- `teachers` 和 `students` 数组使用其名称作为键选择器编织在一起。
- 生成的名称将写入控制台。

另请参阅

- System.Linq
- 如何查找两个列表之间的差集 (LINQ) (C#)

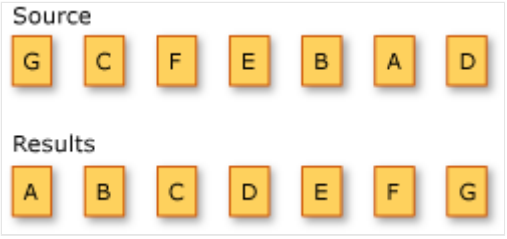
对数据排序 (C#)

项目 • 2024/06/06

排序操作基于一个或多个属性对序列的元素进行排序。 第一个排序条件对元素执行主要排序。 通过指定第二个排序条件，您可以对每个主要排序组内的元素进行排序。

> **ⓘ 重要**
>
> 这些示例使用 System.Collections.Generic.IEnumerable<T> 数据源。 基于 System.Linq.IQueryProvider 的数据源使用 System.Linq.IQueryable<T> 数据源和**表达式树**。 表达式树对允许的 C# 语法有**限制**。 此外，每个 `IQueryProvider` 数据源（如 EF Core）可能会施加更多限制。 查看数据源的文档。

下图展示了对一系列字符执行按字母顺序排序操作的结果。



下节列出了对数据进行排序的标准查询运算符方法。

方法

展开表

方法名	描述	C# 查询表达式语法	更多信息
OrderBy	按升序对值排序。	`orderby`	Enumerable.OrderBy Queryable.OrderBy
OrderByDescending	按降序对值排序。	`orderby … descending`	Enumerable.OrderByDescending Queryable.OrderByDescending
ThenBy	按升序执行次要排序。	`orderby …, …`	Enumerable.ThenBy Queryable.ThenBy
ThenByDescending	按降序执行次要排序。	`orderby …, … descending`	Enumerable.ThenByDescending

方法名	描述	C# 查询表达式语法	更多信息
			Queryable.ThenByDescending
Reverse	反转集合中元素的顺序。	不适用。	Enumerable.Reverse Queryable.Reverse

本文中的以下示例使用此区域的常见数据源：

C#

```
public enum GradeLevel
{
    FirstYear = 1,
    SecondYear,
    ThirdYear,
    FourthYear
};

public class Student
{
    public required string FirstName { get; init; }
    public required string LastName { get; init; }
    public required int ID { get; init; }

    public required GradeLevel Year { get; init; }
    public required List<int> Scores { get; init; }

    public required int DepartmentID { get; init; }
}

public class Teacher
{
    public required string First { get; init; }
    public required string Last { get; init; }
    public required int ID { get; init; }
    public required string City { get; init; }
}
public class Department
{
    public required string Name { get; init; }
    public int ID { get; init; }

    public required int TeacherID { get; init; }
}
```

每个 `Student` 都有年级、主要院系和一系列分数。 `Teacher` 还有一个 `City` 属性，用于标识教师的授课校区。 `Department` 具有名称，以及对担任学科主任的 `Teacher` 的引用。

主要升序排序

以下示例演示如何在 LINQ 查询中使用 `orderby` 子句，以按姓氏以升序方式对教师数组进行排序。

C#

```
IEnumerable<string> query = from teacher in teachers
                            orderby teacher.Last
                            select teacher.Last;

foreach (string str in query)
{
    Console.WriteLine(str);
}
```

以下代码显示了使用方法语法编写的等效查询：

C#

```
IEnumerable<string> query = teachers
    .OrderBy(teacher => teacher.Last)
    .Select(teacher => teacher.Last);

foreach (string str in query)
{
    Console.WriteLine(str);
}
```

主要降序排序

下一个示例演示如何在 LINQ 查询中使用 `orderby descending` 子句按姓氏以降序方式对教师进行排序。

C#

```
IEnumerable<string> query = from teacher in teachers
                            orderby teacher.Last descending
                            select teacher.Last;

foreach (string str in query)
{
    Console.WriteLine(str);
}
```

以下代码显示了使用方法语法编写的等效查询：

C#

```
IEnumerable<string> query = teachers
    .OrderByDescending(teacher => teacher.Last)
    .Select(teacher => teacher.Last);

foreach (string str in query)
{
    Console.WriteLine(str);
}
```

次要升序排序

下面的示例演示了如何在 LINQ 查询中使用 `orderby` 子句执行主要排序和次要排序。 教师主要按城市和姓氏排序，两者均按升序排序。

C#

```
IEnumerable<(string, string)> query = from teacher in teachers
                            orderby teacher.City, teacher.Last
                            select (teacher.Last, teacher.City);

foreach ((string last, string city) in query)
{
    Console.WriteLine($"City: {city}, Last Name: {last}");
}
```

以下代码显示了使用方法语法编写的等效查询：

C#

```
IEnumerable<(string, string)> query = teachers
    .OrderBy(teacher => teacher.City)
    .ThenBy(teacher => teacher.Last)
    .Select(teacher => (teacher.Last, teacher.City));

foreach ((string last, string city) in query)
{
    Console.WriteLine($"City: {city}, Last Name: {last}");
}
```

次要降序排序

下面的示例演示如何在 LINQ 查询中使用 `orderby descending` 子句按升序执行主要排序，按降序执行次要排序。 教师主要按城市排序，其次按他们的姓氏排序。

C#

```
IEnumerable<(string, string)> query = from teacher in teachers
                              orderby teacher.City, teacher.Last descending
                              select (teacher.Last, teacher.City);

foreach ((string last, string city) in query)
{
    Console.WriteLine($"City: {city}, Last Name: {last}");
}
```

以下代码显示了使用方法语法编写的等效查询：

C#

```
IEnumerable<(string, string)> query = teachers
    .OrderBy(teacher => teacher.City)
    .ThenByDescending(teacher => teacher.Last)
    .Select(teacher => (teacher.Last, teacher.City));

foreach ((string last, string city) in query)
{
    Console.WriteLine($"City: {city}, Last Name: {last}");
}
```

另请参阅

- System.Linq
- orderby 子句
- 如何按任意词或字段对文本数据进行排序或筛选 (LINQ) (C#)

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

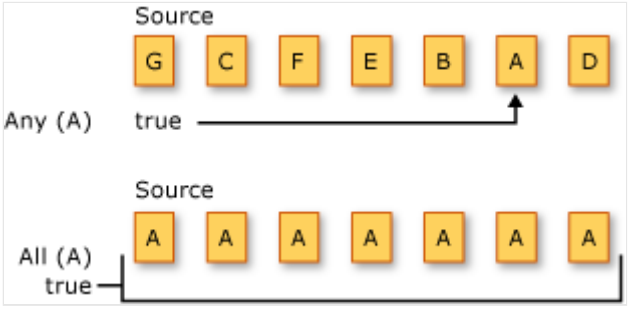
LINQ (C#) 中的限定符运算

项目 • 2024/06/03

限定符运算返回一个 Boolean 值，该值指示序列中是否有一些元素满足条件或是否所有元素都满足条件。

> **ⓘ 重要**
>
> 这些示例使用 System.Collections.Generic.IEnumerable<T> 数据源。 基于 System.Linq.IQueryProvider 的数据源使用 System.Linq.IQueryable<T> 数据源和**表达式树**。 表达式树对允许的 C# 语法有**限制**。 此外，每个 `IQueryProvider` 数据源（如 EF Core）可能会施加更多限制。 查看有个数据源的文档。

下图描述了两个不同源序列上的两个不同限定符运算。 第一个运算询问是否有任何元素为字符“A”。 第二个运算询问是否所有元素都为字符“A”。 这两种方法在此示例中都返回 `true`。



展开表

方法名	描述	C# 查询表达式语法	更多信息
全部	确定是否序列中的所有元素都满足条件。	不适用。	Enumerable.All Queryable.All
任意	确定序列中是否有元素满足条件。	不适用。	Enumerable.Any Queryable.Any
包含	确定序列是否包含指定的元素。	不适用。	Enumerable.Contains Queryable.Contains

全部

以下示例使用 `All` 查找在所有考试中得分均超过 70 的学生。

C#

```
IEnumerable<string> names = from student in students
                            where student.Scores.All(score => score > 70)
                            select $"{student.FirstName} {student.LastName}:
{string.Join(", ", student.Scores.Select(s => s.ToString()))}";

foreach (string name in names)
{
    Console.WriteLine($"{name}");
}

// This code produces the following output:
//
// Cesar Garcia: 71, 86, 77, 97
// Nancy Engström: 75, 73, 78, 83
// Ifunanya Ugomma: 84, 82, 96, 80
```

任意

以下示例使用 `Any` 查找在任何考试中得分超过 95 的学生。

C#

```
IEnumerable<string> names = from student in students
                            where student.Scores.Any(score => score > 95)
                            select $"{student.FirstName} {student.LastName}:
{student.Scores.Max()}";

foreach (string name in names)
{
    Console.WriteLine($"{name}");
}

// This code produces the following output:
//
// Svetlana Omelchenko: 97
// Cesar Garcia: 97
// Debra Garcia: 96
// Ifeanacho Jamuike: 98
// Ifunanya Ugomma: 96
// Michelle Caruana: 97
// Nwanneka Ifeoma: 98
// Martina Mattsson: 96
// Anastasiya Sazonova: 96
// Jesper Jakobsson: 98
// Max Lindgren: 96
```

包含

以下示例使用 `Contains` 查找在某场考试中得分正好为 95 的学生。

C#

```
IEnumerable<string> names = from student in students
                            where student.Scores.Contains(95)
                            select $"{student.FirstName} {student.LastName}:
{string.Join(", ", student.Scores.Select(s => s.ToString()))}";

foreach (string name in names)
{
    Console.WriteLine($"{name}");
}

// This code produces the following output:
//
// Claire O'Donnell: 56, 78, 95, 95
// Donald Urquhart: 92, 90, 95, 57
```

另请参阅

- System.Linq
- 在运行时动态指定谓词筛选器
- 如何查询包含一组指定词语的句子 (LINQ) (C#)

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

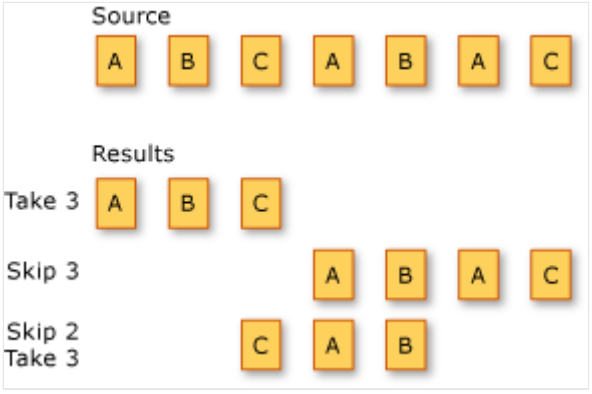
将数据分区 (C#)

项目 • 2024/06/06

LINQ 中的分区是指将输入序列划分为两个部分的操作，无需重新排列元素，然后返回其中一个部分。

> **ⓘ 重要**
>
> 这些示例使用 System.Collections.Generic.IEnumerable<T> 数据源。 基于 System.Linq.IQueryProvider 的数据源使用 System.Linq.IQueryable<T> 数据源和**表达式树**。 表达式树对允许的 C# 语法有**限制**。 此外，每个 `IQueryProvider` 数据源（如 EF Core）可能会施加更多限制。 查看数据源的文档。

下图显示对字符序列进行三种不同的分区操作的结果。 第一个操作返回序列中的前三个元素。 第二个操作跳过前三个元素，返回剩余元素。 第三个操作跳过序列中的前两个元素，返回接下来的三个元素。



下面一节列出了对序列进行分区的标准查询运算符方法。

运算符

展开表

方法名称	说明	C# 查询表达式语法	详细信息
Skip	跳过序列中指定位置之前的元素。	不适用。	Enumerable.Skip Queryable.Skip
SkipWhile	基于谓词函数跳过元素，直到元素不符合条件。	不适用。	Enumerable.SkipWhile Queryable.SkipWhile

方法名称	说明	C# 查询表达式语法	详细信息
Take	获取序列中指定位置之前的元素。	不适用。	Enumerable.Take Queryable.Take
TakeWhile	基于谓词函数获取元素，直到元素不符合条件。	不适用。	Enumerable.TakeWhile Queryable.TakeWhile
Chunk	将序列的元素拆分为指定最大大小的区块。	不适用。	Enumerable.Chunk Queryable.Chunk

以下示例都使用 Enumerable.Range(Int32, Int32) 生成从 0 到 7 的数字序列。

使用 `Take` 方法只获取序列中的第一个元素：

C#

```
foreach (int number in Enumerable.Range(0, 8).Take(3))
{
    Console.WriteLine(number);
}
// This code produces the following output:
// 0
// 1
// 2
```

使用 `Skip` 方法跳过序列中的第一个元素，使用其余元素：

C#

```
foreach (int number in Enumerable.Range(0, 8).Skip(3))
{
    Console.WriteLine(number);
}
// This code produces the following output:
// 3
// 4
// 5
// 6
// 7
```

`TakeWhile` 和 `SkipWhile` 方法也可获取和跳过序列中的元素。 但是，这些方法并不获取固定数量的元素，而是根据条件跳过或获取元素。 `TakeWhile` 会获取序列中的元素，除非元素不符合条件。

C#

```
foreach (int number in Enumerable.Range(0, 8).TakeWhile(n => n < 5))
{
    Console.WriteLine(number);
}
// This code produces the following output:
// 0
// 1
// 2
// 3
// 4
```

只要条件为 true，`SkipWhile` 会跳过第一个元素。 返回不符合条件的第一个元素以及所有后续元素。

C#

```
foreach (int number in Enumerable.Range(0, 8).SkipWhile(n => n < 5))
{
    Console.WriteLine(number);
}
// This code produces the following output:
// 5
// 6
// 7
```

`Chunk` 运算符用于根据给定的 `size` 拆分序列的元素。

C#

```
int chunkNumber = 1;
foreach (int[] chunk in Enumerable.Range(0, 8).Chunk(3))
{
    Console.WriteLine($"Chunk {chunkNumber++}:");
    foreach (int item in chunk)
    {
        Console.WriteLine($"    {item}");
    }

    Console.WriteLine();
}
// This code produces the following output:
// Chunk 1:
//    0
//    1
//    2
//
//Chunk 2:
//    3
//    4
//    5
```

```
//
//Chunk 3:
//    6
//    7
```

上述 C# 代码：

- 依赖于 Enumerable.Range(Int32, Int32) 生成数字序列。
- 应用 `Chunk` 运算符，将序列拆分为最大大小为 3 的块。

另请参阅

- System.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

转换数据类型 (C#)

项目 • 2024/06/06

转换方法可更改输入对象的类型。

> **ⓘ 重要**
>
> 这些示例使用 System.Collections.Generic.IEnumerable<T> 数据源。 基于 System.Linq.IQueryProvider 的数据源使用 System.Linq.IQueryable<T> 数据源和**表达式树**。 表达式树对允许的 C# 语法有**限制**。 此外，每个 `IQueryProvider` 数据源（如 EF Core）可能会施加更多限制。 查看数据源的文档。

LINQ 查询中的转换运算可用于各种应用程序。 以下是一些示例：

- Enumerable.AsEnumerable 方法可用于隐藏类型的标准查询运算符自定义实现。
- Enumerable.OfType 方法可用于为 LINQ 查询启用非参数化集合。
- Enumerable.ToArray、Enumerable.ToDictionary、Enumerable.ToList 和 Enumerable.ToLookup 方法可用于强制执行即时的查询，而不是将其推迟到枚举该查询时。

方法

下表列出了执行数据类型转换的标准查询运算符方法。

本表中名称以“As”开头的转换方法可更改源集合的静态类型，但不对其进行枚举。 名称以“To”开头的方法可枚举源集合，并将项放入相应的集合类型。

展开表

方法名	描述	C# 查询表达式语法	详细信息
AsEnumerable	返回类型化为 IEnumerable<T> 的输入。	不适用。	Enumerable.AsEnumerable
AsQueryable	将（泛型）IEnumerable 转换为（泛型）IQueryable。	不适用。	Queryable.AsQueryable
Cast	将集合中的元素转换为指定类型。	使用显式类型化的范围变量。 例如：	Enumerable.Cast Queryable.Cast

方法名	描述	C# 查询表达式语法	详细信息
		`from string str in words`	
OfType	根据其转换为指定类型的能力筛选值。	不适用。	Enumerable.OfType Queryable.OfType
ToArray	将集合转换为数组。 此方法强制执行查询。	不适用。	Enumerable.ToArray
ToDictionary	根据键选择器函数将元素放入 Dictionary<TKey,TValue>。 此方法强制执行查询。	不适用。	Enumerable.ToDictionary
ToList	将集合转换为 List<T>。 此方法强制执行查询。	不适用。	Enumerable.ToList
ToLookup	根据键选择器函数将元素放入 Lookup<TKey,TElement>（一对多字典）。 此方法强制执行查询。	不适用。	Enumerable.ToLookup

本文中的以下示例使用此区域的常见数据源：

C#

```
public enum GradeLevel
{
    FirstYear = 1,
    SecondYear,
    ThirdYear,
    FourthYear
};

public class Student
{
    public required string FirstName { get; init; }
    public required string LastName { get; init; }
    public required int ID { get; init; }

    public required GradeLevel Year { get; init; }
    public required List<int> Scores { get; init; }

    public required int DepartmentID { get; init; }
}

public class Teacher
{
    public required string First { get; init; }
```

```
    public required string Last { get; init; }
    public required int ID { get; init; }
    public required string City { get; init; }
}
public class Department
{
    public required string Name { get; init; }
    public int ID { get; init; }

    public required int TeacherID { get; init; }
}
```

每个 `Student` 都有年级、主要院系和一系列分数。 `Teacher` 还有一个 `City` 属性，用于标识教师的授课校区。 `Department` 有一个名称，和对担任部门负责人的 `Teacher` 的引用。

查询表达式语法示例

下面的代码示例使用显式类型化的范围变量将类型转换为子类型，然后才访问仅在此子类型上可用的成员。

C#

```
IEnumerable people = students;

var query = from Student student in students
            where student.Year == GradeLevel.ThirdYear
            select student;

foreach (Student student in query)
{
    Console.WriteLine(student.FirstName);
}
```

可以使用方法语法来表示等效查询，如下所示：

C#

```
IEnumerable people = students;

var query = people
    .Cast<Student>()
    .Where(student => student.Year == GradeLevel.ThirdYear);

foreach (Student student in query)
{
    Console.WriteLine(student.FirstName);
}
```

另请参阅

- System.Linq
- from 子句

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 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

LINQ 中的 Join 运算

项目 • 2024/10/15

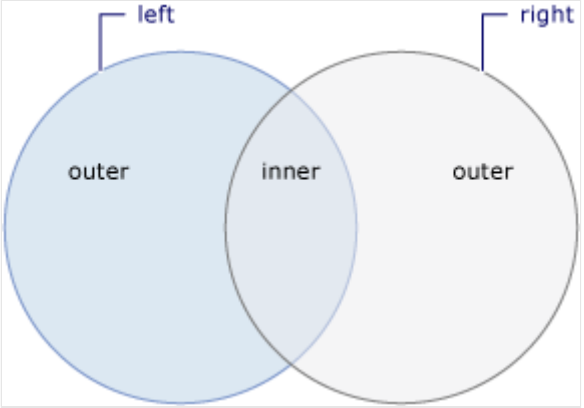
两个数据源的 *join* 就是将一个数据源中的对象与另一个数据源中具有相同公共属性的对象相关联。

> **ⓘ 重要**
>
> 这些示例使用 System.Collections.Generic.IEnumerable<T> 数据源。 基于 System.Linq.IQueryProvider 的数据源使用 System.Linq.IQueryable<T> 数据源和**表达式树**。 表达式树对允许的 C# 语法有**限制**。 此外，每个 `IQueryProvider` 数据源（如 EF Core）可能会施加更多限制。 查看数据源的文档。

当查询所面向的数据源相互之间具有无法直接领会的关系时，联接就成为一项重要的运算。 在面向对象的编程中，联接可能意味着在未建模对象之间进行关联，例如对单向关系进行反向推理。 下面是单向关系的一个示例：`Student` 类有一个表示专业的 `Department` 类型的属性，但 `Department` 类没有作为 `Student` 对象集合的属性。 如果你具有一个 `Department` 对象列表，并且要查找每个部门中的所有学生，则可以使用 join 运算完成此项查找。

LINQ 框架中提供的 join 方法包括 Join 和 GroupJoin。 这些方法执行同等联接，即根据 2 个数据源的键是否相等来匹配这 2 个数据源的联接。 （对于比较，Transact-SQL 支持 `equals` 运算符以外的 join 运算符，例如 `less than` 运算符）在关系数据库术语中，Join 会实现内部 join，这是一种类型的 join，其中仅返回在其他数据集中具有匹配项的那些对象。 GroupJoin 方法在关系数据库术语中没有直接等效项，但实现了内部联接和左外部联接的超集。 左外部 join 是指返回第一个（左侧）数据源的每个元素的 join，即使其他数据源中没有关联元素。

下图显示了一个概念性视图，其中包含两个集合以及这两个集合中的包含在内部 join 或左外部 join 中的元素。



方法

展开表

方法名	描述	C# 查询表达式语法	更多信息
Join	根据键选择器函数联接两个序列并提取值对。	`join … in … on … equals …`	Enumerable.Join Queryable.Join
GroupJoin	根据键选择器函数联接两个序列，并对每个元素的结果匹配项进行分组。	`join … in … on … equals … into …`	Enumerable.GroupJoin Queryable.GroupJoin

本文中的以下示例使用该领域的常见数据源：

C#

```
public enum GradeLevel
{
    FirstYear = 1,
    SecondYear,
    ThirdYear,
    FourthYear
};

public class Student
{
    public required string FirstName { get; init; }
    public required string LastName { get; init; }
    public required int ID { get; init; }

    public required GradeLevel Year { get; init; }
    public required List<int> Scores { get; init; }

    public required int DepartmentID { get; init; }
}

public class Teacher
{
    public required string First { get; init; }
    public required string Last { get; init; }
    public required int ID { get; init; }
    public required string City { get; init; }
}
public class Department
{
    public required string Name { get; init; }
    public int ID { get; init; }
```

```
    public required int TeacherID { get; init; }
}
```

每个 `Student` 都有年级、主要院系和一系列分数。 `Teacher` 还有一个 `City` 属性，用于标识教师的授课校区。 `Department` 有一个名称，以及对担任院系主任的 `Teacher` 的引用。

下面的示例使用 `join … in … on … equals …` 子句基于特定值 join 两个序列：

C#

```
var query = from student in students
            join department in departments on student.DepartmentID equals
department.ID
            select new { Name = $"{student.FirstName} {student.LastName}",
DepartmentName = department.Name };

foreach (var item in query)
{
    Console.WriteLine($"{item.Name} - {item.DepartmentName}");
}
```

可以使用方法语法来表示前面的查询，如以下代码所示：

C#

```
var query = students.Join(departments,
    student => student.DepartmentID, department => department.ID,
    (student, department) => new { Name = $"{student.FirstName}
{student.LastName}", DepartmentName = department.Name });

foreach (var item in query)
{
    Console.WriteLine($"{item.Name} - {item.DepartmentName}");
}
```

下面的示例使用 `join … in … on … equals … into …` 子句基于特定值 join 两个序列，并对每个元素的结果匹配项进行分组：

C#

```
IEnumerable<IEnumerable<Student>> studentGroups = from department in
departments
                    join student in students on department.ID equals
student.DepartmentID into studentGroup
                    select studentGroup;

foreach (IEnumerable<Student> studentGroup in studentGroups)
```

```
{
    Console.WriteLine("Group");
    foreach (Student student in studentGroup)
    {
        Console.WriteLine($"  - {student.FirstName}, {student.LastName}");
    }
}
```

可以使用方法语法来表示前面的查询，如以下示例所示：

C#

```
// Join department and student based on DepartmentId and grouping result
IEnumerable<IEnumerable<Student>> studentGroups =
departments.GroupJoin(students,
    department => department.ID, student => student.DepartmentID,
    (department, studentGroup) => studentGroup);

foreach (IEnumerable<Student> studentGroup in studentGroups)
{
    Console.WriteLine("Group");
    foreach (Student student in studentGroup)
    {
        Console.WriteLine($"  - {student.FirstName}, {student.LastName}");
    }
}
```

执行内部联接

在关系数据库术语中，*内部 join* 会生成一个结果集，在该结果集中，第一个集合的每个元素对于第二个集合中的每个匹配元素都会出现一次。 如果第一个集合中的元素没有匹配元素，则它不会出现在结果集中。 由 C# 中的 `join` 子句调用的 Join 方法可实现内部 join。 以下示例演示如何执行内部 join 的 4 种变体：

- 基于简单键使两个数据源中的元素相关联的简单内部 join。
- 基于*复合*键使两个数据源中的元素相关联的内部 join。 复合键是由多个值组成的键，使你可以基于多个属性使元素相关联。
- 在其中将连续 join 操作相互追加的*多 join*。
- 使用分组 join 实现的内部 join。

单键 join

以下示例将 `Teacher` 对象与 `Deparment` 对象进行匹配，后者的 `TeacherId` 与该 `Teacher` 相匹配。 C# 中的 `select` 子句定义结果对象的外观。 在下面的示例中，结果对象是由院系名称和领导该院系的教师的姓名组成的匿名类型。

C#

```
var query = from department in departments
            join teacher in teachers on department.TeacherID equals
teacher.ID
            select new
            {
                DepartmentName = department.Name,
                TeacherName = $"{teacher.First} {teacher.Last}"
            };

foreach (var departmentAndTeacher in query)
{
    Console.WriteLine($"{departmentAndTeacher.DepartmentName} is managed by
{departmentAndTeacher.TeacherName}");
}
```

使用 Join 方法语法实现相同的结果：

C#

```
var query = teachers
    .Join(departments, teacher => teacher.ID, department =>
department.TeacherID,
        (teacher, department) =>
        new { DepartmentName = department.Name, TeacherName = $"
{teacher.First} {teacher.Last}" });

foreach (var departmentAndTeacher in query)
{
    Console.WriteLine($"{departmentAndTeacher.DepartmentName} is managed by
{departmentAndTeacher.TeacherName}");
}
```

不是院系主任的教师不会出现在最终结果中。

复合键 join

可以使用复合键基于多个属性来比较元素，而不是只基于一个属性使元素相关联。 请为每个集合指定键选择器函数，以返回由要比较的属性组成的匿名类型。 如果对属性进行标记，则它们必须在每个键的匿名类型中具有相同标签。 属性还必须按相同顺序出现。

下面的示例使用 `Teacher` 对象列表和 `Student` 对象列表来确定哪些教师同时还是学生。 这两种类型都具有表示每个人的名字和姓氏的属性。 通过每个列表的元素创建 join 键的函数会返回由属性组成的匿名类型。 join 运算会比较这些复合键是否相等，并从每个列表返回名字和姓氏都匹配的对象对。

C#

```
// Join the two data sources based on a composite key consisting of first
and last name,
// to determine which employees are also students.
IEnumerable<string> query =
    from teacher in teachers
    join student in students on new
    {
        FirstName = teacher.First,
        LastName = teacher.Last
    } equals new
    {
        student.FirstName,
        student.LastName
    }
    select teacher.First + " " + teacher.Last;

string result = "The following people are both teachers and students:\r\n";
foreach (string name in query)
{
    result += $"{name}\r\n";
}
Console.Write(result);
```

使用 Join 方法，如以下示例所示：

C#

```
IEnumerable<string> query = teachers
    .Join(students,
        teacher => new { FirstName = teacher.First, LastName = teacher.Last
},
        student => new { student.FirstName, student.LastName },
        (teacher, student) => $"{teacher.First} {teacher.Last}"
 );

Console.WriteLine("The following people are both teachers and students:");
foreach (string name in query)
{
    Console.WriteLine(name);
}
```

多个 join

可以将任意数量的 join 操作相互追加，以执行多 join。 C# 中的每个 `join` 子句会将指定数据源与上一个 join 的结果相关联。

第一个 `join` 子句根据 `Student` 对象的 `DepartmentID` 与 `Department` 对象的 `ID` 的匹配情况，将学生和院系进行匹配。 它会返回一个包含 `Student` 对象和 `Department` 对象的匿名

类型的序列。

第二个 `join` 子句将第一个 join 返回的匿名类型与基于该教师 ID（与部门主管 ID 匹配）的 `Teacher` 对象相关联。 它会返回一个包含学生姓名、院系名称和院系主任姓名的匿名类型序列。 由于此操作是内部 join，因此只返回第一个数据源中在第二个数据源中具有匹配项的对象。

C#

```
// The first join matches Department.ID and Student.DepartmentID from the
list of students and
// departments, based on a common ID. The second join matches teachers who
lead departments
// with the students studying in that department.
var query = from student in students
    join department in departments on student.DepartmentID equals
department.ID
    join teacher in teachers on department.TeacherID equals teacher.ID
    select new {
        StudentName = $"{student.FirstName} {student.LastName}",
        DepartmentName = department.Name,
        TeacherName = $"{teacher.First} {teacher.Last}"
    };

foreach (var obj in query)
{
    Console.WriteLine($"""The student "{obj.StudentName}" studies in the
department run by "{obj.TeacherName}".""");
}
```

使用多个 Join 方法的等效方法对匿名类型使用同一方法：

C#

```
var query = students
    .Join(departments, student => student.DepartmentID, department =>
department.ID,
        (student, department) => new { student, department })
    .Join(teachers, commonDepartment =>
commonDepartment.department.TeacherID, teacher => teacher.ID,
        (commonDepartment, teacher) => new
        {
            StudentName = $"{commonDepartment.student.FirstName}
{commonDepartment.student.LastName}",
            DepartmentName = commonDepartment.department.Name,
            TeacherName = $"{teacher.First} {teacher.Last}"
        });

foreach (var obj in query)
{
    Console.WriteLine($"""The student "{obj.StudentName}" studies in the
```

```
department run by "{obj.TeacherName}".""");
}
```

使用分组的 join 实现的内部 join

下面的示例演示如何使用分组 join 实现内部 join。 `Department` 对象列表会基于 `Department.ID` 与 `Student.DepartmentID` 属性的匹配情况，分组联接到 `Student` 对象列表中。 分组 join 会创建中间组的集合，其中每个组都包含 `Department` 对象和匹配 `Student` 对象的序列。 第二个 `from` 子句将此序列的序列合并（或平展）为一个较长的序列。 `select` 子句指定最终序列中元素的类型。 该类型是一个由学生姓名和匹配的院系名称组成的匿名类型。

C#

```
var query1 =
    from department in departments
    join student in students on department.ID equals student.DepartmentID
into gj
    from subStudent in gj
    select new
    {
        DepartmentName = department.Name,
        StudentName = $"{subStudent.FirstName} {subStudent.LastName}"
    };
Console.WriteLine("Inner join using GroupJoin():");
foreach (var v in query1)
{
    Console.WriteLine($"{v.DepartmentName} - {v.StudentName}");
}
```

可以使用 GroupJoin 方法实现相同的结果，如下所示：

C#

```
var queryMethod1 = departments
    .GroupJoin(students, department => department.ID, student =>
student.DepartmentID,
        (department, gj) => new { department, gj })
    .SelectMany(departmentAndStudent => departmentAndStudent.gj,
        (departmentAndStudent, subStudent) => new
        {
            DepartmentName = departmentAndStudent.department.Name,
            StudentName = $"{subStudent.FirstName} {subStudent.LastName}"
        });

Console.WriteLine("Inner join using GroupJoin():");
foreach (var v in queryMethod1)
{
```

```
    Console.WriteLine($"{v.DepartmentName} - {v.StudentName}");
}
```

结果等效于通过使用 `join` 子句（不使用 `into` 子句）执行内部 join 来获取的结果集。 以下代码演示了此等效查询：

C#

```
var query2 = from department in departments
    join student in students on department.ID equals student.DepartmentID
    select new
    {
        DepartmentName = department.Name,
        StudentName = $"{student.FirstName} {student.LastName}"
    };

Console.WriteLine("The equivalent operation using Join():");
foreach (var v in query2)
{
    Console.WriteLine($"{v.DepartmentName} - {v.StudentName}");
}
```

为避免链接，可以使用单个 Join 方法，如此处所示：

C#

```
var queryMethod2 = departments.Join(students, departments => departments.ID,
student => student.DepartmentID,
    (department, student) => new
    {
        DepartmentName = department.Name,
        StudentName = $"{student.FirstName} {student.LastName}"
    });

Console.WriteLine("The equivalent operation using Join():");
foreach (var v in queryMethod2)
{
    Console.WriteLine($"{v.DepartmentName} - {v.StudentName}");
}
```

执行分组联接

分组 join 对于生成分层数据结构十分有用。 它将第一个集合中的每个元素与第二个集合中的一组相关元素进行配对。

ⓘ 备注

第一个集合的每个元素都会出现在分组 join 的结果集中（无论是否在第二个集合中找到关联元素）。 在未找到任何相关元素的情况下，该元素的相关元素序列为空。 因此，结果选择器有权访问第一个集合的每个元素。 这与非分组 join 中的结果选择器不同，后者无法访问第一个集合中在第二个集合中没有匹配项的元素。

> ⚠ **警告**
>
> Enumerable.GroupJoin 在传统关系数据库术语中没有直接等效项。 但是，此方法实现了内部联接和左外部联接的超集。 这两个操作都可以按照分组 join 进行编写。 有关详细信息，请参阅 Entity Framework Core，GroupJoin。

本文的第一个示例演示如何执行分组 join。 第二个示例演示如何使用分组 join 创建 XML 元素。

分组 join

下面的示例基于与 `Student.DepartmentID` 属性匹配的 `Department.ID`，来执行类型 `Department` 和 `Student` 的对象的分组 join。 与非分组 join（会为每个匹配生成元素对）不同，分组 join 只为第一个集合的每个元素生成一个结果对象（在此示例中为 `Department` 对象）。 第二个集合中的对应元素（在此示例中为 `Student` 对象）会分组到集合中。 最后，结果选择器函数会为每个匹配都创建一种匿名类型，其中包含 `Department.Name` 和 `Student` 对象集合。

C#

```
var query = from department in departments
    join student in students on department.ID equals student.DepartmentID
into studentGroup
    select new
    {
        DepartmentName = department.Name,
        Students = studentGroup
    };

foreach (var v in query)
{
    // Output the department's name.
    Console.WriteLine($"{v.DepartmentName}:");

    // Output each of the students in that department.
    foreach (Student? student in v.Students)
    {
        Console.WriteLine($"  {student.FirstName} {student.LastName}");
```

```
    }
}
```

在上面的示例中，`query` 变量包含的查询创建了一个列表，其中每个元素都是匿名类型，包含院系名称和在该院系学习的学生的集合。

以下代码显示了使用方法语法的等效查询：

C#

```
var query = departments.GroupJoin(students, department => department.ID,
student => student.DepartmentID,
    (department, Students) => new { DepartmentName = department.Name,
Students });

foreach (var v in query)
{
    // Output the department's name.
    Console.WriteLine($"{v.DepartmentName}:");

    // Output each of the students in that department.
    foreach (Student? student in v.Students)
    {
        Console.WriteLine($"  {student.FirstName} {student.LastName}");
    }
}
```

用于创建 XML 的分组 join

分组联接非常适合于使用 LINQ to XML 创建 XML。 下面的示例类似于上面的示例，不过结果选择器函数不会创建匿名类型，而是创建表示联接对象的 XML 元素。

C#

```
XElement departmentsAndStudents = new("DepartmentEnrollment",
    from department in departments
    join student in students on department.ID equals student.DepartmentID
into studentGroup
    select new XElement("Department",
        new XAttribute("Name", department.Name),
        from student in studentGroup
        select new XElement("Student",
            new XAttribute("FirstName", student.FirstName),
            new XAttribute("LastName", student.LastName)
        )
    )
);
```

```
Console.WriteLine(departmentsAndStudents);
```

以下代码显示了使用方法语法的等效查询：

C#

```
XElement departmentsAndStudents = new("DepartmentEnrollment",
    departments.GroupJoin(students, department => department.ID, student =>
student.DepartmentID,
        (department, Students) => new XElement("Department",
            new XAttribute("Name", department.Name),
            from student in Students
            select new XElement("Student",
                new XAttribute("FirstName", student.FirstName),
                new XAttribute("LastName", student.LastName)
            )
        )
    )
);

Console.WriteLine(departmentsAndStudents);
```

执行左外部联接

左外部 join 是一个 join，其中返回第一个集合的每个元素，无论该元素在第二个集合中是否有任何相关元素。 可以使用 LINQ 通过对分组 join 的结果调用 DefaultIfEmpty 方法来执行左外部 join。

下面的示例演示如何对分组 join 的结果调用 DefaultIfEmpty 方法来执行左外部 join。

若要生成两个集合的左外部 join，第一步是使用分组 join 执行内联 join。 （有关此过程的说明，请参阅执行内联。）在此示例中，`Department` 对象列表基于与学生的 `DepartmentID` 匹配的 `Department` 对象的 ID，内部联接到 `Student` 对象列表。

第二步是在结果集内包含第一个（左）集合的每个元素，即使该元素在右集合中没有匹配的元素也是如此。 这是通过对分组 join 中的每个匹配元素序列调用 DefaultIfEmpty 来实现的。 此示例中，对每个匹配 `Student` 对象的序列调用 DefaultIfEmpty。 如果对于任何 `Department` 对象，匹配的 `Student` 对象的序列为空，则该方法返回一个包含单个默认值的集合，确保结果集合中显示每个 `Department` 对象。

> **备注**

引用类型的默认值为 `null`；因此，该示例在访问每个 `Student` 集合的每个元素之前会先检查是否存在空引用。

C#

```
var query =
    from student in students
    join department in departments on student.DepartmentID equals
department.ID into gj
    from subgroup in gj.DefaultIfEmpty()
    select new
    {
        student.FirstName,
        student.LastName,
        Department = subgroup?.Name ?? string.Empty
    };

foreach (var v in query)
{
    Console.WriteLine($"{v.FirstName:-15} {v.LastName:-15}:
{v.Department}");
}
```

以下代码显示了使用方法语法的等效查询：

C#

```
var query = students.GroupJoin(departments, student => student.DepartmentID,
department => department.ID,
    (student, departmentList) => new { student, subgroup =
departmentList.AsQueryable() })
    .SelectMany(joinedSet => joinedSet.subgroup.DefaultIfEmpty(), (student,
department) => new
    {
        student.student.FirstName,
        student.student.LastName,
        Department = department.Name
    });

foreach (var v in query)
{
    Console.WriteLine($"{v.FirstName:-15} {v.LastName:-15}:
{v.Department}");
}
```

另请参阅

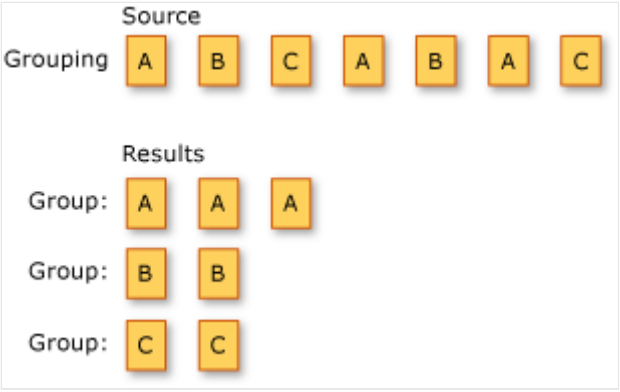
- Join

- GroupJoin
- 匿名类型
- 构建联接和跨产品查询
- join 子句
- group 子句
- 如何 join 不同文件的内容 (LINQ) (C#)
- 如何从多个源填充对象集合 (LINQ) (C#)

对数据分组 (C#)

项目 • 2024/06/06

分组是指将数据分到不同的组，使每组中的元素拥有公共的属性。 下图演示了对字符序列进行分组的结果。 每个组的键是字符。



> **ⓘ 重要**
>
> 这些示例使用 System.Collections.Generic.IEnumerable<T> 数据源。 基于 System.Linq.IQueryProvider 的数据源使用 System.Linq.IQueryable<T> 数据源和**表达式树**。 表达式树对允许的 C# 语法有**限制**。 此外，每个 `IQueryProvider` 数据源（如 EF Core）可能会施加更多限制。 查看数据源的文档。

下表中列出了对数据元素进行分组的标准查询运算符方法。

展开表

方法名	描述	C# 查询表达式语法	详细信息

GroupBy	对共享通用属性的元素进行分组。 一个 IGrouping<TKey,TElement> 对象表示每个组。	`group … by`
		-或-
		`group … by … into …`
			Enumerable.GroupBy
			Queryable.GroupBy

ToLookup	将元素插入基于键选择器函数的 Lookup<TKey,TElement>（一种一对多字典）。	不适用。	Enumerable.ToLookup

下列代码示例根据奇偶性，使用 `group by` 子句对列表中的整数进行分组。

C#

```
List<int> numbers = [35, 44, 200, 84, 3987, 4, 199, 329, 446, 208];

IEnumerable<IGrouping<int, int>> query = from number in numbers
                                         group number by number % 2;

foreach (var group in query)
{
    Console.WriteLine(group.Key == 0 ? "\nEven numbers:" : "\nOdd
numbers:");
    foreach (int i in group)
    {
        Console.WriteLine(i);
    }
}
```

以下代码显示了使用方法语法的等效查询：

C#

```
List<int> numbers = [35, 44, 200, 84, 3987, 4, 199, 329, 446, 208];

IEnumerable<IGrouping<int, int>> query = numbers
    .GroupBy(number => number % 2);

foreach (var group in query)
{
    Console.WriteLine(group.Key == 0 ? "\nEven numbers:" : "\nOdd
numbers:");
    foreach (int i in group)
    {
        Console.WriteLine(i);
    }
}
```

本文中的以下示例使用此区域的常见数据源：

C#

```
public enum GradeLevel
{
    FirstYear = 1,
    SecondYear,
    ThirdYear,
    FourthYear
};

public class Student
{
    public required string FirstName { get; init; }
```

```
    public required string LastName { get; init; }
    public required int ID { get; init; }

    public required GradeLevel Year { get; init; }
    public required List<int> Scores { get; init; }

    public required int DepartmentID { get; init; }
}

public class Teacher
{
    public required string First { get; init; }
    public required string Last { get; init; }
    public required int ID { get; init; }
    public required string City { get; init; }
}
public class Department
{
    public required string Name { get; init; }
    public int ID { get; init; }

    public required int TeacherID { get; init; }
}
```

每个 `Student` 都有年级、主要院系和一系列分数。 `Teacher` 还有一个 `City` 属性，用于标识教师的授课校区。 `Department` 有一个名称，和对担任部门负责人的 `Teacher` 的引用。

对查询结果进行分组

分组是 LINQ 最强大的功能之一。 以下示例演示如何以各种方式对数据进行分组：

- 依据单个属性。
- 依据字符串属性的首字母。
- 依据计算出的数值范围。
- 依据布尔谓词或其他表达式。
- 依据组合键。

此外，最后两个查询将其结果投影到一个新的匿名类型中，该类型仅包含学生的名字和姓氏。 有关详细信息，请参阅 group 子句。

按单个属性分组示例

以下示例演示如何通过使用元素的单个属性作为分组键对源元素进行分组。 键是一个 `enum`，学生的在校时间。 分组操作对该类型使用默认的相等比较器。

```
```

C#

```
var groupByYearQuery =
    from student in students
    group student by student.Year into newGroup
    orderby newGroup.Key
    select newGroup;

foreach (var yearGroup in groupByYearQuery)
{
    Console.WriteLine($"Key: {yearGroup.Key}");
    foreach (var student in yearGroup)
    {
        Console.WriteLine($"\t{student.LastName}, {student.FirstName}");
    }
}
```

以下示例显示了使用方法语法的等效代码：

C#

```
// Variable groupByLastNamesQuery is an IEnumerable<IGrouping<string,
// DataClass.Student>>.
var groupByYearQuery = students
    .GroupBy(student => student.Year)
    .OrderBy(newGroup => newGroup.Key);

foreach (var yearGroup in groupByYearQuery)
{
    Console.WriteLine($"Key: {yearGroup.Key}");
    foreach (var student in yearGroup)
    {
        Console.WriteLine($"\t{student.LastName}, {student.FirstName}");
    }
}
```

按值分组示例

下例演示如何通过使用除对象属性以外的某个项作为分组键对源元素进行分组。 在此示例中，键是学生姓氏的第一个字母。

C#

```
var groupByFirstLetterQuery =
    from student in students
    let firstLetter = student.LastName[0]
    group student by firstLetter;

foreach (var studentGroup in groupByFirstLetterQuery)
{
```

```
    Console.WriteLine($"Key: {studentGroup.Key}");
    foreach (var student in studentGroup)
    {
        Console.WriteLine($"\t{student.LastName}, {student.FirstName}");
    }
}
```

需要嵌套 foreach 才能访问组项。

以下示例显示了使用方法语法的等效代码：

C#

```
var groupByFirstLetterQuery = students
    .GroupBy(student => student.LastName[0]);

foreach (var studentGroup in groupByFirstLetterQuery)
{
    Console.WriteLine($"Key: {studentGroup.Key}");
    foreach (var student in studentGroup)
    {
        Console.WriteLine($"\t{student.LastName}, {student.FirstName}");
    }
}
```

按范围分组示例

以下示例演示如何通过使用某个数值范围作为分组键对源元素进行分组。 然后，查询将结果投影到一个匿名类型中，该类型仅包含学生的名字和姓氏以及该学生所属的百分等级范围。 使用匿名类型的原因是没有必要使用完整的 `Student` 对象来显示结果。 `GetPercentile` 是一个帮助程序函数，它根据学生的平均分数计算百分比。 该方法返回 0 到 10 之间的整数。

C#

```
static int GetPercentile(Student s)
{
    double avg = s.Scores.Average();
    return avg > 0 ? (int)avg / 10 : 0;
}

var groupByPercentileQuery =
    from student in students
    let percentile = GetPercentile(student)
    group new
    {
        student.FirstName,
        student.LastName
    } by percentile into percentGroup
```

```
    orderby percentGroup.Key
    select percentGroup;

foreach (var studentGroup in groupByPercentileQuery)
{
    Console.WriteLine($"Key: {studentGroup.Key * 10}");
    foreach (var item in studentGroup)
    {
        Console.WriteLine($"\t{item.LastName}, {item.FirstName}");
    }
}
```

需要嵌套 foreach 才能循环访问组和组项。 以下示例显示了使用方法语法的等效代码：

C#

```
static int GetPercentile(Student s)
{
    double avg = s.Scores.Average();
    return avg > 0 ? (int)avg / 10 : 0;
}

var groupByPercentileQuery = students
    .Select(student => new { student, percentile = GetPercentile(student) })
    .GroupBy(student => student.percentile)
    .Select(percentGroup => new
    {
        percentGroup.Key,
        Students = percentGroup.Select(s => new { s.student.FirstName,
s.student.LastName })
    })
    .OrderBy(percentGroup => percentGroup.Key);

foreach (var studentGroup in groupByPercentileQuery)
{
    Console.WriteLine($"Key: {studentGroup.Key * 10}");
    foreach (var item in studentGroup.Students)
    {
        Console.WriteLine($"\t{item.LastName}, {item.FirstName}");
    }
}
```

按比较分组示例

以下示例演示如何通过使用布尔比较表达式对源元素进行分组。 在此示例中，布尔表达式会测试学生的平均考试分数是否超过 75。 与上述示例一样，将结果投影到一个匿名类型中，因为不需要完整的源元素。 匿名类型中的属性将成为 `Key` 成员上的属性。

C#

```
var groupByHighAverageQuery =
    from student in students
    group new
    {
        student.FirstName,
        student.LastName
    } by student.Scores.Average() > 75 into studentGroup
    select studentGroup;

foreach (var studentGroup in groupByHighAverageQuery)
{
    Console.WriteLine($"Key: {studentGroup.Key}");
    foreach (var student in studentGroup)
    {
        Console.WriteLine($"\t{student.FirstName} {student.LastName}");
    }
}
```

以下代码显示了使用方法语法的等效查询：

C#

```
var groupByHighAverageQuery = students
    .GroupBy(student => student.Scores.Average() > 75)
    .Select(group => new
    {
        group.Key,
        Students = group.AsEnumerable().Select(s => new { s.FirstName,
s.LastName })
    });

foreach (var studentGroup in groupByHighAverageQuery)
{
    Console.WriteLine($"Key: {studentGroup.Key}");
    foreach (var student in studentGroup.Students)
    {
        Console.WriteLine($"\t{student.FirstName} {student.LastName}");
    }
}
```

按匿名类型分组

以下示例演示如何使用匿名类型来封装包含多个值的键。 在此示例中，第一个键值是学生姓氏的第一个字母。 第二个键值是一个布尔值，指定该学生在第一次考试中的得分是否超过了 85。 可以按照该键中的任何属性对组进行排序。

C#

```
var groupByCompoundKey =
    from student in students
    group student by new
    {
        FirstLetterOfLastName = student.LastName[0],
        IsScoreOver85 = student.Scores[0] > 85
    } into studentGroup
    orderby studentGroup.Key.FirstLetterOfLastName
    select studentGroup;

foreach (var scoreGroup in groupByCompoundKey)
{
    var s = scoreGroup.Key.IsScoreOver85 ? "more than 85" : "less than 85";
    Console.WriteLine($"Name starts with
{scoreGroup.Key.FirstLetterOfLastName} who scored {s}");
    foreach (var item in scoreGroup)
    {
        Console.WriteLine($"\t{item.FirstName} {item.LastName}");
    }
}
```

以下代码显示了使用方法语法的等效查询：

C#

```
var groupByCompoundKey = students
    .GroupBy(student => new
    {
        FirstLetterOfLastName = student.LastName[0],
        IsScoreOver85 = student.Scores[0] > 85
    })
    .OrderBy(studentGroup => studentGroup.Key.FirstLetterOfLastName);

foreach (var scoreGroup in groupByCompoundKey)
{
    var s = scoreGroup.Key.IsScoreOver85 ? "more than 85" : "less than 85";
    Console.WriteLine($"Name starts with
{scoreGroup.Key.FirstLetterOfLastName} who scored {s}");
    foreach (var item in scoreGroup)
    {
        Console.WriteLine($"\t{item.FirstName} {item.LastName}");
    }
}
```

创建嵌套组

以下示例演示如何在 LINQ 查询表达式中创建嵌套组。 首先根据学生年级创建每个组，然后根据每个人的姓名进一步细分为小组。

C#

```
var nestedGroupsQuery =
    from student in students
    group student by student.Year into newGroup1
    from newGroup2 in
    from student in newGroup1
    group student by student.LastName
    group newGroup2 by newGroup1.Key;

foreach (var outerGroup in nestedGroupsQuery)
{
    Console.WriteLine($"DataClass.Student Level = {outerGroup.Key}");
    foreach (var innerGroup in outerGroup)
    {
        Console.WriteLine($"\tNames that begin with: {innerGroup.Key}");
        foreach (var innerGroupElement in innerGroup)
        {
            Console.WriteLine($"\t\t{innerGroupElement.LastName}
{innerGroupElement.FirstName}");
        }
    }
}
```

需要使用三个嵌套的 `foreach` 循环来循环访问嵌套组的内部元素。
（将鼠标光标悬停在迭代变量 `outerGroup`、`innerGroup` 和 `innerGroupElement` 上以查看其实际类型。）

以下代码显示了使用方法语法的等效查询：

C#

```
var nestedGroupsQuery =
    students
    .GroupBy(student => student.Year)
    .Select(newGroup1 => new
    {
        newGroup1.Key,
        NestedGroup = newGroup1
            .GroupBy(student => student.LastName)
    });

foreach (var outerGroup in nestedGroupsQuery)
{
    Console.WriteLine($"DataClass.Student Level = {outerGroup.Key}");
    foreach (var innerGroup in outerGroup.NestedGroup)
    {
        Console.WriteLine($"\tNames that begin with: {innerGroup.Key}");
        foreach (var innerGroupElement in innerGroup)
        {
            Console.WriteLine($"\t\t{innerGroupElement.LastName}
{innerGroupElement.FirstName}");
```

```
        }
    }
}
```

对分组操作执行子查询

本文演示创建查询的两种不同方式，此查询将源数据排序成组，然后分别对每个组执行子查询。 每个示例中的基本方法是使用名为 `newGroup` 的“接续块”对源元素进行分组，然后针对 `newGroup` 生成新的子查询。 针对由外部查询创建的每个新组运行此子查询。 在此特定示例中，最终输出不是一个组，而是一个匿名类型的线性序列。

有关如何分组的详细信息，请参阅 group 子句。 有关接续块的详细信息，请参阅 into。 下面的示例使用内存数据结构作为数据源，但相同的原则适用于任何类型的 LINQ 数据源。

C#

```
var queryGroupMax =
    from student in students
    group student by student.Year into studentGroup
    select new
    {
        Level = studentGroup.Key,
        HighestScore = (
            from student2 in studentGroup
            select student2.Scores.Average()
        ).Max()
    };

var count = queryGroupMax.Count();
Console.WriteLine($"Number of groups = {count}");

foreach (var item in queryGroupMax)
{
    Console.WriteLine($"  {item.Level} Highest Score={item.HighestScore}");
}
```

还可以使用方法语法编写上述代码片段中的查询。 下面的代码片段具有使用方法语法编写的语义上等效的查询。

C#

```
var queryGroupMax =
    students
        .GroupBy(student => student.Year)
        .Select(studentGroup => new
        {
            Level = studentGroup.Key,
```

```
            HighestScore = studentGroup.Max(student2 =>
student2.Scores.Average())
        });

var count = queryGroupMax.Count();
Console.WriteLine($"Number of groups = {count}");

foreach (var item in queryGroupMax)
{
    Console.WriteLine($"  {item.Level} Highest Score={item.HighestScore}");
}
```

另请参阅

- System.Linq
- GroupBy
- IGrouping<TKey,TElement>
- group 子句
- 如何使用组将一个文件拆分成多个文件 (LINQ) (C#)

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

如何操作：使用 LINQ 查询文件和目录

项目 • 2024/04/27

许多文件系统操作实质上是查询，因此非常适合使用 LINQ 方法。 这些查询是非破坏性的。 它们不会更改原始文件或文件夹的内容。 查询不应造成任何副作用。 通常，修改源数据的任何代码（包括执行创建/更新/删除操作的查询）应与仅查询数据的代码分开。

创建准确表示文件系统的内容并适当处理异常的数据源存在一定难度。 本部分中的示例创建 FileInfo 对象的快照集合，该集合表示指定的根文件夹及其所有子文件夹下的所有文件。 每个 FileInfo 的实际状态可能会在开始和结束执行查询期间发生更改。 例如，可以创建 FileInfo 对象的列表来用作数据源。 如果尝试通过查询访问 `Length` 属性，则 FileInfo 对象会尝试访问文件系统来更新 `Length` 的值。 如果该文件不再存在，则会在查询中收到 FileNotFoundException，即使未直接查询文件系统也是如此。

如何查询具有指定特性或名称的文件

此示例演示了如何在指定目录树中查找具有指定文件扩展名（如“.txt”）的所有文件。 它还演示了如何基于时间在树中返回最新或最旧的文件。 无论是在 Windows、Mac 还是 Linux 系统上运行此代码，都可能需要修改许多示例的第一行。

C#

```
string startFolder = """C:\Program Files\dotnet\sdk""";
// Or
// string startFolder = "/usr/local/share/dotnet/sdk";

DirectoryInfo dir = new DirectoryInfo(startFolder);
var fileList = dir.GetFiles("*.*", SearchOption.AllDirectories);

var fileQuery = from file in fileList
                where file.Extension == ".txt"
                orderby file.Name
                select file;

// Uncomment this block to see the full query
// foreach (FileInfo fi in fileQuery)
// {
//    Console.WriteLine(fi.FullName);
// }

var newestFile = (from file in fileQuery
                  orderby file.CreationTime
                  select new { file.FullName, file.CreationTime })
                  .Last();
```

```
Console.WriteLine($"\r\nThe newest .txt file is {newestFile.FullName}.
Creation time: {newestFile.CreationTime}");
```

如何按扩展名对文件分组

本示例演示如何使用 LINQ 来执行高级分组和对文件或文件夹列表执行排序操作。 它还演示如何使用 Skip 和 Take 方法在控制台窗口中对输出进行分页。

下面的查询演示如何按文件扩展名对指定的目录树的内容进行分组。

C#

```
string startFolder = """C:\Program Files\dotnet\sdk""";
// Or
// string startFolder = "/usr/local/share/dotnet/sdk";

int trimLength = startFolder.Length;

DirectoryInfo dir = new DirectoryInfo(startFolder);

var fileList = dir.GetFiles("*.*", SearchOption.AllDirectories);

var queryGroupByExt = from file in fileList
                      group file by file.Extension.ToLower() into fileGroup
                      orderby fileGroup.Count(), fileGroup.Key
                      select fileGroup;

// Iterate through the outer collection of groups.
foreach (var filegroup in queryGroupByExt.Take(5))
{
    Console.WriteLine($"Extension: {filegroup.Key}");
    var resultPage = filegroup.Take(20);

    //Execute the resultPage query
    foreach (var f in resultPage)
    {
        Console.WriteLine($"\t{f.FullName.Substring(trimLength)}");
    }
    Console.WriteLine();
}
```

此程序的输出可能很长，具体取决于本地文件系统的详细信息和 `startFolder` 的设置。 为了能够查看所有结果，此示例演示如何对结果进行分页。 由于每个组是单独枚举的，因此需要嵌套的 `foreach` 循环。

如何查询一组文件夹中的总字节数

此示例演示如何检索由指定文件夹及其所有子文件夹中的所有文件使用的字节总数。 Sum 方法可将 `select` 子句中选择的所有项的值相加。 可以修改此查询以检索指定目录树中的最大或最小文件，方法是调用 Min 或 Max 方法，而不是调用 Sum 方法。

C#

```
string startFolder = """C:\Program Files\dotnet\sdk""";
// Or
// string startFolder = "/usr/local/share/dotnet/sdk";

var fileList = Directory.GetFiles(startFolder, "*.*",
SearchOption.AllDirectories);

var fileQuery = from file in fileList
                let fileLen = new FileInfo(file).Length
                where fileLen > 0
                select fileLen;

// Cache the results to avoid multiple trips to the file system.
long[] fileLengths = fileQuery.ToArray();

// Return the size of the largest file
long largestFile = fileLengths.Max();

// Return the total number of bytes in all the files under the specified
folder.
long totalBytes = fileLengths.Sum();

Console.WriteLine($"There are {totalBytes} bytes in {fileList.Count()} files
under {startFolder}");
Console.WriteLine($"The largest file is {largestFile} bytes.");
```

此示例扩展了前面的示例以执行以下操作：

- 如何检索最大文件的大小（以字节为单位）。
- 如何检索最小文件的大小（以字节为单位）。
- 如何从指定根文件夹下的一个或多个文件夹检索 FileInfo 对象最大或最小文件。
- 如何检索序列（如 10 个最大文件）。
- 如何基于文件大小（以字节为单位）按组对文件进行排序（忽略小于指定大小的文件）。

下面的示例包含五个单独的查询，它们演示如何根据文件大小（以字节为单位）对文件进行查询和分组。 可以修改这些示例，以便使查询基于 FileInfo 对象的其他某个属性。

C#

```
// Return the FileInfo object for the largest file
// by sorting and selecting from beginning of list
FileInfo longestFile = (from file in fileList
```

```
                        let fileInfo = new FileInfo(file)
                        where fileInfo.Length > 0
                        orderby fileInfo.Length descending
                        select fileInfo
                        ).First();

Console.WriteLine($"The largest file under {startFolder} is
{longestFile.FullName} with a length of {longestFile.Length} bytes");

//Return the FileInfo of the smallest file
FileInfo smallestFile = (from file in fileList
                         let fileInfo = new FileInfo(file)
                         where fileInfo.Length > 0
                         orderby fileInfo.Length ascending
                         select fileInfo
                        ).First();

Console.WriteLine($"The smallest file under {startFolder} is
{smallestFile.FullName} with a length of {smallestFile.Length} bytes");

//Return the FileInfos for the 10 largest files
var queryTenLargest = (from file in fileList
                       let fileInfo = new FileInfo(file)
                       let len = fileInfo.Length
                       orderby len descending
                       select fileInfo
                      ).Take(10);

Console.WriteLine($"The 10 largest files under {startFolder} are:");

foreach (var v in queryTenLargest)
{
    Console.WriteLine($"{v.FullName}: {v.Length} bytes");
}

// Group the files according to their size, leaving out
// files that are less than 200000 bytes.
var querySizeGroups = from file in fileList
                      let fileInfo = new FileInfo(file)
                      let len = fileInfo.Length
                      where len > 0
                      group fileInfo by (len / 100000) into fileGroup
                      where fileGroup.Key >= 2
                      orderby fileGroup.Key descending
                      select fileGroup;

foreach (var filegroup in querySizeGroups)
{
    Console.WriteLine($"{filegroup.Key}00000");
    foreach (var item in filegroup)
    {
        Console.WriteLine($"\t{item.Name}: {item.Length}");
    }
}
```

若要返回一个或多个完整的 FileInfo 对象，查询必须首先检查数据中的每个对象，然后按其 Length 属性值对它们进行排序。 随后它便可以返回具有最大长度的单个对象或对象序列。 使用 First 返回列表中的第一个元素。 使用 Take 返回前 n 个元素。 指定降序排序顺序可将最小元素置于列表开头。

如何在目录树中查询重复文件

有时，可能会有相同名称的文件位于多个文件夹中。 此示例显示如何在指定根文件夹下查询此类重复文件名。 第二个示例显示如何查询大小和上次写入时间都匹配的文件。

C#

```
string startFolder = """C:\Program Files\dotnet\sdk""";
// Or
// string startFolder = "/usr/local/share/dotnet/sdk";

DirectoryInfo dir = new DirectoryInfo(startFolder);

IEnumerable<FileInfo> fileList = dir.GetFiles("*.*",
SearchOption.AllDirectories);

// used in WriteLine to keep the lines shorter
int charsToSkip = startFolder.Length;

// var can be used for convenience with groups.
var queryDupNames = from file in fileList
                    group file.FullName.Substring(charsToSkip) by file.Name
into fileGroup
                    where fileGroup.Count() > 1
                    select fileGroup;

foreach (var queryDup in queryDupNames.Take(20))
{
    Console.WriteLine($"Filename = {(queryDup.Key.ToString() == string.Empty
? "[none]" : queryDup.Key.ToString())}");

    foreach (var fileName in queryDup.Take(10))
    {
        Console.WriteLine($"\t{fileName}");
    }
}
```

第一个查询使用关键值来确定匹配项。 它会查找名称相同但内容可能不同的文件。 第二个查询使用复合键来匹配 FileInfo 对象的 3 个属性。 此查询更可能找到名称相同且内容相似或相同的文件。

C#

```
    string startFolder = """C:\Program Files\dotnet\sdk""";
    // Or
    // string startFolder = "/usr/local/share/dotnet/sdk";

    // Make the lines shorter for the console display
    int charsToSkip = startFolder.Length;

    // Take a snapshot of the file system.
    DirectoryInfo dir = new DirectoryInfo(startFolder);
    IEnumerable<FileInfo> fileList = dir.GetFiles("*.*",
SearchOption.AllDirectories);

    // Note the use of a compound key. Files that match
    // all three properties belong to the same group.
    // A named type is used to enable the query to be
    // passed to another method. Anonymous types can also be used
    // for composite keys but cannot be passed across method boundaries
    //
    var queryDupFiles = from file in fileList
                        group file.FullName.Substring(charsToSkip) by
                        (Name: file.Name, LastWriteTime: file.LastWriteTime,
Length: file.Length )
                        into fileGroup
                        where fileGroup.Count() > 1
                        select fileGroup;

    foreach (var queryDup in queryDupFiles.Take(20))
    {
        Console.WriteLine($"Filename = {(queryDup.Key.ToString() ==
string.Empty ? "[none]" : queryDup.Key.ToString())}");

        foreach (var fileName in queryDup)
        {
            Console.WriteLine($"\t{fileName}");
        }
    }
}
```

如何查询文件夹中文本文件的内容

此示例演示如何查询指定目录树中的所有文件、打开每个文件并检查其内容。 此类技术可用于对目录树的内容创建索引或反向索引。 此示例中执行的是简单的字符串搜索。 但是，可使用正则表达式执行类型更复杂的模式匹配。

C#

```
string startFolder = """C:\Program Files\dotnet\sdk""";
// Or
// string startFolder = "/usr/local/share/dotnet/sdk";
```

```
DirectoryInfo dir = new DirectoryInfo(startFolder);

var fileList = dir.GetFiles("*.*", SearchOption.AllDirectories);

string searchTerm = "change";

var queryMatchingFiles = from file in fileList
                         where file.Extension == ".txt"
                         let fileText = File.ReadAllText(file.FullName)
                         where fileText.Contains(searchTerm)
                         select file.FullName;

// Execute the query.
Console.WriteLine($"""The term "{searchTerm}" was found in:""");
foreach (string filename in queryMatchingFiles)
{
    Console.WriteLine(filename);
}
```

如何比较两个文件夹的内容

此示例演示了比较两个文件列表的 3 种方法：

- 通过查询布尔值指定两个文件列表是否相同。
- 通过查询交集检索同时存在于两个文件夹中的文件。
- 通过查询差集检索仅存在于一个文件夹中的文件。

此处的方法适用于比较任何类型的对象序列。

此处的 `FileComparer` 类演示如何将自定义比较器类与标准查询运算符结合使用。 此类不适合在实际方案中使用。 它仅使用每个文件的名称和字节长度来确定每个文件夹的内容是否相同。 在实际方案中，应修改此比较器以执行更严格的等同性检查。

C#

```
// This implementation defines a very simple comparison
// between two FileInfo objects. It only compares the name
// of the files being compared and their length in bytes.
class FileCompare : IEqualityComparer<FileInfo>
{
    public bool Equals(FileInfo? f1, FileInfo? f2)
    {
        return (f1?.Name == f2?.Name &&
                f1?.Length == f2?.Length);
    }

    // Return a hash that reflects the comparison criteria. According to the
    // rules for IEqualityComparer<T>, if Equals is true, then the hash
codes must
    // also be equal. Because equality as defined here is a simple value
```

```
equality, not
    // reference identity, it is possible that two or more objects will
produce the same
    // hash code.
    public int GetHashCode(FileInfo fi)
    {
        string s = $"{fi.Name}{fi.Length}";
        return s.GetHashCode();
    }
}

public static void CompareDirectories()
{
    string pathA = """C:\Program Files\dotnet\sdk\8.0.104""";
    string pathB = """C:\Program Files\dotnet\sdk\8.0.204""";

    DirectoryInfo dir1 = new DirectoryInfo(pathA);
    DirectoryInfo dir2 = new DirectoryInfo(pathB);

    IEnumerable<FileInfo> list1 = dir1.GetFiles("*.*",
SearchOption.AllDirectories);
    IEnumerable<FileInfo> list2 = dir2.GetFiles("*.*",
SearchOption.AllDirectories);

    //A custom file comparer defined below
    FileCompare myFileCompare = new FileCompare();

    // This query determines whether the two folders contain
    // identical file lists, based on the custom file comparer
    // that is defined in the FileCompare class.
    // The query executes immediately because it returns a bool.
    bool areIdentical = list1.SequenceEqual(list2, myFileCompare);

    if (areIdentical == true)
    {
        Console.WriteLine("the two folders are the same");
    }
    else
    {
        Console.WriteLine("The two folders are not the same");
    }

    // Find the common files. It produces a sequence and doesn't
    // execute until the foreach statement.
    var queryCommonFiles = list1.Intersect(list2, myFileCompare);

    if (queryCommonFiles.Any())
    {
        Console.WriteLine($"The following files are in both folders (total
number = {queryCommonFiles.Count()}):");
        foreach (var v in queryCommonFiles.Take(10))
        {
            Console.WriteLine(v.Name); //shows which items end up in result
list
        }
```

```
    }
    else
    {
        Console.WriteLine("There are no common files in the two folders.");
    }

    // Find the set difference between the two folders.
    var queryList1Only = (from file in list1
                          select file)
                          .Except(list2, myFileCompare);

    Console.WriteLine();
    Console.WriteLine($"The following files are in list1 but not list2
(total number = {queryList1Only.Count()}):");
    foreach (var v in queryList1Only.Take(10))
    {
        Console.WriteLine(v.FullName);
    }

    var queryList2Only = (from file in list2
                          select file)
                          .Except(list1, myFileCompare);

    Console.WriteLine();
    Console.WriteLine($"The following files are in list2 but not list1
(total number = {queryList2Only.Count()}:");
    foreach (var v in queryList2Only.Take(10))
    {
        Console.WriteLine(v.FullName);
    }
}
```

如何对带分隔符文件的字段重新排序

逗号分隔值 (CSV) 文件是一种文本文件，通常用于存储电子表格数据或其他由行和列表示的表格数据。 通过使用 Split 方法分隔字段，可以轻松地使用 LINQ 查询和操作 CSV 文件。 事实上，可以将同一技术用于重新排列任何结构化文本行部分；它不局限于 CSV 文件。

在下面的示例中，假设存在三列分别代表学生的“姓氏”、“名字”和“ID”。这些字段基于学生的姓氏按字母顺序排列。 查询将生成一个新序列，其中首先出现的是 ID 列，后面的第二列组合了学生的名字和姓氏。 根据 ID 字段重新排列各行。 结果将保存到新文件，并且不会修改原始数据。 以下文本显示了以下示例中使用的spreadsheet1.csv 文件的内容：

```txt
Adams,Terry,120
Fakhouri,Fadi,116
```

```
Feng,Hanying,117
Garcia,Cesar,114
Garcia,Debra,115
Garcia,Hugo,118
Mortensen,Sven,113
O'Donnell,Claire,112
Omelchenko,Svetlana,111
Tucker,Lance,119
Tucker,Michael,122
Zabokritski,Eugene,121
```

以下代码可读取源文件并重新排列 CSV 文件中的每一列，以对列的顺序进行重新排列：

C#

```
string[] lines = File.ReadAllLines("spreadsheet1.csv");

// Create the query. Put field 2 first, then
// reverse and combine fields 0 and 1 from the old field
IEnumerable<string> query = from line in lines
                            let fields = line.Split(',')
                            orderby fields[2]
                            select $"{fields[2]}, {fields[1]} {fields[0]}";

File.WriteAllLines("spreadsheet2.csv", query.ToArray());

/* Output to spreadsheet2.csv:
111, Svetlana Omelchenko
112, Claire O'Donnell
113, Sven Mortensen
114, Cesar Garcia
115, Debra Garcia
116, Fadi Fakhouri
117, Hanying Feng
118, Hugo Garcia
119, Lance Tucker
120, Terry Adams
121, Eugene Zabokritski
122, Michael Tucker
*/
```

如何使用组将一个文件拆分成多个文件

此示例演示一种进行以下操作的方法：合并两个文件的内容，然后创建一组以新方式整理数据的新文件。 查询使用两个文件的内容。 以下文本显示了第一个文件 names1.txt 的内容：

txt

```
Bankov, Peter
Holm, Michael
Garcia, Hugo
Potra, Cristina
Noriega, Fabricio
Aw, Kam Foo
Beebe, Ann
Toyoshima, Tim
Guy, Wey Yuan
Garcia, Debra
```

第二个文件 names2.txt 包含一组不同的名称，其中一些名称与第一组相同：

txt

```
Liu, Jinghao
Bankov, Peter
Holm, Michael
Garcia, Hugo
Beebe, Ann
Gilchrist, Beth
Myrcha, Jacek
Giakoumakis, Leo
McLin, Nkenge
El Yassir, Mehdi
```

以下代码将查询这两个文件，采用这两个文件的并集，然后为每个组编写一个新文件，由姓氏的第一个字母定义：

C#

```
string[] fileA = File.ReadAllLines("names1.txt");
string[] fileB = File.ReadAllLines("names2.txt");

// Concatenate and remove duplicate names
var mergeQuery = fileA.Union(fileB);

// Group the names by the first letter in the last name.
var groupQuery = from name in mergeQuery
                 let n = name.Split(',')[0]
                 group name by n[0] into g
                 orderby g.Key
                 select g;

foreach (var g in groupQuery)
{
    string fileName = $"testFile_{g.Key}.txt";

    Console.WriteLine(g.Key);
```

```
    using StreamWriter sw = new StreamWriter(fileName);
    foreach (var item in g)
    {
        sw.WriteLine(item);
        // Output to console for example purposes.
        Console.WriteLine($"   {item}");
    }
}
/* Output:
    A
       Aw, Kam Foo
    B
       Bankov, Peter
       Beebe, Ann
    E
       El Yassir, Mehdi
    G
       Garcia, Hugo
       Guy, Wey Yuan
       Garcia, Debra
       Gilchrist, Beth
       Giakoumakis, Leo
    H
       Holm, Michael
    L
       Liu, Jinghao
    M
       Myrcha, Jacek
       McLin, Nkenge
    N
       Noriega, Fabricio
    P
       Potra, Cristina
    T
       Toyoshima, Tim
 */
```

如何联接来自不同文件的内容

本示例演示如何联接两个逗号分隔文件中的数据，这两个文件共享一个用作匹配键的公共值。 如果需要合并来自两个电子表格的数据，或者从一个电子表格和具有另一种格式的文件合并到一个新文件时，此技术很有用。 可以修改此示例以用于任何类型的结构化文本。

以下文本显示了 scores.csv 的内容。 此文件表示电子表格数据。 第 1 列是学生的 ID，第 2 至 5 列是测验分数。

```
txt
```

```
111, 97, 92, 81, 60
112, 75, 84, 91, 39
113, 88, 94, 65, 91
114, 97, 89, 85, 82
115, 35, 72, 91, 70
116, 99, 86, 90, 94
117, 93, 92, 80, 87
118, 92, 90, 83, 78
119, 68, 79, 88, 92
120, 99, 82, 81, 79
121, 96, 85, 91, 60
122, 94, 92, 91, 91
```

以下文本显示了 names.csv 的内容。 此文件表示电子表格，其中包含学生的姓氏、名字和学生 ID。

txt

```
Omelchenko,Svetlana,111
O'Donnell,Claire,112
Mortensen,Sven,113
Garcia,Cesar,114
Garcia,Debra,115
Fakhouri,Fadi,116
Feng,Hanying,117
Garcia,Hugo,118
Tucker,Lance,119
Adams,Terry,120
Zabokritski,Eugene,121
Tucker,Michael,122
```

联接来自包含相关信息的不同文件的内容。 文件 names.csv 包含学生姓名加上 ID 号。 文件 scores.csv 包含 ID 和一组四个测试分数。 以下查询使用 ID 作为匹配关键值将分数联接到学生姓名。 该代码显示在下面的示例中：

C#

```
string[] names = File.ReadAllLines(@"names.csv");
string[] scores = File.ReadAllLines(@"scores.csv");

var scoreQuery = from name in names
                 let nameFields = name.Split(',')
                 from id in scores
                 let scoreFields = id.Split(',')
                 where Convert.ToInt32(nameFields[2]) ==
Convert.ToInt32(scoreFields[0])
                 select $"{nameFields[0]},{scoreFields[1]},
{scoreFields[2]},{scoreFields[3]},{scoreFields[4]}";

Console.WriteLine("\r\nMerge two spreadsheets:");
```

```
foreach (string item in scoreQuery)
{
    Console.WriteLine(item);
}
Console.WriteLine("{0} total names in list", scoreQuery.Count());
/* Output:
Merge two spreadsheets:
Omelchenko, 97, 92, 81, 60
O'Donnell, 75, 84, 91, 39
Mortensen, 88, 94, 65, 91
Garcia, 97, 89, 85, 82
Garcia, 35, 72, 91, 70
Fakhouri, 99, 86, 90, 94
Feng, 93, 92, 80, 87
Garcia, 92, 90, 83, 78
Tucker, 68, 79, 88, 92
Adams, 99, 82, 81, 79
Zabokritski, 96, 85, 91, 60
Tucker, 94, 92, 91, 91
12 total names in list
 */
```

如何在 CSV 文本文件中计算列值

此示例演示如何对 .csv 文件的列执行 Sum、Average、Min 和 Max 等聚合计算。 此处所示的示例原则可以应用于其他类型的结构化文本。

以下文本显示了 scores.csv 的内容。 假定第一列表示学生 ID，后面几列表示四次考试的分数。

txt

```
111, 97, 92, 81, 60
112, 75, 84, 91, 39
113, 88, 94, 65, 91
114, 97, 89, 85, 82
115, 35, 72, 91, 70
116, 99, 86, 90, 94
117, 93, 92, 80, 87
118, 92, 90, 83, 78
119, 68, 79, 88, 92
120, 99, 82, 81, 79
121, 96, 85, 91, 60
122, 94, 92, 91, 91
```

以下文本演示了如何使用 Split 方法将每行文本转换为数组。 每个数组元素表示一列。 最后，每一列中的文本都转换为其数字表示形式。

C#

```
public class SumColumns
{
    public static void SumCSVColumns(string fileName)
    {
        string[] lines = File.ReadAllLines(fileName);

        // Specifies the column to compute.
        int exam = 3;

        // Spreadsheet format:
        // Student ID    Exam#1  Exam#2  Exam#3  Exam#4
        // 111,          97,     92,     81,     60

        // Add one to exam to skip over the first column,
        // which holds the student ID.
        SingleColumn(lines, exam + 1);
        Console.WriteLine();
        MultiColumns(lines);
    }

    static void SingleColumn(IEnumerable<string> strs, int examNum)
    {
        Console.WriteLine("Single Column Query:");

        // Parameter examNum specifies the column to
        // run the calculations on. This value could be
        // passed in dynamically at run time.

        // Variable columnQuery is an IEnumerable<int>.
        // The following query performs two steps:
        // 1) use Split to break each row (a string) into an array
        //    of strings,
        // 2) convert the element at position examNum to an int
        //    and select it.
        var columnQuery = from line in strs
                          let elements = line.Split(',')
                          select Convert.ToInt32(elements[examNum]);

        // Execute the query and cache the results to improve
        // performance. This is helpful only with very large files.
        var results = columnQuery.ToList();

        // Perform aggregate calculations Average, Max, and
        // Min on the column specified by examNum.
        double average = results.Average();
        int max = results.Max();
        int min = results.Min();

        Console.WriteLine($"Exam #{examNum}: Average:{average:##.##} High
Score:{max} Low Score:{min}");
    }

    static void MultiColumns(IEnumerable<string> strs)
    {
```

```
        Console.WriteLine("Multi Column Query:");

        // Create a query, multiColQuery. Explicit typing is used
        // to make clear that, when executed, multiColQuery produces
        // nested sequences. However, you get the same results by
        // using 'var'.

        // The multiColQuery query performs the following steps:
        // 1) use Split to break each row (a string) into an array
        //    of strings,
        // 2) use Skip to skip the "Student ID" column, and store the
        //    rest of the row in scores.
        // 3) convert each score in the current row from a string to
        //    an int, and select that entire sequence as one row
        //    in the results.
        var multiColQuery = from line in strs
                            let elements = line.Split(',')
                            let scores = elements.Skip(1)
                            select (from str in scores
                                    select Convert.ToInt32(str));

        // Execute the query and cache the results to improve
        // performance.
        // ToArray could be used instead of ToList.
        var results = multiColQuery.ToList();

        // Find out how many columns you have in results.
        int columnCount = results[0].Count();

        // Perform aggregate calculations Average, Max, and
        // Min on each column.
        // Perform one iteration of the loop for each column
        // of scores.
        // You can use a for loop instead of a foreach loop
        // because you already executed the multiColQuery
        // query by calling ToList.
        for (int column = 0; column < columnCount; column++)
        {
            var results2 = from row in results
                           select row.ElementAt(column);
            double average = results2.Average();
            int max = results2.Max();
            int min = results2.Min();

            // Add one to column because the first exam is Exam #1,
            // not Exam #0.
            Console.WriteLine($"Exam #{column + 1} Average: {average:##.##}
High Score: {max} Low Score: {min}");
        }
    }
}
/* Output:
    Single Column Query:
    Exam #4: Average:76.92 High Score:94 Low Score:39
```

```
    Multi Column Query:
    Exam #1 Average: 86.08 High Score: 99 Low Score: 35
    Exam #2 Average: 86.42 High Score: 94 Low Score: 72
    Exam #3 Average: 84.75 High Score: 91 Low Score: 65
    Exam #4 Average: 76.92 High Score: 94 Low Score: 39
*/
```

如果文件是制表符分隔文件，只需将 `Split` 方法中的参数更新为 `\t`。

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

如何：使用 LINQ 查询字符串

项目 • 2024/04/27

字符串存储为字符序列。 作为字符序列，可以使用 LINQ 查询它们。 本文中提供了几个示例查询，可查询不同字符或字词的字符串、筛选字符串，或将查询与正则表达式混合。

如何查询字符串中的字符

以下示例查询一个字符串以确定它所包含的数字数量。

C#

```
string aString = "ABCDE99F-J74-12-89A";

// Select only those characters that are numbers
var stringQuery = from ch in aString
                  where Char.IsDigit(ch)
                  select ch;

// Execute the query
foreach (char c in stringQuery)
    Console.Write(c + " ");

// Call the Count method on the existing query.
int count = stringQuery.Count();
Console.WriteLine($"Count = {count}");

// Select all characters before the first '-'
var stringQuery2 = aString.TakeWhile(c => c != '-');

// Execute the second query
foreach (char c in stringQuery2)
    Console.Write(c);
/* Output:
  Output: 9 9 7 4 1 2 8 9
  Count = 8
  ABCDE99F
*/
```

前面的查询演示了如何将字符串视为字符序列。

如何对某个词在字符串中出现的次数计数

以下示例演示如何使用 LINQ 查询对指定词在字符串中出现的次数进行计数。 要执行计数，请首先调用 Split 方法来创建字词数组。 使用 Split 方法会产生性能成本。 如果仅对字符串执行字词计数操作，请考虑改用 Matches 或 IndexOf 方法。

C#

```
string text = """
    Historically, the world of data and the world of objects
    have not been well integrated. Programmers work in C# or Visual Basic
    and also in SQL or XQuery. On the one side are concepts such as classes,
    objects, fields, inheritance, and .NET APIs. On the other side
    are tables, columns, rows, nodes, and separate languages for dealing
with
    them. Data types often require translation between the two worlds; there
are
    different standard functions. Because the object world has no notion of
query, a
    query can only be represented as a string without compile-time type
checking or
    IntelliSense support in the IDE. Transferring data from SQL tables or
XML trees to
    objects in memory is often tedious and error-prone.
    """;

string searchTerm = "data";

//Convert the string into an array of words
char[] separators = ['.', '?', '!', ' ', ';', ':', ','];
string[] source = text.Split(separators,
StringSplitOptions.RemoveEmptyEntries);

// Create the query.  Use the InvariantCultureIgnoreCase comparison to match
"data" and "Data"
var matchQuery = from word in source
                 where word.Equals(searchTerm,
StringComparison.InvariantCultureIgnoreCase)
                 select word;

// Count the matches, which executes the query.
int wordCount = matchQuery.Count();
Console.WriteLine($"""{wordCount} occurrences(s) of the search term "
{searchTerm}" were found.""");
/* Output:
   3 occurrences(s) of the search term "data" were found.
*/
```

前面的查询演示了如何在将字符串拆分为单词序列后将字符串视为单词序列。

如何按任意词或字段对文本数据进行排序或筛选

下面的示例演示如何按行中的任何字段对结构化文本（如以逗号分隔的值）行进行排序。 可以在运行时动态指定该字段。 假定 scores.csv 中的字段表示学生的 ID 号，后跟一系列四个测试分数：

txt

```
111, 97, 92, 81, 60
112, 75, 84, 91, 39
113, 88, 94, 65, 91
114, 97, 89, 85, 82
115, 35, 72, 91, 70
116, 99, 86, 90, 94
117, 93, 92, 80, 87
118, 92, 90, 83, 78
119, 68, 79, 88, 92
120, 99, 82, 81, 79
121, 96, 85, 91, 60
122, 94, 92, 91, 91
```

以下查询根据存储在第二列中的第一个考试分数对行进行排序：

C#

```
// Create an IEnumerable data source
string[] scores = File.ReadAllLines("scores.csv");

// Change this to any value from 0 to 4.
int sortField = 1;

Console.WriteLine($"Sorted highest to lowest by field [{sortField}]:");

// Split the string and sort on field[num]
var scoreQuery = from line in scores
                 let fields = line.Split(',')
                 orderby fields[sortField] descending
                 select line;

foreach (string str in scoreQuery)
{
    Console.WriteLine(str);
}
/* Output (if sortField == 1):
   Sorted highest to lowest by field [1]:
    116, 99, 86, 90, 94
    120, 99, 82, 81, 79
    111, 97, 92, 81, 60
    114, 97, 89, 85, 82
    121, 96, 85, 91, 60
    122, 94, 92, 91, 91
    117, 93, 92, 80, 87
    118, 92, 90, 83, 78
    113, 88, 94, 65, 91
    112, 75, 84, 91, 39
    119, 68, 79, 88, 92
    115, 35, 72, 91, 70
 */
```

前面的查询演示了如何通过将字符串拆分为字段并查询各个字段来操作字符串。

如何查询特定字词的句子

下面的示例显示了如何在文本文件中查找包含指定单词集中每个单词的匹配项的句子。 尽管搜索词数组采用硬编码形式，但也可在运行时以动态方式填充它。 查询将返回包含单词“Historically,”、“data,”和“integrated”的句子。

C#

```
string text = """
Historically, the world of data and the world of objects
have not been well integrated. Programmers work in C# or Visual Basic
and also in SQL or XQuery. On the one side are concepts such as classes,
objects, fields, inheritance, and .NET APIs. On the other side
are tables, columns, rows, nodes, and separate languages for dealing with
them. Data types often require translation between the two worlds; there are
different standard functions. Because the object world has no notion of
query, a
query can only be represented as a string without compile-time type checking
or
IntelliSense support in the IDE. Transferring data from SQL tables or XML
trees to
objects in memory is often tedious and error-prone.
""";

// Split the text block into an array of sentences.
string[] sentences = text.Split(['.', '?', '!']);

// Define the search terms. This list could also be dynamically populated at
run time.
string[] wordsToMatch = [ "Historically", "data", "integrated" ];

// Find sentences that contain all the terms in the wordsToMatch array.
// Note that the number of terms to match is not specified at compile time.
char[] separators = ['.', '?', '!', ' ', ';', ':', ','];
var sentenceQuery = from sentence in sentences
                    let w =
sentence.Split(separators,StringSplitOptions.RemoveEmptyEntries)
                    where w.Distinct().Intersect(wordsToMatch).Count() ==
wordsToMatch.Count()
                    select sentence;

foreach (string str in sentenceQuery)
{
    Console.WriteLine(str);
}
/* Output:
Historically, the world of data and the world of objects have not been well
integrated
*/
```

查询首先将文本拆分为句子，然后将每个句子拆分为包含每个单词的字符串数组。 对于每个数组，Distinct 方法将删除所有重复字词，然后查询将对字词数组和 `wordsToMatch` 数组执行 Intersect 操作。 如果相交数与 `wordsToMatch` 数组的计数相同，将在单词中找到所有单词并返回原始句子。

调用 Split 将使用标点符号作为分隔符，以从字符串中移除它们。 如果未移除标点符号，例如，你可能有一个字符串“Historically”，它不会与 `wordsToMatch` 数组中的“Historically”相匹配。 根据在源文本中找到的标点类型，可能需要使用其他分隔符。

如何将 LINQ 查询与正则表达式合并在一起

此示例显示了如何使用 Regex 类在文本字符串中创建正则表达式来实现更复杂的匹配。 通过 LINQ 查询可以轻松地准确筛选要用正则表达式搜索的文件，并对结果进行改良。

C#

```
string startFolder = """C:\Program Files\dotnet\sdk""";
// Or
// string startFolder = "/usr/local/share/dotnet/sdk";

// Take a snapshot of the file system.
var fileList = from file in Directory.GetFiles(startFolder, "*.*",
SearchOption.AllDirectories)
               let fileInfo = new FileInfo(file)
               select fileInfo;

// Create the regular expression to find all things "Visual".
System.Text.RegularExpressions.Regex searchTerm =
    new System.Text.RegularExpressions.Regex(@"microsoft.net.
(sdk|workload)");

// Search the contents of each .htm file.
// Remove the where clause to find even more matchedValues!
// This query produces a list of files where a match
// was found, and a list of the matchedValues in that file.
// Note: Explicit typing of "Match" in select clause.
// This is required because MatchCollection is not a
// generic IEnumerable collection.
var queryMatchingFiles =
    from file in fileList
    where file.Extension == ".txt"
    let fileText = File.ReadAllText(file.FullName)
    let matches = searchTerm.Matches(fileText)
    where matches.Count > 0
    select new
    {
        name = file.FullName,
        matchedValues = from System.Text.RegularExpressions.Match match in
matches
                        select match.Value
```

```
    };

// Execute the query.
Console.WriteLine($"""The term "{searchTerm}" was found in:""");

foreach (var v in queryMatchingFiles)
{
    // Trim the path a bit, then write
    // the file name in which a match was found.
    string s = v.name.Substring(startFolder.Length - 1);
    Console.WriteLine(s);

    // For this file, write out all the matching strings
    foreach (var v2 in v.matchedValues)
    {
        Console.WriteLine($"  {v2}");
    }
}
```

还可以查询 `RegEx` 搜索返回 MatchCollection 的对象。 结果中仅生成每个匹配项的值。 但也可以使用 LINQ 对该集合执行筛选、排序和分组等各种操作。 由于 MatchCollection 为非泛型 IEnumerable 集合，因此必须显式声明查询中范围变量的类型。

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

.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

LINQ 和集合

项目 • 2024/05/03

大多数集合对元素*序列*建模。 可以使用 LINQ 查询任何集合类型。 其他 LINQ 方法可查找集合中的元素、从集合中的元素计算值，或修改集合或其元素。 这些示例可帮助你了解 LINQ 方法以及如何将其用于集合或其他数据源。

如何查找两个列表之间的差集

此示例演示如何使用 LINQ 对两个字符串列表进行比较，并输出那些位于第一个集合（而不是第二个集合）中的行。 名称的第一个集合存储在文件 *names1.txt* 中：

txt

```
Bankov, Peter
Holm, Michael
Garcia, Hugo
Potra, Cristina
Noriega, Fabricio
Aw, Kam Foo
Beebe, Ann
Toyoshima, Tim
Guy, Wey Yuan
Garcia, Debra
```

名称的第二个集合存储在文件 *names2.txt* 中。 一些名称同时出现在两个序列中。

txt

```
Liu, Jinghao
Bankov, Peter
Holm, Michael
Garcia, Hugo
Beebe, Ann
Gilchrist, Beth
Myrcha, Jacek
Giakoumakis, Leo
McLin, Nkenge
El Yassir, Mehdi
```

以下代码演示如何使用 Enumerable.Except 方法查找在第一个列表中，但不在第二个列表中的元素：

C#

```
// Create the IEnumerable data sources.
string[] names1 = File.ReadAllLines("names1.txt");
string[] names2 = File.ReadAllLines("names2.txt");

// Create the query. Note that method syntax must be used here.
var differenceQuery = names1.Except(names2);

// Execute the query.
Console.WriteLine("The following lines are in names1.txt but not
names2.txt");
foreach (string s in differenceQuery)
    Console.WriteLine(s);
/* Output:
 The following lines are in names1.txt but not names2.txt
 Potra, Cristina
 Noriega, Fabricio
 Aw, Kam Foo
 Toyoshima, Tim
 Guy, Wey Yuan
 Garcia, Debra
 */
```

某些类型的查询操作（例如 Except、Distinct、Union 和 Concat）只能用基于方法的语法表示。

如何合并和比较字符串集合

此示例演示如何合并包含文本行的文件，并对结果排序。 具体而言，此示例演示如何对两组文本行执行串联、联合和交集。 它使用相同的两个文本文件，如前面的示例所示。 该代码显示 Enumerable.Concat、Enumerable.Union 和 Enumerable.Except 的示例。

C#

```
//Put text files in your solution folder
string[] fileA = File.ReadAllLines("names1.txt");
string[] fileB = File.ReadAllLines("names2.txt");

//Simple concatenation and sort. Duplicates are preserved.
var concatQuery = fileA.Concat(fileB).OrderBy(s => s);

// Pass the query variable to another function for execution.
OutputQueryResults(concatQuery, "Simple concatenate and sort. Duplicates are
preserved:");

// Concatenate and remove duplicate names based on
// default string comparer.
var uniqueNamesQuery = fileA.Union(fileB).OrderBy(s => s);
OutputQueryResults(uniqueNamesQuery, "Union removes duplicate names:");
```

```
// Find the names that occur in both files (based on
// default string comparer).
var commonNamesQuery = fileA.Intersect(fileB);
OutputQueryResults(commonNamesQuery, "Merge based on intersect:");

// Find the matching fields in each list. Merge the two
// results by using Concat, and then
// sort using the default string comparer.
string nameMatch = "Garcia";

var tempQuery1 = from name in fileA
                 let n = name.Split(',')
                 where n[0] == nameMatch
                 select name;

var tempQuery2 = from name2 in fileB
                 let n2 = name2.Split(',')
                 where n2[0] == nameMatch
                 select name2;

var nameMatchQuery = tempQuery1.Concat(tempQuery2).OrderBy(s => s);
OutputQueryResults(nameMatchQuery, $"""Concat based on partial name match "
{nameMatch}":""");

static void OutputQueryResults(IEnumerable<string> query, string message)
{
    Console.WriteLine(Environment.NewLine + message);
    foreach (string item in query)
    {
        Console.WriteLine(item);
    }
    Console.WriteLine($"{query.Count()} total names in list");
}
/* Output:
    Simple concatenate and sort. Duplicates are preserved:
    Aw, Kam Foo
    Bankov, Peter
    Bankov, Peter
    Beebe, Ann
    Beebe, Ann
    El Yassir, Mehdi
    Garcia, Debra
    Garcia, Hugo
    Garcia, Hugo
    Giakoumakis, Leo
    Gilchrist, Beth
    Guy, Wey Yuan
    Holm, Michael
    Holm, Michael
    Liu, Jinghao
    McLin, Nkenge
    Myrcha, Jacek
    Noriega, Fabricio
    Potra, Cristina
    Toyoshima, Tim
```

```
    20 total names in list

    Union removes duplicate names:
    Aw, Kam Foo
    Bankov, Peter
    Beebe, Ann
    El Yassir, Mehdi
    Garcia, Debra
    Garcia, Hugo
    Giakoumakis, Leo
    Gilchrist, Beth
    Guy, Wey Yuan
    Holm, Michael
    Liu, Jinghao
    McLin, Nkenge
    Myrcha, Jacek
    Noriega, Fabricio
    Potra, Cristina
    Toyoshima, Tim
    16 total names in list

    Merge based on intersect:
    Bankov, Peter
    Holm, Michael
    Garcia, Hugo
    Beebe, Ann
    4 total names in list

    Concat based on partial name match "Garcia":
    Garcia, Debra
    Garcia, Hugo
    Garcia, Hugo
    3 total names in list
*/
```

如何从多个源填充对象集合

本示例演示如何将来自不同源的数据合并到一系列新的类型。

> **备注**
>
> 请勿尝试将内存中数据或文件系统中的数据与仍在数据库中的数据进行联接。 这种跨域联接可能产生未定义的结果，因为可能为数据库查询和其他类型的源定义了联接操作的不同方式。 此外，如果数据库中的数据量足够大，这样的操作还存在可能导致内存不足的异常的风险。 若要将数据库中的数据联接到内存数据，首先对数据库查询调用 `ToList` 或 `ToArray`，然后对返回的集合执行联接。

此示例使用两个文件。 第一个 *names.csv* 文件包含学生名称和学生 ID。

txt

```
Omelchenko,Svetlana,111
O'Donnell,Claire,112
Mortensen,Sven,113
Garcia,Cesar,114
Garcia,Debra,115
Fakhouri,Fadi,116
Feng,Hanying,117
Garcia,Hugo,118
Tucker,Lance,119
Adams,Terry,120
Zabokritski,Eugene,121
Tucker,Michael,122
```

第二个 *scores.csv* 文件包含第一列中的学生 ID，后跟考试分数。

txt

```
111, 97, 92, 81, 60
112, 75, 84, 91, 39
113, 88, 94, 65, 91
114, 97, 89, 85, 82
115, 35, 72, 91, 70
116, 99, 86, 90, 94
117, 93, 92, 80, 87
118, 92, 90, 83, 78
119, 68, 79, 88, 92
120, 99, 82, 81, 79
121, 96, 85, 91, 60
122, 94, 92, 91, 91
```

下面的示例演示如何使用命名记录 `Student` 存储来自两个内存字符串集合（模拟 .csv 格式的电子表格数据）的合并数据。 该 ID 用作将学生映射到其分数的键。

C#

```
// Each line of names.csv consists of a last name, a first name, and an
// ID number, separated by commas. For example, Omelchenko,Svetlana,111
string[] names = File.ReadAllLines("names.csv");

// Each line of scores.csv consists of an ID number and four test
// scores, separated by commas. For example, 111, 97, 92, 81, 60
string[] scores = File.ReadAllLines("scores.csv");

// Merge the data sources using a named type.
// var could be used instead of an explicit type. Note the dynamic
// creation of a list of ints for the ExamScores member. The first item
// is skipped in the split string because it is the student ID,
// not an exam score.
IEnumerable<Student> queryNamesScores = from nameLine in names
```

```
                                        let splitName = nameLine.Split(',')
                                        from scoreLine in scores
                                        let splitScoreLine =
scoreLine.Split(',')
                                        where Convert.ToInt32(splitName[2])
== Convert.ToInt32(splitScoreLine[0])
                                        select new Student
                                        (
                                            FirstName: splitName[0],
                                            LastName: splitName[1],
                                            ID:
Convert.ToInt32(splitName[2]),
                                            ExamScores: (from scoreAsText in
splitScoreLine.Skip(1)
                                                         select
Convert.ToInt32(scoreAsText)
                                                        ).ToArray()
                                        );

// Optional. Store the newly created student objects in memory
// for faster access in future queries. This could be useful with
// very large data files.
List<Student> students = queryNamesScores.ToList();

// Display each student's name and exam score average.
foreach (var student in students)
{
    Console.WriteLine($"The average score of {student.FirstName}
{student.LastName} is {student.ExamScores.Average()}.");
}
/* Output:
The average score of Omelchenko Svetlana is 82.5.
The average score of O'Donnell Claire is 72.25.
The average score of Mortensen Sven is 84.5.
The average score of Garcia Cesar is 88.25.
The average score of Garcia Debra is 67.
The average score of Fakhouri Fadi is 92.25.
The average score of Feng Hanying is 88.
The average score of Garcia Hugo is 85.75.
The average score of Tucker Lance is 81.75.
The average score of Adams Terry is 85.25.
The average score of Zabokritski Eugene is 83.
The average score of Tucker Michael is 92.
*/
```

在"选择"子句中，将从两个源中的数据初始化每个新的 `Student` 对象。

如果不需要存储查询的结果，那么与命名的类型相比，元组或匿名类型使用起来更方便。下面的示例执行与前面示例相同的任务，但使用的是元组，而不是命名的类型：

```
C#
```

```
// Merge the data sources by using an anonymous type.
// Note the dynamic creation of a list of ints for the
// ExamScores member. We skip 1 because the first string
// in the array is the student ID, not an exam score.
var queryNamesScores2 = from nameLine in names
                        let splitName = nameLine.Split(',')
                        from scoreLine in scores
                        let splitScoreLine = scoreLine.Split(',')
                        where Convert.ToInt32(splitName[2]) ==
Convert.ToInt32(splitScoreLine[0])
                        select (FirstName: splitName[0],
                                LastName: splitName[1],
                                ExamScores: (from scoreAsText in
splitScoreLine.Skip(1)
                                             select
Convert.ToInt32(scoreAsText))
                                             .ToList()
                               );

// Display each student's name and exam score average.
foreach (var student in queryNamesScores2)
{
    Console.WriteLine($"The average score of {student.FirstName}
{student.LastName} is {student.ExamScores.Average()}.");
}
```

如何使用 LINQ 查询 ArrayList

如果使用 LINQ 来查询非泛型 IEnumerable 集合（例如 ArrayList），必须显式声明范围变量的类型，以反映集合中对象的特定类型。 如果有 `Student` 对象的 ArrayList，那么 from 子句应如下所示：

C#

```
var query = from Student s in arrList
//...
```

通过指定范围变量的类型，可将 ArrayList 中的每项强制转换为 `Student`。

在查询表达式中使用显式类型范围变量等效于调用 Cast 方法。 如果无法执行指定的强制转换，Cast 将引发异常。 Cast 和 OfType 是两个标准查询运算符方法，可对非泛型 IEnumerable 类型执行操作。 有关详细信息，请参阅 LINQ 查询操作中的类型关系。 下面的示例演示对 ArrayList 进行查询。

C#

```
ArrayList arrList = new ArrayList();
arrList.Add(
    new Student
    (
        FirstName: "Svetlana",
        LastName: "Omelchenko",
        ExamScores: new int[] { 98, 92, 81, 60 }
    ));
arrList.Add(
    new Student
    (
        FirstName: "Claire",
        LastName: "O'Donnell",
        ExamScores: new int[] { 75, 84, 91, 39 }
    ));
arrList.Add(
    new Student
    (
        FirstName: "Sven",
        LastName: "Mortensen",
        ExamScores: new int[] { 88, 94, 65, 91 }
    ));
arrList.Add(
    new Student
    (
        FirstName: "Cesar",
        LastName: "Garcia",
        ExamScores: new int[] { 97, 89, 85, 82 }
    ));

var query = from Student student in arrList
            where student.ExamScores[0] > 95
            select student;

foreach (Student s in query)
    Console.WriteLine(s.LastName + ": " + s.ExamScores[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 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

如何扩展 LINQ

项目 • 2024/05/03

所有基于 LINQ 的方法都遵循两种类似的模式之一。 它们采用可枚举序列。 它们会返回不同的序列或单个值。 通过形状的一致性，可以通过编写具有类似形状的方法来扩展 LINQ。 事实上，自首次引入 LINQ 以来，.NET 库就在许多 .NET 版本中都获得了新的方法。 在本文中，你将看到通过编写遵循相同模式的自己的方法来扩展 LINQ 的示例。

为 LINQ 查询添加自定义方法

通过向 IEnumerable<T> 接口添加扩展方法扩展可用于 LINQ 查询的方法集。 例如，除了标准平均值或最大值运算，还可创建自定义聚合方法，从一系列值计算单个值。 此外，还可创建一种方法，用作值序列的自定义筛选器或特定数据转换，并返回新的序列。 Distinct、Skip 和 Reverse 就是此类方法的示例。

扩展 IEnumerable<T> 接口时，可以将自定义方法应用于任何可枚举集合。 有关详细信息，请参阅扩展方法。

*聚合*方法可从一组值计算单个值。 LINQ 提供多个聚合方法，包括 Average、Min 和 Max。 可以通过向 IEnumerable<T> 接口添加扩展方法来创建自己的聚合方法。

下面的代码示例演示如何创建名为 `Median` 的扩展方法来计算类型为 `double` 的数字序列的中间值。

C#

```
public static class EnumerableExtension
{
    public static double Median(this IEnumerable<double>? source)
    {
        if (source is null || !source.Any())
        {
            throw new InvalidOperationException("Cannot compute median for a
null or empty set.");
        }

        var sortedList =
            source.OrderBy(number => number).ToList();

        int itemIndex = sortedList.Count / 2;

        if (sortedList.Count % 2 == 0)
        {
            // Even number of items.
            return (sortedList[itemIndex] + sortedList[itemIndex - 1]) / 2;
        }
```

```
        else
        {
            // Odd number of items.
            return sortedList[itemIndex];
        }
    }
}
```

使用从 IEnumerable<T> 接口调用其他聚合方法的方式为任何可枚举集合调用此扩展方法。

下面的代码示例说明如何为类型 `double` 的数组使用 `Median` 方法。

C#

```
double[] numbers = [1.9, 2, 8, 4, 5.7, 6, 7.2, 0];
var query = numbers.Median();

Console.WriteLine($"double: Median = {query}");
// This code produces the following output:
//     double: Median = 4.85
```

可以*重载聚合方法*，以便其接受各种类型的序列。 标准做法是为每种类型都创建一个重载。 另一种方法是创建一个采用泛型类型的重载，并使用委托将其转换为特定类型。 还可以将两种方法结合。

可以为要支持的每种类型创建特定重载。 下面的代码示例演示 `int` 类型的 `Median` 方法的重载。

C#

```
// int overload
public static double Median(this IEnumerable<int> source) =>
    (from number in source select (double)number).Median();
```

现在便可以为 `integer` 和 `double` 类型调用 `Median` 重载了，如以下代码中所示：

C#

```
double[] numbers1 = [1.9, 2, 8, 4, 5.7, 6, 7.2, 0];
var query1 = numbers1.Median();

Console.WriteLine($"double: Median = {query1}");

int[] numbers2 = [1, 2, 3, 4, 5];
var query2 = numbers2.Median();

Console.WriteLine($"int: Median = {query2}");
```

```
// This code produces the following output:
//     double: Median = 4.85
//     int: Median = 3
```

还可以创建接受*泛型对象序列*的重载。 此重载采用委托作为参数，并使用该参数将泛型类型的对象序列转换为特定类型。

下面的代码展示 `Median` 方法的重载，该重载将 Func<T,TResult> 委托作为参数。 此委托采用泛型类型 T 的对象，并返回类型 `double` 的对象。

C#

```
// generic overload
public static double Median<T>(
    this IEnumerable<T> numbers, Func<T, double> selector) =>
    (from num in numbers select selector(num)).Median();
```

现在，可以为任何类型的对象序列调用 `Median` 方法。 如果类型没有它自己的方法重载，必须手动传递委托参数。 在 C# 中，可以使用 lambda 表达式实现此目的。 此外，仅限在 Visual Basic 中，如果使用 `Aggregate` 或 `Group By` 子句而不是方法调用，可以传递此子句范围内的任何值或表达式。

下面的代码示例演示如何为整数数组和字符串数组调用 `Median` 方法。 对于字符串，将计算数组中字符串长度的中值。 该示例演示如何将 Func<T,TResult> 委托参数传递给每个用例的 `Median` 方法。

C#

```
int[] numbers3 = [1, 2, 3, 4, 5];

/*
    You can use the num => num lambda expression as a parameter for the
Median method
    so that the compiler will implicitly convert its value to double.
    If there is no implicit conversion, the compiler will display an error
message.
*/
var query3 = numbers3.Median(num => num);

Console.WriteLine($"int: Median = {query3}");

string[] numbers4 = ["one", "two", "three", "four", "five"];

// With the generic overload, you can also use numeric properties of
objects.
var query4 = numbers4.Median(str => str.Length);

Console.WriteLine($"string: Median = {query4}");
```

```
// This code produces the following output:
//     int: Median = 3
//     string: Median = 4
```

可以使用会返回*值序列*的自定义查询方法来扩展 IEnumerable<T> 接口。 在这种情况下，该方法必须返回类型 IEnumerable<T> 的集合。 此类方法可用于将筛选器或数据转换应用于值序列。

下面的示例演示如何创建名为 `AlternateElements` 的扩展方法，该方法从集合中第一个元素开始按相隔一个元素的方式返回集合中的元素。

C#

```
// Extension method for the IEnumerable<T> interface.
// The method returns every other element of a sequence.
public static IEnumerable<T> AlternateElements<T>(this IEnumerable<T>
source)
{
    int index = 0;
    foreach (T element in source)
    {
        if (index % 2 == 0)
        {
            yield return element;
        }

        index++;
    }
}
```

可使用从 IEnumerable<T> 接口调用其他方法的方式对任何可枚举集合调用此扩展方法，如下面的代码中所示：

C#

```
string[] strings = ["a", "b", "c", "d", "e"];

var query5 = strings.AlternateElements();

foreach (var element in query5)
{
    Console.WriteLine(element);
}
// This code produces the following output:
//     a
//     c
//     e
```

按连续键对结果进行分组

下面的示例演示如何将元素分组为表示连续键子序列的区块。 例如，假设给定下列键值对的序列：

展开表

密钥	值
A	We
A	think
A	that
B	Linq
C	is
A	really
B	cool
B	!

以下组将按此顺序创建：

1. We, think, that
2. Linq
3. is
4. really
5. cool, !

此解决方案是以线程安全扩展方法实现的，该扩展方法以流的方式返回其结果。 换言之，它在源序列中遍历移动时生成其组。 与 `group` 或 `orderby` 运算符不同，它能在读取所有序列之前开始将组返回给调用方。 下面的示例演示该扩展方法以及使用它的客户端代码：

C#

```
public static class ChunkExtensions
{
    public static IEnumerable<IGrouping<TKey, TSource>> ChunkBy<TSource,
TKey>(
            this IEnumerable<TSource> source,
            Func<TSource, TKey> keySelector) =>
                source.ChunkBy(keySelector, EqualityComparer<TKey>.Default);
```

```
    public static IEnumerable<IGrouping<TKey, TSource>> ChunkBy<TSource,
TKey>(
            this IEnumerable<TSource> source,
            Func<TSource, TKey> keySelector,
            IEqualityComparer<TKey> comparer)
    {
        // Flag to signal end of source sequence.
        const bool noMoreSourceElements = true;

        // Auto-generated iterator for the source array.
        IEnumerator<TSource>? enumerator = source.GetEnumerator();

        // Move to the first element in the source sequence.
        if (!enumerator.MoveNext())
        {
            yield break;        // source collection is empty
        }

        while (true)
        {
            var key = keySelector(enumerator.Current);

            Chunk<TKey, TSource> current = new(key, enumerator, value =>
comparer.Equals(key, keySelector(value)));

            yield return current;

            if (current.CopyAllChunkElements() == noMoreSourceElements)
            {
                yield break;
            }
        }
    }
}
```

C#

```
public static class GroupByContiguousKeys
{
    // The source sequence.
    static readonly KeyValuePair<string, string>[] list = [
        new("A", "We"),
        new("A", "think"),
        new("A", "that"),
        new("B", "LINQ"),
        new("C", "is"),
        new("A", "really"),
        new("B", "cool"),
        new("B", "!")
    ];

    // Query variable declared as class member to be available
    // on different threads.
```

```
    static readonly IEnumerable<IGrouping<string, KeyValuePair<string,
string>>> query =
        list.ChunkBy(p => p.Key);

    public static void GroupByContiguousKeys1()
    {
        // ChunkBy returns IGrouping objects, therefore a nested
        // foreach loop is required to access the elements in each "chunk".
        foreach (var item in query)
        {
            Console.WriteLine($"Group key = {item.Key}");
            foreach (var inner in item)
            {
                Console.WriteLine($"\t{inner.Value}");
            }
        }
    }
}
```

`ChunkExtensions` 类

在呈现的 `ChunkExtensions` 类实现代码中，`ChunkBy` 方法中的循环 `while(true)` 循环访问源序列并创建每个区块的副本。 在每次传递中，迭代器前进到源序列的下一个“区块”的第一个元素（由 Chunk 对象代表）。 此循环对应于执行查询的外部 foreach 循环。 在该循环中，代码执行以下操作：

1. 获取当前区块的键并将其分配给 `key` 变量。 源迭代器将遍历源序列，直到找到具有不匹配键的元素。
2. 创建一个新的区块（组）对象，并将其存储在 `current` 变量中。 它具有一个 GroupItem，即当前源元素的副本。
3. 返回该区块。 区块是一个 `IGrouping<TKey,TSource>`，即 ChunkBy 方法的返回值。 区块仅具有其源序列中的第一个元素。 仅当客户端代码 foreach 遍历此区块时，才会返回剩余的元素。 有关详细信息，请参阅 `Chunk.GetEnumerator`。
4. 检查是否存在一下情况：

    - 区块具有其所有源元素的副本，或
    - 迭代器已到达源序列的末尾。

5. 当调用方枚举所有区块项时，`Chunk.GetEnumerator` 方法已复制所有区块项。 如果 `Chunk.GetEnumerator` 循环未枚举区块中的所有元素，则我们需要在此处执行此操作，以避免损坏可能在单独线程上进行调用的客户端的迭代器。

`Chunk` 类

`Chunk` 类是一个或多个具有相同键的源元素的连续组。 区块具有一个键和一个 ChunkItem 对象列表，这些对象是源序列中元素的副本：

C#

```
class Chunk<TKey, TSource> : IGrouping<TKey, TSource>
{
    // INVARIANT: DoneCopyingChunk == true ||
    //   (predicate != null && predicate(enumerator.Current) &&
current.Value == enumerator.Current)

    // A Chunk has a linked list of ChunkItems, which represent the elements
in the current chunk. Each ChunkItem
    // has a reference to the next ChunkItem in the list.
    class ChunkItem
    {
        public ChunkItem(TSource value) => Value = value;
        public readonly TSource Value;
        public ChunkItem? Next;
    }

    public TKey Key { get; }

    // Stores a reference to the enumerator for the source sequence
    private IEnumerator<TSource> enumerator;

    // A reference to the predicate that is used to compare keys.
    private Func<TSource, bool> predicate;

    // Stores the contents of the first source element that
    // belongs with this chunk.
    private readonly ChunkItem head;

    // End of the list. It is repositioned each time a new
    // ChunkItem is added.
    private ChunkItem? tail;

    // Flag to indicate the source iterator has reached the end of the
source sequence.
    internal bool isLastSourceElement;

    // Private object for thread synchronization
    private readonly object m_Lock;

    // REQUIRES: enumerator != null && predicate != null
    public Chunk(TKey key, [DisallowNull] IEnumerator<TSource> enumerator,
[DisallowNull] Func<TSource, bool> predicate)
    {
        Key = key;
        this.enumerator = enumerator;
        this.predicate = predicate;

        // A Chunk always contains at least one element.
        head = new ChunkItem(enumerator.Current);
```

```
        // The end and beginning are the same until the list contains > 1
elements.
        tail = head;

        m_Lock = new object();
    }

    // Indicates that all chunk elements have been copied to the list of
ChunkItems.
    private bool DoneCopyingChunk => tail == null;

    // Adds one ChunkItem to the current group
    // REQUIRES: !DoneCopyingChunk && lock(this)
    private void CopyNextChunkElement()
    {
        // Try to advance the iterator on the source sequence.
        isLastSourceElement = !enumerator.MoveNext();

        // If we are (a) at the end of the source, or (b) at the end of the
current chunk
        // then null out the enumerator and predicate for reuse with the
next chunk.
        if (isLastSourceElement || !predicate(enumerator.Current))
        {
            enumerator = default!;
            predicate = default!;
        }
        else
        {
            tail!.Next = new ChunkItem(enumerator.Current);
        }

        // tail will be null if we are at the end of the chunk elements
        // This check is made in DoneCopyingChunk.
        tail = tail!.Next;
    }

    // Called after the end of the last chunk was reached.
    internal bool CopyAllChunkElements()
    {
        while (true)
        {
            lock (m_Lock)
            {
                if (DoneCopyingChunk)
                {
                    return isLastSourceElement;
                }
                else
                {
                    CopyNextChunkElement();
                }
            }
        }
```

```
    }

    // Stays just one step ahead of the client requests.
    public IEnumerator<TSource> GetEnumerator()
    {
        // Specify the initial element to enumerate.
        ChunkItem? current = head;

        // There should always be at least one ChunkItem in a Chunk.
        while (current != null)
        {
            // Yield the current item in the list.
            yield return current.Value;

            // Copy the next item from the source sequence,
            // if we are at the end of our local list.
            lock (m_Lock)
            {
                if (current == tail)
                {
                    CopyNextChunkElement();
                }
            }

            // Move to the next ChunkItem in the list.
            current = current.Next;
        }
    }

    System.Collections.IEnumerator
System.Collections.IEnumerable.GetEnumerator() => GetEnumerator();
}
```

每个 `ChunkItem`（由 `ChunkItem` 类表示）引用列表中的下一个 `ChunkItem`。 该列表由其 `head`（存储属于此区块的第一个源元素的内容）及其 `tail`（列表的末尾）组成。 每次添加新的 `ChunkItem` 时，都会重新定位尾巴。 如果下一个元素的键与当前区块的键不匹配，或者源中没有更多元素，则链接列表的尾部将设置为 `CopyNextChunkElement` 方法中的 `null`。

`Chunk` 类的 `CopyNextChunkElement` 方法向当前项组添加一个 `ChunkItem`。 它尝试在源序列上递进迭代器。 如果 `MoveNext()` 方法返回 `false`，则表示迭代位于末尾，并且 `isLastSourceElement` 设置为 `true`。

在到达最后一个区块的末尾后调用 `CopyAllChunkElements` 方法。 它首先检查源序列中是否有其他元素。 如果有，在区块的枚举器已耗尽的情况下，此方法将返回 `true`。 在此方法中，当检查专用 `DoneCopyingChunk` 字段是否为 `true` 时，如果 isLastSourceElement 为 `false`，则会向外部迭代器发出信号以继续迭代。

`Chunk` 类的 `GetEnumerator` 方法由内部 foreach 循环调用。 此方法仅领先于客户端请求一个元素。 它仅在客户端请求列表中上一个最后一个元素之后，才添加区块的下一个元素。

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

根据运行时状态进行查询

项目 • 2024/04/26

在大多数 LINQ 查询中，查询的一般形式是在代码中设置的。 可以使用 `where` 子句筛选项、使用 `orderby` 对输出集合进行排序、对项进行分组，或者执行一些计算。 代码可能会提供筛选器参数、排序键或查询中的其他表达式。 但是，查询的一般形式无法更改。 本文介绍使用 System.Linq.IQueryable<T> 接口和实现该接口的类型在运行时修改查询形式所需的技术。

请使用这些技术在运行时构建查询，其中的某些用户输入或运行时状态会更改你要将其用作查询的一部分的查询方法。 你想要通过添加、删除或修改查询子句来编辑查询。

> **备注**
>
> 请确保在 .cs 文件顶部添加 `using System.Linq.Expressions;` 和 `using static System.Linq.Expressions.Expression;`。

考虑针对数据源定义 IQueryable 或 IQueryable<T> 的代码：

C#

```
string[] companyNames = [
    "Consolidated Messenger", "Alpine Ski House", "Southridge Video",
    "City Power & Light", "Coho Winery", "Wide World Importers",
    "Graphic Design Institute", "Adventure Works", "Humongous Insurance",
    "Woodgrove Bank", "Margie's Travel", "Northwind Traders",
    "Blue Yonder Airlines", "Trey Research", "The Phone Company",
    "Wingtip Toys", "Lucerne Publishing", "Fourth Coffee"
];

// Use an in-memory array as the data source, but the IQueryable could have
come
// from anywhere -- an ORM backed by a database, a web request, or any other
LINQ provider.
IQueryable<string> companyNamesSource = companyNames.AsQueryable();
var fixedQry = companyNames.OrderBy(x => x);
```

每次运行前面的代码时，都会执行相同的查询。 让我们学习如何修改查询 - 扩展它或修改它。 从根本上讲，IQueryable 有两个组件：

- Expression - 当前查询的组件的与语言和数据源无关的表示形式，以表达式树的形式表示。
- Provider：LINQ 提供程序的实例，它知道如何将当前查询具体化为一个值或一组值。

在动态查询的上下文中，提供程序通常保持不变；查询的表达式树因查询而异。

表达式树是不可变的；如果需要不同的表达式树并因此需要不同的查询，则需要将现有表达式树转换为新的表达式树。 以下各部分介绍了根据运行时状态，以不同方式进行查询的具体技术：

- 从表达式树中使用运行时状态
- 调用更多 LINQ 方法
- 改变传入到 LINQ 方法的表达式树
- 使用 Expression 中的工厂方法构造 Expression<TDelegate> 表达式树
- 将方法调用节点添加到 IQueryable 的表达式树
- 构造字符串，并使用 动态 LINQ 库

每种技术都可以实现更多功能，但代价是复杂性增加。

从表达式树中使用运行时状态

进行动态查询的最简单方式是通过封闭的变量（如以下代码示例中的 `length`）直接在查询中引用运行时状态：

C#

```
var length = 1;
var qry = companyNamesSource
    .Select(x => x.Substring(0, length))
    .Distinct();

Console.WriteLine(string.Join(",", qry));
// prints: C, A, S, W, G, H, M, N, B, T, L, F

length = 2;
Console.WriteLine(string.Join(",", qry));
// prints: Co, Al, So, Ci, Wi, Gr, Ad, Hu, Wo, Ma, No, Bl, Tr, Th, Lu, Fo
```

内部表达式树以及查询未修改；查询只返回不同的值，因为 `length` 的值已更改。

调用更多 LINQ 方法

通常，Queryable 的内置 LINQ 方法执行两个步骤：

- 在表示方法调用的 MethodCallExpression 中包装当前的表达式树。
- 将包装的表达式树传递回提供程序，以便通过提供程序的 IQueryProvider.Execute 方法返回值；或通过 IQueryProvider.CreateQuery 方法返回转换后的查询对象。

可以将原始查询替换为 System.Linq.IQueryable<T> 返回方法的结果，以获取新的查询。可以使用运行时状态，如以下示例所示：

C#

```
// bool sortByLength = /* ... */;

var qry = companyNamesSource;
if (sortByLength)
{
    qry = qry.OrderBy(x => x.Length);
}
```

改变传入到 LINQ 方法的表达式树

可以将不同的表达式传入到 LINQ 方法，具体取决于运行时状态：

C#

```
// string? startsWith = /* ... */;
// string? endsWith = /* ... */;

Expression<Func<string, bool>> expr = (startsWith, endsWith) switch
{
    ("" or null, "" or null) => x => true,
    (_, "" or null) => x => x.StartsWith(startsWith),
    ("" or null, _) => x => x.EndsWith(endsWith),
    (_, _) => x => x.StartsWith(startsWith) || x.EndsWith(endsWith)
};

var qry = companyNamesSource.Where(expr);
```

你可能还希望使用另一库（如 LinqKit 的 PredicateBuilder）来编写各种子表达式：

C#

```
// This is functionally equivalent to the previous example.

// using LinqKit;
// string? startsWith = /* ... */;
// string? endsWith = /* ... */;

Expression<Func<string, bool>>? expr = PredicateBuilder.New<string>(false);
var original = expr;
if (!string.IsNullOrEmpty(startsWith))
{
    expr = expr.Or(x => x.StartsWith(startsWith));
}
if (!string.IsNullOrEmpty(endsWith))
```

```
{
    expr = expr.Or(x => x.EndsWith(endsWith));
}
if (expr == original)
{
    expr = x => true;
}

var qry = companyNamesSource.Where(expr);
```

使用工厂方法构造表达式树和查询

在到此为止的所有示例中，你知道编译时的元素类型 `string` 并因此知道查询的类型 `IQueryable<string>`。 可以将组件添加到任何元素类型的查询中，或者添加不同的组件，具体取决于元素类型。 可以使用 System.Linq.Expressions.Expression 的工厂方法从头开始创建表达式树，从而在运行时将表达式定制为特定的元素类型。

构造 Expression<TDelegate>

构造要传入到某个 LINQ 方法的表达式时，实际上是在构造 System.Linq.Expressions.Expression<TDelegate> 的实例，其中 `TDelegate` 是某个委托类型，例如 `Func<string, bool>`、`Action` 或自定义委托类型。

System.Linq.Expressions.Expression<TDelegate> 继承自 LambdaExpression，后者表示完整的 lambda 表达式，如以下示例所示：

C#

```
Expression<Func<string, bool>> expr = x => x.StartsWith("a");
```

LambdaExpression 具有两个组件：

1. 参数列表 `(string x)` 由 Parameters 属性表示。
2. 主体 `x.StartsWith("a")` 由 Body 属性表示。

构造 Expression<TDelegate> 的基本步骤如下所示：

1. 使用 Parameter 工厂方法为 lambda 表达式中的每个参数（如果有）定义 ParameterExpression 的对象。

   C#

   ```
   ParameterExpression x = Parameter(typeof(string), "x");
   ```

2. 使用已定义的 ParameterExpression 以及 Expression 中的工厂方法来构造 LambdaExpression 的主体。 例如，表示 `x.StartsWith("a")` 的表达式的构造方式如下：

```
C#

Expression body = Call(
    x,
    typeof(string).GetMethod("StartsWith", [typeof(string)])!,
    Constant("a")
);
```

3. 使用适当的 Lambda 工厂方法重载，将参数和主体包装到编译时类型的 Expression<TDelegate> 中：

```
C#

Expression<Func<string, bool>> expr = Lambda<Func<string, bool>>(body,
x);
```

以下部分介绍了一种方案，在该方案中，你可能需要构造要传递到 LINQ 方法中的 Expression<TDelegate>。 它提供了使用工厂方法完成此操作的完整示例。

在运行时构造完整查询

你想要编写适用于多种实体类型的查询：

```
C#

record Person(string LastName, string FirstName, DateTime DateOfBirth);
record Car(string Model, int Year);
```

对于这些实体类型中的任何一个，你都需要筛选并仅返回那些在其某个 `string` 字段内具有给定文本的实体。 对于 `Person`，你希望搜索 `FirstName` 和 `LastName` 属性：

```
C#

string term = /* ... */;
var personsQry = new List<Person>()
    .AsQueryable()
    .Where(x => x.FirstName.Contains(term) || x.LastName.Contains(term));
```

但对于 `Car`，你希望仅搜索 `Model` 属性：

```
C#
```

```
string term = /* ... */;
var carsQry = new List<Car>()
    .AsQueryable()
    .Where(x => x.Model.Contains(term));
```

尽管可以为 `IQueryable<Person>` 编写一个自定义函数，并为 `IQueryable<Car>` 编写另一个自定义函数，但以下函数会将此筛选添加到任何现有查询，而不考虑特定的元素类型如何。

C#

```
// using static System.Linq.Expressions.Expression;

IQueryable<T> TextFilter<T>(IQueryable<T> source, string term)
{
    if (string.IsNullOrEmpty(term)) { return source; }

    // T is a compile-time placeholder for the element type of the query.
    Type elementType = typeof(T);

    // Get all the string properties on this specific type.
    PropertyInfo[] stringProperties = elementType
        .GetProperties()
        .Where(x => x.PropertyType == typeof(string))
        .ToArray();
    if (!stringProperties.Any()) { return source; }

    // Get the right overload of String.Contains
    MethodInfo containsMethod = typeof(string).GetMethod("Contains",
[typeof(string)])!;

    // Create a parameter for the expression tree:
    // the 'x' in 'x => x.PropertyName.Contains("term")'
    // The type of this parameter is the query's element type
    ParameterExpression prm = Parameter(elementType);

    // Map each property to an expression tree node
    IEnumerable<Expression> expressions = stringProperties
        .Select(prp =>
            // For each property, we have to construct an expression tree
node like x.PropertyName.Contains("term")
            Call(                   // .Contains(...)
                Property(           // .PropertyName
                    prm,            // x
                    prp
                ),
                containsMethod,
                Constant(term)      // "term"
            )
        );
```

```
    // Combine all the resultant expression nodes using ||
    Expression body = expressions
        .Aggregate((prev, current) => Or(prev, current));

    // Wrap the expression body in a compile-time-typed lambda expression
    Expression<Func<T, bool>> lambda = Lambda<Func<T, bool>>(body, prm);

    // Because the lambda is compile-time-typed (albeit with a generic
parameter), we can use it with the Where method
    return source.Where(lambda);
}
```

由于 `TextFilter` 函数采用并返回 IQueryable<T>（而不仅仅是 IQueryable），因此你可以在文本筛选器后添加更多的编译时类型的查询元素。

C#

```
var qry = TextFilter(
        new List<Person>().AsQueryable(),
        "abcd"
    )
    .Where(x => x.DateOfBirth < new DateTime(2001, 1, 1));

var qry1 = TextFilter(
        new List<Car>().AsQueryable(),
        "abcd"
    )
    .Where(x => x.Year == 2010);
```

将方法调用节点添加到 IQueryable<TDelegate> 的表达式树

如果你有 IQueryable（而不是 IQueryable<T>），则不能直接调用泛型 LINQ 方法。 一种替代方法是按上一示例所示构建内部表达式树，并在传入表达树时使用反射来调用适当的 LINQ 方法。

还可以通过在表示调用 LINQ 方法的 MethodCallExpression 中包装整个树来复制 LINQ 方法的功能：

C#

```
IQueryable TextFilter_Untyped(IQueryable source, string term)
{
    if (string.IsNullOrEmpty(term)) { return source; }
    Type elementType = source.ElementType;

    // The logic for building the ParameterExpression and the
LambdaExpression's body is the same as in the previous example,
```

```
    // but has been refactored into the constructBody function.
    (Expression? body, ParameterExpression? prm) =
constructBody(elementType, term);
    if (body is null) { return source; }

    Expression filteredTree = Call(
        typeof(Queryable),
        "Where",
        [elementType],
        source.Expression,
        Lambda(body, prm!)
    );

    return source.Provider.CreateQuery(filteredTree);
}
```

在这种情况下，你没有编译时 `T` 泛型占位符，因此请使用不需要编译时类型信息的 Lambda 重载，这会生成 LambdaExpression，而不是 Expression<TDelegate>。

动态 LINQ 库

使用工厂方法构造表达式树比较复杂；编写字符串较为容易。 动态 LINQ 库 公开了 IQueryable 上的一组扩展方法，这些方法对应于 Queryable 上的标准 LINQ 方法，后者接受采用特殊语法 的字符串而不是表达式树。 该库基于字符串生成相应的表达式树，并可以返回生成的已转换 IQueryable。

例如，可以重新编写上一示例，如下所示：

C#

```
// using System.Linq.Dynamic.Core

IQueryable TextFilter_Strings(IQueryable source, string term)
{
    if (string.IsNullOrEmpty(term)) { return source; }

    var elementType = source.ElementType;

    // Get all the string property names on this specific type.
    var stringProperties =
        elementType.GetProperties()
            .Where(x => x.PropertyType == typeof(string))
            .ToArray();
    if (!stringProperties.Any()) { return source; }

    // Build the string expression
    string filterExpr = string.Join(
        " || ",
        stringProperties.Select(prp => $"{prp.Name}.Contains(@0)")
    );
```

```
    return source.Where(filterExpr, term);
}
```

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

Asynchronous programming with async and await

Article • 09/28/2023

The Task asynchronous programming model (TAP) provides an abstraction over asynchronous code. You write code as a sequence of statements, just like always. You can read that code as though each statement completes before the next begins. The compiler performs many transformations because some of those statements may start work and return a Task that represents the ongoing work.

That's the goal of this syntax: enable code that reads like a sequence of statements, but executes in a much more complicated order based on external resource allocation and when tasks are complete. It's analogous to how people give instructions for processes that include asynchronous tasks. Throughout this article, you'll use an example of instructions for making breakfast to see how the `async` and `await` keywords make it easier to reason about code that includes a series of asynchronous instructions. You'd write the instructions something like the following list to explain how to make a breakfast:

1. Pour a cup of coffee.
2. Heat a pan, then fry two eggs.
3. Fry three slices of bacon.
4. Toast two pieces of bread.
5. Add butter and jam to the toast.
6. Pour a glass of orange juice.

If you have experience with cooking, you'd execute those instructions **asynchronously**. You'd start warming the pan for eggs, then start the bacon. You'd put the bread in the toaster, then start the eggs. At each step of the process, you'd start a task, then turn your attention to tasks that are ready for your attention.

Cooking breakfast is a good example of asynchronous work that isn't parallel. One person (or thread) can handle all these tasks. Continuing the breakfast analogy, one person can make breakfast asynchronously by starting the next task before the first task completes. The cooking progresses whether or not someone is watching it. As soon as you start warming the pan for the eggs, you can begin frying the bacon. Once the bacon starts, you can put the bread into the toaster.

For a parallel algorithm, you'd need multiple cooks (or threads). One would make the eggs, one the bacon, and so on. Each one would be focused on just that one task. Each

cook (or thread) would be blocked synchronously waiting for the bacon to be ready to flip, or the toast to pop.

Now, consider those same instructions written as C# statements:

C#

```
using System;
using System.Threading.Tasks;

namespace AsyncBreakfast
{
    // These classes are intentionally empty for the purpose of this
example. They are simply marker classes for the purpose of demonstration,
contain no properties, and serve no other purpose.
    internal class Bacon { }
    internal class Coffee { }
    internal class Egg { }
    internal class Juice { }
    internal class Toast { }

    class Program
    {
        static void Main(string[] args)
        {
            Coffee cup = PourCoffee();
            Console.WriteLine("coffee is ready");

            Egg eggs = FryEggs(2);
            Console.WriteLine("eggs are ready");

            Bacon bacon = FryBacon(3);
            Console.WriteLine("bacon is ready");

            Toast toast = ToastBread(2);
            ApplyButter(toast);
            ApplyJam(toast);
            Console.WriteLine("toast is ready");

            Juice oj = PourOJ();
            Console.WriteLine("oj is ready");
            Console.WriteLine("Breakfast is ready!");
        }

        private static Juice PourOJ()
        {
            Console.WriteLine("Pouring orange juice");
            return new Juice();
        }

        private static void ApplyJam(Toast toast) =>
            Console.WriteLine("Putting jam on the toast");
```

```
        private static void ApplyButter(Toast toast) =>
            Console.WriteLine("Putting butter on the toast");

        private static Toast ToastBread(int slices)
        {
            for (int slice = 0; slice < slices; slice++)
            {
                Console.WriteLine("Putting a slice of bread in the
toaster");
            }
            Console.WriteLine("Start toasting...");
            Task.Delay(3000).Wait();
            Console.WriteLine("Remove toast from toaster");

            return new Toast();
        }

        private static Bacon FryBacon(int slices)
        {
            Console.WriteLine($"putting {slices} slices of bacon in the
pan");
            Console.WriteLine("cooking first side of bacon...");
            Task.Delay(3000).Wait();
            for (int slice = 0; slice < slices; slice++)
            {
                Console.WriteLine("flipping a slice of bacon");
            }
            Console.WriteLine("cooking the second side of bacon...");
            Task.Delay(3000).Wait();
            Console.WriteLine("Put bacon on plate");

            return new Bacon();
        }

        private static Egg FryEggs(int howMany)
        {
            Console.WriteLine("Warming the egg pan...");
            Task.Delay(3000).Wait();
            Console.WriteLine($"cracking {howMany} eggs");
            Console.WriteLine("cooking the eggs ...");
            Task.Delay(3000).Wait();
            Console.WriteLine("Put eggs on plate");

            return new Egg();
        }

        private static Coffee PourCoffee()
        {
            Console.WriteLine("Pouring coffee");
            return new Coffee();
        }
    }
}
```



The synchronously prepared breakfast took roughly 30 minutes because the total is the sum of each task.

Computers don't interpret those instructions the same way people do. The computer will block on each statement until the work is complete before moving on to the next statement. That creates an unsatisfying breakfast. The later tasks wouldn't be started until the earlier tasks had been completed. It would take much longer to create the breakfast, and some items would have gotten cold before being served.

If you want the computer to execute the above instructions asynchronously, you must write asynchronous code.

These concerns are important for the programs you write today. When you write client programs, you want the UI to be responsive to user input. Your application shouldn't make a phone appear frozen while it's downloading data from the web. When you write server programs, you don't want threads blocked. Those threads could be serving other requests. Using synchronous code when asynchronous alternatives exist hurts your ability to scale out less expensively. You pay for those blocked threads.

Successful modern applications require asynchronous code. Without language support, writing asynchronous code required callbacks, completion events, or other means that obscured the original intent of the code. The advantage of the synchronous code is that

its step-by-step actions make it easy to scan and understand. Traditional asynchronous models forced you to focus on the asynchronous nature of the code, not on the fundamental actions of the code.

Don't block, await instead

The preceding code demonstrates a bad practice: constructing synchronous code to perform asynchronous operations. As written, this code blocks the thread executing it from doing any other work. It won't be interrupted while any of the tasks are in progress. It would be as though you stared at the toaster after putting the bread in. You'd ignore anyone talking to you until the toast popped.

Let's start by updating this code so that the thread doesn't block while tasks are running. The `await` keyword provides a non-blocking way to start a task, then continue execution when that task completes. A simple asynchronous version of the make a breakfast code would look like the following snippet:

C#

```
static async Task Main(string[] args)
{
    Coffee cup = PourCoffee();
    Console.WriteLine("coffee is ready");

    Egg eggs = await FryEggsAsync(2);
    Console.WriteLine("eggs are ready");

    Bacon bacon = await FryBaconAsync(3);
    Console.WriteLine("bacon is ready");

    Toast toast = await ToastBreadAsync(2);
    ApplyButter(toast);
    ApplyJam(toast);
    Console.WriteLine("toast is ready");

    Juice oj = PourOJ();
    Console.WriteLine("oj is ready");
    Console.WriteLine("Breakfast is ready!");
}
```

> **ⓘ Important**
>
> The total elapsed time is roughly the same as the initial synchronous version. The code has yet to take advantage of some of the key features of asynchronous programming.

> **Tip**
>
> The method bodies of the `FryEggsAsync`, `FryBaconAsync`, and `ToastBreadAsync` have all been updated to return `Task<Egg>`, `Task<Bacon>`, and `Task<Toast>` respectively. The methods are renamed from their original version to include the "Async" suffix. Their implementations are shown as part of the **final version** later in this article.

> **Note**
>
> The `Main` method returns `Task`, despite not having a `return` expression—this is by design. For more information, see Evaluation of a void-returning async function.

This code doesn't block while the eggs or the bacon are cooking. This code won't start any other tasks though. You'd still put the toast in the toaster and stare at it until it pops. But at least, you'd respond to anyone that wanted your attention. In a restaurant where multiple orders are placed, the cook could start another breakfast while the first is cooking.

Now, the thread working on the breakfast isn't blocked while awaiting any started task that hasn't yet finished. For some applications, this change is all that's needed. A GUI application still responds to the user with just this change. However, for this scenario, you want more. You don't want each of the component tasks to be executed sequentially. It's better to start each of the component tasks before awaiting the previous task's completion.

Start tasks concurrently

In many scenarios, you want to start several independent tasks immediately. Then, as each task finishes, you can continue other work that's ready. In the breakfast analogy, that's how you get breakfast done more quickly. You also get everything done close to the same time. You'll get a hot breakfast.

The System.Threading.Tasks.Task and related types are classes you can use to reason about tasks that are in progress. That enables you to write code that more closely resembles the way you'd create breakfast. You'd start cooking the eggs, bacon, and toast at the same time. As each requires action, you'd turn your attention to that task, take care of the next action, then wait for something else that requires your attention.

You start a task and hold on to the Task object that represents the work. You'll `await` each task before working with its result.

Let's make these changes to the breakfast code. The first step is to store the tasks for operations when they start, rather than awaiting them:

C#

```
Coffee cup = PourCoffee();
Console.WriteLine("Coffee is ready");

Task<Egg> eggsTask = FryEggsAsync(2);
Egg eggs = await eggsTask;
Console.WriteLine("Eggs are ready");

Task<Bacon> baconTask = FryBaconAsync(3);
Bacon bacon = await baconTask;
Console.WriteLine("Bacon is ready");

Task<Toast> toastTask = ToastBreadAsync(2);
Toast toast = await toastTask;
ApplyButter(toast);
ApplyJam(toast);
Console.WriteLine("Toast is ready");

Juice oj = PourOJ();
Console.WriteLine("Oj is ready");
Console.WriteLine("Breakfast is ready!");
```

The preceding code won't get your breakfast ready any faster. The tasks are all `await`ed as soon as they are started. Next, you can move the `await` statements for the bacon and eggs to the end of the method, before serving breakfast:

C#

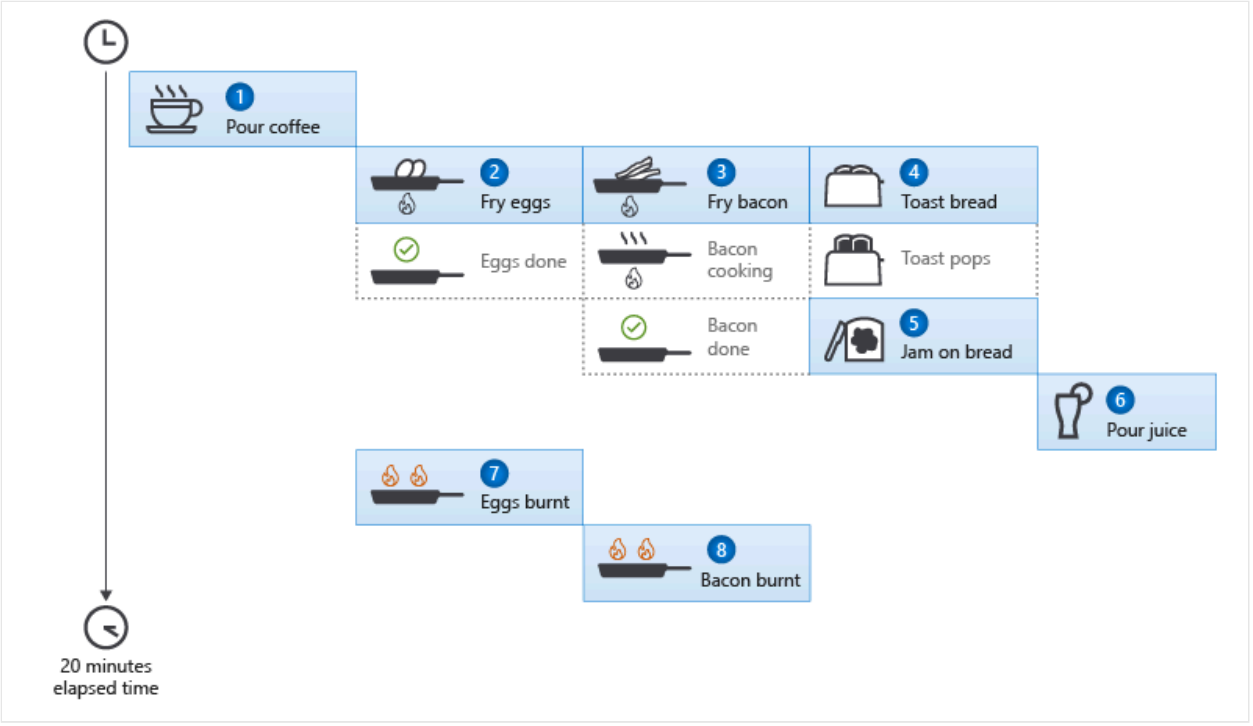
```
Coffee cup = PourCoffee();
Console.WriteLine("Coffee is ready");

Task<Egg> eggsTask = FryEggsAsync(2);
Task<Bacon> baconTask = FryBaconAsync(3);
Task<Toast> toastTask = ToastBreadAsync(2);

Toast toast = await toastTask;
ApplyButter(toast);
ApplyJam(toast);
Console.WriteLine("Toast is ready");
Juice oj = PourOJ();
Console.WriteLine("Oj is ready");

Egg eggs = await eggsTask;
Console.WriteLine("Eggs are ready");
Bacon bacon = await baconTask;
Console.WriteLine("Bacon is ready");
```

```
Console.WriteLine("Breakfast is ready!");
```



The asynchronously prepared breakfast took roughly 20 minutes, this time savings is because some tasks ran concurrently.

The preceding code works better. You start all the asynchronous tasks at once. You await each task only when you need the results. The preceding code may be similar to code in a web application that makes requests to different microservices, then combines the results into a single page. You'll make all the requests immediately, then `await` all those tasks and compose the web page.

Composition with tasks

You have everything ready for breakfast at the same time except the toast. Making the toast is the composition of an asynchronous operation (toasting the bread), and synchronous operations (adding the butter and the jam). Updating this code illustrates an important concept:

> **ⓘ Important**
>
> The composition of an asynchronous operation followed by synchronous work is an asynchronous operation. Stated another way, if any portion of an operation is asynchronous, the entire operation is asynchronous.

The preceding code showed you that you can use Task or Task<TResult> objects to hold running tasks. You `await` each task before using its result. The next step is to create methods that represent the combination of other work. Before serving breakfast, you want to await the task that represents toasting the bread before adding butter and jam. You can represent that work with the following code:

C#

```
static async Task<Toast> MakeToastWithButterAndJamAsync(int number)
{
    var toast = await ToastBreadAsync(number);
    ApplyButter(toast);
    ApplyJam(toast);

    return toast;
}
```

The preceding method has the `async` modifier in its signature. That signals to the compiler that this method contains an `await` statement; it contains asynchronous operations. This method represents the task that toasts the bread, then adds butter and jam. This method returns a Task<TResult> that represents the composition of those three operations. The main block of code now becomes:

C#

```
static async Task Main(string[] args)
{
    Coffee cup = PourCoffee();
    Console.WriteLine("coffee is ready");

    var eggsTask = FryEggsAsync(2);
    var baconTask = FryBaconAsync(3);
    var toastTask = MakeToastWithButterAndJamAsync(2);

    var eggs = await eggsTask;
    Console.WriteLine("eggs are ready");

    var bacon = await baconTask;
    Console.WriteLine("bacon is ready");

    var toast = await toastTask;
    Console.WriteLine("toast is ready");

    Juice oj = PourOJ();
    Console.WriteLine("oj is ready");
    Console.WriteLine("Breakfast is ready!");
}
```

The previous change illustrated an important technique for working with asynchronous code. You compose tasks by separating the operations into a new method that returns a task. You can choose when to await that task. You can start other tasks concurrently.

Asynchronous exceptions

Up to this point, you've implicitly assumed that all these tasks complete successfully. Asynchronous methods throw exceptions, just like their synchronous counterparts. Asynchronous support for exceptions and error handling strives for the same goals as asynchronous support in general: You should write code that reads like a series of synchronous statements. Tasks throw exceptions when they can't complete successfully. The client code can catch those exceptions when a started task is `awaited`. For example, let's assume that the toaster catches fire while making the toast. You can simulate that by modifying the `ToastBreadAsync` method to match the following code:

C#

```
private static async Task<Toast> ToastBreadAsync(int slices)
{
    for (int slice = 0; slice < slices; slice++)
    {
        Console.WriteLine("Putting a slice of bread in the toaster");
    }
    Console.WriteLine("Start toasting...");
    await Task.Delay(2000);
    Console.WriteLine("Fire! Toast is ruined!");
    throw new InvalidOperationException("The toaster is on fire");
    await Task.Delay(1000);
    Console.WriteLine("Remove toast from toaster");

    return new Toast();
}
```

> **Note**
>
> You'll get a warning when you compile the preceding code regarding unreachable code. That's intentional, because once the toaster catches fire, operations won't proceed normally.

Run the application after making these changes, and you'll output similar to the following text:

Console

```
Pouring coffee
Coffee is ready
Warming the egg pan...
putting 3 slices of bacon in the pan
Cooking first side of bacon...
Putting a slice of bread in the toaster
Putting a slice of bread in the toaster
Start toasting...
Fire! Toast is ruined!
Flipping a slice of bacon
Flipping a slice of bacon
Flipping a slice of bacon
Cooking the second side of bacon...
Cracking 2 eggs
Cooking the eggs ...
Put bacon on plate
Put eggs on plate
Eggs are ready
Bacon is ready
Unhandled exception. System.InvalidOperationException: The toaster is on
fire
   at AsyncBreakfast.Program.ToastBreadAsync(Int32 slices) in
Program.cs:line 65
   at AsyncBreakfast.Program.MakeToastWithButterAndJamAsync(Int32 number) in
Program.cs:line 36
   at AsyncBreakfast.Program.Main(String[] args) in Program.cs:line 24
   at AsyncBreakfast.Program.<Main>(String[] args)
```

You'll notice quite a few tasks are completed between when the toaster catches fire and the exception is observed. When a task that runs asynchronously throws an exception, that Task is ***faulted***. The Task object holds the exception thrown in the Task.Exception property. Faulted tasks throw an exception when they're awaited.

There are two important mechanisms to understand: how an exception is stored in a faulted task, and how an exception is unpackaged and rethrown when code awaits a faulted task.

When code running asynchronously throws an exception, that exception is stored in the `Task`. The Task.Exception property is a System.AggregateException because more than one exception may be thrown during asynchronous work. Any exception thrown is added to the AggregateException.InnerExceptions collection. If that `Exception` property is null, a new `AggregateException` is created and the thrown exception is the first item in the collection.

The most common scenario for a faulted task is that the `Exception` property contains exactly one exception. When code `awaits` a faulted task, the first exception in the AggregateException.InnerExceptions collection is rethrown. That's why the output from

this example shows an `InvalidOperationException` instead of an `AggregateException`. Extracting the first inner exception makes working with asynchronous methods as similar as possible to working with their synchronous counterparts. You can examine the `Exception` property in your code when your scenario may generate multiple exceptions.

> **Tip**
>
> We recommend that any argument validation exceptions emerge *synchronously* from task-returning methods. For more information and an example of how to do it, see **Exceptions in task-returning methods**.

Before going on, comment out these two lines in your `ToastBreadAsync` method. You don't want to start another fire:

C#

```
Console.WriteLine("Fire! Toast is ruined!");
throw new InvalidOperationException("The toaster is on fire");
```

Await tasks efficiently

The series of `await` statements at the end of the preceding code can be improved by using methods of the `Task` class. One of those APIs is WhenAll, which returns a Task that completes when all the tasks in its argument list have completed, as shown in the following code:

C#

```
await Task.WhenAll(eggsTask, baconTask, toastTask);
Console.WriteLine("Eggs are ready");
Console.WriteLine("Bacon is ready");
Console.WriteLine("Toast is ready");
Console.WriteLine("Breakfast is ready!");
```

Another option is to use WhenAny, which returns a `Task<Task>` that completes when any of its arguments complete. You can await the returned task, knowing that it has already finished. The following code shows how you could use WhenAny to await the first task to finish and then process its result. After processing the result from the completed task, you remove that completed task from the list of tasks passed to `WhenAny`.

C#

```
var breakfastTasks = new List<Task> { eggsTask, baconTask, toastTask };
while (breakfastTasks.Count > 0)
{
    Task finishedTask = await Task.WhenAny(breakfastTasks);
    if (finishedTask == eggsTask)
    {
        Console.WriteLine("Eggs are ready");
    }
    else if (finishedTask == baconTask)
    {
        Console.WriteLine("Bacon is ready");
    }
    else if (finishedTask == toastTask)
    {
        Console.WriteLine("Toast is ready");
    }
    await finishedTask;
    breakfastTasks.Remove(finishedTask);
}
```

Near the end, you see the line `await finishedTask;`. The line `await Task.WhenAny` doesn't await the finished task. It `await`s the `Task` returned by `Task.WhenAny`. The result of `Task.WhenAny` is the task that has completed (or faulted). You should `await` that task again, even though you know it's finished running. That's how you retrieve its result, or ensure that the exception causing it to fault gets thrown.

After all those changes, the final version of the code looks like this:

C#

```
using System;
using System.Collections.Generic;
using System.Threading.Tasks;

namespace AsyncBreakfast
{
    // These classes are intentionally empty for the purpose of this
example. They are simply marker classes for the purpose of demonstration,
contain no properties, and serve no other purpose.
    internal class Bacon { }
    internal class Coffee { }
    internal class Egg { }
    internal class Juice { }
    internal class Toast { }

    class Program
    {
```

```
        static async Task Main(string[] args)
        {
            Coffee cup = PourCoffee();
            Console.WriteLine("coffee is ready");

            var eggsTask = FryEggsAsync(2);
            var baconTask = FryBaconAsync(3);
            var toastTask = MakeToastWithButterAndJamAsync(2);

            var breakfastTasks = new List<Task> { eggsTask, baconTask,
toastTask };
            while (breakfastTasks.Count > 0)
            {
                Task finishedTask = await Task.WhenAny(breakfastTasks);
                if (finishedTask == eggsTask)
                {
                    Console.WriteLine("eggs are ready");
                }
                else if (finishedTask == baconTask)
                {
                    Console.WriteLine("bacon is ready");
                }
                else if (finishedTask == toastTask)
                {
                    Console.WriteLine("toast is ready");
                }
                await finishedTask;
                breakfastTasks.Remove(finishedTask);
            }

            Juice oj = PourOJ();
            Console.WriteLine("oj is ready");
            Console.WriteLine("Breakfast is ready!");
        }

        static async Task<Toast> MakeToastWithButterAndJamAsync(int number)
        {
            var toast = await ToastBreadAsync(number);
            ApplyButter(toast);
            ApplyJam(toast);

            return toast;
        }

        private static Juice PourOJ()
        {
            Console.WriteLine("Pouring orange juice");
            return new Juice();
        }

        private static void ApplyJam(Toast toast) =>
            Console.WriteLine("Putting jam on the toast");

        private static void ApplyButter(Toast toast) =>
            Console.WriteLine("Putting butter on the toast");
```

```
        private static async Task<Toast> ToastBreadAsync(int slices)
        {
            for (int slice = 0; slice < slices; slice++)
            {
                Console.WriteLine("Putting a slice of bread in the
toaster");
            }
            Console.WriteLine("Start toasting...");
            await Task.Delay(3000);
            Console.WriteLine("Remove toast from toaster");

            return new Toast();
        }

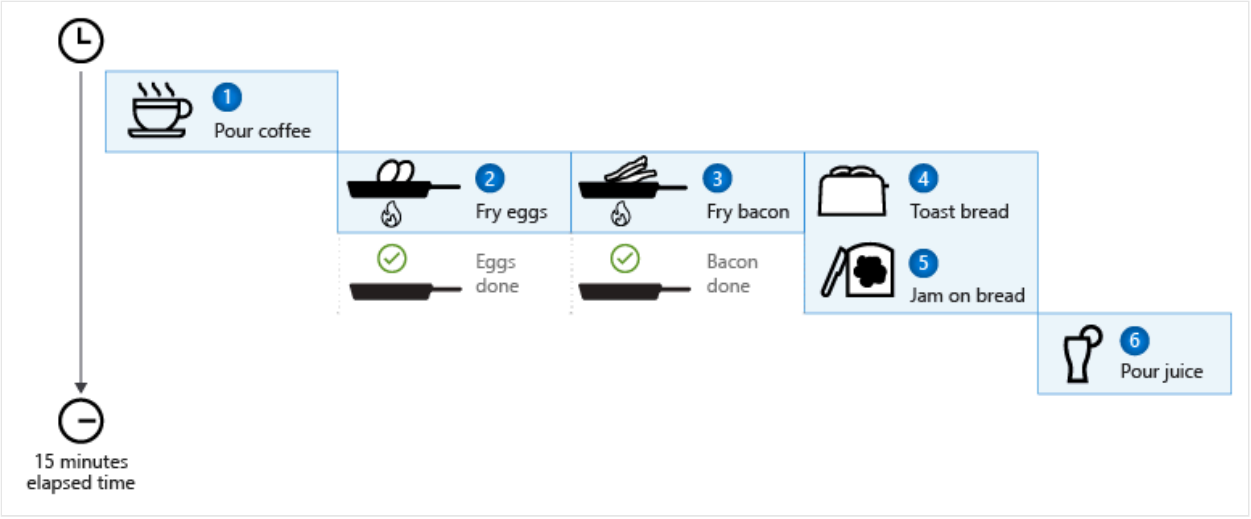
        private static async Task<Bacon> FryBaconAsync(int slices)
        {
            Console.WriteLine($"putting {slices} slices of bacon in the
pan");
            Console.WriteLine("cooking first side of bacon...");
            await Task.Delay(3000);
            for (int slice = 0; slice < slices; slice++)
            {
                Console.WriteLine("flipping a slice of bacon");
            }
            Console.WriteLine("cooking the second side of bacon...");
            await Task.Delay(3000);
            Console.WriteLine("Put bacon on plate");

            return new Bacon();
        }

        private static async Task<Egg> FryEggsAsync(int howMany)
        {
            Console.WriteLine("Warming the egg pan...");
            await Task.Delay(3000);
            Console.WriteLine($"cracking {howMany} eggs");
            Console.WriteLine("cooking the eggs ...");
            await Task.Delay(3000);
            Console.WriteLine("Put eggs on plate");

            return new Egg();
        }

        private static Coffee PourCoffee()
        {
            Console.WriteLine("Pouring coffee");
            return new Coffee();
        }
    }
}
```



The final version of the asynchronously prepared breakfast took roughly 6 minutes because some tasks ran concurrently, and the code monitored multiple tasks at once and only took action when it was needed.

This final code is asynchronous. It more accurately reflects how a person would cook a breakfast. Compare the preceding code with the first code sample in this article. The core actions are still clear from reading the code. You can read this code the same way you'd read those instructions for making a breakfast at the beginning of this article. The language features for `async` and `await` provide the translation every person makes to follow those written instructions: start tasks as you can and don't block waiting for tasks to complete.

Next steps

Explore real world scenarios for asynchronous programs

异步编程

项目 • 2023/10/15

如果需要 I/O 绑定（例如从网络请求数据、访问数据库或读取和写入到文件系统），则需要利用异步编程。 还可以使用 CPU 绑定代码（例如执行成本高昂的计算），对编写异步代码而言，这是一个不错的方案。

C# 拥有语言级别的异步编程模型，让你能轻松编写异步代码，而无需应付回调或受限于支持异步的库。 它遵循基于任务的异步模式 (TAP)。

异步模型概述

异步编程的核心是 `Task` 和 `Task<T>` 对象，这两个对象对异步操作建模。 它们受关键字 `async` 和 `await` 的支持。 在大多数情况下模型十分简单：

- 对于 I/O 绑定代码，等待一个在 `async` 方法中返回 `Task` 或 `Task<T>` 的操作。
- 对于 CPU 绑定代码，等待一个使用 Task.Run 方法在后台线程启动的操作。

`await` 关键字有这奇妙的作用。 它控制执行 `await` 的方法的调用方，且它最终允许 UI 具有响应性或服务具有灵活性。 虽然有方法可处理 `async` 和 `await` 以外的异步代码，但本文重点介绍语言级构造。

> **备注**
>
> 在以下一些示例中，System.Net.Http.HttpClient 类用于从 Web 服务下载某些数据。 这些示例中使用的 `s_httpClient` 对象是 `Program` 类的静态字段（请检查完整示例）：
>
> ```
> private static readonly HttpClient s_httpClient = new();
> ```

I/O 绑定示例：从 Web 服务下载数据

你可能需要在按下按钮时从 Web 服务下载某些数据，但不希望阻止 UI 线程。 可执行如下操作来实现：

C#

```
s_downloadButton.Clicked += async (o, e) =>
{
    // This line will yield control to the UI as the request
    // from the web service is happening.
```

```
    //
    // The UI thread is now free to perform other work.
    var stringData = await s_httpClient.GetStringAsync(URL);
    DoSomethingWithData(stringData);
};
```

代码表示目的（异步下载数据），而不会在与 `Task` 对象的交互中停滞。

CPU 绑定示例：为游戏执行计算

假设你正在编写一个移动游戏，在该游戏中，按下某个按钮将会对屏幕中的许多敌人造成伤害。 执行伤害计算的开销可能极大，而且在 UI 线程中执行计算有可能使游戏在计算执行过程中暂停！

此问题的最佳解决方法是启动一个后台线程，它使用 `Task.Run` 执行工作，并使用 `await` 等待其结果。 这可确保在执行工作时 UI 能流畅运行。

C#

```
static DamageResult CalculateDamageDone()
{
    return new DamageResult()
    {
        // Code omitted:
        //
        // Does an expensive calculation and returns
        // the result of that calculation.
    };
}

s_calculateButton.Clicked += async (o, e) =>
{
    // This line will yield control to the UI while CalculateDamageDone()
    // performs its work. The UI thread is free to perform other work.
    var damageResult = await Task.Run(() => CalculateDamageDone());
    DisplayDamage(damageResult);
};
```

此代码清楚地表达了按钮的单击事件的目的，它无需手动管理后台线程，而是通过非阻止性的方式来实现。

内部原理

在 C# 方面，编译器将代码转换为状态机，它将跟踪类似以下内容：到达 `await` 时暂停执行以及后台作业完成时继续执行。

从理论上讲，这是异步的承诺模型 的实现。

需了解的要点

- 异步代码可用于 I/O 绑定和 CPU 绑定代码，但在每个方案中有所不同。
- 异步代码使用 `Task<T>` 和 `Task`，它们是对后台所完成的工作进行建模的结构。
- `async` 关键字将方法转换为异步方法，这使你能在其正文中使用 `await` 关键字。
- 应用 `await` 关键字后，它将挂起调用方法，并将控制权返还给调用方，直到等待的任务完成。
- 仅允许在异步方法中使用 `await`。

识别 CPU 绑定和 I/O 绑定工作

本指南的前两个示例演示如何将 `async` 和 `await` 用于 I/O 绑定和 CPU 绑定工作。 确定所需执行的操作是 I/O 绑定或 CPU 绑定是关键，因为这会极大影响代码性能，并可能导致某些构造的误用。

以下是编写代码前应考虑的两个问题：

1. 你的代码是否会“等待”某些内容，例如数据库中的数据？

   如果答案为“是”，则你的工作是 **I/O 绑定**。

2. 你的代码是否要执行开销巨大的计算？

   如果答案为“是”，则你的工作是 **CPU 绑定**。

如果你的工作为 **I/O 绑定**，请使用 `async` 和 `await`（而不使用 `Task.Run`）。 不应使用任务并行库。

如果你的工作属于 CPU 绑定，并且你重视响应能力，请使用 `async` 和 `await`，但在另一个线程上使用 `Task.Run` 生成工作。 如果该工作同时适用于并发和并行，还应考虑使用任务并行库。

此外，应始终对代码的执行进行测量。 例如，你可能会遇到这样的情况：多线程处理时，上下文切换的开销高于 CPU 绑定工作的开销。 每种选择都有折衷，应根据自身情况选择正确的折衷方案。

更多示例

下列示例演示了多种使用 C# 编写异步代码的方法。 它们涉及你可能会遇到的一些不同方案。

从网络提取数据

此代码片段从给定的 URL 下载 HTML，并计算 HTML 中字符串".NET"的出现次数。 它使用 ASP.NET 定义 Web API 控制器方法，该方法将执行此任务并返回数字。

> **备注**
>
> 如果打算在生产代码中进行 HTML 分析，则不要使用正则表达式。 改为使用分析库。

C#

```
[HttpGet, Route("DotNetCount")]
static public async Task<int> GetDotNetCount(string URL)
{
    // Suspends GetDotNetCount() to allow the caller (the web server)
    // to accept another request, rather than blocking on this one.
    var html = await s_httpClient.GetStringAsync(URL);
    return Regex.Matches(html, @"\.NET").Count;
}
```

以下是为通用 Windows 应用编写的相同方案，当按下按钮时，它将执行相同的任务：

C#

```
private readonly HttpClient _httpClient = new HttpClient();

private async void OnSeeTheDotNetsButtonClick(object sender, RoutedEventArgs
e)
{
    // Capture the task handle here so we can await the background task
later.
    var getDotNetFoundationHtmlTask =
_httpClient.GetStringAsync("https://dotnetfoundation.org");

    // Any other work on the UI thread can be done here, such as enabling a
Progress Bar.
    // This is important to do here, before the "await" call, so that the
user
    // sees the progress bar before execution of this method is yielded.
    NetworkProgressBar.IsEnabled = true;
    NetworkProgressBar.Visibility = Visibility.Visible;

    // The await operator suspends OnSeeTheDotNetsButtonClick(), returning
control to its caller.
    // This is what allows the app to be responsive and not block the UI
thread.
    var html = await getDotNetFoundationHtmlTask;
    int count = Regex.Matches(html, @"\.NET").Count;

    DotNetCountLabel.Text = $"Number of .NETs on dotnetfoundation.org:
{count}";
```

```
    NetworkProgressBar.IsEnabled = false;
    NetworkProgressBar.Visibility = Visibility.Collapsed;
}
```

等待多个任务完成

你可能发现自己处于需要并行检索多个数据部分的情况。 `Task` API 包含两种方法（即 Task.WhenAll 和 Task.WhenAny），这些方法允许你编写在多个后台作业中执行非阻止等待的异步代码。

此示例演示如何为一组 `User` 捕捉 `userId` 数据。

C#

```
private static async Task<User> GetUserAsync(int userId)
{
    // Code omitted:
    //
    // Given a user Id {userId}, retrieves a User object corresponding
    // to the entry in the database with {userId} as its Id.

    return await Task.FromResult(new User() { id = userId });
}

private static async Task<IEnumerable<User>> GetUsersAsync(IEnumerable<int>
userIds)
{
    var getUserTasks = new List<Task<User>>();
    foreach (int userId in userIds)
    {
        getUserTasks.Add(GetUserAsync(userId));
    }

    return await Task.WhenAll(getUserTasks);
}
```

以下是另一种更简洁的使用 LINQ 进行编写的方法：

C#

```
private static async Task<User[]> GetUsersAsyncByLINQ(IEnumerable<int>
userIds)
{
    var getUserTasks = userIds.Select(id => GetUserAsync(id)).ToArray();
    return await Task.WhenAll(getUserTasks);
}
```

尽管它的代码较少，但在混合 LINQ 和异步代码时需要谨慎使用。 因为 LINQ 使用延迟的执行，因此异步调用将不会像在 `foreach` 循环中那样立刻发生，除非强制所生成的序列通过对 `.ToList()` 或 `.ToArray()` 的调用循环访问。 上述示例使用 Enumerable.ToArray 预先执行查询，并将结果存储在数组中。 这会强制代码 `id => GetUserAsync(id)` 运行并启动任务。

重要信息和建议

对于异步编程，有一些细节需要注意，以防止意外行为。

- **`async` 方法需要在主体中有 `await` 关键字，否则它们将永不暂停！**

  这一点需牢记在心。 如果 `await` 未用在 `async` 方法的主体中，C# 编译器将生成一个警告，但此代码将会以类似普通方法的方式进行编译和运行。 这种方式非常低效，因为由 C# 编译器为异步方法生成的状态机将不会完成任何任务。

- 添加“Async”作为编写的每个异步方法名称的后缀。

  这是 .NET 中的惯例，以便更为轻松地区分同步和异步方法。 未由代码显式调用的某些方法（如事件处理程序或 Web 控制器方法）并不一定适用。 由于它们未由代码显式调用，因此对其显式命名并不重要。

- **`async void` 应仅用于事件处理程序。**

  `async void` 是允许异步事件处理程序工作的唯一方法，因为事件不具有返回类型（因此无法利用 `Task` 和 `Task<T>`）。 其他任何对 `async void` 的使用都不遵循 TAP 模型，且可能存在一定使用难度，例如：
  - `async void` 方法中引发的异常无法在该方法外部被捕获。
  - `async void` 方法很难测试。
  - `async void` 方法可能会导致不良副作用（如果调用方不希望方法是异步的话）。

- **在 LINQ 表达式中使用异步 lambda 时请谨慎**

  LINQ 中的 Lambda 表达式使用延迟执行，这意味着代码可能在你并不希望结束的时候停止执行。 如果编写不正确，将阻塞任务引入其中时可能很容易导致死锁。 此外，此类异步代码嵌套可能会对推断代码的执行带来更多困难。 Async 和 LINQ 的功能都十分强大，但在结合使用两者时应尽可能小心。

- **采用非阻止方式编写等待任务的代码**

  通过阻止当前线程来等待 `Task` 完成的方法可能导致死锁和已阻止的上下文线程，且可能需要更复杂的错误处理方法。 下表提供了关于如何以非阻止方式处理等待任务的指南：

使用以下方式...	而不是...	若要执行此操作...
`await`	`Task.Wait` 或 `Task.Result`	检索后台任务的结果
`await Task.WhenAny`	`Task.WaitAny`	等待任何任务完成
`await Task.WhenAll`	`Task.WaitAll`	等待所有任务完成
`await Task.Delay`	`Thread.Sleep`	等待一段时间

- **如果可能，请考虑使用`ValueTask`**

  从异步方法返回 `Task` 对象可能在某些路径中导致性能瓶颈。 `Task` 是引用类型，因此使用它意味着分配对象。 如果使用 `async` 修饰符声明的方法返回缓存结果或以同步方式完成，那么额外的分配在代码的性能关键部分可能要耗费相当长的时间。 如果这些分配发生在紧凑循环中，则成本会变高。 有关详细信息，请参阅通用的异步返回类型。

- 考虑使用 `ConfigureAwait(false)`

  常见的问题是“应何时使用 Task.ConfigureAwait(Boolean) 方法？”。 该方法允许 `Task` 实例配置其 awaiter。 这是一个重要的注意事项，如果设置不正确，可能会影响性能，甚至造成死锁。 有关 `ConfigureAwait` 的详细信息，请参阅 ConfigureAwait 常见问题解答。

- **编写状态欠缺的代码**

  请勿依赖全局对象的状态或某些方法的执行。 请仅依赖方法的返回值。 为什么？
  - 这样更容易推断代码。
  - 这样更容易测试代码。
  - 混合异步和同步代码更简单。
  - 通常可完全避免争用条件。
  - 通过依赖返回值，协调异步代码可变得简单。
  - （好处）它非常适用于依赖关系注入。

建议的目标是实现代码中完整或接近完整的引用透明度。 这么做能获得可预测、可测试和可维护的代码库。

完整示例

下列代码是示例的 Program.cs 文件的完整文本。

C#

```
using System.Text.RegularExpressions;
using System.Windows;
using Microsoft.AspNetCore.Mvc;

class Button
{
    public Func<object, object, Task>? Clicked
    {
        get;
        internal set;
    }
}

class DamageResult
{
    public int Damage
    {
        get { return 0; }
    }
}

class User
{
    public bool isEnabled
    {
        get;
        set;
    }

    public int id
    {
        get;
        set;
    }
}

public class Program
{
    private static readonly Button s_downloadButton = new();
    private static readonly Button s_calculateButton = new();

    private static readonly HttpClient s_httpClient = new();

    private static readonly IEnumerable<string> s_urlList = new string[]
    {
            "https://learn.microsoft.com",
            "https://learn.microsoft.com/aspnet/core",
            "https://learn.microsoft.com/azure",
            "https://learn.microsoft.com/azure/devops",
            "https://learn.microsoft.com/dotnet",
            "https://learn.microsoft.com/dotnet/desktop/wpf/get-
started/create-app-visual-studio",
            "https://learn.microsoft.com/education",
            "https://learn.microsoft.com/shows/net-core-101/what-is-net",
```

```
            "https://learn.microsoft.com/enterprise-mobility-security",
            "https://learn.microsoft.com/gaming",
            "https://learn.microsoft.com/graph",
            "https://learn.microsoft.com/microsoft-365",
            "https://learn.microsoft.com/office",
            "https://learn.microsoft.com/powershell",
            "https://learn.microsoft.com/sql",
            "https://learn.microsoft.com/surface",
            "https://dotnetfoundation.org",
            "https://learn.microsoft.com/visualstudio",
            "https://learn.microsoft.com/windows",
            "https://learn.microsoft.com/xamarin"
    };

    private static void Calculate()
    {
        // <PerformGameCalculation>
        static DamageResult CalculateDamageDone()
        {
            return new DamageResult()
            {
                // Code omitted:
                //
                // Does an expensive calculation and returns
                // the result of that calculation.
            };
        }

        s_calculateButton.Clicked += async (o, e) =>
        {
            // This line will yield control to the UI while
CalculateDamageDone()
            // performs its work. The UI thread is free to perform other
work.
            var damageResult = await Task.Run(() => CalculateDamageDone());
            DisplayDamage(damageResult);
        };
        // </PerformGameCalculation>
    }

    private static void DisplayDamage(DamageResult damage)
    {
        Console.WriteLine(damage.Damage);
    }

    private static void Download(string URL)
    {
        // <UnblockingDownload>
        s_downloadButton.Clicked += async (o, e) =>
        {
            // This line will yield control to the UI as the request
            // from the web service is happening.
            //
            // The UI thread is now free to perform other work.
            var stringData = await s_httpClient.GetStringAsync(URL);
```

```
            DoSomethingWithData(stringData);
        };
        // </UnblockingDownload>
    }

    private static void DoSomethingWithData(object stringData)
    {
        Console.WriteLine("Displaying data: ", stringData);
    }

    // <GetUsersForDataset>
    private static async Task<User> GetUserAsync(int userId)
    {
        // Code omitted:
        //
        // Given a user Id {userId}, retrieves a User object corresponding
        // to the entry in the database with {userId} as its Id.

        return await Task.FromResult(new User() { id = userId });
    }

    private static async Task<IEnumerable<User>>
GetUsersAsync(IEnumerable<int> userIds)
    {
        var getUserTasks = new List<Task<User>>();
        foreach (int userId in userIds)
        {
            getUserTasks.Add(GetUserAsync(userId));
        }

        return await Task.WhenAll(getUserTasks);
    }
    // </GetUsersForDataset>

    // <GetUsersForDatasetByLINQ>
    private static async Task<User[]> GetUsersAsyncByLINQ(IEnumerable<int>
userIds)
    {
        var getUserTasks = userIds.Select(id => GetUserAsync(id)).ToArray();
        return await Task.WhenAll(getUserTasks);
    }
    // </GetUsersForDatasetByLINQ>

    // <ExtractDataFromNetwork>
    [HttpGet, Route("DotNetCount")]
    static public async Task<int> GetDotNetCount(string URL)
    {
        // Suspends GetDotNetCount() to allow the caller (the web server)
        // to accept another request, rather than blocking on this one.
        var html = await s_httpClient.GetStringAsync(URL);
        return Regex.Matches(html, @"\.NET").Count;
    }
    // </ExtractDataFromNetwork>

    static async Task Main()
```

```
    {
        Console.WriteLine("Application started.");

        Console.WriteLine("Counting '.NET' phrase in websites...");
        int total = 0;
        foreach (string url in s_urlList)
        {
            var result = await GetDotNetCount(url);
            Console.WriteLine($"{url}: {result}");
            total += result;
        }
        Console.WriteLine("Total: " + total);

        Console.WriteLine("Retrieving User objects with list of IDs...");
        IEnumerable<int> ids = new int[] { 1, 2, 3, 4, 5, 6, 7, 8, 9, 0 };
        var users = await GetUsersAsync(ids);
        foreach (User? user in users)
        {
            Console.WriteLine($"{user.id}: isEnabled={user.isEnabled}");
        }

        Console.WriteLine("Application ending.");
    }
}

// Example output:
//
// Application started.
// Counting '.NET' phrase in websites...
// https://learn.microsoft.com: 0
// https://learn.microsoft.com/aspnet/core: 57
// https://learn.microsoft.com/azure: 1
// https://learn.microsoft.com/azure/devops: 2
// https://learn.microsoft.com/dotnet: 83
// https://learn.microsoft.com/dotnet/desktop/wpf/get-started/create-app-
visual-studio: 31
// https://learn.microsoft.com/education: 0
// https://learn.microsoft.com/shows/net-core-101/what-is-net: 42
// https://learn.microsoft.com/enterprise-mobility-security: 0
// https://learn.microsoft.com/gaming: 0
// https://learn.microsoft.com/graph: 0
// https://learn.microsoft.com/microsoft-365: 0
// https://learn.microsoft.com/office: 0
// https://learn.microsoft.com/powershell: 0
// https://learn.microsoft.com/sql: 0
// https://learn.microsoft.com/surface: 0
// https://dotnetfoundation.org: 16
// https://learn.microsoft.com/visualstudio: 0
// https://learn.microsoft.com/windows: 0
// https://learn.microsoft.com/xamarin: 6
// Total: 238
// Retrieving User objects with list of IDs...
// 1: isEnabled= False
// 2: isEnabled= False
// 3: isEnabled= False
```

```
// 4: isEnabled= False
// 5: isEnabled= False
// 6: isEnabled= False
// 7: isEnabled= False
// 8: isEnabled= False
// 9: isEnabled= False
// 0: isEnabled= False
// Application ending.
```

其他资源

- 基于任务的异步编程模型 (C#)。

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

异步编程模型

项目 • 2023/03/28

通过使用异步编程，你可以避免性能瓶颈并增强应用程序的总体响应能力。 但是，编写异步应用程序的传统技术可能比较复杂，使它们难以编写、调试和维护。

C# 支持简化的方法，即异步编程，它利用 .NET 运行时中的异步支持。 编译器可执行开发人员曾进行的高难度工作，且应用程序保留了一个类似于同步代码的逻辑结构。 因此，你只需做一小部分工作就可以获得异步编程的所有好处。

本主题概述了何时以及如何使用异步编程，并包括指向包含详细信息和示例的支持主题的链接。

异步编程提升响应能力

异步对可能会被屏蔽的活动（如 Web 访问）至关重要。 对 Web 资源的访问有时很慢或会延迟。 如果此类活动在同步过程中被屏蔽，整个应用必须等待。 在异步过程中，应用程序可继续执行不依赖 Web 资源的其他工作，直至潜在阻止任务完成。

下表显示了异步编程提高响应能力的典型区域。 列出的 .NET 和 Windows 运行时 API 包含支持异步编程的方法。

展开表

应用程序区域	包含异步方法的 .NET 类型	包含异步方法的 Windows 运行时类型
Web 访问	HttpClient	Windows.Web.Http.HttpClient SyndicationClient
使用文件	JsonSerializer StreamReader StreamWriter XmlReader XmlWriter	StorageFile
使用图像		MediaCapture BitmapEncoder BitmapDecoder
WCF 编程	同步和异步操作	

由于所有与用户界面相关的活动通常共享一个线程，因此，异步对访问 UI 线程的应用程序来说尤为重要。 如果任何进程在同步应用程序中受阻，则所有进程都将受阻。 你的应用程序停止响应，因此，你可能在其等待过程中认为它已经失败。

使用异步方法时，应用程序将继续响应 UI。 例如，你可以调整窗口的大小或最小化窗口；如果你不希望等待应用程序结束，则可以将其关闭。

当设计异步操作时，该基于异步的方法将自动传输的等效对象添加到可从中选择的选项列表中。 开发人员只需要投入较少的工作量即可使你获取传统异步编程的所有优点。

异步方法易于编写

C# 中的 Async 和 Await 关键字是异步编程的核心。 通过这两个关键字，可以使用 .NET Framework、.NET Core 或 Windows 运行时中的资源，轻松创建异步方法（几乎与创建同步方法一样轻松）。 使用 `async` 关键字定义的异步方法简称为“异步方法”。

下面的示例演示了一种异步方法。 你应对代码中的几乎所有内容都很熟悉。

可从 C# 中使用 Async 和 Await 的异步编程中找到可供下载的完整 Windows Presentation Foundation (WPF) 示例。

C#

```
public async Task<int> GetUrlContentLengthAsync()
{
    using var client = new HttpClient();

    Task<string> getStringTask =
        client.GetStringAsync("https://learn.microsoft.com/dotnet");

    DoIndependentWork();

    string contents = await getStringTask;

    return contents.Length;
}

void DoIndependentWork()
{
    Console.WriteLine("Working...");
}
```

可以从前面的示例中了解几种做法。 从方法签名开始。 它包含 `async` 修饰符。 返回类型为 `Task<int>`（有关更多选项，请参阅“返回类型”部分）。 方法名称以 `Async` 结尾。 在方法的主体中，`GetStringAsync` 返回 `Task<string>`。 这意味着在 `await` 任务时，将获得 `string`（`contents`）。 在等待任务之前，可以通过 `GetStringAsync` 执行不依赖于 `string` 的工作。

密切注意 `await` 运算符。 它会暂停 `GetUrlContentLengthAsync`：

- 在 `getStringTask` 完成之前，`GetUrlContentLengthAsync` 无法继续。
- 同时，控件返回至 `GetUrlContentLengthAsync` 的调用方。
- 当 `getStringTask` 完成时，控件将在此处继续。
- 然后，`await` 运算符会从 `getStringTask` 检索 `string` 结果。

return 语句指定整数结果。 任何等待 `GetUrlContentLengthAsync` 的方法都会检索长度值。

如果 `GetUrlContentLengthAsync` 在调用 `GetStringAsync` 和等待其完成期间不能进行任何工作，则你可以通过在下面的单个语句中调用和等待来简化代码。

```
C#
```

```
string contents = await
client.GetStringAsync("https://learn.microsoft.com/dotnet");
```

以下特征总结了使上一个示例成为异步方法的原因：

- 方法签名包含 `async` 修饰符。
- 按照约定，异步方法的名称以“Async”后缀结尾。
- 返回类型为下列类型之一：
  - 如果你的方法有操作数为 `TResult` 类型的返回语句，则为 Task<TResult>。
  - 如果你的方法没有返回语句或具有没有操作数的返回语句，则为 Task。
  - `void`：如果要编写异步事件处理程序。
  - 具有 `GetAwaiter` 方法的任何其他类型。

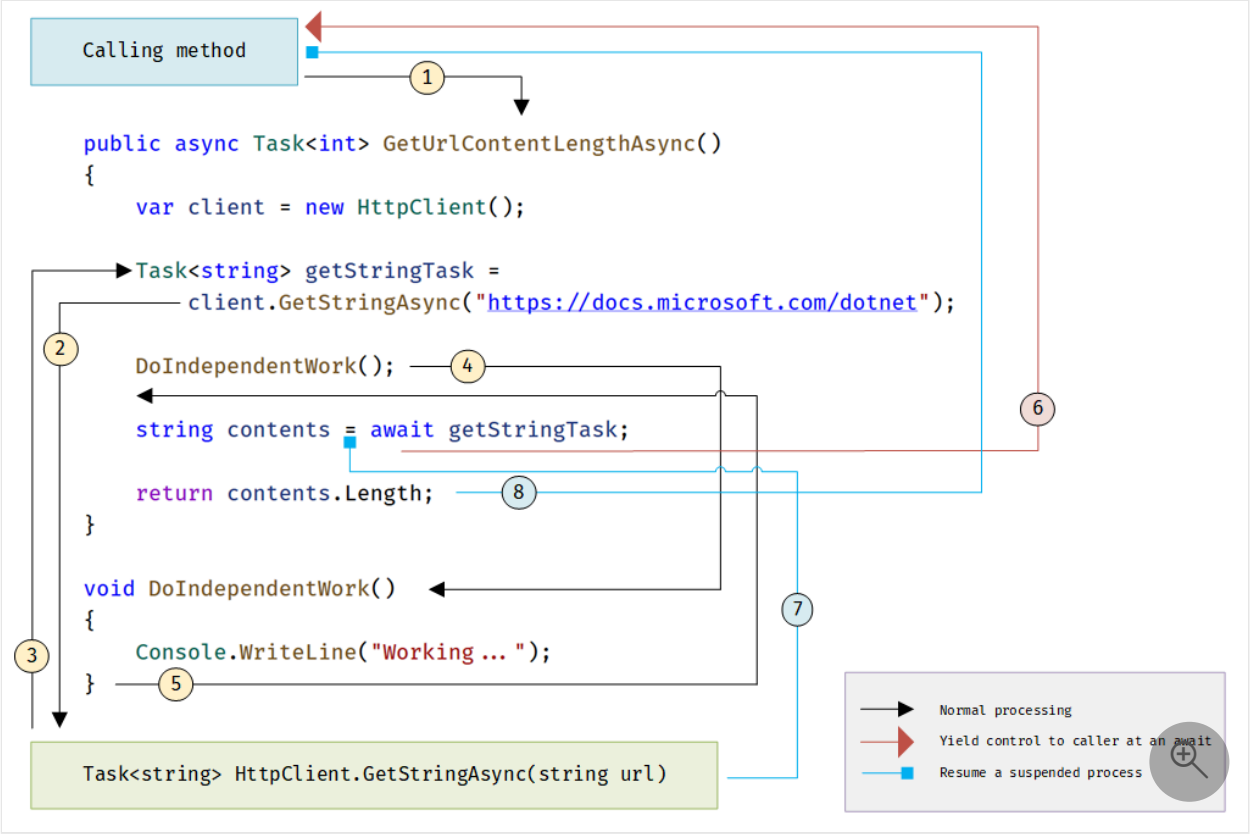
  有关详细信息，请参阅返回类型和参数部分。
- 方法通常包含至少一个 `await` 表达式，该表达式标记一个点，在该点上，直到等待的异步操作完成方法才能继续。 同时，将方法挂起，并且控件返回到方法的调用方。 本主题的下一节将解释悬挂点发生的情况。

在异步方法中，可使用提供的关键字和类型来指示需要完成的操作，且编译器会完成其余操作，其中包括持续跟踪控件以挂起方法返回等待点时发生的情况。 一些常规流程（例如，循环和异常处理）在传统异步代码中处理起来可能很困难。 在异步方法中，元素的编写频率与同步解决方案相同且此问题得到解决。

若要详细了解旧版 .NET Framework 中的异步性，请参阅 TPL 和传统 .NET Framework 异步编程。

异步方法的运行机制

异步编程中最需弄清的是控制流是如何从方法移动到方法的。 下图可引导你完成此过程：



关系图中的数字对应于以下步骤，在调用方法调用异步方法时启动。

1. 调用方法调用并等待 `GetUrlContentLengthAsync` 异步方法。

2. `GetUrlContentLengthAsync` 可创建 HttpClient 实例并调用 GetStringAsync 异步方法以下载网站内容作为字符串。

3. `GetStringAsync` 中发生了某种情况，该情况挂起了它的进程。 可能必须等待网站下载或一些其他阻止活动。 为避免阻止资源，`GetStringAsync` 会将控制权出让给其调用方 `GetUrlContentLengthAsync`。

   `GetStringAsync` 返回 Task<TResult>，其中 `TResult` 为字符串，并且 `GetUrlContentLengthAsync` 将任务分配给 `getStringTask` 变量。 该任务表示调用 `GetStringAsync` 的正在进行的进程，其中承诺当工作完成时产生实际字符串值。

4. 由于尚未等待 `getStringTask`，因此，`GetUrlContentLengthAsync` 可以继续执行不依赖于 `GetStringAsync` 得出的最终结果的其他工作。 该任务由对同步方法 `DoIndependentWork` 的调用表示。

5. `DoIndependentWork` 是完成其工作并返回其调用方的同步方法。

6. `GetUrlContentLengthAsync` 已运行完毕，可以不受 `getStringTask` 的结果影响。 接下来，`GetUrlContentLengthAsync` 需要计算并返回已下载的字符串的长度，但该方法只有在获得字符串的情况下才能计算该值。

   因此，`GetUrlContentLengthAsync` 使用一个 await 运算符来挂起其进度，并把控制权交给调用 `GetUrlContentLengthAsync` 的方法。 `GetUrlContentLengthAsync` 将 `Task<int>` 返回给调用方。 该任务表示对产生下载字符串长度的整数结果的一个承诺。

   > **备注**
   >
   > 如果 `GetStringAsync`（因此 `getStringTask`）在 `GetUrlContentLengthAsync` 等待前完成，则控制会保留在 `GetUrlContentLengthAsync` 中。 如果异步调用过程 `getStringTask` 已完成，并且 `GetUrlContentLengthAsync` 不必等待最终结果，则挂起然后返回到 `GetUrlContentLengthAsync` 将造成成本浪费。

   在调用方法中，处理模式会继续。 在等待结果前，调用方可以开展不依赖于 `GetUrlContentLengthAsync` 结果的其他工作，否则就需等待片刻。 调用方法等待 `GetUrlContentLengthAsync`，而 `GetUrlContentLengthAsync` 等待 `GetStringAsync`。

7. `GetStringAsync` 完成并生成一个字符串结果。 字符串结果不是通过按你预期的方式调用 `GetStringAsync` 所返回的。 （记住，该方法已返回步骤 3 中的一个任务）。 相反，字符串结果存储在表示 `getStringTask` 方法完成的任务中。 await 运算符从 `getStringTask` 中检索结果。 赋值语句将检索到的结果赋给 `contents`。

8. 当 `GetUrlContentLengthAsync` 具有字符串结果时，该方法可以计算字符串长度。 然后，`GetUrlContentLengthAsync` 工作也将完成，并且等待事件处理程序可继续使用。 在此主题结尾处的完整示例中，可确认事件处理程序检索并打印长度结果的值。 如果你不熟悉异步编程，请花 1 分钟时间考虑同步行为和异步行为之间的差异。 当其工作完成时（第 5 步）会返回一个同步方法，但当其工作挂起时（第 3 步和第 6 步），异步方法会返回一个任务值。 在异步方法最终完成其工作时，任务会标记为已完成，而结果（如果有）将存储在任务中。

API 异步方法

你可能想知道从何处可以找到 `GetStringAsync` 等支持异步编程的方法。 .NET Framework 4.5 或更高版本以及 .NET Core 包含许多可与 `async` 和 `await` 结合使用的成员。 可以通过追加到成员名称的“Async”后缀和 Task 或 Task<TResult> 的返回类型，识别这些成员。 例如，`System.IO.Stream` 类包含 CopyToAsync、ReadAsync 和 WriteAsync 等方法，以及同步方法 CopyTo、Read 和 Write。

Windows 运行时也包含许多可以在 Windows 应用中与 `async` 和 `await` 结合使用的方法。 有关详细信息，请参阅线程处理和异步编程进行 UWP 开发；如果使用的是旧版 Windows 运行时，还请参阅异步编程（Windows 应用商店应用）和快速入门：在 C# 或 Visual Basic 中调用异步 API。

线程

异步方法旨在成为非阻止操作。 异步方法中的 `await` 表达式在等待的任务正在运行时不会阻止当前线程。 相反，表达式在继续时注册方法的其余部分并将控件返回到异步方法的调用方。

`async` 和 `await` 关键字不会创建其他线程。 因为异步方法不会在其自身线程上运行，因此它不需要多线程。 只有当方法处于活动状态时，该方法将在当前同步上下文中运行并使用线程上的时间。 可以使用 Task.Run 将占用大量 CPU 的工作移到后台线程，但是后台线程不会帮助正在等待结果的进程变为可用状态。

对于异步编程而言，该基于异步的方法优于几乎每个用例中的现有方法。 具体而言，此方法比 BackgroundWorker 类更适用于 I/O 绑定操作，因为此代码更简单且无需防止争用条件。 结合 Task.Run 方法使用时，异步编程比 BackgroundWorker 更适用于 CPU 绑定操作，因为异步编程将运行代码的协调细节与 `Task.Run` 传输至线程池的工作区分开来。

async 和 await

如果使用 async 修饰符将某种方法指定为异步方法，即启用以下两种功能。

- 标记的异步方法可以使用 await 来指定暂停点。 `await` 运算符通知编译器异步方法：在等待的异步过程完成后才能继续通过该点。 同时，控制返回至异步方法的调用方。

  异步方法在 `await` 表达式执行时暂停并不构成方法退出，只会导致 `finally` 代码块不运行。

- 标记的异步方法本身可以通过调用它的方法等待。

异步方法通常包含 `await` 运算符的一个或多个实例，但缺少 `await` 表达式也不会导致生成编译器错误。 如果异步方法未使用 `await` 运算符标记暂停点，则该方法会作为同步方法执行，即使有 `async` 修饰符，也不例外。 编译器将为此类方法发布一个警告。

`async` 和 `await` 都是上下文关键字。 有关更多信息和示例，请参见以下主题：

- async
- await

返回类型和参数

异步方法通常返回 Task 或 Task<TResult>。 在异步方法中，`await` 运算符应用于通过调用另一个异步方法返回的任务。

如果方法包含指定 `TResult` 类型操作数的 return 语句，将 Task<TResult> 指定为返回类型。

如果方法不含任何 return 语句或包含不返回操作数的 return 语句，则将 Task 用作返回类型。

还可以指定任何其他返回类型，前提是类型包含 `GetAwaiter` 方法。 例如，ValueTask<TResult> 就是这种类型。 可用于 System.Threading.Tasks.Extension NuGet 包。

下面的示例演示如何声明并调用可返回 Task<TResult> 或 Task 的方法：

C#

```
async Task<int> GetTaskOfTResultAsync()
{
    int hours = 0;
    await Task.Delay(0);

    return hours;
}


Task<int> returnedTaskTResult = GetTaskOfTResultAsync();
int intResult = await returnedTaskTResult;
// Single line
// int intResult = await GetTaskOfTResultAsync();

async Task GetTaskAsync()
{
    await Task.Delay(0);
    // No return statement needed
}

Task returnedTask = GetTaskAsync();
await returnedTask;
// Single line
await GetTaskAsync();
```

每个返回的任务表示正在进行的工作。 任务可封装有关异步进程状态的信息，如果未成功，则最后会封装来自进程的最终结果或进程引发的异常。

异步方法也可以具有 `void` 返回类型。 该返回类型主要用于定义需要 `void` 返回类型的事件处理程序。 异步事件处理程序通常用作异步程序的起始点。

无法等待具有 `void` 返回类型的异步方法，并且无效返回方法的调用方捕获不到异步方法引发的任何异常。

异步方法无法声明 in、ref 或 out 参数，但可以调用包含此类参数的方法。 同样，异步方法无法通过引用返回值，但可以调用包含 ref 返回值的方法。

有关详细信息和示例，请参阅异步返回类型 (C#)。

Windows 运行时编程中的异步 API 具有下列返回类型之一（类似于任务）：

- IAsyncOperation<TResult>（对应于 Task<TResult>）
- IAsyncAction（对应于 Task）
- IAsyncActionWithProgress<TProgress>
- IAsyncOperationWithProgress<TResult,TProgress>

命名约定

按照约定，返回常规可等待类型的方法（例如 `Task`、`Task<T>`、`ValueTask` 和 `ValueTask<T>`）应具有以“Async”结束的名称。 启动异步操作但不返回可等待类型的方法不得具有以“Async”结尾的名称，但其开头可以为“Begin”、“Start”或其他表明此方法不返回或引发操作结果的动词。

如果某一约定中的事件、基类或接口协定建议其他名称，则可以忽略此约定。 例如，你不应重命名常用事件处理程序，例如 `OnButtonClick`。

相关文章 (Visual Studio)

展开表

标题	说明
如何使用 Async 和 Await 并行发出多个 Web 请求 (C#)	演示如何同时开始几个任务。
异步返回类型 (C#)	描述异步方法可返回的类型，并解释每种类型适用于的情况。
使用取消令牌作为信号机制来取消任务。	演示如何将以下功能添加到异步解决方案： - 取消任务列表 (C#)

标题	说明
	- 在一段时间后取消任务 (C#) - 在异步任务完成时对其进行处理 (C#)
使用 Async 进行文件访问 (C#)	列出并演示使用 async 和 await 访问文件的好处。
基于任务的异步模式 (TAP)	描述异步模式，该模式基于 Task 和 Task<TResult> 类型。
Channel 9 上的异步相关视频	提供指向有关异步编程的各种视频的链接。

请参阅

- 使用 async 和 await 的异步编程
- async
- await

异步返回类型 （C#）

项目 • 2024/12/28

异步方法可以具有以下返回类型：

- 对于执行操作但不返回任何值的异步方法，Task。
- Task<TResult>，对于返回值的异步方法。
- `void`，对于事件处理程序。
- 任何具有可访问的 `GetAwaiter` 方法的类型。 `GetAwaiter` 方法返回的对象必须实现 System.Runtime.CompilerServices.ICriticalNotifyCompletion 接口。
- IAsyncEnumerable<T>，对于返回*异步流*的异步方法。

有关异步方法的详细信息，请参阅 使用 async 和 await 的异步编程（C#）。

其他几种类型也特定于 Windows 工作负载：

- DispatcherOperation，对于仅限于 Windows 的异步操作。
- IAsyncAction，用于通用 Windows 平台（UWP）应用中不返回值的异步操作。
- IAsyncActionWithProgress<TProgress>，用于在 UWP 应用中进行报告进度但不返回值的异步操作。
- IAsyncOperation<TResult>，适用于 UWP 应用中返回值的异步操作。
- IAsyncOperationWithProgress<TResult,TProgress>，用于在 UWP 应用中报告进度并返回值的异步操作。

任务返回类型

不包含 `return` 语句或包含不返回操作数的 `return` 语句的异步方法通常具有 Task的返回类型。 如果此类方法以同步方式运行，则返回 `void`。 如果在异步方法中使用 Task 返回类型，调用方法可以使用 `await` 运算符暂停调用方的完成，直至被调用的异步方法结束。

在下面的示例中，`WaitAndApologizeAsync` 方法不包含 `return` 语句，因此该方法返回 Task 对象。 返回 `Task` 可等待 `WaitAndApologizeAsync`。 Task 类型不包含 `Result` 属性，因为它没有返回值。

C#

```
public static async Task DisplayCurrentInfoAsync()
{
    await WaitAndApologizeAsync();

    Console.WriteLine($"Today is {DateTime.Now:D}");
    Console.WriteLine($"The current time is {DateTime.Now.TimeOfDay:t}");
```

```
    Console.WriteLine("The current temperature is 76 degrees.");
}

static async Task WaitAndApologizeAsync()
{
    await Task.Delay(2000);

    Console.WriteLine("Sorry for the delay...\n");
}
// Example output:
//    Sorry for the delay...
//
// Today is Monday, August 17, 2020
// The current time is 12:59:24.2183304
// The current temperature is 76 degrees.
```

通过使用 await 语句而不是 await 表达式等待 `WaitAndApologizeAsync`，类似于返回 void 的同步方法的调用语句。 在这种情况下，await 运算符的应用程序不会生成值。 当 `await` 的右操作数是 Task<TResult>时，`await` 表达式将生成 `T`的结果。 当 `await` 的右操作数是 Task 时，`await` 及其操作数是一个语句。

你可以将对 `WaitAndApologizeAsync` 的调用和 await 运算符的应用分开，如以下代码所示。 但是，请记住，`Task` 没有 `Result` 属性，当 await 运算符应用于 `Task`时，不会生成任何值。

以下代码将调用 `WaitAndApologizeAsync` 方法与等待该方法返回的任务分开。

C#

```
Task waitAndApologizeTask = WaitAndApologizeAsync();

string output =
    $"Today is {DateTime.Now:D}\n" +
    $"The current time is {DateTime.Now.TimeOfDay:t}\n" +
    "The current temperature is 76 degrees.\n";

await waitAndApologizeTask;
Console.WriteLine(output);
```

Task<TResult> 返回类型

Task<TResult> 返回类型用于某种异步方法，此异步方法包含 return 语句，其中操作数是 `TResult`。

在以下示例中，`GetLeisureHoursAsync` 方法包含返回整数的 `return` 语句。 方法声明必须指定 `Task<int>`的返回类型。 FromResult 异步方法是返回 DayOfWeek 的操作的占位

符。

```C#
public static async Task ShowTodaysInfoAsync()
{
    string message =
        $"Today is {DateTime.Today:D}\n" +
        "Today's hours of leisure: " +
        $"{await GetLeisureHoursAsync()}";

    Console.WriteLine(message);
}

static async Task<int> GetLeisureHoursAsync()
{
    DayOfWeek today = await Task.FromResult(DateTime.Now.DayOfWeek);

    int leisureHours =
        today is DayOfWeek.Saturday || today is DayOfWeek.Sunday
        ? 16 : 5;

    return leisureHours;
}
// Example output:
//    Today is Wednesday, May 24, 2017
//    Today's hours of leisure: 5
```

当从 `ShowTodaysInfo` 方法内的 await 表达式中调用 `GetLeisureHoursAsync` 时，await 表达式将检索由 `GetLeisureHours` 方法返回的任务中存储的整数值（即 `leisureHours` 的值）。 有关 await 表达式的详细信息，请参阅 await。

可以通过将对 `GetLeisureHoursAsync` 的调用与 `await` 应用程序分开来更好地了解 `await` 如何从 `Task<T>` 检索结果，如以下代码所示。 对非立即等待的方法 `GetLeisureHoursAsync` 的调用返回 `Task<int>`，正如你从方法声明预料的一样。 该任务将分配给示例中的 `getLeisureHoursTask` 变量。 由于 `getLeisureHoursTask` 是 Task<TResult>，因此它包含 `TResult` 类型的 Result 属性。 在这种情况下，`TResult` 表示整数类型。 `await` 应用于 `getLeisureHoursTask`，await 表达式的计算结果为 `getLeisureHoursTask` 的 Result 属性内容。 该值分配给 `ret` 变量。

> **ⓘ 重要**
>
> Result 属性是阻塞属性。 如果在任务完成之前尝试访问它，则当前处于活动状态的线程将被阻止，直到任务完成且值可用。 在大多数情况下，应通过使用 `await` 来访问该值，而不是直接访问属性。

上一个示例检索了 Result 属性的值来阻止主线程，以便 `Main` 方法可以在应用程序结束前将 `message` 打印到控制台。

C#

```
var getLeisureHoursTask = GetLeisureHoursAsync();

string message =
    $"Today is {DateTime.Today:D}\n" +
    "Today's hours of leisure: " +
    $"{await getLeisureHoursTask}";

Console.WriteLine(message);
```

Void 返回类型

在异步事件处理程序中使用 `void` 返回类型，这需要 `void` 返回类型。 对于事件处理程序以外的不返回值的方法，应返回 Task，因为无法等待返回 `void` 的异步方法。 调用此类方法的任何调用方都必须继续执行，而不必等待被调用的异步方法完成。 调用方必须独立于异步方法生成的任何值或异常。

Void 返回异步方法的调用方无法捕获从该方法引发的异常。 此类未经处理的异常可能会导致应用程序失败。 如果返回 Task 或 Task<TResult> 的方法引发异常，异常将存储在返回的任务中。 等待任务时，将重新引发异常。 确保可以生成异常的任何异步方法的返回类型为 Task 或 Task<TResult>，并且等待对方法的调用。

以下示例演示异步事件处理程序的行为。 在示例代码中，异步事件处理程序必须在完成时通知主线程。 然后，主线程可以等待异步事件处理程序完成，然后退出程序。

C#

```
public class NaiveButton
{
    public event EventHandler? Clicked;

    public void Click()
    {
        Console.WriteLine("Somebody has clicked a button. Let's raise the
event...");
        Clicked?.Invoke(this, EventArgs.Empty);
        Console.WriteLine("All listeners are notified.");
    }
}

public class AsyncVoidExample
{
```

```
    static readonly TaskCompletionSource<bool> s_tcs = new
TaskCompletionSource<bool>();

    public static async Task MultipleEventHandlersAsync()
    {
        Task<bool> secondHandlerFinished = s_tcs.Task;

        var button = new NaiveButton();

        button.Clicked += OnButtonClicked1;
        button.Clicked += OnButtonClicked2Async;
        button.Clicked += OnButtonClicked3;

        Console.WriteLine("Before button.Click() is called...");
        button.Click();
        Console.WriteLine("After button.Click() is called...");

        await secondHandlerFinished;
    }

    private static void OnButtonClicked1(object? sender, EventArgs e)
    {
        Console.WriteLine("   Handler 1 is starting...");
        Task.Delay(100).Wait();
        Console.WriteLine("   Handler 1 is done.");
    }

    private static async void OnButtonClicked2Async(object? sender,
EventArgs e)
    {
        Console.WriteLine("   Handler 2 is starting...");
        Task.Delay(100).Wait();
        Console.WriteLine("   Handler 2 is about to go async...");
        await Task.Delay(500);
        Console.WriteLine("   Handler 2 is done.");
        s_tcs.SetResult(true);
    }

    private static void OnButtonClicked3(object? sender, EventArgs e)
    {
        Console.WriteLine("   Handler 3 is starting...");
        Task.Delay(100).Wait();
        Console.WriteLine("   Handler 3 is done.");
    }
}
// Example output:
//
// Before button.Click() is called...
// Somebody has clicked a button. Let's raise the event...
//    Handler 1 is starting...
//    Handler 1 is done.
//    Handler 2 is starting...
//    Handler 2 is about to go async...
//    Handler 3 is starting...
//    Handler 3 is done.
```

```
// All listeners are notified.
// After button.Click() is called...
//    Handler 2 is done.
```

通用的异步返回类型和 ValueTask<TResult>

异步方法可以返回具有返回 awaiter 类型实例的可访问 `GetAwaiter` 方法的所有类型。 此外，从 `GetAwaiter` 方法返回的类型必须具有 System.Runtime.CompilerServices.AsyncMethodBuilderAttribute 属性。 可以通过有关编译器读取的属性的文章或任务类型生成器模式的 C# 规范，了解详细信息。

此功能与 awaitable 表达式相辅相成，后者描述 `await` 操作数的要求。 通用异步返回类型使编译器能够生成返回不同类型的 `async` 方法。 泛化的异步返回类型促进了 .NET 库中的性能改进。 由于 Task 和 Task<TResult> 是引用类型，因此性能关键路径中的内存分配（尤其是在紧张循环中分配时）可能会对性能产生不利影响。 对通用返回类型的支持意味着可以返回轻型值类型而不是引用类型，以避免更多的内存分配。

.NET 提供 System.Threading.Tasks.ValueTask<TResult> 结构作为通用任务返回值的轻型实现。 以下示例使用 ValueTask<TResult> 结构来检索两个骰子卷的值。

C#

```
class Program
{
    static readonly Random s_rnd = new Random();

    static async Task Main() =>
        Console.WriteLine($"You rolled {await GetDiceRollAsync()}");

    static async ValueTask<int> GetDiceRollAsync()
    {
        Console.WriteLine("Shaking dice...");

        int roll1 = await RollAsync();
        int roll2 = await RollAsync();

        return roll1 + roll2;
    }

    static async ValueTask<int> RollAsync()
    {
        await Task.Delay(500);

        int diceRoll = s_rnd.Next(1, 7);
        return diceRoll;
    }
}
// Example output:
```

```
//    Shaking dice...
//    You rolled 8
```

编写通用异步返回类型是一种高级方案，旨在用于专用环境。 请考虑改用 `Task`、`Task<T>`和 `ValueTask<T>` 类型，这些类型涵盖大多数异步代码方案。

可以将 `AsyncMethodBuilder` 属性应用于异步方法（而不是异步返回类型声明），用于替代该类型的生成器。 通常，应应用此属性以使用 .NET 运行时中提供的其他生成器。

使用 IAsyncEnumerable<T> 的异步流

异步方法可能会返回由 *表示的* 异步流IAsyncEnumerable<T>。 异步流提供了一种方法，来枚举在具有重复异步调用的块中生成元素时从流中读取的项。 以下示例显示生成异步流的异步方法：

C#

```
static async IAsyncEnumerable<string> ReadWordsFromStreamAsync()
{
    string data =
        @"This is a line of text.
              Here is the second line of text.
              And there is one more for good measure.
              Wait, that was the penultimate line.";

    using var readStream = new StringReader(data);

    string? line = await readStream.ReadLineAsync();
    while (line != null)
    {
        foreach (string word in line.Split(' ',
StringSplitOptions.RemoveEmptyEntries))
        {
            yield return word;
        }

        line = await readStream.ReadLineAsync();
    }
}
```

前面的示例异步读取字符串中的行。 读取每行后，代码将枚举字符串中的每个单词。 调用方将使用 `await foreach` 语句枚举每个单词。 该方法在需要从源字符串异步读取下一行时等待。

另请参阅

- FromResult
- 在异步任务完成时对其进行处理
- 使用 Async 和 Await 的异步编程 (C#)
- async
- await

在异步任务完成时对其进行处理 (C#)

项目 • 2023/05/23

通过使用 Task.WhenAny，可同时启动多个任务，并在它们完成时逐个对它们进行处理，而不是按照它们的启动顺序进行处理。

下面的示例使用查询来创建一组任务。 每个任务都下载指定网站的内容。 在对 while 循环的每次迭代中，对 WhenAny 的等待调用返回任务集合中首先完成下载的任务。 此任务从集合中删除并进行处理。 循环重复进行，直到集合中不包含任何任务。

先决条件

可以通过以下选项之一来学习本教程：

- 已安装“.NET 桌面开发”工作负载的 Visual Studio 2022。 选择此工作负载时，将自动安装 .NET SDK。
- 具有所选的代码编辑器（例如 Visual Studio Code）的 .NET SDK。

创建示例应用程序

创建新的 .NET Core 控制台应用程序。 可使用 dotnet new console 命令或 Visual Studio 进行创建。

在代码编辑器中打开 Program.cs 文件，并将现有代码替换为以下代码：

C#

```
using System.Diagnostics;

namespace ProcessTasksAsTheyFinish;

class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("Hello World!");
    }
}
```

添加字段

在 `Program` 类定义中，添加以下两个字段：

C#

```
static readonly HttpClient s_client = new HttpClient
{
    MaxResponseContentBufferSize = 1_000_000
};

static readonly IEnumerable<string> s_urlList = new string[]
{
    "https://learn.microsoft.com",
    "https://learn.microsoft.com/aspnet/core",
    "https://learn.microsoft.com/azure",
    "https://learn.microsoft.com/azure/devops",
    "https://learn.microsoft.com/dotnet",
    "https://learn.microsoft.com/dynamics365",
    "https://learn.microsoft.com/education",
    "https://learn.microsoft.com/enterprise-mobility-security",
    "https://learn.microsoft.com/gaming",
    "https://learn.microsoft.com/graph",
    "https://learn.microsoft.com/microsoft-365",
    "https://learn.microsoft.com/office",
    "https://learn.microsoft.com/powershell",
    "https://learn.microsoft.com/sql",
    "https://learn.microsoft.com/surface",
    "https://learn.microsoft.com/system-center",
    "https://learn.microsoft.com/visualstudio",
    "https://learn.microsoft.com/windows",
    "https://learn.microsoft.com/xamarin"
};
```

`HttpClient` 公开发送 HTTP 请求和接收 HTTP 响应的能力。 `s_urlList` 包括应用程序计划处理的所有 URL。

更新应用程序入口点

控制台应用程序的主入口点是 `Main` 方法。 将现有方法替换为以下内容：

C#

```
static Task Main() => SumPageSizesAsync();
```

目前将已更新的 `Main` 方法视为异步 main 方法，这允许将异步入口点引入可执行文件中。 它表示为对 `SumPageSizesAsync` 的调用。

创建异步总和页面大小方法

在 `Main` 方法下，添加 `SumPageSizesAsync` 方法：

C#

```
static async Task SumPageSizesAsync()
{
    var stopwatch = Stopwatch.StartNew();

    IEnumerable<Task<int>> downloadTasksQuery =
        from url in s_urlList
        select ProcessUrlAsync(url, s_client);

    List<Task<int>> downloadTasks = downloadTasksQuery.ToList();

    int total = 0;
    while (downloadTasks.Any())
    {
        Task<int> finishedTask = await Task.WhenAny(downloadTasks);
        downloadTasks.Remove(finishedTask);
        total += await finishedTask;
    }

    stopwatch.Stop();

    Console.WriteLine($"\nTotal bytes returned:  {total:#,#}");
    Console.WriteLine($"Elapsed time:          {stopwatch.Elapsed}\n");
}
```

`while` 循环将删除每次迭代中的一项任务。 完成各项任务后，循环结束。 该方法从实例化和启动 Stopwatch 开始。 然后它会包含一个查询，执行此查询时，将创建任务集合。 每次对以下代码中的 `ProcessUrlAsync` 进行调用都会返回 Task<TResult>，其中 `TResult` 是一个整数：

C#

```
IEnumerable<Task<int>> downloadTasksQuery =
    from url in s_urlList
    select ProcessUrlAsync(url, s_client);
```

由于 LINQ 的延迟执行，因此可调用 Enumerable.ToList 来启动每个任务。

C#

```
List<Task<int>> downloadTasks = downloadTasksQuery.ToList();
```

`while` 循环针对集合中的每个任务执行以下步骤：

1. 等待调用 `WhenAny`，以标识集合中首个已完成下载的任务。

   C#

```
Task<int> finishedTask = await Task.WhenAny(downloadTasks);
```

2. 从集合中移除任务。

```
C#

downloadTasks.Remove(finishedTask);
```

3. 等待 `finishedTask`，由对 `ProcessUrlAsync` 的调用返回。 `finishedTask` 变量是 Task<TResult>，其中 `TResult` 是整数。 任务已完成，但需等待它检索已下载网站的长度，如以下示例所示。 如果任务出错，`await` 将引发存储在 `AggregateException` 中的第一个子异常，这一点与读取 Task<TResult>.Result 属性将引发 `AggregateException` 不同。

```
C#

total += await finishedTask;
```

添加进程方法

在 `SumPageSizesAsync` 方法下添加以下 `ProcessUrlAsync` 方法：

```
C#

static async Task<int> ProcessUrlAsync(string url, HttpClient client)
{
    byte[] content = await client.GetByteArrayAsync(url);
    Console.WriteLine($"{url,-60} {content.Length,10:#,#}");

    return content.Length;
}
```

对于任何给定的 URL，该方法都将使用提供的 `client` 实例以 `byte[]` 形式来获取响应。 将 URL 和长度写入控制台后会返回该长度。

多次运行此程序以验证并不总是以相同顺序显示已下载的长度。

⊗ 注意

如示例所示，可以在循环中使用 `WhenAny` 来解决涉及少量任务的问题。 但是，如果要处理大量任务，可以采用其他更高效的方法。 有关详细信息和示例，请参阅 **Processing tasks as they complete**（在任务完成时处理它们）。

完整示例

下列代码是示例的 Program.cs 文件的完整文本。

C#

```
using System.Diagnostics;

HttpClient s_client = new()
{
    MaxResponseContentBufferSize = 1_000_000
};

IEnumerable<string> s_urlList = new string[]
{
    "https://learn.microsoft.com",
    "https://learn.microsoft.com/aspnet/core",
    "https://learn.microsoft.com/azure",
    "https://learn.microsoft.com/azure/devops",
    "https://learn.microsoft.com/dotnet",
    "https://learn.microsoft.com/dynamics365",
    "https://learn.microsoft.com/education",
    "https://learn.microsoft.com/enterprise-mobility-security",
    "https://learn.microsoft.com/gaming",
    "https://learn.microsoft.com/graph",
    "https://learn.microsoft.com/microsoft-365",
    "https://learn.microsoft.com/office",
    "https://learn.microsoft.com/powershell",
    "https://learn.microsoft.com/sql",
    "https://learn.microsoft.com/surface",
    "https://learn.microsoft.com/system-center",
    "https://learn.microsoft.com/visualstudio",
    "https://learn.microsoft.com/windows",
    "https://learn.microsoft.com/xamarin"
};

await SumPageSizesAsync();

async Task SumPageSizesAsync()
{
    var stopwatch = Stopwatch.StartNew();

    IEnumerable<Task<int>> downloadTasksQuery =
        from url in s_urlList
        select ProcessUrlAsync(url, s_client);
```

```
    List<Task<int>> downloadTasks = downloadTasksQuery.ToList();

    int total = 0;
    while (downloadTasks.Any())
    {
        Task<int> finishedTask = await Task.WhenAny(downloadTasks);
        downloadTasks.Remove(finishedTask);
        total += await finishedTask;
    }

    stopwatch.Stop();

    Console.WriteLine($"\nTotal bytes returned:    {total:#,#}");
    Console.WriteLine($"Elapsed time:              {stopwatch.Elapsed}\n");
}

static async Task<int> ProcessUrlAsync(string url, HttpClient client)
{
    byte[] content = await client.GetByteArrayAsync(url);
    Console.WriteLine($"{url,-60} {content.Length,10:#,#}");

    return content.Length;
}

// Example output:
// https://learn.microsoft.com                                   132,517
// https://learn.microsoft.com/powershell                         57,375
// https://learn.microsoft.com/gaming                             33,549
// https://learn.microsoft.com/aspnet/core                        88,714
// https://learn.microsoft.com/surface                            39,840
// https://learn.microsoft.com/enterprise-mobility-security       30,903
// https://learn.microsoft.com/microsoft-365                      67,867
// https://learn.microsoft.com/windows                            26,816
// https://learn.microsoft.com/xamarin                            57,958
// https://learn.microsoft.com/dotnet                             78,706
// https://learn.microsoft.com/graph                              48,277
// https://learn.microsoft.com/dynamics365                        49,042
// https://learn.microsoft.com/office                             67,867
// https://learn.microsoft.com/system-center                      42,887
// https://learn.microsoft.com/education                          38,636
// https://learn.microsoft.com/azure                             421,663
// https://learn.microsoft.com/visualstudio                       30,925
// https://learn.microsoft.com/sql                                54,608
// https://learn.microsoft.com/azure/devops                       86,034

// Total bytes returned:    1,454,184
// Elapsed time:            00:00:01.1290403
```

另请参阅

- WhenAny
- 使用 Async 和 Await 的异步编程 (C#)

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

异步文件访问 (C#)

项目 • 2023/05/10

可使用异步功能访问文件。 通过使用异步功能，你可以调用异步方法而无需使用回调，也不需要跨多个方法或 lambda 表达式来拆分代码。 若要使同步代码异步，只需调用异步方法而非同步方法，并向代码中添加几个关键字。

可能出于以下原因向文件访问调用中添加异步：

- ✓ 异步使 UI 应用程序响应速度更快，因为启动该操作的 UI 线程可以执行其他操作。 如果 UI 线程必须执行耗时较长的代码（例如超过 50 毫秒），UI 可能会冻结，直到 I/O 完成，此时 UI 线程可以再次处理键盘和鼠标输入及其他事件。
- ✓ 异步可减少对线程的需要，进而提高 ASP.NET 和其他基于服务器的应用程序的可伸缩性。 如果应用程序对每次响应都使用专用线程，同时处理 1000 个请求时，则需要 1000 个线程。 异步操作在等待期间通常不需要使用线程。 异步操作仅需在结束时短暂使用现有 I/O 完成线程。
- ✓ 当前条件下，文件访问操作的延迟可能非常低，但以后可能大幅增加。 例如，文件可能会移动到覆盖全球的服务器。
- ✓ 使用异步功能所增加的开销很小。
- ✓ 异步任务可以轻松地并行运行。

使用适当的类

本主题中的简单示例演示 File.WriteAllTextAsync 和 File.ReadAllTextAsync。 要对文件 I/O 操作进行精细的控制，请使用 FileStream 类，该类有一个可导致在操作系统级别出现异步 I/O 的选项。 使用此选项可避免在许多情况下阻止线程池线程。 若要启用此选项，可在构造函数调用中指定 `useAsync=true` 或 `options=FileOptions.Asynchronous` 参数。

如果通过指定文件路径直接打开 StreamReader 和 StreamWriter，则无法将此选项与这二者配合使用。 但是，如果为二者提供已由 FileStream 类打开的 Stream，则可以使用此选项。 即使线程池线程受到阻止，UI 应用中的异步调用也会更快，因为 UI 线程在等待期间不会受到阻止。

写入文本

下面的示例将文本写入文件。 在每个 await 语句中，该方法会立即退出。 文件 I/O 完成后，该方法将在 await 语句后面的语句中继续。 Async 修饰符位于使用 await 语句的方法定义中。

简单示例

C#

```
public async Task SimpleWriteAsync()
{
    string filePath = "simple.txt";
    string text = $"Hello World";

    await File.WriteAllTextAsync(filePath, text);
}
```

有限控制示例

C#

```
public async Task ProcessWriteAsync()
{
    string filePath = "temp.txt";
    string text = $"Hello World{Environment.NewLine}";

    await WriteTextAsync(filePath, text);
}

async Task WriteTextAsync(string filePath, string text)
{
    byte[] encodedText = Encoding.Unicode.GetBytes(text);

    using var sourceStream =
        new FileStream(
            filePath,
            FileMode.Create, FileAccess.Write, FileShare.None,
            bufferSize: 4096, useAsync: true);

    await sourceStream.WriteAsync(encodedText, 0, encodedText.Length);
}
```

原始示例包含 `await sourceStream.WriteAsync(encodedText, 0, encodedText.Length);` 语句，它是下面两个语句的缩写式：

C#

```
Task theTask = sourceStream.WriteAsync(encodedText, 0, encodedText.Length);
await theTask;
```

第一条语句返回任务，并会导致文件处理启动。 具有 await 的第二条语句将使方法立即退出并返回一个不同的任务。 文件处理稍后完成后，执行将返回到 await 后面的语句

中。

读取文本

下面的示例从文件中读取文本。

简单示例

C#

```
public async Task SimpleReadAsync()
{
    string filePath = "simple.txt";
    string text = await File.ReadAllTextAsync(filePath);

    Console.WriteLine(text);
}
```

有限控制示例

将会缓冲文本，并且在此情况下，会将其放入 StringBuilder。 与前一示例不同，await 的计算将生成一个值。 ReadAsync 方法返回 Task<Int32>，因此在操作完成后 await 的评估会得出 `Int32` 值 `numRead`。 有关详细信息，请参阅异步返回类型 (C#)。

C#

```
public async Task ProcessReadAsync()
{
    try
    {
        string filePath = "temp.txt";
        if (File.Exists(filePath) != false)
        {
            string text = await ReadTextAsync(filePath);
            Console.WriteLine(text);
        }
        else
        {
            Console.WriteLine($"file not found: {filePath}");
        }
    }
    catch (Exception ex)
    {
        Console.WriteLine(ex.Message);
    }
}
```

```
async Task<string> ReadTextAsync(string filePath)
{
    using var sourceStream =
        new FileStream(
            filePath,
            FileMode.Open, FileAccess.Read, FileShare.Read,
            bufferSize: 4096, useAsync: true);

    var sb = new StringBuilder();

    byte[] buffer = new byte[0x1000];
    int numRead;
    while ((numRead = await sourceStream.ReadAsync(buffer, 0,
buffer.Length)) != 0)
    {
        string text = Encoding.Unicode.GetString(buffer, 0, numRead);
        sb.Append(text);
    }

    return sb.ToString();
}
```

并行异步 I/O

下面的示例通过编写 10 个文本文件来演示并行处理。

简单示例

C#

```
public async Task SimpleParallelWriteAsync()
{
    string folder = Directory.CreateDirectory("tempfolder").Name;
    IList<Task> writeTaskList = new List<Task>();

    for (int index = 11; index <= 20; ++ index)
    {
        string fileName = $"file-{index:00}.txt";
        string filePath = $"{folder}/{fileName}";
        string text = $"In file {index}{Environment.NewLine}";

        writeTaskList.Add(File.WriteAllTextAsync(filePath, text));
    }

    await Task.WhenAll(writeTaskList);
}
```

有限控制示例

对于每个文件，WriteAsync 方法将返回一个任务，此任务随后将添加到任务列表中。 `await Task.WhenAll(tasks);` 语句将退出该方法，并在所有任务的文件处理完成时在此方法中继续。

该示例将在任务完成后关闭 `finally` 块中的所有 FileStream 实例。 如果每个 `FileStream` 均已在 `using` 语句中创建，则可能在任务完成前释放 `FileStream`。

任何性能提升都几乎完全来自并行处理而不是异步处理。 异步的优点在于它不会占用多个线程，也不会占用用户界面线程。

C#

```
public async Task ProcessMultipleWritesAsync()
{
    IList<FileStream> sourceStreams = new List<FileStream>();

    try
    {
        string folder = Directory.CreateDirectory("tempfolder").Name;
        IList<Task> writeTaskList = new List<Task>();

        for (int index = 1; index <= 10; ++ index)
        {
            string fileName = $"file-{index:00}.txt";
            string filePath = $"{folder}/{fileName}";

            string text = $"In file {index}{Environment.NewLine}";
            byte[] encodedText = Encoding.Unicode.GetBytes(text);

            var sourceStream =
                new FileStream(
                    filePath,
                    FileMode.Create, FileAccess.Write, FileShare.None,
                    bufferSize: 4096, useAsync: true);

            Task writeTask = sourceStream.WriteAsync(encodedText, 0,
encodedText.Length);
            sourceStreams.Add(sourceStream);

            writeTaskList.Add(writeTask);
        }

        await Task.WhenAll(writeTaskList);
    }
    finally
    {
        foreach (FileStream sourceStream in sourceStreams)
        {
            sourceStream.Close();
        }
```

```
    }
}
```

当使用 WriteAsync 和 ReadAsync 方法时，可以指定可用于取消操作中间流的 CancellationToken。 有关详细信息，请参阅托管线程中的取消。

另请参阅

- 使用 Async 和 Await 的异步编程 (C#)
- 异步返回类型 (C#)

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

取消任务列表

项目 • 2023/03/28

如果不想等待异步控制台应用程序完成，可以取消该应用程序。 通过遵循本主题中的示例，可将取消添加到下载网站内容的应用程序。 可通过将 CancellationTokenSource 实例与每个任务进行关联来取消多个任务。 如果选择 `Enter` 键，则将取消所有尚未完成的任务。

本教程涉及：

- ✓ 创建 .NET 控制台应用程序
- ✓ 编写支持取消的异步应用程序
- ✓ 演示发出取消信号

必备条件

本教程需要的内容如下：

- .NET 5 或更高版本的 SDK
- 集成开发环境 (IDE)
  - 建议使用 Visual Studio 或 Visual Studio Code

创建示例应用程序

创建新的 .NET Core 控制台应用程序。 可通过使用 dotnet new console 命令或从 Visual Studio 进行创建。 在你最喜欢的编辑器中打开 Program.cs 文件。

替换 using 语句

将现有 using 语句替换为以下声明：

C#

```
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Net.Http;
using System.Threading;
using System.Threading.Tasks;
```

添加字段

在 `Program` 类定义中，添加以下三个字段：

C#

```
static readonly CancellationTokenSource s_cts = new
CancellationTokenSource();

static readonly HttpClient s_client = new HttpClient
{
    MaxResponseContentBufferSize = 1_000_000
};

static readonly IEnumerable<string> s_urlList = new string[]
{
    "https://learn.microsoft.com",
    "https://learn.microsoft.com/aspnet/core",
    "https://learn.microsoft.com/azure",
    "https://learn.microsoft.com/azure/devops",
    "https://learn.microsoft.com/dotnet",
    "https://learn.microsoft.com/dynamics365",
    "https://learn.microsoft.com/education",
    "https://learn.microsoft.com/enterprise-mobility-security",
    "https://learn.microsoft.com/gaming",
    "https://learn.microsoft.com/graph",
    "https://learn.microsoft.com/microsoft-365",
    "https://learn.microsoft.com/office",
    "https://learn.microsoft.com/powershell",
    "https://learn.microsoft.com/sql",
    "https://learn.microsoft.com/surface",
    "https://learn.microsoft.com/system-center",
    "https://learn.microsoft.com/visualstudio",
    "https://learn.microsoft.com/windows",
    "https://learn.microsoft.com/xamarin"
};
```

CancellationTokenSource 用于向 CancellationToken 发出请求取消的信号。 `HttpClient` 公开发送 HTTP 请求和接收 HTTP 响应的能力。 `s_urlList` 包括应用程序计划处理的所有 URL。

更新应用程序入口点

控制台应用程序的主入口点是 `Main` 方法。 将现有方法替换为以下内容：

C#

```
static async Task Main()
{
    Console.WriteLine("Application started.");
    Console.WriteLine("Press the ENTER key to cancel...\n");
```

```
    Task cancelTask = Task.Run(() =>
    {
        while (Console.ReadKey().Key != ConsoleKey.Enter)
        {
            Console.WriteLine("Press the ENTER key to cancel...");
        }

        Console.WriteLine("\nENTER key pressed: cancelling downloads.\n");
        s_cts.Cancel();
    });

    Task sumPageSizesTask = SumPageSizesAsync();

    Task finishedTask = await Task.WhenAny(new[] { cancelTask,
sumPageSizesTask });
    if (finishedTask == cancelTask)
    {
        // wait for the cancellation to take place:
        try
        {
            await sumPageSizesTask;
            Console.WriteLine("Download task completed before cancel request
was processed.");
        }
        catch (TaskCanceledException)
        {
            Console.WriteLine("Download task has been cancelled.");
        }
    }

    Console.WriteLine("Application ending.");
}
```

目前将已更新的 `Main` 方法视为异步 main 方法，这允许将异步入口点引入可执行文件中。 将几条说明性消息写入控制台，然后声明名为 `cancelTask` 的 Task 实例，这将读取控制台密钥笔画。 如果按 `Enter`，则会调用 CancellationTokenSource.Cancel()。 这将发出取消信号。 下一步，从 `SumPageSizesAsync` 方法分配 `sumPageSizesTask` 变量。 然后，将这两个任务传递到 Task.WhenAny(Task[])，这会在完成两个任务中的任意一个时继续。

下一个代码块可确保在取消得到处理之前不会退出应用程序。 如果要完成的第一个任务是 `cancelTask`，则等待 `sumPageSizeTask`。 如果已取消，则等待时会引发 System.Threading.Tasks.TaskCanceledException。 块捕获该异常，并输出消息。

创建异步总和页面大小方法

在 `Main` 方法下，添加 `SumPageSizesAsync` 方法：

```
C#
```

```
static async Task SumPageSizesAsync()
{
    var stopwatch = Stopwatch.StartNew();

    int total = 0;
    foreach (string url in s_urlList)
    {
        int contentLength = await ProcessUrlAsync(url, s_client,
s_cts.Token);
        total += contentLength;
    }

    stopwatch.Stop();

    Console.WriteLine($"\nTotal bytes returned:  {total:#,#}");
    Console.WriteLine($"Elapsed time:          {stopwatch.Elapsed}\n");
}
```

该方法从实例化和启动 Stopwatch 开始。 然后会在 `s_urlList` 的每个 URL 中进行循环，并调用 `ProcessUrlAsync`。 对于每次迭代，`s_cts.Token` 都会传递到 `ProcessUrlAsync` 方法中，并且代码将返回 Task<TResult>，其中 `TResult` 是一个整数：

C#

```
int total = 0;
foreach (string url in s_urlList)
{
    int contentLength = await ProcessUrlAsync(url, s_client, s_cts.Token);
    total += contentLength;
}
```

添加进程方法

在 `SumPageSizesAsync` 方法下添加以下 `ProcessUrlAsync` 方法：

C#

```
static async Task<int> ProcessUrlAsync(string url, HttpClient client,
CancellationToken token)
{
    HttpResponseMessage response = await client.GetAsync(url, token);
    byte[] content = await response.Content.ReadAsByteArrayAsync(token);
    Console.WriteLine($"{url,-60} {content.Length,10:#,#}");

    return content.Length;
}
```

对于任何给定的 URL，该方法都将使用提供的 `client` 实例以 `byte[]` 形式来获取响应。 CancellationToken 实例会传递到 HttpClient.GetAsync(String, CancellationToken) 和 HttpContent.ReadAsByteArrayAsync() 方法中。 `token` 用于注册请求取消。 将 URL 和长度写入控制台后会返回该长度。

示例应用程序输出

控制台

```
Application started.
Press the ENTER key to cancel...

https://learn.microsoft.com                                      37,357
https://learn.microsoft.com/aspnet/core                          85,589
https://learn.microsoft.com/azure                               398,939
https://learn.microsoft.com/azure/devops                         73,663
https://learn.microsoft.com/dotnet                               67,452
https://learn.microsoft.com/dynamics365                          48,582
https://learn.microsoft.com/education                            22,924

ENTER key pressed: cancelling downloads.

Application ending.
```

完整示例

下列代码是示例的 Program.cs 文件的完整文本。

C#

```
using System.Diagnostics;

class Program
{
    static readonly CancellationTokenSource s_cts = new
CancellationTokenSource();

    static readonly HttpClient s_client = new HttpClient
    {
        MaxResponseContentBufferSize = 1_000_000
    };

    static readonly IEnumerable<string> s_urlList = new string[]
    {
            "https://learn.microsoft.com",
            "https://learn.microsoft.com/aspnet/core",
            "https://learn.microsoft.com/azure",
            "https://learn.microsoft.com/azure/devops",
```

```
            "https://learn.microsoft.com/dotnet",
            "https://learn.microsoft.com/dynamics365",
            "https://learn.microsoft.com/education",
            "https://learn.microsoft.com/enterprise-mobility-security",
            "https://learn.microsoft.com/gaming",
            "https://learn.microsoft.com/graph",
            "https://learn.microsoft.com/microsoft-365",
            "https://learn.microsoft.com/office",
            "https://learn.microsoft.com/powershell",
            "https://learn.microsoft.com/sql",
            "https://learn.microsoft.com/surface",
            "https://learn.microsoft.com/system-center",
            "https://learn.microsoft.com/visualstudio",
            "https://learn.microsoft.com/windows",
            "https://learn.microsoft.com/xamarin"
    };

    static async Task Main()
    {
        Console.WriteLine("Application started.");
        Console.WriteLine("Press the ENTER key to cancel...\n");

        Task cancelTask = Task.Run(() =>
        {
            while (Console.ReadKey().Key != ConsoleKey.Enter)
            {
                Console.WriteLine("Press the ENTER key to cancel...");
            }

            Console.WriteLine("\nENTER key pressed: cancelling
downloads.\n");
            s_cts.Cancel();
        });

        Task sumPageSizesTask = SumPageSizesAsync();

        Task finishedTask = await Task.WhenAny(new[] { cancelTask,
sumPageSizesTask });
        if (finishedTask == cancelTask)
        {
            // wait for the cancellation to take place:
            try
            {
                await sumPageSizesTask;
                Console.WriteLine("Download task completed before cancel
request was processed.");
            }
            catch (OperationCanceledException)
            {
                Console.WriteLine("Download task has been cancelled.");
            }
        }

        Console.WriteLine("Application ending.");
    }
```

```
    static async Task SumPageSizesAsync()
    {
        var stopwatch = Stopwatch.StartNew();

        int total = 0;
        foreach (string url in s_urlList)
        {
            int contentLength = await ProcessUrlAsync(url, s_client,
s_cts.Token);
            total += contentLength;
        }

        stopwatch.Stop();

        Console.WriteLine($"\nTotal bytes returned:  {total:#,#}");
        Console.WriteLine($"Elapsed time:          {stopwatch.Elapsed}\n");
    }

    static async Task<int> ProcessUrlAsync(string url, HttpClient client,
CancellationToken token)
    {
        HttpResponseMessage response = await client.GetAsync(url, token);
        byte[] content = await response.Content.ReadAsByteArrayAsync(token);
        Console.WriteLine($"{url,-60} {content.Length,10:#,#}");

        return content.Length;
    }
}
```

另请参阅

- CancellationToken
- CancellationTokenSource
- 使用 Async 和 Await 的异步编程 (C#)

后续步骤

在一段时间后取消异步任务 (C#)

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

在一段时间后取消异步任务

项目 • 2023/09/11

如果不希望等待操作结束，可使用 CancellationTokenSource.CancelAfter 方法在一段时间后取消异步操作。 此方法会计划取消未在 `CancelAfter` 表达式指定的时间段内完成的任何关联任务。

此示例添加到取消任务列表 (C#) 中开发的代码，以下载网站列表并显示每个网站的内容长度。

本教程涉及：

- ✔ 更新现有的 .NET 控制台应用程序
- ✔ 计划取消

必备条件

本教程需要的内容如下：

- 在取消任务列表 (C#) 教程中已创建应用程序
- .NET 5 或更高版本的 SDK
- 集成开发环境 (IDE)
  - 建议使用 Visual Studio 或 Visual Studio Code

更新应用程序入口点

将现有的 `Main` 方法替换为以下内容：

C#

```
static async Task Main()
{
    Console.WriteLine("Application started.");

    try
    {
        s_cts.CancelAfter(3500);

        await SumPageSizesAsync();
    }
    catch (OperationCanceledException)
    {
        Console.WriteLine("\nTasks cancelled: timed out.\n");
    }
    finally
```

```
    {
        s_cts.Dispose();
    }

    Console.WriteLine("Application ending.");
}
```

更新后的 `Main` 方法将一些说明性消息写入控制台。 在 try-catch 内，对 CancellationTokenSource.CancelAfter(Int32) 调用安排取消。 一段时间后将发出取消信号。

接下来，等待 `SumPageSizesAsync` 方法。 如果处理所有 URL 的速度比计划取消的速度快，应用程序将结束。 但如果在处理所有 URL 之前触发了计划取消，则会引发 OperationCanceledException。

示例应用程序输出

控制台

```
Application started.

https://learn.microsoft.com                                37,357
https://learn.microsoft.com/aspnet/core                    85,589
https://learn.microsoft.com/azure                         398,939
https://learn.microsoft.com/azure/devops                   73,663

Tasks cancelled: timed out.

Application ending.
```

完整示例

下列代码是示例的 Program.cs 文件的完整文本。

C#

```
using System.Diagnostics;

class Program
{
    static readonly CancellationTokenSource s_cts = new
CancellationTokenSource();

    static readonly HttpClient s_client = new HttpClient
    {
        MaxResponseContentBufferSize = 1_000_000
    };
```

```
    static readonly IEnumerable<string> s_urlList = new string[]
    {
            "https://learn.microsoft.com",
            "https://learn.microsoft.com/aspnet/core",
            "https://learn.microsoft.com/azure",
            "https://learn.microsoft.com/azure/devops",
            "https://learn.microsoft.com/dotnet",
            "https://learn.microsoft.com/dynamics365",
            "https://learn.microsoft.com/education",
            "https://learn.microsoft.com/enterprise-mobility-security",
            "https://learn.microsoft.com/gaming",
            "https://learn.microsoft.com/graph",
            "https://learn.microsoft.com/microsoft-365",
            "https://learn.microsoft.com/office",
            "https://learn.microsoft.com/powershell",
            "https://learn.microsoft.com/sql",
            "https://learn.microsoft.com/surface",
            "https://learn.microsoft.com/system-center",
            "https://learn.microsoft.com/visualstudio",
            "https://learn.microsoft.com/windows",
            "https://learn.microsoft.com/xamarin"
    };

    static async Task Main()
    {
        Console.WriteLine("Application started.");

        try
        {
            s_cts.CancelAfter(3500);

            await SumPageSizesAsync();
        }
        catch (OperationCanceledException)
        {
            Console.WriteLine("\nTasks cancelled: timed out.\n");
        }
        finally
        {
            s_cts.Dispose();
        }

        Console.WriteLine("Application ending.");
    }

    static async Task SumPageSizesAsync()
    {
        var stopwatch = Stopwatch.StartNew();

        int total = 0;
        foreach (string url in s_urlList)
        {
            int contentLength = await ProcessUrlAsync(url, s_client,
s_cts.Token);
```

```
            total += contentLength;
        }

        stopwatch.Stop();

        Console.WriteLine($"\nTotal bytes returned:  {total:#,#}");
        Console.WriteLine($"Elapsed time:          {stopwatch.Elapsed}\n");
    }

    static async Task<int> ProcessUrlAsync(string url, HttpClient client,
CancellationToken token)
    {
        HttpResponseMessage response = await client.GetAsync(url, token);
        byte[] content = await response.Content.ReadAsByteArrayAsync(token);
        Console.WriteLine($"{url,-60} {content.Length,10:#,#}");

        return content.Length;
    }
}
```

另请参阅

- CancellationToken
- CancellationTokenSource
- 使用 Async 和 Await 的异步编程 (C#)
- 取消任务列表 (C#)

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

教程：使用 C# 和 .NET 生成和使用异步流

项目 • 2023/03/29

异步流建立流式处理数据源模型。 数据流经常异步检索或生成元素。 它们为异步流式处理数据源提供了自然编程模型。

在本教程中，你将了解：

- ✓ 创建以异步方式生成数据元素序列的数据源。
- ✓ 以异步方式使用该数据源。
- ✓ 支持异步流的取消和捕获的上下文。
- ✓ 识别新接口和数据源何时优先于先前的同步数据序列。

先决条件

需要将计算机设置为运行 .NET，包括 C# 编译器。 C# 编译器随附于 Visual Studio 2022 或 .NET SDK。

将需要创建 GitHub 访问令牌，以便可以访问 GitHub GraphQL 终结点。 为 GitHub 访问令牌选择以下权限：

- repo:status
- public_repo

将访问令牌保存在安全位置，以便可以使用它来访问 GitHub API 终结点。

> ⚠ **警告**
>
> 保护个人访问令牌。 任何带有你的个人访问令牌的软件都可以使用你的访问权限进行 GitHub API 调用。

本教程假设你熟悉 C# 和 .NET，包括 Visual Studio 或 .NET CLI。

运行初学者应用程序

可以从 asynchronous-programming/snippets 文件夹中的 dotnet/docs 存储库中获取本教程中使用的初学者应用程序的代码。

初学者应用程序是一个控制台应用程序，它使用 GitHub GraphQL 接口检索最近在 dotnet/docs 存储库中编写的问题。 首先来看一下以下初学者应用 `Main` 方法的代码：

C#

```
static async Task Main(string[] args)
{
    //Follow these steps to create a GitHub Access Token
    // https://help.github.com/articles/creating-a-personal-access-token-for-the-command-line/#creating-a-token
    //Select the following permissions for your GitHub Access Token:
    // - repo:status
    // - public_repo
    // Replace the 3rd parameter to the following code with your GitHub access token.
    var key = GetEnvVariable("GitHubKey",
    "You must store your GitHub key in the 'GitHubKey' environment variable",
    "");

    var client = new GitHubClient(new Octokit.ProductHeaderValue("IssueQueryDemo"))
    {
        Credentials = new Octokit.Credentials(key)
    };

    var progressReporter = new progressStatus((num) =>
    {
        Console.WriteLine($"Received {num} issues in total");
    });
    CancellationTokenSource cancellationSource = new CancellationTokenSource();

    try
    {
        var results = await RunPagedQueryAsync(client, PagedIssueQuery, "docs",
            cancellationSource.Token, progressReporter);
        foreach(var issue in results)
            Console.WriteLine(issue);
    }
    catch (OperationCanceledException)
    {
        Console.WriteLine("Work has been cancelled");
    }
}
```

可以将 `GitHubKey` 环境变量设置为个人访问令牌，也可以将对 `GetEnvVariable` 的调用中的最后一个参数替换为个人访问令牌。 如果要与其他人共享源，请不要将访问代码放在源代码中。 不要将访问代码上传到共享源存储库。

在创建 GitHub 客户端后，`Main` 中的代码将创建一个进度报告对象和一个取消令牌。 创建这些对象之后，`Main` 调用 `RunPagedQueryAsync` 来检索最近创建的 250 个问题。 任务完成后，将显示结果。

在运行初学者应用程序时，可以对该应用程序的运行方式进行一些重要观察。 将看到从 GitHub 返回的每个页面的进度报告。 在 GitHub 返回问题的每个新页面之前，可以观察到明显的停顿。 最后，只有在从 GitHub 检索到所有 10 个页面之后，问题才会显示出来。

检查实现情况

该实现揭示了你观察到上一部分中讨论的行为的原因。 检查 `RunPagedQueryAsync` 的代码：

C#

```
private static async Task<JArray> RunPagedQueryAsync(GitHubClient client,
string queryText, string repoName, CancellationToken cancel, IProgress<int>
progress)
{
    var issueAndPRQuery = new GraphQLRequest
    {
        Query = queryText
    };
    issueAndPRQuery.Variables["repo_name"] = repoName;

    JArray finalResults = new JArray();
    bool hasMorePages = true;
    int pagesReturned = 0;
    int issuesReturned = 0;

    // Stop with 10 pages, because these are large repos:
    while (hasMorePages && (pagesReturned++ < 10))
    {
        var postBody = issueAndPRQuery.ToJsonText();
        var response = await client.Connection.Post<string>(new
Uri("https://api.github.com/graphql"),
            postBody, "application/json", "application/json");

        JObject results =
JObject.Parse(response.HttpResponse.Body.ToString()!);

        int totalCount = (int)issues(results)["totalCount"]!;
        hasMorePages = (bool)pageInfo(results)["hasPreviousPage"]!;
        issueAndPRQuery.Variables["start_cursor"] = pageInfo(results)
["startCursor"]!.ToString();
        issuesReturned += issues(results)["nodes"]!.Count();
        finalResults.Merge(issues(results)["nodes"]!);
        progress?.Report(issuesReturned);
        cancel.ThrowIfCancellationRequested();
```

```
    }
    return finalResults;

    JObject issues(JObject result) => (JObject)result["data"]!
["repository"]!["issues"]!;
    JObject pageInfo(JObject result) => (JObject)issues(result)
["pageInfo"]!;
}
```

让我们集中讨论前面代码的分页算法和异步结构。 （有关 GitHub GraphQL API 的详细信息，可以参考 GitHub GraphQL 文档。） `RunPagedQueryAsync` 方法按从最新到最旧的顺序枚举问题。 它每页请求 25 个问题，并检查响应的 `pageInfo` 结构以继续上一页的操作。 这遵循了 GraphQL 对多页响应的标准分页支持。 响应包括 `pageInfo` 对象，该对象包含用于请求上一页的 `hasPreviousPages` 值和 `startCursor` 值。 问题在 `nodes` 数组中。 `RunPagedQueryAsync` 方法将这些节点追加到一个数组中，其中包含所有页面的所有结果。

在检索和还原结果页之后，`RunPagedQueryAsync` 将报告进度并检查是否取消。 如果已请求取消，`RunPagedQueryAsync` 将引发 OperationCanceledException。

此代码中有几个可以改进的元素。 最重要的是，`RunPagedQueryAsync` 必须为返回的所有问题分配存储空间。 该示例在 250 个问题处停止，因为检索所有未决问题需要更多的内存来存储所有检索到的问题。 支持进度报告和取消的协议使得算法在第一次读取时更加难以理解。 涉及更多类型和 API。 必须通过 CancellationTokenSource 及其关联的 CancellationToken 跟踪通信，以了解在何处请求取消，以及在何处授予取消。

异步流可提供更好的方法

异步流和关联语言支持解决了所有这些问题。 生成序列的代码现在可以使用 `yield return` 返回用 `async` 修饰符声明的方法中的元素。 可以通过 `await foreach` 循环来使用异步流，就像通过 `foreach` 循环使用任何序列一样。

这些新语言功能依赖于添加到 .NET Standard 2.1 并在 .NET Core 3.0 中实现的三个新接口：

- System.Collections.Generic.IAsyncEnumerable<T>
- System.Collections.Generic.IAsyncEnumerator<T>
- System.IAsyncDisposable

大多数 C# 开发人员都应该熟悉这三个接口。 它们的行为方式类似于其对应的同步对象：

- System.Collections.Generic.IEnumerable<T>

- System.Collections.Generic.IEnumerator<T>
- System.IDisposable

可能不熟悉的一种类型是 System.Threading.Tasks.ValueTask。 `ValueTask` 结构提供了与 System.Threading.Tasks.Task 类类似的 API。 出于性能方面的原因，这些接口中使用了 `ValueTask`。

转换为异步流

接下来，转换 `RunPagedQueryAsync` 方法以生成异步流。 首先，更改 `RunPagedQueryAsync` 的签名以返回 `IAsyncEnumerable<JToken>`，并从参数列表删除取消令牌和进度对象，如以下代码所示：

C#

```
private static async IAsyncEnumerable<JToken>
RunPagedQueryAsync(GitHubClient client,
    string queryText, string repoName)
```

起始代码在检索页面时处理每个页面，如以下代码所示：

C#

```
finalResults.Merge(issues(results)["nodes"]!);
progress?.Report(issuesReturned);
cancel.ThrowIfCancellationRequested();
```

将这三行替换为以下代码：

C#

```
foreach (JObject issue in issues(results)["nodes"]!)
    yield return issue;
```

还可以在此方法中删除前面的 `finalResults` 声明以及你修改的循环之后的 `return` 语句。

已完成更改以生成异步流。 已完成的方法应与以下代码类似：

C#

```
private static async IAsyncEnumerable<JToken>
RunPagedQueryAsync(GitHubClient client,
    string queryText, string repoName)
```

```
{
    var issueAndPRQuery = new GraphQLRequest
    {
        Query = queryText
    };
    issueAndPRQuery.Variables["repo_name"] = repoName;

    bool hasMorePages = true;
    int pagesReturned = 0;
    int issuesReturned = 0;

    // Stop with 10 pages, because these are large repos:
    while (hasMorePages && (pagesReturned++ < 10))
    {
        var postBody = issueAndPRQuery.ToJsonText();
        var response = await client.Connection.Post<string>(new
Uri("https://api.github.com/graphql"),
            postBody, "application/json", "application/json");

        JObject results =
JObject.Parse(response.HttpResponse.Body.ToString()!);

        int totalCount = (int)issues(results)["totalCount"]!;
        hasMorePages = (bool)pageInfo(results)["hasPreviousPage"]!;
        issueAndPRQuery.Variables["start_cursor"] = pageInfo(results)
["startCursor"]!.ToString();
        issuesReturned += issues(results)["nodes"]!.Count();

        foreach (JObject issue in issues(results)["nodes"]!)
            yield return issue;
    }

    JObject issues(JObject result) => (JObject)result["data"]!
["repository"]!["issues"]!;
    JObject pageInfo(JObject result) => (JObject)issues(result)
["pageInfo"]!;
}
```

接下来，将使用集合的代码更改为使用异步流。 在 `Main` 中找到以下处理问题集合的代码：

C#

```
var progressReporter = new progressStatus((num) =>
{
    Console.WriteLine($"Received {num} issues in total");
});
CancellationTokenSource cancellationSource = new CancellationTokenSource();

try
{
    var results = await RunPagedQueryAsync(client, PagedIssueQuery, "docs",
        cancellationSource.Token, progressReporter);
```

```
    foreach(var issue in results)
        Console.WriteLine(issue);
}
catch (OperationCanceledException)
{
    Console.WriteLine("Work has been cancelled");
}
```

将该代码替换为以下 `await foreach` 循环：

C#

```
int num = 0;
await foreach (var issue in RunPagedQueryAsync(client, PagedIssueQuery,
"docs"))
{
    Console.WriteLine(issue);
    Console.WriteLine($"Received {++num} issues in total");
}
```

新接口 IAsyncEnumerator<T> 派生自 IAsyncDisposable。 这意味着在循环完成时，前面的循环会以异步方式释放流。 可以假设循环类似于以下代码：

C#

```
int num = 0;
var enumerator = RunPagedQueryAsync(client, PagedIssueQuery,
"docs").GetAsyncEnumerator();
try
{
    while (await enumerator.MoveNextAsync())
    {
        var issue = enumerator.Current;
        Console.WriteLine(issue);
        Console.WriteLine($"Received {++num} issues in total");
    }
} finally
{
    if (enumerator != null)
        await enumerator.DisposeAsync();
}
```

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，请参阅有关使用基于任务的异步模式的文章。

异步流支持使用与其他 `async` 方法相同的协议的取消。 要支持取消，请按如下所示修改异步迭代器方法的签名：

C#

```
private static async IAsyncEnumerable<JToken>
RunPagedQueryAsync(GitHubClient client,
    string queryText, string repoName, [EnumeratorCancellation]
CancellationToken cancellationToken = default)
{
    var issueAndPRQuery = new GraphQLRequest
    {
        Query = queryText
    };
    issueAndPRQuery.Variables["repo_name"] = repoName;

    bool hasMorePages = true;
    int pagesReturned = 0;
    int issuesReturned = 0;

    // Stop with 10 pages, because these are large repos:
    while (hasMorePages && (pagesReturned++ < 10))
    {
        var postBody = issueAndPRQuery.ToJsonText();
        var response = await client.Connection.Post<string>(new
Uri("https://api.github.com/graphql"),
            postBody, "application/json", "application/json");

        JObject results =
JObject.Parse(response.HttpResponse.Body.ToString()!);

        int totalCount = (int)issues(results)["totalCount"]!;
        hasMorePages = (bool)pageInfo(results)["hasPreviousPage"]!;
        issueAndPRQuery.Variables["start_cursor"] = pageInfo(results)
["startCursor"]!.ToString();
        issuesReturned += issues(results)["nodes"]!.Count();

        foreach (JObject issue in issues(results)["nodes"]!)
            yield return issue;
    }

    JObject issues(JObject result) => (JObject)result["data"]!
["repository"]!["issues"]!;
    JObject pageInfo(JObject result) => (JObject)issues(result)
["pageInfo"]!;
}
```

System.Runtime.CompilerServices.EnumeratorCancellationAttribute 属性导致编译器生成 IAsyncEnumerator<T> 的代码，该代码使传递给 `GetAsyncEnumerator` 的令牌对作为该参数的异步迭代器的主体可见。 在 `runQueryAsync` 中，可以检查令牌的状态，并在请求时取消进一步的工作。

使用另一个扩展方法 WithCancellation，将取消标记传递给异步流。 可以按如下所示修改枚举问题的循环：

C#

```
private static async Task EnumerateWithCancellation(GitHubClient client)
{
    int num = 0;
    var cancellation = new CancellationTokenSource();
    await foreach (var issue in RunPagedQueryAsync(client, PagedIssueQuery,
"docs")
        .WithCancellation(cancellation.Token))
    {
        Console.WriteLine(issue);
        Console.WriteLine($"Received {++num} issues in total");
    }
}
```

可以从 asynchronous-programming/snippets 文件夹中的 dotnet/docs 存储库中获取已完成的教程的代码。

运行完成的应用程序

再次运行该应用程序。 将其行为与初学者应用程序的行为进行对比。 会在结果的第一页可用立即对其进行枚举。 在请求和检索每个新页面时都会有一个可观察到的暂停，然后快速枚举下一页结果。 不需要 `try` / `catch` 块来处理取消：调用者可以停止枚举集合。 由于异步流在下载每个页面时生成结果，因此可以清楚地报告进度。 返回的每个问题的状态都无缝包含在 `await foreach` 循环中。 不需要回调对象即可跟踪进度。

通过检查代码，可以看到内存使用方面的改进。 不再需要在枚举所有结果之前分配一个集合来存储它们。 调用者可以决定如何使用结果，以及是否需要存储集合。

运行初学者应用程序和已完成的应用程序，可以自行观察实现之间的差异。 可以在完成本教程后删除在开始学习本教程时创建的 GitHub 访问令牌。 如果攻击者获得了对该令牌的访问权限，他们可以使用你的凭据来访问 GitHub API。

在本教程中，你使用异步流从返回数据页的网络 API 读取单个项。 异步流还可以从股票行情自动收录器或传感器设备等“永不结束的流”读取内容。 对 `MoveNextAsync` 的调用将在下一项可用后立即返回它。

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

可为空引用类型

项目 • 2025/01/02

*可为空引用类型*是一组功能，可最大程度地减小代码导致运行时引发 System.NullReferenceException 的可能性。 三项功能，可帮助避免这些异常，包括将引用类型显式标记为*可为 null* 的功能：

- 经过优化的静态流分析，用于在取消引用变量之前确定其是否为 `null`。
- 属性，用于注释 API 以便流分析确定 null 状态。
- 变量注释，可供开发人员用于显式声明变量的预期 null 状态。

编译器在编译时跟踪代码中每个表达式的 *null 状态*。 *空状态* 有两个值之一。

- *not-null*：已知表达式为 not-`null`。
- *maybe-null*：表达式可能是 `null`。

变量批注确定引用类型变量的*为 Null 性*：

- *不可为 null*：如果将值 `null` 或 *maybe-null* 表达式分配给变量，编译器会发出警告。 *不可为 Null* 的变量的默认 null 状态 为 *not-null*。
- *可为 null*：可以为变量赋值 `null` 或 *maybe-null* 表达式。 当变量的 null 状态为 *maybe-null* 时，如果取消引用变量，编译器会发出警告。 变量的默认 null 状态为 *maybe-null*。

本文的其余部分介绍了当你的代码可能**取消引用** `null` 值时，这三个功能区域如何生成警告。 取消引用变量意味着使用 `.`（点）运算符访问其成员之一，如下例所示：

C#

```
string message = "Hello, World!";
int length = message.Length; // dereferencing "message"
```

取消引用值为 `null` 的变量时，运行时会引发 System.NullReferenceException。

当表示法用于在对象为`[]`以下内容时访问对象的成员时`null`，可能会生成类似警告：

C#

```
using System;

public class Collection<T>
{
    private T[] array = new T[100];
    public T this[int index]
```

```
    {
        get => array[index];
        set => array[index] = value;
    }
}

public static void Main()
{
    Collection<int> c = default;
    c[10] = 1;    // CS8602: Possible derefence of null
}
```

本文内容：

- 编译器的 null 状态分析：编译器如何确定表达式为 not-null 或 maybe-null。
- 应用于 API 的属性，这些 API 为编译器的 null 状态分析提供更多上下文。
- 可为 null 的变量注释，用于提供有关变量意向的信息。 批注对于字段、参数和返回值非常有用，用于设置默认 null 状态。
- 控制泛型类型参数的规则。 添加了新约束，因为类型参数可以是引用类型或值类型。 后缀 `?` 针对可为 null 的值类型和可为 null 的引用类型的实现方式不同。
- 可为空上下文可帮助你迁移大型项目。 在应用迁移过程中，你可以在应用的部件中启用可为空上下文的警告和注释。 解决更多警告后，可以为整个项目启用这两个设置。

最后，了解 `struct` 类型和数组中 null 状态分析的已知陷阱。

还可以通过关于 C# 中可为 Null 的安全性的学习模块了解这些概念。

Null 状态分析

Null 状态分析跟踪引用的 *null 状态*。 表达式为“*not-null*”或“*maybe-null*”。 编译器通过两种方式确定变量是否非 null：

1. 该变量已分配给已知*非 null* 的值。
2. 该变量已针对 `null` 进行检查，并且自该检查以来未分配该变量。

编译器无法确定为*非 null* 的任何变量均视为*可能为 null*。 如果意外取消引用 `null` 值，分析会发出警告。 编译器根据 null 状态生成警告。

- 变量为*非 null* 时，可安全地取消引用该变量。
- 变量可能为 null 时，必须先检查该变量，确保其不为 ，然后才能取消引用它。

请考虑以下示例：

```
C#
```

```
string? message = null;

// warning: dereference null.
Console.WriteLine($"The length of the message is {message.Length}");

var originalMessage = message;
message = "Hello, World!";

// No warning. Analysis determined "message" is not-null.
Console.WriteLine($"The length of the message is {message.Length}");

// warning!
Console.WriteLine(originalMessage.Length);
```

在上例中，编译器在打印第一条消息时确定 `message` 是否可能为 null。 对于第二条消息，没有警告。 `originalMessage` 可能为 null，因此最后一行代码发出警告。 下面的示例演示了一个更实际的用途，即遍历节点树直到根，并在遍历过程中处理每个节点：

C#

```
void FindRoot(Node node, Action<Node> processNode)
{
    for (var current = node; current != null; current = current.Parent)
    {
        processNode(current);
    }
}
```

上述代码不会因取消引用变量 `current` 而生成任何警告。 静态分析确定当 `current` 可能为 null 时永不会被取消引用。 访问 `current` 以及将 `null` 传递给 `current.Parent` 操作之前，会检查变量 `current` 是否为 `ProcessNode`。 上述示例演示了编译器如何在初始化、分配或与 比较时确定局部变量的 null 状态。

null 状态分析不会跟踪到调用的方法。 因此，在所有构造函数调用的常见帮助程序方法中初始化的字段可能会生成包含以下消息的警告：

> 在退出构造函数时，不可为 null 的属性“name”必须包含非 null 值。

可以通过以下两种方式之一消除这些警告：帮助程序方法上的构造函数链接或可以为 null 的属性。 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";
    }
}
```

可为 null 的状态分析和编译器生成的警告有助于通过取消引用 `null` 来避免程序错误。 有关解决可为 null 的警告的文章提供了用于更正代码中可能看到的警告的技术。 从空值状态分析生成的诊断仅仅是警告。

API 签名上的属性

null 状态分析需要开发人员的提示才能理解 API 的语义。 某些 API 提供 null 检查，它们应将变量的 null 状态从“可能为 null”更改为“非 null” 。 其他 API 返回非 null 或可能为 null 的表达式，具体取决于输入参数的 null 状态 。 例如，请考虑以下以大写形式显示消息的代码：

C#

```
void PrintMessageUpper(string? message)
{
    if (!IsNull(message))
    {
        Console.WriteLine($"{DateTime.Now}: {message.ToUpper()}");
    }
}

bool IsNull(string? s) => s == null;
```

根据检查，所有开发人员都认为此代码安全，不应生成警告。 但是，编译器不知道 `IsNull` 提供 null 检查，并且将针对 `message.ToUpper()` 语句发出警告，认为 `message` 是一个 *maybe-null* 变量。 若要解决此问题，可使用 NotNullWhen 特性：

C#

```
bool IsNull([NotNullWhen(false)] string? s) => s == null;
```

此属性会通知编译器，如果 `IsNull` 返回 `false`，则参数 `s` 不为 null。 编译器在 块内将 `message` 的 *null 状态*更改为 `if (!IsNull(message)) {...}`。 未发出任何警告。

属性详细说明了用于调用成员的对象实例的参数、返回值和成员的 null 状态。 若要详细了解每个属性，可查看关于可为 null 的引用属性的语言参考文章。 从 .NET 5 起，所有 .NET 运行时 API 都会进行批注。 可通过注释 API 提供有关参数和返回值的 null 状态的语义信息来优化静态分析。

可为 null 的变量注释

*null 状态*分析为本地变量提供可靠的分析。 编译器需要你提供有关成员变量的更多信息。 编译器需要更多信息才能在成员的左括号中设置所有字段的 *null 状态*。 可使用任何可访问的构造函数来初始化对象。 如果某个成员字段曾设为 `null`，则编译器必须在每个方法开始时假定其 null 状态是“可能为 null” 。

可使用能够声明变量是可为 null 引用类型还是不可为 null 引用类型的注释 。 这些注释对变量的 null 状态进行重要声明：

- 引用不应为 null。 不可为 Null 的引用变量的默认状态为 *not-null*。 编译器会强制执行规则，确保即使不先检查变量是否为 null，也能安全地取消引用这些变量：
  - 必须将变量初始化为非 null 值。
  - 变量永远不能赋值为 `null`。 当代码将可能为 null 的表达式分配给不应为 null 的变量时，编译器会发出警告。
- **引用可为 null。** 可为 Null 的引用变量的默认状态为 maybe-null。 编译器会强制执行规则，来确保已正确检查 `null` 引用：
  - 只有在编译器可以保证该值不是`null`时，才能对变量取消引用。
  - 可以使用默认值 `null` 来初始化这些变量，并且在其他代码中可以赋予它们 `null` 的值。
  - 代码将 *maybe-null* 表达式分配给可能为 null 的变量时，编译器不会发出警告。

任何不可为 null 的引用变量都具有*不为 Null* 的初始 *null 状态*。 任何可能为 null 的引用变量的最初 *null 状态*都为 *maybe-null*。

使用与**可为空值类型**相同的语法记录可为空引用类型：将 `?` 附加到变量的类型。 例如，以下变量声明表示可为空的字符串变量 `name`：

```
C#
```

```
string? name;
```

为启用可能为 null 的引用类型时，未将 `?` 附加到类型名称的任何变量都是“**不可为 null 的引用类型**”。 这包括启用此功能时现有代码中的所有引用类型变量。 不过，任何隐式类型本地变量（使用 `var` 声明）都是可为 null 引用类型。 如上述部分所示，静态分析确定局部变量的 *null 状态*，从而在取消引用前确定其是否为 *maybe-null*。

有时，当知道变量不为 null，但编译器确定其 null 状态是“可能为 null”时，必须覆盖警告。 可在变量名称后使用 null 容忍操作符 `!` 来将 null 状态强制为非 null。 例如，如果知道 `name` 变量不为 `null`，但编译器仍发出警告，你可编写以下代码来覆盖编译器的分析：

```
C#
```

```
name!.Length;
```

可为 null 的引用类型和可为 null 的值类型提供类似的语义概念：变量可表示值或对象，或者该变量可以为 `null`。 但可为 null 引用类型和可为 null 值类型的实现方式不同：可为 null 值类型是使用 System.Nullable<T> 实现的，而可为 null 引用类型是使用编译器读取的属性实现的。 例如，`string?` 和 `string` 由同一类型表示：System.String。 但 `int?` 和 `int` 分别由 `System.Nullable<System.Int32>` 和 System.Int32 表示。

可为 null 的引用类型是编译时功能。 这意味着调用方可以忽略警告，故意使用 `null` 作为预期不可为 null 引用的方法的参数。 库作者应纳入针对 null 参数值的运行时检查。 这是 ArgumentNullException.ThrowIfNull 在运行时针对 null 检查参数的首选项。 此外，如果删除了所有可为 null 注释（`?` 和 `!`），则使用可为 null 注释的程序的运行时行为是相同的。 它们的唯一用途是表达设计意向，并为 null 状态分析提供信息。

> ⓘ **重要**
>
> 启用可为 null 的注释后，可以更改 Entity Framework Core 确定是否需要数据成员的方式。 你可在 **Entity Framework Core 基础知识：使用可为 null 的引用类型**一文中了解更多详细信息。

泛型

泛型需要通过详细的规则来处理任何类型参数 `T?` 的 `T`。 由于历史原因以及可为 Null 的值类型和可为 Null 的引用类型的实现各不相同，这些规则必须详细。 可为 Null 的值类型是使用 System.Nullable<T> 结构实现的。 可为 Null 的引用类型实现为向编译器提供语义规则的类型注释。

- 如果 `T` 的类型参数为引用类型，则 `T?` 会引用相应的可为 Null 的引用类型。 例如，如果 `T` 是 `string`，则 `T?` 是 `string?`。
- 如果 `T` 的类型参数是值类型，则 `T?` 将引用相同的值类型 `T`。 例如，如果 `T` 是 `int`，则 `T?` 也是 `int`。
- 如果 `T` 的类型参数是可为 Null 的引用类型，则 `T?` 将引用相同的可为 Null 的引用类型。 例如，如果 `T` 是 `string?`，则 `T?` 也是 `string?`。
- 如果 `T` 的类型参数是可为 Null 的值类型，则 `T?` 将引用相同的可为 Null 的值类型。 例如，如果 `T` 是 `int?`，则 `T?` 也是 `int?`。

对于返回值，`T?` 等效于 `[MaybeNull]T`；对于参数值，`T?` 等效于 `[AllowNull]T`。 有关详细信息，请参阅语言参考中有关 null-state 分析的属性的文章。

可以使用约束指定不同的行为：

- `class` 约束意味着 `T` 必须是不可为 Null 的引用类型（例如 `string`）。 如果使用可为 Null 的引用类型（例如，为 `string?` 使用 `T`），编译器会生成警告。
- `class?` 约束意味着 `T` 必须是引用类型，可以是不可为 Null 的引用类型（`string`），也可以是可为 Null 的引用类型（例如 `string?`）。 当类型参数是可为 Null 的引用类型（例如 `string?`）时，`T?` 的表达式将引用相同的可为 Null 的引用类型（例如 `string?`）。

- `notnull` 约束意味着 `T` 必须是不可为 null 引用类型或不可为 null 值类型。 如果为类型参数使用可为 Null 的引用类型或可为 Null 的值类型，编译器会生成警告。 此外，当 `T` 是值类型时，返回值是该值类型，而不是相应的可为 Null 的值类型。

这些约束帮助为编译器提供有关如何使用 `T` 的更多信息。 这有助于开发人员为 `T` 选择类型，并且可在使用泛型类型的实例时提供更好的 *null 状态*分析。

可为空上下文

*可为空上下文*确定如何处理可为空的引用类型注释，以及由静态 null 状态分析产生的警告。 可空上下文包含两个标志：*批注* 设置和*警告*设置。

对于现有项目，默认禁用了*注释*和*警告*设置。 从 .NET 6（C# 10）开始，这两个标志在*新* 项目中默认启用。 使用两个不同的标志来设置可为空的上下文的原因是：为了更轻松地迁移在引入可为空的引用类型之前开发的大型项目。

对于小型项目，可以启用可为 null 的引用类型、修复警告并继续。 但是，对于大型项目和多项目解决方案，可能会生成大量警告。 可以使用 pragma 在开始使用可为 null 的引用类型时逐文件启用可为 null 的引用类型。 在现有代码库中，防止引发 System.NullReferenceException 的新功能在启用后可能会导致服务中断：

- 所有显式类型引用变量都均解释为不可为 null 引用类型。
- 泛型中 `class` 约束的含义已更改为表示不可为 null 引用类型。
- 由于这些新规则，将生成新警告。

可为 null 注释上下文决定了编译器的行为。 **可为空上下文**设置有四种组合：

- *两者都禁用*：代码是 *nullable-oblivious*。 *禁用* 与启用可为 null 引用类型之前的行为匹配，但新语法生成警告而不是错误。
  - 禁用可为 null 警告。
  - 所有引用类型变量都是可为 null 引用类型。
  - 使用 `?` 后缀来声明可为 null 引用类型会生成警告。
  - 可以使用 null 容忍运算符 `!`，但它不起任何作用。
- *均启用*：编译器启用所有 null 引用分析和所有语言功能。
  - 启用所有新的可为 null 警告。
  - 可使用 `?` 后缀来声明可为 null 引用类型。
  - 没有 `?` 后缀的引用类型变量都是不可为 null 的引用类型。
  - null forgiving 运算符禁止对可能取消引用的 `null` 警告。
- `null` 已启用警告：*当代码可能取消引用* 时，编译器会执行所有 null 分析并发出警告。
  - 启用所有新的可为 null 警告。
  - 使用 `?` 后缀来声明可为 null 引用类型会生成警告。

    - 所有引用类型变量均可为 null。 但是，除非使用 后缀声明成员，否则成员在所有方法的左大括号处都具有非 null 的 null 状态 。
    - 可以使用 null 容忍运算符 `!`。
- *已启用注释*：当代码可能取消引用 `null` 或为可能为 null 的表达式赋予不可为 null 的变量时，编译器不会发出警告。
    - 禁用所有新的可为 null 警告。
    - 可使用 `?` 后缀来声明可为 null 引用类型。
    - 没有 `?` 后缀的引用类型变量都是不可为 null 的引用类型。
    - 可以使用 null 容忍运算符 `!`，但它不起任何作用。

可以使用 .csproj 文件中的 <Nullable> 元素为项目设置可为 null 注释上下文和可为 null 警告上下文。 此元素配置编译器如何解释类型的为 Null 性以及发出哪些警告。 下表显示了允许的值并汇总了它们指定的上下文。

展开表

上下文	取消引用警告	赋值警告	引用类型	`?` 后缀	`!` 运算符
`disable`	已禁用	已禁用	全部可为 null	生成警告	没有作用
`enable`	Enabled	Enabled	不可为 null，除非使用 `?` 声明	声明可为 null 的类型	禁止为可能的 `null` 赋值显示警告
`warnings`	Enabled	不适用	所有成员都可为 null，但在方法的左大括号处，成员被视为 *not-null*	生成警告	禁止为可能的 `null` 赋值显示警告
`annotations`	已禁用	已禁用	不可为 null，除非使用 `?` 声明	声明可为 null 的类型	没有作用

对于已禁用的上下文中编译的代码中的引用类型变量，其为 Null 性未知。 可将 `null` 文本或 *maybe-null* 变量分配给 *Null 性未知*的变量。 但是，nullable-oblivious 变量的默认状态为 not-null 。

可选择最适合你的项目的设置：

- 对于根据诊断或新功能不想更新的旧项目，请选择“禁用”。
- 选择“*警告*”，确定代码可能引发 System.NullReferenceException 的位置。 可先处理这些警告，然后修改代码来启用不可为 null 引用类型。
- 选择“注释”来说明设计意图，然后启用警告。
- 对于希望避免出现 null 引用异常的新项目和活动项目，请选择“启用”。

示例：

```xml
<Nullable>enable</Nullable>
```

还可以使用指令在源代码中的任何位置设置这些相同的标志。 这些指令在迁移大型代码库时最有用。

- `#nullable enable`：将注释和警告标志设置为**启用**。
- `#nullable disable`：将批注和警告标志设置为 **禁用**。
- `#nullable restore`：将批注标志和警告标志还原到项目设置。
- `#nullable disable warnings`：将警告标志设置为 **禁用**。
- `#nullable enable warnings`：将警告标志设置为 **启用**。
- `#nullable restore warnings`：将警告标志还原到项目设置。
- `#nullable disable annotations`：将批注标志设置为 **禁用**。
- `#nullable enable annotations`：将批注标志设置为 **启用**。
- `#nullable restore annotations`：将注释标志还原到项目设置。

对于任何代码行，可设置以下任意组合：

展开表

警告标志	注释标志	用途
项目默认	项目默认	默认
enable	disable	修复分析警告
enable	项目默认	修复分析警告
项目默认	enable	添加类型注释
enable	enable	已迁移的代码
disable	enable	在修复警告之前注释代码
disable	disable	将旧代码添加到已迁移的项目
项目默认	disable	很少
disable	项目默认	很少

通过这九种组合，可精细控制编译器为代码发出的诊断。 你可在正在更新的任何区域中启用更多功能，而不显示尚未准备好解决的其他警告。

> **ⓘ 重要**
>
> 全局可为空上下文不适用于生成的代码文件。 在这两种策略下，都会针对标记为“已生成”的任何源文件禁用可为空上下文。 这意味着生成的文件中的所有 API 都没有批注。 不会为生成的文件生成可为 null 的警告。 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。

默认情况下，可为空注释和警告标志处于**禁用**状态。 这意味着无需更改现有代码即可进行编译，并且不会生成任何新警告。 从 .NET 6 开始，新项目在所有项目模板中包含 `<Nullable>enable</Nullable>` 元素，并且会将这些标志设置为**启用**。

这些选项提供两种不同的策略来更新现有代码库以使用可为 null 的引用类型。

已知缺陷

包含引用类型的数组和结构是可为 null 引用中以及确定 null 安全性的静态分析中的已知缺陷。 在这两种情况下，不可为 null 的引用均可初始化为 `null`，且不会生成警告。

结构

包含不可为 null 的引用类型的结构允许为其分配 `default`，而不会出现任何警告。 请考虑以下示例：

C#

```
using System;

#nullable enable

public struct Student
{
    public string FirstName;
    public string? MiddleName;
    public string LastName;
}
```

```
public static class Program
{
    public static void PrintStudent(Student student)
    {
        Console.WriteLine($"First name: {student.FirstName.ToUpper()}");
        Console.WriteLine($"Middle name: {student.MiddleName?.ToUpper()}");
        Console.WriteLine($"Last name: {student.LastName.ToUpper()}");
    }

    public static void Main() => PrintStudent(default);
}
```

在前面的示例中，不可为 null 引用类型 `PrintStudent(default)` 和 `FirstName` 为 null 时，`LastName` 中未出现警告。

另一种较为常见的情况是处理泛型结构。 请考虑以下示例：

C#

```
#nullable enable

public struct S<T>
{
    public T Prop { get; set; }
}

public static class Program
{
    public static void Main()
    {
        string s = default(S<string>).Prop;
    }
}
```

在上述示例中，属性 `Prop` 的运行时类型为 `null`。 它被分配到不可为 null 的字符串，且不会生成任何警告。

数组

数组也是可为 null 的引用类型中的已知缺陷。 请考虑以下示例，它不会生成任何警告：

C#

```
using System;

#nullable enable

public static class Program
```

```
{
    public static void Main()
    {
        string[] values = new string[10];
        string s = values[0];
        Console.WriteLine(s.ToUpper());
    }
}
```

在前面的示例中，数组的声明显示它保留不可为 null 的字符串，而其元素都已初始化为 `null`。 然后，为变量 `s` 分配一个 `null` 值（数组的第一个元素）。 最后，取消引用变量 `s`，从而导致运行时异常。

另请参阅

- 可以为 Null 的引用类型规范
- 不受约束的类型参数批注
- 可为空引用教程简介
- **Nullable（C# 编译器选项）**
- CS8602：可能取消引用 null 警告

使用可为 Null 的引用类型更新代码库以改进 null 诊断警告

项目 • 2023/04/07

使用可为 Null 的引用类型可以声明是否应为引用类型的变量分配 `null` 值。 当代码可能取消引用 `null` 时会出现的编译器的静态分析和警告是此功能的最重要优势。 启用后，编译器会生成警告，帮助你避免在代码运行时引发 System.NullReferenceException。

如果代码库相对较小，你可以在项目中启用该功能，解决警告，并享受改进后的诊断所带来的好处。 经过一段时间后，更大的代码库可能需要通过结构化程度更高的方法来解决警告，这就需要在解决不同类型的文件中的警告时为某些代码库启用该功能。 本文介绍了更新代码库的不同策略，以及与这些策略相关的折衷方案。 在开始迁移之前，请阅读可为 Null 的引用类型的概念性概述。 此文涵盖了编译器的静态分析、maybe-null 和 not-null 的 null-state 值，以及可为 Null 的注释 。 熟悉这些概念和术语后，便可以开始迁移代码。

规划迁移

无论以哪种方式更新代码库，目标都是在项目中启用可为 Null 的警告和可为 Null 的注释。 实现该目标后，项目中就会使用 `<nullable>Enable</nullable>` 设置。 无需任何预处理器指令即可调整其他位置的设置。

第一种选择是设置项目的默认值。 选项包括：

1. 可为 null 的 disable 为默认值：如果你不向项目文件添加 `Nullable` 元素，则 disable 是默认值。 如果你不主动向代码库添加新文件，请使用此默认值。 主要活动是更新库以使用可为 Null 的引用类型。 使用此默认值意味着在更新每个文件的代码时需要向其添加一个可为 null 的预处理器指令。
2. 可为 null 的 enable 为默认值：如果你正在积极开发新功能，请设置此默认值。 你希望所有新代码都受益于可为 Null 的引用类型和可为 Null 的静态分析。 使用此默认值意味着必须在每个文件的顶部添加 `#nullable disable`。 处理每个文件中的警告时，你将删除这些预处理器指令。
3. 可为 null 的警告为默认值：请为两阶段迁移选择此默认值。 在第一阶段解决警告。 在第二阶段，启用注释以声明变量的预期 null 状态。 使用此默认值意味着必须在每个文件的顶部添加 `#nullable disable`。
4. 可为 null 的注释为默认值。 在解决警告之前注释代码。

启用可为 null 的引用类型作为默认值会增加将预处理器指令添加到每个文件所需的前期工作量。 优点在于，添加到项目的每个新代码文件都支持可为 null 引用类型。 任何新的

代码都可识别可为 null 引用类型；只须更新现有代码。 如果库稳定并且开发重点是采用可为 Null 的引用类型，则禁用可为 Null 的引用类型作为默认值效果更好。 在你注释 API 时，将启用可为 null 引用类型。 完成后，将对整个项目启用可为 null 引用类型。 创建新文件时，必须添加预处理器指令并使其能够感知可为 null 的引用类型。 如果团队中任何开发人员忘记了这一点，则需要将所有代码设置为可识别可为 null 引用类型，这样，新代码现将变为积压工作 (backlog)。

选择哪种策略取决于项目中正在进行多少项活动开发。 项目越成熟稳定，第二种策略的效果越好。 开发的功能越多，第一种策略的效果越好。

> ⓘ **重要**
>
> 全局可为空上下文不适用于生成的代码文件。 在这两种策略下，都会针对标记为“已生成”的任何源文件禁用可为空上下文。 这意味着生成的文件中的所有 API 都没有批注。 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。

了解上下文和警告

启用警告和注释可以控制编译器如何查看引用类型和为 Null 性。 每个类型具有以下三种为 Null 性之一：

- 未知：如果已禁用注释上下文，则所有引用类型的为 Null 性为未知 。
- 不可为 Null：如果已启用注释上下文，则不带注释的引用类型 `C` 不可为 Null 。
- 可为 Null：如果已禁用注释上下文，则带注释的引用类型 `C?` 可为 Null，但可能会发出警告 。 如果已启用注释上下文，则使用 `var` 声明的变量可为 Null。

编译器基于该为 Null 性生成警告：

- 如果为不可为 Null 的类型分配了潜在的 `null` 值，则这些类型会导致出现警告。
- 当可为 Null 的类型的值为 maybe-null 时，取消引用这些类型会导致出现警告 。
- 当未知类型的值为 maybe-null 并已启用警告上下文时，取消引用这些类型会导致出现警告 。

每个变量具有默认的可为 Null 状态，具体取决于它的为 Null 性：

- 可为 Null 的变量的默认 null-state 为 maybe-null 。
- 不可为 Null 的变量的默认 null-state 为 not-null 。
- 可为 Null 的未知变量的默认 null-state 为 not-null 。

在启用可为 Null 的引用类型之前，代码库中的所有声明都是可为 Null 的未知类型。 知道这一点很重要，因为这意味着所有引用类型的默认 null-state 为 not-null 。

解决警告

如果你的项目使用 Entity Framework Core，请阅读使用可为 Null 的引用类型中的相关指导。

开始迁移时，首先应该只启用警告。 所有声明都保持为不可为 Null 的未知类型，但在其 null-state 更改为 may-null 之后，取消引用某个值时会出现警告 。 解决这些警告时，需要根据其他位置的为 Null 性进行检查，然后代码库的可复原性会变得更高。 若要了解适用于不同情况的具体方法，请参阅有关解决可为 Null 的警告的方法的文章。

在继续处理其他代码之前，可以解决警告并在每个文件或类中启用注释。 但是，在启用类型注释之前解决上下文为警告时生成的警告通常更有效。 这样，在解决第一组警告之前，所有类型均为未知。

启用类型注释

解决第一组警告后，可以启用注释上下文。 这会将引用类型从“未知”更改为“不可为 Null”。 使用 `var` 声明的所有变量均可为 Null。 此项更改通常会引入新警告。 解决编译器警告的第一步是在参数和返回类型上使用 `?` 注释，以指示参数或返回值何时可为 `null`。 执行此任务时，目标不只是修复警告。 更重要的目标是让编译器了解潜在 null 值的意图。

特性扩展类型注释

为表示有关变量的 null 状态的附加信息，已添加多个特性。 对于所有参数和返回值，API 的规则可能比 not-null 或 maybe-null 更复杂 。 许多 API 对于变量何时可以或不可以为 `null` 有更复杂的规则。 在这些情况下，可使用属性来表示这些规则。 可以在有关影响可为 Null 分析的属性的文章中找到描述 API 语义的属性。

后续步骤

在启用注释并已解决所有警告后，可将项目的默认上下文设置为 enabled。 如果你在代码中为可为 Null 的注释或警告上下文添加了任何 pragma，可以将其删除。 你可能会不时地看到新警告。 可以编写引入警告的代码。 可以更新可为 null 的引用类型的库依赖项。 这些更新会将该库中的类型从“可为 Null 的未知类型”更改为“不可为 Null”或“可为 Null” 。

还可以通过关于 C# 中可为 Null 的安全性的学习模块了解这些概念。

C# 中的方法

项目 • 2024/05/27

方法是包含一系列语句的代码块。 程序通过调用该方法并指定任何所需的方法参数使语句得以执行。 在 C# 中，每个执行的指令均在方法的上下文中执行。

> **备注**
>
> 本主题讨论命名的方法。 有关匿名函数的信息，请参阅 Lambda 表达式。

方法签名

通过指定以下内容在 `class`、`record` 或 `struct` 中声明方法：

- 可选的访问级别，如 `public` 或 `private`。 默认值为 `private`。
- 可选的修饰符，如 `abstract` 或 `sealed`。
- 返回值，或 `void`（如果该方法不具有）。
- 方法名称。
- 任何方法参数。 方法参数在括号内，并且用逗号分隔。 空括号指示方法不需要任何参数。

这些部分一同构成方法签名。

> **重要**
>
> 出于方法重载的目的，方法的返回类型不是方法签名的一部分。 但是在确定委托和它所指向的方法之间的兼容性时，它是方法签名的一部分。

以下实例定义了一个包含五种方法的名为 `Motorcycle` 的类：

C#

```
namespace MotorCycleExample
{
    abstract class Motorcycle
    {
        // Anyone can call this.
        public void StartEngine() {/* Method statements here */ }

        // Only derived classes can call this.
        protected void AddGas(int gallons) { /* Method statements here */ }
```

```
        // Derived classes can override the base class implementation.
        public virtual int Drive(int miles, int speed) { /* Method
statements here */ return 1; }

        // Derived classes can override the base class implementation.
        public virtual int Drive(TimeSpan time, int speed) { /* Method
statements here */ return 0; }

        // Derived classes must implement this.
        public abstract double GetTopSpeed();
    }
```

`Motorcycle` 类包括一个重载的方法 `Drive`。 两个方法具有相同的名称，但根据其参数类型来区分。

方法调用

方法可以是实例的或静态的。 必须实例化对象以在该实例上调用实例方法；实例方法对该实例及其数据进行操作。 通过引用该方法所属类型的名称来调用静态方法；静态方法不对实例数据进行操作。 尝试通过对象实例调用静态方法会引发编译器错误。

调用方法就像访问字段。 在对象名称（如果调用实例方法）或类型名称（如果调用 `static` 方法）后添加一个句点、方法名称和括号。 自变量列在括号里，并且用逗号分隔。

该方法定义指定任何所需参数的名称和类型。 调用方调用该方法时，它为每个参数提供了称为自变量的具体值。 实参必须与形参类型兼容，但调用代码中使用的实参名（如果有）不需要与方法中定义的形参名相同。 在下面示例中，`Square` 方法包含名为 i 的类型为 `int` 的单个参数。 第一种方法调用将向 `Square` 方法传递名为 num 的 `int` 类型的变量；第二个方法调用将传递数值常量；第三个方法调用将传递表达式。

C#

```
public static class SquareExample
{
    public static void Main()
    {
        // Call with an int variable.
        int num = 4;
        int productA = Square(num);

        // Call with an integer literal.
        int productB = Square(12);

        // Call with an expression that evaluates to int.
        int productC = Square(productA * 3);
    }
```

```
    static int Square(int i)
    {
        // Store input argument in a local variable.
        int input = i;
        return input * input;
    }
}
```

方法调用最常见的形式是使用位置自变量；它会以与方法参数相同的顺序提供自变量。 因此，可在以下示例中调用 `Motorcycle` 类的方法。 例如，`Drive` 方法的调用包含两个与方法语法中的两个参数对应的自变量。 第一个成为`miles`参数的值。 第二个成为`speed`参数的值。

C#

```
class TestMotorcycle : Motorcycle
{
    public override double GetTopSpeed() => 108.4;

    static void Main()
    {
        var moto = new TestMotorcycle();

        moto.StartEngine();
        moto.AddGas(15);
        _ = moto.Drive(5, 20);
        double speed = moto.GetTopSpeed();
        Console.WriteLine("My top speed is {0}", speed);
    }
}
```

调用方法时，也可以使用命名的自变量，而不是位置自变量。 使用命名的自变量时，指定参数名，然后后跟冒号（":"）和自变量。 只要包含了所有必需的自变量，方法的自变量可以任何顺序出现。 下面的示例使用命名的自变量来调用 `TestMotorcycle.Drive` 方法。 在此示例中，命名的自变量以相反于方法参数列表中的顺序进行传递。

C#

```
namespace NamedMotorCycle;

class TestMotorcycle : Motorcycle
{
    public override int Drive(int miles, int speed) =>
        (int)Math.Round((double)miles / speed, 0);

    public override double GetTopSpeed() => 108.4;

    static void Main()
```

```
    {
        var moto = new TestMotorcycle();
        moto.StartEngine();
        moto.AddGas(15);
        int travelTime = moto.Drive(miles: 170, speed: 60);
        Console.WriteLine("Travel time: approx. {0} hours", travelTime);
    }
}
// The example displays the following output:
//      Travel time: approx. 3 hours
```

可以同时使用位置自变量和命名的自变量调用方法。 但是，只有当命名参数位于正确位置时，才能在命名自变量后面放置位置参数。 下面的示例使用一个位置自变量和一个命名的自变量从上一个示例中调用 `TestMotorcycle.Drive` 方法。

C#

```
int travelTime = moto.Drive(170, speed: 55);
```

继承和重写方法

除了类型中显式定义的成员，类型还继承在其基类中定义的成员。 由于托管类型系统中的所有类型都直接或间接继承自 Object 类，因此所有类型都继承其成员，如 Equals(Object)、GetType() 和 ToString()。 下面的示例定义 `Person` 类，实例化两个 `Person` 对象，并调用 `Person.Equals` 方法来确定两个对象是否相等。 但是，`Equals` 方法不是在 `Person` 类中定义；而是继承自 Object。

C#

```
public class Person
{
    public string FirstName = default!;
}

public static class ClassTypeExample
{
    public static void Main()
    {
        Person p1 = new() { FirstName = "John" };
        Person p2 = new() { FirstName = "John" };
        Console.WriteLine("p1 = p2: {0}", p1.Equals(p2));
    }
}
// The example displays the following output:
//      p1 = p2: False
```

类型可以使用 `override` 关键字并提供重写方法的实现来重写继承的成员。 方法签名必须与重写的方法的签名一样。 下面的示例类似于上一个示例，只不过它重写 Equals(Object) 方法。 （它还重写 GetHashCode() 方法，因为这两种方法用于提供一致的结果。）

C#

```
namespace methods;

public class Person
{
    public string FirstName = default!;

    public override bool Equals(object? obj) =>
        obj is Person p2 &&
        FirstName.Equals(p2.FirstName);

    public override int GetHashCode() => FirstName.GetHashCode();
}

public static class Example
{
    public static void Main()
    {
        Person p1 = new() { FirstName = "John" };
        Person p2 = new() { FirstName = "John" };
        Console.WriteLine("p1 = p2: {0}", p1.Equals(p2));
    }
}
// The example displays the following output:
//       p1 = p2: True
```

快速参考

C# 中的所有类型不是值类型就是引用类型。 有关内置值类型的列表，请参阅类型。 默认情况下，值类型和引用类型均按值传递给方法。

按值传递参数

值类型按值传递给方法时，传递的是对象的副本而不是对象本身。 因此，当控件返回调用方时，对已调用方法中的对象的更改对原始对象无影响。

下面的示例按值向方法传递值类型，且调用的方法尝试更改值类型的值。 它定义属于值类型的 `int` 类型的变量，将其值初始化为 20，并将该类型传递给将变量值改为 30 的名为 `ModifyValue` 的方法。 但是，返回方法时，变量的值保持不变。

C#

```
public static class ByValueExample
{
    public static void Main()
    {
        var value = 20;
        Console.WriteLine("In Main, value = {0}", value);
        ModifyValue(value);
        Console.WriteLine("Back in Main, value = {0}", value);
    }

    static void ModifyValue(int i)
    {
        i = 30;
        Console.WriteLine("In ModifyValue, parameter value = {0}", i);
        return;
    }
}
// The example displays the following output:
//      In Main, value = 20
//      In ModifyValue, parameter value = 30
//      Back in Main, value = 20
```

引用类型的对象按值传递到方法中时，将按值传递对对象的引用。 也就是说，该方法接收的不是对象本身，而是指示该对象位置的自变量。 控件返回到调用方法时，如果通过使用此引用更改对象的成员，此更改将反映在对象中。 但是，当控件返回到调用方时，替换传递到方法的对象对原始对象无影响。

下面的示例定义名为 `SampleRefType` 的类（属于引用类型）。 它实例化 `SampleRefType` 对象，将 44 赋予其 `value` 字段，并将该对象传递给 `ModifyObject` 方法。 该示例执行的内容实质上与先前示例相同，即均按值将自变量传递到方法。 但因为使用了引用类型，结果会有所不同。 `ModifyObject` 中所做的对 `obj.value` 字段的修改，也会将 `Main` 方法中的自变量 `rt` 的 `value` 字段更改为 33，如示例中的输出值所示。

C#

```
public class SampleRefType
{
    public int value;
}

public static class ByRefTypeExample
{
    public static void Main()
    {
        var rt = new SampleRefType { value = 44 };
        ModifyObject(rt);
        Console.WriteLine(rt.value);
    }
```

```
    static void ModifyObject(SampleRefType obj) => obj.value = 33;
}
```

按引用传递参数

如果想要更改方法中的自变量值并想要在控件返回到调用方法时反映出这一更改，请按引用传递参数。 要按引用传递参数，请使用 ref 或 out 关键字。 还可以使用 in 关键字，按引用传递值以避免复制，但仍防止修改。

下面的示例与上一个示例完全一样，只是换成按引用将值传递给`ModifyValue`方法。 参数值在 `ModifyValue` 方法中修改时，值中的更改将在控件返回调用方时反映出来。

C#

```
public static class ByRefExample
{
    public static void Main()
    {
        var value = 20;
        Console.WriteLine("In Main, value = {0}", value);
        ModifyValue(ref value);
        Console.WriteLine("Back in Main, value = {0}", value);
    }

    private static void ModifyValue(ref int i)
    {
        i = 30;
        Console.WriteLine("In ModifyValue, parameter value = {0}", i);
        return;
    }
}
// The example displays the following output:
//      In Main, value = 20
//      In ModifyValue, parameter value = 30
//      Back in Main, value = 30
```

引用参数所使用的常见模式涉及交换变量值。 将两个变量按引用传递给一个方法，然后该方法将二者内容进行交换。 下面的示例交换整数值。

C#

```


public static class RefSwapExample
{
    static void Main()
    {
        int i = 2, j = 3;
        Console.WriteLine("i = {0}  j = {1}", i, j);
```

```
        Swap(ref i, ref j);

        Console.WriteLine("i = {0}  j = {1}", i, j);
    }

    static void Swap(ref int x, ref int y) =>
        (y, x) = (x, y);
}
// The example displays the following output:
//      i = 2  j = 3
//      i = 3  j = 2
```

通过传递引用类型的参数，可以更改引用本身的值，而不是其单个元素或字段的值。

参数集合

有时，向方法指定精确数量的自变量这一要求是受限的。 通过使用`params`关键字来指示一个参数是一个参数集合，可通过可变数量的自变量来调用方法。 使用`params`关键字标记的参数必须为集合类型，并且必须是该方法的参数列表中的最后一个参数。

然后，调用方可通过以下四种方式中的任一种来调用`params`参数的方法：

- 传递相应类型的集合，该类型包含所需数量的元素。 该示例使用集合表达式，以便编译器创建适当的集合类型。
- 向该方法传递相应类型的单独自变量的逗号分隔列表。 编译器创建适当的集合类型。
- 传递 `null`。
- 不向参数集合提供参数。

以下示例定义了一个名为`GetVowels`的方法，该方法返回参数集合中的所有元音。 `Main`方法演示了调用方法的全部四种方式。 调用方不需要为包含 `params` 修饰符的形参提供任何实参。 在这种情况下，参数是一个空集合。

C#

```
static class ParamsExample
{
    static void Main()
    {
        string fromArray = GetVowels(["apple", "banana", "pear"]);
        Console.WriteLine($"Vowels from collection expression:
'{fromArray}'");

        string fromMultipleArguments = GetVowels("apple", "banana", "pear");
        Console.WriteLine($"Vowels from multiple arguments:
'{fromMultipleArguments}'");
```

```
        string fromNull = GetVowels(null);
        Console.WriteLine($"Vowels from null: '{fromNull}'");

        string fromNoValue = GetVowels();
        Console.WriteLine($"Vowels from no value: '{fromNoValue}'");
    }

    static string GetVowels(params IEnumerable<string>? input)
    {
        if (input == null || !input.Any())
        {
            return string.Empty;
        }

        char[] vowels = ['A', 'E', 'I', 'O', 'U'];
        return string.Concat(
            input.SelectMany(
                word => word.Where(letter =>
vowels.Contains(char.ToUpper(letter)))));
    }
}

// The example displays the following output:
//     Vowels from array: 'aeaaaea'
//     Vowels from multiple arguments: 'aeaaaea'
//     Vowels from null: ''
//     Vowels from no value: ''
```

在 C# 13 之前，`params`修饰符只能与单维数组结合使用。

可选参数和自变量

方法定义可指定其参数是必需的还是可选的。 默认情况下，参数是必需的。 通过在方法定义中包含参数的默认值来指定可选参数。 调用该方法时，如果未向可选参数提供自变量，则改为使用默认值。

参数的默认值由以下几种表达式中的一种来赋予：

- 常量，例如文本字符串或数字。
- `default(SomeType)` 形式的表达式，其中 `SomeType` 可以是值类型或引用类型。 如果是引用类型，那么它实际上与指定 `null` 相同。 可以使用 `default` 字面量，因为编译器可以从参数的声明中推断出类型。
- `new ValType()` 形式的表达式，其中 `ValType` 是值类型。 此表达式会调用该值类型的隐式无参数构造函数，该函数不是类型的实际成员。

> **备注**

> 在 C# 10 及更高版本中，当 `new ValType()` 形式的表达式调用某一值类型的显式定义的无参数构造函数时，编译器便会生成错误，因为默认参数值必须是编译时常数。 使用 `default(ValType)` 表达式或 `default` 字面量提供默认参数值。 有关无参数构造函数的详细信息，请参阅**结构类型**一文的**结构初始化和默认值**部分。

如果某个方法同时包含必需的和可选的参数，则在参数列表末尾定义可选参数，即在定义完所有必需参数之后定义。

下面的示例定义方法 `ExampleMethod`，它具有一个必需参数和两个可选参数。

C#

```
public class Options
{
    public void ExampleMethod(int required, int optionalInt = default,
                              string? description = default)
    {
        var msg = $"{description ?? "N/A"}: {required} + {optionalInt} =
{required + optionalInt}";
        Console.WriteLine(msg);
    }
}
```

调用方必须为所有可选参数提供自变量，直到为其提供自变量的最后一个可选参数。 例如，在`ExampleMethod`方法中，如果调用方向`description`参数提供自变量，还必须向`optionalInt`参数提供一个自变量。 `opt.ExampleMethod(2, 2, "Addition of 2 and 2");` 是一个有效的方法调用；`opt.ExampleMethod(2, , "Addition of 2 and 0");` 生成编译器错误“缺少自变量”。

如果使用命名的自变量或位置自变量和命名的自变量的组合来调用某个方法，调用方可以省略方法调用中的最后一个位置自变量后的任何自变量。

下面的示例三次调用了 `ExampleMethod` 方法。 前两个方法调用使用位置自变量。 第一个方法同时省略了两个可选自变量，而第二个省略了最后一个自变量。 第三个方法调用向必需的参数提供位置自变量，但使用命名的自变量向 `description` 参数提供值，同时省略 `optionalInt` 自变量。

C#

```
public static class OptionsExample
{
    public static void Main()
    {
        var opt = new Options();
```

```
        opt.ExampleMethod(10);
        opt.ExampleMethod(10, 2);
        opt.ExampleMethod(12, description: "Addition with zero:");
    }
}
// The example displays the following output:
//      N/A: 10 + 0 = 10
//      N/A: 10 + 2 = 12
//      Addition with zero:: 12 + 0 = 12
```

使用可选参数会影响*重载决策*，或影响 C# 编译器决定方法要调用哪个重载时所使用的方式，如下所示：

- 如果方法、索引器或构造函数的每个参数按名称或位置对应于单个自变量，且该自变量可转换为参数的类型，则方法、索引器或构造函数为执行的候选项。
- 如果找到多个候选项，则会将用于首选转换的重载决策规则应用于显式指定的自变量。 将忽略可选形参已省略的实参。
- 如果两个候选项不相上下，则会将没有可选形参的候选项作为首选项，对于这些可选形参，已在调用中为其省略了实参。

返回值

方法可以将值返回到调用方。 如果列在方法名之前的返回类型不是 `void`，则该方法可通过使用 `return` 关键字返回值。 带`return`关键字且后跟与返回类型匹配的变量、常数或表达式的语句会向方法调用方返回该值。 具有非空的返回类型的方法都需要使用`return`关键字来返回值。 `return` 关键字还会停止执行该方法。

如果返回类型为 `void`，没有值的 `return` 语句仍可用于停止执行该方法。 没有`return`关键字，当方法到达代码块结尾时，会停止执行。

例如，这两种方法都使用 `return` 关键字来返回整数：

C#

```
class SimpleMath
{
    public int AddTwoNumbers(int number1, int number2) =>
        number1 + number2;

    public int SquareANumber(int number) =>
        number * number;
}
```

若要使用从方法返回的值，调用方法可以在相同类型的值足够的地方使用该方法调用本身。 也可以将返回值分配给变量。 例如，以下两个代码示例实现了相同的目标：

C#

```
int result = obj.AddTwoNumbers(1, 2);
result = obj.SquareANumber(result);
// The result is 9.
Console.WriteLine(result);
```

C#

```
result = obj.SquareANumber(obj.AddTwoNumbers(1, 2));
// The result is 9.
Console.WriteLine(result);
```

有时，需要方法返回多个值。 使用*元组类型*和*元组文本*返回多个值。 元组类型定义元组元素的数据类型。 元组文本提供返回的元组的实际值。 在下面的示例中，`(string, string, string, int)`定义`GetPersonalInfo`方法返回的元组类型。 表达式`(per.FirstName, per.MiddleName, per.LastName, per.Age)`是元组文本；方法返回`PersonInfo`对象的第一个、中间和系列名称及其使用期限。

C#

```
public (string, string, string, int) GetPersonalInfo(string id)
{
    PersonInfo per = PersonInfo.RetrieveInfoById(id);
    return (per.FirstName, per.MiddleName, per.LastName, per.Age);
}
```

然后，调用方可以通过以下代码使用返回的元组：

C#

```
var person = GetPersonalInfo("111111111");
Console.WriteLine($"{person.Item1} {person.Item3}: age = {person.Item4}");
```

还可向元组类型定义中的元组元素分配名称。 下面的示例展示`GetPersonalInfo`方法的替代版本，该方法使用命名的元素：

C#

```
public (string FName, string MName, string LName, int Age)
GetPersonalInfo(string id)
{
    PersonInfo per = PersonInfo.RetrieveInfoById(id);
    return (per.FirstName, per.MiddleName, per.LastName, per.Age);
}
```

然后可修改上一次对 `GetPersonalInfo` 方法的调用，如下所示：

C#

```
var person = GetPersonalInfo("111111111");
Console.WriteLine($"{person.FName} {person.LName}: age = {person.Age}");
```

如果方法采用数组作为参数并修改各个元素的值，则该方法不需要返回数组。 C# 会按值传递所有引用类型，而数组引用的值是指向该数组的指针。 在下面的示例中，引用该数组的任何代码都能观察到在 `DoubleValues` 方法中对 `values` 数组内容的更改。

C#

```
public static class ArrayValueExample
{
    static void Main()
    {
        int[] values = [2, 4, 6, 8];
        DoubleValues(values);
        foreach (var value in values)
        {
            Console.Write("{0}  ", value);
        }
    }

    public static void DoubleValues(int[] arr)
    {
        for (var ctr = 0; ctr <= arr.GetUpperBound(0); ctr++)
        {
            arr[ctr] *= 2;
        }
    }
}
// The example displays the following output:
//       4  8  12  16
```

扩展方法

通常，可以通过两种方式向现有类型添加方法：

- 修改该类型的源代码。 如果还添加任何专用数据字段来支持该方法，则修改源会成为一项重大更改。
- 在派生类中定义新方法。 无法使用其他类型（如结构和枚举）的继承来通过此方式添加方法。 也不能使用此方式向封闭类“添加”方法。

使用扩展方法，可向现有类型“添加”方法，而无需修改类型本身或在继承的类型中实现新方法。 扩展方法也无需驻留在与其扩展的类型相同的程序集中。 要把扩展方法当作是定义的类型成员一样调用。

有关详细信息，请参阅扩展方法

异步方法

通过使用异步功能，你可以调用异步方法而无需使用显式回调，也不需要跨多个方法或 lambda 表达式来手动拆分代码。

如果用 async 修饰符标记方法，则可以在该方法中使用 await 运算符。 当控件到达异步方法中的 `await` 表达式时，如果等待的任务未完成，控件将返回到调用方，并在等待任务完成前，包含 `await` 关键字的方法中的进度将一直处于挂起状态。 任务完成后，可以在方法中恢复执行。

> **备注**
>
> 异步方法在遇到第一个尚未完成的 awaited 对象或到达异步方法的末尾时（以先发生者为准），将返回到调用方。

异步方法通常具有Task<TResult>、Task、IAsyncEnumerable<T>或`void`返回类型。 `void` 返回类型主要用于定义需要 `void` 返回类型的事件处理程序。 无法等待返回 `void` 的异步方法，并且返回 void 方法的调用方无法捕获该方法引发的异常。 异步方法可以具有任何类似任务的返回类型。

在下面的示例中，`DelayAsync` 是一个异步方法，包含返回整数的 return 语句。 由于它是异步方法，其方法声明必须具有返回类型 `Task<int>`。 因为返回类型是 `Task<int>`，`DoSomethingAsync` 中 `await` 表达式的计算将如以下 `int result = await delayTask` 语句所示得出整数。

C#

```
class Program
{
    static Task Main() => DoSomethingAsync();

    static async Task DoSomethingAsync()
    {
        Task<int> delayTask = DelayAsync();
        int result = await delayTask;

        // The previous two statements may be combined into
        // the following statement.
```

```
        //int result = await DelayAsync();

        Console.WriteLine($"Result: {result}");
    }

    static async Task<int> DelayAsync()
    {
        await Task.Delay(100);
        return 5;
    }
}
// Example output:
//   Result: 5
```

异步方法不能声明任何 in、ref 或 out 参数，但是可以调用具有这类参数的方法。

有关异步方法的详细信息，请参阅使用 Async 和 Await 的异步编程和异步返回类型。

Expression-Bodied 成员

具有立即返回表达式结果，或单个语句作为方法主题的方法定义很常见。 以下是使用 `=>` 定义此类方法的语法快捷方式：

C#

```
public Point Move(int dx, int dy) => new Point(x + dx, y + dy);
public void Print() => Console.WriteLine(First + " " + Last);
// Works with operators, properties, and indexers too.
public static Complex operator +(Complex a, Complex b) => a.Add(b);
public string Name => First + " " + Last;
public Customer this[long id] => store.LookupCustomer(id);
```

如果该方法返回 `void` 或是异步方法，则该方法的主体必须是语句表达式（与 lambda 相同）。 对于属性和索引器，两者必须是只读的，并且不使用 `get` 访问器关键字。

迭代器

迭代器对集合执行自定义迭代，如列表或数组。 迭代器使用 yield return 语句返回元素，每次返回一个。 到达 `yield return` 语句后，会记住当前位置，以便调用方可以请求序列中的下一个元素。

迭代器的返回类型可以是 IEnumerable、IEnumerable<T>、IAsyncEnumerable<T>、IEnumerator 或 IEnumerator<T>。

有关更多信息，请参见 迭代器。

另请参阅

- 访问修饰符
- 静态类和静态类成员
- 继承
- 抽象类、密封类及类成员
- params
- out
- ref
- in
- 传递参数

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

迭代器

项目 • 2023/05/10

编写的几乎每个程序都需要循环访问集合。 因此需要编写代码来检查集合中的每一项。

还需创建迭代器方法，这些方法可为该类的元素生成迭代器。 迭代器是遍历容器的对象，尤其是列表。 迭代器可用于：

- 对集合中的每个项执行操作。
- 枚举自定义集合。
- 扩展 LINQ 或其他库。
- 创建数据管道，以便数据通过迭代器方法在管道中有效流动。

C# 语言提供用于生成和使用序列的功能。 可以同步或异步生成和使用这些序列。 本文概述了这些功能。

使用 foreach 执行循环访问

枚举集合非常简单：使用 `foreach` 关键字枚举集合，从而为集合中的每个元素执行一次嵌入语句：

C#

```
foreach (var item in collection)
{
    Console.WriteLine(item?.ToString());
}
```

就这样。 若要循环访问集合中的所有内容，只需使用 `foreach` 语句。 但 `foreach` 语句并非完美无缺。 它依赖于 .NET Core 库中定义的 2 个泛型接口，才能生成循环访问集合所需的代码：`IEnumerable<T>` 和 `IEnumerator<T>`。 下文对此机制进行了更详细说明。

这 2 种接口还具备相应的非泛型接口：`IEnumerable` 和 `IEnumerator`。 泛型版本是新式代码的首要选项。

异步生成序列时，可以使用 `await foreach` 语句异步使用此序列：

C#

```
await foreach (var item in asyncSequence)
{
Console.WriteLine(item?.ToString());
}
```

如果序列是 System.Collections.Generic.IEnumerable<T>，则使用 `foreach`。 如果序列是 System.Collections.Generic.IAsyncEnumerable<T>，则使用 `await foreach`。 在后一种情况下，序列是异步生成的。

使用迭代器方法的枚举源

借助 C# 语言的另一个强大功能，能够生成创建枚举源的方法。 这些方法称为“迭代器方法”。 迭代器方法用于定义请求时如何在序列中生成对象。 使用 `yield return` 上下文关键字定义迭代器方法。

可编写此方法以生成从 0 到 9 的整数序列：

C#

```
public IEnumerable<int> GetSingleDigitNumbers()
{
    yield return 0;
    yield return 1;
    yield return 2;
    yield return 3;
    yield return 4;
    yield return 5;
    yield return 6;
    yield return 7;
    yield return 8;
    yield return 9;
}
```

上方的代码显示了不同的 `yield return` 语句，以强调可在迭代器方法中使用多个离散 `yield return` 语句这一事实。 可以使用其他语言构造来简化迭代器方法的代码，这也是一贯的做法。 以下方法定义可生成完全相同的数字序列：

C#

```
public IEnumerable<int> GetSingleDigitNumbersLoop()
{
    int index = 0;
    while (index < 10)
        yield return index++;
}
```

不必从中选择一个。 可根据需要提供尽可能多的 `yield return` 语句来满足方法需求：

C#

```
public IEnumerable<int> GetSetsOfNumbers()
{
    int index = 0;
    while (index < 10)
        yield return index++;

    yield return 50;

    index = 100;
    while (index < 110)
        yield return index++;
}
```

上述所有示例都有一个异步对应项。 在每种情况下，将 `IEnumerable<T>` 的返回类型替换为 `IAsyncEnumerable<T>`。 例如，前面的示例将具有以下异步版本：

C#

```
public async IAsyncEnumerable<int> GetSetsOfNumbersAsync()
{
    int index = 0;
    while (index < 10)
        yield return index++;

    await Task.Delay(500);

    yield return 50;

    await Task.Delay(500);

    index = 100;
    while (index < 110)
        yield return index++;
}
```

这是同步和异步迭代器的语法。 我们来看一个真实示例。 假设你正在处理一个 IoT 项目，设备传感器生成了大量数据流。 为了获知数据，需要编写一个对每第 N 个数据元素进行采样的方法。 通过以下小迭代器方法可实现此目的：

C#

```
public static IEnumerable<T> Sample<T>(this IEnumerable<T> sourceSequence,
int interval)
{
    int index = 0;
    foreach (T item in sourceSequence)
    {
        if (index++ % interval == 0)
            yield return item;
```

```
    }
}
```

如果从 IoT 设备读取生成异步序列，则修改方法，如以下方法所示：

C#

```
public static async IAsyncEnumerable<T> Sample<T>(this IAsyncEnumerable<T>
sourceSequence, int interval)
{
    int index = 0;
    await foreach (T item in sourceSequence)
    {
        if (index++ % interval == 0)
            yield return item;
    }
}
```

迭代器方法有一个重要限制：在同一方法中不能同时使用 `return` 语句和 `yield return` 语句。 以下代码无法编译：

C#

```
public IEnumerable<int> GetSingleDigitNumbers()
{
    int index = 0;
    while (index < 10)
        yield return index++;

    yield return 50;

    // generates a compile time error:
    var items = new int[] {100, 101, 102, 103, 104, 105, 106, 107, 108, 109
};
    return items;
}
```

此限制通常不是问题。 可以选择在整个方法中使用 `yield return`，或选择将原始方法分成多个方法，一些使用 `return`，另一些使用 `yield return`。

可略微修改一下最后一个方法，使其可在任何位置使用 `yield return`：

C#

```
public IEnumerable<int> GetFirstDecile()
{
    int index = 0;
    while (index < 10)
        yield return index++;
```

```
    yield return 50;

    var items = new int[] {100, 101, 102, 103, 104, 105, 106, 107, 108, 109
};
    foreach (var item in items)
        yield return item;
}
```

有时，正确的做法是将迭代器方法拆分成 2 个不同的方法。 一个使用 `return`，另一个使用 `yield return`。 考虑这样一种情况：需要基于布尔参数返回一个空集合，或者返回前 5 个奇数。 可编写类似以下 2 种方法的方法：

C#

```
public IEnumerable<int> GetSingleDigitOddNumbers(bool getCollection)
{
    if (getCollection == false)
        return new int[0];
    else
        return IteratorMethod();
}

private IEnumerable<int> IteratorMethod()
{
    int index = 0;
    while (index < 10)
    {
        if (index % 2 == 1)
            yield return index;
        index++;
    }
}
```

看看上面的方法。 第 1 个方法使用标准 `return` 语句返回空集合，或返回第 2 个方法创建的迭代器。 第 2 个方法使用 `yield return` 语句创建请求的序列。

深入了解 `foreach`

`foreach` 语句可扩展为使用 `IEnumerable<T>` 和 `IEnumerator<T>` 接口的标准用语，以便循环访问集合中的所有元素。 还可最大限度减少开发人员因未正确管理资源所造成的错误。

编译器将第 1 个示例中显示的 `foreach` 循环转换为类似于此构造的内容：

C#

```
IEnumerator<int> enumerator = collection.GetEnumerator();
while (enumerator.MoveNext())
{
    var item = enumerator.Current;
    Console.WriteLine(item.ToString());
}
```

编译器生成的确切代码更复杂一些，用于处理 `GetEnumerator()` 返回的对象实现 `IDisposable` 接口的情况。 完整扩展生成的代码更类似如下：

C#

```
{
    var enumerator = collection.GetEnumerator();
    try
    {
        while (enumerator.MoveNext())
        {
            var item = enumerator.Current;
            Console.WriteLine(item.ToString());
        }
    }
    finally
    {
        // dispose of enumerator.
    }
}
```

编译器将第一个异步示例转换为类似于此构造的内容：

C#

```
{
    var enumerator = collection.GetAsyncEnumerator();
    try
    {
        while (await enumerator.MoveNextAsync())
        {
            var item = enumerator.Current;
            Console.WriteLine(item.ToString());
        }
    }
    finally
    {
        // dispose of async enumerator.
    }
}
```

枚举器的释放方式取决于 `enumerator` 类型的特征。 在常规同步情况下，`finally` 子句扩展为：

```C#
finally
{
   (enumerator as IDisposable)?.Dispose();
}
```

常规异步情况扩展为：

```C#
finally
{
    if (enumerator is IAsyncDisposable asyncDisposable)
        await asyncDisposable.DisposeAsync();
}
```

但是，如果 `enumerator` 的类型为已密封类型，并且不存在从类型 `enumerator` 到 `IDisposable` 或 `IAsyncDisposable` 的隐式转换，则 `finally` 子句扩展为一个空白块：

```C#
finally
{
}
```

如果存在从类型 `enumerator` 到 `IDisposable` 的隐式转换，并且 `enumerator` 是不可为 null 的值类型，则 `finally` 子句扩展为：

```C#
finally
{
   ((IDisposable)enumerator).Dispose();
}
```

幸运地是，无需记住所有这些细节。 `foreach` 语句会为你处理所有这些细微差别。 编译器会为所有这些构造生成正确的代码。

C# 中的委托和事件简介

项目 • 2023/11/20

在 .NET 中委托提供*后期绑定*机制。 后期绑定意味着调用方在你所创建的算法中至少提供一个方法来实现算法的一部分。

例如，在天文应用程序中对恒星列表进行排序。 你可以选择按照恒星与地球的距离、恒星的大小或者可以感知的亮度来对它们进行排序。

在所有这些情况下，Sort() 方法本质上执行的是同一操作：基于某种比较方法对列表中的项目进行排序。 对于每个排序顺序，比较两种恒星的代码是不同的。

这些类型的解决方案已在软件中使用了半个世纪。 C# 语言的委托概念提供一流的语言支持和以此概念为中心的类型安全性。

正如你将在本系列文章的后续部分所见，为此类算法编写的 C# 代码是类型安全的。 编译器可确保类型与参数和返回类型相匹配。

函数指针支持类似的方案，其中你需要对调用约定有更多的控制。 使用添加到委托类型的虚方法调用与委托关联的代码。 使用函数指针，可以指定不同的约定。

委托的语言设计目标

语言设计人员针对最终成为委托的功能列举了一些目标。

团队想要拥有可用于任何后期绑定算法的公共语言构造。 委托促使开发人员学习一个概念，并在许多不同的软件问题中使用这同一概念。

其次，该团队希望支持单一和多播方法调用。 （多播委托是将多个方法调用链接在一起的委托。你将在本系列文章的后面部分看到示例。）

团队想要委托在所有 C# 构造中支持开发人员所预期的相同的类型安全性。

最后，团队认识到事件模式是一个特定模式，委托或任何后期绑定算法在这种模式下都很有用。 团队需要确保委托的代码可以为 .NET 事件模式提供基础。

所有这些工作的结果是 C# 和 .NET 中的委托和事件支持。

本系列文章的剩余部分将介绍语言功能、库支持和使用委托和事件时使用的通用语法结构。 本文内容：

- `delegate` 关键字和它所生成的代码。
- `System.Delegate` 类中的功能，以及如何使用这些功能。
- 如何创建类型安全的委托。

- 如何创建可通过委托调用的方法。
- 如何使用 Lambda 表达式来处理委托和事件。
- 作为 LINQ 的构建基块的委托的用途。
- 委托如何成为 .NET 事件模式的基础，以及委托和事件之间的区别。

让我们开始吧。

下一页

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

System.Delegate 和 `delegate` 关键字

项目 • 2023/05/10

上一页

本文介绍 .NET 中支持委托的类以及这些类映射到 `delegate` 关键字的方式。

定义委托类型

我们从“delegate”关键字开始，因为这是你在使用委托时会使用的主要方法。 编译器在你使用 `delegate` 关键字时生成的代码会映射到调用 Delegate 和 MulticastDelegate 类的成员的方法调用。

可使用类似于定义方法签名的语法来定义委托类型。 只需向定义添加 `delegate` 关键字即可。

我们继续使用 List.Sort() 方法作为示例。 第一步是为比较委托创建类型：

```
C#

// From the .NET Core library

// Define the delegate type:
public delegate int Comparison<in T>(T left, T right);
```

编译器会生成一个类，它派生自与使用的签名匹配的 `System.Delegate`（在此例中，是返回一个整数并具有两个参数的方法）。 该委托的类型是 `Comparison`。 `Comparison` 委托类型是泛型类型。 有关泛型的详细信息，请参阅此处。

请注意，语法可能看起来像是声明变量，但它实际上是声明类型。 可以在类中、直接在命名空间中、甚至是在全局命名空间中定义委托类型。

> **备注**
>
> 建议不要直接在全局命名空间中声明委托类型（或其他类型）。

编译器还会为此新类型生成添加和移除处理程序，以便此类的客户端可以对实例的调用列表添加和移除方法。 编译器会强制所添加或移除的方法的签名与声明该方法时使用的签名匹配。

声明委托的实例

定义委托之后，可以创建该类型的实例。 与 C# 中的所有变量一样，不能直接在命名空间中或全局命名空间中声明委托实例。

C#

```
// inside a class definition:

// Declare an instance of that type:
public Comparison<T> comparator;
```

变量的类型是 `Comparison<T>`（前面定义的委托类型）。 变量的名称是 `comparator`。

上面的代码片段在类中声明了一个成员变量。 还可以声明作为局部变量或方法参数的委托变量。

调用委托

可通过调用某个委托来调用处于该委托调用列表中的方法。 在 `Sort()` 方法内部，代码会调用比较方法以确定放置对象的顺序：

C#

```
int result = comparator(left, right);
```

在上面的行中，代码会调用附加到委托的方法。 可将变量视为方法名称，并使用普通方法调用语法调用它。

该代码行进行了不安全假设：不保证目标已添加到委托。 如果未附加目标，则上面的行会导致引发 `NullReferenceException`。 用于解决此问题的惯例比简单 null 检查更加复杂，在此系列的后面部分中会进行介绍。

分配、添加和删除调用目标

这是委托类型的定义方式，以及声明和调用委托实例的方式。

要使用 `List.Sort()` 方法的开发人员需要定义签名与委托类型定义匹配的方法，并将它分配给排序方法使用的委托。 此分配会将方法添加到该委托对象的调用列表。

假设要按长度对字符串列表进行排序。 比较函数可能如下所示：

C#

```
private static int CompareLength(string left, string right) =>
    left.Length.CompareTo(right.Length);
```

方法声明为私有方法。 这没有什么不对。 你可能不希望此方法是公共接口的一部分。 它仍可以在附加到委托时用作比较方法。 调用代码会将此方法附加到委托对象的目标列表，并且可以通过该委托访问它。

通过将该方法传递给 `List.Sort()` 方法来创建该关系：

C#

```
phrases.Sort(CompareLength);
```

请注意，在不带括号的情况下使用方法名称。 将方法用作参数会告知编译器将方法引用转换为可以用作委托调用目标的引用，并将该方法作为调用目标进行附加。

还可以通过声明“`Comparison<string>`”类型的变量并进行分配来显式执行操作：

C#

```
Comparison<string> comparer = CompareLength;
phrases.Sort(comparer);
```

在用作委托目标的方法是小型方法的用法中，经常使用 lambda 表达式语法来执行分配：

C#

```
Comparison<string> comparer = (left, right) =>
left.Length.CompareTo(right.Length);
phrases.Sort(comparer);
```

后续部分中更详细地介绍了如何对委托目标使用 lambda 表达式。

Sort() 示例通常将单个目标方法附加到委托。 但是，委托对象支持将多个目标方法附加到委托对象的调用列表。

委托和 MulticastDelegate 类

上面介绍的语言支持可提供在使用委托时通常需要的功能和支持。 这些功能采用 .NET Core Framework 中的两个类进行构建：Delegate 和 MulticastDelegate。

`System.Delegate` 类及其单个直接子类 `System.MulticastDelegate` 可提供框架支持，以便创建委托、将方法注册为委托目标以及调用注册为委托目标的所有方法。

有趣的是，`System.Delegate` 和 `System.MulticastDelegate` 类本身不是委托类型。 它们为所有特定委托类型提供基础。 相同的语言设计过程要求不能声明派生自 `Delegate` 或 `MulticastDelegate` 的类。 C# 语言规则禁止这样做。

相反，C# 编译器会在你使用 C# 语言关键字声明委托类型时，创建派生自 `MulticastDelegate` 的类的实例。

此设计起源于 C# 和 .NET 的第一版。 设计团队的一个目标是确保在使用委托时，语言强制实施类型安全。 这意味着确保使用正确类型和数量的参数来调用委托。 并且在编译时正确指示任何返回类型。 委托是 1.0 .NET 版本的一部分（在泛型出现之前）。

强制实施此类型安全的最佳方法是让编辑器创建表示所使用的方法签名的具体委托类。

即使不能直接创建派生类，也会使用对这些类定义的方法。 我们来讨论一下在使用委托时会使用的最常见方法。

要记住的首要且最重要的事实是，使用的每个委托都派生自 `MulticastDelegate`。 多播委托意味着通过委托进行调用时，可以调用多个方法目标。 原始设计考虑区分只能附加并调用一个目标方法的委托与可以附加并调用多个目标方法的委托。 该区分被证明在实际中不如最初设想那么有用。 已创建了两个不同的类，并且自初始公开发行以来便一直处于框架中。

对委托最常使用的方法是 `Invoke()` 和 `BeginInvoke()` / `EndInvoke()`。 `Invoke()` 会调用已附加到特定委托实例的所有方法。 如上面所见，通常会通过对委托变量使用方法调用语法来调用委托。 如在此系列后面部分中所见，一些模式可直接使用这些方法。

现在你已了解支持委托的语言语法和类，我们来看一下如何使用、创建和调用强类型委托。

下一页

强类型委托

项目 • 2023/05/10

上一页

在上一篇文章中，使用 `delegate` 关键字创建了特定委托类型。

抽象的 Delegate 类提供用于松散耦合和调用的基础结构。 通过包含和实施添加到委托对象的调用列表的方法的类型安全性，具体的委托类型将变得更加有用。 使用 `delegate` 关键字并定义具体的委托类型时，编译器将生成这些方法。

实际上，无论何时需要不同的方法签名，这都会创建新的委托类型。 一段时间后此操作可能变得繁琐。 每个新功能都需要新的委托类型。

幸运的是，没有必要这样做。 .NET Core 框架包含几个在需要委托类型时可重用的类型。 这些是泛型定义，因此需要新的方法声明时可以声明自定义。

第一个类型是 Action 类型和一些变体：

C#

```
public delegate void Action();
public delegate void Action<in T>(T arg);
public delegate void Action<in T1, in T2>(T1 arg1, T2 arg2);
// Other variations removed for brevity.
```

有关协方差的文章中介绍了泛型类型参数的 `in` 修饰符。

`Action` 委托的变体可包含多达 16 个参数，如 Action<T1,T2,T3,T4,T5,T6,T7,T8,T9,T10,T11,T12,T13,T14,T15,T16>。 重要的是这些定义对每个委托参数使用不同的泛型参数：这样可以具有最大的灵活性。 方法参数不需要但可能是相同的类型。

对任何具有 void 返回类型的委托类型使用一种 `Action` 类型。

此框架还包括几种可用于返回值的委托类型的泛型委托类型：

C#

```
public delegate TResult Func<out TResult>();
public delegate TResult Func<in T1, out TResult>(T1 arg);
public delegate TResult Func<in T1, in T2, out TResult>(T1 arg1, T2 arg2);
// Other variations removed for brevity
```

有关协方差的文章中介绍了所产生的泛型类型参数的 `out` 修饰符。

`Func` 委托的变体可包含多达 16 个输入参数，如 Func<T1,T2,T3,T4,T5,T6,T7,T8,T9,T10,T11,T12,T13,T14,T15,T16,TResult>。 按照约定，结果的类型始终是所有 `Func` 声明中的最后一个类型参数。

对任何返回值的委托类型使用一种 `Func` 类型。

还有一种专门的委托类型 Predicate<T>，此类型返回单个值的测试结果：

C#

```
public delegate bool Predicate<in T>(T obj);
```

你可能会注意到对于任何 `Predicate` 类型，均存在一个在结构上等效的 `Func` 类型，例如：

C#

```
Func<string, bool> TestForString;
Predicate<string> AnotherTestForString;
```

你可能认为这两种类型是等效的。 它们不是。 这两个变量不能互换使用。 一种类型的变量无法赋予另一种类型。 C# 类型系统使用的是已定义类型的名称，而不是其结构。

.NET Core 库中的所有这些委托类型定义意味着你不需要为创建的任何需要委托的新功能定义新的委托类型。 这些泛型定义应已提供大多数情况下所需要的所有委托类型。 只需使用所需的类型参数实例化其中一个类型。 对于可成为泛型算法的算法，这些委托可以用作泛型类型。

这样可以节省时间，并尽量减少为了使用委托而需要创建的新类型的数目。

在下一篇文章中，你将看到在实践中使用委托的几种通用模式。

下一页

委托的常见模式

项目 • 2023/04/07

上一篇

委托提供了一种机制，可实现涉及组件间最小耦合度的软件设计。

此类设计的出色示例为 LINQ。 LINQ 查询表达式模式依赖于其所有功能的委托。 请考虑此简单示例：

C#

```
var smallNumbers = numbers.Where(n => n < 10);
```

这会将数字序列筛选为仅小于值 10 的数字序列。 `Where` 方法使用委托来确定序列的哪些元素可通过筛选器。 创建 LINQ 查询时，为此特定目的提供委托的实现。

Where 方法的原型是：

C#

```
public static IEnumerable<TSource> Where<TSource> (this IEnumerable<TSource>
source, Func<TSource, bool> predicate);
```

此示例重复使用所有方法，这些方法都为 LINQ 的一部分。 它们都依赖于管理特定查询的代码委托。 此 API 设计模式功能强大，需要学习和理解。

此简单示例说明了委托在组件之间仅需要极少耦合度的原因。 不需要创建从特定基类派生的类。 不需要实现特定接口。 唯一的要求是提供对当前任务至关重要的方法实现。

通过委托生成自己的组件

基于此示例，通过使用依赖于委托的设计来创建组件，从而进行生成。

定义一个可用于大型系统中日志消息的组件。 库组件可以在多种不同的环境中和多个不同的平台上使用。 管理日志的组件中有很多常用功能。 它需要接受来自系统中任何组件的消息。 这些消息将具有不同的优先级（核心组件可进行管理）。 消息应当具有其最终存档形式的时间戳。 对于更高级的方案，你可以按源组件筛选消息。

此功能有一个方面会经常发生变化：写入消息的位置。 在某些环境中，它们可能会写入到错误控制台。 在其他环境中，可能会写入一个文件。 其他可能性包括数据库存储、操作系统事件日志或其他文档存储。

还有可能用于不同方案的输出组合。 建议你将消息写入控制台和文件。

基于委托的设计将提供极大的灵活性，从而轻松支持可能在以后添加的存储机制。

基于此设计，主日志组件可以是非虚拟，甚至是密封的类。 你可以插入任何委托集，将消息写入不同的存储介质。 对多播委托的内置支持有助于支持必须将消息写入多个位置（文件和控制台）的情况。

首次实现

我们从小处着手：初始实现会接受新消息并使用任意附加委托编写它们。 你可以从一个将消息写入控制台的委托开始。

C#

```
public static class Logger
{
    public static Action<string>? WriteMessage;

    public static void LogMessage(string msg)
    {
        if (WriteMessage is not null)
            WriteMessage(msg);
    }
}
```

上面的静态类是可以发挥作用的最简单的类。 我们需要编写将消息写入控制台的方法的单个实现：

C#

```
public static class LoggingMethods
{
    public static void LogToConsole(string message)
    {
        Console.Error.WriteLine(message);
    }
}
```

最后，你需要通过将委托附加到记录器中声明的 WriteMessage 委托来进行挂钩：

C#

```
Logger.WriteMessage += LoggingMethods.LogToConsole;
```

实践

到目前为止，我们的示例都相当简单，但仍演示了一些关于委托设计的重要指南。

借助在核心框架中定义的委托类型，用户可更轻松地使用委托。 无需定义新类型，而且使用你库的开发者不需要学习新的专用委托类型。

使用的接口尽可能小且灵活：若要创建新的输出记录器，必须创建一个方法。 该方法可以是静态方法或实例方法。 它可能具有任何访问权限。

设置输出格式

让第一个版本更加可靠，然后开始创建其他日志记录机制。

然后，向 `LogMessage()` 方法添加一些参数，以便日志类创建更多结构化消息：

C#

```
public enum Severity
{
    Verbose,
    Trace,
    Information,
    Warning,
    Error,
    Critical
}
```

C#

```
public static class Logger
{
    public static Action<string>? WriteMessage;

    public static void LogMessage(Severity s, string component, string msg)
    {
        var outputMsg = $"{DateTime.Now}\t{s}\t{component}\t{msg}";
        if (WriteMessage is not null)
            WriteMessage(outputMsg);
    }
}
```

接下来，使用 `Severity` 参数来筛选发送到日志输出的消息。

C#

```
public static class Logger
{
    public static Action<string>? WriteMessage;

    public static Severity LogLevel { get; set; } = Severity.Warning;

    public static void LogMessage(Severity s, string component, string msg)
    {
        if (s < LogLevel)
            return;

        var outputMsg = $"{DateTime.Now}\t{s}\t{component}\t{msg}";
        if (WriteMessage is not null)
            WriteMessage(outputMsg);
    }
}
```

实践

已向日志记录基础结构添加了新功能。 由于记录器组件极其松散地耦合到输出机制，因此可在不影响代码实现记录器托管的情况下添加新功能。

继续构建，你会看到更多的示例，其中显示这种松散的耦合度在更新站点部件方面实现了很高的灵活性，而不会对其他位置做出更改。 实际上，在更大的应用程序中，记录器输出类可能位于不同的程序集中，甚至不需要重新生成。

生成第二个输出引擎

还将附带日志组件。 我们再添加一个将消息记录到文件的输出引擎。 这将是一个更为普及的输出引擎。 它将是一个封装文件操作的类，并确保文件在每次写入后始终处于关闭状态。 这可以确保生成每条消息后将所有数据刷新到磁盘。

下面是基于文件的记录器：

C#

```
public class FileLogger
{
    private readonly string logPath;
    public FileLogger(string path)
    {
        logPath = path;
        Logger.WriteMessage += LogMessage;
    }

    public void DetachLog() => Logger.WriteMessage -= LogMessage;
    // make sure this can't throw.
```

```
    private void LogMessage(string msg)
    {
        try
        {
            using (var log = File.AppendText(logPath))
            {
                log.WriteLine(msg);
                log.Flush();
            }
        }
        catch (Exception)
        {
            // Hmm. We caught an exception while
            // logging. We can't really log the
            // problem (since it's the log that's failing).
            // So, while normally, catching an exception
            // and doing nothing isn't wise, it's really the
            // only reasonable option here.
        }
    }
}
```

创建此类后，可将它进行实例化，然后它会将 LogMessage 方法附加到记录器组件中：

C#

```
var file = new FileLogger("log.txt");
```

这两项并不互相排斥。 你可以附加这两种日志方法并生成要发送到控制台和文件的消息：

C#

```
var fileOutput = new FileLogger("log.txt");
Logger.WriteMessage += LoggingMethods.LogToConsole; // LoggingMethods is the
static class we utilized earlier
```

以后，即使在同一个应用程序中，也可在不对系统产生任何其他问题的情况下删除其中一个委托：

C#

```
Logger.WriteMessage -= LoggingMethods.LogToConsole;
```

实践

现在，你已添加日志记录子系统的第二个输出处理程序。 这需要更多的基础结构来正确支持文件系统。 此委托为实例方法。 其为私有方法。 由于委托基础结构可以连接委托，因此不需要太高的可访问性。

其次，基于委托的设计可实现多种输出方法，且无需额外的代码。 无需生成任何其他基础结构来支持多种输出方法。 它们将变为调用列表上的另一种方法。

需要特别注意文件日志记录输出方法中的代码。 对其进行编码以确保不引发任何异常。 虽然不是绝对必要，但这通常是很好的做法。 如果任意一种委托方法引发异常，将不会调用该调用中剩余的其他委托。

最后请注意，文件记录器必须通过打开和关闭每条日志消息上的文件来管理其资源。 可以选择让文件保持打开状态，并在完成后执行 `IDisposable` 以关闭文件。 这两种方法各有利弊。 两者都在类之间创建了更高的耦合度。

为了支持这两种方案，`Logger` 类中的代码都不需要更新。

处理 NULL 委托

最后，更新 LogMessage 方法，从而在没有选择输出机制的情况下更加可靠。 `WriteMessage` 委托没有附加调用列表时，当前实现将引发 `NullReferenceException`。 你可能更需要在没有附加方法时自行继续的设计。 将 null 条件运算符与 `Delegate.Invoke()` 方法结合使用时，很容易实现该目标：

```
C#
```

```
public static void LogMessage(string msg)
{
    WriteMessage?.Invoke(msg);
}
```

当左操作数（本例中为 `WriteMessage`）为 null 时，null 条件运算符（`?.`）会短路，这意味着不会尝试记录消息。

不会在 `System.Delegate` 或 `System.MulticastDelegate` 的文档中列出 `Invoke()` 方法。 编译器将为声明的所有委托类型生成类型安全的 `Invoke` 方法。 在此示例中，这意味着 `Invoke` 只需要一个 `string` 参数，并且有一个无效返回类型。

实践摘要

你已了解日志组件的起始部分，可以使用其他编写器和其他功能进行扩展。 通过在设计中使用委托，这些不同的组件松散地耦合在一起。 这样可提供多种优势。 可轻松创建新

的输出机制并将它们附加到系统中。 这些机制只需要一种方法：编写日志消息的方法。 这种设计在添加新功能时有复原能力。 所有编写器所需的协定都是为了实现一种方法。 该方法可以是静态方法或实例方法。 它可以是公用、专用或任何其他合法访问。

记录器类可在不引入重大更改的情况下进行任何数量的增强或更改。 与类相似，无法在没有重大更改的风险下修改公共 API。 但是，因为仅通过委托进行记录器和输出引擎之间的耦合，因此不涉及其他类型（如接口或基类）。 耦合度越小越好。

下一页

事件介绍

项目 • 2023/05/10

上一页

和委托类似，事件是*后期绑定*机制。 实际上，事件是建立在对委托的语言支持之上的。

事件是对象用于（向系统中的所有相关组件）广播已发生事情的一种方式。 任何其他组件都可以订阅事件，并在事件引发时得到通知。

你可能已在某些编程中使用过事件。 许多图形系统都具有用于报告用户交互的事件模型。 这些事件会报告鼠标移动、按钮点击和类似的交互。 这是使用事件的最常见情景之一，但并非唯一的情景。

可以定义应针对类引发的事件。 使用事件时，需要注意的一点是特定事件可能没有任何注册的对象。 必须编写代码，以确保在未配置侦听器时不会引发事件。

通过订阅事件，还可在两个对象（事件源和事件接收器）之间创建耦合。 需要确保当不再对事件感兴趣时，事件接收器将从事件源取消订阅。

事件支持的设计目标

事件的语言设计针对这些目标：

- 在事件源和事件接收器之间启用非常小的耦合。 这两个组件可能不会由同一个组织编写，甚至可能会通过完全不同的计划进行更新。
- 订阅事件并从同一事件取消订阅应该非常简单。
- 事件源应支持多个事件订阅服务器。 它还应支持不附加任何事件订阅服务器。

你会发现事件的目标与委托的目标非常相似。 因此，事件语言支持基于委托语言支持构建。

事件的语言支持

用于定义事件以及订阅或取消订阅事件的语法是对委托语法的扩展。

定义使用 `event` 关键字的事件：

C#

```
public event EventHandler<FileListArgs> Progress;
```

该事件（在此示例中，为 `EventHandler<FileListArgs>`）的类型必须为委托类型。 声明事件时，应遵循许多约定。 通常情况下，事件委托类型具有无效的返回。 事件声明应为谓词或谓词短语。 当事件报告已发生的事情时，请使用过去时。 使用现在时谓词（例如 `Closing`）报告将要发生的事情。 通常，使用现在时表示类支持某种类型的自定义行为。 最常见的方案之一是支持取消。 例如，`Closing` 事件可能包括指示是否应继续执行关闭操作的参数。 其他方案可能会允许调用方通过更新事件参数的属性来修改行为。 你可以引发一个事件以指示算法将采取的建议的下一步操作。 事件处理程序可以通过修改事件参数的属性授权不同的操作。

想要引发事件时，使用委托调用语法调用事件处理程序：

C#

```
Progress?.Invoke(this, new FileListArgs(file));
```

如委托部分中所介绍的那样，?. 运算符可以轻松确保在事件没有订阅服务器时不引发事件。

通过使用 `+=` 运算符订阅事件：

C#

```
EventHandler<FileListArgs> onProgress = (sender, eventArgs) =>
    Console.WriteLine(eventArgs.FoundFile);

fileLister.Progress += onProgress;
```

处理程序方法通常为前缀“On”，后跟事件名称，如上所示。

使用 `-=` 运算符取消订阅：

C#

```
fileLister.Progress -= onProgress;
```

请务必为表示事件处理程序的表达式声明局部变量。 这将确保取消订阅删除该处理程序。 如果使用的是 lambda 表达式的主体，则将尝试删除从未附加过的处理程序，此操作为无效操作。

下一篇文章将介绍有关典型事件模式及此示例的不同变体的详细信息。

下一部分

标准 .NET 事件模式

项目 • 2023/05/10

上一篇

.NET 事件通常遵循几种已知模式。 标准化这些模式意味着开发人员可利用这些标准模式的相关知识，将其应用于任何 .NET 事件程序。

让我们开始通览这些标准模式，以便你掌握创建标准事件源、在代码中订阅和处理标准事件所需的全部知识。

事件委托签名

.NET 事件委托的标准签名是：

C#

```
void EventRaised(object sender, EventArgs args);
```

返回类型为 void。 事件基于委托，而且是多播委托。 对任何事件源都支持多个订阅服务器。 来自方法的单个返回值不会扩展到多个事件订阅服务器。 引发事件后事件源的返回值是什么？ 稍后在本文中将介绍如何创建事件协议，以支持事件订阅服务器向事件源报告信息。

参数列表包含两种参数：发件人和事件参数。 `sender` 的编译时类型为 `System.Object`，即使有一个始终正确的更底层派生的类型亦是如此。 按照惯例使用 `object`。

第二种参数通常是派生自 `System.EventArgs` 的类型。 （你将在下一部分中看到不再强制执行此约定。）如果事件类型不需要任何其他参数，你仍将提供这两个参数。 应使用特殊值 `EventArgs.Empty` 来表示事件不包含任何附加信息。

让我们生成一个类，它在目录或遵循模式的任何子目录中列出文件。 此组件为每个找到的与模式相匹配的文件引发事件。

使用事件模型有一些设计优势。 可以创建多个事件侦听器，用于在找到查找的文件时执行不同的操作。 合并不同的侦听器可以创建更可靠的算法。

下面是找到查找的文件时的初始事件参数声明：

C#

```
public class FileFoundArgs : EventArgs
{
```

```
    public string FoundFile { get; }

    public FileFoundArgs(string fileName) => FoundFile = fileName;
}
```

尽管这种类型看上去是小型的仅限数据的类型，但仍应按约定将其设为引用（`class`）类型。 这意味着参数对象将通过引用来传递，并且所有订阅服务器都将查看到任何数据更新。 第一版是不可变对象。 应优先将事件参数类型中的属性设为不可变。 这样一来，一个订阅服务器在其他订阅服务器看到值之前便无法更改值。 （但对此也有例外，如下所示。）

接下来，我们要在 FileSearcher 类中创建事件声明。 利用 `EventHandler<T>` 类型意味着尚无需创建其他类型定义。 只需使用泛型专业化即可。

让我们通过填充 FileSearcher 类来搜索与模式匹配的文件，并在发现匹配时引发正确的事件。

C#

```
public class FileSearcher
{
    public event EventHandler<FileFoundArgs>? FileFound;

    public void Search(string directory, string searchPattern)
    {
        foreach (var file in Directory.EnumerateFiles(directory,
searchPattern))
        {
            RaiseFileFound(file);
        }
    }

    private void RaiseFileFound(string file) =>
        FileFound?.Invoke(this, new FileFoundArgs(file));
}
```

定义并引发类似字段的事件

要将事件添加到类，最简单的方式是将该事件声明为公共字段，如上面的示例中所示：

C#

```
public event EventHandler<FileFoundArgs>? FileFound;
```

看起来它像在声明一个公共字段，这似乎是一个面向对象的不良实践。 你希望通过属性或方法来保护数据访问。 虽然这可能看起来是糟糕的做法，但编译器生成的代码确实会

创建包装器，以便事件对象只能通过安全的方式进行访问。 类似字段的事件上唯一可用的操作是添加处理程序：

```
C#

var fileLister = new FileSearcher();
int filesFound = 0;

EventHandler<FileFoundArgs> onFileFound = (sender, eventArgs) =>
{
    Console.WriteLine(eventArgs.FoundFile);
    filesFound++;
};

fileLister.FileFound += onFileFound;
```

和删除处理程序：

```
C#

fileLister.FileFound -= onFileFound;
```

请注意，处理程序有一个局部变量。 如果使用了 lambda 的正文，则删除操作无法正常进行。 它将成为不同的委托实例，并静默地不执行任何操作。

类之外的代码无法引发事件，也不能执行任何其它操作。

从事件订阅服务器返回值

你的简单版本当前运行正常。 让我们添加另一项功能：取消。

在引发找到的事件时，如果此文件是最后查找到的文件，则侦听器应能够停止进一步的处理。

事件处理程序不返回值，因此需以其它方式进行通信。 标准事件模式使用 `EventArgs` 对象来包含字段，事件订阅服务器使用这些字段进行通信取消。

根据"取消"协定的语义，可使用两种不同的模式。 在两种情况下，你都将为找到的文件事件向 EventArguments 添加布尔字段。

其中一种模式允许任一订阅服务器取消操作。 在此模式下，新字段会初始化为 `false`。 任何订阅服务器都可将其更改为 `true`。 当所有订阅服务器观察到事件已引发后，FileSearcher 组件将检查布尔值，并执行操作。

在第二种模式下，仅当所有订阅服务器都要取消操作时才可取消操作。 在此模式下，新字段会初始化为指示操作应取消，而任何订阅服务器都可将其更改为指示操作应继续。 当所有订阅服务器观察到事件已引发后，FileSearcher 组件将检查布尔，并执行操作。 此模式还有一个额外步骤：组件需知道是否有任何订阅服务器已经看到过该事件。 如果没有订阅服务器，字段会错误地指示取消。

让我们来实现此示例的第一版。 需要将名为 `CancelRequested` 的布尔字段添加到 `FileFoundArgs` 类型：

C#

```
public class FileFoundArgs : EventArgs
{
    public string FoundFile { get; }
    public bool CancelRequested { get; set; }

    public FileFoundArgs(string fileName) => FoundFile = fileName;
}
```

此新字段将自动初始化为 `false`，即 `Boolean` 字段的默认值，因此不会意外取消。 对组件进行的唯一其它更改是在引发事件后检查标志，查看是否有任何订阅服务器提出了取消请求：

C#

```
private void SearchDirectory(string directory, string searchPattern)
{
    foreach (var file in Directory.EnumerateFiles(directory, searchPattern))
    {
        FileFoundArgs args = RaiseFileFound(file);
        if (args.CancelRequested)
        {
            break;
        }
    }
}

private FileFoundArgs RaiseFileFound(string file)
{
    var args = new FileFoundArgs(file);
    FileFound?.Invoke(this, args);
    return args;
}
```

此模式的一个优点是不会造成重大更改。 在此之前没有订阅服务器请求取消，现在也不会有。 没有任何订阅服务器代码需要更新，除非它们想支持新的取消协议。 这是极为松散耦合的。

让我们来更新订阅服务器，使其在找到第一个可执行文件时请求取消：

C#

```
EventHandler<FileFoundArgs> onFileFound = (sender, eventArgs) =>
{
    Console.WriteLine(eventArgs.FoundFile);
    eventArgs.CancelRequested = true;
};
```

添加另一个事件声明

让我们再添加一项功能，并演示事件的其它语言习惯用语。 让我们添加搜索文件时遍历所有子目录的 `Search` 方法的重载。

在拥有多个子目录的目录中，此操作可能要花较长时间。 让我们添加一个在每次新目录搜索开始时引发的事件。 这让订阅服务器可以跟踪进度，并根据进度更新用户。 目前为止，你所创建的所有示例都是公共的。 让我们把这个示例设为内部事件。 这意味着你也可以将这些类型用于参数内部。

首先，创建新的 EventArgs 派生类，用于报告新目录和进度。

C#

```
internal class SearchDirectoryArgs : EventArgs
{
    internal string CurrentSearchDirectory { get; }
    internal int TotalDirs { get; }
    internal int CompletedDirs { get; }

    internal SearchDirectoryArgs(string dir, int totalDirs, int
completedDirs)
    {
        CurrentSearchDirectory = dir;
        TotalDirs = totalDirs;
        CompletedDirs = completedDirs;
    }
}
```

同样，可以根据建议为事件参数设置不可变的引用类型。

接下来，定义事件。 此时，需使用不同的语法。 除使用字段语法之外，还可以显式创建包含添加或删除处理程序的属性。 在本例中，这些处理程序中无需包含额外的代码，但这一步演示了你可以如何创建它们。

C#

```
internal event EventHandler<SearchDirectoryArgs> DirectoryChanged
{
    add { _directoryChanged += value; }
    remove { _directoryChanged -= value; }
}
private EventHandler<SearchDirectoryArgs>? _directoryChanged;
```

在许多方面，此处编写的代码可反映编译器为你已见过的字段事件定义所生成的代码。 创建事件所使用的语法与用于属性的语法是极为相似的。 请注意，处理程序的名称各不相同：`add` 和 `remove`。 通过调用它们来订阅事件，或取消订阅事件。 请注意，还必须声明一个私有支持字段以存储事件变量。 它初始化为 null。

接下来，让我们添加 `Search` 方法的重载，该重载遍历子目录，并引发这两个事件。 要实现此目的，最简单的方法是使用默认参数来指定你要搜索所有目录：

C#

```
public void Search(string directory, string searchPattern, bool
searchSubDirs = false)
{
    if (searchSubDirs)
    {
        var allDirectories = Directory.GetDirectories(directory, "*.*",
SearchOption.AllDirectories);
        var completedDirs = 0;
        var totalDirs = allDirectories.Length + 1;
        foreach (var dir in allDirectories)
        {
            RaiseSearchDirectoryChanged(dir, totalDirs, completedDirs++);
            // Search 'dir' and its subdirectories for files that match the
search pattern:
            SearchDirectory(dir, searchPattern);
        }
        // Include the Current Directory:
        RaiseSearchDirectoryChanged(directory, totalDirs, completedDirs++);

        SearchDirectory(directory, searchPattern);
    }
    else
    {
        SearchDirectory(directory, searchPattern);
    }
}

private void SearchDirectory(string directory, string searchPattern)
{
    foreach (var file in Directory.EnumerateFiles(directory, searchPattern))
    {
        FileFoundArgs args = RaiseFileFound(file);
        if (args.CancelRequested)
        {
```

```
            break;
        }
    }
}

private void RaiseSearchDirectoryChanged(
    string directory, int totalDirs, int completedDirs) =>
    _directoryChanged?.Invoke(
        this,
            new SearchDirectoryArgs(directory, totalDirs, completedDirs));

private FileFoundArgs RaiseFileFound(string file)
{
    var args = new FileFoundArgs(file);
    FileFound?.Invoke(this, args);
    return args;
}
```

此时可运行调用重载的应用程序来搜索所有子目录。 虽然新 `DirectoryChanged` 事件中没有订阅服务器，但使用 `?.Invoke()` 习惯用语可确保此操作正常。

让我们通过添加处理程序来编写一行，用于在控制台窗口显示进度。

C#

```
fileLister.DirectoryChanged += (sender, eventArgs) =>
{
    Console.Write($"Entering '{eventArgs.CurrentSearchDirectory}'.");
    Console.WriteLine($" {eventArgs.CompletedDirs} of {eventArgs.TotalDirs}
completed...");
};
```

你已了解了整个 .NET 生态系统所遵循的模式。 通过学习这些模式和约定，将能够快速编写惯用的 C# 和 .NET。

另请参阅

- 事件简介
- 事件设计
- 处理和引发事件

接下来，将了解在 .NET 的最新版本中关于这些模式的一些更改。

下一页

更新的 .NET Core 事件模式

项目 • 2023/05/10

上一篇

上一篇文章讨论了最常见的事件模式。 .NET Core 的模式较为宽松。 在此版本中，`EventHandler<TEventArgs>` 定义不再要求 `TEventArgs` 必须是派生自 `System.EventArgs` 的类。

这就提高了灵活性，并且还具有后向兼容性。 首先讨论灵活性。 类 System.EventArgs 引入了一个方法 `MemberwiseClone()`，该方法可创建对象的浅表副本。 对于任何派生自 `EventArgs` 的类，该方法必须使用反射才能实现其功能。 该功能在特定的派生类中更容易创建。 实际上，这意味着派生自 System.EventArgs 的类会限制你的设计，且不会为你提供任何附加好处。 其实，可以更改 `FileFoundArgs` 和 `SearchDirectoryArgs` 的定义，使它们不从 `EventArgs` 派生。 该程序的工作原理相同。

如果还要进行一处更改，还可将 `SearchDirectoryArgs` 更改为结构：

C#

```
internal struct SearchDirectoryArgs
{
    internal string CurrentSearchDirectory { get; }
    internal int TotalDirs { get; }
    internal int CompletedDirs { get; }

    internal SearchDirectoryArgs(string dir, int totalDirs, int
completedDirs) : this()
    {
        CurrentSearchDirectory = dir;
        TotalDirs = totalDirs;
        CompletedDirs = completedDirs;
    }
}
```

其他更改为：在输入初始化所有字段的构造函数之前调用无参数构造函数。 若没有此添加，C# 规则将报告先访问属性再分配属性。

不应将 `FileFoundArgs` 从类（引用类型）更改为结构（值类型）。 这是因为处理取消的协议要求通过引用传递事件参数。 如果进行了相同的更改，文件搜索类将永远不会观察到任何事件订阅者所做的任何更改。 结构的新副本将用于每个订阅者，并且该副本将与文件搜索对象所看到的不同。

接下来，让我们考虑这种更改如何具有后向兼容性。 删除约束不会影响任何现有代码。 任何现有的事件参数类型仍然派生自 `System.EventArgs`。 它们将继续从

`System.EventArgs` 派生，其中一个主要原因就是后向兼容性。 任何现有的事件订阅者都是遵循经典模式的事件的订阅者。

遵循类似的逻辑，现在创建的任何事件参数类型在任何现有代码库中都不会有任何订阅者。 只有从 `System.EventArgs` 派生的新事件类型才会破坏这些代码库。

异步事件订阅者

你还需了解最后一个模式：如何正确编写调用异步代码的事件订阅者。 该问题详见 async 和 await 一文。 异步方法可具有一个 void 返回类型，但强烈建议不要使用它。 事件订阅者代码调用异步方法时，只能创建 `async void` 方法。 事件处理程序签名需要该方法。

你需要协调此对立指南。 不管怎样，必须创建安全的 `async void` 方法。 需要实现的模式的基础知识如下：

C#

```
worker.StartWorking += async (sender, eventArgs) =>
{
    try
    {
        await DoWorkAsync();
    }
    catch (Exception e)
    {
        //Some form of logging.
        Console.WriteLine($"Async task failure: {e.ToString()}");
        // Consider gracefully, and quickly exiting.
    }
};
```

首先请注意，处理程序已被标记为异步处理程序。 因为它将被分配给一个事件处理程序委托类型，所以它将有一个 void 返回类型。 这意味着必须遵循处理程序中显示的模式，并且不允许在异步处理程序上下文之外引发异常。 因为它不返回任务，所以没有任何可通过进入故障状态报告错误的任务。 因为方法是异步的，所以不能简单地引发异常。（调用方法继续执行，因为它是 `async`。）实际运行时行为将针对不同环境以不同的方式定义。 它可以终止线程或拥有线程的进程，也可以使进程处于不确定状态。 所有这些潜在的结果都非常不理想。

这就是为什么你应该在自己的 try 块中包装异步任务的 await 语句。 如果它的确导致任务出错，则可以记录该错误。 如果它是应用程序无法从中恢复的错误，则可以迅速优雅地退出此程序。

这些就是 .NET 事件模式的主要更新。 你将在所使用的库中看到许多早期版本的示例。 但是，你也应了解最新的模式是什么。

本系列的下一篇文章将有助于你区分在设计中使用 `delegates` 和 `events`。 它们是类似的概念，该文章将帮助你为程序做出最好的决定。

下一页

区别委托和事件

项目 • 2023/05/10

上一篇

对不熟悉 .NET Core 平台的开发人员而言，在基于 `delegates` 的设计和基于 `events` 的设计之间做出选择是困难的。 委托或事件的选择通常比较难，因为这两种语言功能很相似。 事件甚至是使用委托的语言支持构建的。

它们都提供了一个后期绑定方案：在该方案中，组件通过调用仅在运行时识别的方法进行通信。 它们都支持单个和多个订阅服务器方法。 这称为单播和多播支持。 二者均支持用于添加和删除处理程序的类似语法。 最后，引发事件和调用委托使用完全相同的方法调用语法。 它们甚至都支持与 `?.` 运算符一起使用的相同的 `Invoke()` 方法语法。

鉴于所有这些相似之处，很难确定何时使用何种语法。

侦听事件是可选的

在确定要使用的语言功能时，最重要的考虑因素为是否必须具有附加的订阅服务器。 如果代码必须调用订阅服务器提供的代码，则在需要实现回调时，应使用基于委托的设计。 如果你的代码在不调用任何订阅服务器的情况下可完成其所有工作，则应使用基于事件的设计。

请考虑本部分中生成的示例。 必须为使用 `List.Sort()` 生成的代码提供 comparer 函数，以便对元素进行正确排序。 必须与委托一起提供 LINQ 查询，以便确定要返回的元素。 二者均使用与委托一起生成的设计。

请考虑 `Progress` 事件。 它会报告任务进度。 无论是否具有侦听器，该任务将继续进行。 `FileSearcher` 是另一个示例。 即使没有附加事件订阅服务器，它仍将搜索和查找已找到的所有文件。 即使没有任何订阅服务器侦听事件，UX 控件仍正常工作。 它们都使用基于事件的设计。

返回值需要委托

另一个注意事项是委托方法所需的方法原型。 如你所见，用于事件的委托均具有无效的返回类型。 你还看到，存在创建事件处理程序的惯用语，该事件处理程序通过修改事件参数对象的属性将信息传回到事件源。 虽然这些惯用语可发挥作用，但它们不像从方法返回值那样自然。

请注意，这两种试探法可能经常同时存在：如果委托方法返回值，则可能会以某种方式影响算法。

事件具有专用调用

包含事件的类以外的类只能添加和删除事件侦听器；只有包含事件的类才能调用事件。 事件通常是公共类成员。 相比之下，委托通常作为参数传递，并存储为私有类成员（如果它们全部存储）。

事件侦听器通常具有较长的生存期

事件侦听器通常具有较长的生存期的这一理由不太充分。 但是，你可能会发现，当事件源将在很长一段时间内引发事件时，基于事件的设计会更加自然。 可以在许多系统上看到基于事件的 UX 控件设计示例。 订阅事件后，事件源可能会在程序的整个生存期内引发事件。 （当不再需要事件时，可以取消订阅事件。）

将其与许多基于委托的设计（其中委托用作方法的参数，且在返回该方法后不再使用此委托）进行比较。

仔细评估

以上考虑因素并非固定不变的规则。 相反，它们代表可帮助决定针对特定使用情况的最佳选择的指南。 因为两者类似，所以甚至可以将两者作为原型，并考虑使用更加自然的一种。 两者均能很好地处理后期绑定方案。 使用能与设计进行最佳通讯的一种。

C# 中的版本控制

项目 • 2024/10/01

本教程将介绍版本控制在 .NET 中的含义。 还将介绍对库进行版本控制以及升级到新版本的库时需要考虑的因素。

语言版本

C# 编译器是 .NET SDK 的一部分。 默认情况下，编译器选择与项目所选 TFM 匹配的 C# 语言版本。 如果 SDK 版本大于所选框架，编译器可能会使用更高的语言版本。 通过在项目中设置 `LangVersion` 元素，可以更改默认值。 可以在有关编译器选项的文章中了解如何实现。

> **⚠ 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。

创作库

对于创建 .NET 库以供公共使用的开发人员，经常需要推出新更新。 如何处理此过程关系重大，因为开发人员需确保从现有代码无缝转换到新版本的库。 以下是创建新版本时的几个注意事项：

语义版本控制

语义版本控制 （简称 SemVer）是应用于库版本的命名约定，用于表示特定里程碑事件。 理想情况下，提供给库的版本信息应帮助开发人员确定版本是否与使用相同库的早期版本的项目兼容。

SemVer 的最基本方法是 3 组件格式 `MAJOR.MINOR.PATCH`，其中：

- 进行不兼容的 API 更改时，`MAJOR` 将会增加
- 以后向兼容方式添加功能时，`MINOR` 将会增加
- 进行后向兼容 bug 修复时，`PATCH` 将会增加

将版本信息应用于 .NET 库时，还可通过其他方法指定其他方案，例如预发布版本。

后向兼容

发布新版本的库时，与先前版本的后向兼容很可能成为主要关注事项之一。 如果重新编译时，依赖于先前版本的代码适用于新版本，则新版本的库与先前版本是源兼容的。 在没有重新编译的情况下，如果依赖于先前版本的应用程序适用于新版本，则新版本的库是二进制兼容的。

以下是维护与较旧版本库的后向兼容时的注意事项：

- 虚拟方法：如果在新版本中使虚拟方法成为非虚拟方法，则必须更新替代该方法的项目。 这是一项重大更改，强烈建议不要执行此操作。
- 方法签名：虽然更新方法行为也需要更改其签名，但应创建重载，使调用该方法的代码仍可正常运行。 始终可以使用旧方法签名来调用新方法签名，以使实现保持一致。
- 已过时属性：可在代码中使用此属性指定已弃用且很可能在将来版本中删除的类或类成员。 这可确保使用此库的开发人员能更好地为重大更改做好准备。
- 可选方法参数：如果使以前的可选方法参数变为强制性方法参数或更改它们的默认值，则需要更新不提供这些参数的所有代码。

> **备注**
>
> 将强制性参数变为可选参数应几乎没有影响，尤其是在不更改方法的行为的情况下。

为用户提供的升级到新版本库的方法越简单，用户升级的速度很可能会越快。

应用程序配置文件

.NET 开发人员很可能已在大多数项目类型中遇到过 app.config 文件。 此类简单配置文件对于改进新更新的推出有重要作用。 通常应以以下方式设计库：将可能定期更改的信息存储在 `app.config` 文件中。这样，当更新此类信息时，只需使用新版本的配置文件替换旧版本配置文件即可，而无需重新编译库。

使用库

作为使用由其他开发人员构建的 .NET 库的开发人员，你很可能已经发现新版本的库可能不与项目完全兼容，你经常需要更新代码以使用这些更改。

幸运的是，C# 和 .NET 生态系统附带一些功能和技术，通过这些功能和技术，我们可以轻松更新应用，使其适用于可能引入重大更改的新版本库。

程序集绑定重定向

可使用 app.config 文件更新应用使用的库版本。 通过添加所谓的绑定重定向，可在无需重新编译应用的情况下使用新的库版本。 下面的示例演示了更新应用的 app.config 文件的方法，以便使用 `ReferencedLibrary` 的 `1.0.1` 修补程序版本，而不是最初编译时使用的 `1.0.0` 版本。

XML

```
<dependentAssembly>
    <assemblyIdentity name="ReferencedLibrary"
publicKeyToken="32ab4ba45e0a69a1" culture="en-us" />
    <bindingRedirect oldVersion="1.0.0" newVersion="1.0.1" />
</dependentAssembly>
```

> **备注**
>
> 仅当 `ReferencedLibrary` 的新版本与应用二进制兼容时，此方法有效。 有关确定兼容性时需要注意的更改，请参阅上文中的**后向兼容**部分。

new

使用 `new` 修饰符隐藏基类的继承成员。 这是派生类响应基类中的更新的一种方法。

请参见以下示例：

C#

```
public class BaseClass
{
    public void MyMethod()
    {
        Console.WriteLine("A base method");
    }
}

public class DerivedClass : BaseClass
{
    public new void MyMethod()
    {
        Console.WriteLine("A derived method");
    }
}

public static void Main()
{
```

```
    BaseClass b = new BaseClass();
    DerivedClass d = new DerivedClass();

    b.MyMethod();
    d.MyMethod();
}
```

输出

控制台

```
A base method
A derived method
```

上面的示例演示 `DerivedClass` 如何隐藏 `BaseClass` 中的 `MyMethod`方法。 也就是说，当新版本库中的基类添加派生类中已存在的成员时，在派生类成员上使用 `new` 修饰符即可隐藏基类成员。

未指定 `new` 修饰符时，派生类将默认隐藏基类中的冲突成员，尽管会生成编译器警告，但仍将编译代码。 也就是说，仅需向现有类添加新成员，新版本的库即可与依赖于它的代码实现源兼容和二进制兼容。

override

`override` 修饰符指派生实现会扩展基类成员的实现而不是将其隐藏。 基类成员需要具有应用于自身的 `virtual` 修饰符。

C#

```
public class MyBaseClass
{
    public virtual string MethodOne()
    {
        return "Method One";
    }
}

public class MyDerivedClass : MyBaseClass
{
    public override string MethodOne()
    {
        return "Derived Method One";
    }
}

public static void Main()
{
    MyBaseClass b = new MyBaseClass();
```

```
    MyDerivedClass d = new MyDerivedClass();

    Console.WriteLine("Base Method One: {0}", b.MethodOne());
    Console.WriteLine("Derived Method One: {0}", d.MethodOne());
}
```

输出

控制台

```
Base Method One: Method One
Derived Method One: Derived Method One
```

`override` 修饰符将在编译时计算，如果此修饰符找不到要重写的虚拟成员，编译器将引发错误。

了解所讨论的这些技术以及使用情境，对于简化库版本之间的转换有重要作用。

操作指南 (C#)

项目 • 2024/04/11

在《C# 指南》中的“操作指南”部分，你可快速了解常见问题的答案。 在某些情况下，可能会在多个部分列出相关文章。 我们希望用户可从多个搜索路径找到操作指南。

C# 一般概念

此处介绍了 C# 开发者在实践中经常会用到的几个提示和技巧：

- 使用对象初始值设定项初始化对象。
- 了解向方法传递结构和传递类的区别。
- 使用运算符重载。
- 实现和调用自定义扩展方法。
- 使用扩展方法创建新 enum 类型方法。

类、记录和结构成员

创建类、记录和结构来实现程序。 编写类、记录或结构时常会使用这些方法。

- 声明自动实现的属性。
- 声明和使用读/写属性。
- 定义常量。
- 替代 ToString 方法以提供字符串输出。
- 定义抽象属性。
- 使用 XML 文档功能记录代码。
- 显式实现接口成员，使公共接口保持简洁。
- 显式实现两个接口的成员。

使用集合

这些文章有助于了解如何使用数据集合。

- 使用集合初始值设定项初始化字典。

处理字符串

字符串是用于显示或操作文本的基本数据类型。 这些文章介绍了字符串的常见处理方法。

- 比较字符串。
- 修改字符串内容。
- 确定字符串是否表示数字。
- 使用 String.Split 分隔字符串。
- 将多个字符串合并为一个字符串。
- 在字符串中搜索文本。

在类型间转换

你可能需要将对象转换为其他类型。

- 确定字符串是否表示数字。
- 在表示十六进制数的字符串和数字之间进行转换。
- 将字符串转换为 DateTime。
- 将字节数组转换为 int。
- 将字符串转换为数字。
- 使用模式匹配、as 和 is 运算符安全强制转换为其他类型。
- 定义自定义类型转换。
- 确定类型是否为可为 null 的值类型。
- 在可为 null 和不可为 null 的值类型之间转换。

相等比较和排序比较

可创建类型来定义自己的相等规则，或者定义该类型对象间的自然顺序。

- 基于引用的相等性测试。
- 为类型定义基于值的相等。

异常处理

.NET 程序通过引发异常报告方法未能成功完成其任务。 通过这些文章可了解如何处理异常。

- 使用 try 和 catch 处理异常。
- 使用 finally 子句清理资源。
- 从非 CLS （公共语言规范）异常中恢复。

委托和事件

委托和事件为涉及松散耦合代码块的策略提供了功能。

- 声明、实例化和使用委托。
- 合并多播委托。

事件提供发布或订阅通知的机制。

- 订阅和取消订阅事件。
- 实现接口中声明的事件。
- 代码发布事件时，遵循 .NET 准则。
- 从派生类中引发在基类中定义的事件。
- 实现自定义事件访问器。

LINQ 做法

通过 LINQ 可编写代码来查询任何支持 LINQ 查询表达式模式的数据源。 这些文章有助于你理解该模式并使用不同的数据源。

- 查询集合。
- 在查询表达式中使用 var。
- 从查询返回元素属性的子集。
- 编写使用复杂筛选的查询。
- 对数据源的元素排序。
- 对多个键上的元素排序。
- 控制投影的类型。
- 对某个值在源序列中出现的次数进行计数。
- 计算中间值。
- 合并来自多个源的数据。
- 查找两个序列之间的差集。
- 调试空查询结果。
- 向 LINQ 查询添加自定义方法。

多线程和异步处理

新式程序常使用异步操作。 这些文章可帮助你了解如何使用这些方法。

- 使用 System.Threading.Tasks.Task.WhenAll 提高异步性能。
- 使用 async 和 await 并行发出多个 Web 请求。
- 使用线程池。

程序的命令行参数

通常情况下，C# 程序具有命令行参数。 通过这些文章可了解如何访问和处理这些命令行参数。

- 使用 for 检索所有命令行参数。

反馈

此页面是否有帮助? 是 否

提供产品反馈

如何在 C# 中使用 String.Split 分隔字符串

项目 • 2024/11/07

String.Split 方法通过基于一个或多个分隔符拆分输入字符串来创建子字符串数组。 此方法通常是分隔字边界上的字符串的最简单方法。 它也用于拆分其他特定字符或字符串上的字符串。

> **备注**
>
> 本文中的 C# 示例运行在 Try.NET 内联代码运行程序和演练环境中。 选择“运行”按钮以在交互窗口中运行示例。 执行代码后，可通过再次选择“运行”来修改它并运行已修改的代码。 已修改的代码要么在交互窗口中运行，要么编译失败时，交互窗口将显示所有 C# 编译器错误消息。

String.Split 示例

下方代码将一个常用短语拆分为一个由每个单词组成的字符串数组。

C#

```
string phrase = "The quick brown fox jumps over the lazy dog.";
string[] words = phrase.Split(' ');

foreach (var word in words)
{
    System.Console.WriteLine($"<{word}>");
}
```

分隔符的每个实例都会在返回的数组中产生一个值。 由于 C# 中的数组是零索引的，因此数组中的每个字符串将从 0 索引到由 Array.Length 属性返回的值减去 1：

C#

```
string phrase = "The quick brown fox jumps over the lazy dog.";
string[] words = phrase.Split(' ');

for (int i = 0; i < words.Length; i++)
{
    System.Console.WriteLine($"Index {i}: <{words[i]}>");
}
```

连续的分隔符将生成空字符串作为返回的数组中的值。 下面的示例介绍如何创建空字符串，该示例使用空格字符作为分隔符。

C#

```
string phrase = "The quick brown    fox     jumps over the lazy dog.";
string[] words = phrase.Split(' ');

foreach (var word in words)
{
    System.Console.WriteLine($"<{word}>");
}
```

该行为可以更容易地用逗号分隔值 (CSV) 文件之类的格式表示表格数据。 连续的逗号表示空白列。

可传递可选 StringSplitOptions.RemoveEmptyEntries 参数来排除返回数组中的任何空字符串。 要对返回的集合进行更复杂的处理，可使用 LINQ 来处理结果序列。

String.Split 可使用多个分隔符。 下面的示例使用空格、逗号、句点、冒号和制表符作为分隔字符，这些分隔字符在数组中传递到 Split。 代码底部的循环显示返回数组中的每个单词。

C#

```
char[] delimiterChars = { ' ', ',', '.', ':', '\t' };

string text = "one\ttwo three:four,five six seven";
System.Console.WriteLine($"Original text: '{text}'");

string[] words = text.Split(delimiterChars);
System.Console.WriteLine($"{words.Length} words in text:");

foreach (var word in words)
{
    System.Console.WriteLine($"<{word}>");
}
```

任何分隔符的连续实例都会在输出数组中生成空字符串：

C#

```
char[] delimiterChars = { ' ', ',', '.', ':', '\t' };

string text = "one\ttwo :,five six seven";
System.Console.WriteLine($"Original text: '{text}'");

string[] words = text.Split(delimiterChars);
System.Console.WriteLine($"{words.Length} words in text:");

foreach (var word in words)
{
```

```
    System.Console.WriteLine($"<{word}>");
}
```

String.Split 可采用字符串数组（充当用于分析目标字符串的分隔符的字符序列，而非单个字符）。

C#

```
string[] separatingStrings = { "<<", "..." };

string text = "one<<two......three<four";
System.Console.WriteLine($"Original text: '{text}'");

string[] words = text.Split(separatingStrings,
System.StringSplitOptions.RemoveEmptyEntries);
System.Console.WriteLine($"{words.Length} substrings in text:");

foreach (var word in words)
{
    System.Console.WriteLine(word);
}
```

使用 GitHub Copilot 拆分字符串

可以在 IDE 中使用 GitHub Copilot 生成代码，以使用 `String.Split` C# 拆分字符串。

如果使用 Visual Studio 2022 版本 17.8 或更高版本，则可以尝试在 Visual Studio 中使用 AI 驱动的 GitHub Copilot 生成代码，以基于一个或多个分隔符将输入字符串拆分为子字符串。 在 Copilot Chat 窗口中以提示形式提交问题，如下例所示。 还可以在编辑器窗口中使用内联聊天提交提示。

> **备注**
>
> GitHub Copilot 由 AI 提供支持，因此可能会带来意外和错误。 请确保验证任何生成的代码或建议。 有关 GitHub Copilot 的常规用途、产品影响、人工监督和隐私的更多信息，请参阅 GitHub Copilot 常见问题解答。

以下文本显示了 Copilot 聊天的示例提示：

> 生成 C# 代码以使用 Split.String 将字符串拆分为子字符串。 输入字符串是“你赢得一些。 你失去了一些。分隔符是空格和句点。 提供示例输出。

可以根据要求自定义提示以使用字符串和分隔符。

可以使用聊天功能（如斜杠命令、引用文件、方法或类和线程）来设置意向，并通过已限定范围上下文获取更好的答案。 对于在 IDE 中打开的现有文件，可以使用内联聊天 `/generate code to split string1 in #Filename by delimiter spaces` 提示 GitHub Copilot。

以下输出显示了 Copilot 聊天响应示例：

输出

```
Here is an example of how you can use the Split method in C# to split the
string "You win some. You lose some." using space and period as delimiters:

using System;

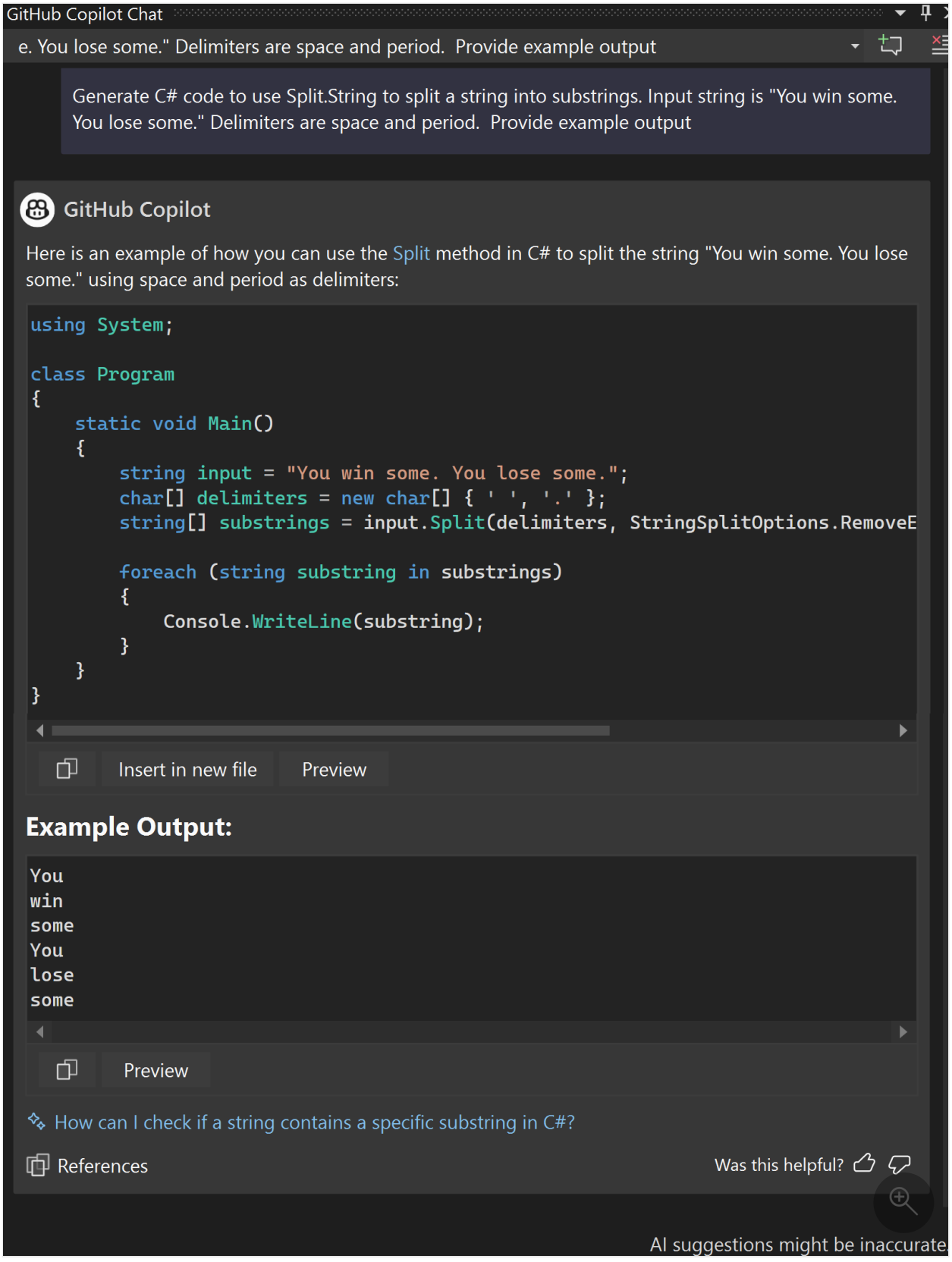
class Program
{
    static void Main()
    {
        string input = "You win some. You lose some.";
        char[] delimiters = new char[] { ' ', '.' };
        string[] substrings = input.Split(delimiters,
StringSplitOptions.RemoveEmptyEntries);

        foreach (string substring in substrings)
        {
            Console.WriteLine(substring);
        }
    }
}

**Example Output**
You
win
some
You
lose
some
```

当 Copilot 返回代码块时，响应包括用于复制代码、将代码插入新文件或预览代码输出的选项。

> **① 备注**
>
> 结果可能与示例响应中显示的结果不同。 AI 模型不确定，这意味着当被问及相同的问题时，它们可以返回不同的响应。 这可能是由于一段时间内进行的额外学习和调整、语言变体、上下文中的更改（如聊天历史记录等）。



有关详细信息，请参阅：

- GitHub Copilot 信任中心
- Visual Studio 中的 GitHub Copilot
- VS Code 中的 GitHub Copilot

另请参阅

- 从字符串中提取元素
- 字符串
- .NET 正则表达式

如何连接多个字符串（C# 指南）

项目 • 2024/12/28

串联 是将一个字符串追加到另一个字符串末尾的过程。 使用 `+` 运算符连接字符串。 对于字符串文本和字符串常量，串联在编译时发生;不会发生运行时串联。 对于字符串变量，串联仅在运行时发生。

> **备注**
>
> 本文中的 C# 示例运行在 Try.NET 内联代码运行程序和演练环境中。 选择 **运行** 按钮，在交互式窗口中运行示例。 执行代码后，可以通过再次选择 **运行** 来修改代码并运行修改的代码。 修改的代码要么在交互式窗口中运行，要么如果编译失败，交互式窗口将显示所有 C# 编译器错误消息。

字符串文本

以下示例将长字符串文本拆分为较小的字符串，以提高源代码中的可读性。 以下代码将较短的字符串连接起来，以创建长字符串字面量。 这些部分在编译时串联成单个字符串。 无论涉及到多少个字符串，均不产生运行时性能开销。

C#

```
// Concatenation of literals is performed at compile time, not run time.
string text = "Historically, the world of data and the world of objects " +
"have not been well integrated. Programmers work in C# or Visual Basic " +
"and also in SQL or XQuery. On the one side are concepts such as classes, "
+
"objects, fields, inheritance, and .NET Framework APIs. On the other side "
+
"are tables, columns, rows, nodes, and separate languages for dealing with "
+
"them. Data types often require translation between the two worlds; there
are " +
"different standard functions. Because the object world has no notion of
query, a " +
"query can only be represented as a string without compile-time type
checking or " +
"IntelliSense support in the IDE. Transferring data from SQL tables or XML
trees to " +
"objects in memory is often tedious and error-prone.";

System.Console.WriteLine(text);
```

`+` 和 `+=` 运算符

若要连接字符串变量，可以使用 `+` 或 `+=` 运算符、字符串内插 或 String.Format、String.Concat、String.Join 或 StringBuilder.Append 方法。 `+` 运算符易于使用，适用于直观的代码。 即使在一个语句中使用多个 `+` 运算符，字符串内容也只复制一次。 以下代码示例演示如何使用 `+` 和 `+=` 运算符连接字符串：

C#

```
string userName = "<Type your name here>";
string dateString = DateTime.Today.ToShortDateString();

// Use the + and += operators for one-time concatenations.
string str = "Hello " + userName + ". Today is " + dateString + ".";
System.Console.WriteLine(str);

str += " How are you today?";
System.Console.WriteLine(str);
```

字符串内插

在某些表达式中，使用字符串内插进行字符串串联更简单，如以下代码所示：

C#

```
string userName = "<Type your name here>";
string date = DateTime.Today.ToShortDateString();

// Use string interpolation to concatenate strings.
string str = $"Hello {userName}. Today is {date}.";
System.Console.WriteLine(str);

str = $"{str} How are you today?";
System.Console.WriteLine(str);
```

> ⓘ **备注**
>
> 在字符串串联操作中，C# 编译器将 null 字符串视为与空字符串相同。

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。

`String.Format`

连接字符串的另一种方法是 String.Format。 从少量组件字符串生成字符串时，此方法非常有效。

StringBuilder

在其他情况下，你可能会在不知道要组合的源字符串数的循环中组合字符串，而源字符串的实际数量可能很大。 StringBuilder 类是为这些方案设计的。 以下代码使用 StringBuilder 类的 Append 方法连接字符串。

C#

```
// Use StringBuilder for concatenation in tight loops.
var sb = new System.Text.StringBuilder();
for (int i = 0; i < 20; i++)
{
    sb.AppendLine(i.ToString());
}
System.Console.WriteLine(sb.ToString());
```

有关详细信息，请阅读选择字符串串联或 StringBuilder 类的原因。

String.Concat 或 String.Join

从集合联接字符串的另一个选项是使用 String.Concat 方法。 如果分隔符应分隔源字符串，请使用 String.Join 方法。 以下代码使用这两种方法合并单词数组：

C#

```
string[] words = { "The", "quick", "brown", "fox", "jumps", "over", "the",
"lazy", "dog." };

var unreadablePhrase = string.Concat(words);
System.Console.WriteLine(unreadablePhrase);

var readablePhrase = string.Join(" ", words);
System.Console.WriteLine(readablePhrase);
```

LINQ 和 Enumerable.Aggregate

最后，可以使用 LINQ 和 Enumerable.Aggregate 方法联接集合中的字符串。 此方法使用 lambda 表达式合并源字符串。 lambda 表达式负责将每个字符串添加到已存在的累积量。 以下示例合并单词数组，在数组中的每个单词之间添加一个空格：

C#

```
string[] words = { "The", "quick", "brown", "fox", "jumps", "over", "the",
"lazy", "dog." };

var phrase = words.Aggregate((partialPhrase, word) =>$"{partialPhrase}
{word}");
System.Console.WriteLine(phrase);
```

此选项可能会比其他连接集合的方法导致更多内存资源的分配，因为它在每次迭代时都会创建一个中间字符串。 如果优化性能至关重要，请考虑 StringBuilder 类或 String.Concat 或 String.Join 方法来连接集合，而不是 `Enumerable.Aggregate`。

另请参阅

- String
- StringBuilder
- 字符串

如何搜索字符串

项目 • 2024/03/13

可以使用两种主要策略搜索字符串中的文本。 String 类的方法搜索特定文本。 正则表达式搜索文本中的模式。

> **备注**
>
> 本文中的 C# 示例运行在 Try.NET 内联代码运行程序和演练环境中。 选择“运行”按钮以在交互窗口中运行示例。 执行代码后，可通过再次选择“运行”来修改它并运行已修改的代码。 已修改的代码要么在交互窗口中运行，要么编译失败时，交互窗口将显示所有 C# 编译器错误消息。

string 类型是 System.String 类的别名，可提供多种有效方法用于搜索字符串的内容。 其中包括 Contains、StartsWith、EndsWith、IndexOf 以及 LastIndexOf。 System.Text.RegularExpressions.Regex 类具备丰富的词汇来对文本中的模式进行搜索。 你将在本文中了解这些技术以及如何选择符合需求的最佳方法。

字符串包含文本吗?

String.Contains、String.StartsWith 和 String.EndsWith 方法搜索字符串中的特定文本。 下面的示例显示了每一个方法以及使用不区分大小写的搜索的差异：

C#

```
string factMessage = "Extension methods have all the capabilities of regular
static methods.";

// Write the string and include the quotation marks.
Console.WriteLine($"\"{factMessage}\"");

// Simple comparisons are always case sensitive!
bool containsSearchResult = factMessage.Contains("extension");
Console.WriteLine($"Contains \"extension\"? {containsSearchResult}");

// For user input and strings that will be displayed to the end user,
// use the StringComparison parameter on methods that have it to specify how
to match strings.
bool ignoreCaseSearchResult = factMessage.StartsWith("extension",
System.StringComparison.CurrentCultureIgnoreCase);
Console.WriteLine($"Starts with \"extension\"? {ignoreCaseSearchResult}
(ignoring case)");

bool endsWithSearchResult = factMessage.EndsWith(".",
```

```
System.StringComparison.CurrentCultureIgnoreCase);
Console.WriteLine($"Ends with '.'? {endsWithSearchResult}");
```

前面的示例演示了使用这些方法的重点。 默认情况下搜索是区分大小写的。 使用 StringComparison.CurrentCultureIgnoreCase 枚举值指定区分大小写的搜索。

寻找的文本出现在字符串的什么位置?

IndexOf 和 LastIndexOf 方法也搜索字符串中的文本。 这些方法返回查找到的文本的位置。 如果未找到文本，则返回 `-1`。 下面的示例显示“methods”第一次出现和最后一次出现的搜索结果，并显示了它们之间的文本。

C#

```
string factMessage = "Extension methods have all the capabilities of regular
static methods.";

// Write the string and include the quotation marks.
Console.WriteLine($"\"{factMessage}\"");

// This search returns the substring between two strings, so
// the first index is moved to the character just after the first string.
int first = factMessage.IndexOf("methods") + "methods".Length;
int last = factMessage.LastIndexOf("methods");
string str2 = factMessage.Substring(first, last - first);
Console.WriteLine($"Substring between \"methods\" and \"methods\":
'{str2}'");
```

使用正则表达式查找特定文本

System.Text.RegularExpressions.Regex 类可用于搜索字符串。 这些搜索的范围可以从简单的内容到复杂的文本模式。

下面的代码示例在一个句子中搜索了“the”或“their”（忽略大小写）。 静态方法 Regex.IsMatch 执行此次搜索。 你向它提供要搜索的字符串以及搜索模式。 在这种情况下，第三个参数指定不区分大小写的搜索。 有关详细信息，请参阅 System.Text.RegularExpressions.RegexOptions。

搜索模式描述你所搜索的文本。 下表描述搜索模式的每个元素。 （下表使用单个 `\`，它在 C# 字符串中必须转义为 `\\`）。

展开表

模式	含义
`the`	匹配文本“the”
`(eir)?`	匹配 0 个或 1 个“eir”
`\s`	与空白符匹配

C#

```
string[] sentences =
{
    "Put the water over there.",
    "They're quite thirsty.",
    "Their water bottles broke."
};

string sPattern = "the(ir)?\\s";

foreach (string s in sentences)
{
    Console.Write($"{s,24}");

    if (System.Text.RegularExpressions.Regex.IsMatch(s, sPattern,
System.Text.RegularExpressions.RegexOptions.IgnoreCase))
    {
        Console.WriteLine($"  (match for '{sPattern}' found)");
    }
    else
    {
        Console.WriteLine();
    }
}
```

提示

搜索精确的字符串时，`string` 方法通常是更好的选择。 搜索源字符串中的一些模式时，正则表达式更适用。

字符串是否遵循模式?

以下代码使用正则表达式验证数组中每个字符串的格式。 验证要求每个字符串具备电话号码的形式：用短划线分隔成三组数字，前两组包含 3 个数字，而第三组包含 4 个数字。 搜索模式采用正则表达式 `^\\d{3}-\\d{3}-\\d{4}$`。 有关更多信息，请参见正则表达式语言 - 快速参考。

展开表

模式	含义
`^`	匹配字符串的开头部分
`\d{3}`	完全匹配 3 位字符
`-`	匹配字符“-”
`\d{4}`	完全匹配 4 位字符
`$`	匹配字符串的结尾部分

C#

```
string[] numbers =
{
    "123-555-0190",
    "444-234-22450",
    "690-555-0178",
    "146-893-232",
    "146-555-0122",
    "4007-555-0111",
    "407-555-0111",
    "407-2-5555",
    "407-555-8974",
    "407-2ab-5555",
    "690-555-8148",
    "146-893-232-"
};

string sPattern = "^\\d{3}-\\d{3}-\\d{4}$";

foreach (string s in numbers)
{
    Console.Write($"{s,14}");

    if (System.Text.RegularExpressions.Regex.IsMatch(s, sPattern))
    {
        Console.WriteLine(" - valid");
    }
    else
    {
        Console.WriteLine(" - invalid");
    }
}
```

此单个搜索模式匹配很多有效字符串。 正则表达式更适用于搜索或验证模式，而不是单个文本字符串。

另请参阅

- 字符串
- System.Text.RegularExpressions.Regex
- .NET 正则表达式
- 正则表达式语言 - 快速参考
- 有关使用 .NET 中字符串的最佳做法

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 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

如何修改用 C# 编写的字符串内容

项目 • 2023/04/08

本文演示通过修改现有 `string` 来生成 `string` 的几种方法。 演示的所有方法均将修改的结果返回为新的 `string` 对象。 为了说明原始字符串和修改后的字符串是不同的实例，示例会将结果存储在新变量中。 运行每个示例时，可以检查原始 `string` 和修改后的新 `string`。

> **备注**
>
> 本文中的 C# 示例运行在 Try.NET 内联代码运行程序和演练环境中。 选择“运行”按钮以在交互窗口中运行示例。 执行代码后，可通过再次选择“运行”来修改它并运行已修改的代码。 已修改的代码要么在交互窗口中运行，要么编译失败时，交互窗口将显示所有 C# 编译器错误消息。

本文中演示了几种方法。 你可以替换现有文本。 可以搜索模式并将匹配的文本替换为其他文本。 可以将字符串视为字符序列。 还可以使用删除空格的简便方法。 选择与你的方案最匹配的方法。

替换文本

下面的代码通过将现有文本替换为替代文本来创建新的字符串。

C#

```
string source = "The mountains are behind the clouds today.";

// Replace one substring with another with String.Replace.
// Only exact matches are supported.
var replacement = source.Replace("mountains", "peaks");
Console.WriteLine($"The source string is <{source}>");
Console.WriteLine($"The updated string is <{replacement}>");
```

上述代码演示了字符串的不可变属性。 在上述示例中可以看到，原始字符串 `source` 并未被修改。 String.Replace 方法创建的是包含修改的新 `string`。

Replace 可替换字符串或单个字符。 在这两种情况下，搜索文本的每个匹配项均被替换。 下面的示例将所有的“ ”都替换为“_”：

C#

```
string source = "The mountains are behind the clouds today.";

// Replace all occurrences of one char with another.
var replacement = source.Replace(' ', '_');
Console.WriteLine(source);
Console.WriteLine(replacement);
```

源字符串并未发生更改，而是通过替换操作返回了一个新的字符串。

去除空格

可使用 String.Trim、String.TrimStart 和 String.TrimEnd 方法删除任何前导空格或尾随空格。 下面的代码就是删除两种空格的示例。 源字符串不会发生变化；这些方法返回带修改内容的新字符串。

C#

```
// Remove trailing and leading white space.
string source = "    I'm wider than I need to be.      ";
// Store the results in a new string variable.
var trimmedResult = source.Trim();
var trimLeading = source.TrimStart();
var trimTrailing = source.TrimEnd();
Console.WriteLine($"<{source}>");
Console.WriteLine($"<{trimmedResult}>");
Console.WriteLine($"<{trimLeading}>");
Console.WriteLine($"<{trimTrailing}>");
```

删除文本

可使用 String.Remove 方法删除字符串中的文本。 此方法移删除特定索引处开始的某些字符。 下面的示例演示如何使用 String.IndexOf（后接 Remove）方法，删除字符串中的文本：

C#

```
string source = "Many mountains are behind many clouds today.";
// Remove a substring from the middle of the string.
string toRemove = "many ";
string result = string.Empty;
int i = source.IndexOf(toRemove);
if (i >= 0)
{
    result= source.Remove(i, toRemove.Length);
}
```

```
Console.WriteLine(source);
Console.WriteLine(result);
```

替换匹配模式

可使用正则表达式将匹配模式的文本替换为新文本，新文本可能由模式定义。 下面的示例使用 System.Text.RegularExpressions.Regex 类从源字符串中查找模式并将其替换为正确的大写。 Regex.Replace(String, String, MatchEvaluator, RegexOptions) 方法使用将替换逻辑提供为其参数之一的函数。 在本示例中，该函数 `LocalReplaceMatchCase` 是在示例方法中声明的本地函数。 `LocalReplaceMatchCase` 使用 System.Text.StringBuilder类，以生成具有正确大写的替换字符串。

正则表达式最适合用于搜索和替换遵循模式的文本，而不是已知的文本。 有关详细信息，请参阅如何搜索字符串。 搜索模式“the\s”搜索“the”后接空格字符的单词。 该部分的模式可确保它不与源字符串中的“there”相匹配。 有关正则表达式语言元素的更多信息，请参阅正则表达式语言 - 快速参考。

C#

```
string source = "The mountains are still there behind the clouds today.";

// Use Regex.Replace for more flexibility.
// Replace "the" or "The" with "many" or "Many".
// using System.Text.RegularExpressions
string replaceWith = "many ";
source = System.Text.RegularExpressions.Regex.Replace(source, "the\\s",
LocalReplaceMatchCase,
    System.Text.RegularExpressions.RegexOptions.IgnoreCase);
Console.WriteLine(source);

string LocalReplaceMatchCase(System.Text.RegularExpressions.Match
matchExpression)
{
    // Test whether the match is capitalized
    if (Char.IsUpper(matchExpression.Value[0]))
    {
        // Capitalize the replacement string
        System.Text.StringBuilder replacementBuilder = new
System.Text.StringBuilder(replaceWith);
        replacementBuilder[0] = Char.ToUpper(replacementBuilder[0]);
        return replacementBuilder.ToString();
    }
    else
    {
        return replaceWith;
    }
}
```

StringBuilder.ToString 方法返回一个不可变的字符串，其中包含 StringBuilder 对象中的内容。

修改单个字符

可从字符串生成字符数组，修改数组的内容，然后从数组的已修改内容创建新的字符串。

下面的示例演示如何替换字符串中的一组字符。 首先，它使用 String.ToCharArray() 方法来创建字符数组。 它使用 IndexOf 方法来查找单词“fox”的起始索引。接下来的三个字符将替换为其他单词。 最终，从更新的字符串数组中构造了新的字符串。

C#

```
string phrase = "The quick brown fox jumps over the fence";
Console.WriteLine(phrase);

char[] phraseAsChars = phrase.ToCharArray();
int animalIndex = phrase.IndexOf("fox");
if (animalIndex != -1)
{
    phraseAsChars[animalIndex++] = 'c';
    phraseAsChars[animalIndex++] = 'a';
    phraseAsChars[animalIndex] = 't';
}

string updatedPhrase = new string(phraseAsChars);
Console.WriteLine(updatedPhrase);
```

以编程方式生成字符串内容

由于字符串是不可变的，因此前面的示例都创建了临时字符串或字符数组。 在高性能方案中，可能需要避免这些堆分配。 .NET Core 提供了一种 String.Create 方法，该方法使你可以通过回调以编程方式填充字符串的字符内容，同时避免中间的临时字符串分配。

C#

```
// constructing a string from a char array, prefix it with some additional
characters
char[] chars = { 'a', 'b', 'c', 'd', '\0' };
int length = chars.Length + 2;
string result = string.Create(length, chars, (Span<char> strContent, char[]
charArray) =>
{
    strContent[0] = '0';
    strContent[1] = '1';
    for (int i = 0; i < charArray.Length; i++)
    {
```

```
        strContent[i + 2] = charArray[i];
    }
});

Console.WriteLine(result);
```

可以使用不安全的代码修改固定块中的字符串，但是强烈建议不要在创建字符串后修改字符串内容。 这样做将以不可预知的方式中断操作。 例如，如果某人暂存一个与你的内容相同的字符串，他们将获得你的副本，并且不希望你修改他们的字符串。

请参阅

- .NET 正则表达式
- 正则表达式语言 - 快速参考

如何：比较 C# 中的字符串

项目 • 2024/03/21

通过比较字符串可以回答两个问题，一个是：“这两个字符串相等吗？”另一个是“排序时，应该按什么顺序排列这些字符串？”

这两个问题非常复杂，因为字符串比较受很多因素的影响：

- 可以选择序号比较或语义比较。
- 可以选择是否区分大小写。
- 可以选择区域性特定的比较。
- 语义比较取决于区域性和平台。

System.StringComparison 枚举字段代表以下选项：

- CurrentCulture：使用区分区域性的排序规则和当前区域性比较字符串。
- CurrentCultureIgnoreCase：通过使用区分区域性的排序规则、当前区域性，并忽略所比较的字符串的大小写，来比较字符串。
- InvariantCulture：使用区分区域性的排序规则和固定区域性比较字符串。
- InvariantCultureIgnoreCase：通过使用区分区域性的排序规则、固定区域性，并忽略所比较的字符串的大小写，来比较字符串。
- Ordinal：使用序号（二进制）排序规则比较字符串。
- OrdinalIgnoreCase：通过使用序号（二进制）区分区域性的排序规则并忽略所比较的字符串的大小写，来比较字符串。

> **备注**
>
> 本文中的 C# 示例运行在 Try.NET 内联代码运行程序和演练环境中。 选择“运行”按钮以在交互窗口中运行示例。 执行代码后，可通过再次选择“运行”来修改它并运行已修改的代码。 已修改的代码要么在交互窗口中运行，要么编译失败时，交互窗口将显示所有 C# 编译器错误消息。

在比较字符串时定义它们的顺序。 通过比较决定字符串的序列。 确定序列顺序后，软件和人工都可以更轻松地进行搜索。 其他比较可能会检查字符串是否相同。 这种一致性检查与相等性检查类似，但是也有一些不同之处，例如可能会忽略大小写的差异。

默认的序号比较

默认情况下，最常见的操作：

- String.Equals

- String.Equality 和 String.Inequality，即相等运算符 == 和 !=，分别执行区分大小写的序号比较。 String.Equals 具有重载，可以提供 StringComparison 参数来更改其排序规则。 下面的示例演示了这一操作：

C#

```
string root = @"C:\users";
string root2 = @"C:\Users";

bool result = root.Equals(root2);
Console.WriteLine($"Ordinal comparison: <{root}> and <{root2}> are {(result
? "equal." : "not equal.")}");

result = root.Equals(root2, StringComparison.Ordinal);
Console.WriteLine($"Ordinal comparison: <{root}> and <{root2}> are {(result
? "equal." : "not equal.")}");

Console.WriteLine($"Using == says that <{root}> and <{root2}> are {(root ==
root2 ? "equal" : "not equal")}");
```

默认序号比较在比较字符串时不会考虑语义规则。 它会比较两个字符串中每个 Char 对象的二进制值。 因此，默认的序号比较也是区分大小写的。

使用 String.Equals 以及 `==` 和 `!=` 运算符的相等性测试不同于使用 String.CompareTo 和 Compare(String, String) 方法的字符串比较。 它们均执行区分大小写的比较。 但是，虽然相等性测试执行顺序比较，但 `CompareTo` 和 `Compare` 方法使用当前区域性执行区域性感知的语言比较。 调用显式指定要执行的比较类型的重载，确保代码意图明确。

不区分大小写的序号比较

采用 String.Equals(String, StringComparison) 方法可指定 StringComparison.OrdinalIgnoreCase 的 StringComparison 值来进行不区分大小写的序号比较。 此外还有静态 String.Compare(String, String, StringComparison) 方法，该方法执行不区分大小写的序号比较，前提是你指定 StringComparison 参数的 StringComparison.OrdinalIgnoreCase 值。 在下面的代码中演示了这些比较：

C#

```
string root = @"C:\users";
string root2 = @"C:\Users";

bool result = root.Equals(root2, StringComparison.OrdinalIgnoreCase);
bool areEqual = String.Equals(root, root2,
StringComparison.OrdinalIgnoreCase);
int comparison = String.Compare(root, root2, comparisonType:
StringComparison.OrdinalIgnoreCase);
```

```
Console.WriteLine($"Ordinal ignore case: <{root}> and <{root2}> are {(result
? "equal." : "not equal.")}");
Console.WriteLine($"Ordinal static ignore case: <{root}> and <{root2}> are
{(areEqual ? "equal." : "not equal.")}");
if (comparison < 0)
    Console.WriteLine($"<{root}> is less than <{root2}>");
else if (comparison > 0)
    Console.WriteLine($"<{root}> is greater than <{root2}>");
else
    Console.WriteLine($"<{root}> and <{root2}> are equivalent in order");
```

执行不区分大小写的序号比较时，这些方法使用固定区域性的大小写约定。

语义比较

许多字符串比较方法（例如 String.StartsWith）默认使用当前区域性的语言规则对其输入进行排序。 这种语言比较有时被称为“文字排序顺序”。在执行语义比较时，一些非字母数字的 Unicode 字符可能分配有特殊的权重。 例如，连字符“-”分配的权重可能很小，所以“co-op”和“coop”在排序顺序中会彼此相邻。 一些非打印控制字符可能会被忽略。 此外，一些 Unicode 字符可能与一系列 Char 实例等效。 下面以德语短句“他们在街上跳舞。”为例；其中，一个字符串中有“ss”(U+0073 U+0073)，另一个字符串中有“ß”(U+00DF)。 （在 Windows 系统中）从语义上说，“ss”在“en-US”和“de-DE”区域性中都等同于德语中的“'ß”。

C#

```
string first = "Sie tanzen auf der Straße.";
string second = "Sie tanzen auf der Strasse.";

Console.WriteLine($"First sentence is <{first}>");
Console.WriteLine($"Second sentence is <{second}>");

bool equal = String.Equals(first, second,
StringComparison.InvariantCulture);
Console.WriteLine($"The two strings {(equal == true ? "are" : "are not")}
equal.");
showComparison(first, second);

string word = "coop";
string words = "co-op";
string other = "cop";

showComparison(word, words);
showComparison(word, other);
showComparison(words, other);
void showComparison(string one, string two)
{
    int compareLinguistic = String.Compare(one, two,
StringComparison.InvariantCulture);
```

```
    int compareOrdinal = String.Compare(one, two, StringComparison.Ordinal);
    if (compareLinguistic < 0)
        Console.WriteLine($"<{one}> is less than <{two}> using invariant
culture");
    else if (compareLinguistic > 0)
        Console.WriteLine($"<{one}> is greater than <{two}> using invariant
culture");
    else
        Console.WriteLine($"<{one}> and <{two}> are equivalent in order
using invariant culture");
    if (compareOrdinal < 0)
        Console.WriteLine($"<{one}> is less than <{two}> using ordinal
comparison");
    else if (compareOrdinal > 0)
        Console.WriteLine($"<{one}> is greater than <{two}> using ordinal
comparison");
    else
        Console.WriteLine($"<{one}> and <{two}> are equivalent in order
using ordinal comparison");
}
```

在 Windows 上，.NET 5 之前，从语义比较改为序号比较时，“cop”、“coop”和“co-op”的排序顺序产生了变化。 使用不同的比较类型时，这两个德语句子的比较结果也就不同了。 在 .NET 5 之前，.NET 全球化 API 使用国家语言支持 (NLS) 库。 在 .NET 5 和更高版本中，.NET 全球化 API 使用 International Components for Unicode (ICU) 库，该库统一所有支持的操作系统中的 .NET 全球化行为。

使用特定区域性的比较

以下示例为 en-US 和 de-DE 区域性存储 CultureInfo 对象。 使用 CultureInfo 对象执行比较以确保执行的是特定于区域性的比较。 所用的区域性会对语义比较产生影响。 以下示例展示使用“en-US”区域性和“de-DE”区域性对两个德语句子进行比较的结果：

C#

```
string first = "Sie tanzen auf der Straße.";
string second = "Sie tanzen auf der Strasse.";

Console.WriteLine($"First sentence is <{first}>");
Console.WriteLine($"Second sentence is <{second}>");

var en = new System.Globalization.CultureInfo("en-US");

// For culture-sensitive comparisons, use the String.Compare
// overload that takes a StringComparison value.
int i = String.Compare(first, second, en,
System.Globalization.CompareOptions.None);
Console.WriteLine($"Comparing in {en.Name} returns {i}.");
```

```
var de = new System.Globalization.CultureInfo("de-DE");
i = String.Compare(first, second, de,
System.Globalization.CompareOptions.None);
Console.WriteLine($"Comparing in {de.Name} returns {i}.");

bool b = String.Equals(first, second, StringComparison.CurrentCulture);
Console.WriteLine($"The two strings {(b ? "are" : "are not")} equal.");

string word = "coop";
string words = "co-op";
string other = "cop";

showComparison(word, words, en);
showComparison(word, other, en);
showComparison(words, other, en);
void showComparison(string one, string two, System.Globalization.CultureInfo
culture)
{
    int compareLinguistic = String.Compare(one, two, en,
System.Globalization.CompareOptions.None);
    int compareOrdinal = String.Compare(one, two, StringComparison.Ordinal);
    if (compareLinguistic < 0)
        Console.WriteLine($"<{one}> is less than <{two}> using en-US
culture");
    else if (compareLinguistic > 0)
        Console.WriteLine($"<{one}> is greater than <{two}> using en-US
culture");
    else
        Console.WriteLine($"<{one}> and <{two}> are equivalent in order
using en-US culture");
    if (compareOrdinal < 0)
        Console.WriteLine($"<{one}> is less than <{two}> using ordinal
comparison");
    else if (compareOrdinal > 0)
        Console.WriteLine($"<{one}> is greater than <{two}> using ordinal
comparison");
    else
        Console.WriteLine($"<{one}> and <{two}> are equivalent in order
using ordinal comparison");
}
```

区分区域性的比较通常用于对用户输入的字符串以及用户输入的其他字符串进行比较和排序。 字符和这些字符的排序约定可能会根据用户计算机的区域设置而有所不同。 即使是包含相同字符的字符串，也可能因当前线程的区域性而具有不同的排序。

数组中的语义排序和字符串搜索

以下示例演示如何在数组中使用依赖当前区域性的语义比较对字符串进行排序和搜索。 使用采用 System.StringComparer 参数的静态 Array 方法。

以下示例演示如何使用当前区域性对字符串数组进行排序：

C#

```
string[] lines = new string[]
{
    @"c:\public\textfile.txt",
    @"c:\public\textFile.TXT",
    @"c:\public\Text.txt",
    @"c:\public\testfile2.txt"
};

Console.WriteLine("Non-sorted order:");
foreach (string s in lines)
{
    Console.WriteLine($"   {s}");
}

Console.WriteLine("\n\rSorted order:");

// Specify Ordinal to demonstrate the different behavior.
Array.Sort(lines, StringComparer.CurrentCulture);

foreach (string s in lines)
{
    Console.WriteLine($"   {s}");
}
```

对数组进行排序后，可以使用二分搜索法搜索条目。 二分搜索法从集合的中间开始搜索，判断集合的哪一半包含所找字符串。 后续的每个比较都将集合的剩余部分再次对半分开。 使用 StringComparer.CurrentCulture 存储数组。 本地函数 `ShowWhere` 显示发现字符串所在位置的信息。 如果未找到字符串，返回的值会指示可以在其中找到字符串的位置。

C#

```
string[] lines = new string[]
{
    @"c:\public\textfile.txt",
    @"c:\public\textFile.TXT",
    @"c:\public\Text.txt",
    @"c:\public\testfile2.txt"
};
Array.Sort(lines, StringComparer.CurrentCulture);

string searchString = @"c:\public\TEXTFILE.TXT";
Console.WriteLine($"Binary search for <{searchString}>");
int result = Array.BinarySearch(lines, searchString,
StringComparer.CurrentCulture);
ShowWhere<string>(lines, result);

Console.WriteLine($"{(result > 0 ? "Found" : "Did not find")}
{searchString}");
```

```
void ShowWhere<T>(T[] array, int index)
{
    if (index < 0)
    {
        index = ~index;

        Console.Write("Not found. Sorts between: ");

        if (index == 0)
            Console.Write("beginning of sequence and ");
        else
            Console.Write($"{array[index - 1]} and ");

        if (index == array.Length)
            Console.WriteLine("end of sequence.");
        else
            Console.WriteLine($"{array[index]}.");
    }
    else
    {
        Console.WriteLine($"Found at index {index}.");
    }
}
```

集合中的序号排序和搜索

以下代码使用 System.Collections.Generic.List<T> 集合类存储字符串。 字符串是通过 List<T>.Sort 方法排序的。 此方法需要对两个字符串进行比较和排序的委托。 String.CompareTo 方法提供该比较函数。 请运行示例并观察顺序。 此排序操作使用区分大小写的序号排序。 你要使用静态 String.Compare 方法指定不同的比较规则。

C#

```
List<string> lines = new List<string>
{
    @"c:\public\textfile.txt",
    @"c:\public\textFile.TXT",
    @"c:\public\Text.txt",
    @"c:\public\testfile2.txt"
};

Console.WriteLine("Non-sorted order:");
foreach (string s in lines)
{
    Console.WriteLine($"   {s}");
}

Console.WriteLine("\n\rSorted order:");
```

```
lines.Sort((left, right) => left.CompareTo(right));
foreach (string s in lines)
{
    Console.WriteLine($"   {s}");
}
```

排序后，可以使用二分搜索法对字符串列表进行搜索。 以下示例演示如何使用相同的比较函数搜索排序列表。 本地函数 `ShowWhere` 显示所查找的文本所在的位置或可能会在位置：

C#

```
List<string> lines = new List<string>
{
    @"c:\public\textfile.txt",
    @"c:\public\textFile.TXT",
    @"c:\public\Text.txt",
    @"c:\public\testfile2.txt"
};
lines.Sort((left, right) => left.CompareTo(right));

string searchString = @"c:\public\TEXTFILE.TXT";
Console.WriteLine($"Binary search for <{searchString}>");
int result = lines.BinarySearch(searchString);
ShowWhere<string>(lines, result);

Console.WriteLine($"{(result > 0 ? "Found" : "Did not find")}
{searchString}");

void ShowWhere<T>(IList<T> collection, int index)
{
    if (index < 0)
    {
        index = ~index;

        Console.Write("Not found. Sorts between: ");

        if (index == 0)
            Console.Write("beginning of sequence and ");
        else
            Console.Write($"{collection[index - 1]} and ");

        if (index == collection.Count)
            Console.WriteLine("end of sequence.");
        else
            Console.WriteLine($"{collection[index]}.");
    }
    else
    {
        Console.WriteLine($"Found at index {index}.");
```

```
    }
}
```

在排序和搜索过程中，请始终确保使用相同的比较类型。 使用不同的比较类型进行排序和搜索会产生意外的结果。

元素或键的类型为 `string` 时，System.Collections.Hashtable、System.Collections.Generic.Dictionary<TKey,TValue> 和 System.Collections.Generic.List<T> 等集合类的构造函数具有 System.StringComparer 参数。 通常，应尽可能使用这些构造函数，并指定 StringComparer.Ordinal 或 StringComparer.OrdinalIgnoreCase。

请参阅

- System.Globalization.CultureInfo
- System.StringComparer
- 字符串
- 比较字符串
- 对应用程序进行全球化和本地化

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

如何捕捉非 CLS 异常

项目 • 2023/05/10

包括 C++/CLI 在内的某些 .NET 语言允许对象引发并非派生自 Exception 的异常。 这类异常被称为非 CLS 异常或非异常。 无法在 C# 中引发非 CLS 异常，但有两种方式可以捕获它们：

- 在 `catch (RuntimeWrappedException e)` 块内捕获。

  默认情况下，Visual C# 程序集将非 CLS 异常作为包装的异常捕获。 如需访问原始异常（可通过 RuntimeWrappedException.WrappedException 属性访问），请使用此方法。 本主题的后续过程将解释如何通过此方式捕获异常。

- 在位于所有其他 `catch` 块之后的常规 catch 块（未指定异常类型的 catch 块）之中。

  如果为了响应非 CLS 异常需要执行某些操作（如写入日志文件），且无需访问异常信息时，请使用此方法。 默认情况下，公共语言运行时包装所有异常。 要禁用此行为，请将此程序集级别属性添加到代码中，通常位于 AssemblyInfo.cs 文件：`[assembly: RuntimeCompatibilityAttribute(WrapNonExceptionThrows = false)]`。

要捕捉非 CLS 异常

在 `catch(RuntimeWrappedException e)` 块中通过 RuntimeWrappedException.WrappedException 属性访问原始异常。

示例

下面示例显示如何捕捉以 C++/CLI 编写的类库所引发的非 CLS 异常。 请注意，在此示例中，C# 客户端代码预先已知被引发的异常类型是 System.String。 可将 RuntimeWrappedException.WrappedException 属性转换回其原始类型，前提是可从代码访问该类型。

C#

```
// Class library written in C++/CLI.
var myClass = new ThrowNonCLS.Class1();

try
{
    // throws gcnew System::String(
    // "I do not derive from System.Exception!");
    myClass.TestThrow();
```

```
}
catch (RuntimeWrappedException e)
{
    String s = e.WrappedException as String;
    if (s != null)
    {
        Console.WriteLine(s);
    }
}
```

请参阅

- RuntimeWrappedException
- 异常和异常处理

Attributes

Article • 03/15/2023

Attributes provide a powerful method of associating metadata, or declarative information, with code (assemblies, types, methods, properties, and so forth). After an attribute is associated with a program entity, the attribute can be queried at run time by using a technique called *reflection*.

Attributes have the following properties:

- Attributes add metadata to your program. *Metadata* is information about the types defined in a program. All .NET assemblies contain a specified set of metadata that describes the types and type members defined in the assembly. You can add custom attributes to specify any additional information that is required.
- You can apply one or more attributes to entire assemblies, modules, or smaller program elements such as classes and properties.
- Attributes can accept arguments in the same way as methods and properties.
- Your program can examine its own metadata or the metadata in other programs by using reflection.

Reflection provides objects (of type Type) that describe assemblies, modules, and types. You can use reflection to dynamically create an instance of a type, bind the type to an existing object, or get the type from an existing object and invoke its methods or access its fields and properties. If you're using attributes in your code, reflection enables you to access them. For more information, see Attributes.

Here's a simple example of reflection using the GetType() method - inherited by all types from the `Object` base class - to obtain the type of a variable:

> **Note**
>
> Make sure you add `using System;` and `using System.Reflection;` at the top of your *.cs* file.

C#

```
// Using GetType to obtain type information:
int i = 42;
Type type = i.GetType();
Console.WriteLine(type);
```

The output is: `System.Int32`.

The following example uses reflection to obtain the full name of the loaded assembly.

C#

```
// Using Reflection to get information of an Assembly:
Assembly info = typeof(int).Assembly;
Console.WriteLine(info);
```

The output is something like: `System.Private.CoreLib, Version=7.0.0.0, Culture=neutral, PublicKeyToken=7cec85d7bea7798e`.

> **Note**
>
> The C# keywords `protected` and `internal` have no meaning in Intermediate Language (IL) and are not used in the reflection APIs. The corresponding terms in IL are *Family* and *Assembly*. To identify an `internal` method using reflection, use the IsAssembly property. To identify a `protected internal` method, use the IsFamilyOrAssembly.

Using attributes

Attributes can be placed on almost any declaration, though a specific attribute might restrict the types of declarations on which it's valid. In C#, you specify an attribute by placing the name of the attribute enclosed in square brackets (`[]`) above the declaration of the entity to which it applies.

In this example, the SerializableAttribute attribute is used to apply a specific characteristic to a class:

C#

```
[Serializable]
public class SampleClass
{
    // Objects of this type can be serialized.
}
```

A method with the attribute DllImportAttribute is declared like the following example:

C#

```
[System.Runtime.InteropServices.DllImport("user32.dll")]
extern static void SampleMethod();
```

More than one attribute can be placed on a declaration as the following example shows:

C#

```
void MethodA([In][Out] ref double x) { }
void MethodB([Out][In] ref double x) { }
void MethodC([In, Out] ref double x) { }
```

Some attributes can be specified more than once for a given entity. An example of such a multiuse attribute is ConditionalAttribute:

C#

```
[Conditional("DEBUG"), Conditional("TEST1")]
void TraceMethod()
{
    // ...
}
```

> **Note**
>
> By convention, all attribute names end with the word "Attribute" to distinguish them from other items in the .NET libraries. However, you do not need to specify the attribute suffix when using attributes in code. For example, `[DllImport]` is equivalent to `[DllImportAttribute]`, but `DllImportAttribute` is the attribute's actual name in the .NET Class Library.

Attribute parameters

Many attributes have parameters, which can be positional, unnamed, or named. Any positional parameters must be specified in a certain order and can't be omitted. Named parameters are optional and can be specified in any order. Positional parameters are specified first. For example, these three attributes are equivalent:

C#

```
[DllImport("user32.dll")]
[DllImport("user32.dll", SetLastError=false, ExactSpelling=false)]
[DllImport("user32.dll", ExactSpelling=false, SetLastError=false)]
```

The first parameter, the DLL name, is positional and always comes first; the others are named. In this case, both named parameters default to false, so they can be omitted. Positional parameters correspond to the parameters of the attribute constructor. Named or optional parameters correspond to either properties or fields of the attribute. Refer to the individual attribute's documentation for information on default parameter values.

For more information on allowed parameter types, see the Attributes section of the C# language specification

Attribute targets

The *target* of an attribute is the entity that the attribute applies to. For example, an attribute may apply to a class, a particular method, or an entire assembly. By default, an attribute applies to the element that follows it. But you can also explicitly identify, for example, whether an attribute is applied to a method, or to its parameter, or to its return value.

To explicitly identify an attribute target, use the following syntax:

```
C#

[target : attribute-list]
```

The list of possible `target` values is shown in the following table.

Expand table

Target value	Applies to
`assembly`	Entire assembly
`module`	Current assembly module
`field`	Field in a class or a struct
`event`	Event
`method`	Method or `get` and `set` property accessors
`param`	Method parameters or `set` property accessor parameters
`property`	Property
`return`	Return value of a method, property indexer, or `get` property accessor
`type`	Struct, class, interface, enum, or delegate

You would specify the `field` target value to apply an attribute to the backing field created for an automatically implemented property.

The following example shows how to apply attributes to assemblies and modules. For more information, see Common Attributes (C#).

C#

```
using System;
using System.Reflection;
[assembly: AssemblyTitleAttribute("Production assembly 4")]
[module: CLSCompliant(true)]
```

The following example shows how to apply attributes to methods, method parameters, and method return values in C#.

C#

```
// default: applies to method
[ValidatedContract]
int Method1() { return 0; }

// applies to method
[method: ValidatedContract]
int Method2() { return 0; }

// applies to parameter
int Method3([ValidatedContract] string contract) { return 0; }

// applies to return value
[return: ValidatedContract]
int Method4() { return 0; }
```

> ⓘ **Note**
>
> Regardless of the targets on which `ValidatedContract` is defined to be valid, the `return` target has to be specified, even if `ValidatedContract` were defined to apply only to return values. In other words, the compiler will not use `AttributeUsage` information to resolve ambiguous attribute targets. For more information, see AttributeUsage.

Common uses for attributes

The following list includes a few of the common uses of attributes in code:

- Marking methods using the `WebMethod` attribute in Web services to indicate that the method should be callable over the SOAP protocol. For more information, see WebMethodAttribute.
- Describing how to marshal method parameters when interoperating with native code. For more information, see MarshalAsAttribute.
- Describing the COM properties for classes, methods, and interfaces.
- Calling unmanaged code using the DllImportAttribute class.
- Describing your assembly in terms of title, version, description, or trademark.
- Describing which members of a class to serialize for persistence.
- Describing how to map between class members and XML nodes for XML serialization.
- Describing the security requirements for methods.
- Specifying characteristics used to enforce security.
- Controlling optimizations by the just-in-time (JIT) compiler so the code remains easy to debug.
- Obtaining information about the caller to a method.

Reflection overview

Reflection is useful in the following situations:

- When you have to access attributes in your program's metadata. For more information, see Retrieving Information Stored in Attributes.
- For examining and instantiating types in an assembly.
- For building new types at run time. Use classes in System.Reflection.Emit.
- For performing late binding, accessing methods on types created at run time. See the article Dynamically Loading and Using Types.

Related sections

For more information:

- Common Attributes (C#)
- Caller Information (C#)
- Attributes
- Reflection
- View Type Information
- Reflection and Generic Types
- System.Reflection.Emit
- Retrieving Information Stored in Attributes

创建自定义特性

项目 • 2023/03/20

可通过定义特性类创建自己的自定义特性，特性类是直接或间接派生自 Attribute 的类，可快速轻松地识别元数据中的特性定义。 假设希望使用编写类型的程序员的姓名来标记该类型。 可能需要定义一个自定义 `Author` 特性类：

C#

```
[System.AttributeUsage(System.AttributeTargets.Class |
                       System.AttributeTargets.Struct)
]
public class AuthorAttribute : System.Attribute
{
    private string Name;
    public double Version;

    public AuthorAttribute(string name)
    {
        Name = name;
        Version = 1.0;
    }
}
```

类名 `AuthorAttribute` 是该特性的名称，即 `Author` 加上 `Attribute` 后缀。 由于该类派生自 `System.Attribute`，因此它是一个自定义特性类。 构造函数的参数是自定义特性的位置参数。 在此示例中，`name` 是位置参数。 所有公共读写字段或属性都是命名参数。 在本例中，`version` 是唯一的命名参数。 请注意，使用 `AttributeUsage` 特性可使 `Author` 特性仅对类和 `struct` 声明有效。

可按如下方式使用这一新特性：

C#

```
[Author("P. Ackerman", Version = 1.1)]
class SampleClass
{
    // P. Ackerman's code goes here...
}
```

`AttributeUsage` 有一个命名参数 `AllowMultiple`，通过此命名参数可一次或多次使用自定义特性。 下面的代码示例创建了一个多用特性。

C#

```
[System.AttributeUsage(System.AttributeTargets.Class |
                       System.AttributeTargets.Struct,
                       AllowMultiple = true)  // Multiuse attribute.
]
public class AuthorAttribute : System.Attribute
{
    string Name;
    public double Version;

    public AuthorAttribute(string name)
    {
        Name = name;

        // Default value.
        Version = 1.0;
    }

    public string GetName() => Name;
}
```

在下面的代码示例中，某个类应用了同一类型的多个特性。

C#

```
[Author("P. Ackerman"), Author("R. Koch", Version = 2.0)]
public class ThirdClass
{
    // ...
}
```

请参阅

- System.Reflection
- 编写自定义特性
- AttributeUsage (C#)

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.

使用反射访问特性

项目 • 2023/03/20

你可以定义自定义特性并将其放入源代码中这一事实，在没有检索该信息并对其进行操作的方法的情况下将没有任何价值。 通过使用反射，可以检索通过自定义特性定义的信息。 主要方法是 `GetCustomAttributes`，它返回对象数组，这些对象在运行时等效于源代码特性。 此方法有许多重载版本。 有关详细信息，请参阅 Attribute。

特性规范，例如：

C#

```
[Author("P. Ackerman", Version = 1.1)]
class SampleClass { }
```

在概念上等效于以下代码：

C#

```
var anonymousAuthorObject = new Author("P. Ackerman")
{
    Version = 1.1
};
```

但是，在为特性查询 `SampleClass` 之前，代码将不会执行。 对 `SampleClass` 调用 `GetCustomAttributes` 会导致构造并初始化一个 `Author` 对象。 如果该类具有其他特性，则将以类似方式构造其他特性对象。 然后 `GetCustomAttributes` 会以数组形式返回 `Author` 对象和任何其他特性对象。 之后你便可以循环访问此数组，根据每个数组元素的类型确定所应用的特性，并从特性对象中提取信息。

下面是完整的示例。 定义自定义特性、将其应用于多个实体，并通过反射对其进行检索。

C#

```
// Multiuse attribute.
[System.AttributeUsage(System.AttributeTargets.Class |
                       System.AttributeTargets.Struct,
                       AllowMultiple = true)  // Multiuse attribute.
]
public class AuthorAttribute : System.Attribute
{
    string Name;
    public double Version;
```

```
    public AuthorAttribute(string name)
    {
        Name = name;

        // Default value.
        Version = 1.0;
    }

    public string GetName() => Name;
}

// Class with the Author attribute.
[Author("P. Ackerman")]
public class FirstClass
{
    // ...
}

// Class without the Author attribute.
public class SecondClass
{
    // ...
}

// Class with multiple Author attributes.
[Author("P. Ackerman"), Author("R. Koch", Version = 2.0)]
public class ThirdClass
{
    // ...
}

class TestAuthorAttribute
{
    public static void Test()
    {
        PrintAuthorInfo(typeof(FirstClass));
        PrintAuthorInfo(typeof(SecondClass));
        PrintAuthorInfo(typeof(ThirdClass));
    }

    private static void PrintAuthorInfo(System.Type t)
    {
        System.Console.WriteLine($"Author information for {t}");

        // Using reflection.
        System.Attribute[] attrs = System.Attribute.GetCustomAttributes(t);
// Reflection.

        // Displaying output.
        foreach (System.Attribute attr in attrs)
        {
            if (attr is AuthorAttribute a)
            {
                System.Console.WriteLine($"   {a.GetName()}, version
{a.Version:f}");
```

```
            }
        }
    }
}
/* Output:
    Author information for FirstClass
       P. Ackerman, version 1.00
    Author information for SecondClass
    Author information for ThirdClass
       R. Koch, version 2.00
       P. Ackerman, version 1.00
*/
```

另请参阅

- System.Reflection
- Attribute
- 检索存储在特性中的信息

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

如何使用特性 (C#) 创建 C/C++ 联合

项目 • 2023/04/06

通过使用特性，可自定义结构在内存中的布局方式。 例如，可使用 `StructLayout(LayoutKind.Explicit)` 和 `FieldOffset` 特性在 C/C++ 中创建所谓的联合。

在此代码段中，`TestUnion` 的所有字段均从内存中的同一位置开始。

C#

```
[System.Runtime.InteropServices.StructLayout(LayoutKind.Explicit)]
struct TestUnion
{
    [System.Runtime.InteropServices.FieldOffset(0)]
    public int i;

    [System.Runtime.InteropServices.FieldOffset(0)]
    public double d;

    [System.Runtime.InteropServices.FieldOffset(0)]
    public char c;

    [System.Runtime.InteropServices.FieldOffset(0)]
    public byte b;
}
```

以下代码是另一个示例，其中的字段从不同的显式设置位置开始。

C#

```
[System.Runtime.InteropServices.StructLayout(LayoutKind.Explicit)]
struct TestExplicit
{
    [System.Runtime.InteropServices.FieldOffset(0)]
    public long lg;

    [System.Runtime.InteropServices.FieldOffset(0)]
    public int i1;

    [System.Runtime.InteropServices.FieldOffset(4)]
    public int i2;

    [System.Runtime.InteropServices.FieldOffset(8)]
    public double d;

    [System.Runtime.InteropServices.FieldOffset(12)]
    public char c;

    [System.Runtime.InteropServices.FieldOffset(14)]
```

```
    public byte b;
}
```

组合的两个整数字段 `i1` 和 `i2` 与 `lg` 共享相同的内存位置。 `lg` 使用前 8 个字节，或 `i1` 使用前 4 个字节且 `i2` 使用后 4 个字节。 使用平台调用时，这种对结构布局的控制很有用。

请参阅

- System.Reflection
- Attribute
- 特性

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

泛型和特性

项目 • 2024/11/21

特性可按与非泛型类型相同的方式应用到泛型类型。 但是，只能将特性应用于开放式泛型类型和封闭式构造泛型类型，而不能应用于部分构造泛型类型。 开放式泛型类型是未指定任何类型参数的类型，例如 `Dictionary<TKey, TValue>`；封闭式构造泛型类型指定所有类型参数，例如 `Dictionary<string, object>`。 部分构造泛型类型指定一些（而非全部）类型参数。 示例为 `Dictionary<string, TValue>`。 *未绑定泛型类型是省略类型参数的泛型类型*，例如`Dictionary<,>`。

以下示例使用此自定义属性：

C#

```
class CustomAttribute : Attribute
{
    public object? info;
}
```

属性可以引用未绑定的泛型类型：

C#

```
public class GenericClass1<T> { }

[CustomAttribute(info = typeof(GenericClass1<>))]
class ClassA { }
```

通过使用适当数量的逗号指定多个类型参数。 在此示例中，`GenericClass2` 具有两个类型参数：

C#

```
public class GenericClass2<T, U> { }

[CustomAttribute(info = typeof(GenericClass2<,>))]
class ClassB { }
```

属性可引用封闭式构造泛型类型：

C#

```
public class GenericClass3<T, U, V> { }
```

```
[CustomAttribute(info = typeof(GenericClass3<int, double, string>))]
class ClassC { }
```

引用泛型类型参数的特性导致一个编译时错误：

C#

```
[CustomAttribute(info = typeof(GenericClass3<int, T, string>))]  //Error
CS0416
class ClassD<T> { }
```

从 C# 11 开始，泛型类型可以从 Attribute 继承：

C#

```
public class CustomGenericAttribute<T> : Attribute { }  //Requires C# 11
```

若要在运行时获取有关泛型类型或类型参数的信息，可使用 System.Reflection 方法。 有关详细信息，请参阅泛型和反射。

另请参阅

- 泛型
- 特性

如何使用反射查询程序集的元数据 (LINQ)

项目 • 2023/03/20

使用 .NET 反射 API 检查 .NET 程序集中的元数据，并创建位于该程序集中的类型、类型成员和参数的集合。 因为这些集合支持泛型 IEnumerable<T> 接口，所以可以使用 LINQ 查询它们。

下面的示例演示了如何将 LINQ 与反射配合使用以检索有关与指定搜索条件匹配的方法的特定元数据。 在这种情况下，该查询在返回数组等可枚举类型的程序集中查找所有方法的名称。

C#

```
Assembly assembly = Assembly.Load("System.Private.CoreLib, Version=7.0.0.0,
Culture=neutral, PublicKeyToken=7cec85d7bea7798e");
var pubTypesQuery = from type in assembly.GetTypes()
                    where type.IsPublic
                    from method in type.GetMethods()
                    where method.ReturnType.IsArray == true
                        || (method.ReturnType.GetInterface(

typeof(System.Collections.Generic.IEnumerable<>).FullName!) != null
                        && method.ReturnType.FullName != "System.String")
                    group method.ToString() by type.ToString();

foreach (var groupOfMethods in pubTypesQuery)
{
    Console.WriteLine("Type: {0}", groupOfMethods.Key);
    foreach (var method in groupOfMethods)
    {
        Console.WriteLine("  {0}", method);
    }
}
```

该示例使用 Assembly.GetTypes 方法返回指定程序集中的类型的数组。 将应用 where 筛选器，以便仅返回公共类型。 对于每个公共类型，子查询使用从 Type.GetMethods 调用返回的 MethodInfo 数组生成。 筛选这些结果，以仅返回其返回类型为数组或实现 IEnumerable<T> 的其他类型的方法。 最后，通过使用类型名称作为键来对这些结果进行分组。

Collaborate with us on GitHub

.NET feedback

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

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.

Open a documentation issue

Provide product feedback

泛型和反射

项目 • 2023/03/20

因为公共语言运行时 (CLR) 能够在运行时访问泛型类型信息，所以可以使用反射获取关于泛型类型的信息，方法与用于非泛型类型的方法相同。 有关详细信息，请参阅运行时中的泛型。

System.Reflection.Emit 命名空间还包含支持泛型的新成员。 请参阅如何：使用反射发出定义泛型类型。

有关泛型反射中使用的术语的固定条件列表，请参阅 IsGenericType 属性注解：

- IsGenericType：如果类型是泛型，则返回 true。
- GetGenericArguments：返回 `Type` 对象的数组，这些对象表示为构造类型提供的类型实参或泛型类型定义的类型形参。
- GetGenericTypeDefinition：返回当前构造类型的基础泛型类型定义。
- GetGenericParameterConstraints：返回表示当前泛型类型参数约束的 `Type` 对象的数组。
- ContainsGenericParameters：如果类型或任何其封闭类型或方法包含未提供特定类型的类型参数，则返回 true。
- GenericParameterAttributes：获取描述当前泛型类型参数的特殊约束的 `GenericParameterAttributes` 标志组合。
- GenericParameterPosition：对于表示类型参数的 `Type` 对象，获取类型参数在声明其类型参数的泛型类型定义或泛型方法定义的类型参数列表中的位置。
- IsGenericParameter：获取一个值，该值指示当前 `Type` 是否表示泛型类型或方法定义中的类型参数。
- IsGenericTypeDefinition：获取一个值，该值指示当前 Type 是否表示可以用来构造其他泛型类型的泛型类型定义。 如果该类型表示泛型类型的定义，则返回 true。
- DeclaringMethod：返回定义当前泛型类型参数的泛型方法，如果类型参数未由泛型方法定义，则返回 null。
- MakeGenericType：替代由当前泛型类型定义的类型参数组成的类型数组的元素，并返回表示结果构造类型的 Type 对象。

此外，MethodInfo 类的成员还为泛型方法启用运行时信息。 有关用于反射泛型方法的术语的固定条件列表，请参阅 IsGenericMethod 属性注解：

- IsGenericMethod：如果方法是泛型，则返回 true。
- GetGenericArguments：返回类型对象的数组，这些对象表示构造泛型方法的类型实参或泛型方法定义的类型形参。
- GetGenericMethodDefinition：返回当前构造方法的基础泛型方法定义。

- ContainsGenericParameters：如果方法或任何其封闭类型包含未提供特定类型的任何类型参数，则返回 true。
- IsGenericMethodDefinition：如果当前 MethodInfo 表示泛型方法的定义，则返回 true。
- MakeGenericMethod：用类型数组的元素替代当前泛型方法定义的类型参数，并返回表示结果构造方法的 MethodInfo 对象。

另请参阅

- 泛型
- 反射类型和泛型类型
- 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

定义和读取自定义特性

项目 • 2023/03/20

使用特性，可以声明的方式将信息与代码相关联。 特性还可以提供能够应用于各种目标的可重用元素。 考虑 ObsoleteAttribute。 它可以应用于类、结构、方法、构造函数等。 用于*声明*元素已过时。 然后，由 C# 编译器负责查找此特性，并执行某响应操作。

此教程介绍如何将特性添加到代码中、如何创建和使用你自己的特性，以及如何使用一些内置到 .NET 中的特性。

先决条件

需要将计算机设置为运行 .NET。 有关安装说明，请访问 .NET 下载 页。 可以在 Windows、Ubuntu Linux、macOS 或 Docker 容器中运行此应用程序。 需要安装常用的代码编辑器。 以下说明使用开放源代码跨平台编辑器 Visual Studio Code 。 不过，可以使用习惯使用的任意工具。

创建应用

安装所有工具后，创建一个新的 .NET 控制台应用。 若要使用命令行生成器，请在常用的命令行管理程序中执行以下命令：

.NET CLI

```
dotnet new console
```

此命令将创建最基本的 .NET 项目文件。 运行 `dotnet restore` 以还原编译此项目所需的依赖项。

无需运行 dotnet restore，因为它由所有需要还原的命令隐式运行，如 `dotnet new`、`dotnet build`、 `dotnet run`、 `dotnet test`、 `dotnet publish` 和 `dotnet pack`。 若要禁用隐式还原，请使用 `--no-restore` 选项。

在执行显式还原有意义的某些情况下，例如 `dotnet restore`中，或在需要显式控制还原发生时间的生成系统中， `dotnet restore` 命令仍然有用。

有关如何使用 NuGet 源的信息，请参阅 dotnet restore 文档。

若要运行程序，请使用 `dotnet run`。 此时，应该可以在控制台中看到“Hello, World”输出。

将特性添加到代码中

在 C# 中，特性是继承自 `Attribute` 基类的类。 所有继承自 `Attribute` 的类都可以用作其他代码块的一种“标记”。 例如，有一个名为 `ObsoleteAttribute` 的特性。 此特性示意代码已过时，不得再使用。 将此特性应用于类（比如说，使用方括号）。

C#

```
[Obsolete]
public class MyClass
{
}
```

虽然此类的名称为 `ObsoleteAttribute`，但只需在代码中使用 `[Obsolete]`。 大多数 C# 代码都遵循此约定。 如果愿意，也可以使用全名 `[ObsoleteAttribute]`。

如果将类标记为已过时，最好说明已过时的*原因*和/或改用的*类*。 在“已过时”特性中包含字符串参数以提供此说明。

C#

```
[Obsolete("ThisClass is obsolete. Use ThisClass2 instead.")]
public class ThisClass
{
}
```

此字符串会作为参数传递给 `ObsoleteAttribute` 构造函数，就像在编写 `var attr = new ObsoleteAttribute("some string")` 一样。

只能向特性构造函数传递以下简单类型/文本类型参数：`bool, int, double, string, Type, enums, etc`和这些类型的数组。 不能使用表达式或变量。 可以使用任何位置参数或已命名参数。

创建你自己的特性

通过定义继承自 `Attribute` 基类的新类来创建特性。

C#

```
public class MySpecialAttribute : Attribute
{
}
```

通过上述代码，可在基本代码中的其他位置将 `[MySpecial]`（或 `[MySpecialAttribute]`）用作特性。

C#

```
[MySpecial]
public class SomeOtherClass
{
}
```

.NET 基类库中的特性（如 `ObsoleteAttribute`）会在编译器中触发某些行为。 不过，你创建的任何特性只作元数据之用，并不会在执行的特性类中生成任何代码。 是否在代码的其他位置使用此元数据由你自行决定。

这里要注意的是“gotcha”。 如前所述，使用特性时，只可将某些类型作为参数传递。 不过，在创建特性类型时，C# 编译器不会阻止你创建这些参数。 在以下示例中，你使用可正常编译的构造函数创建了一个特性。

C#

```
public class GotchaAttribute : Attribute
{
    public GotchaAttribute(Foo myClass, string str)
    {
    }
}
```

不过，无法将此构造函数与特性语法结合使用。

C#

```
[Gotcha(new Foo(), "test")] // does not compile
public class AttributeFail
{
}
```

上述代码导致一个编译器错误，例如 `Attribute constructor parameter 'myClass' has type 'Foo', which is not a valid attribute parameter type`

如何限制特性使用

特性可用于以下“目标”。 前面的示例展示了特性在类上的使用情况，而特性还可用于：

- 程序集
- 类

- 构造函数
- 委托
- 枚举
- 事件
- 字段
- 泛型参数
- 接口
- 方法
- 模块
- 参数
- properties
- 返回值
- 结构

创建特性类时，C# 默认允许对所有可能的特性目标使用此特性。 如果要将特性限制为只能用于特定目标，可以对特性类使用 `AttributeUsageAttribute` 来实现。 没错，就是将特性应用于特性！

C#

```
[AttributeUsage(AttributeTargets.Class | AttributeTargets.Struct)]
public class MyAttributeForClassAndStructOnly : Attribute
{
}
```

如果尝试将上述特性应用于不是类也不是结构的对象，你会得到类似 `Attribute 'MyAttributeForClassAndStructOnly' is not valid on this declaration type. It is only valid on 'class, struct' declarations` 的编译器错误

C#

```
public class Foo
{
    // if the below attribute was uncommented, it would cause a compiler
error
    // [MyAttributeForClassAndStructOnly]
    public Foo()
    { }
}
```

如何使用附加到代码元素的特性

特性只作元数据之用。 不借助一些外在力量，特性其实什么用也没有。

若要查找特性并对其进行操作，需要使用反射。 通过反射，能够以 C# 编写检查其他代码的代码。 例如，可以使用反射获取类的相关信息（在代码开始处添加 `using System.Reflection;`）：

```
C#

TypeInfo typeInfo = typeof(MyClass).GetTypeInfo();
Console.WriteLine("The assembly qualified name of MyClass is " +
typeInfo.AssemblyQualifiedName);
```

输出如下内容：`The assembly qualified name of MyClass is ConsoleApplication.MyClass, attributes, Version=1.0.0.0, Culture=neutral, PublicKeyToken=null`

获取 `TypeInfo` 对象（或 `MemberInfo`、`FieldInfo` 或其他对象）后，就可以使用 `GetCustomAttributes` 方法了。 此方法返回 `Attribute` 对象的集合。 还可以使用 `GetCustomAttribute` 并指定特性类型。

下面的示例展示了对 `MyClass`（在上文中，它包含 `[Obsolete]` 特性）的 `MemberInfo` 实例使用 `GetCustomAttributes`。

```
C#

var attrs = typeInfo.GetCustomAttributes();
foreach(var attr in attrs)
    Console.WriteLine("Attribute on MyClass: " + attr.GetType().Name);
```

输出到控制台：`Attribute on MyClass: ObsoleteAttribute`。 请尝试向 `MyClass` 添加其他特性。

请务必注意，这些 `Attribute` 对象的实例化有延迟。 也就是说，只有使用 `GetCustomAttribute` 或 `GetCustomAttributes`，它们才会实例化。 这些对象每次都会实例化。 连续两次调用 `GetCustomAttributes` 返回两个不同的 `ObsoleteAttribute` 实例。

运行时中的常见特性

许多工具和框架都会使用特性。 NUnit 和 NUnit 测试运行程序都使用 `[Test]` 和 `[TestFixture]` 之类的特性。 ASP.NET MVC 使用 `[Authorize]` 之类的特性，并提供可密切关注 MVC 操作的操作筛选器框架。 PostSharp 使用特性语法，支持在 C# 中进行面向特性的编程。

下面介绍了一些值得注意的 .NET Core 基类库内置特性：

- `[Obsolete]`. 此特性已在上面的示例中使用过，位于 `System` 命名空间中。 可用于提供关于不断变化的基本代码的声明性文档。 可以提供字符串形式的消息，并能使用另一布尔参数将编译器警告升级为编译器错误。
- `[Conditional]`. 此特性位于 `System.Diagnostics` 命名空间中。 可应用于方法（或特性类）。 必须向构造函数传递字符串。 如果该字符串与 `#define` 指令不匹配，则 C# 编译器将删除对该方法（而不是方法本身）的所有调用。 通常，使用此方法进行调试（诊断）目的。
- `[CallerMemberName]`. 此特性可应用于参数，位于 `System.Runtime.CompilerServices` 命名空间中。 `CallerMemberName` 特性用于插入正在调用另一方法的方法的名称。 这是在各种 UI 框架中实现 INotifyPropertyChanged 时清除"神奇字符串"的一种方式。 示例：

C#

```
public class MyUIClass : INotifyPropertyChanged
{
    public event PropertyChangedEventHandler? PropertyChanged;

    public void RaisePropertyChanged([CallerMemberName] string propertyName
= default!)
    {
        PropertyChanged?.Invoke(this, new
PropertyChangedEventArgs(propertyName));
    }

    private string? _name;
    public string? Name
    {
        get { return _name;}
        set
        {
            if (value != _name)
            {
                _name = value;
                RaisePropertyChanged();   // notice that "Name" is not
needed here explicitly
            }
        }
    }
}
```

在上面的代码中，无需使用文本类型 `"Name"` 字符串。 使用 `CallerMemberName` 可防止出现与拼写错误相关的 bug，也可以让重构/重命名操作变得更加顺畅。 特性为 C# 带来了声明性能力，但它们是一种元数据形式的代码，本身并不执行操作。

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

教程：使用默认接口方法更新接口

项目 • 2023/03/22

可以在声明接口成员时定义实现。 最常见的方案是安全地将成员添加到已经由无数客户端发布并使用的接口。

在本教程中，你将了解：

- ✓ 通过使用实现添加方法，安全地扩展接口。
- ✓ 创建参数化实现以提供更大的灵活性。
- ✓ 使实现器能够以替代的形式提供更具体的实现。

先决条件

需要将计算机设置为运行 .NET，包括 C# 编译器。 Visual Studio 2022 或 .NET SDK 随附 C# 编译器。

方案概述

本教程从客户关系库版本 1 开始。 可以在 GitHub 上的示例存储库中获取入门应用程序。 生成此库的公司希望拥有现有应用程序的客户采用其库。 他们为使用其库的用户提供最小接口定义供其实现。 以下是客户的接口定义：

C#

```
public interface ICustomer
{
    IEnumerable<IOrder> PreviousOrders { get; }

    DateTime DateJoined { get; }
    DateTime? LastOrder { get; }
    string Name { get; }
    IDictionary<DateTime, string> Reminders { get; }
}
```

他们定义了表示订单的第二个接口：

C#

```
public interface IOrder
{
    DateTime Purchased { get; }
```

```
    decimal Cost { get; }
}
```

通过这些接口，团队可以为其用户生成一个库，以便为其客户创造更好的体验。 他们的目标是与现有客户建立更深入的关系，并改善他们与新客户的关系。

现在，是时候为下一版本升级库了。 其中一个请求的功能可以为拥有大量订单的客户提供忠实客户折扣。 无论客户何时下单，都会应用这一新的忠实客户折扣。 该特定折扣是每位客户的财产。 `ICustomer` 的每个实现都可以为忠实客户折扣设置不同的规则。

添加此功能的最自然方式是使用用于应用任何忠实客户折扣的方法来增强 `ICustomer` 接口。 此设计建议引起了经验丰富的开发人员的关注："一旦发布，接口就是固定不变的！请勿进行中断性变更！"用于升级接口的默认接口实现。 库作者可以向接口添加新成员，并为这些成员提供默认实现。

默认接口实现使开发人员能够升级接口，同时仍允许任何实现器替代该实现。 库的用户可以接受默认实现作为非中断性变更。 如果他们的业务规则不同，则可以进行替代。

使用默认接口方法升级

团队就最有可能的默认实现达成一致：针对客户的忠实客户折扣。

升级应提供用于设置两个属性的功能：符合折扣条件所需的订单数量以及折扣百分比。 这些功能使其成为用于默认接口方法的完美方案。 可以向 `ICustomer` 接口添加方法，并提供最有可能的实现。 所有现有的和任何新的实现都可以使用默认实现，或者提供其自己的实现。

首先，将新方法添加到接口，包括方法的主体：

C#

```
// Version 1:
public decimal ComputeLoyaltyDiscount()
{
    DateTime TwoYearsAgo = DateTime.Now.AddYears(-2);
    if ((DateJoined < TwoYearsAgo) && (PreviousOrders.Count() > 10))
    {
        return 0.10m;
    }
    return 0;
}
```

库作者编写了用于检查实现的第一个测试：

C#

```
SampleCustomer c = new SampleCustomer("customer one", new DateTime(2010, 5,
31))
{
    Reminders =
    {
        { new DateTime(2010, 08, 12), "childs's birthday" },
        { new DateTime(1012, 11, 15), "anniversary" }
    }
};

SampleOrder o = new SampleOrder(new DateTime(2012, 6, 1), 5m);
c.AddOrder(o);

o = new SampleOrder(new DateTime(2103, 7, 4), 25m);
c.AddOrder(o);

// Check the discount:
ICustomer theCustomer = c;
Console.WriteLine($"Current discount:
{theCustomer.ComputeLoyaltyDiscount()}");
```

注意测试的以下部分：

C#

```
// Check the discount:
ICustomer theCustomer = c;
Console.WriteLine($"Current discount:
{theCustomer.ComputeLoyaltyDiscount()}");
```

从 `SampleCustomer` 到 `ICustomer` 的强制转换是必需的。 `SampleCustomer` 类不需要为 `ComputeLoyaltyDiscount` 提供实现；这由 `ICustomer` 接口提供。 但是，`SampleCustomer` 类不会从其接口继承成员。 该规则没有更改。 若要调用在接口中声明和实现的任何方法，该变量的类型必须是接口的类型，在本示例中为 `ICustomer`。

提供参数化

默认实现存在太多限制。 此系统的许多使用者可能会选择不同的购买数量阈值、不同的会员资格时长或不同的折扣百分比。 通过提供用于设置这些参数的方法，可为更多客户提供更好的升级体验。 让我们添加一个静态方法，该方法可设置控制默认实现的三个参数：

C#

```
// Version 2:
public static void SetLoyaltyThresholds(
```

```
    TimeSpan ago,
    int minimumOrders = 10,
    decimal percentageDiscount = 0.10m)
{
    length = ago;
    orderCount = minimumOrders;
    discountPercent = percentageDiscount;
}
private static TimeSpan length = new TimeSpan(365 * 2, 0,0,0); // two years
private static int orderCount = 10;
private static decimal discountPercent = 0.10m;

public decimal ComputeLoyaltyDiscount()
{
    DateTime start = DateTime.Now - length;

    if ((DateJoined < start) && (PreviousOrders.Count() > orderCount))
    {
        return discountPercent;
    }
    return 0;
}
```

这个小代码片段中展示了许多新的语言功能。 接口现在可以包含静态成员，其中包括字段和方法。 还启用了不同的访问修饰符。 其他字段是专用的，新方法是公共的。 接口成员允许使用任何修饰符。

使用常规公式计算忠实客户折扣但参数有所不同的应用程序不需要提供自定义实现；它们可以通过静态方法设置自变量。 例如，以下代码设置“客户答谢”，奖励任何成为会员超过一个月的客户：

C#

```
ICustomer.SetLoyaltyThresholds(new TimeSpan(30, 0, 0, 0), 1, 0.25m);
Console.WriteLine($"Current discount:
{theCustomer.ComputeLoyaltyDiscount()}");
```

扩展默认实现

目前添加的代码提供了方便的实现，可用于用户需要类似默认实现的项目的方案，或用于提供一组不相关的规则。 对于最后一个功能，让我们稍微重构一下代码，以实现用户可能需要基于默认实现进行生成的方案。

假设有一家想要吸引新客户的初创企业。 他们为新客户的第一笔订单提供 50% 的折扣，而现有客户则会获得标准折扣。 库作者需要将默认实现移入 `protected static` 方法，以便实现此接口的任何类都可以在其实现中重用代码。 接口成员的默认实现也调用此共享方法：

C#

```
public decimal ComputeLoyaltyDiscount() => DefaultLoyaltyDiscount(this);
protected static decimal DefaultLoyaltyDiscount(ICustomer c)
{
    DateTime start = DateTime.Now - length;

    if ((c.DateJoined < start) && (c.PreviousOrders.Count() > orderCount))
    {
        return discountPercent;
    }
    return 0;
}
```

在实现此接口的类的实现中，替代可以调用静态帮助程序方法，并扩展该逻辑以提供“新客户”折扣：

C#

```
public decimal ComputeLoyaltyDiscount()
{
   if (PreviousOrders.Any() == false)
        return 0.50m;
    else
        return ICustomer.DefaultLoyaltyDiscount(this);
}
```

可以在 GitHub 上的示例存储库 中查看整个完成的代码。 可以在 GitHub 上的示例存储库 中获取入门应用程序。

这些新功能意味着，当这些新成员拥有合理的默认实现时，接口可以安全地更新。 精心设计接口，以表达由多个类实现的单个功能概念。 这样一来，在发现针对同一功能概念的新要求时，可以更轻松地升级这些接口定义。

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

教程：当通过默认接口方法创建使用接口的类时实现的混入功能

项目 • 2024/08/05

可以在声明接口成员时定义实现。 此功能提供了一些新功能，可以在其中为接口中声明的功能定义默认实现。 类可以选择何时替代功能、何时使用默认功能以及何时不声明对离散功能的支持。

本教程介绍如何执行下列操作：

- ✓ 使用描述离散功能的实现创建接口。
- ✓ 创建使用默认实现的类。
- ✓ 创建用于替代部分或全部默认实现的类。

先决条件

需要将计算机设置为运行 .NET，包括 C# 编译器。 Visual Studio 2022 或 .NET SDK 随附 C# 编译器。

扩展方法的限制

实现作为接口一部分出现的行为的一种方法是：定义可提供默认行为的扩展方法。 接口声明最少的一组成员，同时为实现该接口的任何类提供更大的外围应用。 例如，Enumerable 中的扩展方法提供作为 LINQ 查询源的任何序列的实现。

扩展方法是使用变量的声明类型在编译时解析的。 实现接口的类可以为任何扩展方法提供更好的实现。 变量声明必须与实现类型匹配，以使编译器能够选择该实现。 当编译时类型与接口匹配时，方法调用解析为扩展方法。 扩展方法的另一个问题是，只要可访问包含扩展方法的类，就可以访问这些方法。 类不能声明是否应提供扩展方法中声明的功能。

可以将默认实现声明为接口方法。 然后，每个类将自动使用默认实现。 可提供更好实现的任何类都可以使用更好的算法来替代接口方法定义。 从某种意义上讲，这种技术听起来与你使用扩展方法的方式类似。

在本文中，你将了解默认接口实现如何启用新方案。

设计应用程序

考虑使用一个家庭自动化应用程序。 你可能在整个房子中使用许多不同类型的灯和指示灯。 每个灯都必须支持 API 以使其打开和关闭，以及报告当前状态。 一些灯和指示灯可能支持其他功能，例如：

- 打开灯，然后定时关闭。
- 使灯闪烁一段时间。

在支持最小集的设备中，可以模拟其中的某些扩展功能。 这表示提供了默认实现。 对于内置了更多功能的设备，设备软件将使用本机功能。 对于其他灯，它们可以选择实现接口并使用默认实现。

对于此方案，默认接口成员提供比扩展方法更好的解决方案。 类创建者可以控制它们选择实现的接口。 它们选择的接口可用作方法。 此外，由于默认情况下默认的接口方法是虚拟的，因此该方法调度始终选择类中的实现。

我们创建代码来演示这些差异。

创建接口

首先创建用于定义所有灯的行为的接口：

C#

```
public interface ILight
{
    void SwitchOn();
    void SwitchOff();
    bool IsOn();
}
```

基本的高架灯具可能会实现此接口，如下面的代码所示：

C#

```
public class OverheadLight : ILight
{
    private bool isOn;
    public bool IsOn() => isOn;
    public void SwitchOff() => isOn = false;
    public void SwitchOn() => isOn = true;

    public override string ToString() => $"The light is {(isOn ? "on" :
"off")}";
}
```

在本教程中，代码不驱动 IoT 设备，但会通过将消息写入控制台的方式来模拟这些活动。 可以浏览代码，但不执行房屋的自动化。

接下来，我们定义一个可在超时后自动关闭的灯的接口：

C#

```
public interface ITimerLight : ILight
{
    Task TurnOnFor(int duration);
}
```

可以向高架灯具添加基本实现，但是更好的解决方案是修改此接口定义以提供 `virtual` 默认实现：

C#

```
public interface ITimerLight : ILight
{
    public async Task TurnOnFor(int duration)
    {
        Console.WriteLine("Using the default interface method for the
ITimerLight.TurnOnFor.");
        SwitchOn();
        await Task.Delay(duration);
        SwitchOff();
        Console.WriteLine("Completed ITimerLight.TurnOnFor sequence.");
    }
}
```

`OverheadLight` 类可以通过声明对接口的支持来实现计时器功能：

C#

```
public class OverheadLight : ITimerLight { }
```

另一种灯类型可能支持更复杂的协议。 它可以为 `TurnOnFor` 提供自己的实现，如以下代码所示：

C#

```
public class HalogenLight : ITimerLight
{
    private enum HalogenLightState
    {
        Off,
        On,
        TimerModeOn
```

```
    }

    private HalogenLightState state;
    public void SwitchOn() => state = HalogenLightState.On;
    public void SwitchOff() => state = HalogenLightState.Off;
    public bool IsOn() => state != HalogenLightState.Off;
    public async Task TurnOnFor(int duration)
    {
        Console.WriteLine("Halogen light starting timer function.");
        state = HalogenLightState.TimerModeOn;
        await Task.Delay(duration);
        state = HalogenLightState.Off;
        Console.WriteLine("Halogen light finished custom timer function");
    }

    public override string ToString() => $"The light is {state}";
}
```

与替代虚拟类方法不同，`HalogenLight` 类中 `TurnOnFor` 的声明不使用 `override` 关键字。

混合和匹配功能

当你引入更高级的功能时，默认界面方法的优势变得更加明显。 使用接口让你可以混合和匹配功能。 它还使每个类创建者可以在默认实现和自定义实现之间进行选择。 我们使用闪烁灯的默认实现来添加一个接口：

C#

```
public interface IBlinkingLight : ILight
{
    public async Task Blink(int duration, int repeatCount)
    {
        Console.WriteLine("Using the default interface method for
IBlinkingLight.Blink.");
        for (int count = 0; count < repeatCount; count++)
        {
            SwitchOn();
            await Task.Delay(duration);
            SwitchOff();
            await Task.Delay(duration);
        }
        Console.WriteLine("Done with the default interface method for
IBlinkingLight.Blink.");
    }
}
```

默认实现使任何灯可以闪烁。 高架灯可以使用默认实现添加计时器和闪烁功能：

C#

```
public class OverheadLight : ILight, ITimerLight, IBlinkingLight
{
    private bool isOn;
    public bool IsOn() => isOn;
    public void SwitchOff() => isOn = false;
    public void SwitchOn() => isOn = true;

    public override string ToString() => $"The light is {(isOn ? "on" :
"off")}";
}
```

新的灯类型 `LEDLight` 直接支持计时器功能和闪烁功能。 这种灯样式同时实现 `ITimerLight` 和 `IBlinkingLight` 接口，并替代 `Blink` 方法：

C#

```
public class LEDLight : IBlinkingLight, ITimerLight, ILight
{
    private bool isOn;
    public void SwitchOn() => isOn = true;
    public void SwitchOff() => isOn = false;
    public bool IsOn() => isOn;
    public async Task Blink(int duration, int repeatCount)
    {
        Console.WriteLine("LED Light starting the Blink function.");
        await Task.Delay(duration * repeatCount);
        Console.WriteLine("LED Light has finished the Blink function.");
    }

    public override string ToString() => $"The light is {(isOn ? "on" :
"off")}";
}
```

`ExtraFancyLight` 可以直接支持闪烁功能和计时器功能：

C#

```
public class ExtraFancyLight : IBlinkingLight, ITimerLight, ILight
{
    private bool isOn;
    public void SwitchOn() => isOn = true;
    public void SwitchOff() => isOn = false;
    public bool IsOn() => isOn;
    public async Task Blink(int duration, int repeatCount)
    {
        Console.WriteLine("Extra Fancy Light starting the Blink function.");
        await Task.Delay(duration * repeatCount);
        Console.WriteLine("Extra Fancy Light has finished the Blink
function.");
    }
```

```
    public async Task TurnOnFor(int duration)
    {
        Console.WriteLine("Extra Fancy light starting timer function.");
        await Task.Delay(duration);
        Console.WriteLine("Extra Fancy light finished custom timer
function");
    }

    public override string ToString() => $"The light is {(isOn ? "on" :
"off")}";
}
```

之前创建的 `HalogenLight` 不支持闪烁。 因此，不要将 `IBlinkingLight` 添加到其支持的接口列表。

使用模式匹配检测灯类型

接下来，我们编写一些测试代码。 可以使用 C# 的模式匹配功能，通过检查灯支持的接口来确定灯的功能。 下面的方法将实践每个灯的支持功能：

C#

```
private static async Task TestLightCapabilities(ILight light)
{
    // Perform basic tests:
    light.SwitchOn();
    Console.WriteLine($"\tAfter switching on, the light is {(light.IsOn() ?
"on" : "off")}");
    light.SwitchOff();
    Console.WriteLine($"\tAfter switching off, the light is {(light.IsOn() ?
"on" : "off")}");

    if (light is ITimerLight timer)
    {
        Console.WriteLine("\tTesting timer function");
        await timer.TurnOnFor(1000);
        Console.WriteLine("\tTimer function completed");
    }
    else
    {
        Console.WriteLine("\tTimer function not supported.");
    }

    if (light is IBlinkingLight blinker)
    {
        Console.WriteLine("\tTesting blinking function");
        await blinker.Blink(500, 5);
        Console.WriteLine("\tBlink function completed");
    }
    else
    {
```

```
        Console.WriteLine("\tBlink function not supported.");
    }
}
```

`Main` 方法中的以下代码按顺序创建每种灯类型，并测试相应的灯：

C#

```
static async Task Main(string[] args)
{
    Console.WriteLine("Testing the overhead light");
    var overhead = new OverheadLight();
    await TestLightCapabilities(overhead);
    Console.WriteLine();

    Console.WriteLine("Testing the halogen light");
    var halogen = new HalogenLight();
    await TestLightCapabilities(halogen);
    Console.WriteLine();

    Console.WriteLine("Testing the LED light");
    var led = new LEDLight();
    await TestLightCapabilities(led);
    Console.WriteLine();

    Console.WriteLine("Testing the fancy light");
    var fancy = new ExtraFancyLight();
    await TestLightCapabilities(fancy);
    Console.WriteLine();
}
```

编译器如何确定最佳实现

此方案显示了没有任何实现的基本接口。 将方法添加到 `ILight` 接口带来了新的复杂性。 管理默认接口方法的语言规则最大程度地减少了对实现多个派生接口的具体类的影响。 我们使用新方法增强原始接口的功能，以演示如何更改其用法。 每个指示灯都可以将其电源状态报告为枚举值：

C#

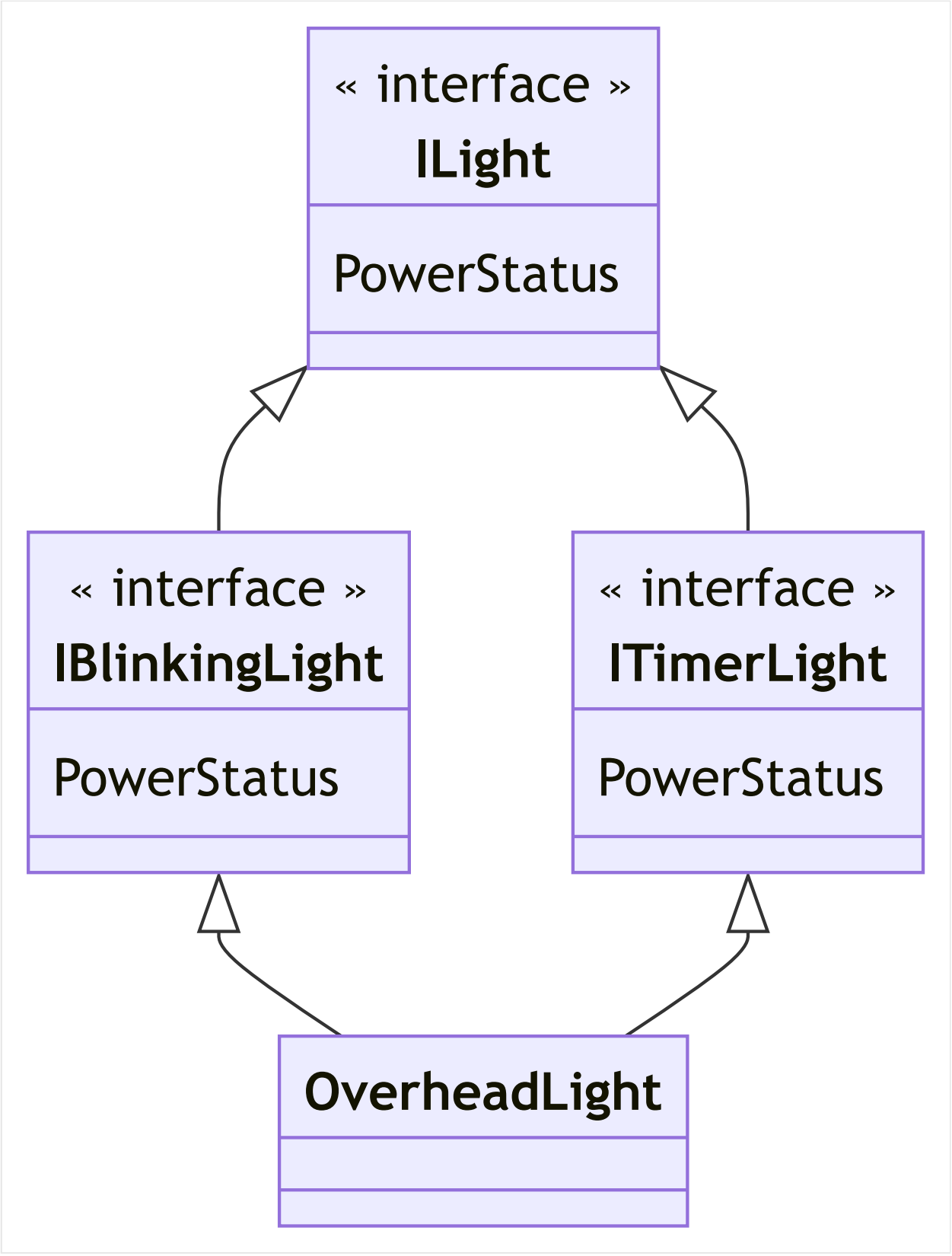
```
public enum PowerStatus
{
    NoPower,
    ACPower,
    FullBattery,
    MidBattery,
    LowBattery
}
```

默认实现不使用电源：

```C#
public interface ILight
{
    void SwitchOn();
    void SwitchOff();
    bool IsOn();
    public PowerStatus Power() => PowerStatus.NoPower;
}
```

即使 `ExtraFancyLight` 声明支持 `ILight` 接口以及派生接口 `ITimerLight` 和 `IBlinkingLight`，这些更改仍会进行清晰的编译。 在 `ILight` 接口中只有一个声明为“最接近”的实现。 任何声明了替代的类都将成为一个“最接近”的实现。 你在前面的类中看到了替代其他派生接口成员的示例。

避免在多个派生接口中替代相同的方法。 这样做会在类实现两个派生接口时创建不明确的方法调用。 编译器不能选取一种更好的方法，因此会发出错误。 例如，如果 `IBlinkingLight` 和 `ITimerLight` 实现了 `PowerStatus` 的替代，则 `OverheadLight` 需要提供更具体的替代。 否则，编译器无法在两个派生接口的实现之间进行选择。 下图显示了这种情况：



上图说明了歧义。 `OverheadLight` 没有提供 `ILight.PowerStatus` 的实现。 `IBlinkingLight` 和 `ITimerLight` 都提供了更具体的覆盖。 对 `OverheadLight` 实例上的 `ILight.PowerStatus` 调用存在歧义。 必须在 `OverheadLight` 中添加新的覆盖来解决歧义。

通常，为避免这种情况，可以使接口定义保持较小并专注于一个功能。 在此方案中，灯的每个功能都是其自己的接口；只有类可以继承多个接口。

此示例演示了一种方案，在该方案中可以定义可混合到类中的离散功能。 通过声明类支持的接口，可以声明任意一组受支持的功能。 使用虚拟默认接口方法使类可以使用或定义任何或所有接口方法的不同实现。 这种语言功能提供了对正在构建的真实系统进行建模的新方法。 默认接口方法提供了一种更清晰的方式来表达相关的类，这些类可能使用这些功能的虚拟实现来混合和匹配不同的功能。

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

Expression Trees

Article • 05/29/2024

Expression trees represent code in a tree-like data structure, where each node is an expression, for example, a method call or a binary operation such as `x < y`.

If you used LINQ, you have experience with a rich library where the `Func` types are part of the API set. (If you aren't familiar with LINQ, you probably want to read the LINQ tutorial and the article about lambda expressions before this one.) Expression Trees provide richer interaction with the arguments that are functions.

You write function arguments, typically using Lambda Expressions, when you create LINQ queries. In a typical LINQ query, those function arguments are transformed into a delegate the compiler creates.

You already write code that uses Expression trees. Entity Framework's LINQ APIs accept Expression trees as the arguments for the LINQ Query Expression Pattern. That enables Entity Framework to translate the query you wrote in C# into SQL that executes in the database engine. Another example is Moq, which is a popular mocking framework for .NET.

When you want to have a richer interaction, you need to use *Expression Trees*. Expression Trees represent code as a structure that you examine, modify, or execute. These tools give you the power to manipulate code during run time. You write code that examines running algorithms, or injects new capabilities. In more advanced scenarios, you modify running algorithms and even translate C# expressions into another form for execution in another environment.

You compile and run code represented by expression trees. Building and running expression trees enables dynamic modification of executable code, the execution of LINQ queries in various databases, and the creation of dynamic queries. For more information about expression trees in LINQ, see How to use expression trees to build dynamic queries.

Expression trees are also used in the dynamic language runtime (DLR) to provide interoperability between dynamic languages and .NET and to enable compiler writers to emit expression trees instead of Microsoft intermediate language (CIL). For more information about the DLR, see Dynamic Language Runtime Overview.

You can have the C# or Visual Basic compiler create an expression tree for you based on an anonymous lambda expression, or you can create expression trees manually by using the System.Linq.Expressions namespace.

When a lambda expression is assigned to a variable of type Expression<TDelegate>, the compiler emits code to build an expression tree that represents the lambda expression.

The following code examples demonstrate how to have the C# compiler create an expression tree that represents the lambda expression `num => num < 5`.

```
C#

Expression<Func<int, bool>> lambda = num => num < 5;
```

You create expression trees in your code. You build the tree by creating each node and attaching the nodes into a tree structure. You learn how to create expressions in the article on building expression trees.

Expression trees are immutable. If you want to modify an expression tree, you must construct a new expression tree by copying the existing one and replacing nodes in it. You use an expression tree visitor to traverse the existing expression tree. For more information, see the article on translating expression trees.

Once you build an expression tree, you execute the code represented by the expression tree.

Limitations

The C# compiler generates expression trees only from expression lambdas (or single-line lambdas). It can't parse statement lambdas (or multi-line lambdas). For more information about lambda expressions in C#, see Lambda Expressions.

There are some newer C# language elements that don't translate well into expression trees. Expression trees can't contain `await` expressions, or `async` lambda expressions. Many of the features added in C# 6 and later don't appear exactly as written in expression trees. Instead, newer features are exposed in expression trees in the equivalent, earlier syntax, where possible. Other constructs aren't available. It means that code that interprets expression trees works the same when new language features are introduced. However, even with these limitations, expression trees do enable you to create dynamic algorithms that rely on interpreting and modifying code that is represented as a data structure. It enables rich libraries such as Entity Framework to accomplish what they do.

Expression trees won't support new expression node types. It would be a breaking change for all libraries interpreting expression trees to introduce new node types. The following list includes most C# language elements that can't be used:

- Conditional methods removed from the output
- base access
- Method group expressions, including *address-of* (&) a method group, and anonymous method expressions
- References to local functions
- Statements, including assignment (`=`) and statement bodied expressions
- Partial methods with only a defining declaration
- Unsafe pointer operations
- dynamic operations
- Coalescing operators with null or default literal left side, null coalescing assignment, and the null propagating operator (?.)
- Multi-dimensional array initializers, indexed properties, and dictionary initializers
- Collection expressions
- throw expressions
- Accessing static virtual or abstract interface members
- Lambda expressions that have attributes
- Interpolated strings
- UTF-8 string conversions or UTF-8 string literals
- Method invocations using variable arguments, named arguments, or optional arguments
- Expressions using System.Index or System.Range, index "from end" (^) operator or range expressions (..)
- async lambda expressions or await expressions, including await foreach and await using
- Tuple literals, tuple conversions, tuple == or !=, or with expressions
- Discards (_), deconstructing assignment, pattern matching is operator, or the pattern matching switch expression
- COM call with `ref` omitted on the arguments
- ref, in or out parameters, `ref` return values, `out` arguments, or any values of ref struct type

表达式树 - 定义代码的数据

项目 • 2023/03/13

表达式树是定义代码的数据结构。 表达式树基于编译器用于分析代码和生成已编译输出的相同结构。 读完本文后，你会注意到表达式树和 Roslyn API 中用于生成分析器和 CodeFixes 的类型之间存在很多相似之处。 （分析器和 CodeFix 是对代码执行静态分析的 NuGet 包，可为开发人员提供针对潜在修复的建议。）概念是类似的，最终结果是可用于以有意义的方式检查源代码的数据结构。 但是，表达式树基于一组与 Roslyn API 不同的类和 API。 以下是一个代码行：

C#

```
var sum = 1 + 2;
```

如果将上面的代码作为一个表达式树进行分析，则该树包含多个节点。 最外面的节点是具有赋值（`var sum = 1 + 2;`）的变量声明语句，该节点包含若干子节点：变量声明、赋值运算符和一个表示等于号右侧的表达式。 该表达式被进一步细分为表示加法运算、该加法左操作数和右操作数的表达式。

让我们稍微深入了解一下构成等于号右侧的表达式。 表达式是 `1 + 2`，一个二进制表达式。 更具体地说，它是一个二进制加法表达式。 二进制加法表达式有两个子表达式，表示加法表达式的左侧和右侧节点。 此处，这两个节点均为常数表达式：左操作数是值 `1`，右操作数是值 `2`。

直观地看，整个语句是一棵树：应从根节点开始，浏览到树中的每个节点，以查看构成该语句的代码：

- 具有赋值（`var sum = 1 + 2;`）的变量声明语句
  - 隐式变量类型声明（`var sum`）
    - 隐式 var 关键字（`var`）
    - 变量名称声明（`sum`）
  - 赋值运算符（`=`）
  - 二进制加法表达式（`1 + 2`）
    - 左操作数（`1`）
    - 加法运算符（`+`）
    - 右操作数（`2`）

前面的树可能看起来很复杂，但它的功能非常强大。 按照相同的过程，分解更加复杂的表达式。 请思考此表达式：

C#

```
var finalAnswer = this.SecretSauceFunction(
    currentState.createInterimResult(), currentState.createSecondValue(1,
2),
    decisionServer.considerFinalOptions("hello")) +
    MoreSecretSauce('A', DateTime.Now, true);
```

上述表达式也是具有赋值的变量声明。 在此情况下，赋值的右侧是一棵更加复杂的树。 你不打算分解此表达式，但请思考一下不同的节点可能是什么。 存在使用当前对象作为接收方的方法调用，其中一个调用具有显式 `this` 接收方，一个调用不具有此接收方。 存在使用其他接收方对象的方法调用，存在不同类型的常量参数。 最后，存在二进制加法运算符。 该二进制加法运算符可能是对重写的加法运算符的方法调用（具体取决于 `SecretSauceFunction()` 或 `MoreSecretSauce()` 的返回类型），解析为对为类定义的二进制加法运算符的静态方法调用。

尽管具有这种感知上的复杂性，但上面的表达式创建了一种树形结构，可以像第一个示例那样轻松地导航此结构。 保持遍历子节点，以查找表达式中的叶节点。 父节点具有对其子节点的引用，且每个节点均具有一个用于介绍节点类型的属性。

表达式树的结构非常一致。 了解基础知识后，你甚至可以理解以表达式树形式表示的最复杂的代码。 优美的数据结构说明了 C# 编译器如何分析最复杂的 C# 程序并从该复杂的源代码创建正确的输出。

熟悉表达式树的结构后，你发现通过快速获得的知识，你可处理许多更加高级的方案。 表达式树的功能非常强大。

除了转换算法以在其他环境中执行之外，表达式树还可在执行代码前轻松编写检查代码的算法。 编写参数为表达式的方法，然后在执行代码之前检查这些表达式。 表达式树是代码的完整表示形式：你看到了任何子表达式的值。 你看到了方法和属性名称。 你看到了任何常数表达式的值。 将表达式树转换为可执行的委托，并执行代码。

通过表达式树的 API，可创建表示几乎任何有效代码构造的树。 但是，出于尽可能简化的考虑，不能在表达式树中创建某些 C# 习惯用语。 其中一个示例就是异步表达式（使用 `async` 和 `await` 关键字）。 如果需要异步算法，则需要直接操作 `Task` 对象，而不是依赖于编译器支持。 另一个示例是创建循环。 通常，通过使用 `for`、`foreach`、`while` 或 `do` 循环创建这些循环。 正如稍后可以在本系列中看到的那样，表达式树的 API 支持单个循环表达式，该表达式包含控制重复循环的 `break` 和 `continue` 表达式。

不能执行的操作是修改表达式树。 表达式树是不可变的数据结构。 如果想要改变（更改）表达式树，则必须创建基于原始树副本但包含所需更改的新树。

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

.NET 运行时对表达式树的支持

项目 • 2023/03/13

存在可与表达式树配合使用的 .NET 运行时中的类的大型列表。 可以在 System.Linq.Expressions 查看完整列表。 让我们来了解一下运行时类的设计方式，而不是枚举完整列表。

在语言设计中，表达式是可计算并返回值的代码主体。 表达式可能很简单：常数表达式 `1` 返回常数值 1。 也可能较复杂：表达式 `(-B + Math.Sqrt(B*B - 4 * A * C)) / (2 * A)` 返回二次方程的一个根（若方程有解）。

System.Linq.Expression 和派生类型

使用表达式树的其中一个难点在于许多不同类型的表达式在程序中的许多位置均有效。 请思考一个赋值表达式。 赋值的右侧可以是常数值、变量、方法调用表达式或其他内容。 语言灵活性意味着，遍历表达式树时，可能会在树的节点中的任意位置遇到许多不同的表达式类型。 因此，使用基表达式类型时，理解起来最简单。 但是，有时你需要了解更多内容。 为此，基表达式类包含 `NodeType` 属性。 它将返回 `ExpressionType`，这是可能的表达式类型的枚举。 知道节点的类型后，将其转换为该类型，并执行特定操作（如果知道表达式节点的类型）。 可以搜索特定的节点类型，然后使用这种表达式的特定属性。

例如，此代码打印变量访问表达式的变量的名称。 以下代码演示了做法，先查看节点类型，再强制转换为变量访问表达式，然后查看特定表达式类型的属性：

C#

```
Expression<Func<int, int>> addFive = (num) => num + 5;

if (addFive is LambdaExpression lambdaExp)
{
    var parameter = lambdaExp.Parameters[0];  -- first

    Console.WriteLine(parameter.Name);
    Console.WriteLine(parameter.Type);
}
```

创建表达式树

`System.Linq.Expression` 类还包含许多创建表达式的静态方法。 这些方法使用为子节点提供的参数创建表达式节点。 通过这种方式，从其叶节点构建一个表达式。 例如，此代

码将生成一个 Add 表达式：

```
C#
```

```
// Addition is an add expression for "1 + 2"
var one = Expression.Constant(1, typeof(int));
var two = Expression.Constant(2, typeof(int));
var addition = Expression.Add(one, two);
```

从这个简单的示例中，你会发现创建和使用表达式树涉及了许多类型。 该复杂性是提供由 C# 语言提供的丰富词汇的功能所必需的。

导航 API

存在映射到 C# 语言的几乎所有语法元素的表达式节点类型。 每种类型都有针对该种语言元素的特定方法。 需要一次性记住的内容很多。 与其尝试记住所有内容，不如采用以下有关使用表达式树的技巧：

1. 查看 `ExpressionType` 枚举的成员以确定应检查的可能节点。 如果想要遍历和理解表达式树，此列表将会有用。
2. 查看 `Expression` 类的静态成员以生成表达式。 这些方法可以从其子节点集生成任何表达式类型。
3. 查看 `ExpressionVisitor` 类，以生成一个经过修改的表达式树。

如果依次查看这三个部分的每个部分，可以发现更多内容。 通过使用这三个步骤中的任意一个步骤，你一定会发现所需的内容。

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 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

执行表达式树

项目 • 2023/03/13

*表达式树*是表示一些代码的数据结构。 它不是经过编译且可执行的代码。 如果想要执行由表达式树表示的 .NET 代码，必须将其转换为可执行的 IL 指令。 执行表达式树可能返回一个值，或者它可能只是执行操作，例如调用方法。

仅可以执行表示 lambda 表达式的表达式树。 表示 Lambda 表达式的表达式树的类型为 LambdaExpression 或 Expression<TDelegate>。 若要执行这些表达式树，请调用 Compile 方法来创建一个可执行的委托，然后调用该委托。

> **备注**
>
> 如果委托的类型未知，也就是说 Lambda 表达式的类型为 **LambdaExpression** 而不是 **Expression<TDelegate>**，则对委托调用 **DynamicInvoke** 方法，而不是直接调用委托。

如果表达式树不表示 Lambda 表达式，可以通过调用 Lambda<TDelegate>(Expression, IEnumerable<ParameterExpression>) 方法创建一个新的 Lambda 表达式，此表达式的主体为原始表达式树。 然后，按本节前面所述执行此 lambda 表达式。

Lambda 表达式到函数

可以将任何 LambdaExpression 或派生自 LambdaExpression 的任何类型转换为可执行的 IL。 其他表达式类型不能直接转换为代码。 此限制在实践中影响不大。 Lambda 表达式是你可通过转换为可执行的中间语言 (IL) 来执行的唯一表达式类型。 （思考直接执行 System.Linq.Expressions.ConstantExpression 意味着什么。这是否意味着任何用处？）System.Linq.Expressions.LambdaExpression 或派生自 `LambdaExpression` 的类型的任何表达式树均可转换为 IL。 表达式类型 System.Linq.Expressions.Expression<TDelegate> 是 .NET Core 库中的唯一具体示例。 它用于表示映射到任何委托类型的表达式。 由于此类型映射到一个委托类型，因此 .NET 可以检查表达式，并为匹配 lambda 表达式签名的适当委托生成 IL。 该委托类型基于表达式类型。 如果想要以强类型的方式使用委托对象，则必须知道返回类型和参数列表。 `LambdaExpression.Compile()` 方法返回 `Delegate` 类型。 必须将它强制转换为正确的委托类型，以便使任何编译时工具检查参数列表或返回类型。

在大多数情况下，表达式和其对应的委托之间存在简单映射。 例如，由 `Expression<Func<int>>` 表示的表达式树将被转换为 `Func<int>` 类型的委托。 对于具有任

何返回类型和参数列表的 Lambda 表达式，存在这样的委托类型：该类型是由该 Lambda 表达式表示的可执行代码的目标类型。

System.Linq.Expressions.LambdaExpression 类型包含用于将表达式树转换为可执行代码的 LambdaExpression.Compile 和 LambdaExpression.CompileToMethod 成员。 `Compile` 方法创建委托。 `CompileToMethod` 方法通过表示表达式树的已编译输出的 IL 更新 System.Reflection.Emit.MethodBuilder 对象。

> **ⓘ 重要**
>
> `CompileToMethod` 仅在 .NET Framework 中可用，在 .NET Core 或 .NET 5 及更高版本中不可用。

还可以选择性地提供 System.Runtime.CompilerServices.DebugInfoGenerator，它接收生成的委托对象的符号调试信息。 `DebugInfoGenerator` 提供有关生成的委托的完整调试信息。

使用下面的代码将表达式转换为委托：

C#

```
Expression<Func<int>> add = () => 1 + 2;
var func = add.Compile(); // Create Delegate
var answer = func(); // Invoke Delegate
Console.WriteLine(answer);
```

下面的代码示例演示了在编译和执行表达式树时使用的具体类型。

C#

```
Expression<Func<int, bool>> expr = num => num < 5;

// Compiling the expression tree into a delegate.
Func<int, bool> result = expr.Compile();

// Invoking the delegate and writing the result to the console.
Console.WriteLine(result(4));

// Prints True.

// You can also use simplified syntax
// to compile and run an expression tree.
// The following line can replace two previous statements.
Console.WriteLine(expr.Compile()(4));

// Also prints True.
```

下面的代码示例演示如何通过创建 lambda 表达式并执行它来执行代表幂运算的表达式树。 示例最后显示幂运算的结果。

C#

```
// The expression tree to execute.
BinaryExpression be = Expression.Power(Expression.Constant(2d),
Expression.Constant(3d));

// Create a lambda expression.
Expression<Func<double>> le = Expression.Lambda<Func<double>>(be);

// Compile the lambda expression.
Func<double> compiledExpression = le.Compile();

// Execute the lambda expression.
double result = compiledExpression();

// Display the result.
Console.WriteLine(result);

// This code produces the following output:
// 8
```

执行和生存期

通过调用在调用 `LambdaExpression.Compile()` 时创建的委托来执行代码。 上述代码 `add.Compile()` 返回委托。 可以通过调用 `func()` 来调用该委托，它将执行代码。

该委托表示表达式树中的代码。 可以保留该委托的句柄并在稍后调用它。 不需要在每次想要执行表达式树所表示的代码时编译表达式树。 （请记住，表达式树是不可变的，并且在之后编译同一表达式树会创建执行相同代码的委托。）

⊗ 注意

不要通过避免不必要的编译调用来创建用于提高性能的任何更复杂的缓存机制。 比较两个任意的表达式树，确定如果它们表示相同的算法，是否也是一项耗时的操作。 通过避免对 `LambdaExpression.Compile()` 的任何额外调用所节省的计算时间可能多于执行代码（该代码用于确定两个不同的表达式树是否可产生相同的可执行代码）所花费的时间。

注意事项

将 lambda 表达式编译为委托并调用该委托是可对表达式树执行的最简单的操作之一。 但是，即使是执行这个简单的操作，也存在一些必须注意的事项。

Lambda 表达式将对表达式中引用的任何局部变量创建闭包。 必须保证作为委托的一部分的任何变量在调用 `Compile` 的位置处和执行结果委托时可用。 编译器确保变量处于范围内。 但是，如果表达式访问用于实现 `IDisposable` 的变量，则代码可能在表达式树仍保留有对象时释放该对象。

例如，此代码正常工作，因为 `int` 不实现 `IDisposable`：

C#

```
private static Func<int, int> CreateBoundFunc()
{
    var constant = 5; // constant is captured by the expression tree
    Expression<Func<int, int>> expression = (b) => constant + b;
    var rVal = expression.Compile();
    return rVal;
}
```

委托已捕获对局部变量 `constant` 的引用。 在稍后执行 `CreateBoundFunc` 返回的函数之后，可随时访问该变量。

但是，请考虑可实现 System.IDisposable 的以下（人为设计的）类：

C#

```
public class Resource : IDisposable
{
    private bool _isDisposed = false;
    public int Argument
    {
        get
        {
            if (!_isDisposed)
                return 5;
            else throw new ObjectDisposedException("Resource");
        }
    }

    public void Dispose()
    {
        _isDisposed = true;
    }
}
```

如果将它用于以下代码中所示的表达式中，则在执行 `Resource.Argument` 属性引用的代码时将出现 System.ObjectDisposedException：

C#

```
private static Func<int, int> CreateBoundResource()
{
    using (var constant = new Resource()) // constant is captured by the
expression tree
    {
        Expression<Func<int, int>> expression = (b) => constant.Argument +
b;
        var rVal = expression.Compile();
        return rVal;
    }
}
```

从此方法返回的委托已对释放了的 `constant` 对象闭包。 （它已被释放，因为它已在 `using` 语句中进行声明。）

现在，在执行从此方法返回的委托时，将在执行时引发 `ObjectDisposedException`。

出现表示编译时构造的运行时错误确实很奇怪，但这是使用表达式树时的正常现象。

此问题存在大量的排列，因此很难提供用于避免此问题的一般性指导。 定义表达式时，请谨慎访问局部变量，并且在创建通过公共 API 返回的表达式树时，谨慎访问当前对象（由 `this` 表示）中的状态。

表达式中的代码可能引用其他程序集中的方法或属性。 对表达式进行定义、编译或在调用生成的委托时，该程序集必须可访问。 在它不存在的情况下，将遇到 `ReferencedAssemblyNotFoundException`。

总结

可以编译表示 lambda 表达式的表达式树，以创建可执行的委托。 表达式树提供了一种机制，用于执行表达式树所表示的代码。

表达式树表示会为创建的任意给定构造执行的代码。 只要编译和执行代码的环境匹配创建表达式的环境，则一切将按预期进行。 如果未按预期进行，那么错误是可以预知的，并且将在使用表达式树的任何代码的第一批测试中捕获这些错误。

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

解释表达式

项目 • 2023/03/13

下列代码示例展示如何分解表示 Lambda 表达式 `num => num < 5` 的表达式树。

C#

```
// Add the following using directive to your code file:
// using System.Linq.Expressions;

// Create an expression tree.
Expression<Func<int, bool>> exprTree = num => num < 5;

// Decompose the expression tree.
ParameterExpression param = (ParameterExpression)exprTree.Parameters[0];
BinaryExpression operation = (BinaryExpression)exprTree.Body;
ParameterExpression left = (ParameterExpression)operation.Left;
ConstantExpression right = (ConstantExpression)operation.Right;

Console.WriteLine("Decomposed expression: {0} => {1} {2} {3}",
                  param.Name, left.Name, operation.NodeType, right.Value);

// This code produces the following output:

// Decomposed expression: num => num LessThan 5
```

现在，让我们编写一些代码来检查*表达式树*的结构。 表达式树中的每个节点是派生自 `Expression` 的类的对象。

该设计使得访问表达式树中的所有节点成为相对直接的递归操作。 常规策略是从根节点开始并确定它是哪种节点。

如果节点类型具有子级，则以递归方式访问该子级。 在每个子节点中，重复在根节点处使用的步骤：确定类型，且如果该类型具有子级，则访问每个子级。

检查不具有子级的表达式

让我们首先访问一个简单的表达式树中的每个节点。 下面是创建常数表达式然后检查其属性的代码：

C#

```
var constant = Expression.Constant(24, typeof(int));

Console.WriteLine($"This is a/an {constant.NodeType} expression type");
```

```
Console.WriteLine($"The type of the constant value is {constant.Type}");
Console.WriteLine($"The value of the constant value is {constant.Value}");
```

前面的代码打印以下输出：

输出

```
This is a/an Constant expression type
The type of the constant value is System.Int32
The value of the constant value is 24
```

现在，让我们来编写将检查此表达式的代码，并写出有关它的一些重要属性。

加法表达式

让我们从本节简介处的加法示例开始。

C#

```
Expression<Func<int>> sum = () => 1 + 2;
```

> ⓘ **备注**
>
> 请勿使用 `var` 声明此表达式树，因为委托的自然类型为 `Func<int>`，而不是 `Expression<Func<int>>`。

根节点是 `LambdaExpression`。 为了获得 `=>` 运算符右侧的有用代码，需要找到 `LambdaExpression` 的子级之一。 你将通过本部分中的所有表达式来实现此目的。 父节点确实有助于找到 `LambdaExpression` 的返回类型。

若要检查此表达式中的每个节点，需要以递归方式访问许多节点。 下面是一个简单的首次实现：

C#

```
Expression<Func<int, int, int>> addition = (a, b) => a + b;

Console.WriteLine($"This expression is a {addition.NodeType} expression
type");
Console.WriteLine($"The name of the lambda is {((addition.Name == null) ? "
<null>" : addition.Name)}");
Console.WriteLine($"The return type is {addition.ReturnType.ToString()}");
Console.WriteLine($"The expression has {addition.Parameters.Count}
arguments. They are:");
```

```
foreach (var argumentExpression in addition.Parameters)
{
    Console.WriteLine($"\tParameter Type:
{argumentExpression.Type.ToString()}, Name: {argumentExpression.Name}");
}

var additionBody = (BinaryExpression)addition.Body;
Console.WriteLine($"The body is a {additionBody.NodeType} expression");
Console.WriteLine($"The left side is a {additionBody.Left.NodeType}
expression");
var left = (ParameterExpression)additionBody.Left;
Console.WriteLine($"\tParameter Type: {left.Type.ToString()}, Name:
{left.Name}");
Console.WriteLine($"The right side is a {additionBody.Right.NodeType}
expression");
var right = (ParameterExpression)additionBody.Right;
Console.WriteLine($"\tParameter Type: {right.Type.ToString()}, Name:
{right.Name}");
```

此示例打印以下输出：

输出

```
This expression is a/an Lambda expression type
The name of the lambda is <null>
The return type is System.Int32
The expression has 2 arguments. They are:
        Parameter Type: System.Int32, Name: a
        Parameter Type: System.Int32, Name: b
The body is a/an Add expression
The left side is a Parameter expression
        Parameter Type: System.Int32, Name: a
The right side is a Parameter expression
        Parameter Type: System.Int32, Name: b
```

在前面的代码示例中，你会注意到很多重复。 让我们将其清理干净，并生成一个更加通用的表达式节点访问者。 这将要求编写递归算法。 任何节点都可能是具有子级的类型。 具有子级的任何节点都要求访问这些子级并确定该节点是什么。 下面是利用递归访问加法运算的已清理的版本：

C#

```
using System.Linq.Expressions;

namespace Visitors;
// Base Visitor class:
public abstract class Visitor
{
    private readonly Expression node;

    protected Visitor(Expression node) => this.node = node;
```

```
    public abstract void Visit(string prefix);

    public ExpressionType NodeType => node.NodeType;
    public static Visitor CreateFromExpression(Expression node) =>
        node.NodeType switch
        {
            ExpressionType.Constant => new
ConstantVisitor((ConstantExpression)node),
            ExpressionType.Lambda => new
LambdaVisitor((LambdaExpression)node),
            ExpressionType.Parameter => new
ParameterVisitor((ParameterExpression)node),
            ExpressionType.Add => new BinaryVisitor((BinaryExpression)node),
            _ => throw new NotImplementedException($"Node not processed yet:
{node.NodeType}"),
        };
}

// Lambda Visitor
public class LambdaVisitor : Visitor
{
    private readonly LambdaExpression node;
    public LambdaVisitor(LambdaExpression node) : base(node) => this.node =
node;

    public override void Visit(string prefix)
    {
        Console.WriteLine($"{prefix}This expression is a {NodeType}
expression type");
        Console.WriteLine($"{prefix}The name of the lambda is {((node.Name
== null) ? "<null>" : node.Name)}");
        Console.WriteLine($"{prefix}The return type is {node.ReturnType}");
        Console.WriteLine($"{prefix}The expression has
{node.Parameters.Count} argument(s). They are:");
        // Visit each parameter:
        foreach (var argumentExpression in node.Parameters)
        {
            var argumentVisitor = CreateFromExpression(argumentExpression);
            argumentVisitor.Visit(prefix + "\t");
        }
        Console.WriteLine($"{prefix}The expression body is:");
        // Visit the body:
        var bodyVisitor = CreateFromExpression(node.Body);
        bodyVisitor.Visit(prefix + "\t");
    }
}

// Binary Expression Visitor:
public class BinaryVisitor : Visitor
{
    private readonly BinaryExpression node;
    public BinaryVisitor(BinaryExpression node) : base(node) => this.node =
node;
```

```
    public override void Visit(string prefix)
    {
        Console.WriteLine($"{prefix}This binary expression is a {NodeType}
expression");
        var left = CreateFromExpression(node.Left);
        Console.WriteLine($"{prefix}The Left argument is:");
        left.Visit(prefix + "\t");
        var right = CreateFromExpression(node.Right);
        Console.WriteLine($"{prefix}The Right argument is:");
        right.Visit(prefix + "\t");
    }
}

// Parameter visitor:
public class ParameterVisitor : Visitor
{
    private readonly ParameterExpression node;
    public ParameterVisitor(ParameterExpression node) : base(node)
    {
        this.node = node;
    }

    public override void Visit(string prefix)
    {
        Console.WriteLine($"{prefix}This is an {NodeType} expression type");
        Console.WriteLine($"{prefix}Type: {node.Type}, Name: {node.Name},
ByRef: {node.IsByRef}");
    }
}

// Constant visitor:
public class ConstantVisitor : Visitor
{
    private readonly ConstantExpression node;
    public ConstantVisitor(ConstantExpression node) : base(node) =>
this.node = node;

    public override void Visit(string prefix)
    {
        Console.WriteLine($"{prefix}This is an {NodeType} expression type");
        Console.WriteLine($"{prefix}The type of the constant value is
{node.Type}");
        Console.WriteLine($"{prefix}The value of the constant value is
{node.Value}");
    }
}
```

此算法是访问任意 `LambdaExpression` 的算法的基础。 你创建的代码仅查找它可能遇到的表达式树节点组的一小部分。 但是，你仍可以从其结果中获益匪浅。 （遇到新的节点类型时，`Visitor.CreateFromExpression` 方法中的默认案例会将消息打印到错误控制台。如此，你便知道要添加新的表达式类型。）

在上面的加法表达式中运行此访问者时，将获得以下输出：

输出

```
This expression is a/an Lambda expression type
The name of the lambda is <null>
The return type is System.Int32
The expression has 2 argument(s). They are:
        This is an Parameter expression type
        Type: System.Int32, Name: a, ByRef: False
        This is an Parameter expression type
        Type: System.Int32, Name: b, ByRef: False
The expression body is:
        This binary expression is a Add expression
        The Left argument is:
                This is an Parameter expression type
                Type: System.Int32, Name: a, ByRef: False
        The Right argument is:
                This is an Parameter expression type
                Type: System.Int32, Name: b, ByRef: False
```

现在既已构建更通用的访问者实现，便可以访问和处理更多不同类型的表达式了。

具有更多操作数的加法表达式

让我们尝试更复杂的示例，但仍限制节点类型仅为加法：

C#

```
Expression<Func<int>> sum = () => 1 + 2 + 3 + 4;
```

在访问者算法上运行这些示例之前，请尝试思考可能的输出是什么。 请记住，`+` 运算符是*二元运算符*：它必须具有两个子级，分别表示左右操作数。 有几种可行的方法来构造可能正确的树：

C#

```
Expression<Func<int>> sum1 = () => 1 + (2 + (3 + 4));
Expression<Func<int>> sum2 = () => ((1 + 2) + 3) + 4;

Expression<Func<int>> sum3 = () => (1 + 2) + (3 + 4);
Expression<Func<int>> sum4 = () => 1 + ((2 + 3) + 4);
Expression<Func<int>> sum5 = () => (1 + (2 + 3)) + 4;
```

可以看到可能的答案分为两种，以便着重于最有可能正确的答案。 第一种表示*右结合*表达式。 第二种表示*左结合*表达式。 这两种格式的优点是，格式可以缩放为任意数量的加

法表达式。

如果确实通过该访问者运行此表达式，则会看到此输出，它验证简单的加法表达式是否为左结合。

为了运行此示例并查看完整的表达式树，你对源表达式树进行了一次更改。 当表达式树包含所有常量时，所得到的树仅包含 `10` 的常量值。 编译器执行所有加法运算，并将表达式缩减为其最简单的形式。 只需在表达式中添加一个变量即可看到原始的树：

C#

```
Expression<Func<int, int>> sum = (a) => 1 + a + 3 + 4;
```

创建可得出此总和的访问者并运行该访问者，会看到以下输出：

输出

```
This expression is a/an Lambda expression type
The name of the lambda is <null>
The return type is System.Int32
The expression has 1 argument(s). They are:
        This is an Parameter expression type
        Type: System.Int32, Name: a, ByRef: False
The expression body is:
        This binary expression is a Add expression
        The Left argument is:
                This binary expression is a Add expression
                The Left argument is:
                        This binary expression is a Add expression
                        The Left argument is:
                                This is an Constant expression type
                                The type of the constant value is
System.Int32
                                The value of the constant value is 1
                        The Right argument is:
                                This is an Parameter expression type
                                Type: System.Int32, Name: a, ByRef: False
                The Right argument is:
                        This is an Constant expression type
                        The type of the constant value is System.Int32
                        The value of the constant value is 3
        The Right argument is:
                This is an Constant expression type
                The type of the constant value is System.Int32
                The value of the constant value is 4
```

可以通过访问者代码运行任何其他示例，并查看其表示的树。 下面是上述 `sum3` 表达式（使用附加参数来阻止编译器计算常量）的一个示例：

C#

```
Expression<Func<int, int, int>> sum3 = (a, b) => (1 + a) + (3 + b);
```

下面是访问者的输出：

输出

```
This expression is a/an Lambda expression type
The name of the lambda is <null>
The return type is System.Int32
The expression has 2 argument(s). They are:
        This is an Parameter expression type
        Type: System.Int32, Name: a, ByRef: False
        This is an Parameter expression type
        Type: System.Int32, Name: b, ByRef: False
The expression body is:
        This binary expression is a Add expression
        The Left argument is:
                This binary expression is a Add expression
                The Left argument is:
                        This is an Constant expression type
                        The type of the constant value is System.Int32
                        The value of the constant value is 1
                The Right argument is:
                        This is an Parameter expression type
                        Type: System.Int32, Name: a, ByRef: False
        The Right argument is:
                This binary expression is a Add expression
                The Left argument is:
                        This is an Constant expression type
                        The type of the constant value is System.Int32
                        The value of the constant value is 3
                The Right argument is:
                        This is an Parameter expression type
                        Type: System.Int32, Name: b, ByRef: False
```

请注意，括号不是输出的一部分。 表达式树中不存在表示输入表达式中的括号的节点。 表达式树的结构包含传达优先级所需的所有信息。

扩展此示例

此示例仅处理最基本的表达式树。 在本部分中看到的代码仅处理常量整数和二进制 `+` 运算符。 作为最后一个示例，让我们更新访问者以处理更加复杂的表达式。 让我们将它用于以下阶乘表达式：

C#

```
Expression<Func<int, int>> factorial = (n) =>
    n == 0 ?
    1 :
    Enumerable.Range(1, n).Aggregate((product, factor) => product * factor);
```

此代码表示数学*阶乘*函数的一个可能的实现。 编写此代码的方式强调了通过将 Lambda 表达式分配到表达式来生成表达式树的两个限制。 首先，Lambda 语句是不允许的。 这意味着无法使用循环、块、if/else 语句和 C# 中常用的其他控件结构。 你只能使用表达式。 其次，不能以递归方式调用同一表达式。 如果该表达式已是一个委托，则可以通过递归方式进行调用，但不能在其表达式树的形式中调用它。 在有关生成表达式树的部分中，了解克服这些限制的技巧。

在此表达式中，你遇到所有这些类型的节点：

1. Equal（二进制表达式）
2. Multiply（二进制表达式）
3. Conditional（`? :` 表达式）
4. 方法调用表达式（调用 `Range()` 和 `Aggregate()`）

修改访问者算法的其中一个方法是持续执行它，并在每次到达 `default` 子句时编写节点类型。 经过几次迭代之后，已看到每个可能的节点。 这样便万事俱备了。 结果类似于：

C#

```
public static Visitor CreateFromExpression(Expression node) =>
    node.NodeType switch
    {
        ExpressionType.Constant    => new
ConstantVisitor((ConstantExpression)node),
        ExpressionType.Lambda      => new
LambdaVisitor((LambdaExpression)node),
        ExpressionType.Parameter   => new
ParameterVisitor((ParameterExpression)node),
        ExpressionType.Add         => new
BinaryVisitor((BinaryExpression)node),
        ExpressionType.Equal       => new
BinaryVisitor((BinaryExpression)node),
        ExpressionType.Multiply    => new BinaryVisitor((BinaryExpression)
node),
        ExpressionType.Conditional => new
ConditionalVisitor((ConditionalExpression) node),
        ExpressionType.Call        => new
MethodCallVisitor((MethodCallExpression) node),
        _ => throw new NotImplementedException($"Node not processed yet:
{node.NodeType}"),
    };
```

`ConditionalVisitor` 和 `MethodCallVisitor` 处理这两个节点：

C#

```
public class ConditionalVisitor : Visitor
{
    private readonly ConditionalExpression node;
    public ConditionalVisitor(ConditionalExpression node) : base(node)
    {
        this.node = node;
    }

    public override void Visit(string prefix)
    {
        Console.WriteLine($"{prefix}This expression is a {NodeType}
expression");
        var testVisitor = Visitor.CreateFromExpression(node.Test);
        Console.WriteLine($"{prefix}The Test for this expression is:");
        testVisitor.Visit(prefix + "\t");
        var trueVisitor = Visitor.CreateFromExpression(node.IfTrue);
        Console.WriteLine($"{prefix}The True clause for this expression
is:");
        trueVisitor.Visit(prefix + "\t");
        var falseVisitor = Visitor.CreateFromExpression(node.IfFalse);
        Console.WriteLine($"{prefix}The False clause for this expression
is:");
        falseVisitor.Visit(prefix + "\t");
    }
}

public class MethodCallVisitor : Visitor
{
    private readonly MethodCallExpression node;
    public MethodCallVisitor(MethodCallExpression node) : base(node)
    {
        this.node = node;
    }

    public override void Visit(string prefix)
    {
        Console.WriteLine($"{prefix}This expression is a {NodeType}
expression");
        if (node.Object == null)
            Console.WriteLine($"{prefix}This is a static method call");
        else
        {
            Console.WriteLine($"{prefix}The receiver (this) is:");
            var receiverVisitor = Visitor.CreateFromExpression(node.Object);
            receiverVisitor.Visit(prefix + "\t");
        }

        var methodInfo = node.Method;
        Console.WriteLine($"{prefix}The method name is
{methodInfo.DeclaringType}.{methodInfo.Name}");
```

```
        // There is more here, like generic arguments, and so on.
        Console.WriteLine($"{prefix}The Arguments are:");
        foreach (var arg in node.Arguments)
        {
            var argVisitor = Visitor.CreateFromExpression(arg);
            argVisitor.Visit(prefix + "\t");
        }
    }
}
```

且表达式树的输出为：

输出

```
This expression is a/an Lambda expression type
The name of the lambda is <null>
The return type is System.Int32
The expression has 1 argument(s). They are:
        This is an Parameter expression type
        Type: System.Int32, Name: n, ByRef: False
The expression body is:
        This expression is a Conditional expression
        The Test for this expression is:
                This binary expression is a Equal expression
                The Left argument is:
                        This is an Parameter expression type
                        Type: System.Int32, Name: n, ByRef: False
                The Right argument is:
                        This is an Constant expression type
                        The type of the constant value is System.Int32
                        The value of the constant value is 0
        The True clause for this expression is:
                This is an Constant expression type
                The type of the constant value is System.Int32
                The value of the constant value is 1
        The False clause for this expression is:
                This expression is a Call expression
                This is a static method call
                The method name is System.Linq.Enumerable.Aggregate
                The Arguments are:
                        This expression is a Call expression
                        This is a static method call
                        The method name is System.Linq.Enumerable.Range
                        The Arguments are:
                                This is an Constant expression type
                                The type of the constant value is
System.Int32
                                The value of the constant value is 1
                                This is an Parameter expression type
                                Type: System.Int32, Name: n, ByRef: False
                        This expression is a Lambda expression type
                        The name of the lambda is <null>
                        The return type is System.Int32
```

```
                        The expression has 2 arguments. They are:
                                This is an Parameter expression type
                                Type: System.Int32, Name: product, ByRef:
False
                                This is an Parameter expression type
                                Type: System.Int32, Name: factor, ByRef:
False
                        The expression body is:
                                This binary expression is a Multiply
expression
                                The Left argument is:
                                        This is an Parameter expression type
                                        Type: System.Int32, Name: product,
ByRef: False
                                The Right argument is:
                                        This is an Parameter expression type
                                        Type: System.Int32, Name: factor,
ByRef: False
```

扩展示例库

本部分中的示例演示访问和检查表达式树中的节点的核心技术。 它简化了你将遇到的节点类型，以专注于访问表达式树中的节点的核心任务。

首先，访问者只处理整数常量。 常量值可以是任何其他数值类型，且 C# 语言支持这些类型之间的转换和提升。 此代码的更可靠版本可反映所有这些功能。

即使最后一个示例也只可识别可能的节点类型的一部分。 你仍可以向它添加许多导致失败的表达式。 完整的实现包含在名为 ExpressionVisitor 的 .NET Standard 中，且可以处理所有可能的节点类型。

最后，在本文中所使用的库是为演示和学习目的而生成。 它未进行优化。 它使结构清晰，并强调用于访问节点和对此进行分析的技术。

即使存在这些限制，在编写阅读和理解表达式树的算法这方面应当是没有问题的。

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

生成表达式树

项目 • 2023/03/13

C# 编译器创建了迄今为止你看到的所有表达式树。 你创建了一个 Lambda 表达式，将其分配给一个类型为 `Expression<Func<T>>` 或某种相似类型的变量。 很多情况下，需要在运行时在内存中生成一个表达式。

表达式树是不可变的。 不可变意味着必须以从叶到根的方式生成表达式树。 用于生成表达式树的 API 体现了这一点：用于生成节点的方法将其所有子级用作自变量。 让我们通过几个示例来了解相关技巧。

创建节点

你将使用在这些部分中一直使用的加法表达式开始：

```
C#

Expression<Func<int>> sum = () => 1 + 2;
```

若要构造该表达式树，需要先构造叶节点。 叶节点是常量。 使用 Constant 方法创建节点：

```
C#

var one = Expression.Constant(1, typeof(int));
var two = Expression.Constant(2, typeof(int));
```

接下来，将生成加法表达式：

```
C#

var addition = Expression.Add(one, two);
```

生成加法表达式后，就可以创建 Lambda 表达式：

```
C#

var lambda = Expression.Lambda(addition);
```

此 Lambda 表达式不包含任何自变量。 在本节的后续部分，你将了解如何将自变量映射到参数并生成更复杂的表达式。

对于此类表达式，可以将所有调用合并到单个语句中：

C#

```
var lambda2 = Expression.Lambda(
    Expression.Add(
        Expression.Constant(1, typeof(int)),
        Expression.Constant(2, typeof(int))
    )
);
```

生成树

上一节介绍了在内存中生成表达式树的基础知识。 更复杂的树通常意味着更多的节点类型，并且树中有更多的节点。 让我们再浏览一个示例，了解通常在创建表达式树时创建的其他两个节点类型：自变量节点和方法调用节点。 生成一个表达式树以创建此表达式：

C#

```
Expression<Func<double, double, double>> distanceCalc =
    (x, y) => Math.Sqrt(x * x + y * y);
```

首先，创建 `x` 和 `y` 的参数表达式：

C#

```
var xParameter = Expression.Parameter(typeof(double), "x");
var yParameter = Expression.Parameter(typeof(double), "y");
```

按照你所看到的模式创建乘法和加法表达式：

C#

```
var xSquared = Expression.Multiply(xParameter, xParameter);
var ySquared = Expression.Multiply(yParameter, yParameter);
var sum = Expression.Add(xSquared, ySquared);
```

接下来，需要为调用 `Math.Sqrt` 创建方法调用表达式。

C#

```
var sqrtMethod = typeof(Math).GetMethod("Sqrt", new[] { typeof(double) }) ??
throw new InvalidOperationException("Math.Sqrt not found!");
```

```
var distance = Expression.Call(sqrtMethod, sum);
```

如果未找到方法，`GetMethod` 调用可能会返回 `null`。 最有可能的是因为方法名称拼写错误。 否则，这可能意味着没有加载所需的程序集。 最后，将方法调用放入 Lambda 表达式，并确保定义 Lambda 表达式的自变量：

C#

```
var distanceLambda = Expression.Lambda(
    distance,
    xParameter,
    yParameter);
```

在这个更复杂的示例中，你看到了创建表达式树通常使用的其他几种技巧。

首先，在使用它们之前，需要创建表示参数或局部变量的对象。 创建这些对象后，可以在表达式树中任何需要的位置使用它们。

其次，需要使用反射 API 的一个子集来创建 System.Reflection.MethodInfo 对象，以便创建表达式树以访问该方法。 必须仅限于 .NET Core 平台上提供的反射 API 的子集。 同样，这些技术将扩展到其他表达式树。

深入了解代码生成

不仅限于使用这些 API 可以生成的代码。 但是，要生成的表达式树越复杂，代码就越难以管理和阅读。

让我们生成一个与此代码等效的表达式树：

C#

```
Func<int, int> factorialFunc = (n) =>
{
    var res = 1;
    while (n > 1)
    {
        res = res * n;
        n--;
    }
    return res;
};
```

前面的代码没有生成表达式树，只生成了委托。 使用 `Expression` 类不能生成语句 lambda。 下面是生成相同的功能所需的代码。 没有用于生成 `while` 循环的 API，需要生成一个包含条件测试的循环和一个用于中断循环的标签目标。

C#

```
var nArgument = Expression.Parameter(typeof(int), "n");
var result = Expression.Variable(typeof(int), "result");

// Creating a label that represents the return value
LabelTarget label = Expression.Label(typeof(int));

var initializeResult = Expression.Assign(result, Expression.Constant(1));

// This is the inner block that performs the multiplication,
// and decrements the value of 'n'
var block = Expression.Block(
    Expression.Assign(result,
        Expression.Multiply(result, nArgument)),
    Expression.PostDecrementAssign(nArgument)
);

// Creating a method body.
BlockExpression body = Expression.Block(
    new[] { result },
    initializeResult,
    Expression.Loop(
        Expression.IfThenElse(
            Expression.GreaterThan(nArgument, Expression.Constant(1)),
            block,
            Expression.Break(label, result)
        ),
        label
    )
);
```

用于生成阶乘函数的表达式树的代码相对更长、更复杂，它充满了标签和 break 语句以及在日常编码任务中想要避免的其他元素。

本部分编写了用于访问此表达式树中所有节点的代码，并编写了在此示例中创建的节点的相关信息。 可以在 dotnet/docs GitHub 存储库查看或下载示例代码。 生成并运行这些示例，自行动手试验。

将代码构造映射到表达式

下面的代码示例展示了使用 API 表示 Lambda 表达式 `num => num < 5` 的表达式树。

C#

```
// Manually build the expression tree for
// the lambda expression num => num < 5.
ParameterExpression numParam = Expression.Parameter(typeof(int), "num");
ConstantExpression five = Expression.Constant(5, typeof(int));
```

```
BinaryExpression numLessThanFive = Expression.LessThan(numParam, five);
Expression<Func<int, bool>> lambda1 =
    Expression.Lambda<Func<int, bool>>(
        numLessThanFive,
        new ParameterExpression[] { numParam });
```

表达式树 API 还支持赋值表达式和控制流表达式，比如循环、条件块和 `try-catch` 块等。 相对于通过 C# 编译器和 Lambda 表达式创建表达式树，还可利用 API 创建更加复杂的表达式树。 下列示例展示如何创建计算数字阶乘的表达式树。

C#

```
// Creating a parameter expression.
ParameterExpression value = Expression.Parameter(typeof(int), "value");

// Creating an expression to hold a local variable.
ParameterExpression result = Expression.Parameter(typeof(int), "result");

// Creating a label to jump to from a loop.
LabelTarget label = Expression.Label(typeof(int));

// Creating a method body.
BlockExpression block = Expression.Block(
    // Adding a local variable.
    new[] { result },
    // Assigning a constant to a local variable: result = 1
    Expression.Assign(result, Expression.Constant(1)),
        // Adding a loop.
        Expression.Loop(
           // Adding a conditional block into the loop.
           Expression.IfThenElse(
               // Condition: value > 1
               Expression.GreaterThan(value, Expression.Constant(1)),
               // If true: result *= value --
               Expression.MultiplyAssign(result,
                   Expression.PostDecrementAssign(value)),
               // If false, exit the loop and go to the label.
               Expression.Break(label, result)
           ),
       // Label to jump to.
       label
    )
);

// Compile and execute an expression tree.
int factorial = Expression.Lambda<Func<int, int>>(block, value).Compile()
(5);

Console.WriteLine(factorial);
// Prints 120.
```

有关详细信息，请参阅在 Visual Studio 2010 中使用表达式树生成动态方法，该方法也适用于 Visual Studio 的更高版本。

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

转换表达式树

项目 • 2023/03/13

本文介绍如何访问表达式树中的每个节点，同时生成该表达式树的已修改副本。 你将转换表达式树以了解算法，以便可以将其转换到另一个环境中。 你将更改已创建的算法。 这可能是为了添加日志记录、拦截方法调用并跟踪它们，或其他目的。

生成的用于转换表达式树的代码是你已看到的用于访问树中所有节点的代码的扩展。 转换表达式树时，会访问所有节点，并在访问它们的同时生成新树。 新树可包含对原始节点的引用或已放置在树中的新节点。

我们来访问一个表达式树，并创建一个具有一些替换节点的新树。 在本例中，我们将任何常数替换为其 10 倍大的常数。 要么就保持表达式树不变。 通过将常数节点替换为执行乘法运算的新节点来进行此替换，而不必读取常数的值并将其替换为新的常数。

此处，在找到常数节点后，创建一个新乘法节点（其子节点是原始常数和常数 `10`）：

C#

```
private static Expression ReplaceNodes(Expression original)
{
    if (original.NodeType == ExpressionType.Constant)
    {
        return Expression.Multiply(original, Expression.Constant(10));
    }
    else if (original.NodeType == ExpressionType.Add)
    {
        var binaryExpression = (BinaryExpression)original;
        return Expression.Add(
            ReplaceNodes(binaryExpression.Left),
            ReplaceNodes(binaryExpression.Right));
    }
    return original;
}
```

通过将原始节点替换为替代项来创建新树。 通过编译并执行替换的树对此更改进行验证。

C#

```
var one = Expression.Constant(1, typeof(int));
var two = Expression.Constant(2, typeof(int));
var addition = Expression.Add(one, two);
var sum = ReplaceNodes(addition);
var executableFunc = Expression.Lambda(sum);

var func = (Func<int>)executableFunc.Compile();
```

```
var answer = func();
Console.WriteLine(answer);
```

生成新树是两者的结合：访问现有树中的节点，和创建新节点并将其插入树中。 上一示例演示了表达式树不可变这一点的重要性。 请注意，在上面的代码中创建的新树混合了新创建的节点和现有树中的节点。 节点可以在这两个树中使用，因为现有树中的节点无法修改。 重用节点可显著提高内存效率。 相同的节点可能会在整个树或多个表达式树中遍历使用。 由于不能修改节点，因此可以在需要时随时重用相同的节点。

遍历并执行加法

让我们通过生成遍历加法节点的树并计算结果的第二个访问者来对新树进行验证。 对目前见到的访问者进行一些修改。 在此新版本中，访问者将返回到目前为止加法运算的部分总和。 对于常数表达式，该总和即为常数表达式的值。 对于加法表达式，遍历这些树后，其结果为左操作数和右操作数的总和。

C#

```
var one = Expression.Constant(1, typeof(int));
var two = Expression.Constant(2, typeof(int));
var three = Expression.Constant(3, typeof(int));
var four = Expression.Constant(4, typeof(int));
var addition = Expression.Add(one, two);
var add2 = Expression.Add(three, four);
var sum = Expression.Add(addition, add2);

// Declare the delegate, so you can call it
// from itself recursively:
Func<Expression, int> aggregate = null!;
// Aggregate, return constants, or the sum of the left and right operand.
// Major simplification: Assume every binary expression is an addition.
aggregate = (exp) =>
    exp.NodeType == ExpressionType.Constant ?
    (int)((ConstantExpression)exp).Value :
    aggregate(((BinaryExpression)exp).Left) +
aggregate(((BinaryExpression)exp).Right);

var theSum = aggregate(sum);
Console.WriteLine(theSum);
```

此处有相当多的代码，但这些概念易于理解。 此代码访问首次深度搜索后的子级。 当它遇到常数节点时，访问者将返回该常数的值。 访问者访问这两个子级之后，这些子级计算出了为该子树计算的总和。 加法节点现在可以计算其总和。 在访问了表达式树中的所有节点后，计算出了总和。 可以通过在调试器中运行示例并跟踪执行来跟踪执行。

让我们通过遍历树，来更轻松地跟踪如何分析节点以及如何计算总和。 下面是包含大量跟踪信息的聚合方法的更新版本：

C#

```
private static int Aggregate(Expression exp)
{
    if (exp.NodeType == ExpressionType.Constant)
    {
        var constantExp = (ConstantExpression)exp;
        Console.Error.WriteLine($"Found Constant: {constantExp.Value}");
        if (constantExp.Value is int value)
        {
            return value;
        }
        else
        {
            return 0;
        }
    }
    else if (exp.NodeType == ExpressionType.Add)
    {
        var addExp = (BinaryExpression)exp;
        Console.Error.WriteLine("Found Addition Expression");
        Console.Error.WriteLine("Computing Left node");
        var leftOperand = Aggregate(addExp.Left);
        Console.Error.WriteLine($"Left is: {leftOperand}");
        Console.Error.WriteLine("Computing Right node");
        var rightOperand = Aggregate(addExp.Right);
        Console.Error.WriteLine($"Right is: {rightOperand}");
        var sum = leftOperand + rightOperand;
        Console.Error.WriteLine($"Computed sum: {sum}");
        return sum;
    }
    else throw new NotSupportedException("Haven't written this yet");
}
```

在 `sum` 表达式中运行该版本将生成以下输出：

输出

```
10
Found Addition Expression
Computing Left node
Found Addition Expression
Computing Left node
Found Constant: 1
Left is: 1
Computing Right node
Found Constant: 2
Right is: 2
Computed sum: 3
```

```
Left is: 3
Computing Right node
Found Addition Expression
Computing Left node
Found Constant: 3
Left is: 3
Computing Right node
Found Constant: 4
Right is: 4
Computed sum: 7
Right is: 7
Computed sum: 10
10
```

跟踪输出，并在上面的代码中跟随。 应当能够看出代码如何在遍历树的同时访问代码和计算总和，并得出总和。

现在，让我们来看看另一个运行，其表达式由 `sum1` 给出：

C#

```
Expression<Func<int>> sum1 = () => 1 + (2 + (3 + 4));
```

下面是通过检查此表达式得到的输出：

输出

```
Found Addition Expression
Computing Left node
Found Constant: 1
Left is: 1
Computing Right node
Found Addition Expression
Computing Left node
Found Constant: 2
Left is: 2
Computing Right node
Found Addition Expression
Computing Left node
Found Constant: 3
Left is: 3
Computing Right node
Found Constant: 4
Right is: 4
Computed sum: 7
Right is: 7
Computed sum: 9
Right is: 9
Computed sum: 10
10
```

虽然最终结果是相同的，但树遍历有所不同。 节点的访问顺序不同，因为树是以首先发生的不同运算构造的。

创建修改后的副本

创建新的**控制台应用程序**项目。 为 `System.Linq.Expressions` 命名空间的文件添加 `using` 指令。 向项目中添加 `AndAlsoModifier` 类。

C#

```
public class AndAlsoModifier : ExpressionVisitor
{
    public Expression Modify(Expression expression)
    {
        return Visit(expression);
    }

    protected override Expression VisitBinary(BinaryExpression b)
    {
        if (b.NodeType == ExpressionType.AndAlso)
        {
            Expression left = this.Visit(b.Left);
            Expression right = this.Visit(b.Right);

            // Make this binary expression an OrElse operation instead of an
AndAlso operation.
            return Expression.MakeBinary(ExpressionType.OrElse, left, right,
b.IsLiftedToNull, b.Method);
        }

        return base.VisitBinary(b);
    }
}
```

此类继承 ExpressionVisitor 类，并且专用于修改表示条件 `AND` 运算的表达式。 它将运算从条件 `AND` 更改为条件 `OR`。 此类替代基类型的 VisitBinary 方法，因为条件 `AND` 表达式表示为二进制表达式。 在 `VisitBinary` 方法中，如果传递给它的表达式表示条件 `AND` 操作，那么代码将构造一个新的表达式，此表达式包含条件 `OR` 运算符，而不是条件 `AND` 运算符。 如果传递给 `VisitBinary` 的表达式不表示条件 `AND` 运算，那么此方法遵从基类实现。 基类方法构造类似于所传递的表达式树的节点，但是这些节点将子树替换为访问者以递归方式生成的表达式树。

为 `System.Linq.Expressions` 命名空间的文件添加 `using` 指令。 向 Program.cs 文件中的 `Main` 方法添加代码以创建表达式树，并将其传递给对其进行修改的方法。

C#

```
Expression<Func<string, bool>> expr = name => name.Length > 10 &&
name.StartsWith("G");
Console.WriteLine(expr);

AndAlsoModifier treeModifier = new AndAlsoModifier();
Expression modifiedExpr = treeModifier.Modify((Expression)expr);

Console.WriteLine(modifiedExpr);

/*  This code produces the following output:

    name => ((name.Length > 10) && name.StartsWith("G"))
    name => ((name.Length > 10) || name.StartsWith("G"))
*/
```

此代码创建的表达式中包含条件 `AND` 运算。 然后，此代码创建 `AndAlsoModifier` 类的实例，并将表达式传递给此类的 `Modify` 方法。 输出原始以及修改后的表达式树以显示更改。 编译并运行该应用程序。

了解更多

此示例演示了要生成的用于遍历和解释表达式树表示的算法的代码的一小部分。 有关生成将表达式树转换为其他语言的通用库的信息，请阅读 Matt Warren 的这一系列。 它详细介绍了如何转换可能在表达式树中找到的任意代码。

你现在已经见识到了表达式树的真正强大功能。 你可以检查一组代码、对该代码进行任意更改，并执行更改后的版本。 由于表达式树是不可变的，你可以通过使用现有树的组件创建新树。 重用节点可使创建修改的表达式树所需的内存量最小。

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.

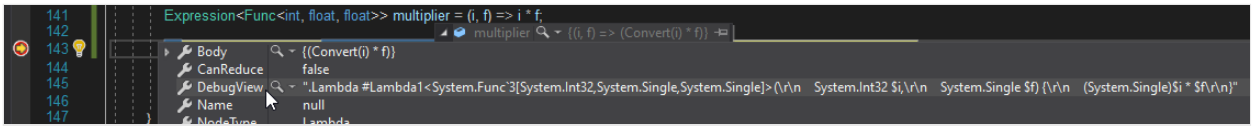
Open a documentation issue

Provide product feedback

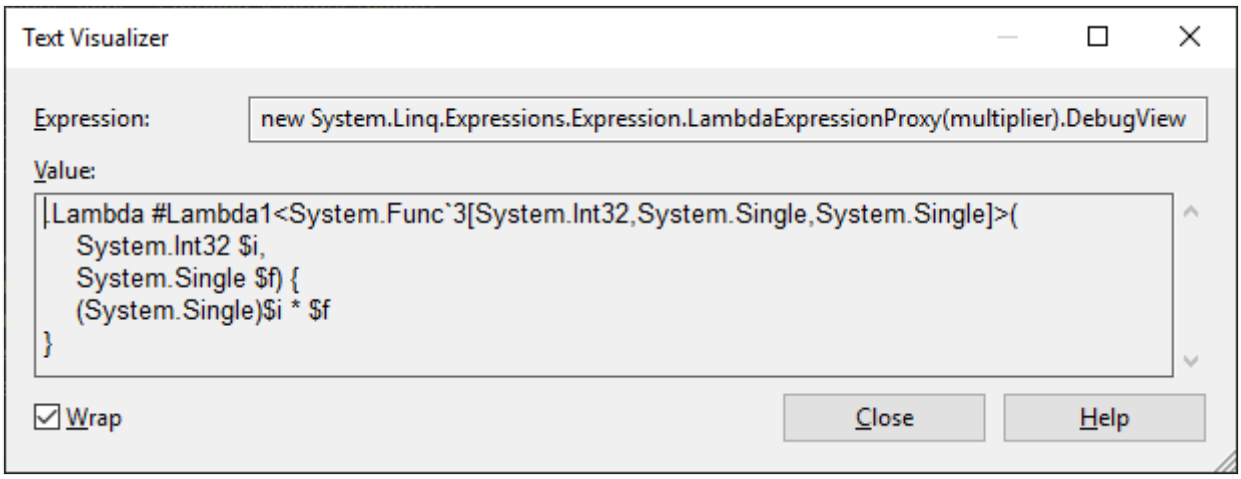
在 Visual Studio 中调试表达式树

项目 • 2024/03/03

可以在调试应用程序时分析表达式树的结构和内容。 要快速了解表达式树结构，可以使用 `DebugView` 属性，该属性`DebugView`表示表达式树。 `DebugView` 仅在调试模式下可用。

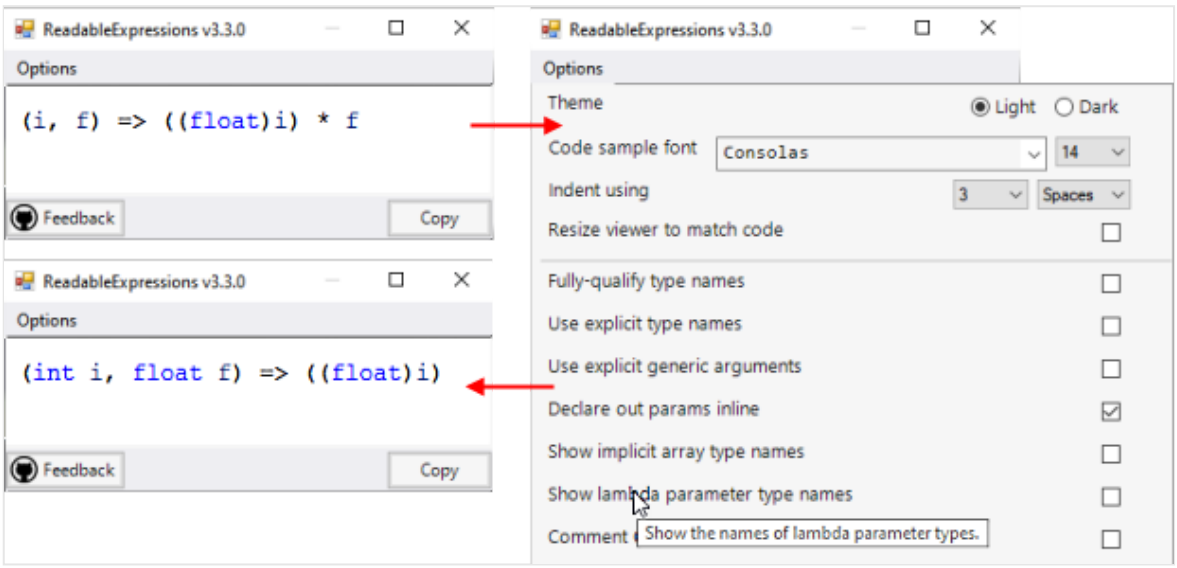


由于 `DebugView` 是一个字符串，因此可以使用`DebugView`在多行中进行查看，方法是从 标签旁边的放大镜图标中选择“文本可视化工具”。

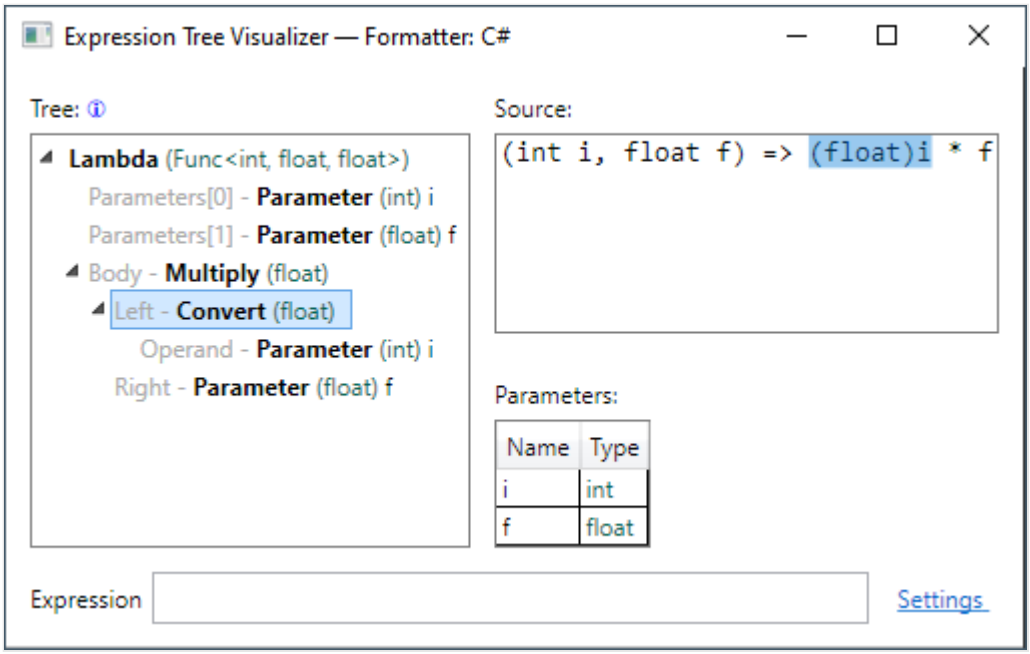


或者，可以为表达式树安装和使用自定义可视化工具，例如：

- 可读表达式 （MIT 许可证，可以从 Visual Studio Marketplace 中获取）使用各种呈现选项将表达式树呈现为可设置主题的 C# 代码：

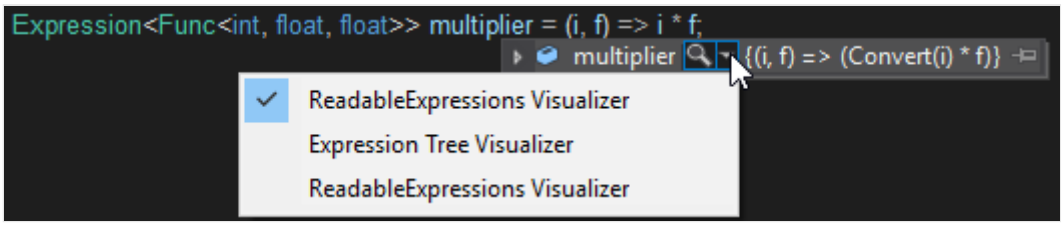


- 表达式树可视化工具 （MIT 许可证）提供表达式树及其各个节点的树视图：



打开表达式树的可视化工具

选择"数据提示"、"监视"窗口、"自动"窗口或"本地"窗口中表达式树旁边显示的放大镜图标。 显示可用的可视化工具列表：



选择要使用的可视化工具。

请参阅

- 在 Visual Studio 中进行调试
- 创建自定义可视化工具
- DebugView 语法

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 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

DebugView 语法

项目 • 2023/03/13

DebugView 属性（仅在调试时可用）提供表达式树的字符串呈现。 大部分语法都相当容易理解；特殊情况将在以下部分中介绍。

每个示例都后跟块注释，其中包含 DebugView。

ParameterExpression

ParameterExpression 变量名称的开头显示有 `$` 符号。

如果参数没有名称，则会为其分配一个自动生成的名称，例如 `$var1` 或 `$var2`。

C#

```
ParameterExpression numParam =  Expression.Parameter(typeof(int), "num");
/*
    $num
*/

ParameterExpression numParam =  Expression.Parameter(typeof(int));
/*
    $var1
*/
```

ConstantExpression

对于表示整数值、字符串和 `null` 的 ConstantExpression 对象，将显示常数的值。

对于使用标准后缀作为 C# 原义字符的数值类型，将后缀添加到值。 下表显示与各种数值类型关联的后缀。

类型	关键字	Suffix
System.UInt32	uint	U
System.Int64	long	L
System.UInt64	ulong	UL
System.Double	double	D
System.Single	float	F

类型	关键字	Suffix
System.Decimal	decimal	M

C#

```
int num = 10;
ConstantExpression expr = Expression.Constant(num);
/*
    10
*/

double num = 10;
ConstantExpression expr = Expression.Constant(num);
/*
    10D
*/
```

BlockExpression

如果 BlockExpression 对象的类型与块中最后一个表达式的类型不同，则该类型将显示在尖括号（`<` 和 `>`）内。 否则，将不显示 BlockExpression 对象的类型。

C#

```
BlockExpression block = Expression.Block(Expression.Constant("test"));
/*
    .Block() {
        "test"
    }
*/

BlockExpression block =  Expression.Block(typeof(Object),
Expression.Constant("test"));
/*
    .Block<System.Object>() {
        "test"
    }
*/
```

LambdaExpression

显示 LambdaExpression 对象及其委托类型。

如果 Lambda 表达式没有名称，则会为它分配一个自动生成的名称，例如 `#Lambda1` 或 `#Lambda2`。

C#

```
LambdaExpression lambda =  Expression.Lambda<Func<int>>
(Expression.Constant(1));
/*
    .Lambda #Lambda1<System.Func'1[System.Int32]>() {
        1
    }
*/

LambdaExpression lambda =  Expression.Lambda<Func<int>>
(Expression.Constant(1), "SampleLambda", null);
/*
    .Lambda #SampleLambda<System.Func'1[System.Int32]>() {
        1
    }
*/
```

LabelExpression

如果指定 LabelExpression 对象的默认值，则在 LabelTarget 对象之前显示此值。

`.Label` 令牌指示标签的开头。 `.LabelTarget` 令牌指示要跳转到的目标的目的地。

如果标签没有名称，则会为它分配一个自动生成的名称，例如 `#Label1` 或 `#Label2`。

C#

```
LabelTarget target = Expression.Label(typeof(int), "SampleLabel");
BlockExpression block = Expression.Block(
    Expression.Goto(target, Expression.Constant(0)),
    Expression.Label(target, Expression.Constant(-1))
);
/*
    .Block() {
        .Goto SampleLabel { 0 };
        .Label
            -1
        .LabelTarget SampleLabel:
    }
*/

LabelTarget target = Expression.Label();
BlockExpression block = Expression.Block(
    Expression.Goto(target),
    Expression.Label(target)
);
/*
    .Block() {
        .Goto #Label1 { };
```

```
        .Label
        .LabelTarget #Label1:
    }
*/
```

Checked 运算符

Checked 运算符在运算符前面显示 `#` 符号。 例如，checked 加号显示为 `#+`。

C#

```
Expression expr = Expression.AddChecked( Expression.Constant(1),
Expression.Constant(2));
/*
    1 #+ 2
*/

Expression expr = Expression.ConvertChecked( Expression.Constant(10.0),
typeof(int));
/*
    #(System.Int32)10D
*/
```


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

System.Linq.Expressions.Expression.Add 方法

项目 • 2024/02/01

本文提供了此 API 参考文档的补充说明。

该方法 Add 返回一个 BinaryExpression 属性 Method 设置为实现方法。 该 Type 属性设置为节点的类型。 如果提升节点，则IsLifted两者`true`均为IsLiftedToNull属性。 否则，它们是 `false`。 Conversion 属性为 `null`。

以下信息描述了实现方法、节点类型以及节点是否已提升。

实现方法

以下规则确定操作的所选实现方法：

- Type如果任`left`一类型的属性或`right`表示重载加法运算符的用户定义类型，则MethodInfo表示该方法是实现方法。
- 否则，如果 `left`为 .键入和 `right`。类型为数值类型，实现方法是 `null`。

节点类型和提升与非提升

如果实现方法不是 `null`：

- 如果 `left`。键入和 `right`。类型可分配给实现方法的相应参数类型，不会解除节点。 节点的类型是实现方法的返回类型。
- 如果满足以下两个条件，则会提升节点，节点的类型是与实现方法的返回类型相对应的可为 null 类型：
  - `left`.键入和 `right`。类型都是至少一个可为 null 的值类型，相应的不可为 null 类型等于实现方法的相应参数类型。
  - 实现方法的返回类型是不可为 null 的值类型。

如果实现方法是 `null`：

- 如果 `left`。键入和 `right`。类型都是不可为 null 的，不会解除节点。 节点的类型是预定义加法运算符的结果类型。
- 如果 `left`。键入和 `right`。类型都是可为 null 的，将提升节点。 节点的类型是与预定义加法运算符的结果类型相对应的可为 null 类型。

System.Linq.Expressions.BinaryExpression 类

项目 • 2024/01/11

本文提供了此 API 参考文档的补充说明。

该 BinaryExpression 类表示具有二进制运算符的表达式。

下表汇总了工厂方法，这些方法可用于创建 BinaryExpression 具有由属性表示 NodeType 的特定节点类型的工厂方法。 每个表都包含特定类运算的信息，例如算术或按位运算。

二进制算术运算

展开表

节点类型	Factory 方法
Add	Add
AddChecked	AddChecked
Divide	Divide
Modulo	Modulo
Multiply	Multiply
MultiplyChecked	MultiplyChecked
Power	Power
Subtract	Subtract
SubtractChecked	SubtractChecked

按位操作

展开表

节点类型	Factory 方法
And	And
Or	Or

节点类型	Factory 方法
ExclusiveOr	ExclusiveOr

Shift 操作

展开表

节点类型	Factory 方法
LeftShift	LeftShift
RightShift	RightShift

条件布尔运算

展开表

节点类型	Factory 方法
AndAlso	AndAlso
OrElse	OrElse

比较运算

展开表

节点类型	Factory 方法
Equal	Equal
NotEqual	NotEqual
GreaterThanOrEqual	GreaterThanOrEqual
GreaterThan	GreaterThan
LessThan	LessThan
LessThanOrEqual	LessThanOrEqual

合并操作

展开表

节点类型	Factory 方法
Coalesce	Coalesce

数组索引操作

展开表

节点类型	Factory 方法
ArrayIndex	ArrayIndex

此外， MakeBinary 方法还可用于创建一个 BinaryExpression。 这些工厂方法可用于创建 BinaryExpression 表示二进制操作的任何节点类型。 这些方法的参数类型 NodeType 指定所需的节点类型。

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

Interoperability Overview

Article • 02/25/2023

Interoperability enables you to preserve and take advantage of existing investments in unmanaged code. Code that runs under the control of the common language runtime (CLR) is *managed code*, and code that runs outside the CLR is *unmanaged code*. COM, COM+, C++ components, ActiveX components, and Microsoft Windows API are examples of unmanaged code.

.NET enables interoperability with unmanaged code through platform invoke services, the System.Runtime.InteropServices namespace, C++ interoperability, and COM interoperability (COM interop).

Platform Invoke

Platform invoke is a service that enables managed code to call unmanaged functions implemented in dynamic link libraries (DLLs), such as the Microsoft Windows API. It locates and invokes an exported function and marshals its arguments (integers, strings, arrays, structures, and so on) across the interoperation boundary as needed.

For more information, see Consuming Unmanaged DLL Functions and How to use platform invoke to play a WAV file.

> **Note**
>
> The Common Language Runtime (CLR) manages access to system resources. Calling unmanaged code that is outside the CLR bypasses this security mechanism, and therefore presents a security risk. For example, unmanaged code might call resources in unmanaged code directly, bypassing CLR security mechanisms. For more information, see Security in .NET.

C++ Interop

You can use C++ interop, also known as It Just Works (IJW), to wrap a native C++ class. C++ interop enables code authored in C# or another .NET language to access it. You write C++ code to wrap a native DLL or COM component. Unlike other .NET languages, Visual C++ has interoperability support that enables managed and unmanaged code in the same application and even in the same file. You then build the C++ code by using the **/clr** compiler switch to produce a managed assembly. Finally, you add a reference to

the assembly in your C# project and use the wrapped objects just as you would use other managed classes.

Exposing COM Components to C#

You can consume a COM component from a C# project. The general steps are as follows:

1. Locate a COM component to use and register it. Use regsvr32.exe to register or un–register a COM DLL.
2. Add to the project a reference to the COM component or type library. When you add the reference, Visual Studio uses the Tlbimp.exe (Type Library Importer), which takes a type library as input, to output a .NET interop assembly. The assembly, also named a runtime callable wrapper (RCW), contains managed classes and interfaces that wrap the COM classes and interfaces that are in the type library. Visual Studio adds to the project a reference to the generated assembly.
3. Create an instance of a class defined in the RCW. Creating an instance of that class creates an instance of the COM object.
4. Use the object just as you use other managed objects. When the object is reclaimed by garbage collection, the instance of the COM object is also released from memory.

For more information, see Exposing COM Components to the .NET Framework.

Exposing C# to COM

COM clients can consume C# types that have been correctly exposed. The basic steps to expose C# types are as follows:

1. Add interop attributes in the C# project. You can make an assembly COM visible by modifying C# project properties. For more information, see Assembly Information Dialog Box.
2. Generate a COM type library and register it for COM usage. You can modify C# project properties to automatically register the C# assembly for COM interop. Visual Studio uses the Regasm.exe (Assembly Registration Tool), using the `/tlb` command-line switch, which takes a managed assembly as input, to generate a type library. This type library describes the `public` types in the assembly and adds registry entries so that COM clients can create managed classes.

For more information, see Exposing .NET Framework Components to COM and Example COM Class.

See also

- Improving Interop Performance
- Introduction to Interoperability between COM and .NET
- Introduction to COM Interop in Visual Basic
- Marshaling between Managed and Unmanaged Code
- Interoperating with Unmanaged Code
- Registering Assemblies with COM

COM 类示例

项目 • 2023/05/10

下面的代码是将公开为 COM 对象的类的示例。 在将此代码放置在 .cs 文件中并添加到项目后，将“注册 COM 互操作”属性设置为“True”。 有关详细信息，请参阅如何：注册 COM 互操作组件。

对 COM 公开 C# 对象需要声明类接口、“事件接口”（如有必要）和类本身。 类成员必须遵循这些规则才能显示在 COM 中：

- 类必须是公开的。
- 属性、方法和事件必须是公开的。
- 必须在类接口上声明属性和方法。
- 必须在事件接口中声明事件。

该类中未在这些接口中声明的其他公共成员对 COM 不可见，但它们对其他 .NET 对象可见。 若要对 COM 公开属性和方法，则必须在类接口上声明这些属性和方法，将它们标记为 `DispId` 属性，并在类中实现它们。 你在接口中声明的成员的顺序是用于 COM vtable 的顺序。 若要从类中公开事件，则必须在事件接口上声明这些事件并将其标记为 `DispId` 属性。 此类不应实现此接口。

此类实现此类接口；它可以实现多个接口，但第一个实现是默认类接口。 在此处实现向 COM 公开的方法和属性。 它们必须是公共的，并且必须匹配类接口中的声明。 此外，在此处声明此类引发的事件。 它们必须是公共的，并且必须匹配事件接口中的声明。

示例

C#

```
using System.Runtime.InteropServices;

namespace project_name
{
    [Guid("EAA4976A-45C3-4BC5-BC0B-E474F4C3C83F")]
    public interface ComClass1Interface
    {
    }

    [Guid("7BD20046-DF8C-44A6-8F6B-687FAA26FA71"),
        InterfaceType(ComInterfaceType.InterfaceIsIDispatch)]
    public interface ComClass1Events
    {
    }

    [Guid("0D53A3E8-E51A-49C7-944E-E72A2064F938"),
```

```
        ClassInterface(ClassInterfaceType.None),
        ComSourceInterfaces(typeof(ComClass1Events))]
    public class ComClass1 : ComClass1Interface
    {
    }
}
```

另请参阅

- 互操作性
- “项目设计器”->“生成”页 (C#)

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

演练：用 C# 进行 Office 编程

项目 • 2023/03/09

C# 提供了改进 Microsoft Office 编程的功能。 有用的 C# 功能包括命名参数和可选参数以及类型为 `dynamic` 的返回值。 在 COM 编程中，可以省略 `ref` 关键字并获得索引属性的访问权限。

两种语言都支持嵌入类型信息，从而允许在不向用户的计算机部署主互操作程序集 (PIA) 的情况下部署与 COM 组件交互的程序集。 有关详细信息，请参见演练：嵌入托管程序集中的类型。

本演练演示 Office 编程上下文中的这些功能，但其中许多功能在常规编程中也极为有用。 本演练将使用 Excel 外接应用程序创建 Excel 工作簿。 然后，将创建包含工作簿链接的 Word 文档。 最后，将介绍如何启用和禁用 PIA 依赖项。

> **ⓘ 重要**
>
> **VSTO (Visual Studio Tools for Office)** 依赖于 **.NET Framework**。 COM 加载项也可以使用 .NET Framework 编写。 不能使用 **.NET Core 和 .NET 5+**（最新版本的 .NET）创建 Office 加载项。 这是因为 .NET Core/.NET 5+ 无法在同一进程中与 .NET Framework 协同工作，并可能导致加载项加载失败。 可以继续使用 .NET Framework 编写适用于 Office 的 VSTO 和 COM 加载项。 Microsoft 不会更新 VSTO 或 COM 加载项平台以使用 .NET Core 或 .NET 5+。 可以利用 .NET Core 和 .NET 5+（包括 ASP.NET Core）创建 **Office Web 加载项**的服务器端。

先决条件

若要完成本演练，计算机上必须安装 Microsoft Office Excel 和 Microsoft Office Word。

> **ⓘ 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。

设置 Excel 加载项应用程序

1. 启动 Visual Studio。

2. 在 “**文件**” 菜单上，指向 “**新建**”，然后选择 “**项目**”。
3. 在“安装的模板”窗格中，展开“C#”，再展开“Office”，然后选择 Office 产品的版本年份。
4. 在“模板”窗格中，选择“Excel <version> 加载项”。
5. 查看“模板”窗格的顶部，确保“.NET Framework 4”或更高版本出现在“目标框架”框中。
6. 如果需要，在“名称”框中键入项目的名称。
7. 选择“确定”。
8. 新项目将出现在“解决方案资源管理器”中。

添加引用

1. 在“解决方案资源管理器”中，右键单击你的项目名称，然后选择“添加引用”。 此时会显示“添加引用”对话框。
2. 在“程序集”选项卡上，在“组件名称”列表中选择“Microsoft.Office.Interop.Excel”版本 `<version>.0.0.0`（有关 Office 产品版本号的键，请参阅 Microsoft 版本），然后按住 Ctrl 键并选择“Microsoft.Office.Interop.Word”，`version <version>.0.0.0`。 如果未看到程序集，可能需要安装它们（请参阅如何：安装 Office 主互操作程序集）。
3. 选择“确定”。

添加必要的 Imports 语句或 using 指令

在“解决方案资源管理器”中，右键单击“ThisAddIn.cs”文件，然后选择“查看代码”。 将下列 `using` 指令 (C#) 添加到代码文件的顶部（如果它们尚不存在）。

C#

```
using System.Collections.Generic;
using Excel = Microsoft.Office.Interop.Excel;
using Word = Microsoft.Office.Interop.Word;
```

创建银行帐户列表

在“解决方案资源管理器”中，右键单击你的项目名称，选择“添加”，然后选择“类”。 将类命名为 Account.cs。 选择 **添加** 。 将 `Account` 类的定义替换为以下代码。 类定义使用自动实现的属性。

C#

```
class Account
{
    public int ID { get; set; }
    public double Balance { get; set; }
}
```

若要创建包含两个帐户的 `bankAccounts` 列表，请将以下代码添加到 ThisAddIn.cs 中的 `ThisAddIn_Startup` 方法。 列表声明使用集合初始值设定项。

C#

```
var bankAccounts = new List<Account>
{
    new Account
    {
        ID = 345,
        Balance = 541.27
    },
    new Account
    {
        ID = 123,
        Balance = -127.44
    }
};
```

将数据导出到 Excel

在相同的文件中，将以下方法添加到 `ThisAddIn` 类。 该方法设置 Excel 工作薄并将数据导出到工作簿。

C#

```
void DisplayInExcel(IEnumerable<Account> accounts,
           Action<Account, Excel.Range> DisplayFunc)
{
    var excelApp = this.Application;
    // Add a new Excel workbook.
    excelApp.Workbooks.Add();
    excelApp.Visible = true;
    excelApp.Range["A1"].Value = "ID";
    excelApp.Range["B1"].Value = "Balance";
    excelApp.Range["A2"].Select();

    foreach (var ac in accounts)
    {
        DisplayFunc(ac, excelApp.ActiveCell);
        excelApp.ActiveCell.Offset[1, 0].Select();
    }
```

```
    // Copy the results to the Clipboard.
    excelApp.Range["A1:B3"].Copy();
}
```

- 方法 Add 有一个*可选参数*，用于指定特定的模板。 如果希望使用形参的默认值，你可以借助可选形参以忽略该形参的实参。 由于上一个示例没有参数，`Add` 将使用默认模板并创建新的工作簿。 C# 早期版本中的等效语句要求提供一个占位符参数：`excelApp.Workbooks.Add(Type.Missing)`。 有关详细信息，请参阅命名参数和可选参数。
- Range 对象的 `Range` 和 `Offset` 属性使用“索引属性”功能。 此功能允许你通过以下典型 C# 语法从 COM 类型使用这些属性。 索引属性还允许你使用 `Value` 对象的 `Range` 属性，因此不必使用 `Value2` 属性。 `Value` 属性已编入索引，但索引是可选的。 在以下示例中，可选自变量和索引属性配合使用。

C#

```
// Visual C# 2010 provides indexed properties for COM programming.
excelApp.Range["A1"].Value = "ID";
excelApp.ActiveCell.Offset[1, 0].Select();
```

你不能创建自己的索引属性。 该功能仅支持使用现有索引属性。

在 `DisplayInExcel` 的末尾添加以下代码以将列宽调整为适合内容。

C#

```
excelApp.Columns[1].AutoFit();
excelApp.Columns[2].AutoFit();
```

这些新增内容介绍了 C# 中的另一功能：处理从 COM 主机返回的 `Object` 值（如 Office），就像它们具有 dynamic 类型一样。 当嵌入互操作类型具有其默认值时，COM 对象将被自动视为 `dynamic`；当你使用 EmbedInteropTypes 编译器选项引用程序集时，该对象被视为 `True` 或等效值。 有关嵌入互操作类型的详细信息，请参阅本文后面部分的“查找 PIA 引用”和“还原 PIA 依赖项”程序。 有关 `dynamic` 的详细信息，请参阅 dynamic 或使用类型 dynamic。

调用 DisplayInExcel

在 `ThisAddIn_StartUp` 方法的末尾添加以下代码。 对 `DisplayInExcel` 的调用包含两个参数。 第一个参数是已处理的帐户列表的名称。 第二个参数是定义如何处理数据的多行 Lambda 表达式。 每个帐户的 `ID` 和 `balance` 值都显示在相邻的单元格中，如果余额小于零，则相应的行显示为红色。 有关详细信息，请参阅 Lambda 表达式。

C#

```
DisplayInExcel(bankAccounts, (account, cell) =>
// This multiline lambda expression sets custom processing rules
// for the bankAccounts.
{
    cell.Value = account.ID;
    cell.Offset[0, 1].Value = account.Balance;
    if (account.Balance < 0)
    {
        cell.Interior.Color = 255;
        cell.Offset[0, 1].Interior.Color = 255;
    }
});
```

若要运行程序，请按 F5。 出现包含帐户数据的 Excel 工作表。

添加 Word 文档

在 `ThisAddIn_StartUp` 方法末尾添加以下代码，以创建包含指向 Excel 工作簿的链接的 Word 文档。

C#

```
var wordApp = new Word.Application();
wordApp.Visible = true;
wordApp.Documents.Add();
wordApp.Selection.PasteSpecial(Link: true, DisplayAsIcon: true);
```

此代码演示 C# 中的几个功能：在 COM 编程、命名参数和可选参数中省略 `ref` 关键字的功能。PasteSpecial 方法有七个参数，所有这些参数都是可选的引用参数。 通过命名实参和可选实参，你可以指定希望按名称访问的形参并仅将实参发送到这些形参。 在本示例中，实参指示创建指向剪贴板上工作簿的链接（形参 `Link`）并在 Word 文档中将该链接显示为图标（形参 `DisplayAsIcon`）。 C# 还允许忽略这些参数的 `ref` 关键字。

运行应用程序

按 F5 运行该应用程序。 Excel 启动并显示包含 `bankAccounts` 中两个帐户的信息的表。 然后，出现包含指向 Excel 表的 Word 文档。

清理已完成的项目

在 Visual Studio 中，选择“生成”菜单上的“清理解决方案”。 否则，每次在计算机上打开 Excel 时都会运行加载项。

查找 PIA 引用

1. 再次运行应用程序，但不选择“清理解决方案”。
2. 选择“开始”。 找到“Microsoft Visual Studio <版本>”，然后打开开发人员命令提示。
3. 在“Visual Studio 的开发人员命令提示”窗口中键入 `ildasm`，然后按 Enter。 此时将出现 IL DASM 窗口。
4. 在 IL DASM 窗口的“文件”菜单上，选择“文件”>“打开”。 双击“Visual Studio <版本>”，然后双击“项目”。 打开项目的文件夹，在 bin/Debug 文件夹中查找*项目名称*.dll。 双击 *项目名称*.dll。 新窗口将显示项目的属性以及对其他模块和程序集的引用。 程序集包括命名空间 `Microsoft.Office.Interop.Excel` 和 `Microsoft.Office.Interop.Word`。 在 Visual Studio 中，编译器默认将所需的类型从引用的 PIA 导入程序集。 有关详细信息，请参阅如何：查看程序集内容。
5. 双击“清单”图标。 此时将出现包含程序集列表的窗口，这些程序集包含项目所引用的项。 `Microsoft.Office.Interop.Excel` 和 `Microsoft.Office.Interop.Word` 不在列表中。 由于已将项目所需的类型导入程序集，因此无需安装对 PIA 的引用。 将类型导入程序集可简化部署。 用户计算机上不必存在 PIA。 应用程序不需要部署特定版本的 PIA。 应用程序可以使用多个版本的 Office，前提是所有版本中都存在必要的 API。 由于不再需要部署 PIA，你可以提前创建可与多个版本的 Office（包括之前的版本）配合使用的应用程序。 你的代码不能使用你正在使用的 Office 版本中不可用的任何 API。 特定 API 在早期版本中是否可用并不总是很清楚。 不建议使用早期版本的 Office。
6. 关闭清单窗口和程序集窗口。

还原 PIA 依赖项

1. 在解决方案资源管理器中选择“显示所有文件”按钮。 展开“引用”文件夹并选择“Microsoft.Office.Interop.Excel”。 按 F4 以显示“属性”窗口。
2. 在“属性”窗口中，将“嵌入互操作类型”属性从“True”更改为“False”。
3. 对 `Microsoft.Office.Interop.Word` 重复此程序中的步骤 1 和 2。
4. 在 C# 中，在 `Autofit` 方法的末尾注释掉对 `DisplayInExcel` 的两次调用。
5. 按 F5 以验证项目是否仍正确运行。
6. 重复上一个程序的步骤 1-3 以打开程序集窗口。 注意，`Microsoft.Office.Interop.Word` 和 `Microsoft.Office.Interop.Excel` 不再位于嵌入程序集列表中。
7. 双击“清单”图标并滚动引用程序集的列表。 `Microsoft.Office.Interop.Word` 和 `Microsoft.Office.Interop.Excel` 均位于列表中。 由于应用程序引用 Excel 和 Word

PIA 并且“嵌入互操作类型”属性为“False”，因此最终用户的计算机上必须存在两个程序集。

8. 在 Visual Studio 中，选择“生成”菜单上的“清理解决方案”以清理完成的项目。

请参阅

- 自动实现的属性 (C#)
- 对象和集合初始值设定项
- Visual Studio Tools for Office (VSTO)
- 命名参数和可选参数
- dynamic
- 使用类型 dynamic
- Lambda 表达式 (C#)
- 演练：在 Visual Studio 中嵌入 Microsoft Office 程序集中的类型信息
- 演练：嵌入托管程序集中的类型
- 演练：创建你的第一个 Excel VSTO 外接程序

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

如何使用平台调用播放 WAV 文件

项目 • 2023/04/07

下面的 C# 代码示例说明了如何使用平台调用服务在 Windows 操作系统中播放 WAV 声音文件。

示例

此示例代码使用 DllImportAttribute 将 `winmm.dll` 的 `PlaySound` 方法入口点导入为 `Form1 PlaySound()`。 本示例具有一个带按钮的简单 Windows 窗体。 单击该按钮将打开一个标准的 Windows OpenFileDialog 对话框，以便你可以打开要播放的文件。 选中波形文件后，该文件将使用 winmm.dll 库的 `PlaySound()` 方法播放。 有关此方法的详细信息，请参阅使用 PlaySound 功能处理波形音频文件。 浏览并选择具有 .wav 扩展名的文件，然后选择“打开”以使用平台调用播放波形文件。 文本框中显示所选文件的完整路径。

C#

```
using System.Runtime.InteropServices;

namespace WinSound;

public partial class Form1 : Form
{
    private TextBox textBox1;
    private Button button1;

    public Form1()  // Constructor.
    {
        InitializeComponent();
    }

    [DllImport("winmm.DLL", EntryPoint = "PlaySound", SetLastError = true,
CharSet = CharSet.Unicode, ThrowOnUnmappableChar = true)]
    private static extern bool PlaySound(string szSound, System.IntPtr hMod,
PlaySoundFlags flags);

    [System.Flags]
    public enum PlaySoundFlags : int
    {
        SND_SYNC = 0x0000,
        SND_ASYNC = 0x0001,
        SND_NODEFAULT = 0x0002,
        SND_LOOP = 0x0008,
        SND_NOSTOP = 0x0010,
        SND_NOWAIT = 0x00002000,
        SND_FILENAME = 0x00020000,
        SND_RESOURCE = 0x00040004
```

```
    }

    private void button1_Click(object sender, System.EventArgs e)
    {
        var dialog1 = new OpenFileDialog();

        dialog1.Title = "Browse to find sound file to play";
        dialog1.InitialDirectory = @"c:\";
        //<Snippet5>
        dialog1.Filter = "Wav Files (*.wav)|*.wav";
        //</Snippet5>
        dialog1.FilterIndex = 2;
        dialog1.RestoreDirectory = true;

        if (dialog1.ShowDialog() == DialogResult.OK)
        {
            textBox1.Text = dialog1.FileName;
            PlaySound(dialog1.FileName, new System.IntPtr(),
PlaySoundFlags.SND_SYNC);
        }
    }

    private void Form1_Load(object sender, EventArgs e)
    {
        // Including this empty method in the sample because in the IDE,
        // when users click on the form, generates code that looks for a
default method
        // with this name. We add it here to prevent confusion for those
using the samples.
    }
}
```

通过筛选器设置对“打开文件”对话框进行筛选，以仅显示扩展名为 .wav 的文件。

编译代码

在 Visual Studio 中创建一个新的 C# Windows Forms 应用程序项目，并将其命名为“WinSound”。 复制前面的代码，将其粘贴到 Form1.cs 文件的内容中。 复制以下代码，然后将其粘贴到 方法中 Form1.Designer.cs 文件的任何现有代码之后。

C#

```
this.button1 = new System.Windows.Forms.Button();
this.textBox1 = new System.Windows.Forms.TextBox();
this.SuspendLayout();
//
// button1
//
this.button1.Location = new System.Drawing.Point(192, 40);
this.button1.Name = "button1";
this.button1.Size = new System.Drawing.Size(88, 24);
```

```
this.button1.TabIndex = 0;
this.button1.Text = "Browse";
this.button1.Click += new System.EventHandler(this.button1_Click);
//
// textBox1
//
this.textBox1.Location = new System.Drawing.Point(8, 40);
this.textBox1.Name = "textBox1";
this.textBox1.Size = new System.Drawing.Size(168, 20);
this.textBox1.TabIndex = 1;
this.textBox1.Text = "File path";
//
// Form1
//
this.AutoScaleDimensions = new System.Drawing.SizeF(5, 13);
this.ClientSize = new System.Drawing.Size(292, 266);
this.Controls.Add(this.textBox1);
this.Controls.Add(this.button1);
this.Name = "Form1";
this.Text = "Platform Invoke WinSound C#";
this.ResumeLayout(false);
this.PerformLayout();
```

编译并运行该代码。

另请参阅

- 平台调用详解
- 用平台调用封送数据

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

如何在 COM 互操作编程中使用索引属性

项目 • 2023/09/21

结合使用索引属性与 C# 中的其他功能（如命名参数和可选参数、一种新类型 (dynamic) 以及嵌入类型信息）可以增强 Microsoft Office 编程的功能。

> **ⓘ 重要**
>
> VSTO (Visual Studio Tools for Office) 依赖于 .NET Framework。 COM 加载项也可以使用 .NET Framework 编写。 不能使用 .NET Core 和 .NET 5+（最新版本的 .NET）创建 Office 加载项。 这是因为 .NET Core/.NET 5+ 无法在同一进程中与 .NET Framework 协同工作，并可能导致加载项加载失败。 可以继续使用 .NET Framework 编写适用于 Office 的 VSTO 和 COM 加载项。 Microsoft 不会更新 VSTO 或 COM 加载项平台以使用 .NET Core 或 .NET 5+。 可以利用 .NET Core 和 .NET 5+（包括 ASP.NET Core）创建 Office Web 加载项的服务器端。

在早期版本的 C# 中，仅当 `get` 方法没有参数且 `set` 方法有且只有一个值参数时，方法才能作为属性访问。 但是，并非所有 COM 属性都符合上述限制。 例如，Excel Range[] 属性具有一个 `get` 访问器，它需要该范围名称的一个参数。 过去，由于无法直接访问 `Range` 属性，必须使用 `get_Range` 方法，如以下示例所示。

{language}

```
// Visual C# 2008 and earlier.
var excelApp = new Excel.Application();
// . . .
Excel.Range targetRange = excelApp.get_Range("A1", Type.Missing);
```

利用索引属性，你可以改为编写以下代码：

{language}

```
// Visual C# 2010.
var excelApp = new Excel.Application();
// . . .
Excel.Range targetRange = excelApp.Range["A1"];
```

上一示例还使用了可选实参功能，以便忽略 `Type.Missing`。

利用索引属性，你可以编写以下代码。

{language}

```
// Visual C# 2010.
targetRange.Value = "Name";
```

你不能创建自己的索引属性。 该功能仅支持使用现有索引属性。

示例

以下代码显示完整示例。 有关如何设置访问 Office API 的项目的详细信息，请参阅如何使用 C# 功能访问 Office 互操作对象。

{language}

```
// You must add a reference to Microsoft.Office.Interop.Excel to run
// this example.
using System;
using Excel = Microsoft.Office.Interop.Excel;

namespace IndexedProperties
{
    class Program
    {
        static void Main(string[] args)
        {
            CSharp2010();
        }

        static void CSharp2010()
        {
            var excelApp = new Excel.Application();
            excelApp.Workbooks.Add();
            excelApp.Visible = true;

            Excel.Range targetRange = excelApp.Range["A1"];
            targetRange.Value = "Name";
        }

        static void CSharp2008()
        {
            var excelApp = new Excel.Application();
            excelApp.Workbooks.Add(Type.Missing);
            excelApp.Visible = true;

            Excel.Range targetRange = excelApp.get_Range("A1",
Type.Missing);
            targetRange.set_Value(Type.Missing, "Name");
            // Or
            //targetRange.Value2 = "Name";
        }
```

```
    }
}
```

请参阅

- 命名参数和可选参数
- dynamic
- 使用类型 dynamic

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

如何访问 Office 互操作对象

项目 • 2023/03/09

C# 具有一些功能，可简化对 Office API 对象的访问。 这些新功能包括命名实参和可选实参、名为 `dynamic` 的新类型，以及在 COM 方法中将实参传递为引用形参（就像它们是值形参）的功能。

在本文中，你将使用这些新功能来编写代码，创建并显示 Microsoft Office Excel 工作表。 你将编写代码来添加包含链接到 Excel 工作表的图标的 Office Word 文档。

若要完成本演练，你的计算机上必须安装 Microsoft Office Excel 2007 和 Microsoft Office Word 2007 或更高版本。

> **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。

> **重要**
>
> VSTO (Visual Studio Tools for Office) 依赖于 .NET Framework。 COM 加载项也可以使用 .NET Framework 编写。 不能使用 .NET Core 和 .NET 5+（最新版本的 .NET）创建 Office 加载项。 这是因为 .NET Core/.NET 5+ 无法在同一进程中与 .NET Framework 协同工作，并可能导致加载项加载失败。 可以继续使用 .NET Framework 编写适用于 Office 的 VSTO 和 COM 加载项。 Microsoft 不会更新 VSTO 或 COM 加载项平台以使用 .NET Core 或 .NET 5+。 可以利用 .NET Core 和 .NET 5+（包括 ASP.NET Core）创建 Office Web 加载项的服务器端。

创建新的控制台应用程序

1. 启动 Visual Studio。
2. 在 “**文件**” 菜单上，指向 “**新建**”，然后选择 “**项目**”。 此时将出现“新建项目” 对话框。
3. 在“已安装的模板”窗格中，展开“C#”，然后选择“Windows”。
4. 查看“新建项目”对话框的顶部，确保将“.NET Framework 4”（或更高版本）选为目标框架。
5. 在“**模板**”窗格中，选择“**控制台应用程序**”。
6. 在“名称”字段中键入项目的名称。

7. 选择“确定”。

新项目将出现在“解决方案资源管理器”中。

添加引用

1. 在“解决方案资源管理器”中，右键单击你的项目名称，然后选择“添加引用”。 此时会显示“添加引用”对话框。
2. 在“程序集”页上，在“组件名称”列表中选择“Microsoft.Office.Interop.Word”，然后按住 Ctrl 键并选择“Microsoft.Office.Interop.Excel”。 如果没有看到程序集，可能需要进行安装。 请参阅如何：安装 Office 主互操作程序集。
3. 选择“确定” 。

添加必要的 using 指令

在“解决方案资源管理器”中，右键单击“Program.cs”文件，然后选择“查看代码”。 将以下 `using` 指令添加到代码文件的顶部：

C#

```
using Excel = Microsoft.Office.Interop.Excel;
using Word = Microsoft.Office.Interop.Word;
```

创建银行帐户列表

将以下类定义粘贴到“Program.cs”中的 `Program` 类下。

C#

```
public class Account
{
    public int ID { get; set; }
    public double Balance { get; set; }
}
```

将以下代码添加到 `Main` 方法，以创建包含两个帐户的 `bankAccounts` 列表。

C#

```
// Create a list of accounts.
var bankAccounts = new List<Account> {
    new Account {
                  ID = 345678,
```

```
                    Balance = 541.27
                },
    new Account {
                    ID = 1230221,
                    Balance = -127.44
                }
};
```

声明将帐户信息导出到 Excel 的方法

1. 将以下方法添加到 `Program` 类以设置 Excel 工作表。 方法 Add 有一个可选参数，用于指定特定的模板。 如果希望使用形参的默认值，你可以借助可选形参以忽略该形参的实参。 由于你没有提供参数，`Add` 会使用默认模板并创建新的工作簿。 C# 早期版本中的等效语句要求提供一个占位符参数：`ExcelApp.Workbooks.Add(Type.Missing)`。

C#

```
static void DisplayInExcel(IEnumerable<Account> accounts)
{
    var excelApp = new Excel.Application();
    // Make the object visible.
    excelApp.Visible = true;

    // Create a new, empty workbook and add it to the collection returned
    // by property Workbooks. The new workbook becomes the active workbook.
    // Add has an optional parameter for specifying a particular template.
    // Because no argument is sent in this example, Add creates a new
workbook.
    excelApp.Workbooks.Add();

    // This example uses a single workSheet. The explicit type casting is
    // removed in a later procedure.
    Excel._Worksheet workSheet = (Excel.Worksheet)excelApp.ActiveSheet;
}
```

在 `DisplayInExcel` 的末尾添加以下代码。 代码将值插入工作表第一行的前两列。

C#

```
// Establish column headings in cells A1 and B1.
workSheet.Cells[1, "A"] = "ID Number";
workSheet.Cells[1, "B"] = "Current Balance";
```

在 `DisplayInExcel` 的末尾添加以下代码。 `foreach` 循环将帐户列表中的信息放入工作表连续行的前两列。

C#

```
var row = 1;
foreach (var acct in accounts)
{
    row++;
    workSheet.Cells[row, "A"] = acct.ID;
    workSheet.Cells[row, "B"] = acct.Balance;
}
```

在 `DisplayInExcel` 的末尾添加以下代码以将列宽调整为适合内容。

C#

```
workSheet.Columns[1].AutoFit();
workSheet.Columns[2].AutoFit();
```

早期版本的 C# 要求显式强制转换这些操作，因为 `ExcelApp.Columns[1]` 返回 `Object` 且 `AutoFit` 为 Excel Range 方法。 以下各行显示强制转换。

C#

```
((Excel.Range)workSheet.Columns[1]).AutoFit();
((Excel.Range)workSheet.Columns[2]).AutoFit();
```

C# 自动将返回的 `Object` 转换为 `dynamic`，前提是程序集由 EmbedInteropTypes 编译器选项引用，或 Excel 的“嵌入互操作类型”属性为 true。 True 是此属性的默认值。

运行项目

在 `Main` 的末尾添加以下行。

C#

```
// Display the list in an Excel spreadsheet.
DisplayInExcel(bankAccounts);
```

按 Ctrl+F5。 出现包含两个帐户数据的 Excel 工作表。

添加 Word 文档

以下代码可以打开 Word 应用程序并创建链接到 Excel 工作表的图标。 将方法 `CreateIconInWordDoc`（在此步骤后面提供）粘贴到 `Program` 类中。 `CreateIconInWordDoc` 使用命名参数和可选参数来降低对 Add 和 PasteSpecial 方法调用的复杂性。 这些调用合并了其他两项新功能，这两项新功能简化了具有引用参数的 COM 方法的调用。 首先，你可以将实参发送到引用形参，就像它们是值形参一样。 即，你可以直接发送值，而无需为每个引用参数创建变量。 编译器会生成临时变量以保存参数值，并将在你从调用返回时丢弃变量。 其次，你可以忽略参数列表中的 `ref` 关键字。

`Add` 方法有四个引用参数，所有引用参数都是可选的。 如果希望使用其默认值，可以忽略任何或所有形参的实参。

`PasteSpecial` 方法可插入剪贴板的内容。 该方法有七个引用参数，所有引用参数都是可选的。 以下代码为其中两个形参指定实参：`Link` 用于创建指向剪贴板内容源的链接，`DisplayAsIcon` 用于将链接显示为图标。 可以对其中两个实参（忽略其他实参）使用命名实参。 尽管这些实参是引用形参，你也不必使用 `ref` 关键字，或者创建变量以实参形式发送。 你可以直接发送值。

C#

```
static void CreateIconInWordDoc()
{
    var wordApp = new Word.Application();
    wordApp.Visible = true;

    // The Add method has four reference parameters, all of which are
    // optional. Visual C# allows you to omit arguments for them if
    // the default values are what you want.
    wordApp.Documents.Add();

    // PasteSpecial has seven reference parameters, all of which are
    // optional. This example uses named arguments to specify values
    // for two of the parameters. Although these are reference
    // parameters, you do not need to use the ref keyword, or to create
    // variables to send in as arguments. You can send the values directly.
    wordApp.Selection.PasteSpecial( Link: true, DisplayAsIcon: true);
}
```

在 `Main` 的末尾添加以下语句。

C#

```
// Create a Word document that contains an icon that links to
// the spreadsheet.
CreateIconInWordDoc();
```

在 `DisplayInExcel` 的末尾添加以下语句。 `Copy` 方法可将工作表添加到剪贴板。

C#

```
// Put the spreadsheet contents on the clipboard. The Copy method has one
// optional parameter for specifying a destination. Because no argument
// is sent, the destination is the Clipboard.
workSheet.Range["A1:B3"].Copy();
```

按 Ctrl+F5。 将出现包含图标的 Word 文档。 双击该图标以将工作表置于前台。

设置嵌入互操作类型属性

当调用运行时不需要主互操作程序集 (PIA) 的 COM 类型时，可能实现其他增强功能。 删除 PIA 的依赖项可实现版本独立性并且更易于部署。 若要详细了解不使用 PIA 编程的优势，请参阅演练：嵌入托管程序集中的类型。

此外，由于 `dynamic` 类型表示在 COM 方法中声明的必需类型和返回类型，因此更易于编程。 具有类型 `dynamic` 的变量在运行时以前均不会计算，从而消除了显式强制转换的必要性。 有关更多信息，请参见使用类型 dynamic。

默认行为是嵌入类型信息，而不是使用 PIA。 由于该默认行为，之前的几个示例也得到简化。 不需要任何显式强制转换。 例如，`worksheet` 中 `DisplayInExcel` 的声明会写为 `Excel._Worksheet workSheet = excelApp.ActiveSheet` 而非 `Excel._Worksheet workSheet = (Excel.Worksheet)excelApp.ActiveSheet`。 在相同方法中对 `AutoFit` 的调用还将要求在不进行默认行为的情况下显式强制转换，因为 `ExcelApp.Columns[1]` 返回 `Object`，并且 `AutoFit` 为 Excel 方法。 以下代码显示强制转换。

C#

```
((Excel.Range)workSheet.Columns[1]).AutoFit();
((Excel.Range)workSheet.Columns[2]).AutoFit();
```

若要更改默认行为并使用 PIA 代替嵌入类型信息，请展开“解决方案资源管理器”中的“引用”节点，然后选择“Microsoft.Office.Interop.Excel”或“Microsoft.Office.Interop.Word”。 如果看不到“属性”窗口，请按 F4。 在属性列表中找到“嵌入互操作类型”，将其值更改为“False”。 同样地，还可以通过在命令提示符处使用 References 编译器选项代替 EmbedInteropTypes 进行编译。

将其他格式添加到表格

将在 `AutoFit` 中对 `DisplayInExcel` 的两个调用替换为以下语句。

C#

```
// Call to AutoFormat in Visual C# 2010.
workSheet.Range["A1", "B3"].AutoFormat(
    Excel.XlRangeAutoFormat.xlRangeAutoFormatClassic2);
```

AutoFormat 方法有七个值参数，所有引用参数都是可选的。 使用命名自变量和可选自变量，你可以为这些自变量中的所有或部分提供自变量，也可以不为它们中的任何一个提供。 在上一条语句中，仅为其中一个形参 `Format` 提供了实参。 由于 `Format` 是参数列表中的第一个参数，因此无需提供参数名称。 但是，如果包含参数名称，语句可能更易于理解，如以下代码所示。

C#

```
// Call to AutoFormat in Visual C# 2010.
workSheet.Range["A1", "B3"].AutoFormat(Format:
    Excel.XlRangeAutoFormat.xlRangeAutoFormatClassic2);
```

按 Ctrl+F5 查看结果。 可以在列出的 XlRangeAutoFormat 枚举中找到其他格式。

示例

以下代码显示完整示例。

C#

```
using System.Collections.Generic;
using Excel = Microsoft.Office.Interop.Excel;
using Word = Microsoft.Office.Interop.Word;

namespace OfficeProgrammingWalkthruComplete
{
    class Walkthrough
    {
        static void Main(string[] args)
        {
            // Create a list of accounts.
            var bankAccounts = new List<Account>
            {
                new Account {
                              ID = 345678,
                              Balance = 541.27
                            },
                new Account {
                              ID = 1230221,
                              Balance = -127.44
                            }
            };
```

```
            // Display the list in an Excel spreadsheet.
            DisplayInExcel(bankAccounts);

            // Create a Word document that contains an icon that links to
            // the spreadsheet.
            CreateIconInWordDoc();
        }

        static void DisplayInExcel(IEnumerable<Account> accounts)
        {
            var excelApp = new Excel.Application();
            // Make the object visible.
            excelApp.Visible = true;

            // Create a new, empty workbook and add it to the collection
returned
            // by property Workbooks. The new workbook becomes the active
workbook.
            // Add has an optional parameter for specifying a particular
template.
            // Because no argument is sent in this example, Add creates a
new workbook.
            excelApp.Workbooks.Add();

            // This example uses a single workSheet.
            Excel._Worksheet workSheet = excelApp.ActiveSheet;

            // Earlier versions of C# require explicit casting.
            //Excel._Worksheet workSheet =
(Excel.Worksheet)excelApp.ActiveSheet;

            // Establish column headings in cells A1 and B1.
            workSheet.Cells[1, "A"] = "ID Number";
            workSheet.Cells[1, "B"] = "Current Balance";

            var row = 1;
            foreach (var acct in accounts)
            {
                row++;
                workSheet.Cells[row, "A"] = acct.ID;
                workSheet.Cells[row, "B"] = acct.Balance;
            }

            workSheet.Columns[1].AutoFit();
            workSheet.Columns[2].AutoFit();

            // Call to AutoFormat in Visual C#. This statement replaces the
            // two calls to AutoFit.
            workSheet.Range["A1", "B3"].AutoFormat(
                Excel.XlRangeAutoFormat.xlRangeAutoFormatClassic2);

            // Put the spreadsheet contents on the clipboard. The Copy
method has one
            // optional parameter for specifying a destination. Because no
argument
```

```
            // is sent, the destination is the Clipboard.
            workSheet.Range["A1:B3"].Copy();
        }

        static void CreateIconInWordDoc()
        {
            var wordApp = new Word.Application();
            wordApp.Visible = true;

            // The Add method has four reference parameters, all of which
are
            // optional. Visual C# allows you to omit arguments for them if
            // the default values are what you want.
            wordApp.Documents.Add();

            // PasteSpecial has seven reference parameters, all of which are
            // optional. This example uses named arguments to specify values
            // for two of the parameters. Although these are reference
            // parameters, you do not need to use the ref keyword, or to
create
            // variables to send in as arguments. You can send the values
directly.
            wordApp.Selection.PasteSpecial(Link: true, DisplayAsIcon: true);
        }
    }

    public class Account
    {
        public int ID { get; set; }
        public double Balance { get; set; }
    }
}
```

请参阅

- Type.Missing
- dynamic
- 命名参数和可选参数
- 如何在 Office 编程中使用命名参数和可选参数

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

如何在 Office 编程中使用命名自变量和可选自变量

项目 • 2024/06/03

命名参数和可选参数增强了 C# 编程中的便利性、灵活性和可读性。 另外，这些功能显著方便了对 COM 接口（如 Microsoft Office 自动化 API）的访问。

> **ⓘ 重要**
>
> VSTO (Visual Studio Tools for Office) 依赖于 .NET Framework。 COM 加载项也可以使用 .NET Framework 编写。 不能使用 .NET Core 和 .NET 5+（最新版本的 .NET）创建 Office 加载项。 这是因为 .NET Core/.NET 5+ 无法在同一进程中与 .NET Framework 协同工作，并可能导致加载项加载失败。 可以继续使用 .NET Framework 编写适用于 Office 的 VSTO 和 COM 加载项。 Microsoft 不会更新 VSTO 或 COM 加载项平台以使用 .NET Core 或 .NET 5+。 可以利用 .NET Core 和 .NET 5+（包括 ASP.NET Core）创建 Office Web 加载项的服务器端。

在下面的示例中，方法 ConvertToTable 具有 16 个参数，用于表示表的各种特性，例如列数和行数、格式设置、边框、字体以及颜色。 由于大多数时候都不需要为所有 16 个参数指定特定值，因此所有这些参数都是可选的。 但是，如果没有命名参数和可选参数，则必须提供值或占位符值。 有了命名参数和可选参数，则只需为项目所需的参数指定值。

必须在计算机上安装 Microsoft Office Word 才能完成这些过程。

> **ⓘ 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。

创建新的控制台应用程序

启动 Visual Studio。 在“**文件**”菜单上，指向“**新建**”，然后选择“**项目**”。 在“模板类别”窗格中，展开“C#”，然后选择“Windows”。 查看“模板”窗格的顶部，确保“.NET Framework 4”出现在“目标框架”框中。 在“**模板**”窗格中，选择“**控制台应用程序**”。 在“名称”字段中键入项目的名称。 选择“确定”。 新项目将出现在“解决方案资源管理器”中。

添加引用

在“解决方案资源管理器”中，右键单击你的项目名称，然后选择“添加引用”。 此时会显示“添加引用”对话框。 在“.NET”页上的“组件名称”列表中，选择“Microsoft.Office.Interop.Word” 。 选择“确定”。

添加必要的 using 指令

在“解决方案资源管理器”中，右键单击“Program.cs”文件，然后选择“查看代码”。 将以下 `using` 指令添加到代码文件的顶部：

C#

```
using Word = Microsoft.Office.Interop.Word;
```

在 Word 文档中显示文本

在“Program.cs”的 `Program` 类中，添加以下方法以创建 Word 应用程序和 Word 文档。 Add 方法具有四个可选参数。 此示例使用这些参数的默认值。 因此，调用语句中不必有参数。

C#

```
static void DisplayInWord()
{
    var wordApp = new Word.Application();
    wordApp.Visible = true;
    // docs is a collection of all the Document objects currently
    // open in Word.
    Word.Documents docs = wordApp.Documents;

    // Add a document to the collection and name it doc.
    Word.Document doc = docs.Add();
}
```

将以下代码添加到方法的末尾，以定义要在文档何处显示文本，以及要显示什么文本：

C#

```
// Define a range, a contiguous area in the document, by specifying
// a starting and ending character position. Currently, the document
// is empty.
Word.Range range = doc.Range(0, 0);

// Use the InsertAfter method to insert a string at the end of the
```

```
// current range.
range.InsertAfter("Testing, testing, testing. . .");
```

运行应用程序

将以下语句添加到 Main：

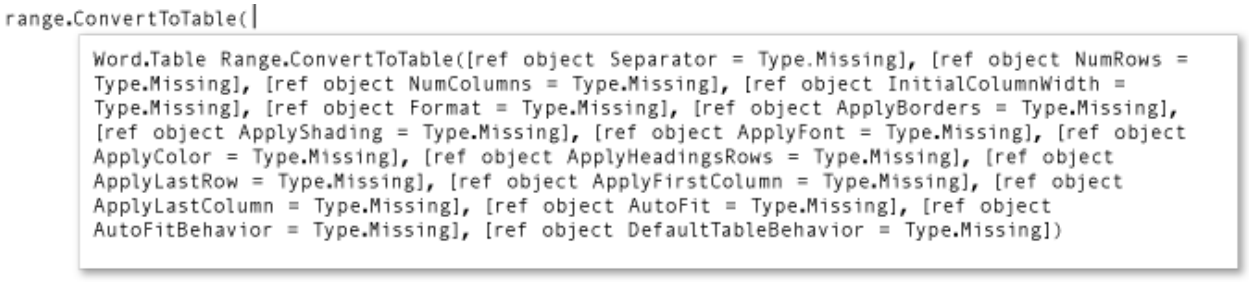
C#

```
DisplayInWord();
```

按 Ctrl+F5 运行项目。 此时会出现一个 Word 文档，其中包含指定的文本。

将文本更改为表

使用 `ConvertToTable` 方法将文本放入表中。 该方法具有 16 个可选参数。 IntelliSense 将可选参数放入括号中，如下图所示。 `Type.Missing` 的默认值是 `System.Type.Missing` 的简单名称。



通过使用命名实参和可选实参，可以只对要更改的形参指定值。 将以下代码添加到方法 `DisplayInWord` 的末尾以创建一个表。 此参数指定 `range` 中文本字符串内的逗号分隔表的各个单元格。

C#

```
// Convert to a simple table. The table will have a single row with
// three columns.
range.ConvertToTable(Separator: ",");
```

按 Ctrl+F5 运行项目。

使用其他参数进行试验

更改表以使其具有一列三行，将 `DisplayInWord` 中的最后一行替换为以下语句，然后按 `CTRL`+`F5`。

C#

```
range.ConvertToTable(Separator: ",", AutoFit: true, NumColumns: 1);
```

为表指定预定义的格式，将 `DisplayInWord` 中的最后一行替换为以下语句，然后按 `CTRL`+`F5`。 格式可以为任何 WdTableFormat 常量。

C#

```
range.ConvertToTable(Separator: ",", AutoFit: true, NumColumns: 1,
    Format: Word.WdTableFormat.wdTableFormatElegant);
```

示例

下面的代码包括完整的示例：

C#

```
using System;
using Word = Microsoft.Office.Interop.Word;

namespace OfficeHowTo
{
    class WordProgram
    {
        static void Main(string[] args)
        {
            DisplayInWord();
        }

        static void DisplayInWord()
        {
            var wordApp = new Word.Application();
            wordApp.Visible = true;
            // docs is a collection of all the Document objects currently
            // open in Word.
            Word.Documents docs = wordApp.Documents;

            // Add a document to the collection and name it doc.
            Word.Document doc = docs.Add();

            // Define a range, a contiguous area in the document, by
specifying
            // a starting and ending character position. Currently, the
document
```

```
            // is empty.
            Word.Range range = doc.Range(0, 0);

            // Use the InsertAfter method to insert a string at the end of
the
            // current range.
            range.InsertAfter("Testing, testing, testing. . .");

            // You can comment out any or all of the following statements to
            // see the effect of each one in the Word document.

            // Next, use the ConvertToTable method to put the text into a
table.
            // The method has 16 optional parameters. You only have to
specify
            // values for those you want to change.

            // Convert to a simple table. The table will have a single row
with
            // three columns.
            range.ConvertToTable(Separator: ",");

            // Change to a single column with three rows..
            range.ConvertToTable(Separator: ",", AutoFit: true, NumColumns:
1);

            // Format the table.
            range.ConvertToTable(Separator: ",", AutoFit: true, NumColumns:
1,
                Format: Word.WdTableFormat.wdTableFormatElegant);
        }
    }
}
```

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

使用类型 dynamic

项目 • 2023/06/13

`dynamic` 类型是一种静态类型，但类型为 `dynamic` 的对象会跳过静态类型检查。 大多数情况下，该对象就像具有类型 `object` 一样。 编译器假定 `dynamic` 元素支持任何操作。 因此，无需确定对象是从 COM API、动态语言（例如 IronPython）、HTML 文档对象模型 (DOM)、反射还是程序中的其他位置获取自己的值。 但是，如果代码无效，则会在运行时出现错误。

例如，如果以下代码中的实例方法 `exampleMethod1` 只有一个形参，则编译器会将对该方法的第一个调用 `ec.exampleMethod1(10, 4)` 识别为无效，因为它包含两个实参。 该调用将导致编译器错误。 编译器不会检查对该方法的第二个调用 `dynamic_ec.exampleMethod1(10, 4)`，因为 `dynamic_ec` 的类型为 `dynamic`。 因此，不会报告编译器错误。 但是，该错误不会被无限期疏忽。 它将在运行时出现，并导致运行时异常。

C#

```
static void Main(string[] args)
{
    ExampleClass ec = new ExampleClass();
    // The following call to exampleMethod1 causes a compiler error
    // if exampleMethod1 has only one parameter. Uncomment the line
    // to see the error.
    //ec.exampleMethod1(10, 4);

    dynamic dynamic_ec = new ExampleClass();
    // The following line is not identified as an error by the
    // compiler, but it causes a run-time exception.
    dynamic_ec.exampleMethod1(10, 4);

    // The following calls also do not cause compiler errors, whether
    // appropriate methods exist or not.
    dynamic_ec.someMethod("some argument", 7, null);
    dynamic_ec.nonexistentMethod();
}
```

C#

```
class ExampleClass
{
    public ExampleClass() { }
    public ExampleClass(int v) { }

    public void exampleMethod1(int i) { }
```

```
    public void exampleMethod2(string str) { }
}
```

在这些示例中，编译器的作用是将有关每个语句的预期作用的信息一起打包到 `dynamic` 对象或表达式。 运行时检查存储的信息，任何无效的语句都会导致运行时异常。

大多数动态操作的结果是其本身 `dynamic`。 例如，如果将鼠标指针放在以下示例中使用的 `testSum` 上，则 IntelliSense 将显示类型“（局部变量）dynamic testSum”。

C#

```
dynamic d = 1;
var testSum = d + 3;
// Rest the mouse pointer over testSum in the following statement.
System.Console.WriteLine(testSum);
```

结果不为 `dynamic` 的操作包括：

- 从 `dynamic` 到另一种类型的转换。
- 包括类型为 `dynamic` 的自变量的构造函数调用。

例如，以下声明中 `testInstance` 的类型为 `ExampleClass`，而不是 `dynamic`：

C#

```
var testInstance = new ExampleClass(d);
```

转换

动态对象和其他类型之间的转换非常简单。 转换使开发人员能够在动态行为和非动态行为之间切换。

可以将任何内容隐式转换为 `dynamic`，如以下示例所示。

C#

```
dynamic d1 = 7;
dynamic d2 = "a string";
dynamic d3 = System.DateTime.Today;
dynamic d4 = System.Diagnostics.Process.GetProcesses();
```

相反，可以将任何隐式转换动态应用于类型 `dynamic` 的任何表达式。

C#

```
int i = d1;
string str = d2;
DateTime dt = d3;
System.Diagnostics.Process[] procs = d4;
```

使用类型为 dynamic 的参数重载决策

如果方法调用中的一个或多个参数的类型为 `dynamic`，或者方法调用的接收方的类型为 `dynamic`，则会在运行时（而不是在编译时）进行重载决策。 在以下示例中，如果唯一可访问的 `exampleMethod2` 方法接受字符串参数，则将 `d1` 作为参数发送不会导致编译器错误，但却会导致运行时异常。 重载决策之所以会在运行时失败，是因为 `d1` 的运行时类型为 `int`，而 `exampleMethod2` 要求为字符串。

C#

```
// Valid.
ec.exampleMethod2("a string");

// The following statement does not cause a compiler error, even though ec
is not
// dynamic. A run-time exception is raised because the run-time type of d1
is int.
ec.exampleMethod2(d1);
// The following statement does cause a compiler error.
//ec.exampleMethod2(7);
```

动态语言运行时

动态语言运行时 (DLR) 提供了支持 C# 中 `dynamic` 类型的基础结构，还提供了 IronPython 和 IronRuby 等动态编程语言的实现。 有关 DLR 的详细信息，请参阅动态语言运行时概述。

COM 互操作

通过将类型指定为 `object`，许多 COM 方法都允许参数类型和返回类型发生变化。 COM 互操作要求必须显式强制转换值，以便与 C# 中的强类型变量保持协调。 如果使用 EmbedInteropTypes（C# 编译器选项）选项进行编译，则可以通过引入 `dynamic` 类型将 COM 签名中出现的 `object` 看作是 `dynamic` 类型，从而避免大量的强制转换。 有关将 `dynamic` 类型与 COM 对象配合使用的详细信息，请参阅有关如何使用 C# 功能访问 Office 互操作对象的文章。

相关文章

Title	描述
dynamic	描述 `dynamic` 关键字的用法。
动态语言运行时概述	提供有关 DLR 的概述，DLR 是一种运行时环境，它将一组适用于动态语言的服务添加到公共语言运行时 (CLR)。
演练：创建和使用动态对象	提供有关如何创建自定义动态对象以及创建访问 `IronPython` 库的对象的分步说明。

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

演练：使用 C# 创建和使用动态对象

项目 • 2023/05/09

动态对象会在运行时（而非编译时）公开属性和方法等成员。 动态对象使你能够创建对象以处理与静态类型或格式不匹配的结构。 例如，可以使用动态对象来引用 HTML 文档对象模型 (DOM)，该模型包含有效 HTML 标记元素和特性的任意组合。 由于每个 HTML 文档都是唯一的，因此在运行时将确定特定 HTML 文档的成员。 引用 HTML 元素的特性的常用方法是，将该特性的名称传递给该元素的 `GetProperty` 方法。 若要引用 HTML 元素 `<div id="Div1">` 的 `id` 特性，首先获取对 `<div>` 元素的引用，然后使用 `divElement.GetProperty("id")`。 如果使用动态对象，则可以将 `id` 特性引用为 `divElement.id`。

动态对象还提供对 IronPython 和 IronRuby 等动态语言的便捷访问。 可以使用动态对象来引用在运行时解释的动态脚本。

使用晚期绑定引用动态对象。 将后期绑定对象的类型指定为 `dynamic`。有关详细信息，请参阅 dynamic。

可以使用 System.Dynamic 命名空间中的类来创建自定义动态对象。 例如，可以创建 ExpandoObject 并在运行时指定该对象的成员。 还可以创建继承 DynamicObject 类的自己的类型。 然后，可以替代 DynamicObject 类的成员以提供运行时动态功能。

本文包含两个独立的演练：

- 创建一个自定义对象，该对象会将文本文件的内容作为对象的属性动态公开。
- 创建使用 `IronPython` 库的项目。

先决条件

- 安装了具有 .NET 桌面开发工作负载的 Visual Studio 2022 版本 17.3 或更高版本。 当你选择此工作负载时，就包含了 .NET 7 SDK。

> **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。

- 对于第二个演练，安装 IronPython for .NET。 转到其下载页以获取最新版本。

创建自定义动态对象

第一个演练定义了搜索文本文件内容的自定义动态对象。 动态属性指定要搜索的文本。 例如，如果调用代码指定 `dynamicFile.Sample`，则动态类将返回一个字符串泛型列表，其中包含该文件中以“Sample”开头的所有行。 搜索不区分大小写。 动态类还支持两个可选参数。 第一个参数是一个搜索选项枚举值，它指定动态类应在行的开头、行的结尾或行中任意位置搜索匹配项。 第二个参数指定动态类应在搜索之前去除每行中的前导空格和尾部空格。 例如，如果调用代码指定 `dynamicFile.Sample(StringSearchOption.Contains)`，则动态类将在行中的任意位置搜索“Sample”。 如果调用代码指定 `dynamicFile.Sample(StringSearchOption.StartsWith, false)`，动态类将在每行的开头搜索“Sample”，但不会删除前导空格和尾随空格。 动态类的默认行为是在每行的开头搜索匹配项，并删除前导空格和尾部空格。

创建自定义动态类

启动 Visual Studio。 选择“创建新项目”。 在“创建新项目”对话框中，选择 C#，然后依次选择“控制台应用程序”和“下一步”。 在“配置新项目”对话框中，输入 `DynamicSample` 作为“项目名称”，然后选择“下一步” 。 在“其他信息”对话框中，为“目标框架”选择“.NET 7.0 (当前)”，然后选择“创建”。 在“解决方案资源管理器”中，右键单击 DynamicSample 项目，然后选择“添加”>“类”。 在“名称”框中，键入 `ReadOnlyFile`，然后选择“添加”。 在 ReadOnlyFile.cs 或 ReadOnlyFile.vb 文件的顶部，添加以下代码以导入 System.IO 和 System.Dynamic 命名空间。

C#

```
using System.IO;
using System.Dynamic;
```

自定义动态对象使用一个枚举来确定搜索条件。 在类语句的前面，添加以下枚举定义。

C#

```
public enum StringSearchOption
{
    StartsWith,
    Contains,
    EndsWith
}
```

更新类语句以继承 `DynamicObject` 类，如以下代码示例所示。

C#

```
class ReadOnlyFile : DynamicObject
```

将以下代码添加到 `ReadOnlyFile` 类，定义一个用于文件路径的私有字段，并定义 `ReadOnlyFile` 类的构造函数。

C#

```
// Store the path to the file and the initial line count value.
private string p_filePath;

// Public constructor. Verify that file exists and store the path in
// the private variable.
public ReadOnlyFile(string filePath)
{
    if (!File.Exists(filePath))
    {
        throw new Exception("File path does not exist.");
    }

    p_filePath = filePath;
}
```

1. 将下面的 `GetPropertyValue` 方法添加到 `ReadOnlyFile` 类。 `GetPropertyValue` 方法接收搜索条件作为输入，并返回文本文件中符合该搜索条件的行。 由 `ReadOnlyFile` 类提供的动态方法将调用 `GetPropertyValue` 方法以检索其各自的结果。

C#

```
public List<string> GetPropertyValue(string propertyName,
                                      StringSearchOption StringSearchOption =
StringSearchOption.StartsWith,
                                      bool trimSpaces = true)
{
    StreamReader sr = null;
    List<string> results = new List<string>();
    string line = "";
    string testLine = "";

    try
    {
        sr = new StreamReader(p_filePath);

        while (!sr.EndOfStream)
        {
            line = sr.ReadLine();

            // Perform a case-insensitive search by using the specified
search options.
            testLine = line.ToUpper();
```

```
            if (trimSpaces) { testLine = testLine.Trim(); }

            switch (StringSearchOption)
            {
                case StringSearchOption.StartsWith:
                    if (testLine.StartsWith(propertyName.ToUpper())) {
results.Add(line); }
                    break;
                case StringSearchOption.Contains:
                    if (testLine.Contains(propertyName.ToUpper())) {
results.Add(line); }
                    break;
                case StringSearchOption.EndsWith:
                    if (testLine.EndsWith(propertyName.ToUpper())) {
results.Add(line); }
                    break;
            }
        }
    }
    catch
    {
        // Trap any exception that occurs in reading the file and return
null.
        results = null;
    }
    finally
    {
        if (sr != null) {sr.Close();}
    }

    return results;
}
```

在 `GetPropertyValue` 方法后，添加以下代码以替代 DynamicObject 类的 TryGetMember 方法。 请求动态类的成员且未指定任何参数时，将调用 TryGetMember 方法。 `binder` 参数包含有关被引用成员的信息，而 `result` 参数则引用为指定的成员返回的结果。 TryGetMember 方法会返回一个布尔值，如果请求的成员存在，则返回的布尔值为 `true`，否则返回的布尔值为 `false`。

C#

```
// Implement the TryGetMember method of the DynamicObject class for dynamic
member calls.
public override bool TryGetMember(GetMemberBinder binder,
                                  out object result)
{
    result = GetPropertyValue(binder.Name);
    return result == null ? false : true;
}
```

在 `TryGetMember` 方法后，添加以下代码以替代 DynamicObject 类的 TryInvokeMember 方法。 使用参数请求动态类的成员时，将调用 TryInvokeMember 方法。 `binder` 参数包含有关被引用成员的信息，而 `result` 参数则引用为指定的成员返回的结果。 `args` 参数包含一个传递给成员的参数的数组。 TryInvokeMember 方法会返回一个布尔值，如果请求的成员存在，则返回的布尔值为 `true`，否则返回的布尔值为 `false`。

`TryInvokeMember` 方法的自定义版本期望第一个参数为上一步骤中定义的 `StringSearchOption` 枚举中的值。 `TryInvokeMember` 方法期望第二个参数为一个布尔值。如果这两个参数有一个或全部为有效值，则会将它们传递给 `GetPropertyValue` 方法以检索结果。

C#

```
// Implement the TryInvokeMember method of the DynamicObject class for
// dynamic member calls that have arguments.
public override bool TryInvokeMember(InvokeMemberBinder binder,
                                      object[] args,
                                      out object result)
{
    StringSearchOption StringSearchOption = StringSearchOption.StartsWith;
    bool trimSpaces = true;

    try
    {
        if (args.Length > 0) { StringSearchOption =
(StringSearchOption)args[0]; }
    }
    catch
    {
        throw new ArgumentException("StringSearchOption argument must be a
StringSearchOption enum value.");
    }

    try
    {
        if (args.Length > 1) { trimSpaces = (bool)args[1]; }
    }
    catch
    {
        throw new ArgumentException("trimSpaces argument must be a Boolean
value.");
    }

    result = GetPropertyValue(binder.Name, StringSearchOption, trimSpaces);

    return result == null ? false : true;
}
```

保存并关闭该文件。

创建示例文本文件

在“解决方案资源管理器”中，右键单击 DynamicSample 项目，然后选择“添加”>“新建项目”。 在“已安装的模板”窗格中，选择“常规”，然后选择“文本文件”模板。 保留“名称”框中的默认名称 TextFile1.txt，然后选择“添加”。 将以下文本复制到 TextFile1.txt 文件。

text

```
List of customers and suppliers

Supplier: Lucerne Publishing (https://www.lucernepublishing.com/)
Customer: Preston, Chris
Customer: Hines, Patrick
Customer: Cameron, Maria
Supplier: Graphic Design Institute (https://www.graphicdesigninstitute.com/)
Supplier: Fabrikam, Inc. (https://www.fabrikam.com/)
Customer: Seubert, Roxanne
Supplier: Proseware, Inc. (http://www.proseware.com/)
Customer: Adolphi, Stephan
Customer: Koch, Paul
```

保存并关闭文件。

创建一个使用自定义动态对象的示例应用程序

在“解决方案资源管理器”中，双击 Program.cs 文件。 将以下代码添加到 `Main` 过程，为 TextFile1.txt 文件创建 `ReadOnlyFile` 类的实例。 代码将使用晚期绑定来调用动态成员，并检索包含字符串“Customer”的文本行。

C#

```
dynamic rFile = new ReadOnlyFile(@"..\..\..\TextFile1.txt");
foreach (string line in rFile.Customer)
{
    Console.WriteLine(line);
}
Console.WriteLine("----------------------------");
foreach (string line in rFile.Customer(StringSearchOption.Contains, true))
{
    Console.WriteLine(line);
}
```

保存文件，然后按 `Ctrl+F5` 生成并运行应用程序。

调用动态语言库

以下演练创建的项目将访问以动态语言 IronPython 编写的库。

创建自定义动态类

在 Visual Studio 中，选择“文件”>“新建”>“项目”。 在“创建新项目”对话框中，选择 C#，然后依次选择“控制台应用程序”和“下一步”。 在“配置新项目”对话框中，输入 `DynamicIronPythonSample` 作为“项目名称”，然后选择“下一步”。 在“其他信息”对话框中，为“目标框架”选择“.NET 7.0 (当前)”，然后选择“创建”。 安装 IronPython NuGet 包。 编辑 Program.cs 文件。 在文件的顶部，添加以下代码以从 IronPython 库和 `System.Linq` 命名空间导入 `Microsoft.Scripting.Hosting` 和 `IronPython.Hosting` 命名空间。

C#

```
using System.Linq;
using Microsoft.Scripting.Hosting;
using IronPython.Hosting;
```

在 Main 方法中，添加以下代码以创建用于托管 IronPython 库的新 `Microsoft.Scripting.Hosting.ScriptRuntime` 对象。 `ScriptRuntime` 对象加载 IronPython 库模块 random.py。

C#

```
// Set the current directory to the IronPython libraries.
System.IO.Directory.SetCurrentDirectory(
   Environment.GetFolderPath(Environment.SpecialFolder.ProgramFiles) +
   @"\IronPython 2.7\Lib");

// Create an instance of the random.py IronPython library.
Console.WriteLine("Loading random.py");
ScriptRuntime py = Python.CreateRuntime();
dynamic random = py.UseFile("random.py");
Console.WriteLine("random.py loaded.");
```

在用于加载 random.py 模块的代码之后，添加以下代码以创建一个整数数组。 数组传递给 random.py 模块的 `shuffle` 方法，该方法对数组中的值进行随机排序。

C#

```
// Initialize an enumerable set of integers.
int[] items = Enumerable.Range(1, 7).ToArray();

// Randomly shuffle the array of integers by using IronPython.
for (int i = 0; i < 5; i++)
{
```

```
    random.shuffle(items);
    foreach (int item in items)
    {
        Console.WriteLine(item);
    }
    Console.WriteLine("--------------------");
}
```

保存文件，然后按 Ctrl+F5 生成并运行应用程序。

请参阅

- System.Dynamic
- System.Dynamic.DynamicObject
- 使用类型 dynamic
- dynamic
- 实现动态接口（可从 Microsoft TechNet 下载 PDF）

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

Reduce memory allocations using new C# features

Article • 10/17/2023

> **ⓘ Important**
>
> The techniques described in this section improve performance when applied to *hot paths* in your code. *Hot paths* are those sections of your codebase that are executed often and repeatedly in normal operations. Applying these techniques to code that isn't often executed will have minimal impact. Before making any changes to improve performance, it's critical to measure a baseline. Then, analyze that baseline to determine where memory bottlenecks occur. You can learn about many cross platform tools to measure your application's performance in the section on Diagnostics and instrumentation. You can practice a profiling session in the tutorial to Measure memory usage in the Visual Studio documentation.
>
> Once you've measured memory usage and have determined that you can reduce allocations, use the techniques in this section to reduce allocations. After each successive change, measure memory usage again. Make sure each change has a positive impact on the memory usage in your application.

Performance work in .NET often means removing allocations from your code. Every block of memory you allocate must eventually be freed. Fewer allocations reduce time spent in garbage collection. It allows for more predictable execution time by removing garbage collections from specific code paths.

A common tactic to reduce allocations is to change critical data structures from `class` types to `struct` types. This change impacts the semantics of using those types. Parameters and returns are now passed by value instead of by reference. The cost of copying a value is negligible if the types are small, three words or less (considering one word being of natural size of one integer). It's measurable and can have real performance impact for larger types. To combat the effect of copying, developers can pass these types by `ref` to get back the intended semantics.

The C# `ref` features give you the ability to express the desired semantics for `struct` types without negatively impacting their overall usability. Prior to these enhancements, developers needed to resort to `unsafe` constructs with pointers and raw memory to achieve the same performance impact. The compiler generates *verifiably safe code* for the new `ref` related features. *Verifiably safe code* means the compiler detects possible

buffer overruns or accessing unallocated or freed memory. The compiler detects and prevents some errors.

Pass and return by reference

Variables in C# store *values*. In `struct` types, the value is the contents of an instance of the type. In `class` types, the value is a reference to a block of memory that stores an instance of the type. Adding the `ref` modifier means that the variable stores the *reference* to the value. In `struct` types, the reference points to the storage containing the value. In `class` types, the reference points to the storage containing the reference to the block of memory.

In C#, parameters to methods are *passed by value*, and return values are *return by value*. The *value* of the argument is passed to the method. The *value* of the return argument is the return value.

The `ref`, `in`, `ref readonly`, or `out` modifier indicates that the argument is *passed by reference*. A *reference* to the storage location is passed to the method. Adding `ref` to the method signature means the return value is *returned by reference*. A *reference* to the storage location is the return value.

You can also use *ref assignment* to have a variable refer to another variable. A typical assignment copies the *value* of the right hand side to the variable on the left hand side of the assignment. A *ref assignment* copies the memory location of the variable on the right hand side to the variable on the left hand side. The `ref` now refers to the original variable:

C#

```
int anInteger = 42; // assignment.
ref int location = ref anInteger; // ref assignment.
ref int sameLocation = ref location; // ref assignment

Console.WriteLine(location); // output: 42

sameLocation = 19; // assignment

Console.WriteLine(anInteger); // output: 19
```

When you *assign* a variable, you change its *value*. When you *ref assign* a variable, you change what it refers to.

You can work directly with the storage for values using `ref` variables, pass by reference, and ref assignment. Scope rules enforced by the compiler ensure safety when working directly with storage.

The `ref readonly` and `in` modifiers both indicate that the argument should be passed by reference and can't be reassigned in the method. The difference is that `ref readonly` indicates that the method uses the parameter as a variable. The method might capture the parameter, or it might return the parameter by readonly reference. In those cases, you should use the `ref readonly` modifier. Otherwise, the `in` modifier offers more flexibility. You don't need to add the `in` modifier to an argument for an `in` parameter, so you can update existing API signatures safely using the `in` modifier. The compiler issues a warning if you don't add either the `ref` or `in` modifier to an argument for a `ref readonly` parameter.

Ref safe context

C# includes rules for `ref` expressions to ensure that a `ref` expression can't be accessed where the storage it refers to is no longer valid. Consider the following example:

C#

```
public ref int CantEscape()
{
    int index = 42;
    return ref index; // Error: index's ref safe context is the body of
CantEscape
}
```

The compiler reports an error because you can't return a reference to a local variable from a method. The caller can't access the storage being referred to. The *ref safe context* defines the scope in which a `ref` expression is safe to access or modify. The following table lists the *ref safe contexts* for variable types. `ref` fields can't be declared in a `class` or a non-ref `struct`, so those rows aren't in the table:

Expand table

Declaration	***ref safe context***
non-ref local	block where local is declared
non-ref parameter	current method
`ref`, `ref readonly`, `in` parameter	calling method

Declaration	*ref safe context*
`out` parameter	current method
`class` field	calling method
non-ref `struct` field	current method
`ref` field of `ref struct`	calling method

A variable can be `ref` returned if its *ref safe context* is the calling method. If its *ref safe context* is the current method or a block, `ref` return is disallowed. The following snippet shows two examples. A member field can be accessed from the scope calling a method, so a class or struct field's *ref safe context* is the calling method. The *ref safe context* for a parameter with the `ref`, or `in` modifiers is the entire method. Both can be `ref` returned from a member method:

C#

```
private int anIndex;

public ref int RetrieveIndexRef()
{
    return ref anIndex;
}

public ref int RefMin(ref int left, ref int right)
{
    if (left < right)
        return ref left;
    else
        return ref right;
}
```

> **Note**
>
> When the `ref readonly` or `in` modifier is applied to a parameter, that parameter can be returned by `ref readonly`, not `ref`.

The compiler ensures that a reference can't escape its *ref safe context*. You can use `ref` parameters, `ref return`, and `ref` local variables safely because the compiler detects if you've accidentally written code where a `ref` expression could be accessed when its storage isn't valid.

Safe context and ref structs

`ref struct` types require more rules to ensure they can be used safely. A `ref struct` type can include `ref` fields. That requires the introduction of a *safe context*. For most types, the *safe context* is the calling method. In other words, a value that's not a `ref struct` can always be returned from a method.

Informally, the *safe context* for a `ref struct` is the scope where all of its `ref` fields can be accessed. In other words, it's the intersection of the *ref safe context* of all its `ref` fields. The following method returns a `ReadOnlySpan<char>` to a member field, so its *safe context* is the method:

C#

```
private string longMessage = "This is a long message";

public ReadOnlySpan<char> Safe()
{
    var span = longMessage.AsSpan();
    return span;
}
```

In contrast, the following code emits an error because the `ref field` member of the `Span<int>` refers to the stack allocated array of integers. It can't escape the method:

C#

```
public Span<int> M()
{
    int length = 3;
    Span<int> numbers = stackalloc int[length];
    for (var i = 0; i < length; i++)
    {
        numbers[i] = i;
    }
    return numbers; // Error! numbers can't escape this method.
}
```

Unify memory types

The introduction of System.Span<T> and System.Memory<T> provide a unified model for working with memory. System.ReadOnlySpan<T> and System.ReadOnlyMemory<T> provide readonly versions for accessing memory. They all provide an abstraction over a block of memory storing an array of similar elements. The difference is that `Span<T>` and

`ReadOnlySpan<T>` are `ref struct` types whereas `Memory<T>` and `ReadOnlyMemory<T>` are `struct` types. Spans contain a `ref field`. Therefore instances of a span can't leave its *safe context*. The *safe context* of a `ref struct` is the *ref safe context* of its `ref field`. The implementation of `Memory<T>` and `ReadOnlyMemory<T>` remove this restriction. You use these types to directly access memory buffers.

Improve performance with ref safety

Using these features to improve performance involves these tasks:

- *Avoid allocations*: When you change a type from a `class` to a `struct`, you change how it's stored. Local variables are stored on the stack. Members are stored inline when the container object is allocated. This change means fewer allocations and that decreases the work the garbage collector does. It might also decrease memory pressure so the garbage collector runs less often.
- *Preserve reference semantics*: Changing a type from a `class` to a `struct` changes the semantics of passing a variable to a method. Code that modified the state of its parameters needs modification. Now that the parameter is a `struct`, the method is modifying a copy of the original object. You can restore the original semantics by passing that parameter as a `ref` parameter. After that change, the method modifies the original `struct` again.
- *Avoid copying data*: Copying larger `struct` types can impact performance in some code paths. You can also add the `ref` modifier to pass larger data structures to methods by reference instead of by value.
- *Restrict modifications*: When a `struct` type is passed by reference, the called method could modify the state of the struct. You can replace the `ref` modifier with the `ref readonly` or `in` modifiers to indicate that the argument can't be modified. Prefer `ref readonly` when the method captures the parameter or returns it by readonly reference. You can also create `readonly struct` types or `struct` types with `readonly` members to provide more control over what members of a `struct` can be modified.
- *Directly manipulate memory*: Some algorithms are most efficient when treating data structures as a block of memory containing a sequence of elements. The `Span` and `Memory` types provide safe access to blocks of memory.

None of these techniques require `unsafe` code. Used wisely, you can get performance characteristics from safe code that was previously only possible by using unsafe techniques. You can try the techniques yourself in the tutorial on reducing memory allocations.

教程：通过 `ref` safety 减少内存分配

项目 • 2023/06/05

通常，.NET 应用程序的性能优化涉及两种方法。 首先，减少堆分配的数量和大小。 其次，减少复制数据的频率。 Visual Studio 提供了出色的工具，可帮助分析应用程序如何使用内存。 确定应用在何处进行了不必要的分配后，可以进行更改以最大程度地减少这些分配。 将 `class` 类型转换为 `struct` 类型。 使用 `ref` safety 功能来保留语义并最大程度地减少额外的复制。

使用 Visual Studio 17.5 获得本教程的最佳体验。 用于分析内存使用情况的 .NET 对象分配工具是 Visual Studio 的一部分。 可以使用 Visual Studio Code 和命令行来运行应用程序并执行所有更改。 但是，你将无法看到更改的分析结果。

你将使用的应用程序是 IoT 应用程序的模拟，该应用程序监视多个传感器，以确定入侵者是否已进入具有重要信息的机密库。 IoT 传感器不断发送数据，用于度量空气中氧气 (O2) 和二氧化碳 (CO2) 的混合。 它们还会报告温度和相对湿度。 其中每个值一直略有波动。 然而，当一个人进入房间时，变化会更大一点，而且总是在同一个方向上：氧气减少，二氧化碳增加，温度上升，相对湿度也上升。 当传感器合并显示增加时，将触发入侵者警报。

在本教程中，你将运行应用程序，对内存分配进行度量，然后通过减少分配数来提高性能。 示例浏览器中提供了源代码。

探索初学者应用程序

下载应用程序并运行初学者示例。 初学者应用程序可以正常工作，但由于它在每个度量周期分配许多小型对象，因此其性能会随着时间推移而缓慢下降。

控制台

```
Press <return> to start simulation

Debounced measurements:
    Temp:      67.332
    Humidity:  41.077%
    Oxygen:    21.097%
    CO2 (ppm): 404.906
Average measurements:
    Temp:      67.332
    Humidity:  41.077%
    Oxygen:    21.097%
    CO2 (ppm): 404.906

Debounced measurements:
```

```
    Temp:      67.349
    Humidity:  46.605%
    Oxygen:    20.998%
    CO2 (ppm): 408.707
Average measurements:
    Temp:      67.349
    Humidity:  46.605%
    Oxygen:    20.998%
    CO2 (ppm): 408.707
```

删除了许多行。

控制台

```
Debounced measurements:
    Temp:      67.597
    Humidity:  46.543%
    Oxygen:    19.021%
    CO2 (ppm): 429.149
Average measurements:
    Temp:      67.568
    Humidity:  45.684%
    Oxygen:    19.631%
    CO2 (ppm): 423.498
Current intruders: 3
Calculated intruder risk: High

Debounced measurements:
    Temp:      67.602
    Humidity:  46.835%
    Oxygen:    19.003%
    CO2 (ppm): 429.393
Average measurements:
    Temp:      67.568
    Humidity:  45.684%
    Oxygen:    19.631%
    CO2 (ppm): 423.498
Current intruders: 3
Calculated intruder risk: High
```

可以浏览代码以了解应用程序的工作原理。 主程序运行模拟。 按 `<Enter>` 后，会创建一个房间，并收集一些初始基线数据：

C#

```
Console.WriteLine("Press <return> to start simulation");
Console.ReadLine();
var room = new Room("gallery");
var r = new Random();

int counter = 0;
```

```
room.TakeMeasurements(
    m =>
    {
        Console.WriteLine(room.Debounce);
        Console.WriteLine(room.Average);
        Console.WriteLine();
        counter++;
        return counter < 20000;
    });
```

建立基线数据后，会在房间上运行模拟，其中随机数生成器会确定是否有入侵者进入房间：

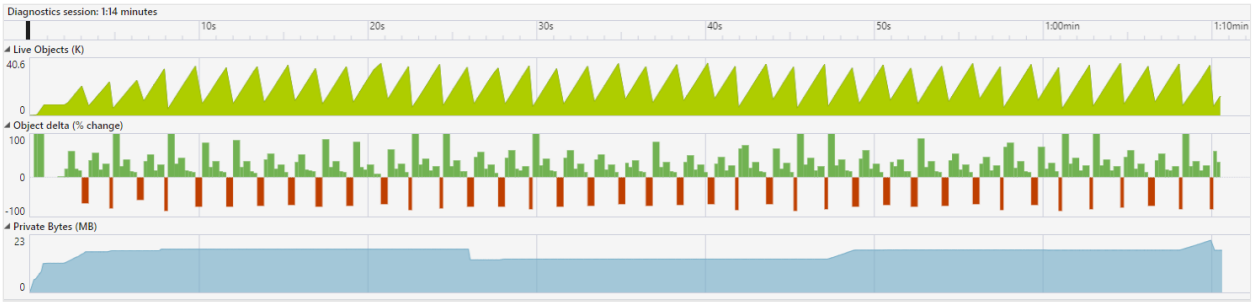
C#

```
counter = 0;
room.TakeMeasurements(
    m =>
    {
        Console.WriteLine(room.Debounce);
        Console.WriteLine(room.Average);
        room.Intruders += (room.Intruders, r.Next(5)) switch
        {
            ( > 0, 0) => -1,
            ( < 3, 1) => 1,
            _ => 0
        };

        Console.WriteLine($"Current intruders: {room.Intruders}");
        Console.WriteLine($"Calculated intruder risk: {room.RiskStatus}");
        Console.WriteLine();
        counter++;
        return counter < 200000;
    });
```

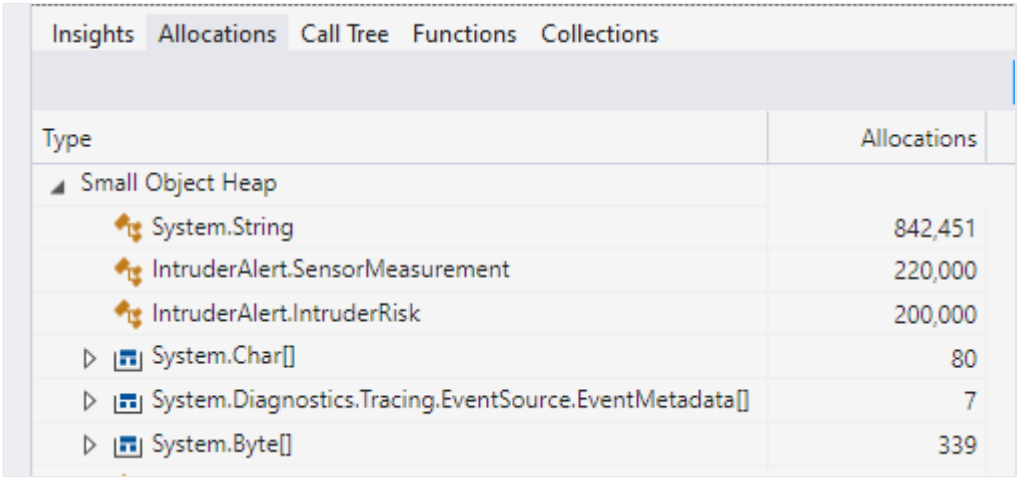
其他类型包含度量值、防反跳度量值（即最近 50 个度量值的平均值）以及所有度量值的平均值。

接下来，使用 .NET 对象分配工具运行应用程序。 请确保使用的是 `Release` 版本，而不是 `Debug` 版本。 在“调试”菜单上，打开“性能探查器”。 仅选中“.NET 对象分配跟踪”选项。 运行应用程序以完成。 探查器度量对象分配，并报告分配和垃圾回收周期。 应看到类似于以下图像的图：



上图显示，尽量减少分配将带来性能优势。 在实时对象图中可以看到锯齿图像。 这说明创建了大量对象，而这些对象很快就变为垃圾。 稍后会收集这些对象，如对象增量图中所示。 向下的红色条形表示垃圾回收周期。

接下来，请查看图表下方的“分配”选项卡。 下表显示分配最多的类型：



System.String 类型占分配最多。 最重要的任务应该是尽量减少字符串分配的频率。 此应用程序不断将大量格式化输出打印到控制台。 对于此模拟，我们希望保留消息，因此我们将专注于接下来的两行：`SensorMeasurement` 类型和 `IntruderRisk` 类型。

双击 `SensorMeasurement` 行。 可以看到，所有分配都发生在 `static` 方法 `SensorMeasurement.TakeMeasurement` 中。 可以在以下代码片段中看到该方法：

C#

```
public static SensorMeasurement TakeMeasurement(string room, int intruders)
{
    return new SensorMeasurement
    {
        CO2 = (CO2Concentration + intruders * 10) + (20 *
generator.NextDouble() - 10.0),
        O2 = (O2Concentration - intruders * 0.01) + (0.005 *
generator.NextDouble() - 0.0025),
        Temperature = (TemperatureSetting + intruders * 0.05) + (0.5 *
generator.NextDouble() - 0.25),
        Humidity = (HumiditySetting + intruders * 0.005) + (0.20 *
generator.NextDouble() - 0.10),
        Room = room,
        TimeRecorded = DateTime.Now
```

```
    };
}
```

每次度量都会分配一个新的 `SensorMeasurement` 对象，该对象为 `class` 类型。 创建的每个 `SensorMeasurement` 都会导致堆分配。

将类更改为结构

以下代码演示了 `SensorMeasurement` 的初始声明：

C#

```
public class SensorMeasurement
{
    private static readonly Random generator = new Random();

    public static SensorMeasurement TakeMeasurement(string room, int
intruders)
    {
        return new SensorMeasurement
        {
            CO2 = (CO2Concentration + intruders * 10) + (20 *
generator.NextDouble() - 10.0),
            O2 = (O2Concentration - intruders * 0.01) + (0.005 *
generator.NextDouble() - 0.0025),
            Temperature = (TemperatureSetting + intruders * 0.05) + (0.5 *
generator.NextDouble() - 0.25),
            Humidity = (HumiditySetting + intruders * 0.005) + (0.20 *
generator.NextDouble() - 0.10),
            Room = room,
            TimeRecorded = DateTime.Now
        };
    }

    private const double CO2Concentration = 409.8; // increases with people.
    private const double O2Concentration = 0.2100; // decreases
    private const double TemperatureSetting = 67.5; // increases
    private const double HumiditySetting = 0.4500; // increases

    public required double CO2 { get; init; }
    public required double O2 { get; init; }
    public required double Temperature { get; init; }
    public required double Humidity { get; init; }
    public required string Room { get; init; }
    public required DateTime TimeRecorded { get; init; }

    public override string ToString() => $"""
            Room: {Room} at {TimeRecorded}:
                Temp:      {Temperature:F3}
                Humidity:  {Humidity:P3}
                Oxygen:    {O2:P3}
```

```
                CO2 (ppm): {CO2:F3}
            """;
}
```

类型最初创建为 `class`，因为它包含许多 `double` 度量值。 它比要在热路径中复制的要大。 但是，该决定意味着大量的分配。 将类型从 `class` 更改为 `struct`。

从 `class` 更改为 `struct` 会引起一些编译器错误，因为原始代码在几个位置使用了 `null` 引用检查。 第一个在 `AddMeasurement` 方法的 `DebounceMeasurement` 类中：

C#

```
public void AddMeasurement(SensorMeasurement datum)
{
    int index = totalMeasurements % debounceSize;
    recentMeasurements[index] = datum;
    totalMeasurements++;
    double sumCO2 = 0;
    double sumO2 = 0;
    double sumTemp = 0;
    double sumHumidity = 0;
    for (int i = 0; i < debounceSize; i++)
    {
        if (recentMeasurements[i] is not null)
        {
            sumCO2 += recentMeasurements[i].CO2;
            sumO2+= recentMeasurements[i].O2;
            sumTemp+= recentMeasurements[i].Temperature;
            sumHumidity += recentMeasurements[i].Humidity;
        }
    }
    O2 = sumO2 / ((totalMeasurements > debounceSize) ? debounceSize :
totalMeasurements);
    CO2 = sumCO2 / ((totalMeasurements > debounceSize) ? debounceSize :
totalMeasurements);
    Temperature = sumTemp / ((totalMeasurements > debounceSize) ?
debounceSize : totalMeasurements);
    Humidity = sumHumidity / ((totalMeasurements > debounceSize) ?
debounceSize : totalMeasurements);
}
```

`DebounceMeasurement` 类型包含具有 50 个度量值的数组。 传感器的读数报告为最近 50 个度量值的平均值。 这可以减少读数中的干扰。 在获取整整 50 个读数之前，这些值为 `null`。 代码检查 `null` 引用以报告系统启动时的正确平均值。 将 `SensorMeasurement` 类型更改为结构后，必须使用不同的测试。 `SensorMeasurement` 类型包括房间标识符的 `string`，因此可以改用该测试：

C#

```
if (recentMeasurements[i].Room is not null)
```

其他三个编译器错误都位于在房间中重复执行度量的方法中：

C#

```
public void TakeMeasurements(Func<SensorMeasurement, bool>
MeasurementHandler)
{
    SensorMeasurement? measure = default;
    do {
        measure = SensorMeasurement.TakeMeasurement(Name, Intruders);
        Average.AddMeasurement(measure);
        Debounce.AddMeasurement(measure);
    } while (MeasurementHandler(measure));
}
```

在初学者方法中，`SensorMeasurement` 的局部变量是可以为 null 的引用：

C#

```
SensorMeasurement? measure = default;
```

现在，`SensorMeasurement` 为 `struct`，而不是 `class`，可为空是可以为 null 的值类型。可以将声明更改为值类型，以修复剩余的编译器错误：

C#

```
SensorMeasurement measure = default;
```

解决编译器错误后，应检查代码以确保语义未更改。 由于 `struct` 类型是通过值传递的，因此在方法返回后，对方法参数所做的修改将不可见。

> **ⓘ 重要**
>
> 将类型从 `class` 更改为 `struct` 可能会更改程序的语义。 将 `class` 类型传递给方法时，方法中所做的任何更改都对参数进行更改。 将 `struct` 类型传递给方法时，方法中所做的更改将对参数的副本进行更改。 这意味着任何通过设计修改其参数的方法都应更新为在已从 `class` 更改为 `struct` 的任何参数类型上使用 `ref` 修饰符。

`SensorMeasurement` 类型不包含任何更改状态的方法，因此在本示例中不考虑此问题。 可以通过向 `SensorMeasurement` 结构添加 `readonly` 修饰符来证明这一点：

C#

```
public readonly struct SensorMeasurement
```

编译器强制实施 `SensorMeasurement` 结构的 `readonly` 性质。 如果检查代码时错过了某个修改状态的方法，编译器会告诉你。 应用仍会生成，且不会出错，因此此类型为 `readonly`。 在将类型从 `class` 更改为 `struct` 时添加 `readonly` 修饰符有助于查找修改 `struct` 状态的成员。

避免创建副本

你已从应用中删除了大量不必要的分配。 `SensorMeasurement` 类型不会显示在表中的任何位置。

现在，每次将 `SensorMeasurement` 结构用作参数或返回值时，它都会执行额外的工作复制该结构。 `SensorMeasurement` 结构包含四个双精度值、一个 DateTime 和一个 `string`。 该结构明显大于引用。 让我们将 `ref` 或 `in` 修饰符添加到使用 `SensorMeasurement` 类型的位置。

下一步是查找返回度量值或将度量值作为参数的方法，并尽可能使用引用。 从 `SensorMeasurement` 结构开始。 静态 `TakeMeasurement` 方法创建并返回一个新的 `SensorMeasurement`：

C#

```
public static SensorMeasurement TakeMeasurement(string room, int intruders)
{
    return new SensorMeasurement
    {
        CO2 = (CO2Concentration + intruders * 10) + (20 *
generator.NextDouble() - 10.0),
        O2 = (O2Concentration - intruders * 0.01) + (0.005 *
generator.NextDouble() - 0.0025),
        Temperature = (TemperatureSetting + intruders * 0.05) + (0.5 *
generator.NextDouble() - 0.25),
        Humidity = (HumiditySetting + intruders * 0.005) + (0.20 *
generator.NextDouble() - 0.10),
        Room = room,
        TimeRecorded = DateTime.Now
    };
}
```

我们将其保留原样，通过值返回。 如果尝试通过 `ref` 返回，则会收到编译器错误。 无法将 `ref` 返回到方法中本地创建的新结构。 不可变结构的设计意味着只能在构造时设置度

量值。 此方法必须创建新的度量结构。

让我们再看一下 `DebounceMeasurement.AddMeasurement`。 应将 `in` 修饰符添加到 `measurement` 参数：

C#

```
public void AddMeasurement(in SensorMeasurement datum)
{
    int index = totalMeasurements % debounceSize;
    recentMeasurements[index] = datum;
    totalMeasurements++;
    double sumCO2 = 0;
    double sumO2 = 0;
    double sumTemp = 0;
    double sumHumidity = 0;
    for (int i = 0; i < debounceSize; i++)
    {
        if (recentMeasurements[i].Room is not null)
        {
            sumCO2 += recentMeasurements[i].CO2;
            sumO2+= recentMeasurements[i].O2;
            sumTemp+= recentMeasurements[i].Temperature;
            sumHumidity += recentMeasurements[i].Humidity;
        }
    }
    O2 = sumO2 / ((totalMeasurements > debounceSize) ? debounceSize :
totalMeasurements);
    CO2 = sumCO2 / ((totalMeasurements > debounceSize) ? debounceSize :
totalMeasurements);
    Temperature = sumTemp / ((totalMeasurements > debounceSize) ?
debounceSize : totalMeasurements);
    Humidity = sumHumidity / ((totalMeasurements > debounceSize) ?
debounceSize : totalMeasurements);
}
```

这可以保存一个复制操作。 `in` 参数是对调用方已创建的副本的引用。 还可以使用 `Room` 类型中的 `TakeMeasurement` 方法保存副本。 此方法演示了在通过 `ref` 传递参数时编译器如何提供安全性。 `Room` 类型中的初始 `TakeMeasurement` 方法采用 `Func<SensorMeasurement, bool>` 的参数。 如果尝试向该声明添加 `in` 或 `ref` 修饰符，编译器将报告错误。 不能将 `ref` 参数传递给 Lambda 表达式。 编译器无法保证调用的表达式不会复制引用。 如果 Lambda 表达式捕获引用，则引用的生存期可能比它所引用的值长。 在 ref safe to escape 范围之外进行访问会导致内存损坏。 `ref` 安全规则不允许。 可以在 ref safety 功能概述中了解详细信息。

保留语义

最后一组更改不会对此应用程序的性能产生重大影响，因为这些类型不是在热路径中创建的。 这些更改演示了可在性能优化中使用的一些其他方法。 让我们看一下初始 `Room` 类：

C#

```
public class Room
{
    public AverageMeasurement Average { get; } = new ();
    public DebounceMeasurement Debounce { get; } = new ();
    public string Name { get; }

    public IntruderRisk RiskStatus
    {
        get
        {
            var CO2Variance = (Debounce.CO2 - Average.CO2) > 10.0 / 4;
            var O2Variance = (Average.O2 - Debounce.O2) > 0.005 / 4.0;
            var TempVariance = (Debounce.Temperature - Average.Temperature)
> 0.05 / 4.0;
            var HumidityVariance = (Debounce.Humidity - Average.Humidity) >
0.20 / 4;
            IntruderRisk risk = IntruderRisk.None;
            if (CO2Variance) { risk++; }
            if (O2Variance) { risk++; }
            if (TempVariance) { risk++; }
            if (HumidityVariance) { risk++; }
            return risk;
        }
    }

    public int Intruders { get; set; }


    public Room(string name)
    {
        Name = name;
    }

    public void TakeMeasurements(Func<SensorMeasurement, bool>
MeasurementHandler)
    {
        SensorMeasurement? measure = default;
        do {
            measure = SensorMeasurement.TakeMeasurement(Name, Intruders);
            Average.AddMeasurement(measure);
            Debounce.AddMeasurement(measure);
        } while (MeasurementHandler(measure));
    }
}
```

此类型包含多个属性。 有些是 `class` 类型。 创建 `Room` 对象涉及多个分配。 一个用于 `Room` 本身，一个用于它包含的 `class` 类型的每个成员。 可以将其中两个属性从 `class` 类型转换为 `struct` 类型：`DebounceMeasurement` 和 `AverageMeasurement` 类型。 让我们使用这两种类型完成该转换。

将 `DebounceMeasurement` 类型从 `class` 更改为 `struct`。 这会导致编译器错误 `CS8983: A 'struct' with field initializers must include an explicitly declared constructor`。 可以通过添加空的无参数构造函数来解决此问题：

```
C#

public DebounceMeasurement() { }
```

有关此要求的详细信息，请参阅有关的语言参考文章。

Object.ToString() 替代不会修改结构的任何值。 可向该方法声明添加 `readonly` 修饰符。 `DebounceMeasurement` 类型是可变的，因此需要注意修改不会影响已丢弃的副本。 `AddMeasurement` 方法会修改对象的状态。 它在 `TakeMeasurements` 方法中从 `Room` 类中调用。 你希望在调用该方法后保留这些更改。 可以更改 `Room.Debounce` 属性以返回对 `DebounceMeasurement` 类型的单个实例的引用：

```
C#

private DebounceMeasurement debounce = new();
public ref readonly DebounceMeasurement Debounce { get { return ref
debounce; } }
```

上一个示例中有一些更改。 首先，属性是只读属性，它返回对此房间拥有的实例的只读引用。 它现在由实例化 `Room` 对象时初始化的声明字段提供支持。 进行这些更改后，你将更新 `AddMeasurement` 方法的实现。 它使用专用支持字段 `debounce`，而不是只读属性 `Debounce`。 这样，这些更改将发生在初始化期间创建的单个实例上。

相同的方法适用于 `Average` 属性。 首先，将 `AverageMeasurement` 类型从 `class` 修改为 `struct`，并在 `ToString` 方法上添加 `readonly` 修饰符：

```
C#

namespace IntruderAlert;

public struct AverageMeasurement
{
    private double sumCO2 = 0;
    private double sumO2 = 0;
    private double sumTemperature = 0;
```

```
    private double sumHumidity = 0;
    private int totalMeasurements = 0;

    public AverageMeasurement() { }

    public readonly double CO2 => sumCO2 / totalMeasurements;
    public readonly double O2 => sumO2 / totalMeasurements;
    public readonly double Temperature => sumTemperature /
totalMeasurements;
    public readonly double Humidity => sumHumidity / totalMeasurements;

    public void AddMeasurement(in SensorMeasurement datum)
    {
        totalMeasurements++;
        sumCO2 += datum.CO2;
        sumO2 += datum.O2;
        sumTemperature += datum.Temperature;
        sumHumidity+= datum.Humidity;
    }

    public readonly override string ToString() => $"""
        Average measurements:
            Temp:      {Temperature:F3}
            Humidity:  {Humidity:P3}
            Oxygen:    {O2:P3}
            CO2 (ppm): {CO2:F3}
        """;
}
```

然后，按照对 `Debounce` 属性使用的相同方法修改 `Room` 类。 `Average` 属性将 `readonly ref` 返回给平均度量值的专用字段。 `AddMeasurement` 方法修改内部字段。

C#

```
private AverageMeasurement average = new();
public  ref readonly AverageMeasurement Average { get { return ref average;
} }
```

避免装箱

还有最后一个更改可以提高性能。 主程序打印房间的统计信息，包括风险评估：

C#

```
Console.WriteLine($"Current intruders: {room.Intruders}");
Console.WriteLine($"Calculated intruder risk: {room.RiskStatus}");
```

对生成的 `ToString` 调用将枚举值装箱。 可以通过在 `Room` 类中编写一个替代来避免这种情况，该替代根据估计风险的值设置字符串的格式：

C#

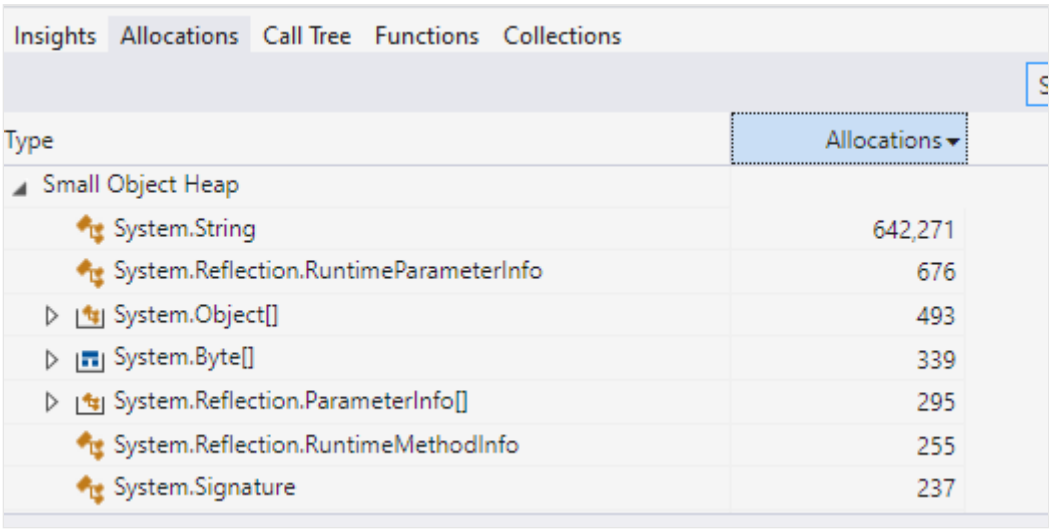
```
public override string ToString() =>
    $"Calculated intruder risk: {RiskStatus switch
    {
        IntruderRisk.None => "None",
        IntruderRisk.Low => "Low",
        IntruderRisk.Medium => "Medium",
        IntruderRisk.High => "High",
        IntruderRisk.Extreme => "Extreme",
        _ => "Error!"
    }}, Current intruders: {Intruders.ToString()}";
```

然后，修改主程序中的代码以调用此新 `ToString` 方法：

C#

```
Console.WriteLine(room.ToString());
```

使用探查器运行应用，并查看更新的表了解分配。



你已删除大量分配，并为应用提供了性能提升。

在应用程序中使用 ref safety

这些方法是低级别性能优化。 当应用于热路径时，以及你度量了更改前后的影响时，它们可以提高应用程序的性能。 在大多数情况下，你将遵循的周期是：

- 度量分配：确定分配最多的类型，以及何时可以减少堆分配。

- 将类转换为结构：很多时候，类型可以从 `class` 转换为 `struct`。 你的应用使用堆栈空间，而不是进行堆分配。
- 保留语义：将 `class` 转换为 `struct` 可能会影响参数和返回值的语义。 任何修改其参数的方法现在都应使用 `ref` 修饰符标记这些参数。 这可确保对正确的对象进行修改。 同样，如果属性或方法返回值应由调用方修改，则应使用 `ref` 修饰符标记该返回值。
- 避免复制：将大型结构作为参数传递时，可以使用 `in` 修饰符标记参数。 可以传递字节较少的引用，并确保方法不会修改原始值。 还可以通过 `readonly ref` 返回值，以返回无法修改的引用。

使用这些方法可以提高代码热路径的性能。

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

教程：编写自定义字符串内插处理程序

项目 • 2025/01/08

在本教程中，您将学习如何：

- ✓ 实现字符串内插处理程序模式
- ✓ 在字符串内插操作中与接收方交互。
- ✓ 向字符串内插处理程序添加自变量
- ✓ 了解用于字符串内插的新程序库功能

先决条件

需要将计算机设置为运行 .NET。 Visual Studio 2022 或 .NET SDK 中提供了 C# 编译器。

本教程假定你熟悉 C# 和 .NET，包括 Visual Studio 或 .NET CLI。

可以编写自定义 *内插字符串处理程序*。 内插字符串处理程序是处理内插字符串中的占位符表达式的类型。 如果没有自定义处理程序，占位符的处理方式类似于 String.Format。 每个占位符的格式设置为文本，然后连接组件以形成生成的字符串。

您可以为任何需要使用结果字符串信息的场景编写处理程序。 会被使用吗？ 格式有哪些约束？ 一些示例包括：

- 可能不需要任何生成的字符串都大于某些限制，例如 80 个字符。 可以处理内插字符串以填充固定长度缓冲区，并在达到该缓冲区长度后停止处理。
- 你可能具有表格格式，并且每个占位符必须具有固定长度。 自定义处理程序可以强制实施，而不是强制所有客户端代码一致。

在本教程中，将为其中一个核心性能方案创建字符串内插处理程序：日志记录库。 根据配置的日志级别，不需要构造日志消息的工作。 如果日志记录已禁用，则不需要从内插字符串表达式构造字符串。 永远不会打印该消息，因此可以跳过任何字符串串联。 此外，不需要执行占位符中使用的任何表达式，包括生成堆栈跟踪。

内插字符串处理程序可以确定是否将使用格式化字符串，并且仅在需要时执行必要的工作。

初始实现

我们从支持不同级别的基本 `Logger` 类开始：

```
C#
```

```
public enum LogLevel
{
    Off,
    Critical,
    Error,
    Warning,
    Information,
    Trace
}

public class Logger
{
    public LogLevel EnabledLevel { get; init; } = LogLevel.Error;

    public void LogMessage(LogLevel level, string msg)
    {
        if (EnabledLevel < level) return;
        Console.WriteLine(msg);
    }
}
```

此 `Logger` 支持六个不同的级别。 当消息未通过日志级别筛选器时，将没有输出。 记录器的公用 API 接受一个已完全格式化的字符串作为消息。 创建字符串的所有工作都已完成。

实现处理程序模式

此步骤将生成一个内插字符串处理程序，用于重新创建当前行为。 内插字符串处理程序的类型必须具有以下特征：

- System.Runtime.CompilerServices.InterpolatedStringHandlerAttribute 应用于该类型。
- 具有两个 `int` 参数的构造函数，`literalLength` 和 `formattedCount`。 （允许更多参数）。
- 具有 `public void AppendLiteral(string s)` 签名的公共 `AppendLiteral` 方法。
- 具有 `public void AppendFormatted<T>(T t)` 签名的一般公共 `AppendFormatted` 方法。

在内部，生成器会创建格式化字符串，并为客户端提供一个成员来检索该字符串。 下面的代码演示了满足这些需求的 `LogInterpolatedStringHandler` 类型：

C#

```
[InterpolatedStringHandler]
public struct LogInterpolatedStringHandler
{
```

```
    // Storage for the built-up string
    StringBuilder builder;

    public LogInterpolatedStringHandler(int literalLength, int
formattedCount)
    {
        builder = new StringBuilder(literalLength);
        Console.WriteLine($"\tliteral length: {literalLength},
formattedCount: {formattedCount}");
    }

    public void AppendLiteral(string s)
    {
        Console.WriteLine($"\tAppendLiteral called: {{{s}}}");

        builder.Append(s);
        Console.WriteLine($"\tAppended the literal string");
    }

    public void AppendFormatted<T>(T t)
    {
        Console.WriteLine($"\tAppendFormatted called: {{{t}}} is of type
{typeof(T)}");

        builder.Append(t?.ToString());
        Console.WriteLine($"\tAppended the formatted object");
    }

    internal string GetFormattedText() => builder.ToString();
}
```

现在可以在 `Logger` 类中将重载添加到 `LogMessage`，以尝试使用新的内插字符串处理程序：

C#

```
public void LogMessage(LogLevel level, LogInterpolatedStringHandler builder)
{
    if (EnabledLevel < level) return;
    Console.WriteLine(builder.GetFormattedText());
}
```

无需删除原始 `LogMessage` 方法，当参数是内插字符串表达式时，编译器首选具有内插处理程序参数的方法，而不是具有 `string` 参数的方法。

可以使用以下代码作为主程序验证是否调用了新处理程序：

C#

```
var logger = new Logger() { EnabledLevel = LogLevel.Warning };
var time = DateTime.Now;

logger.LogMessage(LogLevel.Error, $"Error Level. CurrentTime: {time}. This
is an error. It will be printed.");
logger.LogMessage(LogLevel.Trace, $"Trace Level. CurrentTime: {time}. This
won't be printed.");
logger.LogMessage(LogLevel.Warning, "Warning Level. This warning is a
string, not an interpolated string expression.");
```

运行应用程序将生成类似于以下文本的输出：

PowerShell

```
        literal length: 65, formattedCount: 1
        AppendLiteral called: {Error Level. CurrentTime: }
        Appended the literal string
        AppendFormatted called: {10/20/2021 12:19:10 PM} is of type
System.DateTime
        Appended the formatted object
        AppendLiteral called: {. This is an error. It will be printed.}
        Appended the literal string
Error Level. CurrentTime: 10/20/2021 12:19:10 PM. This is an error. It will
be printed.
        literal length: 50, formattedCount: 1
        AppendLiteral called: {Trace Level. CurrentTime: }
        Appended the literal string
        AppendFormatted called: {10/20/2021 12:19:10 PM} is of type
System.DateTime
        Appended the formatted object
        AppendLiteral called: {. This won't be printed.}
        Appended the literal string
Warning Level. This warning is a string, not an interpolated string
expression.
```

通过输出跟踪，可以看到编译器如何添加代码来调用处理程序并生成字符串：

- 编译器添加一个调用来构造处理程序，并传递格式字符串中文本的总长度，以及占位符的数量。
- 编译器为字面字符串的每个部分以及每个占位符添加对 `AppendLiteral` 和 `AppendFormatted` 的调用。
- 编译器调用 `LogMessage` 方法，并使用 `CoreInterpolatedStringHandler` 作为参数。

最后，请注意，最后一个警告不会调用内插字符串处理程序。 自变量是一个 `string`，因此该调用使用一个字符串参数来调用另一个重载。

> **ⓘ 重要**

仅在绝对必要的情况下才使用 `ref struct` 来处理插值字符串。 使用 `ref struct` 将具有限制，因为它们必须存储在堆栈上。 例如，如果内插字符串孔包含 `await` 表达式，则它们将不起作用，因为编译器需要将处理程序存储在编译器生成的 `IAsyncStateMachine` 实现中。

向处理程序添加更多功能

上一版本的内插字符串处理程序实现了该模式。 为了避免处理每个占位符表达式，需要在处理程序中获取更多信息。 在本部分中，你将改进处理程序，以减少当构造的字符串未被写入日志时的工作量。 使用 System.Runtime.CompilerServices.InterpolatedStringHandlerArgumentAttribute 指定公共 API 的参数与处理程序构造函数参数之间的映射。 这为处理程序提供了确定是否应评估内插字符串所需的信息。

我们从对处理程序的更改开始。 首先，添加字段以跟踪处理程序是否已启用。 向构造函数添加两个参数：一个用于指定此消息的日志级别，另一个参数是对日志对象的引用：

C#

```
private readonly bool enabled;

public LogInterpolatedStringHandler(int literalLength, int formattedCount,
Logger logger, LogLevel logLevel)
{
    enabled = logger.EnabledLevel >= logLevel;
    builder = new StringBuilder(literalLength);
    Console.WriteLine($"\tliteral length: {literalLength}, formattedCount:
{formattedCount}");
}
```

接下来，使用该字段，使处理程序仅在使用最终字符串时追加文本或格式化对象：

C#

```
public void AppendLiteral(string s)
{
    Console.WriteLine($"\tAppendLiteral called: {{{s}}}");
    if (!enabled) return;

    builder.Append(s);
    Console.WriteLine($"\tAppended the literal string");
}

public void AppendFormatted<T>(T t)
{
    Console.WriteLine($"\tAppendFormatted called: {{{t}}} is of type
```

```
{typeof(T)}");
    if (!enabled) return;

    builder.Append(t?.ToString());
    Console.WriteLine($"\tAppended the formatted object");
}
```

接下来，需要更新 `LogMessage` 声明，以便编译器将附加参数传递给处理程序的构造函数。 这是使用处理程序自变量上的 System.Runtime.CompilerServices.InterpolatedStringHandlerArgumentAttribute 进行处理的：

C#

```
public void LogMessage(LogLevel level,
[InterpolatedStringHandlerArgument("", "level")]
LogInterpolatedStringHandler builder)
{
    if (EnabledLevel < level) return;
    Console.WriteLine(builder.GetFormattedText());
}
```

此属性指定了 `LogMessage` 的自变量列表，这些参数映射到所需的 `literalLength` 和 `formattedCount` 参数之后的参数。 空字符串（“”），指定接收方。 编译器将 `this` 所表示的 `Logger` 对象的值替换为处理程序构造函数的下一个参数。 编译器将 `level` 的值替换为以下参数。 可以为写入的任何处理程序提供任意数量的参数。 添加的参数是字符串参数。

可以使用相同的测试代码运行此版本。 这一次，你将看到以下结果：

PowerShell

```
        literal length: 65, formattedCount: 1
        AppendLiteral called: {Error Level. CurrentTime: }
        Appended the literal string
        AppendFormatted called: {10/20/2021 12:19:10 PM} is of type
System.DateTime
        Appended the formatted object
        AppendLiteral called: {. This is an error. It will be printed.}
        Appended the literal string
Error Level. CurrentTime: 10/20/2021 12:19:10 PM. This is an error. It will
be printed.
        literal length: 50, formattedCount: 1
        AppendLiteral called: {Trace Level. CurrentTime: }
        AppendFormatted called: {10/20/2021 12:19:10 PM} is of type
System.DateTime
        AppendLiteral called: {. This won't be printed.}
```

```
Warning Level. This warning is a string, not an interpolated string
expression.
```

可以看到正在调用 `AppendLiteral` 和 `AppendFormat` 方法，但它们未执行任何工作。 处理程序确定不需要最终字符串，因此处理程序不会生成它。 还有一些需要改进的地方。

首先，可以添加 `AppendFormatted` 的一个重载，该重载将自变量约束为实现 System.IFormattable 的一个类型。 此重载使调用方能够在占位符中添加格式字符串。 进行此更改时，我们还会将其他 `AppendFormatted` 和 `AppendLiteral` 方法的返回类型从 `void` 更改为 `bool`（如果其中任一方法具有不同的返回类型，则会出现编译错误）。 这种更改将实现短路。 方法返回 `false` 以指示应停止内插字符串表达式的处理。 返回 `true` 则指示应继续。 在此示例中，当不需要生成的字符串时，你将使用它来停止处理。 短路支持更精细的操作。 一旦表达式达到特定长度，就可以停止处理表达式，以支持固定长度缓冲区。 或者某些条件可能指示不需要剩余的元素。

C#

```
public void AppendFormatted<T>(T t, string format) where T : IFormattable
{
    Console.WriteLine($"\tAppendFormatted (IFormattable version) called: {t}
with format {{{format}}} is of type {typeof(T)},");

    builder.Append(t?.ToString(format, null));
    Console.WriteLine($"\tAppended the formatted object");
}
```

通过此添加，可以在内插字符串表达式中指定格式字符串：

C#

```
var time = DateTime.Now;

logger.LogMessage(LogLevel.Error, $"Error Level. CurrentTime: {time}. The
time doesn't use formatting.");
logger.LogMessage(LogLevel.Error, $"Error Level. CurrentTime: {time:t}. This
is an error. It will be printed.");
logger.LogMessage(LogLevel.Trace, $"Trace Level. CurrentTime: {time:t}. This
won't be printed.");
```

第一条消息上的 `:t` 指定当前时间的“短时间格式”。 上一个示例演示了 `AppendFormatted` 方法的一个重载，你可以为处理程序创建该重载。 无需为要设置格式的对象指定泛型参数。 你可能有更有效的方法可将创建的类型转换为字符串。 可以编写采用这些类型的 `AppendFormatted` 的重载，而不是泛型自变量。 编译器选取最佳重载。 运行时使用此技术将 System.Span<T> 转换为字符串输出。 可以添加一个整数参数来指定输出的对齐（带或不带 IFormattable）。 .NET 6 附带的

System.Runtime.CompilerServices.DefaultInterpolatedStringHandler 包含 AppendFormatted 的 9 个重载，用于不同的用途。 在为您的特定需求构建处理程序时，可以将其用作参考。

现在运行示例，你将看到，对于 `Trace` 消息，只调用了第一个 `AppendLiteral`：

PowerShell

```
        literal length: 60, formattedCount: 1
        AppendLiteral called: Error Level. CurrentTime:
        Appended the literal string
        AppendFormatted called: 10/20/2021 12:18:29 PM is of type
System.DateTime
        Appended the formatted object
        AppendLiteral called: . The time doesn't use formatting.
        Appended the literal string
Error Level. CurrentTime: 10/20/2021 12:18:29 PM. The time doesn't use
formatting.
        literal length: 65, formattedCount: 1
        AppendLiteral called: Error Level. CurrentTime:
        Appended the literal string
        AppendFormatted (IFormattable version) called: 10/20/2021 12:18:29
PM with format {t} is of type System.DateTime,
        Appended the formatted object
        AppendLiteral called: . This is an error. It will be printed.
        Appended the literal string
Error Level. CurrentTime: 12:18 PM. This is an error. It will be printed.
        literal length: 50, formattedCount: 1
        AppendLiteral called: Trace Level. CurrentTime:
Warning Level. This warning is a string, not an interpolated string
expression.
```

你可以对处理程序的构造函数进行最后一次更新，以提高效率。 处理程序可以添加一个最终的 `out bool` 参数。 将该参数设置为 `false` 指示根本不应调用处理程序来处理内插字符串表达式：

C#

```
public LogInterpolatedStringHandler(int literalLength, int formattedCount,
Logger logger, LogLevel level, out bool isEnabled)
{
    isEnabled = logger.EnabledLevel >= level;
    Console.WriteLine($"\tliteral length: {literalLength}, formattedCount:
{formattedCount}");
    builder = isEnabled ? new StringBuilder(literalLength) : default!;
}
```

此更改意味着可以删除 `enabled` 字段。 然后，可以将 `AppendLiteral` 和 `AppendFormatted` 的返回类型更改为 `void`。 现在，运行示例时，会看到以下输出：

PowerShell

```
        literal length: 60, formattedCount: 1
        AppendLiteral called: Error Level. CurrentTime:
        Appended the literal string
        AppendFormatted called: 10/20/2021 12:19:10 PM is of type
System.DateTime
        Appended the formatted object
        AppendLiteral called: . The time doesn't use formatting.
        Appended the literal string
Error Level. CurrentTime: 10/20/2021 12:19:10 PM. The time doesn't use
formatting.
        literal length: 65, formattedCount: 1
        AppendLiteral called: Error Level. CurrentTime:
        Appended the literal string
        AppendFormatted (IFormattable version) called: 10/20/2021 12:19:10
PM with format {t} is of type System.DateTime,
        Appended the formatted object
        AppendLiteral called: . This is an error. It will be printed.
        Appended the literal string
Error Level. CurrentTime: 12:19 PM. This is an error. It will be printed.
        literal length: 50, formattedCount: 1
Warning Level. This warning is a string, not an interpolated string
expression.
```

指定 `LogLevel.Trace` 时的唯一输出是构造函数的输出。 处理程序指示它未启用，因此未调用任何 `Append` 方法。

此示例演示了内插字符串处理程序的重要点，尤其是在使用日志记录库时。 占位符中可能不会出现任何副作用。 将以下代码添加到主程序，并在操作中查看此行为：

C#

```
int index = 0;
int numberOfIncrements = 0;
for (var level = LogLevel.Critical; level <= LogLevel.Trace; level++)
{
    Console.WriteLine(level);
    logger.LogMessage(level, $"{level}: Increment index a few times
{index++}, {index++}, {index++}, {index++}, {index++}");
    numberOfIncrements += 5;
}
Console.WriteLine($"Value of index {index}, value of numberOfIncrements:
{numberOfIncrements}");
```

可以看到 `index` 变量在循环的每次迭代中递增了五次。 由于占位符仅针对 `Critical`、`Error` 和 `Warning` 级别（而不是 `Information` 和 `Trace`）进行评估，因此 `index` 的最终值与预期不匹配：

PowerShell

```
Critical
Critical: Increment index a few times 0, 1, 2, 3, 4
Error
Error: Increment index a few times 5, 6, 7, 8, 9
Warning
Warning: Increment index a few times 10, 11, 12, 13, 14
Information
Trace
Value of index 15, value of numberOfIncrements: 25
```

内插字符串处理程序可以更好地控制内插字符串表达式转换为字符串的方式。 .NET 运行时团队使用此功能来提高多个方面的性能。 可以在自己的库中使用相同的功能。 若要进一步探索，请看看 System.Runtime.CompilerServices.DefaultInterpolatedStringHandler。 它提供了比您在这里构建的更完整的实现。 你将看到 `Append` 方法还可以有更多重载。

.NET Compiler Platform SDK

项目 • 2024/04/11

编译器在验证代码语法和语义时生成应用代码的详细模型。 此模型可用于根据源代码生成可执行输出。 .NET Compiler Platform SDK 提供对此模型的访问权限。 我们越来越依赖 IntelliSense、重构、智能重命名、“查找所有引用”和“转到定义”等集成开发环境 (IDE) 功能来提高工作效率。 我们依靠代码分析工具来提升代码质量，并依靠代码生成器来帮助构造应用。 随着这些工具越来越智能化，它们需要越来越多地访问仅由编译器在处理应用代码时创建的模型。 这就是 Roslyn API 的核心任务所在：打开“不透明匣”，让工具和最终用户能够共享编译器生成的大量代码相关信息。 通过 Roslyn，编译器成为平台（而不是不透明的源代码输入和目标代码输出转换器）：可用于在工具和应用程序中完成代码相关任务的 API。

.NET Compiler Platform SDK 概念

.NET Compiler Platform SDK 极大地降低了创建以代码为中心的工具和应用的门槛。 它将在元编程、代码生成和转换、C# 和 Visual Basic 语言的交互式使用，以及在域特定语言中嵌入 C# 和 Visual Basic 等方面，创造许多创新的机遇。

.NET Compiler Platform SDK 使你能够生成分析器和代码修补程序，用于发现和更正编码错误。 分析器可理解语法（代码的结构）和语义，从而检测应更正的做法。 代码修补程序建议一处或多处修复，以修复分析器或编译器诊断发现的编码错误。 通常情况下，分析器和关联的代码修补程序一起打包在一个项目中。

分析器和代码修补程序通过静态分析来理解代码。 它们既不运行代码，也未带来其他测试方面的好处。 但是，它们可以指出经常导致 bug 的做法、无法维护的代码或标准原则冲突。

除了分析器和代码修补程序外，.NET Compiler Platform SDK 还允许你生成代码重构。 它还提供了一组 API，可便于检查和理解 C# 或 Visual Basic 代码库。 由于可以使用这一基准代码，因此能够利用 .NET Compiler Platform SDK 提供的语法和语义分析 API，更轻松地编写分析器和代码修补程序。 从复制由编译器执行的分析的繁重任务中解放出来，可以将精力放在为项目或库查找常见编码错误并进行修复的更为重要的任务上。

这带来的一个小小的好处是，分析器和代码修补程序较小，在 Visual Studio 中加载时占用的内存比为了理解项目中的代码而编写自己的基准代码占用的内存少得多。 利用编译器和 Visual Studio 使用的相同类，可以创建自己的静态分析工具。 也就是说，团队可以使用分析器和代码修补程序，而不会对 IDE 的性能造成显著影响。

在下列三个主要方案中，需要编写分析器和代码修补程序：

1. *强制执行团队编码标准*
2. *提供库包方面的指导*
3. *提供常规指南*

强制执行团队编码标准

许多团队都有编码标准，通过由其他团队成员评审代码的形式强制执行。 分析器和代码修补程序可以让这个过程更加高效。 当开发人员与团队中的其他成员共享工作内容时，其他成员会执行代码评审。 在有任何注释前，开发人员一直都在完成新功能。 开发人员不符合团队做法的习惯就会得到强化，时间可能长达几周之久。

分析器在开发人员编写代码的同时运行。 这样，开发人员就可以获得即时反馈，从而立即遵循相关指南。 只要开始原型设计，开发人员就养成了编写符合标准的代码的习惯。 当功能可供人员评审时，所有标准指南就已强制执行。

团队可以生成分析器和代码修补程序，以发现违反团队编码做法的最常见做法。 这些分析器和代码修补程序可以安装到每个开发人员的计算机上，以强制执行标准。

> **提示**
>
> 在构建你自己的分析器之前，请先查看内置分析器。 有关详细信息，请参阅**代码样式规则**。

提供库包方面的指导

NuGet 上有大量适用于 .NET 开发人员的库。 一些来自 Microsoft，一些来自第三方公司，另一些来自社区成员和志愿者。 如果开发人员可以成功使用这些库后，它们的采用率和评价就会有所提升。

除了提供文档之外，还可以提供分析程序和代码修补程序，用于发现和更正库的常见错误用法。 这些即时更正有助于开发人员更快地成功使用库。

可以将分析器和代码修补程序与 NuGet 上的库一起打包。 在这种情况下，每个安装了 NuGet 包的开发人员都还将安装分析器包。 所有使用库的开发人员都会立即从团队获得指导，即有关错误和建议更正做法的即时反馈。

提供常规指南

.NET 开发者社区已发现效果理想的体验模式和最好避免使用的模式。 一些社区成员已创建分析器来强制执行这些推荐模式。 随着掌握的信息越来越多，新见解是无止境的。

这些分析器可以上传到 Visual Studio Marketplace 中，并由开发人员使用 Visual Studio 进行下载。 语言和平台的新手可以快速了解公认做法，并尽早在 .NET 之旅中高效工作。 随着使用越来越广泛，这些做法就会被社区采用。

后续步骤

.NET Compiler Platform SDK 包括用于代码生成、分析和重构的最新语言对象模型。 此部分从概念上概述了 .NET Compiler Platform SDK。 有关更多详细信息，可以参阅快速入门、示例和教程部分。

若要详细了解 .NET Compiler Platform SDK 概念，可以参阅下列五个主题：

- 使用语法可视化工具浏览代码
- 了解编译器 API 模型
- 处理语法
- 处理语义
- 处理工作区

若要开始，需要安装 .NET 编译器平台 SDK：

安装说明 - Visual Studio 安装程序

在“Visual Studio 安装程序”中查找“.NET Compiler Platform SDK”有两种不同的方法：

使用 Visual Studio 安装程序进行安装 - 工作负荷视图

Visual Studio 扩展开发工作负荷中不会自动选择 .NET Compiler Platform SDK。 必须将其作为可选组件进行选择。

1. 运行“Visual Studio 安装程序”
2. 选择“修改”
3. 检查“Visual Studio 扩展开发”工作负荷。
4. 在摘要树中打开“Visual Studio 扩展开发”节点。
5. 选中“.NET Compiler Platform SDK”框。 将在可选组件最下面找到它。

还可使“DGML 编辑器”在可视化工具中显示关系图：

1. 在摘要树中打开“单个组件”节点。
2. 选中“DGML 编辑器”框

使用 Visual Studio 安装程序进行安装 - 各组件选项卡

1. 运行“Visual Studio 安装程序”
2. 选择“修改”
3. 选择“单个组件”选项卡
4. 选中“.NET Compiler Platform SDK”框。 将在“编译器、生成工具和运行时”部分最上方找到它。

还可使“DGML 编辑器”在可视化工具中显示关系图：

1. 选中“DGML 编辑器”框。 将在“代码工具”部分下找到它。

反馈

此页面是否有帮助？ 是 否

提供产品反馈

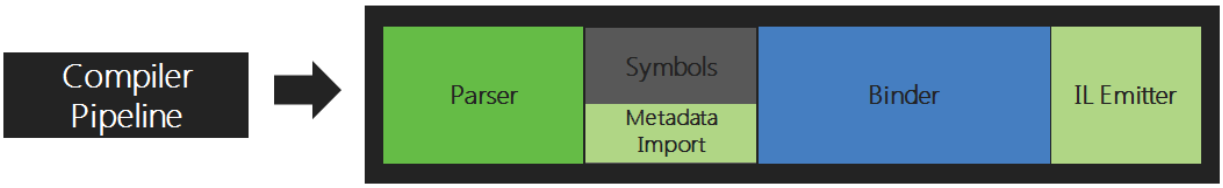
了解 .NET Compiler Platform SDK 模型

项目 • 2023/11/30

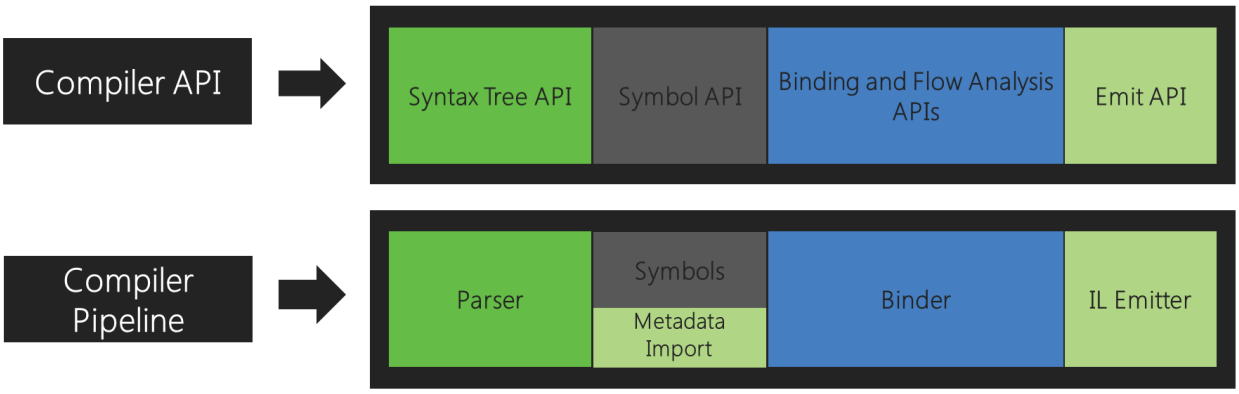
编译器按照结构化规则处理代码，这些规则通常不同于用户读取和理解代码的方式。 基本了解编译器使用的模型对于了解生成基于 Roslyn 的工具时使用的 API 至关重要。

编译器管道功能区域

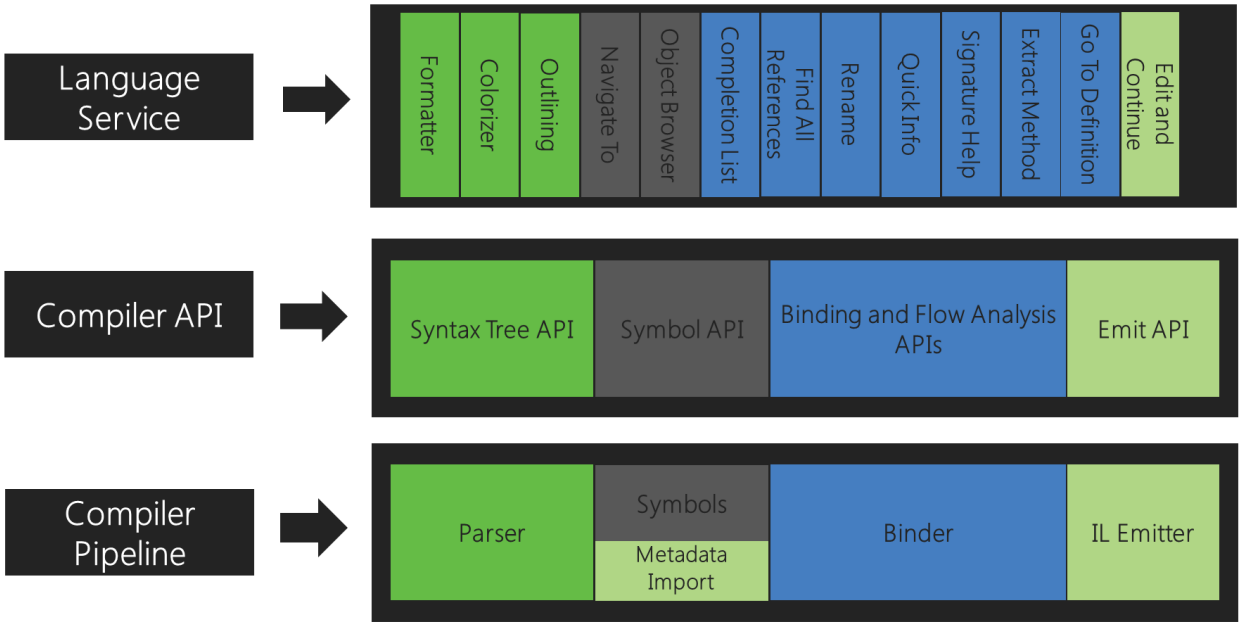
.NET Compiler Platform SDK 通过提供镜像传统编译器管道的 API 层，作为使用者向用户公开 C# 和 Visual Basic 编译器的代码分析。



此管道的每个阶段都是一个单独的组件。 首先，分析阶段标记化源文本，并将其分析为遵循语言语法的语法。 随后，声明阶段分析源和导入的元数据，形成命名符号。 然后，绑定阶段将代码中的标识符与符号匹配。 最后，发出阶段发出所有信息均由编译器生成的程序集。



.NET Compiler Platform SDK 对应于每个阶段提供一个对象模型，该模型允许访问该阶段的信息。 分析阶段公开一个语法树，声明阶段公开一个分层符号表，绑定阶段公开编译器语义分析的结果，发出阶段公开生成 IL 字节代码的 API。



每个编译器将这些组件合并在一起，组成一个端到端整体。

这些 API 与 Visual Studio 使用的 API 相同。 例如，代码大纲和格式设置功能使用语法树，对象浏览器和导航功能使用符号表，重构和转到定义使用语义模型，编辑并继续使用所有这些信息，包括发出 API 。

API 层

.NET 编译器 SDK 包含多个 API 层：编译器 API、诊断 API、脚本 API 和工作区 API。

编译器 API

编译器层包含的对象模型（语法模型和语义模型）对应于在编译器管道的每个阶段公开的信息。 编译器层还包含编译器的单次调用的不可变快照，包括程序集引用、编译器选项和源代码文件。 C# 语言和 Visual Basic 语言由两个不同的 API 表示。 这两个 API 的形状类似，但针对每种语言的高保真进行了定制。 此层不包含 Visual Studio 组件的依赖项。

诊断 API

在编译器分析过程中，编译器会生成一组诊断，包括语法、语义、明确赋值以及各种警告和信息性诊断的所有内容。 编译器 API 层通过一个可扩展 API 公开诊断，该可扩展 API 允许将用户定义的分析器插入编译过程。 它支持随编译器定义的诊断一起生成用户定义的诊断，例如由 StyleCop 等工具生成的诊断。 以这种方式生成诊断有以下优势：与 MSBuild 和 Visual Studio 等工具（具体取决于对根据策略停止生成等体验进行的诊断）自然地集成、在编辑器中显示实时波形曲线，以及建议代码修复。

脚本 API

托管 API 和脚本 API 构建在编译器层之上。 可以使用脚本 API 运行代码片段并累积运行时执行上下文。 C# 交互式 REPL （读取–求值–打印循环）可使用这些 API。 借助 REPL，可将 C# 用作脚本语言，在编写代码的同时，以交互方式运行代码。

工作区 API

工作区层包含工作区 API，是对整个解决方案执行代码分析和重构的起点。 它协助你将解决方案中项目的全部相关信息整理到单个对象模型中，可便于你直接访问编译器层对象模型，而无需分析文件、配置选项或管理项目到项目依赖关系。

此外，工作区层还包含一组 API，用于实现在 Visual Studio IDE 等主机环境中使用的代码分析和重构工具。 相关示例包括查找所有引用、格式设置和代码生成 API。

此层不包含 Visual Studio 组件的依赖项。

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 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

使用语法

项目 • 2023/05/10

“语法树”是一种由编译器 API 公开的基础的不可变数据结构。 这些树表示源代码的词法和语法结构。 它们有两个重要用途：

- 支持使用工具（如 IDE、加载项、代码分析工具和重构）查看和处理用户项目中源代码的语法结构。
- 支持使用工具（如重构和 IDE）以自然的方式创建、修改和重新排列源代码，而无需直接编辑文本。 通过创建和操作语法树，可轻松使用工具创建和重新排列源代码。

语法树

语法树是用于编译、代码分析、绑定、重构、IDE 功能和代码生成的主要结构。 要理解任意部分的源代码，都必须先加以识别，然后将其分类为众多已知结构化语言元素之一。

语法树具有三个关键特性：

- 它们完全保真地保存所有源信息。 完全保真意味着语法树包含可在源文本中找到的每份信息、每个语法结构、每个词法标记，以及它们之间的所有其他内容，包括空格、注释和预处理器指令。 例如，源中提及的所有文本都完全按照键入的形式表示。 语法树还可在程序不完整或格式错误时，通过表示出已跳过或缺少的标记，来捕获源代码中的错误。
- 它们可以生成其分析源的确切文本。 可从任何语法节点获取以该节点为根的子树的文本表示形式。 此功能意味着语法树可以用作一种构造和编辑源文本的方法。 创建树即会隐式创建等效文本，对现有树进行更改会创建新树，可高效地编辑文本。
- 它们是不可变的，并且是线程安全的。 获取的树是代码当前状态的快照，不会更改。 这可让多个用户同时在不同线程中与同一语法树进行交互，而不会锁定或重复。 由于语法树恒定不变，并且不可直接对其进行修改，因此工厂方法可通过创建树的另一个快照来帮助创建和修改语法树。 语法树可高效重用基础节点，因此几乎无需使用额外的内存便可快速重新生成新版本。

语法树实际上是一个树形数据结构，其中非终端结构化元素是其他元素的父元素。 每个语法树都由节点、标记和琐碎内容组成。

语法节点

语法节点是语法树的一个主要元素。 这些节点表示声明、语句、子句和表达式等语法构造。 语法节点的每个类别都由派生自 Microsoft.CodeAnalysis.SyntaxNode 的单独类表

示。 节点类集是不可扩展的。

所有语法节点都是语法树中的非终端节点，这意味着这些节点始终有其他节点和标记作为子元素。 作为另一个节点的子级，每个节点都具有可通过 SyntaxNode.Parent 属性访问的父节点。 由于节点和树恒定不变，因此节点的父节点永远不会更改。 树的根以 null 为父级。

每个节点都包含 SyntaxNode.ChildNodes() 方法，可根据子节点在源文本中的位置按顺序返回子节点列表。 此列表中不包含标记。 每个节点还包含用于检查子代的方法（例如，DescendantNodes、DescendantTokens 或 DescendantTrivia），这些子代表示以该节点为根的子树中存在的所有节点、标记或琐碎内容的列表。

此外，每个语法节点子类通过强类型属性公开所有相同的子级。 例如，BinaryExpressionSyntax 节点类具有三个特定于二元运算符的其他属性：Left、OperatorToken 和 Right。 Right 和 ExpressionSyntax 的类型为 Left，OperatorToken 的类型为 SyntaxToken。

某些语法节点具有可选子级。 例如，IfStatementSyntax 具有可选的 ElseClauseSyntax。 如果没有子级，则该属性返回 null。

语法标记

语法标记是语言语法的终端，表示代码的最小语法片段。 它们从不作为其他节点或标记的父级。 语法标记包含关键字、标识符、文本和标点。

为了提高效率，SyntaxToken 类型为 CLR 值类型。 因此，与语法节点不同，所有类型的标记都采用同一结构，但包含各种属性，这些属性的意义取决于表示的标记类型。

例如，整数文本标记表示数字值。 除了原始源文本和标记范围外，文本标记还包含 Value 属性，用于告知精确解码的整数值。 此属性类型化为 Object，因为它可能属于多个基元类型之一。

ValueText 属性与 Value 属性告知相同的信息；但前者始终类型化为 String。 C# 源文本中的标识符可能包括 Unicode 转义字符，但转义序列本身的语法不被视为标识符名称的一部分。 因此，虽然标记跨越的原始文本包含转义序列，但 ValueText 属性却不包含转义序列。 而是包括转义识别的 Unicode 字符。 例如，如果源文本包含写为 `\u03C0` 的标识符，则此标记的 ValueText 属性返回 `π`。

语法琐碎内容

语法琐碎内容表示对正常理解代码基本上没有意义的源文本部分，例如空格、注释和预处理器指令。 与语法标记类似，琐碎内容为值类型。 单个

Microsoft.CodeAnalysis.SyntaxTrivia 类型用于描述各种类型的琐碎内容。

由于琐碎内容不是正常语言语法的一部分，并且可以出现在任意两个标记之间的任意位置，因此它们不会作为节点的子级包含在语法树中。 但由于它们对于实现重构等功能和完全保真地保留源文本非常重要，因此还是包含在语法树内。

可通过检查标记的 SyntaxToken.LeadingTrivia 或 SyntaxToken.TrailingTrivia 集合来访问琐碎内容。 分析源文本时，琐碎内容序列与标记关联。 通常情况下，一个标记拥有其后位于同一行中下一个标记之前的任意琐碎内容。 该行之后的任意琐碎内容与下一个标记关联。 源文件中的第一个标记可获取所有初始琐碎内容，而最后一个琐碎内容序列附加到文件尾标记，否则其宽度为零。

与语法节点和语法标记不同，语法琐碎内容没有父级。 但由于它们是树的一部分，且每个琐碎内容都与一个标记关联，因此可使用 SyntaxTrivia.Token 属性访问关联的标记。

范围

每个节点、标记或琐碎内容都知道其在源文本内的位置和包含的字符数。 文本位置表示为一个 32 位整数，是一个从零开始的 `char` 索引。 TextSpan 对象表示开始位置和字符计数，都表示为整数。 如果 TextSpan 的长度为零，则其表示两个字符之间的位置。

每个节点具有两个 TextSpan 属性：Span 和 FullSpan。

Span 属性表示从节点子树中第一个标记的开头到最后一个标记末尾的文本范围。 此范围不包括任何前导或尾随琐碎内容。

FullSpan 属性表示的文本范围包括节点的正常范围，加上任何前导或尾随琐碎内容的范围。

例如：

```
C#

      if (x > 3)
      {
||        // this is bad
          |throw new Exception("Not right.");|  // better exception?||
      }
```

块内语句节点的范围由单个竖线 (|) 指示。 它包含字符 `throw new Exception("Not right.");`。 完整的范围由双竖线 (||) 指示。 它包含的字符与前导和尾随琐碎内容的相关范围和字符相同。

种类

每个节点、标记或琐碎内容都具有 System.Int32 类型的 SyntaxNode.RawKind 属性，标识所表示的确切语法元素。 此值可强制转换为特定语言的枚举。 每种语言（C# 或 Visual Basic）都具有单个 `SyntaxKind` 枚举（分别为 Microsoft.CodeAnalysis.CSharp.SyntaxKind 和 Microsoft.CodeAnalysis.VisualBasic.SyntaxKind），其中列出了语法中所有可能的节点、标记和琐碎内容。 可通过访问 CSharpExtensions.Kind 或 VisualBasicExtensions.Kind 扩展方法自动完成此转换。

RawKind 属性可轻松消除共享同一节点类的语法节点类型的歧义。 对于标记和琐碎内容，此属性是区分不同元素类型的唯一方法。

例如，一个 BinaryExpressionSyntax 类具有 Left、OperatorToken 和 Right 作为子级。 Kind 属性可辨别它是 AddExpression、SubtractExpression 还是 MultiplyExpression 类型的语法节点。

> **提示**
>
> 建议使用 **IsKind**（对于 C#）或 **IsKind**（对于 VB）扩展方法来检查类别。

错误

即使源文本中包含语法错误，也会公开可与源双向转换的完整语法树。 当分析程序遇到不符合语言定义语法的代码时，使用两种方法之一来创建语法树：

- 如果分析程序需要特定种类的标记，但未找到该标记，则其可在语法树中所需标记位置插入缺失的标记。 缺失的标记表示本应存在，但只有空范围的实际标记，并且其 SyntaxNode.IsMissing 属性返回 `true`。
- 分析程序可能会跳过标记，直至发现可继续分析的标记。 在这种情况下，跳过的令牌附加为 SkippedTokensTrivia 类型的琐碎内容节点。

使用语义

项目 • 2023/05/10

语法树表示源代码的词法和语法结构。 尽管此信息已足以描述源中的所有声明和逻辑，但不足以识别正在引用的内容。 一个名称可能表示：

- 一种类型
- 一个字段
- 一种方法
- 一个局部变量

尽管以上每一项都各不相同，但要确定标识符实际上引用哪一项，通常需要深入了解语言规则。

源代码中还表示了程序元素，并且程序也可引用以前编译的库，这些库打包在程序集文件中。 尽管没有任何可用于程序集的源代码（因此没有任何可用于程序集的语法节点或语法树），程序仍可引用其中的元素。

对于这些任务，需要语义模型 。

除了源代码的语法模型外，语义模型还封装了语言规则，提供了一种将标识符与正在引用的正确程序元素正确匹配的简单办法。

编译

编译是编译 C# 或 Visual Basic 程序所需的所有内容的表示形式，其中包括所有程序集引用、编译器选项和源文件。

由于所有这些信息都存储在一个位置，因此可以更详细地描述源代码中包含的元素。 编译以符号的形式表示每个声明的类型、成员或变量。 编译包含多种方法，可帮助查找和关联在源代码中声明的符号或作为元数据从程序集中导出的符号。

与语法树类似，编译是不可变的。 创建编译后，创建者或可能与之共享该编译的任何其他用户均不可进行更改。 但是，可以从现有编译创建新编译，指定进行的更改。 例如，可以创建与现有编译各方面均相同、但可能包含其他源文件或程序集引用的编译。

符号

符号表示源代码声明的不同元素，或作为元数据从程序集中导出。 每个命名空间、类型、方法、属性、字段、事件、参数或局部变量都由符号表示。

Compilation 类型的各种方法和属性可帮助查找符号。 例如，可以通过声明的类型的通用元数据名称来查找该类型的符号。 还可以访问整个符号表，该表是以全局命名空间为根的符号树。

符号还包含编译器根据源或元数据（如其他引用的符号）确定的其他信息。 每种类型的符号都由派生自 ISymbol 的单独接口表示，每种符号都有其自己的方法和属性，详细说明了编译器收集的信息。 其中许多属性直接引用了其他符号。 例如，IMethodSymbol.ReturnType 属性指示该方法返回的实际类型符号。

符号提供了源代码与元数据之间命名空间、类型和成员的共同表示形式。 例如，在源代码中声明的方法以及从元数据导入的方法均由具有相同属性的 IMethodSymbol 表示。

符号在概念上类似于 System.Reflection API 表示的 CLR 类型系统，但前者更丰富，因为它们不仅仅可以对类型建模。 命名空间、局部变量和标签都是符号。 此外，符号是语言概念，而不是 CLR 概念的表示形式。 存在很多重叠，但也有许多有意义的区别。 例如，在 C# 或 Visual Basic 中，迭代器方法是单个符号。 但是，当迭代器方法转换为 CLR 元数据时，它是一种类型和多个方法。

语义模型

语义模型表示单个源文件的所有语义信息。 可使用语义模型发现以下内容：

- 在源中特定位置引用的符号。
- 任何表达式的结果类型。
- 所有诊断（错误和警告）。
- 变量流入和流出源区域的方式。
- 更多推理问题的答案。

使用工作区

项目 • 2023/05/10

工作区 层是对整个解决方案执行代码分析和重构的起点。 此层内的工作区 API 有助于将解决方案中项目的全部相关信息组织为单个对象模型，可让用户直接访问编译器层对象模型（如源文本、语法树、语义模型和编译），而无需分析文件、配置选项，或管理项目内依赖项。

托管环境（如 IDE）提供对应于打开的解决方案的工作区。 此外，只需加载解决方案文件，即可在 IDE 外部使用此模型。

工作区

工作区是作为项目集合的解决方案的活动表示形式，每个工作区都包含一个文档集合。 工作区通常与随用户键入或操作属性而不断更改的主机环境关联。

Workspace 提供对当前解决方案模型的访问权限。 主机环境中发生更改时，工作区会引发相应事件，并更新 Workspace.CurrentSolution 属性。 例如，用户在一个源文档对应的文本编辑器中键入时，工作区使用一个事件来指示解决方案的整体模型已更改以及修改了哪个文档。 然后，可通过分析新模型的正确性、突出显示重要区域，或针对代码更改提供建议来响应这些更改。

此外，还可创建与主机环境断开连接或用于没有主机环境的应用程序的独立工作区。

解决方案、项目和文档

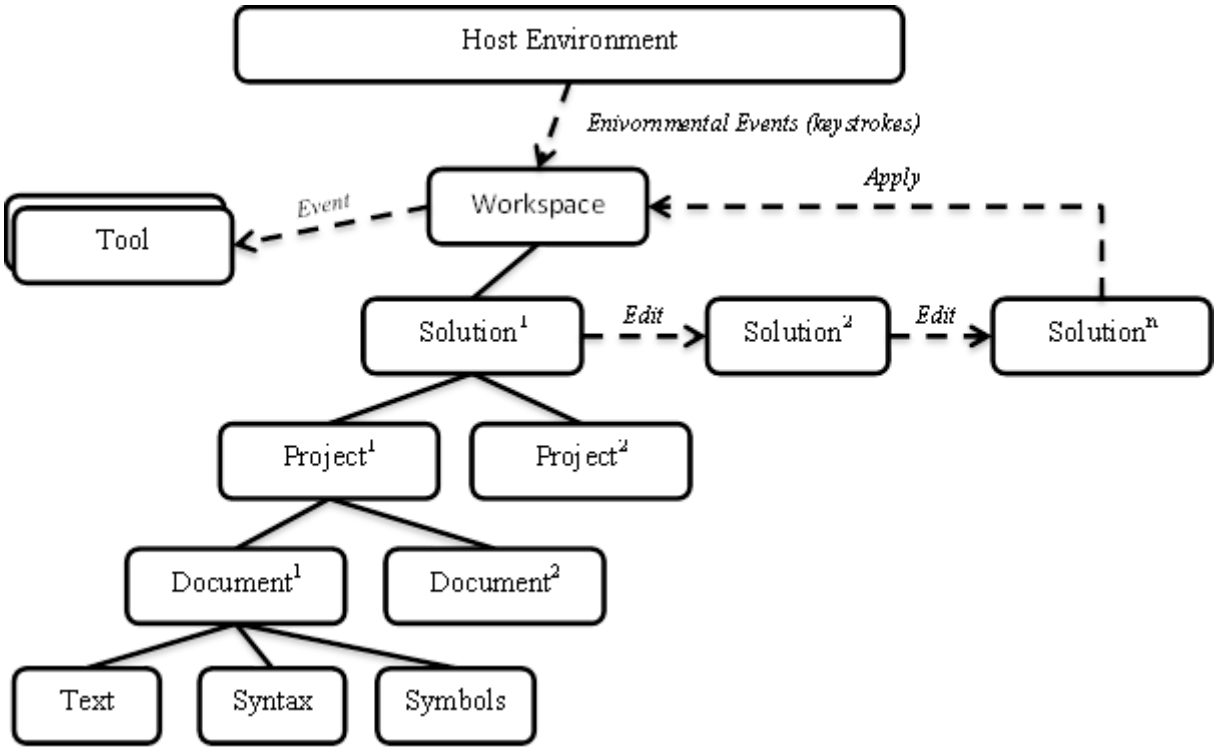
尽管每次按下一个键时工作区都可能会更改，仍可以隔离方式使用解决方案模型。

一个解决方案即是项目和文档的一个不可变模型。 这意味着可共享模型，而不会锁定或重复。 从 Workspace.CurrentSolution 属性获取解决方案实例后，该实例永远不会更改。 但是，与语法树和编译类似，可通过基于现有解决方案和特定更改构造新实例来修改解决方案。 要获取工作区以反映所做的更改，必须将更改后的解决方案显式应用回工作区。

项目是整体不可变解决方案模型的一部分。 它表示所有源代码文档、分析和编译选项，以及程序集和项目到项目引用。 可从项目中访问相应的编译，而无需确定项目依赖项或分析任何源文件。

文档也是整体不可变解决方案模型的一部分。 文档表示单个源文件，可从其中访问文件、语法树和语义模型的文本。

下图表示了如何将工作区关联到主机环境和工具，以及如何进行编辑。



总结

Roslyn 公开了一组编译器 API 和工作区 API，提供了有关源代码的丰富信息，并完全保真地记录了 C# 和 Visual Basic 语言。 .NET Compiler Platform SDK 极大地降低了创建以代码为中心的工具和应用程序的门槛。 它将在元编程、代码生成和转换、C# 和 Visual Basic 语言的交互式使用，以及在域特定语言中嵌入 C# 和 Visual Basic 等方面，创造许多创新的机遇。

使用 Visual Studio 中的 Roslyn 语法可视化工具浏览代码

项目 • 2023/05/10

本文概述了 .NET Compiler Platform（“Roslyn”）SDK 附带的语法可视化工具。 语法可视化工具是一种帮助用户检查和浏览语法树的工具窗口。 如果要理解想要分析的代码模型，该工具必不可少。 在使用 .NET Compiler Platform（“Roslyn”）SDK 开发应用程序时，它还可以提供调试帮助。 在首次创建分析器时打开该工具。 该可视化工具可帮助你理解 API 所使用的模型。 你也可以使用类似 SharpLab 或 LINQPad 这样的工具来检查代码和理解语法树。

安装说明 - Visual Studio 安装程序

在“Visual Studio 安装程序”中查找“.NET Compiler Platform SDK”有两种不同的方法：

使用 Visual Studio 安装程序进行安装 - 工作负荷视图

Visual Studio 扩展开发工作负荷中不会自动选择 .NET Compiler Platform SDK。 必须将其作为可选组件进行选择。

1. 运行“Visual Studio 安装程序”
2. 选择“修改”
3. 检查“Visual Studio 扩展开发”工作负荷。
4. 在摘要树中打开“Visual Studio 扩展开发”节点。
5. 选中“.NET Compiler Platform SDK”框。 将在可选组件最下面找到它。

还可使“DGML 编辑器”在可视化工具中显示关系图：

1. 在摘要树中打开“单个组件”节点。
2. 选中“DGML 编辑器”框

使用 Visual Studio 安装程序进行安装 - 各组件选项卡

1. 运行“Visual Studio 安装程序”
2. 选择“修改”
3. 选择“单个组件”选项卡
4. 选中“.NET Compiler Platform SDK”框。 将在“编译器、生成工具和运行时”部分最上方找到它。

还可使“DGML 编辑器”在可视化工具中显示关系图：

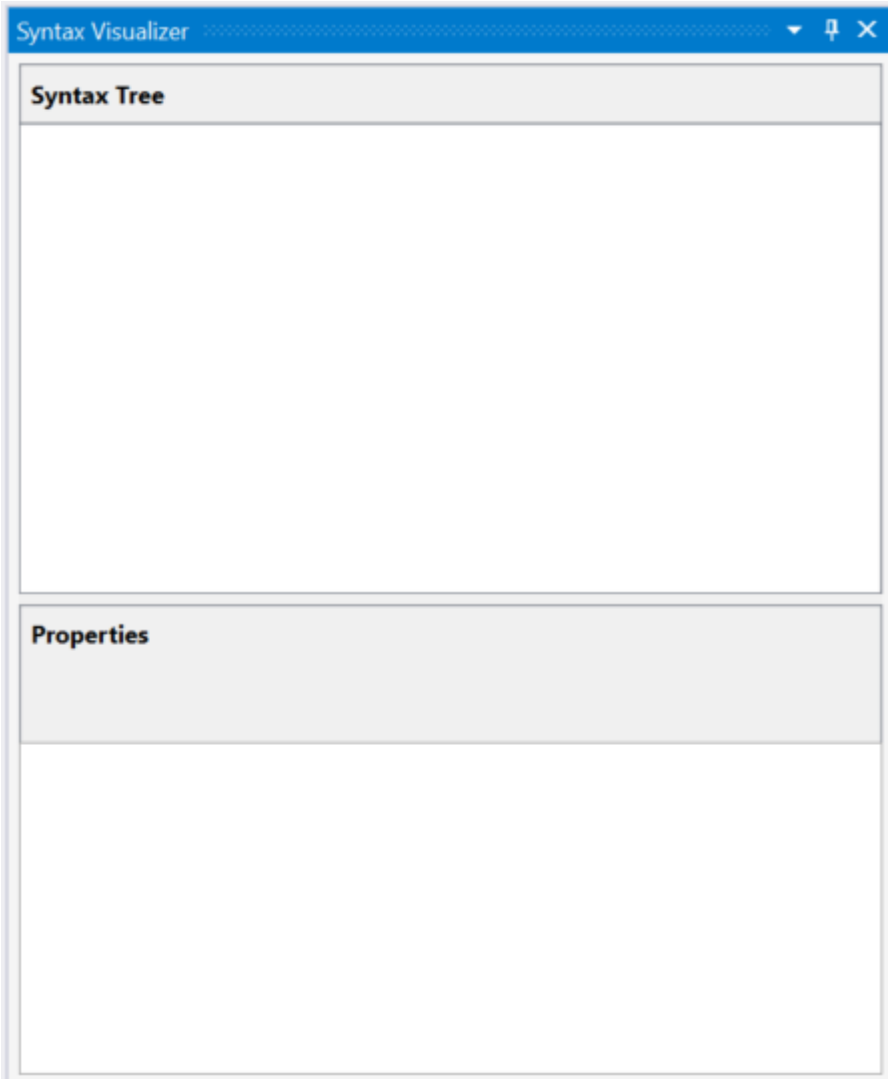
1. 选中“DGML 编辑器”框。 将在“代码工具”部分下找到它。

通过阅读本概述文章，熟悉 .NET Compiler Platform SDK 中用到的概念。 其中介绍了语法树、节点、标记和琐事。

语法可视化工具

使用“语法可视化工具”可以检查 Visual Studio IDE 当前活动的编辑器窗口中的 C# 或 Visual Basic 代码文件的语法树 。 通过单击“视图”>“其他窗口”>“语法可视化工具”，可以启动可视化工具 。 还可以使用右上角的“快速启动”工具栏 。 键入“语法”，然后应该会显示用于开启语法可视化工具的命令 。

此命令会以浮动工具窗口的形式打开语法可视化工具。 如果没有打开代码编辑器窗口，则显示为空白，如下图所示。

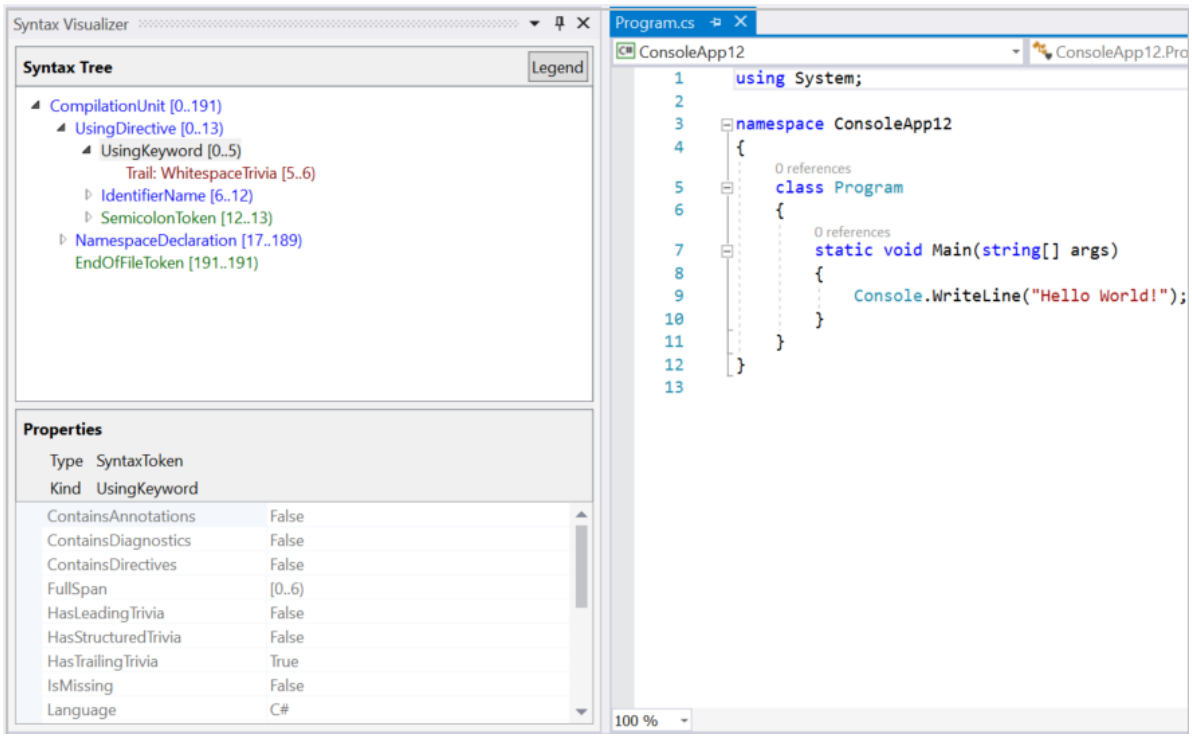


将此工具窗口停靠在 Visual Studio 中方便操作的位置，例如左侧。 可视化工具显示关于当前代码文件的信息。

使用 File>New Project 命令新建项目 。 可以创建 Visual Basic 项目或 C# 项目。 当 Visual Studio 打开此项目的主代码文件时，可视化工具会显示它的语法树。 可以打开此

Visual Studio 实例中的任何现有 C#/Visual Basic 文件，可视化工具会显示该文件的语法树。 如果在 Visual Studio 中打开了多个代码文件，可视化工具会显示当前活动的代码文件（键盘焦点所在的代码文件）的语法树。

C# Visual Basic



如前面的图像所示，可视化工具窗口在顶部显示语法树，在底部显示属性网格。 属性网格显示当前在树中选中的项的属性，包括项的 .NET 类型和种类（SyntaxKind） 。

语法树包含三种类型的项：节点、标记和琐事 。 可以在使用语法一文中阅读更多关于这些类型的内容。 每个类型的项都会用不同的颜色表示。 单击“图例”按钮，概览所使用的颜色。

树中的每个项还会显示各自的范围 。 范围就是文本文件中的节点的索引（起始位置和结束位置） 。 在前面的 C# 示例中，所选择的“UsingKeyword [0..5)”标记的范围宽度为 5 个字符，即 [0..5) 。 “[..)”表示起始索引是范围的一部分，而结束索引并不包含在内。

在树中进行导航有两种方式：

- 展开或单击树中的项。 可视化工具自动选择与代码编辑器中的项的范围对应的文本。
- 单击或选择代码编辑器中的文本。 在前面的 Visual Basic 示例中，如果在代码编辑器中选择了包含“Module Module1”的那一行，则可视化工具会在树中自动导航至对应的 ModuleStatement 节点。

可视化工具会突出显示树中的项，该项的范围与编辑器中所选择的文本的范围最匹配。

可视化工具会刷新树，以匹配活动代码文件中的修改。 将调用添加至 `Main()`.中的 `Console.WriteLine()`。 键入内容时，可视化工具会刷新树。

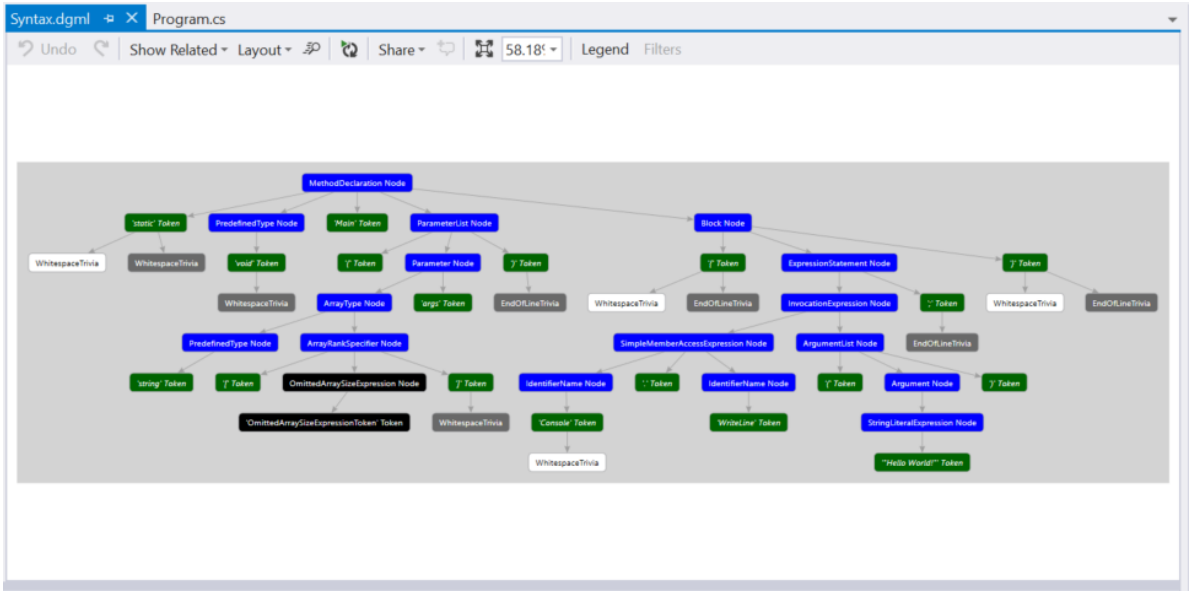
请在键入 `Console.` 后暂停键入。 树中有一些粉色的项。 这说明在所键入的代码中存在错误（通常也成为“诊断”）。 这些错误会附加到语法树的节点、标记和琐碎内容中。 可视化工具会显示哪些项存在错误，并用粉色突出显示其背景。 将鼠标悬停在该项上可以查看任何粉色项的错误。 可视化工具只显示语法错误（这些错误与键入代码的语法相关）；不会显示任何语义错误。

语法关系图

右键单击树中的任何项，然后单击“查看定向语法关系图”。

C# Visual Basic

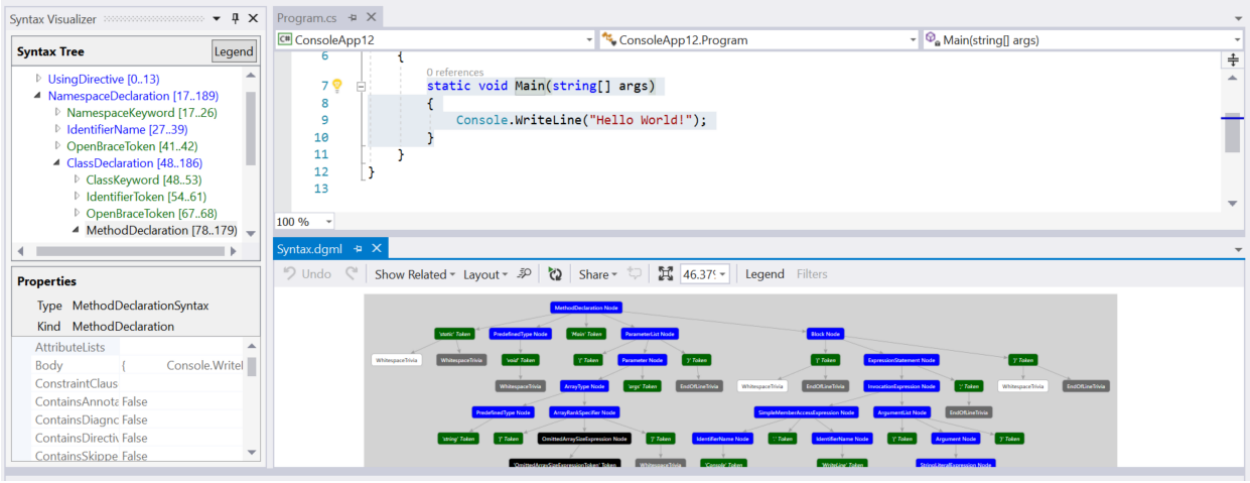
可视化工具会以图解形式显示以所选项为根的关系子树。 针对 C# 示例中对应于 `Main()` 方法的 MethodDeclaration 节点，尝试以下步骤。 可视化工具显示如下所示的语法关系图：



语法关系图查看器中有一个选项，可用于显示其着色方案的图例。 还可以将鼠标悬停在语法关系图中的每个项上，以查看该项对应的属性。

可以重复查看树中不同项的语法关系图，这些关系图会始终显示在 Visual Studio 的同一窗口中。 可以将此窗口停靠在 Visual Studio 中方便操作的位置，这样在查看新的语法关系图时就不需要在选项卡之间切换了。 通常放在底部（代码编辑器窗口的下面）会比较方便。

下面是可视化工具窗口以及语法关系图窗口所采用的停靠布局：

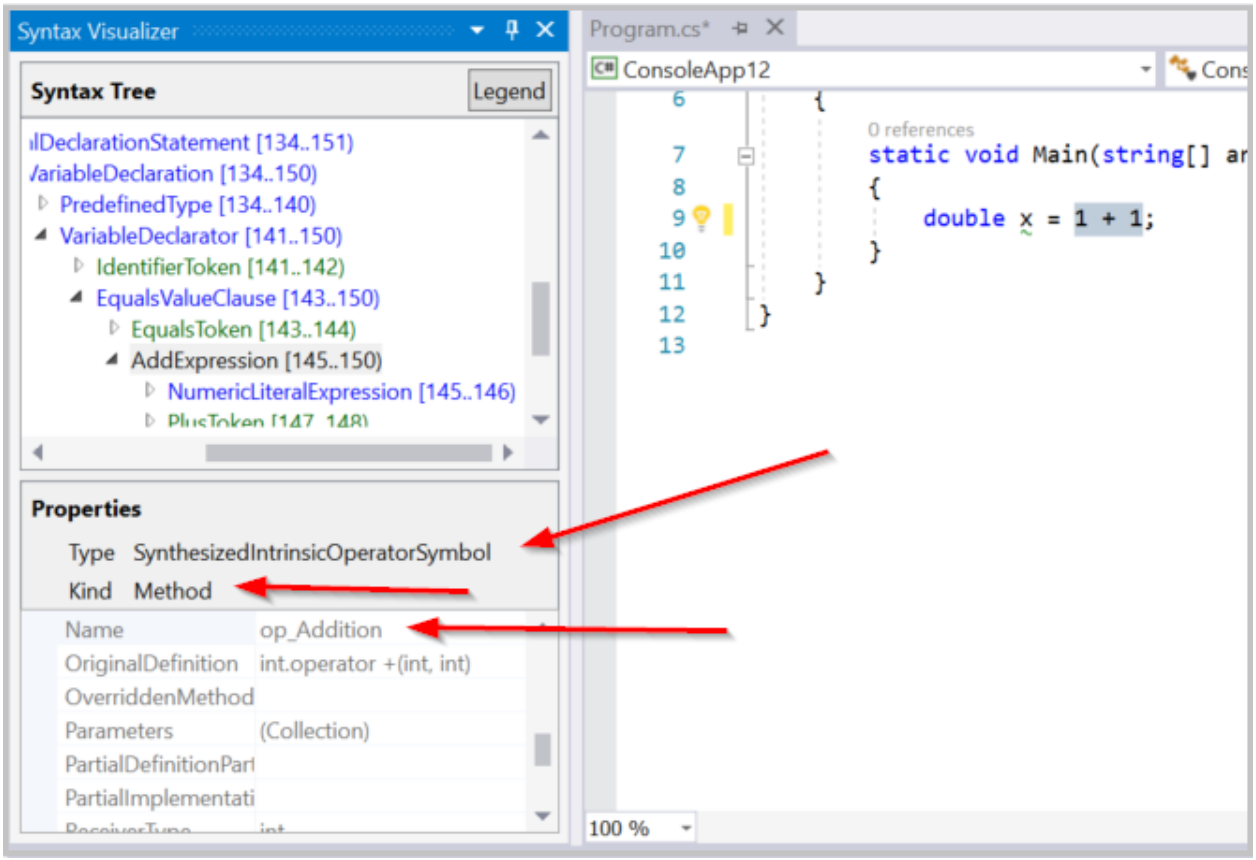


另一种选择在双监视器配置中，将语法关系图窗口放在第二个监视器上。

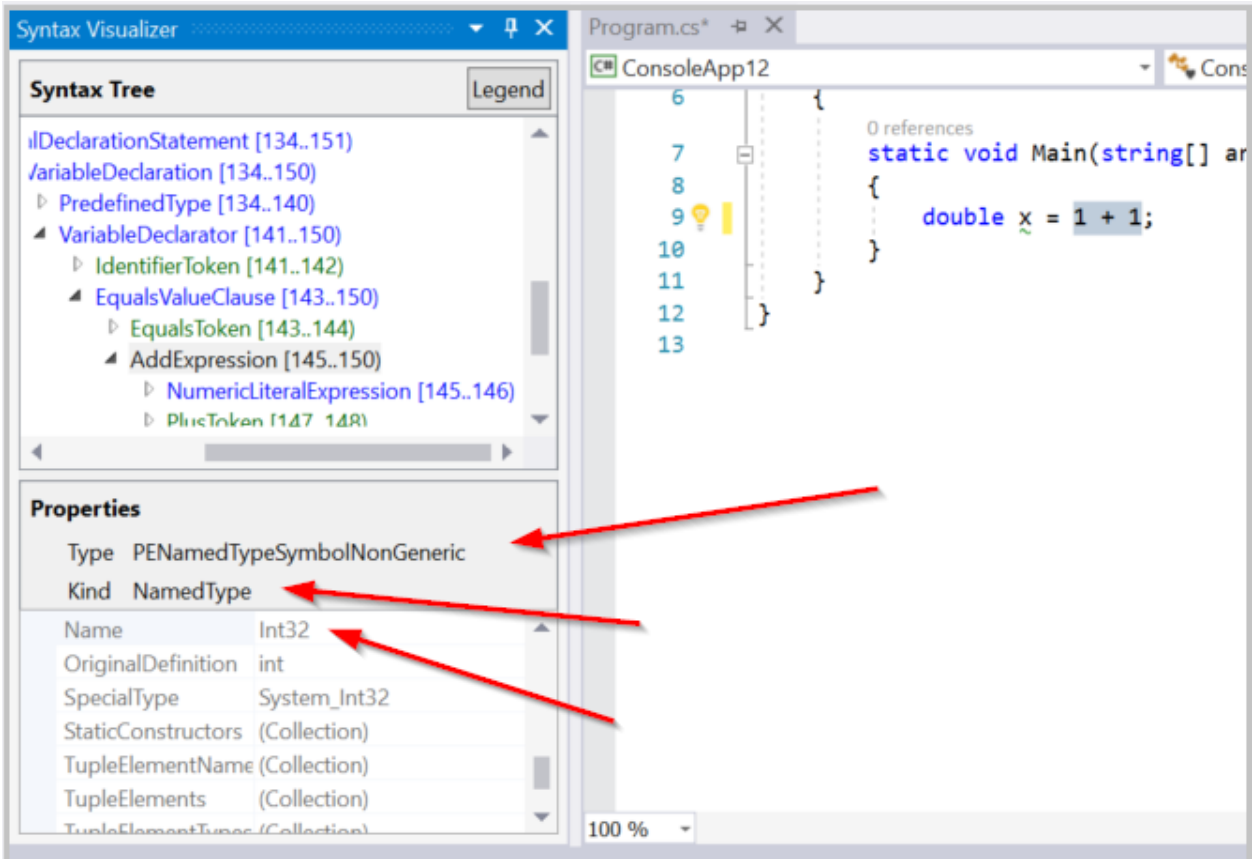
检查语义

语法可视化工具可以对符号和语义信息进行基本检查。 在 C# 示例中的 Main() 内键入 `double x = 1 + 1;`。 然后在代码编辑器窗口中选择表达式 `1 + 1`。 可视化工具突出显示了 AddExpression 节点 。 右键单击 AddExpression，然后单击“查看符号(如果有)” 。 请注意，大部分菜单项都带有“如果有”这个限定条件。 语法可视化工具检查节点的属性，包括不是所有节点都有的属性。

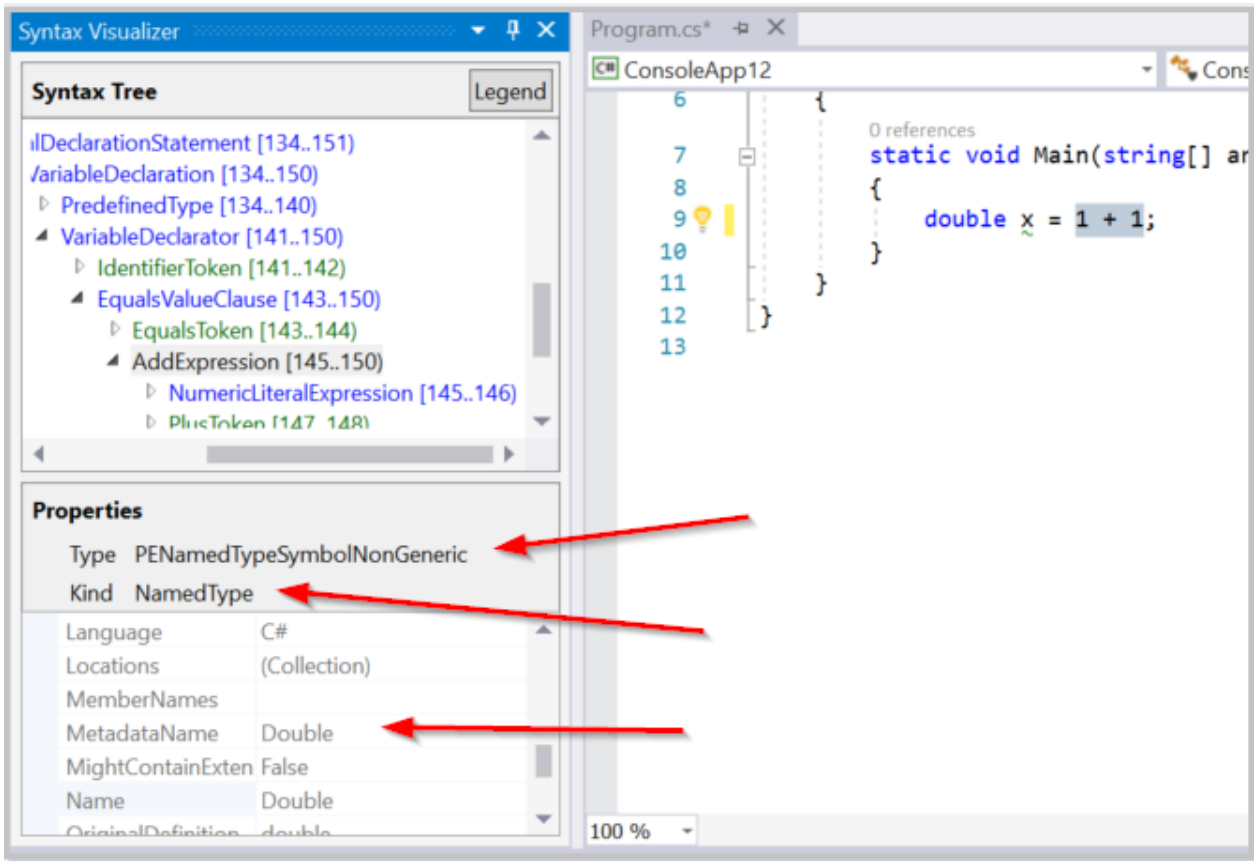
可视化工具中的属性网格更新如下图所示：该表达式的符号是 SynthesizedIntrinsicOperatorSymbol ，其中种类 = 方法 。



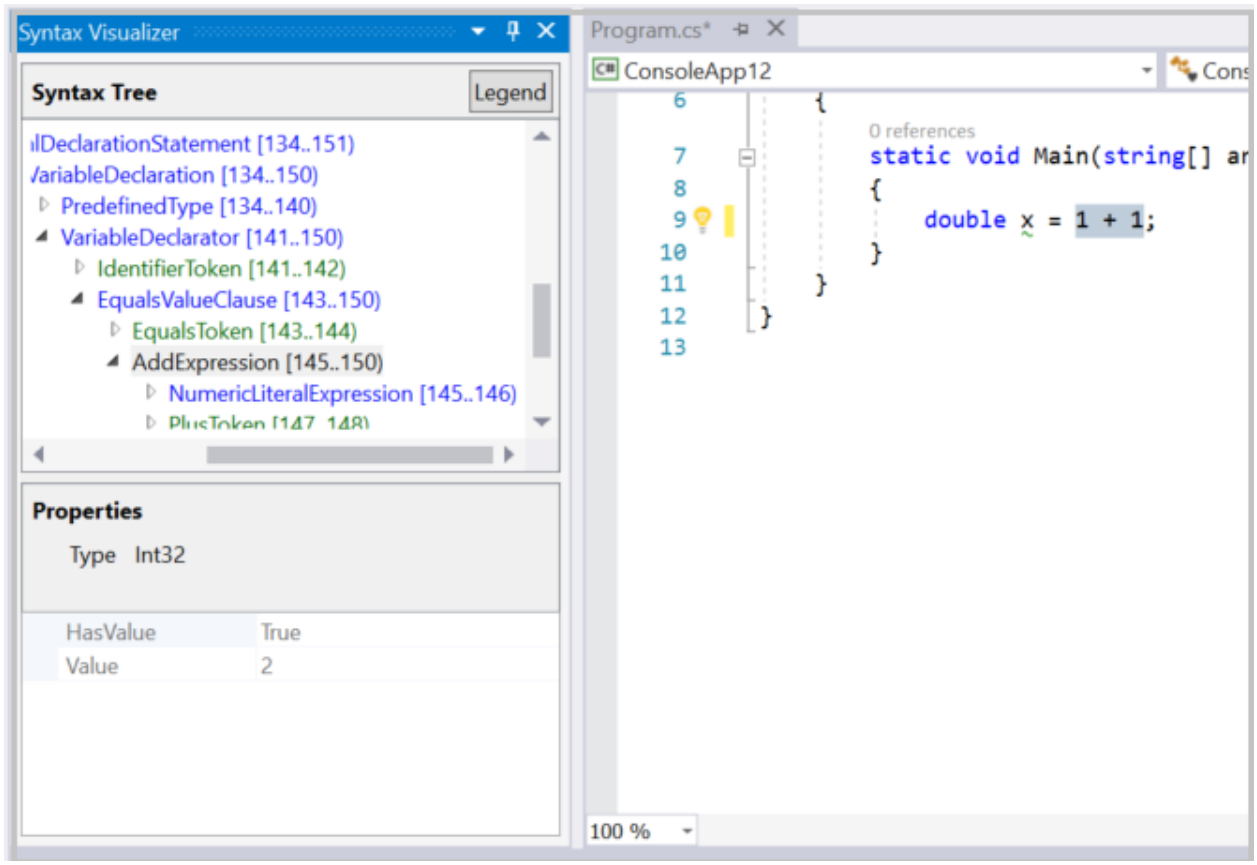
针对同一个 AddExpression 节点，请尝试“查看 TypeSymbol (如果有)” 。 如下图所示，可视化工具中的属性网格已更新，指示所选表达式的类型为 `Int32`。



针对同一个 AddExpression 节点，请尝试“查看转换后的 TypeSymbol (如果有)” 。 属性网格的更新内容指示虽然表达式的类型为 `Int32`，但转换后的表达式类型为 `Double`，如下图所示。 此节点包含转换后的类型符号信息，因为 `Int32` 表达式所在的上下文要求必须转换为 `Double` 型。 此转换满足了为赋值运算符左侧的变量 `x` 指定的类型为 `Double` 型的要求。



最后，针对同一 AddExpression 节点，尝试“查看常数值(如果有)” 。 属性网格显示该表达式的值是一个值为 `2` 的编译时常数。



在 Visual Basic 中也可以重复上述示例。 在 Visual Basic 文件中键入 `Dim x As Double = 1 + 1`。 在代码编辑器窗口中选择表达式 `1 + 1`。 可视化工具突出显示了对应的

AddExpression 节点 。 对此 AddExpression 节点重复前面所述的步骤，应该能看到相同的结果 。

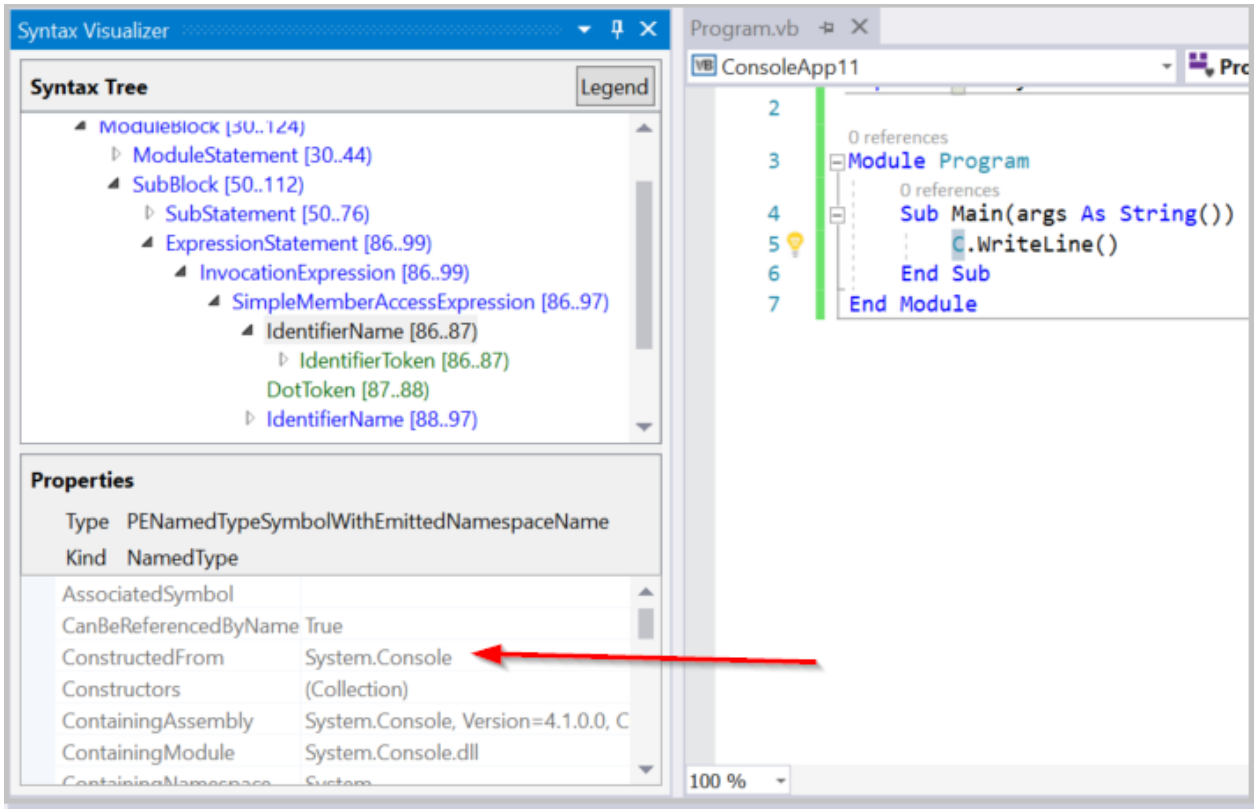
检查 Visual Basic 中的更多代码。 使用以下代码更新主 Visual Basic 文件：

VB

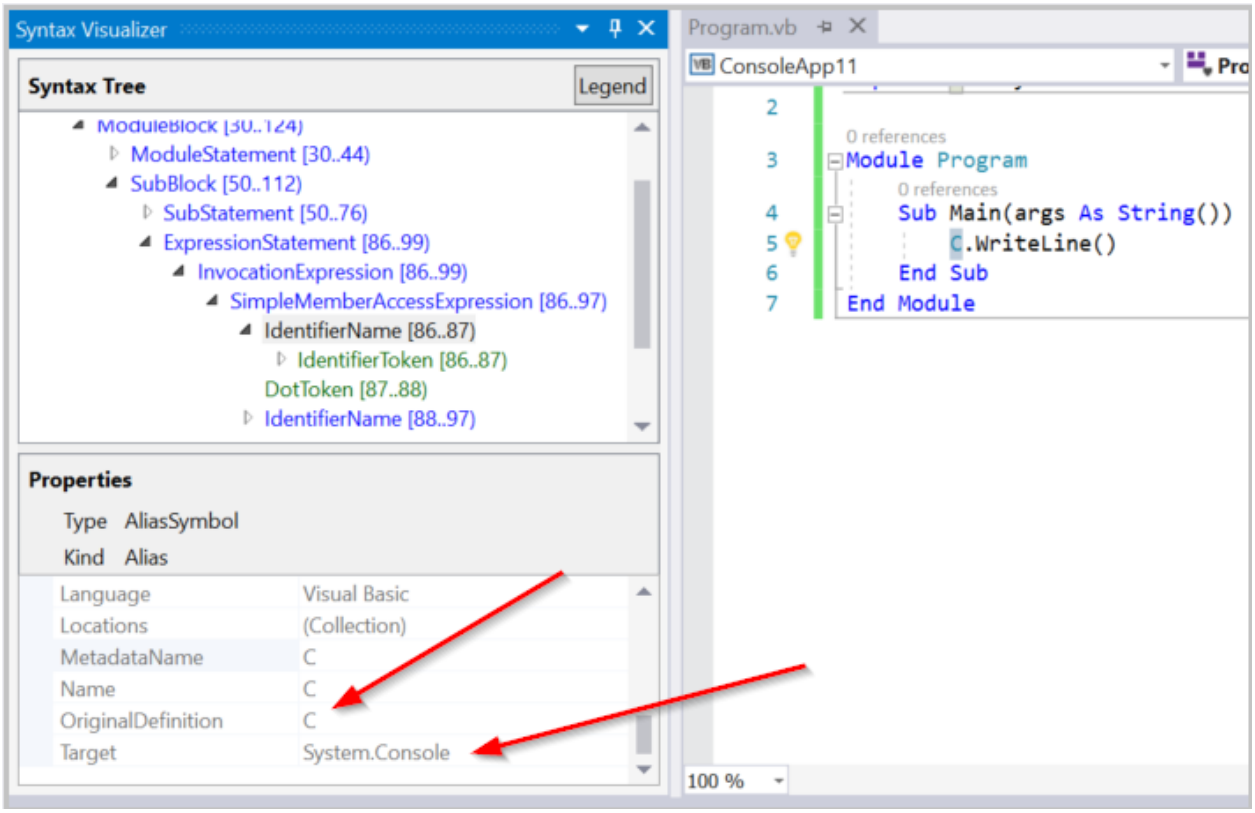
```
Imports C = System.Console

Module Program
    Sub Main(args As String())
        C.WriteLine()
    End Sub
End Module
```

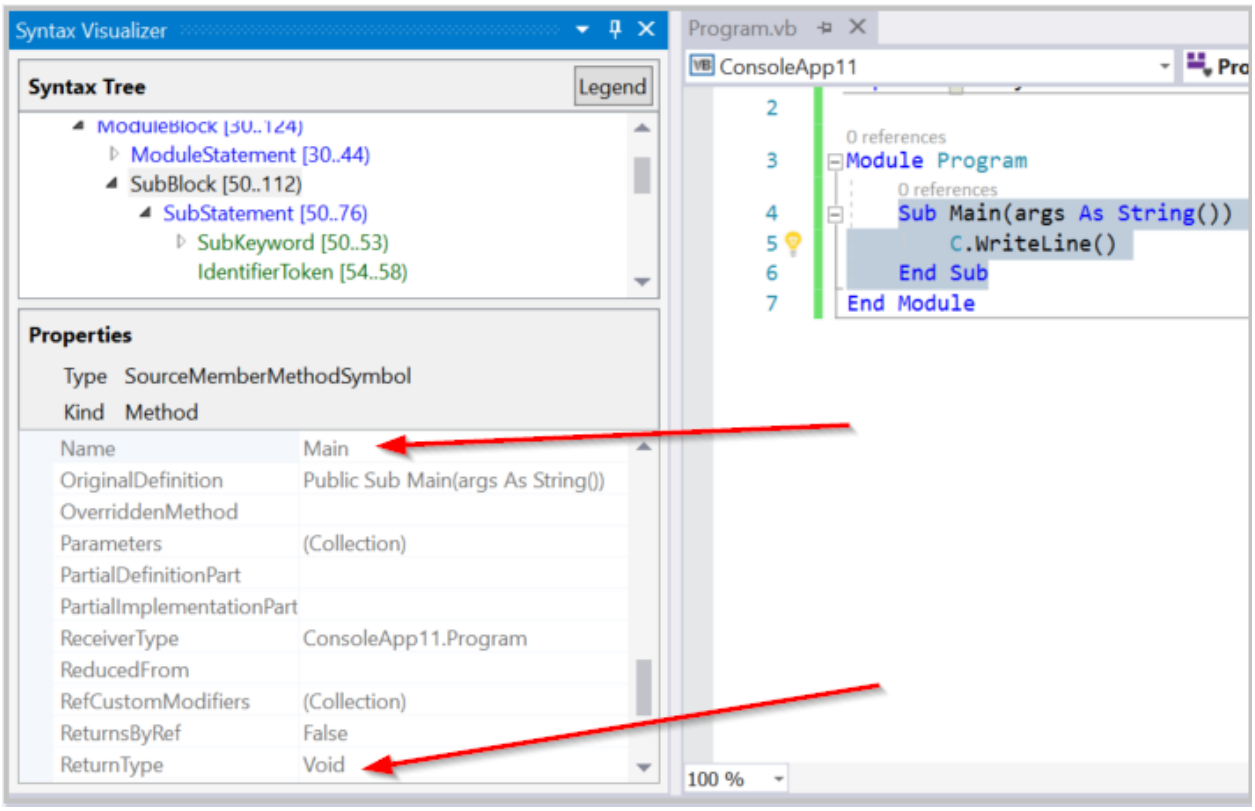
此代码引入了映射到文件顶部的 `System.Console` 类型的 `C` 别名，并在 `Main()` 内使用此别名。 选择在 `Main()` 方法的 `C.WriteLine()` 中使用此别名 `C`。 可视化工具会选择对应的 IdentifierName 节点 。 右键单击此节点，并单击“查看符号(如果有)” 。 属性网格指示此标识符绑定至 `System.Console` 类型，如下图所示：



针对同一 IdentifierName 节点，尝试“查看 AliasSymbol (如果有)” 。 属性网格指示该标识符为绑定至 `System.Console` 目标的别名 `C`。 换而言之，属性网格会提供对应于标识符 `C` 的 AliasSymbol 的相关信息 。



检查对应于任何声明的类型、方法和属性的符号。 在可视化工具中选择对应节点，单击“查看符号(如果有)”。 选择 `Sub Main()` 方法，包括该方法的正文。 针对可视化工具中对应的 SubBlock 节点，单击“查看符号(如果有)”。 属性网格显示此 SubBlock 节点的 MethodSymbol 的名称为 `Main`，返回类型为 `Void`。



在 C# 中可以轻松重复上述 Visual Basic 示例。 为别名键入 `using C = System.Console;` 以代替 `Imports C = System.Console`。 在 C# 中完成的上述步骤会在可视化工具窗口中产

生相同的结果。

语义检查操作只能用于节点。 不能用于标记和琐事。 并非所有节点都有相关的语义信息可供检查。 如果某个节点不具备相关的语义信息，单击“查看 * 符号(如果有)”会显示空白的属性网格。

可阅读使用语义概述文档，详细了解执行语义分析的 API。

关闭语法可视化工具

在不使用可视化工具窗口检查源代码时，可以关闭该窗口。 当你在浏览代码、编辑和更改源时，语法可视化工具会更新显示的内容。 在你没有使用它的时候，这种情况会分散人的注意力。

选择诊断 ID

项目 • 2023/12/22

诊断 ID 是与给定诊断关联的字符串，例如编译器错误或分析器生成的诊断。

ID 来自各种 API，例如：

- DiagnosticDescriptor.Id
- ObsoleteAttribute.DiagnosticId
- ExperimentalAttribute.DiagnosticId

诊断 ID 还用作源中的标识符，例如，在 #pragma 警告禁用或 .editorconfig 文件中。

注意事项

- 诊断 ID 应是唯一的
- 诊断 ID 必须是 C# 中的合法标识符
- 诊断 ID 长度应小于 15 个字符
- 诊断 ID 应采用 `<PREFIX><number>` 形式
  - 前缀特定于项目
  - 该数字表示特定的诊断

> **备注**
>
> 更改诊断 ID 是源中断性变更，因为如果 ID 发生更改，将会忽略现有抑制。

不要将前缀限制为两个字符（例如 `CSXXX` 和 `CAXXXX`）。 请改用较长的前缀来避免冲突。 例如，`System.*` 诊断使用 `SYSLIB` 作为其前缀。

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

语法分析入门

项目 • 2023/04/08

在本教程中，你将了解“语法 API” 。 语法 API 提供对描述 C# 或 Visual Basic 程序的数据结构的访问。 这些数据结构具备足够的详细信息，它们可以充分表示任何规模的任何程序。 这些结构可以描述准确编译并运行的完整程序。 当你在编辑器中编写程序时，它们还可以描述这些尚不完整的程序。

要启用此丰富的表达式，组成语法 API 的数据结构和 API 必然是复杂的。 让我们看看数据结构是什么样的，从典型的“Hello World”程序的数据结构开始：

C#

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

namespace HelloWorld
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Hello World!");
        }
    }
}
```

查看以前程序的文本。 识别熟悉的元素。 整个文本代表一个源文件，或者一个“编译单元” 。 源文件的前三行是 using 指令 。 剩余源包含在命名空间声明中 。 命名空间声明包含一个子类声明 。 类声明包含一个方法声明 。

语法 API 使用代表编译单元的根创建树结构。 树中的节点代表 using 指令、命名空间声明和程序中的所有其他元素。 树结构一直延伸到最底层：字符串“Hello World!”是一个字符串文本标记，是参数的子代。 语法 API 提供对程序结构的访问。 你可以查询特定代码实践、浏览整个树以理解代码并通过修改现有树来新建树。

该简介概述了使用语法 API 可以访问的信息类型。 语法 API 仅仅是描述你从 C# 中熟悉的代码结构的正式 API。 完整的功能包括关于设置代码格式（包括换行符、空格和缩进）的信息。 在人工程序员或编译者编写和读取代码时，你可以使用此信息完整表示代码。 使用此结构可以在深有意义的级别上与源代码进行交互。 它不再是文本字符串，而是代表 C# 程序结构的数据。

若要开始，需要安装 .NET 编译器平台 SDK：

安装说明 - Visual Studio 安装程序

在“Visual Studio 安装程序”中查找“.NET Compiler Platform SDK”有两种不同的方法：

使用 Visual Studio 安装程序进行安装 - 工作负荷视图

Visual Studio 扩展开发工作负荷中不会自动选择 .NET Compiler Platform SDK。 必须将其作为可选组件进行选择。

1. 运行“Visual Studio 安装程序”
2. 选择“修改”
3. 检查“Visual Studio 扩展开发”工作负荷。
4. 在摘要树中打开“Visual Studio 扩展开发”节点。
5. 选中“.NET Compiler Platform SDK”框。 将在可选组件最下面找到它。

还可使“DGML 编辑器”在可视化工具中显示关系图：

1. 在摘要树中打开“单个组件”节点。
2. 选中“DGML 编辑器”框

使用 Visual Studio 安装程序进行安装 - 各组件选项卡

1. 运行“Visual Studio 安装程序”
2. 选择“修改”
3. 选择“单个组件”选项卡
4. 选中“.NET Compiler Platform SDK”框。 将在“编译器、生成工具和运行时”部分最上方找到它。

还可使“DGML 编辑器”在可视化工具中显示关系图：

1. 选中“DGML 编辑器”框。 将在“代码工具”部分下找到它。

了解语法树

将语法 API 用于 C# 代码结构的所有分析。 语法 API 公开分析程序、语法树和用于分析并构造语法树的实用程序 。 这是搜索特定语法元素的代码或读取程序代码的方式。

语法树是 C# 和 Visual Basic 编译器用于理解 C# 和 Visual Basic 程序的数据结构。 语法树由生成项目时或开发人员按 F5 时所运行的分析程序生成。 语法树对语言完全保真；代码文件中的每一位信息都在树中。 将语法树写入文本会再现已分析的完全原始文本。 语法树也是不可变的； 一旦创建语法树，就不能再更改。 树的使用者可以在多个线程上对

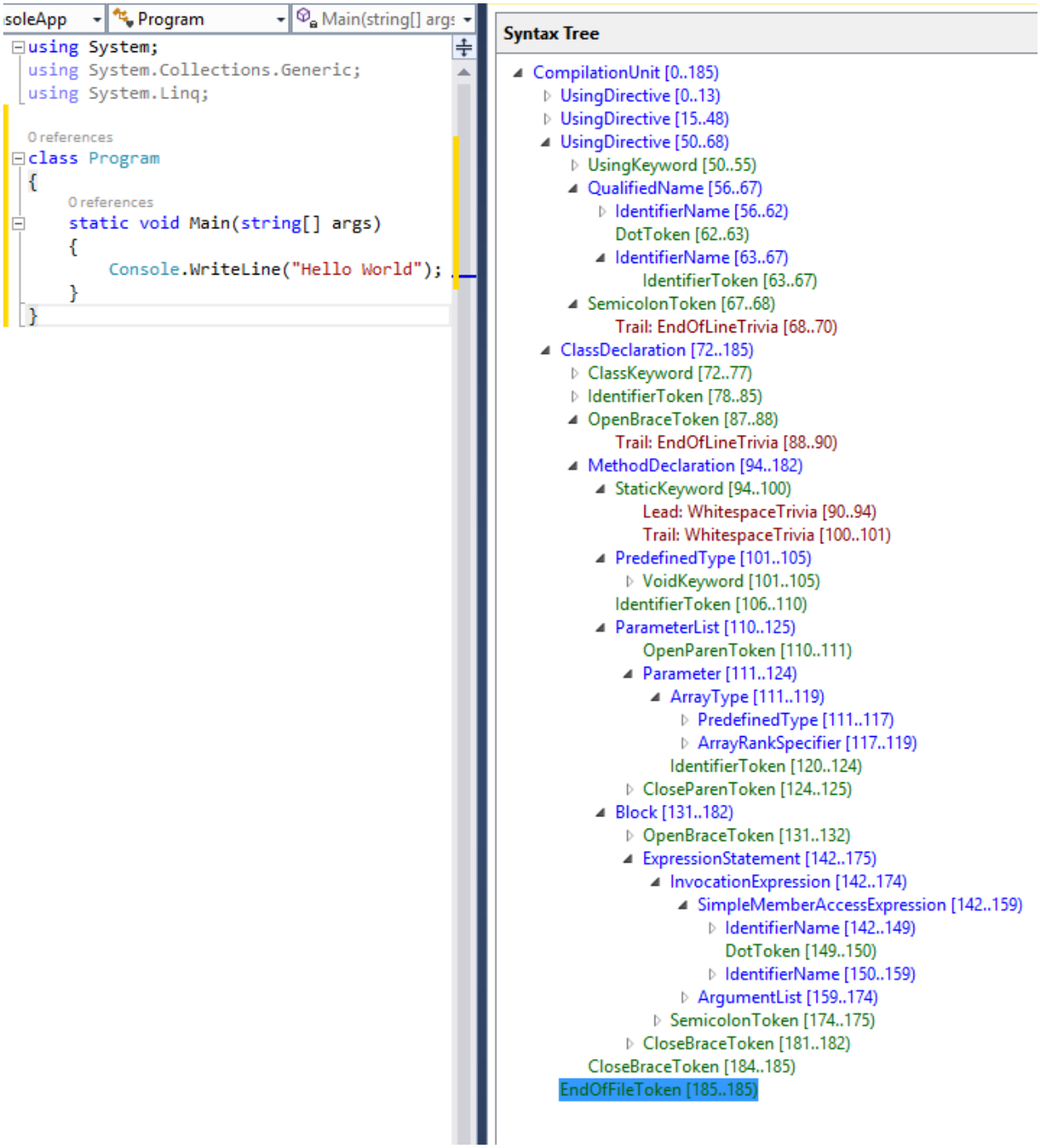
树进行分析，不需要锁或其他并发度量，很清楚数据是不会更改的。 可使用 API 新建树，新建的树就是对现有树进行修改后的成果。

语法树的四个主要构建基块为：

- Microsoft.CodeAnalysis.SyntaxTree 类，它的实例代表整个分析树。 SyntaxTree 是一种带有语言特定派生类的抽象类。 使用 Microsoft.CodeAnalysis.CSharp.CSharpSyntaxTree（或 Microsoft.CodeAnalysis.VisualBasic.VisualBasicSyntaxTree）类的分析方法对 C#（或 Visual Basic）中的文本进行分析。
- Microsoft.CodeAnalysis.SyntaxNode 类，它的实例表示声明、语句、子句和表达式等语法构造。
- Microsoft.CodeAnalysis.SyntaxToken 结构，它代表独立的关键词、标识符、运算符或标点。
- 最后是 Microsoft.CodeAnalysis.SyntaxTrivia 结构，它代表语法上不重要的信息，例如标记、预处理指令和注释之间的空格。

琐事、标记和节点分层次地构成了树，树完全代表了 Visual Basic 或 C# 代码片段中的所有内容。 你可以使用语法可视化工具窗口 查看此结构。 在 Visual Studio 中，选择“视图”>“其他窗口”>“语法可视化工具”。 例如使用语法可视化工具检查上述 C# 源文件，如下图所示：

SyntaxNode：蓝色 | SyntaxToken：绿色 | SyntaxTrivia：红色



通过在此树结构中导航，可以查找代码文件中的所有语句、表达式、标记或空格位。

在使用语法 API 查找代码文件中的内容时，大多数方案都涉及检查代码的小片段或者搜索特定语句或片段。 接下来的两个示例展示浏览代码结构或搜索单个语句的典型使用形式。

遍历树

你可通过两种方式检查语法树中的节点。 可以遍历该树以检查每个节点，也可以查询特定元素或节点。

手动遍历

可以在我们的 GitHub 存储库 中看到此示例的已完成代码。

> **备注**
>
> 语法树类型使用继承描述不同的语法元素，这些语法元素在程序中的不同位置生效。 使用这些 API 通常意味着将属性或集合成员强制转换为特定的派生类型。 在以下示例中，作业和强制转换分别是独立的语句，采用显式类型化变量。 你可以读取代码以查看 API 的返回类型以及所返回对象的运行时类型。 在实践中，更常见的是使用隐式类型化变量并靠 API 名称来描述要检查的对象的类型。

新建 C#"独立代码分析工具"项目：

- 在 Visual Studio 中，选择"文件" >"新建" >"项目"，显示新建项目对话框。
- 在"Visual C#" >"扩展性" 下，选择"独立代码分析工具"。
- 将项目命名为 SyntaxTreeManualTraversal，然后单击"确定"。

你将分析之前展示过的基本"Hello World!"程序。 为 Hello World 程序添加文本，作为 `Program` 类中的常量：

C#

```
        const string programText =
@"using System;
using System.Collections;
using System.Linq;
using System.Text;

namespace HelloWorld
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine(""Hello, World!"");
        }
    }
}";
```

下一步，添加下列代码以生成 `programText` 常量中的代码文本的语法树。 将下面这行代码添加到 `Main` 方法中：

C#

```
SyntaxTree tree = CSharpSyntaxTree.ParseText(programText);
CompilationUnitSyntax root = tree.GetCompilationUnitRoot();
```

这两行代码创建树并检索树的根节点。 现在，可以检查树的节点。 将这几行代码添加到 `Main` 方法以显示树中根节点的部分属性：

C#

```
WriteLine($"The tree is a {root.Kind()} node.");
WriteLine($"The tree has {root.Members.Count} elements in it.");
WriteLine($"The tree has {root.Usings.Count} using statements. They are:");
foreach (UsingDirectiveSyntax element in root.Usings)
    WriteLine($"\t{element.Name}");
```

运行应用程序，查看代码获取到的关于树中根节点的信息。

通常要遍历树以了解代码。 在此示例中，你通过分析已知代码探索 API。 添加下列代码检查 `root` 节点的第一个成员：

C#

```
MemberDeclarationSyntax firstMember = root.Members[0];
WriteLine($"The first member is a {firstMember.Kind()}.");
var helloWorldDeclaration = (NamespaceDeclarationSyntax)firstMember;
```

该成员为 Microsoft.CodeAnalysis.CSharp.Syntax.NamespaceDeclarationSyntax。 它代表 `namespace HelloWorld` 声明范围内的所有内容。 添加下列代码检查 `HelloWorld` 命名空间内声明了哪些节点：

C#

```
WriteLine($"There are {helloWorldDeclaration.Members.Count} members declared
in this namespace.");
WriteLine($"The first member is a
{helloWorldDeclaration.Members[0].Kind()}.");
```

运行程序查看你了解到的内容。

现在你了解声明为 Microsoft.CodeAnalysis.CSharp.Syntax.ClassDeclarationSyntax，声明一个该类型的新变量以检查类声明。 此类只包含一个成员：`Main` 方法。 添加以下代码找到 `Main` 方法，并将其强制转换为 Microsoft.CodeAnalysis.CSharp.Syntax.MethodDeclarationSyntax。

C#

```
var programDeclaration =
(ClassDeclarationSyntax)helloWorldDeclaration.Members[0];
WriteLine($"There are {programDeclaration.Members.Count} members declared in
the {programDeclaration.Identifier} class.");
WriteLine($"The first member is a {programDeclaration.Members[0].Kind()}.");
var mainDeclaration =
(MethodDeclarationSyntax)programDeclaration.Members[0];
```

方法声明节点包含关于该方法的所有语法信息。 让我们显示 `Main` 方法的返回类型、参数的数量和类型以及该方法的正文。 添加以下代码：

C#

```
WriteLine($"The return type of the {mainDeclaration.Identifier} method is
{mainDeclaration.ReturnType}.");
WriteLine($"The method has {mainDeclaration.ParameterList.Parameters.Count}
parameters.");
foreach (ParameterSyntax item in mainDeclaration.ParameterList.Parameters)
    WriteLine($"The type of the {item.Identifier} parameter is
{item.Type}.");
WriteLine($"The body text of the {mainDeclaration.Identifier} method
follows:");
WriteLine(mainDeclaration.Body?.ToFullString());

var argsParameter = mainDeclaration.ParameterList.Parameters[0];
```

运行程序，查看已获取的关于此程序的所有信息：

text

```
The tree is a CompilationUnit node.
The tree has 1 elements in it.
The tree has 4 using statements. They are:
        System
        System.Collections
        System.Linq
        System.Text
The first member is a NamespaceDeclaration.
There are 1 members declared in this namespace.
The first member is a ClassDeclaration.
There are 1 members declared in the Program class.
The first member is a MethodDeclaration.
The return type of the Main method is void.
The method has 1 parameters.
The type of the args parameter is string[].
The body text of the Main method follows:
        {
            Console.WriteLine("Hello, World!");
        }
```

查询方法

除了遍历树，还可以使用 Microsoft.CodeAnalysis.SyntaxNode 上定义的查询方法来探索语法树。 任何一个熟悉 XPath 的人都可以立刻掌握这些方法。 可以结合 LINQ 使用这些方法快速查找树中的内容。 SyntaxNode 具备类似 DescendantNodes、AncestorsAndSelf 和 ChildNodes 的查询方法。

你可以使用这些查询方法对 `Main` 方法查找参数，而不是在树中进行导航。 将以下代码添加到 `Main` 方法末尾：

C#

```
var firstParameters = from methodDeclaration in root.DescendantNodes()
                                        .OfType<MethodDeclarationSyntax>()
                      where methodDeclaration.Identifier.ValueText == "Main"
                      select
methodDeclaration.ParameterList.Parameters.First();

var argsParameter2 = firstParameters.Single();

WriteLine(argsParameter == argsParameter2);
```

第一个语句使用 LINQ 表达式和 DescendantNodes 方法查找与前面的示例相同的参数。

运行程序后能看到 LINQ 表达式已找到的参数，结果与树的手动导航一样。

此示例使用 `WriteLine` 语句，在遍历至语法树的相关信息时显示这些信息。 你还可以通过在调试程序下运行完成的程序了解更多内容。 你可以检查更多属性和方法，它们是为 Hello World 程序创建的语法树的一部分。

语法查看器

你经常需要查找语法树中特定类型的所有节点，例如某个文件中的每个属性声明。 通过扩展 Microsoft.CodeAnalysis.CSharp.CSharpSyntaxWalker 类并重写 VisitPropertyDeclaration(PropertyDeclarationSyntax) 方法，处理语法树中的每个属性声明，且事先无需了解它的结构。 CSharpSyntaxWalker 是 CSharpSyntaxVisitor 中的一个特定类型，它以递归方式访问节点以及节点的每个子级。

此示例实现了检查语法树的 CSharpSyntaxWalker。 它收集所找到的不导入 `System` 命名空间的 `using` 指令。

新建 C#“独立代码分析工具”项目 ，将其命名为 SyntaxWalker 。

可以在我们的 GitHub 存储库 中看到此示例的已完成代码。 GitHub 上的示例包含本教程介绍的两个项目。

如前面的示例所示，你可以定义字符串常量来保存将要分析的程序的文本：

C#

```
        const string programText =
@"using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using Microsoft.CodeAnalysis;
using Microsoft.CodeAnalysis.CSharp;

namespace TopLevel
{
    using Microsoft;
    using System.ComponentModel;

    namespace Child1
    {
        using Microsoft.Win32;
        using System.Runtime.InteropServices;

        class Foo { }
    }

    namespace Child2
    {
        using System.CodeDom;
        using Microsoft.CSharp;

        class Bar { }
    }
}";
```

此源文本包含的 `using` 指令分散在四个不同的位置：文件级、顶级命名空间以及两个嵌套命名空间。 此示例重点介绍使用 CSharpSyntaxWalker 类以查询代码的核心方案。 通过访问根语法树的每个节点来查找 using 声明会很麻烦。 替代方法是创建派生类，并改用只在树中的当前节点为 using 指令时才会调用的方法。 访问器不会在任何其他节点类型上做任何工作。 这一方法检查每个 `using` 语句并生成命名空间的集合，其中包含的命名空间都不在 `System` 命名空间中。 生成一个检查所有 `using` 语句（但仅检查 `using` 语句）的 CSharpSyntaxWalker。

现在，你已定义程序文本，需要创建 `SyntaxTree` 并获取该树的根：

C#

```
SyntaxTree tree = CSharpSyntaxTree.ParseText(programText);
CompilationUnitSyntax root = tree.GetCompilationUnitRoot();
```

接下来，创建一个新类。 在 Visual Studio 中，依次选择“项目” >“添加新项” 。 在“添加新项”对话框中键入 UsingCollector.cs 作为文件名 。

在 `UsingCollector` 类中实现 `using` 访问器功能。 首先，从 CSharpSyntaxWalker 派生 `UsingCollector` 类。

C#

```
class UsingCollector : CSharpSyntaxWalker
```

需要存储空间来保存收集的命名空间节点。 在 `UsingCollector` 类中声明公共只读属性；使用此变量来存储你找到的 UsingDirectiveSyntax 节点：

C#

```
public ICollection<UsingDirectiveSyntax> Usings { get; } = new
List<UsingDirectiveSyntax>();
```

基类，CSharpSyntaxWalker 实现访问语法树中每个节点的逻辑。 派生类重写你感兴趣的特定节点所调用的方法。 在这种情况下，你对任何 `using` 指令都感兴趣。 也就是说必须重写 VisitUsingDirective(UsingDirectiveSyntax) 方法。 此方法的一个参数是 Microsoft.CodeAnalysis.CSharp.Syntax.UsingDirectiveSyntax 对象。 这是使用访问器的一项重要优势：它们调用已重写的方法，这些方法所包含的参数已经强制转换为特定节点类型。 Microsoft.CodeAnalysis.CSharp.Syntax.UsingDirectiveSyntax 类有 Name 属性，该属性存储要导入的命名空间的名称。 它是一个 Microsoft.CodeAnalysis.CSharp.Syntax.NameSyntax。 在 VisitUsingDirective(UsingDirectiveSyntax) 重写中添加以下代码：

C#

```
public override void VisitUsingDirective(UsingDirectiveSyntax node)
{
    WriteLine($"\tVisitUsingDirective called with {node.Name}.");
    if (node.Name.ToString() != "System" &&
        !node.Name.ToString().StartsWith("System."))
    {
        WriteLine($"\t\tSuccess. Adding {node.Name}.");
        this.Usings.Add(node);
    }
}
```

如前面的示例所示，你已添加各种 `WriteLine` 语句来协助理解此方法。 你可以查看此方法的调用时间以及每次调用时向它传递的参数。

最后，需要添加两行代码以创建 `UsingCollector` 并让其访问根节点，收集所有 `using` 语句。 然后，添加 `foreach` 循环以显示收集器找到的所有 `using` 语句：

C#

```
var collector = new UsingCollector();
collector.Visit(root);
foreach (var directive in collector.Usings)
{
    WriteLine(directive.Name);
}
```

编译并运行该程序。 您应看到以下输出：

控制台

```
        VisitUsingDirective called with System.
        VisitUsingDirective called with System.Collections.Generic.
        VisitUsingDirective called with System.Linq.
        VisitUsingDirective called with System.Text.
        VisitUsingDirective called with Microsoft.CodeAnalysis.
                Success. Adding Microsoft.CodeAnalysis.
        VisitUsingDirective called with Microsoft.CodeAnalysis.CSharp.
                Success. Adding Microsoft.CodeAnalysis.CSharp.
        VisitUsingDirective called with Microsoft.
                Success. Adding Microsoft.
        VisitUsingDirective called with System.ComponentModel.
        VisitUsingDirective called with Microsoft.Win32.
                Success. Adding Microsoft.Win32.
        VisitUsingDirective called with System.Runtime.InteropServices.
        VisitUsingDirective called with System.CodeDom.
        VisitUsingDirective called with Microsoft.CSharp.
                Success. Adding Microsoft.CSharp.
Microsoft.CodeAnalysis
Microsoft.CodeAnalysis.CSharp
Microsoft
Microsoft.Win32
Microsoft.CSharp
Press any key to continue . . .
```

祝贺你！ 你已使用语法 API 查找特定类型的 C# 语句和 C# 源代码中的声明 。

语义分析入门

项目 • 2023/05/10

本教程假定你熟悉语法 API。 语法分析入门一文提供了详细介绍。

在本教程中，你将了解“符号”和“绑定 API”。 这些 API 提供关于程序语义含义的信息。 它们帮助你就程序中的符号所代表的类型进行问答。

需要安装 .NET Compiler Platform SDK：

安装说明 - Visual Studio 安装程序

在“Visual Studio 安装程序”中查找“.NET Compiler Platform SDK”有两种不同的方法：

使用 Visual Studio 安装程序进行安装 - 工作负荷视图

Visual Studio 扩展开发工作负荷中不会自动选择 .NET Compiler Platform SDK。 必须将其作为可选组件进行选择。

1. 运行“Visual Studio 安装程序”
2. 选择“修改”
3. 检查“Visual Studio 扩展开发”工作负荷。
4. 在摘要树中打开“Visual Studio 扩展开发”节点。
5. 选中“.NET Compiler Platform SDK”框。 将在可选组件最下面找到它。

还可使“DGML 编辑器”在可视化工具中显示关系图：

1. 在摘要树中打开“单个组件”节点。
2. 选中“DGML 编辑器”框

使用 Visual Studio 安装程序进行安装 - 各组件选项卡

1. 运行“Visual Studio 安装程序”
2. 选择“修改”
3. 选择“单个组件”选项卡
4. 选中“.NET Compiler Platform SDK”框。 将在“编译器、生成工具和运行时”部分最上方找到它。

还可使“DGML 编辑器”在可视化工具中显示关系图：

1. 选中“DGML 编辑器”框。 将在“代码工具”部分下找到它。

了解编译和符号

随着 .NET 编译器 SDK 的使用越来越多，你会越来越熟悉语法 API 和语义 API 之间的区别。 “语法 API”使你可以看到程序的结构。 但是经常会需要更多关于程序的语义或含义的信息。 虽然可以独立地对松散的代码文件或 Visual Basic 或 C# 代码的代码片段进行语法上的分析，但是凭空提出类似“这是什么类型的变量？”这样的问题毫无意义。 类型名称的含义可能取决于程序集引用、命名空间导入或其他代码文件。 使用**语义 API** 回答这些问题，特别是 Microsoft.CodeAnalysis.Compilation 类。

Compilation 实例类似于编译器所看见的单个项目，且代表编译 Visual Basic 或 C# 程序所需的一切。 编译包括一组要编译的源文件、程序集引用和编译器选项。 可以使用此上下文中所有的其他信息来推断代码的含义。 Compilation 允许你查找“符号” - 类似名称和其他表达式引用的类型、命名空间、成员和变量的实体。 将名称和表达式与“符号”进行关联的过程被称为“绑定”。

与 Microsoft.CodeAnalysis.SyntaxTree 类似，Compilation 是一个带有语言特定派生类的抽象类。 在创建编译实例时，必须在 Microsoft.CodeAnalysis.CSharp.CSharpCompilation（或 Microsoft.CodeAnalysis.VisualBasic.VisualBasicCompilation）类上调用工厂方法。

查询符号

在本教程中，你会再次看到“Hello World”程序。 这次你将在该程序中查询符号，以理解这些符号所代表的类型。 在命名空间中查询类型，并学习如何查找类型上可用的方法。

可以在我们的 GitHub 存储库 中看到此示例的已完成代码。

> **备注**
>
> 语法树类型使用继承描述不同的语法元素，这些语法元素在程序中的不同位置生效。 使用这些 API 通常意味着将属性或集合成员强制转换为特定的派生类型。 在以下示例中，作业和强制转换分别是独立的语句，采用显式类型化变量。 你可以读取代码以查看 API 的返回类型以及所返回对象的运行时类型。 在实践中，更常见的是使用隐式类型化变量并靠 API 名称来描述要检查的对象的类型。

新建 C#“独立代码分析工具”项目：

- 在 Visual Studio 中，选择“文件” >“新建” >“项目”，显示新建项目对话框。
- 在“Visual C#” >“扩展性”下，选择“独立代码分析工具”。
- 将项目命名为“SemanticQuickStart”并单击“确定”。

你将分析之前展示过的基本“Hello World!”程序。 为 Hello World 程序添加文本，作为 `Program` 类中的常量：

C#

```
        const string programText =
@"using System;
using System.Collections.Generic;
using System.Text;

namespace HelloWorld
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine(""Hello, World!"");
        }
    }
}";
```

下一步，添加以下代码，为 `programText` 常量中的代码文本生成语法树。 将下面这行代码添加到 `Main` 方法中：

C#

```
SyntaxTree tree = CSharpSyntaxTree.ParseText(programText);

CompilationUnitSyntax root = tree.GetCompilationUnitRoot();
```

接下来，从已创建的树生成 CSharpCompilation。 “Hello World”示例依赖于 String 和 Console 类型。 需要引用在编译中声明这两种类型的程序集。 将下面这行代码添加到 `Main` 方法以创建语法树的编译，包括对相应程序集的引用：

C#

```
var compilation = CSharpCompilation.Create("HelloWorld")
    .AddReferences(MetadataReference.CreateFromFile(
        typeof(string).Assembly.Location))
    .AddSyntaxTrees(tree);
```

CSharpCompilation.AddReferences 方法将引用添加到编译。
MetadataReference.CreateFromFile 方法加载程序集作为引用。

查询语义模型

如果有 Compilation，你可以向它查询 Compilation 所包含的任何 SyntaxTree 的 SemanticModel。 你可将语义模型看作通常从智能感知中获取的所有信息的来源。 SemanticModel 可回答“此位置中范围内的名称是什么？”、“可通过此方法访问哪些成

员？”、“此文本块中使用了什么变量？”和“此名称/表达式指的是什么？”等问题。添加此声明以创建语义模型：

```
C#

SemanticModel model = compilation.GetSemanticModel(tree);
```

绑定名称

Compilation 从 SyntaxTree 创建 SemanticModel。 创建模型后，你可以查询以找到第一个 `using` 指令，并检索 `System` 命名空间的符号信息。 将这两行代码添加到 `Main` 方法以创建语义模型，并检索第一个 using 语句的符号：

```
C#

// Use the syntax tree to find "using System;"
UsingDirectiveSyntax usingSystem = root.Usings[0];
NameSyntax systemName = usingSystem.Name;

// Use the semantic model for symbol information:
SymbolInfo nameInfo = model.GetSymbolInfo(systemName);
```

上述代码示例演示如何绑定第一个 `using` 指令中的名称以检索 `System` 命名空间的 Microsoft.CodeAnalysis.SymbolInfo。 上述代码还说明，使用语法模型查找代码的结构；使用语义模型理解它的含义。 `System`语法模型在 using 语句中找到字符串 。 `System`语义模型具有关于 命名空间中所定义类型的全部信息。

可以从 SymbolInfo 对象获取使用 SymbolInfo.Symbol 属性的 Microsoft.CodeAnalysis.ISymbol。 此属性返回此表达式所引用的符号。 对于不引用任何内容的表达式（例如数字参数），此属性为 `null`。 若 SymbolInfo.Symbol 不为 null，ISymbol.Kind 表示符号的类型。 在此示例中，ISymbol.Kind 属性是 SymbolKind.Namespace。 将以下代码添加到 `Main` 方法。 它检索 `System` 命名空间的符号，然后将 `System` 命名空间中声明的所有子命名空间显示出来：

```
C#

var systemSymbol = (INamespaceSymbol?)nameInfo.Symbol;
if (systemSymbol?.GetNamespaceMembers() is not null)
{
    foreach (INamespaceSymbol ns in systemSymbol?.GetNamespaceMembers()!)
    {
        Console.WriteLine(ns);
    }
}
```

运行该程序，然后应看到以下输出：

输出

```
System.Collections
System.Configuration
System.Deployment
System.Diagnostics
System.Globalization
System.IO
System.Numerics
System.Reflection
System.Resources
System.Runtime
System.Security
System.StubHelpers
System.Text
System.Threading
Press any key to continue . . .
```

> **备注**
>
> 该输出不包含 `System` 命名空间的每个子命名空间。 它显示此编译中存在的每个命名空间，只引用声明了 `System.String` 的程序集。 此编译不了解其他程序集中所声明的任何命名空间

绑定表达式

上述代码显示如何通过绑定到名称来查找符号。 在可以绑定的 C# 程序中还有其他不是名称的表达式。 为演示此功能，我们绑定到简单的字符串文本。

“Hello World”程序包含 Microsoft.CodeAnalysis.CSharp.Syntax.LiteralExpressionSyntax，即向控制台显示的“Hello, World!”字符串。

通过在程序中查找单个字符串文本，可以找到“Hello, World!”字符串。 找到语法节点后，从语义模型中获取该节点的类型信息。 将以下代码添加到 `Main` 方法：

C#

```
// Use the syntax model to find the literal string:
LiteralExpressionSyntax helloWorldString = root.DescendantNodes()
.OfType<LiteralExpressionSyntax>()
.Single();

// Use the semantic model for type information:
TypeInfo literalInfo = model.GetTypeInfo(helloWorldString);
```

Microsoft.CodeAnalysis.TypeInfo 结构包括 TypeInfo.Type 属性，此属性可启用对关于文本类型的语义信息的访问。 在此例中为 `string` 类型。 添加将此属性分配至本地变量的声明：

```
C#

var stringTypeSymbol = (INamedTypeSymbol?)literalInfo.Type;
```

要完成本教程，让我们生成 LINQ 查询，该查询会创建一个在 `string` 类型上声明并返回 `string` 的所有公共方法的序列。 此查询比较复杂，我们将逐行生成，然后将其重新构造为单个查询。 此查询的源是 `string` 类型上所声明全部成员的序列：

```
C#

var allMembers = stringTypeSymbol?.GetMembers();
```

该源序列包含所有成员（包括属性和字段），使用 ImmutableArray<T>.OfType 方法对其进行筛选以查找属于 Microsoft.CodeAnalysis.IMethodSymbol 对象的元素：

```
C#

var methods = allMembers?.OfType<IMethodSymbol>();
```

接下来，添加另一个筛选器，只返回这些属于公共方法并返回 `string` 的方法：

```
C#

var publicStringReturningMethods = methods?
    .Where(m => SymbolEqualityComparer.Default.Equals(m.ReturnType,
stringTypeSymbol) &&
    m.DeclaredAccessibility == Accessibility.Public);
```

只选择名称属性，且只通过删除任何重载来区分名称：

```
C#

var distinctMethods = publicStringReturningMethods?.Select(m =>
m.Name).Distinct();
```

还可以使用 LINQ 查询语法生成完整查询，然后在控制台中显示所有方法名称：

```
C#
```

```
foreach (string name in (from method in stringTypeSymbol?
                         .GetMembers().OfType<IMethodSymbol>()
                         where
SymbolEqualityComparer.Default.Equals(method.ReturnType, stringTypeSymbol)
&&
                         method.DeclaredAccessibility ==
Accessibility.Public
                         select method.Name).Distinct())
{
    Console.WriteLine(name);
}
```

生成并运行程序。 应会看到以下输出：

输出

```
Join
Substring
Trim
TrimStart
TrimEnd
Normalize
PadLeft
PadRight
ToLower
ToLowerInvariant
ToUpper
ToUpperInvariant
ToString
Insert
Replace
Remove
Format
Copy
Concat
Intern
IsInterned
Press any key to continue . . .
```

你已使用语义 API 来查找并显示关于属于此程序的符号的信息。

语法转换入门

项目 • 2023/05/09

本教程基于语法分析入门和语义分析入门快速入门中介绍的概念和技巧。 如果尚未执行此操作，应在开始之前完成这些快速入门。

在本快速入门教程，你将了解用于创建和转换语法树的技巧。 结合你在前面的快速入门中了解的技巧，可以创建第一个命令行重构！

安装说明 - Visual Studio 安装程序

在“Visual Studio 安装程序”中查找“.NET Compiler Platform SDK”有两种不同的方法：

使用 Visual Studio 安装程序进行安装 - 工作负荷视图

Visual Studio 扩展开发工作负荷中不会自动选择 .NET Compiler Platform SDK。 必须将其作为可选组件进行选择。

1. 运行“Visual Studio 安装程序”
2. 选择“修改”
3. 检查“Visual Studio 扩展开发”工作负荷。
4. 在摘要树中打开“Visual Studio 扩展开发”节点。
5. 选中“.NET Compiler Platform SDK”框。 将在可选组件最下面找到它。

还可使“DGML 编辑器”在可视化工具中显示关系图：

1. 在摘要树中打开“单个组件”节点。
2. 选中“DGML 编辑器”框

使用 Visual Studio 安装程序进行安装 - 各组件选项卡

1. 运行“Visual Studio 安装程序”
2. 选择“修改”
3. 选择“单个组件”选项卡
4. 选中“.NET Compiler Platform SDK”框。 将在“编译器、生成工具和运行时”部分最上方找到它。

还可使“DGML 编辑器”在可视化工具中显示关系图：

1. 选中“DGML 编辑器”框。 将在“代码工具”部分下找到它。

不可变性和 .NET 编译器平台

不可变性是 .NET 编译器平台的基本原则。 在创建不可变数据结构后，不能对其进行更改。 不可变数据结构可以由多个使用者同时安全地共享和分析。 一个使用者以不可预测的方式影响另一个使用者不存在任何风险。 分析器不需要锁或其他并发度量值。 此规则适用于语法树、编译、符号、语义模型，以及你所遇到的所有其他数据结构。 API 不是修改现有结构，而是基于旧结构的特定差异创建新对象。 将此概念应用到语法树中，以使用转换来创建新树。

创建和转换树

选择两个策略之一进行语法转换。 当你在寻找要替换的特定节点时，或者想要在其中插入新代码的特定位置时，最好使用工厂方法。 当你想要扫描一个你想要替换的代码模式的整个项目时，最好使用重写工具。

使用工厂方法创建节点

第一个语法转换演示工厂方法。 将 `using System.Collections;` 语句替换为 `using System.Collections.Generic;` 语句。 此示例演示如何使用 Microsoft.CodeAnalysis.CSharp.SyntaxFactory 工厂方法创建 Microsoft.CodeAnalysis.CSharp.CSharpSyntaxNode 对象。 对于每一类节点、令牌或琐事，都有创建该类型实例的工厂方法。 可以通过以自下而上的方式按层次结构组合节点来创建语法树。 然后，转换现有程序，用你所创建的新树替换现有节点。

启动 Visual Studio，并新建 C#“独立代码分析工具”项目。 在 Visual Studio 中，选择“文件” >“新建” >“项目”，显示新建项目对话框。 在“Visual C#”>“扩展性”下，选择“独立代码分析工具”。 本快速入门教程有两个示例项目，因此将解决方案命名为“SyntaxTransformationQuickStart”，并将项目命名为“ConstructionCS”。 单击 “**确定**” 。

此项目使用 Microsoft.CodeAnalysis.CSharp.SyntaxFactory 类方法构造 Microsoft.CodeAnalysis.CSharp.Syntax.NameSyntax 来表示 `System.Collections.Generic` 命名空间。

将以下 using 指令添加到 `Program.cs` 顶部。

C#

```
using static Microsoft.CodeAnalysis.CSharp.SyntaxFactory;
using static System.Console;
```

创建 **命名语法节点** 以创建表示 `using System.Collections.Generic;` 语句的树。 NameSyntax 是在 C# 中显示的四种类型名称的基类。 将这四种类型名称组合在一起，以创建任何可通过 C# 语言显示的名称：

- Microsoft.CodeAnalysis.CSharp.Syntax.NameSyntax，表示简单的单个标识符名称，如 `System` 和 `Microsoft`。
- Microsoft.CodeAnalysis.CSharp.Syntax.GenericNameSyntax，表示泛型类型或方法名称，如 `List<int>`。
- Microsoft.CodeAnalysis.CSharp.Syntax.QualifiedNameSyntax，表示窗体 `<left-name>.<right-identifier-or-generic-name>` 的限定名称，如 `System.IO`。
- Microsoft.CodeAnalysis.CSharp.Syntax.AliasQualifiedNameSyntax，表示使用程序集外部别名的名称，如 `LibraryV2::Foo`。

若要创建 NameSyntax 节点，请使用 IdentifierName(String) 方法。 在 `Program.cs` 的 `Main` 方法中添加以下代码：

C#

```
NameSyntax name = IdentifierName("System");
WriteLine($"\tCreated the identifier {name}");
```

前面的代码创建 IdentifierNameSyntax 对象，并将其分配给变量 `name`。 许多 Roslyn API 返回基类，使其更轻松地处理相关类型。 变量 `name`，即 NameSyntax，可以在生成 QualifiedNameSyntax 时重用。 在生成示例时，不要使用类型推理。 你将自动执行此项目中的这一步。

你已创建名称。 现在，可以通过构建 QualifiedNameSyntax 在树中生成更多节点。 新树使用 `name` 作为左侧名称，并使用 `Collections` 命名空间新的 IdentifierNameSyntax 作为 QualifiedNameSyntax 的右侧。 将下列代码添加到 `program.cs`：

C#

```
name = QualifiedName(name, IdentifierName("Collections"));
WriteLine(name.ToString());
```

再次运行代码并查看结果。 你将构建一个表示代码的节点树。 你将继续运行此模式，以便生成命名空间 `System.Collections.Generic` 的 QualifiedNameSyntax。 将下列代码添加到 `Program.cs`：

C#

```
name = QualifiedName(name, IdentifierName("Generic"));
WriteLine(name.ToString());
```

再次运行此程序，以查看你已为要添加的代码生成的树。

创建修改后的树

你已构建一个小型语法树，其中包含一个语句。 用于创建新节点的 API 是创建单个语句或其他小代码块的正确选择。 但是，若要生成更大的代码块，应使用替换节点或将节点插入到现有树的方法。 请记住语法树不可变。 语法 API 不提供用于完成构造后修改现有语法树的任何机制。 相反，它提供基于对现有数的更改生成新树的方法。 `With*` 方法在派生自 SyntaxNode 的具体类中定义，或者在 SyntaxNodeExtensions 类中声明的扩展方法中定义。 这些方法通过将更改应用于现有节点的子属性来创建一个新节点。 此外，ReplaceNode 扩展方法可以用来替换子树中的子代节点。 此方法还会更新父结点，以指向新创建的子节点，并在整个树中重复这一过程，该过程称为“重新遍历树”。

下一步是创建一个表示整个（小型）程序的树，然后修改它。 将以下代码添加到 `Program` 类的开头：

C#

```
        private const string sampleCode =
@"using System;
using System.Collections;
using System.Linq;
using System.Text;

namespace HelloWorld
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine(""Hello, World!"");
        }
    }
}";
```

> **备注**
>
> 该示例代码使用 `System.Collections` 命名空间而不是 `System.Collections.Generic` 命名空间。

接下来，将以下代码添加到 `Main` 方法的底部来分析文本，并创建树：

C#

```
SyntaxTree tree = CSharpSyntaxTree.ParseText(sampleCode);
var root = (CompilationUnitSyntax)tree.GetRoot();
```

此示例使用 WithName(NameSyntax) 方法将 UsingDirectiveSyntax 节点中的名称替换为在前面代码中构造的名称。

使用 WithName(NameSyntax) 方法创建一个新的 UsingDirectiveSyntax 节点，将 `System.Collections` 名称更新为在前面代码中创建的名称。 将以下代码添加到 `Main` 方法底部：

C#

```
var oldUsing = root.Usings[1];
var newUsing = oldUsing.WithName(name);
WriteLine(root.ToString());
```

运行程序，并仔细查看输出。 `newUsing` 尚未置于根树中。 原始树尚未更改。

使用 ReplaceNode 扩展方法添加以下代码以创建新树。 新树是将现有导入替换为更新后的 `newUsing` 节点的结果。 将此新树分配给现有 `root` 变量：

C#

```
root = root.ReplaceNode(oldUsing, newUsing);
WriteLine(root.ToString());
```

再次运行程序。 现在，树正确地导入了 `System.Collections.Generic` 命名空间。

使用 `SyntaxRewriters` 转换树

`With*` 和 ReplaceNode 方法提供了方便的方法来转换语法树的单独分支。 Microsoft.CodeAnalysis.CSharp.CSharpSyntaxRewriter 类在语法树上执行多个转换。 Microsoft.CodeAnalysis.CSharp.CSharpSyntaxRewriter 类是 Microsoft.CodeAnalysis.CSharp.CSharpSyntaxVisitor<TResult> 的一个子类。 CSharpSyntaxRewriter 将转换应用于特定类型的 SyntaxNode。 你可以将转换应用于多个类型的 SyntaxNode 对象，只要它们显示在语法树中。 本快速入门教程中的第二个项目创建命令行重构，以便在可以使用类型推理的任何位置删除本地变量声明中的显式类型。

新建 C#“独立代码分析工具”项目。 在 Visual Studio 中，右键单击 `SyntaxTransformationQuickStart` 解决方案节点。 选择“添加”>“新项目”以显示“新项目对话框”。 在“Visual C#” >“扩展性” 下，选择“独立代码分析工具” 。 给项目 `TransformationCS` 命名，然后单击“确定”。

第一步是创建一个派生自 CSharpSyntaxRewriter 的类，以执行转换。 向项目添加一个新类文件。 在 Visual Studio 中，依次选择“项目”>“添加类...”。在“添加新项”对话框中键入 `TypeInferenceRewriter.cs` 作为文件名。

使用指令将以下内容添加到 `TypeInferenceRewriter.cs` 文件：

C#

```
using Microsoft.CodeAnalysis;
using Microsoft.CodeAnalysis.CSharp;
using Microsoft.CodeAnalysis.CSharp.Syntax;
```

接下来，使 `TypeInferenceRewriter` 类扩展 CSharpSyntaxRewriter 类：

C#

```
public class TypeInferenceRewriter : CSharpSyntaxRewriter
```

添加以下代码以声明一个私有只读字段，以保存 SemanticModel 并在构造函数中将其初始化。 稍后你将需要此字段以确定可以使用类型推理的位置：

C#

```
private readonly SemanticModel SemanticModel;

public TypeInferenceRewriter(SemanticModel semanticModel) => SemanticModel =
semanticModel;
```

重写 VisitLocalDeclarationStatement(LocalDeclarationStatementSyntax) 方法：

C#

```
public override SyntaxNode
VisitLocalDeclarationStatement(LocalDeclarationStatementSyntax node)
{

}
```

> ⓘ **备注**
>
> 许多 Roslyn API 声明返回类型，它们是返回的实际运行时类型的基类。 在许多情况下，一种类型的节点可能会被另一种节点完全替换，甚至删除。 在此示例中，VisitLocalDeclarationStatement(LocalDeclarationStatementSyntax) 方法返回

SyntaxNode，而不是派生类型的 LocalDeclarationStatementSyntax。 此重写工具根据现有节点返回一个新的 LocalDeclarationStatementSyntax。

本快速入门教程处理本地变量声明。 你无法将其扩展到其他声明，如 `foreach` 循环、`for` 循环、LINQ 表达式和 lambda 表达式。 此外，此重写工具仅转换最简单形式的声明：

C#

```
Type variable = expression;
```

如果想要自行浏览，请考虑扩展这些类型变量声明的已完成示例：

C#

```
// Multiple variables in a single declaration.
Type variable1 = expression1,
     variable2 = expression2;
// No initializer.
Type variable;
```

将以下代码添加到 `VisitLocalDeclarationStatement` 方法主体以跳过重写这些形式的声明：

C#

```
if (node.Declaration.Variables.Count > 1)
{
    return node;
}
if (node.Declaration.Variables[0].Initializer == null)
{
    return node;
}
```

该方法表明，通过返回未修改的 `node` 参数没有发生重写。 如果上述两个 `if` 表达式都为 true，则该节点表示一个可能的初始化声明。 添加这些语句以提取声明中指定的类型名称，并使用 SemanticModel 字段将其绑定来获取类型符号：

C#

```
var declarator = node.Declaration.Variables.First();
var variableTypeName = node.Declaration.Type;

var variableType = (ITypeSymbol)SemanticModel
```

```
    .GetSymbolInfo(variableTypeName)
    .Symbol;
```

现在，添加此语句以绑定初始值设定项表达式：

C#

```
var initializerInfo =
SemanticModel.GetTypeInfo(declarator.Initializer.Value);
```

最后，如果初始值设定项表达式的类型与指定类型相匹配，则添加以下 `if` 语句，将现有类型名称替换为 `var` 关键字：

C#

```
if (SymbolEqualityComparer.Default.Equals(variableType,
initializerInfo.Type))
{
    TypeSyntax varTypeName = SyntaxFactory.IdentifierName("var")
        .WithLeadingTrivia(variableTypeName.GetLeadingTrivia())
        .WithTrailingTrivia(variableTypeName.GetTrailingTrivia());

    return node.ReplaceNode(variableTypeName, varTypeName);
}
else
{
    return node;
}
```

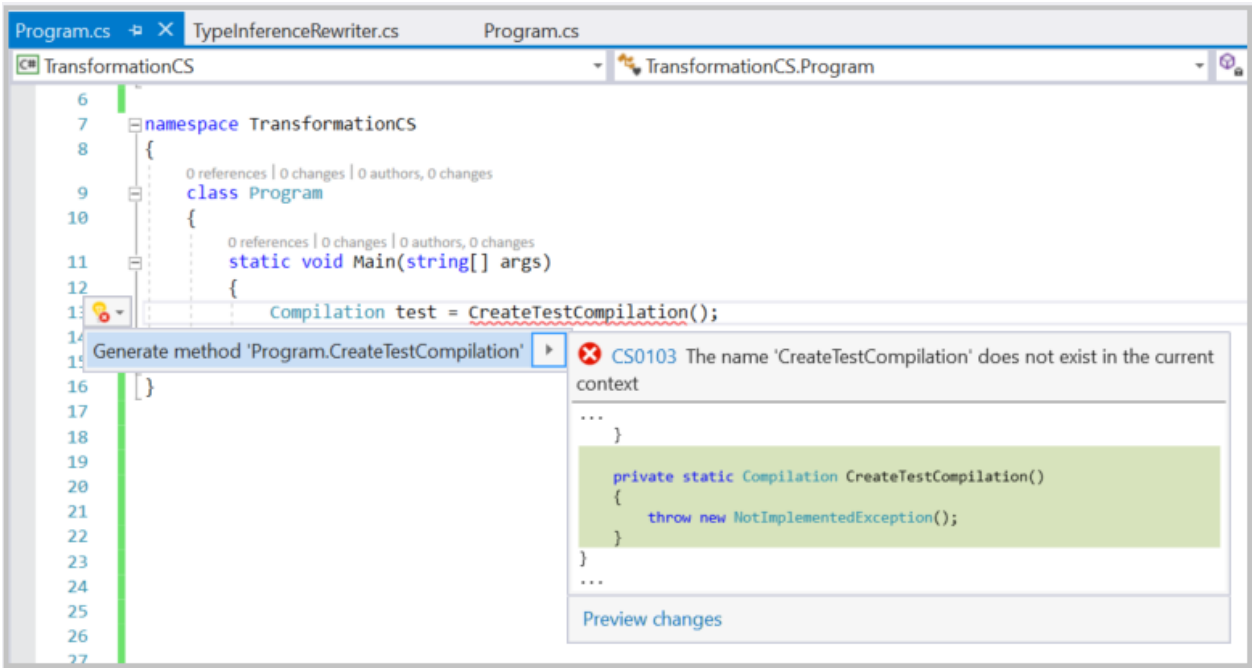
此条件是必需的，因为声明可能将初始值设定项表达式转换为基类或接口。 如果这是需要的，则分配左侧和右侧的类型不匹配。 在这些情况下删除显式类型将更改程序语义。 `var` 指定为标识符而不是关键字，因为 `var` 是上下文关键字。 前导和尾随琐事（空白）从旧类型名转换为 `var` 关键字以保持垂直空白和缩进。 使用 `ReplaceNode`（而非 `With*` 来转换 LocalDeclarationStatementSyntax 更为简单，因为类型名称实际上是声明语句的孙级。

你已完成 `TypeInferenceRewriter`。 现在返回到 `Program.cs` 文件来完成该示例。 创建测试 Compilation 并从中获取 SemanticModel。 使用该 SemanticModel 尝试 `TypeInferenceRewriter`。 你将在最后执行此步骤。 在此期间，声明一个表示测试编译的占位符变量：

C#

```
Compilation test = CreateTestCompilation();
```

暂停一段时间后，应会看到错误波形曲线，报告不存在 `CreateTestCompilation` 方法。 按 Ctrl+句点打开灯泡，然后按 Enter 以调用“生成方法存根(Stub)”命令。 此命令将在 `Program` 类中生成 `CreateTestCompilation` 方法的方法存根(Stub)。 稍后你将返回填写此方法：



编写以下代码以循环访问测试 Compilation 中的每个 SyntaxTree。 对于每一个，都使用 SemanticModel 初始化该树的新 `TypeInferenceRewriter`：

C#

```
foreach (SyntaxTree sourceTree in test.SyntaxTrees)
{
    SemanticModel model = test.GetSemanticModel(sourceTree);

    TypeInferenceRewriter rewriter = new TypeInferenceRewriter(model);

    SyntaxNode newSource = rewriter.Visit(sourceTree.GetRoot());

    if (newSource != sourceTree.GetRoot())
    {
        File.WriteAllText(sourceTree.FilePath, newSource.ToFullString());
    }
}
```

在你创建的 `foreach` 语句中，添加以下代码以在每个源树上执行转换。 如果进行了任何编辑，这段代码将有条件地写出新的转换树。 如果遇到一个或多个可以使用类型推理进行简化的本地变量声明，则重写工具应该只修改一个树：

C#

```
SyntaxNode newSource = rewriter.Visit(sourceTree.GetRoot());

if (newSource != sourceTree.GetRoot())
{
    File.WriteAllText(sourceTree.FilePath, newSource.ToFullString());
}
```

应看到 `File.WriteAllText` 代码下的波形曲线。 选择灯泡，并添加所需的 `using System.IO;` 语句。

即将完成！ 还剩一步，即创建测试 Compilation。 因为你在本快速入门教程期间尚未使用类型推理，所以它将是一个完美的测试用例。 遗憾的是，从 C# 项目文件中创建编译不在本演练范围内。 但幸运的是，如果你已仔细按照说明进行操作，那还是有希望的。 将 `CreateTestCompilation` 方法的内容替换为以下代码。 它将创建一个与本快速入门教程所述的项目相匹配的测试编译：

C#

```
String programPath = @"..\..\..\Program.cs";
String programText = File.ReadAllText(programPath);
SyntaxTree programTree =
               CSharpSyntaxTree.ParseText(programText)
                               .WithFilePath(programPath);

String rewriterPath = @"..\..\..\TypeInferenceRewriter.cs";
String rewriterText = File.ReadAllText(rewriterPath);
SyntaxTree rewriterTree =
               CSharpSyntaxTree.ParseText(rewriterText)
                               .WithFilePath(rewriterPath);

SyntaxTree[] sourceTrees = { programTree, rewriterTree };

MetadataReference mscorlib =
        MetadataReference.CreateFromFile(typeof(object).Assembly.Location);
MetadataReference codeAnalysis =

MetadataReference.CreateFromFile(typeof(SyntaxTree).Assembly.Location);
MetadataReference csharpCodeAnalysis =

MetadataReference.CreateFromFile(typeof(CSharpSyntaxTree).Assembly.Location)
;

MetadataReference[] references = { mscorlib, codeAnalysis,
csharpCodeAnalysis };

return CSharpCompilation.Create("TransformationCS",
    sourceTrees,
    references,
    new CSharpCompilationOptions(OutputKind.ConsoleApplication));
```

运行项目，祈求好运吧。 在 Visual Studio 中，选择“调试”>“启动调试”。 应该会收到 Visual Studio 的提醒，指示项目中的文件已更改。 单击“全部同意”以重载已修改的文件。 检查这些文件以观察效果。 请注意，如果没有所有那些显式和冗余类型的说明符，代码会看起来更加简洁。

祝贺你！ 你已使用编译器 API 编写你自己的重构，以便在 C# 项目的所有文件中搜索某些语法模式、分析匹配这些模式的源代码语义，并对其进行转换。 现在，你已正式成为重构作者了！

教程：编写第一个分析器和代码修补程序

项目 • 2023/05/10

.NET Compiler Platform SDK 提供面向 C# 或 Visual Basic 代码创建自定义诊断（分析器）、代码修补程序、代码重构和诊断抑制器所需的工具。 分析器包含可识别规则冲突的代码。 代码修补程序包含修复冲突的代码。 实现的规则可以是从代码结构到编码样式再到命名约定之类的任何内容。 .NET Compiler Platform 在开发人员编写代码时提供运行分析的框架，以及用于修复代码的所有 Visual Studio UI 功能：显示编辑器中的波形曲线、填充 Visual Studio 错误列表、创建“灯泡”建议，并显示建议修补程序的丰富预览。

在本教程中，将探讨使用 Roslyn API 创建分析器以及随附的代码修补程序。 分析器是一种执行源代码分析并向用户报告问题的方法。 可以选择将代码修补程序与分析器相关联，来表示对用户源代码的修改。 本教程将创建一个分析器，用于查找可以使用 `const` 修饰符声明的但未执行此操作的局部变量声明。 随附的代码修补程序修改这些声明来添加 `const` 修饰符。

先决条件

- Visual Studio 2019 版本 16.8 或更高版本

必须通过 Visual Studio 安装程序安装 .NET 编译器平台 SDK：

安装说明 - Visual Studio 安装程序

在“Visual Studio 安装程序”中查找“.NET Compiler Platform SDK”有两种不同的方法：

使用 Visual Studio 安装程序进行安装 - 工作负荷视图

Visual Studio 扩展开发工作负荷中不会自动选择 .NET Compiler Platform SDK。 必须将其作为可选组件进行选择。

1. 运行“Visual Studio 安装程序”
2. 选择“修改”
3. 检查“Visual Studio 扩展开发”工作负荷。
4. 在摘要树中打开“Visual Studio 扩展开发”节点。
5. 选中“.NET Compiler Platform SDK”框。 将在可选组件最下面找到它。

还可使“DGML 编辑器”在可视化工具中显示关系图：

1. 在摘要树中打开“单个组件”节点。

2. 选中“DGML 编辑器”框

使用 Visual Studio 安装程序进行安装 - 各组件选项卡

1. 运行“Visual Studio 安装程序”
2. 选择“修改”
3. 选择“单个组件”选项卡
4. 选中“.NET Compiler Platform SDK”框。 将在“编译器、生成工具和运行时”部分最上方找到它。

还可使“DGML 编辑器”在可视化工具中显示关系图：

1. 选中“DGML 编辑器”框。 将在“代码工具”部分下找到它。

可以通过几个步骤创建和验证分析器：

1. 创建解决方案。
2. 注册分析器名称和描述。
3. 报告分析器警告和建议。
4. 实现代码修复以接受建议。
5. 通过单元测试改进分析。

创建解决方案

- 在 Visual Studio 中，选择“文件”>“新建”>“项目...”，显示“新建项目”对话框。
- 在“Visual C#”>“扩展性”下，选择“随附代码修补程序的分析器 (.NET Standard)”。
- 给项目“MakeConst”命名，然后单击“确定”。

> **备注**
>
> 你可能会收到错误（MSB4062: 无法加载 "CompareBuildTaskVersion" 任务）。 若要解决此问题，请使用 NuGet 包管理器或在包管理器控制台窗口中使用 `Update-Package` 更新解决方案中的 NuGet 包。

探索分析器模板

随附代码修补程序的分析器模板会创建五个项目：

- MakeConst，其中包含分析器。
- MakeConst.CodeFixes，其中包含代码修补程序。
- MakeConst.Package，用于生成分析器和代码修补程序的 NuGet 包。

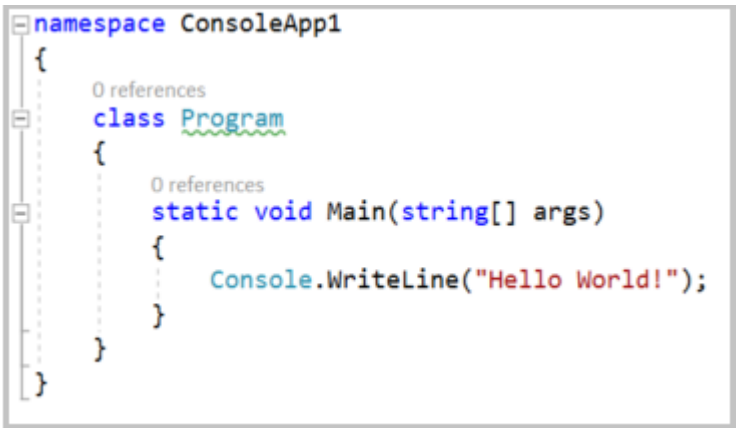
- MakeConst.Test，这是一个单元测试项目。
- MakeConst.Vsix，这是默认的启动项目，它将启动加载了新分析器的第二个 Visual Studio 实例。 按 F5 启动 VSIX 项目。

> **备注**
>
> 分析器应以 .NET Standard 2.0 为目标，因为它们可在 .NET Core 环境（命令行生成）和 .NET Framework 环境 (Visual Studio) 中运行。

> **提示**
>
> 在运行分析器时，请启动 Visual Studio 的第二个副本。 此第二个副本使用不同的注册表配置单元来存储设置。 这样便可以将 Visual Studio 两个副本中的可视化设置区分开来。 可以选择 Visual Studio 实验性运行的不同主题。 此外，不要在设置中漫游，也不要使用 Visual Studio 的实验性运行登录到 Visual Studio 帐户。 这样可以使设置保持不同。
>
> 该配置单元不仅包括正在开发的分析器，而且还包括任何以前打开的分析器。 若要重置 Roslyn 配置单元，需要从 %LocalAppData%\Microsoft\VisualStudio 中手动将其删除。 Roslyn 配置单元的文件夹名称将以 `Roslyn` 结尾，例如 `16.0_9ae182f9Roslyn`。 请注意，你可能需要在删除配置单元后清除解决方案并重新生成。

在刚刚启动的第二个 Visual Studio 实例中，创建一个新的 C# 控制台应用程序项目（任何目标框架都可用 -- 分析器在源级别工作。）悬停在带波浪下划线的标记上，将显示分析器提供的警告文本。

该模板创建一个分析器，它报告有关类型名称包含小写字母的每种类型声明的警告，如下图所示：



该模板还提供了一种代码修补程序，它可以将包含小写字符的任何类型名称更改为大写字母。 可以单击显示警告的灯泡，以查看建议的更改。 接受建议的更改会更新解决方案中

的类型名称和所有对该类型的引用。 现在你已了解初始分析器的操作，关闭第二个 Visual Studio 实例，并返回到分析器项目。

无需启动 Visual Studio 的第二个副本和创建新代码来测试分析器中的每一项更改。 该模板还为你创建了单元测试项目。 该项目包含两个测试。 `TestMethod1` 显示了在不触发诊断的情况下分析代码的典型测试格式。 `TestMethod2` 显示了先触发诊断然后应用建议的代码修补程序的测试格式。 在构建分析器和代码修补程序时，为不同的代码结构编写测试，以验证你的工作。 分析器的单元测试比使用 Visual Studio 以交互方式进行测试的速度更快。

> **提示**
>
> 当你知道哪些代码构造应触发和不应触发分析器时，分析器单元测试是一个很好的工具。 在 Visual Studio 的另一个副本加载分析器是用于浏览并找到你可能未曾想到的构造的绝佳工具。

在本教程中，你将编写一个分析器，用于向用户报告可以转换为局部常量的任何局部变量声明。 例如，考虑以下代码：

```
C#

int x = 0;
Console.WriteLine(x);
```

在上面的代码中，会向 `x` 分配常量值，并且永远不会被修改。 可以使用 `const` 修饰符声明：

```
C#

const int x = 0;
Console.WriteLine(x);
```

涉及到确定变量是否可以保持不变的分析，需要进行句法分析、初始值设定项的常量分析和数据流分析，以确保永远不会写入该变量。 .NET Compiler Platform 提供了 API，以便更轻松地执行此分析。

创建分析器注册

该模板将在 MakeConstAnalyzer.cs 文件中创建初始 `DiagnosticAnalyzer` 类。 此初始分析器显示每个分析器的两个重要属性。

- 每个诊断分析器必须提供 `[DiagnosticAnalyzer]` 属性，用于描述其操作所用的语言。
- 每个诊断分析器都必须是（直接或间接地）从 DiagnosticAnalyzer 类派生的。

该模板还显示属于任何分析器的基本功能：

1. 注册操作。 此操作表示应触发分析器以检查存在冲突的代码的代码更改。 当 Visual Studio 检测到匹配注册操作的代码编辑时，它调用分析器的注册方法。
2. 创建诊断。 当分析器检测到冲突，它会创建一个诊断对象，Visual Studio 使用该对象来通知用户有关冲突的信息。

在重写的 DiagnosticAnalyzer.Initialize(AnalysisContext) 方法中注册操作。 在本教程中，你将访问“语法节点”寻找局部声明，并查看哪些具有常量值。 如果声明可以是常量，分析器将创建并报告诊断。

第一步是更新注册常量和 `Initialize` 方法，以便这些常量指示“Make Const”分析器。 大多数字符串常量在字符串资源文件中定义。 应遵循此做法，以便更轻松地实现本地化。 打开“MakeConst”分析器项目的“Resources.resx”。 将显示资源编辑器。 更新字符串资源，如下所示：

- 将 `AnalyzerDescription` 更改为“Variables that are not modified should be made constants.”。
- 将 `AnalyzerMessageFormat` 更改为“Variable '{0}' can be made constant”。
- 将 `AnalyzerTitle` 更改为“Variable can be made constant”。

完成后，资源编辑器应如下图所示：

Name	Value	Comment
AnalyzerDescription	Variables that are not modified should be made constants.	An optional longer localizable description of the diagnostic.
AnalyzerMessageFormat	Variable '{0}' can be made constant	The format-able message the diagnostic displays.
AnalyzerTitle	Variable can be made constant	The title of the diagnostic.

剩余的更改在分析器文件中。 在 Visual Studio 中打开“MakeConstAnalyzer.cs”。 将注册操作从作用于符号的操作更改为作用于语法的操作。 在 `MakeConstAnalyzerAnalyzer.Initialize` 方法中，找到在符号上注册操作的行：

C#

```
context.RegisterSymbolAction(AnalyzeSymbol, SymbolKind.NamedType);
```

使用下面的行替换它：

C#

```
context.RegisterSyntaxNodeAction(AnalyzeNode,
SyntaxKind.LocalDeclarationStatement);
```

完成此更改后，可以删除 `AnalyzeSymbol` 方法。 此分析器检查 SyntaxKind.LocalDeclarationStatement，而不是 SymbolKind.NamedType 语句。 请注意，`AnalyzeNode` 下面有红色波浪线。 刚添加的代码引用未声明的 `AnalyzeNode` 方法。 使用以下代码声明该方法：

C#

```
private void AnalyzeNode(SyntaxNodeAnalysisContext context)
{
}
```

将 `Category` 更改为 MakeConstAnalyzer.cs 中的“Usage”，如以下代码所示：

C#

```
private const string Category = "Usage";
```

查找可以是常量的局部声明

可以开始编写 `AnalyzeNode` 方法的第一个版本了。 应查找可以是 `const` 但实际不是的单个局部声明，如以下代码所示：

C#

```
int x = 0;
Console.WriteLine(x);
```

第一步是查找局部声明。 将以下代码添加到 MakeConstAnalyzer.cs 中的 `AnalyzeNode`：

C#

```
var localDeclaration = (LocalDeclarationStatementSyntax)context.Node;
```

此强制转换始终会成功，因为分析器注册了对局部声明的更改，并且只注册了局部声明。 没有其他节点类型会触发对 `AnalyzeNode` 方法的调用。 接下来，检查任何 `const` 修饰符的声明。 一旦找到，请立即返回。 以下代码用于查找局部声明上的任何 `const` 修饰符：

C#

```
// make sure the declaration isn't already const:
if (localDeclaration.Modifiers.Any(SyntaxKind.ConstKeyword))
{
    return;
}
```

最后，需要检查变量是否可能是 `const`。 这意味着确保在其初始化后永远不会对其赋值。

将使用 SyntaxNodeAnalysisContext 执行一些语义分析。 使用 `context` 参数确定局部变量声明是否可为 `const`。 Microsoft.CodeAnalysis.SemanticModel 表示单个源文件中的所有语义信息。 可参阅涵盖了语义模型的文章了解详细信息。 将使用 Microsoft.CodeAnalysis.SemanticModel 在局部声明语句上执行数据流分析。 然后，使用此数据流分析的结果确保局部变量不会在任何其他位置用新值来编写。 调用 GetDeclaredSymbol 扩展方法来检索变量的 ILocalSymbol，并检查确认它不包含在数据流分析的 DataFlowAnalysis.WrittenOutside 集中。 在 `AnalyzeNode` 方法的末尾添加以下代码：

C#

```
// Perform data flow analysis on the local declaration.
DataFlowAnalysis dataFlowAnalysis =
context.SemanticModel.AnalyzeDataFlow(localDeclaration);

// Retrieve the local symbol for each variable in the local declaration
// and ensure that it is not written outside of the data flow analysis
region.
VariableDeclaratorSyntax variable =
localDeclaration.Declaration.Variables.Single();
ISymbol variableSymbol = context.SemanticModel.GetDeclaredSymbol(variable,
context.CancellationToken);
if (dataFlowAnalysis.WrittenOutside.Contains(variableSymbol))
{
    return;
}
```

刚添加的代码可确保变量不会修改，并因此可以进行 `const` 操作。 现在可以引发诊断了。 将以下代码添加为 `AnalyzeNode` 的最后一行：

C#

```
context.ReportDiagnostic(Diagnostic.Create(Rule, context.Node.GetLocation(),
localDeclaration.Declaration.Variables.First().Identifier.ValueText));
```

可以通过按 F5 运行分析器来检查进度。 可以加载前面创建的控制台应用程序，然后添加以下测试代码：

C#

```
int x = 0;
Console.WriteLine(x);
```

应显示灯泡，且分析器应报告诊断。 但是，根据你的 Visual Studio 版本，你将看到：

- 灯泡，它仍使用模板生成的代码修补程序，并告知你可以用大写。
- 位于编辑器顶部的横幅消息，它显示“MakeConstCodeFixProvider”遇到错误，已被禁用。 这是因为代码修复提供程序还未发生更改，仍希望找到 `TypeDeclarationSyntax` 元素而不是 `LocalDeclarationStatementSyntax` 元素。

下一部分将说明如何编写代码修补程序。

编写代码修补程序

分析器可以提供一个或多个代码修补程序。 代码修补程序定义解决报告问题的编辑。 对于你创建的分析器，可以提供将插入 const 关键字的代码修补程序：

diff

```
- int x = 0;
+ const int x = 0;
Console.WriteLine(x);
```

用户从编辑器的灯泡 UI 中选择它，Visual Studio 更改代码。

打开 CodeFixResources.resx 文件，并将 `CodeFixTitle` 更改为“Make constant”。

打开由模板添加的“MakeConstCodeFixProvider.cs”文件。 此代码修补程序已绑定到由诊断分析器生成的诊断 ID，但它尚没有实施正确的代码转换。

接下来，删除 `MakeUppercaseAsync` 方法。 它不再适用。

所有代码修复提供程序都派生自 CodeFixProvider。 它们都重写 CodeFixProvider.RegisterCodeFixesAsync(CodeFixContext) 以报告可用的代码修补程序。 在 `RegisterCodeFixesAsync` 中，将正在搜索的上级节点类型更改为 LocalDeclarationStatementSyntax 以匹配诊断：

C#

```
var declaration =
root.FindToken(diagnosticSpan.Start).Parent.AncestorsAndSelf().OfType<LocalD
eclarationStatementSyntax>().First();
```

接下来，更改用于注册代码修补程序的最后一行。 修补程序将创建新的文档，该文档通过将 `const` 修饰符添加到现有声明生成：

C#

```
// Register a code action that will invoke the fix.
context.RegisterCodeFix(
    CodeAction.Create(
        title: CodeFixResources.CodeFixTitle,
        createChangedDocument: c => MakeConstAsync(context.Document,
declaration, c),
        equivalenceKey: nameof(CodeFixResources.CodeFixTitle)),
    diagnostic);
```

你会注意到刚在符号 `MakeConstAsync` 上添加的代码中的红色波浪线。 添加的 `MakeConstAsync` 声明如以下代码所示：

C#

```
private static async Task<Document> MakeConstAsync(Document document,
    LocalDeclarationStatementSyntax localDeclaration,
    CancellationToken cancellationToken)
{
}
```

新的 `MakeConstAsync` 方法会将表示用户源文件的 Document 转换到现包含 `const` 声明的新 Document。

创建一个新的 `const` 关键字标记，以在声明语句的开头处插入。 请注意，首先从声明语句的第一个标记中删除任何前导琐碎内容，然后将其附加到 `const` 标记。 将以下代码添加到 `MakeConstAsync` 方法中：

C#

```
// Remove the leading trivia from the local declaration.
SyntaxToken firstToken = localDeclaration.GetFirstToken();
SyntaxTriviaList leadingTrivia = firstToken.LeadingTrivia;
LocalDeclarationStatementSyntax trimmedLocal =
localDeclaration.ReplaceToken(
    firstToken, firstToken.WithLeadingTrivia(SyntaxTriviaList.Empty));

// Create a const token with the leading trivia.
```

```
SyntaxToken constToken = SyntaxFactory.Token(leadingTrivia,
SyntaxKind.ConstKeyword,
SyntaxFactory.TriviaList(SyntaxFactory.ElasticMarker));
```

接下来，使用以下代码向声明添加 `const` 标记：

C#

```
// Insert the const token into the modifiers list, creating a new modifiers
list.
SyntaxTokenList newModifiers = trimmedLocal.Modifiers.Insert(0, constToken);
// Produce the new local declaration.
LocalDeclarationStatementSyntax newLocal = trimmedLocal
    .WithModifiers(newModifiers)
    .WithDeclaration(localDeclaration.Declaration);
```

接下来，设置要匹配 C# 格式设置规则的新声明的格式。 对所做的更改进行格式设置以匹配现有代码，这可创建更好的体验。 紧接着在现有代码后面添加以下语句：

C#

```
// Add an annotation to format the new local declaration.
LocalDeclarationStatementSyntax formattedLocal =
newLocal.WithAdditionalAnnotations(Formatter.Annotation);
```

此代码需要新命名空间。 将下面的 `using` 指令添加到文件的顶部：

C#

```
using Microsoft.CodeAnalysis.Formatting;
```

最后一步是进行编辑。 此过程包括三个步骤：

1. 获取现有文档的句柄。
2. 通过将现有声明替换为新声明来创建一个新文档。
3. 返回新文档。

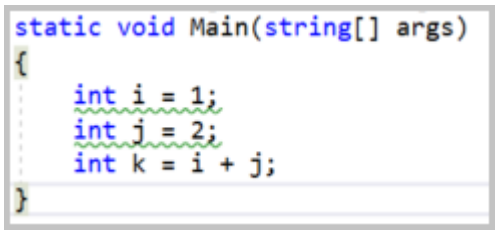
在 `MakeConstAsync` 方法的末尾添加以下代码：

C#

```
// Replace the old local declaration with the new local declaration.
SyntaxNode oldRoot = await
document.GetSyntaxRootAsync(cancellationToken).ConfigureAwait(false);
SyntaxNode newRoot = oldRoot.ReplaceNode(localDeclaration, formattedLocal);
```

```
// Return document with transformed tree.
return document.WithSyntaxRoot(newRoot);
```

代码修补程序已准备就绪。 按 F5 在第二个 Visual Studio 实例中运行分析器项目。 在第二个 Visual Studio 实例中，创建一个新的 C# 控制台应用程序项目并向 Main 方法添加使用常量值初始化的几个局部变量声明。 你将看到它们被报告为警告，如下所示。



现在已经有了很大的进展。 可以进行 `const` 操作的声明下具有波浪线。 但仍有工作要做。 如果将 `const` 添加到依次以 `i`、`j` 和 `k` 开头的声明，该过程会很有效。 不过，如果以从 `k` 开始的不同顺序添加 `const` 修饰符，分析器会生成错误：`k` 无法声明为 `const`，除非 `i` 和 `j` 均已进行 `const` 处理。 必须执行详细分析，以确保处理可以声明和初始化变量的不同方式。

生成单元测试

分析器和代码修补程序在简单的单个声明情况下工作，可以对其进行 const 处理。 在许多可能的声明语句中，该实现会出错。 可以通过使用模板编写的单元测试库来处理这种情况。 它要比反复打开 Visual Studio 的第二个副本快得多。

打开单元测试项目中的“MakeConstUnitTests.cs”文件。 该模板会创建两个测试，这些测试遵循分析器和代码修补程序单元测试的两种常见模式。 `TestMethod1` 显示测试模式，确保分析器在不应报告诊断的情况下不会执行此操作。 `TestMethod2` 演示用于报告诊断和运行代码修补程序的模式。

该模板使用 Microsoft.CodeAnalysis.Testing 包进行单元测试。

> **提示**
>
> 测试库支持特殊标记语法，其中包括以下内容：
>
> - `[|text|]`：表示报告 `text` 的诊断信息。 默认情况下，此格式只可用于测试由 `DiagnosticAnalyzer.SupportedDiagnostics` 提供了正好一个 `DiagnosticDescriptor` 的分析器。
> - `{|ExpectedDiagnosticId:text|}`：表示针对 `text` 报告 **Id** 为 `ExpectedDiagnosticId` 的诊断信息。

将 `MakeConstUnitTest` 类中的模板测试替换为以下测试方法：

C#

```
        [TestMethod]
        public async Task LocalIntCouldBeConstant_Diagnostic()
        {
            await VerifyCS.VerifyCodeFixAsync(@"
using System;

class Program
{
    static void Main()
    {
        [|int i = 0;|]
        Console.WriteLine(i);
    }
}
", @"
using System;

class Program
{
    static void Main()
    {
        const int i = 0;
        Console.WriteLine(i);
    }
}
");
        }
```

运行此测试，确保测试通过。 在 Visual Studio 中，通过选择“测试”>“Windows”>“测试资源管理器”来打开“测试资源管理器”。 然后，选择“全部运行”。

为有效声明创建测试

作为一般规则，分析器应尽可能使工作量最小化以快速退出。 Visual Studio 调用注册分析器作为用户编辑代码。 响应能力是一项关键要求。 有多个代码测试用例，不应引发诊断。 分析器已处理这些测试中的一个，其中变量在初始化后进行了分配。 添加以下测试方法来表示这种情况：

C#

```
        [TestMethod]
        public async Task VariableIsAssigned_NoDiagnostic()
        {
            await VerifyCS.VerifyAnalyzerAsync(@"
using System;
```

```
class Program
{
    static void Main()
    {
        int i = 0;
        Console.WriteLine(i++);
    }
}
");
        }
```

此测试也通过了。 接下来，为尚未处理的情况添加测试方法：

- 已经是 `const` 的声明，因为它们已为 const 类型：

  C#

  ```
          [TestMethod]
          public async Task VariableIsAlreadyConst_NoDiagnostic()
          {
              await VerifyCS.VerifyAnalyzerAsync(@"
  using System;

  class Program
  {
      static void Main()
      {
          const int i = 0;
          Console.WriteLine(i);
      }
  }
  ");
          }
  ```

- 没有初始值设定项的声明，因为没有要使用的值：

  C#

  ```
          [TestMethod]
          public async Task NoInitializer_NoDiagnostic()
          {
              await VerifyCS.VerifyAnalyzerAsync(@"
  using System;

  class Program
  {
      static void Main()
      {
          int i;
          i = 0;
  ```

```
            Console.WriteLine(i);
        }
    }
    ");
            }
```

- 初始值设定项不是常量的声明，因为它们不能是编译时常量：

C#

```
            [TestMethod]
            public async Task InitializerIsNotConstant_NoDiagnostic()
            {
                await VerifyCS.VerifyAnalyzerAsync(@"
    using System;

    class Program
    {
        static void Main()
        {
            int i = DateTime.Now.DayOfYear;
            Console.WriteLine(i);
        }
    }
    ");
            }
```

甚至可能更加复杂，因为 C# 允许多个声明作为一条语句。 请考虑以下测试用例字符串常量：

C#

```
            [TestMethod]
            public async Task MultipleInitializers_NoDiagnostic()
            {
                await VerifyCS.VerifyAnalyzerAsync(@"
    using System;

    class Program
    {
        static void Main()
        {
            int i = 0, j = DateTime.Now.DayOfYear;
            Console.WriteLine(i);
            Console.WriteLine(j);
        }
    }
    ");
            }
```

变量 `i` 可以常量化，但变量 `j` 不能。 因此，此语句不能成为 const 声明。

再次运行测试，将看到这些新测试用例失败。

更新分析器以忽略正确声明

需要对分析器的 `AnalyzeNode` 方法进行一些增强，以筛选出匹配这些条件的代码。 它们是所有相关条件，因此类似的更改将解决所有这些条件。 对 `AnalyzeNode` 进行以下更改：

- 语义分析检查单个变量声明。 此代码必须位于 `foreach` 循环中，以检查同一语句中声明的所有变量。
- 每个声明变量需要有初始值设定项。
- 每个声明的变量的初始值设定项必须是编译时常量。

在 `AnalyzeNode` 方法中，替换原始语义分析：

C#

```
// Perform data flow analysis on the local declaration.
DataFlowAnalysis dataFlowAnalysis =
context.SemanticModel.AnalyzeDataFlow(localDeclaration);

// Retrieve the local symbol for each variable in the local declaration
// and ensure that it is not written outside of the data flow analysis
region.
VariableDeclaratorSyntax variable =
localDeclaration.Declaration.Variables.Single();
ISymbol variableSymbol = context.SemanticModel.GetDeclaredSymbol(variable,
context.CancellationToken);
if (dataFlowAnalysis.WrittenOutside.Contains(variableSymbol))
{
    return;
}
```

使用以下代码片段：

C#

```
// Ensure that all variables in the local declaration have initializers that
// are assigned with constant values.
foreach (VariableDeclaratorSyntax variable in
localDeclaration.Declaration.Variables)
{
    EqualsValueClauseSyntax initializer = variable.Initializer;
    if (initializer == null)
    {
        return;
```

```
    }

    Optional<object> constantValue =
context.SemanticModel.GetConstantValue(initializer.Value,
context.CancellationToken);
    if (!constantValue.HasValue)
    {
        return;
    }
}

// Perform data flow analysis on the local declaration.
DataFlowAnalysis dataFlowAnalysis =
context.SemanticModel.AnalyzeDataFlow(localDeclaration);

foreach (VariableDeclaratorSyntax variable in
localDeclaration.Declaration.Variables)
{
    // Retrieve the local symbol for each variable in the local declaration
    // and ensure that it is not written outside of the data flow analysis
region.
    ISymbol variableSymbol =
context.SemanticModel.GetDeclaredSymbol(variable,
context.CancellationToken);
    if (dataFlowAnalysis.WrittenOutside.Contains(variableSymbol))
    {
        return;
    }
}
```

第一个 `foreach` 循环将使用语法分析检查每个变量声明。 第一次检查可保证该变量具有初始值设定项。 第二次检查可保证初始值设定项是一个常量。 第二个循环具有原始语义分析。 语义检查是在一个单独循环中，因为它对性能具有更大的影响。 再次运行测试，应看到它们全部通过。

添加最后的润饰

即将完成。 分析器还要处理一些其他条件。 用户编写代码时，Visual Studio 将调用分析器。 通常情况下，分析器将针对无法进行编译的代码进行调用。 诊断分析器的 `AnalyzeNode` 方法不会检查以查看常量值是否可转换为变量类型。 因此，当前实现会不假思索地将不正确的声明（如 `int i = "abc"`）转换为局部常量。 为这种情况添加测试方法：

C#

```
        [TestMethod]
        public async Task DeclarationIsInvalid_NoDiagnostic()
        {
            await VerifyCS.VerifyAnalyzerAsync(@"
```

```
using System;

class Program
{
    static void Main()
    {
        int x = {|CS0029:""abc""|};
    }
}
");
        }
```

此外，无法正确处理引用类型。 允许用于引用类型的唯一常量值为 `null`，System.String 这种情况除外，后者允许字符串。 换而言之，`const string s = "abc"` 是合法的，但 `const object s = "abc"` 不是。 此代码片段验证以下条件：

C#

```
        [TestMethod]
        public async Task DeclarationIsNotString_NoDiagnostic()
        {
            await VerifyCS.VerifyAnalyzerAsync(@"
using System;

class Program
{
    static void Main()
    {
        object s = ""abc"";
    }
}
");
        }
```

为全面起见，需要添加另一个测试以确保你可以为字符串创建常量声明。 以下代码片段定义引发诊断的代码和在应用修补程序后的代码：

C#

```
        [TestMethod]
        public async Task StringCouldBeConstant_Diagnostic()
        {
            await VerifyCS.VerifyCodeFixAsync(@"
using System;

class Program
{
    static void Main()
    {
        [|string s = ""abc"";|]
```

```
    }
}
", @"
using System;

class Program
{
    static void Main()
    {
        const string s = ""abc"";
    }
}
");
        }
```

最后，如使用关键字 `var` 声明变量，代码修补程序将执行错误操作，并生成 `const var` 声明，C# 语言不支持该声明。 若要修复此 bug，代码修补程序必须将 `var` 关键字替换为推断类型的名称：

C#

```
        [TestMethod]
        public async Task VarIntDeclarationCouldBeConstant_Diagnostic()
        {
            await VerifyCS.VerifyCodeFixAsync(@"
using System;

class Program
{
    static void Main()
    {
        [|var item = 4;|]
    }
}
", @"
using System;

class Program
{
    static void Main()
    {
        const int item = 4;
    }
}
");
        }

        [TestMethod]
        public async Task VarStringDeclarationCouldBeConstant_Diagnostic()
        {
            await VerifyCS.VerifyCodeFixAsync(@"
using System;
```

```
class Program
{
    static void Main()
    {
        [|var item = ""abc"";|]
    }
}
", @"
using System;

class Program
{
    static void Main()
    {
        const string item = ""abc"";
    }
}
");
        }
```

幸运的是，所有上述 bug 可以使用你刚刚了解的相同技术解决。

若要修复第一个 bug，请先打开“MakeConstAnalyzer.cs”，并找到 foreach 循环，将检查其中每个局部声明的初始值设定项以确保向其分配常量值。 在第一个 foreach 循环之前，立即调用 `context.SemanticModel.GetTypeInfo()` 来检索有关局部声明的声明类型的详细信息：

C#

```
TypeSyntax variableTypeName = localDeclaration.Declaration.Type;
ITypeSymbol variableType =
context.SemanticModel.GetTypeInfo(variableTypeName,
context.CancellationToken).ConvertedType;
```

然后，在 `foreach` 循环中，检查每个初始值设定项，以确保它可以转换为变量类型。 确保初始值设定项为常量后，添加以下检查：

C#

```
// Ensure that the initializer value can be converted to the type of the
// local declaration without a user-defined conversion.
Conversion conversion =
context.SemanticModel.ClassifyConversion(initializer.Value, variableType);
if (!conversion.Exists || conversion.IsUserDefined)
{
    return;
}
```

下一次更改建立在最后一次更改之上。 在第一个 foreach 循环的右大括号前，添加以下代码以检查当常量为字符串或 NULL 时局部声明的类型。

C#

```
// Special cases:
//  * If the constant value is a string, the type of the local declaration
//    must be System.String.
//  * If the constant value is null, the type of the local declaration must
//    be a reference type.
if (constantValue.Value is string)
{
    if (variableType.SpecialType != SpecialType.System_String)
    {
        return;
    }
}
else if (variableType.IsReferenceType && constantValue.Value != null)
{
    return;
}
```

必须在代码修复提供程序中编写更多代码以将 `var` 关键字替换为正确类型名称。 返回到 MakeConstCodeFixProvider.cs。 要添加的代码将执行以下步骤：

- 检查声明是否为 `var` 声明，如果它是：
- 创建新类型的推断类型。
- 确保类型声明不是别名。 如果是这样，则声明 `const var` 是合法的。
- 确保 `var` 不是此程序中的类型名称。 （如果是这样，则 `const var` 是合法的）。
- 简化完整类型名称

这听起来好像有很多代码。 其实不然。 将声明和初始化 `newLocal` 的行替换为以下代码。 在初始化 `newModifiers` 之后立即进行：

C#

```
// If the type of the declaration is 'var', create a new type name
// for the inferred type.
VariableDeclarationSyntax variableDeclaration =
localDeclaration.Declaration;
TypeSyntax variableTypeName = variableDeclaration.Type;
if (variableTypeName.IsVar)
{
    SemanticModel semanticModel = await
document.GetSemanticModelAsync(cancellationToken).ConfigureAwait(false);

    // Special case: Ensure that 'var' isn't actually an alias to another
type
    // (e.g. using var = System.String).
```

```
    IAliasSymbol aliasInfo = semanticModel.GetAliasInfo(variableTypeName,
cancellationToken);
    if (aliasInfo == null)
    {
        // Retrieve the type inferred for var.
        ITypeSymbol type = semanticModel.GetTypeInfo(variableTypeName,
cancellationToken).ConvertedType;

        // Special case: Ensure that 'var' isn't actually a type named
'var'.
        if (type.Name != "var")
        {
            // Create a new TypeSyntax for the inferred type. Be careful
            // to keep any leading and trailing trivia from the var keyword.
            TypeSyntax typeName =
SyntaxFactory.ParseTypeName(type.ToDisplayString())
                .WithLeadingTrivia(variableTypeName.GetLeadingTrivia())
                .WithTrailingTrivia(variableTypeName.GetTrailingTrivia());

            // Add an annotation to simplify the type name.
            TypeSyntax simplifiedTypeName =
typeName.WithAdditionalAnnotations(Simplifier.Annotation);

            // Replace the type in the variable declaration.
            variableDeclaration =
variableDeclaration.WithType(simplifiedTypeName);
        }
    }
}
// Produce the new local declaration.
LocalDeclarationStatementSyntax newLocal =
trimmedLocal.WithModifiers(newModifiers)
                            .WithDeclaration(variableDeclaration);
```

需要添加一个 `using` 指令才能使用 Simplifier 类型：

C#

```
using Microsoft.CodeAnalysis.Simplification;
```

运行测试，它们应全部通过。 通过运行已完成的分析器自行庆祝。 按 `Ctrl`+`F5` 在加载了 Roslyn Preview 扩展的第二个 Visual Studio 实例中运行分析器项目。

- 在第二个 Visual Studio 实例，创建一个新的 C# 控制台应用程序项目并将 `int x = "abc";` 添加到 Main 方法。 由于第一个 bug 已修复，应不会报告针对此局部变量声明的警告（尽管像预期那样出现了编译器错误）。
- 接下来，将 `object s = "abc";` 添加到 Main 方法。 由于第二个 bug 已修复，应不会报告任何警告。

- 最后，添加另一个使用 `var` 关键字的局部变量。 你将看到一个警告和显示在左下方的一个建议。
- 将编辑器插入点移到波浪下划线，然后按 `Ctrl`+`.` 显示建议的代码修补程序。 选择代码修补程序，请注意，`var` 关键字现已正确处理。

最后，添加以下代码：

```
C#
```

```
int i = 2;
int j = 32;
int k = i + j;
```

完成这些更改后，仅在前两个变量上有红色波浪线。 将 `const` 同时添加到 `i` 和 `j`，你将获得一个有关 `k` 的新警告，因为它现在可以是 `const`。

祝贺你！ 你已创建第一个 .NET Compiler Platform 扩展来执行即时代码分析，以便检测问题，并提供了用于快速更正的修补程序。 在此过程中，你已了解很多代码 API 是 .NET Compiler Platform SDK (Roslyn API) 的一部分。 可以在我们的示例 GitHub 存储库中根据完成的示例 来检查工作。

其他资源

- 语法分析入门
- 语义分析入门

编程概念 (C#)

项目 • 2024/04/11

此部分介绍了 C# 语言中的编程概念。

本节内容

展开表

Title	描述
.NET 中的程序集	介绍了如何创建和使用程序集。
使用 Async 和 Await 的异步编程 (C#)	介绍了如何在 C# 中使用 async 和 await 关键字编写异步解决方案。 其中包括演练。
特性 (C#)	介绍了如何使用特性提供编程元素（如类型、字段、方法和属性）的附加信息。
集合 (C#)	介绍了 .NET 提供的一些类型集合。 展示了如何使用简单的集合和键/值对集合。
协变和逆变 (C#)	介绍了如何在接口和委托中启用隐式转换泛型类型参数。
表达式树 (C#)	介绍了如何使用表达式树来启用动态修改可执行代码。
迭代器 (C#)	介绍了用于单步执行集合并一次返回一个元素的迭代器。
语言集成查询 (LINQ) (C#)	介绍了 C# 语言语法中强大的查询功能，以及用于查询关系数据库、XML 文档、数据集和内存中集合的模型。
反射 (C#)	介绍了如何使用反射来动态创建类型实例、将类型绑定到现有对象，或从现有对象获取类型并调用其方法或访问其字段和属性。
序列化 (C#)	还介绍了有关二进制、XML 和 SOAP 序列化的关键概念。

相关章节

- 性能提示

  介绍了多项有助于提升应用程序性能的基本规则。

反馈

此页面是否有帮助? 是 否

提供产品反馈

协变和逆变 (C#)

项目 • 2024/04/11

在 C# 中，协变和逆变能够实现数组类型、委托类型和泛型类型参数的隐式引用转换。协变保留分配兼容性，逆变则与之相反。

以下代码演示分配兼容性、协变和逆变之间的差异。

C#

```
// Assignment compatibility.
string str = "test";
// An object of a more derived type is assigned to an object of a less
derived type.
object obj = str;

// Covariance.
IEnumerable<string> strings = new List<string>();
// An object that is instantiated with a more derived type argument
// is assigned to an object instantiated with a less derived type argument.
// Assignment compatibility is preserved.
IEnumerable<object> objects = strings;

// Contravariance.
// Assume that the following method is in the class:
static void SetObject(object o) { }
Action<object> actObject = SetObject;
// An object that is instantiated with a less derived type argument
// is assigned to an object instantiated with a more derived type argument.
// Assignment compatibility is reversed.
Action<string> actString = actObject;
```

数组的协变使派生程度更大的类型的数组能够隐式转换为派生程度更小的类型的数组。但是此操作不是类型安全的操作，如以下代码示例所示。

C#

```
object[] array = new String[10];
// The following statement produces a run-time exception.
// array[0] = 10;
```

对方法组的协变和逆变支持允许将方法签名与委托类型相匹配。 这样，不仅可以将具有匹配签名的方法分配给委托，还可以分配与委托类型指定的派生类型相比，返回派生程度更大的类型的方法（协变）或接受具有派生程度更小的类型的参数的方法（逆变）。 有关详细信息，请参阅委托中的变体 (C#) 和使用委托中的变体 (C#)。

以下代码示例演示对方法组的协变和逆变支持。

C#

```
static object GetObject() { return null; }
static void SetObject(object obj) { }

static string GetString() { return ""; }
static void SetString(string str) { }

static void Test()
{
    // Covariance. A delegate specifies a return type as object,
    // but you can assign a method that returns a string.
    Func<object> del = GetString;

    // Contravariance. A delegate specifies a parameter type as string,
    // but you can assign a method that takes an object.
    Action<string> del2 = SetObject;
}
```

在 .NET Framework 4 或更高版本中，C# 支持在泛型接口和委托中使用协变和逆变，并允许隐式转换泛型类型参数。 有关详细信息，请参阅泛型接口中的变体 (C#) 和委托中的变体 (C#)。

以下代码示例演示泛型接口的隐式引用转换。

C#

```
IEnumerable<String> strings = new List<String>();
IEnumerable<Object> objects = strings;
```

如果泛型接口或委托的泛型参数被声明为协变或逆变，该泛型接口或委托则被称为“变体”。 凭借 C#，能够创建自己的变体接口和委托。 有关详细信息，请参阅创建变体泛型接口 (C#) 和委托中的变体 (C#)。

相关主题

展开表

Title	描述
泛型接口中的变体 (C#)	讨论泛型接口中的协变和逆变，提供 .NET 中的变体泛型接口列表。
创建变体泛型接口 (C#)	演示如何创建自定义变体接口。
在泛型集合的接口中使用变体 (C#)	演示 IEnumerable<T> 接口和 IComparable<T> 接口中对协变和逆变的支持如何帮助重复使用代码。

Title	描述
委托中的变体 (C#)	讨论泛型委托和非泛型委托中的协变和逆变，并提供 .NET 中的变体泛型委托列表。
使用委托中的变体 (C#)	演示如何使用非泛型委托中的协变和逆变支持以将方法签名与委托类型相匹配。
对 Func 和 Action 泛型委托使用变体 (C#)	演示 `Func` 委托和 `Action` 委托中对协变和逆变的支持如何帮助重复使用代码。

反馈

此页面是否有帮助? 是 否

提供产品反馈

泛型接口中的变体 (C#)

项目 • 2023/05/10

.NET Framework 4 引入了对多个现有泛型接口的变体支持。 变体支持允许实现这些接口的类进行隐式转换。

自 .NET Framework 4 起，以下接口为变体：

- IEnumerable<T>（T 是协变）
- IEnumerator<T>（T 是协变）
- IQueryable<T>（T 是协变）
- IGrouping<TKey,TElement>（`TKey` 和 `TElement` 都是协变）
- IComparer<T>（T 是逆变）
- IEqualityComparer<T>（T 是逆变）
- IComparable<T>（T 是逆变）

自 .NET Framework 4.5 起，以下接口是变体：

- IReadOnlyList<T>（T 是协变）
- IReadOnlyCollection<T>（T 是协变）

协变允许方法具有的返回类型比接口的泛型类型参数定义的返回类型的派生程度更大。 若要演示协变功能，请考虑以下泛型接口：`IEnumerable<Object>` 和 `IEnumerable<String>`。 `IEnumerable<String>` 接口不继承 `IEnumerable<Object>` 接口。 但是，`String` 类型会继承 `Object` 类型，在某些情况下，建议为这些接口互相指派彼此的对象。 下面的代码示例对此进行了演示。

```
C#

IEnumerable<String> strings = new List<String>();
IEnumerable<Object> objects = strings;
```

在旧版 .NET Framework 中，此代码会导致 C# 中出现编译错误；如果启用 `Option Strict` 条件，则会导致在 Visual Basic 中出现编译错误。 但现在可使用 `strings` 代替 `objects`，如上例所示，因为 IEnumerable<T> 接口是协变接口。

逆变允许方法具有的实参类型比接口的泛型形参定义的类型的派生程度更小。 若要演示逆变，假设已创建了 `BaseComparer` 类来比较 `BaseClass` 类的实例。 `BaseComparer` 类实现 `IEqualityComparer<BaseClass>` 接口。 因为 IEqualityComparer<T> 接口现在是逆变接口，因此可使用 `BaseComparer` 比较继承 `BaseClass` 类的类的实例。 下面的代码示例对此进行了演示。

C#

```
// Simple hierarchy of classes.
class BaseClass { }
class DerivedClass : BaseClass { }

// Comparer class.
class BaseComparer : IEqualityComparer<BaseClass>
{
    public int GetHashCode(BaseClass baseInstance)
    {
        return baseInstance.GetHashCode();
    }
    public bool Equals(BaseClass x, BaseClass y)
    {
        return x == y;
    }
}
class Program
{
    static void Test()
    {
        IEqualityComparer<BaseClass> baseComparer = new BaseComparer();

        // Implicit conversion of IEqualityComparer<BaseClass> to
        // IEqualityComparer<DerivedClass>.
        IEqualityComparer<DerivedClass> childComparer = baseComparer;
    }
}
```

有关更多示例，请参阅在泛型集合的接口中使用变体 (C#)。

只有引用类型才支持使用泛型接口中的变体。 值类型不支持变体。 例如，无法将 `IEnumerable<int>` 隐式转换为 `IEnumerable<object>`，因为整数由值类型表示。

C#

```
IEnumerable<int> integers = new List<int>();
// The following statement generates a compiler error,
// because int is a value type.
// IEnumerable<Object> objects = integers;
```

还需记住，实现变体接口的类仍是固定类。 例如，虽然 List<T> 实现协变接口 IEnumerable<T>，但不能将 `List<String>` 隐式转换为 `List<Object>`。 以下代码示例阐释了这一点。

C#

```
// The following line generates a compiler error
// because classes are invariant.
// List<Object> list = new List<String>();

// You can use the interface object instead.
IEnumerable<Object> listObjects = new List<String>();
```

请参阅

- 在泛型集合的接口中使用变体 (C#)
- 创建变体泛型接口 (C#)
- 泛型接口
- 委托中的变体 (C#)

创建变体泛型接口 (C#)

项目 • 2023/05/10

接口中的泛型类型参数可以声明为协变或逆变。 协变允许接口方法具有与泛型类型参数定义的返回类型相比，派生程度更大的返回类型。 逆变允许接口方法具有与泛型形参指定的实参类型相比，派生程度更小的实参类型。 具有协变或逆变泛型类型参数的泛型接口称为“变体”。

> **备注**
>
> .NET Framework 4 引入了对多个现有泛型接口的变体支持。 有关 .NET 中变体接口的列表，请参阅**泛型接口中的变体 (C#)**。

声明变体泛型接口

可通过对泛型类型参数使用 `in` 和 `out` 关键字来声明变体泛型接口。

> **重要**
>
> C# 中的 `ref`、`in` 和 `out` 参数不能为变体。 值类型也不支持变体。

可以使用 `out` 关键字将泛型类型参数声明为协变。 协变类型必须满足以下条件：

- 类型仅用作接口方法的返回类型，不用作方法参数的类型。 下例演示了此要求，其中类型 `R` 为声明的协变。

  C#

  ```
  interface ICovariant<out R>
  {
      R GetSomething();
      // The following statement generates a compiler error.
      // void SetSomething(R sampleArg);

  }
  ```

  此规则有一个例外。 如果具有用作方法参数的逆变泛型委托，则可将类型用作该委托的泛型类型参数。 下例中的类型 `R` 演示了此情形。 有关详细信息，请参阅委托中的变体 (C#) 和对 Func 和 Action 泛型委托使用变体 (C#)。

  C#

```
interface ICovariant<out R>
{
    void DoSomething(Action<R> callback);
}
```

- 类型不用作接口方法的泛型约束。 下面的代码阐释了这一点。

```
C#

interface ICovariant<out R>
{
    // The following statement generates a compiler error
    // because you can use only contravariant or invariant types
    // in generic constraints.
    // void DoSomething<T>() where T : R;
}
```

可以使用 `in` 关键字将泛型类型参数声明为逆变。 逆变类型只能用作方法参数的类型，不能用作接口方法的返回类型。 逆变类型还可用于泛型约束。 以下代码演示如何声明逆变接口，以及如何将泛型约束用于其方法之一。

```
C#

interface IContravariant<in A>
{
    void SetSomething(A sampleArg);
    void DoSomething<T>() where T : A;
    // The following statement generates a compiler error.
    // A GetSomething();
}
```

此外，还可以在同一接口中同时支持协变和逆变，但需应用于不同的类型参数，如以下代码示例所示。

```
C#

interface IVariant<out R, in A>
{
    R GetSomething();
    void SetSomething(A sampleArg);
    R GetSetSomethings(A sampleArg);
}
```

实现变体泛型接口

在类中实现变体泛型接口时，所用语法和用于固定接口的语法相同。 以下代码示例演示如何在泛型类中实现协变接口。

C#

```
interface ICovariant<out R>
{
    R GetSomething();
}
class SampleImplementation<R> : ICovariant<R>
{
    public R GetSomething()
    {
        // Some code.
        return default(R);
    }
}
```

实现变体接口的类是固定类。 例如，考虑下面的代码。

C#

```
// The interface is covariant.
ICovariant<Button> ibutton = new SampleImplementation<Button>();
ICovariant<Object> iobj = ibutton;

// The class is invariant.
SampleImplementation<Button> button = new SampleImplementation<Button>();
// The following statement generates a compiler error
// because classes are invariant.
// SampleImplementation<Object> obj = button;
```

扩展变体泛型接口

扩展变体泛型接口时，必须使用 `in` 和 `out` 关键字来显式指定派生接口是否支持变体。 编译器不会根据正在扩展的接口来推断变体。 例如，考虑以下接口。

C#

```
interface ICovariant<out T> { }
interface IInvariant<T> : ICovariant<T> { }
interface IExtCovariant<out T> : ICovariant<T> { }
```

尽管 `IInvariant<T>` 接口和 `IExtCovariant<out T>` 接口扩展的是同一个接口，但泛型类型参数 `T` 在前者中为固定参数，在后者中为协变参数。 此规则也适用于逆变泛型类型参数。

无论泛型类型参数 `T` 在接口中是协变还是逆变，都可以创建一个接口来扩展这两类接口，只要在扩展接口中，该 `T` 泛型类型参数为固定参数。 以下代码示例阐释了这一点。

C#

```
interface ICovariant<out T> { }
interface IContravariant<in T> { }
interface IInvariant<T> : ICovariant<T>, IContravariant<T> { }
```

但是，如果泛型类型参数 `T` 在一个接口中声明为协变，则无法在扩展接口中将其声明为逆变，反之亦然。 以下代码示例阐释了这一点。

C#

```
interface ICovariant<out T> { }
// The following statement generates a compiler error.
// interface ICoContraVariant<in T> : ICovariant<T> { }
```

避免多义性

实现变体泛型接口时，变体有时可能会导致多义性。 应避免这样的多义性。

例如，如果在一个类中使用不同的泛型类型参数来显式实现同一变体泛型接口，便会产生多义性。 在这种情况下，编译器不会产生错误，但未指定将在运行时选择哪个接口实现。 这种多义性可能导致代码中出现小 bug。 请考虑以下代码示例。

C#

```
// Simple class hierarchy.
class Animal { }
class Cat : Animal { }
class Dog : Animal { }

// This class introduces ambiguity
// because IEnumerable<out T> is covariant.
class Pets : IEnumerable<Cat>, IEnumerable<Dog>
{
    IEnumerator<Cat> IEnumerable<Cat>.GetEnumerator()
    {
        Console.WriteLine("Cat");
        // Some code.
        return null;
    }

    IEnumerator IEnumerable.GetEnumerator()
    {
        // Some code.
        return null;
```

```
    }

    IEnumerator<Dog> IEnumerable<Dog>.GetEnumerator()
    {
        Console.WriteLine("Dog");
        // Some code.
        return null;
    }
}
class Program
{
    public static void Test()
    {
        IEnumerable<Animal> pets = new Pets();
        pets.GetEnumerator();
    }
}
```

在此示例中，没有指定 `pets.GetEnumerator` 方法如何在 `Cat` 和 `Dog` 之间选择。 这可能导致代码中出现问题。

请参阅

- 泛型接口中的变体 (C#)
- 对 Func 和 Action 泛型委托使用变体 (C#)

在泛型集合的接口中使用变体 (C#)

项目 • 2023/05/10

协变接口允许其方法返回的派生类型多于接口中指定的派生类型。 逆变接口允许其方法接受派生类型少于接口中指定的类型的参数。

在.NET Framework 4 中，多个现有接口已变为协变和逆变接口。 包括 IEnumerable<T> 和 IComparable<T>。 这使你可将对基类型的泛型集合进行操作的那些方法重用于派生类型的集合。

有关 .NET 中变体接口的列表，请参阅泛型接口中的变体 (C#)。

转换泛型集合

下例阐释了 IEnumerable<T> 接口中的协变支持的益处。 `PrintFullName` 方法接受 `IEnumerable<Person>` 类型的集合作为参数。 但可将该方法重用于 `IEnumerable<Employee>` 类型的集合，因为 `Employee` 继承 `Person`。

C#

```
// Simple hierarchy of classes.
public class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
}

public class Employee : Person { }

class Program
{
    // The method has a parameter of the IEnumerable<Person> type.
    public static void PrintFullName(IEnumerable<Person> persons)
    {
        foreach (Person person in persons)
        {
            Console.WriteLine("Name: {0} {1}",
            person.FirstName, person.LastName);
        }
    }

    public static void Test()
    {
        IEnumerable<Employee> employees = new List<Employee>();

        // You can pass IEnumerable<Employee>,
        // although the method expects IEnumerable<Person>.
```

```
        PrintFullName(employees);

    }
}
```

比较泛型集合

下例阐释了 IEqualityComparer<T> 接口中的逆变支持的益处。 `PersonComparer` 类实现 `IEqualityComparer<Person>` 接口。 但可以重用此类来比较 `Employee` 类型的对象序列，因为 `Employee` 继承 `Person`。

C#

```
// Simple hierarchy of classes.
public class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
}

public class Employee : Person { }

// The custom comparer for the Person type
// with standard implementations of Equals()
// and GetHashCode() methods.
class PersonComparer : IEqualityComparer<Person>
{
    public bool Equals(Person x, Person y)
    {
        if (Object.ReferenceEquals(x, y)) return true;
        if (Object.ReferenceEquals(x, null) ||
            Object.ReferenceEquals(y, null))
            return false;
        return x.FirstName == y.FirstName && x.LastName == y.LastName;
    }
    public int GetHashCode(Person person)
    {
        if (Object.ReferenceEquals(person, null)) return 0;
        int hashFirstName = person.FirstName == null
            ? 0 : person.FirstName.GetHashCode();
        int hashLastName = person.LastName.GetHashCode();
        return hashFirstName ^ hashLastName;
    }
}

class Program
{

    public static void Test()
    {
```

```
        List<Employee> employees = new List<Employee> {
               new Employee() {FirstName = "Michael", LastName =
"Alexander"},
               new Employee() {FirstName = "Jeff", LastName = "Price"}
            };

        // You can pass PersonComparer,
        // which implements IEqualityComparer<Person>,
        // although the method expects IEqualityComparer<Employee>.

        IEnumerable<Employee> noduplicates =
            employees.Distinct<Employee>(new PersonComparer());

        foreach (var employee in noduplicates)
            Console.WriteLine(employee.FirstName + " " + employee.LastName);
    }
}
```

请参阅

- 泛型接口中的变体 (C#)

委托中的变体 (C#)

项目 • 2023/05/09

.NET Framework 3.5 引入了变体支持，用于在 C# 中匹配所有委托的方法签名和委托类型。 这表明不仅可以将具有匹配签名的方法分配给委托，还可以将返回派生程度较大的派生类型的方法分配给委托（协变），或者如果方法所接受参数的派生类型所具有的派生程度小于委托类型指定的程度（逆变），也可将其分配给委托。 这包括泛型委托和非泛型委托。

例如，思考以下代码，该代码具有两个类和两个委托：泛型和非泛型。

C#

```
public class First { }
public class Second : First { }
public delegate First SampleDelegate(Second a);
public delegate R SampleGenericDelegate<A, R>(A a);
```

创建 `SampleDelegate` 或 `SampleGenericDelegate<A, R>` 类型的委托时，可以将以下任一方法分配给这些委托。

C#

```
// Matching signature.
public static First ASecondRFirst(Second second)
{ return new First(); }

// The return type is more derived.
public static Second ASecondRSecond(Second second)
{ return new Second(); }

// The argument type is less derived.
public static First AFirstRFirst(First first)
{ return new First(); }

// The return type is more derived
// and the argument type is less derived.
public static Second AFirstRSecond(First first)
{ return new Second(); }
```

以下代码示例说明了方法签名与委托类型之间的隐式转换。

C#

```
// Assigning a method with a matching signature
// to a non-generic delegate. No conversion is necessary.
```

```
SampleDelegate dNonGeneric = ASecondRFirst;
// Assigning a method with a more derived return type
// and less derived argument type to a non-generic delegate.
// The implicit conversion is used.
SampleDelegate dNonGenericConversion = AFirstRSecond;

// Assigning a method with a matching signature to a generic delegate.
// No conversion is necessary.
SampleGenericDelegate<Second, First> dGeneric = ASecondRFirst;
// Assigning a method with a more derived return type
// and less derived argument type to a generic delegate.
// The implicit conversion is used.
SampleGenericDelegate<Second, First> dGenericConversion = AFirstRSecond;
```

有关更多示例，请参阅在委托中使用变体 (C#) 和对 Func 和 Action 泛型委托使用变体 (C#)。

泛型类型参数中的变体

在 .NET Framework 4 或更高版本中，可以启用委托之间的隐式转换，以便在具有泛型类型参数所指定的不同类型按变体的要求继承自对方时，可以将这些类型的泛型委托分配给对方。

若要启用隐式转换，必须使用 `in` 或 `out` 关键字将委托中的泛型参数显式声明为协变或逆变。

以下代码示例演示了如何创建一个具有协变泛型类型参数的委托。

C#

```
// Type T is declared covariant by using the out keyword.
public delegate T SampleGenericDelegate <out T>();

public static void Test()
{
    SampleGenericDelegate <String> dString = () => " ";

    // You can assign delegates to each other,
    // because the type T is declared covariant.
    SampleGenericDelegate <Object> dObject = dString;
}
```

如果仅使用变体支持来匹配方法签名和委托类型，且不使用 `in` 和 `out` 关键字，则可能会发现有时可以使用相同的 lambda 表达式或方法实例化委托，但不能将一个委托分配给另一个委托。

在以下代码示例中，`SampleGenericDelegate<String>` 不能显式转换为 `SampleGenericDelegate<Object>`，尽管 `String` 继承 `Object`。 可以使用 `out` 关键字标记泛型参数 `T` 解决此问题。

```
C#
```

```
public delegate T SampleGenericDelegate<T>();

public static void Test()
{
    SampleGenericDelegate<String> dString = () => " ";

    // You can assign the dObject delegate
    // to the same lambda expression as dString delegate
    // because of the variance support for
    // matching method signatures with delegate types.
    SampleGenericDelegate<Object> dObject = () => " ";

    // The following statement generates a compiler error
    // because the generic type T is not marked as covariant.
    // SampleGenericDelegate <Object> dObject = dString;

}
```

.NET 中具有变体类型参数的泛型委托

.NET Framework 4 在几个现有泛型委托中引入了泛型类型参数的变体支持：

- System 命名空间的 `Action` 委托，例如 Action<T> 和 Action<T1,T2>
- System 命名空间的 `Func` 委托，例如 Func<TResult> 和 Func<T,TResult>
- Predicate<T> 委托
- Comparison<T> 委托
- Converter<TInput,TOutput> 委托

有关详细信息和示例，请参阅对 Func 和 Action 泛型委托使用变体 (C#)。

声明泛型委托中的变体类型参数

如果泛型委托具有协变或逆变泛型类型参数，则该委托可被称为“变体泛型委托”。

可以使用 `out` 关键字声明泛型委托中的泛型类型参数协变。 协变类型只能用作方法返回类型，而不能用作方法参数的类型。 以下代码示例演示了如何声明协变泛型委托。

C#

```
public delegate R DCovariant<out R>();
```

可以使用 `in` 关键字声明泛型委托中的泛型类型参数逆变。 逆变类型只能用作方法参数的类型，而不能用作方法返回类型。 以下代码示例演示了如何声明逆变泛型委托。

C#

```
public delegate void DContravariant<in A>(A a);
```

> ⓘ **重要**
>
> C# 中的 `ref`、 `in` 和 `out` 参数不能标记为变体。

可以在同一个委托中支持变体和协变，但这只适用于不同类型的参数。 这在下面的示例中显示。

C#

```
public delegate R DVariant<in A, out R>(A a);
```

实例化和调用变体泛型委托

可以像实例化和调用固定委托一样，实例化和调用变体委托。 在以下示例中，该委托通过 lambda 表达式进行实例化。

C#

```
DVariant<String, String> dvariant = (String str) => str + " ";
dvariant("test");
```

合并变体泛型委托

请勿合并变体委托。 Combine 方法不支持变体委托转换，并且要求委托的类型完全相同。 这会导致在使用 Combine 方法或使用 `+` 运算符合并委托时出现运行时异常，如以下代码示例中所示。

C#

```
Action<object> actObj = x => Console.WriteLine("object: {0}", x);
Action<string> actStr = x => Console.WriteLine("string: {0}", x);
// All of the following statements throw exceptions at run time.
// Action<string> actCombine = actStr + actObj;
// actStr += actObj;
// Delegate.Combine(actStr, actObj);
```

泛型类型参数中用于值和引用类型的变体

泛型类型参数的变体仅支持引用类型。 例如，`DVariant<int>` 不能隐式转换为 `DVariant<Object>` 或 `DVariant<long>`，因为整数是值类型。

以下示例演示了泛型类型参数中的变体不支持值类型。

C#

```
// The type T is covariant.
public delegate T DVariant<out T>();

// The type T is invariant.
public delegate T DInvariant<T>();

public static void Test()
{
    int i = 0;
    DInvariant<int> dInt = () => i;
    DVariant<int> dVariantInt = () => i;

    // All of the following statements generate a compiler error
    // because type variance in generic parameters is not supported
    // for value types, even if generic type parameters are declared
variant.
    // DInvariant<Object> dObject = dInt;
    // DInvariant<long> dLong = dInt;
    // DVariant<Object> dVariantObject = dVariantInt;
    // DVariant<long> dVariantLong = dVariantInt;
}
```

请参阅

- 泛型
- 对 Func 和 Action 泛型委托使用变体 (C#)
- 如何合并委托（多播委托）

使用委托中的变体 (C#)

项目 • 2023/04/07

向委托分配方法时，协变和逆变为匹配委托类型和方法签名提供了灵活性。 协变允许方法具有的派生返回类型多于委托中定义的类型。 逆变允许方法具有的派生参数类型少于委托类型中的类型。

示例 1：协变

描述

本示例演示如何将委托与具有返回类型的方法一起使用，这些返回类型派生自委托签名中的返回类型。 `DogsHandler` 返回的数据类型属于 `Dogs` 类型，它派生自委托中定义的 `Mammals` 类型。

代码

C#

```
class Mammals {}
class Dogs : Mammals {}

class Program
{
    // Define the delegate.
    public delegate Mammals HandlerMethod();

    public static Mammals MammalsHandler()
    {
        return null;
    }

    public static Dogs DogsHandler()
    {
        return null;
    }

    static void Test()
    {
        HandlerMethod handlerMammals = MammalsHandler;

        // Covariance enables this assignment.
        HandlerMethod handlerDogs = DogsHandler;
    }
}
```

示例 2：逆变

描述

本示例演示如何将委托与具有参数的方法一起使用，这些参数的类型是委托签名参数类型的基类型。 通过逆变可以使用一个事件处理程序而不是多个单独的处理程序。 下面的示例使用两个委托：

- 定义 KeyEventHandler 事件签名的 KeyEventHandler 委托。 其签名为：

  C#

  ```
  public delegate void KeyEventHandler(object sender, KeyEventArgs e)
  ```

- 定义 MouseEventHandler 事件签名的 MouseEventHandler 委托。 其签名为：

  C#

  ```
  public delegate void MouseEventHandler(object sender, MouseEventArgs e)
  ```

该示例定义了一个具有 EventArgs 参数的事件处理程序，并使用它来处理 `Button.KeyDown` 和 `Button.MouseClick` 事件。 它可以这样做是因为 EventArgs 是 KeyEventArgs 和 MouseEventArgs 的基类型。

代码

C#

```
// Event handler that accepts a parameter of the EventArgs type.
private void MultiHandler(object sender, System.EventArgs e)
{
    label1.Text = System.DateTime.Now.ToString();
}

public Form1()
{
    InitializeComponent();

    // You can use a method that has an EventArgs parameter,
    // although the event expects the KeyEventArgs parameter.
    this.button1.KeyDown += this.MultiHandler;

    // You can use the same method
    // for an event that expects the MouseEventArgs parameter.
    this.button1.MouseClick += this.MultiHandler;
```

```
}
```

请参阅

- 委托中的变体 (C#)
- 对 Func 和 Action 泛型委托使用变体 (C#)

对 Func 和 Action 泛型委托使用变体 (C#)

项目 • 2023/04/07

这些示例演示如何使用 `Func` 和 `Action` 泛型委托中的协变和逆变来启用重用方法并为代码中提供更多的灵活性。

有关协变和逆变的详细信息，请参阅委托中的变体 (C#)。

使用具有协变类型参数的委托

下例阐释了泛型 `Func` 委托中的协变支持的益处。 `FindByTitle` 方法采用 `String` 类型的一个参数，并返回 `Employee` 类型的一个对象。 但是，可将此方法分配给 `Func<String, Person>` 委托，因为 `Employee` 继承 `Person`。

C#

```
// Simple hierarchy of classes.
public class Person { }
public class Employee : Person { }
class Program
{
    static Employee FindByTitle(String title)
    {
        // This is a stub for a method that returns
        // an employee that has the specified title.
        return new Employee();
    }

    static void Test()
    {
        // Create an instance of the delegate without using variance.
        Func<String, Employee> findEmployee = FindByTitle;

        // The delegate expects a method to return Person,
        // but you can assign it a method that returns Employee.
        Func<String, Person> findPerson = FindByTitle;

        // You can also assign a delegate
        // that returns a more derived type
        // to a delegate that returns a less derived type.
        findPerson = findEmployee;

    }
}
```

使用具有逆变类型参数的委托

下例阐释了泛型 `Action` 委托中的逆变支持的益处。 `AddToContacts` 方法采用 `Person` 类型的一个参数。 但是，可将此方法分配给 `Action<Employee>` 委托，因为 `Employee` 继承 `Person`。

C#

```
public class Person { }
public class Employee : Person { }
class Program
{
    static void AddToContacts(Person person)
    {
        // This method adds a Person object
        // to a contact list.
    }

    static void Test()
    {
        // Create an instance of the delegate without using variance.
        Action<Person> addPersonToContacts = AddToContacts;

        // The Action delegate expects
        // a method that has an Employee parameter,
        // but you can assign it a method that has a Person parameter
        // because Employee derives from Person.
        Action<Employee> addEmployeeToContacts = AddToContacts;

        // You can also assign a delegate
        // that accepts a less derived parameter to a delegate
        // that accepts a more derived parameter.
        addEmployeeToContacts = addPersonToContacts;
    }
}
```

请参阅

- 协变和逆变 (C#)
- 泛型

迭代器 (C#)

项目 • 2023/05/09

迭代器可用于逐步迭代集合，例如列表和数组。

迭代器方法或 `get` 访问器可对集合执行自定义迭代。 迭代器方法使用 yield return 语句返回元素，每次返回一个。 到达 `yield return` 语句时，会记住当前在代码中的位置。 下次调用迭代器函数时，将从该位置重新开始执行。

通过 foreach 语句或 LINQ 查询从客户端代码中使用迭代器。

在以下示例中，`foreach` 循环的首次迭代导致 `SomeNumbers` 迭代器方法继续执行，直至到达第一个 `yield return` 语句。 此迭代返回的值为 3，并保留当前在迭代器方法中的位置。 在循环的下次迭代中，迭代器方法的执行将从其暂停的位置继续，直至到达 `yield return` 语句后才会停止。 此迭代返回的值为 5，并再次保留当前在迭代器方法中的位置。 到达迭代器方法的结尾时，循环便已完成。

C#

```
static void Main()
{
    foreach (int number in SomeNumbers())
    {
        Console.Write(number.ToString() + " ");
    }
    // Output: 3 5 8
    Console.ReadKey();
}

public static System.Collections.IEnumerable SomeNumbers()
{
    yield return 3;
    yield return 5;
    yield return 8;
}
```

迭代器方法或 `get` 访问器的返回类型可以是 IEnumerable、IEnumerable<T>、IEnumerator 或 IEnumerator<T>。

可以使用 `yield break` 语句来终止迭代。

> **① 备注**
>
> 对于本主题中除简单迭代器示例以外的所有示例，请为 `System.Collections` 和 `System.Collections.Generic` 命名空间加入 using 指令。

简单的迭代器

下例包含一个位于 for 循环内的 `yield return` 语句。 在 `Main` 中，`foreach` 语句体的每次迭代都会创建一个对迭代器函数的调用，并将继续到下一个 `yield return` 语句。

C#

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

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

创建集合类

在以下示例中，`DaysOfTheWeek` 类实现 IEnumerable 接口，此操作需要 GetEnumerator 方法。 编译器隐式调用 `GetEnumerator` 方法，此方法返回 IEnumerator。

`GetEnumerator` 方法通过使用 `yield return` 语句每次返回 1 个字符串。

C#

```
static void Main()
{
    DaysOfTheWeek days = new DaysOfTheWeek();

    foreach (string day in days)
    {
        Console.Write(day + " ");
```

```
    }
    // Output: Sun Mon Tue Wed Thu Fri Sat
    Console.ReadKey();
}

public class DaysOfTheWeek : IEnumerable
{
    private string[] days = { "Sun", "Mon", "Tue", "Wed", "Thu", "Fri",
"Sat" };

    public IEnumerator GetEnumerator()
    {
        for (int index = 0; index < days.Length; index++)
        {
            // Yield each day of the week.
            yield return days[index];
        }
    }
}
```

下例创建了一个包含动物集合的 `Zoo` 类。

引用类实例 (`theZoo`) 的 `foreach` 语句隐式调用 `GetEnumerator` 方法。 引用 `Birds` 和 `Mammals` 属性的 `foreach` 语句使用 `AnimalsForType` 命名迭代器方法。

C#

```
static void Main()
{
    Zoo theZoo = new Zoo();

    theZoo.AddMammal("Whale");
    theZoo.AddMammal("Rhinoceros");
    theZoo.AddBird("Penguin");
    theZoo.AddBird("Warbler");

    foreach (string name in theZoo)
    {
        Console.Write(name + " ");
    }
    Console.WriteLine();
    // Output: Whale Rhinoceros Penguin Warbler

    foreach (string name in theZoo.Birds)
    {
        Console.Write(name + " ");
    }
    Console.WriteLine();
    // Output: Penguin Warbler

    foreach (string name in theZoo.Mammals)
    {
```

```
            Console.Write(name + " ");
        }
        Console.WriteLine();
        // Output: Whale Rhinoceros

        Console.ReadKey();
    }

    public class Zoo : IEnumerable
    {
        // Private members.
        private List<Animal> animals = new List<Animal>();

        // Public methods.
        public void AddMammal(string name)
        {
            animals.Add(new Animal { Name = name, Type = Animal.TypeEnum.Mammal
    });
        }

        public void AddBird(string name)
        {
            animals.Add(new Animal { Name = name, Type = Animal.TypeEnum.Bird
    });
        }

        public IEnumerator GetEnumerator()
        {
            foreach (Animal theAnimal in animals)
            {
                yield return theAnimal.Name;
            }
        }

        // Public members.
        public IEnumerable Mammals
        {
            get { return AnimalsForType(Animal.TypeEnum.Mammal); }
        }

        public IEnumerable Birds
        {
            get { return AnimalsForType(Animal.TypeEnum.Bird); }
        }

        // Private methods.
        private IEnumerable AnimalsForType(Animal.TypeEnum type)
        {
            foreach (Animal theAnimal in animals)
            {
                if (theAnimal.Type == type)
                {
                    yield return theAnimal.Name;
                }
            }
```

```
    }

    // Private class.
    private class Animal
    {
        public enum TypeEnum { Bird, Mammal }

        public string Name { get; set; }
        public TypeEnum Type { get; set; }
    }
}
```

对泛型列表使用迭代器

在以下示例中，Stack<T> 泛型类实现 IEnumerable<T> 泛型接口。 Push 方法将值分配给类型为 `T` 的数组。 GetEnumerator 方法通过使用 `yield return` 语句返回数组值。

除了泛型 GetEnumerator 方法，还必须实现非泛型 GetEnumerator 方法。 这是因为从 IEnumerable 继承了 IEnumerable<T>。 非泛型实现遵从泛型实现的规则。

本示例使用命名迭代器来支持通过各种方法循环访问同一数据集合。 这些命名迭代器为 `TopToBottom` 和 `BottomToTop` 属性，以及 `TopN` 方法。

`BottomToTop` 属性在 `get` 访问器中使用迭代器。

C#

```
static void Main()
{
    Stack<int> theStack = new Stack<int>();

    //  Add items to the stack.
    for (int number = 0; number <= 9; number++)
    {
        theStack.Push(number);
    }

    // Retrieve items from the stack.
    // foreach is allowed because theStack implements IEnumerable<int>.
    foreach (int number in theStack)
    {
        Console.Write("{0} ", number);
    }
    Console.WriteLine();
    // Output: 9 8 7 6 5 4 3 2 1 0

    // foreach is allowed, because theStack.TopToBottom returns
IEnumerable(Of Integer).
    foreach (int number in theStack.TopToBottom)
    {
```

```
        Console.Write("{0} ", number);
    }
    Console.WriteLine();
    // Output: 9 8 7 6 5 4 3 2 1 0

    foreach (int number in theStack.BottomToTop)
    {
        Console.Write("{0} ", number);
    }
    Console.WriteLine();
    // Output: 0 1 2 3 4 5 6 7 8 9

    foreach (int number in theStack.TopN(7))
    {
        Console.Write("{0} ", number);
    }
    Console.WriteLine();
    // Output: 9 8 7 6 5 4 3

    Console.ReadKey();
}

public class Stack<T> : IEnumerable<T>
{
    private T[] values = new T[100];
    private int top = 0;

    public void Push(T t)
    {
        values[top] = t;
        top++;
    }
    public T Pop()
    {
        top--;
        return values[top];
    }

    // This method implements the GetEnumerator method. It allows
    // an instance of the class to be used in a foreach statement.
    public IEnumerator<T> GetEnumerator()
    {
        for (int index = top - 1; index >= 0; index--)
        {
            yield return values[index];
        }
    }

    IEnumerator IEnumerable.GetEnumerator()
    {
        return GetEnumerator();
    }

    public IEnumerable<T> TopToBottom
    {
```

```
        get { return this; }
    }

    public IEnumerable<T> BottomToTop
    {
        get
        {
            for (int index = 0; index <= top - 1; index++)
            {
                yield return values[index];
            }
        }
    }

    public IEnumerable<T> TopN(int itemsFromTop)
    {
        // Return less than itemsFromTop if necessary.
        int startIndex = itemsFromTop >= top ? 0 : top - itemsFromTop;

        for (int index = top - 1; index >= startIndex; index--)
        {
            yield return values[index];
        }
    }

}
```

语法信息

迭代器可用作一种方法，或一个 `get` 访问器。 不能在事件、实例构造函数、静态构造函数或静态终结器中使用迭代器。

必须存在从 `yield return` 语句中的表达式类型到迭代器返回的 `IEnumerable<T>` 类型参数的隐式转换。

在 C# 中，迭代器方法不能有任何 `in`、`ref` 或 `out` 参数。

在 C# 中，`yield` 不是保留字，只有在 `return` 或 `break` 关键字之前使用时才有特殊含义。

技术实现

即使将迭代器编写成方法，编译器也会将其转换为实际上是状态机的嵌套类。 只要客户端代码中的 `foreach` 循环继续，此类就会跟踪迭代器的位置。

若要查看编译器执行的操作，可使用 Ildasm.exe 工具查看为迭代器方法生成的 Microsoft 中间语言代码。

为类或结构创建迭代器时，不必实现整个 IEnumerator 接口。 编译器检测到迭代器时，会自动生成 IEnumerator 或 IEnumerator<T> 接口的 `Current`、`MoveNext` 和 `Dispose` 方法。

在 `foreach` 循环（或对 `IEnumerator.MoveNext` 的直接调用）的每次后续迭代中，下一个迭代器代码体都会在上一个 `yield return` 语句之后恢复。 然后继续下一个 `yield return` 语句，直至到达迭代器体的结尾，或直至遇到 `yield break` 语句。

迭代器不支持 IEnumerator.Reset 方法。 若要从头开始重新迭代，必须获取新的迭代器。 在迭代器方法返回的迭代器上调用 Reset 会引发 NotSupportedException。

有关其他信息，请参阅 C# 语言规范。

迭代器的使用

需要使用复杂代码填充列表序列时，使用迭代器可保持 `foreach` 循环的简单性。 需执行以下操作时，这可能很有用：

- 在第一次 `foreach` 循环迭代之后，修改列表序列。
- 避免在 `foreach` 循环的第一次迭代之前完全加载大型列表。 一个示例是用于加载一批表格行的分页提取。 另一个示例是 EnumerateFiles 方法，该方法在 .NET 中实现迭代器。
- 在迭代器中封装生成列表。 使用迭代器方法，可生成该列表，然后在循环中产出每个结果。

请参阅

- System.Collections.Generic
- IEnumerable<T>
- foreach, in
- 对数组使用 foreach
- 泛型

语句（C# 编程指南）

项目 • 2024/03/13

程序执行的操作采用语句表达。 常见操作包括声明变量、赋值、调用方法、循环访问集合，以及根据给定条件分支到一个或另一个代码块。 语句在程序中的执行顺序称为“控制流”或“执行流”。 根据程序对运行时所收到的输入的响应，在程序每次运行时控制流可能有所不同。

语句可以是以分号结尾的单行代码，也可以是语句块中的一系列单行语句。 语句块括在括号 {} 中，并且可以包含嵌套块。 以下代码演示了两个单行语句示例和一个多行语句块：

C#

```
    public static void Main()
    {
        // Declaration statement.
        int counter;

        // Assignment statement.
        counter = 1;

        // Error! This is an expression, not an expression statement.
        // counter + 1;

        // Declaration statements with initializers are functionally
        // equivalent to  declaration statement followed by assignment
statement:
        int[] radii = [15, 32, 108, 74, 9]; // Declare and initialize an
array.
        const double pi = 3.14159; // Declare and initialize  constant.

        // foreach statement block that contains multiple statements.
        foreach (int radius in radii)
        {
            // Declaration statement with initializer.
            double circumference = pi * (2 * radius);

            // Expression statement (method invocation). A single-line
            // statement can span multiple text lines because line breaks
            // are treated as white space, which is ignored by the compiler.
            System.Console.WriteLine("Radius of circle #{0} is {1}.
Circumference = {2:N2}",
                                     counter, radius, circumference);

            // Expression statement (postfix increment).
            counter++;
        } // End of foreach statement block
    } // End of Main method body.
} // End of SimpleStatements class.
```

```
/*
   Output:
    Radius of circle #1 = 15. Circumference = 94.25
    Radius of circle #2 = 32. Circumference = 201.06
    Radius of circle #3 = 108. Circumference = 678.58
    Radius of circle #4 = 74. Circumference = 464.96
    Radius of circle #5 = 9. Circumference = 56.55
*/
```

语句的类型

下表列出了 C# 中的各种语句类型及其关联的关键字，并提供指向包含详细信息的主题的链接：

展开表

类别	C# 关键字/说明
声明语句	声明语句引入新的变量或常量。 变量声明可以选择为变量赋值。 在常量声明中必须赋值。
表达式语句	用于计算值的表达式语句必须在变量中存储该值。
选择语句	选择语句用于根据一个或多个指定条件分支到不同的代码段。 有关详情，请参阅以下主题： • if • switch
迭代语句	迭代语句用于遍历集合（如数组），或重复执行同一组语句直到满足指定的条件。 有关详情，请参阅以下主题： • do • for • foreach • while
跳转语句	跳转语句将控制转移给另一代码段。 有关详情，请参阅以下主题： • break • continue • goto • return • yield
异常处理语句	异常处理语句用于从运行时发生的异常情况正常恢复。 有关详情，请参阅以下主题： • throw • try-catch

类别	C# 关键字/说明
	• try-finally • try-catch-finally
checked 和 unchecked	`checked` 和 `unchecked` 语句用于指定将结果存储在变量中、但该变量过小而不能容纳结果值时，是否允许整型数值运算导致溢出。
`await` 语句	如果用 async 修饰符标记方法，则可以使用该方法中的 await 运算符。 在控制到达异步方法的 `await` 表达式时，控制将返回到调用方，该方法中的进程将挂起，直到等待的任务完成为止。 任务完成后，可以在方法中恢复执行。 有关简单示例，请参阅方法的“异步方法”一节。 有关详细信息，请参阅 async 和 await 的异步编程。
`yield return` 语句	迭代器对集合执行自定义迭代，如列表或数组。 迭代器使用 yield return 语句返回元素，每次返回一个。 到达 `yield return` 语句时，会记住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。 有关更多信息，请参见 迭代器。
`fixed` 语句	fixed 语句禁止垃圾回收器重定位可移动的变量。 有关详细信息，请参阅 fixed。
`lock` 语句	lock 语句用于限制一次仅允许一个线程访问代码块。 有关详细信息，请参阅 lock。
带标签的语句	可以为语句指定一个标签，然后使用 goto 关键字跳转到该带标签的语句。 （参见下一行中的示例。）
空语句	空语句只含一个分号。 不执行任何操作，可以在需要语句但不需要执行任何操作的地方使用。

声明语句

以下代码显示了具有和不具有初始赋值的变量声明的示例，以及具有必要初始化的常量声明。

C#

```
// Variable declaration statements.
double area;
double radius = 2;

// Constant declaration statement.
const double pi = 3.14159;
```

表达式语句

以下代码显示了表达式语句的示例，包括赋值、使用赋值创建对象和方法调用。

C#

```
// Expression statement (assignment).
area = 3.14 * (radius * radius);

// Error. Not  statement because no assignment:
//circ * 2;

// Expression statement (method invocation).
System.Console.WriteLine();

// Expression statement (new object creation).
System.Collections.Generic.List<string> strings =
    new System.Collections.Generic.List<string>();
```

空语句

以下示例演示了空语句的两种用法：

C#

```
void ProcessMessages()
{
    while (ProcessMessage())
        ; // Statement needed here.
}

void F()
{
    //...
    if (done) goto exit;
//...
exit:
    ; // Statement needed here.
}
```

嵌入式语句

某些语句（如迭代语句）后面始终跟有一条嵌入式语句。 此嵌入式语句可以是单个语句，也可以是语句块中括在括号 {} 内的多个语句。 甚至可以在括号 {} 内包含单行嵌入式语句，如以下示例所示：

C#

```
// Recommended style. Embedded statement in  block.
foreach (string s in System.IO.Directory.GetDirectories(
                        System.Environment.CurrentDirectory))
{
    System.Console.WriteLine(s);
}

// Not recommended.
foreach (string s in System.IO.Directory.GetDirectories(
                        System.Environment.CurrentDirectory))
    System.Console.WriteLine(s);
```

未括在括号 {} 内的嵌入式语句不能作为声明语句或带标签的语句。 下面的示例对此进行了演示：

C#

```
if(pointB == true)
    //Error CS1023:
    int radius = 5;
```

将该嵌入式语句放在语句块中以修复错误：

C#

```
if (b == true)
{
    // OK:
    System.DateTime d = System.DateTime.Now;
    System.Console.WriteLine(d.ToLongDateString());
}
```

嵌套语句块

语句块可以嵌套，如以下代码所示：

C#

```
foreach (string s in System.IO.Directory.GetDirectories(
    System.Environment.CurrentDirectory))
{
    if (s.StartsWith("CSharp"))
    {
        if (s.EndsWith("TempFolder"))
        {
            return s;
```

```
        }
    }
}
return "Not found.";
```

无法访问的语句

如果编译器认为在任何情况下控制流都无法到达特定语句，将生成警告 CS0162，如下例所示：

C#

```
// An over-simplified example of unreachable code.
const int val = 5;
if (val < 4)
{
    System.Console.WriteLine("I'll never write anything."); //CS0162
}
```

C# 语言规范

有关详细信息，请参阅 C# 语言规范中的语句部分。

另请参阅

- 语句关键字
- 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

Expression-bodied 成员（C# 编程指南）

项目 • 2023/08/17

通过表达式主体定义，可通过简洁可读的形式提供成员的实现。 只要任何支持的成员（如方法或属性）的逻辑包含单个表达式，就可以使用表达式主体定义。 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。

表达式主体定义可用于以下类型成员：

- 方法
- 只读属性
- 属性
- 构造函数
- 终结器
- 索引器

方法

expression-bodied 方法包含单个表达式，它返回的值的类型与方法的返回类型匹配；或者，对于返回 `void` 的方法，其表达式则执行某些操作。 例如，替代 ToString 方法的类型通常包含单个表达式，该表达式返回当前对象的字符串表示形式。

下面的示例定义 `Person` 类，该类通过表达式主体定义替代 ToString。 它还定义向控制台显示名称的 `DisplayName` 方法。 `ToString` 表达式主体定义中未使用 `return` 关键字。

```
C#

using System;

namespace ExpressionBodiedMembers;

public class Person
{
   public Person(string firstName, string lastName)
   {
      fname = firstName;
      lname = lastName;
   }
```

```
   private string fname;
   private string lname;

   public override string ToString() => $"{fname} {lname}".Trim();
   public void DisplayName() => Console.WriteLine(ToString());
}

class Example
{
   static void Main()
   {
      Person p = new Person("Mandy", "Dejesus");
      Console.WriteLine(p);
      p.DisplayName();
   }
}
```

有关详细信息，请参阅方法（C# 编程指南）。

只读属性

可使用表达式主体定义来实现只读属性。 为此，请使用以下语法：

C#

```
PropertyType PropertyName => expression;
```

下面的示例定义 `Location` 类，其只读 `Name` 属性以表达式主体定义的形式实现，该表达式主体定义返回私有 `locationName` 字段值：

C#

```
public class Location
{
   private string locationName;

   public Location(string name)
   {
      locationName = name;
   }

   public string Name => locationName;
}
```

有关属性的详细信息，请参阅属性（C# 编程指南）。

属性

可使用表达式主体定义来实现属性 `get` 和 `set` 访问器。 下面的示例演示其实现方法：

C#

```
public class Location
{
   private string locationName;

   public Location(string name) => Name = name;

   public string Name
   {
      get => locationName;
      set => locationName = value;
   }
}
```

有关属性的详细信息，请参阅属性（C# 编程指南）。

构造函数

构造函数的表达式主体定义通常包含单个赋值表达式或一个方法调用，该方法调用可处理构造函数的参数，也可初始化实例状态。

以下示例定义 `Location` 类，其构造函数具有一个名为“name”的字符串参数。 表达式主体定义向 `Name` 属性分配参数。

C#

```
public class Location
{
   private string locationName;

   public Location(string name) => Name = name;

   public string Name
   {
      get => locationName;
      set => locationName = value;
   }
}
```

有关详细信息，请参阅构造函数（C# 编程指南）。

终结器

终结器的表达式主体定义通常包含清理语句，例如释放非托管资源的语句。

下面的示例定义了一个终结器，该终结器使用表达式主体定义来指示已调用该终结器。

C#

```
public class Destroyer
{
   public override string ToString() => GetType().Name;

   ~Destroyer() => Console.WriteLine($"The {ToString()} finalizer is
executing.");
}
```

有关详细信息，请参阅终结器（C# 编程指南）。

索引器

与使用属性一样，如果 `get` 访问器包含返回值的单个表达式或 `set` 访问器执行简单的赋值，则索引器 `get` 和 `set` 访问器包含表达式主体定义。

下面的示例定义名为 `Sports` 的类，其中包含一个内部 String 数组，该数组包含一些体育运动的名称。 索引器的 `get` 和 `set` 访问器都以表达式主体定义的形式实现。

C#

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

namespace SportsExample;

public class Sports
{
   private string[] types = { "Baseball", "Basketball", "Football",
                              "Hockey", "Soccer", "Tennis",
                              "Volleyball" };

   public string this[int i]
   {
      get => types[i];
      set => types[i] = value;
   }
}
```

有关详细信息，请参阅索引器（C# 编程指南）。

另请参阅

- 适用于表达式主体成员的 .NET 代码样式规则

相等性比较（C# 编程指南）

项目 • 2024/03/13

有时需要比较两个值是否相等。 在某些情况下，测试的是“值相等性”，也称为“等效性”，这意味着两个变量包含的值相等。 在其他情况下，必须确定两个变量是否引用内存中的同一基础对象。 此类型的相等性称为“引用相等性”或“标识”。 本主题介绍这两种相等性，并提供指向其他主题的链接，供用户了解详细信息。

引用相等性

引用相等性指两个对象引用均引用同一基础对象。 这可以通过简单的赋值来实现，如下面的示例所示。

C#

```
using System;
class Test
{
    public int Num { get; set; }
    public string Str { get; set; }

    public static void Main()
    {
        Test a = new Test() { Num = 1, Str = "Hi" };
        Test b = new Test() { Num = 1, Str = "Hi" };

        bool areEqual = System.Object.ReferenceEquals(a, b);
        // False:
        System.Console.WriteLine("ReferenceEquals(a, b) = {0}", areEqual);

        // Assign b to a.
        b = a;

        // Repeat calls with different results.
        areEqual = System.Object.ReferenceEquals(a, b);
        // True:
        System.Console.WriteLine("ReferenceEquals(a, b) = {0}", areEqual);
    }
}
```

在此代码中，创建了两个对象，但在赋值语句后，这两个引用所引用的是同一对象。 因此，它们具有引用相等性。 使用 ReferenceEquals 方法确定两个引用是否引用同一对象。

引用相等性的概念仅适用于引用类型。 由于在将值类型的实例赋给变量时将产生值的副本，因此值类型对象无法具有引用相等性。 因此，永远不会有两个未装箱结构引用内存

中的同一位置。 此外，如果使用 ReferenceEquals 比较两个值类型，结果将始终为 `false`，即使对象中包含的值都相同也是如此。 这是因为会将每个变量装箱到单独的对象实例中。 有关详细信息，请参阅如何测试引用相等性（标识）。

值相等性

值相等性指两个对象包含相同的一个或多个值。 对于基元值类型（例如 int 或 bool），针对值相等性的测试简单明了。 可以使用 == 运算符，如下面的示例所示。

C#

```
int a = GetOriginalValue();
int b = GetCurrentValue();

// Test for value equality.
if (b == a)
{
    // The two integers are equal.
}
```

对于大多数其他类型，针对值相等性的测试较为复杂，因为它需要用户了解类型对值相等性的定义方式。 对于具有多个字段或属性的类和结构，值相等性的定义通常指所有字段或属性都具有相同的值。 例如，如果 pointA.X 等于 pointB.X，并且 pointA.Y 等于 pointB.Y，则可以将两个 `Point` 对象定义为相等。 对记录来说，值相等性是指如果记录类型的两个变量类型相匹配，且所有属性和字段值都一致，那么记录类型的两个变量是相等的。

但是，并不要求类型中的所有字段均相等。 只需子集相等即可。 比较不具所有权的类型时，应确保明确了解相等性对于该类型是如何定义的。 若要详细了解如何在自己的类和结构中定义值相等性，请参阅如何为类型定义值相等性。

浮点值的值相等性

由于二进制计算机上的浮点算法不精确，因此浮点值（double 和 float）的相等比较会出现问题。 有关更多信息，请参阅 System.Double 主题中的备注部分。

相关主题

展开表

Title	描述
如何测试引用相等性（标识）	介绍如何确定两个变量是否具有引用相等性。
如何为类型定义值相等性	介绍如何为类型提供值相等性的自定义定义。
类型	提供有关 C# 类型系统的信息以及指向附加信息的链接。
记录	提供有关记录类型的信息，默认情况下，记录类型会测试值相等性。

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

如何为类或结构定义值相等性（C# 编程指南）

项目 • 2023/04/07

记录自动实现值相等性。 当你的类型为数据建模并应实现值相等性时，请考虑定义 `record` 而不是 `class`。

定义类或结构时，需确定为类型创建值相等性（或等效性）的自定义定义是否有意义。 通常，预期将类型的对象添加到集合时，或者这些对象主要用于存储一组字段或属性时，需实现值相等性。 可以基于类型中所有字段和属性的比较结果来定义值相等性，也可以基于子集进行定义。

在任何一种情况下，类和结构中的实现均应遵循 5 个等效性保证条件（对于以下规则，假设 `x`、`y` 和 `z` 都不为 null）：

1. 自反属性：`x.Equals(x)` 将返回 `true`。
2. 对称属性：`x.Equals(y)` 返回与 `y.Equals(x)` 相同的值。
3. 可传递属性：如果 `(x.Equals(y) && y.Equals(z))` 返回 `true`，则 `x.Equals(z)` 将返回 `true`。
4. 只要未修改 x 和 y 引用的对象，`x.Equals(y)` 的连续调用就将返回相同的值。
5. 任何非 null 值均不等于 null。 然而，当 `x` 为 null 时，`x.Equals(y)` 将引发异常。 这会违反规则 1 或 2，具体取决于 `Equals` 的参数。

定义的任何结构都已具有其从 Object.Equals(Object) 方法的 System.ValueType 替代中继承的值相等性的默认实现。 此实现使用反射来检查类型中的所有字段和属性。 尽管此实现可生成正确的结果，但与专门为类型编写的自定义实现相比，它的速度相对较慢。

类和结构的值相等性的实现详细信息有所不同。 但是，类和结构都需要相同的基础步骤来实现相等性：

1. 替代虚拟Object.Equals(Object) 方法。 大多数情况下，`bool Equals( object obj )` 实现应只调入作为 System.IEquatable<T> 接口的实现的类型特定 `Equals` 方法。（请参阅步骤 2。）
2. 通过提供类型特定的 `Equals` 方法实现 System.IEquatable<T> 接口。 实际的等效性比较将在此接口中执行。 例如，可能决定通过仅比较类型中的一两个字段来定义相等性。 不会从 `Equals` 引发异常。 对于与继承相关的类：

- 此方法应仅检查类中声明的字段。 它应调用 `base.Equals` 来检查基类中的字段。 （如果类型直接从 Object 中继承，则不会调用 `base.Equals`，因为 Object.Equals(Object) 的 Object 实现会执行引用相等性检查。）
- 仅当要比较的变量的运行时类型相同时，才应将两个变量视为相等。 此外，如果变量的运行时和编译时类型不同，请确保使用运行时类型的 `Equals` 方法的 `IEquatable` 实现。 确保始终正确比较运行时类型的一种策略是仅在 `sealed` 类中实现 `IEquatable`。 有关详细信息，请参阅本文后续部分的类示例。

3. 可选，但建议这样做：重载 == 和 != 运算符。
4. 替代 Object.GetHashCode，以便具有值相等性的两个对象生成相同的哈希代码。
5. 可选：若要支持“大于”或“小于”定义，请为类型实现 IComparable<T> 接口，并同时重载 < 和 > 运算符。

> **备注**
>
> 从 C# 9.0 开始，可以使用记录来获取值相等性语义，而不需要任何不必要的样板代码。

类示例

下面的示例演示如何在类（引用类型）中实现值相等性。

C#

```
namespace ValueEqualityClass;

class TwoDPoint : IEquatable<TwoDPoint>
{
    public int X { get; private set; }
    public int Y { get; private set; }

    public TwoDPoint(int x, int y)
    {
        if (x is (< 1 or > 2000) || y is (< 1 or > 2000))
        {
            throw new ArgumentException("Point must be in range 1 - 2000");
        }
        this.X = x;
        this.Y = y;
    }

    public override bool Equals(object obj) => this.Equals(obj as
```

```
TwoDPoint);

    public bool Equals(TwoDPoint p)
    {
        if (p is null)
        {
            return false;
        }

        // Optimization for a common success case.
        if (Object.ReferenceEquals(this, p))
        {
            return true;
        }

        // If run-time types are not exactly the same, return false.
        if (this.GetType() != p.GetType())
        {
            return false;
        }

        // Return true if the fields match.
        // Note that the base class is not invoked because it is
        // System.Object, which defines Equals as reference equality.
        return (X == p.X) && (Y == p.Y);
    }

    public override int GetHashCode() => (X, Y).GetHashCode();

    public static bool operator ==(TwoDPoint lhs, TwoDPoint rhs)
    {
        if (lhs is null)
        {
            if (rhs is null)
            {
                return true;
            }

            // Only the left side is null.
            return false;
        }
        // Equals handles case of null on right side.
        return lhs.Equals(rhs);
    }

    public static bool operator !=(TwoDPoint lhs, TwoDPoint rhs) => !(lhs ==
rhs);
}

// For the sake of simplicity, assume a ThreeDPoint IS a TwoDPoint.
class ThreeDPoint : TwoDPoint, IEquatable<ThreeDPoint>
{
    public int Z { get; private set; }

    public ThreeDPoint(int x, int y, int z)
```

```
        : base(x, y)
    {
        if ((z < 1) || (z > 2000))
        {
            throw new ArgumentException("Point must be in range 1 - 2000");
        }
        this.Z = z;
    }

    public override bool Equals(object obj) => this.Equals(obj as
ThreeDPoint);

    public bool Equals(ThreeDPoint p)
    {
        if (p is null)
        {
            return false;
        }

        // Optimization for a common success case.
        if (Object.ReferenceEquals(this, p))
        {
            return true;
        }

        // Check properties that this class declares.
        if (Z == p.Z)
        {
            // Let base class check its own fields
            // and do the run-time type comparison.
            return base.Equals((TwoDPoint)p);
        }
        else
        {
            return false;
        }
    }

    public override int GetHashCode() => (X, Y, Z).GetHashCode();

    public static bool operator ==(ThreeDPoint lhs, ThreeDPoint rhs)
    {
        if (lhs is null)
        {
            if (rhs is null)
            {
                // null == null = true.
                return true;
            }

            // Only the left side is null.
            return false;
        }
        // Equals handles the case of null on right side.
        return lhs.Equals(rhs);
```

```
    }

    public static bool operator !=(ThreeDPoint lhs, ThreeDPoint rhs) => !
(lhs == rhs);
}

class Program
{
    static void Main(string[] args)
    {
        ThreeDPoint pointA = new ThreeDPoint(3, 4, 5);
        ThreeDPoint pointB = new ThreeDPoint(3, 4, 5);
        ThreeDPoint pointC = null;
        int i = 5;

        Console.WriteLine("pointA.Equals(pointB) = {0}",
pointA.Equals(pointB));
        Console.WriteLine("pointA == pointB = {0}", pointA == pointB);
        Console.WriteLine("null comparison = {0}", pointA.Equals(pointC));
        Console.WriteLine("Compare to some other type = {0}",
pointA.Equals(i));

        TwoDPoint pointD = null;
        TwoDPoint pointE = null;

        Console.WriteLine("Two null TwoDPoints are equal: {0}", pointD ==
pointE);

        pointE = new TwoDPoint(3, 4);
        Console.WriteLine("(pointE == pointA) = {0}", pointE == pointA);
        Console.WriteLine("(pointA == pointE) = {0}", pointA == pointE);
        Console.WriteLine("(pointA != pointE) = {0}", pointA != pointE);

        System.Collections.ArrayList list = new
System.Collections.ArrayList();
        list.Add(new ThreeDPoint(3, 4, 5));
        Console.WriteLine("pointE.Equals(list[0]): {0}",
pointE.Equals(list[0]));

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}

/* Output:
    pointA.Equals(pointB) = True
    pointA == pointB = True
    null comparison = False
    Compare to some other type = False
    Two null TwoDPoints are equal: True
    (pointE == pointA) = False
    (pointA == pointE) = False
    (pointA != pointE) = True
```

```
    pointE.Equals(list[0]): False
*/
```

在类（引用类型）上，两种 Object.Equals(Object) 方法的默认实现均执行引用相等性比较，而不是值相等性检查。 实施者替代虚方法时，目的是为其指定值相等性语义。

即使类不重载 `==` 和 `!=` 运算符，也可将这些运算符与类一起使用。 但是，默认行为是执行引用相等性检查。 在类中，如果重载 `Equals` 方法，则应重载 `==` 和 `!=` 运算符，但这并不是必需的。

> **ⓘ 重要**
>
> 前面的示例代码可能无法按照预期的方式处理每个继承方案。 考虑下列代码：
>
> ```
> C#
>
> TwoDPoint p1 = new ThreeDPoint(1, 2, 3);
> TwoDPoint p2 = new ThreeDPoint(1, 2, 4);
> Console.WriteLine(p1.Equals(p2)); // output: True
> ```
>
> 根据此代码报告，尽管 `z` 值有所不同，但 `p1` 等于 `p2`。 由于编译器会根据编译时类型选取 `IEquatable` 的 `TwoDPoint` 实现，因而会忽略该差异。
>
> `record` 类型的内置值相等性可以正确处理这类场景。 如果 `TwoDPoint` 和 `ThreeDPoint` 是 `record` 类型，则 `p1.Equals(p2)` 的结果会是 `False`。 有关详细信息，请参阅 **record 类型继承层次结果中的相等性**。

结构示例

下面的示例演示如何在结构（值类型）中实现值相等性：

```
C#

namespace ValueEqualityStruct
{
    struct TwoDPoint : IEquatable<TwoDPoint>
    {
        public int X { get; private set; }
        public int Y { get; private set; }

        public TwoDPoint(int x, int y)
            : this()
        {
            if (x is (< 1 or > 2000) || y is (< 1 or > 2000))
            {
```

```
                throw new ArgumentException("Point must be in range 1 -
2000");
            }
            X = x;
            Y = y;
        }

        public override bool Equals(object? obj) => obj is TwoDPoint other
&& this.Equals(other);

        public bool Equals(TwoDPoint p) => X == p.X && Y == p.Y;

        public override int GetHashCode() => (X, Y).GetHashCode();

        public static bool operator ==(TwoDPoint lhs, TwoDPoint rhs) =>
lhs.Equals(rhs);

        public static bool operator !=(TwoDPoint lhs, TwoDPoint rhs) => !
(lhs == rhs);
    }

    class Program
    {
        static void Main(string[] args)
        {
            TwoDPoint pointA = new TwoDPoint(3, 4);
            TwoDPoint pointB = new TwoDPoint(3, 4);
            int i = 5;

            // True:
            Console.WriteLine("pointA.Equals(pointB) = {0}",
pointA.Equals(pointB));
            // True:
            Console.WriteLine("pointA == pointB = {0}", pointA == pointB);
            // True:
            Console.WriteLine("object.Equals(pointA, pointB) = {0}",
object.Equals(pointA, pointB));
            // False:
            Console.WriteLine("pointA.Equals(null) = {0}",
pointA.Equals(null));
            // False:
            Console.WriteLine("(pointA == null) = {0}", pointA == null);
            // True:
            Console.WriteLine("(pointA != null) = {0}", pointA != null);
            // False:
            Console.WriteLine("pointA.Equals(i) = {0}", pointA.Equals(i));
            // CS0019:
            // Console.WriteLine("pointA == i = {0}", pointA == i);

            // Compare unboxed to boxed.
            System.Collections.ArrayList list = new
System.Collections.ArrayList();
            list.Add(new TwoDPoint(3, 4));
            // True:
            Console.WriteLine("pointA.Equals(list[0]): {0}",
```

```
pointA.Equals(list[0]));

            // Compare nullable to nullable and to non-nullable.
            TwoDPoint? pointC = null;
            TwoDPoint? pointD = null;
            // False:
            Console.WriteLine("pointA == (pointC = null) = {0}", pointA ==
pointC);
            // True:
            Console.WriteLine("pointC == pointD = {0}", pointC == pointD);

            TwoDPoint temp = new TwoDPoint(3, 4);
            pointC = temp;
            // True:
            Console.WriteLine("pointA == (pointC = 3,4) = {0}", pointA ==
pointC);

            pointD = temp;
            // True:
            Console.WriteLine("pointD == (pointC = 3,4) = {0}", pointD ==
pointC);

            Console.WriteLine("Press any key to exit.");
            Console.ReadKey();
        }
    }

    /* Output:
        pointA.Equals(pointB) = True
        pointA == pointB = True
        Object.Equals(pointA, pointB) = True
        pointA.Equals(null) = False
        (pointA == null) = False
        (pointA != null) = True
        pointA.Equals(i) = False
        pointE.Equals(list[0]): True
        pointA == (pointC = null) = False
        pointC == pointD = True
        pointA == (pointC = 3,4) = True
        pointD == (pointC = 3,4) = True
    */
}
```

对于结构，Object.Equals(Object)（System.ValueType 中的替代版本）的默认实现通过使用反射来比较类型中每个字段的值，从而执行值相等性检查。 实施者替代结构中的 `Equals` 虚方法时，目的是提供更高效的方法来执行值相等性检查，并选择根据结构字段或属性的某个子集来进行比较。

除非结构显式重载了 == 和 != 运算符，否则这些运算符无法对结构进行运算。

请参阅

- 相等性比较
- C# 编程指南

如何测试引用相等性（标识） （C# 编程指南）

项目 • 2023/04/07

无需实现任何自定义逻辑，即可支持类型中的引用相等性比较。 此功能由静态 Object.ReferenceEquals 方法向所有类型提供。

以下示例演示如何确定两个变量是否具有引用相等性，即它们引用内存中的同一对象。

该示例还演示 Object.ReferenceEquals 为何始终为值类型返回 `false`，以及您为何不应使用 ReferenceEquals 来确定字符串相等性。

示例

C#

```
using System.Text;

namespace TestReferenceEquality
{
    struct TestStruct
    {
        public int Num { get; private set; }
        public string Name { get; private set; }

        public TestStruct(int i, string s) : this()
        {
            Num = i;
            Name = s;
        }
    }

    class TestClass
    {
        public int Num { get; set; }
        public string? Name { get; set; }
    }

    class Program
    {
        static void Main()
        {
            // Demonstrate reference equality with reference types.
            #region ReferenceTypes

            // Create two reference type instances that have identical
values.
```

```
            TestClass tcA = new TestClass() { Num = 1, Name = "New
TestClass" };
            TestClass tcB = new TestClass() { Num = 1, Name = "New
TestClass" };

            Console.WriteLine("ReferenceEquals(tcA, tcB) = {0}",
                              Object.ReferenceEquals(tcA, tcB)); // false

            // After assignment, tcB and tcA refer to the same object.
            // They now have reference equality.
            tcB = tcA;
            Console.WriteLine("After assignment: ReferenceEquals(tcA, tcB) =
{0}",
                              Object.ReferenceEquals(tcA, tcB)); // true

            // Changes made to tcA are reflected in tcB. Therefore, objects
            // that have reference equality also have value equality.
            tcA.Num = 42;
            tcA.Name = "TestClass 42";
            Console.WriteLine("tcB.Name = {0} tcB.Num: {1}", tcB.Name,
tcB.Num);
            #endregion

            // Demonstrate that two value type instances never have
reference equality.
            #region ValueTypes

            TestStruct tsC = new TestStruct( 1, "TestStruct 1");

            // Value types are copied on assignment. tsD and tsC have
            // the same values but are not the same object.
            TestStruct tsD = tsC;
            Console.WriteLine("After assignment: ReferenceEquals(tsC, tsD) =
{0}",
                              Object.ReferenceEquals(tsC, tsD)); // false
            #endregion

            #region stringRefEquality
            // Constant strings within the same assembly are always interned
by the runtime.
            // This means they are stored in the same location in memory.
Therefore,
            // the two strings have reference equality although no
assignment takes place.
            string strA = "Hello world!";
            string strB = "Hello world!";
            Console.WriteLine("ReferenceEquals(strA, strB) = {0}",
                            Object.ReferenceEquals(strA, strB)); // true

            // After a new string is assigned to strA, strA and strB
            // are no longer interned and no longer have reference equality.
            strA = "Goodbye world!";
            Console.WriteLine("strA = \"{0}\" strB = \"{1}\"", strA, strB);

            Console.WriteLine("After strA changes, ReferenceEquals(strA,
```

```
strB) = {0}",
                            Object.ReferenceEquals(strA, strB)); // false

            // A string that is created at runtime cannot be interned.
            StringBuilder sb = new StringBuilder("Hello world!");
            string stringC = sb.ToString();
            // False:
            Console.WriteLine("ReferenceEquals(stringC, strB) = {0}",
                            Object.ReferenceEquals(stringC, strB));

            // The string class overloads the == operator to perform an
equality comparison.
            Console.WriteLine("stringC == strB = {0}", stringC == strB); //
true

            #endregion

            // Keep the console open in debug mode.
            Console.WriteLine("Press any key to exit.");
            Console.ReadKey();
        }
    }
}

/* Output:
    ReferenceEquals(tcA, tcB) = False
    After assignment: ReferenceEquals(tcA, tcB) = True
    tcB.Name = TestClass 42 tcB.Num: 42
    After assignment: ReferenceEquals(tsC, tsD) = False
    ReferenceEquals(strA, strB) = True
    strA = "Goodbye world!" strB = "Hello world!"
    After strA changes, ReferenceEquals(strA, strB) = False
    ReferenceEquals(stringC, strB) = False
    stringC == strB = True
*/
```

在 System.Object 通用基类中实现 `Equals` 也会执行引用相等性检查，但最好不要使用这种检查，因为如果恰好某个类替代了此方法，结果可能会出乎意料。 以上情况同样适用于 `==` 和 `!=` 运算符。 当它们作用于引用类型时，`==` 和 `!=` 的默认行为是执行引用相等性检查。 但是，派生类可重载运算符，执行值相等性检查。 为了尽量降低错误的可能性，当需要确定两个对象是否具有引用相等性时，最好始终使用 ReferenceEquals。

运行时始终暂存同一程序集内的常量字符串。 也就是说，仅维护每个唯一文本字符串的一个实例。 但是，运行时不能保证会暂存在运行时创建的字符串，也不保证会暂存不同程序集中两个相等的常量字符串。

请参阅

- 相等比较

强制转换和类型转换（C# 编程指南）

项目 • 2024/03/20

由于 C# 是在编译时静态类型化的，因此变量在声明后就无法再次声明，或无法分配另一种类型的值，除非该类型可以隐式转换为变量的类型。 例如，`string` 无法隐式转换为 `int`。 因此，在将 `i` 声明为 `int` 后，无法将字符串“Hello”分配给它，如以下代码所示：

```
C#
```

```
int i;

// error CS0029: can't implicitly convert type 'string' to 'int'
i = "Hello";
```

但有时可能需要将值复制到其他类型的变量或方法参数中。 例如，可能需要将一个整数变量传递给参数类型化为 `double` 的方法。 或者可能需要将类变量分配给接口类型的变量。 这些类型的操作称为类型转换。 在 C# 中，可以执行以下几种类型的转换：

- **隐式转换**：不需要特殊语法，因为转换始终会成功，并且不会丢失任何数据。 示例包括从较小整数类型到较大整数类型的转换以及从派生类到基类的转换。
- **显式转换（强制转换）** ：必须使用强制转换表达式，才能执行显式转换。 在转换中可能丢失信息时或在出于其他原因转换可能不成功时，必须进行强制转换。 典型的示例包括从数值到精度较低或范围较小的类型的转换和从基类实例到派生类的转换。
- **用户定义的转换**：用户定义的转换使用你可以定义的特殊方法，以支持在不具有基类和派生类关系的自定义类型之间实现显式和隐式转换。 有关详细信息，请参阅用户定义转换运算符。
- **使用帮助程序类进行转换**：若要在非兼容类型（如整数和 System.DateTime 对象，或十六进制字符串和字节数组）之间转换，可使用 System.BitConverter 类、System.Convert 类和内置数值类型的 `Parse` 方法（如 Int32.Parse）。 有关详细信息，请参见如何将字节数组转换为 int、如何将字符串转换为数字和如何在十六进制字符串与数值类型之间转换。

隐式转换

对于内置数值类型，如果要存储的值无需截断或四舍五入即可适应变量，则可以进行隐式转换。 对于整型类型，这意味着源类型的范围是目标类型范围的正确子集。 例如，long

类型的变量（64 位整数）能够存储 int（32 位整数）可存储的任何值。 在下面的示例中，编译器先将右侧的 `num` 值隐式转换为 `long` 类型，再将它赋给 `bigNum`。

C#

```
// Implicit conversion. A long can
// hold any value an int can hold, and more!
int num = 2147483647;
long bigNum = num;
```

有关所有隐式数值转换的完整列表，请参阅内置数值转换一文的隐式数值转换表部分。

对于引用类型，隐式转换始终存在于从一个类转换为该类的任何一个直接或间接的基类或接口的情况。 由于派生类始终包含基类的所有成员，因此不必使用任何特殊语法。

C#

```
Derived d = new Derived();

// Always OK.
Base b = d;
```

显式转换

但是，如果进行转换可能会导致信息丢失，则编译器会要求执行显式转换，显式转换也称为*强制转换*。 强制转换是显式告知编译器以下信息的一种方式：你打算进行转换且你知道可能会发生数据丢失，或者你知道强制转换有可能在运行时失败。 若要执行强制转换，请在要转换的值或变量前面的括号中指定要强制转换到的类型。 下面的程序将 double 强制转换为 int。如不强制转换，程序将无法编译。

C#

```
class Test
{
    static void Main()
    {
        double x = 1234.7;
        int a;
        // Cast double to int.
        a = (int)x;
        System.Console.WriteLine(a);
    }
}
// Output: 1234
```

有关支持的显式数值转换的完整列表，请参阅内置数值转换一文的显式数值转换部分。

对于引用类型，如果需要从基类型转换为派生类型，则必须进行显式强制转换：

C#

```
// Create a new derived type.
Giraffe g = new Giraffe();

// Implicit conversion to base type is safe.
Animal a = g;

// Explicit conversion is required to cast back
// to derived type. Note: This will compile but will
// throw an exception at run time if the right-side
// object is not in fact a Giraffe.
Giraffe g2 = (Giraffe)a;
```

引用类型之间的强制转换操作不会更改基础对象的运行时类型；它只更改用作对该对象引用的值的类型。 有关详细信息，请参阅多态性。

运行时的类型转换异常

在某些引用类型转换中，编译器无法确定强制转换是否会有效。 正确进行编译的强制转换操作有可能在运行时失败。 如下面的示例所示，类型转换在运行时失败将导致引发 InvalidCastException。

C#

```
class Animal
{
    public void Eat() => System.Console.WriteLine("Eating.");

    public override string ToString() => "I am an animal.";
}

class Reptile : Animal { }
class Mammal : Animal { }

class UnSafeCast
{
    static void Main()
    {
        Test(new Mammal());

        // Keep the console window open in debug mode.
        System.Console.WriteLine("Press any key to exit.");
        System.Console.ReadKey();
    }

    static void Test(Animal a)
    {
```

```
        // System.InvalidCastException at run time
        // Unable to cast object of type 'Mammal' to type 'Reptile'
        Reptile r = (Reptile)a;
    }
}
```

`Test` 方法有一个 `Animal` 形式参数，因此，将实际参数 `a` 显式强制转换为 `Reptile` 会造成危险的假设。 更安全的做法是不要做出假设，而是检查类型。 C# 提供 is 运算符，使你可以在实际执行强制转换之前测试兼容性。 有关详细信息，请参阅如何使用模式匹配以及 as 和 is 运算符安全地进行强制转换。

C# 语言规范

有关详细信息，请参阅 C# 语言规范中的转换部分。

另请参阅

- 类型
- 强制转换表达式
- 用户定义转换运算符
- 通用类型转换
- 如何将字符串转换为数字

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

装箱和取消装箱（C# 编程指南）

项目 • 2023/04/07

装箱是将值类型转换为 `object` 类型或由此值类型实现的任何接口类型的过程。 常见语言运行时 (CLR) 对值类型进行装箱时，会将值包装在 System.Object 实例中并将其存储在托管堆中。 取消装箱将从对象中提取值类型。 装箱是隐式的；取消装箱是显式的。 装箱和取消装箱的概念是类型系统 C# 统一视图的基础，其中任一类型的值都被视为一个对象。

下例将整型变量 `i` 进行了装箱并分配给对象 `o`。

C#

```
int i = 123;
// The following line boxes i.
object o = i;
```

然后，可以将对象 `o` 取消装箱并分配给整型变量 `i`：

C#

```
o = 123;
i = (int)o;  // unboxing
```

以下示例演示如何在 C# 中使用装箱。

C#

```
// String.Concat example.
// String.Concat has many versions. Rest the mouse pointer on
// Concat in the following statement to verify that the version
// that is used here takes three object arguments. Both 42 and
// true must be boxed.
Console.WriteLine(String.Concat("Answer", 42, true));

// List example.
// Create a list of objects to hold a heterogeneous collection
// of elements.
List<object> mixedList = new List<object>();

// Add a string element to the list.
mixedList.Add("First Group:");

// Add some integers to the list.
for (int j = 1; j < 5; j++)
{
    // Rest the mouse pointer over j to verify that you are adding
    // an int to a list of objects. Each element j is boxed when
```

```
    // you add j to mixedList.
    mixedList.Add(j);
}

// Add another string and more integers.
mixedList.Add("Second Group:");
for (int j = 5; j < 10; j++)
{
    mixedList.Add(j);
}

// Display the elements in the list. Declare the loop variable by
// using var, so that the compiler assigns its type.
foreach (var item in mixedList)
{
    // Rest the mouse pointer over item to verify that the elements
    // of mixedList are objects.
    Console.WriteLine(item);
}

// The following loop sums the squares of the first group of boxed
// integers in mixedList. The list elements are objects, and cannot
// be multiplied or added to the sum until they are unboxed. The
// unboxing must be done explicitly.
var sum = 0;
for (var j = 1; j < 5; j++)
{
    // The following statement causes a compiler error: Operator
    // '*' cannot be applied to operands of type 'object' and
    // 'object'.
    //sum += mixedList[j] * mixedList[j]);

    // After the list elements are unboxed, the computation does
    // not cause a compiler error.
    sum += (int)mixedList[j] * (int)mixedList[j];
}

// The sum displayed is 30, the sum of 1 + 4 + 9 + 16.
Console.WriteLine("Sum: " + sum);

// Output:
// Answer42True
// First Group:
// 1
// 2
// 3
// 4
// Second Group:
// 5
// 6
// 7
// 8
// 9
// Sum: 30
```

性能

相对于简单的赋值而言，装箱和取消装箱过程需要进行大量的计算。 对值类型进行装箱时，必须分配并构造一个新对象。 取消装箱所需的强制转换也需要进行大量的计算，只是程度较轻。 有关更多信息，请参阅性能。

装箱

装箱用于在垃圾回收堆中存储值类型。 装箱是值类型到 `object` 类型或到此值类型所实现的任何接口类型的隐式转换。 对值类型装箱会在堆中分配一个对象实例，并将该值复制到新的对象中。

请看以下值类型变量的声明：

```
C#

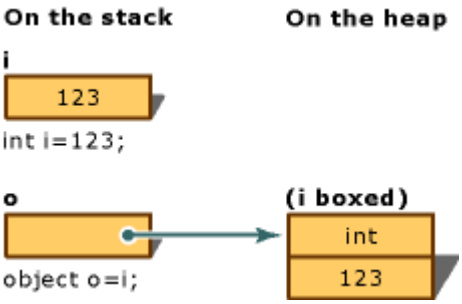
int i = 123;
```

以下语句对变量 `i` 隐式应用了装箱操作：

```
C#

// Boxing copies the value of i into object o.
object o = i;
```

此语句的结果是在堆栈上创建对象引用 `o`，而在堆上则引用 `int` 类型的值。 该值是赋给变量 `i` 的值类型值的一个副本。 以下装箱转换图说明了 `i` 和 `o` 这两个变量之间的差异：



还可以像下面的示例一样执行显式装箱，但显式装箱从来不是必需的：

```
C#

int i = 123;
object o = (object)i;  // explicit boxing
```

示例

此示例使用装箱将整型变量 `i` 转换为对象 `o`。 这样一来，存储在变量 `i` 中的值就从 `123` 更改为 `456`。 该示例表明原始值类型和装箱的对象使用不同的内存位置，因此能够存储不同的值。

C#

```
class TestBoxing
{
    static void Main()
    {
        int i = 123;

        // Boxing copies the value of i into object o.
        object o = i;

        // Change the value of i.
        i = 456;

        // The change in i doesn't affect the value stored in o.
        System.Console.WriteLine("The value-type value = {0}", i);
        System.Console.WriteLine("The object-type value = {0}", o);
    }
}
/* Output:
    The value-type value = 456
    The object-type value = 123
*/
```

取消装箱

取消装箱是从 `object` 类型到值类型或从接口类型到实现该接口的值类型的显式转换。 取消装箱操作包括：

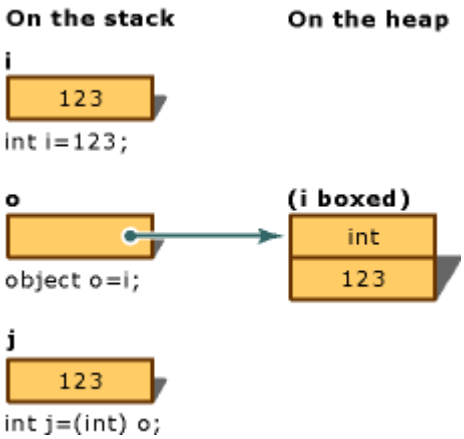
- 检查对象实例，以确保它是给定值类型的装箱值。
- 将该值从实例复制到值类型变量中。

下面的语句演示装箱和取消装箱两种操作：

C#

```
int i = 123;      // a value type
object o = i;     // boxing
int j = (int)o;   // unboxing
```

下图演示了上述语句的结果：



要在运行时成功取消装箱值类型，被取消装箱的项必须是对一个对象的引用，该对象是先前通过装箱该值类型的实例创建的。 尝试取消装箱 `null` 会导致 NullReferenceException。 尝试取消装箱对不兼容值类型的引用会导致 InvalidCastException。

示例

下面的示例演示无效的取消装箱及引发的 `InvalidCastException`。 使用 `try` 和 `catch`，在发生错误时显示错误信息。

C#

```
class TestUnboxing
{
    static void Main()
    {
        int i = 123;
        object o = i;  // implicit boxing

        try
        {
            int j = (short)o;  // attempt to unbox

            System.Console.WriteLine("Unboxing OK.");
        }
        catch (System.InvalidCastException e)
        {
            System.Console.WriteLine("{0} Error: Incorrect unboxing.",
e.Message);
        }
    }
}
```

此程序输出：

```
Specified cast is not valid. Error: Incorrect unboxing.
```

如果将下列语句：

```
C#
int j = (short)o;
```

更改为：

```
C#
int j = (int)o;
```

将执行转换，并将得到以下输出：

```
Unboxing OK.
```

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# 编程指南
- 引用类型
- 值类型

如何将字节数组转换为 int（C# 编程指南）

项目 • 2023/04/07

此示例演示如何使用 BitConverter 类将字节数组转换为 int 然后又转换回字节数组。 例如，在从网络读取字节之后，可能需要将字节转换为内置数据类型。 除了示例中的 ToInt32(Byte[], Int32) 方法之外，下表还列出了 BitConverter 类中将字节（来自字节数组）转换为其他内置类型的方法。

返回类型	方法
`bool`	ToBoolean(Byte[], Int32)
`char`	ToChar(Byte[], Int32)
`double`	ToDouble(Byte[], Int32)
`short`	ToInt16(Byte[], Int32)
`int`	ToInt32(Byte[], Int32)
`long`	ToInt64(Byte[], Int32)
`float`	ToSingle(Byte[], Int32)
`ushort`	ToUInt16(Byte[], Int32)
`uint`	ToUInt32(Byte[], Int32)
`ulong`	ToUInt64(Byte[], Int32)

示例

此示例初始化字节数组，并在计算机体系结构为 little-endian（即首先存储最低有效字节）的情况下反转数组，然后调用 ToInt32(Byte[], Int32) 方法以将数组中的四个字节转换为 `int`。 ToInt32(Byte[], Int32) 的第二个参数指定字节数组的起始索引。

> **备注**
>
> 输出可能会根据计算机体系结构的字节顺序而不同。

```
C#
```

```
byte[] bytes = { 0, 0, 0, 25 };

// If the system architecture is little-endian (that is, little end first),
// reverse the byte array.
if (BitConverter.IsLittleEndian)
    Array.Reverse(bytes);

int i = BitConverter.ToInt32(bytes, 0);
Console.WriteLine("int: {0}", i);
// Output: int: 25
```

在本示例中，将调用 BitConverter 类的 GetBytes(Int32) 方法，将 `int` 转换为字节数组。

> ⓘ **备注**
>
> 输出可能会根据计算机体系结构的字节顺序而不同。

C#

```
byte[] bytes = BitConverter.GetBytes(201805978);
Console.WriteLine("byte array: " + BitConverter.ToString(bytes));
// Output: byte array: 9A-50-07-0C
```

请参阅

- BitConverter
- IsLittleEndian
- 类型

如何将字符串转换为数字（C# 编程指南）

项目 • 2024/11/07

你可以调用数值类型（`int`、`long`、`double` 等）中找到的 `Parse` 或 `TryParse` 方法或使用 System.Convert 类中的方法将 `string` 转换为数字。

调用 `TryParse` 方法（例如，int.TryParse("11", out number)）或 `Parse` 方法（例如，var number = int.Parse("11")）会稍微高效和简单一些。 使用 Convert 方法对于实现 IConvertible 的常规对象更有用。

对预期字符串会包含的数值类型（如 System.Int32 类型）使用 `Parse` 或 `TryParse` 方法。 Convert.ToInt32 方法在内部使用 Parse。 `Parse` 方法返回转换后的数字； `TryParse` 方法返回布尔值，该值指示转换是否成功，并以 `out` 参数形式返回转换后的数字。 如果字符串的格式无效，则 `Parse` 会引发异常，但 `TryParse` 会返回 `false`。 调用 `Parse` 方法时，应始终使用异常处理来捕获分析操作失败时的 FormatException。

调用 Parse 或 TryParse 方法

`Parse` 和 `TryParse` 方法会忽略字符串开头和末尾的空格，但所有其他字符都必须是组成合适数值类型（`int`、`long`、`ulong`、`float`、`decimal` 等）的字符。 如果组成数字的字符串中有任何空格，都会导致错误。 例如，可以使用 `decimal.TryParse` 分析“10”、“10.3”或“ 10 ”，但不能使用此方法分析从“10X”、“1 0”（注意嵌入的空格）、“10 .3”（注意嵌入的空格）、“10e1”（`float.TryParse` 在此处适用）等中分析出 10。 无法成功分析值为 `null` 或 String.Empty 的字符串。 在尝试通过调用 String.IsNullOrEmpty 方法分析字符串之前，可以检查字符串是否为 Null 或为空。

下面的示例演示了对 `Parse` 和 `TryParse` 的成功调用和不成功的调用。

C#

```
using System;

public static class StringConversion
{
    public static void Main()
    {
        string input = String.Empty;
        try
        {
            int result = Int32.Parse(input);
            Console.WriteLine(result);
```

```
        }
        catch (FormatException)
        {
            Console.WriteLine($"Unable to parse '{input}'");
        }
        // Output: Unable to parse ''

        try
        {
            int numVal = Int32.Parse("-105");
            Console.WriteLine(numVal);
        }
        catch (FormatException e)
        {
            Console.WriteLine(e.Message);
        }
        // Output: -105

        if (Int32.TryParse("-105", out int j))
        {
            Console.WriteLine(j);
        }
        else
        {
            Console.WriteLine("String could not be parsed.");
        }
        // Output: -105

        try
        {
            int m = Int32.Parse("abc");
        }
        catch (FormatException e)
        {
            Console.WriteLine(e.Message);
        }
        // Output: Input string was not in a correct format.

        const string inputString = "abc";
        if (Int32.TryParse(inputString, out int numValue))
        {
            Console.WriteLine(numValue);
        }
        else
        {
            Console.WriteLine($"Int32.TryParse could not parse
'{inputString}' to an int.");
        }
        // Output: Int32.TryParse could not parse 'abc' to an int.
    }
}
```

下面的示例演示了一种分析字符串的方法，该字符串应包含前导数字字符（包括十六进制字符）和尾随的非数字字符。 在调用 TryParse 方法之前，它从字符串的开头向新字符串分配有效字符。 因为要分析的字符串包含少量字符，所以本示例调用 String.Concat 方法将有效字符分配给新字符串。 对于较大的字符串，可以改用 StringBuilder 类。

C#

```
using System;

public static class StringConversion
{
    public static void Main()
    {
        var str = "  10FFxxx";
        string numericString = string.Empty;
        foreach (var c in str)
        {
            // Check for numeric characters (hex in this case) or leading or
trailing spaces.
            if ((c >= '0' && c <= '9') || (char.ToUpperInvariant(c) >= 'A'
&& char.ToUpperInvariant(c) <= 'F') || c == ' ')
            {
                numericString = string.Concat(numericString, c.ToString());
            }
            else
            {
                break;
            }
        }

        if (int.TryParse(numericString,
System.Globalization.NumberStyles.HexNumber, null, out int i))
        {
            Console.WriteLine($"'{str}' --> '{numericString}' --> {i}");
        }
        // Output: '  10FFxxx' --> '  10FF' --> 4351

        str = "   -10FFXXX";
        numericString = "";
        foreach (char c in str)
        {
            // Check for numeric characters (0-9), a negative sign, or
leading or trailing spaces.
            if ((c >= '0' && c <= '9') || c == ' ' || c == '-')
            {
                numericString = string.Concat(numericString, c);
            }
            else
            {
                break;
            }
        }
```

```
        if (int.TryParse(numericString, out int j))
        {
            Console.WriteLine($"'{str}' --> '{numericString}' --> {j}");
        }
        // Output: '    -10FFXXX' --> '    -10' --> -10
    }
}
```

调用 Convert 方法

下表列出了 Convert 类中可用于将字符串转换为数字的一些方法。

展开表

数值类型	方法
`decimal`	ToDecimal(String)
`float`	ToSingle(String)
`double`	ToDouble(String)
`short`	ToInt16(String)
`int`	ToInt32(String)
`long`	ToInt64(String)
`ushort`	ToUInt16(String)
`uint`	ToUInt32(String)
`ulong`	ToUInt64(String)

下面的示例调用 Convert.ToInt32(String) 方法将输入字符串转换为 int。该示例将捕获由此方法引发的两个最常见异常：FormatException 和 OverflowException。 如果生成的数字可以在不超过 Int32.MaxValue 的情况下递增，则示例将向结果添加 1 并显示输出。

C#

```
using System;

public class ConvertStringExample1
{
    static void Main(string[] args)
    {
        int numVal = -1;
        bool repeat = true;
```

```
        while (repeat)
        {
            Console.Write("Enter a number between -2,147,483,648 and
+2,147,483,647 (inclusive): ");

            string? input = Console.ReadLine();

            // ToInt32 can throw FormatException or OverflowException.
            try
            {
                numVal = Convert.ToInt32(input);
                if (numVal < Int32.MaxValue)
                {
                    Console.WriteLine("The new value is {0}", ++numVal);
                }
                else
                {
                    Console.WriteLine("numVal cannot be incremented beyond
its current value");
                }
            }
            catch (FormatException)
            {
                Console.WriteLine("Input string is not a sequence of
digits.");
            }
            catch (OverflowException)
            {
                Console.WriteLine("The number cannot fit in an Int32.");
            }

            Console.Write("Go again? Y/N: ");
            string? go = Console.ReadLine();
            if (go?.ToUpper() != "Y")
            {
                repeat = false;
            }
        }
    }
}
// Sample Output:
//   Enter a number between -2,147,483,648 and +2,147,483,647 (inclusive):
473
//   The new value is 474
//   Go again? Y/N: y
//   Enter a number between -2,147,483,648 and +2,147,483,647 (inclusive):
2147483647
//   numVal cannot be incremented beyond its current value
//   Go again? Y/N: y
//   Enter a number between -2,147,483,648 and +2,147,483,647 (inclusive):
-1000
//   The new value is -999
//   Go again? Y/N: n
```

使用 GitHub Copilot 将字符串转换为数字

可以在 IDE 中使用 GitHub Copilot 生成代码，以将字符串转换为 C# 中的数字。

如果使用 Visual Studio 2022 版本 17.8 或更高版本，则可以尝试在 Visual Studio 中使用 AI 驱动的 GitHub Copilot 生成代码，以将字符串转换为数字。 在 Copilot Chat 窗口中以提示形式提交问题，如下例所示。 还可以在编辑器窗口中使用内联聊天提交提示。

> **① 备注**
>
> GitHub Copilot 由 AI 提供支持，因此可能会带来意外和错误。 请确保验证任何生成的代码或建议。 有关 GitHub Copilot 的常规用途、产品影响、人工监督和隐私的更多信息，请参阅 GitHub Copilot 常见问题解答 。

以下文本显示了 Copilot 聊天的示例提示：

> 生成 C# 代码，以使用 TryParse 将字符串“123”转换为数字。 提供示例输出。

可以根据要求自定义提示以使用字符串。

可以使用聊天功能（如斜杠命令、引用文件、方法或类和线程）来设置意向，并通过已限定范围上下文获取更好的答案。 对于在 IDE 中打开的现有文件，可以使用内联聊天 `/generate code to convert the string string1 in #Filename to a number` 提示 GitHub Copilot。

以下输出显示了 Copilot 聊天响应示例：

输出

```
Here is a C# code snippet to convert the string "123" to a number using
TryParse:

using System;

class Program
{
    static void Main()
    {
        string numberString = "123";
        int number;
        bool success = int.TryParse(numberString, out number);

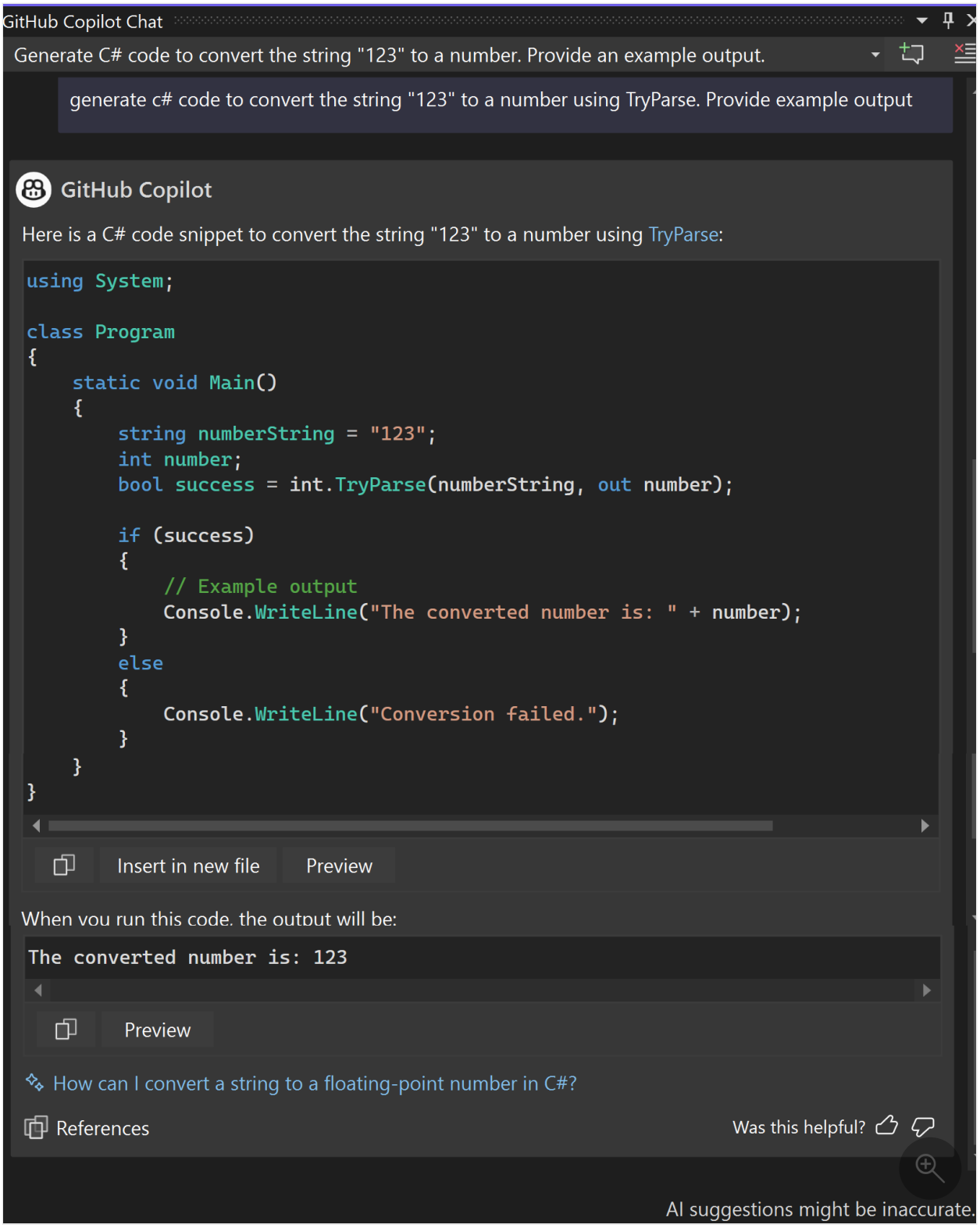
        if (success)
        {
            // Example output
            Console.WriteLine("The converted number is: " + number);
```

```
        }
        else
        {
            Console.WriteLine("Conversion failed.");
        }
    }
}

When you run this code, the output will be:
The converted number is: 123
```

当 Copilot 返回代码块时，响应包括用于复制代码、将代码插入新文件或预览代码输出的选项。

> **① 备注**
>
> 结果可能与示例响应中显示的结果不同。 AI 模型不确定，这意味着当被问及相同的问题时，它们可以返回不同的响应。 这可能是由于一段时间内进行的额外学习和调整、语言变体、上下文中的更改（如聊天历史记录等）。



有关详细信息，请参阅：

- GitHub Copilot 信任中心
- Visual Studio 中的 GitHub Copilot
- VS Code 中的 GitHub Copilot

如何在十六进制字符串与数值类型之间转换（C# 编程指南）

项目 • 2023/05/10

以下示例演示如何执行下列任务：

- 获取字符串中每个字符的十六进制值。
- 获取与十六进制字符串中的每个值对应的 char。
- 将十六进制 `string` 转换为 int。
- 将十六进制 `string` 转换为 float。
- 将字节数组转换为十六进制 `string`。

示例

此示例输出 `string` 中每个字符的十六进制值。 首先，将 `string` 分析为字符数组。 然后，对每个字符调用 ToInt32(Char)获取相应的数值。 最后，在 `string` 中将数字的格式设置为十六进制表示形式。

C#

```
string input = "Hello World!";
char[] values = input.ToCharArray();
foreach (char letter in values)
{
    // Get the integral value of the character.
    int value = Convert.ToInt32(letter);
    // Convert the integer value to a hexadecimal value in string form.
    Console.WriteLine($"Hexadecimal value of {letter} is {value:X}");
}
/* Output:
    Hexadecimal value of H is 48
    Hexadecimal value of e is 65
    Hexadecimal value of l is 6C
    Hexadecimal value of l is 6C
    Hexadecimal value of o is 6F
    Hexadecimal value of   is 20
    Hexadecimal value of W is 57
    Hexadecimal value of o is 6F
    Hexadecimal value of r is 72
    Hexadecimal value of l is 6C
    Hexadecimal value of d is 64
```

```
    Hexadecimal value of ! is 21
 */
```

此示例分析十六进制值的 `string` 并输出对应于每个十六进制值的字符。 首先，调用 Split(Char[]) 方法以获取每个十六进制值作为数组中的单个 `string`。 然后，调用 ToInt32(String, Int32)将十六进制值转换为表示为 int 的十进制值。示例中演示了 2 种不同方法，用于获取对应于该字符代码的字符。 第 1 种方法是使用 ConvertFromUtf32(Int32)，它将对应于整型参数的字符作为 `string` 返回。 第 2 种方法是将 `int` 显式转换为 char。

C#

```
string hexValues = "48 65 6C 6C 6F 20 57 6F 72 6C 64 21";
string[] hexValuesSplit = hexValues.Split(' ');
foreach (string hex in hexValuesSplit)
{
    // Convert the number expressed in base-16 to an integer.
    int value = Convert.ToInt32(hex, 16);
    // Get the character corresponding to the integral value.
    string stringValue = Char.ConvertFromUtf32(value);
    char charValue = (char)value;
    Console.WriteLine("hexadecimal value = {0}, int value = {1}, char value
= {2} or {3}",
                      hex, value, stringValue, charValue);
}
/* Output:
    hexadecimal value = 48, int value = 72, char value = H or H
    hexadecimal value = 65, int value = 101, char value = e or e
    hexadecimal value = 6C, int value = 108, char value = l or l
    hexadecimal value = 6C, int value = 108, char value = l or l
    hexadecimal value = 6F, int value = 111, char value = o or o
    hexadecimal value = 20, int value = 32, char value =   or
    hexadecimal value = 57, int value = 87, char value = W or W
    hexadecimal value = 6F, int value = 111, char value = o or o
    hexadecimal value = 72, int value = 114, char value = r or r
    hexadecimal value = 6C, int value = 108, char value = l or l
    hexadecimal value = 64, int value = 100, char value = d or d
    hexadecimal value = 21, int value = 33, char value = ! or !
*/
```

此示例演示了将十六进制 `string` 转换为整数的另一种方法，即调用 Parse(String, NumberStyles) 方法。

C#

```
string hexString = "8E2";
int num = Int32.Parse(hexString,
System.Globalization.NumberStyles.HexNumber);
```

```
Console.WriteLine(num);
//Output: 2274
```

下面的示例演示了如何使用 System.BitConverter 类和 UInt32.Parse 方法将十六进制 `string` 转换为 float。

C#

```
string hexString = "43480170";
uint num = uint.Parse(hexString,
System.Globalization.NumberStyles.AllowHexSpecifier);

byte[] floatVals = BitConverter.GetBytes(num);
float f = BitConverter.ToSingle(floatVals, 0);
Console.WriteLine("float convert = {0}", f);

// Output: 200.0056
```

下面的示例演示了如何使用 System.BitConverter 类将字节数组转换为十六进制字符串。

C#

```
byte[] vals = { 0x01, 0xAA, 0xB1, 0xDC, 0x10, 0xDD };

string str = BitConverter.ToString(vals);
Console.WriteLine(str);

str = BitConverter.ToString(vals).Replace("-", "");
Console.WriteLine(str);

/*Output:
  01-AA-B1-DC-10-DD
  01AAB1DC10DD
 */
```

下面的示例演示如何通过调用 .NET 5.0 中引入的 Convert.ToHexString 方法，将字节数组转换为十六进制字符串。

C#

```
byte[] array = { 0x64, 0x6f, 0x74, 0x63, 0x65, 0x74 };

string hexValue = Convert.ToHexString(array);
Console.WriteLine(hexValue);

/*Output:
  646F74636574
 */
```

请参阅

- 标准数字格式字符串
- 类型
- 如何确定字符串是否表示数值

使用 Override 和 New 关键字进行版本控制（C# 编程指南）

项目 • 2023/04/07

C# 语言经过专门设计，以便不同库中的基类与派生类之间的版本控制可以不断向前发展，同时保持后向兼容。 这具有多方面的意义。例如，这意味着在基类中引入与派生类中的某个成员具有相同名称的新成员在 C# 中是完全支持的，不会导致意外行为。 它还意味着类必须显式声明某方法是要替代一个继承方法，还是本身就是一个隐藏具有类似名称的继承方法的新方法。

在 C# 中，派生类可以包含与基类方法同名的方法。

- 如果派生类中的方法前面没有 new 或 override 关键字，则编译器将发出警告，该方法将如同存在 `new` 关键字一样执行操作。
- 如果派生类中的方法前面带有 `new` 关键字，则该方法被定义为独立于基类中的方法。
- 如果派生类中的方法前面带有 `override` 关键字，则派生类的对象将调用该方法，而不是调用基类方法。
- 若要将 `override` 关键字应用于派生类中的方法，必须以虚拟形式定义基类方法。
- 可以从派生类中使用 `base` 关键字调用基类方法。
- `override`、`virtual` 和 `new` 关键字还可以用于属性、索引器和事件中。

默认情况下，C# 方法为非虚方法。 如果某个方法被声明为虚方法，则继承该方法的任何类都可以实现它自己的版本。 若要使方法成为虚方法，需要在基类的方法声明中使用 `virtual` 修饰符。 然后，派生类可以使用 `override` 关键字替代基虚方法，或使用 `new` 关键字隐藏基类中的虚方法。 如果 `override` 关键字和 `new` 关键字均未指定，编译器将发出警告，并且派生类中的方法将隐藏基类中的方法。

为了在实践中演示上述情况，暂时假定公司 A 创建了一个名为 `GraphicsClass` 的类，程序将使用此类。 `GraphicsClass` 如下所示：

C#

```
class GraphicsClass
{
    public virtual void DrawLine() { }
    public virtual void DrawPoint() { }
}
```

公司使用此类，并且你在添加新方法时将其用于派生自己的类：

C#

```
class YourDerivedGraphicsClass : GraphicsClass
{
    public void DrawRectangle() { }
}
```

你的应用程序运行正常，未出现问题，直到公司 A 发布了 `GraphicsClass` 的新版本，类似于以下代码：

C#

```
class GraphicsClass
{
    public virtual void DrawLine() { }
    public virtual void DrawPoint() { }
    public virtual void DrawRectangle() { }
}
```

现在，`GraphicsClass` 的新版本中包含一个名为 `DrawRectangle` 的方法。 开始时，没有出现任何问题。 新版本仍然与旧版本保持二进制兼容。 已经部署的任何软件都将继续正常工作，即使新类已安装到这些计算机系统上。 在你的派生类中，对方法 `DrawRectangle` 的任何现有调用将继续引用你的版本。

但是，一旦你使用 `GraphicsClass` 的新版本重新编译应用程序，就会收到来自编译器的警告 CS0108。 此警告提示，必须考虑你所期望的 `DrawRectangle` 方法在应用程序中的工作方式。

如果需要自己的方法替代新的基类方法，请使用 `override` 关键字：

C#

```
class YourDerivedGraphicsClass : GraphicsClass
{
    public override void DrawRectangle() { }
}
```

`override` 关键字可确保派生自 `YourDerivedGraphicsClass` 的任何对象都将使用 `DrawRectangle` 的派生类版本。 派生自 `YourDerivedGraphicsClass` 的对象仍可以使用 base 关键字访问 `DrawRectangle` 的基类版本：

C#

```
base.DrawRectangle();
```

如果不需要自己的方法替代新的基类方法，则需要注意以下事项。 为了避免这两个方法之间发生混淆，可以重命名你的方法。 这可能很耗费时间且容易出错，而且在某些情况下并不可行。 但是，如果项目相对较小，则可以使用 Visual Studio 的重构选项来重命名方法。 有关详细信息，请参阅重构类和类型（类设计器）。

或者，也可以通过在派生类定义中使用关键字 `new` 来防止出现该警告：

C#

```
class YourDerivedGraphicsClass : GraphicsClass
{
    public new void DrawRectangle() { }
}
```

使用 `new` 关键字可告诉编译器你的定义将隐藏基类中包含的定义。 这是默认行为。

替代和方法选择

当在类中对方法进行命名时，如果有多个方法与调用兼容（例如，存在两种同名的方法，并且其参数与传递的参数兼容），则 C# 编译器将选择最佳方法进行调用。 以下方法将是兼容的：

C#

```
public class Derived : Base
{
    public override void DoWork(int param) { }
    public void DoWork(double param) { }
}
```

在 `Derived` 的一个实例中调用 `DoWork` 时，C# 编译器将首先尝试使该调用与最初在 `Derived` 上声明的 `DoWork` 版本兼容。 替代方法不被视为是在类上进行声明的，而是在基类上声明的方法的新实现。 仅当 C# 编译器无法将方法调用与 `Derived` 上的原始方法匹配时，才尝试将该调用与具有相同名称和兼容参数的替代方法匹配。 例如：

C#

```
int val = 5;
Derived d = new Derived();
d.DoWork(val);  // Calls DoWork(double).
```

由于变量 `val` 可以隐式转换为 double 类型，因此 C# 编译器将调用 `DoWork(double)`，而不是 `DoWork(int)`。 有两种方法可以避免此情况。 首先，避免将新方法声明为与虚方法相同的名称。 其次，可以通过将 `Derived` 的实例强制转换为 `Base` 来使 C# 编译器搜索基类方法列表，从而使其调用虚方法。 由于是虚方法，因此将调用 `Derived` 上的 `DoWork(int)` 的实现。 例如：

```
C#

((Base)d).DoWork(val);  // Calls DoWork(int) on Derived.
```

有关 `new` 和 `override` 的更多示例，请参阅了解何时使用 Override 和 New 关键字。

请参阅

- C# 编程指南
- C# 类型系统
- 方法
- 继承

了解何时使用 Override 和 New 关键字（C# 编程指南）

项目 • 2023/04/07

在 C# 中，派生类中的方法可具有与基类中的方法相同的名称。 可使用 new 和 override 关键字指定方法的交互方式。 `override` 修饰符用于扩展基类 `virtual` 方法，而 `new` 修饰符用于隐藏可访问的基类方法 。 本主题中的示例阐释了这种差异。

在控制台应用程序中，声明以下两个类：`BaseClass` 和 `DerivedClass`。 `DerivedClass` 继承自 `BaseClass`。

C#

```
class BaseClass
{
    public void Method1()
    {
        Console.WriteLine("Base - Method1");
    }
}

class DerivedClass : BaseClass
{
    public void Method2()
    {
        Console.WriteLine("Derived - Method2");
    }
}
```

在 `Main` 方法中，声明变量 `bc`、`dc` 和 `bcdc`。

- `bc` 为 `BaseClass` 类型，其值为 `BaseClass` 类型。
- `dc` 为 `DerivedClass` 类型，其值为 `DerivedClass` 类型。
- `bcdc` 为 `BaseClass` 类型，其值为 `DerivedClass` 类型。 需注意此变量。

由于 `bc` 和 `bcdc` 具有 `BaseClass` 类型，因此它们只能直接访问 `Method1`，除非使用强制转换。 变量 `dc` 可同时访问 `Method1` 和 `Method2`。 下面的代码演示了这些关系。

C#

```
class Program
{
    static void Main(string[] args)
```

```
    {
        BaseClass bc = new BaseClass();
        DerivedClass dc = new DerivedClass();
        BaseClass bcdc = new DerivedClass();

        bc.Method1();
        dc.Method1();
        dc.Method2();
        bcdc.Method1();
    }
    // Output:
    // Base - Method1
    // Base - Method1
    // Derived - Method2
    // Base - Method1
}
```

接着，将以下 `Method2` 方法添加到 `BaseClass`。 此方法的签名与 `DerivedClass` 中 `Method2` 方法的签名匹配。

C#

```
public void Method2()
{
    Console.WriteLine("Base - Method2");
}
```

由于 `BaseClass` 现在具有 `Method2` 方法，因此可以为 `BaseClass` 变量 `bc` 和 `bcdc` 添加第二个调用语句，如下面的代码所示。

C#

```
bc.Method1();
bc.Method2();
dc.Method1();
dc.Method2();
bcdc.Method1();
bcdc.Method2();
```

当生成项目时，你将看到在 `BaseClass` 中添加 `Method2` 方法将引发警告。 警告显示 `DerivedClass` 中的 `Method2` 方法隐藏了 `BaseClass` 中的 `Method2` 方法。 如果希望获得该结果，则建议使用 `Method2` 定义中的 `new` 关键字。 或者，可重命名 `Method2` 方法之一来消除警告，但这始终不实用。

添加 `new` 之前，请运行程序，查看其他调用语句生成的输出。 显示以下结果。

C#

```
// Output:
// Base - Method1
// Base - Method2
// Base - Method1
// Derived - Method2
// Base - Method1
// Base - Method2
```

`new` 关键字可以保留生成该输出的关系，但它会禁止显示警告。 具有 `BaseClass` 类型的变量继续访问 `BaseClass` 的成员，而具有 `DerivedClass` 类型的变量首先继续访问 `DerivedClass` 中的成员，然后再考虑从 `BaseClass` 继承的成员。

若要禁止显示警告，请将 `new` 修饰符添加到 `DerivedClass` 中的 `Method2` 定义，如下面的代码所示。 可在 `public` 前后添加修饰符。

C#

```
public new void Method2()
{
    Console.WriteLine("Derived - Method2");
}
```

再次运行该程序，确认输出未发生更改。 此外，确认不再显示警告。 通过使用 `new`，断言你知道它修饰的成员将隐藏从基类继承的成员。 有关通过继承隐藏名称的详细信息，请参阅 new 修饰符。

若要将此行为与使用 `override` 的效果进行对比，请将以下方法添加到 `DerivedClass`。 可在 `public` 前后添加 `override` 修饰符。

C#

```
public override void Method1()
{
    Console.WriteLine("Derived - Method1");
}
```

将 `virtual` 修饰符添加到 `BaseClass` 中的 `Method1` 定义。 可在 `public` 前后添加 `virtual` 修饰符。

C#

```
public virtual void Method1()
{
    Console.WriteLine("Base - Method1");
}
```

再次运行该项目。 尤其注意以下输出的最后两行。

C#

```
// Output:
// Base - Method1
// Base - Method2
// Derived - Method1
// Derived - Method2
// Derived - Method1
// Base - Method2
```

使用 `override` 修饰符可使 `bcdc` 访问 `DerivedClass` 中定义的 `Method1` 方法。 通常，这是继承层次结构中所需的行为。 让具有从派生类创建的值的对象使用派生类中定义的方法。 可使用 `override` 扩展基类方法实现该行为。

下面的代码包括完整的示例。

C#

```
using System;
using System.Text;

namespace OverrideAndNew
{
    class Program
    {
        static void Main(string[] args)
        {
            BaseClass bc = new BaseClass();
            DerivedClass dc = new DerivedClass();
            BaseClass bcdc = new DerivedClass();

            // The following two calls do what you would expect. They call
            // the methods that are defined in BaseClass.
            bc.Method1();
            bc.Method2();
            // Output:
            // Base - Method1
            // Base - Method2

            // The following two calls do what you would expect. They call
            // the methods that are defined in DerivedClass.
            dc.Method1();
            dc.Method2();
            // Output:
            // Derived - Method1
            // Derived - Method2

            // The following two calls produce different results, depending
            // on whether override (Method1) or new (Method2) is used.
```

```
            bcdc.Method1();
            bcdc.Method2();
            // Output:
            // Derived - Method1
            // Base - Method2
        }
    }

    class BaseClass
    {
        public virtual void Method1()
        {
            Console.WriteLine("Base - Method1");
        }

        public virtual void Method2()
        {
            Console.WriteLine("Base - Method2");
        }
    }

    class DerivedClass : BaseClass
    {
        public override void Method1()
        {
            Console.WriteLine("Derived - Method1");
        }

        public new void Method2()
        {
            Console.WriteLine("Derived - Method2");
        }
    }
}
```

下列阐释了不同上下文中的类似行为。 该示例定义了三个类：一个名为 `Car` 的基类和两个由其派生的 `ConvertibleCar` 和 `Minivan`。 基类中包含 `DescribeCar` 方法。 该方法给出了对一辆车的基本描述，然后调用 `ShowDetails` 提供其他信息。 这三个类中的每一个类都定义了 `ShowDetails` 方法。 `new` 修饰符用于定义 `ConvertibleCar` 类中的 `ShowDetails`。 `override` 修饰符用于定义 `Minivan` 类中的 `ShowDetails`。

C#

```
// Define the base class, Car. The class defines two methods,
// DescribeCar and ShowDetails. DescribeCar calls ShowDetails, and each
derived
// class also defines a ShowDetails method. The example tests which version
of
// ShowDetails is selected, the base class method or the derived class
method.
class Car
```

```
{
    public void DescribeCar()
    {
        System.Console.WriteLine("Four wheels and an engine.");
        ShowDetails();
    }

    public virtual void ShowDetails()
    {
        System.Console.WriteLine("Standard transportation.");
    }
}

// Define the derived classes.

// Class ConvertibleCar uses the new modifier to acknowledge that
ShowDetails
// hides the base class method.
class ConvertibleCar : Car
{
    public new void ShowDetails()
    {
        System.Console.WriteLine("A roof that opens up.");
    }
}

// Class Minivan uses the override modifier to specify that ShowDetails
// extends the base class method.
class Minivan : Car
{
    public override void ShowDetails()
    {
        System.Console.WriteLine("Carries seven people.");
    }
}
```

该示例测试被调用的 `ShowDetails` 版本。 下面的方法 `TestCars1` 为每个类声明了一个实例，并在每个实例上调用 `DescribeCar`。

C#

```
public static void TestCars1()
{
    System.Console.WriteLine("\nTestCars1");
    System.Console.WriteLine("----------");

    Car car1 = new Car();
    car1.DescribeCar();
    System.Console.WriteLine("----------");

    // Notice the output from this test case. The new modifier is
    // used in the definition of ShowDetails in the ConvertibleCar
    // class.
```

```
    ConvertibleCar car2 = new ConvertibleCar();
    car2.DescribeCar();
    System.Console.WriteLine("----------");

    Minivan car3 = new Minivan();
    car3.DescribeCar();
    System.Console.WriteLine("----------");
}
```

`TestCars1` 将生成以下输出。 请特别注意 `car2` 的结果，该结果可能不是你需要的内容。 对象的类型是 `ConvertibleCar`，但 `DescribeCar` 不会访问 `ConvertibleCar` 类中定义的 `ShowDetails` 版本，因为方法已声明包含 `new` 修饰符声明，而不是 `override` 修饰符。 因此， `ConvertibleCar` 对象与 `Car` 对象显示的说明相同。 比较 `car3` 的结果，这是一个 `Minivan` 对象。 在这种情况下，`Minivan` 类中声明的 `ShowDetails` 方法会替代 `Car` 类中声明的 `ShowDetails` 方法，显示的说明描述小型货车。

C#

```
// TestCars1
// ----------
// Four wheels and an engine.
// Standard transportation.
// ----------
// Four wheels and an engine.
// Standard transportation.
// ----------
// Four wheels and an engine.
// Carries seven people.
// ----------
```

`TestCars2` 创建具有 `Car` 类型的对象列表。 对象的值由 `Car` 类、`ConvertibleCar` 类和 `Minivan` 类实例化所得。 对列表中的每个元素调用 `DescribeCar`。 以下代码显示 `TestCars2` 的定义。

C#

```
public static void TestCars2()
{
    System.Console.WriteLine("\nTestCars2");
    System.Console.WriteLine("----------");

    var cars = new List<Car> { new Car(), new ConvertibleCar(),
        new Minivan() };

    foreach (var car in cars)
    {
        car.DescribeCar();
```

```
        System.Console.WriteLine("----------");
    }
}
```

显示以下输出。 请注意，它与 `TestCars1` 显示的输出相同。 不调用 `ConvertibleCar` 类的 `ShowDetails` 方法，不管该对象的类型是 `ConvertibleCar`（在 `TestCars1` 中）还是 `Car`（在 `TestCars2` 中）。 相反，在这两种情况下，`car3` 从 `Minivan` 调用 `ShowDetails` 方法，不管它拥有类型 `Minivan` 还是类型 `Car`。

C#

```
// TestCars2
// ----------
// Four wheels and an engine.
// Standard transportation.
// ----------
// Four wheels and an engine.
// Standard transportation.
// ----------
// Four wheels and an engine.
// Carries seven people.
// ----------
```

方法 `TestCars3` 和方法 `TestCars4` 完成示例。 这些方法直接调用 `ShowDetails`，先从声明具有类型 `ConvertibleCar` 和 `Minivan`（`TestCars3`）的对象开始，然后再转到声明具有类型 `Car`（`TestCars4`）的对象。 以下代码定义了这两种方法。

C#

```
public static void TestCars3()
{
    System.Console.WriteLine("\nTestCars3");
    System.Console.WriteLine("----------");
    ConvertibleCar car2 = new ConvertibleCar();
    Minivan car3 = new Minivan();
    car2.ShowDetails();
    car3.ShowDetails();
}

public static void TestCars4()
{
    System.Console.WriteLine("\nTestCars4");
    System.Console.WriteLine("----------");
    Car car2 = new ConvertibleCar();
    Car car3 = new Minivan();
    car2.ShowDetails();
    car3.ShowDetails();
}
```

这两种方法将产生以下输出，输出对应本主题第一个示例的结果。

C#

```
// TestCars3
// ----------
// A roof that opens up.
// Carries seven people.

// TestCars4
// ----------
// Standard transportation.
// Carries seven people.
```

以下代码显示了完整项目及其输出。

C#

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace OverrideAndNew2
{
    class Program
    {
        static void Main(string[] args)
        {
            // Declare objects of the derived classes and test which version
            // of ShowDetails is run, base or derived.
            TestCars1();

            // Declare objects of the base class, instantiated with the
            // derived classes, and repeat the tests.
            TestCars2();

            // Declare objects of the derived classes and call ShowDetails
            // directly.
            TestCars3();

            // Declare objects of the base class, instantiated with the
            // derived classes, and repeat the tests.
            TestCars4();
        }

        public static void TestCars1()
        {
            System.Console.WriteLine("\nTestCars1");
            System.Console.WriteLine("----------");

            Car car1 = new Car();
```

```
            car1.DescribeCar();
            System.Console.WriteLine("----------");

            // Notice the output from this test case. The new modifier is
            // used in the definition of ShowDetails in the ConvertibleCar
            // class.
            ConvertibleCar car2 = new ConvertibleCar();
            car2.DescribeCar();
            System.Console.WriteLine("----------");

            Minivan car3 = new Minivan();
            car3.DescribeCar();
            System.Console.WriteLine("----------");
        }
        // Output:
        // TestCars1
        // ----------
        // Four wheels and an engine.
        // Standard transportation.
        // ----------
        // Four wheels and an engine.
        // Standard transportation.
        // ----------
        // Four wheels and an engine.
        // Carries seven people.
        // ----------

        public static void TestCars2()
        {
            System.Console.WriteLine("\nTestCars2");
            System.Console.WriteLine("----------");

            var cars = new List<Car> { new Car(), new ConvertibleCar(),
                new Minivan() };

            foreach (var car in cars)
            {
                car.DescribeCar();
                System.Console.WriteLine("----------");
            }
        }
        // Output:
        // TestCars2
        // ----------
        // Four wheels and an engine.
        // Standard transportation.
        // ----------
        // Four wheels and an engine.
        // Standard transportation.
        // ----------
        // Four wheels and an engine.
        // Carries seven people.
        // ----------

        public static void TestCars3()
```

```
        {
            System.Console.WriteLine("\nTestCars3");
            System.Console.WriteLine("----------");
            ConvertibleCar car2 = new ConvertibleCar();
            Minivan car3 = new Minivan();
            car2.ShowDetails();
            car3.ShowDetails();
        }
        // Output:
        // TestCars3
        // ----------
        // A roof that opens up.
        // Carries seven people.

        public static void TestCars4()
        {
            System.Console.WriteLine("\nTestCars4");
            System.Console.WriteLine("----------");
            Car car2 = new ConvertibleCar();
            Car car3 = new Minivan();
            car2.ShowDetails();
            car3.ShowDetails();
        }
        // Output:
        // TestCars4
        // ----------
        // Standard transportation.
        // Carries seven people.
    }

    // Define the base class, Car. The class defines two virtual methods,
    // DescribeCar and ShowDetails. DescribeCar calls ShowDetails, and each
derived
    // class also defines a ShowDetails method. The example tests which
version of
    // ShowDetails is used, the base class method or the derived class
method.
    class Car
    {
        public virtual void DescribeCar()
        {
            System.Console.WriteLine("Four wheels and an engine.");
            ShowDetails();
        }

        public virtual void ShowDetails()
        {
            System.Console.WriteLine("Standard transportation.");
        }
    }

    // Define the derived classes.

    // Class ConvertibleCar uses the new modifier to acknowledge that
ShowDetails
```

```
    // hides the base class method.
    class ConvertibleCar : Car
    {
        public new void ShowDetails()
        {
            System.Console.WriteLine("A roof that opens up.");
        }
    }

    // Class Minivan uses the override modifier to specify that ShowDetails
    // extends the base class method.
    class Minivan : Car
    {
        public override void ShowDetails()
        {
            System.Console.WriteLine("Carries seven people.");
        }
    }

}
```

请参阅

- C# 编程指南
- C# 类型系统
- 使用 Override 和 New 关键字进行版本控制
- base
- abstract

如何替代 ToString 方法（C# 编程指南）

项目 • 2024/03/13

C# 中的每个类或结构都可隐式继承 Object 类。 因此，C# 中的每个对象都会获取 ToString 方法，该方法返回该对象的字符串表示形式。 例如，类型为 `int` 的所有变量都有一个 `ToString` 方法，使它们可以将其内容作为字符串返回：

C#

```
int x = 42;
string strx = x.ToString();
Console.WriteLine(strx);
// Output:
// 42
```

创建自定义类或结构时，应替代 ToString 方法，以向客户端代码提供有关你的类型的信息。

若要深入了解如何通过 `ToString` 方法使用格式字符串和其他类型的自定义格式设置，请参阅格式化类型。

> **ⓘ 重要**
>
> 决定通过此方法提供信息内容时，请考虑你的类或结构是否会被不受信任的代码使用。 请务必确保不提供可能被恶意代码利用的任何信息。

替代类或结构中的 `ToString` 方法：

1. 声明具有下列修饰符和返回类型的 `ToString` 方法：

   C#

   ```
   public override string ToString(){}
   ```

2. 实现该方法，使其返回一个字符串。

   下面的示例返回类的名称，但特定于该类的特定实例的数据除外。

   C#

   ```
   class Person
   {
       public string Name { get; set; }
   ```

```
    public int Age { get; set; }

    public override string ToString()
    {
        return "Person: " + Name + " " + Age;
    }
}
```

可以测试 `ToString` 方法，如以下代码示例所示：

C#

```
Person person = new Person { Name = "John", Age = 12 };
Console.WriteLine(person);
// Output:
// Person: John 12
```

另请参阅

- IFormattable
- C# 类型系统
- 字符串
- string
- override
- virtual
- 格式设置类型

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

成员（C# 编程指南）

项目 • 2024/03/13

类和结构具有表示其数据和行为的成员。 类的成员包括在类中声明的所有成员，以及在该类的继承层次结构中的所有类中声明的所有成员（构造函数和析构函数除外）。 基类中的私有成员被继承，但不能从派生类访问。

下表列出类或结构中可包含的成员类型：

展开表

成员	描述
字段	字段是在类范围声明的变量。 字段可以是内置数值类型或其他类的实例。 例如，日历类可能具有一个包含当前日期的字段。
常量	常量是在编译时设置其值并且不能更改其值的字段。
属性	属性是类中可以像类中的字段一样访问的方法。 属性可以为类字段提供保护，以避免字段在对象不知道的情况下被更改。
方法	方法定义类可以执行的操作。 方法可接受提供输入数据的参数，并可通过参数返回输出数据。 方法还可以不使用参数而直接返回值。
事件	事件向其他对象提供有关发生的事情（如单击按钮或成功完成某个方法）的通知。 事件是使用委托定义和触发的。
运算符	重载运算符被视为类型成员。 重载运算符时，将其定义为类型中的公共静态方法。 有关详细信息，请参阅运算符重载。
索引器	使用索引器可以用类似于数组的方式为对象建立索引。
构造函数	构造函数是首次创建对象时调用的方法。 它们通常用于初始化对象的数据。
终结器	C# 中很少使用终结器。 终结器是当对象即将从内存中移除时由运行时执行引擎调用的方法。 它们通常用来确保任何必须释放的资源都得到适当的处理。
嵌套类型	嵌套类型是在其他类型中声明的类型。 嵌套类型通常用于描述仅由包含它们的类型使用的对象。

另请参阅

- 类

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

抽象类、密封类及类成员（C# 编程指南）

项目 • 2024/03/13

使用 abstract 关键字可以创建不完整且必须在派生类中实现的类和 class 成员。

使用 sealed 关键字可以防止继承以前标记为 virtual 的类或某些类成员。

抽象类和类成员

通过在类定义前面放置关键字 `abstract`，可以将类声明为抽象类。 例如：

C#

```
public abstract class A
{
    // Class members here.
}
```

抽象类不能实例化。 抽象类的用途是提供一个可供多个派生类共享的通用基类定义。 例如，类库可以定义一个抽象类，将其用作多个类库函数的参数，并要求使用该库的程序员通过创建派生类来提供自己的类实现。

抽象类也可以定义抽象方法。 方法是将关键字 `abstract` 添加到方法的返回类型的前面。 例如：

C#

```
public abstract class A
{
    public abstract void DoWork(int i);
}
```

抽象方法没有实现，所以方法定义后面是分号，而不是常规的方法块。 抽象类的派生类必须实现所有抽象方法。 当抽象类从基类继承虚方法时，抽象类可以使用抽象方法重写该虚方法。 例如：

C#

```
// compile with: -target:library
public class D
{
    public virtual void DoWork(int i)
```

```
    {
        // Original implementation.
    }
}

public abstract class E : D
{
    public abstract override void DoWork(int i);
}

public class F : E
{
    public override void DoWork(int i)
    {
        // New implementation.
    }
}
```

如果将 `virtual` 方法声明为 `abstract`，则该方法对于从抽象类继承的所有类而言仍然是虚方法。 继承抽象方法的类无法访问方法的原始实现，因此在上一示例中，类 F 上的 `DoWork` 无法调用类 D 上的 `DoWork`。通过这种方式，抽象类可强制派生类向虚拟方法提供新的方法实现。

密封类和类成员

通过在类定义前面放置关键字 `sealed`，可以将类声明为密封类。 例如：

C#

```
public sealed class D
{
    // Class members here.
}
```

密封类不能用作基类。 因此，它也不能是抽象类。 密封类禁止派生。 由于密封类从不用作基类，所以有些运行时优化可以略微提高密封类成员的调用速度。

在对基类的虚成员进行重写的派生类上，方法、索引器、属性或事件可以将该成员声明为密封成员。 在用于以后的派生类时，这将取消成员的虚效果。 方法是在类成员声明中将 `sealed` 关键字置于 override 关键字前面。 例如：

C#

```
public class D : C
{
```

```
    public sealed override void DoWork() { }
}
```

另请参阅

- C# 类型系统
- 继承
- 方法
- 字段
- 如何定义抽象属性

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

静态类和静态类成员（C# 编程指南）

项目 • 2024/03/20

静态类基本上与非静态类相同，但存在一个差别：静态类无法实例化。 换句话说，无法使用 new 运算符创建类类型的变量。 由于不存在任何实例变量，因此可以使用类名本身访问静态类的成员。 例如，如果你具有一个静态类，该类名为 `UtilityClass`，并且具有一个名为 `MethodA` 的公共静态方法，如下面的示例所示：

C#

```
UtilityClass.MethodA();
```

静态类可以用作只对输入参数进行操作并且不必获取或设置任何内部实例字段的方法集的方便容器。 例如，在 .NET 类库中，静态 System.Math 类包含执行数学运算，而无需存储或检索对 Math 类特定实例唯一的数据的方法。 即，通过指定类名和方法名称来应用类的成员，如下面的示例所示。

C#

```
double dub = -3.14;
Console.WriteLine(Math.Abs(dub));
Console.WriteLine(Math.Floor(dub));
Console.WriteLine(Math.Round(Math.Abs(dub)));

// Output:
// 3.14
// -4
// 3
```

与所有类类型的情况一样，加载引用静态类的程序时，.NET 运行时会加载该类的类型信息。 程序无法确切指定类被加载的时间。 但是，可保证它会加载，并在程序中首次引用类之前初始化其字段并调用其静态构造函数。 静态构造函数只调用一次，在程序所驻留的应用程序域的生存期内，静态类会保留在内存中。

> **备注**
>
> 若要创建仅允许创建本身的一个实例的非静态类，请参阅**在 C# 中实现单一实例**。

以下列表提供静态类的主要功能：

- 只包含静态成员。
- 无法实例化。

- 会进行密封。
- 不能包含实例构造函数。

因此，创建静态类基本上与创建只包含静态成员和私有构造函数的类相同。 私有构造函数可防止类进行实例化。 使用静态类的优点是编译器可以进行检查，以确保不会意外地添加任何实例成员。 编译器会保证此类的实例无法创建。

静态类会进行密封，因此不能继承。 它们不能继承任何类或接口（除了 Object）。 静态类不能包含实例构造函数。 但是，它们可以包含静态构造函数。 如果非静态类包含了需要进行有意义的初始化的静态成员，则它也应该定义一个静态构造器。 有关详细信息，请参阅静态构造函数。

示例

下面是静态类的一个示例，它包含将温度从摄氏度转换为华氏度以及从华氏度转换为摄氏度的两个方法：

C#

```
public static class TemperatureConverter
{
    public static double CelsiusToFahrenheit(string temperatureCelsius)
    {
        // Convert argument to double for calculations.
        double celsius = Double.Parse(temperatureCelsius);

        // Convert Celsius to Fahrenheit.
        double fahrenheit = (celsius * 9 / 5) + 32;

        return fahrenheit;
    }

    public static double FahrenheitToCelsius(string temperatureFahrenheit)
    {
        // Convert argument to double for calculations.
        double fahrenheit = Double.Parse(temperatureFahrenheit);

        // Convert Fahrenheit to Celsius.
        double celsius = (fahrenheit - 32) * 5 / 9;

        return celsius;
    }
}

class TestTemperatureConverter
{
    static void Main()
    {
```

```
        Console.WriteLine("Please select the convertor direction");
        Console.WriteLine("1. From Celsius to Fahrenheit.");
        Console.WriteLine("2. From Fahrenheit to Celsius.");
        Console.Write(":");

        string? selection = Console.ReadLine();
        double F, C = 0;

        switch (selection)
        {
            case "1":
                Console.Write("Please enter the Celsius temperature: ");
                F =
TemperatureConverter.CelsiusToFahrenheit(Console.ReadLine() ?? "0");
                Console.WriteLine("Temperature in Fahrenheit: {0:F2}", F);
                break;

            case "2":
                Console.Write("Please enter the Fahrenheit temperature: ");
                C =
TemperatureConverter.FahrenheitToCelsius(Console.ReadLine() ?? "0");
                Console.WriteLine("Temperature in Celsius: {0:F2}", C);
                break;

            default:
                Console.WriteLine("Please select a convertor.");
                break;
        }

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
/* Example Output:
    Please select the convertor direction
    1. From Celsius to Fahrenheit.
    2. From Fahrenheit to Celsius.
    :2
    Please enter the Fahrenheit temperature: 20
    Temperature in Celsius: -6.67
    Press any key to exit.
 */
```

静态成员

非静态类可以包含静态方法、字段、属性或事件。 即使不存在类的任何实例，也可对类调用静态成员。 静态成员始终按类名（而不是实例名称）进行访问。 静态成员只有一个副本存在（与创建的类的实例数无关）。 静态方法和属性无法在其包含类型中访问非静态字段和事件，它们无法访问任何对象的实例变量，除非在方法参数中显式传递它。

更典型的做法是声明具有一些静态成员的非静态类（而不是将整个类都声明为静态）。 静态字段的两个常见用途是保留实例化的对象数的计数，或是存储必须在所有实例间共享的值。

静态方法可以进行重载，但不能进行替代，因为它们属于类，而不属于类的任何实例。

虽然字段不能声明为 `static const`，const 字段在其行为方面本质上是静态的。 它属于类型，而不属于类型的实例。 因此，可以使用用于静态字段的相同 `ClassName.MemberName` 表示法来访问 `const` 字段。 无需进行对象实例化。

C# 不支持静态局部变量（即在方法范围中声明的变量）。

可在成员的返回类型之前使用 `static` 关键字声明静态类成员，如下面的示例所示：

C#

```
public class Automobile
{
    public static int NumberOfWheels = 4;

    public static int SizeOfGasTank
    {
        get
        {
            return 15;
        }
    }

    public static void Drive() { }

    public static event EventType? RunOutOfGas;

    // Other non-static fields and properties...
}
```

在首次访问静态成员之前以及在调用构造函数（如果有）之前，会初始化静态成员。 若要访问静态类成员，请使用类的名称（而不是变量名称）指定成员的位置，如下面的示例所示：

C#

```
Automobile.Drive();
int i = Automobile.NumberOfWheels;
```

如果类包含静态字段，则提供在类加载时初始化它们的静态构造函数。

对静态方法的调用会采用 Microsoft 中间语言 (MSIL) 生成调用指令，而对实例方法的调用会生成 `callvirt` 指令，该指令还会检查是否存在 null 对象引用。 但是在大多数时候，两者之间的性能差异并不显著。

C# 语言规范

有关详细信息，请参阅 C# 语言规范中的静态类、静态和实例成员以及静态构造函数。 该语言规范是 C# 语法和用法的权威资料。

另请参阅

- static
- 类
- class
- 静态构造函数
- 实例构造函数

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

.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

访问修饰符（C# 编程指南）

项目 • 2024/11/21

所有类型和类型成员都具有可访问性级别。 该级别可以控制是否可以从你的程序集或其他程序集中的其他代码中使用它们。 程序集是通过在单个编译中编译一个或多个 *.cs* 文件而创建的 *.dll* 或 *.exe*。 可以使用以下访问修饰符在进行声明时指定类型或成员的可访问性：

- 公共：任何程序集中的代码都可以访问此类型或成员。 包含类型的可访问性级别控制该类型的公共成员的可访问性级别。
- 专用：只有在同一 `class` 或 `struct` 中声明的代码才能访问此成员。
- 受保护：只有同一 `class` 或派生的 `class` 中的代码才能访问此类型或成员。
- 内部：只有同一程序集中的代码才能访问此类型或成员。
- 受保护的内部：只有同一程序集中的代码或另一个程序集的派生类中的代码才能访问此类型或成员。
- 专用受保护：只有同一程序集和同一类/派生类中的代码才能访问该类型或成员。
- 文件：只有同一文件中的代码可以访问类型或成员。

类型上的 record 修饰符会导致编译器合成额外的成员。 `record` 修饰符不会影响 `record class` 或 `record struct` 的默认可访问性。

摘要表

展开表

调用方的位置	`public`	`protected internal`	`protected`	`internal`	`private protected`	`private`	`file`
在文件中	✔	✔	✔	✔	✔	✔	✔
在类内	✔	✔	✔	✔	✔	✔	✘
派生类（相同程序集）	✔	✔	✔	✔	✔	✘	✘
非派生类（相同程序集）	✔	✔	✘	✔	✘	✘	✘
派生类（不同程序集）	✔	✔	✔	✘	✘	✘	✘
非派生类（不同程序	✔	✘	✘	✘	✘	✘	✘

调用方的位置	`public`	`protected internal`	`protected`	`internal`	`private protected`	`private`	`file`
集)							

下面的示例演示如何在类型和成员上指定访问修饰符：

```
C#

public class Bicycle
{
    public void Pedal() { }
}
```

不是所有访问修饰符都可以在所有上下文中由所有类型或成员使用。 在某些情况下，包含类型的可访问性会限制其成员的可访问性。

分部类或分部成员的多个声明必须具有相同的可访问性。 如果分部类或成员的一个声明不包含访问修饰符，则其他声明不能声明访问修饰符。 如果分部类或分部方法的多个声明声明了不同的可访问性，编译器会生成错误。

类和结构可访问性

直接在命名空间中声明的类和结构（没有嵌套在其他类或结构中）可以具有 `public`、`internal` 或 `file`。 如果未指定任何访问修饰符，则默认设置为 `internal`。

结构成员（包括嵌套的类和结构）可以声明为 `public`、`internal` 或 `private`。 类成员（包括嵌套的类和结构）可以声明为 `public`、`protected internal`、`protected`、`internal`、`private protected` 或 `private`。 默认情况下，类成员和结构成员（包括嵌套的类和结构）的访问级别为 `private`。

派生类不能具有高于其基类型的可访问性。 不能声明派生自内部类 `A` 的公共类 `B`。 如果允许这样，则它将具有使 `A` 公开的效果，因为可从派生类访问 `A` 的所有 `protected` 或 `internal` 成员。

可以通过使用 `InternalsVisibleToAttribute` 启用特定的其他程序集访问内部类型。 有关详细信息，请参阅友元程序集。

其他类型

在命名空间内直接声明的接口可以声明为 `public` 或 `internal`，就像类和结构一样，接口默认设置为 `internal` 访问级别。 接口成员默认为 `public`，因为接口的用途是启用其

他类型以访问类或结构。 接口成员声明可以包含任何访问修饰符。 在 `interface` 成员上使用访问修饰符来提供接口的所有实现者所需的通用实现。

默认情况下，在命名空间中声明的 delegate 类型具有 `internal` 访问权限。

有关访问修饰符的详细信息，请参阅可访问性级别页面。

成员可访问性

可以使用六种访问类型中的任意一种来声明 `class` 或 `struct` 的成员（包括嵌套的类和结构）。 结构成员无法声明为 `protected`、`protected internal` 或 `private protected`，因为结构不支持继承。

通常情况下，成员的可访问性不大于包含该成员的类型的可访问性。 但是，如果 `internal` 类的 `public` 成员实现了接口方法或替代了在公共基类中定义的虚拟方法，则可从该程序集的外部访问该成员。

为字段、属性或事件的任何成员的类型必须至少与该成员本身具有相同的可访问性。 同样，任何方法、索引器或委托的返回类型和参数类型必须至少与该成员本身具有相同的可访问性。 例如，除非 `C` 也是 `public`，否则不能具有返回类 `C` 的 `public` 方法 `M`。 同样，如果 `A` 声明为 `private`，则不能具有类型 `A` 的 `protected` 属性。

用户定义的运算符始终必须声明为 `public` 和 `static`。 有关详细信息，请参阅运算符重载。

若要设置 `class` 或 `struct` 成员的访问级别，请向成员声明添加适当的关键字，如以下示例中所示。

C#

```
// public class:
public class Tricycle
{
    // protected method:
    protected void Pedal() { }

    // private field:
    private int _wheels = 3;

    // protected internal property:
    protected internal int Wheels
    {
        get { return _wheels; }
    }
}
```

终结器不能具有可访问性修饰符。 `enum` 类型的成员始终为 `public`，并且不能应用访问修饰符。

仅允许在顶级（非嵌套）类型声明中使用 `file` 访问修饰符。

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

- 指定修饰符顺序（样式规则 IDE0036）
- C# 类型系统
- 接口
- 可访问性级别
- private
- public
- internal
- 受保护
- protected internal
- private protected
- sealed
- class
- struct
- interface
- 匿名类型

字段（C# 编程指南）

项目 • 2023/06/01

字段是在类或结构中直接声明的任意类型的变量。 字段是其包含类型的成员。

类或结构可能具有实例字段和/或静态字段。 实例字段特定于类型的实例。 如果你有包含实例字段 `F` 的类 `T`，则可以创建两个类型为 `T` 的对象并修改每个对象中 `F` 的值，而不会影响另一个对象中的值。 与此相比，静态字段属于类型本身，并在该类型的所有实例之间共享。 只能使用类型名称访问静态字段。 如果按实例名称访问静态字段，将出现 CS0176 编译时错误。

通常，应为字段声明 `private` 或 `protected` 可访问性。 类型向客户端代码公开的数据应通过方法、属性和索引器提供。 通过使用这些构造间接访问内部字段，可以防止出现无效的输入值。 存储由公共属性公开的数据的私有字段称为后备存储或支持字段。 可以声明 `public` 字段，但随后无法阻止使用你的类型的代码将该字段设置为无效值，或者以其他方式更改对象的数据。

字段通常存储必须对多个类型方法可访问且存储时间必须长于任何单个方法的生存期的数据。 例如，表示日历日期的类型可能具有三个整数字段：一个用于月、一个用于日、一个用于年。 不在单个方法作用域外使用的变量应声明为方法主体本身中的局部变量。

字段是通过指定访问级别在类或结构块中声明的，其后跟类型，再跟字段的名称。 例如：

C#

```
public class CalendarEntry
{

    // private field (Located near wrapping "Date" property).
    private DateTime _date;

    // Public property exposes _date field safely.
    public DateTime Date
    {
        get
        {
            return _date;
        }
        set
        {
            // Set some reasonable boundaries for likely birth dates.
            if (value.Year > 1900 && value.Year <= DateTime.Today.Year)
            {
                _date = value;
            }
            else
```

```
            {
                throw new ArgumentOutOfRangeException("Date");
            }
        }
    }

    // public field (Generally not recommended).
    public string? Day;

    // Public method also exposes _date field safely.
    // Example call: birthday.SetDate("1975, 6, 30");
    public void SetDate(string dateString)
    {
        DateTime dt = Convert.ToDateTime(dateString);

        // Set some reasonable boundaries for likely birth dates.
        if (dt.Year > 1900 && dt.Year <= DateTime.Today.Year)
        {
            _date = dt;
        }
        else
        {
            throw new ArgumentOutOfRangeException("dateString");
        }
    }

    public TimeSpan GetTimeSpan(string dateString)
    {
        DateTime dt = Convert.ToDateTime(dateString);

        if (dt.Ticks < _date.Ticks)
        {
            return _date - dt;
        }
        else
        {
            throw new ArgumentOutOfRangeException("dateString");
        }
    }
}
```

若要访问实例中的字段，请在实例名称后添加一个句点，后跟字段的名称，如 `instancename._fieldName` 中所示。 例如：

C#

```
CalendarEntry birthday = new CalendarEntry();
birthday.Day = "Saturday";
```

声明字段时，可以使用赋值运算符为字段指定一个初始值。 例如，若要为 `Day` 字段自动赋值 `"Monday"`，则需要声明 `Day`，如以下示例所示：

C#

```
public class CalendarDateWithInitialization
{
    public string Day = "Monday";
    //...
}
```

字段会在对象实例的构造函数被调用之前即刻初始化。 如果构造函数分配了字段的值，则它将覆盖在字段声明期间给定的任何值。 有关详细信息，请参阅使用构造函数。

> **备注**
>
> 字段初始化表达式不能引用其他实例字段。

可以将字段标记为 public、private、protected、internal、protected internal 或 private protected。 这些访问修饰符定义该类型的用户访问该字段的方式。 有关详细信息，请参阅访问修饰符。

可以选择性地将字段声明为static。 静态字段可供调用方在任何时候进行调用，即使不存在任何类型的实例。 有关详细信息，请参阅静态类和静态类成员。

可以将字段声明为 readonly。 只能在初始化期间或在构造函数中为只读字段赋值。 `static readonly` 字段类似于常量，只不过 C# 编译器在编译时不具有对静态只读字段的值的访问权限，而只有在运行时才具有访问权限。 有关详细信息，请参阅常量。

可以将字段声明为 required。 必填字段必须由构造函数初始化，或者在创建对象时由对象初始值设定项初始化。 将 System.Diagnostics.CodeAnalysis.SetsRequiredMembersAttribute 特性添加到初始化所有必需成员的任何构造函数声明中。

不能在同一字段上组合 `required` 修饰符和 `readonly` 修饰符。 但是，属性只能是 `required` 和 `init`。

从 C# 12 开始，主构造函数参数是声明字段的替代方法。 如果类型具有必须在初始化时提供的依赖项，则可以创建提供这些依赖项的主构造函数。 可以捕获这些参数并将其用于代替类型中声明的字段。 对于 record 类型，主构造函数参数显示为公共属性。

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# 编程指南
- C# 类型系统
- 使用构造函数
- 继承
- 访问修饰符
- 抽象类、密封类及类成员

常量（C# 编程指南）

项目 • 2023/11/20

常量是不可变的值，在编译时是已知的，在程序的生命周期内不会改变。 常量使用 const 修饰符声明。 只有 C# 内置类型可声明为 `const`。 除 String 以外的引用类型常量只能使用 null 值进行初始化。 用户定义的类型（包括类、结构和数组）不能为 `const`。 使用 readonly 修饰符创建在运行时一次性（例如在构造函数中）初始化的类、结构或数组，此后不能更改。

C# 不支持 `const` 方法、属性或事件。

枚举类型使你能够为整数内置类型定义命名常量（例如 `int`、`uint`、`long` 等）。 有关详细信息，请参阅枚举。

常量在声明时必须初始化。 例如：

C#

```
class Calendar1
{
    public const int Months = 12;
}
```

在此示例中，常量 `Months` 始终为 12，即使类本身也无法更改它。 实际上，当编译器遇到 C# 源代码中的常量标识符（例如，`Months`）时，它直接将文本值替换到它生成的中间语言 (IL) 代码中。 因为运行时没有与常量相关联的变量地址，所以 `const` 字段不能通过引用传递，并且不能在表达式中显示为左值。

> **备注**
>
> 引用其他代码（如 DLL）中定义的常量值时要格外小心。 如果新版本的 DLL 定义了新的常量值，则程序仍将保留旧的文本值，直到根据新版本重新编译。

可以同时声明多个同一类型的常量，例如：

C#

```
class Calendar2
{
    public const int Months = 12, Weeks = 52, Days = 365;
}
```

如果不创建循环引用，则用于初始化常量的表达式可以引用另一个常量。 例如：

C#

```
class Calendar3
{
    public const int Months = 12;
    public const int Weeks = 52;
    public const int Days = 365;

    public const double DaysPerWeek = (double) Days / (double) Weeks;
    public const double DaysPerMonth = (double) Days / (double) Months;
}
```

可以将常量标记为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常量的方式。 有关详细信息，请参阅访问修饰符。

常量是作为静态字段访问的，因为常量的值对于该类型的所有实例都是相同的。 不使用 `static` 关键字来声明这些常量。 不在定义常量的类中的表达式必须使用类名、句点和常量名称来访问该常量。 例如：

C#

```
int birthstones = Calendar.Months;
```

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# 编程指南
- 属性
- 类型
- readonly
- C# 不可变性（第一部分）：不可变性种类

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 feedback

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

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

如何定义抽象属性（C# 编程指南）

项目 • 2024/03/13

以下示例演示如何定义抽象属性。 抽象属性声明不提供属性访问器的实现，它声明该类支持属性，而将访问器实现留给派生类。 以下示例演示如何实现从基类继承抽象属性。

此示例由三个文件组成，其中每个文件都单独编译，产生的程序集由下一次编译引用：

- abstractshape.cs：包含抽象 `Area` 属性的 `Shape` 类。
- shapes.cs：`Shape` 类的子类。
- shapetest.cs：测试程序，用于显示一些 `Shape` 派生对象的区域。

若要编译该示例，请使用以下命令：

```
csc abstractshape.cs shapes.cs shapetest.cs
```

这将创建可执行文件 shapetest.exe。

示例

此文件声明 `Shape` 类，该类包含 `double` 类型的 `Area` 属性。

C#

```
// compile with: csc -target:library abstractshape.cs
public abstract class Shape
{
    private string name;

    public Shape(string s)
    {
        // calling the set accessor of the Id property.
        Id = s;
    }

    public string Id
    {
        get
        {
            return name;
        }

        set
        {
            name = value;
        }
```

```
    }

    // Area is a read-only property - only a get accessor is needed:
    public abstract double Area
    {
        get;
    }

    public override string ToString()
    {
        return $"{Id} Area = {Area:F2}";
    }
}
```

- 属性的修饰符放置在属性声明中。 例如：

  ```
  C#

  public abstract double Area
  ```

- 声明抽象属性时（如本示例中的 `Area`），只需指明哪些属性访问器可用即可，不要实现它们。 在此示例中，仅 get 访问器可用，因此该属性是只读属性。

下面的代码演示 `Shape` 的三个子类，并演示它们如何替代 `Area` 属性来提供自己的实现。

```
C#

// compile with: csc -target:library -reference:abstractshape.dll shapes.cs
public class Square : Shape
{
    private int side;

    public Square(int side, string id)
        : base(id)
    {
        this.side = side;
    }

    public override double Area
    {
        get
        {
            // Given the side, return the area of a square:
            return side * side;
        }
    }
}

public class Circle : Shape
{
    private int radius;
```

```
    public Circle(int radius, string id)
        : base(id)
    {
        this.radius = radius;
    }

    public override double Area
    {
        get
        {
            // Given the radius, return the area of a circle:
            return radius * radius * System.Math.PI;
        }
    }
}

public class Rectangle : Shape
{
    private int width;
    private int height;

    public Rectangle(int width, int height, string id)
        : base(id)
    {
        this.width = width;
        this.height = height;
    }

    public override double Area
    {
        get
        {
            // Given the width and height, return the area of a rectangle:
            return width * height;
        }
    }
}
```

下面的代码演示一个创建若干 `Shape` 派生对象并输出其区域的测试程序。

C#

```
// compile with: csc -reference:abstractshape.dll;shapes.dll shapetest.cs
class TestClass
{
    static void Main()
    {
        Shape[] shapes =
        {
            new Square(5, "Square #1"),
            new Circle(3, "Circle #1"),
            new Rectangle( 4, 5, "Rectangle #1")
```

```
        };

        System.Console.WriteLine("Shapes Collection");
        foreach (Shape s in shapes)
        {
            System.Console.WriteLine(s);
        }
    }
}
/* Output:
    Shapes Collection
    Square #1 Area = 25.00
    Circle #1 Area = 28.27
    Rectangle #1 Area = 20.00
*/
```

另请参阅

- C# 类型系统
- 抽象类、密封类及类成员
- 属性

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

如何在 C# 中定义常量

项目 • 2023/04/07

常量是在编译时设置其值并且永远不能更改其值的字段。 使用常量可以为特殊值提供有意义的名称，而不是数字文本（“幻数”）。

> **备注**
>
> 在 C# 中，不能以 C 和 C++ 中通常采用的方式使用 #define 预处理器指令定义常量。

若要定义整型类型（`int`、`byte` 等）的常量值，请使用枚举类型。 有关详细信息，请参阅枚举。

若要定义非整型常量，一种方法是将它们分组到一个名为 `Constants` 的静态类。 这要求对常量的所有引用都在其前面加上该类名，如下例所示。

示例

C#

```
static class Constants
{
    public const double Pi = 3.14159;
    public const int SpeedOfLight = 300000; // km per sec.
}

class Program
{
    static void Main()
    {
        double radius = 5.3;
        double area = Constants.Pi * (radius * radius);
        int secsFromSun = 149476000 / Constants.SpeedOfLight; // in km
        Console.WriteLine(secsFromSun);
    }
}
```

使用类名限定符有助于确保你和使用该常量的其他人了解它是常量并且不能修改。

请参阅

- C# 类型系统

属性（C# 编程指南）

项目 • 2024/11/14

*属性*是一种成员，它提供灵活的机制来读取、写入或计算数据字段的值。 属性显示为公共数据成员，但它们作为名为*访问器*的特殊方法来实现。 此功能使调用方能够轻松访问数据，还有助于提高数据的安全性和灵活性。 属性语法是字段的自然延伸。 字段定义存储位置：

C#

```
public class Person
{
    public string? FirstName;

    // Omitted for brevity.
}
```

自动实现的属性

属性定义包含 `get` 和 `set` 访问器的声明，这两个访问器用于检索该属性的值以及对其赋值：

C#

```
public class Person
{
    public string? FirstName { get; set; }

    // Omitted for brevity.
}
```

前面的示例显示了自动 *实现的属性*。 此编译器会为该属性生成一个隐藏的支持字段。 编译器还实现 `get` 和 `set` 访问器的正文。 任何属性都应用于自动实现的属性。 你可以通过在属性上指定 `field:` 标记，将属性应用于编译器生成的支持字段。

你可以通过在属性的右大括号后面设置一个值，将属性初始化为默认值以外的其他值。 对于 `FirstName` 属性的的初始值，你可能更希望设置为空字符串而非 `null`。 你可以指定该空字符串，如以下代码所示：

C#

```
public class Person
{
```

```
    public string FirstName { get; set; } = string.Empty;

    // Omitted for brevity.
}
```

字段支持的属性

在 C# 13 中，可以使用关键字预览功能在访问器 field 中添加验证或其他逻辑。 关键字 `field` 访问属性的编译器合成后盾字段。 它使你可以编写属性访问器，而无需显式声明单独的后盾字段。

C#

```
public class Person
{
    public string? FirstName
    {
        get;
        set => field = value.Trim();
    }

    // Omitted for brevity.
}
```

> **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。

必需的属性

前面的示例允许调用方使用默认构造函数创建 `Person`，而无需设置 `FirstName` 属性。 属性已将类型更改为*可以为 null* 的字符串。 从 C# 11 开始，你可以*要求*调用方设置属性：

C#

```
public class Person
{
```

```
    public Person() { }

    [SetsRequiredMembers]
    public Person(string firstName) => FirstName = firstName;

    public required string FirstName { get; init; }

    // Omitted for brevity.
}
```

上述代码对 `Person` 类进行了两项更改。 首先，`FirstName` 属性声明包含了 `required` 修饰符。 这意味着任何创建新 `Person` 的代码都必须使用对象初始值设定项来设置此属性。 其次，采用 `firstName` 参数的构造函数具有 System.Diagnostics.CodeAnalysis.SetsRequiredMembersAttribute 特性。 此特性通知编译器此构造函数设置了*所有所有* `required` 成员。 使用此构造函数的调用方不需要使用对象初始值设定项来设置 `required` 属性。

> **ⓘ 重要**
>
> 不要将 `required` 与*不可为 null* 混淆。 将 `required` 属性设置为 `null` 或 `default` 是有效的。 如果类型不可为 null，例如这些示例中的 `string`，则编译器会发出警告。

C#

```
var aPerson = new Person("John");
aPerson = new Person{ FirstName = "John"};
// Error CS9035: Required member `Person.FirstName` must be set:
//aPerson2 = new Person();
```

表达式主体定义

属性访问器通常由单行语句组成。 访问器将分配或返回表达式的结果。 可以将这些属性作为 expression-bodied 成员来实现。 `=>` 令牌后跟用于为属性赋值或从属性中检索值的表达式，即组成了表达式主体定义。

只读属性可以将 `get` 访问器作为 expression-bodied 成员实现。 下面的示例将只读 `Name` 属性作为 expression-bodied 成员实现：

C#

```
public class Person
{
    public Person() { }
```

```
    [SetsRequiredMembers]
    public Person(string firstName, string lastName)
    {
        FirstName = firstName;
        LastName = lastName;
    }

    public required string FirstName { get; init; }
    public required string LastName { get; init; }

    public string Name => $"{FirstName} {LastName}";

    // Omitted for brevity.
}
```

`Name` 属性为计算属性。 `Name` 没有支持字段。 该属性每次都会计算它。

访问控制

前面的示例显示了读/写属性。 你还可以创建只读属性，或者对 set 和 get 访问器提供不同的可访问性。 假设类 `Person` 应仅启用从类中的其他方法更改属性的值 `FirstName` 。 你可以为 set 访问器`private`提供辅助功能，而不是`internal` `public`：

C#

```
public class Person
{
    public string? FirstName { get; private set; }

    // Omitted for brevity.
}
```

可以从任意代码读取 `FirstName` 属性，但只能从 `Person` 类中的代码对其赋值。

可以向 set 和 get 访问器添加任何严格访问修饰符。 单个访问器上的访问修饰符必须比属性的访问更严格。 上述代码是合法的，因为 `FirstName` 属性为 `public`，但 set 访问器为 `private`。 不能声明具有 `public` 访问器的 `private` 属性。 属性声明还可以声明为 `protected`、 `internal`、 `protected internal`，甚至 `private`。

`set` 访问器有两个特殊的访问修饰符：

- `set` 访问器可以将 `init` 作为其访问修饰符。 只能从对象初始值设定项或类型的构造函数调用该 `set` 访问器。 它比 `set` 访问器上的 `private` 更严格。
- 自动实现的属性可以在不使用访问器的情况下`set`声明`get`访问器。 在这种情况下，编译器只允许从类型的构造函数调用 `set` 访问器。 它比 `set` 访问器上的 `init`

访问器更严格。

请按照这种方式修改 `Person` 类，如下所示：

```C#
public class Person
{
    public Person(string firstName) => FirstName = firstName;

    public string FirstName { get; }

    // Omitted for brevity.
}
```

前面的示例要求调用方使用包含 `FirstName` 参数的构造函数。 调用方无法使用对象初始值设定项向属性分配值。 若要支持初始值设定项，可以将 `set` 访问器设置为 `init` 访问器，如以下代码所示：

```C#
public class Person
{
    public Person() { }
    public Person(string firstName) => FirstName = firstName;

    public string? FirstName { get; init; }

    // Omitted for brevity.
}
```

这些修饰符通常与 `required` 修饰符配合合适，以强制进行适当的初始化。

具有支持字段的属性

可以将计算属性和存储的私有字段混合起来，创建*缓存的计算属性*。 例如，更新 `FullName` 属性，以便在第一次访问时进行字符串格式设置：

```C#
public class Person
{
    public Person() { }

    [SetsRequiredMembers]
    public Person(string firstName, string lastName)
    {
```

```
        FirstName = firstName;
        LastName = lastName;
    }

    public required string FirstName { get; init; }
    public required string LastName { get; init; }

    private string? _fullName;
    public string FullName
    {
        get
        {
            if (_fullName is null)
                _fullName = $"{FirstName} {LastName}";
            return _fullName;
        }
    }
}
```

此实现可正常工作，因为 `FirstName` 和 `LastName` 属性为只读属性。 用户可以更改其名字。 若要更新 `FirstName` 和 `LastName` 属性以允许使用 `set` 访问器，则需要使 `fullName` 的任何缓存值失效。 修改 `FirstName` 和 `LastName` 属性的 `set` 访问器，以便重新计算 `fullName` 字段：

C#

```
public class Person
{
    private string? _firstName;
    public string? FirstName
    {
        get => _firstName;
        set
        {
            _firstName = value;
            _fullName = null;
        }
    }

    private string? _lastName;
    public string? LastName
    {
        get => _lastName;
        set
        {
            _lastName = value;
            _fullName = null;
        }
    }

    private string? _fullName;
    public string FullName
```

```
    {
        get
        {
            if (_fullName is null)
                _fullName = $"{FirstName} {LastName}";
            return _fullName;
        }
    }
}
```

此最终版本仅在必要时计算 `FullName` 属性。 如果以前计算的版本有效，则使用它。 否则，计算会更新缓存的值。 使用此类的开发人员无需了解实现的细枝末节。 这些内部更改不会影响 Person 对象的使用。

从 C# 13 开始，你可以在 `partial` 类中创建 partial 属性。 属性的 `partial` 实现声明不能是自动实现的属性。 自动实现的属性使用与声明分部属性声明相同的语法。

属性

属性是类或对象中的一种智能字段形式。 从对象外部，它们看起来像对象中的字段。 但是，属性可以通过丰富的 C# 功能来实现。 你可以提供验证、不同的可访问性、迟缓计算或方案所需的任何要求。

- 无需自定义访问器代码的简单属性可以作为表达式正文定义或 自动实现的属性实现。
- 属性允许类公开获取和设置值的公共方法，而隐藏实现或验证代码。
- get 属性访问器用于返回属性值，而 set 属性访问器用于分配新值。 init 属性访问器仅用于在对象构造过程中分配新值。 这些访问器可以具有不同的访问级别。 有关详细信息，请参阅限制访问器可访问性。
- value 关键字用于定义由 `set` 或 `init` 访问器分配的值。
- 属性可以是*读-写*属性（既有 `get` 访问器又有 `set` 访问器）、*只读*属性（有 `get` 访问器，但没有 `set` 访问器）或*只写*访问器（有 `set` 访问器，但没有 `get` 访问器）。 只写属性很少见。

C# 语言规范

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

另请参阅

- 索引器

- init 关键字
- get 关键字
- set 关键字

使用属性（C# 编程指南）

项目 • 2024/11/14

属性结合了字段和方法的多个方面。 对于对象的用户来说，属性似乎是一个字段，访问属性需要相同的语法。 对于类的实现者来说，属性是一两个代码块，表示 get 访问器和/或 set 或 init 访问器。 读取属性时，执行 `get` 访问器的代码块；向属性赋予值时，执行 `set` 或 `init` 访问器的代码块。 将不带 `set` 访问器的属性视为只读。 将不带 `get` 访问器的属性视为只写。 将具有以上两个访问器的属性视为读写。 可通过 `init` 访问器而不是 `set` 访问器使该属性能够设置为对象初始化的一部分并在其他情况下使其只读。

与字段不同，属性不会被归类为变量。 因此，不能将属性作为 ref 或 out 参数传递。

属性有许多用途：

- 它们可在允许更改数据之前验证数据。
- 它们可以透明方式将从其他源（例如数据库）检索数据的类上的数据公开。
- 它们可在数据发生更改（例如引发事件或更改其他字段的值）时执行操作。

通过依次指定字段的访问级别、属性类型、属性名、声明 `get` 访问器和/或 `set` 访问器的代码块，在类块中声明属性。 例如：

C#

```
public class Date
{
    private int _month = 7;  // Backing store

    public int Month
    {
        get => _month;
        set
        {
            if ((value > 0) && (value < 13))
            {
                _month = value;
            }
        }
    }
}
```

此示例中将 `Month` 声明为属性，以便 `set` 访问器可确保 `Month` 值设置在 1 至 12 之间。 `Month` 属性使用私有字段跟踪实际值。 属性数据的真实位置通常称为属性的“后备存储”，属性通常使用专用字段作为后备存储。 将字段被标记为私有，确保只能通过调用属性来对其进行更改。 有关公共和专用访问限制的详细信息，请参阅访问修饰符。 自动实现的属性为简单属性声明提供简化的语法。 有关详细信息，请参阅 自动实现的属性。

从 C# 13 开始，可以使用字段支持的属性向set自动实现的属性的访问器添加验证，如以下示例所示：

C#

```
public class DateExample
{
    public int Month
    {
        get;
        set
        {
            if ((value > 0) && (value < 13))
            {
                field = value;
            }
        }
    }
}
```

> **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。

get 访问器

`get` 访问器的正文类似于方法。 它必须返回属性类型的值。 C# 编译器和实时 (JIT) 编译器检测 `get` 访问器的常见实现模式并优化这些模式。 例如，可能会优化返回字段而不执行任何计算的 `get` 访问器对该字段的内存读取。 自动实现的属性遵循此模式，并从这些优化中获益。 但是，无法内联虚拟 `get` 访问器方法，因为编译器在编译时不知道在运行时实际可调用哪些方法。 以下示例显示一个 `get` 访问器，它返回私有字段 `_name` 的值：

C#

```
    class Employee
{
    private string _name;  // the name field
```

```
    public string Name => _name;     // the Name property
}
```

引用属性时，除了作为赋值目标外，还调用 `get` 访问器读取属性值。 例如：

C#

```
var employee= new Employee();
//...

System.Console.Write(employee.Name);  // the get accessor is invoked here
```

`get` 访问器必须是表达式主体成员，或在 return 或 throw 语句中结束，且控件无法流出访问器主体。

> ⚠ **警告**
>
> 使用访问器更改对象的 `get` 状态通常是一种错误的编程样式。 此规则的一个例外是*延迟计算*的属性，其中仅当首次访问属性时，才会计算属性的值。

`get` 访问器可以用于返回字段值或计算并返回字段值。 例如：

C#

```
class Manager
{
    private string _name;
    public string Name => _name != null ? _name : "NA";
}
```

在前面的示例中，如果不向 `Name` 属性赋值，则返回值 `NA`。

set 访问器

`set` 访问器类似于返回类型为 void 的方法。 它使用名为 `value` 的隐式参数，该参数的类型为属性的类型。 编译器和 JIT 编译器还识别 `set` 或 `init` 访问器的常见模式。 这些常见模式经过优化，直接写入支持字段的内存。 在下面的示例中，将 `set` 访问器添加到 `Name` 属性：

C#

```
class Student
{
```

```
    private string _name;  // the name field
    public string Name    // the Name property
    {
        get => _name;
        set => _name = value;
    }
}
```

向属性赋值时，通过使用提供新值的自变量调用 `set` 访问器。 例如：

C#

```
var student = new Student();
student.Name = "Joe";  // the set accessor is invoked here

System.Console.Write(student.Name);  // the get accessor is invoked here
```

为 `set` 访问器中的本地变量声明使用隐式参数名 `value` 是错误的。

init 访问器

用于创建 `init` 访问器的代码与用于创建 `set` 访问器的代码相同，只不过前者使用的关键字是 `init` 而不是 `set`。 不同之处在于，`init` 访问器只能在构造函数中使用，或通过对象初始值设定项使用。

备注

可以将属性标记为 `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属性的方式。 相同属性的 `get` 和 `set` 访问器可以具有不同的访问修饰符。 例如，`get` 可能为 `public` 允许从类型外部进行只读访问；而 `set` 可能为 `private` 或 `protected`。 有关详细信息，请参阅访问修饰符。

可以通过使用 `static` 关键字将属性声明为静态属性。 静态属性可供调用方在任何时候使用，即使不存在类的任何实例。 有关详细信息，请参阅静态类和静态类成员。

可以通过使用 virtual 关键字将属性标记为虚拟属性。 虚拟属性可使派生类使用 override 关键字重写属性行为。 有关这些选项的详细信息，请参阅继承。

重写虚拟属性的属性也可以是 sealed，指定对于派生类，它不再是虚拟的。 最后，可以将属性声明为 abstract。 抽象属性不定义类中的实现，派生类必须写入自己的实现。 有关这些选项的详细信息，请参阅抽象类、密封类及类成员。

> **备注**
>
> 在 static 属性的访问器上使用 virtual、abstract 或 override 修饰符是错误的。

示例

此示例演示实例、静态和只读属性。 它接收通过键盘键入的员工姓名，按 1 递增 `NumberOfEmployees`，并显示员工姓名和编号。

C#

```
public class Employee
{
    public static int NumberOfEmployees;
    private static int _counter;
    private string _name;

    // A read-write instance property:
    public string Name
    {
        get => _name;
        set => _name = value;
    }

    // A read-only static property:
    public static int Counter => _counter;

    // A Constructor:
    public Employee() => _counter = ++NumberOfEmployees; // Calculate the
employee's number:
}
```

Hidden 属性示例

此示例演示如何访问由派生类中同名的另一属性隐藏的基类中的属性：

C#

```
public class Employee
{
    private string _name;
    public string Name
    {
        get => _name;
        set => _name = value;
    }
}
```

```
public class Manager : Employee
{
    private string _name;

    // Notice the use of the new modifier:
    public new string Name
    {
        get => _name;
        set => _name = value + ", Manager";
    }
}

class TestHiding
{
    public static void Test()
    {
        Manager m1 = new Manager();

        // Derived class property.
        m1.Name = "John";

        // Base class property.
        ((Employee)m1).Name = "Mary";

        System.Console.WriteLine("Name in the derived class is: {0}",
m1.Name);
        System.Console.WriteLine("Name in the base class is: {0}",
((Employee)m1).Name);
    }
}
/* Output:
    Name in the derived class is: John, Manager
    Name in the base class is: Mary
*/
```

以下内容是前一示例中的要点：

- 派生类中的属性 `Name` 隐藏基类中的属性 `Name`。 在这种情况下，`new` 修饰符用于派生类的属性声明中：

  C#

  ```
  public new string Name
  ```

- `(Employee)` 转换用于访问基类中的隐藏属性：

  C#

  ```
  ((Employee)m1).Name = "Mary";
  ```

有关隐藏成员的详细信息，请参阅 new 修饰符。

Override 属性示例

在此示例中，两个类（`Cube` 和 `Square`）实现抽象类 `Shape`，并重写其抽象 `Area` 属性。请注意属性上的 override 修饰符的使用。 程序接受将边长作为输入，计算正方形和立方体的面积。 它还接受将面积作为输入，计算正方形和立方体的相应边长。

C#

```
abstract class Shape
{
    public abstract double Area
    {
        get;
        set;
    }
}

class Square : Shape
{
    public double side;

    //constructor
    public Square(double s) => side = s;

    public override double Area
    {
        get => side * side;
        set => side = System.Math.Sqrt(value);
    }
}

class Cube : Shape
{
    public double side;

    //constructor
    public Cube(double s) => side = s;

    public override double Area
    {
        get => 6 * side * side;
        set => side = System.Math.Sqrt(value / 6);
    }
}

class TestShapes
{
    static void Main()
    {
```

```
        // Input the side:
        System.Console.Write("Enter the side: ");
        double side = double.Parse(System.Console.ReadLine());

        // Compute the areas:
        Square s = new Square(side);
        Cube c = new Cube(side);

        // Display the results:
        System.Console.WriteLine("Area of the square = {0:F2}", s.Area);
        System.Console.WriteLine("Area of the cube = {0:F2}", c.Area);
        System.Console.WriteLine();

        // Input the area:
        System.Console.Write("Enter the area: ");
        double area = double.Parse(System.Console.ReadLine());

        // Compute the sides:
        s.Area = area;
        c.Area = area;

        // Display the results:
        System.Console.WriteLine("Side of the square = {0:F2}", s.side);
        System.Console.WriteLine("Side of the cube = {0:F2}", c.side);
    }
}
/* Example Output:
    Enter the side: 4
    Area of the square = 16.00
    Area of the cube = 96.00

    Enter the area: 24
    Side of the square = 4.90
    Side of the cube = 2.00
*/
```

请参阅

- 属性
- 接口属性
- 自动实现的属性
- 分部属性

接口属性（C# 编程指南）

项目 • 2024/10/26

可以在接口上声明属性。 下面的示例声明一个接口属性访问器：

C#

```
public interface ISampleInterface
{
    // Property declaration:
    string Name
    {
        get;
        set;
    }
}
```

接口属性通常没有主体。 访问器指示属性是读写、只读还是只写。 与类和结构不同，在没有正文的情况下声明访问器不会声明 自动实现的属性。 接口可为成员（包括属性）定义默认实现。 在接口中为属性定义默认实现的情况非常少，因为接口无法定义实例数据字段。

示例

在此示例中，接口 `IEmployee` 具有读写属性 `Name` 和只读属性 `Counter`。 类 `Employee` 实现 `IEmployee` 接口，并使用这两个属性。 程序读取新员工的姓名以及当前员工数，并显示员工名称和计算的员工数。

可以使用属性的完全限定名称，它引用其中声明成员的接口。 例如：

C#

```
string IEmployee.Name
{
    get { return "Employee Name"; }
    set { }
}
```

前面的示例演示了如何显式接口实现。 例如，如果 `Employee` 类正在实现接口 `ICitizen` 和接口 `IEmployee`，而这两个接口都具有 `Name` 属性，则需要用到显式接口成员实现。 即是说下列属性声明：

C#

```
string IEmployee.Name
{
    get { return "Employee Name"; }
    set { }
}
```

在 `IEmployee` 接口中实现 `Name` 属性，而以下声明：

C#

```
string ICitizen.Name
{
    get { return "Citizen Name"; }
    set { }
}
```

在 `ICitizen` 接口中实现 `Name` 属性。

C#

```
interface IEmployee
{
    string Name
    {
        get;
        set;
    }

    int Counter
    {
        get;
    }
}

public class Employee : IEmployee
{
    public static int numberOfEmployees;

    private string _name;
    public string Name  // read-write instance property
    {
        get => _name;
        set => _name = value;
    }

    private int _counter;
    public int Counter  // read-only instance property
    {
        get => _counter;
    }
```

```
    // constructor
    public Employee() => _counter = ++numberOfEmployees;
}
```

C#

```
System.Console.Write("Enter number of employees: ");
Employee.numberOfEmployees = int.Parse(System.Console.ReadLine());

Employee e1 = new Employee();
System.Console.Write("Enter the name of the new employee: ");
e1.Name = System.Console.ReadLine();

System.Console.WriteLine("The employee information:");
System.Console.WriteLine("Employee number: {0}", e1.Counter);
System.Console.WriteLine("Employee name: {0}", e1.Name);
```

示例输出

控制台

```
Enter number of employees: 210
Enter the name of the new employee: Hazem Abolrous
The employee information:
Employee number: 211
Employee name: Hazem Abolrous
```

另请参阅

- 属性
- 使用属性
- 属性与索引器之间的比较
- 索引器
- 接口

限制访问器可访问性（C# 编程指南）

项目 • 2024/10/30

属性或索引器的 get 和 set 部分称为访问器。 默认情况下，这些访问器具有与其所属属性或索引器相同的可见性或访问级别。 有关详细信息，请参阅可访问性级别。 不过，有时限制对其中某个访问器的访问是有益的。 通常，限制 `set` 访问器的可访问性，同时保持 `get` 访问器可公开访问。 例如：

C#

```
private string _name = "Hello";

public string Name
{
    get
    {
        return _name;
    }
    protected set
    {
        _name = value;
    }
}
```

在此示例中，名为 `Name` 的属性定义 `get` 访问器和 `set` 访问器。 `get` 访问器接收该属性本身的可访问性级别（此示例中为 `public`），而对于 `set` 访问器，则通过对该访问器本身应用 protected 访问修饰符来进行显式限制。

> **备注**
>
> 本文中的示例不使用 **自动实现的属性**。 *自动实现的属性* 提供一个简洁的语法，用于在不需要自定义后盾字段时声明属性。

对访问器的访问修饰符的限制

对属性或索引器使用访问修饰符受以下条件的制约：

- 不能对接口或显式接口成员实现使用访问器修饰符。
- 仅当属性或索引器同时具有 `set` 和 `get` 访问器时，才能使用访问器修饰符。 这种情况下，只允许对其中一个访问器使用修饰符。
- 如果属性或索引器具有 override 修饰符，则访问器修饰符必须与重写的访问器的访问器（如有）匹配。

- 访问器的可访问性级别必须比属性或索引器本身的可访问性级别具有更严格的限制。

重写访问器的访问修饰符

重写属性或索引器时，被重写的访问器对重写代码而言必须是可访问的。 此外，属性/索引器及其访问器的可访问性都必须与相应的被重写属性/索引器及其访问器匹配。 例如：

C#

```
public class Parent
{
    public virtual int TestProperty
    {
        // Notice the accessor accessibility level.
        protected set { }

        // No access modifier is used here.
        get { return 0; }
    }
}
public class Kid : Parent
{
    public override int TestProperty
    {
        // Use the same accessibility level as in the overridden accessor.
        protected set { }

        // Cannot use access modifier here.
        get { return 0; }
    }
}
```

实现接口

使用访问器实现接口时，访问器不一定有访问修饰符。 但如果使用一个访问器（如 `get`）实现接口，则另一个访问器可以具有访问修饰符，如下面的示例所示：

C#

```
public interface ISomeInterface
{
    int TestProperty
    {
        // No access modifier allowed here
        // because this is an interface.
        get;
    }
```

```
}

public class TestClass : ISomeInterface
{
    public int TestProperty
    {
        // Cannot use access modifier here because
        // this is an interface implementation.
        get { return 10; }

        // Interface property does not have set accessor,
        // so access modifier is allowed.
        protected set { }
    }
}
```

访问器可访问性域

如果对访问器使用访问修饰符，则访问器的可访问性域由该修饰符确定。

如果未对访问器使用访问修饰符，则访问器的可访问性域由属性或索引器的可访问性级别确定。

示例

下面的示例包含三个类：`BaseClass`、`DerivedClass` 和 `MainClass`。 每个类的 `BaseClass`、`Name` 和 `Id` 都有两个属性。 该示例演示在使用限制性访问修饰符（如 protected 或 private）时，`BaseClass` 的 `Id` 属性如何隐藏 `DerivedClass` 的 `Id` 属性。 因此，向该属性赋值时将调用 `BaseClass` 类中的属性。 将访问修饰符替换为 public 将使该属性可供访问。

该示例还演示了 `DerivedClass` 中 `Name` 属性的 `set` 访问器的限制性访问修饰符（例如 `private` 或 `protected`）如何阻止访问派生类中的访问器。 在向该修饰符赋值时，它会生成一个错误，或者访问同名的基类属性（如果可访问）。

C#

```
public class BaseClass
{
    private string _name = "Name-BaseClass";
    private string _id = "ID-BaseClass";

    public string Name
    {
        get { return _name; }
        set { }
```

```
    }

    public string Id
    {
        get { return _id; }
        set { }
    }
}

public class DerivedClass : BaseClass
{
    private string _name = "Name-DerivedClass";
    private string _id = "ID-DerivedClass";

    new public string Name
    {
        get
        {
            return _name;
        }

        // Using "protected" would make the set accessor not accessible.
        set
        {
            _name = value;
        }
    }

    // Using private on the following property hides it in the Main Class.
    // Any assignment to the property will use Id in BaseClass.
    new private string Id
    {
        get
        {
            return _id;
        }
        set
        {
            _id = value;
        }
    }
}

class MainClass
{
    static void Main()
    {
        BaseClass b1 = new BaseClass();
        DerivedClass d1 = new DerivedClass();

        b1.Name = "Mary";
        d1.Name = "John";

        b1.Id = "Mary123";
        d1.Id = "John123";  // The BaseClass.Id property is called.
```

```
        System.Console.WriteLine("Base: {0}, {1}", b1.Name, b1.Id);
        System.Console.WriteLine("Derived: {0}, {1}", d1.Name, d1.Id);

        // Keep the console window open in debug mode.
        System.Console.WriteLine("Press any key to exit.");
        System.Console.ReadKey();
    }
}
/* Output:
    Base: Name-BaseClass, ID-BaseClass
    Derived: John, ID-BaseClass
*/
```

注释

注意，如果将声明 `new private string Id` 替换为 `new public string Id`，则得到如下输出：

`Name and ID in the base class: Name-BaseClass, ID-BaseClass` `Name and ID in the derived class: John, John123`

请参阅

- 属性
- 索引器
- 访问修饰符
- 仅初始化属性
- 必需的属性

如何声明和使用读/写属性（C# 编程指南）

项目 • 2023/04/07

属性提供了公共数据成员的便利性，且不会产生未受保护、不可控制和未经验证地访问对象的数据的风险。 属性声明访问器：从基础数据成员中赋值和检索值的特殊方法。 set 访问器可分配数据成员，get 访问器检索数据成员值。

此示例演示具有两个属性的 `Person` 类：`Name`（字符串）和 `Age`（整型）。 这两个属性均提供 `get` 和 `set` 访问器，因此它们被视为读/写属性。

示例

C#

```
class Person
{
    private string _name = "N/A";
    private int _age = 0;

    // Declare a Name property of type string:
    public string Name
    {
        get
        {
            return _name;
        }
        set
        {
            _name = value;
        }
    }

    // Declare an Age property of type int:
    public int Age
    {
        get
        {
            return _age;
        }

        set
        {
            _age = value;
        }
    }
```

```
    public override string ToString()
    {
        return "Name = " + Name + ", Age = " + Age;
    }
}

public class Wrapper
{
    private string _name = "N/A";
    public string Name
    {
        get
        {
            return _name;
        }
        private set
        {
            _name = value;
        }
    }

}

class TestPerson
{
    static void Main()
    {
        // Create a new Person object:
        Person person = new Person();

        // Print out the name and the age associated with the person:
        Console.WriteLine("Person details - {0}", person);

        // Set some values on the person object:
        person.Name = "Joe";
        person.Age = 99;
        Console.WriteLine("Person details - {0}", person);

        // Increment the Age property:
        person.Age += 1;
        Console.WriteLine("Person details - {0}", person);

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
/* Output:
    Person details - Name = N/A, Age = 0
    Person details - Name = Joe, Age = 99
    Person details - Name = Joe, Age = 100
*/
```

可靠编程

在前面的示例中，`Name` 和 `Age` 属性为 public，同时包含 `get` 和 `set` 访问器。 公共访问器使得任何对象均可读取和写入这些属性。 但是，有时需要排除其中的一个访问器。 可以省略 `set` 访问器以使属性为只读：

C#

```
public string Name
{
    get
    {
        return _name;
    }
    private set
    {
        _name = value;
    }
}
```

或者，可以公开一个访问器，但使另一个访问器为私有或受保护。 有关详细信息，请参阅非对称性访问器可访问性。

声明属性后，便可将这些属性用作类的字段。 在获取和设置属性的值时，属性允许一种自然的语法，如以下语句所示：

C#

```
person.Name = "Joe";
person.Age = 99;
```

在属性 `set` 方法中，特殊的 `value` 变量为可用。 此变量包含用户指定的值，例如：

C#

```
_name = value;
```

请注意在 `Person` 对象上递增 `Age` 属性的简洁语法：

C#

```
person.Age += 1;
```

如果将单独的 `set` 和 `get` 方法用于模型属性，则等效的代码可能如下所示：

C#

```
person.SetAge(person.GetAge() + 1);
```

`ToString` 方法在此示例中被重写：

C#

```
public override string ToString()
{
    return "Name = " + Name + ", Age = " + Age;
}
```

请注意，`ToString` 未显式用于程序中。 默认情况下，通过 `WriteLine` 调用对其进行调用。

请参阅

- 属性
- C# 类型系统

自动实现的属性

项目 • 2024/11/14

当属性访问器中不需要其他逻辑时，自动实现的属性使属性声明更加简洁。 它们还允许客户端代码创建对象。 当你声明以下示例中所示的属性时，编译器将创建仅可以通过该属性的 `get` 和 `set` 访问器访问的专用、匿名支持字段。 `init` 访问器也可以声明为自动实现的属性。

以下示例演示了一个具有一些自动实现属性的简单类：

C#

```
// This class is mutable. Its data can be modified from
// outside the class.
public class Customer
{
    // Auto-implemented properties for trivial get and set
    public double TotalPurchases { get; set; }
    public string Name { get; set; }
    public int CustomerId { get; set; }

    // Constructor
    public Customer(double purchases, string name, int id)
    {
        TotalPurchases = purchases;
        Name = name;
        CustomerId = id;
    }

    // Methods
    public string GetContactInfo() { return "ContactInfo"; }
    public string GetTransactionHistory() { return "History"; }

    // .. Additional methods, events, etc.
}

class Program
{
    static void Main()
    {
        // Initialize a new object.
        Customer cust1 = new Customer(4987.63, "Northwind", 90108);

        // Modify a property.
        cust1.TotalPurchases += 499.99;
    }
}
```

无法在接口中声明自动实现的属性。 自动实现的属性和字段支持的属性声明专用实例支持字段，接口无法声明实例字段。 在接口中声明属性而不定义正文声明具有访问器的属性。 实现该接口的每个类型都必须实现该属性。

可以初始化自动实现的属性，类似于字段：

C#

```
public string FirstName { get; set; } = "Jane";
```

上一示例中所示的类是可变的。 客户端代码在创建后可以更改对象中的值。 在包含重要行为（方法）以及数据的复杂类中，通常有必要具有公共属性。 但是，对于那些仅封装一组值（数据）且很少或没有行为的小型类或结构，应该使用以下选项之一使对象不可变：

- 只声明 `get` 访问器（除了能在构造函数中可变，在其他任何位置都不可变）。
- 声明 `get` 访问器和 `init` 访问器（除了能在对象构造函数中可变，在其他任何位置都不可变）。
- 将 `set` 访问器声明为专用（对使用者不可变）。

有关详细信息，请参阅 如何实现具有自动实现属性的轻型类。

可能需要将验证添加到自动实现的属性。 C# 13 将字段支持的属性添加为预览功能。 使用 `field` 关键字访问自动实现属性的编译器合成后盾字段。 例如，可以确保 `FirstName` 前面的示例中的属性不能设置为 `null` 或空字符串：

C#

```
public string FirstName
{
    get;
    set
    {
        field = (string.IsNullOrWhiteSpace(value) is false
            ? value
            : throw new ArgumentException(nameof(value), "First name can't
be whitespace or null"));
    }
} = "Jane";
```

此功能使你可以向访问器添加逻辑，而无需显式声明后盾字段。 使用 `field` 关键字访问编译器生成的后盾字段。

> ⓘ **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。

另请参阅

- 使用自动实现的属性（样式规则 IDE0032）
- 属性
- 修饰符

如何实现具有自动实现属性的轻型类

项目 • 2024/10/26

此示例演示如何创建一个不可变的轻型类，该类仅用于封装一组自动实现的属性。 当你必须使用引用类型语义时，请使用此种构造而不是结构。

可通过以下方法来实现不可变的属性：

- 仅声明 get 访问器，使属性除了能在该类型的构造函数中可变，在其他任何位置都不可变。
- 声明 init 访问器而不是 `set` 访问器，这使属性只能在构造函数中进行设置，或者通过使用对象初始值设定项设置。
- 将 set 访问器声明为专用。 属性可在该类型中设置，但它对于使用者是不可变的。

可以将 required 修饰符添加到属性声明中，以强制调用方将属性设置为初始化新对象的一部分。

下面的示例显示了只有 get 访问器的属性与具有 get 和 private set 的属性的区别。

C#

```
class Contact
{
    public string Name { get; }
    public string Address { get; private set; }

    public Contact(string contactName, string contactAddress)
    {
        // Both properties are accessible in the constructor.
        Name = contactName;
        Address = contactAddress;
    }

    // Name isn't assignable here. This will generate a compile error.
    //public void ChangeName(string newName) => Name = newName;

    // Address is assignable here.
    public void ChangeAddress(string newAddress) => Address = newAddress;
}
```

示例

以下示例演示了实现具有自动实现属性的不可变类的两种方法。 这两种方法均使用 private `set` 声明其中一个属性，使用单独的 `get` 声明另一个属性。 第一个类仅使用构造函数来初始化属性，第二个类则使用可调用构造函数的静态工厂方法。

C#

```
// This class is immutable. After an object is created,
// it cannot be modified from outside the class. It uses a
// constructor to initialize its properties.
class Contact
{
    // Read-only property.
    public string Name { get; }

    // Read-write property with a private set accessor.
    public string Address { get; private set; }

    // Public constructor.
    public Contact(string contactName, string contactAddress)
    {
        Name = contactName;
        Address = contactAddress;
    }
}

// This class is immutable. After an object is created,
// it cannot be modified from outside the class. It uses a
// static method and private constructor to initialize its properties.
public class Contact2
{
    // Read-write property with a private set accessor.
    public string Name { get; private set; }

    // Read-only property.
    public string Address { get; }

    // Private constructor.
    private Contact2(string contactName, string contactAddress)
    {
        Name = contactName;
        Address = contactAddress;
    }

    // Public factory method.
    public static Contact2 CreateContact(string name, string address)
    {
        return new Contact2(name, address);
    }
}

public class Program
{
    static void Main()
    {
        // Some simple data sources.
        string[] names = ["Terry Adams","Fadi Fakhouri", "Hanying Feng",
                          "Cesar Garcia", "Debra Garcia"];
        string[] addresses = ["123 Main St.", "345 Cypress Ave.", "678 1st
```

```
Ave",
                                "12 108th St.", "89 E. 42nd St."];

        // Simple query to demonstrate object creation in select clause.
        // Create Contact objects by using a constructor.
        var query1 = from i in Enumerable.Range(0, 5)
                     select new Contact(names[i], addresses[i]);

        // List elements cannot be modified by client code.
        var list = query1.ToList();
        foreach (var contact in list)
        {
            Console.WriteLine("{0}, {1}", contact.Name, contact.Address);
        }

        // Create Contact2 objects by using a static factory method.
        var query2 = from i in Enumerable.Range(0, 5)
                         select Contact2.CreateContact(names[i],
addresses[i]);

        // Console output is identical to query1.
        var list2 = query2.ToList();

        // List elements cannot be modified by client code.
        // CS0272:
        // list2[0].Name = "Eugene Zabokritski";
    }
}

/* Output:
    Terry Adams, 123 Main St.
    Fadi Fakhouri, 345 Cypress Ave.
    Hanying Feng, 678 1st Ave
    Cesar Garcia, 12 108th St.
    Debra Garcia, 89 E. 42nd St.
*/
```

编译器为每个自动实现的属性创建后盾字段。 这些字段无法直接从源代码进行访问。

请参阅

- 属性
- struct
- 对象和集合初始值设定项

方法（C# 编程指南）

项目 • 2023/04/07

方法是包含一系列语句的代码块。 程序通过调用该方法并指定任何所需的方法参数使语句得以执行。 在 C# 中，每个执行的指令均在方法的上下文中执行。

`Main` 方法是每个 C# 应用程序的入口点，并在启动程序时由公共语言运行时 (CLR) 调用。 在使用顶级语句的应用程序中，`Main` 方法由编译器生成并包含所有顶级语句。

> **备注**
>
> 本文讨论命名的方法。 有关匿名函数的信息，请参阅 Lambda **表达式**。

方法签名

通过指定访问级别（如 `public` 或 `private`）、可选修饰符（如 `abstract` 或 `sealed`）、返回值、方法的名称以及任何方法参数，在类、结构或接口中声明方法。 这些部件一起构成方法的签名。

> **重要**
>
> 出于方法重载的目的，方法的返回类型不是方法签名的一部分。 但是在确定委托和它所指向的方法之间的兼容性时，它是方法签名的一部分。

方法参数在括号内，并且用逗号分隔。 空括号指示方法不需要任何参数。 此类包含四种方法：

C#

```
abstract class Motorcycle
{
    // Anyone can call this.
    public void StartEngine() {/* Method statements here */ }

    // Only derived classes can call this.
    protected void AddGas(int gallons) { /* Method statements here */ }

    // Derived classes can override the base class implementation.
    public virtual int Drive(int miles, int speed) { /* Method statements
here */ return 1; }

    // Derived classes must implement this.
```

```
    public abstract double GetTopSpeed();
}
```

方法访问

调用对象上的方法就像访问字段。 在对象名之后添加一个句点、方法名和括号。 参数列在括号里，并且用逗号分隔。 因此，可在以下示例中调用 `Motorcycle` 类的方法：

C#

```
class TestMotorcycle : Motorcycle
{

    public override double GetTopSpeed()
    {
        return 108.4;
    }

    static void Main()
    {

        TestMotorcycle moto = new TestMotorcycle();

        moto.StartEngine();
        moto.AddGas(15);
        moto.Drive(5, 20);
        double speed = moto.GetTopSpeed();
        Console.WriteLine("My top speed is {0}", speed);
    }
}
```

方法形参与实参

该方法定义指定任何所需参数的名称和类型。 调用代码调用该方法时，它为每个参数提供了称为参数的具体值。 参数必须与参数类型兼容，但调用代码中使用的参数名（如果有）不需要与方法中定义的参数名相同。 例如：

C#

```
public void Caller()
{
    int numA = 4;
    // Call with an int variable.
    int productA = Square(numA);

    int numB = 32;
    // Call with another int variable.
```

```
    int productB = Square(numB);

    // Call with an integer literal.
    int productC = Square(12);

    // Call with an expression that evaluates to int.
    productC = Square(productA * 3);
}

int Square(int i)
{
    // Store input argument in a local variable.
    int input = i;
    return input * input;
}
```

按引用传递与按值传递

默认情况下，将值类型的实例传递给方法时，传递的是其副本而不是实例本身。 因此，对参数的更改不会影响调用方法中的原始实例。 若要按引用传递值类型实例，请使用 `ref` 关键字。 有关详细信息，请参阅传递值类型参数。

引用类型的对象传递到方法中时，将传递对对象的引用。 也就是说，该方法接收的不是对象本身，而是指示该对象位置的参数。 如果通过使用此引用更改对象的成员，即使是按值传递该对象，此更改也会反映在调用方法的参数中。

通过使用 `class` 关键字创建引用类型，如以下示例所示：

C#

```
public class SampleRefType
{
    public int value;
}
```

现在，如果将基于此类型的对象传递到方法，则将传递对对象的引用。 下面的示例将 `SampleRefType` 类型的对象传递到 `ModifyObject` 方法：

C#

```
public static void TestRefType()
{
    SampleRefType rt = new SampleRefType();
    rt.value = 44;
    ModifyObject(rt);
    Console.WriteLine(rt.value);
}
```

```
static void ModifyObject(SampleRefType obj)
{
    obj.value = 33;
}
```

该示例执行的内容实质上与先前示例相同，均按值将自变量传递到方法。 但是因为使用了引用类型，结果有所不同。 `ModifyObject` 中所做的对形参 `value` 的 `obj` 字段的修改，也会更改 `value` 方法中实参 `rt` 的 `TestRefType` 字段。 `TestRefType` 方法显示 33 作为输出。

有关如何通过引用和值传递引用类型的详细信息，请参阅传递引用类型参数和引用类型。

返回值

方法可以将值返回到调用方。 如果列在方法名之前的返回类型不是 `void`，则该方法可通过使用 return 语句返回值。 带 `return` 关键字，后跟与返回类型匹配的值的语句将该值返回到方法调用方。

值可以按值或按引用返回到调用方。 如果在方法签名中使用 `ref` 关键字且其跟随每个 `return` 关键字，值将按引用返回到调用方。 例如，以下方法签名和返回语句指示该方法按对调用方的引用返回名为 `estDistance` 的变量。

C#

```
public ref double GetEstimatedDistance()
{
    return ref estDistance;
}
```

`return` 关键字还会停止执行该方法。 如果返回类型为 `void`，没有值的 `return` 语句仍可用于停止执行该方法。 没有 `return` 关键字，当方法到达代码块结尾时，将停止执行。 具有非空的返回类型的方法都需要使用 `return` 关键字来返回值。 例如，这两种方法都使用 `return` 关键字来返回整数：

C#

```
class SimpleMath
{
    public int AddTwoNumbers(int number1, int number2)
    {
        return number1 + number2;
    }

    public int SquareANumber(int number)
    {
```

```
        return number * number;
    }
}
```

若要使用从方法返回的值，调用方法可以在相同类型的值足够的地方使用该方法调用本身。 也可以将返回值分配给变量。 例如，以下两个代码示例实现了相同的目标：

C#

```
int result = obj.AddTwoNumbers(1, 2);
result = obj.SquareANumber(result);
// The result is 9.
Console.WriteLine(result);
```

C#

```
result = obj.SquareANumber(obj.AddTwoNumbers(1, 2));
// The result is 9.
Console.WriteLine(result);
```

在这种情况下，使用本地变量 `result`存储值是可选的。 此步骤可以帮助提高代码的可读性，或者如果需要存储该方法整个范围内自变量的原始值，则此步骤可能很有必要。

若要使用按引用从方法返回的值，必须声明 ref local 变量（如果想要修改其值）。 例如，如果 `Planet.GetEstimatedDistance` 方法按引用返回 Double 值，则可以将其定义为具有如下所示代码的 ref local 变量：

C#

```
ref double distance = ref Planet.GetEstimatedDistance();
```

如果调用函数将数组传递到 `M`，则无需从修改数组内容的方法 `M` 返回多维数组。 你可能会从 `M` 返回生成的数组以获得值的良好样式或功能流，但这是不必要的，因为 C# 按值传递所有引用类型，且数组引用的值是指向数组的指针。 在方法 `M` 中，引用该数组的任何代码都能观察到数组内容的任何更改，如以下示例所示：

C#

```
static void Main(string[] args)
{
    int[,] matrix = new int[2, 2];
    FillMatrix(matrix);
    // matrix is now full of -1
}

public static void FillMatrix(int[,] matrix)
```

```
{
    for (int i = 0; i < matrix.GetLength(0); i++)
    {
        for (int j = 0; j < matrix.GetLength(1); j++)
        {
            matrix[i, j] = -1;
        }
    }
}
```

异步方法

通过使用异步功能，你可以调用异步方法而无需使用显式回调，也不需要跨多个方法或 lambda 表达式来手动拆分代码。

如果用 async 修饰符标记方法，则可以在该方法中使用 await 运算符。 当控件到达异步方法中的 await 表达式时，控件将返回到调用方，并在等待任务完成前，方法中进度将一直处于挂起状态。 任务完成后，可以在方法中恢复执行。

> **备注**
>
> 异步方法在遇到第一个尚未完成的 awaited 对象或到达异步方法的末尾时（以先发生者为准），将返回到调用方。

异步方法通常具有 Task<TResult>、Task、IAsyncEnumerable<T> 或 `void` 返回类型。 `void` 返回类型主要用于定义需要 `void` 返回类型的事件处理程序。 无法等待返回 `void` 的异步方法，并且返回 void 方法的调用方无法捕获该方法引发的异常。 异步方法可以具有任何类似任务的返回类型。

在以下示例中， `DelayAsync` 是具有 Task<TResult>返回类型的异步方法。 `DelayAsync` 具有返回整数的 `return` 语句。 因此， `DelayAsync` 的方法声明必须具有 `Task<int>`的返回类型。 因为返回类型是 `Task<int>`， `await` 中 `DoSomethingAsync` 表达式的计算如以下语句所示得出整数： `int result = await delayTask`。

`Main` 方法就是一个具有 Task 返回类型的异步方法示例。 它会转到 `DoSomethingAsync` 方法，因为它使用单个行进行表示，所以可省略 `async` 和 `await` 关键字。 因为 `DoSomethingAsync` 是异步方法，调用 `DoSomethingAsync` 的任务必须等待，如以下语句所示： `await DoSomethingAsync();`。

C#

```
class Program
{
```

```
    static Task Main() => DoSomethingAsync();

    static async Task DoSomethingAsync()
    {
        Task<int> delayTask = DelayAsync();
        int result = await delayTask;

        // The previous two statements may be combined into
        // the following statement.
        //int result = await DelayAsync();

        Console.WriteLine($"Result: {result}");
    }

    static async Task<int> DelayAsync()
    {
        await Task.Delay(100);
        return 5;
    }
}
// Example output:
//   Result: 5
```

异步方法不能声明任何 ref 或 out 参数，但是可以调用具有这类参数的方法。

有关异步方法的详细信息，请参阅使用 Async 和 Await 的异步编程和异步返回类型。

表达式主体定义

具有立即仅返回表达式结果，或单个语句作为方法主题的方法定义很常见。 以下是使用 `=>` 定义此类方法的语法快捷方式：

C#

```
public Point Move(int dx, int dy) => new Point(x + dx, y + dy);
public void Print() => Console.WriteLine(First + " " + Last);
// Works with operators, properties, and indexers too.
public static Complex operator +(Complex a, Complex b) => a.Add(b);
public string Name => First + " " + Last;
public Customer this[long id] => store.LookupCustomer(id);
```

如果该方法返回 `void` 或是异步方法，则该方法的主体必须是语句表达式（与 lambda 相同）。 对于属性和索引器，两者必须是只读的，并且不使用 `get` 访问器关键字。

迭代器

迭代器对集合执行自定义迭代，如列表或数组。 迭代器使用 yield return 语句返回元素，每次返回一个。 到达 `yield return` 语句时，会记住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。

通过使用 foreach 语句从客户端代码调用迭代器。

迭代器的返回类型可以是 IEnumerable、IEnumerable<T>、IAsyncEnumerable<T>、IEnumerator 或 IEnumerator<T>。

有关更多信息，请参见 迭代器。

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# 编程指南
- C# 类型系统
- 访问修饰符
- 静态类和静态类成员
- 继承
- 抽象类、密封类及类成员
- params
- out
- ref
- 方法参数

本地函数（C# 编程指南）

项目 • 2024/12/29

本地函数是一种嵌套在另一成员中的类型的方法。 仅能从其包含成员中调用它们。 可以在以下位置中声明和调用本地函数：

- 方法（尤其是迭代器方法和异步方法）
- 构造函数
- 属性访问器
- 事件访问器
- 匿名方法
- Lambda 表达式
- 终结器
- 其他本地函数

但是，不能在 expression-bodied 成员中声明本地函数。

> **备注**
>
> 在某些情况下，可以使用 lambda 表达式实现本地函数也支持的功能。 有关比较，请参阅**本地函数与 lambda 表达式**。

本地函数可使代码意图明确。 任何读取代码的人都可以看到，此方法不可调用，包含方法除外。 对于团队项目，它们也使得其他开发人员无法直接从类或结构中的其他位置错误调用此方法。

本地函数语法

本地函数被定义为包含成员中的嵌套方法。 其定义具有以下语法：

```
C#

<modifiers> <return-type> <method-name> <parameter-list>
```

> **备注**
>
> `<parameter-list>` 不应包含使用**上下文关键字** `value` 命名的参数。 编译器创建临时变量“value”，其中包含引用的外部变量，这在以后会导致歧义，还可能导致意外行为。

可以将以下修饰符用于本地函数：

- async
- unsafe
- static 静态本地函数无法捕获局部变量或实例状态。
- extern 外部本地函数必须为 `static`。

在包含成员中定义的所有本地变量（包括其方法参数）都可在非静态本地函数中访问。

与方法定义不同，本地函数定义不能包含成员访问修饰符。 由于所有本地函数都是私有的，包括访问修饰符（如 `private` 关键字），因此会生成编译器错误 CS0106，“修饰符”private“对此项无效。

以下示例定义了一个名为 `AppendPathSeparator` 的本地函数，该函数对于名为 `GetText` 的方法是私有的：

C#

```
private static string GetText(string path, string filename)
{
     var reader = File.OpenText($"{AppendPathSeparator(path)}{filename}");
     var text = reader.ReadToEnd();
     return text;

     string AppendPathSeparator(string filepath)
     {
        return filepath.EndsWith(@"\") ? filepath : filepath + @"\";
     }
}
```

可以将属性应用于本地函数、其参数和类型参数，如以下示例所示：

C#

```
#nullable enable
private static void Process(string?[] lines, string mark)
{
    foreach (var line in lines)
    {
        if (IsValid(line))
        {
            // Processing logic...
        }
    }

    bool IsValid([NotNullWhen(true)] string? line)
    {
        return !string.IsNullOrEmpty(line) && line.Length >= mark.Length;
```

```
    }
}
```

前面的示例使用特殊属性来帮助编译器在可为空的上下文中进行静态分析。

本地函数和异常

本地函数的一个实用功能是可以允许立即显示异常。 对于迭代器方法，仅在枚举返回的序列时才显示异常，而非在检索迭代器时。 对于异步方法，在等待返回的任务时，将观察到异步方法中引发的任何异常。

以下示例定义 `OddSequence` 方法，用于枚举指定范围中的奇数。 因为它会将一个大于 100 的数字传递到 `OddSequence` 迭代器方法，该方法将引发 ArgumentOutOfRangeException。 如示例中的输出所示，仅当循环访问数字时才显示异常，而非检索迭代器时。

C#

```
public class IteratorWithoutLocalExample
{
   public static void Main()
   {
      IEnumerable<int> xs = OddSequence(50, 110);
      Console.WriteLine("Retrieved enumerator...");

      foreach (var x in xs)  // line 11
      {
         Console.Write($"{x} ");
      }
   }

   public static IEnumerable<int> OddSequence(int start, int end)
   {
      if (start < 0 || start > 99)
         throw new ArgumentOutOfRangeException(nameof(start), "start must be
between 0 and 99.");
      if (end > 100)
         throw new ArgumentOutOfRangeException(nameof(end), "end must be
less than or equal to 100.");
      if (start >= end)
         throw new ArgumentException("start must be less than end.");

      for (int i = start; i <= end; i++)
      {
         if (i % 2 == 1)
            yield return i;
      }
   }
}
```

```
// The example displays the output like this:
//
//    Retrieved enumerator...
//    Unhandled exception. System.ArgumentOutOfRangeException: end must be
less than or equal to 100. (Parameter 'end')
//    at IteratorWithoutLocalExample.OddSequence(Int32 start, Int32
end)+MoveNext() in IteratorWithoutLocal.cs:line 22
//    at IteratorWithoutLocalExample.Main() in IteratorWithoutLocal.cs:line
11
```

如果将迭代器逻辑放入本地函数，则在检索枚举器时会引发参数验证异常，如下面的示例所示：

C#

```
public class IteratorWithLocalExample
{
   public static void Main()
   {
      IEnumerable<int> xs = OddSequence(50, 110);  // line 8
      Console.WriteLine("Retrieved enumerator...");

      foreach (var x in xs)
      {
         Console.Write($"{x} ");
      }
   }

   public static IEnumerable<int> OddSequence(int start, int end)
   {
      if (start < 0 || start > 99)
         throw new ArgumentOutOfRangeException(nameof(start), "start must be
between 0 and 99.");
      if (end > 100)
         throw new ArgumentOutOfRangeException(nameof(end), "end must be
less than or equal to 100.");
      if (start >= end)
         throw new ArgumentException("start must be less than end.");

      return GetOddSequenceEnumerator();

      IEnumerable<int> GetOddSequenceEnumerator()
      {
         for (int i = start; i <= end; i++)
         {
            if (i % 2 == 1)
               yield return i;
         }
      }
   }
}
// The example displays the output like this:
//
```

```
//    Unhandled exception. System.ArgumentOutOfRangeException: end must be
less than or equal to 100. (Parameter 'end')
//    at IteratorWithLocalExample.OddSequence(Int32 start, Int32 end) in
IteratorWithLocal.cs:line 22
//    at IteratorWithLocalExample.Main() in IteratorWithLocal.cs:line 8
```

本地函数与 Lambda 表达式

一目了然，本地函数和 lambda 表达式 相似。 在许多情况下，选择使用 Lambda 表达式还是本地函数是风格和个人偏好的问题。 但是，应该注意，从两者中选用一种的时机和条件其实是存在差别的。

让我们检查一下阶乘算法的本地函数实现和 lambda 表达式实现之间的差异。 下面是使用本地函数的版本：

C#

```
public static int LocalFunctionFactorial(int n)
{
    return nthFactorial(n);

    int nthFactorial(int number) => number < 2
        ? 1
        : number * nthFactorial(number - 1);
}
```

此版本使用 Lambda 表达式：

C#

```
public static int LambdaFactorial(int n)
{
    Func<int, int> nthFactorial = default(Func<int, int>);

    nthFactorial = number => number < 2
        ? 1
        : number * nthFactorial(number - 1);

    return nthFactorial(n);
}
```

命名

本地函数的命名方式与方法相同。 Lambda 表达式是一种匿名方法，需要分配给 `delegate` 类型的变量，通常是 `Action` 或 `Func` 类型。 声明本地函数时，此过程类似于编写普通方法；声明一个返回类型和一个函数签名。

函数签名和 Lambda 表达式类型

Lambda 表达式依赖于为其分配的 `Action`/`Func` 变量的类型来确定参数和返回类型。 在本地函数中，因为语法非常类似于编写常规方法，所以参数类型和返回类型已经是函数声明的一部分。

某些 lambda 表达式具有 *自然类型*，这使编译器能够推断 lambda 表达式的返回类型和参数类型。

明确赋值

Lambda 表达式是在运行时声明和分配的对象。 若要使用 lambda 表达式，需要明确分配它：必须声明分配给它的 `Action`/`Func` 变量以及分配给它的 lambda 表达式。 请注意，`LambdaFactorial` 必须先声明和初始化 Lambda 表达式 `nthFactorial`，然后再对其进行定义。 否则，会导致分配前引用 `nthFactorial` 时出现编译时错误。

本地函数在编译时定义。 由于未将它们分配给变量，因此可以从范围内的任意代码位置引用它们；在第一个示例 `LocalFunctionFactorial` 中，可以在 `return` 语句的之前或之后声明本地函数，而不会触发任何编译器错误。

这些区别意味着使用本地函数创建递归算法会更轻松。 你可以声明和定义一个调用自身的本地函数。 必须先声明 Lambda 表达式并为其分配默认值，然后才能将其重新分配给引用相同 lambda 表达式的正文。

实现为委托

Lambda 表达式在声明时转换为委托。 本地函数更加灵活，可以像传统方法一样编写，也可以作为委托编写。 只有在用作委托时，本地函数才转换为委托。

如果声明了本地函数，但只是通过像调用方法一样调用该函数来引用该函数，它将不会转换成委托。

变量捕获

明确分配的规则也会影响本地函数或 Lambda 表达式捕获的任何变量。 编译器可以执行静态分析，因此本地函数能够在封闭范围内明确分配捕获的变量。 请看以下示例：

C#

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

```
    return y;

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

编译器可以确定 `LocalFunction` 在调用时明确分配 `y`。 因为在 `LocalFunction` 语句之前调用了 `return`，所以在 `y` 语句中明确分配了 `return`。

当局部函数捕获封闭范围内的变量时，就像委托类型一样，局部函数是用闭包来实现的。

堆分配

根据它们的用途，本地函数可以避免 Lambda 表达式始终需要的堆分配。 如果本地函数永远不会转换为委托，并且本地函数捕获的变量都不会被其他转换为委托的 lambda 或本地函数捕获，则编译器可以避免堆分配。

请看以下异步示例：

C#

```
public async Task<string> PerformLongRunningWorkLambda(string address, int
index, string name)
{
    if (string.IsNullOrWhiteSpace(address))
        throw new ArgumentException(message: "An address is required",
paramName: nameof(address));
    if (index < 0)
        throw new ArgumentOutOfRangeException(paramName: nameof(index),
message: "The index must be non-negative");
    if (string.IsNullOrWhiteSpace(name))
        throw new ArgumentException(message: "You must supply a name",
paramName: nameof(name));

    Func<Task<string>> longRunningWorkImplementation = async () =>
    {
        var interimResult = await FirstWork(address);
        var secondResult = await SecondStep(index, name);
        return $"The results are {interimResult} and {secondResult}.
Enjoy.";
    };

    return await longRunningWorkImplementation();
}
```

此 lambda 表达式的关闭包含 `address`、`index`和 `name` 变量。 对于本地函数，实现关闭的对象可以是 `struct` 类型。 该结构类型将通过引用传递给本地函数。 实现中的这个差异将保存在分配上。

lambda 表达式的实例化需要额外的内存分配，这可能会影响时间关键代码路径的性能。本地函数不会产生此开销。

如果你知道本地函数不会转换为委托，并且本地函数捕获的变量都不会被其他转换为委托的 lambda 或本地函数捕获，则可以通过将本地函数声明为 `static` 本地函数来确保避免在堆上对其进行分配。

> **提示**
>
> 启用 .NET 代码样式规则 IDE0062 以确保始终标记本地函数 `static`。

> **备注**
>
> 等效于此方法的本地函数还将类用于闭包。 实现详细信息包括本地函数的闭包是作为 `class` 还是 `struct` 实现。 本地函数可能使用 `struct`，而 lambda 将始终使用 `class`。

C#

```
public async Task<string> PerformLongRunningWork(string address, int index,
string name)
{
    if (string.IsNullOrWhiteSpace(address))
        throw new ArgumentException(message: "An address is required",
paramName: nameof(address));
    if (index < 0)
        throw new ArgumentOutOfRangeException(paramName: nameof(index),
message: "The index must be non-negative");
    if (string.IsNullOrWhiteSpace(name))
        throw new ArgumentException(message: "You must supply a name",
paramName: nameof(name));

    return await longRunningWorkImplementation();

    async Task<string> longRunningWorkImplementation()
    {
        var interimResult = await FirstWork(address);
        var secondResult = await SecondStep(index, name);
        return $"The results are {interimResult} and {secondResult}.
Enjoy.";
    }
}
```

`yield` 关键字的用法

在本示例中尚未演示的最后一个优点是，可将本地函数作为迭代器实现，使用 `yield return` 语法生成一系列值。

```
C#
```

```
public IEnumerable<string> SequenceToLowercase(IEnumerable<string> input)
{
    if (!input.Any())
    {
        throw new ArgumentException("There are no items to convert to
lowercase.");
    }

    return LowercaseIterator();

    IEnumerable<string> LowercaseIterator()
    {
        foreach (var output in input.Select(item => item.ToLower()))
        {
            yield return output;
        }
    }
}
```

`yield return` 语句不允许在 lambda 表达式中使用。 有关详细信息，请参阅编译器错误 CS1621。

尽管本地函数看起来与 lambda 表达式相比显得多余，但它们实际上具有不同的目的和用途。 如果想要编写仅从上下文或其他方法中调用的函数，则使用本地函数更高效。

C# 语言规范

有关详细信息，请参阅 C# 语言规范中的本地函数声明部分。

另请参阅

- 使用本地函数而不是 Lambda（样式规则 IDE0039）
- 方法

隐式类型本地变量（C# 编程指南）

项目 • 2023/03/14

可声明局部变量而无需提供显式类型。 `var` 关键字指示编译器通过初始化语句右侧的表达式推断变量的类型。 推断类型可以是内置类型、匿名类型、用户定义类型或 .NET 类库中定义的类型。 有关如何使用 `var` 初始化数组的详细信息，请参阅隐式类型化数组。

以下示例演示使用 `var` 声明局部变量的各种方式：

C#

```
// i is compiled as an int
var i = 5;

// s is compiled as a string
var s = "Hello";

// a is compiled as int[]
var a = new[] { 0, 1, 2 };

// expr is compiled as IEnumerable<Customer>
// or perhaps IQueryable<Customer>
var expr =
    from c in customers
    where c.City == "London"
    select c;

// anon is compiled as an anonymous type
var anon = new { Name = "Terry", Age = 34 };

// list is compiled as List<int>
var list = new List<int>();
```

重要的是了解 `var` 关键字并不意味着“变体”，并且并不指示变量是松散类型或是后期绑定。 它只表示由编译器确定并分配最适合的类型。

在以下上下文中，可使用 `var` 关键字：

- 在局部变量（在方法范围内声明的变量）上，如前面的示例所示。
- 在 for 初始化语句中。

  C#

  ```
  for (var x = 1; x < 10; x++)
  ```

- 在 foreach 初始化语句中。

```C#
foreach (var item in list) {...}
```

- 在 using 域间中。

```C#
using (var file = new StreamReader("C:\\myfile.txt")) {...}
```

有关详细信息，请参阅如何在查询表达式中使用隐式类型化局部变量和数组。

var 和匿名类型

在许多情况下，使用 `var` 是可选的，只是一种语法便利。 但是，在使用匿名类型初始化变量时，如果需要在以后访问对象的属性，则必须将变量声明为 `var`。 这是 LINQ 查询表达式中的常见方案。 有关详细信息，请参阅匿名类型。

从源代码角度来看，匿名类型没有名称。 因此，如果使用 `var` 初始化了查询变量，则访问返回对象序列中的属性的唯一方法是在 `foreach` 语句中将 `var` 用作迭代变量的类型。

```C#
class ImplicitlyTypedLocals2
{
    static void Main()
    {
        string[] words = { "aPPLE", "BlUeBeRrY", "cHeRry" };

        // If a query produces a sequence of anonymous types,
        // then use var in the foreach statement to access the properties.
        var upperLowerWords =
             from w in words
             select new { Upper = w.ToUpper(), Lower = w.ToLower() };

        // Execute the query
        foreach (var ul in upperLowerWords)
        {
            Console.WriteLine("Uppercase: {0}, Lowercase: {1}", ul.Upper,
ul.Lower);
        }
    }
}
/* Outputs:
    Uppercase: APPLE, Lowercase: apple
    Uppercase: BLUEBERRY, Lowercase: blueberry
    Uppercase: CHERRY, Lowercase: cherry
 */
```

备注

以下限制适用于隐式类型化变量声明：

- 仅当局部变量在相同语句中进行声明和初始化时，才能使用 `var`；变量不能初始化为 null，也不能初始化为方法组或匿名函数。
- `var` 不能在类范围内对字段使用。
- 使用 `var` 声明的变量不能在初始化表达式中使用。 换句话说，此表达式是合法的：`int i = (i = 20);`，但是此表达式会生成编译时错误：`var i = (i = 20);`
- 不能在相同语句中初始化多个隐式类型化变量。
- 如果一种名为 `var` 的类型处于范围内，则 `var` 关键字会解析为该类型名称，不会被视为隐式类型化局部变量声明的一部分。

带 `var` 关键字的隐式类型只能应用于本地方法范围内的变量。 隐式类型不可用于类字段，因为 C# 编译器在处理代码时会遇到逻辑悖论：编译器需要知道字段的类型，但它在分析赋值表达式前无法确定类型，而表达式在不知道类型的情况下无法进行计算。 考虑下列代码：

```C#
private var bookTitles;
```

`bookTitles` 是类型为 `var` 的类字段。 由于该字段没有要计算的表达式，编译器无法推断出 `bookTitles` 应该是哪种类型。 此外，向该字段添加表达式（就像对本地变量执行的操作一样）也是不够的：

```C#
private var bookTitles = new List<string>();
```

当编译器在代码编译期间遇到字段时，它会在处理与其关联的任何表达式之前记录每个字段的类型。 编译器在尝试分析 `bookTitles` 时遇到相同的悖论：它需要知道字段的类型，但编译器通常会通过分析表达式来确定 `var` 的类型，这在事先不知道类型的情况下无法实现。

你可能会发现，对于在其中难以确定查询变量的确切构造类型的查询表达式，`var` 也可能会十分有用。 这可能会针对分组和排序操作发生。

当变量的特定类型在键盘上键入时很繁琐、或是显而易见、或是不会提高代码的可读性时，`var` 关键字也可能非常有用。 `var` 采用此方法提供帮助的一个示例是针对嵌套泛型类型（如用于分组操作的类型）。 在下面的查询中，查询变量的类型是 `IEnumerable<IGrouping<string, Student>>`。 只要你和必须维护你的代码的其他人了解这一点，使用隐式类型化实现便利性和简便性时便不会出现问题。

```
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;
```

使用 `var` 有助于简化代码，但是它的使用应该限制在需要使用它的情况下，或在它可使代码更易于读取的情况下。 有关何时正确使用 `var` 的详细信息，请参阅 C# 编码指南一文中的隐式类型本地变量节。

请参阅

- C# 参考
- 隐式类型化数组
- 如何在查询表达式中使用隐式类型化局部变量和数组
- 匿名类型
- 对象和集合初始值设定项
- var
- C# 中的 LINQ
- LINQ（语言集成查询）
- 迭代语句
- using 语句

如何在查询表达式中使用隐式类型的局部变量和数组（C# 编程指南）

项目 • 2024/03/13

每当需要编译器确定本地变量类型时，均可使用隐式类型本地变量。 必须使用隐式类型本地变量来存储匿名类型，匿名类型通常用于查询表达式中。 以下示例说明了在查询中可以使用和必须使用隐式类型本地变量的情况。

隐式类型本地变量使用 var 上下文关键字进行声明。 有关详细信息，请参阅隐式类型本地变量和隐式类型数组。

示例

以下示例演示必须使用 `var` 关键字的常见情景：用于生成一系列匿名类型的查询表达式。 在此情景中，必须使用 `var` 隐式类型化 `foreach` 语句中的查询变量和迭代变量，因为你无权访问匿名类型的类型名称。 有关匿名类型的详细信息，请参阅匿名类型。

C#

```
private static void QueryNames(char firstLetter)
{
    // Create the query. Use of var is required because
    // the query produces a sequence of anonymous types:
    // System.Collections.Generic.IEnumerable<????>.
    var studentQuery =
        from student in students
        where student.FirstName[0] == firstLetter
        select new { student.FirstName, student.LastName };

    // Execute the query and display the results.
    foreach (var anonType in studentQuery)
    {
        Console.WriteLine("First = {0}, Last = {1}", anonType.FirstName,
anonType.LastName);
    }
}
```

虽然以下示例在类似的情景中使用 `var` 关键字，但使用 `var` 是可选项。 因为 `student.LastName` 是字符串，所以执行查询将返回一系列字符串。 因此，`queryId` 的类型可声明为 `System.Collections.Generic.IEnumerable<string>`，而不是 `var`。 为方便起见，请使用关键字 `var`。 在示例中，虽然 `foreach` 语句中的迭代变量被显式类型化为字符串，但可改为使用 `var` 对其进行声明。 因为迭代变量的类型不是匿名类型，因此使用

`var` 是可选项，而不是必需项。 请记住，`var` 本身不是类型，而是发送给编译器用于推断和分配类型的指令。

C#

```
// Variable queryId could be declared by using
// System.Collections.Generic.IEnumerable<string>
// instead of var.
var queryId =
    from student in students
    where student.Id > 111
    select student.LastName;

// Variable str could be declared by using var instead of string.
foreach (string str in queryId)
{
    Console.WriteLine("Last name: {0}", str);
}
```

另请参阅

- 扩展方法
- LINQ（语言集成查询）
- 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

扩展方法（C# 编程指南）

项目 • 2024/03/15

扩展方法使你能够向现有类型“添加”方法，而无需创建新的派生类型、重新编译或以其他方式修改原始类型。 扩展方法是一种静态方法，但可以像扩展类型上的实例方法一样进行调用。 对于用 C#、F# 和 Visual Basic 编写的客户端代码，调用扩展方法与调用在类型中定义的方法没有明显区别。

最常见的扩展方法是 LINQ 标准查询运算符，它将查询功能添加到现有的 System.Collections.IEnumerable 和 System.Collections.Generic.IEnumerable<T> 类型。 若要使用标准查询运算符，请先使用 `using System.Linq` 指令将它们置于范围中。 然后，任何实现了 IEnumerable<T> 的类型看起来都具有 GroupBy、OrderBy、Average 等实例方法。 在 IEnumerable<T> 类型的实例（如 List<T> 或 Array）后键入“dot”时，可以在 IntelliSense 语句完成中看到这些附加方法。

OrderBy 示例

下面的示例演示如何对一个整数数组调用标准查询运算符 `OrderBy` 方法。 括号里面的表达式是一个 lambda 表达式。 很多标准查询运算符采用 Lambda 表达式作为参数，但这不是扩展方法的必要条件。 有关详细信息，请参阅 Lambda 表达式。

C#

```
class ExtensionMethods2
{

    static void Main()
    {
        int[] ints = [10, 45, 15, 39, 21, 26];
        var result = ints.OrderBy(g => g);
        foreach (var i in result)
        {
            System.Console.Write(i + " ");
        }
    }
}
//Output: 10 15 21 26 39 45
```

扩展方法被定义为静态方法，但它们是通过实例方法语法进行调用的。 它们的第一个参数指定方法操作的类型。 该参数位于 this 修饰符之后。 仅当你使用 `using` 指令将命名空间显式导入到源代码中之后，扩展方法才位于范围中。

下面的示例演示为 System.String 类定义的一个扩展方法。 它是在非嵌套的、非泛型静态类内部定义的：

C#

```
namespace ExtensionMethods
{
    public static class MyExtensions
    {
        public static int WordCount(this string str)
        {
            return str.Split(new char[] { ' ', '.', '?' },
                             StringSplitOptions.RemoveEmptyEntries).Length;
        }
    }
}
```

可使用此 `WordCount` 指令将 `using` 扩展方法置于范围中：

C#

```
using ExtensionMethods;
```

而且，可以使用以下语法从应用程序中调用该扩展方法：

C#

```
string s = "Hello Extension Methods";
int i = s.WordCount();
```

在代码中，可以使用实例方法语法调用该扩展方法。 编译器生成的中间语言 (IL) 会将代码转换为对静态方法的调用。 并未真正违反封装原则。 扩展方法无法访问它们所扩展的类型中的专用变量。

`MyExtensions` 类和 `WordCount` 方法都是 `static`，可以像所有其他 `static` 成员那样对其进行访问。 `WordCount` 方法可以像其他 `static` 方法一样调用，如下所示：

C#

```
string s = "Hello Extension Methods";
int i = MyExtensions.WordCount(s);
```

上述 C# 代码：

- 声明并分配一个名为 `s` 和值为 `"Hello Extension Methods"` 的新 `string`。
- 调用 `MyExtensions.WordCount` 给定自变量 `s`。

有关详细信息，请参阅如何实现和调用自定义扩展方法。

通常，你更多时候是调用扩展方法而不是实现你自己的扩展方法。 由于扩展方法是使用实例方法语法调用的，因此不需要任何特殊知识即可从客户端代码中使用它们。 若要为特定类型启用扩展方法，只需为在其中定义这些方法的命名空间添加 `using` 指令。 例如，若要使用标准查询运算符，请将此 `using` 指令添加到代码中：

```
C#

using System.Linq;
```

（你可能还必须添加对 System.Core.dll 的引用。）你将注意到，标准查询运算符现在作为可供大多数 IEnumerable<T> 类型使用的附加方法显示在 IntelliSense 中。

在编译时绑定扩展方法

可以使用扩展方法来扩展类或接口，但不能重写扩展方法。 与接口或类方法具有相同名称和签名的扩展方法永远不会被调用。 编译时，扩展方法的优先级总是比类型本身中定义的实例方法低。 换句话说，如果某个类型具有一个名为 `Process(int i)` 的方法，而你有一个具有相同签名的扩展方法，则编译器总是绑定到该实例方法。 当编译器遇到方法调用时，它首先在该类型的实例方法中寻找匹配的方法。 如果未找到任何匹配方法，编译器将搜索为该类型定义的任何扩展方法，并且绑定到它找到的第一个扩展方法。

示例

下面的示例演示 C# 编译器在确定是将方法调用绑定到类型上的实例方法还是绑定到扩展方法时所遵循的规则。 静态类 `Extensions` 包含为任何实现了 `IMyInterface` 的类型定义的扩展方法。 类 `A`、`B` 和 `C` 都实现了该接口。

`MethodB` 扩展方法永远不会被调用，因为它的名称和签名与这些类已经实现的方法完全匹配。

如果编译器找不到具有匹配签名的实例方法，它会绑定到匹配的扩展方法（如果存在这样的方法）。

```
C#

// Define an interface named IMyInterface.
namespace DefineIMyInterface
{
    public interface IMyInterface
    {
        // Any class that implements IMyInterface must define a method
        // that matches the following signature.
        void MethodB();
```

```
    }
}

// Define extension methods for IMyInterface.
namespace Extensions
{
    using System;
    using DefineIMyInterface;

    // The following extension methods can be accessed by instances of any
    // class that implements IMyInterface.
    public static class Extension
    {
        public static void MethodA(this IMyInterface myInterface, int i)
        {
            Console.WriteLine
                ("Extension.MethodA(this IMyInterface myInterface, int i)");
        }

        public static void MethodA(this IMyInterface myInterface, string s)
        {
            Console.WriteLine
                ("Extension.MethodA(this IMyInterface myInterface, string
s)");
        }

        // This method is never called in ExtensionMethodsDemo1, because
each
        // of the three classes A, B, and C implements a method named
MethodB
        // that has a matching signature.
        public static void MethodB(this IMyInterface myInterface)
        {
            Console.WriteLine
                ("Extension.MethodB(this IMyInterface myInterface)");
        }
    }
}

// Define three classes that implement IMyInterface, and then use them to
test
// the extension methods.
namespace ExtensionMethodsDemo1
{
    using System;
    using Extensions;
    using DefineIMyInterface;

    class A : IMyInterface
    {
        public void MethodB() { Console.WriteLine("A.MethodB()"); }
    }

    class B : IMyInterface
    {
```

```
        public void MethodB() { Console.WriteLine("B.MethodB()"); }
        public void MethodA(int i) { Console.WriteLine("B.MethodA(int i)");
}
    }

    class C : IMyInterface
    {
        public void MethodB() { Console.WriteLine("C.MethodB()"); }
        public void MethodA(object obj)
        {
            Console.WriteLine("C.MethodA(object obj)");
        }
    }

    class ExtMethodDemo
    {
        static void Main(string[] args)
        {
            // Declare an instance of class A, class B, and class C.
            A a = new A();
            B b = new B();
            C c = new C();

            // For a, b, and c, call the following methods:
            //      -- MethodA with an int argument
            //      -- MethodA with a string argument
            //      -- MethodB with no argument.

            // A contains no MethodA, so each call to MethodA resolves to
            // the extension method that has a matching signature.
            a.MethodA(1);           // Extension.MethodA(IMyInterface, int)
            a.MethodA("hello");     // Extension.MethodA(IMyInterface,
string)

            // A has a method that matches the signature of the following
call
            // to MethodB.
            a.MethodB();            // A.MethodB()

            // B has methods that match the signatures of the following
            // method calls.
            b.MethodA(1);           // B.MethodA(int)
            b.MethodB();            // B.MethodB()

            // B has no matching method for the following call, but
            // class Extension does.
            b.MethodA("hello");     // Extension.MethodA(IMyInterface,
string)

            // C contains an instance method that matches each of the
following
            // method calls.
            c.MethodA(1);           // C.MethodA(object)
            c.MethodA("hello");     // C.MethodA(object)
            c.MethodB();            // C.MethodB()
```

```
        }
    }
}
/* Output:
    Extension.MethodA(this IMyInterface myInterface, int i)
    Extension.MethodA(this IMyInterface myInterface, string s)
    A.MethodB()
    B.MethodA(int i)
    B.MethodB()
    Extension.MethodA(this IMyInterface myInterface, string s)
    C.MethodA(object obj)
    C.MethodA(object obj)
    C.MethodB()
 */
```

常见使用模式

集合功能

过去，创建"集合类"通常是为了使给定类型实现 System.Collections.Generic.IEnumerable<T> 接口，并实现对该类型集合的功能。 创建这种类型的集合对象没有任何问题，但也可以通过对 System.Collections.Generic.IEnumerable<T> 使用扩展来实现相同的功能。 扩展的优势是允许从任何集合（如 System.Array 或实现该类型 System.Collections.Generic.IEnumerable<T> 的 System.Collections.Generic.List<T>）调用功能。 可以在本文前面的内容中找到使用 Int32 的数组的示例。

特定于层的功能

使用洋葱架构或其他分层应用程序设计时，通常具有一组域实体或数据传输对象，可用于跨应用程序边界进行通信。 这些对象通常不包含任何功能，或者只包含适用于应用程序的所有层的最少功能。 使用扩展方法可以添加特定于每个应用程序层的功能，而无需使用其他层中不需要的方法来向下加载对象。

C#

```
public class DomainEntity
{
    public int Id { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }
}

static class DomainEntityExtensions
{
    static string FullName(this DomainEntity value)
```

```
        => $"{value.FirstName} {value.LastName}";
}
```

扩展预定义类型

当需要创建可重用功能时，我们无需创建新对象，而是可以扩展现有类型，例如 .NET 或 CLR 类型。 例如，如果不使用扩展方法，我们可能会创建 `Engine` 或 `Query` 类，对可从代码中的多个位置调用的 SQL Server 执行查询。 但是，如果换做使用扩展方法扩展 System.Data.SqlClient.SqlConnection 类，就可以从与 SQL Server 连接的任何位置执行该查询。 其他示例可能是向 System.String 类添加常见功能、扩展 System.IO.Stream 和 System.Exception 对象的数据处理功能以实现特定的错误处理功能。 这些用例的类型仅受想象力和判断力的限制。

使用 `struct` 类型扩展预定义类型可能很困难，因为它们已通过值传递给方法。 这意味着将对结构的副本进行任何结构更改。 扩展方法退出后，将不显示这些更改。 可以将 `ref` 修饰符添加到第一个参数，使其成为 `ref` 扩展方法。 `ref` 关键字可以在 `this` 关键字之前或之后显示，不会有任何语义差异。 添加 `ref` 修饰符表示第一个参数是按引用传递的。 在这种情况下，可以编写扩展方法来更改要扩展的结构的状态（请注意，私有成员不可访问）。 仅允许值类型或受结构约束的泛型类型（有关详细信息，请参阅 struct 约束）作为 `ref` 扩展方法的第一个参数。 以下示例演示如何使用 `ref` 扩展方法直接修改内置类型，而无需重新分配结果或使用 `ref` 关键字传递函数：

C#

```
public static class IntExtensions
{
    public static void Increment(this int number)
        => number++;

    // Take note of the extra ref keyword here
    public static void RefIncrement(this ref int number)
        => number++;
}

public static class IntProgram
{
    public static void Test()
    {
        int x = 1;

        // Takes x by value leading to the extension method
        // Increment modifying its own copy, leaving x unchanged
        x.Increment();
        Console.WriteLine($"x is now {x}"); // x is now 1

        // Takes x by reference leading to the extension method
```

```
        // RefIncrement changing the value of x directly
        x.RefIncrement();
        Console.WriteLine($"x is now {x}"); // x is now 2
    }
}
```

下一个示例演示用户定义的结构类型的 `ref` 扩展方法：

C#

```
public struct Account
{
    public uint id;
    public float balance;

    private int secret;
}

public static class AccountExtensions
{
    // ref keyword can also appear before the this keyword
    public static void Deposit(ref this Account account, float amount)
    {
        account.balance += amount;

        // The following line results in an error as an extension
        // method is not allowed to access private members
        // account.secret = 1; // CS0122
    }
}

public static class AccountProgram
{
    public static void Test()
    {
        Account account = new()
        {
            id = 1,
            balance = 100f
        };

        Console.WriteLine($"I have ${account.balance}"); // I have $100

        account.Deposit(50f);
        Console.WriteLine($"I have ${account.balance}"); // I have $150
    }
}
```

通用准则

尽管通过修改对象的代码来添加功能，或者在合理和可行的情况下派生新类型等方式仍是可取的，但扩展方法已成为在整个 .NET 生态系统中创建可重用功能的关键选项。 对于原始源不受控制、派生对象不合适或不可用，或者不应在功能适用范围之外公开功能的情况，扩展方法是一个不错的选择。

有关派生类型的详细信息，请参阅继承。

在使用扩展方法来扩展你无法控制其源代码的类型时，你需要承受该类型实现中的更改会导致扩展方法失效的风险。

如果确实为给定类型实现了扩展方法，请记住以下几点：

- 如果扩展方法与该类型中定义的方法具有相同的签名，则扩展方法永远不会被调用。
- 在命名空间级别将扩展方法置于范围中。 例如，如果你在一个名为 `Extensions` 的命名空间中具有多个包含扩展方法的静态类，则这些扩展方法将全部由 `using Extensions;` 指令置于范围中。

针对已实现的类库，不应为了避免程序集的版本号递增而使用扩展方法。 如果要向你拥有源代码的库中添加重要功能，请遵循适用于程序集版本控制的 .NET 准则。 有关详细信息，请参阅程序集版本控制。

另请参阅

- 并行编程示例（这些示例包括许多示例扩展方法）
- Lambda 表达式
- 标准查询运算符概述
- Conversion rules for Instance parameters and their impact（实例参数及其影响的转换规则）
- Extension methods Interoperability between languages（语言间扩展方法的互操作性）
- Extension methods and Curried Delegates（扩展方法和扩充委托）
- Extension method Binding and Error reporting（扩展方法绑定和错误报告）

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 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

如何实现和调用自定义扩展方法（C# 编程指南）

项目 • 2024/10/04

本文将介绍如何为任意 .NET 类型实现自定义扩展方法。 客户端代码可以使用扩展方法。 客户端项目必须引用包含它们的程序集。 客户端项目必须添加一个 using 指令，该指令指定在其中定义扩展方法的命名空间。

定义和调用扩展方法：

1. 定义包含扩展方法的静态类。 该类不能嵌套在另一个类型内，并且必须对客户端代码可见。 有关可访问性规则的详细信息，请参阅访问修饰符。
2. 将扩展方法实现为静态方法，并且使其可见性至少与所在类的可见性相同。
3. 此方法的第一个参数指定方法所操作的类型；此参数前面必须加上 this 修饰符。
4. 在调用代码中，添加 `using` 指令，用于指定包含扩展方法类的命名空间。
5. 将方法作为类型的实例方法调用。

> **备注**
>
> 第一个参数并不是由调用代码指定，因为它表示要在其上应用运算符的类型，并且编译器已经知道对象的类型。 你只需通过 `n` 提供形参 2 的实参。

以下示例实现 `CustomExtensions.StringExtension` 类中名为 `WordCount` 的扩展方法。 此方法对 String 类进行操作，该类指定为第一个方法参数。 将 `CustomExtensions` 命名空间导入应用程序命名空间，并在 `Main` 方法内部调用此方法。

C#

```
using CustomExtensions;

string s = "The quick brown fox jumped over the lazy dog.";
// Call the method as if it were an
// instance method on the type. Note that the first
// parameter is not specified by the calling code.
int i = s.WordCount();
System.Console.WriteLine("Word count of s is {0}", i);


namespace CustomExtensions
{
    // Extension methods must be defined in a static class.
    public static class StringExtension
    {
        // This is the extension method.
```

```
        // The first parameter takes the "this" modifier
        // and specifies the type for which the method is defined.
        public static int WordCount(this string str)
        {
            return str.Split(new char[] {' ', '.','?'},
StringSplitOptions.RemoveEmptyEntries).Length;
        }
    }
}
```

重载解析首选类型本身定义的实例或静态方法，而不是扩展方法。 扩展方法无法访问扩展类中的任何隐私数据。

另请参阅

- 扩展方法
- LINQ（语言集成查询）
- 静态类和静态类成员
- 受保护
- internal
- public
- this
- namespace

如何为枚举创建新方法（C# 编程指南）

项目 • 2024/03/13

可使用扩展方法添加特定于某个特定枚举类型的功能。

示例

在下面的示例中，`Grades` 枚举表示学生可能在班里收到的字母等级分。 该示例将一个名为 `Passing` 的扩展方法添加到 `Grades` 类型中，以便该类型的每个实例现在都“知道”它是否表示合格的等级分。

C#

```
    using System;

    namespace EnumExtension
    {
        // Define an extension method in a non-nested static class.
        public static class Extensions
    {
        public static Grades minPassing = Grades.D;
        public static bool Passing(this Grades grade)
        {
            return grade >= minPassing;
        }
    }

    public enum Grades { F = 0, D=1, C=2, B=3, A=4 };
    class Program
    {
        static void Main(string[] args)
        {
            Grades g1 = Grades.D;
            Grades g2 = Grades.F;
            Console.WriteLine("First {0} a passing grade.", g1.Passing() ?
"is" : "is not");
            Console.WriteLine("Second {0} a passing grade.", g2.Passing() ?
"is" : "is not");

            Extensions.minPassing = Grades.C;
            Console.WriteLine("\r\nRaising the bar!\r\n");
            Console.WriteLine("First {0} a passing grade.", g1.Passing() ?
"is" : "is not");
            Console.WriteLine("Second {0} a passing grade.", g2.Passing() ?
"is" : "is not");
        }
    }
  }
/* Output:
```

```
    First is a passing grade.
    Second is not a passing grade.

    Raising the bar!

    First is not a passing grade.
    Second is not a passing grade.
*/
```

请注意，`Extensions` 类还包含一个动态更新的静态变量，并且扩展方法的返回值反映了该变量的当前值。 这表明在幕后，将在定义扩展方法的静态类上直接调用这些方法。

另请参阅

- 扩展方法

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 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

命名实参和可选实参（C# 编程指南）

项目 • 2024/03/20

通过命名实参，你可以为形参指定实参，方法是将实参与该形参的名称匹配，而不是与形参在形参列表中的位置匹配。 通过*可选参数*，你可以为某些形参省略实参。 这两种技术都可与方法、索引器、构造函数和委托一起使用。

使用命名参数和可选参数时，将按实参出现在实参列表（而不是形参列表）中的顺序计算这些实参。

通过命名形参和可选形参，你可以为所选形参提供实参。 此功能极大地方便了对 COM 接口（例如 Microsoft Office 自动化 API）的调用。

命名参数

有了命名实参，你将不再需要将实参的顺序与所调用方法的形参列表中的形参顺序相匹配。 每个形参的实参都可按形参名称进行指定。 例如，通过以函数定义的顺序按位置发送实参，可以调用打印订单详细信息（例如卖家姓名、订单号和产品名称）的函数。

```
C#

PrintOrderDetails("Gift Shop", 31, "Red Mug");
```

如果不记得形参的顺序，但却知道其名称，则可以按任意顺序发送实参。

```
C#

PrintOrderDetails(orderNum: 31, productName: "Red Mug", sellerName: "Gift
Shop");
PrintOrderDetails(productName: "Red Mug", sellerName: "Gift Shop", orderNum:
31);
```

命名实参还可以标识每个实参所表示的含义，从而改进代码的可读性。 在下面的示例方法中，`sellerName` 不得为 NULL 或空白符。 由于 `sellerName` 和 `productName` 都是字符串类型，所以使用命名实参而不是按位置发送实参是有意义的，可以区分这两种类型并减少代码阅读者的困惑。

当命名实参与位置实参一起使用时，只要

- 没有后接任何位置实参或

  ```
  C#
  ```

```
PrintOrderDetails("Gift Shop", 31, productName: "Red Mug");
```

- 它们用在正确位置。 在以下示例中，形参 `orderNum` 位于正确的位置，但未显式命名。

  C#

  ```
  PrintOrderDetails(sellerName: "Gift Shop", 31, productName: "Red Mug");
  ```

遵循任何无序命名参数的位置参数无效。

C#

```
// This generates CS1738: Named argument specifications must appear after
all fixed arguments have been specified.
PrintOrderDetails(productName: "Red Mug", 31, "Gift Shop");
```

示例

以下代码执行本节以及某些其他节中的示例。

C#

```
class NamedExample
{
    static void Main(string[] args)
    {
        // The method can be called in the normal way, by using positional
arguments.
        PrintOrderDetails("Gift Shop", 31, "Red Mug");

        // Named arguments can be supplied for the parameters in any order.
        PrintOrderDetails(orderNum: 31, productName: "Red Mug", sellerName:
"Gift Shop");
        PrintOrderDetails(productName: "Red Mug", sellerName: "Gift Shop",
orderNum: 31);

        // Named arguments mixed with positional arguments are valid
        // as long as they are used in their correct position.
        PrintOrderDetails("Gift Shop", 31, productName: "Red Mug");
        PrintOrderDetails(sellerName: "Gift Shop", 31, productName: "Red
Mug");
        PrintOrderDetails("Gift Shop", orderNum: 31, "Red Mug");

        // However, mixed arguments are invalid if used out-of-order.
        // The following statements will cause a compiler error.
        // PrintOrderDetails(productName: "Red Mug", 31, "Gift Shop");
```

```
        // PrintOrderDetails(31, sellerName: "Gift Shop", "Red Mug");
        // PrintOrderDetails(31, "Red Mug", sellerName: "Gift Shop");
    }

    static void PrintOrderDetails(string sellerName, int orderNum, string
productName)
    {
        if (string.IsNullOrWhiteSpace(sellerName))
        {
            throw new ArgumentException(message: "Seller name cannot be null
or empty.", paramName: nameof(sellerName));
        }

        Console.WriteLine($"Seller: {sellerName}, Order #: {orderNum},
Product: {productName}");
    }
}
```

可选自变量

方法、构造函数、索引器或委托的定义可以指定其形参为必需还是可选。 任何调用都必须为所有必需的形参提供实参，但可以为可选的形参省略实参。

每个可选形参都有一个默认值作为其定义的一部分。 如果没有为该形参发送实参，则使用默认值。 默认值必须是以下类型的表达式之一：

- 常量表达式；
- `new ValType()` 形式的表达式，其中 `ValType` 是值类型，例如 enum 或 struct；
- default(ValType) 形式的表达式，其中 `ValType` 是值类型。

可选参数定义于参数列表的末尾和必需参数之后。 如果调用方为一系列可选形参中的任意一个形参提供了实参，则它必须为前面的所有可选形参提供实参。 实参列表中不支持使用逗号分隔的间隔。 例如，在以下代码中，使用一个必选形参和两个可选形参定义实例方法 `ExampleMethod`。

C#

```
public void ExampleMethod(int required, string optionalstr = "default
string",
    int optionalint = 10)
```

下面对 `ExampleMethod` 的调用会导致编译器错误，原因是为第三个形参而不是为第二个形参提供了实参。

C#

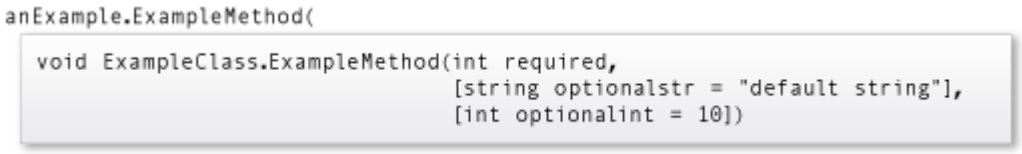
```
//anExample.ExampleMethod(3, ,4);
```

但是，如果知道第三个形参的名称，则可以使用命名实参来完成此任务。

C#

```
anExample.ExampleMethod(3, optionalint: 4);
```

IntelliSense 使用括号表示可选形参，如下图所示：



> **备注**
>
> 此外，还可通过使用 .NET OptionalAttribute 类声明可选参数。 `OptionalAttribute` 形参不需要默认值。 但是，如果需要默认值，请查看 DefaultParameterValueAttribute 类。

示例

在以下示例中，`ExampleClass` 的构造函数具有一个可选形参。 实例方法 `ExampleMethod` 具有一个必选形参（`required`）和两个可选形参（`optionalstr` 和 `optionalint`）。 `Main` 中的代码演示了可用于调用构造函数和方法的不同方式。

C#

```
namespace OptionalNamespace
{
    class OptionalExample
    {
        static void Main(string[] args)
        {
            // Instance anExample does not send an argument for the
constructor's
            // optional parameter.
            ExampleClass anExample = new ExampleClass();
            anExample.ExampleMethod(1, "One", 1);
            anExample.ExampleMethod(2, "Two");
            anExample.ExampleMethod(3);

            // Instance anotherExample sends an argument for the
constructor's
```

```
            // optional parameter.
            ExampleClass anotherExample = new ExampleClass("Provided name");
            anotherExample.ExampleMethod(1, "One", 1);
            anotherExample.ExampleMethod(2, "Two");
            anotherExample.ExampleMethod(3);

            // The following statements produce compiler errors.

            // An argument must be supplied for the first parameter, and it
            // must be an integer.
            //anExample.ExampleMethod("One", 1);
            //anExample.ExampleMethod();

            // You cannot leave a gap in the provided arguments.
            //anExample.ExampleMethod(3, ,4);
            //anExample.ExampleMethod(3, 4);

            // You can use a named parameter to make the previous
            // statement work.
            anExample.ExampleMethod(3, optionalint: 4);
        }
    }

    class ExampleClass
    {
        private string _name;

        // Because the parameter for the constructor, name, has a default
        // value assigned to it, it is optional.
        public ExampleClass(string name = "Default name")
        {
            _name = name;
        }

        // The first parameter, required, has no default value assigned
        // to it. Therefore, it is not optional. Both optionalstr and
        // optionalint have default values assigned to them. They are
optional.
        public void ExampleMethod(int required, string optionalstr =
"default string",
            int optionalint = 10)
        {
            Console.WriteLine(
                $"{_name}: {required}, {optionalstr}, and {optionalint}.");
        }
    }

    // The output from this example is the following:
    // Default name: 1, One, and 1.
    // Default name: 2, Two, and 10.
    // Default name: 3, default string, and 10.
    // Provided name: 1, One, and 1.
    // Provided name: 2, Two, and 10.
    // Provided name: 3, default string, and 10.
```

```
    // Default name: 3, default string, and 4.
}
```

前面的代码演示了一些示例，其中不会正确应用可选形参。 第一个示例说明了必须为第一个形参提供实参，这是必需的。

调用方信息特性

调用方信息属性（例如 CallerFilePathAttribute、CallerLineNumberAttribute、CallerMemberNameAttribute 和 CallerArgumentExpressionAttribute）用于获取方法调用方的相关信息。 在调试或需要记录有关方法调用的信息时，这些属性特别有用。

这些属性是具有编译器提供的默认值的可选参数。 调用方不应为这些参数显式提供值。

COM 接口

命名实参和可选实参，以及对动态对象的支持大大提高了与 COM API（例如 Office Automation API）的互操作性。

例如，Microsoft Office Excel 的 Range 接口中的 AutoFormat 方法有七个可选形参。 这些形参如下图所示：

```
excelApp.get_Range("A1", "B4").AutoFormat(
```

```
dynamic Range.AutoFormat([Excel.XlRangeAutoFormat Format = 1],
[object Number = Type.Missing], [object Font = Type.Missing],
[object Alignment = Type.Missing], [object Border = Type.Missing],
[object Pattern = Type.Missing], [object Width = Type.Missing])
```

但是，可以通过使用命名实参和可选实参来大大简化对 `AutoFormat` 的调用。 如果不希望更改形参的默认值，则可以通过使用命名实参和可选实参来省略可选形参的实参。 在下面的调用中，仅为 7 个形参中的其中一个指定了值。

C#

```
var excelApp = new Microsoft.Office.Interop.Excel.Application();
excelApp.Workbooks.Add();
excelApp.Visible = true;

var myFormat =

Microsoft.Office.Interop.Excel.XlRangeAutoFormat.xlRangeAutoFormatAccounting
1;

excelApp.Range["A1", "B4"].AutoFormat( Format: myFormat );
```

有关详细信息和示例，请参阅如何在 Office 编程中使用命名参数和可选参数和如何使用 C# 功能访问 Office 互操作性对象。

重载决策

使用命名实参和可选实参将在以下方面对重载决策产生影响：

- 如果方法、索引器或构造函数的每个参数是可选的，或按名称或位置对应于调用语句中的单个自变量，且该自变量可转换为参数的类型，则方法、索引器或构造函数为执行的候选项。
- 如果找到多个候选项，则会将用于首选转换的重载决策规则应用于显式指定的自变量。 将忽略可选形参已省略的实参。
- 如果两个候选项不相上下，则会将没有可选形参的候选项作为首选项，对于这些可选形参，已在调用中为其省略了实参。 重载决策通常首选具有较少形参的候选项。

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。

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

构造函数（C# 编程指南）

项目 • 2023/04/07

每当创建类或结构的实例时，将会调用其构造函数。 类或结构可能具有采用不同参数的多个构造函数。 使用构造函数，程序员能够设置默认值、限制实例化，并编写灵活易读的代码。 有关详细信息和示例，请参阅实例构造函数和使用构造函数。

有多个操作在初始化新实例时进行。 这些操作按以下顺序执行：

1. 实例字段设置为 0。 这通常由运行时来完成。
2. 字段初始值设定项运行。 派生程度最高类型的字段初始值设定项运行。
3. 基类型字段初始值设定项运行。 以直接基开头从每个基类型到 System.Object 的字段初始值设定项。
4. 基实例构造函数运行。 以 Object.Object 开头从每个基类到直接基类的任何实例构造函数。
5. 实例构造函数运行。 该类型的实例构造函数运行。
6. 对象初始值设定项运行。 如果表达式包含任何对象初始值设定项，后者会在实例构造函数运行后运行。 对象初始值设定项按文本顺序运行。

初始化新实例时，将执行上述操作。 如果 `struct` 的新实例设置为其 `default` 值，则所有实例字段都设置为 0。

如果静态构造函数尚未运行，静态构造函数会在任何实例构造函数操作执行之前运行。

构造函数语法

构造函数是一种方法，其名称与其类型的名称相同。 其方法签名仅包含可选访问修饰符、方法名称和其参数列表；它不包含返回类型。 以下示例演示一个名为 `Person` 的类的构造函数。

C#

```
public class Person
{
   private string last;
   private string first;

   public Person(string lastName, string firstName)
   {
      last = lastName;
      first = firstName;
   }
```

```
    // Remaining implementation of Person class.
}
```

如果某个构造函数可以作为单个语句实现，则可以使用表达式主体定义。 以下示例定义 `Location` 类，其构造函数具有一个名为“name”的字符串参数。 表达式主体定义给 `locationName` 字段分配参数。

C#

```
public class Location
{
   private string locationName;

   public Location(string name) => Name = name;

   public string Name
   {
      get => locationName;
      set => locationName = value;
   }
}
```

静态构造函数

前面的示例具有所有已展示的实例构造函数，这些构造函数创建一个新对象。 类或结构也可以具有静态构造函数，该静态构造函数初始化类型的静态成员。 静态构造函数是无参数构造函数。 如果未提供静态构造函数来初始化静态字段，C# 编译器会将静态字段初始化为其默认值，如 C# 类型的默认值中所列。

以下示例使用静态构造函数来初始化静态字段。

C#

```
public class Adult : Person
{
   private static int minimumAge;

   public Adult(string lastName, string firstName) : base(lastName,
firstName)
   { }

   static Adult()
   {
      minimumAge = 18;
   }

   // Remaining implementation of Adult class.
}
```

也可以通过表达式主体定义来定义静态构造函数，如以下示例所示。

```C#
public class Child : Person
{
   private static int maximumAge;

   public Child(string lastName, string firstName) : base(lastName,
firstName)
   { }

   static Child() => maximumAge = 18;

   // Remaining implementation of Child class.
}
```

有关详细信息和示例，请参阅静态构造函数。

本节内容

使用构造函数

实例构造函数

私有构造函数

静态构造函数

如何编写复制构造函数

请参阅

- C# 编程指南
- C# 类型系统
- 终结器
- static
- Why Do Initializers Run In The Opposite Order As Constructors? Part One（为何初始值设定项作为构造函数以相反顺序运行？第一部分）

使用构造函数（C# 编程指南）

项目 • 2023/06/01

实例化类或结构时，将会调用其构造函数。 构造函数与该类或结构具有相同名称，并且通常初始化新对象的数据成员。

在下面的示例中，通过使用简单构造函数定义了一个名为 `Taxi` 的类。 然后使用 new 运算符对该类进行实例化。 在为新对象分配内存之后，`new` 运算符立即调用 `Taxi` 构造函数。

C#

```
public class Taxi
{
    public bool IsInitialized;

    public Taxi()
    {
        IsInitialized = true;
    }
}

class TestTaxi
{
    static void Main()
    {
        Taxi t = new Taxi();
        Console.WriteLine(t.IsInitialized);
    }
}
```

不带任何参数的构造函数称为“无参数构造函数”。 每当使用 `new` 运算符实例化对象且不为 `new` 提供任何参数时，会调用无参数构造函数。 C# 12 引入了主构造函数。 主构造函数指定为初始化新对象而必须提供的参数。 有关详细信息，请参阅实例构造函数。

除非类是静态的，否则 C# 编译器将为无构造函数的类提供一个公共的无参数构造函数，以便该类可以实例化。 有关详细信息，请参阅静态类和静态类成员。

通过将构造函数设置为私有构造函数，可以阻止类被实例化，如下所示：

C#

```
class NLog
{
    // Private Constructor:
    private NLog() { }
```

```
    public static double e = Math.E;  //2.71828...
}
```

有关详细信息，请参阅私有构造函数。

结构类型的构造函数类似于类构造函数。 使用 `new` 实例化结构类型时，将调用构造函数。 将 `struct` 设置为其 `default` 值时，运行时会将结构中的所有内存初始化为 0。 在 C# 10 之前，`structs` 不能包含显式无参数构造函数，因为编译器会自动提供一个。 有关详细信息，请参阅结构类型一文的结构初始化和默认值部分。

以下代码使用 Int32 的无参数构造函数，因此可确保整数已初始化：

C#

```
int i = new int();
Console.WriteLine(i);
```

但是，下面的代码会导致编译器错误，因为它不使用 `new`，而且尝试使用尚未初始化的对象：

C#

```
int i;
Console.WriteLine(i);
```

或者，可将基于 `structs` 的对象（包括所有内置数值类型）初始化或赋值后使用，如下面的示例所示：

C#

```
int a = 44;  // Initialize the value type...
int b;
b = 33;      // Or assign it before using it.
Console.WriteLine("{0}, {1}", a, b);
```

类和结构都可以定义采用参数的构造函数，包括主构造函数。 必须通过 `new` 语句或 base 语句调用带参数的构造函数。 类和结构还可以定义多个构造函数，并且二者均无需定义无参数构造函数。 例如：

C#

```
public class Employee
{
    public int Salary;
```

```
    public Employee() { }

    public Employee(int annualSalary)
    {
        Salary = annualSalary;
    }

    public Employee(int weeklySalary, int numberOfWeeks)
    {
        Salary = weeklySalary * numberOfWeeks;
    }
}
```

可使用下面任一语句创建此类：

C#

```
Employee e1 = new Employee(30000);
Employee e2 = new Employee(500, 52);
```

构造函数可以使用 `base` 关键字调用基类的构造函数。 例如：

C#

```
public class Manager : Employee
{
    public Manager(int annualSalary)
        : base(annualSalary)
    {
        //Add further instructions here.
    }
}
```

在此示例中，在执行构造函数块之前调用基类的构造函数。 `base` 关键字可带参数使用，也可不带参数使用。 构造函数的任何参数都可用作 `base` 的参数，或用作表达式的一部分。 有关详细信息，请参阅 base。

在派生类中，如果不使用 `base` 关键字来显式调用基类构造函数，则将隐式调用无参数构造函数（若有）。 下面的构造函数声明等效：

C#

```
public Manager(int initialData)
{
    //Add further instructions here.
}
```

C#

```
public Manager(int initialData)
    : base()
{
    //Add further instructions here.
}
```

如果基类没有提供无参数构造函数，派生类必须使用 `base` 显式调用基类构造函数。

构造函数可以使用 this 关键字调用同一对象中的另一构造函数。 和 `base` 一样，`this` 可带参数使用也可不带参数使用，构造函数中的任何参数都可用作 `this` 的参数，或者用作表达式的一部分。 例如，可以使用 `this` 重写前一示例中的第二个构造函数：

C#

```
public Employee(int weeklySalary, int numberOfWeeks)
    : this(weeklySalary * numberOfWeeks)
{
}
```

上一示例中使用 `this` 关键字会导致此构造函数被调用：

C#

```
public Employee(int annualSalary)
{
    Salary = annualSalary;
}
```

可以将构造函数标记为public、private、protected、internal、protected internal 或 private protected。 这些访问修饰符定义类的用户构造该类的方式。 有关详细信息，请参阅访问修饰符。

可使用 static 关键字将构造函数声明为静态构造函数。 在访问任何静态字段之前，都将自动调用静态构造函数，它们用于初始化静态类成员。 有关详细信息，请参阅静态构造函数。

C# 语言规范

有关详细信息，请参阅 C# 语言规范中的实例构造函数和静态构造函数。 该语言规范是 C# 语法和用法的权威资料。

另请参阅

- C# 编程指南
- C# 类型系统
- 构造函数
- 终结器

实例构造函数（C# 编程指南）

项目 • 2024/06/03

声明一个实例构造函数，以指定在使用 new 表达式创建某个类型的新实例时所执行的代码。 要初始化静态类或非静态类中的静态变量，可以定义静态构造函数。

如以下示例所示，可以在一种类型中声明多个实例构造函数：

C#

```
class Coords
{
    public Coords()
        : this(0, 0)
    {  }

    public Coords(int x, int y)
    {
        X = x;
        Y = y;
    }

    public int X { get; set; }
    public int Y { get; set; }

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

class Example
{
    static void Main()
    {
        var p1 = new Coords();
        Console.WriteLine($"Coords #1 at {p1}");
        // Output: Coords #1 at (0,0)

        var p2 = new Coords(5, 3);
        Console.WriteLine($"Coords #2 at {p2}");
        // Output: Coords #2 at (5,3)
    }
}
```

在上个示例中，第一个无参数构造函数调用两个参数都等于 `0` 的第二个构造函数。 要执行此操作，请使用 `this` 关键字。

在派生类中声明实例构造函数时，可以调用基类的构造函数。 为此，请使用 `base` 关键字，如以下示例所示：

C#

```
abstract class Shape
{
    public const double pi = Math.PI;
    protected double x, y;

    public Shape(double x, double y)
    {
        this.x = x;
        this.y = y;
    }

    public abstract double Area();
}

class Circle : Shape
{
    public Circle(double radius)
        : base(radius, 0)
    {  }

    public override double Area() => pi * x * x;
}

class Cylinder : Circle
{
    public Cylinder(double radius, double height)
        : base(radius)
    {
        y = height;
    }

    public override double Area() => (2 * base.Area()) + (2 * pi * x * y);
}

class Example
{
    static void Main()
    {
        double radius = 2.5;
        double height = 3.0;

        var ring = new Circle(radius);
        Console.WriteLine($"Area of the circle = {ring.Area():F2}");
        // Output: Area of the circle = 19.63

        var tube = new Cylinder(radius, height);
        Console.WriteLine($"Area of the cylinder = {tube.Area():F2}");
        // Output: Area of the cylinder = 86.39
    }
}
```

无参数构造函数

如果某个类没有显式实例构造函数，C# 将提供可用于实例化该类实例的无参数构造函数，如以下示例所示：

C#

```
public class Person
{
    public int age;
    public string name = "unknown";
}

class Example
{
    static void Main()
    {
        var person = new Person();
        Console.WriteLine($"Name: {person.name}, Age: {person.age}");
        // Output:  Name: unknown, Age: 0
    }
}
```

该构造函数根据相应的初始值设定项初始化实例字段和属性。 如果字段或属性没有初始值设定项，其值将设置为字段或属性类型的默认值。 如果在某个类中声明至少一个实例构造函数，则 C# 不提供无参数构造函数。

structure 类型始终提供无参数构造函数： 无参数构造函数是可生成某种类型的默认值的隐式无参数构造函数或显式声明的无参数构造函数。 有关详细信息，请参阅结构类型一文的结构初始化和默认值部分。

主构造函数

从 C# 12 开始，可以在类和结构中声明主构造函数。 将任何参数放在类型名称后面的括号中：

C#

```
public class NamedItem(string name)
{
    public string Name => name;
}
```

主构造函数的参数位于声明类型的整个主体中。 它们可以初始化属性或字段。 它们可用作方法或局部函数中的变量。 它们可以传递给基本构造函数。

主构造函数指示这些参数对于类型的任何实例是必需的。 任何显式编写的构造函数都必须使用 `this(...)` 初始化表达式语法来调用主构造函数。 这可确保主构造函数参数绝对由所有构造函数分配。 对于任何 `class` 类型（包括 `record class` 类型），当主构造函数存在时，不会发出隐式无参数构造函数。 对于任何 `struct` 类型（包括 `record struct` 类型），始终发出隐式无参数构造函数，并始终将所有字段（包括主构造函数参数）初始化为 0 位模式。 如果编写显式无参数构造函数，则必须调用主构造函数。 在这种情况下，可以为主构造函数参数指定不同的值。 以下代码演示主构造函数的示例。

C#

```
// name isn't captured in Widget.
// width, height, and depth are captured as private fields
public class Widget(string name, int width, int height, int depth) :
NamedItem(name)
{
    public Widget() : this("N/A", 1,1,1) {} // unnamed unit cube

    public int WidthInCM => width;
    public int HeightInCM => height;
    public int DepthInCM => depth;

    public int Volume => width * height * depth;
}
```

可以通过在属性上指定 `method:` 目标，可以将属性添加到合成的主要构造函数方法：

C#

```
[method: MyAttribute]
public class TaggedWidget(string name)
{
   // details elided
}
```

如果未指定 `method` 目标，则属性将放置在类上而不是方法上。

在 `class` 和 `struct` 类型中，主构造函数参数在类型主体中的任意位置可用。 可将参数实现为捕获的专用字段。 如果对参数的唯一引用是初始化表达式和构造函数调用，则不会在专用字段中捕获该参数。 在该类型的其他成员中使用会导致编译器在专用字段中捕获该参数。

如果该类型包含 `record` 修饰符，则编译器将合成一个与主构造函数参数同名的公共属性。 对于 `record class` 类型，如果主构造函数参数使用与基主构造函数相同的名称，则该属性是基 `record class` 类型的公共属性。 它在派生的 `record class` 类型中不会重复。 不会为非 `record` 类型生成这些属性。

另请参阅

- 类、结构和记录
- 构造函数
- 终结器
- base
- this
- 主构造函数功能规范

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 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

私有构造函数（C# 编程指南）

项目 • 2024/03/13

私有构造函数是一种特殊的实例构造函数。 它通常用于只包含静态成员的类中。 如果类具有一个或多个私有构造函数而没有公共构造函数，则其他类（除嵌套类外）无法创建该类的实例。 例如：

C#

```
class NLog
{
    // Private Constructor:
    private NLog() { }

    public static double e = Math.E;  //2.71828...
}
```

声明空构造函数可阻止自动生成无参数构造函数。 请注意，如果不对构造函数使用访问修饰符，则在默认情况下它仍为私有构造函数。 但是，通常会显式地使用 private 修饰符来清楚地表明该类不能被实例化。

当没有实例字段或实例方法（例如 Math 类）时或者当调用方法以获得类的实例时，私有构造函数可用于阻止创建类的实例。 如果类中的所有方法都是静态的，可考虑使整个类成为静态的。 有关详细信息，请参阅静态类和静态类成员。

示例

下面是使用私有构造函数的类的示例。

C#

```
public class Counter
{
    private Counter() { }

    public static int currentCount;

    public static int IncrementCount()
    {
        return ++currentCount;
    }
}

class TestCounter
{
    static void Main()
```

```
    {
        // If you uncomment the following statement, it will generate
        // an error because the constructor is inaccessible:
        // Counter aCounter = new Counter();   // Error

        Counter.currentCount = 100;
        Counter.IncrementCount();
        Console.WriteLine("New count: {0}", Counter.currentCount);

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
// Output: New count: 101
```

请注意，如果取消注释该示例中的以下语句，它将生成一个错误，因为该构造函数受其保护级别的限制而不可访问：

C#

```
// Counter aCounter = new Counter();   // Error
```

另请参阅

- C# 类型系统
- 构造函数
- 终结器
- private
- public

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

静态构造函数（C# 编程指南）

项目 • 2024/08/05

静态构造函数用于初始化任何静态数据，或执行仅需执行一次的特定操作。 它会在创建第一个实例或引用任何静态成员之前自动调用。 静态构造函数最多调用一次。

C#

```
class SimpleClass
{
    // Static variable that must be initialized at run time.
    static readonly long baseline;

    // Static constructor is called at most one time, before any
    // instance constructor is invoked or member is accessed.
    static SimpleClass()
    {
        baseline = DateTime.Now.Ticks;
    }
}
```

有多个操作在静态初始化时执行。 这些操作按以下顺序执行：

1. 静态字段设置为 0。 通常由运行时执行此初始化。
2. 静态字段初始值设定项运行。 派生程度最高类型的静态字段初始值设定项运行。
3. 基类型静态字段初始值设定项运行。 以直接基开头从每个基类型到 System.Object 的静态字段初始值设定项。
4. 所有静态构造函数运行。 任何静态构造函数都会运行，从 Object.Object 的最终基类到每一个基类，再到类型。 静态构造函数执行的顺序没有指定。 但是，在创建任何实例之前，层次结构中的所有静态构造函数都会运行。

ⓘ 重要

静态构造函数在创建任何实例之前运行的规则有一个重要例外。 如果静态字段初始值设定项创建了该类型的实例，那么该初始值设定项（包括对实例构造函数的任何调用）将在静态构造函数运行之前运行。 这在单一实例模式中最为常见，如以下示例所示：

C#

```
public class Singleton
{
    // Static field initializer calls instance constructor.
    private static Singleton instance = new Singleton();
```

```
    private Singleton()
    {
        Console.WriteLine("Executes before static constructor.");
    }

    static Singleton()
    {
        Console.WriteLine("Executes after instance constructor.");
    }

    public static Singleton Instance => instance;
}
```

模块初始化表达式可以替代静态构造函数。 有关详细信息，请参阅模块初始化表达式的规范。

备注

静态构造函数具有以下属性：

- 静态构造函数不使用访问修饰符或不具有参数。
- 类或结构只能有一个静态构造函数。
- 静态构造函数不能继承或重载。
- 静态构造函数不能直接调用，只能由公共语言运行时 (CLR) 调用。 它会自动调用。
- 用户无法控制在程序中执行静态构造函数的时间。
- 自动调用静态构造函数。 它在创建第一个实例或引用该类（不是其基类）中声明的任何静态成员之前初始化类。 静态构造函数在实例构造函数之前运行。 如果静态构造函数类中存在静态字段变量初始值设定项，它们将以在类声明中显示的文本顺序运行。 初始值设定项紧接着静态构造函数之前运行。
- 如果未提供静态构造函数来初始化静态字段，会将所有静态字段初始化为其默认值，如 C# 类型的默认值中所列。
- 如果静态构造函数引发异常，运行时不会再次调用该函数，并且类型在应用程序域的生存期内会保持未初始化状态。 大多数情况下，当静态构造函数无法实例化一个类型时，或者当静态构造函数中发生未经处理的异常时，将引发 TypeInitializationException 异常。 对于未在源代码中显式定义的静态构造函数，故障排除可能需要检查中间语言 (IL) 代码。
- 静态构造函数的存在将防止添加 BeforeFieldInit 类型属性。 这将限制运行时优化。
- 声明为 `static readonly` 的字段只能在其声明时或在静态构造函数中分配。 如果不需要显式静态构造函数，请在声明时初始化静态字段，而不是通过静态构造函数，以实现更好的运行时优化。
- 运行时在单个应用程序域中多次调用静态构造函数。 该调用是基于特定类型的类在锁定区域中进行的。 静态构造函数的主体中不需要额外的锁定机制。 若要避免死锁的风险，请勿阻止静态构造函数和初始值设定项中的当前线程。 例如，不要等待任

务、线程、等待句柄或事件，不要获取锁定，也不要执行阻止并行操作，如并行循环、`Parallel.Invoke` 和并行 LINQ 查询。

> **备注**
>
> 尽管不可直接访问，但应记录显式静态构造函数的存在，以帮助故障排除初始化异常。

用法

- 静态构造函数的一种典型用法是在类使用日志文件且将构造函数用于将条目写入到此文件中时使用。
- 静态构造函数对于创建非托管代码的包装类也非常有用，这种情况下构造函数可调用 `LoadLibrary` 方法。
- 也可在静态构造函数中轻松地对无法在编译时通过类型参数约束检查的类型参数强制执行运行时检查。

示例

在此示例中，类 `Bus` 具有静态构造函数。 创建 `Bus` 的第一个实例（`bus1`）时，将调用该静态构造函数，以便初始化类。 示例输出验证即使创建了两个 `Bus` 的实例，静态构造函数也仅运行一次，并且在实例构造函数运行前运行。

C#

```
public class Bus
{
    // Static variable used by all Bus instances.
    // Represents the time the first bus of the day starts its route.
    protected static readonly DateTime globalStartTime;

    // Property for the number of each bus.
    protected int RouteNumber { get; set; }

    // Static constructor to initialize the static variable.
    // It is invoked before the first instance constructor is run.
    static Bus()
    {
        globalStartTime = DateTime.Now;

        // The following statement produces the first line of output,
        // and the line occurs only once.
        Console.WriteLine("Static constructor sets global start time to
{0}",
            globalStartTime.ToLongTimeString());
```

```
    }

    // Instance constructor.
    public Bus(int routeNum)
    {
        RouteNumber = routeNum;
        Console.WriteLine("Bus #{0} is created.", RouteNumber);
    }

    // Instance method.
    public void Drive()
    {
        TimeSpan elapsedTime = DateTime.Now - globalStartTime;

        // For demonstration purposes we treat milliseconds as minutes to
simulate
        // actual bus times. Do not do this in your actual bus schedule
program!
        Console.WriteLine("{0} is starting its route {1:N2} minutes after
global start time {2}.",
                          this.RouteNumber,
                          elapsedTime.Milliseconds,
                          globalStartTime.ToShortTimeString());
    }
}

class TestBus
{
    static void Main()
    {
        // The creation of this instance activates the static constructor.
        Bus bus1 = new Bus(71);

        // Create a second bus.
        Bus bus2 = new Bus(72);

        // Send bus1 on its way.
        bus1.Drive();

        // Wait for bus2 to warm up.
        System.Threading.Thread.Sleep(25);

        // Send bus2 on its way.
        bus2.Drive();

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
/* Sample output:
    Static constructor sets global start time to 3:57:08 PM.
    Bus #71 is created.
    Bus #72 is created.
    71 is starting its route 6.00 minutes after global start time 3:57 PM.
```

```
    72 is starting its route 31.00 minutes after global start time 3:57 PM.
*/
```

C# 语言规范

有关详细信息，请参阅 C# 语言规范中的静态构造函数部分。

另请参阅

- C# 类型系统
- 构造函数
- 静态类和静态类成员
- 终结器
- 构造函数设计准则
- 安全警告 - CA2121：静态构造函数应为私有
- 模块初始值设定项

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

.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

如何编写复制构造函数（C# 编程指南）

项目 • 2024/03/13

C # 记录为对象提供复制构造函数，但对于类，你必须自行编写。

> **ⓘ 重要**
>
> 编写适用于类层次结构中所有派生类型的复制构造函数可能很困难。 如果类不是 `sealed`，则强烈建议考虑创建 `record class` 类型的层次结构，以使用编译器合成的复制构造函数。

示例

在下面的示例中，`Person`类定义一个复制构造函数，该函数使用 `Person` 的实例作为其参数。 该参数的属性值分配给 `Person` 的新实例的属性。 该代码包含一个备用复制构造函数，该函数发送要复制到该类的实例构造函数的实例的 `Name` 和 `Age` 属性。 `Person` 类为 `sealed`，因此无法通过仅复制基类来声明可能会引发错误的派生类型。

C#

```
public sealed class Person
{
    // Copy constructor.
    public Person(Person previousPerson)
    {
        Name = previousPerson.Name;
        Age = previousPerson.Age;
    }

    //// Alternate copy constructor calls the instance constructor.
    //public Person(Person previousPerson)
    //    : this(previousPerson.Name, previousPerson.Age)
    //{
    //}

    // Instance constructor.
    public Person(string name, int age)
    {
        Name = name;
        Age = age;
    }

    public int Age { get; set; }

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

```
    public string Details()
    {
        return Name + " is " + Age.ToString();
    }
}

class TestPerson
{
    static void Main()
    {
        // Create a Person object by using the instance constructor.
        Person person1 = new Person("George", 40);

        // Create another Person object, copying person1.
        Person person2 = new Person(person1);

        // Change each person's age.
        person1.Age = 39;
        person2.Age = 41;

        // Change person2's name.
        person2.Name = "Charles";

        // Show details to verify that the name and age fields are distinct.
        Console.WriteLine(person1.Details());
        Console.WriteLine(person2.Details());

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
// Output:
// George is 39
// Charles is 41
```

请参阅

- ICloneable
- 记录
- C# 类型系统
- 构造函数
- 终结器

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

终结器（C# 编程指南）

项目 • 2023/03/14

终结器（以前称为析构器）用于在垃圾回收器收集类实例时执行任何必要的最终清理操作。 在大多数情况下，通过使用 System.Runtime.InteropServices.SafeHandle 或派生类包装任何非托管句柄，可以免去编写终结器的过程。

备注

- 无法在结构中定义终结器。 它们仅用于类。
- 一个类只能有一个终结器。
- 不能继承或重载终结器。
- 不能手动调用终结器。 可以自动调用它们。
- 终结器不使用修饰符或参数。

例如，以下是类 `Car` 的终结器声明。

C#

```
class Car
{
    ~Car()  // finalizer
    {
        // cleanup statements...
    }
}
```

终结器也可以作为表达式主体定义实现，如下面的示例所示。

C#

```
public class Destroyer
{
   public override string ToString() => GetType().Name;

   ~Destroyer() => Console.WriteLine($"The {ToString()} finalizer is
executing.");
}
```

终结器隐式调用对象基类上的 Finalize。 因此，对终结器的调用会隐式转换为以下代码：

C#

```
protected override void Finalize()
{
    try
    {
        // Cleanup statements...
    }
    finally
    {
        base.Finalize();
    }
}
```

这种设计意味着，对继承链（从派生程度最高到派生程度最低）中的所有实例以递归方式调用 `Finalize` 方法。

> **备注**
>
> 不应使用空终结器。 如果类包含终结器，会在 `Finalize` 队列中创建一个条目。 此队列由垃圾回收器处理。 当 GC 处理队列时，它会调用每个终结器。 不必要的终结器（包括空的终结器、仅调用基类终结器的终结器，或者仅调用条件性发出的方法的终结器）会导致不必要的性能损失。

程序员无法控制何时调用终结器，因为这由垃圾回收器决定。 垃圾回收器检查应用程序不再使用的对象。 如果它认为某个对象符合终止条件，则调用终结器（如果有），并回收用来存储此对象的内存。 可以通过调用 Collect 强制进行垃圾回收，但多数情况下应避免此调用，因为它可能会造成性能问题。

> **备注**
>
> 对于终结器是否在应用程序终止过程中运行，这特定于每个 **.NET 的实现**。 在应用程序终止时，.NET Framework 会尽一切合理努力为尚未被执行垃圾回收的对象调用终结器，除非此类清理操作已被禁止（例如，通过调用库方法 `GC.SuppressFinalize`）。 .NET 5（包括 .NET Core）及更高版本不会在应用程序终止过程中调用终结器。 有关详细信息，请参阅 GitHub 问题 **dotnet/csharpstandard #291**。

如果需要在应用程序退出时确保清理操作能够可靠地执行，请为 System.AppDomain.ProcessExit 事件注册一个处理程序。 该处理程序将确保在应用程序退出之前，为所有需要执行清理操作的对象调用了 IDisposable.Dispose()（或 IAsyncDisposable.DisposeAsync()）。 因为你不能直接调用 Finalize，而且你也不能保证垃圾回收器在退出前调用了所有终结器，所以必须使用 `Dispose` 或 `DisposeAsync` 来确保资源得到释放。

使用终结器释放资源

一般来说，对于开发人员，C# 所需的内存管理比不面向带垃圾回收的运行时的语言要少。 这是因为 .NET 垃圾回收器会隐式管理对象的内存分配和释放。 但是，如果应用程序封装非托管的资源，例如窗口、文件和网络连接，则应使用终结器释放这些资源。 当对象符合终止条件时，垃圾回收器会运行对象的 `Finalize` 方法。

显式释放资源

如果应用程序正在使用昂贵的外部资源，我们还建议在垃圾回收器释放对象前显式释放资源。 若要释放资源，请从 IDisposable 接口实现 `Dispose` 方法，对对象执行必要的清理。 这样可大大提高应用程序的性能。 如果调用 `Dispose` 方法失败，那么即使拥有对资源的显式控制，终结器也会成为清除资源的一个保障。

有关清除资源的详细信息，请参阅以下文章：

- 清理未托管资源（清理未托管资源）
- 实现 Dispose 方法
- 实现 DisposeAsync 方法
- using 语句

示例

以下示例创建了三个类，并且这三个类构成了一个继承链。 类 `First` 是基类，`Second` 派生自 `First`，`Third` 派生自 `Second`。 这三个类都具有终结器。 在 `Main` 中，已创建派生程度最高的类的一个实例。 此代码的输出取决于应用程序所面向的 .NET 实现：

- .NET Framework：输出显示当应用程序终止时，这三个类的终结器将按照派生程度最高到最低的顺序自动进行调用。
- .NET 5（包括 .NET Core）或更高版本：没有输出，因为在应用程序终止时，此 .NET 的实现不调用终结器。

C#

```
class First
{
    ~First()
    {
        System.Diagnostics.Trace.WriteLine("First's finalizer is called.");
    }
}

class Second : First
```

```
{
    ~Second()
    {
        System.Diagnostics.Trace.WriteLine("Second's finalizer is called.");
    }
}

class Third : Second
{
    ~Third()
    {
        System.Diagnostics.Trace.WriteLine("Third's finalizer is called.");
    }
}

/*
Test with code like the following:
    Third t = new Third();
    t = null;

When objects are finalized, the output would be:
Third's finalizer is called.
Second's finalizer is called.
First's finalizer is called.
*/
```

C# 语言规范

有关详细信息，请参阅 C# 语言规范中的终结器部分。

另请参阅

- IDisposable
- C# 编程指南
- 构造函数
- 垃圾回收

对象和集合初始值设定项（C# 编程指南）

项目 • 2024/05/26

使用 C# 可以在单条语句中实例化对象或集合并执行成员分配。

对象初始值设定项

使用对象初始值设定项，你可以在创建对象时向对象的任何可访问字段或属性分配值，而无需调用后跟赋值语句行的构造函数。 利用对象初始值设定项语法，你可为构造函数指定参数或忽略参数（以及括号语法）。 以下示例演示如何使用具有命名类型 `Cat` 的对象初始值设定项以及如何调用无参数构造函数。 注意自动实现的属性在 `Cat` 类中的用法。 有关详细信息，请参阅自动实现的属性。

C#

```
public class Cat
{
    // Auto-implemented properties.
    public int Age { get; set; }
    public string? Name { get; set; }

    public Cat()
    {
    }

    public Cat(string name)
    {
        this.Name = name;
    }
}
```

C#

```
Cat cat = new Cat { Age = 10, Name = "Fluffy" };
Cat sameCat = new Cat("Fluffy"){ Age = 10 };
```

对象初始值设定项语法允许你创建一个实例，然后将具有其分配属性的新建对象指定给赋值中的变量。

除了分配字段和属性外，对象初始值设定项还可以设置索引器。 请思考这个基本的 `Matrix` 类：

C#

```
public class Matrix
{
    private double[,] storage = new double[3, 3];

    public double this[int row, int column]
    {
        // The embedded array will throw out of range exceptions as
appropriate.
        get { return storage[row, column]; }
        set { storage[row, column] = value; }
    }
}
```

可以使用以下代码初始化标识矩阵：

C#

```
var identity = new Matrix
{
    [0, 0] = 1.0,
    [0, 1] = 0.0,
    [0, 2] = 0.0,

    [1, 0] = 0.0,
    [1, 1] = 1.0,
    [1, 2] = 0.0,

    [2, 0] = 0.0,
    [2, 1] = 0.0,
    [2, 2] = 1.0,
};
```

包含可访问资源库的任何可访问索引器都可以用作对象初始值设定项中的表达式之一，这与参数的数量或类型无关。 索引参数构成左侧赋值，而表达式右侧是值。 例如，如果 `IndexersExample` 具有适当的索引器，则以下初始值设定项都是有效的：

C#

```
var thing = new IndexersExample
{
    name = "object one",
    [1] = '1',
    [2] = '4',
    [3] = '9',
    Size = Math.PI,
    ['C',4] = "Middle C"
}
```

对于要进行编译的前面的代码，`IndexersExample` 类型必须具有以下成员：

```
C#

public string name;
public double Size { set { ... }; }
public char this[int i] { set { ... }; }
public string this[char c, int i] { set { ... }; }
```

具有匿名类型的对象初始值设定项

尽管对象初始值设定项可用于任何上下文中，但它们在 LINQ 查询表达式中特别有用。 查询表达式常使用只能通过使用对象初始值设定项进行初始化的匿名类型，如下面的声明所示。

```
C#

var pet = new { Age = 10, Name = "Fluffy" };
```

利用匿名类型，LINQ 查询表达式中的 `select` 子句可以将原始序列的对象转换为其值和形状可能不同于原始序列的对象。 建议只存储某个序列中每个对象的部分信息。 在下面的示例中，假定产品对象 (`p`) 包含很多字段和方法，而你只想创建包含产品名和单价的对象序列。

```
C#

var productInfos =
    from p in products
    select new { p.ProductName, p.UnitPrice };
```

执行此查询时，`productInfos` 变量包含一系列对象，这些对象可以在 `foreach` 语句中进行访问，如下面的示例所示：

```
C#

foreach(var p in productInfos){...}
```

新的匿名类型中的每个对象都具有两个公共属性，这两个属性接收与原始对象中的属性或字段相同的名称。 你还可在创建匿名类型时重命名字段；下面的示例将 `UnitPrice` 字段重命名为 `Price`。

```
C#
```

```
select new {p.ProductName, Price = p.UnitPrice};
```

带 `required` 修饰符的对象初始值设定项

可以使用 `required` 关键字强制调用方使用对象初始值设定项设置属性或字段的值。 不需要将所需属性设置为构造函数参数。 编译器可确保所有调用方初始化这些值。

C#

```
public class Pet
{
    public required int Age;
    public string Name;
}

// `Age` field is necessary to be initialized.
// You don't need to initialize `Name` property
var pet = new Pet() { Age = 10};

// Compiler error:
// Error CS9035 Required member 'Pet.Age' must be set in the object
initializer or attribute constructor.
// var pet = new Pet();
```

通常的做法是保证对象正确初始化，尤其是在要管理多个字段或属性，并且不希望将它们全部包含在构造函数中时。

带 `init` 访问器的对象初始值设定项

确保无人更改设计，并且可以使用访问器来限制 `init` 对象。 它有助于限制属性值的设置。

C#

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

// The `LastName` property can be set only during initialization. It CAN'T
be modified afterwards.
// The `FirstName` property can be modified after initialization.
var pet = new Person() { FirstName = "Joe", LastName = "Doe"};

// You can assign the FirstName property to a different value.
```

```
pet.FirstName = "Jane";

// Compiler error:
// Error CS8852  Init - only property or indexer 'Person.LastName' can only
be assigned in an object initializer,
//               or on 'this' or 'base' in an instance constructor or an
'init' accessor.
// pet.LastName = "Kowalski";
```

必需的仅限 init 的属性支持不可变结构，同时允许该类型用户使用自然语法。

具有类类型属性的对象初始值设定项

初始化对象时，考虑类类型属性的含义至关重要：

C#

```
public class HowToClassTypedInitializer
{
    public class EmbeddedClassTypeA
    {
        public int I { get; set; }
        public bool B { get; set; }
        public string S { get; set; }
        public EmbeddedClassTypeB ClassB { get; set; }

        public override string ToString() => $"{I}|{B}|{S}|||{ClassB}";

        public EmbeddedClassTypeA()
        {
            Console.WriteLine($"Entering EmbeddedClassTypeA constructor.
Values are: {this}");
            I = 3;
            B = true;
            S = "abc";
            ClassB = new() { BB = true, BI = 43 };
            Console.WriteLine($"Exiting EmbeddedClassTypeA constructor.
Values are: {this})");
        }
    }

    public class EmbeddedClassTypeB
    {
        public int BI { get; set; }
        public bool BB { get; set; }
        public string BS { get; set; }

        public override string ToString() => $"{BI}|{BB}|{BS}";

        public EmbeddedClassTypeB()
        {
```

```
            Console.WriteLine($"Entering EmbeddedClassTypeB constructor.
Values are: {this}");
            BI = 23;
            BB = false;
            BS = "BBBabc";
            Console.WriteLine($"Exiting EmbeddedClassTypeB constructor.
Values are: {this})");
        }
    }

    public static void Main()
    {
        var a = new EmbeddedClassTypeA
        {
            I = 103,
            B = false,
            ClassB = { BI = 100003 }
        };
        Console.WriteLine($"After initializing EmbeddedClassTypeA: {a}");

        var a2 = new EmbeddedClassTypeA
        {
            I = 103,
            B = false,
            ClassB = new() { BI = 100003 } //New instance
        };
        Console.WriteLine($"After initializing EmbeddedClassTypeA a2:
{a2}");
    }

    // Output:
    //Entering EmbeddedClassTypeA constructor Values are: 0|False||||
    //Entering EmbeddedClassTypeB constructor Values are: 0|False|
    //Exiting EmbeddedClassTypeB constructor Values are: 23|False|BBBabc)
    //Exiting EmbeddedClassTypeA constructor Values are:
3|True|abc|||43|True|BBBabc)
    //After initializing EmbeddedClassTypeA:
103|False|abc|||100003|True|BBBabc
    //Entering EmbeddedClassTypeA constructor Values are: 0|False||||
    //Entering EmbeddedClassTypeB constructor Values are: 0|False|
    //Exiting EmbeddedClassTypeB constructor Values are: 23|False|BBBabc)
    //Exiting EmbeddedClassTypeA constructor Values are:
3|True|abc|||43|True|BBBabc)
    //Entering EmbeddedClassTypeB constructor Values are: 0|False|
    //Exiting EmbeddedClassTypeB constructor Values are: 23|False|BBBabc)
    //After initializing EmbeddedClassTypeA a2:
103|False|abc|||100003|False|BBBabc
}
```

以下示例演示了对于 ClassB，初始化过程如何涉及到更新特定的值，同时保留原始实例中的其他值。 初始值设定项重用当前实例：ClassB 的值为：`100003`（此处分配的新

值）、`true`（在 EmbeddedClassTypeA 的初始化中保留的值）、`BBBabc`（EmbeddedClassTypeB 中未更改的默认值）。

集合初始值设定项

在初始化实现 IEnumerable 的集合类型和初始化使用适当的签名作为实例方法或扩展方法的 `Add` 时，集合初始值设定项允许指定一个或多个元素初始值设定项。 元素初始值设定项可以是值、表达式或对象初始值设定项。 通过使用集合初始值设定项，无需指定多个调用；编译器将自动添加这些调用。

下面的示例演示了两个简单的集合初始值设定项：

C#

```
List<int> digits = new List<int> { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
List<int> digits2 = new List<int> { 0 + 1, 12 % 3, MakeInt() };
```

下面的集合初始值设定项使用对象初始值设定项来初始化上一个示例中定义的 `Cat` 类的对象。 各个对象初始值设定项括在大括号中且用逗号隔开。

C#

```
List<Cat> cats = new List<Cat>
{
    new Cat{ Name = "Sylvester", Age=8 },
    new Cat{ Name = "Whiskers", Age=2 },
    new Cat{ Name = "Sasha", Age=14 }
};
```

如果集合的 `Add` 方法允许，则可以将 null 指定为集合初始值设定项中的一个元素。

C#

```
List<Cat?> moreCats = new List<Cat?>
{
    new Cat{ Name = "Furrytail", Age=5 },
    new Cat{ Name = "Peaches", Age=4 },
    null
};
```

如果集合支持读取/写入索引，可以指定索引元素。

C#

```
var numbers = new Dictionary<int, string>
{
    [7] = "seven",
    [9] = "nine",
    [13] = "thirteen"
};
```

前面的示例生成调用 Item[TKey] 以设置值的代码。 还可使用以下语法初始化字典和其他关联容器。 请注意，它使用具有多个值的对象，而不是带括号和赋值的索引器语法：

C#

```
var moreNumbers = new Dictionary<int, string>
{
    {19, "nineteen" },
    {23, "twenty-three" },
    {42, "forty-two" }
};
```

此初始值设定项示例调用 Add(TKey, TValue)，将这三个项添加到字典中。 由于编译器生成的方法调用不同，这两种初始化关联集合的不同方法的行为略有不同。 这两种变量都适用于 `Dictionary` 类。 其他类型根据它们的公共 API 可能只支持两者中的一种。

具有集合只读属性初始化的对象初始值设定项

某些类可能具有属性为只读的集合属性，如以下示例中 `CatOwner` 的 `Cats` 属性：

C#

```
public class CatOwner
{
    public IList<Cat> Cats { get; } = new List<Cat>();
}
```

由于无法为属性分配新列表，因此你无法使用迄今为止讨论的集合初始值设定项语法：

C#

```
CatOwner owner = new CatOwner
{
    Cats = new List<Cat>
    {
        new Cat{ Name = "Sylvester", Age=8 },
        new Cat{ Name = "Whiskers", Age=2 },
        new Cat{ Name = "Sasha", Age=14 }
```

```
    }
};
```

但是，可以通过省略列表创建（`new List<Cat>`），使用初始化语法将新条目添加到 `Cats`，如下所示：

C#

```
CatOwner owner = new CatOwner
{
    Cats =
    {
        new Cat{ Name = "Sylvester", Age=8 },
        new Cat{ Name = "Whiskers", Age=2 },
        new Cat{ Name = "Sasha", Age=14 }
    }
};
```

要添加的条目集显示在大括号中。 上述代码与编写代码相同：

C#

```
CatOwner owner = new ();
owner.Cats.Add(new Cat{ Name = "Sylvester", Age=8 });
owner.Cats.Add(new Cat{ Name = "Whiskers", Age=2 });
owner.Cats.Add(new Cat{ Name = "Sasha", Age=14 });
```

示例

下例结合了对象和集合初始值设定项的概念。

C#

```
public class InitializationSample
{
    public class Cat
    {
        // Auto-implemented properties.
        public int Age { get; set; }
        public string? Name { get; set; }

        public Cat() { }

        public Cat(string name)
        {
            Name = name;
        }
    }
```

```
    public static void Main()
    {
        Cat cat = new Cat { Age = 10, Name = "Fluffy" };
        Cat sameCat = new Cat("Fluffy"){ Age = 10 };

        List<Cat> cats = new List<Cat>
        {
            new Cat { Name = "Sylvester", Age = 8 },
            new Cat { Name = "Whiskers", Age = 2 },
            new Cat { Name = "Sasha", Age = 14 }
        };

        List<Cat?> moreCats = new List<Cat?>
        {
            new Cat { Name = "Furrytail", Age = 5 },
            new Cat { Name = "Peaches", Age = 4 },
            null
        };

        // Display results.
        System.Console.WriteLine(cat.Name);

        foreach (Cat c in cats)
        {
            System.Console.WriteLine(c.Name);
        }

        foreach (Cat? c in moreCats)
        {
            if (c != null)
            {
                System.Console.WriteLine(c.Name);
            }
            else
            {
                System.Console.WriteLine("List element has null value.");
            }
        }
    }
    // Output:
    //Fluffy
    //Sylvester
    //Whiskers
    //Sasha
    //Furrytail
    //Peaches
    //List element has null value.
}
```

以下示例演示了一个对象，该对象实现 IEnumerable 并包含具有多个参数的 `Add` 方法。它使用一个集合初始值设定项，其中列表中的每个项都有多个元素，这些元素对应于 `Add` 方法的签名。

C#

```
public class FullExample
{
    class FormattedAddresses : IEnumerable<string>
    {
        private List<string> internalList = new List<string>();
        public IEnumerator<string> GetEnumerator() =>
internalList.GetEnumerator();

        System.Collections.IEnumerator
System.Collections.IEnumerable.GetEnumerator() =>
internalList.GetEnumerator();

        public void Add(string firstname, string lastname,
            string street, string city,
            string state, string zipcode) => internalList.Add($"""
            {firstname} {lastname}
            {street}
            {city}, {state} {zipcode}
            """
            );
    }

    public static void Main()
    {
        FormattedAddresses addresses = new FormattedAddresses()
        {
            {"John", "Doe", "123 Street", "Topeka", "KS", "00000" },
            {"Jane", "Smith", "456 Street", "Topeka", "KS", "00000" }
        };

        Console.WriteLine("Address Entries:");

        foreach (string addressEntry in addresses)
        {
            Console.WriteLine("\r\n" + addressEntry);
        }
    }

    /*
        * Prints:

        Address Entries:

        John Doe
        123 Street
        Topeka, KS 00000

        Jane Smith
        456 Street
        Topeka, KS 00000
```

```
        */
}
```

`Add` 方法可使用 `params` 关键字来获取可变数量的自变量，如下例中所示。 此示例还演示了索引器的自定义实现，以使用索引初始化集合。 从 C# 13 开始，`params` 参数不再局限于数组。 它可以是集合类型或接口。

C#

```
public class DictionaryExample
{
    class RudimentaryMultiValuedDictionary<TKey, TValue> :
IEnumerable<KeyValuePair<TKey, List<TValue>>> where TKey : notnull
    {
        private Dictionary<TKey, List<TValue>> internalDictionary = new
Dictionary<TKey, List<TValue>>();

        public IEnumerator<KeyValuePair<TKey, List<TValue>>> GetEnumerator()
=> internalDictionary.GetEnumerator();

        System.Collections.IEnumerator
System.Collections.IEnumerable.GetEnumerator() =>
internalDictionary.GetEnumerator();

        public List<TValue> this[TKey key]
        {
            get => internalDictionary[key];
            set => Add(key, value);
        }

        public void Add(TKey key, params TValue[] values) => Add(key,
(IEnumerable<TValue>)values);

        public void Add(TKey key, IEnumerable<TValue> values)
        {
            if (!internalDictionary.TryGetValue(key, out List<TValue>?
storedValues))
            {
                internalDictionary.Add(key, storedValues = new List<TValue>
());
            }
            storedValues.AddRange(values);
        }
    }

    public static void Main()
    {
        RudimentaryMultiValuedDictionary<string, string>
rudimentaryMultiValuedDictionary1
            = new RudimentaryMultiValuedDictionary<string, string>()
            {
                {"Group1", "Bob", "John", "Mary" },
```

```
                {"Group2", "Eric", "Emily", "Debbie", "Jesse" }
            };
        RudimentaryMultiValuedDictionary<string, string>
rudimentaryMultiValuedDictionary2
            = new RudimentaryMultiValuedDictionary<string, string>()
            {
                ["Group1"] = new List<string>() { "Bob", "John", "Mary" },
                ["Group2"] = new List<string>() { "Eric", "Emily", "Debbie",
"Jesse" }
            };
        RudimentaryMultiValuedDictionary<string, string>
rudimentaryMultiValuedDictionary3
            = new RudimentaryMultiValuedDictionary<string, string>()
            {
                {"Group1", new string []{ "Bob", "John", "Mary" } },
                { "Group2", new string[]{ "Eric", "Emily", "Debbie", "Jesse"
} }
            };

        Console.WriteLine("Using first multi-valued dictionary created with
a collection initializer:");

        foreach (KeyValuePair<string, List<string>> group in
rudimentaryMultiValuedDictionary1)
        {
            Console.WriteLine($"\r\nMembers of group {group.Key}: ");

            foreach (string member in group.Value)
            {
                Console.WriteLine(member);
            }
        }

        Console.WriteLine("\r\nUsing second multi-valued dictionary created
with a collection initializer using indexing:");

        foreach (KeyValuePair<string, List<string>> group in
rudimentaryMultiValuedDictionary2)
        {
            Console.WriteLine($"\r\nMembers of group {group.Key}: ");

            foreach (string member in group.Value)
            {
                Console.WriteLine(member);
            }
        }
        Console.WriteLine("\r\nUsing third multi-valued dictionary created
with a collection initializer using indexing:");

        foreach (KeyValuePair<string, List<string>> group in
rudimentaryMultiValuedDictionary3)
        {
            Console.WriteLine($"\r\nMembers of group {group.Key}: ");

            foreach (string member in group.Value)
```

```
            {
                Console.WriteLine(member);
            }
        }
    }

    /*
        * Prints:

        Using first multi-valued dictionary created with a collection
initializer:

        Members of group Group1:
        Bob
        John
        Mary

        Members of group Group2:
        Eric
        Emily
        Debbie
        Jesse

        Using second multi-valued dictionary created with a collection
initializer using indexing:

        Members of group Group1:
        Bob
        John
        Mary

        Members of group Group2:
        Eric
        Emily
        Debbie
        Jesse

        Using third multi-valued dictionary created with a collection
initializer using indexing:

        Members of group Group1:
        Bob
        John
        Mary

        Members of group Group2:
        Eric
        Emily
        Debbie
        Jesse
        */
}
```

另请参阅

- 使用对象初始值设定项（样式规则 IDE0017）
- 使用集合初始值设定项（样式规则 IDE0028）
- C# 中的 LINQ
- 匿名类型

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

如何使用对象初始值设定项初始化对象（C# 编程指南）

项目 • 2024/05/15

可以使用对象初始值设定项以声明方式初始化类型对象，而无需显式调用类型的构造函数。

以下示例演示如何将对象初始值设定项用于命名对象。 编译器通过首先访问无参数实例构造函数，然后处理成员初始化来处理对象初始值设定项。 因此，如果无参数构造函数在类中声明为 `private`，则需要公共访问的对象初始值设定项将失败。

如果要定义匿名类型，则必须使用对象初始值设定项。 有关详细信息，请参阅如何在查询中返回元素属性的子集。

示例

下面的示例演示如何使用对象初始值设定项初始化新的 `StudentName` 类型。 此示例在 `StudentName` 类型中设置属性：

C#

```
public class HowToObjectInitializers
{
    public static void Main()
    {
        // Declare a StudentName by using the constructor that has two
parameters.
        StudentName student1 = new StudentName("Craig", "Playstead");

        // Make the same declaration by using an object initializer and
sending
        // arguments for the first and last names. The parameterless
constructor is
        // invoked in processing this declaration, not the constructor that
has
        // two parameters.
        StudentName student2 = new StudentName
        {
            FirstName = "Craig",
            LastName = "Playstead"
        };

        // Declare a StudentName by using an object initializer and sending
        // an argument for only the ID property. No corresponding
constructor is
        // necessary. Only the parameterless constructor is used to process
```

```
object
        // initializers.
        StudentName student3 = new StudentName
        {
            ID = 183
        };

        // Declare a StudentName by using an object initializer and sending
        // arguments for all three properties. No corresponding constructor
is
        // defined in the class.
        StudentName student4 = new StudentName
        {
            FirstName = "Craig",
            LastName = "Playstead",
            ID = 116
        };

        Console.WriteLine(student1.ToString());
        Console.WriteLine(student2.ToString());
        Console.WriteLine(student3.ToString());
        Console.WriteLine(student4.ToString());
    }
    // Output:
    // Craig  0
    // Craig  0
    //   183
    // Craig  116

    public class StudentName
    {
        // This constructor has no parameters. The parameterless constructor
        // is invoked in the processing of object initializers.
        // You can test this by changing the access modifier from public to
        // private. The declarations in Main that use object initializers
will
        // fail.
        public StudentName() { }

        // The following constructor has parameters for two of the three
        // properties.
        public StudentName(string first, string last)
        {
            FirstName = first;
            LastName = last;
        }

        // Properties.
        public string? FirstName { get; set; }
        public string? LastName { get; set; }
        public int ID { get; set; }

        public override string ToString() => FirstName + "  " + ID;
    }
}
```

对象初始值设定项可用于在对象中设置索引器。 下面的示例定义了一个 `BaseballTeam` 类，该类使用索引器获取和设置不同位置的球员。 初始值设定项可以根据位置的缩写或每个位置的棒球记分卡的编号来分配球员：

C#

```
public class HowToIndexInitializer
{
    public class BaseballTeam
    {
        private string[] players = new string[9];
        private readonly List<string> positionAbbreviations = new
List<string>
        {
            "P", "C", "1B", "2B", "3B", "SS", "LF", "CF", "RF"
        };

        public string this[int position]
        {
            // Baseball positions are 1 - 9.
            get { return players[position-1]; }
            set { players[position-1] = value; }
        }
        public string this[string position]
        {
            get { return players[positionAbbreviations.IndexOf(position)]; }
            set { players[positionAbbreviations.IndexOf(position)] = value;
}
        }
    }

    public static void Main()
    {
        var team = new BaseballTeam
        {
            ["RF"] = "Mookie Betts",
            [4] = "Jose Altuve",
            ["CF"] = "Mike Trout"
        };

        Console.WriteLine(team["2B"]);
    }
}
```

下一个示例演示使用带参数和不带参数的构造函数执行构造函数和成员初始化的顺序：

C#

另请参阅

- 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 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

如何使用集合初始值设定项初始化字典（C# 编程指南）

项目 • 2024/03/25

Dictionary<TKey,TValue> 包含键/值对集合。 其 Add 方法采用两个参数，一个用于键，一个用于值。 若要初始化 Dictionary<TKey,TValue> 或其 `Add` 方法采用多个参数的任何集合，一种方法是将每组参数括在大括号中，如下面的示例中所示。 另一种方法是使用索引初始值设定项，如下面的示例所示。

> **备注**
>
> 我们以具有重复键的情况来举例说明初始化集合的这两种方法之间的主要区别：
>
> ```
> C#
>
> { 111, new StudentName { FirstName="Sachin", LastName="Karnik", ID=211
> } },
> { 111, new StudentName { FirstName="Dina", LastName="Salimzianova",
> ID=317 } },
> ```
>
> Add 方法将引发 ArgumentException：`'An item with the same key has already been added. Key: 111'`，而示例的第二部分（公共读/写索引器方法）将使用相同的键静默覆盖已存在的条目。

示例

在下面的代码示例中，使用类型 `StudentName` 的实例初始化 Dictionary<TKey,TValue>。 第一个初始化使用具有两个参数的 `Add` 方法。 编译器为每对 `int` 键和 `StudentName` 值生成对 `Add` 的调用。 第二个初始化使用 `Dictionary` 类的公共读取/写入索引器方法：

```
C#

public class HowToDictionaryInitializer
{
    class StudentName
    {
        public string? FirstName { get; set; }
        public string? LastName { get; set; }
        public int ID { get; set; }
    }

    public static void Main()
```

```
    {
        var students = new Dictionary<int, StudentName>()
        {
            { 111, new StudentName { FirstName="Sachin", LastName="Karnik",
ID=211 } },
            { 112, new StudentName { FirstName="Dina",
LastName="Salimzianova", ID=317 } },
            { 113, new StudentName { FirstName="Andy", LastName="Ruth",
ID=198 } }
        };

        foreach(var index in Enumerable.Range(111, 3))
        {
            Console.WriteLine($"Student {index} is
{students[index].FirstName} {students[index].LastName}");
        }
        Console.WriteLine();

        var students2 = new Dictionary<int, StudentName>()
        {
            [111] = new StudentName { FirstName="Sachin", LastName="Karnik",
ID=211 },
            [112] = new StudentName { FirstName="Dina",
LastName="Salimzianova", ID=317 } ,
            [113] = new StudentName { FirstName="Andy", LastName="Ruth",
ID=198 }
        };

        foreach (var index in Enumerable.Range(111, 3))
        {
            Console.WriteLine($"Student {index} is
{students2[index].FirstName} {students2[index].LastName}");
        }
    }
}
```

请注意，在第一个声明中，集合中的每个元素有两对大括号。 最内层的大括号中括住了 `StudentName` 的对象初始值设定项，最外层的大括号则括住了要添加到 `students` Dictionary<TKey,TValue> 的键/值对的初始值设定项。 最后，字典的整个集合初始值设定项被括在大括号中。 在第二个初始化中，左侧赋值是键，右侧是将对象初始值设定项用于 `StudentName` 的值。

另请参阅

- 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 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

嵌套类型（C# 编程指南）

项目 • 2024/03/13

在类、构造或接口中定义的类型称为嵌套类型。 例如

C#

```
public class Container
{
    class Nested
    {
        Nested() { }
    }
}
```

不论外部类型是类、接口还是构造，嵌套类型均默认为 private；仅可从其包含类型中进行访问。 在上一个示例中，`Nested` 类无法访问外部类型。

还可指定访问修饰符来定义嵌套类型的可访问性，如下所示：

- “类”的嵌套类型可以是 public、protected、internal、protected internal、private 或 private protected。

  但是，在密封类中定义 `protected`、`protected internal` 或 `private protected` 嵌套类将产生编译器警告 CS0628“封闭类汇中声明了新的受保护成员”。

  另请注意，使嵌套类型在外部可见违反了代码质量规则 CA1034“嵌套类型不应是可见的”。

- 构造的嵌套类型可以是 public、internal 或 private。

以下示例使 `Nested` 类为 public：

C#

```
public class Container
{
    public class Nested
    {
        Nested() { }
    }
}
```

嵌套类型（或内部类型）可访问包含类型（或外部类型）。 若要访问包含类型，请将其作为参数传递给嵌套类型的构造函数。 例如：

C#

```
public class Container
{
    public class Nested
    {
        private Container? parent;

        public Nested()
        {
        }
        public Nested(Container parent)
        {
            this.parent = parent;
        }
    }
}
```

嵌套类型可以访问其包含类型可以访问的所有成员。 它可以访问包含类型的私有成员和受保护成员（包括所有继承的受保护成员）。

在前面的声明中，类 `Nested` 的完整名称为 `Container.Nested`。 这是用来创建嵌套类新实例的名称，如下所示：

C#

```
Container.Nested nest = new Container.Nested();
```

另请参阅

- C# 类型系统
- 访问修饰符
- 构造函数
- CA1034 规则

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

分部类和方法（C# 编程指南）

项目 • 2024/11/14

拆分一个类、一个结构、一个接口或一个方法的定义到两个或更多的文件中是可能的。每个源文件包含类型或方法定义的一部分，编译应用程序时将把所有部分组合起来。

分部类

在以下几种情况下需要拆分类定义：

- 通过单独的文件声明某个类可以让多位程序员同时对该类进行处理。
- 你可以向该类中添加代码，而不必重新创建包括自动生成的源代码的源文件。Visual Studio 在创建Windows 窗体、Web 服务包装器代码等时会使用这种方法。你可以创建使用这些类的代码，这样就不需要修改由Visual Studio生成的文件。
- 源代码生成器可以在类中生成额外的功能。

若要拆分类定义，请使用 分部 关键字修饰符。 实际上，每个分部类通常在单独的文件中定义，以便随着时间的推移更轻松地管理和扩展类。

以下示例 `Employee` 演示如何将类划分为两个文件：Employee_Part1.cs和Employee_Part2.cs。

C#

```
// This is in Employee_Part1.cs
public partial class Employee
{
    public void DoWork()
    {
    }
}

// This is in Employee_Part2.cs
public partial class Employee
{
    public void GoToLunch()
    {
    }
}

//Main program demonstrating the Employee class usage
public class Program
{
    public static void Main()
    {
        Employee emp = new Employee();
        emp.DoWork();
```

```
        emp.GoToLunch();
    }
}

// Expected Output:
// Employee is working.
// Employee is at lunch.
```

`partial` 关键字指示可在命名空间中定义该类、结构或接口的其他部分。 所有部分都必须使用 `partial` 关键字。 在编译时，各个部分都必须可用来形成最终的类型。 各个部分必须具有相同的可访问性，如 `public`、`private` 等。

如果将任意部分声明为抽象的，则整个类型都被视为抽象的。 如果将任意部分声明为密封的，则整个类型都被视为密封的。 如果任意部分声明基类型，则整个类型都将继承该类。

指定基类的所有部分必须一致，但忽略基类的部分仍继承该基类型。 各个部分可以指定不同的基接口，最终类型将实现所有分部声明所列出的全部接口。 在某一分部定义中声明的任何类、结构或接口成员可供所有其他部分使用。 最终类型是所有部分在编译时的组合。

> **备注**
>
> `partial` 修饰符不可用于委托或枚举声明中。

下面的示例演示嵌套类型可以是分部的，即使它们所嵌套于的类型本身并不是分部的也如此。

C#

```
class Container
{
    partial class Nested
    {
        void Test() { }
    }

    partial class Nested
    {
        void Test2() { }
    }
}
```

编译时会对分部类型定义的属性进行合并。 以下面的声明为例：

C#

```
[SerializableAttribute]
partial class Moon { }

[ObsoleteAttribute]
partial class Moon { }
```

它们等效于以下声明：

C#

```
[SerializableAttribute]
[ObsoleteAttribute]
class Moon { }
```

将从所有分部类型定义中对以下内容进行合并：

- XML 注释。 但是，如果分部成员的两个声明都包含注释，则仅包括实现成员的注释。
- interfaces
- 泛型类型参数属性
- class 特性
- 成员

以下面的声明为例：

C#

```
partial class Earth : Planet, IRotate { }
partial class Earth : IRevolve { }
```

它们等效于以下声明：

C#

```
class Earth : Planet, IRotate, IRevolve { }
```

限制

处理分部类定义时需遵循下面的几个规则：

- 要作为同一类型的各个部分的所有分部类型定义都必须使用 `partial` 进行修饰。 例如，下面的类声明会生成错误：

  C#

```
public partial class A { }
//public class A { }  // Error, must also be marked partial
```

- `partial` 修饰符只能出现在紧靠关键字 `class`、`struct` 或 `interface` 前面的位置。
- 分部类型定义中允许使用嵌套的分部类型，如下面的示例中所示：

```
C#

partial class ClassWithNestedClass
{
    partial class NestedClass { }
}

partial class ClassWithNestedClass
{
    partial class NestedClass { }
}
```

- 要成为同一类型的各个部分的所有分部类型定义都必须在同一程序集和同一模块（.exe 或 .dll 文件）中进行定义。 分部定义不能跨越多个模块。
- 类名和泛型类型参数在所有的分部类型定义中都必须匹配。 泛型类型可以是分部的。 每个分部声明都必须以相同的顺序使用相同的参数名。
- 下面用于分部类型定义中的关键字是可选的，但是如果某关键字出现在一个分部类型定义中，则必须在相同类型的其他分部定义中指定相同的关键字：
  - 公共
  - private
  - 受保护
  - internal
  - abstract
  - sealed
  - 基类
  - new 修饰符（嵌套部分）
  - 泛型约束

有关详细信息，请参阅类型参数的约束。

示例

在以下示例中，类的`Coords`字段和构造函数在一个分部类定义（`Coords_Part1.cs`）中声明，方法`PrintCoords`在另一个分部类定义中声明。 `Coords_Part2.cs` 这种分离演示了如何跨多个文件划分分部类，以便更轻松地保持可维护性。

```
C#
```

```
// This is in Coords_Part1.cs
public partial class Coords
{
    private int x;
    private int y;

    public Coords(int x, int y)
    {
        this.x = x;
        this.y = y;
    }
}

// This is in Coords_Part2.cs
public partial class Coords
{
    public void PrintCoords()
    {
        Console.WriteLine("Coords: {0},{1}", x, y);
    }
}

// Main program demonstrating the Coords class usage
class TestCoords
{
    static void Main()
    {
        Coords myCoords = new Coords(10, 15);
        myCoords.PrintCoords();

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
// Output: Coords: 10,15
```

从下面的示例可以看出，你也可以开发分部结构和接口。

C#

```
partial interface ITest
{
    void Interface_Test();
}

partial interface ITest
{
    void Interface_Test2();
}

partial struct S1
```

```
{
    void Struct_Test() { }
}

partial struct S1
{
    void Struct_Test2() { }
}
```

分部成员

分部类或结构可以包含分部成员。 类的一个部分包含成员的签名。 可以在同一部分或另一部分中定义实现。

当签名遵循以下规则时，分部方法不需要实现：

- 声明未包含任何访问修饰符。 默认情况下，该方法具有 private 访问权限。
- 返回类型为 void。
- 没有任何参数具有 out 修饰符。
- 方法声明不能包括以下任何修饰符：
  - virtual
  - override
  - sealed
  - new
  - extern

当未提供实现时，在编译时会移除该方法以及对该方法的所有调用。

任何不符合所有这些限制的方法（包括属性和索引器）都必须提供实现。 此实现可以由源生成器提供。 不能使用自动实现的属性实现部分属性 。 编译器无法区分自动实现的属性和分部属性的声明声明。

从 C# 13 开始，部分属性的实现声明可以使用 字段支持的属性 来定义实现声明。 字段支持的属性提供简洁的语法， `field` 其中关键字访问该属性的编译器合成后备字段。 例如，可以编写以下内容：

C#

```
// in file1.cs
public partial class PropertyBag
{
    // Defining declaration
    public partial int MyProperty { get; set; }
}

// In file2.cs
```

```
public partial class PropertyBag
{
    // Defining declaration
    public partial int MyProperty { get => field; set; }
}
```

可以在或`get` `set`访问器中使用，也可以同时使用`field`两者。

> **ⓘ 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。

分部方法允许类的某个部分的实现者声明成员。 类的另一部分的实现者可以定义该成员。 在以下两个情形中，此分离很有用：生成样板代码的模板和源生成器。

- **模板代码**：模板保留方法名称和签名，以使生成的代码可以调用方法。 这些方法遵循允许开发人员决定是否实现方法的限制。 如果未实现该方法，编译器会移除方法签名以及对该方法的所有调用。 调用该方法（包括调用中的任何参数计算结果）在运行时没有任何影响。 因此，分部类中的任何代码都可以随意地使用分部方法，即使未提供实现也是如此。 未实现该方法时调用该方法不会导致编译时错误或运行时错误。
- **源生成器**：源生成器提供成员的实现。 开发人员可以添加成员声明（通常由源生成器读取属性）。 开发人员可以编写调用这些成员的代码。 源生成器在编译过程中运行并提供实现。 在这种情况下，不会遵循不经常实现的分部成员的限制。

C#

```
// Definition in file1.cs
partial void OnNameChanged();

// Implementation in file2.cs
partial void OnNameChanged()
{
  // method body
}
```

- 分部成员声明必须以上下文关键字 partial 开头。
- 分部类型的两个部分中的分部成员签名必须匹配。

- 分部成员可以有 static 和 unsafe 修饰符。
- 分部成员可能是泛型成员。 约束在定义和实现方法声明时必须相同。 参数和类型参数名称在定义和实现方法声明时不必相同。
- 你可以为已定义并实现的分部方法生成委托，但不能为没有实现的分部方法生成委托。

C# 语言规范

有关详细信息，请参阅 C# 语言规范中的分部类型和分部方法。 该语言规范是 C# 语法和用法的权威资料。 分部方法的新功能在功能规范中定义。

另请参阅

- 类
- 结构类型
- 接口
- 分部（类型）

如何在查询中返回元素属性的子集（C# 编程指南）

项目 • 2024/03/13

当下列两个条件都满足时，可在查询表达式中使用匿名类型：

- 只想返回每个源元素的某些属性。
- 无需在执行查询的方法的范围之外存储查询结果。

如果只想从每个源元素中返回一个属性或字段，则只需在 `select` 子句中使用点运算符。例如，若要只返回每个 `student` 的 `ID`，可以按如下方式编写 `select` 子句：

C#

```
select student.ID;
```

示例

下面的示例演示如何使用匿名类型只返回每个源元素的符合指定条件的属性子集。

C#

```
private static void QueryByScore()
{
    // Create the query. var is required because
    // the query produces a sequence of anonymous types.
    var queryHighScores =
        from student in students
        where student.ExamScores[0] > 95
        select new { student.FirstName, student.LastName };

    // Execute the query.
    foreach (var obj in queryHighScores)
    {
        // The anonymous type's properties were not named. Therefore
        // they have the same names as the Student properties.
        Console.WriteLine(obj.FirstName + ", " + obj.LastName);
    }
}
/* Output:
Adams, Terry
Fakhouri, Fadi
Garcia, Cesar
Omelchenko, Svetlana
```

```
Zabokritski, Eugene
*/
```

请注意，如果未指定名称，匿名类型会使用源元素的名称作为其属性名称。 若要为匿名类型中的属性指定新名称，请按如下方式编写 `select` 语句：

C#

```
select new { First = student.FirstName, Last = student.LastName };
```

如果在上一个示例中这样做，则 `Console.WriteLine` 语句也必须更改：

C#

```
Console.WriteLine(student.First + " " + student.Last);
```

编译代码

要运行此代码，请使用 System.Linq 的 `using` 指令将该类复制并粘贴到 C# 控制台应用程序中。

另请参阅

- 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

显式接口实现（C# 编程指南）

项目 • 2024/03/13

如果一个类实现的两个接口包含签名相同的成员，则在该类上实现此成员会导致这两个接口将此成员用作其实现。 如下示例中，所有对 `Paint` 的调用皆调用同一方法。 第一个示例定义类型：

C#

```
public interface IControl
{
    void Paint();
}
public interface ISurface
{
    void Paint();
}
public class SampleClass : IControl, ISurface
{
    // Both ISurface.Paint and IControl.Paint call this method.
    public void Paint()
    {
        Console.WriteLine("Paint method in SampleClass");
    }
}
```

下面的示例调用方法：

C#

```
SampleClass sample = new SampleClass();
IControl control = sample;
ISurface surface = sample;

// The following lines all call the same method.
sample.Paint();
control.Paint();
surface.Paint();

// Output:
// Paint method in SampleClass
// Paint method in SampleClass
// Paint method in SampleClass
```

但你可能不希望为这两个接口都调用相同的实现。 若要调用不同的实现，根据所使用的接口，可以显式实现接口成员。 显式接口实现是一个类成员，只通过指定接口进行调用。 通过在类成员前面加上接口名称和句点可命名该类成员。 例如：

C#

```
public class SampleClass : IControl, ISurface
{
    void IControl.Paint()
    {
        System.Console.WriteLine("IControl.Paint");
    }
    void ISurface.Paint()
    {
        System.Console.WriteLine("ISurface.Paint");
    }
}
```

类成员 `IControl.Paint` 仅通过 `IControl` 接口可用，`ISurface.Paint` 仅通过 `ISurface` 可用。 这两个方法实现相互独立，两者均不可直接在类上使用。 例如：

C#

```
SampleClass sample = new SampleClass();
IControl control = sample;
ISurface surface = sample;

// The following lines all call the same method.
//sample.Paint(); // Compiler error.
control.Paint();  // Calls IControl.Paint on SampleClass.
surface.Paint();  // Calls ISurface.Paint on SampleClass.

// Output:
// IControl.Paint
// ISurface.Paint
```

显式实现还用于处理两个接口分别声明名称相同的不同成员（例如属性和方法）的情况。 若要实现两个接口，类必须对属性 `P` 或方法 `P` 使用显式实现，或对二者同时使用，从而避免编译器错误。 例如：

C#

```
interface ILeft
{
    int P { get;}
}
interface IRight
{
    int P();
}

class Middle : ILeft, IRight
{
    public int P() { return 0; }
```

```
    int ILeft.P { get { return 0; } }
}
```

显式接口实现没有访问修饰符，因为它不能作为其定义类型的成员进行访问。 而只能在通过接口实例调用时访问。 如果为显式接口实现指定访问修饰符，将收到编译器错误 CS0106。 有关详细信息，请参阅 interface（C# 参考）。

你可以为在接口中声明的成员定义一个实现。 如果类从接口继承方法实现，则只能通过接口类型的引用访问该方法。 继承的成员不会显示为公共接口的一部分。 下面的示例定义接口方法的默认实现：

C#

```
public interface IControl
{
    void Paint() => Console.WriteLine("Default Paint method");
}
public class SampleClass : IControl
{
    // Paint() is inherited from IControl.
}
```

下面的示例调用默认实现：

C#

```
var sample = new SampleClass();
//sample.Paint();// "Paint" isn't accessible.
var control = sample as IControl;
control.Paint();
```

任何实现 `IControl` 接口的类都可以重写默认的 `Paint` 方法，作为公共方法或作为显式接口实现。

另请参阅

- 面向对象的编程
- 接口
- 继承

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

如何显式实现接口成员（C# 编程指南）

项目 • 2024/03/13

本示例声明一个接口`IDimensions`和一个类`Box`，显式实现了接口成员`GetLength`和`GetWidth`。 通过接口实例`dimensions`访问这些成员。

示例

C#

```
interface IDimensions
{
    float GetLength();
    float GetWidth();
}

class Box : IDimensions
{
    float lengthInches;
    float widthInches;

    Box(float length, float width)
    {
        lengthInches = length;
        widthInches = width;
    }
    // Explicit interface member implementation:
    float IDimensions.GetLength()
    {
        return lengthInches;
    }
    // Explicit interface member implementation:
    float IDimensions.GetWidth()
    {
        return widthInches;
    }

    static void Main()
    {
        // Declare a class instance box1:
        Box box1 = new Box(30.0f, 20.0f);

        // Declare an interface instance dimensions:
        IDimensions dimensions = box1;

        // The following commented lines would produce compilation
        // errors because they try to access an explicitly implemented
        // interface member from a class instance:
        //System.Console.WriteLine("Length: {0}", box1.GetLength());
```

```
        //System.Console.WriteLine("Width: {0}", box1.GetWidth());

        // Print out the dimensions of the box by calling the methods
        // from an instance of the interface:
        System.Console.WriteLine("Length: {0}", dimensions.GetLength());
        System.Console.WriteLine("Width: {0}", dimensions.GetWidth());
    }
}
/* Output:
    Length: 30
    Width: 20
*/
```

可靠编程

- 请注意，注释掉了 `Main` 方法中以下行，因为它们将产生编译错误。 显式实现的接口成员不能从类实例访问：

  C#

  ```
  //System.Console.WriteLine("Length: {0}", box1.GetLength());
  //System.Console.WriteLine("Width: {0}", box1.GetWidth());
  ```

- 另请注意 `Main` 方法中的以下行成功输出了框的尺寸，因为这些方法是从接口实例调用的：

  C#

  ```
  System.Console.WriteLine("Length: {0}", dimensions.GetLength());
  System.Console.WriteLine("Width: {0}", dimensions.GetWidth());
  ```

另请参阅

- 面向对象的编程
- 接口
- 如何显式实现两个接口的成员

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

如何显式实现两个接口的成员（C# 编程指南）

项目 • 2024/03/13

显式接口实现还允许程序员实现具有相同成员名称的两个接口，并为每个接口成员各提供一个单独的实现。 本示例同时以公制单位和英制单位显示框的尺寸。 Box 类实现 IEnglishDimensions 和 IMetricDimensions 两个接口，它们表示不同的度量系统。 两个接口有相同的成员名称 Length 和 Width。

示例

C#

```
// Declare the English units interface:
interface IEnglishDimensions
{
    float Length();
    float Width();
}

// Declare the metric units interface:
interface IMetricDimensions
{
    float Length();
    float Width();
}

// Declare the Box class that implements the two interfaces:
// IEnglishDimensions and IMetricDimensions:
class Box : IEnglishDimensions, IMetricDimensions
{
    float lengthInches;
    float widthInches;

    public Box(float lengthInches, float widthInches)
    {
        this.lengthInches = lengthInches;
        this.widthInches = widthInches;
    }

    // Explicitly implement the members of IEnglishDimensions:
    float IEnglishDimensions.Length() => lengthInches;

    float IEnglishDimensions.Width() => widthInches;

    // Explicitly implement the members of IMetricDimensions:
    float IMetricDimensions.Length() => lengthInches * 2.54f;
```

```
    float IMetricDimensions.Width() => widthInches * 2.54f;

    static void Main()
    {
        // Declare a class instance box1:
        Box box1 = new Box(30.0f, 20.0f);

        // Declare an instance of the English units interface:
        IEnglishDimensions eDimensions = box1;

        // Declare an instance of the metric units interface:
        IMetricDimensions mDimensions = box1;

        // Print dimensions in English units:
        System.Console.WriteLine("Length(in): {0}", eDimensions.Length());
        System.Console.WriteLine("Width (in): {0}", eDimensions.Width());

        // Print dimensions in metric units:
        System.Console.WriteLine("Length(cm): {0}", mDimensions.Length());
        System.Console.WriteLine("Width (cm): {0}", mDimensions.Width());
    }
}
/* Output:
    Length(in): 30
    Width (in): 20
    Length(cm): 76.2
    Width (cm): 50.8
*/
```

可靠编程

如果希望默认度量采用英制单位，请正常实现 Length 和 Width 方法，并从 IMetricDimensions 接口显式实现 Length 和 Width 方法：

C#

```
// Normal implementation:
public float Length() => lengthInches;
public float Width() => widthInches;

// Explicit implementation:
float IMetricDimensions.Length() => lengthInches * 2.54f;
float IMetricDimensions.Width() => widthInches * 2.54f;
```

这种情况下，可以从类实例访问英制单位，从接口实例访问公制单位：

C#

```
public static void Test()
{
```

```
    Box box1 = new Box(30.0f, 20.0f);
    IMetricDimensions mDimensions = box1;

    System.Console.WriteLine("Length(in): {0}", box1.Length());
    System.Console.WriteLine("Width (in): {0}", box1.Width());
    System.Console.WriteLine("Length(cm): {0}", mDimensions.Length());
    System.Console.WriteLine("Width (cm): {0}", mDimensions.Width());
}
```

另请参阅

- 面向对象的编程
- 接口
- 如何显式实现接口成员

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

委托（C# 编程指南）

项目 • 2024/04/11

委托是一种引用类型，表示对具有特定参数列表和返回类型的方法的引用。 在实例化委托时，你可以将其实例与任何具有兼容签名和返回类型的方法相关联。 你可以通过委托实例调用方法。

委托用于将方法作为参数传递给其他方法。 事件处理程序就是通过委托调用的方法。 你可以创建一个自定义方法，当发生特定事件时，某个类（如 Windows 控件）就可以调用你的方法。 下面的示例演示了一个委托声明：

C#

```
public delegate int PerformCalculation(int x, int y);
```

可将任何可访问类或结构中与委托类型匹配的任何方法分配给委托。 该方法可以是静态方法，也可以是实例方法。 此灵活性意味着你可以通过编程方式来更改方法调用，还可以向现有类中插入新代码。

> **备注**
>
> 在方法重载的上下文中，方法的签名不包括返回值。 但在委托的上下文中，签名包括返回值。 换句话说，方法和委托必须具有相同的返回类型。

将方法作为参数进行引用的能力使委托成为定义回调方法的理想选择。 可编写一个比较应用程序中两个对象的方法。 该方法可用在排序算法的委托中。 由于比较代码与库分离，因此排序方法可能更常见。

对于类似的方案，已将函数指针添加到 C# 9，其中你需要对调用约定有更多的控制。 使用添加到委托类型的虚方法调用与委托关联的代码。 使用函数指针，可以指定不同的约定。

委托概述

委托具有以下属性：

- 委托类似于 C++ 函数指针，但委托完全面向对象，不像 C++ 指针会记住函数，委托会同时封装对象实例和方法。
- 委托允许将方法作为参数进行传递。
- 委托可用于定义回调方法。
- 委托可以链接在一起；例如，可以对一个事件调用多个方法。

- 方法不必与委托类型完全匹配。 有关详细信息，请参阅使用委托中的变体。
- 使用 Lambda 表达式可以更简练地编写内联代码块。 Lambda 表达式（在某些上下文中）可编译为委托类型。 若要详细了解 lambda 表达式，请参阅 lambda 表达式。

本节内容

- 使用委托
- 何时使用委托，而不是接口（C# 编程指南）
- 带有命名方法的委托与带有匿名方法的委托
- 使用委托中的变体
- 如何合并委托（多播委托）
- 如何声明、实例化和使用委托

C# 语言规范

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

重要章节

- C# 3.0 手册（第三版）：面向 C# 3.0 程序员的超过 250 个解决方案中的委托、事件和 Lambda 表达式
- 学习 C# 3.0：C# 3.0 基础知识中的委托和事件

另请参阅

- Delegate
- C# 编程指南
- 事件

反馈

此页面是否有帮助? 是 否

提供产品反馈

使用委托（C# 编程指南）

项目 • 2023/11/20

委托是安全封装方法的类型，类似于 C 和 C++ 中的函数指针。 与 C 函数指针不同的是，委托是面向对象的、类型安全的和可靠的。 委托的类型由委托的名称确定。 以下示例声明名为 `Callback` 的委托，该委托可以封装采用字符串作为参数并返回 void 的方法：

C#

```
public delegate void Callback(string message);
```

委托对象通常可采用两种方式进行构造，一种是提供委托将封装的方法的名称，另一种是使用 lambda 表达式。 以这种方式实例化委托后，可以调用该委托。 调用委托会调用附加到委托实例的方法。 调用方传递到委托的参数将传递到该方法，并且委托会将方法的返回值（如果有）返回到调用方。 例如：

C#

```
// Create a method for a delegate.
public static void DelegateMethod(string message)
{
    Console.WriteLine(message);
}
```

C#

```
// Instantiate the delegate.
Callback handler = DelegateMethod;

// Call the delegate.
handler("Hello World");
```

委托类型派生自 .NET 中的 Delegate 类。 委托类型是密封的，它们不能派生自 Delegate，也不能从其派生出自定义类。 由于实例化的委托是一个对象，因此可以作为实参传递或分配给一个属性。 这允许方法接受委托作为参数并在稍后调用委托。 这被称为异步回调，是在长进程完成时通知调用方的常用方法。 当以这种方式使用委托时，使用委托的代码不需要知道要使用的实现方法。 功能类似于封装接口提供的功能。

回调的另一个常见用途是定义自定义比较方法并将该委托传递到短方法。 它允许调用方的代码成为排序算法的一部分。 以下示例方法使用 `Del` 类型作为参数：

C#

```
public static void MethodWithCallback(int param1, int param2, Callback
callback)
{
    callback("The number is: " + (param1 + param2).ToString());
}
```

然后，你可以将上面创建的委托传递到该方法：

C#

```
MethodWithCallback(1, 2, handler);
```

并将以下输出接收到控制台：

控制台

```
The number is: 3
```

以抽象方式使用委托时，`MethodWithCallback` 不需要直接调用控制台，记住，其不必设计为具有控制台。 `MethodWithCallback` 的作用是简单准备字符串并将字符串传递到其他方法。 由于委托的方法可以使用任意数量的参数，此功能特别强大。

当委托构造为封装实例方法时，委托将同时引用实例和方法。 委托不知道除其所封装方法以外的实例类型，因此委托可以引用任何类型的对象，只要该对象上有与委托签名匹配的方法。 当委托构造为封装静态方法时，委托仅引用方法。 请考虑以下声明：

C#

```
public class MethodClass
{
    public void Method1(string message) { }
    public void Method2(string message) { }
}
```

加上之前显示的静态 `DelegateMethod`，我们现在已有三个 `Del` 实例可以封装的方法。

调用时，委托可以调用多个方法。 这被称为多播。 若要向委托的方法列表（调用列表）添加其他方法，只需使用加法运算符或加法赋值运算符（“+”或“+=”）添加两个委托。 例如：

C#

```
var obj = new MethodClass();
Callback d1 = obj.Method1;
Callback d2 = obj.Method2;
```

```
Callback d3 = DelegateMethod;

//Both types of assignment are valid.
Callback allMethodsDelegate = d1 + d2;
allMethodsDelegate += d3;
```

此时，`allMethodsDelegate` 的调用列表中包含三个方法，分别为 `Method1`、`Method2` 和 `DelegateMethod`。 原有的三个委托（`d1`、`d2` 和 `d3`）保持不变。 调用 `allMethodsDelegate` 时，将按顺序调用所有三个方法。 如果委托使用引用参数，引用将按相反的顺序传递到所有这三个方法，并且一种方法进行的任何更改都将在另一种方法上见到。 当方法引发未在方法内捕获到的异常时，该异常将传递到委托的调用方，并且不会调用调用列表中的后续方法。 如果委托具有返回值和/或输出参数，它将返回上次调用方法的返回值和参数。 若要删除调用列表中的方法，请使用减法运算符或减法赋值运算符（`-` 或 `-=`）。 例如：

C#

```
//remove Method1
allMethodsDelegate -= d1;

// copy AllMethodsDelegate while removing d2
Callback oneMethodDelegate = allMethodsDelegate - d2;
```

由于委托类型派生自 `System.Delegate`，因此可以在委托上调用该类定义的方法和属性。 例如，若要查询委托调用列表中方法的数量，你可以编写：

C#

```
int invocationCount = d1.GetInvocationList().GetLength(0);
```

调用列表中具有多个方法的委托派生自 MulticastDelegate，该类属于 `System.Delegate` 的子类。 由于这两个类都支持 `GetInvocationList`，因此在其他情况下，上述代码也将产生作用。

多播委托广泛用于事件处理中。 事件源对象将事件通知发送到已注册接收该事件的接收方对象。 若要注册一个事件，接收方需要创建用于处理该事件的方法，然后为该方法创建委托并将委托传递到事件源。 事件发生时，源调用委托。 然后，委托将对接收方调用事件处理方法，从而提供事件数据。 给定事件的委托类型由事件源确定。 有关详细信息，请参阅事件。

在编译时比较分配的两个不同类型的委托将导致编译错误。 如果委托实例是静态的 `System.Delegate` 类型，则允许比较，但在运行时将返回 false。 例如：

C#

```
delegate void Callback1();
delegate void Callback2();

static void method(Callback1 d, Callback2 e, System.Delegate f)
{
    // Compile-time error.
    //Console.WriteLine(d == e);

    // OK at compile-time. False if the run-time type of f
    // is not the same as that of d.
    Console.WriteLine(d == f);
}
```

另请参阅

- C# 编程指南
- 委托
- 使用委托中的变体
- 委托中的变体
- 对 Func 和 Action 泛型委托使用变体
- 事件

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

.NET feedback

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

提出文档问题

提供产品反馈

带有命名方法的委托与匿名方法（C# 编程指南）

项目 • 2025/01/06

委托可以与命名方法相关联。 使用命名方法实例化委托时，该方法作为参数传递，例如：

C#

```
// Declare a delegate.
delegate void WorkCallback(int x);

// Define a named method.
void DoWork(int k) { /* ... */ }

// Instantiate the delegate using the method as a parameter.
WorkCallback d = obj.DoWork;
```

这称为使用命名方法。 使用命名方法构造的委托可以封装静态方法或实例方法。 命名方法是在早期版本的 C# 中实例化委托的唯一方式。 但是，如果创建新方法会造成多余开销，C# 允许你实例化委托并立即指定调用委托时委托将处理的代码块。 代码块可包含 Lambda 表达式或匿名方法。

作为委托参数传递的方法必须具有与委托声明相同的签名。 委托实例可以封装静态方法或实例方法。

> **备注**
>
> 尽管委托可以使用 out 参数，但不建议将该委托与多播事件委托配合使用，因为你无法知道将调用哪个委托。

具有单个重载的方法组具有*自然类型*。 这意味着编译器可以推断委托类型的返回类型和参数类型：

C#

```
var read = Console.Read; // Just one overload; Func<int> inferred
var write = Console.Write; // ERROR: Multiple overloads, can't choose
```

示例

以下是声明和使用委托的简单示例。 请注意，委托 `MultiplyCallback` 与关联的方法 `MultiplyNumbers` 具有相同的签名

C#

```
// Declare a delegate
delegate void MultiplyCallback(int i, double j);

class MathClass
{
    static void Main()
    {
        MathClass m = new MathClass();

        // Delegate instantiation using "MultiplyNumbers"
        MultiplyCallback d = m.MultiplyNumbers;

        // Invoke the delegate object.
        Console.WriteLine("Invoking the delegate using 'MultiplyNumbers':");
        for (int i = 1; i <= 5; i++)
        {
            d(i, 2);
        }

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }

    // Declare the associated method.
    void MultiplyNumbers(int m, double n)
    {
        Console.Write(m * n + " ");
    }
}
/* Output:
    Invoking the delegate using 'MultiplyNumbers':
    2 4 6 8 10
*/
```

在下面的示例中，一个委托映射到静态方法和实例方法，并返回来自两种方法的具体信息。

C#

```
// Declare a delegate
delegate void Callback();

class SampleClass
{
    public void InstanceMethod()
    {
```

```
        Console.WriteLine("A message from the instance method.");
    }

    static public void StaticMethod()
    {
        Console.WriteLine("A message from the static method.");
    }
}

class TestSampleClass
{
    static void Main()
    {
        var sc = new SampleClass();

        // Map the delegate to the instance method:
        Callback d = sc.InstanceMethod;
        d();

        // Map to the static method:
        d = SampleClass.StaticMethod;
        d();
    }
}
/* Output:
    A message from the instance method.
    A message from the static method.
*/
```

另请参阅

- 委托
- 如何合并委托（多播委托）
- 事件

如何合并委托（多播委托）（C# 编程指南）

项目 • 2024/03/13

此示例演示如何创建多播委托。 委托对象的一个有用属性在于可通过使用 `+` 运算符将多个对象分配到一个委托实例。 多播委托包含已分配委托列表。 此多播委托被调用时会依次调用列表中的委托。 仅可合并类型相同的委托。

`-` 运算符可用于从多播委托中删除组件委托。

示例

C#

```
using System;

// Define a custom delegate that has a string parameter and returns void.
delegate void CustomCallback(string s);

class TestClass
{
    // Define two methods that have the same signature as CustomCallback.
    static void Hello(string s)
    {
        Console.WriteLine($"  Hello, {s}!");
    }

    static void Goodbye(string s)
    {
        Console.WriteLine($"  Goodbye, {s}!");
    }

    static void Main()
    {
        // Declare instances of the custom delegate.
        CustomCallback hiDel, byeDel, multiDel, multiMinusHiDel;

        // In this example, you can omit the custom delegate if you
        // want to and use Action<string> instead.
        //Action<string> hiDel, byeDel, multiDel, multiMinusHiDel;

        // Initialize the delegate object hiDel that references the
        // method Hello.
        hiDel = Hello;

        // Initialize the delegate object byeDel that references the
        // method Goodbye.
        byeDel = Goodbye;
```

```
        // The two delegates, hiDel and byeDel, are combined to
        // form multiDel.
        multiDel = hiDel + byeDel;

        // Remove hiDel from the multicast delegate, leaving byeDel,
        // which calls only the method Goodbye.
        multiMinusHiDel = multiDel - hiDel;

        Console.WriteLine("Invoking delegate hiDel:");
        hiDel("A");
        Console.WriteLine("Invoking delegate byeDel:");
        byeDel("B");
        Console.WriteLine("Invoking delegate multiDel:");
        multiDel("C");
        Console.WriteLine("Invoking delegate multiMinusHiDel:");
        multiMinusHiDel("D");
    }
}
/* Output:
Invoking delegate hiDel:
  Hello, A!
Invoking delegate byeDel:
  Goodbye, B!
Invoking delegate multiDel:
  Hello, C!
  Goodbye, C!
Invoking delegate multiMinusHiDel:
  Goodbye, D!
*/
```

另请参阅

- MulticastDelegate
- 事件

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

如何声明、实例化和使用委托（C# 编程指南）

项目 • 2023/04/07

可以使用以下任一方法声明委托：

- 使用匹配签名声明委托类型并声明方法：

C#

```
// Declare a delegate.
delegate void Del(string str);

// Declare a method with the same signature as the delegate.
static void Notify(string name)
{
    Console.WriteLine($"Notification received for: {name}");
}
```

C#

```
// Create an instance of the delegate.
Del del1 = new Del(Notify);
```

- 将方法组分配给委托类型：

C#

```
// C# 2.0 provides a simpler way to declare an instance of Del.
Del del2 = Notify;
```

- 声明匿名方法：

C#

```
// Instantiate Del by using an anonymous method.
Del del3 = delegate(string name)
    { Console.WriteLine($"Notification received for: {name}"); };
```

- 使用 lambda 表达式：

C#

```
// Instantiate Del by using a lambda expression.
Del del4 = name =>  { Console.WriteLine($"Notification received for:
{name}"); };
```

有关详细信息，请参阅 Lambda 表达式。

下面的示例演示如何声明、实例化和使用委托。 `BookDB` 类封装用来维护书籍数据库的书店数据库。 它公开一个方法 `ProcessPaperbackBooks`，用于在数据库中查找所有平装书并为每本书调用委托。 使用的 `delegate` 类型名为 `ProcessBookCallback`。 `Test` 类使用此类打印平装书的书名和平均价格。

使用委托提升书店数据库和客户端代码之间的良好分隔功能。 客户端代码程序不知道如何存储书籍或书店代码如何查找平装书。 书店代码不知道它在找到平装书之后对其执行什么处理。

示例

C#

```
// A set of classes for handling a bookstore:
namespace Bookstore
{
    using System.Collections;

    // Describes a book in the book list:
    public struct Book
    {
        public string Title;        // Title of the book.
        public string Author;       // Author of the book.
        public decimal Price;       // Price of the book.
        public bool Paperback;      // Is it paperback?

        public Book(string title, string author, decimal price, bool
paperBack)
        {
            Title = title;
            Author = author;
            Price = price;
            Paperback = paperBack;
        }
    }

    // Declare a delegate type for processing a book:
    public delegate void ProcessBookCallback(Book book);

    // Maintains a book database.
    public class BookDB
    {
```

```
        // List of all books in the database:
        ArrayList list = new ArrayList();

        // Add a book to the database:
        public void AddBook(string title, string author, decimal price, bool
paperBack)
        {
            list.Add(new Book(title, author, price, paperBack));
        }

        // Call a passed-in delegate on each paperback book to process it:
        public void ProcessPaperbackBooks(ProcessBookCallback processBook)
        {
            foreach (Book b in list)
            {
                if (b.Paperback)
                    // Calling the delegate:
                    processBook(b);
            }
        }
    }
}

// Using the Bookstore classes:
namespace BookTestClient
{
    using Bookstore;

    // Class to total and average prices of books:
    class PriceTotaller
    {
        int countBooks = 0;
        decimal priceBooks = 0.0m;

        internal void AddBookToTotal(Book book)
        {
            countBooks += 1;
            priceBooks += book.Price;
        }

        internal decimal AveragePrice()
        {
            return priceBooks / countBooks;
        }
    }

    // Class to test the book database:
    class Test
    {
        // Print the title of the book.
        static void PrintTitle(Book b)
        {
            Console.WriteLine($"   {b.Title}");
        }
```

```
        // Execution starts here.
        static void Main()
        {
            BookDB bookDB = new BookDB();

            // Initialize the database with some books:
            AddBooks(bookDB);

            // Print all the titles of paperbacks:
            Console.WriteLine("Paperback Book Titles:");

            // Create a new delegate object associated with the static
            // method Test.PrintTitle:
            bookDB.ProcessPaperbackBooks(PrintTitle);

            // Get the average price of a paperback by using
            // a PriceTotaller object:
            PriceTotaller totaller = new PriceTotaller();

            // Create a new delegate object associated with the nonstatic
            // method AddBookToTotal on the object totaller:
            bookDB.ProcessPaperbackBooks(totaller.AddBookToTotal);

            Console.WriteLine("Average Paperback Book Price: ${0:#.##}",
                    totaller.AveragePrice());
        }

        // Initialize the book database with some test books:
        static void AddBooks(BookDB bookDB)
        {
            bookDB.AddBook("The C Programming Language", "Brian W. Kernighan
and Dennis M. Ritchie", 19.95m, true);
            bookDB.AddBook("The Unicode Standard 2.0", "The Unicode
Consortium", 39.95m, true);
            bookDB.AddBook("The MS-DOS Encyclopedia", "Ray Duncan", 129.95m,
false);
            bookDB.AddBook("Dogbert's Clues for the Clueless", "Scott
Adams", 12.00m, true);
        }
    }
}
/* Output:
Paperback Book Titles:
   The C Programming Language
   The Unicode Standard 2.0
   Dogbert's Clues for the Clueless
Average Paperback Book Price: $23.97
*/
```

可靠编程

- 声明委托。

以下语句声明新的委托类型。

```
C#

public delegate void ProcessBookCallback(Book book);
```

每个委托类型描述自变量的数量和类型，以及它可以封装的方法的返回值类型。 每当需要一组新的自变量类型或返回值类型，则必须声明一个新的委托类型。

- 实例化委托。

  声明委托类型后，则必须创建委托对象并将其与特定的方法相关联。 在上例中，你通过将 `PrintTitle` 方法传递给 `ProcessPaperbackBooks` 方法执行此操作，如下面的示例所示：

  ```
  C#

  bookDB.ProcessPaperbackBooks(PrintTitle);
  ```

  这将创建一个新的与静态方法 `Test.PrintTitle` 关联的委托对象。 同样，如下面的示例所示，传递对象 `totaller` 中的非静态方法 `AddBookToTotal`：

  ```
  C#

  bookDB.ProcessPaperbackBooks(totaller.AddBookToTotal);
  ```

  在这两种情况下，都将新的委托对象传递给 `ProcessPaperbackBooks` 方法。

  创建委托后，它与之关联的方法就永远不会更改；委托对象是不可变的。

- 调用委托。

  创建委托对象后，通常会将委托对象传递给将调用该委托的其他代码。 委托对象是通过使用委托对象的名称调用的，后跟用圆括号括起来的将传递给委托的自变量。 下面是一个委托调用示例：

  ```
  C#

  processBook(b);
  ```

  委托可以同步调用（如在本例中）或通过使用 `BeginInvoke` 和 `EndInvoke` 方法异步调用。

另请参阅

- C# 编程指南
- 事件
- 委托

字符串和字符串字面量

项目 • 2024/04/11

字符串是值为文本的 String 类型对象。 文本在内部存储为 Char 对象的依序只读集合。 在 C# 字符串末尾没有 null 终止字符；因此，一个 C# 字符串可以包含任何数量的嵌入的 null 字符 ('\0')。 字符串的 Length 属性表示其包含的 `Char` 对象数量，而非 Unicode 字符数。 若要访问字符串中的各个 Unicode 码位，请使用 StringInfo 对象。

string 与System.String

在 C# 中，`string` 关键字是 String 的别名。 因此，`String` 和 `string` 是等效的（虽然建议使用提供的别名 `string`），因为即使不使用 `using System;`，它也能正常工作。 `String` 类提供了安全创建、操作和比较字符串的多种方法。 此外，C# 语言重载了部分运算符，以简化常见字符串操作。 有关关键字的详细信息，请参阅 string。 有关类型及其方法的详细信息，请参阅 String。

声明和初始化字符串

可以使用各种方法声明和初始化字符串，如以下示例中所示：

C#

```
// Declare without initializing.
string message1;

// Initialize to null.
string message2 = null;

// Initialize as an empty string.
// Use the Empty constant instead of the literal "".
string message3 = System.String.Empty;

// Initialize with a regular string literal.
string oldPath = "c:\\Program Files\\Microsoft Visual Studio 8.0";

// Initialize with a verbatim string literal.
string newPath = @"c:\Program Files\Microsoft Visual Studio 9.0";

// Use System.String if you prefer.
System.String greeting = "Hello World!";

// In local variables (i.e. within a method body)
// you can use implicit typing.
var temp = "I'm still a strongly-typed System.String!";
```

```
// Use a const string to prevent 'message4' from
// being used to store another string value.
const string message4 = "You can't get rid of me!";

// Use the String constructor only when creating
// a string from a char*, char[], or sbyte*. See
// System.String documentation for details.
char[] letters = { 'A', 'B', 'C' };
string alphabet = new string(letters);
```

不要使用 new 运算符创建字符串对象，除非使用字符数组初始化字符串。

使用 Empty 常量值初始化字符串，以新建字符串长度为零的 String 对象。 长度为零的字符串文本表示法是""。 通过使用 Empty 值（而不是 null）初始化字符串，可以减少 NullReferenceException 发生的可能性。 尝试访问字符串前，先使用静态 IsNullOrEmpty(String) 方法验证字符串的值。

字符串的不可变性

字符串对象是"不可变的"：它们在创建后无法更改。 看起来是在修改字符串的所有 String 方法和 C# 运算符实际上都是在新的字符串对象中返回结果。 在下面的示例中，当 `s1` 和 `s2` 的内容被串联在一起以形成单个字符串时，两个原始字符串没有被修改。 `+=` 运算符创建一个新的字符串，其中包含组合的内容。 这个新对象被分配给变量 `s1`，而分配给 `s1` 的原始对象被释放，以供垃圾回收，因为没有任何其他变量包含对它的引用。

C#

```
string s1 = "A string is more ";
string s2 = "than the sum of its chars.";

// Concatenate s1 and s2. This actually creates a new
// string object and stores it in s1, releasing the
// reference to the original object.
s1 += s2;

System.Console.WriteLine(s1);
// Output: A string is more than the sum of its chars.
```

由于字符串"modification"实际上是一个新创建的字符串，因此，必须在创建对字符串的引用时使用警告。 如果创建了字符串的引用，然后"修改"了原始字符串，则该引用将继续指向原始对象，而非指向修改字符串时所创建的新对象。 以下代码阐释了此行为：

C#

```
string str1 = "Hello ";
string str2 = str1;
```

```
str1 += "World";

System.Console.WriteLine(str2);
//Output: Hello
```

有关如何创建基于修改的新字符串的详细信息，例如原始字符串上的搜索和替换操作，请参阅如何修改字符串内容。

带引号的字符串字面量

带引号的字符串字面量在同一行上以单个双引号字符 (") 开头和结尾。 带引号的字符串字面量最适合匹配单个行且不包含任何转义序列的字符串。 带引号的字符串字面量必须嵌入转义字符，如以下示例所示：

C#

```
string columns = "Column 1\tColumn 2\tColumn 3";
//Output: Column 1        Column 2        Column 3

string rows = "Row 1\r\nRow 2\r\nRow 3";
/* Output:
    Row 1
    Row 2
    Row 3
*/

string title = "\"The \u00C6olean Harp\", by Samuel Taylor Coleridge";
//Output: "The Æolean Harp", by Samuel Taylor Coleridge
```

逐字字符串文本

对于多行字符串、包含反斜杠字符或嵌入双引号的字符串，逐字字符串字面量更方便。 逐字字符串将新的行字符作为字符串文本的一部分保留。 使用双引号在逐字字符串内部嵌入引号。 下面的示例演示逐字字符串的一些常见用法：

C#

```
string filePath = @"C:\Users\scoleridge\Documents\";
//Output: C:\Users\scoleridge\Documents\

string text = @"My pensive SARA ! thy soft cheek reclined
    Thus on mine arm, most soothing sweet it is
    To sit beside our Cot,...";
/* Output:
My pensive SARA ! thy soft cheek reclined
    Thus on mine arm, most soothing sweet it is
```

```
    To sit beside our Cot,...
*/

string quote = @"Her name was ""Sara.""";
//Output: Her name was "Sara."
```

原始字符串文本

从 C# 11 开始，可以使用原始字符串字面量更轻松地创建多行字符串，或使用需要转义序列的任何字符。 原始字符串字面量无需使用转义序列。 你可以编写字符串，包括空格格式，以及你希望在输出中显示该字符串的方式。 原始字符串字面量：

- 以至少三个双引号字符序列（`"""`）开头和结尾。 可以使用三个以上的连续字符开始和结束序列，以支持包含三个（或更多）重复引号字符的字符串字面量。
- 单行原始字符串字面量需要左引号和右引号字符位于同一行上。
- 多行原始字符串字面量需要左引号和右引号字符位于各自的行上。
- 在多行原始字符串字面量中，会删除右引号左侧的任何空格。

以下示例演示了这些规则：

C#

```
string singleLine = """Friends say "hello" as they pass by.""";
string multiLine = """
    "Hello World!" is typically the first program someone writes.
    """;
string embeddedXML = """
       <element attr = "content">
           <body style="normal">
               Here is the main text
           </body>
           <footer>
               Excerpts from "An amazing story"
           </footer>
       </element >
       """;
// The line "<element attr = "content">" starts in the first column.
// All whitespace left of that column is removed from the string.

string rawStringLiteralDelimiter = """"
    Raw string literals are delimited
    by a string of at least three double quotes,
    like this: """
    """";
```

以下示例演示了基于这些规则报告的编译器错误：

C#

```
// CS8997: Unterminated raw string literal.
var multiLineStart = """This
    is the beginning of a string
    """;

// CS9000: Raw string literal delimiter must be on its own line.
var multiLineEnd = """
    This is the beginning of a string """;

// CS8999: Line does not start with the same whitespace as the closing line
// of the raw string literal
var noOutdenting = """
    A line of text.
Trying to outdent the second line.
    """;
```

前两个示例无效，因为多行原始字符串字面量需要让左引号和右引号序列在其自己的行上。 第三个示例无效，因为文本已从右引号序列中缩进。

使用带引号的字符串字面量或逐字字符串字面量时，如果生成的文本包括需要转义序列的字符，应考虑原始字符串字面量。 原始字符串字面量将更易于你和其他人阅读，因为它更类似于输出文本。 例如，请考虑包含格式化 JSON 字符串的以下代码：

C#

```
string jsonString = """
{
  "Date": "2019-08-01T00:00:00-07:00",
  "TemperatureCelsius": 25,
  "Summary": "Hot",
  "DatesAvailable": [
    "2019-08-01T00:00:00-07:00",
    "2019-08-02T00:00:00-07:00"
  ],
  "TemperatureRanges": {
    "Cold": {
      "High": 20,
      "Low": -10
    },
    "Hot": {
      "High": 60,
      "Low": 20
    }
            },
  "SummaryWords": [
    "Cool",
    "Windy",
    "Humid"
  ]
```

```
}
""";
```

将该文本与 JSON 序列化示例中的等效文本（没有使用此新功能）进行比较。

字符串转义序列

展开表

转义序列	字符名称	Unicode 编码
\'	单引号	0x0027
\"	双引号	0x0022
\	反斜杠	0x005C
\0	null	0x0000
\a	警报	0x0007
\b	Backspace	0x0008
\f	换页	0x000C
\n	换行	0x000A
\r	回车	0x000D
\t	水平制表符	0x0009
\v	垂直制表符	0x000B
\u	Unicode 转义序列 (UTF-16)	`\uHHHH`（范围：0000 - FFFF；示例：`\u00E7` =“ç”）
\U	Unicode 转义序列 (UTF-32)	`\U00HHHHHH`（范围：000000 - 10FFFF；示例：`\U0001F47D` = "👽"）
\x	除长度可变外，Unicode 转义序列与“\u”类似	`\xH[H][H][H]`（范围：0 - FFFF；示例：`\x00E7`、`\x0E7` 或 `\xE7` =“ç”）

> **⚠ 警告**
>
> 使用 `\x` 转义序列且指定的位数小于 4 个十六进制数字时，如果紧跟在转义序列后面的字符是有效的十六进制数字（即 0-9、A-F 和 a-f），则这些字符将被解释为转义序列的一部分。 例如，`\xA1` 会生成“¡”，即码位 U+00A1。 但是，如果下一个字符是“A”或“a”，则转义序列将转而被解释为 `\xA1A` 并生成“ਚ”（即码位 U+0A1A）。

在此类情况下，如果指定全部 4 个十六进制数字（例如 `\x00A1`），则可能导致解释出错。

备注

在编译时，逐字字符串被转换为普通字符串，并具有所有相同的转义序列。 因此，如果在调试器监视窗口中查看逐字字符串，将看到由编译器添加的转义字符，而不是来自你的源代码的逐字字符串版本。 例如，原义字符串 `@"C:\files.txt"` 在监视窗口中显示为“C:\files.txt”。

格式字符串

格式字符串是在运行时以动态方式确定其内容的字符串。 格式字符串是通过将内插表达式或占位符嵌入字符串大括号内创建的。 大括号（`{...}`）中的所有内容都将解析为一个值，并在运行时以格式化字符串的形式输出。 有两种方法创建格式字符串：字符串内插和复合格式。

字符串内插

在 C# 6.0 及更高版本中提供，内插字符串由 `$` 特殊字符标识，并在大括号中包含内插表达式。 如果不熟悉字符串内插，请参阅字符串内插 - C# 交互式教程快速概览。

使用字符串内插来改善代码的可读性和可维护性。 字符串内插可实现与 `String.Format` 方法相同的结果，但提高了易用性和内联清晰度。

C#

```
var jh = (firstName: "Jupiter", lastName: "Hammon", born: 1711, published:
1761);
Console.WriteLine($"{jh.firstName} {jh.lastName} was an African American
poet born in {jh.born}.");
Console.WriteLine($"He was first published in {jh.published} at the age of
{jh.published - jh.born}.");
Console.WriteLine($"He'd be over {Math.Round((2018d - jh.born) / 100d) *
100d} years old today.");

// Output:
// Jupiter Hammon was an African American poet born in 1711.
// He was first published in 1761 at the age of 50.
// He'd be over 300 years old today.
```

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，可以将原始字符串字面量与字符串内插结合使用。 使用三个或更多个连续双引号开始和结束格式字符串。 如果输出字符串应包含 `{` 或 `}` 字符，则可以使用额外的 `$` 字符来指定开始和结束内插的 `{` 和 `}` 字符数。 输出中包含任何更少的 `{` 或 `}` 字符序列。 以下示例演示了如何使用该功能来显示点与原点的距离，以及如何将点置于大括号中：

C#

```
int X = 2;
int Y = 3;

var pointMessage = $$"""The point {{{X}}, {{Y}}} is {{Math.Sqrt(X * X + Y * Y)}} from the origin.""";

Console.WriteLine(pointMessage);
// Output:
// The point {2, 3} is 3.605551275463989 from the origin.
```

复合格式设置

String.Format 利用大括号中的占位符创建格式字符串。 此示例生成与上面使用的字符串内插方法类似的输出。

C#

```
var pw = (firstName: "Phillis", lastName: "Wheatley", born: 1753, published: 1773);
Console.WriteLine("{0} {1} was an African American poet born in {2}.",
pw.firstName, pw.lastName, pw.born);
Console.WriteLine("She was first published in {0} at the age of {1}.",
pw.published, pw.published - pw.born);
Console.WriteLine("She'd be over {0} years old today.", Math.Round((2018d - pw.born) / 100d) * 100d);

// Output:
// Phillis Wheatley was an African American poet born in 1753.
// She was first published in 1773 at the age of 20.
// She'd be over 300 years old today.
```

有关设置 .NET 类型格式的详细信息，请参阅 .NET 中的格式设置类型。

子字符串

子字符串是包含在字符串中的任何字符序列。 使用 Substring 方法可以通过原始字符串的一部分新建字符串。 可以使用 IndexOf 方法搜索一次或多次出现的子字符串。 使用 Replace 方法可以将出现的所有指定子字符串替换为新字符串。 与 Substring 方法一样，Replace 实际返回的是新字符串，且不修改原始字符串。 有关详细信息，请参阅如何搜索字符串和如何修改字符串内容。

C#

```
string s3 = "Visual C# Express";
System.Console.WriteLine(s3.Substring(7, 2));
// Output: "C#"

System.Console.WriteLine(s3.Replace("C#", "Basic"));
// Output: "Visual Basic Express"

// Index values are zero-based
int index = s3.IndexOf("C");
// index = 7
```

访问单个字符

可以使用包含索引值的数组表示法来获取对单个字符的只读访问权限，如下面的示例中所示：

C#

```
string s5 = "Printing backwards";

for (int i = 0; i < s5.Length; i++)
{
    System.Console.Write(s5[s5.Length - i - 1]);
}
// Output: "sdrawkcab gnitnirP"
```

如果 String 方法不提供修改字符串中的各个字符所需的功能，可以使用 StringBuilder 对象“就地”修改各个字符，再新建字符串来使用 StringBuilder 方法存储结果。 在下面的示例中，假定必须以特定方式修改原始字符串，然后存储结果以供未来使用：

C#

```
string question = "hOW DOES mICROSOFT wORD DEAL WITH THE cAPS lOCK KEY?";
System.Text.StringBuilder sb = new System.Text.StringBuilder(question);

for (int j = 0; j < sb.Length; j++)
{
    if (System.Char.IsLower(sb[j]) == true)
        sb[j] = System.Char.ToUpper(sb[j]);
```

```
    else if (System.Char.IsUpper(sb[j]) == true)
        sb[j] = System.Char.ToLower(sb[j]);
}
// Store the new string.
string corrected = sb.ToString();
System.Console.WriteLine(corrected);
// Output: How does Microsoft Word deal with the Caps Lock key?
```

Null 字符串和空字符串

空字符串是包含零个字符的 System.String 对象实例。 空字符串常用在各种编程方案中，表示空文本字段。 可以对空字符串调用方法，因为它们是有效的 System.String 对象。 对空字符串进行了初始化，如下所示：

C#

```
string s = String.Empty;
```

相比较而言，null 字符串并不指 System.String 对象实例，只要尝试对 null 字符串调用方法，都会引发 NullReferenceException。 但是，可以在串联和与其他字符串的比较操作中使用 null 字符串。 以下示例说明了对 null 字符串的引用会引发和不会引发意外的某些情况：

C#

```
string str = "hello";
string nullStr = null;
string emptyStr = String.Empty;

string tempStr = str + nullStr;
// Output of the following line: hello
Console.WriteLine(tempStr);

bool b = (emptyStr == nullStr);
// Output of the following line: False
Console.WriteLine(b);

// The following line creates a new empty string.
string newStr = emptyStr + nullStr;

// Null strings and empty strings behave differently. The following
// two lines display 0.
Console.WriteLine(emptyStr.Length);
Console.WriteLine(newStr.Length);
// The following line raises a NullReferenceException.
//Console.WriteLine(nullStr.Length);

// The null character can be displayed and counted, like other chars.
```

```
string s1 = "\x0" + "abc";
string s2 = "abc" + "\x0";
// Output of the following line: * abc*
Console.WriteLine("*" + s1 + "*");
// Output of the following line: *abc *
Console.WriteLine("*" + s2 + "*");
// Output of the following line: 4
Console.WriteLine(s2.Length);
```

使用 stringBuilder 快速创建字符串

.NET 中的字符串操作进行了高度的优化，在大多数情况下不会显著影响性能。 但是，在某些情况下（例如，执行数百次或数千次的紧密循环），字符串操作可能影响性能。 StringBuilder 类创建字符串缓冲区，用于在程序执行多个字符串操控时提升性能。 使用 StringBuilder 字符串，还可以重新分配各个字符，而内置字符串数据类型则不支持这样做。 例如，此代码更改字符串的内容，而无需创建新的字符串：

C#

```
System.Text.StringBuilder sb = new System.Text.StringBuilder("Rat: the ideal pet");
sb[0] = 'C';
System.Console.WriteLine(sb.ToString());
//Outputs Cat: the ideal pet
```

在以下示例中，StringBuilder 对象用于通过一组数字类型创建字符串：

C#

```
var sb = new StringBuilder();

// Create a string composed of numbers 0 - 9
for (int i = 0; i < 10; i++)
{
    sb.Append(i.ToString());
}
Console.WriteLine(sb);  // displays 0123456789

// Copy one character of the string (not possible with a System.String)
sb[0] = sb[9];

Console.WriteLine(sb);  // displays 9123456789
```

字符串、扩展方法和 LINQ

由于 String 类型实现 IEnumerable<T>，因此可以对字符串使用 Enumerable 类中定义的扩展方法。 为了避免视觉干扰，这些方法已从 String 类型的 IntelliSense 中排除，但它们仍然可用。 还可以使用字符串上的 LINQ 查询表达式。 有关详细信息，请参阅 LINQ 和字符串。

相关文章

- 如何修改字符串内容：阐明了转换字符串和修改字符串内容的方法。
- 如何比较字符串：演示了如何对字符串执行序号和特定于区域性的比较。
- 如何串联多个字符串：演示了将多个字符串联接成一个字符串的多种方式。
- 如何使用 String.Split 分析字符串：包含代码示例，演示了如何使用 String.Split 方法来分析字符串。
- 如何搜索字符串：说明了如何在字符串中使用搜索来搜索特定的文本或模式。
- 如何确定字符串是否表示数值：演示了如何安全地分析一个字符串，以查看其是否具有有效的数值。
- 字符串内插：介绍了字符串内插功能，它提供了一种方便的语法来格式化字符串。
- 基本字符串操作：提供了介绍如何使用 System.String和 System.Text.StringBuilder 方法执行基本字符串操作的文章链接。
- 分析字符串：介绍了如何将 .NET 基类型的字符串表示形式转换为相应类型的实例。
- 在 .NET 中分析日期和时间字符串：展示了如何将字符串（如“01/24/2008”）转换为 System.DateTime 对象。
- 比较字符串：包括有关如何比较字符串的信息，并提供 C# 和 Visual Basic 中的示例。
- 使用 StringBuilder 类：介绍了如何使用 StringBuilder 类来创建和修改动态字符串对象。
- LINQ 和字符串：提供了有关如何使用 LINQ 查询来执行各种字符串操作的信息。

反馈

此页面是否有帮助? 是 否

提供产品反馈

如何确定字符串是否表示数值（C# 编程指南）

项目 • 2023/04/08

若要确定字符串是否是指定数值类型的有效表示形式，请使用由所有基元数值类型以及如 DateTime 和 IPAddress 等类型实现的静态 `TryParse` 方法。 以下示例演示如何确定“108”是否为有效的 int。

C#

```
int i = 0;
string s = "108";
bool result = int.TryParse(s, out i); //i now = 108
```

如果该字符串包含非数字字符，或者数值对于指定的特定类型而言太大或太小，则 `TryParse` 将返回 false 并将 out 参数设置为零。 否则，它将返回 true 并将 out 参数设置为字符串的数值。

> **备注**
>
> 字符串可能仅包含数字字符，但对于你使用的 `TryParse` 方法的类型仍然无效。 例如，“256”不是 `byte` 的有效值，但对 `int` 有效。 “98.6”不是 `int` 的有效值，但它是有效的 `decimal`。

示例

以下示例演示如何对 `long`、`byte` 和 `decimal` 值的字符串表示形式使用 `TryParse`。

C#

```
string numString = "1287543"; //"1287543.0" will return false for a long
long number1 = 0;
bool canConvert = long.TryParse(numString, out number1);
if (canConvert == true)
Console.WriteLine("number1 now = {0}", number1);
else
Console.WriteLine("numString is not a valid long");

byte number2 = 0;
numString = "255"; // A value of 256 will return false
canConvert = byte.TryParse(numString, out number2);
```

```
if (canConvert == true)
Console.WriteLine("number2 now = {0}", number2);
else
Console.WriteLine("numString is not a valid byte");

decimal number3 = 0;
numString = "27.3"; //"27" is also a valid decimal
canConvert = decimal.TryParse(numString, out number3);
if (canConvert == true)
Console.WriteLine("number3 now = {0}", number3);
else
Console.WriteLine("number3 is not a valid decimal");
```

可靠编程

基元数值类型还实现 `Parse` 静态方法，如果字符串不是有效数字，该方法将引发异常。`TryParse` 通常更高效，因为如果数值无效，它仅返回 false。

.NET 安全性

请务必使用 `TryParse` 或 `Parse` 方法验证控件（如文本框和组合框）中的用户输入。

请参阅

- 如何将字节数组转换为 int
- 如何将字符串转换为数字
- 如何在十六进制字符串与数值类型之间转换
- 分析数值字符串
- 格式设置类型

索引器（C# 编程指南）

项目 • 2024/08/23

索引器允许类或结构的实例就像数组一样进行索引。 无需显式指定类型或实例成员，即可设置或检索索引值。 索引器类似于属性，不同之处在于它们的访问器需要使用参数。

以下示例定义了一个泛型类，其中包含用于赋值和检索值的简单 get 和 set 访问器方法。 `Program` 类创建了此类的一个实例，用于存储字符串。

C#

```
using System;

class SampleCollection<T>
{
   // Declare an array to store the data elements.
   private T[] arr = new T[100];

   // Define the indexer to allow client code to use [] notation.
   public T this[int i]
   {
      get { return arr[i]; }
      set { arr[i] = value; }
   }
}

class Program
{
   static void Main()
   {
      var stringCollection = new SampleCollection<string>();
      stringCollection[0] = "Hello, World";
      Console.WriteLine(stringCollection[0]);
   }
}
// The example displays the following output:
//       Hello, World.
```

① 备注

有关更多示例，请参阅**相关部分**。

表达式主体定义

索引器的 get 或 set 访问器包含一个用于返回或设置值的语句很常见。 为了支持这种情况，表达式主体成员提供了一种经过简化的语法。 自 C# 6 起，可以表达式主体成员的形式实现只读索引器，如以下示例所示。

C#

```
using System;

class SampleCollection<T>
{
   // Declare an array to store the data elements.
   private T[] arr = new T[100];
   int nextIndex = 0;

   // Define the indexer to allow client code to use [] notation.
   public T this[int i] => arr[i];

   public void Add(T value)
   {
      if (nextIndex >= arr.Length)
         throw new IndexOutOfRangeException($"The collection can hold only
{arr.Length} elements.");
      arr[nextIndex++] = value;
   }
}

class Program
{
   static void Main()
   {
      var stringCollection = new SampleCollection<string>();
      stringCollection.Add("Hello, World");
      System.Console.WriteLine(stringCollection[0]);
   }
}
// The example displays the following output:
//       Hello, World.
```

请注意，`=>` 引入了表达式主体，并未使用 `get` 关键字。

自 C# 7.0 起，get 和 set 访问器均可作为表达式主体成员实现。 在这种情况下，必须使用 `get` 和 `set` 关键字。 例如：

C#

```
using System;

class SampleCollection<T>
{
   // Declare an array to store the data elements.
   private T[] arr = new T[100];
```

```
   // Define the indexer to allow client code to use [] notation.
   public T this[int i]
   {
      get => arr[i];
      set => arr[i] = value;
   }
}

class Program
{
   static void Main()
   {
      var stringCollection = new SampleCollection<string>();
      stringCollection[0] = "Hello, World.";
      Console.WriteLine(stringCollection[0]);
   }
}
// The example displays the following output:
//       Hello, World.
```

索引器概述

- 使用索引器可以用类似于数组的方式为对象建立索引。
- `get` 取值函数返回值。 `set` 取值函数分配值。
- this 关键字用于定义索引器。
- value 关键字用于定义由 `set` 访问器分配的值。
- 索引器不必根据整数值进行索引；由你决定如何定义特定的查找机制。
- 索引器可被重载。
- 索引器可以有多个形参，例如当访问二维数组时。

相关章节

- 使用索引器
- 接口中的索引器
- 属性与索引器之间的比较
- 限制访问器可访问性

C# 语言规范

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

请参阅

- C# 编程指南
- 属性

反馈

此页面是否有帮助? 是 否

提供产品反馈

使用索引器（C# 编程指南）

项目 • 2024/08/20

索引器使你可从语法上方便地创建类、结构或接口，以便客户端应用程序可以像访问数组一样访问它们。 编译器会生成一个 `Item` 属性（或者如果存在 IndexerNameAttribute，也可以生成一个命名属性）和适当的访问器方法。 在主要目标是封装内部集合或数组的类型中，常常要实现索引器。 例如，假设有一个类 `TempRecord`，它表示 24 小时的周期内在 10 个不同时间点所记录的温度（单位为华氏度）。 此类包含一个 `float[]` 类型的数组 `temps`，用于存储温度值。 通过在此类中实现索引器，客户端可采用 `float temp = tempRecord[4]` 的形式（而非 `float temp = tempRecord.temps[4]`）访问 `TempRecord` 实例中的温度。 索引器表示法不但简化了客户端应用程序的语法；还使类及其目标更容易直观地为其它开发者所理解。

若要在类或结构上声明索引器，请使用 this 关键字，如以下示例所示：

```
C#

// Indexer declaration
public int this[int index]
{
    // get and set accessors
}
```

> **ⓘ 重要**
>
> 通过声明索引器，可自动在对象上生成一个名为 `Item` 的属性。 无法从实例**成员访问表达式**直接访问 `Item` 属性。 此外，如果通过索引器向对象添加自己的 `Item` 属性，则将收到 **CS0102 编译器错误**。 要避免此错误，请使用 IndexerNameAttribute 重命名本文后面详述的索引器。

注解

索引器及其参数的类型必须至少具有和索引器相同的可访问性。 有关可访问性级别的详细信息，请参阅访问修饰符。

有关如何在接口上使用索引器的详细信息，请参阅接口索引器。

索引器的签名由其形参的数目和类型所组成。 它不包含索引器类型或形参的名称。 如果要在相同类中声明多个索引器，则它们的签名必须不同。

索引器未分类为变量；因此，索引器值不能按引用（作为 ref 或 out 参数）传递，除非其值是引用（即按引用返回。）

若要使索引器的名称可为其他语言所用，请使用 System.Runtime.CompilerServices.IndexerNameAttribute，如以下示例所示：

C#

```
// Indexer declaration
[System.Runtime.CompilerServices.IndexerName("TheItem")]
public int this[int index]
{
    // get and set accessors
}
```

此索引器被索引器名称属性重写，因此其名称为 `TheItem`。 默认情况下，默认名称为 `Item`。

示例 1

下列示例演示如何声明专用数组字段 `temps` 和索引器。 索引器可以实现对实例 `tempRecord[i]` 的直接访问。 若不使用索引器，则将数组声明为公共成员，并直接访问其成员 `tempRecord.temps[i]`。

C#

```
public class TempRecord
{
    // Array of temperature values
    float[] temps =
    [
        56.2F, 56.7F, 56.5F, 56.9F, 58.8F,
        61.3F, 65.9F, 62.1F, 59.2F, 57.5F
    ];

    // To enable client code to validate input
    // when accessing your indexer.
    public int Length => temps.Length;

    // Indexer declaration.
    // If index is out of range, the temps array will throw the exception.
    public float this[int index]
    {
        get => temps[index];
        set => temps[index] = value;
    }
}
```

请注意，当评估索引器访问时（例如在 `Console.Write` 语句中），将调用 get 访问器。 因此，如果不存在 `get` 访问器，则会发生编译时错误。

C#

```
var tempRecord = new TempRecord();

// Use the indexer's set accessor
tempRecord[3] = 58.3F;
tempRecord[5] = 60.1F;

// Use the indexer's get accessor
for (int i = 0; i < 10; i++)
{
    Console.WriteLine($"Element #{i} = {tempRecord[i]}");
}
```

使用其他值进行索引

C# 不将索引参数类型限制为整数。 例如，对索引器使用字符串可能有用。 通过搜索集合内的字符串并返回相应的值，可以实现此类索引器。 访问器可被重载，因此字符串和整数版本可以共存。

示例 2

下面的示例声明了存储星期几的类。 `get` 访问器采用字符串（星期几）并返回对应的整数。 例如，“Sunday”返回 0，“Monday”返回 1，依此类推。

C#

```
// Using a string as an indexer value
class DayCollection
{
    string[] days = ["Sun", "Mon", "Tues", "Wed", "Thurs", "Fri", "Sat"];

    // Indexer with only a get accessor with the expression-bodied
definition:
    public int this[string day] => FindDayIndex(day);

    private int FindDayIndex(string day)
    {
        for (int j = 0; j < days.Length; j++)
        {
            if (days[j] == day)
            {
                return j;
            }
```

```
        }

        throw new ArgumentOutOfRangeException(
            nameof(day),
            $"Day {day} is not supported.\nDay input must be in the form
\"Sun\", \"Mon\", etc");
    }
}
```

使用示例 2

C#

```
var week = new DayCollection();
Console.WriteLine(week["Fri"]);

try
{
    Console.WriteLine(week["Made-up day"]);
}
catch (ArgumentOutOfRangeException e)
{
    Console.WriteLine($"Not supported input: {e.Message}");
}
```

示例 3

下面的示例声明了使用 System.DayOfWeek 存储星期几的类。 `get` 访问器采用 `DayOfWeek`（表示星期几的值）并返回对应的整数。 例如，`DayOfWeek.Sunday` 返回 0，`DayOfWeek.Monday` 返回 1，依此类推。

C#

```
using Day = System.DayOfWeek;

class DayOfWeekCollection
{
    Day[] days =
    [
        Day.Sunday, Day.Monday, Day.Tuesday, Day.Wednesday,
        Day.Thursday, Day.Friday, Day.Saturday
    ];

    // Indexer with only a get accessor with the expression-bodied
definition:
    public int this[Day day] => FindDayIndex(day);

    private int FindDayIndex(Day day)
```

```
    {
        for (int j = 0; j < days.Length; j++)
        {
            if (days[j] == day)
            {
                return j;
            }
        }
        throw new ArgumentOutOfRangeException(
            nameof(day),
            $"Day {day} is not supported.\nDay input must be a defined
System.DayOfWeek value.");
    }
}
```

使用示例 3

C#

```
var week = new DayOfWeekCollection();
Console.WriteLine(week[DayOfWeek.Friday]);

try
{
    Console.WriteLine(week[(DayOfWeek)43]);
}
catch (ArgumentOutOfRangeException e)
{
    Console.WriteLine($"Not supported input: {e.Message}");
}
```

可靠编程

提高索引器的安全性和可靠性有两种主要方法：

- 请确保结合某一类型的错误处理策略，以处理万一客户端代码传入无效索引值的情况。 在本文前面的第一个示例中，TempRecord 类提供了 Length 属性，使客户端代码能在将输入传递给索引器之前对其进行验证。 也可将错误处理代码放入索引器自身内部。 请确保为用户记录在索引器的访问器中引发的任何异常。

- 在可接受的程度内，为 get 和 set 访问器的可访问性设置尽可能多的限制。 这一点对 `set` 访问器尤为重要。 有关详细信息，请参阅限制访问器可访问性。

另请参阅

- 索引器

- 属性

接口中的索引器（C# 编程指南）

项目 • 2024/08/20

可以在接口上声明索引器。 接口索引器的访问器与类索引器的访问器有所不同，差异如下：

- 接口访问器不使用修饰符。
- 接口访问器通常没有正文。

访问器的用途是指示索引器为读写、只读还是只写。 可以为接口中定义的索引器提供实现，但这种情况非常少。 索引器通常定义 API 来访问数据字段，而数据字段无法在接口中定义。

下面是接口索引器访问器的示例：

C#

```
public interface ISomeInterface
{
    //...

    // Indexer declaration:
    string this[int index]
    {
        get;
        set;
    }
}
```

索引器的签名必须不同于同一接口中声明的所有其他索引器的签名。

示例

下面的示例演示如何实现接口索引器。

C#

```
// Indexer on an interface:
public interface IIndexInterface
{
    // Indexer declaration:
    int this[int index]
    {
        get;
        set;
    }
```

```
}

// Implementing the interface.
class IndexerClass : IIndexInterface
{
    private int[] arr = new int[100];
    public int this[int index]   // indexer declaration
    {
        // The arr object will throw IndexOutOfRange exception.
        get => arr[index];
        set => arr[index] = value;
    }
}
```

C#

```
IndexerClass test = new IndexerClass();
System.Random rand = System.Random.Shared;
// Call the indexer to initialize its elements.
for (int i = 0; i < 10; i++)
{
    test[i] = rand.Next();
}
for (int i = 0; i < 10; i++)
{
    System.Console.WriteLine($"Element #{i} = {test[i]}");
}

/* Sample output:
    Element #0 = 360877544
    Element #1 = 327058047
    Element #2 = 1913480832
    Element #3 = 1519039937
    Element #4 = 601472233
    Element #5 = 323352310
    Element #6 = 1422639981
    Element #7 = 1797892494
    Element #8 = 875761049
    Element #9 = 393083859
*/
```

在前面的示例中，可通过使用接口成员的完全限定名来使用显示接口成员实现。 例如

C#

```
string IIndexInterface.this[int index]
{
}
```

但仅当类采用相同的索引签名实现多个接口时，才需用到完全限定名称以避免歧义。 例如，如果 `Employee` 类正在实现接口 `ICitizen` 和接口 `IEmployee`，而这两个接口具有相同的索引签名，则需要用到显式接口成员实现。 即是说以下索引器声明：

C#

```
string IEmployee.this[int index]
{
}
```

在 `IEmployee` 接口中实现索引器，而以下声明：

C#

```
string ICitizen.this[int index]
{
}
```

在 `ICitizen` 接口中实现索引器。

另请参阅

- 索引器
- 属性
- 接口

属性和索引器之间的比较（C# 编程指南）

项目 • 2024/10/26

索引器与属性相似。 除下表所示的差别外，对属性访问器定义的所有规则也适用于索引器访问器。

展开表

Property	索引器
允许以将方法视作公共数据成员的方式调用方法。	通过在对象自身上使用数组表示法，允许访问对象内部集合的元素。
通过简单名称访问。	通过索引访问。
可为静态成员或实例成员。	必须是实例成员。
属性的 get 访问器没有任何参数。	索引器的 `get` 访问器具有与索引器相同的形参列表。
属性的 set 访问器包含隐式 `value` 参数。	索引器的 `set` 访问器具有与索引器相同的形参列表，value 参数也是如此。
支持使用 自动实现的属性缩短语法。	支持仅使用索引器的 expression-bodied 成员。

另请参阅

- 索引器
- 属性

事件（C# 编程指南）

项目 • 2024/04/11

类 或对象可以通过事件向其他类或对象通知发生的相关事情。 发送（或引发）事件的类称为“发布者”，接收（或处理）事件的类称为“订阅者”。

在典型的 C# Windows 窗体或 Web 应用程序中，可订阅由按钮和列表框等控件引发的事件。 可以使用 Visual C# 集成开发环境 (IDE) 来浏览控件发布的事件，并选择想要处理的事件。 借助 IDE，可轻松自动添加空白事件处理程序方法以及要订阅该事件的代码。 有关详细信息，请参阅如何订阅和取消订阅事件。

事件概述

事件具有以下属性：

- 发行者确定何时引发事件；订户确定对事件作出何种响应。
- 一个事件可以有多个订户。 订户可以处理来自多个发行者的多个事件。
- 没有订户的事件永远也不会引发。
- 事件通常用于表示用户操作，例如单击按钮或图形用户界面中的菜单选项。
- 当事件具有多个订户时，引发该事件时会同步调用事件处理程序。 若要异步调用事件，请参阅 “使用异步方式调用同步方法”。
- 在 .NET 类库中，事件基于 EventHandler 委托和 EventArgs 基类。

相关章节

有关详细信息，请参见:

- 如何订阅和取消订阅事件
- 如何发布符合 .NET 准则的事件
- 如何在派生类中引发基类事件
- 如何实现接口事件
- 如何实现自定义事件访问器

C# 语言规范

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

重要章节

C# 3.0 手册（第三版）：面向 C# 3.0 程序员的超过 250 个解决方案中的委托、事件和 Lambda 表达式

学习 C# 3.0：C# 3.0 基础知识中的委托和事件

另请参阅

- EventHandler
- C# 编程指南
- 委托
- 在 Windows 窗体中创建事件处理程序

反馈

此页面是否有帮助? 是 否

提供产品反馈

如何订阅和取消订阅事件（C# 编程指南）

项目 • 2023/03/10

如果想编写引发事件时调用的自定义代码，则可以订阅由其他类发布的事件。 例如，可以订阅某个按钮的 `click` 事件，以使应用程序在用户单击该按钮时执行一些有用的操作。

使用 Visual Studio IDE 订阅事件

1. 如果看不到“属性”窗口，请在“设计”视图中，右键单击要为其创建事件处理程序的窗体或控件，然后选择“属性” 。

2. 在“属性”窗口的顶部，单击“事件”图标 。

3. 双击要创建的事件，例如 `Load` 事件。

   Visual C# 会创建一个空事件处理程序方法，并将其添加到你的代码中。 或者，也可以在“代码”视图中手动添加代码。 例如，下面的代码行声明了一个在 `Form` 类引发 `Load` 事件时调用的事件处理程序方法。

   ```
   C#

   private void Form1_Load(object sender, System.EventArgs e)
   {
       // Add your form load event handling code here.
   }
   ```

   还会在项目的 Form1.Designer.cs 文件的 `InitializeComponent` 方法中自动生成订阅该事件所需的代码行。 该代码行类似于：

   ```
   C#

   this.Load += new System.EventHandler(this.Form1_Load);
   ```

以编程方式订阅事件

1. 定义一个事件处理程序方法，其签名与该事件的委托签名匹配。 例如，如果事件基于 EventHandler 委托类型，则下面的代码表示方法存根：

   ```
   C#
   ```

```
void HandleCustomEvent(object sender, CustomEventArgs a)
{
   // Do something useful here.
}
```

2. 使用加法赋值运算符 (`+=`) 来为事件附加事件处理程序。 在下面的示例中，假设名为 `publisher` 的对象拥有一个名为 `RaiseCustomEvent` 的事件。 请注意，订户类需要引用发行者类才能订阅其事件。

```
C#

publisher.RaiseCustomEvent += HandleCustomEvent;
```

还可以使用 Lambda 表达式来指定事件处理程序：

```
C#

public Form1()
{
    InitializeComponent();
    this.Click += (s,e) =>
        {
            MessageBox.Show(((MouseEventArgs)e).Location.ToString());
        };
}
```

使用匿名函数订阅事件

如果以后不必取消订阅某个事件，则可以使用加法赋值运算符 (`+=`) 将匿名函数作为事件处理程序进行附加。 在下面的示例中，假设名为 `publisher` 的对象拥有一个名为 `RaiseCustomEvent` 的事件，并且还定义了一个 `CustomEventArgs` 类以承载某些类型的专用事件信息。 请注意，订户类需要引用 `publisher` 才能订阅其事件。

```
C#

publisher.RaiseCustomEvent += (object o, CustomEventArgs e) =>
{
  string s = o.ToString() + " " + e.ToString();
  Console.WriteLine(s);
};
```

如果使用匿名函数订阅事件，事件的取消订阅过程将比较麻烦。 这种情况下若要取消订阅，请返回到该事件的订阅代码，将该匿名函数存储在委托变量中，然后将此委托添加到

该事件中。 如果必须在后面的代码中取消订阅某个事件，则建议不要使用匿名函数订阅此事件。 有关匿名函数的详细信息，请参阅 Lambda 表达式。

取消订阅

若要防止在引发事件时调用事件处理程序，请取消订阅该事件。 为了防止资源泄露，应在释放订户对象之前取消订阅事件。 在取消订阅事件之前，在发布对象中作为该事件的基础的多播委托会引用封装了订户的事件处理程序的委托。 只要发布对象保持该引用，垃圾回收功能就不会删除订户对象。

取消订阅事件

- 使用减法赋值运算符 (`-=`) 取消订阅事件：

  ```
  C#

  publisher.RaiseCustomEvent -= HandleCustomEvent;
  ```

  所有订户都取消订阅事件后，发行者类中的事件实例将设置为 `null`。

请参阅

- 事件
- event
- 如何发布符合 .NET 准则的事件
- - 和 -= 运算符
- + 和 += 运算符

如何在派生类中引发基类事件（C# 编程指南）

项目 • 2024/03/13

下面的简单示例演示用于在基类中声明事件，以便也可以从派生类引发它们的标准方法。 此模式广泛用于 .NET 类库中的 Windows 窗体类。

创建可以用作其他类的基类的类时，应考虑到以下事实：事件是特殊类型的委托，只能从声明它们的类中进行调用。 派生类不能直接调用在基类中声明的事件。 虽然有时可能需要只能由基类引发的事件，不过在大多数情况下，应使派生类可以调用基类事件。 为此，可以在包装事件的基类中创建受保护的调用方法。 通过调用或重写此调用方法，派生类可以间接调用事件。

> **备注**
>
> 不要在基类中声明虚拟事件并在派生类中重写它们。 C# 编译器不会处理这些事件，并且无法预知派生事件的订阅者是否实际上会订阅基类事件。

示例

C#

```
namespace BaseClassEvents
{
    // Special EventArgs class to hold info about Shapes.
    public class ShapeEventArgs : EventArgs
    {
        public ShapeEventArgs(double area)
        {
            NewArea = area;
        }

        public double NewArea { get; }
    }

    // Base class event publisher
    public abstract class Shape
    {
        protected double _area;

        public double Area
        {
            get => _area;
```

```
            set => _area = value;
        }

        // The event. Note that by using the generic EventHandler<T> event
type
        // we do not need to declare a separate delegate type.
        public event EventHandler<ShapeEventArgs> ShapeChanged;

        public abstract void Draw();

        //The event-invoking method that derived classes can override.
        protected virtual void OnShapeChanged(ShapeEventArgs e)
        {
            // Safely raise the event for all subscribers
            ShapeChanged?.Invoke(this, e);
        }
    }

    public class Circle : Shape
    {
        private double _radius;

        public Circle(double radius)
        {
            _radius = radius;
            _area = 3.14 * _radius * _radius;
        }

        public void Update(double d)
        {
            _radius = d;
            _area = 3.14 * _radius * _radius;
            OnShapeChanged(new ShapeEventArgs(_area));
        }

        protected override void OnShapeChanged(ShapeEventArgs e)
        {
            // Do any circle-specific processing here.

            // Call the base class event invocation method.
            base.OnShapeChanged(e);
        }

        public override void Draw()
        {
            Console.WriteLine("Drawing a circle");
        }
    }

    public class Rectangle : Shape
    {
        private double _length;
        private double _width;

        public Rectangle(double length, double width)
```

```
        {
            _length = length;
            _width = width;
            _area = _length * _width;
        }

        public void Update(double length, double width)
        {
            _length = length;
            _width = width;
            _area = _length * _width;
            OnShapeChanged(new ShapeEventArgs(_area));
        }

        protected override void OnShapeChanged(ShapeEventArgs e)
        {
            // Do any rectangle-specific processing here.

            // Call the base class event invocation method.
            base.OnShapeChanged(e);
        }

        public override void Draw()
        {
            Console.WriteLine("Drawing a rectangle");
        }
    }

    // Represents the surface on which the shapes are drawn
    // Subscribes to shape events so that it knows
    // when to redraw a shape.
    public class ShapeContainer
    {
        private readonly List<Shape> _list;

        public ShapeContainer()
        {
            _list = new List<Shape>();
        }

        public void AddShape(Shape shape)
        {
            _list.Add(shape);

            // Subscribe to the base class event.
            shape.ShapeChanged += HandleShapeChanged;
        }

        // ...Other methods to draw, resize, etc.

        private void HandleShapeChanged(object sender, ShapeEventArgs e)
        {
            if (sender is Shape shape)
            {
                // Diagnostic message for demonstration purposes.
```

```
                Console.WriteLine($"Received event. Shape area is now
{e.NewArea}");

                // Redraw the shape here.
                shape.Draw();
            }
        }
    }

    class Test
    {
        static void Main()
        {
            //Create the event publishers and subscriber
            var circle = new Circle(54);
            var rectangle = new Rectangle(12, 9);
            var container = new ShapeContainer();

            // Add the shapes to the container.
            container.AddShape(circle);
            container.AddShape(rectangle);

            // Cause some events to be raised.
            circle.Update(57);
            rectangle.Update(7, 7);

            // Keep the console window open in debug mode.
            Console.WriteLine("Press any key to continue...");
            Console.ReadKey();
        }
    }
}
/* Output:
        Received event. Shape area is now 10201.86
        Drawing a circle
        Received event. Shape area is now 49
        Drawing a rectangle
 */
```

另请参阅

- 事件
- 委托
- 访问修饰符
- 在 Windows 窗体中创建事件处理程序

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

如何实现接口事件（C# 编程指南）

项目 • 2023/03/10

接口可以声明事件。 下面的示例演示如何在类中实现接口事件。 这些规则基本上都与实现任何接口方法或属性时的相同。

在类中实现接口事件

在类中声明事件，然后在相应区域中调用它。

C#

```
namespace ImplementInterfaceEvents
{
    public interface IDrawingObject
    {
        event EventHandler ShapeChanged;
    }
    public class MyEventArgs : EventArgs
    {
        // class members
    }
    public class Shape : IDrawingObject
    {
        public event EventHandler ShapeChanged;
        void ChangeShape()
        {
            // Do something here before the event…

            OnShapeChanged(new MyEventArgs(/*arguments*/));

            // or do something here after the event.
        }
        protected virtual void OnShapeChanged(MyEventArgs e)
        {
            ShapeChanged?.Invoke(this, e);
        }
    }

}
```

示例

下面的示例演示如何处理不太常见的情况：类继承自两个或多个接口，且每个接口都具有相同名称的事件。 在这种情况下，你必须为至少其中一个事件提供显式接口实现。 为事

件编写显式接口实现时，还必须编写 `add` 和 `remove` 事件访问器。 通常这些访问器由编译器提供，但在这种情况下编译器不提供它们。

通过提供自己的访问器，可以指定两个事件是由类中的同一个事件表示，还是由不同事件表示。 例如，如果根据接口规范应在不同时间引发事件，可以在类中将每个事件与单独实现关联。 在下面的示例中，订阅服务器确定它们通过将形状引用转换为 `IShape` 或 `IDrawingObject` 接收哪个 `OnDraw` 事件。

C#

```
namespace WrapTwoInterfaceEvents
{
    using System;

    public interface IDrawingObject
    {
        // Raise this event before drawing
        // the object.
        event EventHandler OnDraw;
    }
    public interface IShape
    {
        // Raise this event after drawing
        // the shape.
        event EventHandler OnDraw;
    }

    // Base class event publisher inherits two
    // interfaces, each with an OnDraw event
    public class Shape : IDrawingObject, IShape
    {
        // Create an event for each interface event
        event EventHandler PreDrawEvent;
        event EventHandler PostDrawEvent;

        object objectLock = new Object();

        // Explicit interface implementation required.
        // Associate IDrawingObject's event with
        // PreDrawEvent
        #region IDrawingObjectOnDraw
        event EventHandler IDrawingObject.OnDraw
        {
            add
            {
                lock (objectLock)
                {
                    PreDrawEvent += value;
                }
            }
            remove
            {
```

```
                lock (objectLock)
                {
                    PreDrawEvent -= value;
                }
            }
        }
        #endregion
        // Explicit interface implementation required.
        // Associate IShape's event with
        // PostDrawEvent
        event EventHandler IShape.OnDraw
        {
            add
            {
                lock (objectLock)
                {
                    PostDrawEvent += value;
                }
            }
            remove
            {
                lock (objectLock)
                {
                    PostDrawEvent -= value;
                }
            }
        }

        // For the sake of simplicity this one method
        // implements both interfaces.
        public void Draw()
        {
            // Raise IDrawingObject's event before the object is drawn.
            PreDrawEvent?.Invoke(this, EventArgs.Empty);

            Console.WriteLine("Drawing a shape.");

            // Raise IShape's event after the object is drawn.
            PostDrawEvent?.Invoke(this, EventArgs.Empty);
        }
    }
    public class Subscriber1
    {
        // References the shape object as an IDrawingObject
        public Subscriber1(Shape shape)
        {
            IDrawingObject d = (IDrawingObject)shape;
            d.OnDraw += d_OnDraw;
        }

        void d_OnDraw(object sender, EventArgs e)
        {
            Console.WriteLine("Sub1 receives the IDrawingObject event.");
        }
    }
```

```
    // References the shape object as an IShape
    public class Subscriber2
    {
        public Subscriber2(Shape shape)
        {
            IShape d = (IShape)shape;
            d.OnDraw += d_OnDraw;
        }

        void d_OnDraw(object sender, EventArgs e)
        {
            Console.WriteLine("Sub2 receives the IShape event.");
        }
    }

    public class Program
    {
        static void Main(string[] args)
        {
            Shape shape = new Shape();
            Subscriber1 sub = new Subscriber1(shape);
            Subscriber2 sub2 = new Subscriber2(shape);
            shape.Draw();

            // Keep the console window open in debug mode.
            System.Console.WriteLine("Press any key to exit.");
            System.Console.ReadKey();
        }
    }
}
/* Output:
    Sub1 receives the IDrawingObject event.
    Drawing a shape.
    Sub2 receives the IShape event.
*/
```

另请参阅

- C# 编程指南
- 事件
- 委托
- 显式接口实现
- 如何在派生类中引发基类事件

如何实现自定义事件访问器（C# 编程指南）

项目 • 2023/03/10

事件是一种特殊的多播委托，只能从声明它的类中进行调用。 客户端代码通过提供对应在引发事件时调用的方法的引用来订阅事件。 这些方法通过事件访问器添加到委托的调用列表中，事件访问器类似于属性访问器，不同之处在于事件访问器命名为 `add` 和 `remove`。 在大多数情况下，无需提供自定义事件访问器。 如果代码中没有提供自定义事件访问器，编译器将自动添加它们。 但在某些情况下，可能需要提供自定义行为。 主题如何实现接口事件中介绍了这样一种情况。

示例

下面的示例演示如何实现自定义的 add 和 remove 事件访问器。 虽然可以替换访问器内的任何代码，但建议先锁定事件，再添加或删除新的事件处理程序方法。

C#

```
event EventHandler IDrawingObject.OnDraw
{
    add
    {
        lock (objectLock)
        {
            PreDrawEvent += value;
        }
    }
    remove
    {
        lock (objectLock)
        {
            PreDrawEvent -= value;
        }
    }
}
```

请参阅

- 事件
- event

泛型类型参数 - （C# 编程指南）

项目 • 2024/03/13

在泛型类型或方法定义中，类型参数是在其创建泛型类型的一个实例时，客户端指定的特定类型的占位符。 泛型类（例如泛型介绍中列出的 `GenericList<T>`）无法按原样使用，因为它不是真正的类型；它更像是类型的蓝图。 若要使用 `GenericList<T>`，客户端代码必须通过指定尖括号内的类型参数来声明并实例化构造类型。 此特定类的类型参数可以是编译器可识别的任何类型。 可创建任意数量的构造类型实例，其中每个使用不同的类型参数，如下所示：

C#

```
GenericList<float> list1 = new GenericList<float>();
GenericList<ExampleClass> list2 = new GenericList<ExampleClass>();
GenericList<ExampleStruct> list3 = new GenericList<ExampleStruct>();
```

在 `GenericList<T>` 的每个实例中，类中出现的每个 `T` 在运行时均会被替换为类型参数。 通过这种替换，我们已通过使用单个类定义创建了三个单独的类型安全的有效对象。 有关 CLR 如何执行此替换的详细信息，请参阅运行时中的泛型。

可在有关命名约定的文章中了解泛型类型参数的命名约定。

另请参阅

- System.Collections.Generic
- 泛型
- C++ 模板和 C# 泛型之间的区别

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

类型参数的约束（C# 编程指南）

项目 • 2024/08/01

约束告知编译器类型参数必须具备的功能。 在没有任何约束的情况下，类型参数可以是任何类型。 编译器只能假定 System.Object 的成员，它是任何 .NET 类型的最终基类。 有关详细信息，请参阅使用约束的原因。 如果客户端代码使用不满足约束的类型，编译器将发出错误。 通过使用 `where` 上下文关键字指定约束。 下表列出了各种类型的约束：

展开表

约束	说明
`where T : struct`	类型参数必须是不可为 null 的值类型，其中包含 `record struct` 类型。 有关可为 null 的值类型的信息，请参阅可为 null 的值类型。 由于所有值类型都具有可访问的无参数构造函数（无论是声明的还是隐式的），因此 `struct` 约束表示 `new()` 约束，并且不能与 `new()` 约束结合使用。 `struct` 约束也不能与 `unmanaged` 约束结合使用。
`where T : class`	类型参数必须是引用类型。 此约束还应用于任何类、接口、委托或数组类型。 在可为 null 的上下文中，`T` 必须是不可为 null 的引用类型。
`where T : class?`	类型参数必须是可为 null 或不可为 null 的引用类型。 此约束还应用于任何类、接口、委托或数组类型（包括记录）。
`where T : notnull`	类型参数必须是不可为 null 的类型。 参数可以是不可为 null 的引用类型，也可以是不可为 null 的值类型。
`where T : unmanaged`	类型参数必须是不可为 null 的非托管类型。 `unmanaged` 约束表示 `struct` 约束，且不能与 `struct` 约束或 `new()` 约束结合使用。
`where T : new()`	类型参数必须具有公共无参数构造函数。 与其他约束一起使用时，`new()` 约束必须最后指定。 `new()` 约束不能与 `struct` 和 `unmanaged` 约束结合使用。
`where T :` *基类名*	类型参数必须是指定的基类或派生自指定的基类。 在可为 null 的上下文中，`T` 必须是从指定基类派生的不可为 null 的引用类型。
`where T :` *基类名*`?`	类型参数必须是指定的基类或派生自指定的基类。 在可为 null 的上下文中，`T` 可以是从指定基类派生的可为 null 或不可为 null 的类型。
`where T :` *接口名*	类型参数必须是指定的接口或实现指定的接口。 可指定多个接口约束。 约束接口也可以是泛型。 在的可为 null 的上下文中，`T` 必须是实现指定接口的不可为 null 的类型。
`where T :` *接口名*`?`	类型参数必须是指定的接口或实现指定的接口。 可指定多个接口约束。 约束接口也可以是泛型。 在可为 null 的上下文中，`T` 可以是可为 null 的引用类型、不可为 null 的引用类型或值类型。 `T` 不能是可为 null 的值类型。

约束	说明
`where T : U`	为 `T` 提供的类型参数必须是为 `U` 提供的参数或派生自为 `U` 提供的参数。 在可为 null 的上下文中，如果 `U` 是不可为 null 的引用类型，`T` 必须是不可为 null 的引用类型。 如果 `U` 是可为 null 的引用类型，则 `T` 可以是可为 null 的引用类型，也可以是不可为 null 的引用类型。
`where T : default`	重写方法或提供显式接口实现时，如果需要指定不受约束的类型参数，此约束可解决歧义。 `default` 约束表示基方法，但不包含 `class` 或 `struct` 约束。 有关详细信息，请参阅default约束规范建议。
`where T : allows ref struct`	此反约束声明 `T` 的类型参数可以是 `ref struct` 类型。 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`。

某些约束是互斥的，而某些约束必须按指定顺序排列：

- 最多可应用 `struct`、`class`、`class?`、`notnull` 和 `unmanaged` 约束中的一个。 如果提供这些约束中的任何一个，则它必须是为该类型参数指定的第一个约束。
- 基类约束（`where T : Base` 或 `where T : Base?`）不能与 `struct`、`class`、`class?`、`notnull` 或 `unmanaged` 约束中的任何一个结合使用。
- 无论哪种形式，都最多只能应用一个基类约束。 如果想要支持可为 null 的基类型，请使用 `Base?`。
- 不能将接口不可为 null 和可为 null 的形式命名为约束。
- `new()` 约束不能与 `struct` 或 `unmanaged` 约束结合使用。 如果指定 `new()` 约束，则它必须是该类型参数的最后一个约束。 反约束（如果适用）可以遵循 `new()` 约束。
- `default` 约束只能应用于替代或显式接口实现。 它不能与 `struct` 或 `class` 约束结合使用。
- `allows ref struct` 反约束不能与 `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。

使用约束的原因

约束指定类型参数的功能和预期。 声明这些约束意味着你可以使用约束类型的操作和方法调用。 如果泛型类或方法对泛型成员使用除简单赋值之外的任何操作，包括调用 System.Object 不支持的任何方法，则对类型参数应用约束。 例如，基类约束告诉编译器，只有此类型的对象或派生自此类型的对象可替换该类型参数。 编译器有了此保证后，就能够允许在泛型类中调用该类型的方法。 以下代码示例演示可通过应用基类约束添加到（泛型介绍中的）`GenericList<T>` 类的功能。

```
C#
```

```
public class Employee
{
    public Employee(string name, int id) => (Name, ID) = (name, id);
    public string Name { get; set; }
    public int ID { get; set; }
}

public class GenericList<T> where T : Employee
{
    private class Node
    {
        public Node(T t) => (Next, Data) = (null, t);

        public Node? Next { get; set; }
        public T Data { get; set; }
    }

    private Node? head;

    public void AddHead(T t)
    {
        Node n = new Node(t) { Next = head };
        head = n;
    }

    public IEnumerator<T> GetEnumerator()
    {
        Node? current = head;

        while (current != null)
        {
            yield return current.Data;
            current = current.Next;
        }
    }

    public T? FindFirstOccurrence(string s)
    {
        Node? current = head;
        T? t = null;

        while (current != null)
        {
            //The constraint enables access to the Name property.
            if (current.Data.Name == s)
            {
                t = current.Data;
                break;
            }
            else
            {
                current = current.Next;
            }
        }
```

```
        return t;
    }
}
```

约束使泛型类能够使用 `Employee.Name` 属性。 约束指定类型 `T` 的所有项都保证是 `Employee` 对象或从 `Employee` 继承的对象。

可以对同一类型参数应用多个约束，并且约束自身可以是泛型类型，如下所示：

C#

```
class EmployeeList<T> where T : notnull, Employee, IComparable<T>, new()
{
    // ...
    public void AddDefault()
    {
        T t = new T();
        // ...
    }
}
```

在应用 `where T : class` 约束时，请避免对类型参数使用 `==` 和 `!=` 运算符，因为这些运算符仅测试引用标识而不测试值相等性。 即使在用作参数的类型中重载这些运算符也会发生此行为。 下面的代码说明了这一点；即使 String 类重载 `==` 运算符，输出也为 false。

C#

```
public static void OpEqualsTest<T>(T s, T t) where T : class
{
    System.Console.WriteLine(s == t);
}

private static void TestStringEquality()
{
    string s1 = "target";
    System.Text.StringBuilder sb = new System.Text.StringBuilder("target");
    string s2 = sb.ToString();
    OpEqualsTest<string>(s1, s2);
}
```

编译器只知道 `T` 在编译时是引用类型，并且必须使用对所有引用类型都有效的默认运算符。 如果必须测试值相等性，请应用 `where T : IEquatable<T>` 或 `where T : IComparable<T>` 约束，并在用于构造泛型类的任何类中实现该接口。

约束多个参数

可以对多个参数应用多个约束，对一个参数应用多个约束，如下例所示：

C#

```
class Base { }
class Test<T, U>
    where U : struct
    where T : Base, new()
{ }
```

未绑定的类型参数

没有约束的类型参数（如公共类 `SampleClass<T>{}` 中的 T）称为未绑定的类型参数。 未绑定的类型参数具有以下规则：

- 不能使用 `!=` 和 `==` 运算符，因为无法保证具体的类型参数能支持这些运算符。
- 可以在它们与 `System.Object` 之间来回转换，或将它们显式转换为任何接口类型。
- 可以将它们与 null 进行比较。 将未绑定的参数与 `null` 进行比较时，如果类型参数为值类型，则该比较始终返回 false。

类型参数作为约束

在具有自己类型参数的成员函数必须将该参数约束为包含类型的类型参数时，将泛型类型参数用作约束非常有用，如下例所示：

C#

```
public class List<T>
{
    public void Add<U>(List<U> items) where U : T {/*...*/}
}
```

在上述示例中，`T` 在 `Add` 方法的上下文中是一个类型约束，而在 `List` 类的上下文中是一个未绑定的类型参数。

类型参数还可在泛型类定义中用作约束。 必须在尖括号中声明该类型参数以及任何其他类型参数：

C#

```
//Type parameter V is used as a type constraint.
public class SampleClass<T, U, V> where T : V { }
```

类型参数作为泛型类的约束的作用非常有限，因为编译器除了假设类型参数派生自 `System.Object` 以外，不会做其他任何假设。 如果要在两个类型参数之间强制继承关系，可以将类型参数用作泛型类的约束。

`notnull` 约束

可以使用 `notnull` 约束指定类型参数必须是不可为 null 的值类型或不可为 null 的引用类型。 与大多数其他约束不同，如果类型参数违反 `notnull` 约束，编译器会生成警告而不是错误。

`notnull` 约束仅在可为 null 上下文中使用时才有效。 如果在过时的可为 null 上下文中添加 `notnull` 约束，编译器不会针对违反约束的情况生成任何警告或错误。

`class` 约束

可为 null 的上下文中的 `class` 约束指定类型参数必须是不可为 null 的引用类型。 在可为 null 上下文中，当类型参数是可为 null 的引用类型时，编译器会生成警告。

`default` 约束

添加可为空引用类型会使泛型类型或方法中的 `T?` 使用复杂化。 `T?` 可以与 `struct` 或 `class` 约束一起使用，但必须存在其中一项。 使用 `class` 约束时，`T?` 引用了 `T` 的可为空引用类型。 可在这两个约束均未应用时使用 `T?`。 在这种情况下，对于值类型和引用类型，`T?` 解读为 `T?`。 但是，如果 `T` 是 Nullable<T>的实例，则 `T?` 与 `T` 相同。 换句话说，它不会成为 `T??`。

由于现在可在没有 `class` 或 `struct` 约束的情况下使用 `T?`，因此在重写或显式接口实现中可能会出现歧义。 在这两种情况下，重写不包含约束，但从基类继承。 当基类不应用 `class` 或 `struct` 约束时，派生类需要通过某种方式在不使用任一种约束的情况下指定应用于基方法的重写。 派生方法应用 `default` 约束。 `default` 约束不阐明 `class` 和 `struct` 约束。

非托管约束

可使用 `unmanaged` 约束来指定类型参数必须是不可为 null 的非托管类型。 通过 `unmanaged` 约束，用户能编写可重用例程，从而使用可作为内存块操作的类型，如以下示例所示：

```
C#
```

```
unsafe public static byte[] ToByteArray<T>(this T argument) where T :
unmanaged
{
    var size = sizeof(T);
    var result = new Byte[size];
    Byte* p = (byte*)&argument;
    for (var i = 0; i < size; i++)
        result[i] = *p++;
    return result;
}
```

以上方法必须在 `unsafe` 上下文中编译，因为它并不是在已知的内置类型上使用 `sizeof` 运算符。 如果没有 `unmanaged` 约束，则 `sizeof` 运算符不可用。

`unmanaged` 约束表示 `struct` 约束，且不能与其结合使用。 因为 `struct` 约束表示 `new()` 约束，且 `unmanaged` 约束也不能与 `new()` 约束结合使用。

委托约束

可以使用 System.Delegate 或 System.MulticastDelegate 作为基类约束。 CLR 始终允许此约束，但 C# 语言不允许。 使用 `System.Delegate` 约束，用户能够以类型安全的方式编写使用委托的代码。 以下代码定义了合并两个同类型委托的扩展方法：

C#

```
public static TDelegate? TypeSafeCombine<TDelegate>(this TDelegate source,
TDelegate target)
    where TDelegate : System.Delegate
    => Delegate.Combine(source, target) as TDelegate;
```

可使用上述方法来合并相同类型的委托：

C#

```
Action first = () => Console.WriteLine("this");
Action second = () => Console.WriteLine("that");

var combined = first.TypeSafeCombine(second);
combined!();

Func<bool> test = () => true;
// Combine signature ensures combined delegates must
// have the same type.
//var badCombined = first.TypeSafeCombine(test);
```

如果对最后一行取消注释，它将不会编译。 `first` 和 `test` 均为委托类型，但它们是不同的委托类型。

枚举约束

还可指定 System.Enum 类型作为基类约束。 CLR 始终允许此约束，但 C# 语言不允许。 使用 `System.Enum` 的泛型提供类型安全的编程，缓存使用 `System.Enum` 中静态方法的结果。 以下示例查找枚举类型的所有有效的值，然后生成将这些值映射到其字符串表示形式的字典。

C#

```
public static Dictionary<int, string> EnumNamedValues<T>() where T :
System.Enum
{
    var result = new Dictionary<int, string>();
    var values = Enum.GetValues(typeof(T));

    foreach (int item in values)
        result.Add(item, Enum.GetName(typeof(T), item)!);
    return result;
}
```

`Enum.GetValues` 和 `Enum.GetName` 使用反射，这会对性能产生影响。 可调用 `EnumNamedValues` 来生成可缓存和重用的集合，而不是重复执行需要反射才能实施的调用。

如以下示例所示，可使用它来创建枚举并生成其值和名称的字典：

C#

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

C#

```
var map = EnumNamedValues<Rainbow>();
```

```
foreach (var pair in map)
    Console.WriteLine($"{pair.Key}:\t{pair.Value}");
```

类型参数实现声明的接口

某些场景要求为类型参数提供的参数实现该接口。 例如：

C#

```
public interface IAdditionSubtraction<T> where T : IAdditionSubtraction<T>
{
    static abstract T operator +(T left, T right);
    static abstract T operator -(T left, T right);
}
```

此模式使 C# 编译器能够确定重载运算符或任何 `static virtual` 或 `static abstract` 方法的包含类型。 它提供的语法使得可以在包含类型上定义加法和减法运算符。 如果没有此约束，需要将参数和自变量声明为接口，而不是类型参数：

C#

```
public interface IAdditionSubtraction<T> where T : IAdditionSubtraction<T>
{
    static abstract IAdditionSubtraction<T> operator +(
        IAdditionSubtraction<T> left,
        IAdditionSubtraction<T> right);

    static abstract IAdditionSubtraction<T> operator -(
        IAdditionSubtraction<T> left,
        IAdditionSubtraction<T> right);
}
```

上述语法要求实现者对这些方法使用显式接口实现。 提供额外的约束使接口能够根据类型参数来定义运算符。 实现接口的类型可以隐式实现接口方法。

Allows ref struct

`allows ref struct` 反约束声明相应的类型参数可以是 ref struct 类型。 该类型参数的实例必须遵循以下规则：

- 它不能被装箱。
- 它参与引用安全规则。
- 不能在不允许 `ref struct` 类型的地方使用实例，例如 `static` 字段。
- 实例可以使用 `scoped` 修饰符进行标记。

不会继承 `allows ref struct` 子句。 在以下代码中：

```C#
class SomeClass<T, S>
    where T : allows ref struct
    where S : T
{
    // etc
}
```

`S` 的参数不能是 `ref struct`，因为 `S` 没有 `allows ref struct` 子句。

具有 `allows ref struct` 子句的类型参数不能用作类型参数，除非相应的类型参数也具有 `allows ref struct` 子句。 下面的示例说明了此规则：

```C#
public class Allow<T> where T : allows ref struct
{

}

public class Disallow<T>
{
}

public class Example<T> where T : allows ref struct
{
    private Allow<T> fieldOne; // Allowed. T is allowed to be a ref struct

    private Disallow<T> fieldTwo; // Error. T is not allowed to be a ref
struct
}
```

前面的示例表明，对于一个可能是 `ref struct` 类型的类型参数，不能将其替换为不能是 `ref struct` 类型的类型参数。

另请参阅

- System.Collections.Generic
- 泛型介绍
- 泛型类
- new 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

泛型类（C# 编程指南）

项目 • 2024/10/31

泛型类封装不特定于特定数据类型的操作。 泛型类最常见用法是用于链接列表、哈希表、堆栈、队列和树等集合。 无论存储数据的类型如何，添加项和从集合删除项等操作的执行方式基本相同。

对于大多数需要集合类的方案，推荐做法是使用 .NET 类库中提供的集合类。 有关使用这些类的详细信息，请参阅 .NET 中的泛型集合。

通常，创建泛型类是从现有具体类开始，然后每次逐个将类型更改为类型参数，直到泛化和可用性达到最佳平衡。 创建自己的泛型类时，需要考虑以下重要注意事项：

- 要将哪些类型泛化为类型参数。

  通常，可参数化的类型越多，代码就越灵活、其可重用性就越高。 但过度泛化会造成其他开发人员难以阅读或理解代码。

- 要将何种约束（如有）应用到类型参数（请参阅类型参数的约束）。

  其中一个有用的规则是，应用最大程度的约束，同时仍可处理必须处理的类型。 例如，如果知道泛型类仅用于引用类型，则请应用类约束。 这可防止将类意外用于值类型，并使你可在 `T` 上使用 `as` 运算符和检查 null 值。

- 是否将泛型行为分解为基类和子类。

  因为泛型类可用作基类，所以非泛型类的相同设计注意事项在此也适用。 请参阅本主题后文有关从泛型基类继承的规则。

- 实现一个泛型接口还是多个泛型接口。

  例如，如果要设计用于在基于泛型的集合中创建项的类，则可能必须实现一个接口，例如 IComparable<T>，其中 `T` 为类的类型。

有关简单泛型类的示例，请参阅泛型介绍。

类型参数和约束的规则对于泛型类行为具有多种含义，尤其是在继承性和成员可访问性方面。 应当了解一些术语，然后再继续。 对于泛型类 `Node<T>`，客户端代码可以通过指定类型参数来引用该类，以创建封闭构造类型（`Node<int>`）;或者保留未指定类型参数（例如，指定泛型基类时）创建打开的构造类型（`Node<T>`）。 泛型类可继承自具体的封闭式构造或开放式构造基类：

```
C#
```

```
class BaseNode { }
class BaseNodeGeneric<T> { }

// concrete type
class NodeConcrete<T> : BaseNode { }

//closed constructed type
class NodeClosed<T> : BaseNodeGeneric<int> { }

//open constructed type
class NodeOpen<T> : BaseNodeGeneric<T> { }
```

非泛型类（即，具体类）可继承自封闭式构造基类，但不可继承自开放式构造类或类型参数，因为运行时客户端代码无法提供实例化基类所需的类型参数。

C#

```
//No error
class Node1 : BaseNodeGeneric<int> { }

//Generates an error
//class Node2 : BaseNodeGeneric<T> {}

//Generates an error
//class Node3 : T {}
```

继承自开放式构造类型的泛型类必须对非此继承类共享的任何基类类型参数提供类型参数，如下方代码所示：

C#

```
class BaseNodeMultiple<T, U> { }

//No error
class Node4<T> : BaseNodeMultiple<T, int> { }

//No error
class Node5<T, U> : BaseNodeMultiple<T, U> { }

//Generates an error
//class Node6<T> : BaseNodeMultiple<T, U> {}
```

继承自开放式构造类型的泛型类必须指定作为基类型上约束超集或表示这些约束的约束：

C#

```
class NodeItem<T> where T : System.IComparable<T>, new() { }
class SpecialNodeItem<T> : NodeItem<T> where T : System.IComparable<T>,
```

```
new() { }
```

泛型类型可使用多个类型参数和约束，如下所示：

```
C#

class SuperKeyType<K, V, U>
    where U : System.IComparable<U>
    where V : new()
{ }
```

开放式构造和封闭式构造类型可用作方法参数：

```
C#

void Swap<T>(List<T> list1, List<T> list2)
{
    //code to swap items
}

void Swap(List<int> list1, List<int> list2)
{
    //code to swap items
}
```

如果一个泛型类实现一个接口，则该类的所有实例均可强制转换为该接口。

泛型类是不变量。 换而言之，如果一个输入参数指定 `List<BaseClass>`，且你尝试提供 `List<DerivedClass>`，则会出现编译时错误。

另请参阅

- System.Collections.Generic
- 泛型
- Saving the State of Enumerators（保存枚举器状态）
- An Inheritance Puzzle, Part One（继承测验，第一部分）

泛型接口（C# 编程指南）

项目 • 2024/03/13

为泛型集合类或表示集合中的项的泛型类定义接口通常很有用处。 为避免对值类型执行装箱和取消装箱操作，最好对泛型类使用泛型接口，例如 IComparable<T>。 .NET 类库定义多个泛型接口，以便用于 System.Collections.Generic 命名空间中的集合类。 有关这些接口的详细信息，请参阅泛型接口。

接口被指定为类型参数上的约束时，仅可使用实现接口的类型。 如下代码示例演示一个派生自 `GenericList<T>` 类的 `SortedList<T>` 类。 有关详细信息，请参阅泛型介绍。 `SortedList<T>` 添加约束 `where T : IComparable<T>`。 此约束可使 `SortedList<T>` 中的 `BubbleSort` 方法在列表元素上使用泛型 CompareTo 方法。 在此示例中，列表元素是一个实现 `IComparable<Person>` 的简单类 `Person`。

C#

```
//Type parameter T in angle brackets.
public class GenericList<T> : System.Collections.Generic.IEnumerable<T>
{
    protected Node head;
    protected Node current = null;

    // Nested class is also generic on T
    protected class Node
    {
        public Node next;
        private T data;  //T as private member datatype

        public Node(T t)  //T used in non-generic constructor
        {
            next = null;
            data = t;
        }

        public Node Next
        {
            get { return next; }
            set { next = value; }
        }

        public T Data  //T as return type of property
        {
            get { return data; }
            set { data = value; }
        }
    }

    public GenericList()  //constructor
    {
```

```
        head = null;
    }

    public void AddHead(T t)  //T as method parameter type
    {
        Node n = new Node(t);
        n.Next = head;
        head = n;
    }

    // Implementation of the iterator
    public System.Collections.Generic.IEnumerator<T> GetEnumerator()
    {
        Node current = head;
        while (current != null)
        {
            yield return current.Data;
            current = current.Next;
        }
    }

    // IEnumerable<T> inherits from IEnumerable, therefore this class
    // must implement both the generic and non-generic versions of
    // GetEnumerator. In most cases, the non-generic method can
    // simply call the generic method.
    System.Collections.IEnumerator
System.Collections.IEnumerable.GetEnumerator()
    {
        return GetEnumerator();
    }
}

public class SortedList<T> : GenericList<T> where T : System.IComparable<T>
{
    // A simple, unoptimized sort algorithm that
    // orders list elements from lowest to highest:

    public void BubbleSort()
    {
        if (null == head || null == head.Next)
        {
            return;
        }
        bool swapped;

        do
        {
            Node previous = null;
            Node current = head;
            swapped = false;

            while (current.next != null)
            {
                //  Because we need to call this method, the SortedList
                //  class is constrained on IComparable<T>
```

```
                if (current.Data.CompareTo(current.next.Data) > 0)
                {
                    Node tmp = current.next;
                    current.next = current.next.next;
                    tmp.next = current;

                    if (previous == null)
                    {
                        head = tmp;
                    }
                    else
                    {
                        previous.next = tmp;
                    }
                    previous = tmp;
                    swapped = true;
                }
                else
                {
                    previous = current;
                    current = current.next;
                }
            }
        } while (swapped);
    }
}

// A simple class that implements IComparable<T> using itself as the
// type argument. This is a common design pattern in objects that
// are stored in generic lists.
public class Person : System.IComparable<Person>
{
    string name;
    int age;

    public Person(string s, int i)
    {
        name = s;
        age = i;
    }

    // This will cause list elements to be sorted on age values.
    public int CompareTo(Person p)
    {
        return age - p.age;
    }

    public override string ToString()
    {
        return name + ":" + age;
    }

    // Must implement Equals.
    public bool Equals(Person p)
    {
```

```
        return (this.age == p.age);
    }
}

public class Program
{
    public static void Main()
    {
        //Declare and instantiate a new generic SortedList class.
        //Person is the type argument.
        SortedList<Person> list = new SortedList<Person>();

        //Create name and age values to initialize Person objects.
        string[] names =
        [
            "Franscoise",
            "Bill",
            "Li",
            "Sandra",
            "Gunnar",
            "Alok",
            "Hiroyuki",
            "Maria",
            "Alessandro",
            "Raul"
        ];

        int[] ages = [45, 19, 28, 23, 18, 9, 108, 72, 30, 35];

        //Populate the list.
        for (int x = 0; x < 10; x++)
        {
            list.AddHead(new Person(names[x], ages[x]));
        }

        //Print out unsorted list.
        foreach (Person p in list)
        {
            System.Console.WriteLine(p.ToString());
        }
        System.Console.WriteLine("Done with unsorted list");

        //Sort the list.
        list.BubbleSort();

        //Print out sorted list.
        foreach (Person p in list)
        {
            System.Console.WriteLine(p.ToString());
        }
        System.Console.WriteLine("Done with sorted list");
    }
}
```

可将多个接口指定为单个类型上的约束，如下所示：

C#

```
class Stack<T> where T : System.IComparable<T>, IEnumerable<T>
{
}
```

一个接口可定义多个类型参数，如下所示：

C#

```
interface IDictionary<K, V>
{
}
```

适用于类的继承规则也适用于接口：

C#

```
interface IMonth<T> { }

interface IJanuary : IMonth<int> { }  //No error
interface IFebruary<T> : IMonth<int> { }  //No error
interface IMarch<T> : IMonth<T> { }    //No error
                                       //interface IApril<T>  : IMonth<T, U>
{}  //Error
```

如果泛型接口是协变的（即，仅使用自身的类型参数作为返回值），那么这些接口可继承自非泛型接口。 在 .NET 类库中，IEnumerable<T> 继承自 IEnumerable，因为 IEnumerable<T> 在 GetEnumerator 的返回值和 Current 属性 Getter 中仅使用 `T`。

具体类可实现封闭式构造接口，如下所示：

C#

```
interface IBaseInterface<T> { }

class SampleClass : IBaseInterface<string> { }
```

只要类形参列表提供接口所需的所有实参，泛型类即可实现泛型接口或封闭式构造接口，如下所示：

C#

```
interface IBaseInterface1<T> { }
interface IBaseInterface2<T, U> { }

class SampleClass1<T> : IBaseInterface1<T> { }          //No error
class SampleClass2<T> : IBaseInterface2<T, string> { }  //No error
```

控制方法重载的规则对泛型类、泛型结构或泛型接口内的方法一样。 有关详细信息，请参阅泛型方法。

从 C# 11 开始，接口可以声明 `static abstract` 或 `static virtual` 成员。 声明任一 `static abstract` 或 `static virtual` 成员的接口几乎始终是泛型接口。 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。 这些成员通常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。 然后，编译器使用提供的类型参数来解析声明成员的类型。

另请参阅

- 泛型介绍
- interface
- 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

.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

泛型方法（C# 编程指南）

项目 • 2024/04/01

泛型方法是通过类型参数声明的方法，如下所示：

C#

```
static void Swap<T>(ref T lhs, ref T rhs)
{
    T temp;
    temp = lhs;
    lhs = rhs;
    rhs = temp;
}
```

如下示例演示使用类型参数的 `int` 调用方法的一种方式：

C#

```
public static void TestSwap()
{
    int a = 1;
    int b = 2;

    Swap<int>(ref a, ref b);
    System.Console.WriteLine(a + " " + b);
}
```

还可省略类型参数，编译器将推断类型参数。 如下 `Swap` 调用等效于之前的调用：

C#

```
Swap(ref a, ref b);
```

类型推理的相同规则适用于静态方法和实例方法。 编译器可基于传入的方法参数推断类型参数；而无法仅根据约束或返回值推断类型参数。 因此，类型推理不适用于不具有参数的方法。 类型推理发生在编译时，之后编译器尝试解析重载的方法签名。 编译器将类型推理逻辑应用于共用同一名称的所有泛型方法。 在重载解决方案步骤中，编译器仅包含在其上类型推理成功的泛型方法。

在泛型类中，非泛型方法可访问类级别类型参数，如下所示：

C#

```
class SampleClass<T>
{
    void Swap(ref T lhs, ref T rhs) { }
}
```

如果定义一个具有与包含类相同的类型参数的泛型方法，则编译器会生成警告 CS0693，因为在该方法范围内，向内 `T` 提供的参数会隐藏向外 `T` 提供的参数。 如果需要使用类型参数（而不是类实例化时提供的参数）调用泛型类方法所具备的灵活性，请考虑为此方法的类型参数提供另一标识符，如下方示例中 `GenericList2<T>` 所示。

C#

```
class GenericList<T>
{
    // CS0693.
    void SampleMethod<T>() { }
}

class GenericList2<T>
{
    // No warning.
    void SampleMethod<U>() { }
}
```

使用约束在方法中的类型参数上实现更多专用操作。 此版 `Swap<T>` 现名为 `SwapIfGreater<T>`，仅可用于实现 IComparable<T> 的类型参数。

C#

```
void SwapIfGreater<T>(ref T lhs, ref T rhs) where T : System.IComparable<T>
{
    T temp;
    if (lhs.CompareTo(rhs) > 0)
    {
        temp = lhs;
        lhs = rhs;
        rhs = temp;
    }
}
```

泛型方法可重载在数个泛型参数上。 例如，以下方法可全部位于同一类中：

C#

```
void DoWork() { }
void DoWork<T>() { }
```

```
void DoWork<T, U>() { }
```

还可使用类型参数作为方法的返回类型。 下面的代码示例显示一个返回 `T` 类型数组的方法：

C#

```
T[] Swap<T>(T a, T b)
{
    return [b, a];
}
```

C# 语言规范

有关详细信息，请参阅 C# 语言规范。

另请参阅

- System.Collections.Generic
- 泛型介绍
- 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

泛型和数组（C# 编程指南）

项目 • 2024/03/13

下限为零的单维数组自动实现 IList<T>。 这可使你创建可使用相同代码循环访问数组和其他集合类型的泛型方法。 此技术的主要用处在于读取集合中的数据。 IList<T> 接口无法用于添加元素或从数组删除元素。 如果在此上下文中尝试对数组调用 IList<T> 方法（例如 RemoveAt），则会引发异常。

如下代码示例演示具有 IList<T> 输入参数的单个泛型方法如何可循环访问列表和数组（此例中为整数数组）。

C#

```
class Program
{
    static void Main()
    {
        int[] arr = [0, 1, 2, 3, 4];
        List<int> list = new List<int>();

        for (int x = 5; x < 10; x++)
        {
            list.Add(x);
        }

        ProcessItems<int>(arr);
        ProcessItems<int>(list);
    }

    static void ProcessItems<T>(IList<T> coll)
    {
        // IsReadOnly returns True for the array and False for the List.
        System.Console.WriteLine
            ("IsReadOnly returns {0} for this collection.",
            coll.IsReadOnly);

        // The following statement causes a run-time exception for the
        // array, but not for the List.
        //coll.RemoveAt(4);

        foreach (T item in coll)
        {
            System.Console.Write(item?.ToString() + " ");
        }
        System.Console.WriteLine();
    }
}
```

另请参阅

- System.Collections.Generic
- 泛型
- 数组
- 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

泛型委托（C# 编程指南）

项目 • 2024/03/13

委托可以定义它自己的类型参数。 引用泛型委托的代码可以指定类型参数以创建封闭式构造类型，就像实例化泛型类或调用泛型方法一样，如以下示例中所示：

```
C#

public delegate void Del<T>(T item);
public static void Notify(int i) { }

Del<int> m1 = new Del<int>(Notify);
```

C# 2.0 版具有一种称为方法组转换的新功能，适用于具体委托类型和泛型委托类型，使你能够使用此简化语法编写上一行：

```
C#

Del<int> m2 = Notify;
```

在泛型类中定义的委托可以用类方法使用的相同方式来使用泛型类类型参数。

```
C#

class Stack<T>
{
    public delegate void StackDelegate(T[] items);
}
```

引用委托的代码必须指定包含类的类型参数，如下所示：

```
C#

private static void DoWork(float[] items) { }

public static void TestStack()
{
    Stack<float> s = new Stack<float>();
    Stack<float>.StackDelegate d = DoWork;
}
```

根据典型设计模式定义事件时，泛型委托特别有用，因为发件人参数可以为强类型，无需在它和 Object 之间强制转换。

C#

```
delegate void StackEventHandler<T, U>(T sender, U eventArgs);

class Stack<T>
{
    public class StackEventArgs : System.EventArgs { }
    public event StackEventHandler<Stack<T>, StackEventArgs>? StackEvent;

    protected virtual void OnStackChanged(StackEventArgs a)
    {
        if (StackEvent is not null)
            StackEvent(this, a);
    }
}

class SampleClass
{
    public void HandleStackChange<T>(Stack<T> stack, Stack<T>.StackEventArgs
args) { }
}

public static void Test()
{
    Stack<double> s = new Stack<double>();
    SampleClass o = new SampleClass();
    s.StackEvent += o.HandleStackChange;
}
```

另请参阅

- System.Collections.Generic
- 泛型介绍
- 泛型方法
- 泛型类
- 泛型接口
- 委托
- 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

.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++ 模板和 C# 泛型之间的区别（C# 编程指南）

项目 • 2023/04/07

C# 泛型和 C++ 模板均是支持参数化类型的语言功能。 但是，两者之间存在很多不同。 在语法层次，C# 泛型是参数化类型的一个更简单的方法，而不具有 C++ 模板的复杂性。 此外，C# 不试图提供 C++ 模板所具有的所有功能。 在实现层次，主要区别在于 C# 泛型类型的替换在运行时执行，从而为实例化对象保留了泛型类型信息。 有关详细信息，请参阅运行时中的泛型。

以下是 C# 泛型和 C++ 模板之间的主要差异：

- C# 泛型的灵活性与 C++ 模板不同。 例如，虽然可以调用 C# 泛型类中的用户定义的运算符，但是无法调用算术运算符。
- C# 不允许使用非类型模板参数，如 `template C<int i> {}`。
- C# 不支持显式定制化；即特定类型模板的自定义实现。
- C# 不支持部分定制化：部分类型参数的自定义实现。
- C# 不允许将类型参数用作泛型类型的基类。
- C# 不允许类型参数具有默认类型。
- 在 C# 中，泛型类型参数本身不能是泛型，但是构造类型可以用作泛型。 C++ 允许使用模板参数。
- C++ 允许在模板中使用可能并非对所有类型参数有效的代码，随后针对用作类型参数的特定类型检查此代码。 C# 要求类中编写的代码可处理满足约束的任何类型。 例如，在 C++ 中可以编写一个函数，此函数对类型参数的对象使用算术运算符 `+` 和 `-`，在实例化具有不支持这些运算符的类型的模板时，此函数将产生错误。 C# 不允许此操作；唯一允许的语言构造是可以从约束中推断出来的构造。

另请参阅

- C# 编程指南
- 泛型介绍
- 模板

运行时中的泛型（C# 编程指南）

项目 • 2024/04/23

泛型类型或方法编译为公共中间语言 (CIL) 时，它包含将其标识为具有类型参数的元数据。 如何使用泛型类型的 CIL 根据所提供的类型参数是值类型还是引用类型而有所不同。

使用值类型作为参数首次构造泛型类型时，运行时创建专用的泛型类型，CIL 内的适当位置替换提供的一个或多个参数。 为每个用作参数的唯一值类型一次创建专用化泛型类型。

例如，假定程序代码声明了一个由整数构造的堆栈：

C#

```
Stack<int>? stack;
```

此时，运行时生成一个专用版 Stack<T> 类，其中用整数相应地替换其参数。 现在，每当程序代码使用整数堆栈时，运行时都重新使用已生成的专用 Stack<T> 类。 在下面的示例中创建了两个整数堆栈实例，且它们共用 `Stack<int>` 代码的一个实例：

C#

```
Stack<int> stackOne = new Stack<int>();
Stack<int> stackTwo = new Stack<int>();
```

但是，假定在代码中另一点上再创建一个将不同值类型（例如 `long` 或用户定义结构）作为参数的 Stack<T> 类。 其结果是，运行时在 CIL 中生成另一个版本的泛型类型并在适当位置替换 `long`。 转换已不再必要，因为每个专用化泛型类本机包含值类型。

对于引用类型，泛型的作用方式略有不同。 首次使用任意引用类型构造泛型类型时，运行时创建一个专用化泛型类型，用对象引用替换 CIL 中的参数。 之后，每次使用引用类型作为参数实例化已构造的类型时，无论何种类型，运行时皆重新使用先前创建的专用版泛型类型。 原因可能在于所有引用大小相同。

例如，假定有两个引用类型、一个 `Customer` 类和一个 `Order` 类，并假定已创建 `Customer` 类型的堆栈：

C#

```
class Customer { }
class Order { }
```

```C#
Stack<Customer> customers;
```

此时，运行时生成一个专用版 Stack<T> 类，此类存储之后会被填写的引用类型，而不是存储数据。 假定下一行代码创建另一引用类型的堆栈，其名为 `Order`：

```C#
Stack<Order> orders = new Stack<Order>();
```

不同于值类型，不会为 `Order` 类型创建 Stack<T> 类的另一专用版。 相反，创建专用版 Stack<T> 类的实例并将 `orders` 变量设置为引用此实例。 假定之后遇到一行创建 `Customer` 类型堆栈的代码：

```C#
customers = new Stack<Customer>();
```

与之前使用通过 `Order` 类型创建的 Stack<T> 类一样，会创建专用 Stack<T> 类的另一个实例。 其中包含的指针设置为引用 `Customer` 类型大小的内存区。 由于引用类型的数量因程序不同而有较大差异，因此通过将编译器为引用类型的泛型类创建的专用类的数量减少至 1，泛型的 C# 实现可极大减少代码量。

此外，使用值类型或引用类型参数实例化泛型 C# 类时，反射可在运行时对其进行查询，且其实际类型和类型参数皆可被确定。

另请参阅

- System.Collections.Generic
- 泛型介绍
- 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

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

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