

Contents

C# 文档

入门

简介

类型

程序构建基块

主要语言区域

教程

选择你的第一课

基于浏览器的教程

Hello world

C# 中的数字

分支和循环

列表集合

在本地环境中工作

设置你的环境

C# 中的数字

分支和循环

列表集合

C# 中的新增功能

C# 9.0

C# 8.0

C# 7.0-7.3

编译器的重大更改

C# 版本历史记录

与 .NET 库的关系

版本兼容性

教程

浏览记录类型

探索顶级语句

[探索对象中的模式](#)

[使用默认接口方法安全地更新接口](#)

[用默认接口方法创建 mixin 功能](#)

[探索索引和范围](#)

[使用可为空引用类型](#)

[将应用升级为可为空引用类型](#)

[生成和使用异步流](#)

教程

[类简介](#)

[面向对象的编程](#)

[探索字符串内插 - 交互式](#)

[探索字符串内插 - 在环境中](#)

[字符串内插的高级方案](#)

[控制台应用程序](#)

[REST 客户端](#)

[C# 和 .NET 中的继承](#)

[使用 LINQ](#)

[使用特性](#)

[使用模式匹配生成数据驱动的算法](#)

C# 概念

[C# 类型系统](#)

[可为空引用类型](#)

[选择用于启用可为空引用类型的策略](#)

[命名空间](#)

[基本类型](#)

[类](#)

[析构元组和其他类型](#)

[接口](#)

[方法](#)

[属性](#)

[索引器](#)

[弃元](#)

[泛型](#)

[迭代器](#)

[委托和事件](#)

[委托简介](#)

[System.Delegate 和 delegate 关键字](#)

[强类型委托](#)

[委托的常见模式](#)

[事件简介](#)

[标准 .NET 事件模式](#)

[已更新的 .NET 事件模式](#)

[区别委托和事件](#)

[语言集成查询 \(LINQ\)](#)

[LINQ 概述](#)

[查询表达式基础](#)

[C# 中的 LINQ](#)

[在 C# 中编写 LINQ 查询](#)

[查询对象的集合](#)

[从方法中返回查询](#)

[在内存中存储查询结果](#)

[对查询结果进行分组](#)

[创建嵌套组](#)

[对分组操作执行子查询](#)

[按连续键对结果进行分组](#)

[在运行时动态指定谓词筛选器](#)

[执行内部联接](#)

[执行分组联接](#)

[执行左外部联接](#)

[对 join 子句的结果进行排序](#)

[使用组合键进行联接](#)

[执行自定义联接操作](#)

[在查询表达式中处理 null 值](#)

[在查询表达式中处理异常](#)

[模式匹配](#)

[编写安全高效的代码](#)

[表达式树](#)

[表达式树简介](#)

[表达式树说明](#)

[支持表达式树的框架类型](#)

[执行表达式](#)

[解释表达式](#)

[生成表达式](#)

[翻译表达式](#)

[总结](#)

[本机互操作性](#)

[记录代码](#)

[版本控制](#)

[操作指南 C# 文章](#)

[文章索引](#)

[将字符串拆分为子字符串](#)

[连接字符串](#)

[搜索字符串](#)

[修改字符串内容](#)

[比较字符串](#)

[使用模式匹配以及 is/as 运算符安全地进行强制转换](#)

[.NET Compiler Platform SDK \(Roslyn API\)](#)

[.NET Compiler Platform SDK \(Roslyn API\) 概述](#)

[了解编译器 API 模型](#)

[使用语法](#)

[使用语义](#)

[使用工作区](#)

[使用语法可视化工具浏览代码](#)

[快速入门](#)

[语法分析](#)

[语义分析](#)

语法转换
教程
生成首个分析器和代码修补程序

C# 编程指南

概述

C# 程序内部探究

C# 程序所含内容

C# 程序的通用结构

标识符名称

C# 编码约定

Main() 和命令行参数

概述

命令行自变量

如何显示命令行参数

Main() 返回值

顶级语句

编程概念

概述

异步编程

概述

异步编程场景

异步编程模型

异步返回类型

取消任务

取消任务列表

在一段时间后取消任务

在异步任务完成时对其进行处理

异步文件访问

属性

概述

创建自定义特性

使用反射访问特性

如何使用特性创建 C/C++ 联合 集合

协变和逆变

概述

泛型接口中的变体

创建变体泛型接口

在泛型集合的接口中使用变体

委托中的变体

在委托中使用变体

对 Func 和 Action 泛型委托使用变体

表达式树

概述

如何执行表达式树

如何修改表达式树

如何使用表达式树来生成动态查询

在 Visual Studio 中调试表达式树

DebugView 语法

迭代器

语言集成查询 (LINQ)

概述

C# 中的 LINQ 入门

LINQ 查询简介

LINQ 和泛型类型

基本 LINQ 查询操作

使用 LINQ 进行数据转换

LINQ 查询操作中的类型关系

LINQ 中的查询语法和方法语法

支持 LINQ 的 C# 功能

演练:用 C# 编写查询 (LINQ)

标准查询运算符概述

概述

标准查询运算符的查询表达式语法

[标准查询运算符按执行方式的分类](#)

[对数据进行排序](#)

[Set 运算](#)

[筛选数据](#)

[限定符运算](#)

[投影运算](#)

[数据分区](#)

[联接运算](#)

[数据分组](#)

[生成运算](#)

[相等运算](#)

[元素运算](#)

[转换数据类型](#)

[串联运算](#)

[聚合运算](#)

[LINQ to Objects](#)

[概述](#)

[LINQ 和字符串](#)

[操作指南](#)

[如何对某个词在字符串中出现的次数进行计数 \(LINQ\)](#)

[如何查询包含一组指定词语的句子 \(LINQ\)](#)

[如何查询字符串中的字符 \(LINQ\)](#)

[如何将 LINQ 查询与正则表达式合并在一起](#)

[如何查找两个列表之间的差集 \(LINQ\)](#)

[如何按任意词或字段对文本数据进行排序或筛选 \(LINQ\)](#)

[如何重新排列带分隔符的文件的字段 \(LINQ\)](#)

[如何合并和比较字符串集合 \(LINQ\)](#)

[如何从多个源填充对象集合 \(LINQ\)](#)

[如何使用组将一个文件拆分成多个文件 \(LINQ\)](#)

[如何联接不同文件的内容 \(LINQ\)](#)

[如何在 CSV 文本文件中计算列值 \(LINQ\)](#)

[LINQ 和反射](#)

如何使用反射查询程序集的元数据 (LINQ)

LINQ 和文件目录

概述

[如何查询具有指定特性或名称的文件](#)

[如何按扩展名对文件分组 \(LINQ\)](#)

[如何查询一组文件夹中的总字节数 \(LINQ\)](#)

[如何比较两个文件夹的内容 \(LINQ\)](#)

[如何查询目录树中的一个或多个最大的文件 \(LINQ\)](#)

[如何在目录树中查询重复文件 \(LINQ\)](#)

[如何:查询文件夹中的文件的内容 \(LINQ\)](#)

如何使用 LINQ 查询 ArrayList

如何为 LINQ 查询添加自定义方法

[LINQ to ADO.NET\(门户网站页\)](#)

[启用数据源以进行 LINQ 查询](#)

[对 LINQ 的 Visual Studio IDE 和工具支持](#)

反射

序列化 (C#)

概述

[如何将对象数据写入 XML 文件](#)

[如何从 XML 文件中读取对象数据](#)

[演练:在 Visual Basic 中保持对象](#)

语句、表达式和运算符

概述

语句

Expression-Bodied 成员

匿名函数

概述

[如何在查询中使用 Lambda 表达式](#)

相等和相等性比较

相等比较

[如何为类型定义值相等性](#)

[如何测试引用相等性\(标识\)](#)

类型

[使用和定义类型](#)

[强制转换和类型转换](#)

[装箱和取消装箱](#)

[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](#)

[使用类型 dynamic](#)

[演练: 创建并使用动态对象 \(C# 和 Visual Basic\)](#)

类、结构和记录

[概述](#)

[类](#)

[记录](#)

[对象](#)

[继承](#)

[多形性](#)

[概述](#)

[使用 Override 和 New 关键字进行版本控制](#)

[了解何时使用 Override 和 New 关键字](#)

[如何重写 ToString 方法](#)

[成员](#)

[成员概述](#)

[抽象类、密封类及类成员](#)

[静态类和静态类成员](#)

[访问修饰符](#)

[字段](#)

[常量](#)

[如何定义抽象属性](#)

[如何在 C# 中定义常量](#)

[属性](#)

[属性概述](#)

[使用属性](#)

[接口属性](#)

[限制访问器可访问性](#)

[如何声明和使用读/写属性](#)

[自动实现的属性](#)

[如何: 使用自动实现的属性实现轻量类](#)

[方法](#)

[方法概述](#)

[本地函数](#)

[ref 返回值和局部变量](#)

[参数](#)

[快速参考](#)

[传递值类型参数](#)

[传递引用类型参数](#)

[如何了解向方法传递结构和向方法传递类引用之间的区别](#)

[隐式类型的局部变量](#)

[如何在查询表达式中使用隐式类型的局部变量和数组](#)

[扩展方法](#)

[如何实现和调用自定义扩展方法](#)

[如何为枚举创建新方法](#)

[命名自变量和可选自变量](#)

[如何在 Office 编程中使用命名自变量和可选自变量](#)

[构造函数](#)

[构造函数概述](#)

[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](#)

[终结器](#)

[对象和集合初始值设定项](#)

[如何使用对象初始值设定项初始化对象](#)

[如何使用集合初始值设定项初始化字典](#)

[嵌套类型](#)

[分部类和方法](#)

[匿名类型](#)

[如何在查询中返回元素属性的子集](#)

[接口](#)

[概述](#)

[显式接口实现](#)

[如何显式实现接口成员](#)

[如何显式实现两个接口的成员](#)

[委托](#)

[概述](#)

[使用委托](#)

[带有命名方法的委托与匿名方法](#)

[如何合并委托\(多播委托\)\(C# 编程指南\)](#)

[如何声明、实例化和使用委托](#)

[数组](#)

[概述](#)

[单维数组](#)

[多维数组](#)

[交错数组](#)

[对数组使用 foreach](#)

[将数组作为参数传递](#)

[隐式类型的数组](#)

[字符串](#)

[使用字符串进行编程](#)

[如何确定字符串是否表示数值](#)

[索引器](#)

[概述](#)

[使用索引器](#)

[接口中的索引器](#)

[属性和索引器之间的比较](#)

[事件](#)

概述

[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](#)

泛型

[概述](#)

[泛型类型参数](#)

[类型参数的约束](#)

[泛型类](#)

[泛型接口](#)

[泛型方法](#)

[泛型和数组](#)

[泛型委托](#)

[C++ 模板和 C# 泛型之间的区别](#)

[运行时中的泛型](#)

[泛型和反射](#)

[泛型和特性](#)

命名空间

[概述](#)

[Using 命名空间](#)

[如何使用 My 命名空间](#)

XML 文档注释

[概述](#)

[建议的文档注释标记](#)

[处理 XML 文件](#)

[文档标记分隔符](#)

[如何使用 XML 文档功能](#)

[文档标记参考](#)

`<c>`

`<code>`

`cref 特性`

`<example>`

`<exception>`

`<include>`

`<inheritdoc>`

`<list>`

`<para>`

`<param>`

`<paramref>`

`<permission>`

`<remarks>`

`<returns>`

`<see>`

`<seealso>`

`<summary>`

`<typeparam>`

`<typeparamref>`

`<value>`

异常和异常处理

`概述`

`使用异常`

`异常处理`

`创建和引发异常`

`编译器生成的异常`

`如何使用 try-catch 处理异常`

`如何使用 finally 执行清理代码`

`如何捕捉非 CLS 异常`

文件系统和注册表

`概述`

`如何循环访问目录树`

`如何获取有关文件、文件夹和驱动器的信息`

`如何创建文件或文件夹`

[如何复制、删除和移动文件和文件夹](#)

[如何提供文件操作进度对话框](#)

[如何写入文本文件](#)

[如何读取文本文件中的内容](#)

[如何一次一行地读取文本文件](#)

[如何在注册表中创建注册表项](#)

[互操作性](#)

[.NET 互操作性](#)

[互操作性概述](#)

[如何使用 C# 功能访问 Office 互操作对象](#)

[如何在 COM 互操作编程中使用索引属性](#)

[如何使用平台调用播放 WAV 文件](#)

[演练:Office 编程\(C# 和 Visual Basic\)](#)

[COM 类示例](#)

[语言参考](#)

[概述](#)

[配置语言版本](#)

[类型](#)

[值类型](#)

[概述](#)

[整型数值类型](#)

[nint 和 nuint 本机整数类型](#)

[浮点型数值类型](#)

[内置数值转换](#)

[bool](#)

[char](#)

[枚举类型](#)

[结构类型](#)

[元组类型](#)

[可以为 null 的值类型](#)

[引用类型](#)

[引用类型的功能](#)

[内置引用类型](#)

[记录 \(record\)](#)

[class](#)

[接口](#)

[可为空引用类型](#)

[void](#)

[var](#)

[内置类型](#)

[非托管类型](#)

[默认值](#)

[关键字](#)

[概述](#)

[修饰符](#)

[访问修饰符](#)

[快速参考](#)

[可访问性级别](#)

[可访问性域](#)

[可访问性级别的使用限制](#)

[内部](#)

[private](#)

[protected](#)

[public](#)

[受保护的内部](#)

[私有受保护](#)

[abstract](#)

[async](#)

[const](#)

[event](#)

[extern](#)

[in\(泛型修饰符\)](#)

[new\(成员修饰符\)](#)

[out\(泛型修饰符\)](#)

`override`

`readonly`

`sealed`

`static`

`unsafe`

`virtual`

`volatile`

语句关键字

`语句类别`

`选择语句`

`if-else`

`开关`

`迭代语句`

`do`

`for`

`foreach, in`

`while`

`跳转语句`

`break`

`continue`

`goto`

`return`

`异常处理语句`

`throw`

`try-catch`

`try-finally`

`try-catch-finally`

`Checked 和 Unchecked`

`概述`

`checked`

`unchecked`

`fixed 语句`

`lock` 语句

方法参数

快速参考

`params`

`in`(参数修饰符)

`ref`

`out`(参数修饰符)

命名空间关键字

命名空间

`using`

`using` 的上下文

`using` 指令

`using static` 指令

`using` 语句

外部别名

泛型类型约束关键字

`new` 约束

`where`

访问关键字

`base`

`this`

文字关键字

`null`

`true` 和 `false`

`default`

上下文关键字

快速参考

`add`

`get`

`init`

`partial`(类型)

`partial`(方法)

[remove](#)

[set](#)

[when\(筛选条件\)](#)

[value](#)

[yield](#)

查询关键字

[快速参考](#)

[from 子句](#)

[where 子句](#)

[select 子句](#)

[group 子句](#)

[更改为](#)

[orderby 子句](#)

[join 子句](#)

[let 子句](#)

[ascending](#)

[descending](#)

[on](#)

[equals](#)

[by](#)

[in](#)

运算符和表达式

概述

算术运算符

Boolean 逻辑运算符

按位和移位运算符

相等运算符

比较运算符

成员访问运算符和表达式

类型测试运算符和强制转换表达式

用户定义的转换运算符

与指针相关的运算符

赋值运算符

Lambda 表达式

模式

+ += 运算符

- -= 运算符

? : 运算符

! (null 包容) 运算符

?? 和 ??= 运算符

=> 运算符

:: 运算符

await 运算符

默认值表达式

delegate 运算符

is 运算符

nameof 表达式

new 运算符

sizeof 运算符

stackalloc 表达式

switch 表达式

true 和 false 运算符

with 表达式

运算符重载

特殊字符

概述

\$ -- 字符串内插

@ -- 逐字字符串标识符

编译器读取的属性

全局属性

调用方信息

可为空的静态分析

杂项

不安全代码和指针

[预处理器指令](#)

[编译器选项](#)

[如何设置选项](#)

[语言选项](#)

[输出选项](#)

[输入选项](#)

[错误和警告选项](#)

[代码生成选项](#)

[安全选项](#)

[资源选项](#)

[杂项选项](#)

[高级选项](#)

[编译器消息](#)

[C# 6.0 草稿规范](#)

[C# 7.0 - 9.0 建议](#)

[演练](#)

C# 语言介绍

2021/3/9 • [Edit Online](#)

C#(读作“See Sharp”)是一种新式编程语言，不仅面向对象，还类型安全。开发人员利用 C# 能够生成在 .NET 生态系统中运行的多种安全可靠的应用程序。C# 源于 C 语言系列，C、C++、Java 和 JavaScript 程序员很快就可以上手使用。本教程概述了 C# 8 及更高版本中该语言的主要组件。如果想要通过交互式示例探索语言，请尝试 [C# 简介](#) 教程。

C# 是面向对象的、面向组件的编程语言。C# 提供了语言构造来直接支持这些概念，让 C# 成为一种非常自然的语言，可用于创建和使用软件组件。自诞生之日起，C# 就添加了支持新工作负载和新兴软件设计实践的功能。

多项 C# 功能有助于创建可靠且持久的应用程序。[垃圾回收](#)* 自动回收不可访问的未用对象所占用的内存。[可以为 null 的类型](#) 可防范不引用已分配对象的变量。[异常处理](#) 提供了一种结构化且可扩展的方法来进行错误检测和恢复。[Lambda 表达式](#) 支持函数编程技术。[语言集成查询 \(LINQ\)](#) 语法规则创建一个公共模式，用于处理来自任何源的数据。[异步操作](#) 语言支持提供用于构建分布式系统的语法。C# 有[统一类型系统](#)*。所有 C# 类型(包括 `int` 和 `double` 等基元类型)均继承自一个根 `object` 类型。所有类型共用一组通用运算。任何类型的值都可以一致地进行存储、传输和处理。此外，C# 还支持用户定义的[引用类型](#)和[值类型](#)。C# 允许动态分配轻型结构的对象和内嵌存储。C# 支持泛型方法和类型，因此增强了类型安全性和性能。C# 可提供迭代器，使集合类的实现者可以定义客户端代码的自定义行为。

C# 强调版本控制，以确保程序和库以兼容方式随时间推移而变化。C# 设计中受版本控制直接影响的方面包括：单独的 `virtual` 和 `override` 修饰符，关于方法重载决策的规则，以及对显式接口成员声明的支持。

.NET 体系结构

C# 程序在 .NET 上运行，而 .NET 是名为公共语言运行时 (CLR) 的虚执行系统和一组类库。CLR 是 Microsoft 对公共语言基础结构 (CLI) 国际标准的实现。CLI 是创建执行和开发环境的基础，语言和库可以在其中无缝地协同工作。

用 C# 编写的源代码被编译成符合 CLI 规范的[中间语言 \(IL\)](#)。IL 代码和资源(如位图和字符串)存储在扩展名通常为 .dll 的程序集中。程序集包含一个介绍程序集的类型、版本和区域性的清单。

执行 C# 程序时，程序集将加载到 CLR。CLR 会直接执行实时 (JIT) 编译，将 IL 代码转换成本机指令。CLR 可提供其他与自动垃圾回收、异常处理和资源管理相关的服务。CLR 执行的代码有时称为“托管代码”(而不是“非托管代码”)，被编译成面向特定平台的本机语言。

语言互操作性是 .NET 的一项重要功能。C# 编译器生成的 IL 代码符合公共类型规范 (CTS)。通过 C# 生成的 IL 代码可以与通过 .NET 版本的 F#、Visual Basic、C++ 或其他 20 多种与 CTS 兼容的任何语言所生成的代码进行交互。一个程序集可能包含多个用不同 .NET 语言编写的模块，且类型可以相互引用，就像是用同一种语言编写的一样。

除了运行时服务之外，.NET 还包含大量库。这些库支持多种不同的工作负载。它们已整理到命名空间中，这些命名空间提供各种实用功能，包括文件输入输出、字符串控制、XML 分析、Web 应用程序框架和 Windows 窗体控件。典型的 C# 应用程序广泛使用 .NET 类库来处理常见的“管道”零碎工作。

有关 .NET 的详细信息，请参阅 [.NET 概述](#)。

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：

```
using System;

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

"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。

"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。按照约定，`Main` 静态方法是 C# 程序的入口点。

程序的输出是由 `System` 命名空间中 `Console` 类的 `WriteLine` 方法生成。此类由标准类库提供。默认情况下，编译器会自动引用标准类库。

类型和变量

C# 有两种类型：值类型和引用类型。值类型的变量直接包含它们的数据。引用类型的变量存储对数据(称为“对象”的引用。对于引用类型，两个变量可以引用同一个对象；对一个变量执行的运算可能会影响另一个变量引用的对象。借助值类型，每个变量都有自己的数据副本；因此，对一个变量执行的运算不会影响另一个变量(`ref` 和 `out` 参数变量除外)。

标识符为变量名称。标识符是不包含任何空格的 unicode 字符序列。如果标识符的前缀为 `@`，则该标识符可以是 C# 保留字。在与其他语言交互时，使用保留字作为标识符很有用。

C# 的值类型进一步分为：简单类型、枚举类型、结构类型、可以为 `null` 的值类型和元组值类型。C# 引用类型又细分为类类型、接口类型、数组类型和委托类型。

以下大纲概述了 C# 的类型系统。

- 值类型

- 简单类型

- 有符号整型：`sbyte`、`short`、`int`、`long`
 - 无符号整型：`byte`、`ushort`、`uint`、`ulong`
 - Unicode 字符：`char`，表示 UTF-16 代码单元
 - IEEE 二进制浮点：`float`、`double`
 - 高精度十进制浮点数：`decimal`
 - 布尔值：`bool`，表示布尔值(`true` 或 `false`)

- 枚举类型

- `enum E { ... }` 格式的用户定义类型。`enum` 类型是一种包含已命名常量的独特类型。每个 `enum` 类型都有一个基础类型(必须是八种整型类型之一)。`enum` 类型的值集与基础类型的值集相同。

- 结构类型

- 格式为 `struct S { ... }` 的用户定义类型
 - 可以为 `null` 的值类型
 - 值为 `null` 的其他所有值类型的扩展

- 元组值类型
 - 格式为 `(T1, T2, ...)` 的用户定义类型
- 引用类型
 - 类类型
 - 其他所有类型的最终基类: `object`
 - Unicode 字符串: `string`, 表示 UTF-16 代码单元序列
 - 格式为 `class C {...}` 的用户定义类型
 - 接口类型
 - 格式为 `interface I {...}` 的用户定义类型
 - 数组类型
 - 一维、多维和交错。例如: `int[]`、`int[,]` 和 `int[][]`
 - 委托类型
 - 格式为 `delegate int D(...)` 的用户定义类型

C# 程序使用 **类型声明** 创建新类型。类型声明指定新类型的名称和成员。用户可定义以下六种 C# 类型:类类型、结构类型、接口类型、枚举类型、委托类型和元组值类型。

- `class` 类型定义包含数据成员(字段)和函数成员(方法、属性及其他)的数据结构。类类型支持单一继承和多形性, 即派生类可以扩展和专门针对基类的机制。
- `struct` 类型定义包含数据成员和函数成员的结构, 这一点与类类型相似。不过, 与类不同的是, 结构是值类型, 通常不需要进行堆分配。结构类型不支持用户指定的继承, 并且所有结构类型均隐式继承自类型 `object`。
- `interface` 类型将协定定义为一组已命名的公共成员。实现 `interface` 的 `class` 或 `struct` 必须提供接口成员的实现代码。`interface` 可以继承自多个基接口, `class` 和 `struct` 可以实现多个接口。
- `delegate` 类型表示引用包含特定参数列表和返回类型的函数。通过委托, 可以将方法视为可分配给变量并可作为参数传递的实体。委托类同于函数式语言提供的函数类型。它们还类似于其他一些语言中存在的“函数指针”概念。与函数指针不同, 委托是面向对象且类型安全的。

`class`、`struct`、`interface` 和 `delegate` 类型全部都支持泛型, 因此可以使用其他类型对它们进行参数化。

C# 支持任意类型的一维和多维数组。与上述类型不同, 数组类型无需先声明即可使用。相反, 数组类型是通过在类型名称后面添加方括号构造而成。例如, `int[]` 是 `int` 类型的一维数组, `int[,]` 是 `int` 类型的二维数组, `int[][]` 是由 `int` 类型的一维数组或“交错”数组构成的一维数组。

可以为 `null` 的类型不需要单独定义。对于所有不可以为 `null` 的类型 `T`, 都有对应的可以为 `null` 的类型 `T?`, 后者可以包含附加值 `null`。例如, `int?` 是可保存任何 32 位整数或 `null` 值的类型, `string?` 是可以保存任何 `string` 或 `null` 值的类型。

C# 采用统一的类型系统, 因此任意类型的值都可视为 `object`。每种 C# 类型都直接或间接地派生自 `object` 类类型, 而 `object` 是所有类型的最终基类。只需将值视为类型 `object`, 即可将引用类型的值视为对象。通过执行 **装箱** 和 **取消装箱操作**, 可以将值类型的值视为对象。在以下示例中, `int` 值被转换成 `object`, 然后又恢复成 `int`。

```
int i = 123;
object o = i;    // Boxing
int j = (int)o; // Unboxing
```

将值类型的值分配给 `object` 对象引用时, 会分配一个“箱”来保存此值。该箱是引用类型的实例, 此值会被复制到该箱。相反, 当 `object` 引用被显式转换成值类型时, 将检查引用的 `object` 是否是具有正确值类型的箱。如果检查成功, 则会将箱中的值复制到值类型。

C# 的统一类型系统实际上意味着“按需”将值类型视为 `object` 引用。鉴于这种统一性, 使用类型 `object` 的常

规用途库可以与派生自 `object` 的所有类型结合使用，包括引用类型和值类型。

C# 有多种 变量，其中包括字段、数组元素、局部变量和参数。变量表示存储位置。每个变量都具有一种类型，用于确定可以在变量中存储哪些值，如下文所述。

- 不可以为 `null` 的值类型
 - 具有精确类型的值
- 可以为 `null` 的值类型
 - `null` 值或具有精确类型的值
- `object`
 - `null` 引用、对任意引用类型的对象的引用，或对任意值类型的装箱值的引用
- 类类型
 - `null` 引用、对类类型实例的引用，或对派生自类类型的类实例的引用
- 接口类型
 - `null` 引用、对实现接口类型的类类型实例的引用，或对实现接口类型的值类型的装箱值的引用
- 数组类型
 - `null` 引用、对数组类型实例的引用，或对兼容的数组类型实例的引用
- 委托类型
 - `null` 引用或对兼容的委托类型实例的引用

程序结构

C# 中的关键组织结构概念包括 [程序](#)、[命名空间](#)、[类型](#)、[成员](#) 和 [程序集](#)*。程序声明类型，而类型则包含成员，并被整理到命名空间中。类型示例包括类、结构和接口。成员示例包括字段、方法、属性和事件。编译完的 C# 程序实际上会打包到程序集中。程序集的文件扩展名通常为 `.exe` 或 `.dll`，具体视其分别实现的是应用程序还是库_*而定。

作为一个小示例，请考虑包含以下代码的程序集：

```

using System;

namespace Acme.Collections
{
    public class Stack<T>
    {
        Entry _top;

        public void Push(T data)
        {
            _top = new Entry(_top, data);
        }

        public T Pop()
        {
            if (_top == null)
            {
                throw new InvalidOperationException();
            }
            T result = _top.Data;
            _top = _top.Next;

            return result;
        }

        class Entry
        {
            public Entry Next { get; set; }
            public T Data { get; set; }

            public Entry(Entry next, T data)
            {
                Next = next;
                Data = data;
            }
        }
    }
}

```

此类的完全限定的名称为 `Acme.Collections.Stack`。此类包含多个成员：一个 `top` 字段、两个方法（`Push` 和 `Pop`）和一个 `Entry` 嵌套类。`Entry` 类还包含三个成员：一个 `next` 字段、一个 `data` 字段和一个构造函数。`Stack` 是泛型类。它具有一个类型参数 `T`，在使用时替换为具体类型。

NOTE

堆栈是一个“先进后出”(FILO) 集合。添加到堆栈顶部的新元素。删除元素时，将从堆栈顶部删除该元素。

程序集包含中间语言 (IL) 指令形式的可执行代码和元数据形式的符号信息。执行前，.NET 公共语言运行时的实时 (JIT) 编译器会将程序集中的 IL 代码转换为特定于处理器的代码。

由于程序集是包含代码和元数据的自描述功能单元，因此无需在 C# 中使用 `#include` 指令和头文件。只需在编译程序时引用特定的程序集，即可在 C# 程序中使用此程序集中包含的公共类型和成员。例如，此程序使用 `acme.dll` 程序集中的 `Acme.Collections.Stack` 类：

```
using System;
using Acme.Collections;

class Example
{
    public static void Main()
    {
        var s = new Stack<int>();
        s.Push(1); // stack contains 1
        s.Push(10); // stack contains 1, 10
        s.Push(100); // stack contains 1, 10, 100
        Console.WriteLine(s.Pop()); // stack contains 1, 10
        Console.WriteLine(s.Pop()); // stack contains 1
        Console.WriteLine(s.Pop()); // stack is empty
    }
}
```

若要编译此程序，需要引用包含前面示例中定义的堆栈类的程序集。

C# 程序可以存储在多个源文件中。在编译 C# 程序时，将同时处理所有源文件，并且源文件可以自由地相互引用。从概念上讲，就好像所有源文件在被处理之前都连接到一个大文件。在 C# 中永远都不需要使用前向声明，因为声明顺序无关紧要(极少数例外情况除外)。C# 并不限制源文件只能声明一种公共类型，也不要求源文件的文件名必须与其中声明的类型相匹配。

本教程中的后续文章介绍了这些组织块。

类型和成员

2021/3/5 • • [Edit Online](#)

作为面向对象的语言，C# 支持封装、继承和多态性这些概念。类可能会直接继承一个父类，并且可以实现任意数量的接口。若要用方法重写父类中的虚方法，必须使用 `override` 关键字，以免发生意外重定义。在 C# 中，结构就像是轻量级类，是可以实现接口但不支持继承的堆栈分配类型。C# 还可提供记录，这些记录是主要用于存储数据值的类类型。

类和对象

类是最基本的 C# 类型。类是一种数据结构，可在同一个单元中就将状态(字段)和操作(方法和其他函数成员)结合起来。类为类实例(亦称为“对象”)提供了定义。类支持 继承 和 多形性，即 派生类 可以扩展和专门针对 基类 的机制。

新类使用类声明进行创建。类声明以标头开头。标头指定以下内容：

- 类的特性和修饰符
- 类的名称
- 基类(从基类继承时)
- 接口由该类实现。

标头后面是类主体，由在分隔符 `{` 和 `}` 内编写的成员声明列表组成。

以下代码展示的是简单类 `Point` 的声明：

```
public class Point
{
    public int X { get; }
    public int Y { get; }

    public Point(int x, int y) => (X, Y) = (x, y);
}
```

类实例是使用 `new` 运算符进行创建，此运算符为新实例分配内存，调用构造函数来初始化实例，并返回对实例的引用。以下语句创建两个 `Point` 对象，并将对这些对象的引用存储在两个变量中：

```
var p1 = new Point(0, 0);
var p2 = new Point(10, 20);
```

当无法再访问对象时，对象占用的内存会被自动回收。没有必要也无法在 C# 中显式解除分配对象。

类型参数

泛型类定义 [类型参数](#)*。类型参数是用尖括号括起来的类型参数名称列表。类型参数跟在类名后面。然后，可以在类声明的主体中使用类型参数来定义类成员。在以下示例中，`Pair` 的类型参数是 `TFirst` 和 `TSecond`：

```
public class Pair<TFirst, TSecond>
{
    public TFirst First { get; }
    public TSecond Second { get; }

    public Pair(TFirst first, TSecond second) =>
        (First, Second) = (first, second);
}
```

声明为需要使用类型参数的类类型被称为“泛型类类型”。结构、接口和委托类型也可以是泛型。使用泛型类时，必须为每个类型参数提供类型自变量：

```
var pair = new Pair<int, string>(1, "two");
int i = pair.First;      // TFirst int
string s = pair.Second; // TSecond string
```

包含类型自变量的泛型类型(如上面的 `Pair<int, string>`)被称为 **构造泛型类型**。

基类

类声明可以指定基类。在类名和类型参数后面加上冒号和基类的名称。省略基类规范与从 `object` 类型派生相同。在以下示例中，`Point3D` 的基类是 `Point` 在第一个示例中，`Point` 的基类是 `object`：

```
public class Point3D : Point
{
    public int Z { get; set; }

    public Point3D(int x, int y, int z) : base(x, y)
    {
        Z = z;
    }
}
```

类继承其基类的成员。继承意味着一个类隐式包含其基类的几乎所有成员。类不继承实例、静态构造函数以及终结器。派生类可以在其继承的成员中添加新成员，但无法删除继承成员的定义。在上面的示例中，`Point3D` 从 `Point` 继承了 `X` 和 `Y` 成员，每个 `Point3D` 实例均包含三种属性(`X`、`Y` 和 `Z`)。

可以将类类型隐式转换成其任意基类类型。类类型的变量可以引用相应类的实例或任意派生类的实例。例如，类声明如上，`Point` 类型的变量可以引用 `Point` 或 `Point3D`：

```
Point a = new Point(10, 20);
Point b = new Point3D(10, 20, 30);
```

结构

类定义可支持继承和多形性的类型。它们使你能够基于派生类的层次结构创建复杂的行为。相比之下，[结构](#) 类型是较为简单的类型，其主要目的是存储数据值*。结构不能声明基类型；它们从 `System.ValueType` 隐式派生。不能从 `struct` 类型派生其他 `struct` 类型。这些类型已隐式密封。

```
public struct Point
{
    public double X { get; }
    public double Y { get; }

    public Point(double x, double y) => (X, Y) = (x, y);
}
```

接口

接口定义了可由类和结构实现的协定。接口可以包含方法、属性、事件和索引器。接口通常不提供所定义成员的实现，仅指定必须由实现接口的类或结构提供的成员。

接口可以采用“多重继承”。在以下示例中，接口 `IComboBox` 同时继承自 `ITextBox` 和 `IListBox`。

```
interface IControl
{
    void Paint();
}

interface ITextBox : IControl
{
    void SetText(string text);
}

interface IListBox : IControl
{
    void SetItems(string[] items);
}

interface IComboBox : ITextBox, IListBox { }
```

类和结构可以实现多个接口。在以下示例中，类 `EditBox` 同时实现 `IControl` 和 `IDataBound`。

```
interface IDataBound
{
    void Bind(Binder b);
}

public class EditBox : IControl, IDataBound
{
    public void Paint() { }
    public void Bind(Binder b) { }
}
```

当类或结构实现特定接口时，此类或结构的实例可以隐式转换成相应的接口类型。例如

```
EditBox editBox = new EditBox();
IControl control = editBox;
IDataBound dataBound = editBox;
```

枚举

枚举类型定义了一组常数值。以下 `enum` 声明了定义不同根蔬菜的常数：

```
public enum SomeRootVegetable
{
    HorseRadish,
    Radish,
    Turnip
}
```

还可以定义一个 `enum` 作为标志组合使用。以下声明为四季声明了一组标志。可以随意搭配季节组合，包括 `All` 值（包含所有季节）：

```
[Flags]
public enum Seasons
{
    None = 0,
    Summer = 1,
    Autumn = 2,
    Winter = 4,
    Spring = 8,
    All = Summer | Autumn | Winter | Spring
}
```

以下示例显示了前面两个枚举的声明：

```
var turnip = SomeRootVegetable.Turnip;

var spring = Seasons.Spring;
var startingOnEquinox = Seasons.Spring | Seasons.Autumn;
var theYear = Seasons.All;
```

可为 null 的类型

任何类型的变量都可以声明为“不可为 null”或“可为 null”。可为 null 的变量包含一个额外的 `null` 值，表示没有值。可为 null 的值类型(结构或枚举)由 `System.Nullable<T>` 表示。不可为 null 和可为 null 的引用类型都由基础引用类型表示。这种区别由编译器和某些库读取的元数据体现。当可为 null 的引用在没有先对照 `null` 检查其值的情况下取消引用时，编译器会发出警告。当对不可为 null 的引用分配了可能为 `null` 的值时，编译器也会发出警告。以下示例声明了“可为 null 的 int”，并将其初始化为 `null`。然后将值设置为 `5`。该例通过“可为 null 的字符串”演示了同一概念。有关详细信息，请参阅[可为 null 的值类型](#)和[可为 null 的引用类型](#)。

```
int? optionalInt = default;
optionalInt = 5;
string? optionalText = default;
optionalText = "Hello World.;"
```

元组

C# 支持[元组](#)，后者提供了简洁的语法来将多个数据元素分组成一个轻型数据结构*。通过声明 `(` 和 `)` 之间的成员的类型和名称来实例化元组，如下例所示：

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

元组为具有多个成员的数据结构提供了一种替代方法，且无需使用下一篇文章中介绍的构建基块。



程序构建基块

2021/5/7 • • [Edit Online](#)

上文中介绍的类型是使用以下构建基块生成的：[成员*](#)、[表达式和语句*](#)。

成员

`class` 的成员要么是静态成员，要么是实例成员。静态成员属于类，而实例成员则属于对象（类实例）。

以下列表概述了类可以包含的成员类型。

- **常量**：与类相关联的常量值
- **字段**：与类关联的变量
- **方法**：类可执行的操作
- **属性**：与读取和写入类的已命名属性相关联的操作
- **索引器**：与将类实例编入索引（像处理数组一样）相关联的操作
- **事件**：类可以生成的通知
- **运算符**：类支持的转换和表达式运算符
- **构造函数**：初始化类实例或类本身所需的操作
- **终结器**：永久放弃类实例之前执行的操作
- **类型**：类声明的嵌套类型

辅助功能

每个类成员都有关联的可访问性，用于控制能够访问成员的程序文本区域。可访问性有六种可能的形式。以下内容对访问修饰符进行了汇总。

- `public`：访问不受限制。
- `private`：访问仅限于此类。
- `protected`：访问仅限于此类或派生自此类的类。
- `internal`：仅可访问当前程序集（`.exe` 或 `.dll`）。
- `protected internal`：仅可访问此类、从此类中派生的类，或者同一程序集中的类。
- `private protected`：仅可访问此类或同一程序集中从此类中派生的类。

字段

字段 是与类或类实例相关联的变量。

使用静态修饰符声明的字段定义的是静态字段。静态字段只指明一个存储位置。无论创建多少个类实例，永远只有一个静态字段副本。

不使用静态修饰符声明的字段定义的是实例字段。每个类实例均包含相应类的所有实例字段的单独副本。

在以下示例中，每个 `Color` 类实例均包含 `R`、`G` 和 `B` 实例字段的单独副本，但只包含 `Black`、`White`、`Red`、`Green` 和 `Blue` 静态字段的一个副本：

```
public class Color
{
    public static readonly Color Black = new Color(0, 0, 0);
    public static readonly Color White = new Color(255, 255, 255);
    public static readonly Color Red = new Color(255, 0, 0);
    public static readonly Color Green = new Color(0, 255, 0);
    public static readonly Color Blue = new Color(0, 0, 255);

    public byte R;
    public byte G;
    public byte B;

    public Color(byte r, byte g, byte b)
    {
        R = r;
        G = g;
        B = b;
    }
}
```

如上面的示例所示，可以使用 `readonly` 修饰符声明 **只读字段**。只能在字段声明期间或在同一个类的构造函数中向只读字段赋值。

方法

方法是实现对象或类可执行的计算或操作的成员。静态方法是通过类进行访问。实例方法是通过类实例进行访问。

方法可能包含一个参数列表，这些参数表示传递给方法的值或变量引用。方法具有返回类型，它用于指定方法计算和返回的值的类型。如果方法未返回值，则它的返回类型为 `void`。

方法可能也包含一组类型参数，必须在调用方法时指定类型自变量，这一点与类型一样。与类型不同的是，通常可以根据方法调用的自变量推断出类型自变量，无需显式指定。

在声明方法的类中，方法的 **签名** 必须是唯一的。方法签名包含方法名称、类型参数数量及其参数的数量、修饰符和类型。方法签名不包含返回类型。

当方法主体是单个表达式时，可使用紧凑表达式格式定义方法，如下例中所示：

```
public override string ToString() => "This is an object";
```

参数

参数用于将值或变量引用传递给方法。方法参数从调用方法时指定的 **自变量** 中获取其实际值。有四类参数：值参数、引用参数、输出参数和参数数组。

值参数用于传递输入自变量。值参数对应于局部变量，从为其传递的自变量中获取初始值。修改值形参不会影响为其传递的实参。

可以指定默认值，从而省略相应的自变量，这样值参数就是可选的。

引用参数用于按引用传递自变量。为引用参数传递的自变量必须是一个带有明确值的变量。在方法执行期间，引用参数指出的存储位置与自变量相同。引用参数使用 `ref` 修饰符进行声明。下面的示例展示了如何使用 `ref` 参数。

```

static void Swap(ref int x, ref int y)
{
    int temp = x;
    x = y;
    y = temp;
}

public static void SwapExample()
{
    int i = 1, j = 2;
    Swap(ref i, ref j);
    Console.WriteLine($"{i} {j}"); // "2 1"
}

```

输出参数用于按引用传递自变量。输出参数与引用参数类似，不同之处在于，不要求向调用方提供的自变量显式赋值。输出参数使用 `out` 修饰符进行声明。下面的示例演示如何通过 C# 7 中引入的语法使用 `out` 参数。

```

static void Divide(int x, int y, out int result, out int remainder)
{
    result = x / y;
    remainder = x % y;
}

public static void OutUsage()
{
    Divide(10, 3, out int res, out int rem);
    Console.WriteLine($"{res} {rem}"); // "3 1"
}

```

参数数组允许向方法传递数量不定的自变量。参数数组使用 `params` 修饰符进行声明。参数数组只能是方法的最后一个参数，且参数数组的类型必须是一维数组类型。`System.Console` 类的 `Write` 和 `WriteLine` 方法是参数数组用法的典型示例。它们的声明方式如下。

```

public class Console
{
    public static void Write(string fmt, params object[] args) { }
    public static void WriteLine(string fmt, params object[] args) { }
    // ...
}

```

在使用参数数组的方法中，参数数组的行为与数组类型的常规参数完全相同。不过，在调用包含形参数组的方法时，要么可以传递形参数组类型的一个实参，要么可以传递形参数组的元素类型的任意数量实参。在后一种情况下，数组实例会自动创建，并初始化为包含给定的自变量。以下示例：

```

int x, y, z;
x = 3;
y = 4;
z = 5;
Console.WriteLine("x={0} y={1} z={2}", x, y, z);

```

等同于编写以下代码：

```
int x = 3, y = 4, z = 5;

string s = "x={0} y={1} z={2}";
object[] args = new object[3];
args[0] = x;
args[1] = y;
args[2] = z;
Console.WriteLine(s, args);
```

方法主体和局部变量

方法主体指定了在调用方法时执行的语句。

方法主体可以声明特定于方法调用的变量。此类变量称为 **局部变量**。局部变量声明指定了类型名称、变量名称以及可能的初始值。下面的示例声明了初始值为零的局部变量 **i** 和无初始值的局部变量 **j**。

```
class Squares
{
    public static void WriteSquares()
    {
        int i = 0;
        int j;
        while (i < 10)
        {
            j = i * i;
            Console.WriteLine($"{i} x {i} = {j}");
            i = i + 1;
        }
    }
}
```

C# 要求必须先 **明确赋值** 局部变量，然后才能获取其值。例如，如果上述 **i** 的声明未包含初始值，那么编译器会在后续使用 **i** 时报告错误，因为在后续使用时 **i** 不会在程序中得到明确赋值。

方法可以使用 **return** 语句将控制权返回给调用方。在返回 **void** 的方法中，**return** 语句无法指定表达式。在不返回 **void** 的方法中，**return** 语句必须包括用于计算返回值的表达式。

静态和实例方法

使用 **static** 修饰符声明的方法是静态方法。静态方法不对特定的实例起作用，只能直接访问静态成员。

未使用 **static** 修饰符声明的方法是实例方法。实例方法对特定的实例起作用，并能够访问静态和实例成员。其中调用实例方法的实例可以作为 **this** 显式访问。在静态方法中引用 **this** 会生成错误。

以下 **Entity** 类包含静态和实例成员。

```

class Entity
{
    static int s_nextSerialNo;
    int _serialNo;

    public Entity()
    {
        _serialNo = s_nextSerialNo++;
    }

    public int GetSerialNo()
    {
        return _serialNo;
    }

    public static int GetNextSerialNo()
    {
        return s_nextSerialNo;
    }

    public static void SetNextSerialNo(int value)
    {
        s_nextSerialNo = value;
    }
}

```

每个 `Entity` 实例均有一个序列号(很可能包含此处未显示的其他一些信息)。`Entity` 构造函数(类似于实例方法)将新实例初始化为包含下一个可用的序列号。由于构造函数是实例成员,因此可以访问 `_serialNo` 实例字段和 `s_nextSerialNo` 静态字段。

`GetNextSerialNo` 和 `SetNextSerialNo` 静态方法可以访问 `s_nextSerialNo` 静态字段,但如果直接访问 `_serialNo` 实例字段,则会生成错误。

下例显示了 `Entity` 类的用法。

```

Entity.SetNextSerialNo(1000);
Entity e1 = new Entity();
Entity e2 = new Entity();
Console.WriteLine(e1.GetSerialNo());           // Outputs "1000"
Console.WriteLine(e2.GetSerialNo());           // Outputs "1001"
Console.WriteLine(Entity.GetNextSerialNo());   // Outputs "1002"

```

`SetNextSerialNo` 和 `GetNextSerialNo` 静态方法在类中进行调用,而 `GetSerialNo` 实例方法则是在类实例中进行调用。

虚方法、重写方法和抽象方法

如果实例方法声明中有 `virtual` 修饰符,可以将实例方法称为“**虚方法**”。如果没有 `virtual` 修饰符,可以将实例方法称为“**非虚方法**”。

调用虚方法时,为其调用方法的实例的 **运行时类型** 决定了要调用的实际方法实现代码。调用非虚方法时,实例的 **编译时类型** 是决定性因素。

可以在派生类中 **重写虚方法**。如果实例方法声明中有 `override` 修饰符,那么实例方法可以重写签名相同的继承虚方法。虚方法声明引入了新方法。重写方法声明通过提供现有继承的虚方法的新实现,专门针对该方法。

抽象方法是没有实现代码的虚方法。抽象方法使用 `abstract` 修饰符进行声明,仅可在抽象类中使用。必须在所有非抽象派生类中重写抽象方法。

下面的示例声明了一个抽象类 `Expression`,用于表示表达式树节点;还声明了三个派生类(`Constant`、`VariableReference` 和 `Operation`),用于实现常量、变量引用和算术运算的表达式树节点。(该示例与表达式树

类型相似，但与它无关）。

```
public abstract class Expression
{
    public abstract double Evaluate(Dictionary<string, object> vars);
}

public class Constant : Expression
{
    double _value;

    public Constant(double value)
    {
        _value = value;
    }

    public override double Evaluate(Dictionary<string, object> vars)
    {
        return _value;
    }
}

public class VariableReference : Expression
{
    string _name;

    public VariableReference(string name)
    {
        _name = name;
    }

    public override double Evaluate(Dictionary<string, object> vars)
    {
        object value = vars[_name] ?? throw new Exception($"Unknown variable: {_name}");
        return Convert.ToDouble(value);
    }
}

public class Operation : Expression
{
    Expression _left;
    char _op;
    Expression _right;

    public Operation(Expression left, char op, Expression right)
    {
        _left = left;
        _op = op;
        _right = right;
    }

    public override double Evaluate(Dictionary<string, object> vars)
    {
        double x = _left.Evaluate(vars);
        double y = _right.Evaluate(vars);
        switch (_op)
        {
            case '+': return x + y;
            case '-': return x - y;
            case '*': return x * y;
            case '/': return x / y;

            default: throw new Exception("Unknown operator");
        }
    }
}
```

上面的四个类可用于进行算术表达式建模。例如，使用这些类的实例，可以按如下方式表示表达式 $x + 3$ 。

```
Expression e = new Operation(
    new VariableReference("x"),
    '+',
    new Constant(3));
```

调用 `Expression` 实例的 `Evaluate` 方法可以计算给定的表达式并生成 `double` 值。此方法需要使用自变量 `Dictionary`，其中包含变量名称（作为项键）和值（作为项值）。因为 `Evaluate` 是一个抽象方法，因此派生自 `Expression` 的非抽象类必须替代 `Evaluate`。

`Constant` 的 `Evaluate` 实现代码只返回存储的常量。`VariableReference` 实现代码查找字典中的变量名称，并返回结果值。`Operation` 实现代码先计算左右操作数（以递归方式调用其 `Evaluate` 方法），然后执行给定的算术运算。

以下程序使用 `Expression` 类根据不同的 `x` 和 `y` 值计算表达式 $x * (y + 2)$ 。

```
Expression e = new Operation(
    new VariableReference("x"),
    '*',
    new Operation(
        new VariableReference("y"),
        '+',
        new Constant(2)
    )
);
Dictionary<string, object> vars = new Dictionary<string, object>();
vars["x"] = 3;
vars["y"] = 5;
Console.WriteLine(e.Evaluate(vars)); // "21"
vars["x"] = 1.5;
vars["y"] = 9;
Console.WriteLine(e.Evaluate(vars)); // "16.5"
```

方法重载

借助方法 重载，同一类中可以有多个同名的方法，只要这些方法具有唯一签名即可。编译如何调用重载的方法时，编译器使用 重载决策 来确定要调用的特定方法。重载决策会查找与自变量匹配度最高的一种方法。如果找不到任何最佳匹配项，则会报告错误。下面的示例展示了重载决策的实际工作方式。`UsageExample` 方法中每个调用的注释指明了调用的方法。

```

class OverloadingExample
{
    static void F() => Console.WriteLine("F()");
    static void F(object x) => Console.WriteLine("F(object)");
    static void F(int x) => Console.WriteLine("F(int)");
    static void F(double x) => Console.WriteLine("F(double)");
    static void F<T>(T x) => Console.WriteLine("F<T>(T)");
    static void F(double x, double y) => Console.WriteLine("F(double, double)");

    public static void UsageExample()
    {
        F();                  // Invokes F()
        F(1);                // Invokes F(int)
        F(1.0);              // Invokes F(double)
        F("abc");             // Invokes F<string>(string)
        F((double)1);         // Invokes F(double)
        F((object)1);         // Invokes F(object)
        F<int>(1);            // Invokes F<int>(int)
        F(1, 1);              // Invokes F(double, double)
    }
}

```

如示例所示，可将自变量显式转换成确切的参数类型和类型自变量，随时选择特定的方法。

其他函数成员

包含可执行代码的成员统称为类的 **函数成员**。上一部分介绍了作为主要函数成员类型的方法。此部分将介绍 C# 支持的其他类型函数成员：构造函数、属性、索引器、事件、运算符和终结器。

下面的示例展示了 `MyList<T>` 泛型类，用于实现对象的可扩充列表。此类包含最常见类型函数成员的多个示例。

```

public class MyList<T>
{
    const int DefaultCapacity = 4;

    T[] _items;
    int _count;

    public MyList(int capacity = DefaultCapacity)
    {
        _items = new T[capacity];
    }

    public int Count => _count;

    public int Capacity
    {
        get => _items.Length;
        set
        {
            if (value < _count) value = _count;
            if (value != _items.Length)
            {
                T[] newItems = new T[value];
                Array.Copy(_items, 0, newItems, 0, _count);
                _items = newItems;
            }
        }
    }

    public T this[int index]
    {
        get => _items[index];
    }
}

```

```

        set
    {
        _items[index] = value;
        OnChanged();
    }
}

public void Add(T item)
{
    if (_count == Capacity) Capacity = _count * 2;
    _items[_count] = item;
    _count++;
    OnChanged();
}
protected virtual void OnChanged() =>
    Changed?.Invoke(this, EventArgs.Empty);

public override bool Equals(object other) =>
    Equals(this, other as MyList<T>);

static bool Equals(MyList<T> a, MyList<T> b)
{
    if (Object.ReferenceEquals(a, null)) return Object.ReferenceEquals(b, null);
    if (Object.ReferenceEquals(b, null) || a._count != b._count)
        return false;
    for (int i = 0; i < a._count; i++)
    {
        if (!object.Equals(a._items[i], b._items[i]))
        {
            return false;
        }
    }
    return true;
}

public event EventHandler Changed;

public static bool operator ==(MyList<T> a, MyList<T> b) =>
    Equals(a, b);

public static bool operator !=(MyList<T> a, MyList<T> b) =>
    !Equals(a, b);
}

```

构造函数

C# 支持实例和静态构造函数。实例构造函数是实现初始化类实例所需执行的操作的成员。静态构造函数是实现在首次加载类时初始化类本身所需执行的操作的成员。

构造函数的声明方式与方法一样，都没有返回类型，且与所含类同名。如果构造函数声明包含 `static` 修饰符，则声明的是静态构造函数。否则，声明的是实例构造函数。

实例构造函数可重载并且可具有可选参数。例如，`MyList<T>` 类声明一个具有单个可选 `int` 参数的实例构造函数。实例构造函数使用 `new` 运算符进行调用。下面的语句使用包含和不包含可选自变量的 `MyList` 类构造函数来分配两个 `MyList<string>` 实例。

```

MyList<string> list1 = new MyList<string>();
MyList<string> list2 = new MyList<string>(10);

```

与其他成员不同，实例构造函数不会被继承。类中只能包含实际上已在该类中声明的实例构造函数。如果没有为类提供实例构造函数，则会自动提供不含参数的空实例构造函数。

“属性”

属性是字段的自然扩展。两者都是包含关联类型的已命名成员，用于访问字段和属性的语法也是一样的。不

过，与字段不同的是，属性不指明存储位置。相反，属性包含访问器，用于指定在读取或写入属性值时执行的语句。

属性的声明方式与字段相似，区别是属性声明以在分隔符 `{` 和 `}` 之间写入的 `get` 访问器或 `set` 访问器结束，而不是以分号结束。同时具有 `get` 访问器和 `set` 访问器的属性是“读写属性”。只有 `get` 访问器的属性是“只读属性”。只有 `set` 访问器的属性是“只写属性”。

`get` 访问器对应于包含属性类型的返回值的无参数方法。`set` 访问器对应于包含一个名为 `value` 的参数但不含返回类型的方法。`get` 访问器会计算属性的值。`set` 访问器会为属性提供新值。当属性是赋值的目标，或者是 `++` 或 `--` 的操作数时，会调用 `set` 访问器。在引用了属性的其他情况下，会调用 `get` 访问器。

`MyList<T>` 类声明以下两个属性：`Count` 和 `Capacity`（分别为只读和读写）。以下示例代码展示了如何使用这些属性：

```
MyList<string> names = new MyList<string>();
names.Capacity = 100;    // Invokes set accessor
int i = names.Count;    // Invokes get accessor
int j = names.Capacity; // Invokes get accessor
```

类似于字段和方法，C# 支持实例属性和静态属性。静态属性使用静态修饰符进行声明，而实例属性则不使用静态修饰符进行声明。

属性的访问器可以是虚的。如果属性声明包含 `virtual`、`abstract` 或 `override` 修饰符，则适用于属性的访问器。

索引器

借助 索引器 成员，可以将对象编入索引（像处理数组一样）。索引器的声明方式与属性类似，不同之处在于，索引器成员名称格式为 `this` 后跟在分隔符 `[` 和 `]` 内写入的参数列表。这些参数在索引器的访问器中可用。类似于属性，索引器分为读写、只读和只写索引器，且索引器的访问器可以是虚的。

`MyList<T>` 类声明一个需要使用 `int` 参数的读写索引器。借助索引器，可以使用 `int` 值将 `MyList<T>` 实例编入索引。例如：

```
MyList<string> names = new MyList<string>();
names.Add("Liz");
names.Add("Martha");
names.Add("Beth");
for (int i = 0; i < names.Count; i++)
{
    string s = names[i];
    names[i] = s.ToUpper();
}
```

索引器可被重载。一个类可声明多个索引器，只要其参数的数量或类型不同即可。

事件

借助 事件 成员，类或对象可以提供通知。事件的声明方式与字段类似，区别是事件声明包括 `event` 关键字，且类型必须是委托类型。

在声明事件成员的类中，事件的行为与委托类型的字段完全相同（前提是事件不是抽象的，且不声明访问器）。字段存储对委托的引用，委托表示已添加到事件的事件处理程序。如果没有任何事件处理程序，则字段为 `null`。

`MyList<T>` 类声明一个 `Changed` 事件成员，指明已向列表添加了新项。`Changed` 事件由 `OnChanged` 虚方法引发，此方法会先检查事件是否是 `null`（即不含任何处理程序）。引发事件的概念恰恰等同于调用由事件表示的委托。不存在用于引发事件的特殊语言构造。

客户端通过 事件处理程序 响应事件。使用 `+=` 和 `-=` 运算符分别可以附加和删除事件处理程序。下面的示例

展示了如何向 `MyList<string>` 的 `Changed` 事件附加事件处理程序。

```
class EventExample
{
    static int s_changeCount;

    static void ListChanged(object sender, EventArgs e)
    {
        s_changeCount++;
    }

    public static void Usage()
    {
        var names = new MyList<string>();
        names.Changed += new EventHandler(ListChanged);
        names.Add("Liz");
        names.Add("Martha");
        names.Add("Beth");
        Console.WriteLine(s_changeCount); // "3"
    }
}
```

对于需要控制事件的基础存储的高级方案，事件声明可以显式提供 `add` 和 `remove` 访问器，这与属性的 `set` 访问器类似。

运算符

运算符是定义向类实例应用特定表达式运算符的含义的成员。可以定义三种类型的运算符：一元运算符、二元运算符和转换运算符。所有运算符都必须声明为 `public` 和 `static`。

`MyList<T>` 类会声明两个运算符：`operator ==` 和 `operator !=`。对于向 `MyList` 实例应用这些运算符的表达式来说，这些重写的运算符向它们赋予了新的含义。具体而言，这些运算符定义的是两个 `MyList<T>` 实例的相等性（使用其 `Equals` 方法比较所包含的每个对象）。下面的示例展示了如何使用 `==` 运算符比较两个 `MyList<int>` 实例。

```
MyList<int> a = new MyList<int>();
a.Add(1);
a.Add(2);
MyList<int> b = new MyList<int>();
b.Add(1);
b.Add(2);
Console.WriteLine(a == b); // Outputs "True"
b.Add(3);
Console.WriteLine(a == b); // Outputs "False"
```

第一个 `Console.WriteLine` 输出 `True`，因为两个列表包含的对象不仅数量相同，而且值和顺序也相同。如果 `MyList<T>` 未定义 `operator ==`，那么第一个 `Console.WriteLine` 会输出 `False`，因为 `a` 和 `b` 引用不同的 `MyList<int>` 实例。

终结器

终结器是实现完成类实例所需的操作的成员。通常，需要使用终结器来释放非托管资源。终结器既不能包含参数和可访问性修饰符，也不能进行显式调用。实例的终结器在垃圾回收期间自动调用。有关详细信息，请参阅[终结器一文](#)。

垃圾回收器在决定何时收集对象和运行终结器时有很大自由度。具体而言，终结器的调用时间具有不确定性，可以在任意线程上执行终结器。因为这样或那样的原因，只有在没有其他可行的解决方案时，类才能实现终结器。

处理对象析构的更好方法是使用 `using` 语句。

表达式

表达式是在 操作数 和 运算符 的基础之上构造而成。表达式的运算符指明了向操作数应用的运算。运算符的示例包括 `+`、`-`、`*`、`/` 和 `new`。操作数的示例包括文本、字段、局部变量和表达式。

如果某个表达式包含多个运算符，则运算符的优先顺序控制各个运算符的计算顺序。例如，表达式 `x + y * z` 相当于计算 `x + (y * z)`，因为 `*` 运算符的优先级高于 `+` 运算符。

如果操作数两边的两个运算符的优先级相同，那么运算符的 结合性 决定了运算的执行顺序：

- 除了赋值运算符和 `null` 合并运算符之外，所有二元运算符均为左结合运算符，即从左向右执行运算。例如，`x + y + z` 将计算为 `(x + y) + z`。
- 赋值运算符、`null` 合并 `??` 和 `??=` 运算符和条件运算符 `?:` 为右结合运算符，即从右向左执行运算。例如，`x = y = z` 将计算为 `x = (y = z)`。

可以使用括号控制优先级和结合性。例如，`x + y * z` 先计算 `y` 乘 `z`，并将结果与 `x` 相加，而 `(x + y) * z` 则先计算 `x` 加 `y`，然后将结果与 `z` 相乘。

大部分运算符可 [重载](#)。借助运算符重载，可以为一个或两个操作数为用户定义类或结构类型的运算指定用户定义运算符实现代码。

C# 提供多个运算符用于执行 [算术](#)、[逻辑](#)、[按位和移位](#) 运算以及 [相等](#) 和 [排序](#) 比较。

要了解按优先级排序的完整 C# 运算符列表，请参阅 [C# 运算符](#)。

语句

程序操作使用 语句 进行表示。C# 支持几种不同的语句，其中许多语句是从嵌入语句的角度来定义的。

- 使用 `代码块`，可以在允许编写一个语句的上下文中编写多个语句。代码块是由一系列在分隔符 `{` 和 `}` 内编写的语句组成。
- `声明语句` 用于声明局部变量和常量。
- `表达式语句` 用于计算表达式。可用作语句的表达式包括方法调用、使用 `new` 运算符的对象分配、使用 `=` 和复合赋值运算符的赋值、使用 `++` 和 `--` 运算符和 `await` 表达式的递增和递减运算。
- `选择语句` 用于根据一些表达式的值从多个可能的语句中选择一个以供执行。此组包含 `if` 和 `switch` 语句。
- `迭代语句` 用于重复执行嵌入语句。此组包含 `while`、`do`、`for` 和 `foreach` 语句。
- `跳转语句` 用于转移控制权。此组包含 `break`、`continue`、`goto`、`throw`、`return` 和 `yield` 语句。
- `try ... catch` 语句用于捕获在代码块执行期间发生的异常，`try ... finally` 语句用于指定始终执行的最终代码，无论异常发生与否。
- `checked` 和 `unchecked` 语句用于控制整型类型算术运算和转换的溢出检查上下文。
- `lock` 语句用于获取给定对象的相互排斥锁定，执行语句，然后解除锁定。
- `using` 语句用于获取资源，执行语句，然后释放资源。

下面列出了可使用的语句类型：

- 局部变量声明。
- 局部常量声明。
- 表达式语句。
- `if` 语句。
- `switch` 语句。
- `while` 语句。
- `do` 语句。
- `for` 语句。

- `foreach` 语句。
- `break` 语句。
- `continue` 语句。
- `goto` 语句。
- `return` 语句。
- `yield` 语句。
- `throw` 和 `try` 语句。
- `checked` 和 `unchecked` 语句。
- `lock` 语句。
- `using` 语句。



主要语言区域

2021/5/7 • [Edit Online](#)

数组、集合和 LINQ

C# 和 .NET 提供了许多不同的集合类型。数组包含由语言定义的语法。泛型集合类型列在 `System.Collections.Generic` 命名空间中。专用集合包括 `System.Span<T>` (用于访问堆栈帧上的连续内存), 以及 `System.Memory<T>` (用于访问托管堆上的连续内存)。所有集合(包括数组、`Span<T>` 和 `Memory<T>`)都遵循一种统一的迭代原则。使用 `System.Collections.Generic.IEnumerable<T>` 接口。这种统一的原则意味着任何集合类型都可以与 LINQ 查询或其他算法一起使用。你可以使用 `IEnumerable<T>` 编写方法, 这些算法适用于任何集合。

数组

数组是一种数据结构*, 其中包含许多通过计算索引访问的变量。数组中的变量(亦称为数组的“元素”)均为同一种类型。我们将这种类型称为数组的“元素类型”。

数组类型是引用类型, 声明数组变量只是为引用数组实例预留空间。实际的数组实例是在运行时使用 `new` 运算符动态创建而成。`new` 运算符指定了新数组实例的长度, 然后在此实例的生存期内固定使用这个长度。数组元素的索引介于 `0` 到 `Length - 1` 之间。`new` 运算符自动将数组元素初始化为其默认值(例如, 所有数值类型的默认值为 0, 所有引用类型的默认值为 `null`)。

以下示例创建 `int` 元素数组, 初始化此数组, 然后打印此数组的内容。

```
int[] a = new int[10];
for (int i = 0; i < a.Length; i++)
{
    a[i] = i * i;
}
for (int i = 0; i < a.Length; i++)
{
    Console.WriteLine($"a[{i}] = {a[i]}");
}
```

此示例创建并在“一维数组”上进行操作。C# 还支持多维数组。数组类型的维数(亦称为数组类型的秩)是 1 与数组类型方括号内的逗号数量相加的结果。以下示例分别分配一维、二维、三维数组。

```
int[] a1 = new int[10];
int[,] a2 = new int[10, 5];
int[, ,] a3 = new int[10, 5, 2];
```

`a1` 数组包含 10 个元素, `a2` 数组包含 50 个元素 (10×5), `a3` 数组包含 100 个元素 ($10 \times 5 \times 2$)。数组的元素类型可以是任意类型(包括数组类型)。包含数组类型元素的数组有时称为“交错数组”, 因为元素数组的长度不必全都一样。以下示例分配由 `int` 数组构成的数组:

```
int[][] a = new int[3][];
a[0] = new int[10];
a[1] = new int[5];
a[2] = new int[20];
```

第一行创建包含三个元素的数组, 每个元素都是 `int[]` 类型, 并且初始值均为 `null`。接下来的代码行将这三个元素初始化为引用长度不同的各个数组实例。

通过 `new` 运算符，可以使用“数组初始值设定项”(在分隔符 `{` 和 `}` 内编写的表达式列表)指定数组元素的初始值。以下示例分配 `int[]`，并将其初始化为包含三个元素。

```
int[] a = new int[] { 1, 2, 3 };
```

可从 `{` 和 `}` 内的表达式数量推断出数组的长度。数组初始化可以进一步缩短，这样就不用重新声明数组类型了。

```
int[] a = { 1, 2, 3 };
```

以上两个示例等同于以下代码：

```
int[] t = new int[3];
t[0] = 1;
t[1] = 2;
t[2] = 3;
int[] a = t;
```

`foreach` 语句可用于枚举任何集合的元素。以下代码从前一个示例中枚举数组：

```
foreach (int item in a)
{
    Console.WriteLine(item);
}
```

`foreach` 语句使用 `IEnumerable<T>` 接口，因此适用于任何集合。

字符串内插

C# [字符串内插](#)使你能够通过定义表达式(其结果放置在格式字符串中)来设置字符串格式。例如，以下示例从一组天气数据显示给定日期的温度：

```
Console.WriteLine($"The low and high temperature on {weatherData.Date:MM-DD-YYYY}");
Console.WriteLine($"      was {weatherData.LowTemp} and {weatherData.HighTemp}.");
// Output (similar to):
// The low and high temperature on 08-11-2020
//      was 5 and 30.
```

内插字符串通过 `$` 标记来声明。字符串插内插计算 `{` 和 `}` 之间的表达式，然后将结果转换为 `string`，并将括号内的文本替换为表达式的字符串结果。第一个表达式 (`{weatherData.Date:MM-DD-YYYY}`) 中的 `:` 指定格式字符串。在前一个示例中，这指定日期应以“MM-DD-YYYY”格式显示。

模式匹配

C# 语言提供[模式匹配](#)表达式来查询对象的状态并基于该状态执行代码。你可以检查属性和字段的类型和值，以确定要执行的操作。`switch` 表达式是模式匹配的主要表达式。

委托和 Lambda 表达式

[委托类型](#)表示对具有特定参数列表和返回类型的方法的引用。通过委托，可以将方法视为可分配给变量并可作为参数传递的实体。委托还类似于其他一些语言中存在的“函数指针”概念。与函数指针不同，委托是面向对象且类型安全的。

下面的示例声明并使用 `Function` 委托类型。

```
delegate double Function(double x);

class Multiplier
{
    double _factor;

    public Multiplier(double factor) => _factor = factor;

    public double Multiply(double x) => x * _factor;
}

class DelegateExample
{
    static double[] Apply(double[] a, Function f)
    {
        var result = new double[a.Length];
        for (int i = 0; i < a.Length; i++) result[i] = f(a[i]);
        return result;
    }

    public static void Main()
    {
        double[] a = { 0.0, 0.5, 1.0 };
        double[] squares = Apply(a, (x) => x * x);
        double[] sines = Apply(a, Math.Sin);
        Multiplier m = new Multiplier(2.0);
        double[] doubles = Apply(a, m.Multiply);
    }
}
```

`Function` 委托类型实例可以引用需要使用 `double` 自变量并返回 `double` 值的方法。`Apply` 方法将给定的 `Function` 应用于 `double[]` 的元素，从而返回包含结果的 `double[]`。在 `Main` 方法中，`Apply` 用于向 `double[]` 应用三个不同的函数。

委托可以引用静态方法（如上面示例中的 `Square` 或 `Math.Sin`）或实例方法（如上面示例中的 `m.Multiply`）。引用实例方法的委托还会引用特定对象，通过委托调用实例方法时，该对象会变成调用中的 `this`。

还可以使用匿名函数创建委托，这些函数是在声明时创建的“内联方法”。匿名函数可以查看周围方法的局部变量。以下示例不创建类：

```
double[] doubles = Apply(a, (double x) => x * 2.0);
```

委托不知道或不在意其所引用的方法的类。重要的是，引用的方法具有与委托相同的参数和返回类型。

async/await

C# 支持含两个关键字的异步程序：`async` 和 `await`。将 `async` 修饰符添加到方法声明中，以声明这是异步方法。`await` 运算符通知编译器异步等待结果完成。控件返回给调用方，该方法返回一个管理异步工作状态的结构。结构通常是 `System.Threading.Tasks.Task<TResult>`，但可以是任何支持 awaite 模式的类型。这些功能使你能够编写这样的代码：以其同步对应项的形式读取，但以异步方式执行。例如，以下代码会下载 Microsoft Docs 的主页：

```

public async Task<int> RetrieveDocsHomePage()
{
    var client = new HttpClient();
    byte[] content = await client.GetByteArrayAsync("https://docs.microsoft.com/");

    Console.WriteLine($"{nameof(RetrieveDocsHomePage)}: Finished downloading.");
    return content.Length;
}

```

这一小型示例显示了异步编程的主要功能：

- 方法声明包含 `async` 修饰符。
- 方法 `await` 的主体是 `GetByteArrayAsync` 方法的返回。
- `return` 语句中指定的类型与方法的 `Task<T>` 声明中的类型参数匹配。(返回 `Task` 的方法将使用不带任何参数的 `return` 语句)。

属性

C# 程序中的类型、成员和其他实体支持使用修饰符来控制其行为的某些方面。例如，方法的可访问性是由 `public`、`protected`、`internal` 和 `private` 修饰符控制。C# 整合了这种能力，以便可以将用户定义类型的声明性信息附加到程序实体，并在运行时检索此类信息。程序通过定义和使用特性来指定此类额外的声明性信息。

以下示例声明了 `HelpAttribute` 特性，可将其附加到程序实体，以提供指向关联文档的链接。

```

public class HelpAttribute : Attribute
{
    string _url;
    string _topic;

    public HelpAttribute(string url) => _url = url;

    public string Url => _url;

    public string Topic
    {
        get => _topic;
        set => _topic = value;
    }
}

```

所有特性类都派生自 .NET 库提供的 `Attribute` 基类。特性的应用方式为，在相关声明前的方括号内指定特性的名称以及任意自变量。如果特性的名称以 `Attribute` 结尾，那么可以在引用特性时省略这部分名称。例如，可按如下方法使用 `HelpAttribute`。

```

[Help("https://docs.microsoft.com/dotnet/csharp/tour-of-csharp/features")]
public class Widget
{
    [Help("https://docs.microsoft.com/dotnet/csharp/tour-of-csharp/features",
        Topic = "Display")]
    public void Display(string text) { }
}

```

此示例将 `HelpAttribute` 附加到 `Widget` 类。还向此类中的 `Display` 方法附加了另一个 `HelpAttribute`。特性类的公共构造函数控制了将特性附加到程序实体时必须提供的信息。可以通过引用特性类的公共读写属性(如上面示例对 `Topic` 属性的引用)，提供其他信息。

可以在运行时使用反射来读取和操纵特性定义的元数据。如果使用这种方法请求获取特定特性，便会调用特性类的构造函数(在程序源中提供信息)。返回生成的特性实例。如果是通过属性提供其他信息，那么在特性实例

返回前，这些属性会设置为给定值。

下面的代码示例展示了如何获取与 `Widget` 类及其 `Display` 方法相关联的 `HelpAttribute` 实例。

```
Type widgetType = typeof(Widget);

object[] widgetClassAttributes = widgetType.GetCustomAttributes(typeof(HelpAttribute), false);

if (widgetClassAttributes.Length > 0)
{
    HelpAttribute attr = (HelpAttribute)widgetClassAttributes[0];
    Console.WriteLine($"Widget class help URL : {attr.Url} - Related topic : {attr.Topic}");
}

System.Reflection.MethodInfo displayMethod = widgetType.GetMethod(nameof(Widget.Display));

object[] displayMethodAttributes = displayMethod.GetCustomAttributes(typeof(HelpAttribute), false);

if (displayMethodAttributes.Length > 0)
{
    HelpAttribute attr = (HelpAttribute)displayMethodAttributes[0];
    Console.WriteLine($"Display method help URL : {attr.Url} - Related topic : {attr.Topic}");
}
```

了解详细信息

可以通过试用其中一个[教程](#)来了解更多关于 C# 的内容。



C# 简介

2021/5/7 • [Edit Online](#)

欢迎学习 C# 教程简介。这些课程先介绍你可在浏览器中运行的交互式代码。在开始这些互动课程之前，可在[C# 101 视频系列](#)中了解 C# 的基础知识。

开始的几个课程通过小篇幅的代码片段介绍了 C# 概念。读者将了解 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。这些全都是交互式课程，读者可以在几分钟内编写并运行代码。无需事先了解编程或 C# 语言，即可学习头几个课程。

可在不同的环境中尝试这些教程。你将学习的概念是相同的。区别在于你更喜欢哪种体验：

- [在浏览器中的文档平台上](#)：该体验在文档页面中嵌入了一个可运行的 C# 代码窗口。你在浏览器中编写和执行 C# 代码。
- [在 Microsoft Learn 体验中](#)。此学习路径包含多个模块，其中介绍了 C# 的基本知识。
- [在 Binder 上的 Jupyter 中](#)。可在 Binder 上的 Jupyter 笔记本中实验 C# 代码。
- [在本地计算机上](#)。联机浏览后，可在计算机上[下载 .NET Core 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。每个教程结束时，可以决定是继续在线学习还是在自己的计算机上学习下一节课。可以访问相关链接设置自己的环境并继续在自己的计算机上学习下一个教程。

Hello World

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。还可使用 Microsoft Learn 上的路径或 [Binder 上的 Jupyter](#)。

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。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](#) 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本知识，这是程序决定选择不同操作的基本依据。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。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。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上面列出的课程。

101 Linq 示例

此示例需要 `dotnet-try` 全局工具。安装该工具并克隆 `try-samples` 存储库后，可以通过一组以交互方式运行的

101示例来了解语言集成查询(LINQ)。你可以探索用于查询、浏览和转换数据序列的各种方法。

设置本地环境

2021/3/5 • • [Edit Online](#)

在计算机上演练教程的第一步是设置开发环境。选择以下备选方案之一：

- 若要使用 .NET CLI 以及所选的文本或代码编辑器，请参阅 .NET 教程 [Hello World 10 分钟入门](#)。本教程介绍了如何在 Windows、Linux 或 macOS 上创建开发环境。
- 若要使用 .NET CLI 和 Visual Studio Code，请安装 [.NET SDK](#) 和 [Visual Studio Code](#)。
- 若要使用 Visual Studio 2019，请参阅[教程：在 Visual Studio 中创建简单的 C# 控制台应用](#)。

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。
- “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本知识，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。

处理 C# 中的整数和浮点数

2021/5/7 • [Edit Online](#)

本教程以交互方式介绍了 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。

先决条件

本教程要求安装一台虚拟机，以用于本地开发。在 Windows、Linux 或 macOS 上，可以使用 .NET CLI 创建、生成和运行应用程序。在 Mac 或 Windows 上，可以使用 Visual Studio 2019。有关设置说明，请参阅[设置本地环境](#)。

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：

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

在常用编辑器中，打开 Program.cs，并将文件内容替换为以下代码：

```
using System;

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：

```
// 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。

TIP

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：

```
using System;

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：

```
//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：

```
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：

```
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`。

```
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：

```
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：

```
using System;
```

```
// 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：

```
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：

```
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：

```
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 `float` 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：

```
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：

```
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：

```
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：

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

众所周知，`0.3` 循环小数与 `1/3` 并不完全相等。

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：

```
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：

```
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。

NOTE

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](#)

通过分支和循环语句了解条件逻辑

2021/5/7 • [Edit Online](#)

本教程介绍了如何编写代码，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。

先决条件

本教程要求安装一台虚拟机，以用于本地开发。在 Windows、Linux 或 macOS 上，可以使用 .NET CLI 创建、生成和运行应用程序。在 Mac 和 Windows 上，可以使用 Visual Studio 2019。有关创建说明，请参阅[创建本地环境](#)。

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：

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

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：

```
using System;

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。你应在控制台上看到以下消息：“The answer is greater than 10. (答案大于 10)”。修改 `b` 的声明，让总和小于 10：

```
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。

TIP

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):

```
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。

IMPORTANT

`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:

```
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");
}
```

TIP

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:

```
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:

```

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：

```

using System;

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：

```
//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：

```
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。

IMPORTANT

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：

```
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：

```
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：

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

另一个循环可以生成列：

```
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：

```
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：

- [If 和 else 语句](#)
- [While 语句](#)
- [Do 语句](#)
- [For 语句](#)

了解如何使用泛型列表类型管理数据集合

2021/5/7 • [Edit Online](#)

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。在 Windows、Linux 或 macOS 上，可以使用 .NET CLI 创建、生成和运行应用程序。在 Mac 和 Windows 上，可以使用 Visual Studio 2019。有关设置说明，请参阅[设置本地环境](#)。

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`。

在常用编辑器中，打开 `Program.cs`，并将现有代码替换为以下代码：

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

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：

```
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：

```
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：

```
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：

```
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：

```
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：

```

using System;
using System.Collections.Generic;

WorkWithString();

void WorkWithString()
{
    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：

```
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：

```
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。

TIP

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

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# 9.0 中的新增功能

2021/5/7 • [Edit Online](#)

C# 9.0 向 C# 语言添加了以下功能和增强功能：

- [记录](#)
- [仅限 `Init` 的资源库](#)
- [顶级语句](#)
- [模式匹配增强功能](#)
- [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](#)
- [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](#)
- [支持代码生成器](#)
 - [模块初始值设定项](#)
 - [分部方法的新功能](#)

.NET 5 支持 C# 9.0。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 .NET SDK。

记录类型

C# 9.0 引入了记录类型。可使用 `record` 关键字定义一个引用类型，用来提供用于封装数据的内置功能。通过使用位置参数或标准属性语法，可以创建具有不可变属性的记录类型：

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

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

此外，还可以创建具有可变属性和字段的记录类型：

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

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

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

可使用[结构类型](#)来设计以数据为中心的类型，这些类型提供值相等性，并且很少或没有任何行为。但对于相对较大的数据模型，结构类型有一些缺点：

- 它们不支持继承。
- 它们在确定值相等性时效率较低。对于值类型，[ValueType.Equals](#) 方法使用反射来查找所有字段。对于记录，编译器将生成 [Equals](#) 方法。实际上，记录中的值相等性实现的速度明显更快。
- 在某些情况下，它们会占用更多内存，因为每个实例都有所有数据的完整副本。记录类型是[引用类型](#)，因此，记录实例只包含对数据的引用。

属性定义的位置语法

在创建实例时，可以使用位置参数来声明记录的属性，并初始化属性值：

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

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

当你为属性定义使用位置语法时，编译器将创建以下内容：

- 为记录声明中提供的每个位置参数提供一个公共的 [init-only](#) 自动实现的属性。[init-only](#) 属性只能在构造函数中或使用属性初始值设定项来设置。
- 主构造函数，它的参数与记录声明上的位置参数匹配。
- 一个 [Deconstruct](#) 方法，对记录声明中提供的每个位置参数都有一个 [out](#) 参数。

有关详细信息，请参阅有关记录的 C# 语言参考文章中的[位置语法](#)。

不可变性

记录类型不一定是不可变的。可以用 [set](#) 访问器和非 [readonly](#) 的字段来声明属性。虽然记录可以是可变的，但它们使创建不可变的数据模型变得更容易。使用位置语法创建的属性是不可变的。

如果希望以数据为中心的类型是线程安全的，或者哈希表中的哈希代码保持不变，那么不可变性很有用。它可以防止在通过引用方法传递参数而该方法意外更改参数值时发生的 bug。

记录类型特有的功能是由编译器合成的方法实现的，这些方法都不会通过修改对象状态来影响不可变性。

值相等性

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。对于其他引用类型，相等是指标识。也就是说，如果一个引用类型的两个变量引用同一个对象，那么这两个变量是相等的。

下面的示例说明了记录类型的值相等性：

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

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

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

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

在 `class` 类型中，可以手动替代相等性方法和运算符以实现值相等性，但开发和测试这种代码会非常耗时，而且容易出错。内置此功能可防止在添加或更改属性或字段时忘记更新自定义替代代码导致的 bug。

有关详细信息，请参阅有关记录的 C# 语言参考文章中的[值相等性](#)。

非破坏性变化

如果需要改变记录实例的不可变属性，可以使用 `with` 表达式来实现非破坏性变化。`with` 表达式创建一个新的记录实例，该实例是现有记录实例的一个副本，修改了指定属性和字段。使用[对象初始值设定项语法](#)来指定要更改的值，如以下示例中所示：

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

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

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

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

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

有关详细信息，请参阅有关记录的 C# 语言参考文章中的[非破坏性变化](#)。

用于显示的内置格式设置

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

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

对于引用类型，将显示属性所引用的对象的类型名称，而不是属性值。在下面的示例中，数组是一个引用类型，因此显示的是 `System.String[]`，而不是实际的数组元素值：

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

有关详细信息，请参阅有关记录的 C# 语言参考文章中的[内置格式](#)。

继承

一条记录可以从另一条记录继承。但是，记录不能从类继承，类也不能从记录继承。

下面的示例说明了具有位置属性语法的继承：

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

要使两个记录变量相等，运行时类型必须相等。包含变量的类型可能不同。下面的代码示例中说明了这一点：

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

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

在本示例中，所有实例都具有相同的属性和相同的属性值。尽管两者都是 `Person` 类型变量，但 `student == teacher` 会返回 `False`。尽管一个是 `Person` 变量，另一个是 `Student` 变量，但 `student == student2` 会返回 `True`。

派生类型和基类型的所有公共属性和字段都包含在 `ToString` 输出中，如以下示例所示：

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

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

有关详细信息，请参阅有关记录的 C# 语言参考文章中的[继承](#)。

仅限 init 的资源库

仅限 init 的资源库提供一致的语法来初始化对象的成员。属性初始值设定项可明确哪个值正在设置哪个属性。缺点是这些属性必须是可设置的。从 C# 9.0 开始，可为属性和索引器创建 `init` 访问器，而不是 `set` 访问器。调用方可使用属性初始化表达式语法在创建表达式中设置这些值，但构造完成后，这些属性将变为只读。仅限 init 的资源库提供了一个窗口用来更改状态。构造阶段结束时，该窗口关闭。在完成所有初始化（包括属性初始化表达式和 with 表达式）之后，构造阶段实际上就结束了。

可在编写的任何类型中声明仅限 `init` 的资源库。例如，以下结构定义了天气观察结构：

```
public struct WeatherObservation
{
    public DateTime RecordedAt { get; init; }
    public decimal TemperatureInCelsius { get; init; }
    public decimal PressureInMillibars { get; init; }

    public override string ToString() =>
        $"At {RecordedAt:h:mm tt} on {RecordedAt:M/d/yyyy}: " +
        $"Temp = {TemperatureInCelsius}, with {PressureInMillibars} pressure";
}
```

调用方可使用属性初始化表达式语法来设置值，同时仍保留不变性：

```
var now = new WeatherObservation
{
    RecordedAt = DateTime.Now,
    TemperatureInCelsius = 20,
    PressureInMillibars = 998.0m
};
```

初始化后尝试更改观察值会导致编译器错误：

```
// Error! CS8852.
now.TemperatureInCelsius = 18;
```

对于从派生类设置基类属性，仅限 init 的资源库很有用。它们还可通过基类中的帮助程序来设置派生属性。位置记录使用仅限 init 的资源库声明属性。这些设置器可在 with 表达式中使用。可为定义的任何 `class`、`struct` 或 `record` 声明仅限 init 的资源库。

有关详细信息，请查看 [init\(C# 参考\)](#)。

顶级语句

顶级语句从许多应用程序中删除了不必要的流程。请考虑规范的“Hello World!”程序：

```
using System;

namespace HelloWorld
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Hello World!");
        }
    }
}
```

只有一行代码执行所有操作。借助顶级语句，可使用 `using` 指令和执行操作的一行替换所有样本：

```
using System;

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

如果需要单行程序，可删除 `using` 指令，并使用完全限定的类型名称：

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

应用程序中只有一个文件可使用顶级语句。如果编译器在多个源文件中找到顶级语句，则是错误的。如果将顶级语句与声明的程序入口点方法（通常为 `Main` 方法）结合使用，也会出现错误。从某种意义上讲，可认为一个文件包含通常位于 `Program` 类的 `Main` 方法中的语句。

此功能最常见的用途之一是创建材料。C# 初级开发人员可以用一两行代码 编写规范的“Hello World!”。不需要额外的工作。不过，经验丰富的开发人员还会发现此功能的许多用途。顶级语句可提供类似脚本的试验体验，这与 Jupyter 笔记本提供的很类似。顶级语句非常适合小型控制台程序和实用程序。[Azure Functions](#) 是顶级语句的理想用例。

最重要的是，顶层语句不会限制应用程序的范围或复杂程度。这些语句可访问或使用任何 .NET 类。它们也不会限制你对命令行参数或返回值的使用。顶级语句可访问名为 `args` 的字符串数组。如果顶级语句返回整数值，则该值将成为来自合成 `Main` 方法的整数返回代码。顶级语句可能包含异步表达式。在这种情况下，合成入口点将返回 `Task` 或 `Task<int>`。

有关详细信息，请参阅 C# 编程指南中的[顶级语句](#)。

模式匹配增强功能

C# 9 包括新的模式匹配改进：

- 类型模式要求在变量是一种类型时匹配
- 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给定常数。

这些模式丰富了模式的语法。请考虑下列示例：

```
public static bool IsLetter(this char c) =>
    c is >= 'a' and <= 'z' or >= 'A' and <= 'Z';
```

使用可选的括号来明确 `and` 的优先级高于 `or`：

```
public static bool IsLetterOrSeparator(this char c) =>
    c is (>= 'a' and <= 'z') or (>= 'A' and <= 'Z') or '.' or ',';
```

最常见的用途之一是用于 NULL 检查的新语法：

```
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。

有关详细信息，请查看[模式\(C# 参考\)](#)。

有关详细信息，请参阅[模式](#)一文中的[关系模式](#)和[逻辑模式](#)部分。

性能和互操作性

3 项新功能改进了对需要高性能的本机互操作性和低级别库的支持：本机大小的整数、函数指针和省略 `localsinit` 标志。

本机大小的整数 `nint` 和 `nuint` 是整数类型。它们由基础类型 `System.IntPtr` 和 `System.UIntPtr` 表示。编译器将这些类型的其他转换和操作作为本机整数公开。本机大小的整数定义 `.MaxValue` 或 `.MinValue` 的属性。这些值不能表示为编译时编译时，因为它们取决于目标计算机上整数的本机大小。这些值在运行时是只读的。可在以下范围内对 `nint` 使用常量值：`[int.MinValue .. int.MaxValue]`。可在以下范围内对 `nuint` 使用常量值：`[uint.MinValue .. uint.MaxValue]`。编译器使用 `System.Int32` 和 `System.UInt32` 类型为所有一元和二元运算符执行常量折叠。如果结果不满足 32 位，操作将在运行时执行，且不会被视为常量。在广泛使用整数数学且需要尽可能快的性能的情况下，本机大小的整数可提高性能。有关详细信息，请参阅[nint 和 nuint 类型](#)

函数指针提供了一种简单的语法来访问 IL 操作码 `ldftn` 和 `calli`。可使用新的 `delegate*` 语法声明函数指针。`delegate*` 类型是指针类型。调用 `delegate*` 类型会使用 `calli`，而不是使用在 `Invoke()` 方法上采用 `callvirt` 的委托。从语法上讲，调用是相同的。函数指针调用使用 `managed` 调用约定。在 `delegate*` 语法后面添加 `unmanaged` 关键字，以声明想要 `unmanaged` 调用约定。可使用 `delegate*` 声明中的属性来指定其他调用约定。有关详细信息，请参阅[不安全代码和指针类型](#)。

最后，可添加 `System.Runtime.CompilerServices.SkipLocalsInitAttribute` 来指示编译器不要发出 `localsinit` 标志。此标志指示 CLR 对所有局部变量进行零初始化。从 1.0 开始，`localsinit` 标志一直是 C# 的默认行为。但在某些情况下，额外的零初始化可能会对性能产生可衡量的影响，特别是在使用 `stackalloc` 时。在这些情况下，可添加 `SkipLocalsInitAttribute`。可将它添加到单个方法或属性中，或者添加到 `class`、`struct`、`interface`，甚至是模块中。此属性不会影响 `abstract` 方法，它会影响为实现生成的代码。有关详细信息，请参阅[SkipLocalsInit 属性](#)。

这些功能在某些情况下可提高性能。仅应在采用前后对这些功能进行仔细的基准测试之后使用它们。涉及本机大小整数的代码必须在使用不同整数大小的多个目标平台上进行测试。其他功能需要不安全的代码。

调整和完成功能

还有其他很多功能有助于更高效地编写代码。在 C# 9.0 中，已知创建对象的类型时，可在 `new` 表达式中省略该

类型。最常见的用法是在字段声明中：

```
private List<WeatherObservation> _observations = new();
```

当需要创建新对象作为参数传递给方法时，也可使用目标类型 `new`。请考虑使用以下签名的 `ForecastFor()` 方法：

```
public WeatherForecast ForecastFor(DateTime forecastDate, WeatherForecastOptions options)
```

可按如下所示调用该方法：

```
var forecast = station.ForecastFor(DateTime.Now.AddDays(2), new());
```

此功能还有一个不错的用途是，将其与仅限 `init` 的属性组合使用来初始化新对象：

```
WeatherStation station = new() { Location = "Seattle, WA" };
```

可使用 `return new();` 语句返回由默认构造函数创建的实例。

类似的功能可改进[条件表达式](#)的目标类型解析。进行此更改后，两个表达式无需从一个隐式转换到另一个，而是都可隐式转换为目标类型。你可能不会注意到此更改。你会注意到，某些以前需要强制转换或无法编译的条件表达式现在可以正常工作。

从 C# 9.0 开始，可将 `static` 修饰符添加到[Lambda 表达式或匿名方法](#)。静态 Lambda 表达式类似于 `static` 局部函数：静态 Lambda 或匿名方法无法捕获局部变量或实例状态。`static` 修饰符可防止意外捕获其他变量。

协变返回类型为[重写](#)方法的返回类型提供了灵活性。重写方法可返回从重写基方法的返回类型派生的类型。这对于记录和其他支持虚拟克隆或工厂方法的类型很有用。

此外，`foreach` 循环将识别并使用扩展方法 `GetEnumerator`，否则将满足 `foreach` 模式。此更改意味着 `foreach` 与其他基于模式的构造（例如异步模式和基于模式的析构）一致。实际上，此更改意味着可以为任何类型添加 `foreach` 支持。在设计中，应将其限制为在枚举对象有意义时使用。

接下来，可使用弃元作为 Lambda 表达式的参数。这样可免于为参数命名，并且编译器也可避免使用它。可将 `_` 用于任何参数。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。

最后，现在可将属性应用于[本地函数](#)。例如，可将[可为空的属性注释](#)应用于本地函数。

支持代码生成器

最后两项功能支持 C# 代码生成器。C# 代码生成器是可编写的组件，类似于 roslyn 分析器或代码修补程序。区别在于，代码生成器会在编译过程中分析代码并编写新的源代码文件。典型的代码生成器会在代码中搜索属性或其他约定。

代码生成器使用 Roslyn 分析 API 读取属性或其他代码元素。通过该信息，它将新代码添加到编译中。源生成器只能添加代码，不能修改编译中的任何现有代码。

为代码生成器添加的两项功能是“分部方法语法”和“模块初始化表达式”的扩展。首先是对分部方法的更改。在 C# 9.0 之前，分部方法为 `private`，但不能指定访问修饰符、不能返回 `void`，也不能具有 `out` 参数。这些限制意味着，如果未提供任何方法实现，编译器会删除对分部方法的所有调用。C# 9.0 消除了这些限制，但要求分部方法声明必须具有实现。代码生成器可提供这种实现。为了避免引入中断性变更，编译器会考虑没有访问修饰符的任何分部方法，以遵循旧规则。如果分部方法包括 `private` 访问修饰符，则由新规则控制该分部方法。有关详细信息，请查看[分部方法 \(C# 参考\)](#)。

代码生成器的第二项新功能是模块初始化表达式。模块初始化表达式是附加了 [ModuleInitializerAttribute](#) 属性的方法。在整个模块中进行任何其他字段访问或方法调用之前，运行时将调用这些方法。模块初始化表达式方法：

- 必须是静态的
- 必须没有参数
- 必须返回 void
- 不能是泛型方法
- 不能包含在泛型类中
- 必须能够从包含模块访问

最后一个要点实际上意味着该方法及其包含类必须是内部的或公共的。方法不能为本地函数。有关详细信息，请参阅 [ModuleInitializer 属性](#)。

C# 8.0 中的新增功能

2021/5/7 • [Edit Online](#)

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](#)
- [异步可释放](#)
- [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](#)

".NET Core 3.x" 和 ".NET Standard 2.1" 支持 C# 8.0。有关详细信息，请参阅 [C# 语言版本控制](#)。

本文的剩余部分将简要介绍这些功能。如果有详细讲解的文章，则将提供指向这些教程和概述的链接。可以使用 `dotnet try` 全局工具在环境中浏览这些功能：

1. 安装 [dotnet-try](#) 全局工具。
2. 克隆 [dotnet/try-samples](#) 存储库。
3. 将当前目录设置为 `try-samples` 存储库的 `csharp8` 子目录。
4. 运行 `dotnet try`。

Readonly 成员

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

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

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

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

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

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

```
warning CS8656: Call to non-readonly member 'Point.Distance.get' from a 'readonly' member results in an
implicit copy of 'this'
```

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

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

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

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

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

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

有关详细信息，请参阅[结构类型](#)一文中的 `readonly` 实例成员部分。

默认接口方法

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

默认接口方法会影响很多方案和语言元素。我们的第一个教程介绍如何[使用默认实现更新接口](#)。

在更多位置中使用更多模式

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

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

除了可以在新位置使用新模式之外，C# 8.0 还添加了“递归模式”。递归模式是可以包含其他模式的模式。

switch 表达式

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

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

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

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

这里有几个语法改进：

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

将其与使用经典 `switch` 语句的等效代码进行对比：

```

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

```

有关详细信息，请参阅 [switch 表达式](#)。

属性模式

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

```

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

```

模式匹配为表达此算法创建了简洁的语法。

有关详细信息，请参阅[模式一文的属性模式部分](#)。

元组模式

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

```

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

```

消息指示获胜者。弃元表示平局(石头剪刀布游戏)的三种组合或其他文本输入。

位置模式

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

```
public class Point
{
    public int X { get; }
    public int Y { get; }

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

    public void Deconstruct(out int x, out int y) =>
        (x, y) = (X, Y);
}
```

此外，请考虑以下表示象限的各种位置的枚举：

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

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

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

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

可在此[模式匹配高级教程](#)中探索模式匹配方法。有关位置模式的详细信息，请参阅[模式的位置模式](#)部分。

using 声明

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

```

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

```

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

```

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

```

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

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

静态本地函数

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

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

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

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

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

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

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

可处置的 ref 结构

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

可为空引用类型

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

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

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

可以在[可为空引用类型](#)的概述中了解该功能的更多信息。可以在此[可为空引用类型教程](#)中的新应用程序中自行尝试。在[迁移应用程序以使用可为空引用类型教程](#)中了解迁移现有代码库以使用可为空引用类型的步骤。

异步流

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

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

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

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

可以使用 `await foreach` 语句来枚举序列：

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

可以在[创建和使用异步流](#)的教程中自行尝试异步流。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。

异步可释放

从 C# 8.0 开始，语言支持实现 `System.IAsyncDisposable` 接口的异步可释放类型。可使用 `await using` 语句来处理异步可释放对象。有关详细信息，请参阅[实现 DisposeAsync 方法](#)。

索引和范围

索引和范围为访问序列中的单个元素或范围提供了简洁的语法。

此语言支持依赖于两个新类型和两个新运算符：

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

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

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

请看以下几个示例。请考虑以下数组，用其顺数索引和倒数索引进行注释：

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

可以使用 `^1` 索引检索最后一个词：

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

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

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

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

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

下面的示例为开始和/或结束创建了开放范围：

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

此外可以将范围声明为变量：

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

然后可以在 `[` 和 `]` 字符中使用该范围：

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

不仅数组支持索引和范围。还可以将索引和范围用于 `string`、`Span<T>` 或 `ReadOnlySpan<T>`。有关详细信息，请参阅[索引和范围的类型支持](#)。

可在有关[索引和范围](#)的教程中详细了解索引和范围。

Null 合并赋值

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

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

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

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

有关详细信息，请参阅 [?? 和 ??= 运算符](#) 一文。

非托管构造类型

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

例如，假设泛型 `Coords<T>` 类型有以下定义：

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

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

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

有关详细信息，请参阅[非托管类型](#)。

嵌套表达式中的 `stackalloc`

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

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

内插逐字字符串的增强功能

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

C# 7.0 - C# 7.3 中的新增功能

2021/5/7 • [Edit Online](#)

C# 7.0 - C# 7.3 为 C# 开发体验带来了大量功能和增量改进。本文概述了新的语言功能和编译器选项。说明中描述了 C# 7.3 的行为, C# 7.3 是基于 .NET Framework 的应用程序支持的最新版本。

C# 7.1 中添加了[语言版本选择](#)配置元素, 因此你可以在项目文件中指定编译器语言版本。

C# 7.0-7.3 将这些功能和主题添加到了 C# 语言中:

- [元组和弃元](#)

- 可以创建包含多个公共字段的轻量级未命名类型。编译器和 IDE 工具可理解这些类型的语义。
- 弃元是指在不关心所赋予的值时, 赋值中使用的临时只写变量。在对元组和用户定义类型进行解构, 以及在使用 `out` 参数调用方法时, 它们的作用最大。

- [模式匹配](#)

- 可以基于任意类型和这些类型的成员的值创建分支逻辑。

- [async Main 方法](#)

- 应用程序的入口点可以含有 `async` 修饰符。

- [本地函数](#)

- 可以将函数嵌套在其他函数内, 以限制其范围和可见性。

- [更多的 expression-bodied 成员](#)

- 可使用表达式创作的成员列表有所增长。

- [throw 表达式](#)

- 可以在之前因为 `throw` 是语句而不被允许的代码构造中引发异常。

- [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。

- [数字文本语法改进](#)

- 新令牌可提高数值常量的可读性。

- [out 变量](#)

- 可以将 `out` 值内联作为参数声明到使用这些参数的方法中。

- [非尾随命名参数](#)

- 命名的参数可后接位置参数。

- [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。

- [改进了重载解析](#)

- 用于解决重载解析歧义的新规则。

- [编写安全高效代码的技巧](#)

- 结合了多项语法改进, 可使用引用语义处理值类型。

最后, 编译器中添加了新的选项:

- 控制[应用程序集生成](#)的 `-refout` 和 `-refonly`。
- `-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。

本文的其余部分概述了每个功能。你将了解每项功能背后的原理和语法。可以使用 `dotnet try` 全局工具在环境中浏览这些功能:

1. 安装 [dotnet-try](#) 全局工具。
2. 克隆 [dotnet/try-samples](#) 存储库。
3. 将当前目录设置为 try-samples 存储库的 csharp7 子目录。
4. 运行 `dotnet try`。

元组和弃元

C# 为用于说明设计意图的类和结构提供了丰富的语法。但是，这种丰富的语法有时会需要额外的工作，但益处却很少。你可能经常编写需要包含多个数据元素的简单结构的方法。为了支持这些方案，已将元组添加到了 C#。元组是包含多个字段以表示数据成员的轻量级数据结构。这些字段没有经过验证，并且你无法定义自己的方法。C# 元组类型支持 `==` 和 `!=`。有关详细信息，

NOTE

低于 C# 7.0 的版本中也提供元组，但它们效率低下且不具有语言支持。这意味着元组元素只能作为 `Item1` 和 `Item2` 等引用。C# 7.0 引入了对元组的语言支持，可利用更有效的新元组类型向元组字段赋予语义名称。

可以通过为每个成员赋值来创建元组，并可选择为元组的每个成员提供语义名称：

```
(string Alpha, string Beta) namedLetters = ("a", "b");
Console.WriteLine($"{namedLetters.Alpha}, {namedLetters.Beta}");
```

`namedLetters` 元组包含称为 `Alpha` 和 `Beta` 的字段。这些名称仅存在于编译时且不保留，例如在运行时使用反射来检查元组时。

在进行元组赋值时，还可以指定赋值右侧的字段的名称：

```
var alphabetStart = (Alpha: "a", Beta: "b");
Console.WriteLine($"{alphabetStart.Alpha}, {alphabetStart.Beta}");
```

在某些时候，你可能想要解包从方法返回的元组的成员。可通过为元组中的每个值声明单独的变量来实现此目的。这种解包操作称为解构元组：

```
(int max, int min) = Range(numbers);
Console.WriteLine(max);
Console.WriteLine(min);
```

还可以为 .NET 中的任何类型提供类似的析构。编写 `Deconstruct` 方法，用作类的成员。`Deconstruct` 方法为你提供提取每个属性的一组 `out` 参数。考虑提供析构函数方法的此 `Point` 类，该方法提取 `X` 和 `Y` 坐标：

```
public class Point
{
    public Point(double x, double y)
        => (X, Y) = (x, y);

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

    public void Deconstruct(out double x, out double y) =>
        (x, y) = (X, Y);
}
```

可以通过向元组分配 `Point` 来提取各个字段：

```
var p = new Point(3.14, 2.71);
(double X, double Y) = p;
```

在初始化元组时，许多时候，赋值操作右侧的变量名与用于元组元素的名称相同：元组元素的名称可通过用于初始化元组的变量进行推断：

```
int count = 5;
string label = "Colors used in the map";
var pair = (count, label); // element names are "count" and "label"
```

若要详细了解此功能，可以参阅[元组类型](#)一文。

通常，在进行元组解构或使用 `out` 参数调用方法时，必须定义一个其值无关紧要且你不打算使用的变量。为处理此情况，C# 增添了对弃元的支持。弃元是一个名为 `_`（下划线字符）的只写变量，可向单个变量赋予要放弃的所有值。弃元类似于未赋值的变量；不可在代码中使用弃元（赋值语句除外）。

在以下方案中支持弃元：

- 在对元组或用户定义的类型进行解构时。
- 在使用 `out` 参数调用方法时。
- 在使用 `is` 和 `switch` 语句匹配操作的模式中。
- 在要将某赋值的值显式标识为弃元时用作独立标识符。

以下示例定义了 `QueryCityDataForYears` 方法，它返回一个包含两个不同年份的城市数据的六元组。本例中，方法调用仅与此方法返回的两个人口值相关，因此在进行元组解构时，将元组中的其余值视为弃元。

```

using System;
using System.Collections.Generic;

public class Example
{
    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

```

有关详细信息，请参阅[弃元](#)。

模式匹配

模式匹配是一组功能，利用这些功能，你可以通过新的方式在代码中表示控制流。你可以测试变量的类型、值或其属性的值。这些方法可创建可读性更佳的代码流。

模式匹配支持 `is` 表达式和 `switch` 表达式。每个表达式都允许检查对象及其属性以确定该对象是否满足所寻求的模式。使用 `when` 关键字来指定模式的其他规则。

`is` 模式表达式扩展了常用 `is` 运算符以查询关于其类型的对象，并在一条指令分配结果。以下代码检查变量是否为 `int`，如果是，则将其添加到当前总和：

```

if (input is int count)
    sum += count;

```

前面的小型示例演示了 `is` 表达式的增强功能。可以针对值类型和引用类型进行测试，并且可以将成功结果分配给类型正确的新变量。

`switch` 匹配表达式具有常见的语法，它基于已包含在 C# 语言中的 `switch` 语句。更新后的 `switch` 语句有几个新构造：

- `switch` 表达式的控制类型不再局限于整数类型、`Enum` 类型、`string` 或与这些类型之一对应的可为 `null` 的

类型。可能会使用任何类型。

- 可以在每个 `case` 标签中测试 `switch` 表达式的类型。与 `is` 表达式一样，可以为该类型指定一个新变量。
- 可以添加 `when` 子句以进一步测试该变量的条件。
- `case` 标签的顺序现在很重要。执行匹配的第一个分支；其他将跳过。

以下代码演示了这些新功能：

```
public static int SumPositiveNumbers(IEnumerable<object> sequence)
{
    int sum = 0;
    foreach (var i in sequence)
    {
        switch (i)
        {
            case 0:
                break;
            case IEnumerable<int> childSequence:
            {
                foreach(var item in childSequence)
                    sum += (item > 0) ? item : 0;
                break;
            }
            case int n when n > 0:
                sum += n;
                break;
            case null:
                throw new NullReferenceException("Null found in sequence");
            default:
                throw new InvalidOperationException("Unrecognized type");
        }
    }
    return sum;
}
```

- `case 0:` 是常量模式。
- `case IEnumerable<int> childSequence:` 是声明模式。
- `case int n when n > 0:` 是具有附加 `when` 条件的声明模式。
- `case null:` 是 `null` 常量模式。
- `default:` 是常见的默认事例。

自 C# 7.1 起，`is` 和 `switch` 类型模式的模式表达式的类型可能为泛型类型参数。这可能在检查 `struct` 或 `class` 类型且要避免装箱时最有用。

可以在 [C# 中的模式匹配](#) 中了解有关模式匹配的更多信息。

异步 `main` 方法

异步 `Main` 方法使你能够在 `Main` 方法中使用 `await` 关键字。在过去，需要编写：

```
static int Main()
{
    return DoAsyncWork().GetAwaiter().GetResult();
}
```

现在，您可以编写：

```
static async Task<int> Main()
{
    // This could also be replaced with the body
    // DoAsyncWork, including its await expressions:
    return await DoAsyncWork();
}
```

如果程序不返回退出代码，可以声明返回 `Task` 的 `Main` 方法：

```
static async Task Main()
{
    await SomeAsyncMethod();
}
```

如需了解更多详情，可以阅读编程指南中的[异步 Main](#) 一文。

本地函数

许多类的设计都包括仅从一个位置调用的方法。这些额外的私有方法使每个方法保持小且集中。本地函数使你能够在另一个方法的上下文内声明方法。本地函数使得类的阅读者更容易看到本地方法仅从声明它的上下文中调用。

对于本地函数有两个常见的用例：公共迭代器方法和公共异步方法。这两种类型的方法都生成报告错误的时间晚于程序员期望时间的代码。在迭代器方法中，只有在调用枚举返回的序列的代码时才会观察到任何异常。在异步方法中，只有当返回的 `Task` 处于等待状态时才会观察到任何异常。以下示例演示如何使用本地函数将参数验证与迭代器实现分离：

```
public static IEnumerable<char> AlphabetSubset3(char start, char end)
{
    if (start < 'a' || start > 'z')
        throw new ArgumentOutOfRangeException(paramName: nameof(start), message: "start must be a letter");
    if (end < 'a' || end > 'z')
        throw new ArgumentOutOfRangeException(paramName: nameof(end), message: "end must be a letter");

    if (end <= start)
        throw new ArgumentException($"{nameof(end)} must be greater than {nameof(start)}");

    return alphabetSubsetImplementation();

    IEnumerable<char> alphabetSubsetImplementation()
    {
        for (var c = start; c < end; c++)
            yield return c;
    }
}
```

可以对 `async` 方法采用相同的技术，以确保在异步工作开始之前引发由参数验证引起的异常：

```

public 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 longRunningWorkImplementation();

    async Task<string> longRunningWorkImplementation()
    {
        var interimResult = await FirstWork(address);
        var secondResult = await SecondStep(index, name);
        return $"The results are {interimResult} and {secondResult}. Enjoy.";
    }
}

```

现在支持此语法：

```

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

```

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

NOTE

本地函数支持的某些设计也可以使用 lambda 表达式来完成。有关详细信息，请参阅[本地函数与 Lambda 表达式](#)。

更多的 expression-bodied 成员

C# 6 为成员函数和只读属性引入了 expression-bodied 成员。C# 7.0 扩展了可作为表达式实现的允许的成员。在 C# 7.0 中，你可以在属性和索引器上实现构造函数、终结器以及 `get` 和 `set` 访问器。以下代码演示了每种情况的示例：

```

// Expression-bodied constructor
public ExpressionMembersExample(string label) => this.Label = label;

// Expression-bodied finalizer
~ExpressionMembersExample() => Console.Error.WriteLine("Finalized!");

private string label;

// Expression-bodied get / set accessors.
public string Label
{
    get => label;
    set => this.label = value ?? "Default label";
}

```

NOTE

本示例不需要终结器，但显示它是为了演示语法。不应在类中实现终结器，除非有必要发布非托管资源。还应考虑使用 `SafeHandle` 类，而不是直接管理非托管资源。

这些 expression-bodied 成员的新位置代表了 C# 语言的一个重要里程碑：这些功能由致力于开发开放源代码 Roslyn 项目的社区成员实现。

将方法更改为 expression bodied 成员是二进制兼容的更改。

引发表达式

在 C# 中，`throw` 始终是一个语句。因为 `throw` 是一个语句而非表达式，所以在某些 C# 构造中无法使用它。它们包括条件表达式、null 合并表达式和一些 lambda 表达式。添加 expression-bodied 成员将添加更多位置，在这些位置中，`throw` 表达式会很有用。为了可以编写这些构造，C# 7.0 引入了 [throw 表达式](#)。

这使得编写更多基于表达式的代码变得更容易。不需要其他语句来进行错误检查。

默认文本表达式

默认文本表达式是针对默认值表达式的一项增强功能。这些表达式将变量初始化为默认值。过去会这么编写：

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

现在，可以省略掉初始化右侧的类型：

```
Func<string, bool> whereClause = default;
```

有关详细信息，请参阅 [default 运算符](#) 一文中的 [default 文本](#) 部分。

数字文本语法改进

误读的数值常量可能使第一次阅读代码时更难理解。位掩码或其他符号值容易产生误解。C# 7.0 包括两项新功能，可用于以最可读的方式写入数字来用于预期用途：二进制文本和数字分隔符。

在创建位掩码时，或每当数字的二进制表示形式使代码最具可读性时，以二进制形式写入该数字：

```
public const int Sixteen = 0b0001_0000;
public const int ThirtyTwo = 0b0010_0000;
public const int SixtyFour = 0b0100_0000;
public const int OneHundredTwentyEight = 0b1000_0000;
```

常量开头的 `0b` 表示该数字以二进制数形式写入。二进制数可能会很长，因此通过引入 `_` 作为数字分隔符通常更容易查看位模式，如前面示例中的二进制常量所示。数字分隔符可以出现在常量的任何位置。对于十进制数字，通常将其用作千位分隔符。十六进制和二进制文本可采用 `_` 开头：

```
public const long BillionsAndBillions = 100_000_000_000;
```

数字分隔符也可以与 `decimal`、`float` 和 `double` 类型一起使用：

```
public const double AvogadroConstant = 6.022_140_857_747_474e23;
public const decimal GoldenRatio = 1.618_033_988_749_894_848_204_586_834_365_638_117_720_309_179M;
```

综观来说，你可以声明可读性更强的数值常量。

`out` 变量

支持 `out` 参数的现有语法已在 C# 7 中得到改进。现在可以在方法调用的参数列表中声明 `out` 变量，而不是编写单独的声明语句：

```
if (int.TryParse(input, out int result))
    Console.WriteLine(result);
else
    Console.WriteLine("Could not parse input");
```

为清楚起见，我们建议你指定 `out` 变量的类型，如前面的示例所示。但是，该语言支持使用隐式类型的局部变量：

```
if (int.TryParse(input, out var answer))
    Console.WriteLine(answer);
else
    Console.WriteLine("Could not parse input");
```

- 代码更易于阅读。
 - 在使用 `out` 变量的地方声明 `out` 变量，而不是在上面的代码行中声明。
- 无需分配初始值。
 - 通过在方法调用中使用 `out` 变量的位置声明该变量，使得在分配它之前不可能意外使用它。

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

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

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

非尾随命名参数

方法调用现可使用位于位置参数前面的命名参数（若这些命名参数的位置正确）。如需了解详情，请参阅[命名参数和可选参数](#)。

private protected 访问修饰符

新的复合访问修饰符：`private protected` 指示可通过包含同一程序集中声明的类或派生类来访问成员。虽然 `protected internal` 允许通过同一程序集中的类或派生类进行访问，但 `private protected` 限制对同一程序集中声明的派生类的访问。

如需了解详情，请参阅语言参考中的[访问修饰符](#)。

改进了重载候选选项

在每个版本中，对重载解析规则进行了更新，以解决多义方法调用具有“明显”选择的情况。此版本添加了三个新

规则，以帮助编译器选取明显的选择：

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

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

启用更高效的安全代码

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

以下新增功能支持使安全代码获得更好的性能的主题：

- 无需固定即可访问固定的字段。
- 可以重新分配 `ref` 本地变量。
- 可以使用 `stackalloc` 数组上的初始值设定项。
- 可以对支持模式的任何类型使用 `fixed` 语句。
- 可以使用其他泛型约束。
- 针对实参的 `in` 修饰符，指定形参通过引用传递，但不通过调用方法修改。将 `in` 修饰符添加到参数是源兼容的更改。
- 针对方法返回的 `ref readonly` 修饰符，指示方法通过引用返回其值，但不允许写入该对象。如果向某个值赋予返回值，则添加 `ref readonly` 修饰符是源兼容的更改。将 `readonly` 修饰符添加到现有的 `ref` 返回语句是不兼容的更改。它要求调用方更新 `ref` 本地变量的声明以包含 `readonly` 修饰符。
- `readonly struct` 声明，指示结构不可变，且应作为 `in` 参数传递到其成员方法。将 `readonly` 修饰符添加到现有的结构声明是二进制兼容的更改。
- `ref struct` 声明，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。将 `ref` 修饰符添加到现有 `struct` 声明是不兼容的更改。`ref struct` 不能是类的成员，也不能用于可能在堆上分配的其他位置。

可以在[编写安全高效的代码](#)中详细了解所有这些更改。

Ref 局部变量和返回结果

此功能允许使用并返回对变量的引用的算法，这些变量在其他位置定义。一个示例是使用大型矩阵并查找具有某些特征的单个位置。下面的方法在矩阵中向该存储返回“引用”：

```
public static ref int Find(int[,] matrix, Func<int, bool> predicate)
{
    for (int i = 0; i < matrix.GetLength(0); i++)
        for (int j = 0; j < matrix.GetLength(1); j++)
            if (predicate(matrix[i, j]))
                return ref matrix[i, j];
    throw new InvalidOperationException("Not found");
}
```

可以将返回值声明为 `ref` 并在矩阵中修改该值，如以下代码所示：

```
ref var item = ref MatrixSearch.Find(matrix, (val) => val == 42);
Console.WriteLine(item);
item = 24;
Console.WriteLine(matrix[4, 2]);
```

C# 语言还有多个规则，可保护你免于误用 `ref` 局部变量和返回结果：

- 必须将 `ref` 关键字添加到方法签名和方法中的所有 `return` 语句中。
 - 这清楚地表明，该方法在整个方法中通过引用返回。
- 可以将 `ref return` 分配给值变量或 `ref` 变量。
 - 调用方控制是否复制返回值。在分配返回值时省略 `ref` 修饰符表示调用方需要该值的副本，而不是对存储的引用。
- 不可向 `ref` 本地变量赋予标准方法返回值。
 - 因为那将禁止类似 `ref int i = sequence.Count();` 这样的语句
- 不能将 `ref` 返回给其生存期不超出方法执行的变量。
 - 这意味着不可返回对本地变量或对类似作用域变量的引用。
- `ref` 局部变量和返回结果不可用于异步方法。
 - 编译器无法知道异步方法返回时，引用的变量是否已设置为其最终值。

添加 `ref` 局部变量和 `ref` 返回结果可通过避免复制值或多次执行取消引用操作，允许更为高效的算法。

向返回值添加 `ref` 是源兼容的更改。现有代码会进行编译，但在分配时复制 `ref` 返回值。调用方必须将存储的返回值更新为 `ref` 局部变量，从而将返回值存储为引用。

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

```
ref VeryLargeStruct refLocal = ref veryLargeStruct; // initialization
refLocal = ref anotherVeryLargeStruct; // reassigned, refLocal refers to different storage.
```

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

有关详细信息，请参阅 `ref` 关键字一文。

条件 `ref` 表达式

最后，条件表达式可能生成 `ref` 结果而不是值。例如，你将编写以下内容以检索对两个数组之一中第一个元素的引用：

```
ref var r = ref (arr != null ? ref arr[0] : ref otherArr[0]);
```

变量 `r` 是对 `arr` 或 `otherArr` 中第一个值的引用。

有关详细信息，请参阅语言参考中的 [条件运算符 \(?\)](#)。

`in` 参数修饰符

`in` 关键字补充了现有的 `ref` 和 `out` 关键字，以按引用传递参数。`in` 关键字指定按引用传递参数，但调用的方法不修改值。

你可以声明按值或只读引用传递的重载，如以下代码所示：

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

按值(前面示例中的第一个)传递的重载比按只读引用传递的重载更好。若要使用只读引用参数调用版本，必须

在调用方法前添加 `in` 修饰符。

有关详细信息, 请参阅有关 [in 参数修饰符](#) 的文章。

更多类型支持 `fixed` 语句

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

有关详细信息, 请参阅语言参考中的 [fixed 语句](#) 一文。

索引 `fixed` 字段不需要进行固定

请考虑此结构:

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

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

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

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

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

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

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

有关详细信息, 请参阅有关 [fixed 语句](#) 的文章。

`stackalloc` 数组支持初始值设定项

当你对数组中的元素的值进行初始值设定时, 你已能够指定该值:

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

现在，可向使用 `stackalloc` 进行声明的数组应用同一语法：

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

有关详细信息，请参阅 [stackalloc 运算符](#) 一文。

增强的泛型约束

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

现在也可以使用新的 `unmanaged` 约束来指定类型参数必须是不可为 null 的“[非托管类型](#)”。

有关详细信息，请参阅有关 `where` 泛型约束和类型参数的约束的文章。

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

通用的异步返回类型

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。

新语言功能意味着异步方法返回类型不限于 `Task`、`Task<T>` 和 `void`。返回类型必须仍满足异步模式，这意味着 `GetAwaiter` 方法必须是可访问的。作为一个具体示例，已将 `ValueTask` 类型添加到 .NET 中，以使用这一新语言功能：

```
public async ValueTask<int> Func()  
{  
    await Task.Delay(100);  
    return 5;  
}
```

NOTE

你需要添加 NuGet 包 `System.Threading.Tasks.Extensions` 才能使用 `ValueTask<TResult>` 类型。

此增强功能对于库作者最有用，可避免在性能关键型代码中分配 `Task`。

新的编译器选项

新的编译器选项支持 C# 程序的新版本和 DevOps 方案。

引用程序集生成

有两种新的编译器选项可生成仅引用程序集：`ProduceReferenceAssembly` 和 `ProduceOnlyReferenceAssembly`。链接的文章详细介绍了这些选项和引用程序集。

公共或开放源代码签名

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

有关详细信息，请参阅 [PublicSign 编译器选项](#) 一文。

pathmap

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

有关详细信息，请参阅 [PathMap 编译器选项](#)一文。

了解 C# 编译器中的所有重大更改

2021/5/7 • [Edit Online](#)

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：

- [.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 5.0 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

2021/5/7 • [Edit Online](#)

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。

IMPORTANT

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 [Exception](#)，将会检查每个 `throw` 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# 生成工具将最新的主要语言版本视为默认语言版本。主要版本之间可能有单点修正发行版。有关详细信息，请参阅本节中的其他文章。若要使用单点版本中的最新功能，需要配置编译器语言版本并选择版本。自 C# 7.0 起，已有三个单点修正发行版：

- C# 7.3：
 - 自 [Visual Studio 2017 版本 15.7](#) 和 [.NET Core 2.1 SDK](#) 起，开始随附 C# 7.3。
- C# 7.2：
 - 自 [Visual Studio 2017 版本 15.5](#) 和 [.NET Core 2.0 SDK](#) 起，开始随附 C# 7.2。
- C# 7.1：
 - 自 [Visual Studio 2017 版本 15.3](#) 和 [.NET Core 2.0 SDK](#) 起，开始随附 C# 7.1。

C# 1.0 版

回想起来，和 Visual Studio .NET 2002 一起发布的 C# 版本 1.0 非常像 Java。在 [ECMA 制定的设计目标](#) 中，它旨在成为一种“简单、现代、面向对象的常规用途语言”。当时，它和 Java 类似，说明已经实现了上述早期设计目标。

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](#)

C# 版本 1.2

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 `IEnumerator` 实现 `IDisposable` 时，`foreach` 循环中生成的代码会在 `IEnumerator` 上调用 `Dispose`。

C# 2.0 版

从此以后事情变得有趣起来。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

C# 一开始是面向对象的 (OO) 通用语言，而 C# 2.0 版很快改变了这一点。做好基础准备后，他们开始追求解决一些严重影响开发者的难点。且他们以显著的方式追求这些难点。

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 `ArrayList` 的 `ListInt` 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 `foreach` 循环来检查 `List` (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。

不过 C# 依然在追赶 Java 的道路上。当时 Java 已发布包含泛型和迭代器的版本。但是随着语言各自的演化，形势很快发生了变化。

C# 3.0 版

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。我们通常认为 C# 版本的杀手锏是查询表达式，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#

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/函数式混合语言打下基础。

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。

人们需要一些时间来掌握和吸收这种概念，不过已经逐渐做到了。现在又过了几年，代码变得更简洁，功能也更强大了。

C# 4.0 版

C# 版本 4.0 随 Visual Studio 2010 一起发布，很难达到版本 3.0 的创新水平。在 3.0 版中，C# 已经完全从 Java 的阴影中脱颖而出，崭露头角。很快成为一种简洁精炼的语言。

下一版本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](#)

嵌入的互操作类型缓和了部署难点。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# 5.0 版

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](#)

另请参阅

- [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，另现壮大为改观。如果你以前处理过冗长的运行操作以及实现回调的 Web，应该会爱上这项语言功能。

C# 6.0 版

C# 在 3.0 版和 5.0 版对面向对象的语言添加了主要的新功能。版本 6.0 随 Visual Studio 2015 一起发布，通过该版本，它不再推出主导性的杀手锏，而是发布了很多使得 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

这些功能每一个都很有趣。但从整体来看，可以发现一个有趣的模式。在此版本中，C# 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# 7.0 版

C# 7.0 版已与 Visual Studio 2017 一起发布。虽然该版本继承和发展了 C# 6.0，但不包含编译器即服务。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 局部变量和返回结果](#)

其他功能包括：

- [弃元](#)
- [二进制文本和数字分隔符](#)
- [引发表达式](#)

这些都为开发者提供了很棒的新功能，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。

但 C# 的用途更加广泛了。.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# 7.1 版

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：

- [`async` `Main` 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# 7.2 版

C# 7.2 版添加了几个小型语言功能：

- 编写安全高效代码的技巧

- 结合了多项语法改进，可使用引用语义处理值类型。

- 非尾随命名参数

- 命名的参数可后接位置参数。

- 数值文字中的前导下划线

- 数值文字现在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# 7.3 版

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

以下新增功能支持使安全代码获得更好的性能的主题：

- 无需固定即可访问固定的字段。

- 可以重新分配 `ref` 本地变量。

- 可以使用 `stackalloc` 数组上的初始值设定项。

- 可以对支持模式的任何类型使用 `fixed` 语句。

- 可以使用其他泛型约束。

对现有功能进行了以下增强：

- 可以使用元组类型测试 `==` 和 `!=`。

- 可以在多个位置使用表达式变量。

- 可以将属性附加到自动实现的属性的支持字段。

- 由 `in` 区分的参数的方法解析得到了改进。

- 重载解析的多义情况现在变得更少。

新的编译器选项为：

- `-publicsign`，用于启用程序集的开放源代码软件 (OSS) 签名。

- `-pathmap` 用于提供源目录的映射。

C# 8.0 版

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 CLR 功能，而其他功能依赖于仅在 .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。

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

2020/11/2 • [Edit Online](#)

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。如有必要，在针对尚未部署这些类型或成员的环境编写代码时，可使用包含较新版本的语言所需类型的 NuGet 包。

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` - C# `string` 类型是 `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 继续添加 `ValueTuple` 的依赖项，以实现元组语言功能。

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。必须了解如何管理工作中的这些依赖项。

管理依赖项

C# 编译器工具现在从支持的平台上 .NET 库的发布周期分离。实际上，不同的 .NET 库有不同的发布周期：Windows 上的 .NET Framework 作为 Windows 更新发布，.NET Core 在单独的计划中提供，Xamarin 版本的库更新随适用于每个目标平台的 Xamarin 工具提供。

大多数时候，用户都不会注意到这些更改。但是，如果使用的较新版本语言需要该平台上的 .NET 库中尚未包含的功能，则会引用 NuGet 包以提供这些新类型。应用支持的平台会随着新框架的安装而更新，因此可以删除额外的引用。

此分离意味着即使面向没有相应框架的计算机，仍可使用新语言功能。

面向 C# 开发人员的版本和更新注意事项

2020/11/2 • [Edit Online](#)

兼容性是一个非常重要的目标，因为新功能已添加到 C# 语言中。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。

在库中采用新语言功能时，可能需要多加注意。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。在决定采用新功能时，需要考虑两种兼容性：源兼容和二进制兼容。

二进制兼容的更改

如果更新的库可以在不重新生成应用程序和使用它的库的情况下使用，那么对库的更改属于二进制兼容的更改。不需要重新生成从属程序集，也不需要更改任何源代码。二进制兼容的更改也是源兼容的更改。

源兼容的更改

如果使用库的应用程序和库不需要更改源代码，但必须根据新版本重新编译源才能正常工作，那么对库的更改属于源兼容的更改。

不兼容的更改

如果更改既不是源兼容的更改，也不是二进制兼容的更改，则需要在从属库和应用程序中进行源代码更改和重新编译。

评估库

这些兼容性概念会影响库的公共声明和受保护声明，而不会影响其内部实现。在内部采用任何新功能始终是二进制兼容的更改。

二进制兼容的更改提供新语法，可以生成与旧语法相同的公共声明的已编译代码。例如，将方法更改为 expression-bodied 的成员是二进制兼容的更改：

原始代码：

```
public double CalculateSquare(double value)
{
    return value * value;
}
```

新代码：

```
public double CalculateSquare(double value) => value * value;
```

源兼容的更改引入了可更改公共成员的已编译代码的语法，只不过是通过与现有调用站点兼容的方式执行。例如，将方法签名从 by 值参数更改为 `in` by 引用参数是源兼容的更改，但不是二进制兼容的更改：

原始代码：

```
public double CalculateSquare(double value) => value * value;
```

新代码：

```
public double CalculateSquare(in double value) => value * value;
```

[新增功能](#)一文说明了引入会影响公共声明的功能是源兼容的更改还是二进制兼容的更改。

创建记录类型

2021/5/7 • [Edit Online](#)

C# 9 引入了 [记录](#)，这是一种可以创建的新引用类型，而不是类或结构。记录与类不同，区别在于记录类型使用基于值的相等性。两个记录类型的变量在它们的类型和值都相同时，它们是相等的。两个类类型的变量如果引用的对象属于同一类类型并且引用相同的对象，则这两个变量是相等的。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，编译器将生成许多这些成员。

本教程介绍以下操作：

- 决定是否应声明 `class` 或 `record`。
- 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。

必备条件

需要将计算机设置为运行 .NET 5 或更高版本，包括 C# 9.0 或更高版本编译器。自 [Visual Studio 2019 版本 16.8](#) 或 [.NET 5.0 SDK](#) 起，开始随附 C# 9.0 编译器。

记录的特征

通过使用 `record` 关键字（而不是 `class` 或 `struct` 关键字）声明一个类型，可以定义记录。记录是一种引用类型并遵循基于值的相等性语义。为了强制执行值语义，编译器将为记录类型生成多种方法：

- [Object.Equals\(Object\)](#) 的替代。
- 一个虚拟的 `Equals` 方法，其参数为记录类型。
- [Object.GetHashCode\(\)](#) 的替代。
- 用于 `operator ==` 和 `operator !=` 的方法。
- 记录类型实现 [System.IEquatable<T>](#)。

此外，记录还提供 [Object.ToString\(\)](#) 的替代。编译器使用 [Object.ToString\(\)](#) 合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，以后用记录的非破坏性修改。

还可使用更简洁的语法来声明位置记录。声明以下位置记录时，编译器会合成更多方法：

- 主构造函数，它的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 `init-only` 属性。
- 用于从记录中提取属性的 `Deconstruct` 方法。

生成温度数据

数据和统计信息是你要使用记录时所需的内容。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。高温天数越多表示使用空调的时间越多，降温天数越多意味着使用暖气炉的时间越多。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。

此公式基于给定的某一天的平均温度和基准温度。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，我们要创建一个新的应用程序。生成新的控制台应用程序。在名为“`DailyTemperature.cs`”的新文件中创建新的记录类型：

```
public record DailyTemperature(double HighTemp, double LowTemp);
```

上述代码定义了位置记录。你已创建包含两个属性(`HighTemp` 和 `LowTemp`)的引用类型。这些属性是 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。`DailyTemperature` 类型还有一个主构造函数，该构造函数具有两个与这两个属性匹配的参数。使用该主构造函数初始化 `DailyTemperature` 记录：

```
private static DailyTemperature[] data = new DailyTemperature[]
{
    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)
};
```

可将你自己的属性或方法添加到记录，包括位置记录。需要计算每天的平均温度。可将该属性添加到 `DailyTemperature` 记录：

```
public record DailyTemperature(double HighTemp, double LowTemp)
{
    public double Mean => (HighTemp + LowTemp) / 2.0;
}
```

现在需确保你可以使用此数据。将以下代码添加到 `Main` 方法：

```
foreach (var item in data)
    Console.WriteLine(item);
```

运行应用程序，然后你将看到类似于以下显示内容的输出(因空间有限，删除了几行内容)：

```
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。如果希望使用不同的文本，可编写自己的 `ToString` 版本，以防止编译器为你合成一个版本。

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。若要测量一段时间内的采暖，需要忽略平均温度低于基准温度的任何日期。若要测量一段时间内的降温，需要忽略平均温度高于基准温度的任何日期。例如，美国使用 65F 作为采暖和制冷度日数的基准。在此温度下，无需采暖或制冷。如果某一天的平均温度为 70F，则这一天的制冷度日数为 5，采暖度日数为 0。相反，如果某一天的平均温度为 55F，则这一天的采暖度日数为 10，制冷度日数为 0。

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。它们将基准温度和一系列每日温度记录作为主构造函数的参数：

```
public abstract record DegreeDays(double BaseTemperature, IEnumerable<DailyTemperature> TempRecords);

public 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);
}
```

抽象的 `DegreeDays` 记录是 `HeatingDegreeDays` 和 `CoolingDegreeDays` 记录的共享基类。派生记录的主构造函数声明显示了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。基本记录声明并初始化这些属性。派生记录不会隐藏它们，而只会创建和初始化未在其基本记录中声明的参数的属性。在此示例中，派生记录不会添加新的主构造函数参数。通过将以下代码添加到 `Main` 方法来测试代码：

```
var heatingDegreeDays = new HeatingDegreeDays(65, data);
Console.WriteLine(heatingDegreeDays);

var coolingDegreeDays = new CoolingDegreeDays(65, data);
Console.WriteLine(coolingDegreeDays);
```

将获得类似以下显示内容的输出：

```
HeatingDegreeDays { BaseTemperature = 65, TempRecords = record_types.DailyTemperature[], DegreeDays = 85 }
CoolingDegreeDays { BaseTemperature = 65, TempRecords = record_types.DailyTemperature[], DegreeDays = 71.5 }
```

定义编译器合成方法

代码将计算该时间段内正确的采暖和制冷度日数。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你自己的版本的任意编译器合成方法（`clone` 方法除外）。`clone` 方法具有编译器生成的名称，你无法提供不同的实现。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，你需要合成 `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相匹配。

控制台输出中的 `TempRecords` 元素不起作用。它只显示类型。可通过提供自己的合成 `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`，则签名名为 `private bool PrintMembers(StringBuilder builder);`
- 如果记录类型不为 `sealed` 并派生自 `object`（即，它不声明基本记录），则签名名为
`protected virtual bool PrintMembers(StringBuilder builder);`
- 如果记录类型不为 `sealed` 并派生自其他记录，则签名为

```
protected override bool PrintMembers(StringBuilder builder);
```

了解 `PrintMembers` 的目的之后，就可以轻松地理解这些规则。`PrintMembers` 将记录类型中每个属性的相关信息添加到字符串。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：

```
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：

```
protected virtual bool PrintMembers(StringBuilder stringBuilder)
{
    stringBuilder.Append($"BaseTemperature = {BaseTemperature}");
    return true;
}
```

签名声明一个 `virtual protected` 方法来匹配编译器的版本。如果访问器出错，请不要担心；语言会强制执行正确的签名。如果你忘记了任何合成方法的正确修饰符，则编译器会发出警告或错误，帮助你获取正确的签名。

非破坏性修改

位置记录中的合成成员不会修改记录的状态。目的是帮助你更轻松地创建不可变记录。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。位置记录使你可以更轻松地创建不可变引用类型。

创建不可变的引用类型意味着需要使用非破坏性修改。使用 `with 表达式` 创建与现有记录实例类似的新记录实例。这些表达式是一个副本构造，其中包含修改副本的其他赋值。结果是一个新的记录实例，其中每个属性都已从现有记录进行复制并选择性地进行了修改。原始记录未发生更改。

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 41F 作为基准，并测量超出基准的温度。若要使用相同的数据，可创建一条类似于 `coolingDegreeDays` 的新记录，但基准温度不同：

```
// Growing degree days measure warming to determine plant growing rates
var growingDegreeDays = coolingDegreeDays with { BaseTemperature = 41 };
Console.WriteLine(growingDegreeDays);
```

可将计算得出的度数与在较高基准温度下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。在另一种场景中，这是一个优势。对于增长的度日数，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。下面的代码将生成这些累计数据的集合，然后显示这些值：

```
// 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);
}
```

还可使用 `with` 表达式来创建记录的副本。请勿指定 `with` 表达式的大括号之间的任何属性。这意味着将创建一个副本，并且不会更改任何属性：

```
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。记录为引用类型提供了简洁的语法，它的基本用途是存储数据。对于面向对象的类，基本用途是定义责任。本教程重点介绍了位置记录，在这种记录中，你可以使用简洁的语法来声明记录的 `init-only` 属性。编译器会合成记录的多个成员，以复制和比较记录。你可针对记录类型添加所需的任何其他成员。在明确编译器生成的所有成员都不会改变状态的情况下，可以创建不可变的记录类型。可借助 `with` 表达式轻松实现非破坏性修改。

记录提供了另一种定义类型的方法。使用 `class` 定义来创建面向对象的层次结构，这些层次结构重点关注对象的责任和行为。可为数据结构创建 `struct` 类型，这些数据结构可存储数据，并且足够小，以便进行有效复制。当你需要基于值的相等性和比较、不需要复制值以及要使用引用变量时，可以创建 `record` 类型。

若要了解记录的完整说明，可阅读[记录类型的 C# 语言参考文章](#)和[建议的记录类型规范](#)。

教程：在学习过程中，探索使用顶级语句生成代码的想法

2021/5/7 • [Edit Online](#)

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 5，其中包括 C# 9 编译器。自 [Visual Studio 2019 版本 16.9 预览 1](#) 或 [.NET 5.0 SDK](#) 起，开始随附 C# 9 编译器。

本教程假设你熟悉 C# 和 .NET，包括 Visual Studio 或 .NET Core 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：

```
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：

```
using System;

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

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：

```
using System;

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：

```
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：

```
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：

```
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：

```
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：

```
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：

```
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：

```
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：

```
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：

```

using System;
using System.Threading.Tasks;

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：

```
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：

```
using System;
using System.Threading.Tasks;

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：

```
foreach (string s in new[] { "| -", "/ \\", "- |", "\\ /", })  
{  
    Console.WriteLine(s);  
    await Task.Delay(50);  
    Console.WriteLine("\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：

```
using System;  
using MyNamespace;  
  
Console.WriteLine();  
foreach(var s in args)  
{  
    Console.WriteLine(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]);
```

这样就增加了对 `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\(C# 编程指南\)](#)。

使用模式匹配生成类行为以获得更好的代码

2021/5/7 • [Edit Online](#)

C# 中的模式匹配功能可提供语法来表示你的算法。可以使用这些技巧来实现类中的行为。在为真正的对象建模时，可以结合使用面向对象的类设计与面向数据的实现，来提供简洁的代码。

本教程介绍以下操作：

- 使用数据模式表示面向对象的类。
- 使用 C# 的模式匹配功能实现这些模式。
- 利用编译器诊断来验证实现。

先决条件

需要将计算机设置为运行 .NET 5，包括 C# 9.0 编译器。自 [Visual Studio 2019 版本 16.8 预览版](#) 或 [.NET 5.0 SDK 预览版](#) 起，开始随附 C# 9.0 编译器。

生成对运河闸的模拟

本教程将生成一个 C# 类，用于模拟[运河闸](#)。简而言之，运河闸是指船只在两片不同水位的水域间穿行时用于升降它们的设施。一个闸有两个闸门和某种用来更改水位的机制。

在闸正常运转的情况下，闸内的水位与船只驶入侧的水位一致时，船只会进入其中的一个闸门。进入闸内后，水位将发生变化，以与船只驶离闸时所处的水位一致。水位与驶离侧水位一致后，该侧的闸门将打开。通过采用安全措施，可以确保操作人员不会在运河中引发危险情况。只有两个闸门都关闭时，水位才能发生变化。最多只能打开一个闸门。闸内的水位必须与即将打开的闸门外部的水位一致，才能打开该闸门。

可以生成 C# 类来为此行为建模。`CanalLock` 类将支持用于打开或关闭任意一个闸门的命令。它还会包含其他用于升高或降低水位的命令。此类还应支持用于读取两个闸门和水位当前状态的属性。将通过方法实现安全措施。

定义类

将生成一个控制台应用程序，用于测试 `CanalLock` 类。使用 Visual Studio 或 .NET CLI 创建新的 .NET 5 控制台项目。然后，添加一个新类并将其命名为 `CanalLock`。接下来，设计公共 API，但不要实现方法：

```
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}.";
}
```

前面的代码会初始化对象，以便两个闸门都处于关闭状态，并且水位较低。接下来，在 `Main` 方法中编写以下测试代码，以在你生成此类的第一个实现时引导你完成操作：

```

// 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}");
Console.WriteLine(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}");

```

接下来，添加 `CanalLock` 类中每个方法的第一个实现。以下代码将实现类的方法，你无需担心安全规则。稍后将添加安全测试：

```

// 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;
}

```

目前已编写的测试通过。你已实现基本内容。现在，针对第一种故障条件编写测试。前面的测试结束时，两个闸门都已关闭，并且水位设置为较低。添加一个测试，用于尝试打开上闸门：

```

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}");

```

此测试失败，因为该闸门打开了。作为第一个实现，可以使用以下代码修复此问题：

```

// 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");
}

```

测试通过。但是，添加的测试越多，要添加的 `if` 子句也越来越多，并要测试不同的属性。这些方法很快就会变得过于复杂，因为添加了更多的条件语句。

使用模式实现命令

更好的方法是，使用模式确定对象是否处于可执行命令的有效状态。可以表示是否允许将命令作为以下三个变量的函数：闸门状态、水位和新设置：

CCC	CCCC	CC	CC
关闭	关闭	高	关闭
关闭	关闭	低	关闭
关闭	打开	高	打开
已解决	打开	低	已解决
打开	已解决	高	打开
打开	已解决	低	关闭(错误)
打开	打开	高	打开
打开	打开	低	关闭(错误)

表中的第四行和最后一行包含带删除线的文本，因为这些文本无效。现在，要添加的代码应确保上闸门永远不会在水位低时打开。可以将这些状态编码为单个开关表达式（请牢记 `false` 表示“关闭”）：

```
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
};
```

试用此版本。验证代码时测试通过。完整的表格显示了输入与结果的可能组合方式。这意味着，你和其他开发人员可以快速查看此表，确保已包含所有可能的输入。也可以使用编译器来帮助实现这一目的，这样更为简单。添加前面的代码后，可以看到编译器生成以下警告：CS8524 表示开关表达式未包含所有可能的输入。出现该警告的原因是，某个输入为 `enum` 类型。编译器会将“所有可能的输入”解释为基础类型的所有输入，此类型通常为 `int`。此 `switch` 表达式仅检查 `enum` 中声明的值。若要删除此警告，可以为表达式的最后一个分支添加全部捕获弃用模式。此条件会引发异常，因为它表示输入无效：

```
_ => throw new InvalidOperationException("Invalid internal state"),
```

前面的开关分支必须是 `switch` 表达式中的最后一个，因为它与所有输入匹配。通过将它移到序列的较前方进行试验。这将引发编译器错误 CS8510，因为模式中的代码无法访问。开关表达式本身的结构允许编译器针对可能的错误生成错误和警告。借助编译器“安全网格”，可以通过较少的迭代更轻松地生成正确的代码，并可以自由地将开关分支与通配符组合在一起。如果组合生成了意外的不可访问的分支，编译器将发出错误，如果删除了所需的分支，它将发出警告。

第一个更改用于组合命令为关闭闸门的所有分支；这是始终允许的。将以下代码添加为开关表达式中的第一个分支：

```
(false, _, _) => false,
```

添加前面的开关分支后，将收到四个编译器错误，每个命令为 `false` 的分支都有一个错误。新添加的分支已包含这些分支。可以放心地删除这四行。你计划使用这个新的开关分支替换这些条件。

接下来，可以简化命令为打开闸门的四个分支。在水位较高的两种情况下，闸门可以打开。（在其中一种情况下，闸门已打开。）一种水位较低的情况将引发异常，另一种情况不应发生。如果水闸已处于无效状态，引发同样的异常应是安全的。可以对这些分支进行以下简化：

```
(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"),
```

重新运行测试，测试通过。以下是 `SetHighGate` 方法的最终版本：

```

// 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"),
    };
}

```

自行实现模式

现在你已了解了此技巧，接下来请自行填写 `SetLowGate` 和 `SetWaterLevel` 方法。首先，添加以下代码来测试这些方法上的无效操作：

```

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}");

```

重新运行应用程序。可以看到新的测试失败，运河闸进入无效状态。尝试自行实现剩余的方法。用于设置下闸门的方法应类似于用于设置上闸门的方法。用于更改水位的方法包含不同的检查，但应采用相似的结构。将同一过程用于设置水位的方法，你会发现这样很有用。从所有的四个输入开始：两个闸门的状态、水位的当前状态

和请求的新水位。开关表达式应以下列形式开头：

```
CanalLockWaterLevel = (newLevel, CanalLockWaterLevel, LowWaterGateOpen, HighWaterGateOpen) switch
{
    // elided
};
```

要填写 16 个总开关分支。然后进行测试和简化。

你生成的方法与此类似吗？

```
// 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 high gate when
the water is low"),
        _ => 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"),
    };
}
```

你的测试应该可以通过，并且运河闸应安全运转。

总结

在本教程中，你学习了：在对对象的内部状态应用任何更改前，如何使用模式匹配检查该状态。你可以检查属性组合。针对其中任意一种转换生成表格后，测试代码，然后进行简化实现可读性和可维护性。这些初步的重构可能表明需要进一步重构，以验证内部状态或管理其他 API 更改。本教程结合使用了类和对象与更加面向数据的基于模式的方法，来实现这些类。

教程：在 C# 8.0 中使用默认接口方法更新接口

2021/5/7 • [Edit Online](#)

从 .NET Core 3.0 上的 C# 8.0 开始，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 Core，包括 C# 8.0 编译器。自 [Visual Studio 2019 版本 16.3](#) 或 [.NET Core 3.0 SDK](#) 起，开始随附 C# 8.0 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：

```
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：

```
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！这是一项突破性的变革！”C# 8.0 添加了 [默认接口实现](#) 用于升级接口。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。这使其成为用于默认接口成员的完美方案。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：

```
// 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：

```
SampleCustomer c = new SampleCustomer("customer one", new DateTime(2010, 5, 31))  
{  
    Reminders =  
    {  
        { new DateTime(2010, 08, 12), "child'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：

```
// 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

这是一个好的开始。但是，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：

```

// 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：

```

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：

```

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：

```
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，以表达可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。

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

2021/5/7 • [Edit Online](#)

从 .NET Core 3.0 上的 C# 8.0 开始，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。

在本教程中，你将了解：

- 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 Core，包括 C# 8.0 编译器。自 [Visual Studio 2019 版本 16.3](#) 或 [.NET Core 3.0 SDK](#) 或更高版本起，开始随附 C# 8.0 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。

从 C# 8.0 开始，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，默认接口成员是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：

```
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：

```
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：

```
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：

```
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.");
    }
}
```

添加该更改后，`OverheadLight` 类可以通过声明对接口的支持来实现计时器功能：

```
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：

```

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：

```

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：

```

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：

```
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：

```
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：

```

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：

```

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：

```
public enum PowerStatus
{
    NoPower,
    ACPower,
    FullBattery,
    MidBattery,
    LowBattery
}
```

默认实现不使用电源：

```
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。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。

索引和范围

2021/5/7 • • [Edit Online](#)

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。
- 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

此语言支持依赖于两个新类型和两个新运算符：

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

让我们从索引规则开始。请考虑数组 `sequence`。`0` 索引与 `sequence[0]` 相同。`^0` 索引与 `sequence[sequence.Length]` 相同。表达式 `sequence[^0]` 不会引发异常，就像 `sequence[sequence.Length]` 一样。

对于任何数字 `n`，索引 `^n` 与 `sequence[sequence.Length - n]` 相同。

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

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

范围指定范围的开始和末尾。范围是排除的，也就是说“末尾”不包含在范围内。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。将以下代码添加到同一方法中。将其复制并粘贴到交互式窗口的底部。

```
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：

```
string[] lazyDog = words[^2..^0];
foreach (var word in lazyDog)
    Console.WriteLine($"< {word} >");
Console.WriteLine();
```

下面的示例为开始和/或结束创建了开放范围：

```
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.WriteLine($"< {word} >");
Console.WriteLine();
foreach (var word in firstPhrase)
    Console.WriteLine($"< {word} >");
Console.WriteLine();
foreach (var word in lastPhrase)
    Console.WriteLine($"< {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：

```
Index the = ^3;
Console.WriteLine(words[the]);
Range phrase = 1..4;
string[] text = words[phrase];
foreach (var word in text)
    Console.WriteLine($"< {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：

```

int[] numbers = Enumerable.Range(0, 100).ToArray();
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]}");

```

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>](#)。

IMPORTANT

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：

```

var jagged = new int[10][]
{
    new int[10] { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 },
    new int[10] { 10,11,12,13,14,15,16,17,18,19 },
    new int[10] { 20,21,22,23,24,25,26,27,28,29 },
    new int[10] { 30,31,32,33,34,35,36,37,38,39 },
    new int[10] { 40,41,42,43,44,45,46,47,48,49 },
    new int[10] { 50,51,52,53,54,55,56,57,58,59 },
    new int[10] { 60,61,62,63,64,65,66,67,68,69 },
    new int[10] { 70,71,72,73,74,75,76,77,78,79 },
    new int[10] { 80,81,82,83,84,85,86,87,88,89 },
    new int[10] { 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.WriteLine($"{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：

```

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..^(start + 10);
    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)).ToArray();

```

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

2021/5/7 • [Edit Online](#)

C# 8.0 引入了[可为空引用类型](#)，它们以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 Core，包括 C# 8.0 编译器。[Visual Studio 2019](#) 或 [.NET Core 3.0](#) 随附 C# 8.0 编译器。

本教程假设你熟悉 C# 和 .NET，包括 Visual Studio 或 .NET Core 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`。必须选择“可为空引用类型”功能，即使在 C# 8.0 项目中也是如此。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：

```
<Nullable>enable</Nullable>
```

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言。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：

```
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：

```
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：

```

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：

```

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：

```

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：

```

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：

```
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：

```
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：

```

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：

```
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：

```
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：

```
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：

```
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：

```
foreach (var participant in surveyRun.AllParticipants)
{
    Console.WriteLine($"Participant: {participant.Id}");
    if (participant.AnsweredSurvey)
    {
        for (int i = 0; i < surveyRun.Questions.Count; i++)
        {
            var answer = participant.Answer(i);
            Console.WriteLine($"{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

要了解更多信息，请迁移现有应用程序以使用可为空引用类型：

[升级应用程序以使用可为空引用类型](#)

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](#)

教程：使用可为空引用类型迁移现有代码

2021/5/7 • [Edit Online](#)

C# 8 引入了可为空引用类型，它们以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。引用类型的任何现有变量都将被解释为不可为空引用类型。

本教程介绍以下操作：

- 使用代码时启用空引用检查。
- 诊断并更正与 Null 值相关的其他警告。
- 管理可为空启用上下文和可为空禁用上下文之间的接口。
- 控制可为空的批注上下文。

先决条件

需要将计算机设置为运行 .NET Core，包括 C# 8.0 编译器。自 [Visual Studio 2019 版本 16.3](#) 或 [.NET Core 3.0 SDK](#) 起，开始随附 C# 8 编译器。

本教程假设你熟悉 C# 和 .NET，包括 Visual Studio 或 .NET Core CLI。

浏览示例应用程序

即将迁移的示例应用程序是一个 RSS 源阅读器 Web 应用。它从单个 RSS 源中进行读取并显示最新文章的摘要。可以选择任何文章以访问网站。应用程序相对较新，但却是在可以使用可为空引用类型前编写的。应用程序的设计决策代表了合理的原则，但没有利用这一重要的语言功能。

示例应用程序包括验证应用主要功能的单元测试库。如果根据生成的警告更改任何实现，该项目将使安全升级变得更容易。若要下载起始代码，可以访问 [dotnet/samples](#) GitHub 存储库。

迁移项目的目标应该是利用新的语言功能，以便清楚地表达对变量的为 Null 性的意图，并以如下特定方式执行此操作，即在将可为空注释上下文和可为空警告上下文设置为 `enabled` 时，编译器不会生成警告。

将项目升级到 C#8

第一步建议确定迁移任务的范围。首先将项目升级到 C# 8.0(或更高版本)。将 `<LangVersion>` 元素同时添加到 Web 项目和单元测试项目的 csproj 文件中的 PropertyGroup：

```
<LangVersion>8.0</LangVersion>
```

升级语言版本选择 C# 8.0，但不启用可为空注释上下文或可为空警告上下文。重新生成项目，以确保在没有警告的情况下进行生成。

下一步建议打开可为空注释上下文，看看生成了多少警告。在解决方案中的两个 csproj 文件中(直接在 `<LangVersion>` 元素下面)添加以下元素：

```
<Nullable>enable</Nullable>
```

执行测试生成，并注意警告列表。在此小型应用程序中，编译器将生成五个警告，因此可能会启用可为空注释上

下文，并开始为整个项目修复警告。

该策略仅适用于较小的项目。对于任何大型项目，通过为整个代码库启用可为空注释上下文而生成的警告数使得以系统方式修复警告变得更加困难。对于大型企业项目，通常希望一次迁移一个项目。在每个项目中，一次迁移一个类或文件。

警告有助于发现原始设计意图

有两个类生成多个警告。从 `NewsStoryViewModel` 类开始。从两个 csproj 文件中删除 `Nullable` 元素，以便将警告范围限制为正在处理的代码部分。打开 `NewsStoryViewModel.cs` 文件并添加下列指令，以启用 `NewsStoryViewModel` 的可为空注释上下文，并按照类定义对其进行还原：

```
#nullable enable
public class NewsStoryViewModel
{
    public DateTimeOffset Published { get; set; }
    public string Title { get; set; }
    public string Uri { get; set; }
}
(nullable restore)
```

这两条指令可帮助你专注于迁移工作。为你正在积极处理的代码区域生成可为空警告。在做好准备为整个项目启用警告之前，将一直保留这些警告。应使用 `restore` 而不是 `disable` 值，以便在稍后为整个项目启用可为空注释时不会意外禁用上下文。一旦为整个项目启用了可为空注释上下文，就可以从该项目中删除所有 `#nullable` 杂注。

`NewsStoryViewModel` 类是一个数据传输对象 (DTO)，其中两个属性是读/写字符串：

```
public class NewsStoryViewModel
{
    public DateTimeOffset Published { get; set; }
    public string Title { get; set; }
    public string Uri { get; set; }
}
```

这两个属性导致 `CS8618`，“不可为空属性未初始化”。这很清楚：在构造 `NewsStoryViewModel` 时，两个 `string` 属性的默认值都是 `null`。重要的是了解 `NewsStoryViewModel` 对象是如何构造的。查看此类时，无法判断 `null` 值是否是设计的一部分，或者这些对象在创建时是否被设置为非 Null 值。新闻故事在 `NewsService` 类的 `GetNews` 方法中创建：

```
ISyndicationItem item = await feedReader.ReadItem();
var newsStory = _mapper.Map<NewsStoryViewModel>(item);
news.Add(newsStory);
```

前面的代码块中有相当多的内容。此应用程序使用 [AutoMapper](#) NuGet 包从 `ISyndicationItem` 中构造新闻项。你会发现，在这一条语句中构造了新闻故事项，并设置了属性。这意味着 `NewsStoryViewModel` 的设计表明这些属性绝不应该有 `null` 值。这些属性应是不可为空引用类型。这样可以充分表达原始设计意图。实际上，任何都是 `NewsStoryViewModel` 用非 Null 值正确实例化的。这使得以下初始化代码成为一个有效的修复程序：

```
public class NewsStoryViewModel
{
    public DateTimeOffset Published { get; set; }
    public string Title { get; set; } = default!;
    public string Uri { get; set; } = default!;
}
```

`Title` 和 `Uri` 赋值为 `default` (`string` 类型为 `null`) 不会更改程序的运行时行为。`NewsStoryViewModel` 仍然用 `Null` 值构造，但现在编译器不会报告任何警告。`Null` 包容运算符，`default` 表达式后面的 `!` 字符指示编译器前面的表达式不为 `Null`。当其他更改强制对代码库进行更大的更改时，这种方法可能会很方便，但在此应用程序中，有一个相对快速且更好的解决方案：使 `NewsStoryViewModel` 为不可变类型，其中所有属性均在构造函数中设置。对 `NewsStoryViewModel` 进行以下更改：

```
#nullable enable
public class NewsStoryViewModel
{
    public NewsStoryViewModel(DateTimeOffset published, string title, string uri) =>
        (Published, Title, Uri) = (published, title, uri);

    public DateTimeOffset Published { get; }
    public string Title { get; }
    public string Uri { get; }
}
#nullable restore
```

完成更改之后，需要更新配置 AutoMapper 的代码，以便它使用构造函数而不是设置属性。打开 `NewsService.cs`，在文件底部查找以下代码：

```
public class NewsStoryProfile : Profile
{
    public NewsStoryProfile()
    {
        // Create the AutoMapper mapping profile between the 2 objects.
        // ISyndicationItem.Id maps to NewsStoryViewModel.Uri.
        CreateMap<ISyndicationItem, NewsStoryViewModel>()
            .ForMember(dest => dest.Uri, opts => opts.MapFrom(src => src.Id));
    }
}
```

该代码将 `ISyndicationItem` 对象的属性映射到 `NewsStoryViewModel` 属性。希望 AutoMapper 改用构造函数来提供映射。请用以下 automapper 配置替换上面的代码：

```
#nullable enable
public class NewsStoryProfile : Profile
{
    public NewsStoryProfile()
    {
        // Create the AutoMapper mapping profile between the 2 objects.
        // ISyndicationItem.Id maps to NewsStoryViewModel.Uri.
        CreateMap<ISyndicationItem, NewsStoryViewModel>()
            .ForCtorParam("uri", opt => opt.MapFrom(src => src.Id));
    }
}
```

注意，因为此类很小，而且你已经仔细检查过，所以应打开此类声明上面的 `#nullable enable` 指令。对构造函数的更改可能会破坏某些内容，因此有必要在继续之前运行所有测试并测试应用程序。

第一组更改展示了如何发现原始设计指示不应该将变量设置为 `null`。该方法称为“通过构造更正”。在构造对象时，声明该对象及其属性不能为 `null`。编译器的流分析确保这些属性在构造之后不会被设置为 `null`。注意，此构造函数由外部代码调用，而该代码无论是否可为空都能进行调用。新语法不提供运行时检查。外部代码可能会避开编译器的流分析。

其他情况下，类的结构提供了意图的不同线索。打开“Pages”文件夹中的“Error.cshtml.cs”文件。`ErrorViewModel` 包含以下代码：

```
public class ErrorModel : PageModel
{
    public string RequestId { get; set; }

    public bool ShowRequestId => !string.IsNullOrEmpty(RequestId);

    public void OnGet()
    {
        RequestId = Activity.Current?.Id ?? HttpContext.TraceIdentifier;
    }
}
```

在类声明之前添加 `#nullable enable` 指令, 在类声明之后添加 `#nullable restore` 指令。会出现一个警告, 指示 `RequestId` 未初始化。通过查看类, 应确定在某些情况下 `RequestId` 属性应为 Null。`ShowRequestId` 属性的存在表明可能缺失某些值。因为 `null` 有效, 所以在 `string` 类型上添加 `?` 表示 `RequestId` 属性是可为空引用类型。最终类如下所示:

```
#nullable enable
public class ErrorModel : PageModel
{
    public string? RequestId { get; set; }

    public bool ShowRequestId => !string.IsNullOrEmpty(RequestId);

    public void OnGet()
    {
        RequestId = Activity.Current?.Id ?? HttpContext.TraceIdentifier;
    }
}
#nullable restore
```

检查属性的使用情况, 会看到在关联页中, 在标记中呈现属性之前会检查其是否为 Null。这是可为空引用类型的安全使用, 因此已完成此类的检查。

修复 Null 会导致更改

通常情况下, 修复一组警告会在相关代码中创建新的警告。让我们通过修复 `index.cshtml.cs` 类来查看实际警告。打开 `index.cshtml.cs` 文件并检查代码。此文件包含索引页的隐藏代码:

```

public class IndexModel : PageModel
{
    private readonly NewsService _newsService;

    public IndexModel(NewsService newsService)
    {
        _newsService = newsService;
    }

    public string ErrorText { get; private set; }

    public List<NewsStoryViewModel> NewsItems { get; private set; }

    public async Task OnGet()
    {
        string feedUrl = Request.Query["feedurl"];

        if (!string.IsNullOrEmpty(feedUrl))
        {
            try
            {
                NewsItems = await _newsService.GetNews(feedUrl);
            }
            catch (UriFormatException)
            {
                ErrorText = "There was a problem parsing the URL.";
                return;
            }
            catch (WebException ex) when (ex.Status == WebExceptionStatus.NameResolutionFailure)
            {
                ErrorText = "Unknown host name.";
                return;
            }
            catch (WebException ex) when (ex.Status == WebExceptionStatus.ProtocolError)
            {
                ErrorText = "Syndication feed not found.";
                return;
            }
            catch (AggregateException ae)
            {
                ae.Handle((x) =>
                {
                    if (x is XmlException)
                    {
                        ErrorText = "There was a problem parsing the feed. Are you sure that URL is a syndication feed?";
                        return true;
                    }
                    return false;
                });
            }
        }
    }
}

```

添加 `#nullable enable` 指令，将会看到两个警告。`ErrorText` 属性和 `NewsItems` 属性都不会初始化。此类检查会让你认为这两个属性应为可为空引用类型：都具有专用的 setter。在 `OnGet` 方法中正好分配了一个属性。在更改之前，看看这两个属性的使用者。在页面本身，在为任何错误生成标记之前，将检查 `ErrorText` 是否为 Null。检查 `NewsItems` 集合是否为 `null`，并选中以确保集合具有项。快速修复会使这两个属性成为可为空引用类型。更好的解决方法是使集合成为不可为空引用类型，并在检索新闻时向现有集合添加项。第一个解决方法是为 `ErrorText` 的 `string` 类型添加 `?`：

```
public string? ErrorText { get; private set; }
```

此更改不会影响其他代码，因为对 `ErrorText` 属性的任何访问均已通过 Null 检查进行保护。接下来，初始化 `NewsItems` 列表并删除属性资源库，使其成为只读属性：

```
public List<NewsStoryViewModel> NewsItems { get; } = new List<NewsStoryViewModel>();
```

这修复了警告，但引入了错误。`NewsItems` 列表现在是“通过构造更正，但在 `OnGet` 中设置列表的代码必须更改以匹配新的 API。调用 `AddRange` 将新闻项添加到现有列表，而不是赋值：

```
NewsItems.AddRange(await _newsService.GetNews(feedUrl));
```

使用 `AddRange` 而不是赋值意味着 `GetNews` 方法可以返回 `IEnumerable` 而不是 `List`。这节省了一次分配。更改方法的签名，并删除 `ToList` 调用，如下面的代码示例所示：

```
public async Task<IEnumerable<NewsStoryViewModel>> GetNews(string feedUrl)
{
    var news = new List<NewsStoryViewModel>();
    var feedUri = new Uri(feedUrl);

    using (var xmlReader = XmlReader.Create(feedUri.ToString(),
        new XmlReaderSettings { Async = true }))
    {
        try
        {
            var feedReader = new RssFeedReader(xmlReader);

            while (await feedReader.Read())
            {
                switch (feedReader.ElementType)
                {
                    // RSS Item
                    case SyndicationElementType.Item:
                        ISyndicationItem item = await feedReader.ReadItem();
                        var newsStory = _mapper.Map<NewsStoryViewModel>(item);
                        news.Add(newsStory);
                        break;

                    // Something else
                    default:
                        break;
                }
            }
        }
        catch (AggregateException ae)
        {
            throw ae.Flatten();
        }
    }

    return news.OrderByDescending(story => story.Published);
}
```

更改签名也会中断其中一个测试。打开 `SimpleFeedReader.Tests` 项目 `Services` 文件夹中的 `NewsServiceTests.cs` 文件。导航到 `Returns_News_Stories_Given_Valid_Uri` 测试并将 `result` 变量的类型更改为 `IEnumerable<NewsItem>`。更改类型意味着 `Count` 属性不再可用，因此将 `Assert` 中的 `Count` 属性替换为对 `Any()` 的调用：

```
// Act
IEnumerable<NewsStoryViewModel> result =
    await _newsService.GetNews(feedUrl);

// Assert
Assert.True(result.Any());
```

还需要向文件开头添加一个 `using System.Linq` 语句。

这组更改强调了更新包含泛型实例化的代码时需要特别考虑的问题。列表和不可为空类型列表中的元素。其中一个或两者可以是可为空类型。允许所有以下声明：

- `List<NewsStoryViewModel>` : 不可为空视图模型的不可为空列表。
- `List<NewsStoryViewModel?>` : 可为空视图模型的不可为空列表。
- `List<NewsStoryViewModel?>` : 不可为空视图模型的可为空列表。
- `List<NewsStoryViewModel?>?` : 可为空视图模型的可为空列表。

使用外部代码的接口

已经对 `NewsService` 类进行了更改，因此请为该类启用 `#nullable enable` 注释。这不会生成任何新的警告。但是，仔细检查该类有助于说明编译器流分析的一些限制。检查构造函数：

```
public NewsService(IMapper mapper)
{
    _mapper = mapper;
}
```

`IMapper` 参数类型为不可为空引用。它由 ASP.NET Core 基础结构代码调用，因此编译器并不知道 `IMapper` 永远不会为 Null。如果默认的 ASP.NET Core 依赖关系注入 (DI) 容器不能解析必要的服务，就会引发异常，因此代码是正确的。编译器无法验证对公共 API 的所有调用，即使代码是在启用了可为空注释上下文的情况下编译的。此外，尚未选择使用可为空引用类型的项目可能会使用你的库。验证公共 API 的输入，即使已将它们声明为不可为空类型。

获取代码

已经修复了在初始测试编译中标识的警告，因此现在可以为两个项目启用可为空注释上下文。重新生成项目；编译器不会报告任何警告。可以在 [dotnet/samples](#) GitHub 存储库中获取已完成项目的代码。

支持可为空引用类型的新功能可以帮助你发现和修复代码中处理 `null` 值的方式中的潜在错误。启用可为空注释上下文可以表达设计意图：某些变量永远不应为 Null，其他变量可以包含 Null 值。借助这些功能，可以更轻松地声明设计意图。同样，可为空警告上下文指示编译器在违背该意图时发出警告。这些警告指导你进行更新，使代码更具弹性，并减少在执行期间引发 `NullReferenceException` 的可能性。可以控制这些上下文的范围，以便专注于要迁移的代码的本地区域，同时保证其余代码库不受影响。实际上，可以将此迁移任务作为类的常规维护的一部分。本教程演示了迁移应用程序以使用可为空引用类型的过程。可以通过检查 PR [Jon Skeet](#) 来浏览此过程一个更大的实际示例，用于将可为空引用类型合并到 `NodaTime`。此外，还可以在 [Entity Framework Core - 使用可为 null 的引用类型](#) 中学习将可为 null 的引用类型与 Entity Framework Core 结合使用的技术。

教程：使用 C# 8.0 和 .NET Core 3.0 生成和使用异步流

2021/5/7 • [Edit Online](#)

C# 8.0 引入了异步流，这可针对流式处理数据源建模。数据流经常异步检索或生成元素。异步流依赖于 .NET Standard 2.1 中引入的新接口。.NET Core 3.0 以及更高版本支持这些接口。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 Core，包括 C# 8.0 编译器。自 [Visual Studio 2019 版本 16.3](#) 或 [.NET Core 3.0 SDK](#) 起，开始随附 C# 8 编译器。

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。

WARNING

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 Core CLI。

运行初学者应用程序

可以从 [csharp/whats-new/tutorials](#) 文件夹中的 [dotnet/docs](#) 存储库获得本教程中使用的初学者应用程序代码。

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：

```

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：

```

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：

```
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：

```
finalResults.Merge(issues(results)["nodes"]);
progress?.Report(issuesReturned);
cancel.ThrowIfCancellationRequested();
```

将这三行替换为以下代码：

```
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：

```

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：

```

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：

```

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：

```
int num = 0;
var enumerator = runPagedQueryAsync(client, PagedIssueQuery, "docs").GetEnumeratorAsync();
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：

```
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"];
}
```

`EnumeratorCancellationAttribute` 属性导致编译器生成 `IAsyncEnumerator<T>` 的代码，该代码将传递给

`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：

```
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 [csharp/whats-new/tutorials](#) 文件夹中的 [dotnet/docs](#) 存储库获得完成教程的代码。

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。

使用类和对象探索面向对象的编程

2021/5/7 • [Edit Online](#)

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

本教程要求安装一台虚拟机，以用于本地开发。在 Windows、Linux 或 macOS 上，可以使用 .NET CLI 创建、生成和运行应用程序。在 Windows 上，可以使用 Visual Studio 2019。有关创建说明，请参阅[创建本地环境](#)。

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：

```
using System;

namespace classes
{
    class Program
    {
        static void Main(string[] args)
        {
            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***形式的类型来组织代码。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：

```

using System;

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。`BankAccount` 类有五个成员。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。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：

```

public BankAccount(string name, decimal initialBalance)
{
    this.Owner = name;
    this.Balance = initialBalance;
}

```

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)：

```

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。这样做的简单方法是从一个 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：

```
private static int accountNumberSeed = 1234567890;
```

此为数据成员。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：

```
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，实现实存款和取款。与仅更新每笔交易余额相比，这样做有一些优点。历史记录可用于审核所有交易，并管理每日余额。通过在需要时根据所有交易的历史记录计算余额，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。这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：

```
using System;

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)
        {
            this.Amount = amount;
            this.Date = date;
            this.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：

```
private List<Transaction> allTransactions = new List<Transaction>();
```

`List<T>` 类要求导入不同的命名空间。在 `BankAccount.cs` 的开头，添加以下代码：

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

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：

```

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。

这就引入了异常的概念。指明方法无法成功完成工作的标准方法是抛出异常。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` 方法会抛出异常。如果取款金额为负数，或取款后的余额为负数，`MakeWithdrawal` 方法会抛出异常。将以下代码添加到 `allTransactions` 列表的声明后面：

```

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` 语句将引发异常。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：

```
public BankAccount(string name, decimal initialBalance)
{
    this.Number = accountNumberSeed.ToString();
    accountNumberSeed++;

    this.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，对此进行测试：

```
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：

```
// Test that the initial balances must be positive.
try
{
    var invalidAccount = new BankAccount("invalid", -55);
}
catch (ArgumentOutOfRangeException e)
{
    Console.WriteLine("Exception caught creating account with negative balance");
    Console.WriteLine(e.ToString());
}
```

使用 `try` 和 `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` 方法的末尾添加以下代码：

```
// 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：

```
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();
}
```

上面的代码使用 [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：

```
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：

- [If 和 else 语句](#)
- [While 语句](#)
- [Do 语句](#)
- [For 语句](#)

面向对象的编程 (C#)

2021/5/7 • [Edit Online](#)

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：

```
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：

```
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：

```
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：

```
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` 方法中，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：

```
public override void PerformMonthEndTransactions()
{
    if (Balance > 500m)
    {
        var 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：

```

public override void PerformMonthEndTransactions()
{
    if (Balance < 0)
    {
        // Negate the balance to get a positive interest charge:
        var 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：

```

private 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：

```

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：

```

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：

```

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：

```
Unhandled exception. System.ArgumentOutOfRangeException: Amount of deposit must be positive (Parameter 'amount')
  at OOPProgramming.BankAccount.MakeDeposit(Decimal amount, DateTime date, String note) in
BankAccount.cs:line 42
  at OOPProgramming.BankAccount..ctor(String name, Decimal initialBalance) in BankAccount.cs:line 31
  at OOPProgramming.LineOfCreditAccount..ctor(String name, Decimal initialBalance) in
LineOfCreditAccount.cs:line 9
  at OOPProgramming.Program.Main(String[] args) in Program.cs:line 29
```

NOTE

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主体。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：

```
private readonly decimal minimumBalance;

public BankAccount(string name, decimal initialBalance) : this(name, initialBalance, 0) { }

public BankAccount(string name, decimal initialBalance, decimal minimumBalance)
{
    this.Number = accountNumberSeed.ToString();
    accountNumberSeed++;

    this.Owner = name;
    this.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`：

```
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：

```
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：

```
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：

```
public void MakeWithdrawal(decimal amount, DateTime date, string note)
{
    if (amount <= 0)
    {
        throw new ArgumentOutOfRangeException(nameof(amount), "Amount of withdrawal must be positive");
    }
    var overdraftTransaction = CheckWithdrawalLimit(Balance - amount < minimumBalance);
    var withdrawal = new Transaction(-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：

```
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：

```
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。

恭喜，你已完成我们的所有 C# 简介教程。若要了解详细信息，请继续学习更多[教程](#)。

使用字符串内插构造格式化字符串

2021/3/9 • [Edit Online](#)

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：

```
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：

```
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：

```
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：

```
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：

```
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：

```
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：

```
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：

```
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。

```
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：

[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。

C# 中的字符串内插

2020/5/20 • [Edit Online](#)

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](#)。有关 .NET 中设置类型格式的详细信息, 请参阅[设置 .NET 中类型的格式](#)主题。

NOTE

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。

介绍

[字符串插值](#)功能构建在[复合格式设置](#)功能的基础之上, 提供更具有可读性、更方便的语法, 用于将设置了格式的表达式结果包含到结果字符串。

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:

```
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);

// Expected 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:

```
{<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

可通过在内插表达式后接冒号 (":") 和格式字符串来指定受表达式结果类型支持的格式字符串:

```
{<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:

```
var date = new DateTime(1731, 11, 25);
Console.WriteLine($"On {date:dddd, MMMM dd, yyyy} Leonhard Euler introduced the letter e to denote
{Math.E:F5} in a letter to Christian Goldbach.");
// Expected output:
// On Sunday, November 25, 1731 Leonhard Euler introduced the letter e to denote 2.71828 in a letter to
Christian Goldbach.
```

有关详细信息，请参阅[复合格式设置主题的格式字符串组件](#)章节。该部分提供主题链接，这些主题介绍.NET 基类型支持的标准和自定义格式字符串。

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

通过在内插表达式后添加逗号 (",") 和常数表达式来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：

```
{<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：

```
{<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：

```
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}|");

// Expected 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[复合格式设置主题的对齐方式组件](#)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 \$ 字符开头，后跟 @ 字符。从 C# 8.0 开始，可以按任意顺序使用 \$ 和 @ 标记：\$@"..." 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[复合格式设置主题的转义括号](#)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：

```

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.");
var userName = "Jane";
var stringWithEscapes = $"C:\\Users\\{userName}\\Documents";
var verbatimInterpolated = ${@C:\\Users\\{userName}\\Documents"};
Console.WriteLine(stringWithEscapes);
Console.WriteLine(verbatimInterpolated);

// Expected output:
// Find the intersection of the {1, 2, 7, 9} and {7, 9, 12} sets.
// C:\\Users\\Jane\\Documents
// ${@C:\\Users\\Jane\\Documents}

```

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：

```

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。使用内插字符串到 [System.FormattableString](#) 实例的隐式转换，并调用它的 [ToString\(IFormatProvider\)](#) 方法来创建区域性特定的结果字符串。下面的示例演示如何执行此操作：

```

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 = $"{date,20}{number,20:N3}";
foreach (var culture in cultures)
{
    var cultureSpecificMessage = message.ToString(culture);
    Console.WriteLine($"{culture.Name,-10}{cultureSpecificMessage}");
}

// Expected output is like:
// en-US      5/17/18 3:44:55 PM      31,415,926.536
// en-GB      17/05/2018 15:44:55      31,415,926.536
// nl-NL      17-05-18 15:44:55      31.415.926,536
//          05/17/2018 15:44:55      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

可配合 [FormattableString.ToString\(IFormatProvider\)](#) 方法使用静态 [FormattableString.Invariant](#) 方法将内插字符串解析为 [InvariantCulture](#) 的结果字符串。下面的示例演示如何执行此操作：

```
string messageInInvariantCulture = FormattableString.Invariant($"Date and time in invariant culture:  
{DateTime.Now}");  
Console.WriteLine(messageInInvariantCulture);  
  
// Expected output is like:  
// Date and time in invariant culture: 05/17/2018 15:46:24
```

结束语

本教程介绍字符串插值用法的常见方案。有关字符串插值的详细信息，请参阅[字符串插值](#)主题。有关 .NET 中设置类型格式的详细信息，请参阅[设置 .NET 中类型的格式](#)和[复合格式设置](#)主题。

另请参阅

- [String.Format](#)
- [System.FormattableString](#)
- [System.IFormattable](#)
- [字符串](#)

控制台应用

2021/5/7 • • [Edit Online](#)

此教程将介绍 .NET Core 和 C# 语言的许多功能。你将了解：

- .NET Core 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。

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

- 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、Linux、macOS 或 Docker 容器中运行此应用程序。
- 安装最喜爱的代码编辑器。

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，我们先来逐步了解一下如何运行简单的 Hello World 应用程序。创建应用程序之后，在命令提示符处键入 `dotnet restore`。此命令将运行 NuGet 包还原进程。NuGet 是 .NET 程序包管理器。此命令会下载项目缺少的所有依赖项。由于这是一个新项目，尚无任何依赖项，因此首次运行只会下载 .NET Core 框架。执行这一初始步骤后，只需运行 `dotnet restore`，即可添加新的依赖项包，或更新任意依赖项的版本。

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 [Azure DevOps Services 中的持续集成生成](#) 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](#)。

还原包后，运行 `dotnet build`。这将运行生成引擎，并创建应用程序可执行文件。最后，执行 `dotnet run` 来运行应用程序。

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。我们将执行首轮更改。在此文件的最上面，你会看到 `using` 语句：

```
using System;
```

此语句指示编译器，`System` 命名空间中的任何类型都在范围内。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，此程序封闭在名称基于当前目录名的命名空间内。对于本教程，我们将命名空间的名称更改为 `TeleprompterConsole`：

```
namespace TeleprompterConsole
```

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。如果需要有关如何下载本主题示例应用的信息，请参阅[示例和教程](#)主题中的说明。

接下来，在 `Program` 类中添加以下方法（即 `Main` 方法的下方）：

```
static IEnumerable<string> ReadFrom(string file)
{
    string line;
    using (var reader = File.OpenText(file))
    {
        while ((line = reader.ReadLine()) != null)
        {
            yield return line;
        }
    }
}
```

此方法使用两个新命名空间中的类型。为了让编译能够顺利进行，需要在文件的最上面添加以下两行代码：

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

`IEnumerable<T>` 接口是在 `System.Collections.Generic` 命名空间中进行定义。`File` 类是在 `System.IO` 命名空间中进行定义。

这是一种称为“Iterator 方法”的特殊类型 C# 方法。枚举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。`Enumerator` 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。

还有两个 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理。`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` 定义的是隐式类型本地变量。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：

```
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：

```
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：

```
Console.WriteLine();
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();
}
```

由于 `Task` 类位于 `System.Threading.Tasks` 命名空间中，因此需要在文件的最上面添加 `using` 语句：

```
using System.Threading.Tasks;
```

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：

```
var lineLength = 0;
```

然后，在 `yield return word + " ";` 语句后（在右大括号前）添加以下代码：

```
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:

```
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。

可以在 `Main` 方法中调用以下新方法:

```
ShowTeleprompter().Wait();
```

此时，在 `Main` 中，代码确实是同步等待。应尽可能使用 `await` 运算符，而不是采用同步等待的方式。不过，在控制台应用程序的 `Main` 方法中，不能使用 `await` 运算符。这会导致应用程序在所有任务完成前退出。

NOTE

如果使用 C# 7.1 或更高版本，则可以使用 `async Main` 方法创建控制台应用程序。

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:

```
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整个文件的标志 `Done`：

```
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;
        }
    }
}
```

将此类放入新文件，并将此类封闭在 `TeleprompterConsole` 命名空间中，如上所示。还需要添加 `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。这与一直使用的 `using` 语句相反，后者导入命名空间中的所有类。

```
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：

```
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：

```
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 `TeleprompterConfig` 类中调用新方法。现在，需要将 `Main` 更新为调用 `RunTeleprompter` (而不是 `ShowTeleprompter`)：

```
RunTeleprompter().Wait();
```

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 Core CLI。

有关文件 I/O 的详细信息，请参阅[文件和流 I/O](#) 主题。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](#)主题和[异步编程](#)主题。

教程：在 .NET 控制台应用使用 C# 发出 HTTP 请求

2021/5/10 • [Edit Online](#)

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 5.0 或更高版本](#)
- 代码编辑器例如 [Visual Studio Code](#) 为开放源、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：

```
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。

```
cd WebAPIClient
```

```
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](#)。

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. 打开项目目录中的 `Program.cs` 文件，然后向 `Program` 类添加下列异步方法：

```
private static async Task ProcessRepositories()
{
}
```

2. 在“Program.cs”文件顶部添加 `using` 指令，以便 C# 编译器识别 `Task` 类型：

```
using System.Threading.Tasks;
```

如果此时运行 `dotnet build`，编译将会成功，但会警告此方法不含任何 `await` 运算符，因此会同步运行。将在填写方法时添加 `await` 运算符。

3. 将 `Main` 方法替换为以下代码：

```
static async Task Main(string[] args)
{
    await ProcessRepositories();
}
```

此代码：

- 通过添加 `async` 修饰符并将返回类型更改为 `Task`，更改 `Main` 的签名。
- 将 `Console.WriteLine` 语句替换为调用使用 `await` 关键字的 `ProcessRepositories`。

4. 在 `Program` 类中创建 `HttpClient` 静态实例来处理请求和响应。

```
namespace WebAPIClient
{
    class Program
    {
        private static readonly HttpClient client = new HttpClient();

        static async Task Main(string[] args)
        {
            //...
        }
    }
}
```

5. 在 `ProcessRepositories` 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：

```
private static async Task ProcessRepositories()
{
    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 stringTask = client.GetStringAsync("https://api.github.com/orgs/dotnet/repos");

    var msg = await stringTask;
    Console.Write(msg);
}
```

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。
- 调用 `HttpClient.GetStringAsync(String)` 发出 web 请求并检索响应。此方法启动发出 web 请求的任务。当请求返回时，该任务读取响应流并从中提取内容。响应正文以 `String` 形式返回，任务结束时可用。
- 等待响应字符串的任务并打印控制台响应。

6. 将两个 `using` 指令添加到文件顶部：

```
using System.Net.Http;
using System.Net.Http.Headers;
```

7. 生成并运行应用。

```
dotnet run
```

因为 `ProcessRepositories` 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“repo.cs”的文件并添加以下代码：

```
using System;

namespace WebAPIClient
{
    public class Repository
    {
        public string name { get; set; }
    }
}
```

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，可更容易地创建仅适用于 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 `ProcessRepositories` 方法中 `GetStringAsync(String)` 的调用：

```
var streamTask = client.GetStreamAsync("https://api.github.com/orgs/dotnet/repos");
var repositories = await JsonSerializer.DeserializeAsync<List<Repository>>(await streamTask);
```

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` 类，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：

```
var msg = await stringTask;
Console.WriteLine(msg);
```

使用以下代码：

```
foreach (var repo in repositories)
    Console.WriteLine(repo.name);
```

4. 在文件顶部添加以下 `using` 指令：

```
using System.Collections.Generic;
using System.Text.Json;
```

5. 运行应用。

```
dotnet run
```

输出是 .NET Foundation 中的存储库名称列表。

配置反序列化

1. 在“repo.cs”中将 `name` 属性更改为 `Name` 并添加 `[JsonPropertyName]` 属性指定该属性如何出现在 JSON 中。

```
[JsonPropertyName("name")]
public string Name { get; set; }
```

2. 将 `System.Text.Json.Serialization` 命名空间添加到 `using` 指令：

```
using System.Text.Json.Serialization;
```

3. 在“Program.cs”中，更新代码以使用首字母大写的 `Name` 属性：

```
Console.WriteLine(repo.Name);
```

4. 运行应用。

输出相同。

重构代码

`ProcessRepositories` 方法可以执行异步工作，并返回一组存储库。更改该此方法返回 `List<Repository>`，并将用于写入信息的代码移到 `Main` 方法中。

1. 更改 `ProcessRepositories` 的签名，以返回可生成 `Repository` 对象列表的任务：

```
private static async Task<List<Repository>> ProcessRepositories()
```

2. 处理 JSON 响应后返回存储库：

```
var streamTask = client.GetStreamAsync("https://api.github.com/orgs/dotnet/repos");
var repositories = await JsonSerializer.DeserializeAsync<List<Repository>>(await streamTask);
return repositories;
```

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 `Main` 方法以捕获结果并将每个存储库名称写入控制台。现在，`Main` 方法如下所示：

```
public static async Task Main(string[] args)
{
    var repositories = await ProcessRepositories();

    foreach (var repo in repositories)
        Console.WriteLine(repo.Name);
}
```

4. 运行应用。

输出相同。

反序列化更多属性

以下步骤添加代码用以处理收到的 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. 将以下属性添加到 `Repository` 类定义：

```
[JsonPropertyName("description")]
public string Description { get; set; }

[JsonPropertyName("html_url")]
public Uri GitHubHomeUrl { get; set; }

[JsonPropertyName("homepage")]
public Uri Homepage { get; set; }

[JsonPropertyName("watchers")]
public int Watchers { get; set; }
```

`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. 更新 `Main` 方法以显示属性值：

```
foreach (var repo in repositories)
{
    Console.WriteLine(repo.Name);
    Console.WriteLine(repo.Description);
    Console.WriteLine(repo.GitHubHomeUrl);
    Console.WriteLine(repo.Homepage);
    Console.WriteLine(repo.Watchers);
    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：

```
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“repo.cs”中添加 `public` 属性（用 UTC 表示日期和时间）和 `LastPush` `readonly` 属性（返回转换为当地时间的日期）：

```
[JsonPropertyName("pushed_at")]
public DateTime LastPushUtc { get; set; }

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。省略 `set` 访问器是一种在 C# 中定义“只读”属性的方式。（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：

```
Console.WriteLine(repo.LastPush);
```

3. 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。

C# 和 .NET 中的继承

2020/11/2 • [Edit Online](#)

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

本教程假定你已安装 .NET Core SDK。请访问 [.NET Core 下载](#)页进行下载。还需要安装代码编辑器。此教程使用 [Visual Studio Code](#)，但你可以选择使用任何代码编辑器。

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 [Azure DevOps Services 中的持续集成生成](#)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](#)。

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。”

```
using System;

public class A
{
    private int value = 10;

    public class B : A
    {
        public int GetValue()
        {
            return this.value;
        }
    }
}

public class C : A
{
//    public int GetValue()
//    {
//        return this.value;
//    }
}

public class Example
{
    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。

```

public class A
{
    public void Method1()
    {
        // Method implementation.
    }
}

public class B : A
{ }

public class Example
{
    public static void Main()
    {
        B b = new B();
        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。如果尝试重写非虚成员（如以下示例所示），则会生成编译器错误 CS0506：“<member> 无法重写继承的成员 <member>，因为继承的成员没有 `virtual`、`abstract` 或 `override` 标记。”

```

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。

```

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。因为这些规则的存在，如果尝试编译以下代码，则会生成编译器错误 CS0527：“接口列表中的类型“ValueType”不是接口。”此错误消息指明，尽管可以

定义结构实现的接口，但不支持继承。

```
using System;

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：

```
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。

```

using System;
using System.Reflection;

public class Example
{
    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 (var 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";
            }
            var 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。

```
using System;

public class SimpleClass
{}

public class Example
{
    public static void Main()
    {
        SimpleClass sc = new SimpleClass();
        Console.WriteLine(sc.ToString());
    }
}

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

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	基类
class	<code>Object</code>
struct	<code>ValueType, Object</code>
enum	<code>Enum, ValueType, Object</code>
delegate	<code>MulticastDelegate, Delegate, Object</code>

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。

NOTE

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。

```

using System;

public class Automobile
{
    public Automobile(string make, string model, int year)
    {
        if (make == null)
            throw new ArgumentNullException("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("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。

```

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。”如果尝试使用反射来实例化类，则反射方法会抛出 `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 `abstract`，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。如果尝试从密封类派生，则会生成编译器错误 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：

```
using System;

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("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;
    this.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：

```
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

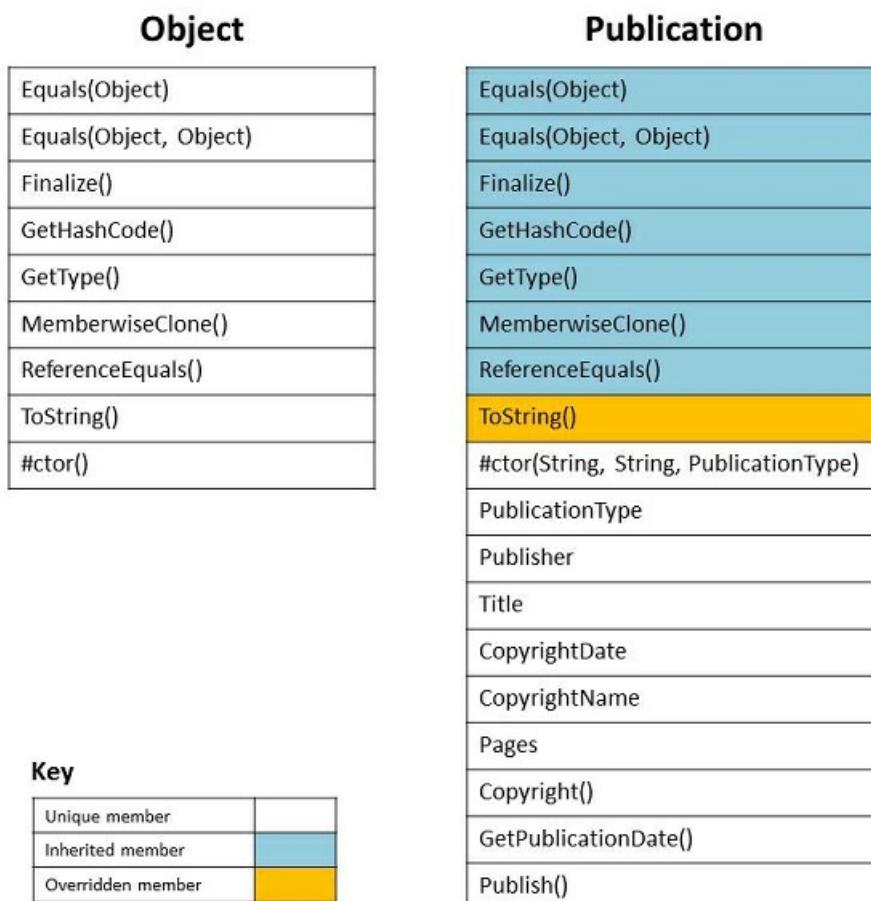
`Copyright` 方法需要将版权所有者的姓名和版权授予年份用作参数，并将它们分配给属性 `CopyrightName` 和 `CopyrightDate`。

- 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。

```

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.");
            ulong nISBN = 0;
            if (! UInt64.TryParse(isbn, out nISBN))
                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("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)
    {
        Book book = obj as Book;
        if (book == null)
            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 `Publication` 构造函数的相应参数。第三个参数是 `author`, 存储在不可变的 `Author` 属性中。其中一个构造函数包含存储在 `ISBN` 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。

Publication

Equals(Object)
Equals(Object, Object)
Finalize()
GetHashCode()
GetType()
MemberwiseClone()
ReferenceEquals(Object, Object)
ToString()
#ctor(String, String, PublicationType)
PublicationType
Publisher
Title
CopyrightDate
CopyrightName
Pages
Copyright()
GetPublicationDate()
Publish()

Book

Equals(Object)
Equals(Object, Object)
Finalize()
GetHashCode()
GetType()
MemberwiseClone()
ReferenceEquals(Object, Object)
ToString()
#ctor(String, String, String)
#ctor(String, String, String, String)
PublicationType
Publisher
Author
Title
CopyrightDate
CopyrightName
ISBN
Pages
Price
Currency
Copyright()
GetPublicationDate()
Publish()
SetPrice()

Key

Unique member	
Inherited member	
Overridden member	

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。

```

using System;
using static System.Console;

public class Example
{
    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");
        WriteLine($"{book.Title} and {book2.Title} are the same publication: " +
                 $""\{((Publication) book).Equals(book2)}\"");
    }

    public static void ShowPublicationInfo(Publication pub)
    {
        string pubDate = pub.GetPublicationDate();
        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，圆和三角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。

```
using System;

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：`Triangle`、`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`)。

```

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。

```
using System;

public class Example
{
    public static void Main()
    {
        Shape[] shapes = { new Rectangle(10, 12), new Square(5),
                           new Circle(3) };
        foreach (var shape in shapes) {
            Console.WriteLine($"{shape}: area, {Shape.GetArea(shape)}; " +
                $"perimeter, {Shape.GetPerimeter(shape)}");
            var rect = shape as Rectangle;
            if (rect != null) {
                Console.WriteLine($"    Is Square: {rect.IsSquare()}, Diagonal: {rect.Diagonal}");
                continue;
            }
            var sq = shape as Square;
            if (sq != null) {
                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
```

请参阅

- [继承\(C# 编程指南\)](#)

使用语言集成查询 (LINQ)

2021/5/7 • [Edit Online](#)

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：

```
// 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：

```

// 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：

```

// 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：

```
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。

```
C:\>dotnet run
{ Suit = clubs, Rank = two }
{ Suit = clubs, Rank = three }
{ Suit = clubs, Rank = four }
{ Suit = clubs, Rank = five }
{ Suit = clubs, Rank = six }
{ Suit = clubs, Rank = seven }
{ Suit = clubs, Rank = eight }
{ Suit = clubs, Rank = nine }
{ Suit = clubs, Rank = ten }
{ Suit = clubs, Rank = jack }
{ Suit = clubs, Rank = queen }
{ Suit = clubs, Rank = king }
{ Suit = clubs, Rank = ace }
{ Suit = diamonds, Rank = two }
{ Suit = diamonds, Rank = three }
{ Suit = diamonds, Rank = four }
{ Suit = diamonds, Rank = five }
{ Suit = diamonds, Rank = six }
{ Suit = diamonds, Rank = seven }
{ Suit = diamonds, Rank = eight }
{ Suit = diamonds, Rank = nine }
{ Suit = diamonds, Rank = ten }
```

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：

```
// 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：

```
// 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：

```
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：

```
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：

```
// 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：

```
public static bool SequenceEquals<T>
    (this IEnumerable<T> first, IEnumerable<T> second)
{
    var firstIter = first.GetEnumerator();
    var secondIter = second.GetEnumerator();

    while (firstIter.MoveNext() && secondIter.MoveNext())
    {
        if (!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：

```

// 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：

```
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，在诸如此类的用例中，这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，你就解决问题。

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。

```
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：

```
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：

```

// 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：

```

public static void Main(string[] args)
{
    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](#)

在 C# 中使用属性

2020/11/2 • [Edit Online](#)

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。

以 `[Obsolete]` 特性为例。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。

此教程将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 Core 中的特性。

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、macOS 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

至此，你已安装所有工具，是时候新建 .NET Core 应用程序了。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：

```
dotnet new console
```

此命令将创建基本的 .NET Core 项目文件。需要执行 `dotnet restore` 来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 [Azure DevOps Services 中的持续集成生成](#) 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。

如何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。它用于示意代码已过时，不得再使用。可以将此特性应用于类（比如说，使用方括号）。

```
[Obsolete]
public class MyClass
{
}
```

请注意，虽然此类的名称为 `ObsoleteAttribute`，但只需在代码中使用 `[Obsolete]`。这是 C# 遵循一项约定。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。为此，可将字符串参数传递给 `Obsolete` 特性。

```
[Obsolete("ThisClass is obsolete. Use ThisClass2 instead.")]
public class ThisClass
{}
```

此字符串会作为自变量传递给 `ObsoleteAttribute` 构造函数，就像在编写

```
var attr = new ObsoleteAttribute("some string")
```

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。

如何创建你自己的特性

创建特性与从 `Attribute` 基类继承一样简单。

```
public class MySpecialAttribute : Attribute
{
}
```

执行上述操作后，我现在可以在基本代码中的其他位置使用 `[MySpecial]`（或 `[MySpecialAttribute]`）特性。

```
[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（此教程稍后将对此进行详细介绍）。

这里要注意的是“gotcha”。如上所述，使用特性时，只允许将某些类型的参数作为自变量传递。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，我使用可正常编译的构造函数创建了特性。

```
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。

```
[Gotcha(new Foo(), "test")] // does not compile
public class AttributeFail
{}
```

上述做法会导致生成编译器错误，如

```
Attribute constructor parameter 'myClass' has type 'Foo', which is not a valid attribute parameter type
```

如何限制特性使用

特性可用于多个“目标”。上述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!

```
[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, 则会看到编译器错误, 如

```
Attribute 'MyAttributeForClassAndStructOnly' is not valid on this declaration type. It is only valid on
'class, struct' declarations
```

```
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。

若要查找并使用特性, 通常需要使用[反射](#)。我不会在此教程中深入介绍反射, 但基本概念就是借助反射, 可以在 C# 中编写用于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;`):

```
TypeInfo typeInfo = typeof(MyClass).GetTypeInfo();
Console.WriteLine("The assembly qualified name of MyClass is " + typeInfo.AssemblyQualifiedName);
```

此代码的打印输出如下:

```
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 `MethodInfo`、`FieldInfo` 等)后, 就可以使用 `GetCustomAttributes` 方法了。这将返回一组 `Attribute` 对象。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) 的 `MemberInfo` 实例使用 `GetCustomAttributes`。

```
var attrs = typeInfo.GetCustomAttributes();
foreach(var attr in attrs)
    Console.WriteLine("Attribute on MyClass: " + attr.GetType().Name);
```

这将在控制台中打印输出 `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` 将返回两个不同的 `ObsoleteAttribute` 实例。

基类库 (BCL) 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, 那么 C# 编译器将删除对该方法(而不是方法本身)的所有调用。此特性通常用于调试(诊断)目的。
- `[CallerMemberName]`. 此特性可应用于参数, 位于 `System.Runtime.CompilerServices` 命名空间中。可用于插入正在调用另一方法的方法的名称。此特性通常用于在各种 UI 框架中实现 `INotifyPropertyChanged` 时清除“神奇字符串”。示例:

```
public class MyUIClass : INotifyPropertyChanged
{
    public event PropertyChangedEventHandler PropertyChanged;

    public void RaisePropertyChanged([CallerMemberName] string propertyName = null)
    {
        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。这样既有助于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。

总结

属性为 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。

教程：使用模式匹配来生成类型驱动和数据驱动的算法

2021/5/7 • [Edit Online](#)

C# 7 引入了基本模式匹配功能。C# 8 和 C# 9 通过新增表达式和模式，扩展了这些功能。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

需要将计算机设置为运行 .NET 5，其中包括 C# 9 编译器。自 [Visual Studio 2019 版本 16.9 预览 1](#) 或 [.NET 5.0 SDK](#) 起，开始随附 C# 9 编译器。

本教程假设你熟悉 C# 和 .NET，包括 Visual Studio 或 .NET Core 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：

```

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，可利用的 `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。

```
using System;
using CommercialRegistration;
using ConsumerVehicleRegistration;
using LiveryRegistration;

namespace toll_calculator
{
    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类型的 `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 `NUL` 对象。

可使用 `Program.cs` 中的以下代码来测试上面的代码:

```

using System;
using CommercialRegistration;
using ConsumerVehicleRegistration;
using LiveryRegistration;

namespace toll_calculator
{
    class Program
    {
        static void Main(string[] args)
        {
            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：

```

vehicle switch
{
    Car {Passengers: 0}      => 2.00m + 0.50m,
    Car {Passengers: 1}      => 2.0m,
    Car {Passengers: 2}      => 2.0m - 0.50m,
    Car c                   => 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：

```

vehicle switch
{
    // ...

    Taxi {Fares: 0}  => 3.50m + 1.00m,
    Taxi {Fares: 1}  => 3.50m,
    Taxi {Fares: 2}  => 3.50m - 0.50m,
    Taxi t           => 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：

```

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 b => 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：

```

vehicle switch
{
    // ...

    DeliveryTruck t when (t.GrossWeightClass > 5000) => 10.00m + 5.00m,
    DeliveryTruck t when (t.GrossWeightClass < 3000) => 10.00m - 2.00m,
    DeliveryTruck t => 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）。完成后的代码如以下代码所示：

```

vehicle switch
{
    Car {Passengers: 0}      => 2.00m + 0.50m,
    Car {Passengers: 1}      => 2.0m,
    Car {Passengers: 2}      => 2.0m - 0.50m,
    Car c                   => 2.00m - 1.0m,

    Taxi {Fares: 0}  => 3.50m + 1.00m,
    Taxi {Fares: 1}  => 3.50m,
    Taxi {Fares: 2}  => 3.50m - 0.50m,
    Taxi t           => 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))
};

```

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：

```

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。如果 `if` 和 `else` 语句使用以下代码表达此内容，则可以使用系列：

```
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：

星期	时段	方向	倍数
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
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：

```
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：

```

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：

```

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：

```

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：

```
(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：

```
(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：

```
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：

```
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](#)

类型 (C# 编程指南)

2021/5/10 • [Edit Online](#)

类型、变量和值

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、参数数量以及类型和种类(值、引用或输出)。.NET 类库定义了一组内置数值类型以及表示各种逻辑构造的更复杂类型(如文件系统、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。
- 在运行时分配变量内存的位置。
- 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：

```
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;
```

NOTE

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：

```
// 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：

```
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` 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来表示整数、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

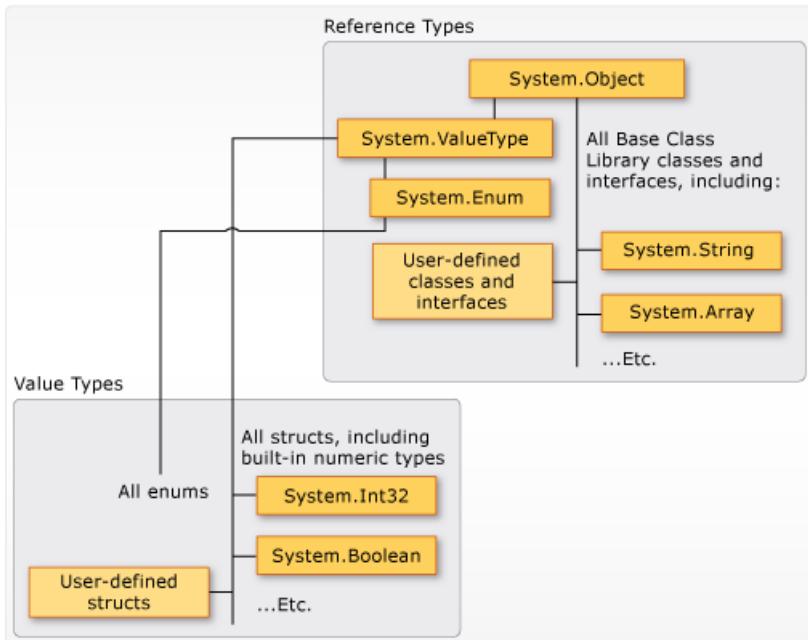
可以使用[结构](#)、[类](#)、[接口枚举](#)和[记录](#)构造来创建你自己的自定义类型。[.NET](#) 类库本身就是 Microsoft 提供的一组自定义类型，以供你在自己的应用程序中使用。默认情况下，类库中最常用的类型在任何 C# 程序中均可用。对于其他类型，只有在显式添加对定义这些类型的程序集的项目引用时才可用。编译器引用程序集之后，你可以声明在源代码的此程序集中声明的类型的变量(和常量)。有关详细信息，请参阅[.NET 类库](#)。

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# 关键字：`int` 等内置数值类型)最终都派生自单个基类型，即 `System.Object`(C# 关键字：`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。



NOTE

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。

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它们的值，这意味着在声明变量的任何上下文中内联分配内存。对于值类型变量，没有单独的堆分配或垃圾回收开销。

值类型分为两类：[结构](#)和[枚举](#)。

内置的数值类型是结构，它们具有可访问的字段和方法：

```
// 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：

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

值类型已密封，这意味着不能从任何值类型(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它实现的任何接口类型；强制转换会导致装箱操作发生，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：

```
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](#)。

另一种值类型是枚举。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：

```
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

定义为类、记录、委托、数组或接口的类型是引用类型。在运行时，当声明引用类型的变量时，该变量会一直包含值 `null`，直至使用 `new` 运算符显式创建对象，或者为该变量分配已经在其他位置使用 `new` 创建的对象，如下所示：

```
MyClass mc = new MyClass();
MyClass mc2 = mc;
```

接口必须与实现它的类对象一起初始化。如果 `MyClass` 实现 `IMyInterface`，则按以下示例所示创建 `IMyInterface` 的实例：

```
IMyInterface iface = new MyClass();
```

创建对象后，内存会在托管堆上进行分配，并且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 CLR 自动内存管理功能（称为“垃圾回收”）回收内存时都会产生开销。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。虽然数组隐式派生自 `System.Array` 类，但可以使用 C# 提供的简化语法声明和使用数组，如以下示例所示：

```
// 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，而其他类可以从你的类继承并重写虚拟方法。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，若要将值 4.56 指定为应按浮点值处理，请在数字后面追加“f”或“F”：`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：

```
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，作为客户端代码在创建类型实例时将提供的实际类型（具体类型）的占位符的类型。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，例如字符串：

```
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

如前所述，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 `?`，创建可以为 `null` 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，可以为 `null` 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：

```
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：

```
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：

- [强制转换和类型转换](#)
- [装箱和取消装箱](#)
- [使用类型 dynamic](#)
- [值类型](#)
- [引用类型](#)
- [类、结构和记录](#)
- [匿名类型](#)
- [泛型](#)

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。

请参阅

- [C# 参考](#)
- [C# 编程指南](#)
- [XML 数据类型转换](#)
- [整型类型](#)

可为空引用类型

2021/5/7 • [Edit Online](#)

C#8.0 引入了“可为空引用类型”和“不可为空引用类型”，使你能够对引用类型变量的属性作出重要声明：

- 引用不应为 null。当变量不应为 null 时，编译器会强制执行规则，以确保在不首先检查它们是否为 null 的情况下，取消引用这些变量是安全的：
 - 必须将变量初始化为非 null 值。
 - 变量永远不能赋值为 `null`。
- 引用可为 null。当变量可以为 null 时，编译器会强制执行不同的规则以确保你已正确检查空引用：
 - 只有当编译器可以保证该值不为 null 时，才可以取消引用该变量。
 - 这些变量可以使用默认的 `null` 值进行初始化，也可以在其他代码中赋值为 `null`。

在 C# 的早期版本中，无法从变量声明中确定设计意图，与处理引用变量相比，这个新功能提供了显著的好处。编译器不提供针对引用类型的空引用异常的安全性：

- 引用可为 null。将引用类型变量初始化为 `null` 或稍后将其指定为 `null` 时，编译器不会发出警告。在没有进行 null 检查的情况下取消引用这些变量，编译器会发出警告。
- 假定引用不为 null。当引用类型被取消引用时，编译器不会发出任何警告。如果将变量设置为可以为 null 的表达式，则编译器会发出警告。

将在编译时发出这些警告。编译器不会在可为 null 的上下文中添加任何 null 检查或其他运行时构造。在运行时，可为 null 的引用和不可为 null 的引用是等效的。

通过添加可为空引用类型，你可以更清楚地声明你的意图。`null` 值是表示变量不引用值的正确方法。请勿使用此功能从代码中删除所有 `null` 值。相反，应该向编译器和其他读取代码的开发人员声明你的意图。通过声明意图，编译器会在你编写与该意图不一致的代码时通知你。

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`：

```
string? name;
```

未将 `?` 附加到类型名称的任何变量都是“不可为 null 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。

编译器使用静态分析来确定可为空引用是否为非 null。如果你在一个可为空引用可能是 null 时对其取消引用，编译器将向你发出警告。可以通过使用 [NULL 包容运算符](#) `!` 后跟变量名称来替代此行为。例如，若知道 `name` 变量不为 null 但编译器仍发出警告，则可以编写以下代码来覆盖编译器的分析：

```
name!.Length;
```

类型为 Null 性

任何引用类型都可以具有四个“为 Null 性”中的一个，它描述了何时生成警告：

- **不可为空**: 无法将 null 分配给此类型的变量。在取消引用之前，无需对此类型的变量进行 null 检查。
- **可为空**: 可将 null 分配给此类型的变量。在不首先检查 `null` 的情况下取消引用此类型的变量时发出警告。
- **无视**: “无视”是 C# 8.0 之前版本的状态。可以取消引用或分配此类型的变量而不发出警告。

- 未知：“未知”通常针对类型参数，其中约束不告知编译器类型是否必须是“可为 null”或“不可为 null”。

变量声明中类型的为 Null 性由声明变量的“可为空上下文”控制。

可为空上下文

可为空上下文可以对编译器如何解释引用类型变量进行精细控制。可以启用或禁用任何给定源代码行的“可为空注释上下文”。可以将 C# 8.0 之前的编译器视为在禁用的可为空上下文中编译所有代码：任何引用类型都可以为空。还可以启用或禁用可为空警告上下文。可为空警告上下文指定编译器使用其流分析生成的警告。

可以使用 .csproj 文件中的 `Nullable` 元素为项目设置可为空注释上下文和可为空警告上下文。此元素配置编译器如何解释类型的为 Null 性以及生成哪些警告。有效设置如下：

- `enable`：“启用”可为空注释上下文。“启用”可为空警告上下文。
 - 引用类型的变量，例如 `string` 是“不可为空”。启用所有为 Null 性警告。
- `warnings`：“禁用”可为空注释上下文。“启用”可为空警告上下文。
 - 引用类型的变量是“无视”。启用所有为 Null 性警告。
- `annotations`：“启用”可为空注释上下文。“禁用”可为空警告上下文。
 - 引用类型的变量（例如字符串）不可为 null。禁用所有为 Null 性警告。
- `disable`：“禁用”可为空注释上下文。“禁用”可为空警告上下文。
 - 引用类型的变量是“无视”，就像早期版本的 C# 一样。禁用所有为 Null 性警告。

示例：

```
<Nullable>enable</Nullable>
```

你还可以使用指令在项目的任何位置设置这些相同的上下文：

- `#nullable enable`：将可为空注释上下文和可为空警告上下文设置为“已启用”。
- `#nullable disable`：将可为空注释上下文和可为空警告上下文设置为“已禁用”。
- `#nullable restore`：将可为空注释上下文和可为空警告上下文还原到项目设置。
- `#nullable disable warnings`：将可为空警告上下文设置为“已禁用”。
- `#nullable enable warnings`：将可为空警告上下文设置为“已启用”。
- `#nullable restore warnings`：将可为空警告上下文还原到项目设置。
- `#nullable disable annotations`：将可为空注释上下文设置为“禁用”。
- `#nullable enable annotations`：将可为空注释上下文设置为“启用”。
- `#nullable restore annotations`：将注释警告上下文还原到项目设置。

IMPORTANT

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”：

- 在 .editorconfig 中，在应用于该文件的部分中指定 `generated_code = true`。
- 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。
- 文件名以 `TemporaryGeneratedFile_` 开头
- 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，可为空注释和警告上下文处于禁用状态，包括新项目。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。

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。

可为空注释上下文

编译器在已禁用的可为空注释上下文中使用以下规则：

- 不能在已禁用的上下文中声明可为空引用。
- 可以为所有引用变量分配 null 值。
- 取消引用引用类型的变量时不会生成警告。
- 可能不会在禁用的上下文中使用 null 包容运算符。

该行为与以前版本的 C# 相同。

编译器在已启用的可为空注释上下文中使用以下规则：

- 引用类型的任何变量都是“不可为空引用”。
- 任何不可为空引用都可以安全地取消引用。
- 任何可为空引用类型(在变量声明中的类型之后由 `?` 标记)可为 null。静态分析确定在取消引用该值时是否已知该值不为 null。否则，编译器会发出警告。
- 你可以使用 null 包容运算符声明可为空引用不为 null。

在已启用的可为空注释上下文中，附加到引用类型的 `?` 字符声明“可为空引用类型”。可将 NULL 包容运算符 `!` 附加到表达式以声明表达式不为 NULL。

可空警告上下文

可空警告上下文与可为空注释上下文不同。即使禁用新注释，也可以启用警告。编译器使用静态流分析来确定任何引用的“null 状态”。当“可空警告上下文”未被“禁用”时，null 状态为“非 null”或“可能为 null”。如果在编译器确定引用“可能为 null”时取消引用该引用，编译器会向你发出警告。除非编译器可以确定以下两个条件之一，否则引用的状态为“可能为 null”：

1. 该变量已明确分配给非 null 值。
2. 在取消引用之前，已检查变量或表达式是否为 null。

在可为 null 警告上下文中取消引用“可能为 null”的变量或表达式时，编译器会生成警告。此外，在将不可为 null 引用类型变量分配给已启用的可为空注释上下文中的可能为 null 变量或表达式时，编译器会生成警告。

属性描述 API

可以向 API 添加属性，以向编译器提供有关参数或返回值何时可以为 null 或不可为 null 的更多信息。可在涉及[可为 null 属性](#)的语言参考中的文章中了解有关这些属性的更多信息。这些属性将通过当前和即将发布的版本添加到 .NET 库中。首先更新最常用的 API。

已知缺陷

包含引用类型的数组和结构是可为 null 的引用类型功能中的已知缺陷。

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：

```

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 的引用类型 `FirstName` 和 `LastName` 为 null 时，`PrintStudent(default)` 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：

```

#nullable enable

public struct Foo<T>
{
    public T Bar { get; set; }
}

public static class Program
{
    public static void Main()
    {
        string s = default(Foo<string>).Bar;
    }
}

```

在前面的示例中，属性 `Bar` 在运行时将为 `null`，并被分配给不可为 null 的字符串，而未出现任何警告。

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，该示例不会生成任何警告：

```

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。

请参阅

- [可为空引用类型规范草案](#)
- [可为空引用教程简介](#)
- [将现有代码库迁移到可为空引用](#)
- [Nullable\(C# 编译器选项\)](#)

更新库以使用可为 null 引用类型，并将可为空规则传达给调用方

2021/5/7 • [Edit Online](#)

添加[可为 null 引用类型](#)意味着可以声明是否允许或希望每个变量使用 `null` 值。此外，还可应用多个属性（`AllowNull`、`DisallowNull`、`MaybeNull`、`NotNull`、`NotNullWhen`、`MaybeNullWhen` 和 `NotNullIfNotNull`），以完整描述参数和返回值的 null 状态。这可为你提供良好的代码编写体验。如果将不可为 null 的变量设置为 `null`，你会收到警告。如果未对可为 null 的变量执行 null 检查就对其取消引用，你会收到警告。更新库可能需要一些时间，但回报是值得的。向编译器提供的有关何时允许或禁止 `null` 值的信息越多，API 用户收到的警告就越详细。首先，让我们看一个熟悉的示例。假设你的库具有以下用于检索资源字符串的 API：

```
bool TryGetMessage(string key, out string message)
```

前面的示例遵循 .NET 中熟悉的 `Try*` 模式。此 API 有两个引用参数：`key` 和 `message` 参数。此 API 具有与这些参数的是否为 null 相关的以下规则：

- 调用方不应将 `null` 作为 `key` 的参数传递。
- 调用方可以传递值为 `null` 的变量作为 `message` 的参数。
- 如果 `TryGetMessage` 方法返回 `true`，则 `message` 的值不为 null。如果返回值是 `false`，则 `message`（及其 null 状态）的值为 null。

`key` 的规则完全可以用变量类型表示：`key` 应是不可为 null 引用类型。`message` 参数更复杂。它允许 `null` 作为参数，但保证成功时，`out` 参数不为 null。对于这些情况，需要使用更丰富的词汇来描述期望。

更新库以使用可为 null 引用需要的不仅仅是在一些变量和类型名称上添加 `?`。前面的示例表明你还需要检查 API，并考虑对每个输入参数的预期。考虑对返回值的保证，以及方法返回时的任何 `out` 或 `ref` 参数。然后，将这些规则传达给编译器，当调用方不遵守这些规则时，编译器将发出警告。

这项工作需要一些时间。首先从策略入手，将库或应用程序设置为可识别可为 null 引用类型，同时均衡其他需求。你将了解如何均衡正在进行的开发，以启用可为 null 引用类型。你将了解泛型类型定义的相关挑战。你将了解如何应用属性来描述各 API 的前置条件和后置条件。

选择用于可为 null 引用类型的策略

第一种选择是，默认应启用还是禁用可为 null 引用类型。有两种策略：

- 对整个项目启用可为 null 引用类型，而在尚未就绪的代码中禁用。
- 仅对带有可为 null 引用类型注释的代码启用可为 null 引用类型。

通过向库添加其他功能而将库更新为使用可为 null 引用类型时，第一种策略最有效。所有新开发都可识别可为 null 引用类型。更新现有代码时，在这些类中启用可为 null 引用类型。

按照第一种策略，须执行以下步骤：

1. 将 `<Nullable>enable</Nullable>` 元素添加到 .csproj 文件，对整个项目启用可为 null 引用类型。
2. 将 `#nullable disable` pragma 添加到项目中的每个源文件。
3. 处理每个文件时，请删除 pragma 并解决所有警告。

第一种策略在将 pragma 添加到每个文件中时，需要的前期工作更多。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 引用类型，则第二种策略效果更好。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。

按照第二种策略，须执行以下步骤：

1. 将 `#nullable enable` pragma 添加到你希望将其设置为可识别可为 null 引用类型的文件。
2. 解决任何警告。
3. 继续执行前两个步骤，直到将整个库都设置为可识别可为 null 引用类型。
4. 将 `<Nullable>enable</Nullable>` 元素添加到 csproj 文件，对整个项目启用可为 null 引用类型。
5. 当不再需要可为 null 引用类型时，删除 `#nullable enable` pragma。

第二种策略所需的前期工作较少。缺点在于，创建新文件时的第一项任务是添加 pragma，并将文件设置为可识别可为 null 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。

IMPORTANT

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。

是否应为可为 null 警告引入中断性变更？

启用可为 null 引用类型之前，变量被视作“忽略可为 null”。启用可为 null 引用类型后，这些变量变为“不可为 null”。如果这些变量未初始化为非 null 值，编译器将发出警告。

另一个可能的警告来源是尚未初始化的返回值。

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能为 null。当引用变量不得为 null 时，最初的声明正确。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。检查警告时，你将面临关于库的下一个决定。是否考虑修改 API 签名，以便更清晰地传达设计意图？对于之前检查的 `TryGetMessage` 方法，更好的 API 签名可能是：

```
string? TryGetMessage(string key);
```

返回值指示成功或失败，如果找到值，将包含该值。在许多情况下，更改 API 签名可改善它们传递 null 值的方式。

但对于公共库或用户群较大的库，你可能不想引入任何 API 签名更改。对于这类情况和其他常见情形，可以应用属性来更清楚地定义参数或返回值何时可能为 `null`。无论是否考虑更改 API 的图面，你都可能会发现，仅使用类型注释不足以描述参数或返回值的 `null` 值。在这些情况下，可应用属性来更清晰地描述 API。

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。在 C# 8 引入可为 null 的引用类型之前编写的所有代

码都是忽略 null 的。这意味着任何引用类型变量都可以为 null，但不需要进行 null 检查。代码“可识别为 null”后，这些规则就会改变。引用类型不应该是 `null` 值，在取消引用之前，必须对照 `null` 检查可为 null 的引用类型。

API 的规则可能更复杂，正如你在 `TryGetValue` API 方案中看到的那样。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。你可以在[影响可为 null 分析的属性](#)一文中找到描述 API 语义的属性。

泛型定义和可为 null 性

正确传达泛型类型和泛型方法的 null 状态时，必须格外小心。还需要额外注意，可为 null 值类型和可为 null 引用类型在本质上有所不同。`int?` 是 `Nullable<int>` 的同义词，而 `string?` 是带有编译器添加的属性的 `string`。因此，如果不知道 `T` 是 `class` 还是 `struct`，编译器就无法为 `T?` 生成正确的代码。

这一事实不意味着不能使用可为 null 类型(值类型或引用类型)作为封闭式泛型类型的类型参数。

`List<string?>` 和 `List<int?>` 都是 `List<T>` 的有效实例化。

它的意思是，如果没有约束，就无法在泛型类或方法声明中使用 `T?`。例如，`Enumerable.FirstOrDefault<TSource>(IQueryable<TSource>)` 将不会被更改为返回 `T?`。你可以通过添加 `struct` 或 `class` 约束来克服此限制。使用上述任一约束，编译器都知道如何为 `T` 和 `T?` 生成代码。

你可能想要将用于某一泛型类型参数的类型限制为不可为 null 类型。通过在该类型参数上添加 `notnull` 约束即可实现此目的。应用该约束后，类型参数将不得为可为 null 类型。

延迟初始化属性、数据传输对象和可为 null 性

指示延迟初始化属性(即在构造后设置)的可为 null 性时，可能需要格外注意，以确保类继续正确地表达最初的设计意图。

包含延迟初始化属性的类型(例如数据传输对象(DTO))通常由外部库进行实例化，例如数据库 ORM(对象关系映射器)、反序列化程序或其他自动填充来自其他源的属性的组件。

在启用可为 null 引用类型之前，请考虑以下 DTO 类(代表学生)：

```
class Student
{
    [Required]
    public string FirstName { get; set; }

    [Required]
    public string LastName { get; set; }

    public string VehicleRegistration { get; set; }
}
```

设计意图(在此例中由 `Required` 属性指示)表明，在此系统中 `FirstName` 和 `LastName` 属性是必需属性，因此不可为 null。

`VehicleRegistration` 属性不是必需属性，因此可能为 null。

启用可为 null 引用类型时，需要指出 DTO 上哪些属性可为 null，这应与最初的意图一致：

```
class Student
{
    [Required]
    public string FirstName { get; set; }

    [Required]
    public string LastName { get; set; }

    public string? VehicleRegistration { get; set; }
}
```

对于此 DTO，可能为 null 的属性只有 `VehicleRegistration`。

但是，编译器会针对 `FirstName` 和 `LastName` 发出 `CS8618` 警告，指示不可为 null 属性未进行初始化。

你可以通过三个方法在保持最初意图的同时解决编译器警告。这些方法均有效；你应选择最适合自己的编码风格和设计要求的方法。

在构造函数中进行初始化

解决未初始化警告的理想方法是在构造函数中初始化相应属性：

```
class Student
{
    public Student(string firstName, string lastName)
    {
        FirstName = firstName;
        LastName = lastName;
    }

    [Required]
    public string FirstName { get; set; }

    [Required]
    public string LastName { get; set; }

    public string? VehicleRegistration { get; set; }
}
```

仅当用于实例化类的库支持在构造函数中传递参数时，此方法才有效。

库可能支持在构造函数中传递某些属性，但不是全部属性。例如，EF Core 对普通列属性支持[构造函数绑定](#)，但对导航属性却不支持。

查看有关实例化类的库的文档，了解库对构造函数绑定的支持范围。

使用带可为 null 支持字段的属性

如果构造函数绑定不起作用，处理此问题的一种方法是使用带可为 null 支持字段的不可为 null 属性：

```
private string? _firstName;

[Required]
public string FirstName
{
    set => _firstName = value;
    get => _firstName
        ?? throw new InvalidOperationException("Uninitialized " + nameof(FirstName))
}
```

在这种情况下，如果在初始化 `FirstName` 属性之前访问该属性，则代码将引发 `InvalidOperationException`，因为错误使用了 API 协定。

注意，使用支持字段时，某些库可能有特殊的注意事项。例如，可能需要配置 EF Core 才能正确使用[支持字段](#)。

将属性初始化为 null

作为使用可为 null 支持字段的一种更简洁替代方法，或者在实例化类的库不适合该方法的情况下，可以在 null 包容运算符 (`!`) 的帮助下将属性直接初始化为 `null!`：

```
[Required]  
public string FirstName { get; set; } = null!;  
  
[Required]  
public string LastName { get; set; } = null!;  
  
public string? VehicleRegistration { get; set; }
```

如果还未正确初始化属性就对其进行访问，那么在运行时绝不会看到实际的 null 值(除非编程 bug 导致出现此结果)。

另请参阅

- [将现有代码库迁移到可为空引用](#)
- [在 EF Core 中使用可为 null 引用类型](#)

命名空间 (C# 编程指南)

2021/3/5 • [Edit Online](#)

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：

```
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。可以使用 `using` 关键字，如此则不必使用完整的名称，如下例所示：

```
using System;
```

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

有关详细信息，请参阅 [using 指令](#)。

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：

```
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](#)。

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。

请参阅

- [C# 编程指南](#)
- [使用命名空间](#)

- 如何使用 My 命名空间
- 标识符名称
- using 指令
- ::运算符

类型、变量和值

2021/3/22 • [Edit Online](#)

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。每个方法签名指定了每个输入参数和返回值的类型。.NET 类库定义了一组内置数值类型以及表示各种逻辑构造的更复杂类型（如文件系统、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。

类型中可存储以下信息：

- 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。
- 在运行时分配变量内存的位置。
- 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**。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：

```
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;
```

NOTE

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：

```
// 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;
```

方法签名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：

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

在声明变量后，不能使用新类型重新声明该变量，并且不能为其分配与其声明的类型不兼容的值。例如，不能声明 `int` 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，在将值赋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](#)。可能导致数据丢失的转换需要在源代码进行强制转换。

有关详细信息，请参阅[强制转换和类型转换](#)。

内置类型

C# 提供了一组标准的内置数值类型来表示整数、浮点值、布尔表达式、文本字符、十进制值和其他数据类型。另外，还有内置 [字符串](#) 和 [对象](#) 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

使用[结构](#)、[类](#)、[记录](#)、[接口](#)和[枚举](#)构造函数自行创建自定义类型。[.NET](#) 类库本身就是 Microsoft 提供的一组自定义类型，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# 程序中均可使用。对于其他类型，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）。

泛型类型

可使用一个或多个类型参数声明，作为客户端代码在创建类型实例时将提供的实际类型（具体类型）的占位符的类型。这种类型称为泛型类型。例如，`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，例如字符串：

```
List<string> strings = new List<string>();
```

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，并能在编译时抛出错误，例如当尝试向上面示例中的 `strings` 对象添加整数时。有关详细信息，请参阅[泛型](#)。

隐式类型、匿名类型和元组类型

如前所述，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）。变量在编译时仍可接收类型，但类型由编译器提供。有关详细信息，请参阅[隐式类型本地变量](#)。

在某些情况下，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是不方便的。为此，你可以创建匿名类型。有关详细信息，请参阅[匿名类型](#)。

经常需要从方法返回多个值。可以创建在单个方法调用中返回多个值的元组类型。有关详细信息，请参阅[元组类型](#)。

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 `Int32` (C# keyword: `int`) 等内置数值类型）最终都派生自单个基类型，即 `Object` (C# keyword: `object`)。此统一类型层次结构称为[通用类型系统 \(CTS\)](#)。有关 C# 中的继承的详细信息，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` 或 `enum` 关键字定义的类型是值类型。有关值类型的详细信息，请参阅[值类型](#)。使用 `class` 关键字定义的类型是引用类型。有关引用类型的详细信息，请参阅[类](#)。引用类型和值类型具有不同的编译时规则和不同的运行时行为。

请参阅

- [结构类型](#)
- [枚举类型](#)
- [类](#)

类 (C# 编程指南)

2021/3/5 • [Edit Online](#)

引用类型

定义为 [类](#) 的一个类型是 [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可能已在其他位置创建的兼容类型的对象，如下面的示例所示：

```
//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。当分配托管堆上的类型和由 CLR 的自动内存管理功能对其进行回收（称为 [垃圾回收](#)）时，需要开销。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：

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

`class` 关键字前面是访问级别。因为此例中使用的是 `public`，所以任何人都可以创建此类的实例。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，后跟对象要基于的类的名称，来创建对象，如：

```
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：

```
Customer object2;
```

不建议创建这样一个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。但实际上可以使用这类引用来引用某个对象，方法是创建新对象，或者将其分配给现有对象，例如：

```
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。

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，可以继承自其他任何未定义为 `sealed` 的类，而且其他类也可以继承自你的类并重写类虚方法。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：

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

类声明基类时，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](#)。

与 C++ 不同，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)。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，`sealed`(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](#)。

示例

以下示例定义了一个公共类，该类包含一个[自动实现的属性](#)、一个方法和一个名为构造函数的特殊方法。有关详细信息，请参阅[属性、方法和构造函数](#)主题。然后使用 `new` 关键字实例化类的实例。

```
using System;

public class Person
{
    // Constructor that takes no arguments:
    public Person()
    {
        Name = "unknown";
    }

    // Constructor that takes one argument:
    public Person(string name)
    {
        Name = name;
    }

    // Auto-implemented readonly property:
    public string Name { get; }

    // Method that overrides the base class (System.Object) implementation.
    public override string ToString()
    {
        return Name;
    }
}
class TestPerson
{
    static void Main()
    {
        // Call the constructor that has no parameters.
        var person1 = new Person();
        Console.WriteLine(person1.Name);

        // Call the constructor that has one parameter.
        var person2 = new Person("Sarah Jones");
        Console.WriteLine(person2.Name);
        // Get the string representation of the person2 instance.
        Console.WriteLine(person2);

        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
// Output:
// unknown
// Sarah Jones
// Sarah Jones
```

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](#)
- [面向对象的编程](#)
- [多态性](#)
- [标识符名称](#)
- [成员](#)
- [方法](#)
- [构造函数](#)

- 终结器
- 对象

析构元组和其他类型

2021/5/7 • [Edit Online](#)

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。按元素逐个执行此操作会比较麻烦，如下例所示。`QueryCityData` 方法返回一个 3 元组，并通过单独的操作将其每个元素分配给一个变量。

```
using System;

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。

从 C# 7.0 开始，用户可从元组中检索多个元素，或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。析构元组时，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。

析构元组

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 4 元组的元素分配给 4 个单独的变量：

```
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 3 元组。

```
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 3 元组时使用类型推论。

```
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。

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

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

这很麻烦，不建议这样做。

- 最后，可将元组析构到已声明的变量中。

```
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.
}
```

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

请注意，还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”。

请注意，不能混合析构左侧上现有变量的声明和赋值。编译器生成错误 CS8184“析构不能混合左侧的声明和表达式”。当成员包括新声明的和现有的变量。

使用弃元析构元组元素

析构元组时，通常只需要关注某些元素的值。从 C# 7.0 开始，便可利用 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，通过下划线字符 (_) 指定弃元。可弃元任意数量的值，且均由单个弃元 `_` 表示。

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 6 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。

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

public class Example
{
    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
```

析构用户定义类型

除了 `record` 和 `DictionaryEntry` 类型，C# 不提供析构非元组类型的内置支持。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`，且要析构的每个值由方法签名中的 `out` 参数指示。例如，下面的 `Person` 类的 `Deconstruct` 方法返回名字、中间名和姓氏：

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

然后，可使用类似于以下的分配来析构名为 `p` 的 `Person` 类的实例：

```
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、城市名和省/市/自治区名。

```

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 Example
{
    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。

使用弃元析构用户定义类型

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。每个弃元均由名为“_”的变量定义，一个析构操作可包含多个弃元。

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。

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

使用扩展方法析构用户定义的类型

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 `System.Reflection.PropertyInfo` 类定义了两个 `Deconstruct` 扩展方法。第一个方法返回一组值，指示属性的特征，包括其类型、是静态还是实例、是否为只读，以及是否已编制索引。第二个方法指示属性的可访问性。因为 get 和 set 访问器的可访问性可能不同，所以布尔值指示属性是否具有单独的 get 和 set 访问器，如果是，则指示它们是否具有相同的可访问性。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。

```
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 Example
{
    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.WriteLine($"\\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
```

析构 `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 `record` 声明中的每个位置参数创建一个带有 `out` 参数的 `Deconstruct` 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

- [弃元](#)
- [元组类型](#)

接口 (C# 编程指南)

2021/5/10 • [Edit Online](#)

接口包含非抽象类或结构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。从 C# 8.0 开始，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。

```
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。有关示例的链接，请参阅 [相关部分](#)。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。

当类或结构实现接口时，类或结构必须为该接口声明的所有成员提供实现，但不提供默认实现。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。

```
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 `set` 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。接口中定义的实现是虚拟的，实现类可能会替代该实现。

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。

相关部分

- [接口属性](#)
- [接口中的索引器](#)
- [如何实现接口事件](#)
- [类、结构和记录](#)
- [继承](#)
- [接口](#)
- [方法](#)
- [多态性](#)
- [抽象类、密封类及类成员](#)
- [属性](#)
- [事件](#)
- [索引器](#)

请参阅

- [C# 编程指南](#)
- [继承](#)
- [标识符名称](#)

(C#) 中的方法

2021/5/7 • • [Edit Online](#)

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。

NOTE

本主题讨论命名的方法。有关匿名函数的信息，请参阅[匿名函数](#)。

方法签名

通过指定在 `class` 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。

IMPORTANT

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：

```
using System;

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();
}
```

请注意，`Motorcycle` 类包括一个重载的方法 `Drive`。两个方法具有相同的名称，但必须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。调用实例方法需要将对象实例化，并对该对象调用方法；实例方法可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。自变量必须与参数类型兼容，但调用代码中使用的自变量名(如果有)不需要与方法中定义的自变量名相同。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。

```
public class Example
{
    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。

```

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);
    }
}

```

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。

```

using System;

class TestMotorcycle : Motorcycle
{
    public override int Drive(int miles, int speed)
    {
        return (int) Math.Round( ((double)miles) / speed, 0);
    }

    public override double GetTopSpeed()
    {
        return 108.4;
    }

    static void Main()
    {

        TestMotorcycle moto = new TestMotorcycle();
        moto.StartEngine();
        moto.AddGas(15);
        var travelTime = moto.Drive(speed: 60, miles: 170);
        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。

```
var 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](#)。

```
using System;

public class Person
{
    public String FirstName;
}

public class Example
{
    public static void Main()
    {
        var p1 = new Person();
        p1.FirstName = "John";
        var p2 = new Person();
        p2.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。）

```
using System;

public class Person
{
    public String FirstName;

    public override bool Equals(object obj)
    {
        var p2 = obj as Person;
        if (p2 == null)
            return false;
        else
            return FirstName.Equals(p2.FirstName);
    }

    public override int GetHashCode()
    {
        return FirstName.GetHashCode();
    }
}

public class Example
{
    public static void Main()
    {
        var p1 = new Person();
        p1.FirstName = "John";
        var p2 = new Person();
        p2.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。

```
using System;

public class Example
{
    public static void Main()
    {
        int 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，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。

```

using System;

public class SampleRefType
{
    public int value;
}

public class Example
{
    public static void Main()
    {
        var rt = new SampleRefType();
        rt.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。

```

using System;

public class Example
{
    public static void Main()
    {
        int value = 20;
        Console.WriteLine("In Main, value = {0}", value);
        ModifyValue(ref value);
        Console.WriteLine("Back in Main, value = {0}", value);
    }

    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。

```
using System;

public class Example
{
    static void Main()
    {
        int i = 2, j = 3;
        System.Console.WriteLine("i = {0}  j = {1}" , i, j);

        Swap(ref i, ref j);

        System.Console.WriteLine("i = {0}  j = {1}" , i, j);
    }

    static void Swap(ref int x, ref int y)
    {
        int temp = x;
        x = y;
        y = temp;
    }
}
// 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。

参数数组

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数组，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数组类型，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方法：

- 传递相应类型的数组，该类型包含所需数量的元素。
- 向该方法传递相应类型的单独自变量的逗号分隔列表。
- 传递 `null`。
- 不向参数数组提供参数。

以下示例定义了一个名为 `GetVowels` 的方法，该方法返回参数数组中的所有元音。`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，参数是一个空数组。

```

using System;
using System.Linq;

class Example
{
    static void Main()
    {
        string fromArray = GetVowels(new[] { "apple", "banana", "pear" });
        Console.WriteLine($"Vowels from array: '{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 string[] input)
    {
        if (input == null || input.Length == 0)
        {
            return string.Empty;
        }

        var vowels = new char[] { '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: 'aeaaaaea'
//     Vowels from multiple arguments: 'aeaaaaea'
//     Vowels from null: ''
//     Vowels from no value: ''

```

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。

参数的默认值必须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。
- `new ValType()` 形式的表达式，其中 `ValType` 是值类型。请注意，这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。
- `default(ValType)` 形式的表达式，其中 `ValType` 是值类型。

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。

```

using System;

public class Options
{
    public void ExampleMethod(int required, int optionalInt = default(int),
                             string description = "Optional Description")
    {
        Console.WriteLine("{0}: {1} + {2} = {3}", description, required,
                          optionalInt, required + optionalInt);
    }
}

```

如果使用位置自变量调用包含多个可选自变量的方法，调用方必须逐一向所有需要自变量的可选参数提供自变量。例如，在使用 `ExampleMethod` 方法的情况下，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。

```

public class Example
{
    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:
//      Optional Description: 10 + 0 = 10
//      Optional Description: 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# 编译器决定方法应调用哪个特定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。这是重载决策中的常规引用的结果，该引用用于参数较少的候选项。

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将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，将停止执行。

例如，这两种方法都使用 `return` 关键字来返回整数：

```
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：

```
int result = obj.AddTwoNumbers(1, 2);
result = obj.SquareANumber(result);
// The result is 9.
Console.WriteLine(result);
```

```
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。

有时，需要方法返回多个值。从 C# 7.0 开始，可以使用元组类型和元组文本轻松实现此目的。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、中间和最后一个名称及其使用期限。

```
public (string, string, string, int) GetPersonalInfo(string id)
{
    PersonInfo per = PersonInfo.RetrieveInfoById(id);
    return (per.FirstName, per.MiddleName, per.LastName, per.Age);
}
```

然后调用方可通过类似以下的代码使用返回的元组：

```
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。下面的示例展示了 `GetPersonalInfo` 方法的替代版本，该方法使用命名的元素：

```
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 `GetPersonInfo` 方法的调用，如下所示：

```
var person = GetPersonalInfo("111111111");
Console.WriteLine($"{person.FName} {person.LName}: age = {person.Age}");
```

如果将数组作为自变量传递给一个方法，并修改各个元素的值，则该方法不一定会返回该数组，尽管选择这么操作的原因是为了实现更好的样式或功能性的值流。这是因为 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。

```
using System;

public class Example
{
    static void Main(string[] args)
    {
        int[] values = { 2, 4, 6, 8 };
        DoubleValues(values);
        foreach (var value in values)
            Console.Write("{0} ", value);
    }

    public static void DoubleValues(int[] arr)
    {
        for (int ctr = 0; ctr <= arr.GetUpperBound(0); ctr++)
            arr[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。当然，如果并不拥有该类型的源代码，则无法执行该操作。并且，如果还添加任何专用数据字段来支持该方法，这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。

NOTE

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。从 C# 7.0 开始, 异步方法可以有[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。

```
using System;
using System.Threading.Tasks;

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

具有立即仅返回表达式结果, 或单个语句作为方法主体的方法定义很常见。以下是使用 `=>` 定义此类方法的语法快捷方式:

```
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, 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>`、`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](#)

属性

2021/3/5 • • [Edit Online](#)

属性是 C# 中的一等公民。借助该语言所定义的语法，开发人员能够编写出准确表达其设计意图的代码。

访问属性时，其行为类似于字段。但与字段不同的是，属性通过访问器实现；访问器用于定义访问属性或为属性赋值时执行的语句。

属性语法

属性语法是字段的自然延伸。字段定义存储位置：

```
public class Person
{
    public string FirstName;
    // remaining implementation removed from listing
}
```

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：

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

    // remaining implementation removed from listing
}
```

上述语法是 **自动属性** 语法。编译器生成支持该属性的字段的存储位置。编译器还实现 `get` 和 `set` 访问器的正文。

有时，需要将属性初始化为其类型默认值以外的值。C# 通过在属性的右括号后设置值达到此目的。对于 `FirstName` 属性的初始值，你可能更希望设置为空字符串而非 `null`。可按如下所示进行指定：

```
public class Person
{
    public string FirstName { get; set; } = string.Empty;

    // remaining implementation removed from listing
}
```

特定初始化对于只读属性最有用，本文后面部分将进行介绍。

你也可以自行定义存储，如下所示：

```
public class Person
{
    public string FirstName
    {
        get { return firstName; }
        set { firstName = value; }
    }
    private string firstName;
    // remaining implementation removed from listing
}
```

属性实现是单个表达式时，可为 `getter` 或 `setter` 使用 expression-bodied 成员：

```
public class Person
{
    public string FirstName
    {
        get => firstName;
        set => firstName = value;
    }
    private string firstName;
    // remaining implementation removed from listing
}
```

在本文中，将在所有适用之处使用此简化的语法。

上述属性定义是读-写属性。注意 `set` 访问器中的关键字 `value`。`set` 访问器始终具有一个名为 `value` 的参数。`get` 访问器必须返回一个值，该值可转换为该属性的类型（本例中为 `string`）。

这就是该语法的基础知识。有许多不同的语法变体，支持着各种不同的设计习惯。接下来我们将了解这些变体，以及每个变体的语法选项。

方案

上述示例介绍了属性定义中最简单的一种情况：不进行验证的读-写属性。通过在 `get` 和 `set` 访问器中编写所需的代码，可以创建多种不同的方案。

验证

可以在 `set` 访问器中编写代码，确保由某个属性表示的值始终有效。例如，假设 `Person` 类的一个规则是姓名不得为空白或空白符。可按如下方式编写：

```
public class Person
{
    public string FirstName
    {
        get => firstName;
        set
        {
            if (string.IsNullOrWhiteSpace(value))
                throw new ArgumentException("First name must not be blank");
            firstName = value;
        }
    }
    private string firstName;
    // remaining implementation removed from listing
}
```

可通过将 `throw` 表达式用作属性资源库验证的一部分来简化前面的示例：

```
public class Person
{
    public string FirstName
    {
        get => firstName;
        set => firstName = (!string.IsNullOrWhiteSpace(value)) ? value : throw new ArgumentException("First name must not be blank");
    }
    private string firstName;
    // remaining implementation removed from listing
}
```

上面的示例强制执行名字不得为空白或空字符的规则。如果开发人员编写

```
hero.FirstName = "";
```

该赋值会引发 `ArgumentException`。由于属性 `set` 访问器必须具有 `void` 返回类型，因此将通过引发异常来报告 `set` 访问器中的错误。

你可以根据自己的情况随意扩展此语法。可以检查不同属性之间的关系，或根据任何外部条件进行验证。任何有效的 C# 语句在属性访问器中都是有效的。

只读

到目前为止，你了解的所有属性定义都是具有公共访问器的读/写属性。但这不是属性唯一有效的可访问性。你可以创建只读属性，或者对 `set` 和 `get` 访问器提供不同的可访问性。假设 `Person` 类只能从该类的其他方法中启用 `FirstName` 属性值更改。可以为 `set` 访问器提供 `private` 可访问性，而不是 `public` 可访问性：

```
public class Person
{
    public string FirstName { get; private set; }

    // remaining implementation removed from listing
}
```

现在，可以从任意代码访问 `FirstName` 属性，但只能从 `Person` 类中的其他代码对其赋值。

可以向 `set` 和 `get` 访问器添加任何严格访问修饰符。在单个访问器上放置的任何访问修饰符都必须比属性定义上的访问修饰符提供更严格的限制。上述做法是合法的，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`。

在 `get` 访问器上放置限制性更强的修饰符也是合法的。例如，可以有一个 `public` 属性，但将 `get` 访问器限制为 `private`。不过实际情况下很少这么做。

还可以限制对属性的修改，以便只能在构造函数或属性初始化表达式中设置属性。可按照这种方式修改 `Person` 类，如下所示：

```
public class Person
{
    public Person(string firstName) => this.FirstName = firstName;

    public string FirstName { get; }

    // remaining implementation removed from listing
}
```

此功能通常用于初始化作为只读属性公开的集合：

```
public class Measurements
{
    public ICollection<DataPoint> points { get; } = new List<DataPoint>();
}
```

计算属性

属性无需只返回某个成员字段的值。可以创建返回计算值的属性。让我们展开 `Person` 对象，返回通过串联名字和姓氏计算得出的全名：

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

    public string LastName { get; set; }

    public string FullName { get { return $"{FirstName} {LastName}"; } }
}
```

上面的示例使用 **字符串内插** 功能来创建全名的格式化字符串。

也可以使用 expression-bodied 成员，以更简洁的方式来创建 `FullName` 计算属性：

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

    public string LastName { get; set; }

    public string FullName => $"{FirstName} {LastName}";
}
```

expression-bodied 成员使用 **lambda 表达式语法** 来定义包含单个表达式的方法。在这里，该表达式返回 `person` 对象的全名。

缓存的计算属性

可以将计算属性和存储的概念混合起来，创建“缓存的计算属性”。例如，可以更新 `FullName` 属性，以便仅在第一次访问该属性时进行字符串格式设置：

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

    public string LastName { get; set; }

    private string fullName;
    public string FullName
    {
        get
        {
            if (fullName == null)
                fullName = $"{FirstName} {LastName}";
            return fullName;
        }
    }
}
```

不过，上面的代码含有 bug。如果代码更新 `FirstName` 或 `LastName` 属性的值，那么，以前计算的 `fullName` 字段将无效。修改 `FirstName` 和 `LastName` 属性的 `set` 访问器，以便重新计算 `fullName` 字段：

```

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 == 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。如果另一个状态更改使以前计算的版本失效，则重新计算。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。这是使用属性公开对象的数据成员的关键原因。

将特性附加到自动实现的属性

从 C# 7.3 开始，可在自动实现的属性中将字段特性附加到编译器生成的支持字段。例如，可考虑添加唯一整数 `Person` 属性的 `Id` 类的修订。使用自动实现的属性编写 `Id` 属性，但是该设计不需要保留 `Id` 属性。

`NonSerializedAttribute` 只能附加到字段，不能附加到属性。可使用特性上的 `field:` 说明符将 `NonSerializedAttribute` 附加到 `Id` 属性的支持字段，如下例所示：

```

public class Person
{
    public string FirstName { get; set; }

    public string LastName { get; set; }

    [field:NonSerialized]
    public int Id { get; set; }

    public string FullName => $"{FirstName} {LastName}";
}

```

该技术适用于附加到自动实现的属性上的支持字段的所有特性。

实现 `INotifyPropertyChanged`

需要在属性访问器中编写代码的最后一种情形是为了支持 `INotifyPropertyChanged` 接口，该接口用于通知数据

绑定客户端值已更改。当属性值发生更改时，该对象引发 `INotifyPropertyChanged.PropertyChanged` 事件来指示更改。数据绑定库则基于该更改来更新显示元素。下面的代码演示如何为此 `person` 类的 `FirstName` 属性实现 `INotifyPropertyChanged`。

```
public class Person : INotifyPropertyChanged
{
    public string FirstName
    {
        get => firstName;
        set
        {
            if (string.IsNullOrWhiteSpace(value))
                throw new ArgumentException("First name must not be blank");
            if (value != firstName)
            {
                PropertyChanged?.Invoke(this,
                    new PropertyChangedEventArgs(nameof(FirstName)));
            }
            firstName = value;
        }
    }
    private string firstName;

    public event PropertyChangedEventHandler PropertyChanged;
    // remaining implementation removed from listing
}
```

? . 运算符称作 `null` 条件运算符。它在计算运算符右侧之前会检查是否存在空引用。最终结果为：如果 `PropertyChanged` 事件没有订阅者，则不执行用于引发该事件的代码。在这种情况下，如果不执行此检查，则会引发 `NullReferenceException`。有关详细信息，请参阅 [events](#)。此示例还使用新的 `nameof` 运算符将属性名称符号转换为其文本表示形式。使用 `nameof` 可以减少输入错属性名称这样的错误。

再次说明，实现 `INotifyPropertyChanged` 是可以在访问器中编写代码以支持所需方案的情况的示例。

总结

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、延迟计算或方案所需的任何要求。

索引器

2021/5/7 • • [Edit Online](#)

索引器类似于属性。很多时候，创建索引器与创建属性所使用的编程语言特性是一样的。索引器使属性可以被索引：使用一个或多个参数引用的属性。这些参数为某些值集合提供索引。

索引器语法

可以通过变量名和方括号访问索引器。将索引器参数放在方括号内：

```
var item = someObject["key"];
someObject["AnotherKey"] = item;
```

使用 `this` 关键字作为属性名声明索引器，并在方括号内声明参数。此声明与前一段中所示的用法相匹配：

```
public int this[string key]
{
    get { return storage.Find(key); }
    set { storage.SetAt(key, value); }
}
```

从最初的示例中，可以看到属性语法和索引器语法之间的关系。此类比在索引器的大部分语法规则中进行。索引器可以使用任何有效的访问修饰符（`public`、`protected`、`internal`、`protected internal`、`internal`、`private` 或 `private protected`）。它们可能是密封、虚拟或抽象的。与属性一样，可以在索引器中为 `get` 和 `set` 访问器指定不同的访问修饰符。你还可以指定只读索引器（忽略 `set` 访问器）或只写索引器（忽略 `get` 访问器）。

属性的各种用法同样适用于索引器。此规则的唯一例外是“自动实现属性”。编译器无法始终为索引器生成正确的存储。

用于引用项的集合中的某个项的参数可区分索引器和属性。只要每个索引器的参数列表是唯一的，就可以对一个类型定义多个索引器。让我们来探讨可能在类定义中使用一个或多个索引器的不同场景。

方案

如果类型的 API 对集合进行建模，并且为集合定义了参数，则需要在此类型中定义索引器。索引器可能直接映射到属于 .NET Core 框架一部分的集合类型，也可能不。除了对集合进行建模，类型还有其他职责。通过索引器可提供与类型的抽象化匹配的 API，而无需公开如何存储或计算此抽象化的值的内部细节。

让我们演练一些使用索引器的常见场景。可以访问[索引器的示例文件夹](#)。有关下载说明，请参阅[示例和教程](#)。

数组和矢量

创建索引器的一个最常见的场景是当类型对数组或矢量进行建模时。可以创建一个索引器用于对已排序的数据列表进行建模。

创建自己的索引器的优点是你可以为集合定义存储以满足你的需求。假设以下场景：类型对历史数据进行建模，并且此历史数据太大而无法立即加载到内存中。需要根据使用情况加载和卸载集合的某些部分。以下示例对此行为进行建模。此示例报告存在多少数据点。此示例按需创建页以存储部分数据。此示例从内存中删除页，以便为较新的请求所需的页腾出空间。

```
public class DataSamples
{
    private class Page
```

```

{
    private readonly List<Measurements> pageData = new List<Measurements>();
    private readonly int startingIndex;
    private readonly int length;
    private bool dirty;
    private DateTime lastAccess;

    public Page(int startingIndex, int length)
    {
        this.startingIndex = startingIndex;
        this.length = length;
        lastAccess = DateTime.Now;

        // This stays as random stuff:
        var generator = new Random();
        for(int i=0; i < length; i++)
        {
            var m = new Measurements
            {
                HiTemp = generator.Next(50, 95),
                LoTemp = generator.Next(12, 49),
                AirPressure = 28.0 + generator.NextDouble() * 4
            };
            pageData.Add(m);
        }
    }

    public bool HasItem(int index) =>
        ((index >= startingIndex) &&
        (index < startingIndex + length));

    public Measurements this[int index]
    {
        get
        {
            lastAccess = DateTime.Now;
            return pageData[index - startingIndex];
        }
        set
        {
            pageData[index - startingIndex] = value;
            dirty = true;
            lastAccess = DateTime.Now;
        }
    }

    public bool Dirty => dirty;
    public DateTime LastAccess => lastAccess;
}

private readonly int totalSize;
private readonly List<Page> pagesInMemory = new List<Page>();

public DataSamples(int totalSize)
{
    this.totalSize = totalSize;
}

public Measurements this[int index]
{
    get
    {
        if (index < 0)
            throw new IndexOutOfRangeException("Cannot index less than 0");
        if (index >= totalSize)
            throw new IndexOutOfRangeException("Cannot index past the end of storage");

        var page = updateCachedPagesForAccess(index);
        return page[index];
    }
}

```

```

        ,
        set
    {
        if (index < 0)
            throw new IndexOutOfRangeException("Cannot index less than 0");
        if (index >= totalSize)
            throw new IndexOutOfRangeException("Cannot index past the end of storage");
        var page = updateCachedPagesForAccess(index);

        page[index] = value;
    }
}

private Page updateCachedPagesForAccess(int index)
{
    foreach (var p in pagesInMemory)
    {
        if (p.HasItem(index))
        {
            return p;
        }
    }
    var startingIndex = (index / 1000) * 1000;
    var nextPage = new Page(startingIndex, 1000);
    addPageToCache(nextPage);
    return nextPage;
}

private void addPageToCache(Page p)
{
    if (pagesInMemory.Count > 4)
    {
        // remove oldest non-dirty page:
        var oldest = pagesInMemory
            .Where(page => !page.Dirty)
            .OrderBy(page => page.LastAccess)
            .FirstOrDefault();
        // Note that this may keep more than 5 pages in memory
        // if too much is dirty
        if (oldest != null)
            pagesInMemory.Remove(oldest);
    }
    pagesInMemory.Add(p);
}
}

```

可以按照此设计惯例对任何类型的集合进行建模，其中有充分的理由不将整个数据集加载到内存集合。请注意，`Page` 类是私有嵌套类，不是公共接口的一部分。向此类的任何用户隐藏这些详细信息。

字典

另一个常见场景是需要对字典或映射进行建模。当类型存储基于键（通常是文本键）的值时出现此情况。本示例创建的字典将命令行参数映射到管理这些选项的 [Lambda 表达式](#)。以下示例演示了两个类：`ArgsActions` 类将命令行选项映射到 `Action` 委托；`ArgsProcessor` 类在遇到此选项时使用 `ArgsActions` 执行每个 `Action`。

```

public class ArgsProcessor
{
    private readonly ArgsActions actions;

    public ArgsProcessor(ArgsActions actions)
    {
        this.actions = actions;
    }

    public void Process(string[] args)
    {
        foreach(var arg in args)
        {
            actions[arg]?.Invoke();
        }
    }
}

public class ArgsActions
{
    readonly private Dictionary<string, Action> argsActions = new Dictionary<string, Action>();

    public Action this[string s]
    {
        get
        {
            Action action;
            Action defaultAction = () => {} ;
            return argsActions.TryGetValue(s, out action) ? action : defaultAction;
        }
    }

    public void SetOption(string s, Action a)
    {
        argsActions[s] = a;
    }
}

```

在此示例中，`ArgsAction` 集合紧密映射到基础集合。`get` 确定是否已配置给定的选项。如果已配置，则返回与此选项相关联的 `Action`。如果未配置，则返回不执行任何操作的 `Action`。公共访问器不包括 `set` 访问器。相反地，设计使用公共方法来设置选项。

多维映射

可以创建使用多个参数的索引器。此外，这些参数未限制为相同的类型。请看以下两个示例。

第一个示例演示为 Mandelbrot 集合生成值的类。有关此集合背后的数学原理的详细信息，请参阅[这篇文章](#)。索引器使用两个双精度型来定义平面 XY 上的一个点。Get 访问器计算迭代的次数，直到确定某个点不在集合中。如果达到最大迭代数，并且点在集合中，则返回类的 `maxIterations` 值。（Mandelbrot 集合常用的计算机生成的图像定义迭代数量的颜色，以便确定一个点是否在集合外部。）

```
public class Mandelbrot
{
    readonly private int maxIterations;

    public Mandelbrot(int maxIterations)
    {
        this.maxIterations = maxIterations;
    }

    public int this [double x, double y]
    {
        get
        {
            var iterations = 0;
            var x0 = x;
            var y0 = y;

            while ((x*x + y * y < 4) &&
                   (iterations < maxIterations))
            {
                var newX = x * x - y * y + x0;
                y = 2 * x * y + y0;
                x = newX;
                iterations++;
            }
            return iterations;
        }
    }
}
```

Mandelbrot 集合在每个 (x,y) 坐标上为实际数值定义值。这将定义一个字典，其中可能包含无限数目的值。因此，集合后面没有任何存储。相反，当代码调用 `get` 访问器时，此类计算每个点的值。未使用任何基础存储。

请查看上一次索引器的使用，其中索引器采用多个不同类型的参数。请考虑一个管理历史温度数据的程序。此索引器使用一个城市和一个日期来设置或获取位置的高温和低温：

```

using DateMeasurements =
    System.Collections.Generic.Dictionary<System.DateTime, IndexersSamples.Common.Measurements>;
using CityDataMeasurements =
    System.Collections.Generic.Dictionary<string, System.Collections.Generic.Dictionary<System.DateTime,
IndexersSamples.Common.Measurements>>;

public class HistoricalWeatherData
{
    readonly CityDataMeasurements storage = new CityDataMeasurements();

    public Measurements this[string city, DateTime date]
    {
        get
        {
            var cityData = default(DateMeasurements);

            if (!storage.TryGetValue(city, out cityData))
                throw new ArgumentOutOfRangeException(nameof(city), "City not found");

            // Strip out any time portion:
            var index = date.Date;
            var measure = default(Measurements);
            if (cityData.TryGetValue(index, out measure))
                return measure;
            throw new ArgumentOutOfRangeException(nameof(date), "Date not found");
        }
        set
        {
            var cityData = default(DateMeasurements);

            if (!storage.TryGetValue(city, out cityData))
            {
                cityData = new DateMeasurements();
                storage.Add(city, cityData);
            }

            // Strip out any time portion:
            var index = date.Date;
            cityData[index] = value;
        }
    }
}

```

此示例创建的索引器将天气数据映射到两个不同的参数：城市（由 `string` 表示）和日期（由 `DateTime` 表示）。内部存储使用两个 `Dictionary` 类来表示此二维字典。公共 API 不再表示基础存储。相反地，凭借索引器的语言特性可以创建表示抽象化的一个公共接口，即使基础存储必须使用不同的核心集合类型也是如此。

一些开发人员可能不熟悉此代码的两部分。这两个 `using` 指令：

```

using DateMeasurements = System.Collections.Generic.Dictionary<System.DateTime,
IndexersSamples.Common.Measurements>;
using CityDataMeasurements = System.Collections.Generic.Dictionary<string,
System.Collections.Generic.Dictionary<System.DateTime, IndexersSamples.Common.Measurements>>;

```

为构造泛型类型创建别名。通过这些语句，稍后代码可以使用更具描述性的 `DateMeasurements` 和 `CityDataMeasurements` 名称，而不是 `Dictionary<DateTime, Measurements>` 和 `Dictionary<string, Dictionary<DateTime, Measurements>>` 的泛型构造。此构造要求在 `=` 符号右侧使用完全限定的类型名称。

另一项技术是对任何用于集合的索引的 `DateTime` 对象剥离时间部分。`.NET` 不包含仅日期类型。开发人员使用 `DateTime` 类型，但使用 `Date` 属性来确保这一天的任何 `DateTime` 对象是对等的。

总结

只要类中有类似于属性的元素就应创建索引器，此属性代表的不是一个值，而是值的集合，其中每一个项由一组参数标识。这些参数可以唯一标识应引用的集合中的项。索引器延伸了属性的概念，索引器中的一个成员被视为类外部的一个数据项，但又类似于内部的一个方法。索引器允许参数在代表项的集合的属性中查找单个项。

弃元 - C# 指南

2021/5/7 • [Edit Online](#)

从 C# 7.0 开始, C# 支持弃元, 这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。

因为只有一个弃元变量, 甚至不为该变量分配存储空间。所以, 弃元可以减少内存分配。弃元使代码意图更加明确。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:

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

从 C# 9.0 开始, 可以使用弃元指定 Lambda 表达式中不使用的输入参数。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 CS0301:“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。

```
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。

```

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。

```
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。

```
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。

```

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。

```

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：

```

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");
}

```

NOTE

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：

```
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：

```
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：“无法在此范围内声明名为“_”的局部变量或参数，因为该名称用于在封闭的局部范围内定义局部变量或参数”。例如：

```
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

- [析构元组和其他类型](#)
- [is 运算符](#)
- [switch 关键字](#)

泛型 (C# 编程指南)

2020/11/2 • [Edit Online](#)

泛型将类型参数的概念引入 .NET，这样就可设计具有以下特征的类和方法：在客户端代码声明并初始化这些类或方法之前，这些类或方法会延迟指定一个或多个类型。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：

```
// 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。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。非泛型集合（如 [ArrayList](#)）不建议使用，并且保留用于兼容性目的。有关详细信息，请参阅 [.NET 中的泛型](#)。

当然，也可以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`。其使用方法如下：

- 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。

请注意，`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`。

```

// 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 IEnumarator<T> GetEnumarator()
    {
        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。只需更改类型参数，即可轻松修改以下代码，创建字符串或任何其他自定义类型的列表：

```
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");
    }
}
```

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。应尽可能使用这些类来代替某些类，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。
- 在泛型数据类型中所用类型的信息可在运行时通过使用反射来获取。

相关章节

- [泛型类型参数](#)
- [类型参数的约束](#)
- [泛型类](#)
- [泛型接口](#)
- [泛型方法](#)
- [泛型委托](#)
- [C++ 模板和 C# 泛型之间的区别](#)
- [泛型和反射](#)
- [运行时中的泛型](#)

C# 语言规范

有关详细信息，请参阅 [C# 语言规范](#)。

请参阅

- [System.Collections.Generic](#)
- [C# 编程指南](#)
- [类型](#)
- [`<typeparam>`](#)
- [`<typeparamref>`](#)

- .NET 中的泛型

迭代器

2021/5/7 • • [Edit Online](#)

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（该对象遍历容器，尤其是列表）。这些方法可用于：

- 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# 语言提供了适用于这两种方案的功能。本文概述了这些功能。

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。还可以[查看或下载本主题的已完成示例](#)。有关下载说明，请参阅[示例和教程](#)。

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：

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

就这么简单。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。

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：

```
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：

```
public IEnumerable<int> GetSingleDigitNumbers()
{
    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：

```
public IEnumerable<int> GetSingleDigitNumbers()
{
    int index = 0;
    while (index < 10)
        yield return index++;

    yield return 50;

    index = 100;
    while (index < 110)
        yield return index++;
}
```

这是基本语法。我们来看一个需要编写迭代器方法的真实示例。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：

```
public static IEnumerable<T> Sample(this IEnumerable<T> sourceSequence, int interval)
{
    int index = 0;
    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。不会编译以下内容：

```
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`：

```
public IEnumerable<int> GetSingleDigitNumbers()
{
    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：

```
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：

```
IEnumerator<int> enumerator = collection.GetEnumerator();
while (enumerator.MoveNext())
{
    var item = enumerator.Current;
    Console.WriteLine(item.ToString());
}
```

上述构造表示由 C# 编译器版本 5 及更高版本生成的代码。在版本 5 之前，`item` 变量的范围有所不同：

```
// C# versions 1 through 4:  
IEnumerator<int> enumerator = collection.GetEnumerator();  
int item = default(int);  
while (enumerator.MoveNext())  
{  
    item = enumerator.Current;  
    Console.WriteLine(item.ToString());  
}
```

此范围更改的原因在于：较早行为可能导致难以诊断出有关 Lambda 表达式的 bug。若要详细了解 lambda 表达式，请参阅 [Lambda 表达式](#)。

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` 的情况。完整扩展生成的代码更类似如下：

```
{  
    var enumerator = collection.GetEnumerator();  
    try  
    {  
        while (enumerator.MoveNext())  
        {  
            var item = enumerator.Current;  
            Console.WriteLine(item.ToString());  
        }  
    } finally  
    {  
        // dispose of enumerator.  
    }  
}
```

枚举器的释放方式取决于 `enumerator` 类型的特征。一般情况下，`finally` 子句扩展为：

```
finally  
{  
    (enumerator as IDisposable)?.Dispose();  
}
```

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` 的隐式转换，则 `finally` 子句扩展为一个空白块：

```
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：

```
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。

委托简介

2021/3/5 • [Edit Online](#)

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。

正如你稍后将在此系列文章中看到的，为与此类似的算法编写的 C# 代码是类型安全的，它使用语言规则和编译器来确保类型与参数匹配，并返回类型。

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非常有用。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。本文剩余部分将介绍语言功能、库支持和使用委托时使用的通用语法结构。

你将了解 `delegate` 关键字和它所生成的代码。你将了解 `System.Delegate` 类中的功能，以及如何使用这些功能。你将了解如何创建类型安全的委托，以及如何创建可以通过委托调用的方法。你还将了解如何使用 Lambda 表达式来处理委托和事件。你将看到作为 LINQ 的构建基块的委托的用途。你将了解委托如何成为 .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](#)

System.Delegate 和 `delegate` 关键字

2020/11/2 • [Edit Online](#)

[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：

```
// 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。

NOTE

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。

```
// 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：

```
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：

```
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：

```
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：

```
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：

```
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。

[下一部分](#)

强类型委托

2021/5/8 • • [Edit Online](#)

[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：

```
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：

```
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 `Predicate<T>` 委托类型，可返回单个值的测试结果：

```
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：

```
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

2020/11/2 • [Edit Online](#)

[上一页](#)

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：

```
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：

```
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。

```
public static class Logger
{
    public static Action<string> WriteMessage;

    public static void LogMessage(string msg)
    {
        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：

```
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：

```
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。

使用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。

设置输出的格式

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：

```
public enum Severity
{
    Verbose,
    Trace,
    Information,
    Warning,
    Error,
    Critical
}
```

```
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}";
        WriteMessage(outputMsg);
    }
}
```

接下来，使用 `Severity` 参数来筛选发送到日志输出的消息。

```
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}";
        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：

```

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：

```
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：

```

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：

```
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，记录器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：

```
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，这些不同的组件非常松散地耦合在一起。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非常强的复原能力。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。

[下一部分](#)

事件介绍

2020/11/2 • [Edit Online](#)

[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：

```
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：

```
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：

```
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：

```
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

2020/3/18 • [Edit Online](#)

[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：

```
void OnEventRaised(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。（在[下一部分](#)中此约定不再强制执行。）即使事件类型无需任何其他参数，你仍将提供这两种参数。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。可以创建多个事件监听器，用于在找到查找的文件时执行不同的操作。合并不同的监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：

```
public class FileEventArgs : EventArgs
{
    public string FoundFile { get; }

    public FileEventArgs(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。

```
public class FileSearcher
{
    public event EventHandler<FileEventArgs> FileFound;

    public void Search(string directory, string searchPattern)
    {
        foreach (var file in Directory.EnumerateFiles(directory, searchPattern))
        {
            FileFound?.Invoke(this, new FileEventArgs(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：

```
public event EventHandler<FileEventArgs> 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：

```
EventHandler<FileEventArgs> onFileFound = (sender, eventArgs) =>
{
    Console.WriteLine(eventArgs.FoundFile);
    filesFound++;
};

fileLister.FileFound += onFileFound;
```

和删除处理程序：

```
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，有两种不同的模式可供使用。在两种情况下，你都将为找到的文件事件向 EventArgs 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 `FileEventArgs` 类型：

```
public class FileEventArgs : EventArgs
{
    public string FoundFile { get; }
    public bool CancelRequested { get; set; }

    public FileEventArgs(string fileName)
    {
        FoundFile = fileName;
    }
}
```

此新字段将自动初始化为 `false`，即布尔字段的默认值，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：

```
public void List(string directory, string searchPattern)
{
    foreach (var file in Directory.EnumerateFiles(directory, searchPattern))
    {
        var args = new FileEventArgs(file);
        FileFound?.Invoke(this, args);
        if (args.CancelRequested)
            break;
    }
}
```

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：

```
EventHandler<FileEventArgs> 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。

```
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。

```
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：

```

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)
        {
            directoryChanged?.Invoke(this,
                new SearchDirectoryArgs(dir, totalDirs, completedDirs++));
            // Search 'dir' and its subdirectories for files that match the search pattern:
            SearchDirectory(dir, searchPattern);
        }
        // Include the Current Directory:
        directoryChanged?.Invoke(this,
            new SearchDirectoryArgs(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))
    {
        var args = new FileFoundArgs(file);
        FileFound?.Invoke(this, args);
        if (args.CancelRequested)
            break;
    }
}

```

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 `ChangeDirectory` 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。

```

fileLister.DirectoryChanged += (sender, eventArgs) =>
{
    Console.WriteLine($"Entering '{eventArgs.CurrentSearchDirectory}'.");
    Console.WriteLine($" {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。

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

2020/3/18 • [Edit Online](#)

[上一篇文章](#)

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 `FileEventArgs` 和 `SearchDirectoryEventArgs` 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 `SearchDirectoryEventArgs` 更改为结构：

```
internal struct SearchDirectoryEventArgs
{
    internal string CurrentSearchDirectory { get; }
    internal int TotalDirs { get; }
    internal int CompletedDirs { get; }

    internal SearchDirectoryEventArgs(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 `FileEventArgs` 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

还有最后一个模式需要了解：如何正确编写调用异步代码的事件订阅服务器。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：

```
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。（调用方法已继续执行，因为它是 `async`。）对于不同的环境，实际的运行时行为将有不同的定义。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

2020/11/2 • [Edit Online](#)

[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。

语言集成查询 (LINQ)

2021/5/7 • [Edit Online](#)

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。

```
class LINQQueryExpressions
{
    static void Main()
    {

        // Specify the data source.
        int[] scores = new int[] { 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](#)。

查询表达式基础

2021/5/10 • [Edit Online](#)

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”（尽管这些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。查询然后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[]`）：

```
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，查询可以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。

```
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：

```
int highScoreCount =
    (from score in scores
    where score > 80
    select score)
    .Count();
```

在前面的示例中，请注意在调用 `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，来表示此行为。这种方法更具可读性，因为它使存储查询的变量与存储结果的查询分开。

```
IEnumerable<int> highScoresQuery3 =  
    from score in scores  
    where score > 80  
    select score;  
  
int 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。每个子句进而包含一个或多个 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其他查询子句的源。

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。

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。

```
static void Main()  
{  
    // Data source.  
    int[] scores = { 90, 71, 82, 93, 75, 82 };  
  
    // Query Expression.  
    IEnumerable<int> scoreQuery = //query variable  
        from score in scores //required  
        where score > 80 // optional  
        orderby score descending // optional  
        select score; //must end with select or group  
  
    // Execute the query to produce the results  
    foreach (int testScore in scoreQuery)  
    {  
        Console.WriteLine(testScore);  
    }  
}  
// Outputs: 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：

```
//Query syntax
IQueryable<City> queryMajorCities =
    from city in cities
    where city.Population > 100000
    select city;

// Method-based syntax
IQueryable<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：

```
int highestScore =
    (from score in scores
     select score)
    .Max();

// or split the expression
IQueryable<int> scoreQuery =
    from score in scores
    select score;

int highScore = scoreQuery.Max();
// the following returns the same result
int highScore = scores.Max();

List<City> largeCitiesList =
    (from country in countries
     from city in country.Cities
     where city.Population > 10000
     select city)
    .ToList();

// or split the expression
IQueryable<City> largeCitiesQuery =
    from country in countries
    from city in country.Cities
    where city.Population > 10000
    select city;

List<City> largeCitiesList2 = largeCitiesQuery.ToList();
```

有关表示查询的不同方式的详细信息，请参阅 [LINQ 中的查询语法和方法语法](#)。

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，本主题中前面演示的查询示例也可以使用隐式类型化进行表示：

```
// Use of var is optional here and in all queries.
// queryCities is an IQueryable<City> just as
// when it is explicitly typed.
var queryCities =
    from city in cities
    where city.Population > 100000
    select city;
```

有关详细信息，请参阅 [隐式类型化局部变量](#) 和 [LINQ 查询操作中的类型关系](#)。

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。

```
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内，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，可使用其他 `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：

```
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名称首字母的 `char` 类型。

```
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。

```
IEnumerable<Country> sortedQuery =  
    from country in countries  
    orderby country.Area  
    select country;
```

`select` 子句可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。请注意，新对象使用对象初始值设定项进行初始化。

```
// Here var is required because the query  
// produces an anonymous type.  
var queryNameAndPop =  
    from country in countries  
    select new { Name = country.Name, Pop = country.Population };
```

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其他查询操作, 则可以这样做。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, 附加子句会筛选出一些组, 然后按升序对组进行排序。若要执行这些附加操作, 需要由 `countryGroup` 表示的延续。

```
// 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。

```
IEnumerable<City> queryCityPop =
    from city in cities
    where city.Population < 200000 && city.Population > 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。

```
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 `Category` 与 `categories` 中的任何字符串均不匹配的产品。`select` 语句投影属性取自 `cat` 和 `prod` 的新类型。

```
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。

```
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 (string s in queryFirstNames)
    Console.WriteLine(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。

```
var queryGroupMax =
    from student in students
    group student by student.GradeLevel into studentGroup
    select new
    {
        Level = studentGroup.Key,
        HighestScore =
            (from student2 in studentGroup
            select student2.Scores.Average())
            .Max()
    };
```

有关详细信息，请参阅[对分组操作执行子查询](#)。

请参阅

- [C# 编程指南](#)
- [语言集成查询 \(LINQ\)](#)
- [查询关键字 \(LINQ\)](#)
- [标准查询运算符概述](#)

C# 中的 LINQ

2020/3/18 • [Edit Online](#)

此部分包含提供有关 LINQ 的详细信息的主题链接。

本节内容

[LINQ 查询简介](#)

介绍所有语言和数据源共有的基本 LINQ 查询操作的 3 个部分。

[LINQ 和泛型类型](#)

简单介绍泛型类型在 LINQ 中的使用。

[使用 LINQ 进行数据转换](#)

介绍转换在查询中检索到的数据时可采用的各种方式。

[LINQ 查询操作中的类型关系](#)

介绍如何在 LINQ 查询操作的 3 个部分中保留和/或转换类型。

[LINQ 中的查询语法和方法语法](#)

比较表达 LINQ 查询的两种方式：方法语法和查询语法。

[支持 LINQ 的 C# 功能](#)

描述 C# 中支持 LINQ 的语言构造。

相关章节

[LINQ 查询表达式](#)

概述 LINQ 中的查询并提供指向其他资源的链接。

[标准查询运算符概述](#)

介绍 LINQ 中使用的标准方法。

在 C# 中编写 LINQ 查询

2020/11/2 • [Edit Online](#)

本文介绍可以用于在 C# 中编写 LINQ 查询的三种方法：

1. 使用查询语法。
2. 使用方法语法。
3. 结合使用查询语法和方法语法。

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。一般情况下，规则是尽可能使用 (1)，每当需要时使用 (2) 和 (3)。

NOTE

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

编写大多数查询的推荐方式是使用查询语法 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。

```
// Query #1.  
List<int> numbers = new List<int>() { 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 };  
  
// The query variable can also be implicitly typed by using var  
IEnumerable<int> filteringQuery =  
    from num in numbers  
    where num < 3 || num > 7  
    select num;  
  
// Query #2.  
IEnumerable<int> orderingQuery =  
    from num in numbers  
    where num < 3 || num > 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];
```

注意，查询类型为 `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。有关详细信息，请参阅 [LINQ 查询介绍](#)。

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，因为它们只表示单个值，不能用作其他查询操作的源。下面的示例演示查询表达式中的方法调用：

```
List<int> numbers1 = new List<int>() { 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 };
List<int> numbers2 = new List<int>() { 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 Action 或 Func 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：

```
// Query #6.
IEnumerable<int> largeNumbersQuery = numbers2.Where(c => c > 15);
```

在上面的查询中，只有查询 #4 立即执行。这是因为它将返回单个值，而不是泛型 **IEnumerable<T>** 集合。该方法本身必须使用 **foreach** 才能计算其值。

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：

```
// var is used for convenience in these queries
var 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。

```
// Query #7.

// Using a query expression with method syntax
int numCount1 =
    (from num in numbers1
     where num < 3 || num > 7
     select num).Count();

// Better: Create a new variable to store
// the method call result
IQueryable<int> numbersQuery =
    from num in numbers1
    where num < 3 || num > 7
    select num;

int 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：

```
var numCount = (from num in numbers...
```

它可以采用方法语法进行编写，如下所示：

```
var numCount = numbers.Where(n => n < 3 || n > 7).Count();
```

它可以使用显式类型化进行编写，如下所示：

```
int numCount = numbers.Where(n => n < 3 || n > 7).Count();
```

另请参阅

- [演练:用 C# 编写查询](#)
- [语言集成查询 \(LINQ\)](#)
- [where 子句](#)

查询对象的集合

2020/3/18 • [Edit Online](#)

此示例演示如何对一系列 `Student` 对象执行简单查询。每个 `Student` 对象均包含一些关于学生的基本信息和表示学生在四门考试中的分数的列表。

该应用程序充当本部分中使用相同 `students` 数据源的许多其他示例的框架。

示例

以下查询将返回在第一堂考试中得分为 90 或更高的学生。

```
public class Student
{
    #region data
    public enum GradeLevel { FirstYear = 1, SecondYear, ThirdYear, FourthYear };

    public string FirstName { get; set; }
    public string LastName { get; set; }
    public int Id { get; set; }
    public GradeLevel Year;
    public List<int> ExamScores;

    protected static List<Student> students = new List<Student>
    {
        new Student {FirstName = "Terry", LastName = "Adams", Id = 120,
            Year = GradeLevel.SecondYear,
            ExamScores = new List<int> { 99, 82, 81, 79}},
        new Student {FirstName = "Fadi", LastName = "Fakhouri", Id = 116,
            Year = GradeLevel.ThirdYear,
            ExamScores = new List<int> { 99, 86, 90, 94}},
        new Student {FirstName = "Hanying", LastName = "Feng", Id = 117,
            Year = GradeLevel.FirstYear,
            ExamScores = new List<int> { 93, 92, 80, 87}},
        new Student {FirstName = "Cesar", LastName = "Garcia", Id = 114,
            Year = GradeLevel.FourthYear,
            ExamScores = new List<int> { 97, 89, 85, 82}},
        new Student {FirstName = "Debra", LastName = "Garcia", Id = 115,
            Year = GradeLevel.ThirdYear,
            ExamScores = new List<int> { 35, 72, 91, 70}},
        new Student {FirstName = "Hugo", LastName = "Garcia", Id = 118,
            Year = GradeLevel.SecondYear,
            ExamScores = new List<int> { 92, 90, 83, 78}},
        new Student {FirstName = "Sven", LastName = "Mortensen", Id = 113,
            Year = GradeLevel.FirstYear,
            ExamScores = new List<int> { 88, 94, 65, 91}},
        new Student {FirstName = "Claire", LastName = "O'Donnell", Id = 112,
            Year = GradeLevel.FourthYear,
            ExamScores = new List<int> { 75, 84, 91, 39}},
        new Student {FirstName = "Svetlana", LastName = "Omelchenko", Id = 111,
            Year = GradeLevel.SecondYear,
            ExamScores = new List<int> { 97, 92, 81, 60}},
        new Student {FirstName = "Lance", LastName = "Tucker", Id = 119,
            Year = GradeLevel.ThirdYear,
            ExamScores = new List<int> { 68, 79, 88, 92}},
        new Student {FirstName = "Michael", LastName = "Tucker", Id = 122,
            Year = GradeLevel.FirstYear,
            ExamScores = new List<int> { 94, 92, 91, 91}},
        new Student {FirstName = "Eugene", LastName = "Zabokritski", Id = 121,
            Year = GradeLevel.FourthYear,
            ExamScores = new List<int> { 96, 85, 91, 60}}
    };
}
```

```

};

#endregion

// Helper method, used in GroupByRange.
protected static int GetPercentile(Student s)
{
    double avg = s.ExamScores.Average();
    return avg > 0 ? (int)avg / 10 : 0;
}

public static void QueryHighScores(int exam, int score)
{
    var highScores = from student in students
                    where student.ExamScores[exam] > score
                    select new {Name = student.FirstName, Score = student.ExamScores[exam]};

    foreach (var item in highScores)
    {
        Console.WriteLine($"{item.Name,-15}{item.Score}");
    }
}

public class Program
{
    public static void Main()
    {
        Student.QueryHighScores(1, 90);

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit");
        Console.ReadKey();
    }
}

```

该查询有意简单，以使你可以进行实验。例如，你可以在 `where` 子句中尝试其他条件或使用 `orderby` 子句对结果进行排序。

另请参阅

- [语言集成查询 \(LINQ\)](#)
- [字符串内插](#)

如何从方法中返回查询 (C# 编程指南)

2020/11/2 • [Edit Online](#)

此示例演示如何以返回值和 `out` 参数形式从方法中返回查询。

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，而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，则认为该方法在返回查询结果（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。

示例

在下面的示例中，第一个方法以返回值的形式返回查询，第二个方法以 `out` 参数的形式返回查询。请注意，在这两种情况下，返回的都是查询，而不是查询结果。

```
class MQ
{
    // QueryMethod1 returns a query as its value.
    IEnumerable<string> QueryMethod1(ref int[] ints)
    {
        var intsToStrings = from i in ints
                            where i > 4
                            select i.ToString();
        return intsToStrings;
    }

    // QueryMethod2 returns a query as the value of parameter returnQ.
    void QueryMethod2(ref int[] ints, out IEnumerable<string> returnQ)
    {
        var intsToStrings = from i in ints
                            where i < 4
                            select i.ToString();
        returnQ = intsToStrings;
    }

    static void Main()
    {
        MQ app = new MQ();

        int[] nums = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };

        // QueryMethod1 returns a query as the value of the method.
        var myQuery1 = app.QueryMethod1(ref nums);

        // Query myQuery1 is executed in the following foreach loop.
        Console.WriteLine("Results of executing myQuery1:");
        // Rest the mouse pointer over myQuery1 to see its type.
        foreach (string s in myQuery1)
        {
            Console.WriteLine(s);
        }

        // You also can execute the query returned from QueryMethod1
        // directly, without using myQuery1.
        Console.WriteLine("\nResults of executing myQuery1 directly:");
        // Rest the mouse pointer over the call to QueryMethod1 to see its
        // return type.
        foreach (string s in app.QueryMethod1(ref nums))
        {
            Console.WriteLine(s);
        }
    }
}
```

```
        }

IEnumerable<string> myQuery2;
// QueryMethod2 returns a query as the value of its out parameter.
app.QueryMethod2(ref nums, out myQuery2);

// Execute the returned query.
Console.WriteLine("\nResults of executing myQuery2:");
foreach (string s in myQuery2)
{
    Console.WriteLine(s);
}

// You can modify a query by using query composition. A saved query
// is nested inside a new query definition that revises the results
// of the first query.
myQuery1 = from item in myQuery1
            orderby item descending
            select item;

// Execute the modified query.
Console.WriteLine("\nResults of executing modified myQuery1:");
foreach (string s in myQuery1)
{
    Console.WriteLine(s);
}

// Keep console window open in debug mode.
Console.WriteLine("Press any key to exit.");
Console.ReadKey();
}
}
```

另请参阅

- [语言集成查询 \(LINQ\)](#)

在内存中存储查询结果

2020/3/18 • [Edit Online](#)

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](#)

建议在存储查询结果时，将返回的集合对象分配给一个新变量，如下面的示例所示：

示例

```
class StoreQueryResults
{
    static List<int> numbers = new List<int>() { 1, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20 };

    static void Main()
    {

        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.
        List<int> factorsofFourList = queryFactorsOfFour.ToList();

        // Iterate the list just to prove it holds data.
        Console.WriteLine(factorsofFourList[2]);
        factorsofFourList[2] = 0;
        Console.WriteLine(factorsofFourList[2]);

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key");
        Console.ReadKey();
    }
}
```

另请参阅

- [语言集成查询 \(LINQ\)](#)

对查询结果进行分组

2020/3/18 • [Edit Online](#)

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](#)。

示例

本主题中的所有示例都使用下列帮助程序类和数据源。

```
public class StudentClass
{
    #region data
    protected enum GradeLevel { FirstYear = 1, SecondYear, ThirdYear, FourthYear };
    protected class Student
    {
        public string FirstName { get; set; }
        public string LastName { get; set; }
        public int ID { get; set; }
        public GradeLevel Year;
        public List<int> ExamScores;
    }

    protected static List<Student> students = new List<Student>
    {
        new Student {FirstName = "Terry", LastName = "Adams", ID = 120,
            Year = GradeLevel.SecondYear,
            ExamScores = new List<int>{ 99, 82, 81, 79}},
        new Student {FirstName = "Fadi", LastName = "Fakhouri", ID = 116,
            Year = GradeLevel.ThirdYear,
            ExamScores = new List<int>{ 99, 86, 90, 94}},
        new Student {FirstName = "Hanying", LastName = "Feng", ID = 117,
            Year = GradeLevel.FirstYear,
            ExamScores = new List<int>{ 93, 92, 80, 87}},
        new Student {FirstName = "Cesar", LastName = "Garcia", ID = 114,
            Year = GradeLevel.FourthYear,
            ExamScores = new List<int>{ 97, 89, 85, 82}},
        new Student {FirstName = "Debra", LastName = "Garcia", ID = 115,
            Year = GradeLevel.ThirdYear,
            ExamScores = new List<int>{ 35, 72, 91, 70}},
        new Student {FirstName = "Hugo", LastName = "Garcia", ID = 118,
            Year = GradeLevel.SecondYear,
            ExamScores = new List<int>{ 92, 90, 83, 78}},
        new Student {FirstName = "Sven", LastName = "Mortensen", ID = 113,
            Year = GradeLevel.FirstYear,
            ExamScores = new List<int>{ 88, 94, 65, 91}},
        new Student {FirstName = "Claire", LastName = "O'Donnell", ID = 112,
            Year = GradeLevel.FourthYear,
            ExamScores = new List<int>{ 96, 85, 78, 89}}
    }
}
```

```

        ExamScores = new List<int>{ 75, 84, 91, 59 },
    new Student {FirstName = "Svetlana", LastName = "Omelchenko", ID = 111,
        Year = GradeLevel.SecondYear,
        ExamScores = new List<int>{ 97, 92, 81, 60 }},
    new Student {FirstName = "Lance", LastName = "Tucker", ID = 119,
        Year = GradeLevel.ThirdYear,
        ExamScores = new List<int>{ 68, 79, 88, 92 }},
    new Student {FirstName = "Michael", LastName = "Tucker", ID = 122,
        Year = GradeLevel.FirstYear,
        ExamScores = new List<int>{ 94, 92, 91, 91 }},
    new Student {FirstName = "Eugene", LastName = "Zabokritski", ID = 121,
        Year = GradeLevel.FourthYear,
        ExamScores = new List<int>{ 96, 85, 91, 60 }}
};

#endregion

//Helper method, used in GroupByRange.
protected static int GetPercentile(Student s)
{
    double avg = s.ExamScores.Average();
    return avg > 0 ? (int)avg / 10 : 0;
}

public void QueryHighScores(int exam, int score)
{
    var highScores = from student in students
                    where student.ExamScores[exam] > score
                    select new {Name = student.FirstName, Score = student.ExamScores[exam]};

    foreach (var item in highScores)
    {
        Console.WriteLine($"{item.Name,-15}{item.Score}");
    }
}
}

public class Program
{
    public static void Main()
    {
        StudentClass sc = new StudentClass();
        sc.QueryHighScores(1, 90);

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit");
        Console.ReadKey();
    }
}

```

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。在此示例中，键是 `string`，即学生的姓氏。还可以使用子字符串作为键。分组操作对该类型使用默认的相等比较器。

将以下方法粘贴到 `StudentClass` 类。将 `Main` 方法中的调用语句更改为 `sc.GroupBySingleProperty()`。

```

public void GroupBySingleProperty()
{
    Console.WriteLine("Group by a single property in an object:");

    // Variable queryLastNames is an IEnumerable<IGrouping<string,
    // DataClass.Student>>.
    var queryLastNames =
        from student in students
        group student by student.LastName into newGroup
        orderby newGroup.Key
        select newGroup;

    foreach (var nameGroup in queryLastNames)
    {
        Console.WriteLine($"Key: {nameGroup.Key}");
        foreach (var student in nameGroup)
        {
            Console.WriteLine($"{student.LastName}, {student.FirstName}");
        }
    }
}

/* Output:
Group by a single property in an object:
Key: Adams
    Adams, Terry
Key: Fakhouri
    Fakhouri, Fadi
Key: Feng
    Feng, Hanying
Key: Garcia
    Garcia, Cesar
    Garcia, Debra
    Garcia, Hugo
Key: Mortensen
    Mortensen, Sven
Key: O'Donnell
    O'Donnell, Claire
Key: Omelchenko
    Omelchenko, Svetlana
Key: Tucker
    Tucker, Lance
    Tucker, Michael
Key: Zabokritski
    Zabokritski, Eugene
*/

```

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。

将以下方法粘贴到 `StudentClass` 类。将 `Main` 方法中的调用语句更改为 `sc.GroupBySubstring()`。

```

public void GroupBySubstring()
{
    Console.WriteLine("\r\nGroup by something other than a property of the object:");

    var queryFirstLetters =
        from student in students
        group student by student.LastName[0];

    foreach (var studentGroup in queryFirstLetters)
    {
        Console.WriteLine($"Key: {studentGroup.Key}");
        // Nested foreach is required to access group items.
        foreach (var student in studentGroup)
        {
            Console.WriteLine($"{student.LastName}, {student.FirstName}");
        }
    }
}

/* Output:
Group by something other than a property of the object:
Key: A
    Adams, Terry
Key: F
    Fakhouri, Fadi
    Feng, Hanying
Key: G
    Garcia, Cesar
    Garcia, Debra
    Garcia, Hugo
Key: M
    Mortensen, Sven
Key: O
    O'Donnell, Claire
    Omelchenko, Svetlana
Key: T
    Tucker, Lance
    Tucker, Michael
Key: Z
    Zabokritski, Eugene
*/

```

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点范围。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。

```

//Helper method, used in GroupByRange.
protected static int GetPercentile(Student s)
{
    double avg = s.ExamScores.Average();
    return avg > 0 ? (int)avg / 10 : 0;
}

```

将以下方法粘贴到 `StudentClass` 类。将 `Main` 方法中的调用语句更改为 `sc.GroupByRange()`。

```

public void GroupByRange()
{
    Console.WriteLine("\r\nGroup by numeric range and project into a new anonymous type:");

    var queryNumericRange =
        from student in students
        let percentile = GetPercentile(student)
        group new { student.FirstName, student.LastName } by percentile into percentGroup
        orderby percentGroup.Key
        select percentGroup;

    // Nested foreach required to iterate over groups and group items.
    foreach (var studentGroup in queryNumericRange)
    {
        Console.WriteLine($"Key: {studentGroup.Key * 10}");
        foreach (var item in studentGroup)
        {
            Console.WriteLine($"{item.LastName}, {item.FirstName}");
        }
    }
}

/* Output:
Group by numeric range and project into a new anonymous type:
Key: 60
    Garcia, Debra
Key: 70
    O'Donnell, Claire
Key: 80
    Adams, Terry
    Feng, Hanying
    Garcia, Cesar
    Garcia, Hugo
    Mortensen, Sven
    Omelchenko, Svetlana
    Tucker, Lance
    Zabokritski, Eugene
Key: 90
    Fakhouri, Fadi
    Tucker, Michael
*/

```

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，结果被投影到一个匿名类型中，因为不需要完整的源元素。请注意，执行查询时，该匿名类型中的属性会变成 `Key` 成员上的属性，并且可以通过名称进行访问。

将以下方法粘贴到 `StudentClass` 类。将 `Main` 方法中的调用语句更改为 `sc.GroupByBoolean()`。

```
public void GroupByBoolean()
{
    Console.WriteLine("\r\nGroup by a Boolean into two groups with string keys");
    Console.WriteLine("\"True\" and \"False\" and project into a new anonymous type:");
    var queryGroupByAverages = from student in students
                                group new { student.FirstName, student.LastName } 
                                by student.ExamScores.Average() > 75 into studentGroup
                                select studentGroup;

    foreach (var studentGroup in queryGroupByAverages)
    {
        Console.WriteLine($"Key: {studentGroup.Key}");
        foreach (var student in studentGroup)
            Console.WriteLine($"{student.FirstName} {student.LastName}");
    }
}
/* Output:
Group by a Boolean into two groups with string keys
"True" and "False" and project into a new anonymous type:
Key: True
    Terry Adams
    Fadi Fakhouri
    Hanying Feng
    Cesar Garcia
    Hugo Garcia
    Sven Mortensen
    Svetlana Omelchenko
    Lance Tucker
    Michael Tucker
    Eugene Zabokritski
Key: False
    Debra Garcia
    Claire O'Donnell
*/
```

示例

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。

将以下方法粘贴到 `StudentClass` 类。将 `Main` 方法中的调用语句更改为 `sc.GroupByCompositeKey()`。

```

public void GroupByCompositeKey()
{
    var queryHighScoreGroups =
        from student in students
        group student by new { FirstLetter = student.LastName[0],
            Score = student.ExamScores[0] > 85 } into studentGroup
        orderby studentGroup.Key.FirstLetter
        select studentGroup;

    Console.WriteLine("\r\nGroup and order by a compound key:");
    foreach (var scoreGroup in queryHighScoreGroups)
    {
        string s = scoreGroup.Key.Score == true ? "more than" : "less than";
        Console.WriteLine($"Name starts with {scoreGroup.Key.FirstLetter} who scored {s} 85");
        foreach (var item in scoreGroup)
        {
            Console.WriteLine($"{item.FirstName} {item.LastName}");
        }
    }
}

/* Output:
Group and order by a compound key:
Name starts with A who scored more than 85
    Terry Adams
Name starts with F who scored more than 85
    Fadi Fakhouri
    Hanying Feng
Name starts with G who scored more than 85
    Cesar Garcia
    Hugo Garcia
Name starts with G who scored less than 85
    Debra Garcia
Name starts with M who scored more than 85
    Sven Mortensen
Name starts with O who scored less than 85
    Claire O'Donnell
Name starts with O who scored more than 85
    Svetlana Omelchenko
Name starts with T who scored less than 85
    Lance Tucker
Name starts with T who scored more than 85
    Michael Tucker
Name starts with Z who scored more than 85
    Eugene Zabokritski
*/

```

另请参阅

- [GroupBy](#)
- [IGrouping< TKey, TElement >](#)
- [语言集成查询 \(LINQ\)](#)
- [group 子句](#)
- [匿名类型](#)
- [对分组操作执行子查询](#)
- [创建嵌套组](#)
- [数据分组](#)

创建嵌套组

2020/3/18 • [Edit Online](#)

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。

示例

NOTE

此示例包含对[查询对象集合内示例代码](#)中所定义对象的引用。

```

public void QueryNestedGroups()
{
    var queryNestedGroups =
        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;

    // Three nested foreach loops are required to iterate
    // over all elements of a grouped group. Hover the mouse
    // cursor over the iteration variables to see their actual type.
    foreach (var outerGroup in queryNestedGroups)
    {
        Console.WriteLine($"DataClass.Student Level = {outerGroup.Key}");
        foreach (var innerGroup in outerGroup)
        {
            Console.WriteLine($"Names that begin with: {innerGroup.Key}");
            foreach (var innerGroupElement in innerGroup)
            {
                Console.WriteLine($"{innerGroupElement.LastName} {innerGroupElement.FirstName}");
            }
        }
    }
}

/*
Output:
DataClass.Student Level = SecondYear
    Names that begin with: Adams
        Adams Terry
    Names that begin with: Garcia
        Garcia Hugo
    Names that begin with: Omelchenko
        Omelchenko Svetlana
DataClass.Student Level = ThirdYear
    Names that begin with: Fakhouri
        Fakhouri Fadi
    Names that begin with: Garcia
        Garcia Debra
    Names that begin with: Tucker
        Tucker Lance
DataClass.Student Level = FirstYear
    Names that begin with: Feng
        Feng Hanying
    Names that begin with: Mortensen
        Mortensen Sven
    Names that begin with: Tucker
        Tucker Michael
DataClass.Student Level = FourthYear
    Names that begin with: Garcia
        Garcia Cesar
    Names that begin with: O'Donnell
        O'Donnell Claire
    Names that begin with: Zabokritski
        Zabokritski Eugene
*/

```

请注意，需要使用 3 个嵌套的 `foreach` 循环来循环访问嵌套组的内部元素。

另请参阅

- [语言集成查询 \(LINQ\)](#)

对分组操作执行子查询

2020/5/20 • [Edit Online](#)

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。请注意，在此特定示例中，最终输出不是组，而是一系列匿名类型。

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。

示例

NOTE

此示例包含对[查询对象集合内示例代码](#)中所定义对象的引用。

```
public void QueryMax()
{
    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()
        };

    int count = queryGroupMax.Count();
    Console.WriteLine($"Number of groups = {count}");

    foreach (var item in queryGroupMax)
    {
        Console.WriteLine($" {item.Level} Highest Score={item.HighestScore}");
    }
}
```

此外，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。

```
public void QueryMaxUsingMethodSyntax()
{
    var queryGroupMax = students
        .GroupBy(student => student.Year)
        .Select(studentGroup => new
    {
        Level = studentGroup.Key,
        HighestScore = studentGroup.Select(student2 => student2.ExamScores.Average()).Max()
    });

    int 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

- [语言集成查询 \(LINQ\)](#)

按连续键对结果进行分组

2020/3/18 • [Edit Online](#)

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：

I	"I"
包含当前请求的 URL 的	We
包含当前请求的 URL 的	think
包含当前请求的 URL 的	that
B	Linq
C	is
包含当前请求的 URL 的	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, !

此解决方案是以扩展方法实现的，该扩展方法是线程安全的，并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，它能在所有序列被读取之前开始将组返回给调用方。

线程安全通过在循环访问源序列的同时创建每个组或区块的副本来实现，如源代码注释中所述。如果源序列具有大型的连续项序列，则公共语言运行时可能会引发 `OutOfMemoryException`。

示例

下面的示例演示该扩展方法以及使用它的客户端代码：

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

namespace ChunkIt
{
    // Static class to contain the extension methods.
    public static class MyExtensions
    {
```

```

public static IEnumerable<IGrouping<TKey, TSource>> ChunkBy<TSouce, TKey>(this IEnumerable<TSouce>
source, Func<TSouce, TKey> keySelector)
{
    return source.ChunkBy(keySelector, EqualityComparer<TKey>.Default);
}

public static IEnumerable<IGrouping<TKey, TSource>> ChunkBy<TSouce, TKey>(this IEnumerable<TSouce>
source, Func<TSouce, TKey> keySelector, IEqualityComparer<TKey> comparer)
{
    // Flag to signal end of source sequence.
    const bool noMoreSourceElements = true;

    // Auto-generated iterator for the source array.
    var enumerator = source.GetEnumerator();

    // Move to the first element in the source sequence.
    if (!enumerator.MoveNext()) yield break;

    // Iterate through source sequence and create a copy of each Chunk.
    // On each pass, the iterator advances to the first element of the next "Chunk"
    // in the source sequence. This loop corresponds to the outer foreach loop that
    // executes the query.
    Chunk<TKey, TSouce> current = null;
    while (true)
    {
        // Get the key for the current Chunk. The source iterator will churn through
        // the source sequence until it finds an element with a key that doesn't match.
        var key = keySelector(enumerator.Current);

        // Make a new Chunk (group) object that initially has one GroupItem, which is a copy of the
        // current source element.
        current = new Chunk<TKey, TSouce>(key, enumerator, value => comparer.Equals(key,
        keySelector(value)));

        // Return the Chunk. A Chunk is an IGrouping<TKey,TSouce>, which is the return value of the
        // ChunkBy method.
        // At this point the Chunk only has the first element in its source sequence. The remaining
        // elements will be
        // returned only when the client code foreach's over this chunk. See Chunk.GetEnumerator for
        // more info.
        yield return current;

        // Check to see whether (a) the chunk has made a copy of all its source elements or
        // (b) the iterator has reached the end of the source sequence. If the caller uses an inner
        // foreach loop to iterate the chunk items, and that loop ran to completion,
        // then the Chunk.GetEnumerator method will already have made
        // copies of all chunk items before we get here. If the Chunk.GetEnumerator loop did not
        // enumerate all elements in the chunk, we need to do it here to avoid corrupting the
        iterator
        // for clients that may be calling us on a separate thread.
        if (current.CopyAllChunkElements() == noMoreSourceElements)
        {
            yield break;
        }
    }

    // A Chunk is a contiguous group of one or more source elements that have the same key. A Chunk
    // has a key and a list of ChunkItem objects, which are copies of the elements in the source
    sequence.
    class Chunk<TKey, TSouce> : IGrouping<TKey, TSouce>
    {
        // 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 = null;
    }

    // The value that is used to determine matching elements
    private readonly TKey key;

    // 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 = false;

    // Private object for thread synchronization
    private object m_Lock;

    // REQUIRES: enumerator != null && predicate != null
    public Chunk(TKey key, Ienumerator<TSource> enumerator, Func<TSource, bool> predicate)
    {
        this.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,
    // and the source enumerator is either at the end, or else on an element with a new key.
    // the tail of the linked list is set to null in the CopyNextChunkElement method if the
    // key of the next element does not match the current chunk's key, or there are no more elements
    in the source.
    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.
        // If MoveNext returns false we are at the end, and isLastSourceElement is set to true
        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 = null;
            predicate = null;
        }
    }
}

```

```

        }

        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. It first checks whether
    // there are more elements in the source sequence. If there are, it
    // Returns true if enumerator for this chunk was exhausted.
    internal bool CopyAllChunkElements()
    {
        while (true)
        {
            lock (m_Lock)
            {
                if (DoneCopyingChunk)
                {
                    // If isLastSourceElement is false,
                    // it signals to the outer iterator
                    // to continue iterating.
                    return isLastSourceElement;
                }
                else
                {
                    CopyNextChunkElement();
                }
            }
        }
    }

    public TKey Key => key;

    // Invoked by the inner foreach loop. This method stays just one step ahead
    // of the client requests. It adds the next element of the chunk only after
    // the clients requests the last element in the list so far.
    public IEnumrator<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();
}

```

```

}

// A simple named type is used for easier viewing in the debugger. Anonymous types
// work just as well with the ChunkBy operator.
public class KeyValuePair
{
    public string Key { get; set; }
    public string Value { get; set; }
}

class Program
{
    // The source sequence.
    public static IEnumerable<KeyValuePair> list;

    // Query variable declared as class member to be available
    // on different threads.
    static IGrouping<string, KeyValuePair>> query;

    static void Main(string[] args)
    {
        // Initialize the source sequence with an array initializer.
        list = new[]
        {
            new KeyValuePair{ Key = "A", Value = "We" },
            new KeyValuePair{ Key = "A", Value = "think" },
            new KeyValuePair{ Key = "A", Value = "that" },
            new KeyValuePair{ Key = "B", Value = "Linq" },
            new KeyValuePair{ Key = "C", Value = "is" },
            new KeyValuePair{ Key = "A", Value = "really" },
            new KeyValuePair{ Key = "B", Value = "cool" },
            new KeyValuePair{ Key = "B", Value = "!" }
        };
        // Create the query by using our user-defined query operator.
        query = list.ChunkBy(p => p.Key);

        // 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($"Group key = {item.Key}");
            foreach (var inner in item)
            {
                Console.WriteLine($"{inner.Value}");
            }
        }

        Console.WriteLine("Press any key to exit");
        Console.ReadKey();
    }
}
}

```

若要在项目中使用扩展方法，请将 `MyExtensions` 静态类复制到新的或现有源代码文件中，并且如有需要，请为它所在的命名空间添加 `using` 指令。

另请参阅

- [语言集成查询 \(LINQ\)](#)

在运行时动态指定谓词筛选器

2020/5/20 • [Edit Online](#)

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。该示例通过两种方式构造。首先，通过对程序中提供的值进行筛选来运行项目。然后通过使用在运行时提供的输入再次运行该项目。

使用 `Contains` 方法进行筛选

1. 打开一个新的控制台应用程序并将其命名为 `PredicateFilters`。
2. 从 [查询对象的集合](#) 复制 `StudentClass` 类，并将其粘贴到类 `Program` 下方的命名空间 `PredicateFilters`。
`StudentClass` 提供 `Student` 对象的列表。
3. 注释禁止 `StudentClass` 中的 `Main` 方法。
4. 将类 `Program` 替换为以下代码：

```
class DynamicPredicates : StudentClass
{
    static void Main(string[] args)
    {
        string[] ids = { "111", "114", "112" };

        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }

    static void QueryByID(string[] ids)
    {
        var queryNames =
            from student in students
            let i = student.ID.ToString()
            where ids.Contains(i)
            select new { student.LastName, student.ID };

        foreach (var name in queryNames)
        {
            Console.WriteLine($"{name.LastName}: {name.ID}");
        }
    }
}
```

5. 将以下行添加到类 `DynamicPredicates` 中 `ids` 声明下的 `Main` 方法。

```
QueryById(ids);
```

6. 运行该项目。

7. 在控制台窗口中显示以下输出：

Garcia: 114

O'Donnell: 112

Omelchenko: 111

8. 下一步是再次运行该项目，此次是通过使用在运行时输入的输入而非数组 `ids` 来执行此操作。在 `Main` 方法中，将 `queryByID(ids)` 更改为 `QueryByID(args)`。
9. 使用命令行参数 `122 117 120 115` 运行该项目。运行项目时，这些值将成为 `args` 的元素，它们是 `Main` 方法的参数。
10. 在控制台窗口中显示以下输出：

Adams: 120

Feng: 117

Garcia: 115

Tucker: 122

使用 switch 语句进行筛选

1. 可以使用 `switch` 语句在预先确定的备选查询中进行选择。在下面的示例中，`studentQuery` 使用不同的 `where` 子句，具体取决于在运行时指定的等级级别或年。
2. 复制下面的方法，并将其粘贴到类 `DynamicPredicates`。

```
// To run this sample, first specify an integer value of 1 to 4 for the command
// line. This number will be converted to a GradeLevel value that specifies which
// set of students to query.
// Call the method: QueryByYear(args[0]);

static void QueryByYear(string level)
{
    GradeLevel year = (GradeLevel)Convert.ToInt32(level);
    IEnumerable<Student> studentQuery = null;
    switch (year)
    {
        case GradeLevel.FirstYear:
            studentQuery = from student in students
                           where student.Year == GradeLevel.FirstYear
                           select student;
            break;
        case GradeLevel.SecondYear:
            studentQuery = from student in students
                           where student.Year == GradeLevel.SecondYear
                           select student;
            break;
        case GradeLevel.ThirdYear:
            studentQuery = from student in students
                           where student.Year == GradeLevel.ThirdYear
                           select student;
            break;
        case GradeLevel.FourthYear:
            studentQuery = from student in students
                           where student.Year == GradeLevel.FourthYear
                           select student;
            break;

        default:
            break;
    }
    Console.WriteLine($"The following students are at level {year}");
    foreach (Student name in studentQuery)
    {
        Console.WriteLine($"{name.LastName}: {name.ID}");
    }
}
```

3. 在 `Main` 方法中，将对 `QueryByID` 的调用替换为以下调用，该调用将 `args` 数组中的第一个元素作为其参数发送：`QueryByYear(args[0])`。

4. 运行该项目，其命令行参数为介于 1 和 4 之间的整数值。

另请参阅

- [语言集成查询 \(LINQ\)](#)
- [where 子句](#)

执行内部联接

2020/5/20 • [Edit Online](#)

在关系数据库术语中，内部联接会生成一个结果集，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` 方法可实现内部联接。

本文演示如何执行内联的四种变体：

- 基于简单键使两个数据源中的元素相关联的简单内部联接。
- 基于复合键使两个数据源中的元素相关联的内部联接。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。
- 在其中将连续联接操作相互追加的多联接。
- 使用分组联接实现的内部联接。

示例 - 简单键联接

下面的示例创建两个集合，其中包含两种用户定义类型 `Person` 和 `Pet` 的对象。查询使用 C# 中的 `join` 子句将 `Person` 对象与 `Owner` 是该 `Person` 的 `Pet` 对象匹配。C# 中的 `select` 子句定义结果对象的外观。在此示例中，结果对象是由所有者名字和宠物姓名组成的匿名类型。

```

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

class Pet
{
    public string Name { get; set; }
    public Person Owner { get; set; }
}

/// <summary>
/// Simple inner join.
/// </summary>
public static void InnerJoinExample()
{
    Person magnus = new Person { FirstName = "Magnus", LastName = "Hedlund" };
    Person terry = new Person { FirstName = "Terry", LastName = "Adams" };
    Person charlotte = new Person { FirstName = "Charlotte", LastName = "Weiss" };
    Person arlene = new Person { FirstName = "Arlene", LastName = "Huff" };
    Person rui = new Person { FirstName = "Rui", LastName = "Raposo" };

    Pet barley = new Pet { Name = "Barley", Owner = terry };
    Pet boots = new Pet { Name = "Boots", Owner = terry };
    Pet whiskers = new Pet { Name = "Whiskers", Owner = charlotte };
    Pet bluemoon = new Pet { Name = "Blue Moon", Owner = rui };
    Pet daisy = new Pet { Name = "Daisy", Owner = magnus };

    // Create two lists.
    List<Person> people = new List<Person> { magnus, terry, charlotte, arlene, rui };
    List<Pet> pets = new List<Pet> { barley, boots, whiskers, bluemoon, daisy };

    // Create a collection of person-pet pairs. Each element in the collection
    // is an anonymous type containing both the person's name and their pet's name.
    var query = from person in people
                join pet in pets on person equals pet.Owner
                select new { OwnerName = person.FirstName, PetName = pet.Name };

    foreach (var ownerAndPet in query)
    {
        Console.WriteLine($"{ownerAndPet.PetName} is owned by {ownerAndPet.OwnerName}");
    }
}

// This code produces the following output:
//
// "Daisy" is owned by Magnus
// "Barley" is owned by Terry
// "Boots" is owned by Terry
// "Whiskers" is owned by Charlotte
// "Blue Moon" is owned by Rui

```

请注意，`Lastname` 是“Huff”的 `Person` 对象未出现在结果集中，因为没有 `Pet` 对象的 `Pet.Owner` 等于该 `Person`。

示例 - 组合键联接

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。为此，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 `Employee` 对象的列表和 `Student` 对象的列表来确定哪些雇员同时还是学生。这两种类型都具有 `String` 类型的 `FirstName` 和 `LastName` 属性。通过每个列表的元素创建联接键的函数会返回由每个元素的

`FirstName` 和 `LastName` 属性组成的匿名类型。联接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。

```
class Employee
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public int EmployeeID { get; set; }
}

class Student
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public int StudentID { get; set; }
}

/// <summary>
/// Performs a join operation using a composite key.
/// </summary>
public static void CompositeKeyJoinExample()
{
    // Create a list of employees.
    List<Employee> employees = new List<Employee> {
        new Employee { FirstName = "Terry", LastName = "Adams", EmployeeID = 522459 },
        new Employee { FirstName = "Charlotte", LastName = "Weiss", EmployeeID = 204467 },
        new Employee { FirstName = "Magnus", LastName = "Hedland", EmployeeID = 866200 },
        new Employee { FirstName = "Vernette", LastName = "Price", EmployeeID = 437139 }};

    // Create a list of students.
    List<Student> students = new List<Student> {
        new Student { FirstName = "Vernette", LastName = "Price", StudentID = 9562 },
        new Student { FirstName = "Terry", LastName = "Earls", StudentID = 9870 },
        new Student { FirstName = "Terry", LastName = "Adams", StudentID = 9913 }};

    // 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 employee in employees
                                  join student in students
                                  on new { employee.FirstName, employee.LastName }
                                  equals new { student.FirstName, student.LastName }
                                  select employee.FirstName + " " + employee.LastName;

    Console.WriteLine("The following people are both employees and students:");
    foreach (string name in query)
        Console.WriteLine(name);
}

// This code produces the following output:
//
// The following people are both employees and students:
// Terry Adams
// Vernette Price
```

示例 - 多联接

可以将任意数量的联接操作相互追加，以执行多联接。C# 中的每个 `join` 子句会将指定数据源与上一个联接的结果相关联。

下面的示例创建三个集合：`Person` 对象的列表、`Cat` 对象的列表和 `Dog` 对象的列表。

C# 中的第一个 `join` 子句基于与 `Cat.Owner` 匹配的 `Person` 对象来匹配人和猫。它返回包含 `Person` 对象和 `Cat.Name` 的匿名类型的序列。

C# 中的第二个 `join` 子句基于由 `Owner` 类型的 `Person` 属性和动物姓名的第一个字母组成的复合键，将第一个联接返回的匿名类型与提供的狗列表中的 `Dog` 对象相关联。它返回包含来自每个匹配对的 `Cat.Name` 和 `Dog.Name` 属性的匿名类型的序列。由于这是内部联接，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。

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

class Pet
{
    public string Name { get; set; }
    public Person Owner { get; set; }
}

class Cat : Pet
{ }

class Dog : Pet
{ }

public static void MultipleJoinExample()
{
    Person magnus = new Person { FirstName = "Magnus", LastName = "Hedlund" };
    Person terry = new Person { FirstName = "Terry", LastName = "Adams" };
    Person charlotte = new Person { FirstName = "Charlotte", LastName = "Weiss" };
    Person arlene = new Person { FirstName = "Arlene", LastName = "Huff" };
    Person rui = new Person { FirstName = "Rui", LastName = "Raposo" };
    Person phyllis = new Person { FirstName = "Phyllis", LastName = "Harris" };

    Cat barley = new Cat { Name = "Barley", Owner = terry };
    Cat boots = new Cat { Name = "Boots", Owner = terry };
    Cat whiskers = new Cat { Name = "Whiskers", Owner = charlotte };
    Cat bluemoon = new Cat { Name = "Blue Moon", Owner = rui };
    Cat daisy = new Cat { Name = "Daisy", Owner = magnus };

    Dog fourwheeldrive = new Dog { Name = "Four Wheel Drive", Owner = phyllis };
    Dog duke = new Dog { Name = "Duke", Owner = magnus };
    Dog denim = new Dog { Name = "Denim", Owner = terry };
    Dog wiley = new Dog { Name = "Wiley", Owner = charlotte };
    Dog snoopy = new Dog { Name = "Snoopy", Owner = rui };
    Dog snickers = new Dog { Name = "Snickers", Owner = arlene };

    // Create three lists.
    List<Person> people =
        new List<Person> { magnus, terry, charlotte, arlene, rui, phyllis };
    List<Cat> cats =
        new List<Cat> { barley, boots, whiskers, bluemoon, daisy };
    List<Dog> dogs =
        new List<Dog> { fourwheeldrive, duke, denim, wiley, snoopy, snickers };

    // The first join matches Person and Cat.Owner from the list of people and
    // cats, based on a common Person. The second join matches dogs whose names start
    // with the same letter as the cats that have the same owner.
    var query = from person in people
                join cat in cats on person equals cat.Owner
                join dog in dogs on
                    new { Owner = person, Letter = cat.Name.Substring(0, 1) }
                    equals new { dog.Owner, Letter = dog.Name.Substring(0, 1) }
                select new { CatName = cat.Name, DogName = dog.Name };

    foreach (var obj in query)
    {
        Console.WriteLine(
            $"The cat \"{obj.CatName}\" shares a house, and the first letter of their name, with \"
            ");
    }
}
```

```

    + the cat "Daisy" shares a house, and the first letter of their name, with "
{obj.DogName}\\".");
}
}

// This code produces the following output:
//
// The cat "Daisy" shares a house, and the first letter of their name, with "Duke".
// The cat "Whiskers" shares a house, and the first letter of their name, with "Wiley".

```

示例 - 使用分组联接的内联

下面的示例演示如何使用分组联接实现内部联接。

在 `query1` 中, `Person` 对象的列表会基于与 `Pet.Owner` 属性匹配的 `Person`, 分组联接到 `Pet` 对象队列中。分组联接会创建中间组的集合, 其中每个组都包含 `Person` 对象和匹配 `Pet` 对象的序列。

通过向查询添加另一个 `from` 子句, 此序列的序列会合并(或平展)为一个较长的序列。最后一个序列的元素的类型由 `select` 子句指定。在此示例中, 该类型是由每个匹配对的 `Person.FirstName` 和 `Pet.Name` 属性组成的匿名类型。

`query1` 的结果等效于通过使用 `join` 子句(不使用 `into` 子句)执行内部联接来获取的结果集。`query2` 变量演示了此等效查询。

```

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

class Pet
{
    public string Name { get; set; }
    public Person Owner { get; set; }
}

/// <summary>
/// Performs an inner join by using GroupJoin().
/// </summary>
public static void InnerGroupJoinExample()
{
    Person magnus = new Person { FirstName = "Magnus", LastName = "Hedlund" };
    Person terry = new Person { FirstName = "Terry", LastName = "Adams" };
    Person charlotte = new Person { FirstName = "Charlotte", LastName = "Weiss" };
    Person arlene = new Person { FirstName = "Arlene", LastName = "Huff" };

    Pet barley = new Pet { Name = "Barley", Owner = terry };
    Pet boots = new Pet { Name = "Boots", Owner = terry };
    Pet whiskers = new Pet { Name = "Whiskers", Owner = charlotte };
    Pet bluemoon = new Pet { Name = "Blue Moon", Owner = terry };
    Pet daisy = new Pet { Name = "Daisy", Owner = magnus };

    // Create two lists.
    List<Person> people = new List<Person> { magnus, terry, charlotte, arlene };
    List<Pet> pets = new List<Pet> { barley, boots, whiskers, bluemoon, daisy };

    var query1 = from person in people
                join pet in pets on person equals pet.Owner into gj
                from subpet in gj
                select new { OwnerName = person.FirstName, PetName = subpet.Name };

    Console.WriteLine("Inner join using GroupJoin():");
    foreach (var v in query1)
    {
        Console.WriteLine($"{v.OwnerName} - {v.PetName}");
    }
}

```

```
}

var query2 = from person in people
    join pet in pets on person equals pet.Owner
    select new { OwnerName = person.FirstName, PetName = pet.Name };

Console.WriteLine("\nThe equivalent operation using Join():");
foreach (var v in query2)
    Console.WriteLine($"{v.OwnerName} - {v.PetName}");
}

// This code produces the following output:
//
// Inner join using GroupJoin():
// Magnus - Daisy
// Terry - Barley
// Terry - Boots
// Terry - Blue Moon
// Charlotte - Whiskers
//
// The equivalent operation using Join():
// Magnus - Daisy
// Terry - Barley
// Terry - Boots
// Terry - Blue Moon
// Charlotte - Whiskers
```

另请参阅

- [Join](#)
- [GroupJoin](#)
- [执行分组联接](#)
- [执行左外部联接](#)
- [匿名类型](#)

执行分组联接

2020/11/2 • [Edit Online](#)

分组联接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。

例如，一个名为 `Student` 的类或关系数据库表可能包含两个字段：`Id` 和 `Name`。另一个名为 `Course` 的类或关系数据库表可能包含两个字段：`StudentId` 和 `CourseTitle`。这两个数据源的分组联接(基于匹配 `Student.Id` 和 `Course.StudentId`)会对具有 `Course` 对象集合(可能为空)的每个 `Student` 进行分组。

NOTE

第一个集合的每个元素都会出现在分组联接的结果集中(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。这与非分组联接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。

WARNING

`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联接进行编写。有关详细信息，请参阅[联接操作](#)和[Entity Framework Core, GroupJoin](#)。

本文的第一个示例演示如何执行分组联接。第二个示例演示如何使用分组联接创建 XML 元素。

示例 - 分组联接

下面的示例基于与 `Pet.Owner` 属性匹配的 `Person`，来执行类型 `Person` 和 `Pet` 的对象的分组联接。与非分组联接(会为每个匹配生成元素对)不同，分组联接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 `Person` 对象)。第二个集合中的对应元素(在此示例中为 `Pet` 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 `Person.FirstName` 和 `Pet` 对象集合。

```

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

class Pet
{
    public string Name { get; set; }
    public Person Owner { get; set; }
}

/// <summary>
/// This example performs a grouped join.
/// </summary>
public static void GroupJoinExample()
{
    Person magnus = new Person { FirstName = "Magnus", LastName = "Hedlund" };
    Person terry = new Person { FirstName = "Terry", LastName = "Adams" };
    Person charlotte = new Person { FirstName = "Charlotte", LastName = "Weiss" };
    Person arlene = new Person { FirstName = "Arlene", LastName = "Huff" };

    Pet barley = new Pet { Name = "Barley", Owner = terry };
    Pet boots = new Pet { Name = "Boots", Owner = terry };
    Pet whiskers = new Pet { Name = "Whiskers", Owner = charlotte };
    Pet bluemoon = new Pet { Name = "Blue Moon", Owner = terry };
    Pet daisy = new Pet { Name = "Daisy", Owner = magnus };

    // Create two lists.
    List<Person> people = new List<Person> { magnus, terry, charlotte, arlene };
    List<Pet> pets = new List<Pet> { barley, boots, whiskers, bluemoon, daisy };

    // Create a list where each element is an anonymous type
    // that contains the person's first name and a collection of
    // pets that are owned by them.
    var query = from person in people
               join pet in pets on person equals pet.Owner into gj
               select new { OwnerName = person.FirstName, Pets = gj };

    foreach (var v in query)
    {
        // Output the owner's name.
        Console.WriteLine($"{v.OwnerName}:");
        // Output each of the owner's pet's names.
        foreach (Pet pet in v.Pets)
            Console.WriteLine($"  {pet.Name}");
    }
}

// This code produces the following output:
//
// Magnus:
//   Daisy
// Terry:
//   Barley
//   Boots
//   Blue Moon
// Charlotte:
//   Whiskers
// Arlene:

```

示例 - 用于创建 XML 的分组联接

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。

```

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

class Pet
{
    public string Name { get; set; }
    public Person Owner { get; set; }
}

/// <summary>
/// This example creates XML output from a grouped join.
/// </summary>
public static void GroupJoinXMLExample()
{
    Person magnus = new Person { FirstName = "Magnus", LastName = "Hedlund" };
    Person terry = new Person { FirstName = "Terry", LastName = "Adams" };
    Person charlotte = new Person { FirstName = "Charlotte", LastName = "Weiss" };
    Person arlene = new Person { FirstName = "Arlene", LastName = "Huff" };

    Pet barley = new Pet { Name = "Barley", Owner = terry };
    Pet boots = new Pet { Name = "Boots", Owner = terry };
    Pet whiskers = new Pet { Name = "Whiskers", Owner = charlotte };
    Pet bluemoon = new Pet { Name = "Blue Moon", Owner = terry };
    Pet daisy = new Pet { Name = "Daisy", Owner = magnus };

    // Create two lists.
    List<Person> people = new List<Person> { magnus, terry, charlotte, arlene };
    List<Pet> pets = new List<Pet> { barley, boots, whiskers, bluemoon, daisy };

    // Create XML to display the hierarchical organization of people and their pets.
    XElement ownersAndPets = new XElement("PetOwners",
        from person in people
        join pet in pets on person equals pet.Owner into gj
        select new XElement("Person",
            new XAttribute("FirstName", person.FirstName),
            new XAttribute("LastName", person.LastName),
            from subpet in gj
            select new XElement("Pet", subpet.Name)));
}

Console.WriteLine(ownersAndPets);
}

// This code produces the following output:
//
// <PetOwners>
//   <Person FirstName="Magnus" LastName="Hedlund">
//     <Pet>Daisy</Pet>
//   </Person>
//   <Person FirstName="Terry" LastName="Adams">
//     <Pet>Barley</Pet>
//     <Pet>Boots</Pet>
//     <Pet>Blue Moon</Pet>
//   </Person>
//   <Person FirstName="Charlotte" LastName="Weiss">
//     <Pet>Whiskers</Pet>
//   </Person>
//   <Person FirstName="Arlene" LastName="Huff" />
// </PetOwners>

```

请参阅

- [Join](#)

- [GroupJoin](#)
- [执行内部联接](#)
- [执行左外部联接](#)
- [匿名类型](#)

执行左外部联接

2020/3/18 • [Edit Online](#)

左外部联接是这样定义的：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 通过对分组联接的结果调用 [DefaultIfEmpty](#) 方法来执行左外部联接。

示例

下面的示例演示如何对分组联接的结果调用 [DefaultIfEmpty](#) 方法来执行左外部联接。

若要生成两个集合的左外部联接，第一步是使用分组联接执行内联。（有关此过程的说明，请参阅[执行内联](#)。）在此示例中，`Person` 对象列表基于与 `Pet.Owner` 匹配的 `Person` 对象内联到 `Pet` 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。这是通过对分组联接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 `Pet` 对象的序列调用 [DefaultIfEmpty](#)。如果对于任何 `Person` 对象，匹配 `Pet` 对象序列均为空，则该方法返回一个包含单个默认值的集合，从而确保结果集合中显示每个 `Person` 对象。

NOTE

引用类型的默认值为 `null`；因此，该示例在访问每个 `Pet` 集合的每个元素之前会先检查是否存在空引用。

```

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

class Pet
{
    public string Name { get; set; }
    public Person Owner { get; set; }
}

public static void LeftOuterJoinExample()
{
    Person magnus = new Person { FirstName = "Magnus", LastName = "Hedlund" };
    Person terry = new Person { FirstName = "Terry", LastName = "Adams" };
    Person charlotte = new Person { FirstName = "Charlotte", LastName = "Weiss" };
    Person arlene = new Person { FirstName = "Arlene", LastName = "Huff" };

    Pet barley = new Pet { Name = "Barley", Owner = terry };
    Pet boots = new Pet { Name = "Boots", Owner = terry };
    Pet whiskers = new Pet { Name = "Whiskers", Owner = charlotte };
    Pet bluemoon = new Pet { Name = "Blue Moon", Owner = terry };
    Pet daisy = new Pet { Name = "Daisy", Owner = magnus };

    // Create two lists.
    List<Person> people = new List<Person> { magnus, terry, charlotte, arlene };
    List<Pet> pets = new List<Pet> { barley, boots, whiskers, bluemoon, daisy };

    var query = from person in people
                join pet in pets on person equals pet.Owner into gj
                from subpet in gj.DefaultIfEmpty()
                select new { person.FirstName, PetName = subpet?.Name ?? String.Empty };

    foreach (var v in query)
    {
        Console.WriteLine($"{v.FirstName} : {v.PetName}");
    }
}

// This code produces the following output:
//
// Magnus:      Daisy
// Terry:       Barley
// Terry:       Boots
// Terry:       Blue Moon
// Charlotte:   Whiskers
// Arlene:

```

请参阅

- [Join](#)
- [GroupJoin](#)
- [执行内部联接](#)
- [执行分组联接](#)
- [匿名类型](#)

对 join 子句的结果进行排序

2020/3/18 • [Edit Online](#)

此示例演示如何对联接运算的结果进行排序。请注意，排序在联接之后执行。虽然可以在联接之前将 `orderby` 子句用于一个或多个源序列，不过通常不建议这样做。某些 LINQ 提供程序可能不会在联接之后保留该排序。

示例

此查询创建一个分组联接，然后基于类别元素（仍处于范围内）对组进行排序。在匿名类型初始值设定项内，一个子查询对来自产品序列的所有匹配元素进行排序。

```
class HowToOrderJoins
{
    #region Data
    class Product
    {
        public string Name { get; set; }
        public int CategoryID { get; set; }
    }

    class Category
    {
        public string Name { get; set; }
        public int ID { get; set; }
    }

    // Specify the first data source.
    List<Category> categories = new List<Category>()
    {
        new Category(){Name="Beverages", ID=001},
        new Category(){ Name="Condiments", ID=002},
        new Category(){ Name="Vegetables", ID=003},
        new Category() { Name="Grains", ID=004},
        new Category() { Name="Fruit", ID=005}
    };

    // Specify the second data source.
    List<Product> products = new List<Product>()
    {
        new Product{Name="Cola", CategoryID=001},
        new Product{Name="Tea", CategoryID=001},
        new Product{Name="Mustard", CategoryID=002},
        new Product{Name="Pickles", CategoryID=002},
        new Product{Name="Carrots", CategoryID=003},
        new Product{Name="Bok Choy", CategoryID=003},
        new Product{Name="Peaches", CategoryID=005},
        new Product{Name="Melons", CategoryID=005},
    };
    #endregion
    static void Main()
    {
        HowToOrderJoins app = new HowToOrderJoins();
        app.OrderJoin1();

        // Keep console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}

void OrderJoin1()
{
}
```

```

    var groupJoinQuery2 =
        from category in categories
        join prod in products on category.ID equals prod.CategoryID into prodGroup
        orderby category.Name
        select new
        {
            Category = category.Name,
            Products = from prod2 in prodGroup
                        orderby prod2.Name
                        select prod2
        };
    foreach (var productGroup in groupJoinQuery2)
    {
        Console.WriteLine(productGroup.Category);
        foreach (var prodItem in productGroup.Products)
        {
            Console.WriteLine($" {prodItem.Name,-10} {prodItem.CategoryID}");
        }
    }
}
/* Output:
   Beverages
     Cola      1
     Tea       1
   Condiments
     Mustard   2
     Pickles   2
   Fruit
     Melons    5
     Peaches   5
   Grains
   Vegetables
     Bok Choy  3
     Carrots   3
*/
}

```

另请参阅

- [语言集成查询 \(LINQ\)](#)
- [orderby 子句](#)
- [join 子句](#)

使用组合键进行联接

2020/3/18 • [Edit Online](#)

此示例演示如何执行想要使用多个键来定义匹配的联接操作。使用组合键来完成此操作。以匿名类型或包含要比较值的命名类型的形式来创建组合键。若要跨方法边界传递查询变量，请为该键使用重写 [Equals](#) 和 [GetHashCode](#) 的命名类型。属性的名称以及属性出现的顺序在每个键中必须相同。

示例

以下示例演示如何使用组合键联接 3 个表中的数据：

```
var query = from o in db.Orders
    from p in db.Products
    join d in db.OrderDetails
        on new {o.OrderID, p.ProductID} equals new {d.OrderID, d.ProductID} into details
        from d in details
    select new {o.OrderID, p.ProductID, d.UnitPrice};
```

组合键上的类型推理取决于这些键中属性的名称，以及属性出现的顺序。如果源序列中属性的名称不同，则必须在键中分配新名称。例如，如果 `Orders` 表和 `OrderDetails` 表为各自的列分别使用不同的名称，则可通过在匿名类型中分配相同的名称来创建组合键：

```
join...on new {Name = o.CustomerName, ID = o.CustID} equals
    new {Name = d.CustName, ID = d.CustID }
```

还可以在 `group` 子句中使用组合键。

另请参阅

- [语言集成查询 \(LINQ\)](#)
- [join 子句](#)
- [group 子句](#)

执行自定义联接操作

2020/3/18 • [Edit Online](#)

此示例演示如何执行无法使用 `join` 子句实现的联接操作。在查询表达式中，`join` 子句只限于同等联接（并针对其进行优化），这类联接到目前为止是最常见的联接操作类型。执行同等联接时，可能会始终使用 `join` 子句获得最佳性能。

但是，在以下情况下不能使用 `join` 子句：

- 当联接依据不等式表达式时（非同等联接）。
- 当联接依据多个等式或不等式表达式时。
- 当必须为联接操作前的右侧（内部）序列引入临时范围变量。

若要执行不是同等联接的联接，可以使用多个 `from` 子句独立引入每个数据源。随后可在 `where` 子句中将谓词表达式应用于每个源的范围变量。表达式还可以采用方法调用的形式。

NOTE

不要将这种类型的自定义联接操作与使用多个 `from` 子句访问内部集合相混淆。有关详细信息，请参阅 [join 子句](#)。

示例

以下示例中的第一个方法演示一个简单的交叉联接。必须谨慎使用交叉联接，因为它们可能会生成非常大的结果集。但是，在创建针对其运行其他查询的源序列的某些情况下，它们可能会十分有用。

第二个方法会生成其类别 ID 在左侧类别列表中列出的所有产品的序列。请注意，其中使用 `let` 子句和 `Contains` 方法创建临时数组。还可以在查询前创建数组并去掉第一个 `from` 子句。

```
class CustomJoins
{
    #region Data

    class Product
    {
        public string Name { get; set; }
        public int CategoryID { get; set; }
    }

    class Category
    {
        public string Name { get; set; }
        public int ID { get; set; }
    }

    // Specify the first data source.
    List<Category> categories = new List<Category>()
    {
        new Category(){Name="Beverages", ID=001},
        new Category(){ Name="Condiments", ID=002},
        new Category(){ Name="Vegetables", ID=003},
    };

    // Specify the second data source.
    List<Product> products = new List<Product>()
```

```

    {
        new Product{Name="Tea", CategoryID=001},
        new Product{Name="Mustard", CategoryID=002},
        new Product{Name="Pickles", CategoryID=002},
        new Product{Name="Carrots", CategoryID=003},
        new Product{Name="Bok Choy", CategoryID=003},
        new Product{Name="Peaches", CategoryID=005},
        new Product{Name="Melons", CategoryID=005},
        new Product{Name="Ice Cream", CategoryID=007},
        new Product{Name="Mackerel", CategoryID=012},
    };
#endregion

static void Main()
{
    CustomJoins app = new CustomJoins();
    app.CrossJoin();
    app.NonEquijoin();

    Console.WriteLine("Press any key to exit.");
    Console.ReadKey();
}

void CrossJoin()
{
    var crossJoinQuery =
        from c in categories
        from p in products
        select new { c.ID, p.Name };

    Console.WriteLine("Cross Join Query:");
    foreach (var v in crossJoinQuery)
    {
        Console.WriteLine($"{v.ID,-5}{v.Name}");
    }
}

void NonEquijoin()
{
    var nonEquijoinQuery =
        from p in products
        let catIds = from c in categories
                    select c.ID
        where catIds.Contains(p.CategoryID) == true
        select new { Product = p.Name, CategoryID = p.CategoryID };

    Console.WriteLine("Non-equijoin query:");
    foreach (var v in nonEquijoinQuery)
    {
        Console.WriteLine($"{v.CategoryID,-5}{v.Product}");
    }
}

/* Output:
Cross Join Query:
1   Tea
1   Mustard
1   Pickles
1   Carrots
1   Bok Choy
1   Peaches
1   Melons
1   Ice Cream
1   Mackerel
2   Tea
2   Mustard
2   Pickles
2   Carrots
2   Bok Choy

```

```
2 Peaches
2 Melons
2 Ice Cream
2 Mackerel
3 Tea
3 Mustard
3 Pickles
3 Carrots
3 Bok Choy
3 Peaches
3 Melons
3 Ice Cream
3 Mackerel
Non-equijoin query:
1 Tea
2 Mustard
2 Pickles
3 Carrots
3 Bok Choy
Press any key to exit.
*/
```

示例

在下面的示例中，查询必须基于匹配键（对于内部（右侧）序列，无法在 `join` 子句本身之前获取这些键）联接两个序列。如果使用 `join` 子句执行了此联接，则必须为每个元素调用 `Split` 方法。使用多个 `from` 子句可使查询避免重复进行方法调用的开销。但是，因为 `join` 经过了优化，所有在此特定情况下，它可能仍然比使用多个 `from` 子句更快。结果的变化主要取决于方法调用的成本高低。

```
class MergeTwoCSVFiles
{
    static void Main()
    {
        // See section Compiling the Code for information about the data files.
        string[] names = System.IO.File.ReadAllLines(@"../../names.csv");
        string[] scores = System.IO.File.ReadAllLines(@"../../scores.csv");

        // Merge the data sources using a named type.
        // You could use var instead of an explicit type for the query.
        IEnumerable<Student> queryNamesScores =
            // Split each line in the data files into an array of strings.
            from name in names
            let x = name.Split(',')
            from score in scores
            let s = score.Split(',')
            // Look for matching IDs from the two data files.
            where x[2] == s[0]
            // If the IDs match, build a Student object.
            select new Student()
            {
                FirstName = x[0],
                LastName = x[1],
                ID = Convert.ToInt32(x[2]),
                ExamScores = (from scoreAsText in s.Skip(1)
                              select Convert.ToInt32(scoreAsText)).
                              ToList()
            };

        // Optional. Store the newly created student objects in memory
        // for faster access in future queries
        List<Student> students = queryNamesScores.ToList();

        foreach (var student in students)
        {
            Console.WriteLine($"The average score of {student.FirstName} {student.LastName} is
```

```
    Console.WriteLine("The average score of {student.FirstName} {student.LastName} is {student.Average()}.");

}

//Keep console window open in debug mode
Console.WriteLine("Press any key to exit.");
Console.ReadKey();
}

}

class Student
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public int ID { get; set; }
    public List<int> ExamScores { get; set; }
}

/* 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.
*/
```

另请参阅

- [语言集成查询 \(LINQ\)](#)
- [join 子句](#)
- [对 Join 子句的结果进行排序](#)

在查询表达式中处理 null 值

2020/4/2 • [Edit Online](#)

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`。

示例

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：

```
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。

示例

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：

```
void TestMethod(Northwind db)
{
    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 };
}
```

请参阅

- [Nullable<T>](#)
- [语言集成查询 \(LINQ\)](#)
- [可以为 null 的值类型](#)

在查询表达式中处理异常

2020/3/18 • [Edit Online](#)

在查询表达式的上下文中可以调用任何方法。但是，我们建议避免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。

示例

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，才可以执行此操作。

```
class ExceptionsOutsideQuery
{
    static void Main()
    {
        // DO THIS with a datasource that might
        // throw an exception. It is easier to deal with
        // outside of the query expression.
        I Enumerable<int> dataSource;
        try
        {
            dataSource = GetData();
        }
        catch (InvalidOperationException)
        {
            // Handle (or don't handle) the exception
            // in the way that is appropriate for your application.
            Console.WriteLine("Invalid operation");
            goto Exit;
        }

        // 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());

        //Keep the console window open in debug mode
        Exit:
        Console.WriteLine("Press any key to exit");
        Console.ReadKey();
    }

    // A data source that is very likely to throw an exception!
    static I Enumerable<int> GetData()
    {
        throw new InvalidOperationException();
    }
}
```

示例

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。

请注意，`try` 块封装 `foreach` 循环，且不对自身进行查询。这是由于 `foreach` 循环正是实际执行查询时的点。有关详细信息，请参阅 [LINQ 查询简介](#)。

```
class QueryThatThrows
{
    static void Main()
    {
        // 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;

        // Runtime exceptions are thrown when query is executed.
        // Therefore they must be handled in the foreach loop.
        try
        {
            foreach (var item in exceptionDemoQuery)
            {
                Console.WriteLine($"Processing {item}");
            }
        }

        // Catch whatever exception you expect to raise
        // and/or do any necessary cleanup in a finally block
        catch (InvalidOperationException e)
        {
            Console.WriteLine(e.Message);
        }

        //Keep the console window open in debug mode
        Console.WriteLine("Press any key to exit");
        Console.ReadKey();
    }

    // Not very useful as a general purpose method.
    static string SomeMethodThatMightThrow(string s)
    {
        if (s[4] == 'C')
            throw new InvalidOperationException();
        return @"C:\newFolder\" + s;
    }
}
/* Output:
   Processing C:\newFolder\fileA.txt
   Processing C:\newFolder\fileB.txt
   Operation is not valid due to the current state of the object.
*/
```

另请参阅

- [语言集成查询 \(LINQ\)](#)

模式匹配概述

2021/5/10 • [Edit Online](#)

“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的新变量。“`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。你可以使用丰富的词汇表达算法。

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：

```
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：

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

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.IDisposable` 接口。如果存在，则其将调用 `Dispose()` 方法用于该对象。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 `IDisposable` 的类型之外，以下代码还可防范 `null`。

```
if (heldReference is IDisposable disposable)
{
    disposable.Dispose();
}
heldReference = null;
```

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：

```
public State PerformOperation(Operation command) =>
    command switch
    {
        Operation.SystemTest => RunDiagnostics(),
        Operation.Start => StartSystem(),
        Operation.Stop => StopSystem(),
        Operation.Reset => ResetToReady(),
        _ => throw new ArgumentException(nameof(command), "Invalid enum value for 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。如果省略开关臂，编译器会警告你尚未处理所有可能输入值。在运行时，如果检查的对象与任何开关臂均不匹配，则 `switch` 表达式会引发异常。可以使用数值常量代替枚举值集。你还可以将这种方法用于表示命令的常量字符串值：

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

```
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。最后两个开关臂用于处理熔点和沸点的案例。如果没有这两个开关臂，编译器将警告你的逻辑未涵盖每个可能的输入。

多个输入

到目前为止，你所看到的所有模式都在检查一个输入。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：

```
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：

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

前两个开关臂检查 `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：

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

上述代码演示了[位置模式](#)，其中表达式的属性已析构。

本文介绍了可以使用 C# 中的模式匹配写入的代码类型。下文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

- [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](#)

编写安全有效的 C# 代码

2021/5/7 • [Edit Online](#)

借助 C# 中的新增功能可编写性能更好的可验证安全代码。若仔细地应用这些技术，则需要不安全代码的方案更少。利用这些功能，可更容易地将对值类型的引用用作方法参数和方法返回。安全完成后，这些技术可以最大程度地减少值类型的复制操作。通过使用值类型，可以使分配和垃圾回收过程的数量降至最低。

本文中的很多示例代码都使用了 C# 7.2 中增加的功能。要使用这些功能，必须将项目配置为使用 C# 7.2 或更高版本。有关设置语言版本的详细信息，请参阅[配置语言版本](#)。

本文重点介绍有效资源管理的技术。使用值类型的优点之一是通常可避免堆分配。缺点是它们按值进行复制。由于存在这种折衷，因此难以优化针对大量数据执行的算法。C# 7.2 中新增的语言功能提供了可使用对值类型的引用来实现安全高效代码的机制。请恰当地使用这些功能，以最大程度地减少分配和复制操作。本文将介绍这些新功能。

本文重点介绍以下资源管理技术：

- 声明一个 `readonly struct` 以表示类型是不可变的。这使编译器可以在使用 `in` 参数时保存防御性副本。
- 如果类型是可变的，请声明 `struct` 成员 `readonly`，以指示该成员不修改状态。
- 当返回值 `struct` 大于 `IntPtr.Size` 且存储生存期大于返回值的方法时，请使用 `ref readonly` 返回。
- 当 `readonly struct` 的大小大于 `IntPtr.Size` 时，出于性能原因，应将其作为 `in` 参数传递。
- 除非使用 `readonly` 修饰符声明 `struct` 或方法仅调用该结构的 `readonly` 成员，否则切勿将其作为 `in` 参数传递。不遵守该指南可能会对性能产生负面影响，并可能导致不明确的行为。
- 使用 `ref struct` 或 `readonly ref struct`（例如 `Span<T>` 或 `ReadOnlySpan<T>`）将内存用作字节序列。

这些技术迫使你在“引用”和“值”方面平衡两个相互竞争的目标。属于[引用类型](#)的变量包含对内存中位置的引用。属于[值类型](#)的变量直接包含它们的值。这些差异突出了对管理内存资源非常重要的关键差异。通常在将“值类型”传递给方法或从方法返回时将其复制。此行为包括在调用值类型的成员时复制 `this` 的值。副本的成本与类型的大小有关。托管堆上分配了“引用类型”。每个新对象都需要一个新的分配，并且随后必须回收。这两种操作都需要花些时间。将引用类型作为参数传递给方法或从方法返回时，将复制引用。

本文使用以下三维点结构的示例概念来解释这些建议：

```
public struct Point3D
{
    public double X;
    public double Y;
    public double Z;
}
```

不同的示例使用该概念的不同实现。

声明不可变值类型的只读结构

使用 `readonly` 修饰符声明 `struct` 将通知编译器你的意图是创建不可变类型。编译器使用以下规则强制执行该设计决策：

- 所有字段成员必须为 `readonly`
- 所有属性都必须是只读的，包括自动实现的属性。

这两个规则足以确保 `readonly struct` 的任何成员都不会修改该结构的状态。`struct` 是不可变的。`Point3D` 结构可以定义为不可变结构，如以下示例所示：

```

readonly public struct ReadonlyPoint3D
{
    public ReadonlyPoint3D(double x, double y, double z)
    {
        this.X = x;
        this.Y = y;
        this.Z = z;
    }

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

```

只要你的设计意图是创建不可变值类型，就请遵循此建议。任何性能改进都是额外权益。`readonly struct` 清楚地表达了你的设计意图。

结构可变时声明 readonly 成员

在 C# 8.0 及更高版本中，结构类型为可变类型时，应将不会引起变化的成员声明为 `readonly`。请考虑其他需要三维点结构的应用程序，但必须支持可变性。以下版本的三维点结构仅将 `readonly` 修饰符添加到不修改结构的成员。当你的设计必须支持某些成员对结构的修改时，但仍然需要对某些成员强制执行只读操作的便利时，请遵循以下示例：

```

public struct Point3D
{
    public Point3D(double x, double y, double z)
    {
        _x = x;
        _y = y;
        _z = z;
    }

    private double _x;
    public double X
    {
        readonly get => _x;
        set => _x = value;
    }

    private double _y;
    public double Y
    {
        readonly get => _y;
        set => _y = value;
    }

    private double _z;
    public double Z
    {
        readonly get => _z;
        set => _z = value;
    }

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

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

```

前面的示例介绍了可在其中应用 `readonly` 修饰符的许多位置：方法、属性和属性访问器。如果使用自动实现的属性，则编译器会将 `readonly` 修饰符添加到 `get` 访问器以获取读写属性。对于仅具有 `get` 访问器的属性，编

译器会将 `readonly` 修饰符添加到自动实现的属性声明中。

向不改变状态的成员添加 `readonly` 修饰符有两个相关的好处。首先，编译器会强制执行你的意图。该成员无法改变结构的状态。其次，访问 `readonly` 成员时，编译器不会创建 `in` 参数的防御性副本。编译器可以安全地进行此优化，因为它可以保证 `readonly` 成员不会修改 `struct`。

尽可能对大型结构使用 `ref readonly return` 语句

当返回的值不是返回方法的本地值时，可以按引用返回值。按引用返回意味着仅复制引用，而不是结构。在以下示例中，`Origin` 属性不能使用 `ref` 返回，因为返回的值是局部变量：

```
public Point3D Origin => new Point3D(0,0,0);
```

但是，可以按引用返回以下属性定义，因为返回的值是静态成员：

```
public struct Point3D
{
    private static Point3D origin = new Point3D(0,0,0);

    // Dangerous! returning a mutable reference to internal storage
    public ref Point3D Origin => ref origin;

    // other members removed for space
}
```

你不希望调用方修改原点，所以应该通过 `ref readonly` 返回值：

```
public struct Point3D
{
    private static Point3D origin = new Point3D(0,0,0);

    public static ref readonly Point3D Origin => ref origin;

    // other members removed for space
}
```

通过返回 `ref readonly` 可以保存复制较大的结构并保留内部数据成员的不变性。

在调用站点，调用方可以选择将 `Origin` 属性用作 `ref readonly` 或值：

```
var originValue = Point3D.Origin;
ref readonly var originReference = ref Point3D.Origin;
```

前面的代码中的第一个分配将创建 `Origin` 常数的副本，并分配该副本。第二个将分配引用。注意，`readonly` 修饰符必须包含在变量声明中。无法修改该修饰符引用对象的引用。尝试执行该操作将导致编译时错误。

`originReference` 的声明需要 `readonly` 修饰符。

编译器可强制使调用方不能修改引用。直接分配给该值的尝试会生成编译时错误。在其他情况下，编译器会分配防御性副本，除非它可安全地使用只读引用。静态分析规则会确定是否可修改结构。当结构为 `readonly struct`，或者成员为结构的 `readonly` 成员时，编译器不会创建防御性副本。无需使用防御性副本即可将结构作为 `in` 参数进行传递。

将 `in` 修饰符应用于大于 `System.IntPtr.Size` 的 `readonly struct` 参数

`in` 关键字补充了现有的 `ref` 和 `out` 关键字，以按引用传递参数。`in` 关键字指定按引用传递参数，但调用的方法不修改值。

这一新增功能可提供完整的词汇，以表达你的设计意图。如果未在方法签名中指定以下任一修饰符，值类型会在传递给调用的方法时进行复制。每个修饰符指定变量按引用传递，避免复制操作。每个修饰符表达一种不同的意图：

- `out`：此方法设置用作此形参的实参的值。
- `ref`：此方法可设置用作此形参的实参的值。
- `in`：此方法不会修改用作此形参的实参的值。

添加 `in` 修饰符，按引用传递参数，并声明设计意图是为了按引用传递参数，避免不必要的复制操作。你不打算修改用作该参数的对象。

这种做法通常可以提高大于 `IntPtr.Size` 的只读值类型的性能。对于简单类型(`sbyte`、`byte`、`short`、`ushort`、`int`、`uint`、`long`、`ulong`、`char`、`float`、`double`、`decimal` 和 `bool` 以及 `enum` 类型)，任何潜在的性能提升都是极小的。实际上，对于小于 `IntPtr.Size` 的类型，使用按引用传递可能会降低性能。

下面的代码演示了一个方法示例，该方法用于计算三维空间中两点间的距离。

```
private static double CalculateDistance(in Point3D point1, in Point3D point2)
{
    double xDifference = point1.X - point2.X;
    double yDifference = point1.Y - point2.Y;
    double zDifference = point1.Z - point2.Z;

    return Math.Sqrt(xDifference * xDifference + yDifference * yDifference + zDifference * zDifference);
}
```

该参数具有两个结构，每个结构包含三个双精度值。一个双精度值有 8 个字节，所以每个参数有 24 个字节。通过指定 `in` 修饰符，可向这些参数传递 4 字节或 8 字节引用，具体取决于计算机的体系结构。大小的差异很小，但是当应用程序使用许多不同的值在一个紧凑的循环中调用此方法时，这些差异将累积。

`in` 修饰符还可以通过其他方式补充 `out` 和 `ref`。无法创建差异仅为是否具有 `in`、`out` 或 `ref` 的方法的重载。这些新规则可扩展始终为 `out` 和 `ref` 参数定义的相同行为。与 `out` 和 `ref` 修饰符类似，值类型未装箱，因为应用了 `in` 修饰符。

`in` 修饰符可能适用于采用以下参数的任何成员：`methods`、`delegates`、`lambdas`、`local functions`、`indexers` 和 `operators`。

`in` 实参的另一个功能是可对 `in` 形参的实参使用文本值或常数。此外，与 `ref` 或 `out` 参数不同，无需在调用站点应用 `in` 修饰符。下面的代码演示调用 `calculateDistance` 方法的两个示例。第一个示例使用按引用传递的两个本地变量。第二个示例包含方法调用过程中创建的临时变量。

```
var distance = CalculateDistance(pt1, pt2);
var fromOrigin = CalculateDistance(pt1, new Point3D());
```

有多种方法可让编译器确保强制执行 `in` 参数的只读性质。首先，调用的方法不能直接分配给 `in` 参数。当值类型为 `struct` 时，不能直接分配给 `in` 参数的任何字段。此外，不能向使用 `ref` 或 `out` 修饰符的任何方法传递 `in` 参数。如果字段为 `struct` 类型且该参数也为 `struct` 类型，则这些规则适用于 `in` 参数的任何字段。事实上，如果所有级别的成员访问类型都是 `structs`，则这些规则适用于多层成员访问。编译器强制将 `struct` 类型作为 `in` 参数传递，它们的 `struct` 成员用作其他方法的参数时，为只读变量。

使用 `in` 参数可避免产生复制操作可能产生的性能成本。这不会改变任何方法调用的语义。所以不需要在调用站点指定 `in` 修饰符。在调用站点省略 `in` 修饰符就会通知编译器你允许它出于以下原因复制参数：

- 存在从实参类型到形参类型的隐式转换，但不是标识转换。
- 该参数是一个表达式，但是没有已知的存储变量。
- 存在的重载因 `in` 是否存在而有所不同。在这种情况下，按值重载的匹配度会更高。

在更新现有代码以使用只读引用参数时，这些规则会很有用。在调用的方法中，可以调用任何使用按值参数的实例方法。在这些方法中，将创建 `in` 参数的副本。由于编译器可为任何 `in` 参数创建临时变量，因此还可指定任何 `in` 参数的默认值。以下代码指定原点(点 0,0)为第二个点的默认值：

```
private static double CalculateDistance2(in Point3D point1, in Point3D point2 = default)
{
    double xDifference = point1.X - point2.X;
    double yDifference = point1.Y - point2.Y;
    double zDifference = point1.Z - point2.Z;

    return Math.Sqrt(xDifference * xDifference + yDifference * yDifference + zDifference * zDifference);
}
```

要强制编译器按引用传递只读参数，请在调用站点的参数上指定 `in` 修饰符，如下列代码所示：

```
distance = CalculateDistance(in pt1, in pt2);
distance = CalculateDistance(in pt1, new Point3D());
distance = CalculateDistance(pt1, in Point3D.Origin);
```

这样可以更轻松地在大型代码库中采用一段时间的 `in` 参数，从而实现性能提升。首先，将 `in` 修饰符添加到方法签名。然后，可以在调用站点添加 `in` 修饰符，并创建 `readonly struct` 类型，让编译器避免在更多位置创建 `in` 参数的防御性副本。

`in` 参数指定还可用于引用类型或数值。但是，这两种情况下获得的好处都是最少的(如果有)。

避免在 `in` 参数中使用可变结构

上述技术解释了如何通过返回引用和按引用传递值来避免创建副本。当参数类型声明为 `readonly struct` 类型时，这些技术最有效。否则，编译器必须在许多情况下创建“防御副本”以强制执行任何参数的只读状态。请考虑下面这个计算三维点到原点距离的示例：

```
private static double CalculateDistance(in Point3D point1, in Point3D point2)
{
    double xDifference = point1.X - point2.X;
    double yDifference = point1.Y - point2.Y;
    double zDifference = point1.Z - point2.Z;

    return Math.Sqrt(xDifference * xDifference + yDifference * yDifference + zDifference * zDifference);
}
```

`Point3D` 结构不是只读结构。此方法的主体中有六个不同的属性访问调用。在首次检查时，你可能认为这些访问是安全的。毕竟，`get` 访问器不应该修改对象的状态。但是没有强制执行的语言规则。它只是通用约定。任何类型都可以实现修改内部状态的 `get` 访问器。如果没有语言保证，编译器必须在调用任何未标记为 `readonly` 修饰符的成员之前创建参数的临时副本。在堆栈上创建临时存储，将参数的值复制到临时存储中，并将每个成员访问的值作为 `this` 参数复制到堆栈中。在许多情况下，当参数类型不是 `readonly struct`，并且该方法调用成员未标记为 `readonly` 时，这些副本会降低性能，使得按值传递比按只读引用传递速度更快。如果将不修改结构状态的所有方法标记为 `readonly`，编译器就可以安全地确定不修改结构状态，并且不需要防御性复制。

相反，如果距离计算使用不可变结构 `ReadonlyPoint3D`，则不需要临时对象：

```
private static double CalculateDistance3(in ReadonlyPoint3D point1, in ReadonlyPoint3D point2 = default)
{
    double xDifference = point1.X - point2.X;
    double yDifference = point1.Y - point2.Y;
    double zDifference = point1.Z - point2.Z;

    return Math.Sqrt(xDifference * xDifference + yDifference * yDifference + zDifference * zDifference);
}
```

当你调用 `readonly struct` 的成员时，编译器会生成更有效的代码：`this` 引用始终是按引用成员方法传递的 `in` 参数，而不是接收器的副本。将 `readonly struct` 用作 `in` 参数时，此优化可以减少复制操作。

不应将可为 null 的值类型作为 `in` 参数传递。`Nullable<T>` 类型未声明为只读结构。这意味着编译器必须为使用参数声明中的 `in` 修饰符传递到方法的任何可以为 null 的值类型参数生成防御性副本。

你可以在 GitHub 上的[示例存储库](#)中看到使用 `BenchmarkDotNet` 演示性能差异的示例程序。它对按值和按引用传递可变结构与按值和按引用传递不可变结构进行了比较。使用不可变结构并按引用传递是最快的。

使用 `ref struct` 类型处理单个堆栈帧上的块或内存

相关语言功能是可声明必须约束为单个堆栈帧的值类型。此限制可使编译器进行多次优化。此功能的主要动机是 `Span<T>` 和相关结构。借助使用 `Span<T>` 类型的新的和更新后的 .NET API，可通过这些增强功能实现性能改进。

在使用通过 `stackalloc` 创建的内存或使用互操作 API 中的内存时，可能具有类似要求。可针对这些需求定义自己的 `ref struct` 类型。

`readonly ref struct` 类型

将结构声明为 `readonly ref` 兼具 `ref struct` 和 `readonly struct` 声明的优点和限制。只读跨度所使用的内存仅限于单个堆栈帧，并且只读跨度使用的内存无法进行修改。

结论

使用值类型可最大限度地减少分配操作的数量：

- 值类型的存储是为局部变量和方法参数分配的堆栈。
- 作为其他对象成员的值类型的存储被分配为该对象的一部分，而不是作为单独的分配。
- 值类型返回值的存储为堆栈分配。

将其与相同情况下的引用类型进行对比：

- 引用类型的存储是为本地变量和方法参数分配的堆。引用存储在堆栈中。
- 作为其他对象成员的引用类型的存储在堆上分别进行分配。包含的对象存储引用。
- 引用类型返回值的存储是堆分配的。对该存储的引用存储在堆栈中。

最小化分配附带权衡。当 `struct` 的大小大于引用大小时，可以复制更多内存。引用通常为 64 位或 32 位，并且取决于目标机器 CPU。

这些权衡通常对性能影响最小。但是，对于大型结构或大型集合，性能影响会增加。对于程序的紧密循环和热路径，影响可能很大。

C# 语言的这些增强功能专为性能关键型算法而设计，在这些算法中，使内存分配最小化是实现必需性能的主要因素。你可能会发现，编写代码时不经常使用这些功能。但是，整个 .NET 中都已采用这些增强功能。随着越来越多的 API 使用这些功能，你会发现应用程序的性能得到提升。

请参阅

- [ref 关键字](#)
- [ref 返回结果和局部变量](#)

表达式树

2020/11/2 • [Edit Online](#)

如果你使用过 LINQ，则会有丰富库（其中 `Func` 类型是 API 集的一部分）的经验。（如果尚不熟悉 LINQ，建议阅读 [LINQ 教程](#)，以及本文前面有关 [Lambda 表达式](#)的文章。）表达式树 提供与作为函数的参数的更丰富的交互。

在创建 LINQ 查询时，通常使用 Lambda 表达式编写函数参数。在典型的 LINQ 查询中，这些函数参数会被转换为编译器创建的委托。

当想要进行更丰富的交互时，需要使用表达式树。表达式树将代码表示为可以检查、修改或执行的结构。这些工具让你能够在运行时操作代码。可以编写检查正在运行的算法的代码，或插入新的功能。在更加高级的方案中，你可以修改正在运行的算法，甚至可以将 C# 表达式转换为另一种形式从而可在另一环境中执行。

你可能已经编写过使用表达式树的代码。实体框架的 LINQ API 接受表达式树，以此作为 LINQ 查询表达式模式的参数。这使得 [Entity Framework](#)（实体框架）能够将使用 C# 编写的查询转换为在数据库引擎中执行的 SQL。另一个示例是 [Moq](#)，它是用于 .NET 的热门模拟框架。

本教程的其余部分将探索表达式树是什么、检查支持表达式树的框架类，并介绍如何使用表达式树。你将学习如何阅读表达式树、如何创建表达式树、如何创建修改的表达式树，以及如何执行表达式树所表示的代码。阅读后，便可以开始使用这些结构来创建丰富的自适应算法。

1. 已解释的表达式树

了解表达式树的结构和概念。

2. 框架类型支持表达式树

了解定义和控制表达式树的结构和类。

3. 执行表达式

了解如何将表示为 Lambda 表达式的表达式树转换为委托，并执行结果委托。

4. 解释表达式

了解如何遍历并检查表达式树，以理解表达式树表示的代码。

5. 生成表达式

了解如何构造表达式树的节点并生成表达式树。

6. 转换表达式

了解如何生成表达式树的修改副本，或将表达式树转换为另一种格式。

7. 总结

查看表达式树上的信息。

表达式树说明

2020/3/18 • [Edit Online](#)

上一步 - 概述

表达式树是定义代码的数据结构。它们基于编译器用于分析代码和生成已编译输出的相同结构。读完本教程后，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 `CodeFixes` 是 NuGet 包，用于对代码执行静态分析，并可为开发人员建议可能的修补程序。)两者概念相似，且最终结果是一种数据结构，该结构允许以有意义的方式对源代码进行检查。但是，表达式树基于一组与 Roslyn API 完全不同的类和 API。

让我们来举一个简单的示例。以下是一个代码行：

```
var sum = 1 + 2;
```

如果要将其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。该表达式是 `1 + 2`。这是一个二进制表达式。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。此处的两个节点都是常量表达式：左操作数是值 `1`，右操作数是值 `2`。

直观地看，整个语句是一个树：应从根节点开始，遍历到树中的每个节点，以查看构成语句的代码：

- 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`)

这可能看起来很复杂，但它功能强大。按照相同的过程，可以分解更加复杂的表达式。请思考此表达式：

```
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。我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。可以保持遍历子节点，以查找表达式中的叶节点。父节点将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，你会发现通过快速获得的知识，你可处理许多越来越高级的方案。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，表达式树还可用于在执行代码前轻松编写检查代码的算法。可以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：可以看到任何子表达式的值。可以看到方法和属性名称。可以看到任何常数表达式的值。还可以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` 循环对其进行创建。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。

[下一步 - 框架类型支持表达式树](#)

支持表达式树的框架类型

2020/11/2 • [Edit Online](#)

上一步 - 已解释的表达式树

存在可与表达式树配合使用的 .NET Core framework 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 framework 类的设计方式，而不是逐一查看完整列表。

在语言设计中，表达式是可计算并返回值的代码主体。表达式可能非常简单：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）。

这一切都始于 `System.Linq.Expression`

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，可以将其转换为该类型，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，此代码将打印变量访问表达式的变量的名称。我的做法是，先查看节点类型，再转换为变量访问表达式，然后查看特定表达式类型的属性：

```
Expression<Func<int, int>> addFive = (num) => num + 5;

if (addFive.NodeType == ExpressionType.Lambda)
{
    var lambdaExp = (LambdaExpression)addFive;

    var parameter = lambdaExp.Parameters.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，可以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：

```
// 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。我不会记住所有内容，而是会采用有关使用表达式树的技巧，如下所示：

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，这将非常有用。
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。

[下一步 - 执行表达式树](#)

执行表达式树

2020/3/18 • [Edit Online](#)

上一部分 -- 支持表达式树的框架类型

表达式树是表示一些代码的数据结构。它不是已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，则必须将其转换为可执行的 IL 指令。

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 `ConstantExpression` 意味着什么。这是否意味着任何用处？)

`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

`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。

在大多数情况下，这将在表达式和其对应的委托之间创建简单映射。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。

`LambdaExpression` 类型包含用于将表达式树转换为可执行代码的 `Compile` 和 `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 `MethodBuilder` 对象。请注意，`CompileToMethod` 仅在完整的桌面框架中可用，不能用于 .NET Core。

还可以选择性地提供 `DebugInfoGenerator`，它将接收生成的委托对象的符号调试信息。这让你可以将表达式树转换为委托对象，并拥有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：

```
Expression<Func<int>> add = () => 1 + 2;
var func = add.Compile(); // Create Delegate
var answer = func(); // Invoke Delegate
Console.WriteLine(answer);
```

请注意，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。必须将其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。

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。可以在上面进行查看，其中 `add.Compile()` 返回了一个委托。通过调用 `func()` 调用该委托将执行代码。

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，且在之后编译同一表达式树将创建执行相同代码的委托。)

在此提醒你不要通过避免不必要的编译调用尝试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，以确定如果它们表示相同的算法，是否也会花费很长的时间来执行。你可能会发现，通过避免对 `LambdaExpression.Compile()` 的任何额外调用所节省的计算时间将多于执行代码(该代码确定可导致相同可执行代码的两个不同表达式树)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。

一般情况下，编译器会确保这一点。但是，如果表达式访问实现 `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，此代码工作正常，因为 `int` 不实现 `IDisposable`：

```
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，请考虑实现 `IDisposable` 的此（人为设计的）类：

```
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;
    }
}
```

如果将其用于如下所示的表达式中，则在执行 `Resource.Argument` 属性引用的代码时将出现

`ObjectDisposedException`：

```
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，且在创建可由公共 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、编译或在调用结果委托时，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。这提供了一种机制，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，那么错误也是很容易预知的，并且将在使用表达式树的任何代码的第一个测试中捕获这些错误。

[下一部分 -- 解释表达式](#)

解释表达式

2020/11/2 • [Edit Online](#)

上一部分 -- 执行表达式

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将是派生自 `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：

```
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}");
```

将打印以下内容：

```
This is 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。下面是该代码：

检查一个简单的加法表达式

让我们从本节简介处的加法示例开始。

```
Expression<Func<int>> sum = () => 1 + 2;
```

我没有使用 `var` 来声明此表达式树，因为此操作无法执行，这是由于赋值右侧是隐式类型而导致的。

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。我们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，将需要以递归方式访问大量节点。下面是一个简单的首次实现：

```

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($"{Environment.NewLine}\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($"{Environment.NewLine}\tParameter Type: {left.Type.ToString()}, Name: {left.Name}");
Console.WriteLine($"The right side is a {additionBody.Right.NodeType} expression");
var right= (ParameterExpression)additionBody.Right;
Console.WriteLine($"{Environment.NewLine}\tParameter Type: {right.Type.ToString()}, Name: {right.Name}");

```

此示例打印以下输出：

```

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

```

你会注意到以上代码示例中的大量重复。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：

```

// 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 => this.node.NodeType;
    public static Visitor CreateFromExpression(Expression node)
    {
        switch(node.NodeType)
        {
            case ExpressionType.Constant:
                return new ConstantVisitor((ConstantExpression)node);
            case ExpressionType.Lambda:
                return new LambdaVisitor((LambdaExpression)node);
            case ExpressionType.Parameter:
                return new ParameterVisitor((ParameterExpression)node);
            case ExpressionType.Add:
                return new BinaryVisitor((BinaryExpression)node);
        }
    }
}

```

```

        return new BinaryVisitor((BinaryExpression)node);
    default:
        Console.Error.WriteLine($"Node not processed yet: {node.NodeType}");
        return default(Visitor);
    }
}

// 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($"{prefix}The return type is {node.ReturnType.ToString()}");
        Console.WriteLine($"{prefix}The expression has {node.Parameters.Count} argument(s). They are:");
        // Visit each parameter:
        foreach (var argumentExpression in node.Parameters)
        {
            var argumentVisitor = Visitor.CreateFromExpression(argumentExpression);
            argumentVisitor.Visit(prefix + "\t");
        }
        Console.WriteLine($"{prefix}The expression body is:");
        // Visit the body:
        var bodyVisitor = Visitor.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($"{prefix}This binary expression is a {NodeType} expression");
        var left = Visitor.CreateFromExpression(node.Left);
        Console.WriteLine($"{prefix}The Left argument is:");
        left.Visit(prefix + "\t");
        var right = Visitor.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($"{prefix}!{node is an {NodeType} expression type");
Console.WriteLine($"{prefix}Type: {node.Type.ToString()}, 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}");
    }
}

```

此算法是可以访问任意 `LambdaExpression` 的算法的基础。其中有许多缺口，即我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 case 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。）

在上面所示的加法表达式中运行此访问者时，将获得以下输出：

```

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。

检查具有许多级别的加法表达式

让我们尝试更复杂的示例，但仍限制节点类型仅为加法：

```
Expression<Func<int>> sum = () => 1 + 2 + 3 + 4;
```

在访问者算法上运行此表达式之前，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：

```
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，则将看到此输出，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，我不得不对源表达式树进行一次更改。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：

```
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，则会看到以下输出：

```
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
```

还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：

```
Expression<Func<int, int, int>> sum3 = (a, b) => (1 + a) + (3 + b);
```

下面是访问者的输出：

```

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。

从此示例扩展

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。让我们这样来改进它：

```

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。我只能使用表达式。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，你将了解克服这些限制的技巧。

在此表达式中，将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，便将看到每个可能的节点。这样便万事俱备了。结果类似于：

```
public static Visitor CreateFromExpression(Expression node)
{
    switch(node.NodeType)
    {
        case ExpressionType.Constant:
            return new ConstantVisitor((ConstantExpression)node);
        case ExpressionType.Lambda:
            return new LambdaVisitor((LambdaExpression)node);
        case ExpressionType.Parameter:
            return new ParameterVisitor((ParameterExpression)node);
        case ExpressionType.Add:
        case ExpressionType.Equal:
        case ExpressionType.Multiply:
            return new BinaryVisitor((BinaryExpression)node);
        case ExpressionType.Conditional:
            return new ConditionalVisitor((ConditionalExpression)node);
        case ExpressionType.Call:
            return new MethodCallVisitor((MethodCallExpression)node);
        default:
            Console.Error.WriteLine($"Node not processed yet: {node.NodeType}");
            return default(Visitor);
    }
}
```

ConditionalVisitor 和 MethodCallVisitor 将处理这两个节点：

```

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：

```
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。我略过了很多可能需要的操作，以便专注于访问表达式树中的节点这一核心任务。

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。你仍可以向其添加许多将导致其失败的表达式。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。我编写它是为了让所使用的结构清晰，

以及强调用于访问节点和对此进行分析的技术。生产实现将更加注重性能。

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。

[下一部分 -- 生成表达式](#)

生成表达式树

2021/5/7 • • [Edit Online](#)

上一步 - 解释表达式

到目前为止，你所看到的所有表达式树都是由 C# 编译器创建的。你所要做的是创建一个 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。这不是创建表达式树的唯一方法。很多情况下，可能需要在运行时在内存中生成一个表达式。

由于这些表达式树是不可变的，所以生成表达式树很复杂。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参数。让我们通过几个示例来了解相关技巧。

创建节点

让我们再次从相对简单的内容开始。我们将使用在这些部分中一直使用的加法表达式：

```
Expression<Func<int>> sum = () => 1 + 2;
```

若要构造该表达式树，必须构造叶节点。叶节点是常量，因此可以使用 `Expression.Constant` 方法创建节点：

```
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：

```
var addition = Expression.Add(one, two);
```

一旦获得了加法表达式，就可以创建 lambda 表达式：

```
var lambda = Expression.Lambda(addition);
```

这是一个非常简单的 Lambda 表达式，因为它不包含任何参数。在本节的后续部分，你将了解如何将实参映射到形参并生成更复杂的表达式。

对于此类简单的表达式，可以将所有调用合并到单个语句中：

```
var lambda = Expression.Lambda(
    Expression.Add(
        Expression.Constant(1, typeof(int)),
        Expression.Constant(2, typeof(int))
    )
);
```

生成树

这是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：参数节点和方法调用节点。

生成一个表达式树以创建此表达式：

```
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：

```
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：

```
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。

```
var sqrtMethod = typeof(Math).GetMethod("Sqrt", new[] { typeof(double) });  
var distance = Expression.Call(sqrtMethod, sum);
```

最后，将方法调用放入 lambda 表达式，并确保定义 lambda 表达式的参数：

```
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 `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。

深度生成代码

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：

```
Func<int, int> factorialFunc = (n) =>  
{  
    var res = 1;  
    while (n > 1)  
    {  
        res = res * n;  
        n--;  
    }  
    return res;  
};
```

请注意上面我未生成表达式树，只是生成了委托。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。它很复杂，这是因为没有用于生成 `while` 循环的 API，而是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。

```
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 语句以及我们在日常编码任务中想要避免的其他元素。

在本部分中，我还更新了用于访问此表达式树中所有节点的访客代码，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。有关下载说明，请参阅[示例和教程](#)。

检查 API

表达式树 API 在 .NET Core 中较难导航，但没关系。它们的用途相当复杂：编写在运行时生成代码的代码。它们必须具有复杂的结构，才能在支持 C# 语言中提供的所有控件结构和尽可能减小 API 表面积之间保持平衡。这种平衡意味着许多控件结构不是由其 C# 构造表示，而是由表示基础逻辑的构造表示，这些基础逻辑由编译器从这些较高级别的构造生成。

另外，此时存在一些不能通过使用 `Expression` 类方法直接生成的 C# 表达式。一般来说，这些将是在 C# 5 和 C# 6 中添加的最新运算符和表达式。（例如，无法生成 `async` 表达式，并且无法直接创建新 `?.` 运算符。）

[下一步 - 转换表达式](#)

转换表达式树

2020/11/2 • [Edit Online](#)

上一部分 -- 生成表达式

在此最后一节中，将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。以下是在两个重要方案中将使用的技巧。第一种是了解表达式树表示的算法，以便可以将其转换到另一个环境中。第二种是何时更改已创建的算法。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。

转换即访问

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。

让我们通过访问表达式树，并创建具有一些替换节点的新树，来查看其工作原理。在此示例中，我们将任何常数替换为其十倍大的常数。要么就保持表达式树不变。我们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，而不必阅读常数的值并将其替换为新的常数。

此处，在找到常数节点后，创建新乘法节点（其子节点是原始常数和常数 10）：

```
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;
}
```

通过替换原始节点，将形成一个包含修改的新树。可以通过编译并执行替换的树对此进行验证。

```
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。

此示例演示了表达式树不可变这一点的重要性。请注意，上面创建的新树混合了新创建的节点和现有树中的节点。这是安全的，因为现有树中的节点无法进行修改。这可以极大提高内存效率。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此进行验证。通过对目前见到的访问者进行一些修改来执行此操作。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。

```
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 we 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，但这些概念是非常容易理解的。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，这些子级将计算出为孩子树计算的总和。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，将计算出总和。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：

```
private static int Aggregate(Expression exp)
{
    if (exp.NodeType == ExpressionType.Constant)
    {
        var constantExp = (ConstantExpression)exp;
        Console.Error.WriteLine($"Found Constant: {constantExp.Value}");
        return (int)constantExp.Value;
    }
    else if (exp.NodeType == ExpressionType.Add)
    {
        var addExp = (BinaryExpression)exp;
        Console.Error.WriteLine("Found Addition Expression");
        Console.Error.WriteLine("Computing Left node");
        var leftOperand = Aggregate(addExp.Left);
        Console.Error.WriteLine($"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");
}
```

在同一表达式中运行该版本将生成以下输出：

```
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：

```
Expression<Func<int> sum1 = () => 1 + (2 + (3 + 4));
```

下面是通过检查此表达式得到的输出：

```
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，但树遍历完全不同。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。

了解更多信息

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

他语言的通用库所需的所有工作的完整讨论, 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。

希望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。这样可使创建修改的表达式树所需的内存量最小。

[下一部分 -- 总结](#)

表达式树摘要

2020/3/18 • [Edit Online](#)

[上一篇 - 翻译表达式](#)

在此系列中，你已了解如何使用表达式树创建将代码解释为数据并基于该代码生成新功能的动态程序。

可以检查表达式树以了解算法的目的。不仅可以检查该代码。可以生成表示原始代码的修改版本的新表达式树。

还可以使用表达式树来查看算法，并将该算法翻译成另一种语言或环境。

限制

存在一些不好翻译成表达式树的较新的 C# 语言元素。表达式树不能包含 `await` 表达式或 `async` lambda 表达式。C# 6 发行中添加的许多功能不会完全按照表达式树中所编写的那样显示。较新的功能可能会显示在表达式树中等效、早期的语法中。这可能不像你想象的那样有局限性。实际上，这意味着在引入新语言功能时，解释表达式树的代码将仍可能照常运行。

即使具有这些限制，通过表达式树，仍可创建依赖于解释和修改表示为数据结构的代码的动态算法。它是一种功能强大的工具，作为 .NET 生态系统的一种功能，它可使丰富的库（如实体框架）完成其所执行的操作。

互操作性 (C# 编程指南)

2020/11/2 • [Edit Online](#)

借助互操作性，可以保留和利用对非托管代码的现有投资工作。在公共语言运行时 (CLR) 控制下运行的代码称为 **托管代码**，不在 CLR 控制下运行的代码称为 **非托管代码**。例如，COM、COM+、C++ 组件、ActiveX 组件和 Microsoft Windows API 都是 **非托管代码**。

借助 .NET，可通过平台调用服务、[System.Runtime.InteropServices](#) 命名空间、C++ 互操作性和 COM 互操作性 (COM 互操作) 实现与 **非托管代码** 的互操作性。

本节内容

[互操作性概述](#)

介绍了实现 C# 托管代码和非托管代码的互操作性的方法。

[如何使用 C# 功能访问 Office 互操作对象](#)

介绍了 Visual C# 为了推动 Office 编程而引入的功能。

[如何在 COM 互操作编程中使用索引属性](#)

介绍了如何使用已编入索引的属性来访问包含参数的 COM 属性。

[如何使用平台调用播放 WAV 文件](#)

介绍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。

[演练:Office 编程](#)

展示了如何创建 Excel 工作簿和包含指向此工作簿的链接的 Word 文档。

[COM 类示例](#)

展示了如何将 C# 类公开为 COM 对象。

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。

请参阅

- [Marshal.ReleaseComObject](#)
- [C# 编程指南](#)
- [与非托管代码交互操作](#)
- [演练:Office 编程](#)

使用 XML 注释记录 C# 代码

2021/5/7 • [Edit Online](#)

XML 文档注释是一种特殊注释，添加在任何用户定义的类型或成员的定义上方。其特殊之处在于其可由编译器处理，由此在编译时生成 XML 文档文件。编译器生成的 XML 文件可以与 .NET 程序集一起分发，以便 Visual Studio 和其他 IDE 可使用 IntelliSense 显示关于类型或成员的快速信息。此外，可以通过 DocFX 和 Sandcastle 等工具运行 XML 文件，由此生成 API 引用网站。

编译器会忽略 XML 文档注释，与对待其他注释一样。

可通过执行下列操作之一在编译时生成 XML 文件：

- 如果要使用 .NET Core 从命令行开发应用程序，可以将 `GenerateDocumentationFile` 元素添加到 .csproj 项目文件的 `<PropertyGroup>` 部分。此外，也可以直接使用 `DocumentationFile` 元素指定文档文件的路径。

下面的示例使用与程序集相同的根文件夹名在项目目录中生成 XML 文件：

```
<GenerateDocumentationFile>true</GenerateDocumentationFile>
```

此表达式等效于以下表达式：

```
<DocumentationFile>bin\$(Configuration)\$(TargetFramework)\$(AssemblyName).xml</DocumentationFile>
```

- 如果使用 Visual Studio 开发应用程序，右键单击项目并选择“属性”。在属性对话框中，选择“生成”选项卡，然后选中“XML 文档文件”。还可以更改编译器写入文件的位置。
- 如果是从命令行编译 .NET 应用程序，编译时请添加 `DocumentationFile` 编译器选项。

XML 文档注释使用三个正斜杠 (`///`) 和 XML 格式的注释正文。例如：

```
/// <summary>
/// This class does something.
/// </summary>
public class SomeClass
{}
```

演练

现在来演练针对一个十分基本的数学库进行文档编制，以使新手开发人员能够容易地理解/参与，以及使第三方开发人员能够轻松使用。

下面是简单数学库的代码：

```

/*
The main Math class
Contains all methods for performing basic math functions
*/
public class Math
{
    // Adds two integers and returns the result
    public static int Add(int a, int b)
    {
        // If any parameter is equal to the max value of an integer
        // and the other is greater than zero
        if ((a == int.MaxValue && b > 0) || (b == int.MaxValue && a > 0))
            throw new System.OverflowException();

        return a + b;
    }

    // Adds two doubles and returns the result
    public static double Add(double a, double b)
    {
        if ((a == double.MaxValue && b > 0) || (b == double.MaxValue && a > 0))
            throw new System.OverflowException();

        return a + b;
    }

    // Subtracts an integer from another and returns the result
    public static int Subtract(int a, int b)
    {
        return a - b;
    }

    // Subtracts a double from another and returns the result
    public static double Subtract(double a, double b)
    {
        return a - b;
    }

    // Multiplies two integers and returns the result
    public static int Multiply(int a, int b)
    {
        return a * b;
    }

    // Multiplies two doubles and returns the result
    public static double Multiply(double a, double b)
    {
        return a * b;
    }

    // Divides an integer by another and returns the result
    public static int Divide(int a, int b)
    {
        return a / b;
    }

    // Divides a double by another and returns the result
    public static double Divide(double a, double b)
    {
        return a / b;
    }
}

```

示例库支持 `int` 和 `double` 数据类型的四种主要算术运算(`add`、`subtract`、`multiply` 和 `divide`)。

现在，希望能够从代码中为使用库但无源代码访问权限的第三方开发人员创建 API 引用文档。如前面所述，XML

文档标记可用于实现此目的。现在介绍 C# 编译器支持的标准 XML 标记。

<summary>

<summary> 标记可添加关于类型或成员的信息摘要。这里通过将其添加到 `Math` 类定义和第一个 `Add` 方法来演示其用法。可随意将其应用于代码的其余部分。

```
/*
The main Math class
Contains all methods for performing basic math functions
*/
/// <summary>
/// The main Math class.
/// Contains all methods for performing basic math functions.
/// </summary>
public class Math
{
    // Adds two integers and returns the result
    /// <summary>
    /// Adds two integers and returns the result.
    /// </summary>
    public static int Add(int a, int b)
    {
        // If any parameter is equal to the max value of an integer
        // and the other is greater than zero
        if ((a == int.MaxValue && b > 0) || (b == int.MaxValue && a > 0))
            throw new System.OverflowException();

        return a + b;
    }
}
```

<summary> 标记很重要，建议包含，因为其内容是 IntelliSense 或 API 参考文档中的类型或成员信息的主要来源。

<remarks>

<remarks> 标记可补充关于 <summary> 标记提供的类型或成员的信息。在此示例中，只需将其添加到类。

```
/*
The main Math class
Contains all methods for performing basic math functions
*/
/// <summary>
/// The main Math class.
/// Contains all methods for performing basic math functions.
/// </summary>
/// <remarks>
/// This class can add, subtract, multiply and divide.
/// </remarks>
public class Math
{
```

<returns>

<returns> 标记描述方法声明的返回值。与以前一样，下面的示例展示第一个 `Add` 方法上的 <returns> 标记。可以对其他方法执行相同操作。

```
// Adds two integers and returns the result
/// <summary>
/// Adds two integers and returns the result.
/// </summary>
/// <returns>
/// The sum of two integers.
/// </returns>
public static int Add(int a, int b)
{
    // If any parameter is equal to the max value of an integer
    // and the other is greater than zero
    if ((a == int.MaxValue && b > 0) || (b == int.MaxValue && a > 0))
        throw new System.OverflowException();

    return a + b;
}
```

<value>

<value> 标记类似于 <returns> 标记, 只不过前者用于属性。假设 Math 库有一个名为 PI 的静态属性, 下面是此标记的用法:

```
/*
The main Math class
Contains all methods for performing basic math functions
*/
/// <summary>
/// The main Math class.
/// Contains all methods for performing basic math functions.
/// </summary>
/// <remarks>
/// This class can add, subtract, multiply and divide.
/// These operations can be performed on both integers and doubles
/// </remarks>
public class Math
{
    /// <value>Gets the value of PI.</value>
    public static double PI { get; }
}
```

<example>

使用 <example> 标记可在 XML 文档中包含一个示例。此操作包括使用子 <code> 标记。

```
// Adds two integers and returns the result
/// <summary>
/// Adds two integers and returns the result.
/// </summary>
/// <returns>
/// The sum of two integers.
/// </returns>
/// <example>
/// <code>
/// int c = Math.Add(4, 5);
/// if (c > 10)
/// {
///     Console.WriteLine(c);
/// }
/// </code>
/// </example>
public static int Add(int a, int b)
{
    // If any parameter is equal to the max value of an integer
    // and the other is greater than zero
    if ((a == int.MaxValue && b > 0) || (b == int.MaxValue && a > 0))
        throw new System.OverflowException();

    return a + b;
}
```

`code` 标记保留较长示例的换行符和缩进。

<para>

<para> 标记可用于设置其父标记内的内容的格式。<para> 通常在标记内使用，如 <remarks> 或 <returns>，作用是将文本分成段落。可以为类定义设置 <remarks> 标记的内容的格式。

```
/*
The main Math class
Contains all methods for performing basic math functions
*/
/// <summary>
/// The main Math class.
/// Contains all methods for performing basic math functions.
/// </summary>
/// <remarks>
/// <para>This class can add, subtract, multiply and divide.</para>
/// <para>These operations can be performed on both integers and doubles.</para>
/// </remarks>
public class Math
{}
```

<c>

还是关于格式设置，使用 <c> 标记可将部分文本标记为代码。与 <code> 标记类似，但是内联的。想要显示快速代码示例并将其作为标记内容一部分时，这很有帮助。现在来更新 Math 类的文档。

```
/*
    The main Math class
    Contains all methods for performing basic math functions
*/
/// <summary>
/// The main <c>Math</c> class.
/// Contains all methods for performing basic math functions.
/// </summary>
public class Math
{
```

<exception>

通过使用 `<exception>` 标记，可以让开发人员了解到，方法有可能引发特定异常。查看 `Math` 库，可以看到，如果满足特定条件，两个 `Add` 方法都会引发异常。如果 `b` 参数为零，则 `Divide` 方法也会引发异常，尽管不是很常见。现在将异常文档添加到此方法。

```
/*
    The main Math class
    Contains all methods for performing basic math functions
*/
/// <summary>
/// The main <c>Math</c> class.
/// Contains all methods for performing basic math functions.
/// </summary>
public class Math
{
    /// <summary>
    /// Adds two integers and returns the result.
    /// </summary>
    /// <returns>
    /// The sum of two integers.
    /// </returns>
    /// <example>
    /// <code>
    /// int c = Math.Add(4, 5);
    /// if (c > 10)
    /// {
    ///     Console.WriteLine(c);
    /// }
    /// </code>
    /// </example>
    /// <exception cref="System.OverflowException">Thrown when one parameter is max
    /// and the other is greater than 0.</exception>
    public static int Add(int a, int b)
    {
        if ((a == int.MaxValue && b > 0) || (b == int.MaxValue && a > 0))
            throw new System.OverflowException();

        return a + b;
    }

    /// <summary>
    /// Adds two doubles and returns the result.
    /// </summary>
    /// <returns>
    /// The sum of two doubles.
    /// </returns>
    /// <exception cref="System.OverflowException">Thrown when one parameter is max
    /// and the other is greater than zero.</exception>
    public static double Add(double a, double b)
    {
        if ((a == double.MaxValue && b > 0) || (b == double.MaxValue && a > 0))
```

```
        throw new System.OverflowException();

    return a + b;
}

/// <summary>
/// Divides an integer by another and returns the result.
/// </summary>
/// <returns>
/// The division of two integers.
/// </returns>
/// <exception cref="System.DivideByZeroException">Thrown when a division by zero occurs.</exception>
public static int Divide(int a, int b)
{
    return a / b;
}

/// <summary>
/// Divides a double by another and returns the result.
/// </summary>
/// <returns>
/// The division of two doubles.
/// </returns>
/// <exception cref="System.DivideByZeroException">Thrown when a division by zero occurs.</exception>
public static double Divide(double a, double b)
{
    return a / b;
}
}
```

cref 属性表示可从当前编译环境中实现的异常引用。其类型可为项目中或引用的程序集中定义的任何类型。如果无法解析编译器的值，则该编译器将发出一条警告。

<see>

使用 **<see>** 标记，可以创建可单击的链接，指向另一个代码元素的文档页面。在下一个示例中，将创建两个 **Add** 方法之间的可单击的链接。

```

/*
    The main Math class
    Contains all methods for performing basic math functions
*/
/// <summary>
/// The main <c>Math</c> class.
/// Contains all methods for performing basic math functions.
/// </summary>
public class Math
{
    /// <summary>
    /// Adds two integers and returns the result.
    /// </summary>
    /// <returns>
    /// The sum of two integers.
    /// </returns>
    /// <example>
    /// <code>
    /// int c = Math.Add(4, 5);
    /// if (c > 10)
    /// {
    ///     Console.WriteLine(c);
    /// }
    /// </code>
    /// </example>
    /// <exception cref="System.OverflowException">Thrown when one parameter is max
    /// and the other is greater than 0.</exception>
    /// See <see cref="Math.Add(double, double)"> to add doubles.
    public static int Add(int a, int b)
    {
        if ((a == int.MaxValue && b > 0) || (b == int.MaxValue && a > 0))
            throw new System.OverflowException();

        return a + b;
    }

    /// <summary>
    /// Adds two doubles and returns the result.
    /// </summary>
    /// <returns>
    /// The sum of two doubles.
    /// </returns>
    /// <exception cref="System.OverflowException">Thrown when one parameter is max
    /// and the other is greater than zero.</exception>
    /// See <see cref="Math.Add(int, int)"> to add integers.
    public static double Add(double a, double b)
    {
        if ((a == double.MaxValue && b > 0) || (b == double.MaxValue && a > 0))
            throw new System.OverflowException();

        return a + b;
    }
}

```

`cref` 是表示可从当前编译环境引用的类型或其成员的必需属性。其类型可为项目中或引用的程序集中定义的任何类型。

<seealso>

可以通过使用 `<see>` 标记的方式使用 `<seealso>` 标记。唯一的区别是其内容通常位于“另请参见”部分。以下将在整数 `Add` 方法上添加 `seealso` 标记，从而在接受整数参数的类中引用其他方法：

```

/*
    The main Math class
    Contains all methods for performing basic math functions
*/
/// <summary>
/// The main <c>Math</c> class.
/// Contains all methods for performing basic math functions.
/// </summary>
public class Math
{
    /// <summary>
    /// Adds two integers and returns the result.
    /// </summary>
    /// <returns>
    /// The sum of two integers.
    /// </returns>
    /// <example>
    /// <code>
    /// int c = Math.Add(4, 5);
    /// if (c > 10)
    /// {
    ///     Console.WriteLine(c);
    /// }
    /// </code>
    /// </example>
    /// <exception cref="System.OverflowException">Thrown when one parameter is max
    /// and the other is greater than 0.</exception>
    /// See <see cref="Math.Add(double, double)"> to add doubles.
    /// <seealso cref="Math.Subtract(int, int)">
    /// <seealso cref="Math.Multiply(int, int)">
    /// <seealso cref="Math.Divide(int, int)">
    public static int Add(int a, int b)
    {
        if ((a == int.MaxValue && b > 0) || (b == int.MaxValue && a > 0))
            throw new System.OverflowException();

        return a + b;
    }
}

```

cref 属性表示可从当前编译环境进行的对类型或其成员的引用。其类型可为项目中或引用的程序集中定义的任何类型。

<param>

使用 **<param>** 标记来描述方法的参数。下面是关于双 **Add** 方法的示例：标记所描述的参数在 **必需的** **name** 属性中指定。

```

/*
    The main Math class
    Contains all methods for performing basic math functions
*/
/// <summary>
/// The main <c>Math</c> class.
/// Contains all methods for performing basic math functions.
/// </summary>
public class Math
{
    /// <summary>
    /// Adds two doubles and returns the result.
    /// </summary>
    /// <returns>
    /// The sum of two doubles.
    /// </returns>
    /// <exception cref="System.OverflowException">Thrown when one parameter is max
    /// and the other is greater than zero.</exception>
    /// See <see cref="Math.Add(int, int)"> to add integers.
    /// <param name="a">A double precision number.</param>
    /// <param name="b">A double precision number.</param>
    public static double Add(double a, double b)
    {
        if ((a == double.MaxValue && b > 0) || (b == double.MaxValue && a > 0))
            throw new System.OverflowException();

        return a + b;
    }
}

```

<typeparam>

<typeparam> 标记的用法与 <param> 标记一样，但前者由泛型类型或方法声明用来描述泛型参数。将快速泛型方法添加到 Math 类，以检查某个数量是否大于另一个数量。

```

/*
    The main Math class
    Contains all methods for performing basic math functions
*/
/// <summary>
/// The main <c>Math</c> class.
/// Contains all methods for performing basic math functions.
/// </summary>
public class Math
{
    /// <summary>
    /// Checks if an IComparable is greater than another.
    /// </summary>
    /// <typeparam name="T">A type that inherits from the IComparable interface.</typeparam>
    public static bool GreaterThan<T>(T a, T b) where T : IComparable
    {
        return a.CompareTo(b) > 0;
    }
}

```

<paramref>

有时可能正在通过一个 <summary> 标记描述一个方法的作用，并且想要引用一个参数。这时 <paramref> 标记就很适合用来实现这一目的。现在来更新双基 Add 方法的摘要。与 <param> 标记一样，参数名称在必需的 name 属性中指定。

```

/*
    The main Math class
    Contains all methods for performing basic math functions
*/
/// <summary>
/// The main <c>Math</c> class.
/// Contains all methods for performing basic math functions.
/// </summary>
public class Math
{
    /// <summary>
    /// Adds two doubles <paramref name="a"/> and <paramref name="b"/> and returns the result.
    /// </summary>
    /// <returns>
    /// The sum of two doubles.
    /// </returns>
    /// <exception cref="System.OverflowException">Thrown when one parameter is max
    /// and the other is greater than zero.</exception>
    /// See <see cref="Math.Add(int, int)"> to add integers.
    /// <param name="a">A double precision number.</param>
    /// <param name="b">A double precision number.</param>
    public static double Add(double a, double b)
    {
        if ((a == double.MaxValue && b > 0) || (b == double.MaxValue && a > 0))
            throw new System.OverflowException();

        return a + b;
    }
}

```

<typeparamref>

<typeparamref> 标记的用法与 <paramref> 标记一样，但前者由泛型类型或方法声明用来描述泛型参数。可以使用之前创建的那个泛型方法。

```

/*
    The main Math class
    Contains all methods for performing basic math functions
*/
/// <summary>
/// The main <c>Math</c> class.
/// Contains all methods for performing basic math functions.
/// </summary>
public class Math
{
    /// <summary>
    /// Checks if an IComparable <typeparamref name="T"/> is greater than another.
    /// </summary>
    /// <typeparam name="T">A type that inherits from the IComparable interface.</typeparam>
    public static bool GreaterThan<T>(T a, T b) where T : IComparable
    {
        return a.CompareTo(b) > 0;
    }
}

```

<list>

使用 <list> 标记可将文档信息格式化为有序列表、无序列表或表格。制作 `Math` 库支持的所有数学操作的无序列表。

```

/*
   The main Math class
   Contains all methods for performing basic math functions
*/
/// <summary>
/// The main <c>Math</c> class.
/// Contains all methods for performing basic math functions.
/// <list type="bullet">
/// <item>
/// <term>Add</term>
/// <description>Addition Operation</description>
/// </item>
/// <item>
/// <term>Subtract</term>
/// <description>Subtraction Operation</description>
/// </item>
/// <item>
/// <term>Multiply</term>
/// <description>Multiplication Operation</description>
/// </item>
/// <item>
/// <term>Divide</term>
/// <description>Division Operation</description>
/// </item>
/// </list>
/// </summary>
public class Math
{
}

```

可以通过将 `type` 属性分别改为 `number` 或 `table` 来制作有序列表或表格。

<inheritdoc>

可以使用 `<inheritdoc>` 标记继承基类、接口和类似方法中的 XML 注释。这样不必复制和粘贴重复的 XML 注释，并自动保持 XML 注释同步。

```

/*
   The IMath interface
   The main Math class
   Contains all methods for performing basic math functions
*/
/// <summary>
/// This is the IMath interface.
/// </summary>
public interface IMath
{
}

/// <inheritdoc/>
public class Math : IMath
{
}

```

将其放在一起

如果已按照本教程的操作方法将标记应用于代码中的所需位置，则代码现应如下所示：

```

/*
   The main Math class
   Contains all methods for performing basic math functions
*/
/// <summary>

```

```

/// The main <c>Math</c> class.
/// Contains all methods for performing basic math functions.
/// <list type="bullet">
/// <item>
/// <term>Add</term>
/// <description>Addition Operation</description>
/// </item>
/// <item>
/// <term>Subtract</term>
/// <description>Subtraction Operation</description>
/// </item>
/// <item>
/// <term>Multiply</term>
/// <description>Multiplication Operation</description>
/// </item>
/// <item>
/// <term>Divide</term>
/// <description>Division Operation</description>
/// </item>
/// </list>
/// </summary>
/// <remarks>
/// <para>This class can add, subtract, multiply and divide.</para>
/// <para>These operations can be performed on both integers and doubles.</para>
/// </remarks>
public class Math
{
    // Adds two integers and returns the result
    /// <summary>
    /// Adds two integers <paramref name="a"/> and <paramref name="b"/> and returns the result.
    /// </summary>
    /// <returns>
    /// The sum of two integers.
    /// </returns>
    /// <example>
    /// <code>
    /// int c = Math.Add(4, 5);
    /// if (c > 10)
    /// {
    ///     Console.WriteLine(c);
    /// }
    /// </code>
    /// </example>
    /// <exception cref="System.OverflowException">Thrown when one parameter is max
    /// and the other is greater than 0.</exception>
    /// See <see cref="Math.Add(double, double)"> to add doubles.
    /// <seealso cref="Math.Subtract(int, int)">
    /// <seealso cref="Math.Multiply(int, int)">
    /// <seealso cref="Math.Divide(int, int)">
    /// <param name="a">An integer.</param>
    /// <param name="b">An integer.</param>
    public static int Add(int a, int b)
    {
        // If any parameter is equal to the max value of an integer
        // and the other is greater than zero
        if ((a == int.MaxValue && b > 0) || (b == int.MaxValue && a > 0))
            throw new System.OverflowException();

        return a + b;
    }

    // Adds two doubles and returns the result
    /// <summary>
    /// Adds two doubles <paramref name="a"/> and <paramref name="b"/> and returns the result.
    /// </summary>
    /// <returns>
    /// The sum of two doubles.
    /// </returns>
    /// <example>

```

```
/// <code>
/// double c = Math.Add(4.5, 5.4);
/// if (c > 10)
/// {
///     Console.WriteLine(c);
/// }
/// </code>
/// </example>
/// <exception cref="System.OverflowException">Thrown when one parameter is max
/// and the other is greater than 0.</exception>
/// See <see cref="Math.Add(int, int)" /> to add integers.
/// <seealso cref="Math.Subtract(double, double)" />
/// <seealso cref="Math.Multiply(double, double)" />
/// <seealso cref="Math.Divide(double, double)" />
/// <param name="a">A double precision number.</param>
/// <param name="b">A double precision number.</param>
public static double Add(double a, double b)
{
    // If any parameter is equal to the max value of an integer
    // and the other is greater than zero
    if ((a == double.MaxValue && b > 0) || (b == double.MaxValue && a > 0))
        throw new System.OverflowException();

    return a + b;
}

// Subtracts an integer from another and returns the result
/// <summary>
/// Subtracts <paramref name="b"/> from <paramref name="a"/> and returns the result.
/// </summary>
/// <returns>
/// The difference between two integers.
/// </returns>
/// <example>
/// <code>
/// int c = Math.Subtract(4, 5);
/// if (c > 1)
/// {
///     Console.WriteLine(c);
/// }
/// </code>
/// </example>
/// See <see cref="Math.Subtract(double, double)" /> to subtract doubles.
/// <seealso cref="Math.Add(int, int)" />
/// <seealso cref="Math.Multiply(int, int)" />
/// <seealso cref="Math.Divide(int, int)" />
/// <param name="a">An integer.</param>
/// <param name="b">An integer.</param>
public static int Subtract(int a, int b)
{
    return a - b;
}

// Subtracts a double from another and returns the result
/// <summary>
/// Subtracts a double <paramref name="b"/> from another double <paramref name="a"/> and returns the
/// result.
/// </summary>
/// <returns>
/// The difference between two doubles.
/// </returns>
/// <example>
/// <code>
/// double c = Math.Subtract(4.5, 5.4);
/// if (c > 1)
/// {
///     Console.WriteLine(c);
/// }
/// </code>
```

```
/// </example>
/// See <see cref="Math.Subtract(int, int)"> to subtract integers.
/// <seealso cref="Math.Add(double, double)">
/// <seealso cref="Math.Multiply(double, double)">
/// <seealso cref="Math.Divide(double, double)">
/// <param name="a">A double precision number.</param>
/// <param name="b">A double precision number.</param>
public static double Subtract(double a, double b)
{
    return a - b;
}

// Multiplies two integers and returns the result
/// <summary>
/// Multiplies two integers <paramref name="a"/> and <paramref name="b"/> and returns the result.
/// </summary>
/// <returns>
/// The product of two integers.
/// </returns>
/// <example>
/// <code>
/// int c = Math.Multiply(4, 5);
/// if (c > 100)
/// {
///     Console.WriteLine(c);
/// }
/// </code>
/// </example>
/// See <see cref="Math.Multiply(double, double)"> to multiply doubles.
/// <seealso cref="Math.Add(int, int)">
/// <seealso cref="Math.Subtract(int, int)">
/// <seealso cref="Math.Divide(int, int)">
/// <param name="a">An integer.</param>
/// <param name="b">An integer.</param>
public static int Multiply(int a, int b)
{
    return a * b;
}

// Multiplies two doubles and returns the result
/// <summary>
/// Multiplies two doubles <paramref name="a"/> and <paramref name="b"/> and returns the result.
/// </summary>
/// <returns>
/// The product of two doubles.
/// </returns>
/// <example>
/// <code>
/// double c = Math.Multiply(4.5, 5.4);
/// if (c > 100.0)
/// {
///     Console.WriteLine(c);
/// }
/// </code>
/// </example>
/// See <see cref="Math.Multiply(int, int)"> to multiply integers.
/// <seealso cref="Math.Add(double, double)">
/// <seealso cref="Math.Subtract(double, double)">
/// <seealso cref="Math.Divide(double, double)">
/// <param name="a">A double precision number.</param>
/// <param name="b">A double precision number.</param>
public static double Multiply(double a, double b)
{
    return a * b;
}

// Divides an integer by another and returns the result
/// <summary>
/// Divides an integer <paramref name="a"/> by another integer <paramref name="b"/> and returns the

```

```

    /// Divides an integer <paramref name="a"/> by another integer <paramref name="b"/> and returns the
result.
    /// </summary>
    /// <returns>
    /// The quotient of two integers.
    /// </returns>
    /// <example>
    /// <code>
    /// int c = Math.Divide(4, 5);
    /// if (c > 1)
    /// {
    ///     Console.WriteLine(c);
    /// }
    /// </code>
    /// </example>
    /// <exception cref="System.DivideByZeroException">Thrown when <paramref name="b"/> is equal to 0.
</exception>
    /// See <see cref="Math.Divide(double, double)"> to divide doubles.
    /// <seealso cref="Math.Add(int, int)">
    /// <seealso cref="Math.Subtract(int, int)">
    /// <seealso cref="Math.Multiply(int, int)">
    /// <param name="a">An integer dividend.</param>
    /// <param name="b">An integer divisor.</param>
public static int Divide(int a, int b)
{
    return a / b;
}

// Divides a double by another and returns the result
/// <summary>
/// Divides a double <paramref name="a"/> by another double <paramref name="b"/> and returns the result.
/// </summary>
/// <returns>
/// The quotient of two doubles.
/// </returns>
/// <example>
/// <code>
/// double c = Math.Divide(4.5, 5.4);
/// if (c > 1.0)
/// {
///     Console.WriteLine(c);
/// }
/// </code>
/// </example>
/// <exception cref="System.DivideByZeroException">Thrown when <paramref name="b"/> is equal to 0.
</exception>
    /// See <see cref="Math.Divide(int, int)"> to divide integers.
    /// <seealso cref="Math.Add(double, double)">
    /// <seealso cref="Math.Subtract(double, double)">
    /// <seealso cref="Math.Multiply(double, double)">
    /// <param name="a">A double precision dividend.</param>
    /// <param name="b">A double precision divisor.</param>
public static double Divide(double a, double b)
{
    return a / b;
}
}

```

可从代码中生成包括可单击的交叉引用的详细文档网站。但将面临另一问题：代码变得难以阅读。需要筛查的信息浩如烟海，这对任何要参与此代码编写的开发人员都是噩梦。幸好有一个 XML 标记，可帮助解决这个问题：

<include>

使用 `<include>` 标记，可以引用单独的 XML 文件中的注释（该文件描述源代码中的类型和成员），而不是将文档注释直接放入源代码文件中。

现在，要将所有 XML 标记移到名为 `docs.xml` 的单独的 XML 文件中。可随时重新命名该文件。

```
<docs>
  <members name="math">
    <Math>
      <summary>
        The main <c>Math</c> class.
        Contains all methods for performing basic math functions.
      </summary>
      <remarks>
        <para>This class can add, subtract, multiply and divide.</para>
        <para>These operations can be performed on both integers and doubles.</para>
      </remarks>
    </Math>
    <AddInt>
      <summary>
        Adds two integers <paramref name="a"/> and <paramref name="b"/> and returns the result.
      </summary>
      <returns>
        The sum of two integers.
      </returns>
      <example>
        <code>
          int c = Math.Add(4, 5);
          if (c > 10)
          {
            Console.WriteLine(c);
          }
        </code>
      </example>
      <exception cref="System.OverflowException">Thrown when one parameter is max
      and the other is greater than 0.</exception>
      See <see cref="Math.Add(double, double)"> to add doubles.
      <seealso cref="Math.Subtract(int, int)">
      <seealso cref="Math.Multiply(int, int)">
      <seealso cref="Math.Divide(int, int)">
      <param name="a">An integer.</param>
      <param name="b">An integer.</param>
    </AddInt>
    <AddDouble>
      <summary>
        Adds two doubles <paramref name="a"/> and <paramref name="b"/> and returns the result.
      </summary>
      <returns>
        The sum of two doubles.
      </returns>
      <example>
        <code>
          double c = Math.Add(4.5, 5.4);
          if (c > 10)
          {
            Console.WriteLine(c);
          }
        </code>
      </example>
      <exception cref="System.OverflowException">Thrown when one parameter is max
      and the other is greater than 0.</exception>
      See <see cref="Math.Add(int, int)"> to add integers.
      <seealso cref="Math.Subtract(double, double)">
      <seealso cref="Math.Multiply(double, double)">
      <seealso cref="Math.Divide(double, double)">
      <param name="a">A double precision number.</param>
      <param name="b">A double precision number.</param>
    </AddDouble>
    <SubtractInt>
      <summary>
        Subtracts <paramref name="b"/> from <paramref name="a"/> and returns the result.
      </summary>
    </SubtractInt>
  </members>
</docs>
```

```
., ....., .  
<returns>  
The difference between two integers.  
</returns>  
<example>  
<code>  
int c = Math.Subtract(4, 5);  
if (c > 1)  
{  
    Console.WriteLine(c);  
}  
</code>  
</example>  
See <see cref="Math.Subtract(double, double)"/> to subtract doubles.  
<seealso cref="Math.Add(int, int)"/>  
<seealso cref="Math.Multiply(int, int)"/>  
<seealso cref="Math.Divide(int, int)"/>  
<param name="a">An integer.</param>  
<param name="b">An integer.</param>  
</SubtractInt>  
<SubtractDouble>  
<summary>  
Subtracts a double <paramref name="b"/> from another double <paramref name="a"/> and returns the  
result.  
</summary>  
<returns>  
The difference between two doubles.  
</returns>  
<example>  
<code>  
double c = Math.Subtract(4.5, 5.4);  
if (c > 1)  
{  
    Console.WriteLine(c);  
}  
</code>  
</example>  
See <see cref="Math.Subtract(int, int)"/> to subtract integers.  
<seealso cref="Math.Add(double, double)"/>  
<seealso cref="Math.Multiply(double, double)"/>  
<seealso cref="Math.Divide(double, double)"/>  
<param name="a">A double precision number.</param>  
<param name="b">A double precision number.</param>  
</SubtractDouble>  
<MultiplyInt>  
<summary>  
Multiplies two integers <paramref name="a"/> and <paramref name="b"/> and returns the result.  
</summary>  
<returns>  
The product of two integers.  
</returns>  
<example>  
<code>  
int c = Math.Multiply(4, 5);  
if (c > 100)  
{  
    Console.WriteLine(c);  
}  
</code>  
</example>  
See <see cref="Math.Multiply(double, double)"/> to multiply doubles.  
<seealso cref="Math.Add(int, int)"/>  
<seealso cref="Math.Subtract(int, int)"/>  
<seealso cref="Math.Divide(int, int)"/>  
<param name="a">An integer.</param>  
<param name="b">An integer.</param>  
</MultiplyInt>  
<MultiplyDouble>  
<summary>  
Multiplies two doubles <paramref name="a"/> and <paramref name="b"/> and returns the result.  
</summary>
```

Multiplicates two doubles `\param{double} name="a"` and `\param{double} name="b"` and returns the result.

`</summary>`

`<returns>`

The product of two doubles.

`</returns>`

`<example>`

`<code>`

```
double c = Math.Multiply(4.5, 5.4);
if (c > 100.0)
{
    Console.WriteLine(c);
}
```

`</code>`

`</example>`

See `<see cref="Math.Multiply(int, int)">` to multiply integers.

`<seealso cref="Math.Add(double, double)">`

`<seealso cref="Math.Subtract(double, double)">`

`<seealso cref="Math.Divide(double, double)">`

`<param name="a">`A double precision number.`</param>`

`<param name="b">`A double precision number.`</param>`

`</MultiplyDouble>`

`<DivideInt>`

`<summary>`

Divides an integer `<paramref name="a">` by another integer `<paramref name="b">` and returns the result.

`</summary>`

`<returns>`

The quotient of two integers.

`</returns>`

`<example>`

`<code>`

```
int c = Math.Divide(4, 5);
if (c > 1)
{
    Console.WriteLine(c);
}
```

`</code>`

`</example>`

`<exception cref="System.DivideByZeroException">`Thrown when `<paramref name="b">` is equal to 0.

`</exception>`

See `<see cref="Math.Divide(double, double)">` to divide doubles.

`<seealso cref="Math.Add(int, int)">`

`<seealso cref="Math.Subtract(int, int)">`

`<seealso cref="Math.Multiply(int, int)">`

`<param name="a">`An integer dividend.`</param>`

`<param name="b">`An integer divisor.`</param>`

`</DivideInt>`

`<DivideDouble>`

`<summary>`

Divides a double `<paramref name="a">` by another double `<paramref name="b">` and returns the result.

`</summary>`

`<returns>`

The quotient of two doubles.

`</returns>`

`<example>`

`<code>`

```
double c = Math.Divide(4.5, 5.4);
if (c > 1.0)
{
    Console.WriteLine(c);
}
```

`</code>`

`</example>`

`<exception cref="System.DivideByZeroException">`Thrown when `<paramref name="b">` is equal to 0.

`</exception>`

See `<see cref="Math.Divide(int, int)">` to divide integers.

`<seealso cref="Math.Add(double, double)">`

`<seealso cref="Math.Subtract(double, double)">`

```

        <seealso cref="Math.Multiply(double, double)"/>
        <param name="a">A double precision dividend.</param>
        <param name="b">A double precision divisor.</param>
    </DivideDouble>
</members>
</docs>

```

在上面的 XML 中，每个成员的文档注释将直接显示在按其作用命名的标记中。现在一个单独的文件中已具有 XML 注释，接下来来看看如何通过使用 `<include>` 标记使代码更易于阅读：

```

/*
The main Math class
Contains all methods for performing basic math functions
*/
/// <include file='docs.xml' path='docs/members[@name="math"]/Math/*'>
public class Math
{
    // Adds two integers and returns the result
    /// <include file='docs.xml' path='docs/members[@name="math"]/AddInt/*'>
    public static int Add(int a, int b)
    {
        // If any parameter is equal to the max value of an integer
        // and the other is greater than zero
        if ((a == int.MaxValue && b > 0) || (b == int.MaxValue && a > 0))
            throw new System.OverflowException();

        return a + b;
    }

    // Adds two doubles and returns the result
    /// <include file='docs.xml' path='docs/members[@name="math"]/AddDouble/*'>
    public static double Add(double a, double b)
    {
        // If any parameter is equal to the max value of an integer
        // and the other is greater than zero
        if ((a == double.MaxValue && b > 0) || (b == double.MaxValue && a > 0))
            throw new System.OverflowException();

        return a + b;
    }

    // Subtracts an integer from another and returns the result
    /// <include file='docs.xml' path='docs/members[@name="math"]/SubtractInt/*'>
    public static int Subtract(int a, int b)
    {
        return a - b;
    }

    // Subtracts a double from another and returns the result
    /// <include file='docs.xml' path='docs/members[@name="math"]/SubtractDouble/*'>
    public static double Subtract(double a, double b)
    {
        return a - b;
    }

    // Multiplies two integers and returns the result
    /// <include file='docs.xml' path='docs/members[@name="math"]/MultiplyInt/*'>
    public static int Multiply(int a, int b)
    {
        return a * b;
    }

    // Multiplies two doubles and returns the result
    /// <include file='docs.xml' path='docs/members[@name="math"]/MultiplyDouble/*'>
    public static double Multiply(double a, double b)
    {
        return a * b;
    }
}

```

```
}

// Divides an integer by another and returns the result
/// <include file='docs.xml' path='docs/members[@name="math"]/DivideInt/*'>
public static int Divide(int a, int b)
{
    return a / b;
}

// Divides a double by another and returns the result
/// <include file='docs.xml' path='docs/members[@name="math"]/DivideDouble/*'>
public static double Divide(double a, double b)
{
    return a / b;
}
}
```

现在好了，代码又变得可读了，并且未丢失任何文档信息。

`file` 属性表示包含文档的 XML 文件的名称。

`path` 属性表示一个 [XPath](#) 查询，该查询的对象为指定的 `file` 中的 `tag name`。

`name` 属性表示位于注释前的标记中的名称说明符。

可用于替换 `name` 的 `id` 属性表示位于注释前的标记的 ID。

用户定义的标记

上述所有标记均表示由 C# 编译器识别的标记。但用户可随意定义自己的标记。Sandcastle 等工具支持其他标记，例如 `<event>` 和 `<note>`，甚至支持[编制命名空间文档](#)。自定义或内部文档生成工具也可与标准标记配合使用，并支持 HTML 到 PDF 等多种输出格式。

建议

出于多种原因，建议编制代码文档。接下来将介绍一些最佳做法、常规使用方案，以及在 C# 代码中使用 XML 文档标记时的需知内容。

- 为保持一致性，应编制所有公共可见类型及其成员的文档。如果必须这么做，请完整全面地完成这一操作。
- 还可使用 XML 注释编制私有成员的文档。但这会公开库的内部(很可能是机密的)运作情况。
- 但至少类型及其成员应具有 `<summary>` 标记，因为其内容是 IntelliSense 所需要的。
- 应使用句号结尾的完整句子编写文档文本。
- 完全支持部分类，并且该类型的文档信息将串联为单个条目。
- 编译器验证 `<exception>`、`<include>`、`<param>`、`<see>`、`<seealso>` 和 `<typeparam>` 标记的语法。
- 编译器验证某些参数，这些参数包含文件路径和对代码其余部分的引用。

另请参阅

- [XML 文档注释\(C# 编程指南\)](#)
- [建议的文档注释标记\(C# 编程指南\)](#)

C# 中的版本控制

2020/4/27 • [Edit Online](#)

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。

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，如预发布版本等。

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。

NOTE

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。

```
<dependentAssembly>
  <assemblyIdentity name="ReferencedLibrary" publicKeyToken="32ab4ba45e0a69a1" culture="en-us" />
  <bindingRedirect oldVersion="1.0.0" newVersion="1.0.1" />
</dependentAssembly>
```

NOTE

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：

```
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

```
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。

```
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

```
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#)

2021/5/7 • [Edit Online](#)

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。](#)
- 即使是 C# 程序员，也可能需要[使用 Visual Basic My 命名空间](#)。
- [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、记录或结构来实现程序。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。
- 在查询中使用 Lambda 表达式。
- 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。

如何在 C# 中使用 String.Split 分隔字符串

2021/3/5 • [Edit Online](#)

`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。

NOTE

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。

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。

```
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。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。

```
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。

```
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：

```
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)。

```
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);
}
```

请参阅

- [从字符串中提取元素](#)
- [C# 编程指南](#)
- [字符串](#)
- [.NET 正则表达式](#)

如何连接多个字符串 (C# 指南)

2021/5/8 • [Edit Online](#)

串联是将一个字符串追加到另一字符串末尾的过程。可使用 `+` 运算符连接字符串。对于字符串文本和字符串常量，会在编译时进行串联，运行时不串联。对于字符串变量，仅在运行时串联。

NOTE

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。

字符串文本

以下示例将长字符串字面量拆分为较短的字符串，从而提高源代码的可读性。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。在编译时将这些部分连接成一个字符串。无论涉及到多少个字符串，均不产生运行时性能开销。

```
// 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，可使用 `+` 或 `+=` 运算符、[字符串内插](#) 或

[String.Format](#)、[String.Concat](#)、[String.Join](#)、[StringBuilder.Append](#) 方法。`+` 运算符易于使用，有利于产生直观代码。即使在一个语句中使用多个 `+` 运算符，字符串内容也仅会被复制一次。以下代码演示使用 `+` 和 `+=` 运算符串联字符串的示例：

```
string userName = "<Type your name here>";  
string dateString = DateTime.Today.ToString("MM dd, yyyy");  
  
// 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：

```
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);
```

NOTE

在字符串串联操作中，C# 编译器将 null 字符串视为空字符串进行处理。

String.Format

另一个字符串连接方法为 [String.Format](#)。此方法非常适合从少量组件字符串生成字符串的情况。

StringBuilder

在其他情况下，可能需要将字符串合并在循环中，此时不知道要合并的源字符串的数量，而且源字符串的实际数量可能很大。[StringBuilder](#) 类专门用于此类方案。以下代码使用 [StringBuilder](#) 类的 [Append](#) 方法串联字符串。

```
// 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

还可使用 [String.Concat](#) 方法联接集合中的字符串。如果源字符串应使用分隔符分隔，请使用 [String.Join](#) 方法。

以下代码使用这两种方法合并单词数组：

```
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.Aggreagte

最后，可以使用 [LINQ](#) 和 [Enumerable.Aggreagte](#) 方法联接集合中的字符串。此方法利用 lambda 表达式合并源字符串。lambda 表达式用于将所有字符串添加到现有累积。下面的示例通过在数组中的每两个单词之间添加一个空格来合并单词数组：

```
string[] words = { "The", "quick", "brown", "fox", "jumps", "over", "the", "lazy", "dog." };

var phrase = words.Aggregate((partialPhrase, word) => $"{partialPhrase} {word}");
System.Console.WriteLine(phrase);
```

与其他集合连接方法相比，该选项可能会导致更多的分配，因为每次迭代它都会创建一个中间字符串。如果优化性能至关重要，请考虑使用 [StringBuilder](#) 类或 [String.Concat](#) 或 [String.Join](#) 方法来连接集合，而不是使用 [Enumerable.Aggregate](#)。

请参阅

- [String](#)
- [StringBuilder](#)
- [C# 编程指南](#)
- [字符串](#)

如何搜索字符串

2020/11/2 • [Edit Online](#)

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。

NOTE

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：

```
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。

```

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 `\\\`)。

元素	说明
<code>the</code>	匹配文本“the”
<code>(eir)?</code>	匹配 0 个或 1 个“eir”
<code>\s</code>	与空白符匹配

```

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.WriteLine($"{s,24}");

    if (System.Text.RegularExpressions.Regex.IsMatch(s, sPattern,
System.Text.RegularExpressions.RegexOptions.IgnoreCase))
    {
        Console.WriteLine($" (match for '{sPattern}' found)");
    }
    else
    {
        Console.WriteLine();
    }
}

```

TIP

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](#)。

正则表达式	说明
<code>^</code>	匹配字符串的开头部分
<code>\d{3}</code>	完全匹配 3 位字符
<code>-</code>	匹配字符“-”
<code>\d{3}</code>	完全匹配 3 位字符
<code>-</code>	匹配字符“-”
<code>\d{4}</code>	完全匹配 4 位字符
<code>\$</code>	匹配字符串的结尾部分

```
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.WriteLine($"{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。

请参阅

- [C# 编程指南](#)
- [字符串](#)
- [LINQ 和字符串](#)
- [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](#)

如何修改以 C# 编写的字符串内容

2020/11/2 • [Edit Online](#)

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`。

NOTE

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。

```
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“替换为”`_`：

```
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。

```
// 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。此方法移除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:

```
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](#)。

```

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.ToUpper(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。

```

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。

```
// 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

2021/3/9 • [Edit Online](#)

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。

NOTE

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，分别为相等运算符 `==` 和 `!=`）

执行区分大小写的序号比较，并在需要的情况下使用当前区域性。下面的示例演示了这一操作：

```
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.OrdinalIgnoreCase);
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。虽然相等性测试执行区分大小写的序号比较，但比较方法使用当前区域性执行区分大小写、区分区域性的比较。由于默认比较方法通常执行不同类型的比较，因此建议始终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

`String.Equals(String, StringComparison)` 方法允许指定 `StringComparison.OrdinalIgnoreCase` 的 `StringComparison` 值对于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，前提是您指定 `StringComparison` 参数的 `StringComparison.OrdinalIgnoreCase` 值。如下代码所示：

```
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, 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

可以使用当前区域性的语义规则来对字符串进行排序。这有时被称为“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。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，在德语中，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”。

```

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");
}

```

此示例说明了语义比较是依赖于操作系统的。交互式窗口的主机为 Linux 主机。语义比较和序号比较的结果是一样的。如果在 Windows 主机上运行同样的示例，会看到如下输出内容：

```

<coop> is less than <co-op> using invariant culture
<coop> is greater than <co-op> using ordinal comparison
<coop> is less than <cop> using invariant culture
<coop> is less than <cop> using ordinal comparison
<co-op> is less than <cop> using invariant culture
<co-op> is less than <cop> using ordinal comparison

```

在 Windows 上，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。

使用特定区域性的比较

此示例为 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：

```

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。此外，在本地 Windows 计算机上尝试下列示例代码，将得到下列结果：

```

<coop> is less than <co-op> using en-US culture
<coop> is greater than <co-op> using ordinal comparison
<coop> is less than <cop> using en-US culture
<coop> is less than <cop> using ordinal comparison
<co-op> is less than <cop> using en-US culture
<co-op> is less than <cop> using ordinal comparison

```

语义比较取决于当前区域性以及操作系统。在进行字符串比较时，请考虑这些内容。

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。

此示例演示如何使用当前区域性对字符串数组进行排序：

```
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。

```

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。

```

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：

```

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`。

引用相等性和字符串集中

这些示例都没有使用 `ReferenceEquals`。此方法确定两个字符串是否为同一对象, 这可能导致字符串比较中出现不一致的结果。以下示例演示了 C# 中的字符串集中功能。如果程序声明了 2 个或多个相同的字符串变量, 则编译器会将其存储在同一位置。通过调用 `ReferenceEquals` 方法, 可以看到这两个字符串实际上引用的是内存中的同一对象。使用 `String.Copy` 方法可避免集中。创建副本后, 两个字符串存储在不同位置(即使它们具有相同的值)。运行下列示例以显示字符串 `a` 和 `b` 是集中的, 也就是说它们共享相同的存储。字符串 `a` 和 `c` 不是。

```
string a = "The computer ate my source code.";
string b = "The computer ate my source code.";

if (String.ReferenceEquals(a, b))
    Console.WriteLine("a and b are interned.");
else
    Console.WriteLine("a and b are not interned.");

string c = String.Copy(a);

if (String.ReferenceEquals(a, c))
    Console.WriteLine("a and c are interned.");
else
    Console.WriteLine("a and c are not interned.");
```

NOTE

测试字符串是否相等时, 使用的方法应显式指定要执行的比较类型。你的代码具备更强的可维护性和可读性。重载采用了 `StringComparison` 枚举参数的 `System.String` 和 `System.Array` 类的方法。指定要执行的比较类型。在测试相等性时, 请避免使用 `==` 和 `!=` 运算符。`String.CompareTo` 实例方法始终执行区分大小写的序号比较。它们主要适用于按字母顺序进行的字符串排序。

可以通过调用 `String.Intern` 方法暂存字符串或检索对现有暂存字符串的引用。要确定字符串是否暂存, 请调用 `String.IsInterned` 方法。

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](#)

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

2020/11/2 • [Edit Online](#)

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：

```

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 { }

class Program
{
    static void Main(string[] args)
    {
        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);
        // Output:
        // I am an animal.
        // SuperNova is not a Mammal
    }

    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");
        }
    }
}

```

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。以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：

```
class Program
{
    static void Main(string[] args)
    {
        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(System.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(System.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：

```
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 { }

class Program
{
    static void Main(string[] args)
    {
        // Use the is operator to verify the type.
        // before performing a cast.
        Giraffe g = new Giraffe();
        UseIsOperator(g);

        // Use the as operator and test for null
        // before referencing the variable.
        UseAsOperator(g);

        // Use the as operator to test
        // an incompatible type.
        SuperNova sn = new SuperNova();
        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 != 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 != null)
    {
        Console.WriteLine(j);
    }
    else
    {
        Console.WriteLine("Could not convert " + val.ToString());
    }
}
```

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。

.NET Compiler Platform SDK

2021/3/5 • [Edit Online](#)

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。

使用 .NET Compiler Platform SDK，可以生成分析器和 代码修补程序，从而发现和更正编码错误。分析器理解语法（代码的结构）和语义，以检测应更正的做法。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。

TIP

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 - 工作负载视图

Visual Studio 扩展开发工作负载中不会自动选择 .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”工作负载。
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。

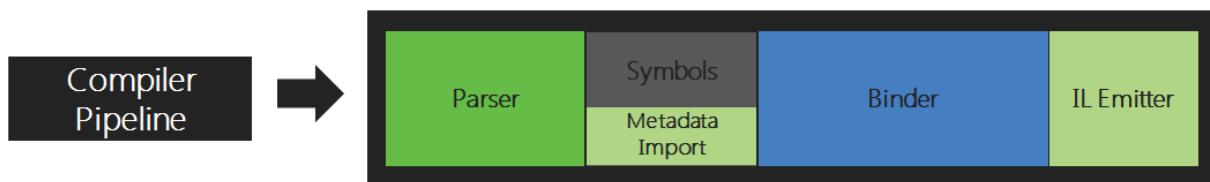
了解 .NET Compiler Platform SDK 模型

2021/3/5 • [Edit Online](#)

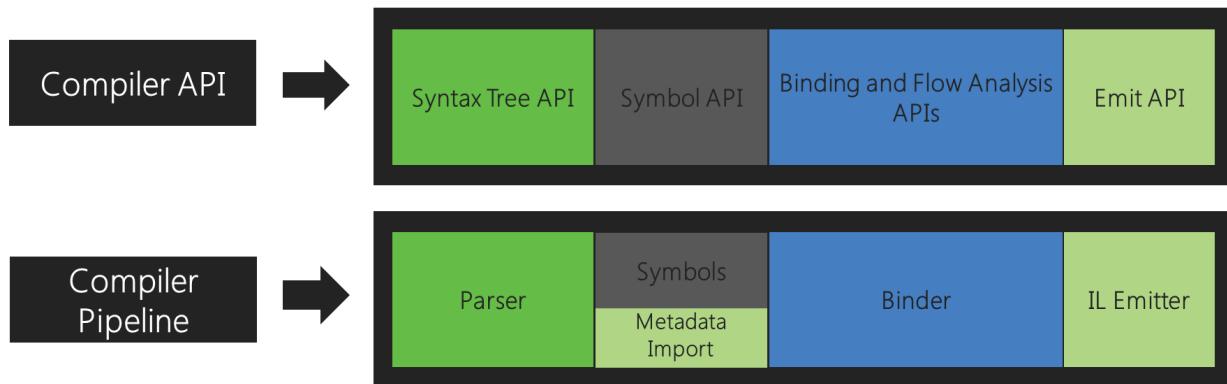
编译器按照结构化规则处理代码，这些规则通常不同于用户读取和理解代码的方式。基本了解编译器使用的模型对于了解生成基于 Roslyn 的工具时使用的 API 至关重要。

编译器管道功能区域

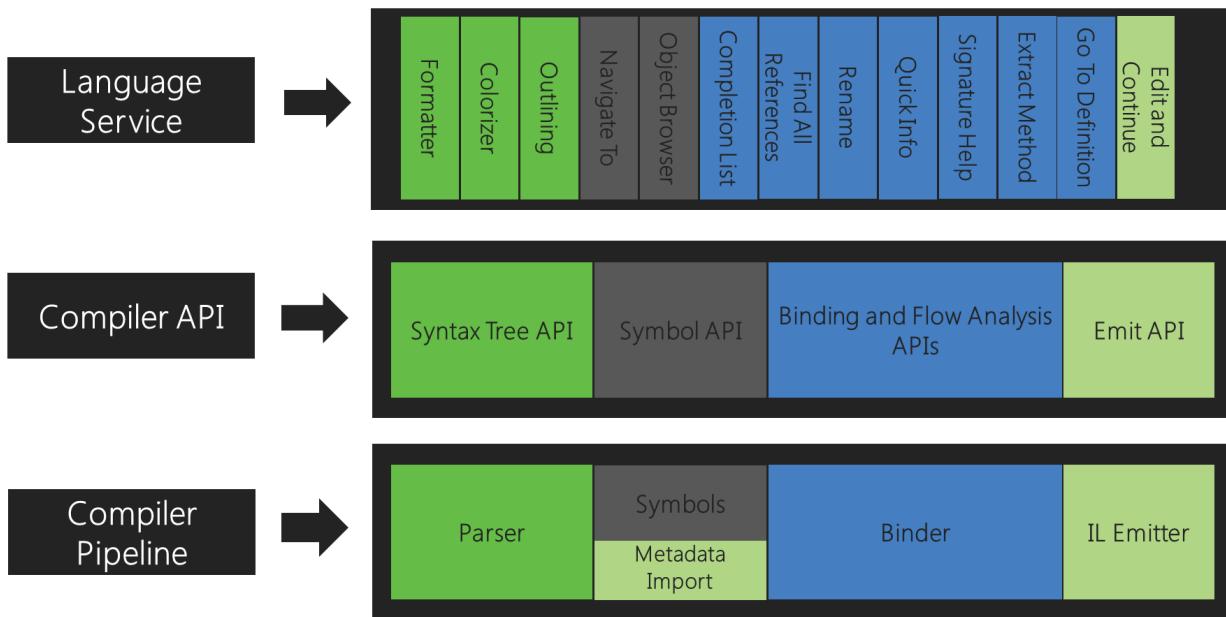
.NET Compiler Platform SDK 通过提供镜像传统编译器管道的 API 层，作为使用者向用户公开 C# 和 Visual Basic 编译器的代码分析。



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 是编译器层的一部分。可使用它们来执行代码片段和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，以交互方式执行代码。

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。

使用语法

2021/3/5 • • [Edit Online](#)

“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：

```
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`），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。

TIP

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

2021/3/5 • • [Edit Online](#)

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

2021/3/5 • [Edit Online](#)

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

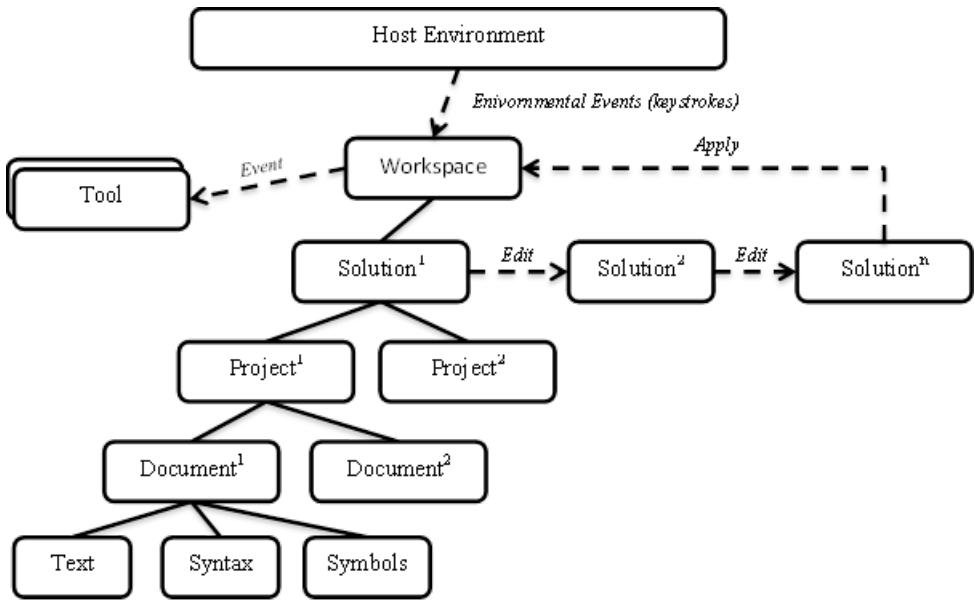
尽管每次按下一个键时工作区都可能会更改，仍可以隔离方式使用解决方案模型。

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

2021/5/7 • [Edit Online](#)

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 - 工作负载视图

Visual Studio 扩展开发工作负载中不会自动选择 .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”工作负载。
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：

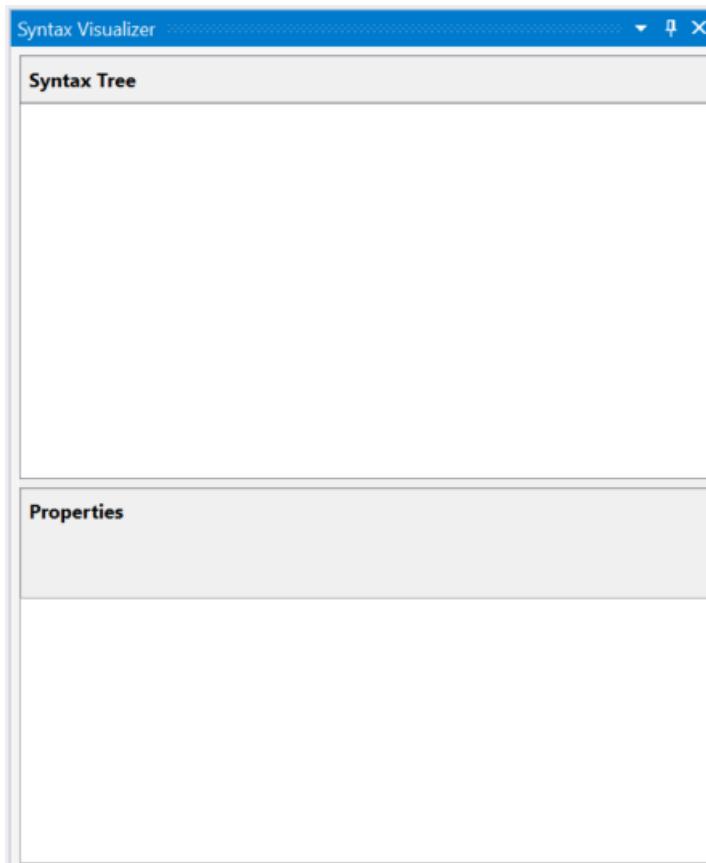
1. 选中“DGML 编辑器”框。将在“代码工具”部分下找到它。

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。

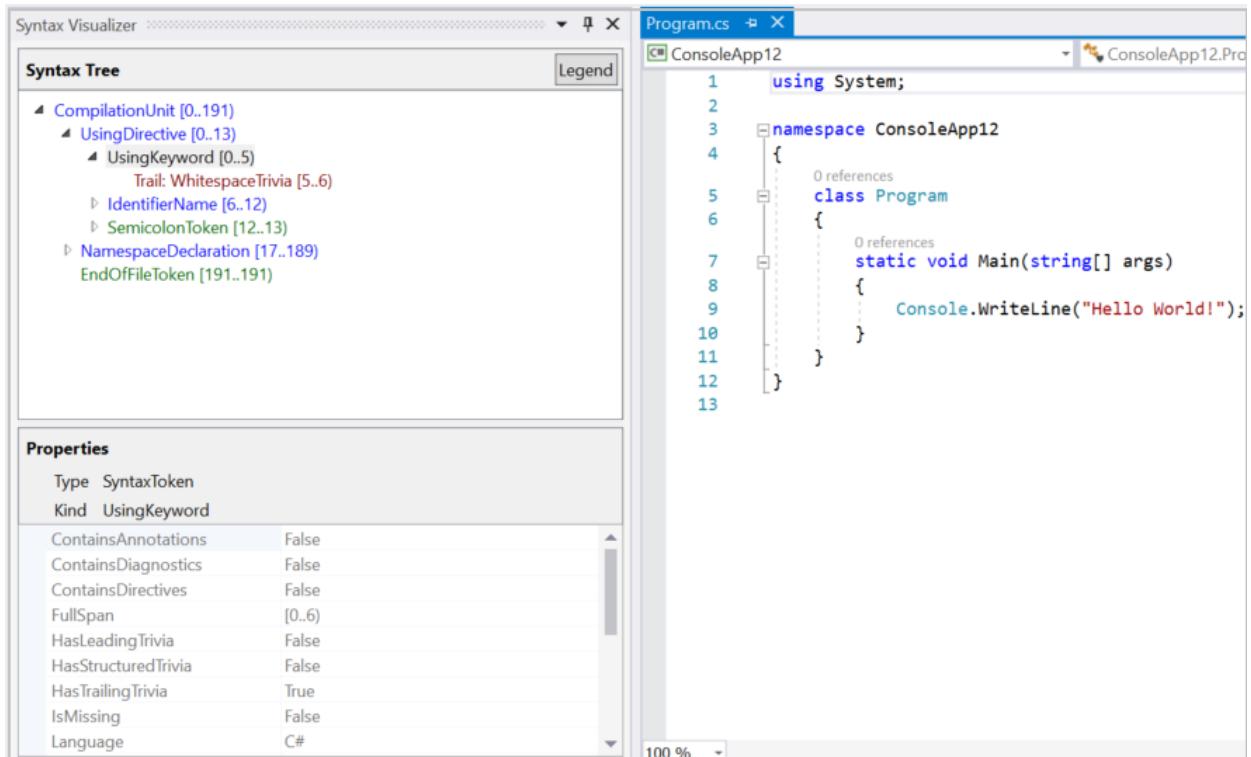
此命令会以浮动工具窗口的形式打开语法可视化工具。如果没有打开代码编辑器窗口，则显示为空白，如下图所示。



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。

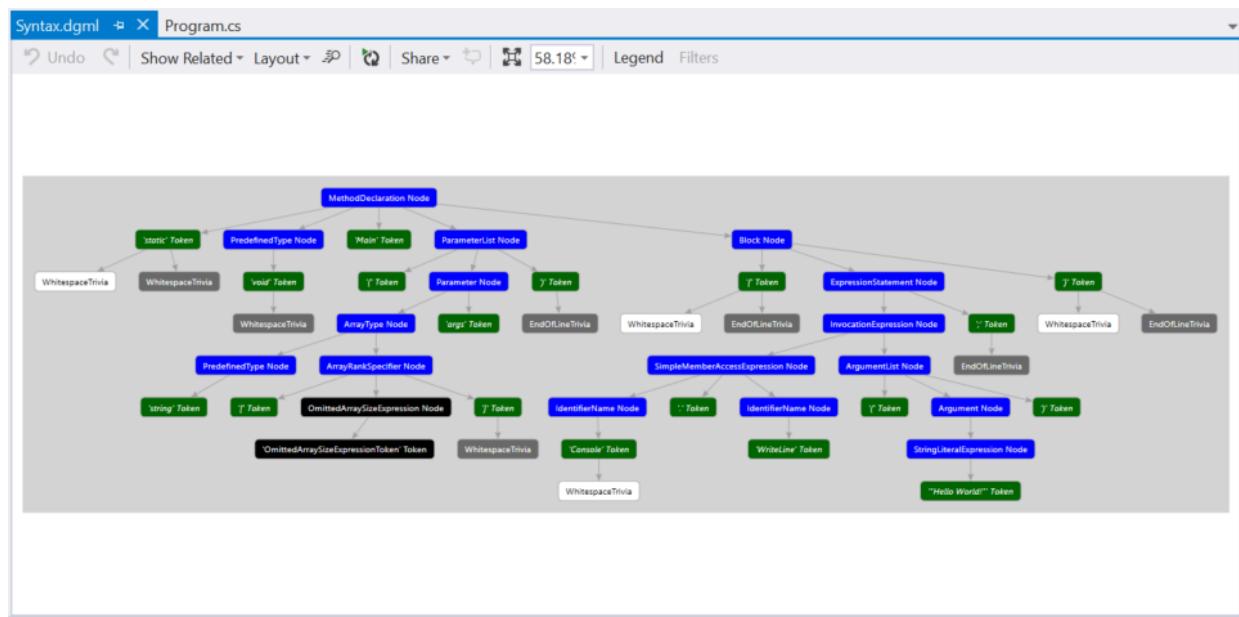
请在键入 `Console.` 后暂停键入。树中有一些粉色的项。这说明在所键入的代码中存在错误(通常也成为“诊断”)。这些错误会附加到语法树的节点、标记和琐碎内容中。可视化工具会显示哪些项存在错误，并用粉色突出显示其背景。将鼠标悬停在该项上可以查看任何粉色项的错误。可视化工具只显示语法错误(这些错误与键入代码的语法相关)；不会显示任何语义错误。

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：

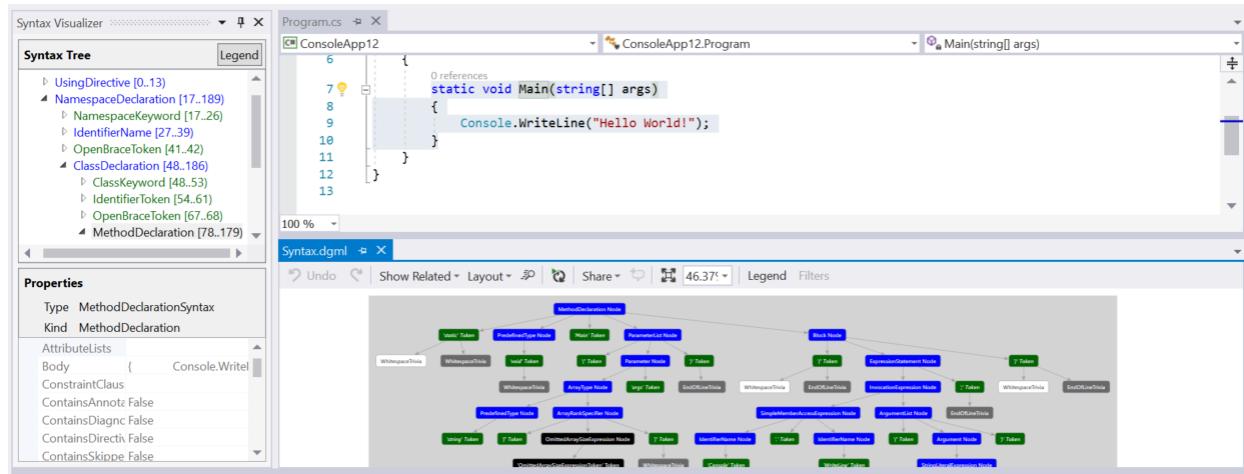


语法关系图查看器中有一个选项，可用于显示其着色方案的图例。还可以将鼠标悬停在语法关系图中的每个项上，以查看该项对应的属性。

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:

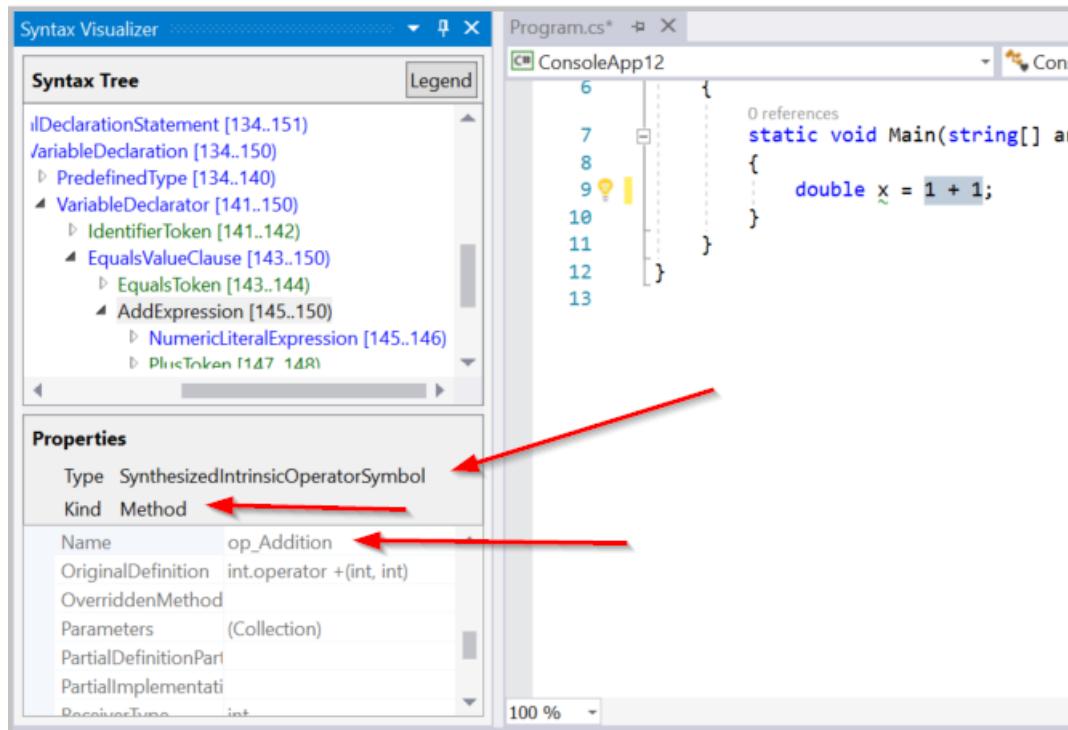


另一种选择在双监视器配置中, 将语法关系图窗口放在第二个监视器上。

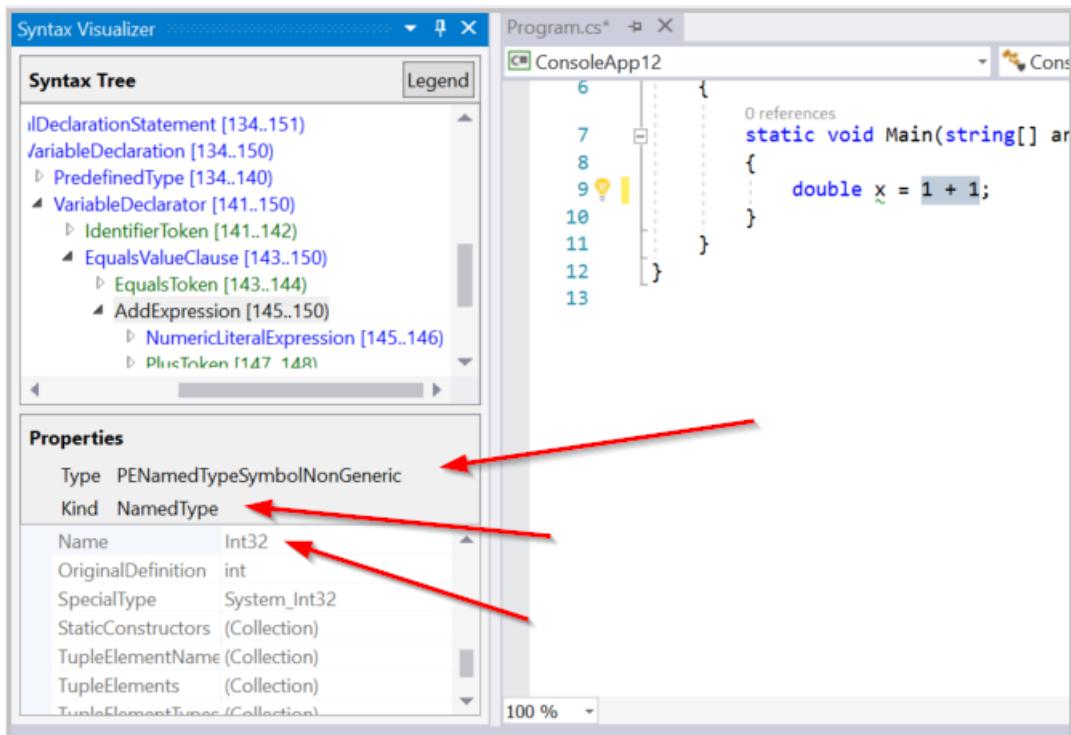
检查语义

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。

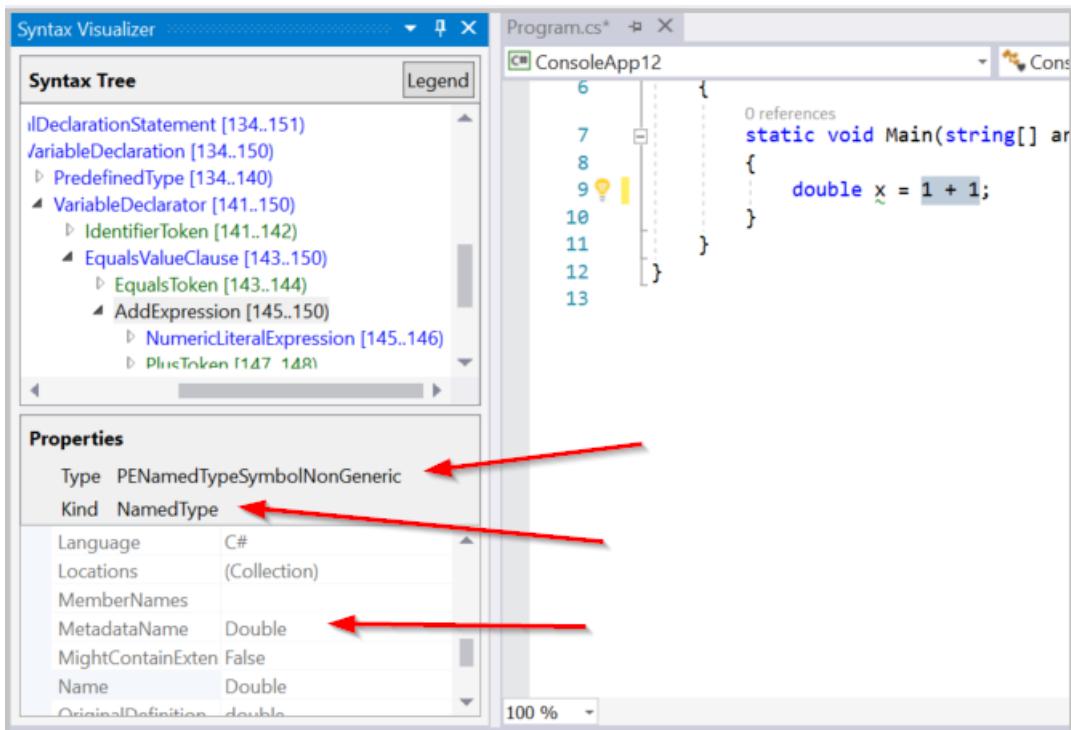
可视化工具中的属性网格更新如下图所示:该表达式的符号是 SynthesizedIntrinsicOperatorSymbol, 其中种类 = 方法。



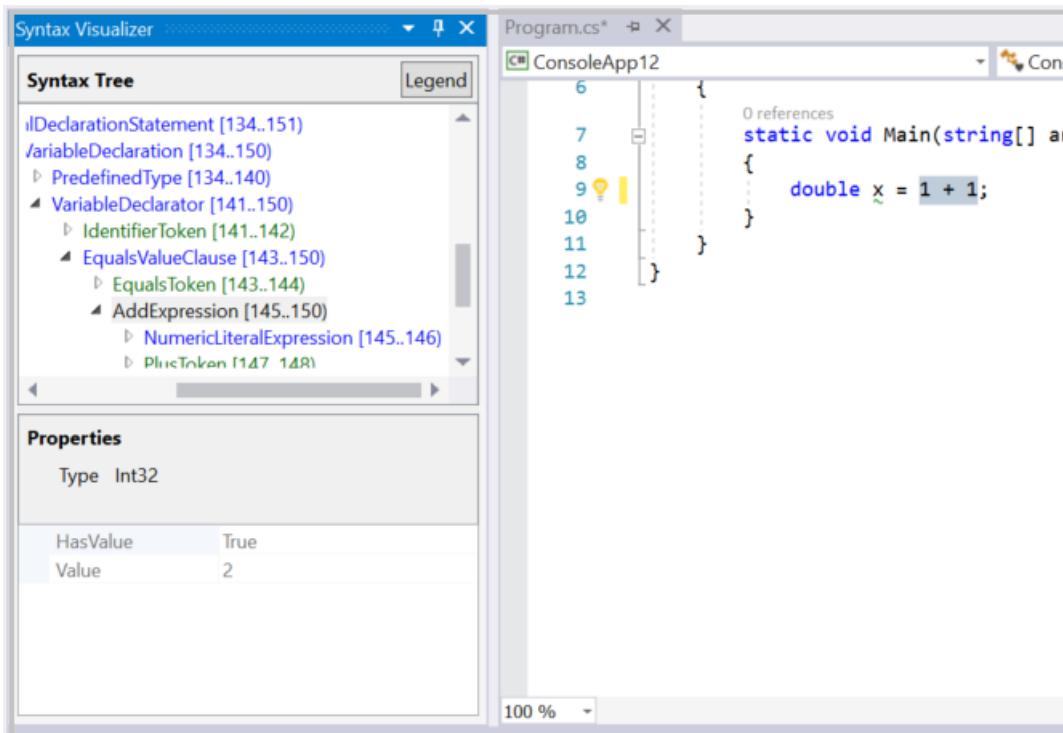
针对同一个 AddExpression 节点, 请尝试“查看 TypeSymbol (如果有)”。如下图所示, 可视化工具中的属性网格已更新, 指示所选表达式的类型为 `Int32`。



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。



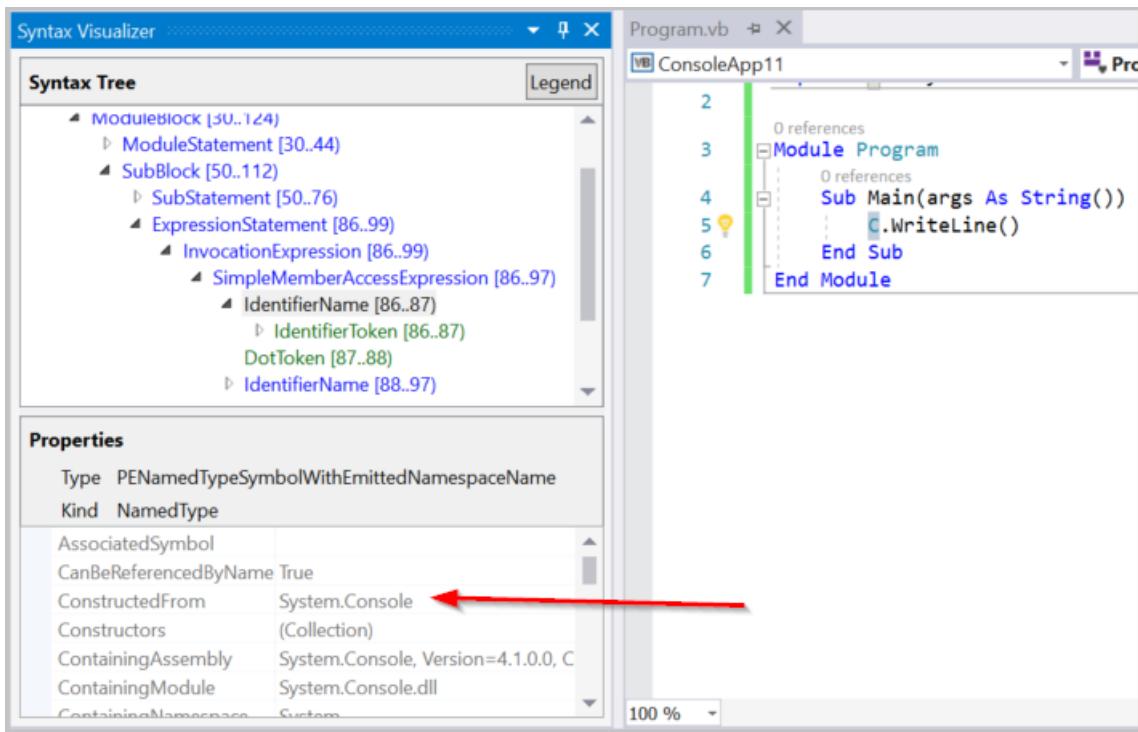
在 Visual Basic 中也可以重复上述示例。在 Visual Basic 文件中键入 `Dim x As Double = 1 + 1`。在代码编辑器窗口中选择表达式 `1 + 1`。可视化工具突出显示了对应的 AddExpression 节点。对此 AddExpression 节点重复前面所述的步骤，应该能看到相同的结果。

检查 Visual Basic 中的更多代码。使用以下代码更新主 Visual Basic 文件：

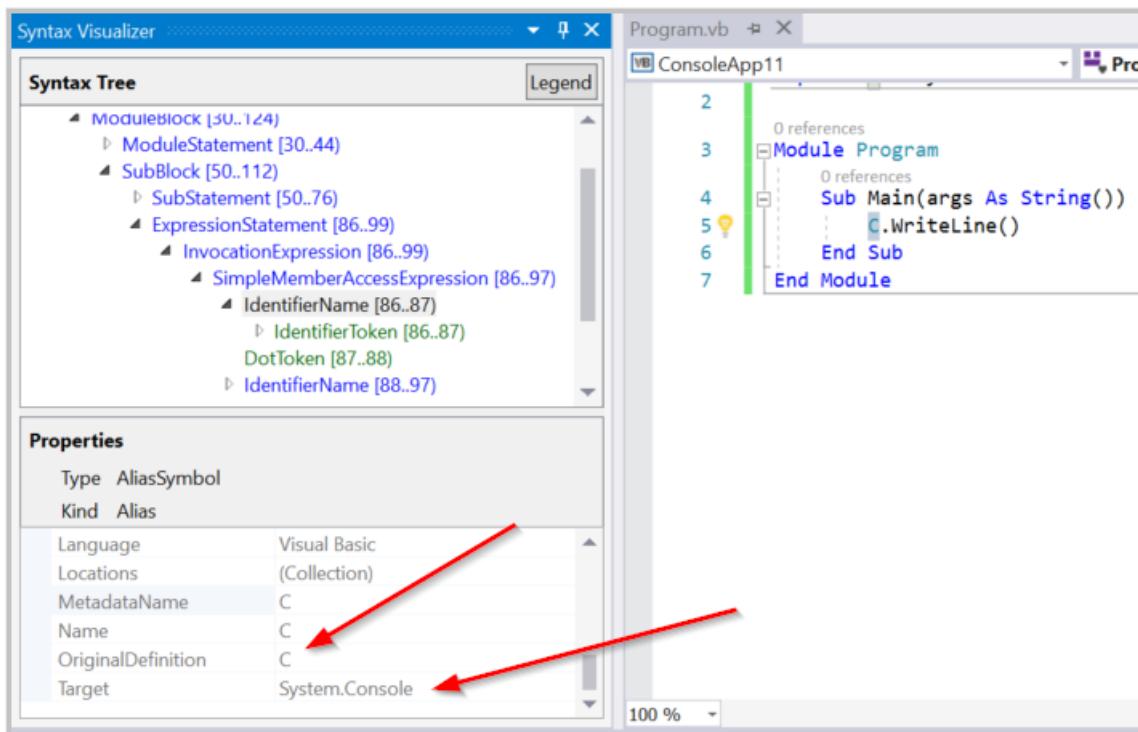
```
Imports C = System.Console

Module Program
    Sub Main(args As String())
        C.WriteLine()
    End Sub
End Module
```

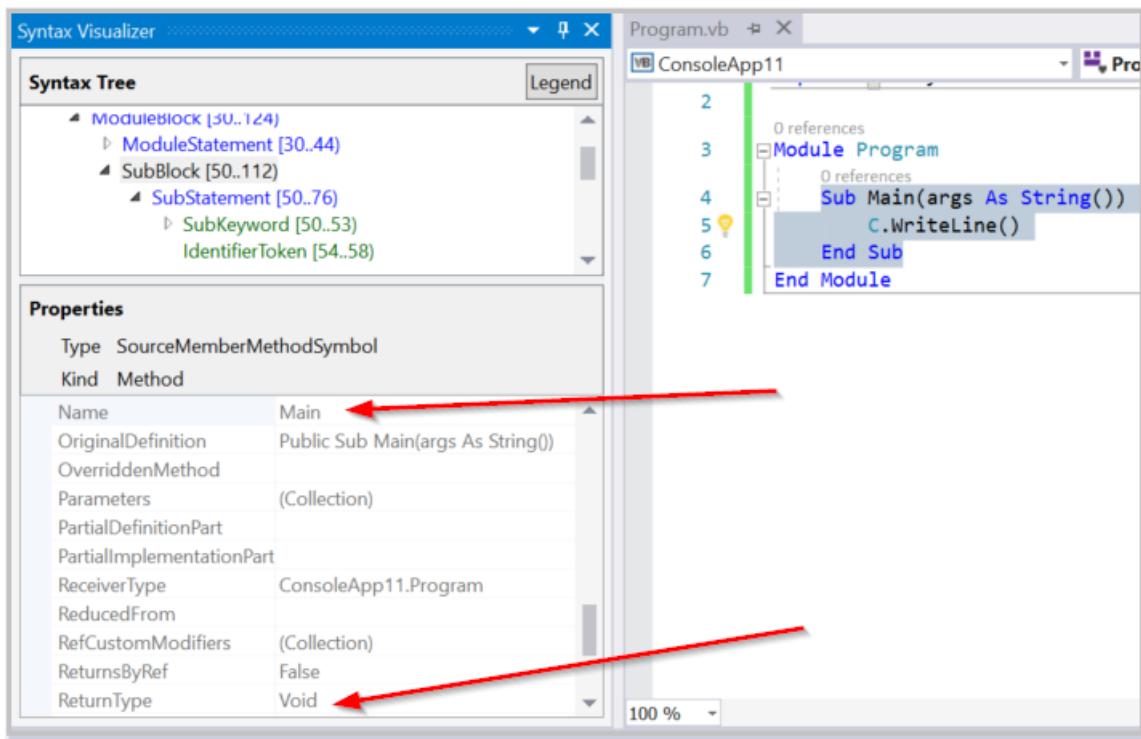
此代码引入了映射到文件顶部的 `System.Console` 类型的 `C` 别名，并在 `Main()` 内使用此别名。选择在 `Main()` 方法的 `C.WriteLine()` 中使用此别名 `C`。可视化工具会选择对应的 IdentifierName 节点。右键单击此节点，并单击“查看符号(如果有)”。属性网格指示此标识符绑定至 `System.Console` 类型，如下图所示：



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。

语法分析入门

2021/5/7 • [Edit Online](#)

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：

```
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。树结构一直到最低级别：字符串“Hello World!”是“字符串文本标记”，即一种参数的子代。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 - 工作负载视图

Visual Studio 扩展开发工作负载中不会自动选择 .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”工作负载。
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。

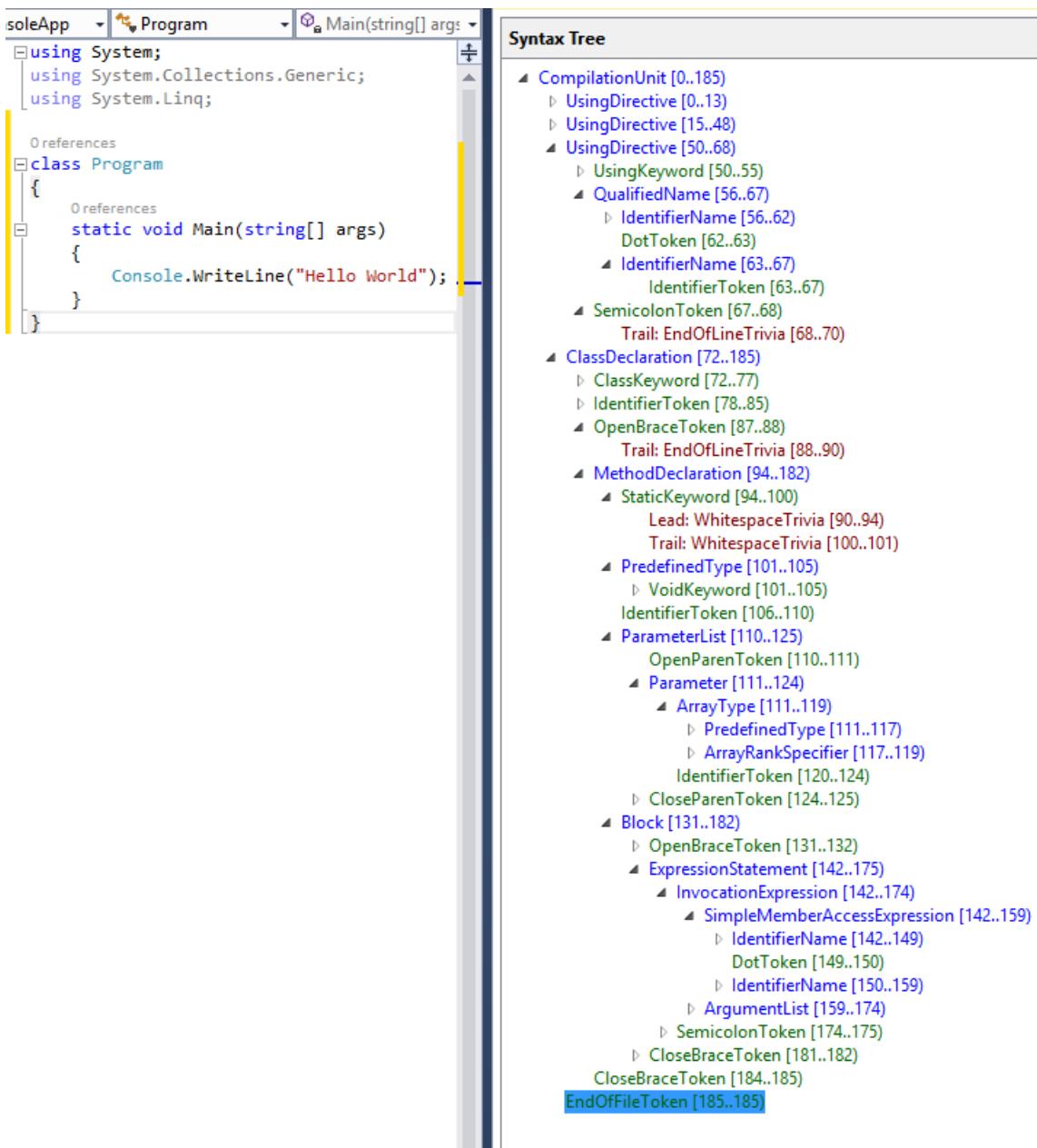
语法树是 C# 和 Visual Basic 编译器用于理解 C# 和 Visual Basic 程序的数据结构。语法树由生成项目时或开发人员按 F5 时所运行的分析程序生成。语法树对语言完全保真；代码文件中的每一位信息都在树中。将语法树写入文本会再现已分析的完全原始文本。语法树也是不可变的；一旦创建语法树，就不能再更改。树的使用者可以在多个线程上对树进行分析，不需要锁或其他并发度量，很清楚数据是不会更改的。可使用 API 新建树，新建的树就是对现有树进行修改后的成果。

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。

NOTE

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”。

你将分析上面展示过的基本“Hello World!” 程序。为 Hello World 程序添加文本，作为 `Program` 类中的常量：

```
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：

```
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：

```
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：

```
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：

```
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。

```
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：

```
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.ToString());

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：

```
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：

```
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：

```
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：

```
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。

```
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：

```
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：

```
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：

```
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：

```
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

2021/5/7 • [Edit Online](#)

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 - 工作负载视图

Visual Studio 扩展开发工作负载中不会自动选择 .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”工作负载。
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。

NOTE

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”。

你将分析上面展示过的基本“Hello World!” 程序。为 Hello World 程序添加文本, 作为 `Program` 类中的常量：

```
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：

```
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：

```
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：

```
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:

```
// 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。语法模型在 `using` 语句中找到字符串 `System`。语义模型具有关于 `System` 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:

```
var systemSymbol = (INamespaceSymbol)nameInfo.Symbol;  
foreach (INamespaceSymbol ns in systemSymbol.GetNamespaceMembers())  
{  
    Console.WriteLine(ns);  
}
```

运行该程序, 然后应看到以下输出:

```
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 . . .
```

NOTE

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”程序包含一个 `Microsoft.CodeAnalysis.CSharp.Syntax.LiteralExpressionSyntax`，即“Hello, World!”向控制台显示的字符串。

通过找到程序中的单个字符串文本 查找到“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：

```
// 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：

```
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：

```
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：

```
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：

```
var publicStringReturningMethods = methods
    .Where(m => m.ReturnType.Equals(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：

```
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：

```
foreach (string name in (from method in stringTypeSymbol
    .GetMembers().OfType<IMethodSymbol>()
    where method.ReturnType.Equals(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：

```
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

2021/5/7 • • [Edit Online](#)

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 - 工作负载视图

Visual Studio 扩展开发工作负载中不会自动选择 .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”工作负载。
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。

```
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：

```
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`：

```
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`：

```
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：

```
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!");
        }
    }
}";
```

NOTE

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：

```
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：

```
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：

```
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：

```
using Microsoft.CodeAnalysis;
using Microsoft.CodeAnalysis.CSharp;
using Microsoft.CodeAnalysis.CSharp.Syntax;
```

接下来，使 `TypeInferenceRewriter` 类扩展 `CSharpSyntaxRewriter` 类：

```
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：

```
private readonly SemanticModel SemanticModel;

public TypeInferenceRewriter(SemanticModel semanticModel) => SemanticModel = semanticModel;
```

重写 `VisitLocalDeclarationStatement(LocalDeclarationStatementSyntax)` 方法：

```
public override SyntaxNode VisitLocalDeclarationStatement(LocalDeclarationStatementSyntax node)
{
}
```

NOTE

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：

```
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：

```
// 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：

```
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：

```
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：

```
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：

```
if (SymbolEqualityComparer.Default.Equals(variableType, initializerInfo.Type))  
{  
    TypeSyntax varTypeName = SyntaxFactory.IdentifierName("var")  
        .WithLeadingTrivia(variableTypeName.GetLeadingTrivia())  
        .WithTrailingTrivia(variableTypeName.GetTrailingTrivia());  
  
    return node.ReplaceNode(variableType, varTypeName);  
}  
else  
{  
    return node;  
}
```

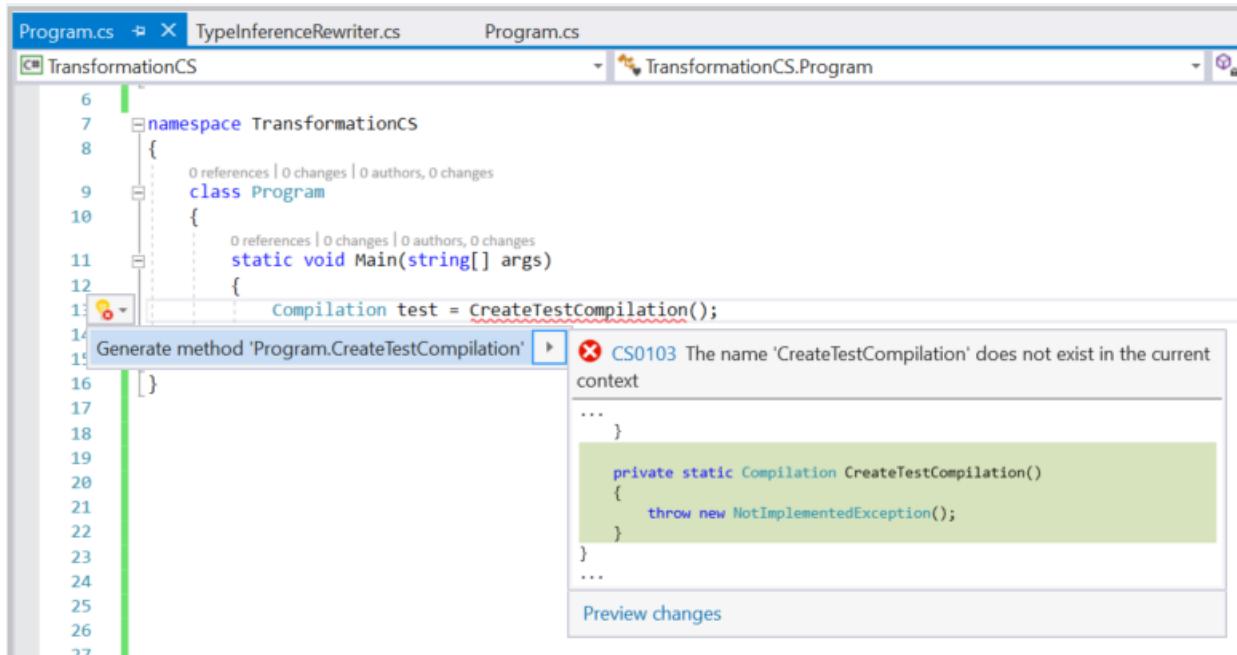
此条件是必需的，因为声明可能将初始值设定项表达式转换为基类或接口。如果这是需要的，则分配左侧和右侧

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：

```
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`。Write the following code to iterate over each `SyntaxTree` in the test `Compilation`. 对于每一个，都使用 `SemanticModel` 初始化该树的新 `TypeInferenceRewriter`：For each one, initialize a new `TypeInferenceRewriter` with the `SemanticModel` for that tree:

```
[!code-csharp[IterateTrees](../../../../samples/snippets/csharp/roslyn-sdk/SyntaxTransformationQuickStart/TransformationCS/Program.cs#IterateTrees "Iterate all the source trees in the test compilation")]
```

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：

```
SyntaxNode newSource = rewriter.Visit(sourceTree.GetRoot());

if (newSource != sourceTree.GetRoot())
{
    File.WriteAllText(sourceTree.FilePath, newSource.ToString());
}
```

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：

```
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

2021/5/8 • [Edit Online](#)

.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所有内容。.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 - 工作负载视图

Visual Studio 扩展开发工作负载中不会自动选择 .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”工作负载。
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”。

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。

NOTE

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。

TIP

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：

```
namespace ConsoleApp1
{
    0 references
    class Program
    {
        0 references
        static void Main(string[] args)
        {
            Console.WriteLine("Hello World!");
        }
    }
}
```

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。

TIP

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：

```
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：

```
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 `MakeConstAnalyzer.Initialize` 方法中，找到在符号上注册操作的行：

```
context.RegisterSymbolAction(AnalyzeSymbol, SymbolKind.NamedType);
```

使用下面的行替换它：

```
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：

```
private void AnalyzeNode(SyntaxNodeAnalysisContext context)
{
}
```

将 `Category` 更改为 `MakeConstAnalyzer.cs` 中的“Usage”，如以下代码所示：

```
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：

```
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`：

```
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：

```
// 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：

```
// 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：

```
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：

```
int x = 0;  
Console.WriteLine(x);
```

应显示灯泡，且分析器应报告诊断。但是，灯泡仍使用模板生成的代码修补程序，并告知你可以用大写。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：

```
- 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，但它尚未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：

```
var declaration =
root.FindToken(diagnosticSpan.Start).Parent.AncestorsAndSelf().OfType<LocalDeclarationStatementSyntax>
().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：

```
// 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：

```
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：

```
// 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：

```
// 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：

```
// 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：

```
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：

```
// 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。

```
static void Main(string[] args)  
{  
    int i = 1;  
    int j = 2;  
    int k = i + j;  
}
```

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。

TIP

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：

```

[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：

```

[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：

```

[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：

```

[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：

```

[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：

```

[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：

```

// 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：

```

// 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：

```

[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：

```
[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：

```
[TestMethod]
public async Task StringCouldBeConstant_Diagnostic()
{
    await VerifyCS.VerifyCodeFixAsync(@""
using System;

class Program
{
    static void Main()
    {
        [|string s = ""abc"";|]
    }
},
",
");
}

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：

```

[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：

```

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：

```
// 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。

```
// 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：

```

// 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.ToString())
                .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 = variableDeclarationWithType(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：

```
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：

```
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](#)

C# 编程指南

2021/5/10 • [Edit Online](#)

此部分详细介绍了 C# 语言主要功能，以及可通过 .NET 在 C# 中使用的功能。

阅读此部分的大部分内容的前提是，你已对 C# 和一般编程概念有一定的了解。如果完全没有接触过编程或 C#，建议参阅 [C# 教程简介](#) 或 [.NET 浏览器内教程](#)，此教程不需要具备任何编程知识。

若要了解特定的关键字、运算符和预处理器指令，请参阅 [C# 参考](#)。若要了解 C# 语言规范，请参阅 [C# 语言规范](#)。

程序部分

[在 C# 程序内部](#)

[Main\(\) 和命令行参数](#)

语言部分

[语句、表达式和运算符](#)

[类型](#)

[类、结构和记录](#)

[接口](#)

[委托](#)

[数组](#)

[字符串](#)

[属性](#)

[索引器](#)

[事件](#)

[泛型](#)

[迭代器](#)

[LINQ 查询表达式](#)

[命名空间](#)

[不安全代码和指针](#)

[XML 文档注释](#)

平台部分

[应用程序域](#)

[.NET 中的程序集](#)

[特性](#)

[集合](#)

[异常和异常处理](#)

[文件系统和注册表\(C# 编程指南\)](#)

[互操作性](#)

[反射](#)

[请参阅](#)

- [C# 参考](#)

C# 程序内部探究

2021/3/5 • [Edit Online](#)

本节讨论 C# 程序的一般结构，并包括标准的“Hello, World!”示例。

本节内容

- [C# 程序的通用结构](#)
- [标识符名称](#)
- [C# 编码约定](#)

相关章节

- [C# 入门](#)
- [C# 编程指南](#)
- [C# 参考](#)
- [示例和教程](#)

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](#)

C# 程序的通用结构 (C# 编程指南)

2020/11/2 • [Edit Online](#)

C# 程序可由一个或多个文件组成。每个文件均可包含零个或多个命名空间。一个命名空间可包含类、结构、接口、枚举、委托等类型以及其他命名空间。下面是包含所有这些元素的 C# 程序主干。

```
// 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 YourMainClass
    {
        static void Main(string[] args)
        {
            //Your program starts here...
        }
    }
}
```

相关章节

更多相关信息：

- [类](#)
- [结构](#)
- [命名空间](#)
- [接口](#)
- [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。

请参阅

- [C# 编程指南](#)
- [在 C# 程序内部](#)
- [C# 参考](#)

标识符名称

2020/3/18 • [Edit Online](#)

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。有效标识符必须遵循以下规则：

- 标识符必须以字母或 `_` 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主题](#)。

命名约定

除了规则之外，在 .NET API 中还使用了许多标识符[命名约定](#)。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，以下约定也很常见：

- 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 `_` 字符。这些名称保留给编译器生成的标识符。

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

- [C# 编程指南](#)
- [在 C# 程序内部](#)
- [C# 参考](#)
- [类](#)
- [结构类型](#)
- [命名空间](#)
- [接口](#)
- [委托](#)

C# 编码约定 (C# 编程指南)

2021/5/7 • [Edit Online](#)

编码约定可实现以下目的：

- 它们为代码创建一致的外观，以确保读取器专注于内容而非布局。
- 它们使得读取器可以通过基于之前的经验进行的假设更快地理解代码。
- 它们便于复制、更改和维护代码。
- 它们展示 C# 最佳做法。

Microsoft 根据本文中的准则来开发样本和文档。

命名约定

- 在不包括 [using 指令](#) 的短示例中，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。

```
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。

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。

```
if ((val1 > val2) && (val1 > val3))  
{  
    // Take appropriate action.  
}
```

注释约定

- 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。

```
// The following declaration creates a query. It does not run  
// the query.
```

- 请勿在注释周围创建格式化的星号块。

语言准则

以下各节介绍 C# 遵循以准备代码示例和样本的做法。

字符串数据类型

- 使用字符串内插来连接短字符串，如下面的代码所示。

```
string displayName = $"{nameList[n].LastName}, {nameList[n].FirstName}";
```

- 要在循环中追加字符串，尤其是在使用大量文本时，请使用 `StringBuilder` 对象。

隐式类型本地变量

- 当变量类型明显来自赋值的右侧时，或者当精度类型不重要时，请对本地变量进行**隐式类型化**。

```
var var1 = "This is clearly a string.";  
var var2 = 27;
```

- 当类型并非明显来自赋值的右侧时，请勿使用 var。请勿假设类型明显来自方法名称。如果变量类型为 new 运算符或显式强制转换，则将其视为明显来自方法名称。

```
int var3 = Convert.ToInt32(Console.ReadLine());  
int var4 = ExampleClass.ResultSoFar();
```

- 请勿依靠变量名称来指定变量的类型。它可能不正确。在以下示例中，变量名称 `inputInt` 会产生误导性。它是字符串。

```
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\(C# 编程指南\)](#)。
 - 使用隐式类型化来确定 `for` 循环中循环变量的类型。

下面的示例在 `for` 语句中使用隐式类型化。

- 请勿使用隐式类型化来确定 `foreach` 循环中循环变量的类型。

下面的示例在 `foreach` 语句中使用显式类型化。

```
foreach (char ch in laugh)
{
    if (ch == 'h')
        Console.WriteLine("H");
    else
        Console.Write(ch);
}
Console.WriteLine();
```

NOTE

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。

无符号数据类型

通常，使用 `int` 而非无符号类型。`int` 的使用在整个 C# 中都很常见，并且当你使用 `int` 时，更易于与其他库交互。

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，请注意不能使用 `var` 替代 `String[]`。

```
string[] vowels1 = { "a", "e", "i", "o", "u" };
```

如果使用显式实例化，则可以使用 `var`。

```
var vowels2 = new string[] { "a", "e", "i", "o", "u" };
```

如果指定数组大小，只能一次初始化一个元素。

```
var vowels3 = new string[5];
vowels3[0] = "a";
vowels3[1] = "e";
// And so on.
```

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。

```
public static Action<string> ActionExample1 = x => Console.WriteLine($"x is: {x}");

public static Action<string, string> ActionExample2 = (x, y) =>
    Console.WriteLine($"x is: {x}, y is {y}");

public static Func<string, int> FuncExample1 = x => Convert.ToInt32(x);

public static 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。

```
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。

```
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。

```
Del exampleDel2 = DelMethod;
exampleDel2("Hey");
```

以下声明使用了完整的语法。

```
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。

```
static string GetValueFromArray(string[] array, int index)
{
    try
    {
        return array[index];
    }
    catch (System.IndexOutOfRangeException ex)
    {
        Console.WriteLine("Index is out of range: {0}", index);
        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`。

```
Font font1 = new Font("Arial", 10.0f);
try
{
    byte charset = font1.GdiCharSet;
}
finally
{
    if (font1 != null)
    {
        ((IDisposable)font1).Dispose();
    }
}
```

可以使用 `using` 语句执行相同的操作。

```
using (Font font2 = new Font("Arial", 10.0f))
{
    byte charset2 = font2.GdiCharSet;
}
```

在 C# 8 及更高版本中，使用无需大括号的新的 `using` 语法：

```
using Font font3 = new Font("Arial", 10.0f);
byte charset3 = font3.GdiCharSet;
```

`&&` 和 `||` 运算符

若要通过跳过不必要的比较来避免异常并提高性能，请在执行比较时使用 `&&`（而不是 `&`）和 `||`（而不是 `|`），如下面的示例所示。

```
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 > 0))
{
    Console.WriteLine("Quotient: {0}", dividend / divisor);
}
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。第二个示例显示了从 C# 9 开始可用的语法。

```
var instance1 = new ExampleClass();
```

```
ExampleClass instance2 = new();
```

前面的声明等效于下面的声明。

```
ExampleClass instance2 = 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。

```
var instance3 = 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。

```
var instance4 = new ExampleClass();
instance4.Name = "Desktop";
instance4.ID = 37414;
instance4.Location = "Redmond";
instance4.Age = 2.3;
```

事件处理

如果你正在定义一个稍后不需要删除的事件处理程序，请使用 lambda 表达式。

```
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。

```
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`。

```
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 大写格式正确大写。

```
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。

```
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。

```
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。
- 在其他查询子句之前使用 `where` 子句, 以确保后面的查询子句作用于经过减少和筛选的数据集。

```
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` 子句以访问内部集合。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。

```
var scoreQuery = from student in students
    from score in student.Scores
    where score > 90
    select new { Last = student.LastName, score };
```

安全性

请遵循[安全编码准则](#)中的准则。

请参阅

- [.NET 运行时编码准则](#)
- [Visual Basic 编码约定](#)
- [安全编码准则](#)

Main() 和命令行参数 (C# 编程指南)

2021/5/7 • [Edit Online](#)

`Main` 方法是 C# 应用程序的入口点。(库和服务不要求使用 `Main` 方法作为入口点)。`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\(C# 编译器选项\)](#)。

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

自 C# 9 起，可以省略 `Main` 方法，并像在 `Main` 方法中一样编写 C# 语句，如下面的示例所示：

```
using System;

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

若要了解如何使用隐式入口点方法编写应用程序代码，请参阅 [顶级语句](#)。

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` 在类或结构中声明。`Main` 必须是 **静态** 方法，不得为 **公共** 方法。(在前面的示例中，它获得的是 **私有** 成员的默认访问权限)。封闭类或结构不一定要是静态的。
- `Main` 可以具有 `void`、`int`，或者以 C# 7.1、`Task` 或 `Task<int>` 返回类型开头。
- 当且仅当 `Main` 返回 `Task` 或 `Task<int>` 时，`Main` 的声明可包括 `async` 修饰符。请注意，该操作可明确排除 `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。

以下是有效 `Main` 签名的列表：

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

上述示例均使用公共访问器修饰符。这是典型操作方式，但不是必需操作方式。

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。

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](#)
- [方法](#)
- [在 C# 程序内部](#)

命令行参数 (C# 编程指南)

2021/5/7 • [Edit Online](#)

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：

MAIN	MAIN
无返回值, 不使用 <code>await</code>	<code>static void Main(string[] args)</code>
返回值, 不使用 <code>await</code>	<code>static int Main(string[] args)</code>
无返回值, 使用 <code>await</code>	<code>static async Task Main(string[] args)</code>
返回值, 使用 <code>await</code>	<code>static async Task<int> Main(string[] args)</code>

如果不使用参数, 可以从方法签名中省略 `args`, 使代码更为简单：

MAIN	MAIN
无返回值, 不使用 <code>await</code>	<code>static void Main()</code>
返回值, 不使用 <code>await</code>	<code>static int Main()</code>
无返回值, 使用 <code>await</code>	<code>static async Task Main()</code>
返回值, 使用 <code>await</code>	<code>static async Task<int> Main()</code>

NOTE

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 `program.cs` 中 `Main` 的签名。Windows 窗体设计器生成的代码创建没有输入参数的 `Main`。还可使用 `Environment.CommandLine` 或 `Environment.GetCommandLineArgs` 从控制台或 Windows 应用程序的任意位置访问命令行参数。

`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:

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

TIP

`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:

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

也可以使用 C# 类型 `long`，其别名为 `Int64`：

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

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

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

有关详细信息，请参阅 [Parse](#) 和 [Convert](#)。

示例

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。

```

// Add a using directive for System if the directive isn't already present.

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}.");
    }
}
// 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，然后导航到包含刚刚创建的文件的文件夹。
3. 输入以下命令以编译应用程序。

```
csc Factorial.cs
```

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：

```
Factorial 3
```

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

NOTE

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。

请参阅

- [System.Environment](#)
- [C# 编程指南](#)
- [Main\(\) 和命令行参数](#)
- [如何显示命令行参数](#)
- [Main\(\) 返回值](#)
- [类](#)

如何显示命令行参数 (C# 编程指南)

2021/5/7 • [Edit Online](#)

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	MAIN
<code>executable.exe a b c</code>	"a" "b" "c"
<code>executable.exe one two</code>	"one" "two"
<code>executable.exe "one two" three</code>	"one two" "three"

NOTE

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。

```
class CommandLine
{
    static void Main(string[] args)
    {
        // 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

- [C# 编程指南](#)

- `Main()` 和命令行参数
- `Main()` 返回值

Main() 返回值 (C# 编程指南)

2021/5/7 • [Edit Online](#)

可以通过以下方式之一定义方法，以从 `Main` 方法返回 `int`：

MAIN ████	MAIN █
不使用 <code>args</code> 或 <code>await</code>	<code>static int Main()</code>
使用 <code>args</code> ，不使用 <code>await</code>	<code>static int Main(string[] args)</code>
不使用 <code>args</code> ，使用 <code>await</code>	<code>static async Task<int> Main()</code>
使用 <code>args</code> 和 <code>await</code>	<code>static async Task<int> Main(string[] args)</code>

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。

MAIN ████	MAIN █
不使用 <code>args</code> 或 <code>await</code>	<code>static void Main()</code>
使用 <code>args</code> ，不使用 <code>await</code>	<code>static void Main(string[] args)</code>
不使用 <code>args</code> ，使用 <code>await</code>	<code>static async Task Main()</code>
使用 <code>args</code> 和 <code>await</code>	<code>static async Task Main(string[] args)</code>

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。

示例

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。

修改 `program.cs` 中的 `Main` 方法，如下所示：

```
// 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。

```
dotnet run

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

示例输出

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

Async Main 返回值

Async Main 返回值将在 `Main` 中调用异步方法时所需的样本代码移动到编译器生成的代码中。以前，需要编写此结构来调用异步代码并确保程序运行至异步操作完成：

```
public static void Main()
{
    AsyncConsoleWork().GetAwaiter().GetResult();
}

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

现在，可以将其替代为：

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

新语法的优点是编译器始终生成正确的代码。

编译器生成的代码

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
```

NOTE

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。

请参阅

- [C# 编程指南](#)
- [C# 参考](#)
- [Main\(\) 和命令行参数](#)
- [如何显示命令行参数](#)

顶级语句 (C# 编程指南)

2021/5/7 • [Edit Online](#)

从 C# 9 开始，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。在这种情况下，编译器将为应用程序生成类和 `Main` 方法入口点。

下面的 `Program.cs` 文件是 C# 9 中一个完整的 C# 程序：

```
using System;

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

一个应用程序必须只能有一个入口点，因此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：

```
using System;

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

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：

```
using System;

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:

```
using System;

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:

```
using System;
using System.Threading.Tasks;

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:

```
using System;

int returnValue = int.Parse(Console.ReadLine());
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。此方法的名称实际上并不是 `Main`，而是代码无法直接引用的实现细节。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`。

IDE	MAIN
<code>await</code> 和 <code>return</code>	<code>static async Task<int> Main(string[] args)</code>
<code>await</code>	<code>static async Task Main(string[] args)</code>
<code>return</code>	<code>static int Main(string[] args)</code>
否 <code>await</code> 或 <code>return</code>	<code>static void Main(string[] args)</code>

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。

顶级语句

另请参阅

- [C# 编程指南](#)
- [方法](#)
- [在 C# 程序内部](#)

编程概念 (C#)

2020/11/2 • [Edit Online](#)

此部分介绍了 C# 语言中的编程概念。

本节内容

TITLE	介绍
.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 <code>async</code> 和 <code>await</code> 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# 语句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。
------	------------------------

使用 `Async` 和 `Await` 的异步编程

2021/3/22 • [Edit Online](#)

[基于任务的异步编程模型 \(TAP\)](#) 提供了异步代码的抽象化。你只需像往常一样将代码编写为一连串语句即可。就如每条语句在下一句开始之前完成一样，你可以流畅地阅读代码。编译器将执行许多转换，因为其中一些语句可能会开始运行并返回表示正在进行的工作的 `Task`。

这就是此语法的目标：支持读起来像一连串语句的代码，但会根据外部资源分配和任务完成时间以更复杂的顺序执行。这与人们为包含异步任务的流程给予指令的方式类似。在本文中，你将通过做早餐的指令示例来查看如何使用 `async` 和 `await` 关键字更轻松地推断包含一系列异步指令的代码。你可能会写出与以下列表类似的指令来解释如何做早餐：

1. 倒一杯咖啡。
2. 加热平底锅，然后煎两个鸡蛋。
3. 煎三片培根。
4. 烤两片面包。
5. 在烤面包上加黄油和果酱。
6. 倒一杯橙汁。

如果你有烹饪经验，便可通过异步方式执行这些指令。你会先开始加热平底锅以备煎蛋，接着再从培根着手。你可将面包放进烤面包机，然后再煎鸡蛋。在此过程的每一步，你都可以先开始一项任务，然后将注意力转移到准备进行的其他任务上。

做早餐是非并行异步工作的一个好示例。单人（或单线程）即可处理所有这些任务。继续讲解早餐的类比，一个人可以以异步方式做早餐，即在第一个任务完成之前开始进行下一个任务。不管是否有人在看着，做早餐的过程都在进行。在开始加热平底锅准备煎蛋的同时就可以开始煎了培根。在开始煎培根后，你可以将面包放进烤面包机。

对于并行算法而言，你则需要多名厨师（或线程）。一名厨师煎鸡蛋，一名厨师煎培根，依次类推。每名厨师将仅专注于一项任务。每名厨师（或线程）都在同步等待需要翻动培根或面包弹出时都将受到阻。

现在，考虑一下编写为 C# 语句的相同指令：

```
using System;
using System.Threading.Tasks;

namespace AsyncBreakfast
{
    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");

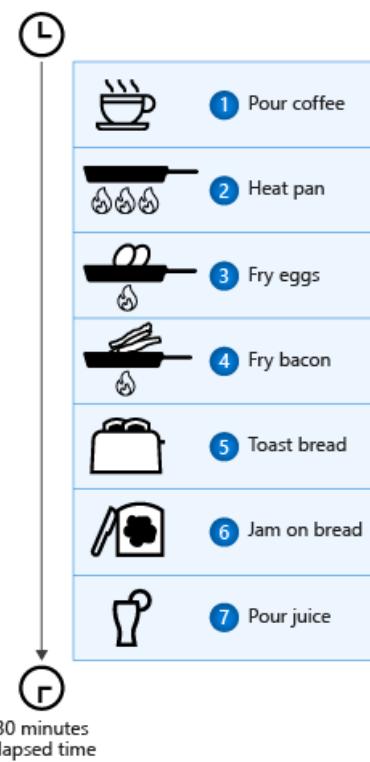
        return new Bacon();
    }

    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();
    }
}

```



同步准备的早餐大约花费了 30 分钟，因为总耗时是每个任务耗时的总和。

NOTE

`Coffee`、`Egg`、`Bacon`、`Toast` 和 `Juice` 类为空。它们是仅用于演示的简单标记类，不含任何属性，且不用于其他用途。

计算机不会按人类的方式来解释这些指令。计算机将阻塞每条语句，直到工作完成，然后再继续运行下一条语句。这将创造出令人不满意的早餐。后续任务直到早前任务完成后才会启动。这样做早餐花费的时间要得多，有些食物在上桌之前就已经凉了。

如果你希望计算机异步执行上述指令，则必须编写异步代码。

这些问题对即将编写的程序而言至关重要。编写客户端程序时，你希望 UI 能够响应用户输入。从 Web 下载数据时，你的应用程序不应让手机出现卡顿。编写服务器程序时，你不希望线程受到阻塞。这些线程可以用于处理其他请求。存在异步替代项的情况下使用同步代码会增加你进行扩展的成本。你需要为这些受阻线程付费。

成功的现代应用程序需要异步代码。在没有语言支持的情况下，编写异步代码需要回调、完成事件，或其他掩盖代码原始意图的方法。同步代码的优点是，它的分步操作使其易于扫描和理解。传统的异步模型迫使你侧重于代码的异步性质，而不是代码的基本操作。

不要阻塞，而要 await

上述代码演示了不正确的实践：构造同步代码来执行异步操作。顾名思义，此代码将阻止执行这段代码的线程执行任何其他操作。在任何任务进行过程中，此代码也不会被中断。就如同你将面包放进烤面包机后盯着此烤面包机一样。你会无视任何跟你说话的人，直到面包弹出。

我们首先更新此代码，使线程在任务运行时不会阻塞。`await` 关键字提供了一种非阻塞方式来启动任务，然后在此任务完成时继续执行。“做早餐”代码的简单异步版本类似于以下片段：

```
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

总运行时间和最初同步版本大致相同。此代码尚未利用异步编程的某些关键功能。

TIP

`FryEggsAsync`、`FryBaconAsync` 和 `ToastBreadAsync` 的方法主体都已更新，现会分别返回 `Task<Egg>`、`Task<Bacon>` 和 `Task<Toast>`。这些方法的名称与其原始版本不同，将包含“`Async`”后缀。它们的实现本文的稍后部分显示为最终版本的一部分。

在煎鸡蛋或培根时，此代码不会阻塞。不过，此代码也不会启动任何其他任务。你还是会将面包放进烤面包机里，然后盯着烤面包机直到面包弹出。但至少，你会回应任何想引起你注意的人。在接受了多份订单的一家餐馆里，厨师可能会在做第一份早餐的同时开始制作另一份早餐。

现在，在等待任何尚未完成的已启动任务时，处理早餐的线程将不会被阻塞。对于某些应用程序而言，此更改是必需的。仅凭借此更改，GUI 应用程序仍然会响应用户。然而，对于此方案而言，你需要更多的内容。你不希望每个组件任务都按顺序执行。最好首先启动每个组件任务，然后再等待之前任务的完成。

同时启动任务

在许多方案中，你希望立即启动若干独立的任务。然后，在每个任务完成时，你可以继续进行已准备的其他工作。在早餐类比中，这就是更快完成做早餐的方法。你也几乎将在同一时间完成所有工作。你将吃到一顿热气腾腾的早餐。

`System.Threading.Tasks.Task` 和相关类型是可以用于推断正在进行中的任务的类。这使你能够编写更类似于实际做早餐方式的代码。你可以同时开始煎鸡蛋、培根和烤面包。由于每个任务都需要操作，所以你会将注意力转移到那个任务上，进行下一个操作，然后等待其他需要你注意的事情。

启动一项任务并等待表示运行的 `Task` 对象。你将首先 `await` 每项任务，然后再处理它的结果。

让我们对早餐代码进行这些更改。第一步是存储任务以便在这些任务启动时进行操作，而不是等待：

```
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!");
```

接下来，可以在提供早餐之前将用于处理培根和鸡蛋的 `await` 语句移动到此方法的末尾：

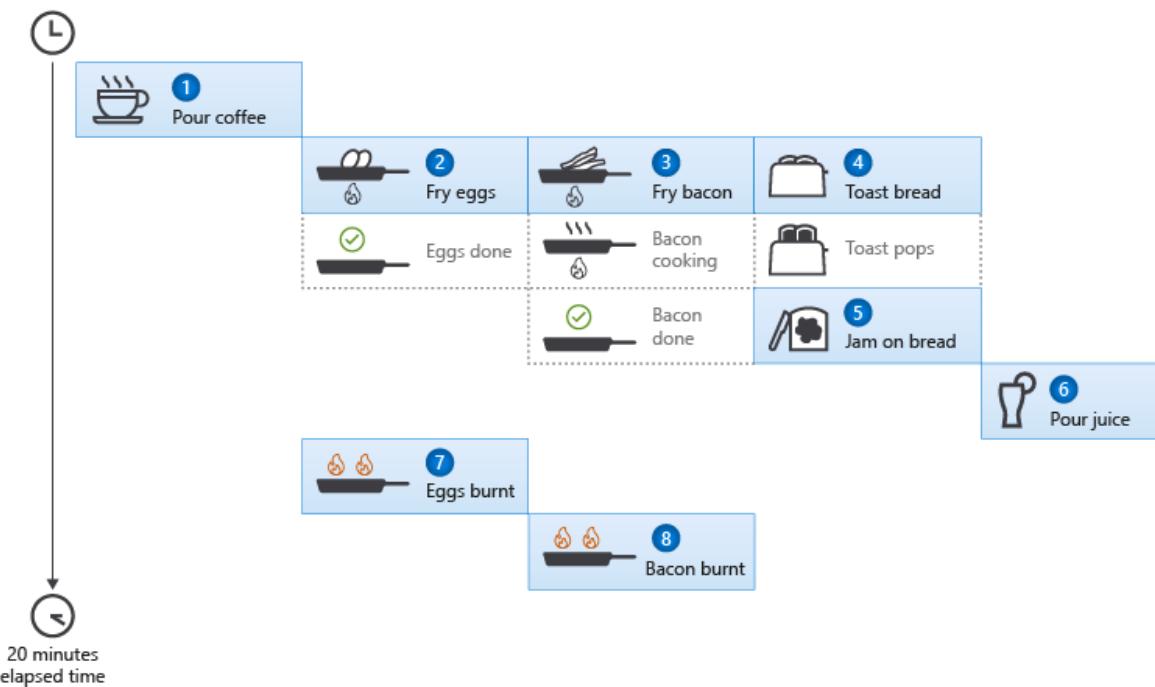
```
Coffee cup = PourCoffee();
Console.WriteLine("coffee is ready");

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!");
```



异步准备的早餐大约花费了 20 分钟，由于一些任务并发运行，因此节约了时间。

上述代码效果更好。你可以一次启动所有的异步任务。你仅在需要结果时才会等待每项任务。上述代码可能类似于 Web 应用程序中请求各种微服务，然后将结果合并到单个页面中的代码。你将立即发出所有请求，然后 `await` 所有这些任务并组成 Web 页面。

与任务组合

除了吐司外，你准备好了做早餐的所有材料。吐司制作由异步操作（烤面包）和同步操作（添加黄油和果酱）组成。更新此代码说明了一个重要的概念：

IMPORTANT

异步操作后跟同步操作的这种组合是一个异步操作。换言之，如果操作的任何部分是异步的，整个操作就是异步的。

上述代码展示了可以使用 `Task` 或 `Task<TResult>` 对象来保存运行中的任务。你首先需要 `await` 每项任务，然后再使用它的结果。下一步是创建表示其他工作组合的方式。在提供早餐之前，你希望等待表示先烤面包再添加黄油和果酱的任务完成。你可以使用以下代码表示此工作：

```
static async Task<Toast> MakeToastWithButterAndJamAsync(int number)
{
    var toast = await ToastBreadAsync(number);
    ApplyButter(toast);
    ApplyJam(toast);

    return toast;
}
```

上述方式的签名中具有 `async` 修饰符。它会向编译器发出信号，说明此方法包含 `await` 语句，也包含异步操作。此方法表示先烤面包，然后再添加黄油和果酱的任务。此方法返回表示这三个操作的组合的 `Task<TResult>`。主要代码块现在变成了：

```

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!");
}

```

上述更改说明了使用异步代码的一项重要技术。你可以通过将操作分离到一个返回任务的新方法中来组合任务。可以选择等待此任务的时间。可以同时启动其他任务。

异步异常

至此，已隐式假定所有这些任务都已成功完成。异步方法会引发异常，就像对应的同步方法一样。对异常和错误处理的异步支持通常与异步支持追求相同的目标：你应该编写读起来像一系列同步语句的代码。当任务无法成功完成时，它们将引发异常。当启动的任务为 `awaited` 时，客户端代码可捕获这些异常。例如，假设烤面包机在烤面包时着火了。可通过修改 `ToastBreadAsync` 方法来模拟这种情况，以匹配以下代码：

```

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

在编译前面的代码时，你将收到一个关于无法访问的代码的警告。这是故意的，因为一旦烤面包机着火，操作就不会正常进行。

执行这些更改后，运行应用程序，输出将类似于以下文本：

```
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)
```

请注意，从烤面包机着火到发现异常，有相当多的任务要完成。当异步运行的任务引发异常时，该任务出错。

Task 对象包含 [Task.Exception](#) 属性中引发的异常。出错的任务在等待时引发异常。

需要理解两个重要机制：异常在出错的任务中的存储方式，以及在代码等待出错的任务时解包并重新引发异常的方式。

当异步运行的代码引发异常时，该异常存储在 [Task](#) 中。[Task.Exception](#) 属性为 [System.AggregateException](#)，因为异步工作期间可能会引发多个异常。引发的任何异常都将添加到 [AggregateException.InnerExceptions](#) 集合中。如果该 [Exception](#) 属性为 NULL，则将创建一个新的 [AggregateException](#) 且引发的异常是该集合中的第一项。

对于出错的任务，最常见的情况是 [Exception](#) 属性只包含一个异常。当代码 [awaits](#) 出错的任务时，将重新引发 [AggregateException.InnerExceptions](#) 集合中的第一个异常。因此，此示例的输出显示 [InvalidOperationException](#) 而不是 [AggregateException](#)。提取第一个内部异常使得使用异步方法与使用其对应的同步方法尽可能相似。当你的场景可能生成多个异常时，可在代码中检查 [Exception](#) 属性。

继续之前，在 [ToastBreadAsync](#) 方法中注释禁止这两行。你不想再引起火灾：

```
Console.WriteLine("Fire! Toast is ruined!");
throw new InvalidOperationException("The toaster is on fire");
```

高效地等待任务

可以通过使用 [Task](#) 类的方法改进上述代码末尾的一系列 [await](#) 语句。其中一个 API 是 [WhenAll](#)，它将返回一个其参数列表中的所有任务都已完成时才完成的 [Task](#)，如以下代码中所示：

```
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!");
```

另一种选择是使用 [WhenAny](#)，它将返回一个当其参数完成时才完成的 [Task<Task>](#)。你可以等待返回的任务，了

解它已经完成了。以下代码展示了可以如何使用 `WhenAny` 等待第一个任务完成，然后再处理其结果。处理已完成任务的结果之后，可以从传递给 `whenAny` 的任务列表中删除此已完成的任务。

```
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");
    }
    breakfastTasks.Remove(finishedTask);
}
```

进行所有这些更改之后，代码的最终版本将如下所示：

```
using System;
using System.Collections.Generic;
using System.Threading.Tasks;

namespace AsyncBreakfast
{
    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");
                }
                breakfastTasks.Remove(finishedTask);
            }

            Juice oj = PourOJ();
            Console.WriteLine("oj is ready");
            Console.WriteLine("Breakfast is ready!");
        }

        static async Task<Toast> MakeToastWithButterAndJamAsync(int number)
```

```
        public void MakeToastAndJuice(int howMany)
    {
        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();
    }

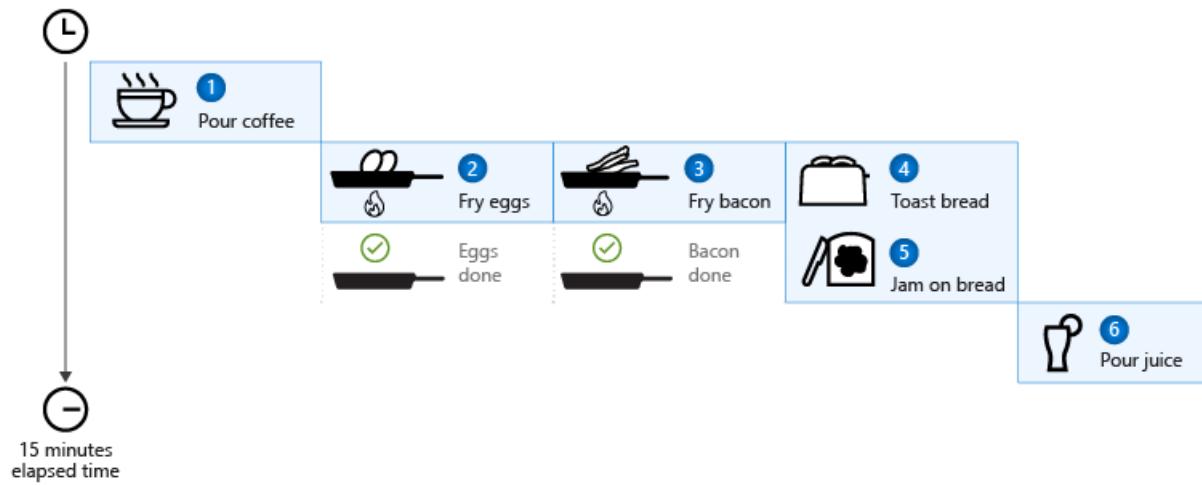
    private static async Task<Bacon> FryBaconAsync(int slices)
    {
        Console.WriteLine($"putting {slices} slices of bacon in the pan");
        Console.WriteLine("cooking first side of bacon...");
        await Task.Delay(3000);
        for (int slice = 0; slice < slices; slice++)
        {
            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();
    }
}
```



异步准备的早餐的最终版本大约花费了 15 分钟，因为一些任务并行运行，并且代码同时监视多个任务，只在需要时才执行操作。

此最终代码是异步的。它更为准确地反映了一个人做早餐的流程。将上述代码与本文中的第一个代码示例进行比较。阅读代码时，核心操作仍然很明确。你可以按照阅读本文开始时早餐制作说明的相同方式阅读此代码。

`async` 和 `await` 的语言功能支持每个人做出转变以遵循这些书面指示：尽可能启动任务，不要在等待任务完成时造成阻塞。

后续步骤

[探索异步程序的真实场景](#)

异步编程

2021/5/7 • • [Edit Online](#)

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。

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：

```
private readonly HttpClient _httpClient = new HttpClient();

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 _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。

```
private DamageResult CalculateDamageDone()
{
    // Code omitted:
    //
    // Does an expensive calculation and returns
    // the result of that calculation.
}

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

异步操作涉及许多移动部分。若要了解 `Task` 和 `Task<T>` 的内部原理，请参阅[深入了解异步](#)一文，以获取详细信息。

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。相关原因在[深入了解异步](#)中说明。

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 <https://dotnetfoundation.org> 主页下载 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。

NOTE

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。

```
private readonly HttpClient _httpClient = new HttpClient();

[HttpGet, Route("DotNetCount")]
public async Task<int> GetDotNetCount()
{
    // Suspends GetDotNetCount() to allow the caller (the web server)
    // to accept another request, rather than blocking on this one.
    var html = await _httpClient.GetStringAsync("https://dotnetfoundation.org");

    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：

```
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。

```

public 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.
}

public 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：

```

public 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.
}

public static async Task<User[]> GetUsersAsync(IEnumerable<int> userIds)
{
    var getUserTasks = userIds.Select(id => GetUserAsync(id));
    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。

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:

....
<code>await</code>	<code>Task.Wait</code> 或 <code>Task.Result</code>	检索后台任务的结果
<code>await Task.WhenAny</code>	<code>Task.WaitAny</code>	等待任何任务完成
<code>await Task.WhenAll</code>	<code>Task.WaitAll</code>	等待所有任务完成
<code>await Task.Delay</code>	<code>Thread.Sleep</code>	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 awainer。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。

其他资源

- [深入了解异步](#)提供了关于任务如何工作的详细信息。
- [基于任务的异步编程模型 \(C#\)](#)。
- 由 Lucian Wischik 所著的 [Six Essential Tips for Async](#)(关于异步的六个要点)是有关异步编程的绝佳资源。

异步编程模型

2020/11/2 • [Edit Online](#)

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# 5 引入了一种简便方法，即异步编程。此方法利用了 .NET Framework 4.5 及更高版本、.NET Core 和 Windows 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。

	.NET	WINDOWS
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) 代码示例。

```
public async Task<int> GetUrlContentLengthAsync()
{
    var client = new HttpClient();

    Task<string> getStringTask =
        client.GetStringAsync("https://docs.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。

```
string contents = await client.GetStringAsync("https://docs.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。
 - 包含 `GetAwaiter` 方法的其他任何类型(自 C# 7.0 起)。
- 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

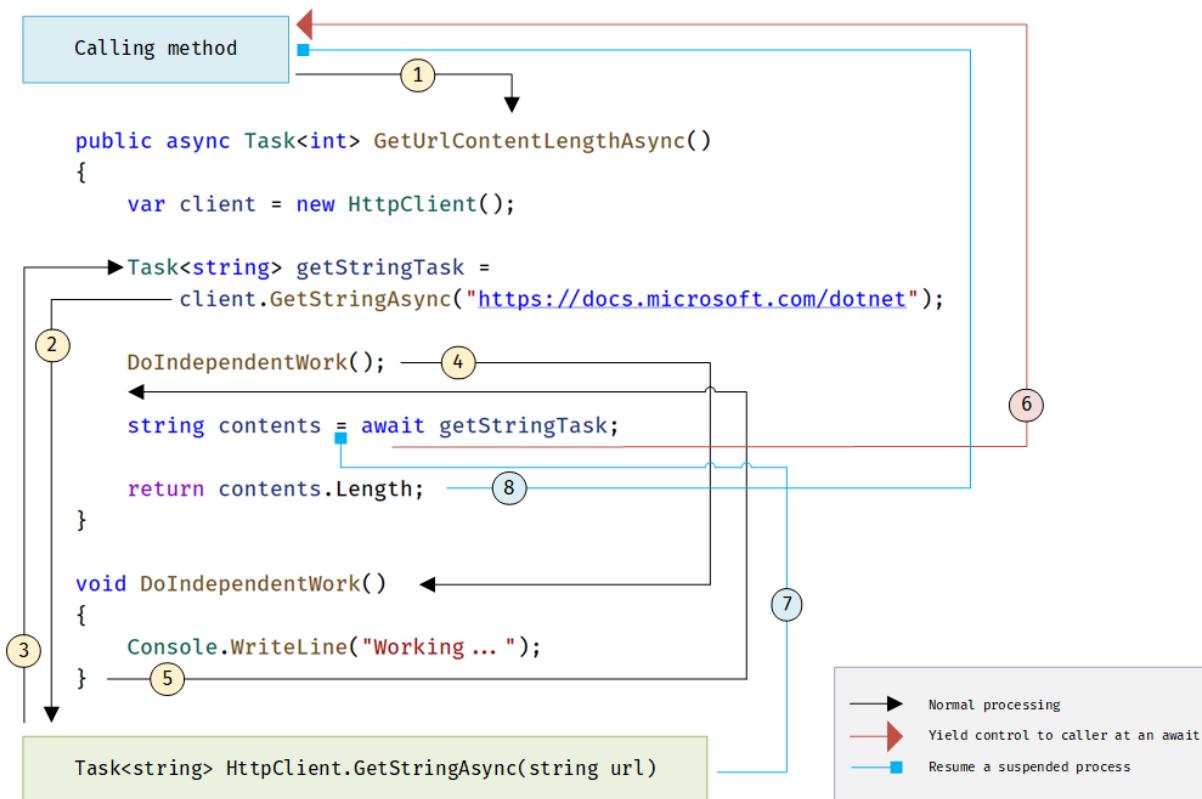
能继续。同时，将方法挂起，并且控件返回到方法的调用方。本主题的下一节将解释悬挂点发生的情况。

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。

- 调用方法调用并等待 `GetUrlContentLengthAsync` 异步方法。
- `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。
- `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。
- 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。
- `DoIndependentWork` 是完成其工作并返回其调用方的同步方法。
- `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。

NOTE

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，将 `Task` 用作返回类型。

自 C# 7.0 起，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：

```
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 \(C#\)](#)。若要详细了解如何在异步方法中捕获异常，请参阅[try-catch](#)。

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`。

相关主题和示例 (Visual Studio)

TITLE	II	II
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。	异步示例:并行发出多个 Web 请求
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： <ul style="list-style-type: none">- 取消任务列表 (C#)- 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 <code>async</code> 和 <code>await</code> 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 <code>Task</code> 和 <code>Task<TResult></code> 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

- [async](#)
- [await](#)
- [异步编程](#)
- [异步概述](#)

异步返回类型 (C#)

2021/5/7 • • [Edit Online](#)

异步方法可以具有以下返回类型：

- `Task`(对于执行操作但不返回任何值的异步方法)。
- `Task<TResult>`(对于返回值的异步方法)。
- `void`(对于事件处理程序)。
- 从 C# 7.0 开始, 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。
- 从 C# 8.0 开始, `IAsyncEnumerable<T>` 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 \(C#\)](#)。

还存在特定于 Windows 工作负载的其他几种类型：

- `DispatcherOperation`, 适用于仅限于 Windows 的异步操作。
- `IAsyncAction`, 适用于 UWP 中不返回值的异步操作。
- `IAsyncActionWithProgress<TProgress>`, 适用于 UWP 中只报告进程但不返回值的异步操作。
- `IAsyncOperation<TResult>`, 适用于 UWP 中返回值的异步操作。
- `IAsyncOperationWithProgress<TResult,TProgress>`, 适用于 UWP 中既报告进程又返回值的异步操作。

Task 返回类型

不包含 `return` 语句的异步方法或包含不返回操作数的 `return` 语句的异步方法通常具有返回类型 `Task`。如果此类方法同步运行, 它们将返回 `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。

下例中的 `WaitAndApologizeAsync` 方法不包含 `return` 语句, 因此该方法会返回 `Task` 对象。返回 `Task` 可等待 `WaitAndApologizeAsync`。`Task` 类型不包含 `Result` 属性, 因为它不具有任何返回值。

```
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。Await 运算符的应用程序在这种情况下不生成值。当 `await` 的右操作数是 `Task<TResult>` 时，`await` 表达式生成的结果为 `T`。当 `await` 的右操作数是 `Task` 时，`await` 及其操作数是一个语句。

可从 `await` 运算符的应用程序中分离对 `WaitAndApologizeAsync` 的调用，如以下代码所示。但是，请记住，`Task` 没有 `Result` 属性，并且当 `await` 运算符应用于 `Task` 时不产生值。

以下代码将调用 `WaitAndApologizeAsync` 方法和等待此方法返回的任务分离。

```
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`。

在下面的示例中，`GetLeisureHoursAsync` 方法包含返回整数的 `return` 语句。因此，该方法声明必须指定 `Task<int>` 的返回类型。`FromResult` 异步方法是返回 `DayOfWeek` 的操作的占位符。

```
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
```

在 `ShowTodaysInfo` 方法中从 `await` 表达式内调用 `GetLeisureHoursAsync` 时，`await` 表达式检索存储在由 `GetLeisureHours` 方法返回的任务中的整数值 (`leisureHours` 的值)。有关 `await` 表达式的详细信息，请参阅 [await](#)。

通过从应用程序 `await` 中分离对 `GetLeisureHoursAsync` 的调用，可以更好地理解 `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。该任务指派给示例中的 `getLeisureHoursTask` 变量。因为 `getLeisureHoursTask` 是 `Task<TResult>`，所以它包含类型 `TResult` 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。此值分配给 `ret` 变量。

IMPORTANT

Result 属性为阻止属性。如果你在其任务完成之前尝试访问它，当前处于活动状态的线程将被阻止，直到任务完成且值为可用。在大多数情况下，应通过使用 `await` 访问此值，而不是直接访问属性。

上一示例通过检索 **Result** 属性的值来阻止主线程，从而使 `Main` 方法可在应用程序结束之前将 `message` 输出到控制台。

```
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。此类方法的任何调用方都必须继续完成，而无需等待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。

返回 `void` 的异步方法的调用方无法捕获从该方法引发的异常，且此类未经处理的异常可能会导致应用程序故障。如果返回 `Task` 或 `Task<TResult>` 的方法引发异常，则该异常存储在返回的任务中。等待任务时，将重新引发异常。因此，请确保可以产生异常的任何异步方法都具有返回类型 `Task` 或 `Task<TResult>`，并确保会等待对方法的调用。

有关如何在异步方法中捕获异常的详细信息，请参阅 [try-catch](#) 文中的 [异步方法中的异常](#) 部分。

以下示例演示异步事件处理程序的行为。在本示例代码中，异步事件处理程序必须在完成时通知主线程。然后，主线程可在退出程序之前等待异步事件处理程序完成。

```
using System;
using System.Threading.Tasks;

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>

从 C# 7.0 起，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，`GetAwaiter` 方法返回的类型必须具有 `System.Runtime.CompilerServices.AsyncMethodBuilderAttribute` 特性。有关详细信息，请参阅[类似任务的返回类型](#)的功能说明。

此功能与 `awaitable 表达式` 相辅相成，后者描述 `await` 操作数的要求。编译器可以使用通用异步返回类型生成返回不同类型的 `async` 方法。通用异步返回类型通过 .NET 库实现性能改进。`Task` 和 `Task<TResult>` 是引用类型，因此，性能关键路径中的内存分配会对性能产生负面影响，尤其当分配出现在紧凑循环中时。支持通用返回类型意味着可返回轻量值类型（而不是引用类型），从而避免额外的内存分配。

.NET 提供 `System.Threading.Tasks.ValueTask<TResult>` 结构作为返回任务的通用值的轻量实现。要使用 `System.Threading.Tasks.ValueTask<TResult>` 类型，必须向项目添加 `System.Threading.Tasks.Extensions` NuGet 包。如下示例使用 `ValueTask<TResult>` 结构检索两个骰子的值。

```

using System;
using System.Threading.Tasks;

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，旨在用于非常特定的环境中。请考虑改用 `Task`、`Task<T>` 和 `ValueTask<T>` 类型，它们覆盖了大多数的异步代码方案。

使用 `IAsyncEnumerable<T>` 的异步流

从 C# 8.0 开始，异步方法可能返回异步流，由 `IAsyncEnumerable<T>` 表示。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：

```

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。读取每一行后，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。当需要从源字符串异步读取下一行时，该方法将等待。

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](#)

取消任务列表 (C#)

2021/3/5 • [Edit Online](#)

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 5.0 或更高版本的 SDK](#)
- 集成开发环境 (IDE)
 - 建议使用 [Visual Studio](#)、[Visual Studio Code](#) 或 [Visual Studio for Mac](#)

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：

```
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：

```

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://docs.microsoft.com",
    "https://docs.microsoft.com/aspnet/core",
    "https://docs.microsoft.com/azure",
    "https://docs.microsoft.com/azure/devops",
    "https://docs.microsoft.com/dotnet",
    "https://docs.microsoft.com/dynamics365",
    "https://docs.microsoft.com/education",
    "https://docs.microsoft.com/enterprise-mobility-security",
    "https://docs.microsoft.com/gaming",
    "https://docs.microsoft.com/graph",
    "https://docs.microsoft.com/microsoft-365",
    "https://docs.microsoft.com/office",
    "https://docs.microsoft.com/powershell",
    "https://docs.microsoft.com/sql",
    "https://docs.microsoft.com/surface",
    "https://docs.microsoft.com/system-center",
    "https://docs.microsoft.com/visualstudio",
    "https://docs.microsoft.com/windows",
    "https://docs.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：

```

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();

    await Task.WhenAny(new[] { cancelTask, sumPageSizesTask });

    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。

创建异步总和页面大小方法

在 `Main` 方法下，添加 `SumPageSizesAsync` 方法：

```
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($"Total bytes returned: {total:#,##}");
    Console.WriteLine($"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：

```
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：

```
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

```
Application started.  
Press the ENTER key to cancel...  
  
https://docs.microsoft.com 37,357  
https://docs.microsoft.com/aspnet/core 85,589  
https://docs.microsoft.com/azure 398,939  
https://docs.microsoft.com/azure/devops 73,663  
https://docs.microsoft.com/dotnet 67,452  
https://docs.microsoft.com/dynamics365 48,582  
https://docs.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。

```
using System;  
using System.Collections.Generic;  
using System.Diagnostics;  
using System.Net.Http;  
using System.Threading;  
using System.Threading.Tasks;  
  
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://docs.microsoft.com",  
        "https://docs.microsoft.com/aspnet/core",  
        "https://docs.microsoft.com/azure",  
        "https://docs.microsoft.com/azure/devops",  
        "https://docs.microsoft.com/dotnet",  
        "https://docs.microsoft.com/dynamics365",  
        "https://docs.microsoft.com/education",  
        "https://docs.microsoft.com/enterprise-mobility-security",  
        "https://docs.microsoft.com/gaming",  
        "https://docs.microsoft.com/graph",  
        "https://docs.microsoft.com/microsoft-365",  
        "https://docs.microsoft.com/office",  
        "https://docs.microsoft.com/powershell",  
        "https://docs.microsoft.com/sql",  
        "https://docs.microsoft.com/surface",  
        "https://docs.microsoft.com/system-center",  
        "https://docs.microsoft.com/visualstudio",  
        "https://docs.microsoft.com/windows",  
        "https://docs.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();

await Task.WhenAny(new[] { cancelTask, sumPageSizesTask });

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($"Total bytes returned: {total:#,##}");
    Console.WriteLine($"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 \(C#\)](#)

在一段时间后取消异步任务 (C#)

2021/3/5 • [Edit Online](#)

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 5.0 或更高版本的 SDK](#)
- 集成开发环境 (IDE)
 - 建议使用 Visual Studio、Visual Studio Code 或 Visual Studio for Mac

更新应用程序入口点

将现有的 `Main` 方法替换为以下内容：

```
static async Task Main()
{
    Console.WriteLine("Application started.");

    try
    {
        s_cts.CancelAfter(3500);

        await SumPageSizesAsync();
    }
    catch (TaskCanceledException)
    {
        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 `TaskCanceledException`。

示例应用程序输出

```
Application started.

https://docs.microsoft.com 37,357
https://docs.microsoft.com/aspnet/core 85,589
https://docs.microsoft.com/azure 398,939
https://docs.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。

```
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Net.Http;
using System.Threading;
using System.Threading.Tasks;

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://docs.microsoft.com",
        "https://docs.microsoft.com/aspnet/core",
        "https://docs.microsoft.com/azure",
        "https://docs.microsoft.com/azure/devops",
        "https://docs.microsoft.com/dotnet",
        "https://docs.microsoft.com/dynamics365",
        "https://docs.microsoft.com/education",
        "https://docs.microsoft.com/enterprise-mobility-security",
        "https://docs.microsoft.com/gaming",
        "https://docs.microsoft.com/graph",
        "https://docs.microsoft.com/microsoft-365",
        "https://docs.microsoft.com/office",
        "https://docs.microsoft.com/powershell",
        "https://docs.microsoft.com/sql",
        "https://docs.microsoft.com/surface",
        "https://docs.microsoft.com/system-center",
        "https://docs.microsoft.com/visualstudio",
        "https://docs.microsoft.com/windows",
        "https://docs.microsoft.com/xamarin"
    };
}

static async Task Main()
{
    Console.WriteLine("Application started.");

    try
    {
        s_cts.CancelAfter(3500);

        await SumPageSizesAsync();
    }
    catch (TaskCanceledException)
```

```

    {
        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 \(C#\)](#)

在异步任务完成时对其进行处理 (C#)

2021/5/7 • [Edit Online](#)

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。

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:

```
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Linq;
using System.Net.Http;
using System.Threading.Tasks;
```

添加字段

在 `Program` 类定义中, 添加以下两个字段:

```
static readonly HttpClient s_client = new HttpClient
{
    MaxResponseContentBufferSize = 1_000_000
};

static readonly IEnumerable<string> s_urlList = new string[]
{
    "https://docs.microsoft.com",
    "https://docs.microsoft.com/aspnet/core",
    "https://docs.microsoft.com/azure",
    "https://docs.microsoft.com/azure/devops",
    "https://docs.microsoft.com/dotnet",
    "https://docs.microsoft.com/dynamics365",
    "https://docs.microsoft.com/education",
    "https://docs.microsoft.com/enterprise-mobility-security",
    "https://docs.microsoft.com/gaming",
    "https://docs.microsoft.com/graph",
    "https://docs.microsoft.com/microsoft-365",
    "https://docs.microsoft.com/office",
    "https://docs.microsoft.com/powershell",
    "https://docs.microsoft.com/sql",
    "https://docs.microsoft.com/surface",
    "https://docs.microsoft.com/system-center",
    "https://docs.microsoft.com/visualstudio",
    "https://docs.microsoft.com/windows",
    "https://docs.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：

```
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：

```

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($"Total bytes returned: {total:#,##}");
    Console.WriteLine($"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：

```

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。

```
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：

- 等待调用 `WhenAny`，以标识集合中首个已完成下载的任务。

```
Task<int> finishedTask = await Task.WhenAny(downloadTasks);
```

- 从集合中移除任务。

```
downloadTasks.Remove(finishedTask);
```

- 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。

```
total += await finishedTask;
```

添加进程方法

在 `SumPageSizesAsync` 方法下添加以下 `ProcessUrlAsync` 方法：

```
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。

Caution

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。

```
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Linq;
using System.Net.Http;
using System.Threading.Tasks;

namespace ProcessTasksAsTheyFinish
{
    class Program
    {
        static readonly HttpClient s_client = new HttpClient
        {
            MaxResponseContentBufferSize = 1_000_000
        };

        static readonly IEnumerable<string> s_urlList = new string[]
        {
            "https://docs.microsoft.com",
            "https://docs.microsoft.com/aspnet/core",
            "https://docs.microsoft.com/azure",
            "https://docs.microsoft.com/azure/devops",
            "https://docs.microsoft.com/dotnet",
            "https://docs.microsoft.com/dynamics365",
            "https://docs.microsoft.com/education",
            "https://docs.microsoft.com/enterprise-mobility-security",
            "https://docs.microsoft.com/gaming",
            "https://docs.microsoft.com/graph",
            "https://docs.microsoft.com/microsoft-365",
            "https://docs.microsoft.com/office",
            "https://docs.microsoft.com/powershell",
            "https://docs.microsoft.com/sql",
            "https://docs.microsoft.com/surface",
            "https://docs.microsoft.com/system-center",
            "https://docs.microsoft.com/visualstudio",
            "https://docs.microsoft.com/windows",
            "https://docs.microsoft.com/xamarin"
        };
    }

    static Task Main() => SumPageSizesAsync();

    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");
}

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://docs.microsoft.com/windows 25,513
// https://docs.microsoft.com/gaming 30,705
// https://docs.microsoft.com/dotnet 69,626
// https://docs.microsoft.com/dynamics365 50,756
// https://docs.microsoft.com/surface 35,519
// https://docs.microsoft.com 39,531
// https://docs.microsoft.com/azure/devops 75,837
// https://docs.microsoft.com/xamarin 60,284
// https://docs.microsoft.com/system-center 43,444
// https://docs.microsoft.com/enterprise-mobility-security 28,946
// https://docs.microsoft.com/microsoft-365 43,278
// https://docs.microsoft.com/visualstudio 31,414
// https://docs.microsoft.com/office 42,292
// https://docs.microsoft.com/azure 401,113
// https://docs.microsoft.com/graph 46,831
// https://docs.microsoft.com/education 25,098
// https://docs.microsoft.com/powershell 58,173
// https://docs.microsoft.com/aspnet/core 87,763
// https://docs.microsoft.com/sql 53,362

// Total bytes returned: 1,249,485
// Elapsed time: 00:00:02.7068725

```

另请参阅

- [WhenAny](#)
- [使用 Async 和 Await 的异步编程 \(C#\)](#)

异步文件访问 (C#)

2021/5/7 • • [Edit Online](#)

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 操作进行有限的控制，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两者配合使用。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

```
public async Task SimpleWriteAsync()
{
    string filePath = "simple.txt";
    string text = $"Hello World";

    await File.WriteAllTextAsync(filePath, text);
}
```

有限控制示例

```
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：

```
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

```
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。`ReadAsStringAsync` 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 \(C#\)](#)。

```

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

```

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。

```
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 \(C#\)](#)

特性 (C#)

2021/5/7 • • [Edit Online](#)

使用特性，可以有效地将元数据或声明性信息与代码(程序集、类型、方法、属性等)相关联。将特性与程序实体相关联后，可以在运行时使用 反射 这项技术查询特性。有关详细信息，请参阅[反射 \(C#\)](#)。

特性具有以下属性：

- 特性向程序添加元数据。[元数据](#) 是程序中定义的类型的相关信息。所有 .NET 程序集都包含一组指定的元数据，用于描述程序集中定义的类型和类型成员。可以添加自定义特性来指定所需的其他任何信息。有关详细信息，请参阅[创建自定义特性 \(C#\)](#)。
- 可以将一个或多个特性应用于整个程序集、模块或较小的程序元素(如类和属性)。
- 特性可以像方法和属性一样接受自变量。
- 程序可使用反射来检查自己的元数据或其他程序中的元数据。有关详细信息，请参阅[使用反射访问特性 \(C#\)](#)。

使用特性

可以将特性附加到几乎任何声明中，尽管特定特性可能会限制可有效附加到的声明的类型。在 C# 中，通过用方括号 ([]) 将特性名称括起来，并置于应用该特性的实体的声明上方以指定特性。

在此示例中，[SerializableAttribute](#) 特性用于将具体特征应用于类：

```
[Serializable]
public class SampleClass
{
    // Objects of this type can be serialized.
}
```

下方示例声明了一个具有特性 [DllImportAttribute](#) 的方法：

```
[System.Runtime.InteropServices.DllImport("user32.dll")]
extern static void SampleMethod();
```

如下方示例所示，可以将多个特性附加到声明中：

```
using System.Runtime.InteropServices;
```

```
void MethodA([In][Out] ref double x) { }
void MethodB([Out][In] ref double x) { }
void MethodC([In, Out] ref double x) { }
```

对于给定实体，一些特性可以指定多次。[ConditionalAttribute](#) 便属于此类多用途特性：

```
[Conditional("DEBUG"), Conditional("TEST1")]
void TraceMethod()
{
    // ...
}
```

NOTE

按照约定，所有特性名称均以“Attribute”一词结尾，以便与 .NET 库中的其他项区分开来。不过，在代码中使用特性时，无需指定特性后缀。例如，`[DllImport]` 等同于 `[DllImportAttribute]`，但 `DllImportAttribute` 是此特性在 .NET 类库中的实际名称。

特性参数

许多属性都有参数，可以是位置参数、未命名参数或已命名参数。必须以特定顺序指定任何位置参数，且不能省略。已命名参数是可选参数，可以通过任何顺序指定。首先指定的是位置参数。例如，下面这三个特性是等同的：

```
[DllImport("user32.dll")]
[DllImport("user32.dll", SetLastError=false, ExactSpelling=false)]
[DllImport("user32.dll", ExactSpelling=false, SetLastError=false)]
```

第一个参数(DLL 名称)是位置参数，始终第一个出现；其他是已命名参数。在此示例中，两个已命名参数的默认值均为 `false`，因此可以省略。位置参数与特性构造函数的参数相对应。已命名或可选参数与该特性的属性或字段相对应。若要了解默认参数值，请参阅各个特性文档。

有关允许的参数类型的详细信息，请参阅 [C# 语言规范](#) 中的属性部分。

特性目标

特性目标是指应用特性的实体。例如，特性可应用于类、特定方法或整个程序集。默认情况下，特性应用于紧跟在它后面的元素。不过，还可以进行显式标识。例如，可以标识为将特性应用于方法，还是应用于其参数或返回值。

若要显式标识特性目标，请使用以下语法：

```
[target : attribute-list]
```

下表列出了可能的 `target` 值。

目标	说明
<code>assembly</code>	整个程序集
<code>module</code>	当前程序集模块
<code>field</code>	类或结构中的字段
<code>event</code>	事件
<code>method</code>	方法或 <code>get</code> 和 <code>set</code> 属性访问器
<code>param</code>	方法参数或 <code>set</code> 属性访问器参数
<code>property</code>	Property
<code>return</code>	方法、属性索引器或 <code>get</code> 属性访问器的返回值
<code>type</code>	结构、类、接口、枚举或委托

指定 `field` 目标值，将特性应用到为自动实现的属性创建的支持字段。

下面的示例展示了如何将特性应用于程序集和模块。有关详细信息，请参阅[通用特性 \(C#\)](#)。

```
using System;
using System.Reflection;
[assembly: AssemblyTitle("Production assembly 4")]
[module: CLSCompliant(true)]
```

以下示例演示如何将特性应用于 C# 中的方法、方法参数和方法返回值。

```
// 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

无论在哪个目标上将 `ValidatedContract` 定义为有效，都必须指定 `return` 目标，即使 `ValidatedContract` 定义为仅应用于返回值也是如此。换言之，编译器不会使用 `AttributeUsage` 信息来解析不明确的特性目标。有关详细信息，请参阅[AttributeUsage \(C#\)](#)。

特性的常见用途

下面列出了代码中特性的一些常见用途：

- 在 Web 服务中使用 `WebMethod` 特性标记方法，以指明方法应可通过 SOAP 协议进行调用。有关详细信息，请参阅[WebMethodAttribute](#)。
- 描述在与本机代码互操作时如何封送方法参数。有关详细信息，请参阅[MarshalAsAttribute](#)。
- 描述类、方法和接口的 COM 属性。
- 使用 `DllImportAttribute` 类调用非托管代码。
- 从标题、版本、说明或商标方面描述程序集。
- 描述要序列化并暂留类的哪些成员。
- 描述如何为了执行 XML 序列化在类成员和 XML 节点之间进行映射。
- 描述的方法的安全要求。
- 指定用于强制实施安全规范的特征。
- 通过实时 (JIT) 编译器控制优化，这样代码就一直都易于调试。
- 获取方法调用方的相关信息。

相关章节

有关详细信息，请参见：

- [创建自定义特性 \(C#\)](#)

- [使用反射访问特性 \(C#\)](#)
- [如何使用特性创建 C/C++ 联合 \(C#\)](#)
- [通用特性 \(C#\)](#)
- [调用方信息 \(C#\)](#)

请参阅

- [C# 编程指南](#)
- [反射 \(C#\)](#)
- [特性](#)
- [在 C# 中使用特性](#)

创建自定义特性 (C#)

2020/11/2 • [Edit Online](#)

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：

```
[System.AttributeUsage(System.AttributeTargets.Class |
                      System.AttributeTargets.Struct)
]
public class AuthorAttribute : System.Attribute
{
    private string name;
    public double version;

    public AuthorAttribute(string name)
    {
        this.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：

```
[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。

```
[System.AttributeUsage(System.AttributeTargets.Class |
                      System.AttributeTargets.Struct,
                      AllowMultiple = true) // multiuse attribute
]
public class AuthorAttribute : System.Attribute
```

在下面的代码示例中，某个类应用了同一类型的多个特性。

```
[Author("P. Ackerman", version = 1.1)]
[Author("R. Koch", version = 1.2)]
class SampleClass
{
    // P. Ackerman's code goes here...
    // R. Koch's code goes here...
}
```

请参阅

- [System.Reflection](#)
- [C# 编程指南](#)
- [编写自定义特性](#)
- [反射 \(C#\)](#)
- [特性 \(C#\)](#)
- [使用反射访问特性 \(C#\)](#)
- [AttributeUsage \(C#\)](#)

使用反射访问特性 (C#)

2020/11/2 • [Edit Online](#)

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。此方法有多个重载版本。有关详细信息，请参阅 [Attribute](#)。

特性规范，例如：

```
[Author("P. Ackerman", version = 1.1)]
class SampleClass
```

在概念上等效于此：

```
Author anonymousAuthorObject = new Author("P. Ackerman");
anonymousAuthorObject.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` 会导致按上述方式构造并初始化一个 `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。

示例

此处是一个完整的示例。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。

```
// Multiuse attribute.
[System.AttributeUsage(System.AttributeTargets.Class |
    System.AttributeTargets.Struct,
    AllowMultiple = true) // Multiuse attribute.
]
public class Author : System.Attribute
{
    string name;
    public double version;

    public Author(string name)
    {
        this.name = name;

        // Default value.
        version = 1.0;
    }

    public string GetName()
    {
        return 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
{
    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 {0}", t);

        // Using reflection.
        System.Attribute[] attrs = System.Attribute.GetCustomAttributes(t); // Reflection.

        // Displaying output.
        foreach (System.Attribute attr in attrs)
        {
            if (attr is Author)
            {
                Author a = (Author)attr;
                System.Console.WriteLine("  {0}, version {1:f}", a.GetName(), a.version);
            }
        }
    }
}
/* 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
*/

```

请参阅

- [System.Reflection](#)
- [Attribute](#)
- [C# 编程指南](#)
- [检索存储在特性中的信息](#)
- [反射 \(C#\)](#)
- [特性 \(C#\)](#)
- [创建自定义特性 \(C#\)](#)

如何使用特性创建 C/C++ 联合 (C#)

2021/5/7 • [Edit Online](#)

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。

示例

在此代码段中，`TestUnion` 的所有字段均从内存中的同一位置开始。

```
// Add a using directive for System.Runtime.InteropServices.  
  
[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;  
}
```

下面是另一个示例，其中的字段从不同的显式设置位置开始。

```
// Add a using directive for System.Runtime.InteropServices.  
  
[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;  
}
```

两个整数字段 `i1` 和 `i2` 共享与 `lg` 相同的内存位置。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](#)
- [C# 编程指南](#)
- [特性](#)
- [反射 \(C#\)](#)
- [特性 \(C#\)](#)
- [创建自定义特性 \(C#\)](#)
- [使用反射访问特性 \(C#\)](#)

集合 (C#)

2020/11/2 • [Edit Online](#)

对于许多应用程序，你会想要创建和管理相关对象的组。有两种方法对对象进行分组：通过创建对象的数组，以及通过创建对象的集合。

数组最适用于创建和使用固定数量的强类型化对象。有关数组的信息，请参阅[数组](#)。

集合提供更灵活的方式来使用对象组。与数组不同，你使用的对象组随着应用程序更改的需要动态地放大和缩小。对于某些集合，你可以为放入集合中的任何对象分配一个密钥，这样你便可以使用该密钥快速检索此对象。

集合是一个类，因此必须在向该集合添加元素之前，声明类的实例。

如果集合中只包含一种数据类型的元素，则可以使用 [System.Collections.Generic](#) 命名空间中的一个类。泛型集合强制类型安全，因此无法向其添加任何其他数据类型。当你从泛型集合检索元素时，你无需确定其数据类型或对其进行转换。

NOTE

在本主题的示例中，针对 `System.Collections.Generic` 和 `System.Linq` 命名空间包括 `using` 指令。

在本主题中

- [使用简单集合](#)
- [集合的类型](#)
 - [System.Collections.Generic](#) 类
 - [System.Collections.Concurrent](#) 类
 - [System.Collections](#) 类
- [实现键/值对集合](#)
- [使用 LINQ 访问集合](#)
- [对集合排序](#)
- [定义自定义集合](#)
- [迭代器](#)

使用简单集合

本部分中的示例使用泛型 `List<T>` 类，通过此类可使用对象的强类型列表。

以下示例创建字符串列表，并通过使用 `foreach` 语句循环访问字符串。

```
// Create a list of strings.  
var salmons = new List<string>();  
salmons.Add("chinook");  
salmons.Add("coho");  
salmons.Add("pink");  
salmons.Add("sockeye");  
  
// Iterate through the list.  
foreach (var salmon in salmons)  
{  
    Console.WriteLine(salmon + " ");  
}  
// Output: chinook coho pink sockeye
```

如果集合中的内容是事先已知的，则可以使用集合初始值设定项来初始化集合。有关详细信息，请参阅[对象和集合初始值设定项](#)。

以下示例与上一示例相同，除了有一个集合初始值设定项用于将元素添加到集合。

```
// Create a list of strings by using a  
// collection initializer.  
var salmons = new List<string> { "chinook", "coho", "pink", "sockeye" };  
  
// Iterate through the list.  
foreach (var salmon in salmons)  
{  
    Console.WriteLine(salmon + " ");  
}  
// Output: chinook coho pink sockeye
```

可以使用 `for` 语句，而不是 `foreach` 语句来循环访问集合。通过按索引位置访问集合元素实现此目的。元素的索引开始于 0，结束于元素计数减 1。

以下示例通过使用 `for` 而不是 `foreach` 循环访问集合中的元素。

```
// Create a list of strings by using a  
// collection initializer.  
var salmons = new List<string> { "chinook", "coho", "pink", "sockeye" };  
  
for (var index = 0; index < salmons.Count; index++)  
{  
    Console.WriteLine(salmons[index] + " ");  
}  
// Output: chinook coho pink sockeye
```

以下示例通过指定要删除的对象，从集合中删除一个元素。

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

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

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

以下示例从一个泛型列表中删除元素。使用以降序进行循环访问的 `for` 语句，而非 `foreach` 语句。这是因为 `RemoveAt` 方法将导致已移除的元素后的元素的索引值减小。

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

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

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

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

```

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

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

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

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

```

集合的类型

许多通用集合由 .NET 提供。每种类型的集合用于特定的用途。

本部分介绍了一些通用集合类：

- [System.Collections.Generic](#) 类
- [System.Collections.Concurrent](#) 类
- [System.Collections](#) 类

[System.Collections.Generic](#) 类

可以使用 [System.Collections.Generic](#) 命名空间中的某个类来创建泛型集合。当集合中的所有项都具有相同的数据类型时，泛型集合会非常有用。泛型集合通过仅允许添加所需的数据类型，强制实施强类型化。

下表列出了 [System.Collections.Generic](#) 命名空间中的一些常用类：

I	II
Dictionary< TKey, TValue >	表示基于键进行组织的键/值对的集合。
List< T >	表示可按索引访问的对象的列表。提供用于对列表进行搜索、排序和修改的方法。
Queue< T >	表示对象的先进先出 (FIFO) 集合。
SortedList< TKey, TValue >	表示基于相关的 IComparer< T > 实现按键进行排序的键/值对的集合。
Stack< T >	表示对象的后进先出 (LIFO) 集合。

有关其他信息, 请参阅[常用集合类型](#)、[选择集合类](#)和[System.Collections.Generic](#)。

System.Collections.Concurrent 类

在 .NET Framework 4 以及更新的版本中, [System.Collections.Concurrent](#) 命名空间中的集合可提供高效的线程安全操作, 以便从多个线程访问集合项。

只要多个线程同时访问集合, 就应使用 [System.Collections.Concurrent](#) 命名空间中的类, 而不是 [System.Collections.Generic](#) 和 [System.Collections](#) 命名空间中的相应类型。有关详细信息, 请参阅[线程安全集合](#)和[System.Collections.Concurrent](#)。

包含在 [System.Collections.Concurrent](#) 命名空间中的一些类为

[BlockingCollection](#)<T>、[ConcurrentDictionary](#)< TKey, TValue >、[ConcurrentQueue](#)< T > 和 [ConcurrentStack](#)< T >。

System.Collections 类

[System.Collections](#) 命名空间中的类不会将元素作为特别类型化的对象存储, 而是作为 [Object](#) 类型的对象存储。

只要可能, 则应使用 [System.Collections.Generic](#) 命名空间或 [System.Collections.Concurrent](#) 命名空间中的泛型集合, 而不是 [System.Collections](#) 命名空间中的旧类型。

下表列出了 [System.Collections](#) 命名空间中的一些常用类:

ArrayList	表示对象的数组, 这些对象的大小会根据需要动态增加。
Hashtable	表示根据键的哈希代码进行组织的键/值对的集合。
Queue	表示对象的先进先出 (FIFO) 集合。
Stack	表示对象的后进先出 (LIFO) 集合。

[System.Collections.Specialized](#) 命名空间提供专门类型化以及强类型化的集合类, 例如只包含字符串的集合以及链接列表和混合字典。

实现键/值对集合

[Dictionary](#)< TKey, TValue > 泛型集合可通过每个元素的键访问集合中的元素。每次对字典的添加都包含一个值和与其关联的键。通过使用键来检索值十分快捷, 因为 [Dictionary](#) 类实现为哈希表。

以下示例创建 [Dictionary](#) 集合并通过使用 [foreach](#) 语句循环访问字典。

```

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

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

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

private static Dictionary<string, Element> BuildDictionary()
{
    var elements = new Dictionary<string, Element>();

    AddToDictionary(elements, "K", "Potassium", 19);
    AddToDictionary(elements, "Ca", "Calcium", 20);
    AddToDictionary(elements, "Sc", "Scandium", 21);
    AddToDictionary(elements, "Ti", "Titanium", 22);

    return elements;
}

private static void AddToDictionary(Dictionary<string, Element> elements,
    string symbol, string name, int atomicNumber)
{
    Element theElement = new Element();

    theElement.Symbol = symbol;
    theElement.Name = name;
    theElement.AtomicNumber = atomicNumber;

    elements.Add(key: theElement.Symbol, value: theElement);
}

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

```

若要转而使用集合初始值设定项生成 `Dictionary` 集合，可使用以下方法替换 `BuildDictionary` 和 `AddToDictionary`。

```

private static Dictionary<string, Element> BuildDictionary2()
{
    return new Dictionary<string, Element>
    {
        {"K",
            new Element() { Symbol="K", Name="Potassium", AtomicNumber=19}},
        {"Ca",
            new Element() { Symbol="Ca", Name="Calcium", AtomicNumber=20}},
        {"Sc",
            new Element() { Symbol="Sc", Name="Scandium", AtomicNumber=21}},
        {"Ti",
            new Element() { Symbol="Ti", Name="Titanium", AtomicNumber=22}}
    };
}

```

以下示例使用 `ContainsKey` 方法和 `Dictionary` 的 `Item[]` 属性按键快速查找某个项。使用 `Item` 属性可通过 C#

中的 `elements[symbol]` 来访问 `elements` 集合中的项。

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

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

以下示例则使用 [TryGetValue](#) 方法按键快速查找某个项。

```
private static void FindInDictionary2(string symbol)
{
    Dictionary<string, Element> elements = BuildDictionary();

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

使用 LINQ 访问集合

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

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

```

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

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

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

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

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

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

```

对集合排序

以下示例阐释了对集合排序的过程。该示例对 `List<T>` 中存储的 `Car` 类的实例进行排序。`Car` 类实现 `IComparable<T>` 接口，此操作需要实现 `CompareTo` 方法。

每次对 `CompareTo` 方法的调用均会执行用于排序的单一比较。`CompareTo` 方法中用户编写的代码针对当前对象与另一个对象的每个比较返回一个值。如果当前对象小于另一个对象，则返回的值小于零；如果当前对象大于另一个对象，则返回的值大于零；如果当前对象等于另一个对象，则返回的值等于零。这使你可以在代码中定义大于、小于和等于条件。

在 `ListCars` 方法中，`cars.Sort()` 语句对列表进行排序。对 `List<T>` 的 `Sort` 方法的此调用将导致为 `List` 中的 `Car` 对象自动调用 `CompareTo` 方法。

```

private static void ListCars()
{
    var cars = new List<Car>
    {
        { new Car() { Name = "car1", Color = "blue", Speed = 20} },
        { new Car() { Name = "car2", Color = "red", Speed = 50} },
        { new Car() { Name = "car3", Color = "green", Speed = 10} },
        { new Car() { Name = "car4", Color = "blue", Speed = 50} },
        { new Car() { Name = "car5", Color = "blue", Speed = 30} },
        { new Car() { Name = "car6", Color = "red", Speed = 60} },
        { new Car() { Name = "car7", Color = "green", Speed = 40} }
    };
}

```

```

    {
        new Car() { Name = "car1", Color = "green", Speed = 50};

    };

    // Sort the cars by color alphabetically, and then by speed
    // in descending order.
    cars.Sort();

    // View all of the cars.
    foreach (Car thisCar in cars)
    {
        Console.Write(thisCar.Color.PadRight(5) + " ");
        Console.Write(thisCar.Speed.ToString() + " ");
        Console.WriteLine(thisCar.Name);
        Console.WriteLine();
    }

    // Output:
    // blue 50 car4
    // blue 30 car5
    // blue 20 car1
    // green 50 car7
    // green 10 car3
    // red 60 car6
    // red 50 car2
}

public class Car : IComparable<Car>
{
    public string Name { get; set; }
    public int Speed { get; set; }
    public string Color { get; set; }

    public int CompareTo(Car other)
    {
        // A call to this method makes a single comparison that is
        // used for sorting.

        // Determine the relative order of the objects being compared.
        // Sort by color alphabetically, and then by speed in
        // descending order.

        // Compare the colors.
        int compare;
        compare = String.Compare(this.Color, other.Color, true);

        // If the colors are the same, compare the speeds.
        if (compare == 0)
        {
            compare = this.Speed.CompareTo(other.Speed);

            // Use descending order for speed.
            compare = -compare;
        }

        return compare;
    }
}

```

定义自定义集合

可以通过实现 [IEnumerable<T>](#) 或 [IEnumerator](#) 接口来定义集合。

尽管可以定义自定义集合，但通常最好使用包含在 .NET 中的集合，这在本文前面的[集合类型](#)中进行了介绍。

以下示例定义一个名为 `AllColors` 的自定义集合类。此类实现 [IEnumerable](#) 接口，此操作需要实现 [GetEnumerator](#) 方法。

`GetEnumerator` 方法返回 `ColorEnumerator` 类的一个实例。 `ColorEnumerator` 实现 `IEnumerator` 接口，此操作需要实现 `Current` 属性、`MoveNext` 方法以及 `Reset` 方法。

```
private static void ListColors()
{
    var colors = new AllColors();

    foreach (Color theColor in colors)
    {
        Console.WriteLine(theColor.Name + " ");
    }
    Console.WriteLine();
    // Output: red blue green
}

// Collection class.
public class AllColors : System.Collections.IEnumerable
{
    Color[] _colors =
    {
        new Color() { Name = "red" },
        new Color() { Name = "blue" },
        new Color() { Name = "green" }
    };

    public System.Collections.IEnumerator GetEnumerator()
    {
        return new ColorEnumerator(_colors);

        // Instead of creating a custom enumerator, you could
        // use the GetEnumerator of the array.
        //return _colors.GetEnumerator();
    }

    // Custom enumerator.
    private class ColorEnumerator : System.Collections.IEnumerator
    {
        private Color[] _colors;
        private int _position = -1;

        public ColorEnumerator(Color[] colors)
        {
            _colors = colors;
        }

        object System.Collections.IEnumerator.Current
        {
            get
            {
                return _colors[_position];
            }
        }

        bool System.Collections.IEnumerator.MoveNext()
        {
            _position++;
            return (_position < _colors.Length);
        }

        void System.Collections.IEnumerator.Reset()
        {
            _position = -1;
        }
    }
}

// Element class.
```

```
public class Color
{
    public string Name { get; set; }
}
```

迭代器

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

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

有关详细信息，请参阅[迭代器 \(C#\)](#)。

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

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

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

请参阅

- [对象和集合初始值设定项](#)
- [编程概念 \(C#\)](#)
- [Option Strict 语句](#)
- [LINQ to Objects \(C#\)](#)
- [并行 LINQ \(PLINQ\)](#)
- [集合和数据结构](#)
- [选择集合类](#)
- [集合内的比较和排序](#)
- [何时使用泛型集合](#)

协变和逆变 (C#)

2020/11/2 • [Edit Online](#)

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。

```
// 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。

```
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。

```

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。

```

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

TITLE	II
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 <code>IEnumerable<T></code> 接口和 <code>IComparable<T></code> 接口中对协变和逆变的支持如何帮助重复使用代码。
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 <code>Func</code> 委托和 <code>Action</code> 委托中对协变和逆变的支持如何帮助重复使用代码。

泛型接口中的变体 (C#)

2020/11/2 • [Edit Online](#)

.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。

```
 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。

```

// 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。

```

IQueryable<int> integers = new List<int>();
// The following statement generates a compiler error,
// because int is a value type.
// IQueryable<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。

```

// The following line generates a compiler error
// because classes are invariant.
// List<Object> list = new List<String>();

// You can use the interface object instead.
IQueryable<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#)

2020/11/2 • [Edit Online](#)

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”。

NOTE

.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。

IMPORTANT

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。

```
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 \(C#\)](#)。

```
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。

```
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。

```
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。

```
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。

```
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。

```
// 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。

```
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。

```
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。

```
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。

```
// Simple class hierarchy.  
class Animal { }  
class Cat : Animal { }  
class Dog : Animal { }  
  
// This class introduces ambiguity  
// because I Enumerable<out T> is covariant.  
class Pets : I Enumerable<Cat>, I Enumerable<Dog>  
{  
    I Enumerator<Cat> I Enumerable<Cat>. GetEnumerator()  
    {  
        Console.WriteLine("Cat");  
        // Some code.  
        return null;  
    }  
  
    I Enumerator I Enumerable. GetEnumerator()  
    {  
        // Some code.  
        return null;  
    }  
  
    I Enumerator<Dog> I Enumerable<Dog>. GetEnumerator()  
    {  
        Console.WriteLine("Dog");  
        // Some code.  
        return null;  
    }  
}  
class Program  
{  
    public static void Test()  
    {  
        I Enumerable<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#)

2021/5/7 • [Edit Online](#)

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`。

```
// 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`。

```

// 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#)

2020/11/2 • [Edit Online](#)

.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。

```
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。

```
// 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。

```
// 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。

```
// 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。

```
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。

```
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。

```
public delegate void DContravariant<in A>(A a);
```

IMPORTANT

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。

```
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。

```
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。

```
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。

```
// 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#)

2020/11/2 • [Edit Online](#)

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

```
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 `Button.KeyDown` 事件签名的 `KeyEventHandler` 委托。其签名为：

```
public delegate void KeyEventHandler(object sender, KeyEventArgs e)
```

- 定义 `Button.MouseClick` 事件签名的 `MouseEventHandler` 委托。其签名为：

```
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

```
// 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#)

2020/11/2 • [Edit Online](#)

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`。

```
// 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`。

```
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](#)

表达式树 (C#)

2020/11/2 • [Edit Online](#)

表达式树以树形数据结构表示代码，其中每一个节点都是一种表达式，比如方法调用和 `x < y` 这样的二元运算等。

你可以对表达式树中的代码进行编辑和运算。这样能够动态修改可执行代码、在不同数据库中执行 LINQ 查询以及创建动态查询。有关 LINQ 中表达式树的详细信息，请参阅[如何使用表达式树生成动态查询 \(C#\)](#)。

表达式树还能用于动态语言运行时 (DLR) 以提供动态语言和 .NET 之间的互操作性，同时保证编译器编写员能够发射表达式树而非 Microsoft 中间语言 (MSIL)。有关 DLR 的详细信息，请参阅[动态语言运行时概述](#)。

你可以基于匿名 lambda 表达式通过 C# 或者 Visual Basic 编译器创建表达式树，或者通过 `System.Linq.Expressions` 名称空间手动创建。

根据 Lambda 表达式创建表达式树

若 lambda 表达式被分配给 `Expression<TDelegate>` 类型的变量，则编译器可以发射代码以创建表示该 lambda 表达式的表达式树。

C# 编译器只能从表达式 Lambda(或单行 Lambda)生成表达式树。它无法解析语句 lambda (或多行 lambda)。有关 C# 中 Lambda 表达式的详细信息，请参阅[Lambda 表达式](#)。

下列代码示例展示如何通过 C# 编译器创建表示 Lambda 表达式 `num => num < 5` 的表达式树。

```
Expression<Func<int, bool>> lambda = num => num < 5;
```

通过 API 创建表达式树

通过 API 创建表达式树需要使用 `Expression` 类。类包含创建特定类型表达式树节点的静态工厂方法，比如表示参数变量的 `ParameterExpression`，或者是表示方法调用的 `MethodCallExpression`。`ParameterExpression` 名称空间还解释了 `MethodCallExpression`、`System.Linq.Expressions` 和另一种具体表达式类型。这些类型来源于抽象类型 `Expression`。

下列代码示例展示如何使用 API 创建表示 Lambda 表达式 `num => num < 5` 的表达式树。

```
// Add the following using directive to your code file:  
// using System.Linq.Expressions;  
  
// 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 });
```

在 .NET Framework 4 或更高版本中，表达式树 API 还支持赋值表达式和控制流表达式，例如循环、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。

```

// 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。

解析表达式树

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。

```

// 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

```

表达式树永久性

表达式树应具有永久性。这意味着如果你想修改某个表达式树，则必须复制该表达式树然后替换其中的节点来创建一个新的表达式树。你可以使用表达式树访问者遍历现有表达式树。有关详细信息，请参阅[如何修改表达式树 \(C#\)](#)。

编译表达式树

`Expression<TDelegate>` 类型提供了 `Compile` 方法以将表达式树表示的代码编译成可执行委托。

下列代码示例展示如何编译表达式树并运行结果代码。

```
// Creating an expression tree.  
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.
```

有关详细信息，请参阅[如何执行表达式树 \(C#\)](#)。

请参阅

- [System.Linq.Expressions](#)
- [如何执行表达式树 \(C#\)](#)
- [如何修改表达式树 \(C#\)](#)
- [Lambda 表达式](#)
- [动态语言运行时概述](#)
- [编程概念 \(C#\)](#)

如何执行表达式树 (C#)

2021/5/7 • • [Edit Online](#)

本主题演示如何执行表达式树。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。

NOTE

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>](#)，则必须对委托调用 [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。

示例

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。

```
// 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
```

编译代码

- 包括 System.Linq.Expressions 命名空间。

请参阅

- [表达式树 \(C#\)](#)
- [如何修改表达式树 \(C#\)](#)

如何修改表达式树 (C#)

2021/5/7 • • [Edit Online](#)

本主题演示如何修改表达式树。表达式树是不可变的，这意味着不能直接对它们进行修改。若要更改表达式树，必须创建现有表达式树的副本，创建此副本后，进行必要的更改。可以使用 [ExpressionVisitor](#) 类遍历现有表达式树，以及复制它访问的每个节点。

修改表达式树

1. 创建新的 **控制台应用程序** 项目。
2. 为 `System.Linq.Expressions` 命名空间的文件添加 `using` 指令。
3. 向项目中添加 `AndAlsoModifier` 类。

```
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`。若要执行此操作，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。

4. 为 `System.Linq.Expressions` 命名空间的文件添加 `using` 指令。
5. 向 `Program.cs` 文件中的 `Main` 方法添加代码以创建表达式树，并将其传递给将对其进行修改的方法。

```
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。

6. 编译并运行该应用程序。

请参阅

- [如何执行表达式树 \(C#\)](#)
- [表达式树 \(C#\)](#)

基于运行时状态进行查询 (C#)

2021/5/10 • [Edit Online](#)

考虑针对数据源定义 [IQueryable](#) 或 [IQueryable<T>](#) 的代码：

```
var companyNames = new[] {
    "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"
};

// We're using 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);
```

每次运行此代码时，都将执行相同的精确查询。这通常不是很有用，因为你可能希望代码根据运行时的情况执行不同的查询。本文介绍如何根据运行时状态执行不同的查询。

IQueryable/IQueryable<T> 和表达式树

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，提供程序通常会保持不变；查询的表达式树将因查询而异。

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，从而转换为新的 [IQueryable](#)。

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
- 调用其他 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](#)

从表达式树中使用运行时状态

假设 LINQ 提供程序支持，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：

```

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 — 尚未修改; 查询只返回不同的值, 因为 `length` 的值已更改。

调用其他 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 `IQueryable<T>` 返回方法的结果, 以获取新的查询。可以基于运行时状态在一定条件下执行此操作, 如以下示例中所示:

```

// 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:

```

// string? startsWith = /* ... */;
// string? endsWith = /* ... */;

Expression<Func<string, bool>> expr = (startsWith, endsWith) switch
{
    ("", "") => x => true,
    (_, "") => x => x.StartsWith(startsWith),
    ("", _) => x => x.EndsWith(endsWith),
    (_, _) => x => x.StartsWith(startsWith) || x.EndsWith(endsWith)
};

var qry = companyNamesSource.Where(expr);

```

你可能还希望使用第三方库(如 [LinqKit](#) 的 `PredicateBuilder`)来编写各种子表达式:

```

// 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，我们已知道编译时的元素类型 — `string` — 并因此知道查询的类型 — `IQueryable<string>`。可能需要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 `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。

`Expression<TDelegate>` 继承自 `LambdaExpression`，后者表示完整的 lambda 表达式，如下所示：

```
Expression<Func<string, bool>> expr = x => x.StartsWith("a");
```

`LambdaExpression` 具有两个组件：

- 参数列表 — `(string x)` — 由 `Parameters` 属性表示
- 主体 — `x.StartsWith("a")` — 由 `Body` 属性表示。

构造 `Expression<TDelegate>` 的基本步骤如下所示：

- 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。

```
ParameterExpression x = Parameter(typeof(string), "x");
```

- 使用你定义的 `ParameterExpression` 和 `Expression` 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：

```
Expression body = Call(
    x,
    typeof(string).GetMethod("StartsWith", new[] { typeof(string) })!,
    Constant("a")
);
```

- 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:

```
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>`), 并提供了使用工厂方法完成此操作的完整示例。

方案

假设你有多个实体类型:

```
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:

```
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:

```
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。

示例

```

// 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", new[] { 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(
                    Property(          // .Contains(...)
                        prm,           // .PropertyName
                        prp           // x
                    ),
                    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。

```

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` 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面所述构建内部表达式树，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：

```

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",
        new[] { elementType },
        source.Expression,
        Lambda(body, prm!)
    );

    return source.Provider.CreateQuery(filteredTree);
}

```

请注意，在这种情况下，你没有编译时 `T` 泛型占位符，因此你将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：

```
// 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);
}
```

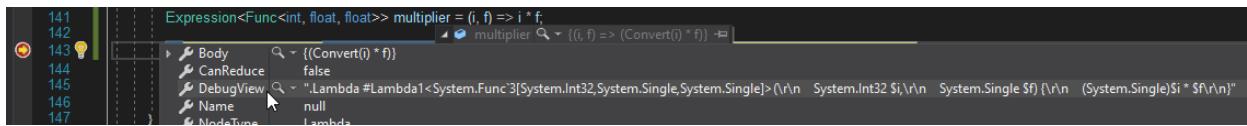
请参阅

- [表达式树 \(C#\)](#)
- [如何执行表达式树 \(C#\)](#)
- [在运行时动态指定谓词筛选器](#)

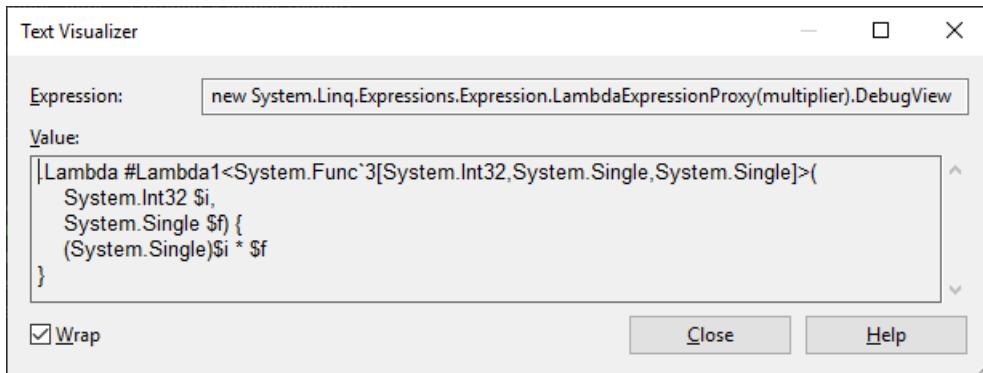
在 Visual Studio 中调试表达式树 (C#)

2020/11/2 • [Edit Online](#)

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，该属性使用特殊语法表示表达式树。(请注意，`DebugView` 仅在调试模式下可用。)

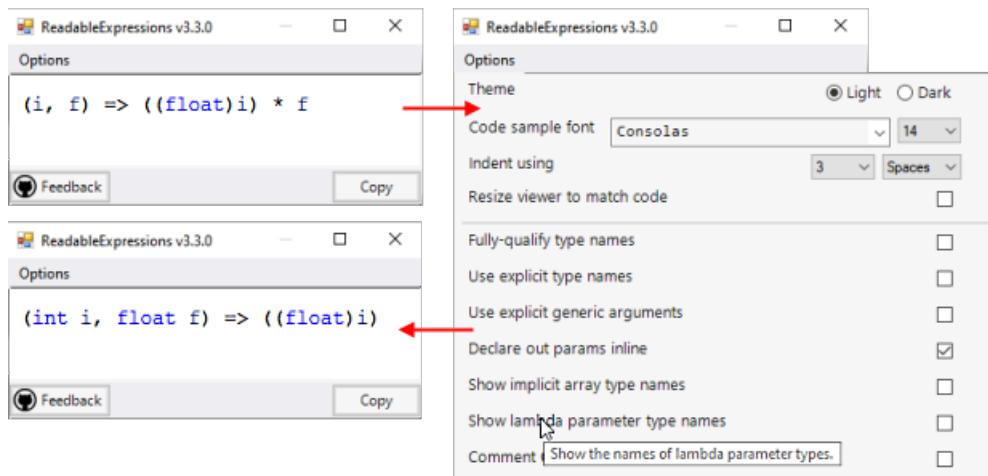


由于 `DebugView` 是一个字符串，因此可以使用内置文本可视化工具在多行中进行查看，方法是从 `DebugView` 标签旁边的放大镜图标中选择“文本可视化工具”。

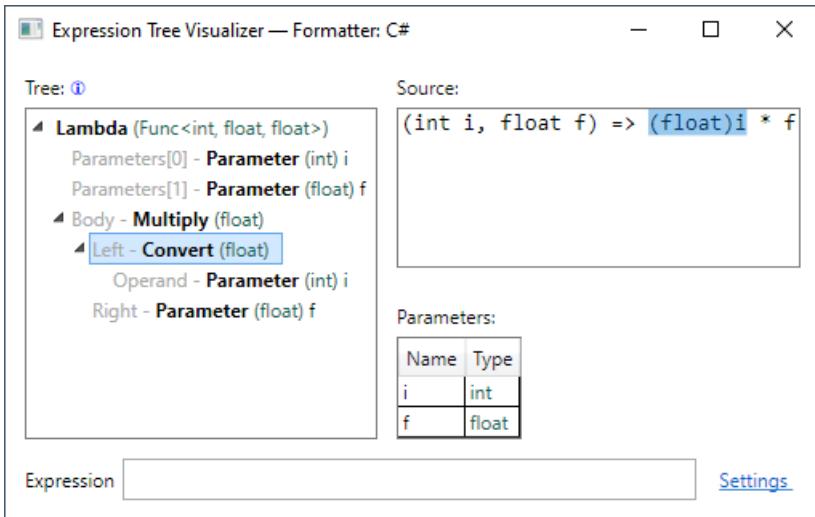


或者，可以为表达式树安装和使用自定义可视化工具，例如：

- 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：



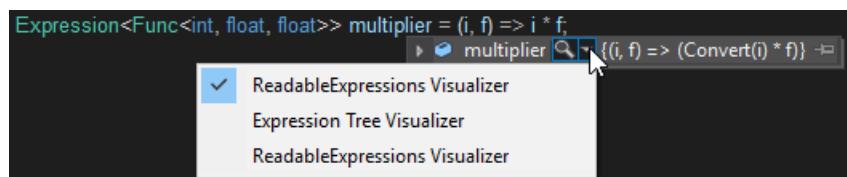
- 表达式树可视化工具 (MIT 许可证) 提供表达式树及其各个节点的树视图：



打开表达式树的可视化工具

1. 单击“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：



2. 单击要使用的可视化工具。

请参阅

- [表达式树 \(C#\)](#)
- [在 Visual Studio 中进行调试](#)
- [创建自定义可视化工具](#)
- [DebugView 语法](#)

DebugView 语法

2021/3/5 • [Edit Online](#)

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`。

示例

```
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。

II	III	SUFFIX
<code>System.UInt32</code>	<code>uint</code>	<code>U</code>
<code>System.Int64</code>	<code>long</code>	<code>L</code>
<code>System.UInt64</code>	<code>ulong</code>	<code>UL</code>
<code>System.Double</code>	<code>double</code>	<code>D</code>
<code>System.Single</code>	<code>float</code>	<code>F</code>
<code>System.Decimal</code>	<code>decimal</code>	<code>M</code>

示例

```
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。

示例

```
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，则会为其分配一个自动生成的名称，例如 `#Lambda1` 或 `#Lambda2`。

示例

```
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，则会为其分配一个自动生成的名称，例如 #Label1 或 #Label2。

示例

```
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 #+。

示例

```
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
*/
```

迭代器 (C#)

2020/11/2 • [Edit Online](#)

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。

```
static void Main()
{
    foreach (int number in SomeNumbers())
    {
        Console.WriteLine(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。

NOTE

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。

```

static void Main()
{
    foreach (int number in EvenSequence(5, 18))
    {
        Console.WriteLine(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。

```

static void Main()
{
    DaysOfTheWeek days = new DaysOfTheWeek();

    foreach (string day in days)
    {
        Console.WriteLine(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。

```
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。

```

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();
}

```

```

        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 `Ilasm.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](#)
- [yield](#)
- [对数组使用 foreach](#)
- [泛型](#)

语言集成查询 (LINQ) (C#)

2021/5/7 • [Edit Online](#)

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。可以使用语言关键字和熟悉的运算符针对强类型化对象集合编写查询。LINQ 系列技术提供了针对对象 (LINQ to Objects)、关系数据库 (LINQ to SQL) 和 XML (LINQ to XML) 的一致查询体验。

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。

在 C# 中可为以下对象编写 LINQ 查询：SQL Server 数据库、XML 文档、ADO.NET 数据集以及支持 [IEnumerable](#) 或泛型 [IEnumerable<T>](#) 接口的任何对象集合。此外，第三方也为许多 Web 服务和其他数据库实现提供了 LINQ 支持。

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。

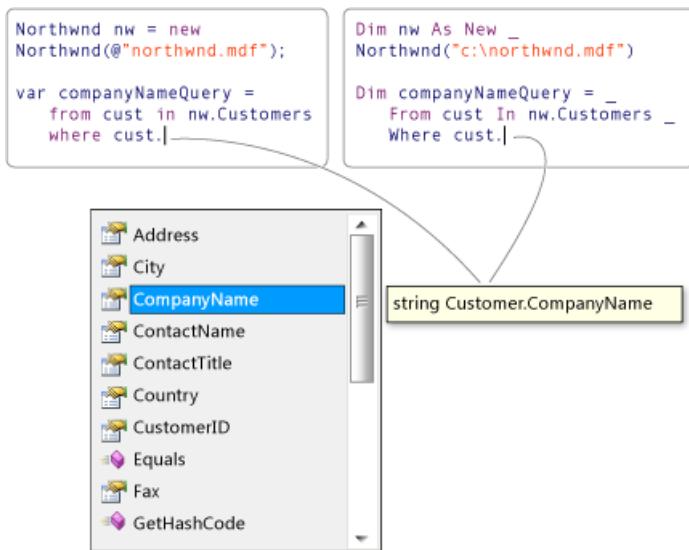
```
class LINQQueryExpressions
{
    static void Main()
    {

        // Specify the data source.
        int[] scores = new int[] { 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
```

下图为 Visual Studio 中的图例，显示了使用 C# 和 Visual Basic 针对 SQL Server 数据库编写的不完整 LINQ 查询，并具有完全类型检查和 IntelliSense 支持：



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](#)。

LINQ 查询简介 (C#)

2020/11/2 • [Edit Online](#)

查询是一种从数据源检索数据的表达式。查询通常用专门的查询语言来表示。随着时间的推移，人们已经为各种数据源开发了不同的语言；例如，用于关系数据库的 SQL 和用于 XML 的 XQuery。因此，开发人员对于他们必须支持的每种数据源或数据格式，都不得不学习一种新的查询语言。LINQ 通过提供处理各种数据源和数据格式的数据的一致模型，简化了这一情况。在 LINQ 查询中，始终会用到对象。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、ADO.NET 数据集、.NET 集合中的数据以及 LINQ 提供程序可用的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。

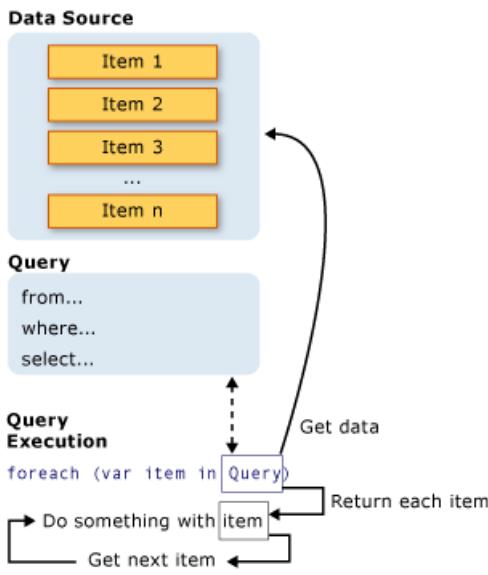
下面的示例演示如何用源代码表示查询操作的三个部分。为方便起见，此示例将一个整数数组用作数据源；但其中涉及的概念同样适用于其他数据源。本主题的其余部分也会引用此示例。

```
class IntroToLINQ
{
    static void Main()
    {
        // The Three Parts of a LINQ Query:
        // 1. Data source.
        int[] numbers = new int[7] { 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.WriteLine("{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，仅通过创建查询变量不会检索到任何数据。



数据源

上例中，数据源是一个数组，因此它隐式支持泛型 `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：

```
// Create a data source from an XML document.
// using System.Xml.Linq;
 XElement contacts = XElement.Load(@"c:\myContactList.xml");
```

借助 LINQ to SQL，首先手动或使用 [Visual Studio 中的 LINQ to SQL 工具](#) 在设计时创建对象关系映射。针对这些对象编写查询，然后由 LINQ to SQL 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>`。

```
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>` 接口或从中继承的接口的任意对象。

NOTE

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。为使编写查询的工作变得更加容易，C# 引入了新的查询

语法。

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`。（如果熟悉 SQL，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。[语言集成查询 \(LINQ\)](#) 一节中详细讨论了这些子句和其他查询子句。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。有关在后台如何构造查询的详细信息，请参阅[标准查询运算符概述 \(C#\)](#)。

NOTE

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](#)。

查询执行

延迟执行

如前所述，查询变量本身只存储查询命令。查询的实际执行将推迟到在 `foreach` 语句中循环访问查询变量之后进行。此概念称为 **延迟执行**，下面的示例对此进行了演示：

```
// Query execution.  
foreach (int num in numQuery)  
{  
    Console.WriteLine("{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，因此可以根据需要随意执行查询。例如，可以通过一个单独的应用程序持续更新数据库。在应用程序中，可以创建一个检索最新数据的查询，并可以按某一时间间隔反复执行该查询以便每次检索不同的结果。

强制立即执行

对一系列源元素执行聚合函数的查询必须首先循环访问这些元素。`Count`、`Max`、`Average` 和 `First` 就属于此类查询。由于查询本身必须使用 `foreach` 以便返回结果，因此这些查询在执行时不使用显式 `foreach` 语句。另外还要注意，这些类型的查询返回单个值，而不是 `IEnumerable` 集合。下面的查询返回源数组中偶数的计数：

```
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。

```
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。

请参阅

- [C# 中的 LINQ 入门](#)
- [演练: 用 C# 编写查询](#)
- [语言集成查询 \(LINQ\)](#)
- [foreach, in](#)
- [查询关键字 \(LINQ\)](#)

LINQ 和泛型类型 (C#)

2020/11/2 • [Edit Online](#)

LINQ 查询基于 .NET Framework 版本 2.0 中引入的泛型类型。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。

```
IEnumerable<Customer> customerQuery =
    from cust in customers
    where cust.City == "London"
    select cust;

foreach (Customer customer in customerQuery)
{
    Console.WriteLine(customer.LastName + ", " + customer.FirstName);
}
```

有关详细信息，请参阅[LINQ 查询操作中的类型关系](#)。

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：

```
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](#)。

请参阅

- [泛型](#)

基本 LINQ 查询操作 (C#)

2020/11/2 • [Edit Online](#)

本主题简要介绍了 LINQ 查询表达式和一些在查询中执行的典型操作。下面各主题中提供了更具体的信息：

[LINQ 查询表达式](#)

[标准查询运算符概述 \(C#\)](#)

[演练:用 C# 编写查询](#)

NOTE

如果你已熟悉查询语言(如 SQL 或 XQuery)，则可以跳过本主题的大部分内容。请参阅下一节中的“`from` 子句”部分，了解 LINQ 查询表达式中的子句顺序。

获取数据源

在 LINQ 查询中，第一步是指定数据源。和大多数编程语言相同，在使用 C# 时也必须先声明变量，然后才能使用它。在 LINQ 查询中，先使用 `from` 子句引入数据源 (`customers`) 和范围变量 (`cust`)。

```
//queryAllCustomers is an IEnumerable<Customer>
var queryAllCustomers = from cust in customers
                        select cust;
```

范围变量就像 `foreach` 循环中的迭代变量，但查询表达式中不会真正发生迭代。当执行查询时，范围变量将充对 `customers` 中每个连续的元素的引用。由于编译器可以推断 `cust` 的类型，因此无需显式指定它。可通过 `let` 子句引入其他范围变量。有关详细信息，请参阅 [let 子句](#)。

NOTE

对于非泛型数据源(例如 `ArrayList`)，必须显式键入范围变量。有关详细信息，请参阅[如何使用 LINQ \(C#\) 和 From 子句](#)查询 `ArrayList`。

筛选

或许，最常见的查询操作是以布尔表达式的形式应用筛选器。筛选器使查询仅返回表达式为 `true` 的元素。将通过使用 `where` 子句生成结果。筛选器实际指定要从源序列排除哪些元素。在下列示例中，仅返回地址位于“London”的 `customers`。

```
var queryLondonCustomers = from cust in customers
                            where cust.City == "London"
                            select cust;
```

可使用熟悉的 C# 逻辑 `AND` 和 `OR` 运算符，在 `where` 子句中根据需要应用尽可能多的筛选器表达式。例如，若要仅返回来自“London”的客户 `AND` 该客户名称为“Devon”，可编写以下代码：

```
where cust.City == "London" && cust.Name == "Devon"
```

要返回来自 London 或 Paris 的客户，可编写以下代码：

```
where cust.City == "London" || cust.City == "Paris"
```

有关详细信息，请参阅 [where 子句](#)。

中间件排序

对返回的数据进行排序通常很方便。`orderby` 子句根据要排序类型的默认比较器，对返回序列中的元素排序。例如，基于 `Name` 属性，可将下列查询扩展为对结果排序。由于 `Name` 是字符串，默认比较器将按字母顺序从 A 到 Z 进行排序。

```
var queryLondonCustomers3 =
    from cust in customers
    where cust.City == "London"
    orderby cust.Name ascending
    select cust;
```

要对结果进行从 Z 到 A 的逆序排序，请使用 `orderby...descending` 子句。

有关详细信息，请参阅 [orderby 子句](#)。

分组

`group` 子句用于对根据您指定的键所获得的结果进行分组。例如，可指定按 `City` 对结果进行分组，使来自 London 或 Paris 的所有客户位于单独的组内。在这种情况下，`cust.City` 是键。

```
// queryCustomersByCity is an IEnumerable<IGrouping<string, Customer>>
var queryCustomersByCity =
    from cust in customers
    group cust by cust.City;

// customerGroup is an IGrouping<string, Customer>
foreach (var customerGroup in queryCustomersByCity)
{
    Console.WriteLine(customerGroup.Key);
    foreach (Customer customer in customerGroup)
    {
        Console.WriteLine("    {0}", customer.Name);
    }
}
```

使用 `group` 子句结束查询时，结果将以列表的形式列出。列表中的每个元素都是具有 `Key` 成员的对象，列表中的元素根据该键被分组。在循环访问生成组序列的查询时，必须使用嵌套 `foreach` 循环。外层循环循环访问每个组，内层循环循环访问每个组的成员。

如果必须引用某个组操作的结果，可使用 `into` 关键字创建能被进一步查询的标识符。下列查询仅返回包含两个以上客户的组：

```
// custQuery is an IEnumerable<IGrouping<string, Customer>>
var custQuery =
    from cust in customers
    group cust by cust.City into custGroup
    where custGroup.Count() > 2
    orderby custGroup.Key
    select custGroup;
```

有关详细信息，请参阅 [group 子句](#)。

联接

联接操作在不同序列间创建关联，这些序列在数据源中未被显式模块化。例如，可通过执行联接来查找所有位置相同的客户和分销商。在 LINQ 中，`join` 子句始终作用于对象集合，而非直接作用于数据库表。

```
var innerJoinQuery =  
    from cust in customers  
    join dist in distributors on cust.City equals dist.City  
    select new { CustomerName = cust.Name, DistributorName = dist.Name };
```

在 LINQ 中，不必像在 SQL 中那样频繁使用 `join`，因为 LINQ 中的外键在对象模型中表示为包含项集合的属性。例如 `Customer` 对象包含 `Order` 对象的集合。不必执行联接，只需使用点表示法访问订单：

```
from order in Customer.Orders...
```

有关详细信息，请参阅 [join 子句](#)。

选择(投影)

`select` 子句生成查询结果并指定每个返回的元素的“形状”或类型。例如，可以指定结果包含的是整个 `Customer` 对象、仅一个成员、成员的子集，还是某个基于计算或新对象创建的完全不同的结果类型。当 `select` 子句生成除源元素副本以外的内容时，该操作称为投影。使用投影转换数据是 LINQ 查询表达式的一种强大功能。有关详细信息，请参阅 [使用 LINQ \(C#\)](#) 和 [select 子句](#) 进行数据转换。

请参阅

- [LINQ 查询表达式](#)
- [演练:用 C# 编写查询](#)
- [查询关键字 \(LINQ\)](#)
- [匿名类型](#)

使用 LINQ 进行数据转换 (C#)

2020/11/2 • [Edit Online](#)

语言集成查询 (LINQ) 不只是检索数据。它也是用于转换数据的强大工具。通过使用 LINQ 查询，可以使用源序列作为输入，并通过多种方式对其进行修改，以创建新的输出序列。通过排序和分组，你可以修改序列本身，而无需修改这些元素本身。但也许 LINQ 查询最强大的功能是创建新类型。这可以在 `select` 子句中完成。例如，可以执行下列任务：

- 将多个输入序列合并为具有新类型的单个输出序列。
- 创建其元素由源序列中每个元素的一个或多个属性组成的输出序列。
- 创建其元素由对源数据执行的操作结果组成的输出序列。
- 创建其他格式的输出序列。例如，可以将数据从 SQL 行或文本文件转换为 XML。

这只是几个例子。当然，可以以各种方式在同一查询中组合这些转换。此外，一个查询的输出序列可以用作新查询的输入序列。

将多个输入联接到一个输出序列中

可以使用 LINQ 查询创建包含元素的输出序列，这些元素来自多个输入序列。以下示例演示如何组合两个内存中数据结构，但相同的原则可应用于组合来自 XML 或 SQL 或数据集源的数据。假设以下两种类类型：

```
class Student
{
    public string First { get; set; }
    public string Last { get; set; }
    public int ID { get; set; }
    public string Street { get; set; }
    public string City { get; set; }
    public List<int> Scores;
}

class Teacher
{
    public string First { get; set; }
    public string Last { get; set; }
    public int ID { get; set; }
    public string City { get; set; }
}
```

以下示例演示了查询：

```

class DataTransformations
{
    static void Main()
    {
        // Create the first data source.
        List<Student> students = new List<Student>()
        {
            new Student { First="Svetlana",
                Last="Omelchenko",
                ID=111,
                Street="123 Main Street",
                City="Seattle",
                Scores= new List<int> { 97, 92, 81, 60 } },
            new Student { First="Claire",
                Last="O'Donnell",
                ID=112,
                Street="124 Main Street",
                City="Redmond",
                Scores= new List<int> { 75, 84, 91, 39 } },
            new Student { First="Sven",
                Last="Mortensen",
                ID=113,
                Street="125 Main Street",
                City="Lake City",
                Scores= new List<int> { 88, 94, 65, 91 } },
        };
    }

    // Create the second data source.
    List<Teacher> teachers = new List<Teacher>()
    {
        new Teacher { First="Ann", Last="Beebe", ID=945, City="Seattle" },
        new Teacher { First="Alex", Last="Robinson", ID=956, City="Redmond" },
        new Teacher { First="Michiyo", Last="Sato", ID=972, City="Tacoma" }
    };

    // Create the query.
    var peopleInSeattle = (from student in students
                           where student.City == "Seattle"
                           select student.Last)
                           .Concat(from teacher in teachers
                                   where teacher.City == "Seattle"
                                   select teacher.Last);

    Console.WriteLine("The following students and teachers live in Seattle:");
    // Execute the query.
    foreach (var person in peopleInSeattle)
    {
        Console.WriteLine(person);
    }

    Console.WriteLine("Press any key to exit.");
    Console.ReadKey();
}
/* Output:
   The following students and teachers live in Seattle:
   Omelchenko
   Beebe
*/

```

有关详细信息，请参阅 [join 子句](#) 和 [select 子句](#)。

选择每个源元素的子集

有两种主要方法来选择源序列中每个元素的子集：

1. 若要仅选择源元素的一个成员，请使用点操作。在以下示例中，假设 `Customer` 对象包含多个公共属性，包括名为 `City` 的字符串。在执行时，此查询将生成字符串的输出序列。

```
var query = from cust in Customers  
            select cust.City;
```

2. 若要创建包含多个源元素属性的元素，可以使用带有命名对象或匿名类型的对象初始值设定项。以下示例演示如何使用匿名类型封装每个 `Customer` 元素的两个属性：

```
var query = from cust in Customer  
            select new {Name = cust.Name, City = cust.City};
```

有关详细信息，请参阅[对象和集合初始值设定项](#)和[匿名类型](#)。

将内存中对象转换为 XML

LINQ 查询可以轻松地在内存中数据结构、SQL 数据库、ADO.NET 数据集和 XML 流或文档之间转换数据。以下示例将内存中数据结构中的对象转换为 XML 元素。

```
class XMLTransform  
{  
    static void Main()  
    {  
        // Create the data source by using a collection initializer.  
        // The Student class was defined previously in this topic.  
        List<Student> students = new List<Student>()  
        {  
            new Student {First="Svetlana", Last="Omelchenko", ID=111, Scores = new List<int>{97, 92, 81, 60}},  
            new Student {First="Claire", Last="O'Donnell", ID=112, Scores = new List<int>{75, 84, 91, 39}},  
            new Student {First="Sven", Last="Mortensen", ID=113, Scores = new List<int>{88, 94, 65, 91}}  
        };  
  
        // 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.First),  
                new XElement("Last", student.Last),  
                new XElement("Scores", scores)  
            ) // end "student"  
        ); // end "Root"  
  
        // Execute the query.  
        Console.WriteLine(studentsToXML);  
  
        // Keep the console open in debug mode.  
        Console.WriteLine("Press any key to exit.");  
        Console.ReadKey();  
    }  
}
```

此代码生成以下 XML 输出：

```
<Root>
  <student>
    <First>Svetlana</First>
    <Last>Omelchenko</Last>
    <Scores>97,92,81,60</Scores>
  </student>
  <student>
    <First>Claire</First>
    <Last>O'Donnell</Last>
    <Scores>75,84,91,39</Scores>
  </student>
  <student>
    <First>Sven</First>
    <Last>Mortensen</Last>
    <Scores>88,94,65,91</Scores>
  </student>
</Root>
```

有关详细信息，请参阅[在 C# 中创建 XML 树 \(LINQ to XML\)](#)。

对源元素执行操作

输出序列可能不包含源序列中的任何元素或元素属性。输出可能是使用源元素作为输入参数而计算得出的值序列。

以下查询将采用表示圆半径的数字序列，计算每个半径范围的面积，并返回输出序列，其中包含以所计算面积进行格式设置的字符串。

输出序列的每个字符串都将使用[字符串内插](#)进行格式设置。内插字符串的左引号前有一个 \$，并且可以在内插字符串内部的大括号内执行操作。执行这些操作后，结果将进行串联。

NOTE

如果查询将被转换为另一个域，则不支持在查询表达式中调用方法。例如，不能在 LINQ to SQL 中调用普通的 C# 方法，因为 SQL Server 没有用于它的上下文。但是，可以将存储过程映射到方法并调用这些方法。有关详细信息，请参阅[存储过程](#)。

```
class FormatQuery
{
    static void Main()
    {
        // Data source.
        double[] radii = { 1, 2, 3 };

        // LINQ query using method syntax.
        IEnumerable<string> output =
            radii.Select(r => $"Area for a circle with a radius of '{r}' = {r * r * Math.PI:F2}");

        /*
        // LINQ query using query syntax.
        IEnumerable<string> output =
            from rad in radii
            select $"Area for a circle with a radius of '{rad}' = {rad * rad * Math.PI:F2}";
        */

        foreach (string s in output)
        {
            Console.WriteLine(s);
        }

        // Keep the console open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}

/* Output:
   Area for a circle with a radius of '1' = 3.14
   Area for a circle with a radius of '2' = 12.57
   Area for a circle with a radius of '3' = 28.27
*/
```

请参阅

- [语言集成查询 \(LINQ\) \(C#\)](#)
- [LINQ to SQL](#)
- [LINQ to DataSet](#)
- [LINQ to XML \(C#\)](#)
- [LINQ 查询表达式](#)
- [select 子句](#)

LINQ 查询操作中的类型关系 (C#)

2020/11/2 • [Edit Online](#)

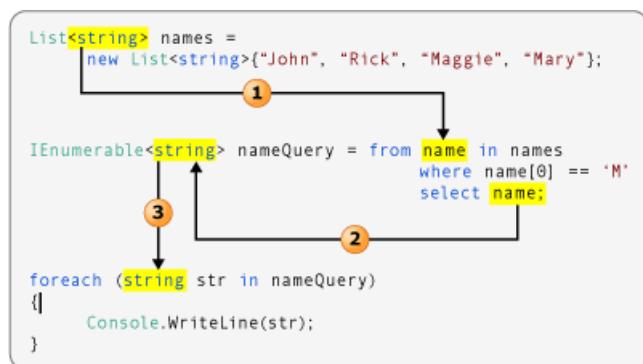
若要有效编写查询，应了解完整的查询操作中的变量类型是如何全部彼此关联的。如果了解这些关系，就能够更容易地理解文档中的 LINQ 示例和代码示例。另外，还能了解在使用 `var` 隐式对变量进行类型化时的后台操作。

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

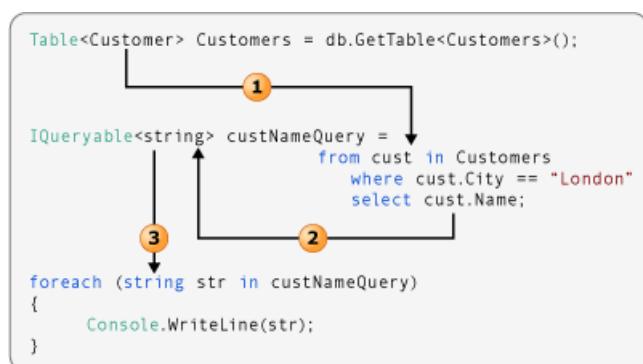
下图演示不对数据执行转换的 LINQ to Objects 查询操作。源包含一个字符串序列，查询输出也是一个字符串序列。



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`。

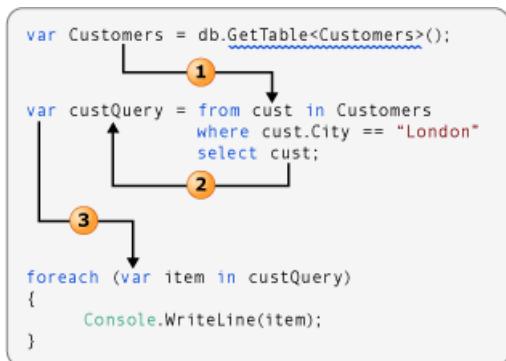
下图演示稍微复杂的转换。`select` 语句返回只捕获原始 `Customer` 对象的两个成员的匿名类型。



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。



有关 `var` 的详细信息，请参阅[隐式类型本地变量](#)。

LINQ 中的查询语法和方法语法 (C#)

2020/11/2 • [Edit Online](#)

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，但是许多人发现查询语法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。因此，即使在开始编写 LINQ 查询时，熟悉如何在查询和查询表达式本身中使用方法语法也十分有用。

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。

```
class QueryVMethodSyntax
{
    static void Main()
    {
        int[] numbers = { 5, 10, 8, 3, 6, 12};

        //Query syntax:
        I Enumerable<int> numQuery1 =
            from num in numbers
            where num % 2 == 0
            orderby num
            select num;

        //Method syntax:
        I Enumerable<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 + " ");
        }

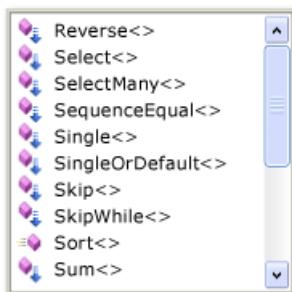
        // Keep the console open in debug mode.
        Console.WriteLine(System.Environment.NewLine);
        Console.WriteLine("Press any key to exit");
        Console.ReadKey();
    }
}
/*
Output:
6 8 10 12
6 8 10 12
*/
```

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>`（如同你会回忆起的那样）。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`）。这些都是标准查询运算符。

```
List<string> list = new List<string>();
list.|
```



虽然看起来似乎 `IEnumerable<T>` 进行了重新定义以包括这些其他方法，不过实际上情况并非如此。标准查询运算符作为一种新类型的方法（称为扩展方法）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(...)`。

若要开始使用 LINQ，你在扩展方法方面实际需要了解的所有内容是如何使用正确的 `using` 指令将它们引入应用程序的范围。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 LINQ to SQL 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额外的扩展方法。

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 `Where` 方法：

`Where(num => num % 2 == 0)`。此内联表达式称为 lambda 表达式。可采用匿名方法、泛型委托或表达式树的形式编写原本必须以更繁琐的形式编写的代码，这是一种便利的方式。在 C# 中，`=>` 是 lambda 运算符（读为“转到”）。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。

若要开始使用 LINQ，不必大量使用 lambda。但是，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。进一步熟悉 lambda 之后，你会发现它们是 LINQ 工具箱中一种强大而灵活的工具。有关详细信息，请参阅[Lambda 表达式](#)。

查询的可组合性

在前面的代码示例中，请注意，`OrderBy` 方法通过对 `Where` 调用使用点运算符来调用。`Where` 会生成经过筛选的序列，然后 `OrderBy` 通过进行排序来对该序列进行操作。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。这是当你使用查询语法编写查询时，编译器在幕后进行的工作。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（即使在执行过它之后）。

支持 LINQ 的 C# 功能

2020/11/2 • [Edit Online](#)

下一节介绍 C# 3.0 中引入的新语言构造。虽然这些新功能在一定程度上都用于 LINQ 查询，但并不限于 LINQ，如果认为有用，在任何情况下都可以使用这些新功能。

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 `IEnumerable` 集合。在编译时，查询语法转换为对 LINQ 提供程序的标准查询运算符扩展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。

```
var query = from str in stringArray
            group str by str[0] into stringGroup
            orderby stringGroup.Key
            select stringGroup;
```

有关详细信息，请参阅 [LINQ 查询表达式](#)。

隐式类型化变量 (`var`)

可以使用 `var` 修饰符来指示编译器推断并分配类型，而不必在声明并初始化变量时显式指定类型，如下所示：

```
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，但它只能用于本地变量。也可以使用隐式类型声明数组。

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：

```
var cust = new Customer { Name = "Mike", Phone = "555-1212" };
```

继续我们的 `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`，我们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：

```
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` 类更多的属性，例如 `OrderSize`，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；我们选择与我们的类相关联的数据。因此，我们现在有填充了我们想要的多个新 `Customer` 的 `IEnumerable`。上述代码也可以使用 LINQ 的方法语法编写：

```
var newLargeOrderCustomers = IncomingOrders.Where(x => x.OrderSize > 5).Select(y => new Customer { Name = y.Name, Phone = y.Phone });
```

有关详情，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

匿名类型由编译器构造，且类型名称只可用于编译器。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：

```
select new {name = cust.Name, phone = cust.Phone};
```

有关详细信息，请参阅[匿名类型](#)。

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。

有关详细信息，请参阅[扩展方法](#)。

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。

有关详细信息，请参见：

- [匿名函数](#)
- [Lambda 表达式](#)
- [表达式树 \(C#\)](#)

请参阅

- [语言集成查询 \(LINQ\) \(C#\)](#)

演练：用 C# 编写查询 (LINQ)

2021/5/7 • [Edit Online](#)

此演练演示用干编写 LINQ 查询表达式的 C# 语言功能。

创建 C# 项目

NOTE

以下说明适用于 Visual Studio。如果使用其他开发环境, 请创建包含对 System.Core.dll 的引用的控制台项目和用于 `System.Linq` 命名空间的 `using` 指令。

在 **Visual Studio** 中创建项目

1. 启动 Visual Studio。
 2. 在菜单栏上, 依次选择“文件”、“新建”、“项目”。
- “**新建项目**”对话框随即打开。
3. 依次展开“已安装”、“模板”、“Visual C#”, 然后选择“控制台应用程序”。
 4. 在“名称”文本框中, 输入不同的名称或接受默认名称, 然后选择“确定”按钮。

新项目将出现在“解决方案资源管理器”中。

5. 注意, 此项目包含对 System.Core.dll 的引用和用于 `System.Linq` 命名空间的 `using` 指令。

创建内存中的数据源

用于查询的数据源是 `Student` 对象的简单列表。每个 `Student` 记录都有名字、姓氏和整数数组(表示该学生在课堂上的测试分数)。将此代码复制到项目中。请注意下列特性:

- `Student` 类包含自动实现的属性。
- 列表中的每个学生都可使用对象初始值设定项进行初始化。
- 列表本身可使用集合初始值设定项进行初始化。

将在不显式调用任何构造函数和使用显式成员访问的情况下初始化并实例化整个数据结构。有关这些新功能的详细信息, 请参阅[自动实现的属性与对象和集合初始值设定项](#)。

添加数据源

- 向项目中的 `Program` 类添加 `Student` 类和经过初始化的学生列表。

```

public class Student
{
    public string First { get; set; }
    public string Last { get; set; }
    public int ID { get; set; }
    public List<int> Scores;
}

// Create a data source by using a collection initializer.
static List<Student> students = new List<Student>
{
    new Student {First="Svetlana", Last="Omelchenko", ID=111, Scores= new List<int> {97, 92, 81, 60}},
    new Student {First="Claire", Last="O'Donnell", ID=112, Scores= new List<int> {75, 84, 91, 39}},
    new Student {First="Sven", Last="Mortensen", ID=113, Scores= new List<int> {88, 94, 65, 91}},
    new Student {First="Cesar", Last="Garcia", ID=114, Scores= new List<int> {97, 89, 85, 82}},
    new Student {First="Debra", Last="Garcia", ID=115, Scores= new List<int> {35, 72, 91, 70}},
    new Student {First="Fadi", Last="Fakhouri", ID=116, Scores= new List<int> {99, 86, 90, 94}},
    new Student {First="Hanying", Last="Feng", ID=117, Scores= new List<int> {93, 92, 80, 87}},
    new Student {First="Hugo", Last="Garcia", ID=118, Scores= new List<int> {92, 90, 83, 78}},
    new Student {First="Lance", Last="Tucker", ID=119, Scores= new List<int> {68, 79, 88, 92}},
    new Student {First="Terry", Last="Adams", ID=120, Scores= new List<int> {99, 82, 81, 79}},
    new Student {First="Eugene", Last="Zabokritski", ID=121, Scores= new List<int> {96, 85, 91, 60}},
    new Student {First="Michael", Last="Tucker", ID=122, Scores= new List<int> {94, 92, 91, 91}}
};

```

向学生列表添加新学生

- 向 `Students` 列表添加一个新 `Student`，并按自己的选择使用名称和测试分数。尝试键入所有新学生信息，以便更好地了解对象初始值设定项的语法。

创建查询

创建简单查询

- 在应用程序的 `Main` 方法中，创建简单查询，执行该查询时，将生成所有在第一次测试中分数高于 90 的学生的列表。注意，由于选定全部 `Student` 对象，所以查询的类型为 `IEnumerable<Student>`。尽管该代码也可以通过使用 `var` 关键字来使用隐式类型化，但可以使用显式类型化清楚地展示结果。（有关 `var` 的详细信息，请参阅[隐式类型化局部变量](#)。）

另请注意，查询的范围变量 `student` 用作指向源中每个 `Student` 引用，提供对每个对象的成员访问。

```

// 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;

```

执行查询

执行查询

- 现在，编写 `foreach` 循环，用于执行查询。注意以下有关代码的注意事项：

- 通过 `foreach` 循环中的迭代变量，可访问返回的序列中的每个元素。
- 此变量的类型是 `Student`，并且可与查询变量 `IEnumerable<Student>` 的类型兼容。

- 添加此代码后，生成并运行应用程序，以在“控制台”窗口中查看结果。

```
// Execute the query.  
// var could be used here also.  
foreach (Student student in studentQuery)  
{  
    Console.WriteLine("{0}, {1}", student.Last, student.First);  
}  
  
// Output:  
// Omelchenko, Svetlana  
// Garcia, Cesar  
// Fakhouri, Fadi  
// Feng, Hanying  
// Garcia, Hugo  
// Adams, Terry  
// Zabokritski, Eugene  
// Tucker, Michael
```

添加其他筛选条件

- 在 `where` 子句中，可以组合多个布尔条件，以便进一步细化查询。以下代码添加了一个条件，以便该查询返回第一个分数高于 90 分，并且最后一个分数低于 80 分的那些学生。`where` 子句应与以下代码类似。

```
where student.Scores[0] > 90 && student.Scores[3] < 80
```

有关详细信息，请参阅 [where 子句](#)。

修改查询

对结果进行排序

- 如果结果按某种顺序排列，则浏览结果会更容易。你可以根据源元素中的任何可访问字段对返回的序列进行排序。例如，以下 `orderby` 子句将结果按照每个学生的姓氏以字母从 A 到 Z 的顺序排列。将以下 `orderby` 子句添加到查询中，紧跟 `where` 语句之后、`select` 语句之前：

```
orderby student.Last ascending
```

- 现在，更改 `orderby` 子句，以便将结果根据第一次测试的分数以倒序（从最高分到最低分）的顺序排列。

```
orderby student.Scores[0] descending
```

- 更改 `WriteLine` 格式字符串，以便查看分数：

```
Console.WriteLine("{0}, {1} {2}", student.Last, student.First, student.Scores[0]);
```

有关详细信息，请参阅 [orderby 子句](#)。

对结果进行分组

- 分组是查询表达式中的强大功能。包含 `group` 子句的查询将生成一系列组，每个组本身包含一个 `Key` 和一个序列，该序列由该组的所有成员组成。以下新查询使用学生的姓的第一个字母作为关键字对学生进行分组。

```
// studentQuery2 is an IEnumerable<IGrouping<char, Student>>  
var studentQuery2 =  
    from student in students  
    group student by student.Last[0];
```

2. 注意，查询类型现已更改。该查询现在生成一系列将 `char` 类型作为键的组，以及一系列 `Student` 对象。由于查询的类型已更改，因此以下代码也将更改 `foreach` 执行循环：

```
// studentGroup is a IGrouping<char, Student>
foreach (var studentGroup in studentQuery2)
{
    Console.WriteLine(studentGroup.Key);
    foreach (Student student in studentGroup)
    {
        Console.WriteLine(" {0}, {1}",
                          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
```

3. 在“控制台”窗口中运行应用程序并查看结果。

有关详细信息，请参阅 [group 子句](#)。

对变量进行隐式类型化

1. `IGroupings` 的显式编码 `IEnumerables` 将快速变得冗长。使用 `var` 可以更方便地编写相同的查询和 `foreach` 循环。`var` 关键字不会更改对象的类型；它仅指示编译器推断类型。将 `studentQuery` 和迭代变量 `group` 的类型更改为 `var`，然后重新运行查询。注意，在内部 `foreach` 循环中，该迭代变量仍类型化为 `Student`，并且查询的工作原理和以前一样。将 `s` 迭代变量更改为 `var`，然后再次运行查询。将看到完全相同的结果。

```
var studentQuery3 =  
    from student in students  
    group student by student.Last[0];  
  
foreach (var groupOfStudents in studentQuery3)  
{  
    Console.WriteLine(groupOfStudents.Key);  
    foreach (var student in groupOfStudents)  
    {  
        Console.WriteLine("  {0}, {1}",  
            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
```

有关 `var` 的详细信息, 请参阅[隐式类型化局部变量](#)。

按照键值对组进行排序

- 运行上一查询时, 会发现这些组不是按字母顺序排序的。若要更改此排序, 必须在 `group` 子句后提供 `orderby` 子句。但若要使用 `orderby` 子句, 首先需要一个标识符, 用作对 `group` 子句创建的组的引用。可以使用 `into` 关键字提供该标识符, 如下所示:

```
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(" {0}, {1}",  
            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
```

运行此查询时，将看到这些组现在已按字母顺序排序。

使用 `let` 引入标识符

1. 可以使用 `let` 关键字来引入查询表达式中任何表达式结果的标识符。此标识符可以提供方便(如下面的示例所示)，也可以通过存储表达式的结果来避免多次计算，从而提高性能。

```

// studentQuery5 is an IEnumerable<string>
// 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

```

有关详细信息，请参阅 [let 子句](#)。

在查询表达式中使用方法语法

- 如 [LINQ 中的查询语法和方法语法](#) 中所述，某些查询操作只能使用方法语法来表示。以下代码为源序列中的每个 `Student` 计算总分，然后对该查询的结果调用 `Average()` 方法来计算班级平均分。

```

var studentQuery6 =
    from student in students
    let totalScore = student.Scores[0] + student.Scores[1] +
        student.Scores[2] + student.Scores[3]
    select totalScore;

double averageScore = studentQuery6.Average();
Console.WriteLine("Class average score = {0}", averageScore);

// Output:
// Class average score = 334.166666666667

```

在 `select` 子句转换或投影

- 查询生成的序列的元素与源序列中的元素不同，这种情况很常见。删除或注释掉以前的查询和执行循环，并将其替换为以下代码。请注意，该查询将返回字符串序列，而不是 `Students`，这种情况将反映在 `foreach` 循环中。

```

IEnumerable<string> studentQuery7 =
    from student in students
    where student.Last == "Garcia"
    select student.First;

Console.WriteLine("The Garcias in the class are:");
foreach (string s in studentQuery7)
{
    Console.WriteLine(s);
}

// Output:
// The Garcias in the class are:
// Cesar
// Debra
// Hugo

```

2. 本演练中前面的代码表明班级平均分大约为 334 分。若要生成总分数高于班级平均分的 `Students` 及其 `Student ID` 的序列，可以在 `select` 语句中使用匿名类型：

```

var studentQuery8 =
    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 studentQuery8)
{
    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

```

后续步骤

熟悉了在 C# 中使用查询的基本情况后，便可以开始阅读你感兴趣的具体类型的 LINQ 提供程序的文档和示例：

[LINQ to SQL](#)

[LINQ to DataSet](#)

[LINQ to XML \(C#\)](#)

[LINQ to Objects \(C#\)](#)

请参阅

- [语言集成查询 \(LINQ\) \(C#\)](#)

- [LINQ 查询表达式](#)

标准查询运算符概述 (C#)

2020/11/2 • [Edit Online](#)

标准查询运算符是组成 LINQ 模式的方法。这些方法中的大多数都作用于序列；其中序列指其类型实现 `IEnumerable<T>` 接口或 `IQueryable<T>` 接口的对象。标准查询运算符提供包括筛选、投影、聚合、排序等在内的查询功能。

共有两组 LINQ 标准查询运算符，一组作用于类型 `IEnumerable<T>` 的对象，另一组作用于类型 `IQueryable<T>` 的对象。构成每个集合的方法分别是 `Enumerable` 和 `Queryable` 类的静态成员。这些方法被定义为作为方法运行目标的类型的扩展方法。可以使用静态方法语法或实例方法语法来调用扩展方法。

此外，多个标准查询运算符方法作用于那些基于 `IEnumerable<T>` 或 `IQueryable<T>` 的类型外的类型。`Enumerable` 类型定义了两种这样的方法，这两种方法都作用于类型 `IEnumerable` 的对象。这些方法 (`Cast<TResult>(IEnumerable)` 和 `OfType<TResult>(IEnumerable)`) 均允许在 LINQ 模式中查询非参数化或非泛型集合。这些方法通过创建一个强类型的对象集合来实现这一点。`Queryable` 类定义了两种类似的方法 `Cast<TResult>(IQueryable)` 和 `OfType<TResult>(IQueryable)`，这两种方法都作用于类型 `Queryable` 的对象。

各个标准查询运算符在执行时间上有所不同，具体情况取决于它们是返回单一值还是值序列。返回单一实例值的这些方法（例如 `Average` 和 `Sum`）立即执行。返回序列的方法会延迟查询执行，并返回一个可枚举的对象。

对于在内存中集合上运行的方法（即扩展 `IEnumerable<T>` 的那些方法），返回的可枚举对象将捕获传递到方法的参数。在枚举该对象时，将使用查询运算符的逻辑，并返回查询结果。

相反，扩展 `IQueryable<T>` 的方法不会实现任何查询行为。它们生成一个表示要执行的查询的表达式树。源 `IQueryable<T>` 对象执行查询处理。

可以在一个查询中将对查询方法的调用链接在一起，这就使得查询的复杂性可能会变得不确定。

下面的代码示例演示如何使用标准查询运算符来获取有关序列的信息。

```
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
```

查询表达式语法

某些使用更频繁的标准查询运算符具有专用的 C# 和 Visual Basic 语言关键语语法，使用这些语法可以在“[查询表达式](#)”中调用这些运算符。有关具有专用关键字及其对应语法的标准查询运算符的详细信息，请参阅[标准查询运算符的查询表达式语法 \(C#\)](#)。

扩展标准查询运算符

通过创建适合于目标域或技术的特定于域的方法，可以增大标准查询运算符的集合。也可以用自己的实现来替换标准查询运算符，这些实现提供诸如远程计算、查询转换和优化之类的附加服务。有关示例，请参见[AsEnumerable](#)。

相关章节

通过以下链接可转到一些文章，这些文章基于功能提供有关各种标准查询运算符的附加信息。

[对数据进行排序 \(C#\)](#)

[集运算 \(C#\)](#)

[筛选数据 \(C#\)](#)

[限定符运算 \(C#\)](#)

[投影运算 \(C#\)](#)

[数据分区 \(C#\)](#)

[联接运算 \(C#\)](#)

[数据分组 \(C#\)](#)

[生成运算 \(C#\)](#)

[相等运算 \(C#\)](#)

[元素运算 \(C#\)](#)

[转换数据类型 \(C#\)](#)

[串联运算 \(C#\)](#)

[聚合运算 \(C#\)](#)

请参阅

- [Enumerable](#)
- [Queryable](#)
- [LINQ 查询简介 \(C#\)](#)
- [标准查询运算符的查询表达式语法 \(C#\)](#)
- [标准查询运算符按执行方式的分类 \(C#\)](#)
- [扩展方法](#)

标准查询运算符的查询表达式语法 (C#)

2020/11/2 • [Edit Online](#)

某些使用更频繁的标准查询运算符具有专用的 C# 语言关键字语法，使用这些语法可以在查询表达式中调用这些运算符。查询表达式是比基于方法的等效项更具可读性的另一种查询表示形式。查询表达式子句在编译时被转换为对查询方法的调用。

查询表达式语法表

下表列出包含等效查询表达式子句的标准查询运算符。

C#	C# 语句
Cast	使用显式类型化范围变量，例如： <pre>from int i in numbers</pre> (有关详细信息，请参阅 from 子句 。)
GroupBy	<pre>group ... by</pre> - 或 - <pre>group ... by ... into ...</pre> (有关详细信息，请参阅 group 子句 。)
GroupJoin<TOuter,TInner,TKey,TResult> (IEnumerable<TOuter>, IEnumerable<TInner>, Func<TOuter,TKey>, Func<TInner,TKey>, Func<TOuter,IEnumerable<TInner>,TResult>)	<pre>join ... in ... on ... equals ... into ...</pre> (有关详细信息，请参阅 join 子句 。)
Join<TOuter,TInner,TKey,TResult>(IEnumerable<TOuter>, IEnumerable<TInner>, Func<TOuter,TKey>, Func<TInner,TKey>, Func<TOuter,TInner,TResult>)	<pre>join ... in ... on ... equals ...</pre> (有关详细信息，请参阅 join 子句 。)
OrderBy<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>)	<pre>orderby</pre> (有关详细信息，请参阅 orderby 子句 。)
OrderByDescending<TSource,TKey> (IEnumerable<TSource>, Func<TSource,TKey>)	<pre>orderby ... descending</pre> (有关详细信息，请参阅 orderby 子句 。)
Select	<pre>select</pre> (有关详细信息，请参阅 let 子句 。)
SelectMany	多个 <pre>from</pre> 子句。 (有关详细信息，请参阅 from 子句 。)



C# ⓘ ⓘ ⓘ ⓘ ⓘ ⓘ

ThenBy<TSource,TKey>(IOrderedEnumerable<TSource>, Func<TSource,TKey>)

orderby ... , ...

(有关详细信息, 请参阅 [orderby 子句](#)。)

ThenByDescending<TSource,TKey>
(IOrderedEnumerable<TSource>, Func<TSource,TKey>)

orderby ... , ... descending

(有关详细信息, 请参阅 [orderby 子句](#)。)

Where

where

(有关详细信息, 请参阅 [where 子句](#)。)

请参阅

- [Enumerable](#)
- [Queryable](#)
- [标准查询运算符概述 \(C#\)](#)
- [标准查询运算符按执行方式的分类 \(C#\)](#)

标准查询运算符按执行方式的分类 (C#)

2020/11/2 • [Edit Online](#)

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。如果你了解不同查询运算符的执行方式，则有助于理解从给定查询中获得的结果。如果数据源是不断变化的，或者如果你要在另一个查询的基础上构建查询，这种帮助尤其明显。本主题根据标准查询运算符的执行方式对其进行分类。

执行方式

即时

立即执行指的是在代码中声明查询的位置读取数据源并执行运算。返回单个不可枚举的结果的所有标准查询运算符都立即执行。

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。这意味着，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。

使用延迟执行的查询运算符可以另外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将读取所有源数据，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。

NOTE

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。

立即	推迟	流式	非流式	未知
Aggregate	TSource	x		
All	Boolean	x		
Any	Boolean	x		
AsEnumerable	<code>IEnumerable<T></code>		x	

XXXXXX	XXXX	XXXX	XXXXXXXX	XXXXXXXXX
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		
Min	单个数值、TSource 或 TResult	X		
OfType	IEnumerable<T>		X	
OrderBy	IOrderedEnumerable<TElement>			X

XXXXXX	XXXX	XXXX	XXXXXXXX	XXXXXXXXX
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	

请参阅

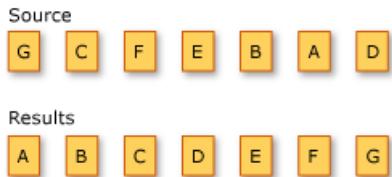
- [Enumerable](#)
- [标准查询运算符概述 \(C#\)](#)
- [标准查询运算符的查询表达式语法 \(C#\)](#)
- [LINQ to Objects \(C#\)](#)

对数据排序 (C#)

2020/11/2 • [Edit Online](#)

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。

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

方法	描述	C# 语法	类
OrderBy	按升序对值排序。	<code>orderby</code>	<code>Enumerable.OrderBy</code> <code>Queryable.OrderBy</code>
OrderByDescending	按降序对值排序。	<code>orderby ... descending</code>	<code>Enumerable.OrderByDescending</code> <code>Queryable.OrderByDescending</code>
ThenBy	按升序执行次要排序。	<code>orderby ..., ...</code>	<code>Enumerable.ThenBy</code> <code>Queryable.ThenBy</code>
ThenByDescending	按降序执行次要排序。	<code>orderby ..., ... descending</code>	<code>Enumerable.ThenByDescending</code> <code>Queryable.ThenByDescending</code>
Reverse	反转集合中元素的顺序。	不适用。	<code>Enumerable.Reverse</code> <code>Queryable.Reverse</code>

查询表达式语法示例

主要排序示例

主要升序排序

下面的示例演示如何在 LINQ 查询中使用 `orderby` 子句按字符串长度对数组中的字符串进行升序排序。

```
string[] words = { "the", "quick", "brown", "fox", "jumps" };

IQueryable<string> query = from word in words
                            orderby word.Length
                            select word;

foreach (string str in query)
    Console.WriteLine(str);

/* This code produces the following output:

the
fox
quick
brown
jumps
*/
```

主要降序排序

下面的示例演示如何在 LINQ 查询中使用 `orderby descending` 子句按字符串的第一个字母对字符串进行降序排序。

```
string[] words = { "the", "quick", "brown", "fox", "jumps" };

IQueryable<string> query = from word in words
                            orderby word.Substring(0, 1) descending
                            select word;

foreach (string str in query)
    Console.WriteLine(str);

/* This code produces the following output:

the
quick
jumps
fox
brown
*/
```

次要排序示例

次要升序排序

下面的示例演示如何在 LINQ 查询中使用 `orderby` 子句对数组中的字符串执行主要和次要排序。首先按字符串长度，其次按字符串的第一个字母，对字符串进行升序排序。

```
string[] words = { "the", "quick", "brown", "fox", "jumps" };

IQueryable<string> query = from word in words
                            orderby word.Length, word.Substring(0, 1)
                            select word;

foreach (string str in query)
    Console.WriteLine(str);

/* This code produces the following output:

    fox
    the
    brown
    jumps
    quick
*/
```

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。首先按字符串长度，其次按字符串的第一个字母，对字符串进行排序。

```
string[] words = { "the", "quick", "brown", "fox", "jumps" };

IQueryable<string> query = from word in words
                            orderby word.Length, word.Substring(0, 1) descending
                            select word;

foreach (string str in query)
    Console.WriteLine(str);

/* This code produces the following output:

    the
    fox
    quick
    jumps
    brown
*/
```

请参阅

- [System.Linq](#)
- [标准查询运算符概述 \(C#\)](#)
- [orderby 子句](#)
- [对 Join 子句的结果进行排序](#)
- [如何按任意词或字段对文本数据进行排序或筛选 \(LINQ\) \(C#\)](#)

集运算 (C#)

2020/11/2 • [Edit Online](#)

LINQ 中的集运算是指根据相同或不同集合(或集)中是否存在等效元素来生成结果集的查询运算。

下节列出了执行集运算的标准查询运算符方法。

方法

方法	描述	C# 语法	类
Distinct	删除集合中的重复值。	不适用。	Enumerable.Distinct Queryable.Distinct
Except	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 Queryable.Except
相交	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 Queryable.Intersect
联合	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 Queryable.Union

比较集运算

Distinct

以下示例演示字符序列上 [Enumerable.Distinct](#) 方法的行为。返回的序列包含输入序列的唯一元素。

a, b, b, c, d, c → a, b, c, d

```
string[] planets = { "Mercury", "Venus", "Venus", "Earth", "Mars", "Earth" };

IEnumerable<string> query = from planet in planets.Distinct()
                             select planet;

foreach (var str in query)
{
    Console.WriteLine(str);
}

/* This code produces the following output:
 *
 * Mercury
 * Venus
 * Earth
 * Mars
 */
```

Except

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。



```
string[] planets1 = { "Mercury", "Venus", "Earth", "Jupiter" };
string[] planets2 = { "Mercury", "Earth", "Mars", "Jupiter" };

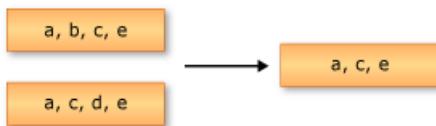
IQueryable<string> query = from planet in planets1.Except(planets2)
                           select planet;

foreach (var str in query)
{
    Console.WriteLine(str);
}

/* This code produces the following output:
 *
 * Venus
 */
```

相交

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。



```
string[] planets1 = { "Mercury", "Venus", "Earth", "Jupiter" };
string[] planets2 = { "Mercury", "Earth", "Mars", "Jupiter" };

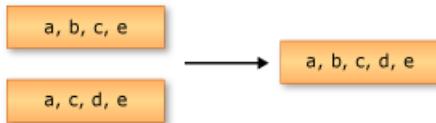
IQueryable<string> query = from planet in planets1.Intersect(planets2)
                           select planet;

foreach (var str in query)
{
    Console.WriteLine(str);
}

/* This code produces the following output:
 *
 * Mercury
 * Earth
 * Jupiter
 */
```

联合

以下示例演示对两个字符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。



```
string[] planets1 = { "Mercury", "Venus", "Earth", "Jupiter" };
string[] planets2 = { "Mercury", "Earth", "Mars", "Jupiter" };

IEnumerable<string> query = from planet in planets1.Union(planets2)
                             select planet;

foreach (var str in query)
{
    Console.WriteLine(str);
}

/* This code produces the following output:
 *
 * Mercury
 * Venus
 * Earth
 * Jupiter
 * Mars
 */
```

请参阅

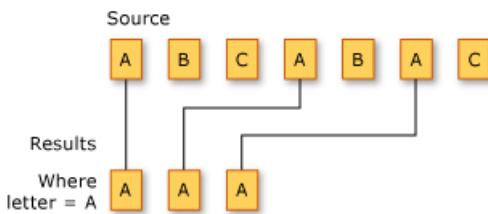
- [System.Linq](#)
- [标准查询运算符概述 \(C#\)](#)
- [如何合并和比较字符串集合 \(LINQ\) \(C#\)](#)
- [如何查找两个列表之间的差集 \(LINQ\) \(C#\)](#)

筛选数据 (C#)

2020/11/2 • [Edit Online](#)

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。它也称为选定内容。

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”。



下一节列出了执行所选内容的标准查询运算符方法。

方法

方法	描述	C# 构造函数	接口
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

查询表达式语法示例

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。

```
string[] words = { "the", "quick", "brown", "fox", "jumps" };

IQueryable<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
*/
```

请参阅

- [System.Linq](#)
- [标准查询运算符概述 \(C#\)](#)
- [where 子句](#)
- [在运行时动态指定谓词筛选器](#)

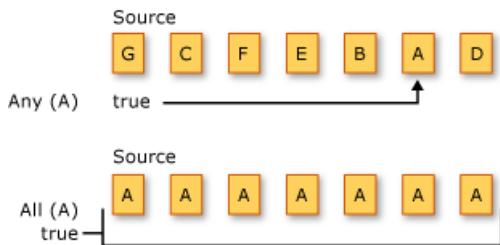
- 如何使用反射查询程序集的元数据 (LINQ) (C#)
- 如何查询具有指定特性或名称的文件 (C#)
- 如何按任意词或字段对文本数据进行排序或筛选 (LINQ) (C#)

限定符运算 (C#)

2020/11/2 • [Edit Online](#)

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。

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一个或多个元素为字符“A”，结果为 `true`。第二个运算询问是否所有元素都为字符“A”，结果为 `true`。



下节列出了执行限定符运算的标准查询运算符方法。

方法

概念	说明	C# 例程	接口
全部	确定是否序列中的所有元素都满足条件。	不适用。	<code>Enumerable.All</code> <code>Queryable.All</code>
任意	确定序列中是否有元素满足条件。	不适用。	<code>Enumerable.Any</code> <code>Queryable.Any</code>
包含	确定序列是否包含指定的元素。	不适用。	<code>Enumerable.Contains</code> <code>Queryable.Contains</code>

查询表达式语法示例

全部

以下示例使用 `All` 检查所有字符串是否为特定长度。

```

class Market
{
    public string Name { get; set; }
    public string[] Items { get; set; }
}

public static void Example()
{
    List<Market> markets = new List<Market>
    {
        new Market { Name = "Emily's", Items = new string[] { "kiwi", "cheery", "banana" } },
        new Market { Name = "Kim's", Items = new string[] { "melon", "mango", "olive" } },
        new Market { Name = "Adam's", Items = new string[] { "kiwi", "apple", "orange" } },
    };

    // Determine which market have all fruit names length equal to 5
    IEnumerable<string> names = from market in markets
                                where market.Items.All(item => item.Length == 5)
                                select market.Name;

    foreach (string name in names)
    {
        Console.WriteLine($"{name} market");
    }

    // This code produces the following output:
    //
    // Kim's market
}

```

任意

以下示例使用 `Any` 检查所有字符串是否以“o”开头。

```

class Market
{
    public string Name { get; set; }
    public string[] Items { get; set; }
}

public static void Example()
{
    List<Market> markets = new List<Market>
    {
        new Market { Name = "Emily's", Items = new string[] { "kiwi", "cheery", "banana" } },
        new Market { Name = "Kim's", Items = new string[] { "melon", "mango", "olive" } },
        new Market { Name = "Adam's", Items = new string[] { "kiwi", "apple", "orange" } },
    };

    // Determine which market have any fruit names start with 'o'
    IEnumerable<string> names = from market in markets
                                where market.Items.Any(item => item.StartsWith("o"))
                                select market.Name;

    foreach (string name in names)
    {
        Console.WriteLine($"{name} market");
    }

    // This code produces the following output:
    //
    // Kim's market
    // Adam's market
}

```

包含

以下示例使用 `Contains` 检查所有数组是否具有特定元素。

```
class Market
{
    public string Name { get; set; }
    public string[] Items { get; set; }
}

public static void Example()
{
    List<Market> markets = new List<Market>
    {
        new Market { Name = "Emily's", Items = new string[] { "kiwi", "cheery", "banana" } },
        new Market { Name = "Kim's", Items = new string[] { "melon", "mango", "olive" } },
        new Market { Name = "Adam's", Items = new string[] { "kiwi", "apple", "orange" } },
    };

    // Determine which market contains fruit names equal 'kiwi'
    IEnumerable<string> names = from market in markets
                                  where market.Items.Contains("kiwi")
                                  select market.Name;

    foreach (string name in names)
    {
        Console.WriteLine($"{name} market");
    }

    // This code produces the following output:
    //
    // Emily's market
    // Adam's market
}
```

请参阅

- [System.Linq](#)
- [标准查询运算符概述 \(C#\)](#)
- [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 \(LINQ\) \(C#\)](#)

投影运算 (C#)

2020/11/2 • [Edit Online](#)

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。

下面一节列出了执行投影的标准查询运算符方法。

方法

描述	示例	C# 代码示例	方法
选择	投影基于转换函数的值。	<code>select</code>	<code>Enumerable.Select</code> <code>Queryable.Select</code>
<code>SelectMany</code>	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 <code>from</code> 子句	<code>Enumerable.SelectMany</code> <code>Queryable.SelectMany</code>

查询表达式语法示例

选择

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。

```
List<string> words = new List<string>() { "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
*/
```

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。

```

List<string> phrases = new List<string>() { "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
*/

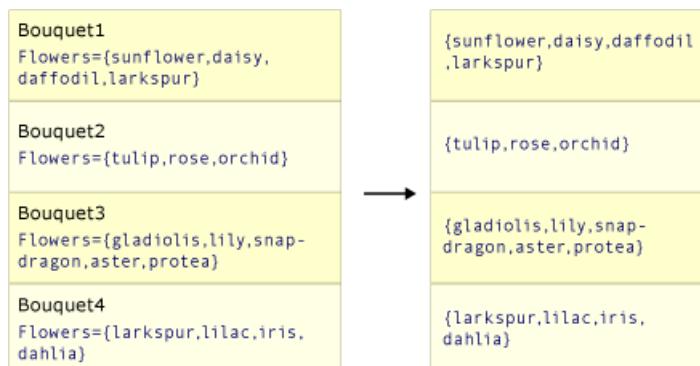
```

Select 与 SelectMany

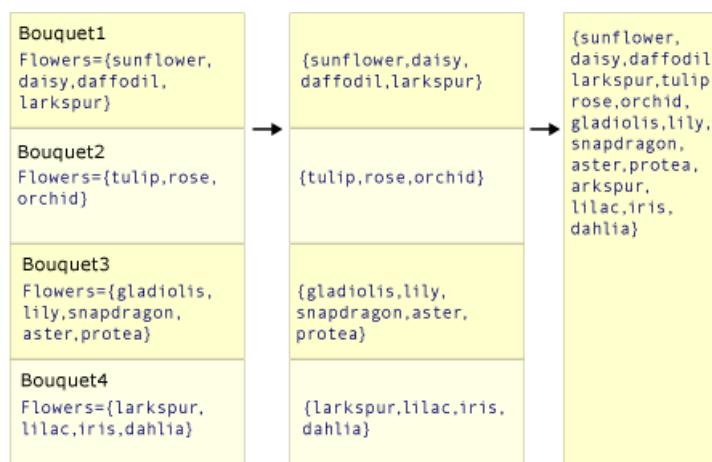
`Select()` 和 `SelectMany()` 的工作都是依据源值生成一个或多个结果值。`Select()` 为每个源值生成一个结果值。因此，总体结果是一个与源集合具有相同元素数目的集合。与之相反，`SelectMany()` 生成单个总体结果，其中包含来自每个源值的串联子集合。作为参数传递到 `SelectMany()` 的转换函数必须为每个源值返回一个可枚举序列。然后，`SelectMany()` 串联这些可枚举序列，以创建一个大的序列。

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据。

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前两项来创建一个“花束”。此示例中, `transform` 函数 `Select<TSource,TResult>(IEnumerable<TSource>, Func<TSource,TResult>)` 使用的“单值”本身即是值的集合。这需要额外的 `foreach` 循环, 以便枚举每个子序列中的每个字符串。

```
class Bouquet
{
    public List<string> Flowers { get; set; }
}

static void SelectVsSelectMany()
{
    List<Bouquet> bouquets = new List<Bouquet>() {
        new Bouquet { Flowers = new List<string> { "sunflower", "daisy", "daffodil", "larkspur" } },
        new Bouquet{ Flowers = new List<string> { "tulip", "rose", "orchid" } },
        new Bouquet{ Flowers = new List<string> { "gladiolus", "lily", "snapdragon", "aster", "protea" } },
        new Bouquet{ Flowers = new List<string> { "larkspur", "lilac", "iris", "dahlia" } }
    };

    // ***** Select *****
    IEnumerable<List<string>> query1 = bouquets.Select(bq => bq.Flowers);

    // ***** SelectMany *****
    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);

    /* This code produces the following output:

    Results by using Select():
    sunflower
    daisy
    daffodil
    larkspur
    tulip
    rose
    orchid
    gladiolus
    lily
    snapdragon
    aster
    protea
    larkspur
    lilac
    iris
    dahlia

    Results by using SelectMany():
    sunflower
    daisy
    daffodil
    larkspur
    tulip
    rose
    orchid
    gladiolus
    lily
    */
}
```

```
snapdragon  
aster  
protea  
larkspur  
lilac  
iris  
dahlia  
*/  
}
```

请参阅

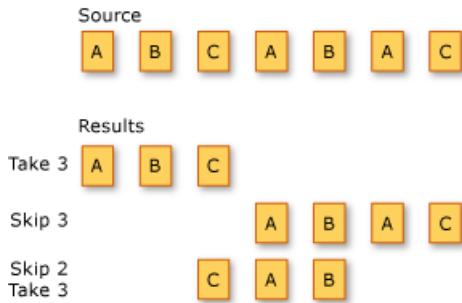
- [System.Linq](#)
- [标准查询运算符概述 \(C#\)](#)
- [select 子句](#)
- [如何从多个源填充对象集合 \(LINQ\) \(C#\)](#)
- [如何使用组将一个文件拆分成多个文件 \(LINQ\) \(C#\)](#)

数据分区 (C#)

2020/11/2 • [Edit Online](#)

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。

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

		C#	
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
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

请参阅

- [System.Linq](#)
- [标准查询运算符概述 \(C#\)](#)

Join 操作 (C#)

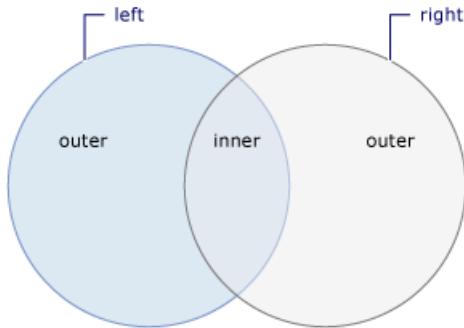
2020/11/2 • [Edit Online](#)

联接两个数据源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。

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, Join 就成为一项重要的运算。在面向对象的编程中, 这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: Customer 类有一个类型为 City 的属性, 但 City 类没有作为 Customer 对象集合的属性。如果你具有一个 City 对象列表, 并且要查找每个城市中的所有客户, 则可以使用联接运算完成此项查找。

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。(与此相较, Transact-SQL 支持除“等于”之外的联接运算符, 例如“小于”运算符。)用关系数据库术语表达, 就是说 [Join](#) 实现了内部联接, 这种联接只返回那些在另一个数据集中具有匹配项的对象。[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。左外部联接是指返回第一个(左侧)数据源的每个元素的联接, 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联接或左外部联接中的元素。



方法

CLR	IL	C#	ECMA
Join	根据键选择器函数 Join 两个序列并提取值对。	<code>join ... in ... on ... equals ...</code>	Enumerable.Join Queryable.Join
GroupJoin	根据键选择器函数 Join 两个序列, 并对每个元素的结果匹配项进行分组。	<code>join ... in ... on ... equals ... into ...</code>	Enumerable.GroupJoin Queryable.GroupJoin

查询表达式语法示例

Join

下面的示例使用 `join ... in ... on ... equals ...` 子句基于特定值联接两个序列:

```

class Product
{
    public string Name { get; set; }
    public int CategoryId { get; set; }
}

class Category
{
    public int Id { get; set; }
    public string CategoryName { get; set; }
}

public static void Example()
{
    List<Product> products = new List<Product>
    {
        new Product { Name = "Cola", CategoryId = 0 },
        new Product { Name = "Tea", CategoryId = 0 },
        new Product { Name = "Apple", CategoryId = 1 },
        new Product { Name = "Kiwi", CategoryId = 1 },
        new Product { Name = "Carrot", CategoryId = 2 },
    };

    List<Category> categories = new List<Category>
    {
        new Category { Id = 0, CategoryName = "Beverage" },
        new Category { Id = 1, CategoryName = "Fruit" },
        new Category { Id = 2, CategoryName = "Vegetable" }
    };

    // Join products and categories based on CategoryId
    var query = from product in products
               join category in categories on product.CategoryId equals category.Id
               select new { product.Name, category.CategoryName };

    foreach (var item in query)
    {
        Console.WriteLine($"{item.Name} - {item.CategoryName}");
    }

    // This code produces the following output:
    //
    // Cola - Beverage
    // Tea - Beverage
    // Apple - Fruit
    // Kiwi - Fruit
    // Carrot - Vegetable
}

```

GroupJoin

下面的示例使用 `join ... in ... on ... equals ... into ...` 子句基于特定值联接两个序列，并对每个元素的结果匹配项进行分组：

```

class Product
{
    public string Name { get; set; }
    public int CategoryId { get; set; }
}

class Category
{
    public int Id { get; set; }
    public string CategoryName { get; set; }
}

public static void Example()
{
    List<Product> products = new List<Product>
    {
        new Product { Name = "Cola", CategoryId = 0 },
        new Product { Name = "Tea", CategoryId = 0 },
        new Product { Name = "Apple", CategoryId = 1 },
        new Product { Name = "Kiwi", CategoryId = 1 },
        new Product { Name = "Carrot", CategoryId = 2 },
    };

    List<Category> categories = new List<Category>
    {
        new Category { Id = 0, CategoryName = "Beverage" },
        new Category { Id = 1, CategoryName = "Fruit" },
        new Category { Id = 2, CategoryName = "Vegetable" }
    };

    // Join categories and product based on CategoryId and grouping result
    var productGroups = from category in categories
                        join product in products on category.Id equals product.CategoryId into productGroup
                        select productGroup;

    foreach (IEnumerable<Product> productGroup in productGroups)
    {
        Console.WriteLine("Group");
        foreach (Product product in productGroup)
        {
            Console.WriteLine($"{product.Name,8}");
        }
    }

    // This code produces the following output:
    //
    // Group
    //     Cola
    //     Tea
    // Group
    //     Apple
    //     Kiwi
    // Group
    //     Carrot
}

```

请参阅

- [System.Linq](#)
- [标准查询运算符概述 \(C#\)](#)
- [匿名类型](#)
- [构建 Join 和跨产品查询](#)
- [join 子句](#)

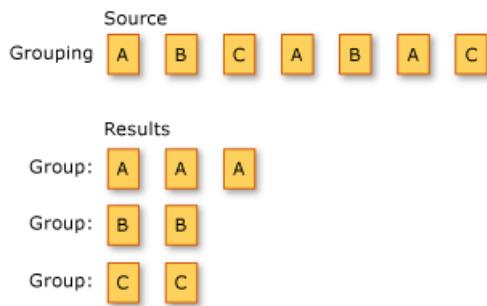
- [使用组合键 Join](#)
- [如何联接不同文件的内容 \(LINQ\) \(C#\)](#)
- [对 Join 子句的结果进行排序](#)
- [执行自定义联接操作](#)
- [执行分组联接](#)
- [执行内部联接](#)
- [执行左外部联接](#)
- [如何从多个源填充对象集合 \(LINQ\) \(C#\)](#)

对数据分组 (C#)

2020/11/2 • [Edit Online](#)

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。



下一节列出了对数据元素进行分组的标准查询运算符方法。

方法

类	方法	C# 例句	方法
GroupBy	对共享通用属性的元素进行分组。每组由一个 IGrouping< TKey, TElement > 对象表示。	<code>group ... by</code> - 或 - <code>group ... by ... into ...</code>	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

查询表达式语法示例

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。

```
List<int> numbers = new List<int>() { 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);
}

/* This code produces the following output:

Odd numbers:
35
3987
199
329

Even numbers:
44
200
84
4
446
208
*/
```

请参阅

- [System.Linq](#)
- [标准查询运算符概述 \(C#\)](#)
- [group 子句](#)
- [创建嵌套组](#)
- [如何按扩展名对文件进行分组 \(LINQ\) \(C#\)](#)
- [对查询结果进行分组](#)
- [对分组操作执行子查询](#)
- [如何使用组将一个文件拆分成多个文件 \(LINQ\) \(C#\)](#)

生成运算 (C#)

2020/11/2 • [Edit Online](#)

生成是指创建新的值序列。

下面一节列出了执行生成的标准查询运算符方法。

方法

方法	描述	C# 语法	类
DefaultIfEmpty	用默认值单一实例集合替换空集合。	不适用。	Enumerable.DefaultIfEmpty Queryable.DefaultIfEmpty
空	返回一个空集合。	不适用。	Enumerable.Empty
范围	生成包含数字序列的集合。	不适用。	Enumerable.Range
Repeat	生成包含一个重复值的集合。	不适用。	Enumerable.Repeat

请参阅

- [System.Linq](#)
- [标准查询运算符概述 \(C#\)](#)

相等运算 (C#)

2020/11/2 • [Edit Online](#)

两个序列，其相应元素相等且具有被视为相等的相同数量的元素。

方法

方法	描述	C# 例程	泛型参数
<code>SequenceEqual</code>	通过以成对方式比较元素确定两个序列是否相等。	不适用。	<code>Enumerable.SequenceEqual</code> <code>Queryable.SequenceEqual</code>

请参阅

- [System.Linq](#)
- [标准查询运算符概述 \(C#\)](#)
- [如何比较两个文件夹的内容 \(LINQ\) \(C#\)](#)

元素运算 (C#)

2020/11/2 • [Edit Online](#)

元素运算从序列中返回唯一、特定的元素。

下节列出了执行元素运算的标准查询运算符方法。

方法

方法	说明	C# API	实现
ElementAt	返回集合中指定索引处的元素。	不适用。	Enumerable.ElementAt Queryable.ElementAt
ElementAtOrDefault	返回集合中指定索引处的元素;如果索引超出范围,则返回默认值。	不适用。	Enumerable.ElementAtOrDefault Queryable.ElementAtOrDefault
First	返回集合的第一个元素或满足条件的第一个元素。	不适用。	Enumerable.First Queryable.First
FirstOrDefault	返回集合的第一个元素或满足条件的第一个元素。如果此类元素不存在,则返回默认值。	不适用。	Enumerable.FirstOrDefault Queryable.FirstOrDefault Queryable.FirstOrDefault<TSource> (IQueryable<TSource>)
上一个	返回集合的最后一个元素或满足条件的最后一个元素。	不适用。	Enumerable.Last Queryable.Last
LastOrDefault	返回集合的最后一个元素或满足条件的最后一个元素。如果此类元素不存在,则返回默认值。	不适用。	Enumerable.LastOrDefault Queryable.LastOrDefault
Single	返回集合的唯一一个元素或满足条件的唯一一个元素。如果没有要返回的元素或要返回多个元素,则引发 InvalidOperationException 。	不适用。	Enumerable.Single Queryable.Single
SingleOrDefault	返回集合的唯一一个元素或满足条件的唯一一个元素。如果没有要返回的元素,则返回默认值。如果要返回多个元素,则引发 InvalidOperationException 。	不适用。	Enumerable.SingleOrDefault Queryable.SingleOrDefault

请参阅

- [System.Linq](#)
- [标准查询运算符概述 \(C#\)](#)
- [如何查询目录树中的一个或多个最大的文件 \(LINQ\) \(C#\)](#)

转换数据类型 (C#)

2020/11/2 • [Edit Online](#)

转换方法可更改输入对象的类型。

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。

方法	描述	C# 代码示例	方法
<code>AsEnumerable</code>	返回类型化为 <code>IEnumerable<T></code> 的输入。	不适用。	<code>Enumerable.AsEnumerable</code>
<code>AsQueryable</code>	将(泛型) <code>IEnumerable</code> 转换为(泛型) <code>IQueryable</code> 。	不适用。	<code>Queryable.AsQueryable</code>
<code>Cast</code>	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： <pre>from string str in words</pre>	<code>Enumerable.Cast</code> <code>Queryable.Cast</code>
<code>OfType</code>	根据其转换为指定类型的能力筛选值。	不适用。	<code>Enumerable.OfType</code> <code>Queryable.OfType</code>
<code>ToArray</code>	将集合转换为数组。此方法强制执行查询。	不适用。	<code>Enumerable.ToArray</code>
<code>ToDictionary</code>	根据键选择器函数将元素放入 <code>Dictionary< TKey, TValue ></code> 。此方法强制执行查询。	不适用。	<code>Enumerable.ToDictionary</code>
<code>ToList</code>	将集合转换为 <code>List< T ></code> 。此方法强制执行查询。	不适用。	<code>Enumerable.ToList</code>

方法	描述	C# 代码示例	说明
ToLookup	根据键选择器函数将元素放入 <code>Lookup< TKey, TElement ></code> (一对多字典)。此方法强制执行查询。	不适用。	<code>Enumerable.ToLookup</code>

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。

```
class Plant
{
    public string Name { get; set; }
}

class CarnivorousPlant : Plant
{
    public string TrapType { get; set; }
}

static void Cast()
{
    Plant[] plants = new Plant[] {
        new CarnivorousPlant { Name = "Venus Fly Trap", TrapType = "Snap Trap" },
        new CarnivorousPlant { Name = "Pitcher Plant", TrapType = "Pitfall Trap" },
        new CarnivorousPlant { Name = "Sundew", TrapType = "Flypaper Trap" },
        new CarnivorousPlant { Name = "Waterwheel Plant", TrapType = "Snap Trap" }
    };

    var query = from CarnivorousPlant cPlant in plants
                where cPlant.TrapType == "Snap Trap"
                select cPlant;

    foreach (Plant plant in query)
        Console.WriteLine(plant.Name);

    /* This code produces the following output:
     *
     * Venus Fly Trap
     * Waterwheel Plant
     */
}
```

请参阅

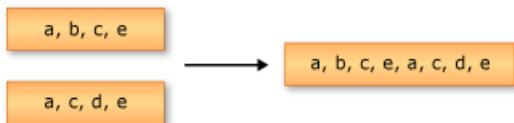
- [System.Linq](#)
- [标准查询运算符概述 \(C#\)](#)
- [from 子句](#)
- [LINQ 查询表达式](#)
- [如何使用 LINQ 查询 ArrayList \(C#\)](#)

串联运算 (C#)

2020/11/2 • [Edit Online](#)

串联是指将一个序列附加到另一个序列的操作。

下图描绘了两个字符序列的串联操作。



下面一节列出了执行串联的标准查询运算符方法。

方法

类	接口	C# 语法	扩展方法
Concat		连接两个序列以组成一个序列。	不适用。 Enumerable.Concat Queryable.Concat

请参阅

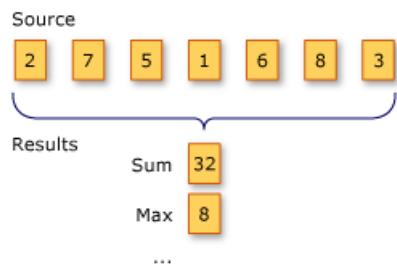
- [System.Linq](#)
- [标准查询运算符概述 \(C#\)](#)
- [如何合并和比较字符串集合 \(LINQ\) \(C#\)](#)

聚合运算 (C#)

2020/11/2 • [Edit Online](#)

聚合运算从值的集合中计算出单个值。例如，从一个月累计的每日温度值计算出日平均温度值就是一个聚合运算。

下图显示对数字序列进行两种不同聚合操作所得结果。第一个操作累加数字。第二个操作返回序列中的最大值。



下节列出了执行聚合运算的标准查询运算符方法。

方法

方法	描述	C# 构造函数	示例
聚合	对集合的值执行自定义聚合运算。	不适用。	Enumerable.Aggregate Queryable.Aggregate
平均值	计算值集合的平均值。	不适用。	Enumerable.Average Queryable.Average
计数	对集合中元素计数，可选择仅对满足谓词函数的元素计数。	不适用。	Enumerable.Count Queryable.Count
LongCount	对大型集合中元素计数，可选择仅对满足谓词函数的元素计数。	不适用。	Enumerable.LongCount Queryable.LongCount
最大值	确定集合中的最大值。	不适用。	Enumerable.Max Queryable.Max
最小值	确定集合中的最小值。	不适用。	Enumerable.Min Queryable.Min
Sum	对集合中的值求和。	不适用。	Enumerable.Sum Queryable.Sum

请参阅

- [System.Linq](#)
- [标准查询运算符概述 \(C#\)](#)
- [如何在 CSV 文本文件中计算列值 \(LINQ\) \(C#\)](#)
- [如何查询目录树中的一个或多个最大的文件 \(LINQ\) \(C#\)](#)
- [如何查询一组文件夹中的总字节数 \(LINQ\) \(C#\)](#)

LINQ to Objects (C#)

2020/11/2 • [Edit Online](#)

术语“LINQ to Objects”指直接将 LINQ 查询与任何 `IEnumerable` 或 `IEnumerable<T>` 集合一起使用，而不使用中间 LINQ 提供程序或 API，例如 [LINQ to SQL](#) 或 [LINQ to XML](#)。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 返回的集合。

从根本上说，“LINQ to Objects”表示一种新的处理集合的方法。采用旧方法，必须编写指定如何从集合检索数据的复杂的 `foreach` 循环。而采用 LINQ 方法，只需编写描述要检索的内容的声明性代码。

此外，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。

通常，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。

本节的目的是使用一些精选示例来演示 LINQ 方法。并不打算详尽说明。

本节内容

[LINQ 和字符串 \(C#\)](#)

阐释如何使用 LINQ 来查询和转换字符串和字符串集合。还包括指向演示这些原则的文章的链接。

[LINQ 和反射 \(C#\)](#)

指向演示 LINQ 如何使用反射的示例的链接。

[LINQ 和文件目录 \(C#\)](#)

阐释如何使用 LINQ 来与文件系统进行交互。还包括指向演示这些概念的文章的链接。

[如何使用 LINQ 查询 ArrayList \(C#\)](#)

演示如何使用 C# 查询 `ArrayList`。

[如何为 LINQ 查询添加自定义方法 \(C#\)](#)

阐释如何通过向 `IEnumerable<T>` 接口中添加扩展方法来扩展可用于 LINQ 查询的方法集。

[语言集成查询 \(LINQ\) \(C#\)](#)

提供指向阐释 LINQ 并提供执行查询的代码示例的文章的链接。

LINQ 和字符串 (C#)

2020/11/2 • [Edit Online](#)

LINQ 可用于查询和转换字符串和字符串集合。这在处理文本文件中的半结构化数据时尤其有用。LINQ 查询可以与传统的字符串函数和正则表达式合并。例如，可以使用 [String.Split](#) 或 [Regex.Split](#) 方法来创建可稍后使用 LINQ 查询或修改的字符串数组。可以使用 LINQ 查询的 `where` 子句中的 [Regex.IsMatch](#) 方法。并且可以使用 LINQ 查询或修改正则表达式返回的 [MatchCollection](#) 结果。

还可以使用本节介绍的技术将半结构化的文本数据转换为 XML。有关详细信息，请参阅[如何从 CSV 文件生成 XML](#)。

本节中的示例分为两类：

查询文本块

可以使用 [String.Split](#) 方法或 [Regex.Split](#) 方法将文本块拆分为可查询的较小字符串数组，从而对其进行查询、分析和修改。可以先将源文本拆分为词语、句、段落、页或任何其他条件，然后根据查询的需要执行其他拆分。

- [如何对某个词在字符串中出现的次数进行计数 \(LINQ\) \(C#\)](#)

演示如何使用 LINQ 进行简单文本查询。

- [如何查询包含一组指定词语的句子 \(LINQ\) \(C#\)](#)

演示如何在任意边界上拆分文本文件以及如何针对每个部分执行查询。

- [如何查询字符串中的字符 \(LINQ\) \(C#\)](#)

演示字符串是可查询类型。

- [如何将 LINQ 查询与正则表达式合并 \(C#\)](#)

演示如何在 LINQ 查询中使用正则表达式，以便对筛选的查询结果进行复杂的模式匹配。

查询文本格式的半结构化数据

许多不同类型的文本文件都包含一系列行，通常具有类似的格式设置，例如制表符分隔或逗号分隔的文件或固定长度的行。将此类文本文件读入内存后，可以使用 LINQ 来查询和/或修改其中的行。LINQ 查询还简化了合并来自多个源的数据的任务。

- [如何查找两个列表之间的差集 \(LINQ\) \(C#\)](#)

演示如何查找出现在一个列表中、但没有出现在另一个列表中的所有字符串。

- [如何按任意词或字段对文本数据进行排序或筛选 \(LINQ\) \(C#\)](#)

演示如何基于任意词或字段对文本行进行排序。

- [如何重新排列带分隔符的文件的字段 \(LINQ\) \(C#\)](#)

演示如何对 .csv 文件的某行中的字段进行重新排序。

- [如何合并和比较字符串集合 \(LINQ\) \(C#\)](#)

演示如何通过各种方式合并字符串列表。

- [如何从多个源填充对象集合 \(LINQ\) \(C#\)](#)

演示如何将多个文本文件作为数据源来创建对象集合。

- [如何联接不同文件的内容 \(LINQ\) \(C#\)](#)

演示如何使用匹配键将两个列表中的字符串合并成单个字符串。

- [如何使用组将一个文件拆分成多个文件 \(LINQ\) \(C#\)](#)

演示如何通过将单个文件用作数据源来创建新文件。

- [如何在 CSV 文本文件中计算列值 \(LINQ\) \(C#\)](#)

演示如何在 .csv 文件中对文本数据执行数学计算。

请参阅

- [语言集成查询 \(LINQ\) \(C#\)](#)

- [如何从 CSV 文件生成 XML](#)

如何对某个词在字符串中出现的次数进行计数 (LINQ) (C#)

2021/5/7 • [Edit Online](#)

此示例演示如何使用 LINQ 查询对指定词在字符串中出现的次数进行计数。请注意，若要执行计数，首先需调用 [Split](#) 方法来创建词数组。[Split](#) 方法存在性能开销。如果只需要统计字符串的字数，则应考虑改用 [Matches](#) 或 [IndexOf](#) 方法。但是，如果性能不是关键问题，或者已拆分句子以对其执行其他类型的查询，则使用 LINQ 来计数词或短语同样有意义。

示例

```
class CountWords
{
    static void Main()
    {
        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
        string[] source = text.Split(new char[] { '.', '?', '!', ' ', ';' , ':' , ',' },
            StringSplitOptions.RemoveEmptyEntries);

        // Create the query. Use ToLowerInvariant to match "data" and "Data"
        var matchQuery = from word in source
                        where word.ToLowerInvariant() == searchTerm.ToLowerInvariant()
                        select word;

        // Count the matches, which executes the query.
        int wordCount = matchQuery.Count();
        Console.WriteLine("{0} occurrence(s) of the search term \"{1}\" were found.", wordCount,
            searchTerm);

        // Keep console window open in debug mode
        Console.WriteLine("Press any key to exit");
        Console.ReadKey();
    }
}
/* Output:
   3 occurrence(s) of the search term "data" were found.
```

编译代码

使用 `System.Linq` 和 `System.IO` 命名空间的 `using` 指令创建 C# 控制台应用程序项目。

请参阅

- [LINQ 和字符串 \(C#\)](#)

如何查询包含一组指定词语的句子 (LINQ) (C#)

2021/5/7 • [Edit Online](#)

此示例演示如何在包含一组指定的词语的每个匹配项的文本文件中查找句子。尽管此示例中的搜索词数组采用硬编码形式，但它也可在运行时以动态方式进行填充。在此示例中，查询将返回包含单词“Historically”、“data,”和“integrated”的句子。

示例

```
class FindSentences
{
    static void Main()
    {
        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(new char[] { '.', '?', '!' });

        // Define the search terms. This list could also be dynamically populated at runtime.
        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.
        var sentenceQuery = from sentence in sentences
                            let w = sentence.Split(new char[] { '.', '?', '!', ' ', ';' , ':' , ',' }, StringSplitOptions.RemoveEmptyEntries)
                            where w.Distinct().Intersect(wordsToMatch).Count() == wordsToMatch.Count()
                            select sentence;

        // Execute the query. Note that you can explicitly type
        // the iteration variable here even though sentenceQuery
        // was implicitly typed.
        foreach (string str in sentenceQuery)
        {
            Console.WriteLine(str);
        }

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit");
        Console.ReadKey();
    }
}
/* Output:
Historically, the world of data and the world of objects have not been well integrated
*/
```

查询运行时首先将文本拆分成句子，然后将句子拆分成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。

在对 `Split` 的调用中，使用标点符号作为分隔符，以从字符串中删除标点符号。如果你没有这样做，则假如你有一个字符串“Historically”，该字符串不会与 `wordsToMatch` 数组中的“Historically”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。

编译代码

使用 `System.Linq` 和 `System.IO` 命名空间的 `using` 指令创建 C# 控制台应用程序项目。

请参阅

- [LINQ 和字符串 \(C#\)](#)

如何查询字符串中的字符 (LINQ) (C#)

2021/5/7 • [Edit Online](#)

因为 `String` 类可实现泛型 `IEnumerable<T>` 接口，因此任何字符串都可以字符序列的形式进行查询。但是，这不是 LINQ 的一般用法。对于复杂的模式匹配操作，请使用 `Regex` 类。

示例

以下示例查询一个字符串以确定它所包含的数字数量。请注意，在第一次执行此查询后将“重用”此查询。这是可能的，因为查询本身并不存储任何实际的结果。

```
class QueryAString
{
    static void Main()
    {
        string aString = "ABCDE99F-J74-12-89A";

        // Select only those characters that are numbers
        IEnumerable<char> 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 = {0}", count);

        // Select all characters before the first '-'
        IEnumerable<char> stringQuery2 = aString.TakeWhile(c => c != '-');

        // Execute the second query
        foreach (char c in stringQuery2)
            Console.Write(c);

        Console.WriteLine(System.Environment.NewLine + "Press any key to exit");
        Console.ReadKey();
    }
}
/* Output:
   Output: 9 9 7 4 1 2 8 9
   Count = 8
   ABCDE99F
*/
```

编译代码

使用 `System.Linq` 和 `System.IO` 命名空间的 `using` 指令创建 C# 控制台应用程序项目。

请参阅

- [LINQ 和字符串 \(C#\)](#)
- [如何将 LINQ 查询与正则表达式合并 \(C#\)](#)

如何将 LINQ 查询与正则表达式合并 (C#)

2021/5/7 • [Edit Online](#)

此示例演示如何使用 [Regex](#) 类在文本字符串中为更复杂的匹配创建正则表达式。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。

示例

```
class QueryWithRegEx
{
    public static void Main()
    {
        // Modify this path as necessary so that it accesses your version of Visual Studio.
        string startFolder = @"C:\Program Files (x86)\Microsoft Visual Studio 14.0\";
        // One of the following paths may be more appropriate on your computer.
        //string startFolder = @"C:\Program Files (x86)\Microsoft Visual Studio\2017\";

        // Take a snapshot of the file system.
        IEnumerable<System.IO.FileInfo> fileList = GetFiles(startFolder);

        // Create the regular expression to find all things "Visual".
        System.Text.RegularExpressions.Regex searchTerm =
            new System.Text.RegularExpressions.Regex(@"Visual (Basic|C#|C\+\+|Studio)");

        // 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 == ".htm"
            let fileText = System.IO.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 \"{0}\" was found in:", searchTerm.ToString());

        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);
            }
        }
    }
}
```

```
// Keep the console window open in debug mode
Console.WriteLine("Press any key to exit");
Console.ReadKey();
}

// This method assumes that the application has discovery
// permissions for all folders under the specified path.
static IEnumerable<System.IO.FileInfo> GetFiles(string path)
{
    if (!System.IO.Directory.Exists(path))
        throw new System.IO.DirectoryNotFoundException();

    string[] fileNames = null;
    List<System.IO.FileInfo> files = new List<System.IO.FileInfo>();

    fileNames = System.IO.Directory.GetFiles(path, "*.*", System.IO.SearchOption.AllDirectories);
    foreach (string name in fileNames)
    {
        files.Add(new System.IO.FileInfo(name));
    }
    return files;
}
}
```

请注意，还可以查询 `RegEx` 搜索返回的 `MatchCollection` 对象。在本例中，只在结果中生成每个匹配项的值。但是，也可以使用 LINQ 对集合执行筛选、排序和分组等各种操作。由于 `MatchCollection` 为非泛型 `IEnumerable` 集合，所以必须显式声明查询中范围变量的类型。

编译代码

使用 `System.Linq` 和 `System.IO` 命名空间的 `using` 指令创建 C# 控制台应用程序项目。

请参阅

- [LINQ 和字符串 \(C#\)](#)
- [LINQ 和文件目录 \(C#\)](#)

如何查找两个列表之间的差集 (LINQ) (C#)

2021/5/7 • [Edit Online](#)

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 names1.txt 中但不在 names2.txt 中的行。

创建数据文件

1. 按照[如何合并和比较字符串集合 \(LINQ\) \(C#\)](#)中的说明，将 names1.txt 和 names2.txt 复制到解决方案文件夹中。

示例

```
class CompareLists
{
    static void Main()
    {
        // Create the I Enumerable data sources.
        string[] names1 = System.IO.File.ReadAllLines(@"../../names1.txt");
        string[] names2 = System.IO.File.ReadAllLines(@"../../names2.txt");

        // Create the query. Note that method syntax must be used here.
        IEnumerable<string> 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);

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit");
        Console.ReadKey();
    }
}
/* 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
*/
```

C# 中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。

编译代码

使用 `System.Linq` 和 `System.IO` 命名空间的 `using` 指令创建 C# 控制台应用程序项目。

请参阅

- [LINQ 和字符串 \(C#\)](#)

如何按任意词或字段对文本数据进行排序或筛选 (LINQ) (C#)

2021/5/7 • [Edit Online](#)

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。

创建包含数据的文件

1. 从主题[如何联接不同文件的内容 \(LINQ\) \(C#\)](#) 复制 scores.csv 数据并将它保存到解决方案文件夹。

示例

```

public class SortLines
{
    static void Main()
    {
        // Create an IEnumerable data source
        string[] scores = System.IO.File.ReadAllLines(@"../../../../scores.csv");

        // Change this to any value from 0 to 4.
        int sortField = 1;

        Console.WriteLine("Sorted highest to lowest by field [{0}]:", sortField);

        // Demonstrates how to return query from a method.
        // The query is executed here.
        foreach (string str in RunQuery(scores, sortField))
        {
            Console.WriteLine(str);
        }

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit");
        Console.ReadKey();
    }

    // Returns the query variable, not query results!
    static IEnumerable<string> RunQuery(IEnumerable<string> source, int num)
    {
        // Split the string and sort on field[num]
        var scoreQuery = from line in source
                         let fields = line.Split(',')
                         orderby fields[num] descending
                         select line;

        return scoreQuery;
    }
}

/* 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
*/

```

此示例还演示如何从方法返回查询变量。

编译代码

使用 `System.Linq` 和 `System.IO` 命名空间的 `using` 指令创建 C# 控制台应用程序项目。

请参阅

- [LINQ 和字符串 \(C#\)](#)

如何重新排列带分隔符的文件的字段 (LINQ) (C#)

2021/5/7 • [Edit Online](#)

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，可以非常轻松地使用 LINQ 来查询和操作 CSV 文件。事实上，可以使用此技术来重新排列任何结构化文本行部分；此技术不局限于 CSV 文件。

在下面的示例中，假设有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。结果保存到新文件，但不修改原始数据。

创建数据文件

- 将以下各行复制到名为 spreadsheet1.csv 的纯文本文件。将此文件保存到项目文件夹。

```
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
```

示例

```
class CSVFiles
{
    static void Main(string[] args)
    {
        // Create the IEnumerable data source
        string[] lines = System.IO.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 x = line.Split(',')
            orderby x[2]
            select x[2] + ", " + (x[1] + " " + x[0]);

        // Execute the query and write out the new file. Note that WriteAllLines
        // takes a string[], so ToArray is called on the query.
        System.IO.File.WriteAllLines(@"../../../../spreadsheet2.csv", query.ToArray());

        Console.WriteLine("Spreadsheet2.csv written to disk. Press any key to exit");
        Console.ReadKey();
    }
}

/* 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
*/
```

编译代码

使用 `System.Linq` 和 `System.IO` 命名空间的 `using` 指令创建 C# 控制台应用程序项目。

请参阅

- [LINQ 和字符串 \(C#\)](#)
- [LINQ 和文件目录 \(C#\)](#)
- [如何从 CSV 文件生成 XML \(C#\)](#)

如何合并和比较字符串集合 (LINQ) (C#)

2021/5/7 • [Edit Online](#)

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简单的串联、联合和交集。

设置项目和文本文件

- 将下面的姓名复制到名为 names1.txt 的文本文件，然后将此文件保存到项目文件夹：

```
Bankov, Peter
Holm, Michael
Garcia, Hugo
Potra, Cristina
Noriega, Fabricio
Aw, Kam Foo
Beebe, Ann
Toyoshima, Tim
Guy, Wey Yuan
Garcia, Debra
```

- 将下面的姓名复制到名为 names2.txt 的文本文件，然后将此文件保存到项目文件夹。请注意，这两个文件拥有一些共同的名称。

```
Liu, Jinghao
Bankov, Peter
Holm, Michael
Garcia, Hugo
Beebe, Ann
Gilchrist, Beth
Myrcha, Jacek
Giakoumakis, Leo
McLin, Nkenge
El Yassir, Mehdi
```

示例

```
class MergeStrings
{
    static void Main(string[] args)
    {
        //Put text files in your solution folder
        string[] fileA = System.IO.File.ReadAllLines(@"../../../../names1.txt");
        string[] fileB = System.IO.File.ReadAllLines(@"../../../../names2.txt");

        //Simple concatenation and sort. Duplicates are preserved.
        IEnumerable<string> 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.
        IEnumerable<string> 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).
IEnumerable<string> 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";

IEnumerable<String> tempQuery1 =
    from name in fileA
    let n = name.Split(',')
    where n[0] == nameMatch
    select name;

IEnumerable<string> tempQuery2 =
    from name2 in fileB
    let n2 = name2.Split(',')
    where n2[0] == nameMatch
    select name2;

IEnumerable<string> nameMatchQuery =
    tempQuery1.Concat(tempQuery2).OrderBy(s => s);
OutputQueryResults(nameMatchQuery, $"Concat based on partial name match \\"{nameMatch}\\":");

// Keep the console window open in debug mode.
Console.WriteLine("Press any key to exit");
Console.ReadKey();
}

static void OutputQueryResults(IEnumerable<string> query, string message)
{
    Console.WriteLine(System.Environment.NewLine + message);
    foreach (string item in query)
    {
        Console.WriteLine(item);
    }
    Console.WriteLine("{0} total names in list", query.Count());
}
/*
* 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, Nkeng
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
```

```
*/
```

编译代码

使用 System.Linq 和 System.IO 命名空间的 `using` 指令创建 C# 控制台应用程序项目。

请参阅

- [LINQ 和字符串 \(C#\)](#)
- [LINQ 和文件目录 \(C#\)](#)

如何从多个源填充对象集合 (LINQ) (C#)

2021/5/7 • [Edit Online](#)

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。

NOTE

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。

创建数据文件

按照[如何联接不同文件中的内容 \(LINQ\) \(C#\)](#) 中的说明，将 names.csv 和 scores.csv 文件复制到项目文件夹。

示例

下面的示例演示如何使用命名类型 `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。第一个字符串集合代表学生姓名和 ID，第二个集合代表学生 ID(在第一列)和四次考试分数。此 ID 用作外键。

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

class Student
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public int ID { get; set; }
    public List<int> ExamScores { get; set; }
}

class PopulateCollection
{
    static void Main()
    {
        // These data files are defined in How to join content from
        // dissimilar files (LINQ).

        // 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 = System.IO.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 = System.IO.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(',')
            let student = new Student
            {
                FirstName = splitName[1],
                LastName = splitName[0],
                ID = int.Parse(splitName[2]),
                ExamScores = scores
                    .Select(scoreLine => int.Parse(scoreLine))
                    .Skip(1)
                    .Take(4)
                    .ToList()
            }
            select student;
    }
}
```

```

        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)).
                      ToList()
    };

    // 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 {0} {1} is {2}.",
            student.FirstName, student.LastName,
            student.ExamScores.Average());
    }

    //Keep console window open in debug mode
    Console.WriteLine("Press any key to exit.");
    Console.ReadKey();
}
*/
/* 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 `select` 子句中，对象初始值设定项使用来自两个源的数据实例化每个新的 `Student` 对象。

如果不需要存储查询的结果，那么和命名类型相比，匿名类型使用起来更方便。如果在执行查询的方法外部传递查询结果，则需要使用命名类型。下面的示例执行与前面示例相同的任务，但使用的是匿名类型，而不是命名类型：

```
// 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 new  
{  
    First = splitName[0],  
    Last = 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 {0} {1} is {2}.",  
        student.First, student.Last, student.ExamScores.Average());  
}
```

请参阅

- [LINQ 和字符串 \(C#\)](#)
- [对象和集合初始值设定项](#)
- [匿名类型](#)

如何使用组将一个文件拆分成多个文件 (LINQ) (C#)

2021/5/7 • [Edit Online](#)

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。

创建数据文件

- 将下面的姓名复制到名为 names1.txt 的文本文件，然后将此文件保存到项目文件夹：

```
Bankov, Peter
Holm, Michael
Garcia, Hugo
Potra, Cristina
Noriega, Fabricio
Aw, Kam Foo
Beebe, Ann
Toyoshima, Tim
Guy, Wey Yuan
Garcia, Debra
```

- 将下面的姓名复制到名为 names2.txt 的文本文件，然后将此文件保存到项目文件夹：注意这两个文件有一些共同的姓名。

```
Liu, Jinghao
Bankov, Peter
Holm, Michael
Garcia, Hugo
Beebe, Ann
Gilchrist, Beth
Myrcha, Jacek
Giakoumakis, Leo
McLin, Nkenge
El Yassir, Mehdi
```

示例

```
class SplitWithGroups
{
    static void Main()
    {
        string[] fileA = System.IO.File.ReadAllLines(@"../../../../names1.txt");
        string[] fileB = System.IO.File.ReadAllLines(@"../../../../names2.txt");

        // Concatenate and remove duplicate names based on
        // default string comparer
        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(',')
                         group name by n[0][0] into g
                         orderby g.Key
                         select g;

        // Create a new file for each group that was created
    }
}
```

```

// Note that nested foreach loops are required to access
// individual items with each group.
foreach (var g in groupQuery)
{
    // Create the new file name.
    string fileName = @"../../testFile_" + g.Key + ".txt";

    // Output to display.
    Console.WriteLine(g.Key);

    // Write file.
    using (System.IO.StreamWriter sw = new System.IO.StreamWriter(fileName))
    {
        foreach (var item in g)
        {
            sw.WriteLine(item);
            // Output to console for example purposes.
            Console.WriteLine("    {0}", item);
        }
    }
}

// Keep console window open in debug mode.
Console.WriteLine("Files have been written. Press any key to exit");
Console.ReadKey();
}

/* 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
*/

```

对于与数据文件位于同一文件夹中的每个组，程序将为这些组编写单独的文件。

编译代码

使用 `System.Linq` 和 `System.IO` 命名空间的 `using` 指令创建 C# 控制台应用程序项目。

请参阅

- [LINQ 和字符串 \(C#\)](#)
- [LINQ 和文件目录 \(C#\)](#)

如何联接不同文件的内容 (LINQ) (C#)

2021/5/7 • [Edit Online](#)

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。

创建数据文件

- 将以下行复制到名为 scores.csv 的文件，并将文件保存到项目文件夹。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。

```
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
```

- 将以下行复制到名为 names.csv 的文件，并将文件保存到项目文件夹。此文件表示电子表格，其中包含学生的姓氏、名字和学生 ID。

```
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
```

示例

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

class JoinStrings
{
    static void Main()
    {
        // Join content from dissimilar files that contain
        // related information. File names.csv contains the student
        // name plus an ID number. File scores.csv contains the ID
        // and a set of four test scores. The following query joins
        // the two files based on the student ID.
```

```

// the scores to the student names by using ID as a
// matching key.

string[] names = System.IO.File.ReadAllLines(@"../../names.csv");
string[] scores = System.IO.File.ReadAllLines(@"../../scores.csv");

// Name:    Last[0],      First[1],  ID[2]
//          Omelchenko,   Svetlana,  11
// Score:   StudentID[0], Exam1[1]  Exam2[2],  Exam3[3],  Exam4[4]
//          111,           97,        92,        81,        60

// This query joins two dissimilar spreadsheets based on common ID value.
// Multiple from clauses are used instead of a join clause
// in order to store results of id.Split.

IEnumerable<string> scoreQuery1 =
    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];

// Pass a query variable to a method and execute it
// in the method. The query itself is unchanged.
OutputQueryResults(scoreQuery1, "Merge two spreadsheets:");

// Keep console window open in debug mode.
Console.WriteLine("Press any key to exit");
Console.ReadKey();
}

static void OutputQueryResults(IEnumerable<string> query, string message)
{
    Console.WriteLine(System.Environment.NewLine + message);
    foreach (string item in query)
    {
        Console.WriteLine(item);
    }
    Console.WriteLine("{0} total names in list", query.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
*/

```

另请参阅

- [LINQ 和字符串 \(C#\)](#)
- [LINQ 和文件目录 \(C#\)](#)

如何在 CSV 文本文件中计算列值 (LINQ) (C#)

2021/5/7 • [Edit Online](#)

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。

创建源文件

- 将以下行复制到名为 scores.csv 的文件，并将文件保存到项目文件夹。假定第一列表示学生 ID，后面几列表示四次考试的分数。

```
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
```

示例

```
class SumColumns
{
    static void Main(string[] args)
    {
        string[] lines = System.IO.File.ReadAllLines(@"../../../../scores.csv");

        // 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);

        Console.WriteLine("Press any key to exit");
        Console.ReadKey();
    }

    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 runtime.

        // 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 #{0}: Average:{1:##.##} High Score:{2} Low Score:{3}",
    examNum, average, max, 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.
    I Enumerable<I Enumerable<int>> 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 ofToList.
    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 max = results2.Max();
    int min = results2.Min();

    // Add one to column because the first exam is Exam #1,
    // not Exam #0.
    Console.WriteLine("Exam #{0} Average: {1:##.##} High Score: {2} Low Score: {3}",
                      column + 1, average, max, 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
*/
```

查询的工作原理是使用 [Split](#) 方法将每一行文本转换为数组。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。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`。

编译代码

使用 `System.Linq` 和 `System.IO` 命名空间的 [using](#) 指令创建 C# 控制台应用程序项目。

请参阅

- [LINQ 和字符串 \(C#\)](#)
- [LINQ 和文件目录 \(C#\)](#)

如何使用反射查询程序集的元数据 (LINQ) (C#)

2021/5/7 • [Edit Online](#)

.NET 反射 API 可用于检查 .NET 程序集中的元数据，以及创建位于该程序集中的类型、类型成员、参数等等的集合。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，该查询将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。

示例

```
using System;
using System.Linq;
using System.Reflection;

class ReflectionHowTO
{
    static void Main()
    {
        Assembly assembly = Assembly.Load("System.Core, Version=3.5.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089");
        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);
            }
        }

        Console.WriteLine("Press any key to exit... ");
        Console.ReadKey();
    }
}
```

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。

请参阅

- [LINQ to Objects \(C#\)](#)

如何使用反射查询程序集的元数据 (LINQ) (C#)

2021/5/7 • [Edit Online](#)

.NET 反射 API 可用于检查 .NET 程序集中的元数据，以及创建位于该程序集中的类型、类型成员、参数等等的集合。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，该查询将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。

示例

```
using System;
using System.Linq;
using System.Reflection;

class ReflectionHowTO
{
    static void Main()
    {
        Assembly assembly = Assembly.Load("System.Core, Version=3.5.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089");
        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);
            }
        }

        Console.WriteLine("Press any key to exit... ");
        Console.ReadKey();
    }
}
```

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。

请参阅

- [LINQ to Objects \(C#\)](#)

LINQ 和文件目录 (C#)

2020/11/2 • [Edit Online](#)

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。

本部分中的查询是非破坏性查询。它们不用于更改原始文件或文件夹的内容。这遵循了查询不应引起任何副作用这条规则。通常，修改源数据的任何代码(包括执行创建/更新/删除运算符的查询)应与只查询数据的代码分开。

本节包含下列主题：

[如何查询具有指定特性或名称的文件 \(C#\)](#)

演示如何通过检查文件的 `FileInfo` 对象的一个或多个属性来搜索文件。

[如何按扩展名对文件进行分组 \(LINQ\) \(C#\)](#)

演示如何根据文件扩展名返回 `FileInfo` 对象组。

[如何查询一组文件夹中的总字节数 \(LINQ\) \(C#\)](#)

演示如何返回指定目录树中所有文件的总字节数。

[如何比较两个文件夹的内容 \(LINQ\) \(C#\)](#)

演示如何返回位于两个指定文件夹中的所有文件，以及仅位于其中一个文件夹中的所有文件。

[如何查询目录树中的一个或多个最大的文件 \(LINQ\) \(C#\)](#)

演示如何返回目录树中的最大文件、最小文件或指定数量的文件。

[如何在目录树中查询重复文件 \(LINQ\) \(C#\)](#)

演示如何对出现在指定目录树中多个位置的所有文件名进行分组。此外，还演示如何根据自定义比较器执行更复杂的比较。

[如何查询文件夹中文件的内容 \(LINQ\) \(C#\)](#)

演示如何循环访问树中的文件夹，打开每个文件以及查询文件的内容。

注释

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，则你会在查询中获得 `FileNotFoundException`，即使未直接查询文件系统也是如此。本部分中的一些查询使用不同的方法，在某些情况下使用该方法不会出现这些特定异常。另一种方法是使用 `FileSystemWatcher` 保持数据源动态更新。

请参阅

- [LINQ to Objects \(C#\)](#)

如何查询具有指定特性或名称的文件 (C#)

2021/5/7 • [Edit Online](#)

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。

示例

```
class FindFileByExtension
{
    // This query will produce the full path for all .txt files
    // under the specified folder including subfolders.
    // It orders the list according to the file name.
    static void Main()
    {
        string startFolder = @"c:\program files\Microsoft Visual Studio 9.0\";

        // Take a snapshot of the file system.
        System.IO.DirectoryInfo dir = new System.IO.DirectoryInfo(startFolder);

        // This method assumes that the application has discovery permissions
        // for all folders under the specified path.
        IEnumerable<System.IO.FileInfo> fileList = dir.GetFiles("*.*",
System.IO.SearchOption.AllDirectories);

        //Create the query
        IEnumerable<System.IO.FileInfo> fileQuery =
            from file in fileList
            where file.Extension == ".txt"
            orderby file.Name
            select file;

        //Execute the query. This might write out a lot of files!
        foreach (System.IO.FileInfo fi in fileQuery)
        {
            Console.WriteLine(fi.FullName);
        }

        // Create and execute a new query by using the previous
        // query as a starting point. fileQuery is not
        // executed again until the call to Last()
        var newestFile =
            (from file in fileQuery
            orderby file.CreationTime
            select new { file.FullName, file.CreationTime })
            .Last();

        Console.WriteLine("\r\nThe newest .txt file is {0}. Creation time: {1}",
            newestFile.FullName, newestFile.CreationTime);

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit");
        Console.ReadKey();
    }
}
```

编译代码

使用 System.Linq 和 System.IO 命名空间的 `using` 指令创建 C# 控制台应用程序项目。

请参阅

- [LINQ to Objects \(C#\)](#)
- [LINQ 和文件目录 \(C#\)](#)

如何按扩展名对文件进行分组 (LINQ) (C#)

2021/5/7 • [Edit Online](#)

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。

示例

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。

```
class GroupByExtension
{
    // This query will sort all the files under the specified folder
    // and subfolder into groups keyed by the file extension.
    private static void Main()
    {
        // Take a snapshot of the file system.
        string startFolder = @"c:\program files\Microsoft Visual Studio 9.0\Common7";

        // Used in WriteLine to trim output lines.
        int trimLength = startFolder.Length;

        // Take a snapshot of the file system.
        System.IO.DirectoryInfo dir = new System.IO.DirectoryInfo(startFolder);

        // This method assumes that the application has discovery permissions
        // for all folders under the specified path.
        IEnumerable<System.IO.FileInfo> fileList = dir.GetFiles(".*",
System.IO.SearchOption.AllDirectories);

        // Create the query.
        var queryGroupByExt =
            from file in fileList
            group file by file.Extension.ToLower() into fileGroup
            orderby fileGroup.Key
            select fileGroup;

        // Display one group at a time. If the number of
        // entries is greater than the number of lines
        // in the console window, then page the output.
        PageOutput(trimLength, queryGroupByExt);
    }

    // This method specifically handles group queries of FileInfo objects with string keys.
    // It can be modified to work for any long listings of data. Note that explicit typing
    // must be used in method signatures. The groupbyExtList parameter is a query that produces
    // groups of FileInfo objects with string keys.
    private static void PageOutput(int rootLength,
                                    IEnumerable<System.Linq.IGrouping<string, System.IO.FileInfo>>
groupByExtList)
    {
        // Flag to break out of paging loop.
        bool goAgain = true;

        // "3" = 1 line for extension + 1 for "Press any key" + 1 for input cursor.
        int numLines = Console.WindowHeight - 3;

        // Iterate through the outer collection of groups.
        foreach (var filegroup in groupByExtList)
        {
            // Start a new extension at the top of a page.
```

```
int currentLine = 0;

// Output only as many lines of the current group as will fit in the window.
do
{
    Console.Clear();
    Console.WriteLine(filegroup.Key == String.Empty ? "[none]" : filegroup.Key);

    // Get 'numLines' number of items starting at number 'currentLine'.
    var resultPage = filegroup.Skip(currentLine).Take(numLines);

    //Execute the resultPage query
    foreach (var f in resultPage)
    {
        Console.WriteLine("\t{0}", f.FullName.Substring(rootLength));
    }

    // Increment the line counter.
    currentLine += numLines;

    // Give the user a chance to escape.
    Console.WriteLine("Press any key to continue or the 'End' key to break...");
    ConsoleKey key = Console.ReadKey().Key;
    if (key == ConsoleKey.End)
    {
        goAgain = false;
        break;
    }
} while (currentLine < filegroup.Count());

if (goAgain == false)
    break;
}

}
```

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。相同的方法适用于 Windows 和 Web 应用程序。请注意，由于代码对组中的项进行分页，因此需要使用 `foreach` 循环。此外，还有一些其他逻辑用于计算列表中的当前位置，以及使用户能够停止分页并退出程序。在此特定情况下，根据原始查询的缓存结果运行分页查询。在其他上下文中，如 LINQ to SQL，则不需要此类缓存。

编译代码

使用 System.Linq 和 System.IO 命名空间的 `using` 指令创建 C# 控制台应用程序项目。

请参阅

- LINQ to Objects (C#)
 - LINQ 和文件目录 (C#)

如何查询一组文件夹中的总字节数 (LINQ) (C#)

2021/5/7 • [Edit Online](#)

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。

示例

`Sum` 方法可将 `select` 子句中选择的所有项的值相加。可轻松修改此查询以检索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 `Sum` 方法。

```

class QuerySize
{
    public static void Main()
    {
        string startFolder = @"c:\program files\Microsoft Visual Studio 9.0\VC#";

        // Take a snapshot of the file system.
        // This method assumes that the application has discovery permissions
        // for all folders under the specified path.
        IEnumerable<string> fileList = System.IO.Directory.GetFiles(startFolder, "*.*",
System.IO.SearchOption.AllDirectories);

        var fileQuery = from file in fileList
                        select GetFileLength(file);

        // 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 {0} bytes in {1} files under {2}",
            totalBytes, fileList.Count(), startFolder);
        Console.WriteLine("The largest files is {0} bytes.", largestFile);

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }

    // This method is used to swallow the possible exception
    // that can be raised when accessing the System.IO.FileInfo.Length property.
    static long GetFileLength(string filename)
    {
        long retval;
        try
        {
            System.IO.FileInfo fi = new System.IO.FileInfo(filename);
            retval = fi.Length;
        }
        catch (System.IO.FileNotFoundException)
        {
            // If a file is no longer present,
            // just add zero bytes to the total.
            retval = 0;
        }
        return retval;
    }
}

```

如果只需对指定目录树中的字节数进行计数，则可以更高效地执行此操作而无需创建 LINQ 查询（这会产生创建列表集合作为数据源的开销）。随着查询变得更加复杂，或者在必须对相同数据源运行多个查询时，LINQ 方法会更加有用。

查询调用单独的方法来获取文件长度。这是为了使用在以下情况下会引发的可能异常：在 `GetFiles` 调用中创建了 `FileInfo` 对象之后，在其他线程中删除了文件。即使已创建 `FileInfo` 对象，该异常也可能出现，因为 `FileInfo` 对象会在首次访问其 `Length` 属性时，尝试使用最近长度刷新该属性。通过将此操作置于查询外部的 try-catch 块中，代码可遵循在查询中避免可能导致副作用的操作这一规则。一般情况下，在使用异常时必须格外谨慎，以确保应用程序不会处于未知状态。

编译代码

使用 System.Linq 和 System.IO 命名空间的 `using` 指令创建 C# 控制台应用程序项目。

请参阅

- [LINQ to Objects \(C#\)](#)
- [LINQ 和文件目录 \(C#\)](#)

如何比较两个文件夹的内容 (LINQ) (C#)

2021/5/7 • [Edit Online](#)

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。

NOTE

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。

示例

```
namespace QueryCompareTwoDirs
{
    class CompareDirs
    {

        static void Main(string[] args)
        {

            // Create two identical or different temporary folders
            // on a local drive and change these file paths.
            string pathA = @"C:\TestDir";
            string pathB = @"C:\TestDir2";

            System.IO.DirectoryInfo dir1 = new System.IO.DirectoryInfo(pathA);
            System.IO.DirectoryInfo dir2 = new System.IO.DirectoryInfo(pathB);

            // Take a snapshot of the file system.
            IEnumerable<System.IO.FileInfo> list1 = dir1.GetFiles("*.*",
System.IO.SearchOption.AllDirectories);
            IEnumerable<System.IO.FileInfo> list2 = dir2.GetFiles("*.*",
System.IO.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:");
        foreach (var v in queryCommonFiles)
        {
            Console.WriteLine(v.FullName); //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.
    // For this example we only check one way.
    var queryList1Only = (from file in list1
                           select file).Except(list2, myFileCompare);

    Console.WriteLine("The following files are in list1 but not list2:");
    foreach (var v in queryList1Only)
    {
        Console.WriteLine(v.FullName);
    }

    // Keep the console window open in debug mode.
    Console.WriteLine("Press any key to exit.");
    Console.ReadKey();
}

}

// 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 : System.Collections.Generic.IEqualityComparer<System.IO.FileInfo>
{
    public FileCompare() { }

    public bool Equals(System.IO.FileInfo f1, System.IO.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(System.IO.FileInfo fi)
    {
        string s = $"{fi.Name}{fi.Length}";
        return s.GetHashCode();
    }
}
}

```

编译代码

使用 System.Linq 和 System.IO 命名空间的 `using` 指令创建 C# 控制台应用程序项目。

请参阅

- [LINQ to Objects \(C#\)](#)
- [LINQ 和文件目录 \(C#\)](#)

如何查询目录树中的一个或多个最大的文件 (LINQ) (C#)

2021/5/7 • [Edit Online](#)

此示例演示与文件大小(以字节为单位)相关的五个查询：

- 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)。

示例

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。可以轻松地修改这些示例，以便使查询基于 [FileInfo](#) 对象的其他某个属性。

```
class QueryBySize
{
    static void Main(string[] args)
    {
        QueryFilesBySize();
        Console.WriteLine("Press any key to exit");
        Console.ReadKey();
    }

    private static void QueryFilesBySize()
    {
        string startFolder = @"c:\program files\Microsoft Visual Studio 9.0\";

        // Take a snapshot of the file system.
        System.IO.DirectoryInfo dir = new System.IO.DirectoryInfo(startFolder);

        // This method assumes that the application has discovery permissions
        // for all folders under the specified path.
        IEnumerable<System.IO.FileInfo> fileList = dir.GetFiles("*.*",
System.IO.SearchOption.AllDirectories);

        //Return the size of the largest file
        long maxSize =
            (from file in fileList
             let len = GetFileLength(file)
             select len)
            .Max();

        Console.WriteLine("The length of the largest file under {0} is {1}",
startFolder, maxSize);

        // Return the FileInfo object for the largest file
        // by sorting and selecting from beginning of list
        System.IO.FileInfo longestFile =
            (from file in fileList
             let len = GetFileLength(file)
             where len > 0
             orderby len descending
```

```

        select file)
        .First();

    Console.WriteLine("The largest file under {0} is {1} with a length of {2} bytes",
                    startFolder, longestFile.FullName, longestFile.Length);

    //Return the FileInfo of the smallest file
    System.IO.FileInfo smallestFile =
        (from file in fileList
         let len = GetFileLength(file)
         where len > 0
         orderby len ascending
         select file).First();

    Console.WriteLine("The smallest file under {0} is {1} with a length of {2} bytes",
                    startFolder, smallestFile.FullName, smallestFile.Length);

    //Return the FileInfos for the 10 largest files
    // queryTenLargest is an IEnumerable<System.IO.FileInfo>
    var queryTenLargest =
        (from file in fileList
         let len = GetFileLength(file)
         orderby len descending
         select file).Take(10);

    Console.WriteLine("The 10 largest files under {0} are:", startFolder);

    foreach (var v in queryTenLargest)
    {
        Console.WriteLine("{0}: {1} bytes", v.FullName, v.Length);
    }

    // Group the files according to their size, leaving out
    // files that are less than 200000 bytes.
    var querySizeGroups =
        from file in fileList
        let len = GetFileLength(file)
        where len > 0
        group file by (len / 100000) into fileGroup
        where fileGroup.Key >= 2
        orderby fileGroup.Key descending
        select fileGroup;

    foreach (var filegroup in querySizeGroups)
    {
        Console.WriteLine(filegroup.Key.ToString() + "0000");
        foreach (var item in filegroup)
        {
            Console.WriteLine("\t{0}: {1}", item.Name, item.Length);
        }
    }
}

// This method is used to swallow the possible exception
// that can be raised when accessing the FileInfo.Length property.
// In this particular case, it is safe to swallow the exception.
static long GetFileLength(System.IO.FileInfo fi)
{
    long retval;
    try
    {
        retval = fi.Length;
    }
    catch (System.IO.FileNotFoundException)
    {
        // If a file is no longer present,
        // just add zero bytes to the total.
        retval = 0;
    }
}

```

```
        return retval;
    }

}
```

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。

查询调用单独的方法来获取文件大小(以字节为单位)，以便使用在以下情况下会引发的可能异常：自在 `GetFiles` 调用中创建了 `FileInfo` 对象以来的时间段内，在其他线程中删除了文件。即使创建了 `FileInfo` 对象，该异常也可能出现，因为 `FileInfo` 对象会在首次访问其 `Length` 属性时，尝试使用最新大小(以字节为单位)刷新该属性。通过将此操作置于查询外部的 try-catch 块中，我们可遵循在查询中避免可能导致副作用的操作这一规则。一般情况下，在使用异常时必须格外谨慎，以确保应用程序不会处于未知状态。

编译代码

使用 `System.Linq` 和 `System.IO` 命名空间的 `using` 指令创建 C# 控制台应用程序项目。

请参阅

- [LINQ to Objects \(C#\)](#)
- [LINQ 和文件目录 \(C#\)](#)

如何在目录树中查询重复文件 (LINQ) (C#)

2021/5/7 • [Edit Online](#)

有时，具有相同名称的文件可能位于多个文件夹中。例如，在 Visual Studio 安装文件夹下，多个文件夹中都有 readme.htm 文件。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。

示例

```
class QueryDuplicateFileNames
{
    static void Main(string[] args)
    {
        // Uncomment QueryDuplicates2 to run that query.
        QueryDuplicates();
        // QueryDuplicates2();

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }

    static void QueryDuplicates()
    {
        // Change the root drive or folder if necessary
        string startFolder = @"c:\program files\Microsoft Visual Studio 9.0\";

        // Take a snapshot of the file system.
        System.IO.DirectoryInfo dir = new System.IO.DirectoryInfo(startFolder);

        // This method assumes that the application has discovery permissions
        // for all folders under the specified path.
        IEnumerable<System.IO.FileInfo> fileList = dir.GetFiles("*.*",
System.IO.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;

        // Pass the query to a method that will
        // output one page at a time.
        PageOutput<string, string>(queryDupNames);
    }

    // A Group key that can be passed to a separate method.
    // Override Equals and GetHashCode to define equality for the key.
    // Override ToString to provide a friendly name for Key.ToString()
    class PortableKey
    {
        public string Name { get; set; }
        public DateTime LastWriteTime { get; set; }
        public long Length { get; set; }

        public override bool Equals(object obj)
        {
            PortableKey key = obj as PortableKey;
            if (key != null)
                return Name == key.Name && LastWriteTime == key.LastWriteTime && Length == key.Length;
            else
                return false;
        }

        public override int GetHashCode()
        {
            return Name.GetHashCode() ^ LastWriteTime.GetHashCode() ^ Length.GetHashCode();
        }
    }
}
```

```

        PortableKey other = (PortableKey)obj;
        return other.LastWriteTime == this.LastWriteTime &&
               other.Length == this.Length &&
               other.Name == this.Name;
    }

    public override int GetHashCode()
    {
        string str = $"{this.LastWriteTime}{this.Length}{this.Name}";
        return str.GetHashCode();
    }
    public override string ToString()
    {
        return $"{this.Name} {this.Length} {this.LastWriteTime}";
    }
}

static void QueryDuplicates2()
{
    // Change the root drive or folder if necessary.
    string startFolder = @"c:\program files\Microsoft Visual Studio 9.0\Common7";

    // Make the lines shorter for the console display
    int charsToSkip = startFolder.Length;

    // Take a snapshot of the file system.
    System.IO.DirectoryInfo dir = new System.IO.DirectoryInfo(startFolder);
    IEnumerable<System.IO.FileInfo> fileList = dir.GetFiles("*.*",
System.IO.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
            new PortableKey { Name = file.Name, LastWriteTime = file.LastWriteTime, Length = file.Length
} into fileGroup
        where fileGroup.Count() > 1
        select fileGroup;

    var list = queryDupFiles.ToList();

    int i = queryDupFiles.Count();

    PageOutput<PortableKey, string>(queryDupFiles);
}

// A generic method to page the output of the QueryDuplications methods
// Here the type of the group must be specified explicitly. "var" cannot
// be used in method signatures. This method does not display more than one
// group per page.
private static void PageOutput<K, V>(IEnumerable<System.Linq.IGrouping<K, V>> groupByExtList)
{
    // Flag to break out of paging loop.
    bool goAgain = true;

    // "3" = 1 line for extension + 1 for "Press any key" + 1 for input cursor.
    int numLines = Console.WindowHeight - 3;

    // Iterate through the outer collection of groups.
    foreach (var filegroup in groupByExtList)
    {
        // Start a new extension at the top of a page.
        int currentLine = 0;

```

```

// Output only as many lines of the current group as will fit in the window.
do
{
    Console.Clear();
    Console.WriteLine("Filename = {0}", filegroup.Key.ToString() == String.Empty ? "[none]" :
filegroup.Key.ToString());

    // Get 'numLines' number of items starting at number 'currentLine'.
    var resultPage = filegroup.Skip(currentLine).Take(numLines);

    //Execute the resultPage query
    foreach (var fileName in resultPage)
    {
        Console.WriteLine("\t{0}", fileName);
    }

    // Increment the line counter.
    currentLine += numLines;

    // Give the user a chance to escape.
    Console.WriteLine("Press any key to continue or the 'End' key to break...");
    ConsoleKey key = Console.ReadKey().Key;
    if (key == ConsoleKey.End)
    {
        goAgain = false;
        break;
    }
} while (currentLine < filegroup.Count());

if (goAgain == false)
    break;
}
}
}

```

第一个查询使用简单键来确定匹配；此查询可以找到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。

编译代码

使用 System.Linq 和 System.IO 命名空间的 `using` 指令创建 C# 控制台应用程序项目。

请参阅

- [LINQ to Objects \(C#\)](#)
- [LINQ 和文件目录 \(C#\)](#)

如何查询文件夹中文本文件的内容 (LINQ) (C#)

2021/5/7 • [Edit Online](#)

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。有关详细信息，请参阅[如何将 LINQ 查询与正则表达式合并 \(C#\)](#)。

示例

```

class QueryContents
{
    public static void Main()
    {
        // Modify this path as necessary.
        string startFolder = @"c:\program files\Microsoft Visual Studio 9.0\";

        // Take a snapshot of the file system.
        System.IO.DirectoryInfo dir = new System.IO.DirectoryInfo(startFolder);

        // This method assumes that the application has discovery permissions
        // for all folders under the specified path.
        IEnumerable<System.IO.FileInfo> fileList = dir.GetFiles("*.*",
System.IO.SearchOption.AllDirectories);

        string searchTerm = @"Visual Studio";

        // Search the contents of each file.
        // A regular expression created with the RegEx class
        // could be used instead of the Contains method.
        // queryMatchingFiles is an IEnumerable<string>.
        var queryMatchingFiles =
            from file in fileList
            where file.Extension == ".htm"
            let fileText = GetFileText(file.FullName)
            where fileText.Contains(searchTerm)
            select file.FullName;

        // Execute the query.
        Console.WriteLine("The term \"{0}\" was found in:", searchTerm);
        foreach (string filename in queryMatchingFiles)
        {
            Console.WriteLine(filename);
        }

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit");
        Console.ReadKey();
    }

    // Read the contents of the file.
    static string GetFileText(string name)
    {
        string fileContents = String.Empty;

        // If the file has been deleted since we took
        // the snapshot, ignore it and return the empty string.
        if (System.IO.File.Exists(name))
        {
            fileContents = System.IO.File.ReadAllText(name);
        }
        return fileContents;
    }
}

```

编译代码

使用 `System.Linq` 和 `System.IO` 命名空间的 `using` 指令创建 C# 控制台应用程序项目。

请参阅

- [LINQ 和文件目录 \(C#\)](#)
- [LINQ to Objects \(C#\)](#)

如何使用 LINQ 查询 ArrayList (C#)

2021/5/7 • [Edit Online](#)

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。例如，如果有 `Student` 对象的 `ArrayList`，那么 `from 子句`应如下所示：

```
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](#)。

示例

下面的示例演示对 `ArrayList` 的简单查询。请注意，本示例在代码调用 `Add` 方法时使用对象初始值设定项，但这不是必需的。

```

using System;
using System.Collections;
using System.Linq;

namespace NonGenericLINQ
{
    public class Student
    {
        public string FirstName { get; set; }
        public string LastName { get; set; }
        public int[] Scores { get; set; }
    }

    class Program
    {
        static void Main(string[] args)
        {
            ArrayList arrList = new ArrayList();
            arrList.Add(
                new Student
                {
                    FirstName = "Svetlana", LastName = "Omelchenko", Scores = new int[] { 98, 92, 81, 60 }
                });
            arrList.Add(
                new Student
                {
                    FirstName = "Claire", LastName = "O'Donnell", Scores = new int[] { 75, 84, 91, 39 }
                });
            arrList.Add(
                new Student
                {
                    FirstName = "Sven", LastName = "Mortensen", Scores = new int[] { 88, 94, 65, 91 }
                });
            arrList.Add(
                new Student
                {
                    FirstName = "Cesar", LastName = "Garcia", Scores = new int[] { 97, 89, 85, 82 }
                });

            var query = from Student student in arrList
                       where student.Scores[0] > 95
                       select student;

            foreach (Student s in query)
                Console.WriteLine(s.LastName + ": " + s.Scores[0]);

            // Keep the console window open in debug mode.
            Console.WriteLine("Press any key to exit.");
            Console.ReadKey();
        }
    }
}

/* Output:
   Omelchenko: 98
   Garcia: 97
*/

```

请参阅

- [LINQ to Objects \(C#\)](#)

如何为 LINQ 查询添加自定义方法 (C#)

2021/5/7 • [Edit Online](#)

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](#)。

添加聚合方法

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。

```
public static class LINQExtension
{
    public static double Median(this IEnumerable<double>? source)
    {
        if (!(source?.Any() ?? false))
        {
            throw new InvalidOperationException("Cannot compute median for a null or empty set.");
        }

        var sortedList = (from number in source
                         orderby number
                         select 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。

```

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
*/

```

重载聚合方法以接受各种类型

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。

为每种类型创建重载

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。

```

//int overload
public static double Median(this IEnumerable<int> source) =>
    (from num in source select (double)num).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：

```

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
Integer: Median = 3
*/

```

创建一般重载

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。

```

// 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。

```
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:

 Integer: Median = 3
 String: Median = 4
*/
```

添加返回序列的方法

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。

```
// 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 i = 0;
    foreach (var element in source)
    {
        if (i % 2 == 0)
        {
            yield return element;
        }
        i++;
    }
}
```

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：

```
string[] strings = { "a", "b", "c", "d", "e" };

var query5 = stringsAlternateElements();

foreach (var element in query5)
{
    Console.WriteLine(element);
}

/*
This code produces the following output:

a
c
e
*/
```

请参阅

- [IEnumerable<T>](#)
- [扩展方法](#)

LINQ to ADO.NET (门户网站页)

2020/11/2 • [Edit Online](#)

通过 LINQ to ADO.NET, 可使用语言集成查询 (LINQ) 编程模型在 ADO.NET 中跨任何可枚举对象进行查询。

NOTE

LINQ to ADO.NET 文档位于 .NET Framework SDK 的 ADO.NET 部分：[LINQ 和 ADO.NET](#)。

有三个独立的 ADO.NET 语言集成查询 (LINQ) 技术：LINQ to DataSet、LINQ to SQL 和 LINQ to Entities。LINQ to DataSet 提供针对 [DataSet](#) 的形式多样的优化查询，LINQ to SQL 可用于直接查询 SQL Server 数据库架构，而 LINQ to Entities 可实现实体数据模型的查询。

LINQ to DataSet

[DataSet](#) 是 ADO.NET 中使用最广泛的某个组件，也是断开编程模型（构建 ADO.NET 的基础）连接的重要元素。

[DataSet](#) 虽然具有突出的优点，但其查询功能也存在限制。

凭借 LINQ to DataSet，可通过使用为其他多个数据源提供的同一查询功能，在 [DataSet](#) 中加入更丰富的查询功能。

有关详细信息，请参阅 [LINQ to DataSet](#)。

LINQ to SQL

LINQ to SQL 提供用于将关系数据作为对象进行管理的运行时基础结构。在 LINQ to SQL 中，关系数据库的数据模型映射到用开发人员所用的编程语言表示的对象模型。执行应用程序时，LINQ to SQL 会将对象模型中的语言集成查询转换为 SQL，然后将其发送到数据库进行执行。数据库返回结果时，LINQ to SQL 会将结果转换回可操作对象。

LINQ to SQL 包括对数据库中的已存储过程和用户定义的函数和对象模型中的继承的支持。

有关详细信息，请参阅 [LINQ to SQL](#)。

LINQ to Entities

通过实体数据模型，在 .NET 环境中将关系数据作为对象公开。这使得对象层成为实现 LINQ 支持的理想目标，开发人员可以采用生成业务逻辑所用的语言来构建数据库查询。此功能称为 LINQ to Entities。有关详细信息，请参阅 [LINQ to Entities](#)。

请参阅

- [LINQ 和 ADO.NET](#)
- [语言集成查询 \(LINQ\) \(C#\)](#)

启用数据源以进行 LINQ 查询

2021/3/5 • [Edit Online](#)

可通过多种方式来扩展 LINQ，以便能够在 LINQ 模式中查询任何数据源。数据源可以是数据结构、Web 服务、文件系统或数据库等。LINQ 模式使客户端可以轻松地查询能够进行 LINQ 查询的数据源，因为查询的语法和模式没有更改。可通过以下方式将 LINQ 扩展到这些数据源：

- 在某个类型中实现 `IEnumerable<T>` 接口，使 LINQ to Objects 能够查询该类型。
- 创建扩展某个类型的标准查询运算符方法（比如 `Where` 和 `Select`），使自定义 LINQ 能够查询该类型。
- 为实现 `IQueryable<T>` 接口的数据源创建一个提供程序。实现此接口的提供程序以表达式树的形式接收 LINQ 查询，提供程序能够以自定义方式（例如以远程方式）执行该查询。
- 为数据源创建一个利用现有 LINQ 技术的提供程序。这种提供程序不仅能够进行查询，而且能够为用户定义的类型插入、更新和删除操作及映射。

本主题将讨论这些选项。

如何使 LINQ 能够查询您的数据源

内存中的数据

通过两种方式，可以使 LINQ 能够查询内存中数据。如果数据的类型实现了 `IEnumerable<T>`，你可以使用 LINQ to Objects 来查询数据。如果无法通过实现 `IEnumerable<T>` 接口启用类型枚举，可以在该类型中定义 LINQ 标准查询运算符方法，或创建扩展该类型的 LINQ 标准查询运算符方法。标准查询运算符的自定义实现应使用延迟执行来返回结果。

远程数据

使 LINQ 能够查询远程数据源的最佳选择是实现 `IQueryable<T>` 接口。但是，这与为数据源扩展提供程序（比如 LINQ to SQL）有所不同。

IQueryable LINQ 提供程序

实现 `IQueryable<T>` 的 LINQ 提供程序之间的复杂性可能差别很大。本节讨论这些不同程度的复杂性。

复杂性较低的 `IQueryable` 提供程序可与 Web 服务的一个方法交互。这种类型的提供程序非常具体，因为它需要所处理的查询中有具体信息。它有封闭的类型系统，可能会公开单一结果类型。大多数查询都是在本地执行的，例如，通过使用标准查询运算符的 `Enumerable` 实现来执行。复杂性较低的提供程序可能只会检查表示查询的表达式树中的一个方法调用表达式，并允许在其他地方处理查询的其余逻辑。

中等复杂性的 `IQueryable` 提供程序可能针对具有部分可表达查询语言的数据源。如果该提供程序针对 Web 服务，它可以与 Web 服务的多个方法进行交互，并基于查询提出的问题选择要调用的方法。与简单提供程序相比，中等复杂性的提供程序的类型系统较丰富，但它仍然是固定类型系统。例如，提供程序可以公开具有可遍历的一对多关系的类型，但将不会为用户定义的类型提供映射技术。

复杂的 `IQueryable` 提供程序（如 LINQ to SQL 提供程序）可将完整的 LINQ 查询转换为可表达查询语言（如 SQL）。与复杂性较低的提供程序相比，复杂的提供程序更为全面，因为它能在查询中处理各种各样的问题。它还具有开放的类型系统，因而必须包含广泛的基础结构来映射用户定义的类型。开发复杂的提供程序需要耗费大量的精力。

请参阅

- [IQueryable<T>](#)
- [IEnumerable<T>](#)
- [Enumerable](#)
- [标准查询运算符概述 \(C#\)](#)
- [LINQ to Objects \(C#\)](#)

对 LINQ 的 Visual Studio IDE 和工具支持 (C#)

2020/11/2 • [Edit Online](#)

Visual Studio 集成开发环境 (IDE) 提供支持 LINQ 应用程序开发的以下功能：

Object Relational Designer

对象关系设计器是一个可视化设计工具，可用于在 [LINQ to SQL](#) 应用程序中通过 C# 生成表示底层数据库中关系数据的类。有关详细信息，请参阅 [Visual Studio 中的 LINQ to SQL 工具](#)。

SQLMetal 命令行工具

SQLMetal 是一个命令行工具，可用于在生成过程中从现有数据库生成供 LINQ to SQL 应用程序使用的类。有关详细信息，请参阅 [SqlMetal.exe \(代码生成工具\)](#)。

LINQ 感知代码编辑器

C# 代码编辑器支持 LINQ 广泛使用 IntelliSense 和格式设置功能。

Visual Studio 调试器支持

Visual Studio 调试器支持查询表达式的调试。有关详细信息，请参阅 [Debugging LINQ \(调试 LINQ\)](#)。

请参阅

- [语言集成查询 \(LINQ\) \(C#\)](#)

反射 (C#)

2021/3/5 • • [Edit Online](#)

反射提供描述程序集、模块和类型的对象 ([Type](#) 类型)。可以使用反射动态地创建类型的实例，将类型绑定到现有对象，或从现有对象中获取类型，然后调用其方法或访问器字段和属性。如果代码中使用了特性，可以利用反射来访问它们。有关更多信息，请参阅[特性](#)。

下面一个简单的反射示例，使用方法 [GetType\(\)](#)(被 [Object](#) 基类的所有类型继承)以获取变量类型：

NOTE

请确保在 .cs 文件顶部添加 `using System;` 和 `using System.Reflection;`。

```
// Using GetType to obtain type information:  
int i = 42;  
Type type = i.GetType();  
Console.WriteLine(type);
```

输出为：`System.Int32`。

下面的示例使用反射获取已加载的程序集的完整名称。

```
// Using Reflection to get information of an Assembly:  
Assembly info = typeof(int).Assembly;  
Console.WriteLine(info);
```

输出为：`System.Private.CoreLib, Version=4.0.0.0, Culture=neutral, PublicKeyToken=7cec85d7bea7798e`。

NOTE

C# 关键字 `protected` 和 `internal` 在中间语言 (IL) 中没有任何意义，且不会用于反射 API 中。在 IL 中对应的术语为“系列”和“程序集”。若要标识 `internal` 使用反射的方法，请使用 [IsAssembly](#) 属性。若要标识 `protected internal` 方法，请使用 [IsFamilyOrAssembly](#)。

反射概述

反射在以下情况下很有用：

- 需要访问程序元数据中的特性时。有关详细信息，请参阅[检索存储在特性中的信息](#)。
- 检查和实例化程序集中的类型。
- 在运行时构建新类型。使用 [System.Reflection.Emit](#) 中的类。
- 执行后期绑定，访问在运行时创建的类型上的方法。请参阅主题[“动态加载和使用类型”](#)。

相关章节

更多相关信息：

- [反射](#)
- [查看类型信息](#)

- 反射类型和泛型类型
- [System.Reflection.Emit](#)
- [检索存储在特性中的信息](#)

另请参阅

- [C# 编程指南](#)
- [.NET 中的程序集](#)

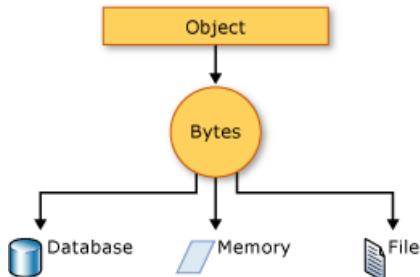
序列化 (C#)

2020/11/2 • [Edit Online](#)

序列化是指将对象转换成字节流，从而存储对象或将对象传输到内存、数据库或文件的过程。它的主要用途是保存对象的状态，以便能够在需要时重新创建对象。反向过程称为“反序列化”。

序列化的工作原理

下图展示了序列化的整个过程：



将对象序列化为带有数据的流。该流还可能包含有关对象类型的信息，例如其版本、区域性和程序集名称。可以将此流中的对象存储在数据库、文件或内存中。

序列化的用途

通过序列化，开发人员可以保存对象的状态，并能在需要时重新创建对象，同时还能存储对象和交换数据。通过序列化，开发人员可以执行如下操作：

- 使用 Web 服务将对象发送到远程应用程序
- 将对象从一个域传递到另一个域
- 将对象通过防火墙传递为 JSON 或 XML 字符串
- 跨应用程序维护安全或用户特定的信息

JSON 序列化

`System.Text.Json` 命名空间包含用于 JavaScript 对象表示法 (JSON) 序列化和反序列化的类。JSON 是一种常用于在 Web 上共享数据的开放标准。

JSON 序列化将对象的公共属性序列化为符合 [RFC 8259 JSON 规范](#) 的字符串、字节数组或流。若要控制 `JsonSerializer` 对类的实例进行序列化或反序列化的方法，请执行以下操作：

- 使用 `JsonSerializerOptions` 对象
- 将 `System.Text.Json.Serialization` 命名空间中的特性应用于类或属性
- 实现自定义转换器

二进制和 XML 序列化

`System.Runtime.Serialization` 命名空间包含用于对二进制和 XML 进行序列化和反序列化的类。

二进制序列化使用二进制编码来生成精简的序列化以供使用，如基于存储或套接字的网络流。在二进制序列化中，所有成员（包括只读成员）都会被序列化，且性能也会有所提升。

WARNING

二进制序列化可能会十分危险。有关详细信息, 请参阅[BinaryFormatter security guide](#)。

XML 序列化将对象的公共字段和属性或方法的参数和返回值序列化成符合特定 XML 架构定义语言 (XSD) 文档要求的 XML 流。XML 序列化生成已转换成 XML 的强类型类, 其中包含公共属性和字段。

[System.Xml.Serialization](#) 包含用于对 XML 进行序列化和反序列化的类。将特性应用于类和类成员, 从而控制 [XmlSerializer](#) 如何序列化或反序列化类的实例。

让对象可序列化

若要对二进制或 XML 进行序列化, 你需要:

- 要序列化的对象
- 包含序列化对象的流
- 一个 [System.Runtime.Serialization.Formatter](#) 实例

将 [SerializableAttribute](#) 特性应用于某个类型, 以指示可对此类型进行序列化的实例。如果尝试对没有 [SerializableAttribute](#) 特性的类型进行序列化, 则会引发异常。

若要防止对字段进行序列化, 请应用 [NonSerializedAttribute](#) 特性。如果可序列化的类型中的一个字段包含指针、句柄或特定环境专用的其他一些数据结构, 且不能在其他环境中有意义地重构, 不妨让其不可序列化。

如果已序列化的类引用被标记为 [SerializableAttribute](#) 的其他类的对象, 那么这些对象也会被序列化。

基本和自定义序列化

可以使用两种方法对二进制和 XML 进行序列化: 基本和自定义。

基本序列化使用 .NET 自动序列化对象。唯一的要求是类应用 [SerializableAttribute](#) 特性。[NonSerializedAttribute](#) 可用于防止特定字段被序列化。

使用基本序列化时, 对象的版本控制可能会产生问题。对于重要的版本控制问题, 可以使用自定义序列化。基本序列化是最简单的序列化执行方式, 但无法提供太多的进程控制。

在自定义序列化中, 可以精确指定要序列化的对象以及具体执行方式。类必须被标记为 [SerializableAttribute](#), 并实现 [ISerializable](#) 接口。如果还希望按自定义方式反序列化对象, 请使用自定义构造函数。

设计器序列化

设计器序列化是一种特殊形式的序列化, 涉及与开发工具相关联的对象暂留。设计器序列化是指将对象图转换成源文件以供日后用于恢复对象图的过程。源文件可以包含代码、标记或 SQL 表信息。

相关主题和示例

[System.Text.Json 概述](#) 演示如何获取 [System.Text.Json](#) 库。

[如何在 .NET 中对 JSON 数据进行序列化和反序列化](#)。演示如何使用 [JsonSerializer](#) 类在 JSON 之间读取和写入对象数据。

演练: 在 Visual Basic 中保持对象 (C#)

展示了如何使用序列化在实例之间暂留对象数据, 以便可以存储值并在下次实例化对象时检索值。

如何从 XML 文件读取对象数据 (C#)

介绍如何使用 [XmlSerializer](#) 类读取之前写入 XML 文件的对象数据。

如何将对象数据写入 XML 文件 (C#)

介绍如何使用 [XmlSerializer](#) 类从某个类将对象写入 XML 文件。

如何将对象数据写入 XML 文件 (C#)

2020/11/2 • [Edit Online](#)

本示例使用 [XmlSerializer](#) 类从某个类将对象写入 XML 文件。

示例

```
public class XMLWrite
{
    static void Main(string[] args)
    {
        WriteXML();
    }

    public class Book
    {
        public String title;
    }

    public static void WriteXML()
    {
        Book overview = new Book();
        overview.title = "Serialization Overview";
        System.Xml.Serialization.XmlSerializer writer =
            new System.Xml.Serialization.XmlSerializer(typeof(Book));

        var path = Environment.GetFolderPath(Environment.SpecialFolder.MyDocuments) +
        "//SerializationOverview.xml";
        System.IO.FileStream file = System.IO.File.Create(path);

        writer.Serialize(file, overview);
        file.Close();
    }
}
```

编译代码

所序列化的类必须有一个公共的无参数构造函数。

可靠编程

以下情况可能会导致异常：

- 进行序列化的类没有公共的无参数构造函数。
- 文件存在且为只读 ([IOException](#))。
- 路径过长 ([PathTooLongException](#))。
- 磁盘已满 ([IOException](#))。

.NET 安全性

此示例在文件尚未存在时创建新文件。如果某个应用程序需要创建文件，则该应用程序需要针对文件夹的 `Create` 访问权限。如果文件已存在，则该应用程序只需要 `Write` 访问权限（这是较弱的特权）。如有可能，在部

署过程中创建文件，并且仅授予针对单个文件的 `Read` 访问权限(而不是针对 `Create` 文件夹的访问权限)会更加安全。

请参阅

- [StreamWriter](#)
- [如何从 XML 文件读取对象数据 \(C#\)](#)
- [序列化 \(C#\)](#)

如何从 XML 文件读取对象数据 (C#)

2020/11/2 • [Edit Online](#)

本示例使用 `XmlSerializer` 类读取之前写入 XML 文件的对象数据。

示例

```
public class Book
{
    public String title;
}

public void ReadXML()
{
    // First write something so that there is something to read ...
    var b = new Book { title = "Serialization Overview" };
    var writer = new System.Xml.Serialization.XmlSerializer(typeof(Book));
    var wfile = new System.IO.StreamWriter(@"c:\temp\SerializationOverview.xml");
    writer.Serialize(wfile, b);
    wfile.Close();

    // Now we can read the serialized book ...
    System.Xml.Serialization.XmlSerializer reader =
        new System.Xml.Serialization.XmlSerializer(typeof(Book));
    System.IO.StreamReader file = new System.IO.StreamReader(
        @"c:\temp\SerializationOverview.xml");
    Book overview = (Book)reader.Deserialize(file);
    file.Close();

    Console.WriteLine(overview.title);
}
```

编译代码

将文件名称“`c:\temp\SerializationOverview.xml`”替换为包含序列化数据的文件的名称。有关序列化数据的详细信息，请参阅[如何将对象数据写入 XML 文件 \(C#\)](#)。

类必须有一个公共的无参数构造函数。

只有公共属性和字段才会进行反序列化。

可靠编程

以下情况可能会导致异常：

- 进行序列化的类没有公共的无参数构造函数。
- 文件中的数据不表示要进行反序列化的类中的数据。
- 该文件不存在 (`IOException`)。

.NET 安全性

始终验证输入，并且绝不会反序列化来自不受信任源的数据。重新创建的对象会在具有对它进行反序列化的代码的权限的本地计算机上运行。在应用程序中使用输入的数据之前，需验证所有的输入内容。

请参阅

- [StreamWriter](#)
- [如何将对象数据写入 XML 文件 \(C#\)](#)
- [序列化 \(C#\)](#)
- [C# 编程指南](#)

演练：使用 C# 保留对象

2021/5/7 • [Edit Online](#)

可使用序列化在实例之间保持对象的数据，以便可存储值并在下次实例化对象时检索这些值。

本演练将创建一个基础的 `Loan` 对象，并将其值保留在文件中。当重新创建该对象时，将检索文件中的数据。

IMPORTANT

如果此文件尚不存在，本示例将新建文件。如果应用程序必须创建文件，则该应用程序必须对文件夹具有 `Create` 权限。可使用访问控制列表设置权限。如果文件已存在，则该应用程序只需要 `Write` 权限（这是较弱的权限）。如有可能，较安全的做法是在部署过程中创建文件并仅向单个文件授予 `Read` 权限（而不是授予文件夹的“创建”权限）。此外，较安全的做法是将数据写入用户文件夹，而不是根文件夹或 `Program Files` 文件夹。

IMPORTANT

此示例将数据存储为二进制格式的文件。不应将这些格式用于敏感数据，如密码或信用卡信息。

先决条件

- 若要生成并运行，请安装 [.NET Core SDK](#)。
- 安装常用的代码编辑器（如果尚未安装）。

TIP

需要安装代码编辑器？试用 [Visual Studio](#)！

- 该示例需要 C# 7.3。请参阅[选择 C# 语言版本](#)

可在 [.NET 示例 GitHub 存储库](#) 在线检查示例代码。

创建 Loan 对象

第一步是创建 `Loan` 类和使用该类的控制台应用程序：

- 创建一个新的应用程序。键入 `dotnet new console -o serialization`，在名为 `serialization` 的子目录下创建新的控制台应用程序。
- 在编辑器中打开应用程序，然后添加名为 `Loan.cs` 的新类。
- 将以下代码添加到 `Loan` 类：

```

public class Loan : INotifyPropertyChanged
{
    public double LoanAmount { get; set; }
    public double InterestRatePercent { get; set; }

    [field:NonSerialized()]
    public DateTime TimeLastLoaded { get; set; }

    public int Term { get; set; }

    private string customer;
    public string Customer
    {
        get { return customer; }
        set
        {
            customer = value;
            PropertyChanged?.Invoke(this,
                new PropertyChangedEventArgs(nameof(Customer)));
        }
    }

    [field: NonSerialized()]
    public event System.ComponentModel.PropertyChangedEventHandler PropertyChanged;

    public Loan(double loanAmount,
               double interestRate,
               int term,
               string customer)
    {
        this.LoanAmount = loanAmount;
        this.InterestRatePercent = interestRate;
        this.Term = term;
        this.customer = customer;
    }
}

```

还需要创建一个使用 `Loan` 类的应用程序。

串行化 Loan 对象

1. 打开 `Program.cs`。添加以下代码：

```
Loan TestLoan = new Loan(10000.0, 7.5, 36, "Neil Black");
```

添加 `PropertyChanged` 事件的事件处理程序和几行以修改 `Loan` 对象并显示此更改。你可以在下列代码中查看添加项：

```

TestLoan.PropertyChanged += (_, __) => Console.WriteLine($"New customer value: {TestLoan.Customer}");

TestLoan.Customer = "Henry Clay";
Console.WriteLine(TestLoan.InterestRatePercent);
TestLoan.InterestRatePercent = 7.1;
Console.WriteLine(TestLoan.InterestRatePercent);

```

现在，可运行该代码，并查看当前输出：

```
New customer value: Henry Clay  
7.5  
7.1
```

重复运行此应用程序始终编写相同值。每次运行该程序都会创建新的 Loan 对象。在现实生活中，利率会定期更改，但不必在每次运行应用程序时都更改利率。序列化代码表示在应用程序的实例之间保存最新的利率。下一步是通过向 Loan 类添加序列化来执行此操作。

使用序列化保持对象

为了保持 Loan 类的值，必须首先使用 `Serializable` 属性标记该类。将下面的代码添加到 Loan 类声明的上方：

```
[Serializable()]
```

`SerializableAttribute` 通知编译器可将类中的所有内容保留到文件中。因为 `PropertyChanged` 事件不表示应该存储的对象图的部分，所以它不应序列化。执行此操作可能将所有附加到该事件的对象序列化。可将 `NonSerializedAttribute` 添加到 `PropertyChanged` 事件处理程序的字段声明。

```
[field: NonSerialized()]  
public event System.ComponentModel.PropertyChangedEventHandler PropertyChanged;
```

从 C# 7.3 开始，可使用 `field` 目标值将特性附加到自动实现的属性的支持字段。以下代码添加 `TimeLastLoaded` 属性并将其标记为不可序列化：

```
[field:NonSerialized()]  
public DateTime TimeLastLoaded { get; set; }
```

下一步是向 LoanApp 应用程序添加序列化代码。为了将该类序列化并将其写入到文件，可使用 `System.IO` 和 `System.Runtime.Serialization.Formatters.Binary` 命名空间。为了避免键入完全限定的名称，可以添加对必要命名空间的引用，如以下代码所示：

```
using System.IO;  
using System.Runtime.Serialization.Formatters.Binary;
```

下一步是添加代码，在创建对象时对文件中的对象进行反序列化。向序列化数据的文件名的类中添加一个常量，如以下代码所示：

```
const string FileName = @"../../../../SavedLoan.bin";
```

接下来，在创建 `TestLoan` 对象的行后添加以下代码：

```
if (File.Exists(FileName))  
{  
    Console.WriteLine("Reading saved file");  
    Stream openFileStream = File.OpenRead(FileName);  
    BinaryFormatter deserializer = new BinaryFormatter();  
    TestLoan = (Loan)deserializer.Deserialize(openFileStream);  
    TestLoan.TimeLastLoaded = DateTime.Now;  
    openFileStream.Close();  
}
```

首先必须检查该文件是否存在。如果存在，则创建 `Stream` 类来读取二进制文件和 `BinaryFormatter` 类，以转换

该文件。还需将流类型转换为 Loan 对象类型。

然后必须添加代码以将该类序列化到文件中。以 `Main` 方式在现有代码后添加以下代码：

```
Stream SaveFileStream = File.Create(FileName);
BinaryFormatter serializer = new BinaryFormatter();
serializer.Serialize(SaveFileStream, TestLoan);
SaveFileStream.Close();
```

此时可再次生成并运行应用程序。首次运行时，请注意起始利率为 7.5，然后更改为 7.1。关闭该应用程序，然后重新运行。现在，应用程序打印已读取所保存文件的消息，即使在代码更改它之前，利率也是 7.1。

请参阅

- [序列化 \(C#\)](#)
- [C# 编程指南](#)

语句、表达式和运算符 (C# 编程指南)

2020/11/2 • [Edit Online](#)

构成应用程序的 C# 代码由关键字、表达式和运算符组成的语句所组成。本节包含有关这些 C# 程序基本元素的信息。

有关详情，请参阅：

- [语句](#)
- [运算符和表达式](#)
- [Expression-Bodied 成员](#)
- [匿名函数](#)
- [相等比较](#)

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](#)
- [强制转换和类型转换](#)

语句 (C# 编程指南)

2020/11/2 • [Edit Online](#)

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重语句块：

```
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：

C# 语句	C# 语句
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	<p>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。有关详细信息, 请参阅下列主题:</p> <ul style="list-style-type: none"> • if • else • switch • case
迭代语句	<p>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。有关详细信息, 请参阅下列主题:</p> <ul style="list-style-type: none"> • do • for • foreach • in • while
跳转语句	<p>跳转语句将控制转移给另一代码段。有关详细信息, 请参阅下列主题:</p> <ul style="list-style-type: none"> • break • continue • default • goto • return • yield
异常处理语句	<p>异常处理语句用于从运行时发生的异常情况正常恢复。有关详细信息, 请参阅下列主题:</p> <ul style="list-style-type: none"> • throw • try-catch • try-finally • try-catch-finally
Checked 和 unchecked	<p>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, 是否允许数值运算导致溢出。有关详细信息, 请参阅 checked 和 unchecked。</p>
<p><code>await</code> 语句</p>	<p>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 <code>await</code> 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。</p> <p>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。</p>

<code>yield return</code> 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 <code>yield return</code> 语句返回元素，每次返回一个。到达 <code>yield return</code> 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 。
<code>fixed</code> 语句	<code>fixed</code> 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 。
<code>lock</code> 语句	<code>lock</code> 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 <code>goto</code> 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。

```
// 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。

```
// 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：

```

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(包括 `do`、`while`、`for` 和 `foreach`)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：

```

// 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：

```

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：

```

if (b == true)
{
    // OK:
    System.DateTime d = System.DateTime.Now;
    System.Console.WriteLine(d.ToString("yyyy-MM-dd"));
}

```

嵌套语句块

语句块可以嵌套，如以下代码所示：

```
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：

```
// 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

- [C# 编程指南](#)
- [语句关键字](#)
- [C# 运算符和表达式](#)

Expression-bodied 成员 (C# 编程指南)

2020/11/2 • [Edit Online](#)

通过表达式主体定义，可采用非常简洁的可读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：

```
member => expression;
```

其中“expression”是有效的表达式。

C# 6 中引入了针对方法和只读属性的表达式主体定义支持，并在 C# 7.0 中进行了扩展。表达式主体定义可用于下表列出的类型成员：

成员	支持的 C# 版本
方法	C# 6
只读属性	C# 6
Property	C# 7.0
构造函数	C# 7.0
终结器	C# 7.0
索引器	C# 7.0

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。请注意，`ToString` 表达式主体定义中未使用 `return` 关键字。

```
using System;

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

从 C# 6 开始，可以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：

```
.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：

```
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

从 C# 7.0 开始，可以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：

```
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。

```
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。

```
using System;

public class Destroyer
{
    public override string ToString() => GetType().Name;

    ~Destroyer() => Console.WriteLine($"The {ToString()} destructor 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，该数组包含大量体育运动的名称。索引器的 `get` 和 `set` 访问器都以表达式主体定义的形式实现。

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

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\(C# 编程指南\)](#)。

匿名函数 - (C# 编程指南)

2020/11/2 • [Edit Online](#)

匿名函数是一个“内联”语句或表达式，可在需要委托类型的地方使用。可以使用匿名函数来初始化命名委托，或传递命名委托（而不是命名委托类型）作为方法参数。

可以使用 [lambda 表达式](#) 或 [匿名方法](#) 来创建匿名函数。建议使用 lambda 表达式，因为它们提供了更简洁和富有表现力的方式来编写内联代码。与匿名方法不同，某些类型的 lambda 表达式可以转换为表达式树类型。

C# 中委托的演变

在 C# 1.0 中，通过使用在代码中其他位置定义的方法显式初始化委托来创建委托的实例。C# 2.0 引入了匿名方法的概念，作为一种编写可在委托调用中执行的未命名内联语句块的方式。C# 3.0 引入了 lambda 表达式，这种表达式与匿名方法的概念类似，但更具表现力并且更简练。这两个功能统称为匿名函数。通常，面向 .NET Framework 3.5 或更高版本的应用程序应使用 lambda 表达式。

下面的示例演示从 C# 1.0 到 C# 3.0 委托创建过程的发展：

```
class Test
{
    delegate void TestDelegate(string s);
    static void M(string s)
    {
        Console.WriteLine(s);
    }

    static void Main(string[] args)
    {
        // Original delegate syntax required
        // initialization with a named method.
        TestDelegate testDelA = new TestDelegate(M);

        // C# 2.0: A delegate can be initialized with
        // inline code, called an "anonymous method." This
        // method takes a string as an input parameter.
        TestDelegate testDelB = delegate(string s) { Console.WriteLine(s); };

        // C# 3.0. A delegate can be initialized with
        // a lambda expression. The lambda also takes a string
        // as an input parameter (x). The type of x is inferred by the compiler.
        TestDelegate testDelC = (x) => { Console.WriteLine(x); };

        // Invoke the delegates.
        testDelA("Hello. My name is M and I write lines.");
        testDelB("That's nothing. I'm anonymous and ");
        testDelC("I'm a famous author.");

        // Keep console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}

/* Output:
Hello. My name is M and I write lines.
That's nothing. I'm anonymous and
I'm a famous author.
Press any key to exit.
*/
```

C# 语言规范

有关详细信息，请参阅 [C# 语言规范](#) 中的 [匿名函数表达式部分](#)。

请参阅

- [语句、表达式和运算符](#)
- [Lambda 表达式](#)
- [委托](#)
- [表达式树 \(C#\)](#)

如何在查询中使用 Lambda 表达式 (C# 编程指南)

2021/5/7 • [Edit Online](#)

不会直接在查询语法中使用 lambda 表达式，而是在方法调用中使用它们，并且查询表达式可以包含方法调用。事实上，一些查询操作只能采用方法语法进行表示。有关查询语法与方法语法之间的差异的详细信息，请参阅 [LINQ 中的查询语法和方法语法](#)。

示例

下面的示例演示如何通过 `Enumerable.Where` 标准查询运算符，在基于方法的查询中使用 lambda 表达式。请注意，此示例中的 `Where` 方法具有一个 `Func<T,TResult>` 委托类型的输入参数，该委托采用整数作为输入并返回一个布尔值。Lambda 表达式可以转换为该委托。如果这是使用 `Queryable.Where` 方法的 LINQ to SQL 查询，则参数类型会是 `Expression<Func<int,bool>>`，但 lambda 表达式看起来完全相同。有关表达式类型的详细信息，请参阅 [System.Linq.Expressions.Expression](#)。

```
class SimpleLambda
{
    static void Main()
    {

        // Data source.
        int[] scores = { 90, 71, 82, 93, 75, 82 };

        // The call to Count forces iteration of the source
        int highScoreCount = scores.Where(n => n > 80).Count();

        Console.WriteLine("{0} scores are greater than 80", highScoreCount);

        // Outputs: 4 scores are greater than 80
    }
}
```

下面的示例演示如何在查询表达式的方法调用中使用 lambda 表达式。需要 lambda 的原因是无法使用查询语法调用 `Sum` 标准查询运算符。

查询首先根据学生的年级(在 `GradeLevel` 枚举中定义)对学生进行分组。然后为每个组添加每个学生的总分。这需要两个 `Sum` 操作。内部 `Sum` 为每个学生计算总分，而外部 `Sum` 保留组中所有学生的正在运行的合并总分。

```
private static void TotalsByGradeLevel()
{
    // This query retrieves the total scores for First Year students, Second Years, and so on.
    // The outer Sum method uses a lambda in order to specify which numbers to add together.
    var categories =
        from student in students
        group student by student.Year into studentGroup
        select new { GradeLevel = studentGroup.Key, TotalScore = studentGroup.Sum(s => s.ExamScores.Sum()) };

    // Execute the query.
    foreach (var cat in categories)
    {
        Console.WriteLine("Key = {0} Sum = {1}", cat.GradeLevel, cat.TotalScore);
    }
}
/*
Outputs:
Key = SecondYear Sum = 1014
Key = ThirdYear Sum = 964
Key = FirstYear Sum = 1058
Key = FourthYear Sum = 974
*/
```

编译代码

若要运行此代码，请将方法复制并粘贴到[查询对象的集合](#)中提供的 `StudentClass` 中，然后从 `Main` 方法调用它。

请参阅

- [Lambda 表达式](#)
- [表达式树 \(C#\)](#)

相等性比较 (C# 编程指南)

2021/3/22 • [Edit Online](#)

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。

```
using System;
class Test
{
    public int Num { get; set; }
    public string Str { get; set; }

    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);

        // Keep the console open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
```

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。

```
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

TITLE	DESCRIPTION
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。
C# 编程指南	提供一些链接，这些链接指向重要 C# 语言功能以及通过 .NET 提供给 C# 的功能的相关详细信息。
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。

另请参阅

- [C# 编程指南](#)

如何为类或结构定义值相等性 (C# 编程指南)

2021/5/7 • [Edit Online](#)

[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。

NOTE

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。

```
using System;

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。

IMPORTANT

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：

```
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：

```
using System;

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:
        }
    }
}
```

```

// raise:
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）

2021/5/7 • [Edit Online](#)

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

```
using System;
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](#)

类型 (C# 编程指南)

2021/5/10 • [Edit Online](#)

类型、变量和值

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、参数数量以及类型和种类(值、引用或输出)。.NET 类库定义了一组内置数值类型以及表示各种逻辑构造的更复杂类型(如文件系统、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。
- 在运行时分配变量内存的位置。
- 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：

```
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;
```

NOTE

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：

```
// 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：

```
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` 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来表示整数、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

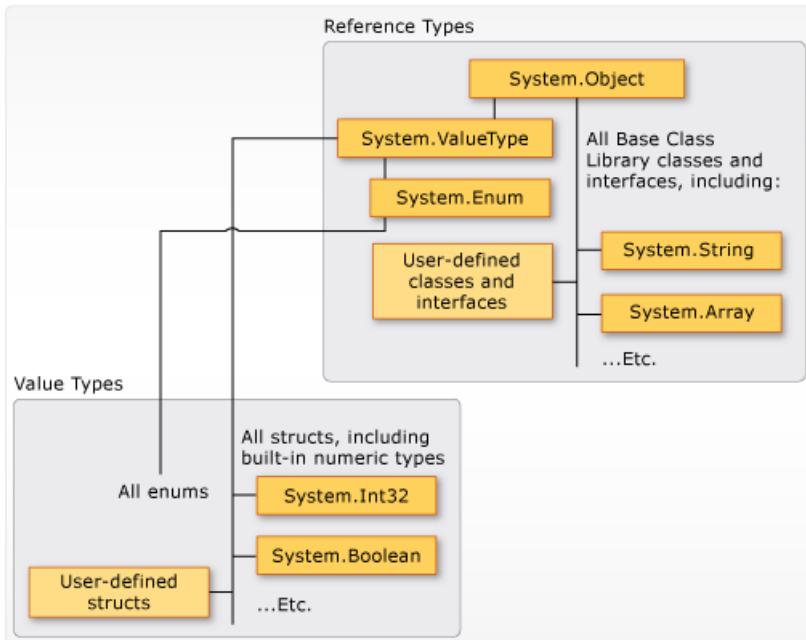
可以使用[结构](#)、[类](#)、[接口枚举](#)和[记录](#)构造来创建你自己的自定义类型。[.NET](#) 类库本身就是 Microsoft 提供的一组自定义类型，以供你在自己的应用程序中使用。默认情况下，类库中最常用的类型在任何 C# 程序中均可用。对于其他类型，只有在显式添加对定义这些类型的程序集的项目引用时才可用。编译器引用程序集之后，你可以声明在源代码的此程序集中声明的类型的变量(和常量)。有关详细信息，请参阅[.NET 类库](#)。

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# 关键字：`int` 等内置数值类型)最终都派生自单个基类型，即 `System.Object`(C# 关键字：`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。



NOTE

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。

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它们的值，这意味着在声明变量的任何上下文中内联分配内存。对于值类型变量，没有单独的堆分配或垃圾回收开销。

值类型分为两类：[结构](#)和[枚举](#)。

内置的数值类型是结构，它们具有可访问的字段和方法：

```
// 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：

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

值类型已密封，这意味着不能从任何值类型(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它实现的任何接口类型；强制转换会导致装箱操作发生，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：

```
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](#)。

另一种值类型是枚举。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：

```
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

定义为类、记录、委托、数组或接口的类型是引用类型。在运行时，当声明引用类型的变量时，该变量会一直包含值 `null`，直至使用 `new` 运算符显式创建对象，或者为该变量分配已经在其他位置使用 `new` 创建的对象，如下所示：

```
MyClass mc = new MyClass();
MyClass mc2 = mc;
```

接口必须与实现它的类对象一起初始化。如果 `MyClass` 实现 `IMyInterface`，则按以下示例所示创建 `IMyInterface` 的实例：

```
IMyInterface iface = new MyClass();
```

创建对象后，内存会在托管堆上进行分配，并且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 CLR 自动内存管理功能（称为“垃圾回收”）回收内存时都会产生开销。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。虽然数组隐式派生自 `System.Array` 类，但可以使用 C# 提供的简化语法声明和使用数组，如以下示例所示：

```
// 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，而其他类可以从你的类继承并重写虚拟方法。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，若要将值 4.56 指定为应按浮点值处理，请在数字后面追加“f”或“F”：`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：

```
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，作为客户端代码在创建类型实例时将提供的实际类型（具体类型）的占位符的类型。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，例如字符串：

```
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

如前所述，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 `?`，创建可以为 `null` 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，可以为 `null` 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：

```
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：

```
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：

- [强制转换和类型转换](#)
- [装箱和取消装箱](#)
- [使用类型 dynamic](#)
- [值类型](#)
- [引用类型](#)
- [类、结构和记录](#)
- [匿名类型](#)
- [泛型](#)

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。

请参阅

- [C# 参考](#)
- [C# 编程指南](#)
- [XML 数据类型转换](#)
- [整型类型](#)

强制转换和类型转换 (C# 编程指南)

2020/11/2 • [Edit Online](#)

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，如下代码所示：

```
int i;

// error CS0029: Cannot 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**：由于这种转换始终会成功且不会导致数据丢失，因此无需使用任何特殊语法。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**：用户定义的转换是使用特殊方法执行，这些方法可定义为在没有基类和派生类关系的自定义类型之间启用显式转换和隐式转换。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`。

```
// 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。

```
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`。如不强制转换则该程序不会进行编译。

```
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：

```
// 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](#)。

```
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](#)。

请参阅

- [C# 编程指南](#)
- [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](#)

装箱和取消装箱 (C# 编程指南)

2020/11/2 • [Edit Online](#)

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`。

```
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`：

```
o = 123;
i = (int)o; // unboxing
```

以下示例演示如何在 C# 中使用装箱。

```
byte[] array = { 0x64, 0x6f, 0x74, 0x63, 0x65, 0x74 };

string hexValue = Convert.ToString(array);
Console.WriteLine(hexValue);

/*Output:
646F74636574
*/
```

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。

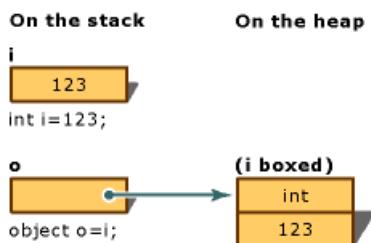
请看以下值类型变量的声明：

```
int i = 123;
```

以下语句对变量 `i` 隐式应用了装箱操作：

```
// 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：

```
int i = 123;
object o = (object)i; // explicit boxing
```

描述

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。

示例

```
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

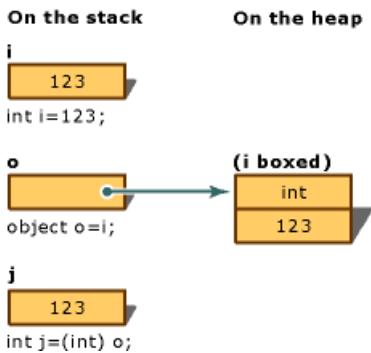
取消装箱是从 `object` 类型到值类型或从接口类型到实现该接口的值类型的显式转换。取消装箱操作包括：

- 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：

```
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。

```
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：

```
int j = (short) o;
```

更改为：

```
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)

2021/3/5 • [Edit Online](#)

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。

类型	方法
<code>bool</code>	<code>ToBoolean(Byte[], Int32)</code>
<code>char</code>	<code>ToChar(Byte[], Int32)</code>
<code>double</code>	<code>ToDouble(Byte[], Int32)</code>
<code>short</code>	<code>ToInt16(Byte[], Int32)</code>
<code>int</code>	<code>ToInt32(Byte[], Int32)</code>
<code>long</code>	<code>ToInt64(Byte[], Int32)</code>
<code>float</code>	<code>ToSingle(Byte[], Int32)</code>
<code>ushort</code>	<code>ToUInt16(Byte[], Int32)</code>
<code>uint</code>	<code>ToUInt32(Byte[], Int32)</code>
<code>ulong</code>	<code>ToUInt64(Byte[], Int32)</code>

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。

NOTE

输出可能会根据计算机体系结构的字节顺序而不同。

```
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
```

示例

在本示例中，将调用 [BitConverter](#) 类的 [GetBytes\(Int32\)](#) 方法，将 `int` 转换为字节数组。

NOTE

输出可能会根据计算机体系结构的字节顺序而不同。

```
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)

2021/3/9 • [Edit Online](#)

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。

```

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。

```
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。

类型	方法
<code>decimal</code>	<code>ToDecimal(String)</code>
<code>float</code>	<code>ToSingle(String)</code>

<code>double</code>	<code>ToDouble(String)</code>
<code>short</code>	<code>ToInt16(String)</code>
<code>int</code>	<code>ToInt32(String)</code>
<code>long</code>	<code>ToInt64(String)</code>
<code>ushort</code>	<code>ToUInt16(String)</code>
<code>uint</code>	<code>ToUInt32(String)</code>
<code>ulong</code>	<code>ToUInt64(String)</code>

下面的示例调用 `Convert.ToInt32(String)` 方法将输入字符串转换为 `int`。该示例将捕获此方法可能引发的最常见的两个异常：`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。

```
using System;

public class ConvertStringExample1
{
    static void Main(string[] args)
    {
        int numVal = -1;
        bool repeat = true;

        while (repeat)
        {
            Console.WriteLine("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
```

如何在十六进制字符串与数值类型之间转换 (C# 编程指南)

2021/3/5 • [Edit Online](#)

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。

```
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
*/
```

示例

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。

```

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 !
*/

```

示例

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。

```

string hexString = "8E2";
int num = Int32.Parse(hexString, System.Globalization.NumberStyles.HexNumber);
Console.WriteLine(num);
//Output: 2274

```

示例

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`。

```

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

```

示例

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。

```
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
*/
```

示例

下面的示例演示如何通过调用 .NET 5.0 中引入的 [Convert.ToString](#) 方法，将字节数组转换为十六进制字符串。

```
byte[] array = { 0x64, 0x6f, 0x74, 0x63, 0x65, 0x74 };

string hexValue = Convert.ToString(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](#)

使用类型 dynamic (C# 编程指南)

2021/5/7 • [Edit Online](#)

C# 4 引入了一个新类型 `dynamic`。该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。在编译时，将假定类型化为 `dynamic` 的元素支持任何操作。因此，不必考虑对象是从 COM API、从动态语言(例如 IronPython)、从 HTML 文档对象模型 (DOM)、从反射还是从程序中的其他位置获取自己的值。但是，如果代码无效，则在运行时会捕获到错误。

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。它将在运行时被捕获，并导致运行时异常。

```
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();
}
```

```
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类型化为 `dynamic` 的对象或表达式。在运行时，将对存储的信息进行检查，并且任何无效的语句都将导致运行时异常。

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”。

```
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`：

```
var testInstance = new ExampleClass(d);
```

下一节“转换”中介绍了转换示例。

转换

动态对象和其他类型之间的转换非常简单。这样，开发人员将能够在动态行为和非动态行为之间切换。

任何对象都可隐式转换为动态类型，如以下示例所示。

```
dynamic d1 = 7;
dynamic d2 = "a string";
dynamic d3 = System.DateTime.Today;
dynamic d4 = System.Diagnostics.Process.GetProcesses();
```

反之，隐式转换也可动态地应用于类型为 `dynamic` 的任何表达式。

```
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` 方法定义为接受字符串参数，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。

```
// 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) 是 .NET Framework 4 中引入的 API。它提供了支持 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

C# 4 包括若干功能，这些功能改善了与 COM API（例如 Office 自动化 API）的互操作体验。这些改进之处包括 `dynamic` 类型以及[命名参数和可选参数](#)的用法。

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。这样，就必须显式强制转换

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。例如，以下语句对比了在使用 `dynamic` 类型和不使用 `dynamic` 类型的情况下如何访问 Microsoft Office Excel 电子表格中的单元格。

```
// Before the introduction of dynamic.  
((Excel.Range)excelApp.Cells[1, 1]).Value2 = "Name";  
Excel.Range range2008 = (Excel.Range)excelApp.Cells[1, 1];
```

```
// After the introduction of dynamic, the access to the Value property and  
// the conversion to Excel.Range are handled by the run-time COM binder.  
excelApp.Cells[1, 1].Value = "Name";  
Excel.Range range2010 = excelApp.Cells[1, 1];
```

相关主题

TITLE	IF
dynamic	描述 <code>dynamic</code> 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 <code>IronPython</code> 库的对象的分步说明。
如何使用 C# 功能访问 Office 互操作对象	演示如何创建一个项目，该项目使用命名参数和可选参数、 <code>dynamic</code> 类型以及可简化对 Office API 对象的访问的其他增强功能。

演练：创建并使用动态对象（C# 和 Visual Basic）

2021/5/7 • [Edit Online](#)

动态对象会在运行时（而非编译时）公开属性和方法等成员。这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。在 C# 中，将晚期绑定对象的类型指定为 `dynamic`。在 Visual Basic 中，将晚期绑定对象的类型指定为 `Object`。有关详细信息，请参阅[动态和早期绑定和晚期绑定](#)。

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。

你可以完成其中一个，也可以同时完成两个；如果要完成两个，则顺序无关紧要。

先决条件

- 安装了具有 .NET Core 桌面开发工作负载的 [Visual Studio 2019 版本 16.9 或更高版本](#)。选择此工作负载时，将自动安装 .NET 5.0 SDK。

NOTE

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)`，则动态类将在每行的开头搜索“Sample”，但不会删除前导空格和尾部空格。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

1. 启动 Visual Studio。
2. 选择“创建新项目”。
3. 在“创建新项目”对话框中，选择 C# 或 Visual Basic，然后依次选择“控制台应用程序”和“下一步”。
4. 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”。
5. 在“其他信息”对话框中，为“目标框架”选择“.NET 5.0 (当前)”，然后选择“创建”。

创建新项目。

6. 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”。

这将添加一个包含 `ReadOnlyFile` 类的新文件。

7. 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。

```
using System.IO;
using System.Dynamic;
```

```
Imports System.IO
Imports System.Dynamic
```

8. 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。

```
public enum StringSearchOption
{
    StartsWith,
    Contains,
    EndsWith
}
```

```
Public Enum StringSearchOption
    StartsWith
    Contains
    EndsWith
End Enum
```

9. 更新类语句以继承 `DynamicObject` 类，如以下代码示例所示。

```
class ReadOnlyFile : DynamicObject
```

```
Public Class ReadOnlyFile
    Inherits DynamicObject
```

10. 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。

```
// 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;  
}
```

```
' Store the path to the file and the initial line count value.  
Private p_filePath As String  
  
' Public constructor. Verify that file exists and store the path in  
' the private variable.  
Public Sub New(ByVal filePath As String)  
    If Not File.Exists(filePath) Then  
        Throw New Exception("File path does not exist.")  
    End If  
  
    p_filePath = filePath  
End Sub
```

11. 将下面的 `GetPropertyValues` 方法添加到 `ReadOnlyFile` 类。`GetPropertyValues` 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 `GetPropertyValues` 方法以检索其各自的结果。

```
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;
}
```

```

Public Function GetPropertyValue(ByVal propertyName As String,
                               Optional ByVal StringSearchOption As StringSearchOption =
StringSearchOption.StartsWith,
                               Optional ByVal trimSpaces As Boolean = True) As List(Of String)

    Dim sr As StreamReader = Nothing
    Dim results As New List(Of String)
    Dim line = ""
    Dim testLine = ""

    Try
        sr = New StreamReader(p_filePath)

        While Not sr.EndOfStream
            line = sr.ReadLine()

            ' Perform a case-insensitive search by using the specified search options.
            testLine = UCASE(line)
            If trimSpaces Then testLine = Trim(testLine)

            Select Case StringSearchOption
                Case StringSearchOption.StartsWith
                    If testLine.StartsWith(UCASE(propertyName)) Then results.Add(line)
                Case StringSearchOption.Contains
                    If testLine.Contains(UCASE(propertyName)) Then results.Add(line)
                Case StringSearchOption.EndsWith
                    If testLine.EndsWith(UCASE(propertyName)) Then results.Add(line)
            End Select
        End While
    Catch
        ' Trap any exception that occurs in reading the file and return Nothing.
        results = Nothing
    Finally
        If sr IsNot Nothing Then sr.Close()
    End Try

    Return results
End Function

```

12. 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`。

```

// 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;
}

```

```

' Implement the TryGetMember method of the DynamicObject class for dynamic member calls.
Public Overrides Function TryGetMember(ByVal binder As GetMemberBinder,
                                       ByRef result As Object) As Boolean
    result = GetPropertyValue(binder.Name)
    Return If(result Is Nothing, False, True)
End Function

```

13. 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，则将它们传递给 `GetPropertyValues` 方法以检索结果。

```
// 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 = GetPropertyValues(binder.Name, StringSearchOption, trimSpaces);

    return result == null ? false : true;
}
```

```

' Implement the TryInvokeMember method of the DynamicObject class for
' dynamic member calls that have arguments.
Public Overrides Function TryInvokeMember(ByVal binder As InvokeMemberBinder,
                                         ByVal args() As Object,
                                         ByRef result As Object) As Boolean

    Dim StringSearchOption As StringSearchOption = StringSearchOption.StartsWith
    Dim trimSpaces = True

    Try
        If args.Length > 0 Then StringSearchOption = CType(args(0), StringSearchOption)
    Catch
        Throw New ArgumentException("StringSearchOption argument must be a StringSearchOption enum
value.")
    End Try

    Try
        If args.Length > 1 Then trimSpaces = CType(args(1), Boolean)
    Catch
        Throw New ArgumentException("trimSpaces argument must be a Boolean value.")
    End Try

    result = GetPropertyValue(binder.Name, StringSearchOption, trimSpaces)

    Return If(result Is Nothing, False, True)
End Function

```

14. 保存并关闭文件。

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“文本文档”模板。保留“名称”框中的默认名称 TextFile1.txt，然后单击“添加”。这会将一个新的文本文档添加到项目中。
- 将以下文本复制到 TextFile1.txt 文件。

```

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

```

3. 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.vb 文件（如果使用 Visual Basic）或 Program.cs 文件（如果使用 Visual C#）。
- 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。

```
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);
}
```

```
Dim rFile As Object = New ReadOnlyFile("../..\\TextFile1.txt")
For Each line In rFile.Customer
    Console.WriteLine(line)
Next
Console.WriteLine("-----")
For Each line In rFile.Customer(StringSearchOption.Contains, True)
    Console.WriteLine(line)
Next
```

3. 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

1. 在 Visual Studio 中，选择“文件” > “新建” > “项目”。
2. 在“创建新项目”对话框中，选择 C# 或 Visual Basic，然后依次选择“控制台应用程序”和“下一步”。
3. 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”。
4. 在“其他信息”对话框中，为“目标框架”选择“.NET 5.0 (当前)”，然后选择“创建”。

创建新项目。

5. 安装 IronPython NuGet 包。
6. 如果使用 Visual Basic，请编辑 `Program.vb` 文件。如果使用 Visual C#，请编辑 `Program.cs` 文件。
7. 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。

```
using System.Linq;
using Microsoft.Scripting.Hosting;
using IronPython.Hosting;
```

```
Imports Microsoft.Scripting.Hosting
Imports IronPython.Hosting
Imports System.Linq
```

8. 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`。

```

// 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.");

```

```

' 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")
Dim py = Python.CreateRuntime()
Dim random As Object = py.UseFile("random.py")
Console.WriteLine("random.py loaded.")

```

9. 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。

```

// 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("-----");
}

```

```

' Initialize an enumerable set of integers.
Dim items = Enumerable.Range(1, 7).ToArray()

' Randomly shuffle the array of integers by using IronPython.
For i = 0 To 4
    random.shuffle(items)
    For Each item In items
        Console.WriteLine(item)
    Next
    Console.WriteLine("-----")
Next

```

10. 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`

- 早期绑定和后期绑定
- dynamic
- 实现动态接口(可从 Microsoft TechNet 下载 PDF)

类、结构和记录 (C# 编程指南)

2021/5/7 • [Edit Online](#)

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，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。

类或记录是引用类型。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。

一般来说，类用于对更复杂的行为或应在类对象创建后进行修改的数据建模。结构最适用于所含大部分数据不得在结构创建后进行修改的小型数据结构。记录类型可用于所含大部分是不得在创建对象后修改的数据的大型数据结构。

示例

在以下示例中，`ProgrammingGuide` 命名空间中的 `CustomClass` 有以下三个成员：实例构造函数、`Number` 属性和 `Multiply` 方法。`Program` 类中的 `Main` 方法创建 `CustomClass` 的实例(对象)，此对象的方法和属性可使用点表示法进行访问。

```

using System;

namespace ProgrammingGuide
{
    // Class definition.
    public class CustomClass
    {
        // Class members.
        //
        // Property.
        public int Number { get; set; }

        // Method.
        public int Multiply(int num)
        {
            return num * Number;
        }

        // Instance Constructor.
        public CustomClass()
        {
            Number = 0;
        }
    }

    // Another class definition that contains Main, the program entry point.
    class Program
    {
        static void Main(string[] args)
        {
            // Create an object of type CustomClass.
            CustomClass custClass = new CustomClass();

            // Set the value of the public property.
            custClass.Number = 27;

            // Call the public method.
            int result = custClass.Multiply(4);
            Console.WriteLine($"The result is {result}.");
        }
    }
}

// The example displays the following output:
//     The result is 108.

```

封装

封装有时称为面向对象的编程的第一支柱或原则。根据封装原则，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](#)。

成员

所有方法、字段、常量、属性和事件都必须在类型中进行声明；这些被称为类型的 成员。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（对于顶级语句的情况，是隐式声明）。

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](#)

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`。有关详细信息，请参阅[访问修饰符](#)。

继承

类(而非结构)支持继承的概念。派生自另一个类(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)。有关详细信息，请参阅[继承和多态性](#)。

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[抽象类、密封类及类成员](#)。

界面

类、结构和记录可以继承多个接口。继承自接口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进行定义的。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[静态](#)。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

无需显式调用构造函数，即可实例化和初始化类或结构对象和对象集合。有关详细信息，请参阅[对象和集合初始值设定项](#)。

匿名类型

如果不方便或没有必要创建已命名的类（例如，使用无需保留或传递给其他方法的数据结构填充列表时），可以使用匿名类型。有关详细信息，请参阅[匿名类型](#)。

扩展方法

可以单独创建类型（其方法可以调用，就像它们属于原始类型一样）来“扩展”类，而无需创建派生类。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[隐式类型局部变量](#)。

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。记录提供以下功能：

- 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值都相同时，它们是相等的。记录与类不同，类使用引用相等性，即：如果两个类类型的变量引用相同的对象，则它们是相等的。

- 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。

- 支持继承层次结构。

支持继承，因为记录的本质是类，而不是结构。

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。

请参阅

- [类](#)

- 对象
- 结构类型
- 记录
- C# 编程指南

类 (C# 编程指南)

2021/3/5 • [Edit Online](#)

引用类型

定义为 [类](#) 的一个类型是 [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可能已在其他位置创建的兼容类型的对象，如下面的示例所示：

```
//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。当分配托管堆上的类型和由 CLR 的自动内存管理功能对其进行回收（称为 [垃圾回收](#)）时，需要开销。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：

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

`class` 关键字前面是访问级别。因为此例中使用的是 `public`，所以任何人都可以创建此类的实例。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，后跟对象要基于的类的名称，来创建对象，如：

```
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：

```
Customer object2;
```

不建议创建这样一个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。但实际上可以使用这类引用来引用某个对象，方法是创建新对象，或者将其分配给现有对象，例如：

```
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。

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，可以继承自其他任何未定义为 `sealed` 的类，而且其他类也可以继承自你的类并重写类虚方法。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：

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

类声明基类时，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](#)。

与 C++ 不同，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)。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，`sealed`(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](#)。

示例

以下示例定义了一个公共类，该类包含一个[自动实现的属性](#)、一个方法和一个名为构造函数的特殊方法。有关详细信息，请参阅[属性、方法和构造函数](#)主题。然后使用 `new` 关键字实例化类的实例。

```
using System;

public class Person
{
    // Constructor that takes no arguments:
    public Person()
    {
        Name = "unknown";
    }

    // Constructor that takes one argument:
    public Person(string name)
    {
        Name = name;
    }

    // Auto-implemented readonly property:
    public string Name { get; }

    // Method that overrides the base class (System.Object) implementation.
    public override string ToString()
    {
        return Name;
    }
}
class TestPerson
{
    static void Main()
    {
        // Call the constructor that has no parameters.
        var person1 = new Person();
        Console.WriteLine(person1.Name);

        // Call the constructor that has one parameter.
        var person2 = new Person("Sarah Jones");
        Console.WriteLine(person2.Name);
        // Get the string representation of the person2 instance.
        Console.WriteLine(person2);

        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
// Output:
// unknown
// Sarah Jones
// Sarah Jones
```

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](#)
- [面向对象的编程](#)
- [多态性](#)
- [标识符名称](#)
- [成员](#)
- [方法](#)
- [构造函数](#)

- 终结器
- 对象

记录 (C# 编程指南)

2021/5/7 • [Edit Online](#)

记录是一个类，它为使用数据模型提供特定的语法和行为。有关类的详细信息，请查看类(C# 编程指南)。

何时使用记录

在下列情况下，请考虑使用记录而不是类：

- 你想要定义对象不可变的引用类型。
- 你想要定义依赖值相等性的数据模型。

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。

值相等性

对记录来说，值相等性表示当类型匹配且所有属性和字段值都匹配时，记录类型的两个变量相等。对于其他引用类型(例如类)，相等性是指引用相等性。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。

记录与类的区别

声明和实例化类时使用的语法与操作记录时的相同。只需将 record 关键字替换为 class 关键字即可。同样地，记录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：

- 可使用位置参数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。

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：

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

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

下面的示例演示了记录中的值相等性：

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

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

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

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

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：

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

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

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

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

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

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。

请参阅

- [类\(C# 编程指南\)](#)
- [记录\(C# 参考\)](#)
- [C# 编程指南](#)
- [面向对象的编程](#)
- [多态性](#)
- [标识符名称](#)
- [成员](#)

- 方法
- 构造函数
- 终结器
- 对象

对象 (C# 编程指南)

2021/3/5 • [Edit Online](#)

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。

NOTE

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对象分配给第一个对象，则两个变量都引用该地址的对象。这一点将在本主题后面部分进行更详细的讨论。

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）。

```

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);

        // Keep the console open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
/*
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。结构的实例也可以使用 `new` 运算符来创建，但这不是必需的，如下面的示例所示：

```

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);

        // Keep the console open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
/*
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](#)。

NOTE

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 `Equals` 方法。

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：

```
// 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;
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](#)

- 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](#)

请参阅

- [C# 编程指南](#)
- [object](#)
- [继承](#)
- [class](#)
- [结构类型](#)
- [new 运算符](#)
- [常规类型系统](#)

继承 (C# 编程指南)

2021/5/10 • [Edit Online](#)

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。

NOTE

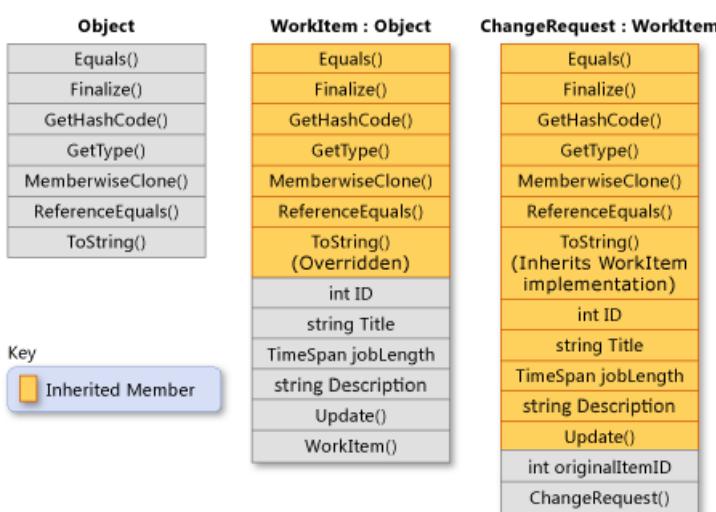
结构不支持继承，但它们可以实现接口。有关详细信息，请参阅[接口](#)。

从概念上讲，派生类是基类的专业化。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业化。

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](#)的文章中的语言参考部分。

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` 添加了自己的五个成员。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：

```
// 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
    }
}

```

```

    // 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：

```

// 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。有关详细信息，请参阅[抽象类、密封类及类成员](#)。

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。有关详细信息，请参阅[抽象类、密封类和类成员](#)。

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](#)。

请参阅

- [C# 编程指南](#)
- [类、结构和记录](#)
- [class](#)

多态性 (C# 编程指南)

2020/11/2 • [Edit Online](#)

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`。

```
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();
    }
}
```

若要更新绘图画面，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)。

```

// 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，它会获得基类的所有方法、字段、属性和事件。派生类的设计器可以针对虚拟方法的行为做出不同的选择：

- 派生类可以重写基类中的虚拟成员，并定义新行为。
- 派生类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：

```

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：

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

BaseClass A = (BaseClass)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。有关详细信息，请参阅[接口](#)。

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：

```
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：

```
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：

```
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：

```
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：

```
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：

```
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](#)。

NOTE

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。

本节内容

- [使用 Override 和 New 关键字进行版本控制](#)
- [了解何时使用 Override 和 New 关键字](#)
- [如何替代 ToString 方法](#)

请参阅

- [C# 编程指南](#)
- [继承](#)

- 抽象类、密封类及类成员
- 方法
- 事件
- 属性
- 索引器
- 类型

使用 Override 和 New 关键字进行版本控制 (C# 编程指南)

2020/11/2 • [Edit Online](#)

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：

```
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：

```
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：

```
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：

```
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：

```
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：

```
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：

```
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：

```
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：

```
((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](#)
- [类和结构](#)
- [方法](#)
- [继承](#)

了解何时使用 Override 和 New 关键字 (C# 编程指南)

2021/5/10 • [Edit Online](#)

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`。

```
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。

```
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。

```
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。

```
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。

```
// 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。

```
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。

```
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。

```
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。

```
// 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。

```
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`。

```
// 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`。

```

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。

```

// 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。

```

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`。

```
// 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。

```
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。

```
// 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。

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace OverrideAndNew2
{
    class Program
    {
        static void Main(string[] args)
```

```
        // 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 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 TestCars4()  
    {  
        System.Console.WriteLine("\nTestCars4");  
        System.Console.WriteLine("-----");  
  
        TestCars1();  
  
        TestCars2();  
  
        TestCars3();  
  
        TestCars4();  
    }  
}
```

```

// 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](#)
- [类、结构和记录](#)
- [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)

2021/5/10 • [Edit Online](#)

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:

```
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](#)。

IMPORTANT

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:

```
public override string ToString(){}  
// Output:  
// Person: John 12
```

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。

```
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:

```
Person person = new Person { Name = "John", Age = 12 };
Console.WriteLine(person);
// Output:
// Person: John 12
```

请参阅

- [IFormattable](#)
- [C# 编程指南](#)
- [类、结构和记录](#)
- [字符串](#)
- [string](#)
- [override](#)
- [virtual](#)
- [格式设置类型](#)

成员 (C# 编程指南)

2020/11/2 • [Edit Online](#)

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：

成员类型	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。

请参阅

- [C# 编程指南](#)
- [类](#)

抽象类、密封类及类成员 (C# 编程指南)

2021/5/10 • [Edit Online](#)

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：

```
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：

```
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：

```
// 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：

```
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：

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

另请参阅

- [C# 编程指南](#)
- [类、结构和记录](#)
- [继承](#)
- [方法](#)
- [字段](#)
- [如何定义抽象属性](#)

静态类和静态类成员 (C# 编程指南)

2021/3/5 • [Edit Online](#)

静态类基本上与非静态类相同，但存在一个差异：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：

```
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。

```
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，加载引用该类的程序时，.NET 运行时会加载静态类的类型信息。程序无法确切指定类加载的时间。但是，可保证进行加载，以及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。

NOTE

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。
- 无法进行实例化。
- 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。编译器可保证无法创建此类的实例。

静态类会进行密封，因此不能继承。它们不能继承自任何类（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

下面是静态类的示例，该类包含将温度从摄氏度从华氏度以及从华氏度转换为摄氏度的两个方法：

```

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());
                Console.WriteLine("Temperature in Fahrenheit: {0:F2}", F);
                break;

            case "2":
                Console.Write("Please enter the Fahrenheit temperature: ");
                C = TemperatureConverter.FahrenheitToCelsius(Console.ReadLine());
                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。即使未创建类的任何实例，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已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`，不过 `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（即在方法范围内声明的变量）。

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：

```
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：

```
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](#) 和 [静态和实例成员](#)。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](#)
- [static](#)
- [类](#)
- [class](#)
- [静态构造函数](#)
- [实例构造函数](#)

访问修饰符 (C# 编程指南)

2021/5/10 • [Edit Online](#)

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。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：

- **public**: 同一程序集中的任何其他代码或引用该程序集的其他程序集都可以访问该类型或成员。
- **private**: 只有同一 `class` 或 `struct` 中的代码可以访问该类型或成员。
- **protected**: 只有同一 `class` 或者从该 `class` 派生的 `class` 中的代码可以访问该类型或成员。
- **internal**: 同一程序集中的任何代码都可以访问该类型或成员，但其他程序集中的代码不可以。
- **protected internal**: 该类型或成员可由对其进行声明的程序集或另一程序集中的派生 `class` 中的任何代码访问。
- **private protected**: 只有在其声明程序集内，通过相同 `class` 中的代码或派生自该 `class` 的类型，才能访问类型或成员。

下面的示例演示如何在类型和成员上指定访问修饰符：

```
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，类型成员的可访问性受到其包含类型的可访问性的限制。

类、记录和结构可访问性

直接在命名空间中声明的类、记录和结构(即，没有嵌套在其他类或结构中的类、记录和结构)可以为 `public` 或 `internal`。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`。不能从包含类型的外部访问私有嵌套类型。

派生类和派生记录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](#)。

类、记录和结构成员可访问性

可以使用六种访问类型中的任意一种声明类和记录成员(包括嵌套的类、记录和结构)。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，如果内部类的 `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](#)。

终结器不能具有可访问性修饰符。

若要设置 `class`、`record` 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。

```
// 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; }  
    }  
}
```

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。这最适用于静态方法，以提供类的所有实现器需要的常见实现。

枚举成员始终为 `public`，并且不能应用任何访问修饰符。

委托类似于类和结构。默认情况下，当在命名空间内直接声明它们时，它们具有 `internal` 访问级别，当将它们嵌套在命名空间内时，它们具有 `private` 访问级别。

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](#)
- [类、结构和记录](#)
- [接口](#)
- [private](#)
- [public](#)
- [internal](#)
- [受保护](#)
- [protected internal](#)
- [private protected](#)
- [class](#)
- [struct](#)
- [interface](#)

字段 (C# 编程指南)

2021/5/10 • [Edit Online](#)

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，静态字段属于类本身，并在该类的所有实例之间共享。只能使用类名称访问静态字段。如果按实例名称访问静态字段，将出现[CS0176](#) 编译时错误。

通常情况下，应仅对具有 private 或 protected 可访问性的变量使用字段。类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。

字段通常存储必须对多个类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。

字段是通过指定该字段的访问级别在类块中声明的，其后跟字段的类型，再跟字段的名称。例如：

```

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();
            }
        }
    }

    // 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();
        }
    }

    public TimeSpan GetTimeSpan(string dateString)
    {
        DateTime dt = Convert.ToDateTime(dateString);

        if (dt.Ticks < _date.Ticks)
        {
            return _date - dt;
        }
        else
        {
            throw new ArgumentOutOfRangeException();
        }
    }
}

```

若要访问对象中的字段，请在对象名称后添加一个句点，后跟字段的名称，如 `objectname._fieldName` 中所示。例如：

```
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`，如下示例所示：

```
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](#)。

NOTE

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[静态](#)。这可使字段可供调用方在任何时候进行调用，即使不存在任何类的实例。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[只读](#)。只能在初始化期间或在构造函数中为只读字段赋值。`static readonly` 字段非常类似于常量，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](#)。

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](#)
- [类、结构和记录](#)
- [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)

2021/5/10 • [Edit Online](#)

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。仅 C# 内置类型(不包括 `System.Object`)可声明为 `const`。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：

```
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。

NOTE

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：

```
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：

```
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：

```
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。

请参阅

- [C# 编程指南](#)
- [类、结构和记录](#)
- [属性](#)
- [类型](#)
- [readonly](#)
- [C# 不可变性\(第一部分\):不可变性种类](#)

如何定义抽象属性 (C# 编程指南)

2021/5/10 • [Edit Online](#)

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。

```
// 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：

```
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。

示例

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。

```

// 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;
        }
    }
}

```

示例

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。

```
// 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
*/
```

请参阅

- [C# 编程指南](#)
- [类、结构和记录](#)
- [抽象类、密封类及类成员](#)
- [属性](#)

如何在 C# 中定义常量

2021/5/10 • [Edit Online](#)

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”）。

NOTE

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

```
using System;

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

- [类、结构和记录](#)

属性 (C# 编程指南)

2021/5/7 • [Edit Online](#)

属性是一种成员，它提供灵活的机制来读取、写入或计算私有字段的值。属性可用作公共数据成员，但它们实际上被称为 [访问器](#) 的特殊方法。这使得可以轻松访问数据，还有助于提高方法的安全性和灵活性。

属性概述

- 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。在 C# 9 及更高版本中，`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 [只写属性](#)(有 `set` 访问器，但没有 `get` 访问器)。只写属性很少出现，常用于限制对敏感数据的访问。
- 不需要自定义访问器代码的简单属性可以作为[表达式主体定义](#)或[自动实现的属性](#)来实现。

具有支持字段的属性

有一个实现属性的基本模式，该模式使用私有支持字段来设置和检索属性值。`get` 访问器返回私有字段的值，`set` 访问器在向私有字段赋值之前可能会执行一些数据验证。这两个访问器还可以在存储或返回数据之前对其进行某些转换或计算。

下面的示例阐释了此模式。在此示例中，`TimePeriod` 类表示时间间隔。在内部，该类将时间间隔以秒为单位存储在名为 `_seconds` 的私有字段中。名为 `Hours` 的读-写属性允许客户以小时为单位指定时间间隔。`get` 和 `set` 访问器都会执行小时与秒之间的必要转换。此外，`set` 访问器还会验证数据，如果小时数无效，则引发 [ArgumentOutOfRangeException](#)。

```

using System;

class TimePeriod
{
    private double _seconds;

    public double Hours
    {
        get { return _seconds / 3600; }
        set {
            if (value < 0 || value > 24)
                throw new ArgumentOutOfRangeException(
                    $"{nameof(value)} must be between 0 and 24.");
            _seconds = value * 3600;
        }
    }
}

class Program
{
    static void Main()
    {
        TimePeriod t = new TimePeriod();
        // The property assignment causes the 'set' accessor to be called.
        t.Hours = 24;

        // Retrieving the property causes the 'get' accessor to be called.
        Console.WriteLine($"Time in hours: {t.Hours}");
    }
}
// The example displays the following output:
//     Time in hours: 24

```

表达式主体定义

属性访问器通常由单行语句组成，这些语句只分配或只返回表达式的结果。可以将这些属性作为 expression-bodied 成员来实现。`=>` 符号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。

从 C# 6 开始，只读属性可以将 `get` 访问器作为 expression-bodied 成员实现。在这种情况下，既不使用 `get` 访问器关键字，也不使用 `return` 关键字。下面的示例将只读 `Name` 属性作为 expression-bodied 成员实现。

```
using System;

public class Person
{
    private string _firstName;
    private string _lastName;

    public Person(string first, string last)
    {
        _firstName = first;
        _lastName = last;
    }

    public string Name => $"{_firstName} {_lastName}";
}

public class Example
{
    public static void Main()
    {
        var person = new Person("Magnus", "Hedlund");
        Console.WriteLine(person.Name);
    }
}

// The example displays the following output:
//      Magnus Hedlund
```

从 C# 7.0 开始，`get` 和 `set` 访问器都可以作为 expression-bodied 成员实现。在这种情况下，必须使用 `get` 和 `set` 关键字。下面的示例阐释如何为这两个访问器使用表达式主体定义。请注意，`return` 关键字不与 `get` 访问器搭配使用。

```
using System;

public class SaleItem
{
    string _name;
    decimal _cost;

    public SaleItem(string name, decimal cost)
    {
        _name = name;
        _cost = cost;
    }

    public string Name
    {
        get => _name;
        set => _name = value;
    }

    public decimal Price
    {
        get => _cost;
        set => _cost = value;
    }
}

class Program
{
    static void Main(string[] args)
    {
        var item = new SaleItem("Shoes", 19.95m);
        Console.WriteLine($"{item.Name}: sells for {item.Price:C2}");
    }
}
// The example displays output like the following:
//     Shoes: sells for $19.95
```

自动实现的属性

在某些情况下，属性 `get` 和 `set` 访问器仅向支持字段赋值或仅从其中检索值，而不包括任何附加逻辑。通过使用自动实现的属性，既能简化代码，还能让 C# 编译器透明地提供支持字段。

如果属性具有 `get` 和 `set`（或 `get` 和 `init`）访问器，则必须自动实现这两个访问器。自动实现的属性通过以下方式定义：使用 `get` 和 `set` 关键字，但不提供任何实现。下面的示例与上一个示例基本相同，只不过 `Name` 和 `Price` 是自动实现的属性。该示例还删除了参数化构造函数，以便通过调用无参数构造函数和对象初始值设置项立即初始化 `SaleItem` 对象。

```
using System;

public class SaleItem
{
    public string Name
    { get; set; }

    public decimal Price
    { get; set; }
}

class Program
{
    static void Main(string[] args)
    {
        var item = new SaleItem{ Name = "Shoes", Price = 19.95m };
        Console.WriteLine($"{item.Name}: sells for {item.Price:C2}");
    }
}
// The example displays output like the following:
//     Shoes: sells for $19.95
```

相关章节

- [使用属性](#)
- [接口属性](#)
- [属性与索引器之间的比较](#)
- [限制访问器可访问性](#)
- [自动实现的属性](#)

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。

请参阅

- [C# 编程指南](#)
- [使用属性](#)
- [索引器](#)
- [get 关键字](#)
- [set 关键字](#)

使用属性 (C# 编程指南)

2021/5/7 • [Edit Online](#)

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` 访问器。读取属性时，执行 `get` 访问器的代码块；向属性赋予新值时，执行 `set` 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。在 C# 9 及更高版本中，可以使用 `init` 访问器代替 `set` 访问器将属性设为只读。

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。

属性具有许多用途：它们可以先验证数据，再允许进行更改；可以在类上透明地公开数据，其中数据实际是从某个其他源（如数据库）检索到的；可以在数据发生更改时采取措施，例如引发事件或更改其他字段的值。

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：

```
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。属性的数据的实际位置通常被称为属性的“后备存储”。属性使用私有字段作为后备存储，这很常见。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](#)。

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。执行 `get` 访问器等效于读取字段的值。例如，在从 `get` 访问器返回私有变量且已启用优化时，对 `get` 访问器方法的调用由编译器内联，因此不存在方法调用开销。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。以下是一个 `get` 访问器，它返回私有字段 `_name` 的值：

```
class Person
{
    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：

```
Person person = new Person();
//...

System.Console.WriteLine(person.Name); // the get accessor is invoked here
```

`get` 访问器必须以 `return` 或 `throw` 语句结尾，且控件不能超出访问器正文。

使用 `get` 访问器更改对象的状态是一种糟糕的编程风格。例如，以下访问器将产生在每次访问 `_number` 字段时更改对象状态的副作用。

```
private int _number;
public int Number => _number++; // Don't do this
```

`get` 访问器可以用于返回字段值或计算并返回字段值。例如：

```
class Employee
{
    private string _name;
    public string Name => _name != null ? _name : "NA";
}
```

在上一个代码段中，如果不给 `Name` 属性赋值，它将返回值 `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。在下面的示例中，将 `set` 访问器添加到 `Name` 属性：

```
class Person
{
    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：

```
Person person = new Person();
person.Name = "Joe"; // the set accessor is invoked here

System.Console.WriteLine(person.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。这使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。这可使派生类使用 `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`。这意味着类中没有实现，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](#)。

NOTE

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。

```
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:
}

class TestEmployee
{
    static void Main()
    {
        Employee.NumberOfEmployees = 107;
        Employee e1 = new Employee();
        e1.Name = "Claude Vige";

        System.Console.WriteLine("Employee number: {0}", Employee.Counter);
        System.Console.WriteLine("Employee name: {0}", e1.Name);
    }
}
/* Output:
   Employee number: 108
   Employee name: Claude Vige
*/
```

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：

```

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
{
    static void Main()
    {
        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：

```
public new string Name
```

- `((Employee))` 转换用于访问基类中的隐藏属性：

```
((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。

```

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: 5
   Area of the square = 25.00
   Area of the cube = 11.25
   Side of the square = 5.00
   Side of the cube = 3.56
*/

```

```
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

- [C# 编程指南](#)
- [属性](#)
- [接口属性](#)
- [自动实现的属性](#)

接口属性 (C# 编程指南)

2020/11/2 • [Edit Online](#)

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：

```
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。与在类和结构中不同，在没有主体的情况下声明访问器不会声明[自动实现的属性](#)。从 C# 8.0 开始，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，因为接口可能不会定义实例数据字段。

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：

```
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：

```
string IEmployee.Name
{
    get { return "Employee Name"; }
    set { }
}
```

在 `IEmployee` 接口中实现 `Name` 属性，而以下声明：

```
string ICitizen.Name
{
    get { return "Citizen Name"; }
    set { }
}
```

在 `ICitizen` 接口中实现 `Name` 属性。

```

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;
}

```

```

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);

```

210 Hazem Abolrous

示例输出

```

Enter number of employees: 210
Enter the name of the new employee: Hazem Abolrous
The employee information:
Employee number: 211
Employee name: Hazem Abolrous

```

请参阅

- [C# 编程指南](#)
- [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)

2020/11/2 • [Edit Online](#)

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。通常是在保持 `get` 访问器可公开访问的情况下，限制 `set` 访问器的可访问性。例如：

```
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。

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：

```

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：

```

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 `DerivedClass` 中 `Name` 属性上 `set` 访问器的限制性访问修饰符(如 `private` 或 `protected`)如何防止对该访问器的访问, 并在向它赋值时生成错误。

```
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 = "Marv";
```

```
        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

- [C# 编程指南](#)
- [属性](#)
- [索引器](#)
- [访问修饰符](#)

如何声明和使用读/写属性 (C# 编程指南)

2021/5/10 • [Edit Online](#)

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。这通过访问器实现：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

```
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;
    }
}

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。这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。例如，省略 `set` 访问器可使属性为只读：

```

public string Name
{
    get
    {
        return _name;
    }
    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，便可使用这些属性，就像它们是类的字段一样。在获取和设置属性的值时，这允许一种非常自然的语法，如以下语句所示：

```

person.Name = "Joe";
person.Age = 99;

```

请注意，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：

```

_name = value;

```

请注意在 `Person` 对象上递增 `Age` 属性的简洁语法：

```

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：

```
person.SetAge(person.GetAge() + 1);
```

`ToString` 方法在此示例中被重写：

```
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

- [C# 编程指南](#)
- [属性](#)
- [类、结构和记录](#)

自动实现的属性 (C# 编程指南)

2021/5/7 • [Edit Online](#)

在 C# 3.0 及更高版本，当属性访问器中不需要任何其他逻辑时，自动实现的属性会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。在 C# 9 及更高版本中，`init` 访问器也可以声明为自动实现的属性。

示例

下列示例演示一个简单的类，它具有某些自动实现的属性：

```
// This class is mutable. Its data can be modified from
// outside the class.
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;
    }
}
```

不能在接口中声明自动实现的属性。自动实现的属性声明一个私有实例支持字段，并且接口可能不声明实例字段。如果在接口中声明属性而不定义主体，请使用访问器声明属性，访问器必须由实现该接口的每个类型实现。

在 C# 6 和更高版本中，你可以像字段一样初始化自动实现属性：

```
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[如何使用自动实现的属性实现轻量类](#)。

请参阅

- [属性](#)
- [修饰符](#)

如何使用自动实现的属性实现轻量类 (C# 编程指南)

2021/5/7 • [Edit Online](#)

本示例演示如何创建一个仅用于封装一组自动实现的属性的不可变轻型类。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。

当你声明一个 `private set` 取值函数时，你无法使用对象初始值设定项来初始化属性。你必须使用构造函数或工厂方法。

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。

```
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

下面的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。

```
// 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";

        // Keep the console open in debug mode.
        Console.WriteLine("Press any key to exit.");
    }
}
```

```
CONSOLE.WRITELINE( "Press any key to exit. " );
Console.ReadKey();
}

/* 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了支持字段。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)

2021/5/10 • [Edit Online](#)

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。

NOTE

本文讨论命名的方法。有关匿名函数的信息，请参阅[匿名函数](#)。

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。

IMPORTANT

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：

```
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：

```

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：

```

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：

```
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：

```
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` 关键字返回值。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，以 C# 7.0 开头。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变量名 `estDistance`。

```
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：

```
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：

```
int result = obj.AddTwoNumbers(1, 2);
result = obj.SquareANumber(result);
// The result is 9.
Console.WriteLine(result);
```

```
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：

```
ref int distance = plant
```

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：

```
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;
        }
    }
}
```

有关详细信息，请参阅 [return](#)。

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。

NOTE

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。从 C# 7.0 开始, 异步方法可以有[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();`。

```
using System;
using System.Threading.Tasks;

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 `=>` 定义此类方法的语言快捷方式:

```
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。当 `yield return` 语句到达时，将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>`、`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。

请参阅

- [C# 编程指南](#)
- [类、结构和记录](#)
- [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](#)
- [return](#)
- [out](#)
- [ref](#)
- [传递参数](#)

本地函数 (C# 编程指南)

2021/3/9 • [Edit Online](#)

从 C# 7.0 开始, C# 支持本地函数。本地函数是一种嵌套在另一成员中的类型的私有方法。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。

NOTE

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：

```
<modifiers> <return-type> <method-name> <parameter-list>
```

可以将以下修饰符用于本地函数：

- `async`
- `unsafe`
- `static` (在 C# 8.0 和更高版本中)。静态本地函数无法捕获局部变量或实例状态。
- `extern` (在 C# 9.0 和更高版本中)。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。因为所有本地函数都是私有的, 包括访问修饰符(如 `private` 关键字)会生成编译器错误 CS0106“修饰符‘private’对于此项无效”。

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:

```
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 + @"\";
    }
}
```

从 C# 9.0 开始，你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：

```
#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。对于方法迭代器，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。

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

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：

```

using System;
using System.Collections.Generic;

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

乍看之下，本地函数和 [Lambda 表达式](#)非常相似。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：

```

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：

```

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。

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，需要对其进行明确赋值：必须声明要分配给它的 `Action / Func` 变量，并为其分配 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，我们可以在 `return` 语句的上方或下方声明本地函数，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。必须声明 Lambda 表达式，赋给默认值，然后才能将其重新赋给引用相同 Lambda 表达式的主体。

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：

```

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 `return` 语句之前调用了 `LocalFunction`，所以在 `return` 语句中明确分配了 `y`。

请注意，当本地函数捕获封闭范围中的变量时，本地函数将作为委托类型实现。

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：

```
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();
}
```

该 lambda 表达式的闭包包含 `address`、`index` 和 `name` 变量。就本地函数而言，实现闭包的对象可能为 `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 表达式所需的实例化意味着额外的内存分配，后者可能是时间关键代码路径中的性能因素。本地函数不会产生这种开销。在以上示例中，本地函数版本具有的分配比 Lambda 表达式版本少 2 个。

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。请注意，此功能在 C# 8.0 及更高版本中提供。

NOTE

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`。

```
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。

```
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;
        }
    }
}
```

Lambda 表达式中不允许使用 `yield return` 语句，请参阅[编译器错误 CS1621](#)。

虽然本地函数对 lambda 表达式可能有点冗余，但实际上它们的目的和用法都不一样。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。

请参阅

- [方法](#)

ref 返回值和局部变量

2020/5/20 • [Edit Online](#)

从 C# 7.0 开始，C# 支持引用返回值（ref 返回值）。借助引用返回值，方法可以将对变量的引用（而不是值）返回给调用方。然后，调用方可以选择将返回的变量视为按值返回或按引用返回。调用方可以新建称为“引用本地”的变量，其本身就是对返回值的引用。

什么是引用返回值？

绝大多数开发者都很熟悉将参数按引用传递给被调用的方法。已调用方法的参数列表包含以引用方式传递的变量。调用方会监测已调用方法对其值做出的任何更改。引用返回值是指，方法返回对某变量的引用（或别名）。相应变量的作用域必须包括方法。相应变量的生存期必须超过方法的返回值。调用方对方法的返回值进行的修改应用于方法返回的变量。

如果声明方法返回引用返回值，表明方法返回变量别名。这样做通常是为了让调用代码有权通过别名访问此变量（包括修改它）。因此，方法的引用返回值不得包含返回类型 `void`。

对于方法能以引用返回值的形式返回的表达式，存在一些限制。具体限制包括：

- 返回值的生存期必须长于方法执行时间。换言之，它不能是返回自身的方法中的本地变量。它可以是实例或类的静态字段，也可传递给方法的参数。尝试返回局部变量将生成编译器错误 CS8168：“无法按引用返回局部 “obj”，因为它不是 ref 局部变量”。
- 返回值不得为文本 `null`。返回 `null` 会生成编译器错误 CS8156“无法在此上下文中使用表达式，因为它可能不是以引用方式返回”。

使用引用返回值的方法可以返回值当前为 `NULL`（未实例化）或可为空的值类型的变量别名。

- 返回值不得为常量、枚举成员、通过属性的按值返回值或 `class` / `struct` 方法。违反此规则会生成编译器错误 CS8156“无法在此上下文中使用表达式，因为它可能不是以引用方式返回”。

此外，禁止对异步方法使用引用返回值。异步方法可能会在执行尚未完成时就返回值，尽管返回值仍未知。

定义 ref 返回值

返回引用返回值的方法必须满足以下两个条件：

- 方法签名在返回类型前面有 `ref` 关键字。
- 方法主体中的每个 `return` 语句都在返回实例的名称前面有 `ref` 关键字。

下面的示例方法满足这些条件，且返回对名为 `p` 的 `Person` 对象的引用：

```
public ref Person GetContactInformation(string fname, string lname)
{
    // ...method implementation...
    return ref p;
}
```

使用 ref 返回值

引用返回值是被调用方法范围内另一个变量的别名。可以将引用返回值的所有使用都解释为，使用它取别名的变量：

- 分配值时，就是将值分配到它取别名的变量。
- 读取值时，就是读取它取别名的变量的值。
- 如果以引用方式返回它，就是返回对相同变量所取的别名。
- 如果以引用方式将它传递到另一个方法，就是传递对它取别名的变量的引用。
- 如果返回引用本地别名，就是返回相同变量的新别名。

ref 局部变量

假设 `GetContactInformation` 方法声明为引用返回：

```
public ref Person GetContactInformation(string fname, string lname)
```

按值分配会读取变量值，并将它分配给新变量：

```
Person p = contacts.GetContactInformation("Brandie", "Best");
```

上面的分配将 `p` 声明为本地变量。它的初始值是通过读取 `GetContactInformation` 返回的值进行复制。之后对 `p` 的任何分配都不会更改 `GetContactInformation` 返回的变量值。变量 `p` 不再是返回的变量的别名。

声明引用本地变量，复制原始值的别名。在下面的分配中，`p` 是从 `GetContactInformation` 返回的变量的别名。

```
ref Person p = ref contacts.GetContactInformation("Brandie", "Best");
```

后续使用 `p` 等同于使用 `GetContactInformation` 返回的变量，因为 `p` 是此变量的别名。对 `p` 所做的更改也会更改从 `GetContactInformation` 返回的变量。

`ref` 关键字用于局部变量声明前面和方法调用前面。

可通过相同方式按引用访问值。在某些情况下，按引用访问值可避免潜在的高开销复制操作，从而提高性能。例如，以下语句显示用户可如何定义一个用于引用值的 `ref` 局部变量值。

```
ref VeryLargeStruct reflocal = ref veryLargeStruct;
```

`ref` 关键字用于局部变量声明前面和第二个示例中的值前面。在这两个示例中，如果无法同时将 `ref` 关键字包含在变量声明和赋值中，则会导致编译器错误 CS8172：“无法使用值对按引用变量进行初始化”。

在低于 C# 7.3 的版本中，无法将 `ref` 局部变量重新分配为，在初始化后引用其他存储。此限制已取消。下面的示例展示了如何重新分配：

```
ref VeryLargeStruct reflocal = ref veryLargeStruct; // initialization
refLocal = ref anotherVeryLargeStruct; // reassigned, refLocal refers to different storage.
```

Ref 局部变量仍必须在声明时进行初始化。

ref 返回结果和 ref 局部变量：示例

下列示例定义存储整数值数组的 `NumberStore` 类。`FindNumber` 方法按引用返回第一个大于或等于作为参数传递的数字的数字。如果没有大于或等于该参数的数字，则方法返回索引 0 中的数字。

```

using System;

class NumberStore
{
    int[] numbers = { 1, 3, 7, 15, 31, 63, 127, 255, 511, 1023 };

    public ref int FindNumber(int target)
    {
        for (int ctr = 0; ctr < numbers.Length; ctr++)
        {
            if (numbers[ctr] >= target)
                return ref numbers[ctr];
        }
        return ref numbers[0];
    }

    public override string ToString() => string.Join(" ", numbers);
}

```

下列示例调用 `NumberStore.FindNumber` 方法来检索大于或等于 16 的第一个值。然后，调用方将该方法返回的值加倍。示例输出表明，`NumberStore` 实例的数组元素值反映了更改。

```

var store = new NumberStore();
Console.WriteLine($"Original sequence: {store.ToString()}");
int number = 16;
ref var value = ref store.FindNumber(number);
value *= 2;
Console.WriteLine($"New sequence:      {store.ToString()}");
// The example displays the following output:
//      Original sequence: 1 3 7 15 31 63 127 255 511 1023
//      New sequence:      1 3 7 15 62 63 127 255 511 1023

```

如果引用返回值不受支持，需要通过返回数组元素及其值的索引来执行此类操作。然后，调用方可使用此索引修改单个方法调用中的值。但调用方也可修改要访问的索引，还可修改其他数组值。

下面的示例展示了如何在高于 C# 7.3 的版本中将 `FindNumber` 方法重写为使用 `ref` 局部重新分配：

```

using System;

class NumberStore
{
    int[] numbers = { 1, 3, 7, 15, 31, 63, 127, 255, 511, 1023 };

    public ref int FindNumber(int target)
    {
        ref int returnVal = ref numbers[0];
        var ctr = numbers.Length - 1;
        while ((ctr >= 0) && numbers[ctr] >= target)
        {
            returnVal = ref numbers[ctr];
            ctr--;
        }
        return ref returnVal;
    }

    public override string ToString() => string.Join(" ", numbers);
}

```

如果查找的数字更接近数组末尾，这第二个版本就更高效且序列更长。

另请参阅

- `ref` 关键字
- 编写安全高效的代码

传递参数 (C# 编程指南)

2020/11/2 • [Edit Online](#)

在 C# 中，实参可以按值或按引用传递给形参。按引用传递使函数成员、方法、属性、索引器、运算符和构造函数可以更改参数的值，并让该更改在调用环境中保持。若要按引用传递参数，且具有更改值的意向，请使用 `ref` 或 `out` 关键字。若要按引用传递，且只有避免复制而不更改值的意向，请使用 `in` 修饰符。为简单起见，本主题的示例中只使用 `ref` 关键字。有关 `in`、`ref` 和 `out` 之间差异的详细信息，请参阅 [in](#)、[ref](#) 及 [out](#)。

以下示例演示值与引用参数之间的差异。

```
class Program
{
    static void Main(string[] args)
    {
        int arg;

        // Passing by value.
        // The value of arg in Main is not changed.
        arg = 4;
        squareVal(arg);
        Console.WriteLine(arg);
        // Output: 4

        // Passing by reference.
        // The value of arg in Main is changed.
        arg = 4;
        squareRef(ref arg);
        Console.WriteLine(arg);
        // Output: 16
    }

    static void squareVal(int valParameter)
    {
        valParameter *= valParameter;
    }

    // Passing by reference
    static void squareRef(ref int refParameter)
    {
        refParameter *= refParameter;
    }
}
```

有关详细信息，请参阅下列主题：

- [传递值类型参数](#)
- [传递引用类型参数](#)

C# 语言规范

有关详细信息，请参阅 [C# 语言规范](#) 中的[参数列表](#)。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](#)
- [方法](#)

传递值类型参数 (C# 编程指南)

2020/11/2 • [Edit Online](#)

一个值类型变量包含它直接与引用类型变量相对的数据，其中包含对其数据的引用。按值将值-类型变量传递给方法，意味着将变量副本传递给该方法。在方法内发生的对该实参进行的任何更改都不会影响存储在形参变量中的原始数据。若要使用已调用的方法来更改参数值，必须使用 `ref` 或 `out` 关键字通过引用传递它。还可以使用 `in` 关键字来按引用传递值参数，以避免复制并同时保证不更改值。为简单起见，下面的示例使用 `ref`。

按值传递值类型

下面的示例演示按值传递值-类型参数。变量 `n` 按值传递给方法 `SquareIt`。在方法内发生的任何更改都不会影响该变量的原始值。

```
class PassingValByVal
{
    static void SquareIt(int x)
        // The parameter x is passed by value.
        // Changes to x will not affect the original value of n.
    {
        x *= x;
        System.Console.WriteLine("The value inside the method: {0}", x);
    }
    static void Main()
    {
        int n = 5;
        System.Console.WriteLine("The value before calling the method: {0}", n);

        SquareIt(n); // Passing the variable by value.
        System.Console.WriteLine("The value after calling the method: {0}", n);

        // Keep the console window open in debug mode.
        System.Console.WriteLine("Press any key to exit.");
        System.Console.ReadKey();
    }
}
/* Output:
   The value before calling the method: 5
   The value inside the method: 25
   The value after calling the method: 5
*/
```

变量 `n` 是值类型。它包含其数据，值为 `5`。调用 `SquareIt` 时，`n` 的内容复制到参数 `x` 中，这是该方法内的平方值。但是在 `Main` 中，`n` 的值在调用 `SquareIt` 方法之后与之前相同。在方法内发生的更改只影响本地变量 `x`。

按引用传递值类型

下面的示例与上述示例相同，除了自变量是作为 `ref` 参数传递的。`x` 在方法中更改时，基础自变量的值 `n` 也更改。

```

class PassingValByRef
{
    static void SquareIt(ref int x)
    // The parameter x is passed by reference.
    // Changes to x will affect the original value of x.
    {
        x *= x;
        System.Console.WriteLine("The value inside the method: {0}", x);
    }
    static void Main()
    {
        int n = 5;
        System.Console.WriteLine("The value before calling the method: {0}", n);

        SquareIt(ref n); // Passing the variable by reference.
        System.Console.WriteLine("The value after calling the method: {0}", n);

        // Keep the console window open in debug mode.
        System.Console.WriteLine("Press any key to exit.");
        System.Console.ReadKey();
    }
}
/* Output:
   The value before calling the method: 5
   The value inside the method: 25
   The value after calling the method: 25
*/

```

在此示例中，它传递的不是 `n` 的值；而是传递 `n` 的引用。参数 `x` 不是 `int`；它是对 `int` 的引用，在这种情况下，是对 `n` 的引用。因此，当 `x` 在方法中进行平方计算时，实际求平方值的就是 `x` 所指的 `n`。

交换值类型

更改自变量值的一个常见示例是交换方法，其中将两个变量传递给方法，并由该方法交换其内容。必须通过引用将自变量传递给交换方法。否则，交换参数在方法中的本地副本，并且调用方法中不会发生更改。下面的示例交换整数值。

```

static void SwapByRef(ref int x, ref int y)
{
    int temp = x;
    x = y;
    y = temp;
}

```

调用 `SwapByRef` 方法时，在调用中使用 `ref` 关键字，如下面的示例中所示。

```
static void Main()
{
    int i = 2, j = 3;
    System.Console.WriteLine("i = {0}  j = {1}" , i, j);

    SwapByRef (ref i, ref j);

    System.Console.WriteLine("i = {0}  j = {1}" , i, j);

    // Keep the console window open in debug mode.
    System.Console.WriteLine("Press any key to exit.");
    System.Console.ReadKey();
}

/* Output:
   i = 2  j = 3
   i = 3  j = 2
*/
```

请参阅

- [C# 编程指南](#)
- [传递参数](#)
- [传递引用类型参数](#)

传递引用类型参数 (C# 编程指南)

2020/11/2 • [Edit Online](#)

引用类型的变量不直接包含其数据;它包含对其数据的引用。如果按值传递引用类型参数,则可能更改属于所引用对象的数据,例如类成员的值。但是,不能更改引用本身的值;例如,不能使用相同引用为新对象分配内存,并将其保留在方法外部。为此,请使用 `ref` 或 `out` 关键字传递参数。为简单起见,下面的示例使用 `ref`。

按值传递引用类型

以下示例演示将引用类型参数 `arr` 按值传递给方法 `Change`。由于该参数是对 `arr` 的引用,因此可以更改数组元素的值。但是,只有在方法内才能将参数重新分配给其他内存位置,且不会影响原始变量 `arr`。

```
class PassingRefByVal
{
    static void Change(int[] pArray)
    {
        pArray[0] = 888; // This change affects the original element.
        pArray = new int[5] {-3, -1, -2, -3, -4}; // This change is local.
        System.Console.WriteLine("Inside the method, the first element is: {0}", pArray[0]);
    }

    static void Main()
    {
        int[] arr = {1, 4, 5};
        System.Console.WriteLine("Inside Main, before calling the method, the first element is: {0}", arr[0]);

        Change(arr);
        System.Console.WriteLine("Inside Main, after calling the method, the first element is: {0}", arr[0]);
    }
}
/* Output:
Inside Main, before calling the method, the first element is: 1
Inside the method, the first element is: -3
Inside Main, after calling the method, the first element is: 888
*/
```

在前面的示例中,数组 `arr` 属于引用类型,传递给不含 `ref` 参数的方法。在这种情况下,将向该方法传递一个指向 `arr` 的引用副本。输出表明,此方法可以更改数组元素的内容,在本示例中将 `1` 更改为了 `888`。但是,通过在 `Change` 方法内使用 `new` 运算符来分配一份新的内存可使变量 `pArray` 引用新的数组。因此,此后的任意更改都不会影响创建于 `Main` 内的原始数组 `arr`。实际上,本示例创建了两个数组,一个在 `Main` 方法内,另一个在 `Change` 方法内。

按引用传递引用类型

除了 `ref` 关键字添加到方法标头和调用,以下示例与上述示例相同。方法中所作的任何更改都会影响调用程序中的原始变量。

```
class PassingRefByRef
{
    static void Change(ref int[] pArray)
    {
        // Both of the following changes will affect the original variables:
        pArray[0] = 888;
        pArray = new int[5] {-3, -1, -2, -3, -4};
        System.Console.WriteLine("Inside the method, the first element is: {0}", pArray[0]);
    }

    static void Main()
    {
        int[] arr = {1, 4, 5};
        System.Console.WriteLine("Inside Main, before calling the method, the first element is: {0}",
        arr[0]);

        Change(ref arr);
        System.Console.WriteLine("Inside Main, after calling the method, the first element is: {0}",
        arr[0]);
    }
}
/* Output:
Inside Main, before calling the method, the first element is: 1
Inside the method, the first element is: -3
Inside Main, after calling the method, the first element is: -3
*/
```

方法内所作的所有更改都将影响 `Main` 中的原始数组。事实上，将使用 `new` 运算符重新分配原始数组。因此，调用 `Change` 方法后，对 `arr` 的任何引用都将指向 `Change` 方法中创建的五元素数组。

交换两个字符串

按引用传递引用类型参数的一个很好的示例是交换字符串。在示例中，`str1` 和 `str2` 两个字符串在 `Main` 中初始化，然后传递给 `SwapStrings` 方法，作为由 `ref` 关键字修改的参数。这两个字符串在该方法以及 `Main` 内交换。

```
class SwappingStrings
{
    static void SwapStrings(ref string s1, ref string s2)
        // The string parameter is passed by reference.
        // Any changes on parameters will affect the original variables.
    {
        string temp = s1;
        s1 = s2;
        s2 = temp;
        System.Console.WriteLine("Inside the method: {0} {1}", s1, s2);
    }

    static void Main()
    {
        string str1 = "John";
        string str2 = "Smith";
        System.Console.WriteLine("Inside Main, before swapping: {0} {1}", str1, str2);

        SwapStrings(ref str1, ref str2);    // Passing strings by reference
        System.Console.WriteLine("Inside Main, after swapping: {0} {1}", str1, str2);
    }
}
/* Output:
   Inside Main, before swapping: John Smith
   Inside the method: Smith John
   Inside Main, after swapping: Smith John
*/
```

在此示例中，需要按引用传递参数，以影响调用程序中的变量。如果同时从方法标头和方法调用中删除 `ref` 关键字，调用程序中不会发生任何更改。

有关字符串的详细信息，请参阅[字符串](#)。

请参阅

- [C# 编程指南](#)
- [传递参数](#)
- [ref](#)
- [in](#)
- [out](#)
- [引用类型](#)

如何了解向方法传递结构和向方法传递类引用之间的区别 (C# 编程指南)

2021/3/5 • [Edit Online](#)

下面的示例演示向方法传递结构和向方法传递类实例之间的区别。在此示例中，这两个参数(结构和类实例)都按值传递，并且两个方法都更改了参数的一个字段的值。但是，由于传递结构和传递类实例时所传递的内容不同，所以这两个方法的结果不同。

因为结构是值类型，所以按值将结构传递给方法时，该方法接收结构参数的副本并在其上运行。该方法无法访问调用方法中的原始结构，因此无法对其进行任何更改。它只能更改副本。

类实例是引用类型，不是值类型。按值将引用类型传递给方法时，方法接收对类实例的引用的副本。也就是说，被调用的方法接收实例的地址副本，而调用方法保留实例的原始地址。调用方法中的类实例具有地址，所调用的方法中的参数具有地址副本，且这两个地址引用同一个对象。由于参数只包含地址副本，因此所调用的方法不能更改调用方法中类实例的地址。但是，被调用的方法可以使用该地址的副本访问原始地址和地址副本引用的类成员。如果所调用的方法更改了类成员，则调用方法中的原始类实例也会更改。

以下示例输出对差异进行了说明。调用 `ClassTaker` 方法更改了类实例的 `willIChange` 字段的值，因为该方法使用参数中的地址来查找类实例的指定字段。调用 `StructTaker` 方法不会更改调用方法中结构的 `willIChange` 字段，因为该参数的值是结构本身的副本，而不是其地址的副本。`StructTaker` 更改副本，对 `StructTaker` 的调用完成时，副本丢失。

示例

```

using System;

class TheClass
{
    public string willIChange;
}

struct TheStruct
{
    public string willIChange;
}

class TestClassAndStruct
{
    static void ClassTaker(TheClass c)
    {
        c.willIChange = "Changed";
    }

    static void StructTaker(TheStruct s)
    {
        s.willIChange = "Changed";
    }

    static void Main()
    {
        TheClass testClass = new TheClass();
        TheStruct testStruct = new TheStruct();

        testClass.willIChange = "Not Changed";
        testStruct.willIChange = "Not Changed";

        ClassTaker(testClass);
        StructTaker(testStruct);

        Console.WriteLine("Class field = {0}", testClass.willIChange);
        Console.WriteLine("Struct field = {0}", testStruct.willIChange);

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
/* Output:
   Class field = Changed
   Struct field = Not Changed
*/

```

请参阅

- [C# 编程指南](#)
- [类](#)
- [结构类型](#)
- [传递参数](#)

隐式类型本地变量 (C# 编程指南)

2020/11/2 • [Edit Online](#)

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：

```
// 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。

```
for (var x = 1; x < 10; x++)
```

- 在 `foreach` 初始化语句中。

```
foreach (var item in list) {...}
```

- 在 `using` 域间中。

```
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。

```
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：

```
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：

```
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。

```
// 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\(语言集成查询\)](#)
- [for](#)
- [foreach, in](#)
- [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)

2021/3/5 • [Edit Online](#)

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](#)。

```
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);
    }
}
```

示例

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。

```
// 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);
}
```

请参阅

- [C# 编程指南](#)
- [扩展方法](#)
- [LINQ\(语言集成查询\)](#)
- [var](#)
- [C# 中的 LINQ](#)

扩展方法 (C# 编程指南)

2020/11/2 • [Edit Online](#)

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](#)。

```
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。参数前面是 `此` 修饰符。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：

```
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：

```
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：

```
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。

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：

```
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。下面的示例演示编译器如何确定要绑定到哪个扩展方法或实例方法。

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）。

```
// Define an interface named IMyInterface.
namespace DefineIMyInterface
{
    using System;

    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 ExtensionMethods
    {
        public static void MethodA(IMyInterface target)
        {
            target.MethodB();
        }
    }
}
```

```

// 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。

```
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。一些其他示例可能是向 `System.String` 类添加常见功能、扩展 `System.IO.File`、`System.IO.Stream` 以及 `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。从 C# 7.2 开始，可以将 `ref` 修饰符添加到扩展方法的第一个参数。添加 `ref` 修饰符意味着第一个参数是通过引用传递的。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。

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](#)。

请参阅

- [C# 编程指南](#)
- [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)

如何实现和调用自定义扩展方法 (C# 编程指南)

2021/3/5 • [Edit Online](#)

本主题将介绍如何为任意 .NET 类型实现自定义扩展方法。客户端代码可以通过以下方法使用扩展方法，添加包含这些扩展方法的 DLL 的引用，以及添加 `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。

此类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. 和调用类型的实例方法那样调用这些方法。

请注意，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。

示例

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。

```
using System.Linq;
using System.Text;
using System;

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;
        }
    }
}

namespace Extension_Methods_Simple
{
    // Import the extension method namespace.
    using CustomExtensions;
    class Program
    {
        static void Main(string[] args)
        {
            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);
        }
    }
}
```

.NET 安全性

扩展方法不存在特定的安全漏洞。始终不会将扩展方法用于模拟类型的现有方法，因为为了支持类型本身定义的实例或静态方法，已解决所有名称冲突。扩展方法无法访问扩展类中的任何隐私数据。

请参阅

- [C# 编程指南](#)
- [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)

2021/3/5 • [Edit Online](#)

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。

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Linq;

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。

请参阅

- [C# 编程指南](#)
- [扩展方法](#)

命名实参和可选实参 (C# 编程指南)

2021/3/5 • [Edit Online](#)

C# 4 介绍命名实参和可选实参。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。

命名实参

有了命名实参，你将不再匹配形参在所调用方法的形参列表中的顺序。每个实参的形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。

```
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。

```
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

```
PrintOrderDetails("Gift Shop", 31, productName: "Red Mug");
```

- 以 C# 7.2 开头，则它们就有效并用在正确位置。在以下示例中，形参 `orderNum` 位于正确的位罝，但未显式命名。

```
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。

```
// 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。

```
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");      // C# 7.2 onwards
        PrintOrderDetails("Gift Shop", orderNum: 31, "Red Mug");                      // C# 7.2 onwards

        // 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}");
    }
}
```

可选实参

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`。

```
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。

```
//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。

```
anExample.ExampleMethod(3, optionalint: 4);
```

IntelliSense 使用括号表示可选形参，如下图所示：

```
anExample.ExampleMethod(  
    void ExampleClass.ExampleMethod(int required,  
        [string optionalstr = "default string"],  
        [int optionalint = 10])
```

NOTE

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。

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。

```
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。

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])
)

```

在 C# 3.0 以及早期版本中，每个形参都需要一个实参，如下例所示。

```

// In C# 3.0 and earlier versions, you need to supply an argument for
// every parameter. The following call specifies a value for the first
// parameter, and sends a placeholder value for the other six. The
// default values are used for those parameters.
var excelApp = new Microsoft.Office.Interop.Excel.Application();
excelApp.Workbooks.Add();
excelApp.Visible = true;

var myFormat =
    Microsoft.Office.Interop.Excel.XlRangeAutoFormat.xlRangeAutoFormatAccounting1;

excelApp.get_Range("A1", "B4").AutoFormat(myFormat, Type.Missing,
    Type.Missing, Type.Missing, Type.Missing, Type.Missing);

```

但是，可以通过使用 C# 4.0 中引入的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。

```
// The following code shows the same call to AutoFormat in C# 4.0. Only  
// the argument for which you want to provide a specific value is listed.  
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。

如何在 Office 编程中使用命名实参和可选实参 (C# 编程指南)

2021/3/5 • [Edit Online](#)

在 C# 4 中引入的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。

在下面的示例中，方法 `ConvertToTable` 具有十六个参数，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十六个参数指定特定值，因此所有这些参数都是可选的。但是，如果没有命名实参和可选实参，则必须为每个形参提供值或占位符值。有了命名实参和可选实参，则只需为项目所需的形参指定值。

必须在计算机上安装 Microsoft Office Word 才能完成这些过程。

NOTE

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

1. 启动 Visual Studio。
2. 在“文件”菜单上，指向“新建”，然后单击“项目”。
3. 在“模板类别”窗格中，展开“Visual C#”，然后单击“Windows”。
4. 查看“模板”窗格的顶部，确保“.NET Framework 4”出现在“目标框架”框中。
5. 在“模板”窗格中，单击“控制台应用程序”。
6. 在“名称”字段中键入项目的名称。
7. 单击“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，然后单击“添加引用”。此时会显示“添加引用”对话框。
2. 在“.NET”页上的“组件名称”列表中，选择“Microsoft.Office.Interop.Word”。
3. 单击“确定”。

添加必要的 using 指令

1. 在“解决方案资源管理器”中，右键单击“Program.cs”文件，然后单击“查看代码”。
2. 将以下 `using` 指令添加到代码文件的顶部：

```
using Word = Microsoft.Office.Interop.Word;
```

在 Word 文档中显示文本

1. 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。

```
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();
}
```

2. 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：

```
// 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. . .");
```

要运行应用程序

1. 将以下语句添加到 Main：

```
DisplayInWord();
```

2. 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

1. 使用 `ConvertToTable` 方法将文本放入表中。该方法具有十六个可选参数。IntelliSense 将可选参数放入括号中，如下图所示。

```
range.ConvertToTable()

Word.Table Range.ConvertToTable([ref object Separator = Type.Missing], [ref object NumRows =
Type.Missing], [ref object NumColumns = Type.Missing], [ref object InitialColumnWidth =
Type.Missing], [ref object Format = Type.Missing], [ref object ApplyBorders = Type.Missing],
[ref object ApplyShading = Type.Missing], [ref object ApplyFont = Type.Missing], [ref object
ApplyColor = Type.Missing], [ref object ApplyHeadingsRows = Type.Missing], [ref object
ApplyLastRow = Type.Missing], [ref object ApplyFirstColumn = Type.Missing], [ref object
ApplyLastColumn = Type.Missing], [ref object AutoFit = Type.Missing], [ref object
AutoFitBehavior = Type.Missing], [ref object DefaultTableBehavior = Type.Missing])
```

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` 的末尾以创建一个简单的表。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。

```
// Convert to a simple table. The table will have a single row with  
// three columns.  
range.ConvertToTable(Separator: ",");
```

在 C# 早期版本中，对 `ConvertToTable` 的调用需要对每个形参使用引用实参，如以下代码所示：

```
// Call to ConvertToTable in Visual C# 2008 or earlier. This code  
// is not part of the solution.  
var missing = Type.Missing;  
object separator = ",";  
range.ConvertToTable(ref separator, ref missing, ref missing,  
    ref missing, ref missing, ref missing,  
    ref missing, ref missing, ref missing,  
    ref missing);
```

2. 按 **Ctrl+F5** 运行项目。

试验其他参数

1. 若要更改表以使其具有一列三行，请将 `DisplayInWord` 中的最后一行替换为以下语句，然后按 **CTRL+F5**。

```
range.ConvertToTable(Separator: ",", AutoFit: true, NumColumns: 1);
```

2. 若要为表指定预定义的格式，请将 `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。

```
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：

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
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);
        }
    }
}

```

请参阅

- [命名参数和可选参数](#)

构造函数 (C# 编程指南)

2021/5/10 • [Edit Online](#)

每当创建类或结构时，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[使用构造函数和实例构造函数](#)。

无参数构造函数

如果没有为类提供构造函数，则 C# 将默认创建一个构造函数，该函数会实例化对象并将成员变量设置为默认值，如[C# 类型的默认值](#)中所列。如果没有为结构提供构造函数，C# 将依赖于隐式无参数构造函数，自动将每个字段初始化为其默认值。有关详细信息和示例，请参阅[实例构造函数](#)。

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。

```
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。

```
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。

```
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。

```
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](#)
- [类、结构和记录](#)
- [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)

2021/5/10 • [Edit Online](#)

创建类或结构时，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。

```
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。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：

```
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与类的构造函数类似，但是 `structs` 不包含显式无参数构造函数，因为编译器将自动提供一个显式无参数构造函数。此构造函数会将 `struct` 中的每个字段初始化为默认值。但是，只有使用 `new` 实例化 `struct` 时，才会调用此无参数构造函数。例如，此代码使用 `Int32` 的无参数构造函数，因此可确保整数已初始化：

```
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：

```
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：

```
int a = 44; // Initialize the value type...
int b;
b = 33; // Or assign it before using it.
Console.WriteLine("{0}, {1}", a, b);
```

因此无需调用值类型的无参数构造函数。

两个类和 `structs` 都可以定义带参数的构造函数。必须通过 `new` 语句或 `base` 语句调用带参数的构造函数。类和 `structs` 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：

```
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：

```
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：

```
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）。这意味着下面的构造函数声明等效：

```
public Manager(int initData)
{
    //Add further instructions here.
}
```

```
public Manager(int initData)
: 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：

```
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：

```
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，它们通常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。

请参阅

- [C# 编程指南](#)
- [类、结构和记录](#)
- [构造函数](#)
- [终结器](#)

实例构造函数 (C# 编程指南)

2021/5/10 • [Edit Online](#)

使用 `new` 表达式创建类的对象时，实例构造函数可用于创建和初始化任意实例成员变量。若要初始化静态类或非静态类中的静态变量，请定义静态构造函数。有关详细信息，请参阅[静态构造函数](#)。

下面的示例演示了实例构造函数：

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

    // constructor
    public Coords()
    {
        x = 0;
        y = 0;
    }
}
```

NOTE

为清楚起见，此类包含公共字段。建议在编程时不要使用公共字段，因为这种做法会使程序中任何位置的任何方法都可以不受限制、不经验证地访问对象的内部组件。数据成员通常应当为私有的，并且只应通过类方法和属性来访问。

只要创建基于 `Coords` 类的对象，就会调用此实例构造函数。诸如此类不带参数的构造函数称为“无参数构造函数”。然而，提供其他构造函数通常十分有用。例如，可以将构造函数添加到 `Coords` 类，以便可以为数据成员指定初始值：

```
// A constructor with two arguments.
public Coords(int x, int y)
{
    this.x = x;
    this.y = y;
}
```

这样便可以用默认或特定的初始值创建 `Coords` 对象，如下所示：

```
var p1 = new Coords();
var p2 = new Coords(5, 3);
```

如果某个类没有构造函数，则会自动生成一个无参数构造函数，并使用默认值来初始化对象字段。例如，`int` 初始化为 0。有关类型默认值的信息，请参阅[C# 类型的默认值](#)。由于 `Coords` 类的无参数构造函数将所有数据成员都初始化为零，因此可以将它完全移除，而不会更改类的工作方式。本主题稍后部分的示例 1 中提供了使用多个构造函数的完整示例，示例 2 中提供了自动生成的构造函数的示例。

也可以用实例构造函数来调用基类的实例构造函数。类构造函数可通过初始值设定项来调用基类的构造函数，如下所示：

```

class Circle : Shape
{
    public Circle(double radius)
        : base(radius, 0)
    {
    }
}

```

在此示例中, `Circle` 类将半径和高度的值传递给 `Shape` (`Circle` 从它派生而来) 提供的构造函数。使用 `Shape` 和 `Circle` 的完整示例完整示例请见本主题中的示例 3。

示例 1

下面的示例说明包含两个类构造函数的类:一个类构造函数不带参数,另一个带有两个参数。

```

class Coords
{
    public int x, y;

    // Default constructor.
    public Coords()
    {
        x = 0;
        y = 0;
    }

    // A constructor with two arguments.
    public Coords(int x, int y)
    {
        this.x = x;
        this.y = y;
    }

    // Override the ToString method.
    public override string ToString()
    {
        return $"({x},{y})";
    }
}

class MainClass
{
    static void Main()
    {
        var p1 = new Coords();
        var p2 = new Coords(5, 3);

        // Display the results using the overriden ToString method.
        Console.WriteLine($"Coords #1 at {p1}");
        Console.WriteLine($"Coords #2 at {p2}");
        Console.ReadKey();
    }
}
/* Output:
Coords #1 at (0,0)
Coords #2 at (5,3)
*/

```

示例 2

在此示例中, `Person` 类没有任何构造函数;在这种情况下,将自动提供无参数构造函数,同时将字段初始化为它们的默认值。

```
using System;

public class Person
{
    public int age;
    public string name;
}

class TestPerson
{
    static void Main()
    {
        var person = new Person();

        Console.WriteLine($"Name: {person.name}, Age: {person.age}");
        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
// Output: Name: , Age: 0
```

请注意，`age` 的默认值为 `0`，`name` 的默认值为 `null`。

示例 3

下面的示例说明使用基类初始值设定项。`Circle` 类派生自常规类 `Shape`，`Cylinder` 类派生自 `circle` 类。每个派生类的构造函数都使用其基类的初始值设定项。

```

using System;

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()
    {
        return pi * x * x;
    }
}

class Cylinder : Circle
{
    public Cylinder(double radius, double height)
        : base(radius)
    {
        y = height;
    }

    public override double Area()
    {
        return (2 * base.Area()) + (2 * pi * x * y);
    }
}

class TestShapes
{
    static void Main()
    {
        double radius = 2.5;
        double height = 3.0;

        Circle ring = new Circle(radius);
        Cylinder tube = new Cylinder(radius, height);

        Console.WriteLine("Area of the circle = {0:F2}", ring.Area());
        Console.WriteLine("Area of the cylinder = {0:F2}", tube.Area());

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
/* Output:
   Area of the circle = 19.63
   Area of the cylinder = 86.39
*/

```

有关调用基类构造函数的更多示例，请参阅 [virtual](#)、[override](#) 和 [base](#)。

请参阅

- [C# 编程指南](#)
- [类、结构和记录](#)
- [构造函数](#)
- [终结器](#)
- [static](#)

私有构造函数 (C# 编程指南)

2021/5/10 • [Edit Online](#)

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：

```
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。

```
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：

```
// Counter aCounter = new Counter(); // Error
```

C# 语言规范

有关详细信息，请参阅 [C# 语言规范](#) 中的[私有构造函数](#)。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](#)
- [类、结构和记录](#)
- [构造函数](#)
- [终结器](#)
- [private](#)
- [public](#)

静态构造函数 (C# 编程指南)

2021/5/10 • [Edit Online](#)

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。将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。

```
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;
    }
}
```

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，并且仅应由公共语言运行时 (CLR) 调用。可以自动调用它们。
- 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。将在创建第一个实例或引用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。调用(而不是分配)分配给事件或委托的静态方法时，将调用类型的静态构造函数。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执行。初始值设定项紧接着执行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，运行时将不会再次调用该函数，并且类型在应用程序域的生存期内将保持未初始化。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` 的字段可能仅被分配为其声明的一部分或在静态构造函数中。如果不显式静态构造函数，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其他锁定机制。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。

NOTE

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。

```
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.ToString());
    }

    // 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.ToString());
    }
}

class TestBus
{
    static void Main()
    {
        Bus bus1 = new Bus(1);
        Bus bus2 = new Bus(2);
    }
}
```

```
// 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。

请参阅

- [C# 编程指南](#)
- [类、结构和记录](#)
- [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](#)

如何编写复制构造函数 (C# 编程指南)

2021/5/10 • [Edit Online](#)

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。

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。

```
using System;

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
```

- [ICloneable](#)
- [记录](#)
- [C# 编程指南](#)
- [类、结构和记录](#)
- [构造函数](#)
- [终结器](#)

终结器 (C# 编程指南)

2021/3/5 • [Edit Online](#)

终结器(也称为析构函数)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。

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。

```
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。

```
using System;

public class Destroyer
{
    public override string ToString() => GetType().Name;

    ~Destroyer() => Console.WriteLine($"The {ToString()} destructor 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：

```
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。

NOTE

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。调用终结器时，会调用垃圾回收器来处理该队列。空终结器只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。

在 .NET Framework 应用程序中（但不在 .NET Core 应用程序中），程序退出时也会调用终结器。

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。

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](#)
- [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。程序运行时，请注意，将按顺序（从派生程度最高到派生程度最低）自动调用这三个类的终结器。

```
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.");
    }
}

class TestDestructors
{
    static void Main()
    {
        Third t = new Third();
    }
}
/* Output (to VS Output Window):
   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 [析构函数](#) 部分。

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)

2020/11/2 • [Edit Online](#)

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。请注意，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](#)。

```
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;
    }
}
```

```
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。

从 C# 6 开始，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：

```
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：

```

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，则这些都是有效的：

```

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：

```

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。

```

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。如果你只想存储某个序列中每个对象的部分信息，则这很有用。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。

```

var productInfos =
    from p in products
    select new { p.ProductName, p.UnitPrice };

```

执行此查询时，`productInfos` 变量将包含一系列对象，这些对象可以在 `foreach` 语句中进行访问，如下面的示例所示：

```

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`。

```
select new {p.ProductName, Price = p.UnitPrice};
```

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简单的值、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：

```
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。请注意，各个对象初始值设定项分别括在大括号中且用逗号隔开。

```
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。

```
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。

```
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。在 C# 6 之前，可以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：

```
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。

示例

下例结合了对象和集合初始值设定项的概念。

```
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.
```

下面的示例展示了实现 `IEnumerable` 且包含具有多个参数的 `Add` 方法的一个对象，它使用在列表中每项具有多个元素的集合初始值设定项，这些元素对应于 `Add` 方法的签名。

```
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。

```
public class DictionaryExample
{
    class RudimentaryMultiValuedDictionary<TKey, TValue> : IEnumerable<KeyValuePair<TKey, List<TValue>>>
    {
        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
*/
}

```

请参阅

- [C# 编程指南](#)
- [C# 中的 LINQ](#)
- [匿名类型](#)

如何使用对象初始值设定项初始化对象 (C# 编程指南)

2021/3/5 • [Edit Online](#)

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：

```
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：

```
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"]);
    }
}
```

请参阅

- [C# 编程指南](#)
- [对象和集合初始值设定项](#)

如何使用集合初始值设定项初始化字典 (C# 编程指南)

2021/3/5 • [Edit Online](#)

`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。

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：

```
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。最内层的大括号中的是 `StudentName` 的对象初始值设定项，最外层的大括号中的是要添加到 `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。

请参阅

- [C# 编程指南](#)
- [对象和集合初始值设定项](#)

嵌套类型 (C# 编程指南)

2021/5/10 • [Edit Online](#)

在类、构造或接口中定义的类型称为嵌套类型。例如

```
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`：

```
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：

```
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：

```
Container.Nested nest = new Container.Nested();
```

请参阅

- [C# 编程指南](#)
- [类、结构和记录](#)
- [访问修饰符](#)
- [构造函数](#)
- [CA1034 规则](#)

分部类和方法 (C# 编程指南)

2021/5/7 • [Edit Online](#)

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：

- 处理大型项目时，使一个类分布于多个独立文件中可以让多位程序员同时对该类进行处理。
- 当使用自动生成的源文件时，你可以添加代码而不需要重新创建源文件。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。
- 若要拆分类定义，请使用 `partial` 关键字修饰符，如下所示：

```
public partial class Employee
{
    public void DoWork()
    {
    }
}

public partial class Employee
{
    public void GoToLunch()
    {
    }
}
```

`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。

NOTE

`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。

```
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：

```
[SerializableAttribute]
partial class Moon { }

[ObsoleteAttribute]
partial class Moon { }
```

它们等效于以下声明：

```
[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
- 接口
- 泛型类型参数属性
- class 特性
- 成员

以下面的声明为例：

```
partial class Earth : Planet, IRotate { }
partial class Earth : IRevolve { }
```

它们等效于以下声明：

```
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：

```
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：

```
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，则该关键字不能与在同一类型的其他分部定义中指定的关键字冲突：

- `public`
- `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 1

描述

下面的示例在一个分部类定义中声明 `Coords` 类的字段和构造函数，在另一个分部类定义中声明成员 `PrintCoords`。

代码

```

public partial class Coords
{
    private int x;
    private int y;

    public Coords(int x, int y)
    {
        this.x = x;
        this.y = y;
    }
}

public partial class Coords
{
    public void PrintCoords()
    {
        Console.WriteLine("Coords: {0},{1}", x, y);
    }
}

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

```

示例 2

描述

从下面的示例可以看出，你也可以开发分部结构和接口。

代码

```

partial interface ITest
{
    void Interface_Test();
}

partial interface ITest
{
    void Interface_Test2();
}

partial struct S1
{
    void Struct_Test() { }
}

partial struct S1
{
    void Struct_Test2() { }
}

```

分部方法

分部类或结构可以包含分部方法。类的一个部分包含方法的签名。可以在同一部分或另一部分中定义实现。如果未提供该实现，则会在编译时删除方法以及对方法的所有调用。根据方法签名，可能需要实现。

分部方法使类的某个部分的实施者能够定义方法(类似于事件)。类的另一部分的实施者可以决定是否实现该方法。如果未实现该方法，编译器会删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。调用但不实现该方法不会导致编译时错误或运行时错误。

在自定义生成的代码时，分部方法特别有用。这些方法允许保留方法名称和签名，因此生成的代码可以调用方法，而开发人员可以决定是否实现方法。与分部类非常类似，分部方法使代码生成器创建的代码和开发人员创建的代码能够协同工作，而不会产生运行时开销。

分部方法声明由两个部分组成：定义和实现。它们可以位于分部类的不同部分中，也可以位于同一部分中。如果不存在实现声明，则编译器会优化定义声明和对方法的所有调用。

```
// Definition in file1.cs
partial void OnNameChanged();

// Implementation in file2.cs
partial void OnNameChanged()
{
    // method body
}
```

- 分部方法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方法签名必须匹配。
- 分部方法可以有 [static](#) 和 [unsafe](#) 修饰符。
- 分部方法可以是泛型的。约束将放在定义分部方法声明上，但也可以选择重复放在实现声明上。参数和类型参数名称在实现声明和定义声明中不必相同。
- 你可以为已定义并实现的分部方法生成[委托](#)，但不能为已经定义但未实现的分部方法生成委托。

在以下情况下，不需要使用分部方法即可实现：

- 没有任何可访问性修饰符(包括默认的[专用](#))。
- 返回 [void](#)。
- 没有任何[输出](#)参数。
- 没有以下任何修饰符：[virtual](#)、[override](#)、[sealed](#)、[new](#) 或 [extern](#)。

任何不符合所有这些限制的方法(例如 `public virtual partial void` 方法)都必须提供实现。

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](#)。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](#)
- [类](#)
- [结构类型](#)
- [接口](#)
- [分部\(类型\)](#)

匿名类型 (C# 编程指南)

2020/11/2 • [Edit Online](#)

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。

可通过使用 `new` 运算符和对象初始值创建匿名类型。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。

```
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。变量 `Product``products` 是对象的集合。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。这将导致在查询中返回较少数量的数据。

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。必须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。

```
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);
}
```

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。

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

备注

匿名类型是直接从[对象](#)派生的[类](#)类型，并且其无法强制转换为除[对象](#)外的任意类型。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。

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，可将参数作为类型对象进行声明。但是，这样做会使强类型化作用无效。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。

请参阅

- [C# 编程指南](#)
- [对象和集合初始值设定项](#)
- [C# 中的 LINQ 入门](#)
- [C# 中的 LINQ](#)

如何在查询中返回元素属性的子集（C# 编程指南）

2021/3/5 • [Edit Online](#)

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：

```
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。

```
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：

```
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：

```
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。

请参阅

- [C# 编程指南](#)
- [匿名类型](#)
- [C# 中的 LINQ](#)

接口 (C# 编程指南)

2021/5/10 • [Edit Online](#)

接口包含非抽象类或结构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。从 C# 8.0 开始，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。

```
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。有关示例的链接，请参阅 [相关部分](#)。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。

当类或结构实现接口时，类或结构必须为该接口声明的所有成员提供实现，但不提供默认实现。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。

```
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 `set` 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。接口中定义的实现是虚拟的，实现类可能会替代该实现。

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。

相关部分

- [接口属性](#)
- [接口中的索引器](#)
- [如何实现接口事件](#)
- [类、结构和记录](#)
- [继承](#)
- [接口](#)
- [方法](#)
- [多态性](#)
- [抽象类、密封类及类成员](#)
- [属性](#)
- [事件](#)
- [索引器](#)

请参阅

- [C# 编程指南](#)
- [继承](#)
- [标识符名称](#)

显式接口实现 (C# 编程指南)

2021/5/10 • [Edit Online](#)

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：

```
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：

```
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：

```
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：

```
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：

```
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 \(C# 参考\)](#)。

从 [C# 8.0](#) 开始，你可以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：

```
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：

```
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

- [C# 编程指南](#)
- [类、结构和记录](#)
- [接口](#)
- [继承](#)

如何显式实现接口成员 (C# 编程指南)

2021/5/10 • [Edit Online](#)

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

```
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：

```
//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：

```
System.Console.WriteLine("Length: {0}", dimensions.GetLength());
System.Console.WriteLine("Width: {0}", dimensions.GetWidth());
```

另请参阅

- [C# 编程指南](#)
- [类、结构和记录](#)
- [接口](#)
- [如何显式实现两个接口的成员](#)

如何显式实现两个接口的成员 (C# 编程指南)

2021/5/10 • [Edit Online](#)

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

```

// 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
*/

```

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：

```
// 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：

```
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

- [C# 编程指南](#)
- [类、结构和记录](#)
- [接口](#)
- [如何显式实现接口成员](#)

委托 (C# 编程指南)

2021/3/5 • [Edit Online](#)

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：

```
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。

NOTE

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 :掌握 C# 3.0 基础知识](#)中的**委托和事件**

请参阅

- [Delegate](#)
- [C# 编程指南](#)
- [事件](#)

使用委托 (C# 编程指南)

2020/11/2 • [Edit Online](#)

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 `Del` 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：

```
public delegate void Del(string message);
```

委托对象通常通过提供委托将封装的方法的名称或使用匿名函数构造。对委托进行实例化后，委托会将对其进行的方法调用传递到该方法。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。这被称为调用委托。实例化的委托可以按封装的方法本身进行调用。例如：

```
// Create a method for a delegate.
public static void DelegateMethod(string message)
{
    Console.WriteLine(message);
}
```

```
// Instantiate the delegate.
Del 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，因此可以作为参数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：

```
public static void MethodWithCallback(int param1, int param2, Del 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：

```
MethodWithCallback(1, 2, handler);
```

并将在输出接收到控制台：

```
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：

```
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：

```
var obj = new MethodClass();
Del d1 = obj.Method1;
Del d2 = obj.Method2;
Del d3 = DelegateMethod;

//Both types of assignment are valid.
Del 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：

```
//remove Method1
allMethodsDelegate -= d1;

// copy AllMethodsDelegate while removing d2
Del 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：

```
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：

```
delegate void Delegate1();
delegate void Delegate2();

static void method(Delegate1 d, Delegate2 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](#)

带有命名方法的委托与匿名方法 (C# 编程指南)

2020/11/2 • [Edit Online](#)

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：

```
// Declare a delegate.  
delegate void Del(int x);  
  
// Define a named method.  
void DoWork(int k) { /* ... */ }  
  
// Instantiate the delegate using the method as a parameter.  
Del 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。有关详细信息，请参阅[匿名函数](#)。

备注

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。

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。

示例 1

以下是声明和使用委托的简单示例。请注意，委托 `Del` 与关联的方法 `MultiplyNumbers` 具有相同的签名

```
// Declare a delegate
delegate void Del(int i, double j);

class MathClass
{
    static void Main()
    {
        MathClass m = new MathClass();

        // Delegate instantiation using "MultiplyNumbers"
        Del 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
*/
```

示例 2

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。

```
// Declare a delegate
delegate void Del();

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:
        Del 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

- [C# 编程指南](#)
- [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）

2021/5/7 • [Edit Online](#)

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

```

using System;

// Define a custom delegate that has a string parameter and returns void.
delegate void CustomDel(string s);

class TestClass
{
    // Define two methods that have the same signature as CustomDel.
    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.
        CustomDel 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;

        // Create the delegate object hiDel that references the
        // method Hello.
        hiDel = Hello;

        // Create 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](#)
- [C# 编程指南](#)
- [事件](#)

如何声明、实例化和使用委托 (C# 编程指南)

2021/5/7 • [Edit Online](#)

在 C# 1.0 和更高版本中，可以如下面的示例所示声明委托。

```
// 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}");  
}
```

```
// Create an instance of the delegate.  
Del del1 = new Del(Notify);
```

C# 2.0 提供了更简单的方法来编写前面的声明，如下面的示例所示。

```
// C# 2.0 provides a simpler way to declare an instance of Del.  
Del del2 = Notify;
```

在 C# 2.0 和更高版本中，还可以使用匿名方法来声明和初始化委托，如下面的示例所示。

```
// Instantiate Del by using an anonymous method.  
Del del3 = delegate(string name)  
{ Console.WriteLine($"Notification received for: {name}"); };
```

在 C# 3.0 和更高版本中，还可以通过使用 lambda 表达式声明和实例化委托，如下面的示例所示。

```
// 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

```
// 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 PriceToller
    {
        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。

```
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：

```
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`：

```
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：

```
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](#)

数组 (C# 编程指南)

2021/5/7 • [Edit Online](#)

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

```
type[] arrayName;
```

示例

下面的示例创建一维数组、多维数组和交错数组：

```
class TestArraysClass
{
    static void Main()
    {
        // Declare a single-dimensional array of 5 integers.
        int[] array1 = new int[5];

        // Declare and set array element values.
        int[] array2 = new int[] { 1, 3, 5, 7, 9 };

        // Alternative syntax.
        int[] array3 = { 1, 2, 3, 4, 5, 6 };

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

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

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

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

数组概述

数组具有以下属性：

- 数组可以是一维、多维或交错的。
- 创建数组实例时，将建立纬度数量和每个纬度的长度。这些值在实例的生存期内无法更改。
- 数值数组元素的默认值设置为零，而引用元素设置为 `null`。
- 交错数组是数组的数组，因此其元素为引用类型且被初始化为 `null`。
- 数组从零开始编制索引：包含 `n` 元素的数组从 `0` 索引到 `n-1`。
- 数组元素可以是任何类型，其中包括数组类型。
- 数组类型是从抽象的基类型 `Array` 派生的引用类型。所有数组都会实现 `IList` 和 `IEnumerable`。可在 C# 中使用 `foreach` 迭代数组。原因是单维数组还实现了 `IList<T>` 和 `IEnumerable<T>`。

作为对象的数组

在 C# 中，数组实际上是对象，而不只是如在 C 和 C++ 中的连续内存的可寻址区域。[Array](#) 是所有数组类型的抽象基类型。可以使用 [Array](#) 具有的属性和其他类成员。例如，使用 [Length](#) 属性来获取数组的长度。以下代码可将 `numbers` 数组的长度 `5` 分配给名为 `lengthOfNumbers` 的变量：

```
int[] numbers = { 1, 2, 3, 4, 5 };
int lengthOfNumbers = numbers.Length;
```

[Array](#) 类可提供多种其他有用的方法和属性，用于对数组进行排序、搜索和复制。以下示例使用 [Rank](#) 属性显示数组的维数。

```
class TestArraysClass
{
    static void Main()
    {
        // Declare and initialize an array.
        int[,] theArray = new int[5, 10];
        System.Console.WriteLine("The array has {0} dimensions.", theArray.Rank);
    }
}
// Output: The array has 2 dimensions.
```

另请参阅

- [如何使用一维数组](#)
- [如何使用多维数组](#)
- [如何使用交错数组](#)
- [对数组使用 foreach](#)
- [将数组作为参数传递](#)
- [隐式类型的数组](#)
- [C# 编程指南](#)
- [集合](#)

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。

一维数组 (C# 编程指南)

2020/11/2 • [Edit Online](#)

使用 `new` 运算符创建一维数组，该运算符指定数组元素类型和元素数目。以下示例声明一个包含五个整数的数组：

```
int[] array = new int[5];
```

此数组包含从 `array[0]` 到 `array[4]` 的元素。数组元素将初始化为元素类型的默认值，`0` 代表整数。

数组可以存储指定的任何元素类型，如声明字符串数组的下例所示：

```
string[] stringArray = new string[6];
```

数组初始化

可以在声明数组时初始化数组的元素。不需要长度说明符，因为可以根据初始化列表中的元素数量推断得出。例如：

```
int[] array1 = new int[] { 1, 3, 5, 7, 9 };
```

下面的代码显示一个字符串数组的声明，其中每个数组元素都由一天的名称初始化：

```
string[] weekDays = new string[] { "Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat" };
```

在声明时初始化数组时，可以避免使用 `new` 表达式和数组类型，如以下代码所示。这称为 [隐式类型化数组](#)：

```
int[] array2 = { 1, 3, 5, 7, 9 };
string[] weekDays2 = { "Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat" };
```

可以在尚未创建数组变量的情况下声明数组变量，但必须使用 `new` 运算符向此变量分配新数组。例如：

```
int[] array3;
array3 = new int[] { 1, 3, 5, 7, 9 }; // OK
//array3 = {1, 3, 5, 7, 9}; // Error
```

值类型和引用类型数组

请考虑以下数组声明：

```
SomeType[] array4 = new SomeType[10];
```

此语句的结果取决于 `SomeType` 是值类型还是引用类型。如果它是值类型，该语句将创建一个 10 个元素的数组，其中每个元素的类型都为 `SomeType`。如果 `SomeType` 是引用类型，该语句将创建一个 10 个元素的数组，其中每个元素都将被初始化为空引用。在两个实例中，元素均初始化为元素类型的默认值。有关值类型和引用类型的详细信息，请参阅[值类型和引用类型](#)。

请参阅

- [Array](#)
- [数组](#)
- [多维数组](#)
- [交错数组](#)

多维数组 (C# 编程指南)

2020/11/2 • [Edit Online](#)

数组可具有多个维度。例如，以下声明创建一个具有四行两列的二维数组。

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

以下声明创建一个具有三个维度(4、2 和 3)的数组。

```
int[, ,] array1 = new int[4, 2, 3];
```

数组初始化

声明后即可初始化数组，如以下示例所示。

```

// Two-dimensional array.
int[,] array2D = new int[,] { { 1, 2 }, { 3, 4 }, { 5, 6 }, { 7, 8 } };
// The same array with dimensions specified.
int[,] array2Da = new int[4, 2] { { 1, 2 }, { 3, 4 }, { 5, 6 }, { 7, 8 } };
// A similar array with string elements.
string[,] array2Db = new string[3, 2] { { "one", "two" }, { "three", "four" },
                                         { "five", "six" } };

// Three-dimensional array.
int[,,] array3D = new int[,,] { { { 1, 2, 3 }, { 4, 5, 6 } },
                                { { 7, 8, 9 }, { 10, 11, 12 } } };
// The same array with dimensions specified.
int[,,] array3Da = new int[2, 2, 3] { { { 1, 2, 3 }, { 4, 5, 6 } },
                                         { { 7, 8, 9 }, { 10, 11, 12 } } };

// Accessing array elements.
System.Console.WriteLine(array2D[0, 0]);
System.Console.WriteLine(array2D[0, 1]);
System.Console.WriteLine(array2D[1, 0]);
System.Console.WriteLine(array2D[1, 1]);
System.Console.WriteLine(array2D[3, 0]);
System.Console.WriteLine(array2Db[1, 0]);
System.Console.WriteLine(array3Da[1, 0, 1]);
System.Console.WriteLine(array3D[1, 1, 2]);

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

// Output:
// 1
// 2
// 3
// 4
// 7
// three
// 8
// 12
// 12 equals 12

```

还可在不指定级别的情况下初始化数组。

```
int[,] array4 = { { 1, 2 }, { 3, 4 }, { 5, 6 }, { 7, 8 } };
```

如果选择在不初始化的情况下声明数组变量，则必须使用 `new` 运算符将数组赋予变量。`new` 的用法如以下示例所示。

```
int[,] array5;
array5 = new int[,] { { 1, 2 }, { 3, 4 }, { 5, 6 }, { 7, 8 } }; // OK
//array5 = {{1,2}, {3,4}, {5,6}, {7,8}}; // Error
```

以下示例将值赋予特定的数组元素。

```
array5[2, 1] = 25;
```

同样，以下示例将获取特定数组元素的值并将其赋予变量 `elementValue`。

```
int elementValue = array5[2, 1];
```

以下代码示例将数组元素初始化为默认值(交错数组除外)。

```
int[,] array6 = new int[10, 10];
```

请参阅

- [C# 编程指南](#)
- [数组](#)
- [一维数组](#)
- [交错数组](#)

交错数组 (C# 编程指南)

2021/3/5 • [Edit Online](#)

交错数组是一个数组，其元素是数组，大小可能不同。交错数组有时称为“数组的数组”。以下示例说明如何声明、初始化和访问交错数组。

下面声明一个具有三个元素的一维数组，其中每个元素都是一维整数数组：

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

必须初始化 `jaggedArray` 的元素后才可使用它。可按下方操作初始化元素：

```
jaggedArray[0] = new int[5];
jaggedArray[1] = new int[4];
jaggedArray[2] = new int[2];
```

每个元素都是一维整数数组。第一个元素是由 5 个整数组成的数组，第二个是由 4 个整数组成的数组，而第三个是由 2 个整数组成的数组。

也可使用初始化表达式通过值来填充数组元素，这种情况下不需要数组大小。例如：

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

还可在声明数组时将其初始化，如：

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

可以使用下面的缩写形式。请注意：不能从元素初始化中省略 `new` 运算符，因为不存在元素的默认初始化：

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

交错数组是数组的数组，因此其元素为引用类型且被初始化为 `null`。

可以如下例所示访问个别数组元素：

```
// Assign 77 to the second element ([1]) of the first array ([0]):  
jaggedArray3[0][1] = 77;  
  
// Assign 88 to the second element ([1]) of the third array ([2]):  
jaggedArray3[2][1] = 88;
```

可以混合使用交错数组和多维数组。下面声明和初始化一个包含大小不同的三个二维数组元素的一维交错数组。有关详细信息，请参阅[多维数组](#)。

```
int[,] jaggedArray4 = new int[3][,]  
{  
    new int[,] { {1,3}, {5,7} },  
    new int[,] { {0,2}, {4,6}, {8,10} },  
    new int[,] { {11,22}, {99,88}, {0,9} }  
};
```

可以如本例所示访问个别元素，示例显示第一个数组的元素 `[1,0]` 的值(值为 `5`)：

```
System.Console.WriteLine("{0}", jaggedArray4[0][1, 0]);
```

方法 `Length` 返回包含在交错数组中的数组的数目。例如，假定已声明了前一个数组，则此行：

```
System.Console.WriteLine(jaggedArray4.Length);
```

返回值 3。

示例

本例生成一个数组，该数组的元素为数组自身。每一个数组元素都有不同的大小。

```
class ArrayTest
{
    static void Main()
    {
        // Declare the array of two elements.
        int[][] arr = new int[2][];

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

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

            for (int j = 0; j < arr[i].Length; j++)
            {
                System.Console.Write("{0}{1}", arr[i][j], j == (arr[i].Length - 1) ? "" : " ");
            }
            System.Console.WriteLine();
        }
        // Keep the console window open in debug mode.
        System.Console.WriteLine("Press any key to exit.");
        System.Console.ReadKey();
    }
}
/* Output:
Element(0): 1 3 5 7 9
Element(1): 2 4 6 8
*/
```

请参阅

- [Array](#)
- [C# 编程指南](#)
- [数组](#)
- [一维数组](#)
- [多维数组](#)

对数组使用 foreach (C# 编程指南)

2020/11/2 • [Edit Online](#)

`foreach` 语句提供一种简单、明了的方法来循环访问数组的元素。

对于单维数组, `foreach` 语句以递增索引顺序处理元素(从索引 0 开始并以索引 `Length - 1` 结束):

```
int[] numbers = { 4, 5, 6, 1, 2, 3, -2, -1, 0 };
foreach (int i in numbers)
{
    System.Console.Write("{0} ", i);
}
// Output: 4 5 6 1 2 3 -2 -1 0
```

对于多维数组, 遍历元素的方式为:首先增加最右边维度的索引, 然后是它左边的一个维度, 以此类推, 向左遍历元素:

```
int[,] numbers2D = new int[3, 2] { { 9, 99 }, { 3, 33 }, { 5, 55 } };
// Or use the short form:
// int[,] numbers2D = { { 9, 99 }, { 3, 33 }, { 5, 55 } };

foreach (int i in numbers2D)
{
    System.Console.Write("{0} ", i);
}
// Output: 9 99 3 33 5 55
```

但对于多维数组, 使用嵌套的 `for` 循环可以更好地控制处理数组元素的顺序。

请参阅

- [Array](#)
- [C# 编程指南](#)
- [数组](#)
- [一维数组](#)
- [多维数组](#)
- [交错数组](#)

将数组作为参数传递 (C# 编程指南)

2020/11/2 • [Edit Online](#)

数组可以作为实参传递给方法形参。由于数组是引用类型，因此方法可以更改元素的值。

将一维数组作为参数传递

可将初始化的一维数组传递给方法。例如，下列语句将一个数组发送给了 Print 方法。

```
int[] theArray = { 1, 3, 5, 7, 9 };
PrintArray(theArray);
```

下面的代码演示 Print 方法的部分实现。

```
void PrintArray(int[] arr)
{
    // Method code.
}
```

可在同一步骤中初始化并传递新数组，如下例所示。

```
PrintArray(new int[] { 1, 3, 5, 7, 9 });
```

示例

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

```

using System;

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

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

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

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

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

        // Assign new values to individual array elements.
        ChangeArrayElements(weekDays);
        // Display the array again to verify that it has changed.
        Console.WriteLine("Array weekDays after the call to ChangeArrayElements:");
        DisplayArray(weekDays);
    }
}

// The example displays the following output:
//      Sun Mon Tue Wed Thu Fri Sat
//
//      Array weekDays after the call to ChangeArray:
//      Sat Fri Thu Wed Tue Mon Sun
//
//      Array weekDays after the call to ChangeArrayElements:
//      Mon Wed Fri Wed Tue Mon Sun

```

将多维数组作为参数传递

通过与传递一维数组相同的方式，向方法传递初始化的多维数组。

```

int[,] theArray = { { 1, 2 }, { 2, 3 }, { 3, 4 } };
Print2DArray(theArray);

```

下列代码演示了 Print 方法的部分声明（该方法接受将二维数组作为其参数）。

```

void Print2DArray(int[,] arr)
{
    // Method code.
}

```

可在一个步骤中初始化并传递新数组，如下例所示：

```
Print2DArray(new int[,] { { 1, 2 }, { 3, 4 }, { 5, 6 }, { 7, 8 } });
```

示例

在下列示例中，初始化一个整数的二维数组，并将其传递至 `Print2DArray` 方法。该方法将显示数组的元素。

```
class ArrayClass2D
{
    static void Print2DArray(int[,] arr)
    {
        // Display the array elements.
        for (int i = 0; i < arr.GetLength(0); i++)
        {
            for (int j = 0; j < arr.GetLength(1); j++)
            {
                System.Console.WriteLine("Element({0},{1})={2}", i, j, arr[i, j]);
            }
        }
    }
    static void Main()
    {
        // Pass the array as an argument.
        Print2DArray(new int[,] { { 1, 2 }, { 3, 4 }, { 5, 6 }, { 7, 8 } });

        // Keep the console window open in debug mode.
        System.Console.WriteLine("Press any key to exit.");
        System.Console.ReadKey();
    }
}
/* Output:
Element(0,0)=1
Element(0,1)=2
Element(1,0)=3
Element(1,1)=4
Element(2,0)=5
Element(2,1)=6
Element(3,0)=7
Element(3,1)=8
*/
```

请参阅

- [C# 编程指南](#)
- [数组](#)
- [一维数组](#)
- [多维数组](#)
- [交错数组](#)

隐式类型的数组 (C# 编程指南)

2020/11/2 • [Edit Online](#)

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

隐式类型化数组通常用于查询表达式、匿名类型、对象和集合初始值设定项。

下列示例演示如何创建隐式类型化数组：

```
class ImplicitlyTypedArraySample
{
    static void Main()
    {
        var a = new[] { 1, 10, 100, 1000 }; // int[]
        var b = new[] { "hello", null, "world" }; // string[]

        // single-dimension jagged array
        var c = new[]
        {
            new[]{1,2,3,4},
            new[]{5,6,7,8}
        };

        // jagged array of strings
        var d = new[]
        {
            new[]{"Luca", "Mads", "Luke", "Dinesh"},
            new[]{"Karen", "Suma", "Frances"}
        };
    }
}
```

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

对象初始值设定项中隐式类型化数组

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

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

请参阅

- [C# 编程指南](#)
- [隐式类型的局部变量](#)
- [数组](#)
- [匿名类型](#)
- [对象和集合初始值设定项](#)
- [var](#)
- [C# 中的 LINQ](#)

字符串 (C# 编程指南)

2020/11/2 • [Edit Online](#)

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，你可以使用你所喜欢的任何一种命名约定。[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：

```
// 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);
```

请注意，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。

字符串对象的不可变性

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。

```
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:

```
string s1 = "Hello ";
string s2 = s1;
s1 += "World";

System.Console.WriteLine(s2);
//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](#)。

常规和逐字字符串文本

在必须嵌入 C# 提供的转义字符时,使用常规字符串文本,如以下示例所示:

```
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 Aeolian Harp", by Samuel Taylor Coleridge
```

当字符串文本包含反斜杠字符(例如在文件路径中)时,出于便捷性和更强的可读性的考虑,使用逐字字符串。由于逐字字符串将新的行字符作为字符串文本的一部分保留,因此可将其用于初始化多行字符串。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:

```

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."

```

字符串转义序列

转义序列	含义	Unicode 值
\'	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)	\U00HHHHHHH (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")

WARNING

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“`\xA1`”，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“`\xA1A`”（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。

NOTE

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。

```
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.
```

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结果。

```
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](#)。

```
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:

```
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:

```
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:

```
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:

```

static void Main()
{
    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：

```

System.Text.StringBuilder sb = new System.Text.StringBuilder("Rat: the ideal pet");
sb[0] = 'C';
System.Console.WriteLine(sb.ToString());
System.Console.ReadLine();

//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：

```

using System;
using System.Text;

namespace CSRefStrings
{
    class TestStringBuilder
    {
        static void Main()
        {
            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
            Console.WriteLine();
        }
    }
}

```

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](#)。

相关主题

如何修改字符串内容	阐明转换字符串并修改字符串内容的方法。
如何比较字符串	演示如何对字符串执行序号和特定于区域性的比较。
如何连接多个字符串	演示将多个字符串联接成一个字符串的多种方式。
如何使用 <code>String.Split</code> 分析字符串	包含代码示例，演示了如何使用 <code>String.Split</code> 方法来分析字符串。
如何搜索字符串	说明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	收录了介绍如何使用 <code>System.String</code> 和 <code>System.Text.StringBuilder</code> 方法执行基本字符串操作的主题链接。

分析字符串	介绍如何将 .NET 基类型的字符串表示形式转换为相应类型的实例。
分析 .NET 中的日期和时间字符串	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 类创建和修改动态字符串对象。
LINQ 和字符串	提供有关如何使用 LINQ 查询来执行各种字符串操作的信息。
C# 编程指南	提供介绍在 C# 中编程构造的主题的链接。

如何确定字符串是否表示数值 (C# 编程指南)

2021/5/7 • [Edit Online](#)

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`。

```
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。

NOTE

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`。

```
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)

2020/11/2 • [Edit Online](#)

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。

```
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.
```

NOTE

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。

```

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：

```

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。

相关部分

- [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](#)

使用索引器 (C# 编程指南)

2020/11/2 • [Edit Online](#)

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。编译器将生成一个 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：

```
// Indexer declaration
public int this[int index]
{
    // get and set accessors
}
```

IMPORTANT

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](#) 来重命名索引器，详细信息如下所示。

备注

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。

索引器值不分类为变量；因此，无法将索引器值作为 ref 或 out 参数来传递。

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：

```
// 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，因此其名称将为 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]`。

```
public class TempRecord
{
    // Array of temperature values
    float[] temps = new float[10]
    {
        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.WriteLine` 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。

```
using System;

class Program
{
    static void Main()
    {
        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]}");
        }

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
    /* Output:
     * Element #0 = 56.2
     * Element #1 = 56.7
     * Element #2 = 56.5
     * Element #3 = 58.3
     * Element #4 = 58.8
     * Element #5 = 60.1
     * Element #6 = 65.9
     * Element #7 = 62.1
     * Element #8 = 59.2
     * Element #9 = 57.5
    */
}
```

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。

```
using System;

// 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

```
using System;

class Program
{
    static void Main(string[] args)
    {
        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}");
        }
    }
    // Output:
    // 5
    // Not supported input: Day Made-up day is not supported.
    // Day input must be in the form "Sun", "Mon", etc (Parameter 'day')
}
```

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。

```
using System;
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

```
using System;

class Program
{
    static void Main()
    {
        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}");
        }
    }
    // Output:
    // 5
    // Not supported input: Day 43 is not supported.
    // Day input must be a defined System.DayOfWeek value. (Parameter 'day')
}
```

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。在本主题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](#)。

请参阅

- [C# 编程指南](#)
- [索引器](#)
- [属性](#)

接口中的索引器 (C# 编程指南)

2020/11/2 • [Edit Online](#)

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：

```
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。

```
// 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 IndexOutOfRangeException exception.
        get => arr[index];
        set => arr[index] = value;
    }
}
```

```

IndexerClass test = new IndexerClass();
System.Random rand = new System.Random();
// 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

```

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：

```

string IEmployee.this[int index]
{
}

```

在 `IEmployee` 接口中实现索引器，而以下声明：

```

string ICitizen.this[int index]
{
}

```

在 `ICitizen` 接口中实现索引器。

请参阅

- [C# 编程指南](#)
- [索引器](#)
- [属性](#)
- [接口](#)

属性和索引器之间的比较 (C# 编程指南)

2020/11/2 • [Edit Online](#)

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。

PROPERTY	INDEXER
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 <code>get</code> 访问器没有任何参数。	索引器的 <code>get</code> 访问器具有与索引器相同的形参列表。
属性的 <code>set</code> 访问器包含隐式 <code>value</code> 参数。	索引器的 <code>set</code> 访问器具有与索引器相同的形参列表， <code>value</code> 参数也是如此。
通过 自动实现的属性 支持简短语法。	支持仅使用索引器的 expression-bodied 成员。

另请参阅

- [C# 编程指南](#)
- [索引器](#)
- [属性](#)

事件 (C# 编程指南)

2020/11/2 • [Edit Online](#)

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：掌握 C# 3.0 基础知识](#)中的[委托和事件](#)

请参阅

- [EventHandler](#)
- [C# 编程指南](#)

- 委托
- 在 Windows 窗体中创建事件处理程序

如何订阅和取消订阅事件 (C# 编程指南)

2021/5/7 • [Edit Online](#)

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。

```
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：

```
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：

```
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。

```
publisher.RaiseCustomEvent += HandleCustomEvent;
```

请注意，前面的语法是 C# 2.0 中的新语法。此语法完全等效于必须使用 `new` 关键字显式创建封装委托的 C# 1.0 语法：

```
publisher.RaiseCustomEvent += new CustomEventHandler(HandleCustomEvent);
```

还可以使用 [Lambda 表达式](#) 来指定事件处理程序：

```
public Form1()
{
    InitializeComponent();
    this.Click += (s,e) =>
    {
        MessageBox.Show(((MouseEventArgs)e).Location.ToString());
    };
}
```

使用匿名方法订阅事件

- 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 (`+=`) 将匿名方法附加到此事件。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。

```
publisher.RaiseCustomEvent += delegate(object o, CustomEventArgs e)
{
    string s = o.ToString() + " " + e.ToString();
    Console.WriteLine(s);
};
```

请务必注意，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，必须返回到该事件的订阅代码，将该匿名方法存储在委托变量中，然后将此委托添加到该事件中。一般来说，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[匿名函数](#)。

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：

```
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](#)

如何发布符合 .NET 准则的事件 (C# 编程指南)

2021/5/7 • [Edit Online](#)

下面的过程演示了如何将遵循标准 .NET 模式的事件添加到类和结构中。.NET 类库中的所有事件均基于 [EventHandler](#) 委托，定义如下：

```
public delegate void EventHandler(object sender, EventArgs e);
```

NOTE

.NET Framework 2.0 引入了泛型版本的委托 [EventHandler<TEventArgs>](#)。下例演示了如何使用这两个版本。

尽管定义的类中的事件可基于任何有效委托类型，甚至是返回值的委托，但一般还是建议使用 [EventHandler](#) 使事件基于 .NET 模式，如下例中所示。

名称 `EventHandler` 可能导致一些混淆，因为它不会实际处理事件。`EventHandler` 和泛型 `EventHandler<TEventArgs>` 均为委托类型。其签名与委托定义匹配的方法或 Lambda 表达式是事件处理程序，并将在引发事件时调用。

发布基于 EventHandler 模式的事件

1. (如果无需随事件一起发送自定义数据，请跳过此步骤转到步骤 3a。) 将自定义数据的类声明为对发布服务器和订阅者类均可见的范围。然后添加所需成员以保留自定义事件数据。在此示例中，将返回一个简单的字符串。

```
public class CustomEventArgs : EventArgs
{
    public CustomEventArgs(string message)
    {
        Message = message;
    }

    public string Message { get; set; }
}
```

2. (如果使用的是泛型版本 `EventHandler<TEventArgs>`，请跳过此步骤。) 声明发布类中的委托。为其指定以 `EventHandler` 结尾的名称。第二个参数指定自定义 `EventArgs` 类型。

```
public delegate void CustomEventHandler(object sender, CustomEventArgs args);
```

3. 使用下列步骤之一来声明发布类中的事件。

- 如果没有任何自定义 `EventArgs` 类，事件类型将为非泛型 `EventHandler` 委托。你无需声明该委托，因为它已在创建 C# 项目时包括的 `System` 命名空间中声明。将以下代码添加到发布服务器类。

```
public event EventHandler RaiseCustomEvent;
```

- 如果使用非泛型版本 `EventHandler` 并且具有派生自 `EventArgs` 的自定义类，请声明发布类中的事件，并将步骤 2 中的委托用作类型。

```
public event CustomEventHandler RaiseCustomEvent;
```

- c. 如果使用泛型版本，则无需自定义委托。而是在发布类中，将事件类型指定为 `EventHandler<CustomEventArgs>`，替换尖括号中自定义类的名称。

```
public event EventHandler<CustomEventArgs> RaiseCustomEvent;
```

示例

下例通过使用自定义 EventArgs 类和 `EventHandler<EventArgs>` 作为事件类型来演示之前的步骤。

```
using System;

namespace DotNetEvents
{
    // Define a class to hold custom event info
    public class CustomEventArgs : EventArgs
    {
        public CustomEventArgs(string message)
        {
            Message = message;
        }

        public string Message { get; set; }
    }

    // Class that publishes an event
    class Publisher
    {
        // Declare the event using EventHandler<T>
        public event EventHandler<CustomEventArgs> RaiseCustomEvent;

        public void DoSomething()
        {
            // Write some code that does something useful here
            // then raise the event. You can also raise an event
            // before you execute a block of code.
            OnRaiseCustomEvent(new CustomEventArgs("Event triggered"));
        }

        // Wrap event invocations inside a protected virtual method
        // to allow derived classes to override the event invocation behavior
        protected virtual void OnRaiseCustomEvent(CustomEventArgs e)
        {
            // Make a temporary copy of the event to avoid possibility of
            // a race condition if the last subscriber unsubscribes
            // immediately after the null check and before the event is raised.
            EventHandler<CustomEventArgs> raiseEvent = RaiseCustomEvent;

            // Event will be null if there are no subscribers
            if (raiseEvent != null)
            {
                // Format the string to send inside the CustomEventArgs parameter
                e.Message += $" at {DateTime.Now}";

                // Call to raise the event.
                raiseEvent(this, e);
            }
        }
    }

    //Class that subscribes to an event
    class Subscriber
```

```
{  
    private readonly string _id;  
  
    public Subscriber(string id, Publisher pub)  
    {  
        _id = id;  
  
        // Subscribe to the event  
        pub.RaiseCustomEvent += HandleCustomEvent;  
    }  
  
    // Define what actions to take when the event is raised.  
    void HandleCustomEvent(object sender, CustomEventArgs e)  
    {  
        Console.WriteLine($"{_id} received this message: {e.Message}");  
    }  
}  
  
class Program  
{  
    static void Main()  
    {  
        var pub = new Publisher();  
        var sub1 = new Subscriber("sub1", pub);  
        var sub2 = new Subscriber("sub2", pub);  
  
        // Call the method that raises the event.  
        pub.DoSomething();  
  
        // Keep the console window open  
        Console.WriteLine("Press any key to continue...");  
        Console.ReadLine();  
    }  
}
```

请参阅

- [Delegate](#)
- [C# 编程指南](#)
- [事件](#)
- [委托](#)

如何在派生类中引发基类事件 (C# 编程指南)

2021/5/7 • [Edit Online](#)

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。

NOTE

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

```
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

- [C# 编程指南](#)
- [事件](#)
- [委托](#)

- 访问修饰符
- 在 Windows 窗体中创建事件处理程序

如何实现接口事件 (C# 编程指南)

2021/5/7 • [Edit Online](#)

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。

```
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。

```
namespace WrapTwoInterfaceEvents
{
    using System;

    public interface IDrawingObject
    {
        // Raise this event before drawing
        // the object
    }
```

```

    // 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)

2021/5/7 • [Edit Online](#)

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自订阅事件。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。

```
event EventHandler IDrawingObject.OnDraw
{
    add
    {
        lock (objectLock)
        {
            PreDrawEvent += value;
        }
    }
    remove
    {
        lock (objectLock)
        {
            PreDrawEvent -= value;
        }
    }
}
```

请参阅

- [事件](#)
- [event](#)

泛型 (C# 编程指南)

2020/11/2 • [Edit Online](#)

泛型将类型参数的概念引入 .NET，这样就可设计具有以下特征的类和方法：在客户端代码声明并初始化这些类或方法之前，这些类或方法会延迟指定一个或多个类型。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：

```
// 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。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。非泛型集合（如 [ArrayList](#)）不建议使用，并且保留用于兼容性目的。有关详细信息，请参阅 [.NET 中的泛型](#)。

当然，也可以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`。其使用方法如下：

- 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。

请注意，`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`。

```

// 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 IEnumarator<T> GetEnumarator()
    {
        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。只需更改类型参数，即可轻松修改以下代码，创建字符串或任何其他自定义类型的列表：

```
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");
    }
}
```

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。应尽可能使用这些类来代替某些类，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。
- 在泛型数据类型中所用类型的信息可在运行时通过使用反射来获取。

相关章节

- [泛型类型参数](#)
- [类型参数的约束](#)
- [泛型类](#)
- [泛型接口](#)
- [泛型方法](#)
- [泛型委托](#)
- [C++ 模板和 C# 泛型之间的区别](#)
- [泛型和反射](#)
- [运行时中的泛型](#)

C# 语言规范

有关详细信息，请参阅 [C# 语言规范](#)。

请参阅

- [System.Collections.Generic](#)
- [C# 编程指南](#)
- [类型](#)
- [`<typeparam>`](#)
- [`<typeparamref>`](#)

- .NET 中的泛型

泛型类型参数 - (C# 编程指南)

2020/11/2 • [Edit Online](#)

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：

```
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](#)。

类型参数命名指南

- 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。

```
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。

```
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"。

```
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。

另请参阅

- [System.Collections.Generic](#)
- [C# 编程指南](#)
- [泛型](#)
- [C++ 模板和 C# 泛型之间的区别](#)

类型参数的约束 (C# 编程指南)

2021/3/5 • [Edit Online](#)

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：

约束	说明
<code>where T : struct</code>	类型参数必须是不可为 null 的值类型。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，因此 <code>struct</code> 约束表示 <code>new()</code> 约束，并且不能与 <code>new()</code> 约束结合使用。 <code>struct</code> 约束也不能与 <code>unmanaged</code> 约束结合使用。
<code>where T : class</code>	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 C#8.0 或更高版本中的可为 null 上下文中， <code>T</code> 必须是不可为 null 的引用类型。
<code>where T : class?</code>	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。
<code>where T : notnull</code>	类型参数必须是不可为 null 的类型。参数可以是 C# 8.0 或更高版本中的不可为 null 的引用类型，也可以是不可为 null 的值类型。
<code>where T : unmanaged</code>	类型参数必须是不可为 null 的非托管类型。 <code>unmanaged</code> 约束表示 <code>struct</code> 约束，且不能与 <code>struct</code> 约束或 <code>new()</code> 约束结合使用。
<code>where T : new()</code>	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， <code>new()</code> 约束必须最后指定。 <code>new()</code> 约束不能与 <code>struct</code> 和 <code>unmanaged</code> 约束结合使用。
<code>where T : <base class name></code>	类型参数必须是指定的基类或派生自指定的基类。在 C# 8.0 及更高版本中的可为 null 上下文中， <code>T</code> 必须是从指定基类派生的不可为 null 的引用类型。
<code>where T : <base class name>?</code>	类型参数必须是指定的基类或派生自指定的基类。在 C# 8.0 及更高版本中的可为 null 上下文中， <code>T</code> 可以是从指定基类派生的可为 null 或不可为 null 的类型。
<code>where T : <interface name></code>	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 C# 8.0 及更高版本中的可为 null 上下文中， <code>T</code> 必须是实现指定接口的不可为 null 的类型。
<code>where T : <interface name>?</code>	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 C# 8.0 中的可为 null 上下文中， <code>T</code> 可以是可为 null 的引用类型、不可为 null 的引用类型或值类型。 <code>T</code> 不能是可为 null 的值类型。

```
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。

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或调用 `System.Object` 不支持的任何方法, 则必须对类型参数应用约束。例如, 基类约束告诉编译器, 仅此类型的对象或派生自此类型的对象可用作类型参数。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。

```

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：

```
class EmployeeList<T> where T : Employee, IEmployee, System.IComparable<T>, new()
{
    // ...
}
```

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`。

```
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, 建议同时应用 `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:

```
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, 则该比较将始终返回 `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:

```
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：

```
//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

从 C# 8.0 开始，在可为 `null` 上下文中，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。`notnull` 约束只能在 `nullable enable` 上下文中使用。如果在可以为 `null` 的不明显上下文中添加 `notnull` 约束，则编译器将生成警告。

与其他约束不同，如果类型参数违反 `notnull` 约束，那么在 `nullable enable` 上下文中编译该代码时，编译器会生成警告。如果在可以为 `null` 的不明显上下文中编译代码，则编译器不会生成任何警告或错误。

从 C# 8.0 开始，在可为 `null` 上下文中，`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。

非托管约束

从 C# 7.3 开始，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：

```
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

同样从 C# 7.3 开始，可将 `System.Delegate` 或 `System.MulticastDelegate` 用作基类约束。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：

```
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：

```
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);
```

如果取消评论最后一行，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

从 C# 7.3 开始，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。

```
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：

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

```
var map = EnumNamedValues<Rainbow>();

foreach (var pair in map)
    Console.WriteLine($"{pair.Key}:\t{pair.Value}");
```

请参阅

- [System.Collections.Generic](#)
- [C# 编程指南](#)
- [泛型介绍](#)
- [泛型类](#)
- [new 约束](#)

泛型类 (C# 编程指南)

2020/11/2 • [Edit Online](#)

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>`，客户端代码可通过指定类型参数来引用类，创建封闭式构造类型(`Node<int>`)。或者，可以不指定类型参数(例如指定泛型基类时)，创建开放式构造类型(`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：

```
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。

```
//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：

```
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：

```
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：

```
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：

```
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](#)
- [C# 编程指南](#)
- [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)

2021/3/22 • [Edit Online](#)

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。为避免对值类型的装箱和取消装箱操作，泛型类的首选项使用泛型接口，例如 `IComparable<T>` 而不是 `IComparable`。`.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。

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>`。这可使 `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`。

```
//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 = new string[]
        {
            "Franscoise",
            "Bill",
            "Li",
            "Sandra",
            "Gunnar",
            "Alok",
            "Hiroyuki",
            "Maria",
            "Alessandro",
            "Raul"
        };

        int[] ages = new int[] { 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：

```

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：

```

interface IDictionary<K, V>
{
}

```

适用于类的继承规则也适用于接口：

```

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：

```

interface IBaseInterface<T> { }

class SampleClass : IBaseInterface<string> { }

```

只要类型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：

```

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](#)。

另请参阅

- [C# 编程指南](#)
- [泛型介绍](#)
- [interface](#)

- 泛型

泛型方法 (C# 编程指南)

2020/11/2 • [Edit Online](#)

泛型方法是通过类型参数声明的方法，如下所示：

```
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：

```
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：

```
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：

```
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。

```
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。

```
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：

```
void DoWork() { }
void DoWork<T>() { }
void DoWork<T, U>() { }
```

C# 语言规范

有关详细信息，请参阅 [C# 语言规范](#)。

请参阅

- [System.Collections.Generic](#)
- [C# 编程指南](#)
- [泛型介绍](#)
- [方法](#)

泛型和数组 (C# 编程指南)

2020/11/2 • [Edit Online](#)

在 C# 2.0 和更高版本中，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)。

```
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](#)
- [C# 编程指南](#)
- [泛型](#)
- [数组](#)
- [泛型](#)

泛型委托 (C# 编程指南)

2020/11/2 • [Edit Online](#)

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：

```
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：

```
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。

```
class Stack<T>
{
    T[] items;
    int index;

    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：

```
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。

```
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)
    {
        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](#)
- [C# 编程指南](#)
- [泛型介绍](#)
- [泛型方法](#)
- [泛型类](#)
- [泛型接口](#)
- [委托](#)
- [泛型](#)

C++ 模板和 C# 泛型之间的区别 (C# 编程指南)

2020/11/2 • [Edit Online](#)

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 <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)

2020/11/2 • [Edit Online](#)

泛型类型或方法编译为 Microsoft 中间语言 (MSIL) 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 MSIL 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，MSIL 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：

```
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：

```
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 MSIL 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 MSIL 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：

```
class Customer { }
class Order { }
```

```
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`：

```
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：

```
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](#)
- [C# 编程指南](#)
- [泛型介绍](#)
- [泛型](#)

泛型和反射 (C# 编程指南)

2020/11/2 • [Edit Online](#)

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](#)。

在 .NET Framework 2.0 中，向 `Type` 类添加了多个新成员来启用泛型类型的运行时信息。有关如何使用这些方法和属性的详细信息，请参阅这些类的文档。`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。

SYSTEM. TYPE	
<code>IsGenericType</code>	如果类型是泛型，则返回 <code>true</code> 。
<code>GetGenericArguments</code>	返回 <code>Type</code> 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。
<code>GetGenericTypeDefinition</code>	返回当前构造类型的基础泛型类型定义。
<code>GetGenericParameterConstraints</code>	返回表示当前泛型类型参数约束的 <code>Type</code> 对象的数组。
<code>ContainsGenericParameters</code>	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 <code>true</code> 。
<code>GenericParameterAttributes</code>	获取描述当前泛型类型参数的特殊约束的 <code>GenericParameterAttributes</code> 标志组合。
<code>GenericParameterPosition</code>	对于表示类型参数的 <code>Type</code> 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。
<code>IsGenericParameter</code>	获取一个值，该值指示当前 <code>Type</code> 是否表示泛型类型或方法定义中的类型参数。
<code>IsGenericTypeDefinition</code>	获取一个值，该值指示当前 <code>Type</code> 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 <code>true</code> 。
<code>DeclaringMethod</code>	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 <code>null</code> 。
<code>MakeGenericType</code>	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 <code>Type</code> 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。

SYSTEM.REFLECTION.MEMBERINFO 成员	
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。

请参阅

- [C# 编程指南](#)
- [泛型](#)
- [反射类型和泛型类型](#)
- [泛型](#)

泛型和特性 (C# 编程指南)

2020/11/2 • [Edit Online](#)

属性可按与非泛型类型相同的方式应用到泛型类型。有关应用特性的详细信息，请参阅[属性](#)。

仅允许自定义属性引用开放式泛型类型(即未向其提供任何类型参数的泛型类型)和封闭式构造泛型类型(即向所有类型参数提供参数的泛型类型)。

以下示例使用此自定义属性：

```
class CustomAttribute : System.Attribute
{
    public System.Object info;
}
```

属性可引用开放式泛型类型：

```
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：

```
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：

```
public class GenericClass3<T, U, V> { }

[CustomAttribute(info = typeof(GenericClass3<int, double, string>))]
class ClassC { }
```

引用泛型类型参数的属性会导致编译时错误：

```
//[CustomAttribute(info = typeof(GenericClass3<int, T, string>))] //Error
class ClassD<T> { }
```

不能从[Attribute](#)继承泛型类型：

```
//public class CustomAtt<T> : System.Attribute {} //Error
```

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](#)

- [C# 编程指南](#)
- [泛型](#)
- [特性](#)

命名空间 (C# 编程指南)

2021/3/5 • [Edit Online](#)

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：

```
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。可以使用 `using` 关键字，如此则不必使用完整的名称，如下例所示：

```
using System;
```

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

有关详细信息，请参阅 [using 指令](#)。

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：

```
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](#)。

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。

请参阅

- [C# 编程指南](#)
- [使用命名空间](#)

- 如何使用 My 命名空间
- 标识符名称
- using 指令
- ::运算符

using 命名空间 (C# 编程指南)

2020/11/2 • [Edit Online](#)

在 C# 编程中，命名空间在两个方面被大量使用。首先，.NET 类使用命名空间来组织它的众多类。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名称和方法名称的范围。

访问命名空间

大多数 C# 应用程序以 `using` 指令开头。本部分列出了应用程序将频繁使用的命名空间，使程序员避免在每次使用包含在命名空间中的方法时都要指定完全限定名称。

例如，通过包括行：

```
using System;
```

在程序的开始，程序员可以使用代码：

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

而不是：

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

命名空间别名

还可以使用 `using` 指令为命名空间创建别名。使用命名空间别名限定符 `::` 访问已设置别名的命名空间的成员。下面的示例演示如何创建和使用命名空间别名：

```
using generics = System.Collections.Generic;

namespace AliasExample
{
    class TestClass
    {
        static void Main()
        {
            generics::Dictionary<string, int> dict = new generics::Dictionary<string, int>()
            {
                ["A"] = 1,
                ["B"] = 2,
                ["C"] = 3
            };

            foreach (string name in dict.Keys)
            {
                System.Console.WriteLine($"{name} {dict[name]}");
            }
            // Output:
            // A 1
            // B 2
            // C 3
        }
    }
}
```

使用命名空间控制范围

`namespace` 关键字用于声明范围。在项目内创建范围有助于整理代码，并允许创建全局唯一类型。在下面的示例中，名为 `SampleClass` 的类在两个命名空间中定义，其中一个嵌套在另一个之中。`.` 令牌用于区分调用的方法。

```

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

    // Create a nested namespace, and define another class.
    namespace NestedNamespace
    {
        class SampleClass
        {
            public void SampleMethod()
            {
                System.Console.WriteLine(
                    "SampleMethod inside NestedNamespace");
            }
        }
    }
}

class Program
{
    static void Main(string[] args)
    {
        // Displays "SampleMethod inside SampleNamespace."
        SampleClass outer = new SampleClass();
        outer.SampleMethod();

        // Displays "SampleMethod inside SampleNamespace."
        SampleNamespace.SampleClass outer2 = new SampleNamespace.SampleClass();
        outer2.SampleMethod();

        // Displays "SampleMethod inside NestedNamespace."
        NestedNamespace.SampleClass inner = new NestedNamespace.SampleClass();
        inner.SampleMethod();
    }
}
}

```

完全限定的名称

命名空间和类型具有指示逻辑层次结构的完全限定名称所描述的唯一标题。例如，语句 `A.B` 意味着 `A` 是命名空间或类型的名称，而 `B` 嵌套在其中。

以下示例中具有嵌套的类和命名空间。完全限定名称表示为跟随每个实体的注释。

```

namespace N1      // N1
{
    class C1      // N1.C1
    {
        class C2  // N1.C1.C2
        {
        }
    }
    namespace N2  // N1.N2
    {
        class C2  // N1.N2.C2
        {
        }
    }
}

```

在前面的代码片段中：

- 命名空间 `N1` 是全局命名空间的成员。其完全限定名称为 `N1`。
- 命名空间 `N2` 是 `N1` 的成员。其完全限定名称为 `N1.N2`。
- 类 `C1` 是 `N1` 的成员。其完全限定名称为 `N1.C1`。
- 类名 `C2` 在此代码中使用了两次。但完全限定名称是唯一的。`C2` 的第一个实例在 `C1` 中声明；因此，其完全限定名称是：`N1.C1.C2`。`C2` 的第二个实例在命名空间 `N2` 中声明；因此，其完全限定名称是 `N1.N2.C2`。

使用前面的代码段，可向命名空间 `N1.N2` 添加新的类成员 `C3`，如下所示：

```

namespace N1.N2
{
    class C3  // N1.N2.C3
    {
    }
}

```

一般情况下，使用[命名空间别名限定符](#) `::` 来引用命名空间别名，或使用 `global::` 来引用全局命名空间，以及使用 `.` 来限定类型或成员。

将 `::` 与引用类型而非引用命名空间的别名一起使用是错误的。例如：

```
using Alias = System.Console;
```

```

class TestClass
{
    static void Main()
    {
        // Error
        //Alias::WriteLine("Hi");

        // OK
        Alias.WriteLine("Hi");
    }
}

```

请记住，单词 `global` 不是预定义的别名；因此，`global.x` 不具有任何特殊含义。仅当将其与 `::` 一起使用时，它才具有特殊意义。

定义名为 global 的别名时将生成编译器警告 CS0440，因为 `global::` 始终引用全局命名空间而非别名。例如，以下行将生成该警告：

```
using global = System.Collections; // Warning
```

将 `::` 与别名一起使用是一个不错的主意，可防止意外引入其他类型。例如，请考虑以下示例：

```
using Alias = System;
```

```
namespace Library
{
    public class C : Alias::Exception { }
}
```

这是可行的，但如果随后要引入名为 `Alias` 的类型，`Alias.` 将改为绑定到该类型。使用 `Alias::Exception` 确保 `Alias` 被视为命名空间别名，而不被误解为类型。

请参阅

- [C# 编程指南](#)
- [命名空间](#)
- [成员访问表达式](#)
- [:: 运算符](#)
- [外部别名](#)

如何使用 My 命名空间 (C# 编程指南)

2020/11/2 • [Edit Online](#)

`Microsoft.VisualBasic.MyServices` 命名空间(在 Visual Basic 中为 `My`)使访问多个 .NET 类变得轻松直观, 让你能够编写与计算机、应用程序、设置、资源等交互的代码。虽然最初设计用于 Visual Basic, 但 `MyServices` 命名空间仍可用于 C# 应用程序。

有关使用 Visual Basic 的 `MyServices` 命名空间的详细信息, 请参阅[使用 My 开发](#)。

添加引用

可以在解决方案中使用 `MyServices` 类之前, 必须添加对 Visual Basic 库的引用。

添加对 Visual Basic 库的引用

1. 在解决方案资源管理器中, 右键单击“引用”节点并选择“添加引用”。
2. 出现“引用”对话框时, 向下滚动列表, 然后选择“Microsoft.VisualBasic.dll”。

同时建议将以下行包括在程序开头的 `using` 部分。

```
using Microsoft.VisualBasic.Devices;
```

示例

此示例调用 `MyServices` 命名空间中包含的各种静态方法。若要编译此代码, 必须向项目添加对 Microsoft.VisualBasic.DLL 的引用。

```
using System;
using Microsoft.VisualBasic.Devices;

class TestMyServices
{
    static void Main()
    {
        // Play a sound with the Audio class:
        Audio myAudio = new Audio();
        Console.WriteLine("Playing sound...");
        myAudio.Play(@"c:\WINDOWS\Media\chimes.wav");

        // Display time information with the Clock class:
        Clock myClock = new Clock();
        Console.Write("Current day of the week: ");
        Console.WriteLine(myClock.LocalTime.DayOfWeek);
        Console.Write("Current date and time: ");
        Console.WriteLine(myClock.LocalTime);

        // Display machine information with the Computer class:
        Computer myComputer = new Computer();
        Console.WriteLine("Computer name: " + myComputer.Name);

        if (myComputer.Network.IsAvailable)
        {
            Console.WriteLine("Computer is connected to network.");
        }
        else
        {
            Console.WriteLine("Computer is not connected to network.");
        }
    }
}
```

并不是 `MyServices` 命名空间中的所有类均可从 C# 应用程序中调用：例如，`FileSystemProxy` 类不兼容。在此特定情况下，可以改为使用属于 `FileSystem` 的静态方法，这些方法也包含在 `VisualBasic.dll` 中。例如，下面介绍了如何使用此类方法来复制目录：

```
// Duplicate a directory
Microsoft.VisualBasic.FileIO.FileSystem.CopyDirectory(
    @"C:\original_directory",
    @"C:\copy_of_original_directory");
```

请参阅

- [C# 编程指南](#)
- [命名空间](#)
- [使用命名空间](#)

XML 文档注释 (C# 编程指南)

2021/5/7 • [Edit Online](#)

使用 C#, 可以通过以下方式为代码创建文档: 将特殊注释字段中的 XML 元素包含在源代码中注释引用的代码块的前面, 例如:

```
/// <summary>
/// This class performs an important function.
/// </summary>
public class MyClass {}
```

使用 [DocumentationFile](#) 选项进行编译时, 编译器会在源代码中搜索所有 XML 标记, 并创建一个 XML 文档文件。若要基于编译器生成的文件创建最终文档, 可以创建一个自定义工具, 也可以使用 [DocFX](#) 或 [Sandcastle](#) 等工具。

若要引用 XML 元素(例如, 你的函数将处理你要在 XML 文档注释中描述的特定 XML 元素), 你可使用标准引用机制(`<` 和 `>`)。若要引用代码引用 (`cref`) 元素中的通用标识符, 可使用转义字符(例如,

`cref="List<T>"`)或大括号 (`cref="List{T}"`)。作为特例, 编译器会将大括号解析为尖括号以在引用通用标识符时使作者能够更轻松地进行文档注释。

NOTE

XML 文档注释不是元数据; 它们不包括在编译的程序集中, 因此无法通过反射对其进行访问。

本节内容

- [建议的文档注释标记](#)
- [处理 XML 文件](#)
- [文档标记分隔符](#)
- [如何使用 XML 文档功能](#)

相关章节

有关详细信息, 请参见:

- [DocumentationFile\(处理文档注释\)](#)

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](#)

建议的文档注释标记 (C# 编程指南)

2021/5/10 • [Edit Online](#)

C# 编译器处理代码中的文档注释，并在文件中将其设置为 XML 格式，该文件的名称通过 /doc 命令行选项指定。若要基于编译器生成的文件创建最终文档，可以创建一个自定义工具，也可以使用 [DocFX](#) 或 [Sandcastle](#) 等工具。

在类型和类型成员等代码构造中处理标记。

NOTE

不可对命名空间应用文档注释。

编译器将处理属于有效 XML 形式的任何标记。下列标记提供用户文档的中常用功能。

Tags

<c>
<code>
<example>
<exception>*
<include>*
<inheritdoc>

<list>
<para>
<param>*
<paramref>
<permission>*
<remarks>

<returns>
<see>*
<seealso>*
<summary>
<typeparam>*
<typeparamref>
<value>

(* 表示编译器验证语法。)

如果希望在文档注释的文本中显示尖括号，请使用 < 和 > 的 HTML 编码，分别为 < 和 >。下面的示例对此编码进行了演示。

```
/// <summary>
/// This property always returns a value &lt; 1.
/// </summary>
```

请参阅

- [C# 编程指南](#)

- DocumentationFile(C# 编译器选项)

- XML 文档注释

处理 XML 文件 (C# 编程指南)

2021/5/7 • [Edit Online](#)

编译器为代码(已标记以生成文档)中的每个构造生成一个 ID 字符串。(有关如何标记代码的信息,请参阅[文档注释的建议标记](#)。)ID 字符串唯一标识构造。处理 XML 文件的程序可以使用 ID 字符串来标识文档应用于的相应 .NET 元数据或反射项目。

ID 字符串

XML 文件不是代码的层次结构表示形式。它是一个简单列表,其中每个元素都有一个生成 ID。

编译器在生成 ID 字符串时应遵循以下规则:

- 字符串不得包含空白符。
- 该字符串的第一部分使用单个字符后跟冒号来标识成员类型。使用下面的成员类型:

II	III	II
N	namespace	无法将文档注释添加到命名空间中,但可以在支持的情况下对它们进行 cref 引用。
T	type	类型可能是类、接口、结构、枚举或委托。
F	Field — 字段	
P	property	包括索引器或其他索引属性。
M	method	包括特殊方法,如构造函数和运算符。
E	event	
!	错误字符串	字符串的其余部分提供有关错误的信息。C# 编译器将生成无法解析的链接的错误信息。

- 该字符串的第二部分是项目的完全限定名称,从命名空间的根开始。用句点分隔项目名称、其封闭类型和命名空间。如果项目名称本身包含句点,会将其替换为哈希符号('#')。假定没有名称中恰好包含哈希符号的项目。例如, String 构造函数的完全限定名称是"System.String.#ctor"。
- 对于属性和方法,括号内的参数列表如下。如果没有任何参数,则不会出现括号。这些参数以逗号分隔。每个参数的编码直接遵循它在 .NET 签名中的编码方式:
 - 基类型。常规的类型(ELEMENT_TYPE_CLASS 或 ELEMENT_TYPE_VALUETYPE)表示为该类型的完全限定名称。
 - 内部类型(例如, ELEMENT_TYPE_I4、ELEMENT_TYPE_OBJECT、ELEMENT_TYPE_STRING、ELEMENT_TYPE_TYPEDBYREF 和 ELEMENT_TYPE_VOID)表示为相应完整类型的完全限定名称。例如, System.Int32 或 System.TypedReference。

- ELEMENT_TYPE_PTR 表示为修改类型之后的“*”。
- ELEMENT_TYPE_BYREF 表示为修改类型之后的“@”。
- ELEMENT_TYPE_PINNED 表示为修改类型之后的“^”。C# 编译器不会生成此类型。
- ELEMENT_TYPE_CMOD_REQ 表示为“|”和修饰符类的完全限定名称，前面是修改类型。C# 编译器不会生成此类型。
- ELEMENT_TYPE_CMOD_OPT 表示为“!”和修饰符类的完全限定名称，前面是修改类型。
- ELEMENT_TYPE_SZARRAY 表示为“[]”，前面是数组的元素类型。
- ELEMENT_TYPE_GENERICARRAY 表示为“[?]”，前面是数组的元素类型。C# 编译器不会生成此类型。
- ELEMENT_TYPE_ARRAY 表示为 `[lowerbound.size,lowerbound.size]`，其中逗号的数量是秩 - 1，每个维度的下限和大小（如果已知）以十进制形式表示。如果未指定下限或大小，则将其省略。如果省略某个特定维度的下限和大小，也会省略“：“。例如，以 1 作为下限并且未指定大小的二维数组是 `[1;1:]`。
- ELEMENT_TYPE_FNPTR 表示为 “=FUNC:`type` (`signature`)”，其中 `type` 是返回类型，`signature` 是方法的自变量。如果没有任何自变量，则省略括号。C# 编译器不会生成此类型。

不表示下列签名组件，因为它们不会用于区分重载方法：

- 调用约定
- 返回类型
- ELEMENT_TYPE_SENTINEL
- 仅针对转换运算符（`op_Implicit` 和 `op_Explicit`），该方法的返回值被编码为“~”，后面是返回类型。
- 对于泛型类型，类型名称后跟反引号，然后是指示泛型类型参数数量的一个数字。例如：

`<member name="T:SampleClass`2">` 是定义为 `public class SampleClass<T, U>` 的类型的标记。

对于将泛型类型用作参数的方法，泛型类型参数被指定为前面加上反引号的数字（例如 `0、`1）。每个数字表示该类型的泛型参数的从零开始的数组表示法。

示例

下面的示例演示如何为类及其成员生成 ID 字符串：

```
namespace N
{
    /// <summary>
    /// Enter description here for class X.
    /// ID string generated is "T:N.X".
    /// </summary>
    public unsafe class X
    {
        /// <summary>
        /// Enter description here for the first constructor.
        /// ID string generated is "M:N.X.#ctor".
        /// </summary>
        public X() { }

        /// <summary>
        /// Enter description here for the second constructor.
        /// ID string generated is "M:N.X.#ctor(System.Int32)".
        /// </summary>
        /// <param name="i">Description for parameter i</param>
    }
}
```

```

/// <param name= i >Describe parameter.</param>
public X(int i) { }

/// <summary>
/// Enter description here for field q.
/// ID string generated is "F:N.X.q".
/// </summary>
public string q;

/// <summary>
/// Enter description for constant PI.
/// ID string generated is "F:N.X.PI".
/// </summary>
public const double PI = 3.14;

/// <summary>
/// Enter description for method f.
/// ID string generated is "M:N.X.f".
/// </summary>
/// <returns>Describe return value.</returns>
public int f() { return 1; }

/// <summary>
/// Enter description for method bb.
/// ID string generated is "M:N.X.bb(System.String,System.Int32@,System.Void*)".
/// </summary>
/// <param name="s">Describe parameter.</param>
/// <param name="y">Describe parameter.</param>
/// <param name="z">Describe parameter.</param>
/// <returns>Describe return value.</returns>
public int bb(string s, ref int y, void* z) { return 1; }

/// <summary>
/// Enter description for method gg.
/// ID string generated is "M:N.X.gg(System.Int16[],System.Int32[0:,0:])".
/// </summary>
/// <param name="array1">Describe parameter.</param>
/// <param name="array">Describe parameter.</param>
/// <returns>Describe return value.</returns>
public int gg(short[] array1, int[,] array) { return 0; }

/// <summary>
/// Enter description for operator.
/// ID string generated is "M:N.X.op>Addition(N.X,N.X)".
/// </summary>
/// <param name="x">Describe parameter.</param>
/// <param name="xx">Describe parameter.</param>
/// <returns>Describe return value.</returns>
public static X operator +(X x, X xx) { return x; }

/// <summary>
/// Enter description for property.
/// ID string generated is "P:N.X.prop".
/// </summary>
public int prop { get { return 1; } set { } }

/// <summary>
/// Enter description for event.
/// ID string generated is "E:N.X.d".
/// </summary>
public event D d;

/// <summary>
/// Enter description for property.
/// ID string generated is "P:N.X.Item(System.String)".
/// </summary>
/// <param name="s">Describe parameter.</param>
/// <returns></returns>
public int this[string s] { get { return 1; } }

```

```
/// <summary>
/// Enter description for class Nested.
/// ID string generated is "T:N.X.Nested".
/// </summary>
public class Nested { }

/// <summary>
/// Enter description for delegate.
/// ID string generated is "T:N.X.D".
/// </summary>
/// <param name="i">Describe parameter.</param>
public delegate void D(int i);

/// <summary>
/// Enter description for operator.
/// ID string generated is "M:N.X.op_Explicit(N.X)~System.Int32".
/// </summary>
/// <param name="x">Describe parameter.</param>
/// <returns>Describe return value.</returns>
public static explicit operator int(X x) { return 1; }
}
```

请参阅

- [C# 编程指南](#)
- [DocumentationFile\(C# 编译器选项\)](#)
- [XML 文档注释](#)

文档标记分隔符 (C# 编程指南)

2021/5/7 • [Edit Online](#)

XML 文档注释需要使用分隔符，用来向编译器指示文档注释开始和结束的位置。可以使用以下采用 XML 文档标记的分隔符：

- `///`

单行分隔符。这是在文档示例中显示的格式，由 C# 项目模板使用。如果在分隔符后面有一个空格字符，那么此字符不会包括在 XML 输出中。

NOTE

在代码编辑器中键入 `///` 分隔符后，Visual Studio 集成开发环境 (IDE) 可自动插入 `<summary>` 和 `</summary>` 标记，并在这些标记中移动游标。可以在“[选项](#)”对话框中打开/关闭此功能。

- `/** */`

多行分隔符。

使用 `/** */` 分隔符时需遵守一些格式设置规则：

- 在包含 `/**` 分隔符的行上，如果行的其余部分为空格，则不将此行作为注释处理。如果 `/**` 分隔符后面的第一个字符为空格，则忽略此空格字符，并处理行的其余部分。否则，将 `/**` 分隔符后面的行的所有文本作为注释的一部分进行处理。
- 在包含 `*/` 分隔符的行中，如果 `*/` 分隔符前面只有空格，此行将被忽略。否则，将 `*/` 分隔符之前的行的文本作为注释的一部分进行处理。
- 对于以 `/**` 分隔符开头的行后面的行，编译器在各行的开头寻找共同模式。此模式可以包含空格和星号 (`*`)，后面跟更多空格。如果编译器在不以 `/**` 分隔符或 `*/` 分隔符开头的各行开头找到共同模式，则忽略此每个行的模式。

以下示例说明了这些规则。

- 以下注释中将被处理的唯一部分是以 `<summary>` 开头的行。三种标记格式产生的注释相同。

```
/** <summary>text</summary> */  
  
/**  
<summary>text</summary>  
*/  
  
/**  
* <summary>text</summary>  
*/
```

- 编译器识别出第二和第三行开头的共同模式“*”。此模式不包括在输出中。

```
/**  
* <summary>  
* text </summary>*/
```

- 编译器在下面的注释中未找到共同模式，因为第三行的第二个字符不是一个星号。因此，第二和第三行上的所有文本将处理为注释的一部分。

```
/**  
 * <summary>  
 *   text </summary>  
 */
```

- 编译器在以下注释中未找到模式，原因有两个。首先，星号前的空格数不一致。其次，第 5 行以制表符开头，这与空格不匹配。因此，第二到第五行的所有文本都作为注释的一部分进行处理。

```
/**  
 * <summary>  
 *   text  
 *   text2  
 *     </summary>  
 */
```

请参阅

- [C# 编程指南](#)
- [XML 文档注释](#)
- [DocumentationFile\(C# 编译器选项\)](#)

如何使用 XML 文档功能

2021/5/7 • [Edit Online](#)

下面的示例提供对某个已存档类型的基本概述。

示例

```
// If compiling from the command line, compile with: -doc:YourFileName.xml

/// <summary>
/// Class level summary documentation goes here.
/// </summary>
/// <remarks>
/// Longer comments can be associated with a type or member through
/// the remarks tag.
/// </remarks>
public class TestClass : TestInterface
{
    /// <summary>
    /// Store for the Name property.
    /// </summary>
    private string _name = null;

    /// <summary>
    /// The class constructor.
    /// </summary>
    public TestClass()
    {
        // TODO: Add Constructor Logic here.
    }

    /// <summary>
    /// Name property.
    /// </summary>
    /// <value>
    /// A value tag is used to describe the property value.
    /// </value>
    public string Name
    {
        get
        {
            if (_name == null)
            {
                throw new System.Exception("Name is null");
            }
            return _name;
        }
    }

    /// <summary>
    /// Description for SomeMethod.
    /// </summary>
    /// <param name="s"> Parameter description for s goes here.</param>
    /// <seealso cref="System.String">
    /// You can use the cref attribute on any tag to reference a type or member
    /// and the compiler will check that the reference exists.
    /// </seealso>
    public void SomeMethod(string s)
    {
    }
}
```

```

/// <summary>
/// Some other method.
/// </summary>
/// <returns>
/// Return values are described through the returns tag.
/// </returns>
/// <seealso cref="SomeMethod(string)">
/// Notice the use of the cref attribute to reference a specific method.
/// </seealso>
public int SomeOtherMethod()
{
    return 0;
}

public int InterfaceMethod(int n)
{
    return n * n;
}

/// <summary>
/// The entry point for the application.
/// </summary>
/// <param name="args"> A list of command line arguments.</param>
static int Main(System.String[] args)
{
    // TODO: Add code to start application here.
    return 0;
}

/// <summary>
/// Documentation that describes the interface goes here.
/// </summary>
/// <remarks>
/// Details about the interface go here.
/// </remarks>
interface TestInterface
{
    /// <summary>
    /// Documentation that describes the method goes here.
    /// </summary>
    /// <param name="n">
    /// Parameter n requires an integer argument.
    /// </param>
    /// <returns>
    /// The method returns an integer.
    /// </returns>
    int InterfaceMethod(int n);
}

```

该示例生成一个包含以下内容的 .xml 文件。

```

<?xml version="1.0"?>
<doc>
    <assembly>
        <name>xm样子</name>
    </assembly>
    <members>
        <member name="T:TestClass">
            <summary>
                Class level summary documentation goes here.
            </summary>
            <remarks>
                Longer comments can be associated with a type or member through
                the remarks tag.
            </remarks>
        </member>
    
```

```

<member name="F:TestClass._name">
    <summary>
        Store for the Name property.
    </summary>
</member>
<member name="M:TestClass.#ctor">
    <summary>
        The class constructor.
    </summary>
</member>
<member name="P:TestClass.Name">
    <summary>
        Name property.
    </summary>
    <value>
        A value tag is used to describe the property value.
    </value>
</member>
<member name="M:TestClass.SomeMethod(System.String)">
    <summary>
        Description for SomeMethod.
    </summary>
    <param name="s"> Parameter description for s goes here.</param>
    <seealso cref="T:System.String">
        You can use the cref attribute on any tag to reference a type or member
        and the compiler will check that the reference exists.
    </seealso>
</member>
<member name="M:TestClass.SomeOtherMethod">
    <summary>
        Some other method.
    </summary>
    <returns>
        Return values are described through the returns tag.
    </returns>
    <seealso cref="M:TestClass.SomeMethod(System.String)">
        Notice the use of the cref attribute to reference a specific method.
    </seealso>
</member>
<member name="M:TestClass.Main(System.String[])>
    <summary>
        The entry point for the application.
    </summary>
    <param name="args"> A list of command line arguments.</param>
</member>
<member name="T:TestInterface">
    <summary>
        Documentation that describes the interface goes here.
    </summary>
    <remarks>
        Details about the interface go here.
    </remarks>
</member>
<member name="M:TestInterface.InterfaceMethod(System.Int32)">
    <summary>
        Documentation that describes the method goes here.
    </summary>
    <param name="n">
        Parameter n requires an integer argument.
    </param>
    <returns>
        The method returns an integer.
    </returns>
</member>
</members>
</doc>

```

编译代码

若要编译该示例，请输入以下命令：

```
csc XMLsample.cs /doc:XMLsample.xml
```

此命令创建 XML 文件 XMLsample.xml，可在浏览器中或使用 **TYPE** 命令查看该文件。

可靠编程

XML 文档以 `///` 开头。创建新项目时，向导会放置一些以 `///` 开头的行。处理这些注释时存在一些限制：

- 文档必须是格式正确的 XML。如果 XML 格式不正确，则会生成警告，并且文档文件将包含一条注释，指出遇到错误。
- 开发人员可以随意创建自己的标记集。有一组 [推荐的标记](#)。部分建议标记具有特殊含义：
 - `<param>` 标记用于描述参数。如果已使用，编译器会验证该参数是否存在，以及文档是否描述了所有参数。如果验证失败，编译器会发出警告。
 - `cref` 属性可以附加到任何标记，以引用代码元素。编译器验证此代码元素是否存在。如果验证失败，编译器会发出警告。编译器在查找 `cref` 属性中描述的类型时会考虑所有 `using` 语句。
 - `<summary>` 标记由 Visual Studio 中的 IntelliSense 用于显示有关某个类型或成员的附加信息。

NOTE

XML 文件不提供有关该类型和成员的完整信息（例如，它不包含任何类型信息）。若要获取有关类型或成员的完整信息，请将文档文件与对实际类型或成员的反射一起使用。

请参阅

- [C# 编程指南](#)
- [DocumentationFile\(C# 编译器选项\)](#)
- [XML 文档注释](#)
- [DocFX 文档处理器](#)
- [Sandcastle 文档处理器](#)

<c> (C# 编程指南)

2021/5/7 • [Edit Online](#)

语法

```
<c>text</c>
```

参数

- `text`

要指示为代码的文本。

备注

使用 `<c>` 标记可以指示应将说明内的文本标记为代码。使用 `<code>` 指示作为代码的多行文本。

使用 [DocumentationFile](#) 进行编译可以将文档注释处理到文件中。

示例

```
// compile with: -doc:DocFileName.xml

/// text for class TestClass
public class TestClass
{
    /// <summary><c>DoWork</c> is a method in the <c>TestClass</c> class.
    /// </summary>
    public static void DoWork(int Int1)
    {
    }

    /// text for Main
    static void Main()
    {
    }
}
```

请参阅

- [C# 编程指南](#)
- [建议的文档注释标记](#)

<code> (C# 编程指南)

2021/5/7 • [Edit Online](#)

语法

```
<code>content</code>
```

参数

- `content`

要标记为代码的文本。

备注

`<code>` 标记用于指示多行代码。使用 `<c>` 指示应将说明内的单行文本标记为代码。

使用 [DocumentationFile](#) 进行编译可以将文档注释处理到文件中。

示例

有关如何使用 `<code>` 标记的示例, 请参阅 [<example>](#) 一文。

请参阅

- [C# 编程指南](#)
- [建议的文档注释标记](#)

cref 属性 (C# 编程指南)

2020/11/2 • [Edit Online](#)

XML 文档标记中的 `cref` 属性是指“代码引用”。它指定标记的内部文本是一个代码元素，例如类型、方法或属性。文档工具(例如 [DocFX](#) 和 [Sandcastle](#))使用 `cref` 属性自动生成指向记录类型或成员的页面的超链接。

示例

下面的示例演示了在 `<see>` 标记中使用的 `cref` 属性。

```
// Save this file as CRefTest.cs
// Compile with: csc CRefTest.cs -doc:Results.xml

namespace TestNamespace
{
    /// <summary>
    /// TestClass contains several cref examples.
    /// </summary>
    public class TestClass
    {
        /// <summary>
        /// This sample shows how to specify the <see cref="TestClass"/> constructor as a cref attribute.
        /// </summary>
        public TestClass()
        { }

        /// <summary>
        /// This sample shows how to specify the <see cref="TestClass(int)"/> constructor as a cref
attribute.
        /// </summary>
        public TestClass(int value)
        { }

        /// <summary>
        /// The GetZero method.
        /// </summary>
        /// <example>
        /// This sample shows how to call the <see cref="GetZero"/> method.
        /// </code>
        /// class TestClass
        /// {
        ///     static int Main()
        ///     {
        ///         return GetZero();
        ///     }
        /// }
        /// </code>
        /// <example>
        public static int GetZero()
        {
            return 0;
        }

        /// <summary>
        /// The GetGenericValue method.
        /// </summary>
        /// <remarks>
        /// This sample shows how to specify the <see cref="GetGenericValue"/> method as a cref attribute.
        /// </remarks>

        public static T GetGenericValue<T>(T para)
```

```
    {
        return para;
    }
}

/// <summary>
/// GenericClass.
/// </summary>
/// <remarks>
/// This example shows how to specify the <see cref="GenericClass{T}" /> type as a cref attribute.
/// </remarks>
class GenericClass<T>
{
    // Fields and members.
}

class Program
{
    static int Main()
    {
        return TestClass.GetZero();
    }
}
```

在编译时，该程序生成以下 XML 文件。请注意，例如 `GetZero` 方法的 `cref` 属性已被编译器转换为 `"M:TestNamespace.TestClass.GetZero"`。“M:”前缀表示“方法”，并且是一种由文档工具（例如 DocFX 和 Sandcastle）识别的约定。有关前缀的完整列表，请参阅[处理 XML 文件](#)。

```

<?xml version="1.0"?>
<doc>
    <assembly>
        <name>CRefTest</name>
    </assembly>
    <members>
        <member name="T:TestNamespace.TestClass">
            <summary>
                TestClass contains several cref examples.
            </summary>
        </member>
        <member name="M:TestNamespace.TestClass.#ctor">
            <summary>
                This sample shows how to specify the <see cref="T:TestNamespace.TestClass"/> constructor as a cref attribute.
            </summary>
        </member>
        <member name="M:TestNamespace.TestClass.#ctor(System.Int32)">
            <summary>
                This sample shows how to specify the <see cref="M:TestNamespace.TestClass.#ctor(System.Int32)"/> constructor as a cref attribute.
            </summary>
        </member>
        <member name="M:TestNamespace.TestClass.GetZero">
            <summary>
                The GetZero method.
            </summary>
            <example>
                This sample shows how to call the <see cref="M:TestNamespace.TestClass.GetZero"/> method.
                <code>
                    class TestClass
                    {
                        static int Main()
                        {
                            return GetZero();
                        }
                    }
                </code>
            </example>
        </member>
        <member name="M:TestNamespace.TestClass.GetGenericValue`1(`0)">
            <summary>
                The GetGenericValue method.
            </summary>
            <remarks>
                This sample shows how to specify the <see cref="M:TestNamespace.TestClass.GetGenericValue`1(`0)"/> method as a cref attribute.
            </remarks>
        </member>
        <member name="T:TestNamespace.GenericClass`1">
            <summary>
                GenericClass.
            </summary>
            <remarks>
                This example shows how to specify the <see cref="T:TestNamespace.GenericClass`1"/> type as a cref attribute.
            </remarks>
        </member>
    </members>
</doc>

```

请参阅

- [XML 文档注释](#)
- [建议的文档注释标记](#)

<example> (C# 编程指南)

2021/5/7 • [Edit Online](#)

语法

```
<example>description</example>
```

参数

- `description`

代码示例的说明。

备注

借助 `<example>` 标记，可以指定如何使用方法或其他库成员的示例。这通常涉及到使用 `<code>` 标记。

使用 [DocumentationFile](#) 进行编译可以将文档注释处理到文件中。

示例

```
// Save this file as CRefTest.cs
// Compile with: csc CRefTest.cs -doc:Results.xml

namespace TestNamespace
{
    /// <summary>
    /// TestClass contains several cref examples.
    /// </summary>
    public class TestClass
    {
        /// <summary>
        /// This sample shows how to specify the <see cref="TestClass"/> constructor as a cref attribute.
        /// </summary>
        public TestClass()
        { }

        /// <summary>
        /// This sample shows how to specify the <see cref="TestClass(int)"/> constructor as a cref
        /// attribute.
        /// </summary>
        public TestClass(int value)
        { }

        /// <summary>
        /// The GetZero method.
        /// </summary>
        /// <example>
        /// This sample shows how to call the <see cref="GetZero"/> method.
        /// </example>
        /// <code>
        /// class TestClass
        /// {
        ///     static int Main()
        ///     {
        ///         return GetZero();
        ///     }
        /// }
        /// </code>
    }
}
```

```
/// </code>
/// </example>
public static int GetZero()
{
    return 0;
}

/// <summary>
/// The GetGenericValue method.
/// </summary>
/// <remarks>
/// This sample shows how to specify the <see cref="GetGenericValue"/> method as a cref attribute.
/// </remarks>

public static T GetGenericValue<T>(T para)
{
    return para;
}

/// <summary>
/// GenericClass.
/// </summary>
/// <remarks>
/// This example shows how to specify the <see cref="GenericClass{T}"/> type as a cref attribute.
/// </remarks>
class GenericClass<T>
{
    // Fields and members.
}

class Program
{
    static int Main()
    {
        return TestClass.GetZero();
    }
}
```

另请参阅

- [C# 编程指南](#)
- [建议的文档注释标记](#)

<exception> (C# 编程指南)

2021/5/7 • [Edit Online](#)

语法

```
<exception cref="member">description</exception>
```

参数

- `cref = "member"`

对当前编译环境中出现的一个异常的引用。编译器检查是否存在给定的异常，并将 `member` 转换为输出 XML 中的规范的元素名称。`member` 必须出现在双引号 (" ") 内。

有关如何设置 `member` 格式以引用泛型类型的详细信息，请参阅 [处理 XML 文件](#)。

- `description`

异常的说明。

备注

`<exception>` 标记可用于指定可引发的异常。此标记可应用于方法、属性、事件和索引器的定义。

使用 [DocumentationFile](#) 进行编译可以将文档注释处理到文件中。

有关异常处理的详细信息，请参阅 [异常和异常处理](#)。

示例

```
// compile with: -doc:DocFileName.xml

/// Comment for class
public class EClass : System.Exception
{
    // class definition...
}

/// Comment for class
class TestClass
{
    /// <exception cref="System.Exception">Thrown when...</exception>
    public void DoSomething()
    {
        try
        {
        }
        catch (EClass)
        {
        }
    }
}
```

请参阅

- [C# 编程指南](#)
- [建议的文档注释标记](#)

<include> (C# 编程指南)

2020/11/2 • [Edit Online](#)

语法

```
<include file='filename' path='tagpath[@name="id"]' />
```

参数

- `filename`

包含文档的 XML 文件的名称。可使用相对于源代码文件的路径限定文件名。使用单引号 (' ') 将 `filename` 括起来。

- `tagpath`

`filename` 中标记的路径，它指向标记 `name`。使用单引号 (' ') 将路径括起来。

- `name`

标记中的名称说明符(位于注释之前)；`name` 将有 `id`。

- `id`

标记的 ID(位于注释之前)。用双引号 (" ") 将 ID 括起来。

备注

通过 `<include>` 标记，可在其他文件中引用描述源代码中类型和成员的注释。这是对直接在源代码文件中放入文档注释的替代方法。通过将文档放入不同文件，可以单独从源代码对文档应用源控件。一人可以签出源代码文件，而其他人可以签出文档文件。

`<include>` 标记使用 XML XPath 语法。有关如何自定义 `<include>` 用法的信息，请参阅 XPath 文档。

示例

这是多文件示例。下面是第一个文件，使用 `<include>`。

```
// compile with: -doc:DocFileName.xml

/// <include file='xml_include_tag.doc' path='MyDocs/MyMembers[@name="test"]/*' />
class Test
{
    static void Main()
    {
    }
}

/// <include file='xml_include_tag.doc' path='MyDocs/MyMembers[@name="test2"]/*' />
class Test2
{
    public void Test()
    {
    }
}
```

第二个文件是 `xml_include_tag.doc`, 包含下列文档注释。

```
<MyDocs>

<MyMembers name="test">
<summary>
The summary for this type.
</summary>
</MyMembers>

<MyMembers name="test2">
<summary>
The summary for this other type.
</summary>
</MyMembers>

</MyDocs>
```

程序输出

使用以下命令行编译 `Test` 和 `Test2` 类时, 便会生成下面的输出: `-doc:DocFileName.xml`。在 Visual Studio 的项目设计器的“生成”窗格中, 指定 XML 文档注释选项。当 C# 编译器发现 `<include>` 标记时, 它将在 `xml_include_tag.doc`(而不是当前源文件)中搜索文档注释。然后编译器生成 `DocFileName.xml`, 这是由文档工具(如 [Sandcastle](#))所使用的文件, 用于生成最终文档。

```
<?xml version="1.0"?>
<doc>
    <assembly>
        <name>xml_include_tag</name>
    </assembly>
    <members>
        <member name="T:Test">
            <summary>
The summary for this type.
            </summary>
        </member>
        <member name="T:Test2">
            <summary>
The summary for this other type.
            </summary>
        </member>
    </members>
</doc>
```

另请参阅

- [C# 编程指南](#)
- [建议的文档注释标记](#)

<inheritdoc> (C# 编程指南)

2021/3/5 • [Edit Online](#)

语法

```
<inheritdoc/>
```

InheritDoc

继承基类、接口和类似方法中的 XML 注释。这样不必复制和粘贴重复的 XML 注释，并自动保持 XML 注释同步。

备注

在基类或接口中添加 XML 注释，并让 InheritDoc 将注释复制到实现类中。

向同步方法添加 XML 注释，并让 InheritDoc 将注释复制到相同方法的异步版本中。

如果要从特定成员复制注释，可以使用 `cref` 特性来指定成员。

示例

```
// compile with: -doc:DocFileName.xml

/// <summary>
/// You may have some primary information about this class.
/// </summary>
public class MainClass
{
}

///<inheritdoc/>
public class TestClass: MainClass
{}
```

```
// compile with: -doc:DocFileName.xml

/// <summary>
/// You may have some primary information about this interface.
/// </summary>
public interface ITestInterface
{
}

///<inheritdoc cref="ITestInterface"/>
public class TestClass : ITestInterface
{}
```

另请参阅

- [C# 编程指南](#)

- 建议的文档注释标记

<list> (C# 编程指南)

2021/5/7 • [Edit Online](#)

语法

```
<list type="bullet|number|table">
  <listheader>
    <term>term</term>
    <description>description</description>
  </listheader>
  <item>
    <term>term</term>
    <description>description</description>
  </item>
</list>
```

参数

- `term`

要定义的术语，将在 `description` 中进行定义。

- `description`

项目符号或编号列表中的项或 `term` 的定义。

备注

`<listheader>` 块用于定义表或定义列表的标题行。定义表时，只需提供标题中的术语的项。

列表中的每个项均使用 `<item>` 块指定。创建定义列表时，需要同时指定 `term` 和 `description`。但是，对于表、项目符号列表或编号列表，只需提供 `description` 的项。

列表或表可根据需要具有多个 `<item>` 块。

使用 [DocumentationFile](#) 进行编译可以将文档注释处理到文件中。

示例

```
// compile with: -doc:DocFileName.xml

/// text for class TestClass
public class TestClass
{
    /// <summary>Here is an example of a bulleted list:
    /// <list type="bullet">
    /// <item>
    /// <description>Item 1.</description>
    /// </item>
    /// <item>
    /// <description>Item 2.</description>
    /// </item>
    /// </list>
    /// </summary>
    static void Main()
    {
    }
}
```

请参阅

- [C# 编程指南](#)
- [建议的文档注释标记](#)

<para> (C# 编程指南)

2021/5/7 • [Edit Online](#)

语法

```
<para>content</para>
```

参数

- `content`

段落文本。

备注

<para> 标记用于标记内，例如 <[summary](#)>、<[remarks](#)> 或 <[returns](#)>，允许向文本添加结构。

使用 [DocumentationFile](#) 进行编译可以将文档注释处理到文件中。

示例

有关使用 `<para>` 的示例，请参阅 <[summary](#)>。

请参阅

- [C# 编程指南](#)
- [建议的文档注释标记](#)

<param> (C# 编程指南)

2021/5/7 • [Edit Online](#)

语法

```
<param name="name">description</param>
```

参数

- `name`

方法参数的名称。用双引号 (" ") 将名称引起来。

- `description`

参数的说明。

备注

在方法声明的注释中，应使用 `<param>` 标记来描述方法参数之一。若要记录多个参数，请使用多个 `<param>` 标记。

`<param>` 标记的文本显示在 IntelliSense、对象浏览器和代码注释 Web 报表中。

使用 [DocumentationFile](#) 进行编译可以将文档注释处理到文件中。

示例

```
// compile with: -doc:DocFileName.xml

/// text for class TestClass
public class TestClass
{
    // Single parameter.
    /// <param name="Int1">Used to indicate status.</param>
    public static void DoWork(int Int1)
    {
    }

    // Multiple parameters.
    /// <param name="Int1">Used to indicate status.</param>
    /// <param name="Float1">Used to specify context.</param>
    public static void DoWork(int Int1, float Float1)
    {
    }

    // text for Main
    static void Main()
    {
    }
}
```

请参阅

- [C# 编程指南](#)
- [建议的文档注释标记](#)

<paramref> (C# 编程指南)

2021/5/7 • [Edit Online](#)

语法

```
<paramref name="name"/>
```

参数

- `name`

要引用的参数的名称。用双引号 (" ") 将名称引起来。

备注

`<paramref>` 标记提供一种方式，用于指示 `<summary>` 或 `<remarks>` 块等代码注释中的单词引用某个参数。可以处理 XML 文件以明显的方式设置此单词的格式，如使用粗体或斜体。

使用 [DocumentationFile](#) 进行编译可以将文档注释处理到文件中。

示例

```
// compile with: -doc:DocFileName.xml

/// text for class TestClass
public class TestClass
{
    /// <summary>DoWork is a method in the TestClass class.
    /// The <paramref name="int1"/> parameter takes a number.
    /// </summary>
    public static void DoWork(int int1)
    {
    }

    /// text for Main
    static void Main()
    {
    }
}
```

另请参阅

- [C# 编程指南](#)
- [建议的文档注释标记](#)

<permission> (C# 编程指南)

2021/5/7 • [Edit Online](#)

语法

```
<permission cref="member">description</permission>
```

参数

- `cref = "member"`

对可从当前编译环境调用的成员或字段的引用。编译器检查是否存在给定的代码元素，并将 `member` 转换为输出 XML 中规范的元素名称。成员必须出现在双引号 (" ") 内。

有关如何创建对泛型类型的 `cref` 引用的信息，请参阅 [cref 特性](#)。

- `description`

对成员访问权限的说明。

备注

使用 `<permission>` 标记可以记录成员访问权限。[PermissionSet](#) 类可指定对成员的访问权限。

使用 [DocumentationFile](#) 进行编译可以将文档注释处理到文件中。

示例

```
// compile with: -doc:DocFileName.xml

class TestClass
{
    /// <permission cref="System.Security.PermissionSet">Everyone can access this method.</permission>
    public static void Test()
    {
    }

    static void Main()
    {
    }
}
```

请参阅

- [C# 编程指南](#)
- [建议的文档注释标记](#)

<remarks> (C# 编程指南)

2021/5/7 • [Edit Online](#)

语法

```
<remarks>description</remarks>
```

参数

- [Description](#)

对成员的说明。

备注

[<remarks>](#) 标记用于添加有关某个类型的信息，从而补充由 [<summary>](#) 指定的信息。此信息显示在对象浏览器窗口中。

使用 [DocumentationFile](#) 进行编译可以将文档注释处理到文件中。

示例

```
// compile with: -doc:DocFileName.xml

/// <summary>
/// You may have some primary information about this class.
/// </summary>
/// <remarks>
/// You may have some additional information about this class.
/// </remarks>
public class TestClass
{
    /// text for Main
    static void Main()
    {
    }
}
```

另请参阅

- [C# 编程指南](#)
- [建议的文档注释标记](#)

<returns> (C# 编程指南)

2021/5/7 • [Edit Online](#)

语法

```
<returns>description</returns>
```

参数

- `description`

返回值的说明。

备注

在方法声明的注释中应使用 `<returns>` 标记来描述返回值。

使用 [DocumentationFile](#) 进行编译可以将文档注释处理到文件中。

示例

```
// compile with: -doc:DocFileName.xml

/// text for class TestClass
public class TestClass
{
    /// <returns>Returns zero.</returns>
    public static int GetZero()
    {
        return 0;
    }

    /// text for Main
    static void Main()
    {
    }
}
```

请参阅

- [C# 编程指南](#)
- [建议的文档注释标记](#)

<see> (C# 编程指南)

2021/5/7 • • [Edit Online](#)

语法

```
/// <see cref="member"/>
// or
/// <see href="link">Link Text</see>
// or
/// <see langword="keyword"/>
```

参数

- `cref="member"`

对可从当前编译环境调用的成员或字段的引用。编译器检查是否存在给定的码位元素，并将 `member` 传递到输出 XML 中的元素名称。将成员置于双引号 (" ") 内。

- `href="link"`

指向给定 URL 的可单击链接。例如，`<see href="https://github.com">GitHub</see>` 生成一个可单击的链接，其中包含文本 GitHub，该文本链接到 `https://github.com`。

- `langword="keyword"`

语言关键字，如 `true`。

备注

`<see>` 标记可用于从文本内指定链接。使用 `<seealso>` 指示文本应该放在“另请参阅”部分中。使用 `eref 属性` 创建指向代码元素的文档页的内部超链接。此外，`href` 还是一个有效属性，将用作超链接。

使用 [DocumentationFile](#) 进行编译可以将文档注释处理到文件中。

下面的示例展示摘要部分中的 `<see>` 标记。

```
// compile with: -doc:DocFileName.xml

/// text for class TestClass
public class TestClass
{
    /// <summary>DoWork is a method in the TestClass class.
    /// <para>Here's how you could make a second paragraph in a description. <see
    cref="System.Console.WriteLine(System.String)"/> for information about output statements.</para>
    /// <seealso cref="TestClass.Main"/>
    /// </summary>
    public static void DoWork(int Int1)
    {
    }

    /// text for Main
    static void Main()
    {
    }
}
```

请参阅

- [C# 编程指南](#)
- [建议的文档注释标记](#)

<seealso> (C# 编程指南)

2021/5/7 • [Edit Online](#)

语法

```
/// <seealso cref="member"/>
// or
/// <seealso href="link">Link Text</seealso>
```

参数

- `cref="member"`

对可从当前编译环境调用的成员或字段的引用。编译器检查是否存在给定的码位元素，并将 `member` 传递到输出 XML 中的元素名称。`member` 必须在双引号 (" ") 内。

有关如何创建对泛型类型的 `cref` 引用的信息，请参阅 [cref 特性](#)。

- `href="link"`

指向给定 URL 的可单击链接。例如，`<seealso href="https://github.com">GitHub</seealso>` 生成一个可单击的链接，其中包含文本 GitHub，该文本链接到 `https://github.com`。

备注

使用 `<seealso>` 标记，可以指定想要在“另请参阅”部分中显示的文本。使用 `<see>` 从文本内指定链接。

使用 [DocumentationFile](#) 进行编译可以将文档注释处理到文件中。

示例

有关使用 `<seealso>` 的示例，请参阅 [<summary>](#)。

请参阅

- [C# 编程指南](#)
- [建议的文档注释标记](#)

<summary> (C# 编程指南)

2021/5/7 • [Edit Online](#)

语法

```
<summary>description</summary>
```

参数

- `description`

对象的摘要。

备注

`<summary>` 标记应当用于描述类型或类型成员。使用 `<remarks>` 可针对某个类型说明添加补充信息。使用 `cref 属性` 可启用文档工具(如 DocFX 和 Sandcastle)来创建指向代码元素的文档页的内部超链接。

`<summary>` 标记的文本是唯一有关 IntelliSense 中的类型的信息源, 它也显示在对象浏览器窗口中。

使用 `DocumentationFile` 进行编译可以将文档注释处理到文件中。若要基于编译器生成的文件创建最终文档, 可以创建一个自定义工具, 也可以使用 DocFX 或 Sandcastle 等工具。

示例

```
// compile with: -doc:DocFileName.xml

/// text for class TestClass
public class TestClass
{
    /// <summary>DoWork is a method in the TestClass class.
    /// <para>Here's how you could make a second paragraph in a description. <see
    cref="System.Console.WriteLine(System.String)">/> for information about output statements.</para>
    /// <seealso cref="TestClass.Main"/>
    /// </summary>
    public static void DoWork(int Int1)
    {
    }

    /// text for Main
    static void Main()
    {
    }
}
```

前面的示例生成下面的 XML 文件。

```

<?xml version="1.0"?>
<doc>
  <assembly>
    <name>YourNamespace</name>
  </assembly>
  <members>
    <member name="T:TestClass">
      text for class TestClass
    </member>
    <member name="M:TestClass.DoWork(System.Int32)">
      <summary>DoWork is a method in the TestClass class.
      <para>Here's how you could make a second paragraph in a description. <see
      cref="M:System.Console.WriteLine(System.String)"/> for information about output statements.</para>
    </summary>
    <seealso cref="M:TestClass.Main"/>
    </member>
    <member name="M:TestClass.Main">
      text for Main
    </member>
  </members>
</doc>

```

cref 示例

下面的示例演示如何对泛型类型进行 `cref` 引用。

```

// compile with: -doc:DocFileName.xml

// the following cref shows how to specify the reference, such that,
// the compiler will resolve the reference
/// <summary cref="C{T}">
/// </summary>
class A { }

// the following cref shows another way to specify the reference,
// such that, the compiler will resolve the reference
// <summary cref="C &lt; T &gt;">

// the following cref shows how to hard-code the reference
/// <summary cref="T:C`1">
/// </summary>
class B { }

/// <summary cref="A">
/// </summary>
/// <typeparam name="T"></typeparam>
class C<T> { }

class Program
{
    static void Main() { }
}

```

前面的示例生成下面的 XML 文件。

```
<?xml version="1.0"?>
<doc>
  <assembly>
    <name>CRefTest</name>
  </assembly>
  <members>
    <member name="T:A">
      <summary cref="T:C`1">
        </summary>
    </member>
    <member name="T:B">
      <summary cref="T:C`1">
        </summary>
    </member>
    <member name="T:C`1">
      <summary cref="T:A">
        </summary>
      <typeparam name="T"></typeparam>
    </member>
  </members>
</doc>
```

请参阅

- [C# 编程指南](#)
- [建议的文档注释标记](#)

<typeparam> (C# 编程指南)

2021/5/7 • [Edit Online](#)

语法

```
<typeparam name="name">description</typeparam>
```

parameters

- `name`

类型参数的名称。用双引号 (" ") 将名称引起来。

- `description`

类型参数的说明。

备注

在泛型类型或方法声明的注释中，应使用 `<typeparam>` 标记来描述类型参数。为泛型类型或方法的每个类型参数添加标记。

有关详细信息，请参阅[泛型](#)。

`<typeparam>` 标记的文本将显示在 IntelliSense、[对象浏览器窗口](#)代码注释 Web 报表。

使用[DocumentationFile](#) 进行编译可以将文档注释处理到文件中。

示例

```
// compile with: -doc:DocFileName.xml

/// comment for class
public class TestClass
{
    /// <summary>
    /// Creates a new array of arbitrary type <typeparamref name="T"/>
    /// </summary>
    /// <typeparam name="T">The element type of the array</typeparam>
    public static T[] mkArray<T>(int n)
    {
        return new T[n];
    }
}
```

另请参阅

- [C# 参考](#)
- [C# 编程指南](#)
- [建议的文档注释标记](#)

<typeparamref> (C# 编程指南)

2021/5/7 • [Edit Online](#)

语法

```
<typeparamref name="name"/>
```

parameters

- `name`

类型参数的名称。用双引号 (" ") 将名称引起来。

备注

有关泛型类型中的类型参数及方法的详细信息，请参阅[泛型](#)。

通过此标记，文档文件的使用者可显著设置字体格式，例如采用斜体。

使用 [DocumentationFile](#) 进行编译可以将文档注释处理到文件中。

示例

```
// compile with: -doc:DocFileName.xml

/// comment for class
public class TestClass
{
    /// <summary>
    /// Creates a new array of arbitrary type <typeparamref name="T"/>
    /// </summary>
    /// <typeparam name="T">The element type of the array</typeparam>
    public static T[] mkArray<T>(int n)
    {
        return new T[n];
    }
}
```

请参阅

- [C# 编程指南](#)
- [建议的文档注释标记](#)

<value> (C# 编程指南)

2021/5/7 • [Edit Online](#)

语法

```
<value>property-description</value>
```

参数

- `property-description`

属性的说明。

备注

`<value>` 标记可用于描述属性表示的值。在 Visual Studio .NET 开发环境中通过代码向导添加属性时，将添加新属性的 `<summary>` 标记。然后，应手动添加 `<value>` 标记，以描述属性表示的值。

使用 [DocumentationFile](#) 进行编译可以将文档注释处理到文件中。

示例

```
// compile with: -doc:DocFileName.xml

/// text for class Employee
public class Employee
{
    private string _name;

    /// <summary>The Name property represents the employee's name.</summary>
    /// <value>The Name property gets/sets the value of the string field, _name.</value>

    public string Name
    {
        get
        {
            return _name;
        }
        set
        {
            _name = value;
        }
    }
}

/// text for class MainClass
public class MainClass
{
    /// text for Main
    static void Main()
    {
    }
}
```

请参阅

- [C# 编程指南](#)
- [建议的文档注释标记](#)

异常和异常处理 (C# 编程指南)

2021/5/7 • [Edit Online](#)

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。

```
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。
- 即使有异常抛出，`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](#)

使用异常 (C# 编程指南)

2021/3/5 • [Edit Online](#)

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：

```
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：

```
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。Catch 块应始终按从最具有针对性（或派生程度最高）到最不具有针对性的顺序排列。例如：

```
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：

```
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(0xFF);
    }
    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)

2021/3/5 • [Edit Online](#)

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。

```
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.
}
```

```
try
{
    // Code to try goes here.
}
finally
{
    // Code to execute after the try block goes here.
}
```

```
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。

```
int GetInt(int[] array, int index)
{
    try
    {
        return array[index];
    }
    catch (IndexOutOfRangeException e)
    {
        throw new ArgumentException(
            "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额外处理。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。

```
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。这表明仅当条件为 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：

```
int GetInt(int[] array, int index)
{
    try
    {
        return array[index];
    }
    catch (IndexOutOfRangeException e) when (index < 0)
    {
        throw new ArgumentException(
            "Parameter index cannot be negative.", e);
    }
    catch (IndexOutOfRangeException e)
    {
        throw new ArgumentException(
            "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：

```

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。有关详细信息，请参阅 [using 语句](#)。

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。

```

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。

请参阅

- [C# 参考](#)
- [try-catch](#)
- [try-finally](#)

- `try-catch-finally`
- `using` 语句

创建和引发异常 (C# 编程指南)

2021/3/5 • • [Edit Online](#)

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` 关键字引发。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：

```
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：

```
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：

```
static int GetValueFromArray(int[] array, int index)
{
    try
    {
        return array[index];
    }
    catch (IndexOutOfRangeException ex)
    {
        throw new ArgumentException("Index is out of range", nameof(index), ex);
    }
}
```

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。

定义异常类

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四个构造函数：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。第四个构造函数用于序列化异常。新的异常类应可序列化。例如：

```
[Serializable]
public class InvalidDepartmentException : Exception
{
    public InvalidDepartmentException() : base() { }
    public InvalidDepartmentException(string message) : base(message) { }
    public InvalidDepartmentException(string message, Exception inner) : base(message, inner) { }

    // A constructor is needed for serialization when an
    // exception propagates from a remoting server to the client.
    protected InvalidDepartmentException(System.Runtime.Serialization.SerializationInfo info,
        System.Runtime.Serialization.StreamingContext context) : base(info, context) { }
}
```

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。

请参阅

- [异常层次结构](#)

编译器生成的异常 (C# 编程指南)

2021/3/5 • [Edit Online](#)

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。

异常	错误条件
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 <code>null</code> 的对象时引发。
OutOfMemoryException	尝试使用 new 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	<code>checked</code> 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 <code>catch</code> 子句来捕获异常时引发。

请参阅

- [try-catch](#)
- [try-finally](#)
- [try-catch-finally](#)

如何使用 try/catch 处理异常 (C# 编程指南)

2021/5/7 • [Edit Online](#)

`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。

```
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` 异常并改为引发更合理的 `ArgumentOutOfRangeException` 异常。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 (C# 编程指南)

2021/5/7 • [Edit Online](#)

`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:

```
static void CodeWithoutCleanup()
{
    FileStream? file = null;
    FileInfo fileInfo = new FileInfo("./file.txt");

    file = fileInfo.OpenWrite();
    file.WriteByte(0xFF);

    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。

```
static void CodeWithCleanup()
{
    FileStream? file = null;
    FileInfo? fileInfo = null;

    try
    {
        fileInfo = new FileInfo("./file.txt");

        file = fileInfo.OpenWrite();
        file.WriteByte(0xFF);
    }
    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
- `try-catch`
- `try-finally`
- `try-catch-finally`

如何捕捉非 CLS 异常

2021/5/7 • • [Edit Online](#)

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。

```
// 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](#)

文件系统和注册表 (C# 编程指南)

2020/11/2 • [Edit Online](#)

下面各文章介绍了如何使用 C# 和 .NET 对文件、文件夹和注册表执行各种基本操作。

本节内容

文章	摘要
如何循环访问目录树	介绍了如何手动循环访问目录树。
如何获取有关文件、文件夹和驱动器的信息	介绍了如何检索有关文件、文件夹和驱动器的信息，如创建时间和大小。
如何创建文件或文件夹	介绍了如何新建文件或文件夹。
如何复制、删除和移动文件和文件夹(C# 编程指南)	介绍了如何复制、删除、移动文件和文件夹。
如何提供文件操作进度对话框	介绍了如何显示特定文件操作的标准 Windows 进度对话框。
如何写入文本文件	介绍了如何将内容写入文本文件。
如何读取文本文件中的内容	介绍了如何读取文本文件中的内容。
如何一次一行地读取文本文件	介绍了如何一次一行地检索文件文本。
如何在注册表中创建注册表项	介绍了如何将注册表项写入系统注册表。

相关章节

- [文件和流 I/O](#)
- [如何复制、删除和移动文件和文件夹\(C# 编程指南\)](#)
- [C# 编程指南](#)
- [System.IO](#)

如何循环访问目录树 (C# 编程指南)

2021/5/7 • [Edit Online](#)

短语“循环访问目录树”的意思是访问特定根文件夹下的每个嵌套子目录中的每个文件，可以是任意深度。不需要打开每个文件。可以以 `string` 的形式只检索文件或子目录的名称，也可以以 `System.IO.FileInfo` 或 `System.IO.DirectoryInfo` 对象的形式检索其他信息。

NOTE

在 Windows 中，术语“目录”和“文件夹”可以互换使用。大多数文档和用户界面文本使用术语“文件夹”，但 .NET 类库使用术语“目录”。

在最简单的情况下，如果你确信拥有指定根目录下的所有目录的访问权限，则可以使用 `System.IO.SearchOption.AllDirectories` 标志。此标志返回与指定的模式匹配的所有嵌套的子目录。下面的示例演示如何使用此标志。

```
root.GetDirectories("*.*", System.IO.SearchOption.AllDirectories);
```

此方法的缺点是，如果指定根目录下的任何子目录引发 `DirectoryNotFoundException` 或 `UnauthorizedAccessException` 异常，则整个方法失败且不返回任何目录。使用 `GetFiles` 方法时也是如此。如果需要处理特定子文件夹中的异常，则必须手动遍历目录树，如以下示例所示。

手动遍历目录树时，可以先处理子目录（前序遍历），或者先处理文件（后序遍历）。如果执行前序遍历，那么在遍历直接位于当前文件夹中的文件之前，先遍历此文件夹下的整棵树。本文档后面的示例执行的是后序遍历，但你可以轻松地修改它们以执行前序遍历。

另一种选择是，是使用递归遍历还是基于堆栈的遍历。本文档后面的示例演示了这两种方法。

如果需要对文件和文件夹执行各种操作，则可以模块化这些示例，方法是将操作重构为可使用单个委托进行调用的单独的函数。

NOTE

NTFS 文件系统可以包含交接点、符号链接和硬链接等形式的重解析点。诸如 `GetFiles` 和 `GetDirectories` 等 .NET 方法不会返回重分析点下的任何子目录。当两个重解析点相互引用时，此行为可防止进入无限循环。通常，处理重解析点时应格外小心，以确保不会无意中修改或删除文件。如果需要精确控制重解析点，请使用平台调用或本机代码直接调用相应的 Win32 文件系统方法。

示例

下面的示例演示如何以递归方式遍历目录树。递归方法是一种很好的方法，但是如果目录树较大且嵌套深度较深，则可能引起堆栈溢出异常。

在每个文件或文件夹上处理的特定异常和执行的特定操作仅作为示例提供。你可以修改此代码来满足你的特定要求。有关详细信息，请参阅代码中的注释。

```
public class RecursiveFileSearch
{
    static System.Collections.Specialized.StringCollection log = new
    System.Collections.Specialized.StringCollection();
```

```

static void Main()
{
    // Start with drives if you have to search the entire computer.
    string[] drives = System.Environment.GetLogicalDrives();

    foreach (string dr in drives)
    {
        System.IO.DriveInfo di = new System.IO.DriveInfo(dr);

        // Here we skip the drive if it is not ready to be read. This
        // is not necessarily the appropriate action in all scenarios.
        if (!di.IsReady)
        {
            Console.WriteLine("The drive {0} could not be read", di.Name);
            continue;
        }
        System.IO.DirectoryInfo rootDir = di.RootDirectory;
        WalkDirectoryTree(rootDir);
    }

    // Write out all the files that could not be processed.
    Console.WriteLine("Files with restricted access:");
    foreach (string s in log)
    {
        Console.WriteLine(s);
    }
    // Keep the console window open in debug mode.
    Console.WriteLine("Press any key");
    Console.ReadKey();
}

static void WalkDirectoryTree(System.IO.DirectoryInfo root)
{
    System.IO.FileInfo[] files = null;
    System.IO.DirectoryInfo[] subDirs = null;

    // First, process all the files directly under this folder
    try
    {
        files = root.GetFiles("*.*");
    }
    // This is thrown if even one of the files requires permissions greater
    // than the application provides.
    catch (UnauthorizedAccessException e)
    {
        // This code just writes out the message and continues to recurse.
        // You may decide to do something different here. For example, you
        // can try to elevate your privileges and access the file again.
        log.Add(e.Message);
    }

    catch (System.IO.DirectoryNotFoundException e)
    {
        Console.WriteLine(e.Message);
    }

    if (files != null)
    {
        foreach (System.IO.FileInfo fi in files)
        {
            // In this example, we only access the existing FileInfo object. If we
            // want to open, delete or modify the file, then
            // a try-catch block is required here to handle the case
            // where the file has been deleted since the call to TraverseTree().
            Console.WriteLine(fi.FullName);
        }
    }

    // Now find all the subdirectories under this directory.
    subDirs = root.GetDirectories();
}

```

```
        subDirs = root.GetDirectories("\\*\\"),
        foreach (System.IO DirectoryInfo dirInfo in subDirs)
    {
        // Recursive call for each subdirectory.
        WalkDirectoryTree(dirInfo);
    }
}
}
```

下面的示例演示如何不使用递归方式遍历目录树中的文件和文件夹。此方法使用泛型 `Stack<T>` 集合类型，此集合类型是一个后进先出 (LIFO) 堆栈。

在每个文件或文件夹上处理的特定异常和执行的特定操作仅作为示例提供。你可以修改此代码来满足你的特定要求。有关详细信息，请参阅代码中的注释。

```
public class StackBasedIteration
{
    static void Main(string[] args)
    {
        // Specify the starting folder on the command line, or in
        // Visual Studio in the Project > Properties > Debug pane.
        TraverseTree(args[0]);

        Console.WriteLine("Press any key");
        Console.ReadKey();
    }

    public static void TraverseTree(string root)
    {
        // Data structure to hold names of subfolders to be
        // examined for files.
        Stack<string> dirs = new Stack<string>(20);

        if (!System.IO.Directory.Exists(root))
        {
            throw new ArgumentException();
        }
        dirs.Push(root);

        while (dirs.Count > 0)
        {
            string currentDir = dirs.Pop();
            string[] subDirs;
            try
            {
                subDirs = System.IO.Directory.GetDirectories(currentDir);
            }
            // An UnauthorizedAccessException exception will be thrown if we do not have
            // discovery permission on a folder or file. It may or may not be acceptable
            // to ignore the exception and continue enumerating the remaining files and
            // folders. It is also possible (but unlikely) that a DirectoryNotFoundException
            // will be raised. This will happen if currentDir has been deleted by
            // another application or thread after our call to Directory.Exists. The
            // choice of which exceptions to catch depends entirely on the specific task
            // you are intending to perform and also on how much you know with certainty
            // about the systems on which this code will run.
            catch (UnauthorizedAccessException e)
            {
                Console.WriteLine(e.Message);
                continue;
            }
            catch (System.IO.DirectoryNotFoundException e)
            {
                Console.WriteLine(e.Message);
                continue;
            }
        }
    }
}
```

```

    }

    string[] files = null;
    try
    {
        files = System.IO.Directory.GetFiles(currentDir);
    }

    catch (UnauthorizedAccessException e)
    {

        Console.WriteLine(e.Message);
        continue;
    }

    catch (System.IO.DirectoryNotFoundException e)
    {
        Console.WriteLine(e.Message);
        continue;
    }
    // Perform the required action on each file here.
    // Modify this block to perform your required task.
    foreach (string file in files)
    {
        try
        {
            // Perform whatever action is required in your scenario.
            System.IO.FileInfo fi = new System.IO.FileInfo(file);
            Console.WriteLine("{0}: {1}, {2}", fi.Name, fi.Length, fi.CreationTime);
        }
        catch (System.IO.FileNotFoundException e)
        {
            // If file was deleted by a separate application
            // or thread since the call to TraverseTree()
            // then just continue.
            Console.WriteLine(e.Message);
            continue;
        }
    }

    // Push the subdirectories onto the stack for traversal.
    // This could also be done before handing the files.
    foreach (string str in subDirs)
        dirs.Push(str);
    }
}
}
}

```

通常，检测每个文件夹以确定应用程序是否有权限打开它是一个很费时的过程。因此，此代码示例只将此部分操作封装在 `try/catch` 块中。你可以修改 `catch` 块，以便在拒绝访问某个文件夹时，可以尝试提升权限，然后再次访问此文件夹。一般来说，仅捕获可以处理的、不会将应用程序置于未知状态的异常。

如果必须在内存或磁盘上存储目录树的内容，那么最佳选择是仅存储每个文件的 `FullName` 属性（类型为 `string`）。然后可以根据需要使用此字符串创建新的 `FileInfo` 或 `DirectoryInfo` 对象，或打开需要进行其他处理的任何文件。

可靠编程

可靠的文件迭代代码必须考虑文件系统的诸多复杂性。有关 Windows 文件系统的详细信息，请参阅 [NTFS 概述](#)。

请参阅

- [System.IO](#)

- [LINQ 和文件目录](#)
- [文件系统和注册表\(C# 编程指南\)](#)

如何获取有关文件、文件夹和驱动器的信息 (C# 编程指南)

2021/5/7 • [Edit Online](#)

在 .NET 中，可以使用以下类访问文件系统信息：

- [System.IO.FileInfo](#)
- [System.IO.DirectoryInfo](#)
- [System.IO.DriveInfo](#)
- [System.IO.Directory](#)
- [System.IO.File](#)

[FileInfo](#) 和 [DirectoryInfo](#) 类表示文件或目录，并包含用于公开 NTFS 文件系统所支持的许多文件特性的属性。它们还包含用于打开、关闭、移动和删除文件和文件夹的方法。可以通过将表示文件、文件夹或驱动器名称的字符串传入构造函数来创建这些类的实例：

```
System.IO.DriveInfo di = new System.IO.DriveInfo(@"C:\");
```

还可以使用对 [DirectoryInfo.GetDirectories](#) [DirectoryInfo.GetFiles](#) 和 [DriveInfo.RootDirectory](#) 的调用来获取文件、文件夹或驱动器的名称。

[System.IO.Directory](#) 和 [System.IO.File](#) 类提供相关静态方法，用于检索有关目录和文件的信息。

示例

下面的示例演示用于访问有关文件和文件夹的信息的各种方法。

```
class FileSysInfo
{
    static void Main()
    {
        // You can also use System.Environment.GetLogicalDrives to
        // obtain names of all logical drives on the computer.
        System.IO.DriveInfo di = new System.IO.DriveInfo(@"C:\");
        Console.WriteLine(di.TotalFreeSpace);
        Console.WriteLine(di.VolumeLabel);

        // Get the root directory and print out some information about it.
        System.IO.DirectoryInfo dirInfo = di.RootDirectory;
        Console.WriteLine(dirInfo.Attributes.ToString());

        // Get the files in the directory and print out some information about them.
        System.IO.FileInfo[] fileNames = dirInfo.GetFiles("*.*");

        foreach (System.IO.FileInfo fi in fileNames)
        {
            Console.WriteLine("{0}: {1}: {2}", fi.Name, fi.LastAccessTime, fi.Length);
        }

        // Get the subdirectories directly that is under the root.
        // See "How to: Iterate Through a Directory Tree" for an example of how to
        // iterate through an entire tree.
        System.IO.DirectoryInfo[] dirInfos = dirInfo.GetDirectories("* *").
```

```

System.IO.DirectoryInfo[] dirInfos = DirectoryInfo.GetDirectories("C:\\");

foreach (System.IO.DirectoryInfo d in dirInfos)
{
    Console.WriteLine(d.Name);
}

// The Directory and File classes provide several static methods
// for accessing files and directories.

// Get the current application directory.
string currentDirName = System.IO.Directory.GetCurrentDirectory();
Console.WriteLine(currentDirName);

// Get an array of file names as strings rather than FileInfo objects.
// Use this method when storage space is an issue, and when you might
// hold on to the file name reference for a while before you try to access
// the file.
string[] files = System.IO.Directory.GetFiles(currentDirName, "*.txt");

foreach (string s in files)
{
    // Create the FileInfo object only when needed to ensure
    // the information is as current as possible.
    System.IO.FileInfo fi = null;
    try
    {
        fi = new System.IO.FileInfo(s);
    }
    catch (System.IO.FileNotFoundException e)
    {
        // To inform the user and continue is
        // sufficient for this demonstration.
        // Your application may require different behavior.
        Console.WriteLine(e.Message);
        continue;
    }
    Console.WriteLine("{0} : {1}", fi.Name, fi.Directory);
}

// Change the directory. In this case, first check to see
// whether it already exists, and create it if it does not.
// If this is not appropriate for your application, you can
// handle the System.IO.IOException that will be raised if the
// directory cannot be found.
if (!System.IO.Directory.Exists(@"C:\Users\Public\TestFolder\")) {
    System.IO.Directory.CreateDirectory(@"C:\Users\Public\TestFolder\");

    System.IO.Directory.SetCurrentDirectory(@"C:\Users\Public\TestFolder\");

    currentDirName = System.IO.Directory.GetCurrentDirectory();
    Console.WriteLine(currentDirName);

    // Keep the console window open in debug mode.
    Console.WriteLine("Press any key to exit.");
    Console.ReadKey();
}
}

```

可靠编程

处理用户指定的路径字符串时，还应针对以下情况处理异常：

- 文件名格式不正确。例如，它包含无效字符或只包含空格。

- 文件名为 null。
- 文件名超过系统定义的最大长度。
- 文件名包含冒号 (:)。

如果应用程序没有足够权限来读取指定文件, `Exists` 方法会返回 `false` (无论路径是否存在); 该方法不会引发异常。

请参阅

- [System.IO](#)
- [C# 编程指南](#)
- [文件系统和注册表\(C# 编程指南\)](#)

如何创建文件或文件夹 (C# 编程指南)

2021/5/7 • [Edit Online](#)

可通过编程方式在计算机上创建文件夹、子文件夹和子文件夹中的文件，并将数据写入文件。

示例

```
public class CreateFileOrFolder
{
    static void Main()
    {
        // Specify a name for your top-level folder.
        string folderName = @"c:\Top-Level Folder";

        // To create a string that specifies the path to a subfolder under your
        // top-level folder, add a name for the subfolder to folderName.
        string pathString = System.IO.Path.Combine(folderName, "SubFolder");

        // You can write out the path name directly instead of using the Combine
        // method. Combine just makes the process easier.
        string pathString2 = @"c:\Top-Level Folder\SubFolder2";

        // You can extend the depth of your path if you want to.
        //pathString = System.IO.Path.Combine(pathString, "SubSubFolder");

        // Create the subfolder. You can verify in File Explorer that you have this
        // structure in the C: drive.
        //    Local Disk (C:)
        //        Top-Level Folder
        //            SubFolder
        System.IO.Directory.CreateDirectory(pathString);

        // Create a file name for the file you want to create.
        string fileName = System.IO.Path.GetRandomFileName();

        // This example uses a random string for the name, but you also can specify
        // a particular name.
        //string fileName = "MyNewFile.txt";

        // Use Combine again to add the file name to the path.
        pathString = System.IO.Path.Combine(pathString, fileName);

        // Verify the path that you have constructed.
        Console.WriteLine("Path to my file: {0}\n", pathString);

        // Check that the file doesn't already exist. If it doesn't exist, create
        // the file and write integers 0 - 99 to it.
        // DANGER: System.IO.File.Create will overwrite the file if it already exists.
        // This could happen even with random file names, although it is unlikely.
        if (!System.IO.File.Exists(pathString))
        {
            using (System.IO.FileStream fs = System.IO.File.Create(pathString))
            {
                for (byte i = 0; i < 100; i++)
                {
                    fs.WriteByte(i);
                }
            }
        }
        else
        {
```

```

        Console.WriteLine("File \'{0}\' already exists.", fileName);
        return;
    }

    // Read and display the data from your file.
    try
    {
        byte[] readBuffer = System.IO.File.ReadAllBytes(pathString);
        foreach (byte b in readBuffer)
        {
            Console.Write(b + " ");
        }
        Console.WriteLine();
    }
    catch (System.IO.IOException e)
    {
        Console.WriteLine(e.Message);
    }

    // Keep the console window open in debug mode.
    System.Console.WriteLine("Press any key to exit.");
    System.Console.ReadKey();
}
// Sample output:

// Path to my file: c:\Top-Level Folder\SubFolder\ttxvause.vv0

//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 8
//3 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99
}

```

如果文件夹已存在, [.CreateDirectory](#) 不执行任何操作, 未引发任何异常。但 [File.Create](#) 用新文件替换现有文件。本示例使用 `if - else` 语句阻止替换现有文件。

通过在示例中作出以下更改, 可根据具有特定名称的文件是否存在来指定不同的结果。如果该文件不存在, 代码就会创建一个文件。如果该文件存在, 代码就会将数据追加到该文件中。

- 指定一个非随机文件名。

```

// Comment out the following line.
//string fileName = System.IO.Path.GetRandomFileName();

// Replace that line with the following assignment.
string fileName = "MyNewFile.txt";

```

- 用以下代码中的 `using` 语句替换 `if - else` 语句。

```

using (System.IO.FileStream fs = new System.IO.FileStream(pathString, FileMode.Append))
{
    for (byte i = 0; i < 100; i++)
    {
        fs.WriteByte(i);
    }
}

```

多次运行此示例, 验证数据是否每次都添加到了文件中。

有关可尝试的 `FileMode` 值, 请参阅 [FileMode](#)。

以下情况可能会导致异常:

- 文件夹名称格式不正确。例如，它包含非法字符或它仅为空格([ArgumentException](#)类)。使用[Path](#)类创建有效的路径名。
- 要创建的文件夹的父文件夹为只读([IOException](#)类)。
- 文件夹名为 `null` ([ArgumentNullException](#)类)。
- 文件夹名过长([PathTooLongException](#)类)。
- 文件夹仅为冒号":"([PathTooLongException](#)类)。

.NET 安全性

可能在部分信任场景中引发 [SecurityException](#) 类的实例。

如果没有创建文件夹的权限，则本示例引发 [UnauthorizedAccessException](#) 类的实例。

请参阅

- [System.IO](#)
- [C# 编程指南](#)
- [文件系统和注册表\(C# 编程指南\)](#)

如何复制、删除和移动文件和文件夹 (C# 编程指南)

2021/5/7 • [Edit Online](#)

以下示例演示如何从 `System.IO` 命名空间使用 `System.IO.File`、`System.IO.Directory`、`System.IO.FileInfo` 和 `System.IO.DirectoryInfo` 类以同步方式复制、移动和删除文件与文件夹。这些示例未提供进度栏或其他任何用户界面。如果希望提供一个标准进度对话框，请参阅[如何提供文件操作进度对话框](#)。

操作多个文件时，请使用 `System.IO.FileSystemWatcher` 提供事件，以便可以计算进度。另一种方法是使用平台调用来调用 Windows Shell 中相应的文件相关方法。有关如何异步执行这些文件操作的信息，请参阅[异步文件 I/O](#)。

示例

以下示例演示如何复制文件和目录。

```
// Simple synchronous file copy operations with no user interface.  
// To run this sample, first create the following directories and files:  
// C:\Users\Public\TestFolder  
// C:\Users\Public\TestFolder\test.txt  
// C:\Users\Public\TestFolder\SubDir\test.txt  
public class SimpleFileCopy  
{  
    static void Main()  
    {  
        string fileName = "test.txt";  
        string sourcePath = @"C:\Users\Public\TestFolder";  
        string targetPath = @"C:\Users\Public\TestFolder\SubDir";  
  
        // Use Path class to manipulate file and directory paths.  
        string sourceFile = System.IO.Path.Combine(sourcePath, fileName);  
        string destFile = System.IO.Path.Combine(targetPath, fileName);  
  
        // To copy a folder's contents to a new location:  
        // Create a new target folder.  
        // If the directory already exists, this method does not create a new directory.  
        System.IO.Directory.CreateDirectory(targetPath);  
  
        // To copy a file to another location and  
        // overwrite the destination file if it already exists.  
        System.IO.File.Copy(sourceFile, destFile, true);  
  
        // To copy all the files in one directory to another directory.  
        // Get the files in the source folder. (To recursively iterate through  
        // all subfolders under the current directory, see  
        // "How to: Iterate Through a Directory Tree.")  
        // Note: Check for target path was performed previously  
        //       in this code example.  
        if (System.IO.Directory.Exists(sourcePath))  
        {  
            string[] files = System.IO.Directory.GetFiles(sourcePath);  
  
            // Copy the files and overwrite destination files if they already exist.  
            foreach (string s in files)  
            {  
                // Use static Path methods to extract only the file name from the path.  
                fileName = System.IO.Path.GetFileName(s);  
                destFile = System.IO.Path.Combine(targetPath, fileName);  
                System.IO.File.Copy(s, destFile, true);  
            }  
        }  
        else  
        {  
            Console.WriteLine("Source path does not exist!");  
        }  
  
        // Keep console window open in debug mode.  
        Console.WriteLine("Press any key to exit.");  
        Console.ReadKey();  
    }  
}
```

以下示例演示如何移动文件和目录。

```

// Simple synchronous file move operations with no user interface.
public class SimpleFileMove
{
    static void Main()
    {
        string sourceFile = @"C:\Users\Public\public\test.txt";
        string destinationFile = @"C:\Users\Public\private\test.txt";

        // To move a file or folder to a new location:
        System.IO.File.Move(sourceFile, destinationFile);

        // To move an entire directory. To programmatically modify or combine
        // path strings, use the System.IO.Path class.
        System.IO.Directory.Move(@"C:\Users\Public\public\test\", @"C:\Users\Public\private");
    }
}

```

以下示例演示如何删除文件和目录。

```

// Simple synchronous file deletion operations with no user interface.
// To run this sample, create the following files on your drive:
// C:\Users\Public\DeleteTest\test1.txt
// C:\Users\Public\DeleteTest\test2.txt
// C:\Users\Public\DeleteTest\SubDir\test2.txt

public class SimpleFileDelete
{
    static void Main()
    {
        // Delete a file by using File class static method...
        if(System.IO.File.Exists(@"C:\Users\Public\DeleteTest\test.txt"))
        {
            // Use a try block to catch IOExceptions, to
            // handle the case of the file already being
            // opened by another process.
            try
            {
                System.IO.File.Delete(@"C:\Users\Public\DeleteTest\test.txt");
            }
            catch (System.IO.IOException e)
            {
                Console.WriteLine(e.Message);
                return;
            }
        }

        // ...or by using FileInfo instance method.
        System.IO.FileInfo fi = new System.IO.FileInfo(@"C:\Users\Public\DeleteTest\test2.txt");
        try
        {
            fi.Delete();
        }
        catch (System.IO.IOException e)
        {
            Console.WriteLine(e.Message);
        }

        // Delete a directory. Must be writable or empty.
        try
        {
            System.IO.Directory.Delete(@"C:\Users\Public\DeleteTest");
        }
        catch (System.IO.IOException e)
        {
            Console.WriteLine(e.Message);
        }
    }
}

```

```
// Delete a directory and all subdirectories with Directory static method...
if(System.IO.Directory.Exists(@"C:\Users\Public\DeleteTest"))
{
    try
    {
        System.IO.Directory.Delete(@"C:\Users\Public\DeleteTest", true);
    }

    catch (System.IO.IOException e)
    {
        Console.WriteLine(e.Message);
    }
}

// ...or with DirectoryInfo instance method.
System.IO.DirectoryInfo di = new System.IO.DirectoryInfo(@"C:\Users\Public\public");
// Delete this dir and all subdirs.
try
{
    di.Delete(true);
}
catch (System.IO.IOException e)
{
    Console.WriteLine(e.Message);
}
}
```

另请参阅

- [System.IO](#)
- [C# 编程指南](#)
- [文件系统和注册表\(C# 编程指南\)](#)
- [如何提供文件操作进度对话框](#)
- [文件和流 I/O](#)
- [通用 I/O 任务](#)

如何提供文件操作进度对话框 (C# 编程指南)

2021/5/7 • [Edit Online](#)

如果在 `Microsoft.VisualBasic` 命名空间中使用 `CopyFile(String, String, UIOption)` 方法，可以在 Windows 中提供显示文件操作进度的标准对话框。

NOTE

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](#)。

在 Visual Studio 中添加引用

1. 在菜单栏上，依次选择“项目”、“添加引用”。

此时将显示“引用管理器”对话框。

2. 在“程序集”区域，选择“Framework”(如果尚未选择它)。

3. 在名称列表中，选择“`Microsoft.VisualBasic`”复选框，然后再选择“确定”按钮以关闭对话框。

示例

以下代码将 `sourcePath` 指定的目录复制到 `destinationPath` 指定的目录。此代码还提供了标准的对话框，其中显示在操作完成前估计的剩余时间量。

```
// The following using directive requires a project reference to Microsoft.VisualBasic.
using Microsoft.VisualBasic.FileIO;

class FileProgress
{
    static void Main()
    {
        // Specify the path to a folder that you want to copy. If the folder is small,
        // you won't have time to see the progress dialog box.
        string sourcePath = @"C:\Windows\symbols\";
        // Choose a destination for the copied files.
        string destinationPath = @"C:\TestFolder";

        FileSystem.CopyDirectory(sourcePath, destinationPath,
            UIOption.AllDialogs);
    }
}
```

请参阅

- [文件系统和注册表\(C# 编程指南\)](#)

如何写入文本文件 (C# 编程指南)

2021/5/7 • [Edit Online](#)

本文提供了几个示例，演示了将文本写入文件的多种方法。前两个示例使用 [System.IO.File](#) 类上的静态便捷方法将任意 `IEnumerable<string>` 的每个元素和 `string` 写入文本文件。第三个示例演示了在写入文件时必须分别处理文本的每一行的情况下，如何将文本添加到文件。在前三个示例中，会覆盖文件中的所有现有内容。最后一个示例演示如何将文本追加到现有文件。

这些示例都将字符串文本写入了文件。如果想设置写入文件的文本的格式，请使用 [Format](#) 方法或 C# [字符串内插](#) 功能。

将字符串的集合写入文件

```
using System.IO;
using System.Threading.Tasks;

class WriteAllLines
{
    public static async Task ExampleAsync()
    {
        string[] lines =
        {
            "First line", "Second line", "Third line"
        };

        await File.WriteAllLinesAsync("WriteLines.txt", lines);
    }
}
```

前面的源代码示例：

- 使用三个值实例化字符串数组。
- 等待对 [File.WriteAllLinesAsync](#) 的调用完成，该调用会执行以下操作：
 - 以异步方式创建一个名为“WriteLines.txt”的文件。如果该文件已存在，则会覆盖该文件。
 - 将给定行写入该文件。
 - 关闭该文件，并根据需要自动刷新和释放。

向文件写入一个字符串

```

using System.IO;
using System.Threading.Tasks;

class WriteAllText
{
    public static async Task ExampleAsync()
    {
        string text =
            "A class is the most powerful data type in C#. Like a structure, " +
            "a class defines the data and behavior of the data type. ";

        await File.WriteAllTextAsync("WriteText.txt", text);
    }
}

```

前面的源代码示例：

- 根据指定的字符串字面量实例化一个字符串。
- 等待对 [File.WriteAllTextAsync](#) 的调用完成，该调用会执行以下操作：
 - 以异步方式创建一个名为“WriteText.txt”的文件。如果该文件已存在，则会覆盖该文件。
 - 将给定文本写入该文件。
 - 关闭该文件，并根据需要自动刷新和释放。

将数组中的选定字符串写入文件

```

using System.IO;
using System.Threading.Tasks;

class StreamWriterOne
{
    public static async Task ExampleAsync()
    {
        string[] lines = { "First line", "Second line", "Third line" };
        using StreamWriter file = new("WriteLines2.txt");

        foreach (string line in lines)
        {
            if (!line.Contains("Second"))
            {
                await file.WriteLineAsync(line);
            }
        }
    }
}

```

前面的源代码示例：

- 使用三个值实例化字符串数组。
- 以 [using 声明](#) 的形式使用 WriteLines2.txt 的文件路径实例化 [StreamWriter](#)。
- 循环访问所有行。
- 有条件地等待对 [StreamWriter.WriteLineAsync\(String\)](#) 的调用完成，该调用在行不包含 "Second" 时将该行写入文件。

将文本追加到现有文件

```
using System.IO;
using System.Threading.Tasks;

class StreamWriterTwo
{
    public static async Task ExampleAsync()
    {
        using StreamWriter file = new("WriteLines2.txt", append: true);
        await file.WriteLineAsync("Fourth line");
    }
}
```

前面的源代码示例：

- 使用三个值实例化字符串数组。
- 以 `using` 声明的形式使用 `WriteLines2.txt` 的文件路径实例化 `StreamWriter`, 并传入要追加的 `true`。
- 等待对 `StreamWriter.WriteLineAsync(String)` 的调用完成, 该调用将字符串作为追加行写入文件。

异常

以下情况可能会导致异常：

- `InvalidOperationException`: 文件已存在并且为只读。
- `PathTooLongException`: 路径名可能太长。
- `IOException`: 磁盘可能已满。

使用文件系统时, 还有其他可能会导致异常的情况, 因此最好进行防御性编程。

请参阅

- [C# 编程指南](#)
- [文件系统和注册表\(C# 编程指南\)](#)
- [示例: 将集合保存到应用程序存储](#)

如何读取文本文件中的内容 (C# 编程指南)

2021/5/7 • [Edit Online](#)

此示例通过使用 `System.IO.File` 类的 `ReadAllText` 和 `ReadAllLines` 静态方法来确定文本文件的内容。

有关使用 `StreamReader` 的示例，请参阅[如何一次一行地读取文本文件](#)。

NOTE

此示例所用的文件是在主题[如何写入文本文件](#)中创建的。

示例

```
class ReadFromFile
{
    static void Main()
    {
        // The files used in this example are created in the topic
        // How to: Write to a Text File. You can change the path and
        // file name to substitute text files of your own.

        // Example #1
        // Read the file as one string.
        string text = System.IO.File.ReadAllText(@"C:\Users\Public\TestFolder\WriteText.txt");

        // Display the file contents to the console. Variable text is a string.
        System.Console.WriteLine("Contents of WriteText.txt = {0}", text);

        // Example #2
        // Read each line of the file into a string array. Each element
        // of the array is one line of the file.
        string[] lines = System.IO.File.ReadAllLines(@"C:\Users\Public\TestFolder\WriteLines2.txt");

        // Display the file contents by using a foreach loop.
        System.Console.WriteLine("Contents of WriteLines2.txt = ");
        foreach (string line in lines)
        {
            // Use a tab to indent each line of the file.
            Console.WriteLine("\t" + line);
        }

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        System.Console.ReadKey();
    }
}
```

编译代码

将代码复制并粘贴到 C# 控制台应用程序。

如果不使用[如何写入文本文件](#)中的文本文件，请将 `ReadAllText` 和 `ReadAllLines` 的参数替换为计算机上的适当路径和文件名。

可靠编程

以下情况可能会导致异常：

- 不存在该文件，或者指定位置不存在该文件。检查文件名的路径和拼写。

.NET 安全性

不要依赖文件名来确定文件的内容。例如，文件 `myFile.cs` 可能不是 C# 源文件。

请参阅

- [System.IO](#)
- [C# 编程指南](#)
- [文件系统和注册表\(C# 编程指南\)](#)

如何一次一行地读取文本文件 (C# 编程指南)

2021/5/7 • [Edit Online](#)

此示例使用 `StreamReader` 类的 `ReadLine` 方法，一次一行地将文本文件内容读入字符串。每个文本行都存储到字符串 `line` 中并显示在屏幕上。

示例

```
int counter = 0;
string line;

// Read the file and display it line by line.
System.IO.StreamReader file =
    new System.IO.StreamReader(@"c:\test.txt");
while((line = file.ReadLine()) != null)
{
    System.Console.WriteLine(line);
    counter++;
}

file.Close();
System.Console.WriteLine("There were {0} lines.", counter);
// Suspend the screen.
System.Console.ReadLine();
```

编译代码

复制代码，并将其粘贴到控制台应用程序的 `Main` 方法中。

将 `"c:\test.txt"` 替换为实际文件名。

可靠编程

以下情况可能会导致异常：

- 文件可能不存在。

.NET 安全性

不要根据文件的名称来判断文件的内容。例如，文件 `myFile.cs` 可能不是 C# 源文件。

请参阅

- [System.IO](#)
- [C# 编程指南](#)
- [文件系统和注册表\(C# 编程指南\)](#)

如何在注册表中创建注册表项 (C# 编程指南)

2021/5/7 • [Edit Online](#)

本示例将值对“Name”和“Isabella”添加到当前用户注册表中的项“Names”之下。

示例

```
Microsoft.Win32.RegistryKey key;
key = Microsoft.Win32.Registry.CurrentUser.CreateSubKey("Names");
key.SetValue("Name", "Isabella");
key.Close();
```

编译代码

- 复制代码，并将其粘贴到控制台应用程序的 `Main` 方法中。
- 将 `Names` 参数替换为直接存在于注册表 `HKEY_CURRENT_USER` 节点下的项的名称。
- 将 `Name` 参数替换为直接存在于“Names”节点下的值的名称。

可靠编程

检查注册表结构，查找适合项的位置。例如，可能需要打开当前用户的 `Software` 项，并用公司的名称创建一项。然后将注册表值添加到公司的项上。

以下情况可能会导致异常：

- 项的名称为空。
- 用户没有创建注册表项的权限。
- 项名称超过 255 个字符的限制。
- 项已关闭。
- 注册表项为只读。

.NET 安全性

将数据写入用户文件夹 `Microsoft.Win32.Registry.CurrentUser` 比写入本地计算机 `Microsoft.Win32.Registry.LocalMachine` 更安全。

创建注册表值时，需要确定该值已存在时应执行的操作。另一进程（可能是恶意进程）可能已创建了该值，并拥有对该值的访问权。将数据放入注册表值后，其他进程即可使用这些数据。若要防止出现这种情况，请使用 `Overload:Microsoft.Win32.RegistryKey.GetValue` 方法。如果项不存在，则该方法返回 `null`。

即使注册表项受访问控制列表 (ACL) 保护，在注册表中以纯文本形式存储机密信息（例如密码）也不安全。

请参阅

- [System.IO](#)
- [C# 编程指南](#)
- [文件系统和注册表 \(C# 编程指南\)](#)

- Read, write and delete from the registry with C#(使用 C# 在注册表中执行读取、写入和删除操作)

互操作性 (C# 编程指南)

2020/11/2 • [Edit Online](#)

借助互操作性，可以保留和利用对非托管代码的现有投资工作。在公共语言运行时 (CLR) 控制下运行的代码称为 **托管代码**，不在 CLR 控制下运行的代码称为 **非托管代码**。例如，COM、COM+、C++ 组件、ActiveX 组件和 Microsoft Windows API 都是非托管代码。

借助 .NET，可通过平台调用服务、[System.Runtime.InteropServices](#) 命名空间、C++ 互操作性和 COM 互操作性 (COM 互操作) 实现与非托管代码的互操作性。

本节内容

[互操作性概述](#)

介绍了实现 C# 托管代码和非托管代码的互操作性的方法。

[如何使用 C# 功能访问 Office 互操作对象](#)

介绍了 Visual C# 为了推动 Office 编程而引入的功能。

[如何在 COM 互操作编程中使用索引属性](#)

介绍了如何使用已编入索引的属性来访问包含参数的 COM 属性。

[如何使用平台调用播放 WAV 文件](#)

介绍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。

[演练:Office 编程](#)

展示了如何创建 Excel 工作簿和包含指向此工作簿的链接的 Word 文档。

[COM 类示例](#)

展示了如何将 C# 类公开为 COM 对象。

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。

请参阅

- [Marshal.ReleaseComObject](#)
- [C# 编程指南](#)
- [与非托管代码交互操作](#)
- [演练:Office 编程](#)

互操作性概述 (C# 编程指南)

2020/11/2 • [Edit Online](#)

本主题描述在 C# 托管代码和非托管代码之间实现互操作性的方法。

平台调用

平台调用是一项服务，它使托管代码能够调用动态链接库 (DLL) 中实现的非托管函数，例如 Microsoft Windows API 中的非托管函数。此服务定位并调用导出的函数，并根据需要跨交互操作边界封送其自变量 (整数、字符串、数组、结构等)。

有关详细信息，请参阅[使用非托管 DLL 函数](#)和[如何使用平台调用播放 WAV 文件](#)。

NOTE

公共语言运行时 (CLR) 管理对系统资源的访问。调用 CLR 外部的非托管代码将避开这种安全机制，因此会带来安全风险。例如，非托管代码可能直接调用非托管代码中的资源，从而避开 CLR 安全机制。有关详细信息，请参阅[.NET 中的安全性](#)。

C++ 互操作

可使用 C++ interop(又称为 It Just Works (IJW)) 包装本机 C++ 类，以便用 C# 或其他 .NET 语言编写的代码可以使用此类。为此，请编写 C++ 代码来包装本机 DLL 或 COM 组件。与其他 .NET 语言不同，Visual C++ 具有互操作性支持，可使托管和非托管代码放置在同一个应用程序(甚至同一个文件)中。然后使用 /clr 编译器开关生成托管程序集，以便生成 C++ 代码。最后，在 C# 项目中添加一个对该程序集的引用，并像使用其他托管类那样使用被包装对象。

向 C# 公开 COM 组件

可以使用 C# 项目中的 COM 组件。常规步骤如下所示：

1. 找到要使用的 COM 组件并注册。使用 regsvr32.exe 注册或注销 COM DLL。
2. 向项目添加对 COM 组件或类型库的引用。

添加引用时，Visual Studio 使用 [Tlbimp.exe\(类型库导入程序\)](#)。此程序需要使用类型库作为输入，以输出 .NET 互操作程序集。该程序集又称为运行时可调用包装器 (RCW)，其中包含包装类型库中的 COM 类和接口的托管类和接口。Visual Studio 向项目添加对生成程序集的引用。

3. 创建在 RCW 中定义的类的实例。这会创建 COM 对象的实例。
4. 像使用其他托管对象那样使用该对象。当垃圾回收对该对象进行回收后，COM 对象的实例也会从内存中释放出来。

有关详细信息，请参阅[向 .NET Framework 公开 COM 组件](#)。

向 COM 公开 C#

COM 客户端可以使用已经正确公开的 C# 类型。公开 C# 类型的基本步骤如下所示：

1. 在 C# 项目中添加互操作特性。

可通过修改 Visual C# 项目属性使程序集 COM 可见。有关详细信息，请参阅[“程序集信息”对话框](#)。

2. 生成 COM 类型库并对它进行注册以供 COM 使用。

可修改 Visual C# 项目属性以自动注册 COM 互操作的 C# 程序集。Visual Studio 通过 `/t:library` 命令行开关使用 [Regasm.exe\(程序集注册工具\)](#)。此工具使用托管组件作为输入，以生成类型库。此类型库描述程序集中的 `public` 类型并添加注册表项，以便 COM 客户端可以创建托管类。

有关详细信息，请参阅[向 COM 公开 .NET Framework 组件](#)和[COM 类示例](#)。

请参阅

- [Improving Interop Performance\(提高互操作性能\)](#)
- [COM 和 .NET 之间的互操作性简介](#)
- [Visual Basic 中的 COM 互操作简介](#)
- [托管代码与非托管代码之间的封送处理](#)
- [与非托管代码交互操作](#)
- [C# 编程指南](#)

如何访问 Office 互操作对象 (C# 编程指南)

2021/5/7 • [Edit Online](#)

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。

在本主题中，你将利用这些新功能来编写创建并显示 Microsoft Office Excel 工作表的代码。然后，你将编写添加包含链接到 Excel 工作表的图标的 Office Word 文档的代码。

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。

NOTE

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

1. 启动 Visual Studio。
2. 在“文件”菜单上，指向“新建”，然后单击“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“Visual C#”，然后单击“Windows”。
4. 查看“新建项目”对话框的顶部，确保“.NET Framework 4”(或更高版本)选为目标框架。
5. 在“模板”窗格中，单击“控制台应用程序”。
6. 在“名称”字段中键入项目的名称。
7. 单击“**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，然后单击“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”。如果未看到程序集，你可能需要确保安装并将其显示出来。请参阅[如何：安装 Office 主互操作程序集](#)。
3. 单击“**确定**”。

添加必要的 using 指令

1. 在“解决方案资源管理器”中，右键单击“Program.cs”文件，然后单击“查看代码”。
2. 将以下 `using` 指令添加到代码文件的顶部：

```
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。

```
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。

```
// 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

- 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(C# 4 中新增)忽略该形参的实参。由于以下代码中未发送任何参数，`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)`。

```
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。

```
// Establish column headings in cells A1 and B1.  
workSheet.Cells[1, "A"] = "ID Number";  
workSheet.Cells[1, "B"] = "Current Balance";
```

3. 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。

```
var row = 1;  
foreach (var acct in accounts)  
{  
    row++;  
    workSheet.Cells[row, "A"] = acct.ID;  
    workSheet.Cells[row, "B"] = acct.Balance;  
}
```

4. 在 `DisplayInExcel` 的末尾添加以下代码以将列宽调整为适合内容。

```
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。

```
((Excel.Range)workSheet.Columns[1]).AutoFit();  
((Excel.Range)workSheet.Columns[2]).AutoFit();
```

C# 4 及更高版本自动将返回的 `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”属性设置为 true。True 是此属性的默认值。

运行项目

1. 在 `Main` 的末尾添加以下行。

```
// Display the list in an Excel spreadsheet.  
DisplayInExcel(bankAccounts);
```

2. 按 Ctrl+F5。

出现包含两个帐户数据的 Excel 工作表。

添加 Word 文档

1. 为了说明 C# 4 及更高版本增强 Office 编程的其他方法，以下代码将打开 Word 应用程序，并创建一个链接到 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 C# 4 中引入的其他两项新功能，简化了对具有引用参数的 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。在 C# 4.0 以及更高版本中，如果希望使用其默认值，可以忽略任何或所有形参的实参。在 C# 3.0 以及更早版本中，由于形参是引用形参，因此必须为每个

形参提供实参且实参必须是变量。

`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。在 C# 4.0 以及更高版本中，你可以对其中两个形参使用命名实参而忽略其他形参。尽管这些是引用形参，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。在 C# 3.0 以及早期版本中，你必须为每个引用形参提供变量实参。

```
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 C# 3.0 以及早期版本中的语言中，需要以下更复杂的代码。

```
static void CreateIconInWordDoc2008()
{
    var wordApp = new Word.Application();
    wordApp.Visible = true;

    // The Add method has four parameters, all of which are optional.
    // In Visual C# 2008 and earlier versions, an argument has to be sent
    // for every parameter. Because the parameters are reference
    // parameters of type object, you have to create an object variable
    // for the arguments that represents 'no value'.

    object useDefaultValue = Type.Missing;

    wordApp.Documents.Add(ref useDefaultValue, ref useDefaultValue,
        ref useDefaultValue, ref useDefaultValue);

    // PasteSpecial has seven reference parameters, all of which are
    // optional. In this example, only two of the parameters require
    // specified values, but in Visual C# 2008 an argument must be sent
    // for each parameter. Because the parameters are reference parameters,
    // you have to construct variables for the arguments.
    object link = true;
    object displayAsIcon = true;

    wordApp.Selection.PasteSpecial( ref useDefaultValue,
        ref link,
        ref useDefaultValue,
        ref displayAsIcon,
        ref useDefaultValue,
        ref useDefaultValue,
        ref useDefaultValue);
}
```

2. 在 `Main` 的末尾添加以下语句。

```
// Create a Word document that contains an icon that links to  
// the spreadsheet.  
CreateIconInWordDoc();
```

3. 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。

```
// 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();
```

4. 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

1. 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。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，由于可以通过使用类型 `dynamic` (而非 `Object`) 表示 COM 方法必需并返回的类型，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需要。有关更多信息，请参见[使用类型 dynamic](#)。

在 C# 4 中，默认行为是嵌入类型信息，而不是使用 PIA。由于该默认行为，因此不需要显式强制转换，之前的几个示例也得到简化。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。

```
((Excel.Range)workSheet.Columns[1]).AutoFit();  
((Excel.Range)workSheet.Columns[2]).AutoFit();
```

2. 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”。
3. 如果看不到“属性”窗口，请按“F4”。
4. 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

1. 将在 `AutoFit` 中对 `DisplayInExcel` 的两个调用替换为以下语句。

```
// Call to AutoFormat in Visual Studio 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，语句则可能更易于理解，如以下代码所示。

```
// Call to AutoFormat in Visual C# 2010.  
workSheet.Range["A1", "B3"].AutoFormat(Format:  
    Excel.XlRangeAutoFormat.xlRangeAutoFormatClassic2);
```

2. 按 Ctrl+F5 查看结果。其他格式在 [XlRangeAutoFormat](#) 枚举中列出。

3. 将步骤 1 中的语句与以下代码比较(以下代码显示 C# 3.0 以及早期版本中要求的参数)。

```
// The AutoFormat method has seven optional value parameters. The  
// following call specifies a value for the first parameter, and uses  
// the default values for the other six.  
  
// Call to AutoFormat in Visual C# 2008. This code is not part of the  
// current solution.  
excelApp.get_Range("A1", "B4").AutoFormat(Excel.XlRangeAutoFormat.xlRangeAutoFormatTable3,  
    Type.Missing, Type.Missing, Type.Missing, Type.Missing, Type.Missing,  
    Type.Missing);
```

示例

以下代码显示完整示例。

```
using System;  
using System.Collections.Generic;  
using System.Linq;  
using Excel = Microsoft.Office.Interop.Excel;  
using Word = Microsoft.Office.Interop.Word;  
  
namespace OfficeProgramminWalkthruComplete  
{  
    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](#)
- [使用类型 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](#)

如何在 COM 互操作编程中使用索引属性 (C# 编程指南)

2021/5/7 • [Edit Online](#)

索引属性改进了在 C# 编程中使用具有参数的 COM 属性的方式。结合使用索引属性与 Visual C# 中的其他功能 (如[命名实参](#)和[可选实参](#)、一种新类型([动态](#))以及[嵌入类型信息](#))可以增强 Microsoft Office 编程。

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, 因此必须使用 `get_Range` 方法, 如以下示例所示。

```
// 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:

```
// Visual C# 2010.  
var excelApp = new Excel.Application();  
// . . .  
Excel.Range targetRange = excelApp.Range["A1"];
```

NOTE

上一示例还使用了[可选实参](#)功能, 以便忽略 `Type.Missing`。

与在 C# 3.0 及更早版本中设置 `Range` 对象的 `Value` 属性的值类似, 需要两个参数。一个为指定范围值类型的可选参数提供实参。另一个提供 `Value` 属性的值。下面的示例说明了这些方法。两者都将 A1 单元格的值设置为 `Name`。

```
// Visual C# 2008.  
targetRange.set_Value(Type.Missing, "Name");  
// Or  
targetRange.Value2 = "Name";
```

利用索引属性, 你可以改为编写以下代码。

```
// 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](#)。

```
// 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();
            //CSharp2008();
        }

        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](#)
- [如何在 Office 编程中使用命名参数和可选参数](#)
- [如何使用 C# 功能访问 Office 互操作对象](#)
- [演练:Office 编程](#)

如何使用平台调用播放 WAV 文件 (C# 编程指南)

2021/5/7 • [Edit Online](#)

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，然后单击“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。

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：

```
dialog1.Filter = "Wav Files (*.wav)|*.wav";
```

```

using System.Windows.Forms;
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:\";
            dialog1.Filter = "Wav Files (*.wav)|*.wav";
            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.
        }
    }
}

```

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”。
- 复制上面的代码，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 `InitializeComponent()` 方法中 Form1.Designer.cs 文件的任何现有代码之后。

后。

```
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();
```

4. 编译并运行该代码。

另请参阅

- [C# 编程指南](#)
- [互操作性概述](#)
- [平台调用详解](#)
- [用平台调用发送数据](#)

演练：Office 编程（C# 和 Visual Basic）

2021/5/7 • [Edit Online](#)

Visual Studio 在 C# 和 Visual Basic 中提供了改进 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。Visual Basic 中的功能包括自动实现的属性、Lambda 表达式语句和集合初始值设定项。

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。

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。

NOTE

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 外接应用程序

- 启动 Visual Studio。
- 在“文件”菜单上，指向“新建”，然后单击“项目”。
- 在“安装的模板”窗格中，展开“Visual Basic”或“Visual C#”，再展开“Office”，然后单击 Office 产品的版本年份。
- 在“模板”窗格中，单击“Excel <version> 外接程序”。
- 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。
- 如果需要，在“名称”框中键入项目的名称。
- 单击“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，然后单击“添加引用”。此时会显示“添加引用”对话框。
- 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，则可能需要确保安装并显示它们（参阅[如何：安装 Office 主互操作程序集](#)）。
- 单击“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.vb”或“ThisAddIn.cs”文件，然后单击“查看代码”。

2. 将以下 `Imports` 语句 (Visual Basic) 或 `using` 指令 (C#) 添加到代码文件的顶部(如果不存在)。

```
using System.Collections.Generic;
using Excel = Microsoft.Office.Interop.Excel;
using Word = Microsoft.Office.Interop.Word;
```

```
Imports Microsoft.Office.Interop
```

创建银行帐户列表

1. 在“解决方案资源管理器”中，右键单击你的项目名称，单击“添加”，然后单击“类”。如果使用的是 Visual Basic，则将类命名为 Account.vb；如果使用的是 C#，则将类命名为 Account.cs。单击 **添加**。
2. 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。有关详细信息，请参阅[自动实现的属性](#)。

```
class Account
{
    public int ID { get; set; }
    public double Balance { get; set; }
}
```

```
Public Class Account
    Property ID As Integer = -1
    Property Balance As Double
End Class
```

3. 若要创建包含两个帐户的 `bankAccounts` 列表，请将以下代码添加到 ThisAddIn.vb 或 ThisAddIn.cs 中的 `ThisAddIn_Startup` 方法。列表声明使用集合初始值设定项。有关详细信息，请参阅[集合初始值设定项](#)。

```
var bankAccounts = new List<Account>
{
    new Account
    {
        ID = 345,
        Balance = 541.27
    },
    new Account
    {
        ID = 123,
        Balance = -127.44
    }
};
```

```
Dim bankAccounts As New List(Of Account) From {
    New Account With {
        .ID = 345,
        .Balance = 541.27
    },
    New Account With {
        .ID = 123,
        .Balance = -127.44
    }
}
```

将数据导出到 Excel

1. 在相同的文件中，将以下方法添加到 `ThisAddIn` 类。该方法设置 Excel 工作簿并将数据导出到工作簿。

```
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();
}
```

```
Sub DisplayInExcel(ByVal accounts As IEnumerable(Of Account),
                   ByVal DisplayAction As Action(Of Account, Excel.Range))

    With Me.Application
        ' Add a new Excel workbook.
        .Workbooks.Add()
        .Visible = True
        .Range("A1").Value = "ID"
        .Range("B1").Value = "Balance"
        .Range("A2").Select()

        For Each ac In accounts
            DisplayAction(ac, .ActiveCell)
            .ActiveCell.Offset(1, 0).Select()
        Next

        ' Copy the results to the Clipboard.
        .Range("A1:B3").Copy()
    End With
End Sub
```

此方法使用 C# 的两项新功能。Visual Basic 中已存在这两项功能。

- 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(C# 4 中新增)忽略该形参的实参。由于上一个示例中未发送任何参数，`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。

```
// Visual C# 2010 provides indexed properties for COM programming.
excelApp.Range["A1"].Value = "ID";
excelApp.ActiveCell.Offset[1, 0].Select();
```

在早期版本的语言中，需要以下特殊语法。

```
// In Visual C# 2008, you cannot access the Range, Offset, and Value  
// properties directly.  
excelApp.get_Range("A1").Value2 = "ID";  
excelApp.ActiveCell.get_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。

有关详细信息，请参阅[如何在 COM 互操作编程中使用索引属性](#)。

2. 在 `DisplayInExcel` 的末尾添加以下代码以将列宽调整为适合内容。

```
excelApp.Columns[1].AutoFit();  
excelApp.Columns[2].AutoFit();
```

```
' Add the following two lines at the end of the With statement.  
.Columns(1).AutoFit()  
.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”设置为其默认值 `True` 时，或者由 `EmbedInteropTypes` 编译器选项引用程序集时，会自动发生这种情况。键入 `dynamic` 允许后期绑定（Visual Basic 已提供该功能）并可避免 C# 3.0 以及早期版本的语言中要求的显式强制转换。

例如，`excelApp.Columns[1]` 返回 `Object`，并且 `AutoFit` 是 Excel `Range` 方法。如果没有 `dynamic`，你必须将 `excelApp.Columns[1]` 返回的对象强制转换为 `Range` 的实例，然后才能调用 `AutoFit` 方法。

```
// Casting is required in Visual C# 2008.  
((Excel.Range)excelApp.Columns[1]).AutoFit();  
  
// Casting is not required in Visual C# 2010.  
excelApp.Columns[1].AutoFit();
```

有关嵌入互操作类型的详细信息，请参阅本主题后面部分的“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`

1. 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。第一个参数是要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](#)。

```
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;  
        }  
    });
```

```

DisplayInExcel(bankAccounts,
    Sub(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 Then
            cell.Interior.Color = RGB(255, 0, 0)
            cell.Offset(0, 1).Interior.Color = RGB(255, 0, 0)
        End If
    End Sub)

```

- 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。

```

var wordApp = new Word.Application();
wordApp.Visible = true;
wordApp.Documents.Add();
wordApp.Selection.PasteSpecial(Link: true, DisplayAsIcon: true);

```

```

Dim wordApp As New Word.Application
wordApp.Visible = True
wordApp.Documents.Add()
wordApp.Selection.PasteSpecial(Link:=True, DisplayAsIcon:=True)

```

此代码展示 C# 中的几项新功能:省略 COM 编程中的 `ref` 关键字、命名参数以及可选参数的能力。

Visual Basic 中已存在这些功能。`PasteSpecial` 方法有七个参数, 所有参数都定义为可选引用参数。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, 发送实参以指示应创建指向剪贴板上工作簿的链接(形参 `Link`)并指示该链接应在 Word 文档中显示为图标(形参 `DisplayAsIcon`)。Visual C# 还允许忽略这些参数的 `ref` 关键字。

要运行应用程序

- 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, 单击“生成”菜单上的“清理解决方案”。否则, 每次在计算机上打开 Excel 时都会运行外接应用程序。

查找 PIA 引用

- 再次运行应用程序, 但不单击“清理解决方案”。
- 选择“开始”。找到“Microsoft Visual Studio <version>”, 然后打开开发人员命令提示。
- 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。
- 在 IL DASM 窗口的“文件”菜单上, 选择“文件”>“打开”。双击“Visual Studio <version>”, 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。注意, 命名空间 `Microsoft.Office.Interop.Excel` 和 `Microsoft.Office.Interop.Word` 包含在程序集中。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` 未包含在列表中。由于项目需要的类型已导入程序集中，因此不需要引用 PIA。这使得部署变得更加容易。用户的计算机上不必存在 PIA，因为应用程序不需要部署特定版本的 PIA，应用程序可设计为与多个版本的 Office 配合使用，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。但是，仅当你的代码不使用你当前所使用 Office 版本中不可用的任何 API 时，此情况才适用。特殊 API 在早期版本中是否可用并不始终明确，因此不建议使用早期版本的 Office。

NOTE

在 Office 2003 以前，Office 并不发布 PIA。因此，生成适用于 Office 2002 或早期版本的互操作程序集的唯一方法是导入 COM 引用。

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，单击“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”属性设置为“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，单击“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 \(Visual Basic\)](#)
- [自动实现的属性 \(C#\)](#)
- [集合初始值设定项](#)
- [对象和集合初始值设定项](#)
- [可选参数](#)
- [按位置和按名称传递自变量](#)
- [命名参数和可选参数](#)
- [早期绑定和后期绑定](#)
- [dynamic](#)
- [使用类型 dynamic](#)
- [Lambda 表达式 \(Visual Basic\)](#)
- [Lambda 表达式 \(C#\)](#)
- [如何在 COM 互操作编程中使用索引属性](#)
- [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
- COM 互操作
- 互操作性

COM 类示例 (C# 编程指南)

2021/5/7 • [Edit Online](#)

下面是将公开为 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 Visual 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将对 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。在接口中声明成员的顺序是用于 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，但第一个实现将为默认类接口。在此处实现向 COM 公开的方法和属性。它们必须标记为公共，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。它们必须标记为公共，并且必须匹配事件接口中的声明。

示例

```
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

- [C# 编程指南](#)

- 互操作性
- “项目设计器”->“生成”页 (C#)

C# 参考

2021/5/7 • [Edit Online](#)

此部分收录了有关 C# 关键字、运算符、特殊字符、预处理器指令、编译器选项以及编译器错误与警告的参考资料。

本节内容

C# 关键字

提供指向有关 C# 关键字和语法的信息的链接。

C# 运算符

提供指向有关 C# 运算符和语法的信息的链接。

C# 特殊字符

收录了主题链接，有助于你了解 C# 中的特殊上下文字符及其用法。

C# 预处理器指令

提供用于嵌入 C# 源代码中的编译器命令有关的信息的链接。

C# 编译器选项

包括有关编译器选项以及如何使用这些选项的信息。

C# 编译器错误

包括演示原因和更正 C# 编译器错误和警告的代码片段。

C# 语言规范

C# 6.0 语言规范。本文是针对 C# 6.0 语言的建议草案。本文档将通过与 ECMA C# 标准委员会协作来进行优化。版本 5.0 已于 2017 年 12 月发布为 [Standard ECMA-334 第 5 版](#) 文档。

已在 C# 6.0 之后的版本中实现的功能将表示在语言规范建议中。这些文档描述了语言规范的增量，以便添加这些新功能。这些文档位于草稿建议窗体中。这些规范将经过优化并提交到 ECMA 标准委员会，以便正式审查并合并到 C# 标准的未来版本。

C# 7.0 规范建议

C# 7.0 中实现了许多新功能。这些功能包括：模式匹配、本地函数、out 变量声明、throw 表达式、二进制文本和数字分隔符。此文件夹包含适用于每个功能的规范。

C# 7.1 规范建议

C# 7.1 中有以下新增功能。首先，可编写返回 `Task` 或 `Task<int>` 的 `Main` 方法。由此可将 `async` 修饰符添加到 `Main`。可推断类型的位置中没有类型时可使用 `default` 表达式。同时，也可推断元组成员名称。最后，可通过泛型使用模式匹配。

C# 7.2 规范建议

C# 7.2 添加了大量小功能。可使用 `in` 关键字通过只读引用传递参数。许多低级更改可支持 `Span` 及相关类型的编译时安全性。在某些情况下，可使用已命名参数，其中后面的参数是位置参数。借助 `private protected` 访问修饰符，可指定调用方仅限于在同一程序集中实现的派生类型。`?:` 运算符可解析位对变量的引用。还可使用前导数字分隔符格式化十六进制和二进制数字。

C# 7.3 规范建议

C# 7.3 是另一个包含多个小更新的重要发布。可对泛型类型参数使用新约束。通过其他更改，可更轻松地使用 `fixed` 字段，包括使用 `stackalloc` 分配。可以重新分配使用 `ref` 关键字声明的本地变量以引用新存储。可将属性放置在自动实现的属性上，该属性针对编译器生成的支持字段。表达式变量可用于初始化表达式中。可以

比较元组的相等性(或不等性)。重载决策也有一些改进。

C# 8.0 规范建议

C# 8.0 可用于 .NET Core 3.0。这些功能包括可为空引用类型、递归模式匹配、默认接口方法、异步流、范围和索引、基于模式的 using 和 using 声明、null 合并分配以及只读实例成员。

C# 9.0 规范建议

C# 9.0 可用于 .NET 5.0。这些功能包括记录、顶层语句、模式匹配增强、仅 init 资源库、目标类型的新表达式、模块初始化表达式、扩展部分方法、静态匿名函数、目标类型的条件表达式、协变返回类型、foreach 循环中的 GetEnumerator 扩展、Lambda 丢弃参数、本地函数的属性、本机大小的整数、函数指针、禁止发出 localsinit 标志和无约束类型参数注释。

相关章节

使用 Visual Studio C# 开发环境

提供指向介绍 IDE 和编辑器的概念性和任务主题的链接。

C# 编程指南

包括有关如何使用 C# 编程语言的信息。

C# 语言版本控制

2021/5/7 • [Edit Online](#)

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

本文中的规则适用于随 Visual Studio 2019 或 .NET SDK 一起提供的编译器。默认情况下，Visual Studio 2017 安装或早期 .NET Core SDK 版本中包含的 C# 编译器以 C# 7.0 为目标。

C# 8.0 仅在 .NET Core 3.x 及更高版本上受支持。许多最新功能需要 .NET Core 3.x 中引入的库和运行时功能：

- [默认接口实现](#)需要使用 .NET Core 3.0 CLR 中的新功能。
- [异步流](#)需要使用新类型 `System.IAsyncDisposable`、`System.Collections.Generic.IAsyncEnumerable<T>` 和 `System.Collections.Generic.IAsyncEnumerator<T>`。
- [索引和范围](#)需要使用新类型 `System.Index` 和 `System.Range`。
- [可为 null 的引用类型](#)利用几个[特性](#)来提供更准确的警告。这些特性是在 .NET Core 3.0 中添加的。其他目标框架并未使用这些特性中的任何一种进行批注。这意味着可为 null 的警告可能无法准确反映潜在问题。

C# 9.0 仅在 .NET 5 及更高版本上受支持。

默认值

编译器根据以下规则确定默认值：

目标	VERSION	C# 语言版本
.NET	5.x	C# 9.0
.NET Core	3.x	C# 8.0
.NET Core	2.x	C# 7.3
.NET Standard	2.1	C# 8.0
.NET Standard	2.0	C# 7.3
.NET Standard	1.x	C# 7.3
.NET Framework	全部	C# 7.3

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

IMPORTANT

Visual Studio 2017 向其创建的所有项目文件添加了 `<LangVersion>latest</LangVersion>` 项。添加此项后，该项表示 C# 7.0。但是，升级到 Visual Studio 2019 后，无论目标框架是什么，该项均表示最新发布的版本。这些项目现在会[覆盖默认行为](#)。你应编辑项目文件，并删除该节点。然后项目将使用建议用于目标框架的编译器版本。

替代默认值

如果必须明确指定 C# 版本，可以通过以下几种方式实现：

- 手动编辑 [项目文件](#)。
- 为 [子目录中的多个项目](#) 设置语言版本。
- 配置 [LangVersion 编译器选项](#)。

TIP

若要了解当前使用的语言版本，请在代码中添加 `#error version` (区分大小写)。这样做可使编译器报告编译器错误 CS8304，并显示一条消息，其中包含正在使用的编译器版本和当前选择的语言版本。有关详细信息，请参阅 [#error\(C# 参考\)](#)。

编辑项目文件

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

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

值 `preview` 使用编译器支持的最新可用的预览 C# 语言版本。

配置多个项目

若要配置多个目录，可以创建包含 `<LangVersion>` 元素的 `Directory.Build.props` 文件。通常是在解决方案目录中完成这件事。将以下内容添加到解决方案目录中的 `Directory.Build.props` 文件：

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

包含该文件的目录的所有子目录中的版本都将使用 C# 预览版。有关详细信息，请查看[自定义生成](#)。

C# 语言版本引用

下表显示当前所有 C# 语言版本。如果编译器较旧，它可能不一定能识别每个值。如果安装的是最新的 .NET SDK，则可以访问列出的所有内容。

"1"	"1"
<code>preview</code>	编译器接受最新预览版中的所有有效语语法。
<code>latest</code>	编译器接受最新发布的编译器版本(包括次要版本)中的语法。
<code>latestMajor</code> (<code>default</code>)	编译器接受最新发布的编译器主要版本中的语法。
<code>9.0</code>	编译器只接受 C# 9.0 或更低版本中所含的语法。
<code>8.0</code>	编译器只接受 C# 8.0 或更低版本中所含的语法。

"1"	1
7.3	编译器只接受 C# 7.3 或更低版本中所含的语法。
7.2	编译器只接受 C# 7.2 或更低版本中所含的语法。
7.1	编译器只接受 C# 7.1 或更低版本中所含的语法。
7	编译器只接受 C# 7.0 或更低版本中所含的语法。
6	编译器只接受 C# 6.0 或更低版本中所含的语法。
5	编译器只接受 C# 5.0 或更低版本中所含的语法。
4	编译器只接受 C# 4.0 或更低版本中所含的语法。
3	编译器只接受 C# 3.0 或更低版本中所含的语法。
ISO-2 (或 2)	编译器只接受 ISO/IEC 23270:2006 C# (2.0) 中所含的语法。
ISO-1 (或 1)	编译器只接受 ISO/IEC 23270:2003 C# (1.0/1.2) 中所含的语法。

TIP

打开 Visual Studio 开发人员命令提示或 Visual Studio 开发人员 PowerShell，运行以下命令以查看计算机上可用的语言版本列表。

```
csc -langversion:?
```

查询此类 ****LangVersion** 编译选项时，会打印如下所示的内容：

```
Supported language versions:
default
1
2
3
4
5
6
7.0
7.1
7.2
7.3
8.0
9.0 (default)
latestmajor
preview
latest
```

值类型 (C# 参考)

2021/3/5 • [Edit Online](#)

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

```
using System;

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

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

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

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

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

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

如前面的示例所示，对值类型变量的操作只影响存储在变量中的值类型实例。

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

```

using System;
using System.Collections.Generic;

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

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

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

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

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

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

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

```

NOTE

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

值类型的种类以及类型约束

值类型可以是以下种类之一：

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

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

你可使用 [struct 约束](#) 指定类型参数为不可为 `null` 的值类型。结构类型和枚举类型都满足 `struct` 约束。从 C# 7.3 开始，你可以在基类约束中使用 `System.Enum`（称为[枚举约束](#)），以指定类型参数为枚举类型。

内置值类型

C# 提供以下内置值类型，也称为“简单类型”：

- [整型数值类型](#)
- [浮点型数值类型](#)
- `bool`，表示布尔值

- `char`, 表示 Unicode UTF-16 字符

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

- 可以使用文字为简单类型提供值。例如，`'A'` 是类型 `char` 的文本，`2001` 是类型 `int` 的文本。
- 可以使用 `const` 关键字声明简单类型的常数。不能具有其他结构类型的常数。
- 常数表达式的操作数都是简单类型的常数，在编译时进行评估。

从 C# 7.0 开始，C# 支持[值元组](#)。值元组是值类型，而不是简单类型。

C# 语言规范

有关更多信息，请参阅[C# 语言规范](#)的以下部分：

- [值类型](#)
- [简单类型](#)
- [变量](#)

另请参阅

- [C# 参考](#)
- [System.ValueType](#)
- [引用类型](#)

整型数值类型 (C# 参考)

2021/5/7 • [Edit Online](#)

整型数值类型 表示整数。所有的整型数值类型均为 [值类型](#)。它们还是 [简单类型](#)，可以使用[文本](#)进行初始化。所有整型数值类型都支持[算术](#)、[位逻辑](#)、[比较](#)和[相等](#)运算符。

整型类型的特征

C# 支持以下预定义整型类型：

C# <code>tt/III</code>	<code>tt</code>	<code>tt</code>	.NET <code>tt</code>
<code>sbyte</code>	-128 到 127	8 位带符号整数	System.SByte
<code>byte</code>	0 到 255	无符号的 8 位整数	System.Byte
<code>short</code>	-32,768 到 32,767	有符号 16 位整数	System.Int16
<code>ushort</code>	0 到 65,535	无符号 16 位整数	System.UInt16
<code>int</code>	-2,147,483,648 到 2,147,483,647	带符号的 32 位整数	System.Int32
<code>uint</code>	0 到 4,294,967,295	无符号的 32 位整数	System.UInt32
<code>long</code>	-9,223,372,036,854,775,808 到 9,223,372,036,854,775,807	64 位带符号整数	System.Int64
<code>ulong</code>	0 到 18,446,744,073,709,551,615	无符号 64 位整数	System.UInt64
<code>nint</code>	取决于平台	带符号的 32 位或 64 位整数	System.IntPtr
<code>nuint</code>	取决于平台	无符号的 32 位或 64 位整数	System.UIntPtr

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

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

表的最后两行中的 `nint` 和 `nuint` 类型是本机大小的整数。在内部它们由所指示的 .NET 类型表示，但在任意情况下关键字和 .NET 类型都是不可互换的。编译器为 `nint` 和 `nuint` 的整数类型提供操作和转换，而不为指针类型 `System.IntPtr` 和 `System.UIntPtr` 提供。有关详细信息，请参阅 [nint](#) 和 [nuint](#) 类型。

每个整型类型的默认值都为零 `0`。除本机大小的类型外，每个整型类型都有 `MinValue` 和 `MaxValue` 常量，提供该类型的最小值和最大值。

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

整数文本

整数文本可以是

- 十进制：不使用任何前缀
- 十六进制：使用 `0x` 或 `0X` 前缀
- 二进制：使用 `0b` 或 `0B` 前缀（在 C# 7.0 和更高版本中可用）

下面的代码演示每种类型的示例：

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

前面的示例还演示了如何将 `_` 用作数字分隔符（从 C# 7.0 开始提供支持）。可以将数字分隔符用于所有类型的数字文本。

整数文本的类型由其后缀确定，如下所示：

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

NOTE

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

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

NOTE

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

- 如果文本的后缀为 `UL`、`U1`、`uL`、`u1`、`LU`、`Lu`、`lU` 或 `lu`，则其类型为 `ulong`。

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

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

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

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

还可以使用强制转换将整数文本所表示的值转换为除确定的文本类型之外的类型：

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

转换

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

C# 语言规范

有关更多信息，请参阅[C# 语言规范](#)的以下部分：

- [整型类型](#)
- [整数文本](#)

请参阅

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

`nint` 和 `nuint` 类型 (C# 参考)

2021/5/7 • [Edit Online](#)

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

本机大小的整数类型在内部表示为 .NET 类型 `System.IntPtr` 和 `System.UIntPtr`。关键字与其他数值类型不同，它们不只是类型的别名。以下语句并不是等效的：

```
nint a = 1;
System.IntPtr a = 1;
```

编译器为 `nint` 和 `nuint` 提供适用于整数类型的操作和转换。

运行时本机整数大小

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

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

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

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

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

MinValue 和 MaxValue

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

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

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

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

常量

可以使用以下范围内的常量值：

- 对于 `nint` : `Int32.MinValue` 到 `Int32.MaxValue`。
- 对于 `nuint` : `UInt32.MinValue` 到 `UInt32.MaxValue`。

转换

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

文本

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

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

不支持的 IntPtr/UIntPtr 成员

`nint` 和 `nuint` 类型不支持 `IntPtr` 和 `UIntPtr` 的以下成员：

- 参数化的构造函数
- `Add(IntPtr, Int32)`
- `CompareTo`
- `Size` - 改用 `sizeOf()`。不支持 `nint.Size`，但你可以使用 `IntPtr.Size` 获取等效的值。
- `Subtract(IntPtr, Int32)`
- `ToInt32`
- `ToInt64`
- `ToPointer`
- `Zero` - 改用 0。

C# 语言规范

有关详细信息，请参阅 C# 9.0 功能建议说明的 [C# 语言规范](#) 和 [本机大小的整数](#) 部分。

另请参阅

- [C# 参考](#)
- [值类型](#)
- [整型数值类型](#)
- [内置数值转换](#)

浮点数值类型 (C# 引用)

2021/3/5 • [Edit Online](#)

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

浮点类型的特征

C# 支持以下预定义浮点类型：

C# <code>关键字</code>	<code>范围</code>	<code>精度</code>	<code>字节</code>	.NET <code>类型</code>
<code>float</code>	$\pm 1.5 \times 10^{-45}$ 至 $\pm 3.4 \times 10^{38}$	大约 6-9 位数字	4 个字节	<code>System.Single</code>
<code>double</code>	$\pm 5.0 \times 10^{-324}$ 到 $\pm 1.7 \times 10^{308}$	大约 15-17 位数字	8 个字节	<code>System.Double</code>
<code>decimal</code>	$\pm 1.0 \times 10^{-28}$ 至 $\pm 7.9228 \times 10^{28}$	28-29 位	16 个字节	<code>System.Decimal</code>

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

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

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

当所需的精度由小数点右侧的位数决定时，`decimal` 类型是合适的。此类数字通常用于财务应用程序、货币金额（例如 \$1.00）、利率（例如 2.625%）等。精确到只有一个小数的偶数用 `decimal` 类型处理会更准确：例如，0.1 可以由 `decimal` 实例精确表示，而没有精确表示 0.1 的 `double` 或 `float` 实例。由于数值类型存在这种差异，因此当你对十进制数据使用 `double` 或 `float` 时，算术计算可能会出现意外的舍入错误。当优化性能比确保准确度更重要时，可以使用 `double` 代替 `decimal`。然而，除了大多数计算密集型应用程序之外，所有应用程序都不会注意到性能上的任何差异。避免使用 `decimal` 的另一个可能原因是最大限度地降低存储需求。例如，ML.NET 使用 `float`，因为对于非常大的数据集，4 个字节与 16 个字节之间的差异合乎情理。有关详细信息，请参阅 [System.Decimal](#)。

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

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

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

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

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

可以使用[标准数字格式字符串](#)或[自定义数字格式字符串](#)设置浮点值的格式。

真实文本

真实文本的类型由其后缀确定，如下所示：

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

下面的代码演示每种类型的示例：

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

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

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

前面的示例还演示了如何将 `_` 用作数字分隔符（从 C# 7.0 开始提供支持）。可以将数字分隔符用于所有类型的数字文本。

还可以使用科学记数法，即指定真实文本的指数部分，如以下示例所示：

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

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

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

转换

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

C# 语言规范

有关更多信息，请参阅[C# 语言规范](#)的以下部分：

- [浮点类型](#)
- [十进制类型](#)
- [真实文本](#)

请参阅

- [C# 参考](#)
- [值类型](#)
- [整型类型](#)
- [标准数字格式字符串](#)
- [.NET 中的数字](#)
- [System.Numerics.Complex](#)

内置数值转换 (C# 参考)

2021/5/7 • [Edit Online](#)

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

隐式数值转换

下表显示内置数值类型之间的预定义隐式转换：

FROM	TO
sbyte	short、int、long、float、double、decimal 或 nint。
byte	short、ushort、int、uint、long、ulong、float、double、decimal、nint 或 uint
short	int、long、float、double、decimal 或 nint
ushort	int、uint、long、ulong、float、double、decimal、nint 或 uint
int	long、float、double、decimal 或 nint
uint	long、ulong、float、double、decimal 或 uint
long	float、double 或 decimal
ulong	float、double 或 decimal
float	double
nint	long、float、double 或 decimal
uint	ulong、float、double 或 decimal

NOTE

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

另请注意

- 任何**整型数值类型**都可以隐式转换为任何**浮点数值类型**。
- 不存在针对 byte 和 sbyte 类型的隐式转换。不存在从 double 和 decimal 类型的隐式转换。
- decimal 类型和 float 或 double 类型之间不存在隐式转换。

- 类型 `int` 的常量表达式的值(例如, 由整数文本所表示的值)如果在目标类型的范围内, 则可隐式转换为 `sbyte`、`byte`、`short`、`ushort`、`uint`、`ulong`、`nint` 或 `nuint`:

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

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

显式数值转换

下表显示不存在[隐式转换](#)的内置数值类型之间的预定义显式转换:

FROM	TO
<code>sbyte</code>	<code>byte</code> 、 <code>ushort</code> 、 <code>uint</code> 、 <code>ulong</code> 或 <code>nuint</code>
<code>byte</code>	<code>sbyte</code>
<code>short</code>	<code>sbyte</code> 、 <code>byte</code> 、 <code>ushort</code> 、 <code>uint</code> 、 <code>ulong</code> 或 <code>nuint</code>
<code>ushort</code>	<code>sbyte</code> 、 <code>byte</code> 或 <code>short</code>
<code>int</code>	<code>sbyte</code> 、 <code>byte</code> 、 <code>short</code> 、 <code>ushort</code> 、 <code>uint</code> 、 <code>ulong</code> 或 <code>nuint</code> 。
<code>uint</code>	<code>sbyte</code> 、 <code>byte</code> 、 <code>short</code> 、 <code>ushort</code> 或 <code>int</code>
<code>long</code>	<code>sbyte</code> 、 <code>byte</code> 、 <code>short</code> 、 <code>ushort</code> 、 <code>int</code> 、 <code>uint</code> 、 <code>ulong</code> 、 <code>nint</code> 或 <code>nuint</code>
<code>ulong</code>	<code>sbyte</code> 、 <code>byte</code> 、 <code>short</code> 、 <code>ushort</code> 、 <code>int</code> 、 <code>uint</code> 、 <code>long</code> 、 <code>nint</code> 或 <code>nuint</code>
<code>float</code>	<code>sbyte</code> 、 <code>byte</code> 、 <code>short</code> 、 <code>ushort</code> 、 <code>int</code> 、 <code>uint</code> 、 <code>long</code> 、 <code>ulong</code> 、 <code>decimal</code> 、 <code>nint</code> 或 <code>nuint</code>
<code>double</code>	<code>sbyte</code> 、 <code>byte</code> 、 <code>short</code> 、 <code>ushort</code> 、 <code>int</code> 、 <code>uint</code> 、 <code>long</code> 、 <code>ulong</code> 、 <code>float</code> 、 <code>decimal</code> 、 <code>nint</code> 或 <code>nuint</code>
<code>decimal</code>	<code>sbyte</code> 、 <code>byte</code> 、 <code>short</code> 、 <code>ushort</code> 、 <code>int</code> 、 <code>uint</code> 、 <code>long</code> 、 <code>ulong</code> 、 <code>float</code> 、 <code>double</code> 、 <code>nint</code> 或 <code>nuint</code>
<code>nint</code>	<code>sbyte</code> 、 <code>byte</code> 、 <code>short</code> 、 <code>ushort</code> 、 <code>int</code> 、 <code>uint</code> 、 <code>ulong</code> 或 <code>nuint</code>
<code>nuint</code>	<code>sbyte</code> 、 <code>byte</code> 、 <code>short</code> 、 <code>ushort</code> 、 <code>int</code> 、 <code>uint</code> 、 <code>long</code> 或 <code>nint</code>

NOTE

显式数值转换可能会导致数据丢失或引发异常, 通常为 [OverflowException](#)。

另请注意

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

C# 语言规范

有关更多信息，请参阅 [C# 语言规范](#) 的以下部分：

- [隐式数值转换](#)
- [显式数值转换](#)

请参阅

- [C# 参考](#)
- [强制转换和类型转换](#)

bool (C# 参考)

2020/11/2 • [Edit Online](#)

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

若要使用 `bool` 类型的值执行逻辑运算，请使用[布尔逻辑运算符](#)。`bool` 类型是[比较](#)和[相等](#)运算符的结果类型。

`bool` 表达式可以是 `if`、`do`、`while` 和 `for` 语句中以及[条件运算符](#) `?:` 中的控制条件表达式。

`bool` 类型的默认值为 `false`。

文本

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

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

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

三值布尔逻辑

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

有关可为空的值类型的详细信息，请参阅[可为空的值类型](#)。

转换

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

C# 语言规范

有关详细信息，请参阅[C# 语言规范](#)中的[bool 类型](#)部分。

请参阅

- [C# 参考](#)
- [值类型](#)
- [true 和 false 运算符](#)

char (C# 参考)

2020/11/2 • [Edit Online](#)

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

II	II	II	.NET II
<code>char</code>	U+0000 到 U+FFFF	16 位	<code>System.Char</code>

`char` 类型的默认值为 `\0`，即 U+0000。

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

字符串 类型将文本表示为 `char` 值的序列。

文本

可以使用以下命令指定 `char` 值：

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

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

如前面的示例所示，你还可以将字符代码的值转换为相应的 `char` 值。

NOTE

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

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

转换

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

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

C# 语言规范

有关详细信息，请参阅 [C# 语言规范](#)的整型类型部分。

请参阅

- [C# 参考](#)
- [值类型](#)
- [字符串](#)
- [System.Text.Rune](#)
- [.NET 中的字符编码](#)

枚举类型 (C# 参考)

2021/3/5 • [Edit Online](#)

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

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

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

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

不能在枚举类型的定义内定义方法。若要向枚举类型添加功能，请创建[扩展方法](#)。

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

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

作为位标志的枚举类型

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

```

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

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

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

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

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

```

有关详细信息和示例，请参阅 [System.FlagsAttribute API 参考页](#) 和 [System.Enum API 参考页](#) 的**非独占成员** 和 **Flags 属性** 部分。

System.Enum 类型和枚举约束

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

从 C# 7.3 开始，你可以在基类约束中使用 `System.Enum` (称为**枚举约束**)，以指定类型参数为枚举类型。所有枚举类型也都满足 `struct` 约束，此约束用于指定类型参数为不可为 null 的值类型。

转换

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

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

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

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

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

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

对于任何枚举类型，都存在分别与 [System.Enum](#) 类型的[装箱](#)和[取消装箱](#)相互转换。

C# 语言规范

有关更多信息，请参阅 [C# 语言规范](#) 的以下部分：

- [枚举](#)
- [枚举值和操作](#)
- [枚举逻辑运算符](#)
- [枚举比较运算符](#)
- [显式枚举转换](#)
- [隐式枚举转换](#)

请参阅

- [C# 参考](#)
- [枚举格式字符串](#)
- [设计准则 - 枚举设计](#)
- [设计准则 - 枚举命名约定](#)
- [switch 语句](#)

结构类型 (C# 参考)

2021/5/10 • [Edit Online](#)

结构类型 (“structure type” 或 “struct type”) 是一种可封装数据和相关功能的 [值类型](#)。使用 `struct` 关键字定义结构类型：

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

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

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

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

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

由于结构类型具有值语义，因此建议定义不可变的 结构类型。

readonly 结构

从 C# 7.2 开始，可以使用 `readonly` 修饰符来声明结构类型为不可变。`readonly` 结构的所有数据成员都必须是只读的，如下所示：

- 任何字段声明都必须具有 `readonly` 修饰符
- 任何属性(包括自动实现的属性)都必须是只读的。在 C# 9.0 和更高版本中，属性可以具有 `init` 访问器。

这样可以保证 `readonly` 结构的成员不会修改该结构的状态。在 C# 8.0 及更高版本中，这意味着除构造函数外的其他实例成员是隐式 `readonly`。

NOTE

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

下面的代码使用 `init-only` 属性资源库定义 `readonly` 结构，此内容在 C# 9.0 及更高版本中提供：

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

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

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

readonly 实例成员

从 C#8.0 开始，还可以使用 `readonly` 修饰符声明实例成员不会修改结构的状态。如果不能将整个结构类型声明为 `readonly`，可使用 `readonly` 修饰符标记不会修改结构状态的实例成员。

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

通常，将 `readonly` 修饰符应用于以下类型的实例成员：

- 方法：

```
public readonly double Sum()
{
    return X + Y;
}
```

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

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

- 属性和索引器：

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

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

NOTE

编译器会将自动实现的属性的 `get` 访问器声明为 `readonly`，而不管属性声明中是否存在 `readonly` 修饰符。

在 C# 9.0 和更高版本中，可以将 `readonly` 修饰符应用于具有 `init` 访问器的属性或索引器：

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

不能将 `readonly` 修饰符应用于结构类型的静态成员。

编译器可以使用 `readonly` 修饰符进行性能优化。有关详细信息, 请参阅[编写安全有效的 C# 代码](#)。

结构类型的设计限制

设计结构类型时, 具有与[类](#)类型相同的功能, 但有以下例外:

- 不能声明无参数构造函数。每个结构类型都已经提供了一个隐式无参数构造函数, 该构造函数生成类型的[默认值](#)。
- 不能在声明实例字段或属性时对它们进行初始化。但是, 可以在其声明中初始化[静态](#)或[常量](#)字段或静态属性。
- 结构类型的构造函数必须初始化该类型的所有实例字段。
- 结构类型不能从其他类或结构类型继承, 也不能作为类的基础类型。但是, 结构类型可以实现[接口](#)。
- 不能在结构类型中声明[终结器](#)。

结构类型的实例化

在 C# 中, 必须先初始化已声明的变量, 然后才能使用该变量。由于结构类型变量不能为 `null` (除非它是[可为空的值类型](#)的变量), 因此, 必须实例化相应类型的实例。有多种方法可实现此目的。

通常, 可使用 `new` 运算符调用适当的构造函数来实例化结构类型。每个结构类型都至少有一个构造函数。这是一个隐式无参数构造函数, 用于生成类型的[默认值](#)。还可以使用[默认值表达式](#)来生成类型的默认值。

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

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

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

在处理[内置值类型](#)的情况下, 请使用相应的文本来指定类型的值。

按引用传递结构类型变量

将结构类型变量作为参数传递给方法或从方法返回结构类型值时, 将复制结构类型的整个实例。这可能会影响高性能方案中涉及大型结构类型的代码的性能。通过按引用传递结构类型变量, 可以避免值复制操作。使用 `ref`、`out` 或 `in` 方法参数修饰符, 指示必须按引用传递参数。使用 `ref` 返回值按引用返回方法结果。有关详细信息, 请参阅[编写安全有效的 C# 代码](#)。

`ref` 结构

从 C# 7.2 开始，可以在结构类型的声明中使用 `ref` 修饰符。`ref` 结构类型的实例在堆栈上分配，并且不能转义到托管堆。为了确保这一点，编译器将 `ref` 结构类型的使用限制如下：

- `ref` 结构不能是数组的元素类型。
- `ref` 结构不能是类或非 `ref` 结构的字段的声明类型。
- `ref` 结构不能实现接口。
- `ref` 结构不能被装箱为 `System.ValueType` 或 `System.Object`。
- `ref` 结构不能是类型参数。
- `ref` 结构变量不能由 `Lambda 表达式` 或 `本地函数` 捕获。
- `ref` 结构变量不能在 `async` 方法中使用。但是，可以在同步方法中使用 `ref` 结构变量，例如，在返回 `Task` 或 `Task<TResult>` 的方法中。
- `ref` 结构变量不能在 `迭代器` 中使用。

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

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

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

```
public readonly ref struct ConversionRequest
{
    public ConversionRequest(double rate, ReadOnlySpan<double> values)
    {
        Rate = rate;
        Values = values;
    }

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

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

struct 约束

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

转换

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

C# 语言规范

有关详细信息，请参阅 [C# 语言规范](#) 中的 [结构部分](#)。

有关 C#7.2 及更高版本中引入的功能的更多信息，请参见以下功能建议说明：

- 只读结构
- 只读实例成员
- 类 ref 类型的编译时安全性

请参阅

- [C# 参考](#)
- [设计指南 - 在类和结构之间选择](#)
- [设计指南 - 结构设计](#)
- [类、结构和记录](#)

元组类型 (C# 参考)

2020/11/2 • [Edit Online](#)

元组功能在 C# 7.0 及更高版本中可用，它提供了简洁的语法，用于将多个数据元素分组成一个轻型数据结构。

下面的示例演示了如何声明元组变量、对它进行初始化并访问其数据成员：

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

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

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

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

从 C# 7.3 开始，元组类型支持[相等运算符](#) `==` 和 `!=`。有关详细信息，请参阅[元组相等](#)部分。

元组类型是[值类型](#)；元组元素是公共字段。这使得元组为可变的值类型。

NOTE

元组功能需要 `System.ValueTuple` 类型和相关的泛型类型（例如 `System.ValueTuple<T1,T2>`），这些类型在 .NET Core 和 .NET Framework 4.7 及更高版本中可用。若要在面向 .NET Framework 4.6.2 或更早版本的项目中使用元组，请将 NuGet 包 `System.ValueTuple` 添加到项目。

可以使用任意数量的元素定义元组：

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

元组的用例

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

```

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

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

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

    var min = int.MaxValue;
    var max = int.MinValue;
    foreach (var i in input)
    {
        if (i < min)
        {
            min = i;
        }
        if (i > max)
        {
            max = i;
        }
    }
    return (min, max);
}

```

如前面的示例所示，可以直接使用返回的元组实例，或者可以在单独的变量中 [析构](#) 它。

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

通常，你会使用元组对相关的数据元素进行松散分组。这通常在专用和内部实用工具方法中有用。对于公共 API 用例，请考虑定义[类](#)或[结构](#)类型。

元组字段名称

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

```

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

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

```

从 C# 7.1 开始，如果未指定字段名称，则可以根据元组初始化表达式中相应变量的名称推断出此名称，如下面的示例所示：

```

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

```

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

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

在这些情况下，你可以显式指定字段的名称，或按字段的默认名称访问字段。

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

```
var a = 1;
var t = (a, b: 2, 3);
Console.WriteLine($"The 1st element is {t.Item1} (same as {t.a}).");
Console.WriteLine($"The 2nd element is {t.Item2} (same as {t.b}).");
Console.WriteLine($"The 3rd element is {t.Item3}.");
// Output:
// The 1st element is 1 (same as 1).
// The 2nd element is 2 (same as 2).
// The 3rd element is 3.
```

[元组赋值](#)和[元组相等比较](#)不会考虑字段名称。

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

元组赋值和析构

C# 支持满足以下两个条件的元组类型之间的赋值：

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

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

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

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

还可以使用赋值运算符 `=` 在单独的变量中析构元组实例。为此，可以使用以下方式之一进行操作：

- 在括号内显式声明每个变量的类型：

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

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

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

- 使用现有变量：

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

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

有关析构元组和其他类型的详细信息，请参阅[析构元组和其他类型](#)。

元组相等

从 C# 7.3 开始，元组类型支持 `==` 和 `!=` 运算符。这些运算符按照元组元素的顺序将左侧操作数的成员与相应的右侧操作数的成员进行比较。

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

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

如前面的示例所示，`==` 和 `!=` 操作不会考虑元组字段名称。

同时满足以下两个条件时，两个元组可比较：

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

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

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

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

元组作为 out 参数

通常，你会将具有 `out` 参数的方法重构为返回元组的方法。但是，在某些情况下，`out` 参数可以是元组类型。

下面的示例演示了如何将元组作为 `out` 参数使用：

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

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

元组与 `System.Tuple`

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

- `ValueTuple` 类型是值类型。`Tuple` 类型是引用类型。
- `ValueTuple` 类型是可变的。`Tuple` 类型是不可变的。
- `ValueTuple` 类型的数据成员是字段。`Tuple` 类型的数据成员是属性。

C# 语言规范

有关详细信息，请参阅以下功能建议说明：

- 推断元组名称（也称为元组投影初始值设定项）
- 支持对元组类型使用 `==` 和 `!=`

请参阅

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

可为空的值类型 (C# 参考)

2020/11/2 • [Edit Online](#)

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

NOTE

C# 8.0 引入了可为空引用类型功能。有关详细信息，请参阅[可为空引用类型](#)。从 C# 2 开始，提供可为空的值类型。

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

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

声明和赋值

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

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

int m2 = 10;
int? m = m2;

bool? flag = null;

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

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

检查可为空值类型的实例

从 C# 7.0 开始，可以将 [is 运算符与类型模式](#) 结合使用，既检查 `null` 的可为空值类型的实例，又检索基础类型的值：

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

始终可以使用以下只读属性来检查和获取可为空值类型变量的值：

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

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

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

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

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

从可为空的值类型转换为基础类型

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

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

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

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

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

```
int? n = null;

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

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

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

提升的运算符

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

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

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

NOTE

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

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

- 既不大于等于 `null`，
- 也不小于 `null`

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

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

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

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

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

装箱和取消装箱

可为空值类型的实例 `T?` 已装箱，如下所示：

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

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

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

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

如何确定可为空的值类型

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

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

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

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

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

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

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

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

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

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

可使用以下示例中提供的代码来确定实例是否是可为空的值类型：

```
int? a = 14;
Console.WriteLine(IsOfNullableType(a)); // output: True

int b = 17;
Console.WriteLine(IsOfNullableType(b)); // output: False

bool IsOfNullableType<T>(T o)
{
    var type = typeof(T);
    return Nullable.GetUnderlyingType(type) != null;
}
```

NOTE

此部分中所述的方法不适用于[可为空的引用类型](#)的情况。

C# 语言规范

有关更多信息，请参阅[C# 语言规范](#)的以下部分：

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

请参阅

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

引用类型 (C# 参考)

2021/5/7 • [Edit Online](#)

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

下列关键字用于声明引用类型：

- [class](#)
- [interface](#)
- [delegate](#)
- [record](#)

C# 也提供了下列内置引用类型：

- [dynamic](#)
- [object](#)
- [string](#)

请参阅

- [C# 参考](#)
- [C# 关键字](#)
- [指针类型](#)
- [值类型](#)

内置引用类型 (C# 引用)

2020/11/2 • [Edit Online](#)

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

对象类型

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

字符串类型

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

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

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

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

+ 运算符连接字符串：

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

这将创建包含“good morning”的字符串对象。

字符串是不可变的，即：字符串对象在创建后，尽管从语法上看似乎可以更改其内容，但事实上并不可行。例如，编写此代码时，编译器实际上会创建一个新的字符串对象来保存新的字符序列，且该新对象将赋给 `b`。已为 `b` 分配的内存(当它包含字符串“h”时)可用于垃圾回收。

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

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

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

同样，[] 运算符也可用于循环访问字符串中的每个字符：

```
string str = "test";

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

字符串文本属于 `string` 类型且可以两种形式编写(带引号和仅 @ 带引号)。带引号字符串括在双引号 ("") 内。

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

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

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

NOTE

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

逐字字符串文本以 @ 开头, 并且也括在双引号内。例如:。

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

逐字字符串的优点是不处理转义序列, 这样就可轻松编写完全限定的 Windows 文件名等:

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

若要在 @-quoted 字符串中包含双引号, 双倍添加即可:

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

委托类型

委托类型的声明与方法签名相似。它有一个返回值和任意数目任意类型的参数:

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

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

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

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

动态类型

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

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

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

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

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

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

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

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

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

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

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

```

using System;

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

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

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

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

    class ExampleClass
    {
        static dynamic field;
        dynamic prop { get; set; }

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

            if (d is int)
            {
                return local;
            }
            else
            {
                return two;
            }
        }
    }
}

// Results:
// Local variable
// 2
// Local variable

```

另请参阅

- [C# 参考](#)
- [C# 关键字](#)
- [事件](#)
- [使用类型 dynamic](#)
- [有关使用字符串的最佳做法](#)
- [基本字符串操作](#)
- [新建字符串](#)
- [类型测试和强制转换运算符](#)
- [如何使用模式匹配以及 is 和 as 运算符安全地进行强制转换](#)

- 演练: 创建和使用动态对象
- System.Object
- System.String
- System.Dynamic.DynamicObject

记录 (C# 参考)

2021/5/10 • [Edit Online](#)

从 C# 9 开始，可以使用 `record` 关键字定义一个引用类型，用来提供用于封装数据的内置功能。通过使用位置参数或标准属性语法，可以创建具有不可变属性的记录类型：

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

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

此外，还可以创建具有可变属性和字段的记录类型：

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

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

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

你还可以使用[结构类型](#)来设计以数据为中心的类型，这些类型提供值相等性，并且很少或没有任何行为。但对于相对较大的数据模型，结构类型有一些缺点：

- 它们不支持继承。
- 它们在确定值相等性时效率较低。对于值类型，`ValueType.Equals` 方法使用反射来查找所有字段。对于记录，编译器将生成 `Equals` 方法。实际上，记录中的值相等性实现的速度明显更快。
- 在某些情况下，它们会占用更多内存，因为每个实例都有所有数据的完整副本。记录类型是[引用类型](#)，因此，记录实例只包含对数据的引用。

属性定义的位置语法

在创建实例时，可以使用位置参数来声明记录的属性，并初始化属性值：

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

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

当你为属性定义使用位置语法时，编译器将创建以下内容：

- 为记录声明中提供的每个位置参数提供一个公共的 init-only 自动实现的属性。init-only 属性只能在构造函数中或使用属性初始值设定项来设置。
- 主构造函数，它的参数与记录声明上的位置参数匹配。
- 一个 Deconstruct 方法，对记录声明中提供的每个位置参数都有一个 out 参数。只有当有两个或更多的位置参数时，才会提供这个方法。此方法解构了使用位置语法定义的属性；它忽略了使用标准属性语法定义的属性。

如果生成的自动实现的属性定义并不是你所需要的，你可以自行定义同名的属性。如果这样做，生成的构造函数和解构函数将使用你的属性定义。例如，下面的示例定义了 FirstName 位置属性 internal 而不是 public。

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

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

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

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

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

不可变性

记录类型不一定是不可变的。可以用 set 访问器和非 readonly 的字段来声明属性。虽然记录可以是可变的，但它们使创建不可变的数据模型变得更容易。

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

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

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

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

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

记录类型特有的功能是由编译器合成的方法实现的，这些方法都不会通过修改对象状态来影响不可变性。

值相等性

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。对于其他引用类型，相等是指标识。也就是说，如果一个引用类型的两个变量引用同一个对象，那么这两个变量是相等的。

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

下面的示例说明了记录类型的值相等性：

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

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

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

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

为了实现值相等性，编译器合成了以下方法：

- [Object.Equals\(Object\)](#) 的替代。

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

- 一个虚拟的 [Equals](#) 方法，其参数为记录类型。此方法实现 [IEquatable<T>](#)。
- [Object.GetHashCode\(\)](#) 的替代。
- 运算符 [==](#) 和 [!=](#) 的替代。

在 [class](#) 类型中，可以手动替代相等性方法和运算符以实现值相等性，但开发和测试这种代码会非常耗时，而且容易出错。内置此功能可防止在添加或更改属性或字段时忘记更新自定义替代代码导致的 bug。

你可以编写自己的实现来替换这些合成的方法。如果记录类型的方法与任何合成方法的签名匹配，则编译器不会合成该方法。

如果在记录类型中提供自己的 [Equals](#) 实现，则还应提供 [GetHashCode](#) 的实现。

非破坏性变化

如果需要改变记录实例的不可变属性，可以使用 `with` 表达式来实现非破坏性变化。`with` 表达式创建一个新的记录实例，该实例是现有记录实例的一个副本，修改了指定属性和字段。使用[对象初始值设定项](#)语法来指定要更改的值，如以下示例中所示：

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

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

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

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

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

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

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

为了实现此功能，编译器合成了一个克隆方法和一个复制构造函数。构造函数接收一个要复制的记录的实例，然后调用克隆方法。当你使用 `with` 表达式时，编译器将创建调用复制构造函数的代码，然后设置在 `with` 表达式中指定的属性。

如果你需要不同的复制行为，可以编写自己的复制构造函数。如果你这样做，编译器将不会合成复制构造函数。如果记录是 `sealed`，则使构造函数为 `private`，否则使其为 `protected`。

你不能替代克隆方法，也不能创建名为 `Clone` 的成员。克隆方法的实际名称是由编译器生成的。

用于显示的内置格式设置

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

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

对于引用类型，将显示属性所引用的对象的类型名称，而不是属性值。在下面的示例中，数组是一个引用类型，因此显示的是 `System.String[]`，而不是实际的数组元素值：

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

为了实现此功能，编译器合成了一个虚拟 `PrintMembers` 方法和一个 `ToString` 替代。`ToString` 替代创建了一个 `StringBuilder` 对象，它的类型名称后跟一个左括号。它调用 `PrintMembers` 以添加属性名称和值，然后添加右括

号。下面的示例展示了类似于合成替代中包含的代码：

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

你可以提供自己的 `PrintMembers` 实现或 `ToString` 替代。本文后面的派生记录中的 `PrintMembers` 格式设置一节中提供了示例。

继承

一条记录可以从另一条记录继承。但是，记录不能从类继承，类也不能从记录继承。

派生记录类型中的位置参数

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

下面的示例说明了具有位置属性语法的继承：

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

继承层次结构中的相等性

要使两个记录变量相等，运行时类型必须相等。包含变量的类型可能不同。下面的代码示例中说明了这一点：

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

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

在本示例中，所有实例都具有相同的属性和相同的属性值。但 `student == teacher` 返回 `False`（尽管这两个都是 `Person` 型变量），`student == student2` 返回 `True`（尽管一个是 `Person` 变量，而另一个是 `Student` 变量）。

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

在比较派生类型的两个实例时，合成的相等性方法会检查基类型和派生类型的所有属性是否相等。合成的 `GetHashCode` 方法从基类型和派生的记录类型中声明的所有属性和字段中使用 `GetHashCode` 方法。

派生记录中的 `with` 表达式

由于合成的克隆方法使用 [协变返回类型](#)，因此 `with` 表达式的结果与表达式的操作数具有相同的运行时类型。运行时类型的所有属性都会被复制，但你只能设置编译时类型的属性，如下面的示例所示：

```
public record Point(int X, int Y)
{
    public int Zbase { get; set; }
};

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

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

    Point p2 = p1 with { X = 5, Y = 6, Zbase = 7 }; // Can't set Name or Zderived
    Console.WriteLine(p2 is NamedPoint); // output: True
    Console.WriteLine(p2);
    // output: NamedPoint { X = 5, Y = 6, Zbase = 7, Name = A, Zderived = 4 }

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

派生记录中的 `PrintMembers` 格式设置

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

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

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

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

- 对于派生自 `object` 的 `sealed` 记录（不声明基本记录）：使用

```
private bool PrintMembers(StringBuilder builder);
```

- 对于派生自其他记录的 `sealed` 记录：使用

```
protected sealed override bool PrintMembers(StringBuilder builder);
```

- 对于非 `sealed` 且派生自对象的记录：使用 `protected virtual bool PrintMembers(StringBuilder builder);`

- 对于非 `sealed` 且派生自其他记录的记录: 使用

```
protected override bool PrintMembers(StringBuilder builder);
```

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

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

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

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

派生记录中的解构函数行为

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

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

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

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

泛型约束

没有任何泛型约束要求类型是记录。记录满足 `class` 约束条件。要对记录类型的特定层次结构进行约束，请像

对基类一样对基本记录进行约束。有关详细信息，请参阅类型参数的约束。

C# 语言规范

有关详细信息，请参阅 C# 语言规范中的类部分。

若要详细了解 C# 9 及更高版本中引入的功能，请参阅以下功能建议说明：

- [记录](#)
- [init-only 资源库](#)
- [协变返回结果](#)

另请参阅

- [C# 参考](#)
- [设计指南 - 在类和结构之间选择](#)
- [设计指南 - 结构设计](#)
- [类、结构和记录](#)
- [with 表达式](#)

class (C# 参考)

2020/11/2 • [Edit Online](#)

使用 `class` 关键字声明类，如下例中所示：

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

备注

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

无	无
无	<code>class ClassA { }</code>
单向	<code>class DerivedClass : BaseClass { }</code>
无，实现两个接口	<code>class ImplClass : IFace1, IFace2 { }</code>
单一，实现一个接口	<code>class ImplDerivedClass : BaseClass, IFace1 { }</code>

直接在命名空间中声明的、未嵌套在其它类中的类，可以是 [公共](#) 或 [内部](#)。默认情况下类为 `internal`。

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

有关详细信息，请参阅[访问修饰符](#)。

可以声明具有类型参数的泛型类。有关更多信息，请参见[泛型类](#)。

一个类可包含下列成员的声明：

- [构造函数](#)
- [常量](#)
- [字段](#)
- [终结器](#)
- [方法](#)
- [属性](#)
- [索引器](#)
- [运算符](#)
- [事件](#)

- 委托
- 类
- 接口
- 结构类型
- 枚举类型

示例

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

```

class Child
{
    private int age;
    private string name;

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

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

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

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

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

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

```

注释

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

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

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

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

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

C# 语言规范

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

请参阅

- [C# 参考](#)
- [C# 编程指南](#)
- [C# 关键字](#)
- [引用类型](#)

interface (C# 参考)

2021/3/5 • • [Edit Online](#)

接口定义协定。实现该协定的任何 `class` 或 `struct` 必须提供接口中定义的成员的实现。从 C# 8.0 开始，接口可为成员定义默认实现。它还可以定义 `static` 成员，以便提供常见功能的单个实现。

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

有关详细信息和示例，请参阅[接口](#)。

示例

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

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

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

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

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

- [方法](#)
- [属性](#)
- [索引器](#)
- [事件](#)

上述成员声明通常不包含主体。从 C# 8.0 开始，接口成员可以声明主体。这称为“默认实现”。具有主体的成员允许接口为不提供重写实现的类和结构提供“默认”实现。此外，从 C# 8.0 开始，接口可以包括：

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

接口不能包含实例状态。虽然现在允许使用静态字段，但接口中不允许使用实例字段。接口中不支持[实例自动](#)

属性，因为它们将隐式声明隐藏的字段。此规则对属性声明有细微影响。在接口声明中，以下代码不会像在 `class` 或 `struct` 中一样声明自动实现的属性。相反，它会声明一个属性，该属性没有默认实现，而必须在该实现接口的任何类型中实现它：

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

一个接口可从一个或多个基接口继承。当接口 [重写基接口中的方法实现](#)时，必须使用[显式接口实现](#)语法。

基类型列表包含基类和接口时，基类必须是列表中的第 1 项。

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

有关显式接口实现的详细信息，请参阅[显式接口实现](#)。

示例

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

```

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

    int Y
    {
        get;
        set;
    }

    double Distance
    {
        get;
    }
}

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

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

    public int Y { get; set; }

    // Property implementation
    public double Distance =>
        Math.Sqrt(X * X + Y * Y);
}

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

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

```

C# 语言规范

有关详细信息，请参阅 [C# 语言规范的接口部分](#) 和 [默认接口成员 - C# 8.0 的功能规范](#)

请参阅

- [C# 参考](#)

- [C# 编程指南](#)
- [C# 关键字](#)
- [引用类型](#)
- [接口](#)
- [使用属性](#)
- [使用索引器](#)

可为 null 的引用类型 (C# 引用)

2021/3/5 • [Edit Online](#)

NOTE

本文介绍可为 null 的引用类型。还可以声明可为 null 的值类型。

由于在可为 null 的感知上下文选择加入了代码，从 C#8.0 开始，可以使用可为 null 的引用类型。可为 null 的引用类型、null 静态分析警告和 null 包容运算符是可选的语言功能。默认情况下，所有功能都将关闭。可为 null 的上下文在项目级使用生成设置进行控制，或在代码中使用 pragmas 进行控制。

在可为 null 的感知上下文中：

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

不可为 null 的引用类型 `T` 和可为 null 的引用类型 `T?` 之间的区别按照编译器对上述规则的解释强制执行的。类型为 `T` 的变量和类型为 `T?` 的变量由相同的 .NET 类型表示。下面的示例声明了一个不可为 null 的字符串和一个可为 null 的字符串，然后使用 null 包容运算符将一个值分配给不可为 null 的字符串：

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

变量 `notNull` 和 `nullable` 都由 `String` 类型表示。因为不可为 null 的类型和可为 null 的类型都存储为相同的类型，所以有几个位置不允许使用可为 null 的引用类型。通常，可为 null 的引用类型不能用作基类或实现的接口。可为 null 的引用类型不能用于任何对象创建或类型测试表达式。可为 null 的引用类型不能是成员访问表达式的类型。下面的示例说明了这些构造：

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

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

可为 null 的引用和静态分析

上一部分中的示例说明了可为 null 的引用类型的性质。可为 null 的引用类型不是新的类类型，而是对现有引用类型的注释。编译器使用这些注释来帮助你查找代码中潜在的 null 引用错误。不可为 null 的引用类型和可为 null 的引用类型在运行时没有区别。编译器不会为不可为 null 的引用类型添加任何运行时检查。这有利于编译

时分析。编译器将生成警告，帮助你查找和修复代码中潜在的 null 错误。你需声明意向，如果代码违反该意向，编译器会发出警告。

在可为 null 的上下文中，编译器对任何引用类型的变量（可为 null 的和不可为 null 的）执行静态分析。编译器跟踪每个引用变量的 null 状态，如“非 null”和“可能为 null”。不可为 null 的引用的默认状态为“非 null”。可为 null 的引用的默认状态为“可能为 null”。

不可为 null 的引用类型在取消引用时应该始终是安全的，因为它们的 null 状态是“非 null”。若强制执行该规则，如果不可为 null 的引用类型没有初始化为非 null 值，编译器将发出警告。必须在声明的位置分配局部变量。每个构造函数都必须分配每个字段，无论是在其主体、被调用的构造函数还是使用字段初始值设定项。如果将不可为 null 的引用分配给状态为“可能为 null”的引用，编译器将发出警告。但是，由于不可为 null 的引用是“非 null”，因此在取消引用这些变量时不会发出警告。

可为 null 的引用类型可以进行初始化或分配给 `null`。因此，静态分析必须在取消对变量的引用之前确定该变量为“非 null”。如果可为 null 的引用被确定为“可能为 null”，则不能将其分配给不可为 null 的引用变量。以下类显示了这些警告的示例：

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

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

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

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

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

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

以下代码段显示了编译器在使用此类时发出警告的位置：

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

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

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

前面的示例演示编译器的静态分析，以确定引用变量的 null 状态。编译器对 null 检查和分配应用语言规则以通知其分析。编译器无法对方法或属性的语义进行假设。如果调用执行 null 检查的方法，则编译器无法得知这些方法会影响变量的 null 状态。可以将许多属性添加到 API，以通知编译器有关参数和返回值的语义。这些属性已应用于 .NET Core 库中的许多常见 API。例如，[IsNullOrEmpty](#) 已经更新，编译器正确地将该方法解释为 null 检查。有关应用于 null 状态静态分析的属性的更多信息，请参阅[可为 null 属性](#)的文章。

设置可为 null 的上下文

可以通过两种方式控制可为 null 的上下文。在项目级别，可以添加 `<Nullable>enable</Nullable>` 项目设置。在单个 C# 源文件中，可以添加 `#nullable enable` 来启用可为 null 的上下文。请参阅关于[设置可为 null 策略](#)的文章。

C# 语言规范

有关详细信息，请参阅 [C# 语言规范](#) 的以下建议：

- [可为空引用类型](#)
- [可为空引用类型规范草案](#)

请参阅

- [C# 参考](#)
- [可以为 null 的值类型](#)

void (C# 参考)

2021/5/7 • • [Edit Online](#)

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

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

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

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

不能将 `void` 用作变量的类型。

另请参阅

- [C# 参考](#)
- [System.Void](#)

var (C# 参考)

2021/5/7 • [Edit Online](#)

从 C# 3.0 开始，在方法范围内声明的变量可以具有隐式“类型”`var`。隐式类型本地变量为强类型，就像用户已经自行声明该类型，但编译器决定类型一样。`i` 的以下两个声明在功能上是等效的：

```
var i = 10; // Implicitly typed.  
int i = 10; // Explicitly typed.
```

IMPORTANT

在启用可为空引用类型的情况下使用 `var` 时，即使表达式类型不可为空，也始终表示可为空引用类型。

`var` 关键字的常见用途是用于构造函数调用表达式。使用 `var` 则不能在变量声明和对象实例化中重复类型名称，如下面的示例所示：

```
var xs = new List<int>();
```

从 C# 9.0 开始，可以使用由目标确定类型的 `new` 表达式作为替代方法：

```
List<int> xs = new();  
List<int>? ys = new();
```

在模式匹配中，在 `var` 模式中使用 `var` 关键字。

示例

下面的示例演示两个查询表达式。在第一个表达式中，`var` 的使用是允许的，但不是必需的，因为查询结果的类型可以明确表述为 `IEnumerable<string>`。不过，在第二个表达式中，`var` 允许结果是一系列匿名类型，且相应类型的名称只可供编译器本身访问。如果使用 `var`，便无法为结果新建类。请注意，在示例 #2 中，`foreach` 迭代变量 `item` 必须也为隐式类型。

```
// Example #1: var is optional when
// the select clause specifies a string
string[] words = { "apple", "strawberry", "grape", "peach", "banana" };
var wordQuery = from word in words
    where word[0] == 'g'
    select word;

// Because each element in the sequence is a string,
// not an anonymous type, var is optional here also.
foreach (string s in wordQuery)
{
    Console.WriteLine(s);
}

// Example #2: var is required because
// the select clause specifies an anonymous type
var custQuery = from cust in customers
    where cust.City == "Phoenix"
    select new { cust.Name, cust.Phone };

// var must be used because each item
// in the sequence is an anonymous type
foreach (var item in custQuery)
{
    Console.WriteLine("Name={0}, Phone={1}", item.Name, item.Phone);
}
```

请参阅

- [C# 参考](#)
- [隐式类型本地变量](#)
- [LINQ 查询操作中的类型关系](#)

内置类型 (C# 参考)

2021/5/7 • [Edit Online](#)

下表列出了 C# 内置值类型：

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

下表列出了 C# 内置引用类型：

C#	.NET
<code>object</code>	<code>System.Object</code>
<code>string</code>	<code>System.String</code>
<code>dynamic</code>	<code>System.Object</code>

在上表中，左侧列中的每个 C# 类型关键字(`nint`, `nuint` 和 `dynamic` 除外)都是相应 .NET 类型的别名。它们是可

互换的。例如，以下声明声明了相同类型的变量：

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

第一个表的最后两行中的 `nint` 和 `nuint` 类型是本机大小的整数。在内部它们由所指示的 .NET 类型表示，但在任意情况下关键字和 .NET 类型都是不可互换的。编译器为 `nint` 和 `nuint` 的整数类型提供操作和转换，而不为指针类型 `System.IntPtr` 和 `System.UIntPtr` 提供。有关详细信息，请参阅 [nint](#) 和 [nuint](#) 类型。

`void` 关键字表示缺少类型。将其用作不返回值的方法的返回类型。

请参阅

- [C# 参考](#)
- [C# 类型的默认值](#)

非托管类型 (C# 参考)

2021/5/7 • [Edit Online](#)

如果某个类型是以下类型之一，则它是非托管类型：

- `sbyte`、`byte`、`short`、`ushort`、`int`、`uint`、`long`、`ulong`、`char`、`float`、`double`、`decimal` 或 `bool`
- 任何枚举类型
- 任何指针类型
- 任何用户定义的 `struct` 类型，只包含非托管类型的字段，并且在 C# 7.3 及更早版本中，不是构造类型(包含至少一个类型参数的类型)

从 C# 7.3 开始，可使用 `unmanaged` 约束指定：类型参数为“非指针、不可为 null 的非托管类型”。

从 C# 8.0 开始，仅包含非托管类型的字段的 构造 结构类型也是非托管类型，如以下示例所示：

```
using System;

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

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

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

泛型结构可以是非托管类型的源，也可以是不是非托管构造类型的源。前面的示例定义一个泛型结构 `Coords<T>`，并提供非托管构造类型的示例。不是非托管类型情况的示例是 `Coords<object>`。它不是非托管性质，因为它具有不是非托管性质的 `object` 类型的字段。如果你希望所有 构造类型都是非托管类型，请在泛型结构的定义中使用 `unmanaged` 约束：

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

C# 语言规范

有关详细信息，请参阅 [C# 语言规范的指针类型部分](#)。

请参阅

- [C# 参考](#)
- [指针类型](#)
- [内存和跨度相关类型](#)
- [sizeof 运算符](#)
- [stackalloc](#)

C# 类型的默认值 (C# 参考)

2021/3/22 • [Edit Online](#)

下表显示 C# 类型的默认值：

II	III
任何引用类型	<code>null</code>
任何内置整数数值类型	0(零)
任何内置浮点型数值类型	0(零)
<code>bool</code>	<code>false</code>
<code>char</code>	<code>'\0'</code> (U + 0000)
<code>enum</code>	表达式 <code>(E)0</code> 生成的值, 其中 <code>E</code> 是枚举标识符。
<code>struct</code>	通过如下设置生成的值: 将所有值类型的字段设置为其默认值, 将所有引用类型的字段设置为 <code>null</code> 。
任何可以为 <code>null</code> 的值类型	<code>HasValue</code> 属性为 <code>false</code> 且 <code>Value</code> 属性未定义的实例。该默认值也称为可以为 <code>null</code> 的值类型的“null”值。

使用 `default` 运算符生成默认类型值, 如下面的示例所示:

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

从 C# 7.1 开始, 可使用 `default` 文本来初始化变量, 使其具有其类型的默认值:

```
int a = default;
```

对于值类型, 隐式无参数构造函数还可生成类型的默认值, 如以下示例所示:

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

在运行时, 如果 `System.Type` 实例表示一个值类型, 则可以使用 `Activator.CreateInstance(Type)` 方法来调用无参数构造函数, 以获取该类型的默认值。

C# 语言规范

有关更多信息, 请参阅 [C# 语言规范](#) 的以下部分:

- [默认值](#)
- [默认构造函数](#)

请参阅

- [C# 参考](#)
- [构造函数](#)

C# 关键字

2021/5/10 • [Edit Online](#)

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

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

```
abstract  
as  
base  
bool  
break  
byte  
case  
catch  
char  
checked  
class  
const  
continue  
decimal  
default  
delegate  
do  
double  
else  
enum  
  
event  
explicit  
extern  
false  
finally  
fixed  
float  
for  
foreach  
goto  
if  
implicit  
in  
int  
interface  
internal  
is  
lock  
long
```

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

static
string
struct
switch
this
throw
true
try
typeof
uint
ulong
unchecked
unsafe
ushort
using
virtual
void
volatile
while
```

上下文关键字

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

```
add
and
alias
ascending
async
await
by
descending
```

dynamic
equals
from

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

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

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

请参阅

- [C# 参考](#)

访问修饰符 (C# 参考)

2020/11/2 • [Edit Online](#)

访问修饰符是关键字，用于指定成员或类型已声明的可访问性。本部分介绍四个访问修饰符：

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

可使用访问修饰符指定以下六个可访问性级别：

- `public` : 访问不受限制。
- `protected` : 访问限于包含类或派生自包含类型的类型。
- `internal` : 访问限于当前程序集。
- `protected internal` : 访问限于当前程序集或派生自包含类型的类型。
- `private` : 访问限于包含类。
- `private protected` : 访问限于包含类或当前程序集中派生自包含类型的类型。

本部分还介绍以下内容：

- **可访问性级别**: 使用四种访问修饰符，声明六个可访问性级别。
- **可访问性域**: 指定可在程序段中的何处引用成员。
- **对使用可访问性级别的限制**: 概括对使用已声明的可访问性级别的限制。

请参阅

- [C# 参考](#)
- [C# 编程指南](#)
- [C# 关键字](#)
- [访问修饰符](#)
- [访问关键字](#)
- [修饰符](#)

可访问性级别 (C# 参考)

2021/5/7 • [Edit Online](#)

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

访问级别	说明
<code>public</code>	访问不受限制。
<code>protected</code>	访问限于包含类或派生自包含类的类型。
<code>internal</code>	访问限于当前程序集。
<code>protected internal</code>	访问限于当前程序集或派生自包含类的类型。
<code>private</code>	访问限于包含类。
<code>private protected</code>	访问限于包含类或当前程序集中派生自包含类的类型。自 C# 7.2 之后可用。

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

命名空间中不允许出现访问修饰符。命名空间没有任何访问限制。

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

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

作为其他类型的成员的嵌套类型可以具有如下表所示的声明的可访问性。

嵌套类型	访问级别	可访问性
<code>enum</code>	<code>public</code>	无
<code>class</code>	<code>private</code>	<code>public</code> <code>protected</code> <code>internal</code> <code>private</code> <code>protected internal</code> <code>private protected</code>

interface	public	public
		protected
		internal
		private *
		protected internal
		private protected
struct	private	public
		internal
		private

* 具有 `private` 可访问性的 `interface` 成员必须具有默认的实现。

嵌套类型的可访问性依赖于它的[可访问域](#)，该域是由已声明的成员可访问性和直接包含类型的可访问域这二者共同确定的。但是，嵌套类型的可访问域不能超出包含类型的可访问域。

C# 语言规范

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

请参阅

- [C# 参考](#)
- [C# 编程指南](#)
- [C# 关键字](#)
- [访问修饰符](#)
- [可访问性域](#)
- [可访问性级别的使用限制](#)
- [访问修饰符](#)
- [public](#)
- [private](#)
- [受保护](#)
- [internal](#)

可访问域 (C# 参考)

2020/11/2 • [Edit Online](#)

成员的可访问域可指定成员可以引用哪些程序分区。如果成员嵌套于其他类型中，则该成员的可访问域是由该成员的[可访问性级别](#)和直接包含类型的可访问域这二者共同确定的。

顶级类型的可访问域至少是在其中进行声明的项目的程序文本。也就是说，该域包含此项目的所有源文件。嵌套类型的可访问域至少是在其中进行声明的类型的程序文本。也就是说，域是类型的主题，其中包括所有嵌套类型。嵌套类型的可访问域决不能超出包含类型的可访问域。下例说明了这些概念。

示例

此示例包含一个顶级类型 `T1` 和两个嵌套类 `M1` 和 `M2`。这两个类包含的字段具有不同的已声明可访问性。在 `Main` 方法中，每条语句后跟注释以指示每个成员的可访问域。请注意，试图引用不可访问成员的语句将被注释掉。如果想要查看通过引用不可访问成员引起的编译器错误，请一次删除一条注释。

```
public class T1
{
    public static int publicInt;
    internal static int internalInt;
    private static int privateInt = 0;

    static T1()
    {
        // T1 can access public or internal members
        // in a public or private (or internal) nested class.
        M1.publicInt = 1;
        M1.internalInt = 2;
        M2.publicInt = 3;
        M2.internalInt = 4;

        // Cannot access the private member privateInt
        // in either class:
        // M1.privateInt = 2; //CS0122
    }

    public class M1
    {
        public static int publicInt;
        internal static int internalInt;
        private static int privateInt = 0;
    }

    private class M2
    {
        public static int publicInt = 0;
        internal static int internalInt = 0;
        private static int privateInt = 0;
    }
}

class MainClass
{
    static void Main()
    {
        // Access is unlimited.
        T1.publicInt = 1;

        // Accessible only in current assembly.
    }
}
```

```
    T1.internalInt = 2;

    // Error CS0122: inaccessible outside T1.
    // T1.privateInt = 3;

    // Access is unlimited.
    T1.M1.publicInt = 1;

    // Accessible only in current assembly.
    T1.M1.internalInt = 2;

    // Error CS0122: inaccessible outside M1.
    //     T1.M1.privateInt = 3;

    // Error CS0122: inaccessible outside T1.
    //     T1.M2.publicInt = 1;

    // Error CS0122: inaccessible outside T1.
    //     T1.M2.internalInt = 2;

    // Error CS0122: inaccessible outside M2.
    //     T1.M2.privateInt = 3;

    // Keep the console open in debug mode.
    System.Console.WriteLine("Press any key to exit.");
    System.Console.ReadKey();
}

}
```

C# 语言规范

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

请参阅

- [C# 参考](#)
- [C# 编程指南](#)
- [C# 关键字](#)
- [访问修饰符](#)
- [可访问性级别](#)
- [可访问性级别的使用限制](#)
- [访问修饰符](#)
- [public](#)
- [private](#)
- [受保护](#)
- [internal](#)

可访问性级别的使用限制 (C# 参考)

2020/11/2 • [Edit Online](#)

在声明中指定类型时，检查类型的可访问性级别是否依赖于成员或其他类型的可访问性级别。例如，直接基类必须至少具有与派生类相同的可访问性。以下声明会导致编译器错误，因为基类 `BaseClass` 的可访问性低于 `MyClass`：

```
class BaseClass {...}
public class MyClass: BaseClass {...} // Error
```

下表汇总了对已声明可访问性级别的限制。

类型	限制
类	类类型的直接基类必须至少具有与类类型本身相同的可访问性。
接口	接口类型的显式基接口必须至少具有与接口类型本身相同的可访问性。
委托	委托类型的返回类型和参数类型必须至少具有与委托类型本身相同的可访问性。
常量	常量的类型必须至少具有与常量本身相同的可访问性。
字段	字段的类型必须至少具有与字段本身相同的可访问性。
方法	方法的返回类型和参数类型必须至少具有与方法本身相同的可访问性。
属性	属性的类型必须至少具有与属性本身相同的可访问性。
事件	事件的类型必须至少具有与事件本身相同的可访问性。
索引器	索引器的类型和参数类型必须至少具有与索引器本身相同的可访问性。
运算符	运算符的类型和参数类型必须至少具有与运算符本身相同的可访问性。
构造函数	构造函数的参数类型必须至少具有与构造函数本身相同的可访问性。

示例

下面的示例包含不同类型的错误声明。每个声明后的注释指示预期编译器错误。

```
// Restrictions on Using Accessibility Levels
// CS0052 expected as well as CS0053, CS0056, and CS0057
// To make the program work, change access level of both class B
// and MyPrivateMethod() to public.

using System;

// A delegate:
delegate int MyDelegate();

class B
{
    // A private method:
    static int MyPrivateMethod()
    {
        return 0;
    }
}

public class A
{
    // Error: The type B is less accessible than the field A.myField.
    public B myField = new B();

    // Error: The type B is less accessible
    // than the constant A.myConst.
    public readonly B myConst = new B();

    public B MyMethod()
    {
        // Error: The type B is less accessible
        // than the method A.MyMethod.
        return new B();
    }

    // Error: The type B is less accessible than the property A.MyProp
    public B MyProp
    {
        set
        {
        }
    }

    MyDelegate d = new MyDelegate(B.MyPrivateMethod);
    // Even when B is declared public, you still get the error:
    // "The parameter B.MyPrivateMethod is not accessible due to
    // protection level."

    public static B operator +(A m1, B m2)
    {
        // Error: The type B is less accessible
        // than the operator A.operator +(A,B)
        return new B();
    }

    static void Main()
    {
        Console.WriteLine("Compiled successfully");
    }
}
```

C# 语言规范

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

请参阅

- [C# 参考](#)
- [C# 编程指南](#)
- [C# 关键字](#)
- [访问修饰符](#)
- [可访问性域](#)
- [可访问性级别](#)
- [访问修饰符](#)
- [public](#)
- [private](#)
- [受保护](#)
- [internal](#)

internal (C# 参考)

2020/11/2 • [Edit Online](#)

`internal` 关键字是类型和类型成员的访问修饰符。

本页涵盖 `internal` 访问。`internal` 关键字也是 `protected internal` 访问修饰符的一部分。

只有在同一程序集的文件中，内部类型或成员才可访问，如下例所示：

```
public class BaseClass
{
    // Only accessible within the same assembly.
    internal static int x = 0;
}
```

有关 `internal` 和其他访问修饰符的比较，请参阅[可访问性级别](#)和[访问修饰符](#)。

有关程序集的详细信息，请参阅[.NET 中的程序集](#)。

内部访问通常用于基于组件的开发，因为它可使一组组件以私有方式进行协作，而不必向应用程序代码的其余部分公开。例如，用于生成图形用户界面的框架可以提供 `Control` 和 `Form` 类，这两个类通过使用具有内部访问权限的成员进行协作。由于这些成员是内部的，因此不会向正在使用框架的代码公开。

从定义具有内部访问权限的类型或成员的程序集外部引用该类型或成员是错误的。

示例

此示例包含两个文件，即 `Assembly1.cs` 和 `Assembly1_a.cs`。第一个文件包含内部基类 `BaseClass`。在第二个文件中，尝试实例化 `BaseClass` 会产生错误。

```
// Assembly1.cs
// Compile with: /target:library
internal class BaseClass
{
    public static int intM = 0;
}
```

```
// Assembly1_a.cs
// Compile with: /reference:Assembly1.dll
class TestAccess
{
    static void Main()
    {
        var myBase = new BaseClass();    // CS0122
    }
}
```

示例

在此示例中，使用在示例 1 中所用的相同文件，并将 `BaseClass` 的可访问性级别更改为 `public`。另将成员 `intM` 的可访问性级别更改为 `internal`。在此例中，可以实例化类，但不能访问内部成员。

```
// Assembly2.cs
// Compile with: /target:library
public class BaseClass
{
    internal static int intM = 0;
}
```

```
// Assembly2_a.cs
// Compile with: /reference:Assembly2.dll
public class TestAccess
{
    static void Main()
    {
        var myBase = new BaseClass(); // Ok.
        BaseClass.intM = 444; // CS0117
    }
}
```

C# 语言规范

有关详细信息，请参阅 [C# 语言规范](#) 中的 [声明的可访问性](#)。该语言规范是 C# 语法和用法的权威资料。

请参阅

- [C# 参考](#)
- [C# 编程指南](#)
- [C# 关键字](#)
- [访问修饰符](#)
- [可访问性级别](#)
- [修饰符](#)
- [public](#)
- [private](#)
- [protected](#)

private (C# 参考)

2020/11/2 • [Edit Online](#)

`private` 关键字是一个成员访问修饰符。

本页涵盖 `private` 访问。`private` 关键字也是 `private protected` 访问修饰符的一部分。

私有访问是允许的最低访问级别。私有成员只有在声明它们的类和结构体中才是可访问的，如以下示例所示：

```
class Employee
{
    private int i;
    double d; // private access by default
}
```

同一体中的嵌套类型也可以访问那些私有成员。

在声明私有成员的类或结构外引用它会导致编译时错误。

有关 `private` 和其他访问修饰符的比较，请参阅[可访问性级别](#)和[访问修饰符](#)。

示例

在此示例中，`Employee` 类包含两个私有数据成员 `name` 和 `salary`。作为私有成员，它们只能通过成员方法来访问。添加名为 `GetName` 和 `Salary` 的公共方法，以便可以对私有成员进行受控的访问。通过公共方法访问 `name` 成员，而通过公共只读属性访问 `salary` 成员。（有关详细信息，请参阅[属性](#)。）

```
class Employee2
{
    private string name = "FirstName, LastName";
    private double salary = 100.0;

    public string GetName()
    {
        return name;
    }

    public double Salary
    {
        get { return salary; }
    }
}

class PrivateTest
{
    static void Main()
    {
        var e = new Employee2();

        // The data members are inaccessible (private), so
        // they can't be accessed like this:
        //     string n = e.name;
        //     double s = e.salary;

        // 'name' is indirectly accessed via method:
        string n = e.GetName();

        // 'salary' is indirectly accessed via property
        double s = e.Salary;
    }
}
```

C# 语言规范

有关详细信息，请参阅 [C# 语言规范](#) 中的 [声明的可访问性](#)。该语言规范是 C# 语法和用法的权威资料。

请参阅

- [C# 参考](#)
- [C# 编程指南](#)
- [C# 关键字](#)
- [访问修饰符](#)
- [可访问性级别](#)
- [修饰符](#)
- [public](#)
- [受保护](#)
- [internal](#)

protected (C# 参考)

2021/3/5 • [Edit Online](#)

`protected` 关键字是一个成员访问修饰符。

NOTE

本页涵盖 `protected` 访问。`protected` 关键字也属于 `protected internal` 和 `private protected` 访问修饰符。

受保护成员在其所在的类中可由派生类实例访问。

有关 `protected` 和其他访问修饰符的比较，请参阅[可访问性级别](#)。

示例

只有在通过派生类类型进行访问时，基类的受保护成员在派生类中才是可访问的。以下面的代码段为例：

```
class A
{
    protected int x = 123;
}

class B : A
{
    static void Main()
    {
        var a = new A();
        var b = new B();

        // Error CS1540, because x can only be accessed by
        // classes derived from A.
        // a.x = 10;

        // OK, because this class derives from A.
        b.x = 10;
    }
}
```

语句 `a.x = 10` 生成错误，因为它是在静态方法 `Main` 中生成的，而不是类 `B` 的实例。

无法保护结构成员，因为无法继承结构。

示例

在此示例中，`DerivedPoint` 类是从 `Point` 派生的。因此，可以从派生类直接访问基类的受保护成员。

```
class Point
{
    protected int x;
    protected int y;
}

class DerivedPoint: Point
{
    static void Main()
    {
        var dpoint = new DerivedPoint();

        // Direct access to protected members.
        dpoint.x = 10;
        dpoint.y = 15;
        Console.WriteLine($"x = {dpoint.x}, y = {dpoint.y}");
    }
}
// Output: x = 10, y = 15
```

如果将 `x` 和 `y` 的访问级别更改为 `private`, 编译器将发出错误消息:

```
'Point.y' is inaccessible due to its protection level.
```

```
'Point.x' is inaccessible due to its protection level.
```

C# 语言规范

有关详细信息, 请参阅 [C# 语言规范](#) 中的 [声明的可访问性](#)。该语言规范是 C# 语法和用法的权威资料。

请参阅

- [C# 参考](#)
- [C# 编程指南](#)
- [C# 关键字](#)
- [访问修饰符](#)
- [可访问性级别](#)
- [修饰符](#)
- [public](#)
- [private](#)
- [internal](#)
- [Internal Virtual 关键字的安全问题](#)

public (C# 参考)

2021/3/5 • [Edit Online](#)

`public` 关键字是类型和类型成员的访问修饰符。公共访问是允许的最高访问级别。对访问公共成员没有限制，如以下示例所示：

```
class SampleClass
{
    public int x; // No access restrictions.
}
```

有关详细信息，请参阅[访问修饰符](#)和[可访问性级别](#)。

示例

在下面的示例中，声明了两个类：`PointTest` 和 `Program`。直接从 `Program` 访问 `PointTest` 的公共成员 `x` 和 `y`。

```
class PointTest
{
    public int x;
    public int y;
}

class Program
{
    static void Main()
    {
        var p = new PointTest();
        // Direct access to public members.
        p.x = 10;
        p.y = 15;
        Console.WriteLine($"x = {p.x}, y = {p.y}");
    }
}
// Output: x = 10, y = 15
```

如果将 `public` 访问级别更改为 `private` 或 `protected`，则会收到错误消息：

"`PointTest.y`"不可访问，因为它受保护级别限制。

C# 语言规范

有关详细信息，请参阅[C# 语言规范](#)中的[声明的可访问性](#)。该语言规范是 C# 语法和用法的权威资料。

请参阅

- [C# 参考](#)
- [C# 编程指南](#)
- [访问修饰符](#)
- [C# 关键字](#)
- [访问修饰符](#)
- [可访问性级别](#)

- 修饰符
- private
- 受保护
- internal

受保护的内部成员 (C# 参考)

2020/11/2 • [Edit Online](#)

`protected internal` 关键字组合是一种成员访问修饰符。可从当前程序集或派生自包含类的类型访问受保护的内部成员。有关 `protected internal` 和其他访问修饰符的比较，请参阅[可访问性级别](#)。

示例

可从包含程序集内的任何类型访问基类的受保护的内部成员。也可在另一程序集中的派生类中访问它，前提是通过派生类类型的变量进行访问。以下面的代码段为例：

```
// Assembly1.cs
// Compile with: /target:library
public class BaseClass
{
    protected internal int myValue = 0;

    class TestAccess
    {
        void Access()
        {
            var baseObject = new BaseClass();
            baseObject.myValue = 5;
        }
    }
}
```

```
// Assembly2.cs
// Compile with: /reference:Assembly1.dll
class DerivedClass : BaseClass
{
    static void Main()
    {
        var baseObject = new BaseClass();
        var derivedObject = new DerivedClass();

        // Error CS1540, because myValue can only be accessed by
        // classes derived from BaseClass.
        // baseObject.myValue = 10;

        // OK, because this class derives from BaseClass.
        derivedObject.myValue = 10;
    }
}
```

此示例包含两个文件，即 `Assembly1.cs` 和 `Assembly2.cs`。第一个文件包含公共基类 `BaseClass` 和另一个类 `TestAccess`。`BaseClass` 拥有受保护的内部成员 `myValue`，由 `TestAccess` 类型访问。在第二个文件中，如果尝试通过 `BaseClass` 的实例访问 `myValue`，会生成错误，但如果尝试通过一个派生类的实例来访问此成员，`DerivedClass` 会成功。

结构成员不能为 `protected internal`，因为无法继承结构。

C# 语言规范

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

请参阅

- [C# 参考](#)
- [C# 编程指南](#)
- [C# 关键字](#)
- [访问修饰符](#)
- [可访问性级别](#)
- [修饰符](#)
- [public](#)
- [private](#)
- [internal](#)
- [Internal Virtual 关键字的安全问题](#)

private protected (C# 参考)

2020/11/2 • [Edit Online](#)

`private protected` 关键字组合是一种成员访问修饰符。仅派生自包含类的类型可访问私有受保护成员，但仅能在其包含程序集中访问。有关 `private protected` 和其他访问修饰符的比较，请参阅[可访问性级别](#)。

NOTE

`private protected` 访问修饰符在 C# 版本 7.2 及更高版本中有效。

示例

仅当变量的静态类型是派生类类型时，可从派生的类型访问基类的私有受保护成员（在其包含程序集中访问）。以下面的代码段为例：

```
public class BaseClass
{
    private protected int myValue = 0;
}

public class DerivedClass1 : BaseClass
{
    void Access()
    {
        var baseObject = new BaseClass();

        // Error CS1540, because myValue can only be accessed by
        // classes derived from BaseClass.
        // baseObject.myValue = 5;

        // OK, accessed through the current derived class instance
        myValue = 5;
    }
}
```

```
// Assembly2.cs
// Compile with: /reference:Assembly1.dll
class DerivedClass2 : BaseClass
{
    void Access()
    {
        // Error CS0122, because myValue can only be
        // accessed by types in Assembly1
        // myValue = 10;
    }
}
```

此示例包含两个文件，即 `Assembly1.cs` 和 `Assembly2.cs`。第一个文件包含公共基类 `BaseClass` 及其派生的类型 `DerivedClass1`。`BaseClass` 拥有私有受保护成员 `myValue`，`DerivedClass1` 尝试以两种方式访问该成员。通过 `BaseClass` 的实例第一次尝试访问 `myValue` 时会产生错误。但是，如果尝试在 `DerivedClass1` 中将其用作继承的成员，则会成功。

在第二个文件中，如果尝试将 `myValue` 作为 `DerivedClass2` 的继承成员进行访问，会生成错误，因为仅 `Assembly1` 中的派生类型可以访问它。

如果 `Assembly1.cs` 包含命名为 `Assembly2` 的 `InternalsVisibleToAttribute`, 则派生类 `DerivedClass1` 将可以访问 `BaseClass` 中声明的 `private protected` 成员。 `InternalsVisibleTo` 使 `private protected` 成员对其他程序集中的派生类可见。

结构成员不能为 `private protected`, 因为无法继承结构。

C# 语言规范

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

请参阅

- [C# 参考](#)
- [C# 编程指南](#)
- [C# 关键字](#)
- [访问修饰符](#)
- [可访问性级别](#)
- [修饰符](#)
- [public](#)
- [private](#)
- [internal](#)
- [Internal Virtual 关键字的安全问题](#)

abstract (C# 参考)

2020/11/2 • [Edit Online](#)

`abstract` 修饰符指示被修改内容的实现已丢失或不完整。`abstract` 修饰符可用于类、方法、属性、索引和事件。在类声明中使用 `abstract` 修饰符来指示某个类仅用作其他类的基类，而不用于自行进行实例化。标记为抽象的成员必须由派生自抽象类的非抽象类来实现。

示例

在此示例中，类 `Square` 必须提供 `GetArea` 的实现，因为它派生自 `Shape`：

```
abstract class Shape
{
    public abstract int GetArea();
}

class Square : Shape
{
    int side;

    public Square(int n) => side = n;

    // GetArea method is required to avoid a compile-time error.
    public override int GetArea() => side * side;

    static void Main()
    {
        var sq = new Square(12);
        Console.WriteLine($"Area of the square = {sq.GetArea()}");
    }
}
// Output: Area of the square = 144
```

抽象类具有以下功能：

- 抽象类不能实例化。
- 抽象类可能包含抽象方法和访问器。
- 无法使用 `sealed` 修饰符来修改抽象类，因为两个修饰符的含义相反。`sealed` 修饰符阻止类被继承，而 `abstract` 修饰符要求类被继承。
- 派生自抽象类的非抽象类，必须包含全部已继承的抽象方法和访问器的实际实现。

在方法或属性声明中使用 `abstract` 修饰符，以指示该方法或属性不包含实现。

抽象方法具有以下功能：

- 抽象方法是隐式的虚拟方法。
- 只有抽象类中才允许抽象方法声明。
- 由于抽象方法声明不提供实际的实现，因此没有方法主体；方法声明仅以分号结尾，且签名后没有大括号 {}。例如：

```
public abstract void MyMethod();
```

实现由方法 `override` 提供，它是非抽象类的成员。

- 在抽象方法声明中使用 `static` 或 `virtual` 修饰符是错误的。

除了声明和调用语法方面不同外，抽象属性的行为与抽象方法相似。

- 在静态属性上使用 `abstract` 修饰符是错误的。
- 通过包含使用 `override` 修饰符的属性声明，可在派生类中重写抽象继承属性。

有关抽象类的详细信息，请参阅[抽象类、密封类及类成员](#)。

抽象类必须为所有接口成员提供实现。

实现接口的抽象类有可能将接口方法映射到抽象方法上。例如：

```
interface I
{
    void M();
}

abstract class C : I
{
    public abstract void M();
}
```

示例

在此示例中，类 `DerivedClass` 派生自抽象类 `BaseClass`。抽象类包含抽象方法 `AbstractMethod`，以及两个抽象属性 `X` 和 `Y`。

```
abstract class BaseClass // Abstract class
{
    protected int _x = 100;
    protected int _y = 150;
    public abstract void AbstractMethod(); // Abstract method
    public abstract int X { get; }
    public abstract int Y { get; }
}

class DerivedClass : BaseClass
{
    public override void AbstractMethod()
    {
        _x++;
        _y++;
    }

    public override int X // overriding property
    {
        get
        {
            return _x + 10;
        }
    }

    public override int Y // overriding property
    {
        get
        {
            return _y + 10;
        }
    }

    static void Main()
    {
        var o = new DerivedClass();
        o.AbstractMethod();
        Console.WriteLine($"x = {o.X}, y = {o.Y}");
    }
}
// Output: x = 111, y = 161
```

在前面的示例中，如果你尝试通过使用如下语句来实例化抽象类：

```
BaseClass bc = new BaseClass(); // Error
```

将遇到一个错误，告知编译器无法创建抽象类“BaseClass”的实例。

C# 语言规范

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

请参阅

- [C# 参考](#)
- [C# 编程指南](#)
- [修饰符](#)
- [virtual](#)
- [override](#)
- [C# 关键字](#)

async (C# 参考)

2020/11/2 • [Edit Online](#)

使用 `async` 修饰符可将方法、[lambda 表达式](#) 或 [匿名方法](#) 指定为异步。如果对方法或表达式使用此修饰符，则其称为异步方法。如下示例定义了一个名为 `ExampleMethodAsync` 的异步方法：

```
public async Task<int> ExampleMethodAsync()
{
    //...
}
```

如果不熟悉异步编程，或者不了解异步方法如何在不阻止调用方线程的情况下使用 `await` 运算符执行可能需要长时间运行的工作，请参阅[使用 Async 和 Await 的异步编程](#)中的说明。如下代码见于一种异步方法中，且调用 `HttpClient.GetStringAsync` 方法：

```
string contents = await httpClient.GetStringAsync(requestUrl);
```

异步方法同步运行，直至到达其第一个 `await` 表达式，此时会将方法挂起，直到等待的任务完成。同时，如下节示例中所示，控件将返回到方法的调用方。

如果 `async` 关键字修改的方法不包含 `await` 表达式或语句，则该方法将同步执行。编译器警告将通知你不包含 `await` 语句的任何异步方法，因为该情况可能表示存在错误。请参阅[编译器警告\(等级 1\)CS4014](#)。

`async` 关键字是上下文关键字，原因在于只有当它修饰方法、[lambda 表达式](#) 或 [匿名方法](#) 时，它才是关键字。在所有其他上下文中，都会将其解释为标识符。

示例

下面的示例展示了异步事件处理程序 `StartButton_Click` 和异步方法 `ExampleMethodAsync` 之间的控制结构和流程。此异步方法的结果是 Web 页面的字符数。此代码适用于在 Visual Studio 中创建的 Windows Presentation Foundation (WPF) 应用或 Windows 应用商店应用；请参见有关设置应用的代码注释。

可以在 Visual Studio 中将此代码作为 Windows Presentation Foundation (WPF) 应用或 Windows 应用商店应用运行。需要一个名为 `StartButton` 的按钮控件和一个名为 `ResultsTextBox` 的文本框控件。切勿忘记设置名称和处理程序，以便获得类似于以下代码的内容：

```
<Button Content="Button" HorizontalAlignment="Left" Margin="88,77,0,0" VerticalAlignment="Top" Width="75"
       Click="StartButton_Click" Name="StartButton"/>
<TextBox HorizontalAlignment="Left" Height="137" Margin="88,140,0,0" TextWrapping="Wrap"
        Text="<Enter a URL>" VerticalAlignment="Top" Width="310" Name="ResultsTextBox"/>
```

将代码作为 WPF 应用运行：

- 将此代码粘贴到 `MainWindow.xaml.cs` 中的 `MainWindow` 类中。
- 添加对 `System.Net.Http` 的引用。
- 为 `System.Net.Http` 添加一个 `using` 指令。

将此代码作为 Windows 应用商店应用运行：

- 将此代码粘贴到 `MainPage.xaml.cs` 中的 `MainPage` 类中。
- 为 `System.Net.Http` 和 `System.Threading.Tasks` 添加 `using` 指令。

```

private async void StartButton_Click(object sender, RoutedEventArgs e)
{
    // ExampleMethodAsync returns a Task<int>, which means that the method
    // eventually produces an int result. However, ExampleMethodAsync returns
    // the Task<int> value as soon as it reaches an await.
    ResultsTextBox.Text += "\n";

    try
    {
        int length = await ExampleMethodAsync();
        // Note that you could put "await ExampleMethodAsync()" in the next line where
        // "length" is, but due to when '+=' fetches the value of ResultsTextBox, you
        // would not see the global side effect of ExampleMethodAsync setting the text.
        ResultsTextBox.Text += String.Format("Length: {0:N0}\n", length);
    }
    catch (Exception)
    {
        // Process the exception if one occurs.
    }
}

public async Task<int> ExampleMethodAsync()
{
    var httpClient = new HttpClient();
    int exampleInt = (await httpClient.GetStringAsync("http://msdn.microsoft.com")).Length;
    ResultsTextBox.Text += "Preparing to finish ExampleMethodAsync.\n";
    // After the following return statement, any method that's awaiting
    // ExampleMethodAsync (in this case, StartButton_Click) can get the
    // integer result.
    return exampleInt;
}
// The example displays the following output:
// Preparing to finish ExampleMethodAsync.
// Length: 53292

```

IMPORTANT

若要深入了解各项任务以及在等待任务期间所执行的代码, 请参阅[使用 Async 和 Await 的异步编程](#)。有关使用类似元素的完整控制台示例, 请参阅[在异步任务完成时对其进行处理 \(C#\)](#)。

返回类型

异步方法可具有以下返回类型:

- `Task`
- `Task<TResult>`
- `void`。对于除事件处理程序以外的代码, 通常不鼓励使用 `async void` 方法, 因为调用方不能 `await` 那些方法, 并且必须实现不同的机制来报告成功完成或错误条件。
- 从 C# 7.0 开始, 任何具有可访问的 `GetAwaiter` 方法的类型。`System.Threading.Tasks.ValueTask<TResult>` 类型属于此类实现。它通过添加 NuGet 包 `System.Threading.Tasks.Extensions` 的方式可用。

此异步方法既不能声明任何 `in`、`ref` 或 `out` 参数, 也不能具有引用返回值, 但它可以调用具有此类参数的方法。

如果异步方法的语句指定一个类型的操作数, 则应指定 `Task<TResult>` 作为方法的返回类型 `TResult`。如果当方法完成时未返回有意义的值, 则应使用 `Task`。即, 对方法的调用将返回一个 `Task`, 但是当 `Task` 完成时, 任何等待 `await` 的所有 `Task` 表达式的计算结果都为 `void`。

你应主要使用 `void` 返回类型来定义事件处理程序, 这些处理程序需要此返回类型。`void` 返回异步方法的调用方不能等待, 并且无法捕获该方法引发的异常。

从 C# 7.0 开始，返回另一个类型（通常为值类型），该类型具有 `GetAwaiter` 方法，可尽可能减少性能关键代码段中的内存分配。

有关详细信息和示例，请参阅[异步返回类型](#)。

请参阅

- [AsyncStateMachineAttribute](#)
- [await](#)
- [使用 Async 和 Await 的异步编程](#)
- [在异步任务完成时对其进行处理](#)

const (C# 参考)

2020/11/2 • [Edit Online](#)

使用 `const` 关键字来声明某个常量字段或常量局部变量。常量字段和常量局部变量不是变量并且不能修改。常量可以为数字、布尔值、字符串或 `null` 引用。不要创建常量来表示你需要随时更改的信息。例如，不要使用常量字段来存储服务的价格、产品版本号或公司的品牌名称。这些值会随着时间发生变化；因为编译器会传播常量，所以必须重新编译通过库编译的其他代码以查看更改。另请参阅 `readonly` 关键字。例如：

```
const int X = 0;
public const double GravitationalConstant = 6.673e-11;
private const string ProductName = "Visual C#";
```

备注

常数声明的类型指定声明引入的成员类型。常量局部变量或常量字段的初始值设定项必须是一个可以隐式转换为目标类型的常量表达式。

常数表达式是在编译时可被完全计算的表达式。因此，对于引用类型的常数，可能的值只能是 `string` 和 `null` 引用。

常数声明可以声明多个常数，例如：

```
public const double X = 1.0, Y = 2.0, Z = 3.0;
```

不允许在常数声明中使用修饰符。

常数可以参与常数表达式，如下所示：

```
public const int C1 = 5;
public const int C2 = C1 + 100;
```

NOTE

`readonly` 关键字与 `const` 关键字不同。`const` 字段只能在该字段的声明中初始化。字段可以在声明或构造函数中初始化。因此，根据所使用的构造函数，`readonly` 字段可能具有不同的值。另外，虽然 `const` 字段是编译时常量，但 `readonly` 字段可用于运行时常量，如此行所示：`public static readonly uint l1 = (uint)DateTime.Now.Ticks;`

示例

```
public class ConstTest
{
    class SampleClass
    {
        public int x;
        public int y;
        public const int C1 = 5;
        public const int C2 = C1 + 5;

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

    static void Main()
    {
        var mC = new SampleClass(11, 22);
        Console.WriteLine($"x = {mC.x}, y = {mC.y}");
        Console.WriteLine($"C1 = {SampleClass.C1}, C2 = {SampleClass.C2}");
    }
}
/* Output
   x = 11, y = 22
   C1 = 5, C2 = 10
*/
```

示例

此示例说明如何将常数用作局部变量。

```
public class SealedTest
{
    static void Main()
    {
        const int C = 707;
        Console.WriteLine($"My local constant = {C}");
    }
}
// Output: My local constant = 707
```

C# 语言规范

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

请参阅

- [C# 参考](#)
- [C# 编程指南](#)
- [C# 关键字](#)
- [修饰符](#)
- [readonly](#)

event (C# 参考)

2020/11/2 • [Edit Online](#)

`event` 关键字用于声明发布服务器类中的事件。

示例

下面的示例演示如何声明和引发使用 `EventHandler` 作为基础委托类型的事件。要查看完整代码示例了解如何使用泛型 `EventHandler<TEventArgs>` 委托类型以及如何订阅事件并创建事件处理方法，请参阅[如何发布符合 .NET 准则的事件](#)。

```
public class SampleEventArgs
{
    public SampleEventArgs(string text) { Text = text; }
    public string Text { get; } // readonly
}

public class Publisher
{
    // Declare the delegate (if using non-generic pattern).
    public delegate void SampleEventHandler(object sender, SampleEventArgs e);

    // Declare the event.
    public event SampleEventHandler SampleEvent;

    // Wrap the event in a protected virtual method
    // to enable derived classes to raise the event.
    protected virtual void RaiseSampleEvent()
    {
        // Raise the event in a thread-safe manner using the ?. operator.
        SampleEvent?.Invoke(this, new SampleEventArgs("Hello"));
    }
}
```

事件是一种特殊的多播委托，仅可以从声明事件的类或结构(发布服务器类)中对其进行调用。如果其他类或结构订阅该事件，则在发布服务器类引发该事件时，将调用其事件处理方法。有关详细信息和代码示例，请参阅[事件和委托](#)。

可以将事件标记为`public`、`private`、`protected`、`internal`、`protected internal` 或 `private protected`。这些访问修饰符定义该类的用户访问该事件的方式。有关详细信息，请参阅[访问修饰符](#)。

关键字和事件

下列关键字应用于事件。

III	II	III
<code>static</code>	使事件可供调用方在任何时候进行调用，即使不存在类的实例。	静态类和静态类成员
<code>virtual</code>	允许派生类使用 重写 关键字重写事件行为。	继承
<code>sealed</code>	指定对于派生类，它不再是虚拟的。	

<code>abstract</code>	编译器不会生成 <code>add</code> 和 <code>remove</code> 事件访问器块，因此派生类必须提供其自己的实现。	

可以通过使用[静态](#)关键字将事件声明为静态事件。这可使事件可供调用方在任何时候进行调用，即使不存在类的实例。有关详细信息，请参阅[静态类和静态类成员](#)。

可以通过使用[虚拟](#)关键字将事件标记为虚事件。这可使派生类使用[重写](#)关键字重写事件行为。有关详细信息，请参阅[继承](#)。重写虚拟事件的事件也可以为[密封](#)，指定对于派生类，它不再是虚拟的。最后，可以声明事件为[抽象](#)，这意味着编译器将不会生成 `add` 和 `remove` 事件访问器块。因此，派生类必须提供其自己的实现。

C# 语言规范

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

请参阅

- [C# 参考](#)
- [C# 编程指南](#)
- [C# 关键字](#)
- [add](#)
- [remove](#)
- [修饰符](#)
- [如何合并委托\(多播委托\)](#)

extern (C# 参考)

2021/3/5 • [Edit Online](#)

`extern` 修饰符用于声明在外部实现的方法。`extern` 修饰符的常见用法是在使用 Interop 服务调入非托管代码时与 `[DllImport]` 特性一起使用。在这种情况下，还必须将方法声明为 `static`，如下面的示例所示：

```
[DllImport("avifil32.dll")]
private static extern void AVIFileInit();
```

`extern` 关键字还可以定义外部程序集别名，使得可以从单个程序集中引用同一组件的不同版本。有关详细信息，请参阅[外部别名](#)。

将 `abstract` 和 `extern` 修饰符一起使用来修改同一成员是错误的做法。使用 `extern` 修饰符意味着方法是在 C# 代码的外部实现的，而使用 `abstract` 修饰符意味着类中未提供方法实现。

`extern` 关键字用于 C# 中时会比用于 C++ 中时受到更多的限制。若要比较 C# 关键字与 C++ 关键字，请参见 C++ 语言参考中的“[使用 extern 指定链接](#)”。

示例 1

在此示例中，程序接收来自用户的字符串并将该字符串显示在消息框中。程序使用从 User32.dll 库导入的 `MessageBox` 方法。

```
//using System.Runtime.InteropServices;
class ExternTest
{
    [DllImport("User32.dll", CharSet=CharSet.Unicode)]
    public static extern int MessageBox(IntPtr h, string m, string c, int type);

    static int Main()
    {
        string myString;
        Console.Write("Enter your message: ");
        myString = Console.ReadLine();
        return MessageBox((IntPtr)0, myString, "My Message Box", 0);
    }
}
```

示例 2

此示例阐释了调入 C 库(本机 DLL)的 C# 程序。

1. 创建以下 C 文件并将其命名为 `cmdll.c`：

```
// cmdll.c
// Compile with: -LD
int __declspec(dllexport) SampleMethod(int i)
{
    return i*10;
}
```

2. 从 Visual Studio 安装目录打开 Visual Studio x64(或 x32)本机工具命令提示符窗口，并通过在命令提示符处键入“`cl -LD cmdll.c`”来编译 `cmdll.c` 文件。

3. 在相同的目录中，创建以下 C# 文件并将其命名为 `cm.cs`：

```
// cm.cs
using System;
using System.Runtime.InteropServices;
public class MainClass
{
    [DllImport("Cm.dll")]
    public static extern int SampleMethod(int x);

    static void Main()
    {
        Console.WriteLine("SampleMethod() returns {0}.", SampleMethod(5));
    }
}
```

4. 从 Visual Studio 安装目录打开一个 Visual Studio x64 (或 x32) 本机工具命令提示符窗口，并通过键入以下内容来编译 `cm.cs` 文件：

```
"csc cm.cs" (针对 x64 命令提示符) 或 "csc -platform:x86 cm.cs" (针对 x32 命令提示符)
```

这将创建可执行文件 `cm.exe`。

5. 运行 `cm.exe`。`SampleMethod` 方法将值 5 传递到 DLL 文件，这将返回该值与 10 相乘后的结果。该程序生成以下输出：

```
SampleMethod() returns 50.
```

C# 语言规范

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

请参阅

- [System.Runtime.InteropServices.DllImportAttribute](#)
- [C# 参考](#)
- [C# 编程指南](#)
- [C# 关键字](#)
- [修饰符](#)

in (泛型修饰符) (C# 参考)

2020/11/2 • [Edit Online](#)

对于泛型类型参数，`in` 关键字可指定类型参数是逆变的。可以在泛型接口和委托中使用 `in` 关键字。

逆变使你使用的类型可以比泛型参数指定的类型派生程度更小。这样可以隐式转换实现协变接口的类以及隐式转换委托类型。引用类型支持泛型类型参数中的协变和逆变，但值类型不支持它们。

仅在类型定义方法参数的类型，而不是方法返回类型时，类型可以在泛型接口或委托中声明为逆变。`In`、`ref` 和 `out` 参数必须是固定的，这意味着它们既不为协变又不为逆变。

具有逆变类型参数的接口使其方法接受的参数的类型可以比接口类型参数指定的类型派生程度更小。例如，在 `IComparer<T>` 接口中，类型 `T` 是逆变的，可以将 `IComparer<Person>` 类型的对象分配给 `IComparer<Employee>` 类型的对象，而无需使用任何特殊转换方法（如果 `Employee` 继承 `Person`）。

可以向逆变委托分配相同类型的其他委托，不过要使用派生程度更小的泛型类型参数。

有关详细信息，请参阅[协变和逆变](#)。

逆变泛型接口

下面的示例演示如何声明、扩展和实现逆变泛型接口。它还演示如何对实现此接口的类使用隐式转换。

```
// Contravariant interface.
interface IContravariant<in A> { }

// Extending contravariant interface.
interface IExtContravariant<in A> : IContravariant<A> { }

// Implementing contravariant interface.
class Sample<A> : IContravariant<A> { }

class Program
{
    static void Test()
    {
        IContravariant<Object> iobj = new Sample<Object>();
        IContravariant<String> istr = new Sample<String>();

        // You can assign iobj to istr because
        // the IContravariant interface is contravariant.
        istr = iobj;
    }
}
```

逆变泛型委托

以下示例演示如何声明、实例化和调用逆变泛型委托。它还演示如何隐式转换委托类型。

```
// Contravariant delegate.  
public delegate void DContravariant<in A>(A argument);  
  
// Methods that match the delegate signature.  
public static void SampleControl(Control control)  
{ }  
public static void SampleButton(Button button)  
{ }  
  
public void Test()  
{  
  
    // Instantiating the delegates with the methods.  
    DContravariant<Control> dControl = SampleControl;  
    DContravariant<Button> dButton = SampleButton;  
  
    // You can assign dControl to dButton  
    // because the DContravariant delegate is contravariant.  
    dButton = dControl;  
  
    // Invoke the delegate.  
    dButton(new Button());  
}
```

C# 语言规范

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

请参阅

- [out](#)
- [协变和逆变](#)
- [修饰符](#)

new 修饰符 (C# 参考)

2020/11/2 • [Edit Online](#)

在用作声明修饰符时，`new` 关键字可以显式隐藏从基类继承的成员。隐藏继承的成员时，该成员的派生版本将替换基类版本。虽然可以不使用 `new` 修饰符来隐藏成员，但将收到编译器警告。如果使用 `new` 来显式隐藏成员，将禁止此警告。

`new` 关键字还可用于[创建类型的实例](#)或用作[泛型类型约束](#)。

若要隐藏继承的成员，请使用相同名称在派生类中声明该成员，并使用 `new` 修饰符对其进行修饰。例如：

```
public class BaseC
{
    public int x;
    public void Invoke() { }
}
public class DerivedC : BaseC
{
    new public void Invoke() { }
}
```

在此示例中，使用 `DerivedC.Invoke` 隐藏了 `BaseC.Invoke`。字段 `x` 不受影响，因为未使用类似名称将其隐藏。

通过继承隐藏名称采用下列形式之一：

- 通常，在类或结构中引入的常数、字段、属性或类型会隐藏与其共享名称的所有基类成员。有三种特殊情况。例如，如果将名称为 `N` 的新字段声明为不可调用的类型，并且基类型将 `N` 声明为一种方法，则新字段在调用语法中不会隐藏基声明。有关详细信息，请参阅 [C# 语言规范](#) 中的[成员查找](#)部分。
- 类或结构中引入的方法会隐藏基类中共享该名称的属性、字段和类型。它还会隐藏具有相同签名的所有基类方法。
- 类或结构中引入的索引器会隐藏具有相同签名的所有基类索引器。

对同一成员同时使用 `new` 和 `override` 是错误的做法，因为这两个修饰符的含义互斥。`new` 修饰符会用同样的名称创建一个新成员并使原始成员变为隐藏。`override` 修饰符会扩展继承成员的实现。

在不隐藏继承成员的声明中使用 `new` 修饰符将会生成警告。

示例

在此示例中，基类 `BaseC` 和派生类 `DerivedC` 使用相同的字段名 `x`，从而隐藏了继承字段的值。此示例演示 `new` 修饰符的用法。另外还演示了如何使用完全限定名访问基类的隐藏成员。

```
public class BaseC
{
    public static int x = 55;
    public static int y = 22;
}

public class DerivedC : BaseC
{
    // Hide field 'x'.
    new public static int x = 100;

    static void Main()
    {
        // Display the new value of x:
        Console.WriteLine(x);

        // Display the hidden value of x:
        Console.WriteLine(BaseC.x);

        // Display the unhidden member y:
        Console.WriteLine(y);
    }
}
/*
Output:
100
55
22
*/
```

示例

在此示例中，嵌套类隐藏了基类中同名的类。此示例演示如何使用 `new` 修饰符来消除警告消息，以及如何使用完全限定名来访问隐藏的类成员。

```
public class BaseC
{
    public class NestedC
    {
        public int x = 200;
        public int y;
    }
}

public class DerivedC : BaseC
{
    // Nested type hiding the base type members.
    new public class NestedC
    {
        public int x = 100;
        public int y;
        public int z;
    }

    static void Main()
    {
        // Creating an object from the overlapping class:
        NestedC c1 = new NestedC();

        // Creating an object from the hidden class:
        BaseC.NestedC c2 = new BaseC.NestedC();

        Console.WriteLine(c1.x);
        Console.WriteLine(c2.x);
    }
}
/*
Output:
100
200
*/
```

如果移除 `new` 修饰符，程序仍将编译和运行，但你会收到以下警告：

```
The keyword new is required on 'MyDerivedC.x' because it hides inherited member 'MyBaseC.x'.
```

C# 语言规范

有关详细信息，请参阅 [C# 语言规范](#) 中的 [new 修饰符](#) 部分。

请参阅

- [C# 参考](#)
- [C# 编程指南](#)
- [C# 关键字](#)
- [修饰符](#)
- [使用 Override 和 New 关键字进行版本控制](#)
- [了解何时使用 Override 和 New 关键字](#)

out (泛型修饰符) (C# 参考)

2020/11/2 • [Edit Online](#)

对于泛型类型参数，`out` 关键字可指定类型参数是协变的。可以在泛型接口和委托中使用 `out` 关键字。

协变使你使用的类型可以比泛型参数指定的类型派生程度更大。这样可以隐式转换实现协变接口的类以及隐式转换委托类型。引用类型支持协变和逆变，但值类型不支持它们。

具有协变类型参数的接口使其方法返回的类型可以比类型参数指定的类型派生程度更大。例如，因为在 .NET Framework 4 的 `IEnumerable<T>` 中，类型 `T` 是协变的，所以可以将 `IEnumerable(Of String)` 类型的对象分配给 `IEnumerable(Of Object)` 类型的对象，而无需使用任何特殊转换方法。

可以向协变委托分配相同类型的其他委托，不过要使用派生程度更大的泛型类型参数。

有关详细信息，请参阅[协变和逆变](#)。

示例 - 协变泛型接口

下面的示例演示如何声明、扩展和实现协变泛型接口。它还演示如何对实现协变接口的类使用隐式转换。

```
// Covariant interface.  
interface ICovariant<out R> { }  
  
// Extending covariant interface.  
interface IExtCovariant<out R> : ICovariant<R> { }  
  
// Implementing covariant interface.  
class Sample<R> : ICovariant<R> { }  
  
class Program  
{  
    static void Test()  
    {  
        ICovariant<Object> iobj = new Sample<Object>();  
        ICovariant<String> istr = new Sample<String>();  
  
        // You can assign istr to iobj because  
        // the ICovariant interface is covariant.  
        iobj = istr;  
    }  
}
```

在泛型接口中，如果类型参数满足以下条件，则可以声明为协变：

- 类型参数只用作接口方法的返回类型，而不用作方法参数的类型。

NOTE

此规则有一个例外。如果在协变接口中将逆变泛型委托用作方法参数，则可以将协变类型用作此委托的泛型类型参数。有关协变和逆变泛型委托的详细信息，请参阅[委托中的变体](#)和[对 Func 和 Action 泛型委托使用变体](#)。

- 类型参数不用作接口方法的泛型约束。

示例 - 协变泛型委托

以下示例演示如何声明、实例化和调用协变泛型委托。它还演示如何隐式转换委托类型。

```
// Covariant delegate.  
public delegate R DCovariant<out R>();  
  
// Methods that match the delegate signature.  
public static Control SampleControl()  
{ return new Control(); }  
  
public static Button SampleButton()  
{ return new Button(); }  
  
public void Test()  
{  
    // Instantiate the delegates with the methods.  
    DCovariant<Control> dControl = SampleControl;  
    DCovariant<Button> dButton = SampleButton;  
  
    // You can assign dButton to dControl  
    // because the DCovariant delegate is covariant.  
    dControl = dButton;  
  
    // Invoke the delegate.  
    dControl();  
}
```

在泛型委托中，如果类型仅用作方法返回类型，而不用作方法参数，则可以声明为协变。

C# 语言规范

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

请参阅

- [泛型接口中的变体](#)
- [in](#)
- [修饰符](#)

override (C# 参考)

2020/11/2 • [Edit Online](#)

扩展或修改继承的方法、属性、索引器或事件的抽象或虚拟实现需要 `override` 修饰符。

在以下示例中，`Square` 类必须提供 `GetArea` 的重写实现，因为 `GetArea` 继承自抽象 `Shape` 类：

```
abstract class Shape
{
    public abstract int GetArea();
}

class Square : Shape
{
    int side;

    public Square(int n) => side = n;

    // GetArea method is required to avoid a compile-time error.
    public override int GetArea() => side * side;

    static void Main()
    {
        var sq = new Square(12);
        Console.WriteLine($"Area of the square = {sq.GetArea()}");
    }
}
// Output: Area of the square = 144
```

`override` 方法提供从基类继承的方法的新实现。通过 `override` 声明重写的方法称为重写基方法。`override` 方法必须具有与重写基方法相同的签名。从 C# 9.0 开始，`override` 方法支持协变返回类型。具体而言，`override` 方法的返回类型可从相应基方法的返回类型派生。在 C# 8.0 和更早版本中，`override` 方法和重写基方法的返回类型必须相同。

不能重写非虚方法或静态方法。重写基方法必须是 `virtual`、`abstract` 或 `override`。

`override` 声明不能更改 `virtual` 方法的可访问性。`override` 方法和 `virtual` 方法必须具有相同级别访问修饰符。

不能使用 `new`、`static` 或 `virtual` 修饰符修改 `override` 方法。

重写属性声明必须指定与继承的属性完全相同的访问修饰符、类型和名称。从 C# 9.0 开始，只读重写属性支持协变返回类型。重写属性必须为 `virtual`、`abstract` 或 `override`。

有关如何使用 `override` 关键字的详细信息，请参阅[使用 Override 和 New 关键字进行版本控制](#)和[了解何时使用 Override 和 New 关键字](#)。有关继承的信息，请参阅[继承](#)。

示例

此示例定义一个名为 `Employee` 的基类和一个名为 `SalesEmployee` 的派生类。`SalesEmployee` 类包含一个额外字段 `salesbonus`，并且重写方法 `CalculatePay` 以将它考虑在内。

```

class TestOverride
{
    public class Employee
    {
        public string name;

        // Basepay is defined as protected, so that it may be
        // accessed only by this class and derived classes.
        protected decimal basepay;

        // Constructor to set the name and basepay values.
        public Employee(string name, decimal basepay)
        {
            this.name = name;
            this.basepay = basepay;
        }

        // Declared virtual so it can be overridden.
        public virtual decimal CalculatePay()
        {
            return basepay;
        }
    }

    // Derive a new class from Employee.
    public class SalesEmployee : Employee
    {
        // New field that will affect the base pay.
        private decimal salesbonus;

        // The constructor calls the base-class version, and
        // initializes the salesbonus field.
        public SalesEmployee(string name, decimal basepay,
                             decimal salesbonus) : base(name, basepay)
        {
            this.salesbonus = salesbonus;
        }

        // Override the CalculatePay method
        // to take bonus into account.
        public override decimal CalculatePay()
        {
            return basepay + salesbonus;
        }
    }

    static void Main()
    {
        // Create some new employees.
        var employee1 = new SalesEmployee("Alice",
                                         1000, 500);
        var employee2 = new Employee("Bob", 1200);

        Console.WriteLine($"Employee1 {employee1.name} earned: {employee1.CalculatePay()}");
        Console.WriteLine($"Employee2 {employee2.name} earned: {employee2.CalculatePay()}");
    }
}
/*
Output:
Employee1 Alice earned: 1500
Employee2 Bob earned: 1200
*/

```

C# 语言规范

有关详细信息，请参阅 [C# 语言规范](#)中的**重写方法**部分。

有关协变返回类型的详细信息，请参阅[功能建议说明](#)。

另请参阅

- [C# 参考](#)
- [继承](#)
- [C# 关键字](#)
- [修饰符](#)
- [abstract](#)
- [virtual](#)
- [new\(修饰符\)](#)
- [多态性](#)

readonly (C# 参考)

2020/11/2 • [Edit Online](#)

`readonly` 关键字是一个可在四个上下文中使用的修饰符：

- 在 [字段声明](#) 中，`readonly` 指示只能在声明期间或在同一个类的构造函数中向字段赋值。可以在字段声明和构造函数中多次分配和重新分配只读字段。

构造函数退出后，不能分配 `readonly` 字段。此规则对于值类型和引用类型具有不同的含义：

- 由于值类型直接包含数据，因此属于 `readonly` 值类型的字段不可变。
- 由于引用类型包含对其数据的引用，因此属于 `readonly` 引用类型的字段必须始终引用同一对象。该对象是可变的。`readonly` 修饰符可防止字段替换为引用类型的其他实例。但是，修饰符不会阻止通过只读字段修改字段的实例数据。

WARNING

包含属于可变引用类型的外部可见只读字段的外部可见类型可能存在安全漏洞，可能会触发警告 CA2104：“不要声明只读可变引用类型。”

- 在 `readonly struct` 类型定义中，`readonly` 指示结构类型是不可变的。有关详细信息，请参阅[结构类型](#)一文中的 `readonly` 结构一节。
- 在结构类型内的实例成员声明中，`readonly` 指示实例成员不修改结构的状态。有关详细信息，请参阅[结构类型](#)一文中的 `readonly` 实例成员部分。
- 在 `ref readonly` 方法返回中，`readonly` 修饰符指示该方法返回一个引用，且不允许向该引用写入内容。

在 C# 7.2 中添加了 `readonly struct` 和 `ref readonly` 上下文。在 C# 8.0 中添加了 `readonly` 结构成员

Readonly 字段示例

在此示例中，即使在类构造函数中给字段 `year` 赋了值，也无法在方法 `ChangeYear` 中更改其值：

```
class Age
{
    readonly int year;
    Age(int year)
    {
        this.year = year;
    }
    void ChangeYear()
    {
        //year = 1967; // Compile error if uncommented.
    }
}
```

只能在下列上下文中对 `readonly` 字段进行赋值：

- 在声明中初始化变量时，例如：

```
public readonly int y = 5;
```

- 在包含实例字段声明的类的实例构造函数中。
- 在包含静态字段声明的类的静态构造函数中。

只有在这些构造函数上下文中，将 `readonly` 字段作为 `out` 或 `ref` 参数传递才有效。

NOTE

`readonly` 关键字不同于 `const` 关键字。`const` 字段只能在该字段的声明中初始化。可以在字段声明和任何构造函数中多次分配 `readonly` 字段。因此，根据所使用的构造函数，`readonly` 字段可能具有不同的值。另外，虽然 `const` 字段是编译时常量，但 `readonly` 字段可用于运行时常量，如下面的示例所示：

```
public static readonly uint timeStamp = (uint)DateTime.Now.Ticks;
```

```
public class SamplePoint
{
    public int x;
    // Initialize a readonly field
    public readonly int y = 25;
    public readonly int z;

    public SamplePoint()
    {
        // Initialize a readonly instance field
        z = 24;
    }

    public SamplePoint(int p1, int p2, int p3)
    {
        x = p1;
        y = p2;
        z = p3;
    }

    public static void Main()
    {
        SamplePoint p1 = new SamplePoint(11, 21, 32);    // OK
        Console.WriteLine($"p1: x={p1.x}, y={p1.y}, z={p1.z}");
        SamplePoint p2 = new SamplePoint();
        p2.x = 55;    // OK
        Console.WriteLine($"p2: x={p2.x}, y={p2.y}, z={p2.z}");
    }
    /*
    Output:
    p1: x=11, y=21, z=32
    p2: x=55, y=25, z=24
    */
}
```

在前面的示例中，如果使用类似以下示例的语句：

```
p2.y = 66;      // Error
```

你将收到编译器错误消息：

无法对只读的字段赋值（构造函数或变量初始值指定项中除外）

Ref readonly 返回示例

`ref return` 上的 `readonly` 修饰符指示返回的引用无法修改。下面的示例返回了一个对来源的引用。它使用 `readonly` 修饰符来指示调用方无法修改来源：

```
private static readonly SamplePoint origin = new SamplePoint(0, 0, 0);
public static ref readonly SamplePoint Origin => ref origin;
```

所返回的类型不需要为 `readonly struct`。`ref` 能返回的任何类型都能由 `ref readonly` 返回。

C# 语言规范

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

你还可查看语言规范建议：

- [readonly ref 和 readonly 结构](#)
- [readonly 结构成员](#)

请参阅

- [C# 参考](#)
- [C# 编程指南](#)
- [C# 关键字](#)
- [修饰符](#)
- [const](#)
- [字段](#)

sealed (C# 参考)

2020/11/2 • [Edit Online](#)

应用于某个类时，`sealed` 修饰符可阻止其他类继承自该类。在下面的示例中，类 `B` 继承自类 `A`，但没有类可以继承自类 `B`。

```
class A {}  
sealed class B : A {}
```

还可以对替代基类中的虚方法或属性的方法或属性使用 `sealed` 修饰符。这使你可以允许类派生自你的类并防止它们替代特定虚方法或属性。

示例

在下面的示例中，`Z` 继承自 `Y`，但 `Z` 无法替代在 `X` 中声明并在 `Y` 中密封的虚函数 `F`。

```
class X  
{  
    protected virtual void F() { Console.WriteLine("X.F"); }  
    protected virtual void F2() { Console.WriteLine("X.F2"); }  
}  
  
class Y : X  
{  
    sealed protected override void F() { Console.WriteLine("Y.F"); }  
    protected override void F2() { Console.WriteLine("Y.F2"); }  
}  
  
class Z : Y  
{  
    // Attempting to override F causes compiler error CS0239.  
    // protected override void F() { Console.WriteLine("Z.F"); }  
  
    // Overriding F2 is allowed.  
    protected override void F2() { Console.WriteLine("Z.F2"); }  
}
```

在类中定义新方法或属性时，可以通过不将它们声明为虚拟，来防止派生类替代它们。

将 `abstract` 修饰符与密封类结合使用是错误的，因为抽象类必须由提供抽象方法或属性的实现的类来继承。

应用于方法或属性时，`sealed` 修饰符必须始终与 `override` 结合使用。

因为结构是隐式密封的，所以无法继承它们。

有关详细信息，请参阅[继承](#)。

有关更多示例，请参阅[抽象类、密封类及类成员](#)。

示例

```
sealed class SealedClass
{
    public int x;
    public int y;
}

class SealedTest2
{
    static void Main()
    {
        var sc = new SealedClass();
        sc.x = 110;
        sc.y = 150;
        Console.WriteLine($"x = {sc.x}, y = {sc.y}");
    }
}
// Output: x = 110, y = 150
```

在上面的示例中，可能会尝试使用以下语句从密封类继承：

```
class MyDerivedC: SealedClass {} // Error
```

结果是出现错误消息：

```
'MyDerivedC': cannot derive from sealed type 'SealedClass'
```

备注

若要确定是否密封类、方法或属性，通常应考虑以下两点：

- 派生类通过可以自定义类而可能获得的潜在好处。
- 派生类可能采用使它们无法再正常工作或按预期工作的方式来修改类的可能性。

C# 语言规范

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

请参阅

- [C# 参考](#)
- [C# 编程指南](#)
- [C# 关键字](#)
- [静态类和静态类成员](#)
- [抽象类、密封类及类成员](#)
- [访问修饰符](#)
- [修饰符](#)
- [override](#)
- [virtual](#)

static (C# 参考)

2020/11/2 • [Edit Online](#)

本页介绍 `static` 修饰符关键字。`static` 关键字也是 `using static` 指令的一部分。

使用 `static` 修饰符可声明属于类型本身而不是属于特定对象的静态成员。`static` 修饰符可用于声明 `static` 类。在类、接口和结构中，可以将 `static` 修饰符添加到字段、方法、属性、运算符、事件和构造函数。`static` 修饰符不能用于索引器或终结器。有关详细信息，请参阅[静态类和静态类成员](#)。

从 C# 8.0 开始，可以将 `static` 修饰符添加到[本地函数](#)。静态本地函数无法捕获局部变量或实例状态。

从 C# 9.0 开始，可将 `static` 修饰符添加到[Lambda 表达式或匿名方法](#)。静态Lambda 表达式或匿名方法无法捕获局部变量或实例状态。

示例 - 静态类

下面的类声明为 `static` 并且只含 `static` 方法：

```
static class CompanyEmployee
{
    public static void DoSomething() { /*...*/ }
    public static void DoSomethingElse() { /*...*/ }
}
```

常数或类型声明是隐式的 `static` 成员。不能通过实例引用 `static` 成员。然而，可以通过类型名称引用它。例如，请考虑以下类：

```
public class MyBaseC
{
    public struct MyStruct
    {
        public static int x = 100;
    }
}
```

若要引用 `static` 成员 `x`，除非可从相同范围访问该成员，否则请使用完全限定的名称 `MyBaseC.MyStruct.x`：

```
Console.WriteLine(MyBaseC.MyStruct.x);
```

尽管类的实例包含该类的所有实例字段的单独副本，但每个 `static` 字段只有一个副本。

不可以使用 `this` 引用 `static` 方法或属性访问器。

如果 `static` 关键字应用于类，则类的所有成员都必须为 `static`。

类、接口和 `static` 类可以具有 `static` 构造函数。在程序开始和实例化类之间的某个时刻调用 `static` 构造函数。

NOTE

`static` 关键字比用于 C++ 中时受到的限制更多。若要与 C++ 关键字进行比较, 请参阅 [Storage classes \(C++\)](#)(存储类 (C++))。

若要演示 `static` 成员, 请考虑表示公司员工的类。假定此类包含计数员工的方法和存储员工人数的字段。方法和字段均不属于任何一个员工实例。相反, 它们属于全体员工这个类。应将其声明为该类的 `static` 成员。

示例 - 静态字段和方法

此示例读取新员工的姓名和 ID, 员工计数器按 1 递增, 并显示新员工信息和新员工人数。此程序从键盘读取员工的当前人数。

```

public class Employee4
{
    public string id;
    public string name;

    public Employee4()
    {
    }

    public Employee4(string name, string id)
    {
        this.name = name;
        this.id = id;
    }

    public static int employeeCounter;

    public static int AddEmployee()
    {
        return ++employeeCounter;
    }
}

class MainClass : Employee4
{
    static void Main()
    {
        Console.Write("Enter the employee's name: ");
        string name = Console.ReadLine();
        Console.Write("Enter the employee's ID: ");
        string id = Console.ReadLine();

        // Create and configure the employee object.
        Employee4 e = new Employee4(name, id);
        Console.WriteLine("Enter the current number of employees: ");
        string n = Console.ReadLine();
        Employee4.employeeCounter = Int32.Parse(n);
        Employee4.AddEmployee();

        // Display the new information.
        Console.WriteLine($"Name: {e.name}");
        Console.WriteLine($"ID: {e.id}");
        Console.WriteLine($"New Number of Employees: {Employee4.employeeCounter}");
    }
}
/*
Input:
Matthias Berndt
AF643G
15
*
Sample Output:
Enter the employee's name: Matthias Berndt
Enter the employee's ID: AF643G
Enter the current number of employees: 15
Name: Matthias Berndt
ID: AF643G
New Number of Employees: 16
*/

```

示例 - 静态初始化

此示例演示了如何使用尚未声明的 `static` 字段来初始化另一个 `static` 字段。在向 `static` 字段显式赋值之后才会定义结果。

```
class Test
{
    static int x = y;
    static int y = 5;

    static void Main()
    {
        Console.WriteLine(Test.x);
        Console.WriteLine(Test.y);

        Test.x = 99;
        Console.WriteLine(Test.x);
    }
}
/*
Output:
0
5
99
*/
```

C# 语言规范

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

请参阅

- [C# 参考](#)
- [C# 编程指南](#)
- [C# 关键字](#)
- [修饰符](#)
- [using static 指令](#)
- [静态类和静态类成员](#)

unsafe (C# 参考)

2021/5/7 • [Edit Online](#)

`unsafe` 关键字表示不安全上下文，该上下文是任何涉及指针的操作所必需的。有关详细信息，请参阅[不安全代码和指针](#)。

可在类型或成员的声明中使用 `unsafe` 修饰符。因此，类型或成员的整个正文范围均被视为不安全上下文。以下面使用 `unsafe` 修饰符声明的方法为例：

```
unsafe static void FastCopy(byte[] src, byte[] dst, int count)
{
    // Unsafe context: can use pointers here.
}
```

不安全上下文的范围从参数列表扩展到方法的结尾，因此也可在以下参数列表中使用指针：

```
unsafe static void FastCopy ( byte* ps, byte* pd, int count ) {...}
```

还可以使用不安全块从而能够使用该块内的不安全代码。例如：

```
unsafe
{
    // Unsafe context: can use pointers here.
}
```

若要编译不安全代码，必须指定 [AllowUnsafeBlocks](#) 编译器选项。不能通过公共语言运行时验证不安全代码。

示例

```
// compile with: -unsafe
class UnsafeTest
{
    // Unsafe method: takes pointer to int.
    unsafe static void SquarePtrParam(int* p)
    {
        *p *= *p;
    }

    unsafe static void Main()
    {
        int i = 5;
        // Unsafe method: uses address-of operator (&).
        SquarePtrParam(&i);
        Console.WriteLine(i);
    }
}
// Output: 25
```

C# 语言规范

有关详细信息，请参阅 [C# 语言规范](#) 中的[不安全代码](#)。该语言规范是 C# 语法和用法的权威资料。

请参阅

- [C# 参考](#)
- [C# 编程指南](#)
- [C# 关键字](#)
- [fixed 语句](#)
- [不安全代码和指针](#)
- [固定大小的缓冲区](#)

virtual (C# 参考)

2020/11/2 • [Edit Online](#)

`virtual` 关键字用于修改方法、属性、索引器或事件声明，并使它们可以在派生类中被重写。例如，此方法可被任何继承它的类替代：

```
public virtual double Area()
{
    return x * y;
}
```

虚拟成员的实现可由派生类中的[替代成员](#)更改。有关如何使用 `virtual` 关键字的详细信息，请参阅[使用 Override 和 New 关键字进行版本控制](#)和[了解何时使用 Override 和 New 关键字](#)。

备注

调用虚拟方法时，将为替代的成员检查该对象的运行时类型。将调用大部分派生类中的该替代成员，如果没有派生类替代该成员，则它可能是原始成员。

默认情况下，方法是非虚拟的。不能替代非虚方法。

`virtual` 修饰符不能与 `static`、`abstract`private` 或 `override` 修饰符一起使用。以下示例显示了虚拟属性：

```

class MyBaseClass
{
    // virtual auto-implemented property. Overrides can only
    // provide specialized behavior if they implement get and set accessors.
    public virtual string Name { get; set; }

    // ordinary virtual property with backing field
    private int num;
    public virtual int Number
    {
        get { return num; }
        set { num = value; }
    }
}

class MyDerivedClass : MyBaseClass
{
    private string name;

    // Override auto-implemented property with ordinary property
    // to provide specialized accessor behavior.
    public override string Name
    {
        get
        {
            return name;
        }
        set
        {
            if (!string.IsNullOrEmpty(value))
            {
                name = value;
            }
            else
            {
                name = "Unknown";
            }
        }
    }
}

```

除声明和调用语法不同外，虚拟属性的行为与虚拟方法相似。

- 在静态属性上使用 `virtual` 修饰符是错误的。
- 通过包括使用 `override` 修饰符的属性声明，可在派生类中替代虚拟继承属性。

示例

在该示例中，`Shape` 类包含 `x`、`y` 两个坐标和 `Area()` 虚拟方法。不同的形状类(如 `Circle`、`Cylinder` 和 `Sphere`)继承 `Shape` 类，并为每个图形计算表面积。每个派生类都有各自的 `Area()` 替代实现。

请注意，继承的类 `Circle`^Sphere` 和 `Cylinder` 均使用初始化基类的构造函数，如下面的声明中所示。

```
public Cylinder(double r, double h): base(r, h) {}
```

根据与方法关联的对象，下面的程序通过调用 `Area()` 方法的相应实现来计算并显示每个对象的相应区域。

```

class TestClass
{
    public class Shape
    {
        double x, y;
    }
}
```

```

public const double PI = Math.PI;
protected double x, y;

public Shape()
{
}

public Shape(double x, double y)
{
    this.x = x;
    this.y = y;
}

public virtual double Area()
{
    return x * y;
}

public class Circle : Shape
{
    public Circle(double r) : base(r, 0)
    {
    }

    public override double Area()
    {
        return PI * x * x;
    }
}

class Sphere : Shape
{
    public Sphere(double r) : base(r, 0)
    {
    }

    public override double Area()
    {
        return 4 * PI * x * x;
    }
}

class Cylinder : Shape
{
    public Cylinder(double r, double h) : base(r, h)
    {
    }

    public override double Area()
    {
        return 2 * PI * x * x + 2 * PI * x * y;
    }
}

static void Main()
{
    double r = 3.0, h = 5.0;
    Shape c = new Circle(r);
    Shape s = new Sphere(r);
    Shape l = new Cylinder(r, h);
    // Display results.
    Console.WriteLine("Area of Circle    = {0:F2}", c.Area());
    Console.WriteLine("Area of Sphere    = {0:F2}", s.Area());
    Console.WriteLine("Area of Cylinder = {0:F2}", l.Area());
}
/*
Output:

```

```
Area of Circle    = 28.27
Area of Sphere   = 113.10
Area of Cylinder = 150.80
*/
```

C# 语言规范

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

请参阅

- [多态性](#)
- [abstract](#)
- [override](#)
- [new\(修饰符\)](#)

volatile (C# 参考)

2020/11/2 • [Edit Online](#)

`volatile` 关键字指示一个字段可以由多个同时执行的线程修改。出于性能原因，编译器，运行时系统甚至硬件都可能重新排列对存储器位置的读取和写入。声明了 `volatile` 的字段不进行这些优化。添加 `volatile` 修饰符可确保所有线程观察易失性写入操作(由任何其他线程执行)时的观察顺序与写入操作的执行顺序一致。不确保从所有执行线程整体来看时所有易失性写入操作均按执行顺序排序。

`volatile` 关键字可应用于以下类型的字段：

- 引用类型。
- 指针类型(在不安全的上下文中)。请注意，虽然指针本身可以是可变的，但是它指向的对象不能是可变的。
换句话说，不能声明“指向可变对象的指针”。
- 简单类型，如 `sbyte`、`byte`、`short`、`ushort`、`int`、`uint`、`char`、`float` 和 `bool`。
- 具有以下基本类型之一的 `enum` 类型：`byte`、`sbyte`、`short`、`ushort`、`int` 或 `uint`。
- 已知为引用类型的泛型类型参数。
- `IntPtr` 和 `UIntPtr`。

其他类型(包括 `double` 和 `long`)无法标记为 `volatile`，因为对这些类型的字段的读取和写入不能保证是原子的。若要保护对这些类型字段的多线程访问，请使用 `Interlocked` 类成员或使用 `lock` 语句保护访问权限。

`volatile` 关键字只能应用于 `class` 或 `struct` 的字段。不能将局部变量声明为 `volatile`。

示例

下面的示例说明如何将公共字段变量声明为 `volatile`。

```
class VolatileTest
{
    public volatile int sharedStorage;

    public void Test(int _i)
    {
        sharedStorage = _i;
    }
}
```

下面的示例演示如何创建辅助线程，并用它与主线程并行执行处理。有关多线程处理的详细信息，请参阅[托管线程处理](#)。

```

public class Worker
{
    // This method is called when the thread is started.
    public void DoWork()
    {
        bool work = false;
        while (!_shouldStop)
        {
            work = !work; // simulate some work
        }
        Console.WriteLine("Worker thread: terminating gracefully.");
    }
    public void RequestStop()
    {
        _shouldStop = true;
    }
    // Keyword volatile is used as a hint to the compiler that this data
    // member is accessed by multiple threads.
    private volatile bool _shouldStop;
}

public class WorkerThreadExample
{
    public static void Main()
    {
        // Create the worker thread object. This does not start the thread.
        Worker workerObject = new Worker();
        Thread workerThread = new Thread(workerObject.DoWork);

        // Start the worker thread.
        workerThread.Start();
        Console.WriteLine("Main thread: starting worker thread...");

        // Loop until the worker thread activates.
        while (!workerThread.IsAlive)
            ;

        // Put the main thread to sleep for 500 milliseconds to
        // allow the worker thread to do some work.
        Thread.Sleep(500);

        // Request that the worker thread stop itself.
        workerObject.RequestStop();

        // Use the Thread.Join method to block the current thread
        // until the object's thread terminates.
        workerThread.Join();
        Console.WriteLine("Main thread: worker thread has terminated.");
    }
    // Sample output:
    // Main thread: starting worker thread...
    // Worker thread: terminating gracefully.
    // Main thread: worker thread has terminated.
}

```

将 `volatile` 修饰符添加到 `_shouldStop` 的声明后，将始终获得相同的结果（类似于前面代码中显示的片段）。但是，如果 `_shouldStop` 成员上没有该修饰符，则行为是不可预测的。`DoWork` 方法可能会优化成员访问，从而导致读取陈旧数据。鉴于多线程编程的性质，读取陈旧数据的次数是不可预测的。不同的程序运行会产生一些不同的结果。

C# 语言规范

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

请参阅

- [C# 语言规范: 可变关键字](#)
- [C# 参考](#)
- [C# 编程指南](#)
- [C# 关键字](#)
- [修饰符](#)
- [lock 语句](#)
- [Interlocked](#)

语句关键字 (C# 参考)

2020/11/2 • [Edit Online](#)

语句是程序指令。除了下表中的引用主题介绍的内容外，语句都是按照顺序执行。下表列出了 C# 语句关键字。有关不用任何关键字表示的语句的详细信息，请参阅[语句](#)。

II	C# III
选择语句	if 、 else 、 switch 、 case
迭代语句	do 、 for 、 foreach 、 in 、 while
跳转语句	break 、 continue 、 default 、 goto 、 return 、 yield
异常处理语句	throw 、 try-catch 、 try-finally 、 try-catch-finally
Checked 和 unchecked	checked 、 unchecked
fixed 语句	fixed
lock 语句	lock

请参阅

- [C# 参考](#)
- [语句](#)
- [C# 关键字](#)

if-else (C# 参考)

2020/11/2 • [Edit Online](#)

`if` 语句基于布尔表达式的值来识别运行哪个语句。在下面的示例中，`bool` 变量 `condition` 已被设置为 `true`，然后被签入到了 `if` 语句。输出为 `The variable is set to true.`。

```
bool condition = true;

if (condition)
{
    Console.WriteLine("The variable is set to true.");
}
else
{
    Console.WriteLine("The variable is set to false.");
}
```

你可以通过将本主题中的示例放入控制台应用的 `Main` 方法中来运行它们。

C# 中的 `if` 语句可以采用两种形式，如以下示例所示。

```
// if-else statement
if (condition)
{
    then-statement;
}
else
{
    else-statement;
}
// Next statement in the program.

// if statement without an else
if (condition)
{
    then-statement;
}
// Next statement in the program.
```

在 `if-else` 语句中，如果 `condition` 计算结果为 `true`，则 `then-statement` 将运行。如果 `condition` 为 `false`，则 `else-statement` 将运行。由于 `condition` 不能同时为 `true` 和 `false`，因此，`then-statement` 语句的 `else-statement` 和 `if-else` 永远不能同时运行。`then-statement` 或 `else-statement` 运行后，控件将转移到 `if` 语句之后的下一个语句。

在不包括 `if` 语句的 `else` 语句中，如果 `condition` 为 `true`，则 `then-statement` 将运行。如果 `condition` 为 `false`，则控件将转移到 `if` 语句之后的下一个语句。

`then-statement` 和 `else-statement` 都可由单个语句或包含在括号中 (`{}`) 的多个语句组成。对于单个语句，括号是可选的，但建议选择。

`then-statement` 和 `else-statement` 中的语句可为任何类型，包括嵌套在原始 `if` 语句中的另一个 `if` 语句。在嵌套的 `if` 语句中，每个 `else` 子句都属于上一个无相应 `if` 的 `else`。在下面的示例中，如果 `Result1` 和 `m > 10` 计算结果都为 `true`，则将显示 `n > 20`。如果 `m > 10` 为 `true` 但 `n > 20` 为 `false`，则将显示 `Result2`。

```
// Try with m = 12 and then with m = 8.  
int m = 12;  
int n = 18;  
  
if (m > 10)  
    if (n > 20)  
    {  
        Console.WriteLine("Result1");  
    }  
else  
{  
    Console.WriteLine("Result2");  
}
```

相反，如果你希望在 `Result2` 为 `false` 的时候显示 `(m > 10)`，则可以通过使用括号来指定此关联，以建立嵌套的 `if` 语句的开头和结尾，如以下示例所示。

```
// Try with m = 12 and then with m = 8.  
if (m > 10)  
{  
    if (n > 20)  
        Console.WriteLine("Result1");  
}  
else  
{  
    Console.WriteLine("Result2");  
}
```

如果条件 `(m > 10)` 的计算结果为 `false`，则显示 `Result2`。

示例

在下例中，当通过键盘输入字符时，该程序将使用嵌套的 `if` 语句来确定输入的字符是否为字母字符。如果输入的字符是字母字符，则程序将检查输入的字符是大写还是小写。每种情况都会显示一条消息。

```

Console.Write("Enter a character: ");
char c = (char)Console.Read();
if (Char.IsLetter(c))
{
    if (Char.ToLower(c))
    {
        Console.WriteLine("The character is lowercase.");
    }
    else
    {
        Console.WriteLine("The character is uppercase.");
    }
}
else
{
    Console.WriteLine("The character isn't an alphabetic character.");
}

//Sample Output:

//Enter a character: 2
//The character isn't an alphabetic character.

//Enter a character: A
//The character is uppercase.

//Enter a character: h
//The character is lowercase.

```

示例

你也可以将 `if` 语句嵌套到 `else` 块中，如以下部分代码所示。示例将 `if` 语句嵌套在两个 `else` 块和一个 `then` 块中。注释指定每个块中哪些条件为 true 哪些条件为 false。

```

// Change the values of these variables to test the results.
bool Condition1 = true;
bool Condition2 = true;
bool Condition3 = true;
bool Condition4 = true;

if (Condition1)
{
    // Condition1 is true.
}
else if (Condition2)
{
    // Condition1 is false and Condition2 is true.
}
else if (Condition3)
{
    if (Condition4)
    {
        // Condition1 and Condition2 are false. Condition3 and Condition4 are true.
    }
    else
    {
        // Condition1, Condition2, and Condition4 are false. Condition3 is true.
    }
}
else
{
    // Condition1, Condition2, and Condition3 are false.
}

```

示例

下面的示例确定了输入的字符是一个小写字母，还是大写字母，还是一个数字。如果所有三个条件都为 false，该字符不是字母数字字符。此示例显示了每种情况的消息内容。

```
Console.WriteLine("Enter a character: ");
char ch = (char)Console.Read();

if (Char.IsUpper(ch))
{
    Console.WriteLine("The character is an uppercase letter.");
}
else if (Char.IsLower(ch))
{
    Console.WriteLine("The character is a lowercase letter.");
}
else if (Char.IsDigit(ch))
{
    Console.WriteLine("The character is a number.");
}
else
{
    Console.WriteLine("The character is not alphanumeric.");
}

//Sample Input and Output:
//Enter a character: E
//The character is an uppercase letter.

//Enter a character: e
//The character is a lowercase letter.

//Enter a character: 4
//The character is a number.

//Enter a character: =
//The character is not alphanumeric.
```

正如 else 块或 then 块中的语句可以是任何有效的语句一样，你可以将任何有效的布尔表达式用于此条件。可使用 `!`、`&&`、`||`、`&`、`|` 和 `^` 等逻辑运算符来创建复合条件。下面的代码演示了示例。

```
// NOT
bool result = true;
if (!result)
{
    Console.WriteLine("The condition is true (result is false).");
}
else
{
    Console.WriteLine("The condition is false (result is true).");
}

// Short-circuit AND
int m = 9;
int n = 7;
int p = 5;
if (m >= n && m >= p)
{
    Console.WriteLine("Nothing is larger than m.");
}

// AND and NOT
if (m >= n && !(p > m))
{
    Console.WriteLine("Nothing is larger than m.");
}

// Short-circuit OR
if (m > n || m > p)
{
    Console.WriteLine("m isn't the smallest.");
}

// NOT and OR
m = 4;
if (!(m >= n || m >= p))
{
    Console.WriteLine("Now m is the smallest.");
}
// Output:
// The condition is false (result is true).
// Nothing is larger than m.
// Nothing is larger than m.
// m isn't the smallest.
// Now m is the smallest.
```

C# 语言规范

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

请参阅

- [C# 参考](#)
- [C# 编程指南](#)
- [C# 关键字](#)
- [?:运算符](#)
- [if-else 语句 \(C++\)](#)
- [switch](#)

switch (C# 参考)

2020/11/2 • [Edit Online](#)

本文介绍 `switch` 语句。有关 `switch` 表达式(在 C# 8.0 中引入)的信息, 请参阅 [表达式和运算符部分中有关 switch 表达式](#) 的文章。

`switch` 是一个选择语句, 它根据与匹配表达式匹配的模式, 从候选列表中选择单个开关部分进行执行。

```
using System;

public class Example
{
    public static void Main()
    {
        int caseSwitch = 1;

        switch (caseSwitch)
        {
            case 1:
                Console.WriteLine("Case 1");
                break;
            case 2:
                Console.WriteLine("Case 2");
                break;
            default:
                Console.WriteLine("Default case");
                break;
        }
    }
}

// The example displays the following output:
//      Case 1
```

如果针对 3 个或更多条件测试单个表达式, `switch` 语句通常用作 [if-else](#) 构造的替代项。例如, 以下 `switch` 语句确定类型为 `color` 的变量是否具有三个值之一:

```

using System;

public enum Color { Red, Green, Blue }

public class Example
{
    public static void Main()
    {
        Color c = (Color) (new Random()).Next(0, 3);
        switch (c)
        {
            case Color.Red:
                Console.WriteLine("The color is red");
                break;
            case Color.Green:
                Console.WriteLine("The color is green");
                break;
            case Color.Blue:
                Console.WriteLine("The color is blue");
                break;
            default:
                Console.WriteLine("The color is unknown.");
                break;
        }
    }
}

```

它相当于使用 `if - else` 构造的以下示例。

```

using System;

public enum Color { Red, Green, Blue }

public class Example
{
    public static void Main()
    {
        Color c = (Color) (new Random()).Next(0, 3);
        if (c == Color.Red)
            Console.WriteLine("The color is red");
        else if (c == Color.Green)
            Console.WriteLine("The color is green");
        else if (c == Color.Blue)
            Console.WriteLine("The color is blue");
        else
            Console.WriteLine("The color is unknown.");
    }
}

// The example displays the following output:
//      The color is red

```

匹配表达式

匹配表达式提供与 `case` 标签中的模式相匹配的值。语法为：

```
switch (expr)
```

在 C# 6 及更低版本中，匹配表达式必须是返回以下类型值的表达式：

- **字符型**。
- **字符串**。

- `bool`。
- 整数值, 例如 `int` 或 `long`。
- 枚举值。

从 C# 7.0 开始, 匹配表达式可以是任何非 null 表达式。

开关部分

`switch` 语句包含一个或多个开关部分。每个 `switch` 部分包含一个或多个 `case` 标签(`case` 或 `default` 标签), 后接一个或多个语句。`switch` 语句最多可包含一个置于任何 `switch` 部分中的 `default` 标签。以下示例显示了一个简单的 `switch` 语句, 该语句包含三个 `switch` 部分, 每个部分包含两个语句。第二个 `switch` 部分包含 `case 2:` 和 `case 3:` 标签。

`switch` 语句中可以包含任意数量的开关部分, 每个开关部分可以具有一个或多个 `case` 标签, 如以下示例所示。但是, 任何两个 `case` 标签不可包含相同的表达式。

```
using System;

public class Example
{
    public static void Main()
    {
        Random rnd = new Random();
        int caseSwitch = rnd.Next(1,4);

        switch (caseSwitch)
        {
            case 1:
                Console.WriteLine("Case 1");
                break;
            case 2:
            case 3:
                Console.WriteLine($"Case {caseSwitch}");
                break;
            default:
                Console.WriteLine($"An unexpected value ({caseSwitch})");
                break;
        }
    }
}

// The example displays output like the following:
//      Case 1
```

`switch` 语句执行中只有一个开关部分。C# 禁止从一个 `switch` 部分继续执行到下一个 `switch` 部分。因此, 下面的代码生成编译器错误 CS0163:“控件不能从一个 `case` 标签 (<code label>) 贯穿到另一个 `case` 标签。”

```
switch (caseSwitch)
{
    // The following switch section causes an error.
    case 1:
        Console.WriteLine("Case 1...");
        // Add a break or other jump statement here.
    case 2:
        Console.WriteLine("... and/or Case 2");
        break;
}
```

通常通过使用 `break`、`goto` 或 `return` 语句显式退出开关部分来满足此要求。不过, 以下代码也有效, 因为它确保程序控制权无法贯穿到 `default` `switch` 部分。

```

switch (caseSwitch)
{
    case 1:
        Console.WriteLine("Case 1...");
        break;
    case 2:
    case 3:
        Console.WriteLine("... and/or Case 2");
        break;
    case 4:
        while (true)
            Console.WriteLine("Endless looping. . .");
    default:
        Console.WriteLine("Default value...");
        break;
}

```

在 `case` 标签与匹配表达式匹配的开关部分中执行语句列表时，先执行第一个语句，再执行整个语句列表，通常执行到跳转语句(如 `break`、`goto case`、`goto label`、`return` 或 `throw`)为止。此时，控件在 `switch` 语句之外进行传输或传输到另一个 `case` 标签。如果使用的是 `goto` 语句，必须将控制权移交给常量标签。此限制是必要的，因为尝试将控制权移交给非常数标签可能会产生不良的副作用，如将控制权移交给代码中的意外位置，或创建无限循环。

Case 标签

每个 `case` 标签指定一个模式与匹配表达式(前面示例中的 `caseSwitch` 变量)进行比较。如果它们匹配，则将控件传输到包含首次匹配 `case` 标签的开关部分。如果 `case` 标签模式与匹配表达式不匹配，控制权会转让给带 `default` `case` 标签的部分(若有)。如果没有 `default` `case`，将不会执行任何 `switch` 部分中的语句，并且会将控制权转让到 `switch` 语句之外。

有关 `switch` 语句和模式匹配的信息，请参阅使用 `switch` 语句的 [模式匹配](#) 部分。

因为 C# 6 仅支持常量模式且禁止重复常量值，所以 `case` 标签定义了互斥值，而且只能有一个模式与匹配表达式匹配。因此，`case` 语句显示的顺序并不重要。

然而，在 C# 7.0 中，因为支持其他模式，所以 `case` 标签不需要定义互斥值，并且多个模式可以与匹配表达式相匹配。因为仅执行包含匹配模式的首次开关部分中的语句，所以 `case` 语句显示的顺序很重要。如果 C# 检测到开关部分的 `case` 语句或语句等效于或是先前语句的子集，它将生成编译错误 CS8120：“开关 `case` 已由先前 `case` 处理。”

以下示例说明了使用各种非互斥模式的 `switch` 语句。如果你移动 `case 0:` `switch` 部分，使之不再是 `switch` 语句中的第一部分，C# 会生成编译器错误，因为值为零的整数是所有整数的子集(由 `case int val` 语句定义的模式)。

```

using System;
using System.Collections.Generic;
using System.Linq;

public class Example
{
    public static void Main()
    {
        var values = new List<object>();
        for (int ctr = 0; ctr <= 7; ctr++) {
            if (ctr == 2)
                values.Add(DiceLibrary.Roll2());
            else if (ctr == 4)
                values.Add(DiceLibrary.Pass());
            else
                values.Add(DiceLibrary.Roll());
        }
    }
}

```

```

    }

    Console.WriteLine($"The sum of { values.Count } die is { DiceLibrary.DiceSum(values) }");
}
}

public static class DiceLibrary
{
    // Random number generator to simulate dice rolls.
    static Random rnd = new Random();

    // Roll a single die.
    public static int Roll()
    {
        return rnd.Next(1, 7);
    }

    // Roll two dice.
    public static List<object> Roll2()
    {
        var rolls = new List<object>();
        rolls.Add(Roll());
        rolls.Add(Roll());
        return rolls;
    }

    // Calculate the sum of n dice rolls.
    public static int DiceSum(IEnumerable<object> values)
    {
        var sum = 0;
        foreach (var item in values)
        {
            switch (item)
            {
                // A single zero value.
                case 0:
                    break;
                // A single value.
                case int val:
                    sum += val;
                    break;
                // A non-empty collection.
                case IEnumerable<object> subList when subList.Any():
                    sum += DiceSum(subList);
                    break;
                // An empty collection.
                case IEnumerable<object> subList:
                    break;
                // A null reference.
                case null:
                    break;
                // A value that is neither an integer nor a collection.
                default:
                    throw new InvalidOperationException("unknown item type");
            }
        }
        return sum;
    }

    public static object Pass()
    {
        if (rnd.Next(0, 2) == 0)
            return null;
        else
            return new List<object>();
    }
}

```

你可以通过以下两种方法之一更正此问题并消除编译器警告：

- 更改开关部分的顺序。
- 在 `case` 标签中使用 `when clause` 子句。

default case

如果匹配表达式与其他任何 `case` 标签都不匹配，`default` case 指定要执行的 switch 部分。如果没有 `default` case，且匹配表达式与其他任何 `case` 标签都不匹配，程序流就会贯穿 `switch` 语句。

`default` case 可以在 `switch` 语句中以任何顺序显示。无论它在源代码中的顺序如何，始终都将在计算所有 `case` 标签后，最后计算它。

使用 `switch` 语句的模式匹配

每个 `case` 语句定义一个模式，如果它与匹配表达式相匹配，则会导致执行其包含的开关部分。所有版本的 C# 都支持常量模式。其余模式从 C# 7.0 开始支持。

常量模式

常量模式测试匹配表达式是否等于指定常量。语法为：

```
case constant:
```

其中 `constant` 是要测试的值。`constant` 可以是以下任何常数表达式：

- `bool` 文本：`true` 或 `false`。
- 任何 `整型` 常数，例如 `int`、`long` 或 `byte`。
- 已声明 `const` 变量的名称。
- 一个枚举常量。
- `字符型` 文本。
- `字符串` 文本。

常数表达式的计算方式如下：

- 如果 `expr` 和 `constant` 均为整型类型，则 C# 相等运算符确定表示式是否返回 `true`（即，是否为 `expr == constant`）。
- 否则，由对静态 `Object.Equals(expr, constant)` 方法的调用确定表达式的值。

以下示例使用常量模式来确定特定日期是否为周末、工作周的第一天、工作周的最后一天或工作周的中间日期。它根据 `DayOfWeek` 枚举的成员计算当前日期的 `DateTime.DayOfWeek` 属性。

```
using System;

class Program
{
    static void Main()
    {
        switch (DateTime.Now.DayOfWeek)
        {
            case DayOfWeek.Sunday:
            case DayOfWeek.Saturday:
                Console.WriteLine("The weekend");
                break;
            case DayOfWeek.Monday:
                Console.WriteLine("The first day of the work week.");
                break;
            case DayOfWeek.Friday:
                Console.WriteLine("The last day of the work week.");
                break;
            default:
                Console.WriteLine("The middle of the work week.");
                break;
        }
    }
}

// The example displays output like the following:
//      The middle of the work week.
```

以下示例使用常量模式在模拟自动咖啡机的控制台应用程序中处理用户输入。

```

using System;

class Example
{
    static void Main()
    {
        Console.WriteLine("Coffee sizes: 1=small 2=medium 3=large");
        Console.Write("Please enter your selection: ");
        string str = Console.ReadLine();
        int cost = 0;

        // Because of the goto statements in cases 2 and 3, the base cost of 25
        // cents is added to the additional cost for the medium and large sizes.
        switch (str)
        {
            case "1":
            case "small":
                cost += 25;
                break;
            case "2":
            case "medium":
                cost += 25;
                goto case "1";
            case "3":
            case "large":
                cost += 50;
                goto case "1";
            default:
                Console.WriteLine("Invalid selection. Please select 1, 2, or 3.");
                break;
        }
        if (cost != 0)
        {
            Console.WriteLine("Please insert {0} cents.", cost);
        }
        Console.WriteLine("Thank you for your business.");
    }
}

// The example displays output like the following:
//      Coffee sizes: 1=small 2=medium 3=large
//      Please enter your selection: 2
//      Please insert 50 cents.
//      Thank you for your business.

```

类型模式

类型模式可启用简洁类型计算和转换。使用 `switch` 语句执行模式匹配时，会测试表达式是否可转换为指定类型，如果可以，则将其转换为该类型的一个变量。语法为：

```
case type varname
```

其中 `type` 是 `expr` 结果要转换到的类型的名称，`varname` 是 `expr` 结果要转换到的对象（如果匹配成功）。自 C# 7.1 起，`expr` 的编译时类型可能为泛型类型参数。

如果以下任一条件成立，则 `case` 表达式为 `true`：

- `expr` 是与 `type` 具有相同类型的一个实例。
- `expr` 是派生自 `type` 的类型的一个实例。换言之，`expr` 结果可以向上转换为 `type` 的一个实例。
- `expr` 具有属于 `type` 的一个基类的编译时类型，`expr` 还具有属于 `type` 或派生自 `type` 的运行时类型。变量的编译时类型是其类型声明中定义的变量类型。变量的运行时类型是分配给该变量的实例类型。
- `expr` 是实现 `type` 接口的类型的一个实例。

如果 case 表达式为 true, 将会明确分配 varname, 并且仅在开关部分中具有本地作用域。

请注意, `null` 与任何类型都不匹配。若要匹配 `null`, 请使用以下 `case` 标签:

```
case null:
```

以下示例使用类型模式来提供有关各种集合类型的信息。

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

class Example
{
    static void Main(string[] args)
    {
        int[] values = { 2, 4, 6, 8, 10 };
        ShowCollectionInformation(values);

        var names = new List<string>();
        names.AddRange(new string[] { "Adam", "Abigail", "Bertrand", "Bridgette" });
        ShowCollectionInformation(names);

        List<int> numbers = null;
        ShowCollectionInformation(numbers);
    }

    private static void ShowCollectionInformation(object coll)
    {
        switch (coll)
        {
            case Array arr:
                Console.WriteLine($"An array with {arr.Length} elements.");
                break;
            case IEnumerable<int> ieInt:
                Console.WriteLine($"Average: {ieInt.Average(s => s)}");
                break;
            case IList list:
                Console.WriteLine($"{list.Count} items");
                break;
            case IEnumerable ie:
                string result = "";
                foreach (var e in ie)
                    result += $"{e} ";
                Console.WriteLine(result);
                break;
            case null:
                // Do nothing for a null.
                break;
            default:
                Console.WriteLine($"A instance of type {coll.GetType().Name}");
                break;
        }
    }
}

// The example displays the following output:
//      An array with 5 elements.
//      4 items
```

可以创建泛型方法, 使用集合的类型作为类型参数(而不是使用 `object`), 如下面的代码所示:

```

using System;
using System.Collections;
using System.Collections.Generic;
using System.Linq;

class Example
{
    static void Main(string[] args)
    {
        int[] values = { 2, 4, 6, 8, 10 };
        ShowCollectionInformation(values);

        var names = new List<string>();
        names.AddRange(new string[] { "Adam", "Abigail", "Bertrand", "Bridgette" });
        ShowCollectionInformation(names);

        List<int> numbers = null;
        ShowCollectionInformation(numbers);
    }

    private static void ShowCollectionInformation<T>(T coll)
    {
        switch (coll)
        {
            case Array arr:
                Console.WriteLine($"An array with {arr.Length} elements.");
                break;
            case IEnumerable<int> ieInt:
                Console.WriteLine($"Average: {ieInt.Average(s => s)}");
                break;
            case IList list:
                Console.WriteLine($"{list.Count} items");
                break;
            case IEnumerable ie:
                string result = "";
                foreach (var e in ie)
                    result += $"{e} ";
                Console.WriteLine(result);
                break;
            case object o:
                Console.WriteLine($"A instance of type {o.GetType().Name}");
                break;
            default:
                Console.WriteLine("Null passed to this method.");
                break;
        }
    }
}

// The example displays the following output:
//     An array with 5 elements.
//     4 items
//     Null passed to this method.

```

泛型版本与第一个示例有两点不同。首先，无法使用 `null` case。无法使用任何常量 case 是因为，编译器无法将任意类型 `T` 转换为除 `object` 之外的任何类型。曾经的 `default` case 现在测试是否有非 `null` `object`。也就是说，`default` case 测试只针对 `null`。

如果没有模式匹配，则可能按以下方式编写此代码。使用类型模式匹配可消除测试转换结果是否为 `null` 或执行重复转换的必要，从而生成更紧凑易读的代码。

```

using System;
using System.Collections;
using System.Collections.Generic;
using System.Linq;

class Example
{
    static void Main(string[] args)
    {
        int[] values = { 2, 4, 6, 8, 10 };
        ShowCollectionInformation(values);

        var names = new List<string>();
        names.AddRange(new string[] { "Adam", "Abigail", "Bertrand", "Bridgette" });
        ShowCollectionInformation(names);

        List<int> numbers = null;
        ShowCollectionInformation(numbers);
    }

    private static void ShowCollectionInformation(object coll)
    {
        if (coll is Array)
        {
            Array arr = (Array) coll;
            Console.WriteLine($"An array with {arr.Length} elements.");
        }
        else if (coll is IEnumerable<int>)
        {
            IEnumerable<int> ieInt = (IEnumerable<int>) coll;
            Console.WriteLine($"Average: {ieInt.Average(s => s)}");
        }
        else if (coll is IList)
        {
            IList list = (IList) coll;
            Console.WriteLine($"{list.Count} items");
        }
        else if (coll is IEnumerable)
        {
            IEnumerable ie = (IEnumerable) coll;
            string result = "";
            foreach (var e in ie)
                result += $"{e} ";
            Console.WriteLine(result);
        }
        else if (coll == null)
        {
            // Do nothing.
        }
        else
        {
            Console.WriteLine($"An instance of type {coll.GetType().Name}");
        }
    }
}

// The example displays the following output:
//     An array with 5 elements.
//     4 items

```

case 语句和 when 子句

从 C# 7.0 开始，因为 case 语句不需要互相排斥，因此可以添加 when 子句来指定必须满足的附加条件使 case 语句计算为 true。when 子句可以是返回布尔值的任何表达式。

下面的示例定义了 `Shape` 基类、从 `Shape` 派生的 `Rectangle` 类以及从 `Rectangle` 派生的 `Square` 类。它使用 `when` 子句，以确保 `ShowShapeInfo` 将已分配相等长度和宽度的 `Rectangle` 对象视为 `Square`（即使它尚未实例化为 `Square` 对象）。此方法不会尝试显示值为 `null` 的对象或面积为零的形状的相关信息。

```
using System;

public abstract class Shape
{
    public abstract double Area { get; }
    public abstract double Circumference { get; }
}

public class Rectangle : Shape
{
    public Rectangle(double length, double width)
    {
        Length = length;
        Width = width;
    }

    public double Length { get; set; }
    public double Width { get; set; }

    public override double Area
    {
        get { return Math.Round(Length * Width,2); }
    }

    public override double Circumference
    {
        get { return (Length + Width) * 2; }
    }
}

public class Square : Rectangle
{
    public Square(double side) : base(side, side)
    {
        Side = side;
    }

    public double Side { get; set; }
}

public class Circle : Shape
{
    public Circle(double radius)
    {
        Radius = radius;
    }

    public double Radius { get; set; }

    public override double Circumference
    {
        get { return 2 * Math.PI * Radius; }
    }

    public override double Area
    {
        get { return Math.PI * Math.Pow(Radius, 2); }
    }
}

public class Example
{
    public static void Main()
```

```

{
    Shape sh = null;
    Shape[] shapes = { new Square(10), new Rectangle(5, 7),
                      sh, new Square(0), new Rectangle(8, 8),
                      new Circle(3) };
    foreach (var shape in shapes)
        ShowShapeInfo(shape);
}

private static void ShowShapeInfo(Shape sh)
{
    switch (sh)
    {
        // Note that this code never evaluates to true.
        case Shape shape when shape == null:
            Console.WriteLine($"An uninitialized shape (shape == null)");
            break;
        case null:
            Console.WriteLine($"An uninitialized shape");
            break;
        case Shape shape when sh.Area == 0:
            Console.WriteLine($"The shape: {sh.GetType().Name} with no dimensions");
            break;
        case Square sq when sh.Area > 0:
            Console.WriteLine("Information about square:");
            Console.WriteLine($"    Length of a side: {sq.Side}");
            Console.WriteLine($"    Area: {sq.Area}");
            break;
        case Rectangle r when r.Length == r.Width && r.Area > 0:
            Console.WriteLine("Information about square rectangle:");
            Console.WriteLine($"    Length of a side: {r.Length}");
            Console.WriteLine($"    Area: {r.Area}");
            break;
        case Rectangle r when sh.Area > 0:
            Console.WriteLine("Information about rectangle:");
            Console.WriteLine($"    Dimensions: {r.Length} x {r.Width}");
            Console.WriteLine($"    Area: {r.Area}");
            break;
        case Shape shape when sh != null:
            Console.WriteLine($"A {sh.GetType().Name} shape");
            break;
        default:
            Console.WriteLine($"The {nameof(sh)} variable does not represent a Shape.");
            break;
    }
}
}

// The example displays the following output:
//     Information about square:
//         Length of a side: 10
//         Area: 100
//     Information about rectangle:
//         Dimensions: 5 x 7
//         Area: 35
//     An uninitialized shape
//     The shape: Square with no dimensions
//     Information about square rectangle:
//         Length of a side: 8
//         Area: 64
//     A Circle shape

```

请注意，不会执行尝试测试 `Shape` 对象是否为 `null` 的示例中的 `when` 子句。测试是否为 `null` 的正确类型模式是 `case null:`。

C# 语言规范

有关详细信息，请参阅 [C# 语言规范](#)中的 `switch` 语句。该语言规范是 C# 语法和用法的权威资料。

请参阅

- [C# 参考](#)
- [C# 编程指南](#)
- [C# 关键字](#)
- [if-else](#)
- [模式匹配](#)

do (C# 参考)

2020/11/2 • [Edit Online](#)

在指定的布尔表达式的计算结果为 `true` 时，`do` 语句会执行一条语句或一个语句块。由于在每次执行循环之后都会计算此表达式，所以 `do-while` 循环会执行一次或多次。这不同于 `while` 循环（该循环执行零次或多次）。

在 `do` 语句块中的任何点，都可使用 `break` 语句中断循环。

可通过使用 `continue` 语句直接步入 `while` 表达式的计算部分。如果表达式计算结果为 `true`，则继续执行循环中的第一个语句。否则，将在循环后的第一个语句处继续执行。

还可以使用 `goto`、`return` 或 `throw` 语句退出 `do-while` 循环。

示例

下面的示例演示 `do` 语句的用法。选择“运行”以运行示例代码。然后可以修改代码并再次运行它。

```
int n = 0;
do
{
    Console.WriteLine(n);
    n++;
} while (n < 5);
```

C# 语言规范

有关详细信息，请参阅 [C# 语言规范](#) 中的 `do` 语句部分。

请参阅

- [C# 参考](#)
- [C# 编程指南](#)
- [C# 关键字](#)
- [while 语句](#)

for (C# 参考)

2020/11/2 • [Edit Online](#)

在指定的布尔表达式的计算结果为 `true` 时, `for` 语句会执行一条语句或一个语句块。

在 `for` 语句块中的任何点上, 可以使用 `break` 语句中断循环, 或者可以使用 `continue` 语句继续执行到循环中的下一次迭代。还可以使用 `goto`、`return` 或 `throw` 语句退出 `for` 循环。

for 语句的结构

`for` 语句定义初始化表达式、条件和迭代器部分 :

```
for (initializer; condition; iterator)
    body
```

三个部分都是可选的。循环体是一个语句或一个语句块。

以下示例显示定义了所有部分的 `for` :

```
for (int i = 0; i < 5; i++)
{
    Console.WriteLine(i);
}
```

“初始化表达式”部分

“初始化表达式”部分的语句仅在进入循环前执行一次。“初始化表达式”部分是下列内容之一 :

- 本地循环变量的声明和初始化, 不能从循环外访问。
- 以下列表中显示用逗号分隔的零个或多个语句表达式:
 - 赋值语句
 - 方法的调用
 - 为 `increment` 表达式添加前缀或后缀, 如 `++i` 或 `i++`
 - 为 `decrement` 表达式添加前缀或后缀, 如 `--i` 或 `i--`
 - 通过使用 `new` 运算符来创建对象
 - `await` 表达式

上例中的“初始化表达式”部分声明和初始化本地循环变量 `i` :

```
int i = 0
```

“条件”部分

“条件”部分(如果存在)必须为布尔表达式。在每次循环迭代前计算该表达式。如果“条件”部分不存在或者布尔表达式的计算结果为 `true`, 则执行下一个循环迭代;否则退出循环。

上例中的“条件”部分确定循环是否根据本地循环变量的值终止 :

```
i < 5
```

“迭代器”部分

“迭代器”部分定义循环主体的每次迭代后将执行的操作。“迭代器”部分包含用逗号分隔的零个或多个以下语句表达式：

- 赋值语句
- 方法的调用
- 为 `increment` 表达式添加前缀或后缀，如 `++i` 或 `i++`
- 为 `decrement` 表达式添加前缀或后缀，如 `--i` 或 `i--`
- 通过使用 `new` 运算符来创建对象
- `await` 表达式

上例中的“迭代器”部分增加本地循环变量：

```
i++
```

示例

下面的示例阐释了几种不太常见的 `for` 语句部分的使用情况：为“初始化表达式”部分中的外部循环变量赋值、同时在“初始化表达式”部分和“迭代器”部分中调用一种方法，以及更改迭代器部分中的两个变量的值。选择“运行”以运行示例代码。然后可以修改代码并再次运行它。

```
int i;
int j = 10;
for (i = 0, Console.WriteLine($"Start: i={i}, j={j}"); i < j; i++, j--, Console.WriteLine($"Step: i={i}, j={j}"))
{
    // Body of the loop.
}
```

以下示例定义无限 `for` 循环：

```
for ( ; ; )
{
    // Body of the loop.
}
```

C# 语言规范

有关详细信息，请参阅 [C# 语言规范](#) 中的 `for` 语句部分。

请参阅

- [C# 参考](#)
- [C# 编程指南](#)
- [C# 关键字](#)
- [foreach, in](#)

foreach · in (C# 参考)

2020/11/2 • [Edit Online](#)

`foreach` 语句为类型实例中实现 `System.Collections.IEnumerable` 或

`System.Collections.Generic.IEnumerable<T>` 接口的每个元素执行语句或语句块，如以下示例所示：

```
var fibNumbers = new List<int> { 0, 1, 1, 2, 3, 5, 8, 13 };
int count = 0;
foreach (int element in fibNumbers)
{
    Console.WriteLine($"Element #{count}: {element}");
    count++;
}
Console.WriteLine($"Number of elements: {count}");
```

`foreach` 语句并不限于这些类型。可以将其与满足以下条件的任何类型的实例一起使用：

- 类型具有公共无参数 `GetEnumerator` 方法，其返回类型为类、结构或接口类型。从 C# 9.0 开始，`GetEnumerator` 方法可以是类型的扩展方法。
- `GetEnumerator` 方法的返回类型具有公共 `current` 属性和公共无参数 `MoveNext` 方法(其返回类型为 `Boolean`)。

下面的示例使用 `foreach` 语句，其中包含 `System.Span<T>` 类型的实例，该实例不实现任何接口：

```
public class IterateSpanExample
{
    public static void Main()
    {
        Span<int> numbers = new int[] { 3, 14, 15, 92, 6 };
        foreach (int number in numbers)
        {
            Console.Write($"{number} ");
        }
        Console.WriteLine();
    }
}
```

从 C# 7.3 开始，如果枚举器的 `Current` 属性返回引用返回值 (`ref T`，其中 `T` 为集合元素类型)，就可以使用 `ref` 或 `ref readonly` 修饰符来声明迭代变量，如下面的示例所示：

```

public class ForeachRefExample
{
    public static void Main()
    {
        Span<int> storage = stackalloc int[10];
        int num = 0;
        foreach (ref int item in storage)
        {
            item = num++;
        }

        foreach (ref readonly var item in storage)
        {
            Console.WriteLine($"{item} ");
        }
        // Output:
        // 0 1 2 3 4 5 6 7 8 9
    }
}

```

从 C# 8.0 开始，可以使用 `await foreach` 语句来使用异步数据流，即实现 `IAsyncEnumerable<T>` 接口的集合类型。异步检索下一个元素时，可能会挂起循环的每次迭代。下面的示例演示如何使用 `await foreach` 语句：

```

await foreach (var item in GenerateSequenceAsync())
{
    Console.WriteLine(item);
}

```

默认情况下，在捕获的上下文中处理流元素。如果要禁用上下文捕获，请使用 `TaskAsyncEnumerableExtensions.ConfigureAwait` 扩展方法。有关同步上下文并捕获当前上下文的详细信息，请参阅[使用基于任务的异步模式](#)。有关异步流的详细信息，请参阅[C# 8.0 新增功能](#)一文中的[异步流部分](#)。

在 `foreach` 语句块中的任何点上，可以使用 `break` 语句中断循环，或者可以使用 `continue` 语句继续执行到循环中的下一次迭代。还可以使用 `goto`、`return` 或 `throw` 语句退出 `foreach` 循环。

如果 `foreach` 语句应用为 `null`，则会引发 `NullReferenceException`。如果 `foreach` 语句的源集合为空，则 `foreach` 循环的正文不会被执行，而是被跳过。

迭代变量的类型

可以使用 `var` 关键字让编译器推断 `foreach` 语句中迭代变量的类型，如以下代码所示：

```

foreach (var item in collection) { }

```

还可以显式指定迭代变量的类型，如以下代码所示：

```

IEnumerable<T> collection = new T[5];
foreach (V item in collection) { }

```

在上述窗体中，集合元素的类型 `T` 必须可隐式或显式地转换为迭代变量的类型 `V`。如果从 `T` 到 `V` 的显式转换在运行时失败，`foreach` 语句将引发 `InvalidCastException`。例如，如果 `T` 是非密封类类型，则 `V` 可以是任何接口类型，甚至可以是 `T` 未实现的接口类型。在运行时，集合元素的类型可以是从 `T` 派生并且实际实现 `V` 的类型。如果不是这样，则会引发 `InvalidCastException`。

C# 语言规范

有关详细信息，请参阅 [C# 语言规范](#)中的 `foreach` 语句部分。

有关 C# 8.0 及更高版本中添加的功能的详细信息，请参阅以下功能建议说明：

- [异步流 \(C# 8.0\)](#)
- 扩展 `GetEnumerator` 支持 `foreach` 循环 (C# 9.0)

请参阅

- [C# 参考](#)
- [C# 关键字](#)
- [对数组使用 foreach](#)
- [for 语句](#)

while (C# 参考)

2020/11/2 • [Edit Online](#)

在指定的布尔表达式的计算结果为 `true` 时，`while` 语句会执行一条语句或一个语句块。由于在每次执行循环之前都会计算此表达式，所以 `while` 循环会执行零次或多次。这不同于 `do` 循环，该循环执行一次或多次。

在 `while` 语句块中的任何点，都可使用 `break` 语句中断循环。

可通过使用 `continue` 语句直接步入 `while` 表达式的计算部分。如果表达式计算结果为 `true`，则继续执行循环中的第一个语句。否则，将在循环后的第一个语句处继续执行。

还可以使用 `goto`、`return` 或 `throw` 语句退出 `while` 循环。

示例

下面的示例演示 `while` 语句的用法。选择“运行”以运行示例代码。然后可以修改代码并再次运行它。

```
int n = 0;
while (n < 5)
{
    Console.WriteLine(n);
    n++;
}
```

C# 语言规范

有关详细信息，请参阅 [C# 语言规范](#) 中的 `while` 语句部分。

请参阅

- [C# 参考](#)
- [C# 编程指南](#)
- [C# 关键字](#)
- [do 语句](#)

break (C# 参考)

2020/11/2 • [Edit Online](#)

`break` 语句将终止其所在位置的最接近封闭循环或 `switch` 语句。控制权将传递给已终止语句后面的语句(若有)。

示例

在此示例中，条件语句包含一个应从 1 计数到 100 的计数器;但 `break` 语句在计数器计数到 4 后终止了循环。

```
class BreakTest
{
    static void Main()
    {
        for (int i = 1; i <= 100; i++)
        {
            if (i == 5)
            {
                break;
            }
            Console.WriteLine(i);
        }

        // Keep the console open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
/*
Output:
1
2
3
4
*/
```

示例

本示例演示 `break` 在 `switch` 语句中的用法。

```
class Switch
{
    static void Main()
    {
        Console.Write("Enter your selection (1, 2, or 3): ");
        string s = Console.ReadLine();
        int n = Int32.Parse(s);

        switch (n)
        {
            case 1:
                Console.WriteLine("Current value is 1");
                break;
            case 2:
                Console.WriteLine("Current value is 2");
                break;
            case 3:
                Console.WriteLine("Current value is 3");
                break;
            default:
                Console.WriteLine("Sorry, invalid selection.");
                break;
        }

        // Keep the console open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}

/*
Sample Input: 1

Sample Output:
Enter your selection (1, 2, or 3): 1
Current value is 1
*/

```

如果输入 4，则输出为：

```
Enter your selection (1, 2, or 3): 4
Sorry, invalid selection.
```

示例

在此示例中，`break` 语句用于中断内层嵌套循环，并将控制权返回给外层循环。控件在嵌套循环中仅向上返回一级。

```

class BreakInNestedLoops
{
    static void Main(string[] args)
    {

        int[] numbers = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
        char[] letters = { 'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j' };

        // Outer loop.
        for (int i = 0; i < numbers.Length; i++)
        {
            Console.WriteLine($"num = {numbers[i]}");

            // Inner loop.
            for (int j = 0; j < letters.Length; j++)
            {
                if (j == i)
                {
                    // Return control to outer loop.
                    break;
                }
                Console.Write($"{letters[j]} ");
            }
            Console.WriteLine();
        }

        // Keep the console open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}

/*
 * Output:
 num = 0

 num = 1
 a
 num = 2
 a b
 num = 3
 a b c
 num = 4
 a b c d
 num = 5
 a b c d e
 num = 6
 a b c d e f
 num = 7
 a b c d e f g
 num = 8
 a b c d e f g h
 num = 9
 a b c d e f g h i
*/

```

示例

在本例中，`break` 语句仅用于在循环的每次迭代中脱离当前分支。循环本身不受属于嵌套 `switch` 语句的 `break` 实例的影响。

```
class BreakFromSwitchInsideLoop
{
    static void Main(string[] args)
    {
        // loop 1 to 3
        for (int i = 1; i <= 3; i++)
        {
            switch(i)
            {
                case 1:
                    Console.WriteLine("Current value is 1");
                    break;
                case 2:
                    Console.WriteLine("Current value is 2");
                    break;
                case 3:
                    Console.WriteLine("Current value is 3");
                    break;
                default:
                    Console.WriteLine("This shouldn't happen.");
                    break;
            }
        }

        // Keep the console open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}

/*
 * Output:
 * Current value is 1
 * Current value is 2
 * Current value is 3
*/
```

C# 语言规范

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

请参阅

- [C# 参考](#)
- [C# 编程指南](#)
- [C# 关键字](#)
- [switch](#)

continue (C# 参考)

2020/11/2 • [Edit Online](#)

`continue` 语句将控制传递到其中出现的封闭 `while`、`do`、`for` 或 `foreach` 语句的下一次迭代。

示例

在本示例中，计数器最初是从 1 到 10 进行计数。通过结合使用 `continue` 语句和表达式 `(i < 9)`，在 `i` 小于 9 的迭代中跳过 `continue` 和 `for` 主体末尾之间的语句。在 `for` 循环的最后两次迭代（其中 `i == 9`, `i == 10`）中，不执行 `continue` 语句，且 `i` 的值将输出到控制台。

```
class ContinueTest
{
    static void Main()
    {
        for (int i = 1; i <= 10; i++)
        {
            if (i < 9)
            {
                continue;
            }
            Console.WriteLine(i);
        }

        // Keep the console open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}

/*
Output:
9
10
*/
```

C# 语言规范

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

请参阅

- [C# 参考](#)
- [C# 编程指南](#)
- [C# 关键字](#)
- [break 语句](#)

goto (C# 参考)

2020/11/2 • [Edit Online](#)

`goto` 语句将程序控制直接传递给标记语句。

`goto` 的一个通常用法是将控制传递给特定的 `switch-case` 标签或 `switch` 语句中的默认标签。

`goto` 语句还用于跳出深嵌套循环。

示例

下面的示例演示了 `goto` 在 `switch` 语句中的使用。

```
class SwitchTest
{
    static void Main()
    {
        Console.WriteLine("Coffee sizes: 1=Small 2=Medium 3=Large");
        Console.Write("Please enter your selection: ");
        string s = Console.ReadLine();
        int n = int.Parse(s);
        int cost = 0;
        switch (n)
        {
            case 1:
                cost += 25;
                break;
            case 2:
                cost += 25;
                goto case 1;
            case 3:
                cost += 50;
                goto case 1;
            default:
                Console.WriteLine("Invalid selection.");
                break;
        }
        if (cost != 0)
        {
            Console.WriteLine($"Please insert {cost} cents.");
        }
        Console.WriteLine("Thank you for your business.");

        // Keep the console open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
/*
Sample Input: 2

Sample Output:
Coffee sizes: 1=Small 2=Medium 3=Large
Please enter your selection: 2
Please insert 50 cents.
Thank you for your business.
*/
```

示例

下面的示例演示了使用 `goto` 跳出嵌套循环。

```
public class GotoTest1
{
    static void Main()
    {
        int x = 200, y = 4;
        int count = 0;
        string[,] array = new string[x, y];

        // Initialize the array.
        for (int i = 0; i < x; i++)
            for (int j = 0; j < y; j++)
                array[i, j] = (++count).ToString();

        // Read input.
        Console.Write("Enter the number to search for: ");

        // Input a string.
        string myNumber = Console.ReadLine();

        // Search.
        for (int i = 0; i < x; i++)
        {
            for (int j = 0; j < y; j++)
            {
                if (array[i, j].Equals(myNumber))
                {
                    goto Found;
                }
            }
        }

        Console.WriteLine($"The number {myNumber} was not found.");
        goto Finish;

        Found:
        Console.WriteLine($"The number {myNumber} is found.");

        Finish:
        Console.WriteLine("End of search.");

        // Keep the console open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}

/*
Sample Input: 44

Sample Output
Enter the number to search for: 44
The number 44 is found.
End of search.
*/
```

C# 语言规范

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

请参阅

- [C# 参考](#)
- [C# 编程指南](#)
- [C# 关键字](#)
- [goto 语句 \(C++\)](#)

return (C# 参考)

2020/11/2 • [Edit Online](#)

`return` 语句可终止它所在的方法的执行，并将控制权返回给调用方法。它还可以返回可选值。如果方法是 `void` 类型，则 `return` 语句可以省略。

如果 `return` 语句位于 `try` 块中，则 `finally` 块（如果存在）会在控制权返回给调用方法之前进行执行。

示例

在下面的示例中，方法 `CalculateArea()` 将局部变量 `area` 作为 `double` 值返回。

```
class ReturnTest
{
    static double CalculateArea(int r)
    {
        double area = r * r * Math.PI;
        return area;
    }

    static void Main()
    {
        int radius = 5;
        double result = CalculateArea(radius);
        Console.WriteLine("The area is {0:0.00}", result);

        // Keep the console open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
// Output: The area is 78.54
```

C# 语言规范

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

请参阅

- [C# 参考](#)
- [C# 编程指南](#)
- [C# 关键字](#)
- [return 语句](#)

throw (C# 参考)

2021/5/7 • • [Edit Online](#)

发出程序执行期间出现异常的信号。

备注

`throw` 的语法为：

```
throw [e];
```

`e` 是一个派生自 `System.Exception` 的类的实例。下例使用 `throw` 语句在传递给名为 `GetNumber` 的方法的参数与内部数组的有效索引不对应时引发 `IndexOutOfRangeException`。

```
using System;

namespace Throw2
{
    public class NumberGenerator
    {
        int[] numbers = { 2, 4, 6, 8, 10, 12, 14, 16, 18, 20 };

        public int GetNumber(int index)
        {
            if (index < 0 || index >= numbers.Length)
            {
                throw new IndexOutOfRangeException();
            }
            return numbers[index];
        }
    }
}
```

然后方法调用方使用 `try-catch` 或 `try-catch-finally` 块来处理引发的异常。下例处理由 `GetNumber` 方法引发的异常。

```
using System;

public class Example
{
    public static void Main()
    {
        var gen = new NumberGenerator();
        int index = 10;
        try
        {
            int value = gen.GetNumber(index);
            Console.WriteLine($"Retrieved {value}");
        }
        catch (IndexOutOfRangeException e)
        {
            Console.WriteLine($"{e.GetType().Name}: {index} is outside the bounds of the array");
        }
    }
}

// The example displays the following output:
//      IndexOutOfRangeException: 10 is outside the bounds of the array
```

重新引发异常

`throw` 也可以用于 `catch` 块，以重新引发在 `catch` 块中处理的异常。在这种情况下，`throw` 不采用异常操作数。当方法将参数从调用方传递给其他库方法时，这是最有用的，库方法引发的异常必须传递给调用方。例如，以下示例重新引发在尝试检索未初始化字符串的第一个字符时引发的 [NullReferenceException](#)。

```

using System;

namespace Throw
{
    public class Sentence
    {
        public Sentence(string s)
        {
            Value = s;
        }

        public string Value { get; set; }

        public char GetFirstCharacter()
        {
            try
            {
                return Value[0];
            }
            catch (NullReferenceException e)
            {
                throw;
            }
        }
    }

    public class Example
    {
        public static void Main()
        {
            var s = new Sentence(null);
            Console.WriteLine($"The first character is {s.GetFirstCharacter()}");
        }
    }
}

// The example displays the following output:
//     Unhandled Exception: System.NullReferenceException: Object reference not set to an instance of an
object.
//         at Sentence.GetFirstCharacter()
//         at Example.Main()

```

IMPORTANT

还可以使用 `catch` 块中的 `throw e` 语句来实例化传递给调用方的新异常。在这种情况下，将不会保留可从 `StackTrace` 属性获得的原始异常的堆栈跟踪。

throw 表达式

从 C# 7.0 开始，`throw` 可以用作表达式和语句。这允许在以前不支持的上下文中引发异常。这些方法包括：

- **条件运算符**。下例使用 `throw` 表达式在向方法传递空字符串数组时引发 `ArgumentException`。在 C# 7.0 之前，此逻辑将需要显示在 `if / else` 语句中。

```

private static void DisplayFirstNumber(string[] args)
{
    string arg = args.Length >= 1 ? args[0] :
        throw new ArgumentException("You must supply an argument");
    if (Int64.TryParse(arg, out var number))
        Console.WriteLine($"You entered {number:F0}");
    else
        Console.WriteLine($"{arg} is not a number.");
}

```

- **null 合并运算符**。在以下示例中，如果分配给 `Name` 属性的字符串为 `null`，则将 `throw` 表达式与 `null` 合并运算符结合使用以引发异常。

```
public string Name
{
    get => name;
    set => name = value ??
        throw new ArgumentNullException(paramName: nameof(value), message: "Name cannot be null");
}
```

- expression-bodied `lambda` 或方法。下例说明了 expression-bodied 方法，由于不支持对 `DateTime` 值的转换，该方法引发 `InvalidCastException`。

```
DateTime ToDateTime(IFormatProvider provider) =>
    throw new InvalidCastException("Conversion to a DateTime is not supported.");
```

C# 语言规范

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

请参阅

- [C# 参考](#)
- [C# 编程指南](#)
- [try-catch](#)
- [C# 关键字](#)
- [如何：显式抛出异常](#)

try-catch (C# 参考)

2021/5/10 • [Edit Online](#)

Try-catch 语句包含一个后接一个或多个 `catch` 子句的 `try` 块，这些子句指定不同异常的处理程序。

引发异常时，公共语言运行时 (CLR) 查找处理此异常的 `catch` 语句。如果当前正在执行的方法不包含此类 `catch` 块，则 CLR 查看调用了当前方法的方法，并以此类推遍历调用堆栈。如果未找到任何 `catch` 块，则 CLR 向用户显示一条未处理的异常消息，并停止执行程序。

`try` 块包含可能导致异常的受保护的代码。将执行此块，直至引发异常或其成功完成。例如，强制转换 `null` 对象的以下尝试会引发 `NullReferenceException` 异常：

```
object o2 = null;
try
{
    int i2 = (int)o2;    // Error
}
```

尽管可以不带参数使用 `catch` 子句来捕获任何类型的异常，但不推荐这种用法。一般情况下，只应捕获你知道如何从其恢复的异常。因此，应始终指定派生自 `System.Exception` 的对象参数，例如：

```
catch (InvalidOperationException e)
{
}
```

可以使用同一 try-catch 语句中的多个特定 `catch` 子句。在这种情况下，`catch` 子句的顺序很重要，因为 `catch` 子句是按顺序检查的。在使用更笼统的子句之前获取特定性更强的异常。如果捕获块的排序使得永不会达到之后的块，则编译器将产生错误。

筛选想要处理的异常的一种方式是使用 `catch` 参数。也可以使用异常筛选器进一步检查该异常以决定是否要对其进行处理。如果异常筛选器返回 `false`，则继续搜索处理程序。

```
catch (ArgumentException e) when (e.ParamName == "...")
{}
```

异常筛选器要优于捕获和重新引发(如下所述)，因为筛选器将保留堆栈不受损坏。如果之后的处理程序转储堆栈，可以查看到异常的原始来源，而不只是重新引发它的最后一个位置。异常筛选器表达式的一个常见用途是日志记录。可以创建一个始终返回 `false` 并输出到日志的筛选器，能在异常通过时进行记录，且无需处理并重新引发它们。

可在 `catch` 块中使用 `throw` 语句以重新引发已由 `catch` 语句捕获的异常。下面的示例从 `IOException` 异常提取源信息，然后向父方法引发异常。

```
catch (FileNotFoundException e)
{
    // FileNotFoundExceptions are handled here.
}
catch (IOException e)
{
    // Extract some information from this exception, and then
    // throw it to the parent method.
    if (e.Source != null)
        Console.WriteLine("IOException source: {0}", e.Source);
    throw;
}
```

你可以捕获一个异常而引发一个不同的异常。执行此操作时，请指定作为内部异常捕获的异常，如以下示例所示。

```
catch (InvalidOperationException e)
{
    // Perform some action here, and then throw a new exception.
    throw new YourCustomException("Put your error message here.", e);
}
```

当指定的条件为 true 时，你还可以重新引发异常，如以下示例所示。

```
catch (InvalidOperationException e)
{
    if (e.Data == null)
    {
        throw;
    }
    else
    {
        // Take some action.
    }
}
```

NOTE

还可以使用异常筛选器以更简洁的方式获取类似的结果(不修改堆栈，如本文档前面的部分所述)。下面的示例中，调用方的行为类似于前面的示例。当 `e.Data` 为 `null` 时，该函数引发 `InvalidOperationException` 返回至调用方。

```
catch (InvalidOperationException e) when (e.Data != null)
{
    // Take some action.
}
```

从 `try` 块内，仅初始化在其中声明的变量。否则，在完成执行块之前，可能会出现异常。例如，在下面的代码示例中，变量 `n` 在 `try` 块内部初始化。尝试在 `Write(n)` 语句的 `try` 块外部使用此变量将生成编译器错误。

```
static void Main()
{
    int n;
    try
    {
        // Do not initialize this variable here.
        n = 123;
    }
    catch
    {
    }
    // Error: Use of unassigned local variable 'n'.
    Console.WriteLine(n);
}
```

有关 `catch` 的详细信息, 请参阅 [try-catch-finally](#)。

异步方法中的异常

异步方法由 `async` 修饰符标记, 通常包含一个或多个 `await` 表达式或语句。`await` 表达式将 `await` 运算符应用于 `Task` 或 `Task<TResult>`。

当控件到达异步方法中的 `await` 时, 将挂起方法中的进度, 直到所等待的任务完成。任务完成后, 可以在方法中恢复执行。有关详细信息, 请参阅 [使用 Async 和 Await 的异步编程](#)。

应用了 `await` 的完成任务可能由于返回此任务的方法中存在未处理的异常而处于错误状态。等待该任务引发异常。如果取消了返回任务的异步进程, 此任务最后也可能为已取消状态。等待已取消的任务时将引发 `OperationCanceledException`。

若要捕获异常, 请在 `try` 块中等待任务并在关联的 `catch` 块中捕获异常。有关示例, 请参阅 [异步方法示例](#) 部分。

任务可能处于错误状态, 因为等待的异步方法中发生了多个异常。例如, 任务可能是对 `Task.WhenAll` 调用的结果。当等待此类任务时, 仅捕捉到其中一个异常, 而且你无法预测将会捕获到哪个异常。有关示例, 请参阅 [Task.WhenAll 示例](#) 部分。

示例

在下面的示例中, `try` 块包含对可能引发异常的 `ProcessString` 方法的调用。`catch` 子句包含只在屏幕上显示一条消息的异常处理程序。当从 `ProcessString` 内部调用 `throw` 语句时, 系统将查找 `catch` 语句并显示消息 `Exception caught`。

```
class TryFinallyTest
{
    static void ProcessString(string s)
    {
        if (s == null)
        {
            throw new ArgumentNullException();
        }
    }

    public static void Main()
    {
        string s = null; // For demonstration purposes.

        try
        {
            ProcessString(s);
        }
        catch (Exception e)
        {
            Console.WriteLine("{0} Exception caught.", e);
        }
    }
}

/*
Output:
System.ArgumentNullException: Value cannot be null.
at TryFinallyTest.Main() Exception caught.
* */
```

两个 catch 块示例

在下面的示例中，使用了两个 catch 块，并捕获到最先出现的最具体的异常。

若要捕获最不具体的异常，你可以将 `ProcessString` 中的 `throw` 语句替换为以下语句：`throw new Exception()`。

如果将最不具体的 catch 块置于示例中第一个，将显示以下错误消息：

A previous catch clause already catches all exceptions of this or a super type ('`System.Exception`')。

```

class ThrowTest3
{
    static void ProcessString(string s)
    {
        if (s == null)
        {
            throw new ArgumentNullException();
        }
    }

    public static void Main()
    {
        try
        {
            string s = null;
            ProcessString(s);
        }
        // Most specific:
        catch (ArgumentNullException e)
        {
            Console.WriteLine("{0} First exception caught.", e);
        }
        // Least specific:
        catch (Exception e)
        {
            Console.WriteLine("{0} Second exception caught.", e);
        }
    }
}
/*
Output:
System.ArgumentNullException: Value cannot be null.
at Test.ThrowTest3.ProcessString(String s) ... First exception caught.
*/

```

异步方法示例

下面的示例阐释异步方法的异常处理。若要捕获异步任务引发的异常，将 `await` 表达式置于 `try` 块中，并在 `catch` 块中捕获该异常。

在示例中取消注释 `throw new Exception` 行以演示异常处理。任务的 `IsFaulted` 属性设置为 `True`，任务的 `Exception.InnerException` 属性设置为异常，并在 `catch` 块中捕获该异常。

取消注释 `throw new OperationCanceledException` 行以演示在取消异步进程时发生的情况。任务的 `IsCanceled` 属性设置为 `true`，并在 `catch` 块中捕获异常。在某些不适用于此示例的情况下，任务的 `IsFaulted` 属性设置为 `true` 且 `IsCanceled` 设置为 `false`。

```

public async Task DoSomethingAsync()
{
    Task<string> theTask = DelayAsync();

    try
    {
        string result = await theTask;
        Debug.WriteLine("Result: " + result);
    }
    catch (Exception ex)
    {
        Debug.WriteLine("Exception Message: " + ex.Message);
    }
    Debug.WriteLine("Task IsCanceled: " + theTask.IsCanceled);
    Debug.WriteLine("Task IsFaulted: " + theTask.IsFaulted);
    if (theTask.Exception != null)
    {
        Debug.WriteLine("Task Exception Message: "
            + theTask.Exception.Message);
        Debug.WriteLine("Task Inner Exception Message: "
            + theTask.Exception.InnerException.Message);
    }
}

private async Task<string> DelayAsync()
{
    await Task.Delay(100);

    // Uncomment each of the following lines to
    // demonstrate exception handling.

    //throw new OperationCanceledException("canceled");
    //throw new Exception("Something happened.");
    return "Done";
}

// Output when no exception is thrown in the awaited method:
//   Result: Done
//   Task IsCanceled: False
//   Task IsFaulted: False

// Output when an Exception is thrown in the awaited method:
//   Exception Message: Something happened.
//   Task IsCanceled: False
//   Task IsFaulted: True
//   Task Exception Message: One or more errors occurred.
//   Task Inner Exception Message: Something happened.

// Output when a OperationCanceledException or TaskCanceledException
// is thrown in the awaited method:
//   Exception Message: canceled
//   Task IsCanceled: True
//   Task IsFaulted: False

```

Task.WhenAll 示例

下面的示例阐释了在多个任务可能导致多个异常的情况中的异常处理。`try` 块等待由 `Task.WhenAll` 的调用返回的任务。应用了 `WhenAll` 的三个任务完成后，该任务完成。

三个任务中的每一个都会导致异常。`catch` 块循环访问异常，这些异常位于由 `Task.WhenAll` 返回的任务的 `Exception.InnerExceptions` 属性中。

```
public async Task DoMultipleAsync()
{
    Task theTask1 = ExcAsync(info: "First Task");
    Task theTask2 = ExcAsync(info: "Second Task");
    Task theTask3 = ExcAsync(info: "Third Task");

    Task allTasks = Task.WhenAll(theTask1, theTask2, theTask3);

    try
    {
        await allTasks;
    }
    catch (Exception ex)
    {
        Debug.WriteLine("Exception: " + ex.Message);
        Debug.WriteLine("Task IsFaulted: " + allTasks.IsFaulted);
        foreach (var inEx in allTasks.Exception.InnerException)
        {
            Debug.WriteLine("Task Inner Exception: " + inEx.Message);
        }
    }
}

private async Task ExcAsync(string info)
{
    await Task.Delay(100);

    throw new Exception("Error-" + info);
}

// Output:
//   Exception: Error-First Task
//   Task IsFaulted: True
//   Task Inner Exception: Error-First Task
//   Task Inner Exception: Error-Second Task
//   Task Inner Exception: Error-Third Task
```

C# 语言规范

有关详细信息，请参阅 [C# 语言规范](#) 中的 [try 语句](#) 部分。

另请参阅

- [C# 参考](#)
- [C# 编程指南](#)
- [C# 关键字](#)
- [try、throw 和 catch 语句 \(C++\)](#)
- [throw](#)
- [try-finally](#)
- [如何：显式引发异常](#)

try-finally (C# 参考)

2021/5/10 • [Edit Online](#)

通过使用 `finally` 块，可以清除 `try` 块中分配的任何资源，即使在 `try` 块中发生异常，也可以运行代码。通常情况下，`finally` 块的语句会在控件离开 `try` 语句时运行。正常执行中，执行 `break`、`continue`、`goto` 或 `return` 语句，或者从 `try` 语句外传播异常都可能会导致发生控件转换。

已处理的异常中会保证运行相关联的 `finally` 块。但是，如果异常未经处理，则 `finally` 块的执行将取决于异常解除操作的触发方式。反过来，这又取决于你计算机的设置方式。只有在 `finally` 子句不运行的情况下，才会涉及程序被立即停止的情况。例如，由于 IL 语句损坏而引发 `InvalidOperationException`。在大多数操作系统上，会在停止和卸载进程时进行合理的资源清理。

通常情况下，当未经处理的异常终止应用程序时，`finally` 块是否运行已不重要。但是，如果 `finally` 块中的语句必须在这种情况下运行，则可以将 `catch` 块添加到 `try - finally` 语句，这是其中一种解决方法。另一种解决方法是，可以捕获可能在调用堆栈上方的 `try - finally` 语句的 `try` 块中引发的异常。也就是说，可以通过以下几种方法来捕获异常：调用包含 `try - finally` 语句的方法、调用该方法或调用堆栈中的任何方法。如果未捕获异常，则 `finally` 块的执行取决于操作系统是否选择触发异常解除操作。

示例

在以下示例中，无效的转换语句会导致 `System.InvalidCastException` 异常。异常未经处理。

```
public class ThrowTestA
{
    public static void Main()
    {
        int i = 123;
        string s = "Some string";
        object obj = s;

        try
        {
            // Invalid conversion; obj contains a string, not a numeric type.
            i = (int)obj;

            // The following statement is not run.
            Console.WriteLine("WriteLine at the end of the try block.");
        }
        finally
        {
            // To run the program in Visual Studio, type CTRL+F5. Then
            // click Cancel in the error dialog.
            Console.WriteLine("\nExecution of the finally block after an unhandled\n" +
                "error depends on how the exception unwind operation is triggered.");
            Console.WriteLine("i = {0}", i);
        }
    }
    // Output:
    // Unhandled Exception: System.InvalidCastException: Specified cast is not valid.
    //
    // Execution of the finally block after an unhandled
    // error depends on how the exception unwind operation is triggered.
    // i = 123
}
```

在以下示例中，`TryCast` 方法导致的异常会在比调用堆栈更远的方法中被捕获。

```
public class ThrowTestB
{
    public static void Main()
    {
        try
        {
            // TryCast produces an unhandled exception.
            TryCast();
        }
        catch (Exception ex)
        {
            // Catch the exception that is unhandled in TryCast.
            Console.WriteLine(
                "Catching the {0} exception triggers the finally block.",
                ex.GetType());

            // Restore the original unhandled exception. You might not
            // know what exception to expect, or how to handle it, so pass
            // it on.
            throw;
        }
    }

    static void TryCast()
    {
        int i = 123;
        string s = "Some string";
        object obj = s;

        try
        {
            // Invalid conversion; obj contains a string, not a numeric type.
            i = (int)obj;

            // The following statement is not run.
            Console.WriteLine("WriteLine at the end of the try block.");
        }
        finally
        {
            // Report that the finally block is run, and show that the value of
            // i has not been changed.
            Console.WriteLine("\nIn the finally block in TryCast, i = {0}.\n", i);
        }
    }
    // Output:
    // In the finally block in TryCast, i = 123.

    // Catching the System.InvalidCastException exception triggers the finally block.

    // Unhandled Exception: System.InvalidCastException: Specified cast is not valid.
}
```

有关 `finally` 的详细信息, 请参阅 [try-catch-finally](#)。

C# 还包含 [using 语句](#), 它以简便语法为 [IDisposable](#) 对象提供类似的功能。

C# 语言规范

有关详细信息, 请参阅 [C# 语言规范](#) 中的 [try 语句](#) 部分。

另请参阅

- [C# 参考](#)
- [C# 编程指南](#)

- C# 关键字
- try、throw 和 catch 语句 (C++)
- throw
- try-catch
- 如何：显式引发异常

try-catch-finally (C# 参考)

2021/5/10 • [Edit Online](#)

`catch` 和 `finally` 的常见用法是获得并使用 `try` 块中的资源、处理 `catch` 块中的异常情况，以及释放 `finally` 块中的资源。

有关重新引发异常的详细信息和示例，请参阅 [try-catch](#) 和 [引发异常](#)。有关 `finally` 块的详细信息，请参阅 [try-finally](#)。

示例

```
public class EHClass
{
    void ReadFile(int index)
    {
        // To run this code, substitute a valid path from your local machine
        string path = @"c:\users\public\test.txt";
        System.IO.StreamReader file = new System.IO.StreamReader(path);
        char[] buffer = new char[10];
        try
        {
            file.ReadBlock(buffer, index, buffer.Length);
        }
        catch (System.IO.IOException e)
        {
            Console.WriteLine("Error reading from {0}. Message = {1}", path, e.Message);
        }
        finally
        {
            if (file != null)
            {
                file.Close();
            }
        }
        // Do something with buffer...
    }
}
```

C# 语言规范

有关详细信息，请参阅 [C# 语言规范](#) 中的 [try 语句](#) 部分。

另请参阅

- [C# 参考](#)
- [C# 编程指南](#)
- [C# 关键字](#)
- [try、throw 和 catch 语句 \(C++\)](#)
- [throw](#)
- [如何：显式引发异常](#)
- [using 语句](#)

Checked 和 Unchecked (C# 参考)

2021/5/7 • [Edit Online](#)

C# 语句既可以在已检查的上下文中执行，也可以在未检查的上下文中执行。在已检查的上下文中，算法溢出引发异常。在未选中的上下文中忽略算术溢出并将结果截断，方法是：丢弃任何不适应目标类型的高序位。

- `checked` 指定已检查的上下文。
- `unchecked` 指定未检查的上下文。

下列操作受溢出检查的影响：

- 表达式在整型上使用下列预定义运算符：

`++`, `--`, 一元 `-`, `+`, `-`, `*`, `/`

- 整型类型之间或从 `float` 或 `double` 到整型类型的显式数字转换。

如果既未指定 `checked`，也未指定 `unchecked`，则非常量表达式（在运行时计算的表达式）的默认上下文将由 `CheckForOverflowUnderflow` 编译器选项的值定义。默认情况下，该选项的值未设置，且算术运算在未选中的上下文中执行。

对于常量表达式（可在编译时完全计算的表达式），将始终选中默认上下文。除非在未选中的上下文中显式放置常量表达式，否则在编译时间计算表达式过程中出现的溢出将导致编译时错误。

请参阅

- [C# 参考](#)
- [C# 编程指南](#)
- [C# 关键字](#)
- [语句关键字](#)

checked (C# 参考)

2020/11/2 • [Edit Online](#)

`checked` 关键字用于对整型类型算术运算和转换显式启用溢出检查。

默认情况下，如果表达式仅包含常量值，且产生的值在目标类型范围之外，则会导致编译器错误。如果表达式包含一个或多个非常量值，则编译器不检测溢出。在下面的示例中，计算赋给 `i2` 的表达式不会导致编译器错误。

```
// The following example causes compiler error CS0220 because 2147483647
// is the maximum value for integers.
//int i1 = 2147483647 + 10;

// The following example, which includes variable ten, does not cause
// a compiler error.
int ten = 10;
int i2 = 2147483647 + ten;

// By default, the overflow in the previous statement also does
// not cause a run-time exception. The following line displays
// -2,147,483,639 as the sum of 2,147,483,647 and 10.
Console.WriteLine(i2);
```

默认情况下，在运行时也不检查这些非常量表达式是否溢出，这些表达式不引发溢出异常。上面的示例显示 -2,147,483,639 作为两个正整数之和。

可以通过编译器选项、环境配置或使用 `checked` 关键字来启用溢出检查。下面的示例演示如何使用 `checked` 表达式或 `checked` 块，在运行时检测由前面的求和计算导致的溢出。两个示例都引发溢出异常。

```
// If the previous sum is attempted in a checked environment, an
// OverflowException error is raised.

// Checked expression.
Console.WriteLine(checked(2147483647 + ten));

// Checked block.
checked
{
    int i3 = 2147483647 + ten;
    Console.WriteLine(i3);
}
```

可以使用 `unchecked` 关键字阻止溢出检查。

示例

此示例演示如何使用 `checked` 启用运行时溢出检查。

```

class OverFlowTest
{
    // Set maxValue to the maximum value for integers.
    static int maxValue = 2147483647;

    // Using a checked expression.
    static int CheckedMethod()
    {
        int z = 0;
        try
        {
            // The following line raises an exception because it is checked.
            z = checked(maxValue + 10);
        }
        catch (System.OverflowException e)
        {
            // The following line displays information about the error.
            Console.WriteLine("CHECKED and CAUGHT: " + e.ToString());
        }
        // The value of z is still 0.
        return z;
    }

    // Using an unchecked expression.
    static int UncheckedMethod()
    {
        int z = 0;
        try
        {
            // The following calculation is unchecked and will not
            // raise an exception.
            z = maxValue + 10;
        }
        catch (System.OverflowException e)
        {
            // The following line will not be executed.
            Console.WriteLine("UNCHECKED and CAUGHT: " + e.ToString());
        }
        // Because of the undetected overflow, the sum of 2147483647 + 10 is
        // returned as -2147483639.
        return z;
    }

    static void Main()
    {
        Console.WriteLine("\nCHECKED output value is: {0}",
            CheckedMethod());
        Console.WriteLine("UNCHECKED output value is: {0}",
            UncheckedMethod());
    }
    /*
Output:
CHECKED and CAUGHT: System.OverflowException: Arithmetic operation resulted
in an overflow.
at ConsoleApplication1.OverFlowTest.CheckedMethod()

CHECKED output value is: 0
UNCHECKED output value is: -2147483639
*/
}

```

C# 语言规范

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

请参阅

- [C# 参考](#)
- [C# 编程指南](#)
- [C# 关键字](#)
- [Checked 和 Unchecked](#)
- [unchecked](#)

unchecked (C# 参考)

2020/11/2 • [Edit Online](#)

`unchecked` 关键字用于取消整型类型的算术运算和转换的溢出检查。

在未经检查的上下文中，如果表达式生成的值超出目标类型的范围，则不会标记溢出。例如，由于以下示例中的计算在 `unchecked` 块或表达式中执行，因此将忽略计算结果对于整数而言过大的事实，并且向 `int1` 赋予值 -2,147,483,639。

```
unchecked
{
    int1 = 2147483647 + 10;
}
int1 = unchecked(ConstantMax + 10);
```

如果删除 `unchecked` 环境，会发生编译错误。由于表达式的所有项都是常量，因此可在编译时检测到溢出。

在编译时和运行时，默认不检查包含非常数项的表达式。请参阅[启用检查](#)，获取有关使用启用了检查的环境的信息。

由于检查溢出需要时间，因此在没有溢出风险的情况下使用取消检查的代码可能会提高性能。但是，如果存在溢出的可能，则应使用启用了检查的环境。

示例

此示例演示如何使用 `unchecked` 关键字。

```

class UncheckedDemo
{
    static void Main(string[] args)
    {
        // int.MaxValue is 2,147,483,647.
        const int ConstantMax = int.MaxValue;
        int int1;
        int int2;
        int variableMax = 2147483647;

        // The following statements are checked by default at compile time. They do not
        // compile.
        //int1 = 2147483647 + 10;
        //int1 = ConstantMax + 10;

        // To enable the assignments to int1 to compile and run, place them inside
        // an unchecked block or expression. The following statements compile and
        // run.
        unchecked
        {
            int1 = 2147483647 + 10;
        }
        int1 = unchecked(ConstantMax + 10);

        // The sum of 2,147,483,647 and 10 is displayed as -2,147,483,639.
        Console.WriteLine(int1);

        // The following statement is unchecked by default at compile time and run
        // time because the expression contains the variable variableMax. It causes
        // overflow but the overflow is not detected. The statement compiles and runs.
        int2 = variableMax + 10;

        // Again, the sum of 2,147,483,647 and 10 is displayed as -2,147,483,639.
        Console.WriteLine(int2);

        // To catch the overflow in the assignment to int2 at run time, put the
        // declaration in a checked block or expression. The following
        // statements compile but raise an overflow exception at run time.
        checked
        {
            //int2 = variableMax + 10;
        }
        //int2 = checked(variableMax + 10);

        // Unchecked sections frequently are used to break out of a checked
        // environment in order to improve performance in a portion of code
        // that is not expected to raise overflow exceptions.
        checked
        {
            // Code that might cause overflow should be executed in a checked
            // environment.
            unchecked
            {
                // This section is appropriate for code that you are confident
                // will not result in overflow, and for which performance is
                // a priority.
            }
            // Additional checked code here.
        }
    }
}

```

C# 语言规范

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

请参阅

- [C# 参考](#)
- [C# 编程指南](#)
- [C# 关键字](#)
- [Checked 和 Unchecked](#)
- [checked](#)

fixed 语句 (C# 参考)

2021/5/7 • [Edit Online](#)

`fixed` 语句可防止垃圾回收器重新定位可移动的变量。`fixed` 语句仅允许存在于**不安全的**上下文中。还可以使用 `fixed` 关键字创建**固定大小的缓冲区**。

`fixed` 语句将为托管变量设置一个指针，并在该语句的执行过程中“单边锁定”该变量。仅可在 `fixed` 上下文中使用指向可移动托管变量的指针。如果没有 `fixed` 上下文，垃圾回收可能会不可预测地重定位变量。C# 编译器只允许将指针分配给 `fixed` 语句中的托管变量。

```
class Point
{
    public int x;
    public int y;
}

unsafe private static void ModifyFixedStorage()
{
    // Variable pt is a managed variable, subject to garbage collection.
    Point pt = new Point();

    // Using fixed allows the address of pt members to be taken,
    // and "pins" pt so that it is not relocated.

    fixed (int* p = &pt.x)
    {
        *p = 1;
    }
}
```

可以通过使用一个数组、字符串、固定大小的缓冲区或变量的地址来初始化指针。以下示例演示变量地址、数组和字符串的使用方式：

```
Point point = new Point();
double[] arr = { 0, 1.5, 2.3, 3.4, 4.0, 5.9 };
string str = "Hello World";

// The following two assignments are equivalent. Each assigns the address
// of the first element in array arr to pointer p.

// You can initialize a pointer by using an array.
fixed (double* p = arr) { /*...*/ }

// You can initialize a pointer by using the address of a variable.
fixed (double* p = &arr[0]) { /*...*/ }

// The following assignment initializes p by using a string.
fixed (char* p = str) { /*...*/ }

// The following assignment is not valid, because str[0] is a char,
// which is a value, not a variable.
//fixed (char* p = &str[0]) { /*...*/ }
```

从 C# 7.3 开始，`fixed` 语句可在数组、字符串、固定大小缓冲区或非托管变量以外的其他类型上执行。实施名为 `GetPinnableReference` 的方法的任何类型都可以被固定。`GetPinnableReference` 必须返回**非托管类型**的 `ref` 变量。<.NET Core 2.0 中引入的 .NET 类型 `System.Span<T>` 和 `System.ReadOnlySpan<T>` 使用此模式，并且可以固

定。下面的示例对此进行了演示：

```
unsafe private static void FixedSpanExample()
{
    int[] PascalsTriangle = {
        1,
        1, 1,
        1, 2, 1,
        1, 3, 3, 1,
        1, 4, 6, 4, 1,
        1, 5, 10, 10, 5, 1
    };

    Span<int> RowFive = new Span<int>(PascalsTriangle, 10, 5);

    fixed (int* ptrToRow = RowFive)
    {
        // Sum the numbers 1,4,6,4,1
        var sum = 0;
        for (int i = 0; i < RowFive.Length; i++)
        {
            sum += *(ptrToRow + i);
        }
        Console.WriteLine(sum);
    }
}
```

如果正在创建应加入此模式的类型，请参阅 [Span<T>.GetPinnableReference\(\)](#) 以查看有关实施此模式的示例。

如果它们都是同一类型，则可以在一个语句中初始化多个指针：

```
fixed (byte* ps = srcarray, pd = dstarray) {...}
```

若要初始化不同类型的指针，只需嵌套 `fixed` 语句，如下面的示例中所示。

```
fixed (int* p1 = &point.x)
{
    fixed (double* p2 = &arr[5])
    {
        // Do something with p1 and p2.
    }
}
```

执行该语句中的代码之后，任何固定的变量都将被解锁并受垃圾回收的约束。因此，请勿指向 `fixed` 语句之外的那些变量。在 `fixed` 语句中声明的变量的作用域为该语句，使此操作更容易：

```
fixed (byte* ps = srcarray, pd = dstarray)
{
    ...
}
// ps and pd are no longer in scope here.
```

在 `fixed` 语句中初始化的指针为只读变量。如果想要修改指针值，必须声明第二个指针变量，并修改它。不能修改在 `fixed` 语句中声明的变量：

```
fixed (byte* ps = srcarray, pd = dstarray)
{
    byte* pSourceCopy = ps;
    pSourceCopy++; // point to the next element.
    ps++; // invalid: cannot modify ps, as it is declared in the fixed statement.
}
```

可以在堆栈上分配内存，在这种情况下，内存不受垃圾回收的约束，因此不需要固定。为此，请使用 [stackalloc 表达式](#)。

C# 语言规范

有关详细信息，请参阅 [C# 语言规范](#) 中的 [fixed 语句](#) 部分。

请参阅

- [C# 参考](#)
- [C# 编程指南](#)
- [C# 关键字](#)
- [unsafe](#)
- [指针类型](#)
- [固定大小的缓冲区](#)

lock 语句 (C# 参考)

2020/11/2 • [Edit Online](#)

`lock` 语句获取给定对象的互斥 lock，执行语句块，然后释放 lock。持有 lock 时，持有 lock 的线程可以再次获取并释放 lock。阻止任何其他线程获取 lock 并等待释放 lock。

`lock` 语句具有以下格式

```
lock (x)
{
    // Your code...
}
```

其中 `x` 是引用类型的表达式。它完全等同于

```
object __lockObj = x;
bool __lockWasTaken = false;
try
{
    System.Threading.Monitor.Enter(__lockObj, ref __lockWasTaken);
    // Your code...
}
finally
{
    if (__lockWasTaken) System.Threading.Monitor.Exit(__lockObj);
}
```

由于该代码使用 `try...finally` 块，即使在 `lock` 语句的正文中引发异常，也会释放 lock。

在 `lock` 语句的正文中不能使用 `await` 运算符。

准则

当同步对共享资源的线程访问时，请锁定专用对象实例(例如，

`private readonly object balanceLock = new object();` 或另一个不太可能被代码无关部分用作 lock 对象的实例。避免对不同的共享资源使用相同的 lock 对象实例，因为这可能导致死锁或锁争用。具体而言，避免将以下对象用作 lock 对象：

- `this` (调用方可能将其用作 lock)。
- `Type` 实例 (可以通过 `typeof` 运算符或反射获取)。
- 字符串实例，包括字符串文本，(这些可能是暂存的)。

尽可能缩短持有锁的时间，以减少锁争用。

示例

以下示例定义了一个 `Account` 类，该类通过锁定专用的 `balanceLock` 实例来同步对其专用 `balance` 字段的访问。使用相同的实例进行锁定可确保尝试同时调用 `Debit` 或 `Credit` 方法的两个线程无法同时更新 `balance` 字段。

```
using System;
using System.Threading.Tasks;
```

```
public class Account
{
    private readonly object balanceLock = new object();
    private decimal balance;

    public Account(decimal initialBalance) => balance = initialBalance;

    public decimal Debit(decimal amount)
    {
        if (amount < 0)
        {
            throw new ArgumentOutOfRangeException(nameof(amount), "The debit amount cannot be negative.");
        }

        decimal appliedAmount = 0;
        lock (balanceLock)
        {
            if (balance >= amount)
            {
                balance -= amount;
                appliedAmount = amount;
            }
        }
        return appliedAmount;
    }

    public void Credit(decimal amount)
    {
        if (amount < 0)
        {
            throw new ArgumentOutOfRangeException(nameof(amount), "The credit amount cannot be negative.");
        }

        lock (balanceLock)
        {
            balance += amount;
        }
    }

    public decimal GetBalance()
    {
        lock (balanceLock)
        {
            return balance;
        }
    }
}

class AccountTest
{
    static async Task Main()
    {
        var account = new Account(1000);
        var tasks = new Task[100];
        for (int i = 0; i < tasks.Length; i++)
        {
            tasks[i] = Task.Run(() => Update(account));
        }
        await Task.WhenAll(tasks);
        Console.WriteLine($"Account's balance is {account.GetBalance()}");
        // Output:
        // Account's balance is 2000
    }

    static void Update(Account account)
    {
        decimal[] amounts = { 0, 2, -3, 6, -2, -1, 8, -5, 11, -6 };
        foreach (var amount in amounts)
        {
```

```
        if (amount >= 0)
    {
        account.Credit(amount);
    }
    else
    {
        account.Debit(Math.Abs(amount));
    }
}
}
```

C# 语言规范

有关详细信息，请参阅 [C# 语言规范](#) 中的 `lock` 语句部分。

请参阅

- [C# 参考](#)
- [C# 关键字](#)
- [System.Threading.Monitor](#)
- [System.Threading.SpinLock](#)
- [System.Threading.Interlocked](#)
- [同步基元概述](#)

方法参数 (C# 参考)

2020/11/2 • [Edit Online](#)

为不具有 `in`、`ref` 或 `out` 的方法声明的参数会按值传递给调用的方法。可以在方法中更改该值，但当控制传递回调用过程时，不会保留更改后的值。可以通过使用方法参数关键字更改此行为。

本部分介绍声明方法参数时可以使用的关键字：

- `params` 指定此参数采用可变数量的参数。
- `in` 指定此参数由引用传递，但只由调用方法读取。
- `ref` 指定此参数由引用传递，可能由调用方法读取或写入。
- `out` 指定此参数由引用传递，由调用方法写入。

请参阅

- [C# 参考](#)
- [C# 编程指南](#)
- [C# 关键字](#)

params (C# 参考)

2020/11/2 • [Edit Online](#)

使用 `params` 关键字可以指定采用数目可变的参数的[方法参数](#)。参数类型必须是一维数组。

在方法声明中的 `params` 关键字之后不允许有任何其他参数，并且在方法声明中只允许有一个 `params` 关键字。

如果 `params` 参数的声明类型不是一维数组，则会发生编译器错误 [CS0225](#)。

使用 `params` 参数调用方法时，可以传入：

- 数组元素类型的参数的逗号分隔列表。
- 指定类型的参数的数组。
- 无参数。如果未发送任何参数，则 `params` 列表的长度为零。

示例

下面的示例演示可向 `params` 形参发送实参的各种方法。

```
public class MyClass
{
    public static void UseParams(params int[] list)
    {
        for (int i = 0; i < list.Length; i++)
        {
            Console.Write(list[i] + " ");
        }
        Console.WriteLine();
    }

    public static void UseParams2(params object[] list)
    {
        for (int i = 0; i < list.Length; i++)
        {
            Console.Write(list[i] + " ");
        }
        Console.WriteLine();
    }

    static void Main()
    {
        // You can send a comma-separated list of arguments of the
        // specified type.
        UseParams(1, 2, 3, 4);
        UseParams2(1, 'a', "test");

        // A params parameter accepts zero or more arguments.
        // The following calling statement displays only a blank line.
        UseParams2();

        // An array argument can be passed, as long as the array
        // type matches the parameter type of the method being called.
        int[] myIntArray = { 5, 6, 7, 8, 9 };
        UseParams(myIntArray);

        object[] myObjArray = { 2, 'b', "test", "again" };
        UseParams2(myObjArray);

        // The following call causes a compiler error because the object
        // array cannot be converted into an integer array.
        //UseParams(myObjArray);

        // The following call does not cause an error, but the entire
        // integer array becomes the first element of the params array.
        UseParams2(myIntArray);
    }
}

/*
Output:
1 2 3 4
1 a test

5 6 7 8 9
2 b test again
System.Int32[]
*/
```

C# 语言规范

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

请参阅

- [C# 参考](#)
- [C# 编程指南](#)
- [C# 关键字](#)
- [方法参数](#)

in 参数修饰符 (C# 参考)

2021/3/22 • [Edit Online](#)

`in` 关键字会导致按引用传递参数，但确保未修改参数。它让形参成为实参的别名，这必须是变量。换而言之，对形参执行的任何操作都是对实参执行的。它类似于 `ref` 或 `out` 关键字，不同之处在于 `in` 参数无法通过调用的方法进行修改。`out` 参数必须由调用的方法进行修改，这些修改在调用上下文中是可观察的，而 `ref` 参数是可以修改的。

```
int readonlyArgument = 44;
InArgExample(readonlyArgument);
Console.WriteLine(readonlyArgument);      // value is still 44

void InArgExample(in int number)
{
    // Uncomment the following line to see error CS8331
    //number = 19;
}
```

前面的示例说明调用站点处通常不需要 `in` 修饰符。仅在方法声明中需要它。

NOTE

`in` 关键字还作为 `foreach` 语句的一部分，或作为 LINQ 查询中 `join` 子句的一部分，与泛型类型参数一起使用来指定该类型参数为逆变。有关在这些上下文使用 `in` 关键字的详细信息，请参阅 [in](#)，其中提供了所有这些用法的链接。

作为 `in` 参数传递的变量在方法调用中传递之前必须进行初始化。但是，所调用的方法可能不会分配值或修改参数。

`in` 参数修饰符可在 C# 7.2 及更高版本中使用。以前的版本生成编译器错误 `CS8107` ("‘readonly 引用’功能在 C# 7.0 中不可用。请使用语言版本 7.2 或更高版本。") 若要配置编译器语言版本，请参阅 [选择 C# 语言版本](#)。

`in`、`ref` 和 `out` 关键字不被视为用于重载决议的方法签名的一部分。因此，如果唯一的不同是一个方法采用 `ref` 或 `in` 参数，而另一个方法采用 `out` 参数，则无法重载这两个方法。例如，以下代码将不会编译：

```
class CS0663_Example
{
    // Compiler error CS0663: "Cannot define overloaded
    // methods that differ only on in, ref and out".
    public void SampleMethod(in int i) { }
    public void SampleMethod(ref int i) { }
}
```

存在 `in` 时可以重载：

```
class InOverloads
{
    public void SampleMethod(in int i) { }
    public void SampleMethod(int i) { }
}
```

重载决策规则

通过理解使用 `in` 参数的动机，可以理解使用按值方法和使用 `in` 参数方法的重载决策规则。定义使用 `in` 参数的方法是一项潜在的性能优化。某些 `struct` 类型参数可能很大，在紧凑的循环或关键代码路径中调用方法时，复制这些结构的成本就很高。方法声明 `in` 参数以指定参数可能按引用安全传递，因为所调用的方法不修改该参数的状态。按引用传递这些参数可以避免（可能产生的）高昂的复制成本。

为调用站点上的参数指定 `in` 通常为可选。按值传递参数和使用 `in` 修饰符按引用传递参数这两种方法并没有语义差异。可以在调用站点选择 `in` 修饰符，因为你不需要指出参数值可能会改变。在调用站点显式添加 `in` 修饰符以确保参数是按引用传递，而不是按值传递。显式使用 `in` 有以下两个效果：

首先，在调用站点指定 `in` 会强制编译器选择使用匹配的 `in` 参数定义的方法。否则，如果两种方法唯一的区别在于是否存在 `in`，则按值重载的匹配度会更高。

第二点，指定 `in` 会声明你想按引用传递参数。结合 `in` 使用的参数必须代表一个可以直接引用的位置。`out` 和 `ref` 参数的相同常规规则适用：不得使用常量、普通属性或其他生成值的表达式。否则，在调用站点省略 `in` 就会通知编译器你将允许它创建临时变量，并按只读引用传递至方法。编译器创建临时变量以克服一些 `in` 参数的限制：

- 临时变量允许将编译时常数作为 `in` 参数。
- 临时变量允许使用属性或 `in` 参数的其他表达式。
- 存在从实参类型到形参类型的隐式转换时，临时变量允许使用实参。

在前面的所有实例中，编译器创建了临时变量，用于存储常数、属性或其他表达式的值。

以下代码阐释了这些规则：

```
static void Method(in int argument)
{
    // implementation removed
}

Method(5); // OK, temporary variable created.
Method(5L); // CS1503: no implicit conversion from long to int
short s = 0;
Method(s); // OK, temporary int created with the value 0
Method(in s); // CS1503: cannot convert from in short to in int
int i = 42;
Method(i); // passed by readonly reference
Method(in i); // passed by readonly reference, explicitly using `in`
```

现在，假设可以使用另一种使用按值参数的方法。结果的变化如以下代码所示：

```
static void Method(int argument)
{
    // implementation removed
}

static void Method(in int argument)
{
    // implementation removed
}

Method(5); // Calls overload passed by value
Method(5L); // CS1503: no implicit conversion from long to int
short s = 0;
Method(s); // Calls overload passed by value.
Method(in s); // CS1503: cannot convert from in short to in int
int i = 42;
Method(i); // Calls overload passed by value
Method(in i); // passed by readonly reference, explicitly using `in`
```

最后一个是按引用传递参数的唯一方法调用。

NOTE

为了简化操作，前面的代码将 `int` 用作参数类型。因为大多数新式计算机中的引用都比 `int` 大，所以将单个 `int` 作为只读引用传递没有任何好处。

in 参数的限制

不能将 `in`、`ref` 和 `out` 关键字用于以下几种方法：

- 异步方法，通过使用 `async` 修饰符定义。
- 迭代器方法，包括 `yield return` 或 `yield break` 语句。
- 扩展方法的第一个参数不能有 `in` 修饰符，除非该参数是结构。
- 扩展方法的第一个参数，其中该参数是泛型类型（即使该类型被约束为结构。）

若要详细了解 `in` 修饰符及其与 `ref` 和 `out` 的区别，可查看[编写安全有效的代码](#)一文。

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。

ref (C# 参考)

2021/5/7 • • [Edit Online](#)

`ref` 关键字指示按引用传递值。它用在四种不同的上下文中：

- 在方法签名和方法调用中，按引用将参数传递给方法。有关详细信息，请参阅[按引用传递参数](#)。
- 在方法签名中，按引用将值返回给调用方。有关详细信息，请参阅[引用返回值](#)。
- 在成员正文中，指示引用返回值是否作为调用方欲修改的引用被存储在本地。或指示局部变量按引用访问另一个值。有关详细信息，请参阅[Ref 局部变量](#)。
- 在 `struct` 声明中，声明 `ref struct` 或 `readonly ref struct`。有关详细信息，请参阅[结构类型](#)一文中的 `ref` 结构一节。

按引用传递参数

在方法的参数列表中使用 `ref` 关键字时，它指示参数按引用传递，而非按值传递。`ref` 关键字让形参成为实参的别名，这必须是变量。换而言之，对形参执行的任何操作都是对实参执行的。

例如，假设调用方传递了局部变量表达式或数组元素访问表达式。然后，调用的方法可以替换 `ref` 参数引用的对象。在这种情况下，调用方的局部变量或数组元素会在该方法返回时引用新的对象。

NOTE

不要混淆通过引用传递的概念与引用类型的概念。这两种概念是不同的。无论方法参数是值类型还是引用类型，均可由 `ref` 修改。当通过引用传递时，不会对值类型装箱。

若要使用 `ref` 参数，方法定义和调用方法均必须显式使用 `ref` 关键字，如下面的示例所示。（除了在进行 COM 调用时，调用方法可忽略 `ref`。）

```
void Method(ref int refArgument)
{
    refArgument = refArgument + 44;
}

int number = 1;
Method(ref number);
Console.WriteLine(number);
// Output: 45
```

传递到 `ref` 或 `in` 形参的实参必须先经过初始化，然后才能传递。该要求与 `out` 形参不同，在传递之前，不需要显式初始化该形参的实参。

类的成员不能具有仅在 `ref`、`in` 或 `out` 方面不同的签名。如果类型的两个成员之间的唯一区别在于其中一个具有 `ref` 参数，而另一个具有 `out` 或 `in` 参数，则会发生编译器错误。例如，以下代码将不会编译。

```
class CS0663_Example
{
    // Compiler error CS0663: "Cannot define overloaded
    // methods that differ only on ref and out".
    public void SampleMethod(out int i) { }
    public void SampleMethod(ref int i) { }
}
```

但是，当一个方法具有 `ref`、`in` 或 `out` 参数，另一个方法具有值传递的参数时，则可以重载方法，如下面的示例所示。

```
class RefOverloadExample
{
    public void SampleMethod(int i) { }
    public void SampleMethod(ref int i) { }
}
```

在其他要求签名匹配的情况下（如隐藏或重写），`in`、`ref` 和 `out` 是签名的一部分，相互之间不匹配。

属性不是变量。它们是方法，不能传递到 `ref` 参数。

不能将 `ref`、`in` 和 `out` 关键字用于以下几种方法：

- 异步方法，通过使用 `async` 修饰符定义。
- 迭代器方法，包括 `yield return` 或 `yield break` 语句。

[扩展方法](#)还限制了对以下这些关键字的使用：

- 不能对扩展方法的第一个参数使用 `out` 关键字。
- 当参数不是结构或是不被约束为结构的泛型类型时，不能对扩展方法的第一个参数使用 `ref` 关键字。
- 除非第一个参数是结构，否则不能使用 `in` 关键字。即使约束为结构，也不能对任何泛型类型使用 `in` 关键字。

按引用传递参数：示例

前面的示例按引用传递值类型。还可使用 `ref` 关键字按引用传递引用类型。按引用传递引用类型使所调用方能够替换调用方中引用参数引用的对象。对象的存储位置按引用参数的值传递到方法。如果更改参数存储位置中的值（以指向新对象），你还可以将存储位置更改为调用方所引用的位置。下面的示例将引用类型的实例作为 `ref` 参数传递。

```

class Product
{
    public Product(string name, int newID)
    {
        ItemName = name;
        ItemID = newID;
    }

    public string ItemName { get; set; }
    public int ItemID { get; set; }
}

private static void ChangeByReference(ref Product itemRef)
{
    // Change the address that is stored in the itemRef parameter.
    itemRef = new Product("Stapler", 99999);

    // You can change the value of one of the properties of
    // itemRef. The change happens to item in Main as well.
    itemRef.ItemID = 12345;
}

private static void ModifyProductsByReference()
{
    // Declare an instance of Product and display its initial values.
    Product item = new Product("Fasteners", 54321);
    System.Console.WriteLine("Original values in Main. Name: {0}, ID: {1}\n",
        item.ItemName, item.ItemID);

    // Pass the product instance to ChangeByReference.
    ChangeByReference(ref item);
    System.Console.WriteLine("Back in Main. Name: {0}, ID: {1}\n",
        item.ItemName, item.ItemID);
}

// This method displays the following output:
// Original values in Main. Name: Fasteners, ID: 54321
// Back in Main. Name: Stapler, ID: 12345

```

有关如何通过值和引用传递引用类型的详细信息，请参阅[传递引用类型参数](#)。

引用返回值

引用返回值（或 `ref` 返回值）是由方法按引用向调用方返回的值。即是说，调用方可以修改方法所返回的值，此更改反映在调用方法中的对象的状态中。

使用 `ref` 关键字来定义引用返回值：

- 在方法签名中。例如，下列方法签名指示 `GetCurrentPrice` 方法按引用返回了 `Decimal` 值。

```
public ref decimal GetCurrentPrice()
```

- 在 `return` 标记和方法的 `return` 语句中返回的变量之间。例如：

```
return ref DecimalArray[0];
```

为方便调用方修改对象的状态，引用返回值必须存储在被显式定义为 `ref 局部变量` 的变量中。

下面是一个更完整的 `ref` 返回示例，同时显示方法签名和方法主体。

```
public static ref int Find(int[,] matrix, Func<int, bool> predicate)
{
    for (int i = 0; i < matrix.GetLength(0); i++)
        for (int j = 0; j < matrix.GetLength(1); j++)
            if (predicate(matrix[i, j]))
                return ref matrix[i, j];
    throw new InvalidOperationException("Not found");
}
```

所调用方法还可能会将返回值声明为 `ref readonly` 以按引用返回值，并坚持调用代码无法修改返回的值。调用方法可以通过将返回值存储在本地 `ref readonly` 变量中来避免复制该值。

有关示例，请参阅 [ref 返回值和 ref 局部变量示例](#)。

ref 局部变量

`ref` 局部变量用于指代使用 `return ref` 返回的值。无法将 `ref` 局部变量初始化为非 `ref` 返回值。也就是说，初始化的右侧必须为引用。任何对 `ref` 本地变量值的修改都将反映在对象的状态中，该对象的方法按引用返回值。

可在以下两个位置使用 `ref` 关键字来定义 `ref` 局部变量：

- 在变量声明之前。
- 紧接在调用按引用返回值的方法之前。

例如，下列语句定义名为 `GetEstimatedValue` 的方法返回的 `ref` 局部变量值：

```
ref decimal estValue = ref Building.GetEstimatedValue();
```

可通过相同方式按引用访问值。在某些情况下，按引用访问值可避免潜在的高开销复制操作，从而提高性能。例如，以下语句显示如何定义一个用于引用值的 `ref` 局部变量。

```
ref VeryLargeStruct reflocal = ref veryLargeStruct;
```

在这两个示例中，必须在两个位置同时使用 `ref` 关键字，否则编译器将生成错误 CS8172：“无法使用值对按引用变量进行初始化”。

从 C# 7.3 开始，`foreach` 语句的迭代变量可以是 `ref` 局部变量，也可以是 `ref readonly` 局部变量。有关详细信息，请参阅 [foreach 语句](#) 一文。

此外，从 C# 7.3 开始，可以使用 `ref 赋值运算符` 重新分配 `ref local` 或 `ref readonly local` 变量。

Ref readonly 局部变量

`Ref readonly` 局部变量用于指代在其签名中具有 `ref readonly` 并使用 `return ref` 的方法或属性返回的值。`ref readonly` 变量将 `ref` 局部变量的属性与 `readonly` 变量结合使用：它是所分配到的存储的别名，且无法修改。

ref 返回值和 ref 局部变量示例

下列示例定义一个具有两个 `String` 字段 (`Title` 和 `Author`) 的 `Book` 类。还定义包含 `Book` 对象的专用数组的 `BookCollection` 类。通过调用 `GetBookByTitle` 方法，可按引用返回个别 `book` 对象。

```

public class Book
{
    public string Author;
    public string Title;
}

public class BookCollection
{
    private Book[] books = { new Book { Title = "Call of the Wild, The", Author = "Jack London" },
                            new Book { Title = "Tale of Two Cities, A", Author = "Charles Dickens" } };
    private Book nobook = null;

    public ref Book GetBookByTitle(string title)
    {
        for (int ctr = 0; ctr < books.Length; ctr++)
        {
            if (title == books[ctr].Title)
                return ref books[ctr];
        }
        return ref nobook;
    }

    public void ListBooks()
    {
        foreach (var book in books)
        {
            Console.WriteLine($"{book.Title}, by {book.Author}");
        }
        Console.WriteLine();
    }
}

```

调用方将 `GetBookByTitle` 方法所返回的值存储为 `ref` 局部变量时，调用方对返回值所做的更改将反映在 `BookCollection` 对象中，如下例所示。

```

var bc = new BookCollection();
bc.ListBooks();

ref var book = ref bc.GetBookByTitle("Call of the Wild, The");
if (book != null)
    book = new Book { Title = "Republic, The", Author = "Plato" };
bc.ListBooks();
// The example displays the following output:
//      Call of the Wild, The, by Jack London
//      Tale of Two Cities, A, by Charles Dickens
//
//      Republic, The, by Plato
//      Tale of Two Cities, A, by Charles Dickens

```

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

- [编写安全高效的代码](#)
- [ref 返回结果和局部变量](#)
- [条件 ref 表达式](#)
- [传递参数](#)

- [方法参数](#)
- [C# 参考](#)
- [C# 编程指南](#)
- [C# 关键字](#)

out 参数修饰符 (C# 参考)

2020/11/2 • [Edit Online](#)

`out` 关键字通过引用传递参数。它让形参成为实参的别名，这必须是变量。换而言之，对形参执行的任何操作都是对实参执行的。它与 `ref` 关键字相似，只不过 `ref` 要求在传递之前初始化变量。它也类似于 `in` 关键字，只不过 `in` 不允许通过调用方法来修改参数值。若要使用 `out` 参数，方法定义和调用方法均必须显式使用 `out` 关键字。例如：

```
int initializeInMethod;
OutArgExample(out initializeInMethod);
Console.WriteLine(initializeInMethod);      // value is now 44

void OutArgExample(out int number)
{
    number = 44;
}
```

NOTE

`out` 关键字也可与泛型类型参数结合使用，以指定该类型参数是协变参数。有关在此上下文中使用 `out` 关键字的详细信息，请参阅 [out\(泛型修饰符\)](#)。

作为 `out` 参数传递的变量在方法调用中传递之前不必进行初始化。但是，被调用的方法需要在返回之前赋一个值。

`in`、`ref` 和 `out` 关键字不被视为用于重载决议的方法签名的一部分。因此，如果唯一的不同是一个方法采用 `ref` 或 `in` 参数，而另一个方法采用 `out` 参数，则无法重载这两个方法。例如，以下代码将不会编译：

```
class CS0663_Example
{
    // Compiler error CS0663: "Cannot define overloaded
    // methods that differ only on ref and out".
    public void SampleMethod(out int i) { }
    public void SampleMethod(ref int i) { }
}
```

但是，如果一个方法采用 `ref`、`in` 或 `out` 参数，而另一个方法采用其他参数，则可以完成重载，如：

```
class OutOverloadExample
{
    public void SampleMethod(int i) { }
    public void SampleMethod(out int i) => i = 5;
}
```

编译器通过将调用站点上的参数修饰符与方法调用中使用的参数修饰符进行匹配，从而选择最佳重载。

属性不是变量，因此不能作为 `out` 参数传递。

不能将 `in`、`ref` 和 `out` 关键字用于以下几种方法：

- 异步方法，通过使用 `async` 修饰符定义。

- 迭代器方法，包括 `yield return` 或 `yield break` 语句。

此外，[扩展方法](#)具有以下限制：

- 不能对扩展方法的第一个参数使用 `out` 关键字。
- 当参数不是结构或是不被约束为结构的泛型类型时，不能对扩展方法的第一个参数使用 `ref` 关键字。
- 除非第一个参数是结构，否则不能使用 `in` 关键字。即使约束为结构，也不能对任何泛型类型使用 `in` 关键字。

声明 `out` 参数

使用 `out` 参数声明方法是返回多个值的经典解决方法。自 C# 7.0 起，可以考虑针对类似方案使用[值元组](#)。下面的示例使用 `out` 返回具有单个方法调用的三个变量。第三个参数分配为 `null`。这使得方法可以选择地返回值。

```
void Method(out int answer, out string message, out string stillNull)
{
    answer = 44;
    message = "I've been returned";
    stillNull = null;
}

int argNumber;
string argMessage, argDefault;
Method(out argNumber, out argMessage, out argDefault);
Console.WriteLine(argNumber);
Console.WriteLine(argMessage);
Console.WriteLine(argDefault == null);

// The example displays the following output:
//      44
//      I've been returned
//      True
```

调用具有 `out` 参数的方法

在 C# 6 及更早版本中，必须先在单独的语句中声明变量，然后才能将其作为 `out` 参数传递。下面的示例先声明了变量 `number`，然后再将它传递给将字符串转换为数字的 `Int32.TryParse` 方法。

```
string numberAsString = "1640";

int number;
if (Int32.TryParse(numberAsString, out number))
    Console.WriteLine($"Converted '{numberAsString}' to {number}");
else
    Console.WriteLine($"Unable to convert '{numberAsString}'");
// The example displays the following output:
//      Converted '1640' to 1640
```

从 C# 7.0 开始，可以在方法调用的参数列表而不是单独的变量声明中声明 `out` 变量。这使得代码更简洁可读，还能防止在方法调用之前无意中向该变量赋值。下面的示例与上一个示例基本相同，不同之处在于它在对 `Int32.TryParse` 方法的调用中定义了 `number` 变量。

```
string numberAsString = "1640";

if (Int32.TryParse(numberAsString, out int number))
    Console.WriteLine($"Converted '{numberAsString}' to {number}");
else
    Console.WriteLine($"Unable to convert '{numberAsString}'");
// The example displays the following output:
//      Converted '1640' to 1640
```

在上一个示例中，`number` 变量被强类型化为 `int`。你也可以声明一个隐式类型本地变量，如以下示例所示。

```
string numberAsString = "1640";

if (Int32.TryParse(numberAsString, out var number))
    Console.WriteLine($"Converted '{numberAsString}' to {number}");
else
    Console.WriteLine($"Unable to convert '{numberAsString}'");
// The example displays the following output:
//      Converted '1640' to 1640
```

C# 语言规范

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

请参阅

- [C# 参考](#)
- [C# 编程指南](#)
- [C# 关键字](#)
- [方法参数](#)

命名空间 (C# 参考)

2020/11/2 • [Edit Online](#)

`namespace` 关键字用于声明包含一组相关对象的作用域。可以使用命名空间来组织代码元素并创建全局唯一类型。

```
namespace SampleNamespace
{
    class SampleClass { }

    interface ISampleInterface { }

    struct SampleStruct { }

    enum SampleEnum { a, b }

    delegate void SampleDelegate(int i);

    namespace Nested
    {
        class SampleClass2 { }
    }
}
```

备注

在命名空间中，可以声明零个或多个以下类型：

- 另一个命名空间
- [class](#)
- [interface](#)
- [struct](#)
- [enum](#)
- [delegate](#)

无论是否在 C# 源文件中显式声明命名空间，编译器都会添加一个默认命名空间。此未命名的命名空间（有时被称为全局命名空间）存在于每个文件中。全局命名空间中的任何标识符都可用于已命名的命名空间。

命名空间隐式具有公共访问权限，这是不可修改的。有关可以分配给命名空间中元素的访问修饰符的讨论，请参阅[访问修饰符](#)。

可以在两个或多个声明中定义一个命名空间。例如，以下示例将两个类定义为 `MyCompany` 命名空间的一部分：

```
namespace MyCompany.Proj1
{
    class MyClass
    {
    }
}

namespace MyCompany.Proj1
{
    class MyClass1
    {
    }
}
```

示例

以下示例显示如何在嵌套命名空间中调用静态方法。

```
namespace SomeNameSpace
{
    public class MyClass
    {
        static void Main()
        {
            Nested.NestedNameSpaceClass.SayHello();
        }
    }

    // a nested namespace
    namespace Nested
    {
        public class NestedNameSpaceClass
        {
            public static void SayHello()
            {
                Console.WriteLine("Hello");
            }
        }
    }
}

// Output: Hello
```

C# 语言规范

有关详细信息，请参阅 [C# 语言规范](#) 中的 [命名空间](#) 部分。

请参阅

- [C# 参考](#)
- [C# 关键字](#)
- [using](#)
- [using static](#)
- [命名空间别名限定符 ::](#)
- [命名空间](#)

using (C# 参考)

2021/3/5 • [Edit Online](#)

`using` 关键字有三个主要用途：

- [using 语句](#) 定义一个范围，在此范围的末尾将释放对象。
- [using 指令](#) 为命名空间创建别名，或导入在其他命名空间中定义的类型。
- [using static 指令](#) 导入单个类的成员。

请参阅

- [C# 参考](#)
- [C# 编程指南](#)
- [C# 关键字](#)
- [命名空间](#)
- [extern](#)

using 指令 (C# 参考)

2021/3/5 • • [Edit Online](#)

using 指令有三种用途：

- 允许在命名空间中使用类型，这样无需在该命名空间中限定某个类型的使用：

```
using System.Text;
```

- 允许访问类型的静态成员和嵌套类型，而无需限定使用类型名称进行访问。

```
using static System.Math;
```

有关详细信息，请参阅 [using static 指令](#)。

- 为命名空间或类型创建别名。这称为 *using 别名指令*。

```
using Project = PC.MyCompany.Project;
```

using 关键字还用于创建 using 语句，此类语句有助于确保正确处理 [IDisposable](#) 对象（如文件和字体）。有关详细信息，请参阅 [using 语句](#)。

Using 静态类型

你可以访问类型的静态成员，而无需限定使用类型名称进行访问：

```
using static System.Console;
using static System.Math;
class Program
{
    static void Main()
    {
        WriteLine(Sqrt(3*3 + 4*4));
    }
}
```

备注

using 指令的范围限于显示它的文件。

可能出现 using 指令的位置：

- 源代码文件的开头，位于任何命名空间或类型定义之前。
- 在任何命名空间中，但位于此命名空间中声明的任何命名空间或类型之前。

否则，将生成编译器错误 [CS1529](#)。

创建 using 别名指令，以便更易于将标识符限定为命名空间或类型。在任何 using 指令中，都必须使用完全限定的命名空间或类型，而无需考虑它之前的 using 指令。using 指令的声明中不能使用 using 别名。例如，以下代码生成一个编译器错误：

```
using s = System.Text;
using s.RegularExpressions; // Generates a compiler error.
```

创建 `using` 指令，以便在命名空间中使用类型而不必指定命名空间。`using` 指令不为你提供对嵌套在指定命名空间中的任何命名空间的访问权限。

命名空间分为两类：用户定义的命名空间和系统定义的命名空间。用户定义的命名空间是在代码中定义的命名空间。有关系统定义的命名空间的列表，请参阅 [.NET API 浏览器](#)。

示例 1

下面的示例显示如何为命名空间定义和使用 `using` 别名：

```
namespace PC
{
    // Define an alias for the nested namespace.
    using Project = PC.MyCompany.Project;
    class A
    {
        void M()
        {
            // Use the alias
            var mc = new Project.MyClass();
        }
    }
    namespace MyCompany
    {
        namespace Project
        {
            public class MyClass { }
        }
    }
}
```

`using` 别名指令的右侧不能有开放式泛型类型。例如，不能为 `List<T>` 创建 `using` 别名，但可以为 `List<int>` 创建 `using` 别名。

示例 2

下面的示例显示如何为类定义 `using` 指令和 `using` 别名：

```
using System;

// Using alias directive for a class.
using AliasToMyClass = NameSpace1.MyClass;

// Using alias directive for a generic class.
using UsingAlias = NameSpace2.MyClass<int>;

namespace NameSpace1
{
    public class MyClass
    {
        public override string ToString()
        {
            return "You are in NameSpace1.MyClass.";
        }
    }
}

namespace NameSpace2
{
    class MyClass<T>
    {
        public override string ToString()
        {
            return "You are in NameSpace2.MyClass.";
        }
    }
}

namespace NameSpace3
{
    class MainClass
    {
        static void Main()
        {
            var instance1 = new AliasToMyClass();
            Console.WriteLine(instance1);

            var instance2 = new UsingAlias();
            Console.WriteLine(instance2);
        }
    }
}
// Output:
//     You are in NameSpace1.MyClass.
//     You are in NameSpace2.MyClass.
```

C# 语言规范

有关详细信息，请参阅 [C# 语言规范](#) 中的 [Using 指令](#)。该语言规范是 C# 语法和用法的权威资料。

请参阅

- [C# 参考](#)
- [C# 编程指南](#)
- [使用命名空间](#)
- [C# 关键字](#)
- [命名空间](#)
- [using 语句](#)

using 静态指令 (C# 参考)

2020/11/2 • [Edit Online](#)

`using static` 指令指定一种类型，无需指定类型名称即可访问其静态成员和嵌套类型。语法为：

```
using static <fully-qualified-type-name>;
```

其中，`fully-qualified-type-name` 是无需指定类型名称即可访问其静态成员和嵌套类型的类型名称。如果你不提供完全限定的类型名称(完整的命名空间名称以及类型名称)，C# 便会生成编译器错误 [CS0246](#)：“找不到类型名称或命名空间名称 ‘type/namespace’ (是否缺少 using 指令或程序集引用?)”。

`using static` 指令适用于任何具有静态成员(或嵌套类型)的类型，即使该类型还具有实例成员。但是，只能通过类型实例来调用实例成员。

`using static` 指令是在 C# 6 中引入的。

备注

通常，调用某个静态成员时，即会提供类型名称以及成员名称。重复输入相同的类型名称来调用该类型的成员将生成详细的晦涩代码。例如，`Circle` 类的以下定义引用 `Math` 类的成员数。

```
using System;

public class Circle
{
    public Circle(double radius)
    {
        Radius = radius;
    }

    public double Radius { get; set; }

    public double Diameter
    {
        get { return 2 * Radius; }
    }

    public double Circumference
    {
        get { return 2 * Radius * Math.PI; }
    }

    public double Area
    {
        get { return Math.PI * Math.Pow(Radius, 2); }
    }
}
```

通过消除每次引用成员时，显式引用 `Math` 类的需求，`using static` 指令将生成更简洁的代码：

```
using System;
using static System.Math;

public class Circle
{
    public Circle(double radius)
    {
        Radius = radius;
    }

    public double Radius { get; set; }

    public double Diameter
    {
        get { return 2 * Radius; }
    }

    public double Circumference
    {
        get { return 2 * Radius * PI; }
    }

    public double Area
    {
        get { return PI * Pow(Radius, 2); }
    }
}
```

`using static` 仅导入可访问的静态成员和指定类型中声明的嵌套类型。不导入继承的成员。可以从任何带 `using static` 指令的已命名类型导入，包括 Visual Basic 模块。如果 F# 顶级函数在元数据中显示为一个已命名类型（其名称是有效的 C# 标识符）的静态成员，则可以导入该 F# 函数。

`using static` 使指定类型中声明的扩展方法可用于扩展方法查找。但是，扩展方法的名称不导入到代码中非限定引用的作用域中。

同一编译单元或命名空间中通过不同 `using static` 命令从不同类型导入的具有相同名称的方法组成一个方法组。这些方法组内的重载解决方法遵循一般 C# 规则。

示例

以下示例使用 `using static` 指令来提供 `Console`、`Math` 和 `String` 类的静态成员，而无需指定其类型名称。

```

using System;
using static System.Console;
using static System.Math;
using static System.String;

class Program
{
    static void Main()
    {
        Write("Enter a circle's radius: ");
        var input = ReadLine();
        if (!IsNullOrEmpty(input) && double.TryParse(input, out var radius)) {
            var c = new Circle(radius);

            string s = "\nInformation about the circle:\n";
            s = s + Format("    Radius: {0:N2}\n", c.Radius);
            s = s + Format("    Diameter: {0:N2}\n", c.Diameter);
            s = s + Format("    Circumference: {0:N2}\n", c.Circumference);
            s = s + Format("    Area: {0:N2}\n", c.Area);
            WriteLine(s);
        }
        else {
            WriteLine("Invalid input...");
        }
    }
}

public class Circle
{
    public Circle(double radius)
    {
        Radius = radius;
    }

    public double Radius { get; set; }

    public double Diameter
    {
        get { return 2 * Radius; }
    }

    public double Circumference
    {
        get { return 2 * Radius * PI; }
    }

    public double Area
    {
        get { return PI * Pow(Radius, 2); }
    }
}
// The example displays the following output:
//      Enter a circle's radius: 12.45
//
//      Information about the circle:
//          Radius: 12.45
//          Diameter: 24.90
//          Circumference: 78.23
//          Area: 486.95

```

在此示例中，`using static` 指令也已经应用于 `Double` 类型。这使得在未指定类型名称情况下调用 `TryParse(String, Double)` 方法成为可能。但是，如此创建的代码可读性较差，因为这样有必要检查 `using static` 指令，以确定所调用的数值类型的 `TryParse` 方法。

请参阅

- [using 指令](#)
- [C# 参考](#)
- [C# 关键字](#)
- [使用命名空间](#)
- [命名空间](#)

using 语句 (C# 参考)

2021/3/5 • [Edit Online](#)

提供可确保正确使用 [IDisposable](#) 对象的方便语法。从 C#8.0 开始，`using` 语句可确保正确使用 [IAsyncDisposable](#) 对象。

示例

下面的示例演示如何使用 `using` 语句。

```
string manyLines=@"This is line one
This is line two
Here is line three
The penultimate line is line four
This is the final, fifth line.";

using (var reader = new StringReader(manyLines))
{
    string? item;
    do {
        item = reader.ReadLine();
        Console.WriteLine(item);
    } while(item != null);
}
```

从 C# 8.0 开始，可以对不需要使用大括号的 `using` 语句使用以下替代语法：

```
string manyLines=@"This is line one
This is line two
Here is line three
The penultimate line is line four
This is the final, fifth line.";

using var reader = new StringReader(manyLines);
string? item;
do {
    item = reader.ReadLine();
    Console.WriteLine(item);
} while(item != null);
```

备注

[File](#) 和 [Font](#) 是访问非托管资源(本例中为文件句柄和设备上下文)的托管类型的示例。有许多其他类别的非托管资源和封装这些资源的类库类型。所有此类类型都必须实现 [IDisposable](#) 接口或 [IAsyncDisposable](#) 接口。

[IDisposable](#) 对象的生存期限于单个方法时，应在 `using` 语句中声明并实例化它。`using` 语句按照正确的方式调用对象上的 [Dispose](#) 方法，并(在按照前面所示方式使用它时)会导致在调用 [Dispose](#) 时对象自身处于范围之外。在 `using` 块中，对象是只读的并且无法进行修改或重新分配。如果对象实现 [IAsyncDisposable](#) 而不是 [IDisposable](#)，`using` 语句将调用 [DisposeAsync](#) 和 [awaits](#) 返回的 [ValueTask](#)。有关 [IAsyncDisposable](#) 的详细信息，请参阅[实现 DisposeAsync 方法](#)。

`using` 语句可确保调用 [Dispose](#) 或 [DisposeAsync](#)，即使 `using` 块中发生异常也是如此。通过将对象放入 `try` 块中，然后调用 `finally` 块中的 [Dispose](#)(或 [DisposeAsync](#))，可以实现相同的结果；实际上，这就是编译器转换 `using` 语句的方式。前面的代码示例在编译时将扩展到以下代码(请注意，使用额外的大括号为对象创建有限范

围)：

```
string manyLines=@"This is line one
This is line two
Here is line three
The penultimate line is line four
This is the final, fifth line.";

{
    var reader = new StringReader(manyLines);
    try {
        string? item;
        do {
            item = reader.ReadLine();
            Console.WriteLine(item);
        } while(item != null);
    } finally
    {
        reader?.Dispose();
    }
}
```

较新的 `using` 语句语法转换为类似的代码。`try` 块在声明变量的位置打开。`finally` 块添加在封闭块的末尾，通常是在方法的末尾。

有关 `try - finally` 语句的详细信息，请参阅 [try-finally](#) 一文。

可在单个 `using` 语句中声明一个类型的多个实例，如下面的示例中所示。注意，在单个语句中声明多个变量时，不能使用隐式类型的变量 (`var`)：

```
string numbers=@"One
Two
Three
Four.";
string letters=@"A
B
C
D.";

using (StringReader left = new StringReader(numbers),
       right = new StringReader(letters))
{
    string? item;
    do {
        item = left.ReadLine();
        Console.Write(item);
        Console.Write("    ");
        item = right.ReadLine();
        Console.WriteLine(item);
    } while(item != null);
}
```

可以使用与 C# 8 一起引入的新语法，合并同一类型的多个声明，如下面的示例中所示：

```

string numbers=@"One
Two
Three
Four.";
string letters=@"A
B
C
D.";

using StringReader left = new StringReader(numbers),
      right = new StringReader(letters);
string? item;
do {
    item = left.ReadLine();
    Console.WriteLine(item);
    Console.WriteLine("    ");
    item = right.ReadLine();
    Console.WriteLine(item);
} while(item != null);

```

可以实例化资源对象，然后将变量传递到 `using` 语句，但这不是最佳做法。在这种情况下，控件退出 `using` 块以后，对象保留在作用域中，但是可能没有访问其未托管资源的权限。换而言之，它不再是完全初始化的。如果尝试在 `using` 块外部使用该对象，则可能导致引发异常。因此，最好在 `using` 语句中实例化该对象并将其范围限制在 `using` 块中。

```

string manyLines=@"This is line one
This is line two
Here is line three
The penultimate line is line four
This is the final, fifth line.";

var reader = new StringReader(manyLines);
using (reader)
{
    string? item;
    do {
        item = reader.ReadLine();
        Console.WriteLine(item);
    } while(item != null);
}
// reader is in scope here, but has been disposed

```

有关释放 `IDisposable` 对象的详细信息，请参阅[使用实现 IDisposable 的对象](#)。

C# 语言规范

有关详细信息，请参阅[C# 语言规范](#)中的 `using` 语句。该语言规范是 C# 语法和用法的权威资料。

请参阅

- [C# 参考](#)
- [C# 编程指南](#)
- [C# 关键字](#)
- [using 指令](#)
- [垃圾回收](#)
- [使用实现 IDisposable 的对象](#)
- [IDisposable 接口](#)
- [C# 8.0 中的 using 语句](#)

外部别名 (C# 参考)

2021/5/7 • [Edit Online](#)

有时你可能不得不引用具有相同的完全限定类型的两个程序集。例如，可能需要在同一应用程序中使用某程序集的两个或多个版本。通过使用外部程序集别名，可在别名命名的根级别命名空间内包装每个程序集的命名空间，使其能够在同一文件中使用。

NOTE

外部关键字还被用作方法修饰符，用于声明在非托管代码中编写的方法。

若要引用具有相同的完全限定类型的两个程序集，必须在命令提示符处指定别名，如下所示：

```
/r:GridV1=grid.dll
```

```
/r:GridV2=grid20.dll
```

这将创建外部别名 `GridV1` 和 `GridV2`。若要从程序中使用这些别名，请通过使用 `extern` 关键字引用它们。例如：

```
extern alias GridV1;
```

```
extern alias GridV2;
```

每个外部别名声明都会引入与全局命名空间并行(但不位于其中)的额外根级别命名空间。因此，可以使用其完全限定的名称(根植于相应的命名空间别名中)无歧义地引用每个程序集的类型。

在上一示例中，`GridV1::Grid` 是 `grid.dll` 中的网格控件，`GridV2::Grid` 是 `grid20.dll` 中的网格控件。

使用 Visual Studio

如果你使用的是 Visual Studio，可以按类似方式提供别名。

在 Visual Studio 中向项目添加 `grid.dll` 和 `grid20.dll` 的引用。打开“属性”选项卡，并将别名从“全局”分别更改为“`GridV1`”和“`GridV2`”。

按上述方式使用这些别名

```
extern alias GridV1;  
  
extern alias GridV2;
```

现在，可以通过使用别名指令为命名空间或类型创建别名。有关详细信息，请参阅 [using 指令](#)。

```
using Class1V1 = GridV1::Namespace.Class1;  
  
using Class1V2 = GridV2::Namespace.Class1;
```

C# 语言规范

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

请参阅

- [C# 参考](#)
- [C# 编程指南](#)
- [C# 关键字](#)
- [::运算符](#)
- [引用\(C# 编译器选项\)](#)

new 约束 (C# 参考)

2020/11/2 • [Edit Online](#)

`new` 约束指定泛型类声明中的类型实参必须有公共的无参数构造函数。若要使用 `new` 约束，则该类型不能为抽象类型。

当泛型类创建类型的新实例时，请将 `new` 约束应用于类型参数，如下面的示例所示：

```
class ItemFactory<T> where T : new()
{
    public T GetNewItem()
    {
        return new T();
    }
}
```

当与其他约束一起使用时，`new()` 约束必须最后指定：

```
public class ItemFactory2<T>
    where T : IComparable, new()
{ }
```

有关详细信息，请参阅[类型参数的约束](#)。

`new` 关键字还可用于[创建类型的实例](#)或用作[成员声明修饰符](#)。

C# 语言规范

有关详细信息，请参阅[C# 语言规范](#)中的[类型参数约束](#)部分。

请参阅

- [C# 参考](#)
- [C# 编程指南](#)
- [C# 关键字](#)
- [泛型](#)

where (泛型类型约束) (C# 参考)

2021/5/7 • [Edit Online](#)

泛型定义中的 `where` 子句指定对用作泛型类型、方法、委托或本地函数中类型参数的参数类型的约束。约束可指定接口、基类或要求泛型类型为引用、值或非托管类型。它们声明类型参数必须具备的功能。

例如，可以声明一个泛型类 `AGenericClass`，以使类型参数 `T` 实现 `IComparable<T>` 接口：

```
public class AGenericClass<T> where T : IComparable<T> { }
```

NOTE

有关查询表达式中的 `where` 子句的详细信息，请参阅 [where 子句](#)。

`where` 子句还可包括基类约束。基类约束表明用作该泛型类型的类型参数的类型具有指定的类作为基类（或者是该基类）。该基类约束一经使用，就必须出现在该类型参数的所有其他约束之前。某些类型不允许作为基类约束：`Object`、`Array` 和 `ValueType`。在 C# 7.3 之前，`Enum`、`Delegate` 和 `MulticastDelegate` 也不允许作为基类约束。以下示例显示现可指定为基类的类型：

```
public class UsingEnum<T> where T : System.Enum { }

public class UsingDelegate<T> where T : System.Delegate { }

public class Multicaster<T> where T : System.MulticastDelegate { }
```

在 C# 8.0 及更高版本中的可为 null 上下文中，强制执行基类类型的为 null 性。如果基类不可为 null（例如 `Base`），则类型参数必须不可为 null。如果基类可为 null（例如 `Base?`），则类型参数可以是可为 null 或不可为 null 的引用类型。当基类不可为 null 时，如果类型参数是可为 null 的引用类型，编译器将发出警告。

`where` 子句可指定类型为 `class` 或 `struct`。`struct` 约束不再需要指定 `System.ValueType` 的基类约束。

`System.ValueType` 类型可能不用作基类约束。以下示例显示 `class` 和 `struct` 约束：

```
class MyClass<T, U>
    where T : class
    where U : struct
{ }
```

在 C# 8.0 及更高版本中的可为 null 上下文中，`class` 约束要求类型是不可为 null 的引用类型。若要允许可为 null 的引用类型，请使用 `class?` 约束，该约束允许可为 null 和不可为 null 的引用类型。

`where` 子句可能包含 `notnull` 约束。`notnull` 约束将类型参数限制为不可为 null 的类型。该类型可以是值类型，也可以是不可为 null 的引用类型。对于在 `nullable enable` 上下文中编译的代码，从 C# 8.0 开始可以使用 `notnull` 约束。与其他约束不同，如果类型参数违反 `notnull` 约束，编译器会生成警告而不是错误。警告仅在 `nullable enable` 上下文中生成。

IMPORTANT

包含 `notnull` 约束的泛型声明可以在可为 null 的不明显上下文中使用，但编译器不会强制执行约束。

```
#nullable enable
class NotNullContainer<T>
    where T : notnull
{
}
#nullable restore
```

`where` 子句还可包括 `unmanaged` 约束。`unmanaged` 约束将类型参数限制为名为“[非托管类型](#)”的类型。
`unmanaged` 约束使得在 C# 中编写低级别的互操作代码变得更容易。此约束支持跨所有非托管类型的可重用例程。`unmanaged` 约束不能与 `class` 或 `struct` 约束结合使用。`unmanaged` 约束强制该类型必须为 `struct`：

```
class UnManagedWrapper<T>
    where T : unmanaged
{ }
```

`where` 子句也可能包括构造函数约束 `new()`。该约束使得能够使用 `new` 运算符创建类型参数的实例。[new\(\) 约束](#)可以让编译器知道：提供的任何类型参数都必须具有可访问的无参数构造函数。例如：

```
public class MyGenericClass<T> where T : IComparable<T>, new()
{
    // The following line is not possible without new() constraint:
    T item = new T();
}
```

`new()` 约束出现在 `where` 子句的最后。`new()` 约束不能与 `struct` 或 `unmanaged` 约束结合使用。所有满足这些约束的类型必须具有可访问的无参数构造函数，这使得 `new()` 约束冗余。

对于多个类型参数，每个类型参数都使用一个 `where` 子句，例如：

```
public interface IMyInterface { }

namespace CodeExample
{
    class Dictionary<TKey, TValue>
        where TKey : IComparable<TKey>
        where TValue : IMyInterface
    {
        public void Add(TKey key, TValue val) { }
    }
}
```

还可将约束附加到泛型方法的类型参数，如以下示例所示：

```
public void MyMethod<T>(T t) where T : IMyInterface { }
```

请注意，对于委托和方法两者来说，描述类型参数约束的语法是一样的：

```
delegate T MyDelegate<T>() where T : new();
```

有关泛型委托的信息，请参阅[泛型委托](#)。

有关约束的语法和用法的详细信息，请参阅[类型参数的约束](#)。

C# 语言规范

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

请参阅

- [C# 参考](#)
- [C# 编程指南](#)
- [泛型介绍](#)
- [new 约束](#)
- [类型参数的约束](#)

base (C# 参考)

2020/3/18 • [Edit Online](#)

`base` 关键字用于从派生类中访问基类的成员：

- 调用基类上已被其他方法重写的方法。
- 指定创建派生类实例时应调用的基类构造函数。

仅允许基类访问在构造函数、实例方法或实例属性访问器中进行。

从静态方法中使用 `base` 关键字是错误的。

所访问的基类是类声明中指定的基类。例如，如果指定 `class ClassB : ClassA`，则从 ClassB 访问 ClassA 的成员，而不考虑 ClassA 的基类。

示例

在本例中，基类 `Person` 和派生类 `Employee` 都有一个名为 `GetInfo` 的方法。通过使用 `base` 关键字，可以从派生类中调用基类的 `GetInfo` 方法。

```
public class Person
{
    protected string ssn = "444-55-6666";
    protected string name = "John L. Malgraine";

    public virtual void GetInfo()
    {
        Console.WriteLine("Name: {0}", name);
        Console.WriteLine("SSN: {0}", ssn);
    }
}

class Employee : Person
{
    public string id = "ABC567EFG";
    public override void GetInfo()
    {
        // Calling the base class GetInfo method:
        base.GetInfo();
        Console.WriteLine("Employee ID: {0}", id);
    }
}

class TestClass
{
    static void Main()
    {
        Employee E = new Employee();
        E.GetInfo();
    }
}

/*
Output
Name: John L. Malgraine
SSN: 444-55-6666
Employee ID: ABC567EFG
*/
```

有关其他示例，请参阅 [new](#)、[virtual](#) 和 [override](#)。

示例

本示例显示如何指定在创建派生类实例时调用的基类构造函数。

```
public class BaseClass
{
    int num;

    public BaseClass()
    {
        Console.WriteLine("in BaseClass()");
    }

    public BaseClass(int i)
    {
        num = i;
        Console.WriteLine("in BaseClass(int i)");
    }

    public int GetNum()
    {
        return num;
    }
}

public class DerivedClass : BaseClass
{
    // This constructor will call BaseClass.BaseClass()
    public DerivedClass() : base()
    {
    }

    // This constructor will call BaseClass.BaseClass(int i)
    public DerivedClass(int i) : base(i)
    {
    }

    static void Main()
    {
        DerivedClass md = new DerivedClass();
        DerivedClass md1 = new DerivedClass(1);
    }
}
/*
Output:
in BaseClass()
in BaseClass(int i)
*/
```

C# 语言规范

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

另请参阅

- [C# 参考](#)
- [C# 编程指南](#)
- [C# 关键字](#)
- [this](#)

this (C# 参考)

2020/3/18 • [Edit Online](#)

`this` 关键字指代类的当前实例，还可用作扩展方法的第一个参数的修饰符。

NOTE

本文介绍 `this` 在类实例中的用法。若要深入了解它在扩展方法中的用法，请参阅[扩展方法](#)。

以下是 `this` 的常见用法：

- 限定类似名称隐藏的成员，例如：

```
public class Employee
{
    private string alias;
    private string name;

    public Employee(string name, string alias)
    {
        // Use this to qualify the members of the class
        // instead of the constructor parameters.
        this.name = name;
        this.alias = alias;
    }
}
```

- 将对象作为参数传递给方法，例如：

```
CalcTax(this);
```

- 声明索引器，例如：

```
public int this[int param]
{
    get { return array[param]; }
    set { array[param] = value; }
}
```

静态成员函数，因为它们存在于类级别且不属于对象，不具有 `this` 指针。在静态方法中引用 `this` 会生成错误。

示例

在此示例中，`this` 用于限定类似名称隐藏的 `Employee` 类成员、`name` 和 `alias`。它还用于将某个对象传递给属于其他类的方法 `CalcTax`。

```

class Employee
{
    private string name;
    private string alias;
    private decimal salary = 3000.00m;

    // Constructor:
    public Employee(string name, string alias)
    {
        // Use this to qualify the fields, name and alias:
        this.name = name;
        this.alias = alias;
    }

    // Printing method:
    public void printEmployee()
    {
        Console.WriteLine("Name: {0}\nAlias: {1}", name, alias);
        // Passing the object to the CalcTax method by using this:
        Console.WriteLine("Taxes: {0:C}", Tax.CalcTax(this));
    }

    public decimal Salary
    {
        get { return salary; }
    }
}

class Tax
{
    public static decimal CalcTax(Employee E)
    {
        return 0.08m * E.Salary;
    }
}

class MainClass
{
    static void Main()
    {
        // Create objects:
        Employee E1 = new Employee("Mingda Pan", "mpan");

        // Display results:
        E1.printEmployee();
    }
}
/*
Output:
Name: Mingda Pan
Alias: mp
Taxes: $240.00
*/

```

C# 语言规范

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

另请参阅

- [C# 参考](#)
- [C# 编程指南](#)

- C# 关键字

- base

- 方法

null (C# 参考)

2020/11/2 • [Edit Online](#)

`null` 关键字是表示不引用任何对象的空引用的文字值。`null` 是引用类型变量的默认值。普通值类型不能为 `NULL`, [可为空的值类型](#)除外。

下面的示例演示 `null` 关键字的一些行为:

```
class Program
{
    class MyClass
    {
        public void MyMethod() { }
    }

    static void Main(string[] args)
    {
        // Set a breakpoint here to see that mc = null.
        // However, the compiler considers it "unassigned."
        // and generates a compiler error if you try to
        // use the variable.
        MyClass mc;

        // Now the variable can be used, but...
        mc = null;

        // ... a method call on a null object raises
        // a run-time NullReferenceException.
        // Uncomment the following line to see for yourself.
        // mc.MyMethod();

        // Now mc has a value.
        mc = new MyClass();

        // You can call its method.
        mc.MyMethod();

        // Set mc to null again. The object it referenced
        // is no longer accessible and can now be garbage-collected.
        mc = null;

        // A null string is not the same as an empty string.
        string s = null;
        string t = String.Empty; // Logically the same as ""

        // Equals applied to any null object returns false.
        bool b = (t.Equals(s));
        Console.WriteLine(b);

        // Equality operator also returns false when one
        // operand is null.
        Console.WriteLine("Empty string {0} null string", s == t ? "equals": "does not equal");

        // Returns true.
        Console.WriteLine("null == null is {0}", null == null);

        // A value type cannot be null
        // int i = null; // Compiler error!

        // Use a nullable value type instead:
        int? i = null;

        // Keep the console window open in debug mode.
        System.Console.WriteLine("Press any key to exit.");
        System.Console.ReadKey();
    }
}
```

C# 语言规范

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

请参阅

- [C# 参考](#)
- [C# 关键字](#)
- [C# 类型的默认值](#)
- [Nothing \(Visual Basic\)](#)

bool (C# 参考)

2020/11/2 • [Edit Online](#)

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

若要使用 `bool` 类型的值执行逻辑运算，请使用[布尔逻辑运算符](#)。`bool` 类型是[比较](#)和[相等](#)运算符的结果类型。

`bool` 表达式可以是 `if`、`do`、`while` 和 `for` 语句中以及[条件运算符](#) `?:` 中的控制条件表达式。

`bool` 类型的默认值为 `false`。

文本

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

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

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

三值布尔逻辑

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

有关可为空的值类型的详细信息，请参阅[可为空的值类型](#)。

转换

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

C# 语言规范

有关详细信息，请参阅[C# 语言规范](#)中的[bool 类型](#)部分。

请参阅

- [C# 参考](#)
- [值类型](#)
- [true 和 false 运算符](#)

default (C# 参考)

2020/11/2 • [Edit Online](#)

`default` 关键字有两种用法：

- 指定 `switch` 语句中的默认标签。
- 作为 `default` 默认运算符或文本生成类型的默认值。

请参阅

- [C# 参考](#)
- [C# 编程指南](#)
- [C# 关键字](#)

上下文关键字 (C# 参考)

2021/5/7 • [Edit Online](#)

上下文关键字用于在代码中提供特定含义，但它不是 C# 中的保留字。本部分介绍下面这些上下文关键字：

III	II
<code>add</code>	定义一个自定义事件访问器，客户端代码订阅事件时会调用该访问器。
<code>and</code>	创建在两个嵌套模式均匹配时匹配的模式。
<code>async</code>	指示修改后的方法、lambda 表达式或匿名方法是异步的。
<code>await</code>	挂起异步方法，直到等待的任务完成。
<code>dynamic</code>	定义一个引用类型，实现发生绕过编译时类型检查的操作。
<code>get</code>	为属性或索引器定义访问器方法。
<code>global</code>	未以其他方式命名的全局命名空间的别名。
<code>init</code>	为属性或索引器定义访问器方法。
<code>nint</code>	定义本机大小的整数数据类型。
<code>not</code>	创建在否定模式不匹配时匹配的模式。
<code>nuint</code>	定义本机大小的无符号整数数据类型。
<code>or</code>	创建在任一嵌套模式匹配时匹配的模式。
<code>partial</code>	在整个同一编译单元内定义分部类、结构和接口。
<code>record</code>	用于定义记录类型。
<code>remove</code>	定义一个自定义事件访问器，客户端代码取消订阅事件时会调用该访问器。
<code>set</code>	为属性或索引器定义访问器方法。
<code>value</code>	用于设置访问器和添加或删除事件处理程序。
<code>var</code>	使编译器能够确定在方法作用域中声明的变量类型。
<code>when</code>	指定 <code>catch</code> 块的筛选条件或 <code>switch</code> 语句的 <code>case</code> 标签。
<code>where</code>	将约束添加到泛型声明。(另请参阅 where)。

¶

¶

`yield`

在迭代器块中使用，用于向枚举数对象返回值或用于表示迭代结束。

C# 3.0 中引入的所有查询关键字也都是上下文相关的。有关详细信息，请参阅[查询关键字 \(LINQ\)](#)。

请参阅

- [C# 参考](#)
- [C# 关键字](#)
- [C# 运算符和表达式](#)

add (C# 参考)

2020/11/2 • [Edit Online](#)

`add` 上下文关键字用于定义一个在客户端代码订阅你的[事件](#)时调用的自定义事件访问器。如果提供自定义
`add` 访问器，还必须提供 `remove` 访问器。

示例

如下示例显示一个具有自定义 `add` 和 `remove` 访问器的事件。有关完整示例，请参阅[如何实现接口事件](#)。

```
class Events : IDrawingObject
{
    event EventHandler PreDrawEvent;

    event EventHandler IDrawingObject.OnDraw
    {
        add => PreDrawEvent += value;
        remove => PreDrawEvent -= value;
    }
}
```

通常不需要提供自己的自定义事件访问器。大多数情况下，使用声明事件时由编译器自动生成的访问器就够了。

请参阅

- [事件](#)

get (C# 参考)

2021/3/5 • [Edit Online](#)

`get` 关键字在属性或索引器中定义访问器方法，它将返回属性值或索引器元素。有关详细信息，请参阅[属性、自动实现的属性和索引器](#)。

下面的示例为名为 `Seconds` 的属性同时定义 `get` 和 `set` 访问器。它使用名为 `_seconds` 的私有字段备份属性值。

```
class TimePeriod
{
    private double _seconds;

    public double Seconds
    {
        get { return _seconds; }
        set { _seconds = value; }
    }
}
```

通常，`get` 访问器包含返回一个值的单个语句，如前面的示例所示。从 C# 7.0 开始，可以将 `get` 访问器作为 expression-bodied 成员实现。以下示例将 `get` 和 `set` 访问器都作为 expression-bodied 成员实现。

```
class TimePeriod
{
    private double _seconds;

    public double Seconds
    {
        get => _seconds;
        set => _seconds = value;
    }
}
```

对于属性的 `get` 和 `set` 访问器不执行除设置或检索私有支持字段中的值以外的任何其他操作的简单情况，可以利用 C# 编译器对自动实现的属性的支持。以下示例将 `Hours` 作为自动实现的属性来实现。

```
class TimePeriod2
{
    public double Hours { get; set; }
}
```

C# 语言规范

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

请参阅

- [C# 参考](#)
- [C# 编程指南](#)
- [C# 关键字](#)

- 属性

init (C# 参考)

2021/5/7 • [Edit Online](#)

在 C# 9 及更高版本中，`init` 关键字在属性或索引器中定义访问器方法。init-only 资源库仅在对象构造期间为属性或索引器元素赋值。有关详细信息和示例，请参阅[属性](#)、[自动实现的属性](#)和[索引器](#)。

以下示例为名为 `Seconds` 的属性同时定义 `get` 和 `init` 访问器。它使用名为 `_seconds` 的私有字段备份属性值。

```
class InitExample
{
    private double _seconds;

    public double Seconds
    {
        get { return _seconds; }
        init { _seconds = value; }
    }
}
```

通常，`init` 访问器包含分配一个值的单个语句，如前面的示例所示。可以将 `init` 访问器作为表达式主体成员实现。下面的示例将 `get` 和 `init` 访问器都作为表达式主体成员实现。

```
class InitExampleExpressionBodied
{
    private double _seconds;

    public double Seconds
    {
        get => _seconds;
        init => _seconds = value;
    }
}
```

对于属性的 `get` 和 `init` 访问器不执行除设置或检索私有支持字段中的值以外的任何其他操作的简单情况，可以利用 C# 编译器对自动实现的属性的支持。以下示例将 `Hours` 作为自动实现的属性来实现。

```
class InitExampleAutoProperty
{
    public double Hours { get; init; }
}
```

C# 语言规范

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

请参阅

- [C# 参考](#)
- [C# 编程指南](#)
- [C# 关键字](#)
- [属性](#)

分部类型 (C# 参考)

2020/11/2 • [Edit Online](#)

通过分部类型可以定义要拆分到多个文件中的类、结构、接口或记录。

在 File1.cs 中：

```
namespace PC
{
    partial class A
    {
        int num = 0;
        void MethodA() { }
        partial void MethodC();
    }
}
```

在 File2.cs 中，声明：

```
namespace PC
{
    partial class A
    {
        void MethodB() { }
        partial void MethodC() { }
    }
}
```

备注

在处理大型项目或自动生成的代码(如 Windows 窗体设计器提供的代码)时，在多个文件间拆分类、结构或接口类型可能会非常有用。分部类型可以包含[分部方法](#)。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。

请参阅

- [C# 参考](#)
- [C# 编程指南](#)
- [修饰符](#)
- [泛型介绍](#)

分部方法 (C# 参考)

2021/5/7 • [Edit Online](#)

分部方法在分部类型的一部分中定义了签名，并在该类型的另一部分中定义了实现。通过分部方法，类设计器可提供与事件处理程序类似的方法挂钩，以便开发者决定是否实现。如果开发者不提供实现，则编译器在编译时删除签名。以下条件适用于分部方法：

- 声明必须以上下文关键字 `partial` 开头。
- 分部类型各部分中的签名必须匹配。

在以下情况下，不需要使用分部方法即可实现：

- 没有任何可访问性修饰符(包括默认的 `专用`)。
- 返回 `void`。
- 没有任何 `输出` 参数。
- 没有以下任何修饰符：`virtual`、`override`、`sealed`、`new` 或 `extern`。

任何不符合所有这些限制的方法(例如 `public virtual partial void` 方法)都必须提供实现。

下列示例显示在分部类的两个部分中定义的分部方法：

```
namespace PM
{
    partial class A
    {
        partial void OnSomethingHappened(string s);
    }

    // This part can be in a separate file.
    partial class A
    {
        // Comment out this method and the program
        // will still compile.
        partial void OnSomethingHappened(String s)
        {
            Console.WriteLine("Something happened: {0}", s);
        }
    }
}
```

分部方法还可用于与源生成器结合。例如，可使用以下模式定义 regex：

```
[RegexGenerated("(dog|cat|fish)")]
partial bool IsPetMatch(string input);
```

有关详细信息，请参阅[分部类和方法](#)。

请参阅

- [C# 参考](#)
- [分部类型](#)

remove (C# 参考)

2020/11/2 • [Edit Online](#)

`remove` 上下文关键字用于定义自定义事件访问器，当客户端代码订阅你的[事件](#)时将调用该访问器。如果提供自定义 `remove` 访问器，还必须提供 `add` 访问器。

示例

以下示例显示具有自定义 `add` 和 `remove` 访问器的事件。有关完整示例，请参阅[如何实现接口事件](#)。

```
class Events : IDrawingObject
{
    event EventHandler PreDrawEvent;

    event EventHandler IDrawingObject.OnDraw
    {
        add => PreDrawEvent += value;
        remove => PreDrawEvent -= value;
    }
}
```

通常不需要提供自己的自定义事件访问器。大多数情况下，使用声明事件时由编译器自动生成的访问器就够了。

请参阅

- [事件](#)

set (C# 参考)

2020/11/2 • [Edit Online](#)

`set` 关键字在属性或索引器中定义访问器，它会向属性或索引器元素分配值。有关详细信息和示例，请参阅[属性、自动实现的属性和索引器](#)。

下面的示例为名为 `Seconds` 的属性同时定义 `get` 和 `set` 访问器。它使用名为 `_seconds` 的私有字段备份属性值。

```
class TimePeriod
{
    private double _seconds;

    public double Seconds
    {
        get { return _seconds; }
        set { _seconds = value; }
    }
}
```

通常，`set` 访问器包含分配一个值的单个语句，如前面的示例所示。从 C# 7.0 开始，可以将 `set` 访问器作为 expression-bodied 成员实现。下面的示例将 `get` 和 `set` 访问器都作为表达式主体成员实现。

```
class TimePeriod
{
    private double _seconds;

    public double Seconds
    {
        get => _seconds;
        set => _seconds = value;
    }
}
```

对于属性的 `get` 和 `set` 访问器不执行除设置或检索私有支持字段中的值以外的任何其他操作的简单情况，可以利用 C# 编译器对自动实现的属性的支持。以下示例将 `Hours` 作为自动实现的属性来实现。

```
class TimePeriod2
{
    public double Hours { get; set; }
}
```

C# 语言规范

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

请参阅

- [C# 参考](#)
- [C# 编程指南](#)
- [C# 关键字](#)

- 属性

when (C# 参考)

2020/11/2 • [Edit Online](#)

可以使用上下文关键字 `when` 在以下上下文中指定筛选条件：

- 在 `try/catch` 或 `try/catch/finally` 块的 `catch` 语句中。
- 在 `switch` 语句的 `case` 标签中。
- 在 `switch 表达式` 中。

`catch` 语句中的 `when`

从 C# 6 开始，`when` 可用于 `catch` 语句中，以指定为执行特定异常处理程序而必须为 `true` 的条件。语法为：

```
catch (ExceptionType [e]) when (expr)
```

其中，`expr` 是一个表达式，其计算结果为布尔值。如果该表达式返回 `true`，则执行异常处理程序；如果返回 `false`，则不执行。

以下示例使用 `when` 关键字有条件地执行 `HttpRequestException` 的处理程序，具体取决于异常消息的文本内容。

```
using System;
using System.Net.Http;
using System.Threading.Tasks;

class Program
{
    static void Main()
    {
        Console.WriteLine(MakeRequest().Result);
    }

    public static async Task<string> MakeRequest()
    {
        var client = new HttpClient();
        var streamTask = client.GetStringAsync("https://localhost:10000");
        try
        {
            var responseText = await streamTask;
            return responseText;
        }
        catch (HttpRequestException e) when (e.Message.Contains("301"))
        {
            return "Site Moved";
        }
        catch (HttpRequestException e) when (e.Message.Contains("404"))
        {
            return "Page Not Found";
        }
        catch (HttpRequestException e)
        {
            return e.Message;
        }
    }
}
```

switch 语句中的 when

从 C# 7.0 开始，`case` 标签无需是互斥的，且 `case` 标签在 `switch` 语句中的显示顺序可以决定要执行的 `switch` 块。`when` 关键字可指定一个筛选条件，该条件使得仅当筛选条件也为 `true` 时，与其相关联的 `case` 标签才为 `true`。语法为：

```
case (expr) when (when-condition):
```

其中，`expr` 是与匹配表达式进行比较的常量模式或类型模式，而 `when-condition` 是任意布尔表达式。

以下示例使用 `when` 关键字针对具有零范围的 `Shape` 对象进行测试，同时针对各种范围大于零的 `Shape` 对象进行测试。

```
using System;

public abstract class Shape
{
    public abstract double Area { get; }
    public abstract double Circumference { get; }
}

public class Rectangle : Shape
{
    public Rectangle(double length, double width)
    {
        Length = length;
        Width = width;
    }

    public double Length { get; set; }
    public double Width { get; set; }

    public override double Area
    {
        get { return Math.Round(Length * Width,2); }
    }

    public override double Circumference
    {
        get { return (Length + Width) * 2; }
    }
}

public class Square : Rectangle
{
    public Square(double side) : base(side, side)
    {
        Side = side;
    }

    public double Side { get; set; }
}

public class Example
{
    public static void Main()
    {
        Shape sh = null;
        Shape[] shapes = { new Square(10), new Rectangle(5, 7),
                           new Rectangle(10, 10), sh, new Square(0) };
        foreach (var shape in shapes)
            ShowShapeInfo(shape);
    }
}
```

```

private static void ShowShapeInfo(Object obj)
{
    switch (obj)
    {
        case Shape shape when shape.Area == 0:
            Console.WriteLine($"The shape: {shape.GetType().Name} with no dimensions");
            break;
        case Square sq when sq.Area > 0:
            Console.WriteLine("Information about the square:");
            Console.WriteLine($"    Length of a side: {sq.Side}");
            Console.WriteLine($"    Area: {sq.Area}");
            break;
        case Rectangle r when r.Area > 0:
            Console.WriteLine("Information about the rectangle:");
            Console.WriteLine($"    Dimensions: {r.Length} x {r.Width}");
            Console.WriteLine($"    Area: {r.Area}");
            break;
        case Shape shape:
            Console.WriteLine($"A {shape.GetType().Name} shape");
            break;
        case null:
            Console.WriteLine($"The {nameof(obj)} variable is uninitialized.");
            break;
        default:
            Console.WriteLine($"The {nameof(obj)} variable does not represent a Shape.");
            break;
    }
}

// The example displays the following output:
//     Information about the square:
//         Length of a side: 10
//         Area: 100
//     Information about the rectangle:
//         Dimensions: 5 x 7
//         Area: 35
//     Information about the rectangle:
//         Dimensions: 10 x 10
//         Area: 100
//     The obj variable is uninitialized.
//     The shape: Square with no dimensions

```

请参阅

- [switch 语句](#)
- [try/catch 语句](#)
- [try/catch/finally 语句](#)

value (C# 参考)

2020/11/2 • [Edit Online](#)

上下文关键字 `value` 在属性和索引器声明的 `set` 访问器中使用。此关键字类似于方法的输入参数。关键字 `value` 引用客户端代码尝试分配给属性或索引器的值。在以下示例中，`MyDerivedClass` 有一个名为 `Name` 的属性，该属性使用 `value` 参数向支持字段 `name` 分配新字符串。从客户端代码的角度来看，该操作写作一个简单的赋值语句。

```
class MyBaseClass
{
    // virtual auto-implemented property. Overrides can only
    // provide specialized behavior if they implement get and set accessors.
    public virtual string Name { get; set; }

    // ordinary virtual property with backing field
    private int num;
    public virtual int Number
    {
        get { return num; }
        set { num = value; }
    }
}

class MyDerivedClass : MyBaseClass
{
    private string name;

    // Override auto-implemented property with ordinary property
    // to provide specialized accessor behavior.
    public override string Name
    {
        get
        {
            return name;
        }
        set
        {
            if (!string.IsNullOrEmpty(value))
            {
                name = value;
            }
            else
            {
                name = "Unknown";
            }
        }
    }
}
```

有关详细信息，请参阅[属性和索引器](#)这两篇文章。

C# 语言规范

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

请参阅

- [C# 参考](#)

- [C# 编程指南](#)
- [C# 关键字](#)

yield (C# 参考)

2021/3/5 • [Edit Online](#)

如果你在语句中使用 `yield` [上下文关键字](#)，则意味着它在其中出现的方法、运算符或 `get` 访问器是迭代器。通过使用 `yield` 定义迭代器，可在实现自定义集合类型的 `IEnumerable<T>` 和 `IEnumerable` 模式时无需其他显式类(保留枚举状态的类，有关示例，请参阅 [IEnumerator](#))。

下面的示例演示了 `yield` 语句的两种形式。

```
yield return <expression>;
yield break;
```

备注

使用 `yield return` 语句可一次返回一个元素。

可通过使用 `foreach` 语句或 LINQ 查询来使用从迭代器方法返回的序列。`foreach` 循环的每次迭代都会调用迭代器方法。迭代器方法运行到 `yield return` 语句时，会返回一个 `expression`，并保留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。

可以使用 `yield break` 语句来终止迭代。

有关迭代器的详细信息，请参阅[迭代器](#)。

迭代器方法和 get 访问器

迭代器的声明必须满足以下要求：

- 返回类型必须为 `IEnumerable`、`IEnumerable<T>`、`IEnumerator` 或 `IEnumerator<T>`。
- 声明不能有任何 `in`、`ref` 或 `out` 参数。

返回 `yield` 或 `IEnumerable` 的迭代器的 `IEnumerator` 类型为 `object`。如果迭代器返回 `IEnumerable<T>` 或 `IEnumerable<T>`，则必须将 `yield return` 语句中的表达式类型隐式转换为泛型类型参数。

以下情形中不能包含 `yield return` 或 `yield break` 语句：

- [Lambda 表达式和匿名方法](#)。
- 包含不安全的块的方法。有关详细信息，请参阅 [unsafe](#)。

异常处理

不能将 `yield return` 语句置于 try-catch 块中。可将 `yield return` 语句置于 try-finally 语句的 try 块中。

可将 `yield break` 语句置于 try 块或 catch 块中，但不能将其置于 finally 块中。

如果 `foreach` 主体(在迭代器方法之外)引发异常，则将执行迭代器方法中的 `finally` 块。

技术实现

以下代码从迭代器方法返回 `IEnumerable<string>`，然后遍历其元素。

```
IEnumerable<string> elements = MyIteratorMethod();
foreach (string element in elements)
{
    ...
}
```

调用 `MyIteratorMethod` 并不执行该方法的主体。相反，该调用会将 `IEnumerable<string>` 返回到 `elements` 变量中。

在 `foreach` 循环迭代时，将为 `MoveNext` 调用 `elements` 方法。此调用将执行 `MyIteratorMethod` 的主体，直至到达下一个 `yield return` 语句。`yield return` 语句返回的表达式不仅决定了循环体使用的 `element` 变量值，还决定了 `elements` 的 `Current` 属性(它是 `IEnumerable<string>`)。

在 `foreach` 循环的每个后续迭代中，迭代器主体的执行将从它暂停的位置继续，直至到达 `yield return` 语句后才会停止。在到达迭代器方法的结尾或 `foreach` 语句时，`yield break` 循环便已完成。

示例

下面的示例包含一个位于 `yield return` 循环内的 `for` 语句。`Main` 方法中的 `foreach` 语句体的每次迭代都会创建对 `Power` 迭代器函数的调用。对迭代器函数的每个调用将继续到 `yield return` 语句的下一次执行(在 `for` 循环的下一次迭代期间发生)。

迭代器方法的返回类型是 `IEnumerable`(一种迭代器接口类型)。当调用迭代器方法时，它将返回一个包含数字幂的可枚举对象。

```
public class PowersOf2
{
    static void Main()
    {
        // Display powers of 2 up to the exponent of 8:
        foreach (int i in Power(2, 8))
        {
            Console.Write("{0} ", i);
        }
    }

    public static System.Collections.Generic.IEnumerable<int> Power(int number, int exponent)
    {
        int result = 1;

        for (int i = 0; i < exponent; i++)
        {
            result = result * number;
            yield return result;
        }
    }

    // Output: 2 4 8 16 32 64 128 256
}
```

示例

下面的示例演示一个作为迭代器的 `get` 访问器。在该示例中，每个 `yield return` 语句返回一个用户定义的类的实例。

```
public static class GalaxyClass
{
    public static void ShowGalaxies()
    {
        var theGalaxies = new Galaxies();
        foreach (Galaxy theGalaxy in theGalaxies.NextGalaxy)
        {
            Debug.WriteLine(theGalaxy.Name + " " + theGalaxy.MegaLightYears.ToString());
        }
    }

    public class Galaxies
    {

        public System.Collections.Generic.IEnumerable<Galaxy> NextGalaxy
        {
            get
            {
                yield return new Galaxy { Name = "Tadpole", MegaLightYears = 400 };
                yield return new Galaxy { Name = "Pinwheel", MegaLightYears = 25 };
                yield return new Galaxy { Name = "Milky Way", MegaLightYears = 0 };
                yield return new Galaxy { Name = "Andromeda", MegaLightYears = 3 };
            }
        }
    }

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

C# 语言规范

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

请参阅

- [C# 参考](#)
- [C# 编程指南](#)
- [foreach, in](#)
- [迭代器](#)

查询关键字 (C# 参考)

2020/11/2 • [Edit Online](#)

本部分包含在查询表达式中使用的上下文关键字。

本节内容

从	到
<code>from</code>	指定数据源和范围变量(类似于迭代变量)。
<code>where</code>	基于由逻辑 AND 和 OR 运算符 (<code>&&</code> 或 <code> </code>) 分隔的一个或多个布尔表达式筛选源元素。
<code>select</code>	指定执行查询时, 所返回序列中元素的类型和形状。
<code>group</code>	根据指定的密钥值对查询结果分组。
<code>into</code>	提供可作为对 <code>join</code> 、 <code>group</code> 或 <code>select</code> 子句结果引用的标识符。
<code>orderby</code>	根据元素类型的默认比较器对查询结果进行升序或降序排序。
<code>join</code>	基于两个指定匹配条件间的相等比较而联接两个数据源。
<code>let</code>	引入范围变量, 在查询表达式中存储子表达式结果。
<code>in</code>	<code>join</code> 子句中的上下文关键字。
<code>on</code>	<code>join</code> 子句中的上下文关键字。
<code>equals</code>	<code>join</code> 子句中的上下文关键字。
<code>by</code>	<code>group</code> 子句中的上下文关键字。
<code>ascending</code>	<code>orderby</code> 子句中的上下文关键字。
<code>descending</code>	<code>orderby</code> 子句中的上下文关键字。

请参阅

- [C# 关键字](#)
- [LINQ\(语言集成查询\)](#)
- [C# 中的 LINQ](#)

from 子句 (C# 参考)

2021/3/5 • [Edit Online](#)

查询表达式必须以 `from` 子句开头。此外，查询表达式可包含也以 `from` 子句开头的子查询。`from` 子句指定下列各项：

- 将在其上运行查询或子查询的数据源。
- 表示源序列中每个元素的本地范围变量。

范围变量和数据源已强类型化。`from` 子句中引用的数据源必须具有 `IEnumerable`、`IEnumerable<T>` 类型之一，或 `IQueryable<T>` 等派生类型。

在下面的示例中，`numbers` 是数据源，`num` 是范围变量。请注意，这两个变量都已强类型化，即使使用 `var` 关键字也是如此。

```
class LowNums
{
    static void Main()
    {
        // A simple data source.
        int[] numbers = { 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 };

        // Create the query.
        // lowNums is an IEnumerable<int>
        var lowNums = from num in numbers
                      where num < 5
                      select num;

        // Execute the query.
        foreach (int i in lowNums)
        {
            Console.Write(i + " ");
        }
    }
}
// Output: 4 1 3 2 0
```

范围变量

数据源实现 `IEnumerable<T>` 时，编译器推断范围变量的类型。例如，如果源具有 `IEnumerable<Customer>` 类型，则范围变量会被推断为 `Customer`。仅在以下情况下必须显式指定类型：源是 `ArrayList` 等非泛型 `IEnumerable` 类型时。有关详细信息，请参阅 [如何使用 LINQ 查询 ArrayList](#)。

在以上示例中，`num` 推断为 `int` 类型。由于强类型化了范围变量，所以可以在其上调用方法，或将其用于其他操作中。例如，不再编写 `select num`，而编写 `select num.ToString()`，使查询表达式返回字符串序列，而不是整数序列。或者可以编写 `select num + 10`，使表达式返回序列 14、11、13、12、10。有关详细信息，请参阅 [select 子句](#)。

范围变量与 `foreach` 语句中的迭代变量类似，一项非常重要的区别除外：范围变量从不实际存储来自源的数据。它只是一种语法上的便利，让查询能够描述执行查询时将发生的情况。有关详细信息，请参阅 [LINQ 查询简介 \(C#\)](#)。

复合 from 子句

在某些情况下，源序列中的每个元素可能本身就是一个序列，或者包含一个序列。例如，数据源可能是 `IEnumerable<Student>`，其中序列中的每个学生对象都包含测试分数的列表。要访问每个 `Student` 元素的内部列表，可以使用复合 `from` 子句。这种方法类似于使用嵌套的 `foreach` 语句。可以向任一 `from` 子句添加 `where` 或 `orderby` 子句筛选结果。下面的示例演示 `Student` 对象的序列，其中每个对象都包含一个整数内部 `List`，表示测验分数。访问内部列表可使用复合 `from` 子句。如有必要，可以在这两个 `from` 子句间插入子句。

```
class CompoundFrom
{
    // The element type of the data source.
    public class Student
    {
        public string LastName { get; set; }
        public List<int> Scores {get; set;}
    }

    static void Main()
    {

        // Use a collection initializer to create the data source. Note that
        // each element in the list contains an inner sequence of scores.
        List<Student> students = new List<Student>
        {
            new Student {LastName="Omelchenko", Scores= new List<int> {97, 72, 81, 60}},
            new Student {LastName="O'Donnell", Scores= new List<int> {75, 84, 91, 39}},
            new Student {LastName="Mortensen", Scores= new List<int> {88, 94, 65, 85}},
            new Student {LastName="Garcia", Scores= new List<int> {97, 89, 85, 82}},
            new Student {LastName="Beebe", Scores= new List<int> {35, 72, 91, 70}}
        };

        // Use a compound from to access the inner sequence within each element.
        // Note the similarity to a nested foreach statement.
        var scoreQuery = from student in students
                          from score in student.Scores
                          where score > 90
                          select new { Last = student.LastName, score };

        // Execute the queries.
        Console.WriteLine("scoreQuery:");
        // Rest the mouse pointer on scoreQuery in the following line to
        // see its type. The type is IEnumerable<'a>, where 'a is an
        // anonymous type defined as new {string Last, int score}. That is,
        // each instance of this anonymous type has two members, a string
        // (Last) and an int (score).
        foreach (var student in scoreQuery)
        {
            Console.WriteLine("{0} Score: {1}", student.Last, student.score);
        }

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}

/*
scoreQuery:
Omelchenko Score: 97
O'Donnell Score: 91
Mortensen Score: 94
Garcia Score: 97
Beebe Score: 91
*/
```

使用多个 from 子句执行联接

复合 `from` 子句用于访问单个数据源中的内部集合。但是，查询也可以包含多个 `from` 子句，这些子句从独立的数据源生成补充查询。通过此方法，可以执行使用 [join 子句](#) 无法实现的某类联接操作。

下面的示例演示两个 `from` 子句如何用于形成两个数据源的完全交叉联接。

```

class CompoundFrom2
{
    static void Main()
    {
        char[] upperCase = { 'A', 'B', 'C' };
        char[] lowerCase = { 'x', 'y', 'z' };

        // The type of joinQuery1 is IEnumerable<'a>, where 'a
        // indicates an anonymous type. This anonymous type has two
        // members, upper and lower, both of type char.
        var joinQuery1 =
            from upper in upperCase
            from lower in lowerCase
            select new { upper, lower };

        // The type of joinQuery2 is IEnumerable<'a>, where 'a
        // indicates an anonymous type. This anonymous type has two
        // members, lower and upper, both of type char.
        var joinQuery2 =
            from lower in lowerCase
            where lower != 'x'
            from upper in upperCase
            select new { lower, upper };

        // Execute the queries.
        Console.WriteLine("Cross join:");
        // Rest the mouse pointer on joinQuery1 to verify its type.
        foreach (var pair in joinQuery1)
        {
            Console.WriteLine("{0} is matched to {1}", pair.upper, pair.lower);
        }

        Console.WriteLine("Filtered non-equijoin:");
        // Rest the mouse pointer over joinQuery2 to verify its type.
        foreach (var pair in joinQuery2)
        {
            Console.WriteLine("{0} is matched to {1}", pair.lower, pair.upper);
        }

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
/* Output:
   Cross join:
   A is matched to x
   A is matched to y
   A is matched to z
   B is matched to x
   B is matched to y
   B is matched to z
   C is matched to x
   C is matched to y
   C is matched to z
   Filtered non-equijoin:
   y is matched to A
   y is matched to B
   y is matched to C
   z is matched to A
   z is matched to B
   z is matched to C
*/

```

若要详细了解使用多个 `from` 子句的联接操作，请参阅[执行左外部联结](#)。

另请参阅

- [查询关键字 \(LINQ\)](#)
- [语言集成查询 \(LINQ\)](#)

where 子句 (C# 参考)

2021/3/5 • • [Edit Online](#)

`where` 子句用在查询表达式中，用于指定将在查询表达式中返回数据源中的哪些元素。它将一个布尔条件(谓词)应用于每个源元素(由范围变量引用)，并返回满足指定条件的元素。一个查询表达式可以包含多个 `where` 子句，一个子句可以包含多个谓词子表达式。

示例

在下面的示例中，`where` 子句筛选出除小于五的所有数字。如果删除 `where` 子句，则会返回数据源中的所有数字。表达式 `num < 5` 是应用于每个元素的谓词。

```
class WhereSample
{
    static void Main()
    {
        // Simple data source. Arrays support IEnumerable<T>.
        int[] numbers = { 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 };

        // Simple query with one predicate in where clause.
        var queryLowNums =
            from num in numbers
            where num < 5
            select num;

        // Execute the query.
        foreach (var s in queryLowNums)
        {
            Console.WriteLine(s.ToString() + " ");
        }
    }
}
//Output: 4 1 3 2 0
```

示例

在单个 `where` 子句中，可以使用 `&&` 和 `||` 运算符根据需要指定任意多个谓词。在下面的示例中，查询将指定两个谓词，以便只选择小于五的偶数。

```
class WhereSample2
{
    static void Main()
    {
        // Data source.
        int[] numbers = { 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 };

        // Create the query with two predicates in where clause.
        var queryLowNums2 =
            from num in numbers
            where num < 5 && num % 2 == 0
            select num;

        // Execute the query
        foreach (var s in queryLowNums2)
        {
            Console.WriteLine(s.ToString() + " ");
        }
        Console.WriteLine();

        // Create the query with two where clause.
        var queryLowNums3 =
            from num in numbers
            where num < 5
            where num % 2 == 0
            select num;

        // Execute the query
        foreach (var s in queryLowNums3)
        {
            Console.WriteLine(s.ToString() + " ");
        }
    }
    // Output:
    // 4 2 0
    // 4 2 0
```

示例

一个 `where` 子句可以包含一个或多个返回布尔值的方法。在下面的示例中，`where` 子句使用一种方法来确定范围变量的当前值是偶数还是奇数。

```

class WhereSample3
{
    static void Main()
    {
        // Data source
        int[] numbers = { 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 };

        // Create the query with a method call in the where clause.
        // Note: This won't work in LINQ to SQL unless you have a
        // stored procedure that is mapped to a method by this name.
        var queryEvenNums =
            from num in numbers
            where IsEven(num)
            select num;

        // Execute the query.
        foreach (var s in queryEvenNums)
        {
            Console.WriteLine(s.ToString() + " ");
        }
    }

    // Method may be instance method or static method.
    static bool IsEven(int i)
    {
        return i % 2 == 0;
    }
}

//Output: 4 8 6 2 0

```

备注

`where` 子句是一种筛选机制。除了不能是第一个或最后一个子句外，它几乎可以放在查询表达式中的任何位置。`where` 子句可以出现在 `group` 子句的前面或后面，具体取决于时必须在对源元素进行分组之前还是之后来筛选源元素。

如果指定的谓词对于数据源中的元素无效，则会发生编译时错误。这是 LINQ 提供的强类型检查的一个优点。

在编译时，`where` 关键字将转换为对 `Where` 标准查询运算符方法的调用。

另请参阅

- [查询关键字 \(LINQ\)](#)
- [from 子句](#)
- [select 子句](#)
- [筛选数据](#)
- [C# 中的 LINQ](#)
- [语言集成查询 \(LINQ\)](#)

select 子句 (C# 参考)

2021/3/5 • [Edit Online](#)

在查询表达式中，`select` 子句指定在执行查询时产生的值的类型。根据计算所有以前的子句以及根据 `select` 子句本身的所有表达式得出结果。查询表达式必须以 `select` 子句或 `group` 子句结尾。

以下示例演示查询表达式中的简单的 `select` 子句。

```
class SelectSample1
{
    static void Main()
    {
        //Create the data source
        List<int> Scores = new List<int>() { 97, 92, 81, 60 };

        // Create the query.
        IEnumerable<int> queryHighScores =
            from score in Scores
            where score > 80
            select score;

        // Execute the query.
        foreach (int i in queryHighScores)
        {
            Console.Write(i + " ");
        }
    }
}
//Output: 97 92 81
```

`select` 子句生成的序列的类型确定查询变量 `queryHighScores` 的类型。在最简单的情况下，`select` 子句仅指定范围变量。这将导致返回的序列包含与数据源类型相同的元素。有关详细信息，请参阅 [LINQ 查询操作中的类型关系](#)。但是，`select` 子句还提供了强大的机制，用于将源数据转换(或投影)为新类型。有关详细信息，请参阅 [使用 LINQ 进行数据转换 \(C#\)](#)。

示例

以下示例展示 `select` 子句可能采用的所有不同窗体。在每个查询中，请注意 `select` 子句和查询变量 (`studentQuery1`、`studentQuery2` 等) 类型之间的关系。

```
class SelectSample2
{
    // Define some classes
    public class Student
    {
        public string First { get; set; }
        public string Last { get; set; }
        public int ID { get; set; }
        public List<int> Scores;
        public ContactInfo GetContactInfo(SelectSample2 app, int id)
        {
            ContactInfo cInfo =
                (from ci in app.contactList
                where ci.ID == id
                select ci)
                .FirstOrDefault();
```

```

        return cinfo;
    }

    public override string ToString()
    {
        return First + " " + Last + ":" + ID;
    }
}

public class ContactInfo
{
    public int ID { get; set; }
    public string Email { get; set; }
    public string Phone { get; set; }
    public override string ToString() { return Email + "," + Phone; }
}

public class ScoreInfo
{
    public double Average { get; set; }
    public int ID { get; set; }
}

// The primary data source
List<Student> students = new List<Student>()
{
    new Student {First="Svetlana", Last="Omelchenko", ID=111, Scores= new List<int>() {97, 92, 81, 60}},
    new Student {First="Claire", Last="O'Donnell", ID=112, Scores= new List<int>() {75, 84, 91, 39}},
    new Student {First="Sven", Last="Mortensen", ID=113, Scores= new List<int>() {88, 94, 65, 91}},
    new Student {First="Cesar", Last="Garcia", ID=114, Scores= new List<int>() {97, 89, 85, 82}},
};

// Separate data source for contact info.
List<ContactInfo> contactList = new List<ContactInfo>()
{
    new ContactInfo {ID=111, Email="SvetlanO@Contoso.com", Phone="206-555-0108"},
    new ContactInfo {ID=112, Email="ClaireO@Contoso.com", Phone="206-555-0298"},
    new ContactInfo {ID=113, Email="SvenMort@Contoso.com", Phone="206-555-1130"},
    new ContactInfo {ID=114, Email="CesarGar@Contoso.com", Phone="206-555-0521"}
};

static void Main(string[] args)
{
    SelectSample2 app = new SelectSample2();

    // Produce a filtered sequence of unmodified Students.
    IEnumerable<Student> studentQuery1 =
        from student in app.students
        where student.ID > 111
        select student;

    Console.WriteLine("Query1: select range_variable");
    foreach (Student s in studentQuery1)
    {
        Console.WriteLine(s.ToString());
    }

    // Produce a filtered sequence of elements that contain
    // only one property of each Student.
    IEnumerable<String> studentQuery2 =
        from student in app.students
        where student.ID > 111
        select student.Last;

    Console.WriteLine("\r\n studentQuery2: select range_variable.Property");
    foreach (string s in studentQuery2)
    {

```

```

        Console.WriteLine(s);
    }

    // Produce a filtered sequence of objects created by
    // a method call on each Student.
    IEnumerable<ContactInfo> studentQuery3 =
        from student in app.students
        where student.ID > 111
        select student.GetContactInfo(app, student.ID);

    Console.WriteLine("\r\n studentQuery3: select range_variable.Method");
    foreach (ContactInfo ci in studentQuery3)
    {
        Console.WriteLine(ci.ToString());
    }

    // Produce a filtered sequence of ints from
    // the internal array inside each Student.
    IEnumerable<int> studentQuery4 =
        from student in app.students
        where student.ID > 111
        select student.Scores[0];

    Console.WriteLine("\r\n studentQuery4: select range_variable[index]");
    foreach (int i in studentQuery4)
    {
        Console.WriteLine("First score = {0}", i);
    }

    // Produce a filtered sequence of doubles
    // that are the result of an expression.
    IEnumerable<double> studentQuery5 =
        from student in app.students
        where student.ID > 111
        select student.Scores[0] * 1.1;

    Console.WriteLine("\r\n studentQuery5: select expression");
    foreach (double d in studentQuery5)
    {
        Console.WriteLine("Adjusted first score = {0}", d);
    }

    // Produce a filtered sequence of doubles that are
    // the result of a method call.
    IEnumerable<double> studentQuery6 =
        from student in app.students
        where student.ID > 111
        select student.Scores.Average();

    Console.WriteLine("\r\n studentQuery6: select expression2");
    foreach (double d in studentQuery6)
    {
        Console.WriteLine("Average = {0}", d);
    }

    // Produce a filtered sequence of anonymous types
    // that contain only two properties from each Student.
    var studentQuery7 =
        from student in app.students
        where student.ID > 111
        select new { student.First, student.Last };

    Console.WriteLine("\r\n studentQuery7: select new anonymous type");
    foreach (var item in studentQuery7)
    {
        Console.WriteLine("{0}, {1}", item.Last, item.First);
    }

    // Produce a filtered sequence of named objects that contain

```

```

// a method return value and a property from each Student.
// Use named types if you need to pass the query variable
// across a method boundary.
IEnumerable<ScoreInfo> studentQuery8 =
    from student in app.students
    where student.ID > 111
    select new ScoreInfo
    {
        Average = student.Scores.Average(),
        ID = student.ID
    };

Console.WriteLine("\r\n studentQuery8: select new named type");
foreach (ScoreInfo si in studentQuery8)
{
    Console.WriteLine("ID = {0}, Average = {1}", si.ID, si.Average);
}

// Produce a filtered sequence of students who appear on a contact list
// and whose average is greater than 85.
IEnumerable<ContactInfo> studentQuery9 =
    from student in app.students
    where student.Scores.Average() > 85
    join ci in app.contactList on student.ID equals ci.ID
    select ci;

Console.WriteLine("\r\n studentQuery9: select result of join clause");
foreach (ContactInfo ci in studentQuery9)
{
    Console.WriteLine("ID = {0}, Email = {1}", ci.ID, ci.Email);
}

// Keep the console window open in debug mode
Console.WriteLine("Press any key to exit.");
Console.ReadKey();
}

/*
 * Output
Query1: select range_variable
Claire O'Donnell:112
Sven Mortensen:113
Cesar Garcia:114

studentQuery2: select range_variable.Property
O'Donnell
Mortensen
Garcia

studentQuery3: select range_variable.Method
ClaireO@Contoso.com,206-555-0298
SvenMort@Contoso.com,206-555-1130
CesarGar@Contoso.com,206-555-0521

studentQuery4: select range_variable[index]
First score = 75
First score = 88
First score = 97

studentQuery5: select expression
Adjusted first score = 82.5
Adjusted first score = 96.8
Adjusted first score = 106.7

studentQuery6: select expression2
Average = 72.25
Average = 84.5
Average = 88.25

studentQuery7: select new anonymous type

```

```
O'Donnell, Claire
Mortensen, Sven
Garcia, Cesar

studentQuery8: select new named type
ID = 112, Average = 72.25
ID = 113, Average = 84.5
ID = 114, Average = 88.25

studentQuery9: select result of join clause
ID = 114, Email = CesarGar@Contoso.com
*/
```

如前面示例中的 `studentQuery8` 所示，有时可能想要返回序列的元素仅包含一部分源元素属性。通过让返回序列尽可能变小，可以减少内存需求并提高执行查询的速度。在 `select` 子句中创建匿名类型并使用对象初始值设定项通过源元素中的相应属性初始化该类型可以完成此操作。有关如何执行此操作的示例，请参阅[对象和集合初始值设定项](#)。

备注

在编译时，`select` 子句被转换为 `Select` 标准查询运算符的方法调用。

请参阅

- [C# 参考](#)
- [查询关键字 \(LINQ\)](#)
- [from 子句](#)
- [分部\(方法\) \(C# 参考\)](#)
- [匿名类型](#)
- [C# 中的 LINQ](#)
- [语言集成查询 \(LINQ\)](#)

group 子句 (C# 参考)

2021/3/5 • • [Edit Online](#)

`group` 子句返回一个 `IGrouping< TKey, TElement >` 对象序列，这些对象包含零个或更多与该组的键值匹配的项。例如，可以按照每个字符串中的第一个字母对字符串序列进行分组。在这种情况下，第一个字母就是键，类型为 `char`，并且存储在每个 `IGrouping< TKey, TElement >` 对象的 `Key` 属性中。编译器可推断键的类型。

可以用 `group` 子句结束查询表达式，如以下示例所示：

```
// Query variable is an IEnumerable<IGrouping<char, Student>>
var studentQuery1 =
    from student in students
    group student by student.Last[0];
```

如果要对每个组执行附加查询操作，可使用上下文关键字 `into` 指定一个临时标识符。使用 `into` 时，必须继续编写该查询，并最终使用一个 `select` 语句或另一个 `group` 子句结束该查询，如以下代码摘录所示：

```
// Group students by the first letter of their last name
// Query variable is an IEnumerable<IGrouping<char, Student>>
var studentQuery2 =
    from student in students
    group student by student.Last[0] into g
    orderby g.Key
    select g;
```

对于含有和不含 `into` 的 `group`，本文中的“示例”部分提供有关其用法的更完整示例。

枚举查询分组的结果

由于 `group` 查询产生的 `IGrouping< TKey, TElement >` 对象实质上是一个由列表组成的列表，因此必须使用嵌套的 `foreach` 循环来访问每一组中的各个项。外部循环用于循环访问组键，内部循环用于循环访问组本身包含的每个项。组可能具有键，但没有元素。下面的 `foreach` 循环执行上述代码示例中的查询：

```
// Iterate group items with a nested foreach. This IGrouping encapsulates
// a sequence of Student objects, and a Key of type char.
// For convenience, var can also be used in the foreach statement.
foreach (IGrouping<char, Student> studentGroup in studentQuery2)
{
    Console.WriteLine(studentGroup.Key);
    // Explicit type for student could also be used here.
    foreach (var student in studentGroup)
    {
        Console.WriteLine(" {0}, {1}", student.Last, student.First);
    }
}
```

键类型

组键可以是任何类型，如字符串、内置数值类型、用户定义的命名类型或匿名类型。

按字符串分组

上述代码示例使用 `char`。可轻松改为指定字符串键，如完整的姓氏：

```
// Same as previous example except we use the entire last name as a key.  
// Query variable is an IEnumerable<IGrouping<string, Student>>  
var studentQuery3 =  
    from student in students  
    group student by student.Last;
```

按布尔值分组

下面的示例演示使用布尔值作为键将结果划分成两个组。请注意，该值由 `group` 子句中的子表达式生成。

```

class GroupSample1
{
    // The element type of the data source.
    public class Student
    {
        public string First { get; set; }
        public string Last { get; set; }
        public int ID { get; set; }
        public List<int> Scores;
    }

    public static List<Student> GetStudents()
    {
        // Use a collection initializer to create the data source. Note that each element
        // in the list contains an inner sequence of scores.
        List<Student> students = new List<Student>
        {
            new Student {First="Svetlana", Last="Omelchenko", ID=111, Scores= new List<int> {97, 72, 81,
60}},

            new Student {First="Claire", Last="O'Donnell", ID=112, Scores= new List<int> {75, 84, 91, 39}},
            new Student {First="Sven", Last="Mortensen", ID=113, Scores= new List<int> {99, 89, 91, 95}},
            new Student {First="Cesar", Last="Garcia", ID=114, Scores= new List<int> {72, 81, 65, 84}},
            new Student {First="Debra", Last="Garcia", ID=115, Scores= new List<int> {97, 89, 85, 82}}
        };

        return students;
    }

    static void Main()
    {
        // Obtain the data source.
        List<Student> students = GetStudents();

        // Group by true or false.
        // Query variable is an IEnumerable<IGrouping<bool, Student>>
        var booleanGroupQuery =
            from student in students
            group student by student.Scores.Average() >= 80; //pass or fail!

        // Execute the query and access items in each group
        foreach (var studentGroup in booleanGroupQuery)
        {
            Console.WriteLine(studentGroup.Key == true ? "High averages" : "Low averages");
            foreach (var student in studentGroup)
            {
                Console.WriteLine("    {0}, {1}:{2}", student.Last, student.First, student.Scores.Average());
            }
        }

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}

/* Output:
Low averages
Omelchenko, Svetlana:77.5
O'Donnell, Claire:72.25
Garcia, Cesar:75.5
High averages
Mortensen, Sven:93.5
Garcia, Debra:88.25
*/

```

按数值范围分组

下一示例使用表达式创建表示百分比范围的数值组键。请注意，该示例使用 `let` 作为方法调用结果的方便存储位

置，因此无需在 `group` 子句中调用该方法两次。若要详细了解如何在查询表达式中安全使用方法，请参阅[在查询表达式中处理异常](#)。

```
class GroupSample2
{
    // The element type of the data source.
    public class Student
    {
        public string First { get; set; }
        public string Last { get; set; }
        public int ID { get; set; }
        public List<int> Scores;
    }

    public static List<Student> GetStudents()
    {
        // Use a collection initializer to create the data source. Note that each element
        // in the list contains an inner sequence of scores.
        List<Student> students = new List<Student>
        {
            new Student {First="Svetlana", Last="Omelchenko", ID=111, Scores= new List<int> {97, 72, 81,
60}},
            new Student {First="Claire", Last="O'Donnell", ID=112, Scores= new List<int> {75, 84, 91, 39}},
            new Student {First="Sven", Last="Mortensen", ID=113, Scores= new List<int> {99, 89, 91, 95}},
            new Student {First="Cesar", Last="Garcia", ID=114, Scores= new List<int> {72, 81, 65, 84}},
            new Student {First="Debra", Last="Garcia", ID=115, Scores= new List<int> {97, 89, 85, 82}}
        };

        return students;
    }

    // This method groups students into percentile ranges based on their
    // grade average. The Average method returns a double, so to produce a whole
    // number it is necessary to cast to int before dividing by 10.
    static void Main()
    {
        // Obtain the data source.
        List<Student> students = GetStudents();

        // Write the query.
        var studentQuery =
            from student in students
            let avg = (int)student.Scores.Average()
            group student by (avg / 10) into g
            orderby g.Key
            select g;

        // Execute the query.
        foreach (var studentGroup in studentQuery)
        {
            int temp = studentGroup.Key * 10;
            Console.WriteLine("Students with an average between {0} and {1}", temp, temp + 10);
            foreach (var student in studentGroup)
            {
                Console.WriteLine("  {0}, {1}:{2}", student.Last, student.First, student.Scores.Average());
            }
        }

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}

/* Output:
   Students with an average between 70 and 80
   Omelchenko, Svetlana:77.5
   O'Donnell, Claire:72.25
   Cesar, Garcia:65.5
   Debra, Garcia:85.5
   Sven, Mortensen:95.5

```

```
Garcia, Cesar:85.5
Students with an average between 80 and 90
Garcia, Debra:88.25
Students with an average between 90 and 100
Mortensen, Sven:93.5
*/
```

按复合键分组

希望按照多个键对元素进行分组时，可使用复合键。使用匿名类型或命名类型来存储键元素，创建复合键。在下面的示例中，假定已经使用名为 `surname` 和 `city` 的两个成员声明了类 `Person`。`group` 子句会为每组姓氏和城市相同的人员创建一个单独的组。

```
group person by new {name = person.surname, city = person.city};
```

如果必须将查询变量传递给其他方法，请使用命名类型。使用键的自动实现的属性创建一个特殊类，然后替代 `Equals` 和 `GetHashCode` 方法。还可以使用结构，在此情况下，并不严格要求替代这些方法。有关详细信息，请参阅[如何使用自动实现的属性实现轻量类](#)和[如何在目录树中查询重复文件](#)。后文包含的代码示例演示了如何将复合键与命名类型结合使用。

示例

下面的示例演示在没有向组应用附加查询逻辑时，将源数据按顺序放入组中的标准模式。这称为不带延续的分组。字符串数组中的元素按照它们的首字母进行分组。查询的结果是 `IGrouping< TKey, TElement >` 类型（包含一个 `char` 类型的公共 `Key` 属性）和一个 `IEnumerable< T >` 集合（在分组中包含每个项）。

`group` 子句的结果是由序列组成的序列。因此，若要访问返回的每个组中的单个元素，请在循环访问组键的循环内使用嵌套的 `foreach` 循环，如以下示例所示。

```
class GroupExample1
{
    static void Main()
    {
        // Create a data source.
        string[] words = { "blueberry", "chimpanzee", "abacus", "banana", "apple", "cheese" };

        // Create the query.
        var wordGroups =
            from w in words
            group w by w[0];

        // Execute the query.
        foreach (var wordGroup in wordGroups)
        {
            Console.WriteLine("Words that start with the letter '{0}':", wordGroup.Key);
            foreach (var word in wordGroup)
            {
                Console.WriteLine(word);
            }
        }

        // Keep the console window open in debug mode
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}

/* Output:
   Words that start with the letter 'b':
   blueberry
   banana
   Words that start with the letter 'c':
   chimpanzee
   cheese
   Words that start with the letter 'a':
   abacus
   apple
*/
```

示例

此示例演示在创建组之后，如何使用通过 `into` 实现的延续对这些组执行附加逻辑。有关详细信息，请参阅 [into](#)。下面的示例查询每个组，仅选择键值为元音的元素。

```

class GroupClauseExample2
{
    static void Main()
    {
        // Create the data source.
        string[] words2 = { "blueberry", "chimpanzee", "abacus", "banana", "apple", "cheese", "elephant",
"umbrella", "anteater" };

        // Create the query.
        var wordGroups2 =
            from w in words2
            group w by w[0] into grps
            where (grps.Key == 'a' || grps.Key == 'e' || grps.Key == 'i'
                || grps.Key == 'o' || grps.Key == 'u')
            select grps;

        // Execute the query.
        foreach (var wordGroup in wordGroups2)
        {
            Console.WriteLine("Groups that start with a vowel: {0}", wordGroup.Key);
            foreach (var word in wordGroup)
            {
                Console.WriteLine("    {0}", word);
            }
        }

        // Keep the console window open in debug mode
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}

/* Output:
   Groups that start with a vowel: a
       abacus
       apple
       anteater
   Groups that start with a vowel: e
       elephant
   Groups that start with a vowel: u
       umbrella
*/

```

注解

在编译时，`group` 子句转换为对 `GroupBy` 方法的调用。

请参阅

- [IGrouping< TKey, TElement >](#)
- [GroupBy](#)
- [ThenBy](#)
- [ThenByDescending](#)
- [查询关键字](#)
- [语言集成查询 \(LINQ\)](#)
- [创建嵌套组](#)
- [对查询结果进行分组](#)
- [对分组操作执行子查询](#)

into (C# 参考)

2021/3/5 • [Edit Online](#)

可使用 `into` 上下文关键字创建临时标识符，将 `group`、`join` 或 `select` 子句的结果存储至新标识符。此标识符本身可以是附加查询命令的生成器。有时称在 `group` 或 `select` 子句中使用新标识符为“延续”。

示例

下面的示例演示使用 `into` 关键字来启用具有推断类型 `IGrouping` 的临时标识符 `fruitGroup`。通过使用该标识符，可对每个组调用 `Count` 方法，并且仅选择那些包含两个或更多个单词的组。

```
class IntoSample1
{
    static void Main()
    {

        // Create a data source.
        string[] words = { "apples", "blueberries", "oranges", "bananas", "apricots" };

        // Create the query.
        var wordGroups1 =
            from w in words
            group w by w[0] into fruitGroup
            where fruitGroup.Count() >= 2
            select new { FirstLetter = fruitGroup.Key, Words = fruitGroup.Count() };

        // Execute the query. Note that we only iterate over the groups,
        // not the items in each group
        foreach (var item in wordGroups1)
        {
            Console.WriteLine("{0} has {1} elements.", item.FirstLetter, item.Words);
        }

        // Keep the console window open in debug mode
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}

/* Output:
   a has 2 elements.
   b has 2 elements.
*/
```

仅当希望对每个组执行附加查询操作时，才需在 `group` 子句中使用 `into`。有关详细信息，请参阅 [group 子句](#)。

有关在 `join` 子句中使用 `into` 的示例，请参见 [join 子句](#)。

另请参阅

- [查询关键字 \(LINQ\)](#)
- [C# 中的 LINQ](#)
- [group 子句](#)

orderby 子句 (C# 参考)

2020/11/2 • [Edit Online](#)

在查询表达式中，`orderby` 子句可导致返回的序列或子序列(组)以升序或降序排序。若要执行一个或多个次级排序操作，可以指定多个键。元素类型的默认比较器执行排序。默认的排序顺序为升序。还可以指定自定义比较器。但是，只适用于使用基于方法的语法。有关详细信息，请参阅[对数据进行排序](#)。

示例

在以下示例中，第一个查询按字母顺序从 A 开始对字词排序，而第二个查询则按降序对相同的字词排序。（`ascending` 关键字是默认排序值，可省略。）

```

class OrderbySample1
{
    static void Main()
    {
        // Create a delicious data source.
        string[] fruits = { "cherry", "apple", "blueberry" };

        // Query for ascending sort.
        IEnumerable<string> sortAscendingQuery =
            from fruit in fruits
            orderby fruit // "ascending" is default
            select fruit;

        // Query for descending sort.
        IEnumerable<string> sortDescendingQuery =
            from w in fruits
            orderby w descending
            select w;

        // Execute the query.
        Console.WriteLine("Ascending:");
        foreach (string s in sortAscendingQuery)
        {
            Console.WriteLine(s);
        }

        // Execute the query.
        Console.WriteLine(Environment.NewLine + "Descending:");
        foreach (string s in sortDescendingQuery)
        {
            Console.WriteLine(s);
        }

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}

/* Output:
Ascending:
apple
blueberry
cherry

Descending:
cherry
blueberry
apple
*/

```

示例

以下示例对学生的姓氏进行主要排序，然后对其名字进行次要排序。

```

class OrderbySample2
{
    // The element type of the data source.
    public class Student
    {
        public string First { get; set; }
        public string Last { get; set; }
        public int ID { get; set; }
    }

    public static List<Student> GetStudents()

```

```

{
    // Use a collection initializer to create the data source. Note that each element
    // in the list contains an inner sequence of scores.
    List<Student> students = new List<Student>
    {
        new Student {First="Svetlana", Last="Omelchenko", ID=111},
        new Student {First="Claire", Last="O'Donnell", ID=112},
        new Student {First="Sven", Last="Mortensen", ID=113},
        new Student {First="Cesar", Last="Garcia", ID=114},
        new Student {First="Debra", Last="Garcia", ID=115}
    };
}

return students;
}
static void Main(string[] args)
{
    // Create the data source.
    List<Student> students = GetStudents();

    // Create the query.
    IEnumerable<Student> sortedStudents =
        from student in students
        orderby student.Last ascending, student.First ascending
        select student;

    // Execute the query.
    Console.WriteLine("sortedStudents:");
    foreach (Student student in sortedStudents)
        Console.WriteLine(student.Last + " " + student.First);

    // Now create groups and sort the groups. The query first sorts the names
    // of all students so that they will be in alphabetical order after they are
    // grouped. The second orderby sorts the group keys in alpha order.
    var sortedGroups =
        from student in students
        orderby student.Last, student.First
        group student by student.Last[0] into newGroup
        orderby newGroup.Key
        select newGroup;

    // Execute the query.
    Console.WriteLine(Environment.NewLine + "sortedGroups:");
    foreach (var studentGroup in sortedGroups)
    {
        Console.WriteLine(studentGroup.Key);
        foreach (var student in studentGroup)
        {
            Console.WriteLine("  {0}, {1}", student.Last, student.First);
        }
    }

    // Keep the console window open in debug mode
    Console.WriteLine("Press any key to exit.");
    Console.ReadKey();
}
/* Output:
sortedStudents:
Garcia Cesar
Garcia Debra
Mortensen Sven
O'Donnell Claire
Omelchenko Svetlana

sortedGroups:
G
  Garcia, Cesar
  Garcia, Debra
M

```

Mortensen, Sven
0
O'Donnell, Claire
Omelchenko, Svetlana
*/

备注

编译时，`orderby` 子句将转换为对 `OrderBy` 方法的调用。`orderby` 子句中的多个关键值将转换为 `ThenBy` 方法调用。

请参阅

- [C# 参考](#)
- [查询关键字 \(LINQ\)](#)
- [C# 中的 LINQ](#)
- [group 子句](#)
- [语言集成查询 \(LINQ\)](#)

join 子句 (C# 参考)

2020/11/2 • [Edit Online](#)

`join` 子句可用于将来自不同源序列并且在对象模型中没有直接关系的元素相关联。唯一的要求是每个源中的元素需要共享某个可以进行比较以判断是否相等的值。例如，食品经销商可能拥有某种产品的供应商列表以及买主列表。例如，可以使用 `join` 子句创建该产品同一指定地区供应商和买主的列表。

`join` 子句将 2 个源序列作为输入。每个序列中的元素都必须是可以与其他序列中的相应属性进行比较的属性，或者包含一个这样的属性。`join` 子句使用特殊 `equals` 关键字比较指定的键是否相等。`join` 子句执行的所有联接都是同等联接。`join` 子句的输出形式取决于执行的联接的具体类型。以下是 3 种最常见的联接类型：

- 内部联接
- 分组联接
- 左外部联接

内部联接

以下示例演示了一个简单的内部同等联接。此查询生成一个“产品名称/类别”对平面序列。同一类别字符串将出现在多个元素中。如果 `categories` 中的某个元素不具有匹配的 `products`，则该类别不会出现在结果中。

```
var innerJoinQuery =
    from category in categories
    join prod in products on category.ID equals prod.CategoryID
    select new { ProductName = prod.Name, Category = category.Name }; //produces flat sequence
```

有关详细信息，请参阅[执行内联](#)。

分组联接

含有 `into` 表达式的 `join` 子句称为分组联接。

```
var innerGroupJoinQuery =
    from category in categories
    join prod in products on category.ID equals prod.CategoryID into prodGroup
    select new { CategoryName = category.Name, Products = prodGroup };
```

分组联接会生成分层的结果序列，该序列将左侧源序列中的元素与右侧源序列中的一个或多个匹配元素相关联。分组联接没有等效的关系术语；它本质上是一个对象数组序列。

如果在右侧源序列中找不到与左侧源中的元素相匹配的元素，则 `join` 子句会为该项生成一个空数组。因此，分组联接基本上仍然是一种内部同等联接，区别在于分组联接将结果序列组织为多个组。

如果只选择分组联接的结果，则可访问各项，但无法识别结果所匹配的项。因此，通常更为有用的做法是：选择分组联接的结果并将其放入一个也包含该项名的新类型中，如上例所示。

当然，还可以将分组联接的结果用作其他子查询的生成器：

```
var innerGroupJoinQuery2 =  
    from category in categories  
    join prod in products on category.ID equals prod.CategoryID into prodGroup  
    from prod2 in prodGroup  
    where prod2.UnitPrice > 2.50M  
    select prod2;
```

有关详细信息，请参阅[执行分组联接](#)。

左外部联接

在左外部联接中，将返回左侧源序列中的所有元素，即使右侧序列中没有其匹配元素也是如此。若要在 LINQ 中执行左外部联接，请结合使用 `DefaultIfEmpty` 方法与分组联接，指定要在某个左侧元素不具有匹配元素时生成的默认右侧元素。可以使用 `null` 作为任何引用类型的默认值，也可以指定用户定义的默认类型。以下示例演示了用户定义的默认类型：

```
var leftOuterJoinQuery =  
    from category in categories  
    join prod in products on category.ID equals prod.CategoryID into prodGroup  
    from item in prodGroup.DefaultIfEmpty(new Product { Name = String.Empty, CategoryID = 0 })  
    select new { CatName = category.Name, ProdName = item.Name };
```

有关详细信息，请参阅[执行左外部联接](#)。

等于运算符

`join` 子句执行同等联接。换言之，只能基于 2 个项之间的相等关系进行匹配。不支持其他类型的比较，例如“大于”或“不等于”。为了表明所有联接都是同等联接，`join` 子句使用 `equals` 关键字而不是 `==` 运算符。`equals` 关键字只能在 `join` 子句中使用，并且其与 `==` 运算符之间存在一个重要差别。对于 `equals`，左键使用外部源序列，而右键使用内部源序列。外部源仅在 `equals` 的左侧位于范围内，而内部源序列仅在其右侧位于范围内。

非同等联接

通过使用多个 `from` 子句将新序列单独引入查询，可以执行非同等联接、交叉联接和其他自定义联接操作。有关详细信息，请参阅[执行自定义联接操作](#)。

对象集合联接与关系表

在 LINQ 查询表达式中，联接操作是在对象集合上执行的。不能使用与 2 个关系表完全相同的方式“联接”对象集合。在 LINQ 中，仅当 2 个源序列没有通过任何关系相互联系时，才需要使用显式 `join` 子句。使用 LINQ to SQL 时，外键表在对象模型中表示为主表的属性。例如，在 Northwind 数据库中，Customer 表与 Orders 表之间具有外键关系。将这两个表映射到对象模型时，Customer 类具有一个 Orders 属性，其中包含与该 Customer 相关联的 Orders 集合。实际上，已经为你执行了联接。

若要深入了解如何在 LINQ to SQL 的上下文中跨相关表执行查询，请参阅[如何：映射数据库关系](#)。

组合键

可通过使用组合键测试多个值是否相等。有关详细信息，请参阅[使用组合键进行联接](#)。还可以在 `group` 子句中使用组合键。

示例

以下示例比较了使用相同的匹配键对相同数据源执行内部联接、分组联接和左外部联接的结果。这些示例中添

加了一些额外的代码，以便在控制台显示中阐明结果。

```
class JoinDemonstration
{
    #region Data

        class Product
        {
            public string Name { get; set; }
            public int CategoryID { get; set; }
        }

        class Category
        {
            public string Name { get; set; }
            public int ID { get; set; }
        }

    // Specify the first data source.
    List<Category> categories = new List<Category>()
    {
        new Category {Name="Beverages", ID=001},
        new Category {Name="Condiments", ID=002},
        new Category {Name="Vegetables", ID=003},
        new Category {Name="Grains", ID=004},
        new Category {Name="Fruit", ID=005}
    };

    // Specify the second data source.
    List<Product> products = new List<Product>()
    {
        new Product {Name="Cola", CategoryID=001},
        new Product {Name="Tea", CategoryID=001},
        new Product {Name="Mustard", CategoryID=002},
        new Product {Name="Pickles", CategoryID=002},
        new Product {Name="Carrots", CategoryID=003},
        new Product {Name="Bok Choy", CategoryID=003},
        new Product {Name="Peaches", CategoryID=005},
        new Product {Name="Melons", CategoryID=005},
    };
    #endregion

    static void Main(string[] args)
    {
        JoinDemonstration app = new JoinDemonstration();

        app.InnerJoin();
        app.GroupJoin();
        app.GroupInnerJoin();
        app.GroupJoin3();
        app.LeftOuterJoin();
        app.LeftOuterJoin2();

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }

    void InnerJoin()
    {
        // Create the query that selects
        // a property from each element.
        var innerJoinQuery =
            from category in categories
            join prod in products on category.ID equals prod.CategoryID
            select new { Category = category.ID, Product = prod.Name };

        Console.WriteLine("InnerJoin:");
    }
}
```

```

// Execute the query. Access results
// with a simple foreach statement.
foreach (var item in innerJoinQuery)
{
    Console.WriteLine("{0,-10}{1}", item.Product, item.Category);
}
Console.WriteLine("InnerJoin: {0} items in 1 group.", innerJoinQuery.Count());
Console.WriteLine(System.Environment.NewLine);
}

void GroupJoin()
{
    // This is a demonstration query to show the output
    // of a "raw" group join. A more typical group join
    // is shown in the GroupInnerJoin method.

    var groupJoinQuery =
        from category in categories
        join prod in products on category.ID equals prod.CategoryID into prodGroup
        select prodGroup;

    // Store the count of total items (for demonstration only).
    int totalItems = 0;

    Console.WriteLine("Simple GroupJoin:");

    // A nested foreach statement is required to access group items.
    foreach (var prodGrouping in groupJoinQuery)
    {
        Console.WriteLine("Group:");
        foreach (var item in prodGrouping)
        {
            totalItems++;
            Console.WriteLine(" {0,-10}{1}", item.Name, item.CategoryID);
        }
    }
    Console.WriteLine("Unshaped GroupJoin: {0} items in {1} unnamed groups", totalItems,
groupJoinQuery.Count());
    Console.WriteLine(System.Environment.NewLine);
}

void GroupInnerJoin()
{
    var groupJoinQuery2 =
        from category in categories
        orderby category.ID
        join prod in products on category.ID equals prod.CategoryID into prodGroup
        select new
        {
            Category = category.Name,
            Products = from prod2 in prodGroup
                        orderby prod2.Name
                        select prod2
        };
}

//Console.WriteLine("GroupInnerJoin:");
int totalItems = 0;

Console.WriteLine("GroupInnerJoin:");
foreach (var productGroup in groupJoinQuery2)
{
    Console.WriteLine(productGroup.Category);
    foreach (var prodItem in productGroup.Products)
    {
        totalItems++;
        Console.WriteLine(" {0,-10} {1}", prodItem.Name, prodItem.CategoryID);
    }
}
Console.WriteLine("GroupInnerJoin: {0} items in {1} named groups", totalItems,
groupJoinQuery2.Count());

```

```

        Console.WriteLine(System.Environment.NewLine);
    }

    void GroupJoin3()
    {

        var groupJoinQuery3 =
            from category in categories
            join product in products on category.ID equals product.CategoryID into prodGroup
            from prod in prodGroup
            orderby prod.CategoryID
            select new { Category = prod.CategoryID, ProductName = prod.Name };

        //Console.WriteLine("GroupInnerJoin:");
        int totalItems = 0;

        Console.WriteLine("GroupJoin3:");
        foreach (var item in groupJoinQuery3)
        {
            totalItems++;
            Console.WriteLine("  {0}:{1}", item.ProductName, item.Category);
        }

        Console.WriteLine("GroupJoin3: {0} items in 1 group", totalItems);
        Console.WriteLine(System.Environment.NewLine);
    }

    void LeftOuterJoin()
    {
        // Create the query.
        var leftOuterQuery =
            from category in categories
            join prod in products on category.ID equals prod.CategoryID into prodGroup
            select prodGroup.DefaultIfEmpty(new Product() { Name = "Nothing!", CategoryID = category.ID });

        // Store the count of total items (for demonstration only).
        int totalItems = 0;

        Console.WriteLine("Left Outer Join:");

        // A nested foreach statement is required to access group items
        foreach (var prodGrouping in leftOuterQuery)
        {
            Console.WriteLine("Group:");
            foreach (var item in prodGrouping)
            {
                totalItems++;
                Console.WriteLine("  {0,-10}{1}", item.Name, item.CategoryID);
            }
        }
        Console.WriteLine("LeftOuterJoin: {0} items in {1} groups", totalItems, leftOuterQuery.Count());
        Console.WriteLine(System.Environment.NewLine);
    }

    void LeftOuterJoin2()
    {
        // Create the query.
        var leftOuterQuery2 =
            from category in categories
            join prod in products on category.ID equals prod.CategoryID into prodGroup
            from item in prodGroup.DefaultIfEmpty()
            select new { Name = item == null ? "Nothing!" : item.Name, CategoryID = category.ID };

        Console.WriteLine("LeftOuterJoin2: {0} items in 1 group", leftOuterQuery2.Count());
        // Store the count of total items
        int totalItems = 0;

        Console.WriteLine("Left Outer Join 2:");
    }
}

```

```

// Groups have been flattened.
foreach (var item in leftOuterQuery2)
{
    totalItems++;
    Console.WriteLine("{0,-10}{1}", item.Name, item.CategoryID);
}
Console.WriteLine("LeftOuterJoin2: {0} items in 1 group", totalItems);
}

/*Output:

InnerJoin:
Cola      1
Tea       1
Mustard   2
Pickles   2
Carrots   3
Bok Choy  3
Peaches   5
Melons    5
InnerJoin: 8 items in 1 group.

Unshaped GroupJoin:
Group:
    Cola      1
    Tea       1
Group:
    Mustard   2
    Pickles   2
Group:
    Carrots   3
    Bok Choy  3
Group:
    Peaches   5
    Melons    5
Unshaped GroupJoin: 8 items in 5 unnamed groups

GroupInnerJoin:
Beverages
    Cola      1
    Tea       1
Condiments
    Mustard   2
    Pickles   2
Vegetables
    Bok Choy  3
    Carrots   3
Grains
Fruit
    Melons    5
    Peaches   5
GroupInnerJoin: 8 items in 5 named groups

GroupJoin3:
    Cola:1
    Tea:1
    Mustard:2
    Pickles:2
    Carrots:3
    Bok Choy:3
    Peaches:5
    Melons:5
GroupJoin3: 8 items in 1 group

```

```
Left Outer Join:  
Group:  
    Cola      1  
    Tea       1  
Group:  
    Mustard   2  
    Pickles   2  
Group:  
    Carrots   3  
    Bok Choy  3  
Group:  
    Nothing!  4  
Group:  
    Peaches   5  
    Melons    5  
LeftOuterJoin: 9 items in 5 groups
```

```
LeftOuterJoin2: 9 items in 1 group  
Left Outer Join 2:  
Cola      1  
Tea       1  
Mustard   2  
Pickles   2  
Carrots   3  
Bok Choy  3  
Nothing!  4  
Peaches   5  
Melons    5  
LeftOuterJoin2: 9 items in 1 group  
Press any key to exit.  
*/
```

备注

后面未跟 `into` 的 `join` 子句转换为 `Join` 方法调用。后面跟 `into` 的 `join` 子句转换为 `GroupJoin` 方法调用。

另请参阅

- [查询关键字 \(LINQ\)](#)
- [语言集成查询 \(LINQ\)](#)
- [联接运算](#)
- [group 子句](#)
- [执行左外部联接](#)
- [执行内部联接](#)
- [执行分组联接](#)
- [对 Join 子句的结果进行排序](#)
- [使用组合键进行联接](#)
- [适用于 Visual Studio 的兼容数据库系统](#)

let 子句 (C# 参考)

2020/11/2 • [Edit Online](#)

在查询表达式中，存储子表达式的结果有时很有帮助，可在后续子句中使用。可以通过 `let` 关键字执行此操作，该关键字创建一个新的范围变量并通过提供的表达式结果初始化该变量。使用值进行初始化后，范围变量不能用于存储另一个值。但是，如果范围变量持有可查询类型，则可以查询该变量。

示例

以两种方式使用以下示例 `let`：

1. 创建一个可以查询其自身的可枚举类型。
2. 使查询仅调用一次范围变量 `word` 上的 `ToLower`。如果不使用 `let`，则不得不调用 `where` 子句中的每个谓词的 `ToLower`。

```
class LetSample1
{
    static void Main()
    {
        string[] strings =
        {
            "A penny saved is a penny earned.",
            "The early bird catches the worm.",
            "The pen is mightier than the sword."
        };

        // Split the sentence into an array of words
        // and select those whose first letter is a vowel.
        var earlyBirdQuery =
            from sentence in strings
            let words = sentence.Split(' ')
            from word in words
            let w = word.ToLower()
            where w[0] == 'a' || w[0] == 'e'
                || w[0] == 'i' || w[0] == 'o'
                || w[0] == 'u'
            select word;

        // Execute the query.
        foreach (var v in earlyBirdQuery)
        {
            Console.WriteLine("{0} starts with a vowel", v);
        }

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}

/* Output:
"A" starts with a vowel
"is" starts with a vowel
"a" starts with a vowel
"earned." starts with a vowel
"early" starts with a vowel
"is" starts with a vowel
*/
```

请参阅

- [C# 参考](#)
- [查询关键字 \(LINQ\)](#)
- [C# 中的 LINQ](#)
- [语言集成查询 \(LINQ\)](#)
- [在查询表达式中处理异常](#)

ascending (C# 参考)

2020/11/2 • [Edit Online](#)

查询表达式中的 [orderby 子句](#) 中使用 `ascending` 上下文关键字指定排序顺序为从小到大。由于 `ascending` 为默认排序顺序，因此无需加以指定。

示例

下面的示例说明 `ascending` 在 [orderby 子句](#) 中的用法。

```
IEnumerable<string> sortAscendingQuery =  
    from vegetable in vegetables  
    orderby vegetable ascending  
    select vegetable;
```

请参阅

- [C# 参考](#)
- [C# 中的 LINQ](#)
- [descending](#)

descending (C# 参考)

2020/11/2 • [Edit Online](#)

查询表达式中的 `orderby` 子句中使用 `descending` 上下文关键字指定排序顺序为从大到小。

示例

下面的示例说明 `descending` 在 `orderby` 子句中的用法。

```
IEnumerable<string> sortDescendingQuery =  
    from vegetable in vegetables  
    orderby vegetable descending  
    select vegetable;
```

请参阅

- [C# 参考](#)
- [C# 中的 LINQ](#)
- [ascending](#)

on (C# 参考)

2020/11/2 • [Edit Online](#)

on 上下文关键字用于在查询表达式的 [join 子句](#) 中指定联接条件。

示例

下面的示例说明了 **on** 在 [join 子句](#) 中的用法。

```
var innerJoinQuery =  
    from category in categories  
    join prod in products on category.ID equals prod.CategoryID  
    select new { ProductName = prod.Name, Category = category.Name };
```

请参阅

- [C# 参考](#)
- [语言集成查询 \(LINQ\)](#)

equals (C# 参考)

2020/11/2 • [Edit Online](#)

`equals` 上下文关键字用于在查询表达式的 `join` 子句中比较两个序列的元素。有关详细信息，请参阅 [join 子句](#)。

示例

下面的示例说明 `equals` 关键字在 `join` 子句中的用法。

```
var innerJoinQuery =
    from category in categories
    join prod in products on category.ID equals prod.CategoryID
    select new { ProductName = prod.Name, Category = category.Name };
```

请参阅

- [语言集成查询 \(LINQ\)](#)

by (C# 参考)

2020/11/2 • [Edit Online](#)

by 上下文关键字用于在查询表达式的 `group` 子句中指定应返回项的分组方式。有关详细信息，请参阅 [group 子句](#)。

示例

下面的示例演示在 `group` 字句中使用 `by` 关键字指定应根据每个学生的姓氏首字母对学生分组。

```
var query = from student in students
             group student by student.LastName[0];
```

另请参阅

- [C# 中的 LINQ](#)

in (C# 参考)

2020/11/2 • [Edit Online](#)

在以下上下文中，使用了 `in` 关键字：

- 泛型接口和委托中的[泛型类型参数](#)。
- 作为[参数修饰符](#)，它允许按引用而不是按值向方法传递参数。
- `foreach` 语句。
- LINQ 查询表达式中的[from 子句](#)。
- LINQ 查询表达式中的[join 子句](#)。

请参阅

- [C# 关键字](#)
- [C# 参考](#)

C# 运算符和表达式 (C# 参考)

2021/5/7 • [Edit Online](#)

C# 提供了许多运算符。其中许多都受到[内置类型](#)的支持，可用于对这些类型的值执行基本操作。这些运算符包括以下组：

- [算术运算符](#)，将对数值操作数执行算术运算
- [比较运算符](#)，将比较数值操作数
- [布尔逻辑运算符](#)，将对 `bool` 操作数执行逻辑运算
- [位运算符和移位运算符](#)，将对整数类型的操作数执行位运算或移位运算
- [相等运算符](#)，将检查其操作数是否相等

通常可以[重载](#)这些运算符，也就是说，可以为用户定义类型的操作数指定运算符行为。

最简单的 C# 表达式是文本(例如[整数](#)和[实数](#))和变量名称。可以使用运算符将它们组合成复杂的表达式。运算符[优先级](#)和[结合性](#)决定了表达式中操作的执行顺序。可以使用括号更改由运算符优先级和结合性决定的计算顺序。

在下面的代码中，表达式的示例位于赋值的右侧：

```
int a, b, c;
a = 7;
b = a;
c = b++;
b = a + b * c;
c = a >= 100 ? b : c / 10;
a = (int)Math.Sqrt(b * b + c * c);

string s = "String literal";
char l = s[s.Length - 1];

var numbers = new List<int>(new[] { 1, 2, 3 });
b = numbers.FindLast(n => n > 1);
```

通常情况下，表达式会生成结果，并可包含在其他表达式中。`void` 方法调用是不生成结果的表达式的示例。它只能用作[语句](#)，如下面的示例所示：

```
Console.WriteLine("Hello, world!");
```

下面是 C# 提供的一些其他类型的表达式：

- [内插字符串表达式](#)，提供创建格式化字符串的便利语法：

```
var r = 2.3;
var message = $"The area of a circle with radius {r} is {Math.PI * r * r:F3}.";
Console.WriteLine(message);
// Output:
// The area of a circle with radius 2.3 is 16.619.
```

- [Lambda 表达式](#)，可用于创建匿名函数：

```

int[] numbers = { 2, 3, 4, 5 };
var maximumSquare = numbers.Max(x => x * x);
Console.WriteLine(maximumSquare);
// Output:
// 25

```

- [查询表达式](#), 可用于直接以 C# 使用查询功能:

```

var scores = new[] { 90, 97, 78, 68, 85 };
IEnumerable<int> highScoresQuery =
    from score in scores
    where score > 80
    orderby score descending
    select score;
Console.WriteLine(string.Join(" ", highScoresQuery));
// Output:
// 97 90 85

```

可使用[表达式主体定义](#)为方法、构造函数、属性、索引器或终结器提供简洁的定义。

运算符优先级

在包含多个运算符的表达式中, 先按优先级较高的运算符计算, 再按优先级较低的运算符计算。在下面的示例中, 首先执行乘法, 因为其优先级高于加法:

```

var a = 2 + 2 * 2;
Console.WriteLine(a); // output: 6

```

使用括号更改运算符优先级所施加的计算顺序:

```

var a = (2 + 2) * 2;
Console.WriteLine(a); // output: 8

```

下表按最高优先级到最低优先级的顺序列出 C# 运算符。每行中运算符的优先级相同。

<code>x.y</code> 、 <code>f(x)</code> 、 <code>a[i]</code> 、 <code>x?.y</code> 、 <code>x?[y]</code> 、 <code>x++</code> 、 <code>x-</code> <code>-x</code> 、 <code>x!</code> 、 <code>new</code> 、 <code>typeof</code> 、 <code>checked</code> 、 <code>unchecked</code> 、 <code>default</code> 、 <code>nameof</code> 、 <code>delegate</code> 、 <code>sizeof</code> 、 <code>stackalloc</code> 、 <code>x->y</code>	基本
<code>+x</code> 、 <code>-x</code> 、 <code>!x</code> 、 <code>~x</code> 、 <code>++x</code> 、 <code>--x</code> 、 <code>^x</code> 、 <code>(T)x</code> 、 <code>await</code> 、 <code>&x</code> 、 <code>*x</code> 、 <code>true</code> 和 <code>false</code>	一元
<code>x.y</code>	范围
<code>switch</code>	<code>switch</code> 表达式
<code>with</code>	<code>with</code> 表达式
<code>x * y</code> 、 <code>x / y</code> 、 <code>x % y</code>	乘法
<code>x + y</code> 、 <code>x - y</code>	加法

`	`
<code>x << y, x >> y</code>	移位
<code>x < y, x > y, x <= y, x >= y, is, as</code>	关系和类型测试
<code>x == y, x != y</code>	相等
<code>x & y</code>	布尔逻辑 AND 或按位逻辑 AND
<code>x ^ y</code>	布尔逻辑 XOR 或按位逻辑 XOR
<code>x y</code>	布尔逻辑 OR 或按位逻辑 OR
<code>x && y</code>	条件“与”
<code>x y</code>	条件“或”
<code>x ?? y</code>	Null 合并运算符
<code>c ? t : f</code>	条件运算符
<code>x = y, x += y, x -= y, x *= y, x /= y, x %= y, x &= y, x = y, x ^= y, x <<= y, x >>= y, x ??= y, x =></code>	赋值和 lambda 声明

运算符结合性

当运算符的优先级相同，运算符的结合性决定了运算的执行顺序：

- 左结合运算符按从左到右的顺序计算。除赋值运算符和 null 合并运算符外，所有二元运算符都是左结合运算符。例如，`a + b - c` 将计算为 `(a + b) - c`。
- 右结合运算符按从右到左的顺序计算。赋值运算符、null 合并运算符和条件运算符 `?:` 是右结合运算符。例如，`x = y = z` 将计算为 `x = (y = z)`。

使用括号更改运算符结合性所施加的计算顺序：

```
int a = 13 / 5 / 2;
int b = 13 / (5 / 2);
Console.WriteLine($"a = {a}, b = {b}"); // output: a = 1, b = 6
```

操作数计算

与运算符的优先级和结合性无关，从左到右计算表达式中的操作数。以下示例展示了运算符和操作数的计算顺序：

`	`
<code>a + b</code>	<code>a, b, +</code>
<code>a + b * c</code>	<code>a, b, c, *, +</code>
<code>a / b + c * d</code>	<code>a, b, /, c, d, *, +</code>

ccc

cccc

a / (b + c) * d

a, b, c, +, /, d, *

通常，会计算所有运算符操作数。但是，某些运算符有条件地计算操作数。也就是说，此类运算符的最左侧操作数的值定义了是否应计算其他操作数，或计算其他哪些操作数。这些运算符有条件逻辑 AND (`&&`) 和 OR (`||`) 运算符、`null` 合并运算符 `??` 和 `??=`、`null` 条件运算符 `?.` 和 `?[]` 以及条件运算符 `?:`。有关详细信息，请参阅每个运算符的说明。

C# 语言规范

有关更多信息，请参阅 [C# 语言规范](#) 的以下部分：

- [表达式](#)
- [运算符](#)

请参阅

- [C# 参考](#)
- [运算符重载](#)
- [表达式树](#)

算术运算符 (C# 参考)

2020/11/2 • [Edit Online](#)

以下运算符对数值类型的操作数执行算术运算：

- 一元 `++`(增量)、`--`(减量)、`+`(加)和`-`(减)运算符
- 二元 `*`(乘法)、`/`(除法)、`%`(余数)、`+`(加法)和`-`(减法)运算符

所有整型和浮点数值类型都支持这些运算符。

对于整型类型，这些运算符(除 `++` 和 `--` 运算符以外)是为 `int`、`uint`、`long` 和 `ulong` 类型定义的。如果操作数都是其他整型类型(`sbyte`、`byte`、`short`、`ushort` 或 `char`)，它们的值将转换为 `int` 类型，这也是一个运算的结果类型。如果操作数是不同的整型类型或浮点类型，它们的值将转换为最接近的包含类型(如果存在该类型)。有关详细信息，请参阅 [C# 语言规范的数值提升部分](#)。`++` 和 `--` 运算符是为所有整型和浮点数值类型以及 `char` 类型定义的。

增量运算符 `++`

一元增量运算符 `++` 按 1 递增其操作数。操作数必须是变量、属性访问或索引器访问。

增量运算符以两种形式进行支持：后缀增量运算符 `x++` 和前缀增量运算符 `++x`。

后缀递增运算符

`x++` 的结果是此操作前的 `x` 的值，如以下示例所示：

```
int i = 3;
Console.WriteLine(i);    // output: 3
Console.WriteLine(i++);
Console.WriteLine(i);    // output: 4
```

前缀增量运算符

`++x` 的结果是此操作后的 `x` 的值，如以下示例所示：

```
double a = 1.5;
Console.WriteLine(a);    // output: 1.5
Console.WriteLine(++a); // output: 2.5
Console.WriteLine(a);    // output: 2.5
```

减量运算符 `--`

一元减量运算符 `--` 按 1 递减其操作数。操作数必须是变量、属性访问或索引器访问。

减量运算符以两种形式进行支持：后缀减量运算符 `x--` 和前缀减量运算符 `--x`。

后缀递减运算符

`x--` 的结果是此操作前的 `x` 的值，如以下示例所示：

```
int i = 3;
Console.WriteLine(i);    // output: 3
Console.WriteLine(i--); // output: 3
Console.WriteLine(i);    // output: 2
```

前缀减量运算符

--**x** 的结果是此操作后的 **x** 的值, 如以下示例所示:

```
double a = 1.5;
Console.WriteLine(a);    // output: 1.5
Console.WriteLine(--a); // output: 0.5
Console.WriteLine(a);    // output: 0.5
```

一元加和减运算符

一元 **+** 运算符返回其操作数的值。一元 **-** 运算符对其操作数的数值取负。

```
Console.WriteLine(+4);      // output: 4

Console.WriteLine(-4);      // output: -4
Console.WriteLine(--(-4)); // output: 4

uint a = 5;
var b = -a;
Console.WriteLine(b);        // output: -5
Console.WriteLine(b.GetType()); // output: System.Int64

Console.WriteLine(-double.NaN); // output: NaN
```

ulong 类型不支持一元 **-** 运算符。

乘法运算符 *****

乘法运算符 ***** 计算其操作数的乘积:

```
Console.WriteLine(5 * 2);      // output: 10
Console.WriteLine(0.5 * 2.5);   // output: 1.25
Console.WriteLine(0.1m * 23.4m); // output: 2.34
```

一元 ***** 运算符是[指针间接运算符](#)。

除法运算符 **/**

除法运算符 **/** 用它的左侧操作数除以右侧操作数。

整数除法

对于整数类型的操作数, **/** 运算符的结果为整数类型, 并且等于两个操作数之商向零舍入后的结果:

```
Console.WriteLine(13 / 5);    // output: 2
Console.WriteLine(-13 / 5);   // output: -2
Console.WriteLine(13 / -5);   // output: -2
Console.WriteLine(-13 / -5); // output: 2
```

若要获取浮点数形式的两个操作数之商, 请使用 **float**、**double** 或 **decimal** 类型:

```
Console.WriteLine(13 / 5.0);      // output: 2.6

int a = 13;
int b = 5;
Console.WriteLine((double)a / b); // output: 2.6
```

浮点除法

对于 `float`、`double` 和 `decimal` 类型，`/` 运算符的结果为两个操作数之商：

```
Console.WriteLine(16.8f / 4.1f); // output: 4.097561
Console.WriteLine(16.8d / 4.1d); // output: 4.09756097560976
Console.WriteLine(16.8m / 4.1m); // output: 4.0975609756097560975609756098
```

如果操作数之一为 `decimal`，那么另一个操作数不得为 `float` 和 `double`，因为 `float` 和 `double` 都无法隐式转换为 `decimal`。必须将 `float` 或 `double` 操作数显式转换为 `decimal` 类型。如需详细了解数值类型之间的转换，请参阅[内置数值转换](#)。

余数运算符 %

余数运算符 `%` 计算左侧操作数除以右侧操作数后的余数。

整数余数

对于整数类型的操作数，`a % b` 的结果是 $a - (a / b) * b$ 得出的值。非零余数的符号与左侧操作数的符号相同，如下例所示：

```
Console.WriteLine(5 % 4); // output: 1
Console.WriteLine(5 % -4); // output: 1
Console.WriteLine(-5 % 4); // output: -1
Console.WriteLine(-5 % -4); // output: -1
```

使用 [Math.DivRem](#) 方法计算整数除法和余数结果。

浮点余数

对于 `float` 和 `double` 操作数，有限的 `x` 和 `y` 的 `x % y` 的结果是值 `z`，因此

- `z` (如果不为零) 的符号与 `x` 的符号相同。
- `z` 的绝对值是 $|x| - n * |y|$ 得出的值，其中 `n` 是小于或等于 $|x| / |y|$ 的最大可能整数，`|x|` 和 `|y|` 分别是 `x` 和 `y` 的绝对值。

NOTE

计算余数的此方法类似于用于整数操作数的方法，但与 IEEE 754 规范不同。如果需要符合 IEEE 754 规范的余数运算，请使用 [Math.IEEERemainder](#) 方法。

有关非限定操作数的 `%` 运算符行为的信息，请参阅 [C# 语言规范的余数运算符](#) 章节。

对于 `decimal` 操作数，余数运算符 `%` 等效于 `System.Decimal` 类型的 [余数运算符](#)。

以下示例演示了具有浮动操作数的余数运算符的行为：

```
Console.WriteLine(-5.2f % 2.0f); // output: -1.2
Console.WriteLine(5.9 % 3.1); // output: 2.8
Console.WriteLine(5.9m % 3.1m); // output: 2.8
```

加法运算符 +

加法运算符 `+` 计算其操作数的和。

```
Console.WriteLine(5 + 4);      // output: 9
Console.WriteLine(5 + 4.3);    // output: 9.3
Console.WriteLine(5.1m + 4.2m); // output: 9.3
```

还可以将 `+` 运算符用于字符串串联和委托组合。有关详细信息, 请参阅 [+ 和 += 运算符](#) 一文。

减法运算符 -

减法运算符 `-` 从其左侧操作数中减去其右侧操作数:

```
Console.WriteLine(47 - 3);      // output: 44
Console.WriteLine(5 - 4.3);     // output: 0.7
Console.WriteLine(7.5m - 2.3m); // output: 5.2
```

还可以使用 `-` 运算符删除委托。有关详细信息, 请参阅 [- 和 -= 运算符](#) 一文。

复合赋值

对于二元运算符 `op`, 窗体的复合赋值表达式

```
x op= y
```

等效于

```
x = x op y
```

不同的是 `x` 只计算一次。

以下示例使用算术运算符演示了复合赋值的用法:

```
int a = 5;
a += 9;
Console.WriteLine(a); // output: 14

a -= 4;
Console.WriteLine(a); // output: 10

a *= 2;
Console.WriteLine(a); // output: 20

a /= 4;
Console.WriteLine(a); // output: 5

a %= 3;
Console.WriteLine(a); // output: 2
```

由于[数值提升](#), `op` 运算的结果可能不会隐式转换为 `x` 的 `T` 类型。在这种情况下, 如果 `op` 是预定义的运算符并且运算的结果可以显式转换为 `x` 的类型 `T`, 则形式为 `x op= y` 的复合赋值表达式等效于

`x = (T)(x op y)`, 但 `x` 仅计算一次。以下示例演示了该行为:

```
byte a = 200;
byte b = 100;

var c = a + b;
Console.WriteLine(c.GetType()); // output: System.Int32
Console.WriteLine(c); // output: 300

a += b;
Console.WriteLine(a); // output: 44
```

你还可以使用 `+=` 和 `-=` 运算符分别订阅和取消订阅事件。有关详细信息，请参阅[如何订阅和取消订阅事件](#)。

运算符优先级和关联性

以下列表对优先级由高到低的顺序对算术运算符进行排序：

- 后缀增量 `x++` 和减量 `x--` 运算符
- 前缀增量 `++x` 和减量 `--x` 以及一元 `+` 和 `-` 运算符
- 乘法 `*`、`/` 和 `%` 运算符
- 加法 `+` 和 `-` 运算符

二元算数运算符是左结合运算符。也就是说，具有相同优先级的运算符按从左至右的顺序计算。

使用括号 `()` 更改由运算符优先级和关联性决定的计算顺序。

```
Console.WriteLine(2 + 2 * 2); // output: 6
Console.WriteLine((2 + 2) * 2); // output: 8

Console.WriteLine(9 / 5 / 2); // output: 0
Console.WriteLine(9 / (5 / 2)); // output: 4
```

如需了解按优先级排序的完整 C# 运算符列表，请参阅[C# 运算符](#)一文中的[运算符优先级](#)部分。

算术溢出和被零除

当算术运算的结果超出所涉及数值类型的可能有限值范围时，算术运算符的行为取决于其操作数的类型。

整数算术溢出

整数被零除总是引发 [DivideByZeroException](#)。

在整数算术溢出的情况下，溢出检查上下文（可能已检查或未检查）将控制引发的行为：

- 在已检查的上下文中，如果在常数表达式中发生溢出，则会发生编译时错误。否则，当在运行时执行此运算时，则引发 [OverflowException](#)。
- 在未检查的上下文中，会通过丢弃任何不适应目标类型的高位来将结果截断。

在使用[已检查和未检查的](#)语句时，可以使用 `checked` 和 `unchecked` 运算符来控制溢出检查上下文，在该上下文中将计算一个表达式：

```
int a = int.MaxValue;
int b = 3;

Console.WriteLine(unchecked(a + b)); // output: -2147483646
try
{
    int d = checked(a + b);
}
catch(OverflowException)
{
    Console.WriteLine($"Overflow occurred when adding {a} to {b}.");
}
```

默认情况下，算术运算发生在 *unchecked* 上下文中。

浮点运算溢出

使用 `float` 和 `double` 类型的算术运算永远不会引发异常。使用这些类型的算术运算的结果可能是表示无穷大和非数字的特殊值之一：

```
double a = 1.0 / 0.0;
Console.WriteLine(a); // output: Infinity
Console.WriteLine(double.IsInfinity(a)); // output: True

Console.WriteLine(double.MaxValue + double.MaxValue); // output: Infinity

double b = 0.0 / 0.0;
Console.WriteLine(b); // output: NaN
Console.WriteLine(double.IsNaN(b)); // output: True
```

对于 `decimal` 类型的操作数，算术溢出始终会引发 `OverflowException`，并且被零除始终会引发 `DivideByZeroException`。

舍入误差

由于实数和浮点运算的浮点表达形式的常规限制，在使用浮点类型的计算中可能会发生舍入误差。也就是说，表达式得出的结果可能与预期的数学运算结果不同。以下示例演示了几种此类情况：

有关详细信息，请参阅 [System.Double](#)、[System.Single](#) 或 [System.Decimal](#) 参考页上的注解。

运算符可重载性

用户定义类型可以重载一元(`++`、`--`、`+` 和 `-`)和二元(`*`、`/`、`%`、`+` 和 `-`)算术运算符。重载二元运算符时，对应的复合赋值运算符也会隐式重载。用户定义类型不能显式重载复合赋值运算符。

C# 语言规范

有关更多信息，请参阅 [C# 语言规范](#)的以下部分：

- [后缀增量和减量运算符](#)
- [前缀增量和减量运算符](#)
- [一元加运算符](#)
- [一元减运算符](#)
- [乘法运算符](#)
- [除法运算符](#)
- [余数运算符](#)
- [加法运算符](#)
- [减法运算符](#)
- [复合赋值](#)
- [checked 和 unchecked 运算符](#)
- [数值提升](#)

请参阅

- [C# 参考](#)
- [C# 运算符和表达式](#)
- [System.Math](#)
- [System.MathF](#)
- [.NET 中的数字](#)

布尔逻辑运算符 (C# 参考)

2020/11/2 • [Edit Online](#)

以下运算符使用 `bool` 操作数执行逻辑运算：

- 一元 `!` (逻辑非) 运算符。
- 二元 `&` (逻辑与)、`|` (逻辑或) 和 `^` (逻辑异或) 运算符。这些运算符始终计算两个操作数。
- 二元 `&&` (条件逻辑与) 和 `||` (条件逻辑或) 运算符。这些运算符仅在必要时才计算右侧操作数。

对于 [整型数值类型](#) 的操作数，`&`、`|` 和 `^` 运算符执行位逻辑运算。有关详细信息，请参阅[位运算符和移位运算符](#)。

逻辑非运算符 !

一元前缀 `!` 运算符计算操作数的逻辑非。也就是说，如果操作数的计算结果为 `false`，它生成 `true`；如果操作数的计算结果为 `true`，它生成 `false`：

```
bool passed = false;
Console.WriteLine(!passed); // output: True
Console.WriteLine(!true); // output: False
```

从 C# 8.0 起，一元后缀 `!` 运算符为 [null 包容运算符](#)。

逻辑 AND 运算符 &

`&` 运算符计算操作数的逻辑与。如果 `x` 和 `y` 的计算结果都为 `true`，则 `x & y` 的结果为 `true`。否则，结果为 `false`。

即使左侧操作数计算结果为 `false`，`&` 运算符也会计算这两个操作数，而在这种情况下，无论右侧操作数的值为何，运算结果都为 `false`。

在下面的示例中，`&` 运算符的右侧操作数是方法调用，无论左侧操作数的值如何，都会执行方法调用：

```
bool SecondOperand()
{
    Console.WriteLine("Second operand is evaluated.");
    return true;
}

bool a = false & SecondOperand();
Console.WriteLine(a);
// Output:
// Second operand is evaluated.
// False

bool b = true & SecondOperand();
Console.WriteLine(b);
// Output:
// Second operand is evaluated.
// True
```

[条件逻辑 AND 运算符](#) `&&` 也计算操作数的逻辑 AND，但如果左侧操作数的计算结果为 `false`，它就不会计算右侧操作数。

对于整型数值类型的操作数，`&` 运算符计算其操作数的位逻辑 AND。一元 `&` 运算符是 address-of 运算符。

逻辑异或运算符 ^

`^` 运算符计算操作数的逻辑异或（亦称为“逻辑 XOR”）。如果 `x` 计算结果为 `true` 且 `y` 计算结果为 `false`，或者 `x` 计算结果为 `false` 且 `y` 计算结果为 `true`，那么 `x ^ y` 的结果为 `true`。否则，结果为 `false`。也就是说，对于 `bool` 操作数，`^` 运算符的计算结果与不等运算符 `!=` 相同。

```
Console.WriteLine(true ^ true);    // output: False
Console.WriteLine(true ^ false);   // output: True
Console.WriteLine(false ^ true);   // output: True
Console.WriteLine(false ^ false);  // output: False
```

对于整型数值类型的操作数，`^` 运算符计算其操作数的位逻辑异或。

逻辑或运算符 |

`|` 运算符计算操作数的逻辑或。如果 `x` 或 `y` 的计算结果为 `true`，则 `x | y` 的结果为 `true`。否则，结果为 `false`。

即使左侧操作数计算结果为 `true`，`|` 运算符也会计算这两个操作数，而在这种情况下，无论右侧操作数的值为何，运算结果都为 `true`。

在下面的示例中，`|` 运算符的右侧操作数是方法调用，无论左侧操作数的值如何，都会执行方法调用：

```
bool SecondOperand()
{
    Console.WriteLine("Second operand is evaluated.");
    return true;
}

bool a = true | SecondOperand();
Console.WriteLine(a);
// Output:
// Second operand is evaluated.
// True

bool b = false | SecondOperand();
Console.WriteLine(b);
// Output:
// Second operand is evaluated.
// True
```

条件逻辑 OR 运算符 `||` 也计算操作数的逻辑 OR，但如果左侧操作数的计算结果为 `true`，它就不会计算右侧操作数。

对于整型数值类型的操作数，`|` 运算符计算其操作数的位逻辑 OR。

条件逻辑 AND 运算符 &&

条件逻辑与运算符 `&&`（亦称为“短路”逻辑与运算符）计算操作数的逻辑与。如果 `x` 和 `y` 的计算结果都为 `true`，则 `x && y` 的结果为 `true`。否则，结果为 `false`。如果 `x` 的计算结果为 `false`，则不计算 `y`。

在下面的示例中，`&&` 运算符的右侧操作数是方法调用，如果左侧操作数的计算结果为 `false`，就不执行方法调用：

```

bool SecondOperand()
{
    Console.WriteLine("Second operand is evaluated.");
    return true;
}

bool a = false && SecondOperand();
Console.WriteLine(a);
// Output:
// False

bool b = true && SecondOperand();
Console.WriteLine(b);
// Output:
// Second operand is evaluated.
// True

```

逻辑 AND 运算符 `&` 也计算操作数的逻辑 AND, 但始终计算两个操作数。

条件逻辑或运算符 `||`

条件逻辑或运算符 `||` (亦称为“短路”逻辑或运算符)计算操作数的逻辑或。如果 `x` 或 `y` 的计算结果为 `true`, 则 `x || y` 的结果为 `true`。否则, 结果为 `false`。如果 `x` 的计算结果为 `true`, 则不计算 `y`。

在下面的示例中, `||` 运算符的右侧操作数是方法调用, 如果左侧操作数的计算结果为 `true`, 就不执行方法调用:

```

bool SecondOperand()
{
    Console.WriteLine("Second operand is evaluated.");
    return true;
}

bool a = true || SecondOperand();
Console.WriteLine(a);
// Output:
// True

bool b = false || SecondOperand();
Console.WriteLine(b);
// Output:
// Second operand is evaluated.
// True

```

逻辑 OR 运算符 `|` 也计算操作数的逻辑 OR, 但始终计算两个操作数。

可以为 `null` 的布尔逻辑运算符

对于 `bool?` 操作数, `&` (逻辑与) 和 `|` (逻辑或) 运算符支持三值逻辑, 如下所示:

- 仅当其两个操作数的计算结果都为 `true` 时, `&` 运算符才生成 `true`。如果 `x` 或 `y` 的计算结果为 `false`, 则 `x & y` 将生成 `false` (即使另一个操作数的计算结果为 `null`)。否则, `x & y` 的结果为 `null`。
- 仅当其两个操作数的计算结果都为 `false` 时, `|` 运算符才生成 `false`。如果 `x` 或 `y` 的计算结果为 `true`, 则 `x | y` 将生成 `true` (即使另一个操作数的计算结果为 `null`)。否则, `x | y` 的结果为 `null`。

下表显示了该语义:

X	Y	X & Y	X Y
true	true	true	true
true	false	false	true
true	null	null	true
false	true	false	true
false	false	false	false
false	null	false	null
null	true	null	true
null	false	false	null
null	null	null	null

这些运算符的行为不同于值类型可以为 null 的典型运算符行为。通常情况下，为值类型的操作数定义的运算符也能与值类型可以为 null 的相应操作数一起使用。如果其任一操作数的计算结果为 `null`，此类运算符都会生成 `null`。不过，即使操作数之一的计算结果为 `null`，`&` 和 `|` 运算符也可以生成非 null。有关值类型可为空的运算符行为的详细信息，请参阅[可为空的值类型一文的提升的运算符](#)部分。

此外，你还可以将 `!` 和 `^` 运算符与 `bool?` 操作数结合使用，如下例所示：

```
bool? test = null;
Display(!test);           // output: null
Display(test ^ false);    // output: null
Display(test ^ null);     // output: null
Display(true ^ null);     // output: null

void Display(bool? b) => Console.WriteLine(b is null ? "null" : b.Value.ToString());
```

条件逻辑运算符 `&&` 和 `||` 不支持 `bool?` 操作数。

复合赋值

对于二元运算符 `op`，窗体的复合赋值表达式

```
x op= y
```

等效于

```
x = x op y
```

不同的是 `x` 只计算一次。

`&`、`|` 和 `^` 运算符支持复合赋值，如下面的示例所示：

```
bool test = true;
test &= false;
Console.WriteLine(test); // output: False

test |= true;
Console.WriteLine(test); // output: True

test ^= false;
Console.WriteLine(test); // output: True
```

NOTE

条件逻辑运算符 `&&` 和 `||` 不支持复合赋值。

运算符优先级

以下列表按优先级从高到低的顺序对逻辑运算符进行排序：

- 逻辑非运算符 `!`
- 逻辑与运算符 `&`
- 逻辑异或运算符 `^`
- 逻辑或运算符 `|`
- 条件逻辑与运算符 `&&`
- 条件逻辑或运算符 `||`

使用括号 `()` 可以更改运算符优先级决定的计算顺序：

```
Console.WriteLine(true | true & false); // output: True
Console.WriteLine((true | true) & false); // output: False

bool Operand(string name, bool value)
{
    Console.WriteLine($"Operand {name} is evaluated.");
    return value;
}

var byDefaultPrecedence = Operand("A", true) || Operand("B", true) && Operand("C", false);
Console.WriteLine(byDefaultPrecedence);
// Output:
// Operand A is evaluated.
// True

var changedOrder = (Operand("A", true) || Operand("B", true)) && Operand("C", false);
Console.WriteLine(changedOrder);
// Output:
// Operand A is evaluated.
// Operand C is evaluated.
// False
```

要了解按优先级排序的完整 C# 运算符列表，请参阅 [C# 运算符](#)一文中的 [运算符优先级](#) 部分。

运算符可重载性

用户定义类型可以重载 `!`、`&`、`|` 和 `^` 运算符。重载二元运算符时，对应的复合赋值运算符也会隐式重载。用户定义类型不能显式重载复合赋值运算符。

用户定义类型无法重载条件逻辑运算符 `&&` 和 `||`。不过，如果用户定义类型以某种方式重载 `true` 和 `false` 运算

符以及 `&` 或 `|` 运算符，可以对相应类型的操作数执行 `&&` 或 `||` 运算。有关详细信息，请参阅 C# 语言规范的[用户定义条件逻辑运算符](#)部分。

C# 语言规范

有关更多信息，请参阅 [C# 语言规范](#) 的以下部分：

- [逻辑非运算符](#)
- [逻辑运算符](#)
- [条件逻辑运算符](#)
- [复合赋值](#)

请参阅

- [C# 参考](#)
- [C# 运算符和表达式](#)
- [位运算符和移位运算符](#)

位运算符和移位运算符 (C# 参考)

2020/11/2 • [Edit Online](#)

以下运算符使用整数类型或 `char` 类型的操作数执行位运算或移位运算：

- 一元 `~` (按位求补) 运算符
- 二进制 `<<` (向左移位) 和 `>>` (向右移位) 移位运算符
- 二进制 `&` (逻辑 AND)、`|` (逻辑 OR) 和 `^` (逻辑异或) 运算符

这些运算符是针对 `int`、`uint`、`long` 和 `ulong` 类型定义的。如果两个操作数都是其他整数类型 (`sbyte`、`byte`、`short`、`ushort` 或 `char`)，它们的值将转换为 `int` 类型，这也是一个运算的结果类型。如果操作数是不同的整数类型，它们的值将转换为最接近的包含整数类型。有关详细信息，请参阅 [C# 语言规范的数值提升部分](#)。

`&`、`|` 和 `^` 运算符也是为 `bool` 类型的操作数定义的。有关详细信息，请参阅 [布尔逻辑运算符](#)。

位运算和移位运算永远不会导致溢出，并且不会在已检查和未检查的上下文中产生相同的结果。

按位求补运算符 ~

`~` 运算符通过反转每个位产生其操作数的按位求补：

```
uint a = 0b_0000_1111_0000_1111_0000_1111_0000_1100;
uint b = ~a;
Console.WriteLine(Convert.ToString(b, toBase: 2));
// Output:
// 11110000111100001111000011110011
```

也可以使用 `~` 符号来声明终结器。有关详细信息，请参阅 [终结器](#)。

左移位运算符 <<

`<<` 运算符将其左侧操作数向左移动右侧操作数定义的位数。

左移运算会放弃超出结果类型范围的高位，并将低阶空位位置设置为零，如以下示例所示：

```
uint x = 0b_1100_1001_0000_0000_0000_0001_0001;
Console.WriteLine($"Before: {Convert.ToString(x, toBase: 2)}");

uint y = x << 4;
Console.WriteLine($"After: {Convert.ToString(y, toBase: 2)}");
// Output:
// Before: 1100100100000000000000000010001
// After: 10010000000000000000000000100010000
```

由于移位运算符仅针对 `int`、`uint`、`long` 和 `ulong` 类型定义，因此运算的结果始终包含至少 32 位。如果左侧操作数是其他整数类型 (`sbyte`、`byte`、`short`、`ushort` 或 `char`)，则其值将转换为 `int` 类型，如以下示例所示：

```
byte a = 0b_1111_0001;

var b = a << 8;
Console.WriteLine(b.GetType());
Console.WriteLine($"Shifted byte: {Convert.ToString(b, toBase: 2)}");
// Output:
// System.Int32
// Shifted byte: 1111000100000000
```

有关 << 运算符的右侧操作数如何定义移位计数的信息,请参阅[移位运算符的移位计数](#)部分。

右移位运算符 >>

>> 运算符将其左侧操作数向右移动右侧操作数定义的位数。

右移位运算会放弃低位，如以下示例所示：

```
uint x = 0b_1001;
Console.WriteLine($"Before: {Convert.ToString(x, toBase: 2), 4}");

uint y = x >> 2;
Console.WriteLine($"After: {Convert.ToString(y, toBase: 2), 4}");
// Output:
// Before: 1001
// After:    10
```

高顺序空位位置是根据左侧操作数类型设置的，如下所示：

- 如果左侧操作数的类型是 `int` 或 `long`, 则右移运算符将执行算术移位: 左侧操作数的最高有效位(符号位)的值将传播到高顺序空位位置。也就是说, 如果左侧操作数为非负, 高顺序空位位置设置为零, 如果为负, 则将该位置设置为1。

- 如果左侧操作数的类型是 `uint` 或 `ulong`，则右移运算符执行逻辑移位：高顺序空位位置始终设置为零。

```
uint c = 0b_1000_0000_0000_0000_0000_0000_0000;
Console.WriteLine($"Before: {Convert.ToString(c, toBase: 2), 32}");

uint d = c >> 3;
Console.WriteLine($"After: {Convert.ToString(d, toBase: 2), 32}");
// Output:
// Before: 10000000000000000000000000000000
// After:    10000000000000000000000000000000
```

有关 [>>](#) 运算符的右侧操作数如何定义移位计数的信息，请参阅[移位运算符的移位计数部分](#)。

逻辑 AND 运算符 &

& 运算符计算其整型操作数的位逻辑 AND:

```
uint a = 0b_1111_1000;
uint b = 0b_1001_1101;
uint c = a & b;
Console.WriteLine(Convert.ToString(c, toBase: 2));
// Output:
// 10011000
```

对于 `bool` 操作数, `&` 运算符对其操作数执行逻辑 AND 运算。一元 `&` 运算符是 [address-of 运算符](#)。

逻辑异或运算符 ^

`^` 运算符计算其整型操作数的位逻辑异或, 也称为位逻辑 XOR:

```
uint a = 0b_1111_1000;
uint b = 0b_0001_1100;
uint c = a ^ b;
Console.WriteLine(Convert.ToString(c, toBase: 2));
// Output:
// 11100100
```

对于 `bool` 操作数, `^` 运算符对其操作数执行逻辑异或运算。

逻辑或运算符 |

`|` 运算符计算其整型操作数的位逻辑 OR:

```
uint a = 0b_1010_0000;
uint b = 0b_1001_0001;
uint c = a | b;
Console.WriteLine(Convert.ToString(c, toBase: 2));
// Output:
// 10110001
```

对于 `bool` 操作数, `|` 运算符对其操作数执行逻辑 OR 运算。

复合赋值

对于二元运算符 `op`, 窗体的复合赋值表达式

```
x op= y
```

等效于

```
x = x op y
```

不同的是 `x` 只计算一次。

以下示例演示了使用位运算符和移位运算符的复合赋值的用法:

```

uint a = 0b_1111_1000;
a &= 0b_1001_1101;
Display(a); // output: 10011000

a |= 0b_0011_0001;
Display(a); // output: 10111001

a ^= 0b_1000_0000;
Display(a); // output: 111001

a <<= 2;
Display(a); // output: 11100100

a >>= 4;
Display(a); // output: 1110

void Display(uint x) => Console.WriteLine($"{Convert.ToString(x, toBase: 2), 8}");

```

由于**数值提升**, `op` 运算的结果可能不会隐式转换为 `x` 的 `T` 类型。在这种情况下, 如果 `op` 是预定义的运算符并且运算的结果可以显式转换为 `x` 的类型 `T`, 则形式为 `x op= y` 的复合赋值表达式等效于

`x = (T)(x op y)`, 但 `x` 仅计算一次。以下示例演示了该行为:

```

byte x = 0b_1111_0001;

int b = x << 8;
Console.WriteLine($"{Convert.ToString(b, toBase: 2)}"); // output: 1111000100000000

x <<= 8;
Console.WriteLine(x); // output: 0

```

运算符优先级

以下列表按位运算符和移位运算符从最高优先级到最低优先级排序:

- 按位求补运算符 `~`
- 移位运算符 `<<` 和 `>>`
- 逻辑与运算符 `&`
- 逻辑异或运算符 `^`
- 逻辑或运算符 `|`

使用括号 `()` 可以更改运算符优先级决定的计算顺序:

```

uint a = 0b_1101;
uint b = 0b_1001;
uint c = 0b_1010;

uint d1 = a | b & c;
Display(d1); // output: 1101

uint d2 = (a | b) & c;
Display(d2); // output: 1000

void Display(uint x) => Console.WriteLine($"{Convert.ToString(x, toBase: 2), 4}");

```

要了解按优先级排序的完整 C# 运算符列表, 请参阅 [C# 运算符](#)一文中的**运算符优先级**部分。

移位运算符的移位计数

对于移位运算符 `<<` 和 `>>`，右侧操作数的类型必须为 `int`，或具有预定义隐式数值转换为 `int` 的类型。

对于 `x << count` 和 `x >> count` 表达式，实际移位计数取决于 `x` 的类型，如下所示：

- 如果 `x` 的类型为 `int` 或 `uint`，则移位计数由右侧操作数的低阶五位定义。也就是说，移位计数通过 `count & 0x1F`（或 `count & 0b_1_1111`）计算得出。
- 如果 `x` 的类型为 `long` 或 `ulong`，则移位计数由右侧操作数的低阶六位定义。也就是说，移位计数通过 `count & 0x3F`（或 `count & 0b_11_1111`）计算得出。

以下示例演示了该行为：

```
int count1 = 0b_0000_0001;
int count2 = 0b_1110_0001;

int a = 0b_0001;
Console.WriteLine($"{a} << {count1} is {a << count1}; {a} << {count2} is {a << count2}");
// Output:
// 1 << 1 is 2; 1 << 225 is 2

int b = 0b_0100;
Console.WriteLine($"{b} >> {count1} is {b >> count1}; {b} >> {count2} is {b >> count2}");
// Output:
// 4 >> 1 is 2; 4 >> 225 is 2
```

NOTE

如前例所示，即使右侧操作符的值大于左侧操作符中的位数，移位运算的结果也不可为零。

枚举逻辑运算符

所有枚举类型还支持 `~`、`&`、`|` 和 `^` 运算符。对于相同枚举类型的操作数，对底层整数类型的相应值执行逻辑运算。例如，对于具有底层类型 `U` 的枚举类型 `T` 的任何 `x` 和 `y`，`x & y` 表达式生成与 `(T)((U)x & (U)y)` 表达式相同的结果。

通常使用具有枚举类型的位逻辑运算符，该枚举类型使用 `Flags` 特性定义。有关详细信息，请参阅枚举类型一文的 [作为位标记的枚举类型](#) 部分。

运算符可重载性

用户定义的类型可以重载 `~`、`<<`、`>>`、`&`、`|` 和 `^` 运算符。重载二元运算符时，对应的复合赋值运算符也会隐式重载。用户定义类型不能显式重载复合赋值运算符。

如果用户定义类型 `T` 重载 `<<` 或 `>>` 运算符，则左侧操作数的类型必须为 `T` 且右侧操作数的类型必须为 `int`。

C# 语言规范

有关更多信息，请参阅 [C# 语言规范](#) 的以下部分：

- [按位求补运算符](#)
- [移位运算符](#)
- [逻辑运算符](#)
- [复合赋值](#)
- [数值提升](#)

另请参阅

- [C# 参考](#)
- [C# 运算符和表达式](#)
- [Boolean 逻辑运算符](#)

相等运算符 (C# 参考)

2021/5/10 • [Edit Online](#)

`==` (相等) 和 `!=` (不等) 运算符检查其操作数是否相等。

相等运算符 `==`

如果操作数相等, 等于运算符 `==` 返回 `true`, 否则返回 `false`。

值类型的相等性

如果内置值类型的值相等, 则其操作数相等:

```
int a = 1 + 2 + 3;
int b = 6;
Console.WriteLine(a == b); // output: True

char c1 = 'a';
char c2 = 'A';
Console.WriteLine(c1 == c2); // output: False
Console.WriteLine(c1 == char.ToLower(c2)); // output: True
```

NOTE

对于 `==`、`<`、`>`、`<=` 和 `>=` 运算符, 如果任何操作数不是数字(`Double.NaN` 或 `Single.NaN`), 则运算的结果为 `false`。这意味着 `NaN` 值不大于、小于或等于任何其他 `double` (或 `float`) 值, 包括 `NaN`。有关更多信息和示例, 请参阅 [Double.NaN](#) 或 [Single.NaN](#) 参考文章。

如果基本整数类型的相应值相等, 则相同枚举类型的两个操作数相等。

用户定义的 `struct` 类型默认情况下不支持 `==` 运算符。要支持 `==` 运算符, 用户定义的结构必须重载它。

从 C# 7.3 开始, `==` 和 `!=` 运算符由 C# 元组支持。有关详细信息, 请参阅[元组类型](#)一文的[元组相等](#)部分。

引用类型的相等性

默认情况下, 如果两个非记录引用类型操作符引用同一对象, 则这两个操作符相等:

```
public class ReferenceTypesEquality
{
    public class MyClass
    {
        private int id;

        public MyClass(int id) => this.id = id;
    }

    public static void Main()
    {
        var a = new MyClass(1);
        var b = new MyClass(1);
        var c = a;
        Console.WriteLine(a == b); // output: False
        Console.WriteLine(a == c); // output: True
    }
}
```

如示例所示，**默认情况下**，用户定义的引用类型支持 `==` 运算符。但是，引用类型可重载 `==` 运算符。如果引用类型重载 `==` 运算符，使用 `Object.ReferenceEquals` 方法来检查该类型的两个引用是否引用同一对象。

记录类型相等性

在 C# 9.0 和更高版本中提供，**记录类型**支持 `==` 和 `!=` 运算符，这些运算符默认提供值相等性语义。也就是说，当两个记录操作数均为 `null` 或所有字段的对应值和自动实现的属性相等时，两个记录操作数都相等。

```
public class RecordTypesEquality
{
    public record Point(int X, int Y, string Name);
    public record TaggedNumber(int Number, List<string> Tags);

    public static void Main()
    {
        var p1 = new Point(2, 3, "A");
        var p2 = new Point(1, 3, "B");
        var p3 = new Point(2, 3, "A");

        Console.WriteLine(p1 == p2); // output: False
        Console.WriteLine(p1 == p3); // output: True

        var n1 = new TaggedNumber(2, new List<string>() { "A" });
        var n2 = new TaggedNumber(2, new List<string>() { "A" });
        Console.WriteLine(n1 == n2); // output: False
    }
}
```

如前面的示例所示，对于非记录引用类型成员，比较的是它们的引用值，而不是所引用的实例。

字符串相等性

如果两个字符串均为 `null` 或者两个字符串实例具有相等长度且在每个字符位置有相同字符，则这两个**字符串**操作数相等：

```
string s1 = "hello!";
string s2 = "HeLLo!";
Console.WriteLine(s1 == s2.ToLower()); // output: True

string s3 = "Hello!";
Console.WriteLine(s1 == s3); // output: False
```

这就是区分大小写的序号比较。有关字符串比较的详细信息，请参阅[如何在 C# 中比较字符串](#)。

委托相等

若两个运行时间类型相同的**委托**操作数均为 `null`，或其调用列表长度系统且在每个位置具有相同的条目，则二者相等：

```
Action a = () => Console.WriteLine("a");

Action b = a + a;
Action c = a + a;
Console.WriteLine(object.ReferenceEquals(b, c)); // output: False
Console.WriteLine(b == c); // output: True
```

有关详细信息，请参阅[C# 语言规范](#)中的**委托相等运算符**部分。

通过计算语义上相同的 **Lambda 表达式**生成的委托不相等，如以下示例所示：

```
Action a = () => Console.WriteLine("a");
Action b = () => Console.WriteLine("a");

Console.WriteLine(a == b); // output: False
Console.WriteLine(a + b == a + b); // output: True
Console.WriteLine(b + a == a + b); // output: False
```

不等运算符 !=

如果操作数不相等，不等于运算符 `!=` 返回 `true`，否则返回 `false`。对于内置类型的操作数，表达式 `x != y` 生成与表达式 `!(x == y)` 相同的结果。有关类型相等性的更多信息，请参阅[相等运算符](#)部分。

下面的示例演示 `!=` 运算符的用法：

```
int a = 1 + 1 + 2 + 3;
int b = 6;
Console.WriteLine(a != b); // output: True

string s1 = "Hello";
string s2 = "Hello";
Console.WriteLine(s1 != s2); // output: False

object o1 = 1;
object o2 = 1;
Console.WriteLine(o1 != o2); // output: True
```

运算符可重载性

用户定义类型可以重载 `==` 和 `!=` 运算符。如果某类型重载这两个运算符之一，它还必须重载另一个运算符。

记录类型不能显式重载 `==` 和 `!=` 运算符。如果需要更改记录类型 `T` 的 `==` 和 `!=` 运算符的行为，请使用以下签名实现 `IEquatable<T>.Equals` 方法：

```
public virtual bool Equals(T? other);
```

C# 语言规范

有关详细信息，请参阅[C# 语言规范](#)中的[关系和类型测试运算符](#)部分。

有关记录类型相等性的详细信息，请参阅[记录功能建议附注](#)中的[相等性成员](#)部分。

请参阅

- [C# 参考](#)
- [C# 运算符和表达式](#)
- [System.IEquatable<T>](#)
- [Object.Equals](#)
- [Object.ReferenceEquals](#)
- [相等性比较](#)
- [比较运算符](#)

比较运算符 (C# 参考)

2020/11/2 • [Edit Online](#)

< (小于)、> (大于)、<= (小于或等于) 和 >= (大于或等于) 比较(也称为关系)运算符比较其操作数。所有整型和浮点数值类型都支持这些运算符。

NOTE

对于 ==、<、>、<= 和 >= 运算符，如果所有操作数都不是数字 (Double.NaN 或 Single.NaN)，则运算结果为 false。这意味着 NaN 值不大于、小于或等于任何其他 double (或 float) 值，包括 NaN。有关更多信息和示例，请参阅 Double.NaN 或 Single.NaN 参考文章。

char 类型也支持比较运算符。在使用 char 操作数时，将比较对应的字符代码。

枚举类型也支持比较运算符。对于相同枚举类型的操作数，基础整数类型的相应值会进行比较。

== 和 != 运算符检查其操作数是否相等。

小于运算符 <

如果左侧操作数小于右侧操作数，< 运算符返回 true，否则返回 false：

```
Console.WriteLine(7.0 < 5.1); // output: False
Console.WriteLine(5.1 < 5.1); // output: False
Console.WriteLine(0.0 < 5.1); // output: True

Console.WriteLine(double.NaN < 5.1); // output: False
Console.WriteLine(double.NaN >= 5.1); // output: False
```

大于运算符 >

如果左侧操作数大于右侧操作数，> 运算符返回 true，否则返回 false：

```
Console.WriteLine(7.0 > 5.1); // output: True
Console.WriteLine(5.1 > 5.1); // output: False
Console.WriteLine(0.0 > 5.1); // output: False

Console.WriteLine(double.NaN > 5.1); // output: False
Console.WriteLine(double.NaN <= 5.1); // output: False
```

小于或等于运算符 <=

如果左侧操作数小于或等于右侧操作数，<= 运算符返回 true，否则返回 false：

```
Console.WriteLine(7.0 <= 5.1); // output: False
Console.WriteLine(5.1 <= 5.1); // output: True
Console.WriteLine(0.0 <= 5.1); // output: True

Console.WriteLine(double.NaN > 5.1); // output: False
Console.WriteLine(double.NaN <= 5.1); // output: False
```

大于或等于运算符 `>=`

如果左侧操作数大于或等于右侧操作数, `>=` 运算符返回 `true`, 否则返回 `false`:

```
Console.WriteLine(7.0 >= 5.1);    // output: True
Console.WriteLine(5.1 >= 5.1);    // output: True
Console.WriteLine(0.0 >= 5.1);    // output: False

Console.WriteLine(double.NaN < 5.1);    // output: False
Console.WriteLine(double.NaN >= 5.1);    // output: False
```

运算符可重载性

用户定义类型可以重载 `<`、`>`、`<=` 和 `>=` 运算符。

如果某类型重载 `<` 或 `>` 运算符之一, 它必须同时重载 `<` 和 `>`。如果某类型重载 `<=` 或 `>=` 运算符之一, 它必须同时重载 `<=` 和 `>=`。

C# 语言规范

有关详细信息, 请参阅 [C# 语言规范](#) 中的[关系和类型测试运算符](#)部分。

请参阅

- [C# 参考](#)
- [C# 运算符和表达式](#)
- [System.IComparable<T>](#)
- [相等运算符](#)

成员访问运算符和表达式 (C# 参考)

2021/5/7 • [Edit Online](#)

访问类型成员时，可以使用以下运算符和表达式：

- `.` (成员访问)：用于访问命名空间或类型的成员
- `[]` (数组元素或索引器访问)：用于访问数组元素或类型索引器
- `?.` 和 `?[]` (null 条件运算符)：仅当操作数为非 null 时才用于执行成员或元素访问运算
- `()` (调用)：用于调用被访问的方法或调用委托
- `^` (从末尾开始索引)：指示元素位置来自序列的末尾
- `..` (范围)：指定可用于获取一系列序列元素的索引范围

成员访问表达式。

可以使用 `.` 标记来访问命名空间或类型的成员，如以下示例所示：

- 使用 `.` 访问命名空间内的嵌套命名空间，如以下 `using directive` 的示例所示：

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

- 使用 `.` 构成限定名称以访问命名空间中的类型，如下面的代码所示：

```
System.Collections.Generic.IEnumerable<int> numbers = new int[] { 1, 2, 3 };
```

使用 `using` 指令 来使用可选的限定名称。

- 使用 `.` 访问 **类型成员** (静态和非静态)，如下面的代码所示：

```
var constants = new List<double>();
constants.Add(Math.PI);
constants.Add(Math.E);
Console.WriteLine($"{constants.Count} values to show:");
Console.WriteLine(string.Join(", ", constants));
// Output:
// 2 values to show:
// 3.14159265358979, 2.71828182845905
```

还可以使用 `.` 访问 [扩展方法](#)。

索引器运算符 []

方括号 `[]` 通常用于数组、索引器或指针元素访问。

数组访问

下面的示例演示如何访问数组元素：

```

int[] fib = new int[10];
fib[0] = fib[1] = 1;
for (int i = 2; i < fib.Length; i++)
{
    fib[i] = fib[i - 1] + fib[i - 2];
}
Console.WriteLine(fib[fib.Length - 1]); // output: 55

double[,] matrix = new double[2,2];
matrix[0,0] = 1.0;
matrix[0,1] = 2.0;
matrix[1,0] = matrix[1,1] = 3.0;
var determinant = matrix[0,0] * matrix[1,1] - matrix[1,0] * matrix[0,1];
Console.WriteLine(determinant); // output: -3

```

如果数组索引超出数组相应维度的边界，将引发 [IndexOutOfRangeException](#)。

如上述示例所示，在声明数组类型或实例化数组实例时，还会使用方括号。

有关数组的详细信息，请参阅[数组](#)。

索引器访问

下面的示例使用 .NET [Dictionary< TKey, TValue >](#) 类型来演示索引器访问：

```

var dict = new Dictionary<string, double>();
dict["one"] = 1;
dict["pi"] = Math.PI;
Console.WriteLine(dict["one"] + dict["pi"]); // output: 4.14159265358979

```

使用索引器，可通过类似于编制数组索引的方式对用户定义类型的实例编制索引。与必须是整数的数组索引不同，可以将索引器参数声明为任何类型。

有关索引器的详细信息，请参阅[索引器](#)。

[] 的其他用法

要了解指针元素访问，请参阅[与指针相关的运算符](#)一文的[指针元素访问运算符 \[\]](#) 部分。

方括号还用于指定属性：

```

[System.Diagnostics.Conditional("DEBUG")]
void TraceMethod() {}

```

Null 条件运算符 ?. 和 ?[]

Null 条件运算符在 C# 6 及更高版本中可用，仅当操作数的计算结果为非 null 时，null 条件运算符才会将[成员访问](#) `?.` 或[元素访问](#) `?[]` 运算应用于其操作数；否则，将返回 `null`。即：

- 如果 `a` 的计算结果为 `null`，则 `a?.x` 或 `a?[]` 的结果为 `null`。
- 如果 `a` 的计算结果为非 `null`，则 `a?.x` 或 `a?[]` 的结果将分别与 `a.x` 或 `a[]` 的结果相同。

NOTE

如果 `a.x` 或 `a[]` 引发异常，则 `a?.x` 或 `a?[]` 将对非 `null` `a` 引发相同的异常。例如，如果 `a` 为非 `null` 数组实例且 `x` 在 `a` 的边界之外，则 `a?[]` 将引发 [IndexOutOfRangeException](#)。

NULL 条件运算符采用最小化求值策略。也就是说，如果条件成员或元素访问运算链中的一个运算返回 `null`，

则链的其余部分不会执行。在以下示例中，如果 `A` 的计算结果为 `null`，则不会计算 `B`；如果 `A` 或 `B` 的计算结果为 `null`，则不会计算 `C`：

```
A?.B?.Do(C);  
A?.B?[C];
```

以下示例演示了 `?.` 和 `?[]` 运算符的用法：

```
double SumNumbers(List<double[]> setsOfNumbers, int indexOfSetToSum)  
{  
    return setsOfNumbers?[indexOfSetToSum]?.Sum() ?? double.NaN;  
}  
  
var sum1 = SumNumbers(null, 0);  
Console.WriteLine(sum1); // output: NaN  
  
var numberSets = new List<double[]>  
{  
    new[] { 1.0, 2.0, 3.0 },  
    null  
};  
  
var sum2 = SumNumbers(numberSets, 0);  
Console.WriteLine(sum2); // output: 6  
  
var sum3 = SumNumbers(numberSets, 1);  
Console.WriteLine(sum3); // output: NaN
```

前面的示例还使用 [Null 合并运算符 `??`](#) 来指定替代表达式，以便在 `null` 条件运算的结果为 `null` 时用于计算。

如果 `a.x` 或 `a[x]` 是不可为 `null` 的值类型 `T`，则 `a?.x` 或 `a? [x]` 属于对应的[可为 `null` 的值类型 `T?`](#)。如果需要 `T` 类型的表达式，请将 Null 合并操作符 `??` 应用于 `null` 条件表达式，如下面的示例所示：

```
int GetSumOfFirstTwoOrDefault(int[] numbers)  
{  
    if ((numbers?.Length ?? 0) < 2)  
    {  
        return 0;  
    }  
    return numbers[0] + numbers[1];  
}  
  
Console.WriteLine(GetSumOfFirstTwoOrDefault(null)); // output: 0  
Console.WriteLine(GetSumOfFirstTwoOrDefault(new int[0])); // output: 0  
Console.WriteLine(GetSumOfFirstTwoOrDefault(new[] { 3, 4, 5 })); // output: 7
```

在前面的示例中，如果不使用 `??` 运算符，则在 `numbers` 为 `null` 时，`numbers?.Length < 2` 的计算结果为 `false`。

Null 条件成员访问运算符 `?.` 也称为 Elvis 运算符。

线程安全的委托调用

使用 `?.` 运算符来检查委托是否非 `null` 并以线程安全的方式调用它（例如，[引发事件](#)时），如下面的代码所示：

```
PropertyChanged?.Invoke(...)
```

该代码等同于将在 C# 5 或更低版本中使用的以下代码：

```
var handler = this.PropertyChanged;
if (handler != null)
{
    handler(...);
}
```

这是一种线程安全方法，可确保只调用非 null `handler`。由于委托实例是不可变的，因此，任何线程都不能更改 `handler` 本地变量所引用的对象。具体而言，如果另一个线程执行的代码从 `PropertyChanged` 事件中取消订阅，并且 `PropertyChanged` 在调用 `handler` 之前变为 `null`，则 `handler` 引用的对象不受影响。`?.` 运算符对其左操作数的计算不超过一次，从而确保在验证为非 `null` 后，不能将其更改为 `null`。

调用表达式 ()

使用括号 `()` 调用方法或调用委托。

以下示例演示如何在使用或不使用参数的情况下调用方法，以及调用委托：

```
Action<int> display = s => Console.WriteLine(s);

var numbers = new List<int>();
numbers.Add(10);
numbers.Add(17);
display(numbers.Count); // output: 2

numbers.Clear();
display(numbers.Count); // output: 0
```

在调用带 `new` 运算符的构造函数时，还可以使用括号。

0 的其他用法

此外可以使用括号来调整表达式中计算操作的顺序。有关详细信息，请参阅 [C# 运算符](#)。

[强制转换表达式](#)，其执行显式类型转换，也可以使用括号。

从末尾运算符 ^ 开始索引

`^` 运算符在 C# 8.0 和更高版本中提供，指示序列末尾的元素位置。对于长度为 `length` 的序列，`^n` 指向与序列开头偏移 `length - n` 的元素。例如，`^1` 指向序列的最后一个元素，`^length` 指向序列的第一个元素。

```
int[] xs = new[] { 0, 10, 20, 30, 40 };
int last = xs[^1];
Console.WriteLine(last); // output: 40

var lines = new List<string> { "one", "two", "three", "four" };
string prelast = lines[^2];
Console.WriteLine(prelast); // output: three

string word = "Twenty";
Index toFirst = ^word.Length;
char first = word[toFirst];
Console.WriteLine(first); // output: T
```

如前面的示例所示，表达式 `^e` 属于 `System.Index` 类型。在表达式 `^e` 中，`e` 的结果必须隐式转换为 `int`。

还可以将 `^` 运算符与 [范围运算符](#)一起使用以创建一个索引范围。有关详细信息，请参阅[索引和范围](#)。

范围运算符 .

.. 运算符在 C# 8.0 和更高版本中提供，指定索引范围的开头和末尾作为其操作数。左侧操作数是范围的包含性开头。右侧操作数是范围的包含性末尾。任一操作数都可以是序列开头或末尾的索引，如以下示例所示：

```
int[] numbers = new[] { 0, 10, 20, 30, 40, 50 };
int start = 1;
int amountToTake = 3;
int[] subset = numbers[start..(start + amountToTake)];
Display(subset); // output: 10 20 30

int margin = 1;
int[] inner = numbers[margin..^margin];
Display(inner); // output: 10 20 30 40

string line = "one two three";
int amountToTakeFromEnd = 5;
Range endIndices = ^amountToTakeFromEnd..^0;
string end = line[endIndices];
Console.WriteLine(end); // output: three

void Display<T>(IEnumerable<T> xs) => Console.WriteLine(string.Join(" ", xs));
```

如前面的示例所示，表达式 `a..b` 属于 `System.Range` 类型。在表达式 `a..b` 中，`a` 和 `b` 的结果必须隐式转换为 `int` 或 `Index`。

可以省略 `..` 运算符的任何操作数来获取无限制范围：

- `a..` 等效于 `a..^0`
- `..b` 等效于 `0..b`
- `..` 等效于 `0..^0`

```
int[] numbers = new[] { 0, 10, 20, 30, 40, 50 };
int amountToDrop = numbers.Length / 2;

int[] rightHalf = numbers[amountToDrop..];
Display(rightHalf); // output: 30 40 50

int[] leftHalf = numbers[..^amountToDrop];
Display(leftHalf); // output: 0 10 20

int[] all = numbers[..];
Display(all); // output: 0 10 20 30 40 50

void Display<T>(IEnumerable<T> xs) => Console.WriteLine(string.Join(" ", xs));
```

有关详细信息，请参阅[索引和范围](#)。

运算符可重载性

`.`、`()`、`^` 和 `..` 运算符无法进行重载。`[]` 运算符也被视为非可重载运算符。使用[索引器](#)以支持对用户定义的类型编制索引。

C# 语言规范

有关更多信息，请参阅[C# 语言规范](#)的以下部分：

- 成员访问
- 元素访问
- Null 条件运算符
- 调用表达式

有关索引和范围的详细信息，请参阅[功能建议说明](#)。

请参阅

- [C# 参考](#)
- [C# 运算符和表达式](#)
- [??\(空合运算符\)](#)
- [:: 运算符](#)

类型测试运算符和强制转换表达式 (C# 引用)

2021/5/10 • [Edit Online](#)

可以使用以下运算符和表达式来执行类型检查或类型转换：

- **is 运算符**: 用于检查表达式的运行时类型是否与给定类型兼容
- **as 运算符**: 用于将表达式显式转换为给定类型(如果其运行时类型与该类型兼容)
- **强制转换表达式**: 执行显式转换
- **typeof 运算符**: 用于获取某个类型的 `System.Type` 实例

is 运算符

`is` 运算符检查表达式结果的运行时类型是否与给定类型兼容。从 C# 7.0 开始，`is` 运算符还会对照某个模式测试表达式结果。

具有类型测试 `is` 运算符的表达式具有以下形式

```
E is T
```

其中 `E` 是返回一个值的表达式，`T` 是类型或类型参数的名称。`E` 不得为匿名方法或 Lambda 表达式。

如果表达式结果为非 `null` 并且满足以下任一条件，则 `is` 运算符将返回 `true`：

- 表达式结果的运行时类型为 `T`。
- 表达式结果的运行时类型派生自类型 `T`、实现接口 `T`，或者存在从其到 `T` 的另一种 **隐式引用转换**。
- 表达式结果的运行时类型是基础类型为 `T` 且 `Nullable<T>.HasValue` 为 `true` 的 **可为空值类型**。
- 存在从表达式结果的运行时类型到类型 `T` 的 **装箱**或**取消装箱**转换。

`is` 运算符不会考虑用户定义的转换。

以下示例演示，如果表达式结果的运行时类型派生自给定类型，即类型之间存在引用转换，`is` 运算符将返回 `true`：

```
public class Base { }

public class Derived : Base { }

public static class IsOperatorExample
{
    public static void Main()
    {
        object b = new Base();
        Console.WriteLine(b is Base); // output: True
        Console.WriteLine(b is Derived); // output: False

        object d = new Derived();
        Console.WriteLine(d is Base); // output: True
        Console.WriteLine(d is Derived); // output: True
    }
}
```

以下示例演示，`is` 运算符将考虑装箱和取消装箱转换，但不会考虑**数值转换**：

```
int i = 27;
Console.WriteLine(i is System.IFormattable); // output: True

object iBoxed = i;
Console.WriteLine(iBoxed is int); // output: True
Console.WriteLine(iBoxed is long); // output: False
```

有关 C# 转换的信息，请参阅 [C# 语言规范的转换](#)一章。

有模式匹配的类型测试

从 C# 7.0 开始，`is` 运算符还会对照某个模式测试表达式结果。下面的示例演示如何使用 [声明模式](#) 来检查表达式的运行时类型：

```
int i = 23;
object iBoxed = i;
int? jNullable = 7;
if (iBoxed is int a && jNullable is int b)
{
    Console.WriteLine(a + b); // output 30
}
```

若要了解受支持的模式，请参阅 [模式](#)。

as 运算符

`as` 运算符将表达式结果显式转换为给定的引用或可以为 `null` 值的类型。如果无法进行转换，则 `as` 运算符返回 `null`。与 [强制转换表达式](#) 不同，`as` 运算符永远不会引发异常。

形式如下的表达式

```
E as T
```

其中，`E` 为返回值的表达式，`T` 为类型或类型参数的名称，生成相同的结果。

```
E is T ? (T)(E) : (T)null
```

不同的是 `E` 只计算一次。

`as` 运算符仅考虑引用、可以为 `null`、装箱和取消装箱转换。不能使用 `as` 运算符执行用户定义的转换。为此，请使用 [强制转换表达式](#)。

下面的示例演示 `as` 运算符的用法：

```
IEnumerable<int> numbers = new[] { 10, 20, 30 };
IList<int> indexable = numbers as IList<int>;
if (indexable != null)
{
    Console.WriteLine(indexable[0] + indexable[indexable.Count - 1]); // output: 40
}
```

NOTE

如之前的示例所示，你需要将 `as` 表达式的结果与 `null` 进行比较，以检查转换是否成功。从 C# 7.0 开始，你可以使用 [is 运算符](#) 测试转换是否成功，如果成功，则将其结果分配给新变量。

强制转换表达式

形式为 `(T)E` 的强制转换表达式将表达式 `E` 的结果显式转换为类型 `T`。如果不存在从类型 `E` 到类型 `T` 的显式转换，则发生编译时错误。在运行时，显式转换可能不会成功，强制转换表达式可能会引发异常。

下面的示例演示显式数值和引用转换：

```
double x = 1234.7;
int a = (int)x;
Console.WriteLine(a); // output: 1234

IEnumerable<int> numbers = new int[] { 10, 20, 30 };
IList<int> list = (IList<int>)numbers;
Console.WriteLine(list.Count); // output: 3
Console.WriteLine(list[1]); // output: 20
```

有关支持的显式转换的信息，请参阅 [C# 语言规范的显式转换](#) 部分。有关如何定义自定义显式或隐式类型转换的信息，请参阅 [用户定义转换运算符](#)。

0 的其他用法

你还可以使用括号调用方法或调用委托。

括号的其他用法是调整表达式中计算操作的顺序。有关详细信息，请参阅 [C# 运算符](#)。

typeof 运算符

`typeof` 运算符用于获取某个类型的 `System.Type` 实例。`typeof` 运算符的实参必须是类型或类型形参的名称，如以下示例所示：

```
void PrintType<T>() => Console.WriteLine(typeof(T));

Console.WriteLine(typeof(List<string>));
PrintType<int>();
PrintType<System.Int32>();
PrintType<Dictionary<int, char>>();
// Output:
// System.Collections.Generic.List`1[System.String]
// System.Int32
// System.Int32
// System.Collections.Generic.Dictionary`2[System.Int32,System.Char]
```

你还可以使用具有未绑定泛型类型的 `typeof` 运算符。未绑定泛型类型的名称必须包含适当数量的逗号，且此数量小于类型参数的数量。以下示例演示了具有未绑定泛型类型的 `typeof` 运算符的用法：

```
Console.WriteLine(typeof(Dictionary<, >));
// Output:
// System.Collections.Generic.Dictionary`2[TKey, TValue]
```

表达式不能为 `typeof` 运算符的参数。若要获取表达式结果的运行时类型的 `System.Type` 实例，请使用 `Object.GetType` 方法。

使用 `typeof` 运算符进行类型测试

使用 `typeof` 运算符来检查表达式结果的运行时类型是否与给定的类型完全匹配。以下示例演示了使用 `typeof` 运算符和 `is` 运算符执行的类型检查之间的差异：

```
public class Animal { }

public class Giraffe : Animal { }

public static class TypeOfExample
{
    public static void Main()
    {
        object b = new Giraffe();
        Console.WriteLine(b is Animal); // output: True
        Console.WriteLine(b.GetType() == typeof(Animal)); // output: False

        Console.WriteLine(b is Giraffe); // output: True
        Console.WriteLine(b.GetType() == typeof(Giraffe)); // output: True
    }
}
```

运算符可重载性

`is`、`as` 和 `typeof` 运算符无法进行重载。

用户定义的类型不能重载 `()` 运算符，但可以定义可由强制转换表达式执行的自定义类型转换。有关详细信息，请参阅[用户定义转换运算符](#)。

C# 语言规范

有关更多信息，请参阅[C# 语言规范](#)的以下部分：

- [is 运算符](#)
- [as 运算符](#)
- [强制转换表达式](#)
- [typeof 运算符](#)

请参阅

- [C# 参考](#)
- [C# 运算符和表达式](#)
- [如何使用模式匹配以及 is 和 as 运算符安全地进行强制转换](#)
- [.NET 中的泛型](#)

用户定义转换运算符 (C# 引用)

2020/11/2 • [Edit Online](#)

用户定义类型可以定义从或到另一个类型的自定义隐式或显式转换。

隐式转换无需调用特殊语法，并且可以在各种情况(例如，在赋值和方法调用中)下发生。预定义的 C# 隐式转换始终成功，且永远不会引发异常。用户定义隐式转换也应如此。如果自定义转换可能会引发异常或丢失信息，请将其定义为显式转换。

`is` 和 `as` 运算符不考虑使用用户定义转换。[强制转换表达式](#)用于调用用户定义显式转换。

`operator` 和 `implicit` 或 `explicit` 关键字分别用于定义隐式转换或显式转换。定义转换的类型必须是该转换的源类型或目标类型。可用两种类型中的任何一种类型来定义两种用户定义类型之间的转换。

下面的示例展示如何定义隐式转换和显式转换：

```
using System;

public readonly struct Digit
{
    private readonly byte digit;

    public Digit(byte digit)
    {
        if (digit > 9)
        {
            throw new ArgumentOutOfRangeException(nameof(digit), "Digit cannot be greater than nine.");
        }
        this.digit = digit;
    }

    public static implicit operator byte(Digit d) => d.digit;
    public static explicit operator Digit(byte b) => new Digit(b);

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

public static class UserDefinedConversions
{
    public static void Main()
    {
        var d = new Digit(7);

        byte number = d;
        Console.WriteLine(number); // output: 7

        Digit digit = (Digit)number;
        Console.WriteLine(digit); // output: 7
    }
}
```

`operator` 关键字也可用于重载预定义的 C# 运算符。有关详细信息，请参阅[运算符重载](#)。

C# 语言规范

有关更多信息，请参阅[C# 语言规范](#)的以下部分：

- [转换运算符](#)

- [用户定义的转换](#)

- [隐式转换](#)

- [显式转换](#)

请参阅

- [C# 参考](#)

- [C# 运算符和表达式](#)

- [运算符重载](#)

- [类型测试和强制转换运算符](#)

- [强制转换和类型转换](#)

- [设计准则 - 转换运算符](#)

- [Chained user-defined explicit conversions in C# \(C# 中链接在一起的用户定义的显式转换\)](#)

指针相关运算符 (C# 参考)

2021/5/7 • [Edit Online](#)

可以使用以下运算符来使用指针：

- 一元 `&` ([address-of](#)) 运算符：用于获取变量的地址
- 一元 `*` ([指针间接](#)) 运算符：用于获取指针指向的变量
- `->` ([成员访问](#)) 和 `[]` ([元素访问](#)) 运算符
- 算术运算符 `+`、`-`、`++` 和 `--`
- 比较运算符 `==`、`!=`、`<`、`>`、`<=` 和 `>=`

有关指针类型的信息，请参阅[指针类型](#)。

NOTE

任何带指针的运算都需要使用 `unsafe` 上下文。必须使用 [AllowUnsafeBlocks](#) 编译器选项来编译包含不安全块的代码。

Address-of 运算符 &

一元 `&` 运算符返回其操作数的地址：

```
unsafe
{
    int number = 27;
    int* pointerToNumber = &number;

    Console.WriteLine($"Value of the variable: {number}");
    Console.WriteLine($"Address of the variable: {(long)pointerToNumber:X}");
}
```

`&` 运算符的操作数必须是固定变量。固定变量是驻留在不受[垃圾回收器](#)操作影响的存储位置的变量。在上述示例中，局部变量 `number` 是固定变量，因为它驻留在堆栈上。驻留在可能受垃圾回收器影响的存储位置的变量（如重定位）称为可移动变量。对象字段和数组元素是可移动变量的示例。如果使用 [fixed 语句](#)“固定”，则可以获取可移动变量的地址。获取的地址仅在 `fixed` 语句块中有效。以下示例显示如何使用 `fixed` 语句和 `&` 运算符：

```
unsafe
{
    byte[] bytes = { 1, 2, 3 };
    fixed (byte* pointerToFirst = &bytes[0])
    {
        // The address stored in pointerToFirst
        // is valid only inside this fixed statement block.
    }
}
```

你也无法获取常量或值的地址。

有关固定变量和可移动变量的详细信息，请参阅[C# 语言规范的固定变量和可移动变量部分](#)。

二进制 `&` 运算符计算其布尔操作数的逻辑 AND 或其整型操作数的位逻辑 AND。

指针间接运算符 *

一元指针间接运算符 `*` 获取其操作数指向的变量。它也称为取消引用运算符。`*` 运算符的操作数必须是指针类型。

```
unsafe
{
    char letter = 'A';
    char* pointerToLetter = &letter;
    Console.WriteLine($"Value of the `letter` variable: {letter}");
    Console.WriteLine($"Address of the `letter` variable: {(long)pointerToLetter:X}");

    *pointerToLetter = 'Z';
    Console.WriteLine($"Value of the `letter` variable after update: {letter}");
}
// Output is similar to:
// Value of the `letter` variable: A
// Address of the `letter` variable: DCB977DDF4
// Value of the `letter` variable after update: Z
```

不能将 `*` 运算符应用于类型 `void*` 的表达式。

二进制 `*` 运算符计算其数值操作数的乘积。

指针成员访问运算符 ->

-> 运算符将指针间接和成员访问合并。也就是说，如果 `x` 是类型为 `T*` 的指针且 `y` 是类型 `T` 的可访问成员，则形式的表达式

```
x->y
```

等效于

```
(*x).y
```

下面的示例演示 `->` 运算符的用法：

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

public class PointerMemberAccessExample
{
    public static unsafe void Main()
    {
        Coords coords;
        Coords* p = &coords;
        p->X = 3;
        p->Y = 4;
        Console.WriteLine(p->ToString()); // output: (3, 4)
    }
}
```

不能将 `->` 运算符应用于类型 `void*` 的表达式。

指针元素访问运算符 []

对于指针类型的表达式 `p`, `p[n]` 形式的指针元素访问计算方式为 `*(p + n)`, 其中 `n` 必须是可隐式转换为 `int`、`uint`、`long` 或 `ulong` 的类型。有关带有指针的 `+` 运算符的行为的信息, 请参阅[向指针中增加或从指针中减少整数值](#)部分。

以下示例演示如何使用指针和 `[]` 运算符访问数组元素:

```
unsafe
{
    char* pointerToChars = stackalloc char[123];

    for (int i = 65; i < 123; i++)
    {
        pointerToChars[i] = (char)i;
    }

    Console.WriteLine("Uppercase letters: ");
    for (int i = 65; i < 91; i++)
    {
        Console.Write(pointerToChars[i]);
    }
}
// Output:
// Uppercase letters: ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

在前面的示例中, `stackalloc` 表达式 在堆栈上分配内存块。

NOTE

指针元素访问运算符不检查越界错误。

不能将 `[]` 用于带有类型为 `void*` 的表达式的指针元素访问。

还可以将 `[]` 运算符用于[数组元素或索引器访问](#)。

指针算术运算符

可以使用指针执行以下算术运算:

- 向指针增加或从指针中减少一个整数值
- 减少两个指针
- 增量或减量指针

无法使用类型为 `void*` 的指针执行这些操作。

有关带数值类型的受支持的算术运算的信息, 请参阅[算术运算符](#)。

向指针增加或从指针中减少整数值

对于类型为 `T*` 的指针 `p` 和可隐式转换为 `int`、`uint`、`long` 或 `ulong` 的类型的表达式 `n`, 加法和减法定义如下:

- `p + n` 和 `n + p` 表达式都生成 `T*` 类型的指针, 该指针是将 `n * sizeof(T)` 添加到 `p` 给出的地址得到的。
- `p - n` 表达式生成类型为 `T*` 的指针, 该指针是从 `p` 给出的地址中减去 `n * sizeof(T)` 得到的。

`sizeof` 运算符 获取类型的大小(以字节为单位)。

以下示例演示了 `+` 运算符与指针的用法：

```
unsafe
{
    const int Count = 3;
    int[] numbers = new int[Count] { 10, 20, 30 };
    fixed (int* pointerToFirst = &numbers[0])
    {
        int* pointerToLast = pointerToFirst + (Count - 1);

        Console.WriteLine($"Value {*pointerToFirst} at address {(long)pointerToFirst}");
        Console.WriteLine($"Value {*pointerToLast} at address {(long)pointerToLast}");
    }
}
// Output is similar to:
// Value 10 at address 1818345918136
// Value 30 at address 1818345918144
```

指针减法

对于类型为 `T*` 的两个指针 `p1` 和 `p2`，表达式 `p1 - p2` 生成 `p1` 和 `p2` 给出的地址之间的差除以 `sizeof(T)` 的值。结果的类型为 `long`。即 `p1 - p2` 计算公式为 `((long)(p1) - (long)(p2)) / sizeof(T)`。

以下示例演示了指针减法：

```
unsafe
{
    int* numbers = stackalloc int[] { 0, 1, 2, 3, 4, 5 };
    int* p1 = &numbers[1];
    int* p2 = &numbers[5];
    Console.WriteLine(p2 - p1); // output: 4
}
```

指针增量和减量

`++` 增量运算符将 1 添加到其指针操作数。`--` 减量运算符从其指针操作数中减去 1。

两种运算符都支持两种形式：后缀 (`p++` 和 `p--`) 和前缀 (`++p` 和 `--p`)。`p++` 和 `p--` 的结果是该运算之前 `p` 的值。`++p` 和 `--p` 的结果是该运算之后 `p` 的值。

以下示例演示了后缀和前缀增量运算符的行为：

```
unsafe
{
    int* numbers = stackalloc int[] { 0, 1, 2 };
    int* p1 = &numbers[0];
    int* p2 = p1;
    Console.WriteLine($"Before operation: p1 - {(long)p1}, p2 - {(long)p2}");
    Console.WriteLine($"Postfix increment of p1: {(long)(p1++)}");
    Console.WriteLine($"Prefix increment of p2: {(long)(++p2)}");
    Console.WriteLine($"After operation: p1 - {(long)p1}, p2 - {(long)p2}");
}
// Output is similar to
// Before operation: p1 - 816489946512, p2 - 816489946512
// Postfix increment of p1: 816489946516
// Prefix increment of p2: 816489946516
// After operation: p1 - 816489946516, p2 - 816489946516
```

指针比较运算符

可以使用 `==`、`!=`、`<`、`>`、`<=` 和 `>=` 运算符来比较任何指针类型（包括 `void*`）的操作数。这些运算符将两

个操作数给出的地址进行比较，就好像它们是无符号整数一样。

有关这些运算符对其他类型操作数的行为的信息，请参阅[等式运算符](#)和[比较运算符](#)一文。

运算符优先级

以下列表按优先级从高到低的顺序对指针相关运算符进行排序：

- 后缀增量 `x++` 和减量 `x--` 运算符以及 `->` 和 `[]` 运算符
- 前缀增量 `++x` 和减量 `--x` 运算符以及 `&` 和 `*` 运算符
- 加法 `+` 和 `-` 运算符
- 比较 `<`、`>`、`<=` 和 `>=` 运算符
- 等式 `=` 和 `!=` 运算符

使用括号 `()` 更改运算符优先级所施加的计算顺序。

如需了解按优先级排序的完整 C# 运算符列表，请参阅[C# 运算符](#)一文中的[运算符优先级](#)部分。

运算符可重载性

用户定义的类型不能重载与指针相关的运算符 `&`、`*`、`->` 和 `[]`。

C# 语言规范

有关更多信息，请参阅[C# 语言规范](#)的以下部分：

- [固定和可移动变量](#)
- [address-of 运算符](#)
- [指针间接](#)
- [指针成员访问](#)
- [指针元素访问](#)
- [指针算术](#)
- [指针增量和减量](#)
- [指针比较](#)

请参阅

- [C# 参考](#)
- [C# 运算符和表达式](#)
- [指针类型](#)
- [unsafe 关键字](#)
- [fixed 关键字](#)
- [stackalloc](#)
- [sizeof 运算符](#)

赋值运算符 (C# 参考)

2021/5/10 • [Edit Online](#)

赋值运算符 `=` 将其右操作数的值赋给变量、属性或由其左操作数给出的索引器元素。赋值表达式的结果是分配给左操作数的值。右操作数类型必须与左操作数类型相同，或可隐式转换为左操作数的类型。

赋值运算符 `=` 为右联运算符，即形式的表达式

```
a = b = c
```

计算结果为

```
a = (b = c)
```

以下示例演示使用局部变量、属性和索引器元素作为其左操作数的赋值运算符的用法：

```
var numbers = new List<double>() { 1.0, 2.0, 3.0 };

Console.WriteLine(numbers.Capacity);
numbers.Capacity = 100;
Console.WriteLine(numbers.Capacity);
// Output:
// 4
// 100

int newFirstElement;
double originalFirstElement = numbers[0];
newFirstElement = 5;
numbers[0] = newFirstElement;
Console.WriteLine(originalFirstElement);
Console.WriteLine(numbers[0]);
// Output:
// 1
// 5
```

ref 赋值运算符

从 C# 7.3 开始，可以使用 ref 赋值运算符 `= ref` 重新分配 `ref local` 或 `ref readonly local` 变量。下面的示例演示 `ref` 赋值运算符的用法：

```
void Display(double[] s) => Console.WriteLine(string.Join(" ", s));

double[] arr = { 0.0, 0.0, 0.0 };
Display(arr);

ref double arrayElement = ref arr[0];
arrayElement = 3.0;
Display(arr);

arrayElement = ref arr[arr.Length - 1];
arrayElement = 5.0;
Display(arr);
// Output:
// 0 0 0
// 3 0 0
// 3 0 5
```

对于 `ref` 赋值运算符，其两个操作数的类型必须相同。

复合赋值

对于二元运算符 `op`，窗体的复合赋值表达式

```
x op= y
```

等效于

```
x = x op y
```

不同的是 `x` 只计算一次。

[算术](#)、[布尔逻辑](#)以及[逻辑位和移位](#)运算符支持复合赋值。

Null 合并赋值

从 C# 8.0 开始，只有在左操作数计算为 `null` 时，才能使用 null 合并赋值运算符 `??=` 将其右操作数的值分配给左操作数。有关详细信息，请参阅 [?? 和 ??= 运算符](#) 一文。

运算符可重载性

用户定义类型不能重载赋值运算符。但是，用户定义类型可以定义到其他类型的隐式转换。这样，可以将用户定义类型的值分配给其他类型的变量、属性或索引器元素。有关详细信息，请参阅 [用户定义转换运算符](#)。

用户定义类型不能显式重载复合赋值运算符。但是，如果用户定义类型重载了二元运算符 `op`，则 `op=` 运算符（如果存在）也将被隐式重载。

C# 语言规范

有关详细信息，请参阅 [C# 语言规范](#) 中的 [分配运算符](#) 部分。

如需了解有关 `ref` 赋值运算符 `= ref` 的详细信息，请参阅 [功能建议说明](#)。

另请参阅

- [C# 参考](#)
- [C# 运算符和表达式](#)

- `ref` 关键字

Lambda 表达式 (C# 引用)

2021/5/10 • [Edit Online](#)

使用 Lambda 表达式来创建匿名函数。使用 [lambda 声明运算符 =>](#) 从其主体中分离 lambda 参数列表。

Lambda 表达式可采用以下任意一种形式：

- [表达式 lambda](#), 表达式为其主体:

```
(input-parameters) => expression
```

- [语句 lambda](#), 语句块作为其主体:

```
(input-parameters) => { <sequence-of-statements> }
```

若要创建 Lambda 表达式, 需要在 Lambda 运算符左侧指定输入参数(如果有), 然后在另一侧输入表达式或语句块。

任何 Lambda 表达式都可以转换为[委托](#)类型。Lambda 表达式可以转换的委托类型由其参数和返回值的类型定义。如果 lambda 表达式不返回值, 则可以将其转换为 [Action](#) 委托类型之一; 否则, 可将其转换为 [Func](#) 委托类型之一。例如, 有 2 个参数且不返回值的 Lambda 表达式可转换为 [Action<T1,T2>](#) 委托。有 1 个参数且不返回值的 Lambda 表达式可转换为 [Func<T,TResult>](#) 委托。以下示例中, lambda 表达式 `x => x * x` (指定名为 `x` 的参数并返回 `x` 平方值) 将分配给委托类型的变量:

```
Func<int, int> square = x => x * x;
Console.WriteLine(square(5));
// Output:
// 25
```

表达式 lambda 还可以转换为[表达式树](#)类型, 如下面的示例所示:

```
System.Linq.Expressions.Expression<Func<int, int>> e = x => x * x;
Console.WriteLine(e);
// Output:
// x => (x * x)
```

可在需要委托类型或表达式树的实例的任何代码中使用 lambda 表达式, 例如, 作为 [Task.Run\(Action\)](#) 方法的参数传递应在后台执行的代码。用 C# 编写 [LINQ](#) 时, 还可以使用 lambda 表达式, 如下例所示:

```
int[] numbers = { 2, 3, 4, 5 };
var squaredNumbers = numbers.Select(x => x * x);
Console.WriteLine(string.Join(" ", squaredNumbers));
// Output:
// 4 9 16 25
```

如果使用基于方法的语法在 [System.Linq.Enumerable](#) 类中(例如, 在 LINQ to Objects 和 LINQ to XML 中)调用 [Enumerable.Select](#) 方法, 则参数为委托类型 [System.Func<T,TResult>](#)。如果在 [System.Linq.Queryable](#) 类中(例如, 在 LINQ to SQL 中)调用 [Queryable.Select](#) 方法, 则参数类型为表达式树类型 [Expression<Func<TSource, TResult>>](#)。在这两种情况下, 都可以使用相同的 lambda 表达式来指定参数值。尽管通过 Lambda 创建的对象实际具有不同的类型, 但其使得 2 个 [Select](#) 调用看起来类似。

表达式 lambda

表达式位于 `=>` 运算符右侧的 lambda 表达式称为“表达式 lambda”。表达式 lambda 会返回表达式的结果，并采用以下基本形式：

```
(input-parameters) => expression
```

表达式 lambda 的主体可以包含方法调用。不过，若要创建在 .NET 公共语言运行时 (CLR) 的上下文之外（如在 SQL Server 中）计算的表达式树，则不得在 lambda 表达式中使用方法调用。在 .NET 公共语言运行时 (CLR) 上下文之外，方法将没有任何意义。

语句 lambda

语句 lambda 与表达式 lambda 类似，只是语句括在大括号中：

```
(input-parameters) => { <sequence-of-statements> }
```

语句 lambda 的主体可以包含任意数量的语句；但是，实际上通常不会多于两个或三个。

```
Action<string> greet = name =>
{
    string greeting = $"Hello {name}!";
    Console.WriteLine(greeting);
};
greet("World");
// Output:
// Hello World!
```

不能使用语句 lambda 创建表达式树。

lambda 表达式的输入参数

将 lambda 表达式的输入参数括在括号中。使用空括号指定零个输入参数：

```
Action line = () => Console.WriteLine();
```

如果 lambda 表达式只有一个输入参数，则括号是可选的：

```
Func<double, double> cube = x => x * x * x;
```

两个或更多输入参数使用逗号加以分隔：

```
Func<int, int, bool> testForEquality = (x, y) => x == y;
```

有时，编译器无法推断输入参数的类型。可以显式指定类型，如下面的示例所示：

```
Func<int, string, bool> isTooLong = (int x, string s) => s.Length > x;
```

输入参数类型必须全部为显式或全部为隐式；否则，便会生成 [CS0748](#) 编译器错误。

从 C# 9.0 开始，可以使用 [弃元](#) 指定 lambda 表达式中不使用的两个或更多输入参数：

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

使用 lambda 表达式[提供事件处理程序](#)时，lambda 弃元参数可能很有用。

NOTE

为了向后兼容，如果只有一个输入参数命名为 `_`，则在 lambda 表达式中，`_` 将被视为该参数的名称。

异步 lambda

通过使用 `async` 和 `await` 关键字，你可以轻松创建包含异步处理的 lambda 表达式和语句。例如，下面的 Windows 窗体示例包含一个调用和等待异步方法 `ExampleMethodAsync` 的事件处理程序。

```
public partial class Form1 : Form
{
    public Form1()
    {
        InitializeComponent();
        button1.Click += button1_Click;
    }

    private async void button1_Click(object sender, EventArgs e)
    {
        await ExampleMethodAsync();
        textBox1.Text += "\r\nControl returned to Click event handler.\n";
    }

    private async Task ExampleMethodAsync()
    {
        // The following line simulates a task-returning asynchronous process.
        await Task.Delay(1000);
    }
}
```

你可以使用异步 lambda 添加同一事件处理程序。若要添加此处理程序，请在 lambda 参数列表前添加 `async` 修饰符，如下面的示例所示：

```
public partial class Form1 : Form
{
    public Form1()
    {
        InitializeComponent();
        button1.Click += async (sender, e) =>
        {
            await ExampleMethodAsync();
            textBox1.Text += "\r\nControl returned to Click event handler.\n";
        };
    }

    private async Task ExampleMethodAsync()
    {
        // The following line simulates a task-returning asynchronous process.
        await Task.Delay(1000);
    }
}
```

有关如何创建和使用异步方法的详细信息，请参阅[使用 Async 和 Await 的异步编程](#)。

lambda 表达式和元组

自 C# 7.0 起, C# 语言提供对元组的内置支持。可以提供一个元组作为 Lambda 表达式的参数, 同时 Lambda 表达式也可以返回元组。在某些情况下, C# 编译器使用类型推理来确定元组组件的类型。

可通过用括号括住用逗号分隔的组件列表来定义元组。下面的示例使用包含三个组件的元组, 将一系列数字传递给 lambda 表达式, 此表达式将每个值翻倍, 然后返回包含乘法运算结果的元组(内含三个组件)。

```
Func<(int, int, int), (int, int, int> doubleThem = ns => (2 * ns.Item1, 2 * ns.Item2, 2 * ns.Item3);
var numbers = (2, 3, 4);
var doubledNumbers = doubleThem(numbers);
Console.WriteLine($"The set {numbers} doubled: {doubledNumbers}");
// Output:
// The set (2, 3, 4) doubled: (4, 6, 8)
```

通常, 元组字段命名为 `Item1`、`Item2` 等等。但是, 可以使用命名组件定义元组, 如以下示例所示。

```
Func<(int n1, int n2, int n3), (int, int, int> doubleThem = ns => (2 * ns.n1, 2 * ns.n2, 2 * ns.n3);
var numbers = (2, 3, 4);
var doubledNumbers = doubleThem(numbers);
Console.WriteLine($"The set {numbers} doubled: {doubledNumbers}");
```

若要详细了解 C# 元组, 请参阅[元组类型](#)。

含标准查询运算符的 lambda

在其他实现中, LINQ to Objects 有一个输入参数, 其类型是泛型委托 `Func<TResult>` 系列中的一种。这些委托使用类型参数来定义输入参数的数量和类型, 以及委托的返回类型。`Func` 委托对于封装用户定义的表达式非常有用, 这些表达式将应用于一组源数据中的每个元素。例如, 假设为 `Func<T, TResult>` 委托类型:

```
public delegate TResult Func<in T, out TResult>(T arg)
```

可以将委托实例化为 `Func<int, bool>` 实例, 其中 `int` 是输入参数, `bool` 是返回值。返回值始终在最后一个类型参数中指定。例如, `Func<int, string, bool>` 定义包含两个输入参数(`int` 和 `string`)且返回类型为 `bool` 的委托。下面的 `Func` 委托在调用后返回布尔值, 以指明输入参数是否等于 5:

```
Func<int, bool> equalsFive = x => x == 5;
bool result = equalsFive(4);
Console.WriteLine(result); // False
```

参数类型为 `Expression<TDelegate>` 时, 也可以提供 Lambda 表达式, 例如在 `Queryable` 类型内定义的标准查询运算符中提供。指定 `Expression<TDelegate>` 参数时, lambda 编译为表达式树。

下面的示例使用 `Count` 标准查询运算符:

```
int[] numbers = { 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 };
int oddNumbers = numbers.Count(n => n % 2 == 1);
Console.WriteLine($"There are {oddNumbers} odd numbers in {string.Join(" ", numbers)}");
```

编译器可以推断输入参数的类型, 或者你也可以显式指定该类型。这个特殊 lambda 表达式将计算那些除以 2 时余数为 1 的整数的数量 (`n`)。

下面的示例生成一个序列, 其中包含 `numbers` 数组中位于 9 之前的所有元素, 因为这是序列中第一个不符合条件的数字:

```
int[] numbers = { 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 };
var firstNumbersLessThanSix = numbers.TakeWhile(n => n < 6);
Console.WriteLine(string.Join(" ", firstNumbersLessThanSix));
// Output:
// 5 4 1 3
```

以下示例通过将输入参数括在括号中来指定多个输入参数。此方法返回 `numbers` 数组中的所有元素，直至遇到值小于其在数组中的序号位置的数字为止：

```
int[] numbers = { 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 };
var firstSmallNumbers = numbers.TakeWhile((n, index) => n >= index);
Console.WriteLine(string.Join(" ", firstSmallNumbers));
// Output:
// 5 4
```

Lambda 表达式中的类型推理

编写 lambda 时，通常不必为输入参数指定类型，因为编译器可以根据 lambda 主体、参数类型以及 C# 语言规范中描述的其他因素来推断类型。对于大多数标准查询运算符，第一个输入是源序列中的元素类型。如果要查询 `IEnumerable<Customer>`，则输入变量将被推断为 `Customer` 对象，这意味着你可以访问其方法和属性：

```
customers.Where(c => c.City == "London");
```

lambda 类型推理的一般规则如下：

- Lambda 包含的参数数量必须与委托类型包含的参数数量相同。
- Lambda 中的每个输入参数必须都能够隐式转换为其对应的委托参数。
- Lambda 的返回值（如果有）必须能够隐式转换为委托的返回类型。

请注意，lambda 表达式本身没有类型，因为通用类型系统没有“lambda 表达式”这一固有概念。不过，有时以一种非正式的方式谈论 lambda 表达式的“类型”会很方便。在这些情况下，类型是指委托类型或 lambda 表达式所转换到的 [Expression](#) 类型。

捕获 lambda 表达式中的外部变量和变量范围

lambda 可以引用外部变量。这些变量是在定义 lambda 表达式的方法中或包含 lambda 表达式的类型中的范围内变量。以这种方式捕获的变量将进行存储以备在 lambda 表达式中使用，即使在其他情况下，这些变量将超出范围并进行垃圾回收。必须明确地分配外部变量，然后才能在 lambda 表达式中使用该变量。下面的示例演示这些规则：

```

public static class VariableScopeWithLambdas
{
    public class VariableCaptureGame
    {
        internal Action<int> updateCapturedLocalVariable;
        internal Func<int, bool> isEqualToCapturedLocalVariable;

        public void Run(int input)
        {
            int j = 0;

            updateCapturedLocalVariable = x =>
            {
                j = x;
                bool result = j > input;
                Console.WriteLine($"'{j}' is greater than {input}: {result}");
            };

            isEqualToCapturedLocalVariable = x => x == j;

            Console.WriteLine($"Local variable before lambda invocation: {j}");
            updateCapturedLocalVariable(10);
            Console.WriteLine($"Local variable after lambda invocation: {j}");
        }
    }

    public static void Main()
    {
        var game = new VariableCaptureGame();

        int gameInput = 5;
        game.Run(gameInput);

        int jTry = 10;
        bool result = game.isEqualToCapturedLocalVariable(jTry);
        Console.WriteLine($"Captured local variable is equal to {jTry}: {result}");

        int anotherJ = 3;
        game.updateCapturedLocalVariable(anotherJ);

        bool equalToAnother = game.isEqualToCapturedLocalVariable(anotherJ);
        Console.WriteLine($"Another lambda observes a new value of captured variable: {equalToAnother}");
    }
}

```

下列规则适用于 lambda 表达式中的变量范围：

- 捕获的变量将不会被作为垃圾回收，直至引用变量的委托符合垃圾回收的条件。
- 在封闭方法中看不到 lambda 表达式内引入的变量。
- lambda 表达式无法从封闭方法中直接捕获 `in`、`ref` 或 `out` 参数。
- lambda 表达式中的 `return` 语句不会导致封闭方法返回。
- 如果相应跳转语句的目标位于 lambda 表达式块之外，lambda 表达式不得包含 `goto`、`break` 或 `continue` 语句。同样，如果目标在块内部，在 lambda 表达式块外部使用跳转语句也是错误的。

从 C# 9.0 开始，可以将 `static` 修饰符应用于 lambda 表达式，以防止由 lambda 无意中捕获本地变量或实例状

态：

```
Func<double, double> square = static x => x * x;
```

静态 lambda 无法从封闭范围中捕获本地变量或实例状态，但可以引用静态成员和常量定义。

C# 语言规范

有关详细信息，请参阅 [C# 语言规范](#) 中的 [匿名函数表达式部分](#)。

有关 C# 9.0 中添加的功能的详细信息，请参阅以下功能建议说明：

- [Lambda 弃元参数](#)
- [静态匿名函数](#)

另请参阅

- [C# 参考](#)
- [C# 运算符和表达式](#)
- [LINQ\(语言集成查询\)](#)
- [表达式树](#)
- [本地函数与 Lambda 表达式](#)
- [Visual Studio 2008 C# 示例\(请参阅 LINQ 示例查询文件和 XQuery 程序\)](#)

模式 (C# 参考)

2021/5/10 • [Edit Online](#)

C# 在 C# 7.0 中引入了模式匹配。自此之后，每个主要 C# 版本都扩展了模式匹配功能。以下 C# 表达式和语句支持模式匹配：

- `is` 表达式
- `switch` 语句
- `switch` 表达式 (在 C# 8.0 中引入)

在这些构造中，可将输入表达式与以下任一模式进行匹配：

- **声明模式**：用于检查表达式的运行时类型，如果匹配成功，则将表达式结果分配给声明的变量。在 C# 7.0 中引入。
- **类型模式**：用于检查表达式的运行时类型。在 C# 9.0 中引入。
- **常量模式**：用于测试表达式结果是否等于指定常量。在 C# 7.0 中引入。
- **关系模式**：用于将表达式结果与指定常量进行比较。在 C# 9.0 中引入。
- **逻辑模式**：用于测试表达式是否与模式的逻辑组合匹配。在 C# 9.0 中引入。
- **属性模式**：用于测试表达式的属性或字段是否与嵌套模式匹配。在 C# 8.0 中引入。
- **位置模式**：用于解构表达式结果并测试结果值是否与嵌套模式匹配。在 C# 8.0 中引入。
- **`var` 模式**：用于匹配任何表达式并将其结果分配给声明的变量。在 C# 7.0 中引入。
- **弃元模式**：用于匹配任何表达式。在 C# 8.0 中引入。

逻辑、**属性**和**位置**模式都是递归模式。也就是说，它们可包含嵌套模式。

有关如何使用这些模式来生成数据驱动算法的示例，请参阅[教程：使用模式匹配来生成类型驱动的算法和数据驱动的算法](#)。

声明和类型模式

使用声明和类型模式检查表达式的运行时类型是否与给定类型兼容。借助声明模式，还可声明新的局部变量。

当声明模式与表达式匹配时，将为该变量分配转换后的表达式结果，如以下示例所示：

```
object greeting = "Hello, World!";
if (greeting is string message)
{
    Console.WriteLine(message.ToLower()); // output: hello, world!
}
```

从 C# 7.0 开始，类型为 `T` 的声明模式在表达式结果为非 NULL 且满足以下任一条件时与表达式匹配：

- 表达式结果的运行时类型为 `T`。
- 表达式结果的运行时类型派生自类型 `T`，实现接口 `T`，或者存在从其到 `T` 的另一种[隐式引用转换](#)。下面的示例演示满足此条件时的两种案例：

```

var numbers = new int[] { 10, 20, 30 };
Console.WriteLine(GetSourceLabel(numbers)); // output: 1

var letters = new List<char> { 'a', 'b', 'c', 'd' };
Console.WriteLine(GetSourceLabel(letters)); // output: 2

static int GetSourceLabel<T>(IEnumerable<T> source) => source switch
{
    Array array => 1,
    ICollection<T> collection => 2,
    _ => 3,
};

```

在上述示例中，在第一次调用 `GetSourceLabel` 方法时，第一种模式与参数值匹配，因为参数的运行时类型 `int[]` 派生自 `Array` 类型。在第二次调用 `GetSourceLabel` 方法时，参数的运行时类型 `List<T>` 并非派生自 `Array` 类型，但却实现 `ICollection<T>` 接口。

- 表达式结果的运行时类型是具有基础类型 `T` 的可为 `null` 的值类型。
- 存在从表达式结果的运行时类型到类型 `T` 的装箱或取消装箱转换。

下面的示例演示最后两个条件：

```

int? xNullable = 7;
int y = 23;
object yBoxed = y;
if (xNullable is int a && yBoxed is int b)
{
    Console.WriteLine(a + b); // output: 30
}

```

如果只想检查表达式类型，可使用弃元 `_` 代替变量名，如以下示例所示：

```

public abstract class Vehicle {}
public class Car : Vehicle {}
public class Truck : Vehicle {}

public static class TollCalculator
{
    public static decimal CalculateToll(this Vehicle vehicle) => vehicle switch
    {
        Car _ => 2.00m,
        Truck _ => 7.50m,
        null => throw new ArgumentNullException(nameof(vehicle)),
        _ => throw new ArgumentException("Unknown type of a vehicle", nameof(vehicle)),
    };
}

```

从 C# 9.0 开始，可对此使用类型模式，如以下示例所示：

```

public static decimal CalculateToll(this Vehicle vehicle) => vehicle switch
{
    Car => 2.00m,
    Truck => 7.50m,
    null => throw new ArgumentNullException(nameof(vehicle)),
    _ => throw new ArgumentException("Unknown type of a vehicle", nameof(vehicle)),
};

```

类似于声明模式，当表达式结果为非 `null` 且其运行时类型满足任何上述条件时，类型模式则与表达式匹配。

有关详细信息，请参阅功能方案说明的[声明模式](#)和[类型模式](#)部分。

常量模式

从 C# 7.0 开始，可使用常量模式来测试表达式结果是否等于指定的常量，如以下示例所示：

```
public static decimal GetGroupTicketPrice(int visitorCount) => visitorCount switch
{
    1 => 12.0m,
    2 => 20.0m,
    3 => 27.0m,
    4 => 32.0m,
    0 => 0.0m,
    _ => throw new ArgumentException($"Not supported number of visitors: {visitorCount}",
        nameof(visitorCount)),
};
```

在常量模式中，可使用任何常量表达式，例如：

- [integer](#) 或 [floating-point](#) 数值文本
- [char](#) 或 [string](#) 文本
- 布尔值 `true` 或 `false`
- [enum](#) 值
- 声明[常量](#)字段或本地的名称
- `null`

常量模式用于检查 `null`，如以下示例所示：

```
if (input is null)
{
    return;
}
```

编译器保证在计算表达式 `x is null` 时，不会调用用户重载的相等运算符 `==`。

从 C# 9.0 开始，可使用[否定](#) `null` 常量模式来检查非 NULL，如以下示例所示：

```
if (input is not null)
{
    // ...
}
```

有关详细信息，请参阅功能建议说明的[常量模式](#)部分。

关系模式

从 C# 9.0 开始，可使用关系模式将表达式结果与常量进行比较，如以下示例所示：

```

Console.WriteLine(Classify(13)); // output: Too high
Console.WriteLine(Classify(double.NaN)); // output: Unknown
Console.WriteLine(Classify(2.4)); // output: Acceptable

static string Classify(double measurement) => measurement switch
{
    < -4.0 => "Too low",
    > 10.0 => "Too high",
    double.NaN => "Unknown",
    _ => "Acceptable",
};

```

在关系模式中，可使用[关系运算符](#) `<`、`>`、`<=` 或 `>=` 中的任何一个。关系模式的右侧部分必须是常数表达式。常数表达式可以是 `integer`、`floating-point`、`char` 或 `enum` 类型。

要检查表达式结果是否在某个范围内，请将其与[合取 and 模式](#)匹配，如以下示例所示：

```

Console.WriteLine(GetCalendarSeason(new DateTime(2021, 3, 14))); // output: spring
Console.WriteLine(GetCalendarSeason(new DateTime(2021, 7, 19))); // output: summer
Console.WriteLine(GetCalendarSeason(new DateTime(2021, 2, 17))); // output: winter

static string GetCalendarSeason(DateTime date) => date.Month switch
{
    >= 3 and < 6 => "spring",
    >= 6 and < 9 => "summer",
    >= 9 and < 12 => "autumn",
    12 or (>= 1 and < 3) => "winter",
    _ => throw new ArgumentOutOfRangeException(nameof(date), $"Date with unexpected month: {date.Month}."),
};

```

如果表达式结果为 `null` 或未能通过可为空或取消装箱转换转换为常量类型，则关系模式与表达式不匹配。

有关详细信息，请参阅功能建议说明的[关系模式](#)部分。

逻辑模式

从 C# 9.0 开始，可使用 `not`、`and` 和 `or` 模式连结符来创建以下逻辑模式：

- 否定 `not` 模式在否定模式与表达式不匹配时与表达式匹配。下面的示例说明如何否定常量 `null` 模式来检查表达式是否为非空值：

```

if (input is not null)
{
    // ...
}

```

- 合取 `and` 模式在两个模式都与表达式匹配时与表达式匹配。以下示例显示如何组合[关系模式](#)来检查值是否在某个范围内：

```

Console.WriteLine(Classify(13)); // output: High
Console.WriteLine(Classify(-100)); // output: Too low
Console.WriteLine(Classify(5.7)); // output: Acceptable

static string Classify(double measurement) => measurement switch
{
    < -40.0 => "Too low",
    >= -40.0 and < 0 => "Low",
    >= 0 and < 10.0 => "Acceptable",
    >= 10.0 and < 20.0 => "High",
    >= 20.0 => "Too high",
    double.NaN => "Unknown",
};

```

- 析取 `or` 模式在任一模式与表达式匹配时与表达式匹配，如以下示例所示：

```

Console.WriteLine(GetCalendarSeason(new DateTime(2021, 1, 19))); // output: winter
Console.WriteLine(GetCalendarSeason(new DateTime(2021, 10, 9))); // output: autumn
Console.WriteLine(GetCalendarSeason(new DateTime(2021, 5, 11))); // output: spring

static string GetCalendarSeason(DateTime date) => date.Month switch
{
    3 or 4 or 5 => "spring",
    6 or 7 or 8 => "summer",
    9 or 10 or 11 => "autumn",
    12 or 1 or 2 => "winter",
    _ => throw new ArgumentOutOfRangeException(nameof(date), $"Date with unexpected month: {date.Month}."),
};

```

如前面的示例所示，可在模式中重复使用模式连结符。

- `and` 模式连结符的优先级高于 `or`。要显式指定优先级，请使用括号，如以下示例所示：

```
static bool IsLetter(char c) => c is (>= 'a' and <= 'z') or (>= 'A' and <= 'Z');
```

NOTE

检查模式的顺序是不确定的。在运行时，可先检查 `or` 和 `and` 模式的右侧嵌套模式。

有关详细信息，请参阅功能建议说明的[模式连结符](#)部分。

属性模式

从 C# 8.0 开始，可使用属性模式来将表达式的属性或字段与嵌套模式进行匹配，如以下示例所示：

```
static bool IsConferenceDay(DateTime date) => date is { Year: 2020, Month: 5, Day: 19 or 20 or 21 };
```

当表达式结果为非 NULL 且每个嵌套模式都与表达式结果的相应属性或字段匹配时，属性模式将与表达式匹配。

还可将运行时类型检查和变量声明添加到属性模式，如以下示例所示：

```

Console.WriteLine(TakeFive("Hello, world!")); // output: Hello
Console.WriteLine(TakeFive("Hi!")); // output: Hi!
Console.WriteLine(TakeFive(new[] { '1', '2', '3', '4', '5', '6', '7' })); // output: 12345
Console.WriteLine(TakeFive(new[] { 'a', 'b', 'c' })); // output: abc

static string TakeFive(object input) => input switch
{
    string { Length: >= 5 } s => s.Substring(0, 5),
    string s => s,

    ICollection<char> { Count: >= 5 } symbols => new string(symbols.Take(5).ToArray()),
    ICollection<char> symbols => new string(symbols.ToArray()),

    null => throw new ArgumentNullException(nameof(input)),
    _ => throw new ArgumentException("Not supported input type."),
};

```

属性模式是一种递归模式。也就是说，可以将任何模式用作嵌套模式。使用属性模式将部分数据与嵌套模式进行匹配，如以下示例所示：

```

public record Point(int X, int Y);
public record Segment(Point Start, Point End);

static bool IsAnyEndAtOrigin(Segment segment) =>
    segment is { Start: { X: 0, Y: 0 } } or { End: { X: 0, Y: 0 } };

```

前面的示例使用 C# 9.0 及更高版本中提供的两个功能：[或](#) 模式连接符和记录类型。

有关详细信息，请参阅功能建议说明的[属性模式](#)部分。

位置模式

从 C# 8.0 开始，可使用位置模式来解构表达式结果并将结果值与相应的嵌套模式匹配，如以下示例所示：

```

public readonly struct Point
{
    public int X { get; }
    public int Y { get; }

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

    public void Deconstruct(out int x, out int y) => (x, y) = (X, Y);
}

static string Classify(Point point) => point switch
{
    (0, 0) => "Origin",
    (1, 0) => "positive X basis end",
    (0, 1) => "positive Y basis end",
    _ => "Just a point",
};

```

在前面的示例中，表达式的类型包含 [Deconstruct](#) 方法，该方法用于解构表达式结果。还可将[元组类型](#)的表达式与位置模式进行匹配。这样，就可将多个输入与各种模式进行匹配，如以下示例所示：

```

static decimal GetGroupTicketPriceDiscount(int groupSize, DateTime visitDate)
    => (groupSize, visitDate.DayOfWeek) switch
    {
        (<= 0, _) => throw new ArgumentException("Group size must be positive."),
        (_, DayOfWeek.Saturday or DayOfWeek.Sunday) => 0.0m,
        (>= 5 and < 10, DayOfWeek.Monday) => 20.0m,
        (>= 10, DayOfWeek.Monday) => 30.0m,
        (>= 5 and < 10, _) => 12.0m,
        (>= 10, _) => 15.0m,
        _ => 0.0m,
    };

```

前面的示例使用在 C# 9.0 及更高版本中提供的[关系](#)和[逻辑](#)模式。

- 可在位置模式中使用元组元素的名称和 `Deconstruct` 参数, 如以下示例所示:

```

var numbers = new List<int> { 1, 2, 3 };
if (SumAndCount(numbers) is (Sum: var sum, Count: > 0))
{
    Console.WriteLine($"Sum of [{string.Join(" ", numbers)}] is {sum}"); // output: Sum of [1 2 3] is 6
}

static (double Sum, int Count) SumAndCount(IEnumerable<int> numbers)
{
    int sum = 0;
    int count = 0;
    foreach (int number in numbers)
    {
        sum += number;
        count++;
    }
    return (sum, count);
}

```

还可通过以下任一方式扩展位置模式:

- 添加运行时类型检查和变量声明, 如以下示例所示:

```

public record Point2D(int X, int Y);
public record Point3D(int X, int Y, int Z);

static string PrintIfAllCoordinatesArePositive(object point) => point switch
{
    Point2D (> 0, > 0) p => p.ToString(),
    Point3D (> 0, > 0, > 0) p => p.ToString(),
    _ => string.Empty,
};

```

前面的示例使用隐式提供 `Deconstruct` 方法的[位置记录](#)。

- 在位置模式中使用[属性模式](#), 如以下示例所示:

```

public record WeightedPoint(int X, int Y)
{
    public double Weight { get; set; }
}

static bool IsInDomain(WeightedPoint point) => point is (>= 0, >= 0) { Weight: >= 0.0 };

```

- 结合前面的两种用法, 如以下示例所示:

```
if (input is WeightedPoint (> 0, > 0) { Weight: > 0.0 } p)
{
    // ...
}
```

位置模式是一种递归模式。也就是说，可以将任何模式用作嵌套模式。

有关详细信息，请参阅功能建议说明的[位置模式](#)部分。

var 模式

从 C# 7.0 开始，可使用 `var` 模式来匹配任何表达式（包括 `null`），并将其结果分配给新的局部变量，如以下示例所示：

```
static bool IsAcceptable(int id, int absLimit) =>
    SimulateDataFetch(id) is var results
    && results.Min() >= -absLimit
    && results.Max() <= absLimit;

static int[] SimulateDataFetch(int id)
{
    var rand = new Random();
    return Enumerable
        .Range(start: 0, count: 5)
        .Select(s => rand.Next(minValue: -10, maxValue: 11))
        .ToArray();
}
```

需要布尔表达式中的临时变量来保存中间计算的结果时，`var` 模式很有用。当需要在 `switch` 表达式或语句的 `when` 大小写临界子句中执行其他检查时，也可使用 `var` 模式，如以下示例所示：

```
public record Point(int X, int Y);

static Point Transform(Point point) => point switch
{
    var (x, y) when x < y => new Point(-x, y),
    var (x, y) when x > y => new Point(x, -y),
    var (x, y) => new Point(x, y),
};

static void TestTransform()
{
    Console.WriteLine(Transform(new Point(1, 2))); // output: Point { X = -1, Y = 2 }
    Console.WriteLine(Transform(new Point(5, 2))); // output: Point { X = 5, Y = -2 }
}
```

在前面的示例中，模式 `var (x, y)` 等效于[位置模式](#) `(var x, var y)`。

在 `var` 模式中，声明变量的类型是与该模式匹配的表达式的编译时类型。

有关详细信息，请参阅功能建议说明的[Var 模式](#)部分。

弃元模式

从 C# 8.0 开始，可使用弃元模式 `_` 来匹配任何表达式，包括 `null`，如以下示例所示：

```
Console.WriteLine(GetDiscountInPercent(DayOfWeek.Friday)); // output: 5.0
Console.WriteLine(GetDiscountInPercent(null)); // output: 0.0
Console.WriteLine(GetDiscountInPercent((DayOfWeek)10)); // output: 0.0

static decimal GetDiscountInPercent(DayOfWeek? dayOfWeek) => dayOfWeek switch
{
    DayOfWeek.Monday => 0.5m,
    DayOfWeek.Tuesday => 12.5m,
    DayOfWeek.Wednesday => 7.5m,
    DayOfWeek.Thursday => 12.5m,
    DayOfWeek.Friday => 5.0m,
    DayOfWeek.Saturday => 2.5m,
    DayOfWeek.Sunday => 2.0m,
    _ => 0.0m,
};
```

在前面的示例中，弃元模式用于处理 `null` 以及没有相应的 `DayOfWeek` 枚举成员的任何整数值。这可保证示例中的 `switch` 表达式可处理所有可能的输入值。如果没有在 `switch` 表达式中使用弃元模式，并且该表达式的任何模式均与输入不匹配，则运行时会引发异常。如果 `switch` 表达式未处理所有可能的输入值，则编译器会生成警告。

弃元模式不能是 `is` 表达式或 `switch` 语句中的模式。在这些案例中，要匹配任何表达式，请使用带有弃元 `var _` 的 `var` 模式。

有关详细信息，请参阅功能建议说明的[弃元模式](#)部分。

带括号模式

从 C# 9.0 开始，可在任何模式两边加上括号。通常，这样做是为了强调或更改[逻辑模式](#)中的优先级，如以下示例所示：

```
if (input is not (float or double))
{
    return;
}
```

C# 语言规范

有关详细信息，请参阅以下功能建议说明：

- [C# 7.0 的模式匹配](#)
- [递归模式匹配\(在 C# 8.0 中引入\)](#)
- [C# 9.0 的模式匹配更改](#)

另请参阅

- [C# 参考](#)
- [C# 运算符和表达式](#)
- [教程: 使用模式匹配来构建类型驱动和数据驱动的算法](#)

+ 和 += 运算符 (C# 参考)

2021/5/10 • [Edit Online](#)

内置整型和浮点数值类型、字符串类型以及委托类型支持 + 和 += 运算符。

有关算术 + 运算符的信息，请参阅[一元加和减运算符](#)和[算术运算符](#)文章的[加法运算符 +](#)部分。

字符串串联

当其中的一个操作数是字符串类型或两个操作数都是字符串类型时，+ 运算符将其操作数的字符串表示形式（`null` 的字符串表示形式）串联在一起：

```
Console.WriteLine("Forgot" + "white space");
Console.WriteLine("Probably the oldest constant: " + Math.PI);
Console.WriteLine(null + "Nothing to add.");
// Output:
// Forgotwhite space
// Probably the oldest constant: 3.14159265358979
// Nothing to add.
```

从 C# 6 开始，[字符串内插](#)提供了格式化字符串更为便捷的方式：

```
Console.WriteLine($"Probably the oldest constant: {Math.PI:F2}");
// Output:
// Probably the oldest constant: 3.14
```

委托组合

对于委托类型相同的操作数，+ 运算符在调用时返回新的委托实例，调用左侧的操作数，然后调用右侧的操作数。如果任何操作数均为 `null`，则 + 运算符将返回另一个操作数（也可能是 `null`）的值。下面的示例演示如何组合使用委托和 + 运算符：

```
Action a = () => Console.Write("a");
Action b = () => Console.Write("b");
Action ab = a + b;
ab(); // output: ab
```

若要执行委托删除，请使用 - 运算符。

有关委托类型的详细信息，请参阅[委托](#)。

加法赋值运算符 +=

使用 += 运算符的表达式，例如

```
x += y
```

等效于

```
x = x + y
```

不同的是 `x` 只计算一次。

下面的示例演示 `+=` 运算符的用法：

```
int i = 5;
i += 9;
Console.WriteLine(i);
// Output: 14

string story = "Start. ";
story += "End.";
Console.WriteLine(story);
// Output: Start. End.

Action printer = () => Console.Write("a");
printer(); // output: a

Console.WriteLine();
printer += () => Console.Write("b");
printer(); // output: ab
```

在订阅事件时，还可以使用 `+=` 运算符来指定事件处理程序方法。有关详细信息，请参阅[如何：订阅和取消订阅事件](#)。

运算符可重载性

用户定义的类型可以重载 `+` 运算符。重载二元 `+` 运算符后，也会隐式重载 `+=` 运算符。用户定义类型不能显式重载 `+=` 运算符。

C# 语言规范

有关详细信息，请参阅[C# 语言规范的一元加运算符](#)和[加法运算符](#)部分。

请参阅

- [C# 参考](#)
- [C# 运算符和表达式](#)
- [如何连接多个字符串](#)
- [事件](#)
- [算术运算符](#)
- [- 和 -= 运算符](#)

- 和 -= 运算符 (C# 参考)

2021/5/10 • [Edit Online](#)

内置[整型](#)和[浮点数字](#)类型以及[委托](#)类型支持 `-` 和 `-=` 运算符。

有关算术 `-` 运算符的信息, 请参阅[一元加和减运算符](#)和[算术运算符](#)文章的[减法运算符 - 部分](#)。

委托删除

对于[委托](#)类型相同的操作数, `-` 运算符返回如下计算的委托实例:

- 如果两个操作数都为非空, 并且右侧操作数的调用列表是左侧操作数调用列表的正确连续子列表, 则该操作的结果是通过从左侧操作数的调用列表中删除右侧操作数的条目而获得的新调用列表。如果右侧操作数的列表与左侧操作数列表中的多个连续子列表匹配, 则仅删除最右侧的匹配子列表。如果删除行为导致出现空列表, 则结果为 `null`。

```
Action a = () => Console.WriteLine("a");
Action b = () => Console.WriteLine("b");

var abbaab = a + b + b + a + a + b;
abbaab(); // output: abbaab
Console.WriteLine();

var ab = a + b;
var abba = abbaab - ab;
abba(); // output: abba
Console.WriteLine();

var nihil = abbaab - abbaab;
Console.WriteLine(nihil is null); // output: True
```

- 如果右侧操作数的调用列表不是左侧操作数调用列表的正确连续子列表, 则该操作的结果是左侧操作数。例如, 删除不属于多播委托的委托不会执行任何操作, 从而导致不变的多播委托。

```
Action a = () => Console.WriteLine("a");
Action b = () => Console.WriteLine("b");

var abbaab = a + b + b + a + a + b;
var aba = a + b + a;

var first = abbaab - aba;
first(); // output: abbaab
Console.WriteLine();
Console.WriteLine(object.ReferenceEquals(abbaab, first)); // output: True

Action a2 = () => Console.WriteLine("a");
var changed = aba - a;
changed(); // output: ab
Console.WriteLine();
var unchanged = aba - a2;
unchanged(); // output: aba
Console.WriteLine();
Console.WriteLine(object.ReferenceEquals(aba, unchanged)); // output: True
```

前面的示例还演示了在删除委托期间对委托实例进行比较。例如, 通过计算相同的[Lambda 表达式](#)生成的委托不相等。有关委托相等性的详细信息, 请参阅[C# 语言规范的委托相等运算符部分](#)。

- 如果左侧操作数为 `null`，则操作结果为 `null`。如果右侧操作数为 `null`，则操作的结果是左侧操作数。

```
Action a = () => Console.WriteLine("a");

var nothing = null - a;
Console.WriteLine(nothing is null); // output: True

var first = a - null;
a(); // output: a
Console.WriteLine();
Console.WriteLine(object.ReferenceEquals(first, a)); // output: True
```

若要合并委托，请使用 [+ 运算符](#)。

有关委托类型的详细信息，请参阅[委托](#)。

减法赋值运算符 -=

使用 `-=` 运算符的表达式，例如

```
x -= y
```

等效于

```
x = x - y
```

不同的是 `x` 只计算一次。

下面的示例演示 `-=` 运算符的用法：

```
int i = 5;
i -= 9;
Console.WriteLine(i);
// Output: -4

Action a = () => Console.WriteLine("a");
Action b = () => Console.WriteLine("b");
var printer = a + b + a;
printer(); // output: aba

Console.WriteLine();
printer -= a;
printer(); // output: ab
```

还可以使用 `-=` 运算符指定在取消订阅[事件](#)时要删除的事件处理程序方法。有关详细信息，请参阅[如何订阅和取消订阅事件](#)。

运算符可重载性

用户定义的类型可以重载 `-` 运算符。重载二元 `-` 运算符后，也会隐式重载 `-=` 运算符。用户定义类型不能显式重载 `-=` 运算符。

C# 语言规范

有关详细信息，请参阅[C# 语言规范的一元减运算符](#)和[减法运算符](#)部分。

另请参阅

- [C# 参考](#)
- [C# 运算符和表达式](#)
- [事件](#)
- [算术运算符](#)
- [+ 和 += 运算符](#)

? : 运算符 (C# 参考)

2021/5/10 • [Edit Online](#)

条件运算符 (?:) 也称为三元条件运算符，用于计算布尔表达式，并根据布尔表达式的计算结果为 true 还是 false 来返回两个表达式中的一个结果，如以下示例所示：

```
string GetWeatherDisplay(double tempInCelsius) => tempInCelsius < 20.0 ? "Cold." : "Perfect!";

Console.WriteLine(GetWeatherDisplay(15)); // output: Cold.
Console.WriteLine(GetWeatherDisplay(27)); // output: Perfect!
```

如上述示例所示，条件运算符的语法如下所示：

```
condition ? consequent : alternative
```

condition 表达式的计算结果必须为 true 或 false。若 condition 的计算结果为 true，将计算 consequent，其结果成为运算结果。若 condition 的计算结果为 false，将计算 alternative，其结果成为运算结果。只会计算 consequent 或 alternative。

从 C# 9.0 开始，条件表达式由目标确定类型。也就是说，如果条件表达式的目标类型是已知的，则 consequent 和 alternative 的类型必须可隐式转换为目标类型，如以下示例所示：

```
var rand = new Random();
var condition = rand.NextDouble() > 0.5;

int? x = condition ? 12 : null;

IEnumerable<int> xs = x is null ? new List<int>() { 0, 1 } : new int[] { 2, 3 };
```

如果条件表达式的目标类型是未知的（例如使用 var 关键字时）或者采用 C# 8.0 及更早版本，则 consequent 和 alternative 的类型必须相同，或者必须存在从一种类型到另一种类型的隐式转换：

```
var rand = new Random();
var condition = rand.NextDouble() > 0.5;

var x = condition ? 12 : (int?)null;
```

条件运算符为右联运算符，即形式的表达式

```
a ? b : c ? d : e
```

计算结果为

```
a ? b : (c ? d : e)
```

TIP

可以使用以下助记键设备记住条件运算符的计算方式：

```
is this condition true ? yes : no
```

ref 条件表达式

从 C# 7.2 开始，可通过 ref 条件表达式有条件地分配 `ref local` 或 `ref readonly local` 变量。还可以使用 ref 条件表达式作为引用返回值或 `ref` 方法参数。

ref 条件表达式的语法如下所示：

```
condition ? ref consequent : ref alternative
```

ref 条件表达式与原始的条件运算符相似，仅计算两个表达式其中之一：`consequent` 或 `alternative`。

在 ref 条件表达式中，`consequent` 和 `alternative` 的类型必须相同。ref 条件表达式不由目标确定类型。

下面的示例演示 ref 条件表达式的用法：

```
var smallArray = new int[] { 1, 2, 3, 4, 5 };
var largeArray = new int[] { 10, 20, 30, 40, 50 };

int index = 7;
ref int refValue = ref ((index < 5) ? ref smallArray[index] : ref largeArray[index - 5]);
refValue = 0;

index = 2;
((index < 5) ? ref smallArray[index] : ref largeArray[index - 5]) = 100;

Console.WriteLine(string.Join(" ", smallArray));
Console.WriteLine(string.Join(" ", largeArray));
// Output:
// 1 2 100 4 5
// 10 20 0 40 50
```

条件运算符和 if..else 语句

需要根据条件计算值时，使用条件运算符而不是 `if-else` 语句可以使代码更简洁。下面的示例演示了将整数归类为负数或非负数的两种方法：

```
int input = new Random().Next(-5, 5);

string classify;
if (input >= 0)
{
    classify = "nonnegative";
}
else
{
    classify = "negative";
}

classify = (input >= 0) ? "nonnegative" : "negative";
```

运算符可重载性

用户定义类型不能重载条件运算符。

C# 语言规范

有关详细信息, 请参阅 [C# 语言规范的条件运算符部分](#)。

有关 C# 7.2 及更高版本中添加的功能的详细信息, 请参阅以下功能建议说明:

- [ref 条件表达式 \(C# 7.2\)](#)
- [目标类型的条件表达式 \(C# 9.0\)](#)

请参阅

- [C# 参考](#)
- [C# 运算符和表达式](#)
- [if-else 语句](#)
- [?. 和 ?\[\] 运算符](#)
- [?? 和 ??= 运算符](#)
- [ref 关键字](#)

! (null 包容) 运算符 (C# 参考)

2021/5/10 • [Edit Online](#)

在 C# 8.0 及更高版本中可用，一元后缀 `!` 运算符是 null 包容运算符或 null 抑制运算符。在已启用的[可为空的注释上下文](#)中，可以使用 null 包容运算符来声明可为空的引用类型的表达式 `x` 不为 `null`：`x!`。一元前缀 `!` 运算符是[逻辑非运算符](#)。

null 包容运算符在运行时不起作用。它仅通过更改表达式的 null 状态来影响编译器的静态流分析。在运行时，表达式 `x!` 的计算结果为基础表达式 `x` 的结果。

有关可为空引用类型特性的详细信息，请参见[可为空引用类型](#)。

示例

null 包容运算符的一个用例是测试参数验证逻辑。例如，请考虑以下类：

```
#nullable enable
public class Person
{
    public Person(string name) => Name = name ?? throw new ArgumentNullException(nameof(name));

    public string Name { get; }
}
```

使用[测试框架](#)，可以在构造函数中为验证逻辑创建以下测试：

```
[TestMethod, ExpectedException(typeof(ArgumentNullException))]
public void NullNameShouldThrowTest()
{
    var person = new Person(null!);
}
```

如果不使用 null 包容运算符，编译器将为前面的代码生成以下警告：

`Warning CS8625: Cannot convert null literal to non-nullable reference type`。通过使用 null 包容运算符，可以告知编译器传递 `null` 是预期行为，不应发出警告。

如果你明确知道某个表达式不能为 `null`，但编译器无法识别它，也可以使用 null 包容运算符。在下面的示例中，如果 `IsValid` 方法返回 `true`，则其参数不是 `null`，可以放心取消对它的引用：

```
public static void Main()
{
    Person? p = Find("John");
    if (IsValid(p))
    {
        Console.WriteLine($"Found {p!.Name}");
    }
}

public static bool IsValid(Person? person)
=> person is not null && person.Name is not null;
```

如果没有 null 包容运算符，编译器将为 `p.Name` 代码生成以下警告：

`Warning CS8602: Dereference of a possibly null reference`。

如果可以修改 `IsValid` 方法，则可使用 `NotNullWhen` 属性告知编译器，当方法返回 `true` 时，`IsValid` 方法的参数不能是 `null`：

```
public static void Main()
{
    Person? p = Find("John");
    if (IsValid(p))
    {
        Console.WriteLine($"Found {p.Name}");
    }
}

public static bool IsValid([NotNullWhen(true)] Person? person)
=> person is not null && person.Name is not null;
```

在前面的例子中，不需要使用 `null` 容包运算符，因为编译器有足够的信息来发现 `p` 不能是 `if` 语句中的 `null`。如需深入了解允许你提供有关变量 `null` 状态的其他信息的属性，请参阅[使用属性升级 API 以定义 null 期望值](#)。

C# 语言规范

有关详细信息，请参阅[可为空的引用类型规范草案](#)的 `null` 容包性运算符部分。

请参阅

- [C# 参考](#)
- [C# 运算符和表达式](#)
- [教程: 使用可为空引用类型进行设计](#)

?? 和 ??= 运算符 (C# 参考)

2021/5/10 • [Edit Online](#)

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

C# 8.0 及更高版本中可使用空合并赋值运算符 `??=`, 该运算符仅在左侧操作数的求值结果为 `null` 时, 才将其右侧操作数的值赋值给左操作数。如果左操作数的计算结果为非 `null`, 则 `??=` 运算符不会计算其右操作数。

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

(numbers ??= new List<int>()).Add(5);
Console.WriteLine(string.Join(" ", numbers)); // output: 5

numbers.Add(a ??= 0);
Console.WriteLine(string.Join(" ", numbers)); // output: 5 0
Console.WriteLine(a); // output: 0
```

`??=` 运算符的左操作数必须是变量、属性或索引器元素。

在 C# 7.3 及更早版本中, `??` 运算符左操作数的类型必须是引用类型或可以为 `null` 的值类型。从 C# 8.0 版本开始, 该要求替换为以下内容: `??` 和 `??=` 运算符的左操作数的类型必须是可以为 `null` 的值类型。特别是从 C# 8.0 开始, 可以使用具有无约束类型参数的 `null` 合并运算符:

```
private static void Display<T>(T a, T backup)
{
    Console.WriteLine(a ?? backup);
}
```

`null` 合并运算符是右结合运算符。也就是说, 是窗体的表达式

```
a ?? b ?? c
d ??= e ??= f
```

会像这样求值

```
a ?? (b ?? c)
d ??= (e ??= f)
```

示例

`??` 和 `??=` 运算符在以下应用场景中很有用:

- 在包含 `null` 条件运算符 `?.` 和 `?[]` 的表达式中, 当包含 `null` 条件运算的表达式结果为 `null` 时, 可以使用 `??` 运算符来提供替代表达式用于求值:

```
double SumNumbers(List<double[]> setsOfNumbers, int indexOfSetToSum)
{
    return setsOfNumbers?[indexOfSetToSum]?.Sum() ?? double.NaN;
}

var sum = SumNumbers(null, 0);
Console.WriteLine(sum); // output: NaN
```

- 当使用[可以为 null 值类型](#)并且需要提供基础值类型的值时，可以使用 `??` 运算符指定当可以为 null 的类型的值为 `null` 时要提供的值：

```
int? a = null;
int b = a ?? -1;
Console.WriteLine(b); // output: -1
```

如果可以为 null 的类型的值为 `null` 时要使用的值应为基础值类型的默认值，请使用 [Nullable<T>.GetValueOrDefault\(\)](#) 方法。

- 从 C# 7.0 开始，可以使用 `throw 表达式` 作为 `??` 运算符的右操作数，以使参数检查代码更简洁：

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

前面的示例还演示了如何使用 [expression-bodied 成员](#) 来定义属性。

- 从 C# 8.0 开始，可以使用 `??=` 运算符将这样的代码

```
if (variable is null)
{
    variable = expression;
}
```

替换为以下代码：

```
variable ??= expression;
```

运算符可重载性

运算符 `??` 和 `??=` 无法进行重载。

C# 语言规范

有关 `??` 运算符的详细信息，请参阅 [C# 语言规范的 null 合并运算符部分](#)。

有关 `??=` 运算符的详细信息，请参阅 [功能建议说明](#)。

另请参阅

- [C# 参考](#)
- [C# 运算符和表达式](#)
- [?. 和 ?\[\] 运算符](#)

- ?: 运算符

=> 运算符 (C# 参考)

2021/5/10 • [Edit Online](#)

=> 令牌支持两种形式：作为 [lambda 运算符](#)、作为成员名称的分隔符和[表达式主体定义](#)中的成员实现。

lambda 运算符

在 [lambda 表达式](#) 中，lambda 运算符 `=>` 将左侧的输入参数与右侧的 lambda 主体分开。

以下示例使用带有方法语法的 [LINQ](#) 功能来演示 lambda 表达式的用法：

```
string[] words = { "bot", "apple", "apricot" };
int minimalLength = words
    .Where(w => w.StartsWith("a"))
    .Min(w => w.Length);
Console.WriteLine(minimalLength); // output: 5

int[] numbers = { 4, 7, 10 };
int product = numbers.Aggregate(1, (interim, next) => interim * next);
Console.WriteLine(product); // output: 280
```

lambda 表达式的输入参数在编译时是强类型。当编译器可以推断输入参数的类型时，如前面的示例所示，可以省略类型声明。如果需要指定输入参数的类型，则必须对每个参数执行类型声明，如以下示例所示：

```
int[] numbers = { 4, 7, 10 };
int product = numbers.Aggregate(1, (int interim, int next) => interim * next);
Console.WriteLine(product); // output: 280
```

以下示例显示如何在没有输入参数的情况下定义 lambda 表达式：

```
Func<string> greet = () => "Hello, World!";
Console.WriteLine(greet());
```

有关详细信息，请参阅 [Lambda 表达式](#)。

表达式主体定义

表达式主体定义具有下列常规语法：

```
member => expression;
```

其中 `expression` 是有效的表达式。`expression` 的返回类型必须可隐式转换为成员的返回类型。如果成员的返回类型为 `void`，或者如果成员是构造函数、终结器、属性或索引器 `set` 访问器，则 `expression` 必须为语句表达式。由于表达式的结果被丢弃，该表达式的返回类型可以是任何类型。

以下示例演示了用于 `Person.ToString()` 方法的表达式主体定义：

```
public override string ToString() => $"{fname} {lname}".Trim();
```

它是以下方法定义的简写版：

```
public override string ToString()
{
    return $"{fname} {lname}".Trim();
}
```

自 C#6 起，支持方法、运算符和只读属性的表达式主体定义。自 C# 7.0 起，支持构造函数、终结器、属性和索引器访问器的表达式主体定义。

有关详细信息，请参阅 [“Expression-bodied 成员”](#)。

运算符可重载性

不能重载 `=>` 运算符。

C# 语言规范

有关 lambda 运算符的详细信息，请参阅 [C# 语言规范中的匿名函数表达式](#)部分。

请参阅

- [C# 参考](#)
- [C# 运算符和表达式](#)

:: 运算符 (C# 参考)

2021/5/10 • [Edit Online](#)

使用命名空间别名限定符 `::` 访问已设置别名的命名空间的成员。只能使用两个标识符之间的 `::` 限定符。左侧标识符可以是以下任意别名：

- 使用 [using 别名指令](#) 创建的命名空间别名：

```
using forwinforms = System.Drawing;
using forwpf = System.Windows;

public class Converters
{
    public static forwpf::Point Convert(forwinforms::Point point) => new forwpf::Point(point.X,
    point.Y);
}
```

- 外部别名。

- `global` 别名，该别名是全局命名空间别名。全局命名空间是包含未在命名空间中声明的命名空间和类型的命名空间。与 `::` 限定符一起使用时，`global` 别名始终引用全局命名空间，即使存在用户定义的 `global` 命名空间别名也是如此。

以下示例使用 `global` 别名访问 .NET `System` 命名空间，该命名空间是全局命名空间的成员。如果没有 `global` 别名，则将访问用户定义的 `System` 命名空间（该命名空间是 `MyCompany.MyProduct` 命名空间的成员）：

```
namespace MyCompany.MyProduct.System
{
    class Program
    {
        static void Main() => global::System.Console.WriteLine("Using global alias");
    }

    class Console
    {
        string Suggestion => "Consider renaming this class";
    }
}
```

NOTE

仅当 `global` 关键字是 `::` 限定符的左侧标识符时，该关键字才是全局命名空间别名。

此外，还可以使用 `.` 令牌来访问设置了别名的命名空间的成员。不过，`.` 令牌还可用于访问类型成员。`::` 限定符确保其左侧标识符始终引用命名空间别名，即使存在同名的类型或命名空间也是如此。

C# 语言规范

有关详细信息，请参阅 [C# 语言规范](#) 中的 [命名空间别名限定符](#) 部分。

请参阅

- [C# 参考](#)
- [C# 运算符和表达式](#)
- [using 命名空间](#)

await 运算符 (C# 参考)

2021/5/10 • [Edit Online](#)

`await` 运算符暂停对其所属的 `async` 方法的求值，直到其操作数表示的异步操作完成。异步操作完成后，`await` 运算符将返回操作的结果(如果有)。当 `await` 运算符应用到表示已完成操作的操作数时，它将立即返回操作的结果，而不会暂停其所属的方法。`await` 运算符不会阻止计算异步方法的线程。当 `await` 运算符暂停其所属的异步方法时，控件将返回到方法的调用方。

在下面的示例中，`HttpClient.GetByteArrayAsync` 方法返回 `Task<byte[]>` 实例，该实例表示在完成时生成字节数组的异步操作。在操作完成之前，`await` 运算符将暂停 `DownloadDocs MainPageAsync` 方法。当 `DownloadDocs MainPageAsync` 暂停时，控件将返回到 `Main` 方法，该方法是 `DownloadDocs MainPageAsync` 的调用方。`Main` 方法将执行，直至它需要 `DownloadDocs MainPageAsync` 方法执行的异步操作的结果。当 `GetByteArrayAsync` 获取所有字节时，将计算 `DownloadDocs MainPageAsync` 方法的其余部分。之后，将计算 `Main` 方法的其余部分。

```
using System;
using System.Net.Http;
using System.Threading.Tasks;

public class AwaitOperator
{
    public static async Task Main()
    {
        Task<int> downloading = DownloadDocsMainPageAsync();
        Console.WriteLine($"{nameof(Main)}: Launched downloading.");

        int bytesLoaded = await downloading;
        Console.WriteLine($"{nameof(Main)}: Downloaded {bytesLoaded} bytes.");
    }

    private static async Task<int> DownloadDocsMainPageAsync()
    {
        Console.WriteLine($"{nameof(DownloadDocsMainPageAsync)}: About to start downloading.");

        var client = new HttpClient();
        byte[] content = await client.GetByteArrayAsync("https://docs.microsoft.com/en-us/");

        Console.WriteLine($"{nameof(DownloadDocsMainPageAsync)}: Finished downloading.");
        return content.Length;
    }
}
// Output similar to:
// DownloadDocsMainPageAsync: About to start downloading.
// Main: Launched downloading.
// DownloadDocsMainPageAsync: Finished downloading.
// Main: Downloaded 27700 bytes.
```

上一个示例使用异步 `Main` 方法，该方法从 C# 7.1 开始可用。有关详细信息，请参阅 [Main 方法中的 await 运算符部分](#)。

NOTE

有关异步编程的介绍，请参阅[使用 async 和 await 的异步编程](#)。利用 `async` 和 `await` 的异步编程遵循[基于任务的异步模式](#)。

只能在通过 `async` 关键字修改的方法、[lambda 表达式](#)或[匿名方法](#)中使用 `await` 运算符。在异步方法中，不能在

同步函数的主体、`lock` 语句块内以及不安全的上下文中使用 `await` 运算符。

`await` 运算符的操作数通常是一个 .NET 类型：`Task`、`Task<TResult>`、`ValueTask` 或 `ValueTask<TResult>`。但是，任何可等待表达式都可以是 `await` 运算符的操作数。有关详细信息，请参阅 [C# 语言规范中的可等待表达式](#) 部分。

如果表达式 `t` 的类型为 `Task<TResult>` 或 `ValueTask<TResult>`，则表达式 `await t` 的类型为 `TResult`。如果 `t` 的类型为 `Task` 或 `ValueTask`，则 `await t` 的类型为 `void`。在这两种情况下，如果 `t` 引发异常，则 `await t` 将重新引发异常。有关如何处理异常的详细信息，请参阅 [try-catch 语句](#) 一文中的 [异步方法中的异常](#) 部分。

`async` 和 `await` 关键字在 C# 5 和更高版本中都可用。

异步流和可释放对象

从 C# 8.0 开始，可以使用异步流和可释放对象。

可使用 `await foreach` 语句来使用异步数据流。有关详细信息，请参阅 [foreach 语句](#) 文章，和 [C# 8.0 新增功能](#) 文章的 [异步流](#) 一节。

可使用 `await using` 语句来处理异步可释放对象，即其类型可实现 `IAsyncDisposable` 接口的对象。有关详细信息，请参阅 [实现 DisposeAsync 方法](#) 一文中的 [使用异步可释放对象](#) 部分。

Main 方法中的 await 运算符

从 C# 7.1 开始，作为应用程序入口点的 `Main` 方法可以返回 `Task` 或 `Task<int>`，使其成为异步的，以便在其主体中使用 `await` 运算符。在较早的 C# 版本中，为了确保 `Main` 方法等待异步操作完成，可以检索由相应的异步方法返回的 `Task<TResult>` 实例的 `Task<TResult>.Result` 属性值。对于不生成值的异步操作，可以调用 `Task.Wait` 方法。有关如何选择语言版本的信息，请参阅 [C# 语言版本管理](#)。

C# 语言规范

有关详细信息，请参阅 [C# 语言规范](#) 中的 [Await 表达式](#) 部分。

请参阅

- [C# 参考](#)
- [C# 运算符和表达式](#)
- [async](#)
- [任务异步编程模型](#)
- [异步编程](#)
- [深入了解异步](#)
- [演练：使用 async 和 await 访问 Web](#)
- [教程：使用 C# 8.0 和 .NET Core 3.0 生成和使用异步流](#)

default value 表达式 (C# 参考)

2021/5/10 • [Edit Online](#)

default value 表达式生成类型的默认值。有两种类型的 default value 表达式: default 运算符调用和 default 文本。

你还可以将 `default` 关键字用作 `switch` 语句中的默认用例标签。

default 运算符

`default` 运算符的实参必须是类型或类型形参的名称, 如以下示例所示:

```
Console.WriteLine(default(int)); // output: 0
Console.WriteLine(default(object) is null); // output: True

void DisplayDefaultOf<T>()
{
    var val = default(T);
    Console.WriteLine($"Default value of {typeof(T)} is {(val == null ? "null" : val.ToString())}.");
}

DisplayDefaultOf<int?>();
DisplayDefaultOf<System.Numerics.Complex>();
DisplayDefaultOf<System.Collections.Generic.List<int>>();
// Output:
// Default value of System.Nullable`1[System.Int32] is null.
// Default value of System.Numerics.Complex is (0, 0).
// Default value of System.Collections.Generic.List`1[System.Int32] is null.
```

default 文本

从 C# 7.1 开始, 当编译器可以推断表达式类型时, 可以使用 `default` 文本生成类型的默认值。`default` 文本表达式生成与 `default(T)` 表达式(其中, `T` 是推断的类型)相同的值。可以在以下任一情况下使用 `default` 文本:

- 对变量进行赋值或初始化时。
- 在声明可选方法参数的默认值时。
- 在方法调用中提供参数值时。
- 在 `return` 语句中或作为表达式主体成员中的表达式时。

下面的示例演示 `default` 文本的用法:

```
T[] InitializeArray<T>(int length, T initialValue = default)
{
    if (length < 0)
    {
        throw new ArgumentOutOfRangeException(nameof(length), "Array length must be nonnegative.");
    }

    var array = new T[length];
    for (var i = 0; i < length; i++)
    {
        array[i] = initialValue;
    }
    return array;
}

void Display<T>(T[] values) => Console.WriteLine($"[ {string.Join(", ", values)} ]");

Display(InitializeArray<int>(3)); // output: [ 0, 0, 0 ]
Display(InitializeArray<bool>(4, default)); // output: [ False, False, False, False ]

System.Numerics.Complex fillValue = default;
Display(InitializeArray(3, fillValue)); // output: [ (0, 0), (0, 0), (0, 0) ]
```

C# 语言规范

有关详细信息，请参阅 [C# 语言规范](#) 中的 [默认值表达式](#) 部分。

有关 `default` 文本的详细信息，请参阅 [功能建议说明](#)。

请参阅

- [C# 参考](#)
- [C# 运算符和表达式](#)
- [C# 类型的默认值](#)
- [.NET 中的泛型](#)

delegate 运算符 - (C# 参考)

2021/5/10 • [Edit Online](#)

`delegate` 运算符创建一个可以转换为委托类型的匿名方法：

```
Func<int, int, int> sum = delegate (int a, int b) { return a + b; };
Console.WriteLine(sum(3, 4)); // output: 7
```

NOTE

从 C# 3 开始，lambda 表达式提供了一种更简洁和富有表现力的方式来创建匿名函数。使用 [=> 运算符](#) 构造 lambda 表达式：

```
Func<int, int, int> sum = (a, b) => a + b;
Console.WriteLine(sum(3, 4)); // output: 7
```

有关 lambda 表达式功能的更多信息（例如，如何捕获外部变量），请参阅 [lambda 表达式](#)。

使用 `delegate` 运算符时，可以省略参数列表。如果这样做，可以将创建的匿名方法转换为具有任何参数列表的委托类型，如以下示例所示：

```
Action greet = delegate { Console.WriteLine("Hello!"); };
greet();

Action<int, double> introduce = delegate { Console.WriteLine("This is world!"); };
introduce(42, 2.7);

// Output:
// Hello!
// This is world!
```

这是 lambda 表达式不支持的匿名方法的唯一功能。在所有其他情况下，lambda 表达式是编写内联代码的首选方法。

从 C# 9.0 开始，可以使用 [弃元](#) 指定该方法不使用的两个或更多个匿名方法输入参数：

```
Func<int, int, int> constant = delegate (int _, int _) { return 42; };
Console.WriteLine(constant(3, 4)); // output: 42
```

为实现向后兼容性，如果只有一个参数名为 `_`，则将 `_` 视为匿名方法中该参数的名称。

从 C# 9.0 开始，可以在匿名方法的声明中使用 `static` 修饰符：

```
Func<int, int, int> sum = static delegate (int a, int b) { return a + b; };
Console.WriteLine(sum(10, 4)); // output: 14
```

静态匿名方法无法从封闭范围捕获局部变量或实例状态。

还可以使用 `delegate` 关键字声明 [委托类型](#)。

C# 语言规范

有关详细信息，请参阅 [C# 语言规范](#) 中的 [匿名函数表达式](#) 部分。

另请参阅

- [C# 参考](#)
- [C# 运算符和表达式](#)
- [=> 运算符](#)

is 运算符 (C# 参考)

2021/5/10 • [Edit Online](#)

在 C# 6 及更低版本中，`is` 运算符检查表达式的结果是否与给定的类型相匹配。有关类型测试 `is` 运算符的信息，请参阅文章[类型测试和强制转换运算符](#)的 `is` 运算符部分。

从 C# 7.0 开始，还可使用 `is` 运算符将表达式与模式相匹配，如下例所示：

```
static bool IsFirstSummerMonday(DateTime date) => date is { Month: 6, Day: <=7, DayOfWeek: DayOfWeek.Monday };
```

在前面的示例中，`is` 运算符将表达式与[关系模式](#)和带有嵌套常量的[属性模式](#)相匹配。

`is` 运算符在以下应用场景中很有用：

- 检查表达式的运行时类型，如下例所示：

```
int i = 34;
object iBoxed = i;
int? jNullable = 42;
if (iBoxed is int a && jNullable is int b)
{
    Console.WriteLine(a + b); // output 76
}
```

前面的示例演示[声明模式](#)的用法。

- 检查是否为 `null`，如下例所示：

```
if (input is null)
{
    return;
}
```

将表达式与 `null` 匹配时，编译器保证不会调用用户重载的 `==` 或 `!=` 运算符。

- 从 C# 9.0 开始，可使用[否定模式](#)执行非 `null` 检查，如下例所示：

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

有关 `is` 运算符支持的模式的完整列表，请参阅[模式](#)。

C# 语言规范

有关详细信息，请参阅[C# 语言规范](#)的 `is` 运算符部分以及下面的 C# 语言建议：

- [模式匹配](#)
- [使用泛型的模式匹配](#)

另请参阅

- [C# 参考](#)
- [C# 运算符和表达式](#)
- [模式](#)
- [教程: 使用模式匹配来构建类型驱动和数据驱动的算法](#)
- [类型测试和强制转换运算符](#)

nameof 表达式 (C# 参考)

2021/5/10 • [Edit Online](#)

nameof 表达式可生成变量、类型或成员的名称作为字符串常量：

```
Console.WriteLine(nameof(System.Collections.Generic)); // output: Generic
Console.WriteLine(nameof(List<int>)); // output: List
Console.WriteLine(nameof(List<int>.Count)); // output: Count
Console.WriteLine(nameof(List<int>.Add)); // output: Add

var numbers = new List<int> { 1, 2, 3 };
Console.WriteLine(nameof(numbers)); // output: numbers
Console.WriteLine(nameof(numbers.Count)); // output: Count
Console.WriteLine(nameof(numbers.Add)); // output: Add
```

如前面的示例所示，对于类型和命名空间，生成的名称不是完全限定的名称。

在逐字标识符的情况下，@ 字符不是名称的一部分，如以下示例所示：

```
var @new = 5;
Console.WriteLine(nameof(@new)); // output: new
```

nameof 表达式在编译时进行求值，在运行时无效。

可以使用 nameof 表达式使参数检查代码更易于维护：

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

nameof 表达式在 C# 6 及更高版本中提供。

C# 语言规范

有关详细信息，请参阅 [C# 语言规范](#) 中的 [Nameof 表达式](#) 部分。

请参阅

- [C# 参考](#)
- [C# 运算符和表达式](#)

new 运算符 (C# 参考)

2021/5/10 • [Edit Online](#)

`new` 运算符创建类型的新实例。

`new` 关键字还可用作[成员声明修饰符](#)或[泛型类型约束](#)。

构造函数调用

要创建类型的新实例，通常使用 `new` 运算符调用该类型的某个[构造函数](#)：

```
var dict = new Dictionary<string, int>();
dict["first"] = 10;
dict["second"] = 20;
dict["third"] = 30;

Console.WriteLine(string.Join(" ", dict.Select(entry => $"{entry.Key}: {entry.Value}")));
// Output:
// first: 10; second: 20; third: 30
```

可以使用带有 `new` 运算符的[对象或集合初始值设定项](#)实例化和初始化一个语句中的对象，如下例所示：

```
var dict = new Dictionary<string, int>
{
    ["first"] = 10,
    ["second"] = 20,
    ["third"] = 30
};

Console.WriteLine(string.Join(" ", dict.Select(entry => $"{entry.Key}: {entry.Value}")));
// Output:
// first: 10; second: 20; third: 30
```

从 C# 9.0 开始，构造调用表达式由目标确定类型。也就是说，如果已知表达式的目标类型，则可以省略类型名称，如下面的示例所示：

```
List<int> xs = new();
List<int> ys = new(capacity: 10_000);
List<int> zs = new() { Capacity = 20_000 };

Dictionary<int, List<int>> lookup = new()
{
    [1] = new() { 1, 2, 3 },
    [2] = new() { 5, 8, 3 },
    [5] = new() { 1, 0, 4 }
};
```

如前面的示例所示，在由目标确定类型的 `new` 表达式中始终使用括号。

如果 `new` 表达式的目标类型未知(例如使用 `var` 关键字时)，必须指定类型名称。

数组创建

还可以使用 `new` 运算符创建数组实例，如下例所示：

```
var numbers = new int[3];
numbers[0] = 10;
numbers[1] = 20;
numbers[2] = 30;

Console.WriteLine(string.Join(", ", numbers));
// Output:
// 10, 20, 30
```

使用数组初始化语法创建数组实例，并在一个语句中使用元素填充该实例。以下示例显示可以执行该操作的各种方法：

```
var a = new int[3] { 10, 20, 30 };
var b = new int[] { 10, 20, 30 };
var c = new[] { 10, 20, 30 };
Console.WriteLine(c.GetType()); // output: System.Int32[]
```

有关数组的详细信息，请参阅[数组](#)。

匿名类型的实例化

要创建匿名类型的实例，请使用 `new` 运算符和对象初始值设定项语法：

```
var example = new { Greeting = "Hello", Name = "World" };
Console.WriteLine($"{example.Greeting}, {example.Name}!");
// Output:
// Hello, World!
```

类型实例的析构

无需销毁此前创建的类型实例。引用和值类型的实例将自动销毁。包含值类型的上下文销毁后，值类型的实例随之销毁。在引用类型的最后一次引用被删除后，[垃圾回收器](#)会在某个非指定的时间销毁其实例。

对于包含非托管资源的类型实例（例如，文件句柄），建议采用确定性的清理来确保尽快释放其包含的资源。有关详细信息，请参阅[System.IDisposable API 参考](#)和[using 语句](#)一文。

运算符可重载性

用户定义的类型不能重载 `new` 运算符。

C# 语言规范

有关详细信息，请参阅[C# 语言规范的 new 运算符部分](#)。

有关条件由目标确定类型的 `new` 表达式的详细信息，请参阅[功能建议说明](#)。

另请参阅

- [C# 参考](#)
- [C# 运算符和表达式](#)
- [对象和集合初始值设定项](#)

sizeof 运算符 (C# 参考)

2021/5/10 • [Edit Online](#)

`sizeof` 运算符返回给定类型的变量所占用的字节数。`sizeof` 运算符的参数必须是一个非托管类型的名称，或是一个限定为非托管类型的类型参数。

`sizeof` 运算符需要不安全上下文。但下表中的表达式在编译时被计算为相应的常数值，并不需要“不安全”的上下文：

EXPRESSION	SIZE
<code>sizeof(sbyte)</code>	1
<code>sizeof(byte)</code>	1
<code>sizeof(short)</code>	2
<code>sizeof(ushort)</code>	2
<code>sizeof(int)</code>	4
<code>sizeof(uint)</code>	4
<code>sizeof(long)</code>	8
<code>sizeof(ulong)</code>	8
<code>sizeof(char)</code>	2
<code>sizeof(float)</code>	4
<code>sizeof(double)</code>	8
<code>sizeof(decimal)</code>	16
<code>sizeof(bool)</code>	1

下列情况也不需要使用不安全的上下文：`sizeof` 运算符的操作数是枚举类型的名称。

下面的示例演示 `sizeof` 运算符的用法：

```
using System;

public struct Point
{
    public Point(byte tag, double x, double y) => (Tag, X, Y) = (tag, x, y);

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

public class SizeOfOperator
{
    public static void Main()
    {
        Console.WriteLine(sizeof(byte)); // output: 1
        Console.WriteLine(sizeof(double)); // output: 8

        DisplaySizeOf<Point>(); // output: Size of Point is 24
        DisplaySizeOf<decimal>(); // output: Size of System.Decimal is 16

        unsafe
        {
            Console.WriteLine(sizeof(Point*)); // output: 8
        }
    }

    static unsafe void DisplaySizeOf<T>() where T : unmanaged
    {
        Console.WriteLine($"Size of {typeof(T)} is {sizeof(T)}");
    }
}
```

`sizeof` 运算符返回公共语言运行时将在托管内存中分配的字节数。对于[结构](#)类型，该值包括了填充(如有)，如前例所示。`sizeof` 运算符的结果可能异于 [Marshal.SizeOf](#) 方法的结果，该方法返回某个类型在[非托管](#)内存中的大小。

C# 语言规范

有关详细信息，请参阅 [C# 语言规范](#) 的 [sizeof 运算符](#) 部分。

另请参阅

- [C# 参考](#)
- [C# 运算符和表达式](#)
- [指针相关的运算符](#)
- [指针类型](#)
- [内存和跨度相关类型](#)
- [.NET 中的泛型](#)

stackalloc 表达式 (C# 参考)

2021/5/10 • [Edit Online](#)

`stackalloc` 表达式在堆栈上分配内存块。该方法返回时，将自动丢弃在方法执行期间创建的堆栈中分配的内存块。不能显式释放使用 `stackalloc` 分配的内存。堆栈中分配的内存块不受[垃圾回收](#)的影响，也不必通过 `fixed` 语句固定。

可以将 `stackalloc` 表达式的结果分配给以下任一类型的变量：

- 从 C# 7.2 开始，`System.Span<T>` 或 `System.ReadOnlySpan<T>` 将如下例所示：

```
int length = 3;
Span<int> numbers = stackalloc int[length];
for (var i = 0; i < length; i++)
{
    numbers[i] = i;
}
```

将堆栈中分配的内存块分配给 `Span<T>` 或 `ReadOnlySpan<T>` 变量时，不必使用 `unsafe` 上下文。

使用这些类型时，可以在[条件表达式](#)或赋值表达式中使用 `stackalloc` 表达式，如以下示例所示：

```
int length = 1000;
Span<byte> buffer = length <= 1024 ? stackalloc byte[length] : new byte[length];
```

从 C#8.0 开始，只要允许使用 `Span<T>` 或 `ReadOnlySpan<T>` 变量，就可以在其他表达式中使用 `stackalloc` 表达式，如下例所示：

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

NOTE

建议尽可能使用 `Span<T>` 或 `ReadOnlySpan<T>` 类型来处理堆栈中分配的内存。

- 指针类型，如以下示例所示：

```
unsafe
{
    int length = 3;
    int* numbers = stackalloc int[length];
    for (var i = 0; i < length; i++)
    {
        numbers[i] = i;
    }
}
```

如前面的示例所示，在使用指针类型时必须使用 `unsafe` 上下文。

对于指针类型，只能在局部变量声明中使用 `stackalloc` 表达式来初始化变量。

堆栈上可用的内存量存在限制。如果在堆栈上分配过多的内存，会引发 [StackOverflowException](#)。为避免这种情况，请遵循以下规则：

- 限制使用 `stackalloc` 分配的内存量：

```
const int MaxStackLimit = 1024;
Span<byte> buffer = inputLength <= MaxStackLimit ? stackalloc byte[inputLength] : new
byte[inputLength];
```

由于堆栈上可用的内存量取决于执行代码的环境，因此在定义实际限值时请持保守态度。

- 避免在循环内使用 `stackalloc`。在循环外分配内存块，然后在循环内重用它。

新分配的内存的内容未定义。在使用之前应对其进行初始化。例如，可以使用 `Span<T>.Clear` 方法将所有项设置为 `T` 类型的默认值。

从 C# 7.3 开始，可以使用数组初始值设定项语法来定义新分配内存的内容。下面的示例演示执行此操作的各种方法：

```
Span<int> first = stackalloc int[3] { 1, 2, 3 };
Span<int> second = stackalloc int[] { 1, 2, 3 };
ReadOnlySpan<int> third = stackalloc[] { 1, 2, 3 };
```

在表达式 `stackalloc T[E]` 中，`T` 必须是[非托管类型](#)，并且 `E` 的计算结果必须为非负 `int` 值。

安全性

使用 `stackalloc` 会自动启用公共语言运行时 (CLR) 中的缓冲区溢出检测功能。如果检测到缓冲区溢出，则将尽快终止进程，以便将执行恶意代码的可能性降到最低。

C# 语言规范

有关详细信息，请参阅 [C# 语言规范](#) 以及允许嵌入上下文中的 `stackalloc` 功能建议说明的[堆栈分配](#)部分。

请参阅

- [C# 参考](#)
- [C# 运算符和表达式](#)
- [指针相关的运算符](#)
- [指针类型](#)
- [内存和跨度相关类型](#)
- [stackalloc 注意事项](#)

switch 表达式 (C# 参考)

2021/5/10 • [Edit Online](#)

从 C# 8.0 开始，可以使用 `switch` 表达式，根据与输入表达式匹配的模式，对候选表达式列表中的单个表达式进行求值。有关在语句上下文中支持 `switch` 类语义的 `switch` 语句的信息，请参阅 [switch 语句](#) 一文。

下面的示例演示了一个 `switch` 表达式，该表达式将在线地图中表示视觉方向的 `enum` 中的值转换为相应的基本方位：

```
public static class SwitchExample
{
    public enum Direction
    {
        Up,
        Down,
        Right,
        Left
    }

    public enum Orientation
    {
        North,
        South,
        East,
        West
    }

    public static Orientation ToOrientation(Direction direction) => direction switch
    {
        Direction.Up     => Orientation.North,
        Direction.Right => Orientation.East,
        Direction.Down   => Orientation.South,
        Direction.Left   => Orientation.West,
        _                => throw new ArgumentOutOfRangeException(nameof(direction), $"Not expected direction value: {direction}"),
    };

    public static void Main()
    {
        var direction = Direction.Right;
        Console.WriteLine($"Map view direction is {direction}");
        Console.WriteLine($"Cardinal orientation is {ToOrientation(direction)}");
        // Output:
        // Map view direction is Right
        // Cardinal orientation is East
    }
}
```

上述示例展示了 `switch` 表达式的基本元素：

- 后跟 `switch` 关键字的表达式。在上述示例中，这是 `direction` 方法参数。
- `switch` expression arm，用逗号分隔。每个 `switch` expression arm 都包含一个模式、一个可选的 `case guard`、`=>` 标记和一个表达式。

在上述示例中，`switch` 表达式使用以下模式：

- 常数模式**，用于处理 `Direction` 枚举的定义值。
- 弃元模式**用于处理没有相应的 `Direction` 枚举成员的任何整数值(例如 `(Direction)10`)。这会使 `switch` 表

达式详尽。

有关 `switch` 表达式支持的模式的信息, 请参阅[模式](#)。

`switch` 表达式的结果是第一个 `switch` expression arm 的表达式的值, 该 `switch` expression arm 的模式与范围表达式匹配, 并且它的 case guard(如果存在)求值为 `true`。`switch` expression arm 按文本顺序求值。

如果无法选择较低的 `switch` expression arm, 编译器会发出错误, 因为较高的 `switch` expression arm 匹配其所有值。

Case guard

模式或许表现力不够, 无法指定用于计算 arm 的表达式的条件。在这种情况下, 可以使用 case guard。这是一个附加条件, 必须与匹配模式同时满足。可以在模式后面的 `when` 关键字之后指定一个 case guard, 如以下示例所示:

```
public readonly struct Point
{
    public Point(int x, int y) => (X, Y) = (x, y);

    public int X { get; }
    public int Y { get; }
}

static Point Transform(Point point) => point switch
{
    { X: 0, Y: 0 }                  => new Point(0, 0),
    { X: var x, Y: var y } when x < y => new Point(x + y, y),
    { X: var x, Y: var y } when x > y => new Point(x - y, y),
    { X: var x, Y: var y }          => new Point(2 * x, 2 * y),
};
```

上述示例使用带有嵌套 `var` 模式的属性模式。

非详尽的 `switch` 表达式

如果 `switch` 表达式的模式均未捕获输入值, 则运行时将引发异常。在 .NET Core 3.0 及更高版本中, 异常是 `System.Runtime.CompilerServices.SwitchExpressionException`。在 .NET Framework 中, 异常是 `InvalidOperationException`。如果 `switch` 表达式未处理所有可能的输入值, 则编译器会生成警告。

TIP

为了保证 `switch` 表达式处理所有可能的输入值, 请为 `switch` expression arm 提供弃元模式。

C# 语言规范

有关详细信息, 请参阅[功能建议说明的 `switch` 表达式部分](#)。

另请参阅

- [C# 参考](#)
- [C# 运算符和表达式](#)
- [模式](#)
- [教程: 使用模式匹配来构建类型驱动和数据驱动的算法](#)
- [switch 语句](#)

true 和 false 运算符 (C# 参考)

2021/5/10 • [Edit Online](#)

`true` 运算符返回 `bool` 值 `true`，以指明其操作数一定为 `true`。`false` 运算符返回 `bool` 值 `false`，以指明其操作数一定为 `false`。无法确保 `true` 和 `false` 运算符互补。也就是说，`true` 和 `false` 运算符可能同时针对同一个操作数返回 `bool` 值 `false`。如果某类型定义这两个运算符之一，它还必须定义另一个运算符。

TIP

如需支持三值逻辑(例如，在使用支持三值布尔类型的数据库时)，请使用 `bool?` 类型。C# 提供 `&` 和 `|` 运算符，它们通过 `bool?` 操作数支持三值逻辑。有关详细信息，请参阅[布尔逻辑运算符](#)一文的[可以为 null 的布尔逻辑运算符](#)部分。

布尔表达式

包含已定义 `true` 运算符的类型可以是 `if`、`do`、`while` 和 `for` 语句以及[条件运算符](#) `?:` 中控制条件表达式的结果的类型。有关详细信息，请参阅[C# 语言规范](#)中的[Boolean 表达式](#)部分。

用户定义的条件逻辑运算符

如果包含已定义 `true` 和 `false` 运算符的类型以某种方式重载逻辑 OR 运算符 `|` 或逻辑 AND 运算符 `&`，可以对相应类型的操作数分别执行[条件逻辑 OR 运算符](#) `||` 或[条件逻辑 AND 运算符](#) `&&` 运算。有关详细信息，请参阅[C# 语言规范](#)的[用户定义条件逻辑运算符](#)部分。

示例

下面的示例演示了定义 `true` 和 `false` 运算符的类型。此外，该类型还重载了逻辑 AND 运算符 `&`，因此，也可以对相应类型的操作数计算运算符 `&&`。

```

using System;

public struct LaunchStatus
{
    public static readonly LaunchStatus Green = new LaunchStatus(0);
    public static readonly LaunchStatus Yellow = new LaunchStatus(1);
    public static readonly LaunchStatus Red = new LaunchStatus(2);

    private int status;

    private LaunchStatus(int status)
    {
        this.status = status;
    }

    public static bool operator true(LaunchStatus x) => x == Green || x == Yellow;
    public static bool operator false(LaunchStatus x) => x == Red;

    public static LaunchStatus operator &(LaunchStatus x, LaunchStatus y)
    {
        if (x == Red || y == Red || (x == Yellow && y == Yellow))
        {
            return Red;
        }

        if (x == Yellow || y == Yellow)
        {
            return Yellow;
        }

        return Green;
    }

    public static bool operator ==(LaunchStatus x, LaunchStatus y) => x.status == y.status;
    public static bool operator !=(LaunchStatus x, LaunchStatus y) => !(x == y);

    public override bool Equals(object obj) => obj is LaunchStatus other && this == other;
    public override int GetHashCode() => status;
}

public class LaunchStatusTest
{
    public static void Main()
    {
        LaunchStatus okToLaunch = GetFuelLaunchStatus() && GetNavigationLaunchStatus();
        Console.WriteLine(okToLaunch ? "Ready to go!" : "Wait!");
    }

    static LaunchStatus GetFuelLaunchStatus()
    {
        Console.WriteLine("Getting fuel launch status...");
        return LaunchStatus.Red;
    }

    static LaunchStatus GetNavigationLaunchStatus()
    {
        Console.WriteLine("Getting navigation launch status...");
        return LaunchStatus.Yellow;
    }
}

```

请注意 `&&` 运算符的短路行为。当 `GetFuelLaunchStatus` 方法返回 `LaunchStatus.Red` 时，不会进行计算的 `&&` 运算符的右侧操作数。这是因为 `LaunchStatus.Red` 一定为 `false`。然后，逻辑 AND 运算符的结果不依赖右侧操作数的值。示例的输出如下所示：

```
Getting fuel launch status...
Wait!
```

另请参阅

- [C# 参考](#)
- [C# 运算符和表达式](#)

with 表达式 (C# 参考)

2021/5/10 • [Edit Online](#)

`with` 表达式在 C# 9.0 及更高版本中可用，使用修改的特定属性和字段生成其记录操作数的副本：

```
using System;

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

    public static void Main()
    {
        var p1 = new NamedPoint("A", 0, 0);
        Console.WriteLine($"{nameof(p1)}: {p1}"); // output: p1: NamedPoint { Name = A, X = 0, Y = 0 }

        var p2 = p1 with { Name = "B", X = 5 };
        Console.WriteLine($"{nameof(p2)}: {p2}"); // output: p2: NamedPoint { Name = B, X = 5, Y = 0 }

        var p3 = p1 with
        {
            Name = "C",
            Y = 4
        };
        Console.WriteLine($"{nameof(p3)}: {p3}"); // output: p3: NamedPoint { Name = C, X = 0, Y = 4 }

        Console.WriteLine($"{nameof(p1)}: {p1}"); // output: p1: NamedPoint { Name = A, X = 0, Y = 0 }
    }
}
```

如上一个示例所示，你可以使用[对象初始值设定项](#)语法来指定要修改的成员及其新值。在 `with` 表达式中，左侧操作数必须为记录类型。

如以下示例所示，`with` 表达式结果与表达式操作数的运行时类型相同：

```
using System;

public class InheritanceExample
{
    public record Point(int X, int Y);
    public record NamedPoint(string Name, int X, int Y) : Point(X, Y);

    public static void Main()
    {
        Point p1 = new NamedPoint("A", 0, 0);
        Point p2 = p1 with { X = 5, Y = 3 };
        Console.WriteLine(p2 is NamedPoint); // output: True
        Console.WriteLine(p2); // output: NamedPoint { X = 5, Y = 3, Name = A }

    }
}
```

对于引用类型成员，在复制记录时仅复制对实例的引用。副本和原始记录都具有对同一引用类型实例的访问权限。以下示例演示了该行为：

```

using System;
using System.Collections.Generic;

public class ExampleWithReferenceType
{
    public record TaggedNumber(int Number, List<string> Tags)
    {
        public string PrintTags() => string.Join(", ", Tags);
    }

    public static void Main()
    {
        var original = new TaggedNumber(1, new List<string> { "A", "B" });

        var copy = original with { Number = 2 };
        Console.WriteLine($"Tags of {nameof(copy)}: {copy.PrintTags()}");
        // output: Tags of copy: A, B

        original.Tags.Add("C");
        Console.WriteLine($"Tags of {nameof(copy)}: {copy.PrintTags()}");
        // output: Tags of copy: A, B, C
    }
}

```

任何记录类型都具有复制构造函数。这是一个包含记录类型的单个参数的构造函数。它将参数的状态复制到新的记录实例。在评估 `with` 表达式时，将调用复制构造函数，以基于原始记录实例化新记录实例化。之后，新实例将根据指定的修改进行更新。默认情况下，复制构造函数是隐式的，即编译器生成的。如果需要自定义记录复制语义，请显式声明具有所需行为的复制构造函数。下面的示例使用显式复制构造函数更新前面的示例。复制记录时，新的复制行为是复制列表项而不是列表引用：

```

using System;
using System.Collections.Generic;

public class UserDefinedCopyConstructorExample
{
    public record TaggedNumber(int Number, List<string> Tags)
    {
        protected TaggedNumber(TaggedNumber original)
        {
            Number = original.Number;
            Tags = new List<string>(original.Tags);
        }

        public string PrintTags() => string.Join(", ", Tags);
    }

    public static void Main()
    {
        var original = new TaggedNumber(1, new List<string> { "A", "B" });

        var copy = original with { Number = 2 };
        Console.WriteLine($"Tags of {nameof(copy)}: {copy.PrintTags()}");
        // output: Tags of copy: A, B

        original.Tags.Add("C");
        Console.WriteLine($"Tags of {nameof(copy)}: {copy.PrintTags()}");
        // output: Tags of copy: A, B
    }
}

```

有关详细信息，请参阅[记录功能建议说明](#)的以下部分：

- [with 表达式](#)
- [复制和克隆成员](#)

另请参阅

- [C# 参考](#)
- [C# 运算符和表达式](#)
- [记录](#)

运算符重载 (C# 引用)

2021/3/5 • • [Edit Online](#)

用户定义的类型可重载预定义的 C# 运算符。也就是说，当一个或两个操作数都是某类型时，此类型可提供操作的自定义实现。[可重载运算符](#)部分介绍了哪些 C# 运算符可重载。

使用 `operator` 关键字来声明运算符。运算符声明必须符合以下规则：

- 同时包含 `public` 和 `static` 修饰符。
- 一元运算符有一个输入参数。二元运算符有两个输入参数。在每种情况下，都至少有一个参数必须具有类型 `T` 或 `T?`，其中 `T` 是包含运算符声明的类型。

下面的示例定义了一个表示有理数的简单结构。该结构会重载一些[算术运算符](#)：

```

using System;

public readonly struct Fraction
{
    private readonly int num;
    private readonly int den;

    public Fraction(int numerator, int denominator)
    {
        if (denominator == 0)
        {
            throw new ArgumentException("Denominator cannot be zero.", nameof(denominator));
        }
        num = numerator;
        den = denominator;
    }

    public static Fraction operator +(Fraction a) => a;
    public static Fraction operator -(Fraction a) => new Fraction(-a.num, a.den);

    public static Fraction operator +(Fraction a, Fraction b)
        => new Fraction(a.num * b.den + b.num * a.den, a.den * b.den);

    public static Fraction operator -(Fraction a, Fraction b)
        => a + (-b);

    public static Fraction operator *(Fraction a, Fraction b)
        => new Fraction(a.num * b.num, a.den * b.den);

    public static Fraction operator /(Fraction a, Fraction b)
    {
        if (b.num == 0)
        {
            throw new DivideByZeroException();
        }
        return new Fraction(a.num * b.den, a.den * b.num);
    }

    public override string ToString() => $"{num} / {den}";
}

public static class OperatorOverloading
{
    public static void Main()
    {
        var a = new Fraction(5, 4);
        var b = new Fraction(1, 2);
        Console.WriteLine(-a); // output: -5 / 4
        Console.WriteLine(a + b); // output: 14 / 8
        Console.WriteLine(a - b); // output: 6 / 8
        Console.WriteLine(a * b); // output: 5 / 8
        Console.WriteLine(a / b); // output: 10 / 4
    }
}

```

可以通过[定义从 `int` 到 `Fraction`](#)的隐式转换来扩展前面的示例。然后，重载运算符将支持这两种类型的参数。也就是说，可以将一个整数添加到一个分数中，得到一个分数结果。

还可以使用 `operator` 关键字来定义自定义类型转换。有关详细信息，请参阅[用户定义转换运算符](#)。

可重载运算符

下表提供了 C# 运算符可重载性的相关信息：

一元运算符	二元运算符
<code>+x</code> 、 <code>-x</code> 、 <code>!x</code> 、 <code>~x</code> 、 <code>++</code> 、 <code>--</code> 、 <code>true</code> 、 <code>false</code>	这些一元运算符可以进行重载。
<code>x + y</code> 、 <code>x - y</code> 、 <code>x * y</code> 、 <code>x / y</code> 、 <code>x % y</code> 、 <code>x & y</code> 、 <code>x y</code> 、 <code>x ^ y</code> 、 <code>x << y</code> 、 <code>x >> y</code> 、 <code>x == y</code> 、 <code>x != y</code> 、 <code>x < y</code> 、 <code>x > y</code> 、 <code>x <= y</code> 、 <code>x >= y</code>	这些二元运算符可以进行重载。某些运算符必须成对重载；有关详细信息，请查看此表后面的备注。
<code>x && y</code> 、 <code>x y</code>	无法重载条件逻辑运算符。但是，如果具有已重载的 <code>true</code> 和 <code>false</code> 运算符的类型还以某种方式重载了 <code>&</code> 或 <code> </code> 运算符，则可针对此类型的操作数分别计算 <code>&&</code> 或 <code> </code> 运算符。有关详细信息，请参阅 C# 语言规范的用户定义条件逻辑运算符部分 。
<code>a[i]</code> 、 <code>a?[i]</code>	元素访问不被视为可重载运算符，但你可定义 索引器 。
<code>(T)x</code>	强制转换运算符不能重载，但可以定义可由强制转换表达式执行的自定义类型转换。有关详细信息，请参阅 用户定义转换运算符 。
<code>+=</code> 、 <code>-=</code> 、 <code>*=</code> 、 <code>/=</code> 、 <code>%=</code> 、 <code>&=</code> 、 <code> =</code> 、 <code>^=</code> 、 <code><<=</code> 、 <code>>>=</code>	复合赋值运算符不能显式重载。但在重载二元运算符时，也会隐式重载相应的复合赋值运算符（若有）。例如，使用 <code>+</code> （可以进行重载）计算 <code>+=</code> 。
<code>^x</code> 、 <code>x = y</code> 、 <code>x.y</code> 、 <code>x?.y</code> 、 <code>c ? t : f</code> 、 <code>x ?? y</code> 、 <code>x ??= y</code> 、 <code>x.y</code> 、 <code>x-y</code> 、 <code>=></code> 、 <code>f(x)</code> 、 <code>as</code> 、 <code>await</code> 、 <code>checked</code> 、 <code>unchecked</code> 、 <code>default</code> 、 <code>delegate</code> 、 <code>is</code> 、 <code>nameof</code> 、 <code>new</code> 、 <code>sizeof</code> 、 <code>stackalloc</code> 、 <code>switch</code> 、 <code>typeof</code> 、 <code>with</code>	这些运算符无法进行重载。

NOTE

比较运算符必须成对重载。也就是说，如果重载一对运算符中的任一个，则另一个运算符也必须重载。此类配对如下所示：

- `==` 和 `!=` 运算符
- `<` 和 `>` 运算符
- `<=` 和 `>=` 运算符

C# 语言规范

有关更多信息，请参阅 [C# 语言规范](#) 的以下部分：

- [运算符重载](#)
- [运算符](#)

请参阅

- [C# 参考](#)
- [C# 运算符和表达式](#)
- [用户定义转换运算符](#)
- [设计准则 - 运算符重载](#)
- [设计准则 - 相等运算符](#)
- [重载运算符为何在 C# 中始终是静态的？](#)

C# 特殊字符

2020/11/2 • [Edit Online](#)

特殊字符是预定义的上下文字符，用于修改最前面插入了此类字符的程序元素(文本字符串、标识符或属性名称)。C# 支持以下特殊字符：

- `@`:逐字字符串标识符字符。
- `$`:内插的字符串字符。

请参阅

- [C# 参考](#)
- [C# 编程指南](#)

\$ - 字符串内插 (C# 参考)

2020/5/6 • • [Edit Online](#)

`$` 特殊字符将字符串文本标识为内插字符串。内插字符串是可能包含内插表达式的字符串文本。将内插字符串解析为结果字符串时，带有内插表达式的项会替换为表达式结果的字符串表示形式。从 C# 6 开始可以使用此功能。

与使用[字符串复合格式设置](#)功能创建格式化字符串相比，字符串内插提供的语法更具可读性，且更加方便。下面的示例使用了这两种功能生成同样的输出结果：

```
string name = "Mark";
var date = DateTime.Now;

// Composite formatting:
Console.WriteLine("Hello, {0}! Today is {1}, it's {2:HH:mm} now.", name, date.DayOfWeek, date);
// String interpolation:
Console.WriteLine($"Hello, {name}! Today is {date.DayOfWeek}, it's {date:HH:mm} now.");
// Both calls produce the same output that is similar to:
// Hello, Mark! Today is Wednesday, it's 19:40 now.
```

内插字符串的结构

若要将字符串标识为内插字符串，可在该字符串前面加上 `$` 符号。字符串文本开头的 `$` 和 `"` 之间不能有任何空格。

具备内插表达式的项的结构如下所示：

```
{<interpolationExpression>[,<alignment>][:<formatString>]}
```

括号中的元素是可选的。下表说明了每个元素：

元素	描述
<code>interpolationExpression</code>	生成需要设置格式的结果的表达式。 <code>null</code> 的字符串表示形式为 String.Empty 。
<code>alignment</code>	常数表达式，它的值定义表达式结果的字符串表示形式中的最小字符数。如果值为正，则字符串表示形式为右对齐；如果值为负，则为左对齐。有关详细信息，请参阅 对齐组件 。
<code>formatString</code>	受表达式结果类型支持的格式字符串。有关更多信息，请参阅 格式字符串组件 。

以下示例使用上述可选的格式设置组件：

```
Console.WriteLine($"|{"Left",-7}|{"Right",7}|");

const int FieldWidthRightAligned = 20;
Console.WriteLine($"{Math.PI,FieldWidthRightAligned} - default formatting of the pi number");
Console.WriteLine($"{Math.PI,FieldWidthRightAligned:F3} - display only three decimal digits of the pi
number");
// Expected output is:
// |Left    | Right|
//      3.14159265358979 - default formatting of the pi number
//                      3.142 - display only three decimal digits of the pi number
```

特殊字符

要在内插字符串生成的文本中包含大括号 "{" 或 "}"，请使用两个大括号，即 "{{" 或 "}}。有关详细信息，请参阅[转义大括号](#)。

因为冒号 ":" 在内插表达式项中具有特殊含义，为了在内插表达式中使用[条件运算符](#)，请将表达式放在括号内。

以下示例演示如何将大括号含入结果字符串中，以及如何在内插表达式中使用条件运算符：

```
string name = "Horace";
int age = 34;
Console.WriteLine($"He asked, \"Is your name {name}?\", but didn't wait for a reply :-{{
Console.WriteLine($"{name} is {age} year{(age == 1 ? "" : "s")} old.");
// Expected output is:
// He asked, "Is your name Horace?", but didn't wait for a reply :-
// Horace is 34 years old.
```

内插逐字字符串以 \$ 字符开头，后跟 @ 字符。有关逐字字符串的详细信息，请参阅[字符串和逐字标识符](#)主题。

NOTE

从 C# 8.0 开始，可以按任意顺序使用 \$ 和 @ 标记：\$@"..." 和 @\$"..." 均为有效的内插逐字字符串。在早期 C# 版本中，\$ 标记必须出现在 @ 标记之前。

隐式转换和指定 `IFormatProvider` 实现的方式

内插字符串有 3 种隐式转换：

1. 将内插字符串转换为 `String` 实例，该类例是内插字符串的解析结果，其中内插表达式项被替换为结果的格式设置正确的字符串表示形式。此转换使用 `CurrentCulture` 设置表达式结果的格式。
2. 将内插字符串转换为表示复合格式字符串的 `FormattableString` 实例，同时也将表达式结果格式化。这允许通过单个 `FormattableString` 实例创建多个包含区域性特定内容的结果字符串。要执行此操作，请调用以下方法之一：
 - `ToString()` 重载，生成 `CurrentCulture` 的结果字符串。
 - `Invariant` 方法，生成 `InvariantCulture` 的结果字符串。
 - `ToString(IFormatProvider)` 方法，生成特定区域性的结果字符串。还可以使用 `ToString(IFormatProvider)` 方法，以提供支持自定义格式设置的 `IFormatProvider` 接口的用户定义实现。有关详细信息，请参阅在 .NET 中设置类型格式一文中的[使用 `ICustomFormatter` 进行自定义格式设置](#)部分。
3. 将内插字符串转换为 `IFormattable` 实例，使用此实例也可通过单个 `IFormattable` 实例创建多个包含区域性特定内容的结果字符串。

以下示例通过隐式转换为 [FormattableString](#) 来创建特定于区域性的结果字符串：

```
double speedOfLight = 299792.458;
FormattableString message = $"The speed of light is {speedOfLight:N3} km/s.";

System.Globalization.CultureInfo.CurrentCulture = System.Globalization.CultureInfo.GetCultureInfo("nl-NL");
string messageInCurrentCulture = message.ToString();

var specificCulture = System.Globalization.CultureInfo.GetCultureInfo("en-IN");
string messageInSpecificCulture = message.ToString(specificCulture);

string messageInInvariantCulture = FormattableString.Invariant(message);

Console.WriteLine($"{System.Globalization.CultureInfo.CurrentCulture,-10} {messageInCurrentCulture}");
Console.WriteLine($"{specificCulture,-10} {messageInSpecificCulture}");
Console.WriteLine($"{Invariant,-10} {messageInInvariantCulture}");

// Expected output is:
// nl-NL      The speed of light is 299,792,458 km/s.
// en-IN      The speed of light is 2,99,792.458 km/s.
// Invariant   The speed of light is 299,792.458 km/s.
```

其他资源

如果你不熟悉字符串内插，请参阅 [C# 中的字符串内插](#) 交互式教程。还可查看另一个 [C# 中的字符串内插](#) 教程，该教程演示了如何使用内插字符串生成带格式的字符串。

内插字符串编译

如果内插字符串类型为 `string`，则通常将其转换为 [String.Format](#) 方法调用。如果分析的行为等同于串联，则编译器可将 [String.Format](#) 替换为 [String.Concat](#)。

如果内插字符串类型为 [IFormattable](#) 或 [FormattableString](#)，则编译器会生成对 [FormattableStringFactory.Create](#) 方法的调用。

C# 语言规范

有关详细信息，请参阅 [C# 语言规范](#) 的 [内插字符串](#) 部分。

请参阅

- [C# 参考](#)
- [C# 特殊字符](#)
- [字符串](#)
- [标准数字格式字符串](#)
- [复合格式设置](#)
- [String.Format](#)

@ (C# 参考)

2020/11/2 • [Edit Online](#)

@ 特殊字符用作原义标识符。它具有以下用途：

- 使 C# 关键字用作标识符。@ 字符可作为代码元素的前缀，编译器将把此代码元素解释为标识符而非 C# 关键字。下面的示例使用 @ 字符定义其在 for 循环中使用的名为 for 的标识符。

```
string[] @for = { "John", "James", "Joan", "Jamie" };
for (int ctr = 0; ctr < @for.Length; ctr++)
{
    Console.WriteLine($"Here is your gift, {@for[ctr]}!");
}
// The example displays the following output:
//      Here is your gift, John!
//      Here is your gift, James!
//      Here is your gift, Joan!
//      Here is your gift, Jamie!
```

- 指示将原义解释字符串。@ 字符在此实例中定义原义标识符。简单转义序列(如代表反斜杠的 "\\")、十六进制转义序列(如代表大写字母 A 的 @"\x0041")和 Unicode 转义序列(如代表大写字母 A 的 @"\u0041")都将按字面解释。只有引号转义序列 ("") 不会按字面解释；因为它生成一个双引号。此外，如果是逐字内插字符串，大括号转义序列({{ 和 }})不按字面解释；它们会生成单个大括号字符。下面的示例分别使用常规字符串和原义字符串定义两个相同的文件路径。这是原义字符串的较常见用法之一。

```
string filename1 = @"c:\documents\files\u0066.txt";
string filename2 = "c:\\documents\\files\\u0066.txt";

Console.WriteLine(filename1);
Console.WriteLine(filename2);
// The example displays the following output:
//      c:\documents\files\u0066.txt
//      c:\documents\files\u0066.txt
```

下面的示例演示定义包含相同字符序列的常规字符串和原义字符串的效果。

```
string s1 = "He said, \"This is the last \u0063hance\x0021\"";
string s2 = @"He said, ""This is the last \u0063hance\x0021""";

Console.WriteLine(s1);
Console.WriteLine(s2);
// The example displays the following output:
//      He said, "This is the last chance!"
//      He said, "This is the last \u0063hance\x0021"
```

- 使编译器在命名冲突的情况下区分两种属性。属性是派生自 Attribute 的类。其类型名称通常包含后缀 Attribute，但编译器不会强制进行此转换。随后可在代码中按其完整类型名称(例如 [InfoAttribute])或短名称(例如 [Info])引用此属性。但是，如果两个短名称相同，并且一个类型名称包含 Attribute 后缀而另一类型名称不包含，则会出现命名冲突。例如，由于编译器无法确定将 Info 还是 InfoAttribute 属性应用于 Example 类，因此下面的代码无法编译。有关详细信息，请参阅 CS1614。

```
using System;

[AttributeUsage(AttributeTargets.Class)]
public class Info : Attribute
{
    private string information;

    public Info(string info)
    {
        information = info;
    }
}

[AttributeUsage(AttributeTargets.Method)]
public class InfoAttribute : Attribute
{
    private string information;

    public InfoAttribute(string info)
    {
        information = info;
    }
}

[Info("A simple executable.")] // Generates compiler error CS1614. Ambiguous Info and InfoAttribute.
// Prepend '@' to select 'Info'. Specify the full name 'InfoAttribute' to select it.
public class Example
{
    [InfoAttribute("The entry point.")]
    public static void Main()
    {
    }
}
```

请参阅

- [C# 参考](#)
- [C# 编程指南](#)
- [C# 特殊字符](#)

保留的特性：程序集级别特性

2020/4/23 • [Edit Online](#)

大多数特性应用于特定语言元素，如类或方法；但是，一些特性是全局特性 - 它们应用于整个程序集或模块。例如，[AssemblyVersionAttribute](#) 属性可用于将版本信息嵌入程序集，如下所示：

```
[assembly: AssemblyVersion("1.0.0.0")]
```

全局特性出现在源代码中任何顶级 `using` 指令之后和任何类型、模块或命名空间声明之前。全局特性可以出现在多个源文件中，但必须在单个编译过程中编译这些文件。Visual Studio 将全局特性添加到 .NET Framework 项目中的 `AssemblyInfo.cs` 文件中。这些特性不会添加到 .NET Core 项目中。

程序集特性是提供程序集相关信息的值。它们分为以下几类：

- 程序集标识特性
- 信息性特性
- 程序集清单特性

程序集标识特性

三个特性（与强名称（如果适用））组合起来可以确定程序集的标识：名称、版本和区域性。这些特性构成程序集的全名，在代码中引用程序集时必需使用。可使用特性设置程序集的版本和区域性。但是，创建程序集时，由编译器、[程序集信息对话框](#) 中的 Visual Studio IDE 或程序集链接器（Al.exe）设置名称值。程序集名称基于程序集清单。[AssemblyFlagsAttribute](#) 属性指定程序集的多个副本是否可以共存。

下表显示标识特性。

特性的名称	该特性的功能
AssemblyVersionAttribute	指定程序集的版本。
AssemblyCultureAttribute	指定程序集支持的区域性。
AssemblyFlagsAttribute	指定程序集是否支持在同一计算机上、同一进程中或同一应用程序域中并行执行。

信息性特性

你可使用信息性特性为程序集提供其他公司或产品信息。下表显示 `System.Reflection` 命名空间中定义的信息性属性。

特性的名称	该特性的功能
AssemblyProductAttribute	指定程序集清单的产品名称。
AssemblyTrademarkAttribute	指定程序集清单的商标。
AssemblyInformationalVersionAttribute	为程序集清单指定信息性版本。

AssemblyCompanyAttribute	为程序集清单指定公司名称。
AssemblyCopyrightAttribute	定义为程序集清单指定版权的自定义属性。
AssemblyFileVersionAttribute	设置 Win32 文件版本资源的特定版本号。
CLSCompliantAttribute	指示程序集是否符合公共语言规范 (CLS)。

程序集清单特性

程序集清单特性可用于提供程序集清单中的信息。特性包括标题、说明、默认别名和配置。下表显示 [System.Reflection](#) 命名空间中定义的程序集清单属性。

AssemblyTitleAttribute	为程序集清单指定程序集标题。
AssemblyDescriptionAttribute	为程序集清单指定程序集说明。
AssemblyConfigurationAttribute	为程序集清单指定程序集配置(如零售或调试)。
AssemblyDefaultAliasAttribute	定义程序集清单的友好默认别名

保留的特性：确定调用方信息

2020/4/23 • [Edit Online](#)

使用信息属性，可以获取有关方法调用方的信息。可以获取源代码的文件路径、源代码中的行号和调用方的成员名称。若要获取成员调用方信息，可以使用应用于可选参数的特性。每个可选参数指定一个默认值。下表列出在 `System.Runtime.CompilerServices` 命名空间中定义的调用方信息特性：

特性的名称	描述	返回类型
<code>CallerFilePathAttribute</code>	包含调用方的源文件的完整路径。完整路径是编译时的路径。	<code>String</code>
<code>CallerLineNumberAttribute</code>	源文件中调用方法的行号。	<code>Integer</code>
<code>CallerMemberNameAttribute</code>	调用方的方法名称或属性名称。	<code>String</code>

此信息有助于你编写跟踪、调试和创建诊断工具。下面的示例演示如何使用调用方信息特性。每次调用 `TraceMessage` 方法时，调用方信息将替换为可选参数的变量。

```
public void DoProcessing()
{
    TraceMessage("Something happened.");
}

public void TraceMessage(string message,
    [CallerMemberName] string memberName = "",
    [CallerFilePath] string sourceFilePath = "",
    [CallerLineNumber] int sourceLineNumber = 0)
{
    Trace.WriteLine("message: " + message);
    Trace.WriteLine("member name: " + memberName);
    Trace.WriteLine("source file path: " + sourceFilePath);
    Trace.WriteLine("source line number: " + sourceLineNumber);
}

// Sample Output:
// message: Something happened.
// member name: DoProcessing
// source file path: c:\Visual Studio Projects\CallerInfoCS\CallerInfoCS\Form1.cs
// source line number: 31
```

为每个可选参数指定显式默认值。不能将调用方信息特性应用于未指定为可选的参数。调用方信息特性不会使参数成为可选参数。相反，它们会在忽略此参数时影响传入的默认值。在编译时，调用方信息值将作为文本传入中间语言 (IL)。与异常的 `StackTrace` 属性的结果不同，这些结果不受模糊处理的影响。你可显式提供可选参数来控制调用方信息或隐藏调用方信息。

成员名称

可以使用 `CallerMemberName` 特性来避免将成员名称指定为所调用的方法的 `String` 参数。通过使用这种技术，可以避免“重命名重构”不更改 `String` 值的问题。此好处对于以下任务特别有用：

- 使用跟踪和诊断例程。
- 在绑定数据时实现 `INotifyPropertyChanged` 接口。此接口允许对象的属性通知绑定控件该属性已更改，以便此控件能够显示更新的信息。如果没有 `CallerMemberName` 特性，则必须将属性名称指定为文本。

以下图表显示在使用 `CallerMemberName` 特性时返回的成员名称。

方法、属性或事件	从中发起调用的方法、属性或事件的名称。
构造函数	字符串“.ctor”
静态构造函数	字符串“.cctor”
析构函数	字符串“Finalize”
用户定义的运算符或转换	为成员生成的名称，例如，“op_Addition”。
特性构造函数	要应用特性的方法或属性的名称。如果该特性是成员中的任何元素（如参数、返回值或泛型参数），则此结果是与该元素关联的成员的名称。
无包含的成员（例如，程序集级别或应用于类型的特性）	可选参数的默认值。

请参阅

- [命名参数和可选参数](#)
- [System.Reflection](#)
- [Attribute](#)
- [特性](#)

保留的特性有助于编译器的 null 状态静态分析

2021/5/7 • [Edit Online](#)

在可为 null 的上下文中，编译器对代码执行静态分析，以确定所有引用类型变量的 null 状态：

- 非 null：静态分析确定将变量分配给非 null 值。
- 可能为 null：静态分析无法确定变量是否被赋值为非 null 值。

可以应用向编译器提供有关 API 语义的信息的特性。此类信息有助于编译器执行静态分析，并确定变量何时不为 null。本文提供每个特性的简要说明以及它们的使用方法。所有示例都假设使用 C# 8.0 或更高版本，并且代码处于可为 null 的上下文中。

首先，让我们看一个熟悉的示例。假设你的库具有以下用于检索资源字符串的 API：

```
bool TryGetMessage(string key, out string message)
{
    message = "";
    return true;
}
```

前面的示例遵循 .NET 中熟悉的 `Try*` 模式。此 API 有两个引用参数：`key` 和 `message` 参数。此 API 具有与这些参数的是否为 null 相关的以下规则：

- 调用方不应将 `null` 作为 `key` 的参数传递。
- 调用方可以传递值为 `null` 的变量作为 `message` 的参数。
- 如果 `TryGetMessage` 方法返回 `true`，则 `message` 的值不为 null。如果返回值是 `false`，则 `message`（及其 null 状态）的值为 null。

`key` 的规则可以用变量类型表示：`key` 应是不可为 null 的引用类型。`message` 参数更复杂。它允许 `null` 作为参数，但保证成功时，`out` 参数不为 null。对于这些情况，需要使用更丰富的词汇来描述期望。

为表示有关变量的 null 状态的附加信息，已添加多个特性。在 C# 8 引入可为 null 的引用类型之前编写的所有代码都是忽略 null 的。这意味着任何引用类型变量都可以为 null，但不需要进行 null 检查。代码“可识别为 null”后，这些规则就会改变。引用类型不应该是 `null` 值，在取消引用之前，必须对照 `null` 检查可为 null 的引用类型。

API 的规则可能更复杂，正如你在 `TryGetValue` API 方案中看到的那样。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，可使用以下属性之一来表示这些规则：

- `AllowNull`：不可为 null 的参数可以为 null。
- `DisallowNull`：可为 null 的参数不应为 null。
- `MaybeNull`：不可为 null 的返回值可以为 null。
- `NotNull`：可为 null 的返回值永远不会为 null。
- `MaybeNullWhen`：当方法返回指定的 `bool` 值时，不可为 null 的参数可以为 null。
- `NotNullWhen`：当方法返回指定的 `bool` 值时，可以为 null 的参数不会为 null。
- `NotNullIfNotNull`：如果指定参数的参数不为 null，则返回值不为 null。
- `DoesNotReturn`：方法从不返回。换句话说，它总是引发异常。
- `DoesNotReturnIf`：如果关联的 `bool` 参数具有指定值，则此方法永远不会返回。
- `MemberNotNull`：当方法返回时，列出的成员不会为 null。
- `MemberNotNullWhen`：当方法返回指定的 `bool` 值时，列出的成员不会为 null。

上述说明是对每个特性的快速参考。以下各节介绍了行为和含义。

添加这些特性将为编译器提供有关 API 规则的更多信息。当调用代码在可为 null 的上下文中编译时，编译器将在调用方违反这些规则时发出警告。这些特性不会启用对实现进行更多检查。

指定前提条件：AllowNull 和 DisallowNull

请考虑一个从不返回 `null` 的读/写属性，因为它具有合理的默认值。调用方在将 `null` 设置为该默认值时将其传递给 `set` 访问器。例如，假设一个消息系统要求在聊天室中输入一个屏幕名称。如果未提供任何内容，系统将生成一个随机名称：

```
public string ScreenName
{
    get => _screenName;
    set => _screenName = value ?? GenerateRandomScreenName();
}
private string _screenName;
```

当你在忽略可为 null 的上下文中编译前面的代码时，一切都是正常的。启用可为 null 的引用类型后，`ScreenName` 属性将成为不可为 null 的引用。这对于 `get` 访问器是正确的：它从不返回 `null`。调用方不需要检查返回的 `null` 属性。但现在将属性设置为 `null` 将生成警告。若要支持这种类型的代码，请向属性添加 [System.Diagnostics.CodeAnalysis.AllowNullAttribute](#) 特性，如下面的代码所示：

```
[AllowNull]
public string ScreenName
{
    get => _screenName;
    set => _screenName = value ?? GenerateRandomScreenName();
}
private string _screenName = GenerateRandomScreenName();
```

可能需要为 [System.Diagnostics.CodeAnalysis](#) 添加一个 `using` 指令才能使用本文中讨论的特性和其他特性。特性应用于属性，而不是 `set` 访问器。`AllowNull` 特性指定前置条件，并且仅适用于参数。`get` 访问器有一个返回值，但没有参数。因此，`AllowNull` 特性只适用于 `set` 访问器。

前面的示例演示了在参数上添加 `AllowNull` 特性时要查找的内容：

1. 该变量的一般约定是它不应为 `null`，因此需要一个不可为 null 的引用类型。
2. 有允许此参数为 `null` 的方案，尽管这些方案不常用。

大多数情况下，属性或 `in`、`out` 和 `ref` 参数需要此特性。当变量通常为非 null 时，`AllowNull` 属性是最佳选择，但需要允许 `null` 作为前提条件。

将此与使用 `DisallowNull` 的方案相比：使用此特性可以指定可为 null 引用类型的参数不应为 `null`。请考虑一个特性，其中 `null` 是默认值，但客户端只能将其设置为非 null 值。考虑下列代码：

```
public string ReviewComment
{
    get => _comment;
    set => _comment = value ?? throw new ArgumentNullException(nameof(value), "Cannot set to null");
}
string _comment;
```

前面的代码是表达设计的最佳方式，`ReviewComment` 可以是 `null`，但不能设置为 `null`。代码可识别为 null 后，你就可以使用 [System.Diagnostics.CodeAnalysis.DisallowNullAttribute](#) 向调用方更清楚地表达此概念：

```
[DisallowNull]
public string? ReviewComment
{
    get => _comment;
    set => _comment = value ?? throw new ArgumentNullException(nameof(value), "Cannot set to null");
}
string? _comment;
```

在可为 `null` 的上下文中, `ReviewComment` `get` 访问器可以返回默认值 `null`。编译器会警告在访问之前必须进行检查。此外, 它警告调用方, 即使它可能是 `null`, 调用方也不应显式地将其设置为 `null`。`DisallowNull` 特性还指定了前置条件, 它不影响 `get` 访问器。当你观察到以下特征时, 可以使用 `DisallowNull` 特性:

1. 在核心方案中(通常是在首次实例化时), 变量可以是 `null`。
2. 变量不应显式设置为 `null`。

这些情况在忽略 `null` 的代码中很常见。可能是在两个不同的初始化操作中设置了对象属性。可能只有在某些异步工作完成后才能设置某些属性。

可使用 `AllowNull` 和 `DisallowNull` 特性指定变量上的前置条件可能与这些变量上的可为 `null` 注释不匹配。这两个特性提供了关于 API 特征的更多细节。此附加信息有助于调用方正确使用 API。请记住, 可使用以下特性指定前提条件:

- `AllowNull`: 不可为 `null` 的参数可以为 `null`。
- `DisallowNull`: 可为 `null` 的参数不应为 `null`。

指定后置条件: `MaybeNull` 和 `NotNull`

假设你有使用以下签名的方法:

```
public Customer FindCustomer(string lastName, string firstName)
```

你可能已经编写了类似的方法, 以便在未找到所查找的名称时返回 `null`。`null` 清楚地表明未找到记录。在本例中, 你可能会将返回类型从 `Customer` 更改为 `Customer?`。将返回值声明为可为 `null` 的引用类型可以清楚地指定此 API 的意图。

由于[泛型定义和为 Null 性](#)中介绍的原因, 这种技术不能用于泛型方法。你可能具有遵循类似模式的泛型方法:

```
public T Find<T>(IEnumerable<T> sequence, Func<T, bool> predicate)
```

不能指定返回值为 `T?`。当找不到所需项时, 此方法返回 `null`。由于无法声明 `T?` 返回类型, 因此需要将 `MaybeNull` 注释添加到方法返回:

```
[return: MaybeNull]
public T Find<T>(IEnumerable<T> sequence, Func<T, bool> predicate)
```

前面的代码通知调用方, 协定暗示了一个不可为 `null` 的类型, 但是返回值可能实际上为 `null`。当 API 应为不可为 `null` 的类型(通常是泛型类型参数)时, 请使用 `MaybeNull` 特性, 但可能会有返回 `null` 的情况。

还可以指定返回值或者参数不为 `null`, 即使该类型是可为 `null` 的引用类型。以下方法是一种帮助器方法, 如果其第一个参数为 `null`, 则引发该方法:

```
public static void ThrowWhenNull(object value, string valueExpression = "")  
{  
    if (value is null) throw new ArgumentNullException(valueExpression);  
}
```

可以按如下方式调用此例程：

```
public static void LogMessage(string message)  
{  
    ThrowWhenNull(message, nameof(message));  
  
    Console.WriteLine(message.Length);  
}
```

启用 null 引用类型后，需要确保前面的代码在编译时没有警告。当方法返回时，`value` 参数保证不为 null。但是，可以使用 null 引用调用 `ThrowWhenNull`。可以将 `value` 设为可为 null 的引用类型，并将 `NotNull` 后置条件添加到参数声明中：

```
public static void ThrowWhenNull([NotNull] object value, string valueExpression = "") =>  
    _ = value ?? throw new ArgumentNullException(valueExpression);
```

前面的代码清楚地表达了现有协定：调用方可以传递具有 `null` 值的变量，但如果该方法在未引发异常的情况下返回，则保证该参数永远不为 null。

可以使用以下特性指定无条件后置条件：

- `MaybeNull`: 不可为 null 的返回值可以为 null。
- `NotNull`: 可为 null 的返回值永远不会为 null。

指定有条件后置条件：`NotNullWhen`、`MaybeNullWhen` 和
`NotNullIfNotNull`

你可能很熟悉 `string` 方法 `String.IsNullOrEmpty(String)`。当参数为 null 或为空字符串时，此方法返回 `true`。这是一种 null 检查格式：如果方法返回 `false`，调用方不需要 null 检查参数。若要使这样的方法可识别为 null，需要将参数设置为可为 null 的引用类型，并添加 `NotNullWhen` 特性：

```
bool IsNullOrEmpty([NotNullWhen(false)] string? value)
```

这通知编译器，任何返回值为 `false` 的代码都不需要 null 检查。添加特性通知编译器的静态分析，`IsNullOrEmpty` 执行必要的 null 检查：当它返回 `false` 时，参数不是 `null`。

```
string? userInput = GetUserInput();  
if (!string.IsNullOrEmpty(userInput))  
{  
    int messageLength = userInput.Length; // no null check needed.  
}  
// null check needed on userInput here.
```

对于 .NET Core 3.0，将对 `String.IsNullOrEmpty(String)` 方法进行注释，如上面所示。代码库中可能有类似的方法来检查对象的状态是否为 null 值。编译器无法识别自定义的 null 检查方法，你需要自己添加注释。添加特性时，编译器的静态分析将知道何时对测试变量进行了 null 检查。

这些特性的另一个用途是 `Try*` 模式。`ref` 和 `out` 变量的后置条件通过返回值进行通信。请考虑前面所示的

方法：

```
bool TryGetMessage(string key, out string message)
{
    message = "";
    return true;
}
```

前面的方法遵循典型的.NET 习惯用法：返回值指示是否将 `message` 设置为已找到的值，或者，如果未找到消息，则为默认值。如果方法返回 `true`，`message` 的值不为 `null`；否则，该方法将 `message` 设置为 `null`。

你可以使用 `NotNullWhen` 特性来传达这个习惯用法。更新可为 `null` 的引用类型的签名时，需要将 `message` 设为 `string?` 并添加一个特性：

```
bool TryGetMessage(string key, [NotNullWhen(true)] out string? message)
{
    message = "";
    return true;
}
```

在前面的示例中，`message` 的值在 `TryGetMessage` 返回 `true` 时不为 `null`。你应该以相同的方式在代码库中注释类似的方法：参数可以是 `null`，并且已知在方法返回 `true` 时不为 `null`。

还可能需要一个最终特性。有时，返回值的 `null` 状态取决于一个或多个参数的 `null` 状态。只要某些参数不是 `null`，这些方法将返回非 `null` 值。若要正确地注释这些方法，可以使用 `NotNullIfNotNull` 特性。请考虑以下方法：

```
string GetTopLevelDomainFromFullUrl(string url)
```

如果 `url` 参数不为 `null`，则输出不是 `null`。启用可为 `null` 的引用后，只要 API 永不接受 `null` 参数，该签名就能正常运行。但是，如果参数可以为 `null`，那么返回值也可以为 `null`。可以将签名更改为以下代码：

```
string? GetTopLevelDomainFromFullUrl(string? url)
```

这也是可行的，但通常会强制调用方实现额外的 `null` 检查。协定是，只有当参数 `url` 是 `null` 时，返回值才会是 `null`。若要表达该协定，你需要注释此方法，如以下代码所示：

```
[return: NotNullIfNotNull("url")]
string? GetTopLevelDomainFromFullUrl(string? url)
```

返回值和参数都用 `?` 进行了注释，这表明两者都可以是 `null`。该特性进一步阐明了当 `url` 参数不是 `null` 时，返回值不会为 `null`。

可以使用以下特性指定条件的后置条件：

- `MaybeNullWhen`: 当方法返回指定的 `bool` 值时，不可为 `null` 的参数可以为 `null`。
- `NotNullWhen`: 当方法返回指定的 `bool` 值时，可以为 `null` 的参数不会为 `null`。
- `NotNullIfNotNull`: 如果指定参数的参数不为 `null`，则返回值不为 `null`。

构造函数帮助程序方法：`MemberNotNull` 和 `MemberNotNullWhen`。

这些特性指定了将构造函数中的公共代码重构为帮助程序方法时的意图。C# 编译器分析构造函数和字段初始值设定项，以确保在每个构造函数返回之前，所有不可为 `null` 的引用字段都已初始化。然而，C# 编译器不会通过

所有帮助程序方法跟踪字段赋值。当字段没有在构造函数中直接初始化，而在帮助程序方法中初始化时，编译器会发出警告 `CS8618`。可以将 `MemberNotNullAttribute` 添加到方法声明中，并指定在方法中初始化为非 `NULL` 值的字段。例如，考虑以下情况：

```
public class Container
{
    private string _uniqueIdentifier; // must be initialized.
    private string? _optionalMessage;

    public Container()
    {
        Helper();
    }

    public Container(string message)
    {
        Helper();
        _optionalMessage = message;
    }

    [MemberNotNull(nameof(_uniqueIdentifier))]
    private void Helper()
    {
        _uniqueIdentifier = DateTime.Now.Ticks.ToString();
    }
}
```

可以指定多个字段名称作为 `MemberNotNull` 特性构造函数的参数。

`MemberNotNullWhenAttribute` 有 `bool` 参数。在帮助程序方法返回指明帮助程序方法是否初始化了字段的 `bool` 的情况下，可以使用 `MemberNotNullWhen`。

验证无法访问的代码

某些方法（通常是异常帮助程序或其他实用工具方法）始终通过引发异常来退出。或者，帮助程序可以基于布尔参数的值引发异常。

在第一种情况下，可以将 `DoesNotReturn` 特性添加到方法声明中。编译器通过三种方式为你提供帮助。首先，如果存在方法可以退出而不抛出异常的路径，则编译器会发出警告。其次，编译器将调用此方法后的任何代码标记为“不可访问”，直到找到合适的 `catch` 子句。第三，无法访问的代码不会影响任何 `null` 状态。请考虑此方法：

```
[DoesNotReturn]
private void FailFast()
{
    throw new InvalidOperationException();
}

public void SetState(object containedField)
{
    if (!isInitialized)
    {
        FailFast();
    }

    // unreachable code:
    _field = containedField;
}
```

在第二种情况下，需将 `DoesNotReturnIf` 特性添加到方法的布尔参数中。你可以修改前面的示例，如下所示：

```
private void FailFastIf([DoesNotReturnIf(false)] bool isValid)
{
    if (!isValid)
    {
        throw new InvalidOperationException();
    }
}

public void SetFieldState(object containedField)
{
    FailFastIf(isInitialized);

    // unreachable code when "isInitialized" is false:
    _field = containedField;
}
```

总结

IMPORTANT

官方文档会跟踪最新的 C# 版本。我们目前正在为 C# 9.0 编写文档。根据所使用的 C# 版本，有些功能可能不可用。项目的默认 C# 版本基于目标框架。有关详细信息，请参阅 [C# 语言版本控制默认设置](#)。

添加可为 null 的引用类型提供了一个初始词汇表，用于描述 API 对可能为 `null` 的变量的期望。这些特性提供了更丰富的词汇来将变量的 null 状态描述为前置条件和后置条件。这些特性更清楚地描述了你的期望，并为使用 API 的开发人员提供了更好的体验。

在为可为 null 的上下文中更新库时，添加这些特性可指导用户正确使用 API。这些特性有助于你全面描述参数和返回值的 null 状态：

- [AllowNull](#): 不可为 null 的参数可以为 null。
- [DisallowNull](#): 可为 null 的参数不应为 null。
- [MaybeNull](#): 不可为 null 的返回值可以为 null。
- [NotNull](#): 可为 null 的返回值永远不会为 null。
- [MaybeNullWhen](#): 当方法返回指定的 `bool` 值时，不可为 null 的参数可以为 null。
- [NotNullWhen](#): 当方法返回指定的 `bool` 值时，可以为 null 的参数不会为 null。
- [NotNullIfNotNull](#): 如果指定参数的参数不为 null，则返回值不为 null。
- [DoesNotReturn](#): 方法从不返回。换句话说，它总是引发异常。
- [DoesNotReturnIf](#): 如果关联的 `bool` 参数具有指定值，则此方法永远不会返回。

保留属性：其他

2021/5/7 • [Edit Online](#)

这些特性可应用于代码中的元素。它们为这些元素添加语义。编译器使用这些语义来更改其输出，并报告使用你的代码的开发人员可能犯的错误。

Conditional 特性

`Conditional` 特性使得方法执行依赖于预处理标识符。`Conditional` 属性是 `ConditionalAttribute` 的别名，可以应用于方法或特性类。

在以下示例中，`Conditional` 应用于启用或禁用显示特定于程序的诊断信息的方法：

```
#define TRACE_ON
using System;
using System.Diagnostics;

namespace AttributeExamples
{
    public class Trace
    {
        [Conditional("TRACE_ON")]
        public static void Msg(string msg)
        {
            Console.WriteLine(msg);
        }
    }

    public class TraceExample
    {
        public static void Main()
        {
            Trace.Msg("Now in Main...");
            Console.WriteLine("Done.");
        }
    }
}
```

如果未定义 `TRACE_ON` 标识符，则不会显示跟踪输出。在交互式窗口中自己探索。

`Conditional` 特性通常与 `DEBUG` 标识符一起使用，以启用调试生成(而非发布生成)中的跟踪和日志记录功能，如下例所示：

```
[Conditional("DEBUG")]
static void DebugMethod()
{
}
```

当调用标记为条件的方法时，指定的预处理符号是否存在将决定是包含还是省略该调用。如果定义了符号，则将包括调用；否则，将忽略该调用。条件方法必须是类或结构声明中的方法，而且必须具有 `void` 返回类型。与将方法封闭在 `#if...#endif` 块内相比，`Conditional` 更简洁且较不容易出错。

如果某个方法具有多个 `Conditional` 特性，则如果定义了一个或多个条件符号(通过使用 OR 运算符将这些符号逻辑链接在一起)，则包含对该方法的调用。在以下示例中，存在 `A` 或 `B` 将导致方法调用：

```
[Conditional("A"), Conditional("B")]
static void DoIfAorB()
{
    // ...
}
```

使用带有特性类的 `Conditional`

`Conditional` 特性还可应用于特性类定义。在以下示例中，如果定义了 `DEBUG`，则自定义特性 `Documentation` 将仅向元数据添加信息。

```
[Conditional("DEBUG")]
public class DocumentationAttribute : System.Attribute
{
    string text;

    public DocumentationAttribute(string text)
    {
        this.text = text;
    }
}

class SampleClass
{
    // This attribute will only be included if DEBUG is defined.
    [Documentation("This method displays an integer.")]
    static void DoWork(int i)
    {
        System.Console.WriteLine(i.ToString());
    }
}
```

`Obsolete` 特性

`Obsolete` 特性将代码元素标记为不再推荐使用。使用标记为已过时的实体会生成警告或错误。`Obsolete` 特性是一次性特性，可以应用于任何允许特性的实体。`Obsolete` 是 `ObsoleteAttribute` 的别名。

在以下示例中，`Obsolete` 特性应用于类 `A` 和方法 `B.OldMethod`。因为应用于 `B.OldMethod` 的特性构造函数的第二个参数设置为 `true`，所以此方法将导致编译器错误，而使用类 `A` 只会生成警告。但是，调用 `B.NewMethod` 不会生成任何警告或错误。例如，将其与先前的定义一起使用时，以下代码会生成两个警告和一个错误：

```

using System;

namespace AttributeExamples
{
    [Obsolete("use class B")]
    public class A
    {
        public void Method() { }

    }

    public class B
    {
        [Obsolete("use NewMethod", true)]
        public void OldMethod() { }

        public void NewMethod() { }
    }

    public static class ObsoleteProgram
    {
        public static void Main()
        {
            // Generates 2 warnings:
            A a = new A();

            // Generate no errors or warnings:
            B b = new B();
            b.NewMethod();

            // Generates an error, compilation fails.
            // b.OldMethod();
        }
    }
}

```

作为特性构造函数的第一个参数提供的字符串将作为警告或错误的一部分显示。将生成类 `A` 的两个警告：一个用于声明类引用，另一个用于类构造函数。`Obsolete` 特性可以在不带参数的情况下使用，但建议说明改为使用哪个项目。

AttributeUsage 特性

`AttributeUsage` 特性确定自定义特性类的使用方式。`AttributeUsageAttribute` 是应用到自定义特性定义的特性。
`AttributeUsage` 特性帮助控制：

- 可能应用到的具体程序元素特性。除非使用限制，否则特性可能应用到以下任意程序元素：
 - 程序集 (assembly)
 - name
 - field
 - event
 - method
 - param
 - property
 - return
 - type
- 某特性是否可多次应用于单个程序元素。
- 特性是否由派生类继承。

显式应用时，默认设置如下示例所示：

```
[AttributeUsage(AttributeTargets.All,
    AllowMultiple = false,
    Inherited = true)]
class NewAttribute : Attribute { }
```

在此示例中，`NewAttribute` 类可应用于任何受支持的程序元素。但是它对每个实体仅能应用一次。特性应用于基类时，它由派生类继承。

`AllowMultiple` 和 `Inherited` 参数是可选的，因此以下代码具有相同效果：

```
[AttributeUsage(AttributeTargets.All)]
class NewAttribute : Attribute { }
```

第一个 `AttributeUsageAttribute` 参数必须是 `AttributeTargets` 枚举的一个或多个元素。可将多个目标类型与 OR 运算符链接在一起，如下例所示：

```
[AttributeUsage(AttributeTargets.Property | AttributeTargets.Field)]
class NewPropertyOrFieldAttribute : Attribute { }
```

从 C# 7.3 开始，特性可应用于自动实现的属性的属性或支持字段。特性应用于属性，除非在特性上指定 `field` 说明符。都在以下示例中进行了演示：

```
class MyClass
{
    // Attribute attached to property:
    [NewPropertyOrField]
    public string Name { get; set; }

    // Attribute attached to backing field:
    [field:NewPropertyOrField]
    public string Description { get; set; }
}
```

如果 `AllowMultiple` 参数为 `true`，那么结果特性可多次应用于单个实体，如以下示例所示：

```
[AttributeUsage(AttributeTargets.Class, AllowMultiple = true)]
class MultiUse : Attribute { }

[MultiUse]
[MultiUse]
class Class1 { }

[MultiUse, MultiUse]
class Class2 { }
```

在本例中，`MultiUseAttribute` 可重复应用，因为 `AllowMultiple` 设置为 `true`。所显示的两种用于应用多个特性的格式均有效。

如果 `Inherited` 为 `false`，那么该特性不是由特性类派生的类继承。例如：

```
[AttributeUsage(AttributeTargets.Class, Inherited = false)]
class NonInheritedAttribute : Attribute { }

[NonInherited]
class BClass { }

class DClass : BClass { }
```

在本例中，`NonInheritedAttribute` 不会通过继承应用于 `DClass`。

ModuleInitializer 特性

从 C# 9 开始，`ModuleInitializer` 属性标记程序集加载时运行时调用的方法。`ModuleInitializer` 是 `ModuleInitializerAttribute` 的别名。

`ModuleInitializer` 属性只能应用于以下方法：

- 静态方法。
- 无参数方法。
- 返回 `void`。
- 能够从包含模块(即 `internal` 或 `public`)访问的方法。
- 不是泛型的方法。
- 没有包含在泛型类中的方法。
- 不是本地函数的方法。

`ModuleInitializer` 属性可应用于多种方法。在这种情况下，运行时调用它们的顺序是确定的，但未指定。

下面的示例阐释了如何使用多个模块初始化表达式方法。`Init1` 和 `Init2` 方法在 `Main` 之前运行，并且每种方法都将一个字符串添加到 `Text` 属性。因此，当 `Main` 运行时，`Text` 属性已具有来自两个初始化表达式方法中的字符串。

```
using System;

internal class ModuleInitializerExampleMain
{
    public static void Main()
    {
        Console.WriteLine(ModuleInitializerExampleModule.Text);
        //output: Hello from Init1! Hello from Init2!
    }
}
```

```
using System.Runtime.CompilerServices;

internal class ModuleInitializerExampleModule
{
    public static string? Text { get; set; }

    [ModuleInitializer]
    public static void Init1()
    {
        Text += "Hello from Init1! ";
    }

    [ModuleInitializer]
    public static void Init2()
    {
        Text += "Hello from Init2! ";
    }
}
```

源代码生成器有时需要生成初始化代码。模块初始化表达式为该代码提供了一个标准的驻留位置。

SkipLocalsInit 特性

从 C# 9 开始，`SkipLocalsInit` 属性可防止编译器在发出到元数据时设置 `.locals init` 标志。`SkipLocalsInit` 属性是一个单用途属性，可应用于方法、属性、类、结构、接口或模块，但不能应用于程序集。`SkipLocalsInit` 是 `SkipLocalsInitAttribute` 的别名。

`.locals init` 标志会导致 CLR 将方法中声明的所有局部变量初始化为其默认值。由于编译器还可以确保在为变量赋值之前永远不使用变量，因此通常不需要使用 `.locals init`。但是，在某些情况下，额外的零初始化可能会对性能产生显著影响，例如使用 `stackalloc` 在堆栈上分配一个数组时。在这些情况下，可添加 `SkipLocalsInit` 属性。如果直接应用于方法，该属性会影响该方法及其所有嵌套函数，包括 lambda 和局部函数。如果应用于类型或模块，则它会影响嵌套在内的所有方法。此属性不会影响抽象方法，但会影响为实现生成的代码。

此属性需要 `AllowUnsafeBlocks` 编译器选项。这是为了发出信号，在某些情况下，代码可以查看未分配的内存（例如，读取未初始化的堆栈分配的内存）。

下面的示例阐释 `SkipLocalsInit` 属性对使用 `stackalloc` 的方法的影响。该方法显示分配整数数组后内存中的任何内容。

```
[SkipLocalsInit]
static void ReadUninitializedMemory()
{
    Span<int> numbers = stackalloc int[120];
    for (int i = 0; i < 120; i++)
    {
        Console.WriteLine(numbers[i]);
    }
}
// output depends on initial contents of memory, for example:
//0
//0
//0
//168
//0
//1271631451
//32767
//38
//0
//0
//0
//38
// Remaining rows omitted for brevity.
```

若要亲自尝试此代码，请在 .csproj 文件中设置 `AllowUnsafeBlocks` 编译器选项：

```
<PropertyGroup>
    ...
    <AllowUnsafeBlocks>true</AllowUnsafeBlocks>
</PropertyGroup>
```

请参阅

- [Attribute](#)
- [System.Reflection](#)
- [特性](#)
- [反射](#)

不安全代码和指针类型

2021/5/7 • [Edit Online](#)

你编写的大多数 C# 代码都是“可验证的安全代码”。可验证的安全代码表示 .NET 工具可以验证代码是否安全。通常，安全代码不会直接使用指针访问内存，也不会分配原始内存，而是创建托管对象。

C# 支持 `unsafe` 上下文，你可在其中编写不可验证的代码。在 `unsafe` 上下文中，代码可使用指针、分配和释放内存块，以及使用函数指针调用方法。C# 中的不安全代码不一定是危险的，它只是其安全性不可验证的代码。

不安全代码具有以下属性：

- 可将方法、类型和代码块定义为不安全。
- 在某些情况下，通过移除数组绑定检查，不安全代码可提高应用程序的性能。
- 调用需要指针的本机函数时，需使用不安全代码。
- 使用不安全代码将引发安全风险和稳定性风险。
- 必须使用 `AllowUnsafeBlocks` 编译器选项来编译包含不安全块的代码。

指针类型

在不安全的上下文中，类型除了是值类型或引用类型外，还可以是指针类型。指针类型声明采用下列形式之一：

```
type* identifier;  
void* identifier; //allowed but not recommended
```

在指针类型中的 `*` 之前指定的类型被称为“referent 类型”。只有 [非托管类型](#) 可为引用类型。

指针类型不从 [对象](#) 继承，并且指针类型与 `object` 之间不存在转换。此外，装箱和取消装箱不支持指针。但是，你可在不同的指针类型之间以及指针类型和整型之间进行转换。

当在同一声明中声明多个指针时，星号 (`*`) 仅与基础类型一起写入，而不是用作每个指针名称的前缀。例如：

```
int* p1, p2, p3; // Ok  
int *p1, *p2, *p3; // Invalid in C#
```

指针不能指向引用或包含引用的[结构](#)，因为无法对对象引用进行垃圾回收，即使有指针指向它也是如此。垃圾回收器并不跟踪是否有任何类型的指针指向对象。

`MyType*` 类型的指针变量的值为 `MyType` 类型的变量的地址。下面是指针类型声明的示例：

- `int* p : p` 是指向整数的指针。
- `int** p : p` 是指向整数的指针的指针。
- `int*[] p : p` 是指向整数的指针的一维数组。
- `char* p : p` 是指向字符的指针。
- `void* p : p` 是指向未知类型的指针。

指针间接寻址运算符 `*` 可用于访问位于指针变量所指向的位置的内容。例如，请考虑以下声明：

```
int* myVariable;
```

表达式 `*myVariable` 表示在 `int` 中包含的地址处找到的 `myVariable` 变量。

关于 `fixed` 语句 的文章中有几个指针示例。下面的示例使用 `unsafe` 关键字和 `fixed` 语句，并显示如何递增内部指针。你可将此代码粘贴到控制台应用程序的 Main 函数中来运行它。这些示例必须使用 AllowUnsafeBlocks 编译器选项集进行编译。

```
// Normal pointer to an object.  
int[] a = new int[5] { 10, 20, 30, 40, 50 };  
// Must be in unsafe code to use interior pointers.  
unsafe  
{  
    // Must pin object on heap so that it doesn't move while using interior pointers.  
    fixed (int* p = &a[0])  
    {  
        // p is pinned as well as object, so create another pointer to show incrementing it.  
        int* p2 = p;  
        Console.WriteLine(*p2);  
        // Incrementing p2 bumps the pointer by four bytes due to its type ...  
        p2 += 1;  
        Console.WriteLine(*p2);  
        p2 += 1;  
        Console.WriteLine(*p2);  
        Console.WriteLine("-----");  
        Console.WriteLine(*p);  
        // Dereferencing p and incrementing changes the value of a[0] ...  
        *p += 1;  
        Console.WriteLine(*p);  
        *p += 1;  
        Console.WriteLine(*p);  
    }  
}  
  
Console.WriteLine("-----");  
Console.WriteLine(a[0]);  
  
/*  
Output:  
10  
20  
30  
-----  
10  
11  
12  
-----  
12  
*/
```

你无法对 `void*` 类型的指针应用间接寻址运算符。但是，你可以使用强制转换将 `void` 指针转换为任何其他指针类型，反之亦然。

指针可以为 `null`。将间接寻址运算符应用于 `null` 指针将导致由实现定义的行为。

在方法之间传递指针会导致未定义的行为。考虑这种方法，该方法通过 `in`、`out` 或 `ref` 参数或作为函数结果返回一个指向局部变量的指针。如果已在固定块中设置指针，则它指向的变量不再是固定的。

下表列出了可在不安全的上下文中对指针执行的运算符和语句：

运算符	操作
<code>*</code>	执行指针间接寻址。
<code>-></code>	通过指针访问结构的成员。

操作符	功能
<code>[]</code>	为指针建立索引。
<code>&</code>	获取变量的地址。
<code>++</code> 和 <code>--</code>	递增和递减指针。
<code>+</code> 和 <code>-</code>	执行指针算法。
<code>==</code> 、 <code>!=</code> 、 <code><</code> 、 <code>></code> 、 <code><=</code> 和 <code>>=</code>	比较指针。
<code>stackalloc</code>	在堆栈上分配内存。
<code>fixed</code> 语句	临时固定变量以便找到其地址。

要详细了解与指针相关的运算符，请参阅[与指针相关的运算符](#)。

任何指针类型都可以隐式转换为 `void*` 类型。可以为任何指针类型分配值 `null`。可以使用强制转换表达式将任何指针类型显式转换为任何其他指针类型。也可以将任何整数类型转换为指针类型，或将任何指针类型转换为整数类型。这些转换需要显式转换。

以下示例将 `int*` 转换为 `byte*`。请注意，指针指向变量的最低寻址字节。如果结果连续递增，直达到 `int` 的大小(4 字节)，可显示变量的其余字节。

```
int number = 1024;

unsafe
{
    // Convert to byte:
    byte* p = (byte*)&number;

    System.Console.Write("The 4 bytes of the integer:");

    // Display the 4 bytes of the int variable:
    for (int i = 0 ; i < sizeof(int) ; ++i)
    {
        System.Console.Write(" {0:X2}", *p);
        // Increment the pointer:
        p++;
    }
    System.Console.WriteLine();
    System.Console.WriteLine("The value of the integer: {0}", number);

    /* Output:
       The 4 bytes of the integer: 00 04 00 00
       The value of the integer: 1024
    */
}
```

固定大小的缓冲区

在 C# 中，可以使用 `fixed` 语句来创建在数据结构中具有固定大小的数组的缓冲区。当编写与其他语言或平台的数据源进行互操作的方法时，固定大小的缓冲区很有用。固定的数组可以采用允许用于常规结构成员的任何属性或修饰符。唯一的限制是数组类型必须为 `bool`、`byte`、`char`、`short`、`int`、`long`、`sbyte`、`ushort`、`uint`、`ulong`、`float` 或 `double`。

```
private fixed char name[30];
```

在安全代码中，包含数组的 C# 结构不包含该数组的元素，而该结构包含对这些元素的引用。当在[不安全的代码](#)块中使用数组时，可以在[结构](#)中嵌入固定大小的数组。

以下 `struct` 的大小不依赖于数组中的元素数，因为 `pathName` 是一个引用：

```
public struct PathArray
{
    public char[] pathName;
    private int reserved;
}
```

`struct` 可以在不安全代码中包含嵌入的数组。在下面的示例中，`fixedBuffer` 数组具有固定的大小。使用 `fixed` 语句建立指向第一个元素的指针。通过此指针访问数组的元素。`fixed` 语句将 `fixedBuffer` 实例字段固定到内存中的特定位置。

```
internal unsafe struct Buffer
{
    public fixed char fixedBuffer[128];
}

internal unsafe class Example
{
    public Buffer buffer = default;
}

private static void AccessEmbeddedArray()
{
    var example = new Example();

    unsafe
    {
        // Pin the buffer to a fixed location in memory.
        fixed (char* charPtr = example.buffer.fixedBuffer)
        {
            *charPtr = 'A';
        }
        // Access safely through the index:
        char c = example.buffer.fixedBuffer[0];
        Console.WriteLine(c);

        // Modify through the index:
        example.buffer.fixedBuffer[0] = 'B';
        Console.WriteLine(example.buffer.fixedBuffer[0]);
    }
}
```

包含 128 个元素的 `char` 数组的大小为 256 个字节。在固定大小的 `char` 缓冲区中，每个字符总是占用 2 个字节，不考虑编码。甚至在使用 `CharSet = CharSet.Auto` 或 `CharSet = CharSet.Ansi` 将 `char` 缓冲区封送到 API 方法或结构时，此数组大小也是相同的。有关更多信息，请参见[CharSet](#)。

前面的示例演示访问未固定的 `fixed` 字段，此功能从 C# 7.3 开始提供。

另一常见的固定大小的数组是 `bool` 数组。`bool` 数组中的元素大小始终为 1 个字节。`bool` 数组不适用于创建位数组或缓冲区。

固定大小的缓冲区使用 [System.Runtime.CompilerServices.UnsafeValueTypeAttribute](#) 进行编译，它指示公共语言运行时 (CLR) 某个类型包含可能溢出的非托管数组。使用 `stackalloc` 分配的内存还会在 CLR 中自动启用缓冲区溢出检测功能。前面的示例演示如何在 `unsafe struct` 中存在固定大小的缓冲区。

```
internal unsafe struct Buffer
{
    public fixed char fixedBuffer[128];
}
```

为 `Buffer` 生成 C# 的编译器的特性如下：

```
internal struct Buffer
{
    [StructLayout(LayoutKind.Sequential, Size = 256)]
    [CompilerGenerated]
    [UnsafeValueType]
    public struct <fixedBuffer>e__FixedBuffer
    {
        public char FixedElementField;
    }

    [FixedBuffer(typeof(char), 128)]
    public <fixedBuffer>e__FixedBuffer fixedBuffer;
}
```

固定大小的缓冲区与常规数组的区别体现在以下方面：

- 只能在 `unsafe` 上下文中使用。
- 只能是结构的实例字段。
- 它们始终是矢量或一维数组。
- 声明应包括长度，如 `fixed char id[8]`。不能使用 `fixed char id[]`。

如何使用指针来复制字节数组

下面的示例使用指针将字节从一个数组复制到另一个数组。

此示例使用 `unsafe` 关键字，使你可以在 `Copy` 方法中使用指针。`fixed` 语句用于声明指向源数组和目标数组的指针。`fixed` 语句将源数组和目标数组的位置固定在内存中，以便它们不会被垃圾回收所移动。当完成 `fixed` 块后，将取消固定数组的内存块。因为此示例中的 `Copy` 方法使用 `unsafe` 关键字，所以必须使用 `AllowUnsafeBlocks` 编译器选项对其进行编译。

此示例使用索引而非第二个非托管的指针访问这两个数组的元素。`pSource` 和 `pTarget` 指针的声明固定数组。从 C# 7.3 开始可以使用此功能。

```
static unsafe void Copy(byte[] source, int sourceOffset, byte[] target,
    int targetOffset, int count)
{
    // If either array is not instantiated, you cannot complete the copy.
    if ((source == null) || (target == null))
    {
        throw new System.ArgumentException();
    }

    // If either offset, or the number of bytes to copy, is negative, you
    // cannot complete the copy.
    if ((sourceOffset < 0) || (targetOffset < 0) || (count < 0))
    {
        throw new System.ArgumentException();
    }

    // If the number of bytes from the offset to the end of the array is
    // less than the number of bytes you want to copy, you cannot complete
    // the copy.
    if ((source.Length - sourceOffset < count) ||
        (target.Length - targetOffset < count))
```

```

        (target.Length - targetOffset < count))
    {
        throw new System.ArgumentException();
    }

    // The following fixed statement pins the location of the source and
    // target objects in memory so that they will not be moved by garbage
    // collection.
    fixed (byte* pSource = source, pTarget = target)
    {
        // Copy the specified number of bytes from source to target.
        for (int i = 0; i < count; i++)
        {
            pTarget[targetOffset + i] = pSource[sourceOffset + i];
        }
    }
}

static void UnsafeCopyArrays()
{
    // Create two arrays of the same length.
    int length = 100;
    byte[] byteArray1 = new byte[length];
    byte[] byteArray2 = new byte[length];

    // Fill byteArray1 with 0 - 99.
    for (int i = 0; i < length; ++i)
    {
        byteArray1[i] = (byte)i;
    }

    // Display the first 10 elements in byteArray1.
    System.Console.WriteLine("The first 10 elements of the original are:");
    for (int i = 0; i < 10; ++i)
    {
        System.Console.Write(byteArray1[i] + " ");
    }
    System.Console.WriteLine("\n");

    // Copy the contents of byteArray1 to byteArray2.
    Copy(byteArray1, 0, byteArray2, 0, length);

    // Display the first 10 elements in the copy, byteArray2.
    System.Console.WriteLine("The first 10 elements of the copy are:");
    for (int i = 0; i < 10; ++i)
    {
        System.Console.Write(byteArray2[i] + " ");
    }
    System.Console.WriteLine("\n");

    // Copy the contents of the last 10 elements of byteArray1 to the
    // beginning of byteArray2.
    // The offset specifies where the copying begins in the source array.
    int offset = length - 10;
    Copy(byteArray1, offset, byteArray2, 0, length - offset);

    // Display the first 10 elements in the copy, byteArray2.
    System.Console.WriteLine("The first 10 elements of the copy are:");
    for (int i = 0; i < 10; ++i)
    {
        System.Console.Write(byteArray2[i] + " ");
    }
    System.Console.WriteLine("\n");
    /* Output:
       The first 10 elements of the original are:
       0 1 2 3 4 5 6 7 8 9

       The first 10 elements of the copy are:
       0 1 2 3 4 5 6 7 8 9
    */
}

```

```
The first 10 elements of the copy are:  
90 91 92 93 94 95 96 97 98 99  
*/  
}
```

函数指针

C# 提供 `delegate` 类型来定义安全函数指针对象。调用委托时，需要实例化从 `System.Delegate` 派生的类型并对其 `Invoke` 方法进行虚拟方法调用。该虚拟调用使用 IL 指令 `callvirt`。在性能关键的代码路径中，使用 IL 指令 `calli` 效率更高。

可以使用 `delegate*` 语法定义函数指针。编译器将使用 `calli` 指令来调用函数，而不是实例化 `delegate` 对象并调用 `Invoke`。以下代码声明了两种方法，它们使用 `delegate` 或 `delegate*` 来组合两个类型相同的对象。第一种方法使用 `System.Func<T1,T2,TResult>` 委托类型。第二种方法使用具有相同参数和返回类型的 `delegate*` 声明：

```
public static T Combine<T>(Func<T, T, T> combinator, T left, T right) =>  
    combinator(left, right);  
  
public static T UnsafeCombine<T>(delegate*<T, T, T> combinator, T left, T right) =>  
    combinator(left, right);
```

以下代码显示如何声明静态本地函数并使用指向该本地函数的指针调用 `UnsafeCombine` 方法：

```
static int localMultiply(int x, int y) => x * y;  
int product = UnsafeCombine(&localMultiply, 3, 4);
```

前面的代码说明了有关作为函数指针使用的函数的几个规则：

- 函数指针只能在 `unsafe` 上下文中声明。
- 只能在 `unsafe` 上下文中调用采用 `delegate*` (或返回 `delegate*`) 的方法。
- 只可在 `static` 函数上使用 `&` 运算符获取函数的地址。(此规则适用于成员函数和本地函数)。

此语法与声明 `delegate` 类型和使用指针具有相似之处。`delegate` 上的后缀 `*` 表示声明是函数指针。将方法组分配给函数指针时，`&` 表示操作采用方法的地址。

可以使用关键字 `managed` 和 `unmanaged` 为 `delegate*` 指定调用约定。另外，对于 `unmanaged` 函数指针，可以指定调用约定。下面的声明显示每个示例。第一个声明使用 `managed` 调用约定，这是默认值。后面三个使用 `unmanaged` 调用约定。每个声明都指定以下某个 ECMA 335 调用约定：`Cdecl`、`Stdcall`、`Fastcall` 或 `Thiscall`。最后的声明使用 `unmanaged` 调用约定，指示 CLR 选择平台的默认调用约定。CLR 将在运行时选择调用约定。

```
public static T ManagedCombine<T>(delegate* managed<T, T, T> combinator, T left, T right) =>  
    combinator(left, right);  
public static T CDeclCombine<T>(delegate* unmanaged[Cdecl]<T, T, T> combinator, T left, T right) =>  
    combinator(left, right);  
public static T StdcallCombine<T>(delegate* unmanaged[Stdcall]<T, T, T> combinator, T left, T right) =>  
    combinator(left, right);  
public static T FastcallCombine<T>(delegate* unmanaged[Fastcall]<T, T, T> combinator, T left, T right) =>  
    combinator(left, right);  
public static T ThiscallCombine<T>(delegate* unmanaged[Thiscall]<T, T, T> combinator, T left, T right) =>  
    combinator(left, right);  
public static T UnmanagedCombine<T>(delegate* unmanaged<T, T, T> combinator, T left, T right) =>  
    combinator(left, right);
```

可以在 C# 9.0 的[函数指针](#)建议中详细了解函数指针。

C# 语言规范

有关详细信息，请参阅 [C# 语言规范](#)中的[不安全代码](#)部分。

C# 预处理器指令

2021/5/8 • [Edit Online](#)

尽管编译器没有单独的预处理器，但本节中所述指令的处理方式与有预处理器时一样。可使用这些指令来帮助条件编译。不同于 C 和 C++ 指令，不能使用这些指令来创建宏。预处理器指令必须是一行中唯一的说明。

可为空上下文

`#nullable` 预处理器指令将设置可为空注释上下文和可为空警告上下文。此指令控制是否可为空注释是否有效，以及是否给出为 Null 性警告。每个上下文要么处于已禁用状态，要么处于已启用状态。

可在项目级别(C# 源代码之外)指定这两个上下文。`#nullable` 指令控制注释和警告上下文，并优先于项目级设置。指令会设置其控制的上下文，直到另一个指令替代它，或直到源文件结束为止。

指令的效果如下所示：

- `#nullable disable`：将可为空注释和警告上下文设置为“已禁用”。
- `#nullable enable`：将可为空注释和警告上下文设置为“已启用”。
- `#nullable restore`：将可为空注释和警告上下文还原为项目设置。
- `#nullable disable annotations`：将可为空注释上下文设置为“已禁用”。
- `#nullable enable annotations`：将可为空注释上下文设置为“已启用”。
- `#nullable restore annotations`：将可为空注释上下文还原为项目设置。
- `#nullable disable warnings`：将可为空警告上下文设置为“已禁用”。
- `#nullable enable warnings`：将可为空警告上下文设置为“已启用”。
- `#nullable restore warnings`：将可为空警告上下文还原为项目设置。

条件编译

使用四个预处理器指令来控制条件编译：

- `#if`：打开条件编译，其中仅在定义了指定的符号时才会编译代码。
- `#elif`：关闭前面的条件编译，并基于是否定义了指定的符号打开一个新的条件编译。
- `#else`：关闭前面的条件编译，如果没有定义前面指定的符号，打开一个新的条件编译。
- `#endif`：关闭前面的条件编译。

如果 C# 编译器遇到 `#if` 指令，最后跟着一个 `#endif` 指令，则仅当定义指定的符号时，它才编译这些指令之间的代码。与 C 和 C++ 不同，不能将数字值分配给符号。C# 中的 `#if` 语句是布尔值，且仅测试是否已定义该符号。例如：

```
#if DEBUG
    Console.WriteLine("Debug version");
#endif
```

可以使用运算符 `== (相等)` 和 `!= (不相等)` 来测试 `bool` 值是 `true` 还是 `false`。`true` 表示定义该符号。语句 `#if DEBUG` 具有与 `#if (DEBUG == true)` 相同的含义。可以使用 `&& (and)`、`|| (or)` 和 `! (not)` 运算符来计算是否已定义多个符号。还可以用括号对符号和运算符进行分组。

`#if` 以及 `#else`、`#elif`、`#endif`、`#define` 和 `#undef` 指令，允许基于是否存在一个或多个符号包括或排除代码。条件编译在编译调试版本的代码或编译特定配置的代码时会很有用。

以 `#if` 指令开头的条件指令必须以 `#endif` 指令显式终止。`#define` 允许你定义一个符号。通过将该符号用作传递给 `#if` 指令的表达式，该表达式的计算结果为 `true`。还可以通过 `DefineConstants` 编译器选项来定义符号。可以通过 `#undef` 取消定义符号。使用 `#define` 创建的符号的作用域是在其中定义它的文件。使用 `DefineConstants` 或 `#define` 定义的符号与具有相同名称的变量不冲突。也就是说，变量名称不应传递给预处理器指令，且符号仅能由预处理器指令评估。

`#elif` 可以创建复合条件指令。如果之前的 `#if` 和任何之前的可选 `#elif` 指令表达式的值都不为 `true`，则计算 `#elif` 表达式。如果 `#elif` 表达式计算结果为 `true`，编译器将计算 `#elif` 和下一条件指令间的所有代码。例如：

```
#define VC7
//...
#if debug
    Console.WriteLine("Debug build");
#elif VC7
    Console.WriteLine("Visual Studio 7");
#endif
```

`#else` 允许创建复合条件指令，因此，如果先前 `#if` 或(可选) `#elif` 指令中的任何表达式的计算结果都不是 `true`，则编译器将对介于 `#else` 和下一个 `#endif` 之间的所有代码进行求值。`#endif`(`#endif`) 必须是 `#else` 之后的下一个预处理器指令。

`#endif` 指定条件指令的末尾，以 `#if` 指令开头。

此外，生成系统还会感知表示 SDK 样式项目中不同目标框架的预定义预处理器符号。在创建可以面向多个 .NET 版本的应用程序时，这些符号会很有用。

SDK	非 SDK
.NET Framework	NETFRAMEWORK, NET48, NET472, NET471, NET47, NET462, NET461, NET46, NET452, NET451, NET45, NET40, NET35, NET20
.NET Standard	NETSTANDARD, NETSTANDARD2_1, NETSTANDARD2_0, NETSTANDARD1_6, NETSTANDARD1_5, NETSTANDARD1_4, NETSTANDARD1_3, NETSTANDARD1_2, NETSTANDARD1_1, NETSTANDARD1_0
.NET 5(和 .NET Core)	NET, NET5_0, NETCOREAPP, NETCOREAPP3_1, NETCOREAPP3_0, NETCOREAPP2_2, NETCOREAPP2_1, NETCOREAPP2_0, NETCOREAPP1_1, NETCOREAPP1_0

NOTE

对于传统的非 SDK 样式的项目，必须通过项目的属性页面在 Visual Studio 中为不同目标框架手动配置条件编译符号。

其他预定义符号包括 `DEBUG` 和 `TRACE` 常数。你可以使用 `#define` 替代项目的值集。例如，会根据生成配置属性(“调试”或者“发布”模式)自动设置 `DEBUG` 符号。

下例显示如何在文件上定义 `MYTEST` 符号，然后测试 `MYTEST` 和 `DEBUG` 符号的值。此示例的输出取决于是在“调试”还是“发布”配置模式下生成项目。

```

#define MYTEST
using System;
public class MyClass
{
    static void Main()
    {
#if (DEBUG && !MYTEST)
        Console.WriteLine("DEBUG is defined");
#elif (!DEBUG && MYTEST)
        Console.WriteLine("MYTEST is defined");
#elif (DEBUG && MYTEST)
        Console.WriteLine("DEBUG and MYTEST are defined");
#else
        Console.WriteLine("DEBUG and MYTEST are not defined");
#endif
    }
}

```

下例显示如何针对不同的目标框架进行测试，以便在可能时使用较新的 API：

```

public class MyClass
{
    static void Main()
    {
#if NET40
        WebClient _client = new WebClient();
#else
        HttpClient _client = new HttpClient();
#endif
    }
    //...
}

```

定义符号

使用以下两个预处理器指令来定义或取消定义条件编译的符号：

- `#define` : 定义符号。
- `#undef` : 取消定义符号。

使用 `#define` 来定义符号。将符号用作传递给 `#if` 指令的表达式时，该表达式的计算结果为 `true`，如以下示例所示：

```

#define VERBOSE

#if VERBOSE
    Console.WriteLine("Verbose output version");
#endif

```

NOTE

`#define` 指令不能用于声明常量值，这与 C 和 C++ 中的通常做法一样。C# 中的常量最好定义为类或结构的静态成员。如果具有多个此类常量，请考虑创建一个单独的“常量”类来容纳它们。

符号可用于指定编译的条件。可通过 `#if` 或 `#elif` 测试符号。还可以使用 [ConditionalAttribute](#) 来执行条件编译。可以定义符号，但不能为符号分配值。文件中必须先出现 `#define` 指令，才能使用并非同时也是预处理器

指令的任何指示。还可以通过 [DefineConstants](#) 编译器选项来定义符号。可以通过 `#undef` 取消定义符号。

定义区域

可以使用以下两个预处理器指令来定义可在大纲中折叠的代码区域：

- `#region` : 启动区域。
- `#endregion` : 结束区域。

利用 `#region`，可以指定在使用代码编辑器的[大纲](#)功能时可展开或折叠的代码块。在较长的代码文件中，折叠或隐藏一个或多个区域十分便利，这样，可将精力集中于当前处理的文件部分。下面的示例演示如何定义区域：

```
#region MyClass definition
public class MyClass
{
    static void Main()
    {
    }
}
#endregion
```

`#region` 块必须通过 `#endregion` 指令终止。`#region` 块不能与 `#if` 块重叠。但是，可以将 `#region` 块嵌套在 `#if` 块内，或将 `#if` 块嵌套在 `#region` 块内。

错误和警告信息

使用以下指令指示编译器生成用户定义的编译器错误和警告，并控制行信息：

- `#error` : 使用指定的消息生成编译器错误。
- `#warning` : 使用指定的消息生成编译器警告。
- `#line` : 更改用编译器消息输出的行号。

`#error` 可从代码中的特定位置生成 [CS1029](#) 用户定义的错误。例如：

```
#error Deprecated code in this method.
```

NOTE

编译器以特殊的方式处理 `#error version` 并报告编译器错误 CS8304，消息中包含使用的编译器和语言版本。

`#warning` 允许你从代码中的特定位置生成 [CS1030](#) 第一级编译器警告。例如：

```
#warning Deprecated code in this method.
```

借助 `#line`，可修改编译器的行号及(可选)用于错误和警告的文件名输出。

以下示例演示如何报告与行号相关联的两个警告。`#line 200` 指令将下一行的行号强制设为 200(尽管默认值为 #6)；在执行下一个 `#line` 指令前，文件名都会报告为“特殊”。`#line default` 指令将行号恢复至默认行号，这会对上一指令重新编号的行进行计数。

```

class MainClass
{
    static void Main()
    {
#line 200 "Special"
        int i;
        int j;
#line default
        char c;
        float f;
#line hidden // numbering not affected
        string s;
        double d;
    }
}

```

编译产生以下输出：

```

Special(200,13): warning CS0168: The variable 'i' is declared but never used
Special(201,13): warning CS0168: The variable 'j' is declared but never used
MainClass.cs(9,14): warning CS0168: The variable 'c' is declared but never used
MainClass.cs(10,15): warning CS0168: The variable 'f' is declared but never used
MainClass.cs(12,16): warning CS0168: The variable 's' is declared but never used
MainClass.cs(13,16): warning CS0168: The variable 'd' is declared but never used

```

可在生成过程的自动、中间步骤中使用 `#line` 指令。例如，如果已从原始源代码文件中删除行，但仍希望编译器基于文件中的原始行号生成输出，可在删除行后，使用 `#line` 来模拟原始行号。

`#line hidden` 指令能对调试程序隐藏连续行，当开发者逐行执行代码时，介于 `#line hidden` 和下一 `#line` 指令（假设它不是其他 `#line hidden` 指令）间的任何行都将被跳过。还可通过此选项允许 ASP.NET 区分用户定义和计算机生成的代码。虽然此功能主要用于 ASP.NET，但可能更多的源生成器会利用此功能。

`#line hidden` 指令不影响错误报告中的文件名或行号。也就是说，如果编译器在隐藏块中发现错误，编译器将报告错误的当前文件名和行号。

`#line filename` 指令可指定要在编译器输出中显示的文件名。默认情况下，将使用源代码文件的实际名称。该文件名必须以双引号 ("") 引起来，且必须位于行号之后。

下列示例演示调试程序如何忽略代码中的隐藏行。运行示例时，它将显示三行文本。但是，如果按照示例所示设置断点，并按 F10 逐行执行代码，调试程序将忽略隐藏行。即使在隐藏行设置断点，调试程序仍将忽略它。

```

// preprocessor_linehidden.cs
using System;
class MainClass
{
    static void Main()
    {
        Console.WriteLine("Normal line #1."); // Set break point here.
#line hidden
        Console.WriteLine("Hidden line.");
#line default
        Console.WriteLine("Normal line #2.");
    }
}

```

杂注

`#pragma` 为编译器给出特殊指令以编译它所在的文件。这些指令必须受编译器支持。换句话说，不能使用 `#pragma` 创建自定义的预处理指令。

- `#pragma warning` : 启用或禁用警告。
- `#pragma checksum` : 生成校验和。

```
#pragma pragma-name pragma-arguments
```

其中 `pragma-name` 是可识别 `pragma` 的名称, `pragma-arguments` 是特定于 `pragma` 的参数。

#pragma warning

`#pragma warning` 可以启用或禁用特定警告。

```
#pragma warning disable warning-list
#pragma warning restore warning-list
```

其中 `warning-list` 是以逗号分隔的警告编号的列表。“CS”前缀是可选的。未指定警告编号时, `disable` 会禁用所有警告, `restore` 会启用所有警告。

NOTE

若要在 Visual Studio 中查找警告编号, 请生成项目, 然后在“输出”窗口中查找警告编号。

`disable` 从源文件的下一行开始生效。警告会在后面的 `restore` 行上还原。如果文件中没有 `restore`, 则在同一编译中任何之后文件的第一行, 警告将还原为其默认状态。

```
// pragma_warning.cs
using System;

#pragma warning disable 414, CS3021
[CLSCompliant(false)]
public class C
{
    int i = 1;
    static void Main()
    {
    }
}
#pragma warning restore CS3021
[CLSCompliant(false)] // CS3021
public class D
{
    int i = 1;
    public static void F()
    {
    }
}
```

#pragma checksum

生成源文件的校验和以帮助调试 ASP.NET 页面。

```
#pragma checksum "filename" "{guid}" "checksum bytes"
```

其中, `"filename"` 是需要监视更改或更新的文件的名称, `"{guid}"` 是哈希算法的全局唯一标识符 (GUID), `"checksum_bytes"` 是表示校验和字节的十六进制数字的字符串。必须是偶数个十六进制数字。奇数个数字会导致编译时警告出现, 且指令遭忽略。

Visual Studio 调试器使用校验和确保它可始终找到正确的源。编译器为源文件计算校验和, 然后将输出发出到

程序数据库 (PDB) 文件。调试器随后使用 PDB 针对它为源文件计算的校验和进行比较。

此解决方案不适用于 ASP.NET 项目，因为计算出的校验和是针对生成的源文件，而不是 .aspx 文件。为解决此问题，`#pragma checksum` 为 ASP.NET 页面提供校验和支持。

在 Visual C# 中创建 ASP.NET 项目时，生成的源文件包含 .aspx 文件（从该文件生成源）的校验和。编译器随后将此信息写入 PDB 文件中。

如果编译器在文件中没有找到 `#pragma checksum` 指令，它将计算校验和，并将该值写入 PDB 文件。

```
class TestClass
{
    static int Main()
    {
        #pragma checksum "file.cs" "{406EA660-64CF-4C82-B6F0-42D48172A799}" "ab007f1d23d9" // New checksum
    }
}
```

C# 编译器选项

2021/5/7 • • [Edit Online](#)

本节介绍 C# 编译器解释的选项。可以通过两种不同的方法设置 .NET 项目中的编译器选项：

- 在 *.csproj 文件中指定选项：可以为 *.csproj 文件中的任何编译器选项添加 XML 元素。元素名称与编译器选项相同。用 XML 元素的值设置编译器选项的值。有关项目文件中设置选项的详细信息，请参阅[适用于 .NET SDK 项目的 MSBuild 属性](#)一文。
- 使用 Visual Studio 属性页：Visual Studio 提供了属性页，可用于编辑生成属性。若要了解有关详细信息，请参阅[管理项目和解决方案属性 - Windows](#) 或[管理项目和解决方案属性 - Mac](#)。

.NET Framework 项目

IMPORTANT

本部分仅适用于 .NET Framework 项目。

除上述机制以外，还可以使用两种附加方法为 .NET Framework 项目设置编译器选项：

- .NET Framework 项目的命令行参数：.NET Framework 项目使用 csc.exe 而不是 `dotnet build` 生成项目。可以为 .NET Framework 项目指定 csc.exe 的命令行参数。
- 已编译的 ASP.NET 页面：.NET Framework 项目使用 web.config 文件的一部分来编译页面。对于新的生成系统和 ASP.NET Core 项目，将从项目文件中设置选项。

某些编译器选项的单词从 csc.exe 和 .NET Framework 项目更改为新的 MSBuild 系统。本部分使用的是新语法。每个页面顶部同时列出了这两个版本。对于 csc.exe，所有参数都会在选项和冒号后列出。例如，`-doc` 选项将为：

```
-doc:DocFile.xml
```

通过在命令提示符处键入 C# 编译器的可执行文件名称 (csc.exe)，可调用该编译器。

对于 .NET Framework 项目，还可以从命令行运行 csc.exe。每个编译器选项均有两种形式：`-option` 和 `/option`。在 .NET Framework Web 项目中，在 web.config 文件中指定用于编译代码隐藏的选项。有关详细信息，请参阅[<compiler> 元素](#)。

如果使用“Visual Studio 开发人员命令提示”窗口，系统将设置所有必需的环境变量。有关如何访问此工具的信息，请参阅[Visual Studio 开发人员命令提示](#)。

csc.exe 可执行文件通常位于 Windows 目录下的 Microsoft.NET\Framework\ <Version> 文件夹中。根据每台计算机上的具体配置，此位置可能有所不同。如果计算机上安装了不止一个版本的 .NET Framework，你将发现此文件的多个版本。有关此类安装的详细信息，请参阅[如何：确定安装的 .NET Framework 版本](#)。

用于语言功能规则的 C# 编译器选项

2021/5/7 • [Edit Online](#)

以下选项控制编译器如何解释语言功能。新的 MSBuild 语法以粗体显示。旧的 csc.exe 语法以 `code style` 显示。

- `CheckForOverflowUnderflow / -checked` : 生成溢出检查。
- `AllowUnsafeBlocks / -unsafe` : 允许“不安全”代码。
- `DefineConstants / -define` : 定义条件编译符号。
- `LangVersion / -langversion` : 指定语言版本, 如 `default` (最新主版本) 或 `latest` (最新版本, 包括次要版本)。
- `Nullable / -nullable` : 启用可为空上下文或可为空警告。

CheckForOverflowUnderflow

`CheckForOverflowUnderflow` 选项指定产生的值超出数据类型范围的整数算法语句是否将导致运行时异常。

```
<CheckForOverflowUnderflow>true</CheckForOverflowUnderflow>
```

`checked` 或 `unchecked` 关键字范围内的整数算法语句不受 `CheckForOverflowUnderflow` 选项的影响。如果不在 `checked` 或 `unchecked` 关键字范围内的整数算法语句产生的值超出数据类型范围, 并且 `CheckForOverflowUnderflow` 为 `true`, 则该语句将在运行时导致异常。如果 `CheckForOverflowUnderflow` 为 `false`, 则该语句在运行时不会导致异常。此选项的默认值为“`-checked-`” `false`; 溢出检查已禁用。

AllowUnsafeBlocks

`AllowUnsafeBlocks` 编译器选项允许使用 `unsafe` 关键字的代码进行编译。此选项的默认值为 `false`, 表示不允许不安全代码。

```
<AllowUnsafeBlocks>true</AllowUnsafeBlocks>
```

有关不安全代码的详细信息, 请参阅[不安全代码和指针](#)。

DefineConstants

`DefineConstants` 选项将定义程序中所有源代码文件的符号。

```
<DefineConstants>name;name2</DefineConstants>
```

此选项指定要定义的一个或多个符号的名称。`DefineConstants` 选项具有与 `#define` 预处理器指令相同的效果, 只不过编译器选项对项目中的所有文件都有效。符号在源文件中保持已定义状态, 直到源文件中的 `#undef` 指令删除该定义。当你使用 `-define` 选项时, 一个文件中的 `#undef` 指令不影响项目中的其他源代码文件。可以将由此选项创建的符号同 `#if`、`#else`、`#elif` 和 `#endif` 一起使用, 对源文件进行条件编译。C# 编译器本身不定义源代码中使用的符号或宏;所有符号定义必须都是用户定义的。

NOTE

同 C++ 等语言一样, C# `#define` 指令不允许为符号赋值。例如, `#define` 不能用于创建宏或定义常数。如果需要定义一个常数, 请使用 `enum` 变量。若要创建 C++ 风格的宏, 请考虑泛型等替代项。由于宏非常容易出错, 所以 C# 不允许使用宏, 但提供了更安全的替代项。

LangVersion

使编译器仅接受包含在所选 C# 语言规范中的语法。

```
<LangVersion>9.0</LangVersion>
```

以下为有效值:

"1"	1
<code>preview</code>	编译器接受最新预览版中的所有有效语言语法。
<code>latest</code>	编译器接受最新发布的编译器版本(包括次要版本)中的语法。
<code>latestMajor</code> (<code>default</code>)	编译器接受最新发布的编译器主要版本中的语法。
<code>9.0</code>	编译器只接受 C# 9.0 或更低版本中所含的语法。
<code>8.0</code>	编译器只接受 C# 8.0 或更低版本中所含的语法。
<code>7.3</code>	编译器只接受 C# 7.3 或更低版本中所含的语法。
<code>7.2</code>	编译器只接受 C# 7.2 或更低版本中所含的语法。
<code>7.1</code>	编译器只接受 C# 7.1 或更低版本中所含的语法。
<code>7</code>	编译器只接受 C# 7.0 或更低版本中所含的语法。
<code>6</code>	编译器只接受 C# 6.0 或更低版本中所含的语法。
<code>5</code>	编译器只接受 C# 5.0 或更低版本中所含的语法。
<code>4</code>	编译器只接受 C# 4.0 或更低版本中所含的语法。
<code>3</code>	编译器只接受 C# 3.0 或更低版本中所含的语法。
<code>ISO-2</code> (或 <code>2</code>)	编译器只接受 ISO/IEC 23270:2006 C# (2.0) 中所含的语法。
<code>ISO-1</code> (或 <code>1</code>)	编译器只接受 ISO/IEC 23270:2003 C# (1.0/1.2) 中所含的语法。

默认语言版本依赖于应用程序的目标框架以及所安装的 SDK 或 Visual Studio 的版本。这些规则是在 [C# 语言版本控制](#) 中定义的。

C# 应用程序引用的元数据不受 LangVersion 编译器选项约束。

每个版本的 C# 编译器都包含语言规范的扩展，因此 LangVersion 不提供早期版本编译器的同等功能。

此外，虽然 C# 版本更新通常与主要的 .NET Framework 版本一致，但新的语法和功能不一定绑定到该特定的 Framework 版本。虽然新功能肯定需要与 C# 修订版一起发布的新编译器更新，但每项具体功能都有自己的最小 .NET API 或公共语言运行时要求，这些要求通过包括 NuGet 包或其他库允许功能在下层框架上运行。

无论使用哪一项 LangVersion 设置，请使用当前版本的公共语言运行时来创建 .exe 或 .dll。友元程序集和 [ModuleAssemblyName](#) 是一个例外，它们在 -langversion:ISO-1 下工作。

若要了解指定 C# 语言版本的其他方式，请参阅 [C# 语言版本控制](#)。

有关如何以编程方式设置此编译器选项的信息，请参阅 [LanguageVersion](#)。

C# 语言规范

VERSION	LINK	LINK
C# 7.0 和更高版本		当前不可用
C# 6.0	链接	C# 语言规范版本 6 - 非官方草稿:.NET Foundation
C# 5.0	下载 PDF	标准 ECMA-334 第 5 版
C# 3.0	下载 DOC	C# 语言规范版本 3.0 :Microsoft Corporation
C# 2.0	下载 PDF	标准 ECMA-334 第 4 版
C# 1.2	下载 DOC	C# 语言规范版本 1.2 :Microsoft Corporation
C# 1.0	下载 DOC	C# 语言规范版本 1.0 :Microsoft Corporation

支持所有语言功能所需的最低 SDK 版本

下表列出了支持相应语言版本的 C# 编译器的 SDK 的最低版本：

C#	SDK
C# 8.0	Microsoft Visual Studio/生成工具 2019, 版本 16.3 或 .NET Core 3.0 SDK
C# 7.3	Microsoft Visual Studio/生成工具 2017, 版本 15.7
C# 7.2	Microsoft Visual Studio/生成工具 2017, 版本 15.5
C# 7.1	Microsoft Visual Studio/生成工具 2017, 版本 15.3
C# 7.0	Microsoft Visual Studio/生成工具 2017
C# 6	Microsoft Visual Studio/生成工具 2015

C# 版本	SDK 版本
C# 5	Microsoft Visual Studio/生成工具 2012 或捆绑的 .NET Framework 4.5 编译器
C# 4	Microsoft Visual Studio/生成工具 2010 或捆绑的 .NET Framework 4.0 编译器
C# 3	Microsoft Visual Studio/生成工具 2008 或捆绑的 .NET Framework 3.5 编译器
C# 2	Microsoft Visual Studio/生成工具 2005 或捆绑的 .NET Framework 2.0 编译器
C# 1.0/1.2	Microsoft Visual Studio/生成工具 .NET 2002 或捆绑的 .NET Framework 1.0 编译器

Nullable

使用 `Nullable` 选项可指定可为空上下文。

```
<Nullable>enable</Nullable>
```

参数必须为以下项之一：`enable`、`disable`、`warnings` 或 `annotations`。`enable` 参数启用可为空上下文。指定 `disable` 将禁用可为空上下文。如果提供 `warnings` 参数，将启用可为空警告上下文。如果指定 `annotations` 参数，将启用可为空注释上下文。

流分析用于在可执行代码中推断变量的为空性。推断出的变量的为空性与变量声明的为空性无关。即使有条件地省略方法调用，也会对其进行分析。例如，发布模式下的 `Debug.Assert`。

采用以下特性进行注释的方法调用也将影响流分析：

- 简单的前提条件：`AllowNullAttribute` 和 `DisallowNullAttribute`
- 简单的后置条件：`MaybeNullAttribute` 和 `NotNullAttribute`
- 有条件的后置条件：`MaybeNullWhenAttribute` 和 `NotNullWhenAttribute`
- `DoesNotReturnIfAttribute`(例如 `Debug.Assert` 的 `DoesNotReturnIf(false)`) 和 `DoesNotReturnAttribute`
- `NotNullIfNotNullAttribute`
- 成员后置条件：`MemberNotNullAttribute(String)` 和 `MemberNotNullAttribute(String[])`

IMPORTANT

全局可为空上下文不适用于生成的代码文件。无论此设置如何，都会针对标记为“已生成”的任何源文件禁用可为空上下文。可采用四种方法将文件标记为“已生成”：

- 在 `.editorconfig` 中，在应用于该文件的部分中指定 `generated_code = true`。
- 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。
- 文件名以 `TemporaryGeneratedFile_` 开头
- 文件名用以 `.designer.cs`、`.generated.cs`、`.g.cs` 或 `.g.i.cs` 结尾。

生成器可以选择使用 `#nullable` 预处理器指令。

用于控制编译器输出的 C# 编译器选项

2021/5/7 • [Edit Online](#)

以下选项控制编译器输出生成。新的 MSBuild 语法以粗体显示。旧的 csc.exe 语法以 `code style` 显示。

- **DocumentationFile** / `-doc` : 从 `///` 注释生成 XML 文档文件。
- **OutputAssembly** / `-out` : 指定输出程序集文件。
- **PlatformTarget** / `-platform` : 指定目标平台 CPU。
- **ProduceReferenceAssembly** / `-refout` : 生成引用程序集。
- **TargetType** `-target` : 指定输出程序集的类型。

DocumentationFile

使用 DocumentationFile 选项可以在 XML 文件中放置文档注释。若要详细了解如何记录代码，请参阅[建议的文档注释标记](#)。该值指定输出 XML 文件的路径。此 XML 文件包含编译的源代码文件中的注释。

```
<DocumentationFile>path/to/file.xml</DocumentationFile>
```

包含 Main 或顶级语句的源代码文件首先输出到 XML 中。通常需要将生成的 .xml 文件与 [IntelliSense](#) 一起使用。
.xml 文件名必须与程序集名称相同。xml 文件必须与程序集位于同一目录中。在 Visual Studio 项目中引用程序集时，也会找到 .xml 文件。若要详细了解如何生成代码注释，请参阅[提供代码注释](#)。除非用
`<TargetType:Module>` 进行编译，否则 `file` 将包含 `<assembly>` 和 `</assembly>` 标记，用于指定包含输出文件的
程序集清单的文件名。例如，请参阅[如何使用 XML 文档功能](#)。

NOTE

DocumentationFile 选项适用于项目中的所有文件。若要禁用与特定文件或一段代码的文档注释相关的警告，请使用
`#pragma 警告`。

OutputAssembly

OutputAssembly 选项指定输出文件的名称。输出路径指定放置编译器输出的文件夹。

```
<OutputAssembly>folder</OutputAssembly>
```

指定想要创建的文件的完整名称和扩展名。如果不指定输出文件的名称，MSBuild 将使用项目的名称指定输出
程序集的名称。旧样式的项目使用以下规则：

- .exe 将采用包含 `Main` 方法或顶级语句的源代码文件中的名称。
- .dll 或者 .netmodule 将从第一个源代码文件中获取其名称。

在编译时生成的任何模块都将成为与编译时生成的程序集关联的文件。使用 [ildasm.exe](#) 查看程序集清单，了解
关联文件。

为使 exe 成为[友元程序集](#)的目标，OutputAssembly 编译器选项是必需的。

PlatformTarget

指定 CLR 的哪个版本可以运行程序集。

```
<PlatformTarget>anycpu</PlatformTarget>
```

- anycpu(默认值)将程序集编译成可在任意平台上运行。您的应用程序将尽可能作为 64 位进程运行;当只有 32 位模式可用时,才会回退到 32 位。
- anycpu32bitpreferred 将程序集编译成可在任意平台上运行。在同时支持 64 位和 32 位应用程序的系统上,您的应用程序将以32 位模式运行。只能为面向 .NET Framework 4.5 或更高版本的项目指定此选项。
- ARM 将程序集编译成可以在具有高级 RISC 计算机 (ARM) 处理器的计算机上运行。
- ARM64 编译程序集以在由 64 位 CLR 在具有支持 A64 指令集的高级 RISC 计算机 (ARM) 处理器的计算机上运行。
- x64 将程序集编译成可由支持 AMD64 或 EM64T 指令集的计算机上的 64 位 CLR 运行。
- x86 将程序集编译成可由 32 位、x86 可兼容 CLR 运行。
- Itanium 将程序集编译成可由配有 Itanium 处理器的计算机上的 64 位 CLR 运行。

在 64 位 Windows 操作系统上:

- 用 x86 编译的程序集将在 WOW64 下运行的 32 位 CLR 上执行。
- 用 anycpu 编译的 DLL 将在加载它的进程所在的同一 CLR 上执行。
- 用 anycpu 编译的可执行文件将在 64 位 CLR 上执行。
- 用 anycpu32bitpreferred 编译的可执行文件将在 32 位 CLR 上执行。

anycpu32bitpreferred 设置只对可执行文件 (.EXE) 有效, 并且需要 .NET Framework 4.5 或更高版本。有关开发 Windows 64 位操作系统上运行的应用程序的详细信息, 请参阅 [64 位应用程序](#)。

在 Visual Studio 中, 可以从项目的“生成”属性页中设置 PlatformTarget 选项。

对于 .NET Core 和 .NET 5 及更高版本, anycpu 的行为有一些额外的细微差别。设置 anycpu 后, 请发布应用并将其与 x86 `dotnet.exe` 或 x64 `dotnet.exe` 一起执行。对于自包含应用, `dotnet publish` 步骤会将配置 RID 的可执行文件打包。

ProduceReferenceAssembly

ProduceReferenceAssembly 选项指定应输出引用程序集的文件路径。它在 Emit API 中转换为 `metadataPeStream`。
。 `filepath` 指定引用程序集的路径。通常情况下, 应与主程序集的路径匹配。建议的约定(由 MSBuild 采用)是, 将引用程序集放入与主程序集相关的“ref/”子文件夹中。

```
<ProduceReferenceAssembly>filepath</ProduceReferenceAssembly>
```

引用程序集是一种特殊类型的程序集, 它只包含表示库的公共 API 外围应用所需的最少元数据量。它们包括在生成工具中引用程序集时所需的所有成员的声明。引用程序集不包括所有成员实现以及私有成员的声明。这些成员对其 API 协定没有明显影响。有关详细信息, 请参阅 .NET 指南中的[引用程序集](#)。

ProduceReferenceAssembly 和 [ProduceOnlyReferenceAssembly](#) 选项是互斥的。

TargetType

TargetType 编译器选项可指定为以下形式之一:

- library:用于创建代码库。library 是默认值。
- exe:用于创建 .exe 文件。
- module:用于创建模块。
- winexe:用于创建 Windows 程序。
- winmdobj:用于创建 .winmdobj 中间文件。

- **appcontainerexe**: 用于为 Windows 8.x 应用商店应用创建 .exe 文件。

NOTE

对于 .NET Framework 目标, 除非指定 module, 否则, 此选项会导致将 .NET Framework 程序集清单放入输出文件中。有关详细信息, 请参阅 [.NET 中的程序集和公共属性](#)。

```
<TargetType>library</TargetType>
```

编译器每次编译只创建一个程序集清单。关于编译中所有文件的信息都放在程序集清单中。在命令行生成多个输出文件时, 只能创建一个程序集清单, 且该清单必须放置在命令行上指定的第一个输出文件中。

如果你创建了一个程序集, 则可以用 [CLSCompliantAttribute](#) 属性指示全部或部分代码是符合 CLS 的。

库

library 选项会导致编译器创建动态链接库 (DLL) 而不是可执行文件 (EXE)。将创建具有 .dll 扩展名的 DLL。除非使用 [OutputAssembly](#) 选项指定, 否则输出文件的名称采用第一个输入文件的名称。生成 .dll 文件时, 不需要 [Main](#) 方法。

exe

exe 选项会导致编译器创建一个可执行的 (EXE) 控制台应用程序。将创建扩展名为 .exe 的可执行文件。使用 [winexe](#) 创建 Windows 程序可执行文件。除非使用 [OutputAssembly](#) 选项进行指定, 否则输出文件的名称将采用包含入口点 ([Main](#) 方法或顶级语句) 的输入文件的名称。编译为 .exe 文件的源代码文件中只需要一个入口点。如果代码具有多个附带 [Main](#) 方法的类, 则可使用 [StartupObject](#) 编译器选项指定包含 [Main](#) 方法的类。

name

此选项会导致编译器不生成程序集清单。默认情况下, 使用此选项编译时所创建的输出文件具有扩展名 .netmodule。.NET 运行时无法加载没有程序集清单的文件。但是, 此类文件可以通过 [AddModules](#) 合并到程序集的程序集清单中。如果在一次编译中创建了多个模块, 某个模块中的 [内部](#) 类型将适用于编译中的其他模块。如果一个模块中的代码引用另一模块中的 [internal](#) 类型, 则两个模块必须通过 [AddModules](#) 合并到一个程序集清单中。Visual Studio 开发环境中不支持创建模块。

winexe

winexe 选项会导致编译器创建可执行的 (EXE) Windows 程序。将创建扩展名为 .exe 的可执行文件。Windows 程序是通过 .NET 库或 Windows API 提供用户界面的程序。使用 [exe](#) 创建控制台应用程序。除非使用 [OutputAssembly](#) 选项进行指定, 否则输出文件的名称将采用包含 [Main](#) 方法的输入文件的名称。编译为 .exe 文件的源代码文件中只需要一个 [Main](#) 方法。如果代码具有多个附带 [Main](#) 方法的类, 则可使用 [StartupObject](#) 选项指定包含 [Main](#) 方法的类。

winmdobj

如果使用 [winmdobj](#) 选项, 则编译器会创建一个 .winmdobj 中间文件, 你可以将此文件转换为 Windows 运行时二进制 (.winmd) 文件。之后, 除了托管语言程序外, JavaScript 和 C++ 程序也可以使用该 .winmd 文件。

winmdobj 设置会向编译器发出信号, 表示需要中间模块。随后, .winmdobj 文件可作为 [WinMDExp](#) 导出工具的输入, 生成 Windows 元数据 (.winmd) 文件。.winmd 文件既包含原始库的代码, 也包含 JavaScript 或 C++ 以及 Windows 运行时使用的 WinMD 元数据的代码。使用 [winmdobj](#) 编译器选项所编译文件的输出只能作 [WinMDExp](#) 导出工具的输入。.winmdobj 文件本身并没有被直接引用。除非使用 [OutputAssembly](#) 选项, 否则输出文件的名称将采用第一个输入文件的名称。不需要使用 [Main](#) 方法。

appcontainerexe

如果使用 [appcontainerexe](#) 编译器选项, 则编译器会创建一个 Windows 可执行 (.exe) 文件, 该文件必须在应用容器中运行。此选项与 [-target:winexe](#) 等效, 但专门用于 Windows 8.x 应用商店应用。

为了要求应用在应用容器中运行，此选项在[可移植可执行 \(PE\)](#)文件中设置了一个位。设置该位时，如果 CreateProcess 方法尝试在应用容器外启动该可执行文件，就会发生错误。除非使用 [OutputAssembly](#) 选项，否则输出文件的名称将采用包含 [Main](#) 方法的输入文件的名称。

可指定输入的 C# 编译器选项

2021/5/7 • [Edit Online](#)

以下选项控制编译器输入。新的 MSBuild 语法以粗体显示。旧的 csc.exe 语法以 `code style` 显示。

- **References** / `-reference` 或 `-references` : 引用来自一个或多个指定程序集文件的元数据。
- **AddModules** / `-addmodule` : 添加模块(将使用 `target:module` 创建的模块添加到此程序集。)
- **EmbedInteropTypes** / `-link` : 嵌入指定互操作程序集文件中的元数据。

参考

References 选项使编译器将指定文件中的 `public` 类型信息导入当前项目，从而使你可从指定的程序集文件中引用元数据。

```
<Reference Include="filename" />
```

`filename` 是包含程序集清单的文件的名称。若要导入多个文件，请为每个文件加上一个单独的 Reference 元素。可以定义一个别名作为 Reference 元素的子元素：

```
<Reference Include="filename.dll">
  <Aliases>LS</Aliases>
</Reference>
```

在上面的示例中，`LS` 是表示根命名空间的有效 C# 标识符，该根命名空间将包含程序集 filename.dll 中的所有命名空间。导入的文件必须包含一个清单。使用 [AdditionalLibPaths](#) 指定一个或多个程序集引用所在的目录。[AdditionalLibPaths](#) 主题还讨论了编译器在其中搜索程序集的目录。为使编译器可以识别程序集(而不是模块)中的某个类型，需要强制解析此类型，这可以通过定义此类型的实例来完成。还可通过其他方法为编译器解析程序集中的类型名称：例如，如果从程序集中继承类型，编译器就能识别类型名称。有时，需要在一个程序集内引用同一组件的两个不同版本。为此，请在每个文件的 References 元素上使用 Aliases 元素，以区分这两个文件。此别名将用作组件名的限定符，并解析为其中一个文件中的组件。

NOTE

在 Visual Studio 中，请使用“添加引用”命令。有关详细信息，请参阅 [How to: Add or Remove References By Using the Reference Manager](#)。

AddModules

此选项将添加一个模块，该模块通过将 `<TargetType>module</TargetType>` 切换到当前编译进行创建：

```
<AddModule Include=file1 />
<AddModule Include=file2 />
```

其中 `file1`、`file2` 是包含元数据的输出文件。该文件不能包含程序集清单。若要导入多个文件，请用逗号或分号将文件名隔开。通过 AddModules 添加的所有模块在运行时必须位于与输出文件相同的目录中。也就是说，在编译时可在任何目录中指定模块，但在运行时该模块必须位于应用程序目录中。如果在运行时该模块不位于应用程序目录中，你将收到 [TypeLoadException](#)。`file` 不能包含程序集。例如，如果输出文件是使用 module 的 `TargetType` 创建的，那么它的元数据可通过 AddModules 导入。

如果输出文件是使用 [TargetType](#) 选项而不是 module 创建的，则它的元数据不能通过 AddModules 导入，但可以通过 [References](#) 选项导入。

EmbedInteropTypes

使编译器让指定程序集中的 COM 类型信息可供当前正在编译的项目使用。

```
<References>
  <EmbedInteropTypes>file1;file2;file3</EmbedInteropTypes>
</References>
```

其中 `file1;file2;file3` 是以分号分隔的程序集文件名的列表。如果文件名包含空格，则将名称括在引号内。使用 [EmbedInteropTypes](#) 选项可以部署具有嵌入类型信息的应用程序。应用程序随后可以使用运行时程序集中实现嵌入类型信息的类型，而无需引用运行时程序集。如果发布了各种版本的运行时程序集，则包含嵌入类型信息的应用程序可以使用各种版本，而无需重新编译。有关示例，请参阅[演练：嵌入托管程序集中的类型](#)。

当你使用 COM 互操作时，[EmbedInteropTypes](#) 选项尤其有用。可以嵌入 COM 类型，以便应用程序在目标计算机上不再需要主互操作程序集 (PIA)。[EmbedInteropTypes](#) 选项指示编译器将引用的互操作程序集中的 COM 类型信息嵌入到生成的已编译代码中。COM 类型由 CLSID (GUID) 值进行标识。因此，应用程序可以在安装了具有相同 CLSID 值的相同 COM 类型的目标计算机上运行。自动执行 Microsoft Office 的应用程序是一个很好的示例。由于 Office 等应用程序通常在不同版本间保持相同的 CLSID 值，因此只要在目标计算机上安装了 .NET Framework 4 或更高版本，并且应用程序使用引用的 COM 类型中包含的方法、属性或事件，应用程序便可以使用引用的 COM 类型。[EmbedInteropTypes](#) 选项只嵌入接口、结构和委托。不支持嵌入 COM 类。

NOTE

在代码中创建嵌入 COM 类型的实例时，必须使用适当的接口创建该实例。尝试使用组件类创建嵌入 COM 类型的实例会导致错误。

与 [References](#) 编译器选项一样，[EmbedInteropTypes](#) 编译器选项使用 Csc.rsp 响应文件，该文件引用常用的 .NET 程序集。如果你不希望编译器使用 Csc.rsp 文件，可以使用 [NoConfig](#) 编译器选项。

```
// The following code causes an error if ISampleInterface is an embedded interop type.
ISampleInterface<SampleType> sample;
```

对于具有类型是从互操作程序集嵌入的泛型参数的类型，如果该类型来自外部程序集，则无法使用这种类型。此限制不适用于接口。例如，考虑在 [Microsoft.Office.Interop.Excel](#) 程序集中定义的 [Range](#) 接口。如果某个库从 [Microsoft.Office.Interop.Excel](#) 程序集嵌入互操作类型，并且公开的一个方法返回具有类型是 [Range](#) 接口的参数的泛型类型，则该方法必须返回泛型接口，如下面的代码示例所示。

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using Microsoft.Office.Interop.Excel;

public class Utility
{
    // The following code causes an error when called by a client assembly.
    public List<Range> GetRange1()
    {
        return null;
    }

    // The following code is valid for calls from a client assembly.
    public IList<Range> GetRange2()
    {
        return null;
    }
}
```

在下面的示例中，客户端代码可以调用返回 [IList](#) 泛型接口的方法而不会出现错误。

```
public class Client
{
    public void Main()
    {
        Utility util = new Utility();

        // The following code causes an error.
        List<Range> rangeList1 = util.GetRange1();

        // The following code is valid.
        List<Range> rangeList2 = (List<Range>)util.GetRange2();
    }
}
```

用于报告错误和警告的 C# 编译器选项

2021/5/7 • [Edit Online](#)

以下选项控制编译器如何报告错误和警告。新的 MSBuild 语法以粗体显示。旧的 csc.exe 语法以 `code style` 显示。

- **WarningLevel** / `-warn` : 设置警告等级。
- **TreatWarningsAsErrors** / `-warnaserror` : 将所有警告视为错误
- **WarningsAsErrors** / `-warnaserror` : 将一个或多个警告视为错误
- **WarningsNotAsErrors** / `-warnaserror` : 将一个或多个警告不视为错误
- **DisabledWarnings** / `-nowarn` : 设置禁用的警告的列表。
- **CodeAnalysisRuleSet** / `-ruleset` : 指定可禁用特定诊断的规则集文件。
- **ErrorLog** / `-errorlog` : 指定要记录所有编译器和分析器诊断的文件。
- **ReportAnalyzer** / `-reportanalyzer` : 报告其他分析器信息, 如执行时间。

WarningLevel

WarningLevel 选项指定编译器显示的警告等级。

```
<WarningLevel>3</WarningLevel>
```

此元素值是要为编译显示的警告等级:较低的数字仅显示高严重性警告。较高的数字显示更多警告。该值必须是零或正整数:

0	关闭发出所有警告消息。
1	显示严重警告消息。
2	显示等级 1 警告以及某些不太严重的警告, 如有关隐藏类成员的警告。
3	显示等级 2 警告以及某些不太严重的警告, 如有关经常计算为 <code>true</code> 或 <code>false</code> 的表达式的警告。
4(默认值)	显示所有等级 3 警告以及信息性警告。
5	显示等级 4 警告以及 C# 9.0 附带的编译器中的 其他警告 。
大于 5	任何大于 5 的值都将被视为 5。为了确保在编译器更新了新的警告等级时, 你始终能收到所有警告, 可以输入一个任意的大值(例如 <code>9999</code>)。

若要获取有关错误或警告的信息, 可以在帮助索引中查找错误代码。有关获取错误或警告信息的其他方法, 请参阅 [C# 编译器错误](#)。使用 [TreatWarningsAsErrors](#) 将所有警告视为错误。使用 [DisabledWarnings](#) 禁用某些警告。

TreatWarningsAsErrors

TreatWarningsAsErrors 选项将所有警告视为错误。你还可以使用 TreatWarningsAsErrors 仅将部分警告设置为错误。如果启用 TreatWarningsAsErrors，则可以使用 TreatWarningsAsErrors 列出不应被视为错误的警告。

```
<TreatWarningsAsErrors></TreatWarningsAsErrors>
```

而是将所有警告消息报告为错误。生成过程会停止(不生成输出文件)。默认情况下，TreatWarningsAsErrors 不生效，这意味着警告不会阻止生成输出文件。(可选)如果希望仅将一些特定警告视为错误，则可以指定视为错误的警告编号的逗号分隔列表。可以使用 [Nullable](#) 的简写形式指定所有为 Null 性警告的集合。使用 [WarningLevel](#) 指定你希望编译器显示的警告等级。使用 [DisabledWarnings](#) 禁用某些警告。

WarningsAsErrors 和 WarningsNotAsErrors

WarningsAsErrors 和 WarningsNotAsErrors 选项会覆盖警告列表的 TreatWarningsAsErrors 选项。

允许将警告 0219 和 0168 视为错误：

```
<WarningsAsErrors>0219,0168</WarningsAsErrors>
```

禁止将相同的警告视为错误：

```
<WarningsNotAsErrors>0219,0168</WarningsNotAsErrors>
```

可以使用 WarningsAsErrors 将一组警告配置为错误。将所有警告都设置为错误时，使用 WarningsNotAsErrors 配置一组不应为错误的警告。

DisabledWarnings

使用 DisabledWarnings 选项可以禁止编译器显示一个或多个警告。使用逗号分隔多个警告编号。

```
<DisabledWarnings>number1, number2</DisabledWarnings>
```

number1, number2 是你希望编译器禁止显示的警告编号。指定警告标识符的数值部分。例如，如果要禁止显示 CS0028，可以指定 `<DisabledWarnings>28</DisabledWarnings>`。编译器会以无提示方式忽略传递给 DisabledWarnings 的警告编号，这些编号在之前的版本中有效，但已被移除。例如，CS0679 在 Visual Studio .NET 2002 的编译器中有效，但后来已被移除。

通过 DisabledWarnings 选项不能禁止显示以下警告：

- 编译器警告(等级 1)CS2002
- 编译器警告(等级 1)CS2023
- 编译器警告(等级 1)CS2029

CodeAnalysisRuleSet

指定可配置特定诊断的规则集文件。

```
<CodeAnalysisRuleSet>MyConfiguration.ruleset</CodeAnalysisRuleSet>
```

其中 `MyConfiguration.ruleset` 是规则集文件的路径。有关使用规则集的详细信息，请参阅[有关规则集的 Visual Studio 文档](#)中的文章。

ErrorLog

指定要记录所有编译器和分析器诊断的文件。

```
<ErrorLog>compiler-diagnostics.sarif</ErrorLog>
```

ErrorLog 选项会导致编译器输出[静态分析结果交换格式 \(SARIF\) 日志](#)。SARIF 日志通常由分析编译器和分析器诊断结果的工具来读取。

ReportAnalyzer

报告其他分析器信息，如执行时间。

```
<ReportAnalyzer>true</ReportAnalyzer>
```

ReportAnalyzer 选项会导致编译器发出额外的 MSBuild 日志信息，这些信息详细说明生成中分析器的性能特征。它通常由分析器作者在验证分析器时使用。

控制代码生成的 C# 编译器选项

2021/5/7 • • [Edit Online](#)

下面的选项控制编译器生成的代码。新的 MSBuild 语法以粗体显示。旧的 csc.exe 语法以 `code style` 显示。

- **DebugType** / `-debug` : 发出(或不发出)调试信息。
- **Optimize** / `-optimize` : 启用优化。
- **Deterministic** / `-deterministic` : 从相同的输入源生成每字节对等的输出。
- **ProduceOnlyReferenceAssembly** / `-refonly` : 生成引用程序集作为主输出, 而非生成完整程序集。

DebugType

DebugType 选项将使编译器生成调试信息, 并将此信息放置在一个或多个输出文件中。默认情况下, 将为调试生成配置添加调试信息。默认情况下, 为发布生成配置禁用调试信息。

```
<DebugType>pdbonly</DebugType>
```

自 C# 6.0 起, 对于所有编译器版本而言, pdbonly 与 full 之间没有任何区别。请选择 pdbonly。若要更改 .pdb 文件的位置, 请参阅 [PdbFile](#)。

以下为有效值:

VALUE	IF
<code>full</code>	使用当前平台的默认格式向 .pdb 文件发出调试信息: Windows: Windows pdb 文件。 Linux/macOS: 可移植 PDB 文件。
<code>pdbonly</code>	与 <code>full</code> 相同。有关详细信息, 请参阅下面的注释。
<code>portable</code>	使用跨平台可移植 PDB 格式向 .pdb 文件发出调试信息。
<code>embedded</code>	使用 可移植 PDB 格式向 .dll/.exe 自身(未生成 .pdb 文件)发出调试信息。

IMPORTANT

以下信息仅适用于 C# 6.0 以前的编译器。此元素的值可以是 `full` 或 `pdbonly`。full 参数(在不指定 pdbonly 时生效)允许将调试器附加到正在运行的程序。指定 pdbonly 后, 可以在调试器中启动程序时进行源代码调试, 但仅在正在运行的程序附加到调试器时才显示汇编程序。使用此选项创建调试版本。如果使用 Full, 请注意, 对经过优化的 JIT 代码的速度和大小会存在一定影响, 使用 full 时对代码质量的影响较小。建议使用 pdbonly 或不使用 PDB 生成发布代码。pdbonly 和 full 之间的一个区别在于, 使用 full, 编译器将发出 `DebuggableAttribute`, 用于告知 JIT 编译器有可用调试信息。因此, 在使用 full 时, 如果代码包含设置为 false 的 `DebuggableAttribute`, 将出现错误。有关如何配置应用程序的调试性能的详细信息, 请参阅 [令映像更易于调试](#)。

优化

Optimize 选项启用或禁用编译器执行的优化, 使输出文件更小、更快、更有效。默认情况下, 为发布生成配置启用 Optimize 选项。默认情况下, 为调试生成配置禁用此选项。

```
<Optimize>true</Optimize>
```

在 Visual Studio 中，可以从项目的“生成”属性页中设置 Optimize 选项。

Optimize 还指示公共语言运行时在运行时优化代码。默认情况下，禁用优化。指定 Optimize+ 可启用优化。生成程序集使用的模块时，请使用与程序集所使用的相同 Optimize 设置。可以将 Optimize 和 [Debug](#) 选项组合使用。

具有确定性

如果输入相同，则会导致编译器生成的程序集其逐字节输出在整个编译期间中相同。

```
<Deterministic>true</Deterministic>
```

默认情况下，一组给定输入的编译器输出是唯一的，因为编译器会添加时间戳和随意数字生成的 MVID。使用 `<Deterministic>` 选项生成确定性的程序集，只要输入保持不变，该程序集的二进制内容在整个编译中都是相同的。在此类生成中，时间戳和 MVID 字段会被替换为从所有编译输入的哈希派生的值。编译器会考虑影响确定性的以下输入：

- 命令行参数序列。
- 编译器 .rsp 响应文件的内容。
- 所用编译器的精确版本及其引用的程序集。
- 当前目录路径。
- 直接或间接地显式传递到编译器的所有文件的二进制内容，包括：
 - 源文件
 - 引用的程序集
 - 引用的模块
 - 资源
 - 强名称密钥文件
 - @ 响应文件
 - 分析器
 - 规则集
 - 分析器可能使用的其他文件
- 当前区域性(针对生成诊断和异常消息的语言)。
- 在未指定编码情况下使用的默认编码(或当前代码页)。
- 编译器搜索路径(例如，由 `-lib` 或 `-recurse` 指定)上文件是否存在及其内容。
- 运行编译器的公共语言运行时 (CLR) 平台。
- `%LIBPATH%` 的值，该值会影响分析器的依赖项加载。

可使用确定性编译来确定是否从可信源编译二进制内容。当源公开可用时，确定性输出会很有用。它还可以确定生成步骤是否依赖于生成过程中使用的二进制文件的更改。

ProduceOnlyReferenceAssembly

`ProduceOnlyReferenceAssembly` 选项表示应输出引用程序集(而不是实现程序集)作为主输出。

`ProduceOnlyReferenceAssembly` 参数以无提示方式禁用输出 PDB，因为无法执行引用程序集。

```
<ProduceOnlyReferenceAssembly>true</ProduceOnlyReferenceAssembly>
```

引用程序集是一种特殊类型的程序集。引用程序集只包含表示库的公共 API 外围应用所需的最少元数据量。它

们包括在生成工具中引用程序集时所需的所有成员的声明，但不包括所有成员实现以及对其 API 协定没有明显影响的私有成员的声明。有关详细信息，请参阅[引用程序集](#)。

ProduceOnlyReferenceAssembly 和 [ProduceReferenceAssembly](#) 选项是互斥的。

用于安全选项的 C# 编译器选项

2021/5/7 • [Edit Online](#)

以下选项可控制编译器安全选项。新的 MSBuild 语法以粗体显示。旧的 csc.exe 语法以 `code style` 显示。

- **PublicSign** / `-publicsign` : 公开对程序集签名。
- **DelaySign** / `-delaysign` : 仅使用强名称密钥的公共部分对程序集进行延迟签名。
- **KeyFile** / `-keyfile` : 指定强名称密钥文件。
- **KeyContainer** / `-keycontainer` : 指定强名称密钥容器。
- **HighEntropyVA** / `-highentropyva` : 启用高熵地址空间布局随机化 (ASLR)

PublicSign

此选项会导致编译器应用公钥，但不会实际对程序集签名。PublicSign 选项还会在程序集中设置位，以告知运行时该文件已签名。

```
<PublicSign>true</PublicSign>
```

PublicSign 选项需要使用 [KeyFile](#) 或 [KeyContainer](#) 选项。keyFile 和 KeyContainer 选项指定公钥。PublicSign 和 PublicSign 选项互斥。公共签名有时称为“假签名”或“OSS 签名”，它包括输出程序集中的公钥并设置“已签名”标记。公共签名实际上并不使用私钥对程序集进行签名。开发人员为开放源代码项目使用公共签名。当人们无权访问用于对程序集进行签名的私钥时，他们会生成与已发布的“完全签名”程序集兼容的程序集。由于很少有使用者实际需要检查程序集是否完全签名，因此这些公开生成的程序集几乎适用于每个使用完全签名程序集的方案。

DelaySign

此选项会使编译器在输出文件中保留空间，以便以后添加数字签名。

```
<DelaySign>true</DelaySign>
```

如果需要完全签名的程序集，请使用 DelaySign-。如果仅需要将公钥置于程序集中，则使用 DelaySign。除非与 [KeyFile](#) 或 [KeyContainer](#) 一同使用，否则 DelaySign 选项将不起作用。[KeyContainer](#) 和 [PublicSign](#) 选项互斥。在请求完全签名的程序集时，编译器会对包含清单(程序集元数据)的文件进行哈希处理，并使用私钥对哈希进行签名。该操作创建一个数字签名，它存储在包含清单的文件中。在对程序集延迟签名时，编译器不会计算和存储签名。相反，编译器会在文件中保留空间，便于稍后添加签名。

使用 DelaySign，测试人员可以将程序集放入全局缓存中。测试完成后，可使用[程序集链接器](#)实用工具将私钥置于程序集中，对程序集进行完全签名。有关详细信息，请参阅[创建和使用具有强名称的程序集](#)和[延迟为程序集签名](#)。

KeyFile

指定包含加密密钥的文件名。

```
<KeyFile>filename</KeyFile>
```

`file` 是包含强名称密钥的文件的名称。使用此选项时，编译器在程序集清单中插入指定字段的公钥，然后使用私钥对最终的程序集进行签名。若要生成密钥文件，请在命令行键入 `sn -k file`。如果你使用 `-target:module`

进行编译，密钥文件的名称将保存在模块中，并在你使用 [AddModules](#) 编译程序集时，合并到创建的程序集中。你也可以使用 [Keycontainer](#) 将加密信息传递给编译器。如果需要部分签名的程序集，请使用 [DelaySign](#)。如果在同一编译中同时指定 KeyFile 和 KeyContainer，则编译器将首先尝试使用密钥容器。如果成功，则使用密钥容器中的信息对程序集签名。如果编译器没有找到密钥容器，它将尝试使用通过 [KeyFile](#) 指定的文件。如果成功，则使用密钥文件中的信息对程序集签名，并且将密钥信息安装到密钥容器中。在下一次编译中，密钥容器将生效。密钥文件可能仅包含公钥。有关详细信息，请参阅[创建和使用具有强名称的程序集](#)和[延迟为程序集签名](#)。

KeyContainer

指定加密密钥容器的名称。

```
<KeyContainer>container</KeyContainer>
```

`container` 是强名称密钥容器的名称。当使用 `KeyContainer` 选项时，编译器将创建一个可共享的组件。编译器在程序集清单中插入指定容器的公钥，然后使用私钥对最终的程序集进行签名。若要生成密钥文件，请在命令行键入 `sn -k file`。`sn -i` 将密钥对安装到容器中。编译器在 CoreCLR 上运行时，不支持此选项。若要在基于 CoreCLR 生成时对程序集进行签名，请使用 `KeyFile` 选项。如果你使用 `TargetType` 进行编译，那么，当你使用 `AddModules` 将此模块编译到程序集时，密钥文件的名称将保存在模块中，而且会并入程序集。还可以将此选项指定为任何 Microsoft 中间语言 (MSIL) 模块的源代码中的自定义特性 (`System.Reflection.AssemblyKeyNameAttribute`)。此外，可使用 `KeyFile` 将加密信息传递给编译器。使用 `DelaySign` 可将公钥添加到程序集清单中，但在程序集被测试之前，会对其进行签名。有关详细信息，请参阅[创建和使用具有强名称的程序集](#)和[延迟为程序集签名](#)。

HighEntropyVA

HighEntropyVA 编译器选项可告知 Windows 内核，特定的可执行文件是否支持高熵地址空间布局随机化 (ASLR)。

```
<HighEntropyVA>true</HighEntropyVA>
```

此选项指定 64 位可执行文件或由 [PlatformTarget](#) 编译器选项标记的可执行文件支持高熵虚拟地址空间。默认情况下，此选项处于禁用状态。可以使用 HighEntropyVA 启用它。

当随机化进程的地址空间布局包含在 ASLR 中时，HighEntropyVA 选项允许 Windows 内核的兼容版本使用更高程度的熵。使用更高程度的熵意味着，可向内存区域（例如堆栈或堆）分配更多的地址。因此，猜测特定内存区域的位置会更加困难。HighEntropyVA 编译器选项需要目标可执行文件及其依赖的任何模块在作为 64 位进程运行时，能够处理大于 4 GB 的指针值。

指定资源的 C# 编译器选项

2021/5/7 • [Edit Online](#)

以下选项控制 C# 编译器如何创建或导入 Win32 资源。新的 MSBuild 语法以粗体显示。旧的 csc.exe 语法以 `code style` 显示。

- **Win32Resource / `-win32res`** : 指定 Win32 资源文件 (.res)。
- **Win32Icon / `-win32icon`** : 引用来自一个或多个指定程序集文件的元数据。
- **Win32Manifest / `-win32manifest`** : 指定 Win32 清单文件 (.xml)。
- **NoWin32Manifest / `-nowin32manifest`** : 不包括默认的 Win32 清单。
- **Resources / `-resource`** : 嵌入指定的资源(简短形式:/res)。
- **LinkResources / `-linkresources`** : 将指定的资源链接到此程序集。

Win32Resource

Win32Resource 选项会在输出文件中插入 Win32 资源。

```
<Win32Resource>filename</Win32Resource>
```

`filename` 是要添加到输出文件的资源文件。Win32 资源可以包含版本或位图(图标)信息, 这些信息有助于在文件资源管理器中标识您的应用程序。如果不指定此选项, 编译器将根据程序集版本生成版本信息。

Win32Icon

Win32Icon 选项在输出文件中插入 .ico 文件, 为输出文件提供在文件资源管理器中所需的外观。

```
<Win32Icon>filename</Win32Icon>
```

`filename` 是要添加到输出文件的 .ico 文件。可以使用[资源编译器](#)创建 .ico 文件。在编译 Visual C++ 程序时会调用资源编译器;.ico 文件是从 .rc 文件创建的。

Win32Manifest

使用 Win32Manifest 选项可以指定要嵌入到项目的可移植可执行 (PE) 文件中的用户定义的 Win32 应用程序清单文件。

```
<Win32Manifest>filename</Win32Manifest>
```

`filename` 是自定义清单文件的名称和位置。默认情况下, C# 编译器嵌入可指定“asInvoker”的请求执行级别的应用程序清单。它在生成该可执行文件的同一文件夹中创建清单。如果要提供自定义清单(例如, 指定“highestAvailable”或“requireAdministrator”的请求执行级别的清单), 请使用此选项指定文件名。

NOTE

此选项和 Win32Resources 选项是互斥的。如果尝试在同一命令行中同时使用这两个选项, 将收到一个生成错误。

如果应用程序没有用于指定请求执行级别的应用程序清单, 将受到 Windows 中“用户帐户控制”功能下的文件和

注册表虚拟化的影响。有关详细信息, 请参阅[用户帐户控制](#)。

如果满足下列任一条件, 则应用程序会受到虚拟化的影响:

- 使用 NoWin32Manifest 选项, 并且在随后的生成步骤中未提供清单, 或者没有通过使用 Win32Resource 选项将其包含在 Windows 资源 (.res) 文件中。
- 提供的自定义清单未指定请求执行级别。

Visual Studio 创建默认 .manifest 文件, 并将它与可执行文件一起存储在“调试”和“发布”目录中。可以用任意文本编辑器创建一个清单, 然后将该文件添加到项目中, 从而添加自定义清单。或者, 也可以右键单击“解决方案资源管理器”中的“项目”图标, 选择“添加新项”, 然后选择“应用程序清单文件”。添加完新的或现有清单文件后, 该文件将显示在“清单”下拉列表中。有关详细信息, 请参阅[“项目设计器”->“应用程序”页 \(C#\)](#)。

提供应用程序清单的操作可以作为自定义生成后步骤, 也可以通过使用 NoWin32Manifest 选项作为 Win32 资源文件的组成部分。如果希望应用程序受到 Windows Vista 的文件或注册表虚拟化的影响, 请使用该选项。

NoWin32Manifest

使用 NoWin32Manifest 选项可指示编译器不将任何应用程序清单嵌入到可执行文件中。

```
<NoWin32Manifest />
```

使用此选项时, 除非在 Win32 资源文件或以后的生成步骤中提供应用程序清单, 否则应用程序会受到 Windows Vista 上虚拟化的影响。

在 Visual Studio 的“应用程序属性”页中, 通过在“清单”下拉列表中选择“创建不带清单的应用程序”选项来设置此选项。有关详细信息, 请参阅[“项目设计器”->“应用程序”页 \(C#\)](#)。

资源

将指定资源嵌入输出文件。

```
<Resources Include=filename>
  <LogicalName>identifier</LogicalName>
  <Access>accessibility-modifier</Access>
</Resources>
```

`filename` 是要嵌入到输出文件的 .NET 资源文件。`identifier` (可选)是资源的逻辑名称;用于加载资源的名称。默认值是文件的名称。`accessibility-modifier` (可选)是资源的可访问性:public 或 private。默认值为 public。默认情况下, 如果使用 C# 编译器创建资源, 则这些资源在程序集中是公有的。若要使资源变为私有, 请将 `private` 指定为可访问性修饰符。不允许使用 `public` 或 `private` 以外的任何其他可访问性。例如, 如果 `filename` 是由 `Resgen.exe` 创建的或在开发环境中创建的 .NET 资源文件, 则可使用 `System.Resources.ResourceManager` 命名空间中的成员来访问它。有关详细信息, 请参阅 `System.Resources.ResourceManager`。对于所有其他资源, 请使用 `Assembly` 类中的 `GetManifestResource` 方法在运行时访问资源。输出文件中资源的顺序由项目文件中所指定的顺序决定。

LinkResources

在输出文件中创建指向 .NET 资源的链接。不会在输出文件中添加资源文件。LinkResources 不同于会在输出文件中嵌入资源文件的 Resource 选项。

```
<LinkResources Include="filename">
  <LogicalName>identifier</LogicalName>
  <Access>accessibility-modifier</Access>
</LinkResources>
```

`filename` 是要从程序集链接到的 .NET 资源文件。`identifier` (可选)是资源的逻辑名称;用于加载资源的名称。默认值是文件的名称。`accessibility-modifier` (可选)是资源的可访问性:public 或 private。默认值为 public。默认情况下,如果使用 C# 编译器创建链接资源,则这些资源在程序集中是公有的。若要使资源变为私有,请将 `private` 指定为可访问性修饰符。不允许使用 `public` 或 `private` 以外的任何其他修饰符。例如,如果 `filename` 是由 [Resgen.exe](#) 创建的或在开发环境中创建的 .NET 资源文件,则可使用 [System.Resources](#) 命名空间中的成员来访问它。有关详细信息,请参阅 [System.Resources.ResourceManager](#)。对于所有其他资源,请使用 [Assembly](#) 类中的 `GetManifestResource` 方法在运行时访问资源。`filename` 中指定的文件可为任何格式。例如,你可能希望生成程序集的本机 DLL 部分,从而可将它安装到全局程序集缓存中,并且可从该程序集中的托管代码访问它。可在程序集链接器中执行相同的操作。有关详细信息,请参阅 [Al.exe\(程序集链接器\)](#) 和[使用程序集和全局程序集缓存](#)。

其他 C# 编译器选项

2021/5/7 • [Edit Online](#)

下面的选项控制其他编译器行为。新的 MSBuild 语法以粗体显示。旧的 csc.exe 语法以 `code style` 显示。

- **ResponseFiles** / `-@`: 读取响应文件以获取更多选项。
- **NoLogo** / `-nologo`: 禁止显示编译器版权消息。
- **NoConfig** / `-noconfig`: 不自动包括 CSC.RSP 文件。

ResponseFiles

通过 ResponseFiles 选项，可以指定包含编译器选项和要编译的源代码文件的文件。

```
<ResponseFiles>response_file</ResponseFiles>
```

`response_file` 指定一个文件来列出编译器选项或要编译的源代码文件。编译器选项和源代码文件将由编译器处理，就像它们在命令行中被指定一样。若要在一次编译中指定多个响应文件，请指定多个响应文件选项。在响应文件中，多个编译器选项和源代码文件可以出现在同一行中。单个编译器选项的指定必须出现在同一行中（不能跨行）。响应文件的注释可以 `#` 符号开始。从响应文件内指定编译器选项就如同在命令行发出这些命令。编译器在读取命令选项时会进行处理。命令行参数可以重写先前在响应文件中列出的选项。反之，响应文件中的选项也将重写先前在命令行或其他响应文件中列出的选项。C# 提供 `csc.rsp` 文件，该文件与 `csc.exe` 文件位于同一目录。有关响应文件格式的详细信息，请参阅 [NoConfig](#)。不能在 Visual Studio 开发环境中设置此编译器选项，也不能以编程方式对其进行更改。以下几行来自示例响应文件：

```
# build the first output file
-target:exe -out:MyExe.exe source1.cs source2.cs
```

NoLogo

NoLogo 选项可在编译器启动时禁止显示登录版权标志，并在编译期间禁止显示信息性消息。

```
<NoLogo>true</NoLogo>
```

NoConfig

NoConfig 选项会告知编译器不要使用 `csc.rsp` 文件进行编译。

```
<NoConfig>true</NoConfig>
```

`csc.rsp` 文件引用 .NET Framework 随附的所有程序集。Visual Studio .NET 开发环境包括的实际引用取决于项目类型。可以修改 `csc.rsp` 文件，并指定每次编译时应包括的其他编译器选项。如果你不需要编译器查询和使用 `csc.rsp` 文件中的设置，请指定 NoConfig。此编译器选项在 Visual Studio 中不可用，并且无法以编程方式更改。

高级 C# 编译器选项

2021/5/7 • [Edit Online](#)

以下选项支持高级方案。新的 MSBuild 语法以粗体显示。旧的 `csc.exe` 语法以 `code style` 显示。

- `MainEntryPoint`、`StartupObject` / `-main` : 指定包含入口点的类型。
- `PdbFile` / `-pdb` : 指定调试信息文件名。
- `PathMap` / `-pathmap` : 指定编译器输出的源路径名的映射。
- `ApplicationConfiguration` / `-appconfig` : 指定包含程序集绑定设置的应用程序配置文件。
- `AdditionalLibPaths` / `-lib` : 指定要在其中搜索引用的其他目录。
- `GenerateFullPaths` / `-fullpath` : 编译器生成完全限定的路径。
- `PreferredUILang` / `-preferreduilang` : 指定首选输出语言名称。
- `BaseAddress` / `-baseaddress` : 指定要生成的库的基址。
- `ChecksumAlgorithm` / `-checksumalgorithm` : 指定用于计算 PDB 中存储的源文件校验和的算法。
- `CodePage` / `-codepage` : 指定在打开源文件时使用的代码页。
- `Utf8Output` / `-utf8output` : 以 UTF-8 编码输出编译器消息。
- `FileAlignment` / `-filealign` : 指定用于输出文件节的对齐方式。
- `ErrorEndLocation` / `-errorendlocation` : 输出每个错误结尾位置的行和列。
- `NoStandardLib` / `-nostdlib` : 不引用标准库 mscorelib.dll。
- `SubsystemVersion` / `-subsystemversion` : 指定此程序集的子系统版本。
- `ModuleAssemblyName` / `-moduleassemblyname` : 此模块所属程序集的名称。

MainEntryPoint 或 StartupObject

如果多个类包含 `Main` 方法，此选项将指定包含程序入口点的类。

```
<StartupObject>MyNamespace.Program</StartupObject>
```

或

```
<MainEntryPoint>MyNamespace.Program</MainEntryPoint>
```

其中 `Program` 是包含 `Main` 方法的类型。提供的类名必须是完全限定类名；它必须包括完整命名空间（包含类），后跟类名。例如，当 `Main` 方法位于 `MyApplication.Core` 命名空间中的 `Program` 类中时，编译器选项必须为 `-main:MyApplication.Core.Program`。如果编译包含具有 `Main` 方法的多个类型，则可以指定哪个类型包含 `Main` 方法。

NOTE

此选项不能用于包含 [顶级语句](#) 的项目，即使该项目包含一个或多个 `Main` 方法也是如此。

PdbFile

`PdbFile` 编译器选项指定调试符号文件的名称和位置。`filename` 值指定调试符号文件的名称和位置。

```
<PdbFile>filename</PdbFile>
```

指定 [DebugType](#) 后, 编译器将在创建输出文件 (.exe 或 .dll) 的相同目录中创建 .pdb 文件。.pdb 文件与输出文件的名称具有相同的基文件名。PdbFile 允许为 .pdb 文件指定非默认的文件名和位置。不能在 Visual Studio 开发环境中设置此编译器选项, 也不能以编程方式对其进行更改。

PathMap

PathMap 编译器选项指定如何将物理路径映射到编译器输出的源路径名称。此选项将编译器在其上运行的计算机上的每个物理路径映射到应写入输出文件的相应路径。在下面的示例中, `path1` 是当前环境中源文件的完整路径, `sourcePath1` 是任何输出文件中替代 `path1` 的源路径。若要指定多个映射的源路径, 请用分号分隔每个路径。

```
<PathMap>path1=sourcePath1;path2=sourcePath2</PathMap>
```

编译器将源路径写入其输出, 原因如下:

1. 将 [CallerFilePathAttribute](#) 应用于可选参数时, 会将源路径替换为参数。
2. PDB 文件中嵌入的源路径。
3. PDB 文件的路径嵌入到 PE(可移植的可执行文件)文件中。

ApplicationConfiguration

ApplicationConfiguration 编译器选项使 C# 应用程序能够在程序集绑定时将程序集的应用程序配置 (app.config) 文件的位置指定为公共语言运行时 (CLR)。

```
<ApplicationConfiguration>file</ApplicationConfiguration>
```

其中 `file` 是包含程序集绑定设置的应用程序配置文件。ApplicationConfiguration 的一种用途是处理高级方案, 在这种方案中, 程序集必须同时引用特定引用程序集的 .NET Framework 版本和 .NET Framework for Silverlight 版本。例如, 在 Windows Presentation Foundation (WPF) 中编写的 XAML 设计器可能需要为设计器用户界面引用 WPF 桌面以及随附于 Silverlight 的 WPF 子集。同一设计器程序集必须访问这两个程序集。默认情况下, 单独引用会导致编译器错误, 因为程序集绑定将这两个程序集视为等效。借助 ApplicationConfiguration 编译器选项, 可通过使用 `<supportPortability>` 标记指定禁用默认行为的 app.config 文件的位置, 如以下示例所示。

```
<supportPortability PKT="7cec85d7bea7798e" enable="false"/>
```

编译器将文件的位置传递给 CLR 的程序集绑定逻辑。

NOTE

若要使用已在项目中设置的 app.config 文件, 请将属性标记 `<UseAppConfigForCompiler>` 添加到 .csproj 文件, 并将其值设置为 `true`。若要指定不同的 app.config 文件, 请添加属性标记 `<AppConfigForCompiler>` 并将其值设置为该文件的位置。

以下示例展示一个 app.config 文件, 通过使用该文件, 应用程序能够同时引用任何 .NET Framework 程序集(同时存在于后述两个实现中)的 .NET Framework 实现和 .NET Framework for Silverlight 实现。

ApplicationConfiguration 编译器选项指定此 app.config 文件的位置。

```
<configuration>
  <runtime>
    <assemblyBinding>
      <supportPortability PKT="7cec85d7bea7798e" enable="false"/>
      <supportPortability PKT="31bf3856ad364e35" enable="false"/>
    </assemblyBinding>
  </runtime>
</configuration>
```

AdditionalLibPaths

AdditionalLibPaths 选项指定通过 [References](#) 选项引用的程序集的位置。

```
<AdditionalLibPaths>dir1[,dir2]</AdditionalLibPaths>
```

其中 `dir1` 是在当前工作目录(从中调用编译器的目录)或公共语言运行时的系统目录中未找到引用程序集时, 编译器将在其中进行查找的目录。 `dir2` 是要在其中搜索程序集引用的一个或多个其他目录。用逗号分隔每个目录名称, 且中间没有空格。编译器按以下顺序搜索未完全限定的程序集引用:

1. 当前工作目录。
2. 公共语言运行时系统目录。
3. 由 AdditionalLibPaths 指定的目录。
4. 由 LIB 环境变量指定的目录。

使用 Reference 指定程序集引用。AdditionalLibPaths 是累加的。每一次指定的值都追加到以前的值中。由于程序集清单中未指定依赖程序集的路径, 因此应用程序将查找并使用全局程序集缓存中的程序集。编译器可以引用程序集并不表示公共语言运行时可以在运行时找到并加载程序集。有关运行时如何搜索引用的程序集的详细信息, 请参阅[运行时如何定位程序集](#)。

GenerateFullPaths

GenerateFullPaths 选项导致编译器在列出编译错误和警告时指定文件的完整路径。

```
<GenerateFullPaths>true</GenerateFullPaths>
```

默认情况下, 由编译所产生的错误和警告指定其中发现错误的文件的名称。GenerateFullPaths 选项导致编译器指定文件的完整路径。此编译器选项在 Visual Studio 中不可用, 并且无法以编程方式更改。

PreferredUILang

通过使用 PreferredUILang 编译器选项, 可指定 C# 编译器用于显示输出(如错误消息)的语言。

```
<PreferredUILang>language</PreferredUILang>
```

其中 `language` 是用于编译器输出的语言的[语言名称](#)。可以使用 PreferredUILang 编译器选项指定 C# 编译器用于错误消息和其他命令行输出的语言。如果未安装针对该语言的语言包, 将改用操作系统的语言设置。

BaseAddress

通过 BaseAddress 选项, 可指定加载 DLL 的首选基址。若要深入了解何时且为何要使用此选项, 请参阅 [Larry Osterman 的网络日志](#)。

```
<BaseAddress>address</BaseAddress>
```

其中 `address` 是 DLL 的基址。可将此地址指定为十进制数、十六进制数或八进制数。DLL 的默认基址由 .NET 公共语言运行时设置。此地址中的低序字将被舍入取整。例如，如果指定 `0x11110001`，它将被舍入为 `0x11110000`。要完成 DLL 的签名过程，请使用具有 `-R` 选项的 SN.EXE。

ChecksumAlgorithm

此选项控制用于对 PDB 中的源文件进行编码的校验和算法。

```
<ChecksumAlgorithm>algorithm</ChecksumAlgorithm>
```

`algorithm` 必须是 `SHA1`（默认）或 `SHA256`。

CodePage

如果所需页不是系统的当前默认代码页，则此选项指定编译期间要使用的代码页。

```
<CodePage>id</CodePage>
```

其中 `id` 是要用于编译中所有源代码文件的代码页 ID。编译器将首先尝试将所有源文件解释为 UTF-8。如果源代码文件使用除 UTF-8 以外的编码并使用除 7 位 ASCII 字符以外的字符，请使用 `CodePage` 选项指定应使用的代码页。`CodePage` 适用于编译中的所有源代码文件。有关如何查找系统上支持哪些代码页的信息，请参阅 [GetCPIInfo](#)。

Utf8Output

`Utf8Output` 选项使用 UTF-8 编码显示编译器输出。

```
<Utf8Output>true</Utf8Output>
```

在某些国际配置中，编译器输出无法在控制台上正确显示。使用 `Utf8Output` 并将编译器输出重定向到文件。

FileAlignment

`FileAlignment` 选项用于指定输出文件中各节的大小。有效值为 512、1024、2048、4096 和 8192。这些值以字节为单位。

```
<FileAlignment>number</FileAlignment>
```

在 Visual Studio 中，可以从项目的“生成”属性的“高级”页中设置 `FileAlignment` 选项。每一节都将在是 `FileAlignment` 值的倍数的边界上对齐。没有固定的默认值。如果未指定 `FileAlignment`，公共语言运行时在编译时会选取一个默认值。通过指定节的大小，可以影响输出文件的大小。修改节的大小可能对将在较小设备上运行的程序有用。使用 [DUMPBIN](#) 可查看有关输出文件中各节的信息。

ErrorEndLocation

指示编译器输出每个错误结尾位置的行和列。

```
<ErrorEndLocation>filename</ErrorEndLocation>
```

默认情况下，编译器会为所有错误和警告在源代码中写入起始位置。当此选项设置为 true 时，编译器会为每个错误和警告写入起始和结尾位置。

NoStandardLib

NoStandardLib 可防止导入 mscorelib.dll，后者定义了整个 System 命名空间。

```
<NoStandardLib>true</NoStandardLib>
```

如果你想要定义或创建自己的 System 命名空间和对象，请使用此选项。如果不指定 NoStandardLib，则 mscorelib.dll 将被导入到程序中（与指定 `<NoStandardLib>false</NoStandardLib>` 相同）。

SubsystemVersion

指定可执行文件可以运行的子系统的最低版本。大多数情况下，此选项确保该可执行文件可以使用早期 Windows 版本中未提供的安全功能。

NOTE

若要指定子系统本身，请使用 [TargetType](#) 编译器选项。

```
<SubsystemVersion>major.minor</SubsystemVersion>
```

`major.minor` 指定所需的子系统最低版本，以主版本和次要版本之间使用点标记的方式表示。例如，你可以指定应用程序不能在 Windows 7 之前的操作系统上运行。将此选项的值设置为 6.01，如本文后面的表中所述。将 `major` 和 `minor` 的值指定为整数。`minor` 版本中的前导零不会更改版本，但尾随零会。例如，6.1 和 6.01 表示相同的版本，但 6.10 表示另一个版本。建议次要版本用两位数表示，以免混淆。

下表列出了常见的 Windows 子系统版本。

WINDOWS 版本	SubVersion
Windows Server 2003	5.02
Windows Vista	6.00
Windows 7	6.01
Windows Server 2008	6.01
Windows 8	6.02

SubsystemVersion 编译器选项的默认值取决于以下列表中的条件：

- 只要设置了以下列表中的任意编译器选项，则默认值为 6.02：
 - `/target:appcontainerexe`
 - `/target:winmdobj`
 - `-platform:arm`
- 如果使用 MSBuild，面向 .NET Framework 4.5，并且未设置先前在此列表中指定的任何编译器选项，则默认值

为 6.00。

- 如果前面的条件均不符合，则默认值为 4.00。

ModuleAssemblyName

指定 .netmodule 可以访问其非公共类型的程序集的名称。

```
<ModuleAssemblyName>assembly_name</ModuleAssemblyName>
```

生成 .netmodule 时，应使用 ModuleAssemblyName 并满足以下条件：

- .netmodule 需要具有访问现有程序集中非公共类型的权限。
- 知道生成后的 .netmodule 所在程序集的名称。
- 现有程序集已经获得友元程序集访问权限，可访问将在其中生成 .netmodule 的程序集。

有关生成 .netmodule 的详细信息，请参阅模块的 [TargetType](#) 选项。有关友元程序集的详细信息，请参阅[友元程序集](#)。

C# 编译器错误

2021/5/8 • [Edit Online](#)

一些 C# 编译器错误有对应的主题，其中介绍了导致错误生成的原因，在某些情况下还介绍了错误修复方法。请执行下列操作之一，查看是否有特定错误消息的相关帮助资料。

- 如果使用的是 Visual Studio，请在“[输出窗口](#)”中选择错误号（例如 CS0029），然后按 F1 键。
- 在目录的“[按标题筛选](#)”框中键入错误号。

如果通过这些步骤无法找到有关错误的信息，请转到此页末尾，然后发送包含错误号或文本的反馈。

若要了解如何在 C# 中配置错误和警告选项，请参阅 [C# 编译器选项](#) 或 Visual Studio“[项目设计器](#)”->“[生成](#)”页 ([C#](#))。

NOTE

以下说明中的某些 Visual Studio 用户界面元素在计算机上出现的名称或位置可能会不同。这些元素取决于你所使用的 Visual Studio 版本和你所使用的设置。有关详细信息，请参阅[个性化设置 IDE](#)。

另请参阅

- [C# 编译器选项](#)
- “[项目设计器](#)”->“[生成](#)”页 ([C#](#))
- [WarningLevel \(C# 编译器选项\)](#)
- [DisabledWarnings \(C# 编译器选项\)](#)

简介

2021/5/7 • [Edit Online](#)

C#(读作“See Sharp”)是一种简单易用的新式编程语言，不仅面向对象，还类型安全。C#在C语言系列中具有其根，并将对C、c++和Java程序员立即熟悉。C#由ECMA国际标准化为`ecma-334`标准，并通过ISO/IEC作为`iso/iec 23270 standard`进行标准化。Microsoft针对.NET Framework的c#编译器是这两个标准的一致性实现。

C#是一种面向对象的语言。不仅如此，C#还进一步支持面向组件的编程。当代软件设计越来越依赖采用自描述的独立功能包形式的软件组件。此类组件的关键特征包括：为编程模型提供属性、方法和事件；包含提供组件声明性信息的特性；包含自己的文档。C#提供了语言构造，以直接支持这些概念，使c#成为一种非常自然的语言，用于创建和使用软件组件。

多个c#功能有助于构建强大的持久应用程序：`垃圾回收`会自动回收未使用的对象占用的内存，`异常处理`提供了一种结构化的可扩展方法用于错误检测和恢复；以及语言的`类型安全`设计使无法从未初始化的变量中进行读取，将数组索引到其边界之外，或执行未检查的类型转换。

C#采用统一的类型系统。所有C#类型（包括`int`和`double`等基元类型）均继承自一个根`object`类型。因此，所有类型共用一组通用运算，任何类型的值都可以一致地进行存储、传输和处理。此外，C#还支持用户定义的引用类型和值类型，从而支持对象动态分配以及轻量级结构的内嵌式存储。

为了确保C#程序和库能够持续兼容，C#设计非常注重版本控制。许多编程语言很少关注这个问题，因此，当引入新版依赖库时，用这些语言编写的程序会出现更多不必要的中断现象。C#设计中受版本控制加强直接影响的方面包括：单独的`virtual`和`override`修饰符，关于方法重载决策的规则，以及对显式接口成员声明的支持。

本章的其余部分介绍了c#语言的重要功能。尽管后面的章节介绍了以详细的方式（有时也是数学方式）的规则和例外，但本章采用的是以完整性为代价来提高清晰度和简洁性。目的是为读者提供一篇有助于编写早期程序和阅读后续章节的语言。

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：

```
using System;

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

C#源文件的文件扩展名通常为`.cs`。假设“Hello, World”程序存储在文件中`hello.cs`，则可以使用命令行通过Microsoft c#编译器编译该程序

```
csc hello.cs
```

这会生成一个名为的可执行程序集`hello.exe`。此应用程序在运行时生成的输出为

```
Hello, World
```

"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。

"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。按照约定，`Main` 静态方法是程序的入口点。

程序的输出是由 `System` 命名空间中 `Console` 类的 `WriteLine` 方法生成。此类由 .NET Framework 类库提供，默认情况下，由 Microsoft c# 编译器自动引用。请注意，c# 本身没有单独的运行时库。相反，.NET Framework 是 c# 的运行时库。

程序结构

C# 中的关键组织概念是 *** 程序**、*命名空间*、类型、成员 和 *程序集*。C# 程序由一个或多个源文件组成。程序声明类型，而类型则包含成员，并被整理到命名空间中。类型示例包括类和接口。成员示例包括字段、方法、属性和事件。编译完的 C# 程序实际上会打包到程序集中。程序集通常具有文件扩展名 `.exe` 或 `.dll`，具体取决于它们是否实现 *应用程序* 或 _ 库*。

示例

```
using System;

namespace Acme.Collections
{
    public class Stack
    {
        Entry top;

        public void Push(object data) {
            top = new Entry(top, data);
        }

        public object Pop() {
            if (top == null) throw new InvalidOperationException();
            object result = top.data;
            top = top.next;
            return result;
        }

        class Entry
        {
            public Entry next;
            public object data;

            public Entry(Entry next, object data) {
                this.next = next;
                this.data = data;
            }
        }
    }
}
```

在名为的 `Stack` 命名空间中声明一个名为的类 `Acme.Collections`。此类的完全限定的名称为 `Acme.Collections.Stack`。此类包含多个成员:一个 `top` 字段、两个方法(`Push` 和 `Pop`)和一个 `Entry` 嵌套类。`Entry` 类还包含三个成员:一个 `next` 字段、一个 `data` 字段和一个构造函数。假定示例的源代码存储在 `acme.cs` 文件中，以下命令行

```
csc /t:library acme.cs
```

将示例编译成库(不含 `Main` 入口点的代码), 并生成 `acme.dll` 程序集。

程序集包含以 *中间语言 * 为形式的可执行代码 (IL) 指令和格式为 `_ metadata *` 的符号信息。执行前, 程序集中的 IL 代码会被 .NET 公共语言运行时的实时 (JIT) 编译器自动转换成处理器专属代码。

由于程序集是包含代码和元数据的自描述功能单元, 因此无需在 C# 中使用 `#include` 指令和头文件。只需在编译程序时引用特定的程序集, 即可在 C# 程序中使用此程序集中包含的公共类型和成员。例如, 此程序使用 `acme.dll` 程序集中的 `Acme.Collections.Stack` 类:

```
using System;
using Acme.Collections;

class Test
{
    static void Main() {
        Stack s = new Stack();
        s.Push(1);
        s.Push(10);
        s.Push(100);
        Console.WriteLine(s.Pop());
        Console.WriteLine(s.Pop());
        Console.WriteLine(s.Pop());
    }
}
```

如果程序存储在文件中 `test.cs` , `test.cs` 则编译时, `acme.dll` 可以使用编译器的选项来引用程序集 `/r` :

```
csc /r:acme.dll test.cs
```

这会创建 `test.exe` 可执行程序集, 它将在运行时输出以下内容:

```
100
10
1
```

使用 C#, 可以将程序的源文本存储在多个源文件中。编译多文件 C# 程序时, 可以将所有源文件一起处理, 并且源文件可以随意相互引用。从概念上讲, 就像是所有源文件在处理前被集中到一个大文件中一样。在 C# 中, 永远都不需要使用前向声明, 因为声明顺序无关紧要(除了极少数的例外情况)。C# 并不限制源文件只能声明一种公共类型, 也不要求源文件的文件名必须与其中声明的类型相匹配。

类型和变量

C# 中有两种类型: *值类型* 和 *引用类型*。值类型的变量直接包含数据, 而引用类型的变量则存储对数据(称为“对象”的引用。对于引用类型, 两个变量可以引用同一对象;因此, 对一个变量执行的运算可能会影响另一个变量引用的对象。借助值类型, 每个变量都有自己的数据副本;因此, 对一个变量执行的运算不会影响另一个变量(`ref` 和 `out` 参数变量除外)。

C# 的值类型进一步划分为 `简单类型`、`枚举类型`、`结构类型` 和 `可以为 null 的类型`, 并且 C# 的引用类型进一步分为 `类类型`、`接口类型`、`数组类型` 和 `委托类型`。

下表概述了 C# 的类型系统。

值类型	简单类型	有符号的整型: <code>sbyte</code> 、 <code>short</code> 、 <code>int</code> 、 <code>long</code>
		无符号的整型: <code>byte</code> 、 <code>ushort</code> 、 <code>uint</code> 、 <code>ulong</code>
		Unicode 字符: <code>char</code>
		IEEE 浮点: <code>float</code> 、 <code>double</code>
		高精度小数: <code>decimal</code>
		布尔: <code>bool</code>
	枚举类型	格式为 <code>enum E {...}</code> 的用户定义类型
	结构类型	格式为 <code>struct S {...}</code> 的用户定义类型
	可为 null 的类型	值为 <code>null</code> 的其他所有值类型的扩展
引用类型	课程类型	其他所有类型的最终基类: <code>object</code>
		Unicode 字符串: <code>string</code>
		格式为 <code>class C {...}</code> 的用户定义类型
	接口类型	格式为 <code>interface I {...}</code> 的用户定义类型
	数组类型	一维和多维, 例如 <code>int[]</code> 和 <code>int[,]</code>
	委托类型	格式为的用户定义的类型, 例如 <code>delegate int D(...)</code>

八个整型类型支持带符号或不带符号格式的 8 位、16 位、32 位和 64 位值。

这两个浮点类型(`float` 和 `double`)使用32位单精度和64位双精度 IEEE 754 格式表示。

`decimal` 类型是适用于财务和货币计算的 128 位数据类型。

C# 的 `bool` 类型用于表示布尔值(或的值) `true` `false`。

C# 使用 Unicode 编码处理字符和字符串。 `char` 类型表示 UTF-16 代码单元, `string` 类型表示一系列 UTF-16 代码单元。

下表总结了 c# 的数值类型。

BIT	BITS	BIT	BIT/BIT
有符号整型	8	sbyte	-128 ... 127
	16	short	-32768 ... 32,767
	32	int	-2147483648 ... 2, 147, 483, 647
	64	long	-9223372036854775808 ... 9, 223, 372, 036, 854, 77 5, 807
无符号的整型	8	byte	0 ... 255
	16	ushort	0 ... 65, 535
	32	uint	0 ... 4,294次、967、
	64	ulong	0 ... 18,446,744,073、为 709,551,615
浮点	32	float	1.5×10^{-45} 到 3.4×10^{38} , 7位精度
	64	double	5.0×10^{-324} 到 1.7×10^{308} , 15位精度
小数	128	decimal	1.0×10^{-28} 到 7.9×10^{28} , 28位精度

C# 程序使用 **类型声明** 创建新类型。类型声明指定新类型的名称和成员。用户可定义以下五种 C# 类型：类类型、结构类型、接口类型、枚举类型和委托类型。

类类型定义包含数据成员的数据结构，(字段) 和函数成员(方法、属性和其他)。类类型支持单一继承和多形性，即派生类可以扩展和专门针对基类的机制。

结构类型类似于类类型，因为它表示包含数据成员和函数成员的结构。但是，与类不同的是，结构是值类型，不需要进行堆分配。结构类型不支持用户指定的继承，并且所有结构类型均隐式继承自类型 `object`。

接口类型将协定定义为公共函数成员的命名集。实现接口的类或结构必须提供接口的函数成员的实现。接口可以从多个基接口继承，类或结构可以实现多个接口。

委托类型表示对具有特定参数列表和返回类型的方法的引用。通过委托，可以将方法视为可分配给变量并可作为参数传递的实体。委托类似于其他一些语言中的函数指针概念，但与函数指针不同的是，委托不仅面向对象，还类型安全。

类、结构、接口和委托类型都支持泛型，因此可以用其他类型参数化。

枚举类型是具有已命名常数的不同类型。每个枚举类型都有一个基础类型，该类型必须是八个整型类型之一。枚举类型的值集与基础类型的值集相同。

C# 支持任意类型的一维和多维数组。与上述类型不同，数组类型无需先声明即可使用。相反，数组类型是通过在类型名称后面添加方括号构造而成。例如，是的一维数组，是的二维数组，是的一维数组，`int[]` `int` `int[,]` `int` `int[][]` 它是 `int` 的一维数组。

可以为 null 的类型也无需声明即可使用。对于每个不可以为 null 的值类型 `T`，都有一个对应的可以为 null 的类型 `T?`，该类型可以保存附加值 `null`。例如，`int?` 是一个可以容纳任何32位整数或值的类型 `null`。

C# 的类型系统是统一的，因此任何类型的值都可以被视为对象。每种 C# 类型都直接或间接地派生自 `object` 类类型，而 `object` 是所有类型的最终基类。只需将值视为类型 `object`，即可将引用类型的值视为对象。值类型的值通过执行 装箱 和 取消装箱 操作被视为对象。在以下示例中，`int` 值被转换成 `object`，然后又恢复成 `int`。

```
using System;

class Test
{
    static void Main()
    {
        int i = 123;
        object o = i;           // Boxing
        int j = (int)o;         // Unboxing
    }
}
```

当值类型的值转换为类型时 `object`，将分配一个对象实例（也称为“box”）来保存值，并将该值复制到该框中。相反，将 `object` 引用强制转换为值类型时，会进行检查以确定所引用的对象是否为正确的值类型的框，如果检查成功，则会将框中的值复制出来。

C# 的统一类型系统实际上意味着值类型可以“按需”变为对象。鉴于这种统一性，使用类型 `object` 的常规用途库可以与引用类型和值类型结合使用。

C# 有多种 变量，其中包括 字段、数组元素、局部变量 和 参数。变量表示存储位置，每个变量都有一种类型，用于确定可以在变量中存储的值，如下表所示。

不可以为 null 的值类型	具有精确类型的值
可以为 null 的值类型	空值或该精确类型的值
<code>object</code>	空引用、对任何引用类型的对象的引用或对任何值类型的装箱值的引用
类类型	空引用、对该类类型的实例的引用，或对派生自该类类型的类的实例的引用
接口类型	空引用、对实现接口类型的类类型的实例的引用，或对实现接口类型的值类型的装箱值的引用
数组类型	空引用、对该数组类型的实例的引用，或对兼容的数组类型实例的引用
委托类型	空引用或对该委托类型的实例的引用

表达式

表达式是从操作数和运算符构造的。表达式的运算符指明了向操作数应用的运算。运算符的示例包括 `+`、`-`、`*`、`/` 和 `new`。操作数的示例包括文本、字段、局部变量和表达式。

如果表达式包含多个运算符，则运算符的优先级控制各个运算符的计算顺序。例如，表达式 `x + y - z` 相当于计算 `x + (y * z)`，因为 `*` 运算符的优先级高于 `+` 运算符。

大部分运算符可 **重载**。借助运算符重载，可以为一个或两个操作数为用户定义类或结构类型的运算指定用户定义运算符实现代码。

下表总结了 c# 的运算符，并按优先级从高到低的顺序列出了运算符类别。同一类别的运算符具有相等的优先级。

优先级	表达式	说明
主	<code>x.m</code>	成员访问
	<code>x(...)</code>	方法和委托调用
	<code>x[...]</code>	数组和索引器访问
	<code>x++</code>	后递增
	<code>x--</code>	后递减
	<code>new T(...)</code>	对象和委托创建
	<code>new T(...){...}</code>	使用初始值设定项创建对象
	<code>new {...}</code>	匿名对象初始值设定项
	<code>new T[...]</code>	数组创建
	<code>typeof(T)</code>	获取 <code>T</code> 的 <code>System.Type</code> 对象
	<code>checked(x)</code>	在已检查的上下文中计算表达式
	<code>unchecked(x)</code>	在未检查的上下文中计算表达式
	<code>default(T)</code>	获取类型为 <code>T</code> 的默认值
	<code>delegate {...}</code>	匿名函数(匿名方法)
一元	<code>+x</code>	标识
	<code>-x</code>	否定
	<code>!x</code>	逻辑非
	<code>~x</code>	按位求反
	<code>++x</code>	前递增
	<code>--x</code>	前递减
	<code>(T)x</code>	将 <code>x</code> 显式转换为类型 <code>T</code>
	<code>await x</code>	异步等待 <code>x</code> 完成

乘法性的	$x * y$	乘法
	x / y	除法
	$x \% y$	余数
累加性	$x + y$	相加、字符串串联、委托组合
	$x - y$	相减、委托移除
移位	$x \ll y$	左移
	$x \gg y$	右移
关系和类型测试	$x < y$	小于
	$x > y$	大于
	$x \leq y$	小于或等于
	$x \geq y$	大于或等于
	$x \text{ is } T$	如果 x 是 T , 则返回 $true$; 否则, 返回 $false$
	$x \text{ as } T$	返回类型为 T 的 x ; 如果 x 的类型不是 T , 则返回 $null$
等式	$x == y$	等于
	$x != y$	不等于
逻辑与	$x \& y$	整型按位 AND, 布尔型逻辑 AND
逻辑 XOR	$x ^ y$	整型按位 XOR, 布尔型逻辑 XOR
逻辑或	$x y$	整型按位“或”, 布尔型逻辑“或”
条件“与”	$x \&& y$	y 仅当 x 为时才计算 $true$
条件“或”	$x y$	y 仅当 x 为时才计算 $false$
null 合并	$x ?? y$	如果为, 则计算结果为 y x $null$, x 否则为
条件逻辑	$x ? y : z$	如果 x 为 $true$, 则计算 y ; 如果 x 为 $false$, 则计算 z
赋值或匿名函数	$x = y$	分配

	<code>x op= y</code>	复合赋值;支持的运算符 * = / = % = += -= 为 <<= >>= & = ^ = =
	<code>(T x) => y</code>	匿名函数(lambda 表达式)

语句

程序操作使用 **语句** 进行表示。C# 支持几种不同的语句，其中许多语句是从嵌入语句的角度来定义的。

使用 **代码块**，可以在允许编写一个语句的上下文中编写多个语句。代码块是由一系列在分隔符 `{` 和 `}` 内编写的语句组成。

声明语句 用于声明局部变量和常量。

表达式语句 用于计算表达式。可用作语句的表达式包括：方法调用、使用运算符的对象分配 `new`、使用 `=` 和复合赋值运算符的赋值、使用 `++` 和 `--` 运算符和 `await` 表达式递增和递减运算。

选择语句 用于根据一些表达式的值从多个可能的语句中选择一个以供执行。这一类语句包括 `if` 和 `switch` 语句。

迭代语句 用于重复执行嵌入语句。这一类语句包括 `while`、`do`、`for` 和 `foreach` 语句。

跳转语句 用于转移控制权。这一类语句包括 `break`、`continue`、`goto`、`throw`、`return` 和 `yield` 语句。

`try ... catch` 语句用于捕获在代码块执行期间发生的异常，`try ... finally` 语句用于指定始终执行的最终代码，无论异常发生与否。

`checked` 和 `unchecked` 语句用于控制整型算术运算和转换的溢出检查上下文。

`lock` 语句用于获取给定对象的相互排斥锁定，执行语句，然后解除锁定。

`using` 语句用于获取资源，执行语句，然后释放资源。

下面是每种语句的示例

局部变量声明

```
static void Main() {
    int a;
    int b = 2, c = 3;
    a = 1;
    Console.WriteLine(a + b + c);
}
```

局部常量声明

```
static void Main() {
    const float pi = 3.1415927f;
    const int r = 25;
    Console.WriteLine(pi * r * r);
}
```

表达式语句

```
static void Main() {
    int i;
    i = 123;           // Expression statement
    Console.WriteLine(i); // Expression statement
    i++;              // Expression statement
    Console.WriteLine(i); // Expression statement
}
```

if 语句

```
static void Main(string[] args) {
    if (args.Length == 0) {
        Console.WriteLine("No arguments");
    }
    else {
        Console.WriteLine("One or more arguments");
    }
}
```

switch 语句

```
static void Main(string[] args) {
    int n = args.Length;
    switch (n) {
        case 0:
            Console.WriteLine("No arguments");
            break;
        case 1:
            Console.WriteLine("One argument");
            break;
        default:
            Console.WriteLine("{0} arguments", n);
            break;
    }
}
```

while 语句

```
static void Main(string[] args) {
    int i = 0;
    while (i < args.Length) {
        Console.WriteLine(args[i]);
        i++;
    }
}
```

do 语句

```
static void Main() {
    string s;
    do {
        s = Console.ReadLine();
        if (s != null) Console.WriteLine(s);
    } while (s != null);
}
```

for 语句

```
static void Main(string[] args) {
    for (int i = 0; i < args.Length; i++) {
        Console.WriteLine(args[i]);
    }
}
```

foreach 语句

```
static void Main(string[] args) {
    foreach (string s in args) {
        Console.WriteLine(s);
    }
}
```

break 语句

```
static void Main() {
    while (true) {
        string s = Console.ReadLine();
        if (s == null) break;
        Console.WriteLine(s);
    }
}
```

continue 语句

```
static void Main(string[] args) {
    for (int i = 0; i < args.Length; i++) {
        if (args[i].StartsWith("/")) continue;
        Console.WriteLine(args[i]);
    }
}
```

goto 语句

```
static void Main(string[] args) {
    int i = 0;
    goto check;
loop:
    Console.WriteLine(args[i++]);
check:
    if (i < args.Length) goto loop;
}
```

return 语句

```
static int Add(int a, int b) {
    return a + b;
}

static void Main() {
    Console.WriteLine(Add(1, 2));
    return;
}
```

yield 语句

```

static IEnumerable<int> Range(int from, int to) {
    for (int i = from; i < to; i++) {
        yield return i;
    }
    yield break;
}

static void Main() {
    foreach (int x in Range(-10,10)) {
        Console.WriteLine(x);
    }
}

```

throw 和 try 语句

```

static double Divide(double x, double y) {
    if (y == 0) throw new DivideByZeroException();
    return x / y;
}

static void Main(string[] args) {
    try {
        if (args.Length != 2) {
            throw new Exception("Two numbers required");
        }
        double x = double.Parse(args[0]);
        double y = double.Parse(args[1]);
        Console.WriteLine(Divide(x, y));
    }
    catch (Exception e) {
        Console.WriteLine(e.Message);
    }
    finally {
        Console.WriteLine("Good bye!");
    }
}

```

checked 和 unchecked 语句

```

static void Main() {
    int i = int.MaxValue;
    checked {
        Console.WriteLine(i + 1);          // Exception
    }
    unchecked {
        Console.WriteLine(i + 1);          // Overflow
    }
}

```

lock 语句

```
class Account
{
    decimal balance;
    public void Withdraw(decimal amount) {
        lock (this) {
            if (amount > balance) {
                throw new Exception("Insufficient funds");
            }
            balance -= amount;
        }
    }
}
```

using 语句

```
static void Main() {
    using (TextWriter w = File.CreateText("test.txt")) {
        w.WriteLine("Line one");
        w.WriteLine("Line two");
        w.WriteLine("Line three");
    }
}
```

类和对象

*类_是C#的最基本类型。类是一种数据结构，可在单个单元中将状态(字段)和操作(方法和其他函数成员)结合起来。类为动态创建的类(也称为*对象*)*实例*提供定义。类支持*继承*和*多态性*，即*派生类*可以扩展和专用化_基类*的机制。

新类使用类声明进行创建。类声明的开头是标头，指定了类的特性和修饰符、类名、基类(若指定)以及类实现的接口。标头后面是类主体，由在分隔符{}内编写的成员声明列表组成。

以下是简单类 Point 的声明：

```
public class Point
{
    public int x, y;

    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
}
```

类实例是使用 new 运算符进行创建，此运算符为新实例分配内存，调用构造函数来初始化实例，并返回对实例的引用。以下语句创建两个 Point 对象，并将对这些对象的引用存储在两个变量中：

```
Point p1 = new Point(0, 0);
Point p2 = new Point(10, 20);
```

当不再使用对象时，将自动回收对象占用的内存。既没必要，也无法在C#中显式解除分配对象。

成员

类的成员为*静态成员_或_实例成员*。静态成员属于类，而实例成员则属于对象(类实例)。

下表提供了类可以包含的成员种类的概述。

常量	与类相关联的常量值
字段	类的常量
方法	类可以执行的计算和操作
属性	与读取和写入类的已命名属性相关联的操作
索引器	与将类实例编入索引(像处理数组一样)相关联的操作
事件	类可以生成的通知
运算符	类支持的转换和表达式运算符
构造函数	初始化类实例或类本身所需的操作
析构函数	永久放弃类实例前要执行的操作
类型	类声明的嵌套类型

可访问性

每个类成员都有关联的可访问性，用于控制能够访问成员的程序文本区域。可访问性有五种可能的形式。下表汇总了这些更新。

访问权限	说明
<code>public</code>	访问不受限
<code>protected</code>	只能访问此类或派生自此类的类
<code>internal</code>	只能访问此程序
<code>protected internal</code>	只能访问此程序或派生自此类的类
<code>private</code>	只能访问此类

类型参数

类定义可能会按如下方式指定一组类型参数：在类名后面用尖括号括住类型参数名称列表。然后，可以在类声明的主体中使用类型参数来定义类成员。在以下示例中，`Pair` 的类型参数是 `TFirst` 和 `TSecond`：

```
public class Pair<TFirst,TSecond>
{
    public TFirst First;
    public TSecond Second;
}
```

声明为采用类型参数的类类型称为泛型类类型。结构、接口和委托类型也可以是泛型。

使用泛型类时，必须为每个类型参数提供类型自变量：

```
Pair<int,string> pair = new Pair<int,string> { First = 1, Second = "two" };
int i = pair.First;      // TFirst is int
string s = pair.Second; // TSecond is string
```

提供类型参数的泛型类型(如上文所示 `Pair<int,string>`)称为构造类型。

基类

类声明可能会按如下方式指定基类:在类名和类型参数后面编写冒号和基类名。省略基类规范与从 `object` 类型派生相同。在以下示例中, `Point3D` 的基类是 `Point`, `Point` 的基类是 `object`:

```
public class Point
{
    public int x, y;

    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
}

public class Point3D: Point
{
    public int z;

    public Point3D(int x, int y, int z): base(x, y) {
        this.z = z;
    }
}
```

类继承其基类的成员。继承意味着类隐式包含其基类的所有成员(实例和静态构造函数除外)和基类的析构函数。派生类可以为其继承的类添加新成员,但无法删除继承成员的定义。在上面的示例中, `Point3D` 从 `Point` 继承了 `x` 和 `y` 字段, 每个 `Point3D` 实例均包含三个字段(`x`、`y` 和 `z`)。

可以将类类型隐式转换成其任意基类类型。因此, 类类型的变量可以引用相应类的实例或任意派生类的实例。例如, 类声明如上, `Point` 类型的变量可以引用 `Point` 或 `Point3D`:

```
Point a = new Point(10, 20);
Point b = new Point3D(10, 20, 30);
```

字段

字段是与类或类实例相关联的变量。

使用修饰符声明的字段 `static` 定义 **静态字段**。静态字段只指明一个存储位置。无论创建多少个类实例,永远只有一个静态字段副本。

未使用修饰符声明的字段 `static` 定义 **实例字段**。每个类实例均包含相应类的所有实例字段的单独副本。

在以下示例中, 每个 `Color` 类实例均包含 `r`、`g` 和 `b` 实例字段的单独副本, 但分别只包含 `Black`、`White`、`Red`、`Green` 和 `Blue` 静态字段的一个副本:

```

public class Color
{
    public static readonly Color Black = new Color(0, 0, 0);
    public static readonly Color White = new Color(255, 255, 255);
    public static readonly Color Red = new Color(255, 0, 0);
    public static readonly Color Green = new Color(0, 255, 0);
    public static readonly Color Blue = new Color(0, 0, 255);
    private byte r, g, b;

    public Color(byte r, byte g, byte b) {
        this.r = r;
        this.g = g;
        this.b = b;
    }
}

```

如上面的示例所示，可以使用 `readonly` 修饰符声明 **只读字段**。对字段的赋值 `readonly` 只能作为字段声明的一部分出现，或者出现在同一类的构造函数中。

方法

方法 是实现可由对象或类执行的计算或操作的成员。通过类访问 **静态方法**。**实例方法** 可通过类的实例进行访问。

方法具有 (可能为空的 **parameters**) 列表 (表示传递给方法的值或变量引用) 和一个 **返回类型** (指定方法计算并返回的值的类型)。如果不返回值，则为方法的返回类型 `void`。

方法可能也包含一组类型参数，必须在调用方法时指定类型自变量，这一点与类型一样。与类型不同的是，通常可以根据方法调用的自变量推断出类型自变量，无需显式指定。

在声明方法的类中，方法的 **签名** 必须是唯一的。方法签名包含方法名称、类型参数数量及其参数的数量、修饰符和类型。方法签名不包含返回类型。

参数

参数用于将值或变量引用传递给方法。方法参数从调用方法时指定的 **自变量** 中获取其实际值。有四类参数：值参数、引用参数、输出参数和参数数组。

值参数 用于传递输入参数。值参数对应于局部变量，从为其传递的自变量中获取初始值。修改值参数不会影响为其传递的自变量。

可以指定默认值，从而省略相应的自变量，这样值参数就是可选的。

引用参数 用于传递输入和输出参数。为引用参数传递的自变量必须是变量，并且在方法执行期间，引用参数指定的存储位置与自变量相同。引用参数使用 `ref` 修饰符进行声明。下面的示例展示了如何使用 `ref` 参数。

```

using System;

class Test
{
    static void Swap(ref int x, ref int y) {
        int temp = x;
        x = y;
        y = temp;
    }

    static void Main() {
        int i = 1, j = 2;
        Swap(ref i, ref j);
        Console.WriteLine("{0} {1}", i, j);           // Outputs "2 1"
    }
}

```

输出参数 用于传递输出参数。输出参数与引用参数类似，不同之处在于，调用方提供的自变量的初始值并不重要。输出参数使用 `out` 修饰符进行声明。下面的示例展示了如何使用 `out` 参数。

```
using System;

class Test
{
    static void Divide(int x, int y, out int result, out int remainder) {
        result = x / y;
        remainder = x % y;
    }

    static void Main() {
        int res, rem;
        Divide(10, 3, out res, out rem);
        Console.WriteLine("{0} {1}", res, rem);      // Outputs "3 1"
    }
}
```

参数数组 允许向方法传递数量不定的自变量。参数数组使用 `params` 修饰符进行声明。参数数组只能是方法的最后一个参数，且参数数组的类型必须是一维数组类型。`System.Console` 类的 `Write` 和 `WriteLine` 方法是参数数组用法的典型示例。它们的声明方式如下。

```
public class Console
{
    public static void Write(string fmt, params object[] args) {...}
    public static void WriteLine(string fmt, params object[] args) {...}
    ...
}
```

在使用参数数组的方法中，参数数组的行为与数组类型的常规参数完全相同。不过，在调用包含参数数组的方法时，要么可以传递参数数组类型的一个自变量，要么可以传递参数数组的元素类型的任意数量自变量。在后一种情况下，数组实例会自动创建，并初始化为包含给定的自变量。以下示例：

```
Console.WriteLine("x={0} y={1} z={2}", x, y, z);
```

等同于编写以下代码：

```
string s = "x={0} y={1} z={2}";
object[] args = new object[3];
args[0] = x;
args[1] = y;
args[2] = z;
Console.WriteLine(s, args);
```

方法主体和局部变量

方法的主体指定在调用方法时要执行的语句。

方法主体可以声明特定于方法调用的变量。此类变量称为 **局部变量**。局部变量声明指定了类型名称、变量名称以及可能的初始值。下面的示例声明了初始值为零的局部变量 `i` 和无初始值的局部变量 `j`。

```

using System;

class Squares
{
    static void Main() {
        int i = 0;
        int j;
        while (i < 10) {
            j = i * i;
            Console.WriteLine("{0} x {0} = {1}", i, j);
            i = i + 1;
        }
    }
}

```

C# 要求必须先 **明确赋值** 局部变量，然后才能获取其值。例如，如果上面的 `i` 声明未包含初始值，那么编译器会在后面使用 `i` 时报告错误，因为在后面使用时 `i` 不会在程序中进行明确赋值。

方法可以使用 `return` 语句将控制权返回给调用方。在返回 `void` 的方法中，`return` 语句无法指定表达式。在返回非的方法中 `void`，`return` 语句必须包含用于计算返回值的表达式。

静态和实例方法

使用 `static` 修饰符声明的方法是静态方法。静态方法不对特定的实例起作用，只能直接访问静态成员。

未使用 `static` 修饰符声明的方法是实例方法。实例方法对特定的实例起作用，并能够访问静态和实例成员。其中调用实例方法的实例可以作为 `this` 显式访问。在静态方法中引用 `this` 会生成错误。

以下 `Entity` 类包含静态和实例成员。

```

class Entity
{
    static int nextSerialNo;
    int serialNo;

    public Entity() {
        serialNo = nextSerialNo++;
    }

    public int GetSerialNo() {
        return serialNo;
    }

    public static int GetNextSerialNo() {
        return nextSerialNo;
    }

    public static void SetNextSerialNo(int value) {
        nextSerialNo = value;
    }
}

```

每个 `Entity` 实例均有一个序列号(很可能包含此处未显示的其他一些信息)。`Entity` 构造函数(类似于实例方法)将新实例初始化为包含下一个可用的序列号。由于构造函数是实例成员，因此可以访问 `serialNo` 实例字段和 `nextSerialNo` 静态字段。

`GetNextSerialNo` 和 `SetNextSerialNo` 静态方法可以访问 `nextSerialNo` 静态字段，但如果直接访问 `serialNo` 实例字段，则会生成错误。

下例显示了 `Entity` 类的用法。

```
using System;

class Test
{
    static void Main() {
        Entity.SetNextSerialNo(1000);
        Entity e1 = new Entity();
        Entity e2 = new Entity();
        Console.WriteLine(e1.GetSerialNo());           // Outputs "1000"
        Console.WriteLine(e2.GetSerialNo());           // Outputs "1001"
        Console.WriteLine(Entity.GetNextSerialNo());   // Outputs "1002"
    }
}
```

请注意，`SetNextSerialNo` 和 `GetNextSerialNo` 静态方法是在类中调用，而 `GetSerialNo` 实例方法则是在类实例中调用。

虚方法、重写方法和抽象方法

当实例方法声明包含修饰符时 `virtual`，该方法被称为 **“虚拟方法”**。当不 `virtual` 存在修饰符时，此方法被称为 **“非虚方法”**。

调用虚方法时，调用发生的实例的 **“运行时类型”** 将确定要调用的实际方法实现。在非虚拟方法调用中，实例的 **“编译时类型”** 是确定因素。

可以在派生类中 **重写** 虚方法。当实例方法声明包含修饰符时 `override`，此方法将重写具有相同签名的继承的虚方法。但如果虚方法声明中引入新方法，重写方法声明通过提供相应方法的新实现代码，专门针对现有的继承虚方法。

抽象 方法是无实现的虚方法。抽象方法使用修饰符进行声明 `abstract`，只允许在也声明的类中使用 `abstract`。必须在所有非抽象派生类中重写抽象方法。

下面的示例声明了一个抽象类 `Expression`，用于表示表达式树节点；还声明了三个派生类（`Constant`、`VariableReference` 和 `Operation`），用于实现常量、变量引用和算术运算的表达式树节点。（这类似于，但不与 [表达式树类型](#) 中引入的表达式树类型混淆）。

```

using System;
using System.Collections;

public abstract class Expression
{
    public abstract double Evaluate(Hashtable vars);
}

public class Constant: Expression
{
    double value;

    public Constant(double value) {
        this.value = value;
    }

    public override double Evaluate(Hashtable vars) {
        return value;
    }
}

public class VariableReference: Expression
{
    string name;

    public VariableReference(string name) {
        this.name = name;
    }

    public override double Evaluate(Hashtable vars) {
        object value = vars[name];
        if (value == null) {
            throw new Exception("Unknown variable: " + name);
        }
        return Convert.ToDouble(value);
    }
}

public class Operation: Expression
{
    Expression left;
    char op;
    Expression right;

    public Operation(Expression left, char op, Expression right) {
        this.left = left;
        this.op = op;
        this.right = right;
    }

    public override double Evaluate(Hashtable vars) {
        double x = left.Evaluate(vars);
        double y = right.Evaluate(vars);
        switch (op) {
            case '+': return x + y;
            case '-': return x - y;
            case '*': return x * y;
            case '/': return x / y;
        }
        throw new Exception("Unknown operator");
    }
}

```

上面的四个类可用于进行算术表达式建模。例如，使用这些类的实例，可以按如下方式表示表达式 `x + 3`。

```
Expression e = new Operation(
    new VariableReference("x"),
    '+',
    new Constant(3));
```

调用 `Expression` 实例的 `Evaluate` 方法可以计算给定的表达式并生成 `double` 值。此方法采用作为参数的 `Hashtable`，其中包含变量名称（作为项的键）和值（为项）的值。`Evaluate` 方法是一个虚拟抽象方法，这意味着非抽象派生类必须重写它以提供实际实现。

`Constant` 的 `Evaluate` 实现代码只返回存储的常量。的 `VariableReference` 实现查找哈希表中的变量名并返回结果值。`Operation` 实现代码先计算左右操作数（以递归方式调用其 `Evaluate` 方法），然后执行给定的算术运算。

以下程序使用 `Expression` 类根据不同的 `x` 和 `y` 值计算表达式 `x * (y + 2)`。

```
using System;
using System.Collections;

class Test
{
    static void Main() {
        Expression e = new Operation(
            new VariableReference("x"),
            '*',
            new Operation(
                new VariableReference("y"),
                '+',
                new Constant(2)
            )
        );
        Hashtable vars = new Hashtable();
        vars["x"] = 3;
        vars["y"] = 5;
        Console.WriteLine(e.Evaluate(vars));           // Outputs "21"
        vars["x"] = 1.5;
        vars["y"] = 9;
        Console.WriteLine(e.Evaluate(vars));           // Outputs "16.5"
    }
}
```

方法重载

方法 *重载* 允许同一类中的多个方法具有相同的名称，只要它们具有唯一的签名。在编译重载方法的调用时，编译器使用 *重载决策* 来确定要调用的特定方法。重载决策查找与自变量最匹配的方法；如果找不到最佳匹配项，则会报告错误。下面的示例展示了重载决策的实际工作方式。`Main` 方法中每个调用的注释指明了实际调用的方法。

```

class Test
{
    static void F() {
        Console.WriteLine("F()");
    }

    static void F(object x) {
        Console.WriteLine("F(object)");
    }

    static void F(int x) {
        Console.WriteLine("F(int)");
    }

    static void F(double x) {
        Console.WriteLine("F(double)");
    }

    static void F<T>(T x) {
        Console.WriteLine("F<T>(T)");
    }

    static void F(double x, double y) {
        Console.WriteLine("F(double, double)");
    }

    static void Main() {
        F();                      // Invokes F()
        F(1);                     // Invokes F(int)
        F(1.0);                   // Invokes F(double)
        F("abc");                 // Invokes F(object)
        F((double)1);             // Invokes F(double)
        F((object)1);             // Invokes F(object)
        F<int>(1);                // Invokes F<T>(T)
        F(1, 1);                  // Invokes F(double, double)
    }
}

```

如示例所示，可以随时将自变量显式转换成确切的参数类型，并/或显式提供类型自变量，从而选择特定的方法。

其他函数成员

包含可执行代码的成员统称为类的 **函数成员**。上一部分介绍了作为主要函数成员类型的方法。本节介绍 C# 支持的其他类型的函数成员：构造函数、属性、索引器、事件、运算符和析构函数。

下面的代码演示了一个名为 `List<T>` 的泛型类，该类实现对象的可扩充列表。此类包含最常见类型函数成员的多个示例。

```

public class List<T> {
    // Constant...
    const int defaultCapacity = 4;

    // Fields...
    T[] items;
    int count;

    // Constructors...
    public List(int capacity = defaultCapacity) {
        items = new T[capacity];
    }

    // Properties...
    public int Count {
        get { return count; }
    }

    public int Capacity {

```

```

        public int Capacity {
            get {
                return items.Length;
            }
            set {
                if (value < count) value = count;
                if (value != items.Length) {
                    T[] newItems = new T[value];
                    Array.Copy(items, 0, newItems, 0, count);
                    items = newItems;
                }
            }
        }

        // Indexer...
        public T this[int index] {
            get {
                return items[index];
            }
            set {
                items[index] = value;
                OnChanged();
            }
        }

        // Methods...
        public void Add(T item) {
            if (count == Capacity) Capacity = count * 2;
            items[count] = item;
            count++;
            OnChanged();
        }
        protected virtual void OnChanged() {
            if (Changed != null) Changed(this, EventArgs.Empty);
        }
        public override bool Equals(object other) {
            return Equals(this, other as List<T>);
        }
        static bool Equals(List<T> a, List<T> b) {
            if (a == null) return b == null;
            if (b == null || a.Count != b.Count) return false;
            for (int i = 0; i < a.Count; i++) {
                if (!object.Equals(a.Items[i], b.Items[i])) {
                    return false;
                }
            }
            return true;
        }

        // Event...
        public event EventHandler Changed;

        // Operators...
        public static bool operator ==(List<T> a, List<T> b) {
            return Equals(a, b);
        }
        public static bool operator !=(List<T> a, List<T> b) {
            return !Equals(a, b);
        }
    }
}

```

构造函数

C# 支持实例和静态构造函数。***实例构造函数** 是一个成员，用于实现初始化类的实例所需的操作。**_静态构造函数*** 是一个成员，用于实现第一次加载时初始化类本身所需的操作。

构造函数的声明方式与方法一样，都没有返回类型，且与所含类同名。如果构造函数声明包含 `static` 修饰符，则声明的是静态构造函数。否则，声明的是实例构造函数。

可以重载实例构造函数。例如，`List<T>` 类声明两个实例构造函数：一个没有参数，另一个需要使用 `int` 参数。

实例构造函数使用 `new` 运算符进行调用。以下语句 `List<string>` 使用类的每个构造函数分配两个实例 `List`

。

```
List<string> list1 = new List<string>();
List<string> list2 = new List<string>(10);
```

与其他成员不同，实例构造函数不能被继承，且类中只能包含实际已声明的实例构造函数。如果没有为类提供实例构造函数，则会自动提供不含参数的空实例构造函数。

“属性”

*Properties _ 是字段的自然扩展。两者都是包含关联类型的已命名成员，用于访问字段和属性的语法也是一样的。不过，与字段不同的是，属性不指明存储位置。相反，属性具有 _ *访问器**，用于指定读取或写入它们的值时要执行的语句。

属性的声明方式与字段类似，不同之处在于声明以 `get` 取值函数和/或分隔符结尾，`set` 而不是 `{` 以 `}` 分号结束。同时具有 `get` 访问器和访问器的属性 `set` 是一个 读写属性，只有一个 `get` 访问器的属性是 只读属性，并且只有一个 `set` 访问器的属性是一个 只写属性。

`get` 访问器与具有属性类型返回值的无参数方法相对应。除了作为赋值的目标，当在表达式中引用属性时，将 `get` 调用属性的访问器来计算属性的值。

`set` 访问器对应于方法，该方法具有一个名为的参数 `value`，没有返回类型。当属性作为赋值的目标或作为或的操作数进行引用时 `++` `--`，将 `set` 使用提供新值的自变量调用访问器。

`List<T>` 类声明以下两个属性：`Count` 和 `Capacity`（分别为只读和读写）。下面的示例展示了如何使用这些属性。

```
List<string> names = new List<string>();
names.Capacity = 100;           // Invokes set accessor
int i = names.Count;          // Invokes get accessor
int j = names.Capacity;        // Invokes get accessor
```

类似于字段和方法，C# 支持实例属性和静态属性。静态属性是用修饰符声明的 `static`，实例属性是在没有它的情况下声明的。

属性的访问器可以是虚的。如果属性声明包含 `virtual`、`abstract` 或 `override` 修饰符，则适用于属性的访问器。

索引器

借助 索引器 成员，可以将对象编入索引（像处理数组一样）。索引器的声明方式与属性类似，不同之处在于，索引器成员名称格式为 `this` 后跟在分隔符 `[` 和 `]` 内写入的参数列表。这些参数在索引器的访问器中可用。类似于属性，索引器分为读写、只读和只写索引器，且索引器的访问器可以是虚的。

`List` 类声明一个需要使用 `int` 参数的读写索引器。借助索引器，可以使用 `int` 值将 `List` 实例编入索引。例如

```
List<string> names = new List<string>();
names.Add("Liz");
names.Add("Martha");
names.Add("Beth");
for (int i = 0; i < names.Count; i++) {
    string s = names[i];
    names[i] = s.ToUpper();
}
```

索引器可以进行重载。也就是说，类可以声明多个索引器，只要其参数的数量或类型不同即可。

事件

借助 **事件** 成员，类或对象可以提供通知。事件的声明方式与字段类似，区别是事件声明包括 `event` 关键字，且类型必须是委托类型。

在声明事件成员的类中，事件的行为与委托类型的字段完全相同（前提是事件不是抽象的，且不声明访问器）。字段存储对委托的引用，委托表示已添加到事件的事件处理程序。如果不存在任何事件句柄，则字段为 `null`。

`List<T>` 类声明一个 `Changed` 事件成员，指明已向列表添加了新项。此 `Changed` 事件由 `OnChanged` 虚拟方法引发，该方法首先检查事件是否 `null`（表示）没有处理程序。引发事件的概念恰恰等同于调用由事件表示的委托，因此，没有用于引发事件的特殊语言构造。

客户端通过 **事件处理程序** 响应事件。使用 `+=` 和 `-=` 运算符分别可以附加和删除事件处理程序。下面的示例展示了如何向 `List<string>` 的 `Changed` 事件附加事件处理程序。

```
using System;

class Test
{
    static int changeCount;

    static void ListChanged(object sender, EventArgs e) {
        changeCount++;
    }

    static void Main() {
        List<string> names = new List<string>();
        names.Changed += new EventHandler(ListChanged);
        names.Add("Liz");
        names.Add("Martha");
        names.Add("Beth");
        Console.WriteLine(changeCount);           // Outputs "3"
    }
}
```

对于需要控制事件的基础存储的高级方案，事件声明可以显式提供 `add` 和 `remove` 访问器，这在某种程度上与属性的 `set` 访问器类似。

运算符

运算符 是定义向类实例应用特定表达式运算符的含义的成员。可以定义三种类型的运算符：一元运算符、二元运算符和转换运算符。所有运算符都必须声明为 `public` 和 `static`。

`List<T>` 类声明两个运算符（`operator==` 和 `operator!=`），因此定义了向 `List` 实例应用这些运算符的表达式的新含义。具体而言，这些运算符定义的是两个 `List<T>` 实例的相等性（使用其 `Equals` 方法比较所包含的每个对象）。下面的示例展示了如何使用 `==` 运算符比较两个 `List<int>` 实例。

```

using System;

class Test
{
    static void Main() {
        List<int> a = new List<int>();
        a.Add(1);
        a.Add(2);
        List<int> b = new List<int>();
        b.Add(1);
        b.Add(2);
        Console.WriteLine(a == b);           // Outputs "True"
        b.Add(3);
        Console.WriteLine(a == b);           // Outputs "False"
    }
}

```

第一个 `Console.WriteLine` 输出 `True`，因为两个列表包含的对象不仅数量相同，而且值和顺序也相同。如果 `List<T>` 未定义 `operator==`，那么第一个 `Console.WriteLine` 会输出 `False`，因为 `a` 和 `b` 引用不同的 `List<int>` 实例。

析构函数

析构函数 是实现析构类的实例所需的操作的成员。析构函数不能有参数，它们不能具有可访问性修饰符，也不能被显式调用。在垃圾回收过程中会自动调用实例的析构函数。

垃圾回收器允许使用广泛的纬度来确定何时收集对象和运行析构函数。具体而言，析构函数调用的时间是不确定的，析构函数可以在任何线程上执行。出于这些原因和其他原因，类应仅在没有其他任何解决方案可行时才实现析构函数。

处理对象析构的更好方法是使用 `using` 语句。

结构

结构 是可以包含数据成员和函数成员的数据结构，这一点与类一样；与类不同的是，结构是值类型，无需进行堆分配。结构类型的变量直接存储结构数据，而类类型的变量存储对动态分配的对象的引用。结构类型不支持用户指定的继承，并且所有结构类型均隐式继承自类型 `object`。

结构对包含值语义的小型数据结构特别有用。复数、坐标系中的点或字典中的键值对都是结构的典型示例。对小型数据结构使用结构（而不是类）在应用程序执行的内存分配次数上存在巨大差异。例如，以下程序创建并初始化包含 100 个点的数组。通过将 `Point` 实现为类，可单独实例化 101 个对象，一个对象用于数组，其他所有对象分别用于 100 个元素。

```

class Point
{
    public int x, y;

    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
}

class Test
{
    static void Main() {
        Point[] points = new Point[100];
        for (int i = 0; i < 100; i++) points[i] = new Point(i, i);
    }
}

```

另一种方法是创建 `Point` 结构。

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

    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
}
```

现在，仅实例化一个对象（即用于数组的对象），`Point` 实例存储内嵌在数组中。

结构构造函数使用 `new` 运算符进行调用，但这并不表示要分配内存。结构构造函数只返回结构值本身（通常在堆栈的临时位置中），并在必要时复制此值，而非动态分配对象并返回对此对象的引用。

借助类，两个变量可以引用同一对象；因此，对一个变量执行的运算可能会影响另一个变量引用的对象。借助结构，每个变量都有自己的数据副本；因此，对一个变量执行的运算不会影响另一个变量。例如，下面的代码段生成的输出取决于 `Point` 是类还是结构。

```
Point a = new Point(10, 10);
Point b = a;
a.x = 20;
Console.WriteLine(b.x);
```

如果 `Point` 是一个类，则输出为 `20`，`a` 并且 `b` 引用相同的对象。如果 `Point` 是一个结构，则输出为，`10` 因为的赋值 `a` `b` 创建值的副本，而此副本不受对的后续赋值的影响 `a.x`。

以上示例突出显示了结构的两个限制。首先，复制整个结构通常比复制对象引用效率更低，因此通过结构进行的赋值和值参数传递可能比通过引用类型成本更高。其次，除 `ref` 和 `out` 参数以外，无法创建对结构的引用，这就表示在很多应用场景中都不能使用结构。

数组

Array_ 是一种数据结构，其中包含可通过计算索引访问的多个变量。数组中包含的变量（也称为数组的 *元素*）都属于同一类型，并且此类型称为数组的 *_元素类型**。

数组类型是引用类型，声明数组变量只是为引用数组实例预留空间。实际的数组实例是在运行时使用运算符动态创建的 `new`。`new` 运算符指定了新数组实例的长度，然后在此实例的生存期内固定使用这个长度。数组元素的索引介于 `0` 到 `Length - 1` 之间。`new` 运算符自动将数组元素初始化为其默认值（例如，所有数值类型的默认值为 `0`，所有引用类型的默认值为 `null`）。

以下示例创建 `int` 元素数组，初始化此数组，然后打印输出此数组的内容。

```
using System;

class Test
{
    static void Main() {
        int[] a = new int[10];
        for (int i = 0; i < a.Length; i++) {
            a[i] = i * i;
        }
        for (int i = 0; i < a.Length; i++) {
            Console.WriteLine("a[{0}] = {1}", i, a[i]);
        }
    }
}
```

此示例将创建一个一维数组，并对其进行操作。C# 还支持多维数组。数组类型的维度的数量（也称为数组类型的秩）是一个加在数组类型方括号之间的逗号的数目。下面的示例分配一个一维、二维和三维数组。

```
int[] a1 = new int[10];
int[,] a2 = new int[10, 5];
int[,,] a3 = new int[10, 5, 2];
```

a1 数组包含 10 个元素，a2 数组包含 50 个元素 (10×5)，a3 数组包含 100 个元素 ($10 \times 5 \times 2$)。

数组的元素类型可以是任意类型（包括数组类型）。包含数组类型元素的数组有时称为 **交错数组**，因为元素数组的长度不必全都一样。以下示例分配由 int 数组构成的数组：

```
int[][] a = new int[3][];
a[0] = new int[10];
a[1] = new int[5];
a[2] = new int[20];
```

第一行创建包含三个元素的数组，每个元素都是 int[] 类型，并且初始值均为 null。后面的代码行将这三个元素初始化为引用长度不同的各个数组实例。

通过 new 运算符，可以使用“数组初始值设定项”（在分隔符 {} 和 {} 内编写的表达式列表）指定数组元素的初始值。以下示例分配 int[]，并将其初始化为包含三个元素。

```
int[] a = new int[] {1, 2, 3};
```

请注意，数组的长度是从和之间的表达式数推断出来 {} 的 {}。局部变量和字段声明可以进一步缩短，这样就不用重新声明数组类型了。

```
int[] a = {1, 2, 3};
```

以上两个示例等同于以下示例：

```
int[] t = new int[3];
t[0] = 1;
t[1] = 2;
t[2] = 3;
int[] a = t;
```

接口

接口 定义了可由类和结构实现的协定。接口可以包含方法、属性、事件和索引器。接口不提供所定义的成员的实现代码，仅指定必须由实现接口的类或结构提供的成员。

接口可以采用 **多重继承**。在以下示例中，接口 `IComboBox` 同时继承自 `ITextBox` 和 `IListBox`。

```
interface IControl
{
    void Paint();
}

interface ITextBox: IControl
{
    void SetText(string text);
}

interface IListBox: IControl
{
    void SetItems(string[] items);
}

interface IComboBox: ITextBox, IListBox {}
```

类和结构可以实现多个接口。在以下示例中，类 `EditBox` 同时实现 `IControl` 和 `IDataBound`。

```
interface IDataBound
{
    void Bind(Binder b);
}

public class EditBox: IControl, IDataBound
{
    public void Paint() {...}
    public void Bind(Binder b) {...}
}
```

当类或结构实现特定接口时，此类或结构的实例可以隐式转换成相应的接口类型。例如

```
EditBox editBox = new EditBox();
IControl control = editBox;
IDataBound dataBound = editBox;
```

如果已知实例不是静态地实现特定接口，可以使用动态类型显式转换功能。例如，下面的语句使用动态类型强制转换来获取对象的 `IControl` 和 `IDataBound` 接口实现。由于对象的实际类型为，因此 `EditBox` 强制转换成功。

```
object obj = new EditBox();
IControl control = (IControl)obj;
IDataBound dataBound = (IDataBound)obj;
```

在以前的 `EditBox` 类中，`Paint` 接口中的方法 `IControl` 和接口中的 `Bind` 方法 `IDataBound` 是使用成员实现的 `public`。C# 还支持 **显式接口成员实现**，使用该类或结构可以避免成为成员 `public`。显式接口成员实现代码是使用完全限定的接口成员名称进行编写。例如，`EditBox` 类可以使用显式接口成员实现代码来实现 `IControl.Paint` 和 `IDataBound.Bind` 方法，如下所示。

```
public class EditBox: IControl, IDataBound
{
    void IControl.Paint() {...}
    void IDataBound.Bind(Binder b) {...}
}
```

显式接口成员只能通过接口类型进行访问。例如，`IControl.Paint` `EditBox` 只有先将 `EditBox` 引用转换为接口类型，才能调用上一个类提供的实现 `IControl`。

```
EditBox editBox = new EditBox();
editBox.Paint();                                // Error, no such method
IControl control = editBox;
control.Paint();                                // Ok
```

枚举

枚举类型 是包含一组已命名常量的独特值类型。下面的示例声明并使用名为 `Color` 三个常数值、`Red`、`Green` 和 `Blue` 的枚举类型。

```
using System;

enum Color
{
    Red,
    Green,
    Blue
}

class Test
{
    static void PrintColor(Color color) {
        switch (color) {
            case Color.Red:
                Console.WriteLine("Red");
                break;
            case Color.Green:
                Console.WriteLine("Green");
                break;
            case Color.Blue:
                Console.WriteLine("Blue");
                break;
            default:
                Console.WriteLine("Unknown color");
                break;
        }
    }

    static void Main() {
        Color c = Color.Red;
        PrintColor(c);
        PrintColor(Color.Blue);
    }
}
```

每个枚举类型都有一个对应的整型类型，称为枚举类型的 **基础类型**。不显式声明基础类型的枚举类型具有基础类型 `int`。枚举类型的存储格式和可能值的范围由其基础类型决定。枚举类型可以采用的值集不受其枚举成员限制。特别是，枚举的基础类型的任何值都可以转换为枚举类型，并且是该枚举类型的一个不同的有效值。

下面的示例声明一个名为 `Alignment` 且基础类型为的枚举类型 `sbyte`。

```
enum Alignment: sbyte
{
    Left = -1,
    Center = 0,
    Right = 1
}
```

如前面的示例所示，枚举成员声明可以包含指定成员的值的常量表达式。每个枚举成员的常数值必须在该枚举的基础类型的范围内。当枚举成员声明未显式指定一个值时，如果该成员是枚举) 类型中的第一个成员，则将其值指定为零 (值为零。

可以使用类型强制转换将枚举值转换为整数值，反之亦然。例如

```
int i = (int)Color.Blue;           // int i = 2;
Color c = (Color)2;                // Color c = Color.Blue;
```

任何枚举类型的默认值都是整数值零，转换为枚举类型。如果变量自动初始化为默认值，则这是给定给枚举类型的变量的值。为了使枚举类型的默认值易于使用，文本 `0` 隐式转换为任何枚举类型。因此，可以运行以下命令。

```
Color c = 0;
```

委托

委托类型 表示对具有特定参数列表和返回类型的方法的引用。通过委托，可以将方法视为可分配给变量并可作为参数传递的实体。委托类似于其他一些语言中的函数指针概念，但与函数指针不同的是，委托不仅面向对象，还类型安全。

下面的示例声明并使用 `Function` 委托类型。

```

using System;

delegate double Function(double x);

class Multiplier
{
    double factor;

    public Multiplier(double factor) {
        this.factor = factor;
    }

    public double Multiply(double x) {
        return x * factor;
    }
}

class Test
{
    static double Square(double x) {
        return x * x;
    }

    static double[] Apply(double[] a, Function f) {
        double[] result = new double[a.Length];
        for (int i = 0; i < a.Length; i++) result[i] = f(a[i]);
        return result;
    }

    static void Main() {
        double[] a = {0.0, 0.5, 1.0};
        double[] squares = Apply(a, Square);
        double[] sines = Apply(a, Math.Sin);
        Multiplier m = new Multiplier(2.0);
        double[] doubles = Apply(a, m.Multiply);
    }
}

```

`Function` 委托类型实例可以引用需要使用 `double` 自变量并返回 `double` 值的方法。`Apply` 方法将给定的 `Function` 应用于 `double[]` 的元素，从而返回包含结果的 `double[]`。在 `Main` 方法中，`Apply` 用于向 `double[]` 应用三个不同的函数。

委托可以引用静态方法（如上面示例中的 `Square` 或 `Math.Sin`）或实例方法（如上面示例中的 `m.Multiply`）。引用实例方法的委托还会引用特定对象，通过委托调用实例方法时，该对象会变成调用中的 `this`。

还可以使用匿名函数创建委托，这些函数是便捷创建的“内联方法”。匿名函数可以查看周围方法的局部变量。因此，可以更轻松地编写上面的乘数示例，而无需使用 `Multiplier` 类：

```
double[] doubles = Apply(a, (double x) => x * 2.0);
```

委托的一个有趣且有用的属性是，它不知道也不关心所引用的方法的类；只关心引用的方法是否具有与委托相同的参数和返回类型。

属性

C# 程序中的类型、成员和其他实体支持使用修饰符来控制其行为的某些方面。例如，方法的可访问性是由 `public`、`protected`、`internal` 和 `private` 修饰符控制。C# 整合了这种能力，以便可以将用户定义类型的声明性信息附加到程序实体，并在运行时检索此类信息。程序通过定义和使用 `特性` 来指定此附加声明性信息。

以下示例声明了 `HelpAttribute` 特性，可将其附加到程序实体，以提供指向关联文档的链接。

```

using System;

public class HelpAttribute: Attribute
{
    string url;
    string topic;

    public HelpAttribute(string url) {
        this.url = url;
    }

    public string Url {
        get { return url; }
    }

    public string Topic {
        get { return topic; }
        set { topic = value; }
    }
}

```

所有特性类均派生自 `System.Attribute` .NET Framework 提供的基类。特性的应用方式为，在相关声明前的方括号内指定特性的名称以及任意自变量。如果特性的名称以结尾 `Attribute`，则引用该属性时可以省略此部分名称。例如，可按如下方法使用 `HelpAttribute` 特性。

```

[Help("http://msdn.microsoft.com/.../MyClass.htm")]
public class Widget
{
    [Help("http://msdn.microsoft.com/.../MyClass.htm", Topic = "Display")]
    public void Display(string text) {}
}

```

此示例将附加 `HelpAttribute` 到 `Widget` 类，并将另一类附加 `HelpAttribute` 到 `Display` 类中的方法。特性类的公共构造函数控制了将特性附加到程序实体时必须提供的信息。可以通过引用特性类的公共读写属性(如上面示例对 `Topic` 属性的引用)，提供其他信息。

下面的示例演示如何使用反射在运行时检索给定程序实体的属性信息。

```

using System;
using System.Reflection;

class Test
{
    static void ShowHelp(MemberInfo member) {
        HelpAttribute a = Attribute.GetCustomAttribute(member,
            typeof(HelpAttribute)) as HelpAttribute;
        if (a == null) {
            Console.WriteLine("No help for {0}", member);
        }
        else {
            Console.WriteLine("Help for {0}:", member);
            Console.WriteLine("  Url={0}, Topic={1}", a.Url, a.Topic);
        }
    }

    static void Main() {
        ShowHelp(typeof(Widget));
        ShowHelp(typeof(Widget).GetMethod("Display"));
    }
}

```

通过反射请求获得特定特性时，将调用特性类的构造函数（由程序源提供信息），并返回生成的特性实例。如果是通过属性提供其他信息，那么在特性实例返回前，这些属性会设置为给定值。

C # 7 的模式匹配

2021/5/7 • • [Edit Online](#)

C # 的模式匹配扩展可实现与功能语言的代数数据类型和模式匹配的许多优势，但这种方式与基础语言的外观顺畅集成。基本功能包括：[记录类型](#)，这些类型的语义含义由数据形状描述;和模式匹配，这是一个新的表达式窗体，用于启用这些数据类型的极简洁的多级分解。此方法的元素通过编程语言 F# 和 Scala 中的相关功能来实现。

Is 表达式

`is` 扩展运算符以针对 [模式](#) 测试表达式。

```
relational_expression
: relational_expression 'is' pattern
;
```

这种形式的 `relational_expression` 除了 c # 规范中的现有窗体外。如果标记左侧的 `relational_expression` `is` 未指定值或没有类型，则会发生编译时错误。

该模式的每个 [标识符](#) 都引入一个新的局部变量，该局部变量在运算符 (后是 [明确赋值的](#)) `is` `true` 即 [在](#) `true` 时 [明确赋值](#)。

注意: 在技术上，和 `constant_pattern` 中的 [类型](#) 之间存在二义性 `is-expression`，其中可能是限定标识符的有效分析。我们尝试将其绑定为类型以与以前版本的语言兼容;仅当此操作失败时，我们会按我们在其他上下文中的方式解决该问题，第一件事情 (它必须是常量或) 类型。这种多义性仅出现在表达式的右侧 `is`。

模式

在 `is` 运算符和 `switch_statement` 中使用模式来表示要比较传入数据的数据的形状。模式可以是递归的，以便可以将部分数据与子模式进行匹配。

```
pattern
: declaration_pattern
| constant_pattern
| var_pattern
;

declaration_pattern
: type simple_designation
;

constant_pattern
: shift_expression
;

var_pattern
: 'var' simple_designation
;
```

注意: 在技术上，和 `constant_pattern` 中的 [类型](#) 之间存在二义性 `is-expression`，其中可能是限定标识符的有效分析。我们尝试将其绑定为类型以与以前版本的语言兼容;仅当此操作失败时，我们会按我们在其他上下文中的方式解决该问题，第一件事情 (它必须是常量或) 类型。这种多义性仅出现在表达式的右侧 `is`。

声明模式

如果测试成功，则 *declaration_pattern* 两个测试表达式是否为给定的类型并将其转换为该类型。如果 *simple_designation* 是一个标识符，则会引入给定标识符的给定类型的局部变量。当模式匹配操作的结果为 true 时，将 明确分配 该局部变量。

```
declaration_pattern
  : type simple_designation
  ;
```

此表达式的运行时语义是根据模式中的 *类型* 测试左侧 *relational_expression* 操作数的运行时类型。如果它属于此运行时类型（或某些子类型），则的结果 *is operator* 为 *true*。它声明一个由 *标识符* 命名的新局部变量，在结果为时，将为其分配左操作数的值 *true*。

左侧和给定类型的静态类型的某些组合被视为不兼容并导致编译时错误。*E T* 如果存在标识转换、隐式引用转换、装箱转换、显式引用转换或从到的取消装箱转换，则将静态类型的值称为模式与类型兼容 *E T*。如果类型为的表达式 *E* 与类型模式中的类型不兼容，则这是编译时错误。

注意：在 c# 7.1 中，我们扩展此项，以便在输入类型或类型 *T* 为开放类型时允许模式匹配操作。此段落替换为以下内容：

左侧和给定类型的静态类型的某些组合被视为不兼容并导致编译时错误。*E T* 如果存在标识转换、隐式引用转换、装箱转换、显式引用转换或从到的取消装箱转换，*E T* 或者 *E* 或 *T* 是开放类型，则将静态类型的值称为模式与类型兼容。如果类型为的表达式 *E* 与类型模式中的类型不兼容，则这是编译时错误。

声明模式对于执行引用类型的运行时类型测试非常有用，并且替换了方法

```
var v = expr as Type;
if (v != null) { // code using v }
```

稍微简单一些

```
if (expr is Type v) { // code using v }
```

如果 *类型* 是可以为 *null* 的值类型，则是错误的。

声明模式可用于测试可为 *null* 的类型的值：*Nullable<T> T T2 id* 如果值为非 *null*，并且的类型 *T2* 为 *T* 或的某个基类型或接口，*T* 则类型（或装箱）类型的值匹配。例如，在代码片段中

```
int? x = 3;
if (x is int v) { // code using v }
```

语句的条件 *if* 是 *true* 在运行时，变量在 *v* 3 块内保存类型的值 *int*。

常量模式

```
constant_pattern
  : shift_expression
  ;
```

常数模式根据常数值测试表达式的值。常数可以是任何常数表达式，如文本、声明的变量的名称 *const*、枚举常量或 *typeof* 表达式。

如果 *e* 和 *c* 均为整型类型，则当表达式的结果为时，该模式将被视为匹配 *e == c* *true*。

否则，如果返回，则认为模式是匹配的 `object.Equals(e, c)` `true`。在这种情况下，如果 `e` 的静态类型与常量的类型不兼容，则会发生编译时错误。

Var 模式

```
var_pattern
: 'var' simple_designation
;
```

表达式 `e` 与 `var_pattern` 总是匹配。换言之，与 `var` 模式匹配始终会成功。如果 `simple_designation` 是标识符，则在运行时，`e` 的值将绑定到新引入的局部变量。局部变量的类型为 `e` 的静态类型。

如果名称绑定到类型，则是错误的 `var`。

Switch 语句

`switch` 扩展了语句，以选择执行第一个具有与 `switch` 表达式相匹配的模式的块。

```
switch_label
: 'case' complex_pattern case_guard? ':'
| 'case' constant_expression case_guard? ':'
| 'default' ':'
;

case_guard
: 'when' expression
;
```

未定义模式匹配的顺序。允许编译器按顺序匹配模式，并重复使用已匹配的模式的结果来计算其他模式的匹配结果。

如果存在 `case` 保护，则其表达式的类型为 `bool`。它作为附加条件进行评估，此条件必须满足，才能认为需要满足此条件。

如果 `switch_label` 在运行时不起作用，则是错误的，因为在以前的情况下，其模式归入。[TODO: 我们应该更精确地了解编译器要求使用哪种方法才能达到这一判断。]

当且仅当该事例块精确包含一个 `switch_label` 时，在 `switch_label` 中声明的模式变量将在其 `case` 块中明确赋值。

[TODO: 应指定何时可以访问 `switch` 块。]

模式变量的范围

在模式中声明的变量的作用域如下：

- 如果模式为 `case` 标签，则该变量的作用域为 事例块。

否则，将在 `is_pattern` 表达式中声明变量，并且其作用域基于直接包含 `is_pattern` 表达式的表达式的构造，如下所示：

- 如果表达式位于 `expression-bodied lambda` 表达式中，则其作用域为 `lambda` 的主体。
- 如果表达式在 `expression-bodied` 方法或属性中，则它的作用域是方法或属性的主体。
- 如果表达式位于 `when` 子句的子句中，则 `catch` 该表达式的作用域是该 `catch` 子句。
- 如果表达式位于 `iteration_statement` 中，则它的作用域只是该语句。
- 否则，如果表达式是在其他语句形式中，则它的作用域是包含语句的作用域。

为了确定作用域，`embedded_statement` 被视为位于其自身的作用域中。例如，`if_statement` 的语法为

```
if_statement
: 'if' '(' boolean_expression ')' embedded_statement
| 'if' '(' boolean_expression ')' embedded_statement 'else' embedded_statement
;
```

因此, 如果 *if_statement* 的受控语句声明一个模式变量, 则其作用域限制为该 *embedded_statement*:

```
if (x) M(y is var z);
```

在此示例中, 的作用域 *z* 为嵌入语句 *M(y is var z)*。

其他情况下, 出于其他(原因(例如, 在参数的默认值或属性中)错误, 这两者都是错误, 因为这些上下文需要常量表达式)。

在 c# 7.3 中, 我们添加了以下上下文, 可在其中声明模式变量:

- 如果表达式在 构造函数初始值设定项 中, 则其作用域为 构造函数初始值设定项 和构造函数的主体。
- 如果表达式在字段初始值设定项中, 其作用域就是它出现的 *equals_value_clause*。
- 如果表达式所在的查询子句所指定的值将转换为 lambda 体, 则该表达式的作用域只是该表达式。

对句法歧义的更改

在某些情况下, c# 语法不明确, 语言规范说明了如何解决这些歧义:

7.6.5.2 语法多义性

简单名称(§ 7.6.3)和成员访问(第 7.6.5)的生产可以在表达式语法中带来歧义。例如, 语句:

```
F(G<A,B>(7));
```

可以解释为对 *F* 具有两个参数和的的 *G < A* 调用 *B > (7)*。另外, 它也可以解释为 *F* 使用一个自变量进行调用, 该参数是对 *G* 具有两个类型参数和一个正则参数的泛型方法的调用。

如果可以将标记序列(在上下文中)为 简单名称(§ 7.6.3), 成员访问(§ 7.6.5)或 指针成员访问(§ 18.5.2)第 0 § 4.4.1 中, 将检查紧跟结束标记的标记 *>*。如果是

```
( ) ] } : ; , . ? == != | ^
```

然后, 将 类型参数列表 保留为 简单名称、成员访问 或 指针成员访问 的一部分, 并且会丢弃标记序列的任何其他可能分析。否则, 类型参数列表 不会被视为 简单名称、成员访问 或 > 指针成员访问 的一部分, 即使没有其他可能的标记序列分析也是如此。请注意, 在 命名空间或类型名称(§3.8 中分析 类型参数列表 时, 不应用这些规则。语句

```
F(G<A,B>(7));
```

根据此规则, 将解释为 *F* 使用一个参数调用, 该参数是对 *G* 具有两个类型参数和一个正则参数的泛型方法的调用。语句

```
F(G < A, B > 7);
F(G < A, B >> 7);
```

每个都将解释为对 *F* 具有两个参数的的调用。语句

```
x = F < A > +y;
```

将解释为小于运算符、大于运算符和一元正运算符，就像编写了语句 `x = (F < A) > (+y)`，而不是使用后跟二元加运算符的 **类型参数列表** 的 **简单名称**。在语句中

```
x = y is C<T> + z;
```

标记 `C<T>` 被解释为带有 **类型参数列表** 的 **命名空间或类型名称**。

C # 7 中引入了许多更改，使这些歧义消除规则不再足以处理语言的复杂性。

Out 变量声明

现在可以在 out 参数中声明变量：

```
M(out Type name);
```

但类型可以是泛型：

```
M(out A<B> name);
```

由于参数的语言语法使用 *expression*，因此此上下文服从消除规则。在这种情况下，关闭 `>` 后跟 **标识符**，该标识符不是允许它被视为 **类型参数列表** 的标记之一。因此，我建议将 **标识符** 添加到用于触发 **类型参数列表** 的消除歧义的令牌集。

元组和析构声明

元组文本运行完全相同的问题。考虑元组表达式

```
(A < B, C > D, E < F, G > H)
```

在旧的 c # 6 用于分析参数列表的规则下，此方法将分析为具有四个元素的元组，以 `A < B` 第一个元素开头。但是，如果在析构的左侧出现这种情况，我们需要由 **标识符** 标记触发的歧义消除，如上所述：

```
(A<B,C> D, E<F,G> H) = e;
```

这是声明两个变量的析构声明，其中第一个变量的类型为 `A<B,C>`，名称为 `D`。换言之，元组文本包含两个表达式，其中每个表达式都是一个声明表达式。

为了简化规范和编译器，我建议将此元组文本作为两元素元组进行分析，无论其出现在赋值) 的左侧 (。这是上一节中所述歧义消除的自然结果。

模式匹配

模式匹配会引入一个新的上下文，在此上下文中会出现表达式类型的多义性。运算符的右侧 `is` 是一个类型。现在，它可以是类型或表达式，如果它是类型，则可以后跟标识符。从技术上讲，可以更改现有代码的含义：

```
var x = e is T < A > B;
```

可在 c # 6 规则中将其分析为

```
var x = ((e is T) < A) > B;
```

但在 "c # 7 规则" 下的 "不符合" 建议的歧义" (将被分析为

```
var x = e is T<A> B;
```

, 它声明 `B` 类型的变量 `T<A>`。幸运的是, 本机编译器和 Roslyn 编译器都有 bug, 因此它们给出了 c # 6 代码的语法错误。因此, 不需要考虑这种特殊的重大更改。

模式匹配引入了其他标记, 这些标记应促进选择类型的多义性解析。下面的现有有效 c # 6 代码的示例将中断, 无需额外消除规则:

```
var x = e is A<B> && f;           // &&
var x = e is A<B> || f;            // ||
var x = e is A<B> & f;            // &
var x = e is A<B>[];             // [
```

消除歧义规则的建议更改

我建议修订规范, 以更改歧义令牌的列表

```
( ) ] } : ; , . ? == != | ^
```

to

```
( ) ] } : ; , . ? == != | ^ && || & [
```

在某些上下文中, 我们将 标识符 视为歧义标记。这些上下文包括:要消除的标记的序列紧靠在 `is` `case` `out` 分析元组文本的第一个元素时, 或在分析元组文本的第一个元素时出现 `(`, 这种情况下, 令牌前面是 `(` 或 `:`, 标识符后跟 `,` 或元组文本的后续元素。

修改后消除规则

修改后的消除歧义规则如下所示

如果可以将标记序列 (在上下文中) 为 简单名称 (§ 7.6.3), 成员访问 (§ 7.6.5) 或 指针成员访问 (§ 18.5.2) 第 0 §4.4.1 中, 将检查紧跟在结束标记后面的标记 `>`, 查看其是否为

- 之一 `()] } : ; , . ? == != | ^ && || & [`; 或
- 关系运算符之一 `< > <= >= is as`; 或
- 上下文查询关键字显示在查询表达式中; 或
- 在某些上下文中, 我们将 标识符 视为歧义标记。这些上下文包括:要消除的标记的序列紧靠在 `is` `case` `out` 分析元组文本的第一个元素时, 或在分析元组文本的第一个元素时出现 `(`, 这种情况下, 令牌前面是 `(` 或 `:`, 标识符后跟 `,` 或元组文本的后续元素。

如果以下标记在此列表中, 或在此类上下文中为标识符, 则会将 类型参数列表 保留为 简单名称、成员访问 或 指针成员访问 的一部分, 并且会丢弃标记序列的任何其他可能分析。否则, 类型参数列表 不被视为 简单名称、成员访问 或 指针访问权限 的一部分, 即使没有其他可能的标记序列分析也是如此。请注意, 在) 命名空间或类型名称 (§3.8 中分析 类型参数列表 时, 不应用这些规则。

由于此建议造成重大更改

由于此建议的消除歧义规则，不能识别重大更改。

有趣的示例

下面是这些歧义消除规则的一些有趣的结果：

表达式 `(A < B, C > D)` 是具有两个元素的元组，每个元素都是比较。

表达式 `(A<B,C> D, E)` 是具有两个元素的元组，其中第一个元素是声明表达式。

调用 `M(A < B, C > D, E)` 具有三个参数。

调用 `M(out A<B,C> D, E)` 具有两个参数，第一个参数是 `out` 声明。

表达式 `e is A C` 使用声明表达式。

Case 标签 `case A C:` 使用声明表达式。

模式匹配的一些示例

Is-As

我们可以替换用法

```
var v = expr as Type;
if (v != null) {
    // code using v
}
```

稍微简单、直接

```
if (expr is Type v) {
    // code using v
}
```

测试可为 null

我们可以替换用法

```
Type? v = x?.y?.z;
if (v.HasValue) {
    var value = v.GetValueOrDefault();
    // code using value
}
```

稍微简单、直接

```
if (x?.y?.z is Type value) {
    // code using value
}
```

算术简化

假设我们定义一组递归类型，以表示每个单独的提议（的表达式）：

```

abstract class Expr;
class X() : Expr;
class Const(double Value) : Expr;
class Add(Expr Left, Expr Right) : Expr;
class Mult(Expr Left, Expr Right) : Expr;
class Neg(Expr Value) : Expr;

```

现在, 我们可以定义一个函数来计算表达式的 (unreduced) 导数:

```

Expr Deriv(Expr e)
{
    switch (e) {
        case X(): return Const(1);
        case Const(*): return Const(0);
        case Add(var Left, var Right):
            return Add(Deriv(Left), Deriv(Right));
        case Mult(var Left, var Right):
            return Add(Mult(Deriv(Left), Right), Mult(Left, Deriv(Right)));
        case Neg(var Value):
            return Neg(Deriv(Value));
    }
}

```

表达式 simplifier 演示位置模式:

```

Expr Simplify(Expr e)
{
    switch (e) {
        case Mult(Const(0), *): return Const(0);
        case Mult(*, Const(0)): return Const(0);
        case Mult(Const(1), var x): return Simplify(x);
        case Mult(var x, Const(1)): return Simplify(x);
        case Mult(Const(var l), Const(var r)): return Const(l*r);
        case Add(Const(0), var x): return Simplify(x);
        case Add(var x, Const(0)): return Simplify(x);
        case Add(Const(var l), Const(var r)): return Const(l+r);
        case Neg(Const(var k)): return Const(-k);
        default: return e;
    }
}

```

C# 演练

2020/11/2 • [Edit Online](#)

演练提供有关常见方案的分步说明，这使它们成为了解产品或特定功能部分的良好开端。

本部分包含指向 C# 线程演练的链接。

本节内容

- [在异步任务完成时对其进行处理](#)

演示如何使用 `async` 和 `await` 创建异步解决方案。

- [用 C# 或 Visual Basic 创建一个 Windows 运行时组件，然后从 JavaScript 中调用该组件](#)

演示如何创建 Windows 运行时类型，将其打包到 Windows 运行时组件中，然后从使用 JavaScript 为 Windows 生成的 Windows 8.x 应用商店应用调用该组件。

- [Office 编程\(C# 和 Visual Basic\)](#)

演示如何通过使用 C# 和 Visual Basic 创建 Excel 工作簿和 Word 文档。

- [创建并使用动态对象\(C# 和 Visual Basic\)](#)

演示如何创建动态公开文本文件内容的自定义对象、如何创建使用 `IronPython` 库的项目。

- [使用 Visual C# 创作复合控件](#)

演示如何创建简单复合控件以及通过继承扩展其功能。

- [创建一个利用 Visual Studio 设计时功能的 Windows 窗体控件](#)

演示如何为自定义控件创建自定义设计器。

- [使用 Visual C# 从 Windows 窗体控件继承](#)

演示如何创建简单的集成按钮控件。此按钮从标准 Windows 窗体按钮集成功能并公开自定义成员。

- [在设计时调试自定义 Windows 窗体控件](#)

描述如何调试自定义控件的设计时行为。

- [演练:使用设计操作执行常规任务](#)

演示通常执行的部分任务，如在 `TabControl` 上添加或删除选项卡、将控件停靠在其父级，以及更改 `SplitContainer` 控件的方向。

- [用 C# 编写查询 \(LINQ\)](#)

演示用于编写 LINQ 查询表达式的 C# 语言功能。

- [操作数据 \(C#\) \(LINQ to SQL\)](#)

描述在数据库中添加、修改以及删除数据的 LINQ to SQL 方案。

- [简单对象模型和查询 \(C#\) \(LINQ to SQL\)](#)

演示如何创建实体类和筛选实体类的简单查询。

- [仅使用存储过程 \(C#\) \(LINQ to SQL\)](#)

演示如何使用 LINQ to SQL 来访问数据(仅执行存储过程)。

- [跨关系查询 \(C#\) \(LINQ to SQL\)](#)

演示如何使用 LINQ to SQL 关联来表示数据库中的外键关系。

- [用 C# 编写可视化工具](#)

演示如何通过使用 C# 来编写简单的可视化工具。

相关章节

- [部署示例和演练](#)

提供常见部署方案的分步示例。

另请参阅

- [C# 编程指南](#)
- [Visual Studio 示例](#)