Dê a sua opinião sobre a experiência de download do PDF.

Documentação do C#

Saiba como escrever qualquer aplicativo usando a linguagem de programação C# na plataforma .NET.

Saiba como programar

INTRODUÇÃO

Aprenda C# | Tutoriais, cursos, vídeos e muito mais

Tutoriais de C# iniciantes em seu navegador

VIDEO

Série de vídeos iniciantes em C#

TUTORIAL

Tutoriais autoguiado

Tutorial no navegador

REFERÊNCIA

C# no Q&A

Idiomas em fóruns da comunidade técnica do .NET

C# no Stack Overflow

C# no Discord

Fundamentos

VISÃO GERAL

Um tour por C#

Dentro de um programa C#

Série de vídeos realça o C#

Estratégia de desenvolvimento de linguagem C#

CONCEITO

Sistema de tipos

Programação orientada a objetos

Técnicas funcionais

Exceções

Estilo de codificação

TUTORIAL

Exibir linha de comando

Introdução às classes

C orientado ao objeto#

Convertendo tipos

Correspondência de padrões

Usar LINQ para consultar dados

Quais as novas

NOVIDADES

Novidades no C# 14

Novidades no C# 13

Novidades no C# 12

Novidades no C# 11

TUTORIAL

Explorar tipos de registro

Explorar instruções de nível superior

REFERÊNCIA

Alterações interruptivas no compilador C#

Compatibilidade de versão

Principais conceitos

VISÃO GERAL

Estratégia de linguagem C#

Conceitos de programação

CONCEITO

LINQ (Consulta Integrada à Linguagem)

Programação assíncrona

TREINAMENTO

Aprenda C# para desenvolvedores Java

Aprenda C# para desenvolvedores de JavaScript

Aprenda C# para desenvolvedores de Python

Conceitos avançados

REFERÊNCIA

Reflexão e atributos

Árvores de expressão

Interoperabilidade nativa

Engenharia de desempenho

SDK do .NET Compiler Platform

Referência de linguagem C#

REFERÊNCIA

Referência da linguagem

Palavras-chave

Operadores e expressões

Tokens

Padrão de linguagem C#

REFERÊNCIA

Visão geral

Especificação da linguagem C#

Especificações de funcionalidades

Manter contato

REFERÊNCIA

Comunidade de desenvolvedores do .NET

YouTube

Gorjeio

Um tour pela linguagem C#

19/06/2025

A linguagem C# é a linguagem mais popular para a plataforma .NET, um ambiente de desenvolvimento gratuito, multiplataforma e de código aberto. Os programas C# podem ser executados em muitos dispositivos diferentes, desde dispositivos de Internet das Coisas (IoT) até a nuvem e todos os outros lugares. Você pode escrever aplicativos para telefones, desktops e laptops e servidores.

O C# é uma linguagem de uso geral multiplataforma que torna os desenvolvedores produtivos ao escrever um código de alto desempenho. Com milhões de desenvolvedores, o C# é a linguagem .NET mais popular. O C# tem amplo suporte no ecossistema e em todas as cargas de trabalho do .NET. Com base em princípios orientados a objetos, ele incorpora muitos recursos de outros paradigmas, especialmente a programação funcional. Recursos de baixo nível dão suporte a cenários de alta eficiência sem escrever código não seguro. A maioria dos runtimes e bibliotecas do .NET são escritos em C# e avanços no C# geralmente beneficiam todos os desenvolvedores do .NET.

C# está na família de linguagens C. A sintaxe C# é familiar se você usou C, C++, JavaScript, TypeScript ou Java. Assim como C e C++, os ponto e vírgula (`;`) definem o fim das instruções. Os identificadores C# diferenciam maiúsculas de minúsculas. C# tem o mesmo uso de colchetes, `{` e `}`, instruções de controle como `if`, `else` e `switch`, e construções de loop como `for`, e `while`. C# também possui uma instrução `foreach` para qualquer tipo de coleção.

Olá, mundo

O programa "Hello, World" é usado tradicionalmente para introduzir uma linguagem de programação. Este é para C#:

C#

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

A linha que começa com `//` é um *comentário de linha única*. Comentários de linha única em C# começam com `//` e continuam até o final da linha atual. C# também suporta *comentários multilinhas*. Comentários de várias linhas começam com `/*` e terminam com `*/`. O método `WriteLine` da classe `Console`, que está no `System` namespace, produz a saída do programa. Essa classe é fornecida pelas bibliotecas de classes padrão, que, por padrão, são referenciadas automaticamente em todos os programas C#. Outro formulário de programa exige que você

declare a classe e o método que contêm o ponto de entrada do programa. O compilador sintetiza esses elementos quando você usa instruções de nível superior.

Esse formato alternativo ainda é válido e contém muitos dos conceitos básicos em todos os programas C#. Muitos exemplos de C# existentes usam o seguinte formato equivalente:

```C#
using System;
namespace TourOfCsharp;

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

O programa anterior "Olá, Mundo" começa com uma `using` diretiva que faz referência ao `System` namespace. Namespaces fornecem um meio hierárquico de organizar bibliotecas e programas em C#. Namespaces contêm tipos e outros namespaces—por exemplo, o namespace `System` contém muitos tipos, como a classe `Console` referenciada no programa, e muitos outros namespaces, como `IO` e `Collections`. A diretiva `using` que faz referência a um determinado namespace permite o uso não qualificado dos tipos que são membros desse namespace. Devido à diretiva `using`, o programa pode usar `Console.WriteLine` como um atalho para `System.Console.WriteLine`. No exemplo anterior, esse namespace foi implicitamente incluído.

A classe `Program` declarada pelo programa "Hello, World" tem um único membro, o método chamado `Main`. O método `Main` é declarado com o modificador `static`. Embora os métodos de instância possam fazer referência a uma determinada instância de objeto delimitador usando a palavra-chave `this`, métodos estáticos operam sem referência a um objeto específico. Por convenção, quando não há instruções de nível superior, um método estático chamado `Main` serve como ponto de entrada de um programa C#. A classe que contém o `Main` método normalmente é nomeada `Program`.

> **Dica**
>
> Os exemplos neste artigo fornecem uma primeira visão do código C#. Alguns exemplos podem mostrar elementos de C# com os quais você não está familiarizado. Quando estiver pronto para aprender C#, comece com nossos tutoriais para iniciantes ou

mergulhe nos links de cada seção. Se você tiver experiência em Java, JavaScript, TypeScript ou Python, leia nossas dicas para ajudá-lo a encontrar as informações necessárias para aprender rapidamente o C#.

Recursos familiares do C#

C# é acessível para iniciantes, mas oferece recursos avançados para desenvolvedores experientes que escrevem aplicativos especializados. Você pode ser produtivo rapidamente. Você pode aprender técnicas mais especializadas conforme necessário para suas aplicações.

Os aplicativos C# se beneficiam do gerenciamento automático de memória do .NET Runtime. Os aplicativos C# também usam as extensas bibliotecas de tempo de execução fornecidas pelo .NET SDK. Alguns componentes são independentes de plataforma, como bibliotecas de sistemas de arquivos, coleções de dados e bibliotecas matemáticas. Outras são específicas para uma única carga de trabalho, como as bibliotecas Web ASP.NET Core ou a biblioteca .NET MAUI UI. Um rico ecossistema de código aberto no NuGet aumenta as bibliotecas que fazem parte do tempo de execução. Essas bibliotecas fornecem ainda mais componentes que você pode usar.

C# é uma linguagem *fortemente tipada*. Cada variável que você declara possui um tipo conhecido em tempo de compilação. O compilador ou ferramentas de edição informam se você está usando esse tipo incorretamente. Você pode corrigir esses erros antes mesmo de executar o programa. Tipos de dados fundamentais são integrados à linguagem e ao tempo de execução: tipos de valor como `int`, `double`, `char`, tipos de referência como `string`, matrizes e outras coleções. Ao escrever seus programas, você cria seus próprios tipos. Esses tipos podem ser `struct` tipos para valores ou `class` tipos que definem o comportamento orientado a objetos. Você pode adicionar o modificador `record` aos tipos `struct` ou `class` para que o compilador sintetize o código para comparações de igualdade. Você também pode criar definições `interface`, que definem um contrato, ou um conjunto de membros, que um tipo que implementa essa interface deve fornecer. Você também pode definir tipos e métodos genéricos. Genéricos usam *parâmetros de tipo* para fornecer um espaço reservado para um tipo real quando usados.

Ao escrever o código, você define funções, também chamadas de métodos, como membros dos tipos `struct` e `class`. Esses métodos definem o comportamento dos seus tipos. Os métodos podem estar sobrecarregados, com diferentes números ou tipos de parâmetros. Os métodos podem opcionalmente retornar um valor. Além dos métodos, os tipos C# podem ter propriedades, que são elementos de dados apoiados por funções chamadas *acessadores*. Os tipos C# podem definir events, que permitem que um tipo notifique os assinantes sobre ações

importantes. C# oferece suporte a técnicas orientadas a objetos, como herança e polimorfismo para tipos `class`.

Os aplicativos C# usam exceções para relatar e tratar erros. Essa prática é familiar se você usou C++ ou Java. Seu código lança uma exceção quando não consegue fazer o que foi planejado. Outro código, não importa quantos níveis subam na pilha de chamadas, pode opcionalmente ser recuperado usando um bloco `try` - `catch`.

Recursos distintos do C#

Alguns elementos do C# podem ser menos familiares.

C# fornece correspondência de padrões. Essas expressões permitem inspecionar dados e tomar decisões com base em suas características. A correspondência de padrões fornece uma ótima sintaxe para fluxo de controle baseado em dados. O código a seguir mostra como os métodos para as operações booleanas *e*, *or* e *xor* podem ser expressos usando a sintaxe de correspondência de padrões:

C#

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

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

As expressões de correspondência de padrões podem ser simplificadas usando `_` como um resumo para qualquer valor. O exemplo a seguir mostra como você pode simplificar os métodos *e*:

C#

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

Os exemplos anteriores também declaram *tuplas*, que são estruturas de dados leves. Uma *tupla* é uma sequência ordenada e de comprimento fixo de valores com nomes opcionais e tipos específicos. Você delimita a sequência com os caracteres `(` e `)`. A declaração `(left, right)` define uma tupla com dois valores boolianos: `left` e `right`. Cada braço de comutador declara um valor de tupla, como `(true, true)`. As tuplas fornecem sintaxe conveniente para declarar um único valor com vários valores.

As expressões de coleção fornecem uma sintaxe comum para fornecer valores de coleção. Você escreve valores ou expressões entre os caracteres `[` e `]`, e o compilador converte essa expressão no tipo de coleção necessário.

C#

```
int[] numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];
List<string> names = ["Alice", "Bob", "Charlie", "David"];

IEnumerable<int> moreNumbers = [.. numbers, 11, 12, 13];
IEnumerable<string> empty = [];
```

O exemplo anterior mostra diferentes tipos de coleção que podem ser inicializados usando expressões de coleção. Um exemplo usa a `[]` expressão de coleção vazia para declarar uma coleção vazia. Outro exemplo usa o `..` *elemento spread* para expandir uma coleção e adicionar todos os seus valores à expressão de coleção.

Você pode usar expressões de *índice* e *intervalo* para recuperar um ou mais elementos de uma coleção indexável:

C#

```
string second = names[1]; // 0-based index
string last = names[^1]; // ^1 is the last element
```

```
int[] smallNumbers = numbers[0..5]; // 0 to 4
```

O `^` índice indica *desde o final* e não desde o início. O elemento `^0` está logo após o final da coleção, então `^1` é o último elemento. A `..` em uma expressão de intervalo indica o intervalo de elementos a serem incluídos. O intervalo começa com o primeiro índice e inclui todos os elementos até, mas não incluindo, o elemento no último índice.

Consulta integrada de linguagem (LINQ) fornece uma sintaxe comum baseada em padrões para consultar ou transformar qualquer coleção de dados. LINQ unifica a sintaxe para consultar coleções na memória, dados estruturados como XML ou JSON, armazenamento de banco de dados e até mesmo APIs de dados baseadas em nuvem. Você aprende um conjunto de sintaxe e pode pesquisar e manipular dados independentemente de seu armazenamento. A consulta a seguir encontra todos os alunos cuja média de notas é superior a 3,5:

C#

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

A consulta anterior funciona para muitos tipos de armazenamento representados por `Students`. Pode ser uma coleção de objetos, uma tabela de banco de dados, um blob de armazenamento em nuvem ou uma estrutura XML. A mesma sintaxe de consulta funciona para todos os tipos de armazenamento.

O modelo de programação assíncrona baseado em tarefas permite que você escreva código que parece ser executado de forma síncrona, mesmo que seja executado de forma assíncrona. Ele utiliza as palavras-chave `async` e `await` para descrever métodos assíncronos e quando uma expressão é avaliada de forma assíncrona. O exemplo a seguir aguarda uma solicitação da Web assíncrona. Quando a operação assíncrona for concluída, o método retornará o comprimento da resposta:

C#

```
public static async Task<int> GetPageLengthAsync(string endpoint)
{
    var client = new HttpClient();
    var uri = new Uri(endpoint);
    byte[] content = await client.GetByteArrayAsync(uri);
    return content.Length;
}
```

C# também oferece suporte a uma instrução `await foreach` para iterar uma coleção apoiada por uma operação assíncrona, como uma API de paginação GraphQL. O exemplo a seguir lê

dados em partes, retornando um iterador que fornece acesso a cada elemento quando estiver disponível:

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

Os chamadores podem iterar a coleção usando uma instrução `await foreach`:

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

Finalmente, como parte do ecossistema .NET, você pode usar o Visual Studio ou o Visual Studio Code com o C# DevKit. Essas ferramentas fornecem uma compreensão avançada de C#, incluindo o código que você escreve. Eles também fornecem recursos de depuração.

Introdução ao C#

Artigo • 05/05/2025

Bem-vindo à introdução aos tutoriais do C#. Essas lições começam com o código interativo que você pode executar no navegador. Você pode aprender as noções básicas do C# na série de vídeos C# for Beginners antes de iniciar essas lições interativas. https://www.youtube-nocookie.com/embed/9THmGiSPjBQ?si=3kUKFtOMLpEzeq7J

As primeiras lições explicam os conceitos de C# usando pequenos snippets de código. Você aprenderá os conceitos básicos da sintaxe C# e como trabalhar com tipos de dados, como cadeias de caracteres, números e boolianos. É tudo interativo e você escreverá e executará código em poucos minutos. Essas primeiras lições não pressupõem nenhum conhecimento prévio sobre programação ou a linguagem C#. Cada lição se baseia nas lições anteriores. Você deve fazê-los em ordem. No entanto, se você tiver alguma experiência de programação, poderá ignorar ou deslizar as primeiras lições e começar com novos conceitos.

Você pode experimentar esses tutoriais em ambientes diferentes. Os conceitos que você aprenderá são os mesmos. A diferença é qual experiência você prefere:

- No navegador, na plataforma de documentos: essa experiência insere uma janela de código C# executável em páginas de documentos. Você escreve e executa código C# no navegador.
- Na experiência de treinamento do Microsoft Learn. Este roteiro de aprendizagem contém vários módulos que ensinam as noções básicas do C#.

Olá, mundo

No tutorial Hello world, você criará o programa C# mais básico. Você explorará o tipo `string` e como manipular texto. Você também pode usar o caminho no treinamento do Microsoft Learn.

Números em C#

No tutorial Números no C# tutorial, você aprenderá como os computadores armazenam números e como executar cálculos com diferentes tipos numéricos. Você aprenderá os conceitos básicos do arredondamento e como executar cálculos matemáticos usando C#.

Tuplas e tipos

No tutorial Tuplas e tipos , você aprenderá a criar tipos em C#. Você pode criar tipos de *tuplas*, *registros*, *struct* e *classes*. Os recursos desses diferentes tipos de tipos refletem seus diferentes usos.

Loops e branches

O tutorial Branches e loops ensina os conceitos básicos da seleção de caminhos diferentes de execução de código com base nos valores armazenados em variáveis. Você aprenderá os conceitos básicos do fluxo de controle, que é a base de como os programas tomam decisões e escolhem ações diferentes.

Coleção de listas

A lição Coleções de lista fornece um tour pelo tipo Coleções de lista que armazena as sequências de dados. Você aprenderá a adicionar e remover itens, pesquisar itens e classificar as listas. Você explorará diferentes tipos de listas.

Correspondência de padrões

A lição de correspondência de padrões fornece uma introdução à *correspondência de padrões*. A correspondência de padrões permite comparar uma expressão com um padrão. O sucesso da correspondência determina qual lógica de programa deve ser seguida. Os padrões podem comparar tipos, propriedades de um tipo ou conteúdo de uma lista. Você pode combinar vários padrões usando `and`, `or` e `not` lógica. Os padrões fornecem um vocabulário avançado para inspecionar dados e tomar decisões em seu programa com base nessa inspeção.

Configurar o ambiente local

Depois de concluir esses tutoriais, configure um ambiente de desenvolvimento. Você desejará:

- O .NET SDK mais recente
- Editor do Visual Studio Code
- O DevKit C#

Instruções de instalação

No Windows, esse Arquivo de configuração do WinGet para instalar todos os pré-requisitos. Se você já tiver algo instalado, o WinGet ignorará essa etapa.

1. Baixe o arquivo e clique duas vezes para executá-lo.
2. Leia o contrato de licença, digite `y` e selecione `Enter` quando solicitado a aceitar.
3. Se você receber um prompt de UAC (Controle de Conta de Usuário) piscando na barra de tarefas, permita que a instalação continue.

Em outras plataformas, você precisa instalar cada um desses componentes separadamente.

1. Baixe o instalador recomendado na página de download do SDK do .NET e clique duas vezes para executá-lo. A página de download detecta sua plataforma e recomenda o instalador mais recente para sua plataforma.
2. Baixe o instalador mais recente do Visual Studio Code home page e clique duas vezes para executá-lo. Essa página também detecta sua plataforma e o link deve estar correto para o sistema.
3. Clique no botão "Instalar" na página de extensão do DevKit em C#. Isso abre o código do Visual Studio e pergunta se você deseja instalar ou habilitar a extensão. Selecione "instalar".

Introduction to C# - interactive tutorial

12/03/2025

This tutorial teaches you C# interactively, using your browser to write C# and see the results of compiling and running your code. It contains a series of lessons that begin with a "Hello World" program. These lessons teach you the fundamentals of the C# language.

> **Dica**
>
> When a code snippet block includes the "Run" button, that button opens the interactive window, or replaces the existing code in the interactive window. When the snippet doesn't include a "Run" button, you can copy the code and add it to the current interactive window.

Run your first program

Run the following code in the interactive window.

```C#
Console.WriteLine("Hello, World!");
```

Congratulations! You ran your first C# program. It's a simple program that prints the message "Hello World!" It used the Console.WriteLine method to print that message. `Console` is a type that represents the console window. `WriteLine` is a method of the `Console` type that prints a line of text to that text console.

Let's move on and explore more. The rest of this lesson explores working with the `string` type, which represents text in C#. Like the `Console` type, the `string` type has methods. The `string` methods work with text.

Declare and use variables

Your first program printed the `string` "Hello World!" on the screen.

> **Dica**
>
> As you explore C# (or any programming language), you make mistakes when you write code. The **compiler** finds those errors and report them to you. When the output contains

error messages, look closely at the example code, and the code in the interactive window to see what to fix. That exercise helps you learn the structure of C# code.

Your first program is limited to printing one message. You can write more useful programs by using *variables*. A *variable* is a symbol you can use to run the same code with different values. Let's try it! Start with the following code:

```C#
string aFriend = "Bill";
Console.WriteLine(aFriend);
```

The first line declares a variable, `aFriend`, and assigns it a value, "Bill". The second line prints the name.

You can assign different values to any variable you declare. You can change the name to one of your friends. Add these two lines in the preceding interactive window following the code you already added. Make sure you keep the declaration of the `aFriend` variable and its initial assignment.

> **ⓘ Importante**
>
> Don't delete the declaration of `aFriend`. Add the following code at the end of the preceding interactive window:

```C#
aFriend = "Maira";
Console.WriteLine(aFriend);
```

Notice that the same line of code prints two different messages, based on the value stored in the `aFriend` variable.

You might notice that the word "Hello" was missing in the last two messages. Let's fix that now. Modify the lines that print the message to the following code:

```C#
Console.WriteLine("Hello " + aFriend);
```

Select **Run** again to see the results.

You've been using `+` to build strings from **variables** and **constant** strings. There's a better way. You can place a variable between `{` and `}` characters to tell C# to replace that text with the value of the variable.

This process is called String interpolation.

If you add a `$` before the opening quote of the string, you can then include variables, like `aFriend`, inside the string between curly braces. Give it a try:

Select **Run** again to see the results. Instead of "Hello {aFriend}", the message should be "Hello Maira".

```
C#

Console.WriteLine($"Hello {aFriend}");
```

Work with strings

Your last edit was our first look at what you can do with strings. Let's explore more.

You're not limited to a single variable between the curly braces. Try the following code:

```
C#

string firstFriend = "Maria";
string secondFriend = "Sage";
Console.WriteLine($"My friends are {firstFriend} and {secondFriend}");
```

Strings are more than a collection of letters. You can find the length of a string using `Length`. `Length` is a **property** of a string and it returns the number of characters in that string. Add the following code at the bottom of the interactive window:

```
C#

Console.WriteLine($"The name {firstFriend} has {firstFriend.Length} letters.");
Console.WriteLine($"The name {secondFriend} has {secondFriend.Length} letters.");
```

Dica

Now is a good time to explore on your own. You learned that `Console.WriteLine()` writes text to the screen. You learned how to declare variables and concatenate strings together. Experiment in the interactive window. The window has a feature called *IntelliSense* that

makes suggestions for what you can do. Type a `.` after the `d` in `firstFriend`. You see a list of suggestions for properties and methods you can use.

You've been using a *method*, Console.WriteLine, to print messages. A *method* is a block of code that implements some action. It has a name, so you can access it.

Trim

Suppose your strings have leading or trailing spaces that you don't want to display. You want to **trim** the spaces from the strings. The Trim method and related methods TrimStart and TrimEnd do that work. You can just use those methods to remove leading and trailing spaces. Try the following code:

C#

```
string greeting = "      Hello World!       ";
Console.WriteLine($"[{greeting}]");

string trimmedGreeting = greeting.TrimStart();
Console.WriteLine($"[{trimmedGreeting}]");

trimmedGreeting = greeting.TrimEnd();
Console.WriteLine($"[{trimmedGreeting}]");

trimmedGreeting = greeting.Trim();
Console.WriteLine($"[{trimmedGreeting}]");
```

The square brackets `[` and `]` help visualize what the `Trim`, `TrimStart,` and, `TrimEnd` methods do. The brackets show where whitespace starts and ends.

This sample reinforces a couple of important concepts for working with strings. The methods that manipulate strings return new string objects rather than making modifications in place. You can see that each call to any of the `Trim` methods returns a new string but doesn't change the original message.

Replace

There are other methods available to work with a string. For example, you probably used a search and replace command in an editor or word processor before. The Replace method does something similar in a string. It searches for a substring and replaces it with different text. The Replace method takes two **parameters**. These parameters are the strings between the parentheses. The first string is the text to search for. The second string is the text to replace it

with. Try it for yourself. Add this code. Type it in to see the hints as you start typing `.Re` after the `sayHello` variable:

```
C#

string sayHello = "Hello World!";
Console.WriteLine(sayHello);
sayHello = sayHello.Replace("Hello", "Greetings");
Console.WriteLine(sayHello);
```

Two other useful methods make a string ALL CAPS or all lower case. Try the following code. Type it in to see how **IntelliSense** provides hints as you start to type `To`:

```
C#

Console.WriteLine(sayHello.ToUpper());
Console.WriteLine(sayHello.ToLower());
```

Search strings

The other part of a *search and replace* operation is to find text in a string. You can use the Contains method for searching. It tells you if a string contains a substring inside it. Try the following code to explore Contains:

```
C#

string songLyrics = "You say goodbye, and I say hello";
Console.WriteLine(songLyrics.Contains("goodbye"));
Console.WriteLine(songLyrics.Contains("greetings"));
```

The Contains method returns a *boolean* value which tells you if the string you were searching for was found. A *boolean* stores either a `true` or a `false` value. When displayed as text output, they're capitalized: `True` and `False`, respectively. You learn more about *boolean* values in a later lesson.

Challenge

There are two similar methods, StartsWith and EndsWith that also search for substrings in a string. These methods find a substring at the beginning or the end of the string. Try to modify the previous sample to use StartsWith and EndsWith instead of Contains. Search for "You" or "goodbye" at the beginning of a string. Search for "hello" or "goodbye" at the end of a string.

> ⓘ **Observação**
>
> Watch your punctuation when you test for the text at the end of the string. If the string ends with a period, you must check for a string that ends with a period.

You should get `true` for starting with "You" and ending with "hello" and `false` for starting with or ending with "goodbye".

Did you come up with something like the following (expand to see the answer):
▼ Details
:::code language="csharp" interactive="try-dotnet-method" source="./snippets/HelloWorld/Program.cs" id="Challenge":::

You completed the "Hello C#" introduction to C# tutorial. You can select the **Numbers in C#** tutorial to start the next interactive tutorial, or you can visit the .NET site to download the .NET SDK, create a project on your machine, and keep coding. The "Next steps" section brings you back to these tutorials.

For further reading on the `string` type:

- C# programming guide article on strings.
- How to tips on working with strings.

Como usar números inteiros e de ponto flutuante em C#

Artigo • 16/04/2025

Este tutorial ensina sobre os tipos numéricos em C#. Você escreve pequenas quantidades de código e, em seguida, compila e executa esse código. O tutorial contém uma série de lições que exploram números e operações matemáticas em C#. Essas lições ensinam os conceitos básicos da linguagem C#.

> **Dica**
>
> Quando um bloco de snippet de código inclui o botão "Executar", esse botão abre a janela interativa ou substitui o código existente na janela interativa. Quando o snippet não inclui um botão "Executar", você pode copiar o código e adicioná-lo à janela interativa atual.

Explorar aritmética de inteiros

Execute o código a seguir na janela interativa.

C#

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

O código anterior demonstra operações matemáticas fundamentais com inteiros. O `int` tipo representa um **inteiro**, um número inteiro positivo ou negativo. Use o `+` símbolo para adição. Outras operações matemáticas comuns para inteiros incluem:

- `-` para subtração
- `*` para multiplicação
- `/` para divisão

Comece explorando essas operações diferentes. Modifique a terceira linha para experimentar cada uma dessas operações. Por exemplo, para tentar a subtração, substitua o `+` por um `-` conforme mostrado na linha seguinte.

C#

```
int c = a - b;
```

Experimente. Selecione o botão "Executar". Em seguida, tente a multiplicação `*` e a divisão `/`. Você também pode experimentar escrevendo várias operações matemáticas na mesma linha, se desejar.

> **Dica**
>
> Ao explorar C# (ou qualquer linguagem de programação), você comete erros ao escrever código. O compilador localiza esses erros e os reporta a você. Quando a saída contiver mensagens de erro, examine atentamente o código de exemplo e o código na janela interativa para ver o que corrigir. Esse exercício ajuda você a aprender a estrutura do código C#.

Explorar a ordem das operações

A linguagem C# define a precedência de diferentes operações matemáticas com regras consistentes com as regras que você aprendeu em matemática. Multiplicação e divisão têm precedência sobre adição e subtração. Explore isso executando o seguinte código na janela interativa:

C#

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

A saída demonstra que a multiplicação é executada antes da adição.

Você pode forçar uma ordem de operação diferente adicionando parênteses em torno da operação ou das operações que deseja executar primeiro. Adicione as seguintes linhas à janela interativa:

C#

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

Explore mais combinando várias operações diferentes. Substitua a quarta linha no código anterior por algo assim:

```
C#

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

Você pode notar um comportamento interessante para números inteiros. A divisão de inteiros sempre produz um resultado inteiro, mesmo quando você espera que o resultado inclua uma parte decimal ou fracionária.

Experimente o seguinte código:

```
C#

int a = 7;
int b = 4;
int c = 3;
int d = (a + b) / c;
Console.WriteLine(d);
```

Explorar a precisão e os limites de inteiros

Esse último exemplo mostrou que a divisão de inteiros trunca o resultado. Você pode obter o **resto** usando o operador **resto**, o `%` caractere:

```
C#

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

O tipo inteiro de C# difere dos inteiros matemáticos de uma outra maneira: o `int` tipo tem limites mínimos e máximos. Experimente o seguinte código para ver esses limites:

```
C#

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

Se um cálculo produzir um valor que exceda esses limites, você terá uma condição de **subfluxo** ou **estouro** . A resposta parece quebrar de um limite para o outro. Para ver um exemplo, adicione estas duas linhas na janela interativa:

C#

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

Observe que a resposta está muito próxima do número inteiro mínimo (negativo). É o mesmo que `min + 2`. A operação de adição **transbordou** os valores permitidos para inteiros. A resposta é um número negativo grande, pois um estouro "envolve" do maior valor de inteiro possível para o menor.

Há outros tipos numéricos com limites e precisão diferentes que você usaria quando o `int` tipo não atende às suas necessidades. Vamos explorar esses tipos de números em seguida.

Trabalhar com o tipo Double

O `double` tipo numérico representa um número de ponto flutuante de precisão dupla. Esses termos podem ser novos para você. Um número de **ponto flutuante** é útil para representar números nonintegrais que podem ser grandes ou pequenos em magnitude. **Precisão dupla** é um termo relativo que descreve o número de dígitos binários usados para armazenar o valor. **Números de precisão dupla** têm o dobro do número de dígitos binários como **precisão única**. Em computadores modernos, é mais comum usar precisão dupla do que números de precisão única. **Números de precisão simples** são declarados usando a `float` palavra-chave. Vamos explorar. Execute o seguinte código e veja o resultado:

C#

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

Observe que a resposta inclui a parte decimal do quociente. Experimente uma expressão ligeiramente mais complicada com duplas. Você pode usar os seguintes valores ou substituir outros números:

C#

```
double a = 19;
double b = 23;
double c = 8;
double d = (a + b) / c;
Console.WriteLine(d);
```

O intervalo de um valor duplo é maior que valores inteiros. Experimente o seguinte código na janela interativa:

```
C#

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

Esses valores são impressos em notação científica. O número antes do `E` é o significando. O número após o `E` é o expoente, como uma potência de 10.

Assim como números decimais em matemática, as duplas em C# podem ter erros de arredondamento. Experimente este código:

```
C#

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

Você sabe que `0.3` é `3/10` e não exatamente o mesmo que `1/3`. Da mesma forma, `0.33` é `33/100`. Esse valor está mais próximo `1/3`, mas ainda não é exato. Não importa quantas casas decimais você adicione, um erro de arredondamento permanece.

Desafio

Experimente outros cálculos com números grandes, números pequenos, multiplicação e divisão usando o `double` tipo. Tente cálculos mais complicados.

Trabalhar com tipos decimais

Ainda há outro tipo: o tipo `decimal`. O `decimal` tipo tem um intervalo menor, mas maior precisão que `double`. Vamos dar uma olhada.

```
C#

decimal min = decimal.MinValue;
decimal max = decimal.MaxValue;
```

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

Observe que o intervalo é menor que o `double` tipo. Você pode ver a maior precisão com o tipo decimal ao tentar o seguinte código:

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

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

Observe que a matemática que usa o tipo decimal tem mais dígitos à direita do ponto decimal.

Os números com sufixo `M` são a maneira de indicar que uma constante deve utilizar o tipo `decimal`. Caso contrário, o compilador assume o tipo `double`.

> ⓘ **Observação**
>
> A letra `M` foi escolhida como a letra mais visualmente distinta entre as palavras-chave `double` e `decimal`.

Desafio

Escreva um código que calcula a área de um círculo cujo raio é de 2,50 centímetros. Lembre-se de que a área de um círculo é o raio quadrado multiplicado por PI. Uma dica: o .NET contém uma constante para PI, Math.PI que você pode usar para esse valor. Math.PI, como todas as constantes declaradas no `System.Math` namespace, é um `double` valor. Por esse motivo, você deve usar os valores de `double` em vez de `decimal` para esse desafio.

Você deve obter uma resposta entre 19 e 20.

Depois de experimentá-lo, abra o painel de detalhes para ver como você fez:
▼ Details
:::code language="csharp" interactive="try-dotnet-method" source="./snippets/NumbersInCsharp/Program.cs" id="Challenge":::

Experimente outras fórmulas se quiser.

Você concluiu o tutorial interativo "Números em C#". Você pode selecionar o link **Branches e Loops** para iniciar o próximo tutorial interativo ou visitar o site do .NET para baixar o SDK do

.NET, criar um projeto em seu computador e continuar codificando. A seção "Próximas etapas" traz você de volta a esses tutoriais.

Você pode saber mais sobre números em C# nos seguintes artigos:

- Tipos numéricos integrais
- Tipos numéricos de ponto flutuante
- Conversões numéricas internas

Criar tipos em C#

Artigo • 23/04/2025

Este tutorial ensina sobre como criar tipos em C#. Você escreve pequenas quantidades de código e, em seguida, compila e executa esse código. O tutorial contém uma série de lições que exploram diferentes tipos de tipos em C#. Essas lições ensinam os conceitos básicos da linguagem C#.

> **Dica**
>
> Quando um bloco de snippet de código inclui o botão "Executar", esse botão abre a janela interativa ou substitui o código existente na janela interativa. Quando o snippet não inclui um botão "Executar", você pode copiar o código e adicioná-lo à janela interativa atual.

Os tutoriais anteriores funcionavam com texto e números. Cadeias de caracteres e números são *tipos simples*: cada um armazena um único valor. À medida que seus programas aumentam, você precisa trabalhar com estruturas de dados mais sofisticadas. O C# fornece diferentes tipos de tipos que você pode definir quando precisa de estruturas de dados com mais campos, propriedades ou comportamento. Vamos começar a explorar esses tipos.

Tuplas

Tuplas são uma sequência ordenada de valores com um comprimento fixo. Cada elemento de uma tupla tem um tipo e um nome opcional. O código a seguir declara uma tupla que representa um ponto 2D. Selecione o botão "Executar" para colar o código a seguir na janela interativa e executá-lo.

C#

```
var pt = (X: 1, Y: 2);

var slope = (double)pt.Y / (double)pt.X;
Console.WriteLine($"A line from the origin to the point {pt} has a slope of
{slope}.");
```

> **Dica**
>
> Ao explorar C# (ou qualquer linguagem de programação), você comete erros ao escrever código. O **compilador** localiza esses erros e os relata para você. Quando a saída contiver mensagens de erro, examine atentamente o código de exemplo e o código na janela

interativa para ver o que corrigir. Esse exercício ajuda você a aprender a estrutura do código C#.

Você pode reatribuir qualquer membro de uma tupla. Adicione o código a seguir na janela interativa após o código existente. Pressione "Executar" novamente para ver os resultados.

```
C#

pt.X = pt.X + 5;
Console.WriteLine($"The point is now at {pt}.");
```

Você também pode criar uma nova tupla que é uma cópia modificada da tupla original usando a expressão `with`. Adicione o seguinte código após o código já na janela interativa e pressione "Executar" para ver os resultados:

```
C#

var pt2 = pt with { Y = 10 };
Console.WriteLine($"The point 'pt2' is at {pt2}.");
```

A tupla `pt2` contém o `X` valor de `pt` (6) e `pt2.Y` é 10.

Tuplas são tipos estruturais. Em outras palavras, os tipos de tuplas não têm nomes como `string` ou `int`. Um tipo de tupla é definido pelo número de membros, conhecido como *aridade* e pelos tipos desses membros. Os nomes dos membros são para conveniência. Você pode atribuir uma tupla a uma tupla com a mesma aridade e tipos, mesmo que os membros tenham nomes diferentes. Você pode adicionar o seguinte código após o código que você já escreveu na janela interativa e experimentá-lo:

```
C#

var subscript = (A: 0, B: 0);
subscript = pt;
Console.WriteLine(subscript);
```

A variável `subscript` tem dois membros, ambos inteiros. Ambos `subscript` e `pt` representam instâncias do mesmo tipo de tupla: uma tupla que contém dois `int` membros.

Tuplas são fáceis de criar: você declara vários membros entre parênteses. Todos os itens a seguir declaram diferentes tuplas de diferentes variedades e tipos de membros. Adicione o seguinte código para criar novos tipos de tuplas:

```
C#
```

```
var namedData = (Name: "Morning observation", Temp: 17, Wind: 4);
var person = (FirstName: "", LastName: "");
var order = (Product: "guitar picks", style: "triangle", quantity: 500, UnitPrice:
0.10m);
```

As tuplas são fáceis de criar, mas são limitadas em seus recursos. Os tipos de tuplas não têm nomes, portanto, você não pode dar significado ao conjunto de valores. Os tipos de tuplas não podem adicionar comportamento. O C# tem outros tipos de tipos que você pode criar quando seu tipo define o comportamento.

Criar tipos de registro

As tuplas são ótimas para aqueles momentos em que você deseja vários valores na mesma estrutura. Elas são leves e podem ser declaradas à medida que são usadas. À medida que seu programa se desenvolve, você pode perceber que usa o mesmo tipo de tupla em todo o seu código. Se o aplicativo funcionar no espaço do grafo 2D, as tuplas que representam pontos poderão ser comuns. Depois de descobrir isso, você poderá declarar um `record` tipo que armazene esses valores e forneça mais recursos. O exemplo de código a seguir usa um `Main` método para representar o ponto de entrada do programa. Dessa forma, você pode declarar um `record` tipo anterior ao ponto de entrada no código. Pressione o botão "Executar" no código a seguir para substituir o exemplo existente pelo código a seguir.

> ⚠ **Aviso**
>
> Não copie e cole. A janela interativa deve ser redefinida para executar o exemplo a seguir. Se você cometer um erro, a janela será interrompida e você precisará atualizar a página para continuar.

O código a seguir declara e usa um `record` tipo para representar um `Point` e, em seguida, usa essa `Point` estrutura no `Main` método:

C#

```
public record Point(int X, int Y);

public static void Main()
{
    Point pt = new Point(1, 1);
    var pt2 = pt with { Y = 10 };
    Console.WriteLine($"The two points are {pt} and {pt2}");
}
```

A declaração `record` é uma linha de código única para o tipo `Point` que armazena os valores `X` e `Y` em propriedades de leitura única. Você usa o nome `Point` sempre que usar esse tipo. Tipos nomeados corretamente, como `Point`, fornecem informações sobre como o tipo é usado. O `Main` método mostra como usar uma `with` expressão para criar um novo ponto que seja uma cópia modificada do ponto existente. A linha `pt2 = pt with { Y = 10 }` diz "`pt2` tem os mesmos valores que `pt`, exceto que `Y` é atribuído a 10." Você pode adicionar qualquer número de propriedades para alterar em uma única expressão `with`.

A declaração anterior `record` é uma única linha de código que termina em `;`, como todas as instruções C#. Você pode adicionar comportamento a um `record` tipo declarando *membros*. Um membro de registro pode ser uma função ou pode consistir em mais elementos de dados. Os membros de um tipo estão na declaração de tipo, entre `{` e `}` caracteres. Substitua a declaração de registro feita pelo seguinte código:

C#

```
public record Point(int X, int Y)
{
    public double Slope() => (double)Y / (double)X;
}
```

Em seguida, adicione o seguinte código ao `Main` método após a linha que contém a `with` expressão:

C#

```
double slope = pt.Slope();
Console.WriteLine($"The slope of {pt} is {slope}");
```

Você adicionou formalidade à *tupla* que representa um valor `X` e `Y`. Você o criou em `record` que definiu um tipo nomeado e incluiu um membro para calcular a inclinação Um `record` tipo é uma abreviação para um `record class`: um `class` tipo que inclui comportamento extra. Você pode modificar o `Point` tipo para torná-lo um `record struct` também:

C#

```
public record struct Point(int X, int Y)
{
    public double Slope() => (double) Y / (double) X;
}
```

Um `record struct` é um `struct` tipo que inclui o comportamento extra adicionado a todos os `record` tipos.

Tipos de estrutura, classe e interface

Todos os tipos nomeados em C# são tipos `class` ou tipos `struct`. A `class` é um *tipo de referência*. A `struct` é um *tipo de valor*. Variáveis de um tipo de valor armazenam o conteúdo da instância diretamente na memória. Em outras palavras, um `record struct Point` armazena dois inteiros: `X` e `Y`. As variáveis de um tipo de referência armazenam uma referência, ou ponteiro, para o armazenamento da instância. Em outras palavras, um `record class Point` armazena uma referência a um bloco de memória que contém os valores para `X` e `Y`.

Na prática, isso significa que os tipos de valor são copiados quando atribuídos, mas uma cópia de uma instância de classe é uma cópia da *referência*. Essa referência copiada refere-se à mesma instância de um ponto, com o mesmo armazenamento para `X` e `Y`.

O modificador `record`, instrui o compilador a gravar vários membros para você. Você pode saber mais no artigo sobre tipos de registro na seção de conceitos básicos.

Ao declarar um `record` tipo, você declara que seu tipo deve usar um conjunto padrão de comportamentos para comparações de igualdade, atribuição e cópia de instâncias desse tipo. Os registros são a melhor opção quando o armazenamento de dados relacionados é a principal responsabilidade do seu tipo. À medida que você adiciona mais comportamentos, considere usar `struct` ou `class` tipos sem o `record` modificador.

Você usa `struct` tipos para tipos de valor quando um comportamento mais sofisticado é necessário, mas a principal responsabilidade é armazenar valores. Você usa `class` tipos para usar idiomas orientados a objetos, como encapsulamento, herança e polimorfismo.

Você também pode definir `interface` tipos para declarar contratos comportamentais que diferentes tipos devem implementar. Os tipos `struct` e `class` podem implementar interfaces.

Normalmente, você usa todos esses tipos em programas e bibliotecas maiores. Depois de instalar o SDK do .NET, você poderá explorar esses tipos usando tutoriais sobre classes na seção de conceitos básicos.

Você concluiu o tutorial interativo "Criar tipos em C#". Você pode selecionar o link **Branches e Loops** para iniciar o próximo tutorial interativo ou visitar o site do .NET para baixar o SDK do .NET, criar um projeto em seu computador e continuar codificando. A seção "Próximas etapas" traz você de volta a esses tutoriais.

Você pode saber mais sobre tipos em C# nos seguintes artigos:

- Tipos em C#
- Registros
- Classes

Instruções e loops em C# `if` – tutorial de lógica condicional

Artigo • 24/03/2025

Este tutorial ensina como escrever código C# que examina variáveis e altera o caminho de execução com base nessas variáveis. Você escreve código C# e vê os resultados da compilação e execução dele. O tutorial contém uma série de lições que exploram as construções de ramificação e looping em C#. Essas lições ensinam os conceitos básicos da linguagem C#.

> **Dica**
>
> Quando um bloco de snippet de código inclui o botão "Executar", esse botão abre a janela interativa ou substitui o código existente na janela interativa. Quando o snippet não inclui um botão "Executar", você pode copiar o código e adicioná-lo à janela interativa atual.

Execute o código a seguir na janela interativa. Selecione **Executar**:

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

Modifique a declaração de `b` para que a soma seja menor que 10:

```C#
int b = 3;
```

Selecione o botão **Executar** novamente. Como a resposta é menor que 10, nada é impresso. A condição que você está testando é falsa. Você não tem nenhum código para executar, pois escreveu apenas uma das ramificações possíveis de uma instrução `if`: a ramificação verdadeira.

> **Dica**

Ao explorar C# (ou qualquer linguagem de programação), você comete erros ao escrever código. O compilador localiza esses erros e os reporta a você. Quando a saída contiver mensagens de erro, examine atentamente o código de exemplo e o código na janela interativa para ver o que corrigir. Esse exercício ajuda você a aprender a estrutura do código C#.

Este primeiro exemplo mostra o poder dos tipos `if` e boolianos. Um *booliano* é uma variável que pode ter um dos dois valores: `true` ou `false`. O C# define um tipo especial, `bool` para variáveis boolianas. A instrução `if` verifica o valor de um `bool`. Quando o valor é `true`, a instrução após o `if` é executada. Caso contrário, ela é ignorada.

Esse processo de verificação de condições e execução de instruções com base nessas condições é eficiente. Vamos explorar mais.

Fazer com que "if" e "else" trabalhem juntos

Para executar código diferente nas ramificações verdadeiras e falsas, crie uma ramificação `else` que seja executada quando a condição for falsa. Experimente o seguinte código:

C#

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

A instrução após a palavra-chave `else` é executada somente quando a condição que está sendo testada é `false`. Combinar `if` e `else` com condições boolianas fornece toda a energia necessária.

> **Importante**
>
> A indentação sob as declarações `if` e `else` é para leitores humanos. A linguagem C# não trata a indentação ou o espaço em branco como significativo. A instrução após a palavra-chave `if` ou `else` é executada com base na condição. Todos os exemplos neste tutorial seguem uma prática comum de indentar linhas com base no fluxo de controle das instruções.

Já que a indentação não é significativa, é necessário usar `{` e `}` para indicar quando você deseja que mais de uma instrução faça parte do bloco que é executado condicionalmente. Os programadores em C# normalmente usam essas chaves em todas as cláusulas `if` e `else`. O exemplo a seguir é o mesmo que você criou. Experimente.

C#

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

Dica

No restante deste tutorial, todos os exemplos de código incluem as chaves, seguindo as práticas aceitas.

Você pode testar condições mais complicadas:

C#

```
int a = 5;
int b = 3;
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");
}
```

O símbolo `==` testa a *igualdade*. Usar `==` distingue o teste de igualdade de atribuição, que você viu em `a = 5`.

O `&&` representa "e". Isso significa que as duas condições devem ser verdadeiras para executar a instrução no branch verdadeiro. Estes exemplos também mostram que você

pode ter várias instruções em cada branch condicional, desde que você coloque-as entre `{` e `}`.

Você também pode usar `||` para representar "ou":

```
C#

if ((a + b + c > 10) || (a == b))
```

Modifique os valores de `a`, `b` e `c` e alterne entre `&&` e `||` para explorar. Você obtém mais compreensão de como os operadores `&&` e `||` funcionam.

Use loops para repetir operações

Outro conceito importante para criar programas maiores são os **loops**. Você usa loops para repetir instruções que deseja executar mais de uma vez. Experimente este código na janela interativa:

```
C#

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

A instrução `while` verifica uma condição e executa a instrução após a `while`. Ela repete a verificação da condição e a execução dessas instruções até que a condição seja false.

Há um outro operador novo neste exemplo. O `++` após a variável `counter` é o operador **increment**. Ele adiciona 1 ao valor do contador e armazena esse valor na variável de contador.

> ⓘ **Importante**
>
> Verifique se a condição de loop `while` muda para false à medida que você executa o código. Caso contrário, você criará um loop infinito em que seu programa nunca terminará. Não vamos demonstrar isso, pois o mecanismo que executa seu código atinge o tempo limite e você não vê nenhuma saída do programa.

O loop `while` testa a condição antes de executar o código após o `while`. O loop `do`... `while` executa o código primeiro e, em seguida, verifica a condição. Tem esta aparência:

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

Esse loop `do` e o loop de `while` anterior funcionam da mesma forma.

Vamos passar para a última instrução de loop.

Trabalhar com o loop for

Outra instrução de loop comum que você vê no código C# é o loop `for`. Experimente este código na janela interativa:

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

O loop `for` anterior faz o mesmo trabalho que o loop de `while` e o loop `do` que você já usou. A instrução `for` tem três partes que controlam como ela funciona:

- A primeira parte é a **do inicializador**: `int counter = 0;` declara que `counter` é a variável de loop e define seu valor inicial como `0`.
- A parte intermediária é a **de condição**: `counter < 10` declara que esse loop `for` continua a ser executado desde que o valor do contador seja menor que 10.
- A parte final é a **do iterador**: `counter++` especifica como modificar a variável de loop depois de executar o bloco após a instrução `for`. Aqui, ela especifica que `counter` é incrementado com 1 sempre que o bloco for executado.

Experimente essas condições por conta própria. Experimente cada uma das seguintes alterações:

- Altere o inicializador para iniciar com um valor diferente.
- Altere a condição para parar em um valor diferente.

Quando terminar, vamos continuar a escrever algum código por conta própria para usar o que você aprendeu.

Há uma outra instrução de loop que não está abordada neste tutorial: a instrução `foreach`. A instrução `foreach` repete sua instrução para cada item em uma sequência de itens. Com mais frequência, ele é usado com *coleções*. Ele é abordado no próximo tutorial.

Loops aninhados criados

Um loop `while`, `do` ou `for` pode ser aninhado dentro de outro loop para criar uma matriz usando a combinação de cada item no loop externo com cada item no loop interno. Vamos fazer isso para criar um conjunto de pares alfanuméricos para representar linhas e colunas.

Um loop `for` pode gerar as linhas:

C#

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

Outro loop pode gerar as colunas:

C#

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

Você pode aninhar um loop dentro do outro para formar pares:

C#

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

Você pode perceber que o loop externo é incrementado uma vez após cada execução completa do loop interno. Inverta o aninhamento de linha e da coluna e confira as alterações.

Combinar branches e loops

Agora que você viu a instrução `if` e as construções de looping na linguagem C#, veja se você pode escrever código C# para encontrar a soma de todos os inteiros de 1 a 20 que são divisível por 3. Aqui estão algumas dicas:

- O operador `%` retorna o restante de uma operação de divisão.
- A instrução `if` especifica a condição para determinar se um número deve fazer parte da soma.
- O loop `for` pode ajudá-lo a repetir uma série de etapas para todos os números de 1 a 20.

Experimente você mesmo. Em seguida, verifique como você fez. Como dica, você deve obter 63 como resposta.

Você pensou em algo assim?
▼ Details
:::code language="csharp" interactive="try-dotnet-method" source="./snippets/BranchesAndLoops/Program.cs" id="Challenge":::

Você concluiu o tutorial interativo "branches e loops". Você pode selecionar o link **coleções de listas** para iniciar o próximo tutorial interativo ou visitar o site do .NET para baixar o SDK do .NET, criar um projeto no computador e continuar programando. A seção "Próximas etapas" traz você de volta a esses tutoriais.

Você pode saber mais sobre esses conceitos nestes artigos:

- Instruções de seleção
- Instruções de iteração

Saiba como gerenciar coletas de dados usando a Lista<T> em C#

Artigo • 22/05/2025

Este tutorial introdutório fornece uma introdução à linguagem C# e aos conceitos básicos da classe.

Este tutorial ensina o C# interativamente, usando seu navegador para escrever código C# e ver os resultados da compilação e execução do código. Ele contém uma série de lições que criam, modificam e exploram coleções e matrizes. Você trabalha principalmente com a List<T> classe.

Um exemplo de lista básica

> **Dica**
>
> Quando um bloco de snippet de código inclui o botão "Executar", esse botão abre a janela interativa ou substitui o código existente na janela interativa. Quando o snippet não inclui um botão "Executar", você pode copiar o código e adicioná-lo à janela interativa atual.

Execute o código a seguir na janela interativa. Substitua `<name>` pelo nome e selecione **Executar**:

C#

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

Você criou uma lista de strings, adicionou três nomes a essa lista e imprimiu os nomes em maiúsculas. Você está usando conceitos que aprendeu em tutoriais anteriores para percorrer a lista.

O código para exibir nomes usa o recurso de interpolação de cadeia de caracteres. Quando você precede um `string` com o `$` caractere, é possível inserir o código C# na declaração de cadeia de caracteres. A string real substitui o código C# pelo valor que ele gera. Neste exemplo, ele substitui o `{name.ToUpper()}` por cada nome, convertido em letras maiúsculas, porque você chamou o método String.ToUpper.

Vamos continuar explorando.

Modificar o conteúdo da lista

A coleção criada usa o tipo List<T>. Esse tipo armazena sequências de elementos. Especifique o tipo dos elementos entre os colchetes angulares.

Um aspecto importante desse List<T> tipo é que ele pode crescer ou reduzir, permitindo que você adicione ou remova elementos. Você pode ver os resultados modificando o conteúdo após exibi-lo. Adicione o código a seguir após o código que você já escreveu (o loop que imprime o conteúdo):

C#

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

Você adicionou mais dois nomes ao final da lista. Você removeu um também. A saída desse bloco de código mostra o conteúdo inicial e, em seguida, imprime uma linha em branco e o novo conteúdo.

O List<T> permite que você faça referência a itens individuais por **índice** também. Você acessa itens usando os tokens `[` e `]`. Adicione o seguinte código após o que você já escreveu e experimente:

C#

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

Você não tem permissão para acessar após o final da lista. Você pode verificar por quanto tempo a lista está usando a Count propriedade. Adicione o seguinte código:

C#

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

Selecione **Executar** novamente para ver os resultados. Em C#, os índices começam em 0, portanto, o maior índice válido é um a menos do que o número de itens na lista.

Pesquisar e classificar listas

Nossos exemplos usam listas relativamente pequenas, mas seus aplicativos geralmente podem criar listas com muitos outros elementos, às vezes numerando em milhares. Para encontrar elementos nessas coleções maiores, você precisa procurar diferentes itens na lista. O IndexOf método pesquisa um item e retorna o índice do item. Se o item não estiver na lista, `IndexOf` retornará `-1`. Tente ver como funciona. Adicione o seguinte código após o que você escreveu até agora:

C#

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

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

Talvez você não saiba se um item está na lista, portanto, você sempre deve verificar o índice retornado por IndexOf. Se for `-1`, o item não foi encontrado.

Os itens em sua lista também podem ser classificados. O Sort método classifica todos os itens da lista em sua ordem normal (em ordem alfabética para cadeias de caracteres). Adicione este código e execute novamente:

C#

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

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

Listas de outros tipos

Você vem utilizando o tipo `string` em listas até agora. Vamos criar um List<T> com um tipo diferente. Vamos criar um conjunto de números. Exclua o código que você escreveu até agora e substitua-o pelo seguinte código:

C#

```
List<int> fibonacciNumbers = [1, 1];
```

Isso cria uma lista de inteiros e define os dois primeiros inteiros como o valor 1. A *Sequência de Fibonacci*, uma sequência de números, começa com dois 1. Cada próximo número fibonacci é encontrado tomando a soma dos dois números anteriores. Adicione este código:

C#

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

fibonacciNumbers.Add(previous + previous2);

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

Pressione **Executar** para ver os resultados;

Desafio

Veja se você pode reunir alguns dos conceitos desta e das lições anteriores. Expanda o que você criou até agora com números de Fibonacci. Tente escrever o código para gerar os primeiros 20 números na sequência. (Como dica, o 20º número de Fibonacci é 6765.)

Você pensou em algo assim?
▼ Details
:::code language="csharp" interactive="try-dotnet-method" source="./snippets/ListCollection/Program.cs" id="Answer":::

Com cada iteração do loop, você está tomando os dois últimos inteiros na lista, resumindo-os e adicionando esse valor à lista. O loop é repetido até que você tenha adicionado 20 itens à lista.

Você concluiu o tutorial interativo da lista, que é a introdução final ao tutorial interativo de C#. Você pode visitar o site do .NET para baixar o SDK do .NET, criar um projeto em seu computador e continuar codificando. A seção "Próximas etapas" traz você de volta a esses tutoriais. Ou você pode continuar com o tutorial Explore programação orientada a objetos com classes e objetos.

Saiba mais sobre as coleções do .NET nos seguintes artigos:

- Selecionando um tipo de coleção
- Tipos de coleções de uso comum
- Quando usar coleções genéricas

Comparar dados com padrões

Artigo • 03/05/2025

Este tutorial ensina como usar a correspondência de padrões para inspecionar dados em C#. Você escreve pequenas quantidades de código e, em seguida, compila e executa esse código. O tutorial contém uma série de lições que exploram diferentes tipos de tipos em C#. Essas lições ensinam os conceitos básicos da linguagem C#.

> **Dica**
>
> Quando um bloco de snippet de código inclui o botão "Executar", esse botão abre a janela interativa ou substitui o código existente na janela interativa. Quando o snippet não inclui um botão "Executar", você pode copiar o código e adicioná-lo à janela interativa atual.

Os tutoriais anteriores demonstraram tipos embutidos e tipos que você define como tuplas ou tipos de registros. Instâncias desses tipos podem ser verificadas em relação a um *padrão*. Se uma instância corresponde a um padrão determina as ações que seu programa executa. Vamos começar a explorar como você pode usar padrões.

Igualar a um valor

Todos os exemplos neste tutorial usam a entrada de texto que representa uma série de transações bancárias como entrada de CSV (valores separados por vírgulas). Em cada uma das amostras, você pode corresponder o registro com um padrão usando uma expressão `is` ou uma expressão `switch`. Este primeiro exemplo divide cada linha no caractere `,` e, em seguida, *verifica* o primeiro campo da string contra o valor "DEPOSIT" ou "WITHDRAWAL" usando uma expressão `is`. Quando corresponde, o valor da transação é adicionado ou deduzido do saldo da conta atual. Para vê-lo funcionar, pressione o botão "Executar":

C#

```
string bankRecords = """
    DEPOSIT,   10000, Initial balance
    DEPOSIT,     500, regular deposit
    WITHDRAWAL, 1000, rent
    DEPOSIT,    2000, freelance payment
    WITHDRAWAL,  300, groceries
    DEPOSIT,     700, gift from friend
    WITHDRAWAL,  150, utility bill
    DEPOSIT,    1200, tax refund
    WITHDRAWAL,  500, car maintenance
    DEPOSIT,     400, cashback reward
```

```
    WITHDRAWAL,  250, dining out
    DEPOSIT,    3000, bonus payment
    WITHDRAWAL,  800, loan repayment
    DEPOSIT,     600, stock dividends
    WITHDRAWAL,  100, subscription fee
    DEPOSIT,    1500, side hustle income
    WITHDRAWAL,  200, fuel expenses
    DEPOSIT,     900, refund from store
    WITHDRAWAL,  350, shopping
    DEPOSIT,    2500, project milestone payment
    WITHDRAWAL,  400, entertainment
    """;

double currentBalance = 0.0;
var reader = new StringReader(bankRecords);

string? line;
while ((line = reader.ReadLine()) is not null)
{
    if (string.IsNullOrWhiteSpace(line)) continue;
    // Split the line based on comma delimiter and trim each part
    string[] parts = line.Split(',');

    string? transactionType = parts[0]?.Trim();
    if (double.TryParse(parts[1].Trim(), out double amount))
    {
        // Update the balance based on transaction type
        if (transactionType?.ToUpper() is "DEPOSIT")
            currentBalance += amount;
        else if (transactionType?.ToUpper() is "WITHDRAWAL")
            currentBalance -= amount;

        Console.WriteLine($"{line.Trim()} => Parsed Amount: {amount}, New Balance:
{currentBalance}");
    }
}
```

Verifique a saída. Você pode ver que cada linha é processada comparando o valor do texto no primeiro campo. O exemplo anterior pode ser construído da mesma forma usando o `==` operador para testar se dois `string` valores são iguais. Comparar uma variável com uma constante é um bloco de construção básico para correspondência de padrões. Vamos explorar mais dos blocos de construção que fazem parte da correspondência de padrões.

Correspondências de Enum

Outro uso comum para correspondência de padrões é corresponder aos valores de um tipo `enum`. Este próximo exemplo processa os registros de entrada para criar uma *tupla* em que a primeira entrada é um `enum` valor que indica um depósito ou um saque. O segundo valor é o valor da transação. Para vê-lo funcionar, pressione o botão "Executar":

> ⚠ **Aviso**
>
> Não copie e cole. A janela interativa deve ser redefinida para executar os exemplos a seguir. Se você cometer um erro, a janela será interrompida e você precisará atualizar a página para continuar.

C#

```
public static class ExampleProgram
{
    const string bankRecords = """
    DEPOSIT,    10000, Initial balance
    DEPOSIT,      500, regular deposit
    WITHDRAWAL, 1000, rent
    DEPOSIT,     2000, freelance payment
    WITHDRAWAL,   300, groceries
    DEPOSIT,      700, gift from friend
    WITHDRAWAL,   150, utility bill
    DEPOSIT,     1200, tax refund
    WITHDRAWAL,   500, car maintenance
    DEPOSIT,      400, cashback reward
    WITHDRAWAL,   250, dining out
    DEPOSIT,     3000, bonus payment
    WITHDRAWAL,   800, loan repayment
    DEPOSIT,      600, stock dividends
    WITHDRAWAL,   100, subscription fee
    DEPOSIT,     1500, side hustle income
    WITHDRAWAL,   200, fuel expenses
    DEPOSIT,      900, refund from store
    WITHDRAWAL,   350, shopping
    DEPOSIT,     2500, project milestone payment
    WITHDRAWAL,   400, entertainment
    """;

    public static void Main()
    {
        double currentBalance = 0.0;

        foreach (var transaction in TransactionRecords(bankRecords))
        {
            if (transaction.type == TransactionType.Deposit)
                currentBalance += transaction.amount;
            else if (transaction.type == TransactionType.Withdrawal)
                currentBalance -= transaction.amount;
            Console.WriteLine($"{transaction.type} => Parsed Amount:
{transaction.amount}, New Balance: {currentBalance}");
        }
    }

    static IEnumerable<(TransactionType type, double amount)>
TransactionRecords(string inputText)
```

```
    {
        var reader = new StringReader(inputText);
        string? line;
        while ((line = reader.ReadLine()) is not null)
        {
            string[] parts = line.Split(',');

            string? transactionType = parts[0]?.Trim();
            if (double.TryParse(parts[1].Trim(), out double amount))
            {
                // Update the balance based on transaction type
                if (transactionType?.ToUpper() is "DEPOSIT")
                    yield return (TransactionType.Deposit, amount);
                else if (transactionType?.ToUpper() is "WITHDRAWAL")
                    yield return (TransactionType.Withdrawal, amount);
            }
            yield return (TransactionType.Invalid, 0.0);
        }
    }
}

public enum TransactionType
{
    Deposit,
    Withdrawal,
    Invalid
}
```

O exemplo anterior também usa uma instrução `if` para verificar o valor de uma expressão `enum` . Outra forma de correspondência de padrões usa uma `switch` expressão. Vamos explorar essa sintaxe e como usá-la.

Resultados exaustivos com `switch`

Uma série de `if` declarações pode avaliar uma sequência de condições. No entanto, o compilador não pode dizer se uma série de `if` instruções são *exaustivas* ou se as condições posteriores `if` são *subsumidas* por condições anteriores. A `switch` expressão garante que ambas as características sejam atendidas, o que resulta em menos bugs em seus aplicativos. Vamos tentar e experimentar. Copie o código a seguir. Substitua as duas `if` declarações na janela interativa pela expressão `switch` que você copiou. Depois de modificar o código, pressione o botão "Executar" na parte superior da janela interativa para executar o novo exemplo.

C#

```
currentBalance += transaction switch
{
    (TransactionType.Deposit, var amount) => amount,
```

```
    (TransactionType.Withdrawal, var amount) => -amount,
    _ => 0.0,
};
```

Ao executar o código, você verá que ele funciona da mesma forma. Para demonstrar *a subsumpição*, reordene os braços do comutador, conforme mostrado no trecho de código a seguir.

```C#
currentBalance += transaction switch
{
    (TransactionType.Deposit, var amount) => amount,
    _ => 0.0,
    (TransactionType.Withdrawal, var amount) => -amount,
};
```

Depois de reordenar os braços do comutador, pressione o botão "Executar". O compilador gera um erro porque o ramo com `_` corresponde a cada valor. Como resultado, aquele braço final com `TransactionType.Withdrawal` nunca é executado. O compilador informa que algo está errado em seu código.

O compilador emitirá um aviso se a expressão testada em uma `switch` expressão puder conter valores que não correspondem a nenhum braço de comutador. Se alguns valores não corresponderem a qualquer condição, a `switch` expressão não será *exaustiva*. O compilador também emitirá um aviso se alguns valores da entrada não corresponderem a nenhum dos braços do comutador. Por exemplo, se você remover a linha com `_ => 0.0,`, quaisquer valores inválidos não corresponderão. Em tempo de execução, isso falharia. Depois de instalar o SDK do .NET e criar programas em seu ambiente, você poderá testar esse comportamento. A experiência online não exibe avisos na janela de saída.

Padrões de tipo

Para concluir este tutorial, vamos explorar mais um bloco de construção para correspondência de padrões: o *padrão de tipo*. Um *padrão de tipo* testa uma expressão em tempo de execução para ver se é o tipo especificado. Você pode usar um teste de tipo com uma `is` expressão ou uma `switch` expressão. Vamos modificar o exemplo atual de duas maneiras. Primeiro, em vez de uma tupla, vamos criar `Deposit` e `Withdrawal` tipos de registro que representam as transações. Adicione as seguintes declarações na parte inferior da janela interativa:

```C#
```

```
public record Deposit(double Amount, string description);
public record Withdrawal(double Amount, string description);
```

Em seguida, adicione esse método após o `Main` método para analisar o texto e retornar uma série de registros:

C#

```
public static IEnumerable<object?> TransactionRecordType(string inputText)
{
    var reader = new StringReader(inputText);
    string? line;
    while ((line = reader.ReadLine()) is not null)
    {
        string[] parts = line.Split(',');

        string? transactionType = parts[0]?.Trim();
        if (double.TryParse(parts[1].Trim(), out double amount))
        {
            // Update the balance based on transaction type
            if (transactionType?.ToUpper() is "DEPOSIT")
                yield return new Deposit(amount, parts[2]);
            else if (transactionType?.ToUpper() is "WITHDRAWAL")
                yield return new Withdrawal(amount, parts[2]);
        }
        yield return default;
    }
}
```

Por fim, substitua o `foreach` loop no `Main` método pelo seguinte código:

C#

```
foreach (var transaction in TransactionRecordType(bankRecords))
{
    currentBalance += transaction switch
    {
        Deposit d => d.Amount,
        Withdrawal w => -w.Amount,
        _ => 0.0,
    };
    Console.WriteLine($" {transaction} => New Balance: {currentBalance}");
}
```

Em seguida, pressione o botão "Executar" para ver os resultados. Esta versão final testa a entrada em relação a um *tipo*.

A correspondência de padrões fornece um vocabulário para comparar uma expressão com características. Os padrões podem incluir o tipo da expressão, valores de tipos, valores de propriedade e combinações deles. Comparar expressões com um padrão pode ser mais claro do que várias `if` comparações. Você explorou alguns dos padrões que pode usar para corresponder a expressões. Há muitas outras maneiras de usar a correspondência de padrões em seus aplicativos. Primeiro, visite o site do .NET para baixar o SDK do .NET, criar um projeto em seu computador e continuar codificando. Ao explorar, você pode saber mais sobre a correspondência de padrões em C# nos seguintes artigos:

- Padrões correspondentes em C#
- Explorar o tutorial de correspondência de padrões
- Cenário de correspondência de padrões

Estratégia de C# anotada

19/06/2025

Continuaremos evoluindo em C# para atender às necessidades em mudança dos desenvolvedores e continuaremos sendo uma linguagem de programação de última geração. Inovaremos de forma ansiosa e ampla em colaboração com as equipes responsáveis por bibliotecas do .NET, ferramentas de desenvolvedor e suporte à carga de trabalho, ao mesmo tempo em que tomaremos cuidado para permanecer dentro do espírito da linguagem. Reconhecendo a diversidade de domínios em que o C# está sendo usado, preferiremos aprimoramentos de linguagem e desempenho que beneficiem todos ou a maioria dos desenvolvedores e mantenham um alto compromisso com a compatibilidade com versões anteriores. Continuaremos capacitando o ecossistema mais amplo do .NET e aumentando seu papel no futuro do C#, mantendo a administração das decisões de design.

Como a estratégia orienta C#

A estratégia do C# orienta nossas decisões sobre a evolução do C# e essas anotações fornecem insights sobre como pensamos sobre as instruções-chave.

> "vamos inovar de forma ansiosa e ampla"

A comunidade C# continua a crescer e a linguagem C# continua evoluindo para atender às necessidades e expectativas da comunidade. Nos inspiramos em muitas fontes para selecionar recursos que beneficiam um grande segmento de desenvolvedores C# e que fornecem melhorias consistentes na produtividade, legibilidade e desempenho.

> "ter cuidado para permanecer dentro do espírito da linguagem"

Avaliamos novas ideias no espírito e na história da linguagem C#. Priorizamos inovações que fazem sentido para a maioria dos desenvolvedores em C# existentes.

> "melhorias que beneficiam todos ou a maioria dos desenvolvedores"

Os desenvolvedores usam C# em todas as cargas de trabalho do .NET. Os desenvolvedores criam front-ends e back-ends da Web, aplicativos nativos de nuvem e aplicativos da área de trabalho com C#. O C# habilita aplicativos multiplataforma. Nos concentramos em novos recursos que têm o maior impacto, seja diretamente, seja permitindo melhorias em bibliotecas comuns. O desenvolvimento de recursos de linguagem inclui integração com nossas ferramentas de desenvolvedor e recursos de aprendizagem.

> "alto compromisso com a compatibilidade com versões anteriores"

Respeitamos que haja uma grande quantidade de código C# em uso hoje. Qualquer possível alteração interruptiva é cuidadosamente considerada em relação à escala e ao impacto da interrupção na comunidade C#.

> "mantendo a administração"

O design da linguagem C# ocorre ao ar livre com a participação da comunidade. Qualquer pessoa pode propor novos recursos em C# em nossos repositórios do GitHub. A Equipe de Design de Linguagem toma as decisões finais depois de considerar o feedback da comunidade.

Roteiro para desenvolvedores Java aprendendo C#

Artigo • 30/04/2025

O C# e o Java têm muitas semelhanças. Ao aprender C#, você pode aplicar grande parte do seu conhecimento prévio de programação em Java:

1. ***Sintaxe semelhante***: Java e C# estão na família C de idiomas. Essa similaridade significa que você já pode ler e entender C#. Há algumas diferenças, mas a maior parte da sintaxe é igual a Java e C. As chaves e os ponto-e-vírgula são familiares. As instruções de controle, como `if`, `else` e `switch`, são as mesmas. As instruções de loop `for`, `while` e `do`...`while` são as mesmas. As mesmas palavras-chave para `class` e `interface` estão nos dois idiomas. Os modificadores de acesso de `public` até `private` são os mesmos. Mesmo muitos dos tipos internos usam as mesmas palavras-chave: `int`, `string` e `double`.
2. ***Paradigma orientado a objeto***: Java e C# são linguagens orientadas a objetos. Os conceitos de polimorfismo, abstração e encapsulamento se aplicam em ambas as linguagens. Ambos adicionaram novos constructos, mas os principais recursos ainda são relevantes.
3. ***Fortemente tipado***: Java e C# são linguagens fortemente tipadas. Você declara o tipo de dados de variáveis, explicitamente ou implicitamente. O compilador impõe a segurança do tipo. O compilador captura erros relacionados ao tipo em seu código antes de executar o código.
4. ***Multiplataforma***: Java e C# são multiplataforma. Você pode executar suas ferramentas de desenvolvimento em sua plataforma favorita. Seu aplicativo pode ser executado em várias plataformas. Sua plataforma de desenvolvimento não precisa necessariamente corresponder à plataforma de destino.
5. ***Tratamento de exceção***: Java e C# geram exceções para indicar erros. Ambos usam blocos `try` - `catch` - `finally` para lidar com exceções. As classes exception têm nomes e hierarquias de herança semelhantes. Uma diferença é que o C# não tem o conceito de *exceções verificadas*. Qualquer método pode (em teoria) gerar qualquer exceção.
6. ***Bibliotecas padrão***: o runtime do .NET e a JSL (Biblioteca Padrão java) têm suporte para tarefas comuns. Ambos têm ecossistemas extensos para outros pacotes de software livre. No C#, o gerenciador de pacotes é NuGet. É análogo ao Maven.
7. ***Coleta de Lixo***: ambos os idiomas empregam o gerenciamento automático de memória por meio da coleta de lixo. O runtime recupera a memória de objetos que não são referenciados. Uma diferença é que o C# permite que você crie tipos de valor, como tipos `struct`.

Você pode trabalhar produtivamente em C# quase imediatamente devido às semelhanças. À medida que você progride, você deve aprender recursos e idiomas em C# que não estão

disponíveis em Java:

1. ***Correspondência de padrões***: a correspondência de padrões permite instruções condicionais concisas e expressões com base na forma de estruturas de dados complexas. A is instrução verifica se uma variável "é" algum padrão. A expressão baseada em switch padrão fornece uma sintaxe avançada para inspecionar uma variável e tomar decisões com base em suas características.
2. ***Interpolação de cadeia de caracteres*** e ***literais de cadeia de caracteres brutos***: a interpolação de cadeia de caracteres permite inserir expressões avaliadas em uma cadeia de caracteres, em vez de usar identificadores posicionais. Literais de cadeia de caracteres brutos fornecem uma maneira de minimizar sequências de escape no texto.
3. ***Tipos anuláveis e não anuláveis***: O C# dá suporte a *tipos de valor anuláveis* e *tipos de referência anuláveis* acrescentando o `?` sufixo a um tipo. Para tipos que permitem valor nulo, o compilador avisa se você não verificar `null` antes de desreferenciar a expressão. Para tipos não anuláveis, o compilador avisa se você pode estar atribuindo um valor `null` a essa variável. Tipos de referência não anuláveis minimizam erros de programação que lançam um System.NullReferenceException.
4. ***Extensões***: em C#, você pode criar membros que *estendem* uma classe ou interface. As extensões fornecem um novo comportamento para um tipo de uma biblioteca ou todos os tipos que implementam uma determinada interface.
5. ***LINQ***: A LINQ (consulta integrada à linguagem) fornece uma sintaxe comum para consultar e transformar dados, independentemente de seu armazenamento.
6. ***Funções locais***: em C#, você pode aninhar funções dentro de métodos ou outras funções locais. As funções locais fornecem mais uma camada de encapsulamento.

Há outros recursos em C# que não existem em Java. Recursos como async e await modelam operações assíncronas na sintaxe sequencial. A instrução using libera automaticamente recursos não relacionados à memória.

Há também alguns recursos semelhantes entre C# e Java que têm diferenças sutis, mas importantes:

1. ***Propriedades*** e ***indexadores***: ambas as propriedades e indexadores (tratando uma classe como uma matriz ou dicionário) têm suporte de idioma. Em Java, eles estão nomeando convenções para métodos que começam com `get` e `set`.
2. ***Registros***: em C#, os registros podem ser tipos de `class` (referência) ou `struct` (valor). Os registros C# podem ser imutáveis, mas não precisam necessariamente ser imutáveis.
3. ***As tuplas*** têm sintaxe diferente em C# e Java.
4. ***Os atributos*** são semelhantes às anotações java.

Por fim, há recursos de linguagem Java que não estão disponíveis em C#:

1. ***Exceções verificadas***: em C#, qualquer método poderia teoricamente gerar qualquer exceção.
2. ***Covariância de matriz marcada***: em C#, as matrizes não são covariantes com segurança. Você deve usar as classes e interfaces de coleção genéricas se precisar de estruturas covariantes.

No geral, aprender C# para um desenvolvedor experiente em Java deve ser suave. C# tem expressões familiares suficientes para você ser produtivo à medida que aprende os novos idiomas.

Roteiro para desenvolvedores JavaScript e TypeScript aprendendo C#

19/06/2025

C#, TypeScript e JavaScript são todos membros da família C de idiomas. As semelhanças entre os idiomas ajudam você a se tornar produtivo rapidamente em C#.

1. ***Sintaxe semelhante***: JavaScript, TypeScript e C# estão na família C de idiomas. Essa similaridade significa que você já pode ler e entender C#. Há algumas diferenças, mas a maior parte da sintaxe é igual a JavaScript e C. As chaves e os pontos-e-vírgulas são familiares. As instruções de controle, como `if`, `else` e `switch`, são as mesmas. As instruções de loop `for`, `while` e `do`...`while` são as mesmas. As mesmas palavras-chave para `class` e `interface` estão em C# e TypeScript. Os modificadores de acesso em TypeScript e C#, de `public` até `private`, são os mesmos.
2. ***O token `=>` token***: todas as linguagens de programação dão suporte a definições de função leves. Em C#, *elas são conhecidas como expressões lambda*, em JavaScript, normalmente são chamadas *de funções de seta*.
3. ***Hierarquias de função***: todas as três linguagens dão suporte a funções locais, que são funções definidas em outras funções.
4. ***Assíncrono/Espera***: todas as três linguagens compartilham as mesmas `async` `await` palavras-chave para programação assíncrona.
5. ***Coleta de lixo***: todos os três idiomas dependem de um coletor de lixo para o gerenciamento automático de memória.
6. ***Modelo de evento***: a sintaxe do event C#é semelhante ao modelo do JavaScript para eventos do DOM (modelo de objeto de documento).
7. ***Gerenciador de pacotes***: o NuGet é o gerenciador de pacotes mais comum para C# e .NET, semelhante ao npm para aplicativos JavaScript. As bibliotecas C# são entregues em assemblies.

Ao aprender C#, você aprenderá conceitos que não fazem parte do JavaScript. Alguns desses conceitos podem ser familiares se você usar o TypeScript:

1. ***Sistema de tipagem do C#***: o C# é uma linguagem fortemente tipada. Cada variável tem um tipo e esse tipo não pode ser alterado. Você define os tipos `class` ou `struct`. Você pode definir interface definições que definem o comportamento implementado por outros tipos. O TypeScript inclui muitos desses conceitos, mas como o TypeScript é criado em JavaScript, o sistema de tipos não é tão estrito.
2. ***Correspondência de padrões***: a correspondência de padrões permite instruções condicionais concisas e expressões com base na forma de estruturas de dados complexas. A is expressão verifica se uma variável "é" algum padrão. A expressão baseada em switch

padrão fornece uma sintaxe avançada para inspecionar uma variável e tomar decisões com base em suas características.

3. ***Interpolação de cadeia de caracteres*** e ***literais de cadeia de caracteres brutos***: a interpolação de cadeia de caracteres permite inserir expressões avaliadas em uma cadeia de caracteres, em vez de usar identificadores posicionais. Literais de cadeia de caracteres brutos fornecem uma maneira de minimizar sequências de escape no texto.
4. ***Tipos anuláveis e não anuláveis***: O C# dá suporte a *tipos de valor anuláveis* e *tipos de referência anuláveis* acrescentando o `?` sufixo a um tipo. Para tipos que permitem valor nulo, o compilador avisa se você não verificar `null` antes de desreferenciar a expressão. Para tipos não anuláveis, o compilador avisa se você pode estar atribuindo um valor `null` a essa variável. Esses recursos podem minimizar a geração de uma System.NullReferenceException pelo seu aplicativo. A sintaxe pode ser conhecida pelo uso de `?` no TypeScript para propriedades opcionais.
5. ***LINQ***: A LINQ (consulta integrada à linguagem) fornece uma sintaxe comum para consultar e transformar dados, independentemente de seu armazenamento.

À medida que você aprende mais outras diferenças se tornam aparentes, mas muitas dessas diferenças são menores no escopo.

Alguns recursos e expressões familiares de JavaScript e TypeScript não estão disponíveis em C#:

1. ***tipos dinâmicos***: O C# usa digitação estática. Uma declaração de variável inclui o tipo e esse tipo não pode ser alterado. Há um tipo de dynamic em C# que fornece vinculação em tempo de execução.
2. ***Herança prototípica***: a herança em C# é parte integrante da declaração de tipo. Uma declaração C# `class` indica qualquer classe base. No JavaScript, você pode definir a `__proto__` propriedade para definir o tipo base em qualquer instância.
3. ***Idioma interpretado***: o código C# deve ser compilado antes de executá-lo. O código JavaScript pode ser executado diretamente no navegador.

Além disso, mais alguns recursos do TypeScript não estão disponíveis em C#:

1. ***Tipos*** de união: O C# não dá suporte a tipos de união. No entanto, as propostas de design estão em andamento.
2. ***Decoradores***: C# não tem decoradores. Alguns decoradores comuns, como `@sealed`, são palavras-chave reservadas em C#. Outros decoradores comuns podem ter atributos correspondentes. Para outros decoradores, você pode criar seus próprios atributos.
3. ***Sintaxe mais indulgente***: o compilador C# analisa o código mais estritamente do que o JavaScript exige.

Se você estiver criando um aplicativo Web, considere usar o Blazor para criar seu aplicativo. Blazor é uma estrutura web completa desenvolvida para .NET e C#. Os componentes blazor podem ser executados no servidor, como assemblies .NET ou no cliente usando WebAssembly. O Blazor dá suporte à interoperabilidade com suas bibliotecas JavaScript ou TypeScript favoritas.

Roteiro para desenvolvedores do Python aprendendo o C#

17/03/2025

C# e Python compartilham conceitos semelhantes. Esses constructos familiares o ajudam a aprender o C# quando você já conhece o Python.

1. ***Orientado a objeto***: Python e C# são linguagens orientadas aos objetos. Todos os conceitos em torno das classes no Python se aplicam no C#, mesmo que a sintaxe seja diferente.
2. ***Multiplataforma***: Python e C# são linguagens de multiplataforma. Os aplicativos escritos em qualquer idioma podem ser executados em várias plataformas.
3. ***Coleta de lixo***: ambas as linguagens empregam o gerenciamento automático de memória pela coleta de lixo. O runtime recupera a memória de objetos que não são referenciados.
4. ***Fortemente tipado***: Python e C# são linguagens fortemente tipadas. A coerção de tipos não ocorre de maneira implícita. Há diferenças descritas posteriormente, pois o C# é tipado estaticamente, enquanto o Python é digitado dinamicamente.
5. ***Assíncrono/Aguardar***: os recurso `async` e `await` do Python foram inspirados diretamente pelos suportes `async` e `await` do C#.
6. ***Padrões correspondentes***: a expressão `match` do Python e os padrões correspondentes são semelhantes à expressão padrões correspondentes `switch` do C#. Use-as para inspecionar uma expressão de dados complexa para determinar se ela corresponde a um padrão.
7. ***Palavras-chave de instrução***: Python e C# compartilham muitas palavras-chave, como `if`, `else`, `while`, `for`, e muitas outras. Embora nem todas as sintaxes sejam iguais, há similaridade suficiente para ler o C# se você conhecer o Python.

Ao aprender C#, você descobre esses conceitos importantes em que o C# é diferente do Python:

1. ***Recuo versus tokens***: no Python, as linhas novas e o recuo são elementos sintáticos de primeira classe. No C#, o espaço em branco não é significativo. Tokens, como `;` instruções separadas e outros tokens `{` e `}` escopo do bloco de controle para `if` e outras instruções de bloco. Entretanto, para facilitar a leitura, a maioria dos estilos de codificação (incluindo o estilo usado nesses documentos) usa o recuo para reforçar os escopos de bloco declarados por `{` e `}`.
2. ***Digitação estática***: no C#, uma declaração de variável inclui seu tipo. Reatribuir uma variável em um objeto de um tipo diferente gera um erro do compilador. No Python, o tipo poderá ser alterado quando for reatribuído.

3. ***Tipos que permitem valor nulo***: as variáveis no C# podem ser *anuláveis* ou *não anuláveis*. Um tipo não anulável é aquele que não pode ser nulo (ou nada). Ele sempre se refere a um objeto válido. Por outro lado, um tipo anulável pode se referir a um objeto válido ou nulo.
4. ***LINQ***: as palavras-chave da expressão de consulta que compõem a Consulta Integrada à Linguagem (LINQ) não são palavras-chave no Python. Porém, as bibliotecas do Python, como `itertools`, `more-itertools` e `py-linq`, fornecem funcionalidades semelhantes.
5. ***Genéricos***: os genéricos no C# usam digitação estática no C# para fazer declarações sobre os argumentos fornecidos para os parâmetros de tipo. Um algoritmo genérico pode precisar especificar restrições que um tipo de argumento deve satisfazer.

Por fim, há os recursos de linguagem Python que não estão disponíveis no C#:

1. ***Digitação estrutural (pato)***: no C#, os tipos têm nomes e declarações. Exceto por tuplas, os tipos com a mesma estrutura não são intercambiáveis.
2. ***REPL***: o C# não tem um REPL (Read-Eval-Print Loop) para criar rapidamente protótipos de soluções.
3. ***Espaço em branco significativo***: você precisa usar corretamente as chaves `{` e `}` para observar o escopo de bloco.

Aprenda no C# se você souber que o Python é uma jornada tranquila. Os idiomas têm conceitos e expressões semelhantes a serem usados.

Estrutura geral de um programa C#

22/06/2025

Os programas C# consistem em um ou mais arquivos. Cada arquivo contém zero ou mais namespaces. Um namespace contém tipos como classes, structs, interfaces, enumerações e delegados ou outros namespaces. O exemplo a seguir é o esqueleto de um programa C# que contém todos esses elementos.

C#

```
using System;

Console.WriteLine("Hello world!");

namespace YourNamespace
{
    class YourClass
    {
    }

    struct YourStruct
    {
    }

    interface IYourInterface
    {
    }

    delegate int YourDelegate();

    enum YourEnum
    {
    }

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

O exemplo anterior usa *declarações de nível superior* para o ponto de entrada do programa. Somente um arquivo pode ter instruções de nível superior. O ponto de entrada do programa é a primeira linha de texto do programa nesse arquivo. Nesse caso, é `Console.WriteLine("Hello world!");`. Você também pode criar um método estático chamado Main como o ponto de entrada do programa, conforme mostrado no exemplo a seguir:

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

    struct YourStruct
    {
    }

    interface IYourInterface
    {
    }

    delegate int YourDelegate();

    enum YourEnum
    {
    }

    namespace YourNestedNamespace
    {
        struct YourStruct
        {
        }
    }

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

Nesse caso, o programa será iniciado na primeira linha do método `Main`, que é `Console.WriteLine("Hello world!");`

Expressões e instruções

Os programas C# são criados usando *expressões* e *instruções*. As expressões produzem um valor e as instruções executam uma ação:

Uma *expressão* é uma combinação de valores, variáveis, operadores e chamadas de método que são avaliadas como um único valor. As expressões produzem um resultado e podem ser

usadas sempre que um valor é esperado. Os exemplos a seguir são expressões:

- `42` (valor literal)
- `x + y` (operação aritmética)
- `Math.Max(a, b)` (chamada de método)
- `condition ? trueValue : falseValue` (expressão condicional)
- `new Person("John")` (criação de objeto)

Uma *instrução* é uma instrução completa que executa uma ação. Instruções não retornam valores; Em vez disso, eles controlam o fluxo do programa, declaram variáveis ou executam operações. Os exemplos a seguir são instruções:

- `int x = 42;` (declaração de declaração)
- `Console.WriteLine("Hello");` (instrução expression - encapsula uma expressão de chamada de método)
- `if (condition) { /* code */ }` (instrução condicional)
- `return result;` (instrução return)

A distinção chave: expressões são avaliadas como valores, enquanto as instruções executam ações. Alguns constructos, como chamadas de método, podem ser ambos. Por exemplo, `Math.Max(a, b)` é uma expressão quando usada em `int result = Math.Max(a, b);`, mas se torna uma instrução de expressão quando escrita sozinha como `Math.Max(a, b);`.

Para obter informações detalhadas sobre instruções, consulte Instruções. Para obter informações sobre membros aptos à expressão e outros recursos de expressão, consulte membros aptos para expressão.

Seções relacionadas

Você aprenderá sobre esses elementos de programa na seção "tipos " do guia de conceitos básicos.

- Classes
- Estruturas
- Namespaces
- Interfaces
- Enumerações
- Delegados

Especificação da linguagem C#

Para obter mais informações, veja Noções básicas na Especificação da linguagem C#. A especificação de idioma é a fonte definitiva para a sintaxe e o uso de C#.

Main() e argumentos de linha de comando

Artigo • 31/01/2025

O método `Main` é o ponto de entrada de um aplicativo C#. Quando o aplicativo é iniciado, o método `Main` é o primeiro método invocado.

Pode haver apenas um ponto de entrada em um programa C#. Se tiver mais de uma classe que tenha um método `Main`, você deverá compilar seu programa com a opção do compilador **StartupObject** para especificar qual método `Main` será usado como ponto de entrada. Para obter mais informações, consulte **StartupObject** (opções de compilador do C#).
Veja abaixo o exemplo em que a primeira linha executada exibirá o número de argumentos de linha de comando:

C#

```
class TestClass
{
    static void Main(string[] args)
    {
        Console.WriteLine(args.Length);
    }
}
```

Você também pode usar Instruções de nível superior em um arquivo como o ponto de entrada no seu aplicativo. Assim como o método `Main`, as instruções de nível superior também podem retornar valores e acessar argumentos de linha de comando. Para obter mais informações, consulte Instruções de nível superior.
O exemplo a seguir usa um loop `foreach` para exibir os argumentos de linha de comando usando a variável `args` e, no final do programa, retorna um código de êxito (`0`):

C#

```
using System.Text;

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

foreach (var arg in args)
{
    builder.AppendLine($"Argument={arg}");
```

```
}

Console.WriteLine(builder.ToString());

return 0;
```

Visão geral

- O método `Main` é o ponto de entrada de um programa executável; é onde o controle do programa começa e termina.
- `Main` precisa ser declarado dentro de uma classe ou struct. A delimitação `class` pode ser `static`.
- `Main` deve ser static.
- `Main` pode ter qualquer modificador de acesso (exceto `file`).
- `Main` pode ter o tipo de retorno `void`, `int`, `Task` ou `Task<int>`.
- Se e somente se `Main` retornar um `Task` ou `Task<int>`, a declaração de `Main` pode incluir o modificador async. Isso exclui especificamente um método `async void Main`.
- O método `Main` pode ser declarado com ou sem um parâmetro `string[]` que contém os argumentos de linha de comando. Ao usar o Visual Studio para criar aplicativos do Windows, você pode adicionar o parâmetro manualmente ou usar o método GetCommandLineArgs() para obter os argumentos de linha de comando. Os parâmetros são lidos como argumentos de linha de comando indexados por zero. Ao contrário do C e C++, o nome do programa não é tratado como o primeiro argumento de linha de comando na matriz `args`, mas é o primeiro elemento do método GetCommandLineArgs().

A seguinte lista mostra as declarações `Main` mais comuns:

C#

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

Os exemplos anteriores não especificam um modificador de acesso, portanto, eles são implicitamente `private` por padrão. Isso é típico, mas é possível especificar qualquer

modificador de acesso explícito.

> **Dica**
>
> A adição dos tipos de retorno `async`, `Task` e `Task<int>` simplifica o código do programa quando os aplicativos do console precisam iniciar e realizar operações assíncronas `await` no `Main`.

Valores de retorno de Main()

Você pode retornar um `int` do método `Main` ao definir o método de uma das seguintes maneiras:

Expandir a tabela

Declaração `Main`	**Código do método `Main`**
`static int Main()`	Nenhum uso de `args` ou `await`
`static int Main(string[] args)`	Usa `args`, nenhum uso de `await`
`static async Task<int> Main()`	Nenhum uso de `args`, usa `await`
`static async Task<int> Main(string[] args)`	Usa `args` e `await`

Se o valor retornado de `Main` não for usado, o retorno de `void` ou `Task` permite um código um pouco mais simples.

Expandir a tabela

Declaração `Main`	**Código do método `Main`**
`static void Main()`	Nenhum uso de `args` ou `await`
`static void Main(string[] args)`	Usa `args`, nenhum uso de `await`
`static async Task Main()`	Nenhum uso de `args`, usa `await`
`static async Task Main(string[] args)`	Usa `args` e `await`

No entanto, o retorno de `int` ou `Task<int>` habilita o programa a comunicar informações de status para outros programas ou scripts, que invocam o arquivo executável.

O exemplo a seguir mostra como o código de saída para o processo pode ser acessado.

Este exemplo usa ferramentas de linha de comando do .NET Core. Se você não estiver familiarizado com as ferramentas de linha de comando do .NET Core, poderá aprender sobre elas neste artigo de introdução.

Crie um novo aplicativo ao executar `dotnet new console`. Modifique o método `Main` em *Program.cs* da seguinte maneira:

C#

```
class MainReturnValTest
{
    static int Main()
    {
        //...
        return 0;
    }
}
```

Lembre-se de salvar este programa como *MainReturnValTest.cs*.

Quando um programa é executado no Windows, qualquer valor retornado da função `Main` é armazenado em uma variável de ambiente. Essa variável de ambiente pode ser recuperada usando `ERRORLEVEL` de um arquivo em lotes ou `$LastExitCode` do PowerShell.

Você pode construir o aplicativo usando o comando dotnet CLI `dotnet build`.

Em seguida, crie um script do PowerShell para executar o aplicativo e exibir o resultado. Cole o código a seguir em um arquivo de texto e salve-o como `test.ps1` na pasta que contém o projeto. Execute o script do PowerShell ao digitar `test.ps1` no prompt do PowerShell.

Como o código retorna zero, o arquivo em lotes relatará êxito. No entanto, se você alterar o MainReturnValTest.cs para retornar um valor diferente de zero e recompilar o programa, a execução subsequente do script do PowerShell reportará falha.

PowerShell

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

```
}
Write-Host "Return value = " $LastExitCode
```

Saída

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

Valores retornados de Async Main

Quando você declara um valor retornado `async` para `Main`, o compilador gera o código clichê para chamar métodos assíncronos em `Main`. Se você não especificar a palavra-chave `async`, precisará escrever esse código por conta própria, conforme mostrado no exemplo a seguir. O código no exemplo garante que o programa seja executado até que a operação assíncrona seja concluída:

C#

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

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

Este código clichê pode ser substituído por:

C#

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

    private static async Task<int> AsyncConsoleWork()
    {
        return 0;
```

```
    }
}
```

Em ambos os exemplos, o corpo principal do programa está no corpo do método `AsyncConsoleWork()`.

A vantagem de declarar `Main` como `async` é que o compilador sempre gera o código correto.

Quando o ponto de entrada do aplicativo retorna um `Task` ou `Task<int>`, o compilador gera um novo ponto de entrada que chama o método de ponto de entrada declarado no código do aplicativo. Supondo que esse ponto de entrada é chamado `$GeneratedMain`, o compilador gera o código a seguir para esses pontos de entrada:

- `static Task Main()` resulta no compilador emitindo o equivalente a `private static void $GeneratedMain() => Main().GetAwaiter().GetResult();`
- `static Task Main(string[])` resulta no compilador emitindo o equivalente a `private static void $GeneratedMain(string[] args) => Main(args).GetAwaiter().GetResult();`
- `static Task<int> Main()` resulta no compilador emitindo o equivalente a `private static int $GeneratedMain() => Main().GetAwaiter().GetResult();`
- `static Task<int> Main(string[])` resulta no compilador emitindo o equivalente a `private static int $GeneratedMain(string[] args) => Main(args).GetAwaiter().GetResult();`

> **Observação**
>
> Se os exemplos usassem o modificador `async` no método `Main`, o compilador geraria o mesmo código.

Argumentos de linha de comando

Você pode enviar argumentos para o método `Main` definindo o método de uma das seguintes maneiras:

Expandir a tabela

Declaração `Main`	**Código do método** `Main`
`static void Main(string[] args)`	Nenhum valor retornado, nenhum uso de `await`

Declaração `Main`	Código do método `Main`
`static int Main(string[] args)`	Valor retornado, nenhum uso de `await`
`static async Task Main(string[] args)`	Nenhum valor retornado, usa `await`
`static async Task<int> Main(string[] args)`	Valor retornado, usa `await`

Se os argumentos não forem usados, você poderá omitir `args` da assinatura do método para um código ligeiramente mais simples:

Expandir a tabela

Declaração `Main`	Código do método `Main`
`static void Main()`	Nenhum valor retornado, nenhum uso de `await`
`static int Main()`	Valor retornado, nenhum uso de `await`
`static async Task Main()`	Nenhum valor retornado, usa `await`
`static async Task<int> Main()`	Valor retornado, usa `await`

Observação

Você também pode usar Environment.CommandLine ou Environment.GetCommandLineArgs para acessar os argumentos de linha de comando de qualquer ponto em um console ou um aplicativo do Windows Forms. Para habilitar os argumentos de linha de comando na declaração do método `Main` em um aplicativo do Windows Forms, você precisa modificar manualmente a declaração de `Main`. O código gerado pelo designer do Windows Forms cria um `Main` sem um parâmetro de entrada.

O parâmetro do método `Main` é uma matriz String que representa os argumentos de linha de comando. Geralmente você determina se os argumentos existem testando a propriedade `Length`, por exemplo:

C#

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

> **Dica**
>
> A matriz `args` não pode ser nula. Portanto, é seguro acessar a propriedade `Length` sem verificação de nulos.

Você também pode converter os argumentos de cadeia de caracteres em tipos numéricos, usando a classe Convert ou o método `Parse`. Por exemplo, a instrução a seguir converte o `string` em um número `long` usando o método Parse:

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

Também é possível usar o tipo `long` de C#, que funciona como alias de `Int64`:

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

Você também pode usar o método da classe `Convert`, o `ToInt64`, para fazer a mesma coisa:

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

Para obter mais informações, consulte Parse e Convert.

> **Dica**
>
> A análise de argumentos de linha de comando pode ser complexa. Considere usar a biblioteca System.CommandLine (atualmente em beta) para simplificar o processo.

O exemplo a seguir mostra como usar argumentos de linha de comando em um aplicativo de console. O aplicativo recebe um argumento em tempo de execução, converte o argumento em um número inteiro e calcula o fatorial do número. Se nenhum argumento for fornecido, o aplicativo emitirá uma mensagem que explica o uso correto do programa.

Para compilar e executar o aplicativo em um prompt de comando, siga estas etapas:

1. Cole o código a seguir em qualquer editor de texto e, em seguida, salve o arquivo como um arquivo de texto com o nome *Factorial.cs*.

C#

```
public class Functions
{
    public static long Factorial(int n)
    {
        // Test for invalid input.
        if ((n < 0) || (n > 20))
        {
            return -1;
        }

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

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

        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;
        }

        long result = Functions.Factorial(num);

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

        return 0;
```

```
    }
}
```

No início do método `Main`, o programa testa se argumentos de entrada não foram fornecidos comparando o comprimento do argumento `args` com `0` e exibe a ajuda se nenhum argumento for encontrado.
Se forem fornecidos argumentos (`args.Length` for maior que 0), o programa tentará converter os argumentos de entrada em números. Isso gerará uma exceção se o argumento não for um número.
Depois que o fatorial é calculado, armazenado na variável `result` do tipo `long`, o resultado detalhado é impresso dependendo da variável `result`.

2. Na tela **Inicial** ou no menu **Iniciar**, abra uma janela **Prompt de Comando do Desenvolvedor** do Visual Studio e, em seguida, navegue até a pasta que contém o arquivo que você acabou de criar.

3. Digite o seguinte comando para compilar o aplicativo.

   `dotnet build`

   Se seu aplicativo não tiver erros de compilação, um arquivo executável chamado *Factorial.exe* será criado.

4. Digite o seguinte comando para calcular o fatorial de 3:

   `dotnet run -- 3`

5. Se 3 for inserido na linha de comando como o argumento do programa, a saída será exibida: `The factorial of 3 is 6.`

> ⓘ **Observação**
>
> Ao executar um aplicativo no Visual Studio, você pode especificar argumentos de linha de comando na Página de depuração, Designer de Projeto.

Especificação da linguagem C#

Para obter mais informações, consulte a Especificação da linguagem C#. A especificação da linguagem é a fonte definitiva para a sintaxe e o uso de C#.

Confira também

- System.Environment
- Como exibir argumentos de linha de comando

Instruções de nível superior – Programas sem métodos `Main`

Artigo • 27/05/2025

Não é preciso incluir explicitamente um método `Main` em um projeto de aplicativo de console. Em vez disso, você pode usar o recurso de *instruções de nível superior* para minimizar o volume de código que precisa escrever.

Instruções de nível superior permitem que você escreva código executável diretamente na raiz de um arquivo, eliminando a necessidade de encapsular seu código em uma classe ou método. Isso significa que você pode criar programas sem a cerimônia de uma classe `Program` e um método `Main`. Nesse caso, o compilador gera uma classe `Program` com um método de ponto de entrada para o aplicativo. O nome do método gerado não é `Main`, é um detalhe de implementação que seu código não pode referenciar diretamente.

Aqui está um arquivo *Program.cs* que é um programa C# completo:

```
C#

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

Instruções de nível superior permitem que você escreva programas simples para utilitários pequenos, como o Azure Functions e o GitHub Actions. Elas também simplificam para novos programadores C# começar a aprender e escrever código.

As seções a seguir explicam as regras sobre o que você pode ou não fazer com instruções de nível superior.

Apenas um arquivo de nível superior

Cada aplicativo deve ter apenas um ponto de entrada. Cada projeto pode ter apenas um arquivo com instruções de nível superior. Colocar instruções de nível superior em mais de um arquivo de um projeto resulta no seguinte erro do compilador:

> CS8802 Somente uma unidade de compilação pode conter instruções de nível superior.

Um projeto pode ter qualquer número de arquivos de código-fonte que não tenham instruções de nível superior.

Nenhum outro ponto de entrada

Você pode escrever um método `Main` explicitamente, mas ele não pode funcionar como um ponto de entrada. O compilador emite o seguinte aviso:

> CS7022 O ponto de entrada do programa é o código global, ignorando o ponto de entrada "Main()".

Em um projeto com instruções de nível superior, você não pode usar a opção do compilador `-main` para selecionar o ponto de entrada, mesmo que o projeto tenha um ou mais métodos `Main`.

`using` diretivas

Para o arquivo único que contém instruções de nível superior, as diretivas `using` devem vir primeiro nesse arquivo, como neste exemplo:

```
C#

using System.Text;

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

foreach (var arg in args)
{
    builder.AppendLine($"Argument={arg}");
}

Console.WriteLine(builder.ToString());

return 0;
```

Namespace global

As instruções de nível superior estão implicitamente no namespace global.

Namespaces e definições de tipo

Um arquivo com instruções de nível superior também pode conter namespaces e definições de tipo, mas devem vir após as instruções de nível superior. Por exemplo:

```
C#
```

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

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

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

`args`

Instruções de nível superior podem fazer referência à variável `args` para acessar quaisquer argumentos de linha de comando que foram inseridos. A variável `args` nunca será nula, mas o `Length` dela será zero se nenhum argumento de linha de comando tiver sido fornecido. Por exemplo:

C#

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

`await`

Você pode chamar um método assíncrono usando `await`. Por exemplo:

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

Código de saída do processo

Para retornar um valor `int` quando o aplicativo terminar, use a instrução `return` como faria em um método `Main` que retorna um `int`. Por exemplo:

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

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

Método de ponto de entrada implícito

O compilador gera um método para servir como o ponto de entrada do programa para um projeto com instruções de nível superior. A assinatura do método depende se as instruções de nível superior contêm a palavra-chave `await` ou a instrução `return`. A tabela a seguir mostra como seria a assinatura do método, usando o nome de método `Main` na tabela para fins de conveniência.

Expandir a tabela

O código de nível superior contém	Assinatura `Main` implícita
`await` e `return`	`static async Task<int> Main(string[] args)`
`await`	`static async Task Main(string[] args)`
`return`	`static int Main(string[] args)`
Não `await` nem `return`	`static void Main(string[] args)`

Especificação da linguagem C#

Para obter mais informações, consulte a Especificação da linguagem C#. A especificação da linguagem é a fonte definitiva para a sintaxe e o uso de C#.

Especificação de recurso – Instruções de nível superior

O sistema de tipo C#

19/06/2025

C# é uma linguagem fortemente tipada. Cada variável e constante tem um tipo, assim como todas as expressões que são avaliadas como um valor. Cada declaração de método especifica um nome, o tipo e a forma (valor, referência ou saída) de cada parâmetro de entrada e do valor de retorno. A biblioteca de classes .NET define tipos numéricos internos e tipos complexos que representam uma ampla variedade de constructos. Isso inclui o sistema de arquivos, conexões de rede, coleções e matrizes de objetos e datas. Um programa C# típico usa tipos da biblioteca de classes e tipos definidos pelo usuário que modelam os conceitos específicos para o domínio de problema do programa.

As informações armazenadas em um tipo podem incluir os seguintes itens:

- O espaço de armazenamento que uma variável do tipo requer.
- Os valores máximos e mínimos que ele pode representar.
- Os membros (métodos, campos, eventos e assim por diante) que ele contém.
- O tipo base do qual ele herda.
- As interfaces que ele implementa.
- As operações permitidas.

O compilador usa informações de tipo para garantir que todas as operações executadas em seu código sejam *de tipo seguro*. Por exemplo, se você declarar uma variável de tipo int, o compilador permitirá que você use a variável em operações de adição e subtração. Se você tentar executar essas mesmas operações em uma variável de tipo bool, o compilador gerará um erro, conforme mostrado no exemplo a seguir:

C#

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

bool test = true;

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

Observação

Os desenvolvedores C e C++, observem que, em C#, `bool` não é conversível para `int`.

O compilador insira as informações de tipo no arquivo executável como metadados. O CLR (Common Language Runtime) usa esses metadados em tempo de execução para garantir ainda mais a segurança do tipo quando aloca e recupera a memória.

Especificando tipos em declarações de variável

Ao declarar uma variável ou constante em um programa, você deve especificar seu tipo ou usar a var palavra-chave para permitir que o compilador infera o tipo. O exemplo a seguir mostra algumas declarações variáveis que usam tipos numéricos internos e tipos complexos definidos pelo usuário:

C#

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

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

Os tipos de parâmetros de método e valores retornados são especificados na declaração do método. A assinatura a seguir mostra um método que requer um `int` argumento de entrada e retorna uma cadeia de caracteres:

C#

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

Depois de declarar uma variável, você não poderá reenviá-la com um novo tipo e não poderá atribuir um valor não compatível com seu tipo declarado. Por exemplo, você não pode declarar um `int` e, em seguida, atribuir a ele um valor booliano de `true`. No entanto, os valores podem ser convertidos em outros tipos, por exemplo, quando são atribuídos a novas variáveis ou

passados como argumentos de método. Uma *conversão de tipo* que não causa perda de dados é executada automaticamente pelo compilador. Uma conversão que pode causar perda de dados requer um *cast* no código-fonte.

Para obter mais informações, consulte Conversões Cast e Conversões de Tipo.

Tipos internos

O C# fornece um conjunto padrão de tipos internos. Eles representam inteiros, valores de ponto flutuante, expressões boolianas, caracteres de texto, valores decimais e outros tipos de dados. Também há tipos `string` e `object` internos. Esses tipos estão disponíveis para você usar em qualquer programa C#. Para obter a lista completa dos tipos internos, consulte tipos internos.

Tipos personalizados

Use as construções struct, class, interface, enume record para criar seus próprios tipos personalizados. A própria biblioteca de classes do .NET é uma coleção de tipos personalizados que você pode usar em seus próprios aplicativos. Por padrão, os tipos usados com mais frequência na biblioteca de classes estão disponíveis em qualquer programa C#. Outras pessoas ficam disponíveis somente quando você adiciona explicitamente uma referência de projeto ao assembly que as define. Depois que o compilador tiver uma referência ao assembly, você poderá declarar variáveis (e constantes) dos tipos declarados nesse assembly no código-fonte. Para obter mais informações, consulte a Biblioteca de Classes do .NET.

Uma das primeiras decisões tomadas ao definir um tipo é decidir qual construção usar para seu tipo. A lista a seguir ajuda a tomar essa decisão inicial. Há sobreposição nas escolhas. Na maioria dos cenários, mais de uma opção é uma opção razoável.

- Se o tamanho do armazenamento de dados for pequeno, não mais do que 64 bytes, escolha um `struct` ou `record struct`.
- Se o tipo for imutável ou se você quiser mutação não estruturativa, escolha um `struct` ou `record struct`.
- Se o seu tipo deve ter semântica de valor para igualdade, escolha um `record class` ou `record struct`.
- Se o tipo for usado principalmente para armazenar dados, não para comportamento, escolha um `record class` ou `record struct`.
- Se o tipo fizer parte de uma hierarquia de herança, escolha um `record class` ou um `class`.
- Se o tipo usar polimorfismo, escolha um `class`.

- Se a finalidade primária for o comportamento, escolha um `class`.

O sistema de tipo comum

É importante entender dois pontos fundamentais sobre o sistema de tipos no .NET:

- Ele dá suporte ao princípio da herança. Os tipos podem derivar de outros tipos, chamados *de tipos base*. O tipo derivado herda (com algumas restrições) os métodos, as propriedades e outros membros do tipo base. O tipo base pode, por sua vez, derivar de algum outro tipo, nesse caso, o tipo derivado herda os membros de ambos os tipos base em sua hierarquia de herança. Todos os tipos, incluindo tipos numéricos internos, como System.Int32 (palavra-chave C#: `int`), derivam, em última análise, de um único tipo base, que é System.Object (palavra-chave C#: object). Essa hierarquia de tipo unificado é chamada de CTS (Common Type System). Para obter mais informações sobre herança em C#, consulte Herança.
- Cada tipo no CTS é definido como um tipo de *valor* ou um *tipo de referência*. Esses tipos incluem todos os tipos personalizados na biblioteca de classes do .NET e também seus próprios tipos definidos pelo usuário. Os tipos que você define usando a `struct` palavra-chave são tipos de valor; todos os tipos numéricos internos são `structs`. Os tipos que você define usando a palavra-chave `class` ou a palavra-chave `record` são tipos de referência. Tipos de referência e tipos de valor têm regras de tempo de compilação diferentes e comportamento de tempo de execução diferente.

A ilustração a seguir mostra a relação entre tipos de valor e tipos de referência no CTS.

> ⓘ **Observação**
>
> Você pode ver que os tipos mais usados são todos organizados no System namespace. No entanto, o namespace no qual um tipo está contido não tem relação com se é um tipo de valor ou tipo de referência.

Classes e structs são duas das construções básicas do sistema de tipo comum no .NET. Cada uma é essencialmente uma estrutura de dados que encapsula um conjunto de dados e comportamentos que pertencem juntos como uma unidade lógica. Os dados e comportamentos são os *membros* da classe, struct ou registro. Os membros incluem seus métodos, propriedades, eventos e assim por diante, conforme listado posteriormente neste artigo.

Uma declaração de classe, struct ou registro é como um blueprint usado para criar instâncias ou objetos em tempo de execução. Se você definir uma classe, struct ou registro nomeado `Person`, `Person` será o nome do tipo. Se você declarar e inicializar uma variável `p` do tipo `Person`, `p` será considerado um objeto ou instância de `Person`. Várias instâncias do mesmo tipo de `Person` podem ser criadas e cada instância pode ter valores diferentes em suas propriedades e campos.

Uma classe é um tipo de referência. Quando um objeto do tipo é criado, a variável à qual o objeto é atribuído contém apenas uma referência a essa memória. Quando a referência de objeto é atribuída a uma nova variável, a nova variável se refere ao objeto original. As alterações feitas por meio de uma variável são refletidas na outra variável porque ambas se referem aos mesmos dados.

Um struct é um tipo de valor. Quando um struct é criado, a variável à qual o struct é atribuído contém os dados reais do struct. Quando o struct é atribuído a uma nova variável, ele é copiado. A nova variável e a variável original, portanto, contêm duas cópias separadas dos mesmos dados. As alterações feitas em uma cópia não afetam a outra cópia.

Os tipos de registro podem ser tipos de referência (`record class`) ou tipos de valor (`record struct`). Os tipos de registro contêm métodos que dão suporte à igualdade de valor.

Em geral, as classes são usadas para modelar um comportamento mais complexo. As classes normalmente armazenam dados que devem ser modificados após a criação de um objeto de classe. Structs são mais adequados para estruturas de dados pequenas. Os structs normalmente armazenam dados que não devem ser modificados após a criação do struct. Os tipos de registro são estruturas de dados com membros adicionais sintetizados pelo compilador. Normalmente, os registros armazenam dados que não devem ser modificados após a criação do objeto.

Tipos de valor

Os tipos de valor derivam de System.ValueType, que deriva de System.Object. Os tipos que derivam de System.ValueType têm um comportamento especial no CLR. As variáveis de tipo de valor contêm diretamente seus valores. A memória de um struct é embutida em qualquer contexto em que a variável seja declarada. Não há nenhuma alocação de heap separada ou sobrecarga de coleta de lixo para variáveis de tipo de valor. É possível declarar tipos `record struct` que são tipos de valor e incluir os membros sintetizados para registros.

Há duas categorias de tipos de valor: `struct` e `enum`.

Os tipos numéricos internos são structs e têm campos e métodos que você pode acessar:

```
C#

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

Mas você declara e atribui valores a eles como se fossem tipos simples de não agregação:

```
C#

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

Os tipos de valor são lacrados. Você não pode derivar um tipo de qualquer tipo de valor, por exemplo, System.Int32. Você não pode definir um struct para herdar de qualquer classe ou struct definido pelo usuário porque um struct só pode herdar de System.ValueType. No entanto, um struct pode implementar uma ou mais interfaces. É possível converter um tipo struct em qualquer tipo de interface que ele implementa. Essa conversão faz com que uma operação de *conversão boxing* encapsule o struct dentro de um objeto de tipo de referência no heap gerenciado. As operações de conversão boxing ocorrem quando você passa um tipo de valor para um método que usa um System.Object ou qualquer tipo de interface como parâmetro de entrada. Para obter mais informações, consulte Boxing e Unboxing.

Use a palavra-chave struct para criar seus próprios tipos de valor personalizados. Normalmente, um struct é usado como um contêiner para um pequeno conjunto de variáveis relacionadas, conforme mostrado no exemplo a seguir:

```
C#

public struct Coords
{
```

```
    public int x, y;

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

Para obter mais informações sobre structs, consulte Tipos de estrutura. Para obter mais informações sobre tipos de valor, consulte Tipos de valor.

A outra categoria de tipos de valor é `enum`. Uma enumeração define um conjunto de constantes integrais nomeadas. Por exemplo, a enumeração System.IO.FileMode na biblioteca de classes .NET contém um conjunto de inteiros constantes nomeados que especificam como um arquivo deve ser aberto. Ele é definido conforme mostrado no exemplo a seguir:

C#

```
public enum FileMode
{
    CreateNew = 1,
    Create = 2,
    Open = 3,
    OpenOrCreate = 4,
    Truncate = 5,
    Append = 6,
}
```

A constante System.IO.FileMode.Create tem um valor de 2. No entanto, o nome é muito mais significativo para humanos que leem o código-fonte e, por esse motivo, é melhor usar enumerações em vez de números literais constantes. Para obter mais informações, consulte System.IO.FileMode.

Todas as enumerações herdam de System.Enum, que herda de System.ValueType. Todas as regras que se aplicam a structs também se aplicam a enumerações. Para obter mais informações sobre enumerações, consulte tipos de enumeração.

Tipos de referência

Um tipo que é definido como `class`, `record`, delegate, matriz ou interface é um reference type.

Quando você declara uma variável de um reference type, ela contém o valor null até que você o atribua com uma instância desse tipo ou crie uma usando o new operador. A criação e a atribuição de uma classe são demonstradas no exemplo a seguir:

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

Um interface não pode ser instanciado diretamente usando o operador new. Em vez disso, crie e atribua uma instância de uma classe que implementa a interface. Considere o seguinte exemplo:

```C#
MyClass myClass = new MyClass();

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

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

Quando o objeto é criado, a memória é alocada no heap gerenciado. A variável contém apenas uma referência ao local do objeto. Os tipos no heap gerenciado exigem sobrecarga quando são alocados e recuperados. *A coleta de lixo* é a funcionalidade de gerenciamento automático de memória do CLR, que executa o reaproveitamento. No entanto, a coleta de lixo também é altamente otimizada e, na maioria dos cenários, não cria um problema de desempenho. Para obter mais informações sobre a coleta de lixo, consulte Gerenciamento Automático de Memória.

Todas as matrizes são tipos de referência, mesmo que seus elementos sejam tipos de valor. As matrizes derivam implicitamente da classe System.Array. Declare e use-os com a sintaxe simplificada fornecida pelo C#, conforme mostrado no exemplo a seguir:

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

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

Os tipos de referência dão suporte total à herança. Quando você cria uma classe, é possível herdar de qualquer outra interface ou classe que não esteja definida como selada. Outras classes podem herdar de sua classe e substituir seus métodos virtuais. Para obter mais informações sobre como criar suas próprias classes, consulte Classes, structs e registros. Para obter mais informações sobre herança e métodos virtuais, consulte Herança.

Tipos de valores literais

Em C#, os valores literais recebem um tipo do compilador. Você pode especificar como um literal numérico deve ser digitado acrescentando uma letra ao final do número. Por exemplo, para especificar que o valor `4.56` deve ser tratado como um `float`, acrescente um "f" ou "F" após o número: `4.56f`. Se nenhuma letra for anexada, o compilador inferirá um tipo para o literal. Para obter mais informações sobre quais tipos podem ser especificados com sufixos de letra, consulte tipos numéricos integrais e tipos numéricos de ponto flutuante.

Como os literais são tipados e todos os tipos derivam basicamente de System.Object, você pode escrever e compilar o código como o seguinte:

C#

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

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

Tipos genéricos

Um tipo pode ser declarado com um ou mais *parâmetros de tipo* que servem como um espaço reservado para o tipo real (o *tipo concreto*). O código do cliente fornece o tipo concreto quando cria uma instância do tipo. Esses tipos são chamados *de tipos genéricos*. Por exemplo, o tipo System.Collections.Generic.List<T> .NET tem um parâmetro de tipo que, por convenção, recebe o nome `T`. Ao criar uma instância do tipo, especifique o tipo dos objetos que a lista contém, por exemplo: `string`

C#

```
List<string> stringList = new List<string>();
stringList.Add("String example");
// compile time error adding a type other than a string:
stringList.Add(4);
```

O uso do parâmetro de tipo possibilita reutilizar a mesma classe para manter qualquer tipo de elemento, sem precisar converter cada elemento em objeto. Classes de coleção genérica são chamadas *de coleções fortemente tipadas* porque o compilador conhece o tipo específico dos elementos da coleção e pode gerar um erro no momento da compilação se, por exemplo, você

tentar adicionar um inteiro ao `stringList` objeto no exemplo anterior. Para obter mais informações, consulte Genéricos.

Tipos implícitos, tipos anônimos e tipos de valor anuláveis

Você pode digitar implicitamente uma variável local (mas não membros de classe) usando a var palavra-chave. A variável ainda recebe um tipo em tempo de compilação, mas o tipo é fornecido pelo compilador. Para obter mais informações, consulte Variáveis locais de tipo implícito.

Pode ser inconveniente criar um tipo nomeado para conjuntos simples de valores relacionados que você não pretende armazenar ou passar fora dos limites do método. Você pode criar *tipos anônimos* para essa finalidade. Para obter mais informações, consulte Tipos Anônimos.

Tipos de valor comuns não podem ter um valor de null. No entanto, você pode criar *tipos de valor anuláveis* acrescentando um `?` após o tipo. Por exemplo, `int?` é um `int` tipo que também pode ter o valor null. Os tipos que permitem valor nulo são instâncias do tipo struct genérico System.Nullable<T>. Tipos de valor anuláveis são especialmente úteis quando você está passando dados de e para bancos de dados nos quais valores numéricos podem ser `null`. Para obter mais informações, consulte tipos de valor anuláveis.

Tipo de tempo de compilação e tipo de tempo de execução

Uma variável pode ter diferentes tipos de tempo de compilação e tempo de execução. O *tipo de tempo de compilação* é o tipo declarado ou inferido da variável no código-fonte. O *tipo de tempo de execução* é o tipo da instância referenciada por essa variável. Geralmente, esses dois tipos são os mesmos, como no exemplo a seguir:

C#

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

Em outros casos, o tipo de tempo de compilação é diferente, conforme mostrado nos dois exemplos a seguir:

C#

```
object anotherMessage = "This is another string of characters";
```

```
IEnumerable<char> someCharacters = "abcdefghijklmnopqrstuvwxyz";
```

Em ambos os exemplos acima, o tipo de tempo de execução é um `string`. O tipo de tempo de compilação está `object` na primeira linha e `IEnumerable<char>` na segunda.

Se os dois tipos forem diferentes para uma variável, é importante entender quando o tipo de tempo de compilação e o tipo de tempo de execução se aplicam. O tipo de tempo de compilação determina todas as ações executadas pelo compilador. Essas ações do compilador incluem resolução de chamada de método, resolução de sobrecarga e conversões implícitas e explícitas disponíveis. O tipo de tempo de execução determina todas as ações resolvidas em tempo de execução. Essas ações em tempo de execução incluem o despacho de chamadas de método virtual, a avaliação das expressões `is` e `switch`, e outras APIs de teste de tipo. Para entender melhor como seu código interage com tipos, reconheça qual ação se aplica a qual tipo.

Seções relacionadas

Para obter mais informações, consulte os seguintes artigos:

- Tipos internos
- Tipos de valor
- Tipos de referência

Especificação da linguagem C#

Para obter mais informações, consulte a Especificação da Linguagem C# . A especificação de idioma é a fonte definitiva para a sintaxe e o uso de C#.

Declarar namespaces para organizar tipos

Artigo • 19/12/2024

Os namespaces são usados intensamente em programações de C# de duas maneiras. Em primeiro lugar, o .NET usa namespaces para organizar suas muitas classes, da seguinte maneira:

```
C#

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

System é um namespace e Console é uma classe nesse namespace. A palavra-chave `using` pode ser usada para que o nome completo não seja necessário, como no exemplo a seguir:

```
C#

using System;
```

```
C#

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

Para saber mais, confira Diretiva using.

> **ⓘ Importante**
>
> Os modelos C# para o .NET 6 usam *instruções de nível superior*. Se você já tiver atualizado para o .NET 6, talvez seu aplicativo não corresponda ao código descrito neste artigo. Para obter mais informações, consulte o artigo sobre **Novos modelos C# geram instruções de nível superior**
>
> O SDK do .NET 6 também adiciona um conjunto de diretivas *implícitas* `global using` para projetos que usam os seguintes SDKs:
>
> - Microsoft.NET.Sdk
> - Microsoft.NET.Sdk.Web
> - Microsoft.NET.Sdk.Worker

Essas diretivas implícitas `global using` incluem os namespaces mais comuns para o tipo de projeto.

Para obter mais informações, consulte o artigo sobre [Diretivas de uso implícito]

Em segundo lugar, declarar seus próprios namespaces pode ajudar a controlar o escopo dos nomes de classe e de método em projetos de programação maiores. Use a palavra-chave namespace para declarar um namespace, como no exemplo a seguir:

C#

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

O nome do namespace deve ser um nome do identificador válido em C#.

Você pode declarar um namespace para todos os tipos definidos nesse arquivo, conforme mostrado no exemplo a seguir:

C#

```
namespace SampleNamespace;

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

A vantagem dessa nova sintaxe é que ela é mais simples, economizando espaço horizontal e chaves. Isso facilita a leitura do seu código.

Visão geral dos namespaces

Os namespaces têm as seguintes propriedades:

- Eles organizam projetos de códigos grandes.
- Eles são delimitados com o uso do operador `.`.
- A diretiva `using` elimina a necessidade de especificar o nome do namespace para cada classe.
- O namespace `global` é o namespace "raiz": `global::System` sempre se refere ao namespace System .NET.

Especificação da linguagem C#

Para saber mais, confira a seção Namespaces da Especificação da linguagem C#.

Introdução às classes

Artigo • 15/08/2024

Tipos de referência

Um tipo que é definido como uma class é um *tipo de referência*. No runtime, quando você declara uma variável de um tipo de referência, a variável contém o valor null até que você crie explicitamente uma instância da classe usando o operador new ou atribua a ela um objeto de um tipo compatível criado em outro lugar, conforme mostrado no exemplo a seguir:

C#

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

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

Quando o objeto é criado, memória suficiente é alocada no heap gerenciado para o objeto específico, e a variável contém apenas uma referência para o local do objeto. A memória usada por um objeto é recuperada pela funcionalidade de gerenciamento automático de memória do CLR, que é conhecida como *coleta de lixo*. Para obter mais informações sobre a coleta de lixo, consulte Gerenciamento automático de memória e coleta de lixo.

Declarando classes

As classes são declaradas usando a palavra-chave `class`, seguida por um identificador exclusivo, conforme mostrado no exemplo a seguir:

C#

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

Um modificador de acesso opcional precede a palavra-chave `class`. O acesso padrão para um tipo `class` é `internal`. Como public é usado nesse caso, qualquer pessoa pode criar instâncias dessa classe. O nome da classe segue a palavra-chave `class`. O nome da classe deve ser um nome do identificador válido em C#. O restante da definição é o corpo da classe, em que o comportamento e os dados são definidos. Campos, propriedades, métodos e eventos em uma classe são coletivamente denominados de *membros de classe*.

Criando objetos

Embora eles sejam usados algumas vezes de maneira intercambiável, uma classe e um objeto são coisas diferentes. Uma classe define um tipo de objeto, mas não é um objeto em si. Um objeto é uma entidade concreta com base em uma classe e, às vezes, é conhecido como uma instância de uma classe.

Os objetos podem ser criados usando a palavra-chave `new` seguida pelo nome da classe, dessa maneira:

```
C#

Customer object1 = new Customer();
```

Quando uma instância de uma classe é criada, uma referência ao objeto é passada de volta para o programador. No exemplo anterior, `object1` é uma referência a um objeto que é baseado em `Customer`. Esta referência refere-se ao novo objeto, mas não contém os dados de objeto. Na verdade, você pode criar uma referência de objeto sem criar um objeto:

```
C#

Customer object2;
```

Não recomendamos a criação de referências de objeto, que não faz referência a um objeto, porque tentar acessar um objeto por meio de uma referência desse tipo falhará em tempo de execução. Uma referência pode se referir a um objeto, criando um novo objeto ou atribuindo-a a um objeto existente, como abaixo:

```
C#

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

Esse código cria duas referências de objeto que fazem referência ao mesmo objeto. Portanto, qualquer alteração no objeto feita por meio de `object3` será refletida no usos posteriores de `object4`. Como os objetos que são baseados em classes são referenciados por referência, as classes são conhecidas como tipos de referência.

Construtores e inicialização

As seções anteriores introduziram a sintaxe para declarar um tipo de classe e criar uma instância desse tipo. Ao criar uma instância de um tipo, você deseja garantir que seus campos e propriedades sejam inicializados para valores úteis. Há várias maneiras de inicializar valores:

- Aceitar valores padrão
- Inicializadores de campo
- Parâmetros do construtor
- Inicializadores de objeto

Cada tipo .NET tem um valor padrão. Normalmente, esse valor é 0 para tipos de número e `null` para todos os tipos de referência. Você pode contar com esse valor padrão quando for razoável em seu aplicativo.

Quando o padrão .NET não é o valor certo, você pode definir um valor inicial usando um *inicializador de campo*:

C#

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

Você pode exigir que os chamadores forneçam um valor inicial definindo um *construtor* responsável por definir esse valor inicial:

C#

```
public class Container
{
    private int _capacity;

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

A partir do C# 12, você pode definir um *construtor primário* como parte da declaração de classe:

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

Adicionar parâmetros ao nome da classe define o *construtor primário*. Esses parâmetros estão disponíveis no corpo da classe, que inclui seus membros. Você pode usá-los para inicializar campos ou em qualquer outro lugar em que eles sejam necessários.

Você também pode usar o modificador `required` em uma propriedade e permitir que os chamadores usem um *inicializador de objeto* para definir o valor inicial da propriedade:

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

A adição da palavra-chave `required` determina que os chamadores devem definir essas propriedades como parte de uma expressão `new`:

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

Herança de classe

As classes dão suporte completo à *herança*, uma característica fundamental da programação orientada a objetos. Quando você cria uma classe, é possível herdar de qualquer outra classe que não esteja definida como sealed. Outras classes podem herdar de sua classe e substituir métodos virtuais de classe. Além disso, você pode implementar uma ou mais interfaces.

A herança é realizada usando uma *derivação*, o que significa que uma classe é declarada usando uma *classe base*, da qual ela herda o comportamento e os dados. Uma classe

base é especificada ao acrescentar dois-pontos e o nome de classe base depois do nome de classe derivada, dessa maneira:

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

Quando uma declaração de classe inclui uma classe base, ela herda todos os membros da classe base, exceto os construtores. Para obter mais informações, consulte Herança.

Uma classe no C# só pode herdar diretamente de uma classe base. No entanto, como uma classe base pode herdar de outra classe, uma classe poderia herdar indiretamente várias classes base. Além disso, uma classe pode implementar diretamente uma ou mais interfaces. Para obter mais informações, consulte Interfaces.

Uma classe pode ser declarada como abstract. Uma classe abstrata contém métodos abstratos que têm uma definição de assinatura, mas não têm implementação. As classes abstratas não podem ser instanciadas. Elas só podem ser usadas por meio de classes derivadas que implementam os métodos abstratos. Por outro lado, uma classe lacrada não permite que outras classes sejam derivadas dela. Para obter mais informações, consulte Classes e Membros de Classes Abstratos e Lacrados.

As definições de classe podem ser divididas entre arquivos de origem diferentes. Para obter mais informações, consulte Classes parciais e métodos.

Especificação da Linguagem C#

Para obter mais informações, consulte a Especificação da linguagem C#. A especificação da linguagem é a fonte definitiva para a sintaxe e o uso de C#.

Introdução aos tipos de registro em C#

Artigo • 06/02/2025

Um registro em C# é uma classe ou uma struct que oferece uma sintaxe e comportamento especiais para trabalhar com modelos de dados. O modificador `record` instrui o compilador a sintetizar membros que são úteis para tipos cuja função primária é armazenar dados. Esses membros incluem uma sobrecarga de ToString() e membros que dão suporte à igualdade de valor.

Quando usar registros

Considere usar um registro no lugar de uma classe ou struct nos seguintes cenários:

- Você deseja definir um modelo de dados que depende da igualdade de valor.
- Você deseja definir um tipo para o qual os objetos são imutáveis.

Igualdade de valores

Para registros, a igualdade de valor significa que duas variáveis de um tipo de registro são iguais se os tipos corresponderem e todos os valores de propriedade e campo forem iguais. Para outros tipos de referência, como classes, igualdade significa igualdade de referência por padrão, a menos que a igualdade de valor tenha sido implementada. Ou seja, duas variáveis de um tipo de classe são iguais se se referirem ao mesmo objeto. Métodos e operadores que determinam a igualdade de duas instâncias de registro usam igualdade de valor.

Nem todos os modelos de dados funcionam bem com igualdade de valor. Por exemplo, Entity Framework Core depende da igualdade de referência para garantir que ele use apenas uma instância de um tipo de entidade para algo que, conceitualmente, seja uma única entidade. Por esse motivo, os tipos de registro não são apropriados para uso como tipos de entidade no Entity Framework Core.

Imutabilidade

Um tipo imutável é aquele que impede que você altere quaisquer valores de propriedade ou campo de um objeto depois de instanciado. A imutabilidade pode ser útil quando você precisa que um tipo seja thread-safe ou que um código hash permaneça o mesmo em uma tabela de hash. Os registros fornecem sintaxe concisa para criar e trabalhar com tipos imutáveis.

A imutabilidade não é apropriada para todos os cenários de dados. Entity Framework Core, por exemplo, não dá suporte à atualização com tipos de entidade imutáveis.

Como os registros diferem de classes e structs

A mesma sintaxe que declara e instancia classes ou structs pode ser usada com registros. Basta substituir a palavra-chave `class` pelo `record` ou usar `record struct` em vez de `struct`. Da mesma forma, a mesma sintaxe para expressar relações de herança é suportada por classes de registro. Os registros diferem das classes das seguintes maneiras:

- Você pode usar parâmetros posicionais em um construtor primário para criar e instanciar um tipo com propriedades imutáveis.
- Os mesmos métodos e operadores que indicam igualdade de referência ou desigualdade em classes (como Object.Equals(Object) e `==`), indicam igualdade de valor ou desigualdade nos registros.
- Você pode usar uma expressão with para criar uma cópia de um objeto imutável com novos valores em propriedades selecionadas.
- O método `ToString` de um registro cria uma cadeia de caracteres formatada que mostra o nome do tipo de um objeto e os nomes e valores de todas as suas propriedades públicas.
- Um registro pode herdar de outro registro. Um registro não pode herdar de uma classe, e uma classe não pode herdar de um registro.

Os structs de registro diferem dos structs, pois o compilador sintetiza os métodos de igualdade e `ToString`. O compilador sintetiza um método `Deconstruct` para estruturas de registro posicionais.

O compilador sintetiza uma propriedade public init-only para cada parâmetro de construtor primário em um `record class`. Em um `record struct`, o compilador sintetiza uma propriedade pública de leitura/gravação. O compilador não cria propriedades para parâmetros de construtor primário em tipos de `class` e `struct` que não incluem o modificador `record`.

Exemplos

O exemplo a seguir define um registro público que usa parâmetros posicionais para declarar e instanciar um registro. Em seguida, ele imprime o nome do tipo e os valores de propriedade:

```
C#
```

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

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

}
```

O exemplo a seguir demonstra a igualdade de valor nos registros:

```
C#

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

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

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

O exemplo a seguir demonstra o uso de uma expressão `with` para copiar um objeto imutável e alterar uma das propriedades:

```
C#

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

public class Program
{
    public static void Main()
```

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

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

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

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

Para obter mais informações, confira Registros (referência de C#).

Especificação da linguagem C#

Para obter mais informações, consulte a Especificação da Linguagem C# . A especificação de idioma é a fonte definitiva para a sintaxe e o uso de C#.

Interfaces – definir comportamento para vários tipos

Artigo • 18/03/2023

Uma interface contém definições para um grupo de funcionalidades relacionadas que uma class ou um struct podem implementar. Uma interface pode definir métodos `static`, que devem ter uma implementação. Uma interface pode definir uma implementação padrão para membros. Uma interface pode não declarar dados de instância, como campos, propriedades implementadas automaticamente ou eventos semelhantes a propriedades.

Usando interfaces, você pode, por exemplo, incluir o comportamento de várias fontes em uma classe. Essa funcionalidade é importante em C# porque a linguagem não dá suporte a várias heranças de classes. Além disso, use uma interface se você deseja simular a herança para structs, pois eles não podem herdar de outro struct ou classe.

Você define uma interface usando a palavra-chave interface, como mostra o exemplo a seguir.

C#

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

O nome da interface deve ser um nome do identificador válido em C#. Por convenção, os nomes de interface começam com uma letra maiúscula `I`.

Qualquer classe ou struct que implemente a interface IEquatable<T> deve conter uma definição para um método Equals que corresponda à assinatura que a interface especifica. Como resultado, você pode contar com uma classe que implementa `IEquatable<T>` para conter um método `Equals` com o qual uma instância da classe pode determinar se é igual a outra instância da mesma classe.

A definição de `IEquatable<T>` não fornece uma implementação para `Equals`. Uma classe ou estrutura pode implementar várias interfaces, mas uma classe só pode herdar de uma única classe.

Para obter mais informações sobre classes abstratas, consulte Classes e membros de classes abstratos e lacrados.

As interfaces podem conter métodos, propriedades, eventos, indexadores ou qualquer combinação desses quatro tipos de membros. As interfaces podem conter construtores estáticos, campos, constantes ou operadores. A partir do C# 11, os membros da interface que não são campos podem ser `static abstract`. Uma interface não pode conter campos de instância, construtores de instância ou finalizadores. Os membros da interface são públicos por padrão, e você pode especificar explicitamente modificadores de acessibilidade, como `public`, `protected`, `internal`, `private`, `protected internal` ou `private protected`. Um membro `private` deve ter uma implementação padrão.

Para implementar um membro de interface, o membro correspondente da classe de implementação deve ser público, não estático e ter o mesmo nome e assinatura do membro de interface.

> ⓘ **Observação**
>
> Quando uma interface declara membros estáticos, um tipo que implementa essa interface também pode declarar membros estáticos com a mesma assinatura. Eles são distintos e identificados exclusivamente pelo tipo que declara o membro. O membro estático declarado em um tipo *não* substitui o membro estático declarado na interface.

Uma classe ou struct que implementa uma interface deve fornecer uma implementação para todos os membros declarados sem uma implementação padrão fornecida pela interface. No entanto, se uma classe base implementa uma interface, qualquer classe que é derivada da classe base herda essa implementação.

O exemplo a seguir mostra uma implementação da interface IEquatable<T>. A classe de implementação, `Car`, deverá fornecer uma implementação do método Equals.

C#

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

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

As propriedades e os indexadores de uma classe podem definir acessadores extras para uma propriedade ou o indexador que é definido em uma interface. Por exemplo, uma interface pode declarar uma propriedade que tem um acessador get. A classe que implementa a interface pode declarar a mesma propriedade tanto com um acessador `get` quanto com um set. No entanto, se a propriedade ou o indexador usa a implementação explícita, os acessadores devem corresponder. Para obter mais informações sobre a implementação explícita, consulte Implementação de interface explícita e Propriedades da interface.

Uma interface pode herdar de uma ou mais interfaces. A interface derivada herda os membros de suas interfaces base. Uma classe que implementa uma interface derivada deve implementar todos os membros na interface derivada, incluindo todos os membros das interfaces base da interface derivada. Essa classe pode ser convertida implicitamente para a interface derivada ou para qualquer uma de suas interfaces base. Uma classe pode incluir uma interface várias vezes por meio das classes base que ela herda ou por meio de interfaces que outras interfaces herdam. No entanto, a classe poderá fornecer uma implementação de uma interface apenas uma vez e somente se a classe declarar a interface como parte da definição de classe (`class ClassName : InterfaceName`). Se a interface é herdada porque é herdada de uma classe base que implementa a interface, a classe base fornece a implementação dos membros da interface. No entanto, a classe derivada pode reimplementar qualquer membro de interface virtual em vez de usar a implementação herdada. Quando as interfaces declaram uma implementação padrão de um método, qualquer classe que implemente essa interface herda essa implementação (você precisa converter a instância da classe para o tipo de interface para acessar a implementação padrão no membro da interface).

Uma classe base também pode implementar membros de interface usando membros virtuais. Nesse caso, uma classe derivada pode alterar o comportamento da interface substituindo os membros virtuais. Para obter mais informações sobre membros virtuais, consulte Polimorfismo.

Resumo de interfaces

Uma interface tem as propriedades a seguir:

- Nas versões do C# anteriores à 8.0, uma interface é como uma classe base abstrata que contém apenas membros abstratos. Qualquer classe ou struct que implemente uma interface deve implementar todos os seus membros.
- A partir do C# 8.0, uma interface pode definir implementações padrão para alguns ou todos os seus membros. Uma classe ou struct que implementa a interface não

precisa implementar membros que tenham implementações padrão. Para obter mais informações, consulte método de interface padrão.

- Uma interface não pode ser instanciada diretamente. Seus membros são implementados por qualquer classe ou struct que implemente a interface.
- Uma classe ou struct pode implementar várias interfaces. Uma classe pode herdar uma classe base e também implementar uma ou mais interfaces.

Classes e métodos genéricos

19/06/2025

Os genéricos introduzem o conceito de parâmetros de tipo ao .NET. Genéricos possibilitam o design de classes e métodos que adiam a especificação de um ou mais parâmetros de tipo até que você use a classe ou o método em seu código. Por exemplo, ao usar um parâmetro de tipo genérico `T`, você pode escrever uma única classe que outro código de cliente pode usar sem incorrer o custo ou risco de conversões de runtime ou operações de conversão boxing, conforme mostrado aqui:

C#

```
// Declare the generic class.
public class GenericList<T>
{
    public void Add(T item) { }
}

public class ExampleClass { }

class TestGenericList
{
    static void Main()
    {
        // Create a list of type int.
        GenericList<int> list1 = new();
        list1.Add(1);

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

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

Classes e métodos genéricos combinam reutilização, segurança de tipo e eficiência de uma maneira que seus equivalentes não genéricos não podem. Parâmetros de tipo genérico são substituídos pelos argumentos de tipo durante a compilação. No exemplo anterior, o compilador substitui `T` por `int`. Genéricos são usados mais frequentemente em coleções e nos métodos que operam sobre elas. O System.Collections.Generic namespace contém várias classes de coleção de base genérica. As coleções não genéricas, como ArrayList não são recomendadas e são mantidas apenas para fins de compatibilidade. Para saber mais, confira Genéricos no .NET.

Você também pode criar tipos e métodos genéricos personalizados para fornecer suas próprias soluções generalizadas e padrões de design que são seguros por tipo e eficientes. O exemplo de código a seguir mostra uma classe de lista vinculada genérica simples para fins de demonstração. (Na maioria dos casos, você deve usar a List<T> classe fornecida pelo .NET em vez de criar sua própria.) O parâmetro `T` de tipo é usado em vários locais em que um tipo concreto normalmente seria usado para indicar o tipo do item armazenado na lista:

- Como o tipo de um parâmetro de método no método `AddHead`.
- Como o tipo de retorno da propriedade `Data` na classe `Node` aninhada.
- Como o tipo de `data` do membro particular na classe aninhada.

`T` está disponível na classe aninhada `Node`. Quando `GenericList<T>` é instanciado com um tipo concreto, por exemplo, como um `GenericList<int>`, cada ocorrência de `T` é substituída por `int`.

C#

```
// Type parameter T in angle brackets.
public class GenericList<T>
{
    // The nested class is also generic, and
    // holds a data item of type T.
    private class Node(T t)
    {
        // T as property type.
        public T Data { get; set; } = t;

        public Node? Next { get; set; }
    }

    // First item in the linked list
    private Node? head;

    // T as parameter type.
    public void AddHead(T t)
    {
        Node n = new(t);
        n.Next = head;
        head = n;
    }

    // T in method return type.
    public IEnumerator<T> GetEnumerator()
    {
        Node? current = head;

        while (current is not null)
        {
            yield return current.Data;
            current = current.Next;
```

```
        }
    }
}
```

O exemplo de código a seguir mostra como o código do cliente usa a classe genérica `GenericList<T>` para criar uma lista de inteiros. Se você alterar o argumento de tipo, o código a seguir criará listas de cadeias de caracteres ou qualquer outro tipo personalizado:

C#

```
// A generic list of int.
GenericList<int> list = new();

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

// Write them to the console.
foreach (int i in list)
{
    Console.WriteLine(i);
}

Console.WriteLine("Done");
```

Observação

Tipos genéricos não se limitam a classes. Os exemplos anteriores usam os tipos `class`, mas você pode definir os tipos genéricos `interface` e `struct`, inclusive os tipos `record`.

Visão geral de genéricos

- Use tipos genéricos para maximizar a reutilização, a segurança do tipo e o desempenho do código.
- O uso mais comum de genéricos é criar classes de coleção.
- A biblioteca de classes do .NET contém várias classes de coleção genéricas no System.Collections.Generic namespace. As coleções genéricas devem ser usadas sempre que possível, em vez de classes como ArrayList no System.Collections namespace.
- Você pode criar suas próprias interfaces genéricas, classes, métodos, eventos e delegados.
- Classes genéricas podem ser restritas para habilitar o acesso a métodos em determinados tipos de dados.

- Você pode obter informações em tempo de execução sobre os tipos usados em um tipo de dados genérico usando reflexão.

Especificação da linguagem C#

Para obter mais informações, consulte a Especificação da Linguagem C# .

Consulte também

- Genéricos no .NET
- System.Collections.Generic

Tipos anônimos

Artigo • 23/11/2024

Os tipos anônimos fornecem um modo conveniente de encapsular um conjunto de propriedades somente leitura em um único objeto sem a necessidade de primeiro definir explicitamente um tipo. O nome do tipo é gerado pelo compilador e não está disponível no nível do código-fonte. O tipo de cada propriedade é inferido pelo compilador.

Crie tipos anônimos usando o operador new com um inicializador de objeto. Para obter mais informações sobre inicializadores de objeto, consulte Inicializadores de Objeto e Coleção.

O exemplo a seguir mostra um tipo anônimo que é inicializado com duas propriedades chamadas `Amount` e `Message`.

C#

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

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

Tipos anônimos normalmente são usados na cláusula select de uma expressão de consulta para retornar um subconjunto das propriedades de cada objeto na sequência de origem. Para obter mais informações sobre consultas, consulte LINQ no C#.

Os tipos anônimos contêm uma ou mais propriedades públicas somente leitura. Nenhum outro tipo de membros da classe, como métodos ou eventos, é válido. A expressão que é usada para inicializar uma propriedade não pode ser `null`, uma função anônima ou um tipo de ponteiro.

O cenário mais comum é inicializar um tipo anônimo com propriedades de outro tipo. No exemplo a seguir, suponha que existe uma classe com o nome `Product`. A classe `Product` inclui `Color` e `Price` propriedades, juntamente com outras propriedades nas quais você não está interessado:

C#

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

```
    public string? Name {get;set;}
    public string? Category {get;set;}
    public string? Size {get;set;}
}
```

A declaração do tipo anônimo começa com a palavra-chave `new`. A declaração inicializa um novo tipo que usa apenas duas propriedades de `Product`. O uso de tipos anônimos faz com que uma menor quantidade de dados seja retornada na consulta.

Quando você não especifica os nomes de membros no tipo anônimo, o compilador dá aos membros de tipo anônimo o mesmo nome da propriedade que está sendo usada para inicializá-los. Forneça um nome para a propriedade que está sendo inicializada com uma expressão, como mostrado no exemplo anterior. No exemplo a seguir, os nomes das propriedades do tipo anônimo são `Color` e `Price` . As instâncias são itens da `products` coleção de `Product` tipos:

C#

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

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

Dica

Você pode usar a regra de estilo .NET **IDE0037** para impor se os nomes de membros inferidos ou explícitos são preferenciais.

Também é possível definir um campo por objeto de outro tipo: classe, struct ou até mesmo outro tipo anônimo. Isso é feito usando a variável que contém esse objeto exatamente como no exemplo a seguir, onde dois tipos anônimos são criados usando tipos definidos pelo usuário já instanciados. Em ambos os casos, o `product` campo no tipo `shipment` anônimo e `shipmentWithBonus` será do tipo `Product` que contém seus valores padrão de cada campo. E o campo `bonus` será do tipo anônimo criado pelo compilador.

C#

```
var product = new Product();
var bonus = new { note = "You won!" };
```

```
var shipment = new { address = "Nowhere St.", product };
var shipmentWithBonus = new { address = "Somewhere St.", product, bonus };
```

Normalmente, ao usar um tipo anônimo para inicializar uma variável, a variável é declarada como uma variável local de tipo implícito usando var. O nome do tipo não pode ser especificado na declaração da variável, porque apenas o compilador tem acesso ao nome subjacente do tipo anônimo. Para obter mais informações sobre `var`, consulte Variáveis de local digitadas implicitamente.

Você pode criar uma matriz de elementos de tipo anônimo combinando uma variável local de tipo implícito e uma matriz de tipo implícito, como mostrado no exemplo a seguir.

C#

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

Os tipos anônimos são tipos class que derivam diretamente de object e que não podem ser convertidos para qualquer tipo, exceto para object. O compilador fornece um nome para cada tipo anônimo, embora o seu aplicativo não possa acessá-lo. Do ponto de vista do Common Language Runtime, um tipo anônimo não é diferente de qualquer outro tipo de referência.

Se dois ou mais inicializadores de objeto anônimos em um assembly especificarem uma sequência de propriedades que estão na mesma ordem e que têm os mesmos nomes e tipos, o compilador tratará os objetos como instâncias do mesmo tipo. Eles compartilham o mesmo tipo de informação gerado pelo compilador.

Tipos anônimos dão suporte a mutações não destrutivas na forma de expressões with. Isso permite que você crie uma nova instância de um tipo anônimo em que uma ou mais propriedades têm novos valores:

C#

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

Você não pode declarar que um campo, uma propriedade, um evento ou um tipo de retorno de um método tem um tipo anônimo. Da mesma forma, não pode declarar que um parâmetro formal de um método, propriedade, construtor ou indexador tem um tipo anônimo. Para passar um tipo anônimo ou uma coleção que contenha tipos

anônimos como um argumento para um método, você pode declarar o parâmetro como tipo `object`. No entanto, usar `object` para tipos anônimos anula o propósito da tipagem forte. Se você precisa armazenar os resultados da consulta ou passá-los fora do limite do método, considere o uso de uma estrutura ou classe com denominação comum em vez de um tipo anônimo.

Como os métodos Equals e GetHashCode em tipos anônimos são definidos em termos dos métodos das propriedades `Equals` e `GetHashCode`, duas instâncias do mesmo tipo anônimo são iguais somente se todas as suas propriedades forem iguais.

> ⓘ **Observação**
>
> O nível de acessibilidade de um tipo anônimo é `internal`, portanto, dois tipos anônimos definidos em assemblies diferentes não são do mesmo tipo. Portanto, instâncias de tipos anônimos não podem ser iguais umas às outras quando definidas em assemblies diferentes, mesmo quando todas as suas propriedades são iguais.

Tipos anônimos substituem o método ToString, concatenando o nome e a saída `ToString` de cada propriedade cercada por chaves.

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

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

Visão geral das técnicas orientadas a objetos em C#

Artigo • 21/04/2025

Em C#, a definição de um tipo — uma classe, struct ou registro — é como um blueprint que especifica o que o tipo pode fazer. Um objeto é basicamente um bloco de memória alocado e configurado de acordo com o blueprint. Este artigo fornece uma visão geral desses blueprints e seus recursos. O próximo artigo desta série apresenta objetos.

Encapsulamento

Às vezes, o *encapsulamento* é conhecido como o primeiro pilar ou princípio da programação orientada a objetos. Uma classe ou struct pode especificar o quão acessível cada um de seus membros é para codificar fora da classe ou struct. Os membros não destinados a consumidores fora da classe ou assembly são ocultados para limitar o potencial de erros de codificação ou explorações maliciosas. Para obter mais informações, consulte o tutorial de programação orientada a objetos.

Membros

Os *membros* de um tipo incluem todos os métodos, campos, constantes, propriedades e eventos. Em C#, não há variáveis ou métodos globais como há em algumas outras linguagens. Até mesmo o ponto de entrada de um programa, o `Main` método, deve ser declarado dentro de uma classe ou struct (implicitamente quando você usa instruções de nível superior).

A lista a seguir inclui todos os vários tipos de membros que podem ser declarados em uma classe, struct ou registro.

- Campos
- Constantes
- Propriedades
- Métodos
- Construtores
- Eventos
- Finalizadores
- Indexadores
- Operadores
- Tipos aninhados

Para obter mais informações, consulte Membros.

Acessibilidade

Alguns métodos e propriedades devem ser chamados ou acessados a partir de código fora de uma classe ou struct, que são conhecidos como *código do cliente*. Outros métodos e propriedades podem ser usados apenas na classe ou struct em si. É importante limitar a acessibilidade do código para que somente o código do cliente pretendido possa alcançá-lo. Especifique o quão acessíveis seus tipos e seus membros são para o código do cliente usando os seguintes modificadores de acesso:

- público
- Protegido
- interno
- internos protegidos
- privado
- privado protegido.

A acessibilidade padrão é `private`.

Herança

Classes (mas não structs) dão suporte ao conceito de herança. Uma classe derivada de outra classe, chamada *classe base*, contém automaticamente todos os membros públicos, protegidos e internos da classe base, exceto seus construtores e finalizadores.

As classes podem ser declaradas como abstratas, o que significa que um ou mais de seus métodos não têm implementação. Embora as classes abstratas não possam ser instanciadas diretamente, elas podem servir como classes base para outras classes que fornecem a implementação ausente. As classes também podem ser declaradas como lacradas para impedir que outras classes herdem delas.

Para obter mais informações, consulte Herança e Polimorfismo.

Interfaces

Classes, structs e registros podem implementar várias interfaces. Implementar a partir de uma interface significa que o tipo implementa todos os métodos definidos na interface. Para obter mais informações, consulte Interfaces.

Tipos genéricos

Classes, structs e registros podem ser definidos com um ou mais parâmetros de tipo. O código do cliente fornece o tipo quando cria uma instância do tipo. Por exemplo, a List<T> classe no System.Collections.Generic namespace é definida com um parâmetro de tipo. O código do cliente cria uma instância de um `List<string>` ou `List<int>` para especificar o tipo que a lista contém. Para obter mais informações, consulte Genéricos.

Tipos estáticos

Classes (mas não structs ou registros) podem ser declaradas como `static`. Uma classe estática pode conter apenas membros estáticos e não pode ser instanciada com a `new` palavra-chave. Uma cópia da classe é carregada na memória quando o programa é carregado e seus membros são acessados por meio do nome da classe. Classes, structs e registros podem conter membros estáticos. Para obter mais informações, consulte classes estáticas e membros de classe estática.

Tipos aninhados

Uma classe, um struct ou um registro podem ser aninhados em outra classe, struct ou registro. Para obter mais informações, consulte Tipos aninhados.

Tipos parciais

Você pode definir parte de uma classe, struct ou método em um arquivo de código e outra parte em um arquivo de código separado. Para obter mais informações, consulte Classes e métodos parciais.

Inicializadores de objeto

Você pode instanciar e inicializar objetos de classe ou struct e coleções de objetos, atribuindo valores às suas propriedades. Para obter mais informações, consulte Como inicializar objetos usando um inicializador de objeto.

Tipos anônimos

Em situações em que não é conveniente ou necessário criar uma classe nomeada, você usa tipos anônimos. Os membros de dados nomeados definem tipos anônimos. Para obter mais informações, consulte tipos anônimos.

Membros da Extensão

Você pode "estender" uma classe sem criar uma classe derivada criando um tipo separado. Esse tipo contém métodos que podem ser chamados como se pertencessem ao tipo original. Para obter mais informações, consulte Métodos de extensão.

Variáveis Locais Tipadas Implicitamente

Em uma classe ou método struct, você pode usar a digitação implícita para instruir o compilador a determinar o tipo de uma variável no tempo de compilação. Para obter mais informações, consulte var (referência de C#).

Arquivos

Você pode adicionar o `record` modificador a uma classe ou a um struct. Os registros são tipos com comportamento interno para igualdade baseada em valor. Um registro (seja `record class` ou `record struct`) fornece os seguintes recursos:

- Sintaxe concisa para criar um tipo de referência com propriedades imutáveis.
- Igualdade de valor. Duas variáveis de um tipo de registro são iguais se tiverem o mesmo tipo e se, para cada campo, os valores em ambos os registros forem iguais. As classes usam igualdade de referência: duas variáveis de um tipo de classe são iguais se se referirem ao mesmo objeto.
- Sintaxe concisa para mutação não estruturativa. Uma `with` expressão permite que você crie uma nova instância de registro que seja uma cópia de uma instância existente, mas com valores de propriedade especificados alterados.
- Formatação embutida para exibição. O `ToString` método imprime o nome do tipo de registro e os nomes e valores das propriedades públicas.
- Suporte para hierarquias de herança em classes de registro. Classes de registro dão suporte à herança. Os structs de registro não dão suporte à herança.

Para saber mais, confira Registros.

Especificação da linguagem C#

Para obter mais informações, consulte a Especificação da Linguagem C# . A especificação de idioma é a fonte definitiva para a sintaxe e o uso de C#.

Objetos – criar instâncias de tipos

Artigo • 08/04/2025

Uma definição de classe ou struct é como um blueprint que especifica o que o tipo pode fazer. Um objeto é basicamente um bloco de memória que foi alocado e configurado de acordo com o blueprint. Um programa pode criar muitos objetos da mesma classe. Os objetos também são chamados de instâncias e podem ser armazenados em uma variável nomeada ou em uma matriz ou coleção. Código do cliente é o código que usa essas variáveis para chamar os métodos e acessar as propriedades públicas do objeto. Em uma linguagem orientada a objetos, como C#, um programa típico consiste em vários objetos interagindo dinamicamente.

> ⓘ **Observação**
>
> Os tipos estáticos se comportam de forma diferente do descrito aqui. Para obter mais informações, consulte Classes estáticas e membros de classe estática.

Instâncias Struct vs. Instâncias de classe

Como as classes são tipos de referência, uma variável de um objeto de classe contém uma referência ao endereço do objeto no heap gerenciado. Se uma segunda variável do mesmo tipo for atribuída à primeira variável, ambas as variáveis se referirão ao objeto nesse endereço. Este ponto é discutido com mais detalhes mais adiante neste artigo.

As instâncias de classes são criadas usando o new operador. No exemplo a seguir, `Person` é o tipo e `person1` e `person2` são instâncias ou objetos desse tipo.

C#

```
using System;

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

Como os structs são tipos de valor, uma variável de um objeto struct contém uma cópia de todo o objeto. Instâncias de structs também podem ser criadas usando o `new` operador, mas isso não é necessário, conforme mostrado no exemplo a seguir:

C#

```
using System;

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

    public class Application
    {
        static void Main()
        {
            // Create  struct instance and initialize by using "new".
            // Memory is allocated on thread stack.
            Person p1 = new Person("Alex", 9);
            Console.WriteLine($"p1 Name = {p1.Name} Age = {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 = {p2.Name} Age = {p2.Age}");

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

A memória de `p1` e `p2` é alocada na pilha de thread. Essa memória é recuperada junto com o tipo ou método no qual ela é declarada. Esse é um dos motivos pelos quais os structs são copiados na atribuição. Por outro lado, a memória alocada a uma instância de classe é recuperada automaticamente (o lixo é coletado) pelo Common Language Runtime quando todas as referências ao objeto tiveram saído do escopo. Não é possível destruir deterministicamente um objeto de classe como você pode no C++. Para obter mais informações sobre a coleta de lixo no .NET, consulte Coleta de Lixo.

> ⓘ **Observação**
>
> A alocação e a desalocação de memória no heap gerenciado é altamente otimizada no Common Language Runtime. Na maioria dos casos, não há uma diferença significativa quanto ao custo do desempenho de alocar uma instância da classe no heap em vez de alocar uma instância de struct na pilha.

Identidade do Objeto vs. Igualdade de Valor

Ao comparar dois objetos para igualdade, primeiro você deve distinguir se deseja saber se as duas variáveis representam o mesmo objeto na memória ou se os valores de um ou mais de seus campos são equivalentes. Se você pretende comparar valores, deve considerar se os objetos são instâncias de tipos de valor (structs) ou tipos de referência (classes, delegados, matrizes).

- Para determinar se duas instâncias de classe se referem ao mesmo local na memória (o que significa que elas têm a mesma *identidade*), use o método estático Object.Equals . (System.Object é a classe base implícita para todos os tipos de valor e tipos de referência, incluindo structs e classes definidos pelo usuário.)

- Para determinar se os campos de instância em duas instâncias de struct têm os mesmos valores, use o ValueType.Equals método. Como todos os structs herdam implicitamente de System.ValueType, você chama o método diretamente em seu objeto, conforme mostrado no exemplo a seguir:

  C#

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

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

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

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

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

  A implementaçãoSystem.ValueType de `Equals` usa conversão boxing e reflexão em alguns casos. Para obter informações sobre como fornecer um algoritmo de igualdade eficiente específico ao seu tipo, consulte Como definir a igualdade de valor para um tipo. Os registros são tipos de referência que usam semântica de valor para igualdade.

- Para determinar se os valores dos campos em duas instâncias de classe são iguais, você pode usar o Equals método ou o operador == . No entanto, use-os apenas se a classe os tiver substituído ou sobrecarregado para fornecer uma definição personalizada do que "igualdade" significa para objetos desse tipo. A classe também pode implementar a IEquatable<T> interface ou a IEqualityComparer<T> interface. Ambas as interfaces fornecem métodos que podem ser usados para testar a igualdade de valor. Ao projetar

suas próprias classes que substituem `Equals`, siga as diretrizes declaradas em Como definir a igualdade de valor para um tipo e Object.Equals(Object).

Seções relacionadas

Para obter mais informações:

- Classes
- Construtores
- Finalizadores
- Eventos
- objeto
- Herança
- classe
- Tipos de estrutura
- novo operador
- Sistema de Tipo Comum

Herança – derivar tipos para criar um comportamento mais especializado

19/06/2025

A herança, juntamente com o encapsulamento e o polimorfismo, é uma das três características principais da programação orientada a objetos. A herança permite que você crie novas classes que reutilizem, estendam e modifiquem o comportamento definido em outras classes. A classe cujos membros são herdados é chamada *de classe base* e a classe que herda esses membros é chamada *de classe derivada*. Uma classe derivada pode ter apenas uma classe base direta. No entanto, a herança é transitiva. Se `ClassC` for derivado de `ClassB`, e `ClassB` for derivado de `ClassA`, `ClassC` herdará os membros declarados em `ClassB` e `ClassA`.

> ⓘ **Observação**
>
> Os structs não dão suporte à herança, mas podem implementar interfaces.

Conceitualmente, uma classe derivada é uma especialização da classe base. Por exemplo, se você tiver uma classe `Animal` base, poderá ter uma classe derivada nomeada `Mammal` e outra classe derivada nomeada `Reptile`. A `Mammal` é um `Animal`, e um `Reptile` é um `Animal`, mas cada classe derivada representa diferentes especializações da classe base.

As declarações de interface podem definir uma implementação padrão para seus membros. Essas implementações são herdadas por interfaces derivadas e por classes que implementam essas interfaces. Para obter mais informações sobre métodos de interface padrão, consulte o artigo sobre interfaces.

Quando você define uma classe para derivar de outra classe, a classe derivada obtém implicitamente todos os membros da classe base, exceto seus construtores e finalizadores. A classe derivada reutiliza o código na classe base sem precisar reimplementá-lo. Você pode adicionar mais membros na classe derivada. A classe derivada estende a funcionalidade da classe base.

A ilustração a seguir mostra uma classe `WorkItem` que representa um item de trabalho em algum processo de negócios. Como todas as classes, ela deriva de System.Object e herda todos os seus métodos. `WorkItem` adiciona seis membros próprios. Esses membros incluem um construtor, pois os construtores não são herdados. A classe `ChangeRequest` herda `WorkItem` e representa um tipo específico de item de trabalho. `ChangeRequest` adiciona mais dois membros aos que herda de `WorkItem` e de Object. Ele deve adicionar seu próprio construtor e também

adiciona `originalItemID`. A propriedade `originalItemID` permite que a `ChangeRequest` instância seja associada ao original `WorkItem` ao qual a solicitação de alteração se aplica.

O exemplo a seguir mostra como as relações de classe demonstradas na ilustração anterior são expressas em C#. O exemplo também mostra como `WorkItem` substitui o método Object.ToStringvirtual e como a `ChangeRequest` classe herda a `WorkItem` implementação do método. O primeiro bloco define as classes:

C#

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

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

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

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

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

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

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

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

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

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

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

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

```
        this.jobLength = jobLen;

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

Este próximo bloco mostra como usar as classes base e derivadas:

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

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

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

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

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

Métodos abstratos e virtuais

Quando uma classe base declara um método como virtual, uma classe derivada pode override o método com sua própria implementação. Se uma classe base declarar um membro como abstract, esse método deverá ser substituído em qualquer classe não abstrata que herda diretamente dessa classe. Se uma classe derivada for abstrata, ela herdará membros abstratos sem implementá-los. Membros abstratos e virtuais são a base para o polimorfismo, que é a

segunda característica primária da programação orientada a objetos. Para obter mais informações, consulte Polymorphism.

Classes base abstratas

Você pode declarar uma classe como abstrata se quiser impedir a instanciação direta usando o novo operador. Uma classe abstrata só poderá ser usada se uma nova classe for derivada dela. Uma classe abstrata pode conter uma ou mais assinaturas de método que são declaradas como abstratas. Essas assinaturas especificam os parâmetros e o valor retornado, mas não têm nenhuma implementação (corpo do método). Uma classe abstrata não precisa conter membros abstratos; no entanto, se uma classe contiver um membro abstrato, a própria classe deverá ser declarada como abstrata. Classes derivadas que não são abstratas devem fornecer a implementação para quaisquer métodos abstratos de uma classe base abstrata.

Interfaces

Uma *interface* é um tipo de referência que define um conjunto de membros. Todas as classes e structs que implementam essa interface devem implementar esse conjunto de membros. Uma interface pode definir uma implementação padrão para qualquer um ou todos esses membros. Uma classe pode implementar várias interfaces, embora possa derivar apenas de uma única classe base direta.

Interfaces são usadas para definir recursos específicos para classes que não têm necessariamente uma relação do tipo “é um”. Por exemplo, a System.IEquatable<T> interface pode ser implementada por qualquer classe ou struct para determinar se dois objetos do tipo são equivalentes (no entanto, o tipo define equivalência). IEquatable<T> não implica o mesmo tipo de relação "é um" que existe entre uma classe base e uma classe derivada (por exemplo, um `Mammal` é um `Animal`). Para obter mais informações, consulte Interfaces.

Impedindo a derivação adicional

Uma classe pode impedir que outras classes herdem dela, ou de qualquer um de seus membros, declarando-se ou o membro como sealed.

Ocultação de membros da classe base pela classe derivada

Uma classe derivada pode ocultar membros da classe base declarando membros com o mesmo nome e assinatura. O modificador new pode ser usado para indicar explicitamente que

o membro não pretende ser uma substituição do membro base. O uso de new não é necessário, mas um aviso do compilador será gerado se new não for usado. Para obter mais informações, consulte Versioning com as palavras-chave Override e New e Conhecendo Quando Usar as Palavras-Chave Override e New.

Polimorfismo

Artigo • 07/04/2023

O polimorfismo costuma ser chamado de o terceiro pilar da programação orientada a objetos, depois do encapsulamento e a herança. O polimorfismo é uma palavra grega que significa "de muitas formas" e tem dois aspectos distintos:

- Em tempo de execução, os objetos de uma classe derivada podem ser tratados como objetos de uma classe base, em locais como parâmetros de método, coleções e matrizes. Quando esse polimorfismo ocorre, o tipo declarado do objeto não é mais idêntico ao seu tipo de tempo de runtime.
- As classes base podem definir e implementar métodos*virtuais* e as classes derivadas podem substituí-los, o que significa que elas fornecem sua própria definição e implementação. Em tempo de execução, quando o código do cliente chama o método, o CLR procura o tipo de tempo de execução do objeto e invoca a substituição do método virtual. No código-fonte, você pode chamar um método em uma classe base e fazer com que a versão de uma classe derivada do método seja executada.

Os métodos virtuais permitem que você trabalhe com grupos de objetos relacionados de maneira uniforme. Por exemplo, suponha que você tem um aplicativo de desenho que permite que um usuário crie vários tipos de formas sobre uma superfície de desenho. Você não sabe no tempo de compilação quais tipos específicos de formas o usuário criará. No entanto, o aplicativo precisa manter controle de todos os diferentes tipos de formas que são criados e atualizá-los em resposta às ações do mouse do usuário. Você pode usar o polimorfismo para resolver esse problema em duas etapas básicas:

1. Crie uma hierarquia de classes em que cada classe de forma específica derive de uma classe base comum.
2. Use um método virtual para invocar o método adequado em qualquer classe derivada por meio de uma única chamada para o método da classe base.

Primeiro, crie uma classe base chamada `Shape` e as classes derivadas como `Rectangle`, `Circle` e `Triangle`. Atribua à classe `Shape` um método virtual chamado `Draw` e substitua-o em cada classe derivada para desenhar a forma especial que a classe representa. Crie um objeto `List<Shape>` e adicione um `Circle`, `Triangle` e `Rectangle` a ele.

```
C#
```

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

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

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

Para atualizar a superfície de desenho, use um loop foreach para iterar na lista e chamar o método `Draw` em cada objeto `Shape` na lista. Mesmo que cada objeto na lista tenha um tipo declarado `Shape`, é o tipo de runtime (a versão de substituição do método em cada classe derivada) que será invocado.

```
C#
```

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

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

Em C#, cada tipo é polimórfico porque todos os tipos, incluindo tipos definidos pelo usuário, herdam de Object.

Visão geral sobre o polimorfismo

Membros virtuais

Quando uma classe derivada herda de uma classe base, ela inclui todos os membros da classe base. Todo o comportamento declarado na classe base faz parte da classe derivada. Isso permite que objetos da classe derivada sejam tratados como objetos da classe base. Os modificadores de acesso (`public`, `protected` `private` e assim por diante) determinam se esses membros estão acessíveis a partir da implementação de classe derivada. Os métodos virtuais dão ao designer opções diferentes para o comportamento da classe derivada:

- A classe derivada pode substituir membros virtuais na classe base, definindo o novo comportamento.
- A classe derivada pode herdar o método de classe base mais próximo sem substituí-lo, preservando o comportamento existente, mas permitindo que outras

classes derivadas substituam o método.

- A classe derivada pode definir uma nova implementação não virtual desses membros que ocultam as implementações de classe base.

Uma classe derivada poderá substituir um membro de classe base somente se o membro da classe base tiver sido declarado como virtual ou abstrato. O membro derivado deve usar a palavra-chave override para indicar explicitamente que o método destina-se a participar da invocação virtual. O código a seguir mostra um exemplo:

C#

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

Os campos não podem ser virtuais, apenas os métodos, as propriedades, os eventos e os indexadores podem ser virtuais. Quando uma classe derivada substitui um membro virtual, esse membro é chamado, mesmo quando uma instância dessa classe está sendo acessada como uma instância da classe base. O código a seguir mostra um exemplo:

C#

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

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

Os métodos e propriedades virtuais permitem que classes derivadas estendam uma classe base sem a necessidade de usar a implementação da classe base de um método. Para obter mais informações, consulte Controle de versão com as palavras-chave override e new. Uma interface fornece uma outra maneira de definir um método ou conjunto de métodos cuja implementação é deixada para classes derivadas.

Ocultar membros da classe base com novos membros

Se quiser que sua classe derivada tenha um membro com o mesmo nome que um membro em uma classe base, você poderá usar a palavra-chave new para ocultar o membro da classe base. A palavra-chave `new` é colocada antes do tipo de retorno de um membro de classe que está sendo substituído. O código a seguir mostra um exemplo:

C#

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

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

Os membros da classe base oculta podem ser acessados do código do cliente, por meio da seleção da instância da classe derivada em uma instância da classe base. Por exemplo:

C#

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

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

Impedir que classes derivadas substituam membros virtuais

Os membros virtuais permanecem virtuais, independentemente de quantas classes foram declaradas entre o membro virtual e a classe que o declarou originalmente. Se a classe `A` declara um membro virtual, a classe `B` deriva de `A` e a classe `C` deriva de `B`, a classe `C` herda o membro virtual e tem a opção de substituí-lo, independentemente de a classe `B` ter declarado uma substituição para esse membro. O código a seguir mostra um exemplo:

C#

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

Uma classe derivada pode interromper a herança virtual, declarando uma substituição como sealed. Isso exige a colocação da palavra-chave `sealed` antes da palavra-chave `override` na declaração de membro de classe. O código a seguir mostra um exemplo:

C#

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

No exemplo anterior, o método `DoWork` não é mais virtual para nenhuma classe derivada de `C`. Ele ainda é virtual para as instâncias de `C`, mesmo se elas foram convertidas para o tipo `B` ou tipo `A`. Os métodos lacrados podem ser substituídos por classes derivadas usando a palavra-chave `new`, como mostra o exemplo a seguir:

C#

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

Neste caso, se `DoWork` é chamado em `D` usando uma variável do tipo `D`, o novo `DoWork` é chamado. Se uma variável do tipo `C`, `B` ou `A` é usada para acessar uma instância de `D`,

uma chamada de `DoWork` seguirá as regras de herança virtual, encaminhando as chamadas para a implementação de `DoWork` na classe `C`.

Acessar membros virtuais da classe base de classes derivadas

A classe derivada que substituiu um método ou propriedade ainda pode acessar o método ou propriedade na classe base usando a palavra-chave `base`. O código a seguir mostra um exemplo:

C#

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

Para obter mais informações, consulte base.

> **Observação**
>
> Recomendamos que os membros virtuais usem `base` para chamar a implementação da classe base do membro em sua própria implementação. Deixar o comportamento da classe base ocorrer permite que a classe derivada se concentre na implementação de comportamento específico para a classe derivada. Se a implementação da classe base não é chamado, cabe à classe derivada tornar seu comportamento compatível com o comportamento da classe base.

Visão geral dos padrões correspondentes

Artigo • 27/01/2025

A *correspondência de padrões* é uma técnica em que você testa uma expressão para determinar se ela tem determinadas características. A correspondência de padrões C# fornece uma sintaxe mais concisa para testar expressões e tomar medidas quando uma expressão corresponde. A "expressão is" dá suporte à correspondência de padrões para testar uma expressão e declarar condicionalmente uma nova variável para o resultado dessa expressão. A "expressão switch" permite que você execute ações com base no padrão de primeira correspondência de uma expressão. Essas duas expressões dão suporte a um vocabulário avançado de *padrões*.

Este artigo fornece uma visão geral dos cenários em que você pode usar a correspondência de padrões. Essas técnicas podem melhorar a legibilidade e a correção do código. Para obter uma discussão completa sobre todos os padrões que você pode aplicar, consulte o artigo sobre padrões na referência de linguagem.

Verificações nulas

Um dos cenários mais comuns para correspondência de padrões é garantir que os valores não sejam `null`. Você pode testar e converter um tipo de valor anulável em seu tipo subjacente durante o teste de `null` usando o seguinte exemplo:

C#

```
int? maybe = 12;

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

O código anterior é um *padrão de declaração* para testar o tipo da variável e atribuí-lo a uma nova variável. As regras de idioma tornam essa técnica mais segura do que muitas outras. A variável `number` só é acessível e atribuída na parte verdadeira da cláusula `if`. Se você tentar acessá-la em outro lugar, na cláusula `else` ou após o bloco `if`, o

compilador emitirá um erro. Em segundo lugar, como você não está usando o operador `==`, esse padrão funciona quando um tipo sobrecarrega o operador `==`. Isso o torna uma maneira ideal de verificar os valores de referência nulos, adicionando o padrão `not`:

C#

```
string? message = ReadMessageOrDefault();

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

O exemplo anterior usou um *padrão constante* para comparar a variável com `null`. `not` é um *padrão lógico* que corresponde quando o padrão negado não corresponde.

Testes de tipo

Outro uso comum para correspondência de padrões é testar uma variável para ver se ela corresponde a um determinado tipo. Por exemplo, o código a seguir testa se uma variável não é nula e implementa a interface System.Collections.Generic.IList<T>. Se isso acontecer, ele usará a propriedade ICollection<T>.Count nessa lista para localizar o índice do meio. O padrão de declaração não corresponde a um valor `null`, independentemente do tipo de tempo de compilação da variável. O código abaixo protege contra `null` e um tipo que não implementa `IList`.

C#

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

```
    }
}
```

Os mesmos testes podem ser aplicados em uma expressão `switch` para testar uma variável em vários tipos diferentes. Você pode usar essas informações para criar algoritmos melhores com base no tipo em tempo de execução específico.

Comparar valores discretos

Você também pode testar uma variável para encontrar uma correspondência em valores específicos. O código a seguir mostra um exemplo em que você testa um valor em relação a todos os valores possíveis declarados em uma enumeração:

C#

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

O exemplo anterior demonstra uma expedição de método com base no valor de uma enumeração. O caso `_` final é um *padrão de descarte* que corresponde a todos os valores. Ele identifica quaisquer condições de erro em que o valor não corresponda a um dos valores `enum` definidos. Se você omitir esse braço de comutador, o compilador avisará que sua expressão padrão não manipula todos os valores de entrada possíveis. Em tempo de execução, a expressão `switch` lançará uma exceção se o objeto que está sendo examinado não corresponder a nenhum braço switch. Você pode usar constantes numéricas em vez de um conjunto de valores de enumeração. Você também pode usar essa técnica semelhante para valores de cadeia de caracteres constantes que representam os comandos:

C#

```
public State PerformOperation(string command) =>
   command switch
   {
       "SystemTest" => RunDiagnostics(),
       "Start" => StartSystem(),
       "Stop" => StopSystem(),
```

```
        "Reset" => ResetToReady(),
        _ => throw new ArgumentException("Invalid string value for command",
nameof(command)),
    };
```

O exemplo anterior mostra o mesmo algoritmo, mas usa valores de cadeia de caracteres em vez de uma enumeração. Você usaria esse cenário se seu aplicativo respondesse a comandos de texto em vez de um formato de dados regular. A partir do C# 11, você também pode usar um `Span<char>` ou `ReadOnlySpan<char>` para testar valores de cadeia de caracteres constantes, conforme mostrado no exemplo a seguir:

C#

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

Em todos esses exemplos, o *padrão de descarte* garante que você identifique todas as entradas. O compilador ajuda você a garantir que todos os valores de entrada possíveis sejam identificados.

Padrões relacionais

Os *padrões relacionais* podem ser usados para testar como um valor se compara às constantes. Por exemplo, o código a seguir retorna o estado da água com base na temperatura em Fahrenheit:

C#

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

O código anterior também demonstra o `and` conjuntivo para verificar se ambos os padrões relacionais correspondem. Você também pode usar um padrão disjuntivo `or` para verificar se um dos padrões corresponde. Os dois padrões relacionais ficam entre parênteses, que podem ser usados em qualquer padrão para maior clareza. Os dois últimos braços switch identificam os casos para o ponto de fusão e o ponto de ebulição. Sem esses dois braços, o compilador avisa que sua lógica não cobre todas as entradas possíveis.

O código anterior também demonstra outro recurso importante que o compilador fornece para expressões de correspondência de padrões: o compilador avisa se você não identifica todos os valores de entrada. O compilador também emitirá um aviso se o padrão de um braço de comutador for coberto por um padrão anterior. Isso lhe dá liberdade para refatorar e reordenar as expressões de alternância. Outra maneira de gravar a mesma expressão pode ser:

C#

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

A lição principal no exemplo anterior e qualquer outra refatoração ou reordenação é que o compilador valida que seu código manipula todas as entradas possíveis.

Várias entradas

Todos os padrões vistos até agora foram para verificar uma entrada. Você pode gravar padrões que examinam várias propriedades de um objeto. Considere o registro `Order` a seguir:

C#

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

O tipo de registro posicional anterior declara dois membros em posições explícitas. Primeiro, `Items` aparece e, em seguida, o `Cost` da ordem. Para saber mais, confira Registros.

O código a seguir examina o número de itens e o valor de uma ordem para calcular um preço com desconto:

```
C#

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

Os dois primeiros braços examinam duas propriedades do `Order`. O terceiro examina apenas o custo. O próximo verifica `null` e o final corresponde a qualquer outro valor. Se o tipo `Order` definir um método Deconstruct adequado, você poderá omitir os nomes de propriedade do padrão e usar a desconstrução para examinar as propriedades:

```
C#

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

O código anterior demonstra o *padrão posicional* em que as propriedades são desconstruídas para a expressão.

Você também pode comparar uma propriedade com `{ }`, que coincide com qualquer valor não nulo. Considere a seguinte declaração, que armazena medidas com uma anotação opcional:

```
C#

public record class Observation(int Value, string Units, string Name)
{
    public string? Annotation { get; set; }
}
```

Você pode testar se uma determinada observação tem uma anotação não nula usando a seguinte expressão de correspondência de padrões:

C#

```
if (observation.Annotation is { })
{
    Console.WriteLine($"Observation description: {observation.Annotation}");
}
```

Padrões de lista

Você pode verificar os elementos em uma lista ou uma matriz usando um *padrão de lista*. Um padrão de lista fornece um meio para aplicar um padrão a qualquer elemento de uma sequência. Além disso, você pode aplicar o *padrão de descarte* (`_`) para corresponder a qualquer elemento ou aplicar um *padrão de fatia* para corresponder a zero ou mais elementos.

Os padrões de lista são uma ferramenta valiosa quando os dados não seguem uma estrutura regular. Você pode usar a correspondência de padrões para testar a forma e os valores dos dados em vez de transformá-los em um conjunto de objetos.

Considere o seguinte trecho de um arquivo de texto que contém transações bancárias:

Saída

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

É um formato CSV, mas algumas das linhas têm mais colunas do que outras. Pior ainda para o processamento, uma coluna do tipo `WITHDRAWAL` tem texto gerado pelo usuário e pode conter uma vírgula no texto. Um *padrão de lista* que inclui o padrão *descarte*, o padrão *constante* e o padrão *var* para capturar os dados de processos de valor neste formato:

C#

```
decimal balance = 0m;
foreach (string[] transaction in ReadRecords())
```

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

O exemplo anterior usa uma matriz de cadeia de caracteres, em que cada elemento é um campo na linha. As teclas de expressão `switch` no segundo campo, que determina o tipo de transação, e o número de colunas restantes. Cada linha garante que os dados estão no formato correto. O padrão de descarte (`_`) ignora o primeiro campo, com a data da transação. O segundo campo corresponde ao tipo de transação. As correspondências de elemento restantes pulam para o campo com a quantidade. A partida final usa o padrão *var* para capturar a representação de cadeia de caracteres do valor. A expressão calcula o valor a ser adicionado ou subtraído do saldo.

Os padrões de lista permitem que você corresponda na forma de uma sequência de elementos de dados. Você usa os padrões de *descarte* e *fatia* para corresponder ao local dos elementos. Você usa outros padrões para corresponder às características sobre elementos individuais.

Este artigo fez um tour pelos tipos de código que você pode gravar com correspondência de padrões em C#. Os artigos a seguir mostram mais exemplos de uso dos padrões em cenários e o vocabulário completo dos padrões disponíveis para uso.

Confira também

- Usar padrões correspondentes para evitar a verificação 'is' seguida por uma conversão (regras de estilo IDE0020 e IDE0038)
- Exploração: usar a correspondência de padrões para criar seu comportamento de classe para obter um código melhor
- Tutorial: Usar padrões correspondentes para criar algoritmos controlados por tipo e controlados por dados
- Referência: Correspondência de padrões

Descartes – conceitos básicos do C#

19/06/2025

Descartes são variáveis de espaço reservado intencionalmente não utilizadas no código do aplicativo. Os descartes são equivalentes a variáveis não atribuídas; eles não têm um valor. Um descarte comunica a intenção para o compilador e outras pessoas que leem seu código: você pretendia ignorar o resultado de uma expressão. Talvez você queira ignorar o resultado de uma expressão, um ou mais membros de uma expressão de tupla, um parâmetro `out` para um método ou o destino de uma expressão de correspondência de padrões.

Os descartes tornam a intenção do seu código clara. Um descarte indica que nosso código nunca usa a variável. Eles melhoram sua legibilidade e manutenibilidade.

Você indica que uma variável é um descarte atribuindo-lhe o sublinhado (`_`) como seu nome. Por exemplo, a chamada de método a seguir retorna uma tupla na qual o primeiro e o segundo valores são descartes. `area` é uma variável declarada anteriormente definida como o terceiro componente retornado por `GetCityInformation`:

```
C#

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

Você pode usar descartes para especificar parâmetros de entrada não utilizados de uma expressão lambda. Para obter mais informações, consulte os parâmetros de entrada de uma seção de expressão lambda do artigo expressões Lambda .

Quando `_` é um descarte válido, tentar recuperar seu valor ou usá-lo em uma operação de atribuição gera o erro do compilador CS0103, "O nome '_' não existe no contexto atual". Esse erro ocorre porque `_` não é atribuído um valor e pode até não ser atribuído um local de armazenamento. Se fosse uma variável real, você não poderia descartar mais de um valor, como o exemplo anterior.

Desconstrução de objeto e de tupla

Descartes são úteis para trabalhar com tuplas quando seu código de aplicativo usa alguns elementos da tupla, mas ignora outros. Por exemplo, o método a seguir `QueryCityDataForYears` retorna uma tupla com o nome de uma cidade, sua área, um ano, a população da cidade para esse ano, um segundo ano e a população da cidade para esse segundo ano. O exemplo mostra a alteração na população entre esses dois anos. Entre os dados disponíveis da tupla, não estamos preocupados com a área da cidade e sabemos o

nome da cidade e as duas datas em tempo de design. Como resultado, estamos interessados apenas nos dois valores de população armazenados na tupla e podemos lidar com seus valores restantes como descartes.

C#

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

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

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

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

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

Para obter mais informações sobre desconstruir tuplas com descartes, consulte Desconstruindo tuplas e outros tipos.

O `Deconstruct` método de uma classe, estrutura ou interface também permite recuperar e desconstruir um conjunto específico de dados de um objeto. Você poderá usar descartes quando estiver interessado em trabalhar com apenas um subconjunto dos valores desconstruídos. O exemplo a seguir desconstrói um `Person` objeto em quatro cadeias de caracteres (o primeiro e os sobrenomes, a cidade e o estado), mas descarta o sobrenome e o estado.

C#

```
using System;
```

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

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

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

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

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

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

```
        }
    }
}
```

Para obter mais informações sobre desconstruir tipos definidos pelo usuário com descartes, consulte Desconstruindo tuplas e outros tipos.

Correspondência de padrões com `switch`

O *padrão de descarte* pode ser usado na correspondência de padrões com a expressão switch. Cada expressão, inclusive `null`, sempre corresponde ao padrão de descarte.

O exemplo a seguir define um `ProvidesFormatInfo` método que usa uma `switch` expressão para determinar se um objeto fornece uma IFormatProvider implementação e testa se o objeto é `null`. Ele também usa o padrão de descarte para lidar com objetos não nulos de qualquer outro tipo.

C#

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

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

Chamadas para métodos com parâmetros `out`

Ao chamar o `Deconstruct` método para desconstruir um tipo definido pelo usuário (uma instância de uma classe, estrutura ou interface), você pode descartar os valores de argumentos

individuais `out` . Mas você também pode descartar o valor dos `out` argumentos ao chamar qualquer método com um `out` parâmetro.

O exemplo a seguir chama o método DateTime.TryParse(String, out DateTime) para determinar se a representação de cadeia de caracteres de uma data é válida na cultura atual. Como o exemplo se preocupa apenas com a validação da cadeia de caracteres de data e não com a análise dela para extrair a data, o `out` argumento para o método é um descarte.

C#

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

Um descarte autônomo

Você pode usar um descarte autônomo para indicar qualquer variável que você optar por ignorar. Um uso típico é usar uma atribuição para garantir que um argumento não seja nulo. O código a seguir usa um descarte para forçar uma atribuição. O lado direito da atribuição usa o operador de avaliação de nulo para lançar um System.ArgumentNullException quando o argumento é `null`. O código não precisa do resultado da atribuição, portanto, ele é descartado. A expressão força uma verificação de nulo. O descarte esclarece sua intenção: o resultado da atribuição não é necessário ou usado.

C#

```
public static void Method(string arg)
{
    _ = arg ?? throw new ArgumentNullException(paramName: nameof(arg), message:
```

```
"arg can't be null");

    // Do work with arg.
}
```

O exemplo a seguir usa um descarte autônomo para ignorar o Task objeto retornado por uma operação assíncrona. Atribuir a tarefa tem o efeito de suprimir a exceção que a operação gera quando está prestes a ser concluída. Isso deixa sua intenção clara: você deseja descartar o `Task`, e ignorar todos os erros gerados a partir dessa operação assíncrona.

C#

```
private static async Task ExecuteAsyncMethods()
{
    Console.WriteLine("About to launch a task...");
    _ = Task.Run(() =>
    {
        var iterations = 0;
        for (int ctr = 0; ctr < int.MaxValue; ctr++)
            iterations++;
        Console.WriteLine("Completed looping operation...");
        throw new InvalidOperationException();
    });
    await Task.Delay(5000);
    Console.WriteLine("Exiting after 5 second delay");
}
// The example displays output like the following:
//       About to launch a task...
//       Completed looping operation...
//       Exiting after 5 second delay
```

Sem atribuir a tarefa a um descarte, o código a seguir gera um aviso do compilador:

C#

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

```
    await Task.Delay(5000);
    Console.WriteLine("Exiting after 5 second delay");
```

> ⓘ **Observação**
>
> Se você executar um dos dois exemplos anteriores usando um depurador, o depurador interromperá o programa quando a exceção for lançada. Sem um depurador anexado, a exceção é silenciosamente ignorada em ambos os casos.

`_` também é um identificador válido. Quando usado fora de um contexto com suporte, `_` é tratado não como um descarte, mas como uma variável válida. Se um identificador nomeado `_` já estiver no escopo, o uso de `_` como um descarte autônomo poderá resultar em:

- Modificação acidental do valor da variável no escopo `_` atribuindo-lhe o valor do descarte pretendido. Por exemplo:

  C#

  ```
  private static void ShowValue(int _)
  {
     byte[] arr = [0, 0, 1, 2];
     _ = BitConverter.ToInt32(arr, 0);
     Console.WriteLine(_);
  }
   // The example displays the following output:
   //       33619968
  ```

- Um erro do compilador por violar a segurança de tipos. Por exemplo:

  C#

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

Consulte também

- Remover valor de expressão desnecessário (regra de estilo IDE0058)

- Remover atribuição de valor desnecessária (regra de estilo IDE0059)
- Remover parâmetro não utilizado (regra de estilo IDE0060)
- Desconstruindo tuplas e outros tipos
- Operador is
- switch expressão

Desconstruindo tuplas e outros tipos

Artigo • 03/01/2025

Uma tupla fornece uma maneira leve de recuperar vários valores de uma chamada de método. Mas depois de recuperar a tupla, você precisa lidar com seus elementos individuais. Trabalhar elemento por elemento é incômodo, conforme mostra o exemplo a seguir. O método `QueryCityData` retorna uma tupla de três e cada um de seus elementos é atribuído a uma variável em uma operação separada.

C#

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

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

         // Do something with the data.
    }

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

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

Recuperar vários valores de propriedade e de campo de um objeto pode ser igualmente complicado: é preciso atribuir um valor de campo ou de propriedade a uma variável, membro por membro.

Você pode recuperar vários elementos de uma tupla ou recuperar vários valores calculados, de campo e de propriedade de um objeto em uma só operação *deconstruct*. Para desconstruir uma tupla, você atribui os elementos dela a variáveis individuais. Quando você desconstrói um objeto, você atribui os elementos dela a variáveis individuais.

Tuplas

O C# conta com suporte interno à desconstrução de tuplas, que permite que você descompacte todos os itens em uma tupla em uma única operação. A sintaxe geral para desconstruir uma tupla é semelhante à sintaxe para definir uma: coloque as variáveis para as quais cada elemento deve ser atribuído entre parênteses no lado esquerdo de uma instrução de atribuição. Por exemplo, a instrução a seguir atribui os elementos de uma tupla de quatro a quatro variáveis separadas:

```
C#

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

Há três maneiras de desconstruir uma tupla:

- Você pode declarar explicitamente o tipo de cada campo dentro de parênteses. O exemplo a seguir usa essa abordagem para desconstruir a tupla de três retornada pelo método `QueryCityData`.

  ```
  C#

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

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

- Você pode usar a palavra-chave `var` de modo que o C# infira o tipo de cada variável. Você coloca a palavra-chave `var` fora dos parênteses. O exemplo a seguir usa a inferência de tipos ao desconstruir a tupla de três retornada pelo método `QueryCityData`.

  ```
  C#

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

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

  Você também pode usar a palavra-chave `var` individualmente com qualquer uma ou todas as declarações de variável dentro dos parênteses.

  ```
  C#
  ```

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

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

O exemplo anterior é complicado e não é recomendado.

- Por fim, você pode desconstruir a tupla em variáveis já declaradas.

C#

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

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

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

- Você pode misturar declaração de variável e atribuição em uma desconstrução.

C#

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

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

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

Você não pode especificar um tipo específico fora dos parênteses, mesmo se todos os campos na tupla tiverem o mesmo tipo. Isso gera um erro do compilador CS8136, "O formulário de desconstrução 'var (...)' não permite um tipo específico para 'var'.".

Você deve atribuir cada elemento da tupla a uma variável. Se você omitir qualquer elemento, o compilador gerará o erro CS8132, "Não é possível desconstruir uma tupla de 'x' elementos em 'y' variáveis".

Elementos tupla com descartes

Geralmente, ao desconstruir uma tupla, você está interessado nos valores de apenas alguns elementos. Você pode aproveitar o suporte do C# para *descartes*, que são variáveis somente de gravação cujos valores você escolheu ignorar. Em uma atribuição, você declara um descarte com um caractere de sublinhado ("_"). Você pode descartar quantos valores desejar; um único descarte, `_`, representa todos os valores descartados.

O exemplo a seguir ilustra o uso de tuplas com descartes. O método `QueryCityDataForYears` a seguir retorna uma tupla de seis com o nome de uma cidade, sua área, um ano, a população da cidade nesse ano, um segundo ano e população da cidade nesse segundo ano. O exemplo mostra a alteração na população entre esses dois anos. Entre os dados disponíveis da tupla, não estamos preocupados com a área da cidade e sabemos o nome da cidade e as duas datas em tempo de design. Como resultado, estamos interessados apenas nos dois valores de população armazenados na tupla e podemos lidar com seus valores restantes como descartes.

C#

```
using System;

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

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

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

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

```
            return (name, area, year1, population1, year2, population2);
        }

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

Tipos definidos pelo usuário

O C# oferece suporte integrado para desconstrução de tipos de tuplas, record e DictionaryEntry. No entanto, como o autor de uma classe, um struct ou uma interface, você pode permitir instâncias do tipo a ser desconstruído implementando um ou mais métodos `Deconstruct`. O método retorna nulo. Um parâmetro out na assinatura do método representa cada valor a ser desconstruído. Por exemplo, o seguinte método `Deconstruct` de uma classe `Person` retorna o primeiro nome, nome do meio e sobrenome:

C#

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

Em seguida, você pode desconstruir uma instância da classe `Person` denominada `p` com uma atribuição semelhante à seguinte:

C#

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

O exemplo a seguir sobrecarrega o método `Deconstruct` para retornar várias combinações de propriedades de um objeto `Person`. As sobrecargas individuais retornam:

- Um primeiro nome e um sobrenome.
- Um primeiro nome, um nome do meio e o nome de família.
- Um nome, um nome de família, um nome de cidade e um nome de estado.

C#

```
using System;

public class Person
```

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

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

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

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

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

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

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

Vários métodos `Deconstruct` com o mesmo número de parâmetros são ambíguos. Você deve ter cuidado ao definir métodos `Deconstruct` com diferentes números de parâmetros ou "aridade". Métodos `Deconstruct` com o mesmo número de parâmetros não podem ser distinguidos durante a resolução de sobrecarga.

Tipo definido pelo usuário com descartes

Assim como você faria com tuplas, você pode usar descartes para ignorar os itens selecionados retornados por um método `Deconstruct`. Uma variável chamada "_" representa um descarte. Uma única operação de desconstrução pode incluir vários descartes.

O exemplo a seguir desconstrói um objeto `Person` em quatro cadeias de caracteres (o primeiro e os nomes de família, a cidade e o estado), mas descarta o nome da família e o estado.

C#

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

Métodos de extensão de desconstrução

Se você não for o autor de uma classe, struct ou interface, ainda poderá descontruir objetos desse tipo implementando um ou mais `Deconstruct` métodos de extensão para retornar os valores nos quais você está interessado.

O exemplo a seguir define dois métodos de extensão `Deconstruct` para a classe System.Reflection.PropertyInfo. O primeiro retorna um conjunto de valores que indicam as características da propriedade. O segundo indica a acessibilidade da propriedade. Valores booleanos indicam se a propriedade tem métodos de acesso get e set separados ou diferentes níveis de acessibilidade. Se houver apenas um acessador ou ambos os acessadores get e set têm a mesma acessibilidade, a variável `access` indica a acessibilidade da propriedade como um todo. Caso contrário, a acessibilidade dos acessadores get e set é indicada pelas variáveis `getAccess` e `setAccess`.

C#

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

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

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

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

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

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

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

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

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

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

        if (getter != null)
        {
            if (getter.IsPublic)
                getAccessTemp = "public";
            else if (getter.IsPrivate)
                getAccessTemp = "private";
            else if (getter.IsAssembly)
                getAccessTemp = "internal";
```

```
            else if (getter.IsFamily)
                getAccessTemp = "protected";
            else if (getter.IsFamilyOrAssembly)
                getAccessTemp = "protected internal";
        }

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

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

public class ExampleExtension
{
    public static void Main()
    {
        Type dateType = typeof(DateTime);
        PropertyInfo prop = dateType.GetProperty("Now");
        var (isStatic, isRO, isIndexed, propType) = prop;
        Console.WriteLine($"\nThe {dateType.FullName}.{prop.Name}
property:");
        Console.WriteLine($"   PropertyType: {propType.Name}");
        Console.WriteLine($"   Static:       {isStatic}");
        Console.WriteLine($"   Read-only:    {isRO}");
        Console.WriteLine($"   Indexed:      {isIndexed}");

        Type listType = typeof(List<>);
        prop = listType.GetProperty("Item",
                                    BindingFlags.Public |
BindingFlags.NonPublic | BindingFlags.Instance | BindingFlags.Static);
        var (hasGetAndSet, sameAccess, accessibility, getAccessibility,
```

```
setAccessibility) = prop;
        Console.Write($"\nAccessibility of the {listType.FullName}.
{prop.Name} property: ");

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

Método de extensão para tipos de sistema

Alguns tipos de sistema fornecem o método `Deconstruct` como uma conveniência. Por exemplo, o tipo System.Collections.Generic.KeyValuePair<TKey,TValue> fornece essa funcionalidade. Quando você está iterando em System.Collections.Generic.Dictionary<TKey,TValue>, cada elemento é um `KeyValuePair<TKey, TValue>` e pode ser desconstruído. Considere o seguinte exemplo:

C#

```
Dictionary<string, int> snapshotCommitMap =
new(StringComparer.OrdinalIgnoreCase)
{
    ["https://github.com/dotnet/docs"] = 16_465,
    ["https://github.com/dotnet/runtime"] = 114_223,
    ["https://github.com/dotnet/installer"] = 22_436,
    ["https://github.com/dotnet/roslyn"] = 79_484,
    ["https://github.com/dotnet/aspnetcore"] = 48_386
};

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

```
10th, 2021.");
}
```

Tipos `record`

Quando você declara um tipo de registro usando dois ou mais parâmetros posicionais, o compilador cria um método `Deconstruct` com um parâmetro `out` para cada parâmetro posicional na declaração `record`. Para obter mais informações, consulte Sintaxe posicional para definição de propriedade e Comportamento de desconstrutor em registros derivados.

Confira também

- Declaração de variável de desconstrução (regra de estilo IDE0042)
- Descartes
- Tipos de tupla

Exceções e tratamento de exceções

19/06/2025

Os recursos de tratamento de exceção da linguagem C# ajudam você a lidar com situações inesperadas ou excepcionais que ocorrem quando um programa está em execução. O tratamento de exceções usa as palavras-chave `try`, `catch` e `finally` para tentar ações que podem não ter êxito, lidar com falhas quando você decide que é razoável fazê-lo e limpar os recursos posteriormente. As exceções podem ser geradas pelo CLR (Common Language Runtime), por bibliotecas .NET ou de terceiros ou pelo código do aplicativo. As exceções são criadas usando a `throw` palavra-chave.

Em muitos casos, uma exceção pode ser gerada não por um método que seu código chamou diretamente, mas por outro método mais abaixo na pilha de chamadas. Quando uma exceção é lançada, o CLR desenrola a pilha, procurando um método que tenha um bloco `catch` para o tipo de exceção específico e executará o primeiro bloco `catch` desse tipo encontrado. Se não encontrar nenhum bloco apropriado `catch` em qualquer lugar na pilha de chamadas, ele encerrará o processo e exibirá uma mensagem para o usuário.

Neste exemplo, um método testa a divisão por zero e captura o erro. Sem a manipulação de exceções, esse programa terminaria com um erro de exceção não tratada **DivideByZeroException**.

C#

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

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

        try
        {
            result = SafeDivision(a, b);
            Console.WriteLine($"{a} divided by {b} = {result}");
        }
        catch (DivideByZeroException)
        {
```

```
            Console.WriteLine("Attempted divide by zero.");
        }
    }
}
```

Visão geral de exceções

As exceções têm as seguintes propriedades:

- Exceções são tipos que, em última análise, derivam de `System.Exception`.
- Use um bloco `try` nas instruções que podem lançar exceções.
- Depois que ocorre uma exceção no bloco `try`, o fluxo de controle salta para o primeiro manipulador de exceção associado presente ao longo da pilha de chamadas. Em C#, a `catch` palavra-chave é usada para definir um manipulador de exceção.
- Se nenhum manipulador de exceção para uma determinada exceção estiver presente, o programa interromperá a execução com uma mensagem de erro.
- Não capture uma exceção, a menos que você possa manipulá-la e deixar o aplicativo em um estado conhecido. Se você capturar `System.Exception`, relance-o usando a palavra-chave `throw` no final do bloco `catch`.
- Se um `catch` bloco definir uma variável de exceção, você poderá usá-la para obter mais informações sobre o tipo de exceção que ocorreu.
- As exceções podem ser geradas explicitamente por um programa usando a `throw` palavra-chave.
- Objetos de exceção contêm informações detalhadas sobre o erro, como o estado da pilha de chamadas e uma descrição de texto do erro.
- O código em um `finally` bloco é executado independentemente de uma exceção ser gerada. Use um `finally` bloco para liberar recursos, por exemplo, para fechar todos os fluxos ou arquivos que foram abertos no `try` bloco.
- As exceções gerenciadas no .NET são implementadas sobre o mecanismo de tratamento de exceção estruturado do Win32. Para obter mais informações, consulte Tratamento Estruturado de Exceções (C/C++) e Um Curso Intensivo sobre as Profundezas do Tratamento de Exceções Estruturadas do Win32 .

Especificação da linguagem C#

Para obter mais informações, consulte Exceções na Especificação da Linguagem C#. A especificação de idioma é a fonte definitiva para a sintaxe e o uso de C#.

Consulte também

- System.Exception
- Declarações de tratamento de exceção
- Exceções

Usar exceções

19/06/2025

No C#, os erros no programa em tempo de execução são propagados por meio do programa usando um mecanismo chamado exceções. As exceções são geradas pelo código que encontra um erro e capturadas pelo código que pode corrigir o erro. As exceções podem ser geradas pelo tempo de execução do .NET ou pelo código em um programa. Depois que uma exceção é gerada, ela se propaga pela pilha de chamadas até que uma instrução `catch` para a exceção seja encontrada. Exceções não capturadas são tratadas por um manipulador de exceção genérico fornecido pelo sistema que exibe uma caixa de diálogo.

As exceções são representadas por classes derivadas de Exception. Essa classe identifica o tipo de exceção e contém propriedades que têm detalhes sobre a exceção. Gerar uma exceção envolve a criação de uma instância de uma classe derivada de exceção, opcionalmente configurando propriedades da exceção e, em seguida, lançando o objeto usando a `throw` palavra-chave. Por exemplo:

C#

```
class CustomException : Exception
{
    public CustomException(string message)
    {
    }
}
private static void TestThrow()
{
    throw new CustomException("Custom exception in TestThrow()");
}
```

Depois que uma exceção é lançada, o runtime verifica a instrução atual para ver se ela está dentro de um `try` bloco. Se estiver, todos os blocos `catch` associados ao bloco `try` serão verificados para ver se eles podem capturar a exceção. `Catch` blocos normalmente especificam tipos de exceção; se o tipo do `catch` bloco for do mesmo tipo que a exceção ou uma classe base da exceção, o `catch` bloco poderá manipular o método. Por exemplo:

C#

```
try
{
    TestThrow();
}
catch (CustomException ex)
{
```

```
    System.Console.WriteLine(ex.ToString());
}
```

Se a instrução que lança uma exceção não estiver dentro de um bloco `try` ou se o bloco `try` que a encerra não tiver um bloco `catch` correspondente, o runtime verificará o método de chamada para uma instrução `try` e blocos `catch`. O runtime continuará acima na pilha de chamada, pesquisando um bloco `catch` compatível. Depois que o `catch` bloco é encontrado e executado, o controle é passado para a próxima instrução após esse `catch` bloco.

Uma `try` instrução pode conter mais de um `catch` bloco. A primeira `catch` instrução que pode lidar com a exceção é executada; todas as instruções a seguir `catch` , mesmo que sejam compatíveis, são ignoradas. Ordenar blocos catch do mais específico (ou mais derivado) para o menos específico. Por exemplo:

C#

```
using System;
using System.IO;

namespace Exceptions
{
    public class CatchOrder
    {
        public static void Main()
        {
            try
            {
                using (var sw = new StreamWriter("./test.txt"))
                {
                    sw.WriteLine("Hello");
                }
            }
            // Put the more specific exceptions first.
            catch (DirectoryNotFoundException ex)
            {
                Console.WriteLine(ex);
            }
            catch (FileNotFoundException ex)
            {
                Console.WriteLine(ex);
            }
            // Put the least specific exception last.
            catch (IOException ex)
            {
                Console.WriteLine(ex);
            }
            Console.WriteLine("Done");
        }
```

```
    }
}
```

Antes que o `catch` bloco seja executado, o runtime verifica se há `finally` blocos. Os blocos `Finally` permitem que o programador limpe qualquer estado ambíguo que pode ser deixado de um bloco `try` cancelado ou libere quaisquer recursos externos (como identificadores de gráfico, conexões de banco de dados ou fluxos de arquivo) sem esperar o coletor de lixo no runtime finalizar os objetos. Por exemplo:

C#

```
static void TestFinally()
{
    FileStream? file = null;
    //Change the path to something that works on your machine.
    FileInfo fileInfo = new System.IO.FileInfo("./file.txt");

    try
    {
        file = fileInfo.OpenWrite();
        file.WriteByte(0xF);
    }
    finally
    {
        // Closing the file allows you to reopen it immediately - otherwise
IOException is thrown.
        file?.Close();
    }

    try
    {
        file = fileInfo.OpenWrite();
        Console.WriteLine("OpenWrite() succeeded");
    }
    catch (IOException)
    {
        Console.WriteLine("OpenWrite() failed");
    }
}
```

Se `WriteByte()` gerar uma exceção, o código no segundo bloco `try` que tentar reabrir o arquivo falharia se `file.Close()` não for chamado, e o arquivo permaneceria bloqueado. Como `finally` os blocos são executados mesmo se uma exceção for lançada, o `finally` bloco no exemplo anterior permite que o arquivo seja fechado corretamente e ajuda a evitar um erro.

Se nenhum bloco compatível `catch` for encontrado na pilha de chamadas após a geração de uma exceção, ocorrerá uma das três coisas:

- Se a exceção estiver dentro de um finalizador, o finalizador será anulado e o finalizador base, se houver, será chamado.
- Se a pilha de chamadas contiver um construtor estático ou um inicializador de campo estático, uma TypeInitializationException será gerada, com a exceção original atribuída à propriedade InnerException da nova exceção.
- Se o início do thread for atingido, o thread será encerrado.

Tratamento de exceções (Guia de Programação em C#)

19/06/2025

Um bloco try é usado por programadores C# para particionar código que pode ser afetado por uma exceção. Os blocos catch associados são usados para lidar com quaisquer exceções resultantes. Um bloco finally contém o código que é executado, seja lançada uma exceção ou não no bloco `try`, como liberação de recursos alocados no bloco `try`. Um `try` bloco requer um ou mais blocos associados `catch` , ou um `finally` bloco ou ambos.

Os exemplos a seguir mostram uma `try-catch` instrução, uma `try-finally` instrução e uma `try-catch-finally` instrução.

C#

```
try
{
    // Code to try goes here.
}
catch (SomeSpecificException ex)
{
    // Code to handle the exception goes here.
    // Only catch exceptions that you know how to handle.
    // Never catch base class System.Exception without
    // rethrowing it at the end of the catch block.
}
```

C#

```
try
{
    // Code to try goes here.
}
finally
{
    // Code to execute after the try block goes here.
}
```

C#

```
try
{
    // Code to try goes here.
}
catch (SomeSpecificException ex)
{
```

```
    // Code to handle the exception goes here.
}
finally
{
    // Code to execute after the try (and possibly catch) blocks
    // goes here.
}
```

Um `try` bloco sem um `catch` ou `finally` bloco causa um erro do compilador.

Blocos catch

Um bloco `catch` pode especificar o tipo de exceção a ser capturado. A especificação de tipo é chamada de *filtro de exceção*. O tipo de exceção deve ser derivado de Exception. Em geral, não especifique Exception como o filtro de exceção, a menos que você saiba como lidar com todas as exceções que podem ser lançadas no bloco `try` ou que você tenha incluído uma instrução throw no final do bloco `catch`.

Vários `catch` blocos com classes de exceção diferentes podem ser encadeados juntos. Os `catch` blocos são avaliados de cima para baixo em seu código, mas apenas um `catch` bloco é executado para cada exceção gerada. O primeiro `catch` bloco que especifica o tipo exato ou uma classe base da exceção lançada é executado. Se nenhum `catch` bloco especificar uma classe de exceção correspondente, um `catch` bloco que não tem nenhum tipo será selecionado, se houver um presente na instrução. É importante posicionar `catch` blocos com as classes de exceção mais específicas (ou seja, as mais derivadas) primeiro.

Capture exceções quando as seguintes condições forem verdadeiras:

- Você tem uma boa compreensão de por que a exceção pode ser gerada e você pode implementar uma recuperação específica, como solicitar que o usuário insira um novo nome de arquivo ao capturar um FileNotFoundException objeto.
- Você pode criar e lançar uma exceção nova e mais específica.

  C#

  ```
  int GetInt(int[] array, int index)
  {
      try
      {
          return array[index];
      }
      catch (IndexOutOfRangeException e)
      {
          throw new ArgumentOutOfRangeException(
              "Parameter index is out of range.", e);
  ```

```
    }
}
```

- Você deseja tratar parcialmente uma exceção antes de passá-la para mais tratamento. No exemplo a seguir, o bloco `catch` é usado para adicionar uma entrada no log de erros antes de relançar a exceção.

```C#
try
{
    // Try to access a resource.
}
catch (UnauthorizedAccessException e)
{
    // Call a custom error logging procedure.
    LogError(e);
    // Re-throw the error.
    throw;
}
```

Você também pode especificar *filtros de exceção* para adicionar uma expressão booliana a uma cláusula catch. Filtros de exceção indicam que uma cláusula catch específica corresponde somente quando essa condição é verdadeira. No exemplo a seguir, ambas as cláusulas catch usam a mesma classe de exceção, mas uma condição extra é verificada para criar uma mensagem de erro diferente:

```C#
int GetInt(int[] array, int index)
{
    try
    {
        return array[index];
    }
    catch (IndexOutOfRangeException e) when (index < 0)
    {
        throw new ArgumentOutOfRangeException(
            "Parameter index cannot be negative.", e);
    }
    catch (IndexOutOfRangeException e)
    {
        throw new ArgumentOutOfRangeException(
            "Parameter index cannot be greater than the array size.", e);
    }
}
```

Um filtro de exceção que sempre retorna `false` pode ser usado para examinar todas as exceções, mas não processá-las. Um uso típico é registrar exceções:

C#

```
public class ExceptionFilter
{
    public static void Main()
    {
        try
        {
            string? s = null;
            Console.WriteLine(s.Length);
        }
        catch (Exception e) when (LogException(e))
        {
        }
        Console.WriteLine("Exception must have been handled");
    }

    private static bool LogException(Exception e)
    {
        Console.WriteLine($"\tIn the log routine. Caught {e.GetType()}");
        Console.WriteLine($"\tMessage: {e.Message}");
        return false;
    }
}
```

O `LogException` método sempre retorna `false`, nenhuma cláusula `catch` corresponde a este filtro de exceção. A cláusula catch pode ser geral, usando System.Exceptione cláusulas posteriores podem processar classes de exceção mais específicas.

Blocos Finally

Um bloco `finally` permite que você limpe as ações que são realizadas em um bloco `try`. Se estiver presente, o `finally` bloco será executado por último, após o `try` bloco e qualquer bloco correspondente `catch` . Um `finally` bloco sempre é executado, se uma exceção é lançada ou um `catch` bloco que corresponde ao tipo de exceção é encontrado.

O bloco `finally` pode ser usado para liberar recursos, como fluxos de arquivos, conexões de banco de dados e identificadores de gráficos, sem esperar que o coletor de lixo no runtime finalize os objetos.

No exemplo a seguir, o `finally` bloco é usado para fechar um arquivo aberto no `try` bloco. Observe que o estado do identificador de arquivo é verificado antes do arquivo ser fechado. Se o `try` bloco não puder abrir o arquivo, o identificador de arquivo ainda terá o valor `null` e o `finally` bloco não tentará fechá-lo. Em vez disso, se o arquivo for aberto com êxito no `try` bloco, o `finally` bloco fechará o arquivo aberto.

C#

```
FileStream? file = null;
FileInfo fileinfo = new System.IO.FileInfo("./file.txt");
try
{
    file = fileinfo.OpenWrite();
    file.WriteByte(0xF);
}
finally
{
    // Check for null because OpenWrite might have failed.
    file?.Close();
}
```

Especificação da linguagem C#

Para obter mais informações, consulte Exceções e a instrução Try na Especificação da Linguagem C#. A especificação de idioma é a fonte definitiva para a sintaxe e o uso de C#.

Consulte também

- Referência de C#
- Declarações de tratamento de exceção
- Instrução Using

Criar e lançar exceções

Artigo • 18/03/2025

Exceções são usadas para indicar que ocorreu um erro durante a execução do programa. Objetos de exceção que descrevem um erro são criados e, em seguida, *lançados* om a instrução ou expressão throw. Em seguida, o runtime pesquisa o manipulador de exceção mais compatível.

Os programadores devem gerar exceções quando uma ou mais das seguintes condições forem verdadeiras:

- O método não pode concluir sua funcionalidade definida. Por exemplo, se um parâmetro para um método tiver um valor inválido:

  C#

  ```
  static void CopyObject(SampleClass original)
  {
      _ = original ?? throw new ArgumentException("Parameter cannot be
  null", nameof(original));
  }
  ```

- É feita uma chamada inadequada a um objeto, com base no estado do objeto. Um exemplo pode ser a tentativa de gravar em um arquivo somente leitura. Nos casos em que um estado de objeto não permite uma operação, gere uma instância de InvalidOperationException ou um objeto com base em uma derivação dessa classe. O código a seguir é um exemplo de um método que gera um objeto InvalidOperationException:

  C#

  ```
  public class ProgramLog
  {
      FileStream logFile = null!;
      public void OpenLog(FileInfo fileName, FileMode mode) { }

      public void WriteLog()
      {
          if (!logFile.CanWrite)
          {
              throw new InvalidOperationException("Logfile cannot be
  read-only");
          }
          // Else write data to the log and return.
  ```

```
    }
}
```

- Quando um argumento para um método causa uma exceção. Nesse caso, a exceção original deve ser capturada e uma instância ArgumentException deve ser criada. A exceção original deve ser passada para o construtor do ArgumentException como o parâmetro InnerException:

```C#
static int GetValueFromArray(int[] array, int index)
{
    try
    {
        return array[index];
    }
    catch (IndexOutOfRangeException e)
    {
        throw new ArgumentOutOfRangeException(
            "Parameter index is out of range.", e);
    }
}
```

> **Observação**
>
> O exemplo anterior mostra como usar a propriedade `InnerException`. É intencionalmente simplificado. Na prática, você deve verificar se um índice está na faixa antes de usá-lo. Você pode usar essa técnica de encapsular uma exceção quando um membro de um parâmetro gera uma exceção que você não poderia prever antes de chamar o membro.

As exceções contêm uma propriedade chamada StackTrace. Essa cadeia de caracteres contém o nome dos métodos na pilha de chamadas atual, juntamente com o nome do arquivo e o número da linha em que a exceção foi gerada para cada método. Um objeto StackTrace é criado automaticamente pelo CLR (Common Language Runtime) no ponto da instrução `throw`, de modo que as exceções devem ser lançadas do ponto em que o rastreamento de pilha deve começar.

Todas as exceções contêm uma propriedade chamada Message. Essa cadeia de caracteres deve ser definida para explicar o motivo da exceção. Informações confidenciais à segurança não devem ser colocadas no texto da mensagem. Além de Message, ArgumentException contém uma propriedade chamada ParamName que deve ser definida como o nome do argumento que causou a geração da exceção. Em um setter de propriedade, ParamName deve ser definido como `value`.

Métodos públicos e protegidos geram exceções sempre que não conseguem concluir suas funções pretendidas. A classe de exceção gerada é a exceção mais específica disponível que atende às condições de erro. Essas exceções devem ser documentadas como parte da funcionalidade de classe, e classes ou atualizações derivadas para a classe original devem manter o mesmo comportamento para compatibilidade com versões anteriores.

Coisas a evitar ao lançar exceções

A lista a seguir identifica as práticas a serem evitadas ao lançar exceções:

- Não use exceções para alterar o fluxo de um programa como parte da execução comum. Use exceções para relatar e manipular condições de erro.
- As exceções não devem ser retornadas como um valor retornado ou um parâmetro em vez de serem lançadas.
- Não lance System.Exception, System.SystemException, System.NullReferenceException ou System.IndexOutOfRangeException intencionalmente de seu próprio código-fonte.
- Não crie exceções que podem ser lançadas no modo de depuração, mas não no modo de versão. Em vez disso, use o Debug Assert para identificar erros em tempo de execução durante a fase de desenvolvimento.

Exceções em métodos de retorno de tarefa

Os métodos declarados com o modificador `async` têm algumas considerações especiais quando se trata de exceções. As exceções geradas em um método `async` são armazenadas na tarefa retornada e não surgem até que, por exemplo, a tarefa seja aguardada. Para obter mais informações sobre exceções armazenadas, consulte exceções assíncronas.

Recomendamos que você valide argumentos e gere exceções correspondentes, como ArgumentException e ArgumentNullException, antes de inserir as partes assíncronas de seus métodos. Ou seja, essas exceções de validação devem surgir de forma síncrona antes do início do trabalho. O snippet de código a seguir mostra um exemplo em que, se as exceções forem geradas, as exceções ArgumentException surgirão de forma síncrona, enquanto a InvalidOperationException seria armazenada na tarefa retornada.

C#

```
// Non-async, task-returning method.
// Within this method (but outside of the local function),
// any thrown exceptions emerge synchronously.
```

```
public static Task<Toast> ToastBreadAsync(int slices, int toastTime)
{
    if (slices is < 1 or > 4)
    {
        throw new ArgumentException(
            "You must specify between 1 and 4 slices of bread.",
            nameof(slices));
    }

    if (toastTime < 1)
    {
        throw new ArgumentException(
            "Toast time is too short.", nameof(toastTime));
    }

    return ToastBreadAsyncCore(slices, toastTime);

    // Local async function.
    // Within this function, any thrown exceptions are stored in the task.
    static async Task<Toast> ToastBreadAsyncCore(int slices, int time)
    {
        for (int slice = 0; slice < slices; slice++)
        {
            Console.WriteLine("Putting a slice of bread in the toaster");
        }
        // Start toasting.
        await Task.Delay(time);

        if (time > 2_000)
        {
            throw new InvalidOperationException("The toaster is on fire!");
        }

        Console.WriteLine("Toast is ready!");

        return new Toast();
    }
}
```

Definir classes de exceção

Os programas podem lançar uma classe de exceção predefinida no namespace System (exceto quando observado anteriormente) ou criar suas próprias classes de exceção derivando de Exception. As classes derivadas devem definir pelo menos três construtores: um construtor sem parâmetros, um que define a propriedade da mensagem e outro que define as propriedades Message e InnerException. Por exemplo:

```
C#
```

```
[Serializable]
public class InvalidDepartmentException : Exception
{
    public InvalidDepartmentException() : base() { }
    public InvalidDepartmentException(string message) : base(message) { }
    public InvalidDepartmentException(string message, Exception inner) :
base(message, inner) { }
}
```

Adicione novas propriedades à classe de exceção quando os dados fornecidos são úteis para resolver a exceção. Se novas propriedades forem adicionadas à classe de exceção derivada, `ToString()` deverá ser substituída para retornar as informações adicionadas.

Especificação da linguagem C#

Para obter mais informações, veja Exceções e A declaração throw na Especificação da Linguagem C#. A especificação de idioma é a fonte definitiva para a sintaxe e o uso de C#.

Consulte também

- Hierarquia de Exceção

Exceções geradas pelo compilador

19/06/2025

Algumas exceções são geradas automaticamente pelo runtime do .NET quando as operações básicas falham. Essas exceções e suas condições de erro são listadas na tabela a seguir.

Expandir a tabela

Exceção	Descrição
ArithmeticException	Uma classe base para exceções que ocorrem durante operações aritméticas, como DivideByZeroException e OverflowException.
ArrayTypeMismatchException	Gerada quando uma matriz não pode armazenar um determinado elemento porque o tipo real do elemento é incompatível com o tipo real da matriz.
DivideByZeroException	Lançada quando é feita uma tentativa de dividir um valor inteiro por zero.
IndexOutOfRangeException	Lançada quando é feita uma tentativa de indexar uma matriz quando o índice é menor que zero ou fora dos limites da matriz.
InvalidCastException	Gerada quando uma conversão explícita de um tipo base em uma interface ou em um tipo derivado falha durante a execução.
NullReferenceException	Gerado quando uma tentativa é feita para referenciar um objeto cujo valor é nulo.
OutOfMemoryException	Gerado quando uma tentativa de alocar memória usando o novo operador falha. Essa exceção indica que a memória disponível para o common language runtime foi esgotada.
OverflowException	Lançada quando uma operação aritmética em um contexto `checked` estoura.
StackOverflowException	Lançado quando a pilha de execução fica esgotada por ter muitas chamadas de método pendentes; geralmente indica uma recursão muito profunda ou infinita.
TypeInitializationException	Lançada quando um construtor estático gera uma exceção e nenhuma cláusula `catch` compatível existe para capturá-la.

Consulte também

- Declarações de tratamento de exceção

Regras e convenções de nomenclatura do identificador C#

Artigo • 10/02/2025

Um identificador é o nome que você atribui a um tipo (classe, interface, struct, delegado ou enum), membro, variável ou namespace.

Regras de nomenclatura

Os identificadores válidos devem seguir essas regras. O compilador C# produz um erro para qualquer identificador que não siga estas regras:

- Os identificadores devem começar com uma letra ou caractere sublinhado (`_`).
- Os identificadores podem conter caracteres de letra Unicode, caracteres de dígitos decimais, caracteres de conexão Unicode, caracteres de combinação Unicode ou caracteres de formatação Unicode. Para obter mais informações sobre categorias Unicode, consulte o banco de dados de categoria unicode .

Você pode declarar identificadores que correspondem às palavras-chave C# usando o prefixo `@` no identificador. O `@` não faz parte do nome do identificador. Por exemplo, `@if` declara um identificador chamado `if`. Esses identificadores textuais são destinados principalmente para interoperabilidade com os identificadores declarados em outras linguagens.

Para obter uma definição completa de identificadores válidos, consulte o artigo Identificadores de na Especificação da Linguagem C#.

> ⓘ **Importante**
>
> A especificação da linguagem C# permite apenas letra (Lu, Ll, Lt, Lm ou Nl), dígito (Nd), conexão (Pc), combinação (Mn ou Mc) e categorias de formatação (Cf). Qualquer coisa fora disso é automaticamente substituída usando `_`. Isso pode afetar determinados caracteres Unicode.

Convenções de nomenclatura

Além das regras, as convenções para nomes de identificador são usadas em todas as APIs do .NET. Essas convenções fornecem consistência para nomes, mas o compilador

não os impõe. Você é livre para usar convenções diferentes em seus projetos.

Por convenção, os programas C# usam `PascalCase` para nomes de tipos, namespaces e todos os membros públicos. Além disso, a equipe de `dotnet/docs` usa as seguintes convenções, adotadas do estilo de codificação da equipe do .NET Runtime :

- Os nomes de interface começam com uma maiúscula `I`.
- Os tipos de atributo terminam com a palavra `Attribute`.
- Os tipos enumerados usam um substantivo singular para nonflags e um substantivo plural para sinalizadores.
- Os identificadores não devem conter dois caracteres de sublinhado consecutivo (`_`). Esses nomes são reservados para identificadores gerados pelo compilador.
- Use nomes significativos e descritivos para variáveis, métodos e classes.
- Prefira clareza em vez de brevidade.
- Use PascalCase para nomes de classe e nomes de método.
- Use camelCase para parâmetros de método e variáveis locais.
- Use PascalCase para nomes constantes, campos e constantes locais.
- Os campos de instância privada começam com um sublinhado (`_`) e o texto restante usa CamelCase.
- Os campos estáticos começam com `s_`. Essa convenção não é o comportamento padrão do Visual Studio, nem parte das diretrizes de design do Framework, mas é configurável na configuração do editor.
- Evite usar abreviações ou acrônimos em nomes, exceto para abreviações amplamente conhecidas e aceitas.
- Use namespaces significativos e descritivos que seguem a notação de nome de domínio inverso.
- Escolha os nomes de assembly que representam a finalidade primária do assembly.
- Evite usar nomes de letra única, exceto para contadores de loop simples. Além disso, exemplos de sintaxe que descrevem a sintaxe de constructos C# geralmente usam os seguintes nomes de letra única que correspondem à convenção usada na especificação da linguagem C#. Exemplos de sintaxe são uma exceção à regra.
  - Use `S` para estruturas, `C` para classes.

- Use `M` para métodos.
- Use `v` para variáveis, `p` para parâmetros.
- Use `r` para parâmetros `ref`.

> **Dica**
>
> Você pode impor convenções de nomenclatura que dizem respeito à capitalização, prefixos, sufixos e separadores de palavras usando as regras de nomenclatura do estilo de código .

Nos exemplos a seguir, as diretrizes relativas aos elementos marcados `public` também são aplicáveis ao trabalhar com elementos `protected` e `protected internal`, todos destinados a serem visíveis para chamadores externos.

Caso Pascal

Use pascal casing ("PascalCasing") ao nomear um tipo `class`, `interface`, `struct` ou `delegate`.

C#

```
public class DataService
{
}
```

C#

```
public record PhysicalAddress(
    string Street,
    string City,
    string StateOrProvince,
    string ZipCode);
```

C#

```
public struct ValueCoordinate
{
}
```

C#

```
public delegate void DelegateType(string message);
```

Ao nomear um `interface`, use Pascal Case, além de aplicar ao nome um prefixo `I`. Este prefixo indica claramente aos consumidores que se trata de um `interface`.

C#

```
public interface IWorkerQueue
{
}
```

Ao nomear membros `public` de tipos, como campos, propriedades, eventos, use pascal casing. Além disso, use pascal casing para todos os métodos e funções locais.

C#

```
public class ExampleEvents
{
    // A public field, these should be used sparingly
    public bool IsValid;

    // An init-only property
    public IWorkerQueue WorkerQueue { get; init; }

    // An event
    public event Action EventProcessing;

    // Method
    public void StartEventProcessing()
    {
        // Local function
        static int CountQueueItems() => WorkerQueue.Count;
        // ...
    }
}
```

Ao gravar registros posicionais, use Pascal Case para parâmetros, pois são as propriedades públicas do registro.

C#

```
public record PhysicalAddress(
    string Street,
    string City,
    string StateOrProvince,
    string ZipCode);
```

Para obter mais informações sobre registros posicionais, consulte Sintaxe posicional para definição de propriedade.

Caso camel

Use camel casing ("camelCasing") ao nomear campos `private` ou `internal` e dê a eles o prefixo `_`. Use camel casing ao nomear variáveis locais, incluindo instâncias de um tipo delegado.

C#

```
public class DataService
{
    private IWorkerQueue _workerQueue;
}
```

Dica

Ao editar o código do C# que segue essas convenções de nomenclatura em um IDE que permite a conclusão da instrução, digitar `_` mostrará todos os membros no escopo do objeto.

Ao trabalhar com campos `static` que são `private` ou `internal`, use o prefixo `s_` e, para threads estáticas, use `t_`.

C#

```
public class DataService
{
    private static IWorkerQueue s_workerQueue;

    [ThreadStatic]
    private static TimeSpan t_timeSpan;
}
```

Ao gravar os parâmetros de método, use Camel Case.

C#

```
public T SomeMethod<T>(int someNumber, bool isValid)
{
}
```

Para obter mais informações sobre convenções de nomenclatura em C#, consulte o estilo de codificação da equipe do .NET Runtime .

Diretrizes para a nomenclatura de parâmetros de tipo

As diretrizes a seguir se aplicam a parâmetros de tipo em parâmetros de tipo genérico. Parâmetros de tipo são os espaços reservados para argumentos em um tipo genérico ou um método genérico. Você pode ler mais sobre parâmetros de tipo genérico no guia de programação do C#.

- **Nomeie** parâmetros de tipo genérico com nomes descritivos, a menos que um nome de letra única seja completamente autoexplicativo e um nome descritivo não adicione valor.

  ./snippets/coding-conventions

  ```
  public interface ISessionChannel<TSession> { /*...*/ }
  public delegate TOutput Converter<TInput, TOutput>(TInput from);
  public class List<T> { /*...*/ }
  ```

- **Considere** usando `T` como o nome do parâmetro de tipo para tipos com um único parâmetro de tipo de letra.

  ./snippets/coding-conventions

  ```
  public int IComparer<T>() => 0;
  public delegate bool Predicate<T>(T item);
  public struct Nullable<T> where T : struct { /*...*/ }
  ```

- **Insira** o prefixo "T" em nomes descritivos de parâmetro de tipo.

  ./snippets/coding-conventions

  ```
  public interface ISessionChannel<TSession>
  {
      TSession Session { get; }
  }
  ```

- **Considere** indicando restrições colocadas em um parâmetro de tipo no nome do parâmetro. Por exemplo, um parâmetro restrito a `ISession` pode ser chamado `TSession`.

A regra de análise de código CA1715 pode ser usada para garantir que os parâmetros de tipo sejam nomeados adequadamente.

Convenções de nomenclatura extras

- Em exemplos que não incluem o uso de diretivas, use as qualificações do namespace. Se você souber que um namespace é importado por padrão em um projeto, não será necessário qualificar totalmente os nomes desse namespace. Os nomes qualificados poderão ser quebrados após um ponto (.) se forem muito longos para uma única linha, conforme mostrado no exemplo a seguir.

  C#

  ```
  var currentPerformanceCounterCategory = new System.Diagnostics.
      PerformanceCounterCategory();
  ```

- Você não precisa alterar os nomes de objetos que foram criados usando as ferramentas de designer do Visual Studio para torná-los adequados a outras diretrizes.

Convenções comuns de código C#

Artigo • 22/01/2025

Convenções de código são essenciais para manter a legibilidade, a consistência e a colaboração de código em uma equipe de desenvolvimento. É mais fácil de entender, manter e estender o código que segue as práticas do setor e as diretrizes estabelecidas. A maioria dos projetos impõe um estilo consistente por meio de convenções de código. Os projetos dotnet/docs e dotnet/samples não são exceção. Nesta série de artigos, você aprenderá nossas convenções de codificação e as ferramentas que usamos para aplicá-las. Você pode tomar nossas convenções como estão ou modificá-las para atender às necessidades de sua equipe.

Escolhemos nossas convenções com base nas seguintes metas:

1. *Correção*: nossos exemplos são copiados e colados em seus aplicativos. Esperamos isso, portanto, precisamos criar um código resiliente e correto, mesmo após várias edições.
2. *Ensino*: a finalidade de nossos exemplos é ensinar tudo do .NET e C#. Por esse motivo, não colocamos restrições em nenhum recurso de idioma ou API. Em vez disso, esses exemplos ensinam quando um recurso é uma boa opção.
3. *Consistência*: os leitores esperam uma experiência consistente em nosso conteúdo. Todos os exemplos devem estar em conformidade com o mesmo estilo.
4. *Adoção*: atualizamos agressivamente nossos exemplos para usar novos recursos de linguagem. Essa prática conscientiza os novos recursos e os torna mais familiares para todos os desenvolvedores C#.

> **Importante**
>
> Essas diretrizes são usadas pela Microsoft para desenvolver exemplos e documentação. Elas foram adotados das diretrizes do **.NET runtime, estilo de codificação C#** e **compilador C# (roslyn)**. Escolhemos essas diretrizes devido à sua adoção ao longo de vários anos de desenvolvimento de Software Livre. Essas diretrizes ajudam os membros da comunidade a participar dos projetos de runtime e compilador. Eles devem ser um exemplo de convenções comuns em C# e não uma lista autoritativa (consulte **Diretrizes de Design da Estrutura** para obter diretrizes detalhadas).
>
> As *metas de ensino* e *adoção* são as razões pelas quais a convenção de codificação de documentos difere das convenções de runtime e compilador. O runtime e o compilador têm métricas de desempenho estritas para caminhos frequentes. Não se pode dizer o mesmo de muitos outros aplicativos. Nossa meta de *ensino*

determina que não proibimos nenhuma construção. Em vez disso, os exemplos mostram quando os constructos devem ser usados. Atualizamos exemplos mais agressivamente do que a maioria dos aplicativos de produção. Nossa meta de *adoção* determina que mostramos o código que você deve escrever hoje, mesmo quando o código escrito no ano passado não precisa de alterações.

Este artigo explica nossas diretrizes. As diretrizes evoluem ao longo do tempo e você encontrará exemplos que não seguem nossas diretrizes. Damos as boas-vindas a PRs que trazem esses exemplos para a conformidade, ou problemas que chamam nossa atenção para exemplos que devemos atualizar. Nossas diretrizes são de software livre e damos as boas-vindas a PRs e problemas. No entanto, se o envio alterar essas recomendações, abra um problema para discussão primeiro. Você é bem-vindo a usar nossas diretrizes ou adaptá-las às suas necessidades.

Ferramentas e analisadores

As ferramentas podem ajudar sua equipe a impor as convenções deles. Você pode habilitar a análise de código para impor as regras que preferir. Você também pode criar uma editorconfig para que o Visual Studio imponha automaticamente suas diretrizes de estilo. Como ponto de partida, você pode copiar o dotnet/docs.*editorconfig* para usar nosso estilo.

Essas ferramentas facilitam a adoção de suas diretrizes preferenciais para sua equipe. O Visual Studio aplica as regras em todos os arquivos *.editorconfig* no escopo para formatar seu código. Você pode usar várias configurações para impor convenções corporativas, convenções de equipe e até convenções de projeto granulares.

A análise de código produz avisos e diagnósticos quando detecta violações de regra. Você configura as regras que deseja aplicar ao seu projeto. Em seguida, cada build de CI notifica os desenvolvedores quando eles violam qualquer uma das regras.

IDs de diagnóstico

- Escolha IDs de diagnóstico apropriadas ao criar seus analisadores

Diretrizes de linguagem

As seções a seguir descrevem as práticas que a equipe de documentos do .NET segue para preparar exemplos e exemplos de código. Em geral, siga estas práticas:

- Utilize recursos de linguagem moderna e versões em C# sempre que possível.

- Evite constructos de idioma desatualizados.
- Capture apenas exceções que possam ser tratadas adequadamente; evite capturar exceções gerais. Por exemplo, um exemplo de código não deve capturar o tipo System.Exception sem um filtro de exceção.
- Use tipos de exceção específicos para fornecer mensagens de erro significativas.
- Use consultas LINQ e métodos para manipulação de coleção para melhorar a legibilidade do código.
- Use programação assíncrona com assíncrono e aguarde operações associadas a E/S.
- Tenha cuidado com deadlocks e use Task.ConfigureAwait quando apropriado.
- Use as palavras-chave de idioma para tipos de dados em vez dos tipos de runtime. Por exemplo, use `string` em vez de System.String, ou `int` em vez de System.Int32. Essa recomendação inclui o uso dos tipos `nint` e `nuint`.
- Use `int` em vez de tipos não assinados. O uso de `int` é comum em todo o C# e é mais fácil interagir com outras bibliotecas quando você usa `int`. As exceções são para a documentação específica para tipos de dados não assinados.
- Use `var` somente quando um leitor puder inferir o tipo da expressão. Os leitores exibem nossos exemplos na plataforma de documentos. Eles não têm dicas de focalização ou ferramenta que exibem o tipo de variáveis.
- Escreva código com clareza e simplicidade em mente.
- Evite lógica de código excessivamente complexa e complicada.

Há diretrizes mais específicas a seguir.

Dados de cadeia de caracteres

- Use a interpolação de cadeia de caracteres para concatenar cadeias de caracteres curtas, como é mostrado no código a seguir.

  C#

  ```
  string displayName = $"{nameList[n].LastName},
  {nameList[n].FirstName}";
  ```

- Para acrescentar cadeias de caracteres em loops, especialmente quando você estiver trabalhando com grandes quantidades de texto, use um objeto System.Text.StringBuilder.

  C#

  ```
  var phrase =
  "lalalalalalalalalalalalalalalalalalalalalalalalalalalalalalalalalalalala";
  ```

```
var manyPhrases = new StringBuilder();
for (var i = 0; i < 10000; i++)
{
    manyPhrases.Append(phrase);
}
//Console.WriteLine("tra" + manyPhrases);
```

- Prefira literais de cadeia de caracteres brutos a sequências de escape ou cadeias de caracteres verbatim.

```
C#

var message = """
    This is a long message that spans across multiple lines.
    It uses raw string literals. This means we can
    also include characters like \n and \t without escaping them.
    """;
```

- Use a interpolação de cadeia de caracteres baseada em expressão em vez de interpolação de cadeia de caracteres posicional.

```
C#

// Execute the queries.
Console.WriteLine("scoreQuery:");
foreach (var student in scoreQuery)
{
    Console.WriteLine($"{student.Last} Score: {student.score}");
}
```

Construtores e inicialização

- Use o caso Pascal para parâmetros de construtor primário em tipos de registro:

```
C#

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

- Use minúsculas concatenadas para parâmetros de construtor primário em tipos de classe e struct.

- Use propriedades `required` em vez de construtores para forçar a inicialização de valores de propriedade:

```
C#
```

```
public class LabelledContainer<T>(string label)
{
    public string Label { get; } = label;
    public required T Contents
    {
        get;
        init;
    }
}
```

Matrizes e coleções

- Use expressões de coleção para inicializar todos os tipos de coleção:

C#

```
string[] vowels = [ "a", "e", "i", "o", "u" ];
```

Delegados

- Use Func<> e Action<>, em vez de definir tipos delegados. Em uma classe, defina o método delegado.

C#

```
Action<string> actionExample1 = x => Console.WriteLine($"x is: {x}");

Action<string, string> actionExample2 = (x, y) =>
    Console.WriteLine($"x is: {x}, y is {y}");

Func<string, int> funcExample1 = x => Convert.ToInt32(x);

Func<int, int, int> funcExample2 = (x, y) => x + y;
```

- Chame o método usando a assinatura definida pelo delegado `Func<>` ou `Action<>`.

C#

```
actionExample1("string for x");

actionExample2("string for x", "string for y");

Console.WriteLine($"The value is {funcExample1("1")}");

Console.WriteLine($"The sum is {funcExample2(1, 2)}");
```

- Se você criar instâncias de um tipo delegado, use a sintaxe concisa. Em uma classe, defina o tipo delegado e um método que tenha uma assinatura correspondente.

```
C#

public delegate void Del(string message);

public static void DelMethod(string str)
{
    Console.WriteLine("DelMethod argument: {0}", str);
}
```

- Crie uma instância do tipo delegado e a chame. A declaração a seguir mostra a sintaxe condensada.

```
C#

Del exampleDel2 = DelMethod;
exampleDel2("Hey");
```

- A declaração a seguir usa a sintaxe completa.

```
C#

Del exampleDel1 = new Del(DelMethod);
exampleDel1("Hey");
```

`try-catch` Instruções e `using` no tratamento de exceções

- Use uma instrução try-catch para a maioria da manipulação de exceções.

```
C#

static double ComputeDistance(double x1, double y1, double x2, double
y2)
{
    try
    {
        return Math.Sqrt((x1 - x2) * (x1 - x2) + (y1 - y2) * (y1 -
y2));
    }
    catch (System.ArithmeticException ex)
    {
        Console.WriteLine($"Arithmetic overflow or underflow: {ex}");
        throw;
```

```
    }
}
```

- Simplifique o código usando a instrução using do #C. Se você tiver uma instrução try-finally na qual o único código do bloco `finally` é uma chamada para o método Dispose, use, em vez disso, uma instrução `using`.

  No exemplo a seguir, a instrução `try-finally` chama apenas `Dispose` no bloco `finally`.

  C#

  ```
  Font bodyStyle = new Font("Arial", 10.0f);
  try
  {
      byte charset = bodyStyle.GdiCharSet;
  }
  finally
  {
      bodyStyle?.Dispose();
  }
  ```

  Você pode fazer a mesma coisa com uma instrução `using`.

  C#

  ```
  using (Font arial = new Font("Arial", 10.0f))
  {
      byte charset2 = arial.GdiCharSet;
  }
  ```

  Use a nova sintaxe using que não requer chaves:

  C#

  ```
  using Font normalStyle = new Font("Arial", 10.0f);
  byte charset3 = normalStyle.GdiCharSet;
  ```

Operadores `&&` e `||`

- Use &&& em vez de & e || em vez de | quando executar comparações, conforme mostrado no exemplo a seguir.

  C#

```
Console.Write("Enter a dividend: ");
int dividend = Convert.ToInt32(Console.ReadLine());

Console.Write("Enter a divisor: ");
int divisor = Convert.ToInt32(Console.ReadLine());

if ((divisor != 0) && (dividend / divisor) is var result)
{
    Console.WriteLine("Quotient: {0}", result);
}
else
{
    Console.WriteLine("Attempted division by 0 ends up here.");
}
```

Se o divisor for 0, a segunda cláusula na instrução `if` causará um erro em tempo de execução. Mas o operador && entra em curto-circuito quando a primeira expressão é falsa. Ou seja, ele não avalia a segunda expressão. O operador & avalia ambas, resultando em um erro em tempo de execução quando `divisor` é 0.

Operador `new`

- Use uma das formas concisas de instanciação de objeto quando o tipo de variável corresponder ao tipo de objeto, conforme mostrado nas declarações a seguir. Esse formulário não é válido quando a variável é um tipo de interface ou uma classe base do tipo de runtime.

  C#

  ```
  var firstExample = new ExampleClass();
  ```

  C#

  ```
  ExampleClass instance2 = new();
  ```

  As declarações anteriores são equivalentes à declaração a seguir.

  C#

  ```
  ExampleClass secondExample = new ExampleClass();
  ```

- Use inicializadores de objeto para simplificar a criação de objeto, conforme mostrado no exemplo a seguir.

```C#
var thirdExample = new ExampleClass { Name = "Desktop", ID = 37414,
    Location = "Redmond", Age = 2.3 };
```

O exemplo a seguir define as mesmas propriedades do exemplo anterior, mas não usa inicializadores.

```C#
var fourthExample = new ExampleClass();
fourthExample.Name = "Desktop";
fourthExample.ID = 37414;
fourthExample.Location = "Redmond";
fourthExample.Age = 2.3;
```

Manipulação de eventos

- Use uma expressão lambda para definir um manipulador de eventos que você não precisa remover mais tarde:

```C#
public Form2()
{
    this.Click += (s, e) =>
        {
            MessageBox.Show(
                ((MouseEventArgs)e).Location.ToString());
        };
}
```

A expressão lambda reduz a definição convencional a seguir.

```C#
public Form1()
{
    this.Click += new EventHandler(Form1_Click);
}

void Form1_Click(object? sender, EventArgs e)
{
    MessageBox.Show(((MouseEventArgs)e).Location.ToString());
}
```

Membros estáticos

Chame membros estáticos usando o nome de classe: *ClassName.StaticMember*. Essa prática torna o código mais legível, tornando o acesso estático limpo. Não qualifique um membro estático definido em uma classe base com o nome de uma classe derivada. Embora esse código seja compilado, a legibilidade do código será enganosa e o código poderá ser interrompido no futuro se você adicionar um membro estático com o mesmo nome à classe derivada.

Consultas LINQ

- Use nomes significativos para variáveis de consulta. O exemplo a seguir usa `seattleCustomers` para os clientes que estão localizados em Seattle.

  C#

  ```
  var seattleCustomers = from customer in Customers
                         where customer.City == "Seattle"
                         select customer.Name;
  ```

- Use aliases para se certificar de que os nomes de propriedades de tipos anônimos sejam colocados corretamente em maiúsculas, usando o padrão Pascal-Case.

  C#

  ```
  var localDistributors =
      from customer in Customers
      join distributor in Distributors on customer.City equals
  distributor.City
      select new { Customer = customer, Distributor = distributor };
  ```

- Renomeie propriedades quando os nomes de propriedades no resultado forem ambíguos. Por exemplo, se a consulta retornar um nome de cliente e um nome de distribuidor, em vez de deixá-los como uma forma de `Name` no resultado, renomeie-os para esclarecer `CustomerName` é o nome de um cliente e `DistributorName` é o nome de um distribuidor.

  C#

  ```
  var localDistributors2 =
      from customer in Customers
      join distributor in Distributors on customer.City equals
  distributor.City
  ```

```
    select new { CustomerName = customer.Name, DistributorName =
distributor.Name };
```

- Usa a digitação implícita na declaração de variáveis de consulta e de intervalo. Essa orientação sobre digitação implícita em consultas LINQ substitui as regras gerais para variáveis locais de tipo implícito. As consultas LINQ geralmente usam projeções que criam tipos anônimos. Outras expressões de consulta criam resultados com tipos genéricos aninhados. Variáveis de tipo implícito geralmente são mais legíveis.

```C#
var seattleCustomers = from customer in Customers
                       where customer.City == "Seattle"
                       select customer.Name;
```

- Alinhe as cláusulas de consulta na cláusula from, conforme mostrado nos exemplos anteriores.

- Use as cláusulas where antes de outras cláusulas de consulta, para garantir que as cláusulas de consulta posteriores operem no conjunto de dados filtrado e reduzido.

```C#
var seattleCustomers2 = from customer in Customers
                        where customer.City == "Seattle"
                        orderby customer.Name
                        select customer;
```

- Acesse coleções internas com várias cláusulas `from`, em vez de uma cláusula join. Por exemplo, cada coleção de objetos `Student` pode conter um conjunto de pontuações no teste. Quando a consulta a seguir é executada, ela retorna cada pontuação acima de 90, juntamente com o nome da família do aluno que recebeu a pontuação.

```C#
var scoreQuery = from student in students
                 from score in student.Scores
                 where score > 90
                 select new { Last = student.LastName, score };
```

Variáveis locais de tipo implícito

- Use a digitação implícita para variáveis locais quando o tipo da variável for óbvio do lado direito da atribuição.

```
C#

var message = "This is clearly a string.";
var currentTemperature = 27;
```

- Não use var quando o tipo não for aparente do lado direito da atribuição. Não suponha que o tipo esteja claro partir do nome de um método. Um tipo de variável é considerado claro se for um operador `new`, uma conversão explícita ou uma atribuição a um valor literal.

```
C#

int numberOfIterations = Convert.ToInt32(Console.ReadLine());
int currentMaximum = ExampleClass.ResultSoFar();
```

- Não use nomes de variáveis para especificar o tipo da variável. Ele pode não estar correto. Em vez disso, use o tipo para especificar o tipo e use o nome da variável para indicar as informações semânticas da variável. O exemplo a seguir deve usar `string` para o tipo e algo como `iterations` indicar o significado das informações lidas do console.

```
C#

var inputInt = Console.ReadLine();
Console.WriteLine(inputInt);
```

- Evite o uso de `var` em vez de dynamic. Use `dynamic` quando quiser uma inferência de tipo em tempo de execução. Para obter mais informações, confira Como usar o tipo dinâmico (Guia de Programação do C#).

- Use a digitação implícita para a variável de loop em loops for.

  O exemplo a seguir usa digitação implícita em uma instrução `for`.

```
C#

var phrase =
"lalalalalalalalalalalalalalalalalalalalalalalalalalalalalalalalalala";
var manyPhrases = new StringBuilder();
```

```
for (var i = 0; i < 10000; i++)
{
    manyPhrases.Append(phrase);
}
//Console.WriteLine("tra" + manyPhrases);
```

- Não use a digitação implícita para determinar o tipo da variável em loop nos loops foreach. Na maioria dos casos, o tipo de elementos na coleção não é imediatamente evidente. O nome da coleção não deve ser usado apenas para inferir o tipo dos elementos.

  O exemplo a seguir usa digitação explícita em uma instrução `foreach`.

  C#

  ```
  foreach (char ch in laugh)
  {
      if (ch == 'h')
      {
          Console.Write("H");
      }
      else
      {
          Console.Write(ch);
      }
  }
  Console.WriteLine();
  ```

- use o tipo implícito para as sequências de resultados em consultas LINQ. A seção em LINQ explica que muitas consultas LINQ resultam em tipos anônimos em que tipos implícitos devem ser usados. Outras consultas resultam em tipos genéricos aninhados em `var` que é mais legível.

  > ⓘ **Observação**
  >
  > Tenha cuidado para não alterar acidentalmente um tipo de elemento da coleção iterável. Por exemplo, é fácil alternar de System.Linq.IQueryable para System.Collections.IEnumerable em uma instrução `foreach`, o que altera a execução de uma consulta.

Alguns de nossos exemplos explicam o *tipo* natural de uma expressão. Esses exemplos devem usar `var` para que o compilador escolha o tipo natural. Embora esses exemplos sejam menos óbvios, o uso de `var` é necessário para o exemplo. O texto deve explicar o comportamento.

Declarações de namespace com escopo de arquivo

A maioria dos arquivos de código declara um único namespace. Portanto, nossos exemplos devem usar as declarações de namespace com escopo de arquivo:

C#

```
namespace MySampleCode;
```

Coloque as diretivas “using” fora da declaração de namespace

Quando uma diretiva `using` está fora de uma declaração de namespace, esse namespace importado é seu nome totalmente qualificado. O nome totalmente qualificado é mais claro. Quando a diretiva `using` está dentro do namespace, ela pode ser relativa a esse namespace ou ao seu nome totalmente qualificado.

C#

```
using Azure;

namespace CoolStuff.AwesomeFeature
{
    public class Awesome
    {
        public void Stuff()
        {
            WaitUntil wait = WaitUntil.Completed;
            // ...
        }
    }
}
```

Supondo que haja uma referência (direta ou indireta) à classe WaitUntil.

Agora, vamos fazer uma pequena alteração:

C#

```
namespace CoolStuff.AwesomeFeature
{
    using Azure;

    public class Awesome
    {
        public void Stuff()
        {
```

```
            WaitUntil wait = WaitUntil.Completed;
            // ...
        }
    }
}
```

E ela será compilada hoje. E amanhã. Mas, em algum momento da próxima semana, o código anterior (intocado) falha com dois erros:

Console

```
- error CS0246: The type or namespace name 'WaitUntil' could not be found
(are you missing a using directive or an assembly reference?)
- error CS0103: The name 'WaitUntil' does not exist in the current context
```

Uma das dependências introduzidas nessa classe em um namespace termina com `.Azure`:

C#

```
namespace CoolStuff.Azure
{
    public class SecretsManagement
    {
        public string FetchFromKeyVault(string vaultId, string secretId) {
return null; }
    }
}
```

Uma diretiva `using` colocada dentro de um namespace diferencia contexto e complica a resolução do nome. Neste exemplo, é o primeiro namespace que ela encontra.

- `CoolStuff.AwesomeFeature.Azure`
- `CoolStuff.Azure`
- `Azure`

Adicionar um novo namespace que corresponda a `CoolStuff.Azure` ou `CoolStuff.AwesomeFeature.Azure` seria combinado antes do namespace `Azure` global. Você poderia resolver isso adicionando o modificador `global::` à declaração `using`. No entanto, é mais fácil colocar declarações `using` fora do namespace.

C#

```
namespace CoolStuff.AwesomeFeature
{
    using global::Azure;
```

```
    public class Awesome
    {
        public void Stuff()
        {
            WaitUntil wait = WaitUntil.Completed;
            // ...
        }
    }
}
```

Diretrizes de estilo

Em geral, use o seguinte formato para exemplos de código:

- Use quatro espaços para recuo. Não use guias.
- Alinhe o código consistentemente para melhorar a legibilidade.
- Limite linhas a 65 caracteres para aprimorar a legibilidade de código em documentos, especialmente em telas de dispositivos móveis.
- Melhore a clareza e a experiência do usuário dividindo instruções longas em várias linhas.
- Use o estilo "Allman" para chaves: abra e feche sua própria nova linha. As chaves se alinham com o nível de recuo atual.
- As quebras de linha devem ocorrer antes dos operadores binários, se necessário.

Estilo de comentário

- Use comentários de linha única (`//`) para breves explicações.

- Evite comentários de várias linhas (`/* */`) para explicações mais longas.
  Os comentários nos exemplos de código não são localizados. Isso significa que as explicações inseridas no código não são traduzidas. Um texto explicativo mais longo deve ser colocado no artigo complementar, para que possa ser localizado.

- Para descrever métodos, classes, campos e todos os membros públicos, use comentários XML.

- Coloque o comentário em uma linha separada, não no final de uma linha de código.

- Comece o texto do comentário com uma letra maiúscula.

- Termine o texto do comentário com um ponto final.

- Insira um espaço entre o delimitador de comentário (`//`) e o texto do comentário, conforme mostrado no exemplo a seguir.

  C#

  ```
  // The following declaration creates a query. It does not run
  // the query.
  ```

Convenções de layout

Um bom layout usa formatação para enfatizar a estrutura do código e para facilitar a leitura de código. Exemplos e amostras Microsoft estão em conformidade com as seguintes convenções:

- Use as configurações padrão do Editor de códigos (recuo inteligente, recuos de quatro caracteres, guias salvas como espaços). Para obter mais informações, consulte Opções, Editor de Texto, C#, Formatação.
- Gravar apenas uma instrução por linha.
- Gravar apenas uma declaração por linha.
- Se as linhas de continuação não forem recuadas automaticamente, recue-as uma parada de tabulação (quatro espaços).
- Adicione pelo menos uma linha em branco entre as definições de método e de propriedade.
- Use parênteses para criar cláusulas em uma expressão aparente, conforme mostrado no código a seguir.

  C#

  ```
  if ((startX > endX) && (startX > previousX))
  {
      // Take appropriate action.
  }
  ```

As exceções são quando o exemplo explica o operador ou a precedência de expressão.

Segurança

Siga as diretrizes em Diretrizes de codificação segura.

Como exibir argumentos de linha de comando

Artigo • 08/06/2023

Os argumentos fornecidos a um executável na linha de comando são acessíveis em instruções de nível superior ou por meio de um parâmetro opcional para `Main`. Os argumentos são fornecidos na forma de uma matriz de cadeias de caracteres. Cada elemento da matriz contém um argumento. O espaço em branco entre os argumentos é removido. Por exemplo, considere essas invocações de linha de comando de um executável fictício:

Entrada na linha de comando	Matriz de cadeias de caracteres passada a Main
executável.exe a b c	"a" "b" "c"
executável.exe um dois	"um" "dois"
executável.exe "um dois" três	"one two" "three"

> **Observação**
>
> Quando estiver executando um aplicativo no Visual Studio, você pode especificar argumentos de linha de comando na **Página de depuração, Designer de Projeto**.

Exemplo

Este exemplo exibe os argumentos de linha de comando passados a um aplicativo de linha de comando. A saída mostrada é para a primeira entrada da tabela acima.

C#

```
// The Length property provides the number of array elements.
Console.WriteLine($"parameter count = {args.Length}");
```

```
for (int i = 0; i < args.Length; i++)
{
    Console.WriteLine($"Arg[{i}] = [{args[i]}]");
}

/* Output (assumes 3 cmd line args):
    parameter count = 3
    Arg[0] = [a]
    Arg[1] = [b]
    Arg[2] = [c]
*/
```

Confira também

- Visão geral de System.CommandLine
- Tutorial: introdução a System.CommandLine

Explorar programação orientada a objeto com classes e objetos

17/06/2025

Neste tutorial, você vai compilar um aplicativo de console e ver os recursos básicos orientados a objeto que fazem parte da linguagem C#.

Pré-requisitos

- O .NET SDK mais recente
- Editor do Visual Studio Code
- O DevKit C#

Instruções de instalação

No Windows, esse Arquivo de configuração do WinGet para instalar todos os pré-requisitos. Se você já tiver algo instalado, o WinGet ignorará essa etapa.

1. Baixe o arquivo e clique duas vezes para executá-lo.
2. Leia o contrato de licença, digite `y` e selecione `Enter` quando solicitado a aceitar.
3. Se você receber um prompt de UAC (Controle de Conta de Usuário) piscando na barra de tarefas, permita que a instalação continue.

Em outras plataformas, você precisa instalar cada um desses componentes separadamente.

1. Baixe o instalador recomendado na página de download do SDK do .NET e clique duas vezes para executá-lo. A página de download detecta sua plataforma e recomenda o instalador mais recente para sua plataforma.
2. Baixe o instalador mais recente do Visual Studio Code home page e clique duas vezes para executá-lo. Essa página também detecta sua plataforma e o link deve estar correto para o sistema.
3. Clique no botão "Instalar" na página de extensão do DevKit em C#. Isso abre o código do Visual Studio e pergunta se você deseja instalar ou habilitar a extensão. Selecione "instalar".

Criar o aplicativo

Usando uma janela de terminal, crie um diretório chamado *Classes*. Você compilará o aplicativo nesse diretório. Altere para esse diretório e digite `dotnet new console` na janela do console.

Esse comando cria o aplicativo. Abra *Program.cs*. Ele deverá ser parecido com:

```C#
// See https://aka.ms/new-console-template for more information
Console.WriteLine("Hello, World!");
```

Neste tutorial, você criará novos tipos que representam uma conta bancária. Normalmente, os desenvolvedores definem cada classe em um arquivo de texto diferente. Isso facilita o gerenciamento à medida que o tamanho do programa aumenta. Crie um novo arquivo chamado *BankAccount.cs* no diretório *Classes*.

Esse arquivo conterá a definição de uma ***conta bancária***. A programação Orientada a Objetos organiza o código por meio da criação de tipos na forma de ***classes***. Essas classes contêm o código que representa uma entidade específica. A classe `BankAccount` representa uma conta bancária. O código implementa operações específicas por meio de métodos e propriedades. Neste tutorial, a conta bancária dá suporte a este comportamento:

1. Ela tem um número com 10 dígitos que identifica exclusivamente a conta bancária.
2. Ela tem uma cadeia de caracteres que armazena o nome ou os nomes dos proprietários.
3. O saldo pode ser recuperado.
4. Ela aceita depósitos.
5. Ele aceita saques.
6. O saldo inicial deve ser positivo.
7. Os saques não podem resultar em um saldo negativo.

Definir o tipo de conta bancária

Você pode começar criando as noções básicas de uma classe que define esse comportamento. Crie um novo arquivo usando o comando **File:New**. Nomeie-o *bankAccount.cs*. Adicione o seguinte código ao arquivo *BankAccount.cs*:

```C#
namespace Classes;

public class BankAccount
{
    public string Number { get; }
    public string Owner { get; set; }
    public decimal Balance { get; }

    public void MakeDeposit(decimal amount, DateTime date, string note)
    {
    }
```

```
    public void MakeWithdrawal(decimal amount, DateTime date, string note)
    {
    }
}
```

Antes de continuar, vamos dar uma olhada no que você compilou. A declaração `namespace` fornece uma maneira de organizar logicamente seu código. Este tutorial é relativamente pequeno, portanto, você colocará todo o código em um namespace.

`public class BankAccount` define a classe ou o tipo que você está criando. Tudo que vem dentro de `{` e `}` logo após a declaração da classe define o estado e o comportamento da classe. Há cinco ***membros*** na classe `BankAccount`. Os três primeiros são ***propriedades***. Propriedades são elementos de dados que podem ter um código que impõe a validação ou outras regras. Os dois últimos são ***métodos***. Os métodos são blocos de código que executam uma única função. A leitura dos nomes de cada um dos membros deve fornecer informações suficientes para você, ou outro desenvolvedor, entender o que a classe faz.

Abrir uma nova conta

O primeiro recurso a ser implementado serve para abrir uma conta bancária. Quando um cliente abre uma conta, ele deve fornecer um saldo inicial e informações sobre o proprietário, ou proprietários, dessa conta.

A criação de novo objeto do tipo `BankAccount` significa a definição de um ***construtor*** que atribui esses valores. Um ***construtor*** é um membro que tem o mesmo nome da classe. Ele é usado para inicializar objetos desse tipo de classe. Adicione o seguinte construtor ao tipo `BankAccount`. Insira o código a seguir acima da declaração de `MakeDeposit`:

C#

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

O código anterior identifica as propriedades do objeto que está sendo construído, incluindo o qualificador `this`. Esse qualificador geralmente é opcional e omitido. Você também poderia ter escrito:

C#

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

O qualificador `this` só é necessário quando uma variável ou parâmetro locais têm o mesmo nome desse campo ou propriedade. O qualificador `this` é omitido durante todo o restante deste artigo, a menos que seja necessário.

Construtores são chamados quando você cria um objeto usando new. Substitua a linha `Console.WriteLine("Hello World!");` no arquivo *Program.cs* pelo seguinte código (substitua `<name>` pelo seu nome):

C#

```
using Classes;

var account = new BankAccount("<name>", 1000);
Console.WriteLine($"Account {account.Number} was created for {account.Owner} with
{account.Balance} initial balance.");
```

Vamos executar o que você construiu até agora. Se você estiver usando o Visual Studio, selecione **Iniciar sem depuração** no menu **Depurar**. Se estiver usando uma linha de comando, digite `dotnet run` no diretório em que criou seu projeto.

Você notou que o número da conta está em branco? É hora de corrigir isso. O número da conta deve ser atribuído na construção do objeto. Mas não é responsabilidade do chamador criá-lo. O código da classe `BankAccount` deve saber como atribuir novos números de conta. Uma maneira simples de fazer isso é começar com um número de 10 dígitos. Incremente-o à medida que novas contas são criadas. Por fim, armazene o número da conta atual quando um objeto for construído.

Adicione uma declaração de membro à classe `BankAccount`. Coloque a seguinte linha de código após a chave de abertura `{` no início da classe `BankAccount`:

C#

```
private static int s_accountNumberSeed = 1234567890;
```

O `accountNumberSeed` é um membro de dados. Ele é `private`, o que significa que ele só pode ser acessado pelo código dentro da classe `BankAccount`. É uma maneira de separar as responsabilidades públicas (como ter um número de conta) da implementação privada (como

os números de conta são gerados). Ele também é `static`, o que significa que é compartilhado por todos os objetos `BankAccount`. O valor de uma variável não estática é exclusivo para cada instância do objeto `BankAccount`. O `accountNumberSeed` é um campo `private static` e, portanto, tem o prefixo `s_` de acordo com as convenções de nomenclatura do C#. O campo de `s` denotação `static` e de `_` denotação `private`. Adicione as duas linhas a seguir ao construtor para atribuir o número da conta. Coloque-as depois da linha que diz `this.Balance = initialBalance`:

```
C#

Number = s_accountNumberSeed.ToString();
s_accountNumberSeed++;
```

Digite `dotnet run` para ver os resultados.

Criar depósitos e saques

A classe da conta bancária precisa aceitar depósitos e saques para funcionar corretamente. Vamos implementar depósitos e saques criando um diário de todas as transações da conta. Rastrear todas as transações tem algumas vantagens em comparação à simples atualização do saldo em cada transação. O histórico pode ser usado para auditar todas as transações e gerenciar os saldos diários. Calcular o saldo do histórico de todas as transações quando necessário assegura que todos os erros corrigidos em uma única transação serão refletidos corretamente no saldo no próximo cálculo.

Vamos começar criando um novo tipo para representar uma transação. Essa transação é um tipo simples que não tem qualquer responsabilidade. Ele precisa de algumas propriedades. Crie um novo arquivo chamado *Transaction.cs*. Adicione os seguintes códigos a ela:

```
C#

namespace Classes;

public class Transaction
{
    public decimal Amount { get; }
    public DateTime Date { get; }
    public string Notes { get; }

    public Transaction(decimal amount, DateTime date, string note)
    {
        Amount = amount;
        Date = date;
        Notes = note;
```

```
    }
}
```

Agora, vamos adicionar um List<T> de `Transaction` objetos à classe `BankAccount`. Adicione a seguinte declaração após o construtor no arquivo *BankAccount.cs*:

```C#
private List<Transaction> _allTransactions = new List<Transaction>();
```

Agora, vamos calcular corretamente o `Balance`. O saldo atual pode ser encontrado somando os valores de todas as transações. De acordo com a forma atual do código, você só pode obter o saldo inicial da conta. Portanto, você terá que atualizar a propriedade `Balance`. Substitua a linha `public decimal Balance { get; }` em *BankAccount.cs* pelo seguinte código:

```C#
public decimal Balance
{
    get
    {
        decimal balance = 0;
        foreach (var item in _allTransactions)
        {
            balance += item.Amount;
        }

        return balance;
    }
}
```

Este exemplo mostra um aspecto importante das ***propriedades***. Agora, você está calculando o saldo quando outro programador solicita o valor. Seu cálculo enumera todas as transações e fornece a soma como o saldo atual.

Depois, implemente os métodos `MakeDeposit` e `MakeWithdrawal`. Esses métodos vão impor as duas últimas regras: que o saldo inicial deve ser positivo, e que qualquer saque não pode criar um saldo negativo.

Essas regras introduzem o conceito de ***exceções***. A forma padrão de indicar que um método não pode concluir seu trabalho com êxito é lançar uma exceção. O tipo de exceção e a mensagem associada a ele descrevem o erro. Aqui, o método `MakeDeposit` lançará uma exceção se o valor do depósito for menor ou igual a 0. O método `MakeWithdrawal` lançará uma exceção se o valor do saque for menor ou igual a 0 ou se a aplicação do saque resultar em um saldo negativo. Adicione o seguinte código depois da declaração da lista `_allTransactions`:

C#

```
public void MakeDeposit(decimal amount, DateTime date, string note)
{
    if (amount <= 0)
    {
        throw new ArgumentOutOfRangeException(nameof(amount), "Amount of deposit
must be positive");
    }
    var deposit = new Transaction(amount, date, note);
    _allTransactions.Add(deposit);
}

public void MakeWithdrawal(decimal amount, DateTime date, string note)
{
    if (amount <= 0)
    {
        throw new ArgumentOutOfRangeException(nameof(amount), "Amount of
withdrawal must be positive");
    }
    if (Balance - amount < 0)
    {
        throw new InvalidOperationException("Not sufficient funds for this
withdrawal");
    }
    var withdrawal = new Transaction(-amount, date, note);
    _allTransactions.Add(withdrawal);
}
```

A throw instrução**gera** uma exceção. A execução do bloco atual é encerrada e o controle transferido para o bloco `catch` da primeira correspondência encontrado na pilha de chamadas. Você adicionará um bloco `catch` para testar esse código um pouco mais tarde.

O construtor deve receber uma alteração para que adicione uma transação inicial, em vez de atualizar o saldo diretamente. Como você já escreveu o método `MakeDeposit`, chame-o de seu construtor. O construtor concluído deve ter esta aparência:

C#

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

    Owner = name;
    MakeDeposit(initialBalance, DateTime.Now, "Initial balance");
}
```

DateTime.Now é uma propriedade que retorna a data e a hora atuais. Teste esse código adicionando alguns depósitos e saques em seu método `Main`, seguindo o código que cria um novo `BankAccount`:

C#

```
account.MakeWithdrawal(500, DateTime.Now, "Rent payment");
Console.WriteLine(account.Balance);
account.MakeDeposit(100, DateTime.Now, "Friend paid me back");
Console.WriteLine(account.Balance);
```

Em seguida, teste se você está recebendo condições de erro ao tentar criar uma conta com um saldo negativo. Adicione o seguinte código após o código anterior que você acabou de adicionar:

C#

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

Use a instrução try-catch para marcar um bloco de código que possa gerar exceções e detectar esses erros esperados. Use a mesma técnica a fim de testar o código que gera uma exceção para um saldo negativo. Adicione o seguinte código antes da declaração de `invalidAccount` no seu método `Main`:

C#

```
// Test for a negative balance.
try
{
    account.MakeWithdrawal(750, DateTime.Now, "Attempt to overdraw");
}
catch (InvalidOperationException e)
{
    Console.WriteLine("Exception caught trying to overdraw");
    Console.WriteLine(e.ToString());
}
```

Salve o arquivo e digite `dotnet run` para testá-lo.

Desafio – registrar em log todas as transações

Para concluir este tutorial, escreva o método `GetAccountHistory` que cria um `string` para o histórico de transações. Adicione esse método ao tipo `BankAccount`:

C#

```
public string GetAccountHistory()
{
    var report = new System.Text.StringBuilder();

    decimal balance = 0;
    report.AppendLine("Date\t\tAmount\tBalance\tNote");
    foreach (var item in _allTransactions)
    {
        balance += item.Amount;
        report.AppendLine($"
{item.Date.ToShortDateString()}\t{item.Amount}\t{balance}\t{item.Notes}");
    }

    return report.ToString();
}
```

O histórico usa a classe StringBuilder para formatar uma cadeia de caracteres que contém uma linha para cada transação. Você viu o código de formatação da cadeia de caracteres anteriormente nesses tutoriais. Um caractere novo é `\t`. Ele insere uma guia para formatar a saída.

Adicione esta linha para testá-la no *Program.cs*:

C#

```
Console.WriteLine(account.GetAccountHistory());
```

Execute o programa para ver os resultados.

Próximas etapas

Se você não conseguir avançar, veja a origem deste tutorial em nosso repositório GitHub.

Você pode continuar com o tutorial de programação orientada a objeto.

Saiba mais sobre esses conceitos nestes artigos:

- Instruções de seleção
- Instruções de iteração

programação Object-Oriented (C#)

19/06/2025

C# é uma linguagem de programação orientada a objeto. Os quatro princípios básicos da programação orientada a objetos são:

- *Abstração* Modelando os atributos e interações relevantes de entidades como classes para definir uma representação abstrata de um sistema.
- *Encapsulação* Ocultando o estado interno e a funcionalidade de um objeto e permitindo apenas o acesso por meio de um conjunto público de funções.
- *Herança* Capacidade de criar novas abstrações com base em abstrações existentes.
- *Polimorfismo* Capacidade de implementar propriedades ou métodos herdados de diferentes maneiras em várias abstrações.

No tutorial anterior, introdução às classes, você viu tanto *abstração* quanto *encapsulamento*. A `BankAccount` classe forneceu uma abstração para o conceito de uma conta bancária. Você pode modificar sua implementação sem afetar nenhum dos códigos que usaram a `BankAccount` classe. Ambas as classes `BankAccount` e `Transaction` fornecem o encapsulamento dos componentes necessários para descrever esses conceitos no código.

Neste tutorial, você estenderá esse aplicativo para usar *herança* e *polimorfismo* para adicionar novos recursos. Você também adicionará recursos à `BankAccount` classe, aproveitando as técnicas de *abstração* e *encapsulamento* que aprendeu no tutorial anterior.

Criar diferentes tipos de contas

Depois de criar esse programa, você recebe solicitações para adicionar recursos a ele. Funciona muito bem na situação em que há apenas um tipo de conta bancária. Com o tempo, as necessidades são alteradas e os tipos de conta relacionados são solicitados:

- Uma conta de ganho de juros que acumula juros no final de cada mês.
- Uma linha de crédito que pode ter um saldo negativo, mas quando há um saldo, há uma taxa de juros a cada mês.
- Uma conta de cartão presente pré-paga que começa com um único depósito, e só pode ser paga. Ele pode ser preenchido novamente uma vez no início de cada mês.

Todas essas contas diferentes são semelhantes à `BankAccount` classe definida no tutorial anterior. Você pode copiar esse código, renomear as classes e fazer modificações. Essa técnica funcionaria a curto prazo, mas seria mais trabalhosa ao longo do tempo. Todas as alterações seriam copiadas em todas as classes afetadas.

Em vez disso, você pode criar novos tipos de conta bancária que herdam métodos e dados da `BankAccount` classe criada no tutorial anterior. Essas novas classes podem estender a `BankAccount` classe com o comportamento específico necessário para cada tipo:

C#

```
public class InterestEarningAccount : BankAccount
{
}

public class LineOfCreditAccount : BankAccount
{
}

public class GiftCardAccount : BankAccount
{
}
```

Cada uma dessas classes *herda* o comportamento compartilhado de sua *classe base* compartilhada, a `BankAccount` classe. Escreva as implementações para funcionalidades novas e diferentes em cada uma das *classes derivadas*. Essas classes derivadas já têm todo o comportamento definido na `BankAccount` classe.

É uma boa prática criar cada nova classe em um arquivo de origem diferente. No Visual Studio, você pode clicar com o botão direito do mouse no projeto e selecionar *adicionar classe* para adicionar uma nova classe em um novo arquivo. No Visual Studio Code, selecione *Arquivo* e *Novo* para criar um novo arquivo de origem. Em qualquer ferramenta, nomeie o arquivo para corresponder à classe: *InterestEarningAccount.cs*, *LineOfCreditAccount.cs* e *GiftCardAccount.cs*.

Ao criar as classes conforme mostrado no exemplo anterior, você descobrirá que nenhuma das classes derivadas compila. Um construtor é responsável por inicializar um objeto. Um construtor de classe derivada deve inicializar a classe derivada e fornecer instruções sobre como inicializar o objeto de classe base incluído na classe derivada. A inicialização adequada normalmente ocorre sem nenhum código extra. A `BankAccount` classe declara um construtor público com a seguinte assinatura:

C#

```
public BankAccount(string name, decimal initialBalance)
```

O compilador não gera um construtor padrão quando você define um construtor por conta própria. Isso significa que cada classe derivada deve chamar explicitamente esse construtor.

Você declara um construtor que pode passar argumentos para o construtor de classe base. O código a seguir mostra o construtor para o `InterestEarningAccount`:

```
C#

public InterestEarningAccount(string name, decimal initialBalance) : base(name,
initialBalance)
{
}
```

Os parâmetros para esse novo construtor correspondem ao tipo de parâmetro e aos nomes do construtor de classe base. Você usa a sintaxe `: base()` para indicar uma chamada para um construtor de classe base. Algumas classes definem vários construtores e essa sintaxe permite que você escolha qual construtor de classe base você chama. Depois de atualizar os construtores, você pode desenvolver o código para cada uma das classes derivadas. Os requisitos para as novas classes podem ser declarados da seguinte maneira:

- Uma conta de ganho de juros:
  - Obterá um crédito de 2% do saldo final do mês.
- Uma linha de crédito:
  - Pode ter um saldo negativo, mas não ser maior em valor absoluto do que o limite de crédito.
  - Incorrerá em uma cobrança de juros a cada mês em que o saldo de fim de mês não seja 0.
  - Incorrerá em uma taxa em cada retirada que ultrapassa o limite de crédito.
- Uma conta de cartão-presente
  - Pode ser preenchido novamente com um valor especificado uma vez por mês, no último dia do mês.

Você pode ver que todos esses três tipos de conta têm uma ação que ocorre no final de cada mês. No entanto, cada tipo de conta faz tarefas diferentes. Você usa *o polimorfismo* para implementar esse código. Crie um único `virtual` método na `BankAccount` classe:

```
C#

public virtual void PerformMonthEndTransactions() { }
```

O código anterior mostra como você usa a `virtual` palavra-chave para declarar um método na classe base para o qual uma classe derivada pode fornecer uma implementação diferente. Um `virtual` método é um método em que qualquer classe derivada pode optar por reimplementar. As classes derivadas usam a `override` palavra-chave para definir a nova implementação. Normalmente, você se refere a isso como "substituindo a implementação da

classe base". A `virtual` palavra-chave especifica que as classes derivadas podem substituir o comportamento. Você também pode declarar `abstract` métodos em que as classes derivadas devem substituir o comportamento. A classe base não fornece uma implementação para um `abstract` método. Em seguida, você precisa definir a implementação para duas das novas classes que você criou. Comece com : `InterestEarningAccount`

C#

```
public override void PerformMonthEndTransactions()
{
    if (Balance > 500m)
    {
        decimal interest = Balance * 0.02m;
        MakeDeposit(interest, DateTime.Now, "apply monthly interest");
    }
}
```

Adicione o seguinte código ao `LineOfCreditAccount`. O código nega o saldo para calcular uma taxa de juros positiva que é retirada da conta:

C#

```
public override void PerformMonthEndTransactions()
{
    if (Balance < 0)
    {
        // Negate the balance to get a positive interest charge:
        decimal interest = -Balance * 0.07m;
        MakeWithdrawal(interest, DateTime.Now, "Charge monthly interest");
    }
}
```

A `GiftCardAccount` classe precisa de duas alterações para implementar sua funcionalidade de fim de mês. Primeiro, modifique o construtor para incluir um valor opcional a ser adicionado a cada mês:

C#

```
private readonly decimal _monthlyDeposit = 0m;

public GiftCardAccount(string name, decimal initialBalance, decimal monthlyDeposit
= 0) : base(name, initialBalance)
    => _monthlyDeposit = monthlyDeposit;
```

O construtor fornece um valor padrão para o valor `monthlyDeposit` para que os chamadores possam omitir um `0` sem depósito mensal. Em seguida, substitua o método

`PerformMonthEndTransactions` para adicionar o depósito mensal, caso tenha sido definido como um valor não zero no construtor.

C#

```
public override void PerformMonthEndTransactions()
{
    if (_monthlyDeposit != 0)
    {
        MakeDeposit(_monthlyDeposit, DateTime.Now, "Add monthly deposit");
    }
}
```

A substituição aplica o conjunto de depósito mensal no construtor. Adicione o código a seguir ao método `Main` para testar essas alterações para o `GiftCardAccount` e o `InterestEarningAccount`.

C#

```
var giftCard = new GiftCardAccount("gift card", 100, 50);
giftCard.MakeWithdrawal(20, DateTime.Now, "get expensive coffee");
giftCard.MakeWithdrawal(50, DateTime.Now, "buy groceries");
giftCard.PerformMonthEndTransactions();
// can make additional deposits:
giftCard.MakeDeposit(27.50m, DateTime.Now, "add some additional spending money");
Console.WriteLine(giftCard.GetAccountHistory());

var savings = new InterestEarningAccount("savings account", 10000);
savings.MakeDeposit(750, DateTime.Now, "save some money");
savings.MakeDeposit(1250, DateTime.Now, "Add more savings");
savings.MakeWithdrawal(250, DateTime.Now, "Needed to pay monthly bills");
savings.PerformMonthEndTransactions();
Console.WriteLine(savings.GetAccountHistory());
```

Verifique os resultados. Agora, adicione um conjunto semelhante de código de teste para :`LineOfCreditAccount`

C#

```
var lineOfCredit = new LineOfCreditAccount("line of credit", 0);
// How much is too much to borrow?
lineOfCredit.MakeWithdrawal(1000m, DateTime.Now, "Take out monthly advance");
lineOfCredit.MakeDeposit(50m, DateTime.Now, "Pay back small amount");
lineOfCredit.MakeWithdrawal(5000m, DateTime.Now, "Emergency funds for repairs");
lineOfCredit.MakeDeposit(150m, DateTime.Now, "Partial restoration on repairs");
lineOfCredit.PerformMonthEndTransactions();
Console.WriteLine(lineOfCredit.GetAccountHistory());
```

Ao adicionar o código anterior e executar o programa, você verá algo parecido com o seguinte erro:

Console

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

Observação

A saída real inclui o caminho completo para a pasta com o projeto. Os nomes das pastas foram omitidos por brevidade. Além disso, dependendo do formato de código, os números de linha podem ser ligeiramente diferentes.

Esse código falha porque pressupõe `BankAccount` que o saldo inicial deve ser maior que 0. Outra suposição feita na `BankAccount` classe é que o saldo não pode ficar negativo. Em vez disso, qualquer retirada que sobrecarrega a conta é rejeitada. Ambas as suposições precisam mudar. A linha de conta de crédito começa em 0 e geralmente terá um saldo negativo. Além disso, se um cliente empresta muito dinheiro, ele incorre em uma taxa. A transação é aceita, apenas custa mais. A primeira regra pode ser implementada adicionando um argumento opcional ao `BankAccount` construtor que especifica o saldo mínimo. O padrão é `0`. A segunda regra requer um mecanismo que permite que classes derivadas modifiquem o algoritmo padrão. De certa forma, a classe base "pergunta" ao tipo derivado o que deve acontecer quando há um cheque especial. O comportamento padrão é rejeitar a transação lançando uma exceção.

Vamos começar adicionando um segundo construtor que inclui um parâmetro opcional `minimumBalance` . Esse novo construtor faz todas as ações feitas pelo construtor existente. Além disso, ele define a propriedade de saldo mínimo. Você pode copiar o corpo do construtor existente, mas isso significa dois locais a serem alterados no futuro. Em vez disso, você pode usar o *encadeamento de construtor* para que um construtor chame outro. O código a seguir mostra os dois construtores e o novo campo adicional:

C#

```
private readonly decimal _minimumBalance;

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

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

    Owner = name;
    _minimumBalance = minimumBalance;
    if (initialBalance > 0)
        MakeDeposit(initialBalance, DateTime.Now, "Initial balance");
}
```

O código anterior mostra duas novas técnicas. Primeiro, o `minimumBalance` campo é marcado como `readonly`. Isso significa que o valor não pode ser alterado depois que o objeto é construído. Depois que um `BankAccount` é criado, o `minimumBalance` não pode ser alterado. Em segundo lugar, o construtor que recebe dois parâmetros tem `: this(name, initialBalance, 0) { }` como sua implementação. A `: this()` expressão chama o outro construtor, aquele com três parâmetros. Essa técnica permite que você tenha uma única implementação para inicializar um objeto, embora o código do cliente possa escolher um dos muitos construtores.

Essa implementação chamará `MakeDeposit` somente se o saldo inicial for maior que `0`. Isso preserva a regra de que os depósitos devem ser positivos, mas permite que a conta de crédito abra com um `0` saldo.

Agora que a `BankAccount` classe tem um campo somente leitura para o saldo mínimo, a alteração final é mudar o código rígido `0` para `minimumBalance` no método `MakeWithdrawal`.

C#

```
if (Balance - amount < _minimumBalance)
```

Depois de estender a `BankAccount` classe, você pode modificar o `LineOfCreditAccount` construtor para chamar o novo construtor base, conforme mostrado no seguinte código:

C#

```
public LineOfCreditAccount(string name, decimal initialBalance, decimal
creditLimit) : base(name, initialBalance, -creditLimit)
{
}
```

Observe que o construtor `LineOfCreditAccount` altera o sinal do parâmetro `creditLimit` para corresponder ao significado do parâmetro `minimumBalance`.

Regras de cheque especial diferentes

O último recurso a ser adicionado permite cobrar `LineOfCreditAccount` uma taxa por ultrapassar o limite de crédito em vez de recusar a transação.

Uma técnica é definir uma função virtual na qual você implementa o comportamento necessário. A `BankAccount` classe refatora o `MakeWithdrawal` método em dois métodos. O novo método executa a ação especificada quando a retirada leva o saldo abaixo do mínimo. O método existente `MakeWithdrawal` tem o seguinte código:

C#

```
public void MakeWithdrawal(decimal amount, DateTime date, string note)
{
    if (amount <= 0)
    {
        throw new ArgumentOutOfRangeException(nameof(amount), "Amount of
withdrawal must be positive");
    }
    if (Balance - amount < _minimumBalance)
    {
        throw new InvalidOperationException("Not sufficient funds for this
withdrawal");
    }
    var withdrawal = new Transaction(-amount, date, note);
    _allTransactions.Add(withdrawal);
}
```

Substitua-o pelo seguinte código:

C#

```
public void MakeWithdrawal(decimal amount, DateTime date, string note)
{
    if (amount <= 0)
    {
        throw new ArgumentOutOfRangeException(nameof(amount), "Amount of
withdrawal must be positive");
    }
    Transaction? overdraftTransaction = CheckWithdrawalLimit(Balance - amount <
_minimumBalance);
    Transaction? withdrawal = new(-amount, date, note);
    _allTransactions.Add(withdrawal);
    if (overdraftTransaction != null)
        _allTransactions.Add(overdraftTransaction);
```

```
}

protected virtual Transaction? CheckWithdrawalLimit(bool isOverdrawn)
{
    if (isOverdrawn)
    {
        throw new InvalidOperationException("Not sufficient funds for this
withdrawal");
    }
    else
    {
        return default;
    }
}
```

O método adicionado é `protected`, o que significa que ele só pode ser chamado de classes derivadas. Essa declaração impede que outros clientes chamem o método. Também é `virtual` para que as classes derivadas possam alterar o comportamento. O tipo de retorno é .`Transaction?` A `?` anotação indica que o método pode retornar `null`. Adicione a seguinte implementação no `LineOfCreditAccount` para cobrar uma taxa quando o limite de saque for excedido.

C#

```
protected override Transaction? CheckWithdrawalLimit(bool isOverdrawn) =>
    isOverdrawn
    ? new Transaction(-20, DateTime.Now, "Apply overdraft fee")
    : default;
```

A substituição retorna uma transação de taxa quando a conta é sacada. Se o saque não ultrapassar o limite, o método retornará uma `null` transação. Isso indica que não há taxa. Teste essas alterações adicionando o seguinte código ao método `Main` na `Program` classe:

C#

```
var lineOfCredit = new LineOfCreditAccount("line of credit", 0, 2000);
// How much is too much to borrow?
lineOfCredit.MakeWithdrawal(1000m, DateTime.Now, "Take out monthly advance");
lineOfCredit.MakeDeposit(50m, DateTime.Now, "Pay back small amount");
lineOfCredit.MakeWithdrawal(5000m, DateTime.Now, "Emergency funds for repairs");
lineOfCredit.MakeDeposit(150m, DateTime.Now, "Partial restoration on repairs");
lineOfCredit.PerformMonthEndTransactions();
Console.WriteLine(lineOfCredit.GetAccountHistory());
```

Execute o programa e verifique os resultados.

Resumo

Se você ficou preso, poderá ver a origem deste tutorial em nosso repositório do GitHub.

Este tutorial demonstrou muitas das técnicas usadas na programação Object-Oriented:

- Você usou *Abstração* quando definiu classes para cada um dos diferentes tipos de conta. Essas classes descreveram o comportamento desse tipo de conta.
- Você usou *Encapsulamento* quando manteve muitos detalhes `private` em cada classe.
- Você usou *Herança* ao aproveitar a implementação já criada na classe `BankAccount` para economizar código.
- Você usou *o Polimorfismo* ao criar `virtual` métodos que as classes derivadas poderiam substituir para criar um comportamento específico para esse tipo de conta.

Herança em C# e .NET

19/06/2025

Este tutorial apresenta a herança em C#. A herança é um recurso de linguagens de programação orientadas a objetos que permite definir uma classe base que fornece funcionalidade específica (dados e comportamento) e definir classes derivadas que herdam ou substituem essa funcionalidade.

Pré-requisitos

- O .NET SDK mais recente
- Editor do Visual Studio Code
- O DevKit C#

Instruções de instalação

No Windows, esse Arquivo de configuração do WinGet para instalar todos os pré-requisitos. Se você já tiver algo instalado, o WinGet ignorará essa etapa.

1. Baixe o arquivo e clique duas vezes para executá-lo.
2. Leia o contrato de licença, digite `y` e selecione `Enter` quando solicitado a aceitar.
3. Se você receber um prompt de UAC (Controle de Conta de Usuário) piscando na barra de tarefas, permita que a instalação continue.

Em outras plataformas, você precisa instalar cada um desses componentes separadamente.

1. Baixe o instalador recomendado na página de download do SDK do .NET e clique duas vezes para executá-lo. A página de download detecta sua plataforma e recomenda o instalador mais recente para sua plataforma.
2. Baixe o instalador mais recente do Visual Studio Code home page e clique duas vezes para executá-lo. Essa página também detecta sua plataforma e o link deve estar correto para o sistema.
3. Clique no botão "Instalar" na página de extensão do DevKit em C#. Isso abre o código do Visual Studio e pergunta se você deseja instalar ou habilitar a extensão. Selecione "instalar".

Como executar os exemplos

Para criar e executar os exemplos neste tutorial, use o utilitário dotnet na linha de comando. Siga estas etapas para cada exemplo:

1. Crie um diretório para armazenar o exemplo.

2. Digite o comando dotnet new console em um prompt de comando para criar um novo projeto do .NET Core.

3. Copie e cole o código do exemplo no editor de código.

4. Digite o comando dotnet restore na linha de comando para carregar ou restaurar as dependências do projeto.

   Não é necessário executar dotnet restore, pois ele é executado implicitamente por todos os comandos que exigem uma restauração, como `dotnet new`, `dotnet build`, `dotnet run`, `dotnet test`, `dotnet publish` e `dotnet pack`. Para desabilitar a restauração implícita, use a opção `--no-restore`.

   O comando `dotnet restore` ainda é útil em determinados cenários em que realizar uma restauração explícita faz sentido, como compilações de integração contínua no Azure DevOps Services ou em sistemas de compilação que precisam controlar explicitamente quando a restauração ocorrerá.

   Para obter informações sobre como gerenciar feeds do NuGet, confira a documentação do dotnet restore.

5. Insira o comando executar dotnet para compilar e executar o exemplo.

Plano de fundo: O que é herança?

A herança é um dos atributos fundamentais da programação orientada a objetos. Ele permite que você defina uma classe filha que reutiliza (herda), estende ou modifica o comportamento de uma classe base. A classe cujos membros são herdados é chamada de *classe base*. A classe que herda os membros da classe base é chamada de *classe derivada*.

O C# e o .NET dão suporte apenas à *herança única* . Ou seja, uma classe só pode herdar de uma única classe. No entanto, a herança é transitiva, o que permite definir uma hierarquia de herança para um conjunto de tipos. Em outras palavras, o tipo `D` pode herdar do tipo `C`, que herda do tipo `B`, que herda do tipo de classe base `A`. Como a herança é transitiva, os membros do tipo `A` estão disponíveis para digitar `D`.

Nem todos os membros de uma classe base são herdados por classes derivadas. Os seguintes membros não são herdados:

- Construtores estáticos, que inicializam os dados estáticos de uma classe.

- Construtores de instância, que você chama para criar uma nova instância da classe. Cada classe deve definir seus próprios construtores.
- Finalizadores, que são chamados pelo coletor de lixo do runtime para destruir instâncias de uma classe.

Embora todos os outros membros de uma classe base sejam herdados por classes derivadas, se eles são visíveis ou não dependem de sua acessibilidade. A acessibilidade de um membro afeta sua visibilidade para classes derivadas da seguinte maneira:

- Membros Privados são visíveis apenas em classes derivadas que estão aninhadas em sua classe base. Caso contrário, elas não estarão visíveis em classes derivadas. No exemplo a seguir, `A.B` é uma classe aninhada da `A` qual deriva e `C` deriva de `A`. O campo privado `A._value` é visível no A.B. No entanto, se você remover os comentários do `C.GetValue` método e tentar compilar o exemplo, ele produzirá o erro do compilador CS0122: "'A._value' está inacessível devido ao seu nível de proteção".

```C#
public class A
{
    private int _value = 10;

    public class B : A
    {
        public int GetValue()
        {
            return _value;
        }
    }
}

public class C : A
{
    //    public int GetValue()
    //    {
    //        return _value;
    //    }
}

public class AccessExample
{
    public static void Main(string[] args)
    {
        var b = new A.B();
        Console.WriteLine(b.GetValue());
    }
}
```

```
// The example displays the following output:
//       10
```

- Os membros protegidos são visíveis apenas em classes derivadas.
- Membros internos são visíveis apenas em classes derivadas localizadas no mesmo assembly que a classe base. Elas não são visíveis em classes derivadas localizadas em um assembly diferente da classe base.
- Os membros públicos são visíveis em classes derivadas e fazem parte da interface pública da classe derivada. Membros herdados públicos podem ser chamados exatamente como se fossem definidos na classe derivada. No exemplo a seguir, a classe `A` define um método chamado `Method1` e a classe `B` herda da classe `A`. Em seguida, o exemplo chama `Method1` como se fosse um método de instância em `B`.

C#

```
public class A
{
    public void Method1()
    {
        // Method implementation.
    }
}

public class B : A
{ }

public class Example
{
    public static void Main()
    {
        B b = new ();
        b.Method1();
    }
}
```

Classes derivadas também podem *substituir* membros herdados fornecendo uma implementação alternativa. Para poder substituir um membro, o membro na classe base deve ser marcado com a palavra-chave virtual . Por padrão, os membros da classe base não são marcados como `virtual` e não podem ser substituídos. A tentativa de substituir um membro não virtual, como o seguinte exemplo faz, gera o erro do compilador CS0506: "O <membro> não pode substituir o membro herdado <membro>, pois não está marcado como virtual, abstrato ou de substituição."

C#

```
public class A
{
    public void Method1()
    {
        // Do something.
    }
}

public class B : A
{
    public override void Method1() // Generates CS0506.
    {
        // Do something else.
    }
}
```

Em alguns casos, uma classe derivada *deve* substituir a implementação da classe base. Os membros da classe base marcados com a palavra-chave abstrata exigem que as classes derivadas as substituam. A tentativa de compilar o exemplo a seguir gera o erro do compilador CS0534, "<classe> não implementa o membro abstrato herdado <membro>", porque a classe `B` não fornece implementação para `A.Method1`.

C#

```
public abstract class A
{
    public abstract void Method1();
}

public class B : A // Generates CS0534.
{
    public void Method3()
    {
        // Do something.
    }
}
```

A herança se aplica apenas a classes e interfaces. Outras categorias de tipo (structs, delegados e enumerações) não dão suporte à herança. Devido a essas regras, a tentativa de compilar código como o exemplo a seguir produz um erro do compilador CS0527: "Tipo 'ValueType' na lista de interfaces não é uma interface". A mensagem de erro indica que, embora você possa definir as interfaces que um struct implementa, não há suporte para a herança.

C#

```
public struct ValueStructure : ValueType // Generates CS0527.
{
```

```
}
```

Herança implícita

Além de qualquer tipo que eles possam herdar através de herança única, todos os tipos no sistema de tipos .NET herdam implicitamente de Object ou de um tipo derivado dele. A funcionalidade comum de Object está disponível para qualquer tipo.

Para ver o que significa herança implícita, vamos definir uma nova classe, `SimpleClass` que é simplesmente uma definição de classe vazia:

C#

```
public class SimpleClass
{ }
```

Em seguida, você pode usar reflexão (que permite inspecionar os metadados de um tipo para obter informações sobre esse tipo) para obter uma lista dos membros que pertencem ao `SimpleClass` tipo. Embora você não tenha definido nenhum membro em sua `SimpleClass` classe, a saída do exemplo indica que ele realmente tem nove membros. Um desses membros é um construtor sem parâmetros (ou padrão) que é fornecido automaticamente para o `SimpleClass` tipo pelo compilador C#. Os oito restantes são membros do tipo Object, do qual todas as classes e interfaces do sistema de tipos .NET herdam implicitamente.

C#

```
using System.Reflection;

public class SimpleClassExample
{
    public static void Main()
    {
        Type t = typeof(SimpleClass);
        BindingFlags flags = BindingFlags.Instance | BindingFlags.Static |
BindingFlags.Public |
                             BindingFlags.NonPublic |
BindingFlags.FlattenHierarchy;
        MemberInfo[] members = t.GetMembers(flags);
        Console.WriteLine($"Type {t.Name} has {members.Length} members: ");
        foreach (MemberInfo member in members)
        {
            string access = "";
            string stat = "";
            var method = member as MethodBase;
            if (method != null)
            {
```

```
                if (method.IsPublic)
                    access = " Public";
                else if (method.IsPrivate)
                    access = " Private";
                else if (method.IsFamily)
                    access = " Protected";
                else if (method.IsAssembly)
                    access = " Internal";
                else if (method.IsFamilyOrAssembly)
                    access = " Protected Internal ";
                if (method.IsStatic)
                    stat = " Static";
            }
            string output = $"{member.Name} ({member.MemberType}): {access}{stat},
Declared by {member.DeclaringType}";
            Console.WriteLine(output);
        }
    }
}
// The example displays the following output:
//  Type SimpleClass has 9 members:
//  ToString (Method):  Public, Declared by System.Object
//  Equals (Method):  Public, Declared by System.Object
//  Equals (Method):  Public Static, Declared by System.Object
//  ReferenceEquals (Method):  Public Static, Declared by System.Object
//  GetHashCode (Method):  Public, Declared by System.Object
//  GetType (Method):  Public, Declared by System.Object
//  Finalize (Method):  Internal, Declared by System.Object
//  MemberwiseClone (Method):  Internal, Declared by System.Object
//  .ctor (Constructor):  Public, Declared by SimpleClass
```

A herança implícita da Object classe disponibiliza esses métodos para a `SimpleClass` classe:

- O método público `ToString` , que converte um `SimpleClass` objeto em sua representação de cadeia de caracteres, retorna o nome de tipo totalmente qualificado. Nesse caso, o `ToString` método retorna a cadeia de caracteres "SimpleClass".

- Três métodos que testam a igualdade de dois objetos: o método de instância `Equals(Object)` pública, o método estático `Equals(Object, Object)` público e o método estático `ReferenceEquals(Object, Object)` público. Por padrão, esses métodos testam a igualdade de referência; ou seja, para ser igual, duas variáveis de objeto devem se referir ao mesmo objeto.

- O método público `GetHashCode` , que calcula um valor que permite que uma instância do tipo seja usada em coleções de hash.

- O método público `GetType` , que retorna um Type objeto que representa o `SimpleClass` tipo.

- O método protegido Finalize , que foi projetado para liberar recursos não gerenciados antes que a memória de um objeto seja recuperada pelo coletor de lixo.
- O método protegido MemberwiseClone , que cria um clone superficial do objeto atual.

Por causa da herança implícita, você pode chamar qualquer membro herdado de um `SimpleClass` objeto como se fosse realmente um membro definido na `SimpleClass` classe. Por exemplo, o exemplo a seguir chama o `SimpleClass.ToString` método, que `SimpleClass` herda de Object.

C#

```
public class EmptyClass
{ }

public class ClassNameExample
{
    public static void Main()
    {
        EmptyClass sc = new();
        Console.WriteLine(sc.ToString());
    }
}
// The example displays the following output:
//        EmptyClass
```

A tabela a seguir lista as categorias de tipos que você pode criar em C# e os tipos dos quais eles herdam implicitamente. Cada tipo base disponibiliza um conjunto diferente de membros por meio de herança para tipos derivados implicitamente.

Expandir a tabela

Categoria de tipo	Herda implicitamente de
classe	Object
Struct	ValueType, Object
enumeração	Enum ValueType Object
delegar	MulticastDelegate Delegate Object

Herança e um relacionamento "é um(a)"

Normalmente, a herança é usada para expressar uma relação "é um" entre uma classe base e uma ou mais classes derivadas, em que as classes derivadas são versões especializadas da

classe base; a classe derivada é um tipo da classe base. Por exemplo, a `Publication` classe representa uma publicação de qualquer tipo, e as classes `Book` e `Magazine` representam tipos específicos de publicações.

> ⓘ **Observação**
>
> Uma classe ou struct pode implementar uma ou mais interfaces. Embora a implementação da interface seja apresentada geralmente como uma alternativa para herança única, ou como uma forma de usar a herança com structs, ela tem como objetivo expressar um relacionamento diferente (um relacionamento "pode fazer") entre uma interface e seu tipo de implementação em comparação com a herança. Uma interface define um subconjunto de funcionalidade (como a capacidade de testar a igualdade, comparar ou classificar objetos ou dar suporte à análise e formatação sensíveis à cultura) que a interface disponibiliza para seus tipos de implementação.

Observe que "é um(a)" também expressa o relacionamento entre um tipo e uma instanciação específica desse tipo. No exemplo a seguir, `Automobile` é uma classe que tem três propriedades exclusivas somente leitura: `Make`, o fabricante do automóvel; `Model`, o tipo de automóvel; e `Year`, seu ano de fabricação. Sua `Automobile` classe também tem um construtor cujos argumentos são atribuídos aos valores de propriedade e substitui o Object.ToString método para produzir uma cadeia de caracteres que identifica exclusivamente a `Automobile` instância em vez da `Automobile` classe.

C#

```
public class Automobile
{
    public Automobile(string make, string model, int year)
    {
        if (make == null)
            throw new ArgumentNullException(nameof(make), "The make cannot be
null.");
        else if (string.IsNullOrWhiteSpace(make))
            throw new ArgumentException("make cannot be an empty string or have
space characters only.");
        Make = make;

        if (model == null)
            throw new ArgumentNullException(nameof(model), "The model cannot be
null.");
        else if (string.IsNullOrWhiteSpace(model))
            throw new ArgumentException("model cannot be an empty string or have
space characters only.");
        Model = model;

        if (year < 1857 || year > DateTime.Now.Year + 2)
```

```
            throw new ArgumentException("The year is out of range.");
        Year = year;
    }

    public string Make { get; }

    public string Model { get; }

    public int Year { get; }

    public override string ToString() => $"{Year} {Make} {Model}";
}
```

Nesse caso, você não deve depender da herança para representar marcas e modelos de carros específicos. Por exemplo, você não precisa definir um `Packard` tipo para representar automóveis fabricados pela Packard Motor Car Company. Em vez disso, você pode representá-los criando um `Automobile` objeto com os valores apropriados passados para seu construtor de classe, como o exemplo a seguir faz.

C#

```
using System;

public class Example
{
    public static void Main()
    {
        var packard = new Automobile("Packard", "Custom Eight", 1948);
        Console.WriteLine(packard);
    }
}
// The example displays the following output:
//        1948 Packard Custom Eight
```

Um relacionamento é-um(a) baseado na herança é mais bem aplicado a uma classe base e em classes derivadas que adicionam outros membros à classe base, ou que exigem funcionalidades adicionais não incluídas na classe base.

Projetando a classe base e as classes derivadas

Vamos examinar o processo de criação de uma classe base e suas classes derivadas. Nesta seção, você definirá uma classe base, `Publication` que representa uma publicação de qualquer tipo, como um livro, uma revista, um jornal, um diário, um artigo etc. Você também definirá uma `Book` classe que deriva de `Publication`. Você poderia estender facilmente o exemplo para definir outras classes derivadas, como `Magazine`, , `Journal` `Newspaper` e `Article`.

A classe base de Publicação

Ao projetar sua `Publication` classe, você precisa tomar várias decisões de design:

- Quais membros incluir em sua classe base `Publication` e se os `Publication` membros fornecem implementações de método ou se `Publication` é uma classe base abstrata que serve como um modelo para suas classes derivadas.

  Nesse caso, a `Publication` classe fornecerá implementações de método. A seção Criar classes base abstratas e suas classes derivadas contém um exemplo que usa uma classe base abstrata para definir os métodos que as classes derivadas devem substituir. Classes derivadas são gratuitas para fornecer qualquer implementação adequada para o tipo derivado.

  A capacidade de reutilizar código (ou seja, várias classes derivadas compartilham a declaração e a implementação de métodos de classe base e não precisam substituí-los) é uma vantagem das classes base não abstratas. Portanto, você deverá adicionar membros à `Publication` se o código precisar ser compartilhado por um ou mais tipos `Publication` especializados. Se você não fornecer implementações de classe base com eficiência, você acabará tendo que fornecer implementações de membro praticamente idênticas em classes derivadas, em vez de uma única implementação na classe base. A necessidade de manter o código duplicado em vários locais é uma possível fonte de bugs.

  Para maximizar a reutilização de código e criar uma hierarquia de herança lógica e intuitiva, você deseja ter certeza de que inclui na `Publication` classe apenas os dados e a funcionalidade que são comuns a todas ou à maioria das publicações. Em seguida, as classes derivadas implementam membros exclusivos para os tipos específicos de publicação que eles representam.

- Até onde estender sua hierarquia de classes. Deseja desenvolver uma hierarquia de três ou mais classes, em vez de simplesmente uma classe base e uma ou mais classes derivadas? Por exemplo, `Publication` pode ser uma classe base de `Periodical`, que, por sua vez, é uma classe base de `Magazine`, `Journal` e `Newspaper`.

  Para seu exemplo, você usará a hierarquia pequena de uma `Publication` classe e uma única classe derivada. `Book` Você poderia facilmente estender o exemplo para criar várias classes adicionais que derivam de `Publication`, como `Magazine` e `Article`.

- Se faz sentido instanciar a classe base. Se isso não acontecer, você deverá aplicar a palavra-chave abstrata à classe. Caso contrário, sua `Publication` classe poderá ser instanciada chamando seu construtor de classe. Se for feita uma tentativa de instanciar uma classe marcada com a `abstract` palavra-chave por uma chamada direta ao

construtor de classe, o compilador C# gerará o erro CS0144, "Não é possível criar uma instância da classe ou interface abstrata". Se for feita uma tentativa de instanciar a classe usando reflexão, o método de reflexão gerará um MemberAccessException.

Por padrão, uma classe base pode ser instanciada chamando seu construtor de classe. Você não precisa definir explicitamente um construtor de classe. Se um não estiver presente no código-fonte da classe base, o compilador C# fornecerá automaticamente um construtor padrão (sem parâmetros).

Para seu exemplo, você marcará a `Publication` classe como abstrata para que ela não possa ser instanciada. Uma `abstract` classe sem métodos `abstract` indica que essa classe representa um conceito abstrato que é compartilhado entre várias classes concretas (como uma `Book`, `Journal`).

- Se as classes derivadas devem herdar a implementação de classe base de membros específicos, se elas têm a opção de substituir a implementação da classe base ou se devem fornecer uma implementação. Use a palavra-chave abstrata para forçar classes derivadas a fornecer uma implementação. Use a palavra-chave virtual para permitir que classes derivadas substituam um método de classe base. Por padrão, os métodos definidos na classe base *não* são substituíveis.

  A `Publication` classe não tem métodos `abstract` , mas a classe em si é `abstract`.

- Se uma classe derivada representa a classe final na hierarquia de herança e não pode ser usada como uma classe base para classes derivadas adicionais. Por padrão, qualquer classe pode servir como uma classe base. Você pode aplicar a palavra-chave selada para indicar que uma classe não pode servir como uma classe base para classes adicionais. Tentativa de derivar de uma classe lacrada gerou o erro do compilador CS0509: "não é possível derivar do tipo lacrado <typeName>."

  Para seu exemplo, você marcará sua classe derivada como `sealed`.

O exemplo a seguir mostra o código-fonte da `Publication` classe, bem como uma `PublicationType` enumeração retornada pela `Publication.PublicationType` propriedade. Além dos membros herdados de Object, a classe `Publication` define os seguintes membros exclusivos e substituições de membro:

C#

```
public enum PublicationType { Misc, Book, Magazine, Article };

public abstract class Publication
{
    private bool _published = false;
```

```
    private DateTime _datePublished;
    private int _totalPages;

    public Publication(string title, string publisher, PublicationType type)
    {
        if (string.IsNullOrWhiteSpace(publisher))
            throw new ArgumentException("The publisher is required.");
        Publisher = publisher;

        if (string.IsNullOrWhiteSpace(title))
            throw new ArgumentException("The title is required.");
        Title = title;

        Type = type;
    }

    public string Publisher { get; }

    public string Title { get; }

    public PublicationType Type { get; }

    public string? CopyrightName { get; private set; }

    public int CopyrightDate { get; private set; }

    public int Pages
    {
        get { return _totalPages; }
        set
        {
            if (value <= 0)
                throw new ArgumentOutOfRangeException(nameof(value), "The number
of pages cannot be zero or negative.");
            _totalPages = value;
        }
    }

    public string GetPublicationDate()
    {
        if (!_published)
            return "NYP";
        else
            return _datePublished.ToString("d");
    }

    public void Publish(DateTime datePublished)
    {
        _published = true;
        _datePublished = datePublished;
    }

    public void Copyright(string copyrightName, int copyrightDate)
    {
        if (string.IsNullOrWhiteSpace(copyrightName))
```

```
            throw new ArgumentException("The name of the copyright holder is
required.");
        CopyrightName = copyrightName;

        int currentYear = DateTime.Now.Year;
        if (copyrightDate < currentYear - 10 || copyrightDate > currentYear + 2)
            throw new ArgumentOutOfRangeException($"The copyright year must be
between {currentYear - 10} and {currentYear + 1}");
        CopyrightDate = copyrightDate;
    }

    public override string ToString() => Title;
}
```

- Um construtor

  Como a `Publication` classe é `abstract`, ela não pode ser instanciada diretamente do código, como o exemplo a seguir:

  ```C#
  var publication = new Publication("Tiddlywinks for Experts", "Fun and Games",
                                    PublicationType.Book);
  ```

  No entanto, seu construtor de instância pode ser chamado diretamente de construtores de classes derivadas, como mostra o código-fonte da classe `Book`.

- Duas propriedades relacionadas à publicação

  `Title` é uma propriedade somente de leitura String cujo valor é fornecido pela chamada do construtor `Publication`.

  `Pages` é uma propriedade de leitura-gravação Int32 que indica quantas páginas totais a publicação tem. O valor é armazenado em um campo privado chamado `totalPages`. O lançamento deve ser de um número positivo ou de um ArgumentOutOfRangeException.

- Membros relacionados ao publicador

  Duas propriedades somente leitura, `Publisher` e `Type`. Os valores são fornecidos originalmente pela chamada ao construtor da classe `Publication`.

- Membros relacionados à publicação

  Dois métodos `Publish` e `GetPublicationDate`, definir e retornar a data de publicação. O `Publish` método define um sinalizador privado `published` para `true` quando ele é chamado e atribui a data passada a ele como um argumento para o campo privado

`datePublished` . O método `GetPublicationDate` retorna a cadeia de caracteres "NYP" se o sinalizador `published` for `false`, e o valor do campo `datePublished` for `true`.

- Membros relacionados a direitos autorais

  O método `Copyright` usa o nome do titular dos direitos autorais e o ano dos direitos como argumentos e os atribui às propriedades `CopyrightName` e `CopyrightDate`.

- Uma substituição do método `ToString`

  Se um tipo não sobrescrever o método Object.ToString, ele retornará o nome totalmente qualificado do tipo, o que é pouco útil para diferenciar uma instância de outra. A `Publication` classe substitui Object.ToString para retornar o valor da `Title` propriedade.

A figura a seguir ilustra a relação entre sua classe base `Publication` e sua classe implicitamente herdada Object .

A classe `Book`

A `Book` classe representa um livro como um tipo especializado de publicação. O exemplo a seguir mostra o código-fonte da `Book` classe.

C#

```
using System;

public sealed class Book : Publication
{
    public Book(string title, string author, string publisher) :
           this(title, string.Empty, author, publisher)
    { }

    public Book(string title, string isbn, string author, string publisher) :
base(title, publisher, PublicationType.Book)
    {
        // isbn argument must be a 10- or 13-character numeric string without "-"
characters.
        // We could also determine whether the ISBN is valid by comparing its
checksum digit
        // with a computed checksum.
        //
        if (!string.IsNullOrEmpty(isbn))
        {
            // Determine if ISBN length is correct.
            if (!(isbn.Length == 10 | isbn.Length == 13))
                throw new ArgumentException("The ISBN must be a 10- or 13-
character numeric string.");
            if (!ulong.TryParse(isbn, out _))
                throw new ArgumentException("The ISBN can consist of numeric
characters only.");
        }
        ISBN = isbn;

        Author = author;
    }

    public string ISBN { get; }

    public string Author { get; }

    public decimal Price { get; private set; }

    // A three-digit ISO currency symbol.
    public string? Currency { get; private set; }

    // Returns the old price, and sets a new price.
    public decimal SetPrice(decimal price, string currency)
    {
        if (price < 0)
            throw new ArgumentOutOfRangeException(nameof(price), "The price cannot
be negative.");
        decimal oldValue = Price;
```

```
        Price = price;

        if (currency.Length != 3)
            throw new ArgumentException("The ISO currency symbol is a 3-character
string.");
        Currency = currency;

        return oldValue;
    }

    public override bool Equals(object? obj)
    {
        if (obj is not Book book)
            return false;
        else
            return ISBN == book.ISBN;
    }

    public override int GetHashCode() => ISBN.GetHashCode();

    public override string ToString() => $"{(string.IsNullOrEmpty(Author) ? "" :
Author + ", ")}{Title}";
}
```

Além dos membros herdados de `Publication`, a classe `Book` define os seguintes membros exclusivos e substituições de membro:

- Dois construtores

  Os dois `Book` construtores compartilham três parâmetros comuns. Dois, *título* e *editor*, correspondem aos parâmetros do `Publication` construtor. O terceiro é *author*, que é armazenado em uma propriedade pública `Author` imutável. Um construtor inclui um parâmetro *isbn*, que é armazenado na propriedade automática `ISBN`.

  O primeiro construtor usa essa palavra-chave para chamar o outro construtor. O encadeamento do construtor é um padrão comum na definição de construtores. Construtores com menos parâmetros fornecem valores padrão ao chamar o construtor com o maior número de parâmetros.

  O segundo construtor usa a palavra-chave base para passar o título e o nome do editor para o construtor de classe base. Se você não fizer uma chamada explícita para um construtor de classe base no código-fonte, o compilador C# fornecerá automaticamente uma chamada para o construtor padrão ou sem parâmetros da classe base.

- Uma propriedade de leitura somente `ISBN`, que retorna o ISBN do objeto `Book`, um número exclusivo de 10 ou 13 dígitos. O ISBN é fornecido como um argumento para um

dos `Book` construtores. O ISBN é armazenado em um campo de backup privado, que é gerado automaticamente pelo compilador.

- Uma propriedade `Author` somente leitura. O nome do autor é fornecido como um argumento para os dois construtores `Book` e é armazenado na propriedade.

- Duas propriedades somente leitura relacionadas ao preço, `Price` e `Currency`. Seus valores são fornecidos como argumentos em uma `SetPrice` chamada de método. A `Currency` propriedade é o símbolo de moeda ISO de três dígitos (por exemplo, USD para o dólar americano). Os símbolos de moeda ISO podem ser recuperados da propriedade ISOCurrencySymbol. Ambas as propriedades são somente leitura externa, mas ambas podem ser definidas por código na `Book` classe.

- Um `SetPrice` método, que define os valores das propriedades `Price` e `Currency`. Esses valores são retornados por essas mesmas propriedades.

- Substitui o método `ToString` (herdado de `Publication`) e os métodos Object.Equals(Object) e GetHashCode (herdados de Object).

  A menos que seja substituído, o Object.Equals(Object) método testa a igualdade de referência. Ou seja, duas variáveis de objeto são consideradas iguais se se referirem ao mesmo objeto. Na classe `Book`, por outro lado, dois objetos `Book` deverão ser considerados iguais se possuírem o mesmo ISBN.

  Ao substituir o Object.Equals(Object) método, você também deve substituir o GetHashCode método, que retorna um valor que o runtime usa para armazenar itens em coleções de hash para recuperação eficiente. O código hash deve retornar um valor consistente com o teste de igualdade. Como você substituiu Object.Equals(Object) para retornar `true`, se as propriedades de ISBN de dois objetos `Book` forem iguais, retorne o código hash computado chamando o método GetHashCode da cadeia de caracteres retornada pela propriedade `ISBN`.

A figura a seguir ilustra a relação entre a `Book` classe e `Publication` sua classe base.

Agora você pode instanciar um `Book` objeto, invocar seus membros exclusivos e herdados e passá-lo como um argumento para um método que espera um parâmetro de tipo `Publication` ou de tipo `Book`, como mostra o exemplo a seguir.

C#

```
public class ClassExample
{
    public static void Main()
```

```
    {
        var book = new Book("The Tempest", "0971655819", "Shakespeare, William",
                            "Public Domain Press");
        ShowPublicationInfo(book);
        book.Publish(new DateTime(2016, 8, 18));
        ShowPublicationInfo(book);

        var book2 = new Book("The Tempest", "Classic Works Press", "Shakespeare,
William");
        Console.Write($"{book.Title} and {book2.Title} are the same publication: "
+
              $"{((Publication)book).Equals(book2)}");
    }

    public static void ShowPublicationInfo(Publication pub)
    {
        string pubDate = pub.GetPublicationDate();
        Console.WriteLine($"{pub.Title}, " +
                  $"{(pubDate == "NYP" ? "Not Yet Published" : "published on " +
pubDate):d} by {pub.Publisher}");
    }
}
// The example displays the following output:
//        The Tempest, Not Yet Published by Public Domain Press
//        The Tempest, published on 8/18/2016 by Public Domain Press
//        The Tempest and The Tempest are the same publication: False
```

Criando classes base abstratas e suas classes derivadas

No exemplo anterior, você definiu uma classe base que forneceu uma implementação para vários métodos para permitir que classes derivadas compartilhassem código. Em muitos casos, no entanto, não se espera que a classe base forneça uma implementação. Em vez disso, a classe base é uma *classe abstrata* que declara *métodos abstratos*; ele serve como um modelo que define os membros que cada classe derivada deve implementar. Normalmente, em uma classe base abstrata, a implementação de cada tipo derivado é exclusiva para esse tipo. Você marcou a classe com a palavra-chave abstrata porque não fazia sentido instanciar um `Publication` objeto, embora a classe tenha fornecido implementações de funcionalidade comuns às publicações.

Por exemplo, cada forma geométrica bidimensional fechada inclui duas propriedades: área, a extensão interna da forma; e perímetro, ou a distância ao longo das bordas da forma. A maneira como essas propriedades são calculadas, no entanto, depende completamente da forma específica. A fórmula para calcular o perímetro (ou circunferência) de um círculo, por exemplo, é diferente da de um quadrado. A `Shape` classe é uma `abstract` classe com `abstract`

métodos. Isso indica que as classes derivadas compartilham a mesma funcionalidade, mas essas classes derivadas implementam essa funcionalidade de forma diferente.

O exemplo a seguir define uma classe base abstrata nomeada `Shape` que define duas propriedades: `Area` e `Perimeter`. Além de marcar a classe com a palavra-chave abstrata , cada membro da instância também é marcado com a palavra-chave abstrata . Nesse caso, `Shape` também substitui o Object.ToString método para retornar o nome do tipo, em vez de seu nome totalmente qualificado. E define dois membros estáticos, `GetArea` e `GetPerimeter`, que permitem que os chamadores recuperem facilmente a área e o perímetro de uma instância de qualquer classe derivada. Quando você passa uma instância de uma classe derivada para um desses métodos, o runtime chama a substituição do método da classe derivada.

C#

```
public abstract class Shape
{
    public abstract double Area { get; }

    public abstract double Perimeter { get; }

    public override string ToString() => GetType().Name;

    public static double GetArea(Shape shape) => shape.Area;

    public static double GetPerimeter(Shape shape) => shape.Perimeter;
}
```

Você pode então derivar algumas classes de `Shape` que representam formas específicas. O exemplo a seguir define três classes, `Square` `Rectangle` e `Circle`. Cada um usa uma fórmula exclusiva para essa forma específica para calcular a área e o perímetro. Algumas das classes derivadas também definem propriedades, como `Rectangle.Diagonal` e `Circle.Diameter`, que são exclusivas da forma que representam.

C#

```
using System;

public class Square : Shape
{
    public Square(double length)
    {
        Side = length;
    }

    public double Side { get; }

    public override double Area => Math.Pow(Side, 2);
```

```
    public override double Perimeter => Side * 4;

    public double Diagonal => Math.Round(Math.Sqrt(2) * Side, 2);
}

public class Rectangle : Shape
{
    public Rectangle(double length, double width)
    {
        Length = length;
        Width = width;
    }

    public double Length { get; }

    public double Width { get; }

    public override double Area => Length * Width;

    public override double Perimeter => 2 * Length + 2 * Width;

    public bool IsSquare() => Length == Width;

    public double Diagonal => Math.Round(Math.Sqrt(Math.Pow(Length, 2) +
Math.Pow(Width, 2)), 2);
}

public class Circle : Shape
{
    public Circle(double radius)
    {
        Radius = radius;
    }

    public override double Area => Math.Round(Math.PI * Math.Pow(Radius, 2), 2);

    public override double Perimeter => Math.Round(Math.PI * 2 * Radius, 2);

    // Define a circumference, since it's the more familiar term.
    public double Circumference => Perimeter;

    public double Radius { get; }

    public double Diameter => Radius * 2;
}
```

O exemplo a seguir usa objetos derivados de `Shape`. Ele cria uma matriz de objetos derivados de `Shape` e chama os métodos estáticos da classe `Shape`, que retorna valores de propriedade `Shape`. O runtime recupera os valores das propriedades substituídas dos tipos derivados. O exemplo também converte cada `Shape` objeto na matriz em seu tipo derivado e, se a conversão for bem-sucedida, recuperará as propriedades dessa subclasse específica de `Shape`.

C#

```
using System;

public class Example
{
    public static void Main()
    {
        Shape[] shapes = { new Rectangle(10, 12), new Square(5),
                    new Circle(3) };
        foreach (Shape shape in shapes)
        {
            Console.WriteLine($"{shape}: area, {Shape.GetArea(shape)}; " +
                              $"perimeter, {Shape.GetPerimeter(shape)}");
            if (shape is Rectangle rect)
            {
                Console.WriteLine($"   Is Square: {rect.IsSquare()}, Diagonal:
{rect.Diagonal}");
                continue;
            }
            if (shape is Square sq)
            {
                Console.WriteLine($"   Diagonal: {sq.Diagonal}");
                continue;
            }
        }
    }
}
// The example displays the following output:
//         Rectangle: area, 120; perimeter, 44
//            Is Square: False, Diagonal: 15.62
//         Square: area, 25; perimeter, 20
//            Diagonal: 7.07
//         Circle: area, 28.27; perimeter, 18.85
```

Como converter com segurança usando a correspondência de padrões e os operadores is e as

19/06/2025

Como os objetos são polimórficos, é possível que uma variável de um tipo de classe base mantenha um tipo derivado. Para acessar os membros da instância do tipo derivado, é necessário converter o valor de volta para o tipo derivado. No entanto, uma conversão cria o risco de lançar um InvalidCastException. O C# fornece declarações de correspondência de padrões que executam uma conversão condicionalmente somente quando ela for bem-sucedida. O C# também fornece os operadores is e as para testar se um valor é de um determinado tipo.

O exemplo a seguir mostra como usar a instrução `is` de correspondência de padrões:

C#

```
var g = new Giraffe();
var a = new Animal();
FeedMammals(g);
FeedMammals(a);
// Output:
// Eating.
// Animal is not a Mammal

SuperNova sn = new SuperNova();
TestForMammals(g);
TestForMammals(sn);
// 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");
    }
}

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 { }
```

O exemplo anterior demonstra alguns recursos de sintaxe de correspondência de padrões. A `if (a is Mammal m)` instrução combina o teste com uma atribuição de inicialização. A atribuição ocorre somente quando o teste é bem-sucedido. A variável `m` está somente no escopo na instrução `if` inserida em ela foi atribuída. Você não pode acessar `m` mais tarde no mesmo método. O exemplo anterior também mostra como usar o as operador para converter um objeto em um tipo especificado.

Você também pode usar a mesma sintaxe para testar se um tipo de valor anulável tiver um valor, conforme mostrado no exemplo a seguir:

C#

```
int i = 5;
PatternMatchingNullable(i);

int? j = null;
PatternMatchingNullable(j);

double d = 9.78654;
PatternMatchingNullable(d);

PatternMatchingSwitch(i);
PatternMatchingSwitch(j);
PatternMatchingSwitch(d);
```

```
static void PatternMatchingNullable(ValueType? val)
{
    if (val is int j) // Nullable types are not allowed in patterns
    {
        Console.WriteLine(j);
    }
    else if (val is null) // If val is a nullable type with no value, this
expression is true
    {
        Console.WriteLine("val is a nullable type with the null value");
    }
    else
    {
        Console.WriteLine("Could not convert " + val.ToString());
    }
}

static void PatternMatchingSwitch(ValueType? val)
{
    switch (val)
    {
        case int number:
            Console.WriteLine(number);
            break;
        case long number:
            Console.WriteLine(number);
            break;
        case decimal number:
            Console.WriteLine(number);
            break;
        case float number:
            Console.WriteLine(number);
            break;
        case double number:
            Console.WriteLine(number);
            break;
        case null:
            Console.WriteLine("val is a nullable type with the null value");
            break;
        default:
            Console.WriteLine("Could not convert " + val.ToString());
            break;
    }
}
```

O exemplo anterior demonstra outros recursos de correspondência de padrões a serem usados com conversões. Você pode testar uma variável para o padrão nulo verificando especificamente o `null` valor. Quando o valor do runtime da variável é `null`, uma instrução `is` que verifica um tipo sempre retorna `false`. A instrução de correspondência de padrões `is` não permite um tipo de valor anulável, como `int?` ou `Nullable<int>`, mas você pode testar para qualquer outro tipo de valor. Os `is` padrões do exemplo anterior não são limitados aos tipos

de valor anuláveis. Você também pode usar esses padrões para testar se uma variável de um tipo de referência tem um valor ou se é `null`.

O exemplo anterior também mostra como você usa o padrão de tipo em uma `switch` instrução em que a variável pode ser um dos muitos tipos diferentes.

Se você quiser testar se uma variável é de um determinado tipo, mas não atribuí-la a uma nova variável, você pode usar os operadores `is` e `as` para tipos de referência e tipos de valor anuláveis. O código a seguir mostra como usar as instruções `is` e `as` que faziam parte da linguagem C# antes da correspondência de padrões ser introduzida para testar se uma variável é de um determinado tipo:

C#

```
// Use the is operator to verify the type.
// before performing a cast.
Giraffe g = new();
UseIsOperator(g);

// Use the as operator and test for null
// before referencing the variable.
UseAsOperator(g);

// Use pattern matching to test for null
// before referencing the variable
UsePatternMatchingIs(g);

// Use the as operator to test
// an incompatible type.
SuperNova sn = new();
UseAsOperator(sn);

// Use the as operator with a value type.
// Note the implicit conversion to int? in
// the method body.
int i = 5;
UseAsWithNullable(i);

double d = 9.78654;
UseAsWithNullable(d);

static void UseIsOperator(Animal a)
{
    if (a is Mammal)
    {
        Mammal m = (Mammal)a;
        m.Eat();
    }
}

static void UsePatternMatchingIs(Animal a)
```

```
{
    if (a is Mammal m)
    {
        m.Eat();
    }
}

static void UseAsOperator(object o)
{
    Mammal? m = o as Mammal;
    if (m is not null)
    {
        Console.WriteLine(m.ToString());
    }
    else
    {
        Console.WriteLine($"{o.GetType().Name} is not a Mammal");
    }
}

static void UseAsWithNullable(System.ValueType val)
{
    int? j = val as int?;
    if (j is not null)
    {
        Console.WriteLine(j);
    }
    else
    {
        Console.WriteLine("Could not convert " + val.ToString());
    }
}
class Animal
{
    public void Eat() => Console.WriteLine("Eating.");
    public override string ToString() => "I am an animal.";
}
class Mammal : Animal { }
class Giraffe : Mammal { }

class SuperNova { }
```

Como você pode ver comparando esse código com o código de correspondência de padrões, a sintaxe de correspondência de padrões fornece recursos mais robustos combinando o teste e a atribuição em uma única instrução. Use a sintaxe de correspondência de padrões sempre que possível.

Tutorial: Usar padrões correspondentes para criar algoritmos controlados por tipo e controlados por dados

17/06/2025

É possível escrever uma funcionalidade que se comporte como se você tivesse estendido tipos que poderiam estar em outras bibliotecas. Outro uso dos padrões é criar a funcionalidade de que seu aplicativo precisa, mas que não é um recurso fundamental do tipo que está sendo estendido.

Neste tutorial, você aprenderá como:

- ✓ Reconhecer situações em que a correspondência de padrões deverá ser usada.
- ✓ Usar expressões de correspondência de padrões para implementar o comportamento com base em tipos e valores de propriedade.
- ✓ Combinar a correspondência de padrões com outras técnicas para criar algoritmos completos.

Pré-requisitos

- O .NET SDK mais recente
- Editor do Visual Studio Code
- O DevKit C#

Instruções de instalação

No Windows, esse Arquivo de configuração do WinGet para instalar todos os pré-requisitos. Se você já tiver algo instalado, o WinGet ignorará essa etapa.

1. Baixe o arquivo e clique duas vezes para executá-lo.
2. Leia o contrato de licença, digite `y` e selecione `Enter` quando solicitado a aceitar.
3. Se você receber um prompt de UAC (Controle de Conta de Usuário) piscando na barra de tarefas, permita que a instalação continue.

Em outras plataformas, você precisa instalar cada um desses componentes separadamente.

1. Baixe o instalador recomendado na página de download do SDK do .NET e clique duas vezes para executá-lo. A página de download detecta sua plataforma e recomenda o instalador mais recente para sua plataforma.

2. Baixe o instalador mais recente do Visual Studio Code home page e clique duas vezes para executá-lo. Essa página também detecta sua plataforma e o link deve estar correto para o sistema.
3. Clique no botão "Instalar" na página de extensão do DevKit em C#. Isso abre o código do Visual Studio e pergunta se você deseja instalar ou habilitar a extensão. Selecione "instalar".

Este tutorial pressupõe que você esteja familiarizado com o C# e .NET, incluindo o Visual Studio ou a CLI do .NET.

Cenários para a correspondência de padrões

O desenvolvimento moderno geralmente inclui a integração de dados de várias fontes e a apresentação de informações e insights de dados em um único aplicativo coeso. Você e sua equipe não terão controle ou acesso a todos os tipos que representam os dados de entrada.

O design orientado a objeto clássico exigiria a criação de tipos em seu aplicativo que representassem cada tipo de dados das várias fontes de dados. O aplicativo, então, trabalharia com esses novos tipos, criaria hierarquias de herança, métodos virtuais e implementaria abstrações. Essas técnicas funcionam e, às vezes, são as melhores ferramentas. Outras vezes, é possível escrever menos código. Você pode escrever um código mais claro usando técnicas que separam os dados das operações que manipulam esses dados.

Neste tutorial, você vai criar e explorar um aplicativo que usa dados recebidos de várias fontes externas para um único cenário. Você verá como a **correspondência de padrões** fornece uma maneira eficiente para consumir e processar esses dados de maneiras que não eram parte do sistema original.

Imagine, por exemplo, uma área metropolitana principal que está implantando pedágios e preços diferenciados em horário de pico para gerenciar o tráfego. Você escreve um aplicativo que calcula o pedágio de um veículo com base em seu tipo. Posteriormente, as melhorias vão incorporar preços com base no número de ocupantes do veículo. Outros aprimoramentos vão adicionar o preço com base na hora e no dia da semana.

Com base nessa breve descrição, você pode ter elaborado rapidamente uma hierarquia de objetos para modelar esse sistema. No entanto, seus dados são provenientes de várias fontes, como outros sistemas de gerenciamento de registro do veículo. Esses sistemas fornecem classes diferentes para modelar aqueles dados, e você não tem um modelo de objeto único o qual seja possível usar. Neste tutorial, você usará essas classes simplificadas para criar o modelo para os dados do veículo, a partir desses sistemas externos, conforme mostrado no código a seguir:

```C#
namespace ConsumerVehicleRegistration
{
    public class Car
    {
        public int Passengers { get; set; }
    }
}

namespace CommercialRegistration
{
    public class DeliveryTruck
    {
        public int GrossWeightClass { get; set; }
    }
}

namespace LiveryRegistration
{
    public class Taxi
    {
        public int Fares { get; set; }
    }

    public class Bus
    {
        public int Capacity { get; set; }
        public int Riders { get; set; }
    }
}
```

Faça o download do código inicial no repositório dotnet/samples do GitHub. É possível ver que as classes de veículos são de sistemas diferentes, e estão em namespaces diferentes. Nenhuma base comum de classe, além da `System.Object` pode ser aproveitada.

Designs de correspondência de padrões

O cenário usado neste tutorial destaca os tipos de problemas que a correspondência de padrões pode resolver de forma adequada:

- Os objetos com os quais você precisa trabalhar não estão em uma hierarquia de objetos que corresponde aos seus objetivos. É possível que você esteja trabalhando com classes que fazem parte dos sistemas não relacionados.
- A funcionalidade que você está adicionando não faz parte da abstração central dessas classes. A tarifa paga por um veículo *muda* de acordo com diferentes tipos de veículos, mas o pedágio não é uma função principal do veículo.

Quando a *forma* dos dados e as *operações* nos dados não são descritas em conjunto, o recurso de correspondência padrões no C# facilita o trabalho.

Implementar os cálculos básicos de pedágio

O cálculo mais básico do pedágio dependerá apenas do tipo do veículo:

- Um `Car` será R$2,00.
- Um `Taxi` será R$3,50.
- Um `Bus` será R$5,00.
- Um `DeliveryTruck` será R$10,00.

Crie uma nova classe `TollCalculator` e implemente a correspondência de padrões no tipo de veículo para obter a quantidade do pedágio. O código a seguir mostra a implementação inicial do `TollCalculator`.

C#

```
using System;
using CommercialRegistration;
using ConsumerVehicleRegistration;
using LiveryRegistration;

namespace Calculators;

public class TollCalculator
{
    public decimal CalculateToll(object vehicle) =>
        vehicle switch
    {
        Car c           => 2.00m,
        Taxi t          => 3.50m,
        Bus b           => 5.00m,
        DeliveryTruck t => 10.00m,
        { }             => throw new ArgumentException(message: "Not a known
vehicle type", paramName: nameof(vehicle)),
        null            => throw new ArgumentNullException(nameof(vehicle))
    };
}
```

O código anterior usa uma expressão switch (não igual a uma instrução switch) que testa o padrão de instrução. A **expressão switch** inicia-se com a variável, `vehicle` no código anterior, seguida pela palavra-chave `switch`. Em seguida, estão os **braços switch** dentro de chaves. A expressão `switch` faz outros refinamentos na sintaxe que circunda a instrução `switch`. A palavra-chave `case` é omitida, e o resultado de cada braço é uma expressão. Os dois últimos braços apresentam um novo recurso de linguagem. O caso `{ }` corresponde a qualquer objeto

não nulo que não correspondia a um braço anterior. Este braço captura qualquer tipo incorreto passado para esse método. O caso `{ }` precisa seguir os casos para cada tipo de veículo. Se a ordem for revertida, o caso `{ }` terá precedência. Finalmente, o `null` padrão constante detecta quando `null` é passado para esse método. O padrão `null` pode ser o último, porque os outros padrões correspondem apenas a um objeto não nulo do tipo correto.

Você pode testar esse código usando o seguinte código no `Program.cs`:

C#

```
using System;
using CommercialRegistration;
using ConsumerVehicleRegistration;
using LiveryRegistration;

using toll_calculator;

var tollCalc = new TollCalculator();

var car = new Car();
var taxi = new Taxi();
var bus = new Bus();
var truck = new DeliveryTruck();

Console.WriteLine($"The toll for a car is {tollCalc.CalculateToll(car)}");
Console.WriteLine($"The toll for a taxi is {tollCalc.CalculateToll(taxi)}");
Console.WriteLine($"The toll for a bus is {tollCalc.CalculateToll(bus)}");
Console.WriteLine($"The toll for a truck is {tollCalc.CalculateToll(truck)}");

try
{
    tollCalc.CalculateToll("this will fail");
}
catch (ArgumentException e)
{
    Console.WriteLine("Caught an argument exception when using the wrong type");
}
try
{
    tollCalc.CalculateToll(null!);
}
catch (ArgumentNullException e)
{
    Console.WriteLine("Caught an argument exception when using null");
}
```

Esse código está incluído no projeto inicial, mas é comentado. Remova os comentários e você pode testar o que escreveu.

Você está começando a ver como os padrões podem ajudar a criar algoritmos em que o código e os dados estão separados. A expressão `switch` testa o tipo e produz valores diferentes com base nos resultados. Mas isso é somente o começo.

Adicionar preços de acordo com a ocupação do veículo

A autoridade de pedágio deseja incentivar que os veículos viagem com a capacidade máxima de pessoas. Eles decidiram cobrar valores mais altos quando os veículos tiverem poucos passageiros e oferecer redução da tarifa para veículos com a capacidade total ocupada:

- Os carros e táxis com nenhum passageiro pagam uma taxa adicional de R$ 0,50.
- Os carros e táxis com dois passageiros obtêm um desconto de R$ 0,50.
- Os carros e táxis com três ou mais passageiros obtêm um desconto de R$ 1,00.
- Os ônibus com menos de 50% da capacidade completa pagam uma taxa adicional de R$ 2,00.
- Os ônibus com 90% da capacidade de passageiros completa, ganham um desconto de R$ 1,00.

Essas regras podem ser implementadas usando o padrão de propriedade na mesma expressão switch. Um padrão de propriedade compara um valor de propriedade com um valor constante. O padrão de propriedade examina as propriedades do objeto depois que o tipo foi determinado. O único caso de um `Car` se expande para quatro casos diferentes:

C#

```
vehicle switch
{
    Car {Passengers: 0} => 2.00m + 0.50m,
    Car {Passengers: 1} => 2.0m,
    Car {Passengers: 2} => 2.0m - 0.50m,
    Car                 => 2.00m - 1.0m,

    // ...
};
```

Os três primeiros casos testam o tipo como um `Car`, em seguida, verificam o valor da propriedade `Passengers`. Se ambos corresponderem, essa expressão é avaliada e retornada.

Você também expande os casos para táxis de maneira semelhante:

C#

```
vehicle switch
{
    // ...

    Taxi {Fares: 0}  => 3.50m + 1.00m,
    Taxi {Fares: 1}  => 3.50m,
    Taxi {Fares: 2}  => 3.50m - 0.50m,
    Taxi             => 3.50m - 1.00m,

    // ...
};
```

Em seguida, implemente as regras de ocupação expandindo os casos para os ônibus, conforme mostrado no exemplo a seguir:

C#

```
vehicle switch
{
    // ...

    Bus b when ((double)b.Riders / (double)b.Capacity) < 0.50 => 5.00m + 2.00m,
    Bus b when ((double)b.Riders / (double)b.Capacity) > 0.90 => 5.00m - 1.00m,
    Bus => 5.00m,

    // ...
};
```

A autoridade de pedágio não está preocupada com o número de passageiros nos caminhões de carga. Em vez disso, ela ajusta a quantidade de pedágios com base na classe de peso dos caminhões da seguinte maneira:

- Os caminhões mais de 5000 quilos pagam uma taxa adicional de R$ 5,00.
- Os caminhões leves abaixo de 3.000 lb recebem um desconto de US$ 2,00.

Essa regra é implementada com o código a seguir:

C#

```
vehicle switch
{
    // ...

    DeliveryTruck t when (t.GrossWeightClass > 5000) => 10.00m + 5.00m,
    DeliveryTruck t when (t.GrossWeightClass < 3000) => 10.00m - 2.00m,
    DeliveryTruck => 10.00m,
};
```

O código anterior mostra a cláusula `when` de um braço switch. Você usa a cláusula `when` para testar as condições, com exceção da igualdade, em uma propriedade. Ao terminar, o método será muito semelhante ao seguinte código:

C#

```
vehicle switch
{
    Car {Passengers: 0}        => 2.00m + 0.50m,
    Car {Passengers: 1}        => 2.0m,
    Car {Passengers: 2}        => 2.0m - 0.50m,
    Car                        => 2.00m - 1.0m,

    Taxi {Fares: 0}  => 3.50m + 1.00m,
    Taxi {Fares: 1}  => 3.50m,
    Taxi {Fares: 2}  => 3.50m - 0.50m,
    Taxi             => 3.50m - 1.00m,

    Bus b when ((double)b.Riders / (double)b.Capacity) < 0.50 => 5.00m + 2.00m,
    Bus b when ((double)b.Riders / (double)b.Capacity) > 0.90 => 5.00m - 1.00m,
    Bus => 5.00m,

    DeliveryTruck t when (t.GrossWeightClass > 5000) => 10.00m + 5.00m,
    DeliveryTruck t when (t.GrossWeightClass < 3000) => 10.00m - 2.00m,
    DeliveryTruck => 10.00m,

    { }     => throw new ArgumentException(message: "Not a known vehicle type",
paramName: nameof(vehicle)),
    null    => throw new ArgumentNullException(nameof(vehicle))
};
```

Muitos desses braços switch são exemplos de **padrões recursivos**. Por exemplo, `Car { Passengers: 1}` mostra um padrão constante dentro de um padrão de propriedade.

É possível fazer esse código menos repetitivo, usando switches aninhados. O `Car` e `Taxi` têm quatro braços diferentes nos exemplos anteriores. Em ambos os casos, você pode criar um padrão de declaração que alimenta um padrão constante. Essa técnica é mostrada no código a seguir:

C#

```
public decimal CalculateToll(object vehicle) =>
    vehicle switch
    {
        Car c => c.Passengers switch
        {
            0 => 2.00m + 0.5m,
            1 => 2.0m,
            2 => 2.0m - 0.5m,
            _ => 2.00m - 1.0m
```

```
        },

        Taxi t => t.Fares switch
        {
            0 => 3.50m + 1.00m,
            1 => 3.50m,
            2 => 3.50m - 0.50m,
            _ => 3.50m - 1.00m
        },

        Bus b when ((double)b.Riders / (double)b.Capacity) < 0.50 => 5.00m +
2.00m,
        Bus b when ((double)b.Riders / (double)b.Capacity) > 0.90 => 5.00m -
1.00m,
        Bus b => 5.00m,

        DeliveryTruck t when (t.GrossWeightClass > 5000) => 10.00m + 5.00m,
        DeliveryTruck t when (t.GrossWeightClass < 3000) => 10.00m - 2.00m,
        DeliveryTruck t => 10.00m,

        { }  => throw new ArgumentException(message: "Not a known vehicle type",
paramName: nameof(vehicle)),
        null => throw new ArgumentNullException(nameof(vehicle))
    };
```

Na amostra anterior, o uso de uma expressão recursiva significa não repetir os braços `Car` e `Taxi` contendo braços filho que testam o valor da propriedade. Essa técnica não é usada para os braços `Bus` e `DeliveryTruck` porque esses estão testando intervalos da propriedade, e não valores discretos.

Adicionar preço de horário de pico

Para um último recurso, a autoridade de pedágio quer adicionar um preço os horários de pico. Durante os horários de pico da manhã e do final da tarde, os pedágios serão dobrados. Essa regra afetará apenas o tráfego em uma direção: entrada para a cidade, no período da manhã, e de saída da cidade, no período da tarde. Em outros períodos durante o dia útil, os pedágios aumentam 50%. Nos períodos da noite e madrugada e de manhã cedo, as tarifas são 25% mais baratas. Durante o fim de semana, a taxa é normal, independentemente da hora. Você pode usar uma série de instruções `if` e `else` para expressar isso usando o seguinte código:

C#

```
public decimal PeakTimePremiumIfElse(DateTime timeOfToll, bool inbound)
{
    if ((timeOfToll.DayOfWeek == DayOfWeek.Saturday) ||
        (timeOfToll.DayOfWeek == DayOfWeek.Sunday))
    {
        return 1.0m;
```

```
    }
    else
    {
        int hour = timeOfToll.Hour;
        if (hour < 6)
        {
            return 0.75m;
        }
        else if (hour < 10)
        {
            if (inbound)
            {
                return 2.0m;
            }
            else
            {
                return 1.0m;
            }
        }
        else if (hour < 16)
        {
            return 1.5m;
        }
        else if (hour < 20)
        {
            if (inbound)
            {
                return 1.0m;
            }
            else
            {
                return 2.0m;
            }
        }
        else // Overnight
        {
            return 0.75m;
        }
    }
}
```

O código anterior funciona corretamente, mas não é legível. Você precisa encadear todos os casos de entrada e as instruções aninhadas `if` para raciocinar sobre o código. Em vez disso, você usará a correspondência de padrões para esse recurso, mas poderá integrá-lo a outras técnicas. É possível criar uma única expressão de correspondência de padrões que leva em conta todas as combinações de direção, dia da semana e hora. O resultado seria uma expressão complicada. Seria difícil de ler e entender. O que dificulta garantir a exatidão. Em vez disso, combine esses métodos para criar uma tupla de valores que descreve de forma concisa todos os estados. Em seguida, use a correspondência de padrões para calcular um multiplicador para o pedágio. A tupla contém três condições distintas:

- O dia é um dia da semana ou do fim de semana.
- A faixa de tempo é quando o pedágio é coletado.
- A direção é para a cidade ou da cidade

A tabela a seguir mostra as combinações de valores de entrada e multiplicador de preços para os horários de pico:

Expandir a tabela

Dia	Hora	Direção	Prêmio
Dia de Semana	horário de pico da manhã	entrada	x 2,00
Dia de Semana	horário de pico da manhã	saída	x 1,00
Dia de Semana	hora do dia	entrada	x 1,50
Dia de Semana	hora do dia	saída	x 1,50
Dia de Semana	horário de pico do fim da tarde	entrada	x 1,00
Dia de Semana	horário de pico do fim da tarde	saída	x 2,00
Dia de Semana	noite e madrugada	entrada	x 0,75
Dia de Semana	noite e madrugada	saída	x 0,75
Fim de Semana	horário de pico da manhã	entrada	x 1,00
Fim de Semana	horário de pico da manhã	saída	x 1,00
Fim de Semana	hora do dia	entrada	x 1,00
Fim de Semana	hora do dia	saída	x 1,00
Fim de Semana	horário de pico do fim da tarde	entrada	x 1,00
Fim de Semana	horário de pico do fim da tarde	saída	x 1,00
Fim de Semana	noite e madrugada	entrada	x 1,00
Fim de Semana	noite e madrugada	saída	x 1,00

Há 16 combinações diferentes das três variáveis. Ao combinar algumas das condições, você simplificará a expressão switch.

O sistema que coleta os pedágios usa uma estrutura DateTime para a hora em que o pedágio foi cobrado. Construa métodos de membro que criam as variáveis da tabela anterior. A seguinte função usa como correspondência de padrões a expressão switch para expressar se um DateTime representa um fim de semana ou um dia útil:

C#

```
private static bool IsWeekDay(DateTime timeOfToll) =>
    timeOfToll.DayOfWeek switch
    {
        DayOfWeek.Monday    => true,
        DayOfWeek.Tuesday   => true,
        DayOfWeek.Wednesday => true,
        DayOfWeek.Thursday  => true,
        DayOfWeek.Friday    => true,
        DayOfWeek.Saturday  => false,
        DayOfWeek.Sunday    => false
    };
```

Esse método está correto, mas é redundante. Para simplificar, faça conforme mostrado no código a seguir:

C#

```
private static bool IsWeekDay(DateTime timeOfToll) =>
    timeOfToll.DayOfWeek switch
    {
        DayOfWeek.Saturday => false,
        DayOfWeek.Sunday => false,
        _ => true
    };
```

Depois, adicione uma função semelhante para categorizar o tempo nos blocos:

C#

```
private enum TimeBand
{
    MorningRush,
    Daytime,
    EveningRush,
    Overnight
}

private static TimeBand GetTimeBand(DateTime timeOfToll) =>
    timeOfToll.Hour switch
    {
        < 6 or > 19 => TimeBand.Overnight,
        < 10 => TimeBand.MorningRush,
        < 16 => TimeBand.Daytime,
        _ => TimeBand.EveningRush,
    };
```

Adicione um `enum` privado para converter cada intervalo de tempo em um valor discreto. Em seguida, o método `GetTimeBand` usa padrões relacionais e padrões or conjuntivos. Um padrão relacional permite testar um valor numérico usando `<`, `>`, `<=` ou `>=`. O padrão `or` testa se uma expressão corresponde a um ou mais padrões. Você também pode usar um padrão `and` para garantir que uma expressão corresponda a dois padrões distintos e um padrão `not` para testar se uma expressão não corresponde a um padrão.

Depois de criar esses métodos, é possível usar outra expressão `switch` com o **padrão de tupla** para calcular o preço premium. Você pode construir uma expressão `switch` com todos os 16 braços:

C#

```
public decimal PeakTimePremiumFull(DateTime timeOfToll, bool inbound) =>
    (IsWeekDay(timeOfToll), GetTimeBand(timeOfToll), inbound) switch
    {
        (true, TimeBand.MorningRush, true) => 2.00m,
        (true, TimeBand.MorningRush, false) => 1.00m,
        (true, TimeBand.Daytime, true) => 1.50m,
        (true, TimeBand.Daytime, false) => 1.50m,
        (true, TimeBand.EveningRush, true) => 1.00m,
        (true, TimeBand.EveningRush, false) => 2.00m,
        (true, TimeBand.Overnight, true) => 0.75m,
        (true, TimeBand.Overnight, false) => 0.75m,
        (false, TimeBand.MorningRush, true) => 1.00m,
        (false, TimeBand.MorningRush, false) => 1.00m,
        (false, TimeBand.Daytime, true) => 1.00m,
        (false, TimeBand.Daytime, false) => 1.00m,
        (false, TimeBand.EveningRush, true) => 1.00m,
        (false, TimeBand.EveningRush, false) => 1.00m,
        (false, TimeBand.Overnight, true) => 1.00m,
        (false, TimeBand.Overnight, false) => 1.00m,
    };
```

O código acima funciona, mas pode ser simplificado. Todas as oito combinações para o fim de semana têm o mesmo pedágio. É possível substituir todas as oito pela seguinte linha:

C#

```
(false, _, _) => 1.0m,
```

Tanto o tráfego de entrada quanto o de saída têm o mesmo multiplicador durante o dia e a noite, nos dias úteis. Esses quatro braços switch podem ser substituídos por estas duas linhas:

C#

```
(true, TimeBand.Overnight, _) => 0.75m,
(true, TimeBand.Daytime, _)   => 1.5m,
```

O código deverá ser semelhante ao seguinte após essas duas alterações:

```C#
public decimal PeakTimePremium(DateTime timeOfToll, bool inbound) =>
    (IsWeekDay(timeOfToll), GetTimeBand(timeOfToll), inbound) switch
    {
        (true, TimeBand.MorningRush, true)  => 2.00m,
        (true, TimeBand.MorningRush, false) => 1.00m,
        (true, TimeBand.Daytime,     _)     => 1.50m,
        (true, TimeBand.EveningRush, true)  => 1.00m,
        (true, TimeBand.EveningRush, false) => 2.00m,
        (true, TimeBand.Overnight,   _)     => 0.75m,
        (false, _,                   _)     => 1.00m,
    };
```

Por fim, você pode remover os dois horários de pico em que é pago o preço normal. Quando remover essas braços, substitua o `false` por um descarte (`_`) no braço switch final. O método concluído será o seguinte:

```C#
public decimal PeakTimePremium(DateTime timeOfToll, bool inbound) =>
    (IsWeekDay(timeOfToll), GetTimeBand(timeOfToll), inbound) switch
    {
        (true, TimeBand.Overnight, _) => 0.75m,
        (true, TimeBand.Daytime, _) => 1.5m,
        (true, TimeBand.MorningRush, true) => 2.0m,
        (true, TimeBand.EveningRush, false) => 2.0m,
        _ => 1.0m,
    };
```

Este exemplo destaca uma das vantagens da correspondência de padrões: os branches de padrões são avaliados na ordem. Se você os reorganizar para que um branch anterior trate um dos casos posteriores, o compilador emitirá um aviso sobre o código inacessível. Essas regras de linguagem tornam as simplificações anteriores mais fáceis com a certeza de que o código não foi alterado.

A correspondência de padrões torna alguns tipos de código mais legíveis e oferece uma alternativa às técnicas orientadas a objeto quando não é possível adicionar o código às classes. A nuvem está fazendo com que os dados e a funcionalidade existam separadamente. A *forma* dos dados e as *operações* nela não são necessariamente descritas juntas. Neste tutorial, você utilizou os dados existentes de maneiras completamente diferentes de sua função original. A

correspondência de padrões proporcionou a capacidade de escrever a funcionalidade que substituiu esses tipos, ainda que não tenha sido possível estendê-los.

Próximas etapas

Baixe o código concluído no repositório dotnet/samples do GitHub. Explore os padrões por conta própria e adicione essa técnica em suas atividades regulares de codificação. Aprender essas técnicas lhe oferece outra maneira de abordar problemas e criar novas funcionalidades.

Confira também

- Padrões
- Expressão switch

Como manipular uma exceção usando try/catch

Artigo • 07/04/2023

A finalidade de um bloco try-catch é capturar e manipular uma exceção gerada pelo código de trabalho. Algumas exceções podem ser manipuladas em um bloco `catch` e o problema pode ser resolvido sem que a exceção seja gerada novamente. No entanto, com mais frequência, a única coisa que você pode fazer é certificar-se de que a exceção apropriada seja gerada.

Exemplo

Neste exemplo, IndexOutOfRangeException não é a exceção mais apropriada: ArgumentOutOfRangeException faz mais sentido para o método porque o erro é causado pelo argumento `index` passado pelo chamador.

C#

```
static int GetInt(int[] array, int index)
{
    try
    {
        return array[index];
    }
    catch (IndexOutOfRangeException e)  // CS0168
    {
        Console.WriteLine(e.Message);
        // Set IndexOutOfRangeException to the new exception's
InnerException.
        throw new ArgumentOutOfRangeException("index parameter is out of
range.", e);
    }
}
```

Comentários

O código que causa uma exceção fica dentro do bloco `try`. Uma instrução `catch` é adicionada imediatamente após para lidar com `IndexOutOfRangeException`, caso ocorra. O bloco `catch` manipula o `IndexOutOfRangeException` e gera o `ArgumentOutOfRangeException` mais apropriado. Para fornecer ao chamador tantas informações quanto possível, considere especificar a exceção original como o

InnerException da nova exceção. Uma vez que a propriedade InnerException é somente leitura, você precisa atribuí-la no construtor da nova exceção.

Como executar código de limpeza usando finally

Artigo • 12/03/2024

O propósito de uma instrução `finally` é garantir que a limpeza necessária de objetos, normalmente objetos que estão mantendo recursos externos, ocorra imediatamente, mesmo que uma exceção seja lançada. Um exemplo dessa limpeza é chamar Close em um FileStream imediatamente após o uso, em vez de esperar que o objeto passe pela coleta de lixo feita pelo Common Language Runtime, da seguinte maneira:

C#

```
static void CodeWithoutCleanup()
{
    FileStream? file = null;
    FileInfo fileInfo = new FileInfo("./file.txt");

    file = fileInfo.OpenWrite();
    file.WriteByte(0xF);

    file.Close();
}
```

Exemplo

Para transformar o código anterior em uma instrução `try-catch-finally`, o código de limpeza é separado do código funcional, da seguinte maneira.

C#

```
static void CodeWithCleanup()
{
    FileStream? file = null;
    FileInfo? fileInfo = null;

    try
    {
        fileInfo = new FileInfo("./file.txt");

        file = fileInfo.OpenWrite();
        file.WriteByte(0xF);
    }
    catch (UnauthorizedAccessException e)
    {
        Console.WriteLine(e.Message);
```

```
    }
    finally
    {
        file?.Close();
    }
}
```

Como uma exceção pode ocorrer a qualquer momento no bloco `try` antes da chamada `OpenWrite()` ou a própria chamada `OpenWrite()` poderia falhar, não há garantia de que o arquivo esteja aberto quanto tentarmos fechá-lo. O bloco `finally` adiciona uma verificação para checar se o objeto FileStream não é `null` antes de chamar o método Close. Sem a verificação de `null`, o bloco `finally` poderia gerar uma NullReferenceException própria, mas a geração de exceções em blocos `finally` deve ser evitada, se possível.

Uma conexão de banco de dados é outra boa candidata a ser fechada em um bloco `finally`. Como o número de conexões permitidas para um servidor de banco de dados é, às vezes, limitado, você deve fechar conexões de banco de dados assim que possível. Se uma exceção for gerada antes que você possa fechar a conexão, usar o bloco `finally` será melhor do que aguardar a coleta de lixo.

Confira também

- Instrução using
- Instruções para manipulação de exceções

Colaborar conosco no GitHub

A fonte deste conteúdo pode ser encontrada no GitHub, onde você também pode criar e revisar problemas e solicitações de pull. Para obter mais informações, confira o nosso guia para colaboradores.

.NET

Comentários do .NET

O .NET é um projeto código aberto. Selecione um link para fornecer comentários:

Abrir um problema de documentação

Fornecer comentários sobre o produto

Novidades no C# 14

13/06/2025

O C# 14 inclui os novos recursos a seguir. Você pode experimentar esses recursos usando a versão mais recente do Visual Studio 2022 ou o SDK do .NET 10:

- Membros de extensão
- Atribuição condicional nula
- nameof dá suporte a tipos genéricos não associados
- Conversões mais implícitas para Span<T> e ReadOnlySpan<T>
- Modificadores em parâmetros lambda simples
- Propriedades com suporte de field
- eventos e construtores partial
- Operadores de atribuição composta definidos pelo usuário

O C# 14 tem suporte no **.NET 10**. Para obter mais informações, confira Controle de versão da linguagem C#.

Você pode baixar o SDK do .NET 10 mais recente na página de downloads do .NET. Você também pode baixar o Visual Studio 2022, que inclui o SDK do .NET 10.

Novos recursos são adicionados à página "Novidades no C#" quando estão disponíveis em versões de visualização pública. A seção conjunto de trabalho da página de status do recurso roslyn acompanha quando os recursos futuros são mesclados na ramificação principal. Este artigo foi atualizado pela última vez para o .NET 10 Versão Prévia 1.

Você encontra alterações interruptivas introduzidas no C# 14 em nosso artigo sobre alterações interruptivas.

> **Observação**
>
> Estamos interessados em seus comentários sobre esses recursos. Se você encontrar problemas com qualquer um desses novos recursos, crie um problema no repositório dotnet/roslyn.

Membros de extensão

O C# 14 adiciona uma nova sintaxe para definir *membros de extensão*. A nova sintaxe permite que você declare propriedades de *extensão* além dos métodos de extensão. Você também pode declarar membros de extensão que estendem o tipo, em vez de uma instância do tipo. Em outras palavras, esses novos membros de extensão podem aparecer como membros

estáticos do tipo que você estende. O exemplo de código a seguir mostra um exemplo dos diferentes tipos de membros de extensão que você pode declarar:

```C#
public static class Enumerable
{
    // Extension block
    extension<TSource>(IEnumerable<TSource> source) // extension members for
IEnumerable<TSource>
    {
        // Extension property:
        public bool IsEmpty => !source.Any();
        // Extension indexer:
        public TSource this[int index] => source.Skip(index).First();

        // Extension method:
        public IEnumerable<TSource> Where(Func<TSource, bool> predicate) { ... }
    }

    // extension block, with a receiver type only
    extension<TSource>(IEnumerable<TSource>) // static extension members for
IEnumerable<Source>
    {
        // static extension method:
        public static IEnumerable<TSource> Combine(IEnumerable<TSource> first,
IEnumerable<TSource> second) { ... }

        // static extension property:
        public static IEnumerable<TSource> Identity => Enumerable.Empty<TSource>
();
    }
}
```

Os membros no primeiro bloco de extensão são chamados como se fossem membros da instância de `IEnumerable<TSource>`, por exemplo `sequence.IsEmpty`. Os membros no segundo bloco de extensão são chamados como se fossem membros estáticos de `IEnumerable<TSource>`, por exemplo `IEnumerable<int>.Identity`.

Você pode saber mais detalhes lendo o artigo sobre membros de extensão no guia de programação, o artigo de referência de idioma sobre a extension palavra-chave e a especificação do novo recurso de membros de extensão.

A palavra-chave `field`

O token `field` permite que você escreva o corpo de um acessador de propriedade sem declarar um campo de apoio explícito. O token `field` é substituído por um campo de backup sintetizado pelo compilador.

Por exemplo, no passado, se você quisesse garantir que uma `string` propriedade não pudesse ser definida para `null`, você teria que declarar um campo de apoio e implementar ambos os métodos de acesso.

```
C#

private string _msg;
public string Message
{
    get => _msg;
    set => _msg = value ?? throw new ArgumentNullException(nameof(value));
}
```

Agora você pode simplificar seu código para:

```
C#

public string Message
{
    get;
    set => field = value ?? throw new ArgumentNullException(nameof(value));
}
```

Você pode declarar um corpo para um ou ambos os acessadores de uma propriedade de campo.

Há uma possível alteração incompatível ou confusão ao ler código em tipos que também incluem um símbolo chamado `field`. Você pode usar `@field` ou `this.field` desambiguar entre a `field` palavra-chave e o identificador ou renomear o símbolo atual `field` para fornecer melhor distinção.

Se você experimentar esse recurso e tiver comentários, comente sobre o problema do recurso no `csharplang` repositório.

A palavra-chave contextual field está em C# 13 como um recurso de visualização.

Conversões de intervalo implícitas

O C# 14 apresenta suporte de primeira classe para System.Span<T> e System.ReadOnlySpan<T> no idioma. Esse suporte envolve novas conversões implícitas que permitem uma programação mais natural com esses tipos.

`Span<T>` e `ReadOnlySpan<T>` são usados de várias maneiras principais em C# e no runtime. Sua introdução melhora o desempenho sem arriscar a segurança. O C# 14 reconhece a relação e dá

suporte a algumas conversões entre `ReadOnlySpan<T>`, `Span<T>` e `T[]`. Os tipos de alcance podem ser receptores de métodos de extensão, se compor com outras conversões e auxiliar em cenários de inferência de tipo genérico.

Você pode encontrar a lista de conversões implícitas de span no artigo sobre tipos internos na seção de referência da linguagem. Você pode saber mais detalhes lendo a especificação da funcionalidade para tipos de segmento de primeira classe.

Tipos genéricos não associados e `nameof`

A partir do C# 14, o argumento de `nameof` pode ser um tipo genérico não vinculado. Por exemplo, `nameof(List<>)` é avaliado como `List`. Em versões anteriores do C#, somente tipos genéricos fechados, como `List<int>`, poderiam ser usados para retornar o `List` nome.

Parâmetros lambda simples com modificadores

Você pode adicionar modificadores de parâmetro, como `scoped`, `ref`, `in`, `out` ou `ref readonly` a parâmetros de expressão lambda sem especificar o tipo de parâmetro:

C#

```
delegate bool TryParse<T>(string text, out T result);
// ...
TryParse<int> parse1 = (text, out result) => Int32.TryParse(text, out result);
```

Anteriormente, a adição de modificadores só era permitida quando as declarações de parâmetro incluíam os tipos para os parâmetros. A declaração anterior exigiria tipos em todos os parâmetros:

C#

```
TryParse<int> parse2 = (string text, out int result) => Int32.TryParse(text, out
result);
```

O `params` modificador ainda requer uma lista de parâmetros explicitamente tipada.

Você pode ler mais sobre essas alterações no artigo sobre expressões lambda na referência de linguagem C#.

Membros com mais parcialidade

Agora você pode declarar construtores de instância e eventos como membros parciais.

Construtores parciais e eventos parciais devem incluir exatamente uma *declaração de definição* e uma *declaração de implementação*.

Somente a declaração de implementação de um construtor parcial pode incluir um inicializador de construtor: `this()` ou `base()`. Somente uma declaração de tipo parcial pode incluir a sintaxe do construtor primário.

A declaração de implementação de um evento parcial deve incluir os acessadores `add` e `remove`. A declaração de definição declara um evento semelhante a um campo.

Atribuição composta definida pelo usuário

Saiba mais na especificação do recurso para atribuição composta definida pelo usuário.

Atribuição condicional nula

Os operadores de acesso a membros com condição nula, `?.` e `?[]`, podem agora ser usados no lado esquerdo de uma atribuição ou atribuição composta.

Antes do C# 14, você precisava verificar se uma variável é nula antes de atribuir a uma propriedade.

```C#
if (customer is not null)
{
    customer.Order = GetCurrentOrder();
}
```

Você pode simplificar o código anterior usando o `?.` operador:

```C#
customer?.Order = GetCurrentOrder();
```

O lado direito do `=` operador é avaliado somente quando o lado esquerdo não é nulo. Se `customer` for nulo, o código não chamará `GetCurrentOrder`.

Além da atribuição, você pode usar operadores de acesso de membro condicional nulo com operadores de atribuição composta (`+=`, `-=` e outros). No entanto, incrementar e diminuir, `++` e `--` não são permitidos.

Saiba mais no artigo de referência de idioma sobre o acesso condicional ao membro e a especificação de recurso para atribuição condicional nula.

Consulte também

- Novidades no .NET 10

Novidades no C# 13

29/05/2025

O C# 13 inclui os novos recursos a seguir. Você pode experimentar esses recursos usando a versão mais recente do Visual Studio 2022 ou o SDK do .NET 9 :

- params coleções
- Novo tipo lock e semântica.
- Nova sequência de escape – \e.
- Melhorias do tipo natural do grupo de métodos
- Acesso implícito de indexador em inicializadores de objeto
- Habilitar locais ref e contextos unsafe em iteradores e métodos assíncronos
- Habilitar tipos ref struct para implementar interfaces.
- Permitir tipos ref struct como argumentos para parâmetros de tipo em programação genérica.
- propriedades parciais e indexadores agora são permitidos em tipos de `partial`.
- A prioridade de resolução de sobrecarga permite que os autores da biblioteca designem uma sobrecarga como melhor do que as outras.

A partir do Visual Studio 17.12, o C# 13 inclui a palavra-chave contextual field como um recurso de visualização.

O C# 13 tem suporte em **.NET 9**. Para obter mais informações, confira Controle de versão da linguagem C#.

Você pode baixar o SDK do .NET 9 mais recente na página de downloads do .NET . Você também pode baixar do Visual Studio 2022, que inclui o SDK do .NET 9.

Você encontra alterações interruptivas introduzidas no C# 13 em nosso artigo sobre alterações interruptivas.

> **Observação**
>
> Estamos interessados em seus comentários sobre esses recursos. Se você encontrar problemas com qualquer um desses novos recursos, crie um **problema** no repositório **dotnet/roslyn**.

Coleções `params`

O modificador de `params` não se limita aos tipos de matriz. Agora você pode usar `params` com qualquer tipo de coleção reconhecido, incluindo System.Span<T>, System.ReadOnlySpan<T>e

tipos que implementam System.Collections.Generic.IEnumerable<T> e têm um método `Add`. Além dos tipos concretos, as interfaces System.Collections.Generic.IEnumerable<T>, System.Collections.Generic.IReadOnlyCollection<T>, System.Collections.Generic.IReadOnlyList<T>, System.Collections.Generic.ICollection<T>e System.Collections.Generic.IList<T> também podem ser usadas.

Quando um tipo de interface é usado, o compilador sintetiza o armazenamento para os argumentos fornecidos. Saiba mais sobre a especificação do recurso para coleções params.

Por exemplo, as declarações de método podem declarar intervalos como parâmetros `params`:

C#

```
public void Concat<T>(params ReadOnlySpan<T> items)
{
    for (int i = 0; i < items.Length; i++)
    {
        Console.Write(items[i]);
        Console.Write(" ");
    }
    Console.WriteLine();
}
```

Novo objeto de bloqueio

O runtime do .NET 9 inclui um novo tipo para sincronização de thread, o tipo System.Threading.Lock. Esse tipo fornece melhor sincronização de thread por meio de sua API. O método Lock.EnterScope() insere um escopo exclusivo. O `ref struct` retornado dá suporte ao padrão `Dispose()` para sair do escopo exclusivo.

A instrução C# lock reconhece se o alvo do bloqueio é um objeto `Lock`. Nesse caso, ele usa a API atualizada, em vez da API tradicional usando System.Threading.Monitor. O compilador também reconhece se você converter um objeto `Lock` em outro tipo e o código baseado em `Monitor` será gerado. Você pode ler mais na especificação de funcionalidade do novo objeto de bloqueio .

Esse recurso permite que você, ao alterar o tipo de objeto `lock`, obtenha os benefícios do novo tipo de biblioteca. Nenhum outro código precisa ser alterado.

Nova sequência de escape

Você pode usar `\e` como uma sequência de escape de literal de caractere para o caractere `ESCAPE`, `U+001B`Unicode. Anteriormente, você usava `\u001b` ou `\x1b`. Usar `\x1b` não foi

recomendado porque se os próximos caracteres após `1b` forem dígitos hexadecimal válidos, esses caracteres se tornarão parte da sequência de escape.

Tipo natural do grupo de métodos

Esse recurso faz pequenas otimizações para sobrecarregar a resolução envolvendo grupos de métodos. Um *grupo de métodos* é um método, juntamente com todas as suas sobrecargas com o mesmo nome. O comportamento anterior era que o compilador construísse o conjunto completo de métodos candidatos para um grupo de métodos. Se um tipo natural fosse necessário, o tipo natural era determinado a partir do conjunto completo de métodos candidatos.

O novo comportamento é podar o conjunto de métodos candidatos em cada escopo, removendo os métodos candidatos que não são aplicáveis. Normalmente, os métodos removidos são métodos genéricos com a aridade errada ou restrições que não são atendidas. O processo continuará para o próximo escopo externo somente se nenhum método candidato for encontrado. Esse processo segue mais de perto o algoritmo geral para resolução de sobrecarga. Se todos os métodos candidatos encontrados em um determinado escopo não corresponderem, o grupo de métodos não terá um tipo natural.

Você pode ler os detalhes das alterações na especificação da proposta.

Acesso implícito ao índice

O operador de índice "do final" implícito, `^`, agora é permitido em uma expressão de inicializador de objeto para coleções unidimensionais. Por exemplo, agora você pode inicializar uma matriz de dimensão única usando um inicializador de objeto, conforme mostrado no código a seguir:

C#

```
public class TimerRemaining
{
    public int[] buffer { get; set; } = new int[10];
}

var countdown = new TimerRemaining()
{
    buffer =
    {
        [^1] = 0,
        [^2] = 1,
        [^3] = 2,
        [^4] = 3,
        [^5] = 4,
```

```
        [^6] = 5,
        [^7] = 6,
        [^8] = 7,
        [^9] = 8,
        [^10] = 9
    }
};
```

A classe `TimerRemaining` inclui uma matriz de `buffer` inicializada em um comprimento de 10. O exemplo anterior atribui valores a essa matriz usando o operador de índice "do final" (`^`), criando efetivamente uma matriz que conta de 9 para 0.

Em versões antes do C# 13, o operador `^` não pode ser usado em um inicializador de objeto. Você precisa indexar os elementos da frente.

`ref` e `unsafe` em iteradores e métodos de `async`

Esse recurso e os dois recursos a seguir permitem que `ref struct` tipos usem novos constructos. Você não usará esses recursos, a menos que crie seus próprios tipos de `ref struct`. É mais provável que você veja um benefício indireto à medida que System.Span<T> e System.ReadOnlySpan<T> obtêm mais funcionalidade.

Antes do C# 13, os métodos de iterador (métodos que usam `yield return`) e métodos `async` não podiam declarar variáveis de `ref` locais nem poderiam ter um contexto `unsafe`.

Na versão C# 13, métodos `async` podem declarar `ref` variáveis locais, ou variáveis locais de tipo `ref struct`. No entanto, essas variáveis não podem ser acessadas em um limite `await`. Eles também não podem ser acessados em um limite `yield return`.

Essa restrição descontraída permite que o compilador autorize o uso seguro de variáveis locais `ref` e tipos `ref struct` em mais locais. Você pode usar com segurança tipos como System.ReadOnlySpan<T> nesses métodos. O compilador informa se você viola as regras de segurança.

Da mesma forma, o C# 13 permite contextos `unsafe` em métodos de iterador. Entretanto, todas as instruções `yield return` e `yield break` devem estar em contextos seguros.

`allows ref struct`

Antes do C# 13, `ref struct` tipos não podiam ser declarados como o argumento de tipo para um tipo ou método genérico. Agora, declarações de tipo genérico podem adicionar uma anti-restrição, `allows ref struct`. Essa antirrestringência declara que o argumento de tipo

fornecido para esse parâmetro de tipo pode ser do tipo `ref struct`. O compilador impõe regras de segurança ref em todas as instâncias desse parâmetro de tipo.

Por exemplo, você pode declarar um tipo genérico como o seguinte código:

```C#
public class C<T> where T : allows ref struct
{
    // Use T as a ref struct:
    public void M(scoped T p)
    {
        // The parameter p must follow ref safety rules
    }
}
```

Isso permite que tipos como System.Span<T> e System.ReadOnlySpan<T> sejam usados com algoritmos genéricos, quando aplicável. Você pode saber mais nas atualizações para where e no artigo do guia de programação sobre restrições genéricas .

Interfaces `ref struct`

Antes do C# 13, os tipos de `ref struct` não podiam implementar interfaces. A partir do C# 13, eles podem. Você pode declarar que um tipo de `ref struct` implementa uma interface. Mas para garantir a segurança das regras de ref, um tipo `ref struct` não pode ser convertido em um tipo de interface. Essa conversão é de boxing e pode violar a segurança de ref. Declarações de método de interface explícitas em `ref struct` só podem ser acessadas através de um parâmetro de tipo, que seja allows ref struct. Além disso, os tipos `ref struct` devem implementar todos os métodos declarados em uma interface, incluindo esses métodos com uma implementação padrão.

Saiba mais nas atualizações sobre ref struct tipos e a adição da restrição genérica allows ref struct.

Membros com mais parcialidade

Você pode declarar propriedades `partial` e indexadores `partial` no C# 13. As propriedades parciais e os indexadores geralmente seguem as mesmas regras que os métodos `partial`: você cria uma declaração *para declarar* e uma declaração *para implementar*. As assinaturas das duas declarações devem corresponder. Uma restrição é que você não pode usar uma declaração de propriedade automática para *implementar* uma propriedade parcial. As propriedades que não definem um corpo são consideradas a *declaração correspondente*.

```C#
public partial class C
{
    // Declaring declaration
    public partial string Name { get; set; }
}

public partial class C
{
    // implementation declaration:
    private string _name;
    public partial string Name
    {
        get => _name;
        set => _name = value;
    }
}
```

Saiba mais no artigo sobre membros parciais.

Prioridade de resolução de sobrecarga

No C# 13, o compilador reconhece a OverloadResolutionPriorityAttribute para preferir uma sobrecarga em vez de outra. Os autores da biblioteca podem usar esse atributo para garantir que uma sobrecarga nova e melhor seja preferida em vez de uma sobrecarga existente. Por exemplo, você pode adicionar uma nova sobrecarga que seja mais eficiente. Você não deseja interromper o código existente que usa sua biblioteca, mas deseja que os usuários atualizem para a nova versão quando eles recompilarem. Você pode usar Prioridade na resolução de sobrecarga para informar ao compilador qual sobrecarga deve ser priorizada. As sobrecargas com a prioridade mais alta são preferenciais.

Esse recurso destina-se aos autores da biblioteca para evitar ambiguidade ao adicionar novas sobrecargas. Os autores da biblioteca devem ter cuidado com esse atributo para evitar confusão.

A palavra-chave `field`

A palavra-chave contextual field está em C# 13 como um recurso de visualização. O token `field` acessa o campo de backup sintetizado do compilador em um acessador de propriedade. Ele permite que você escreva um corpo acessador sem declarar um campo de backup explícito em sua declaração de tipo. Você pode declarar um corpo para um ou ambos os acessadores de uma propriedade de campo.

O recurso `field` é lançado como um recurso em versão preliminar. Queremos aprender com suas experiências usando-a. Há uma possível alteração incompatível ou confusão ao ler código em tipos que também incluem um campo chamado `field`. Você pode usar `@field` ou `this.field` para desambiguar entre a palavra-chave `field` e o identificador.

> **Importante**
>
> A palavra-chave `field` é um recurso de visualização no C# 13. Você deve usar o .NET 9 e definir seu elemento `<LangVersion>` para `preview` no arquivo de projeto para usar a palavra-chave contextual `field`.
>
> Você deve ter cuidado ao usar o recurso de palavra-chave `field` em uma classe que tenha um campo chamado `field`. A nova palavra-chave `field` oculta um campo chamado `field` dentro do escopo de um acessador de propriedade. Você pode alterar o nome da variável `field` ou usar o token `@` para referenciar o identificador `field` como `@field`. Para saber mais, leia a especificação do recurso para a palavra-chave field.

Se você testar esse recurso e receber feedback, adicione-o ao problema de recurso no repositório `csharplang`.

Consulte também

- Novidades no .NET 9

Novidades do C# 12

Artigo • 21/03/2024

O C# 12 inclui os novos recursos a seguir. Você pode testar esses recursos usando a versão mais recente do Visual Studio 2022 ou o SDK do.NET 8.

- Construtores primários: introduzidos no Visual Studio 2022 versão 17.6 versão prévia 2.
- Expressões de coleção: introduzidas no Visual Studio 2022 versão 17.7 versão prévia 5.
- Matrizes embutidas: introduzidas no Visual Studio 2022 versão 17.7 versão prévia 3.
- Parâmetros opcionais em expressões lambda: introduzidos no Visual Studio 2022 versão 17.5, versão prévia 2.
- parâmetros ref readonly: introduzidos no Visual Studio 2022 versão 17.8 versão prévia 2.
- Alias qualquer tipo: introduzido no Visual Studio 2022 versão 17.6 versão prévia 3.
- Atributo experimental: introduzido no Visual Studio 2022 versão 17.7 versão prévia 3.
- Interceptores - *versão prévia do recurso* introduzida no Visual Studio 2022 versão 17.7 versão prévia 3.

C# 12 é compatível com **.NET 8**. Para obter mais informações, consulte Controle de versão da linguagem C#.

Você pode baixar o SDK do .NET 8 mais recente na página de downloads do .NET. Você também pode baixar o Visual Studio 2022, que inclui o SDK do .NET 8.

> **Observação**
>
> Estamos interessados em seus comentários sobre esses recursos. Se você encontrar problemas com qualquer um desses novos recursos, crie um **problema** no repositório **dotnet/roslyn**.

Construtores primários

Agora, você pode criar construtores primários em qualquer `class` e `struct`. Os construtores primários não são mais restritos a tipos `record`. Os parâmetros do construtor primário estão no escopo de todo o corpo da classe. Para garantir que todos os parâmetros do construtor primário sejam atribuídos definitivamente, todos os construtores declarados explicitamente devem chamar o construtor primário usando a sintaxe `this()`. Adicionar um construtor primário a um `class` impede que o compilador declare um construtor implícito sem parâmetros. Em um `struct`, o construtor implícito sem parâmetros inicializa todos os campos, incluindo parâmetros de construtor primário para o padrão de 0 bits.

O compilador gera propriedades públicas para parâmetros de construtor primários somente em tipos `record`, tipos `record class` ou `record struct`. Classes e structs não registrados podem nem sempre querer esse comportamento para parâmetros de construtor primário.

Você pode saber mais sobre construtores primários no tutorial para explorar construtores primários e no artigo sobre construtores de instâncias.

Expressões de coleção

As expressões de coleção apresentam uma nova sintaxe concisa para criar valores comuns de coleção. É possível incorporar outras coleções nesses valores usando um operador spread `..`.

Vários tipos semelhantes à coleção podem ser criados sem a necessidade de suporte a BCL externo. Esses tipos são:

- Tipos de matriz, como `int[]`.
- System.Span<T> e System.ReadOnlySpan<T>.
- Tipos que dão suporte a inicializadores de coleção, como System.Collections.Generic.List<T>.

Os exemplos a seguir mostram usos de expressões de coleção:

C#

```
// Create an array:
int[] a = [1, 2, 3, 4, 5, 6, 7, 8];

// Create a list:
List<string> b = ["one", "two", "three"];

// Create a span
Span<char> c  = ['a', 'b', 'c', 'd', 'e', 'f', 'h', 'i'];
```

```
// Create a jagged 2D array:
int[][] twoD = [[1, 2, 3], [4, 5, 6], [7, 8, 9]];

// Create a jagged 2D array from variables:
int[] row0 = [1, 2, 3];
int[] row1 = [4, 5, 6];
int[] row2 = [7, 8, 9];
int[][] twoDFromVariables = [row0, row1, row2];
```

O operador *spread*, `..` em uma expressão de coleção substitui seu argumento pelos elementos dessa coleção. O argumento deve ser um tipo de coleção. Os exemplos a seguir mostram como o operador de spread funciona:

C#

```
int[] row0 = [1, 2, 3];
int[] row1 = [4, 5, 6];
int[] row2 = [7, 8, 9];
int[] single = [.. row0, .. row1, .. row2];
foreach (var element in single)
{
    Console.Write($"{element}, ");
}
// output:
// 1, 2, 3, 4, 5, 6, 7, 8, 9,
```

O operando de um operador spread é uma expressão que pode ser enumerada. O operador spread avalia cada elemento da expressão de enumerações.

Você pode usar expressões de coleção em qualquer lugar que precisar de uma coleção de elementos. Eles podem especificar o valor inicial de uma coleção ou ser passados como argumentos para métodos que utilizam tipos de coleção. Você pode saber mais sobre expressões de coleção no artigo de referência de linguagem sobre expressões de coleção ou a especificação do recurso.

Parâmetros `ref readonly`

O C# adicionou parâmetros `in` como uma maneira de passar referências somente leitura. Os parâmetros `in` permitem variáveis e valores e podem ser usados sem nenhuma anotação nos argumentos.

A adição de parâmetros `ref readonly` permite mais clareza para APIs que podem estar usando parâmetros `ref` ou parâmetros `in`:

- As APIs criadas antes da introdução de `in` podem usar `ref`, mesmo que o argumento não tenha sido modificado. Essas APIs podem ser atualizadas com `ref readonly`. Não será uma alteração significativa para os chamadores, como seria se o parâmetro `ref` fosse alterado para `in`. Um exemplo é System.Runtime.InteropServices.Marshal.QueryInterface.
- APIs que tomam um parâmetro `in`, mas logicamente exigem uma variável. Uma expressão de valor não funciona. Um exemplo é System.ReadOnlySpan<T>.ReadOnlySpan<T>(T).
- APIs que usam `ref` porque exigem uma variável, mas não modificam essa variável. Um exemplo é System.Runtime.CompilerServices.Unsafe.IsNullRef.

Para saber mais sobre parâmetros `ref readonly`, consulte o artigo sobre modificadores de parâmetro na referência de idioma ou a especificação de recurso de parâmetros somente leitura ref.

Parâmetros lambda padrão

Agora, você pode definir valores padrão para parâmetros em expressões lambda. A sintaxe e as regras são iguais à adição de valores padrão para argumentos a qualquer método ou função local.

Saiba mais sobre parâmetros padrão em expressões lambda no artigo sobre expressões lambda.

Alias de qualquer tipo

Você pode usar a diretiva de alias `using` para alias de qualquer tipo, não apenas tipos nomeados. Isso significa que você pode criar aliases semânticos para tipos de tupla, tipos de matriz, tipos de ponteiro ou outros tipos não seguros. Para obter mais informações, confira a especificação de recurso.

Matrizes embutidas

Matrizes embutidas são usadas pela equipe de runtime e outros autores de biblioteca para melhorar o desempenho em seus aplicativos. Matrizes embutidas permitem que um desenvolvedor crie uma matriz de tamanho fixo em um tipo `struct`. Um struct com um buffer embutido deve fornecer características de desempenho semelhantes a um buffer de tamanho fixo não seguro. Você provavelmente não declarará suas próprias matrizes embutidas, mas as usará de forma transparente quando elas forem expostas como System.Span<T> ou System.ReadOnlySpan<T> objetos de APIs de runtime.

Uma *matriz embutida* é declarada semelhante à seguinte `struct`:

```C#
[System.Runtime.CompilerServices.InlineArray(10)]
public struct Buffer
{
    private int _element0;
}
```

Você os usa como qualquer outra matriz:

```C#
var buffer = new Buffer();
for (int i = 0; i < 10; i++)
{
    buffer[i] = i;
}

foreach (var i in buffer)
{
    Console.WriteLine(i);
}
```

A diferença é que o compilador pode aproveitar as informações conhecidas sobre uma matriz embutida. Você provavelmente consumirá matrizes embutidas como faria com qualquer outra matriz. Para obter mais informações sobre como declarar matrizes embutidas, confira a referência de linguagem sobre tiposstruct.

Atributo experimental

Tipos, métodos ou assemblies podem ser marcados com a indicação System.Diagnostics.CodeAnalysis.ExperimentalAttribute de um recurso experimental. O compilador emitirá um aviso se você acessar um método ou digitar anotado com o ExperimentalAttribute. Todos os tipos incluídos em um assembly marcado com o atributo `Experimental` são experimentais. Você pode ler mais no artigo sobre Atributos gerais lidos pelo compilador ou na especificação de recursos.

Interceptores

> ⚠ **Aviso**

Os interceptores são um recurso experimental, disponível no modo de visualização com C# 12. O recurso pode estar sujeito a alterações significativas ou à remoção em uma versão futura. Portanto, não é recomendável para aplicativos de produção ou liberados.

Para usar interceptadores, o projeto do usuário deve especificar a propriedade `<InterceptorsPreviewNamespaces>`. Essa é uma lista de namespaces que têm permissão para conter interceptadores.

Por exemplo:
`<InterceptorsPreviewNamespaces>$(InterceptorsPreviewNamespaces);Microsoft.AspNetCore.Http.Generated;MyLibrary.Generated</InterceptorsPreviewNamespaces>`

Um *interceptador* é um método que pode substituir declarativamente uma chamada a um método *interceptável* por uma chamada a si mesmo em tempo de compilação. Essa substituição ocorre pelo fato de o interceptador declarar os locais de origem das chamadas que ele intercepta. Os interceptadores fornecem uma instalação limitada para alterar a semântica do código existente adicionando um novo código a uma compilação, por exemplo, em um gerador de origem.

Você usa um *interceptador* como parte de um gerador de origem para modificar uma compilação de origem existente, em vez de adicionar código a ela. O gerador de fontes de dados substitui as chamadas para um método interceptável por uma chamada para o método *interceptor*.

Se você estiver interessado em experimentar interceptadores, saiba mais lendo a especificação do recurso. Se você usar o recurso, certifique-se de estar atualizado com todas as alterações na especificação do recurso para esse recurso experimental. Se o recurso for finalizado, adicionaremos mais diretrizes neste site.

Confira também

- Novidades do .NET 8

Colaborar conosco no GitHub

A fonte deste conteúdo pode ser encontrada no GitHub, onde você também pode criar e revisar problemas e solicitações

.NET

Comentários do .NET

O .NET é um projeto código aberto. Selecione um link para fornecer comentários:

de pull. Para obter mais informações, confira o nosso guia para colaboradores.

Abrir um problema de documentação

Fornecer comentários sobre o produto

Novidades do C# 11

Artigo • 15/03/2024

Os seguintes recursos foram adicionados em C# 11:

- Literais brutos de cadeia de caracteres
- Suporte matemático genérico
- Atributos genéricos
- Cadeia de caracteres UTF-8 literais
- Linhas novas em expressões de interpolação de cadeia de caracteres
- Padrões de lista
- Tipos de locais de arquivos
- Membros necessários
- Structs de padrão automático
- Correspondência de padrão Span<char> em uma constante string
- Escopo estendido nameof
- IntPtr numérico
- Campos ref e scoped ref
- Conversão aprimorada do grupo de métodos para delegado
- Ciclo de aviso 7

O C# 11 tem suporte no **.NET 7**. Para obter mais informações, consulte Controle de versão da linguagem C#.

Você pode baixar o SDK mais recente do .NET 7 na página de downloads do .NET. Você também pode baixar o Visual Studio 2022, que inclui o SDK do .NET 7.

> **Observação**
>
> Estamos interessados em seus comentários sobre esses recursos. Se você encontrar problemas com qualquer um desses novos recursos, crie um **problema** no repositório **dotnet/roslyn**.

Atributos genéricos

Você pode declarar uma classe genérica cuja classe base é System.Attribute. Este recurso fornece uma sintaxe mais conveniente para atributos que exigem um parâmetro System.Type. Anteriormente, você precisaria criar um atributo com um `Type` como parâmetro de construtor:

```C#
// Before C# 11:
public class TypeAttribute : Attribute
{
   public TypeAttribute(Type t) => ParamType = t;

   public Type ParamType { get; }
}
```

E para aplicar o atributo, você usa o operador typeof:

```C#
[TypeAttribute(typeof(string))]
public string Method() => default;
```

Usando esse novo recurso, você pode criar um atributo genérico em vez disso:

```C#
// C# 11 feature:
public class GenericAttribute<T> : Attribute { }
```

Em seguida, especifique o tipo de parâmetro para usar o atributo:

```C#
[GenericAttribute<string>()]
public string Method() => default;
```

Você deve fornecer todos os parâmetros de tipo ao aplicar o atributo. Em outras palavras, o tipo genérico deve ser totalmente construído. No exemplo acima, os parênteses vazios (`(` e `)`) podem ser omitidos, pois o atributo não tem argumentos.

```C#
public class GenericType<T>
{
   [GenericAttribute<T>()] // Not allowed! generic attributes must be fully
constructed types.
   public string Method() => default;
}
```

Os argumentos de tipo devem atender às mesmas restrições do operador typeof. Tipos que exigem anotações de metadados não são permitidos. Por exemplo, os seguintes

tipos não são permitidos como o parâmetro de tipo:

- `dynamic`
- `string?` (ou qualquer tipo de referência anulável)
- `(int X, int Y)` (ou qualquer outro tipo de tupla usando a sintaxe de tupla C#).

Esses tipos não são representados diretamente em metadados. Elas incluem anotações que descrevem o tipo. Em todos os casos, você pode usar o tipo subjacente em vez disso:

- `object` para `dynamic`.
- `string` em vez de `string?`.
- `ValueTuple<int, int>` em vez de `(int X, int Y)`.

Suporte matemático genérico

Há vários recursos de linguagem que permitem suporte matemático genérico:

- Membros `static virtual` em interfaces
- operadores verificados definidos pelo usuário
- operadores de deslocamento flexíveis
- operador de deslocamento para a direita sem sinal

Você pode adicionar membros `static abstract` ou `static virtual` em interfaces para definir interfaces que incluam operadores sobrecarregados, outros membros estáticos e propriedades estáticas. O cenário principal para esse recurso é usar operadores matemáticos em tipos genéricos. Por exemplo, você pode implementar a interface `System.IAdditionOperators<TSelf, TOther, TResult>` em um tipo que implementa o `operator +`. Outras interfaces definem outras operações matemáticas ou valores bem definidos. Você pode aprender sobre a nova sintaxe no artigo sobre interfaces. Interfaces que incluem métodos `static virtual` normalmente são interfaces genéricas. Além disso, a maioria declarará uma restrição de que o parâmetro de tipo implemente a interface declarada.

Você pode saber mais e experimentar o recurso no tutorial Explorar membros da interface abstrata estática ou na postagem no blog sobre Versão prévia do recurso no .NET 6 – matemática genérica.

A matemática genérica criou outros requisitos na linguagem.

- *Operador de deslocamento para a direita sem sinal*: antes do C# 11, para forçar um deslocamento para a direita sem sinal, você precisaria converter qualquer tipo

inteiro com sinal em um tipo sem sinal, executar a mudança e, em seguida, converter o resultado de volta para um tipo com sinal. A partir do C# 11, você pode usar o `>>>`, o *operador de deslocamento sem sinal*.

- *Requisitos de operador de deslocamento para a direita sem sinal*: o C# 11 remove o requisito de que o segundo operando deve ser um `int` ou implicitamente conversível para `int`. Essa alteração permite que os tipos que implementam interfaces matemáticas genéricas sejam usados nesses locais.
- *operadores marcados e desmarcados definidos pelo usuário*: os desenvolvedores agora podem definir operadores aritméticos `checked` e `unchecked`. O compilador gera chamadas para a variante correta com base no contexto atual. Você pode ler mais sobre operadores `checked` no artigo sobre operadores aritméticos.

Numérico `IntPtr` e `UIntPtr`

Os tipos `nint` e `nuint` agora têm como alias System.IntPtr e System.UIntPtr, respectivamente.

Novas linhas em interpolações de cadeia de caracteres

O texto dentro dos caracteres `{` e `}` para uma interpolação de cadeia de caracteres agora pode abranger várias linhas. O texto entre os marcadores `{` e `}` é analisado como C#. Qualquer C# legal, inclusive novas linhas, é permitido. Esse recurso facilita a leitura de interpolações de cadeia de caracteres que usam expressões C# mais longas, como expressões `switch` de padrões correspondentes ou consultas LINQ.

Você pode saber mais sobre o recurso de novas linhas no artigo de interpolações de cadeia de caracteres na referência de linguagem.

Padrões de lista

Padrões de lista estendem a correspondência de padrões para corresponder a sequências de elementos em uma lista ou em uma matriz. Por exemplo, `sequence is [1, 2, 3]` é `true` quando a `sequence` é um matriz ou uma lista de três inteiros (1, 2 e 3). Você pode fazer a correspondência de elementos usando qualquer padrão, inclusive uma constante, tipo, propriedade e padrões relacionais. O padrão de descarte (`_`) corresponde a qualquer elemento único e o novo *padrão de intervalo* (`..`) corresponde a qualquer sequência de zero ou mais elementos.

Você pode saber mais detalhes sobre padrões de lista no artigo de padrões correspondentes na referência de linguagem.

Conversão aprimorada do grupo de métodos para delegado

O padrão C# em conversões de grupo de métodos agora inclui o seguinte item:

> - A conversão é permitida (mas não é necessária) para usar uma instância delegada existente que já contenha essas referências.

As versões anteriores do padrão proibiam o compilador de reutilizar o objeto delegado criado para uma conversão de grupo de métodos. O compilador C# 11 armazena em cache o objeto delegado criado de uma conversão de grupo de métodos e reutiliza esse único objeto delegado. Esse recurso foi disponibilizado pela primeira vez no Visual Studio 2022 versão 17.2 como uma versão prévia do recurso e no .NET 7 Versão Prévia 2.

Literais de cadeia de caracteres bruta

Literais de cadeia de caracteres bruta são um novo formato para literais de cadeia de caracteres. Literais de cadeia de caracteres bruta podem conter texto arbitrário, incluindo espaços em branco, novas linhas, aspas inseridas e outros caracteres especiais sem a necessidade de sequências de escape. Um literal de cadeia de caracteres bruta começa com pelo menos três caracteres de aspas duplas (""""). Ele termina com o mesmo número de caracteres de aspas duplas. Normalmente, um literal de cadeia de caracteres bruta usa três aspas duplas em uma única linha para iniciar a cadeia de caracteres e três aspas duplas em uma linha separada para terminar a cadeia de caracteres. As novas linhas que seguem as aspas de abertura e precedem as aspas de fechamento não estão incluídas no conteúdo final:

C#

```
string longMessage = """
    This is a long message.
    It has several lines.
        Some are indented
                more than others.
    Some should start at the first column.
    Some have "quoted text" in them.
    """;
```

Qualquer espaço em branco à esquerda das aspas duplas de fechamento será removido do literal da cadeia de caracteres. Os literais da cadeia de caracteres bruta podem ser combinados com a interpolação de cadeia de caracteres para incluir chaves no texto de saída. Vários caracteres `$` indicam quantas chaves consecutivas iniciam e terminam a interpolação:

```
C#

var location = $$"""
   You are at {{{Longitude}}, {{Latitude}}}
   """;
```

O exemplo anterior especifica que duas chaves iniciam e terminam uma interpolação. A terceira chave de abertura e de fechamento repetidas são incluídas na cadeia de caracteres de saída.

Você pode saber mais sobre literais de cadeia de caracteres bruta no artigo sobre cadeias de caracteres no guia de programação e nos artigos de referência de linguagem sobre literais de cadeia de caracteres e cadeias de caracteres interpoladas.

Structs de padrão automático

O compilador C# 11 garante que todos os campos de um tipo `struct` sejam inicializados para seu valor padrão como parte da execução de um construtor. Essa alteração significa que qualquer campo ou propriedade automática não inicializados por um construtor são inicializados automaticamente pelo compilador. Os structs em que o construtor não atribui definitivamente todos os campos agora são compilados, e todos os campos não inicializados explicitamente são definidos para o valor padrão. Você pode ler mais sobre como essa alteração afeta a inicialização de structs no artigo sobre structs.

Correspondência de padrão `Span<char>` e `ReadOnlySpan<char>` em uma constante `string`

Você conseguiu testar se um `string` tinha um valor constante específico usando padrões correspondentes para várias versões. Agora você pode usar a mesma lógica de padrões correspondentes com variáveis que são `Span<char>` ou `ReadOnlySpan<char>`.

Escopo nameof estendido

Nomes de parâmetros de tipo e nomes de parâmetro agora estão no escopo quando usados em uma expressão `nameof` em uma declaração de atributo nesse método. Essa funcionalidade significa que você pode usar o operador `nameof` para especificar o nome de um parâmetro de método em um atributo na declaração de método ou parâmetro. Esse recurso costuma ser útil para adicionar atributos para análise anulável.

Cadeia de caracteres UTF-8 literais

Você pode especificar o sufixo `u8` em um literal de cadeia de caracteres para especificar a codificação de caracteres UTF-8. Se o aplicativo precisar de cadeias de caracteres UTF-8, para constantes de cadeia de caracteres HTTP ou protocolos de texto semelhantes, você pode usar esse recurso para simplificar a criação de cadeias de caracteres UTF-8.

Você pode saber mais sobre literais de cadeia de caracteres UTF-8 na seção literal de cadeia de caracteres do artigo sobre tipos de referência internos.

Membros necessários

Você pode adicionar o modificador required a propriedades e campos para impor construtores e chamadores para inicializar esses valores. O System.Diagnostics.CodeAnalysis.SetsRequiredMembersAttribute pode ser adicionado aos construtores para informar ao compilador que um construtor inicializa *todos* os membros necessários.

Para obter mais informações sobre os membros necessários, consulte a seção somente init do artigo de propriedades.

Campos `ref` e variáveis `ref scoped`

Você pode declarar campos `ref` dentro de uma classe ref struct. Isso dá suporte a tipos como System.Span<T> sem atributos especiais ou tipos internos ocultos.

Você pode adicionar o modificador scoped a qualquer declaração `ref`. Isso limita o escopo para o qual a referência pode escapar.

Tipos de locais de arquivo

A partir do C# 11, você pode usar o modificador de acesso `file` para criar um tipo cuja visibilidade está no escopo do arquivo de origem no qual ele é declarado. Esse recurso ajuda os autores do gerador de origem a evitar colisões de nomenclatura. Você pode

saber mais sobre esse recurso no artigo sobre tipos com escopo de arquivo na referência de linguagem.

Confira também

- Novidades do .NET 7

Colaborar conosco no GitHub

A fonte deste conteúdo pode ser encontrada no GitHub, onde você também pode criar e revisar problemas e solicitações de pull. Para obter mais informações, confira o nosso guia para colaboradores.

Comentários do .NET

O .NET é um projeto código aberto. Selecione um link para fornecer comentários:

Abrir um problema de documentação

Fornecer comentários sobre o produto

Saiba mais sobre as mudanças disruptivas no compilador C#

Artigo • 22/05/2025

Para encontrar alterações significativas desde a versão do C# 10, consulte Alterações significativas no Roslyn após o .NET 6.0.100 até o .NET 7.0.100.

A equipe roslyn mantém uma lista de alterações significativas nos compiladores C# e Visual Basic. Você pode encontrar informações sobre essas alterações nesses links no repositório GitHub:

- Alterações significativas no Roslyn no C# 10.0/.NET 6
- Alterações significativas no Roslyn após o .NET 5
- Alterações significativas no VS2019 versão 16.8 introduzidas com .NET 5 e C# 9.0
- Alterações significativas no VS2019 Atualização 1 e posterior em comparação com o VS2019
- Alterações significativas desde VS2017 (C# 7)
- Alterações significativas no Roslyn 3.0 (VS2019) de Roslyn 2.* (VS2017)
- Alterações significativas no Roslyn 2.0 (VS2017) do Roslyn 1.* (VS2015) e do compilador C# nativo (VS2013 e anterior).
- Alterações significativas no Roslyn 1.0 (VS2015) do compilador C# nativo (VS2013 e anterior).
- Alteração de versão do Unicode no C# 6

O histórico da linguagem C#

Artigo • 13/01/2025

Este artigo fornece um histórico de cada versão principal da linguagem C#. A equipe C# continua a inovar e a adicionar novos recursos. Os status detalhados do recurso de linguagem, incluindo os recursos considerados para as versões futuras, podem ser encontrados no repositório dotnet/roslyn no GitHub. Para descobrir quando um recurso específico foi adicionado ao idioma, consulte o arquivo de histórico de versões em C# no repositório dotnet/csharplang no GitHub.

> **Importante**
>
> A linguagem C# depende de tipos e métodos nos quais a especificação C# é definida como uma *biblioteca padrão* para alguns dos recursos. A plataforma .NET fornece esses tipos e métodos em alguns pacotes. Um exemplo é o processamento de exceção. Cada instrução ou expressão `throw` é verificada para garantir que o objeto que está sendo gerado é derivado de **Exception**. Da mesma forma, cada `catch` é verificado para garantir que o tipo que está sendo capturado é derivado de **Exception**. Cada versão pode adicionar novos requisitos. Para usar os recursos de linguagem mais recentes em ambientes mais antigos, talvez seja necessário instalar bibliotecas específicas. Essas dependências estão documentadas na página de cada versão específica. Saiba mais sobre as **relações entre linguagem e biblioteca** para obter informações sobre essa dependência.

C# versão 13

lançado em novembro de 2024

O C# 13 inclui os seguintes novos recursos:

- Coleções de `params`: o modificador `params` não está limitado a tipos de array. Agora, você pode usar `params` com qualquer tipo de coleção reconhecido, incluindo `Span<T>`, assim como tipos de interface.
- Novo tipo e semântica de `lock`: se um System.Threading.Lockfor o destino de uma instrução `lock`, o compilador gerará o código para usar o método Lock.EnterScope() para inserir um escopo exclusivo. O `ref struct` retornado dá suporte ao padrão `Dispose()` para sair do escopo exclusivo.
- Nova sequência de escape - `\e`: você pode usar `\e` como uma sequência de escape de caractere literal para o caractere `ESCAPE`, Unicode `U+001B`.

- Pequenas otimizações na resolução de sobrecarga envolvendo grupos de métodos.
- Acesso implícito de indexador em inicializadores de objeto: o operador de índice implícito "do final", `^`, agora é permitido em uma expressão de inicializador de objeto.
- Você pode usar locais `ref` e contextos `unsafe` em iteradores e métodos assíncronos.
- Você pode usar os tipos `ref struct` para implementar interfaces.
- Você pode permitir tipos `ref struct` como argumentos para parâmetros de tipo em tipos genéricos.
- Propriedades parciais e indexadores agora são permitidos em tipos de `partial`.
- A prioridade de resolução de sobrecarga permite que os autores da biblioteca designem uma sobrecarga como melhor do que as outras.

E a palavra-chave contextual `field` para acessar o campo de backup gerado pelo compilador em uma propriedade implementada automaticamente foi lançada como um recurso de visualização.

C# versão 12

Lançado em novembro de 2023

Os seguintes recursos foram adicionados no C# 12:

- Construtores primários – você pode criar construtores primários em qualquer tipo `class` ou `struct`.
- Expressões de coleção – uma nova sintaxe para especificar expressões de coleção, incluindo o elemento de distribuição, (`..e`), para expandir qualquer coleção.
- Matrizes embutidas – permitem que um desenvolvedor crie uma matriz de tamanho fixo em um tipo `struct`.
- Parâmetros opcionais em expressões lambda – você pode definir valores padrão para parâmetros em expressões lambda.
- ref readonly parâmetros - `ref readonly` – permitem mais clareza para APIs que possam estar usando parâmetros `ref` ou parâmetros `in`.
- Alias de qualquer tipo – Você pode usar a diretiva de alias `using` para alias de qualquer tipo, não apenas tipos nomeados.
- Atributo experimental – indica um recurso experimental.

E Interceptores – foi lançado como *Versão prévia do recurso*.

No geral, o C# 12 fornece novos recursos que tornam você mais produtivo ao escrevercódigo C#. A sintaxe que você já conhecia está disponível em mais locais. Outra sintaxe permite consistência para conceitos relacionados.

C# versão 11

Lançado em novembro de 2022

Os seguintes recursos foram adicionados em C# 11:

- Literais brutos de cadeia de caracteres
- Suporte matemático genérico
- Atributos genéricos
- Cadeia de caracteres UTF-8 literais
- Linhas novas em expressões de interpolação de cadeia de caracteres
- Padrões de lista
- Tipos de locais de arquivos
- Membros necessários
- Structs de padrão automático
- Correspondência de padrão Span<char> em uma constante string
- Escopo estendido nameof
- IntPtr numérico
- refCampos e scoped ref
- Conversão aprimorada do grupo de métodos para delegado
- Ciclo de aviso 7

O C# 11 apresenta *matemática genérica* e vários recursos que dão suporte a essa meta. Você pode escrever algoritmos numéricos uma vez para todos os tipos de número. Há mais recursos para facilitar o trabalho com os tipos de `struct`, como membros necessários e structs de padrão automático. O trabalho com cadeias de caracteres fica mais fácil com literais de cadeia de caracteres brutas, nova linha em interpolações de cadeia de caracteres e literais de cadeia de caracteres UTF-8. Recursos como os tipos de locais de arquivo permitem que os geradores de origem sejam mais simples. Por fim, os padrões de lista adicionam mais suporte para correspondência de padrões.

C# versão 10

Lançado em novembro de 2021

O C# 10 adiciona os seguintes recursos e aprimoramentos à linguagem C#:

- Estruturas de registro

- Aprimoramentos de tipos de estrutura
- Manipuladores de cadeia de caracteres interpolada
- global usingDiretivas
- Declaração de namespace com escopo de arquivo
- Padrões de propriedade estendida
- As expressões Lambda podem ter um tipo natural, em que o compilador pode inferir um tipo delegado da expressão lambda ou do grupo de métodos.
- As expressões lambda poderão declarar um tipo de retorno quando o compilador não puder inferi-lo.
- Atributos podem ser aplicados a expressões lambda.
- No C# 10, `const` as cadeias de caracteres podem ser inicializadas usando a interpolação de cadeia de caracteres se todos os espaços reservados forem cadeias de caracteres constantes.
- No C# 10, você pode adicionar o `sealed` modificador ao substituir `ToString` em um tipo de registro.
- Os avisos para atribuição definitiva e análise de estado nulo são mais precisos.
- Permitir atribuição e declaração na mesma desconstrução.
- Permissão do atributo AsyncMethodBuilder em métodos
- Atributo CallerArgumentExpression
- O C# 10 dá suporte a um novo formato para o pragma `#line`.

Mais recursos estavam disponíveis no modo de *visualização*. Para usar esses recursos, você deve definir <LangVersion> como Preview em seu projeto:

- Atributos genéricos, posteriormente neste artigo.
- membros abstratos estáticos em interfaces.

C# 10 continua trabalhando em temas de remoção de cerimônia, separação de dados de algoritmos e aprimoramento do desempenho para o Runtime do .NET.

Muitos dos recursos significam que você digitará menos código para expressar os mesmos conceitos. *Structs de registro* sintetizam muitos dos mesmos métodos que as *classes de registro*. Structs e tipos anônimos dão suporte *a expressões*. *Diretivas de uso global* e *declarações de namespace com escopo de arquivo* significam expressar dependências e organização de namespace com mais clareza. *Melhorias lambda* facilitam a declaração de expressões lambda onde são usadas. Novos padrões de propriedade e melhorias de desconstrução criam um código mais conciso.

Os novos manipuladores de cadeia de caracteres interpolados e o comportamento `AsyncMethodBuilder` podem melhorar o desempenho. Esses recursos de linguagem foram aproveitados no .NET Runtime para obter aprimoramentos de desempenho no .NET 6.

C# 10 também marca uma mudança para a cadência anual de versões do .NET. Como nem todos os recursos podem ser concluídos em um período anual, você pode experimentar alguns recursos de "versão prévia" no C# 10. Os *atributos genéricos* e os *membros abstratos estáticos em interfaces* podem ser usados, mas eles são versões prévias do recurso e podem ser alterados antes da versão final.

C# versão 9

Lançado em novembro de 2020

C# 9 foi lançada com o .NET 5. É a versão de linguagem padrão para qualquer assembly direcionado à versão do .NET 5. Ela contém os seguintes recursos novos e aprimorados:

- Registros
- Setters somente init
- Instruções de nível superior
- Aprimoramentos nos padrões correspondentes: padrões relacionais e padrões lógicos
- Desempenho e interoperabilidade
  - Inteiros de tamanho nativo
  - Ponteiros de função
  - Suprimir a emissão do sinalizador localsinit
  - Inicializadores de módulo
  - Novos recursos para métodos parciais
- Ajustar e concluir recursos
  - Expressões com tipo de destino new
  - static funções anônimas
  - Expressão condicional com tipo de destino
  - Tipos de retorno covariantes
  - Suporte à extensão GetEnumerator para loops foreach
  - Parâmetros discard de lambda
  - Atributos em funções locais

C# 9 continua três dos temas de versões anteriores: remoção da cerimônia, separação dos dados de algoritmos e fornecimento de mais padrões em mais lugares.

Instruções de nível superior significam que seu programa principal é mais simples de ler. Há menos necessidade de cerimônia: um namespace, uma classe `Program` e `static void Main()` são todos desnecessários.

A introdução de records fornece uma sintaxe concisa para tipos de referência que seguem semântica de valor para manter a igualdade. Você usará esses tipos para definir

contêineres de dados que normalmente definem o comportamento mínimo. Os setters somente init fornecem o recurso de mutação não destrutiva (expressões `with`) nos registros. C# 9 também adiciona tipos de retorno covariantes para que os registros derivados possam substituir métodos virtuais e retornar um tipo derivado do tipo de retorno do método base.

Os recursos de padrões correspondentes foram expandidos de várias maneiras. Agora, os tipos numéricos dão suporte aos *padrões de intervalo*. Os padrões podem ser combinados usando padrões `and`, `or` e `not`. É possível adicionar parênteses para esclarecer padrões mais complexos:

O C# 9 inclui novas melhorias de correspondência de padrões:

- ***Padrões de tipo*** correspondem a um objeto correspondente a um tipo específico
- ***Padrões entre parênteses*** impõem ou enfatizam a precedência de combinações de padrões
- ***Padrões conjuntivos*** `and` exigem que ambos os padrões correspondam
- ***Padrões conjuntivos*** `or` exigem que ambos os padrões correspondam
- ***Padrões conjuntivos*** `not` exigem que ambos os padrões correspondam
- ***Os padrões relacionais*** exigem que a entrada seja menor que, maior que, menor ou igual ou maior que ou igual a uma determinada constante

Esses padrões enriquecem a sintaxe para padrões. Um dos usos mais comuns é uma nova sintaxe para uma verificação nula:

C#

```
if (e is not null)
{
    // ...
}
```

Qualquer um desses padrões pode ser usado em qualquer contexto em que os padrões são permitidos: expressões de padrão `is`, expressões `switch`, padrões aninhados e o padrão de uma instrução `switch` do rótulo `case`.

Outro conjunto de recursos dá suporte à computação de alto desempenho em C#:

- Os tipos `nint` e `nuint` modelam os tipos inteiros de tamanho nativo na CPU de destino.
- Os ponteiros de função fornecem funcionalidade semelhante a delegado, evitando as alocações necessárias para criar um objeto delegado.
- A instrução `localsinit` pode ser omitida para salvar instruções.

Desempenho e interoperabilidade

Outro conjunto de aprimoramentos dá suporte a cenários em que os *geradores de código* adicionam funcionalidade:

- Inicializadores de módulo são métodos que o runtime chama quando um assembly é carregado.
- Métodos parciais dão suporte a novos modificadores acessíveis e tipos de retorno não nulos. Nesses casos, uma implementação deve ser fornecida.

Ajustar e concluir recursos

C# 9 adiciona muitos outros pequenos recursos que melhoram a produtividade do desenvolvedor, escrevendo e lendo código:

- Expressões `new` de tipo de destino
- Funções anônimas de `static`
- Expressão condicional com tipo de destino
- Suporte a `GetEnumerator()` de extensão para loops `foreach`
- Expressões Lambda podem declarar parâmetros de descarte
- Atributos podem ser aplicados a funções locais

A versão C# 9 continua o trabalho para manter C# uma linguagem de programação moderna e de uso geral. Os recursos continuam a dar suporte a cargas de trabalho e tipos de aplicativo modernos.

C# versão 8.0

Lançado em setembro de 2019

C# 8.0 é a primeira grande versão em C# que tem como destino especificamente o .NET Core. Alguns recursos dependem de novos recursos de Common Language Runtime (CLR), outros, de tipos de biblioteca adicionados somente no .NET Core. O C# 8.0 adiciona os seguintes recursos e aprimoramentos à linguagem C#:

- Membros somente leitura
- Métodos de interface padrão
- Aprimoramentos de correspondência de padrões:
  - Expressões Switch
  - Padrões da propriedade
  - Padrões de tupla
  - Padrões posicionais

- Declarações using
- Funções locais estáticas
- Estruturas ref descartáveis
- Tipos de referência anuláveis
- Fluxos assíncronos
- Índices e intervalos
- Atribuição de coalescência nula
- Tipos construídos não gerenciados
- Stackalloc em expressões aninhadas
- Aprimoramento de cadeias de caracteres verbatim interpoladas

Os membros de interface padrão exigem aprimoramentos na CLR. Esses recursos foram adicionados na CLR para .NET Core 3.0. Intervalos e índices, além de fluxos assíncronos, exigem novos tipos nas bibliotecas do .NET Core 3.0. Tipos de referência anuláveis, implementados no compilador, são muito mais úteis quando bibliotecas são anotadas para fornecer informações semânticas sobre o estado nulo de argumentos e valores retornados. Essas anotações estão sendo adicionadas nas bibliotecas do .NET Core.

C# versão 7.3

Lançado em maio de 2018

Há dois temas principais para a versão C# 7.3. Um tema fornece recursos que permitem que o código seguro tenha o mesmo desempenho que o código não seguro. O segundo tema fornece melhorias incrementais aos recursos existentes. Novas opções do compilador também foram adicionadas a essa versão.

Os novos recursos a seguir são compatíveis com o tema de melhor desempenho para código seguro:

- Você pode acessar campos fixos sem fixação.
- Você pode reatribuir variáveis locais `ref`.
- Você pode usar inicializadores em matrizes `stackalloc`.
- Você pode usar instruções `fixed` com qualquer tipo compatível com um padrão.
- Você pode usar mais restrições genéricas.

Os seguintes recursos e aprimoramentos foram feitos nos recursos existentes:

- Você pode testar `==` e `!=` com tipos de tupla.
- Você pode usar variáveis de expressão em mais locais.
- Você pode anexar atributos ao campo de suporte de propriedades implementadas automaticamente.

- A resolução de métodos quando os argumentos se diferenciam por `in` foi aprimorada.
- A resolução de sobrecarga agora tem menos casos ambíguos.

As novas opções do compilador são:

- -publicsign para habilitar a assinatura de Software de código aberto (OSS) de assemblies.
- -pathmap para fornecer um mapeamento para diretórios de origem.

C# versão 7.2

Lançado em novembro de 2017

C# 7.2 adicionou vários recursos de linguagem menores:

- Inicializadores em matrizes `stackalloc`.
- Uso de instruções `fixed` com qualquer tipo compatível com um padrão.
- Acesso a campos fixos sem fixação.
- Reatribuição de variáveis locais `ref`.
- Declaração de tipos `readonly struct`, para indicar que uma struct é imutável e deve ser passada como um parâmetro `in` para seus métodos de membro.
- Adição do modificador `in` em parâmetros, para especificar que um argumento seja passado por referência, mas não modificado pelo método chamado.
- Uso do modificador `ref readonly` nos retornos de método, para indicar que um método retorna seu valor por referência, mas não permite gravações nesse objeto.
- Declaração de tipos `ref struct`, para indicar que um tipo de struct acessa a memória gerenciada diretamente e deve sempre ser alocado por pilha.
- Use restrições mais genéricas.
- Argumentos nomeados que não estejam à direita::
  - Os argumentos nomeados podem ser seguidos por argumentos posicionais.
- Sublinhados à esquerda em literais numéricos:
  - Agora os literais numéricos podem ter sublinhados à esquerda, antes dos dígitos impressos.
- private protectedmodificadores de acesso:
  - O modificador de acesso `private protected` permite o acesso a classes derivadas no mesmo assembly.
- Expressões `ref` condicionais:
  - O resultado de uma expressão condicional (`?:`) agora já pode ser uma referência.

C# versão 7.1

Lançado em agosto de 2017

C# começou a lançar *versões de ponto* com C# 7.1. Essa versão adiciona a o elemento de configuração de seleção de versão da linguagem, três novos recursos de linguagem e um novo comportamento do compilador.

Os novos recursos de linguagem nesta versão são:

- asyncMain método
  - O ponto de entrada para um aplicativo pode ter o modificador `async`.
- default Expressões literais
  - Use expressões literais padrão em expressões de valor padrão quando o tipo de destino pode ser inferido.
- Nomes de elementos de tupla inferidos
  - Em muitos casos, os nomes dos elementos de tupla podem ser inferidos com base na inicialização da tupla.
- Restrições em parâmetros de tipo genérico
  - Você pode usar expressões de correspondência de padrão em variáveis cujo tipo é um parâmetro de tipo genérico.

Por fim, o compilador traz duas opções -refout e -refonly, que controlam a geração de assembly de referência.

C# versão 7.0

Lançado em março de 2017

A versão 7.0 de C# foi lançada com o Visual Studio 2017. Esta versão tem algumas coisas interessantes e evolutivas na mesma direção que o C# 6.0. Aqui estão alguns dos novos recursos:

- Variáveis out
- Tuplas e desconstrução
- Correspondência de padrões
- Funções locais
- Membros aptos para expressão expandidos
- Ref locals
- Retornos de referências

Outros recursos incluíam:

- Descartes
- Literais binários e os separadores de dígito
- Expressões throw

Todas essas funcionalidades oferecem novos recursos para desenvolvedores e a oportunidade de escrever um código mais limpo do que nunca. Um ponto alto é a condensação da declaração de variáveis a serem usadas com a palavra-chave `out` e a permissão de vários valores retornados por meio de tupla. Agora o .NET Core tem qualquer sistema operacional como destino e tem a visão firme na nuvem e na portabilidade. Essas novas funcionalidades certamente ocupam a mente e o tempo dos designers da linguagem, além de levarem a novos recursos.

C# versão 6.0

Lançado em julho de 2015

A versão 6.0, lançada com o Visual Studio 2015, trouxe muitos recursos menores que tornaram a programação em C# mais produtiva. Eis algumas delas:

- Importações estáticas
- Filtros de exceção
- Inicializadores de propriedade automática
- Membros aptos para expressão
- Propagador nulo
- Interpolação de cadeia de caracteres
- Operador nameof

Outros novos recursos incluem:

- Inicializadores de índice
- Await em blocos catch/finally
- Valores padrão para propriedades somente getter

Mas, se você os observar em conjunto, verá um padrão interessante. Nesta versão, o C# começou a eliminar o clichê de linguagem para tornar o código mais conciso e legível. Portanto, para os fãs de código simples e conciso, essa versão da linguagem foi um grande benefício.

Fizeram ainda outra coisa com esta versão, embora não seja um recurso de linguagem tradicional em si. Lançaram Roslyn, o compilador como um serviço. Agora o compilador de C# é escrito em C#, e você pode usar o compilador como parte de seus esforços de programação.

C# versão 5.0

Lançado em agosto de 2012

A versão 5.0 de C#, lançada com o Visual Studio 2012, era uma versão focada da linguagem. Quase todo o esforço para essa versão foi dedicado a outro conceito inovador de linguagem: os modelos `async` e `await` para programação assíncrona. Aqui está a lista dos recursos principais:

- Membros assíncronos
- Atributos de informações do chamador
- Code Project: Caller Info Attributes in C# 5.0 (Code Project: Atributos de informações do chamador em C# 5.0)

O atributo de informações do chamador permite facilmente recuperar informações sobre o contexto no qual você está executando sem recorrer a uma infinidade de código de reflexão clichê. Ele tem muitos usos em diagnóstico e tarefas de registro em log.

Mas `async` e `await` são as verdadeiras estrelas dessa versão. Quando esses recursos foram lançados em 2012, o C# virou o jogo novamente, implantando a assincronia na linguagem como uma participante da maior importância.

C# versão 4.0

Lançado em abril de 2010

O C# versão 4.0, lançado com o Visual Studio 2010, introduziu alguns novos recursos interessantes:

- Associação dinâmica
- Argumentos opcionais/nomeados
- Genérico covariante e contravariante
- Tipos de interoperabilidade inseridos

Tipos de interoperabilidade inseridos facilitaram os problemas de implantação da criação de assemblies de interoperabilidade COM para seu aplicativo. A contravariância e a covariância genérica oferecem maior capacidade para usar genéricos, mas eles são um tanto acadêmicos e provavelmente mais apreciados por autores de estruturas e bibliotecas. Os parâmetros nomeados e opcionais permitem eliminar várias sobrecargas de método e oferecem conveniência. Mas nenhum desses recursos é exatamente uma alteração de paradigma.

O recurso principal foi a introdução da palavra-chave `dynamic`. A palavra-chave `dynamic` introduziu na versão 4.0 do C# a capacidade de substituir o compilador na tipagem em tempo de compilação. Com o uso da palavra-chave dinâmica, você pode criar constructos semelhantes a linguagens dinamicamente tipadas, como JavaScript. Você pode criar um `dynamic x = "a string"` e, em seguida, adicionar seis a ela, deixando que o runtime decida o que acontece em seguida.

Associação dinâmica tem potencial de erros, mas também grande eficiência na linguagem.

C# versão 3.0

Lançado em novembro de 2007

O C# versão 3.0 chegou no final de 2007, juntamente com o Visual Studio 2008, porém o pacote completo de recursos de linguagem veio, na verdade, com o C# versão 3.5. Esta versão foi o marco de uma alteração significativa no crescimento do C#. Ela estabeleceu o C# como uma linguagem de programação realmente formidável. Vamos dar uma olhada em alguns recursos importantes nesta versão:

- Propriedades autoimplementadas
- Tipos anônimos
- Expressões de consulta
- Expressões lambda
- Árvores de expressão
- Métodos de extensão
- Variáveis locais de tipo implícito
- Métodos parciais
- Inicializadores de objeto e de coleção

Numa retrospectiva, muitos desses recursos parecerem inevitáveis e inseparáveis. Todos eles se encaixam estrategicamente. O melhor recurso dessa versão do C# era a expressão de consulta, também conhecida como Consulta Integrada à Linguagem (LINQ).

Uma exibição mais detalhada analisa árvores de expressão, expressões lambda e tipos anônimos como a base na qual o LINQ é construído. Mas, de uma forma ou de outra, o C# 3.0 apresentou um conceito revolucionário. O C# 3.0 começou a definir as bases para transformar o C# em uma linguagem híbrida orientada a objeto e funcional.

Especificamente, agora você pode escrever consultas declarativas no estilo SQL para executar operações em coleções, entre outras coisas. Em vez de escrever um loop `for`

para calcular a média de uma lista de inteiros, agora você pode fazer isso simplesmente como `list.Average()`. A combinação de expressões de consulta e métodos de extensão tornou uma lista de inteiros muito mais inteligente.

C# versão 2.0

Lançado em novembro de 2005

Vamos dar uma olhada em alguns recursos principais do C# 2.0, lançado em 2005, junto com o Visual Studio 2005:

- Genéricos
- Tipos parciais
- Métodos anônimos
- Tipos de valor anuláveis
- Iteradores
- Covariância e contravariância

Outros recursos do C# 2.0 adicionaram funcionalidades a recursos existentes:

- Acessibilidade separada getter/setter
- Conversões de grupo de método (delegados)
- Classes estáticas
- Inferência de delegado

Embora C# possa ter começado como uma linguagem orientada a objeto (OO) genérica, a versão 2.0 do C# tratou de mudar isso rapidamente. Com genéricos, tipos e métodos podem operar em um tipo arbitrário enquanto ainda mantêm a segurança de tipos. Por exemplo, ter um List<T> permite que você tenha `List<string>` ou `List<int>` e execute operações fortemente tipadas nessas cadeias de caracteres ou inteiros enquanto itera neles. Usar genéricos é melhor do que criar um `ListInt` que deriva de `ArrayList` ou converter de `Object` para cada operação.

A versão 2.0 do C# trouxe iteradores. Em resumo, o iteradores permitem que você examine todos os itens em um `List` (ou outros tipos Enumeráveis) com um loop `foreach`. Ter iteradores como uma parte importante da linguagem aprimorou drasticamente a legibilidade da linguagem e a capacidade das pessoas de raciocinar a respeito do código.

C# versão 1.2

Lançado em abril de 2003

C# versão 1.2 fornecida com o Visual Studio .NET 2003. Ele continha algumas pequenas melhorias na linguagem. Muito notável é que, a partir desta versão, o código gerado em um loop `foreach` chamou Dispose, em um IEnumerator, quando o IEnumerator implementou IDisposable.

C# versão 1.0

Lançado em janeiro de 2002

Ao olhar para trás, a versão 1.0 de C#, lançada com o Visual Studio .NET 2002, ficou muito parecida com o Java. Como parte de suas metas de design declaradas para ECMA, ela procurou ser uma "linguagem simples, moderna e orientada a objetos para uso geral". Na época, parecer com Java significava que havia atingido essas metas de design iniciais.

Mas agora, se examinar novamente a C# 1.0, você poderá se sentir um pouco confuso. Carecia das funcionalidades assíncronas internas e algumas das funcionalidades relacionadas a genéricos que você nem valoriza. Na verdade, ela não tinha nada relacionado a genéricos. E a LINQ? Ainda não estava disponível. Essas adições levariam alguns anos para sair.

A versão 1.0 do C# parecia ter poucos recursos, em comparação com os dias de hoje. Você teria que escrever código um tanto detalhado. Mas, ainda assim, você poderia iniciar em algum lugar. A versão 1.0 do C# era uma alternativa viável ao Java na plataforma Windows.

Os principais recursos do C# 1.0 incluíam:

- Classes
- Estruturas
- Interfaces
- Eventos
- Propriedades
- Representantes
- Operadores e expressões
- Instruções
- Atributos

Artigo publicado originalmente no blog NDepend, cortesia de Erik Dietrich e Patrick Smacchia.

Relações entre recursos de linguagem e tipos de biblioteca

19/06/2025

A definição de linguagem C# requer uma biblioteca padrão para ter determinados tipos e determinados membros acessíveis nesses tipos. O compilador gera código que usa esses tipos e membros necessários para muitos recursos de linguagem diferentes. Por esse motivo, as versões do C# têm suporte apenas para a versão correspondente do .NET e mais recentes. Isso garante o comportamento correto em tempo de execução e a disponibilidade de todos os tipos e membros necessários.

Essa dependência da funcionalidade de biblioteca padrão faz parte da linguagem C# desde sua primeira versão. Nessa versão, exemplos incluíam:

- Exception – usado para todas as exceções geradas pelo compilador.
- String - sinônimo de `string`.
- Int32 - sinônimo de `int`.

Essa primeira versão foi simples: o compilador e a biblioteca padrão foram enviados juntos, e havia apenas uma versão de cada uma.

As versões subsequentes do C# ocasionalmente adicionam novos tipos ou membros às dependências. Os exemplos incluem:INotifyCompletion,CallerFilePathAttribute e CallerMemberNameAttribute. O C# 7.0 adicionou uma dependência em ValueTuple para implementar o recurso de linguagem de tuplas. O C# 8 requer System.Index e System.Range para intervalos e índices, entre outros recursos. Cada nova versão pode adicionar requisitos adicionais.

A equipe de design de linguagem trabalha para minimizar a área de superfície dos tipos e membros necessários em uma biblioteca padrão em conformidade. Essa meta é equilibrada em relação a um design limpo em que novos recursos de biblioteca são incorporados perfeitamente ao idioma. Haverá novos recursos em versões futuras do C# que exigem novos tipos e membros em uma biblioteca padrão. As ferramentas do compilador C# agora são dissociadas do ciclo de lançamento das bibliotecas do .NET em plataformas com suporte.

Considerações sobre versão e atualização para os desenvolvedores de C#

Artigo • 30/06/2023

A compatibilidade é uma meta importante quando novos recursos são adicionados à linguagem C#. Em quase todos os casos, o código existente pode ser recompilado com uma nova versão do compilador sem nenhum problema. A equipe de runtime do .NET também tem uma meta para garantir a compatibilidade das bibliotecas atualizadas. Em quase todos os casos, quando seu aplicativo é iniciado de um runtime atualizado com bibliotecas atualizadas, o comportamento é exatamente o mesmo que nas versões anteriores.

A versão de linguagem usada para compilar seu aplicativo normalmente corresponde ao TFM (moniker da estrutura de destino) de runtime referenciado em seu projeto. Para obter mais informações sobre como alterar a versão de linguagem padrão, confira o artigo intitulado configurar sua versão de linguagem. Esse comportamento padrão garante a compatibilidade máxima.

Quando *alterações interruptivas* são introduzidas, elas são classificadas como:

- *Alteração interruptiva binária*: uma alteração interruptiva binária causa um comportamento diferente, incluindo possivelmente falha, em seu aplicativo ou biblioteca quando iniciado usando um novo runtime. Você deve recompilar seu aplicativo para incorporar essas alterações. O binário existente não funcionará corretamente.
- *Alteração interruptiva de origem*: uma alteração interruptiva de origem altera o significado do código-fonte. Você precisa fazer edições de código-fonte antes de compilar seu aplicativo com a versão mais recente do idioma. O binário existente será executado corretamente com o host e o runtime mais recentes. Observe que, para a sintaxe da linguagem, uma *alteração interruptiva de origem* também é uma *alteração comportamental*, conforme definido nas alterações interruptivas do runtime.

Quando uma alteração interruptiva binária afeta seu aplicativo, você precisa recompilar seu aplicativo, mas não precisa editar nenhum código-fonte. Quando uma alteração interruptiva de origem afeta seu aplicativo, o binário existente ainda é executado corretamente em ambientes com o runtime e as bibliotecas atualizados. No entanto, você deve fazer alterações de origem para recompilar com a nova versão do idioma e o

runtime. Se uma alteração for interruptiva de origem e interruptiva binária, você deverá recompilar seu aplicativo com a versão mais recente e fazer atualizações de origem.

Devido à meta de evitar alterações interruptivas da equipe de linguagem C# e da equipe de runtime, atualizar seu aplicativo normalmente é uma questão de atualizar o TFM e recompilar o aplicativo. No entanto, para bibliotecas distribuídas publicamente, você deve avaliar cuidadosamente sua política quanto a TFMs com suporte e versões de linguagem com suporte. Você poderá estar criando uma nova biblioteca com recursos encontrados na versão mais recente, e precisará garantir que os aplicativos criados com versões anteriores do compilador possam usá-la. Ou você pode estar atualizando uma biblioteca existente e muitos dos seus usuários podem não ter versões atualizadas ainda.

Como introduzir alterações interruptivas em suas bibliotecas

Ao adotar novos recursos de linguagem na API pública da biblioteca, você deve avaliar se a adoção do recurso introduz uma alteração interruptiva binária ou de origem para os usuários da biblioteca. Todas as alterações na implementação interna que não aparecem nas interfaces `public` ou `protected` são compatíveis.

> ⓘ **Observação**
>
> Se você usar o **System.Runtime.CompilerServices.InternalsVisibleToAttribute** para habilitar tipos para ver membros internos, os membros internos poderão introduzir alterações interruptivas.

Uma *alteração interruptiva binária* exige que os usuários recompilem o código para usar a nova versão. Por exemplo, considere este método público:

C#

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

Se você adicionar o modificador `in` ao método, essa será uma alteração interruptiva binária:

C#

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

Os usuários devem recompilar qualquer aplicativo que use o método `CalculateSquare` para que a nova biblioteca funcione corretamente.

Uma *alteração interruptiva de origem* exige que os usuários alterem o código antes de recompilarem. Por exemplo, pense neste tipo:

```
C#

public class Person
{
    public string FirstName { get; }
    public string LastName { get; }

    public Person(string firstName, string lastName) => (FirstName,
LastName) = (firstName, lastName);

    // other details omitted
}
```

Em uma versão mais recente, você gostaria de aproveitar os membros sintetizados gerados para tipos `record`. Você faz as seguintes alterações:

```
C#

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

A alteração anterior requer alterações para qualquer tipo derivado de `Person`. Todas essas declarações devem adicionar o modificador `record` às declarações.

Impacto de alterações interruptivas

Ao adicionar uma *alteração interruptiva binária* à biblioteca, você força todos os projetos que usam sua biblioteca a recompilar. No entanto, nenhum código-fonte nesses projetos precisa ser alterado. Como resultado, o impacto da alteração interruptiva é razoavelmente pequeno para cada projeto.

Ao fazer uma *alteração interruptiva de origem* em sua biblioteca, você exige que todos os projetos façam alterações de origem para usar sua nova biblioteca. Se a alteração necessária exigir novos recursos de linguagem, você força esses projetos a atualizar para a mesma versão de idioma e TFM que você está usando agora. Você exigiu mais trabalho para seus usuários e, possivelmente, forçou-os a atualizar também.

O impacto de qualquer alteração interruptiva que você faz depende do número de projetos que têm uma dependência em sua biblioteca. Se sua biblioteca for usada

internamente por alguns aplicativos, você poderá reagir a todas alterações interruptivas em todos os projetos afetados. No entanto, se sua biblioteca for baixada publicamente, você deverá avaliar o impacto potencial e considerar alternativas:

- Você pode adicionar novas APIs que são APIs existentes paralelas.
- Você pode considerar builds paralelos para diferentes TFMs.
- Você pode considerar a multiplataforma.

Tutorial: Explorar construtores primários

Artigo • 28/06/2023

O C# 12 apresenta *construtores primários*, uma sintaxe concisa para declarar construtores cujos parâmetros estão disponíveis em qualquer lugar no corpo do tipo.

Neste tutorial, você aprenderá como:

- ✓ Quando declarar um construtor primário em seu tipo
- ✓ Como chamar construtores primários de outros construtores
- ✓ Como usar parâmetros do construtor primário em membros do tipo
- ✓ Onde os parâmetros do construtor primário são armazenados

Pré-requisitos

Você precisará configurar seu computador para executar o .NET 8 ou posterior, incluindo o compilador C# 12 ou posterior. O compilador C# 12 está disponível a partir do Visual Studio 2022 versão 17.7 ou do SDK do .NET 8.

Construtores primários

Você pode adicionar parâmetros a uma declaração `struct` ou `class` para criar um *construtor primário*. Os parâmetros do construtor primário estão no escopo em toda a definição de classe. É importante exibir os parâmetros do construtor primário como *parâmetros* mesmo que estejam no escopo em toda a definição de classe. Várias regras esclarecem que eles são parâmetros:

1. Pode ser que os parâmetros do construtor primário não sejam armazenados se não forem necessários.
2. Os parâmetros do construtor primário não são membros da classe. Por exemplo, o parâmetro de construtor primário `param` não pode ser acessado como `this.param`.
3. Parâmetros de construtor primário podem ser atribuídos.
4. Os parâmetros do construtor primário não se tornam propriedades, exceto em tipos record.

Essas regras são iguais aos parâmetros de qualquer método, incluindo outras declarações de construtor.

Os usos mais comuns para um parâmetro de construtor primário são:

1. Como um argumento para uma invocação de construtor `base()`.

2. Para inicializar um campo ou propriedade de membro.
3. Referenciando o parâmetro de construtor em um membro de instância.

Todos os outros construtores de uma classe **devem** chamar o construtor primário, direta ou indiretamente, por meio de uma invocação de construtor `this()`. Essa regra garante que os parâmetros do construtor primário sejam atribuídos em qualquer lugar no corpo do tipo.

Inicializar propriedade

O código a seguir inicializa duas propriedades somente leitura computadas a partir dos parâmetros do construtor primário:

```C#
public readonly struct Distance(double dx, double dy)
{
    public readonly double Magnitude = Math.Sqrt(dx * dx + dy * dy);
    public readonly double Direction = Math.Atan2(dy, dx);
}
```

O código anterior demonstra um construtor primário usado para inicializar propriedades somente leitura calculadas. Os inicializadores de campo para `Magnitude` e `Direction` usam os parâmetros do construtor primário. Os parâmetros do construtor primário não são usados em nenhum outro lugar no struct. O struct anterior é como se você tivesse escrito o seguinte código:

```C#
public readonly struct Distance
{
    public readonly double Magnitude { get; }

    public readonly double Direction { get; }

    public Distance(double dx, double dy)
    {
        Magnitude = Math.Sqrt(dx * dx + dy * dy);
        Direction = Math.Atan2(dy, dx);
    }
}
```

O novo recurso facilita o uso de inicializadores de campo quando você precisa de argumentos a fim de inicializar um campo ou propriedade.

Criar estado mutável

Os exemplos anteriores usam os parâmetros do construtor primário para inicializar propriedades somente leitura. Você também pode usar construtores primários quando as propriedades não forem somente leitura. Considere o seguinte código:

C#

```
public struct Distance(double dx, double dy)
{
    public readonly double Magnitude => Math.Sqrt(dx * dx + dy * dy);
    public readonly double Direction => Math.Atan2(dy, dx);

    public void Translate(double deltaX, double deltaY)
    {
        dx += deltaX;
        dy += deltaY;
    }

    public Distance() : this(0,0) { }
}
```

No exemplo anterior, o método `Translate` altera os componentes `dx` e `dy`. Isso requer que as propriedades `Magnitude` e `Direction` sejam computadas quando acessadas. O operador `=>` designa um acessador `get` apto para expressão, enquanto o operador `=` designa um inicializador. Esta versão adiciona um construtor sem parâmetros ao struct. O construtor sem parâmetros deve invocar o construtor primário para que todos os parâmetros do construtor primário sejam inicializados.

No exemplo anterior, as propriedades do construtor primário são acessadas em um método. Portanto, o compilador cria campos ocultos para representar cada parâmetro. O código a seguir mostra aproximadamente o que o compilador gera. Os nomes de campos reais são identificadores MSIL válidos, mas não identificadores C# válidos.

C#

```
public struct Distance
{
    private double __unspeakable_dx;
    private double __unspeakable_dy;

    public readonly double Magnitude => Math.Sqrt(__unspeakable_dx *
__unspeakable_dx + __unspeakable_dy * __unspeakable_dy);
    public readonly double Direction => Math.Atan2(__unspeakable_dy,
__unspeakable_dx);

    public void Translate(double deltaX, double deltaY)
    {
```

```
        __unspeakable_dx += deltaX;
        __unspeakable_dy += deltaY;
    }

    public Distance(double dx, double dy)
    {
        __unspeakable_dx = dx;
        __unspeakable_dy = dy;
    }
    public Distance() : this(0, 0) { }
}
```

É importante entender que o primeiro exemplo não exigia que o compilador criasse um campo para armazenar o valor dos parâmetros do construtor primário. O segundo exemplo usou o parâmetro de construtor primário dentro de um método, portanto, exigia que o compilador criasse armazenamento para eles. O compilador cria o armazenamento para um construtor primário somente quando esse parâmetro é acessado no corpo de um membro do seu tipo. Caso contrário, os parâmetros do construtor primário não serão armazenados no objeto.

Injeção de dependência

Outro uso comum dos construtores primários é especificar parâmetros para injeção de dependência. O código a seguir cria um controlador simples que requer uma interface de serviço para seu uso:

C#

```
public interface IService
{
    Distance GetDistance();
}

public class ExampleController(IService service) : ControllerBase
{
    [HttpGet]
    public ActionResult<Distance> Get()
    {
        return service.GetDistance();
    }
}
```

O construtor primário indica claramente os parâmetros necessários na classe. Use os parâmetros do construtor primário como faria com qualquer outra variável na classe.

Inicializar classe base

Você pode invocar o construtor primário de uma classe base a partir do construtor primário da classe derivada. É a maneira mais fácil de escrever uma classe derivada que deve invocar um construtor primário na classe base. Por exemplo, considere uma hierarquia de classes que representa diferentes tipos de conta como um banco. A classe base seria semelhante ao seguinte código:

C#

```
public class BankAccount(string accountID, string owner)
{
    public string AccountID { get; } = accountID;
    public string Owner { get; } = owner;

    public override string ToString() => $"Account ID: {AccountID}, Owner:
{Owner}";
}
```

Todas as contas bancárias, independentemente do tipo, têm propriedades como o número da conta e um proprietário. No aplicativo concluído, outras funcionalidades comuns seriam adicionadas à classe base.

Muitos tipos exigem uma validação mais específica em parâmetros de construtor. Por exemplo, o `BankAccount` tem requisitos específicos para os parâmetros `owner` e `accountID`: o `owner` não deve ser `null` ou um espaço em branco e `accountID` deve ser uma cadeia de caracteres que contém 10 dígitos. Você pode adicionar essa validação ao atribuir as propriedades correspondentes:

C#

```
public class BankAccount(string accountID, string owner)
{
    public string AccountID { get; } = ValidAccountNumber(accountID)
        ? accountID
        : throw new ArgumentException("Invalid account number",
nameof(accountID));

    public string Owner { get; } = string.IsNullOrWhiteSpace(owner)
        ? throw new ArgumentException("Owner name cannot be empty",
nameof(owner))
        : owner;

    public override string ToString() => $"Account ID: {AccountID}, Owner:
{Owner}";

    public static bool ValidAccountNumber(string accountID) =>
```

```
    accountID?.Length == 10 && accountID.All(c => char.IsDigit(c));
}
```

O exemplo anterior mostra como você pode validar os parâmetros do construtor antes de atribuí-los às propriedades. Você pode usar métodos internos, como String.IsNullOrWhiteSpace(String), ou seu método de validação, como `ValidAccountNumber`. No exemplo anterior, todas as exceções são geradas pelo construtor quando ele invoca os inicializadores. Se um parâmetro de construtor não for usado para atribuir um campo, todas as exceções serão geradas quando o parâmetro do construtor for acessado pela primeira vez.

Uma classe derivada apresentaria uma conta corrente:

C#

```
public class CheckAccount(string accountID, string owner, decimal
overdraftLimit = 0) : BankAccount(accountID, owner)
{
    public decimal CurrentBalance { get; private set; } = 0;

    public void Deposit(decimal amount)
    {
        if (amount < 0)
        {
            throw new ArgumentOutOfRangeException(nameof(amount), "Deposit
amount must be positive");
        }
        CurrentBalance += amount;
    }

    public void Withdrawal(decimal amount)
    {
        if (amount < 0)
        {
            throw new ArgumentOutOfRangeException(nameof(amount),
"Withdrawal amount must be positive");
        }
        if (CurrentBalance - amount < -overdraftLimit)
        {
            throw new InvalidOperationException("Insufficient funds for
withdrawal");
        }
        CurrentBalance -= amount;
    }

    public override string ToString() => $"Account ID: {AccountID}, Owner:
{Owner}, Balance: {CurrentBalance}";
}
```

A classe derivada `CheckingAccount` tem um construtor primário que usa todos os parâmetros necessários na classe base, e outro parâmetro com um valor padrão. O construtor primário chama o construtor base usando a sintaxe `: BankAccount(accountID, owner)`. Essa expressão especifica o tipo para a classe base e os argumentos para o construtor primário.

Não é necessário que sua classe derivada use um construtor primário. Você pode criar um construtor na classe derivada que invoca o construtor primário da classe base, como mostrado no exemplo a seguir:

C#

```
public class LineOfCreditAccount : BankAccount
{
    private readonly decimal _creditLimit;
    public LineOfCreditAccount(string accountID, string owner, decimal
creditLimit) : base(accountID, owner)
    {
        _creditLimit = creditLimit;
    }
    public decimal CurrentBalance { get; private set; } = 0;

    public void Deposit(decimal amount)
    {
        if (amount < 0)
        {
            throw new ArgumentOutOfRangeException(nameof(amount), "Deposit
amount must be positive");
        }
        CurrentBalance += amount;
    }

    public void Withdrawal(decimal amount)
    {
        if (amount < 0)
        {
            throw new ArgumentOutOfRangeException(nameof(amount),
"Withdrawal amount must be positive");
        }
        if (CurrentBalance - amount < -_creditLimit)
        {
            throw new InvalidOperationException("Insufficient funds for
withdrawal");
        }
        CurrentBalance -= amount;
    }

    public override string ToString() => $"{base.ToString()}, Balance:
{CurrentBalance}";
}
```

Há uma preocupação potencial com hierarquias de classe e construtores primários: é possível criar várias cópias de um parâmetro de construtor primário, já que ele é usado em classes base e derivadas. O exemplo de código a seguir cria duas cópias de cada um dos campos `owner` e `accountID`:

C#

```
public class SavingsAccount(string accountID, string owner, decimal
interestRate) : BankAccount(accountID, owner)
{
    public SavingsAccount() : this("default", "default", 0.01m) { }
    public decimal CurrentBalance { get; private set; } = 0;

    public void Deposit(decimal amount)
    {
        if (amount < 0)
        {
            throw new ArgumentOutOfRangeException(nameof(amount), "Deposit
amount must be positive");
        }
        CurrentBalance += amount;
    }

    public void Withdrawal(decimal amount)
    {
        if (amount < 0)
        {
            throw new ArgumentOutOfRangeException(nameof(amount),
"Withdrawal amount must be positive");
        }
        if (CurrentBalance - amount < 0)
        {
            throw new InvalidOperationException("Insufficient funds for
withdrawal");
        }
        CurrentBalance -= amount;
    }

    public void ApplyInterest()
    {
        CurrentBalance *= 1 + interestRate;
    }

    public override string ToString() => $"Account ID: {accountID}, Owner:
{owner}, Balance: {CurrentBalance}";
}
```

A linha realçada mostra que o método `ToString` usa os *parâmetros do construtor primário* (`owner` e `accountID`) em vez das *propriedades da classe base* (`Owner` e `AccountID`). O resultado é que a classe derivada `SavingsAccount` cria o armazenamento

para essas cópias. A cópia na classe derivada é diferente da propriedade na classe base. Se for possível modificar a propriedade da classe base, a instância da classe derivada não verá essa modificação. O compilador emite um aviso para os parâmetros do construtor primário usados em uma classe derivada e passados para um construtor de classe base. Nessa instância, a correção envolve usar as propriedades na interface de classe base.

Resumo

Você pode usar os construtores primários da maneira mais adequada ao seu design. Em classes e structs, os parâmetros do construtor primário são os parâmetros para um construtor que deve ser invocado. Você pode usá-los para inicializar propriedades. Você pode inicializar campos. Essas propriedades ou campos podem ser mutáveis ou imutáveis. Você pode usá-los em métodos. Esses são parâmetros para usar da maneira mais adequada ao seu design. Você pode saber mais sobre construtores primários no artigo do guia de programação em C# sobre construtores de instâncias e a especificação de construtor primário proposta.

Tutorial: Explorar o recurso C# 11 – membros virtuais estáticos em interfaces

17/06/2025

O C# 11 e o .NET 7 incluem *membros virtuais estáticos em interfaces*. Esse recurso permite definir interfaces que incluem operadores sobrecarregados ou outros membros estáticos. Depois de definir interfaces com membros estáticos, você pode usar essas interfaces como restrições para criar tipos genéricos que usam operadores ou outros métodos estáticos. Mesmo que você não crie interfaces com operadores sobrecarregados, provavelmente se beneficiará desse recurso e das classes matemáticas genéricas habilitadas pela atualização da linguagem.

Neste tutorial, você aprenderá como:

- ✓ Definir interfaces com membros estáticos.
- ✓ Use interfaces para definir classes que implementam interfaces com operadores definidos.
- ✓ Crie algoritmos genéricos que dependem de métodos de interface estática.

Pré-requisitos

- O .NET SDK mais recente
- Editor do Visual Studio Code
- O DevKit C#

Métodos de interface abstrato estáticos

Vamos começar com um exemplo. O método a seguir retorna o ponto médio de dois números `double`:

C#

```
public static double MidPoint(double left, double right) =>
    (left + right) / (2.0);
```

A mesma lógica funcionaria para qualquer tipo numérico: `int`, `short`, `long`, `float`, `decimal` ou qualquer tipo que represente um número. Você precisa ter uma maneira de usar os operadores `+` e `/`, e definir um valor para `2`. Você pode usar a interface System.Numerics.INumber<TSelf> para escrever o método anterior como o seguinte método genérico:

C#

```
public static T MidPoint<T>(T left, T right)
    where T : INumber<T> => (left + right) / T.CreateChecked(2);  // note: the
addition of left and right may overflow here; it's just for demonstration purposes
```

Qualquer tipo que implemente a interface INumber<TSelf> deve incluir uma definição para `operator +` e `operator /`. O denominador é definido por `T.CreateChecked(2)` para criar o valor `2` para qualquer tipo numérico, o que força o denominador a ser do mesmo tipo que os dois parâmetros. INumberBase<TSelf>.CreateChecked<TOther>(TOther) cria uma instância do tipo do valor especificado e gera um OverflowException se o valor ficar fora do intervalo representável. (Essa implementação tem o potencial de estouro se `left` e `right` são valores grandes o suficiente. Há algoritmos alternativos que podem evitar esse possível problema.)

Você define membros abstratos estáticos em uma interface usando a sintaxe familiar: você adiciona os modificadores `static` e `abstract` a qualquer membro estático que não forneça uma implementação. O exemplo a seguir define uma interface `IGetNext<T>` que pode ser aplicada a qualquer tipo que substitua `operator ++`:

C#

```
public interface IGetNext<T> where T : IGetNext<T>
{
    static abstract T operator ++(T other);
}
```

A restrição de que o argumento de tipo, `T` implementa `IGetNext<T>`, garante que a assinatura do operador inclua o tipo que contém ou seu argumento de tipo. Muitos operadores impõem que seus parâmetros devem corresponder ao tipo ou ser o parâmetro de tipo restrito para implementar o tipo que contém. Sem essa restrição, o operador `++` não pôde ser definido na interface `IGetNext<T>`.

Você pode criar uma estrutura que cria uma cadeia de caracteres 'A' em que cada incremento adiciona outro caractere à cadeia de caracteres usando o seguinte código:

C#

```
public struct RepeatSequence : IGetNext<RepeatSequence>
{
    private const char Ch = 'A';
    public string Text = new string(Ch, 1);

    public RepeatSequence() {}

    public static RepeatSequence operator ++(RepeatSequence other)
        => other with { Text = other.Text + Ch };
```

```
    public override string ToString() => Text;
}
```

De modo mais geral, você pode criar qualquer algoritmo em que queira definir `++` para significar "produzir o próximo valor desse tipo". O uso dessa interface produz código e resultados claros:

C#

```
var str = new RepeatSequence();

for (int i = 0; i < 10; i++)
    Console.WriteLine(str++);
```

O código anterior gerencia a saída a seguir:

PowerShell

```
A
AA
AAA
AAAA
AAAAA
AAAAAA
AAAAAAA
AAAAAAAA
AAAAAAAAA
AAAAAAAAAA
```

Este pequeno exemplo demonstra a motivação para esse recurso. Você pode usar sintaxe natural para operadores, valores constantes e outras operações estáticas. Você pode explorar essas técnicas ao criar vários tipos que dependem de membros estáticos, incluindo operadores sobrecarregados. Defina as interfaces que correspondem aos recursos de seus tipos e declare o suporte desses tipos para a nova interface.

Matemática genérica

O cenário motivador para permitir métodos estáticos, incluindo operadores, em interfaces é dar suporte a algoritmos matemáticos genéricos. A biblioteca de classes base do .NET 7 contém definições de interface para muitos operadores aritméticos e interfaces derivadas que combinam muitos operadores aritméticos em uma interface `INumber<T>`. Vamos aplicar esses tipos para criar um registro `Point<T>` que possa usar qualquer tipo numérico para `T`. O ponto pode ser movido por alguns `XOffset` e `YOffset` usando o operador `+`.

Comece criando um novo aplicativo console usando `dotnet new` ou o Visual Studio.

A interface pública para o `Translation<T>` e `Point<T>` deve se parecer com o seguinte código:

```C#
// Note: Not complete. This won't compile yet.
public record Translation<T>(T XOffset, T YOffset);

public record Point<T>(T X, T Y)
{
    public static Point<T> operator +(Point<T> left, Translation<T> right);
}
```

Você usa o tipo `record` para os tipos `Translation<T>` e `Point<T>`: ambos armazenam dois valores e eles representam o armazenamento de dados em vez de um comportamento sofisticado. A implementação de `operator +` seria semelhante ao seguinte código:

```C#
public static Point<T> operator +(Point<T> left, Translation<T> right) =>
    left with { X = left.X + right.XOffset, Y = left.Y + right.YOffset };
```

Para que o código anterior seja compilado, você precisará declarar que `T` dá suporte à interface `IAdditionOperators<TSelf, TOther, TResult>`. Essa interface inclui o método estático `operator +`. Ele declara três parâmetros de tipo: um para o operando à esquerda, um para o operando à direita e outro para o resultado. Alguns tipos implementam `+` para diferentes tipos de operando e de resultado. Adicione uma declaração de que o argumento `T` de tipo implementa `IAdditionOperators<T, T, T>`:

```C#
public record Point<T>(T X, T Y) where T : IAdditionOperators<T, T, T>
```

Depois de adicionar essa restrição, sua classe `Point<T>` poderá usar o `+` para seu operador de adição. Adicione a mesma restrição à declaração `Translation<T>`:

```C#
public record Translation<T>(T XOffset, T YOffset) where T : IAdditionOperators<T,
T, T>;
```

A restrição `IAdditionOperators<T, T, T>` impede que um desenvolvedor que usa sua classe crie um `Translation` usando um tipo que não atenda à restrição para a adição a um ponto. Você adicionou as restrições necessárias ao parâmetro de tipo para `Translation<T>` e `Point<T>`, portanto, esse código funciona. Você pode testar adicionando código como o seguinte acima das declarações de `Translation` e `Point` em seu arquivo *Program.cs*:

```C#
var pt = new Point<int>(3, 4);

var translate = new Translation<int>(5, 10);

var final = pt + translate;

Console.WriteLine(pt);
Console.WriteLine(translate);
Console.WriteLine(final);
```

Você pode tornar esse código mais reutilizável declarando que esses tipos implementam as interfaces aritméticas apropriadas. A primeira alteração a ser executada é declarar que `Point<T, T>` implementa a interface `IAdditionOperators<Point<T>, Translation<T>, Point<T>>`. O tipo `Point` usa diferentes tipos para operandos e o resultado. O tipo `Point` já implementa um `operator +` com essa assinatura, portanto, adicionar a interface à declaração é tudo o que você precisa:

```C#
public record Point<T>(T X, T Y) : IAdditionOperators<Point<T>, Translation<T>,
Point<T>>
    where T : IAdditionOperators<T, T, T>
```

Por fim, ao executar a adição, é útil ter uma propriedade que defina o valor de identidade aditivo para esse tipo. Há uma nova interface para esse recurso: IAdditiveIdentity<TSelf,TResult>. Uma tradução de `{0, 0}` é a identidade aditiva: o ponto resultante é o mesmo que o operando à esquerda. A interface `IAdditiveIdentity<TSelf, TResult>` define uma propriedade readonly, `AdditiveIdentity`, que retorna o valor de identidade. A `Translation<T>` precisa de algumas alterações para implementar essa interface:

```C#
using System.Numerics;

public record Translation<T>(T XOffset, T YOffset) :
IAdditiveIdentity<Translation<T>, Translation<T>>
    where T : IAdditionOperators<T, T, T>, IAdditiveIdentity<T, T>
```

```
{
    public static Translation<T> AdditiveIdentity =>
        new Translation<T>(XOffset: T.AdditiveIdentity, YOffset:
T.AdditiveIdentity);
}
```

Há algumas alterações aqui, então vamos analisá-las uma por uma. Primeiro, você declara que o tipo `Translation` implementa a interface `IAdditiveIdentity`:

```C#
public record Translation<T>(T XOffset, T YOffset) :
IAdditiveIdentity<Translation<T>, Translation<T>>
```

Em seguida, você pode tentar implementar o membro da interface, conforme mostrado no seguinte código:

```C#
public static Translation<T> AdditiveIdentity =>
    new Translation<T>(XOffset: 0, YOffset: 0);
```

O código anterior não será compilado, pois `0` depende do tipo. A resposta: use `IAdditiveIdentity<T>.AdditiveIdentity` para `0`. Essa alteração significa que suas restrições agora devem incluir essa `T` implementa `IAdditiveIdentity<T>`. Isso resulta na seguinte implementação:

```C#
public static Translation<T> AdditiveIdentity =>
    new Translation<T>(XOffset: T.AdditiveIdentity, YOffset: T.AdditiveIdentity);
```

Agora que você adicionou essa restrição em `Translation<T>`, precisa adicionar a mesma restrição para `Point<T>`:

```C#
using System.Numerics;

public record Point<T>(T X, T Y) : IAdditionOperators<Point<T>, Translation<T>,
Point<T>>
    where T : IAdditionOperators<T, T, T>, IAdditiveIdentity<T, T>
{
    public static Point<T> operator +(Point<T> left, Translation<T> right) =>
```

```
        left with { X = left.X + right.XOffset, Y = left.Y + right.YOffset };
}
```

Este exemplo deu uma olhada em como as interfaces para composição matemática genérica. Você aprendeu a:

- ✔ Escrever um método que dependia da interface `INumber<T>` para que esse método pudesse ser usado com qualquer tipo numérico.
- ✔ Criar um tipo que depende das interfaces de adição para implementar um tipo que dá suporte apenas a uma operação matemática. Que o tipo declara seu suporte para essas mesmas interfaces para que ele possa ser composto de outras maneiras. Os algoritmos são escritos usando a sintaxe mais natural dos operadores matemáticos.

Experimente esses recursos e registre comentários. Você pode usar o item de menu *Enviar Comentários* no Visual Studio ou criar um novo problema no repositório roslyn no GitHub. Crie algoritmos genéricos que funcionam com qualquer tipo numérico. Criar algoritmos usando essas interfaces em que o argumento de tipo só pode implementar um subconjunto de recursos semelhantes a números. Mesmo que você não crie novas interfaces que usam essas funcionalidades, pode experimentar usá-las em seus algoritmos.

Confira também

- Matemática genérica

Criar tipos de registro

18/12/2024

Registos são tipos que utilizam igualdade baseada em valor . Você pode definir registros como tipos de referência ou tipos de valor. Duas variáveis de um tipo de registro são iguais se as definições de tipo de registro forem idênticas e se, para cada campo, os valores em ambos os registros forem iguais. Duas variáveis de um tipo de classe são iguais se os objetos referidos forem do mesmo tipo de classe e as variáveis se referirem ao mesmo objeto. A igualdade baseada em valor implica outros recursos que você provavelmente deseja em tipos de registro. O compilador gera muitos desses membros quando você declara um `record` em vez de um `class`. O compilador gera exatamente os mesmos métodos para tipos `record struct`.

Neste tutorial, você aprenderá a:

- ✔ Decida se deve adicionar o modificador `record` a um tipo `class`.
- ✔ Declare tipos de registro e tipos de registro posicional.
- ✔ Substitua seus métodos por métodos gerados pelo compilador em registros.

Pré-requisitos

- O mais recente SDK do .NET
- Editor de código do Visual Studio
- O Kit de Desenvolvimento C#

Características dos registos

Você define um registro declarando um tipo com a palavra-chave `record`, modificando uma declaração `class` ou `struct`. Opcionalmente, você pode omitir a palavra-chave `class` para criar um `record class`. Um registro segue a semântica de igualdade baseada em valor. Para impor a semântica de valores, o compilador gera vários métodos para seu tipo de registro (tanto para tipos de `record class` quanto para tipos de `record struct`):

- Uma substituição de Object.Equals(Object).
- Um método de `Equals` virtual cujo parâmetro é o tipo de registro.
- Uma substituição de Object.GetHashCode().
- Métodos para `operator ==` e `operator !=`.
- Os tipos de registro implementam System.IEquatable<T>.

Os registos também fornecem uma substituição de Object.ToString(). O compilador sintetiza métodos para exibir registros usando Object.ToString(). Você explora esses membros enquanto

escreve o código para este tutorial. Os registros suportam expressões `with` para permitir a mutação não destrutiva de registros.

Você também pode declarar *registros posicionais* usando uma sintaxe mais concisa. O compilador sintetiza mais métodos para você quando você declara registros posicionais:

- Um construtor primário cujos parâmetros correspondem aos parâmetros posicionais na declaração de registro.
- Propriedades públicas para cada parâmetro de um construtor primário. Essas propriedades são *somente de inicialização* para tipos `record class` e `readonly record struct`. Para `record struct` tipos, são de *leitura e escrita*.
- Um método `Deconstruct` para extrair propriedades do registro.

Criar dados de temperatura

Dados e estatísticas estão entre os cenários em que você deseja usar registros. Para este tutorial, você cria um aplicativo que calcula *dias de grau* para diferentes usos. *Graus-dias* são uma medida de calor (ou ausência dele) durante um período de dias, semanas ou meses. Os graus-dias rastreiam e prevêem o consumo de energia. Dias mais quentes significam mais ar condicionado e dias mais frios significam mais utilização do forno. Os graus-dias ajudam a gerir as populações de plantas e correlacionam-se com o crescimento das plantas à medida que as estações mudam. Os graus-dias ajudam a rastrear as migrações de espécies animais que viajam para se adaptarem ao clima.

A fórmula baseia-se na temperatura média num determinado dia e numa temperatura de base. Para calcular graus-dias ao longo do tempo, você precisará da temperatura alta e baixa todos os dias por um período de tempo. Vamos começar criando um novo aplicativo. Crie um novo aplicativo de console. Crie um novo tipo de registro em um novo arquivo chamado "DailyTemperature.cs":

C#

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

O código anterior define um registro posicional . O registro `DailyTemperature` é um `readonly record struct`, porque você não pretende herdar dele, e ele deve ser imutável. As propriedades `HighTemp` e `LowTemp` são *init only properties*, o que significa que podem ser definidas no construtor ou usando um inicializador de propriedade. Se quiseres que os parâmetros posicionais sejam de leitura e gravação, declara um `record struct` em vez de um `readonly record struct`. O tipo `DailyTemperature` também tem um construtor primário com dois parâmetros que correspondem às duas propriedades. Você usa o construtor primário para

inicializar um registo `DailyTemperature`. O código a seguir cria e inicializa vários registros `DailyTemperature`. O primeiro usa parâmetros nomeados para esclarecer o `HighTemp` e `LowTemp`. Os inicializadores restantes usam parâmetros posicionais para inicializar o `HighTemp` e `LowTemp`:

C#

```
private static DailyTemperature[] data = [
    new DailyTemperature(HighTemp: 57, LowTemp: 30),
    new DailyTemperature(60, 35),
    new DailyTemperature(63, 33),
    new DailyTemperature(68, 29),
    new DailyTemperature(72, 47),
    new DailyTemperature(75, 55),
    new DailyTemperature(77, 55),
    new DailyTemperature(72, 58),
    new DailyTemperature(70, 47),
    new DailyTemperature(77, 59),
    new DailyTemperature(85, 65),
    new DailyTemperature(87, 65),
    new DailyTemperature(85, 72),
    new DailyTemperature(83, 68),
    new DailyTemperature(77, 65),
    new DailyTemperature(72, 58),
    new DailyTemperature(77, 55),
    new DailyTemperature(76, 53),
    new DailyTemperature(80, 60),
    new DailyTemperature(85, 66)
];
```

Você pode adicionar suas próprias propriedades ou métodos aos registros, incluindo registros posicionais. Você precisa calcular a temperatura média para cada dia. Você pode adicionar essa propriedade ao registro `DailyTemperature`:

C#

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

Vamos nos certificar de que você pode usar esses dados. Adicione o seguinte código ao seu método `Main`:

C#

```
foreach (var item in data)
```

```
    Console.WriteLine(item);
```

Execute seu aplicativo e você verá uma saída semelhante à exibição a seguir (várias linhas removidas por espaço):

CLI do .NET

```
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 }
```

O código anterior mostra o resultado da sobrescrição de `ToString` que foi sintetizada pelo compilador. Se você preferir texto diferente, você pode escrever sua própria versão do `ToString` que impede o compilador de sintetizar uma versão para você.

Calcular dias de grau

Para calcular graus-dias, calcula-se a diferença entre uma temperatura de referência e a temperatura média de um determinado dia. Para medir o calor ao longo do tempo, você descarta todos os dias em que a temperatura média está abaixo da linha de base. Para medir o frio ao longo do tempo, você descarta todos os dias em que a temperatura média está acima da linha de base. Por exemplo, os EUA usam 65 F como base para dias de grau de aquecimento e resfriamento. Essa é a temperatura em que não é necessário aquecimento ou refrigeração. Se um dia tem uma temperatura média de 70 F, esse dia tem cinco graus-dia de arrefecimento e zero graus-dia de aquecimento. Por outro lado, se a temperatura média fosse 55 Fahrenheit, esse dia seria de 10 dias de aquecimento e 0 dias de arrefecimento.

Você pode expressar essas fórmulas como uma pequena hierarquia de tipos de registro: um tipo abstrato de grau-dia e dois tipos concretos para graus-dias de aquecimento e graus-dias de resfriamento. Esses tipos também podem ser registros posicionais. Eles consideram uma temperatura de referência e uma sequência de registos diários de temperatura, como argumentos para o construtor primário.

C#

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

public sealed record HeatingDegreeDays(double BaseTemperature,
IEnumerable<DailyTemperature> TempRecords)
    : DegreeDays(BaseTemperature, TempRecords)
```

```
{
    public double DegreeDays => TempRecords.Where(s => s.Mean <
BaseTemperature).Sum(s => BaseTemperature - s.Mean);
}

public sealed record CoolingDegreeDays(double BaseTemperature,
IEnumerable<DailyTemperature> TempRecords)
    : DegreeDays(BaseTemperature, TempRecords)
{
    public double DegreeDays => TempRecords.Where(s => s.Mean >
BaseTemperature).Sum(s => s.Mean - BaseTemperature);
}
```

O registro `DegreeDays` abstrato é a classe base compartilhada para os registros `HeatingDegreeDays` e `CoolingDegreeDays`. As declarações do construtor primário nos registros derivados mostram como gerenciar a inicialização do registro base. Seu registro derivado declara parâmetros para todos os parâmetros no construtor primário do registro base. O registro base declara e inicializa essas propriedades. O registro derivado não os oculta, mas apenas cria e inicializa propriedades para parâmetros que não são declarados em seu registro base. Neste exemplo, os registros derivados não adicionam novos parâmetros primários do construtor. Teste seu código adicionando o seguinte código ao seu método `Main`:

C#

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

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

Você obtém uma saída como a exibida no ecrã:

CLI do .NET

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

Definir métodos sintetizados pelo compilador

Seu código calcula o número correto de graus-dias de aquecimento e resfriamento durante esse período de tempo. Mas este exemplo mostra por que você pode querer substituir alguns dos métodos sintetizados para registros. Você pode declarar sua própria versão de qualquer um dos métodos sintetizados pelo compilador em um tipo de registro, exceto o método clone.

O método clone tem um nome gerado pelo compilador e você não pode fornecer uma implementação diferente. Esses métodos sintetizados incluem um Construtor de Cópia, os membros da interface System.IEquatable<T>, os testes de igualdade e desigualdade, e GetHashCode(). Para isso, você sintetiza `PrintMembers`. Você também pode declarar o seu próprio `ToString`, mas `PrintMembers` oferece uma opção melhor para cenários de herança. Para fornecer sua própria versão de um método sintetizado, a assinatura deve corresponder ao método sintetizado.

O elemento `TempRecords` na saída do console não é útil. Ele apresenta o tipo, mas nada mais. Você pode alterar esse comportamento fornecendo sua própria implementação do método `PrintMembers` sintetizado. A assinatura depende dos modificadores aplicados à declaração `record`:

- Se um tipo de registro for `sealed` ou um `record struct`, a assinatura será `private bool PrintMembers(StringBuilder builder);`
- Se um tipo de registro não for `sealed` e derivar de `object` (ou seja, não declarar um registro base), a assinatura será `protected virtual bool PrintMembers(StringBuilder builder);`
- Se um tipo de registro não for `sealed` e derivar de outro registro, a assinatura será `protected override bool PrintMembers(StringBuilder builder);`

Estas regras são mais fáceis de compreender através da compreensão do propósito de `PrintMembers`. `PrintMembers` adiciona informações sobre cada propriedade em um tipo de registro a uma cadeia de caracteres. O contrato exige que os registos base adicionem os seus membros à apresentação e pressupõe que os membros derivados adicionem os seus membros. Cada tipo de registo sintetiza uma substituição de `ToString` que é semelhante ao exemplo a seguir para `HeatingDegreeDays`:

C#

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

Você declara um método `PrintMembers` no registro `DegreeDays` que não imprime o tipo da coleção:

```
C#

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

A assinatura declara um método `virtual protected` para corresponder à versão do compilador. Não se preocupe se errar nos métodos de acesso; a linguagem impõe a assinatura correta. Se você esquecer os modificadores corretos para qualquer método sintetizado, o compilador emitirá avisos ou erros que ajudam você a obter a assinatura correta.

Você pode declarar o método `ToString` como `sealed` em um tipo de registro. Isso impede que os registros derivados forneçam uma nova implementação. Os registos derivados ainda conterão o override de `PrintMembers`. Você selaria `ToString` se não quisesse que ele exibisse o tipo de tempo de execução do registro. No exemplo anterior, você perderia as informações sobre onde o registro estava medindo graus-dias de aquecimento ou resfriamento.

Mutação não destrutiva

Os membros sintetizados em uma classe de registro posicional não modificam o estado do registro. O objetivo é que você possa criar registros imutáveis com mais facilidade. Lembre-se de que você declara uma `readonly record struct` para criar uma estrutura de registro imutável. Analise novamente as declarações anteriores para `HeatingDegreeDays` e `CoolingDegreeDays`. Os membros adicionados executam cálculos sobre os valores do registo, mas não mudam de estado. Os registros posicionais facilitam a criação de tipos de referência imutáveis.

Criar tipos de referência imutáveis significa que você deseja usar mutações não destrutivas. Você cria novas instâncias de registro que são semelhantes às instâncias de registro existentes usando expressões with. Essas expressões são uma construção de cópia com atribuições extras que modificam a cópia. O resultado é uma nova instância de registro em que cada propriedade foi copiada do registro existente e, opcionalmente, modificada. O registo original mantém-se inalterado.

Vamos adicionar um par de recursos ao seu programa que demonstram expressões `with`. Primeiro, vamos criar um novo registo para computar graus-dia acumulados usando os mesmos dados. *Graus-dias de crescimento* normalmente utiliza 41 F como valor base e mede

temperaturas acima do valor base. Para usar os mesmos dados, você pode criar um novo registro semelhante ao `coolingDegreeDays`, mas com uma temperatura base diferente:

```
C#

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

Você pode comparar o número de graus computados com os números gerados com uma temperatura de base mais alta. Lembre-se de que os registros são *tipos de referência* e essas cópias são cópias superficiais. A matriz dos dados não é copiada, mas ambos os registros se referem aos mesmos dados. Esse facto é uma vantagem num outro cenário. Para os dias de soma térmica, é útil acompanhar o total dos últimos cinco dias. Você pode criar novos registros com dados de origem diferentes usando expressões `with`. O código a seguir cria uma coleção dessas acumulações e, em seguida, exibe os valores:

```
C#

// showing moving accumulation of 5 days using range syntax
List<CoolingDegreeDays> movingAccumulation = new();
int rangeSize = (data.Length > 5) ? 5 : data.Length;
for (int start = 0; start < data.Length - rangeSize; start++)
{
    var fiveDayTotal = growingDegreeDays with { TempRecords = data[start..(start +
rangeSize)] };
    movingAccumulation.Add(fiveDayTotal);
}
Console.WriteLine();
Console.WriteLine("Total degree days in the last five days");
foreach(var item in movingAccumulation)
{
    Console.WriteLine(item);
}
```

Você também pode usar expressões `with` para criar cópias de registros. Não especifique propriedades entre as chaves na expressão `with`. Isso significa criar uma cópia e não alterar nenhuma propriedade:

```
C#

var growingDegreeDaysCopy = growingDegreeDays with { };
```

Execute o aplicativo concluído para ver os resultados.

Resumo

Este tutorial mostrou vários aspetos dos registos. Os registros fornecem sintaxe concisa para tipos em que o uso fundamental é o armazenamento de dados. Para classes orientadas a objetos, o uso fundamental é definir responsabilidades. Este tutorial concentrou-se em *registros posicionais*, onde você pode usar uma sintaxe concisa para declarar as propriedades de um registro. O compilador sintetiza vários membros do registro para copiar e comparar registros. Pode adicionar quaisquer outros membros de que necessite para os seus tipos de registo. Você pode criar tipos de registro imutáveis sabendo que nenhum dos membros gerados pelo compilador mudaria de estado. As `with` expressões tornam fácil o suporte a mutações não destrutivas.

Os registros adicionam outra maneira de definir tipos. Você utiliza definições `class` para criar hierarquias orientadas a objetos que se concentram nas responsabilidades e no comportamento dos objetos. Você cria tipos de `struct` para estruturas de dados que armazenam dados e são pequenas o suficiente para copiar com eficiência. Você cria tipos de `record` quando deseja igualdade e comparação baseadas em valor, não deseja copiar valores e deseja usar variáveis de referência. Quando deseja as características de registos para um tipo que seja pequeno o suficiente para copiar com eficiência, você cria tipos de `record struct`.

Você pode saber mais sobre registos no artigo de referência da linguagem C# para o tipo de registo, a especificação proposta para tipo de registo e a especificação de estrutura de registo .

Tutorial: Explorar ideias usando instruções de nível superior para criar código conforme você aprende

Artigo • 22/01/2025

Neste tutorial, você aprenderá a:

- ✓ Conheça as regras que regem o uso de instruções de nível superior.
- ✓ Use instruções de nível superior para explorar algoritmos.
- ✓ Refatore explorações em componentes reutilizáveis.

Pré-requisitos

Você precisa configurar seu computador para executar o .NET 6 ou posterior. O compilador C# está disponível a partir do Visual Studio 2022 ou do .NET SDK.

Este tutorial pressupõe que você esteja familiarizado com o C# e .NET, incluindo o Visual Studio ou a CLI do .NET.

Comece a explorar

As instruções de nível superior permitem evitar a cerimônia extra necessária colocando o ponto de entrada do programa em um método estático em uma classe. O ponto de partida típico para um novo aplicativo de console se parece com o seguinte código:

C#

```
using System;

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

O código anterior é o resultado da execução do comando `dotnet new console` e da criação de um novo aplicativo de console. Essas 11 linhas contêm apenas uma linha de código executável. Você pode simplificar o programa usando a nova funcionalidade de instruções de nível superior. Isso possibilita remover todas as linhas deste programa, exceto duas.

C#

```
// See https://aka.ms/new-console-template for more information
Console.WriteLine("Hello, World!");
```

> ⓘ **Importante**
>
> Os modelos C# para o .NET 6 usam *instruções de nível superior*. Se você já tiver atualizado para o .NET 6, talvez seu aplicativo não corresponda ao código descrito neste artigo. Para obter mais informações, consulte o artigo sobre Novos modelos C# geram instruções de nível superior
>
> O SDK do .NET 6 também adiciona um conjunto de diretrizes implícitas de para projetos que usam os seguintes SDKs:
>
> - Microsoft.NET.Sdk
> - Microsoft.NET.Sdk.Web
> - Microsoft.NET.Sdk.Worker
>
> Essas diretivas implícitas `global using` incluem os namespaces mais comuns para o tipo de projeto.
>
> Para obter mais informações, consulte o artigo sobre Diretivas de uso implícito

Esse recurso simplifica sua exploração de novas ideias. Você pode usar instruções de nível superior em cenários de script ou para explorar. Depois de ter o básico funcionando, você pode começar a refatorar o código e criar métodos, classes ou outros conjuntos para componentes reutilizáveis que você criou. As instruções de nível superior habilitam experimentações rápidas e tutoriais iniciantes. Elas também fornecem um caminho tranquilo da experimentação a programas completos.

As instruções de nível superior são executadas na ordem em que aparecem no arquivo. As instruções de nível superior só podem ser usadas em um arquivo de origem em seu aplicativo. O compilador gera um erro se você usá-los em mais de um arquivo.

Criar um computador de resposta mágico do .NET

Neste tutorial, vamos criar um aplicativo de console que responda a uma pergunta "sim" ou "não" com uma resposta aleatória. Você cria a funcionalidade passo a passo. Você pode se concentrar em sua tarefa em vez da cerimônia necessária para a estrutura de um programa típico. Em seguida, quando estiver satisfeito com a funcionalidade, você poderá refatorar o aplicativo conforme julgar adequado.

Um bom ponto de partida é escrever a pergunta novamente no console. Você pode começar gravando o seguinte código:

C#

```
Console.WriteLine(args);
```

Você não declara uma variável `args`. Para o arquivo de origem único que contém suas instruções de nível superior, o compilador reconhece `args` como significando os argumentos de linha de comando. O tipo de argumento é um `string[]`, como em todos os programas C#.

O código pode ser testado executando o comando `dotnet run` a seguir:

CLI do .NET

```
dotnet run -- Should I use top level statements in all my programs?
```

Os argumentos após a linha de comando `--` são passados para o programa. Você pode ver o tipo da variável `args` impressa no console:

Console

```
System.String[]
```

Para escrever a pergunta no console, é necessário enumerar os argumentos e separá-los com um espaço. Substitua a chamada `WriteLine` pelo código a seguir:

C#

```
Console.WriteLine();
foreach(var s in args)
{
    Console.Write(s);
```

```
    Console.Write(' ');
}
Console.WriteLine();
```

Agora, quando você executa o programa, ele exibe corretamente a pergunta como uma cadeia de caracteres de argumentos.

Responder com uma resposta aleatória

Após ecoar a pergunta, você pode adicionar o código para gerar a resposta aleatória. Comece adicionando uma matriz de respostas possíveis:

C#

```
string[] answers =
[
    "It is certain.",       "Reply hazy, try again.",     "Don’t count on
it.",
    "It is decidedly so.",  "Ask again later.",           "My reply is no.",
    "Without a doubt.",     "Better not tell you now.",   "My sources say
no.",
    "Yes – definitely.",    "Cannot predict now.",        "Outlook not so
good.",
    "You may rely on it.",  "Concentrate and ask again.", "Very doubtful.",
    "As I see it, yes.",
    "Most likely.",
    "Outlook good.",
    "Yes.",
    "Signs point to yes.",
];
```

Essa matriz tem dez respostas afirmativas, cinco não confirmadas e cinco negativas. Em seguida, adicione o seguinte código para gerar e exibir uma resposta aleatória da matriz:

C#

```
var index = new Random().Next(answers.Length - 1);
Console.WriteLine(answers[index]);
```

Você pode executar o aplicativo novamente para ver os resultados. Você deve ver algo semelhante à seguinte saída:

CLI do .NET

```
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.
```

O código para gerar uma resposta inclui uma declaração de variável em suas instruções de nível superior. O compilador inclui essa declaração no método de `Main` gerado pelo compilador. Como essas declarações de variável são variáveis locais, você não pode incluir o modificador de `static`.

Esse código responde às perguntas, mas vamos adicionar mais um recurso. Você gostaria que seu aplicativo de perguntas simulasse o pensamento sobre a resposta. Você pode fazer isso adicionando um pouco de animação do ASCII e pausando enquanto trabalha. Adicione o seguinte código após a linha que ecoa a pergunta:

C#

```
for (int i = 0; i < 20; i++)
{
    Console.Write("| -");
    await Task.Delay(50);
    Console.Write("\b\b\b");
    Console.Write("/ \\");
    await Task.Delay(50);
    Console.Write("\b\b\b");
    Console.Write("- |");
    await Task.Delay(50);
    Console.Write("\b\b\b");
    Console.Write("\\ /");
    await Task.Delay(50);
    Console.Write("\b\b\b");
}
Console.WriteLine();
```

Também será necessário adicionar uma diretiva `using` à parte superior do arquivo de origem:

C#

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

As diretivas `using` devem aparecer antes de qualquer outra instrução no arquivo. Caso contrário, é um erro do compilador. Você pode executar o programa novamente e ver a animação. Isso torna uma experiência melhor. Experimente a duração do atraso para corresponder ao seu gosto.

O código anterior cria um conjunto de linhas de rotação separadas por um espaço. A adição da palavra-chave `await` instrui o compilador a gerar o ponto de entrada do

programa como um método que tem o modificador `async` e retorna um System.Threading.Tasks.Task. Esse programa não retorna um valor, portanto, o ponto de entrada do programa retorna um `Task`. Se o programa retornar um valor inteiro, você adicionará uma instrução return ao final de suas instruções de nível superior. Essa instrução return especificaria o valor inteiro a ser retornado. Se suas instruções de nível superior incluírem uma expressão `await`, o tipo de retorno se tornará System.Threading.Tasks.Task<TResult>.

Refatoração para o futuro

Seu programa deve ser parecido com o seguinte código:

C#

```
Console.WriteLine();
foreach(var s in args)
{
    Console.Write(s);
    Console.Write(' ');
}
Console.WriteLine();

for (int i = 0; i < 20; i++)
{
    Console.Write("| -");
    await Task.Delay(50);
    Console.Write("\b\b\b");
    Console.Write("/ \\");
    await Task.Delay(50);
    Console.Write("\b\b\b");
    Console.Write("- |");
    await Task.Delay(50);
    Console.Write("\b\b\b");
    Console.Write("\\ /");
    await Task.Delay(50);
    Console.Write("\b\b\b");
}
Console.WriteLine();

string[] answers =
[
    "It is certain.",       "Reply hazy, try again.",     "Don't count on
it.",
    "It is decidedly so.",  "Ask again later.",           "My reply is no.",
    "Without a doubt.",     "Better not tell you now.",   "My sources say
no.",
    "Yes – definitely.",    "Cannot predict now.",        "Outlook not so
good.",
    "You may rely on it.",  "Concentrate and ask again.", "Very doubtful.",
    "As I see it, yes.",
```

```
    "Most likely.",
    "Outlook good.",
    "Yes.",
    "Signs point to yes.",
];

var index = new Random().Next(answers.Length - 1);
Console.WriteLine(answers[index]);
```

O código anterior é razoável. Funciona. Mas não é reutilizável. Agora que o aplicativo está funcionando, é hora de retirar as partes reutilizáveis.

Um candidato é o código que exibe a animação de espera. Esse snippet pode se tornar um método:

Você pode começar criando uma função local em seu arquivo. Substitua a animação atual pelo seguinte:

C#

```
await ShowConsoleAnimation();

static async Task ShowConsoleAnimation()
{
    for (int i = 0; i < 20; i++)
    {
        Console.Write("| -");
        await Task.Delay(50);
        Console.Write("\b\b\b");
        Console.Write("/ \\");
        await Task.Delay(50);
        Console.Write("\b\b\b");
        Console.Write("- |");
        await Task.Delay(50);
        Console.Write("\b\b\b");
        Console.Write("\\ /");
        await Task.Delay(50);
        Console.Write("\b\b\b");
    }
    Console.WriteLine();
}
```

O código anterior cria uma função local no método principal. Esse código ainda não é reutilizável. Então, extraia esse código em uma classe. Crie um novo arquivo chamado *utilities.cs* e adicione o seguinte código:

C#

```
namespace MyNamespace
{
```

```
    public static class Utilities
    {
        public static async Task ShowConsoleAnimation()
        {
            for (int i = 0; i < 20; i++)
            {
                Console.Write("| -");
                await Task.Delay(50);
                Console.Write("\b\b\b");
                Console.Write("/ \\");
                await Task.Delay(50);
                Console.Write("\b\b\b");
                Console.Write("- |");
                await Task.Delay(50);
                Console.Write("\b\b\b");
                Console.Write("\\ /");
                await Task.Delay(50);
                Console.Write("\b\b\b");
            }
            Console.WriteLine();
        }
    }
}
```

Um arquivo com instruções de nível superior também pode conter namespaces e tipos no final do arquivo, após as instruções de nível superior. Porém, para este tutorial, você coloca o método de animação em um arquivo separado para torná-lo mais facilmente reutilizável.

Por fim, você pode limpar o código de animação para remover algumas duplicações usando o laço `foreach` para iterar por meio do conjunto de elementos de animação definidos na matriz `animations`.
O método completo de `ShowConsoleAnimation` após o refatoramento deve ser semelhante ao seguinte código:

C#

```
public static async Task ShowConsoleAnimation()
{
    string[] animations = ["| -", "/ \\", "- |", "\\ /"];
    for (int i = 0; i < 20; i++)
    {
        foreach (string s in animations)
        {
            Console.Write(s);
            await Task.Delay(50);
            Console.Write("\b\b\b");
        }
    }
```

```
    Console.WriteLine();
}
```

Agora você tem um aplicativo completo e refatorou as partes reutilizáveis para uso futuro. Você pode chamar o novo método utilitário de suas instruções de nível superior, conforme exibido abaixo na versão final do programa principal:

C#

```
using MyNamespace;

Console.WriteLine();
foreach(var s in args)
{
    Console.Write(s);
    Console.Write(' ');
}
Console.WriteLine();

await Utilities.ShowConsoleAnimation();

string[] answers =
[
    "It is certain.",       "Reply hazy, try again.",     "Don’t count on
it.",
    "It is decidedly so.",  "Ask again later.",           "My reply is no.",
    "Without a doubt.",     "Better not tell you now.",   "My sources say
no.",
    "Yes – definitely.",    "Cannot predict now.",        "Outlook not so
good.",
    "You may rely on it.",  "Concentrate and ask again.", "Very doubtful.",
    "As I see it, yes.",
    "Most likely.",
    "Outlook good.",
    "Yes.",
    "Signs point to yes.",
];

var index = new Random().Next(answers.Length - 1);
Console.WriteLine(answers[index]);
```

O exemplo anterior adiciona a chamada a `Utilities.ShowConsoleAnimation` e insere outra diretiva `using`.

Resumo

As instruções de nível superior facilitam a criação de programas simples para uso para explorar novos algoritmos. É possível experimentar algoritmos tentando diferentes

snippets de código. Depois de aprender o que funciona, você pode refatorar o código para facilitar a manutenção.

As instruções de nível superior simplificam os programas que se baseiam em aplicativos de console. Esses aplicativos incluem funções do Azure, ações do GitHub e outros pequenos utilitários. Para obter mais informações, consulte Instruções de nível superior (Guia de Programação em C#).

Índices e intervalos

Artigo • 14/11/2023

Intervalos e índices fornecem uma sintaxe sucinta para acessar elementos únicos ou intervalos em uma sequência.

Neste tutorial, você aprenderá como:

- ✓ Use a sintaxe para intervalos em uma sequência.
- ✓ Defina implicitamente uma classe Range.
- ✓ Compreenda as decisões de design para o início e o fim de cada sequência.
- ✓ Conheça cenários para os tipos Index e Range.

Suporte a idioma para intervalos e índices

Índices e intervalos fornecem uma sintaxe sucinta para acessar elementos únicos ou intervalos em uma sequência.

Este suporte à linguagem depende de dois tipos novos e dois operadores novos:

- System.Index representa um índice em uma sequência.
- O índice do operador final ^, que especifica que um índice é relativo ao final de uma sequência.
- System.Range representa um subintervalo de uma sequência.
- O operador de intervalo .., que especifica o início e o fim de um intervalo como seus operandos.

Vamos começar com as regras para índices. Considere uma matriz `sequence`. O índice `0` é o mesmo que `sequence[0]`. O índice `^0` é o mesmo que `sequence[sequence.Length]`. A expressão `sequence[^0]` gera uma exceção, assim como `sequence[sequence.Length]` o faz. Para qualquer número `n`, o índice `^n` é o mesmo que `sequence.Length - n`.

C#

```
string[] words = [
                // index from start    index from end
    "The",      // 0                   ^9
    "quick",    // 1                   ^8
    "brown",    // 2                   ^7
    "fox",      // 3                   ^6
    "jumps",    // 4                   ^5
    "over",     // 5                   ^4
    "the",      // 6                   ^3
    "lazy",     // 7                   ^2
```

```
    "dog"       // 8                    ^1
];              // 9 (or words.Length) ^0
```

Você pode recuperar a última palavra com o índice `^1`. Adicione o código a seguir abaixo da inicialização:

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

Um intervalo especifica o *início* e o *final* de um intervalo. O início do intervalo é inclusivo, mas o final do intervalo é exclusivo, o que significa que o *início* está incluído no intervalo, mas o *final* não está incluído no intervalo. O intervalo `[0..^0]` representa todo o intervalo, assim como `[0..sequence.Length]` representa todo o intervalo.

O código a seguir cria um subintervalo com as palavras "quick", "brown" e "fox". Ele inclui `words[1]` até `words[3]`. O elemento `words[4]` não está no intervalo.

```C#
string[] quickBrownFox = words[1..4];
foreach (var word in quickBrownFox)
    Console.Write($"< {word} >");
Console.WriteLine();
```

O código a seguir retorna o intervalo com "lazy" e "dog". Ele inclui `words[^2]` e `words[^1]`. O índice final `words[^0]` não está incluído. Adicione o seguinte código também:

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

Os exemplos a seguir criam intervalos abertos para o início, fim ou ambos:

```C#
string[] allWords = words[..]; // contains "The" through "dog".
string[] firstPhrase = words[..4]; // contains "The" through "fox"
string[] lastPhrase = words[6..]; // contains "the", "lazy" and "dog"
foreach (var word in allWords)
    Console.Write($"< {word} >");
```

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

Você também pode declarar intervalos ou índices como variáveis. A variável então pode ser usada dentro dos caracteres `[` e `]`:

C#

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

O exemplo a seguir mostra muitos dos motivos para essas escolhas. Modifique `x`, `y` e `z` para tentar combinações diferentes. Quando você testar, use valores em que `x` é menor que `y` e `y` é menor que `z` para as combinações válidas. Adicione o seguinte código a um novo método. Tente usar combinações diferentes:

C#

```
int[] numbers = [..Enumerable.Range(0, 100)];
int x = 12;
int y = 25;
int z = 36;

Console.WriteLine($"{numbers[^x]} is the same as {numbers[numbers.Length -
x]}");
Console.WriteLine($"{numbers[x..y].Length} is the same as {y - x}");

Console.WriteLine("numbers[x..y] and numbers[y..z] are consecutive and
disjoint:");
Span<int> x_y = numbers[x..y];
Span<int> y_z = numbers[y..z];
Console.WriteLine($"\tnumbers[x..y] is {x_y[0]} through {x_y[^1]},
numbers[y..z] is {y_z[0]} through {y_z[^1]}");

Console.WriteLine("numbers[x..^x] removes x elements at each end:");
Span<int> x_x = numbers[x..^x];
Console.WriteLine($"\tnumbers[x..^x] starts with {x_x[0]} and ends with
{x_x[^1]}");

Console.WriteLine("numbers[..x] means numbers[0..x] and numbers[x..] means
```

```
numbers[x..^0]");
Span<int> start_x = numbers[..x];
Span<int> zero_x = numbers[0..x];
Console.WriteLine($"\t{start_x[0]}..{start_x[^1]} is the same as
{zero_x[0]}..{zero_x[^1]}");
Span<int> z_end = numbers[z..];
Span<int> z_zero = numbers[z..^0];
Console.WriteLine($"\t{z_end[0]}..{z_end[^1]} is the same as {z_zero[0]}..
{z_zero[^1]}");
```

Não apenas matrizes dão suporte a índices e intervalos. Você também pode usar índices e intervalos com cadeia de caracteres, Span<T> ou ReadOnlySpan<T>.

Conversões de expressão de operador de intervalo implícito

Ao usar a sintaxe de expressão do operador de intervalo, o compilador converte implicitamente os valores inicial e final em um Index e a partir deles cria uma instância Range. O código a seguir mostra um exemplo de conversão implícita da sintaxe de expressão do operador de intervalo e sua alternativa explícita correspondente:

C#

```
Range implicitRange = 3..^5;

Range explicitRange = new(
    start: new Index(value: 3, fromEnd: false),
    end: new Index(value: 5, fromEnd: true));

if (implicitRange.Equals(explicitRange))
{
    Console.WriteLine(
        $"The implicit range '{implicitRange}' equals the explicit range
'{explicitRange}'");
}
// Sample output:
//     The implicit range '3..^5' equals the explicit range '3..^5'
```

> ⓘ **Importante**
>
> Conversões implícitas de **Int32** para **Index** lançam uma **ArgumentOutOfRangeException** quando o valor é negativo. Da mesma forma, o construtor `Index` lança uma `ArgumentOutOfRangeException` quando o parâmetro `value` é negativo.

Suporte de tipo para índices e intervalos

Índices e intervalos fornecem sintaxe clara e concisa para acessar um único elemento ou um intervalo de elementos em uma sequência. Uma expressão de índice normalmente retorna o tipo dos elementos de uma sequência. Uma expressão de intervalo normalmente retorna o mesmo tipo de sequência que a sequência de origem.

Qualquer tipo que forneça um indexador com um parâmetro Index ou Range dá suporte explicitamente a índices ou intervalos, respectivamente. Um indexador que usa um único parâmetro Range pode retornar um tipo de sequência diferente, como System.Span<T>.

> ⓘ **Importante**
>
> O desempenho do código usando o operador de intervalo depende do tipo do operando da sequência.
>
> A complexidade temporal do operador de intervalo depende do tipo de sequência. Por exemplo, se a sequência for uma `string` ou uma matriz, o resultado será uma cópia da seção especificada da entrada. Portanto, a complexidade de tempo será *O(N)* (em que N é o comprimento do intervalo). Por outro lado, se for um **System.Span<T>** ou um **System.Memory<T>**, o resultado fará referência ao mesmo repositório de backup, o que significa que não há cópia e a operação é *O(1)*.
>
> Além da complexidade de tempo, isso causa alocações e cópias extras, afetando o desempenho. No código sensível ao desempenho, considere usar `Span<T>` ou `Memory<T>` como o tipo de sequência, já que o operador de intervalo não aloca para eles.

Um tipo é **contável** se tiver uma propriedade nomeada `Length` ou `Count` com um getter acessível e um tipo de retorno `int`. Um tipo contável que não dá suporte explicitamente a índices ou intervalos pode fornecer um suporte implícito para eles. Para obter mais informações, consulte as seções de suporte a índice implícito e suporte a intervalo implícito da nota de proposta de recurso. Intervalos que usam suporte de intervalo implícito retornam o mesmo tipo de sequência da sequência de origem.

Por exemplo, os seguintes tipos .NET dão suporte a índices e intervalos: String, Span<T> e ReadOnlySpan<T>. O List<T> suporta índices, mas não dá suporte a intervalos.

Array tem um comportamento com mais nuances. Matrizes de dimensão única dão suporte a índices e intervalos. Matrizes multidimensionais não dão suporte a

indexadores ou intervalos. O indexador de uma matriz multidimensional tem vários parâmetros, não um único parâmetro. Matrizes denteadas, também conhecidas como uma matriz de matrizes, dão suporte a intervalos e indexadores. O exemplo a seguir mostra como iterar uma subseção retangular de uma matriz denteada. É iterada a seção no centro, excluindo-se as três primeiras e últimas linhas e as duas primeiras e últimas colunas de cada linha selecionada:

C#

```
int[][] jagged =
[
   [0, 1, 2, 3, 4, 5, 6, 7, 8, 9],
   [10,11,12,13,14,15,16,17,18,19],
   [20,21,22,23,24,25,26,27,28,29],
   [30,31,32,33,34,35,36,37,38,39],
   [40,41,42,43,44,45,46,47,48,49],
   [50,51,52,53,54,55,56,57,58,59],
   [60,61,62,63,64,65,66,67,68,69],
   [70,71,72,73,74,75,76,77,78,79],
   [80,81,82,83,84,85,86,87,88,89],
   [90,91,92,93,94,95,96,97,98,99],
];

var selectedRows = jagged[3..^3];

foreach (var row in selectedRows)
{
    var selectedColumns = row[2..^2];
    foreach (var cell in selectedColumns)
    {
        Console.Write($"{cell}, ");
    }
    Console.WriteLine();
}
```

Em todos os casos, o operador de intervalo para Array aloca uma matriz para armazenar os elementos retornados.

Cenários para intervalos e índices

Geralmente, você usará intervalos e índices quando quiser analisar uma parte de uma sequência maior. A nova sintaxe é mais clara porque lê exatamente qual parte da sequência está envolvida. A função local `MovingAverage` usa um Range como seu argumento. O método então enumera apenas esse intervalo ao calcular o mínimo, máximo e média. Experimente o seguinte código em seu projeto:

C#

```
int[] sequence = Sequence(1000);

for(int start = 0; start < sequence.Length; start += 100)
{
    Range r = start..(start+10);
    var (min, max, average) = MovingAverage(sequence, r);
    Console.WriteLine($"From {r.Start} to {r.End}:    \tMin: {min},\tMax:
{max},\tAverage: {average}");
}

for (int start = 0; start < sequence.Length; start += 100)
{
    Range r = ^(start + 10)..^start;
    var (min, max, average) = MovingAverage(sequence, r);
    Console.WriteLine($"From {r.Start} to {r.End}:  \tMin: {min},\tMax:
{max},\tAverage: {average}");
}

(int min, int max, double average) MovingAverage(int[] subSequence, Range
range) =>
    (
        subSequence[range].Min(),
        subSequence[range].Max(),
        subSequence[range].Average()
    );

int[] Sequence(int count) => [..Enumerable.Range(0, count).Select(x => (int)
(Math.Sqrt(x) * 100))];
```

Uma observação sobre índices e matrizes de intervalo

Ao tirar um intervalo de uma matriz, o resultado é uma matriz copiada da matriz inicial, em vez de referenciada. Modificar valores na matriz resultante não altera os valores na matriz inicial.

Por exemplo:

C#

```
var arrayOfFiveItems = new[] { 1, 2, 3, 4, 5 };

var firstThreeItems = arrayOfFiveItems[..3]; // contains 1,2,3
firstThreeItems[0] =  11; // now contains 11,2,3

Console.WriteLine(string.Join(",", firstThreeItems));
Console.WriteLine(string.Join(",", arrayOfFiveItems));

// output:
```

```
// 11,2,3
// 1,2,3,4,5
```

Confira também

- Operadores e expressões de acesso de membro

Colaborar conosco no GitHub

A fonte deste conteúdo pode ser encontrada no GitHub, onde você também pode criar e revisar problemas e solicitações de pull. Para obter mais informações, confira o nosso guia para colaboradores.

.NET

.NET feedback

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

Abrir um problema de documentação

Fornecer comentários sobre o produto

Tutorial: Expressar sua intenção de design com mais clareza com tipos de referência anuláveis e não anuláveis

Artigo • 07/03/2025

Tipos de referência anuláveis complementam tipos de referência da mesma forma que tipos de valor anuláveis complementam tipos de valor. Você declara uma variável como um tipo de referência **anulável** acrescentando um `?` ao tipo. Por exemplo, `string?` representa um `string` anulável. Você pode usar esses novos tipos para expressar mais claramente sua intenção de design: algumas variáveis *sempre devem ter um valor*, outras *podem estar perdendo um valor*.

Neste tutorial, você aprenderá a:

- ✓ Incorporar tipos de referência anuláveis e não anuláveis em seus designs
- ✓ Habilitar verificações de tipo de referência que permitem valor nulo em todo o código.
- ✓ Escreva o código em que o compilador impõe essas decisões de design.
- ✓ Usar o recurso de referência que permite valor nulo em seus próprios designs

Pré-requisitos

Você precisará configurar seu computador para executar o .NET, incluindo o compilador C#. O compilador C# está disponível com do Visual Studio 2022 ou o SDK do .NET .

Este tutorial pressupõe que você esteja familiarizado com C# e .NET, incluindo o Visual Studio ou a CLI do .NET.

Incorpore tipos de referência anuláveis em seus designs

Neste tutorial, você criará uma biblioteca que modela a execução de uma pesquisa. O código usa tipos de referência anuláveis e tipos de referência não anuláveis para representar os conceitos do mundo real. As perguntas da pesquisa nunca podem ser nulas. Um entrevistado pode preferir não responder a uma pergunta. As respostas podem ser `null` nesse caso.

O código que você escreverá para este exemplo expressa essa intenção e o compilador impõe essa intenção.

Criar o aplicativo e habilitar tipos de referência anuláveis

Crie um novo aplicativo de console no Visual Studio ou na linha de comando usando `dotnet new console`. Nomeie o aplicativo `NullableIntroduction`. Depois de criar o aplicativo, você precisará especificar que todo o projeto é compilado em um **contexto de anotação anulável** habilitado. Abra o arquivo *.csproj* e adicione um elemento `Nullable` ao elemento `PropertyGroup`. Defina seu valor como `enable`. Você precisa aceitar o recurso de **tipos de referência que permitem valor nulo** em projetos anteriores ao C# 11. Isso ocorre porque, depois que o recurso é ativado, as declarações de variável de referência existentes se tornam **tipos de referência não anuláveis**. Embora essa decisão ajude a encontrar problemas em que o código existente pode não ter verificações nulas adequadas, ela pode não refletir com precisão sua intenção de design original:

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

Antes do .NET 6, novos projetos não incluem o elemento `Nullable`. A partir do .NET 6, novos projetos incluem o elemento `<Nullable>enable</Nullable>` no arquivo de projeto.

Projetar os tipos para o aplicativo

Este aplicativo de pesquisa requer a criação de várias classes:

- Uma classe que modela a lista de perguntas.
- Uma classe que modela uma lista de pessoas contatadas para a pesquisa.
- Uma classe que modela as respostas de uma pessoa que fez a pesquisa.

Esses tipos usarão tipos de referência anuláveis e não anuláveis para expressar quais membros são necessários e quais membros são opcionais. Tipos de referência anuláveis comunicam claramente essa intenção de design:

- As perguntas que fazem parte da pesquisa nunca podem ser nulas: não faz sentido fazer uma pergunta vazia.
- Os entrevistados nunca podem ser nulos. Você desejará acompanhar as pessoas que você contatou, até mesmo os entrevistados que se recusaram a participar.

- Qualquer resposta a uma pergunta pode ser nula. Os entrevistados podem se recusar a responder algumas ou todas as perguntas.

Se você programou em C#, pode estar tão acostumado a tipos de referência que permitem valores `null` que talvez não tenha percebido outras oportunidades para declarar variáveis como não anuláveis.

- A coleção de perguntas deve ser não anulável.
- A coleção de entrevistados deve ser não anulável.

Ao escrever o código, você verá que um tipo de referência não anulável como o padrão para referências evita erros comuns que podem levar a NullReferenceExceptions. Uma lição deste tutorial é que você tomou decisões sobre quais variáveis poderiam ou não ser `null`. A linguagem não forneceu sintaxe para expressar essas decisões. Agora ele já fornece.

O aplicativo que você criará executa as seguintes etapas:

1. Cria uma pesquisa e adiciona perguntas a ela.
2. Cria um conjunto pseudo-aleatório de entrevistados para a pesquisa.
3. Entre em contato com os entrevistados até que o tamanho da pesquisa preenchida atinja o número da meta.
4. Escreve estatísticas importantes sobre as respostas da pesquisa.

Criar a pesquisa com tipos de referência anuláveis e não anuláveis

O primeiro código que você escreverá cria a pesquisa. Você escreverá classes para modelar uma pergunta da pesquisa e uma execução da pesquisa. Sua pesquisa tem três tipos de perguntas, distinguidas pelo formato da resposta: respostas sim/não, respostas numéricas e respostas de texto. Crie uma classe `public SurveyQuestion`:

C#

```
namespace NullableIntroduction
{
    public class SurveyQuestion
    {
    }
}
```

O compilador interpreta cada declaração de variável de tipo de referência como um tipo de referência **não anulável** para o código em um contexto de anotação anulável

habilitado. Você pode ver seu primeiro aviso adicionando propriedades para o texto da pergunta e o tipo de pergunta, conforme mostrado no código a seguir:

C#

```
namespace NullableIntroduction
{
    public enum QuestionType
    {
        YesNo,
        Number,
        Text
    }

    public class SurveyQuestion
    {
        public string QuestionText { get; }
        public QuestionType TypeOfQuestion { get; }
    }
}
```

Como você não inicializou `QuestionText`, o compilador emite um aviso de que uma propriedade não anulável não foi inicializada. Seu design exige que o texto da pergunta seja não nulo, portanto, você adiciona um construtor para inicializá-lo e o valor `QuestionType` também. A definição de classe concluída se parece com o seguinte código:

C#

```
namespace NullableIntroduction;

public enum QuestionType
{
    YesNo,
    Number,
    Text
}

public class SurveyQuestion
{
    public string QuestionText { get; }
    public QuestionType TypeOfQuestion { get; }

    public SurveyQuestion(QuestionType typeOfQuestion, string text) =>
        (TypeOfQuestion, QuestionText) = (typeOfQuestion, text);
}
```

A adição do construtor removerá o aviso. O argumento construtor também é um tipo de referência não anulável, portanto, o compilador não emite avisos.

Em seguida, crie uma classe de `public` chamada `SurveyRun`. Essa classe contém uma lista de `SurveyQuestion` objetos e métodos para adicionar perguntas à pesquisa, conforme mostrado no código a seguir:

C#

```
using System.Collections.Generic;

namespace NullableIntroduction
{
    public class SurveyRun
    {
        private List<SurveyQuestion> surveyQuestions = new
List<SurveyQuestion>();

        public void AddQuestion(QuestionType type, string question) =>
            AddQuestion(new SurveyQuestion(type, question));
        public void AddQuestion(SurveyQuestion surveyQuestion) =>
surveyQuestions.Add(surveyQuestion);
    }
}
```

Como antes, você deve inicializar o objeto de lista para um valor não nulo ou o compilador emite um aviso. Não há verificações nulas na segunda sobrecarga de `AddQuestion` porque elas não são necessárias: você declarou essa variável como não anulável. Seu valor não pode ser `null`.

Alterne para *Program.cs* em seu editor e substitua o conteúdo de `Main` pelas seguintes linhas de código:

C#

```
var surveyRun = new SurveyRun();
surveyRun.AddQuestion(QuestionType.YesNo, "Has your code ever thrown a
NullReferenceException?");
surveyRun.AddQuestion(new SurveyQuestion(QuestionType.Number, "How many
times (to the nearest 100) has that happened?"));
surveyRun.AddQuestion(QuestionType.Text, "What is your favorite color?");
```

Como o projeto inteiro está em um contexto de anotação anulável habilitado, você receberá avisos quando passar `null` para qualquer método que espera um tipo de referência não anulável. Experimente adicionando a seguinte linha a `Main`:

C#

```
surveyRun.AddQuestion(QuestionType.Text, default);
```

Criar respondentes e obter respostas para a pesquisa

Em seguida, escreva o código que gera respostas para a pesquisa. Esse processo envolve várias tarefas pequenas:

1. Crie um método que gere objetos entrevistados. Isso representa as pessoas solicitadas a preencher a pesquisa.
2. Crie lógica para simular fazer as perguntas a um entrevistado e coletar respostas ou observando que um entrevistado não respondeu.
3. Repita até que respondentes suficientes tenham respondido à pesquisa.

Você precisará de uma classe para representar uma resposta de pesquisa, então adicione-a agora. Habilitar o suporte para tipos que permitem valor nulo. Adicione uma propriedade `Id` e um construtor que a inicialize, conforme mostrado no seguinte código:

C#

```
namespace NullableIntroduction
{
    public class SurveyResponse
    {
        public int Id { get; }

        public SurveyResponse(int id) => Id = id;
    }
}
```

Em seguida, adicione um método `static` para criar novos participantes gerando uma ID aleatória:

C#

```
private static readonly Random randomGenerator = new Random();
public static SurveyResponse GetRandomId() => new
SurveyResponse(randomGenerator.Next());
```

A principal responsabilidade dessa classe é gerar as respostas de um participante às perguntas da pesquisa. Essa responsabilidade tem algumas etapas:

1. Peça participação na pesquisa. Se a pessoa não consentir, retorne uma resposta ausente (ou nula).

2. Faça cada pergunta e grave a resposta. Cada resposta também pode estar ausente (ou nula).

Adicione o seguinte código à classe `SurveyResponse`:

C#

```
private Dictionary<int, string>? surveyResponses;
public bool AnswerSurvey(IEnumerable<SurveyQuestion> questions)
{
    if (ConsentToSurvey())
    {
        surveyResponses = new Dictionary<int, string>();
        int index = 0;
        foreach (var question in questions)
        {
            var answer = GenerateAnswer(question);
            if (answer != null)
            {
                surveyResponses.Add(index, answer);
            }
            index++;
        }
    }
    return surveyResponses != null;
}

private bool ConsentToSurvey() => randomGenerator.Next(0, 2) == 1;

private string? GenerateAnswer(SurveyQuestion question)
{
    switch (question.TypeOfQuestion)
    {
        case QuestionType.YesNo:
            int n = randomGenerator.Next(-1, 2);
            return (n == -1) ? default : (n == 0) ? "No" : "Yes";
        case QuestionType.Number:
            n = randomGenerator.Next(-30, 101);
            return (n < 0) ? default : n.ToString();
        case QuestionType.Text:
        default:
            switch (randomGenerator.Next(0, 5))
            {
                case 0:
                    return default;
                case 1:
                    return "Red";
                case 2:
                    return "Green";
                case 3:
                    return "Blue";
            }
            return "Red. No, Green. Wait.. Blue... AAARGGGGGHHH!";
```

```
    }
}
```

O armazenamento das respostas da pesquisa é um `Dictionary<int, string>?`, indicando que ele pode ser um valor nulo. Você está usando o novo recurso de linguagem para declarar sua intenção de design, tanto para o compilador quanto para qualquer pessoa que leia seu código mais tarde. Se você desreferenciar `surveyResponses` sem verificar o valor do `null` primeiro, receberá um aviso do compilador. Você não recebe um aviso no método `AnswerSurvey` porque o compilador pode determinar que a variável `surveyResponses` foi definida como um valor não nulo acima.

Usar `null` para respostas ausentes realça um ponto-chave para trabalhar com tipos de referência anuláveis: sua meta não é remover todos os valores `null` do seu programa. Em vez disso, seu objetivo é garantir que o código que você escreve expresse a intenção do seu design. Valores ausentes são um conceito necessário para expressar em seu código. O valor `null` é uma maneira clara de expressar esses valores ausentes. Tentar remover todos os valores `null` leva apenas a definir alguma outra maneira de expressar esses valores ausentes sem `null`.

Em seguida, você precisa escrever o método `PerformSurvey` na classe `SurveyRun`. Adicione o seguinte código na classe `SurveyRun`:

C#

```
private List<SurveyResponse>? respondents;
public void PerformSurvey(int numberOfRespondents)
{
    int respondentsConsenting = 0;
    respondents = new List<SurveyResponse>();
    while (respondentsConsenting < numberOfRespondents)
    {
        var respondent = SurveyResponse.GetRandomId();
        if (respondent.AnswerSurvey(surveyQuestions))
            respondentsConsenting++;
        respondents.Add(respondent);
    }
}
```

Aqui novamente, sua escolha de um `List<SurveyResponse>?` anulável indica que a resposta pode ser nula. Isso indica que a pesquisa ainda não foi dada a nenhum entrevistado. Observe que os entrevistados são adicionados até que um número suficiente tenha consentido.

A última etapa para executar a pesquisa é adicionar uma chamada para executar a pesquisa no final do método `Main`:

```
C#

surveyRun.PerformSurvey(50);
```

Examinar as respostas da pesquisa

A última etapa é exibir os resultados da pesquisa. Você adicionará código a muitas das classes que você escreveu. Esse código demonstra o valor da distinção de tipos de referência anuláveis e não anuláveis. Comece adicionando os dois membros com corpo de expressão à classe `SurveyResponse`:

```
C#

public bool AnsweredSurvey => surveyResponses != null;
public string Answer(int index) => surveyResponses?.GetValueOrDefault(index)
?? "No answer";
```

Como `surveyResponses` é um tipo de referência anulável, são necessárias verificações de nulidade antes de desreferenciá-la. O método `Answer` retorna uma string não anulável, portanto, temos que cobrir o caso de uma resposta ausente usando o operador de coalescência nula.

Em seguida, adicione esses três membros com corpo de expressão à classe `SurveyRun`:

```
C#

public IEnumerable<SurveyResponse> AllParticipants => (respondents ??
Enumerable.Empty<SurveyResponse>());
public ICollection<SurveyQuestion> Questions => surveyQuestions;
public SurveyQuestion GetQuestion(int index) => surveyQuestions[index];
```

O membro `AllParticipants` deve levar em conta que a variável `respondents` pode ser nula, mas o valor retornado não pode ser nulo. Se você alterar essa expressão removendo o `??` e a sequência vazia a seguir, o compilador avisará que o método pode retornar `null` e sua assinatura de retorno retornará um tipo não anulável.

Por fim, adicione o seguinte loop na parte inferior do método `Main`:

```
C#
```

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

Você não precisa de nenhuma verificação de `null` neste código porque criou as interfaces subjacentes para que todas elas retornem tipos de referência não anuláveis.

Obter o código

Obtenha o código do tutorial concluído em nosso repositório de amostras na pasta csharp/NullableIntroduction.

Experimente alterando as declarações de tipo entre tipos de referência anuláveis e não anuláveis. Veja como isso gera avisos diferentes para garantir que um `null` não será acidentalmente cancelado.

Próximas etapas

Saiba como usar o tipo de referência anulável ao usar o Entity Framework:

Conceitos Básicos do Entity Framework Core: trabalhando com tipos de referência anuláveis

Usar a correspondência de padrões para criar o comportamento de classe para um código melhor

Artigo • 30/12/2024

Os recursos de correspondência de padrões em C# fornecem sintaxe para expressar seus algoritmos. Você pode usar essas técnicas para implementar o comportamento em suas classes. Você pode combinar o design de classe orientada a objetos com uma implementação orientada a dados para fornecer código conciso durante a modelagem de objetos do mundo real.

Neste tutorial, você aprenderá a:

- ✓ Expresse suas classes orientadas a objetos usando padrões de dados.
- ✓ Implemente esses padrões usando os recursos de correspondência de padrões do C#.
- ✓ Aproveite o diagnóstico do compilador para validar sua implementação.

Pré-requisitos

Você precisa configurar seu computador para executar o .NET. Baixe o Visual Studio 2022 ou o SDK do .NET.

Criar uma simulação de um bloqueio de canal

Neste tutorial, você criará uma classe C# que simula uma eclusa de canal. Resumidamente, um bloqueio de canal é um dispositivo que levanta e baixa barcos enquanto viajam entre dois trechos de água em níveis diferentes. Um bloqueio tem dois portões e algum mecanismo para alterar o nível da água.

Em sua operação normal, um barco entra em um dos portões enquanto o nível da água na fechadura corresponde ao nível da água no lado em que o barco entra. Uma vez na eclusa, o nível da água é ajustado para corresponder ao nível da água de onde o barco sairá da eclusa. Depois que o nível da água corresponder a esse lado, o portão no lado de saída será aberto. As medidas de segurança garantem que um operador não possa criar uma situação perigosa no canal. O nível da água só pode ser alterado quando ambos os portões estão fechados. No máximo, um portão pode ser aberto. Para abrir um portão, o nível da água na fechadura deve corresponder ao nível da água fora do portão que está sendo aberto.

Você pode criar uma classe C# para modelar esse comportamento. Uma classe `CanalLock` ofereceria suporte a comandos para abrir ou fechar qualquer porta. Teria outros comandos para levantar ou baixar a água. A classe também deve dar suporte a propriedades para ler o estado atual dos portões e do nível da água. Os métodos implementam as medidas de segurança.

Definir uma classe

Você cria um aplicativo de console para testar sua classe de `CanalLock`. Crie um novo projeto de console para .NET 5 usando o Visual Studio ou a CLI do .NET. Em seguida, adicione uma nova classe e nomeie-a `CanalLock`. Em seguida, crie sua API pública, mas deixe os métodos não implementados:

C#

```
public enum WaterLevel
{
    Low,
    High
}
public class CanalLock
{
    // Query canal lock state:
    public WaterLevel CanalLockWaterLevel { get; private set; } =
WaterLevel.Low;
    public bool HighWaterGateOpen { get; private set; } = false;
    public bool LowWaterGateOpen { get; private set; } = false;

    // Change the upper gate.
    public void SetHighGate(bool open)
    {
        throw new NotImplementedException();
    }

    // Change the lower gate.
    public void SetLowGate(bool open)
    {
        throw new NotImplementedException();
    }

    // Change water level.
    public void SetWaterLevel(WaterLevel newLevel)
    {
        throw new NotImplementedException();
    }

    public override string ToString() =>
        $"The lower gate is {(LowWaterGateOpen ? "Open" : "Closed")}. " +
        $"The upper gate is {(HighWaterGateOpen ? "Open" : "Closed")}. " +
```

```
        $"The water level is {CanalLockWaterLevel}.";
}
```

O código anterior inicializa o objeto para que ambos os portões sejam fechados e o nível da água seja baixo. Em seguida, escreva o seguinte código de teste em seu método `Main` para guiá-lo à medida que você cria uma primeira implementação da classe:

C#

```
// Create a new canal lock:
var canalGate = new CanalLock();

// State should be doors closed, water level low:
Console.WriteLine(canalGate);

canalGate.SetLowGate(open: true);
Console.WriteLine($"Open the lower gate:  {canalGate}");

Console.WriteLine("Boat enters lock from lower gate");

canalGate.SetLowGate(open: false);
Console.WriteLine($"Close the lower gate:  {canalGate}");

canalGate.SetWaterLevel(WaterLevel.High);
Console.WriteLine($"Raise the water level: {canalGate}");

canalGate.SetHighGate(open: true);
Console.WriteLine($"Open the higher gate:  {canalGate}");

Console.WriteLine("Boat exits lock at upper gate");
Console.WriteLine("Boat enters lock from upper gate");

canalGate.SetHighGate(open: false);
Console.WriteLine($"Close the higher gate: {canalGate}");

canalGate.SetWaterLevel(WaterLevel.Low);
Console.WriteLine($"Lower the water level: {canalGate}");

canalGate.SetLowGate(open: true);
Console.WriteLine($"Open the lower gate:  {canalGate}");

Console.WriteLine("Boat exits lock at upper gate");

canalGate.SetLowGate(open: false);
Console.WriteLine($"Close the lower gate:  {canalGate}");
```

Em seguida, adicione uma primeira implementação de cada método na classe `CanalLock`. O código a seguir implementa os métodos da classe sem preocupação com as regras de segurança. Você adicionará testes de segurança mais tarde:

C#

```
// Change the upper gate.
public void SetHighGate(bool open)
{
    HighWaterGateOpen = open;
}

// Change the lower gate.
public void SetLowGate(bool open)
{
    LowWaterGateOpen = open;
}

// Change water level.
public void SetWaterLevel(WaterLevel newLevel)
{
    CanalLockWaterLevel = newLevel;
}
```

Os testes que você escreveu até agora foram aprovados. Você implementou o básico. Agora, escreva um teste para a primeira condição de falha. No final dos testes anteriores, ambos os portões estão fechados, e o nível da água está definido como baixo. Adicione um teste para tentar abrir a porta superior:

C#

```
Console.WriteLine("=============================================");
Console.WriteLine("     Test invalid commands");
// Open "wrong" gate (2 tests)
try
{
    canalGate = new CanalLock();
    canalGate.SetHighGate(open: true);
}
catch (InvalidOperationException)
{
    Console.WriteLine("Invalid operation: Can't open the high gate. Water is
low.");
}
Console.WriteLine($"Try to open upper gate: {canalGate}");
```

Esse teste falha porque a porta é aberta. Como primeira implementação, você pode corrigi-la com o seguinte código:

C#

```
// Change the upper gate.
public void SetHighGate(bool open)
{
```

```
    if (open && (CanalLockWaterLevel == WaterLevel.High))
        HighWaterGateOpen = true;
    else if (open && (CanalLockWaterLevel == WaterLevel.Low))
        throw new InvalidOperationException("Cannot open high gate when the
water is low");
}
```

Os testes são aprovados. Mas, à medida que você adiciona mais testes, adiciona mais cláusulas `if` e testa propriedades diferentes. Em breve, esses métodos ficam muito complicados à medida que você adiciona mais condicionais.

Implementar os comandos com padrões

Uma maneira melhor é usar padrões para determinar se o objeto está em um estado válido para executar um comando. Você pode expressar se um comando é permitido como uma função de três variáveis: o estado da porta, o nível da água e a nova configuração:

Expandir a tabela

Nova configuração	Estado do portão	Nível da Água	Resultado
Fechado	Fechado	Alto	Fechado
Fechado	Fechado	Baixo	Fechado
Fechado	Abrir	Alto	Fechado
Fechado	~~Abrir~~	~~Baixo~~	Fechado
Abrir	Fechado	Alto	Abrir
Abrir	Fechado	Baixo	Fechado (Erro)
Abrir	Abrir	Alto	Abrir
~~Abrir~~	~~Abrir~~	~~Baixo~~	~~Fechado (Erro)~~

As quartas e últimas linhas da tabela atingiram o texto porque são inválidas. O código que você está adicionando agora deve garantir que o portão de água alta nunca seja aberto quando a água estiver baixa. Esses estados podem ser codificados como uma única expressão de comutador (lembre-se de que `false` indica "Fechado"):

```
C#

HighWaterGateOpen = (open, HighWaterGateOpen, CanalLockWaterLevel) switch
{
```

```
    (false, false, WaterLevel.High) => false,
    (false, false, WaterLevel.Low) => false,
    (false, true, WaterLevel.High) => false,
    (false, true, WaterLevel.Low) => false, // should never happen
    (true, false, WaterLevel.High) => true,
    (true, false, WaterLevel.Low) => throw new
InvalidOperationException("Cannot open high gate when the water is low"),
    (true, true, WaterLevel.High) => true,
    (true, true, WaterLevel.Low) => false, // should never happen
};
```

Experimente esta versão. Os testes passam, validando o código. A tabela completa mostra as possíveis combinações de entradas e resultados. Isso significa que você e outros desenvolvedores podem examinar rapidamente a tabela e ver que você cobriu todas as entradas possíveis. Ainda mais fácil, o compilador também pode ajudar. Depois de adicionar o código anterior, você poderá ver que o compilador gera um aviso: *CS8524* indica que a expressão de comutador não abrange todas as entradas possíveis. O motivo desse aviso é que uma das entradas é um tipo `enum`. O compilador interpreta "todas as entradas possíveis" como todas as entradas do tipo subjacente, normalmente um `int`. Essa expressão `switch` verifica apenas os valores declarados no `enum`. Para remover o aviso, você pode adicionar um padrão de descarte catch-all para o último braço da expressão. Essa condição gera uma exceção, pois indica uma entrada inválida:

```
C#

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

O braço de comutador anterior deve ser o último na expressão `switch` porque corresponde a todas as entradas. Experimente movendo-o anteriormente na ordem. Isso causa um erro do compilador *CS8510* para código inacessível em um padrão. A estrutura natural de expressões de comutador permite que o compilador gere erros e avisos para possíveis erros. A "rede de segurança" do compilador facilita a criação de código correto em menos iterações e a liberdade de combinar armas com curingas. O compilador emitirá erros se a combinação resultar em braços inacessíveis que você não esperava e avisos se você remover um braço necessário.

A primeira alteração é combinar todos os braços em que o comando deve fechar o portão; isso é sempre permitido. Adicione o seguinte código como o primeiro braço na expressão switch:

```
C#

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

Depois de adicionar o braço de comutador anterior, você receberá quatro erros do compilador, um em cada um dos braços em que o comando é `false`. Esses braços já estão cobertos pelo braço recém-adicionado. Você pode remover com segurança essas quatro linhas. Você pretendia que este novo braço do interruptor substituísse essas condições.

Em seguida, você pode simplificar os quatro braços em que o comando é abrir o portão. Em ambos os casos em que o nível da água é alto, o portão pode ser aberto. (Em um deles, ele já está aberto.) Um caso em que o nível da água é baixo gera uma exceção, e o outro não deve acontecer. Deve ser seguro lançar a mesma exceção se o bloqueio de água já estiver em um estado inválido. Você pode fazer as seguintes simplificações para esses braços:

C#

```
(true, _, WaterLevel.High) => true,
(true, false, WaterLevel.Low) => throw new InvalidOperationException("Cannot
open high gate when the water is low"),
_ => throw new InvalidOperationException("Invalid internal state"),
```

Execute seus testes novamente e eles passam. Esta é a versão final do método `SetHighGate`:

C#

```
// Change the upper gate.
public void SetHighGate(bool open)
{
    HighWaterGateOpen = (open, HighWaterGateOpen, CanalLockWaterLevel)
switch
    {
        (false, _,     _)               => false,
        (true, _,      WaterLevel.High) => true,
        (true, false, WaterLevel.Low)  => throw new
InvalidOperationException("Cannot open high gate when the water is low"),
        _                               => throw new
InvalidOperationException("Invalid internal state"),
    };
}
```

Implementar padrões por conta própria

Agora que você já viu a técnica, complete os métodos `SetLowGate` e `SetWaterLevel` por conta própria. Comece adicionando o seguinte código para testar operações inválidas nesses métodos:

C#

```
Console.WriteLine();
Console.WriteLine();
try
{
    canalGate = new CanalLock();
    canalGate.SetWaterLevel(WaterLevel.High);
    canalGate.SetLowGate(open: true);
}
catch (InvalidOperationException)
{
    Console.WriteLine("invalid operation: Can't open the lower gate. Water
is high.");
}
Console.WriteLine($"Try to open lower gate: {canalGate}");
// change water level with gate open (2 tests)
Console.WriteLine();
Console.WriteLine();
try
{
    canalGate = new CanalLock();
    canalGate.SetLowGate(open: true);
    canalGate.SetWaterLevel(WaterLevel.High);
}
catch (InvalidOperationException)
{
    Console.WriteLine("invalid operation: Can't raise water when the lower
gate is open.");
}
Console.WriteLine($"Try to raise water with lower gate open: {canalGate}");
Console.WriteLine();
Console.WriteLine();
try
{
    canalGate = new CanalLock();
    canalGate.SetWaterLevel(WaterLevel.High);
    canalGate.SetHighGate(open: true);
    canalGate.SetWaterLevel(WaterLevel.Low);
}
catch (InvalidOperationException)
{
    Console.WriteLine("invalid operation: Can't lower water when the high
gate is open.");
}
Console.WriteLine($"Try to lower water with high gate open: {canalGate}");
```

Execute seu aplicativo novamente. Você pode ver que os novos testes falham e o bloqueio do canal entra em um estado inválido. Tente implementar os métodos restantes por conta própria. O método para definir a porta inferior deve ser semelhante ao método para definir a porta superior. O método que altera o nível da água tem verificações diferentes, mas deve seguir uma estrutura semelhante. Você pode achar útil

usar o mesmo processo para o método que define o nível da água. Comece com todas as quatro entradas: o estado de ambos os portões, o estado atual do nível da água e o novo nível de água solicitado. A expressão switch deve começar com:

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

Você terá 16 braços de comutador totais para preencher. Em seguida, teste e simplifique.

Você fez métodos como este?

```C#
// Change the lower gate.
public void SetLowGate(bool open)
{
    LowWaterGateOpen = (open, LowWaterGateOpen, CanalLockWaterLevel) switch
    {
        (false, _, _) => false,
        (true, _, WaterLevel.Low) => true,
        (true, false, WaterLevel.High) => throw new
InvalidOperationException("Cannot open low gate when the water is high"),
        _ => throw new InvalidOperationException("Invalid internal state"),
    };
}

// Change water level.
public void SetWaterLevel(WaterLevel newLevel)
{
    CanalLockWaterLevel = (newLevel, CanalLockWaterLevel, LowWaterGateOpen,
HighWaterGateOpen) switch
    {
        (WaterLevel.Low, WaterLevel.Low, true, false) => WaterLevel.Low,
        (WaterLevel.High, WaterLevel.High, false, true) => WaterLevel.High,
        (WaterLevel.Low, _, false, false) => WaterLevel.Low,
        (WaterLevel.High, _, false, false) => WaterLevel.High,
        (WaterLevel.Low, WaterLevel.High, false, true) => throw new
InvalidOperationException("Cannot lower water when the high gate is open"),
        (WaterLevel.High, WaterLevel.Low, true, false) => throw new
InvalidOperationException("Cannot raise water when the low gate is open"),
        _ => throw new InvalidOperationException("Invalid internal state"),
    };
}
```

Os testes devem ser aprovados e o bloqueio do canal deve operar com segurança.

Resumo

Neste tutorial, você aprendeu a usar a correspondência de padrões para verificar o estado interno de um objeto antes de aplicar as alterações a esse estado. Você pode verificar combinações de propriedades. Depois de criar tabelas para qualquer uma dessas transições, você testa seu código e, em seguida, simplifica a legibilidade e a manutenção. Essas refatorações iniciais podem sugerir refatorações adicionais que validam o estado interno ou gerenciam outras alterações de API. Este tutorial combinou classes e objetos com uma abordagem mais orientada a dados e baseada em padrões para implementar essas classes.

Interpolação de cadeia de caracteres em C#

Artigo • 06/05/2025

Este tutorial mostra como usar a interpolação de cadeia de caracteres para formatar e incluir resultados de expressão em uma cadeia de caracteres de resultado. Os exemplos pressupõem que você esteja familiarizado com conceitos básicos de C# e formatação de tipo .NET. Para obter mais informações sobre tipos de formatação no .NET, consulte Tipos de formatação no .NET.

Introdução

Para identificar uma literal de cadeia de caracteres como uma cadeia de caracteres interpolada, preceda-o com o símbolo `$`. Você pode inserir qualquer expressão C# válida que retorne um valor em uma cadeia de caracteres interpolada. No exemplo a seguir, assim que uma expressão é avaliada, seu resultado é convertido em uma cadeia de caracteres e incluído em uma cadeia de caracteres de resultado:

C#

```
double a = 3;
double b = 4;
Console.WriteLine($"Area of the right triangle with legs of {a} and {b} is {0.5 *
a * b}");
Console.WriteLine($"Length of the hypotenuse of the right triangle with legs of
{a} and {b} is {CalculateHypotenuse(a, b)}");
double CalculateHypotenuse(double leg1, double leg2) => Math.Sqrt(leg1 * leg1 +
leg2 * leg2);
// Output:
// Area of the right triangle with legs of 3 and 4 is 6
// Length of the hypotenuse of the right triangle with legs of 3 and 4 is 5
```

Como mostra o exemplo, você inclui uma expressão em uma cadeia de caracteres interpolada colocando-a com chaves:

C#

```
{<interpolationExpression>}
```

Cadeias de caracteres interpoladas dão suporte a todos os recursos do recurso de formatação de composição de cadeia de caracteres . Isso os torna uma alternativa mais legível ao uso do String.Format método. Cada cadeia de caracteres interpolada deve ter:

- Um literal de cadeia de caracteres que começa com o `$` caractere antes de seu caractere de aspas de abertura. Não pode haver espaços entre o `$` símbolo e o caractere de aspas.

- Uma ou mais *expressões de interpolação*. Você indica uma expressão de interpolação com uma chave de abertura e fechamento (`{` e `}`). Você pode colocar qualquer expressão C# que retorna um valor (incluindo `null`) dentro das chaves.

O C# avalia a expressão entre os caracteres e `}` as `{` seguintes regras:

- Se a expressão de interpolação for avaliada como `null`, uma cadeia de caracteres vazia ("" ou String.Empty) será usada.
- Se a expressão de interpolação não for avaliada `null`, normalmente o `ToString` método do tipo de resultado será chamado.

Como especificar uma cadeia de caracteres de formato para uma expressão de interpolação

Para especificar uma cadeia de caracteres de formato que seja compatível com o tipo do resultado da expressão, siga a expressão de interpolação com um dois-pontos (":") seguido pela cadeia de formato.

C#

```
{<interpolationExpression>:<formatString>}
```

O exemplo a seguir mostra como especificar cadeias de caracteres de formato padrão e personalizado para expressões que produzem resultados numéricos e de data e hora:

C#

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

Para obter mais informações, consulte a seção Componente de cadeia de caracteres de formato do artigo Formatação composta.

Como controlar a largura do campo e o alinhamento da expressão de interpolação formatada

Para especificar a largura mínima do campo e o alinhamento do resultado da expressão formatada, siga a expressão de interpolação com uma vírgula (") e a expressão constante:

```
C#

{<interpolationExpression>,<width>}
```

O exemplo de código a seguir usa a largura mínima do campo para criar uma saída tabular:

```
C#

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}|");
}
// Output:
// Author and Title List
//
// |Author                   |Title                          |
// |Doyle, Arthur Conan      |Hound of the Baskervilles, The |
// |London, Jack             |         Call of the Wild, The |
// |Shakespeare, William     |                  Tempest, The |
```

Se o valor da *largura* for positivo, o resultado da expressão formatada será alinhado à direita; se negativo, é alinhado à esquerda. Remova os `-` sinais antes do especificador de largura e execute o exemplo novamente para ver os resultados.

Se você precisar especificar a largura e uma cadeia de caracteres de formato, comece com o componente de largura:

```
C#

{<interpolationExpression>,<width>:<formatString>}
```

O exemplo a seguir mostra como especificar largura e alinhamento e usa caracteres de pipe ("|") para delimitar campos de texto:

C#

```
const int NameAlignment = -9;
const int ValueAlignment = 7;
double a = 3;
double b = 4;
Console.WriteLine($"Three classical Pythagorean means of {a} and {b}:");
Console.WriteLine($"|{"Arithmetic",NameAlignment}|{0.5 * (a +
b),ValueAlignment:F3}|");
Console.WriteLine($"|{"Geometric",NameAlignment}|{Math.Sqrt(a *
b),ValueAlignment:F3}|");
Console.WriteLine($"|{"Harmonic",NameAlignment}|{2 / (1 / a + 1 /
b),ValueAlignment:F3}|");
// Output:
// Three classical Pythagorean means of 3 and 4:
// |Arithmetic|  3.500|
// |Geometric|  3.464|
// |Harmonic |  3.429|
```

Como mostra a saída de exemplo, se o comprimento do resultado da expressão formatada exceder a largura do campo especificada, o valor de *largura* será ignorado.

Para obter mais informações, consulte a seção Componente de Largura do artigo de formatação composta.

Como usar sequências de escape em uma cadeia de caracteres interpolada

Cadeias de caracteres interpoladas dão suporte a todas as sequências de escape que podem ser usadas em literais de cadeia de caracteres comuns. Para obter mais informações, consulte Sequências de escape de cadeia de caracteres.

Para interpretar sequências de escape literalmente, use um literal de cadeia de caracteres textual. Uma cadeia de caracteres verbatim interpolada começa com os caracteres `$` e `@`. Você pode usar `$` e `@` em qualquer ordem: ambas `$@"..."` e `@$"..."` são cadeias de caracteres verbatim interpoladas válidas.

Para incluir uma chave, "{" ou "}", em uma cadeia de caracteres de resultado, use duas chaves, "{{" ou "}}". Para obter mais informações, consulte a seção Chaves de escape do artigo Formatação composta.

O seguinte exemplo mostra como incluir chaves em uma cadeia de caracteres de resultado e construir uma cadeia de caracteres interpolada textual:

C#

```
var xs = new int[] { 1, 2, 7, 9 };
var ys = new int[] { 7, 9, 12 };
Console.WriteLine($"Find the intersection of the {{{string.Join(", ",xs)}}} and
{{{string.Join(", ",ys)}}} sets.");
// Output:
// Find the intersection of the {1, 2, 7, 9} and {7, 9, 12} sets.

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

A partir do C# 11, você pode usar literais de cadeia de caracteres bruta e interpolada.

Como usar um operador `?:` condicional ternário em uma expressão de interpolação

Como os dois-pontos (:) têm um significado especial em um item com uma expressão de interpolação, para usar um operador condicional em uma expressão, coloque-a entre parênteses, como mostra o seguinte exemplo:

C#

```
var rand = new Random();
for (int i = 0; i < 7; i++)
{
    Console.WriteLine($"Coin flip: {(rand.NextDouble() < 0.5 ? "heads" :
"tails")}");
}
```

Como criar uma cadeia de caracteres de resultado específica a uma cultura com a interpolação de cadeia de caracteres

Por padrão, uma cadeia de caracteres interpolada usa a cultura atual definida pela CultureInfo.CurrentCulture propriedade para todas as operações de formatação.

A partir do .NET 6, você pode usar o método String.Create(IFormatProvider, DefaultInterpolatedStringHandler) para resolve uma cadeia de caracteres interpolada para uma

cadeia de caracteres de resultado específica da cultura, como mostra o exemplo a seguir:

C#

```
var cultures = new System.Globalization.CultureInfo[]
{
    System.Globalization.CultureInfo.GetCultureInfo("en-US"),
    System.Globalization.CultureInfo.GetCultureInfo("en-GB"),
    System.Globalization.CultureInfo.GetCultureInfo("nl-NL"),
    System.Globalization.CultureInfo.InvariantCulture
};
var date = DateTime.Now;
var number = 31_415_926.536;
foreach (var culture in cultures)
{
    var cultureSpecificMessage = string.Create(culture, $"{date,23}
{number,20:N3}");
    Console.WriteLine($"{culture.Name,-10}{cultureSpecificMessage}");
}
// Output is similar to:
// en-US       8/27/2023 12:35:31 PM      31,415,926.536
// en-GB         27/08/2023 12:35:31      31,415,926.536
// nl-NL         27-08-2023 12:35:31      31.415.926,536
//               08/27/2023 12:35:31      31,415,926.536
```

Em versões anteriores do .NET, use a conversão implícita de uma cadeia de caracteres interpolada em uma instância System.FormattableString e chame seu método ToString(IFormatProvider) para criar uma cadeia de caracteres de resultado específica da cultura. O seguinte exemplo mostra como fazer isso:

C#

```
var cultures = new System.Globalization.CultureInfo[]
{
    System.Globalization.CultureInfo.GetCultureInfo("en-US"),
    System.Globalization.CultureInfo.GetCultureInfo("en-GB"),
    System.Globalization.CultureInfo.GetCultureInfo("nl-NL"),
    System.Globalization.CultureInfo.InvariantCulture
};
var date = DateTime.Now;
var number = 31_415_926.536;
FormattableString message = $"{date,23}{number,20:N3}";
foreach (var culture in cultures)
{
    var cultureSpecificMessage = message.ToString(culture);
    Console.WriteLine($"{culture.Name,-10}{cultureSpecificMessage}");
}
// Output is similar to:
// en-US       8/27/2023 12:35:31 PM      31,415,926.536
// en-GB         27/08/2023 12:35:31      31,415,926.536
```

```
// nl-NL          27-08-2023 12:35:31      31.415.926,536
//                08/27/2023 12:35:31      31,415,926.536
```

Como mostra o exemplo, você pode usar uma FormattableString instância para gerar várias cadeias de caracteres de resultado para várias culturas.

Como criar uma cadeia de caracteres de resultado usando a cultura invariável

A partir do .NET 6, use o método String.Create(IFormatProvider, DefaultInterpolatedStringHandler) para converter uma cadeia de caracteres interpolada em uma cadeia de resultado para InvariantCulture, conforme mostra o exemplo a seguir.

C#

```
string message = string.Create(CultureInfo.InvariantCulture, $"Date and time in
invariant culture: {DateTime.Now}");
Console.WriteLine(message);
// Output is similar to:
// Date and time in invariant culture: 05/17/2018 15:46:24
```

Em versões anteriores do .NET, juntamente com o FormattableString.ToString(IFormatProvider) método, você pode usar o método estático FormattableString.Invariant , como mostra o exemplo a seguir:

C#

```
string message = FormattableString.Invariant($"Date and time in invariant culture:
{DateTime.Now}");
Console.WriteLine(message);
// Output is similar to:
// Date and time in invariant culture: 05/17/2018 15:46:24
```

Conclusão

Este tutorial descreve cenários comuns de uso de interpolação de cadeia de caracteres. Para obter mais informações sobre interpolação de cadeia de caracteres, consulte Interpolação de cadeia de caracteres. Para obter mais informações sobre tipos de formatação no .NET, consulte os artigos Tipos de formatação no .NET e Formatação composta.

Consulte também

- String.Format
- System.FormattableString
- System.IFormattable
- Cadeias de caracteres

Aplicativo de console

Artigo • 24/03/2025

Este tutorial ensina vários recursos no .NET e na linguagem C#. O que você aprenderá:

- As noções básicas da CLI do .NET
- A estrutura de um aplicativo de console em C#
- Entrada/Saída do console
- As noções básicas das APIs de E/S de Arquivo no .NET
- As noções básicas da programação assíncrona baseada em tarefa no .NET

Você criará um aplicativo que lê um arquivo de texto e ecoará o conteúdo desse arquivo de texto no console. A saída para o console é conduzida a fim de corresponder à leitura em voz alta. Você pode acelerar ou diminuir o ritmo pressionando as teclas '<' (menor que) ou '>' (maior que). Você pode executar esse aplicativo no Windows, Linux, macOS ou em um contêiner do Docker.

Há muitos recursos neste tutorial. Vamos construí-los um a um.

Pré-requisitos

- O mais recente SDK do .NET
- Editor do Visual Studio Code
- O DevKit C#

Criar o aplicativo

A primeira etapa é criar um novo aplicativo. Abra um prompt de comando e crie um novo diretório para seu aplicativo. Torne ele o diretório atual. Digite o comando `dotnet new console` no prompt de comando. Isso cria os arquivos iniciais para um aplicativo básico "Olá, Mundo!".

Antes de começar a fazer modificações, vamos executar o aplicativo Hello World simples. Depois de criar o aplicativo, digite `dotnet run` no prompt de comando. Esse comando executa o processo de restauração do pacote NuGet, cria o executável do aplicativo e executa o executável.

O código do aplicativo Hello World simples está todo em *Program.cs*. Abra esse arquivo com seu editor de texto favorito. Substitua o código em *Program.cs* pelo seguinte código:

```C#
namespace TeleprompterConsole;

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

Na parte superior do arquivo, observe uma instrução `namespace`. Como outras linguagens orientadas a objetos que você pode ter usado, o C# usa namespaces para organizar tipos. Este programa Hello World não é diferente. Você pode ver que o programa está no namespace com o nome `TeleprompterConsole`.

Como ler e exibir o arquivo

O primeiro recurso a ser adicionado é a capacidade de ler um arquivo de texto e exibir todo esse texto no console. Primeiro, vamos adicionar um arquivo de texto. Copie o arquivo sampleQuotes.txt do repositório GitHub para este exemplo de no seu diretório do projeto. Isso servirá como o script para seu aplicativo. Para obter informações sobre como baixar o aplicativo de exemplo para este tutorial, consulte as instruções em exemplos e tutoriais.

Em seguida, adicione o seguinte método na classe `Program` (logo abaixo do método `Main`):

```C#
static IEnumerable<string> ReadFrom(string file)
{
    string? line;
    using (var reader = File.OpenText(file))
    {
        while ((line = reader.ReadLine()) != null)
        {
            yield return line;
        }
    }
}
```

Esse método é um tipo especial de método C# chamado *método iterador*. Os métodos de iterador retornam sequências que são avaliadas lentamente. Isso significa que cada

item na sequência é gerado conforme solicitado pelo código que consome a sequência. Métodos de iterador são métodos que contêm uma ou mais declarações yield return. O objeto retornado pelo método `ReadFrom` contém o código para gerar cada item na sequência. Neste exemplo, isso envolve a leitura da próxima linha de texto do arquivo de origem e o retorno dessa cadeia de caracteres. Sempre que o código de chamada solicita o próximo item da sequência, o código lê a próxima linha de texto do arquivo e o retorna. Quando o arquivo é completamente lido, a sequência indica que não há mais itens.

Há dois elementos de sintaxe C# que podem ser novos para você. A instrução using neste método gerencia a limpeza de recursos. A variável inicializada na instrução `using` (`reader`, neste exemplo) deve implementar a interface IDisposable. Essa interface define um único método, `Dispose`, que deve ser chamado quando o recurso deve ser liberado. O compilador gera essa chamada quando a execução atingir a chave de fechamento da instrução `using`. O código gerado pelo compilador garante que o recurso seja liberado, mesmo se uma exceção for lançada do código no bloco definido pela instrução using.

A variável `reader` é definida usando a palavra-chave `var`. var define uma variável local *digitada implicitamente*. Isso significa que o tipo da variável é determinado pelo tipo de tempo de compilação do objeto atribuído à variável. Aqui, esse é o valor retornado do método OpenText(String), que é um objeto StreamReader.

Agora, vamos preencher o código para ler o arquivo no método `Main`:

C#

```
var lines = ReadFrom("sampleQuotes.txt");
foreach (var line in lines)
{
    Console.WriteLine(line);
}
```

Execute o programa (usando `dotnet run`) e você pode ver todas as linhas impressas no console.

Adicionar atrasos e formatar a saída

O que você possui está sendo exibido muito rápido para permitir a leitura em voz alta. Agora você precisa adicionar os atrasos na saída. Ao começar, você criará parte do código principal que habilita o processamento assíncrono. No entanto, essas primeiras etapas seguirão alguns antipadrões. Os antipadrões são apontados em comentários à medida que você adiciona o código e o código será atualizado em etapas posteriores.

Há duas etapas para esta seção. Primeiro, você atualizará o método iterador para retornar palavras simples em vez de linhas inteiras. Isso é feito com estas modificações. Substitua a instrução `yield return line;` pelo seguinte código:

```
C#

var words = line.Split(' ');
foreach (var word in words)
{
    yield return word + " ";
}
yield return Environment.NewLine;
```

Em seguida, será necessário modificar a forma como você consume as linhas do arquivo, e adicionar um atraso depois de escrever cada palavra. Substitua a instrução `Console.WriteLine(line)` no método `Main` pelo seguinte bloco:

```
C#

Console.Write(line);
if (!string.IsNullOrWhiteSpace(line))
{
    var pause = Task.Delay(200);
    // Synchronously waiting on a task is an
    // anti-pattern. This will get fixed in later
    // steps.
    pause.Wait();
}
```

Execute o exemplo e verifique a saída. Agora, cada palavra é impressa, seguida por um atraso de 200 ms. No entanto, a saída exibida mostra alguns problemas porque o arquivo de texto de origem tem várias linhas que têm mais de 80 caracteres sem uma quebra de linha. Isso pode ser difícil de ler durante a rolagem da tela. Isso é fácil de corrigir. Você apenas manterá o controle do comprimento de cada linha e gerará uma nova linha sempre que o comprimento da linha atingir um determinado limite. Declare uma variável local após a declaração de `words` no método `ReadFrom` que contém o comprimento da linha:

```
C#

var lineLength = 0;
```

Em seguida, adicione o seguinte código após a instrução `yield return word + " ";` (antes da chave de fechamento):

```C#
lineLength += word.Length + 1;
if (lineLength > 70)
{
    yield return Environment.NewLine;
    lineLength = 0;
}
```

Execute o exemplo e você poderá ler em voz alta em seu ritmo pré-configurado.

Tarefas assíncronas

Nesta etapa final, você adicionará o código para gravar a saída de forma assíncrona em uma tarefa, ao mesmo tempo em que executará outra tarefa para ler a entrada do usuário se ele quiser acelerar ou diminuir a velocidade da exibição de texto ou interromper completamente a exibição de texto. Isso tem algumas etapas e, no final, você terá todas as atualizações necessárias. A primeira etapa é criar um método de retorno de Task assíncrono que representa o código que você criou até agora para ler e exibir o arquivo.

Adicione este método à sua classe `Program` (ele é obtido do corpo de seu método `Main`):

```C#
private static async Task ShowTeleprompter()
{
    var words = ReadFrom("sampleQuotes.txt");
    foreach (var word in words)
    {
        Console.Write(word);
        if (!string.IsNullOrWhiteSpace(word))
        {
            await Task.Delay(200);
        }
    }
}
```

Você observará duas alterações. Primeiro, no corpo do método, em vez de chamar Wait() para aguardar de forma síncrona a conclusão de uma tarefa, essa versão usa a palavra-chave `await`. Para fazer isso, você precisa adicionar o modificador `async` à assinatura do método. Esse método retorna um `Task`. Observe que não há instruções de retorno que retornem um objeto `Task`. Em vez disso, esse objeto `Task` é criado por código que o compilador gera quando você usa o operador `await`. Você pode imaginar

que esse método retorna quando atinge um `await`. O `Task` retornado indica que o trabalho não foi concluído. O método é retomado quando a tarefa aguardada é concluída. Quando ele tiver sido executado para conclusão, o `Task` retornado indica que ele está concluído. O código de chamada pode monitorar essa `Task` retornada para determinar quando ela foi concluída.

Adicione uma palavra-chave `await` antes da chamada para `ShowTeleprompter`:

C#

```
await ShowTeleprompter();
```

Isso exige que você altere a assinatura do método `Main` para:

C#

```
static async Task Main(string[] args)
```

Saiba mais sobre o método async Main em nossa seção de conceitos básicos.

Em seguida, é necessário escrever o segundo método assíncrono a ser lido no Console e ficar atento às teclas "<" (menor que), ">" (maior que) e "X" ou "x". Este é o método que você adiciona para essa tarefa:

C#

```
private static async Task GetInput()
{
    var delay = 200;
    Action work = () =>
    {
        do {
            var key = Console.ReadKey(true);
            if (key.KeyChar == '>')
            {
                delay -= 10;
            }
            else if (key.KeyChar == '<')
            {
                delay += 10;
            }
            else if (key.KeyChar == 'X' || key.KeyChar == 'x')
            {
                break;
            }
        } while (true);
    };
```

```
    await Task.Run(work);
}
```

Isso cria uma expressão lambda para representar um representante Action que lê uma chave do Console e modifica uma variável local que representa o atraso quando o usuário pressiona as teclas '<' (menor que) ou '>' (maior que). O método delegado é concluído quando o usuário pressiona as teclas 'X' ou 'x', que permitem que o usuário interrompa a exibição de texto a qualquer momento. Esse método usa ReadKey() para bloquear e aguardar o usuário pressionar uma tecla.

Para concluir esta funcionalidade, você precisa criar um novo método de retorno `async Task` que inicie ambas essas tarefas (`GetInput` e `ShowTeleprompter`) e também gerencie os dados compartilhados pelas duas tarefas.

É hora de criar uma classe que possa lidar com os dados compartilhados entre essas duas tarefas. Essa classe contém duas propriedades públicas: o atraso e um sinalizador `Done` para indicar que o arquivo foi completamente lido:

C#

```
namespace TelepromptерConsole;

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;
    }
}
```

Coloque essa classe em um novo arquivo e inclua essa classe no namespace `TelepromptерConsole`, conforme mostrado. Você também precisará adicionar uma instrução `using static` no início do arquivo para poder referenciar os métodos `Min` e `Max` sem os nomes de classe ou namespace. Uma instrução using static importa os métodos de uma classe. Isso contrasta com a instrução `using` sem `static`, que importa todas as classes de um namespace.

C#

```
using static System.Math;
```

Em seguida, você precisa atualizar os métodos `ShowTeleprompter` e `GetInput` para usar o novo objeto `config`. Escreva um método final `async` de retorno de `Task` para iniciar as duas tarefas e sair quando a primeira tarefa for concluída:

C#

```
private static async Task RunTeleprompter()
{
    var config = new TelePrompterConfig();
    var displayTask = ShowTeleprompter(config);

    var speedTask = GetInput(config);
    await Task.WhenAny(displayTask, speedTask);
}
```

O novo método aqui é a chamada WhenAny(Task[]). Isso cria uma `Task` que termina assim que qualquer uma das tarefas na lista de argumentos for concluída.

Depois, atualize os métodos `ShowTeleprompter` e `GetInput` para usar o objeto `config` para o atraso:

C#

```
private static async Task ShowTeleprompter(TelePrompterConfig config)
{
    var words = ReadFrom("sampleQuotes.txt");
    foreach (var word in words)
    {
        Console.Write(word);
        if (!string.IsNullOrWhiteSpace(word))
        {
            await Task.Delay(config.DelayInMilliseconds);
        }
    }
    config.SetDone();
}

private static async Task GetInput(TelePrompterConfig config)
{
    Action work = () =>
    {
        do {
            var key = Console.ReadKey(true);
            if (key.KeyChar == '>')
                config.UpdateDelay(-10);
            else if (key.KeyChar == '<')
                config.UpdateDelay(10);
```

```
            else if (key.KeyChar == 'X' || key.KeyChar == 'x')
                config.SetDone();
        } while (!config.Done);
    };
    await Task.Run(work);
}
```

Esta nova versão do `ShowTeleprompter` chama um novo método na classe `TelepromptерConfig`. Agora, você precisa atualizar `Main` para chamar `RunTeleprompter` em vez de `ShowTeleprompter`:

```
C#

await RunTeleprompter();
```

Conclusão

Este tutorial mostrou vários recursos em torno da linguagem C# e das bibliotecas do .NET Core relacionadas ao trabalho em aplicativos de Console. Você pode criar esse conhecimento para explorar mais sobre o idioma e as classes introduzidas aqui. Você já viu os fundamentos de E/S do Arquivo e do Console, uso com bloqueio e sem bloqueio da programação assíncrona controlada por tarefa, um tour pela linguagem C# e como os programas em C# são organizados, além da CLI do .NET.

Para obter mais informações sobre E/S de arquivo, consulte E/S de arquivo e de fluxo. Para obter mais informações sobre o modelo de programação assíncrona usado neste tutorial, consulte Programação Assíncrona Baseada em Tarefa e Programação Assíncrona .

Tutorial: Fazer solicitações HTTP em um aplicativo de console .NET usando C

Artigo • 10/05/2023

Esse tutorial cria um aplicativo que emite solicitações HTTP para um serviço REST no GitHub. O aplicativo lê informações no formato JSON e converte o JSON em objetos C#. A conversão de objetos JSON em C# é conhecida como *desserialização*.

Este tutorial mostra como:

- ✓ Enviar solicitações HTTP.
- ✓ Desserializar respostas JSON.
- ✓ Configurar a desserialização com atributos.

Se preferir seguir com o exemplo final desse tutorial, você pode baixá-lo. Para obter instruções de download, consulte Exemplos e tutoriais.

Pré-requisitos

- SDK do .NET 6.0 ou posterior
- Um editor de código como [Visual Studio Code (um editor de software de código aberto e multiplataforma). Você pode executar o aplicativo de exemplo no Windows, Linux ou macOS ou em um contêiner do Docker.

Criar o aplicativo cliente

1. Abra um prompt de comando e crie um novo diretório para seu aplicativo. Torne ele o diretório atual.

2. Insira o seguinte comando em uma janela do console:

   CLI do .NET

   ```
   dotnet new console --name WebAPIClient
   ```

   Esse comando cria os arquivos iniciais de um aplicativo "Hello World" básico. O nome do projeto é "WebAPIClient".

3. Navegue até o diretório "WebAPIClient" e execute o aplicativo.

   CLI do .NET

```
cd WebAPIClient
```

CLI do .NET

```
dotnet run
```

dotnet run executa dotnet restore automaticamente para restaurar as dependências de que o aplicativo precisa. Ele também executa dotnet build se necessário. Você deve ver a saída do aplicativo `"Hello, World!"`. No terminal, pressione `Ctrl`+`C` para interromper o aplicativo.

Fazer solicitações HTTP

Esse aplicativo chama a API do GitHub para obter informações sobre os projetos no escopo do .NET Foundation. O ponto de extremidade é https://api.github.com/orgs/dotnet/repos. Para recuperar informações, ele faz uma solicitação HTTP GET. O navegador também faz solicitações HTTP GET, para que você possa colar essa URL na barra de endereços do seu navegador e ver as informações recebidas e processadas.

Use a classe HttpClient para fazer solicitações HTTP. HttpClient dá suporte apenas a métodos assíncronos para suas APIs de execução longa. Portanto, as etapas a seguir criam um método assíncrono e o chamam do método Main.

1. Abra o arquivo `Program.cs` no diretório do projeto e substitua seu conteúdo pelo seguinte:

   C#

   ```
   await ProcessRepositoriesAsync();

   static async Task ProcessRepositoriesAsync(HttpClient client)
   {
   }
   ```

   Esse código:

   - Substitui a instrução `Console.WriteLine` por uma chamada para `ProcessRepositoriesAsync`, a qual usa a palavra-chave `await`.
   - Define um método `ProcessRepositoriesAsync` vazio.

2. Na classe `Program`, use um HttpClient para manipular solicitações e respostas ao substituir o conteúdo pelo C# a seguir.

```C#
using System.Net.Http.Headers;

using HttpClient client = new();
client.DefaultRequestHeaders.Accept.Clear();
client.DefaultRequestHeaders.Accept.Add(
    new
MediaTypeWithQualityHeaderValue("application/vnd.github.v3+json"));
client.DefaultRequestHeaders.Add("User-Agent", ".NET Foundation
Repository Reporter");

await ProcessRepositoriesAsync(client);

static async Task ProcessRepositoriesAsync(HttpClient client)
{
}
```

Esse código:

- Configura cabeçalhos HTTP de todas as solicitações:
  - Um cabeçalho Accept para aceitar respostas JSON
  - Um cabeçalho User-Agent. Esses cabeçalhos são verificados pelo código do servidor GitHub e são necessários para recuperar informações do GitHub.

3. No método `ProcessRepositoriesAsync`, chame o ponto de extremidade do GitHub que retorna uma lista de todos os repositórios na organização do .NET Foundation:

```C#
static async Task ProcessRepositoriesAsync(HttpClient client)
{
    var json = await client.GetStringAsync(
        "https://api.github.com/orgs/dotnet/repos");

    Console.Write(json);
}
```

Esse código:

- Aguarda a tarefa retornada do método de chamada HttpClient.GetStringAsync(String). Esse método envia uma solicitação HTTP

GET para o URI especificado. O corpo da resposta é retornado como um String, que está disponível quando a tarefa é concluída.
- A cadeia de caracteres de resposta `json` é impressa no console.

4. Compile o aplicativo e execute-o.

CLI do .NET

```
dotnet run
```

Não há nenhum aviso de build porque o `ProcessRepositoriesAsync` agora contém um operador `await`. A saída é uma longa exibição de texto JSON.

Desserializar o resultado JSON

As etapas a seguir convertem a resposta JSON em objetos C#. Use a classe System.Text.Json.JsonSerializer para desserializar o JSON em objetos.

1. Crie um arquivo chamado *Repository.cs* e adicione o código a seguir:

C#

```
public record class Repository(string name);
```

O código anterior define uma classe para representar o objeto JSON retornado da API do GitHub. Você usará essa classe para exibir uma lista de nomes de repositório.

O JSON de um objeto de repositório contém dezenas de propriedades, mas apenas a propriedade `name` será desserializada. O serializador ignora automaticamente as propriedades JSON para as quais não há correspondência na classe de destino. Esse recurso facilita a criação de tipos que funcionam apenas com um subconjunto de campos em um pacote JSON grande.

A convenção C# é capitalizar a primeira letra de nomes de propriedade, mas a propriedade `name` aqui começa com uma letra minúscula porque corresponde exatamente ao que está no JSON. Posteriormente, você verá como usar nomes de propriedade C# que não correspondem aos nomes de propriedade JSON.

2. Use o serializador para converter JSON em objetos C#. Substitua a chamada para GetStringAsync(String) no método `ProcessRepositoriesAsync` pelas duas linhas a seguir:

```C#
await using Stream stream =
    await
client.GetStreamAsync("https://api.github.com/orgs/dotnet/repos");
var repositories =
    await JsonSerializer.DeserializeAsync<List<Repository>>(stream);
```

O código atualizado substitui GetStringAsync(String) por GetStreamAsync(String). Esse método de serializador usa um fluxo, em vez de uma cadeia de caracteres, como sua fonte.

O primeiro argumento JsonSerializer.DeserializeAsync<TValue>(Stream, JsonSerializerOptions, CancellationToken) é uma expressão `await`. Expressões `await` podem aparecer em quase todo lugar em seu código, apesar de que até o momento, você apenas as viu como parte de uma instrução de atribuição. Os outros dois parâmetros `JsonSerializerOptions` e `CancellationToken`, são opcionais e são omitidos no snippet de código.

O método `DeserializeAsync` é *genérico*, o que significa que você fornece argumentos de tipo para que tipo de objetos devem ser criados a partir do texto JSON. Neste exemplo, você está desserializando para um `List<Repository>`, que é outro objeto genérico, um System.Collections.Generic.List<T>. A classe `List<T>` armazena uma coleção de objetos. O argumento type declara o tipo de objetos armazenados no `List<T>`. O argumento de tipo é seu registro `Repository`, pois o texto JSON representa uma coleção de objetos de repositório.

3. Adicione código para exibir o nome de cada repositório. Substitua as linhas que mostram:

```C#
Console.Write(json);
```

pelo código a seguir:

```C#
foreach (var repo in repositories ?? Enumerable.Empty<Repository>())
    Console.Write(repo.name);
```

4. As seguintes diretivas `using` devem estar presentes na parte superior do arquivo:

```C#
using System.Net.Http.Headers;
using System.Text.Json;
```

5. Execute o aplicativo.

```CLI do .NET
dotnet run
```

A saída é uma lista dos nomes dos repositórios que fazem parte do .NET Foundation.

Configurar a desserialização

1. Em *Repository.cs*, substitua o conteúdo do arquivo pelo C# a seguir.

```C#
using System.Text.Json.Serialization;

public record class Repository(
    [property: JsonPropertyName("name")] string Name);
```

Esse código:

- Altera o nome da propriedade `name` para `Name`.
- Adiciona o JsonPropertyNameAttribute para especificar como essa propriedade aparece no JSON.

2. Em *Program.cs*, atualize o código para usar a nova capitalização da propriedade `Name`:

```C#
foreach (var repo in repositories)
   Console.Write(repo.Name);
```

3. Execute o aplicativo.

A saída é a mesma.

Refatorar o código

O método `ProcessRepositoriesAsync` pode fazer o trabalho assíncrono e retornar uma coleção de repositórios. Altere esse método para retornar `Task<List<Repository>>` e mova o código que grava no console perto do chamador.

1. Altere a assinatura de `ProcessRepositoriesAsync` para retornar uma tarefa cujo resultado é uma lista de objetos `Repository`:

   ```C#
   static async Task<List<Repository>> ProcessRepositoriesAsync()
   ```

2. Retorne os repositórios depois de processar a resposta JSON:

   ```C#
   await using Stream stream =
       await
   client.GetStreamAsync("https://api.github.com/orgs/dotnet/repos");
   var repositories =
       await JsonSerializer.DeserializeAsync<List<Repository>>(stream);
   return repositories ?? new();
   ```

   O compilador gera o objeto `Task<T>` para o calor de retorno porque você marcou esse método como `async`.

3. Modifique o arquivo *Program.cs*, substituindo a chamada para `ProcessRepositoriesAsync` pelo seguinte para capturar os resultados e gravar cada nome do repositório no console.

   ```C#
   var repositories = await ProcessRepositoriesAsync(client);

   foreach (var repo in repositories)
       Console.Write(repo.Name);
   ```

4. Execute o aplicativo.

   A saída é a mesma.

Desserializar mais propriedades

As etapas a seguir adicionam código para processar mais das propriedades no pacote JSON recebido. Você provavelmente não vai querer processar todas as propriedades, mas adicionar mais algumas demonstra outros recursos de C#.

1. Substitua o conteúdo da classe `Repository` pela seguinte definição `record`:

   C#

   ```
   using System.Text.Json.Serialization;

   public record class Repository(
       [property: JsonPropertyName("name")] string Name,
       [property: JsonPropertyName("description")] string Description,
       [property: JsonPropertyName("html_url")] Uri GitHubHomeUrl,
       [property: JsonPropertyName("homepage")] Uri Homepage,
       [property: JsonPropertyName("watchers")] int Watchers);
   ```

   Os tipos Uri e `int` têm funcionalidade interna para converter de e para a representação de cadeia de caracteres. Nenhum código extra é necessário para desserializar do formato de cadeia de caracteres JSON para esses tipos de destino. Se o pacote JSON contiver dados que não são convertidos em um tipo de destino, a ação de serialização gerará uma exceção.

2. Atualize o loop `foreach` no arquivo *Program.cs* para exibir os valores da propriedade:

   C#

   ```
   foreach (var repo in repositories)
   {
       Console.WriteLine($"Name: {repo.Name}");
       Console.WriteLine($"Homepage: {repo.Homepage}");
       Console.WriteLine($"GitHub: {repo.GitHubHomeUrl}");
       Console.WriteLine($"Description: {repo.Description}");
       Console.WriteLine($"Watchers: {repo.Watchers:#,0}");
       Console.WriteLine();
   }
   ```

3. Execute o aplicativo.

   A lista agora inclui as propriedades adicionais.

Adicionar uma propriedade date

A data da última operação de push é formatada dessa forma na resposta JSON:

```JSON
2016-02-08T21:27:00Z
```

Esse formato é para UTC (Tempo Universal Coordenado), portanto, o resultado da desserialização é um valor DateTime cuja propriedade Kind é Utc.

Para obter uma data e hora representadas em seu fuso horário, você precisa escrever um método de conversão personalizado.

1. Em *Repository.cs*, adicione uma propriedade para a representação UTC da data e hora e uma propriedade `LastPush` somente leitura que retorna a data convertida em hora local. O arquivo deve se parecer com o seguinte:

   ```C#
   using System.Text.Json.Serialization;

   public record class Repository(
       [property: JsonPropertyName("name")] string Name,
       [property: JsonPropertyName("description")] string Description,
       [property: JsonPropertyName("html_url")] Uri GitHubHomeUrl,
       [property: JsonPropertyName("homepage")] Uri Homepage,
       [property: JsonPropertyName("watchers")] int Watchers,
       [property: JsonPropertyName("pushed_at")] DateTime LastPushUtc)
   {
       public DateTime LastPush => LastPushUtc.ToLocalTime();
   }
   ```

   A propriedade `LastPush` é definida usando um *membro com corpo de expressão* para o acessador `get`. Não há acessador `set` . Omitir o acessador `set` é uma maneira de definir uma propriedade *somente leitura* em C#. (Sim, você pode criar propriedades *somente gravação* em C#, mas o valor delas é limitado.)

2. Adicione outra instrução de saída em *Program.cs* novamente:

   ```C#
   Console.WriteLine($"Last push: {repo.LastPush}");
   ```

3. O aplicativo completo deve ser semelhante ao seguinte arquivo *Program.cs*:

   ```C#
   using System.Net.Http.Headers;
   using System.Text.Json;
   ```

```
using HttpClient client = new();
client.DefaultRequestHeaders.Accept.Clear();
client.DefaultRequestHeaders.Accept.Add(
    new
MediaTypeWithQualityHeaderValue("application/vnd.github.v3+json"));
client.DefaultRequestHeaders.Add("User-Agent", ".NET Foundation
Repository Reporter");

var repositories = await ProcessRepositoriesAsync(client);

foreach (var repo in repositories)
{
    Console.WriteLine($"Name: {repo.Name}");
    Console.WriteLine($"Homepage: {repo.Homepage}");
    Console.WriteLine($"GitHub: {repo.GitHubHomeUrl}");
    Console.WriteLine($"Description: {repo.Description}");
    Console.WriteLine($"Watchers: {repo.Watchers:#,0}");
    Console.WriteLine($"{repo.LastPush}");
    Console.WriteLine();
}

static async Task<List<Repository>> ProcessRepositoriesAsync(HttpClient
client)
{
    await using Stream stream =
        await
client.GetStreamAsync("https://api.github.com/orgs/dotnet/repos");
    var repositories =
        await JsonSerializer.DeserializeAsync<List<Repository>>
(stream);
    return repositories ?? new();
}
```

4. Execute o aplicativo.

   A saída inclui a data e a hora do último push de cada repositório.

Próximas etapas

Neste tutorial, você criou um aplicativo que faz solicitações da Web e analisa os resultados. Agora, sua versão do aplicativo deve corresponder ao exemplo finalizado.

Saiba mais sobre como configurar a serialização JSON em Como serializar e desserializar (marshal and unmarshal) JSON no .NET.

Trabalhar com a LINQ (Consulta Integrada à Linguagem)

Artigo • 10/05/2023

Introdução

Este tutorial ensina os recursos no .NET Core e da linguagem C#. Você aprenderá a:

- Gerar sequências com a LINQ.
- Escrever métodos que podem ser usados com facilidade em consultas LINQ.
- Distinguir entre avaliação lenta e detalhada.

Você aprenderá essas técnicas ao compilar um aplicativo que demonstra uma das habilidades básicas de qualquer mágico: o embaralhamento faro. Em resumo, um embaralhamento faro é uma técnica em que você divide um baralho de cartas exatamente na metade, então as cartas de cada metade são colocadas em ordem aleatória até recriar o conjunto original.

Os mágicos usam essa técnica porque cada carta é fica em um local conhecido após o embaralhamento e a ordem é um padrão de repetição.

Para os seus propósitos, vamos examinar rapidamente as sequências de manipulação de dados. O aplicativo que você cria constrói um baralho de cartas e executa uma sequência de embaralhamento, sempre gravando a sequência de saída. Você também comparará a ordem atualizada com a ordem original.

Este tutorial tem várias etapas. Após cada etapa, você poderá executar o aplicativo e ver o progresso. Você também poderá ver o exemplo concluído no repositório dotnet/samples do GitHub. Para obter instruções de download, consulte Exemplos e tutoriais.

Pré-requisitos

Você precisará configurar seu computador para executar o .NET Core. Você vai encontrar as instruções de instalação na página de download do .NET Core. Você pode executar esse aplicativo no Windows, no Ubuntu Linux, no OS X ou em um contêiner do Docker. Será necessário instalar o editor de código de sua preferência. As descrições a seguir usam o Visual Studio Code, que é um Editor de código aberto multiplataforma. No entanto, você pode usar quaisquer ferramentas que esteja familiarizado.

Criar o aplicativo

A primeira etapa é criar um novo aplicativo. Abra um prompt de comando e crie um novo diretório para seu aplicativo. Torne ele o diretório atual. Digite o comando `dotnet new console` no prompt de comando. Isso cria os arquivos iniciais de um aplicativo "Olá, Mundo" básico.

Se você nunca usou C# antes, este tutorial explicará a estrutura de um programa C#. Você pode ler e, em seguida, voltar aqui para saber mais sobre o LINQ.

Criar o conjunto de dados

Antes de começar, verifique se as linhas a seguir estão na parte superior do arquivo `Program.cs` gerado pelo `dotnet new console`:

C#

```
// Program.cs
using System;
using System.Collections.Generic;
using System.Linq;
```

Se essas três linhas (instruções `using`) não estiverem na parte superior do arquivo, nosso programa não será compilado.

Agora que você tem todas as referências necessárias, considere o que forma um baralho de cartas. Um baralho de cartas costuma ter quatro naipes, e cada naipe tem treze valores. Normalmente, talvez você pense em criar uma classe `Card` logo de cara e preencher uma coleção de objetos `Card` manualmente. Com o LINQ, dá para ser mais conciso do que a forma comum de criação de um baralho de cartas. Em vez de criar uma classe `Card`, você pode criar duas sequências para representar naipes e valores, respectivamente. Você vai criar um par muito simples de *métodos iteradores* que gerará as valores e naipes como IEnumerable<T>s de cadeias de caracteres:

C#

```
// Program.cs
// The Main() method

static IEnumerable<string> Suits()
{
    yield return "clubs";
    yield return "diamonds";
    yield return "hearts";
```

```
    yield return "spades";
}

static IEnumerable<string> Ranks()
{
    yield return "two";
    yield return "three";
    yield return "four";
    yield return "five";
    yield return "six";
    yield return "seven";
    yield return "eight";
    yield return "nine";
    yield return "ten";
    yield return "jack";
    yield return "queen";
    yield return "king";
    yield return "ace";
}
```

Coloque-as sob o método `Main` em seu arquivo `Program.cs`. Esses dois métodos utilizam a sintaxe `yield return` para produzir uma sequência à medida que eles são executados. O compilador compila um objeto que implementa IEnumerable<T> e gera a sequência de cadeias de caracteres conforme solicitado.

Agora, use esses métodos iteradores para criar o baralho de cartas. Você colocará a consulta do LINQ em nosso método `Main`. Dê uma olhada:

C#

```
// Program.cs
static void Main(string[] args)
{
    var startingDeck = from s in Suits()
                       from r in Ranks()
                       select new { Suit = s, Rank = r };

    // Display each card that we've generated and placed in startingDeck in
the console
    foreach (var card in startingDeck)
    {
        Console.WriteLine(card);
    }
}
```

As várias cláusulas `from` produzem um SelectMany, que cria uma única sequência da combinação entre cada elemento na primeira sequência com cada elemento na segunda sequência. A ordem é importante para nossos objetivos. O primeiro elemento na primeira sequência de fonte (Naipes) é combinado com cada elemento na segunda

sequência (Valores). Isso produz todas as treze cartas do primeiro naipe. Esse processo é repetido com cada elemento na primeira sequência (naipes). O resultado final é um baralho ordenado por naipes, seguido pelos valores.

É importante lembrar que se você optar por escrever seu LINQ na sintaxe de consulta usada acima, ou se decidir usar a sintaxe de método, sempre será possível alternar entre as formas de sintaxe. A consulta acima escrita em sintaxe de consulta pode ser escrita na sintaxe de método como:

```
C#

var startingDeck = Suits().SelectMany(suit => Ranks().Select(rank => new {
Suit = suit, Rank = rank }));
```

O compilador traduz instruções LINQ escritas com a sintaxe de consulta na sintaxe de chamada do método equivalente. Portanto, independentemente de sua escolha de sintaxe, as duas versões da consulta produzem o mesmo resultado. Escolha qual sintaxe funciona melhor para a sua situação: por exemplo, se você estiver trabalhando em uma equipe em que alguns dos membros têm dificuldade com a sintaxe de método, prefira usar a sintaxe de consulta.

Vá em frente e execute o exemplo que você criou neste momento. Ele exibirá todas as 52 cartas do baralho. Talvez seja muito útil executar esse exemplo em um depurador para observar como os métodos `Suits()` e `Ranks()` são executados. Você pode ver claramente que cada cadeia de caracteres em cada sequência é gerada apenas conforme o necessário.

Manipular a ordem

Em seguida, concentre-se em como você vai embaralhar as cartas no baralho. A primeira etapa de qualquer embaralhada é dividir o baralho em dois. Os métodos Take e Skip que fazem parte das APIs do LINQ fornecem esse recurso para você. Coloque-os sob o loop `foreach`:

C#

```
// Program.cs
public static void Main(string[] args)
{
    var startingDeck = from s in Suits()
                       from r in Ranks()
                       select new { Suit = s, Rank = r };

    foreach (var c in startingDeck)
    {
        Console.WriteLine(c);
    }

    // 52 cards in a deck, so 52 / 2 = 26
    var top = startingDeck.Take(26);
    var bottom = startingDeck.Skip(26);
}
```

No entanto, não há método de embaralhamento na biblioteca padrão, portanto, você precisará escrever o seu. O método de embaralhamento que você criará ilustra várias técnicas que você usará com programas baseados em LINQ. Portanto, cada parte desse processo será explicado nas etapas.

Para adicionar funcionalidade ao seu modo de interação com o IEnumerable<T> recebido de volta das consultas do LINQ, precisará escrever alguns tipos especiais de métodos chamados métodos de extensão. Em resumo, um método de extensão é um *método estático* de objetivo especial que adiciona novas funcionalidades a um tipo já existentes, sem ter que modificar o tipo original ao qual você deseja adicionar funcionalidade.

Dê aos seus métodos de extensão uma nova casa adicionando um novo arquivo de classe *estático* ao seu programa chamado `Extensions.cs`, depois, comece a criar o primeiro método de extensão:

C#

```
// Extensions.cs
using System;
```

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

namespace LinqFaroShuffle
{
    public static class Extensions
    {
        public static IEnumerable<T> InterleaveSequenceWith<T>(this
IEnumerable<T> first, IEnumerable<T> second)
        {
            // Your implementation will go here soon enough
        }
    }
}
```

Examine a assinatura do método por um momento, principalmente os parâmetros:

```
C#

public static IEnumerable<T> InterleaveSequenceWith<T> (this IEnumerable<T>
first, IEnumerable<T> second)
```

Você pode ver a adição do modificador `this` no primeiro argumento para o método. Isso significa que você chama o método como se fosse um método de membro do tipo do primeiro argumento. Esta declaração de método também segue um idioma padrão no qual os tipos de entrada e saídas são `IEnumerable<T>`. Essa prática permite que os métodos LINQ sejam encadeados para executar consultas mais complexas.

Naturalmente, como você dividiu o baralho em metades, precisará unir essas metades. No código, isso significa que você vai enumerar as duas sequências adquiridas por meio de Take e Skip ao mesmo tempo, *`interleaving`* os elementos e criará uma sequência: seu baralho não embaralhado. Escrever um método LINQ que funciona com duas sequências exige que você compreenda como IEnumerable<T> funciona.

A interface IEnumerable<T> tem um método: GetEnumerator. O objeto retornado por GetEnumerator tem um método para mover para o próximo elemento e uma propriedade que recupera o elemento atual na sequência. Você usará esses dois membros para enumerar a coleção e retornar os elementos. Esse método de Intercalação será um método iterador, portanto, em vez de criar uma coleção e retornar a coleção, você usará a sintaxe `yield return` mostrada acima.

Aqui está a implementação desse método:

```
C#
```

```
public static IEnumerable<T> InterleaveSequenceWith<T>
    (this IEnumerable<T> first, IEnumerable<T> second)
{
    var firstIter = first.GetEnumerator();
    var secondIter = second.GetEnumerator();

    while (firstIter.MoveNext() && secondIter.MoveNext())
    {
        yield return firstIter.Current;
        yield return secondIter.Current;
    }
}
```

Agora que você escreveu esse método, vá até o método `Main` e embaralhe uma vez:

C#

```
// Program.cs
public static void Main(string[] args)
{
    var startingDeck = from s in Suits()
                       from r in Ranks()
                       select new { Suit = s, Rank = r };

    foreach (var c in startingDeck)
    {
        Console.WriteLine(c);
    }

    var top = startingDeck.Take(26);
    var bottom = startingDeck.Skip(26);
    var shuffle = top.InterleaveSequenceWith(bottom);

    foreach (var c in shuffle)
    {
        Console.WriteLine(c);
    }
}
```

Comparações

Quantos embaralhamentos são necessários para colocar o baralho em sua ordem original? Para descobrir, você precisará escrever um método que determina se duas sequências são iguais. Depois de ter esse método, você precisará colocar o código de embaralhamento em um loop e verificar quando a apresentação estiver na ordem.

Escrever um método para determinar se as duas sequências são iguais deve ser simples. É uma estrutura semelhante para o método que você escreveu para embaralhar as

cartas. Somente desta vez, em vez de o `yield return` rendimento retornar cada elemento, você comparará os elementos correspondentes de cada sequência. Quando toda a sequência tiver sido enumerada, se os elementos corresponderem, as sequências serão as mesmas:

C#

```
public static bool SequenceEquals<T>
    (this IEnumerable<T> first, IEnumerable<T> second)
{
    var firstIter = first.GetEnumerator();
    var secondIter = second.GetEnumerator();

    while ((firstIter?.MoveNext() == true) && secondIter.MoveNext())
    {
        if ((firstIter.Current is not null) &&
!firstIter.Current.Equals(secondIter.Current))
        {
            return false;
        }
    }

    return true;
}
```

Isso mostra uma segunda linguagem LINQ: métodos de terminal. Eles consideram uma sequência como entrada (ou, neste caso, duas sequências) e retornam um único valor escalar. Ao usar métodos de terminal, eles são sempre o método final em uma cadeia de métodos para uma consulta LINQ, por isso, o nome "terminal".

Você pode ver isso em ação ao usá-lo para determinar quando o baralho está em sua ordem original. Coloque o código de embaralhamento dentro de um loop e pare quando a sequência estiver em sua ordem original, aplicando o método `SequenceEquals()`. Você pode ver que esse sempre será o método final em qualquer consulta, porque ele retorna um valor único em vez de uma sequência:

C#

```
// Program.cs
static void Main(string[] args)
{
    // Query for building the deck

    // Shuffling using InterleaveSequenceWith<T>();

    var times = 0;
    // We can re-use the shuffle variable from earlier, or you can make a
new one
    shuffle = startingDeck;
```

```
    do
    {
        shuffle = shuffle.Take(26).InterleaveSequenceWith(shuffle.Skip(26));

        foreach (var card in shuffle)
        {
            Console.WriteLine(card);
        }
        Console.WriteLine();
        times++;

    } while (!startingDeck.SequenceEquals(shuffle));

    Console.WriteLine(times);
}
```

Execute o código que obtivemos até agora e observe como o baralho é reorganizado em cada embaralhamento. Após 8 embaralhamentos (iterações do loop do-while), o baralho retorna à configuração original que estava quando você o criou pela primeira vez a partir da consulta LINQ inicial.

Otimizações

O exemplo que você compilou até agora executa um *embaralhamento externo*, no qual as cartas superiores e inferiores permanecem as mesmas em cada execução. Vamos fazer uma alteração: em vez disso, usaremos um *embaralhamento interno*, em que todas as 52 cartas trocam de posição. Para um embaralhamento interno, intercale o baralho para que a primeira carta da metade inferior torne-se a primeira carta do baralho. Isso significa que a última carta na metade superior torna-se a carta inferior. Essa é uma alteração simples em uma única linha de código. Atualize a consulta atual de embaralhamento, alternando as posições de Take e Skip. Isso alterará a ordem das metades superior e inferior do baralho:

C#

```
shuffle = shuffle.Skip(26).InterleaveSequenceWith(shuffle.Take(26));
```

Execute o programa novamente e você verá que leva 52 iterações para o baralho ser reordenado. Você também começará a observar algumas degradações de desempenho graves à medida que o programa continuar a ser executado.

Existem muitas razões para isso. Você pode abordar uma das principais causas dessa queda de desempenho: uso ineficiente da *avaliação lenta*.

Em resumo, a avaliação lenta informa que a avaliação de uma instrução não será executada até que seu valor seja necessário. Consultas LINQ são instruções avaliadas lentamente. As sequências são geradas somente quando os elementos são solicitados. Geralmente, esse é o principal benefício do LINQ. No entanto, em uso como esse programa, isso causa um crescimento exponencial no tempo de execução.

Lembre-se de que geramos o baralho original usando uma consulta LINQ. Cada embaralhamento é gerado executando três consultas LINQ no baralho anterior. Todos eles são executados lentamente. Isso também significa que eles são executados novamente sempre que a sequência é solicitada. Ao obter a 52ª iteração, você estará regenerando o baralho original muitas e muitas vezes. Vamos escrever um log para demonstrar esse comportamento. Em seguida, você poderá corrigir isso.

Em seu arquivo `Extensions.cs`, digite ou copie o método a seguir. Esse método de extensão cria um novo arquivo chamado `debug.log` em seu diretório do projeto, e registra qual consulta está sendo executada atualmente para o arquivo de log. Este método de extensão pode ser anexado a qualquer consulta para marcar que a consulta foi executada.

C#

```
public static IEnumerable<T> LogQuery<T>
    (this IEnumerable<T> sequence, string tag)
{
    // File.AppendText creates a new file if the file doesn't exist.
    using (var writer = File.AppendText("debug.log"))
    {
        writer.WriteLine($"Executing Query {tag}");
    }

    return sequence;
}
```

Você verá um rabisco vermelho sob `File`, que significa que ele não existe. Ele não será compilado, pois o compilador não sabe o que é `File`. Para resolver esse problema, é preciso que você adicione a linha de código a seguir abaixo da primeira linha em `Extensions.cs`:

C#

```
using System.IO;
```

Isso deve resolver o problema e o erro vermelho desaparece.

Em seguida, instrumente a definição de cada consulta com uma mensagem de log:

C#

```
// Program.cs
public static void Main(string[] args)
{
    var startingDeck = (from s in Suits().LogQuery("Suit Generation")
                        from r in Ranks().LogQuery("Rank Generation")
                        select new { Suit = s, Rank = r
}).LogQuery("Starting Deck");

    foreach (var c in startingDeck)
    {
        Console.WriteLine(c);
    }

    Console.WriteLine();
    var times = 0;
    var shuffle = startingDeck;

    do
    {
        // Out shuffle
        /*
        shuffle = shuffle.Take(26)
            .LogQuery("Top Half")
            .InterleaveSequenceWith(shuffle.Skip(26)
            .LogQuery("Bottom Half"))
            .LogQuery("Shuffle");
        */

        // In shuffle
        shuffle = shuffle.Skip(26).LogQuery("Bottom Half")
                .InterleaveSequenceWith(shuffle.Take(26).LogQuery("Top
Half"))
                .LogQuery("Shuffle");

        foreach (var c in shuffle)
        {
            Console.WriteLine(c);
        }

        times++;
        Console.WriteLine(times);
    } while (!startingDeck.SequenceEquals(shuffle));

    Console.WriteLine(times);
}
```

Observe que você não precisa fazer o registro sempre que acessar uma consulta. Você faz o registro ao criar a consulta original. O programa ainda leva muito tempo para ser executado, mas agora você pode ver o motivo. Se você não tiver paciência para executar o embaralhamento interno com o registro em log ativado, volte para o embaralhamento

externo. Você ainda verá os efeitos da avaliação lenta. Em uma execução, ele faz 2592 consultas, incluindo a geração de todos os valores e naipes.

Aqui, você pode melhorar o desempenho do código para reduzir o número de execuções feitas. Uma correção simples possível é *armazenar em cache* os resultados da consulta do LINQ original que constrói o baralho de cartas. Atualmente, você executa as consultas novamente sempre que o loop do-while passa por uma iteração, construindo novamente o baralho de cartas e o embaralhamento de novo todas as vezes. Para armazenar em cache o baralho de cartas, aproveite os métodos LINQ ToArray e ToList; ao anexá-los às consultas, eles executarão as mesmas ações paras quais foram instruídos, mas agora armazenarão os resultados em uma matriz ou lista, dependendo de qual método você optar por chamar. Anexe o método LINQ ToArray às duas consultas e execute o programa novamente:

C#

```
public static void Main(string[] args)
{
    IEnumerable<Suit>? suits = Suits();
    IEnumerable<Rank>? ranks = Ranks();

    if ((suits is null) || (ranks is null))
        return;

    var startingDeck = (from s in suits.LogQuery("Suit Generation")
                        from r in ranks.LogQuery("Value Generation")
                        select new { Suit = s, Rank = r })
                        .LogQuery("Starting Deck")
                        .ToArray();

    foreach (var c in startingDeck)
    {
        Console.WriteLine(c);
    }

    Console.WriteLine();

    var times = 0;
    var shuffle = startingDeck;

    do
    {
        /*
        shuffle = shuffle.Take(26)
            .LogQuery("Top Half")
            .InterleaveSequenceWith(shuffle.Skip(26).LogQuery("Bottom
Half"))
            .LogQuery("Shuffle")
            .ToArray();
        */
```

```
        shuffle = shuffle.Skip(26)
            .LogQuery("Bottom Half")
            .InterleaveSequenceWith(shuffle.Take(26).LogQuery("Top Half"))
            .LogQuery("Shuffle")
            .ToArray();

        foreach (var c in shuffle)
        {
            Console.WriteLine(c);
        }

        times++;
        Console.WriteLine(times);
    } while (!startingDeck.SequenceEquals(shuffle));

    Console.WriteLine(times);
}
```

Agora, o embaralhamento externo contém 30 consultas. Execute novamente com o embaralhamento interno e você verá melhorias semelhantes: agora, executa 162 consultas.

Observe que esse exemplo é **projetado** para realçar os casos de uso em que a avaliação lenta pode causar problemas de desempenho. Embora seja importante ver onde a avaliação lenta pode afetar o desempenho do código, é igualmente importante entender que nem todas as consultas devem ser executadas avidamente. O desempenho incorrido sem usar ToArray ocorre porque cada nova disposição do baralho de cartas é criada com base na disposição anterior. Usar a avaliação lenta significa que cada nova disposição do baralho é criada do baralho original, até mesmo a execução do código que criou o `startingDeck`. Isso causa uma grande quantidade de trabalho extra.

Na prática, alguns algoritmos funcionam bem usando a avaliação detalhada, e outros executam funcionam melhor usando a avaliação lenta. Para o uso diário, a avaliação lenta é uma opção melhor quando a fonte de dados é um processo separado, como um mecanismo de banco de dados. Para os bancos de dados, a avaliação lenta permite que as consultas mais complexas executem apenas uma viagem de ida e volta para o processo de banco de dados e de volta para o restante do seu código. O LINQ é flexível, não importa se você optar por utilizar a avaliação lenta ou detalhada, portanto, meça seus processos e escolha o tipo de avaliação que ofereça o melhor desempenho.

Conclusão

Neste projeto, abordamos:

- o uso de consultas LINQ para agregar dados em uma sequência significativa
- a produção de métodos de Extensão para adicionar nossa própria funcionalidade personalizada a consultas LINQ
- a localização de áreas em nosso código nas quais nossas consultas LINQ podem enfrentar problemas de desempenho, como diminuição da velocidade
- avaliação lenta e detalhada com relação às consultas LINQ, e as implicações que elas podem ter no desempenho da consulta

Além do LINQ, você aprendeu um pouco sobre uma técnica usada por mágicos para truques de carta. Os mágicos usam o embaralhamento Faro porque podem controlar onde cada carta fica no baralho. Agora que você sabe, não conte para os outros!

Para saber mais sobre o LINQ, consulte:

- LINQ (Consulta Integrada à Linguagem)
- Introdução ao LINQ
- Operações de consulta LINQ básica (C#)
- Transformações de dados com LINQ (C#)
- Sintaxe de consulta e sintaxe de método em LINQ (C#)
- funcionalidades do C# que dão suporte a LINQ

LINQ (Consulta Integrada à Linguagem)

19/06/2025

Language-Integrated Query (LINQ) é o nome de um conjunto de tecnologias com base na integração de recursos de consulta diretamente na linguagem C#. Tradicionalmente, as consultas em relação aos dados são expressas como cadeias de caracteres simples sem verificação de tipo em tempo de compilação ou suporte do IntelliSense. Além disso, você precisa aprender uma linguagem de consulta diferente para cada tipo de fonte de dados: bancos de dados SQL, documentos XML, vários serviços Web e assim por diante. Com o LINQ, uma consulta é um constructo de linguagem de primeira classe, assim como classes, métodos e eventos.

Quando você escreve consultas, a parte mais visível "integrada à linguagem" do LINQ é a expressão de consulta. As expressões de consulta são escritas em uma *sintaxe de consulta* declarativa. Usando a sintaxe de consulta, você executa operações de filtragem, ordenação e agrupamento em fontes de dados com um mínimo de código. Use os mesmos padrões de expressão de consulta para consultar e transformar dados de qualquer tipo de fonte de dados.

O exemplo a seguir mostra uma operação de consulta completa. A operação completa inclui a criação de uma fonte de dados, a definição da expressão de consulta e a execução da consulta em uma foreach instrução.

C#

```
// Specify the data source.
int[] scores = [97, 92, 81, 60];

// Define the query expression.
IEnumerable<int> scoreQuery =
    from score in scores
    where score > 80
    select score;

// Execute the query.
foreach (var i in scoreQuery)
{
    Console.Write(i + " ");
}

// Output: 97 92 81
```

Talvez seja necessário adicionar uma using diretiva `using System.Linq;` para que o exemplo anterior seja compilado. As versões mais recentes do .NET fazem uso de importações implícitas para adicionar essa diretiva como uma importação global. Versões mais antigas exigem que você o adicione ao seu código fonte.

Visão geral da expressão de consulta

- As expressões de consulta consultam e transformam dados de qualquer fonte de dados habilitada para LINQ. Por exemplo, uma única consulta pode recuperar dados de um banco de dados SQL e produzir um fluxo XML como saída.
- As expressões de consulta usam muitos constructos de linguagem C# familiares, o que os torna fáceis de ler.
- As variáveis em uma expressão de consulta são todas fortemente tipadas.
- Uma consulta não é executada até que você itere sobre a variável de consulta, por exemplo, em uma instrução `foreach`.
- No tempo de compilação, as expressões de consulta são convertidas em chamadas de método de operador de consulta padrão de acordo com as regras definidas na especificação C#. Qualquer consulta que possa ser expressa usando a sintaxe de consulta também pode ser expressa usando a sintaxe do método. Em alguns casos, a sintaxe de consulta é mais legível e concisa. Em outros, a sintaxe do método é mais legível. Não há diferença semântica ou de desempenho entre as duas formas diferentes. Para obter mais informações, consulte a especificação da linguagem C# e a visão geral dos operadores de consulta Standard.
- Algumas operações de consulta, como Count ou Max, não têm cláusula de expressão de consulta equivalente e, portanto, devem ser expressas como uma chamada de método. A sintaxe do método pode ser combinada com a sintaxe de consulta de várias maneiras.
- As expressões de consulta podem ser compiladas para árvores de expressão ou delegados, dependendo do tipo ao qual a consulta é aplicada. As consultas IEnumerable<T> são compiladas para representantes. As consultas IQueryable e IQueryable<T> são compiladas para árvores de expressão. Para obter mais informações, consulte Árvores de expressão.

Como habilitar a consulta LINQ da fonte de dados

Dados na memória

Há duas maneiras de habilitar a consulta LINQ de dados na memória. Se os dados forem de um tipo que implementa IEnumerable<T>, consulte os dados usando LINQ to Objects. Se não fizer sentido habilitar a enumeração implementando a IEnumerable<T> interface, defina os métodos do operador de consulta padrão LINQ, nesse tipo ou como métodos de extensão para esse tipo. As implementações personalizadas dos operadores de consulta padrão devem usar a execução adiada para retornar os resultados.

Dados remotos

A melhor opção para habilitar a consulta LINQ de uma fonte de dados remota é implementar a IQueryable<T> interface.

Provedores de LINQ do tipo IQueryable

Os provedores LINQ que implementam IQueryable<T> podem variar amplamente em sua complexidade.

Um provedor `IQueryable` menos complexo pode acessar um único método de um serviço Web. Esse tipo de provedor é muito específico porque espera informações específicas nas consultas que ele manipula. Ele tem um sistema de tipo fechado, talvez expondo um único tipo de resultado. A maior parte da execução da consulta ocorre localmente, por exemplo, usando as Enumerable implementações dos operadores de consulta padrão. Um provedor menos complexo pode examinar apenas uma expressão de chamada de método na árvore de expressão que representa a consulta e permitir que a lógica restante da consulta seja tratada em outro lugar.

Um `IQueryable` provedor de complexidade média pode ter como alvo uma fonte de dados que tenha uma linguagem de consulta parcialmente expressiva. Se ele for direcionado a um serviço Web, ele poderá acessar mais de um método do serviço Web e selecionar qual método chamar com base nas informações que a consulta procura. Um provedor de complexidade média teria um sistema de tipo mais rico do que um provedor simples, mas ainda seria um sistema de tipo fixo. Por exemplo, o provedor pode expor tipos que têm relações um para muitos que podem ser percorridas, mas não fornece tecnologia de mapeamento para tipos definidos pelo usuário.

Um provedor complexo `IQueryable` , como o provedor Entity Framework Core , pode traduzir consultas LINQ completas para uma linguagem de consulta expressiva, como SQL. Um provedor complexo é mais geral porque pode lidar com uma variedade maior de perguntas na consulta. Ele também tem um sistema de tipo aberto e, portanto, deve conter uma infraestrutura extensa para mapear tipos definidos pelo usuário. O desenvolvimento de um provedor complexo requer uma quantidade significativa de esforço.

Introdução às consultas LINQ em C#

18/06/2025

Uma *consulta* é uma expressão que recupera dados de uma fonte de dados. Diferentes fontes de dados têm diferentes linguagens de consulta nativas, por exemplo, SQL para bancos de dados relacionais e XQuery para XML. Os desenvolvedores devem aprender uma nova linguagem de consulta para cada tipo de fonte de dados ou formato de dados que eles devem dar suporte. O LINQ simplifica essa situação oferecendo um modelo de linguagem C# consistente para tipos de fontes de dados e formatos. Em uma consulta LINQ, você sempre trabalha com objetos C#. Use os mesmos padrões básicos de codificação para consultar e transformar dados em documentos XML, bancos de dados SQL, coleções .NET e qualquer outro formato quando um provedor LINQ estiver disponível.

Três partes de uma operação de consulta

Todas as operações de consulta LINQ consistem em três ações distintas:

1. Obtenha a fonte de dados.
2. Crie a consulta.
3. Executar a consulta.

O exemplo a seguir mostra como as três partes de uma operação de consulta são expressas no código-fonte. O exemplo usa uma matriz de inteiros como uma fonte de dados para conveniência; no entanto, os mesmos conceitos também se aplicam a outras fontes de dados. Este exemplo é referenciado em todo o restante deste artigo.

C#

```
// The Three Parts of a LINQ Query:
// 1. Data source.
int[] numbers = [ 0, 1, 2, 3, 4, 5, 6 ];

// 2. Query creation.
// numQuery is an IEnumerable<int>
var numQuery = from num in numbers
               where (num % 2) == 0
               select num;

// 3. Query execution.
foreach (int num in numQuery)
{
    Console.Write("{0,1} ", num);
}
```

A ilustração a seguir mostra a operação de consulta completa. No LINQ, a execução da consulta é distinta da consulta em si. Em outras palavras, você não recupera dados criando uma variável de consulta.

A fonte de dados

A fonte de dados no exemplo anterior é uma matriz, que dá suporte à interface genérica IEnumerable<T> . Esse fato significa que ele pode ser consultado com LINQ. Uma consulta é executada em uma `foreach` instrução e `foreach` requer IEnumerable ou IEnumerable<T>. Tipos que dão suporte IEnumerable<T> ou a uma interface derivada, como o genérico IQueryable<T> , são chamados *de tipos queriráveis*.

Um tipo que pode ser consultado não requer nenhuma modificação ou tratamento especial para servir como uma fonte de dados LINQ. Se os dados de origem ainda não estiverem na memória como um tipo queryable, o provedor LINQ deverá representá-los como tal. Por exemplo, LINQ to XML carrega um documento XML em um tipo que pode ser objeto de consulta XElement.

C#

```
// Create a data source from an XML document.
// using System.Xml.Linq;
XElement contacts = XElement.Load(@"c:\myContactList.xml");
```

Com o EntityFramework, você cria um mapeamento relacional de objeto entre classes C# e seu esquema de banco de dados. Você grava suas consultas nos objetos e, em tempo de execução, o EntityFramework manipula a comunicação com o banco de dados. No exemplo a seguir,

`Customers` representa uma tabela específica no banco de dados e o tipo do resultado da consulta, IQueryable<T>deriva de IEnumerable<T>.

```
C#

Northwnd db = new Northwnd(@"c:\northwnd.mdf");

// Query for customers in London.
IQueryable<Customer> custQuery =
    from cust in db.Customers
    where cust.City == "London"
    select cust;
```

Para obter mais informações sobre como criar tipos específicos de fontes de dados, consulte a documentação dos vários provedores LINQ. No entanto, a regra básica é simples: uma fonte de dados LINQ é qualquer objeto que dê suporte à interface genérica IEnumerable<T> ou uma interface que herda dela, normalmente IQueryable<T>.

> ⓘ **Observação**
>
> Tipos como ArrayList que dão suporte à interface não genérica IEnumerable podem também ser usados como uma fonte de dados para LINQ. Para obter mais informações, consulte Como consultar uma ArrayList com LINQ (C#).

A consulta

A consulta especifica quais informações recuperar da fonte de dados ou das fontes. Opcionalmente, uma consulta também especifica como essas informações devem ser classificadas, agrupadas e moldadas antes de serem retornadas. Uma consulta é armazenada em uma variável de consulta e inicializada com uma expressão de consulta. Você usa a sintaxe de consulta C# para escrever consultas.

A consulta no exemplo anterior retorna todos os números par da matriz de inteiros. A expressão de consulta contém três cláusulas: `from`, `where` e `select`. (Se você estiver familiarizado com o SQL, notou que a ordenação das cláusulas é revertida da ordem no SQL.) A `from` cláusula especifica a fonte de dados, a `where` cláusula aplica o filtro e a `select` cláusula especifica o tipo dos elementos retornados. Todas as cláusulas de consulta são discutidas em detalhes nesta seção. Por enquanto, o ponto importante é que, no LINQ, a variável de consulta em si não toma nenhuma ação e não retorna dados. Ele apenas armazena as informações necessárias para produzir os resultados quando a consulta é executada em algum momento posterior. Para obter mais informações sobre como as consultas são construídas, consulte Visão geral dos operadores de consulta padrão (C#).

> **Observação**
>
> As consultas também podem ser expressas usando a sintaxe do método. Para obter mais informações, consulte Sintaxe de consulta e sintaxe de método no LINQ.

Classificação de operadores de consulta padrão por meio de execução

As implementações do LINQ to Objects dos métodos de operador de consulta padrão são executadas de uma das duas maneiras principais: *imediatas* ou *adiadas*. Os operadores de consulta que usam a execução adiada podem ser divididos adicionalmente em duas categorias: *streaming* e *não streaming*.

Imediato

A execução imediata significa que a fonte de dados é lida e a operação é executada uma vez. Todos os operadores de consulta padrão que retornam um resultado escalar são executados imediatamente. Exemplos dessas consultas são `Count`, `Max` e `Average` `First`. Esses métodos são executados sem uma instrução explícita `foreach` porque a consulta em si deve ser usada `foreach` para retornar um resultado. Essas consultas retornam um único valor, não uma coleção `IEnumerable` . Você pode forçar *qualquer* consulta a ser executada imediatamente usando os métodos Enumerable.ToList ou Enumerable.ToArray. A execução imediata fornece reutilização dos resultados da consulta, não da declaração de consulta. Os resultados são recuperados uma vez e, em seguida, armazenados para uso futuro. A consulta a seguir retorna uma contagem dos números pares na matriz de origem:

C#

```
var evenNumQuery = from num in numbers
                   where (num % 2) == 0
                   select num;

int evenNumCount = evenNumQuery.Count();
```

Para forçar a execução imediata de qualquer consulta e armazenar em cache seus resultados, você pode chamar os métodos ToList ou ToArray.

C#

```
List<int> numQuery2 = (from num in numbers
                       where (num % 2) == 0
```

```
                        select num).ToList();

// or like this:
// numQuery3 is still an int[]

var numQuery3 = (from num in numbers
                 where (num % 2) == 0
                 select num).ToArray();
```

Você também pode forçar a execução colocando o `foreach` loop imediatamente após a expressão de consulta. No entanto, ao chamar `ToList` ou `ToArray`, você também armazena todos os dados em um único objeto de coleção.

Adiado

A execução adiada significa que a operação não é executada no ponto no código em que a consulta é declarada. A operação é executada somente quando a variável de consulta é enumerada, por exemplo, usando uma `foreach` instrução. Os resultados da execução da consulta dependem do conteúdo da fonte de dados quando a consulta é executada em vez de quando a consulta é definida. Se a variável de consulta for enumerada várias vezes, os resultados poderão ser diferentes sempre. Quase todos os operadores de consulta padrão cujo tipo de retorno é IEnumerable<T> ou IOrderedEnumerable<TElement> são executados de forma adiada. A execução adiada fornece a facilidade de reutilização da consulta, uma vez que a consulta busca os dados atualizados da fonte de dados sempre que os resultados da consulta são iterados. O código a seguir mostra um exemplo de execução adiada:

C#

```
foreach (int num in numQuery)
{
    Console.Write("{0,1} ", num);
}
```

A `foreach` declaração também é onde os resultados da consulta são obtidos. Por exemplo, na consulta anterior, a variável `num` de iteração contém cada valor (um de cada vez) na sequência retornada.

Como a variável de consulta em si nunca contém os resultados da consulta, você pode executá-la repetidamente para recuperar dados atualizados. Por exemplo, um aplicativo separado pode atualizar um banco de dados continuamente. Em *seu* aplicativo, você pode criar uma consulta que recupera os dados mais recentes e pode executá-los em intervalos para recuperar os resultados atualizados.

Os operadores de consulta que usam a execução adiada podem, adicionalmente, ser classificados como "com fluxo de dados" e "sem fluxo de dados".

Transmissão ao vivo

Os operadores de streaming não precisam ler todos os dados de origem antes de produzirem elementos. No momento da execução, um operador de streaming executa sua operação em cada elemento de origem conforme ele é lido e produz o elemento, se apropriado. Um operador de streaming continua a ler elementos de origem até que um elemento de resultado possa ser produzido. Isso significa que mais de um elemento de origem pode ser lido para produzir um elemento de resultado.

Sem fluxo de dados

Os operadores sem fluxo de dados precisam ler todos os dados da origem antes de gerar um elemento resultante. Operações como classificação ou agrupamento se enquadram nessa categoria. No momento da execução, os operadores de consulta que não são de fluxo leem todos os dados de origem, os colocam em uma estrutura de dados, executam a operação e produzem os elementos resultantes.

Tabela de classificação

A tabela a seguir classifica cada método de operador de consulta padrão de acordo com seu método de execução.

> **Observação**
>
> Se um operador for marcado em duas colunas, duas sequências de entrada estarão envolvidas na operação e cada sequência será avaliada de forma diferente. Nesses casos, é sempre a primeira sequência na lista de parâmetros que é avaliada de maneira adiada e de streaming.

Expandir a tabela

Operador de consulta padrão	Tipo de retorno	Execução imediata	Execução adiada de streaming	Execução sem fluxo de dados adiada
Aggregate	`TSource`	✓		

Operador de consulta padrão	Tipo de retorno	Execução imediata	Execução adiada de streaming	Execução sem fluxo de dados adiada
All	Boolean	✓		
Any	Boolean	✓		
AsEnumerable	IEnumerable<T>		✓	
Average	Valor numérico único	✓		
Cast	IEnumerable<T>		✓	
Concat	IEnumerable<T>		✓	
Contains	Boolean	✓		
Count	Int32	✓		
DefaultIfEmpty	IEnumerable<T>		✓	
Distinct	IEnumerable<T>		✓	
ElementAt	`TSource`	✓		
ElementAtOrDefault	`TSource?`	✓		
Empty	IEnumerable<T>	✓		
Except	IEnumerable<T>		✓	✓
First	`TSource`	✓		
FirstOrDefault	`TSource?`	✓		
GroupBy	IEnumerable<T>			✓
GroupJoin	IEnumerable<T>		✓	✓
Intersect	IEnumerable<T>		✓	✓
Join	IEnumerable<T>		✓	✓
Last	`TSource`	✓		
LastOrDefault	`TSource?`	✓		
LongCount	Int64	✓		
Max	Valor numérico único, `TSource` ou `TResult?`	✓		

Operador de consulta padrão	Tipo de retorno	Execução imediata	Execução adiada de streaming	Execução sem fluxo de dados adiada
Min	Valor numérico único, `TSource` ou `TResult?`	✓		
OfType	IEnumerable<T>		✓	
OrderBy	IOrderedEnumerable<TElement>			✓
OrderByDescending	IOrderedEnumerable<TElement>			✓
Range	IEnumerable<T>		✓	
Repeat	IEnumerable<T>		✓	
Reverse	IEnumerable<T>			✓
Select	IEnumerable<T>		✓	
SelectMany	IEnumerable<T>		✓	
SequenceEqual	Boolean	✓		
Single	`TSource`	✓		
SingleOrDefault	`TSource?`	✓		
Skip	IEnumerable<T>		✓	
SkipWhile	IEnumerable<T>		✓	
Sum	Valor numérico único	✓		
Take	IEnumerable<T>		✓	
TakeWhile	IEnumerable<T>		✓	
ThenBy	IOrderedEnumerable<TElement>			✓
ThenByDescending	IOrderedEnumerable<TElement>			✓
ToArray	`TSource[]` matriz	✓		
ToDictionary	Dictionary<TKey,TValue>	✓		
ToList	IList<T>	✓		
ToLookup	ILookup<TKey,TElement>	✓		
Union	IEnumerable<T>		✓	

Operador de consulta padrão	Tipo de retorno	Execução imediata	Execução adiada de streaming	Execução sem fluxo de dados adiada
Where	IEnumerable<T>		✓	

LINQ para objetos

"LINQ to Objects" refere-se ao uso de consultas LINQ diretamente com qualquer coleção representada por IEnumerable ou IEnumerable<T>. Você pode usar LINQ para consultar quaisquer coleções enumeráveis, como List<T>, Arrayou Dictionary<TKey,TValue>. A coleção pode ser definida pelo usuário ou um tipo retornado por uma API do .NET. Na abordagem LINQ, você escreve um código declarativo que descreve o que deseja recuperar. LINQ to Objects fornece uma ótima introdução à programação com LINQ.

As consultas LINQ oferecem três vantagens principais em relação aos loops tradicionais `foreach` :

- Eles são mais concisos e legíveis, especialmente ao filtrar várias condições.
- Eles fornecem funcionalidades avançadas de filtragem, ordenação e agrupamento com um mínimo de código de aplicativo.
- Eles podem ser portados para outras fontes de dados com pouca ou nenhuma modificação.

Quanto mais complexa a operação que você deseja executar nos dados, mais benefício você percebe usando LINQ em vez de técnicas tradicionais de iteração.

Armazenar os resultados de uma consulta na memória

Uma consulta é basicamente um conjunto de instruções sobre como recuperar e organizar dados. As consultas são executadas de forma preguiçosa, pois cada item subsequente no resultado é solicitado. Quando você usa `foreach` para iterar os resultados, os itens são retornados na ordem de acesso. Para avaliar uma consulta e armazenar seus resultados sem executar um `foreach` loop, basta chamar um dos seguintes métodos na variável de consulta:

- ToList
- ToArray
- ToDictionary
- ToLookup

Você deve atribuir o objeto de coleção retornado a uma nova variável ao armazenar os resultados da consulta, conforme mostrado no exemplo a seguir:

C#

```
List<int> numbers = [ 1, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20 ];

IEnumerable<int> queryFactorsOfFour = from num in numbers
                                      where num % 4 == 0
                                      select num;

// Store the results in a new variable
// without executing a foreach loop.
var factorsofFourList = queryFactorsOfFour.ToList();

// Read and write from the newly created list to demonstrate that it holds data.
Console.WriteLine(factorsofFourList[2]);
factorsofFourList[2] = 0;
Console.WriteLine(factorsofFourList[2]);
```

Consulte também

- Passo a passo: escrevendo consultas em C#
- foreach, in
- LINQ (Palavras-chave de consulta)

Noções básicas sobre expressões de consulta

Artigo • 03/02/2025

Este artigo apresenta os conceitos básicos relacionados a expressões de consulta em C#.

O que é uma consulta e o que ela faz?

Uma *consulta* é um conjunto de instruções que descreve quais dados recuperar de uma determinada fonte de dados (ou fontes) e que forma e organização os dados retornados devem ter. Uma consulta é diferente dos resultados que ela produz.

Em geral, os dados de origem são organizados de forma lógica como uma sequência de elementos do mesmo tipo. Por exemplo, uma tabela de banco de dados SQL contém uma sequência de linhas. Em um arquivo XML, há uma "sequência" de elementos XML (embora eles estejam organizados hierarquicamente em uma estrutura de árvore). Uma coleção na memória contém uma sequência de objetos.

Do ponto de vista do aplicativo, o tipo específico e a estrutura dos dados de origem original não são importantes. O aplicativo sempre vê os dados de origem como uma coleção de IEnumerable<T> ou de IQueryable<T>. Por exemplo, no LINQ to XML, os dados de origem ficam visíveis como um `IEnumerable`<XElement>.

Dada essa sequência de origem, uma consulta pode executar uma das três ações:

- Recuperar um subconjunto dos elementos para produzir uma nova sequência sem modificar os elementos individuais. A consulta pode, em seguida, classificar ou agrupar a sequência retornada de várias maneiras, conforme mostrado no exemplo a seguir (suponha que `scores` é um `int[]`):

  C#

  ```
  IEnumerable<int> highScoresQuery =
      from score in scores
      where score > 80
      orderby score descending
      select score;
  ```

- Recuperar uma sequência de elementos, como no exemplo anterior, mas transformá-los em um novo tipo de objeto. Por exemplo, uma consulta pode recuperar apenas os nomes de família de determinados registros de clientes em

uma fonte de dados. Ou pode recuperar o registro completo e, em seguida, usá-lo para construir outro tipo de objeto na memória ou até mesmo dados XML antes de gerar a sequência de resultados final. O exemplo a seguir mostra uma projeção de um `int` para um `string`. Observe o novo tipo de `highScoresQuery`.

```
C#

IEnumerable<string> highScoresQuery2 =
    from score in scores
    where score > 80
    orderby score descending
    select $"The score is {score}";
```

- Recuperar um valor singleton sobre os dados de origem, como:
  - O número de elementos que correspondem a uma determinada condição.
  - O elemento que tem o maior ou menor valor.
  - O primeiro elemento que corresponde a uma condição ou a soma dos valores específicos em um conjunto de elementos especificado. Por exemplo, a consulta a seguir retorna o número de notas maiores que 80 da matriz de inteiros `scores`.

    ```
    C#

    var highScoreCount = (
        from score in scores
        where score > 80
        select score
    ).Count();
    ```

    No exemplo anterior, observe o uso de parênteses ao redor da expressão de consulta antes da chamada para o método Enumerable.Count. Você também pode usar uma nova variável para armazenar o resultado concreto.

    ```
    C#

    IEnumerable<int> highScoresQuery3 =
        from score in scores
        where score > 80
        select score;

    var scoreCount = highScoresQuery3.Count();
    ```

No exemplo anterior, a consulta é executada na chamada para `Count`, pois `Count` deve iterar os resultados para determinar o número de elementos retornados por `highScoresQuery`.

O que é uma expressão de consulta?

Uma *expressão de consulta* é uma consulta expressada na sintaxe da consulta. Uma expressão de consulta é um constructo de linguagem de primeira classe. É exatamente como qualquer outra expressão e pode ser usada em qualquer contexto em que uma expressão C# seja válida. Uma expressão de consulta consiste em um conjunto de cláusulas escritas em uma sintaxe declarativa semelhante ao SQL ou XQuery. Cada cláusula, por sua vez, contém uma ou mais expressões C# e essas expressões podem ser ou conter uma expressão de consulta.

Uma expressão de consulta deve começar com uma cláusula from e deve terminar com uma cláusula select ou group. Entre a primeira cláusula `from` e a última cláusula `select` ou `group`, ela pode conter uma ou mais dessas cláusulas opcionais: where, orderby, join, let e até mesmo outras cláusulas from. Você também pode usar a palavra-chave into para habilitar o resultado de uma cláusula `join` ou `group` para servir como fonte para mais cláusulas de consulta na mesma expressão de consulta.

Variável da consulta

Em LINQ, uma variável de consulta é qualquer variável que armazena uma *consulta* em vez dos *resultados* de uma consulta. Mais especificamente, uma variável de consulta é sempre um tipo enumerável que produzirá uma sequência de elementos quando for iterada em uma instrução `foreach` ou uma chamada direta para seu método IEnumerator.MoveNext().

> **Observação**
>
> Os exemplos neste artigo usam a fonte de dados e os dados de exemplo a seguir.

C#

```
record City(string Name, long Population);
record Country(string Name, double Area, long Population, List<City>
Cities);
record Product(string Name, string Category);
```

C#

```
static readonly City[] cities = [
    new City("Tokyo", 37_833_000),
    new City("Delhi", 30_290_000),
    new City("Shanghai", 27_110_000),
    new City("São Paulo", 22_043_000),
    new City("Mumbai", 20_412_000),
    new City("Beijing", 20_384_000),
    new City("Cairo", 18_772_000),
    new City("Dhaka", 17_598_000),
    new City("Osaka", 19_281_000),
    new City("New York-Newark", 18_604_000),
    new City("Karachi", 16_094_000),
    new City("Chongqing", 15_872_000),
    new City("Istanbul", 15_029_000),
    new City("Buenos Aires", 15_024_000),
    new City("Kolkata", 14_850_000),
    new City("Lagos", 14_368_000),
    new City("Kinshasa", 14_342_000),
    new City("Manila", 13_923_000),
    new City("Rio de Janeiro", 13_374_000),
    new City("Tianjin", 13_215_000)
];

static readonly Country[] countries = [
    new Country ("Vatican City", 0.44, 526, [new City("Vatican City",
826)]),
    new Country ("Monaco", 2.02, 38_000, [new City("Monte Carlo", 38_000)]),
    new Country ("Nauru", 21, 10_900, [new City("Yaren", 1_100)]),
    new Country ("Tuvalu", 26, 11_600, [new City("Funafuti", 6_200)]),
    new Country ("San Marino", 61, 33_900, [new City("San Marino", 4_500)]),
    new Country ("Liechtenstein", 160, 38_000, [new City("Vaduz", 5_200)]),
    new Country ("Marshall Islands", 181, 58_000, [new City("Majuro",
28_000)]),
    new Country ("Saint Kitts & Nevis", 261, 53_000, [new City("Basseterre",
13_000)])
];
```

O exemplo de código a seguir mostra uma expressão de consulta simples com uma fonte de dados, uma cláusula de filtragem, uma cláusula de ordenação e nenhuma transformação dos elementos de origem. A cláusula `select` termina a consulta.

C#

```
// Data source.
int[] scores = [90, 71, 82, 93, 75, 82];

// Query Expression.
IEnumerable<int> scoreQuery = //query variable
    from score in scores //required
    where score > 80 // optional
```

```
    orderby score descending // optional
    select score; //must end with select or group

// Execute the query to produce the results
foreach (var testScore in scoreQuery)
{
    Console.WriteLine(testScore);
}

// Output: 93 90 82 82
```

No exemplo anterior, `scoreQuery` é uma *variável de consulta*, o que às vezes é chamado apenas de uma *consulta*. A variável de consulta não armazena nenhum dado de resultado real, que é produzido no loop `foreach`. E quando a instrução `foreach` é executada, os resultados da consulta não são retornados pela variável de consulta `scoreQuery`. Em vez disso, eles são retornados pela variável de iteração `testScore`. A variável `scoreQuery` pode ser iterada em um segundo loop `foreach`. Ele produz os mesmos resultados, desde que nem ele nem a fonte de dados tenham sido modificados.

Uma variável de consulta pode armazenar uma consulta que é expressa na sintaxe de consulta ou na sintaxe de método ou uma combinação das duas. Nos exemplos a seguir, `queryMajorCities` e `queryMajorCities2` são variáveis de consulta:

C#

```
City[] cities = [
    new City("Tokyo", 37_833_000),
    new City("Delhi", 30_290_000),
    new City("Shanghai", 27_110_000),
    new City("São Paulo", 22_043_000)
];

//Query syntax
IEnumerable<City> queryMajorCities =
    from city in cities
    where city.Population > 30_000_000
    select city;

// Execute the query to produce the results
foreach (City city in queryMajorCities)
{
    Console.WriteLine(city);
}

// Output:
// City { Name = Tokyo, Population = 37833000 }
// City { Name = Delhi, Population = 30290000 }

// Method-based syntax
IEnumerable<City> queryMajorCities2 = cities.Where(c => c.Population >
```

```
30_000_000);
// Execute the query to produce the results
foreach (City city in queryMajorCities2)
{
    Console.WriteLine(city);
}
// Output:
// City { Name = Tokyo, Population = 37833000 }
// City { Name = Delhi, Population = 30290000 }
```

Por outro lado, os dois exemplos a seguir mostram variáveis que não são variáveis de consulta, embora sejam inicializadas com uma consulta. Elas não são variáveis de consulta porque armazenam resultados:

C#

```
var highestScore = (
    from score in scores
    select score
).Max();

// or split the expression
IEnumerable<int> scoreQuery =
    from score in scores
    select score;

var highScore = scoreQuery.Max();
// the following returns the same result
highScore = scores.Max();
```

C#

```
var largeCitiesList = (
    from country in countries
    from city in country.Cities
    where city.Population > 10000
    select city
).ToList();

// or split the expression
IEnumerable<City> largeCitiesQuery =
    from country in countries
    from city in country.Cities
    where city.Population > 10000
    select city;
var largeCitiesList2 = largeCitiesQuery.ToList();
```

Tipagem explícita e implícita de variáveis de consulta

Esta documentação normalmente fornece o tipo explícito da variável de consulta para mostrar a relação de tipo entre a variável de consulta e a cláusula select. No entanto, você também pode usar a palavra-chave var para instruir o compilador a inferir o tipo de uma variável de consulta (ou qualquer outra variável local) em tempo de compilação. Por exemplo, o exemplo de consulta que foi mostrado anteriormente neste artigo também pode ser expressado usando a tipagem implícita:

```
C#

var queryCities =
    from city in cities
    where city.Population > 100000
    select city;
```

No exemplo anterior, o uso de "var" é opcional. `queryCities` é um tipo `IEnumerable<City>` digitado implícita ou explicitamente.

Iniciando uma expressão de consulta

Uma expressão de consulta deve começar com uma cláusula `from`. Especifica uma fonte de dados junto com uma variável de intervalo. A variável de intervalo representa cada elemento sucessivo na sequência de origem como a sequência de origem que está sendo percorrida. A variável de intervalo é definida com rigor com base no tipo de elementos na fonte de dados. No exemplo a seguir, como `countries` é uma matriz de objetos `Country`, a variável de intervalo também é tipada como `Country`. Como a variável de intervalo é fortemente tipada, você pode usar o operador ponto para acessar todos os membros disponíveis do tipo.

```
C#

IEnumerable<Country> countryAreaQuery =
    from country in countries
    where country.Area > 20 //sq km
    select country;
```

A variável de intervalo está no escopo até a consulta ser encerrada com ponto e vírgula ou com uma cláusula continuation.

Uma expressão de consulta pode conter várias cláusulas `from`. Use mais cláusulas `from` quando cada elemento na sequência de origem for ele próprio uma coleção ou contiver uma coleção. Por exemplo, suponha que você tem uma coleção de objetos `Country` e cada um dos quais contém uma coleção de objetos `City` chamada `Cities`. Para

consular os objetos `City` em cada `Country`, use duas cláusulas `from` como mostrado aqui:

```
C#

IEnumerable<City> cityQuery =
    from country in countries
    from city in country.Cities
    where city.Population > 10000
    select city;
```

Para obter mais informações, consulte Cláusula from.

Encerrando uma expressão de consulta

Uma expressão de consulta deve ser encerrada com uma cláusula `group` ou uma cláusula `select`.

A cláusula de grupo

Use a cláusula `group` para produzir uma sequência de grupos organizada por uma chave que você especificar. A chave pode ter qualquer tipo de dados. Por exemplo, a consulta a seguir cria uma sequência de grupos que contém um ou mais objetos `Country` e cuja chave é um tipo `char` com o valor sendo a primeira letra dos nomes dos países.

```
C#

var queryCountryGroups =
    from country in countries
    group country by country.Name[0];
```

Para obter mais informações sobre agrupamento, consulte a cláusula do grupo .

Cláusula select

Use a cláusula `select` para produzir todos os outros tipos de sequências. Uma cláusula `select` simples produz apenas uma sequência do mesmo tipo dos objetos contidos na fonte de dados. Neste exemplo, a fonte de dados contém objetos `Country`. A cláusula `orderby` simplesmente classifica os elementos em uma nova ordem e a cláusula `select` produz uma sequência dos objetos `Country` reordenados.

```C#
IEnumerable<Country> sortedQuery =
    from country in countries
    orderby country.Area
    select country;
```

A cláusula `select` pode ser usada para transformar dados de origem em sequências de novos tipos. Essa transformação também é denominada projeção de . No exemplo a seguir, a cláusula `select` *projeta* uma sequência de tipos anônimos que contém apenas um subconjunto dos campos do elemento original. Os novos objetos são inicializados usando um inicializador de objeto.

```C#
var queryNameAndPop =
    from country in countries
    select new
    {
        Name = country.Name,
        Pop = country.Population
    };
```

Portanto, nesse exemplo, o var é obrigatório porque a consulta produz um tipo anônimo.

Para obter mais informações sobre todas as maneiras que uma cláusula `select` pode ser usada para transformar os dados de origem, consulte Cláusula select.

Continuações com *into*

Você pode usar a palavra-chave `into` em uma cláusula `select` ou `group` para criar um identificador temporário que armazena uma consulta. Use a cláusula `into` quando precisar executar operações de consulta adicionais em uma consulta após a operação de agrupamento ou seleção. No exemplo a seguir, `countries` são agrupados de acordo com a população em intervalos de 10 milhões. Depois que esses grupos são criados, mais cláusulas filtram alguns grupos e, em seguida, os grupos são classificados em ordem crescente. Para executar essas operações extras, a continuação representada por `countryGroup` é necessária.

```C#
// percentileQuery is an IEnumerable<IGrouping<int, Country>>
var percentileQuery =
```

```
    from country in countries
    let percentile = (int)country.Population / 1_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);
    }
}
```

Para obter mais informações, consulte into.

Filtragem, ordenação e junção

Entre a cláusula `from` inicial e a cláusula `select` ou `group` final, todas as outras cláusulas (`where`, `join`, `orderby`, `from`, `let`) são opcionais. Todas as cláusulas opcionais podem ser usadas várias vezes ou nenhuma vez no corpo de uma consulta.

A cláusula where

Use a cláusula `where` para filtrar os elementos dos dados de origem com base em uma ou mais expressões de predicado. A cláusula `where` no exemplo a seguir tem um predicado com duas condições.

C#

```
IEnumerable<City> queryCityPop =
    from city in cities
    where city.Population is < 15_000_000 and > 10_000_000
    select city;
```

Para obter mais informações, consulte Cláusula where.

A cláusula orderby

Use a cláusula `orderby` para classificar os resultados em ordem crescente ou decrescente. Você também pode especificar ordens de classificação secundárias. O exemplo a seguir executa uma classificação primária nos objetos `country` usando a

propriedade `Area`. Em seguida, ele executa a classificação secundária usando a propriedade `Population`.

```
C#

IEnumerable<Country> querySortedCountries =
    from country in countries
    orderby country.Area, country.Population descending
    select country;
```

A palavra-chave `ascending` é opcional. Será a ordem de classificação padrão se nenhuma ordem for especificada. Para obter mais informações, consulte Cláusula orderby.

A cláusula de junção

Use a cláusula `join` para associar e/ou combinar elementos de uma fonte de dados com elementos de outra fonte de dados com base em uma comparação de igualdade entre as chaves especificadas em cada elemento. Na LINQ, as operações join são executadas em sequências de objetos cujos elementos são de tipos diferentes. Após ter unido duas sequências, você deve usar uma instrução `select` ou `group` para especificar qual elemento armazenar na sequência de saída. Você também pode usar um tipo anônimo para combinar propriedades de cada conjunto de elementos associados em um novo tipo para a sequência de saída. O exemplo a seguir associa objetos `prod` cuja propriedade `Category` corresponde a uma das categorias na matriz de cadeias de caracteres `categories`. Produtos cuja `Category` não corresponda a nenhuma cadeia de caracteres em `categories` são filtrados. A instrução `select` projeta um novo tipo cujas propriedades são tiradas de `cat` e `prod`.

```
C#

var categoryQuery =
    from cat in categories
    join prod in products on cat equals prod.Category
    select new
    {
        Category = cat,
        Name = prod.Name
    };
```

Você também pode executar uma junção de grupo armazenando os resultados da operação `join` em uma variável temporária usando a palavra-chave into. Para obter mais informações, consulte Cláusula join.

A cláusula let

Use a cláusula `let` para armazenar o resultado de uma expressão, como uma chamada de método, em uma nova variável de intervalo. No exemplo a seguir, a variável de intervalo `firstName` armazena o primeiro elemento da matriz de cadeias de caracteres retornado pelo `Split`.

C#

```
string[] names = ["Svetlana Omelchenko", "Claire O'Donnell", "Sven
Mortensen", "Cesar Garcia"];
IEnumerable<string> queryFirstNames =
    from name in names
    let firstName = name.Split(' ')[0]
    select firstName;

foreach (var s in queryFirstNames)
{
    Console.Write(s + " ");
}

//Output: Svetlana Claire Sven Cesar
```

Para obter mais informações, consulte Cláusula let.

Subconsultas em uma expressão de consulta

Uma cláusula de consulta pode conter uma expressão de consulta, que às vezes é chamada de *subconsulta*. Cada subconsulta começa com sua própria cláusula `from` que não necessariamente aponta para a mesma fonte de dados na primeira cláusula `from`. Por exemplo, a consulta a seguir mostra uma expressão de consulta que é usada na instrução select para recuperar os resultados de uma operação de agrupamento.

C#

```
var queryGroupMax =
    from student in students
    group student by student.Year into studentGroup
    select new
    {
        Level = studentGroup.Key,
        HighestScore = (
            from student2 in studentGroup
            select student2.ExamScores.Average()
        ).Max()
    };
```

Para obter mais informações, confira Executar uma subconsulta em uma operação de agrupamento.

Confira também

- Palavras-chave de consulta (LINQ)
- Visão geral de operadores de consulta padrão

Gravar consultas LINQ em C# para consultar dados

Artigo • 22/01/2025

A maioria das consultas na documentação introdutória do LINQ (Consulta Integrada à Linguagem) é escrita com o uso da sintaxe de consulta declarativa do LINQ. O compilador C# converte a sintaxe da consulta em chamadas de método. Essas chamadas de método implementam os operadores de consulta padrão e têm nomes como `Where`, `Select`, `GroupBy`, `Join`, `Max` e `Average`. Você pode chamá-los diretamente usando a sintaxe de método em vez da sintaxe de consulta.

A sintaxe de consulta e a sintaxe do método são semanticamente idênticas, mas a sintaxe de consulta geralmente é mais simples e fácil de ler. Algumas consultas devem ser expressadas como chamadas de método. Por exemplo, você deve usar uma chamada de método para expressar uma consulta que recupera o número de elementos que correspondem a uma condição especificada. Você também deve usar uma chamada de método para uma consulta que recupera o elemento que tem o valor máximo em uma sequência de origem. A documentação de referência para os operadores de consulta padrão no namespace System.Linq geralmente usa a sintaxe de método. Você deve se familiarizar com como usar a sintaxe do método em consultas e nas próprias expressões de consulta.

Métodos de extensão do operador de consulta padrão

O exemplo a seguir mostra uma *expressão de consulta* simples e a consulta semanticamente equivalente escrita como uma *consulta baseada em método*.

C#

```
int[] numbers = [ 5, 10, 8, 3, 6, 12 ];

//Query syntax:
IEnumerable<int> numQuery1 =
    from num in numbers
    where num % 2 == 0
    orderby num
    select num;

//Method syntax:
IEnumerable<int> numQuery2 = numbers
    .Where(num => num % 2 == 0)
```

```
    .OrderBy(n => n);

foreach (int i in numQuery1)
{
    Console.Write(i + " ");
}
Console.WriteLine(System.Environment.NewLine);
foreach (int i in numQuery2)
{
    Console.Write(i + " ");
}
```

A saída dos dois exemplos é idêntica. O tipo da variável de consulta é o mesmo em ambas as formas: IEnumerable<T>.

No lado direito da expressão, observe que a cláusula `where` agora é expressa como um método de instância no objeto `numbers`, que tem um tipo de `IEnumerable<int>`. Se você estiver familiarizado com a interface genérica IEnumerable<T>, você saberá que ela não possui um método `Where`. No entanto, se você invocar a lista de conclusão do IntelliSense no IDE do Visual Studio, verá não apenas um método `Where`, mas muitos outros métodos, como `Select`, `SelectMany`, `Join` e `Orderby`. Esses métodos implementam os operadores de consulta padrão.

Embora pareça que IEnumerable<T> inclui mais métodos, ele não inclui. Os operadores de consulta padrão são implementados como *métodos de extensão*. Métodos de extensão "estendem" um tipo existente, eles podem ser chamados como se fossem métodos de instância no tipo. Os operadores de consulta padrão estendem IEnumerable<T> e que é por esse motivo que você pode escrever `numbers.Where(...)`. Você coloca as extensões no escopo com as diretivas `using` antes de chamá-las.

Para obter mais informações sobre os métodos de extensão, consulte Métodos de extensão. Para obter mais informações sobre os operadores de consulta padrão, consulte Visão geral de operadores de consulta padrão (C#). Alguns provedores LINQ,

como Entity Framework e LINQ to XML, implementam seus próprios operadores de consulta padrão e métodos de extensão para outros tipos além de IEnumerable<T>.

Expressões lambda

No exemplo anterior, a expressão condicional (`num % 2 == 0`) é passada como um argumento em linha para o método Enumerable.Where: `Where(num => num % 2 == 0).` Essa expressão embutida é uma expressão lambda . É uma maneira conveniente de escrever código que, de outra forma, teria que ser escrito de forma mais complicada. O `num` à esquerda do operador é a variável de entrada, que corresponde a `num` na expressão de consulta. O compilador pode inferir o tipo de `num` porque ele sabe que `numbers` é um tipo IEnumerable<T> genérico. O corpo do lambda é o mesmo que a expressão na sintaxe da consulta ou em qualquer outra expressão ou instrução C#. Ele pode incluir chamadas de método e outras lógicas complexas. O valor retornado é o resultado da expressão. Determinadas consultas só podem ser expressas na sintaxe do método e algumas dessas consultas exigem expressões lambda. As expressões lambda são uma ferramenta poderosa e flexível em sua caixa de ferramentas LINQ.

Possibilidade de composição das consultas

No exemplo de código anterior, o método Enumerable.OrderBy é invocado usando o operador de ponto na chamada para `Where`. `Where` produz uma sequência filtrada e, em seguida, `Orderby` classifica a sequência produzida por `Where`. Como as consultas retornam uma `IEnumerable`, você pode escrevê-las na sintaxe de método encadeando as chamadas de método. O compilador faz essa composição quando você escreve consultas usando a sintaxe de consulta. Como uma variável de consulta não armazena os resultados da consulta, você pode modificá-la ou usá-la como base para uma nova consulta a qualquer momento, mesmo depois de executá-la.

Os exemplos a seguir demonstram algumas consultas LINQ básicas usando cada abordagem listada anteriormente.

> **Observação**
>
> Essas consultas operam em coleções na memória; no entanto, a sintaxe é idêntica à usada em LINQ to Entities e LINQ to XML.

Exemplo – sintaxe de consulta

Você escreve a maioria das consultas com *sintaxe de consulta* para criar *expressões de consulta*. O exemplo a seguir mostra três expressões de consulta. A primeira expressão de consulta demonstra como filtrar ou restringir os resultados aplicando condições com uma cláusula `where`. Ela retorna todos os elementos na sequência de origem cujos valores são maiores que 7 ou menores que 3. A segunda expressão demonstra como ordenar os resultados retornados. A terceira expressão demonstra como agrupar resultados de acordo com uma chave. Esta consulta retorna dois grupos com base na primeira letra da palavra.

C#

```
List<int> numbers = [ 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 ];

// The query variables can also be implicitly typed by using var

// Query #1.
IEnumerable<int> filteringQuery =
    from num in numbers
    where num is < 3 or > 7
    select num;

// Query #2.
IEnumerable<int> orderingQuery =
    from num in numbers
    where num is < 3 or > 7
    orderby num ascending
    select num;

// Query #3.
string[] groupingQuery = ["carrots", "cabbage", "broccoli", "beans",
"barley"];
IEnumerable<IGrouping<char, string>> queryFoodGroups =
    from item in groupingQuery
    group item by item[0];
```

O tipo das consultas é IEnumerable<T>. Todas essas consultas poderiam ser escritas usando var conforme mostrado no exemplo a seguir:

`var query = from num in numbers...`

Em cada exemplo anterior, as consultas não são realmente executadas até que você itere sobre a variável de consulta em uma instrução foreach ou outra instrução.

Exemplo – sintaxe de método

Algumas operações de consulta devem ser expressas como uma chamada de método. Os métodos mais comuns são aqueles que retornam valores numéricos singleton, como

Sum, Max, Min, Average, e assim por diante. Esses métodos sempre devem ser chamados por último em qualquer consulta porque retornam um único valor e não podem servir como origem para uma operação de consulta adicional. O exemplo a seguir mostra uma chamada de método em uma expressão de consulta:

```C#
List<int> numbers1 = [ 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 ];
List<int> numbers2 = [ 15, 14, 11, 13, 19, 18, 16, 17, 12, 10 ];

// Query #4.
double average = numbers1.Average();

// Query #5.
IEnumerable<int> concatenationQuery = numbers1.Concat(numbers2);
```

Se o método tiver parâmetros System.Action ou System.Func<TResult>, esses argumentos serão fornecidos na forma de uma expressão lambda, conforme mostrado no exemplo a seguir:

```C#
// Query #6.
IEnumerable<int> largeNumbersQuery = numbers2.Where(c => c > 15);
```

Nas consultas anteriores, somente a Consulta nº 4 é executada imediatamente, pois retorna um único valor e não uma coleção genérica IEnumerable<T>. O método em si usa `foreach` ou código semelhante para calcular seu valor.

Cada uma das consultas anteriores pode ser escrita usando a tipagem implícita com var, conforme mostrado nos seguintes exemplos:

```C#
// var is used for convenience in these queries
double average = numbers1.Average();
var concatenationQuery = numbers1.Concat(numbers2);
var largeNumbersQuery = numbers2.Where(c => c > 15);
```

Exemplo – sintaxe mista de consulta e do método

Este exemplo mostra como usar a sintaxe do método nos resultados de uma cláusula de consulta. Simplesmente coloque a expressão de consulta entre parênteses e, em

seguida, aplique o operador de ponto e chame o método. No exemplo a seguir, a Query #7 retorna uma contagem dos números cujo valor está entre 3 e 7.

```C#
// Query #7.

// Using a query expression with method syntax
var numCount1 = (
    from num in numbers1
    where num is > 3 and < 7
    select num
).Count();

// Better: Create a new variable to store
// the method call result
IEnumerable<int> numbersQuery =
    from num in numbers1
    where num is > 3 and < 7
    select num;

var numCount2 = numbersQuery.Count();
```

Como a Query #7 retorna um único valor e não uma coleção, a consulta é executada imediatamente.

A consulta anterior pode ser escrita usando a tipagem implícita com `var`, da seguinte maneira:

```C#
var numCount = (from num in numbers...
```

Ela pode ser escrita na sintaxe de método da seguinte maneira:

```C#
var numCount = numbers.Count(n => n is > 3 and < 7);
```

Ela pode ser escrita usando a tipagem explícita da seguinte maneira:

```C#
int numCount = numbers.Count(n => n is > 3 and < 7);
```

Especificar filtros de predicado dinamicamente em tempo de execução

Em alguns casos, você não sabe até o tempo de execução quantos predicados precisa aplicar aos elementos de origem na cláusula `where`. Uma maneira de especificar dinamicamente vários filtros de predicados é usar o método Contains, conforme mostrado no exemplo a seguir. A consulta retorna resultados diferentes com base no valor de `id` quando a consulta é executada.

C#

```
int[] ids = [ 111, 114, 112 ];

var queryNames = from student in students
                 where ids.Contains(student.ID)
                 select new
                 {
                     student.LastName,
                     student.ID
                 };

foreach (var name in queryNames)
{
    Console.WriteLine($"{name.LastName}: {name.ID}");
}

/* Output:
    Garcia: 114
    O'Donnell: 112
    Omelchenko: 111
 */

// Change the ids.
ids = [ 122, 117, 120, 115 ];

// The query will now return different results
foreach (var name in queryNames)
{
    Console.WriteLine($"{name.LastName}: {name.ID}");
}

/* Output:
    Adams: 120
    Feng: 117
    Garcia: 115
    Tucker: 122
 */
```

Observação

Este exemplo usa a seguinte fonte e dados:

C#

```
record City(string Name, long Population);
record Country(string Name, double Area, long Population, List<City>
Cities);
record Product(string Name, string Category);
```

C#

```
static readonly City[] cities = [
    new City("Tokyo", 37_833_000),
    new City("Delhi", 30_290_000),
    new City("Shanghai", 27_110_000),
    new City("São Paulo", 22_043_000),
    new City("Mumbai", 20_412_000),
    new City("Beijing", 20_384_000),
    new City("Cairo", 18_772_000),
    new City("Dhaka", 17_598_000),
    new City("Osaka", 19_281_000),
    new City("New York-Newark", 18_604_000),
    new City("Karachi", 16_094_000),
    new City("Chongqing", 15_872_000),
    new City("Istanbul", 15_029_000),
    new City("Buenos Aires", 15_024_000),
    new City("Kolkata", 14_850_000),
    new City("Lagos", 14_368_000),
    new City("Kinshasa", 14_342_000),
    new City("Manila", 13_923_000),
    new City("Rio de Janeiro", 13_374_000),
    new City("Tianjin", 13_215_000)
];

static readonly Country[] countries = [
    new Country ("Vatican City", 0.44, 526, [new City("Vatican City",
826)]),
    new Country ("Monaco", 2.02, 38_000, [new City("Monte Carlo", 38_000)]),
    new Country ("Nauru", 21, 10_900, [new City("Yaren", 1_100)]),
    new Country ("Tuvalu", 26, 11_600, [new City("Funafuti", 6_200)]),
    new Country ("San Marino", 61, 33_900, [new City("San Marino", 4_500)]),
    new Country ("Liechtenstein", 160, 38_000, [new City("Vaduz", 5_200)]),
    new Country ("Marshall Islands", 181, 58_000, [new City("Majuro",
28_000)]),
    new Country ("Saint Kitts & Nevis", 261, 53_000, [new City("Basseterre",
13_000)])
];
```

Você pode usar instruções de fluxo de controle, como `if... else` ou `switch`, para selecionar entre consultas alternativas predeterminadas. No exemplo a seguir,

`studentQuery` usará outra cláusula `where` se o valor do runtime de `oddYear` é `true` ou `false`.

C#

```
void FilterByYearType(bool oddYear)
{
    IEnumerable<Student> studentQuery = oddYear
        ? (from student in students
           where student.Year is GradeLevel.FirstYear or
GradeLevel.ThirdYear
           select student)
        : (from student in students
           where student.Year is GradeLevel.SecondYear or
GradeLevel.FourthYear
           select student);
    var descr = oddYear ? "odd" : "even";
    Console.WriteLine($"The following students are at an {descr} year
level:");
    foreach (Student name in studentQuery)
    {
        Console.WriteLine($"{name.LastName}: {name.ID}");
    }
}

FilterByYearType(true);

/* Output:
    The following students are at an odd year level:
    Fakhouri: 116
    Feng: 117
    Garcia: 115
    Mortensen: 113
    Tucker: 119
    Tucker: 122
 */

FilterByYearType(false);

/* Output:
    The following students are at an even year level:
    Adams: 120
    Garcia: 114
    Garcia: 118
    O'Donnell: 112
    Omelchenko: 111
    Zabokritski: 121
 */
```

Manipular valores nulos em expressões de consulta

Este exemplo mostra como tratar os possíveis valores nulos em coleções de origem. Uma coleção de objetos, tal como uma IEnumerable<T>, pode conter elementos cujo valor é null. Se uma coleção de origem for `null` ou contiver um elemento cujo valor seja `null` e a consulta não lidar com valores `null`, uma NullReferenceException será gerada ao executar a consulta.

O exemplo a seguir usa esses tipos e matrizes de dados estáticos:

C#

```
record Product(string Name, int CategoryID);
record Category(string Name, int ID);
```

C#

```
static Category?[] categories =
[
    new ("brass", 1),
    null,
    new ("winds", 2),
    default,
    new ("percussion", 3)
];

static Product?[] products =
[
    new Product("Trumpet", 1),
    new Product("Trombone", 1),
    new Product("French Horn", 1),
    null,
    new Product("Clarinet", 2),
    new Product("Flute", 2),
    null,
    new Product("Cymbal", 3),
    new Product("Drum", 3)
];
```

Você pode escrever o código defensivamente para evitar uma exceção de referência nula conforme mostrado no exemplo a seguir:

C#

```
var query1 = from c in categories
             where c != null
```

```
            join p in products on c.ID equals p?.CategoryID
            select new
            {
                Category = c.Name,
                Name = p.Name
            };
```

No exemplo anterior, a cláusula `where` filtra todos os elementos nulos na sequência de categorias. Essa técnica é independente da verificação de nulos na cláusula join. A expressão condicional com null nesse exemplo funciona porque `Products.CategoryID` é do tipo `int?`, que é uma abreviação para `Nullable<int>`.

Em uma cláusula join, se apenas uma das chaves de comparação for um tipo de valor anulável que, você pode converter a outra para um tipo de valor anulável na expressão de consulta. No exemplo a seguir, suponha que `EmployeeID` é uma coluna que contém os valores do tipo `int?`:

C#

```
var query =
    from o in db.Orders
    join e in db.Employees
        on o.EmployeeID equals (int?)e.EmployeeID
    select new { o.OrderID, e.FirstName };
```

Em cada um dos exemplos, a palavra-chave de consulta `equals` é usada. Use também padrões correspondentes, que incluem padrões para `is null` e `is not null`. Esses padrões não são recomendados em consultas LINQ porque os provedores de consulta podem não interpretar a nova sintaxe C# corretamente. Um provedor de consultas é uma biblioteca que converte expressões de consulta C# em um formato de dados nativo, como o Entity Framework Core. Os provedores de consulta implementam a interface System.Linq.IQueryProvider para criar fontes de dados que implementam a interface System.Linq.IQueryable<T>.

Tratar exceções em expressões de consulta

É possível chamar qualquer método no contexto de uma expressão de consulta. Não chame qualquer método em uma expressão de consulta que possa criar um efeito colateral, como modificar o conteúdo da fonte de dados ou gerar uma exceção. Este exemplo mostra como evitar exceções ao chamar métodos em uma expressão de consulta, sem violar as diretrizes gerais sobre tratamento de exceção do .NET. Essas diretrizes afirmam que é aceitável capturar uma exceção específica quando você

entende por que ela foi lançada em um determinado contexto. Para obter mais informações, consulte Melhores práticas para exceções.

O último exemplo mostra como tratar os casos em que é necessário lançar uma exceção durante a execução de uma consulta.

O exemplo a seguir mostra como mover o código de tratamento de exceção para fora de uma expressão de consulta. Essa refatoração só é possível quando o método não depende de nenhuma variável que seja local para a consulta. É mais fácil lidar com exceções fora da expressão de consulta.

C#

```
// A data source that is very likely to throw an exception!
IEnumerable<int> GetData() => throw new InvalidOperationException();

// DO THIS with a datasource that might
// throw an exception.
IEnumerable<int>? dataSource = null;
try
{
    dataSource = GetData();
}
catch (InvalidOperationException)
{
    Console.WriteLine("Invalid operation");
}

if (dataSource is not null)
{
    // If we get here, it is safe to proceed.
    var query = from i in dataSource
                select i * i;

    foreach (var i in query)
    {
        Console.WriteLine(i.ToString());
    }
}
```

No bloco `catch (InvalidOperationException)` no exemplo anterior, manipule (ou não manipule) a exceção da maneira apropriada para seu aplicativo.

Em alguns casos, a melhor resposta para uma exceção que é lançada de dentro de uma consulta poderá ser a interrupção imediata da execução da consulta. O exemplo a seguir mostra como tratar exceções que podem ser geradas de dentro de um corpo de consulta. Suponha que `SomeMethodThatMightThrow` possa causar uma exceção que exija que a execução da consulta seja interrompida.

O bloco `try` inclui o loop `foreach` e não a própria consulta. O loop `foreach` é o ponto em que a consulta é executada. As exceções de runtime são gerada quando a consulta for executada. Portanto, elas devem ser manipuladas no loop `foreach`.

C#

```
// Not very useful as a general purpose method.
string SomeMethodThatMightThrow(string s) =>
    s[4] == 'C' ?
        throw new InvalidOperationException() :
        $"""C:\newFolder\{s}""";

// Data source.
string[] files = ["fileA.txt", "fileB.txt", "fileC.txt"];

// Demonstration query that throws.
var exceptionDemoQuery = from file in files
                         let n = SomeMethodThatMightThrow(file)
                         select n;

try
{
    foreach (var item in exceptionDemoQuery)
    {
        Console.WriteLine($"Processing {item}");
    }
}
catch (InvalidOperationException e)
{
    Console.WriteLine(e.Message);
}

/* Output:
    Processing C:\newFolder\fileA.txt
    Processing C:\newFolder\fileB.txt
    Operation is not valid due to the current state of the object.
 */
```

Lembre-se de capturar qualquer exceção que você espera gerar e/ou fazer qualquer limpeza necessária em um bloco `finally`.

Confira também

- Cláusula where
- Como realizar consultas com base no estado do runtime
- Nullable<T>
- Tipos de valor anuláveis

Relações de tipo em operações de consulta LINQ (C#)

19/06/2025

Para redigir consultas de maneira eficaz, você deve entender como os tipos das variáveis em uma operação completa de consulta se relacionam entre si. Se você entender essas relações, compreenderá com mais facilidade os exemplos de linq e exemplos de código na documentação. Além disso, você entenderá o que ocorre quando as variáveis são digitada implicitamente usando var.

As operações de consulta LINQ são fortemente tipadas na fonte de dados, na própria consulta e na execução da consulta. O tipo das variáveis na consulta deve ser compatível com o tipo dos elementos na fonte de dados e com o tipo da variável de iteração na foreach instrução. Essa digitação forte garante que os erros de tipo sejam capturados no momento da compilação, quando eles podem ser corrigidos antes que os usuários os encontrem.

Para demonstrar essas relações de tipo, a maioria dos exemplos a seguir usam digitação explícita para todas as variáveis. O último exemplo mostra como os mesmos princípios se aplicam mesmo quando você usa digitação implícita usando `var`.

Consultas que não transformam os dados de origem

A ilustração a seguir mostra uma operação de consulta LINQ to Objects que não executa transformações nos dados. A origem contém uma sequência de cadeias de caracteres e a saída da consulta também é uma sequência de cadeias de caracteres.

1. O argumento de tipo da fonte de dados determina o tipo da variável de intervalo.
2. O tipo do objeto selecionado determina o tipo da variável de consulta. Aqui `name` está uma cadeia de caracteres. Portanto, a variável de consulta é um `IEnumerable<string>`.

3. A variável de consulta é iterada na instrução `foreach`. Como a variável de consulta é uma sequência de cadeias de caracteres, a variável de iteração também é uma cadeia de caracteres.

Consultas que transformam os dados de origem

A ilustração a seguir mostra uma operação de consulta LINQ to SQL que executa uma transformação simples nos dados. A consulta usa uma sequência de `Customer` objetos como entrada e seleciona apenas a `Name` propriedade no resultado. Como `Name` é uma cadeia de caracteres, a consulta produz uma sequência de cadeias de caracteres como saída.

```
Table<Customer> Customers = db.GetTable<Customers>();

IQueryable<string> custNameQuery =
                    from cust in Customers
                       where cust.City == "London"
                       select cust.Name;

foreach (string str in custNameQuery)
{
     Console.WriteLine(str);
}
```

1. O argumento de tipo da fonte de dados determina o tipo da variável de intervalo.
2. A `select` instrução retorna a `Name` propriedade em vez do objeto completo `Customer` . Como `Name` é uma cadeia de caracteres, o argumento de `custNameQuery` tipo é `string`, não `Customer`.
3. Como `custNameQuery` é uma sequência de cadeias de caracteres, a `foreach` variável de iteração do loop também deve ser um `string`.

A ilustração a seguir mostra uma transformação um pouco mais complexa. A `select` instrução retorna um tipo anônimo que captura apenas dois membros do objeto original `Customer` .

```
Table<Customer> Customers = db.GetTable<Customers>();

var namePhoneQuery =
                 from cust in Customers
                   where cust.City == "London"
                   select new { name = cust.Name,
                        phone = cust.Phone };

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

1. O argumento de tipo da fonte de dados é sempre o tipo da variável de intervalo na consulta.
2. Como a `select` instrução produz um tipo anônimo, a variável de consulta deve ser digitada implicitamente usando `var`.
3. Como o tipo da variável de consulta está implícito, a variável de iteração no `foreach` loop também deve estar implícita.

Permitindo que o compilador infera informações de tipo

Embora você deva entender as relações de tipo em uma operação de consulta, você tem a opção de permitir que o compilador faça todo o trabalho para você. A palavra-chave var pode ser usada para qualquer variável local em uma operação de consulta. A ilustração a seguir é semelhante ao exemplo número 2 que foi discutido anteriormente. No entanto, o compilador fornece o tipo forte para cada variável na operação de consulta.

LINQ e tipos genéricos (C#)

As consultas LINQ são baseadas em tipos genéricos. Você não precisa de um conhecimento aprofundado de genéricos antes de começar a escrever consultas. No entanto, talvez você queira entender dois conceitos básicos:

1. Ao criar uma instância de uma classe de coleção genérica, como List<T>, você substitui o "T" pelo tipo de objetos que a lista conterá. Por exemplo, uma lista de cadeias de caracteres é expressa como `List<string>`, e uma lista de `Customer` objetos é expressa como `List<Customer>`. Uma lista genérica é fortemente tipada e oferece muitos benefícios em coleções que armazenam seus elementos como Object. Se você tentar adicionar um `Customer` a um `List<string>`, receberá um erro em tempo de compilação. É fácil usar coleções genéricas, porque você não precisa realizar a conversão de tipo em tempo de execução.

2. IEnumerable<T> é a interface que permite que classes de coleção genéricas sejam enumeradas usando a `foreach` instrução. Classes de coleção genérica dão suporte a IEnumerable<T> assim como classes de coleção não genéricas, como ArrayList dão suporte a IEnumerable.

Para obter mais informações sobre genéricos, consulte Genéricos.

Variáveis IEnumerable<T> em consultas LINQ

As variáveis de consulta LINQ são digitada como IEnumerable<T> ou um tipo derivado, como IQueryable<T>. Quando você vê uma variável de consulta que é digitada como `IEnumerable<Customer>`, isso significa apenas que a consulta, quando executada, produzirá uma sequência de zero ou mais `Customer` objetos.

C#

```
IEnumerable<Customer> customerQuery = from cust in customers
                                      where cust.City == "London"
                                      select cust;

foreach (Customer customer in customerQuery)
{
    Console.WriteLine($"{customer.LastName}, {customer.FirstName}");
}
```

Permitir que o compilador manipule declarações de tipo genérico

Se preferir, você poderá evitar a sintaxe genérica usando a palavra-chave var . A `var` palavra-chave instrui o compilador a inferir o tipo de uma variável de consulta examinando a fonte de dados especificada na `from` cláusula. O exemplo a seguir produz o mesmo código compilado do exemplo anterior:

C#

```
var customerQuery2 = from cust in customers
                     where cust.City == "London"
                     select cust;

foreach(var customer in customerQuery2)
{
    Console.WriteLine($"{customer.LastName}, {customer.FirstName}");
}
```

A `var` palavra-chave é útil quando o tipo da variável é óbvio ou quando não é tão importante especificar explicitamente tipos genéricos aninhados, como aqueles produzidos por consultas de grupo. Em geral, recomendamos que, se você usar `var`, perceba que ele pode tornar seu código mais difícil para outras pessoas lerem. Para obter mais informações, consulte Variáveis locais de tipo implícito.

Recursos do C# que dão suporte ao LINQ

18/06/2025

Expressões de consulta

As expressões de consulta usam uma sintaxe declarativa semelhante a SQL ou XQuery para consultar coleções System.Collections.Generic.IEnumerable<T> . No momento da compilação, a sintaxe de consulta é convertida em chamadas de método para a implementação dos métodos padrão de consulta por um provedor LINQ. Os aplicativos controlam os operadores de consulta padrão que estão no escopo, especificando o namespace apropriado com uma diretiva using. A expressão de consulta a seguir usa uma matriz de cadeias de caracteres, agrupa-as de acordo com o primeiro caractere na cadeia de caracteres e ordena os grupos.

C#

```
var query = from str in stringArray
            group str by str[0] into stringGroup
            orderby stringGroup.Key
            select stringGroup;
```

Variáveis de tipo implícito (var)

Você pode usar o modificador var para instruir o compilador a inferir e atribuir o tipo, conforme mostrado aqui:

C#

```
var number = 5;
var name = "Virginia";
var query = from str in stringArray
            where str[0] == 'm'
            select str;
```

Variáveis declaradas como `var` são fortemente tipadas, assim como variáveis cujo tipo você especifica explicitamente. O uso do `var` possibilita a criação de tipos anônimos, mas apenas para variáveis locais. Para obter mais informações, consulte Variáveis locais de tipo implícito.

Inicializadores de objeto e coleção

Inicializadores de objeto e coleção possibilitam inicializar objetos sem chamar explicitamente um construtor para o objeto. Os inicializadores normalmente são usados em expressões de

consulta quando projetam os dados de origem em um novo tipo de dados. Supondo que uma classe nomeada `Customer` com propriedades `Name` e `Phone` públicos, o inicializador de objeto pode ser usado como no seguinte código:

```
C#

var cust = new Customer { Name = "Mike", Phone = "555-1212" };
```

Continuando com sua `Customer` classe, suponha que haja uma fonte de dados chamada `IncomingOrders`, e que para cada pedido com um grande `OrderSize`, você gostaria de criar um novo `Customer` com base nessa ordem. Uma consulta LINQ pode ser executada nesta fonte de dados e usar a inicialização de objeto para preencher uma coleção:

```
C#

var newLargeOrderCustomers = from o in IncomingOrders
                             where o.OrderSize > 5
                             select new Customer { Name = o.Name, Phone = o.Phone
};
```

A fonte de dados pode ter mais propriedades definidas do que a `Customer` classe, como `OrderSize`, mas, com a inicialização de objetos, os dados retornados da consulta são moldados para o tipo de dados desejado; você escolhe os dados relevantes para sua classe. Como resultado, agora você tem um System.Collections.Generic.IEnumerable<T> preenchido com os novos `Customer`s que você queria. O exemplo anterior também pode ser escrito na sintaxe do método LINQ:

```
C#

var newLargeOrderCustomers = IncomingOrders.Where(x => x.OrderSize > 5).Select(y
=> new Customer { Name = y.Name, Phone = y.Phone });
```

A partir do C# 12, você pode usar uma expressão de coleção para inicializar uma coleção.

Para obter mais informações, consulte:

- Inicializadores de objeto e coleção
- Sintaxe de expressão de consulta para operadores de consulta padrão

Tipos anônimos

O compilador constrói um tipo anônimo. O nome do tipo só está disponível para o compilador. Os tipos anônimos fornecem uma maneira conveniente de agrupar um conjunto

de propriedades temporariamente em um resultado de consulta sem precisar definir um tipo nomeado separado. Tipos anônimos são inicializados com uma nova expressão e um inicializador de objeto, conforme mostrado aqui:

```
C#

select new {name = cust.Name, phone = cust.Phone};
```

A partir do C# 7, você pode usar tuplas para criar tipos sem nome.

Métodos de Extensão

Um método de extensão é um método estático que pode ser associado a um tipo, para que ele possa ser chamado como se fosse um método de instância no tipo. Esse recurso permite que você, de fato, "adicione" novos métodos aos tipos existentes sem realmente modificá-los. Os operadores de consulta padrão são um conjunto de métodos de extensão que fornecem funcionalidade de consulta LINQ para qualquer tipo que implemente IEnumerable<T>.

Expressões Lambda

Uma expressão lambda é uma função embutida que usa o operador `=>` para separar os parâmetros de entrada do corpo da função e pode ser convertida, durante a compilação, em um delegado ou em uma árvore de expressão. Na programação LINQ, você encontra expressões lambda ao fazer chamadas de método direto para os operadores de consulta padrão.

Expressões como dados

Os objetos de consulta são componíveis, o que significa que podem ser retornados a partir de um método. Os objetos que representam consultas não armazenam a coleção resultante, mas sim as etapas para produzir os resultados quando necessário. A vantagem de retornar objetos de consulta a partir de métodos é que eles podem ser ainda mais compostos ou modificados. Portanto, qualquer valor ou `out` parâmetro retornado de um método que retorna uma consulta também deve ter esse tipo. Se um método materializa uma consulta em um tipo concreto List<T> ou Array, ele retornará os resultados da consulta em vez da consulta em si. Uma variável de consulta retornada de um método ainda pode ser composta ou modificada.

No exemplo a seguir, o primeiro método `QueryMethod1` retorna uma consulta como um valor retornado e o segundo método `QueryMethod2` retorna uma consulta como um `out` parâmetro

(`returnQ` no exemplo). Em ambos os casos, é uma consulta retornada, não os resultados da consulta.

```C#
IEnumerable<string> QueryMethod1(int[] ints) =>
    from i in ints
    where i > 4
    select i.ToString();

void QueryMethod2(int[] ints, out IEnumerable<string> returnQ) =>
    returnQ = from i in ints
              where i < 4
              select i.ToString();

int[] nums = [ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 ];

var myQuery1 = QueryMethod1(nums);
```

A consulta `myQuery1` é executada no loop foreach a seguir.

```C#
foreach (var s in myQuery1)
{
    Console.WriteLine(s);
}
```

Coloque o ponteiro do mouse sobre `myQuery1` para ver seu tipo.

Você também pode executar a consulta retornada `QueryMethod1` diretamente, sem usar `myQuery1`.

```C#
foreach (var s in QueryMethod1(nums))
{
    Console.WriteLine(s);
}
```

Coloque o ponteiro do mouse sobre a chamada a `QueryMethod1` para ver o tipo de retorno.

`QueryMethod2` retorna uma consulta como o valor de seu `out` parâmetro:

```C#
QueryMethod2(nums, out IEnumerable<string> myQuery2);
```

```
// Execute the returned query.
foreach (var s in myQuery2)
{
    Console.WriteLine(s);
}
```

Você pode modificar uma consulta usando a composição de consulta. Nesse caso, o objeto de consulta anterior é usado para criar um novo objeto de consulta. Esse novo objeto retorna resultados diferentes do objeto de consulta original.

C#

```
myQuery1 = from item in myQuery1
           orderby item descending
           select item;

// Execute the modified query.
Console.WriteLine("\nResults of executing modified myQuery1:");
foreach (var s in myQuery1)
{
    Console.WriteLine(s);
}
```

Tutorial: Escrever consultas em C# usando consulta integrada à linguagem (LINQ)

08/04/2025

Neste tutorial, você cria uma fonte de dados e escreve várias consultas LINQ. Você pode experimentar as expressões de consulta e ver as diferenças nos resultados. Este passo a passo demonstra os recursos da linguagem C# que são usados para escrever expressões de consulta LINQ. Você pode acompanhar e criar o aplicativo e experimentar as consultas por conta própria. Este artigo pressupõe que você tenha instalado o SDK .NET mais recente. Caso contrário, vá para a página Downloads do .NET e instale a versão mais recente em sua máquina.

Primeiro, crie o aplicativo. No console, digite o seguinte comando:

CLI do .NET

```
dotnet new console -o WalkthroughWritingLinqQueries
```

Ou, se preferir o Visual Studio, crie um novo aplicativo de console chamado *WalkthroughWritingLinqQueries*.

Criar uma fonte de dados na memória

A primeira etapa é criar uma fonte de dados para suas consultas. A fonte de dados para as consultas é uma lista simples de `Student` registros. Cada `Student` registro tem um nome, nome de família e uma matriz de inteiros que representa suas pontuações de teste na classe. Adicione um novo arquivo chamado *students.cs* e copie o seguinte código para esse arquivo:

C#

```
namespace WalkthroughWritingLinqQueries;

public record Student(string First, string Last, int ID, int[] Scores);
```

Observe as seguintes características:

- O `Student` registro consiste em propriedades implementadas automaticamente.
- Cada aluno na lista é inicializado com o construtor primário.
- A sequência de pontuações para cada aluno é inicializada com um construtor primário.

Em seguida, crie uma sequência de `Student` registros que sirva como fonte dessa consulta. Abra *Program.cs* e remova o seguinte código clichê:

```
C#

// See https://aka.ms/new-console-template for more information
Console.WriteLine("Hello, World!");
```

Substitua-o pelo seguinte código que cria uma sequência de `Student` registros:

```
C#

using WalkthroughWritingLinqQueries;

// Create a data source by using a collection initializer.
IEnumerable<Student> students =
[
    new Student(First: "Svetlana", Last: "Omelchenko", ID: 111, Scores: [97, 92,
81, 60]),
    new Student(First: "Claire",   Last: "O'Donnell",  ID: 112, Scores: [75, 84,
91, 39]),
    new Student(First: "Sven",     Last: "Mortensen",  ID: 113, Scores: [88, 94,
65, 91]),
    new Student(First: "Cesar",    Last: "Garcia",     ID: 114, Scores: [97, 89,
85, 82]),
    new Student(First: "Debra",    Last: "Garcia",     ID: 115, Scores: [35, 72,
91, 70]),
    new Student(First: "Fadi",     Last: "Fakhouri",   ID: 116, Scores: [99, 86,
90, 94]),
    new Student(First: "Hanying",  Last: "Feng",       ID: 117, Scores: [93, 92,
80, 87]),
    new Student(First: "Hugo",     Last: "Garcia",     ID: 118, Scores: [92, 90,
83, 78]),

    new Student("Lance",   "Tucker",      119, [68, 79, 88, 92]),
    new Student("Terry",   "Adams",       120, [99, 82, 81, 79]),
    new Student("Eugene",  "Zabokritski", 121, [96, 85, 91, 60]),
    new Student("Michael", "Tucker",      122, [94, 92, 91, 91])
];
```

- A sequência de estudantes é inicializada com uma expressão de coleção.
- O `Student` tipo de registro contém a lista estática de todos os alunos.
- Algumas das chamadas do construtor usam argumentos nomeados para esclarecer qual argumento corresponde a qual parâmetro do construtor.

Tente adicionar mais alguns alunos com pontuações de teste diferentes à lista de alunos para se familiarizar mais com o código até agora.

Criar a consulta

Em seguida, crie sua primeira consulta. A sua consulta, quando a executa, produz uma lista de todos os alunos cuja pontuação no primeiro teste foi superior a 90. Como todo o `Student` objeto está selecionado, o tipo da consulta é `IEnumerable<Student>`. Embora o código também possa usar a digitação implícita usando a palavra-chave var , a digitação explícita é usada para ilustrar claramente os resultados. (Para informações adicionais sobre `var`, consulte Variáveis locais digitadas implicitamente.) Adicione o seguinte código ao *Program.cs*, após o código que cria a sequência de alunos.

C#

```
// Create the query.
// The first line could also be written as "var studentQuery ="
IEnumerable<Student> studentQuery =
    from student in students
    where student.Scores[0] > 90
    select student;
```

A variável de intervalo da consulta, `student`, serve como referência para cada `Student` um na fonte, fornecendo acesso de membro para cada objeto.

Executar a consulta

Agora escreva o `foreach` loop que faz com que a consulta seja executada. Cada elemento na sequência retornada é acessado através da variável de iteração no `foreach` loop. O tipo desta variável é `Student`, e o tipo da variável de consulta é compatível, `IEnumerable<Student>`. Depois de adicionar o código a seguir, compile e execute o aplicativo para ver os resultados na janela **Console** .

C#

```
// Execute the query.
// var could be used here also.
foreach (Student student in studentQuery)
{
    Console.WriteLine($"{student.Last}, {student.First}");
}

// Output:
// Omelchenko, Svetlana
// Garcia, Cesar
// Fakhouri, Fadi
// Feng, Hanying
// Garcia, Hugo
```

```
// Adams, Terry
// Zabokritski, Eugene
// Tucker, Michael
```

Para refinar ainda mais a consulta, você pode combinar várias condições booleanas `where` na cláusula. O código a seguir adiciona uma condição para que a consulta retorne os alunos cuja primeira pontuação foi superior a 90 e cuja última pontuação foi inferior a 80. A `where` cláusula deve assemelhar-se ao código seguinte.

```C#
where student.Scores[0] > 90 && student.Scores[3] < 80
```

Experimente a cláusula anterior `where` ou experimente outras condições de filtro. Para obter mais informações, consulte cláusula WHERE.

Ordenar os resultados da consulta

É mais fácil digitalizar os resultados se eles estiverem em algum tipo de ordem. Você pode ordenar a sequência retornada por qualquer campo acessível nos elementos de origem. Por exemplo, a cláusula a seguir `orderby` ordena os resultados em ordem alfabética de A a Z de acordo com o nome de família de cada aluno. Adicione a seguinte `orderby` cláusula à sua consulta, logo após a `where` instrução e antes da `select` instrução:

```C#
orderby student.Last ascending
```

Agora altere a `orderby` cláusula para que ela ordene os resultados em ordem inversa de acordo com a pontuação no primeiro teste, da maior pontuação para a menor pontuação.

```C#
orderby student.Scores[0] descending
```

Altere a cadeia de caracteres de `WriteLine` formato para que você possa ver as pontuações:

```C#
Console.WriteLine($"{student.Last}, {student.First} {student.Scores[0]}");
```

Para obter mais informações, consulte a cláusula de ordenação por .

Agrupar os resultados

O agrupamento é um recurso poderoso em expressões de consulta. Uma consulta com uma cláusula de grupo produz uma sequência de grupos, e cada grupo em si contém uma `Key` e uma sequência que consiste em todos os membros desse grupo. A nova consulta a seguir agrupa os alunos usando a primeira letra do nome da família como chave.

C#

```
IEnumerable<IGrouping<char, Student>> studentQuery =
    from student in students
    group student by student.Last[0];
```

O tipo da consulta foi alterado. Ele agora produz uma sequência de grupos que têm um `char` tipo como chave e uma sequência de `Student` objetos. O código no loop de execução `foreach` também deve mudar.

C#

```
foreach (IGrouping<char, Student> studentGroup in studentQuery)
{
    Console.WriteLine(studentGroup.Key);
    foreach (Student student in studentGroup)
    {
        Console.WriteLine($"   {student.Last}, {student.First}");
    }
}
// Output:
// O
//   Omelchenko, Svetlana
//   O'Donnell, Claire
// M
//   Mortensen, Sven
// G
//   Garcia, Cesar
//   Garcia, Debra
//   Garcia, Hugo
// F
//   Fakhouri, Fadi
//   Feng, Hanying
// T
//   Tucker, Lance
//   Tucker, Michael
// A
//   Adams, Terry
// Z
//   Zabokritski, Eugene
```

Execute o aplicativo e visualize os resultados na janela **Console** . Para obter mais informações, consulte o grupo de cláusulas .

A codificação `IEnumerables` explícita de `IGroupings` pode rapidamente tornar-se entediante. Escreva a mesma consulta e execute o loop de forma muito mais conveniente usando `var`. A `var` palavra-chave não altera os tipos de seus objetos, apenas instrui o compilador a inferir os tipos. Altere o tipo de `studentQuery` e a variável de iteração `group` para `var` e execute novamente a consulta. No loop interno `foreach` , a variável de iteração ainda é digitada como `Student`, e a consulta funciona como antes. Altere a `student` variável de iteração para `var` e execute a consulta novamente. Você vê que obtém exatamente os mesmos resultados.

C#

```
IEnumerable<IGrouping<char, Student>> studentQuery =
    from student in students
    group student by student.Last[0];

foreach (IGrouping<char, Student> studentGroup in studentQuery)
{
    Console.WriteLine(studentGroup.Key);
    foreach (Student student in studentGroup)
    {
        Console.WriteLine($"   {student.Last}, {student.First}");
    }
}
```

Para obter mais informações sobre `var`, consulte Variáveis locais tipadas implicitamente.

Ordenar os grupos pelo seu valor-chave

Os grupos na consulta anterior não estão em ordem alfabética. Você pode fornecer uma `orderby` cláusula após a `group` cláusula. Mas para usar uma `orderby` cláusula, primeiro você precisa de um identificador que sirva de referência para os grupos criados pela `group` cláusula. Você fornece o identificador usando a palavra-chave `into` , da seguinte maneira:

C#

```
var studentQuery4 =
    from student in students
    group student by student.Last[0] into studentGroup
    orderby studentGroup.Key
    select studentGroup;

foreach (var groupOfStudents in studentQuery4)
{
    Console.WriteLine(groupOfStudents.Key);
```

```
    foreach (var student in groupOfStudents)
    {
        Console.WriteLine($"   {student.Last}, {student.First}");
    }
}

// Output:
//A
//   Adams, Terry
//F
//   Fakhouri, Fadi
//   Feng, Hanying
//G
//   Garcia, Cesar
//   Garcia, Debra
//   Garcia, Hugo
//M
//   Mortensen, Sven
//O
//   Omelchenko, Svetlana
//   O'Donnell, Claire
//T
//   Tucker, Lance
//   Tucker, Michael
//Z
//   Zabokritski, Eugene
```

Execute essa consulta e os grupos agora são classificados em ordem alfabética.

Você pode usar a `let` palavra-chave para introduzir um identificador para qualquer resultado de expressão na expressão de consulta. Esse identificador pode ser uma conveniência, como no exemplo a seguir. Ele também pode melhorar o desempenho armazenando os resultados de uma expressão para que ela não precise ser calculada várias vezes.

C#

```
// This query returns those students whose
// first test score was higher than their
// average score.
var studentQuery5 =
    from student in students
    let totalScore = student.Scores[0] + student.Scores[1] +
        student.Scores[2] + student.Scores[3]
    where totalScore / 4 < student.Scores[0]
    select $"{student.Last}, {student.First}";

foreach (string s in studentQuery5)
{
    Console.WriteLine(s);
}

// Output:
```

```
// Omelchenko, Svetlana
// O'Donnell, Claire
// Mortensen, Sven
// Garcia, Cesar
// Fakhouri, Fadi
// Feng, Hanying
// Garcia, Hugo
// Adams, Terry
// Zabokritski, Eugene
// Tucker, Michael
```

Para obter mais informações, consulte o artigo sobre a let cláusula.

Usar sintaxe de método em uma expressão de consulta

Conforme descrito em Sintaxe de consulta e sintaxe de método no LINQ, algumas operações de consulta só podem ser expressas usando sintaxe de método. O código a seguir calcula a pontuação total de cada `Student` na sequência de origem e, em seguida, chama o `Average()` método nos resultados dessa consulta para calcular a média de pontuação da classe.

C#

```
var studentQuery =
    from student in students
    let totalScore = student.Scores[0] + student.Scores[1] +
        student.Scores[2] + student.Scores[3]
    select totalScore;

double averageScore = studentQuery.Average();
Console.WriteLine($"Class average score = {averageScore}");

// Output:
// Class average score = 334.166666666667
```

Para transformar ou projetar na cláusula de seleção

É comum que uma consulta produza uma sequência cujos elementos diferem dos elementos nas sequências de origem. Exclua ou comente sua consulta anterior e loop de execução e substitua-o pelo código a seguir. A consulta retorna uma sequência de cadeias de caracteres (não `Students`), e este facto reflete-se no loop `foreach`.

C#

```
IEnumerable<string> studentQuery =
    from student in students
    where student.Last == "Garcia"
    select student.First;

Console.WriteLine("The Garcias in the class are:");
foreach (string s in studentQuery)
{
    Console.WriteLine(s);
}

// Output:
// The Garcias in the class are:
// Cesar
// Debra
// Hugo
```

O código anterior neste passo a passo indicou que a pontuação média da classe é de aproximadamente 334. Para produzir uma sequência de `Students` cuja pontuação total é maior do que a média da classe, juntamente com os seus `Student ID`, você pode usar um tipo anônimo na instrução `select`:

C#

```
var aboveAverageQuery =
    from student in students
    let x = student.Scores[0] + student.Scores[1] +
        student.Scores[2] + student.Scores[3]
    where x > averageScore
    select new { id = student.ID, score = x };

foreach (var item in aboveAverageQuery)
{
    Console.WriteLine("Student ID: {0}, Score: {1}", item.id, item.score);
}

// Output:
// Student ID: 113, Score: 338
// Student ID: 114, Score: 353
// Student ID: 116, Score: 369
// Student ID: 117, Score: 352
// Student ID: 118, Score: 343
// Student ID: 120, Score: 341
// Student ID: 122, Score: 368
```

Visão geral dos operadores de consulta padrão

19/06/2025

Os *operadores de consulta padrão* são as palavras-chave e os métodos que formam o padrão LINQ. A linguagem C# define palavras-chave de consulta LINQ que você usa para a expressão de consulta mais comum. O compilador converte expressões usando essas palavras-chave para as chamadas de método equivalentes. As duas formas são sinônimos. Outros métodos que fazem parte do System.Linq namespace não têm palavras-chave de consulta equivalentes. Nesses casos, você deve usar a sintaxe do método. Esta seção aborda todas as palavras-chave do operador de consulta. O runtime e outros pacotes NuGet adicionam mais métodos projetados para trabalhar com consultas LINQ a cada versão. Os métodos mais comuns, incluindo aqueles que têm equivalentes de palavra-chave de consulta, são abordados nesta seção. Para obter a lista completa de métodos de consulta compatíveis com o Runtime do .NET, consulte a documentação da System.Linq.Enumerable API. Além dos métodos abordados aqui, essa classe contém métodos para concatenar fontes de dados, computando um único valor de uma fonte de dados, como uma soma, uma média ou outro valor.

> ⓘ **Importante**
>
> Esses exemplos usam uma fonte de dados **System.Collections.Generic.IEnumerable<T>**. Fontes de dados baseadas em **System.Linq.IQueryProvider** usam as fontes de dados **System.Linq.IQueryable<T>** e as **árvores de expressão**. As árvores de expressão possuem **limitações** na sintaxe C# permitida. Além disso, todas as fontes de dados `IQueryProvider`, como **EF Core**, podem impor mais restrições. Verifique a documentação da fonte de dados.

A maioria desses métodos opera em sequências, em que uma sequência é um objeto cujo tipo implementa a IEnumerable<T> interface ou a IQueryable<T> interface. Os operadores de consulta padrão fornecem recursos de consulta, incluindo filtragem, projeção, agregação, classificação e muito mais. Os métodos que compõem cada conjunto são membros estáticos das classes Enumerable e Queryable, respectivamente. Eles são *definidos como métodos de extensão* do tipo em que operam.

A distinção entre IEnumerable<T> e IQueryable<T> sequências determina como a consulta é executada em runtime.

Para `IEnumerable<T>`, o objeto enumerável retornado captura os argumentos que foram passados para o método. Quando esse objeto é enumerado, a lógica do operador de consulta é empregada e os resultados da consulta são retornados.

Para `IQueryable<T>`, a consulta é traduzida para uma árvore de expressão. A árvore de expressão pode ser traduzida para uma consulta nativa quando a fonte de dados pode otimizar a consulta. Bibliotecas como o Entity Framework convertem consultas LINQ em consultas SQL nativas que são executadas no banco de dados.

O exemplo de código a seguir demonstra como os operadores de consulta padrão podem ser usados para obter informações sobre uma sequência.

C#

```
string sentence = "the quick brown fox jumps over the lazy dog";
// Split the string into individual words to create a collection.
string[] words = sentence.Split(' ');

// Using query expression syntax.
var query = from word in words
            group word.ToUpper() by word.Length into gr
            orderby gr.Key
            select new { Length = gr.Key, Words = gr };

// Using method-based query syntax.
var query2 = words.
    GroupBy(w => w.Length, w => w.ToUpper()).
    Select(g => new { Length = g.Key, Words = g }).
    OrderBy(o => o.Length);

foreach (var obj in query)
{
    Console.WriteLine($"Words of length {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
```

Sempre que possível, as consultas nesta seção usam uma sequência de palavras ou números como a origem de entrada. Para consultas em que relações mais complicadas entre objetos são usadas, as seguintes fontes que modelam uma escola são usadas:

```C#
public enum GradeLevel
{
    FirstYear = 1,
    SecondYear,
    ThirdYear,
    FourthYear
};

public class Student
{
    public required string FirstName { get; init; }
    public required string LastName { get; init; }
    public required int ID { get; init; }

    public required GradeLevel Year { get; init; }
    public required List<int> Scores { get; init; }

    public required int DepartmentID { get; init; }
}

public class Teacher
{
    public required string First { get; init; }
    public required string Last { get; init; }
    public required int ID { get; init; }
    public required string City { get; init; }
}

public class Department
{
    public required string Name { get; init; }
    public int ID { get; init; }

    public required int TeacherID { get; init; }
}
```

Cada `Student` tem um nível de escolaridade, um departamento primário e uma série de pontuações. Um `Teacher` também tem uma propriedade `City` que identifica o campus onde o docente ministra aulas. A `Department` tem um nome e uma referência a um `Teacher` que atua como chefe do departamento.

Você pode encontrar o conjunto de dados no repositório de origem.

Tipos de operadores de consulta

Os operadores de consulta padrão diferem no tempo de execução, dependendo se retornam um valor singleton ou uma sequência de valores. Os métodos que retornam um valor singleton

(como Average e Sum) são executados imediatamente. Métodos que retornam uma sequência adiam a execução da consulta e retornam um objeto enumerável. Você pode usar a sequência de saída de uma consulta como a sequência de entrada para outra consulta. As chamadas para métodos de consulta podem ser encadeadas em uma consulta, o que permite que as consultas se tornem arbitrariamente complexas.

Operadores de consulta

Em uma consulta LINQ, a primeira etapa é especificar a fonte de dados. Em uma consulta LINQ, a `from` cláusula vem primeiro para introduzir a fonte de dados (`students`) e a *variável de intervalo* (`student`).

C#

```
//queryAllStudents is an IEnumerable<Student>
var queryAllStudents = from student in students
                       select student;
```

A variável de intervalo é como a variável de iteração em um `foreach` loop, exceto que nenhuma iteração real ocorre em uma expressão de consulta. Quando a consulta é executada, a variável de intervalo serve como uma referência a cada elemento sucessivo em `students`. Como o compilador pode inferir o tipo de `student`, você não precisa especificá-lo explicitamente. Você pode introduzir mais variáveis de intervalo em uma `let` cláusula. Para obter mais informações, consulte Cláusula let.

> **Observação**
>
> Para fontes de dados não genéricas, como ArrayList, a variável de intervalo deve ser explicitamente tipada. Para obter mais informações, consulte Como consultar um ArrayList com LINQ (C#) e a cláusula from.

Depois de obter uma fonte de dados, você poderá executar qualquer número de operações nessa fonte de dados:

- Filtrar dados usando a `where` palavra-chave.
- Solicite dados usando as `orderby` palavras-chave e, opcionalmente `descending` .
- Agrupar dados usando as `group` palavras-chave e, opcionalmente `into` .
- Unir dados usando a `join` palavra-chave.
- Dados do projeto usando a `select` palavra-chave.

Tabela de sintaxe de expressão de consulta

A tabela a seguir lista os operadores de consulta padrão que têm cláusulas de expressão de consulta equivalentes.

Expandir a tabela

Método	Sintaxe da expressão de consulta C#
Cast	Use uma variável de intervalo explicitamente tipada: `from int i in numbers` (Para obter mais informações, consulte Cláusula from.)

GroupBy	`group … by`
	
	- ou -
	
	`group … by … into …`
	
	(Para obter mais informações, consulte cláusula de grupo.)

GroupJoin<TOuter,TInner,TKey,TResult>(IEnumerable<TOuter>, IEnumerable<TInner>, Func<TOuter,TKey>, Func<TInner,TKey>, Func<TOuter,IEnumerable<TInner>, TResult>)	`join … in … on … equals … into …` (Para obter mais informações, consulte cláusula de junção.)
Join<TOuter,TInner,TKey,TResult>(IEnumerable<TOuter>, IEnumerable<TInner>, Func<TOuter,TKey>, Func<TInner,TKey>, Func<TOuter,TInner,TResult>)	`join … in … on … equals …` (Para obter mais informações, consulte cláusula de junção.)
OrderBy<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>)	`orderby` (Para obter mais

Método	Sintaxe da expressão de consulta C#
	informações, consulte a cláusula orderby.)
OrderByDescending<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>)	`orderby … descending` (Para obter mais informações, consulte a cláusula orderby.)
Select	`select` (Para obter mais informações, consulte a cláusula de seleção.)
SelectMany	Várias cláusulas `from`. (Para obter mais informações, consulte Cláusula from.)
ThenBy<TSource,TKey>(IOrderedEnumerable<TSource>, Func<TSource,TKey>)	`orderby …, …` (Para obter mais informações, consulte a cláusula orderby.)
ThenByDescending<TSource,TKey>(IOrderedEnumerable<TSource>, Func<TSource,TKey>)	`orderby …, … descending` (Para obter mais informações, consulte a cláusula orderby.)
Where	`where` (Para obter mais informações, consulte cláusula WHERE.)

Transformações de dados com LINQ

Language-Integrated Query (LINQ) não se trata apenas de recuperar dados. Também é uma ferramenta poderosa para transformar dados. Usando uma consulta LINQ, você pode usar uma sequência de origem como entrada e modificá-la de várias maneiras para criar uma nova sequência de saída. Você pode modificar a sequência em si sem modificar os próprios

elementos classificando e agrupando. Mas talvez o recurso mais poderoso das consultas LINQ seja a capacidade de criar novos tipos. A cláusula select cria um elemento de saída de um elemento de entrada. Use-o para transformar um elemento de entrada em um elemento de saída:

- Mescle várias sequências de entrada em uma única sequência de saída que tenha um novo tipo.
- Crie sequências de saída cujos elementos consistem em apenas uma ou várias propriedades de cada elemento na sequência de origem.
- Crie sequências de saída cujos elementos consistem nos resultados das operações executadas nos dados de origem.
- Crie sequências de saída em um formato diferente. Por exemplo, você pode transformar dados de linhas SQL ou arquivos de texto em XML.

Essas transformações podem ser combinadas de várias maneiras na mesma consulta. Além disso, a sequência de saída de uma consulta pode ser usada como a sequência de entrada para uma nova consulta. O exemplo a seguir transforma objetos em uma estrutura de dados na memória em elementos XML.

C#

```
// Create the query.
var studentsToXML = new XElement("Root",
    from student in students
    let scores = string.Join(",", student.Scores)
    select new XElement("student",
                new XElement("First", student.FirstName),
                new XElement("Last", student.LastName),
                new XElement("Scores", scores)
            ) // end "student"
        ); // end "Root"

// Execute the query.
Console.WriteLine(studentsToXML);
```

O código produz a seguinte saída XML:

XML

```
<Root>
  <student>
    <First>Svetlana</First>
    <Last>Omelchenko</Last>
    <Scores>97,90,73,54</Scores>
  </student>
  <student>
    <First>Claire</First>
```

```
    <Last>O'Donnell</Last>
    <Scores>56,78,95,95</Scores>
  </student>
  ...
  <student>
    <First>Max</First>
    <Last>Lindgren</Last>
    <Scores>86,88,96,63</Scores>
  </student>
  <student>
    <First>Arina</First>
    <Last>Ivanova</Last>
    <Scores>93,63,70,80</Scores>
  </student>
</Root>
```

Para obter mais informações, consulte Criando árvores XML em C# (LINQ to XML).

Você pode usar os resultados de uma consulta como fonte de dados para uma consulta subsequente. Este exemplo mostra como ordenar os resultados de uma operação de junção. Esta consulta cria uma junção de grupos e classifica os grupos com base no elemento de categoria, que ainda está no escopo. Dentro do inicializador de tipo anônimo, uma subconsulta ordena todos os elementos correspondentes da sequência de produtos.

C#

```
var orderedQuery = from department in departments
                   join student in students on department.ID equals
student.DepartmentID into studentGroup
                   orderby department.Name
                   select new
                   {
                       DepartmentName = department.Name,
                       Students = from student in studentGroup
                                  orderby student.LastName
                                    select student
                   };

foreach (var departmentList in orderedQuery)
{
    Console.WriteLine(departmentList.DepartmentName);
    foreach (var student in departmentList.Students)
    {
        Console.WriteLine($"  {student.LastName,-10} {student.FirstName,-10}");
    }
}
/* Output:
Chemistry
  Balzan     Josephine
  Fakhouri   Fadi
  Popov      Innocenty
  Seleznyova Sofiya
```

```
  Vella      Carmen
Economics
  Adams      Terry
  Adaobi     Izuchukwu
  Berggren   Jeanette
  Garcia     Cesar
  Ifeoma     Nwanneka
  Jamuike    Ifeanacho
  Larsson    Naima
  Svensson   Noel
  Ugomma     Ifunanya
Engineering
  Axelsson   Erik
  Berg       Veronika
  Engström   Nancy
  Hicks      Cassie
  Keever     Bruce
  Micallef   Nicholas
  Mortensen  Sven
  Nilsson    Erna
  Tucker     Michael
  Yermolayeva Anna
English
  Andersson  Sarah
  Feng       Hanying
  Ivanova    Arina
  Jakobsson  Jesper
  Jensen     Christiane
  Johansson  Mark
  Kolpakova  Nadezhda
  Omelchenko Svetlana
  Urquhart   Donald
Mathematics
  Frost      Gaby
  Garcia     Hugo
  Hedlund    Anna
  Kovaleva   Katerina
  Lindgren   Max
  Maslova    Evgeniya
  Olsson     Ruth
  Sammut     Maria
  Sazonova   Anastasiya
Physics
  Åkesson    Sami
  Edwards    Amy E.
  Falzon     John
  Garcia     Debra
  Hansson    Sanna
  Mattsson   Martina
  Richardson Don
  Zabokritski Eugene
*/
```

A consulta equivalente usando a sintaxe do método é mostrada no código a seguir:

C#

```
var orderedQuery = departments
    .GroupJoin(students, department => department.ID, student =>
student.DepartmentID,
    (department, studentGroup) => new
    {
        DepartmentName = department.Name,
        Students = studentGroup.OrderBy(student => student.LastName)
    })
    .OrderBy(department => department.DepartmentName);


foreach (var departmentList in orderedQuery)
{
    Console.WriteLine(departmentList.DepartmentName);
    foreach (var student in departmentList.Students)
    {
        Console.WriteLine($"  {student.LastName,-10} {student.FirstName,-10}");
    }
}
```

Embora você possa usar uma `orderby` cláusula com uma ou mais das sequências de origem antes da junção, geralmente não a recomendamos. Alguns provedores LINQ podem não preservar essa ordenação após a junção. Para obter mais informações, consulte Cláusula join.

Consulte também

- Enumerable
- Queryable
- cláusula select
- Métodos de Extensão
- LINQ (Palavras-chave de consulta)
- Tipos anônimos

Filtrando dados em C# com LINQ

19/06/2025

A filtragem refere-se à operação de restrição do conjunto de resultados para conter apenas os elementos que atendem a uma condição especificada. Ele também é chamado de *seleção de elementos* que correspondem à condição especificada.

> **ⓘ Importante**
>
> Esses exemplos usam uma fonte de dados System.Collections.Generic.IEnumerable<T>. Fontes de dados baseadas em System.Linq.IQueryProvider usam as fontes de dados System.Linq.IQueryable<T> e as árvores de expressão. As árvores de expressão possuem limitações na sintaxe C# permitida. Além disso, todas as fontes de dados `IQueryProvider`, como EF Core, podem impor mais restrições. Verifique a documentação da fonte de dados.

A ilustração a seguir mostra os resultados da filtragem de uma sequência de caracteres. O predicado para a operação de filtragem especifica que o caractere deve ser 'A'.

Os métodos de operador de consulta padrão que executam a seleção são listados na tabela a seguir:

Expandir a tabela

Nome do método	Descrição	Sintaxe de expressão de consulta C#	Mais informações
OfType	Seleciona valores, dependendo da capacidade de serem convertidos em um tipo especificado.	Não aplicável.	Enumerable.OfType Queryable.OfType
Onde	Seleciona valores baseados em uma função de predicado.	`where`	Enumerable.Where Queryable.Where

O exemplo a seguir usa a `where` cláusula para filtrar de uma matriz as cadeias de caracteres que têm um comprimento específico.

C#

```
string[] words = ["the", "quick", "brown", "fox", "jumps"];

IEnumerable<string> query = from word in words
                            where word.Length == 3
                            select word;

foreach (string str in query)
{
    Console.WriteLine(str);
}

/* This code produces the following output:

    the
    fox
*/
```

A consulta equivalente usando a sintaxe do método é mostrada no código a seguir:

C#

```
string[] words = ["the", "quick", "brown", "fox", "jumps"];

IEnumerable<string> query =
    words.Where(word => word.Length == 3);

foreach (string str in query)
{
    Console.WriteLine(str);
}

/* This code produces the following output:

    the
    fox
*/
```

Consulte também

- System.Linq
- Cláusula where
- Como consultar os metadados de um assembly usando Reflection (LINQ) (C#)
- Como consultar arquivos com um atributo ou nome especificado (C#)

- Como classificar ou filtrar dados de texto por qualquer palavra ou campo (LINQ) (C#)

Operações de projeção (C#)

17/06/2025

A projeção refere-se à operação de transformação de um objeto em um novo formulário que geralmente consiste apenas nessas propriedades usadas posteriormente. Usando a projeção, você pode construir um novo tipo criado a partir de cada objeto. Você pode projetar uma propriedade e executar uma função matemática nela. Você também pode projetar o objeto original sem alterá-lo.

> **Importante**
>
> Esses exemplos usam uma fonte de dados System.Collections.Generic.IEnumerable<T>. Fontes de dados baseadas em System.Linq.IQueryProvider usam as fontes de dados System.Linq.IQueryable<T> e as árvores de expressão. As árvores de expressão possuem limitações na sintaxe C# permitida. Além disso, todas as fontes de dados `IQueryProvider`, como EF Core, podem impor mais restrições. Verifique a documentação da fonte de dados.

Os métodos de operador de consulta padrão que executam a projeção são listados na seção a seguir.

Métodos

Expandir a tabela

Nomes de método	Descrição	Sintaxe da expressão de consulta C#	Mais informações
Selecionar	Projeta valores baseados em uma função de transformação.	`select`	Enumerable.Select Queryable.Select
SelectMany	Projeta sequências de valores que são baseados em uma função de transformação e, em seguida, os nivela em uma sequência.	Use várias cláusulas `from`	Enumerable.SelectMany Queryable.SelectMany
Nada	Produz uma sequência de tuplas com elementos provenientes de 2 ou 3 sequências especificadas.	Não aplicável.	Enumerable.Zip Queryable.Zip

`Select`

O exemplo a seguir usa a `select` cláusula para projetar a primeira letra de cada cadeia de caracteres em uma lista de cadeias de caracteres.

C#

```
List<string> words = ["an", "apple", "a", "day"];

var query = from word in words
            select word.Substring(0, 1);

foreach (string s in query)
{
    Console.WriteLine(s);
}

/* This code produces the following output:

    a
    a
    a
    d
*/
```

A consulta equivalente usando a sintaxe do método é mostrada no código a seguir:

C#

```
List<string> words = ["an", "apple", "a", "day"];

var query = words.Select(word => word.Substring(0, 1));

foreach (string s in query)
{
    Console.WriteLine(s);
}

/* This code produces the following output:

    a
    a
    a
    d
*/
```

`SelectMany`

O exemplo a seguir usa várias cláusulas `from` para projetar cada palavra de cada cadeia de caracteres em uma lista de cadeias de caracteres.

C#

```
List<string> phrases = ["an apple a day", "the quick brown fox"];

var query = from phrase in phrases
            from word in phrase.Split(' ')
            select word;

foreach (string s in query)
{
    Console.WriteLine(s);
}

/* This code produces the following output:

    an
    apple
    a
    day
    the
    quick
    brown
    fox
*/
```

A consulta equivalente usando a sintaxe do método é mostrada no código a seguir:

C#

```
List<string> phrases = ["an apple a day", "the quick brown fox"];

var query = phrases.SelectMany(phrase => phrase.Split(' '));

foreach (string s in query)
{
    Console.WriteLine(s);
}

/* This code produces the following output:

    an
    apple
    a
    day
    the
    quick
    brown
```

```
    fox
*/
```

O método `SelectMany` também pode formar a combinação de correspondência de cada item na primeira sequência com cada item na segunda sequência:

C#

```
var query = from number in numbers
            from letter in letters
            select (number, letter);

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

A consulta equivalente usando a sintaxe do método é mostrada no código a seguir:

C#

```
var method = numbers
    .SelectMany(number => letters,
    (number, letter) => (number, letter));

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

`Zip`

Há várias sobrecargas para o operador de projeção `Zip`. Todos os `Zip` métodos funcionam em sequências de dois ou mais tipos possivelmente heterogêneos. As duas primeiras sobrecargas retornam tuplas com o tipo posicional correspondente das sequências fornecidas.

Considere as seguintes coleções:

C#

```
// An int array with 7 elements.
IEnumerable<int> numbers = [1, 2, 3, 4, 5, 6, 7];
// A char array with 6 elements.
IEnumerable<char> letters = ['A', 'B', 'C', 'D', 'E', 'F'];
```

Para projetar essas sequências em conjunto, use o Enumerable.Zip<TFirst,TSecond> (IEnumerable<TFirst>, IEnumerable<TSecond>) operador:

C#

```
foreach ((int number, char letter) in numbers.Zip(letters))
{
    Console.WriteLine($"Number: {number} zipped with letter: '{letter}'");
}
// This code produces the following output:
//     Number: 1 zipped with letter: 'A'
//     Number: 2 zipped with letter: 'B'
//     Number: 3 zipped with letter: 'C'
//     Number: 4 zipped with letter: 'D'
//     Number: 5 zipped with letter: 'E'
//     Number: 6 zipped with letter: 'F'
```

> **ⓘ Importante**
>
> A sequência resultante de uma operação zip nunca tem mais comprimento do que a sequência mais curta. As coleções `numbers` e `letters` diferem em comprimento, e a sequência resultante omite o último elemento da coleção `numbers`, pois não tem nada com o qual combinar.

A segunda sobrecarga aceita uma sequência `third`. Vamos criar outra coleção, ou seja `emoji`:

C#

```
// A string array with 8 elements.
IEnumerable<string> emoji = [ "🤯", "🔥", "🎉", "👀", "⭐", "💜", "✓", "💯"];
```

Para projetar essas sequências em conjunto, use o Enumerable.Zip<TFirst,TSecond,TThird> (IEnumerable<TFirst>, IEnumerable<TSecond>, IEnumerable<TThird>) operador:

C#

```
foreach ((int number, char letter, string em) in numbers.Zip(letters, emoji))
{
    Console.WriteLine(
        $"Number: {number} is zipped with letter: '{letter}' and emoji: {em}");
}
// This code produces the following output:
//     Number: 1 is zipped with letter: 'A' and emoji: 🤯
//     Number: 2 is zipped with letter: 'B' and emoji: 🔥
//     Number: 3 is zipped with letter: 'C' and emoji: 🎉
//     Number: 4 is zipped with letter: 'D' and emoji: 👀
```

```
//     Number: 5 is zipped with letter: 'E' and emoji: ⭐
//     Number: 6 is zipped with letter: 'F' and emoji: 💜
```

Assim como a sobrecarga anterior, o `Zip` método projeta uma tupla, mas desta vez com três elementos.

A terceira sobrecarga aceita um `Func<TFirst, TSecond, TResult>` argumento que atua como um seletor de resultados. Você pode projetar uma nova sequência resultante das sequências que estão sendo compactadas.

C#

```
foreach (string result in
    numbers.Zip(letters, (number, letter) => $"{number} = {letter}
({(int)letter})"))
{
    Console.WriteLine(result);
}
// This code produces the following output:
//     1 = A (65)
//     2 = B (66)
//     3 = C (67)
//     4 = D (68)
//     5 = E (69)
//     6 = F (70)
```

Com a sobrecarga anterior `Zip` , a função especificada é aplicada aos elementos correspondentes `number` e `letter`, produzindo uma sequência dos `string` resultados.

`Select` contra `SelectMany`

O trabalho de ambos `Select` e `SelectMany` é produzir um valor de resultado (ou valores) de valores de origem. `Select` produz um valor de resultado para cada valor de origem. O resultado geral é, portanto, uma coleção que tem o mesmo número de elementos que a coleção de origem. Por outro lado, `SelectMany` produz um único resultado geral que contém subcolleções concatenadas de cada valor de origem. A função de transformação que é passada como um argumento para `SelectMany` deve retornar uma sequência enumerável de valores para cada valor de origem. `SelectMany` concatena essas sequências enumeráveis para criar uma sequência grande.

As duas ilustrações a seguir mostram a diferença conceitual entre as ações desses dois métodos. Em cada caso, presuma que a função de seletor (transformação) seleciona a matriz de flores de cada valor de origem.

Esta ilustração mostra como `Select` retorna uma coleção que tem o mesmo número de elementos que a coleção de origem.

Esta ilustração descreve como `SelectMany` concatena a sequência intermediária de matrizes em um valor de resultado final que contém cada valor de cada matriz intermediária.

Exemplo de código

O exemplo a seguir compara o comportamento de `Select` e `SelectMany`. O código cria um "buquê" de flores tirando os itens de cada lista de nomes de flores na coleção de origem. No exemplo a seguir, o "valor único" usado pela função Select<TSource,TResult> (IEnumerable<TSource>, Func<TSource,TResult>) de transformação é uma coleção de valores. Este exemplo requer o loop extra `foreach` para enumerar cada cadeia de caracteres em cada subsequência.

C#

```
class Bouquet
{
    public required List<string> Flowers { get; init; }
```

```
}

static void SelectVsSelectMany()
{
    List<Bouquet> bouquets =
    [
        new Bouquet { Flowers = ["sunflower", "daisy", "daffodil", "larkspur"] },
        new Bouquet { Flowers = ["tulip", "rose", "orchid"] },
        new Bouquet { Flowers = ["gladiolis", "lily", "snapdragon", "aster",
"protea"] },
        new Bouquet { Flowers = ["larkspur", "lilac", "iris", "dahlia"] }
    ];

    IEnumerable<List<string>> query1 = bouquets.Select(bq => bq.Flowers);

    IEnumerable<string> query2 = bouquets.SelectMany(bq => bq.Flowers);

    Console.WriteLine("Results by using Select():");
    // Note the extra foreach loop here.
    foreach (IEnumerable<string> collection in query1)
    {
        foreach (string item in collection)
        {
            Console.WriteLine(item);
        }
    }

    Console.WriteLine("\nResults by using SelectMany():");
    foreach (string item in query2)
    {
        Console.WriteLine(item);
    }
}
```

Consulte também

- System.Linq
- cláusula select
- Como preencher coleções de objetos de várias fontes (LINQ) (C#)
- Como dividir um arquivo em muitos arquivos usando grupos (LINQ) (C#)

Operações de conjunto (C#)

17/06/2025

Definir operações no LINQ referem-se a operações de consulta que produzem um conjunto de resultados baseado na presença ou ausência de elementos equivalentes dentro das mesmas coleções ou coleções separadas.

> **Importante**
>
> Esses exemplos usam uma fonte de dados System.Collections.Generic.IEnumerable<T>. Fontes de dados baseadas em System.Linq.IQueryProvider usam as fontes de dados System.Linq.IQueryable<T> e as árvores de expressão. As árvores de expressão possuem limitações na sintaxe C# permitida. Além disso, todas as fontes de dados `IQueryProvider`, como EF Core, podem impor mais restrições. Verifique a documentação da fonte de dados.

Expandir a tabela

Nomes de método	Descrição	Sintaxe de expressão de consulta em C#	Mais informações
`Distinct` ou `DistinctBy`	Remove os valores duplicados de uma coleção.	Não aplicável.	Enumerable.Distinct Enumerable.DistinctBy Queryable.Distinct Queryable.DistinctBy
`Except` ou `ExceptBy`	Retorna a diferença de conjunto, que significa os elementos de uma coleção que não aparecem em uma segunda coleção.	Não aplicável.	Enumerable.Except Enumerable.ExceptBy Queryable.Except Queryable.ExceptBy
`Intersect` ou `IntersectBy`	Retorna a interseção de conjunto, o que significa os elementos que aparecem em cada uma das duas coleções.	Não aplicável.	Enumerable.Intersect Enumerable.IntersectBy Queryable.Intersect Queryable.IntersectBy
`Union` ou `UnionBy`	Retorna a união de conjunto, o que significa os elementos únicos que aparecem em qualquer uma das duas coleções.	Não aplicável.	Enumerable.Union Enumerable.UnionBy Queryable.Union Queryable.UnionBy

`Distinct` e `DistinctBy`

O exemplo a seguir descreve o comportamento do método Enumerable.Distinct em uma sequência de cadeias de caracteres. A sequência retornada contém os elementos exclusivos da sequência de entrada.

C#

```
string[] words = ["the", "quick", "brown", "fox", "jumped", "over", "the", "lazy",
"dog"];

IEnumerable<string> query = from word in words.Distinct()
                            select word;

foreach (var str in query)
{
    Console.WriteLine(str);
}

/* This code produces the following output:
 *
 * the
 * quick
 * brown
 * fox
 * jumped
 * over
 * lazy
 * dog
 */
```

DistinctBy é uma abordagem alternativa a `Distinct` que usa `keySelector`. `keySelector` é usado como o discriminador comparativo do tipo de origem. No seguinte código, as palavras são discriminados com base no respectivo `Length`, e a primeira palavra de cada tipo é exibida:

C#

```
string[] words = ["the", "quick", "brown", "fox", "jumped", "over", "the", "lazy",
"dog"];

foreach (string word in words.DistinctBy(p => p.Length))
{
    Console.WriteLine(word);
}

// This code produces the following output:
//     the
//     quick
```

```
//      jumped
//      over
```

`Except` e `ExceptBy`

O exemplo a seguir descreve o comportamento de Enumerable.Except. A sequência retornada contém apenas os elementos da primeira sequência de entrada que não estão na segunda sequência de entrada.

> **Observação**
>
> Os exemplos a seguir neste artigo usam as fontes de dados comuns para essa área. Cada `Student` tem um nível de escolaridade, um departamento primário e uma série de pontuações. Um `Teacher` também tem uma propriedade `City` que identifica o campus onde o docente ministra aulas. A `Department` tem um nome e uma referência a um `Teacher` que atua como chefe do departamento.
> Você pode encontrar o conjunto de dados de exemplo no repositório de origem.

C#

```
public enum GradeLevel
{
    FirstYear = 1,
    SecondYear,
    ThirdYear,
    FourthYear
};

public class Student
{
    public required string FirstName { get; init; }
    public required string LastName { get; init; }
    public required int ID { get; init; }

    public required GradeLevel Year { get; init; }
    public required List<int> Scores { get; init; }

    public required int DepartmentID { get; init; }
}

public class Teacher
```

```
{
    public required string First { get; init; }
    public required string Last { get; init; }
    public required int ID { get; init; }
    public required string City { get; init; }
}

public class Department
{
    public required string Name { get; init; }
    public int ID { get; init; }

    public required int TeacherID { get; init; }
}
```

C#

```
string[] words1 = ["the", "quick", "brown", "fox"];
string[] words2 = ["jumped", "over", "the", "lazy", "dog"];

IEnumerable<string> query = from word in words1.Except(words2)
                            select word;

foreach (var str in query)
{
    Console.WriteLine(str);
}

/* This code produces the following output:
 *
 * quick
 * brown
 * fox
 */
```

O método ExceptBy é uma abordagem alternativa a `Except` que usa duas sequências de tipos possivelmente heterogêneos e um `keySelector`. O `keySelector` é o mesmo tipo que o tipo da primeira coleção. Considere a matriz `Teacher` a seguir e IDs de professor a serem excluídas. Para encontrar professores na primeira coleção que não estão na segunda coleção, você pode projetar a ID do professor na segunda coleção:

C#

```
int[] teachersToExclude =
[
    901,    // English
    965,    // Mathematics
    932,    // Engineering
    945,    // Economics
    987,    // Physics
```

```
    901     // Chemistry
];

foreach (Teacher teacher in
    teachers.ExceptBy(
        teachersToExclude, teacher => teacher.ID))
{
    Console.WriteLine($"{teacher.First} {teacher.Last}");
}
```

No código anterior do C#:

- A matriz `teachers` é filtrada apenas para os professores que não estão na matriz `teachersToExclude`.
- A matriz `teachersToExclude` contém o valor `ID` de todos os chefes de departamento.
- A chamada para `ExceptBy` resulta em um novo conjunto de valores que são gravados no console.

O novo conjunto de valores é do tipo `Teacher`, que é o tipo da primeira coleção. Cada `teacher` na matriz `teachers` que não tem um valor de ID correspondente na matriz `teachersToExclude` é gravada no console.

`Intersect` e `IntersectBy`

O exemplo a seguir descreve o comportamento de Enumerable.Intersect. A sequência retornada contém os elementos que são comuns a ambas as sequências de entrada.

C#

```
string[] words1 = ["the", "quick", "brown", "fox"];
string[] words2 = ["jumped", "over", "the", "lazy", "dog"];

IEnumerable<string> query = from word in words1.Intersect(words2)
                            select word;

foreach (var str in query)
{
    Console.WriteLine(str);
}

/* This code produces the following output:
 *
```

```
 * the
 */
```

O método IntersectBy é uma abordagem alternativa a `Intersect` que usa duas sequências de tipos possivelmente heterogêneos e um `keySelector`. O `keySelector` é usado como o discriminatório comparativo do tipo da segunda coleção. Considere as matrizes de alunos e professores a seguir. A consulta corresponde aos itens em cada sequência por nome para localizar os alunos que também são professores:

C#

```
foreach (Student person in
    students.IntersectBy(
        teachers.Select(t => (t.First, t.Last)), s => (s.FirstName, s.LastName)))
{
    Console.WriteLine($"{person.FirstName} {person.LastName}");
}
```

No código anterior do C#:

- A consulta produz a interseção do `Teacher` e `Student` comparando nomes.
- Somente as pessoas encontradas em ambas as matrizes estão presentes na sequência resultante.
- As instâncias resultantes `Student` são gravadas no console.

`Union` e `UnionBy`

O exemplo a seguir descreve uma operação de união em duas sequências de cadeias de caracteres. A sequência retornada contém os elementos exclusivos das duas sequências de entrada.

C#

```
string[] words1 = ["the", "quick", "brown", "fox"];
string[] words2 = ["jumped", "over", "the", "lazy", "dog"];

IEnumerable<string> query = from word in words1.Union(words2)
                            select word;

foreach (var str in query)
```

```
{
    Console.WriteLine(str);
}

/* This code produces the following output:
 *
 * the
 * quick
 * brown
 * fox
 * jumped
 * over
 * lazy
 * dog
*/
```

O método UnionBy é uma abordagem alternativa a `Union` que usa duas sequências do mesmo tipo e uma `keySelector`. `keySelector` é usado como o discriminador comparativo do tipo de origem. A consulta a seguir produz a lista de todas as pessoas que são alunos ou professores. Os alunos que também são professores são adicionados ao conjunto sindical apenas uma vez:

C#

```
foreach (var person in
    students.Select(s => (s.FirstName, s.LastName)).UnionBy(
        teachers.Select(t => (FirstName: t.First, LastName: t.Last)), s =>
(s.FirstName, s.LastName)))
{
    Console.WriteLine($"{person.FirstName} {person.LastName}");
}
```

No código anterior do C#:

- As matrizes `teachers` e `students` são tecidas usando os nomes delas como o seletor de chaves.
- OS nomes resultantes são gravados no console.

Confira também

- System.Linq
- Como localizar a diferença de conjunto entre duas listas (LINQ) (C#)

Classificando dados (C#)

Artigo • 31/10/2024

Uma operação de classificação ordena os elementos de uma sequência com base em um ou mais atributos. O primeiro critério de classificação executa uma classificação primária dos elementos. Especificando um segundo critério de classificação, você pode classificar os elementos dentro de cada grupo de classificação primário.

> **ⓘ Importante**
>
> Esses exemplos usam uma fonte de dados System.Collections.Generic.IEnumerable<T>. Fontes de dados baseadas em System.Linq.IQueryProvider usam as fontes de dados System.Linq.IQueryable<T> e as árvores de expressão. As árvores de expressão possuem limitações na sintaxe C# permitida. Além disso, todas as fontes de dados `IQueryProvider`, como EF Core, podem impor mais restrições. Verifique a documentação da fonte de dados.

A ilustração a seguir mostra os resultados de uma operação de classificação alfabética em uma sequência de caracteres:

Os métodos de operador de consulta padrão que classificam dados estão listados na seção a seguir.

Métodos

Expandir a tabela

Nome do método	Descrição	Sintaxe de expressão de consulta C#	Mais informações
OrderBy	Classifica valores em ordem crescente.	`orderby`	Enumerable.OrderBy Queryable.OrderBy

Nome do método	Descrição	Sintaxe de expressão de consulta C#	Mais informações
OrderByDescending	Classifica valores em ordem decrescente.	`orderby … descending`	Enumerable.OrderByDescending Queryable.OrderByDescending
ThenBy	Executa uma classificação secundária em ordem crescente.	`orderby …, …`	Enumerable.ThenBy Queryable.ThenBy
ThenByDescending	Executa uma classificação secundária em ordem decrescente.	`orderby …, … descending`	Enumerable.ThenByDescending Queryable.ThenByDescending
Reverse	Inverte a ordem dos elementos em uma coleção.	Não aplicável.	Enumerable.Reverse Queryable.Reverse

Observação

Os exemplos a seguir neste artigo usam as fontes de dados comuns para essa área. Cada `Student` tem um nível de escolaridade, um departamento primário e uma série de pontuações. Um `Teacher` também tem uma propriedade `City` que identifica o campus onde o docente ministra aulas. A `Department` tem um nome e uma referência a um `Teacher` que atua como chefe do departamento.
Você pode encontrar o conjunto de dados de exemplo no repositório de origem.

C#

```
public enum GradeLevel
{
    FirstYear = 1,
    SecondYear,
    ThirdYear,
    FourthYear
};

public class Student
{
    public required string FirstName { get; init; }
    public required string LastName { get; init; }
    public required int ID { get; init; }

    public required GradeLevel Year { get; init; }
```

```
    public required List<int> Scores { get; init; }

    public required int DepartmentID { get; init; }
}

public class Teacher
{
    public required string First { get; init; }
    public required string Last { get; init; }
    public required int ID { get; init; }
    public required string City { get; init; }
}

public class Department
{
    public required string Name { get; init; }
    public int ID { get; init; }

    public required int TeacherID { get; init; }
}
```

Classificação crescente primária

O exemplo a seguir demonstra como usar a cláusula `orderby` em uma consulta LINQ para classificar a matriz de professores por nome de família, em ordem crescente.

C#

```
IEnumerable<string> query = from teacher in teachers
                            orderby teacher.Last
                            select teacher.Last;

foreach (string str in query)
{
    Console.WriteLine(str);
}
```

A consulta equivalente escrita usando a sintaxe do método é mostrada no seguinte código:

C#

```
IEnumerable<string> query = teachers
    .OrderBy(teacher => teacher.Last)
    .Select(teacher => teacher.Last);

foreach (string str in query)
{
```

```
    Console.WriteLine(str);
}
```

Classificação decrescente primária

O exemplo a seguir demonstra como usar a cláusula `orderby descending` em uma consulta LINQ para classificar os professores pelo nome da família, em ordem decrescente.

```C#
IEnumerable<string> query = from teacher in teachers
                            orderby teacher.Last descending
                            select teacher.Last;

foreach (string str in query)
{
    Console.WriteLine(str);
}
```

A consulta equivalente escrita usando a sintaxe do método é mostrada no seguinte código:

```C#
IEnumerable<string> query = teachers
    .OrderByDescending(teacher => teacher.Last)
    .Select(teacher => teacher.Last);

foreach (string str in query)
{
    Console.WriteLine(str);
}
```

Classificação crescente secundária

O exemplo a seguir demonstra como usar a cláusula `orderby` em uma consulta LINQ para executar uma classificação primária e secundária. Os professores são classificados principalmente pela cidade e, em segundo lugar, pelo nome da família, ambos em ordem crescente.

```C#
```

```
IEnumerable<(string, string)> query = from teacher in teachers
                                      orderby teacher.City, teacher.Last
                                      select (teacher.Last, teacher.City);

foreach ((string last, string city) in query)
{
    Console.WriteLine($"City: {city}, Last Name: {last}");
}
```

A consulta equivalente escrita usando a sintaxe do método é mostrada no seguinte código:

C#

```
IEnumerable<(string, string)> query = teachers
    .OrderBy(teacher => teacher.City)
    .ThenBy(teacher => teacher.Last)
    .Select(teacher => (teacher.Last, teacher.City));

foreach ((string last, string city) in query)
{
    Console.WriteLine($"City: {city}, Last Name: {last}");
}
```

Classificação decrescente secundária

O exemplo seguinte demonstra como usar a cláusula `orderby descending` em uma consulta de LINQ para executar uma classificação primária, em ordem crescente e uma classificação secundária, em ordem decrescente. Os professores são classificados principalmente pela cidade e, em segundo lugar, pelo nome da família.

C#

```
IEnumerable<(string, string)> query = from teacher in teachers
                                      orderby teacher.City, teacher.Last descending
                                      select (teacher.Last, teacher.City);

foreach ((string last, string city) in query)
{
    Console.WriteLine($"City: {city}, Last Name: {last}");
}
```

A consulta equivalente escrita usando a sintaxe do método é mostrada no seguinte código:

C#

```
IEnumerable<(string, string)> query = teachers
    .OrderBy(teacher => teacher.City)
    .ThenByDescending(teacher => teacher.Last)
    .Select(teacher => (teacher.Last, teacher.City));

foreach ((string last, string city) in query)
{
    Console.WriteLine($"City: {city}, Last Name: {last}");
}
```

Confira também

- System.Linq
- Cláusula orderby
- Como classificar ou filtrar dados de texto por qualquer palavra ou campo (LINQ) (C#)

Operações de quantificador no LINQ (C#)

17/06/2025

As operações de quantificador retornam um Boolean valor que indica se alguns ou todos os elementos em uma sequência atendem a uma condição.

> **ⓘ Importante**
>
> Esses exemplos usam uma fonte de dados System.Collections.Generic.IEnumerable<T>. Fontes de dados baseadas em System.Linq.IQueryProvider usam as fontes de dados System.Linq.IQueryable<T> e as árvores de expressão. As árvores de expressão possuem limitações na sintaxe C# permitida. Além disso, todas as fontes de dados `IQueryProvider`, como EF Core, podem impor mais restrições. Verifique a documentação da fonte de dados.

A ilustração a seguir ilustra duas operações de quantificador diferentes em duas sequências de origem diferentes. A primeira operação pergunta se algum dos elementos é o caractere 'A'. A segunda operação pergunta se todos os elementos são o caractere 'A'. Ambos os métodos retornam `true` neste exemplo.

Expandir a tabela

Nome do método	Descrição	Sintaxe de expressão de consulta C#	Mais informações
Todos	Determina se todos os elementos em uma sequência atendem a uma condição.	Não aplicável.	Enumerable.All Queryable.All
Qualquer	Determina se algum elemento em uma sequência atende a uma condição.	Não aplicável.	Enumerable.Any Queryable.Any
Contém	Determina se uma sequência contém um elemento especificado.	Não aplicável.	Enumerable.Contains Queryable.Contains

Todos

O exemplo a seguir usa o `All` para encontrar alunos que pontuaram acima de 70 em todos os exames.

C#

```
IEnumerable<string> names = from student in students
                            where student.Scores.All(score => score > 70)
                            select $"{student.FirstName} {student.LastName}:
{string.Join(", ", student.Scores.Select(s => s.ToString()))}";

foreach (string name in names)
{
    Console.WriteLine($"{name}");
}

// This code produces the following output:
//
// Cesar Garcia: 71, 86, 77, 97
// Nancy Engström: 75, 73, 78, 83
// Ifunanya Ugomma: 84, 82, 96, 80
```

Qualquer

O exemplo a seguir usa o `Any` para encontrar alunos que obtiveram pontuação superior a 95 em algum exame.

C#

```
IEnumerable<string> names = from student in students
                            where student.Scores.Any(score => score > 95)
                            select $"{student.FirstName} {student.LastName}:
{student.Scores.Max()}";

foreach (string name in names)
{
    Console.WriteLine($"{name}");
}

// This code produces the following output:
//
// Svetlana Omelchenko: 97
// Cesar Garcia: 97
// Debra Garcia: 96
// Ifeanacho Jamuike: 98
// Ifunanya Ugomma: 96
// Michelle Caruana: 97
// Nwanneka Ifeoma: 98
```

```
// Martina Mattsson: 96
// Anastasiya Sazonova: 96
// Jesper Jakobsson: 98
// Max Lindgren: 96
```

Contém

O exemplo a seguir usa o `Contains` para encontrar alunos que pontuaram exatamente 95 em um exame.

```C#
IEnumerable<string> names = from student in students
                            where student.Scores.Contains(95)
                            select $"{student.FirstName} {student.LastName}:
{string.Join(", ", student.Scores.Select(s => s.ToString()))}";

foreach (string name in names)
{
    Console.WriteLine($"{name}");
}

// This code produces the following output:
//
// Claire O'Donnell: 56, 78, 95, 95
// Donald Urquhart: 92, 90, 95, 57
```

Consulte também

- System.Linq
- Especificar filtros de predicado dinamicamente durante a execução
- Como consultar frases que contêm um conjunto de palavras especificado (LINQ) (C#)

Particionamento de dados (C#)

19/06/2025

Particionamento no LINQ refere-se à operação de divisão de uma sequência de entrada em duas seções, sem reorganizar os elementos e, em seguida, retornar uma das seções.

> **Importante**
>
> Esses exemplos usam uma fonte de dados System.Collections.Generic.IEnumerable<T>. Fontes de dados baseadas em System.Linq.IQueryProvider usam as fontes de dados System.Linq.IQueryable<T> e as árvores de expressão. As árvores de expressão possuem limitações na sintaxe C# permitida. Além disso, todas as fontes de dados `IQueryProvider`, como EF Core, podem impor mais restrições. Verifique a documentação da fonte de dados.

A ilustração a seguir mostra os resultados de três operações de particionamento diferentes em uma sequência de caracteres. A primeira operação retorna os três primeiros elementos na sequência. A segunda operação ignora os três primeiros elementos e retorna os elementos restantes. A terceira operação ignora os dois primeiros elementos na sequência e retorna os três elementos seguintes.

Os métodos do operador de consulta padrão que particionam as sequências estão listados na seção a seguir.

Operadores

Expandir a tabela

Nomes de método	Descrição	Sintaxe da expressão de consulta C#	Mais informações
Ignorar	Ignora elementos até uma posição especificada na sequência.	Não aplicável.	Enumerable.Skip Queryable.Skip
SkipWhile	Ignora elementos com base em uma função de predicado até que um elemento não atenda à condição.	Não aplicável.	Enumerable.SkipWhile Queryable.SkipWhile
Tomar	Leva elementos até uma posição especificada em uma sequência.	Não aplicável.	Enumerable.Take Queryable.Take
TakeWhile	Seleciona elementos com base em uma função condicional até que um elemento não satisfaça a condição.	Não aplicável.	Enumerable.TakeWhile Queryable.TakeWhile
Pedaço	Divide os elementos de uma sequência em partes de um tamanho máximo especificado.	Não aplicável.	Enumerable.Chunk Queryable.Chunk

Todos os exemplos a seguir usam Enumerable.Range(Int32, Int32) para gerar uma sequência de números de 0 a 7.

Use o `Take` método para pegar apenas os primeiros elementos em uma sequência:

C#

```
foreach (int number in Enumerable.Range(0, 8).Take(3))
{
    Console.WriteLine(number);
}
// This code produces the following output:
// 0
// 1
// 2
```

Use o `Skip` método para ignorar os primeiros elementos em uma sequência e usar os elementos restantes:

C#

```
foreach (int number in Enumerable.Range(0, 8).Skip(3))
{
    Console.WriteLine(number);
}
// This code produces the following output:
// 3
```

```
// 4
// 5
// 6
// 7
```

Os métodos `TakeWhile` e `SkipWhile` também pegam e ignorar elementos em uma sequência. No entanto, em vez de um número definido de elementos, esses métodos ignoram ou tomam elementos com base em uma condição. `TakeWhile` usa os elementos de uma sequência até que um elemento não corresponda à condição.

C#

```
foreach (int number in Enumerable.Range(0, 8).TakeWhile(n => n < 5))
{
    Console.WriteLine(number);
}
// This code produces the following output:
// 0
// 1
// 2
// 3
// 4
```

`SkipWhile` ignora os primeiros elementos, desde que a condição seja verdadeira. O primeiro elemento que não corresponde à condição e todos os elementos subsequentes são retornados.

C#

```
foreach (int number in Enumerable.Range(0, 8).SkipWhile(n => n < 5))
{
    Console.WriteLine(number);
}
// This code produces the following output:
// 5
// 6
// 7
```

O `Chunk` operador é usado para dividir elementos de uma sequência com base em um determinado `size`.

C#

```
int chunkNumber = 1;
foreach (int[] chunk in Enumerable.Range(0, 8).Chunk(3))
{
    Console.WriteLine($"Chunk {chunkNumber++}:");
    foreach (int item in chunk)
```

```
    {
        Console.WriteLine($"    {item}");
    }

    Console.WriteLine();
}
// This code produces the following output:
// Chunk 1:
//    0
//    1
//    2
//
//Chunk 2:
//    3
//    4
//    5
//
//Chunk 3:
//    6
//    7
```

O código C# anterior:

- Depende de Enumerable.Range(Int32, Int32) para gerar uma sequência de números.
- Aplica o `Chunk` operador, dividindo a sequência em partes com um tamanho máximo de três.

Consulte também

- System.Linq

Convertendo Tipos de Dados (C#)

Artigo • 31/10/2024

Os métodos de conversão alteram o tipo dos objetos de entrada.

> ⓘ **Importante**
>
> Esses exemplos usam uma fonte de dados System.Collections.Generic.IEnumerable<T>. Fontes de dados baseadas em System.Linq.IQueryProvider usam as fontes de dados System.Linq.IQueryable<T> e as árvores de expressão. As árvores de expressão possuem limitações na sintaxe C# permitida. Além disso, todas as fontes de dados `IQueryProvider`, como EF Core, podem impor mais restrições. Verifique a documentação da fonte de dados.

As operações de conversão em consultas LINQ são úteis em diversos aplicativos. A seguir estão alguns exemplos:

- O método Enumerable.AsEnumerable pode ser usado para ocultar a implementação personalizada de um tipo de um operador de consulta padrão.
- O método Enumerable.OfType pode ser usado para habilitar coleções sem parâmetros para consulta LINQ.
- Os métodos Enumerable.ToArray, Enumerable.ToDictionary, Enumerable.ToList e Enumerable.ToLookup podem ser usados para forçar a execução de consulta imediata em vez de adiá-la até que a consulta seja enumerada.

Métodos

A tabela a seguir lista os métodos de operador de consulta padrão que realizam conversões de tipo de dados.

Os métodos de conversão nesta tabela cujos nomes começam com "As" alteram o tipo estático da coleção de origem, mas não a enumeram. Os métodos cujos nomes começam com "To" enumeram a coleção de origem e colocam os itens na coleção de tipo correspondente.

Expandir a tabela

Nome do método	Descrição	Sintaxe de expressão de consulta C#	Mais informações
AsEnumerable	Retorna a entrada digitada como IEnumerable<T>.	Não aplicável.	Enumerable.AsEnumerable
AsQueryable	Converte um IEnumerable (genérico) em um IQueryable (genérico).	Não aplicável.	Queryable.AsQueryable
Conversão	Converte os elementos de uma coleção em um tipo especificado.	Use uma variável de intervalo de tipo explícito. Por exemplo: `from string str in words`	Enumerable.Cast Queryable.Cast
OfType	Filtra valores, dependendo da capacidade de serem convertidos em um tipo especificado.	Não aplicável.	Enumerable.OfType Queryable.OfType
ToArray	Converte uma coleção em uma matriz. Esse método força a execução de consulta.	Não aplicável.	Enumerable.ToArray
ToDictionary	Coloca os elementos em um Dictionary<TKey,TValue> com base em uma função de seletor de chave. Esse método força a execução de consulta.	Não aplicável.	Enumerable.ToDictionary
ToList	Converte uma coleção em um List<T>. Esse método força a execução de consulta.	Não aplicável.	Enumerable.ToList
ToLookup	Coloca os elementos em um Lookup<TKey,TElement> (um dicionário one-to-many) com base em uma função de seletor de chave. Esse método força a execução de consulta.	Não aplicável.	Enumerable.ToLookup

Observação

Os exemplos a seguir neste artigo usam as fontes de dados comuns para essa área. Cada `Student` tem um nível de escolaridade, um departamento primário e uma

série de pontuações. Um `Teacher` também tem uma propriedade `City` que identifica o campus onde o docente ministra aulas. A `Department` tem um nome e uma referência a um `Teacher` que atua como chefe do departamento.
Você pode encontrar o conjunto de dados de exemplo no repositório de origem.

C#

```
public enum GradeLevel
{
    FirstYear = 1,
    SecondYear,
    ThirdYear,
    FourthYear
};

public class Student
{
    public required string FirstName { get; init; }
    public required string LastName { get; init; }
    public required int ID { get; init; }

    public required GradeLevel Year { get; init; }
    public required List<int> Scores { get; init; }

    public required int DepartmentID { get; init; }
}

public class Teacher
{
    public required string First { get; init; }
    public required string Last { get; init; }
    public required int ID { get; init; }
    public required string City { get; init; }
}

public class Department
{
    public required string Name { get; init; }
    public int ID { get; init; }

    public required int TeacherID { get; init; }
}
```

Exemplo de sintaxe de expressão de consulta

O exemplo de código a seguir usa uma variável de intervalo de tipo explícito para converter um tipo em um subtipo antes de acessar um membro que está disponível somente no subtipo.

```C#
IEnumerable people = students;

var query = from Student student in students
            where student.Year == GradeLevel.ThirdYear
            select student;

foreach (Student student in query)
{
    Console.WriteLine(student.FirstName);
}
```

A consulta equivalente pode ser expressa usando a sintaxe do método, conforme mostrado no seguinte exemplo:

```C#
IEnumerable people = students;

var query = people
    .Cast<Student>()
    .Where(student => student.Year == GradeLevel.ThirdYear);

foreach (Student student in query)
{
    Console.WriteLine(student.FirstName);
}
```

Confira também

- System.Linq
- Cláusula From

Join Operações em LINQ

Artigo • 30/01/2025

Uma junção de duas fontes de dados é a associação de objetos em uma fonte de dados com objetos que compartilham um atributo comum em outra fonte de dados.

> ⓘ **Importante**
>
> Esses exemplos usam uma fonte de dados System.Collections.Generic.IEnumerable<T>. Fontes de dados baseadas em System.Linq.IQueryProvider usam as fontes de dados System.Linq.IQueryable<T> e as árvores de expressão. As árvores de expressão possuem limitações na sintaxe C# permitida. Além disso, todas as fontes de dados `IQueryProvider`, como EF Core, podem impor mais restrições. Verifique a documentação da fonte de dados.

A Junção é uma operação importante em consultas que têm como alvo fontes de dados cujas relações entre si não podem ser seguidas diretamente. Na programação orientada a objetos, a união pode significar uma correlação entre objetos que não é modelada, como a direção inversa de um relacionamento unidirecional. Um exemplo de relacionamento unilateral é uma classe `Student` que possui uma propriedade do tipo `Department` que representa o principal, mas a classe `Department` não possui uma propriedade que seja uma coleção de objetos `Student`. Se você tiver uma lista de objetos `Department` e quiser encontrar todos os alunos em cada departamento, poderá usar uma operação de junção para encontrá-los.

Os métodos de junção fornecidos na estrutura LINQ são Join e GroupJoin. Esses métodos executam junção por igualdade ou junções que correspondem duas fontes de dados com base na igualdade de suas chaves. (Para comparação, Transact-SQL dá suporte a operadores de junção diferentes de `equals`, por exemplo, o operador `less than`.) Em termos de banco de dados relacional, Join implementa uma junção interna, um tipo de junção na qual somente os objetos que têm uma correspondência no outro conjunto de dados são retornados. O método GroupJoin não tem equivalente direto em termos de banco de dados relacional, mas ele implementa um superconjunto de junções internas e junções externas esquerdas. Uma junção externa à esquerda é uma junção que retorna cada elemento da primeira origem de dados (à esquerda), mesmo que não tenha elementos correlacionados na outra origem de dados.

A ilustração a seguir mostra uma visão conceitual de dois conjuntos e os elementos dentro desses conjuntos que estão incluídos em uma junção interna ou externa à esquerda.

Métodos

Expandir a tabela

Nome do método	Descrição	Sintaxe de expressão de consulta C#	Mais informações
Join	Une duas sequências com base nas funções de seletor de chave e extrai pares de valores.	`join … in … on … equals …`	Enumerable.Join Queryable.Join
GroupJoin	Une duas sequências baseadas em funções de seletor de chave e agrupa as correspondências resultantes para cada elemento.	`join … in … on … equals … into …`	Enumerable.GroupJoin Queryable.GroupJoin

Observação

Os exemplos a seguir neste artigo usam as fontes de dados comuns para essa área. Cada `Student` tem um nível de escolaridade, um departamento primário e uma série de pontuações. Um `Teacher` também tem uma propriedade `City` que identifica o campus onde o docente ministra aulas. A `Department` tem um nome e uma referência a um `Teacher` que atua como chefe do departamento.
Você pode encontrar o conjunto de dados de exemplo no repositório de origem.

C#

```
public enum GradeLevel
{
    FirstYear = 1,
    SecondYear,
    ThirdYear,
    FourthYear
```

```
};

public class Student
{
    public required string FirstName { get; init; }
    public required string LastName { get; init; }
    public required int ID { get; init; }

    public required GradeLevel Year { get; init; }
    public required List<int> Scores { get; init; }

    public required int DepartmentID { get; init; }
}

public class Teacher
{
    public required string First { get; init; }
    public required string Last { get; init; }
    public required int ID { get; init; }
    public required string City { get; init; }
}

public class Department
{
    public required string Name { get; init; }
    public int ID { get; init; }

    public required int TeacherID { get; init; }
}
```

O exemplo a seguir usa a cláusula `join … in … on … equals …` para unir duas sequências com base no valor específico:

```
C#

var query = from student in students
            join department in departments on student.DepartmentID equals
department.ID
            select new { Name = $"{student.FirstName} {student.LastName}",
DepartmentName = department.Name };

foreach (var item in query)
{
    Console.WriteLine($"{item.Name} - {item.DepartmentName}");
}
```

A consulta anterior pode ser expressa usando a sintaxe do método conforme mostrado no código a seguir:

```
C#
```

```
var query = students.Join(departments,
    student => student.DepartmentID, department => department.ID,
    (student, department) => new { Name = $"{student.FirstName}
{student.LastName}", DepartmentName = department.Name });

foreach (var item in query)
{
    Console.WriteLine($"{item.Name} - {item.DepartmentName}");
}
```

O exemplo a seguir usa a cláusula `join … in … on … equals … into …` para unir duas sequências com base no valor específico e agrupa as correspondências resultantes para cada elemento:

C#

```
IEnumerable<IEnumerable<Student>> studentGroups = from department in
departments
                    join student in students on department.ID equals
student.DepartmentID into studentGroup
                    select studentGroup;

foreach (IEnumerable<Student> studentGroup in studentGroups)
{
    Console.WriteLine("Group");
    foreach (Student student in studentGroup)
    {
        Console.WriteLine($"  - {student.FirstName}, {student.LastName}");
    }
}
```

A consulta anterior pode ser expressa usando a sintaxe do método conforme mostrado no exemplo a seguir:

C#

```
// Join department and student based on DepartmentId and grouping result
IEnumerable<IEnumerable<Student>> studentGroups =
departments.GroupJoin(students,
    department => department.ID, student => student.DepartmentID,
    (department, studentGroup) => studentGroup);

foreach (IEnumerable<Student> studentGroup in studentGroups)
{
    Console.WriteLine("Group");
    foreach (Student student in studentGroup)
    {
        Console.WriteLine($"  - {student.FirstName}, {student.LastName}");
```

```
    }
}
```

Executar junções internas

Em termos de banco de dados relacional, uma junção interna produz um conjunto de resultados no qual cada elemento da primeira coleção aparece uma vez para cada elemento correspondente na segunda coleção. Se um elemento da primeira coleção não tiver elementos correspondentes, ele não aparecerá no conjunto de resultados. O método Join, que é chamado pela cláusula `join` em C#, implementa uma junção interna. Os exemplos a seguir mostram como executar quatro variações de uma junção interna:

- Uma junção interna simples que correlaciona elementos de duas fontes de dados com base em uma chave simples.
- Uma junção interna que correlaciona elementos de duas fontes de dados com base em uma chave *composta*. Uma chave composta, que é uma chave que consiste em mais de um valor, permite que você correlacione os elementos com base em mais de uma propriedade.
- Uma *junção múltipla* na qual operações join sucessivas são acrescentadas umas às outras.
- Uma junção interna que é implementada por meio de uma junção de grupo.

Junção de chave única

O exemplo a seguir combina `Teacher` objetos com `Department` objetos cujo `TeacherId` corresponde a esse `Teacher`. A cláusula `select` em C# define a aparência dos objetos resultantes. No exemplo a seguir, os objetos resultantes são tipos anônimos que consistem no nome do departamento e no nome do professor que lidera o departamento.

C#

```
var query = from department in departments
            join teacher in teachers on department.TeacherID equals
teacher.ID
            select new
            {
                DepartmentName = department.Name,
                TeacherName = $"{teacher.First} {teacher.Last}"
            };

foreach (var departmentAndTeacher in query)
```

```
{
    Console.WriteLine($"{departmentAndTeacher.DepartmentName} is managed by
{departmentAndTeacher.TeacherName}");
}
```

Você obtém os mesmos resultados usando a sintaxe do método Join:

```
C#

var query = teachers
    .Join(departments, teacher => teacher.ID, department =>
department.TeacherID,
        (teacher, department) =>
        new { DepartmentName = department.Name, TeacherName = $"
{teacher.First} {teacher.Last}" });

foreach (var departmentAndTeacher in query)
{
    Console.WriteLine($"{departmentAndTeacher.DepartmentName} is managed by
{departmentAndTeacher.TeacherName}");
}
```

Os professores que não são chefes de departamento não aparecem nos resultados finais.

Junção de chave composta

Em vez de correlacionar os elementos com base em apenas uma propriedade, você pode usar uma chave composta para comparar elementos com base em várias propriedades. Especifique a função de seletor de chave para cada coleção para retornar um tipo anônimo que consiste nas propriedades que você deseja comparar. Se você rotular as propriedades, elas devem ter o mesmo rótulo no tipo anônimo cada chave. As propriedades também devem aparecer na mesma ordem.

O exemplo a seguir usa uma lista de objetos `Teacher` e uma lista de objetos `Student` para determinar quais professores também são alunos. Ambos os tipos possuem propriedades que representam o nome e o sobrenome de cada pessoa. As funções que criam as chaves de junção dos elementos de cada lista retornam um tipo anônimo que consiste nas propriedades. A operação de junção compara essas chaves compostas para verificar a igualdade e retorna pares de objetos de cada lista em que o primeiro nome e o sobrenome correspondem.

```
C#

// Join the two data sources based on a composite key consisting of first
and last name,
```

```
// to determine which employees are also students.
IEnumerable<string> query =
    from teacher in teachers
    join student in students on new
    {
        FirstName = teacher.First,
        LastName = teacher.Last
    } equals new
    {
        student.FirstName,
        student.LastName
    }
    select teacher.First + " " + teacher.Last;

string result = "The following people are both teachers and students:\r\n";
foreach (string name in query)
{
    result += $"{name}\r\n";
}
Console.Write(result);
```

Você pode usar o método Join, conforme mostrado no exemplo a seguir:

```
C#

IEnumerable<string> query = teachers
    .Join(students,
        teacher => new { FirstName = teacher.First, LastName = teacher.Last
},
        student => new { student.FirstName, student.LastName },
        (teacher, student) => $"{teacher.First} {teacher.Last}"
 );

Console.WriteLine("The following people are both teachers and students:");
foreach (string name in query)
{
    Console.WriteLine(name);
}
```

Junção múltipla

Qualquer número de operações de junção pode ser acrescentadas entre si para executar uma junção múltipla. Cada cláusula `join` em C# correlaciona uma fonte de dados especificada com os resultados da junção anterior.

A primeira cláusula `join` corresponde a alunos e departamentos com base em um `Student` objeto `DepartmentID` correspondente a um `Department` objeto `ID`. Ele retorna uma sequência de tipos anônimos que contêm o objeto `Student` e o objeto `Department`.

A segunda cláusula `join` correlaciona os tipos anônimos retornados pela primeira junção com objetos `Teacher` com base na ID do professor correspondente à ID do chefe do departamento. Ele retorna uma sequência de tipos anônimos que contém o nome do aluno, o nome do departamento e o nome do líder do departamento. Como essa operação é uma junção interna, somente os objetos da primeira fonte de dados que têm uma correspondência na segunda fonte de dados são retornados.

C#

```
// The first join matches Department.ID and Student.DepartmentID from the
list of students and
// departments, based on a common ID. The second join matches teachers who
lead departments
// with the students studying in that department.
var query = from student in students
    join department in departments on student.DepartmentID equals
department.ID
    join teacher in teachers on department.TeacherID equals teacher.ID
    select new {
        StudentName = $"{student.FirstName} {student.LastName}",
        DepartmentName = department.Name,
        TeacherName = $"{teacher.First} {teacher.Last}"
    };

foreach (var obj in query)
{
    Console.WriteLine($"""The student "{obj.StudentName}" studies in the
department run by "{obj.TeacherName}"."""");
}
```

O equivalente usando vários métodos Join usa a mesma abordagem com o tipo anônimo:

C#

```
var query = students
    .Join(departments, student => student.DepartmentID, department =>
department.ID,
        (student, department) => new { student, department })
    .Join(teachers, commonDepartment =>
commonDepartment.department.TeacherID, teacher => teacher.ID,
        (commonDepartment, teacher) => new
        {
            StudentName = $"{commonDepartment.student.FirstName}
{commonDepartment.student.LastName}",
            DepartmentName = commonDepartment.department.Name,
            TeacherName = $"{teacher.First} {teacher.Last}"
        });

foreach (var obj in query)
```

```
{
    Console.WriteLine($"""The student "{obj.StudentName}" studies in the
department run by "{obj.TeacherName}"."""");
}
```

Junção interna usando junção agrupada

O exemplo a seguir mostra como implementar uma junção interna usando uma junção de grupo. A lista de objetos `Department` é unida ao grupo à lista de objetos `Student` com base na `Department.ID` correspondência com a propriedade `Student.DepartmentID`. A junção de grupo cria uma coleção de grupos intermediários, em que cada grupo consiste em um objeto `Department` e uma sequência de `Student` objetos correspondentes. A segunda cláusula `from` combina (ou nivela) esta sequência de sequências em uma sequência mais longa. A cláusula `select` especifica o tipo de elementos na sequência final. Esse tipo é anônimo que consiste no nome do aluno e no nome do departamento correspondente.

C#

```
var query1 =
    from department in departments
    join student in students on department.ID equals student.DepartmentID
into gj
    from subStudent in gj
    select new
    {
        DepartmentName = department.Name,
        StudentName = $"{subStudent.FirstName} {subStudent.LastName}"
    };
Console.WriteLine("Inner join using GroupJoin():");
foreach (var v in query1)
{
    Console.WriteLine($"{v.DepartmentName} - {v.StudentName}");
}
```

Os mesmos resultados podem ser obtidos usando o método GroupJoin da seguinte forma:

C#

```
var queryMethod1 = departments
    .GroupJoin(students, department => department.ID, student =>
student.DepartmentID,
        (department, gj) => new { department, gj })
    .SelectMany(departmentAndStudent => departmentAndStudent.gj,
        (departmentAndStudent, subStudent) => new
        {
```

```
            DepartmentName = departmentAndStudent.department.Name,
            StudentName = $"{subStudent.FirstName} {subStudent.LastName}"
        });

Console.WriteLine("Inner join using GroupJoin():");
foreach (var v in queryMethod1)
{
    Console.WriteLine($"{v.DepartmentName} - {v.StudentName}");
}
```

O resultado é equivalente ao conjunto de resultados obtido usando a cláusula `join` sem a cláusula `into` para executar uma junção interna. O código a seguir demonstra esta consulta equivalente:

C#

```
var query2 = from department in departments
    join student in students on department.ID equals student.DepartmentID
    select new
    {
        DepartmentName = department.Name,
        StudentName = $"{student.FirstName} {student.LastName}"
    };

Console.WriteLine("The equivalent operation using Join():");
foreach (var v in query2)
{
    Console.WriteLine($"{v.DepartmentName} - {v.StudentName}");
}
```

Para evitar o encadeamento, o método Join único pode ser utilizado conforme apresentado aqui:

C#

```
var queryMethod2 = departments.Join(students, departments => departments.ID,
student => student.DepartmentID,
    (department, student) => new
    {
        DepartmentName = department.Name,
        StudentName = $"{student.FirstName} {student.LastName}"
    });

Console.WriteLine("The equivalent operation using Join():");
foreach (var v in queryMethod2)
{
    Console.WriteLine($"{v.DepartmentName} - {v.StudentName}");
}
```

Executar junções agrupadas

A junção de grupo é útil para produzir estruturas de dados hierárquicas. Ela combina cada elemento da primeira coleção com um conjunto de elementos correlacionados da segunda coleção.

> ⓘ **Observação**
>
> Cada elemento da primeira coleção aparece no conjunto de resultados de uma junção de grupo, independentemente de elementos correlacionados serem encontrados na segunda coleção. Caso nenhum elemento correlacionado seja encontrado, a sequência de elementos correlacionados desse elemento ficará vazia. O seletor de resultado, portanto, tem acesso a todos os elementos da primeira coleção. Isso difere do seletor de resultados em uma junção que não é de grupo, que não pode acessar elementos da primeira coleção que não têm correspondência na segunda coleção.

> ⚠ **Aviso**
>
> Enumerable.GroupJoin não tem equivalente direto em termos de banco de dados relacionais tradicionais. No entanto, esse método implementa um superconjunto de junções internas e junções externas à esquerda. Essas operações podem ser expressas em termos de uma junção agrupada. Para obter mais informações, veja Entity Framework Core, GroupJoin.

O primeiro exemplo neste artigo mostra como executar uma junção de grupo. O segundo exemplo mostra como usar uma junção de grupo para criar elementos XML.

Junção de grupo

O exemplo a seguir executa uma junção de grupo de objetos do tipo `Department` e `Student` com base no `Department.ID` correspondente à propriedade `Student.DepartmentID`. Ao contrário de uma junção não-grupo, que produz um par de elementos para cada correspondência, a junção de grupo produz apenas um objeto resultante para cada elemento da primeira coleção, que neste exemplo é um objeto `Department`. Os elementos correspondentes da segunda coleção, que neste exemplo são objetos `Student`, são agrupados em uma coleção. Por fim, a função de seletor de resultado cria um tipo anônimo para cada correspondência que consiste em `Department.Name` e em uma coleção de objetos `Student`.

C#

```
var query = from department in departments
    join student in students on department.ID equals student.DepartmentID
into studentGroup
    select new
    {
        DepartmentName = department.Name,
        Students = studentGroup
    };

foreach (var v in query)
{
    // Output the department's name.
    Console.WriteLine($"{v.DepartmentName}:");

    // Output each of the students in that department.
    foreach (Student? student in v.Students)
    {
        Console.WriteLine($"  {student.FirstName} {student.LastName}");
    }
}
```

No exemplo acima, a variável `query` contém a consulta que cria uma lista onde cada elemento é do tipo anônimo que contém o nome do departamento e uma coleção de alunos que estudam naquele departamento.

A consulta equivalente usando a sintaxe do método é mostrada no código a seguir:

C#

```
var query = departments.GroupJoin(students, department => department.ID,
student => student.DepartmentID,
    (department, Students) => new { DepartmentName = department.Name,
Students });

foreach (var v in query)
{
    // Output the department's name.
    Console.WriteLine($"{v.DepartmentName}:");

    // Output each of the students in that department.
    foreach (Student? student in v.Students)
    {
        Console.WriteLine($"  {student.FirstName} {student.LastName}");
    }
}
```

Junte-se ao grupo para criar XML

As junções de grupo são ideais para a criação de XML usando o LINQ to XML. O exemplo a seguir é semelhante ao exemplo anterior, exceto que em vez de criar tipos anônimos, a função de seletor de resultado cria elementos XML que representam os objetos associados.

```
C#

XElement departmentsAndStudents = new("DepartmentEnrollment",
    from department in departments
    join student in students on department.ID equals student.DepartmentID
into studentGroup
    select new XElement("Department",
        new XAttribute("Name", department.Name),
        from student in studentGroup
        select new XElement("Student",
            new XAttribute("FirstName", student.FirstName),
            new XAttribute("LastName", student.LastName)
        )
    )
);

Console.WriteLine(departmentsAndStudents);
```

A consulta equivalente usando a sintaxe do método é mostrada no código a seguir:

```
C#

XElement departmentsAndStudents = new("DepartmentEnrollment",
    departments.GroupJoin(students, department => department.ID, student =>
student.DepartmentID,
        (department, Students) => new XElement("Department",
            new XAttribute("Name", department.Name),
            from student in Students
            select new XElement("Student",
                new XAttribute("FirstName", student.FirstName),
                new XAttribute("LastName", student.LastName)
            )
        )
    )
);

Console.WriteLine(departmentsAndStudents);
```

Executar junções externas esquerdas

Uma junção externa esquerda é uma junção em que cada elemento da primeira coleção é retornado, mesmo que ele tenha elementos correlacionados na segunda coleção. É

possível usar o LINQ para executar uma junção externa esquerda chamando o método DefaultIfEmpty nos resultados de uma junção de grupo.

O exemplo a seguir demonstra como usar o método DefaultIfEmpty nos resultados de uma junção de grupo para executar uma junção externa esquerda.

A primeira etapa da produção de uma junção externa esquerda de duas coleções é executar uma junção interna usando uma junção de grupo. (Consulte Executar junções internas para obter uma explicação desse processo.) Nesse exemplo, a lista de `Department` objetos é unida internamente à lista de `Student` objetos com base no ID de um `Department` objeto que corresponde ao ID do aluno `DepartmentID`.

A segunda etapa é incluir cada elemento da primeira coleção (esquerda) no conjunto de resultados, mesmo que esse elemento não tenha nenhuma correspondência na coleção direita. Isso é feito chamando DefaultIfEmpty em cada sequência de elementos correspondentes da junção de grupo. Neste exemplo, DefaultIfEmpty é chamado em cada sequência de objetos `Student` correspondentes. O método retorna uma coleção que contém um único valor padrão se a sequência de objetos `Student` correspondentes estiver vazia para qualquer objeto `Department`, garantindo que cada objeto `Department` seja representado na coleção de resultados.

> ⓘ **Observação**
>
> O valor padrão para um tipo de referência é `null`; portanto, o exemplo procura uma referência nula antes de acessar cada elemento de cada coleção `Student`.

C#

```
var query =
    from student in students
    join department in departments on student.DepartmentID equals
department.ID into gj
    from subgroup in gj.DefaultIfEmpty()
    select new
    {
        student.FirstName,
        student.LastName,
        Department = subgroup?.Name ?? string.Empty
    };

foreach (var v in query)
{
    Console.WriteLine($"{v.FirstName:-15} {v.LastName:-15}:
{v.Department}");
}
```

A consulta equivalente usando a sintaxe do método é mostrada no código a seguir:

C#

```
var query = students
    .GroupJoin(
        departments,
        student => student.DepartmentID,
        department => department.ID,
        (student, departmentList) => new { student, subgroup =
departmentList })
    .SelectMany(
        joinedSet => joinedSet.subgroup.DefaultIfEmpty(),
        (student, department) => new
        {
            student.student.FirstName,
            student.student.LastName,
            Department = department.Name
        });

foreach (var v in query)
{
    Console.WriteLine($"{v.FirstName:-15} {v.LastName:-15}:
{v.Department}");
}
```

Confira também

- Join
- GroupJoin
- Tipos anônimos
- Formular junções e consultas entre produtos
- Cláusula join
- Cláusula group
- Como unir conteúdo de arquivos diferentes (LINQ) (C#)
- Como preencher coleções de objetos de várias fontes (LINQ) (C#)

Agrupando dados (C#)

Artigo • 07/06/2024

O agrupamento refere-se à operação de colocação de dados em grupos, de modo que os elementos em cada grupo compartilhem um atributo comum. A ilustração a seguir mostra os resultados do agrupamento de uma sequência de caracteres. A chave para cada grupo é o caractere.

> **ⓘ Importante**
>
> Esses exemplos usam uma fonte de dados System.Collections.Generic.IEnumerable<T>. Fontes de dados baseadas em System.Linq.IQueryProvider usam as fontes de dados System.Linq.IQueryable<T> e as árvores de expressão. As árvores de expressão possuem limitações na sintaxe C# permitida. Além disso, todas as fontes de dados `IQueryProvider`, como EF Core, podem impor mais restrições. Verifique a documentação da fonte de dados.

Os métodos do operador de consulta padrão que agrupam elementos de dados estão listados na tabela a seguir.

Expandir a tabela

Nome do método	Descrição	Sintaxe de expressão de consulta C#	Mais informações

GroupBy	Agrupa elementos que compartilham um atributo comum. Um objeto IGrouping<TKey,TElement> representa cada grupo.	`group … by`
		-ou-
		`group … by … into …`
			Enumerable.GroupBy
			Queryable.GroupBy

Nome do método	Descrição	Sintaxe de expressão de consulta C#	Mais informações
ToLookup	Insere os elementos em um Lookup<TKey,TElement> (um dicionário one-to-many) com base em uma função de seletor de chave.	Não aplicável.	Enumerable.ToLookup

O exemplo de código a seguir usa a cláusula `group by` para agrupar números inteiros em uma lista de acordo com o fato de serem pares ou ímpares.

C#

```
List<int> numbers = [35, 44, 200, 84, 3987, 4, 199, 329, 446, 208];

IEnumerable<IGrouping<int, int>> query = from number in numbers
                                         group number by number % 2;

foreach (var group in query)
{
    Console.WriteLine(group.Key == 0 ? "\nEven numbers:" : "\nOdd
numbers:");
    foreach (int i in group)
    {
        Console.WriteLine(i);
    }
}
```

A consulta equivalente usando a sintaxe do método é mostrada no código a seguir:

C#

```
List<int> numbers = [35, 44, 200, 84, 3987, 4, 199, 329, 446, 208];

IEnumerable<IGrouping<int, int>> query = numbers
    .GroupBy(number => number % 2);

foreach (var group in query)
{
    Console.WriteLine(group.Key == 0 ? "\nEven numbers:" : "\nOdd
numbers:");
    foreach (int i in group)
    {
        Console.WriteLine(i);
    }
}
```

Os exemplos a seguir neste artigo usam fontes de dados comuns para esta área:

```C#
public enum GradeLevel
{
    FirstYear = 1,
    SecondYear,
    ThirdYear,
    FourthYear
};

public class Student
{
    public required string FirstName { get; init; }
    public required string LastName { get; init; }
    public required int ID { get; init; }

    public required GradeLevel Year { get; init; }
    public required List<int> Scores { get; init; }

    public required int DepartmentID { get; init; }
}

public class Teacher
{
    public required string First { get; init; }
    public required string Last { get; init; }
    public required int ID { get; init; }
    public required string City { get; init; }
}
public class Department
{
    public required string Name { get; init; }
    public int ID { get; init; }

    public required int TeacherID { get; init; }
}
```

Cada `Student` tem um nível de escolaridade, um departamento primário e uma série de pontuações. Um `Teacher` também tem uma propriedade `City` que identifica o campus onde o docente ministra aulas. A `Department` tem um nome e uma referência a um `Teacher` que atua como chefe do departamento.

Agrupar resultados de consultas

O agrupamento é um dos recursos mais poderosos do LINQ. Os exemplos a seguir mostram como agrupar dados de várias maneiras:

- Por uma única propriedade.
- Pela primeira letra de uma propriedade de cadeia de caracteres.

- Por um intervalo numérico calculado.
- Por predicado booliano ou outra expressão.
- Por uma chave composta.

Além disso, as duas últimas consultas projetam seus resultados em um novo tipo anônimo que contém somente o primeiro nome e sobrenome do estudante. Para obter mais informações, consulte a cláusula group.

Exemplo de agrupamento por propriedade única

O exemplo a seguir mostra como agrupar elementos de origem usando uma única propriedade do elemento como a chave de grupo. A chave é um `enum`, que é o ano do estudante na escola. A operação de agrupamento usa o comparador de igualdade padrão para o tipo.

C#

```
var groupByYearQuery =
    from student in students
    group student by student.Year into newGroup
    orderby newGroup.Key
    select newGroup;

foreach (var yearGroup in groupByYearQuery)
{
    Console.WriteLine($"Key: {yearGroup.Key}");
    foreach (var student in yearGroup)
    {
        Console.WriteLine($"\t{student.LastName}, {student.FirstName}");
    }
}
```

O código equivalente usando a sintaxe do método é mostrada no exemplo a seguir:

C#

```
// Variable groupByLastNamesQuery is an IEnumerable<IGrouping<string,
// DataClass.Student>>.
var groupByYearQuery = students
    .GroupBy(student => student.Year)
    .OrderBy(newGroup => newGroup.Key);

foreach (var yearGroup in groupByYearQuery)
{
    Console.WriteLine($"Key: {yearGroup.Key}");
    foreach (var student in yearGroup)
    {
        Console.WriteLine($"\t{student.LastName}, {student.FirstName}");
```

```
    }
}
```

Exemplo de agrupamento por valor

O exemplo a seguir mostra como agrupar elementos de origem usando algo diferente de uma propriedade do objeto para a chave de grupo. Neste exemplo, a chave é a primeira letra do sobrenome do estudante.

C#

```
var groupByFirstLetterQuery =
    from student in students
    let firstLetter = student.LastName[0]
    group student by firstLetter;

foreach (var studentGroup in groupByFirstLetterQuery)
{
    Console.WriteLine($"Key: {studentGroup.Key}");
    foreach (var student in studentGroup)
    {
        Console.WriteLine($"\t{student.LastName}, {student.FirstName}");
    }
}
```

O foreach aninhado é necessário para acessar itens de grupo.

O código equivalente usando a sintaxe do método é mostrada no exemplo a seguir:

C#

```
var groupByFirstLetterQuery = students
    .GroupBy(student => student.LastName[0]);

foreach (var studentGroup in groupByFirstLetterQuery)
{
    Console.WriteLine($"Key: {studentGroup.Key}");
    foreach (var student in studentGroup)
    {
        Console.WriteLine($"\t{student.LastName}, {student.FirstName}");
    }
}
```

Exemplo de agrupamento por intervalo

O exemplo a seguir mostra como agrupar elementos de origem usando um intervalo numérico como a chave de grupo. Em seguida, a consulta, projeta os resultados em um

tipo anônimo que contém apenas o nome e o sobrenome e o intervalo de percentil ao qual o estudante pertence. Um tipo anônimo é usado porque não é necessário usar o objeto `Student` completo para exibir os resultados. `GetPercentile` é uma função auxiliar que calcula um percentil com base na pontuação média do aluno. O método retorna um número inteiro entre 0 e 10.

C#

```
static int GetPercentile(Student s)
{
    double avg = s.Scores.Average();
    return avg > 0 ? (int)avg / 10 : 0;
}

var groupByPercentileQuery =
    from student in students
    let percentile = GetPercentile(student)
    group new
    {
        student.FirstName,
        student.LastName
    } by percentile into percentGroup
    orderby percentGroup.Key
    select percentGroup;

foreach (var studentGroup in groupByPercentileQuery)
{
    Console.WriteLine($"Key: {studentGroup.Key * 10}");
    foreach (var item in studentGroup)
    {
        Console.WriteLine($"\t{item.LastName}, {item.FirstName}");
    }
}
```

O foreach aninhado é necessário para iterar em grupos e itens de grupo. O código equivalente usando a sintaxe do método é mostrada no exemplo a seguir:

C#

```
static int GetPercentile(Student s)
{
    double avg = s.Scores.Average();
    return avg > 0 ? (int)avg / 10 : 0;
}

var groupByPercentileQuery = students
    .Select(student => new { student, percentile = GetPercentile(student) })
    .GroupBy(student => student.percentile)
    .Select(percentGroup => new
    {
        percentGroup.Key,
```

```
        Students = percentGroup.Select(s => new { s.student.FirstName,
s.student.LastName })
    })
    .OrderBy(percentGroup => percentGroup.Key);

foreach (var studentGroup in groupByPercentileQuery)
{
    Console.WriteLine($"Key: {studentGroup.Key * 10}");
    foreach (var item in studentGroup.Students)
    {
        Console.WriteLine($"\t{item.LastName}, {item.FirstName}");
    }
}
```

Exemplo de agrupamento por comparação

O exemplo a seguir mostra como agrupar elementos de origem usando uma expressão de comparação booliana. Neste exemplo, a expressão booliana testa se a pontuação média de provas do aluno é maior que 75. Como nos exemplos anteriores, os resultados são projetados em um tipo anônimo porque o elemento de origem completo não é necessário. As propriedades no tipo anônimo tornam-se propriedades no membro `Key`.

C#

```
var groupByHighAverageQuery =
    from student in students
    group new
    {
        student.FirstName,
        student.LastName
    } by student.Scores.Average() > 75 into studentGroup
    select studentGroup;

foreach (var studentGroup in groupByHighAverageQuery)
{
    Console.WriteLine($"Key: {studentGroup.Key}");
    foreach (var student in studentGroup)
    {
        Console.WriteLine($"\t{student.FirstName} {student.LastName}");
    }
}
```

A consulta equivalente usando a sintaxe do método é mostrada no código a seguir:

C#

```
var groupByHighAverageQuery = students
    .GroupBy(student => student.Scores.Average() > 75)
    .Select(group => new
```

```
    {
        group.Key,
        Students = group.AsEnumerable().Select(s => new { s.FirstName,
s.LastName })
    });

foreach (var studentGroup in groupByHighAverageQuery)
{
    Console.WriteLine($"Key: {studentGroup.Key}");
    foreach (var student in studentGroup.Students)
    {
        Console.WriteLine($"\t{student.FirstName} {student.LastName}");
    }
}
```

Exemplo de agrupamento por tipo anônimo

O exemplo a seguir mostra como usar um tipo anônimo para encapsular uma chave que contém vários valores. Neste exemplo, o primeiro valor da chave é a primeira letra do sobrenome do estudante. O segundo valor da chave é um booliano que especifica se o aluno tirou mais que 85 na primeira prova. Você pode ordenar os grupos por qualquer propriedade na chave.

C#

```
var groupByCompoundKey =
    from student in students
    group student by new
    {
        FirstLetterOfLastName = student.LastName[0],
        IsScoreOver85 = student.Scores[0] > 85
    } into studentGroup
    orderby studentGroup.Key.FirstLetterOfLastName
    select studentGroup;

foreach (var scoreGroup in groupByCompoundKey)
{
    var s = scoreGroup.Key.IsScoreOver85 ? "more than 85" : "less than 85";
    Console.WriteLine($"Name starts with
{scoreGroup.Key.FirstLetterOfLastName} who scored {s}");
    foreach (var item in scoreGroup)
    {
        Console.WriteLine($"\t{item.FirstName} {item.LastName}");
    }
}
```

A consulta equivalente usando a sintaxe do método é mostrada no código a seguir:

C#

```
var groupByCompoundKey = students
    .GroupBy(student => new
    {
        FirstLetterOfLastName = student.LastName[0],
        IsScoreOver85 = student.Scores[0] > 85
    })
    .OrderBy(studentGroup => studentGroup.Key.FirstLetterOfLastName);

foreach (var scoreGroup in groupByCompoundKey)
{
    var s = scoreGroup.Key.IsScoreOver85 ? "more than 85" : "less than 85";
    Console.WriteLine($"Name starts with
{scoreGroup.Key.FirstLetterOfLastName} who scored {s}");
    foreach (var item in scoreGroup)
    {
        Console.WriteLine($"\t{item.FirstName} {item.LastName}");
    }
}
```

Criar um grupo aninhado

O exemplo a seguir mostra como criar grupos aninhados em uma expressão de consulta LINQ. Cada grupo que é criado de acordo com o ano do aluno ou nível de ensino, é subdividido em grupos com base nos nomes das pessoas.

C#

```
var nestedGroupsQuery =
    from student in students
    group student by student.Year into newGroup1
    from newGroup2 in
    from student in newGroup1
    group student by student.LastName
    group newGroup2 by newGroup1.Key;

foreach (var outerGroup in nestedGroupsQuery)
{
    Console.WriteLine($"DataClass.Student Level = {outerGroup.Key}");
    foreach (var innerGroup in outerGroup)
    {
        Console.WriteLine($"\tNames that begin with: {innerGroup.Key}");
        foreach (var innerGroupElement in innerGroup)
        {
            Console.WriteLine($"\t\t{innerGroupElement.LastName}
{innerGroupElement.FirstName}");
        }
    }
}
```

Os três loops `foreach` aninhados são necessários para iterar sobre os elementos internos de um grupo aninhado.
(Passe o cursor do mouse sobre as variáveis de iteração `outerGroup`, `innerGroup` e `innerGroupElement` para ver o tipo real delas).

A consulta equivalente usando a sintaxe do método é mostrada no código a seguir:

C#

```
var nestedGroupsQuery =
    students
    .GroupBy(student => student.Year)
    .Select(newGroup1 => new
    {
        newGroup1.Key,
        NestedGroup = newGroup1
            .GroupBy(student => student.LastName)
    });

foreach (var outerGroup in nestedGroupsQuery)
{
    Console.WriteLine($"DataClass.Student Level = {outerGroup.Key}");
    foreach (var innerGroup in outerGroup.NestedGroup)
    {
        Console.WriteLine($"\tNames that begin with: {innerGroup.Key}");
        foreach (var innerGroupElement in innerGroup)
        {
            Console.WriteLine($"\t\t{innerGroupElement.LastName}
{innerGroupElement.FirstName}");
        }
    }
}
```

Executar uma subconsulta em uma operação de agrupamento

Este artigo mostra duas maneiras diferentes de criar uma consulta que ordena os dados de origem em grupos e, em seguida, realiza uma subconsulta em cada grupo individualmente. A técnica básica em cada exemplo é agrupar os elementos de origem usando uma *continuação* chamada `newGroup` e, em seguida, gerar uma nova subconsulta de `newGroup`. Essa subconsulta é executada em cada novo grupo criado pela consulta externa. Nesse exemplo específico, a saída final não é um grupo, mas uma sequência simples de tipos anônimos.

Para obter mais informações sobre como agrupar, consulte Cláusula group. Para obter mais informações sobre continuações, consulte into. O exemplo a seguir usa uma

estrutura de dados na memória como a fonte de dados, mas os mesmos princípios se aplicam a qualquer tipo de fonte de dados do LINQ.

C#

```
var queryGroupMax =
    from student in students
    group student by student.Year into studentGroup
    select new
    {
        Level = studentGroup.Key,
        HighestScore = (
            from student2 in studentGroup
            select student2.Scores.Average()
        ).Max()
    };

var count = queryGroupMax.Count();
Console.WriteLine($"Number of groups = {count}");

foreach (var item in queryGroupMax)
{
    Console.WriteLine($"  {item.Level} Highest Score={item.HighestScore}");
}
```

A consulta no trecho anterior acima também pode ser escrita usando a sintaxe do método. O snippet de código a seguir tem uma consulta semanticamente equivalente escrita usando a sintaxe de método.

C#

```
var queryGroupMax =
    students
        .GroupBy(student => student.Year)
        .Select(studentGroup => new
        {
            Level = studentGroup.Key,
            HighestScore = studentGroup.Max(student2 =>
student2.Scores.Average())
        });

var count = queryGroupMax.Count();
Console.WriteLine($"Number of groups = {count}");

foreach (var item in queryGroupMax)
{
    Console.WriteLine($"  {item.Level} Highest Score={item.HighestScore}");
}
```

Confira também

- System.Linq
- GroupBy
- IGrouping<TKey,TElement>
- Cláusula group
- Como dividir um arquivo em vários arquivos usando grupos (LINQ) (C#)

Colaborar conosco no GitHub

A fonte deste conteúdo pode ser encontrada no GitHub, onde você também pode criar e revisar problemas e solicitações de pull. Para obter mais informações, confira o nosso guia para colaboradores.

.NET

Comentários do .NET

O .NET é um projeto código aberto. Selecione um link para fornecer comentários:

Abrir um problema de documentação

Fornecer comentários sobre o produto

Como usar LINQ para consultar arquivos e diretórios

07/06/2025

Muitas operações do sistema de arquivos são essencialmente consultas e, portanto, são adequadas para a abordagem LINQ. Essas consultas não são destrutivas. Eles não alteram o conteúdo dos arquivos ou pastas originais. As consultas não devem causar consequências indesejadas. Em geral, qualquer código (incluindo consultas que executam operações de criação/atualização/exclusão) que modifica os dados de origem deve ser mantido separado do código que apenas consulta os dados.

Há alguma complexidade envolvida na criação de uma fonte de dados que representa com precisão o conteúdo do sistema de arquivos e lida com exceções normalmente. Os exemplos nesta seção criam uma coleção de instantâneos de FileInfo objetos que representa todos os arquivos em uma pasta raiz especificada e todas as suas subpastas. O estado real de cada um FileInfo pode mudar no tempo entre quando você começa e termina a execução de uma consulta. Por exemplo, você pode criar uma lista de FileInfo objetos a serem usados como uma fonte de dados. Se você tentar acessar a `Length` propriedade em uma consulta, o FileInfo objeto tentará acessar o sistema de arquivos para atualizar o valor de `Length`. Se o arquivo não existir mais, você obterá um FileNotFoundException na consulta, mesmo que não esteja consultando o sistema de arquivos diretamente.

Como consultar arquivos com um atributo ou nome especificado

Este exemplo mostra como localizar todos os arquivos que têm uma extensão de nome de arquivo especificada (por exemplo, ".txt") em uma árvore de diretório especificada. Ele também mostra como retornar o arquivo mais recente ou mais antigo na árvore com base no tempo de criação. Talvez seja necessário modificar a primeira linha de muitos dos exemplos, independentemente de você estar executando esse código no Windows, Mac ou em um sistema Linux.

C#

```
string startFolder = """C:\Program Files\dotnet\sdk""";
// Or
// string startFolder = "/usr/local/share/dotnet/sdk";

DirectoryInfo dir = new DirectoryInfo(startFolder);
var fileList = dir.GetFiles("*.*", SearchOption.AllDirectories);
```

```
var fileQuery = from file in fileList
                where file.Extension == ".txt"
                orderby file.Name
                select file;

// Uncomment this block to see the full query
// foreach (FileInfo fi in fileQuery)
// {
//     Console.WriteLine(fi.FullName);
// }

var newestFile = (from file in fileQuery
                  orderby file.CreationTime
                  select new { file.FullName, file.CreationTime })
                  .Last();

Console.WriteLine($"\r\nThe newest .txt file is {newestFile.FullName}. Creation
time: {newestFile.CreationTime}");
```

Como agrupar arquivos por extensão

Este exemplo mostra como o LINQ pode ser usado para executar operações avançadas de agrupamento e classificação em listas de arquivos ou pastas. Ele também mostra como paginar a saída na janela do console usando os métodos Skip e Take.

A consulta a seguir mostra como agrupar o conteúdo de uma árvore de diretório especificada pela extensão de nome de arquivo.

C#

```
string startFolder = """C:\Program Files\dotnet\sdk""";
// Or
// string startFolder = "/usr/local/share/dotnet/sdk";

int trimLength = startFolder.Length;

DirectoryInfo dir = new DirectoryInfo(startFolder);

var fileList = dir.GetFiles("*.*", SearchOption.AllDirectories);

var queryGroupByExt = from file in fileList
                      group file by file.Extension.ToLower() into fileGroup
                      orderby fileGroup.Count(), fileGroup.Key
                      select fileGroup;

// Iterate through the outer collection of groups.
foreach (var filegroup in queryGroupByExt.Take(5))
{
    Console.WriteLine($"Extension: {filegroup.Key}");
    var resultPage = filegroup.Take(20);
```

```
    //Execute the resultPage query
    foreach (var f in resultPage)
    {
        Console.WriteLine($"\t{f.FullName.Substring(trimLength)}");
    }
    Console.WriteLine();
}
```

A saída desse programa pode ser longa, dependendo dos detalhes do sistema de arquivos local e do valor para o qual o `startFolder` está ajustado. Para habilitar a exibição de todos os resultados, este exemplo mostra como percorrer os resultados. Um loop aninhado `foreach` é necessário porque cada grupo é enumerado separadamente.

Como consultar o número total de bytes em um conjunto de pastas

Este exemplo mostra como recuperar o número total de bytes usados por todos os arquivos em uma pasta especificada e todas as suas subpastas. O Sum método adiciona os valores de todos os itens selecionados na `select` cláusula. Você pode modificar essa consulta para recuperar o maior ou menor arquivo na árvore de diretórios especificada, chamando o método Min ou o método Max em vez do Sum.

C#

```
string startFolder = """C:\Program Files\dotnet\sdk""";
// Or
// string startFolder = "/usr/local/share/dotnet/sdk";

var fileList = Directory.GetFiles(startFolder, "*.*",
SearchOption.AllDirectories);

var fileQuery = from file in fileList
                let fileLen = new FileInfo(file).Length
                where fileLen > 0
                select fileLen;

// Cache the results to avoid multiple trips to the file system.
long[] fileLengths = fileQuery.ToArray();

// Return the size of the largest file
long largestFile = fileLengths.Max();

// Return the total number of bytes in all the files under the specified folder.
long totalBytes = fileLengths.Sum();

Console.WriteLine($"There are {totalBytes} bytes in {fileList.Count()} files under
```

```
{startFolder}");
Console.WriteLine($"The largest file is {largestFile} bytes.");
```

Este exemplo estende o exemplo anterior para fazer o seguinte:

- Como recuperar o tamanho em bytes do maior arquivo.
- Como recuperar o tamanho em bytes do arquivo menor.
- Como recuperar o FileInfo arquivo maior ou menor do objeto de uma ou mais pastas em uma pasta raiz especificada.
- Como recuperar uma sequência, como os 10 maiores arquivos.
- Como ordenar arquivos em grupos com base no tamanho do arquivo em bytes, ignorando arquivos que são menores que um tamanho especificado.

O exemplo a seguir contém cinco consultas separadas que mostram como consultar e agrupar arquivos, dependendo do tamanho do arquivo em bytes. Você pode modificar esses exemplos para basear a consulta em alguma outra propriedade do FileInfo objeto.

C#

```
// Return the FileInfo object for the largest file
// by sorting and selecting from beginning of list
FileInfo longestFile = (from file in fileList
                        let fileInfo = new FileInfo(file)
                        where fileInfo.Length > 0
                        orderby fileInfo.Length descending
                        select fileInfo
                        ).First();

Console.WriteLine($"The largest file under {startFolder} is {longestFile.FullName}
with a length of {longestFile.Length} bytes");

//Return the FileInfo of the smallest file
FileInfo smallestFile = (from file in fileList
                         let fileInfo = new FileInfo(file)
                         where fileInfo.Length > 0
                         orderby fileInfo.Length ascending
                         select fileInfo
                        ).First();

Console.WriteLine($"The smallest file under {startFolder} is
{smallestFile.FullName} with a length of {smallestFile.Length} bytes");

//Return the FileInfos for the 10 largest files
var queryTenLargest = (from file in fileList
                       let fileInfo = new FileInfo(file)
                       let len = fileInfo.Length
                       orderby len descending
                       select fileInfo
                      ).Take(10);
```

```
Console.WriteLine($"The 10 largest files under {startFolder} are:");

foreach (var v in queryTenLargest)
{
    Console.WriteLine($"{v.FullName}: {v.Length} bytes");
}

// Group the files according to their size, leaving out
// files that are less than 200000 bytes.
var querySizeGroups = from file in fileList
                      let fileInfo = new FileInfo(file)
                      let len = fileInfo.Length
                      where len > 0
                      group fileInfo by (len / 100000) into fileGroup
                      where fileGroup.Key >= 2
                      orderby fileGroup.Key descending
                      select fileGroup;

foreach (var filegroup in querySizeGroups)
{
    Console.WriteLine($"{filegroup.Key}00000");
    foreach (var item in filegroup)
    {
        Console.WriteLine($"\t{item.Name}: {item.Length}");
    }
}
```

Para retornar um ou mais objetos completos FileInfo , a consulta primeiro deve examinar cada um deles na fonte de dados e, em seguida, classificá-los pelo valor de sua propriedade Length. Em seguida, ele pode retornar o único elemento ou a sequência com o maior comprimento. Use First para retornar o primeiro elemento em uma lista. Use Take para retornar o primeiro n número de elementos. Especifique uma ordem de classificação decrescente para colocar os menores elementos no início da lista.

Como consultar arquivos duplicados em uma árvore de diretório

Às vezes, arquivos com o mesmo nome podem ser localizados em mais de uma pasta. Este exemplo mostra como consultar esses nomes de arquivo duplicados em uma pasta raiz especificada. O segundo exemplo mostra como consultar arquivos cujo tamanho e tempos de LastWrite também correspondem.

C#

```
string startFolder = """C:\Program Files\dotnet\sdk""";
// Or
// string startFolder = "/usr/local/share/dotnet/sdk";
```

```
DirectoryInfo dir = new DirectoryInfo(startFolder);

IEnumerable<FileInfo> fileList = dir.GetFiles("*.*", SearchOption.AllDirectories);

// used in WriteLine to keep the lines shorter
int charsToSkip = startFolder.Length;

// var can be used for convenience with groups.
var queryDupNames = from file in fileList
                    group file.FullName.Substring(charsToSkip) by file.Name into
fileGroup
                    where fileGroup.Count() > 1
                    select fileGroup;

foreach (var queryDup in queryDupNames.Take(20))
{
    Console.WriteLine($"Filename = {(queryDup.Key.ToString() == string.Empty ? "
[none]" : queryDup.Key.ToString())}");

    foreach (var fileName in queryDup.Take(10))
    {
        Console.WriteLine($"\t{fileName}");
    }
}
```

A primeira consulta usa uma chave para determinar uma correspondência. Ele localiza arquivos que têm o mesmo nome, mas cujo conteúdo pode ser diferente. A segunda consulta usa uma chave composta para corresponder a três propriedades do FileInfo objeto. Essa consulta é muito mais provável de encontrar arquivos com o mesmo nome e conteúdo semelhante ou idêntico.

C#

```
    string startFolder = """C:\Program Files\dotnet\sdk""";
    // Or
    // string startFolder = "/usr/local/share/dotnet/sdk";

    // Make the lines shorter for the console display
    int charsToSkip = startFolder.Length;

    // Take a snapshot of the file system.
    DirectoryInfo dir = new DirectoryInfo(startFolder);
    IEnumerable<FileInfo> fileList = dir.GetFiles("*.*",
SearchOption.AllDirectories);

    // Note the use of a compound key. Files that match
    // all three properties belong to the same group.
    // A named type is used to enable the query to be
    // passed to another method. Anonymous types can also be used
    // for composite keys but cannot be passed across method boundaries
    //
    var queryDupFiles = from file in fileList
```

```
                        group file.FullName.Substring(charsToSkip) by
                        (Name: file.Name, LastWriteTime: file.LastWriteTime,
Length: file.Length )
                        into fileGroup
                        where fileGroup.Count() > 1
                        select fileGroup;

    foreach (var queryDup in queryDupFiles.Take(20))
    {
        Console.WriteLine($"Filename = {(queryDup.Key.ToString() == string.Empty ?
"[none]" : queryDup.Key.ToString())}");

        foreach (var fileName in queryDup)
        {
            Console.WriteLine($"\t{fileName}");
        }
    }
}
```

Como consultar o conteúdo de arquivos de texto em uma pasta

Este exemplo mostra como consultar todos os arquivos em uma árvore de diretório especificada, abrir cada arquivo e inspecionar seu conteúdo. Esse tipo de técnica pode ser usado para criar índices ou índices reversos do conteúdo de uma árvore de diretório. Uma pesquisa de cadeia de caracteres simples é executada neste exemplo. No entanto, tipos mais complexos de correspondência de padrões podem ser executados com uma expressão regular.

C#

```
string startFolder = """C:\Program Files\dotnet\sdk""";
// Or
// string startFolder = "/usr/local/share/dotnet/sdk";

DirectoryInfo dir = new DirectoryInfo(startFolder);

var fileList = dir.GetFiles("*.*", SearchOption.AllDirectories);

string searchTerm = "change";

var queryMatchingFiles = from file in fileList
                         where file.Extension == ".txt"
                         let fileText = File.ReadAllText(file.FullName)
                         where fileText.Contains(searchTerm)
                         select file.FullName;

// Execute the query.
Console.WriteLine($"""The term "{searchTerm}" was found in:""");
foreach (string filename in queryMatchingFiles)
```

```
{
    Console.WriteLine(filename);
}
```

Como comparar o conteúdo de duas pastas

Este exemplo demonstra três maneiras de comparar duas listagens de arquivos:

- Consultando um valor booliano que especifica se as duas listas de arquivos são idênticas.
- Consultando a interseção para recuperar os arquivos que estão em ambas as pastas.
- Ao fazer uma consulta para a diferença entre conjuntos, é possível recuperar os arquivos que estão em uma pasta, mas não na outra.

As técnicas mostradas aqui podem ser adaptadas para comparar sequências de objetos de qualquer tipo.

A `FileComparer` classe mostrada aqui demonstra como usar uma classe de comparador personalizada junto com os Operadores de Consulta Padrão. A classe não se destina a ser usada em cenários reais. Ele usa apenas o nome e o comprimento em bytes de cada arquivo para determinar se o conteúdo de cada pasta é idêntico ou não. Em um cenário do mundo real, você deve modificar esse comparador para executar uma verificação de igualdade mais rigorosa.

C#

```
// This implementation defines a very simple comparison
// between two FileInfo objects. It only compares the name
// of the files being compared and their length in bytes.
class FileCompare : IEqualityComparer<FileInfo>
{
    public bool Equals(FileInfo? f1, FileInfo? f2)
    {
        return (f1?.Name == f2?.Name &&
                f1?.Length == f2?.Length);
    }

    // Return a hash that reflects the comparison criteria. According to the
    // rules for IEqualityComparer<T>, if Equals is true, then the hash codes must
    // also be equal. Because equality as defined here is a simple value equality,
not
    // reference identity, it is possible that two or more objects will produce
the same
    // hash code.
    public int GetHashCode(FileInfo fi)
    {
        string s = $"{fi.Name}{fi.Length}";
        return s.GetHashCode();
    }
```

```
}

public static void CompareDirectories()
{
    string pathA = """C:\Program Files\dotnet\sdk\8.0.104""";
    string pathB = """C:\Program Files\dotnet\sdk\8.0.204""";

    DirectoryInfo dir1 = new DirectoryInfo(pathA);
    DirectoryInfo dir2 = new DirectoryInfo(pathB);

    IEnumerable<FileInfo> list1 = dir1.GetFiles("*.*",
SearchOption.AllDirectories);
    IEnumerable<FileInfo> list2 = dir2.GetFiles("*.*",
SearchOption.AllDirectories);

    //A custom file comparer defined below
    FileCompare myFileCompare = new FileCompare();

    // This query determines whether the two folders contain
    // identical file lists, based on the custom file comparer
    // that is defined in the FileCompare class.
    // The query executes immediately because it returns a bool.
    bool areIdentical = list1.SequenceEqual(list2, myFileCompare);

    if (areIdentical == true)
    {
        Console.WriteLine("the two folders are the same");
    }
    else
    {
        Console.WriteLine("The two folders are not the same");
    }

    // Find the common files. It produces a sequence and doesn't
    // execute until the foreach statement.
    var queryCommonFiles = list1.Intersect(list2, myFileCompare);

    if (queryCommonFiles.Any())
    {
        Console.WriteLine($"The following files are in both folders (total number
= {queryCommonFiles.Count()}):");
        foreach (var v in queryCommonFiles.Take(10))
        {
            Console.WriteLine(v.Name); //shows which items end up in result list
        }
    }
    else
    {
        Console.WriteLine("There are no common files in the two folders.");
    }

    // Find the set difference between the two folders.
    var queryList1Only = (from file in list1
                          select file)
                          .Except(list2, myFileCompare);
```

```
    Console.WriteLine();
    Console.WriteLine($"The following files are in list1 but not list2 (total
number = {queryList1Only.Count()}):");
    foreach (var v in queryList1Only.Take(10))
    {
        Console.WriteLine(v.FullName);
    }

    var queryList2Only = (from file in list2
                          select file)
                          .Except(list1, myFileCompare);

    Console.WriteLine();
    Console.WriteLine($"The following files are in list2 but not list1 (total
number = {queryList2Only.Count()}:");
    foreach (var v in queryList2Only.Take(10))
    {
        Console.WriteLine(v.FullName);
    }
}
```

Como reordenar os campos de um arquivo delimitado

Um arquivo CSV (valor separado por vírgulas) é um arquivo de texto que geralmente é usado para armazenar dados de planilha ou outros dados tabulares representados por linhas e colunas. Usando o Split método para separar os campos, é fácil consultar e manipular arquivos CSV usando LINQ. Na verdade, a mesma técnica pode ser usada para reordenar as partes de qualquer linha estruturada de texto; não se limita a arquivos CSV.

No exemplo a seguir, suponha que as três colunas representem o "nome da família", o "nome" e a "ID" dos alunos. Os campos estão em ordem alfabética com base nos nomes da família dos alunos. A consulta produz uma nova sequência na qual a coluna de ID aparece primeiro, seguida por uma segunda coluna que combina o nome do aluno e o nome da família. As linhas são reordenadas de acordo com o campo ID. Os resultados são salvos em um novo arquivo e os dados originais não são modificados. O texto a seguir mostra o conteúdo do arquivo *spreadsheet1.csv* usado no exemplo a seguir:

```txt
Adams,Terry,120
Fakhouri,Fadi,116
Feng,Hanying,117
Garcia,Cesar,114
Garcia,Debra,115
Garcia,Hugo,118
```

```
Mortensen,Sven,113
O'Donnell,Claire,112
Omelchenko,Svetlana,111
Tucker,Lance,119
Tucker,Michael,122
Zabokritski,Eugene,121
```

O código a seguir lê o arquivo de origem e reorganiza cada coluna no arquivo CSV para reorganizar a ordem das colunas:

C#

```
string[] lines = File.ReadAllLines("spreadsheet1.csv");

// Create the query. Put field 2 first, then
// reverse and combine fields 0 and 1 from the old field
IEnumerable<string> query = from line in lines
                            let fields = line.Split(',')
                            orderby fields[2]
                            select $"{fields[2]}, {fields[1]} {fields[0]}";

File.WriteAllLines("spreadsheet2.csv", query.ToArray());

/* Output to spreadsheet2.csv:
111, Svetlana Omelchenko
112, Claire O'Donnell
113, Sven Mortensen
114, Cesar Garcia
115, Debra Garcia
116, Fadi Fakhouri
117, Hanying Feng
118, Hugo Garcia
119, Lance Tucker
120, Terry Adams
121, Eugene Zabokritski
122, Michael Tucker
*/
```

Como dividir um arquivo em muitos arquivos usando grupos

Este exemplo mostra uma maneira de mesclar o conteúdo de dois arquivos e, em seguida, criar um conjunto de novos arquivos que organizam os dados de uma nova maneira. A consulta usa o conteúdo de dois arquivos. O texto a seguir mostra o conteúdo do primeiro arquivo, *names1.txt*:

txt

```
Bankov, Peter
Holm, Michael
Garcia, Hugo
Potra, Cristina
Noriega, Fabricio
Aw, Kam Foo
Beebe, Ann
Toyoshima, Tim
Guy, Wey Yuan
Garcia, Debra
```

O segundo arquivo, *names2.txt*, contém um conjunto diferente de nomes, alguns dos quais estão em comum com o primeiro conjunto:

```txt
Liu, Jinghao
Bankov, Peter
Holm, Michael
Garcia, Hugo
Beebe, Ann
Gilchrist, Beth
Myrcha, Jacek
Giakoumakis, Leo
McLin, Nkenge
El Yassir, Mehdi
```

O código a seguir consulta ambos os arquivos, usa a união de ambos os arquivos e grava um novo arquivo para cada grupo, definido pela primeira letra do nome da família:

```C#
string[] fileA = File.ReadAllLines("names1.txt");
string[] fileB = File.ReadAllLines("names2.txt");

// Concatenate and remove duplicate names
var mergeQuery = fileA.Union(fileB);

// Group the names by the first letter in the last name.
var groupQuery = from name in mergeQuery
                 let n = name.Split(',')[0]
                 group name by n[0] into g
                 orderby g.Key
                 select g;

foreach (var g in groupQuery)
{
    string fileName = $"testFile_{g.Key}.txt";

    Console.WriteLine(g.Key);
```

```
    using StreamWriter sw = new StreamWriter(fileName);
    foreach (var item in g)
    {
        sw.WriteLine(item);
        // Output to console for example purposes.
        Console.WriteLine($"   {item}");
    }
}
/* Output:
    A
       Aw, Kam Foo
    B
       Bankov, Peter
       Beebe, Ann
    E
       El Yassir, Mehdi
    G
       Garcia, Hugo
       Guy, Wey Yuan
       Garcia, Debra
       Gilchrist, Beth
       Giakoumakis, Leo
    H
       Holm, Michael
    L
       Liu, Jinghao
    M
       Myrcha, Jacek
       McLin, Nkenge
    N
       Noriega, Fabricio
    P
       Potra, Cristina
    T
       Toyoshima, Tim
 */
```

Como unir conteúdo de arquivos diferentes

Este exemplo mostra como unir dados de dois arquivos delimitados por vírgulas que compartilham um valor comum que é usado como uma chave correspondente. Essa técnica pode ser útil se você precisar combinar dados de duas planilhas ou de uma planilha e de um arquivo que tenha outro formato em um novo arquivo. Você pode modificar o exemplo para trabalhar com qualquer tipo de texto estruturado.

O texto a seguir mostra o conteúdo de *scores.csv*. O arquivo representa dados de planilha. A coluna 1 é a ID do aluno e as colunas 2 a 5 são pontuações de teste.

```
txt
```

```
111, 97, 92, 81, 60
112, 75, 84, 91, 39
113, 88, 94, 65, 91
114, 97, 89, 85, 82
115, 35, 72, 91, 70
116, 99, 86, 90, 94
117, 93, 92, 80, 87
118, 92, 90, 83, 78
119, 68, 79, 88, 92
120, 99, 82, 81, 79
121, 96, 85, 91, 60
122, 94, 92, 91, 91
```

O texto a seguir mostra o conteúdo de *names.csv*. O arquivo representa uma planilha que contém o nome da família do aluno, o nome e a ID do aluno.

txt

```
Omelchenko,Svetlana,111
O'Donnell,Claire,112
Mortensen,Sven,113
Garcia,Cesar,114
Garcia,Debra,115
Fakhouri,Fadi,116
Feng,Hanying,117
Garcia,Hugo,118
Tucker,Lance,119
Adams,Terry,120
Zabokritski,Eugene,121
Tucker,Michael,122
```

Junte o conteúdo de arquivos diferentes que contêm informações relacionadas. O *names.csv* de arquivo contém o nome do aluno mais um número de ID. O arquivo *scores.csv* contém um ID e um conjunto de quatro resultados de teste. A consulta a seguir une as pontuações aos nomes dos alunos usando a ID como uma chave correspondente. O código é mostrado no exemplo a seguir:

C#

```
string[] names = File.ReadAllLines(@"names.csv");
string[] scores = File.ReadAllLines(@"scores.csv");

var scoreQuery = from name in names
                 let nameFields = name.Split(',')
                 from id in scores
                 let scoreFields = id.Split(',')
                 where Convert.ToInt32(nameFields[2]) ==
Convert.ToInt32(scoreFields[0])
                 select $"{nameFields[0]},{scoreFields[1]},{scoreFields[2]},
```

```
{scoreFields[3]},{scoreFields[4]}";

Console.WriteLine("\r\nMerge two spreadsheets:");
foreach (string item in scoreQuery)
{
    Console.WriteLine(item);
}
Console.WriteLine($"{scoreQuery.Count()} total names in list");
/* 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
 */
```

Como calcular valores de coluna em um arquivo de texto CSV

Este exemplo mostra como executar cálculos agregados como Soma, Média, Min e Máximo nas colunas de um arquivo .csv. Os princípios de exemplo mostrados aqui podem ser aplicados a outros tipos de texto estruturado.

O texto a seguir mostra o conteúdo de *scores.csv*. Suponha que a primeira coluna represente uma ID do aluno e as colunas subsequentes representem pontuações de quatro exames.

```txt
111, 97, 92, 81, 60
112, 75, 84, 91, 39
113, 88, 94, 65, 91
114, 97, 89, 85, 82
115, 35, 72, 91, 70
116, 99, 86, 90, 94
117, 93, 92, 80, 87
118, 92, 90, 83, 78
119, 68, 79, 88, 92
120, 99, 82, 81, 79
```

```
121, 96, 85, 91, 60
122, 94, 92, 91, 91
```

O texto a seguir mostra como usar o Split método para converter cada linha de texto em uma matriz. Cada elemento de matriz representa uma coluna. Por fim, o texto em cada coluna é convertido em sua representação numérica.

C#

```
public static class SumColumns
{
    public static void ProcessColumns(string filePath, string seperator)
    {
        // Divide each exam into a group
        var exams = from line in MatrixFrom(filePath, seperator)
                    from score in line

                    // Identify the column number
                    let colNumber = Array.FindIndex(line, t =>
ReferenceEquals(score, t))

                    // The first column is the student ID, not the exam score
                    // so it needs to be excluded
                    where colNumber > 0

                    // Convert the score from string to int
                    // Group by column number, i.e. one group per exam
                    group double.Parse(score) by colNumber into g
                    select new
                    {
                        Title = $"Exam#{g.Key}",
                        Min = g.Min(),
                        Max = g.Max(),
                        Avg = Math.Round(g.Average(), 2),
                        Total = g.Sum()
                    };

        foreach (var exam in exams)
        {
            Console.WriteLine($"{exam.Title}\t"
            + $"Average:{exam.Avg,6}\t"
            + $"High Score:{exam.Max,3}\t"
            + $"Low Score:{exam.Min,3}\t"
            + $"Total:{exam.Total,5}");
        }
    }

    // Transform the file content to an IEnumerable of string arrays
    // like a matrix
    private static IEnumerable<string[]> MatrixFrom(string filePath, string
seperator)
    {
        using StreamReader reader = File.OpenText(filePath);
```

```
        for (string? line = reader.ReadLine(); line is not null; line =
reader.ReadLine())
        {
            yield return line.Split(seperator, StringSplitOptions.TrimEntries);
        }
    }
}

// Output:
// Exam#1  Average: 86.08  High Score: 99  Low Score: 35   Total: 1033
// Exam#2  Average: 86.42  High Score: 94  Low Score: 72   Total: 1037
// Exam#3  Average: 84.75  High Score: 91  Low Score: 65   Total: 1017
// Exam#4  Average: 76.92  High Score: 94  Low Score: 39   Total:  923
```

Se o arquivo for um arquivo separado por tabulação, basta atualizar o argumento no `SumColumns.ProcessColumns` método para `\t`.

Como usar LINQ para consultar cadeias de caracteres

Artigo • 02/05/2024

As cadeias de caracteres são armazenadas como uma sequência de caracteres. Como uma sequência de caracteres, eles podem ser consultados usando LINQ. Neste artigo, há várias consultas de exemplo que consultam cadeias de caracteres ou palavras diferentes, filtram cadeias de caracteres ou misturam consultas com expressões regulares.

Como consultar para detectar caracteres em uma cadeia de caracteres

O exemplo a seguir consulta uma cadeia de caracteres para determinar quantos dígitos numéricos ela contém.

C#

```
string aString = "ABCDE99F-J74-12-89A";

// Select only those characters that are numbers
var stringQuery = from ch in aString
                  where Char.IsDigit(ch)
                  select ch;

// Execute the query
foreach (char c in stringQuery)
    Console.Write(c + " ");

// Call the Count method on the existing query.
int count = stringQuery.Count();
Console.WriteLine($"Count = {count}");

// Select all characters before the first '-'
var stringQuery2 = aString.TakeWhile(c => c != '-');

// Execute the second query
foreach (char c in stringQuery2)
    Console.Write(c);
/* Output:
  Output: 9 9 7 4 1 2 8 9
  Count = 8
  ABCDE99F
*/
```

A consulta anterior mostra como você pode tratar uma cadeia de caracteres como uma sequência de caracteres.

Como contar ocorrências de uma palavra em uma cadeia de caracteres

Este exemplo a seguir mostra como usar uma consulta LINQ para contar as ocorrências de uma palavra especificada em uma cadeia de caracteres. Para executar a contagem, primeiro o método Split é chamado para criar uma matriz de palavras. Há um custo de desempenho para o método Split. Se for a única operação na cadeia de caracteres for contar as palavras, considere o uso dos métodos Matches ou IndexOf em vez dele.

C#

```
string text = """
    Historically, the world of data and the world of objects
    have not been well integrated. Programmers work in C# or Visual Basic
    and also in SQL or XQuery. On the one side are concepts such as classes,
    objects, fields, inheritance, and .NET APIs. On the other side
    are tables, columns, rows, nodes, and separate languages for dealing
with
    them. Data types often require translation between the two worlds; there
are
    different standard functions. Because the object world has no notion of
query, a
    query can only be represented as a string without compile-time type
checking or
    IntelliSense support in the IDE. Transferring data from SQL tables or
XML trees to
    objects in memory is often tedious and error-prone.
    """;

string searchTerm = "data";

//Convert the string into an array of words
char[] separators = ['.', '?', '!', ' ', ';', ':', ','];
string[] source = text.Split(separators,
StringSplitOptions.RemoveEmptyEntries);

// Create the query.  Use the InvariantCultureIgnoreCase comparison to match
"data" and "Data"
var matchQuery = from word in source
                 where word.Equals(searchTerm,
StringComparison.InvariantCultureIgnoreCase)
                 select word;

// Count the matches, which executes the query.
int wordCount = matchQuery.Count();
Console.WriteLine($"""{wordCount} occurrences(s) of the search term "
```

```
{searchTerm}" were found.""");
/* Output:
   3 occurrences(s) of the search term "data" were found.
*/
```

A consulta anterior mostra como você pode exibir cadeias de caracteres como uma sequência de palavras, depois de dividir uma cadeia de caracteres em uma sequência de palavras.

Como classificar ou filtrar dados de texto por qualquer palavra ou campo

O exemplo a seguir mostra como classificar linhas de texto estruturado, como valores separados por vírgulas, por qualquer campo na linha. O campo pode ser especificado dinamicamente em tempo de execução. Suponha que os campos em *scores.csv* representam o número de ID do aluno, seguido por uma série de quatro resultados de teste:

txt

```
111, 97, 92, 81, 60
112, 75, 84, 91, 39
113, 88, 94, 65, 91
114, 97, 89, 85, 82
115, 35, 72, 91, 70
116, 99, 86, 90, 94
117, 93, 92, 80, 87
118, 92, 90, 83, 78
119, 68, 79, 88, 92
120, 99, 82, 81, 79
121, 96, 85, 91, 60
122, 94, 92, 91, 91
```

A consulta a seguir classifica as linhas com base na pontuação do primeiro exame, armazenada na segunda coluna:

C#

```
// Create an IEnumerable data source
string[] scores = File.ReadAllLines("scores.csv");

// Change this to any value from 0 to 4.
int sortField = 1;

Console.WriteLine($"Sorted highest to lowest by field [{sortField}]:");
```

```
// Split the string and sort on field[num]
var scoreQuery = from line in scores
                 let fields = line.Split(',')
                 orderby fields[sortField] descending
                 select line;

foreach (string str in scoreQuery)
{
    Console.WriteLine(str);
}
/* Output (if sortField == 1):
   Sorted highest to lowest by field [1]:
    116, 99, 86, 90, 94
    120, 99, 82, 81, 79
    111, 97, 92, 81, 60
    114, 97, 89, 85, 82
    121, 96, 85, 91, 60
    122, 94, 92, 91, 91
    117, 93, 92, 80, 87
    118, 92, 90, 83, 78
    113, 88, 94, 65, 91
    112, 75, 84, 91, 39
    119, 68, 79, 88, 92
    115, 35, 72, 91, 70
 */
```

A consulta anterior mostra como você pode manipular cadeias de caracteres dividindo-as em campos e consultando os campos individuais.

Como consultar frases que expressões específicas

O exemplo a seguir mostra como localizar frases em um arquivo de texto que contenham correspondências para cada conjunto de palavras especificado. Embora a matriz de termos de pesquisa seja embutida em código, ela também pode ser preenchida dinamicamente em tempo de execução. A consulta retorna as frases que contêm as palavras “Historically”, “data" e “integrated”.

C#

```
string text = """
Historically, the world of data and the world of objects
have not been well integrated. Programmers work in C# or Visual Basic
and also in SQL or XQuery. On the one side are concepts such as classes,
objects, fields, inheritance, and .NET APIs. On the other side
are tables, columns, rows, nodes, and separate languages for dealing with
them. Data types often require translation between the two worlds; there are
different standard functions. Because the object world has no notion of
query, a
```

```
query can only be represented as a string without compile-time type checking
or
IntelliSense support in the IDE. Transferring data from SQL tables or XML
trees to
objects in memory is often tedious and error-prone.
""";

// Split the text block into an array of sentences.
string[] sentences = text.Split(['.', '?', '!']);

// Define the search terms. This list could also be dynamically populated at
run time.
string[] wordsToMatch = [ "Historically", "data", "integrated" ];

// Find sentences that contain all the terms in the wordsToMatch array.
// Note that the number of terms to match is not specified at compile time.
char[] separators = ['.', '?', '!', ' ', ';', ':', ','];
var sentenceQuery = from sentence in sentences
                    let w =
sentence.Split(separators,StringSplitOptions.RemoveEmptyEntries)
                    where w.Distinct().Intersect(wordsToMatch).Count() ==
wordsToMatch.Count()
                    select sentence;

foreach (string str in sentenceQuery)
{
    Console.WriteLine(str);
}
/* Output:
Historically, the world of data and the world of objects have not been well
integrated
*/
```

A consulta funciona primeiro dividindo o texto em frases e, em seguida, dividindo as sentenças em uma matriz de cadeias de caracteres que contêm cada palavra. Para cada uma dessas matrizes, o método Distinct remove todas as palavras duplicadas e, em seguida, a consulta executa uma operação Intersect na matriz de palavras e na matriz `wordsToMatch`. Se a contagem da interseção for igual à contagem da matriz `wordsToMatch`, todas as palavras foram encontradas nas palavras e a frase original será retornada.

A chamada para Split usa marcas de pontuação como separadores para removê-las da cadeia de caracteres. Se não remover a pontuação, por exemplo, você poderia ter uma cadeia de caracteres “Historically” que não corresponderia a “Historically” na matriz `wordsToMatch`. Talvez você precise usar separadores extras, dependendo dos tipos de pontuação encontrados no texto de origem.

Como combinar consultas LINQ com expressões regulares

O exemplo a seguir mostra como usar a classe Regex para criar uma expressão regular para correspondências mais complexas em cadeias de texto. A consulta LINQ torna fácil a aplicação de filtro exatamente nos arquivos que você deseja pesquisar com a expressão regular e formatar os resultados.

C#

```
string startFolder = """C:\Program Files\dotnet\sdk""";
// Or
// string startFolder = "/usr/local/share/dotnet/sdk";

// Take a snapshot of the file system.
var fileList = from file in Directory.GetFiles(startFolder, "*.*",
SearchOption.AllDirectories)
                let fileInfo = new FileInfo(file)
                select fileInfo;

// Create the regular expression to find all things "Visual".
System.Text.RegularExpressions.Regex searchTerm =
    new System.Text.RegularExpressions.Regex(@"microsoft.net.
(sdk|workload)");

// Search the contents of each .htm file.
// Remove the where clause to find even more matchedValues!
// This query produces a list of files where a match
// was found, and a list of the matchedValues in that file.
// Note: Explicit typing of "Match" in select clause.
// This is required because MatchCollection is not a
// generic IEnumerable collection.
var queryMatchingFiles =
    from file in fileList
    where file.Extension == ".txt"
    let fileText = File.ReadAllText(file.FullName)
    let matches = searchTerm.Matches(fileText)
    where matches.Count > 0
    select new
    {
        name = file.FullName,
        matchedValues = from System.Text.RegularExpressions.Match match in
matches
                        select match.Value
    };

// Execute the query.
Console.WriteLine($"""The term "{searchTerm}" was found in:""");

foreach (var v in queryMatchingFiles)
{
```

```
    // Trim the path a bit, then write
    // the file name in which a match was found.
    string s = v.name.Substring(startFolder.Length - 1);
    Console.WriteLine(s);

    // For this file, write out all the matching strings
    foreach (var v2 in v.matchedValues)
    {
        Console.WriteLine($"  {v2}");
    }
}
```

Você também pode consultar o objeto MatchCollection retornado por uma pesquisa `RegEx`. Apenas o valor de cada correspondência é produzido nos resultados. No entanto, também é possível usar a LINQ para executar todos os tipos de filtragem, classificação e agrupamento nessa coleção. Como MatchCollection é uma coleção IEnumerable não genérica, é necessário declarar explicitamente o tipo da variável de intervalo na consulta.

Colaborar conosco no GitHub

A fonte deste conteúdo pode ser encontrada no GitHub, onde você também pode criar e revisar problemas e solicitações de pull. Para obter mais informações, confira o nosso guia para colaboradores.

.NET

Comentários do .NET

O .NET é um projeto código aberto. Selecione um link para fornecer comentários:

Abrir um problema de documentação

Fornecer comentários sobre o produto

LINQ e coleções

Artigo • 02/05/2024

A maioria das coleções modela uma *sequência* de elementos. Você pode usar LINQ para consultar qualquer tipo de coleção. Outros métodos LINQ encontram elementos em uma coleção, calculam valores dos elementos em uma coleção ou modificam a coleção ou seus elementos. Esses exemplos ajudam você a aprender sobre os métodos LINQ e como usá-los com suas coleções ou outras fontes de dados.

Como descobrir a diferença de conjuntos entre duas listas

Este exemplo mostra como usar o LINQ para comparar duas listas de cadeias de caracteres e retornar as linhas que estão na primeira coleção, mas não na segunda. A primeira coleção de nomes é armazenada no arquivo *names1.txt*:

```txt
Bankov, Peter
Holm, Michael
Garcia, Hugo
Potra, Cristina
Noriega, Fabricio
Aw, Kam Foo
Beebe, Ann
Toyoshima, Tim
Guy, Wey Yuan
Garcia, Debra
```

A segunda coleção de nomes é armazenada no arquivo *names2.txt*. Alguns nomes aparecem em ambas as sequências.

```txt
Liu, Jinghao
Bankov, Peter
Holm, Michael
Garcia, Hugo
Beebe, Ann
Gilchrist, Beth
Myrcha, Jacek
Giakoumakis, Leo
McLin, Nkenge
El Yassir, Mehdi
```

O código a seguir mostra como você pode usar o método Enumerable.Except para localizar elementos na primeira lista que não estão na segunda lista:

C#

```
// Create the IEnumerable data sources.
string[] names1 = File.ReadAllLines("names1.txt");
string[] names2 = File.ReadAllLines("names2.txt");

// Create the query. Note that method syntax must be used here.
var differenceQuery = names1.Except(names2);

// Execute the query.
Console.WriteLine("The following lines are in names1.txt but not
names2.txt");
foreach (string s in differenceQuery)
    Console.WriteLine(s);
/* Output:
 The following lines are in names1.txt but not names2.txt
 Potra, Cristina
 Noriega, Fabricio
 Aw, Kam Foo
 Toyoshima, Tim
 Guy, Wey Yuan
 Garcia, Debra
 */
```

Alguns tipos de operações de consulta, tais como Except, Distinct, Union e Concat, só podem ser expressas em sintaxe baseada em método.

Como combinar e comparar coleções de cadeias de caracteres

Este exemplo mostra como mesclar arquivos que contêm linhas de texto e, em seguida, classificar os resultados. Especificamente, mostra como executar uma concatenação, uma união e uma interseção nos dois conjuntos de linhas de texto. Ele usa os mesmos dois arquivos de texto mostrados no exemplo anterior. O código mostra exemplos dos Enumerable.Concat, Enumerable.Union e Enumerable.Except.

C#

```
//Put text files in your solution folder
string[] fileA = File.ReadAllLines("names1.txt");
string[] fileB = File.ReadAllLines("names2.txt");

//Simple concatenation and sort. Duplicates are preserved.
var concatQuery = fileA.Concat(fileB).OrderBy(s => s);
```

```
// Pass the query variable to another function for execution.
OutputQueryResults(concatQuery, "Simple concatenate and sort. Duplicates are
preserved:");

// Concatenate and remove duplicate names based on
// default string comparer.
var uniqueNamesQuery = fileA.Union(fileB).OrderBy(s => s);
OutputQueryResults(uniqueNamesQuery, "Union removes duplicate names:");

// Find the names that occur in both files (based on
// default string comparer).
var commonNamesQuery = fileA.Intersect(fileB);
OutputQueryResults(commonNamesQuery, "Merge based on intersect:");

// Find the matching fields in each list. Merge the two
// results by using Concat, and then
// sort using the default string comparer.
string nameMatch = "Garcia";

var tempQuery1 = from name in fileA
                 let n = name.Split(',')
                 where n[0] == nameMatch
                 select name;

var tempQuery2 = from name2 in fileB
                 let n2 = name2.Split(',')
                 where n2[0] == nameMatch
                 select name2;

var nameMatchQuery = tempQuery1.Concat(tempQuery2).OrderBy(s => s);
OutputQueryResults(nameMatchQuery, $"""Concat based on partial name match "
{nameMatch}":""");

static void OutputQueryResults(IEnumerable<string> query, string message)
{
    Console.WriteLine(Environment.NewLine + message);
    foreach (string item in query)
    {
        Console.WriteLine(item);
    }
    Console.WriteLine($"{query.Count()} total names in list");
}
/* Output:
    Simple concatenate and sort. Duplicates are preserved:
    Aw, Kam Foo
    Bankov, Peter
    Bankov, Peter
    Beebe, Ann
    Beebe, Ann
    El Yassir, Mehdi
    Garcia, Debra
    Garcia, Hugo
    Garcia, Hugo
    Giakoumakis, Leo
    Gilchrist, Beth
```

```
    Guy, Wey Yuan
    Holm, Michael
    Holm, Michael
    Liu, Jinghao
    McLin, Nkenge
    Myrcha, Jacek
    Noriega, Fabricio
    Potra, Cristina
    Toyoshima, Tim
    20 total names in list

    Union removes duplicate names:
    Aw, Kam Foo
    Bankov, Peter
    Beebe, Ann
    El Yassir, Mehdi
    Garcia, Debra
    Garcia, Hugo
    Giakoumakis, Leo
    Gilchrist, Beth
    Guy, Wey Yuan
    Holm, Michael
    Liu, Jinghao
    McLin, Nkenge
    Myrcha, Jacek
    Noriega, Fabricio
    Potra, Cristina
    Toyoshima, Tim
    16 total names in list

    Merge based on intersect:
    Bankov, Peter
    Holm, Michael
    Garcia, Hugo
    Beebe, Ann
    4 total names in list

    Concat based on partial name match "Garcia":
    Garcia, Debra
    Garcia, Hugo
    Garcia, Hugo
    3 total names in list
*/
```

Como preencher coleções de objetos com dados provenientes de diversas fontes

Este exemplo mostra como mesclar dados de diferentes fontes em uma sequência de novos tipos.

Observação

Não tente unir dados na memória ou dados no sistema de arquivos com os dados que ainda estão em um banco de dados. Essas junções entre domínios podem gerar resultados indefinidos, devido às diferentes formas em que as operações de junção podem ser definidas para consultas de banco de dados e outros tipos de fontes. Além disso, há um risco de que essa operação possa causar uma exceção de falta de memória, se a quantidade de dados no banco de dados for grande o suficiente. Para unir dados de um banco de dados com os dados na memória, primeiro chame `ToList` ou `ToArray` na consulta de banco de dados e, em seguida, realize a junção na coleção retornada.

Este exemplo usa dois arquivos. O primeiro, *names.csv*, contém nomes de alunos e IDs de alunos.

```txt
Omelchenko,Svetlana,111
O'Donnell,Claire,112
Mortensen,Sven,113
Garcia,Cesar,114
Garcia,Debra,115
Fakhouri,Fadi,116
Feng,Hanying,117
Garcia,Hugo,118
Tucker,Lance,119
Adams,Terry,120
Zabokritski,Eugene,121
Tucker,Michael,122
```

O segundo, *scores.csv*, contém IDs de alunos na primeira coluna, seguidas pelas pontuações do exame.

```txt
111, 97, 92, 81, 60
112, 75, 84, 91, 39
113, 88, 94, 65, 91
114, 97, 89, 85, 82
115, 35, 72, 91, 70
116, 99, 86, 90, 94
117, 93, 92, 80, 87
118, 92, 90, 83, 78
119, 68, 79, 88, 92
120, 99, 82, 81, 79
```

```
121, 96, 85, 91, 60
122, 94, 92, 91, 91
```

O exemplo a seguir mostra como usar um registro nomeado `Student` para armazenar dados mesclados, de duas coleções na memória de cadeias de caracteres, que simulam dados de planilha no formato .csv. A ID é usada como a chave para mapear os alunos para suas pontuações.

C#

```
// Each line of names.csv consists of a last name, a first name, and an
// ID number, separated by commas. For example, Omelchenko,Svetlana,111
string[] names = File.ReadAllLines("names.csv");

// Each line of scores.csv consists of an ID number and four test
// scores, separated by commas. For example, 111, 97, 92, 81, 60
string[] scores = File.ReadAllLines("scores.csv");

// Merge the data sources using a named type.
// var could be used instead of an explicit type. Note the dynamic
// creation of a list of ints for the ExamScores member. The first item
// is skipped in the split string because it is the student ID,
// not an exam score.
IEnumerable<Student> queryNamesScores = from nameLine in names
                                        let splitName = nameLine.Split(',')
                                        from scoreLine in scores
                                        let splitScoreLine =
scoreLine.Split(',')
                                        where Convert.ToInt32(splitName[2])
== Convert.ToInt32(splitScoreLine[0])
                                        select new Student
                                        (
                                            FirstName: splitName[0],
                                            LastName: splitName[1],
                                            ID:
Convert.ToInt32(splitName[2]),
                                            ExamScores: (from scoreAsText in
splitScoreLine.Skip(1)
                                                         select
Convert.ToInt32(scoreAsText)
                                                        ).ToArray()
                                        );

// Optional. Store the newly created student objects in memory
// for faster access in future queries. This could be useful with
// very large data files.
List<Student> students = queryNamesScores.ToList();

// Display each student's name and exam score average.
foreach (var student in students)
{
    Console.WriteLine($"The average score of {student.FirstName}
```

```
{student.LastName} is {student.ExamScores.Average()}.");
}
/* Output:
The average score of Omelchenko Svetlana is 82.5.
The average score of O'Donnell Claire is 72.25.
The average score of Mortensen Sven is 84.5.
The average score of Garcia Cesar is 88.25.
The average score of Garcia Debra is 67.
The average score of Fakhouri Fadi is 92.25.
The average score of Feng Hanying is 88.
The average score of Garcia Hugo is 85.75.
The average score of Tucker Lance is 81.75.
The average score of Adams Terry is 85.25.
The average score of Zabokritski Eugene is 83.
The average score of Tucker Michael is 92.
*/
```

Na cláusula select, cada novo objeto `Student` é inicializado dos dados nas duas fontes.

Se você não tiver que armazenar os resultados de uma consulta, as tuplas ou os tipos anônimos poderão ser mais convenientes que os tipos nomeados. O exemplo a seguir realiza a mesma tarefa do exemplo anterior, mas usa tuplas em vez de tipos nomeados:

C#

```
// Merge the data sources by using an anonymous type.
// Note the dynamic creation of a list of ints for the
// ExamScores member. We skip 1 because the first string
// in the array is the student ID, not an exam score.
var queryNamesScores2 = from nameLine in names
                        let splitName = nameLine.Split(',')
                        from scoreLine in scores
                        let splitScoreLine = scoreLine.Split(',')
                        where Convert.ToInt32(splitName[2]) ==
Convert.ToInt32(splitScoreLine[0])
                        select (FirstName: splitName[0],
                                LastName: splitName[1],
                                ExamScores: (from scoreAsText in
splitScoreLine.Skip(1)
                                             select
Convert.ToInt32(scoreAsText))
                                             .ToList()
                               );

// Display each student's name and exam score average.
foreach (var student in queryNamesScores2)
{
    Console.WriteLine($"The average score of {student.FirstName}
{student.LastName} is {student.ExamScores.Average()}.");
}
```

Como consultar um ArrayList com LINQ

Ao usar a LINQ para consultar coleções IEnumerable não genéricas como ArrayList, você deve declarar explicitamente o tipo da variável de intervalo para refletir o tipo específico dos objetos na coleção. Se você tiver um ArrayList de objetos `Student`, sua cláusula from deverá ter uma aparência semelhante a esta:

C#

```
var query = from Student s in arrList
//...
```

Especificando o tipo da variável de intervalo, você está convertendo cada item na ArrayList em um `Student`.

O uso de uma variável de intervalo de tipo explícito em uma expressão de consulta é equivalente a chamar o método Cast. Cast lança uma exceção se a conversão especificada não puder ser realizada. Cast e OfType são os dois métodos de operador de consulta padrão que operam em tipos IEnumerable não genéricos. Para obter mais informações, consulte Relacionamentos de tipo em operações de consulta LINQ. O exemplo a seguir mostra uma consulta sobre um ArrayList.

C#

```
ArrayList arrList = new ArrayList();
arrList.Add(
    new Student
    (
        FirstName: "Svetlana",
        LastName: "Omelchenko",
        ExamScores: new int[] { 98, 92, 81, 60 }
    ));
arrList.Add(
    new Student
    (
        FirstName: "Claire",
        LastName: "O'Donnell",
        ExamScores: new int[] { 75, 84, 91, 39 }
    ));
arrList.Add(
    new Student
    (
        FirstName: "Sven",
        LastName: "Mortensen",
        ExamScores: new int[] { 88, 94, 65, 91 }
    ));
arrList.Add(
    new Student
```

```
    (
        FirstName: "Cesar",
        LastName: "Garcia",
        ExamScores: new int[] { 97, 89, 85, 82 }
    ));

var query = from Student student in arrList
            where student.ExamScores[0] > 95
            select student;

foreach (Student s in query)
    Console.WriteLine(s.LastName + ": " + s.ExamScores[0]);
```

Colaborar conosco no GitHub

A fonte deste conteúdo pode ser encontrada no GitHub, onde você também pode criar e revisar problemas e solicitações de pull. Para obter mais informações, confira o nosso guia para colaboradores.

Comentários do .NET

O .NET é um projeto código aberto. Selecione um link para fornecer comentários:

- Abrir um problema de documentação
- Fornecer comentários sobre o produto

Como estender o LINQ

Artigo • 21/04/2025

Todos os métodos baseados em LINQ seguem um dos dois padrões semelhantes. Eles pegam uma sequência enumerável. Eles retornam uma sequência diferente ou um único valor. A consistência da forma permite que você estenda LINQ escrevendo métodos com uma forma semelhante. Na verdade, as bibliotecas do .NET ganharam novos métodos em muitas versões do .NET desde que o LINQ foi introduzido pela primeira vez. Neste artigo, você verá exemplos de extensão do LINQ escrevendo seus próprios métodos que seguem o mesmo padrão.

Adicionar métodos personalizados para consultas LINQ

Estenda o conjunto de métodos usados para consultas LINQ adicionando métodos de extensão à IEnumerable<T> interface. Por exemplo, além da média padrão ou das operações máximas, você cria um método de agregação personalizado para calcular um único valor de uma sequência de valores. Você também cria um método que funciona como um filtro personalizado ou uma transformação de dados específica para uma sequência de valores e retorna uma nova sequência. Exemplos desses métodos são Distinct, Skipe Reverse.

Ao estender a IEnumerable<T> interface, você pode aplicar seus métodos personalizados a qualquer coleção enumerável. Para obter mais informações, consulte Métodos de extensão.

Um método *de agregação* calcula um único valor de um conjunto de valores. O LINQ fornece vários métodos de agregação, incluindo Average, Mine Max. Você pode criar seu próprio método de agregação adicionando um método de extensão à IEnumerable<T> interface.

A partir do C# 14, você pode declarar um *bloco de extensão* para conter vários membros de extensão. Você declara um bloco de extensão com a palavra-chave `extension` seguida pelo parâmetro receptor entre parênteses. O exemplo de código a seguir mostra como criar um método de extensão chamado `Median` em um bloco de extensão. O método calcula uma mediana para uma sequência de números de tipo `double`.

C#

```
extension(IEnumerable<double>? source)
{
    public double Median()
    {
        if (source is null || !source.Any())
        {
            throw new InvalidOperationException("Cannot compute median for a null
or empty set.");
```

```
        }

        var sortedList =
            source.OrderBy(number => number).ToList();

        int itemIndex = sortedList.Count / 2;

        if (sortedList.Count % 2 == 0)
        {
            // Even number of items.
            return (sortedList[itemIndex] + sortedList[itemIndex - 1]) / 2;
        }
        else
        {
            // Odd number of items.
            return sortedList[itemIndex];
        }
    }
}
```

Você também pode adicionar o `this` modificador a um método estático para declarar um *método de extensão*. O código a seguir mostra o método de extensão equivalente `Median` :

C#

```
public static class EnumerableExtension
{
    public static double Median(this IEnumerable<double>? source)
    {
        if (source is null || !source.Any())
        {
            throw new InvalidOperationException("Cannot compute median for a null
or empty set.");
        }

        var sortedList =
            source.OrderBy(number => number).ToList();

        int itemIndex = sortedList.Count / 2;

        if (sortedList.Count % 2 == 0)
        {
            // Even number of items.
            return (sortedList[itemIndex] + sortedList[itemIndex - 1]) / 2;
        }
        else
        {
            // Odd number of items.
            return sortedList[itemIndex];
        }
    }
}
```

Você chama qualquer método de extensão para qualquer coleção enumerável da mesma forma que chama outros métodos de agregação da IEnumerable<T> interface.

O exemplo de código a seguir mostra como usar o `Median` método para uma matriz de tipo `double`.

C#

```
double[] numbers = [1.9, 2, 8, 4, 5.7, 6, 7.2, 0];
var query = numbers.Median();

Console.WriteLine($"double: Median = {query}");
// This code produces the following output:
//     double: Median = 4.85
```

Você pode *sobrecarregar seu método de agregação* para que ele aceite sequências de vários tipos. A abordagem padrão é criar uma sobrecarga para cada tipo. Outra abordagem é criar uma sobrecarga que pega um tipo genérico e convertê-lo em um tipo específico, usando um delegado. Você também pode combinar ambas as abordagens.

Você pode criar uma sobrecarga específica para cada tipo que deseja dar suporte. O exemplo de código a seguir mostra uma sobrecarga do `Median` método para o `int` tipo.

C#

```
// int overload
public static double Median(this IEnumerable<int> source) =>
    (from number in source select (double)number).Median();
```

Agora você pode chamar as sobrecargas `Median` para os tipos `integer` e `double`, conforme mostrado no código a seguir:

C#

```
double[] numbers1 = [1.9, 2, 8, 4, 5.7, 6, 7.2, 0];
var query1 = numbers1.Median();

Console.WriteLine($"double: Median = {query1}");

int[] numbers2 = [1, 2, 3, 4, 5];
var query2 = numbers2.Median();

Console.WriteLine($"int: Median = {query2}");
// This code produces the following output:
//     double: Median = 4.85
//     int: Median = 3
```

Você também pode criar uma sobrecarga que aceita uma *sequência genérica* de objetos. Essa sobrecarga usa um delegado como um parâmetro e o usa para converter uma sequência de objetos de um tipo genérico em um tipo específico.

O código a seguir mostra uma sobrecarga do `Median` método que usa o Func<T,TResult> delegado como um parâmetro. Esse delegado usa um objeto do tipo genérico T e retorna um objeto do tipo `double`.

C#

```
// generic overload
public static double Median<T>(
    this IEnumerable<T> numbers, Func<T, double> selector) =>
    (from num in numbers select selector(num)).Median();
```

Agora você pode chamar o `Median` método para uma sequência de objetos de qualquer tipo. Se o tipo não tiver sua própria sobrecarga de método, você precisará passar um parâmetro delegado. Em C#, você pode usar uma expressão lambda para essa finalidade. Além disso, somente no Visual Basic, se você usar a `Aggregate` cláusula ou `Group By` em vez da chamada de método, poderá passar qualquer valor ou expressão que esteja no escopo desta cláusula.

O código de exemplo a seguir mostra como chamar o `Median` método para uma matriz de inteiros e uma matriz de cadeias de caracteres. Para cadeias de caracteres, a mediana dos comprimentos das cadeias de caracteres na matriz é calculada. O exemplo mostra como passar o Func<T,TResult> parâmetro delegado para o `Median` método para cada caso.

C#

```
int[] numbers3 = [1, 2, 3, 4, 5];

/*
    You can use the num => num lambda expression as a parameter for the Median
method
    so that the compiler will implicitly convert its value to double.
    If there is no implicit conversion, the compiler will display an error
message.
*/
var query3 = numbers3.Median(num => num);

Console.WriteLine($"int: Median = {query3}");

string[] numbers4 = ["one", "two", "three", "four", "five"];

// With the generic overload, you can also use numeric properties of objects.
var query4 = numbers4.Median(str => str.Length);

Console.WriteLine($"string: Median = {query4}");
// This code produces the following output:
```

```
//      int: Median = 3
//      string: Median = 4
```

Você pode estender a IEnumerable<T> interface com um método de consulta personalizado que retorna uma *sequência de valores*. Nesse caso, o método deve retornar uma coleção do tipo IEnumerable<T>. Esses métodos podem ser usados para aplicar filtros ou transformações de dados a uma sequência de valores.

O exemplo a seguir mostra como criar um método de extensão chamado `AlternateElements` que retorna todos os outros elementos em uma coleção, começando pelo primeiro elemento.

C#

```
// Extension method for the IEnumerable<T> interface.
// The method returns every other element of a sequence.
public static IEnumerable<T> AlternateElements<T>(this IEnumerable<T> source)
{
    int index = 0;
    foreach (T element in source)
    {
        if (index % 2 == 0)
        {
            yield return element;
        }

        index++;
    }
}
```

Você pode chamar esse método de extensão para qualquer coleção enumerável, assim como você chamaria outros métodos da IEnumerable<T> interface, conforme mostrado no código a seguir:

C#

```
string[] strings = ["a", "b", "c", "d", "e"];

var query5 = strings.AlternateElements();

foreach (var element in query5)
{
    Console.WriteLine(element);
}
// This code produces the following output:
//      a
//      c
//      e
```

Cada exemplo mostrado neste artigo tem um *receptor* diferente. Isso significa que cada método deve ser declarado em um bloco de extensão diferente que especifica o receptor exclusivo. O exemplo de código a seguir mostra uma única classe estática com três blocos de extensão diferentes, cada um dos quais contém um dos métodos definidos neste artigo:

C#

```
public static class EnumerableExtension
{
    extension(IEnumerable<double>? source)
    {
        public double Median()
        {
            if (source is null || !source.Any())
            {
                throw new InvalidOperationException("Cannot compute median for a
null or empty set.");
            }

            var sortedList =
                source.OrderBy(number => number).ToList();

            int itemIndex = sortedList.Count / 2;

            if (sortedList.Count % 2 == 0)
            {
                // Even number of items.
                return (sortedList[itemIndex] + sortedList[itemIndex - 1]) / 2;
            }
            else
            {
                // Odd number of items.
                return sortedList[itemIndex];
            }
        }
    }

    extension(IEnumerable<int> source)
    {
        public double Median() =>
            (from number in source select (double)number).Median();
    }

    extension<T>(IEnumerable<T> source)
    {
        public double Median(Func<T, double> selector) =>
            (from num in source select selector(num)).Median();

        public IEnumerable<T> AlternateElements()
        {
            int index = 0;
            foreach (T element in source)
            {
```

```
                if (index % 2 == 0)
                {
                    yield return element;
                }

                index++;
            }
        }
    }
}
```

O bloco de extensão final declara um bloco de extensão genérico. O parâmetro de tipo para o receptor é declarado no `extension` ele próprio.

O exemplo anterior declara um membro de extensão em cada bloco de extensão. Na maioria dos casos, você cria vários membros de extensão para o mesmo receptor. Nesses casos, você deve declarar as extensões para esses membros em um único bloco de extensão.

Consulta com base no estado de tempo de execução

Artigo • 02/05/2024

Na maioria das consultas LINQ, a forma geral da consulta é definida no código. Você pode filtrar itens usando uma cláusula `where`, classificar a coleção de saída usando `orderby`, itens de grupo ou executar alguma computação. Seu código pode fornecer parâmetros para o filtro, a chave de classificação ou outras expressões que fazem parte da consulta. No entanto, a forma geral da consulta não pode ser alterada. Neste artigo, você aprenderá técnicas para usar a interface System.Linq.IQueryable<T> e tipos que a implementam para modificar a forma de uma consulta em tempo de execução.

Você usa essas técnicas para criar consultas em tempo de execução, em que algum estado de entrada ou tempo de execução do usuário altera os métodos de consulta que você deseja usar como parte da consulta. Você edita a consulta adicionando, removendo ou modificando cláusulas de consulta.

> **Observação**
>
> Adicione `using System.Linq.Expressions;` e `using static System.Linq.Expressions.Expression;` na parte superior do seu arquivo *.cs*.

Considere o código que define um IQueryable ou um IQueryable<T> em relação a uma fonte de dados:

C#

```
string[] companyNames = [
    "Consolidated Messenger", "Alpine Ski House", "Southridge Video",
    "City Power & Light", "Coho Winery", "Wide World Importers",
    "Graphic Design Institute", "Adventure Works", "Humongous Insurance",
    "Woodgrove Bank", "Margie's Travel", "Northwind Traders",
    "Blue Yonder Airlines", "Trey Research", "The Phone Company",
    "Wingtip Toys", "Lucerne Publishing", "Fourth Coffee"
];

// Use an in-memory array as the data source, but the IQueryable could have
come
// from anywhere -- an ORM backed by a database, a web request, or any other
LINQ provider.
IQueryable<string> companyNamesSource = companyNames.AsQueryable();
var fixedQry = companyNames.OrderBy(x => x);
```

Toda vez que você executar o código anterior, exatamente a mesma consulta será executada. Vamos aprender a modificar a consulta para estendê-la ou modificá-la. Fundamentalmente, uma IQueryable tem dois componentes:

- Expression – Uma representação (agnóstica quanto à linguagem e à fonte de dados) dos componentes da consulta atual, na forma de uma árvore de expressão.
- Provider – Uma instância de um provedor LINQ, que sabe como materializar a consulta atual em um valor ou conjunto de valores.

No contexto de consulta dinâmica, o provedor geralmente permanece o mesmo, ao passo que a árvore de expressão da consulta varia de uma consulta para a outra.

As árvores de expressão são imutáveis; se você quiser uma árvore de expressão diferente e, portanto, uma consulta diferente, precisará traduzir a árvore de expressão existente para uma nova. As seguintes seções descrevem técnicas específicas para executar consultas diferentes em resposta ao estado de runtime:

- Usar o estado de runtime de dentro da árvore de expressão
- Chamar mais métodos LINQ
- Variar a árvore de expressão passada para os métodos LINQ
- Construir uma árvore de expressão Expression<TDelegate> usando os métodos de fábrica em Expression
- Adicionar nós de chamada de método a uma árvore de expressão de IQueryable
- Construir cadeias de caracteres e usar a Biblioteca dinâmica do LINQ

Cada uma das técnicas permite mais funcionalidades, mas a um custo de maior complexidade.

Usar o estado de runtime de dentro da árvore de expressão

A maneira mais simples de executar consultas dinamicamente é referenciar o estado de runtime diretamente na consulta por meio de uma variável fechada, como `length` no seguinte exemplo de código:

C#

```
var length = 1;
var qry = companyNamesSource
    .Select(x => x.Substring(0, length))
    .Distinct();

Console.WriteLine(string.Join(",", qry));
// prints: C, A, S, W, G, H, M, N, B, T, L, F
```

```
length = 2;
Console.WriteLine(string.Join(",", qry));
// prints: Co, Al, So, Ci, Wi, Gr, Ad, Hu, Wo, Ma, No, Bl, Tr, Th, Lu, Fo
```

A árvore de expressão interna e, portanto, a consulta, não é modificada; a consulta retorna valores diferentes apenas porque o valor de `length` foi alterado.

Chamar mais métodos LINQ

Em geral, os métodos LINQ internos em Queryable executam duas etapas:

- Empacote a árvore de expressão atual em uma MethodCallExpression representando a chamada de método.
- Passar a árvore de expressão empacotada de volta para o provedor, seja para retornar um valor por meio do método IQueryProvider.Execute do provedor ou para retornar um objeto de consulta traduzido por meio do método IQueryProvider.CreateQuery.

Você pode substituir a consulta original pelo resultado do método de retorno System.Linq.IQueryable<T> a fim de obter uma nova consulta. Você pode usar o estado de tempo de execução, como no exemplo a seguir:

C#

```
// bool sortByLength = /* ... */;

var qry = companyNamesSource;
if (sortByLength)
{
    qry = qry.OrderBy(x => x.Length);
}
```

Variar a árvore de expressão passada para os métodos LINQ

Você pode passar expressões diferentes para os métodos LINQ, dependendo do estado de runtime:

C#

```
// string? startsWith = /* ... */;
// string? endsWith = /* ... */;
```

```
Expression<Func<string, bool>> expr = (startsWith, endsWith) switch
{
    ("" or null, "" or null) => x => true,
    (_, "" or null) => x => x.StartsWith(startsWith),
    ("" or null, _) => x => x.EndsWith(endsWith),
    (_, _) => x => x.StartsWith(startsWith) || x.EndsWith(endsWith)
};

var qry = companyNamesSource.Where(expr);
```

Talvez você também queira compor as diversas subexpressões usando outra biblioteca, como o PredicateBuilder da LinqKit:

C#

```
// This is functionally equivalent to the previous example.

// using LinqKit;
// string? startsWith = /* ... */;
// string? endsWith = /* ... */;

Expression<Func<string, bool>>? expr = PredicateBuilder.New<string>(false);
var original = expr;
if (!string.IsNullOrEmpty(startsWith))
{
    expr = expr.Or(x => x.StartsWith(startsWith));
}
if (!string.IsNullOrEmpty(endsWith))
{
    expr = expr.Or(x => x.EndsWith(endsWith));
}
if (expr == original)
{
    expr = x => true;
}

var qry = companyNamesSource.Where(expr);
```

Construir árvores de expressão e consultas usando métodos de fábrica

Em todos os exemplos até este ponto, você conhece o tipo de elemento no tipo de compilação `string` e, portanto, o tipo da consulta `IQueryable<string>`. Talvez seja necessário adicionar componentes a uma consulta de qualquer tipo de elemento ou adicionar componentes diferentes, dependendo do tipo de elemento. Você pode criar árvores de expressão do zero, usando os métodos de fábrica em

System.Linq.Expressions.Expression, personalizando assim a expressão em tempo de execução para um tipo de elemento específico.

Construindo uma expressão <TDelegate>

Quando você constrói uma expressão para passar para um dos métodos LINQ, na verdade você está criando uma instância de System.Linq.Expressions.Expression<TDelegate>, em que `TDelegate` é algum tipo de delegado, como `Func<string, bool>`, `Action` ou um tipo de delegado personalizado.

System.Linq.Expressions.Expression<TDelegate> herda de LambdaExpression, que representa uma expressão lambda completa como o exemplo a seguir:

```
C#

Expression<Func<string, bool>> expr = x => x.StartsWith("a");
```

Uma LambdaExpression tem dois componentes:

1. Uma lista de parâmetros – `(string x)` – representada pela propriedade Parameters.
2. Um corpo – `x.StartsWith("a")` – representado pela propriedade Body.

As etapas básicas para construção de um Expression<TDelegate> são as seguintes:

1. Defina objetos ParameterExpression para cada um dos parâmetros (se houver) na expressão lambda, usando o método de fábrica Parameter.

   ```
   C#

   ParameterExpression x = Parameter(typeof(string), "x");
   ```

2. Construa o corpo do seu LambdaExpression, usando ParameterExpression definido e os métodos de fábrica em Expression. Por exemplo, uma expressão que representa `x.StartsWith("a")` poderia ser construída assim:

   ```
   C#

   Expression body = Call(
       x,
       typeof(string).GetMethod("StartsWith", [typeof(string)])!,
       Constant("a")
   );
   ```

3. Encapsule os parâmetros e o corpo em um Expression<TDelegate> do tipo tempo de compilação, usando a sobrecarga do método de fábrica Lambda apropriada:

```
C#

Expression<Func<string, bool>> expr = Lambda<Func<string, bool>>(body,
x);
```

As seções a seguir descrevem um cenário no qual talvez você queira construir uma Expression<TDelegate> para passar para um método LINQ. Ele fornece um exemplo completo de como fazer isso usando os métodos de fábrica.

Construir uma consulta completa em tempo de execução

Você deseja escrever consultas que funcionam com vários tipos de entidade:

```
C#

record Person(string LastName, string FirstName, DateTime DateOfBirth);
record Car(string Model, int Year);
```

Para qualquer um desses tipos de entidade, você deve filtrar e retornar somente as entidades que têm determinado texto dentro de um dos campos `string` delas. Para `Person`, você deve pesquisar as propriedades `FirstName` e `LastName`:

```
C#

string term = /* ... */;
var personsQry = new List<Person>()
    .AsQueryable()
    .Where(x => x.FirstName.Contains(term) || x.LastName.Contains(term));
```

Já para `Car`, você deve pesquisar apenas a propriedade `Model`:

```
C#

string term = /* ... */;
var carsQry = new List<Car>()
    .AsQueryable()
    .Where(x => x.Model.Contains(term));
```

Embora você possa gravar uma função personalizada para `IQueryable<Person>` e outra para `IQueryable<Car>`, a função a seguir adiciona essa filtragem a qualquer consulta existente, independentemente do tipo do elemento.

C#

```
// using static System.Linq.Expressions.Expression;

IQueryable<T> TextFilter<T>(IQueryable<T> source, string term)
{
    if (string.IsNullOrEmpty(term)) { return source; }

    // T is a compile-time placeholder for the element type of the query.
    Type elementType = typeof(T);

    // Get all the string properties on this specific type.
    PropertyInfo[] stringProperties = elementType
        .GetProperties()
        .Where(x => x.PropertyType == typeof(string))
        .ToArray();
    if (!stringProperties.Any()) { return source; }

    // Get the right overload of String.Contains
    MethodInfo containsMethod = typeof(string).GetMethod("Contains",
[typeof(string)])!;

    // Create a parameter for the expression tree:
    // the 'x' in 'x => x.PropertyName.Contains("term")'
    // The type of this parameter is the query's element type
    ParameterExpression prm = Parameter(elementType);

    // Map each property to an expression tree node
    IEnumerable<Expression> expressions = stringProperties
        .Select(prp =>
            // For each property, we have to construct an expression tree
node like x.PropertyName.Contains("term")
            Call(                   // .Contains(...)
                Property(           // .PropertyName
                    prm,            // x
                    prp
                ),
                containsMethod,
                Constant(term)      // "term"
            )
        );

    // Combine all the resultant expression nodes using ||
    Expression body = expressions
        .Aggregate((prev, current) => Or(prev, current));

    // Wrap the expression body in a compile-time-typed lambda expression
    Expression<Func<T, bool>> lambda = Lambda<Func<T, bool>>(body, prm);
```

```
    // Because the lambda is compile-time-typed (albeit with a generic
parameter), we can use it with the Where method
    return source.Where(lambda);
}
```

Como a função `TextFilter` usa e retorna um IQueryable<T> (e não apenas um IQueryable), você pode adicionar mais elementos de consulta do tipo tempo de compilação após o filtro de texto.

C#

```
var qry = TextFilter(
        new List<Person>().AsQueryable(),
        "abcd"
    )
    .Where(x => x.DateOfBirth < new DateTime(2001, 1, 1));

var qry1 = TextFilter(
        new List<Car>().AsQueryable(),
        "abcd"
    )
    .Where(x => x.Year == 2010);
```

Adicionar nós de chamada de método à árvore de expressão IQueryable<TDelegate>

Se você tiver um IQueryable em vez de IQueryable<T>, não poderá chamar diretamente os métodos LINQ genéricos. Uma alternativa é criar a árvore de expressão interna conforme mostrado no exemplo acima e usar a reflexão para invocar o método LINQ apropriado e passar como parâmetro a árvore de expressão.

Você também pode duplicar a funcionalidade do método LINQ, empacotando toda a árvore em um MethodCallExpression que representa uma chamada ao método LINQ:

C#

```
IQueryable TextFilter_Untyped(IQueryable source, string term)
{
    if (string.IsNullOrEmpty(term)) { return source; }
    Type elementType = source.ElementType;

    // The logic for building the ParameterExpression and the
LambdaExpression's body is the same as in the previous example,
    // but has been refactored into the constructBody function.
    (Expression? body, ParameterExpression? prm) =
constructBody(elementType, term);
    if (body is null) { return source; }
```

```
    Expression filteredTree = Call(
        typeof(Queryable),
        "Where",
        [elementType],
        source.Expression,
        Lambda(body, prm!)
    );

    return source.Provider.CreateQuery(filteredTree);
}
```

Nesse caso, você não tem um espaço reservado genérico `T` em tempo de compilação, portanto, você usa a sobrecarga Lambda que não requer informações do tipo tempo de compilação e que produz um LambdaExpression em vez de um Expression<TDelegate>.

Biblioteca Dinâmica do LINQ

A construção de árvores de expressão usando métodos de fábrica é relativamente complexa; é mais fácil compor cadeias de caracteres. A Biblioteca Dinâmica do LINQ expõe um conjunto de métodos de extensão em IQueryable, correspondentes aos métodos LINQ padrão em Queryable e que aceitam cadeias de caracteres em uma sintaxe especial em vez de árvores de expressão. A biblioteca gera a árvore de expressão apropriada com base na cadeia de caracteres e pode retornar o IQueryable traduzido resultante.

Por exemplo, o exemplo anterior pode ser reescrito da seguinte maneira:

C#

```
// using System.Linq.Dynamic.Core

IQueryable TextFilter_Strings(IQueryable source, string term)
{
    if (string.IsNullOrEmpty(term)) { return source; }

    var elementType = source.ElementType;

    // Get all the string property names on this specific type.
    var stringProperties =
        elementType.GetProperties()
            .Where(x => x.PropertyType == typeof(string))
            .ToArray();
    if (!stringProperties.Any()) { return source; }

    // Build the string expression
    string filterExpr = string.Join(
        " || ",
        stringProperties.Select(prp => $"{prp.Name}.Contains(@0)")
```

```
    );

    return source.Where(filterExpr, term);
}
```

Colaborar conosco no GitHub

A fonte deste conteúdo pode ser encontrada no GitHub, onde você também pode criar e revisar problemas e solicitações de pull. Para obter mais informações, confira o nosso guia para colaboradores.

Comentários do .NET

O .NET é um projeto código aberto. Selecione um link para fornecer comentários:

Abrir um problema de documentação

Fornecer comentários sobre o produto

Programação assíncrona com async e await

Artigo • 31/03/2025

O modelo de Programação Assíncrona de Tarefas (TAP) fornece uma camada de abstração sobre a codificação assíncrona típica. Nesse modelo, você escreve código como uma sequência de instruções, o mesmo que de costume. A diferença é que você pode ler seu código baseado em tarefa à medida que o compilador processa cada instrução e antes de começar a processar a próxima instrução. Para realizar esse modelo, o compilador executa muitas transformações para concluir cada tarefa. Algumas instruções podem iniciar o trabalho e retornar um objeto Task que representa o trabalho em andamento e o compilador deve resolver essas transformações. O objetivo da programação assíncrona da tarefa é habilitar o código que lê como uma sequência de instruções, mas é executado em uma ordem mais complicada. A execução é baseada na alocação de recursos externos e quando as tarefas são concluídas.

O modelo de programação assíncrona da tarefa é análogo à forma como as pessoas dão instruções para processos que incluem tarefas assíncronas. Este artigo usa um exemplo com instruções para fazer café da manhã para mostrar como as palavras-chave `async` e `await` facilitam o raciocínio sobre o código que inclui uma série de instruções assíncronas. As instruções para fazer um café da manhã podem ser fornecidas como uma lista:

1. Sirva uma xícara de café.
2. Aqueça uma panela e, em seguida, frite dois ovos.
3. Frite três fatias de bacon.
4. Torra dois pedaços de pão.
5. Espalhe manteiga e geléia na torrada.
6. Despeje um copo de suco de laranja.

Se você tiver experiência com culinária, poderá concluir estas instruções **de forma assíncrona**. Você começa a aquecer a panela para ovos, em seguida, começar a fritar o bacon. Você coloca o pão na torradeira, depois começa a cozinhar os ovos. Em cada etapa do processo, você inicia uma tarefa e, em seguida, faz a transição para outras tarefas que estão prontas para sua atenção.

Cozinhar café da manhã é um bom exemplo de trabalho assíncrono que não é paralelo. Uma pessoa (ou thread) pode lidar com todas as tarefas. Uma pessoa pode fazer o café da manhã de forma assíncrona iniciando a próxima tarefa antes que a tarefa anterior seja concluída. Cada tarefa de cozimento progride independentemente de alguém estar

assistindo ativamente o processo. Assim que você começar a aquecer a panela para os ovos, você pode começar a fritar o bacon. Depois que o bacon começar a cozinhar, você pode colocar o pão na torradeira.

Para um algoritmo paralelo, você precisa de várias pessoas que cozinham (ou vários threads). Uma pessoa cozinha os ovos, outra frita o bacon, e assim por diante. Cada pessoa se concentra em uma tarefa específica. Cada pessoa que está cozinhando (ou cada thread) está bloqueada sincronizadamente enquanto espera a conclusão da tarefa atual: o bacon pronto para virar, o pão pronto para sair da torradeira, e assim por diante.

Considere a mesma lista de instruções síncronas escritas como instruções de código C#:

C#

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

namespace AsyncBreakfast
{
    // These classes are intentionally empty for the purpose of this
example. They are simply marker classes for the purpose of demonstration,
contain no properties, and serve no other purpose.
    internal class Bacon { }
```

```
    internal class Coffee { }
    internal class Egg { }
    internal class Juice { }
    internal class Toast { }

    class Program
    {
        static void Main(string[] args)
        {
            Coffee cup = PourCoffee();
            Console.WriteLine("coffee is ready");

            Egg eggs = FryEggs(2);
            Console.WriteLine("eggs are ready");

            Bacon bacon = FryBacon(3);
            Console.WriteLine("bacon is ready");

            Toast toast = ToastBread(2);
            ApplyButter(toast);
            ApplyJam(toast);
            Console.WriteLine("toast is ready");

            Juice oj = PourOJ();
            Console.WriteLine("oj is ready");
            Console.WriteLine("Breakfast is ready!");
        }

        private static Juice PourOJ()
        {
            Console.WriteLine("Pouring orange juice");
            return new Juice();
        }

        private static void ApplyJam(Toast toast) =>
            Console.WriteLine("Putting jam on the toast");

        private static void ApplyButter(Toast toast) =>
            Console.WriteLine("Putting butter on the toast");

        private static Toast ToastBread(int slices)
        {
            for (int slice = 0; slice < slices; slice++)
            {
                Console.WriteLine("Putting a slice of bread in the
toaster");
            }
            Console.WriteLine("Start toasting...");
            Task.Delay(3000).Wait();
            Console.WriteLine("Remove toast from toaster");

            return new Toast();
        }

        private static Bacon FryBacon(int slices)
```

```
        {
            Console.WriteLine($"putting {slices} slices of bacon in the
pan");
            Console.WriteLine("cooking first side of bacon...");
            Task.Delay(3000).Wait();
            for (int slice = 0; slice < slices; slice++)
            {
                Console.WriteLine("flipping a slice of bacon");
            }
            Console.WriteLine("cooking the second side of bacon...");
            Task.Delay(3000).Wait();
            Console.WriteLine("Put bacon on plate");

            return new Bacon();
        }

        private static Egg FryEggs(int howMany)
        {
            Console.WriteLine("Warming the egg pan...");
            Task.Delay(3000).Wait();
            Console.WriteLine($"cracking {howMany} eggs");
            Console.WriteLine("cooking the eggs ...");
            Task.Delay(3000).Wait();
            Console.WriteLine("Put eggs on plate");

            return new Egg();
        }

        private static Coffee PourCoffee()
        {
            Console.WriteLine("Pouring coffee");
            return new Coffee();
        }
    }
}
```

Se você interpretar essas instruções como um computador faria, o café da manhã leva cerca de 30 minutos para se preparar. A duração é a soma dos tempos de tarefa individuais. O computador bloqueia cada instrução até que todo o trabalho seja concluído e, então, prossegue para a próxima instrução de tarefa. Essa abordagem pode levar um tempo significativo. No exemplo de café da manhã, o método de computador cria um café da manhã insatisfatório. Tarefas posteriores na lista síncrona, como brindar o pão, não são iniciadas até que as tarefas anteriores sejam concluídas. Alguns alimentos ficam frios antes do café da manhã estar pronto para servir.

Se você quiser que o computador execute instruções de forma assíncrona, você deve escrever código assíncrono. Ao escrever programas cliente, você deseja que a interface do usuário responda à entrada do usuário. Seu aplicativo não deve congelar toda a interação ao baixar dados da Web. Ao escrever programas de servidor, você não deseja bloquear threads que possam estar atendendo a outras solicitações. O uso de código

síncrono quando existem alternativas assíncronas prejudica sua capacidade de escalar de forma mais econômica. Você paga pelos threads bloqueados.

Aplicativos modernos bem-sucedidos exigem código assíncrono. Sem suporte de linguagem, escrever código assíncrono requer retornos de chamada, eventos de conclusão ou outros meios que obscureçam a intenção original do código. A vantagem do código síncrono é a ação passo a passo que facilita a verificação e a compreensão. Modelos assíncronos tradicionais forçam você a se concentrar na natureza assíncrona do código, não nas ações fundamentais do código.

Não bloqueie, aguarde em vez disso

O código anterior realça uma prática de programação infeliz: escrever código síncrono para executar operações assíncronas. O código impede que o thread atual faça qualquer outro trabalho. O código não interrompe o thread enquanto há tarefas em execução. O resultado desse modelo é semelhante ao olhar para a torradeira depois que você coloca o pão. Ignore as interrupções e não inicie outras tarefas até que o pão apareça. Você não tira a manteiga e a geléia da geladeira. Você pode não perceber quando um fogo começa no fogão. Você quer torrar o pão e lidar com outras preocupações ao mesmo tempo. O mesmo acontece com seu código.

Você pode começar atualizando o código para que o thread não seja bloqueado enquanto as tarefas estão em execução. A palavra-chave `await` fornece uma maneira não bloqueante de iniciar uma tarefa e continuar a execução quando a tarefa for concluída. Uma versão simples e assíncrona do código de café da manhã se parece com o seguinte trecho:

C#

```
static async Task Main(string[] args)
{
    Coffee cup = PourCoffee();
    Console.WriteLine("coffee is ready");

    Egg eggs = await FryEggsAsync(2);
    Console.WriteLine("eggs are ready");

    Bacon bacon = await FryBaconAsync(3);
    Console.WriteLine("bacon is ready");

    Toast toast = await ToastBreadAsync(2);
    ApplyButter(toast);
    ApplyJam(toast);
    Console.WriteLine("toast is ready");

    Juice oj = PourOJ();
```

```
    Console.WriteLine("oj is ready");
    Console.WriteLine("Breakfast is ready!");
}
```

O código atualiza os corpos do método de `FryEggsAsync`, `FryBaconAsync` e `ToastBreadAsync` para retornar objetos `Task<Egg>`, `Task<Bacon>` e `Task<Toast>`, respectivamente. Os nomes de método incluem o sufixo "Async". O método `Main` retorna o objeto `Task`, embora não tenha uma expressão `return`, que é por design. Para obter mais informações, consulte Avaliação de uma função assíncrona que retorna nulo.

> ⓘ **Observação**
>
> O código atualizado ainda não aproveita os principais recursos de programação assíncrona, o que pode resultar em tempos de conclusão mais curtos. O código processa as tarefas na mesma quantidade de tempo que a versão síncrona inicial. Para obter as implementações completas do método, consulte a versão final **do código** posteriormente neste artigo.

Vamos aplicar o exemplo de café da manhã ao código atualizado. O thread não bloqueia enquanto os ovos ou o bacon estão sendo cozidos, mas o código também não inicia outras tarefas até que o trabalho atual seja concluído. Você ainda coloca o pão na torradeira e fica olhando para a torradeira até que o pão salte para fora, mas agora você pode responder às interrupções. Em um restaurante onde vários pedidos são feitos, o cozinheiro pode começar um novo pedido enquanto outro já está cozinhando.

No código atualizado, o thread que está fazendo o café da manhã não é bloqueado enquanto aguarda qualquer tarefa iniciada que esteja inacabada. Para alguns aplicativos, essa alteração é tudo o que você precisa. Você pode habilitar seu aplicativo para dar suporte à interação do usuário enquanto os dados são baixados da Web. Em outros cenários, talvez você queira iniciar outras tarefas enquanto aguarda a conclusão da tarefa anterior.

Iniciar tarefas simultaneamente

Para a maioria das operações, você deseja iniciar várias tarefas independentes imediatamente. À medida que cada tarefa for concluída, você iniciará outro trabalho pronto para começar. Ao aplicar essa metodologia ao exemplo de café da manhã, você pode preparar o café da manhã mais rapidamente. Você também prepara tudo quase ao mesmo tempo, para que possa desfrutar de um café da manhã quente.

A classe System.Threading.Tasks.Task e os tipos relacionados são classes que você pode usar para aplicar esse estilo de raciocínio a tarefas que estão em andamento. Essa abordagem permite que você escreva um código que se assemelha mais à maneira como você cria o café da manhã na vida real. Você começa a cozinhar os ovos, bacon e torradas ao mesmo tempo. Como cada item alimentar requer ação, você volta sua atenção para essa tarefa, conclui a ação e, em seguida, espera por outra coisa que exija sua atenção.

Em seu código, você inicia uma tarefa e se mantém no objeto Task que representa o trabalho. Use o método `await` na tarefa para atrasar a atuação no trabalho até que o resultado esteja pronto.

Aplique essas alterações ao código do café da manhã. A primeira etapa é armazenar as tarefas para operações quando elas forem iniciadas, em vez de usar a expressão `await`:

C#

```
Coffee cup = PourCoffee();
Console.WriteLine("Coffee is ready");

Task<Egg> eggsTask = FryEggsAsync(2);
Egg eggs = await eggsTask;
Console.WriteLine("Eggs are ready");

Task<Bacon> baconTask = FryBaconAsync(3);
Bacon bacon = await baconTask;
Console.WriteLine("Bacon is ready");

Task<Toast> toastTask = ToastBreadAsync(2);
Toast toast = await toastTask;
ApplyButter(toast);
ApplyJam(toast);
Console.WriteLine("Toast is ready");

Juice oj = PourOJ();
Console.WriteLine("Oj is ready");
Console.WriteLine("Breakfast is ready!");
```

Essas revisões não ajudam a preparar seu café da manhã mais rápido. A expressão `await` é aplicada a todas as tarefas assim que elas começam. A próxima etapa é mover as expressões de `await` do bacon e dos ovos para o final do método, antes de servir o café da manhã:

C#

```
Coffee cup = PourCoffee();
Console.WriteLine("Coffee is ready");
```

```
Task<Egg> eggsTask = FryEggsAsync(2);
Task<Bacon> baconTask = FryBaconAsync(3);
Task<Toast> toastTask = ToastBreadAsync(2);

Toast toast = await toastTask;
ApplyButter(toast);
ApplyJam(toast);
Console.WriteLine("Toast is ready");
Juice oj = PourOJ();
Console.WriteLine("Oj is ready");

Egg eggs = await eggsTask;
Console.WriteLine("Eggs are ready");
Bacon bacon = await baconTask;
Console.WriteLine("Bacon is ready");

Console.WriteLine("Breakfast is ready!");
```

Agora você tem um café da manhã preparado antecipadamente, que leva cerca de 20 minutos para ficar pronto. O tempo total de cozimento é reduzido porque algumas tarefas são executadas simultaneamente.

As atualizações de código melhoram o processo de preparação reduzindo o tempo de cozimento, mas introduzem uma regressão queimando os ovos e bacon. Você inicia todas as tarefas assíncronas de uma só vez. Você aguarda cada tarefa somente quando precisar dos resultados. O código pode ser semelhante ao programa em um aplicativo Web que faz solicitações para microsserviços diferentes e, em seguida, combina os resultados em uma única página. Faça todas as solicitações imediatamente, aplique a expressão `await` em todas essas tarefas e componha a página web.

Suportar a composição com tarefas

As revisões de código anteriores ajudam a preparar tudo para o café da manhã ao mesmo tempo, exceto a torrada. O processo de fazer a torrada é uma *composição* de uma operação assíncrona (torrar o pão) com operações síncronas (passar manteiga e geleia na torrada). Este exemplo ilustra um conceito importante sobre programação assíncrona:

> ⓘ **Importante**
>
> A composição de uma operação assíncrona seguida pelo trabalho síncrono é uma operação assíncrona. De outra forma, se qualquer parte de uma operação for assíncrona, toda a operação será assíncrona.

Nas atualizações anteriores, você aprendeu a usar objetos Task ou Task<TResult> para realizar tarefas em execução. Aguarde a conclusão de cada tarefa antes de utilizar o resultado. A próxima etapa é criar métodos que representem a combinação de outros trabalhos. Antes de servir o café da manhã, você deve esperar pela tarefa de tostar o pão antes de passar a manteiga e a geleia.

Você pode representar esse trabalho com o seguinte código:

C#

```
static async Task<Toast> MakeToastWithButterAndJamAsync(int number)
{
    var toast = await ToastBreadAsync(number);
    ApplyButter(toast);
    ApplyJam(toast);

    return toast;
}
```

O método `MakeToastWithButterAndJamAsync` tem o modificador `async` em sua assinatura que sinaliza para o compilador que o método contém uma expressão `await` e contém operações assíncronas. Este método representa a tarefa que torra o pão e, em seguida, passa manteiga e geleia. O método retorna um objeto Task<TResult> que representa a composição das três operações.

O bloco principal de código revisado agora tem esta aparência:

C#

```
static async Task Main(string[] args)
{
    Coffee cup = PourCoffee();
    Console.WriteLine("coffee is ready");

    var eggsTask = FryEggsAsync(2);
    var baconTask = FryBaconAsync(3);
    var toastTask = MakeToastWithButterAndJamAsync(2);

    var eggs = await eggsTask;
    Console.WriteLine("eggs are ready");

    var bacon = await baconTask;
    Console.WriteLine("bacon is ready");

    var toast = await toastTask;
    Console.WriteLine("toast is ready");

    Juice oj = PourOJ();
    Console.WriteLine("oj is ready");
    Console.WriteLine("Breakfast is ready!");
}
```

Essa alteração de código ilustra uma técnica importante para trabalhar com código assíncrono. Você compõe tarefas separando as operações em um novo método que retorna uma tarefa. Você pode escolher quando esperar por essa tarefa. Você pode iniciar outras tarefas simultaneamente.

Manipular exceções assíncronas

Até esse ponto, seu código pressupõe implicitamente que todas as tarefas sejam concluídas com êxito. Métodos assíncronos geram exceções, assim como seus equivalentes síncronos. As metas de suporte assíncrono para exceções e tratamento de erros são as mesmas do suporte assíncrono em geral. A melhor prática é escrever um código que seja lido como uma série de instruções síncronas. As tarefas geram exceções quando não podem ser concluídas com êxito. O código do cliente pode capturar essas exceções quando a expressão `await` é aplicada a uma tarefa iniciada.

No exemplo do café da manhã, suponha que a torradeira pegue fogo enquanto torra o pão. Você pode simular esse problema modificando o método `ToastBreadAsync` para corresponder ao seguinte código:

C#

```
private static async Task<Toast> ToastBreadAsync(int slices)
{
```

```
    for (int slice = 0; slice < slices; slice++)
    {
        Console.WriteLine("Putting a slice of bread in the toaster");
    }
    Console.WriteLine("Start toasting...");
    await Task.Delay(2000);
    Console.WriteLine("Fire! Toast is ruined!");
    throw new InvalidOperationException("The toaster is on fire");
    await Task.Delay(1000);
    Console.WriteLine("Remove toast from toaster");

    return new Toast();
}
```

> **Observação**
>
> Ao compilar esse código, você verá um aviso sobre código inacessível. Esse erro é por design. Depois que a torradeira pega fogo, as operações não prossseguem normalmente e o código retorna um erro.

Depois de fazer as alterações de código, execute o aplicativo e verifique a saída:

Console

```
Pouring coffee
Coffee is ready
Warming the egg pan...
putting 3 slices of bacon in the pan
Cooking first side of bacon...
Putting a slice of bread in the toaster
Putting a slice of bread in the toaster
Start toasting...
Fire! Toast is ruined!
Flipping a slice of bacon
Flipping a slice of bacon
Flipping a slice of bacon
Cooking the second side of bacon...
Cracking 2 eggs
Cooking the eggs ...
Put bacon on plate
Put eggs on plate
Eggs are ready
Bacon is ready
Unhandled exception. System.InvalidOperationException: The toaster is on
fire
   at AsyncBreakfast.Program.ToastBreadAsync(Int32 slices) in
Program.cs:line 65
   at AsyncBreakfast.Program.MakeToastWithButterAndJamAsync(Int32 number) in
Program.cs:line 36
```

```
   at AsyncBreakfast.Program.Main(String[] args) in Program.cs:line 24
   at AsyncBreakfast.Program.<Main>(String[] args)
```

Observe que algumas tarefas são concluídas entre o momento em que a torradeira pega fogo e o sistema observa a exceção. Quando uma tarefa que é executada de forma assíncrona gera uma exceção, ocorre uma **falha** nessa tarefa. O objeto `Task` contém a exceção gerada na propriedade Task.Exception. Tarefas com falha geram uma exceção quando a expressão `await` é aplicada à tarefa.

Há dois mecanismos importantes para entender sobre esse processo:

- Como uma exceção é armazenada em uma tarefa com falha
- Como uma exceção é descompactada e relançada quando o código aguarda (`await`) uma tarefa com falha

Quando o código em execução de forma assíncrona gera uma exceção, a exceção é armazenada no objeto `Task`. A propriedade Task.Exception é um objeto System.AggregateException, pois mais de uma exceção pode ser gerada durante o trabalho assíncrono. Qualquer exceção gerada é adicionada à coleção AggregateException.InnerExceptions. Se a propriedade `Exception` for nula, um novo objeto `AggregateException` será criado e a exceção gerada será o primeiro item da coleção.

O cenário mais comum para uma tarefa com falha é que a propriedade `Exception` contém exatamente uma exceção. Quando o seu código espera por uma tarefa com falha, ele relança a primeira exceção AggregateException.InnerExceptions na coleção. Esse resultado é o motivo pelo qual a saída do exemplo mostra um objeto System.InvalidOperationException em vez de um objeto `AggregateException`. Extrair a primeira exceção interna torna o trabalho com métodos assíncronos ser o mais semelhante possível ao trabalho com seus equivalentes síncronos. Você pode examinar a propriedade `Exception` em seu código quando seu cenário pode gerar várias exceções.

> **Dica**
>
> A prática recomendada é que quaisquer exceções de validação de argumento surjam *de forma síncrona* de métodos de retorno de tarefa. Para obter mais informações e exemplos, consulte **Exceções em métodos de retorno de tarefa**.

Antes de continuar para a próxima seção, comente as duas instruções a seguir em seu método `ToastBreadAsync`. Você não quer iniciar outro incêndio:

```C#
Console.WriteLine("Fire! Toast is ruined!");
throw new InvalidOperationException("The toaster is on fire");
```

Aplique expressões await às tarefas com eficiência

Você pode melhorar a série de expressões `await` no final do código anterior usando métodos da classe `Task`. Uma API é o método WhenAll, que retorna um objeto Task que é concluído quando todas as tarefas em sua lista de argumentos são concluídas. O código a seguir demonstra este método:

```C#
await Task.WhenAll(eggsTask, baconTask, toastTask);
Console.WriteLine("Eggs are ready");
Console.WriteLine("Bacon is ready");
Console.WriteLine("Toast is ready");
Console.WriteLine("Breakfast is ready!");
```

Outra opção é usar o método WhenAny, que retorna um objeto `Task<Task>` que é concluído quando qualquer um de seus argumentos é concluído. Você pode aguardar a tarefa retornada porque sabe que a tarefa foi concluída. O código a seguir mostra como você pode usar o método WhenAny para aguardar a primeira tarefa para concluir e, em seguida, processar seu resultado. Depois de processar o resultado da tarefa concluída, você remove a tarefa concluída da lista de tarefas passadas para o método `WhenAny`.

```C#
var breakfastTasks = new List<Task> { eggsTask, baconTask, toastTask };
while (breakfastTasks.Count > 0)
{
    Task finishedTask = await Task.WhenAny(breakfastTasks);
    if (finishedTask == eggsTask)
    {
        Console.WriteLine("Eggs are ready");
    }
    else if (finishedTask == baconTask)
    {
        Console.WriteLine("Bacon is ready");
    }
    else if (finishedTask == toastTask)
    {
        Console.WriteLine("Toast is ready");
```

```
    }
    await finishedTask;
    breakfastTasks.Remove(finishedTask);
}
```

Próximo ao final do snippet de código, observe a expressão `await finishedTask;`. A expressão `await Task.WhenAny` não aguarda a tarefa concluída, mas aguarda o objeto `Task` retornado pelo método `Task.WhenAny`. O resultado do método `Task.WhenAny` é a tarefa concluída (ou com falha). A melhor prática é aguardar a tarefa novamente, mesmo quando você souber que a tarefa está concluída. Dessa maneira, você pode recuperar o resultado da tarefa ou garantir que qualquer exceção que faça a tarefa falhar seja gerada.

Examinar o código final

Veja a aparência da versão final do código:

C#

```
using System;
using System.Collections.Generic;
using System.Threading.Tasks;

namespace AsyncBreakfast
{
    // These classes are intentionally empty for the purpose of this
example. They are simply marker classes for the purpose of demonstration,
contain no properties, and serve no other purpose.
    internal class Bacon { }
    internal class Coffee { }
    internal class Egg { }
    internal class Juice { }
    internal class Toast { }

    class Program
    {
```

```
        static async Task Main(string[] args)
        {
            Coffee cup = PourCoffee();
            Console.WriteLine("coffee is ready");

            var eggsTask = FryEggsAsync(2);
            var baconTask = FryBaconAsync(3);
            var toastTask = MakeToastWithButterAndJamAsync(2);

            var breakfastTasks = new List<Task> { eggsTask, baconTask,
toastTask };
            while (breakfastTasks.Count > 0)
            {
                Task finishedTask = await Task.WhenAny(breakfastTasks);
                if (finishedTask == eggsTask)
                {
                    Console.WriteLine("eggs are ready");
                }
                else if (finishedTask == baconTask)
                {
                    Console.WriteLine("bacon is ready");
                }
                else if (finishedTask == toastTask)
                {
                    Console.WriteLine("toast is ready");
                }
                await finishedTask;
                breakfastTasks.Remove(finishedTask);
            }

            Juice oj = PourOJ();
            Console.WriteLine("oj is ready");
            Console.WriteLine("Breakfast is ready!");
        }

        static async Task<Toast> MakeToastWithButterAndJamAsync(int number)
        {
            var toast = await ToastBreadAsync(number);
            ApplyButter(toast);
            ApplyJam(toast);

            return toast;
        }

        private static Juice PourOJ()
        {
            Console.WriteLine("Pouring orange juice");
            return new Juice();
        }

        private static void ApplyJam(Toast toast) =>
            Console.WriteLine("Putting jam on the toast");

        private static void ApplyButter(Toast toast) =>
            Console.WriteLine("Putting butter on the toast");
```

```
        private static async Task<Toast> ToastBreadAsync(int slices)
        {
            for (int slice = 0; slice < slices; slice++)
            {
                Console.WriteLine("Putting a slice of bread in the
toaster");
            }
            Console.WriteLine("Start toasting...");
            await Task.Delay(3000);
            Console.WriteLine("Remove toast from toaster");

            return new Toast();
        }

        private static async Task<Bacon> FryBaconAsync(int slices)
        {
            Console.WriteLine($"putting {slices} slices of bacon in the
pan");
            Console.WriteLine("cooking first side of bacon...");
            await Task.Delay(3000);
            for (int slice = 0; slice < slices; slice++)
            {
                Console.WriteLine("flipping a slice of bacon");
            }
            Console.WriteLine("cooking the second side of bacon...");
            await Task.Delay(3000);
            Console.WriteLine("Put bacon on plate");

            return new Bacon();
        }

        private static async Task<Egg> FryEggsAsync(int howMany)
        {
            Console.WriteLine("Warming the egg pan...");
            await Task.Delay(3000);
            Console.WriteLine($"cracking {howMany} eggs");
            Console.WriteLine("cooking the eggs ...");
            await Task.Delay(3000);
            Console.WriteLine("Put eggs on plate");

            return new Egg();
        }

        private static Coffee PourCoffee()
        {
            Console.WriteLine("Pouring coffee");
            return new Coffee();
        }
    }
}
```

O código conclui as tarefas assíncronas do café da manhã em cerca de 15 minutos. O tempo total é reduzido porque algumas tarefas são executadas simultaneamente. O código monitora simultaneamente várias tarefas e executa a ação somente conforme necessário.

O código final é assíncrono. Reflete com mais precisão como uma pessoa pode cozinhar o café da manhã. Compare o código final com o primeiro exemplo de código no artigo. As ações principais ainda estão claras lendo o código. Você pode ler o código final da mesma forma que leu a lista de instruções para fazer um café da manhã, conforme mostrado no início do artigo. Os recursos de idioma para as palavras-chave `async` e `await` fornecem a tradução que cada pessoa faz para seguir as instruções escritas: Inicie as tarefas como puder e não bloqueie enquanto aguarda a conclusão das tarefas.

Próxima etapa

Explorar cenários do mundo real para programas assíncronos

Cenários de programação assíncrona

18/06/2025

Se o código implementar cenários associados a E/S para dar suporte a solicitações de dados de rede, acesso ao banco de dados ou leitura/gravações do sistema de arquivos, a programação assíncrona será a melhor abordagem. Você também pode escrever código assíncrono para cenários associados à CPU, como cálculos caros.

O C# tem um modelo de programação assíncrona no nível da linguagem que permite que você escreva facilmente código assíncrono sem precisar fazer malabarismo com retornos de chamada ou em conformidade com uma biblioteca que dê suporte à assíncrona. O modelo segue o que é conhecido como TAP (padrão assíncrono baseado em tarefa).

Explorar o modelo de programação assíncrona

Os `Task` objetos e `Task<T>` representam o núcleo da programação assíncrona. Esses objetos são usados para modelar operações assíncronas por meio do suporte às palavras-chave `async` e `await`. Na maioria dos casos, o modelo é bastante simples para cenários associados a E/S e CPU. Dentro de um método `async`:

- **O código associado a E/S** inicia uma operação representada por um `Task` ou `Task<T>` objeto dentro do `async` método.
- O **código associado à CPU** inicia uma operação em um thread em segundo plano com o método Task.Run.

Em ambos os casos, um ativo `Task` representa uma operação assíncrona que pode não ser concluída.

É na palavra-chave `await` que a mágica acontece. Ele gera controle ao chamador do método que contém a `await` expressão e, por fim, permite que a interface do usuário seja responsiva ou um serviço seja elástico. Embora existam maneiras de abordar código assíncrono além do uso das expressões `async` e `await`, este artigo se concentra nos construtos ao nível da linguagem.

> ⓘ **Observação**
>
> Alguns exemplos apresentados neste artigo usam a System.Net.Http.HttpClient classe para baixar dados de um serviço Web. No código de exemplo, o `s_httpClient` objeto é um campo estático da classe de tipo `Program` :

```
private static readonly HttpClient s_httpClient = new();
```

Para obter mais informações, consulte o **código de exemplo completo** no final deste artigo.

Examinar conceitos subjacentes

Quando você implementa a programação assíncrona no código C#, o compilador transforma seu programa em um computador de estado. Esse constructo rastreia várias operações e estado em seu código, como gerar execução quando o código atinge uma `await` expressão e retomar a execução quando um trabalho em segundo plano é concluído.

Em termos de teoria da ciência da computação, a programação assíncrona é uma implementação do modelo Promise de assíncrona.

No modelo de programação assíncrona, há vários conceitos importantes para entender:

- Você pode usar código assíncrono para código associado a E/S e CPU, mas a implementação é diferente.
- O código assíncrono usa `Task<T>` e `Task` objetos como constructos para modelar o trabalho em execução em segundo plano.
- A `async` palavra-chave declara um método como um método assíncrono, que permite que você use a `await` palavra-chave no corpo do método.
- Quando você aplica a `await` palavra-chave, o código suspende o método de chamada e retorna o controle ao chamador até que a tarefa seja concluída.
- Você só pode usar a `await` expressão em um método assíncrono.

Exemplo vinculado à E/S: baixar dados de um serviço Web

Neste exemplo, quando o usuário seleciona um botão, o aplicativo baixa dados de um serviço Web. Você não deseja bloquear o thread da interface do usuário do aplicativo durante o processo de download. O código a seguir realiza esta tarefa:

C#

```
s_downloadButton.Clicked += async (o, e) =>
{
    // This line will yield control to the UI as the request
    // from the web service is happening.
    //
    // The UI thread is now free to perform other work.
    var stringData = await s_httpClient.GetStringAsync(URL);
```

```
    DoSomethingWithData(stringData);
};
```

O código expressa a intenção (baixar alguns dados de forma assíncrona) sem se prender à interação com objetos `Task`.

Exemplo limitado pela CPU: executar o cálculo do jogo

No exemplo a seguir, um jogo móvel causa danos a vários agentes na tela em resposta a um evento de botão. Executar o cálculo de danos pode ser caro. Executar o cálculo no thread da interface do usuário pode causar problemas de exibição e interação com a interface do usuário durante o cálculo.

A melhor maneira de lidar com a tarefa é começar um thread de segundo plano usando o método `Task.Run` para concluir o trabalho. A operação é realizada usando uma expressão `await`. A operação é retomada quando a tarefa é concluída. Essa abordagem permite que a interface do usuário seja executada sem problemas enquanto o trabalho é concluído em segundo plano.

C#

```
static DamageResult CalculateDamageDone()
{
    return new DamageResult()
    {
        // Code omitted:
        //
        // Does an expensive calculation and returns
        // the result of that calculation.
    };
}

s_calculateButton.Clicked += async (o, e) =>
{
    // This line will yield control to the UI while CalculateDamageDone()
    // performs its work. The UI thread is free to perform other work.
    var damageResult = await Task.Run(() => CalculateDamageDone());
    DisplayDamage(damageResult);
};
```

O código expressa claramente a intenção do evento de botão `Clicked` . Não é necessário o gerenciamento manual de uma thread em segundo plano e ele conclui a tarefa de maneira não bloqueante.

Reconhecer cenários associados à CPU e associados à E/S

Os exemplos anteriores demonstram como usar o modificador `async` e a expressão `await` para trabalhos limitados por E/S e CPU. Um exemplo para cada cenário mostra como o código é diferente com base no local em que a operação está associada. Para se preparar para sua implementação, você precisa entender como identificar quando uma operação está associada a E/S ou associada à CPU. Sua escolha de implementação pode afetar muito o desempenho do código e potencialmente levar a construções de uso indevido.

Há duas perguntas principais a serem abordadas antes de escrever qualquer código:

Expandir a tabela

Pergunta	Cenário	Implementação
O código deve aguardar um resultado ou ação, como dados de um banco de dados?	**Limite de E/S**	Use o modificador `async` e a expressão `await` *sem* o método `Task.Run`. Evite usar a Biblioteca Paralela de Tarefas.
O código deve executar uma computação cara?	**Limitado pela CPU**	Use o modificador `async` e expressão `await`, mas desencadeie o trabalho em outra linha de execução com o método `Task.Run`. Essa abordagem aborda preocupações com a capacidade de resposta da CPU. Se o trabalho for adequado para a simultaneidade e paralelismo, você também deverá considerar o uso da Biblioteca de paralelismo de tarefas.

Sempre meça a execução do código. Você pode descobrir que seu trabalho associado à CPU não é caro o suficiente em comparação com a sobrecarga de alternâncias de contexto ao fazer multithreading. Todas as opções têm compensações. Escolha a compensação correta para sua situação.

Explorar outros exemplos

Os exemplos nesta seção demonstram várias maneiras de escrever código assíncrono em C#. Eles abrangem alguns cenários que você pode encontrar.

Extrair dados de uma rede

O código a seguir baixa HTML de uma determinada URL e conta o número de vezes que a cadeia de caracteres ".NET" ocorre no HTML. O código usa ASP.NET para definir um método de controlador de API Web, que executa a tarefa e retorna a contagem.

> ⓘ **Observação**
>
> Se você pretende fazer análise de HTML no código de produção, não use expressões regulares. Use uma biblioteca de análise.

C#

```
[HttpGet, Route("DotNetCount")]
static public async Task<int> GetDotNetCount(string URL)
{
    // Suspends GetDotNetCount() to allow the caller (the web server)
    // to accept another request, rather than blocking on this one.
    var html = await s_httpClient.GetStringAsync(URL);
    return Regex.Matches(html, @"\.NET").Count;
}
```

Você pode escrever um código semelhante para um Aplicativo Universal do Windows e executar a tarefa de contagem depois que um botão for pressionado.

C#

```
private readonly HttpClient _httpClient = new HttpClient();

private async void OnSeeTheDotNetsButtonClick(object sender, RoutedEventArgs e)
{
    // Capture the task handle here so we can await the background task later.
    var getDotNetFoundationHtmlTask =
_httpClient.GetStringAsync("https://dotnetfoundation.org");

    // Any other work on the UI thread can be done here, such as enabling a
Progress Bar.
    // It's important to do the extra work here before the "await" call,
    // so 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 action 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;
}
```

Aguardar a conclusão de várias tarefas

Em alguns cenários, o código precisa recuperar várias partes de dados simultaneamente. As `Task` APIs fornecem métodos que permitem que você escreva código assíncrono que executa uma espera sem bloqueio em vários trabalhos em segundo plano:

- método Task.WhenAll
- método Task.WhenAny

O exemplo a seguir mostra como você pode capturar dados de `User` objeto para um conjunto de `userId` objetos.

C#

```
private static async Task<User> GetUserAsync(int userId)
{
    // Code omitted:
    //
    // Given a user Id {userId}, retrieves a User object corresponding
    // to the entry in the database with {userId} as its Id.

    return await Task.FromResult(new User() { id = userId });
}

private static async Task<IEnumerable<User>> GetUsersAsync(IEnumerable<int>
userIds)
{
    var getUserTasks = new List<Task<User>>();
    foreach (int userId in userIds)
    {
        getUserTasks.Add(GetUserAsync(userId));
    }

    return await Task.WhenAll(getUserTasks);
}
```

Você pode escrever esse código de forma mais sucinta usando LINQ:

C#

```
private static async Task<User[]> GetUsersAsyncByLINQ(IEnumerable<int> userIds)
{
    var getUserTasks = userIds.Select(id => GetUserAsync(id)).ToArray();
```

```
    return await Task.WhenAll(getUserTasks);
}
```

Embora você escreva menos código usando LINQ, tenha cuidado ao misturar LINQ com código assíncrono. O LINQ usa a execução adiada (ou lenta). Chamadas assíncronas não acontecem imediatamente como ocorrem em um loop `foreach`, a menos que você force a sequência gerada a iterar com uma chamada para o método `.ToList()` ou `.ToArray()`. Este exemplo usa o Enumerable.ToArray método para executar a consulta ansiosamente e armazenar os resultados em uma matriz. Essa abordagem força a instrução `id => GetUserAsync(id)` a executar e iniciar a tarefa.

Examinar considerações sobre programação assíncrona

Com a programação assíncrona, há vários detalhes a serem considerados que podem impedir um comportamento inesperado.

Usar await dentro do corpo do método async()

Ao usar o `async` modificador, você deve incluir uma ou mais `await` expressões no corpo do método. Se o compilador não encontrar uma `await` expressão, o método não produzirá. Embora o compilador gere um aviso, o código ainda é compilado e o compilador executa o método. O computador de estado gerado pelo compilador C# para o método assíncrono não realiza nada, portanto, todo o processo é altamente ineficiente.

Adicionar sufixo "Async" a nomes de método assíncrono

A convenção de estilo .NET é adicionar o sufixo "Assíncrono" a todos os nomes de método assíncrono. Essa abordagem ajuda a diferenciar mais facilmente entre métodos síncronos e assíncronos. Determinados métodos que não são explicitamente chamados pelo código (como manipuladores de eventos ou métodos de controlador web) não se aplicam necessariamente nesse cenário. Como esses itens não são explicitamente chamados pelo código, o uso de nomenclatura explícita não é tão importante.

Exibir "async void" somente de manipuladores de eventos

Os manipuladores de eventos devem declarar `void` tipos de retorno e não podem usar ou retornar `Task` e `Task<T>` objetos como outros métodos fazem. Ao escrever manipuladores de eventos assíncronos, você precisa usar o `async` modificador em um método que retorne `void`

para os manipuladores. Outras implementações de métodos de `async void` retorno não seguem o modelo TAP e podem apresentar desafios:

- Exceções geradas em um `async void` método não podem ser capturadas fora desse método
- `async void` métodos são difíceis de testar
- `async void` métodos podem causar efeitos colaterais negativos se quem os invoca não espera que eles sejam assíncronos

Tenha cuidado com lambdas assíncronas no LINQ

É importante ter cuidado ao implementar lambdas assíncronas em expressões LINQ. As expressões Lambda no LINQ usam a execução adiada, o que significa que o código pode ser executado em um momento inesperado. A introdução de tarefas de bloqueio nesse cenário pode facilmente resultar em um deadlock, se o código não for gravado corretamente. Além disso, o aninhamento de código assíncrono também pode dificultar a ponderação a respeito da execução do código. Técnicas Async e LINQ são poderosas, mas devem ser utilizadas juntas da forma mais cuidadosa e clara possível.

Suspender para tarefas de maneira não bloqueadora

Se o programa precisar do resultado de uma tarefa, escreva o código que implemente a `await` expressão de maneira não bloqueante. Bloquear o thread atual como um meio de aguardar de forma síncrona para que um item de `Task` seja concluído pode resultar em deadlocks e threads de contexto bloqueados. Essa abordagem de programação pode exigir tratamento de erros mais complexo. A tabela a seguir fornece diretrizes sobre como acessar resultados de tarefas de maneira não bloqueante.

Expandir a tabela

Cenário de tarefa	Código atual	Substituir por 'await'
Recuperar o resultado de uma tarefa em segundo plano	`Task.Wait` ou `Task.Result`	`await`
Continuar quando qualquer tarefa for concluída	`Task.WaitAny`	`await Task.WhenAny`
*Continuar quando **todas as** tarefas forem concluídas*	`Task.WaitAll`	`await Task.WhenAll`
Continuar após algum tempo	`Thread.Sleep`	`await Task.Delay`

Considerar o uso do tipo ValueTask

Quando um método assíncrono retorna um `Task` objeto, gargalos de desempenho podem ser introduzidos em determinados caminhos. Como `Task` é um tipo de referência, um `Task` objeto é alocado do heap. Se um método declarado com o `async` modificador retornar um resultado armazenado em cache ou for concluído de forma síncrona, as alocações extras poderão acumular custos de tempo significativos em seções críticas de desempenho do código. Esse cenário pode se tornar caro quando as alocações ocorrem em loops apertados. Para obter mais informações, consulte Tipos de retorno assíncronos generalizados.

Entender quando definir ConfigureAwait(false)

Os desenvolvedores geralmente perguntam sobre quando usar o Task.ConfigureAwait(Boolean) booliano. Essa API permite que uma `Task` instância configure o contexto para o computador de estado que implementa qualquer `await` expressão. Quando o booliano não está definido corretamente, o desempenho pode degradar ou deadlocks podem ocorrer. Para obter mais informações, consulte As perguntas frequentes sobre ConfigureAwait.

Gravar código sem estado

Evite escrever código que dependa do estado dos objetos globais ou da execução de determinados métodos. Em vez disso, depender apenas dos valores retornados dos métodos. Há muitos benefícios ao escrever código menos dependente de estado.

- Mais fácil de raciocinar sobre código
- Mais fácil de testar código
- Mais simples de misturar código assíncrono e síncrono
- Capaz de evitar condições de corrida no código
- Simples de coordenar o código assíncrono que depende de valores de retorno
- (Bônus) Funciona bem com a injeção de dependência no código

Uma meta recomendada é alcançar a Transparência referencial completa ou quase completa em seu código. Essa abordagem resulta em uma base de código previsível, testável e mantenedível.

Examinar o exemplo completo

O código a seguir representa o exemplo completo, que está disponível no arquivo *de exemplo Program.cs* .

C#

```
using System.Text.RegularExpressions;
using System.Windows;
using Microsoft.AspNetCore.Mvc;

class Button
{
    public Func<object, object, Task>? Clicked
    {
        get;
        internal set;
    }
}

class DamageResult
{
    public int Damage
    {
        get { return 0; }
    }
}

class User
{
    public bool isEnabled
    {
        get;
        set;
    }

    public int id
    {
        get;
        set;
    }
}

public class Program
{
    private static readonly Button s_downloadButton = new();
    private static readonly Button s_calculateButton = new();

    private static readonly HttpClient s_httpClient = new();

    private static readonly IEnumerable<string> s_urlList = new string[]
    {
            "https://learn.microsoft.com",
            "https://learn.microsoft.com/aspnet/core",
            "https://learn.microsoft.com/azure",
            "https://learn.microsoft.com/azure/devops",
            "https://learn.microsoft.com/dotnet",
            "https://learn.microsoft.com/dotnet/desktop/wpf/get-started/create-
app-visual-studio",
```

```
            "https://learn.microsoft.com/education",
            "https://learn.microsoft.com/shows/net-core-101/what-is-net",
            "https://learn.microsoft.com/enterprise-mobility-security",
            "https://learn.microsoft.com/gaming",
            "https://learn.microsoft.com/graph",
            "https://learn.microsoft.com/microsoft-365",
            "https://learn.microsoft.com/office",
            "https://learn.microsoft.com/powershell",
            "https://learn.microsoft.com/sql",
            "https://learn.microsoft.com/surface",
            "https://dotnetfoundation.org",
            "https://learn.microsoft.com/visualstudio",
            "https://learn.microsoft.com/windows",
            "https://learn.microsoft.com/maui"
        };

        private static void Calculate()
        {
            // <PerformGameCalculation>
            static DamageResult CalculateDamageDone()
            {
                return new DamageResult()
                {
                    // Code omitted:
                    //
                    // Does an expensive calculation and returns
                    // the result of that calculation.
                };
            }

            s_calculateButton.Clicked += async (o, e) =>
            {
                // This line will yield control to the UI while CalculateDamageDone()
                // performs its work. The UI thread is free to perform other work.
                var damageResult = await Task.Run(() => CalculateDamageDone());
                DisplayDamage(damageResult);
            };
            // </PerformGameCalculation>
        }

        private static void DisplayDamage(DamageResult damage)
        {
            Console.WriteLine(damage.Damage);
        }

        private static void Download(string URL)
        {
            // <UnblockingDownload>
            s_downloadButton.Clicked += async (o, e) =>
            {
                // This line will yield control to the UI as the request
                // from the web service is happening.
                //
                // The UI thread is now free to perform other work.
                var stringData = await s_httpClient.GetStringAsync(URL);
```

```
            DoSomethingWithData(stringData);
        };
        // </UnblockingDownload>
    }

    private static void DoSomethingWithData(object stringData)
    {
        Console.WriteLine($"Displaying data: {stringData}");
    }

    // <GetUsersForDataset>
    private static async Task<User> GetUserAsync(int userId)
    {
        // Code omitted:
        //
        // Given a user Id {userId}, retrieves a User object corresponding
        // to the entry in the database with {userId} as its Id.

        return await Task.FromResult(new User() { id = userId });
    }

    private static async Task<IEnumerable<User>> GetUsersAsync(IEnumerable<int>
userIds)
    {
        var getUserTasks = new List<Task<User>>();
        foreach (int userId in userIds)
        {
            getUserTasks.Add(GetUserAsync(userId));
        }

        return await Task.WhenAll(getUserTasks);
    }
    // </GetUsersForDataset>

    // <GetUsersForDatasetByLINQ>
    private static async Task<User[]> GetUsersAsyncByLINQ(IEnumerable<int>
userIds)
    {
        var getUserTasks = userIds.Select(id => GetUserAsync(id)).ToArray();
        return await Task.WhenAll(getUserTasks);
    }
    // </GetUsersForDatasetByLINQ>

    // <ExtractDataFromNetwork>
    [HttpGet, Route("DotNetCount")]
    static public async Task<int> GetDotNetCount(string URL)
    {
        // Suspends GetDotNetCount() to allow the caller (the web server)
        // to accept another request, rather than blocking on this one.
        var html = await s_httpClient.GetStringAsync(URL);
        return Regex.Matches(html, @"\.NET").Count;
    }
    // </ExtractDataFromNetwork>

    static async Task Main()
```

```
    {
        Console.WriteLine("Application started.");

        Console.WriteLine("Counting '.NET' phrase in websites...");
        int total = 0;
        foreach (string url in s_urlList)
        {
            var result = await GetDotNetCount(url);
            Console.WriteLine($"{url}: {result}");
            total += result;
        }
        Console.WriteLine("Total: " + total);

        Console.WriteLine("Retrieving User objects with list of IDs...");
        IEnumerable<int> ids = new int[] { 1, 2, 3, 4, 5, 6, 7, 8, 9, 0 };
        var users = await GetUsersAsync(ids);
        foreach (User? user in users)
        {
            Console.WriteLine($"{user.id}: isEnabled={user.isEnabled}");
        }

        Console.WriteLine("Application ending.");
    }
}

// Example output:
//
// Application started.
// Counting '.NET' phrase in websites...
// https://learn.microsoft.com: 0
// https://learn.microsoft.com/aspnet/core: 57
// https://learn.microsoft.com/azure: 1
// https://learn.microsoft.com/azure/devops: 2
// https://learn.microsoft.com/dotnet: 83
// https://learn.microsoft.com/dotnet/desktop/wpf/get-started/create-app-visual-
studio: 31
// https://learn.microsoft.com/education: 0
// https://learn.microsoft.com/shows/net-core-101/what-is-net: 42
// https://learn.microsoft.com/enterprise-mobility-security: 0
// https://learn.microsoft.com/gaming: 0
// https://learn.microsoft.com/graph: 0
// https://learn.microsoft.com/microsoft-365: 0
// https://learn.microsoft.com/office: 0
// https://learn.microsoft.com/powershell: 0
// https://learn.microsoft.com/sql: 0
// https://learn.microsoft.com/surface: 0
// https://dotnetfoundation.org: 16
// https://learn.microsoft.com/visualstudio: 0
// https://learn.microsoft.com/windows: 0
// https://learn.microsoft.com/maui: 6
// Total: 238
// Retrieving User objects with list of IDs...
// 1: isEnabled= False
// 2: isEnabled= False
// 3: isEnabled= False
```

```
// 4: isEnabled= False
// 5: isEnabled= False
// 6: isEnabled= False
// 7: isEnabled= False
// 8: isEnabled= False
// 9: isEnabled= False
// 0: isEnabled= False
// Application ending.
```

Links relacionados

- O modelo de programação assíncrona da Tarefa (C#)

Modelo de programação assíncrona baseado em tarefas

18/06/2025

Você pode evitar gargalos de desempenho e aprimorar a capacidade de resposta geral do aplicativo usando programação assíncrona. No entanto, as técnicas tradicionais para desenvolver aplicativos assíncronos podem ser complicadas, dificultando seu desenvolvimento, depuração e manutenção.

O C# dá suporte à abordagem simplificada, programação assíncrona, que aproveita o suporte assíncrono no runtime do .NET. O compilador faz o trabalho difícil que o desenvolvedor costumava fazer e seu aplicativo mantém uma estrutura lógica semelhante a um código síncrono. Como resultado, você obtém todas as vantagens da programação assíncrona com uma fração do esforço.

Este tópico fornece uma visão geral de quando e como usar programação assíncrona e inclui links para tópicos de suporte que contêm detalhes e exemplos.

Async melhora a capacidade de resposta

A assincronia é essencial para atividades que são potencialmente bloqueantes, como o acesso à Web. Às vezes, o acesso a um recurso da Web é lento ou atrasado. Se essa atividade for bloqueada em um processo síncrono, todo o aplicativo deverá aguardar. Em um processo assíncrono, o aplicativo pode continuar com outro trabalho que não depende do recurso da Web até que a tarefa potencialmente bloqueada seja concluída.

A tabela a seguir mostra áreas típicas em que a programação assíncrona melhora a capacidade de resposta. As APIs listadas do .NET e do Windows Runtime contêm métodos que dão suporte à programação assíncrona.

Expandir a tabela

Área do aplicativo	Tipos .NET com métodos assíncronos	Tipos do Windows Runtime com métodos assíncronos
Acesso à Web	HttpClient	Windows.Web.Http.HttpClient SyndicationClient
Trabalhando com arquivos	JsonSerializer StreamReader StreamWriter	StorageFile

Área do aplicativo	Tipos .NET com métodos assíncronos	Tipos do Windows Runtime com métodos assíncronos
	XmlReader XmlWriter	
Trabalhando com imagens		MediaCapture BitmapEncoder BitmapDecoder
Programação do WCF	Operações síncronas e assíncronas	

A assincronia é especialmente importante para aplicativos que acessam o thread de interface de usuário porque todas as atividades relacionadas à interface do usuário normalmente compartilham um único thread. Se qualquer processo for bloqueado em um aplicativo síncrono, todos serão bloqueados. Seu aplicativo para de responder e você pode concluir que ele falhou quando estiver apenas aguardando.

Quando você usa métodos assíncronos, o aplicativo continua respondendo à interface do usuário. Você pode redimensionar ou minimizar uma janela, por exemplo, ou fechar o aplicativo se não quiser esperar que ela seja concluída.

A abordagem baseada em assíncrona adiciona o equivalente a uma transmissão automática à lista de opções que você pode escolher ao projetar operações assíncronas. Ou seja, você obtém todos os benefícios da programação assíncrona tradicional, mas com muito menos esforço do desenvolvedor.

Métodos assíncronos são fáceis de escrever

As palavras-chave assíncronas e await em C# são o coração da programação assíncrona. Usando essas duas palavras-chave, você pode usar recursos no .NET Framework, no .NET Core ou no Windows Runtime para criar um método assíncrono quase tão facilmente quanto criar um método síncrono. Métodos assíncronos que você define usando a `async` palavra-chave são chamados *de métodos assíncronos*.

O exemplo a seguir mostra um método assíncrono. Quase tudo no código deve parecer familiar para você.

Você pode encontrar um exemplo completo do Windows Presentation Foundation (WPF) disponível para download em Programação Assíncrona com async e await em C#.

```
C#
```

```
public async Task<int> GetUrlContentLengthAsync()
{
    using var client = new HttpClient();

    Task<string> getStringTask =
        client.GetStringAsync("https://learn.microsoft.com/dotnet");

    DoIndependentWork();

    string contents = await getStringTask;

    return contents.Length;
}

void DoIndependentWork()
{
    Console.WriteLine("Working...");
}
```

Você pode aprender várias práticas com base no exemplo anterior. Comece com a assinatura do método. Ele inclui o `async` modificador. O tipo de retorno é `Task<int>` (consulte a seção "Tipos de Retorno" para obter mais opções). O nome do método termina em `Async`. No corpo do método, `GetStringAsync` retorna um `Task<string>`. Isso significa que, quando você executar `await` em uma tarefa, obterá uma `string` (`contents`). Antes de aguardar a tarefa, você poderá fazer um trabalho que não dependa da `string` em `GetStringAsync`.

Preste muita atenção no operador `await`. Ele suspende `GetUrlContentLengthAsync`:

- `GetUrlContentLengthAsync` não pode continuar até que `getStringTask` seja concluído.
- Enquanto isso, o controle é retornado ao chamador de `GetUrlContentLengthAsync`.
- O controle será retomado aqui quando a `getStringTask` for concluída.
- `await` O operador então recupera o `string` resultado de `getStringTask`.

A instrução return especifica um resultado inteiro. Os métodos que estão aguardando `GetUrlContentLengthAsync` recuperar o valor de comprimento.

Se `GetUrlContentLengthAsync` não tiver nenhum trabalho que possa fazer entre chamar `GetStringAsync` e aguardar sua conclusão, você poderá simplificar seu código chamando e aguardando na instrução única a seguir.

C#

```
string contents = await
client.GetStringAsync("https://learn.microsoft.com/dotnet");
```

As seguintes características resumem o que torna o exemplo anterior um método assíncrono:

- A assinatura do método inclui um `async` modificador.
- O nome de um método assíncrono, por convenção, termina com um sufixo "Assíncrono".
- O tipo de retorno é um dos seguintes tipos:
  - Task<TResult> se o método tiver uma instrução return na qual o operando tem o tipo `TResult`.
  - Task se o método não tiver nenhuma instrução de retorno ou tiver uma instrução de retorno sem operando.
  - `void` se você estiver escrevendo um manipulador de eventos assíncrono.
  - Qualquer outro tipo que tenha um método `GetAwaiter`.

  Para obter mais informações, consulte a seção Tipos de retorno e parâmetros .
- O método geralmente inclui pelo menos uma `await` expressão, que marca um ponto em que o método não pode continuar até que a operação assíncrona aguardada seja concluída. Enquanto isso, o método é suspenso e o controle retorna ao chamador do método. A próxima seção deste tópico ilustra o que acontece no ponto de suspensão.

Em métodos assíncronos, você usa as palavras-chave e os tipos fornecidos para indicar o que deseja fazer e o compilador faz o restante, incluindo manter o controle do que deve acontecer quando o controle retorna a um ponto de espera em um método suspenso. Alguns processos rotineiros, como loops e tratamento de exceções, podem ser difíceis de lidar no código assíncrono tradicional. Em um método assíncrono, você escreve esses elementos da mesma forma que faria em uma solução síncrona e o problema é resolvido.

Para obter mais informações sobre assíncrona em versões anteriores do .NET Framework, consulte TPL e programação assíncrona tradicional do .NET Framework.

O que acontece em um método assíncrono

O mais importante a entender na programação assíncrona é como o fluxo de controle se move de método para método. O diagrama a seguir leva você ao processo:

Os números no diagrama correspondem às etapas a seguir, iniciadas quando um método de chamada chama o método assíncrono.

1. Um método de chamada chama e aguarda o método assíncrono `GetUrlContentLengthAsync`.

2. `GetUrlContentLengthAsync` cria uma HttpClient instância e chama o GetStringAsync método assíncrono para baixar o conteúdo de um site como uma cadeia de caracteres.

3. Algo acontece em `GetStringAsync` que suspende seu progresso. Talvez ele deva aguardar o download de um site ou alguma outra atividade causadora de bloqueio. Para evitar o bloqueio de recursos, `GetStringAsync` gera controle ao chamador. `GetUrlContentLengthAsync`

   `GetStringAsync` retorna um Task<TResult>, onde `TResult` é uma cadeia de caracteres e `GetUrlContentLengthAsync` atribui a tarefa à `getStringTask` variável. A tarefa representa o processo contínuo para a chamada a `GetStringAsync`, com um compromisso de produzir um valor de cadeia de caracteres real quando o trabalho estiver concluído.

4. Como `getStringTask` ainda não foi aguardado, `GetUrlContentLengthAsync` pode continuar com outros trabalhos que não dependem do resultado final de `GetStringAsync`. Esse trabalho é representado por uma chamada para o método `DoIndependentWork` síncrono.

5. `DoIndependentWork` é um método síncrono que faz seu trabalho e retorna ao chamador.

6. `GetUrlContentLengthAsync` está sem trabalho que ele possa executar sem um resultado de `getStringTask`. `GetUrlContentLengthAsync` Em seguida, deseja calcular e retornar o comprimento da cadeia de caracteres baixada, mas o método não pode calcular esse valor até que o método tenha a cadeia de caracteres.

   Portanto, `GetUrlContentLengthAsync` usa um operador await para suspender seu progresso e gerar controle ao método chamado `GetUrlContentLengthAsync`. `GetUrlContentLengthAsync` retorna um `Task<int>` para o chamador. A tarefa representa uma promessa de produzir um resultado inteiro que é o comprimento da cadeia de caracteres baixada.

   > ⓘ **Observação**
   >
   > Se `GetStringAsync` (e, portanto, `getStringTask`) for concluído antes que `GetUrlContentLengthAsync` o aguarde, o controle permanecerá em `GetUrlContentLengthAsync`. A despesa de suspender e depois voltar a `GetUrlContentLengthAsync` seria desperdiçada se o processo assíncrono `getStringTask` chamado já tiver sido concluído e `GetUrlContentLengthAsync` não precisar aguardar o resultado final.

   Dentro do método de chamada, o padrão de processamento continua. O chamador pode realizar outro trabalho que não dependa do resultado de `GetUrlContentLengthAsync` antes de aguardar esse resultado, ou o chamador pode aguardar imediatamente. O método de chamada está aguardando `GetUrlContentLengthAsync` e `GetUrlContentLengthAsync` está aguardando `GetStringAsync`.

7. `GetStringAsync` completa e produz um resultado de cadeia de caracteres. O resultado da cadeia de caracteres não é retornado pela chamada `GetStringAsync` da maneira que você pode esperar. (Lembre-se de que o método já retornou uma tarefa na etapa 3.) Em vez disso, o resultado da cadeia de caracteres é armazenado na tarefa que representa a conclusão do método. `getStringTask` O operador await recupera o resultado de `getStringTask`. A instrução de atribuição atribui o resultado recuperado a `contents`.

8. Quando `GetUrlContentLengthAsync` tem o resultado da cadeia de caracteres, o método pode calcular o comprimento da cadeia de caracteres. Em seguida, o trabalho de `GetUrlContentLengthAsync` também é concluído e o manipulador de eventos de espera poderá retomar. No exemplo completo no final do tópico, é possível confirmar que o manipulador de eventos recuperou e imprimiu o valor do comprimento do resultado. Se

você não estiver familiarizado com a programação assíncrona, leve um minuto para considerar a diferença entre o comportamento síncrono e assíncrono. Um método síncrono retorna quando seu trabalho é concluído (etapa 5), mas um método assíncrono retorna um valor de tarefa quando seu trabalho é suspenso (etapas 3 e 6). Quando o método assíncrono eventualmente conclui seu trabalho, a tarefa é marcada como concluída e o resultado, se houver, é armazenado na tarefa.

Métodos assíncronos da API

Você pode estar se perguntando onde encontrar métodos como `GetStringAsync` que dão suporte à programação assíncrona. O .NET Framework 4.5 ou superior e o .NET Core contêm muitos membros que trabalham com `async` e `await`. Você pode reconhecê-los pelo sufixo "Assíncrono" acrescentado ao nome do membro e pelo tipo de retorno de Task ou Task<TResult>. Por exemplo, a `System.IO.Stream` classe contém métodos como CopyToAsync, ReadAsynce WriteAsync ao lado dos métodos CopyTosíncronos, Reade Write.

O Windows Runtime também contém muitos métodos que você pode usar com `async` e `await` em aplicativos do Windows. Para obter mais informações, veja Threading e programação assíncrona para o desenvolvimento da UWP e Programação assíncrona (aplicativos da Windows Store) e Início Rápido: chamando APIs assíncronas em C# ou Visual Basic se você usa versões anteriores do Windows Runtime.

Fios

Os métodos assíncronos destinam-se a operações sem bloqueio. Uma `await` expressão em um método assíncrono não bloqueia o thread atual enquanto a tarefa aguardada está em execução. Em vez disso, a expressão inscreve o restante do método como uma continuação e retorna o controle ao chamador do método assíncrono.

As palavras-chave `async` e `await` não fazem com que threads adicionais sejam criadas. Os métodos assíncronos não exigem multithreading porque um método assíncrono não é executado em seu próprio thread. O método é executado no contexto de sincronização atual e usa o tempo no thread somente quando o método está ativo. Você pode usar Task.Run para mover o trabalho associado à CPU para um thread em segundo plano, mas um thread em segundo plano não ajuda com um processo que está apenas aguardando que os resultados fiquem disponíveis.

A abordagem baseada em async para programação assíncrona é preferível às abordagens existentes em quase todos os casos. Essa abordagem é especialmente mais eficiente do que a classe BackgroundWorker para operações de entrada e saída, porque o código é mais simples

e você não precisa se proteger contra condições de corrida. Em combinação com o método Task.Run, a programação assíncrona é melhor do que BackgroundWorker para operações associadas à CPU, pois a programação assíncrona separa os detalhes de coordenação da execução do código do trabalho que `Task.Run` transfere para o pool de threads.

async e await

Se você especificar que um método é um método assíncrono usando o modificador assíncrono , habilite os dois recursos a seguir.

- O método assíncrono marcado pode usar await para designar pontos de suspensão. O `await` operador informa ao compilador que o método assíncrono não pode continuar além desse ponto até que o processo assíncrono aguardado seja concluído. Enquanto isso, o controle retorna para o chamador do método assíncrono.

  A suspensão de um método assíncrono em uma `await` expressão não constitui uma saída do método e `finally` os blocos não são executados.

- O método assíncrono marcado pode ele próprio ser aguardado por métodos que o chamam.

Um método assíncrono normalmente contém uma ou mais ocorrências de um `await` operador, mas a ausência de `await` expressões não causa um erro do compilador. Se um método assíncrono não usar um `await` operador para marcar um ponto de suspensão, o método será executado como um método síncrono, apesar do `async` modificador. O compilador emite um aviso para esses métodos.

`async` e `await` são palavras-chave contextuais. Para obter mais informações e exemplos, consulte os seguintes tópicos:

- assíncrono
- esperar

Tipos e parâmetros de retorno

Um método assíncrono normalmente retorna um Task ou um Task<TResult>. Dentro de um método assíncrono, um `await` operador é aplicado a uma tarefa retornada de uma chamada para outro método assíncrono.

Especifique Task<TResult> como o tipo de retorno se o método contiver uma return instrução que especifica um operando do tipo `TResult`.

Você usará Task como o tipo de retorno se o método não tiver nenhuma instrução de retorno ou tiver uma instrução de retorno que não retorne um operando.

Você também pode especificar qualquer outro tipo de retorno, desde que o tipo inclua um `GetAwaiter` método. ValueTask<TResult> é um exemplo desse tipo. Ele está disponível no pacote NuGet System.Threading.Tasks.Extension .

O exemplo a seguir mostra como você declara e chama um método que retorna um Task<TResult> ou um Task:

C#

```
async Task<int> GetTaskOfTResultAsync()
{
    int hours = 0;
    await Task.Delay(0);

    return hours;
}


Task<int> returnedTaskTResult = GetTaskOfTResultAsync();
int intResult = await returnedTaskTResult;
// Single line
// int intResult = await GetTaskOfTResultAsync();

async Task GetTaskAsync()
{
    await Task.Delay(0);
    // No return statement needed
}

Task returnedTask = GetTaskAsync();
await returnedTask;
// Single line
await GetTaskAsync();
```

Cada tarefa retornada representa o trabalho em andamento. Uma tarefa encapsula informações sobre o estado do processo assíncrono e, eventualmente, o resultado final do processo ou a exceção que o processo gera se não for bem-sucedido.

Um método assíncrono também pode ter um `void` tipo de retorno. Esse tipo de retorno é usado principalmente para definir manipuladores de eventos, em que um `void` tipo de retorno é necessário. Manipuladores de eventos assíncronos geralmente servem como ponto de partida para programas assíncronos.

Um método assíncrono que tem o tipo de retorno `void` não pode ser esperado, e o chamador de um método que retorna nulo não pode capturar nenhuma exceção lançada pelo método.

Um método assíncrono não pode declarar parâmetros de entrada, ref ou saída , mas o método pode chamar métodos que têm esses parâmetros. Da mesma forma, um método assíncrono não pode retornar um valor por referência, embora possa chamar métodos que retornam valores por referência.

Para obter mais informações e exemplos, consulte tipos de retorno assíncronos (C#).

As APIs assíncronas na programação do Windows Runtime têm um dos seguintes tipos de retorno, semelhantes a tarefas:

- IAsyncOperation<TResult>, que corresponde a Task<TResult>
- IAsyncAction, que corresponde a Task
- IAsyncActionWithProgress<TProgress>
- IAsyncOperationWithProgress<TResult,TProgress>

Convenção de nomenclatura

Por convenção, os métodos que retornam tipos normalmente aguardados (por exemplo, `Task`, , `Task<T>` `ValueTask`,) `ValueTask<T>` devem ter nomes que terminam com "Async". Os métodos que iniciam uma operação assíncrona, mas não retornam um tipo aguardável não devem ter nomes que terminam com "Async", mas podem começar com "Begin", "Start" ou algum outro verbo que indique que esse método não retorna nem gera o resultado da operação.

Você pode ignorar a convenção em que um evento, uma classe base ou um contrato de interface sugere um nome diferente. Por exemplo, você não deve renomear manipuladores de eventos comuns, como `OnButtonClick`.

Artigos relacionados (Visual Studio)

Expandir a tabela

Título	Descrição
Como fazer várias solicitações da Web em paralelo usando assíncrono e await (C#)	Demonstra como iniciar várias tarefas ao mesmo tempo.
Tipos de retorno assíncronos (C#)	Ilustra os tipos que os métodos assíncronos podem retornar e explica quando cada tipo é apropriado.
Cancelar tarefas usando um token de cancelamento como mecanismo de sinalização.	Mostra como adicionar a seguinte funcionalidade à sua solução assíncrona: - Cancelar uma lista de tarefas (C#) - Cancelar tarefas após um período de tempo (C#)

Título	Descrição
	- Processar tarefas assíncronas conforme elas forem concluídas (C#)
Usando assíncrono para acesso a arquivos (C#)	Lista e demonstra os benefícios de usar assíncrono e aguardar para acessar arquivos.
Padrão assíncrono baseado em tarefa (TAP)	Descreve um padrão assíncrono, o padrão é baseado nos tipos Task e Task<TResult>.
Vídeos assíncronos no Canal 9	Fornece links para uma variedade de vídeos sobre programação assíncrona.

Consulte também

- Programação assíncrona com async e await
- assíncrono
- esperar

Tipos de retorno assíncronos (C#)

Artigo • 06/01/2025

Os métodos assíncronos podem ter os seguintes tipos de retorno:

- Task, para um método assíncrono que executa uma operação, mas não retorna nenhum valor.
- Task<TResult>, para um método assíncrono que retorna um valor.
- `void`, para um manipulador de eventos.
- Qualquer tipo que tenha um método `GetAwaiter` acessível. O objeto retornado pelo método `GetAwaiter` deve implementar a interface System.Runtime.CompilerServices.ICriticalNotifyCompletion.
- IAsyncEnumerable<T>, para um método assíncrono que retorna um *fluxo assíncrono*.

Para obter mais informações sobre os métodos assíncronos, confira Programação assíncrona com async e await (C#).

Também existem vários outros tipos específicos para cargas de trabalho do Windows:

- DispatcherOperation, para operações assíncronas limitadas ao Windows.
- IAsyncAction, para ações assíncronas em aplicativos da Plataforma Universal do Windows (UWP) que não retornam um valor.
- IAsyncActionWithProgress<TProgress>, para ações assíncronas em aplicativos UWP que relatam o progresso, mas não retornam um valor.
- IAsyncOperation<TResult>, para operações assíncronas em aplicativos UWP que retornam um valor.
- IAsyncOperationWithProgress<TResult,TProgress>, para operações assíncronas em aplicativos UWP que relatam o progresso e retornam um valor.

Tipo de retorno da tarefa

Métodos assíncronos que não contêm uma instrução `return` ou que contêm uma instrução `return` que não retorna um operando geralmente têm um tipo de retorno de Task. Esses métodos retornam `void` se forem executados de forma síncrona. Se você usar um tipo de retorno Task para um método assíncrono, um método de chamada poderá usar um operador `await` para suspender a conclusão do chamador até que o método assíncrono chamado seja concluído.

No exemplo a seguir, o método `WaitAndApologizeAsync` não contém uma instrução `return`, portanto, o método retorna um objeto Task. Retornar `Task` habilita a espera por `WaitAndApologizeAsync`. O tipo Task não inclui uma propriedade `Result` porque não tem nenhum valor retornado.

C#

```
public static async Task DisplayCurrentInfoAsync()
{
    await WaitAndApologizeAsync();

    Console.WriteLine($"Today is {DateTime.Now:D}");
    Console.WriteLine($"The current time is {DateTime.Now.TimeOfDay:t}");
    Console.WriteLine("The current temperature is 76 degrees.");
}

static async Task WaitAndApologizeAsync()
{
    await Task.Delay(2000);

    Console.WriteLine("Sorry for the delay...\n");
}
// Example output:
//    Sorry for the delay...
//
// Today is Monday, August 17, 2020
// The current time is 12:59:24.2183304
// The current temperature is 76 degrees.
```

O `WaitAndApologizeAsync` é aguardado, usando uma instrução await, em vez de uma expressão await, semelhante à instrução de chamada a um método síncrono de retorno void. A aplicação de um operador await nesse caso não produz um valor. Quando o operando à direita de um `await` é um Task<TResult>, a expressão `await` produz um resultado de `T`. Quando o operando direito de `await` é Task, `await` e o operando dele são uma instrução.

Você pode separar a chamada a `WaitAndApologizeAsync` da aplicação de um operador await, como mostrado no código a seguir. No entanto, lembre-se que uma `Task` não tem uma propriedade `Result` e que nenhum valor será produzido quando um operador await for aplicado a uma `Task`.

O código a seguir separa a chamada ao método `WaitAndApologizeAsync` de aguardar a tarefa retornada pelo método.

C#

```
Task waitAndApologizeTask = WaitAndApologizeAsync();

string output =
    $"Today is {DateTime.Now:D}\n" +
    $"The current time is {DateTime.Now.TimeOfDay:t}\n" +
    "The current temperature is 76 degrees.\n";

await waitAndApologizeTask;
Console.WriteLine(output);
```

Tipo de retorno Task<TResult>

O tipo de retorno Task<TResult> é usado para um método assíncrono que contém uma instrução return em que o operando é `TResult`.

No exemplo a seguir, o método `GetLeisureHoursAsync` contém uma instrução `return` que retorna um inteiro. A declaração do método deve especificar um tipo de retorno de `Task<int>`. O método assíncrono FromResult é um espaço reservado para uma operação que retorna DayOfWeek.

C#

```
public static async Task ShowTodaysInfoAsync()
{
    string message =
        $"Today is {DateTime.Today:D}\n" +
        "Today's hours of leisure: " +
        $"{await GetLeisureHoursAsync()}";

    Console.WriteLine(message);
}

static async Task<int> GetLeisureHoursAsync()
{
    DayOfWeek today = await Task.FromResult(DateTime.Now.DayOfWeek);

    int leisureHours =
        today is DayOfWeek.Saturday || today is DayOfWeek.Sunday
        ? 16 : 5;

    return leisureHours;
}
// Example output:
//    Today is Wednesday, May 24, 2017
//    Today's hours of leisure: 5
```

Quando `GetLeisureHoursAsync` é chamado de dentro de uma expressão await no método `ShowTodaysInfo`, a expressão await recupera o valor inteiro (o valor de `leisureHours`) armazenado na tarefa retornada pelo método `GetLeisureHours`. Para obter mais informações sobre expressões await, consulte await.

Você pode entender melhor como `await` recupera o resultado de uma `Task<T>` separando a chamada para `GetLeisureHoursAsync` do aplicativo de `await`, como mostra o código a seguir. Uma chamada ao método `GetLeisureHoursAsync` que não é aguardada imediatamente, retorna um `Task<int>`, como você esperaria da declaração do método. A tarefa é atribuída à variável `getLeisureHoursTask` no exemplo. Como `getLeisureHoursTask` é um Task<TResult>, ele contém uma propriedade Result do tipo `TResult`. Nesse caso, `TResult` representa um tipo inteiro. Quando `await` é aplicado à `getLeisureHoursTask`, a expressão await é avaliada como o conteúdo da propriedade Result de `getLeisureHoursTask`. O valor é atribuído à variável `ret`.

> ⓘ **Importante**
>
> A propriedade **Result** é uma propriedade de bloqueio. Se você tentar acessá-la antes que sua tarefa seja concluída, o thread atualmente ativo será bloqueado até que a tarefa seja concluída e o valor esteja disponível. Na maioria dos casos, você deve acessar o valor usando `await` em vez de acessar a propriedade diretamente.
>
> O exemplo anterior recuperou o valor da propriedade **Result** para bloquear o thread principal para que o método `Main` pudesse imprimir o `message` no console antes do fim do aplicativo.

C#

```
var getLeisureHoursTask = GetLeisureHoursAsync();

string message =
    $"Today is {DateTime.Today:D}\n" +
    "Today's hours of leisure: " +
    $"{await getLeisureHoursTask}";

Console.WriteLine(message);
```

Tipo de retorno nulo

Você usa o tipo de retorno `void` em manipuladores de eventos assíncronos, que exigem um tipo de retorno `void`. Para métodos que não sejam manipuladores de eventos que

não retornam um valor, você deve retornar um Task em vez disso, porque um método assíncrono que retorna `void` não pode ser aguardado. Qualquer chamador desse método deve continuar a conclusão sem esperar que o método assíncrono chamado seja concluído. O chamador deve ser independente de quaisquer valores ou exceções que o método assíncrono gere.

O chamador de um método assíncrono de retorno nulo não pode capturar exceções geradas por meio do método. Essas exceções sem tratamento provavelmente causarão falha no aplicativo. Se um método que retorna um Task ou Task<TResult> gera uma exceção, a exceção é armazenada na tarefa retornada. A exceção é gerada novamente quando a tarefa é aguardada. Verifique se qualquer método assíncrono que possa produzir uma exceção tem um tipo de retorno de Task ou Task<TResult> e se as chamadas para o método são aguardadas.

O exemplo a seguir mostra o comportamento de um manipulador de eventos assíncrono. No código de exemplo, um manipulador de eventos assíncrono deve informar o thread principal quando ele for concluído. Em seguida, o thread principal pode aguardar um manipulador de eventos assíncronos ser concluído antes de sair do programa.

C#

```
public class NaiveButton
{
    public event EventHandler? Clicked;

    public void Click()
    {
        Console.WriteLine("Somebody has clicked a button. Let's raise the
event...");
        Clicked?.Invoke(this, EventArgs.Empty);
        Console.WriteLine("All listeners are notified.");
    }
}

public class AsyncVoidExample
{
    static readonly TaskCompletionSource<bool> s_tcs = new
TaskCompletionSource<bool>();

    public static async Task MultipleEventHandlersAsync()
    {
        Task<bool> secondHandlerFinished = s_tcs.Task;

        var button = new NaiveButton();

        button.Clicked += OnButtonClicked1;
        button.Clicked += OnButtonClicked2Async;
        button.Clicked += OnButtonClicked3;
```

```
        Console.WriteLine("Before button.Click() is called...");
        button.Click();
        Console.WriteLine("After button.Click() is called...");

        await secondHandlerFinished;
    }

    private static void OnButtonClicked1(object? sender, EventArgs e)
    {
        Console.WriteLine("   Handler 1 is starting...");
        Task.Delay(100).Wait();
        Console.WriteLine("   Handler 1 is done.");
    }

    private static async void OnButtonClicked2Async(object? sender,
EventArgs e)
    {
        Console.WriteLine("   Handler 2 is starting...");
        Task.Delay(100).Wait();
        Console.WriteLine("   Handler 2 is about to go async...");
        await Task.Delay(500);
        Console.WriteLine("   Handler 2 is done.");
        s_tcs.SetResult(true);
    }

    private static void OnButtonClicked3(object? sender, EventArgs e)
    {
        Console.WriteLine("   Handler 3 is starting...");
        Task.Delay(100).Wait();
        Console.WriteLine("   Handler 3 is done.");
    }
}
// Example output:
//
// Before button.Click() is called...
// Somebody has clicked a button. Let's raise the event...
//    Handler 1 is starting...
//    Handler 1 is done.
//    Handler 2 is starting...
//    Handler 2 is about to go async...
//    Handler 3 is starting...
//    Handler 3 is done.
// All listeners are notified.
// After button.Click() is called...
//    Handler 2 is done.
```

Tipos de retorno assíncronos generalizados e ValueTask<TResult>

Um método assíncrono pode retornar qualquer tipo que tenha um método `GetAwaiter` acessível que retorne uma instância de um *tipo de aguardador*. Além disso, o tipo retornado do método `GetAwaiter` deve ter o atributo System.Runtime.CompilerServices.AsyncMethodBuilderAttribute. Saiba mais no artigo sobre Atributos lidos pelo compilador ou na especificação de C# para o Padrão de construtor de tipo de tarefa.

Esse recurso é o complemento para expressões aguardáveis, que descreve os requisitos para o operando `await`. Tipos de retorno assíncronos generalizados permitem que o compilador gere métodos `async` que retornam tipos diferentes. Tipos de retorno assíncronos generalizados habilitaram melhorias de desempenho nas bibliotecas do .NET. Como Task e Task<TResult> são tipos de referência, a alocação de memória em caminhos críticos de desempenho, especialmente quando ocorrem alocações em loops apertados, pode afetar negativamente o desempenho. O suporte para tipos de retorno generalizados significa que você pode retornar um tipo de valor leve em vez de um tipo de referência para evitar mais alocações de memória.

O .NET fornece a estrutura System.Threading.Tasks.ValueTask<TResult> como uma implementação leve de um valor generalizado de retorno de tarefa. O exemplo a seguir usa a estrutura ValueTask<TResult> para recuperar o valor de dois rolos de dados.

C#

```
class Program
{
    static readonly Random s_rnd = new Random();

    static async Task Main() =>
        Console.WriteLine($"You rolled {await GetDiceRollAsync()}");

    static async ValueTask<int> GetDiceRollAsync()
    {
        Console.WriteLine("Shaking dice...");

        int roll1 = await RollAsync();
        int roll2 = await RollAsync();

        return roll1 + roll2;
    }

    static async ValueTask<int> RollAsync()
    {
        await Task.Delay(500);

        int diceRoll = s_rnd.Next(1, 7);
        return diceRoll;
    }
}
```

```
// Example output:
//    Shaking dice...
//    You rolled 8
```

Escrever um tipo de retorno assíncrono generalizado é um cenário avançado e é direcionado para uso em ambientes especializados. Considere usar os tipos `Task`, `Task<T>` e `ValueTask<T>`, que abrangem a maioria dos cenários de código assíncrono.

Você pode aplicar o atributo `AsyncMethodBuilder` a um método assíncrono (em vez da declaração de tipo de retorno assíncrono) para substituir o construtor daquele tipo. Normalmente, você aplicaria esse atributo para usar um construtor diferente fornecido no runtime do .NET.

Fluxos assíncronos com IAsyncEnumerable<T>

Um método assíncrono pode retornar um *fluxo assíncrono*, representado por IAsyncEnumerable<T>. Um fluxo assíncrono fornece uma maneira de enumerar itens lidos de um fluxo quando os elementos são gerados em partes com chamadas assíncronas repetidas. O exemplo a seguir mostra um método assíncrono que gera um fluxo assíncrono:

C#

```
static async IAsyncEnumerable<string> ReadWordsFromStreamAsync()
{
    string data =
        @"This is a line of text.
              Here is the second line of text.
              And there is one more for good measure.
              Wait, that was the penultimate line.";

    using var readStream = new StringReader(data);

    string? line = await readStream.ReadLineAsync();
    while (line != null)
    {
        foreach (string word in line.Split(' ',
StringSplitOptions.RemoveEmptyEntries))
        {
            yield return word;
        }

        line = await readStream.ReadLineAsync();
    }
}
```

O exemplo anterior lê linhas de uma cadeia de caracteres de forma assíncrona. Depois que cada linha é lida, o código enumera cada palavra na cadeia de caracteres. Os chamadores enumerariam cada palavra usando a instrução `await foreach`. O método aguarda quando precisa ler assíncronamente a próxima linha da cadeia de caracteres de origem.

Consulte também

- FromResult
- Processar tarefas assíncronas conforme elas são concluídas
- Programação assíncrona com async e await (C#)
- async
- await

Processar tarefas assíncronas conforme elas são concluídas (C#)

Artigo • 23/05/2023

Usando Task.WhenAny, você pode iniciar várias tarefas ao mesmo tempo e processá-las individualmente conforme elas forem concluídas, em vez de processá-las na ordem em que foram iniciadas.

O exemplo a seguir usa uma consulta para criar uma coleção de tarefas. Cada tarefa baixa o conteúdo de um site especificado. Em cada iteração de um loop "while", uma chamada esperada para WhenAny retorna a tarefa na coleção de tarefas que concluir o download primeiro. Essa tarefa é removida da coleção e processada. O loop é repetido até que a coleção não contenha mais tarefas.

Pré-requisitos

Você pode seguir este tutorial usando uma das seguintes opções:

- Visual Studio 2022 com a carga de trabalho **Desenvolvimento de área de trabalho do .NET** instalada. O SDK do .NET é instalado automaticamente quando você seleciona essa carga de trabalho.
- O SDK do .NET com um editor de código de sua escolha, como Visual Studio Code.

Criar aplicativo de exemplo

Criar um novo aplicativo de console do .NET Core. Você pode criar um usando o comando dotnet new console ou do Visual Studio.

Abra o arquivo *Program.cs* no editor de código e substitua o código existente por este:

C#

```
using System.Diagnostics;

namespace ProcessTasksAsTheyFinish;

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

```
    }
}
```

Adicionar campos

Na definição de classe `Program`, adicione os dois seguintes campos:

```C#
static readonly HttpClient s_client = new HttpClient
{
    MaxResponseContentBufferSize = 1_000_000
};

static readonly IEnumerable<string> s_urlList = new string[]
{
    "https://learn.microsoft.com",
    "https://learn.microsoft.com/aspnet/core",
    "https://learn.microsoft.com/azure",
    "https://learn.microsoft.com/azure/devops",
    "https://learn.microsoft.com/dotnet",
    "https://learn.microsoft.com/dynamics365",
    "https://learn.microsoft.com/education",
    "https://learn.microsoft.com/enterprise-mobility-security",
    "https://learn.microsoft.com/gaming",
    "https://learn.microsoft.com/graph",
    "https://learn.microsoft.com/microsoft-365",
    "https://learn.microsoft.com/office",
    "https://learn.microsoft.com/powershell",
    "https://learn.microsoft.com/sql",
    "https://learn.microsoft.com/surface",
    "https://learn.microsoft.com/system-center",
    "https://learn.microsoft.com/visualstudio",
    "https://learn.microsoft.com/windows",
    "https://learn.microsoft.com/xamarin"
};
```

O `HttpClient` expõe a capacidade de enviar solicitações HTTP e receber respostas HTTP. O `s_urlList` contém todas as URLs que o aplicativo planeja processar.

Atualize o ponto de entrada do aplicativo

O principal ponto de entrada no aplicativo de console é o método `Main`. Substitua o método existente pelo seguinte:

```C#
```

```
static Task Main() => SumPageSizesAsync();
```

O método atualizado `Main` agora é considerado um Async main, que permite um ponto de entrada assíncrono no executável. Ele é expresso como uma chamada para `SumPageSizesAsync`.

Criar o método de tamanhos de página de soma assíncrona

Abaixo do método `Main`, adicione o método `SumPageSizesAsync`:

C#

```
static async Task SumPageSizesAsync()
{
    var stopwatch = Stopwatch.StartNew();

    IEnumerable<Task<int>> downloadTasksQuery =
        from url in s_urlList
        select ProcessUrlAsync(url, s_client);

    List<Task<int>> downloadTasks = downloadTasksQuery.ToList();

    int total = 0;
    while (downloadTasks.Any())
    {
        Task<int> finishedTask = await Task.WhenAny(downloadTasks);
        downloadTasks.Remove(finishedTask);
        total += await finishedTask;
    }

    stopwatch.Stop();

    Console.WriteLine($"\nTotal bytes returned:  {total:#,#}");
    Console.WriteLine($"Elapsed time:          {stopwatch.Elapsed}\n");
}
```

O loop `while` remove uma das tarefas em cada iteração. Depois que todas as tarefas forem concluídas, o loop terminará. O método começa instanciando e iniciando um Stopwatch. Em seguida, ele inclui uma consulta que, quando executada, cria uma coleção de tarefas. Cada chamada para `ProcessUrlAsync` no código a seguir retorna um Task<TResult>, em que `TResult` é um inteiro:

C#

```
IEnumerable<Task<int>> downloadTasksQuery =
    from url in s_urlList
    select ProcessUrlAsync(url, s_client);
```

Devido à execução adiada com o LINQ, você chama Enumerable.ToList para iniciar cada tarefa.

```C#
List<Task<int>> downloadTasks = downloadTasksQuery.ToList();
```

O loop `while` executa as seguintes etapas para cada tarefa na coleção:

1. Espera uma chamada para `WhenAny`, com o objetivo de identificar a primeira tarefa na coleção que concluiu o download.

   ```C#
   Task<int> finishedTask = await Task.WhenAny(downloadTasks);
   ```

2. Remove a tarefa da coleção.

   ```C#
   downloadTasks.Remove(finishedTask);
   ```

3. Espera `finishedTask`, que é retornado por uma chamada para `ProcessUrlAsync`. A variável `finishedTask` é uma Task<TResult> em que `TResult` é um inteiro. A tarefa já foi concluída, mas você espera para recuperar o tamanho do site baixado, como mostra o exemplo a seguir. Se a tarefa tiver falha, `await` gerará a primeira exceção filho armazenada no `AggregateException`, ao contrário da leitura da propriedade Task<TResult>.Result, que lançaria o `AggregateException`.

   ```C#
   total += await finishedTask;
   ```

Adicionar método de processo

Adicione o seguinte método `ProcessUrlAsync` abaixo do método `SumPageSizesAsync`:

```C#
static async Task<int> ProcessUrlAsync(string url, HttpClient client)
{
    byte[] content = await client.GetByteArrayAsync(url);
    Console.WriteLine($"{url,-60} {content.Length,10:#,#}");

    return content.Length;
}
```

Para qualquer URL fornecida, o método usará a instância `client` fornecida para obter a resposta como um `byte[]`. O comprimento é retornado depois que a URL e o comprimento são gravados no console.

Execute o programa várias vezes para verificar se os tamanhos baixados não aparecem sempre na mesma ordem.

> ⊗ **Cuidado**
>
> Você pode usar `WhenAny` em um loop, conforme descrito no exemplo, para resolver problemas que envolvem um número pequeno de tarefas. No entanto, outras abordagens são mais eficientes se você tiver um número grande de tarefas para processar. Para obter mais informações e exemplos, consulte **Processando tarefas quando elas são concluídas**.

Exemplo completo

O código a seguir é o texto completo do arquivo *Program.cs* para o exemplo.

```C#
using System.Diagnostics;

HttpClient s_client = new()
{
    MaxResponseContentBufferSize = 1_000_000
};

IEnumerable<string> s_urlList = new string[]
{
    "https://learn.microsoft.com",
    "https://learn.microsoft.com/aspnet/core",
    "https://learn.microsoft.com/azure",
    "https://learn.microsoft.com/azure/devops",
    "https://learn.microsoft.com/dotnet",
    "https://learn.microsoft.com/dynamics365",
```

```
    "https://learn.microsoft.com/education",
    "https://learn.microsoft.com/enterprise-mobility-security",
    "https://learn.microsoft.com/gaming",
    "https://learn.microsoft.com/graph",
    "https://learn.microsoft.com/microsoft-365",
    "https://learn.microsoft.com/office",
    "https://learn.microsoft.com/powershell",
    "https://learn.microsoft.com/sql",
    "https://learn.microsoft.com/surface",
    "https://learn.microsoft.com/system-center",
    "https://learn.microsoft.com/visualstudio",
    "https://learn.microsoft.com/windows",
    "https://learn.microsoft.com/xamarin"
};

await SumPageSizesAsync();

async Task SumPageSizesAsync()
{
    var stopwatch = Stopwatch.StartNew();

    IEnumerable<Task<int>> downloadTasksQuery =
        from url in s_urlList
        select ProcessUrlAsync(url, s_client);

    List<Task<int>> downloadTasks = downloadTasksQuery.ToList();

    int total = 0;
    while (downloadTasks.Any())
    {
        Task<int> finishedTask = await Task.WhenAny(downloadTasks);
        downloadTasks.Remove(finishedTask);
        total += await finishedTask;
    }

    stopwatch.Stop();

    Console.WriteLine($"\nTotal bytes returned:    {total:#,#}");
    Console.WriteLine($"Elapsed time:              {stopwatch.Elapsed}\n");
}

static async Task<int> ProcessUrlAsync(string url, HttpClient client)
{
    byte[] content = await client.GetByteArrayAsync(url);
    Console.WriteLine($"{url,-60} {content.Length,10:#,#}");

    return content.Length;
}

// Example output:
// https://learn.microsoft.com                                        132,517
// https://learn.microsoft.com/powershell                              57,375
// https://learn.microsoft.com/gaming                                  33,549
// https://learn.microsoft.com/aspnet/core                             88,714
// https://learn.microsoft.com/surface                                 39,840
```

```
// https://learn.microsoft.com/enterprise-mobility-security          30,903
// https://learn.microsoft.com/microsoft-365                         67,867
// https://learn.microsoft.com/windows                               26,816
// https://learn.microsoft.com/xamarin                               57,958
// https://learn.microsoft.com/dotnet                                78,706
// https://learn.microsoft.com/graph                                 48,277
// https://learn.microsoft.com/dynamics365                           49,042
// https://learn.microsoft.com/office                                67,867
// https://learn.microsoft.com/system-center                         42,887
// https://learn.microsoft.com/education                             38,636
// https://learn.microsoft.com/azure                                421,663
// https://learn.microsoft.com/visualstudio                          30,925
// https://learn.microsoft.com/sql                                   54,608
// https://learn.microsoft.com/azure/devops                          86,034

// Total bytes returned:    1,454,184
// Elapsed time:            00:00:01.1290403
```

Confira também

- WhenAny
- Programação assíncrona com async e await (C#)

Collaborate with us on GitHub

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

.NET

.NET feedback

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

Open a documentation issue

Provide product feedback

Acesso a arquivos assíncronos (C#)

Artigo • 10/05/2023

Você pode usar o recurso async para acessar arquivos. Usando o recurso async, você pode chamar os métodos assíncronos sem usar retornos de chamada ou dividir seu código em vários métodos ou expressões lambda. Para tornar síncrono um código assíncrono, basta chamar um método assíncrono em vez de um método síncrono e adicionar algumas palavras-chave ao código.

Você pode considerar seguintes motivos para adicionar a assincronia às chamadas de acesso ao arquivo:

- ✓ A assincronia torna os aplicativos de interface do usuário mais responsivos porque o thread de interface do usuário que inicia a operação pode executar outro trabalho. Se o thread de interface do usuário precisar executar o código que leva muito tempo (por exemplo, mais de 50 milissegundos), a interface do usuário poderá congelar até que a E/S seja concluída e o thread da interface do usuário possa processar entradas do mouse e do teclado e outros eventos.
- ✓ A assincronia melhora a escalabilidade do ASP.NET e outros aplicativos baseados em servidor reduzindo a necessidade de threads. Se o aplicativo usar um thread dedicado por resposta e mil solicitações forem tratadas simultaneamente, serão necessários mil threads. As operações assíncronas normalmente não precisam usar um thread durante a espera. Elas podem usar o thread de conclusão de E/S existente rapidamente no final.
- ✓ A latência de uma operação de acesso de arquivo pode ser muito baixa nas condições atuais, mas a latência pode aumentar consideravelmente no futuro. Por exemplo, um arquivo pode ser movido para um servidor que está do outro lado do mundo.
- ✓ A sobrecarga adicional de usar o recurso async é pequena.
- ✓ As tarefas assíncronas podem facilmente ser executadas em paralelo.

Usar classes apropriadas

Os exemplos simples neste tópico demonstram File.WriteAllTextAsync e File.ReadAllTextAsync. Para controle fino sobre as operações de E/S do arquivo, use a classe FileStream, que tem uma opção que faz com que a E/S assíncrona ocorra no nível do sistema operacional. Usando essa opção, você pode evitar o bloqueio de um thread de pool de threads em muitos casos. Para habilitar essa opção, você deve especificar o argumento `useAsync=true` ou `options=FileOptions.Asynchronous` na chamada do construtor.

Você não pode usar essa opção com StreamReader e StreamWriter se você os abrir diretamente especificando um caminho de arquivo. No entanto, você poderá usar essa opção se fornecer um Stream que a classe FileStream abriu. Chamadas assíncronas serão mais rápidas em aplicativos de interface do usuário mesmo se um thread de pool de threads estiver bloqueado, porque o thread de interface do usuário não é bloqueado durante a espera.

Gravar texto

Os exemplos a seguir gravam texto em um arquivo. A cada instrução await, o método sai imediatamente. Quando o arquivo de E/S for concluído, o método continuará na instrução após a instrução await. O modificador async é a definição de métodos que usam a instrução await.

Exemplo simples

C#

```
public async Task SimpleWriteAsync()
{
    string filePath = "simple.txt";
    string text = $"Hello World";

    await File.WriteAllTextAsync(filePath, text);
}
```

Exemplo de controle finito

C#

```
public async Task ProcessWriteAsync()
{
    string filePath = "temp.txt";
    string text = $"Hello World{Environment.NewLine}";

    await WriteTextAsync(filePath, text);
}

async Task WriteTextAsync(string filePath, string text)
{
    byte[] encodedText = Encoding.Unicode.GetBytes(text);

    using var sourceStream =
        new FileStream(
            filePath,
```

```
            FileMode.Create, FileAccess.Write, FileShare.None,
            bufferSize: 4096, useAsync: true);

    await sourceStream.WriteAsync(encodedText, 0, encodedText.Length);
}
```

O exemplo original tem a instrução `await sourceStream.WriteAsync(encodedText, 0, encodedText.Length);`, que é uma contração das duas instruções a seguir:

```
C#

Task theTask = sourceStream.WriteAsync(encodedText, 0, encodedText.Length);
await theTask;
```

A primeira instrução retorna uma tarefa e faz com que o processamento do arquivo seja iniciado. A segunda instrução com o await faz com que o método saia imediatamente e retorne uma tarefa diferente. Quando o processamento do arquivo é concluído posteriormente, a execução retorna para a instrução após a await.

Ler texto

Os exemplos a seguir leem texto de um arquivo.

Exemplo simples

```
C#

public async Task SimpleReadAsync()
{
    string filePath = "simple.txt";
    string text = await File.ReadAllTextAsync(filePath);

    Console.WriteLine(text);
}
```

Exemplo de controle finito

O texto é armazenado em buffer e, nesse caso, colocado em um StringBuilder. Diferentemente do exemplo anterior, a avaliação de await produz um valor. O método ReadAsync retorna um Task<Int32>, portanto, a avaliação do await produz um `Int32` valor `numRead` após a conclusão da operação. Para obter mais informações, consulte Tipos de retorno assíncronos (C#).

C#

```
public async Task ProcessReadAsync()
{
    try
    {
        string filePath = "temp.txt";
        if (File.Exists(filePath) != false)
        {
            string text = await ReadTextAsync(filePath);
            Console.WriteLine(text);
        }
        else
        {
            Console.WriteLine($"file not found: {filePath}");
        }
    }
    catch (Exception ex)
    {
        Console.WriteLine(ex.Message);
    }
}

async Task<string> ReadTextAsync(string filePath)
{
    using var sourceStream =
        new FileStream(
            filePath,
            FileMode.Open, FileAccess.Read, FileShare.Read,
            bufferSize: 4096, useAsync: true);

    var sb = new StringBuilder();

    byte[] buffer = new byte[0x1000];
    int numRead;
    while ((numRead = await sourceStream.ReadAsync(buffer, 0,
buffer.Length)) != 0)
    {
        string text = Encoding.Unicode.GetString(buffer, 0, numRead);
        sb.Append(text);
    }

    return sb.ToString();
}
```

E/S assíncrona paralela

Os exemplos a seguir demonstram o processamento paralelo gravando dez arquivos de texto.

Exemplo simples

C#

```
public async Task SimpleParallelWriteAsync()
{
    string folder = Directory.CreateDirectory("tempfolder").Name;
    IList<Task> writeTaskList = new List<Task>();

    for (int index = 11; index <= 20; ++ index)
    {
        string fileName = $"file-{index:00}.txt";
        string filePath = $"{folder}/{fileName}";
        string text = $"In file {index}{Environment.NewLine}";

        writeTaskList.Add(File.WriteAllTextAsync(filePath, text));
    }

    await Task.WhenAll(writeTaskList);
}
```

Exemplo de controle finito

Para cada arquivo, o método WriteAsync retorna uma tarefa que é então adicionada a uma lista de tarefas. A instrução `await Task.WhenAll(tasks);` sai do método e retoma no método quando o processamento do arquivo é concluído para todas as tarefas.

O exemplo fecha todas as instâncias de FileStream em um bloco `finally` após as tarefas serem concluídas. Se cada `FileStream` foi criado em uma instrução `using`, o `FileStream` pode ter sido descartado antes de a tarefa ter sido concluída.

Qualquer aumento de desempenho é quase que totalmente do processamento paralelo e não o processamento assíncrono. As vantagens da assincronia são que ela não bloqueia vários threads e que ela não bloqueia o thread da interface do usuário.

C#

```
public async Task ProcessMultipleWritesAsync()
{
    IList<FileStream> sourceStreams = new List<FileStream>();

    try
    {
        string folder = Directory.CreateDirectory("tempfolder").Name;
        IList<Task> writeTaskList = new List<Task>();

        for (int index = 1; index <= 10; ++ index)
        {
```

```
            string fileName = $"file-{index:00}.txt";
            string filePath = $"{folder}/{fileName}";

            string text = $"In file {index}{Environment.NewLine}";
            byte[] encodedText = Encoding.Unicode.GetBytes(text);

            var sourceStream =
                new FileStream(
                    filePath,
                    FileMode.Create, FileAccess.Write, FileShare.None,
                    bufferSize: 4096, useAsync: true);

            Task writeTask = sourceStream.WriteAsync(encodedText, 0,
encodedText.Length);
            sourceStreams.Add(sourceStream);

            writeTaskList.Add(writeTask);
        }

        await Task.WhenAll(writeTaskList);
    }
    finally
    {
        foreach (FileStream sourceStream in sourceStreams)
        {
            sourceStream.Close();
        }
    }
}
```

Ao usar os métodos WriteAsync e ReadAsync, você pode especificar um CancellationToken, que pode ser usado para cancelar o fluxo intermediário da operação. Para saber mais, confira Cancelamento em threads gerenciados.

Confira também

- Programação assíncrona com async e await (C#)
- Tipos de retorno assíncronos (C#)

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For

.NET

.NET feedback

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

Open a documentation issue

Provide product feedback

more information, see our contributor guide.

Cancelar uma lista de tarefas

Artigo • 28/03/2023

Você pode cancelar um aplicativo de console assíncrono se não quiser esperar que ele seja concluído. Seguindo o exemplo neste tópico, você pode adicionar um cancelamento a um aplicativo que baixa o conteúdo de uma lista de sites. Você pode cancelar várias tarefas associando a instância CancellationTokenSource a cada tarefa. Se você selecionar a tecla `Enter`, você cancele todas as tarefas que ainda não foram concluídas.

Este tutorial abrange:

- ✓ Criando um aplicativo de console .NET
- ✓ Escrever um aplicativo assíncrono que dá suporte a cancelamento
- ✓ Demonstrando cancelamento de sinalização

Pré-requisitos

Este tutorial exige o seguinte:

- .NET 5 ou SDK posterior
- IDE (ambiente de desenvolvimento integrado)
  - Recomendamos o Visual Studio ou o Visual Studio Code

Criar aplicativo de exemplo

Criar um novo aplicativo de console do .NET Core. Você pode criar um usando o comando ou do dotnet new consoleVisual Studio. Abra o arquivo *Program.cs* em seu editor de código favorito.

Substituir usando instruções

Substitua as instruções de uso existentes por estas declarações:

C#

```
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Net.Http;
```

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

Adicionar campos

Na definição de classe `Program`, adicione estes três campos:

```C#
static readonly CancellationTokenSource s_cts = new
CancellationTokenSource();

static readonly HttpClient s_client = new HttpClient
{
    MaxResponseContentBufferSize = 1_000_000
};

static readonly IEnumerable<string> s_urlList = new string[]
{
    "https://learn.microsoft.com",
    "https://learn.microsoft.com/aspnet/core",
    "https://learn.microsoft.com/azure",
    "https://learn.microsoft.com/azure/devops",
    "https://learn.microsoft.com/dotnet",
    "https://learn.microsoft.com/dynamics365",
    "https://learn.microsoft.com/education",
    "https://learn.microsoft.com/enterprise-mobility-security",
    "https://learn.microsoft.com/gaming",
    "https://learn.microsoft.com/graph",
    "https://learn.microsoft.com/microsoft-365",
    "https://learn.microsoft.com/office",
    "https://learn.microsoft.com/powershell",
    "https://learn.microsoft.com/sql",
    "https://learn.microsoft.com/surface",
    "https://learn.microsoft.com/system-center",
    "https://learn.microsoft.com/visualstudio",
    "https://learn.microsoft.com/windows",
    "https://learn.microsoft.com/xamarin"
};
```

O CancellationTokenSource é usado para sinalizar um cancelamento solicitado para um CancellationToken. O `HttpClient` expõe a capacidade de enviar solicitações HTTP e receber respostas HTTP. O `s_urlList` contém todas as URLs que o aplicativo planeja processar.

Atualize o ponto de entrada do aplicativo

O principal ponto de entrada no aplicativo de console é o método `Main`. Substitua o método existente pelo seguinte:

C#

```
static async Task Main()
{
    Console.WriteLine("Application started.");
    Console.WriteLine("Press the ENTER key to cancel...\n");

    Task cancelTask = Task.Run(() =>
    {
        while (Console.ReadKey().Key != ConsoleKey.Enter)
        {
            Console.WriteLine("Press the ENTER key to cancel...");
        }

        Console.WriteLine("\nENTER key pressed: cancelling downloads.\n");
        s_cts.Cancel();
    });

    Task sumPageSizesTask = SumPageSizesAsync();

    Task finishedTask = await Task.WhenAny(new[] { cancelTask,
sumPageSizesTask });
    if (finishedTask == cancelTask)
    {
        // wait for the cancellation to take place:
        try
        {
            await sumPageSizesTask;
            Console.WriteLine("Download task completed before cancel request
was processed.");
        }
        catch (TaskCanceledException)
        {
            Console.WriteLine("Download task has been cancelled.");
        }
    }

    Console.WriteLine("Application ending.");
}
```

O método atualizado `Main` agora é considerado um Async main, que permite um ponto de entrada assíncrono no executável. Ele grava algumas mensagens instrutivas no console e, em seguida, declara uma instância Task chamada `cancelTask`, que lerá os traços de chave do console. Se a tecla `Enter` for pressionada, será feita uma chamada CancellationTokenSource.Cancel(). Isso sinalizará o cancelamento. Em seguida, a variável `sumPageSizesTask` é atribuída do método `SumPageSizesAsync`. Ambas as tarefas são

então passadas para Task.WhenAny(Task[]), o que continuará quando qualquer uma das duas tarefas tiver sido concluída.

O próximo bloco de código garante que o aplicativo não saia até que o cancelamento seja processado. Se a primeira tarefa a ser concluída for `cancelTask`, o `sumPageSizeTask` será aguardado. Se ele foi cancelado, quando aguardado, ele lança um System.Threading.Tasks.TaskCanceledException. O bloco captura essa exceção e imprime uma mensagem.

Criar o método de tamanhos de página de soma assíncrona

Abaixo do método `Main`, adicione o método `SumPageSizesAsync`:

C#

```
static async Task SumPageSizesAsync()
{
    var stopwatch = Stopwatch.StartNew();

    int total = 0;
    foreach (string url in s_urlList)
    {
        int contentLength = await ProcessUrlAsync(url, s_client,
s_cts.Token);
        total += contentLength;
    }

    stopwatch.Stop();

    Console.WriteLine($"\nTotal bytes returned:  {total:#,#}");
    Console.WriteLine($"Elapsed time:          {stopwatch.Elapsed}\n");
}
```

O método começa instanciando e iniciando um Stopwatch. Em seguida, ele faz loops por cada URL no `s_urlList` e chama `ProcessUrlAsync`. A cada iteração, o método `s_cts.Token` é passado para o método `ProcessUrlAsync` e o código retorna um Task<TResult>, que que `TResult` é um inteiro:

C#

```
int total = 0;
foreach (string url in s_urlList)
{
    int contentLength = await ProcessUrlAsync(url, s_client, s_cts.Token);
```

```
    total += contentLength;
}
```

Adicionar método de processo

Adicione o seguinte método `ProcessUrlAsync` abaixo do método `SumPageSizesAsync`:

C#

```
static async Task<int> ProcessUrlAsync(string url, HttpClient client,
CancellationToken token)
{
    HttpResponseMessage response = await client.GetAsync(url, token);
    byte[] content = await response.Content.ReadAsByteArrayAsync(token);
    Console.WriteLine($"{url,-60} {content.Length,10:#,#}");

    return content.Length;
}
```

Para qualquer URL fornecida, o método usará a instância `client` fornecida para obter a resposta como um `byte[]`. A instância CancellationToken é passada para os métodos HttpClient.GetAsync(String, CancellationToken) e HttpContent.ReadAsByteArrayAsync(). O `token` é usado para registrar-se para cancelamento solicitado. O comprimento é retornado depois que a URL e o comprimento são gravados no console.

Exemplo de saída de aplicativo

Console

```
Application started.
Press the ENTER key to cancel...

https://learn.microsoft.com                                       37,357
https://learn.microsoft.com/aspnet/core                           85,589
https://learn.microsoft.com/azure                                398,939
https://learn.microsoft.com/azure/devops                          73,663
https://learn.microsoft.com/dotnet                                67,452
https://learn.microsoft.com/dynamics365                           48,582
https://learn.microsoft.com/education                             22,924

ENTER key pressed: cancelling downloads.

Application ending.
```

Exemplo completo

O código a seguir é o texto completo do arquivo *Program.cs* para o exemplo.

C#

```
using System.Diagnostics;

class Program
{
    static readonly CancellationTokenSource s_cts = new
CancellationTokenSource();

    static readonly HttpClient s_client = new HttpClient
    {
        MaxResponseContentBufferSize = 1_000_000
    };

    static readonly IEnumerable<string> s_urlList = new string[]
    {
            "https://learn.microsoft.com",
            "https://learn.microsoft.com/aspnet/core",
            "https://learn.microsoft.com/azure",
            "https://learn.microsoft.com/azure/devops",
            "https://learn.microsoft.com/dotnet",
            "https://learn.microsoft.com/dynamics365",
            "https://learn.microsoft.com/education",
            "https://learn.microsoft.com/enterprise-mobility-security",
            "https://learn.microsoft.com/gaming",
            "https://learn.microsoft.com/graph",
            "https://learn.microsoft.com/microsoft-365",
            "https://learn.microsoft.com/office",
            "https://learn.microsoft.com/powershell",
            "https://learn.microsoft.com/sql",
            "https://learn.microsoft.com/surface",
            "https://learn.microsoft.com/system-center",
            "https://learn.microsoft.com/visualstudio",
            "https://learn.microsoft.com/windows",
            "https://learn.microsoft.com/xamarin"
    };

    static async Task Main()
    {
        Console.WriteLine("Application started.");
        Console.WriteLine("Press the ENTER key to cancel...\n");

        Task cancelTask = Task.Run(() =>
        {
            while (Console.ReadKey().Key != ConsoleKey.Enter)
            {
                Console.WriteLine("Press the ENTER key to cancel...");
            }
```

```
            Console.WriteLine("\nENTER key pressed: cancelling
downloads.\n");
            s_cts.Cancel();
        });

        Task sumPageSizesTask = SumPageSizesAsync();

        Task finishedTask = await Task.WhenAny(new[] { cancelTask,
sumPageSizesTask });
        if (finishedTask == cancelTask)
        {
            // wait for the cancellation to take place:
            try
            {
                await sumPageSizesTask;
                Console.WriteLine("Download task completed before cancel
request was processed.");
            }
            catch (TaskCanceledException)
            {
                Console.WriteLine("Download task has been cancelled.");
            }
        }

        Console.WriteLine("Application ending.");
    }

    static async Task SumPageSizesAsync()
    {
        var stopwatch = Stopwatch.StartNew();

        int total = 0;
        foreach (string url in s_urlList)
        {
            int contentLength = await ProcessUrlAsync(url, s_client,
s_cts.Token);
            total += contentLength;
        }

        stopwatch.Stop();

        Console.WriteLine($"\nTotal bytes returned:  {total:#,#}");
        Console.WriteLine($"Elapsed time:          {stopwatch.Elapsed}\n");
    }

    static async Task<int> ProcessUrlAsync(string url, HttpClient client,
CancellationToken token)
    {
        HttpResponseMessage response = await client.GetAsync(url, token);
        byte[] content = await response.Content.ReadAsByteArrayAsync(token);
        Console.WriteLine($"{url,-60} {content.Length,10:#,#}");

        return content.Length;
    }
}
```

Confira também

- CancellationToken
- CancellationTokenSource
- Programação assíncrona com async e await (C#)

Próximas etapas

Cancelar tarefas assíncronas após um período (C#)

Collaborate with us on GitHub

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

.NET

.NET feedback

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

Open a documentation issue

Provide product feedback

Cancelar tarefas assíncronas após um período

Artigo • 12/09/2023

Você pode cancelar uma operação assíncrona após um período usando o método CancellationTokenSource.CancelAfter se não quiser aguardar a conclusão da operação. Esse método agenda o cancelamento de quaisquer tarefas associadas que não são concluídas dentro do período designado pela expressão `CancelAfter`.

Este exemplo adiciona o código desenvolvido em Cancelar uma tarefa assíncrona ou uma lista de tarefas (C#) para baixar uma lista de sites e para exibir o tamanho dos conteúdos de cada um.

Este tutorial abrange:

- ✓ Atualizando um aplicativo de console .NET existente
- ✓ Agendando um cancelamento

Pré-requisitos

Este tutorial exige o seguinte:

- Espera-se que você tenha criado um aplicativo no tutorial Cancelar uma lista de tarefas (C#)
- .NET 5 ou SDK posterior
- IDE (ambiente de desenvolvimento integrado)
  - Recomendamos o Visual Studio ou o Visual Studio Code

Atualize o ponto de entrada do aplicativo.

Substitua o método `Main` existente pelo seguinte:

C#

```
static async Task Main()
{
    Console.WriteLine("Application started.");

    try
    {
        s_cts.CancelAfter(3500);
```

```
        await SumPageSizesAsync();
    }
    catch (OperationCanceledException)
    {
        Console.WriteLine("\nTasks cancelled: timed out.\n");
    }
    finally
    {
        s_cts.Dispose();
    }

    Console.WriteLine("Application ending.");
}
```

O método `Main` atualizado grava algumas mensagens instrutivas no console. Dentro do try-catch, uma chamada para CancellationTokenSource.CancelAfter(Int32) agenda um cancelamento. Isso sinalizará o cancelamento após um período.

Em seguida, o método `SumPageSizesAsync` é aguardado. Se o processamento de todas as URLs ocorrer mais rápido do que o cancelamento agendado, o aplicativo é encerrado. No entanto, se o cancelamento agendado for disparado antes que todas as URLs sejam processadas, um OperationCanceledException será gerado.

Exemplo de saída de aplicativo

Console

```
Application started.

https://learn.microsoft.com                                        37,357
https://learn.microsoft.com/aspnet/core                            85,589
https://learn.microsoft.com/azure                                 398,939
https://learn.microsoft.com/azure/devops                           73,663

Tasks cancelled: timed out.

Application ending.
```

Exemplo completo

O código a seguir é o texto completo do arquivo *Program.cs* para o exemplo.

C#

```
using System.Diagnostics;

class Program
```

```
{
    static readonly CancellationTokenSource s_cts = new
CancellationTokenSource();

    static readonly HttpClient s_client = new HttpClient
    {
        MaxResponseContentBufferSize = 1_000_000
    };

    static readonly IEnumerable<string> s_urlList = new string[]
    {
            "https://learn.microsoft.com",
            "https://learn.microsoft.com/aspnet/core",
            "https://learn.microsoft.com/azure",
            "https://learn.microsoft.com/azure/devops",
            "https://learn.microsoft.com/dotnet",
            "https://learn.microsoft.com/dynamics365",
            "https://learn.microsoft.com/education",
            "https://learn.microsoft.com/enterprise-mobility-security",
            "https://learn.microsoft.com/gaming",
            "https://learn.microsoft.com/graph",
            "https://learn.microsoft.com/microsoft-365",
            "https://learn.microsoft.com/office",
            "https://learn.microsoft.com/powershell",
            "https://learn.microsoft.com/sql",
            "https://learn.microsoft.com/surface",
            "https://learn.microsoft.com/system-center",
            "https://learn.microsoft.com/visualstudio",
            "https://learn.microsoft.com/windows",
            "https://learn.microsoft.com/xamarin"
    };

    static async Task Main()
    {
        Console.WriteLine("Application started.");

        try
        {
            s_cts.CancelAfter(3500);

            await SumPageSizesAsync();
        }
        catch (OperationCanceledException)
        {
            Console.WriteLine("\nTasks cancelled: timed out.\n");
        }
        finally
        {
            s_cts.Dispose();
        }

        Console.WriteLine("Application ending.");
    }

    static async Task SumPageSizesAsync()
```

```
    {
        var stopwatch = Stopwatch.StartNew();

        int total = 0;
        foreach (string url in s_urlList)
        {
            int contentLength = await ProcessUrlAsync(url, s_client,
s_cts.Token);
            total += contentLength;
        }

        stopwatch.Stop();

        Console.WriteLine($"\nTotal bytes returned:  {total:#,#}");
        Console.WriteLine($"Elapsed time:          {stopwatch.Elapsed}\n");
    }

    static async Task<int> ProcessUrlAsync(string url, HttpClient client,
CancellationToken token)
    {
        HttpResponseMessage response = await client.GetAsync(url, token);
        byte[] content = await response.Content.ReadAsByteArrayAsync(token);
        Console.WriteLine($"{url,-60} {content.Length,10:#,#}");

        return content.Length;
    }
}
```

Confira também

- CancellationToken
- CancellationTokenSource
- Programação assíncrona com async e await (C#)
- Cancelar uma lista de tarefas (C#)

Collaborate with us on GitHub

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

.NET feedback

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

Open a documentation issue

Provide product feedback

Tutorial: Gerar e consumir fluxos assíncronos usando C# e .NET

18/06/2025

Os fluxos assíncronos modelam uma fonte de dados de streaming. Os fluxos de dados geralmente recuperam ou geram elementos de forma assíncrona. Eles fornecem um modelo de programação natural para fontes de dados de streaming assíncronas.

Neste tutorial, você aprenderá a:

- ✓ Crie uma fonte de dados que gere uma sequência de elementos de dados de forma assíncrona.
- ✓ Consuma essa fonte de dados de forma assíncrona.
- ✓ Suporte ao cancelamento e contextos capturados para fluxos assíncronos.
- ✓ Reconheça quando a nova interface e a fonte de dados são preferenciais a sequências de dados síncronas anteriores.

Pré-requisitos

Você precisará configurar seu computador para executar o .NET, incluindo o compilador C#. O compilador C# está disponível com o Visual Studio 2022 ou o SDK do .NET.

Você precisará criar um token de acesso do GitHub para poder acessar o ponto de extremidade do GraphQL do GitHub. Selecione as seguintes permissões para o Token de Acesso do GitHub:

- repo:status
- public_repo

Salve o token de acesso em um local seguro para que você possa usá-lo para obter acesso ao ponto de extremidade da API do GitHub.

> ⚠ **Aviso**
>
> Mantenha seu token de acesso pessoal seguro. Qualquer software com seu token de acesso pessoal pode fazer chamadas à API do GitHub usando seus direitos de acesso.

Este tutorial pressupõe que você esteja familiarizado com C# e .NET, incluindo o Visual Studio ou a CLI do .NET.

Executar o aplicativo inicial

Você pode obter o código para o aplicativo de início usado neste tutorial no repositório dotnet/docs na pasta asynchronous-programming/snippets.

O aplicativo inicial é um aplicativo de console que usa a interface GraphQL do GitHub para recuperar problemas recentes escritos no repositório dotnet/docs. Comece examinando o seguinte código para o método inicial do aplicativo `Main`.

C#

```
static async Task Main(string[] args)
{
    //Follow these steps to create a GitHub Access Token
    // https://help.github.com/articles/creating-a-personal-access-token-for-the-
command-line/#creating-a-token
    //Select the following permissions for your GitHub Access Token:
    // - repo:status
    // - public_repo
    // Replace the 3rd parameter to the following code with your GitHub access
token.
    var key = GetEnvVariable("GitHubKey",
    "You must store your GitHub key in the 'GitHubKey' environment variable",
    "");

    var client = new GitHubClient(new
Octokit.ProductHeaderValue("IssueQueryDemo"))
    {
        Credentials = new Octokit.Credentials(key)
    };

    var progressReporter = new progressStatus((num) =>
    {
        Console.WriteLine($"Received {num} issues in total");
    });
    CancellationTokenSource cancellationSource = new CancellationTokenSource();

    try
    {
        var results = await RunPagedQueryAsync(client, PagedIssueQuery, "docs",
            cancellationSource.Token, progressReporter);
        foreach(var issue in results)
            Console.WriteLine(issue);
    }
    catch (OperationCanceledException)
    {
        Console.WriteLine("Work has been cancelled");
    }
}
```

Você pode definir uma variável de ambiente `GitHubKey` para o seu token de acesso pessoal, ou você pode substituir o último argumento na chamada `GetEnvVariable` pelo seu token de acesso pessoal. Não coloque seu código de acesso no código-fonte se você compartilhar a origem com outras pessoas. Nunca carregue códigos de acesso em um repositório de origem compartilhado.

Depois de criar o cliente GitHub, o código em `Main` cria um objeto de relatório de progresso e um token de cancelamento. Depois que esses objetos são criados, `Main` chama `RunPagedQueryAsync` para recuperar os 250 incidentes mais recentes criados. Depois que essa tarefa for concluída, os resultados serão exibidos.

Ao executar o aplicativo inicial, você pode fazer algumas observações importantes sobre como esse aplicativo é executado. Você verá o progresso sendo relatado para cada página que for retornada do GitHub. Você pode observar uma pausa perceptível antes que o GitHub retorne cada nova página de problemas. Por fim, os problemas são exibidos somente depois que todas as 10 páginas foram recuperadas do GitHub.

Examinar a implementação

A implementação revela por que você observou o comportamento discutido na seção anterior. Examine o código para `RunPagedQueryAsync`:

C#

```
private static async Task<JArray> RunPagedQueryAsync(GitHubClient client, string
queryText, string repoName, CancellationToken cancel, IProgress<int> progress)
{
    var issueAndPRQuery = new GraphQLRequest
    {
        Query = queryText
    };
    issueAndPRQuery.Variables["repo_name"] = repoName;

    JArray finalResults = new JArray();
    bool hasMorePages = true;
    int pagesReturned = 0;
    int issuesReturned = 0;

    // Stop with 10 pages, because these are large repos:
    while (hasMorePages && (pagesReturned++ < 10))
    {
        var postBody = issueAndPRQuery.ToJsonText();
        var response = await client.Connection.Post<string>(new
Uri("https://api.github.com/graphql"),
            postBody, "application/json", "application/json");

        JObject results = JObject.Parse(response.HttpResponse.Body.ToString()!);
```

```
        int totalCount = (int)issues(results)["totalCount"]!;
        hasMorePages = (bool)pageInfo(results)["hasPreviousPage"]!;
        issueAndPRQuery.Variables["start_cursor"] = pageInfo(results)
["startCursor"]!.ToString();
        issuesReturned += issues(results)["nodes"]!.Count();
        finalResults.Merge(issues(results)["nodes"]!);
        progress?.Report(issuesReturned);
        cancel.ThrowIfCancellationRequested();
    }
    return finalResults;

    JObject issues(JObject result) => (JObject)result["data"]!["repository"]!
["issues"]!;
    JObject pageInfo(JObject result) => (JObject)issues(result)["pageInfo"]!;
}
```

A primeira coisa que este método faz é criar o objeto POST usando a `GraphQLRequest` classe:

```C#
public class GraphQLRequest
{
    [JsonProperty("query")]
    public string? Query { get; set; }

    [JsonProperty("variables")]
    public IDictionary<string, object> Variables { get; } = new Dictionary<string,
object>();

    public string ToJsonText() =>
        JsonConvert.SerializeObject(this);
}
```

que ajuda a formar o corpo do objeto POST e a convertê-lo corretamente em JSON apresentado como uma cadeia de caracteres única com o método `ToJsonText`, que remove todos os caracteres de nova linha do corpo da solicitação marcando-os com o caractere de escape (barra invertida) `\`.

Vamos nos concentrar no algoritmo de paginação e na estrutura assíncrona do código anterior. (Você pode consultar a documentação do GraphQL do GitHub para obter detalhes sobre a API do GraphQL do GitHub.) O `RunPagedQueryAsync` método enumera os problemas dos mais recentes aos mais antigos. Solicita 25 itens por página e examina a `pageInfo` estrutura da resposta para prosseguir para a página anterior. Isso segue o suporte de paginação padrão do GraphQL para respostas de várias páginas. A resposta inclui um `pageInfo` objeto que inclui um `hasPreviousPages` valor e um `startCursor` valor usado para solicitar a página anterior. Os

problemas estão na `nodes` matriz. O `RunPagedQueryAsync` método acrescenta esses nós a uma matriz que contém todos os resultados de todas as páginas.

Depois de recuperar e restaurar uma página de resultados, `RunPagedQueryAsync` relata o progresso e verifica se há cancelamento. Se o cancelamento tiver sido solicitado, `RunPagedQueryAsync` gerará um OperationCanceledException.

Há vários elementos nesse código que podem ser aprimorados. Acima de tudo, `RunPagedQueryAsync` deve alocar armazenamento para todos os problemas retornados. Esse exemplo para em 250 problemas porque recuperar todos os problemas abertos exigiria muito mais memória para armazenar todos os problemas recuperados. Os protocolos para dar suporte a relatórios de progresso e cancelamento tornam o algoritmo mais difícil de entender em sua primeira leitura. Mais tipos e APIs estão envolvidos. Você deve rastrear as comunicações por meio do CancellationTokenSource e do associado CancellationToken para entender onde o cancelamento é solicitado e onde ele é concedido.

Os fluxos assíncronos fornecem uma maneira mais eficiente

Os fluxos assíncronos e o suporte ao idioma associado abordam todas essas preocupações. O código que gera a sequência agora pode usar `yield return` para retornar elementos em um método que foi declarado com seu modificador `async`. Você pode consumir um fluxo assíncrono usando um `await foreach` loop assim como consome qualquer sequência usando um `foreach` loop.

Esses novos recursos de linguagem dependem de três novas interfaces adicionadas ao .NET Standard 2.1 e implementadas no .NET Core 3.0:

- System.Collections.Generic.IAsyncEnumerable<T>
- System.Collections.Generic.IAsyncEnumerator<T>
- System.IAsyncDisposable

Essas três interfaces devem ser familiares para a maioria dos desenvolvedores do C#. Eles se comportam de maneira semelhante aos seus equivalentes síncronos:

- System.Collections.Generic.IEnumerable<T>
- System.Collections.Generic.IEnumerator<T>
- System.IDisposable

Um tipo que pode não ser familiar é System.Threading.Tasks.ValueTask. O `ValueTask` struct fornece uma API semelhante à System.Threading.Tasks.Task classe. `ValueTask` é usado nessas interfaces por motivos de desempenho.

Converter para fluxos assíncronos

Em seguida, converta o `RunPagedQueryAsync` método para gerar um fluxo assíncrono. Primeiro, altere a assinatura de `RunPagedQueryAsync` para retornar um `IAsyncEnumerable<JToken>`, e remova o token de cancelamento e os objetos de progresso da lista de parâmetros, conforme mostrado no código a seguir.

C#

```
private static async IAsyncEnumerable<JToken> RunPagedQueryAsync(GitHubClient
client,
    string queryText, string repoName)
```

O código inicial processa cada página conforme a página é recuperada, conforme mostrado no seguinte código:

C#

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

Substitua essas três linhas pelo seguinte código:

C#

```
foreach (JObject issue in issues(results)["nodes"]!)
    yield return issue;
```

Você também pode remover a declaração de `finalResults` anterior neste método e a instrução de `return` que segue o loop que você modificou.

Você concluiu as alterações para gerar um fluxo assíncrono. O método concluído deve ser semelhante ao seguinte código:

C#

```
private static async IAsyncEnumerable<JToken> RunPagedQueryAsync(GitHubClient
client,
    string queryText, string repoName)
{
    var issueAndPRQuery = new GraphQLRequest
    {
        Query = queryText
    };
    issueAndPRQuery.Variables["repo_name"] = repoName;
```

```
    bool hasMorePages = true;
    int pagesReturned = 0;
    int issuesReturned = 0;

    // Stop with 10 pages, because these are large repos:
    while (hasMorePages && (pagesReturned++ < 10))
    {
        var postBody = issueAndPRQuery.ToJsonText();
        var response = await client.Connection.Post<string>(new
Uri("https://api.github.com/graphql"),
            postBody, "application/json", "application/json");

        JObject results = JObject.Parse(response.HttpResponse.Body.ToString()!);

        int totalCount = (int)issues(results)["totalCount"]!;
        hasMorePages = (bool)pageInfo(results)["hasPreviousPage"]!;
        issueAndPRQuery.Variables["start_cursor"] = pageInfo(results)
["startCursor"]!.ToString();
        issuesReturned += issues(results)["nodes"]!.Count();

        foreach (JObject issue in issues(results)["nodes"]!)
            yield return issue;
    }

    JObject issues(JObject result) => (JObject)result["data"]!["repository"]!
["issues"]!;
    JObject pageInfo(JObject result) => (JObject)issues(result)["pageInfo"]!;
}
```

Em seguida, você altera o código que consome a coleção para consumir o fluxo assíncrono. Encontre o seguinte código em `Main` que processa a coleção de problemas.

C#

```
var progressReporter = new progressStatus((num) =>
{
    Console.WriteLine($"Received {num} issues in total");
});
CancellationTokenSource cancellationSource = new CancellationTokenSource();

try
{
    var results = await RunPagedQueryAsync(client, PagedIssueQuery, "docs",
        cancellationSource.Token, progressReporter);
    foreach(var issue in results)
        Console.WriteLine(issue);
}
catch (OperationCanceledException)
{
    Console.WriteLine("Work has been cancelled");
}
```

Substitua esse código pelo seguinte `await foreach` loop:

```
C#

int num = 0;
await foreach (var issue in RunPagedQueryAsync(client, PagedIssueQuery, "docs"))
{
    Console.WriteLine(issue);
    Console.WriteLine($"Received {++num} issues in total");
}
```

A nova interface IAsyncEnumerator<T> deriva de IAsyncDisposable. Isso significa que o loop anterior descartará o fluxo de forma assíncrona quando o loop terminar. Você pode imaginar que o loop se parece com o seguinte código:

```
C#

int num = 0;
var enumerator = RunPagedQueryAsync(client, PagedIssueQuery,
"docs").GetAsyncEnumerator();
try
{
    while (await enumerator.MoveNextAsync())
    {
        var issue = enumerator.Current;
        Console.WriteLine(issue);
        Console.WriteLine($"Received {++num} issues in total");
    }
} finally
{
    if (enumerator != null)
        await enumerator.DisposeAsync();
}
```

Por padrão, os elementos de fluxo são processados no contexto capturado. Se você quiser desabilitar a captura do contexto, use o TaskAsyncEnumerableExtensions.ConfigureAwait método de extensão. Para obter mais informações sobre contextos de sincronização e captura do contexto atual, consulte o artigo sobre como consumir o padrão assíncrono baseado em tarefa.

Os fluxos assíncronos dão suporte ao cancelamento usando o mesmo protocolo que outros `async` métodos. Você modificaria a assinatura para o método iterador assíncrono da seguinte maneira para dar suporte ao cancelamento:

```
C#

private static async IAsyncEnumerable<JToken> RunPagedQueryAsync(GitHubClient
client,
```

```
    string queryText, string repoName, [EnumeratorCancellation] CancellationToken
cancellationToken = default)
{
    var issueAndPRQuery = new GraphQLRequest
    {
        Query = queryText
    };
    issueAndPRQuery.Variables["repo_name"] = repoName;

    bool hasMorePages = true;
    int pagesReturned = 0;
    int issuesReturned = 0;

    // Stop with 10 pages, because these are large repos:
    while (hasMorePages && (pagesReturned++ < 10))
    {
        var postBody = issueAndPRQuery.ToJsonText();
        var response = await client.Connection.Post<string>(new
Uri("https://api.github.com/graphql"),
            postBody, "application/json", "application/json");

        JObject results = JObject.Parse(response.HttpResponse.Body.ToString()!);

        int totalCount = (int)issues(results)["totalCount"]!;
        hasMorePages = (bool)pageInfo(results)["hasPreviousPage"]!;
        issueAndPRQuery.Variables["start_cursor"] = pageInfo(results)
["startCursor"]!.ToString();
        issuesReturned += issues(results)["nodes"]!.Count();

        foreach (JObject issue in issues(results)["nodes"]!)
            yield return issue;
    }

    JObject issues(JObject result) => (JObject)result["data"]!["repository"]!
["issues"]!;
    JObject pageInfo(JObject result) => (JObject)issues(result)["pageInfo"]!;
}
```

O atributo System.Runtime.CompilerServices.EnumeratorCancellationAttribute faz com que o compilador gere código para o IAsyncEnumerator<T> que torne o token passado para `GetAsyncEnumerator` visível ao corpo do iterador assíncrono como um argumento. Em `runQueryAsync`, você pode examinar o estado do token e cancelar mais trabalhos, se solicitado.

Você usa outro método de extensão, WithCancellation, para passar o token de cancelamento para o fluxo assíncrono. Você modificaria o loop enumerando os problemas da seguinte maneira:

C#

```
private static async Task EnumerateWithCancellation(GitHubClient client)
{
```

```
    int num = 0;
    var cancellation = new CancellationTokenSource();
    await foreach (var issue in RunPagedQueryAsync(client, PagedIssueQuery,
"docs")
        .WithCancellation(cancellation.Token))
    {
        Console.WriteLine(issue);
        Console.WriteLine($"Received {++num} issues in total");
    }
}
```

Você pode obter o código para o tutorial concluído no repositório dotnet/docs na pasta assíncrona-programação/snippets .

Executar o aplicativo concluído

Execute o aplicativo novamente. Contraste seu comportamento com o comportamento do aplicativo inicial. A primeira página de resultados é enumerada assim que está disponível. Há uma pausa observável à medida que cada nova página é solicitada e recuperada e, em seguida, os resultados da próxima página são enumerados rapidamente. O `try` / `catch` bloco não é necessário para lidar com o cancelamento: o chamador pode parar de enumerar a coleção. O progresso é claramente relatado porque o fluxo assíncrono gera resultados conforme cada página é baixada. O status de cada problema retornado está incluído diretamente no loop `await foreach`. Você não precisa de um objeto de retorno de chamada para acompanhar o progresso.

Você pode ver melhorias no uso da memória examinando o código. Você não precisa mais alocar uma coleção para armazenar todos os resultados antes que eles sejam enumerados. O chamador pode determinar como consumir os resultados e se uma coleção de armazenamento é necessária.

Execute os aplicativos iniciados e concluídos e você pode observar as diferenças entre as implementações por conta própria. Você pode excluir o token de acesso do GitHub que criou quando iniciou este tutorial após a conclusão. Se um invasor tiver acesso a esse token, ele poderá acessar as APIs do GitHub usando suas credenciais.

Neste tutorial, você usou fluxos assíncronos para ler itens individuais de uma API de rede que retorna páginas de dados. Fluxos assíncronos também podem ler de "fluxos sem fim", como um ticker de ações ou dispositivo sensor. A chamada para `MoveNextAsync` retorna o próximo item assim que ele estiver disponível.

Tipos de referência anuláveis

22/06/2025

Tipos de referência anuláveis são um grupo de recursos que minimizam a probabilidade de o código fazer com que o runtime gere System.NullReferenceException. Três recursos que ajudam você a evitar essas exceções, incluindo a capacidade de marcar explicitamente um tipo de referência como anulável:

- Análise aprimorada do fluxo estático que determina se uma variável pode ser `null` antes de desreferenciá-la.
- Atributos que anotam APIs para que a análise de fluxo determine o *estado nulo*.
- Anotações variáveis que os desenvolvedores usam para declarar explicitamente o *null-state* pretendido para uma variável.

O compilador rastreia o *estado nulo* de cada expressão em seu código no momento da compilação. O estado nulo *do* tem um dos dois valores:

- *Não nulo*: a expressão é conhecida como não `null`.
- *possivelmente nulo*: a expressão pode ser `null`.

As anotações de variáveis determinam a *nulidade* de uma variável de tipo de referência:

- *não anulável*: se você atribuir um valor `null` ou uma expressão *possivelmente nula* à variável, o compilador emitirá um aviso. As variáveis que são *não anuláveis* têm um estado nulo padrão de *não nulo*.
- *anulável*: você pode atribuir um valor `null` ou uma expressão *possivelmente nula* à variável. Quando o estado nulo da variável é *possivelmente nulo*, o compilador emite um aviso se você desreferenciar a variável. O estado nulo padrão da variável é *possivelmente nulo*.

O restante deste artigo descreve como essas três áreas de recursos funcionam para produzir avisos quando o código pode estar **desreferenciando** um valor `null`. Desreferenciar uma variável significa acessar um de seus membros usando o operador `.` (ponto), conforme mostrado no exemplo a seguir:

C#

```
string message = "Hello, World!";
int length = message.Length; // dereferencing "message"
```

Quando você desreferencia uma variável cujo valor é `null`, o runtime gera um System.NullReferenceException.

Da mesma forma, pode-se produzir avisos quando a notação `[]` é usada para acessar um membro de um objeto quando o objeto está em `null`:

```
C#

using System;

public class Collection<T>
{
    private T[] array = new T[100];
    public T this[int index]
    {
        get => array[index];
        set => array[index] = value;
    }
}

public static void Main()
{
    Collection<int> c = default;
    c[10] = 1;    // CS8602: Possible dereference of null
}
```

Você saberá mais sobre:

- A análise do estado nulo do compilador: como o compilador determina se uma expressão é não nula ou possivelmente nula.
- Atributos que são aplicados às APIs que fornecem mais contexto para a análise de estado nulo do compilador.
- Anotações de variáveis anuláveis que fornecem informações sobre sua intenção com relação às variáveis. As anotações são úteis para campos, parâmetros e valores retornados para definir o estado nulo padrão.
- As regras que regem os argumentos de tipo genérico. Novas restrições foram adicionadas porque os parâmetros de tipo podem ser tipos de referência ou tipos de valor. O sufixo `?` é implementado de forma diferente para tipos de valor anuláveis e tipos de referência anulável.
- O Contexto anulável ajuda você a migrar grandes projetos. Você pode habilitar avisos e anotações no contexto anulável em partes do seu aplicativo conforme migra. Depois de resolver mais avisos, você pode habilitar ambas as configurações para todo o projeto.

Por fim, você aprenderá as armadilhas conhecidas da análise de estado nulo em tipos `struct` e matrizes.

Você também pode explorar esses conceitos no módulo do Learn sobre Segurança de anuláveis em C#.

Análise de estado nulo

A análise de estado nulo rastreia o *estado nulo* das referências. Uma expressão é *não nulo* ou *talvez nulo*. O compilador determina que uma variável é *not-null* de duas maneiras:

1. A variável recebeu um valor conhecido por *não ser nulo*.
2. A variável foi verificada em relação a `null` e não foi atribuída desde essa verificação.

Qualquer variável que o compilador não possa determinar como não nula é considerada talvez nula. A análise fornece avisos em situações em que você pode acidentalmente desreferenciar um valor `null`. O compilador produz avisos com base no *estado nulo*.

- Quando uma variável é não nula, essa variável pode ser desreferenciada com segurança.
- Quando uma variável é *maybe-null*, essa variável deve ser verificada para garantir que não seja `null` antes de ser desreferenciada.

Considere o seguinte exemplo:

C#

```
string? message = null;

// warning: dereference null.
Console.WriteLine($"The length of the message is {message.Length}");

var originalMessage = message;
message = "Hello, World!";

// No warning. Analysis determined "message" is not-null.
Console.WriteLine($"The length of the message is {message.Length}");

// warning!
Console.WriteLine(originalMessage.Length);
```

No exemplo anterior, o compilador determina que `message` é *maybe-null* quando a primeira mensagem é impressa. Não há nenhum aviso para a segunda mensagem. A linha final de código produz um aviso, já que `originalMessage` pode ser nula. O exemplo a seguir mostra um uso mais prático para percorrer uma árvore de nós até a raiz, processando cada nó durante a passagem:

C#

```
void FindRoot(Node node, Action<Node> processNode)
{
    for (var current = node; current != null; current = current.Parent)
    {
        processNode(current);
```

```
    }
}
```

O código anterior não gera avisos para desreferenciar a variável `current`. A análise estática determina que `current` nunca é desreferenciada quando é *maybe-null*. A variável `current` é verificada em relação a `null` antes de `current.Parent` ser acessada e antes de passar `current` para a ação `ProcessNode`. Os exemplos anteriores mostram como o compilador determina o *null-state* para variáveis locais quando inicializado, atribuído ou comparado a `null`.

A análise de estado nulo não rastreia os métodos chamados. Como resultado, os campos inicializados em um método auxiliar comum chamado por todos os construtores podem gerar um aviso com a seguinte mensagem:

> A propriedade '*name*' não anulável deve conter um valor não nulo ao sair do construtor.

Você pode resolver esses avisos de duas maneiras: *encadeamento de construtores* ou *atributos anuláveis* no método auxiliar. O código a seguir mostra um exemplo de cada um desses casos. A classe `Person` usa um construtor comum chamado por todos os outros construtores. A classe `Student` tem um método auxiliar anotado com o atributo System.Diagnostics.CodeAnalysis.MemberNotNullAttribute:

C#

```
using System.Diagnostics.CodeAnalysis;

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

    public Person(string firstName, string lastName)
    {
        FirstName = firstName;
        LastName = lastName;
    }

    public Person() : this("John", "Doe") { }
}

public class Student : Person
{
    public string Major { get; set; }

    public Student(string firstName, string lastName, string major)
        : base(firstName, lastName)
    {
```

```
        SetMajor(major);
    }

    public Student(string firstName, string lastName) :
        base(firstName, lastName)
    {
        SetMajor();
    }

    public Student()
    {
        SetMajor();
    }

    [MemberNotNull(nameof(Major))]
    private void SetMajor(string? major = default)
    {
        Major = major ?? "Undeclared";
    }
}
```

Análise de estado anulável e os avisos gerados pelo compilador ajudam você a evitar erros de programa ao desreferenciar `null`. O artigo sobre resolução de avisos anuláveis fornece técnicas para corrigir os avisos mais prováveis de serem vistos em seu código. Os diagnósticos produzidos a partir da análise de estado nulo são apenas avisos.

Atributos em assinaturas de API

A análise de estado nulo precisa de dicas dos desenvolvedores para entender a semântica das APIs. Algumas APIs fornecem verificações nulas e devem alterar o *null-state* de uma variável de *maybe-null* para *not-null*. Outras APIs retornam expressões que são *not-null* ou *maybe-null*, dependendo do *null-state* dos argumentos de entrada. Por exemplo, considere o seguinte código que exibe uma mensagem em letras maiúsculas:

C#

```
void PrintMessageUpper(string? message)
{
    if (!IsNull(message))
    {
        Console.WriteLine($"{DateTime.Now}: {message.ToUpper()}");
    }
}

bool IsNull(string? s) => s == null;
```

Com base na inspeção, qualquer desenvolvedor consideraria esse código seguro e não deveria gerar avisos. Entretanto, o compilador não sabe que `IsNull` fornece uma verificação nula e emite um aviso para a instrução `message.ToUpper()`, considerando `message` como uma variável *possivelmente nula*. Use o atributo NotNullWhen para corrigir esse aviso:

```
C#

bool IsNull([NotNullWhen(false)] string? s) => s == null;
```

Esse atributo informa ao compilador que, se `IsNull` retornar `false`, o parâmetro `s` não será nulo. O compilador altera o *estado nulo* de `message` para *não nulo* dentro do bloco `if (!IsNull(message)) {...}`. Nenhum aviso é emitido.

Os atributos fornecem informações detalhadas sobre o estado nulo dos argumentos, valores de retorno e membros da instância do objeto usada para invocar um membro. Os detalhes sobre cada atributo podem ser encontrados no artigo de referência de idioma sobre atributos de referência anuláveis. A partir do .NET 5, todas as APIs de runtime do .NET serão anotadas. Você aprimora a análise estática anotando suas APIs para fornecer informações semânticas sobre o *null-state* dos argumentos e os valores retornados.

Anotações de variáveis anuláveis

A análise de *estado nulo* fornece uma análise robusta para variáveis locais. O compilador precisa de mais informações suas para variáveis-membro. O compilador precisa de mais informações para definir o *estado nulo* de todos os campos no colchete de abertura de um membro. Qualquer um dos construtores acessíveis pode ser usado para inicializar o objeto. Se um campo membro for definido como `null`, o compilador deve assumir que seu *null-state* seja *maybe-null* no início de cada método.

Você usa anotações que podem declarar se uma variável é um **tipo de referência anulável** ou um **tipo de referência não anulável**. Essas anotações fazem instruções importantes sobre o *null-state* das variáveis:

- **Uma referência não deve ser nula**. O estado padrão de uma variável de referência não anulável é *não nulo*. O compilador impõe regras que garantem que seja seguro desreferenciar essas variáveis sem verificar primeiro que ela não está nula:
  - A variável deve ser inicializada para um valor não nulo.
  - A variável nunca pode receber o valor `null`. O compilador emite um aviso quando o código atribui uma expressão *maybe-null* a uma variável que não deve ser nula.
- **Uma referência pode ser nula**. O estado padrão de uma variável de referência anulável é *maybe-null*. O compilador impõe regras para garantir que você verifique corretamente se

há uma referência `null`:

- A variável só pode ser desreferenciada quando o compilador puder garantir que o valor não seja `null`.
- Essas variáveis podem ser inicializadas com o valor de `null` padrão e podem receber o valor `null` em outro código.
- O compilador não emite avisos quando o código atribui uma expressão *possivelmente nula* a uma variável que pode ser nula.

Qualquer variável de referência não anulável tem o *estado nulo* inicial de *não nulo*. Qualquer variável de referência anulável tem o estado inicial *nulo* de *possivelmente nulo*.

Um **tipo de referência que permite valor nulo** é indicado usando a mesma sintaxe que tipos de valor que permitem valor nulo: um `?` é acrescentado ao tipo da variável. Por exemplo, a seguinte declaração de variável representa uma variável string nula, `name`:

```
C#

string? name;
```

Quando os tipos de referência anuláveis estiverem habilitados, qualquer variável em que o `?` não esteja anexado ao nome do tipo é um tipo de referência **não anulável**. Isso inclui todas as variáveis de tipo de referência no código existente quando você habilitar esse recurso. No entanto, todas as variáveis locais tipadas implicitamente (declaradas usando `var`) são **tipos de referência anuláveis**. Como as seções anteriores mostraram, a análise estática determina o *estado nulo* das variáveis locais para verificar se elas estão *possivelmente nulas* antes de desreferenciá-las.

Às vezes, você deve ignorar um aviso quando souber que uma variável não é nula, mas o compilador determina que o seu *null-state* é *maybe-null*. Você usa o operador null-forgiving `!` seguindo um nome de variável para forçar o *estado nulo* para ser *não nulo*. Por exemplo, se você sabe que a variável `name` não é `null`, mas o compilador emite um aviso, é possível escrever o seguinte código para substituir a análise do compilador:

```
C#

name!.Length;
```

Os tipos de referência anuláveis e os tipos de valor anuláveis fornecem um conceito semântico semelhante: uma variável pode representar um valor ou objeto, ou essa variável pode ser `null`. No entanto, tipos de referência anuláveis e tipos de valor anulável são implementados de forma diferente: tipos de valor anuláveis são implementados usando System.Nullable<T> e tipos de referência anuláveis são implementados por atributos lidos pelo compilador. Por

exemplo, `string?` e `string` são ambos representados pelo mesmo tipo: System.String. No entanto, `int?` e `int` são representados por `System.Nullable<System.Int32>` e System.Int32, respectivamente.

Tipos de referência anuláveis são um recurso de tempo de compilação. Isso significa que é possível que os chamadores ignorem avisos, usar `null` intencionalmente como um argumento para um método que espera uma referência não anulável. Os desenvolvedores de bibliotecas devem incluir as verificações em tempo de execução contra valores de argumento nulos. O ArgumentNullException.ThrowIfNull é a opção preferencial para verificar um parâmetro em relação ao nulo no tempo de execução. Além disso, o comportamento de runtime de um programa que usa anotações anuláveis é o mesmo se todas as anotações anuláveis, (`?` e `!`), forem removidas. Sua única finalidade é expressar a intenção de design e fornecer informações para análise de estado nulo.

> ⓘ **Importante**
>
> Habilitar anotações anuláveis pode alterar a forma como o Entity Framework Core determina se um membro de dados é necessário. Você pode conferir mais detalhes no artigo sobre **Conceitos básicos do Entity Framework Core: Trabalhando com tipos de referência anuláveis**.

Genéricos

Os genéricos exigem regras detalhadas para lidar com `T?` para qualquer parâmetro de tipo `T`. As regras são necessariamente detalhadas devido ao histórico e à implementação diferente para um tipo de valor anulável e um tipo de referência anulável. Tipos de valor anuláveis são implementados usando o struct System.Nullable<T>. Tipos de referência anuláveis são implementados como anotações de tipo que fornecem regras semânticas para o compilador.

- Se o argumento de tipo para `T` for um tipo de referência, `T?` referencia o tipo de referência anulável correspondente. Por exemplo, se `T` for um `string`, então `T?` será um `string?`.
- Se o argumento de tipo `T` for um tipo de valor, `T?` referencia o mesmo tipo de valor, `T`. Por exemplo, se `T` for um `int`, o `T?` também será um `int`.
- Se o argumento de tipo para `T` for um tipo de referência anulável, `T?` referencia o mesmo tipo de referência anulável correspondente. Por exemplo, se `T` for um `string?`, então `T?` será um `string?`.
- Se o argumento de tipo para `T` for um tipo de valor anulável, `T?` referencia o mesmo tipo de valor anulável correspondente. Por exemplo, se `T` for um `int?`, então `T?` será um

`int?`.

Para valores retornados, `T?` é equivalente a `[MaybeNull]T`; para valores de argumento, `T?` é equivalente a `[AllowNull]T`. Para obter mais informações, consulte o artigo sobre Atributos para análise de null-state na referência da linguagem.

Você pode especificar um comportamento diferente usando restrições:

- A restrição `class` significa que `T` deve ser um tipo de referência não anulável (por exemplo, `string`). O compilador produz um aviso se você usar um tipo de referência anulável, como `string?` para `T`.
- A restrição `class?` significa que `T` deve ser um tipo de referência, não anulável (`string`) ou um tipo de referência anulável (por exemplo, `string?`). Quando o parâmetro de tipo é um tipo de referência anulável, como `string?`, uma expressão de `T?` referencia esse mesmo tipo de referência anulável, como `string?`.
- A restrição `notnull` significa que `T` deve ser um tipo de referência não anulável ou um tipo de valor não anulável. Se você usar um tipo de referência anulável ou um tipo de valor anulável para o parâmetro de tipo, o compilador produzirá um aviso. Além disso, quando `T` é um tipo de valor, o valor retornado é esse tipo de valor, não o tipo de valor anulável correspondente.

Essas restrições ajudam a fornecer mais informações ao compilador sobre como `T` é usado. Isso ajuda quando os desenvolvedores escolhem o tipo para `T` e proporciona uma melhor análise de *estado nulo* quando uma instância de tipo genérico é usada.

Contexto que permite valor nulo

O *contexto anulável* determina como as anotações de tipo de referência anulável são tratadas e quais avisos são produzidos pela análise de estado nulo estático. O contexto anulável contém dois sinalizadores: a configuração de *anotação* e a configuração de *aviso*.

As configurações de *anotação* e *aviso* são desabilitadas por padrão para projetos existentes. A partir do .NET 6 (C# 10), os dois sinalizadores são habilitados por padrão para *novos* projetos. O motivo para dois sinalizadores distintos para o contexto anulável é facilitar a migração de grandes projetos que são anteriores à introdução de tipos de referência anuláveis.

Em projetos pequenos, você pode habilitar tipos de referência anuláveis, corrigir avisos e continuar. No entanto, para projetos maiores e soluções de vários projetos, isso pode gerar um grande número de avisos. Você pode usar pragmas para habilitar tipos de referência anuláveis arquivo por arquivo à medida que começar a usar os tipos de referência anuláveis. Os novos

recursos que protegem contra o lançamento de um System.NullReferenceException podem ser disruptivos quando ativados em uma base de código existente:

- Todas as variáveis de referência tipadas explicitamente são interpretadas como tipos de referência não anuláveis.
- O significado da restrição `class` em genéricos foi alterado para significar um tipo de referência não anulável.
- Novos avisos são gerados devido a essas novas regras.

O **contexto de anotação anulável** determina o comportamento do compilador. Há quatro combinações para as configurações de **contexto anulável**.

- *ambos desabilitados*: o código é *nullable-oblivious*. *Desabilitar* corresponde ao comportamento anterior à habilitação dos tipos de referência anuláveis, exceto pelo fato de que a nova sintaxe produz avisos em vez de erros.
  - Avisos anuláveis estão desabilitados.
  - Todas as variáveis de tipo de referência são tipos de referência anuláveis.
  - O uso do sufixo `?` para declarar um tipo de referência anulável produz um aviso.
  - Você pode usar o operador tolerante a nulo, `!`, mas ele não tem efeito.
- *ambos habilitados*: o compilador habilita toda a análise de referência nula e todos os recursos de linguagem.
  - Todos os novos avisos anuláveis estão habilitados.
  - Você pode usar o sufixo `?` para declarar um tipo de referência anulável.
  - As variáveis de tipo de referência sem o sufixo `?` são tipos de referência não anuláveis.
  - O operador de tolerância a nulo suprime avisos para uma possível desreferência a `null`.
- *aviso habilitado*: o compilador executa todas as análises nulas e emite avisos quando o código pode desreferenciar `null`.
  - Todos os novos avisos anuláveis estão habilitados.
  - O uso do sufixo `?` para declarar um tipo de referência anulável produz um aviso.
  - Todas as variáveis de tipo de referência têm permissão para serem nulas. No entanto, os membros têm o *null-state* de *not-null* na chave de abertura de todos os métodos, a menos que declarados com o sufixo `?`.
  - Você pode usar o operador de tolerância a nulo, `!`.
- *anotações habilitadas*: o compilador não gera avisos quando o código pode desreferenciar `null`, ou quando você atribui uma expressão possivelmente nula a uma variável não anulável.
  - Todos os avisos anuláveis estão desabilitados.
  - Você pode usar o sufixo `?` para declarar um tipo de referência anulável.
  - As variáveis de tipo de referência sem o sufixo `?` são tipos de referência não anuláveis.
  - Você pode usar o operador tolerante a nulo, `!`, mas ele não tem efeito.

O contexto de anotação anulável e o contexto de aviso anulável podem ser definidos para um projeto usando o <Nullable> elemento em seu arquivo *.csproj*. Esse elemento configura como o compilador interpreta a nulidade de tipos e quais avisos são gerados. A tabela a seguir mostra os valores permitidos e resume os contextos que eles especificam.

Expandir a tabela

Contexto	Avisos de desreferência	Avisos de atribuição	Tipos de referência	sufixo `?`	operador `!`
`disable`	Desabilitado	Desabilitado	Todos são anuláveis	Produz um aviso	Não tem efeito
`enable`	Ativado	Ativado	Não anulável, a menos que declarado com `?`	Declara tipo anulável	Suprime avisos para uma possível atribuição `null`
`warnings`	Ativado	Não aplicável	Todos são anuláveis, mas os membros são considerados *não nulos* na abertura dos métodos	Produz um aviso	Suprime avisos para uma possível atribuição `null`
`annotations`	Desabilitado	Desabilitado	Não anulável, a menos que declarado com `?`	Declara tipo anulável	Não tem efeito

As variáveis de tipo de referência no código compilado em um contexto *disabled* são *nullable-oblivious*. Você pode atribuir um literal `null` ou uma variável *possivelmente nula* para uma variável que seja *alheia anulável*. No entanto, o estado padrão de uma variável *nullable-oblivious* é *not-null*.

Você pode escolher qual configuração é melhor para seu projeto:

- Escolha *disable* em projetos herdados que você não deseja atualizar com base no diagnóstico ou em novos recursos.
- Escolha *avisos* para determinar o local em que seu código poderá gerar System.NullReferenceExceptions. Você pode resolver esses avisos antes de modificar o código para habilitar tipos de referência não anuláveis.
- Escolha *annotations* para expressar sua intenção de design antes de habilitar avisos.
- Escolha *enable* para novos projetos e projetos ativos em que você deseja proteger contra exceções de referência nulas.

Exemplo:

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

Você também pode usar diretivas para definir esses mesmos sinalizadores em qualquer lugar no código-fonte. Essas diretivas são mais úteis quando você está migrando uma grande base de código.

- `#nullable enable`: define os sinalizadores de anotação e aviso como **habilitar**.
- `#nullable disable`: define os sinalizadores de anotação e aviso como **desabilitar**.
- `#nullable restore`: restaura o sinalizador de anotação e o sinalizador de aviso para as configurações do projeto.
- `#nullable disable warnings`: defina o sinalizador de aviso como **desabilitar**.
- `#nullable enable warnings`: defina o sinalizador de aviso como **habilitar**.
- `#nullable restore warnings`: restaura o sinalizador de aviso para as configurações do projeto.
- `#nullable disable annotations`: defina o sinalizador de anotação como **desabilitar**.
- `#nullable enable annotations`: defina o sinalizador de anotação como **habilitar**.
- `#nullable restore annotations`: restaura o sinalizador de anotação para as configurações do projeto.

Para qualquer linha de código, você pode definir qualquer uma das seguintes combinações:

Expandir a tabela

Sinalizador de aviso	Sinalizador de anotação	Utilização
Padrão do projeto	Padrão do projeto	Padrão
habilitar	desabilitar	Corrigir avisos da análise
habilitar	Padrão do projeto	Corrigir avisos da análise
Padrão do projeto	habilitar	Adicionar anotações de tipo
habilitar	habilitar	Código já migrado
desabilitar	habilitar	Anotar código antes de corrigir avisos
desabilitar	desabilitar	Adicionando código herdado ao projeto migrado
Padrão do projeto	desabilitar	Raramente
desabilitar	Padrão do projeto	Raramente

Essas nove combinações fornecem controle refinado sobre os diagnósticos que o compilador emite para seu código. Você pode habilitar mais recursos em qualquer área que esteja atualizando, sem ver mais avisos que você ainda não está pronto para resolver.

> ⓘ **Importante**
>
> O contexto global anulável não se aplica aos arquivos de código gerados. Em qualquer das estratégias, o contexto anulável é *desabilitado* para todo arquivo de origem marcado como gerado. Isso significa que todas as APIs em arquivos gerados não são anotadas. Nenhum aviso anulável é produzido para arquivos gerados. Há quatro maneiras de um arquivo ser marcado como gerado:
>
> 1. No .editorconfig, especifique `generated_code = true` em uma seção que se aplica a esse arquivo.
> 2. Coloque `<auto-generated>` ou `<auto-generated/>` em um comentário na parte superior do arquivo. Ele pode estar em qualquer linha nesse comentário, mas o bloco de comentários deve ser o primeiro elemento do arquivo.
> 3. Inicie o nome do arquivo com *TemporaryGeneratedFile_*
> 4. Termine o nome do arquivo com *.designer.cs*, *.generated.cs*, *.g.cs* ou *.g.i.cs*.
>
> Os geradores podem optar por participar usando a diretiva de pré-processador **#nullable**.

Por padrão, os sinalizadores de aviso e de anotação de anulável são **desabilitados**. Isso significa que seu código existente compila sem alterações e sem gerar nenhum aviso novo. A partir do .NET 6, novos projetos incluem o elemento `<Nullable>enable</Nullable>` em todos os modelos de projeto, configurando esses sinalizadores como **ativado**.

Essas opções fornecem duas estratégias distintas para atualizar uma base de código existente para usar tipos de referência anuláveis.

Armadilhas conhecidas

Matrizes e structs que contêm tipos de referência são armadilhas conhecidas em referências anuláveis e na análise estática que determina a segurança nula. Em ambas as situações, uma referência não anulável pode ser inicializada como `null`, sem gerar avisos.

Estruturas

Um struct que contém tipos de referência não anuláveis permite atribuir `default` para ele sem avisos. Considere o seguinte exemplo:

C#

```
using System;

#nullable enable

public struct Student
{
    public string FirstName;
    public string? MiddleName;
    public string LastName;
}

public static class Program
{
    public static void PrintStudent(Student student)
    {
        Console.WriteLine($"First name: {student.FirstName.ToUpper()}");
        Console.WriteLine($"Middle name: {student.MiddleName?.ToUpper()}");
        Console.WriteLine($"Last name: {student.LastName.ToUpper()}");
    }

    public static void Main() => PrintStudent(default);
}
```

No exemplo anterior, não há nenhum aviso em `PrintStudent(default)`, enquanto os tipos de referência não anuláveis `FirstName` e `LastName` são nulos.

Outro caso mais comum é quando você lida com structs genéricos. Considere o seguinte exemplo:

C#

```
#nullable enable

public struct S<T>
{
    public T Prop { get; set; }
}

public static class Program
{
    public static void Main()
    {
        string s = default(S<string>).Prop;
    }
}
```

No exemplo anterior, a propriedade `Prop` é `null` em tempo de execução. Ele é atribuído a uma cadeia de caracteres não nula sem nenhum aviso.

matrizes

Matrizes também são uma armadilha conhecida em tipos de referência anuláveis. Considere o exemplo a seguir que não produz avisos:

C#

```
using System;

#nullable enable

public static class Program
{
    public static void Main()
    {
        string[] values = new string[10];
        string s = values[0];
        Console.WriteLine(s.ToUpper());
    }
}
```

No exemplo anterior, a declaração da matriz mostra que ela contém cadeias de caracteres não anuláveis, enquanto seus elementos são inicializados para `null`. Em seguida, a variável `s` recebe um valor (`null` o primeiro elemento da matriz). Por fim, a variável `s` é desreferenciada, causando uma exceção de runtime.

Construtores

O construtor de uma classe ainda chamará o finalizador, mesmo quando houver uma exceção lançada por esse construtor.
O exemplo a seguir demonstra esse comportamento:

C#

```
public class A
{
    private string _name;
    private B _b;

    public A(string name)
    {
        ArgumentNullException.ThrowIfNullOrEmpty(name);
        _name = name;
        _b = new B();
    }

  ~A()
  {
```

```
        Dispose();
    }

    public void Dispose()
    {
        _b.Dispose();
        GC.SuppressFinalize(this);
    }
}

public class B: IDisposable
{
    public void Dispose() { }
}

public void Main()
{
    var a = new A(string.Empty);
}
```

No exemplo anterior, o System.NullReferenceException será gerado quando `_b.Dispose();` for executado, se o parâmetro `name` tiver sido `null`. A chamada para `_b.Dispose();` nunca será lançada quando o construtor for concluído com êxito. No entanto, não há nenhum aviso emitido pelo compilador, pois a análise estática não pode determinar se um método (como um construtor) é concluído sem que uma exceção de runtime seja gerada.

Confira também

- Especificação de tipos de referência anuláveis
- Anotações de parâmetro de tipo sem restrição
- Introdução ao tutorial de referências anuláveis
- **Anulável** (Opçõa do compilador C#)
- CS8602: possível desreferência de aviso nulo

Atualizar uma base de código com tipos de referência anuláveis para melhorar os avisos de diagnóstico nulos

Artigo • 07/04/2023

Tipos de referência anuláveis permitem declarar se variáveis de um tipo de referência devem ou não ser atribuídas a um valor `null`. A análise estática e os avisos do compilador quando seu código pode desreferenciar `null` são o benefício mais importante desse recurso. Depois de habilitado, o compilador gera avisos que ajudam você a evitar a geração de System.NullReferenceException quando o código é executado.

Se a base de código for relativamente pequena, você poderá ativar o recurso em seu projeto, endereçar avisos e aproveitar os benefícios do diagnóstico aprimorado. Bases de código maiores podem exigir uma abordagem mais estruturada para endereçar avisos ao longo do tempo, habilitando o recurso para alguns, conforme você aborda avisos em diferentes tipos ou arquivos. Este artigo descreve diferentes estratégias para atualizar uma base de código e as compensações associadas a essas estratégias. Antes de iniciar a migração, leia a visão geral conceitual dos tipos de referência anuláveis. Ele aborda a análise estática do compilador, valores do *estado de nulo* iguais a *talvez nulo* e *não nulo* e anotações anuláveis. Depois de conhecer esses conceitos e termos, você estará pronto para migrar seu código.

Planeje sua migração

Independentemente de como você atualiza sua base de código, a meta é que avisos anuláveis e anotações anuláveis sejam habilitados em seu projeto. Depois de atingir essa meta, você terá a configuração `<nullable>Enable</nullable>` em seu projeto. Você não precisará de nenhuma das diretivas de pré-processador para ajustar as configurações em outro lugar.

A primeira opção é definir o padrão do projeto. Suas opções são:

1. ***Anulável desabilitado como padrão***: *Disable* será o padrão se você não adicionar um elemento `Nullable` ao arquivo de projeto. Use esse padrão quando não estiver ativamente adicionando novos arquivos à base de código. A atividade principal é atualizar a biblioteca para usar tipos de referência anuláveis. Usar esse padrão significa que você adiciona uma diretiva de pré-processador anulável a cada arquivo à medida que atualiza o código deles.

2. ***Anulável habilitado como padrão***: defina esse padrão quando estiver desenvolvendo ativamente novos recursos. Você deseja que todos os novos códigos beneficiem tipos de referência anuláveis e análise estática anulável. Usar esse padrão significa que você deve adicionar `#nullable disable` à parte superior de cada arquivo. Você removerá essas diretivas de pré-processador ao resolver os avisos em cada arquivo.
3. ***Avisos anuláveis como padrão***: escolha esse padrão para uma migração em duas fases. Na primeira fase, aborde os avisos. Na segunda fase, ative as anotações para declarar o *estado de nulo* esperado de uma variável. Usar esse padrão significa que você deve adicionar `#nullable disable` à parte superior de cada arquivo.
4. ***Anotações anuláveis*** como padrão. Faça anotações no código antes de abordar os avisos.

Habilitar anulável como padrão cria um trabalho mais inicial para adicionar as diretivas de pré-processador a cada arquivo. A vantagem é que cada novo arquivo de código adicionado ao projeto será habilitado para anulável. Qualquer novo trabalho será habilitado para anulável; somente o código existente deve ser atualizado. Desabilitar anulável como o padrão funciona melhor se a biblioteca estiver estável e o foco principal do desenvolvimento for adotar tipos de referência anuláveis. Você ativa tipos de referência anuláveis à medida que anota as APIs. Ao terminar, você habilita os tipos de referência anuláveis para o projeto inteiro. Ao criar um arquivo, você deve adicionar as diretivas de pré-processador e habilitá-lo para anulável. Se algum desenvolvedor em sua equipe esquecer, esse novo código agora estará no backlog do trabalho para tornar todo o código habilitado para anulável.

A sua encolha entre essas estratégias depende da quantidade de desenvolvimento ativo que está ocorrendo em seu projeto. Quanto mais maduro e estável o projeto, melhor será a segunda estratégia. Quanto mais recursos sendo desenvolvidos, melhor será a primeira estratégia.

> **ⓘ Importante**
>
> O contexto global anulável não se aplica aos arquivos de código gerados. Em qualquer das estratégias, o contexto anulável é *desabilitado* para todo arquivo de origem marcado como gerado. Isso significa que todas as APIs em arquivos gerados não são anotadas. Há quatro maneiras de um arquivo ser marcado como gerado:
>
> 1. No .editorconfig, especifique `generated_code = true` em uma seção que se aplica a esse arquivo.

2. Coloque `<auto-generated>` ou `<auto-generated/>` em um comentário na parte superior do arquivo. Ele pode estar em qualquer linha nesse comentário, mas o bloco de comentários deve ser o primeiro elemento do arquivo.
3. Inicie o nome do arquivo com *TemporaryGeneratedFile_*
4. Termine o nome do arquivo com *.designer.cs*, *.generated.cs*, *.g.cs* ou *.g.i.cs*.

Os geradores podem aceitar usando a diretiva de pré-processador **#nullable**.

Entender contextos e avisos

Habilitar avisos e anotações controla como o compilador exibe tipos de referência e nulidade. Cada tipo tem uma entre três nulidades:

- *oblivious*: todos os tipos de referência são *nullable oblivious* quando o contexto de anotação está desabilitado.
- *nonnullable*: um tipo de referência não anotado, `C` é *nonnullable* quando o contexto de anotação está habilitado.
- *nullable*: um tipo de referência anotado, `C?`, é *nullable*, mas um aviso pode ser emitido quando o contexto de anotação está desabilitado. As variáveis declaradas com `var` são *nullable* quando o contexto de anotação está habilitado.

O compilador gera avisos com base nessa nulidade:

- Tipos *nonnullable* geram avisos se um valor potencial `null` é atribuído a eles.
- Tipos *nullable* geram avisos se são desreferenciados quando *talvez nulos*.
- Tipos *oblivious* geram avisos se são desreferenciados quando *talvez nulo* e o contexto de aviso está habilitado.

Cada variável tem um estado anulável padrão que depende de sua nulidade:

- As variáveis nullable têm um *estado de nulo* padrão igual a *talvez nulo*.
- As variáveis non-nullable têm um *estado de nulo* padrão igual a *não nulo*.
- As variáveis nullable oblivious têm um *estado de nulo* padrão igual a *não nulo*.

Antes de habilitar tipos de referência anuláveis, todas as declarações da sua base de código podem ser *nullable oblivious*. Isso é importante porque significa que todos os tipos de referência têm um *estado de nulo* padrão igual a *não nulo*.

Abordar avisos

Se o projeto usa o Entity Framework Core, você deve ler as diretrizes dele sobre Como trabalhar com tipos de referência anuláveis.

Ao iniciar a migração, você deve começar habilitando apenas avisos. Todas as declarações permanecem *nullable oblivious*, mas você verá avisos quando desreferenciar um valor após o *estado de nulo* dele mudar para *talvez nulo*. Ao abordar esses avisos, você verificará nulo em mais locais e sua base de código se tornará mais resiliente. Para aprender técnicas específicas adequadas a diferentes situações, confira o artigo sobre Técnicas para resolver avisos anuláveis.

Você pode abordar avisos e habilitar anotações em cada arquivo ou classe antes de continuar com outro código. No entanto, geralmente é mais eficiente resolver os avisos gerados enquanto o contexto é *avisos*, antes de habilitar as anotações de tipo. Assim, todos os tipos são *oblivious* até você abordar o primeiro conjunto de avisos.

Habilitar anotações de tipo

Depois de abordar o primeiro conjunto de avisos, você pode habilitar o *contexto de anotação*. Isso altera os tipos de referência de *oblivious* para *nonnullable*. Todas as variáveis declaradas com `var` são *anuláveis*. Essa alteração geralmente introduz novos avisos. A primeira etapa para abordar os avisos do compilador é usar anotações `?` em tipos de retorno e parâmetro para indicar quando os argumentos ou os valores retornados podem ser `null`. Ao realizar essa tarefa, sua meta não é apenas corrigir avisos. A meta mais importante é fazer com que o compilador entenda sua intenção com os possíveis valores nulos.

Atributos estendem anotações de tipo

Vários atributos foram adicionados para expressar informações adicionais sobre o estado de nulo das variáveis. As regras das suas APIs provavelmente são mais complicadas do que *não nulas* ou *talvez nulo* para todos os parâmetros e valores retornados. Muitas das suas APIs têm regras mais complexas para quando as variáveis podem ou não ser `null`. Nesses casos, você usará atributos para expressar essas regras. Os atributos que descrevem a semântica da sua API são encontrados no artigo sobre Atributos que afetam a análise anulável.

Próximas etapas

Depois de resolver todos os avisos após habilitar anotações, você pode definir o contexto padrão do seu projeto como *habilitado*. Se você adicionou pragmas ao seu

código para o contexto de anotações ou avisos anuláveis, você pode removê-los. Com o tempo, você pode ver novos avisos. Você pode escrever um código que introduz avisos. Uma dependência de biblioteca pode ser atualizada para tipos de referência anuláveis. Essas atualizações vão alterar os tipos dessa biblioteca de *nullable oblivious* para *nonnullable* ou *nullable*.

Você também pode explorar esses conceitos em nosso módulo Learn sobre segurança anulável em C#.

Métodos em C#

Artigo • 17/04/2025

Um método é um bloco de código que contém uma série de instruções. Um programa faz com que as instruções sejam executadas chamando o método e especificando os argumentos de método necessários. No C#, todas as instruções executadas são realizadas no contexto de um método.

> ⓘ **Observação**
>
> Este artigo discute métodos nomeados. Para obter informações sobre funções anônimas, consulte expressões lambda.

Assinaturas de método

Os métodos são declarados em `class`, `record` ou `struct` especificando:

- Um nível de acesso opcional, como `public` ou `private`. O padrão é `private`.
- Modificadores opcionais como `abstract` ou `sealed`.
- O valor retornado ou `void` se o método não tiver nenhum.
- O nome do método.
- Quaisquer parâmetros de método. Os parâmetros de método estão entre parênteses e separados por vírgulas. Parênteses vazios indicam que o método não requer parâmetros.

Essas partes juntas formam a assinatura do método.

> ⓘ **Importante**
>
> Um tipo de retorno de um método não faz parte da assinatura do método para fins de sobrecarga de método. No entanto, ele faz parte da assinatura do método ao determinar a compatibilidade entre um delegado e o método ao qual ele aponta.

O exemplo a seguir define uma classe chamada `Motorcycle` que contém cinco métodos:

C#

```
namespace MotorCycleExample
{
    abstract class Motorcycle
    {
        // Anyone can call this.
        public void StartEngine() {/* Method statements here */ }
```

```
        // Only derived classes can call this.
        protected void AddGas(int gallons) { /* Method statements here */ }

        // Derived classes can override the base class implementation.
        public virtual int Drive(int miles, int speed) { /* Method statements here
*/ return 1; }

        // Derived classes can override the base class implementation.
        public virtual int Drive(TimeSpan time, int speed) { /* Method statements
here */ return 0; }

        // Derived classes must implement this.
        public abstract double GetTopSpeed();
    }
```

A classe `Motorcycle` inclui um método sobrecarregado, `Drive`. Dois métodos têm o mesmo nome, mas suas listas de parâmetros os diferenciam.

Invocação de método

Os métodos podem ser *de instância* ou *estáticos*. Você deve criar uma instância de um objeto para invocar um método de instância nessa instância. Um método de instância opera nessa instância e seus dados. Você invoca um método estático referenciando o nome do tipo ao qual o método pertence; os métodos estáticos não operam nos dados da instância. Tentar chamar um método estático por meio de uma instância do objeto gera um erro do compilador.

Chamar um método é como acessar um campo. Após o nome do objeto (se você estiver chamando um método de instância) ou o nome do tipo (se você estiver chamando um método `static`), adicione um ponto, o nome do método e parênteses. Os argumentos são listados dentro dos parênteses e são separados por vírgulas.

A definição do método especifica os nomes e tipos de quaisquer parâmetros obrigatórios. Quando um chamador invoca o método, ele fornece valores concretos, chamados argumentos, para cada parâmetro. Os argumentos devem ser compatíveis com o tipo de parâmetro, mas o nome do argumento, se for usado no código de chamada, não precisa ser o mesmo do parâmetro nomeado definido no método. No exemplo a seguir, o `Square` método inclui um único parâmetro do tipo `int` chamado *i*. A primeira chamada de método passa ao `Square` método uma variável do tipo `int` nomeado *num*; a segunda, uma constante numérica; e a terceira, uma expressão.

C#

```
public static class SquareExample
{
```

```
    public static void Main()
    {
        // Call with an int variable.
        int num = 4;
        int productA = Square(num);

        // Call with an integer literal.
        int productB = Square(12);

        // Call with an expression that evaluates to int.
        int productC = Square(productA * 3);
    }

    static int Square(int i)
    {
        // Store input argument in a local variable.
        int input = i;
        return input * input;
    }
}
```

A forma mais comum de invocação de método usa argumentos posicionais, ela fornece os argumentos na mesma ordem que os parâmetros de método. Os métodos da classe `Motorcycle`, podem, portanto, ser chamados como no exemplo a seguir. A chamada para o método `Drive`, por exemplo, inclui dois argumentos que correspondem aos dois parâmetros na sintaxe do método. O primeiro se torna o valor do parâmetro `miles`. O segundo se torna o valor do parâmetro `speed`.

C#

```
class TestMotorcycle : Motorcycle
{
    public override double GetTopSpeed() => 108.4;

    static void Main()
    {
        var moto = new TestMotorcycle();

        moto.StartEngine();
        moto.AddGas(15);
        _ = moto.Drive(5, 20);
        double speed = moto.GetTopSpeed();
        Console.WriteLine($"My top speed is {speed}");
    }
}
```

Você também pode usar *argumentos nomeados* em vez de argumentos posicionais ao invocar um método. Ao usar argumentos nomeados, você especifica o nome do parâmetro seguido por dois pontos (":") e o argumento. Os argumentos do método podem aparecer em qualquer

ordem, desde que todos os argumentos necessários estejam presentes. O exemplo a seguir usa argumentos nomeados para invocar o método `TestMotorcycle.Drive`. Neste exemplo, os argumentos nomeados são passados na ordem oposta da lista de parâmetros do método.

C#

```
namespace NamedMotorCycle;

class TestMotorcycle : Motorcycle
{
    public override int Drive(int miles, int speed) =>
        (int)Math.Round((double)miles / speed, 0);

    public override double GetTopSpeed() => 108.4;

    static void Main()
    {
        var moto = new TestMotorcycle();
        moto.StartEngine();
        moto.AddGas(15);
        int travelTime = moto.Drive(speed: 60, miles: 170);
        Console.WriteLine($"Travel time: approx. {travelTime} hours");
    }
}
// The example displays the following output:
//      Travel time: approx. 3 hours
```

Você pode invocar um método usando argumentos posicionais e argumentos nomeados. No entanto, os argumentos posicionais só podem seguir argumentos nomeados quando os argumentos nomeados estiverem nas posições corretas. O exemplo a seguir invoca o método `TestMotorcycle.Drive` do exemplo anterior usando um argumento posicional e um argumento nomeado.

C#

```
int travelTime = moto.Drive(170, speed: 55);
```

Métodos herdados e substituídos

Além dos membros que são definidos explicitamente em um tipo, um tipo herda membros definidos em suas classes base. Como todos os tipos no sistema de tipos gerenciado são herdados direta ou indiretamente da classe Object, todos os tipos herdam seus membros, como Equals(Object), GetType() e ToString(). O exemplo a seguir define uma classe `Person`, instancia dois objetos `Person` e chama o método `Person.Equals` para determinar se os dois

objetos são iguais. O método `Equals`, no entanto, não é definido na classe `Person`; ele é herdado do Object.

C#

```
public class Person
{
    public string FirstName = default!;
}

public static class ClassTypeExample
{
    public static void Main()
    {
        Person p1 = new() { FirstName = "John" };
        Person p2 = new() { FirstName = "John" };
        Console.WriteLine($"p1 = p2: {p1.Equals(p2)}");
    }
}
// The example displays the following output:
//      p1 = p2: False
```

Tipos podem substituir membros herdados usando a palavra-chave `override` e fornecendo uma implementação para o método substituído. A assinatura do método precisa ser igual à do método substituído. O exemplo a seguir é semelhante ao anterior, exceto que ele substitui o método Equals(Object). (Ele também substitui o método GetHashCode(), uma vez que os dois métodos destinam-se a fornecer resultados consistentes.)

C#

```
namespace methods;

public class Person
{
    public string FirstName = default!;

    public override bool Equals(object? obj) =>
        obj is Person p2 &&
        FirstName.Equals(p2.FirstName);

    public override int GetHashCode() => FirstName.GetHashCode();
}

public static class Example
{
    public static void Main()
    {
        Person p1 = new() { FirstName = "John" };
        Person p2 = new() { FirstName = "John" };
        Console.WriteLine($"p1 = p2: {p1.Equals(p2)}");
    }
```

```
}
// The example displays the following output:
//      p1 = p2: True
```

Passando parâmetros

Tipos em C# são *tipos de valor* ou *tipos de referência*. Para obter uma lista de tipos de valor internos, consulte Tipos. Por padrão, os tipos de referência e tipos de valor são passados por valor para um método.

Passando parâmetros por valor

Quando um tipo de valor é passado para um método por valor, uma cópia do objeto, em vez do próprio objeto, é passada para o método. Portanto, as alterações no objeto do método chamado não têm efeito no objeto original quando o controle retorna ao chamador.

O exemplo a seguir passa um tipo de valor para um método por valor e o método chamado tenta alterar o valor do tipo de valor. Ele define uma variável do tipo `int`, que é um tipo de valor, inicializa o valor para 20 e o passa para um método chamado `ModifyValue` que altera o valor da variável para 30. No entanto, quando o método retorna, o valor da variável permanece inalterado.

C#

```
public static class ByValueExample
{
    public static void Main()
    {
        var value = 20;
        Console.WriteLine("In Main, value = {0}", value);
        ModifyValue(value);
        Console.WriteLine("Back in Main, value = {0}", value);
    }

    static void ModifyValue(int i)
    {
        i = 30;
        Console.WriteLine("In ModifyValue, parameter value = {0}", i);
        return;
    }
}
// The example displays the following output:
//      In Main, value = 20
//      In ModifyValue, parameter value = 30
//      Back in Main, value = 20
```

Quando um objeto do tipo de referência é passado para um método por valor, uma referência ao objeto é passada por valor. Ou seja, o método recebe não o objeto em si, mas um argumento que indica o local do objeto. Se você alterar um membro do objeto usando essa referência, a alteração será refletida no objeto quando o controle retornar para o método de chamada. No entanto, substituir o objeto passado para o método não tem efeito no objeto original quando o controle retorna para o chamador.

O exemplo a seguir define uma classe (que é um tipo de referência) chamada `SampleRefType`. Ele cria uma instância de um objeto `SampleRefType`, atribui 44 ao seu campo `value` e passa o objeto para o método `ModifyObject`. Este exemplo faz essencialmente a mesma coisa que o exemplo anterior: ele passa um argumento por valor para um método. Mas como um tipo de referência é usado, o resultado é diferente. A modificação feita em `ModifyObject` para o campo `obj.value` também muda o campo `value` do argumento, `rt`, no método `Main` para 33, como a saída do exemplo mostra.

C#

```
public class SampleRefType
{
    public int value;
}

public static class ByRefTypeExample
{
    public static void Main()
    {
        var rt = new SampleRefType { value = 44 };
        ModifyObject(rt);
        Console.WriteLine(rt.value);
    }

    static void ModifyObject(SampleRefType obj) => obj.value = 33;
}
```

Passando parâmetros por referência

Você passa um parâmetro por referência quando deseja alterar o valor de um argumento em um método e deseja refletir essa alteração quando o controle retorna para o método de chamada. Para passar um parâmetro por referência, use a palavra-chave ref ou out. Você também pode passar um valor por referência para evitar a cópia e ainda evitar modificações usando a palavra-chave in.

O exemplo a seguir é idêntico ao anterior, exceto que o valor é passado por referência para o método `ModifyValue`. Quando o valor do parâmetro é modificado no método `ModifyValue`, a alteração no valor é refletida quando o controle retorna ao chamador.

C#

```
public static class ByRefExample
{
    public static void Main()
    {
        var value = 20;
        Console.WriteLine("In Main, value = {0}", value);
        ModifyValue(ref value);
        Console.WriteLine("Back in Main, value = {0}", value);
    }

    private static void ModifyValue(ref int i)
    {
        i = 30;
        Console.WriteLine("In ModifyValue, parameter value = {0}", i);
        return;
    }
}
// The example displays the following output:
//      In Main, value = 20
//      In ModifyValue, parameter value = 30
//      Back in Main, value = 30
```

Um padrão comum que usa parâmetros pela referência envolve a troca os valores das variáveis. Você passa duas variáveis para um método por referência e o método troca seus conteúdos. O exemplo a seguir troca valores inteiros.

C#

```
public static class RefSwapExample
{
    static void Main()
    {
        int i = 2, j = 3;
        Console.WriteLine($"i = {i}  j = {j}");

        Swap(ref i, ref j);

        Console.WriteLine($"i = {i}  j = {j}");
    }

    static void Swap(ref int x, ref int y) =>
        (y, x) = (x, y);
}
// The example displays the following output:
//      i = 2  j = 3
//      i = 3  j = 2
```

Passar um parâmetro de tipo de referência permite que você altere o valor da própria referência, em vez de o valor de seus campos ou elementos individuais.

Coleções de parâmetros

Às vezes, o requisito de que você especifique o número exato de argumentos para o método é restritivo. Usando a palavra-chave `params` para indicar que um parâmetro é uma coleção de parâmetros, você permite que o método seja chamado com um número variável de argumentos. O parâmetro marcado com a palavra-chave `params` deve ser um tipo de coleção e ele deve ser o último parâmetro na lista de parâmetros do método.

Um chamador pode, então, invocar o método de uma das quatro maneiras para o parâmetro `params`:

- Passando uma coleção do tipo apropriado que contém o número de elementos desejado. O exemplo usa uma expressão de coleção para que o compilador crie um tipo de coleção apropriado.
- Passando uma lista separada por vírgulas de argumentos individuais do tipo apropriado para o método. O compilador cria o tipo de coleção apropriado.
- Passando `null`.
- Não fornecendo um argumento para a coleção de parâmetros.

O exemplo a seguir define um método chamado `GetVowels` que retorna todas as vogais de uma coleção de parâmetros. O método `Main` ilustra todas as quatro maneiras de invocar o método. Os chamadores não precisam fornecer argumentos para parâmetros que incluem o modificador `params`. Nesse caso, o parâmetro é uma coleção vazia.

C#

```
static class ParamsExample
{
    static void Main()
    {
        string fromArray = GetVowels(["apple", "banana", "pear"]);
        Console.WriteLine($"Vowels from collection expression: '{fromArray}'");

        string fromMultipleArguments = GetVowels("apple", "banana", "pear");
        Console.WriteLine($"Vowels from multiple arguments:
'{fromMultipleArguments}'");

        string fromNull = GetVowels(null);
        Console.WriteLine($"Vowels from null: '{fromNull}'");

        string fromNoValue = GetVowels();
        Console.WriteLine($"Vowels from no value: '{fromNoValue}'");
    }
```

```
    static string GetVowels(params IEnumerable<string>? input)
    {
        if (input == null || !input.Any())
        {
            return string.Empty;
        }

        char[] vowels = ['A', 'E', 'I', 'O', 'U'];
        return string.Concat(
            input.SelectMany(
                word => word.Where(letter =>
vowels.Contains(char.ToUpper(letter)))));
    }
}

// The example displays the following output:
//     Vowels from array: 'aeaaaea'
//     Vowels from multiple arguments: 'aeaaaea'
//     Vowels from null: ''
//     Vowels from no value: ''
```

Antes do C# 13, o modificador `params` só pode ser usado com uma única matriz dimensional.

Parâmetros e argumentos opcionais

Uma definição de método pode especificar que os parâmetros são obrigatórios ou que são opcionais. Por padrão, os parâmetros são obrigatórios. Os parâmetros opcionais são especificados incluindo o valor padrão do parâmetro na definição do método. Quando o método for chamado, se nenhum argumento for fornecido para um parâmetro opcional, o valor padrão será usado em vez disso.

Você atribuirá o valor padrão do parâmetro com um dos tipos de expressões a seguir:

- Uma constante, como um número ou uma cadeia de caracteres literal.
- Uma expressão do formulário `default(SomeType)`, em que `SomeType` pode ser um tipo de valor ou um tipo de referência. Se for um tipo de referência, ele será efetivamente o mesmo que especificar `null`. Você pode usar o literal `default`, pois o compilador pode inferir o tipo a partir da declaração do parâmetro.
- Uma expressão da forma `new ValType()`, em que `ValType` é um tipo de valor. Essa expressão invoca o construtor sem parâmetros implícito do tipo de valor, que não é de fato um membro do tipo.

> ⓘ **Observação**

> Quando uma expressão do formulário `new ValType()` invoca o construtor sem parâmetros definido explicitamente de um tipo de valor, o compilador gera um erro, pois o valor do parâmetro padrão deve ser uma constante de tempo de compilação. Use a expressão `default(ValType)` ou o literal `default` para fornecer o valor padrão do parâmetro. Para obter mais informações sobre construtores sem parâmetros, consulte a seção [inicialização de Struct e valores padrão] do artigo [Tipos de estrutura] .

Se um método inclui parâmetros obrigatórios e opcionais, os parâmetros opcionais são definidos no final da lista de parâmetros, após todos os parâmetros obrigatórios.

O exemplo a seguir define um método, `ExampleMethod`, que tem um parâmetro obrigatório e dois opcionais.

C#

```
public class Options
{
    public void ExampleMethod(int required, int optionalInt = default,
                              string? description = default)
    {
        var msg = $"{description ?? "N/A"}: {required} + {optionalInt} = {required
+ optionalInt}";
        Console.WriteLine(msg);
    }
}
```

O chamador deve fornecer um argumento para todos os parâmetros opcionais até o último parâmetro opcional para o qual um argumento é fornecido. No método `ExampleMethod`, por exemplo, se o chamador fornecer um argumento para o parâmetro `description`, ele deverá fornecer também um para o parâmetro `optionalInt`. `opt.ExampleMethod(2, 2, "Addition of 2 and 2");` é uma chamada de método válida, `opt.ExampleMethod(2, , "Addition of 2 and 0");` gera um erro do compilador de “Argumento ausente”.

Se um método for chamado usando argumentos nomeados ou uma combinação de argumentos posicionais e nomeados, o chamador poderá omitir todos os argumentos após o último argumento posicional na chamada do método.

A exemplo a seguir chama o método `ExampleMethod` três vezes. As duas primeiras chamadas de método usam argumentos posicionais. O primeiro omite ambos os argumentos opcionais, enquanto o segundo omite o último argumento. A terceira chamada de método fornece um argumento posicional para o parâmetro obrigatório, mas usa um argumento nomeado para fornecer um valor para o parâmetro `description` enquanto omite o argumento `optionalInt`.

C#

```
public static class OptionsExample
{
    public static void Main()
    {
        var opt = new Options();
        opt.ExampleMethod(10);
        opt.ExampleMethod(10, 2);
        opt.ExampleMethod(12, description: "Addition with zero:");
    }
}
// The example displays the following output:
//      N/A: 10 + 0 = 10
//      N/A: 10 + 2 = 12
//      Addition with zero:: 12 + 0 = 12
```

O uso de parâmetros opcionais afeta a *resolução de sobrecarga* ou a maneira como o compilador C# determina qual sobrecarga invocar para uma chamada de método, da seguinte maneira:

- Um membro é um candidato para execução se cada um de seus parâmetros corresponder por nome ou por posição a um único argumento. Além disso, esse argumento pode ser convertido no tipo do parâmetro.
- Se mais de um candidato for encontrado, as regras de resolução de sobrecarga de conversões preferenciais serão aplicadas aos argumentos que são especificados explicitamente. Os argumentos omitidos para parâmetros opcionais são ignorados.
- Se dois candidatos são considerados igualmente bons, a preferência vai para um candidato que não tenha parâmetros opcionais para os quais argumentos foram omitidos na chamada.

Valores retornados

Os métodos podem retornar um valor para o chamador. Se o tipo de retorno (o tipo listado antes do nome do método) não for `void`, o método poderá retornar o valor usando a palavra-chave `return`. Uma instrução com a palavra-chave `return` seguida por uma variável, constante ou expressão que corresponde ao tipo de retorno retorna esse valor para o chamador do método. Métodos com um tipo de retorno não nulo devem usar a palavra-chave `return` para retornar um valor. A palavra-chave `return` também interrompe a execução do método.

Se o tipo de retorno for `void`, uma instrução `return` sem um valor ainda será útil para interromper a execução do método. Sem a palavra-chave `return`, a execução do método é interrompida quando chega ao final do bloco de código.

Por exemplo, esses dois métodos usam a palavra-chave `return` para retornar inteiros:

```C#
class SimpleMath
{
    public int AddTwoNumbers(int number1, int number2) =>
        number1 + number2;

    public int SquareANumber(int number) =>
        number * number;
}
```

Os exemplos anteriores são membros aptos para expressão. Os membros aptos para expressão retornam o valor retornado pela expressão.

Você também pode optar por definir seus métodos com um corpo de instrução e uma instrução `return`:

```C#
class SimpleMathExtension
{
    public int DivideTwoNumbers(int number1, int number2)
    {
        return number1 / number2;
    }
}
```

Para usar um valor retornado de um método, você pode atribuir o valor retornado a uma variável:

```C#
int result = obj.DivideTwoNumbers(6,2);
// The result is 3.
Console.WriteLine(result);
```

O método de chamada também pode usar a chamada do método onde um valor do mesmo tipo seria suficiente. Por exemplo, os dois exemplos de código a seguir atingem a mesma meta:

```C#
int result = obj.AddTwoNumbers(1, 2);
result = obj.SquareANumber(result);
// The result is 9.
Console.WriteLine(result);
```

```C#
result = obj.SquareANumber(obj.AddTwoNumbers(1, 2));
// The result is 9.
Console.WriteLine(result);
```

Às vezes, você deseja que seu método retorne mais de um único valor. Você usa *tipos de tupla* e *literais de tupla* para retornar vários valores. O tipo de tupla define os tipos de dados dos elementos da tupla. Os literais de tupla fornecem os valores reais da tupla retornada. No exemplo a seguir, `(string, string, string, int)` define o tipo de tupla retornado pelo método `GetPersonalInfo`. A expressão `(per.FirstName, per.MiddleName, per.LastName, per.Age)` é a tupla literal, o método retorna o nome, o nome do meio e o sobrenome, juntamente com a idade, de um objeto `PersonInfo`.

```C#
public (string, string, string, int) GetPersonalInfo(string id)
{
    PersonInfo per = PersonInfo.RetrieveInfoById(id);
    return (per.FirstName, per.MiddleName, per.LastName, per.Age);
}
```

O chamador pode consumir a tupla retornada usando o seguinte código:

```C#
var person = GetPersonalInfo("111111111");
Console.WriteLine($"{person.Item1} {person.Item3}: age = {person.Item4}");
```

Os nomes também podem ser atribuídos aos elementos da tupla na definição de tipo de tupla. O exemplo a seguir mostra uma versão alternativa do método `GetPersonalInfo` que usa elementos nomeados:

```C#
public (string FName, string MName, string LName, int Age) GetPersonalInfo(string
id)
{
    PersonInfo per = PersonInfo.RetrieveInfoById(id);
    return (per.FirstName, per.MiddleName, per.LastName, per.Age);
}
```

A chamada anterior para o método `GetPersonalInfo` pode ser modificada da seguinte maneira:

```C#
```

```
var person = GetPersonalInfo("111111111");
Console.WriteLine($"{person.FName} {person.LName}: age = {person.Age}");
```

Se um método usar uma matriz como parâmetro e modificar o valor de elementos individuais, não será necessário que o método retorne a matriz. O C# passa todos os tipos de referência por valor e o valor de uma referência de matriz é o ponteiro para a matriz. No exemplo a seguir, as alterações no conteúdo da matriz `values` realizados pelo método `DoubleValues` são observáveis por qualquer código que faz referência à matriz.

C#

```
public static class ArrayValueExample
{
    static void Main()
    {
        int[] values = [2, 4, 6, 8];
        DoubleValues(values);
        foreach (var value in values)
        {
            Console.Write("{0}  ", value);
        }
    }

    public static void DoubleValues(int[] arr)
    {
        for (var ctr = 0; ctr <= arr.GetUpperBound(0); ctr++)
        {
            arr[ctr] *= 2;
        }
    }
}
// The example displays the following output:
//       4  8  12  16
```

Métodos de extensão

Normalmente, há duas maneiras de adicionar um método a um tipo existente:

- Modificar o código-fonte para esse tipo. Modificar a fonte criar uma alteração interruptiva, se você também adicionar campos de dados privados para dar suporte ao método.
- Definir o novo método em uma classe derivada. Não é possível adicionar um método dessa forma usando a herança para outros tipos, como estruturas e enumerações. Isso também não pode ser usado para “adicionar” um método a uma classe selada.

Os membros da extensão permitem que você "adicione" membros a um tipo existente sem modificar o tipo em si ou implementar o novo método em um tipo herdado. O membro apto para extensão também não precisa residir no mesmo assembly que o tipo que ele estende. Você chama um método de extensão como se fosse um membro definido de um tipo.

Para obter mais informações, consulte Membros aptos para extensão.

Métodos assíncronos

Usando o recurso async, você pode invocar métodos assíncronos sem usar retornos de chamada explícitos ou dividir manualmente seu código entre vários métodos ou expressões lambda.

Se você marcar um método com o modificador assíncrono , poderá usar o operador await no método. Quando o controle atingir uma expressão `await` no método assíncrono, o controle retornará para o chamador se a tarefa aguardada não estiver concluída, e o progresso no método com a palavra-chave `await` será suspenso até a tarefa aguardada ser concluída. Quando a tarefa for concluída, a execução poderá ser retomada no método.

> **Observação**
>
> Um método assíncrono retorna para o chamador quando encontra o primeiro objeto esperado que ainda não está completo ou chega ao final do método assíncrono, o que ocorrer primeiro.

Um método assíncrono normalmente tem um tipo de retorno Task<TResult>, Task, IAsyncEnumerable<T> ou `void`. O tipo de retorno `void` é usado principalmente para definir manipuladores de eventos, nos quais o tipo de retorno `void` é necessário. Um método assíncrono que retorna `void` não pode ser aguardado e o chamador de um método de retorno nulo não pode capturar as exceções que esse método gera. Um método assíncrono pode ter qualquer tipo de retorno semelhante a uma tarefa.

No exemplo a seguir, `DelayAsync` é um método assíncrono que contém uma instrução return que retorna um inteiro. Como é um método assíncrono, sua declaração de método deve ter um tipo de retorno `Task<int>`. Como o tipo de retorno é `Task<int>`, a avaliação da expressão `await` em `DoSomethingAsync` produz um inteiro, como a instrução `int result = await delayTask` a seguir demonstra.

```
C#
```

```
class Program
{
    static Task Main() => DoSomethingAsync();

    static async Task DoSomethingAsync()
    {
        Task<int> delayTask = DelayAsync();
        int result = await delayTask;

        // The previous two statements may be combined into
        // the following statement.
        //int result = await DelayAsync();

        Console.WriteLine($"Result: {result}");
    }

    static async Task<int> DelayAsync()
    {
        await Task.Delay(100);
        return 5;
    }
}
// Example output:
//   Result: 5
```

Um método assíncrono não pode declarar parâmetros in, ref ou out , mas pode chamar métodos que têm esses parâmetros.

Para obter mais informações sobre métodos assíncronos, consulte Programação assíncrona com async e await e Tipos de retorno assíncronos.

Membros aptos para expressão

É comum ter definições de método que retornam imediatamente com o resultado de uma expressão ou que têm uma única instrução como o corpo do método. Há um atalho de sintaxe para definir esses métodos usando `=>`:

C#

```
public Point Move(int dx, int dy) => new Point(x + dx, y + dy);
public void Print() => Console.WriteLine(First + " " + Last);
// Works with operators, properties, and indexers too.
public static Complex operator +(Complex a, Complex b) => a.Add(b);
public string Name => First + " " + Last;
public Customer this[long id] => store.LookupCustomer(id);
```

Se o método retornar `void` ou for um método assíncrono, o corpo do método deverá ser uma expressão de instrução (igual aos lambdas). Para propriedades e indexadores, eles devem ser somente leitura e você não usa a palavra-chave do acessador `get`.

Iteradores

Um iterador realiza uma iteração personalizada em uma coleção, como uma lista ou uma matriz. Um iterador usa a instrução de retorno de rendimento para retornar cada elemento um de cada vez. Quando uma instrução `yield return` for atingida, o local atual será lembrado para que o chamador possa solicitar o próximo elemento na sequência.

O tipo de retorno de um iterador pode ser IEnumerable, IEnumerable<T>, IAsyncEnumerable<T>, IEnumerator ou IEnumerator<T>.

Para obter mais informações, consulte Iteradores.

Confira também

- Modificadores de acesso
- Classes estáticas e membros de classe estática
- Herança
- Classes abstratas e seladas e membros da classe
- params
-
- Ref
- in
- Passando parâmetros

Iteradores

19/06/2025

Quase todos os programas que você escrever terão alguma necessidade de iterar em uma coleção. Você escreverá um código que examina todos os itens de uma coleção.

Você também criará métodos de iterador, que são métodos que produzem um *iterador* para os elementos dessa classe. Um *iterador* é um objeto que atravessa um contêiner, particularmente listas. Os iteradores podem ser usados para:

- Executando uma ação sobre cada item em uma coleção.
- Enumerar uma coleção personalizada.
- Estender LINQ ou outras bibliotecas.
- Criando um pipeline de dados onde os dados fluem com eficiência através de métodos iterativos.

A linguagem C# fornece recursos para gerar e consumir sequências. Essas sequências podem ser produzidas e consumidas de forma síncrona ou assíncrona. Este artigo fornece uma visão geral desses recursos.

Iterando usando foreach

Enumerar uma coleção é simples: a `foreach` palavra-chave enumera uma coleção, executando a instrução inserida uma vez para cada elemento na coleção:

C#

```
foreach (var item in collection)
{
    Console.WriteLine(item?.ToString());
}
```

Isso é tudo. Para iterar em todo o conteúdo de uma coleção, a instrução `foreach` é tudo o que você precisa. A `foreach` declaração não é mágica, no entanto. Ele depende de duas interfaces genéricas definidas na biblioteca do .NET Core para gerar o código necessário para iterar uma coleção: `IEnumerable<T>` e `IEnumerator<T>`. Esse mecanismo é explicado com mais detalhes abaixo.

Ambas as interfaces também têm equivalentes não genéricos: `IEnumerable` e `IEnumerator`. As versões genéricas são preferenciais para código moderno.

Quando uma sequência é gerada de forma assíncrona, você pode usar a `await foreach` instrução para consumir a sequência de forma assíncrona:

```
C#

await foreach (var item in asyncSequence)
{
Console.WriteLine(item?.ToString());
}
```

Quando uma sequência é uma System.Collections.Generic.IEnumerable<T>, você usa `foreach`. Quando uma sequência é uma System.Collections.Generic.IAsyncEnumerable<T>, você usa `await foreach`. No último caso, a sequência é gerada de forma assíncrona.

Fontes de enumeração com métodos de iteração

Outro ótimo recurso da linguagem C# permite criar métodos que criam uma origem para uma enumeração. Esses métodos são chamados *de métodos de iterador*. Um método iterador define como gerar os objetos em uma sequência quando solicitado. Você usa as `yield return` palavras-chave contextuais para definir um método iterador.

Você pode escrever este método para produzir a sequência de inteiros de 0 a 9:

```
C#

public IEnumerable<int> GetSingleDigitNumbers()
{
    yield return 0;
    yield return 1;
    yield return 2;
    yield return 3;
    yield return 4;
    yield return 5;
    yield return 6;
    yield return 7;
    yield return 8;
    yield return 9;
}
```

O código acima mostra instruções distintas `yield return` para realçar o fato de que você pode usar várias instruções discretas `yield return` em um método iterador. Você pode (e muitas vezes usar) outros constructos de linguagem para simplificar o código de um método iterador. A definição do método abaixo produz exatamente a mesma sequência de números:

```
C#
```

```
public IEnumerable<int> GetSingleDigitNumbersLoop()
{
    int index = 0;
    while (index < 10)
        yield return index++;
}
```

Você não tem que decidir um ou outro. Você pode ter quantas instruções `yield return` forem necessárias para atender as necessidades do seu método:

```C#
public IEnumerable<int> GetSetsOfNumbers()
{
    int index = 0;
    while (index < 10)
        yield return index++;

    yield return 50;

    index = 100;
    while (index < 110)
        yield return index++;
}
```

Todos esses exemplos anteriores teriam um equivalente assíncrono. Em cada caso, você substituiria o tipo de retorno de `IEnumerable<T>` por um `IAsyncEnumerable<T>`. Por exemplo, o exemplo anterior teria a seguinte versão assíncrona:

```C#
public async IAsyncEnumerable<int> GetSetsOfNumbersAsync()
{
    int index = 0;
    while (index < 10)
        yield return index++;

    await Task.Delay(500);

    yield return 50;

    await Task.Delay(500);

    index = 100;
    while (index < 110)
        yield return index++;
}
```

Essa é a sintaxe para iteradores síncronos e assíncronos. Vamos considerar um exemplo do mundo real. Imagine que você está em um projeto de IoT e os sensores de dispositivo geram um fluxo muito grande de dados. Para obter uma ideia dos dados, você pode escrever um método que amostra cada elemento de dados Nth. Esse pequeno método iterador resolve:

C#

```
public static IEnumerable<T> Sample<T>(this IEnumerable<T> sourceSequence, int
interval)
{
    int index = 0;
    foreach (T item in sourceSequence)
    {
        if (index++ % interval == 0)
            yield return item;
    }
}
```

Se a leitura do dispositivo IoT produzir uma sequência assíncrona, você modificará o método como mostra o seguinte método:

C#

```
public static async IAsyncEnumerable<T> Sample<T>(this IAsyncEnumerable<T>
sourceSequence, int interval)
{
    int index = 0;
    await foreach (T item in sourceSequence)
    {
        if (index++ % interval == 0)
            yield return item;
    }
}
```

Há uma restrição importante em métodos de iterador: você não pode ter uma instrução `return` e uma instrução `yield return` no mesmo método. O código a seguir não será compilado:

C#

```
public IEnumerable<int> GetSingleDigitNumbers()
{
    int index = 0;
    while (index < 10)
        yield return index++;

    yield return 50;

    // generates a compile time error:
```

```
    var items = new int[] {100, 101, 102, 103, 104, 105, 106, 107, 108, 109 };
    return items;
}
```

Essa restrição normalmente não é um problema. Você tem a opção de usar `yield return` todo o método ou separar o método original em vários métodos, alguns usando `return` e outros usando `yield return`.

Você pode modificar o último método ligeiramente para usar `yield return` em todos os lugares:

C#

```
public IEnumerable<int> GetFirstDecile()
{
    int index = 0;
    while (index < 10)
        yield return index++;

    yield return 50;

    var items = new int[] {100, 101, 102, 103, 104, 105, 106, 107, 108, 109 };
    foreach (var item in items)
        yield return item;
}
```

Às vezes, a resposta certa é dividir um método iterador em dois métodos diferentes. Um que usa `return`, e um segundo que usa `yield return`. Considere uma situação em que talvez você queira retornar uma coleção vazia ou os cinco primeiros números ímpares, com base em um argumento booliano. Você pode escrever isso como estes dois métodos:

C#

```
public IEnumerable<int> GetSingleDigitOddNumbers(bool getCollection)
{
    if (getCollection == false)
        return new int[0];
    else
        return IteratorMethod();
}

private IEnumerable<int> IteratorMethod()
{
    int index = 0;
    while (index < 10)
    {
        if (index % 2 == 1)
            yield return index;
        index++;
```

```
    }
}
```

Examine os métodos acima. O primeiro usa a instrução padrão `return` para retornar uma coleção vazia ou o iterador criado pelo segundo método. O segundo método usa a `yield return` instrução para criar a sequência solicitada.

Aprofundamento em `foreach`

A instrução `foreach` se expande em uma expressão padrão que usa as interfaces `IEnumerable<T>` e `IEnumerator<T>` para iterar em todos os elementos de uma coleção. Ele também minimiza os erros que os desenvolvedores cometem ao não gerenciar corretamente os recursos.

O compilador converte o `foreach` loop mostrado no primeiro exemplo em algo semelhante a este constructo:

C#

```
IEnumerator<int> enumerator = collection.GetEnumerator();
while (enumerator.MoveNext())
{
    var item = enumerator.Current;
    Console.WriteLine(item.ToString());
}
```

O código exato gerado pelo compilador é mais complicado e lida com situações em que o objeto retornado `GetEnumerator()` implementa a `IDisposable` interface. A expansão completa gera um código mais parecido com este:

C#

```
{
    var enumerator = collection.GetEnumerator();
    try
    {
        while (enumerator.MoveNext())
        {
            var item = enumerator.Current;
            Console.WriteLine(item.ToString());
        }
    }
    finally
    {
        // dispose of enumerator.
```

```
    }
}
```

O compilador converte o primeiro exemplo assíncrono em algo semelhante a este constructo:

```C#
{
    var enumerator = collection.GetAsyncEnumerator();
    try
    {
        while (await enumerator.MoveNextAsync())
        {
            var item = enumerator.Current;
            Console.WriteLine(item.ToString());
        }
    }
    finally
    {
        // dispose of async enumerator.
    }
}
```

A maneira pela qual o enumerador é descartado depende das características do tipo de `enumerator`. No caso síncrono geral, a `finally` cláusula se expande para:

```C#
finally
{
   (enumerator as IDisposable)?.Dispose();
}
```

O caso assíncrono geral se expande para:

```C#
finally
{
    if (enumerator is IAsyncDisposable asyncDisposable)
        await asyncDisposable.DisposeAsync();
}
```

No entanto, se o tipo de `enumerator` for um tipo lacrado e não houver nenhuma conversão implícita do tipo de `enumerator` para `IDisposable` ou `IAsyncDisposable`, a cláusula `finally` se expande para um bloco vazio.

```C#
finally
{
}
```

Se houver uma conversão implícita do tipo `enumerator` para `IDisposable`, e `enumerator` for um tipo de valor não anulável, a cláusula `finally` se expande para:

```C#
finally
{
   ((IDisposable)enumerator).Dispose();
}
```

Felizmente, você não precisa se lembrar de todos esses detalhes. A instrução `foreach` trata todas essas nuances para você. O compilador gerará o código correto para qualquer uma dessas construções.

Introdução a delegados e eventos em C#

19/06/2025

Os delegados fornecem um mecanismo de *associação tardia* no .NET. Ligação tardia (late binding) significa que você cria um algoritmo em que o chamador também fornece pelo menos um método que implementa parte do algoritmo.

Por exemplo, considere classificar uma lista de estrelas em um aplicativo de astronomia. Você pode optar por classificar essas estrelas por sua distância da Terra, ou pela magnitude da estrela, ou seu brilho percebido.

Em todos esses casos, o método Sort() faz essencialmente a mesma coisa: organiza os itens na lista com base em alguma comparação. O código que compara duas estrelas é diferente para cada uma das ordenações de classificação.

Esses tipos de soluções são usados em software há meio século. O conceito de delegado de linguagem C# fornece suporte de linguagem de primeira classe e segurança de tipos em torno do conceito.

Como você verá mais adiante nesta série, o código C# que você escreve para algoritmos como esse é tipo seguro. O compilador garante que os tipos correspondam a argumentos e tipos de retorno.

Os ponteiros de função dão suporte a cenários semelhantes, em que você precisa de mais controle sobre a convenção de chamada. O código associado a um delegado é invocado usando um método virtual adicionado a um tipo de delegado. Usando ponteiros de função, você pode especificar convenções diferentes.

Metas de design de idioma para delegados

Os designers de linguagem enumeraram várias metas para o recurso que eventualmente se tornaram delegados.

A equipe queria um constructo de linguagem comum que pudesse ser usado para qualquer algoritmo de associação tardia. Os delegados permitem que os desenvolvedores aprendam um conceito e usem esse mesmo conceito em vários problemas de software diferentes.

Em segundo lugar, a equipe queria dar suporte a chamadas de método single ou multicast. (Representantes multicast são delegados que encadeiam várias chamadas de método. Você verá exemplos mais adiante nesta série.)

A equipe queria que os delegados oferecessem suporte à mesma segurança de tipo que os desenvolvedores esperam de todos os constructos C#.

Por fim, a equipe reconheceu que um padrão de evento é um padrão específico em que delegados ou qualquer algoritmo de associação tardia são úteis. A equipe queria garantir que o código para delegados pudesse fornecer a base para o padrão de evento .NET.

O resultado de todo esse trabalho era o suporte do delegado e do evento no C# e .NET.

Os artigos restantes nesta série abordarão recursos de linguagem, suporte à biblioteca e idiomas comuns usados quando você trabalha com delegados e eventos. Você saberá mais sobre:

- A `delegate` palavra-chave e qual código ele gera.
- Os recursos na `System.Delegate` classe e como esses recursos são usados.
- Como criar delegados fortemente tipados.
- Como criar métodos que podem ser invocados por meio de delegados.
- Como trabalhar com delegados e eventos usando expressões lambda.
- Como os delegados se tornam um dos blocos de construção para LINQ.
- Como os delegados são a base para o padrão de evento .NET e como eles são diferentes.

Vamos começar.

Próximo

System.Delegate e a `delegate` palavra-chave

Artigo • 22/05/2025

Anterior

Este artigo aborda as classes no .NET que dão suporte a delegados e como elas são mapeadas para a `delegate` palavra-chave.

Definir tipos de delegado

Vamos começar com a palavra-chave 'delegado', porque isso é principalmente o que você usará enquanto trabalha com delegados. O código que o compilador gera quando você usa a palavra-chave `delegate` será mapeado em chamadas de método que invocam membros das classes Delegate e MulticastDelegate.

Você define um tipo de delegado usando uma sintaxe semelhante à definição de uma assinatura de método. Basta adicionar a `delegate` palavra-chave à definição.

Vamos continuar a usar o método List.Sort() como nosso exemplo. A primeira etapa é criar um tipo para o delegado de comparação:

C#

```
// From the .NET Core library

// Define the delegate type:
public delegate int Comparison<in T>(T left, T right);
```

O compilador gera uma classe, derivada de `System.Delegate`, que coincide com a assinatura usada (nesse caso, um método que retorna um inteiro e tem dois argumentos). O tipo desse delegado é `Comparison`. O `Comparison` tipo delegado é um tipo genérico. Para obter mais informações, consulte classes e métodos genéricos.

Observe que a sintaxe pode aparecer como se estivesse declarando uma variável, mas na verdade está declarando um *tipo*. Você pode definir tipos de delegado dentro de classes, diretamente dentro de namespaces ou até mesmo no namespace global.

> ⓘ **Observação**

Não é recomendável declarar tipos de delegado (ou outros tipos) diretamente no namespace global.

O compilador também gera manipuladores de adição e remoção para esse novo tipo para que os clientes dessa classe possam adicionar e remover métodos da lista de invocação de uma instância. O compilador imporá que a assinatura do método que está sendo adicionado ou removido corresponda à assinatura usada ao declarar o método.

Declarar instâncias de delegados

Depois de definir o delegado, você pode criar uma instância desse tipo. Como todas as variáveis em C#, você não pode declarar instâncias delegadas diretamente em um namespace ou no namespace global.

C#

```
// inside a class definition:

// Declare an instance of that type:
public Comparison<T> comparator;
```

O tipo da variável é `Comparison<T>`, o tipo delegado definido anteriormente. O nome da variável é `comparator`.

Esse snippet de código acima declarou uma variável de membro dentro de uma classe. Você também pode declarar variáveis delegada que são variáveis locais ou argumentos para métodos.

Invocar delegados

Você invoca os métodos que estão na lista de invocação de um delegado chamando esse delegado. Dentro do `Sort()` método, o código chamará o método de comparação para determinar qual ordem colocar objetos:

C#

```
int result = comparator(left, right);
```

Na linha acima, o código *invoca* o método anexado ao delegado. Você trata a variável como um nome de método e a invoca usando a sintaxe de chamada de método normal.

Essa linha de código faz uma suposição arriscada: não há garantia de que um alvo tenha sido adicionado ao delegado. Se nenhum destino tiver sido anexado, a linha acima fará com que um `NullReferenceException` seja lançado. Os idiomas usados para resolver esse problema são mais complicados do que uma simples verificação nula e são abordados posteriormente nesta série.

Atribuir, adicionar e remover destinos de invocação

É assim que um tipo de delegado é definido e como as instâncias delegadas são declaradas e invocadas.

Os desenvolvedores que desejam usar o `List.Sort()` método precisam definir um método cuja assinatura corresponde à definição de tipo delegado e atribuí-lo ao delegado usado pelo método de classificação. Essa atribuição adiciona o método à lista de invocação desse objeto delegado.

Suponha que você queria classificar uma lista de cadeias de caracteres por seu comprimento. Sua função de comparação pode ser a seguinte:

C#

```
private static int CompareLength(string left, string right) =>
    left.Length.CompareTo(right.Length);
```

O método é declarado como um método privado. Tudo bem. Talvez você não queira que esse método faça parte de sua interface pública. Ele ainda pode ser usado como o método de comparação quando anexado a um delegado. O código de chamada terá esse método anexado à lista de destino do objeto delegado e poderá acessá-lo por meio desse delegado.

Você cria essa relação passando esse método para o `List.Sort()` método:

C#

```
phrases.Sort(CompareLength);
```

Observe que o nome do método é usado, sem parênteses. Utilizar o método como argumento instrui o compilador a converter a referência do método em uma referência que possa ser usada como alvo de delegação de invocação e anexar esse método como um alvo de invocação.

Você também poderia ter sido explícito declarando uma variável de tipo `Comparison<string>` e fazendo uma atribuição:

```C#
Comparison<string> comparer = CompareLength;
phrases.Sort(comparer);
```

Em usos em que o método usado como um destino delegado é um método pequeno, é comum usar a sintaxe de expressão lambda para executar a atribuição:

```C#
Comparison<string> comparer = (left, right) =>
left.Length.CompareTo(right.Length);
phrases.Sort(comparer);
```

O uso de expressões lambda para destinos delegados é abordado mais em uma seção posterior.

O exemplo Sort() normalmente associa um único método de destino ao delegado. No entanto, os objetos delegados dão suporte a listas de invocação que têm vários métodos de destino anexados a um objeto delegado.

Classes Delegate e MulticastDelegate

O suporte ao idioma descrito acima fornece os recursos e o suporte que você normalmente precisará para trabalhar com delegados. Esses recursos são criados em duas classes na estrutura do .NET Core: Delegate e MulticastDelegate.

A `System.Delegate` classe e sua única subclasse `System.MulticastDelegate` direta fornecem suporte à estrutura para criar delegados, registrar métodos como destinos delegados e invocar todos os métodos registrados como um destino delegado.

Curiosamente, as classes `System.Delegate` e `System.MulticastDelegate` não são tipos de delegados em si mesmas. Eles fornecem a base para todos os tipos de delegados específicos. Esse mesmo processo de design de linguagem determina que você não pode declarar uma classe que deriva de `Delegate` ou `MulticastDelegate`. As regras de linguagem C# o proíbem.

Em vez disso, o compilador C# cria instâncias de uma classe derivada de `MulticastDelegate` quando você usa a palavra-chave da linguagem C# para declarar tipos delegados.

Esse design tem suas raízes na primeira versão do C# e do .NET. Um dos objetivos da equipe de design era garantir que a linguagem assegurasse a segurança de tipos ao usar delegates. Isso significava garantir que os delegados fossem invocados com o tipo e o número corretos

de argumentos. E que qualquer tipo de retorno foi indicado corretamente no momento da compilação. Os delegados faziam parte da versão 1.0 do .NET, que era antes dos genéricos.

A melhor maneira de impor essa segurança de tipo foi o compilador criar as classes delegadas concretas que representavam a assinatura do método que está sendo usada.

Embora não seja possível criar classes derivadas diretamente, você usará os métodos definidos nessas classes. Vamos examinar os métodos mais comuns que você usará quando trabalhar com delegados.

O fato mais importante a lembrar é que cada delegado com o qual você trabalha é derivado de `MulticastDelegate`. Um delegado de multicast significa que mais de um destino de método pode ser invocado ao utilizar um delegado. O design original considerou fazer uma distinção entre delegados nos quais apenas um método-alvo podia ser anexado e invocado e delegados nos quais vários métodos-alvo podiam ser anexados e invocados. Essa distinção provou ser menos útil na prática do que se pensava originalmente. As duas classes diferentes já foram criadas e estão na estrutura desde sua versão pública inicial.

Os métodos que você usará mais com delegados são `Invoke()` e `BeginInvoke()` / `EndInvoke()`. `Invoke()` invocará todos os métodos que foram anexados a uma instância delegada específica. Como você viu acima, você normalmente invoca delegados usando a sintaxe de chamada de método na variável delegada. Como você verá mais adiante nesta série, há padrões que funcionam diretamente com esses métodos.

Agora que você viu a sintaxe da linguagem e as classes que dão suporte aos delegados, vamos examinar como os delegados fortemente tipados são usados, criados e invocados.

Próximo

Delegados Fortemente Tipados

19/06/2025

Anterior

No artigo anterior, você viu que cria tipos de delegado específicos usando a `delegate` palavra-chave.

A classe de Delegado abstrata fornece a infraestrutura para a invocação e acoplamento fraco. Os tipos delegados concretos tornam-se muito mais úteis adotando e impondo segurança de tipo para os métodos que são adicionados à lista de invocação de um objeto delegado. Quando você usa a `delegate` palavra-chave e define um tipo de delegado concreto, o compilador gera esses métodos.

Na prática, isso levaria à criação de novos tipos de delegado sempre que você precisar de uma assinatura de método diferente. Este trabalho pode ficar entediante depois de um tempo. Cada novo recurso requer novos tipos de delegado.

Felizmente, isso não é necessário. A estrutura do .NET Core contém vários tipos que você pode reutilizar sempre que precisar de tipos delegados. Essas são definições genéricas para que você possa declarar personalizações quando precisar de novas declarações de método.

O primeiro desses tipos é o Action tipo e várias variações:

C#

```
public delegate void Action();
public delegate void Action<in T>(T arg);
public delegate void Action<in T1, in T2>(T1 arg1, T2 arg2);
// Other variations removed for brevity.
```

O `in` modificador no argumento de tipo genérico é abordado no artigo sobre covariância.

Há variações do `Action` delegado que contêm até 16 argumentos, como Action<T1,T2,T3,T4,T5,T6,T7,T8,T9,T10,T11,T12,T13,T14,T15,T16>. É importante que essas definições usem argumentos genéricos diferentes para cada um dos argumentos delegados: isso oferece flexibilidade máxima. Os argumentos do método não precisam ser, mas podem ser do mesmo tipo.

Use um dos `Action` tipos para qualquer tipo delegado que tenha um tipo de retorno nulo.

A estrutura também inclui vários tipos de delegado genéricos que você pode usar para tipos delegados que retornam valores:

```C#
public delegate TResult Func<out TResult>();
public delegate TResult Func<in T1, out TResult>(T1 arg);
public delegate TResult Func<in T1, in T2, out TResult>(T1 arg1, T2 arg2);
// Other variations removed for brevity
```

O `out` modificador no argumento de tipo genérico de resultado é abordado no artigo sobre covariância.

Há variações do `Func` delegado com até 16 argumentos de entrada, como Func<T1,T2,T3,T4,T5,T6,T7,T8,T9,T10,T11,T12,T13,T14,T15,T16,TResult>. O tipo do resultado é sempre o último parâmetro de tipo em todas as `Func` declarações, por convenção.

Use um dos `Func` tipos para qualquer tipo delegado que retorne um valor.

Há também um tipo Predicate<T> especializado para um delegado que retorna um teste em um único valor:

```C#
public delegate bool Predicate<in T>(T obj);
```

Você pode observar que, para qualquer `Predicate` tipo, existe um tipo estruturalmente equivalente `Func` , por exemplo:

```C#
Func<string, bool> TestForString;
Predicate<string> AnotherTestForString;
```

Você pode pensar que esses dois tipos são equivalentes. Eles não são. Essas duas variáveis não podem ser usadas de forma intercambiável. Uma variável de um tipo não pode ser atribuída ao outro tipo. O sistema de tipos C# usa os nomes dos tipos definidos, não a estrutura.

Todas essas definições de tipo delegado na Biblioteca do .NET Core devem significar que você não precisa definir um novo tipo de delegado para qualquer novo recurso criado que exija delegados. Essas definições genéricas devem fornecer todos os tipos delegados necessários na maioria das situações. Você pode simplesmente instanciar um desses tipos com os parâmetros de tipo necessários. No caso de algoritmos que podem se tornar genéricos, esses delegados podem ser usados como tipos genéricos.

Isso deve economizar tempo e minimizar o número de novos tipos que você precisa criar para trabalhar com delegados.

No próximo artigo, você verá vários padrões comuns para trabalhar com delegados na prática.

Próximo

Padrões comuns para delegados

19/06/2025

Anterior

Os delegados fornecem um mecanismo que permite designs de software que envolvem acoplamento mínimo entre componentes.

Um excelente exemplo para esse tipo de design é LINQ. O padrão de expressão de consulta LINQ depende de delegados em todas as suas funcionalidades. Considere este exemplo simples:

```
C#

var smallNumbers = numbers.Where(n => n < 10);
```

Isso filtra a sequência de números apenas para aqueles menores que o valor 10. O `Where` método usa um delegado que determina quais elementos de uma sequência passam o filtro. Ao criar uma consulta LINQ, você fornece a implementação do delegado para essa finalidade específica.

O protótipo do método Where é:

```
C#

public static IEnumerable<TSource> Where<TSource> (this IEnumerable<TSource>
source, Func<TSource, bool> predicate);
```

Este exemplo é repetido com todos os métodos que fazem parte do LINQ. Todos eles dependem de delegados para o código que gerencia a consulta específica. Esse padrão de design de API é poderoso para aprender e entender.

Este exemplo simples ilustra como os delegados exigem muito pouco acoplamento entre componentes. Você não precisa criar uma classe derivada de uma classe base específica. Você não precisa implementar uma interface específica. O único requisito é fornecer a implementação de um método fundamental para a tarefa em questão.

Criar componentes próprios com delegados

Vamos aprimorar esse exemplo criando um componente que utiliza um design baseado em delegados.

Vamos definir um componente que pode ser usado para mensagens de log em um sistema grande. Os componentes da biblioteca podem ser usados em vários ambientes diferentes, em várias plataformas diferentes. Há muitos recursos comuns no componente que gerencia os logs. Ele precisará aceitar mensagens de qualquer componente no sistema. Essas mensagens terão prioridades diferentes, que o componente principal pode gerenciar. As mensagens devem ter carimbos de data/hora na sua forma final arquivada. Para cenários mais avançados, você pode filtrar mensagens pelo componente de origem.

Há um aspecto do recurso que mudará com frequência: onde as mensagens são gravadas. Em alguns ambientes, eles podem ser gravados no console de erros. Em outros, um arquivo. Outras possibilidades incluem armazenamento de banco de dados, logs de eventos do sistema operacional ou outro armazenamento de documentos.

Também há combinações de saída que podem ser usadas em cenários diferentes. Talvez você queira gravar mensagens no console e em um arquivo.

Um design baseado em delegados fornecerá muita flexibilidade e facilitará o suporte a mecanismos de armazenamento que podem ser adicionados no futuro.

Nesse design, o componente de log primário pode ser uma classe não virtual, até mesmo selada. Você pode conectar qualquer conjunto de delegados para gravar as mensagens em diferentes mídias de armazenamento. O suporte interno para delegados multicast facilita o suporte a cenários em que as mensagens devem ser gravadas em vários locais (um arquivo e um console).

Uma primeira implementação

Vamos começar com algo pequeno: a nova implementação aceitará novas mensagens e as gravará usando qualquer delegado vinculado. Você pode começar com um delegado que grava mensagens no console.

C#

```
public static class Logger
{
    public static Action<string>? WriteMessage;

    public static void LogMessage(string msg)
    {
        if (WriteMessage is not null)
            WriteMessage(msg);
    }
}
```

A classe estática acima é a coisa mais simples que pode funcionar. Precisamos gravar a implementação única para o método que grava mensagens no console:

```C#
public static class LoggingMethods
{
    public static void LogToConsole(string message)
    {
        Console.Error.WriteLine(message);
    }
}
```

Por fim, você precisa conectar o delegado anexando-o ao delegado WriteMessage declarado no agente:

```C#
Logger.WriteMessage += LoggingMethods.LogToConsole;
```

Práticas

Nosso exemplo até agora é bastante simples, mas ainda demonstra algumas das diretrizes importantes para projetos envolvendo delegados.

O uso dos tipos de delegado definidos na estrutura principal facilita o trabalho dos usuários com os delegados. Você não precisa definir novos tipos e os desenvolvedores que usam sua biblioteca não precisam aprender novos tipos de delegado especializados.

As interfaces usadas são as mais mínimas e flexíveis possíveis: para criar um novo logger de saída, você deve criar um método. Esse método pode ser um método estático ou um método de instância. Ele pode ter qualquer acesso.

Formatar saída

Vamos tornar essa primeira versão um pouco mais robusta e, em seguida, começar a criar outros mecanismos de log.

Em seguida, vamos adicionar alguns argumentos ao método para que sua `LogMessage()` classe de log crie mensagens mais estruturadas:

```C#
```

```
public enum Severity
{
    Verbose,
    Trace,
    Information,
    Warning,
    Error,
    Critical
}
```

C#

```
public static class Logger
{
    public static Action<string>? WriteMessage;

    public static void LogMessage(Severity s, string component, string msg)
    {
        var outputMsg = $"{DateTime.Now}\t{s}\t{component}\t{msg}";
        if (WriteMessage is not null)
            WriteMessage(outputMsg);
    }
}
```

Em seguida, vamos usar esse `Severity` argumento para filtrar as mensagens que são enviadas para a saída do log.

C#

```
public static class Logger
{
    public static Action<string>? WriteMessage;

    public static Severity LogLevel { get; set; } = Severity.Warning;

    public static void LogMessage(Severity s, string component, string msg)
    {
        if (s < LogLevel)
            return;

        var outputMsg = $"{DateTime.Now}\t{s}\t{component}\t{msg}";
        if (WriteMessage is not null)
            WriteMessage(outputMsg);
    }
}
```

Práticas

Você adicionou novos recursos à infraestrutura de registro em log. Como o componente do registrador está fracamente acoplado a qualquer mecanismo de saída, esses novos recursos podem ser adicionados sem impactar nenhum dos códigos que implementam o delegado do registrador.

À medida que você continuar criando isso, verá mais exemplos de como esse acoplamento flexível permite maior flexibilidade na atualização de partes do site sem nenhuma alteração em outros locais. De fato, em um aplicativo maior, as classes de saída do agente podem estar em um assembly diferente e nem mesmo precisar ser recriadas.

Criar um segundo mecanismo de saída

O componente Log está progredindo bem. Vamos adicionar mais um mecanismo de saída que registra mensagens em um arquivo. Esse será um mecanismo de saída de um pouco mais envolvido. Será uma classe que encapsula as operações de arquivo e garante que o arquivo seja sempre fechado após cada gravação. Isso garante que todos os dados sejam liberados para o disco após a geração de cada mensagem.

Aqui está o registrador baseado em arquivo:

C#

```
public class FileLogger
{
    private readonly string logPath;
    public FileLogger(string path)
    {
        logPath = path;
        Logger.WriteMessage += LogMessage;
    }

    public void DetachLog() => Logger.WriteMessage -= LogMessage;
    // make sure this can't throw.
    private void LogMessage(string msg)
    {
        try
        {
            using (var log = File.AppendText(logPath))
            {
                log.WriteLine(msg);
                log.Flush();
            }
        }
        catch (Exception)
        {
            // Hmm. We caught an exception while
            // logging. We can't really log the
            // problem (since it's the log that's failing).
```

```
            // So, while normally, catching an exception
            // and doing nothing isn't wise, it's really the
            // only reasonable option here.
        }
    }
}
```

Depois de ter criado esta classe, você pode instanciá-la e ela anexará seu método LogMessage ao componente Logger.

C#

```
var file = new FileLogger("log.txt");
```

Estes dois não são mutuamente exclusivos. Você pode anexar ambos os métodos de log e gerar mensagens para o console e para um arquivo.

C#

```
var fileOutput = new FileLogger("log.txt");
Logger.WriteMessage += LoggingMethods.LogToConsole; // LoggingMethods is the
static class we utilized earlier
```

Posteriormente, mesmo no mesmo aplicativo, você pode remover um dos delegados sem nenhum outro problema para o sistema:

C#

```
Logger.WriteMessage -= LoggingMethods.LogToConsole;
```

Práticas

Agora, você adicionou um segundo manipulador de saída para o subsistema de log. Este precisa de um pouco mais de infraestrutura para dar suporte correto ao sistema de arquivos. O delegado um método de instância. Também é um método privado. Não há necessidade de maior acessibilidade porque a infraestrutura delegada pode conectar os delegados.

Em segundo lugar, o design baseado em delegado habilita vários métodos de saída sem nenhum código extra. Você não precisa criar nenhuma infraestrutura adicional para dar suporte a vários métodos de saída. Eles simplesmente se tornam outro método na lista de invocação.

Dedique atenção especial ao código no método de saída do registro em log de arquivos. Ele é codificado para garantir que ele não gere exceções. Embora isso nem sempre seja estritamente

necessário, muitas vezes é uma boa prática. Se qualquer um dos métodos delegados gerar uma exceção, os delegados restantes que estão na invocação não serão invocados.

Como última observação, o registrador de arquivos deve gerenciar seus recursos, abrindo e fechando o arquivo para cada mensagem de log. Você pode optar por manter o arquivo aberto e implementar `IDisposable` para fechar o arquivo quando for concluído. Qualquer um dos métodos tem suas vantagens e desvantagens. Ambos criam um pouco mais de acoplamento entre as classes.

Nenhum código na `Logger` classe precisaria ser atualizado para dar suporte a qualquer um dos cenários.

Manipular delegados nulos

Por fim, vamos atualizar o método LogMessage para que ele seja robusto para esses casos quando nenhum mecanismo de saída for selecionado. A implementação atual gerará um `NullReferenceException` quando o `WriteMessage` delegado não tiver uma lista de invocação anexada. Você pode preferir um design que continua de forma discreta quando não há métodos anexados. Isso é fácil usando o operador condicional nulo, combinado com o `Delegate.Invoke()` método:

C#

```
public static void LogMessage(string msg)
{
    WriteMessage?.Invoke(msg);
}
```

O operador condicional nulo (`?.`) provoca um curto-circuito quando o operando esquerdo (`WriteMessage` nesse caso) é nulo, o que significa que nenhuma tentativa é feita para logar uma mensagem.

Você não encontrará o `Invoke()` método listado na documentação para `System.Delegate` ou `System.MulticastDelegate`. O compilador gera um método de tipo seguro `Invoke` para qualquer tipo de delegado declarado. Neste exemplo, isso significa que `Invoke` aceita um único argumento `string` e tem um tipo de retorno void.

Resumo das práticas

Você já viu o início de um componente de log que poderia ser expandido com outros gravadores e outros recursos. Com o uso de delegados no design, esses diferentes

componentes ficam acoplados de maneira flexível. Isso oferece várias vantagens. É fácil criar novos mecanismos de saída e anexá-los ao sistema. Esses outros mecanismos só precisam de um método: o método que grava a mensagem de log. É um design resiliente quando novos recursos são adicionados. O contrato necessário para qualquer gravador é implementar um método. Esse método pode ser estático ou de instância. Pode ser público, privado ou qualquer outro acesso legal.

A classe de agente pode fazer vários aprimoramentos ou alterações sem introduzir alterações interruptivas. Como qualquer classe, você não pode modificar a API pública sem o risco de interromper alterações. Mas, como o acoplamento entre o agente e qualquer mecanismo de saída ocorre somente por meio do delegado, nenhum outro tipo (como interfaces ou classes base) é envolvido. O acoplamento é o menor possível.

Próximo

Introdução a eventos

19/06/2025

Anterior

Eventos são, assim como delegados, um mecanismo de *associação tardia*. Na verdade, os eventos são criados com base no suporte de idioma para delegados.

Os eventos são uma maneira de um objeto difundir (para todos os componentes interessados no sistema) que algo aconteceu. Qualquer outro componente pode assinar o evento e ser notificado quando um evento é gerado.

Você provavelmente usou eventos em algumas de suas codificações. Muitos sistemas gráficos têm um modelo de evento para relatar a interação do usuário. Esses eventos relatariam movimento do mouse, pressionamentos de botão e interações semelhantes. Esse é um dos cenários mais comuns, mas não o único cenário em que eventos são usados.

Você pode definir eventos que devem ser gerados para suas classes. Uma consideração importante ao trabalhar com eventos é que pode não haver nenhum objeto registrado para um evento específico. Você deve escrever seu código para que ele não gere eventos quando nenhum ouvinte estiver configurado.

Assinar um evento também cria um acoplamento entre dois objetos (a origem do evento e o coletor do evento). Você precisa garantir que o coletor do evento cancele a assinatura da origem do evento quando não houver mais interesse nos eventos.

Metas de design para suporte a eventos

O design de linguagem para eventos tem como destino estas metas:

- Habilitar um acoplamento mínimo entre uma origem do evento e um coletor de eventos. Esses dois componentes podem ser escritos por organizações diferentes e podem até ser atualizados em agendas diferentes.
- Deve ser simples assinar um evento e cancelar a assinatura desse mesmo evento.
- As origens do evento devem dar suporte a vários assinantes de eventos. Elas também devem dar suporte a não ter assinantes de evento anexados.

Você pode ver que as metas para eventos são semelhantes às metas dos delegados. É por isso que o suporte à linguagem de eventos é baseado no suporte à linguagem delegada.

Suporte à linguagem para eventos

A sintaxe para definir eventos e assinar ou cancelar a inscrição de eventos é uma extensão da sintaxe para delegados.

Use a `event` palavra-chave para definir um evento:

```
C#

public event EventHandler<FileFoundArgs>? FileFound;
```

O tipo do evento (`EventHandler<FileListArgs>` neste exemplo) deve ser um tipo delegado. Há convenções que você deve seguir ao declarar um evento. Normalmente, o tipo de delegado do evento tem um retorno nulo. As declarações de evento devem ser um verbo ou uma locução verbal. Use o passado ao relatar eventos já ocorridos. Use um verbo tenso atual (por exemplo) `Closing` para relatar algo que está prestes a acontecer. Muitas vezes, usar o tempo presente indica que sua classe dá suporte a algum tipo de comportamento de personalização. Um dos cenários mais comuns é dar suporte ao cancelamento. Por exemplo, um `Closing` evento pode incluir um argumento que indica se a operação de fechamento deve continuar ou não. Outros cenários permitem que os chamadores modifiquem o comportamento atualizando as propriedades dos argumentos do evento. Você pode gerar um evento para indicar uma próxima ação proposta que um algoritmo executará. O manipulador de eventos pode exigir uma ação diferente modificando as propriedades do argumento de evento.

Quando você quiser acionar o evento, chame os manipuladores de eventos usando a sintaxe de invocação delegada:

```
C#

FileFound?.Invoke(this, new FileFoundArgs(file));
```

Conforme discutido na seção sobre delegados, o operador `?.` facilita a garantia de que você não tente acionar o evento quando não houver assinantes desse evento.

Você assina um evento usando o `+=` operador:

```
C#

var fileLister = new FileSearcher();
int filesFound = 0;

EventHandler<FileFoundArgs> onFileFound = (sender, eventArgs) =>
{
    Console.WriteLine(eventArgs.FoundFile);
    filesFound++;
};
```

```
fileLister.FileFound += onFileFound;
```

O método de manipulador normalmente tem o prefixo 'On' seguido pelo nome do evento, conforme mostrado no código anterior.

Você cancela a assinatura usando o `-=` operador:

```C#
fileLister.FileFound -= onFileFound;
```

É importante que você declare uma variável local para a expressão que representa o manipulador de eventos. Isso garante que o cancelamento da assinatura remova o manipulador. Se, em vez disso, você tiver usado o corpo da expressão lambda, você estará tentando remover um manipulador que nunca foi anexado, o que não faz nada.

No próximo artigo, você aprenderá mais sobre padrões de evento típicos e variações diferentes neste exemplo.

Próximo

Padrões de evento .NET padrão

18/06/2025

Anterior

Os eventos do .NET geralmente seguem alguns padrões conhecidos. Padronizar esses padrões significa que os desenvolvedores podem aplicar conhecimento desses padrões, que podem ser aplicados a qualquer programa de eventos do .NET.

Vamos examinar esses padrões padrão para que você tenha todo o conhecimento necessário para criar fontes de eventos padrão e assinar e processar eventos padrão em seu código.

Assinaturas de delegado de evento

A assinatura padrão de um delegado de evento do .NET é:

C#

```
void EventRaised(object sender, EventArgs args);
```

Essa assinatura padrão fornece informações sobre quando os eventos são usados:

- ***O tipo de retorno é nulo***. Os eventos podem ter de zero a muitos ouvintes. A geração de um evento notifica todos os ouvintes. Em geral, os ouvintes não fornecem valores em resposta aos eventos.
- ***Eventos indicam o remetente***: a assinatura do evento inclui o objeto que gerou o evento. Isso fornece a qualquer ouvinte um mecanismo para se comunicar com o remetente. O tipo de tempo de compilação de `sender` é `System.Object`, mas é provável que você conheça um tipo mais derivado que sempre estaria correto. Por convenção, use `object`.
- ***Os eventos empacotam mais informações em uma única estrutura***: o `args` parâmetro é um tipo derivado System.EventArgs que inclui quaisquer informações adicionais necessárias. (Você verá na próxima seção que essa convenção não é mais imposta.) Se o tipo de evento não precisar de mais argumentos, você ainda deverá fornecer ambos os argumentos. Há um valor especial, EventArgs.Empty que você deve usar para indicar que o evento não contém informações adicionais.

Vamos criar uma classe que lista os arquivos em um diretório ou em qualquer um de seus subdiretórios, que seguem um padrão. Esse componente aciona um evento para cada arquivo encontrado que corresponde ao padrão.

O uso de um modelo de evento fornece algumas vantagens de design. Você pode criar vários ouvintes de eventos que realizam ações diferentes quando um arquivo procurado é

encontrado. A combinação de diferentes ouvintes pode criar algoritmos mais robustos.

Aqui está a declaração de argumento de evento inicial para localizar um arquivo procurado:

```C#
public class FileFoundArgs : EventArgs
{
    public string FoundFile { get; }

    public FileFoundArgs(string fileName) => FoundFile = fileName;
}
```

Embora esse tipo se pareça com um tipo pequeno de somente dados, você deve seguir a convenção e torná-lo um tipo de referência (`class`). Isso significa que o objeto de argumento é passado por referência e todas as atualizações nos dados são visíveis para todos os assinantes. A primeira versão é um objeto imutável. É preferível tornar as propriedades em seu tipo de argumento de evento imutáveis. Dessa forma, um assinante não pode alterar os valores antes que outro assinante os veja. (Há exceções a essa prática, como você vê mais tarde.)

Em seguida, precisamos criar a declaração de evento na classe FileSearcher. Usar o tipo System.EventHandler<TEventArgs> significa que você não precisa criar mais uma definição de tipo. Você apenas usa uma especialização genérica.

Vamos preencher a classe FileSearcher para pesquisar arquivos que correspondam a um padrão e acionar o evento correto quando uma correspondência for descoberta.

```C#
public class FileSearcher
{
    public event EventHandler<FileFoundArgs>? FileFound;

    public void Search(string directory, string searchPattern)
    {
        foreach (var file in Directory.EnumerateFiles(directory, searchPattern))
        {
            FileFound?.Invoke(this, new FileFoundArgs(file));
        }
    }
}
```

Definir e gerar eventos semelhantes a campos

A maneira mais simples de adicionar um evento à sua classe é declarar esse evento como um campo público, como no exemplo anterior:

```C#
public event EventHandler<FileFoundArgs>? FileFound;
```

Isso parece estar declarando um campo público, o que parece ser uma prática orientada a objetos ruim. Você deseja proteger o acesso a dados por meio de propriedades ou métodos. Embora esse código possa parecer uma prática incorreta, o código gerado pelo compilador cria wrappers para que os objetos de evento só possam ser acessados de maneiras seguras. As únicas operações disponíveis em um evento semelhante a campo são *adicionar* e *remover* manipulador:

```C#
var fileLister = new FileSearcher();
int filesFound = 0;

EventHandler<FileFoundArgs> onFileFound = (sender, eventArgs) =>
{
    Console.WriteLine(eventArgs.FoundFile);
    filesFound++;
};

fileLister.FileFound += onFileFound;
```

```C#
fileLister.FileFound -= onFileFound;
```

Há uma variável local para o manipulador. Se você tiver usado o corpo de lambda, o manipulador `remove` não funcionará corretamente. Ela seria uma instância diferente do delegado e silenciosamente não faria nada.

O código fora da classe não pode gerar o evento, nem pode executar outras operações.

A partir do C# 14, os eventos podem ser declarados como membros parciais. Uma declaração de evento parcial deve incluir uma *declaração de definição* e uma *declaração de implementação*. A declaração de definição deve usar a sintaxe de evento semelhante a um campo. A declaração de implementação deve declarar os manipuladores `add` e `remove`.

Valor retornados de assinantes de evento

Sua versão simples está funcionando bem. Vamos adicionar outro recurso: cancelamento.

Quando você aciona o evento *Encontrado*, os ouvintes devem conseguir interromper o processamento posterior, caso este arquivo seja o último procurado.

Os manipuladores de eventos não retornam um valor, portanto, você precisa comunicar isso de outra maneira. O padrão de evento usa o objeto `EventArgs` para incluir campos que os assinantes de evento podem usar para comunicar o cancelamento.

Há dois padrões diferentes que podem ser usados, com base na semântica do contrato de cancelamento. Em ambos os casos, você adiciona um campo booliano aos EventArguments para o evento de arquivo encontrado.

Um padrão permitiria a qualquer assinante cancelar a operação. Para esse padrão, o novo campo é inicializado para `false`. Qualquer assinante pode alterá-lo para `true`. Após disparar o evento para todos os assinantes, o componente FileSearcher examina o valor booleano e toma medidas.

O segundo padrão apenas cancelaria a operação se todos os assinantes quisessem que a operação fosse cancelada. Nesse padrão, o novo campo é inicializado para indicar que a operação deve ser cancelada e qualquer assinante poderia alterá-lo para indicar que a operação deve continuar. Depois que todos os assinantes processam o evento gerado, o componente FileSearcher examina o booleano e toma uma atitude. Há uma etapa extra nesse padrão: o componente precisa saber se algum assinante respondeu ao evento. Se não houver nenhum assinante, o campo indicaria incorretamente um cancelamento.

Vamos implementar a primeira versão deste exemplo. Você precisa adicionar um campo booliano chamado `CancelRequested` ao tipo `FileFoundArgs`:

C#

```
public class FileFoundArgs : EventArgs
{
    public string FoundFile { get; }
    public bool CancelRequested { get; set; }

    public FileFoundArgs(string fileName) => FoundFile = fileName;
}
```

Esse novo campo é inicializado automaticamente para `false` para que você não cancele acidentalmente. A única alteração adicional no componente é verificar o sinalizador depois de acionar o evento, para ver se qualquer um dos assinantes solicitou um cancelamento:

C#

```
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;
    }
}
```

Uma vantagem desse padrão é que ele não é uma alteração significativa. Nenhum dos assinantes solicitou cancelamento antes, e ainda não solicitaram. Nenhum código do assinante requer atualizações, a menos que ele deseje dar suporte ao novo protocolo de cancelamento.

Vamos atualizar o assinante para que ele solicite um cancelamento, depois de encontrar o primeiro executável:

```C#
EventHandler<FileFoundArgs> onFileFound = (sender, eventArgs) =>
{
    Console.WriteLine(eventArgs.FoundFile);
    eventArgs.CancelRequested = true;
};
```

Adicionar outra declaração de evento

Vamos adicionar mais um recurso e demonstrar outras expressões de linguagem para eventos. Vamos adicionar uma sobrecarga do método `Search` que percorre todas os subdiretórios pesquisando arquivos.

Esse método pode ser uma operação longa em um diretório com muitos subdiretórios. Vamos adicionar um evento que é acionado no início de cada nova pesquisa de diretório. Esse evento permite que os assinantes acompanhem o progresso e atualizem o usuário quanto ao progresso. Todos os exemplos que você criou até agora são públicos. Vamos tornar esse evento um evento interno. Isso significa que você também pode tornar os tipos de argumentos internos.

Você começa criando a nova classe derivada eventArgs para relatar o novo diretório e o progresso.

```C#
internal class SearchDirectoryArgs : EventArgs
{
    internal string CurrentSearchDirectory { get; }
    internal int TotalDirs { get; }
```

```
    internal int CompletedDirs { get; }

    internal SearchDirectoryArgs(string dir, int totalDirs, int completedDirs)
    {
        CurrentSearchDirectory = dir;
        TotalDirs = totalDirs;
        CompletedDirs = completedDirs;
    }
}
```

Novamente, você pode seguir as recomendações para criar um tipo de referência imutável para os argumentos do evento.

Em seguida, defina o evento. Desta vez, você usará uma sintaxe diferente. Além de usar a sintaxe de campo, você pode criar explicitamente a propriedade de evento com manipuladores add e remove. Neste exemplo, você não precisa de código extra nesses manipuladores, mas isso mostra como você os criaria.

C#

```
internal event EventHandler<SearchDirectoryArgs> DirectoryChanged
{
    add { _directoryChanged += value; }
    remove { _directoryChanged -= value; }
}
private EventHandler<SearchDirectoryArgs>? _directoryChanged;
```

Em muitos aspectos, o código que você escreve aqui espelha o código que o compilador gera para as definições de evento de campo que você viu anteriormente. Você cria o evento usando uma sintaxe semelhante à das propriedades . Observe que os manipuladores têm nomes diferentes: `add` e `remove`. Esses acessadores são chamados para assinar o evento ou cancelar a assinatura do evento. Observe que você também deve declarar um campo de suporte particular para armazenar a variável de evento. Essa variável é inicializada como nula.

Em seguida, vamos adicionar a sobrecarga do método `Search` que percorre os subdiretórios e aciona os dois eventos. A maneira mais fácil é usar um argumento padrão para especificar que você deseja pesquisar todos os diretórios:

C#

```
public void Search(string directory, string searchPattern, bool searchSubDirs =
false)
{
    if (searchSubDirs)
    {
        var allDirectories = Directory.GetDirectories(directory, "*.*",
SearchOption.AllDirectories);
```

```
        var completedDirs = 0;
        var totalDirs = allDirectories.Length + 1;
        foreach (var dir in allDirectories)
        {
            _directoryChanged?.Invoke(this, new (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 (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;
    }
}
```

Neste momento, você pode executar o aplicativo, chamando a sobrecarga para pesquisar todos os subdiretórios. Não há assinantes no novo evento `DirectoryChanged`, mas usar o idioma `?.Invoke()` garante que ele funcione corretamente.

Vamos adicionar um manipulador para escrever uma linha que mostra o progresso na janela do console.

C#

```
fileLister.DirectoryChanged += (sender, eventArgs) =>
{
    Console.Write($"Entering '{eventArgs.CurrentSearchDirectory}'.");
    Console.WriteLine($" {eventArgs.CompletedDirs} of {eventArgs.TotalDirs}
completed...");
};
```

Você viu padrões que são seguidos em todo o ecossistema do .NET. Ao aprender esses padrões e convenções, você está escrevendo C# e .NET de forma idiomática rapidamente.

Confira também

- Introdução a eventos
- Design de evento
- Manipular e gerar eventos

Em seguida, você verá algumas alterações nesses padrões na versão mais recente do .NET.

Próximo

O padrão de evento do .NET Core atualizado

19/06/2025

Anterior

O artigo anterior discutia os padrões de evento mais comuns. O .NET Core tem um padrão mais relaxado. Nesta versão, a `EventHandler<TEventArgs>` definição não tem mais a restrição que `TEventArgs` deve ser uma classe derivada de `System.EventArgs`.

Isso aumenta a flexibilidade para você e é compatível com versões anteriores. Vamos começar com a flexibilidade. A implementação para System.EventArgs usa um método definido em System.Object, que é: MemberwiseClone(), e cria uma cópia superficial do objeto. Esse método deve usar a reflexão para implementar sua funcionalidade para qualquer classe derivada de `EventArgs`. Essa funcionalidade é mais fácil de criar em uma classe derivada específica. Isso significa efetivamente que derivar de System.EventArgs impõe uma restrição que limita seus designs, mas não oferece nenhum benefício adicional. Na verdade, você pode alterar as definições de `FileFoundArgs` e `SearchDirectoryArgs` para que elas não derivam de `EventArgs`. O programa funciona exatamente da mesma forma.

Você também pode alterar o `SearchDirectoryArgs` para um struct se fizer mais uma alteração:

C#

```
internal struct SearchDirectoryArgs
{
    internal string CurrentSearchDirectory { get; }
    internal int TotalDirs { get; }
    internal int CompletedDirs { get; }

    internal SearchDirectoryArgs(string dir, int totalDirs, int completedDirs) :
this()
    {
        CurrentSearchDirectory = dir;
        TotalDirs = totalDirs;
        CompletedDirs = completedDirs;
    }
}
```

A alteração extra é chamar o construtor sem parâmetros antes de entrar no construtor que inicializa todos os campos. Sem essa adição, as regras de C# relatariam que as propriedades estão sendo acessadas antes de serem atribuídas.

Você não deve alterar a `FileFoundArgs` de uma classe (tipo de referência) para um struct (tipo de valor). O protocolo para lidar com o cancelamento requer que você passe argumentos de eventos por referência. Se você fizesse a mesma alteração, a classe de pesquisa de arquivo nunca observaria as alterações feitas por qualquer um dos assinantes do evento. Uma nova cópia da estrutura seria usada para cada assinante e essa cópia seria uma cópia diferente daquela vista pelo objeto de pesquisa de arquivo.

Em seguida, vamos considerar como essa alteração pode ser compatível com versões anteriores. A remoção da restrição não afeta nenhum código existente. Qualquer tipo de argumento de evento existente ainda deriva de `System.EventArgs`. A compatibilidade com versões anteriores é uma das principais razões pelas quais elas continuam derivando de `System.EventArgs`. Todos os assinantes de eventos existentes são assinantes de um evento que seguiu o padrão clássico.

Seguindo uma lógica semelhante, qualquer tipo de argumento de evento criado agora não teria assinantes em nenhuma base de código existente. Novos tipos de evento que não derivam de `System.EventArgs` não quebram essas bases de código.

Eventos com assinantes assíncronos

Você tem um último padrão para aprender: como escrever corretamente os assinantes do evento que chamam o código assíncrono. O desafio é descrito no artigo em async e await. Métodos assíncronos podem ter um tipo de retorno nulo, mas isso não é recomendável. Quando o código do assinante de evento chama um método assíncrono, você não tem escolha a não ser criar um método `async void`. A assinatura do manipulador de eventos requer isso.

Você precisa reconciliar essa orientação oposta. De alguma forma, você deve criar um método seguro `async void` . As noções básicas do padrão que você precisa implementar são mostradas no seguinte código:

C#

```
worker.StartWorking += async (sender, eventArgs) =>
{
    try
    {
        await DoWorkAsync();
    }
    catch (Exception e)
    {
        //Some form of logging.
        Console.WriteLine($"Async task failure: {e.ToString()}");
        // Consider gracefully, and quickly exiting.
```

```
    }
};
```

Primeiro, observe que o manipulador está marcado como um manipulador assíncrono. Como está sendo atribuído a um tipo de delegado de manipulador de eventos, ele tem um tipo de retorno nulo. Isso significa que você deve seguir o padrão mostrado no manipulador e não permitir que exceções sejam descartadas do contexto do manipulador assíncrono. Como ele não retorna uma tarefa, não há nenhuma tarefa que pode relatar o erro entrando no estado de falha. Como o método é assíncrono, o método não pode gerar a exceção. (O método de chamada continua a execução porque é `async`.) O comportamento real do runtime é definido de forma diferente para ambientes diferentes. Ele pode encerrar o thread ou o processo que possui o thread ou deixar o processo em um estado indeterminado. Todos esses resultados potenciais são altamente indesejáveis.

Você deve encapsular a expressão `await` para a tarefa assíncrona em seu próprio bloco try. Se isso causar uma tarefa com falha, você pode registrar o erro em log. Se for um erro do qual seu aplicativo não pode se recuperar, você poderá sair do programa rapidamente e normalmente

Este artigo explicou as principais atualizações para o padrão de evento .NET. Você pode ver muitos exemplos das versões anteriores nas bibliotecas com as quais trabalha. No entanto, você também deve entender quais são os padrões mais recentes. Você pode ver o código concluído para o exemplo em Program.cs .

O próximo artigo desta série ajuda você a distinguir entre usar `delegates` e `events` em seus designs. São conceitos semelhantes e esse artigo ajuda você a tomar a melhor decisão para seus programas.

Próximo

Distinção entre Delegados e Eventos

19/06/2025

Anterior

Os desenvolvedores que são novos na plataforma .NET geralmente lutam ao decidir entre um design baseado `delegates` em um design baseado em `events`. A escolha de delegados ou eventos não é fácil, pois as duas características linguísticas são semelhantes. Os eventos são construídos até mesmo usando o suporte de linguagem para delegados. Uma declaração do manipulador de eventos declara um tipo de delegado.

Ambos oferecem um cenário de associação tardia: eles habilitam cenários em que um componente se comunica chamando um método que só é conhecido em tempo de execução. Ambos dão suporte a métodos de assinante único e múltiplo. Você pode encontrar esses termos chamados de suporte de transmissão única e multicast. Ambos dão suporte a sintaxe semelhante para adicionar e remover manipuladores. Por fim, acionar um evento e chamar um delegado usam exatamente a mesma sintaxe de chamada de método. Ambas até suportam a mesma sintaxe de método `Invoke()` para uso com o operador `?.`.

Com todas essas semelhanças, é fácil ter problemas para determinar quando usar qual.

Ouvir eventos é opcional

A consideração mais importante na determinação de qual recurso de linguagem usar é se deve ou não haver um assinante anexado. Se o seu código precisar chamar o código fornecido pelo assinante, você deverá usar um design baseado em delegados quando precisar implementar o retorno de chamada. Se seu código puder concluir todo o seu trabalho sem chamar nenhum assinante, você deverá usar um design baseado em eventos.

Considere os exemplos criados durante esta seção. O código que você criou usando `List.Sort()` deve receber uma função de comparador para classificar corretamente os elementos. Consultas de LINQ devem receber delegados para determinar quais elementos retornar. Ambos usaram um design criado com delegados.

Considere o evento `Progress`. Relata-se o progresso em uma tarefa. A tarefa continua se há ou não ouvintes. O `FileSearcher` é outro exemplo. Ele ainda pesquisaria e encontraria todos os arquivos procurados, mesmo sem nenhum assinante de eventos vinculado. Os controles de UX ainda funcionam corretamente, mesmo quando não há assinantes escutando os eventos. Ambos usam os designs baseados em eventos.

Valores retornados exigem delegados

Outra consideração é o protótipo do método que você deseja para o método delegado. Como você viu, todos os delegados usados para os eventos têm um tipo de retorno nulo. Há padrões para criar funções manipuladoras de eventos que transmitem informações de volta para as origens do evento por meio da modificação das propriedades do objeto de argumentos do evento. Embora esses idiomas funcionem, eles não são tão naturais quanto retornar um valor de um método.

Observe que essas duas heurísticas geralmente podem estar presentes: se o método delegado retornar um valor, isso afetará o algoritmo de alguma forma.

Eventos têm invocação privada

Classes diferentes daquela em que um evento está contido só podem adicionar e remover ouvintes de eventos; somente a classe que contém o evento pode invocar o evento. Eventos normalmente são membros de classe pública. Em comparação, os delegados geralmente são passados como parâmetros e armazenados como membros de classe privada, se forem armazenados.

Os ouvintes de eventos geralmente têm tempos de vida mais longos

O tempo de vida mais longo dos ouvintes de eventos é uma justificativa um pouco mais fraca. Entretanto, você pode descobrir que designs baseados em eventos são mais naturais quando a origem do evento for gerar eventos durante um longo período de tempo. Você pode ver exemplos de design baseado em eventos para controles de UX em muitos sistemas. Depois de inscrever-se em um evento, a fonte de eventos poderá gerar eventos ao longo do tempo de vida do programa. (Você pode cancelar a assinatura de eventos quando não precisar mais deles.)

Compare isso com vários designs baseados em delegados, em que um delegado é usado como um argumento para um método e o delegado não é usado depois que o método é retornado.

Avaliar cuidadosamente

As considerações acima não são regras duras e rápidas. Em vez disso, eles representam diretrizes que podem ajudá-lo a decidir qual escolha é melhor para seu uso específico. Como elas são semelhantes, você pode até mesmo fazer protótipos das suas e considerar com qual

seria mais natural trabalhar. Ambas lidam bem com cenários de associação tardia. Use aquele que comunica melhor seu design.

Controle de versão em C#

Artigo • 01/10/2024

Neste tutorial, você aprenderá o que significa o controle de versão no .NET. Você também aprenderá sobre os fatores a serem considerados ao fazer o controle de versão de sua biblioteca, bem como ao atualizar para uma nova versão de uma biblioteca.

Versão da linguagem

O compilador C# faz parte do SDK do .NET. Por padrão, o compilador escolhe a versão da linguagem C# que corresponde ao TFM escolhido para seu projeto. Se a versão do SDK for maior do que a estrutura escolhida, o compilador poderá usar uma versão de linguagem maior. Você pode alterar o padrão definindo o elemento `LangVersion` em seu projeto. Você pode aprender como em nosso artigo sobre opções do compilador.

> ⚠ **Aviso**
>
> Não é recomendável definir o elemento `LangVersion` como `latest`. A configuração `latest` significa que o compilador instalado usa sua versão mais recente. Isso pode mudar conforme o computador, tornando as compilações não confiáveis. Além disso, ele habilita recursos de linguagem que podem exigir recursos de tempo de execução ou biblioteca não incluídos no SDK atual.

Criando bibliotecas

Como um desenvolvedor que criou a bibliotecas .NET para uso público, provavelmente você esteve em situações em que precisa distribuir novas atualizações. Como você realiza esse processo é muito importante, pois você precisa garantir que haja uma transição suave do código existente para a nova versão da biblioteca. Aqui estão Vários aspectos a considerar ao criar uma nova versão:

Controle de Versão Semântico

Controle de versão semântico (SemVer, de forma abreviada) é uma convenção de nomenclatura aplicada a versões de sua biblioteca para indicar eventos com marcos específicos. Idealmente, as informações de versão que você fornece a sua biblioteca devem ajudar os desenvolvedores a determinar a compatibilidade com seus projetos que usam versões mais antigas da mesma biblioteca.

A abordagem mais básica ao SemVer é o formato de 3 componentes `MAJOR.MINOR.PATCH`, em que:

- `MAJOR` é incrementado quando você faz alterações em APIs incompatíveis
- `MINOR` é incrementado quando você adiciona funcionalidades de maneira compatível com versões anteriores
- `PATCH` é incrementado quando você faz correções de bugs compatíveis com versões anteriores

Também há maneiras de especificar outros cenários, como versões de pré-lançamento, ao aplicar informações de versão à biblioteca do .NET.

Compatibilidade com versões anteriores

Conforme você lança novas versões de sua biblioteca, a compatibilidade com versões anteriores provavelmente será uma de suas principais preocupações. Uma nova versão da biblioteca será compatível com a origem de uma versão anterior se o código que depende da versão anterior puder, quando recompilado, trabalhar com a nova versão. Uma nova versão da biblioteca será compatível de forma binária se um aplicativo que dependia da versão anterior puder, sem recompilação, trabalhar com a nova versão.

Aqui estão algumas coisas a serem consideradas ao tentar manter a compatibilidade com versões mais antigas de sua biblioteca:

- Métodos virtuais: quando você torna um método em virtual não virtual na nova versão, significa que projetos que substituem esse método precisarão ser atualizados. Essa é uma alteração muito grande e significativa que é altamente desaconselhável.
- Assinaturas de método: quando atualizar o comportamento de um método exigir que você altere também sua assinatura, você deve criar uma sobrecarga para que o código que chamar esse método ainda funcione. Você sempre pode manipular a assinatura de método antiga para chamar a nova assinatura de método para que a implementação permaneça consistente.
- Atributo obsoleto: você pode usar esse atributo no seu código para especificar classes ou membros da classe que foram preteridos e provavelmente serão removidos em versões futuras. Isso garante que os desenvolvedores que utilizam sua biblioteca estarão melhor preparados para alterações significativas.
- Argumentos de método opcionais: quando você tornar argumentos de método que antes eram opcionais em compulsórios ou alterar seu valor padrão, todo código que não fornece esses argumentos precisará ser atualizado.

> ⓘ **Observação**
>
> Tornar argumentos compulsórios em opcionais deve ter muito pouco efeito, especialmente se não alterar o comportamento do método.

Quanto mais fácil for para os usuários atualizarem para a nova versão da sua biblioteca, mais provável será que eles atualizem o quanto antes.

Arquivo de Configuração do Aplicativo

Como um desenvolvedor de .NET, há uma chance muito grande de você já ter encontrado o arquivo o app.config na maioria dos tipos de projeto. Esse arquivo de configuração simples pode fazer muita diferença para melhorar a distribuição de novas atualizações. Em geral, você deve projetar suas bibliotecas de forma que as informações que provavelmente serão alteradas regularmente sejam armazenadas no arquivo `app.config`. Dessa forma, quando essas informações forem atualizadas, o arquivo de configuração de versões mais antigas só precisa ser substituído pelo novo, sem a necessidade de recompilar a biblioteca.

Consumindo bibliotecas

Como um desenvolvedor que consome bibliotecas .NET criadas por outros desenvolvedores, vocês provavelmente está ciente de que uma nova versão de uma biblioteca pode não ser totalmente compatível com seu projeto e pode acabar precisando atualizar seu código para trabalhar com essas alterações.

Para sua sorte, o ecossistema do C# e do .NET tem recursos e técnicas que permitem facilmente atualizar nosso aplicativo para trabalhar com novas versões das bibliotecas que podem introduzir alterações interruptivas.

Redirecionamento de associação de assembly

Você pode usar o arquivo *app.config* para atualizar a versão de uma biblioteca que seu aplicativo usa. Adicionando o que é chamado de *redirecionamento de associação*, você pode usar a nova versão da biblioteca sem precisar recompilar seu aplicativo. O exemplo a seguir mostra como atualizar o arquivo *app.config* de seu aplicativo para uso com a versão de patch `1.0.1` de `ReferencedLibrary` em vez da versão `1.0.0` com que ele foi compilado originalmente.

```
XML
```

```
<dependentAssembly>
    <assemblyIdentity name="ReferencedLibrary"
publicKeyToken="32ab4ba45e0a69a1" culture="en-us" />
    <bindingRedirect oldVersion="1.0.0" newVersion="1.0.1" />
</dependentAssembly>
```

> ⓘ **Observação**
>
> Essa abordagem só funcionará se a nova versão do `ReferencedLibrary` for compatível de forma binária com seu aplicativo. Consulte a seção [Compatibilidade com versões anteriores] acima para ver as alterações importantes ao determinar a compatibilidade.

novo

Você usa o modificador `new` para ocultar membros herdados de uma classe base. Essa é uma maneira das classes derivadas responderem a atualizações em classes base.

Veja o exemplo seguinte:

C#

```
public class BaseClass
{
    public void MyMethod()
    {
        Console.WriteLine("A base method");
    }
}

public class DerivedClass : BaseClass
{
    public new void MyMethod()
    {
        Console.WriteLine("A derived method");
    }
}

public static void Main()
{
    BaseClass b = new BaseClass();
    DerivedClass d = new DerivedClass();

    b.MyMethod();
    d.MyMethod();
}
```

Saída

Console

```
A base method
A derived method
```

No exemplo acima, você pode ver como `DerivedClass` oculta o método `MyMethod` presente em `BaseClass`. Isso significa que quando uma classe base na nova versão de uma biblioteca adiciona um membro que já existe em sua classe derivada, você pode simplesmente usar o modificador `new` no membro de sua classe derivada para ocultar o membro da classe base.

Quando nenhum modificador `new` é especificado, uma classe derivada ocultará por padrão membros conflitantes em uma classe base e, embora um aviso do compilador seja gerado, o código ainda será compilado. Isso significa que simplesmente adicionar novos membros a uma classe existente torna a nova versão da biblioteca compatível com a origem e de forma binária com o código que depende dela.

override

O modificador `override` significa que uma implementação derivada estende a implementação de um membro da classe base, em vez de ocultá-lo. O membro da classe base precisa ter o modificador `virtual` aplicado a ele.

C#

```
public class MyBaseClass
{
    public virtual string MethodOne()
    {
        return "Method One";
    }
}

public class MyDerivedClass : MyBaseClass
{
    public override string MethodOne()
    {
        return "Derived Method One";
    }
}

public static void Main()
{
    MyBaseClass b = new MyBaseClass();
    MyDerivedClass d = new MyDerivedClass();
```

```
    Console.WriteLine("Base Method One: {0}", b.MethodOne());
    Console.WriteLine("Derived Method One: {0}", d.MethodOne());
}
```

Saída

Console

```
Base Method One: Method One
Derived Method One: Derived Method One
```

O modificador `override` é avaliado em tempo de compilação e o compilador gerará um erro se não encontrar um membro virtual para substituir.

Seu conhecimento sobre as técnicas discutidas e sua compreensão das situações em que usá-las, farão muita diferença para facilitar a transição entre versões de uma biblioteca.

Instruções (C#)

19/06/2025

Na seção Como fazer o Guia do C#, você pode encontrar respostas rápidas para perguntas comuns. Em alguns casos, os artigos podem ser listados em várias seções. Queríamos torná-los fáceis de encontrar para vários caminhos de pesquisa.

Conceitos gerais do C#

Há várias dicas e truques que são práticas comuns do desenvolvedor em C#:

- Inicialize objetos usando um inicializador de objeto.
- Use a sobrecarga de operador.
- Implementar e chamar um método de extensão personalizado.
- Crie um novo método para um enum tipo usando métodos de extensão.

Membros de classe, registro e struct

Você cria classes, registros e structs para implementar seu programa. Essas técnicas são comumente usadas ao escrever classes, registros ou structs.

- Declare propriedades implementadas automaticamente.
- Declare e use propriedades de leitura/gravação.
- Definir constantes.
- Substitua o método ToString para fornecer saída de cadeia de caracteres.
- Definir propriedades abstratas.
- Use os recursos da documentação xml para documentar seu código.
- Implemente explicitamente os membros da interface para manter sua interface pública concisa.
- Implemente membros de duas interfaces explicitamente.

Trabalhar com coleções

Esses artigos ajudam você a trabalhar com coleções de dados.

- Inicialize um dicionário com um inicializador de coleção.

Trabalhando com cadeias de caracteres

Cadeias de caracteres são o tipo de dados fundamental usado para exibir ou manipular texto. Esses artigos demonstram práticas comuns com cadeias de caracteres.

- Comparar cadeias de caracteres.
- Modifique o conteúdo de uma cadeia de caracteres.
- Determine se uma cadeia de caracteres representa um número.
- Use String.Split para separar cadeias de caracteres.
- Combine várias cadeias de caracteres em uma.
- Pesquise texto em uma cadeia de caracteres.

Converter entre tipos

Talvez seja necessário converter um objeto em um tipo diferente.

- Determine se uma cadeia de caracteres representa um número.
- Converta entre cadeias de caracteres que representam números hexadecimais e o número.
- Converter uma cadeia de caracteres em um DateTime.
- Converter uma matriz de bytes em um int.
- Converter uma cadeia de caracteres em um número.
- Use a correspondência de padrões, os operadores as e is para converter com segurança para um tipo diferente.
- Defina conversões de tipo personalizadas.
- Determine se um tipo é um tipo de valor anulável.
- Converter entre tipos de valor anuláveis e não anuláveis.

Comparações de ordem e igualdade

Você pode criar tipos que definem suas próprias regras para igualdade ou definem uma ordenação natural entre objetos desse tipo.

- Teste para igualdade baseada em referência.
- Defina a igualdade baseada em valor para um tipo.

Tratamento de exceções

Programas .NET relatam que os métodos não concluíram seu trabalho com sucesso ao lançar exceções. Nestes artigos, você aprenderá a trabalhar com exceções.

- Manipular exceções usando try e catch.
- Limpar recursos usando cláusulas finally.

- Recupere com base em exceções não CLS (Common Language Specification).

Delegados e eventos

Delegados e eventos fornecem uma funcionalidade para estratégias que envolvem blocos de código fracamente acoplados.

- Declarar, instanciar e usar delegados.
- Combine delegados multicast.

Os eventos fornecem um mecanismo para publicar ou assinar notificações.

- Assine e cancele a assinatura de eventos.
- Implementar eventos declarados em interfaces.
- Em conformidade com as diretrizes do .NET quando seu código publica eventos.
- Gere eventos definidos nas classes de base de classes derivadas.
- Implementar acessadores de eventos personalizados.

Práticas do LINQ

O LINQ permite que você escreva código para consultar qualquer fonte de dados que dê suporte ao padrão de expressão de consulta LINQ. Esses artigos ajudam você a entender o padrão e trabalhar com diferentes fontes de dados.

- Consultar uma coleção.
- Use var em expressões de consulta.
- Retornar subconjuntos de propriedades de elemento de uma consulta.
- Escreva consultas com filtragem complexa.
- Classificar elementos de uma fonte de dados.
- Classificar elementos em várias chaves.
- Controlar o tipo de uma projeção.
- Contar ocorrências de um valor em uma sequência de origem.
- Calcular valores intermediários.
- Depurar resultados de consulta vazios.
- Adicione métodos personalizados a consultas LINQ.

Vários threads e processamento assíncrono

Programas modernos geralmente usam operações assíncronas. Esses artigos ajudarão você a aprender a usar essas técnicas.

- Melhorar o desempenho assíncrono usando System.Threading.Tasks.Task.WhenAll.
- Faça várias solicitações da Web em paralelo usando async e await.
- Use um pool de thread.

Argumentos da linha de comando para o programa

Normalmente, os programas C# têm argumentos de linha de comando. Estes artigos ensinam você a acessar e processar esses argumentos de linha de comando.

- Recuperar todos os argumentos de linha de comando com for.

Como separar cadeias de caracteres usando String.Split em C#

Artigo • 07/11/2024

O método String.Split cria uma matriz de subcadeias, dividindo a cadeia de caracteres de entrada com base em um ou mais delimitadores. Esse método geralmente é a maneira mais fácil de separar uma cadeia de caracteres em limites de palavra. Ele também é usado para dividir cadeias de caracteres em outros caracteres específicos ou cadeias de caracteres.

> **Observação**
>
> Os exemplos de C# neste artigo são executados no executador de código embutido Try.NET e no playground. Clique no botão **Executar** para executar um exemplo em uma janela interativa. Ao executar o código, é possível modificá-lo e executar o código modificado clicando em **Executar** novamente. O código modificado será executado na janela interativa ou, se a compilação falhar, a janela interativa exibirá todos as mensagens de erro do compilador C#.

Exemplos de String.Split

O código a seguir divide uma frase comum em uma matriz de cadeias de caracteres para cada palavra.

C#

```
string phrase = "The quick brown fox jumps over the lazy dog.";
string[] words = phrase.Split(' ');

foreach (var word in words)
{
    System.Console.WriteLine($"<{word}>");
}
```

Cada instância de um caractere separador produz um valor na matriz retornada. Como as matrizes em C# são indexadas a zero, cada cadeia de caracteres na matriz é indexada de 0 até o valor retornado pela propriedade Array.Length menos 1:

C#

```
string phrase = "The quick brown fox jumps over the lazy dog.";
string[] words = phrase.Split(' ');

for (int i = 0; i < words.Length; i++)
{
    System.Console.WriteLine($"Index {i}: <{words[i]}>");
}
```

Caracteres separadores consecutivos produzem a cadeia de caracteres vazia como um valor na matriz retornada. Você pode ver como uma cadeia de caracteres vazia é criada no exemplo a seguir, que usa o caractere de espaço como separador.

C#

```
string phrase = "The quick brown    fox     jumps over the lazy dog.";
string[] words = phrase.Split(' ');

foreach (var word in words)
{
    System.Console.WriteLine($"<{word}>");
}
```

Esse comportamento facilita para formatos como arquivos CSV (valores separados por vírgula) que representam dados de tabela. Vírgulas consecutivas representam uma coluna em branco.

Você pode passar um parâmetro StringSplitOptions.RemoveEmptyEntries opcional para excluir as cadeias de caracteres vazias da matriz retornada. Para um processamento mais complicado da coleção retornada, você pode usar o LINQ para manipular a sequência de resultado.

O String.Split pode usar vários caracteres separadores. O exemplo a seguir utiliza espaços, vírgulas, pontos, dois pontos e tabulações como caracteres de separação, que são passados para Split em uma matriz. O loop, na parte inferior do código, exibe cada uma das palavras na matriz retornada.

C#

```
char[] delimiterChars = { ' ', ',', '.', ':', '\t' };

string text = "one\ttwo three:four,five six seven";
System.Console.WriteLine($"Original text: '{text}'");

string[] words = text.Split(delimiterChars);
System.Console.WriteLine($"{words.Length} words in text:");

foreach (var word in words)
```

```
{
    System.Console.WriteLine($"<{word}>");
}
```

As instâncias consecutivas de qualquer separador produzem a cadeia de caracteres vazia na matriz de saída:

C#

```
char[] delimiterChars = { ' ', ',', '.', ':', '\t' };

string text = "one\ttwo :,five six seven";
System.Console.WriteLine($"Original text: '{text}'");

string[] words = text.Split(delimiterChars);
System.Console.WriteLine($"{words.Length} words in text:");

foreach (var word in words)
{
    System.Console.WriteLine($"<{word}>");
}
```

O String.Split pode receber uma matriz de cadeias de caracteres (sequências de caracteres que atuam como separadores para analisar a cadeia de caracteres de destino, em vez de um único caractere).

C#

```
string[] separatingStrings = { "<<", "..." };

string text = "one<<two......three<four";
System.Console.WriteLine($"Original text: '{text}'");

string[] words = text.Split(separatingStrings,
System.StringSplitOptions.RemoveEmptyEntries);
System.Console.WriteLine($"{words.Length} substrings in text:");

foreach (var word in words)
{
    System.Console.WriteLine(word);
}
```

Usar o GitHub Copilot para dividir uma string

Você pode usar o GitHub Copilot no IDE para gerar código para dividir cadeias de caracteres usando `String.Split` em C#.

Se você estiver usando o Visual Studio 2022 versão 17.8 ou posterior, poderá experimentar o GitHub Copilot controlado por IA no Visual Studio para gerar código para dividir uma cadeia de caracteres de entrada em subcadeias de caracteres com base em um ou mais delimitadores. Envie sua pergunta como um prompt na janela de bate-papo do Copilot, como no exemplo a seguir. Você também pode enviar prompts usando o bate-papo em linha na própria janela do editor.

> **Observação**
>
> O GitHub Copilot é alimentado por IA, portanto é passível de surpresas e erros. Verifique qualquer código gerado ou sugestões. Para obter mais informações sobre o uso geral do GitHub Copilot, o impacto do produto, a supervisão humana e a privacidade, consulte Perguntas frequentes sobre o GitHub Copilot.

O texto a seguir mostra um exemplo de prompt para o Copilot Chat:

> Gere código C# para usar Split.String dividir uma cadeia de caracteres em subcadeias de caracteres. A string de entrada é "Você ganha alguns. Você perde alguns." Os delimitadores são espaço e ponto. Forneça um exemplo de saída.

Você pode personalizar o prompt para usar cadeias de caracteres e delimitadores de acordo com seus requisitos.

Você pode usar recursos de bate-papo, como comandos de barra, referências a arquivos, métodos ou classes e threads, para definir a intenção e obter melhores respostas com o contexto com escopo. Para um arquivo existente que está aberto no IDE, você pode solicitar o GitHub Copilot usando o chat embutido com `/generate code to split string1 in #Filename by delimiter spaces` o .

A saída a seguir mostra um exemplo de resposta do Copilot Chat:

Saída

```
Here is an example of how you can use the Split method in C# to split the
string "You win some. You lose some." using space and period as delimiters:

using System;

class Program
{
    static void Main()
    {
        string input = "You win some. You lose some.";
        char[] delimiters = new char[] { ' ', '.' };
```

```
        string[] substrings = input.Split(delimiters,
StringSplitOptions.RemoveEmptyEntries);

        foreach (string substring in substrings)
        {
            Console.WriteLine(substring);
        }
    }
}

**Example Output**
You
win
some
You
lose
some
```

Quando o Copilot retorna um bloco de código, a resposta inclui opções para copiar o código, inserir o código em um novo arquivo ou visualizar a saída do código.

> **Observação**
>
> Seus resultados podem ser diferentes do que é mostrado nas respostas de exemplo. Os modelos de IA não são determinísticos, ou seja, eles podem retornar respostas diferentes quando fazem a mesma pergunta. Isso pode ser devido a aprendizado e adaptação adicionais ao longo do tempo, variação de idioma, mudanças no contexto, como seu histórico de bate-papo e muito mais.

GitHub Copilot Chat

e. You lose some." Delimiters are space and period. Provide example output

Generate C# code to use Split.String to split a string into substrings. Input string is "You win some. You lose some." Delimiters are space and period. Provide example output

GitHub Copilot

Here is an example of how you can use the Split method in C# to split the string "You win some. You lose some." using space and period as delimiters:

```
using System;

class Program
{
    static void Main()
    {
        string input = "You win some. You lose some.";
        char[] delimiters = new char[] { ' ', '.' };
        string[] substrings = input.Split(delimiters, StringSplitOptions.RemoveE

        foreach (string substring in substrings)
        {
            Console.WriteLine(substring);
        }
    }
}
```

Insert in new file Preview

Example Output:

```
You
win
some
You
lose
some
```

Preview

How can I check if a string contains a specific substring in C#?

References Was this helpful?

AI suggestions might be inaccurate

Para saber mais, veja:

- Central de Confiabilidade do GitHub Copilot
- GitHub Copilot no Visual Studio
- GitHub Copilot no VS Code

Confira também

- Extrair elementos de uma cadeia de caracteres
- Cadeias de caracteres
- Expressões regulares do .NET

Como concatenar várias cadeias de caracteres (Guia de C#)

Artigo • 18/02/2025

Concatenação é o processo de acrescentar uma cadeia de caracteres ao final de outra cadeia de caracteres. Você concatena cadeias de caracteres usando o operador `+`. Para literais e constantes de string, a concatenação ocorre em tempo de compilação; não há concatenação em tempo de execução. Para variáveis de cadeia de caracteres, a concatenação ocorre somente em tempo de execução.

> **Observação**
>
> Os exemplos de C# neste artigo são executados no executador de código embutido Try.NET e no playground. Clique no botão **Executar** para executar um exemplo em uma janela interativa. Ao executar o código, é possível modificá-lo e executar o código modificado clicando em **Executar** novamente. O código modificado será executado na janela interativa ou, se a compilação falhar, a janela interativa exibirá todos as mensagens de erro do compilador C#.

> **Dica**
>
> Você pode usar a assistência de IA para concatenar cadeias de caracteres com o GitHub Copilot.

Literais de cadeia de caracteres

O exemplo a seguir divide um literal de cadeia de caracteres longo em cadeias de caracteres menores para melhorar a legibilidade no código-fonte. O código concatena as cadeias de caracteres menores para criar o literal de cadeia de caracteres longa. As partes são concatenadas em uma única cadeia de caracteres em tempo de compilação. Não há custo de desempenho durante a execução, independentemente do número de cadeias de caracteres envolvidas.

C#

```
// Concatenation of literals is performed at compile time, not run time.
string text = "Historically, the world of data and the world of objects " +
"have not been well integrated. Programmers work in C# or Visual Basic " +
"and also in SQL or XQuery. On the one side are concepts such as classes, " +
"objects, fields, inheritance, and .NET Framework APIs. On the other side " +
```

```
"are tables, columns, rows, nodes, and separate languages for dealing with " +
"them. Data types often require translation between the two worlds; there are " +
"different standard functions. Because the object world has no notion of query, a
" +
"query can only be represented as a string without compile-time type checking or "
+
"IntelliSense support in the IDE. Transferring data from SQL tables or XML trees
to " +
"objects in memory is often tedious and error-prone.";

Console.WriteLine(text);
```

Operadores `+` e `+=`

Para concatenar variáveis de cadeia de caracteres, você pode usar os operadores `+` ou `+=`, a interpolação de cadeia de caracteres ou os métodos String.Format, String.Concat, String.Join ou StringBuilder.Append. O operador `+` é fácil de usar e torna o código intuitivo. Mesmo ao usar vários operadores `+` em uma instrução, o conteúdo da cadeia de caracteres será copiado apenas uma vez. O código a seguir mostra dois exemplos de como usar os operadores `+` e `+=` para concatenar cadeias de caracteres:

C#

```
string userName = "<Type your name here>";
string dateString = DateTime.Today.ToShortDateString();

// Use the + and += operators for one-time concatenations.
string str = "Hello " + userName + ". Today is " + dateString + ".";
Console.WriteLine(str);

str += " How are you today?";
Console.WriteLine(str);
```

Interpolação de cadeia de caracteres

Em algumas expressões, é mais fácil concatenar cadeias de caracteres usando a interpolação de cadeia de caracteres, conforme mostra o seguinte código:

C#

```
string userName = "<Type your name here>";
string date = DateTime.Today.ToShortDateString();

// Use string interpolation to concatenate strings.
string str = $"Hello {userName}. Today is {date}.";
Console.WriteLine(str);
```

```
str = $"{str} How are you today?";
Console.WriteLine(str);
```

> ⓘ **Observação**
>
> Em operações de concatenação de cadeia de caracteres, o compilador C# trata uma cadeia de caracteres nula da mesma maneira que uma cadeia de caracteres vazia.

Você pode usar a interpolação de cadeia de caracteres para inicializar uma cadeia de caracteres constante quando todas as expressões usadas para espaços reservados também são cadeias de caracteres constantes.

`String.Format`

Outro método para concatenar cadeias de caracteres é o String.Format. Esse método funciona bem quando você está criando uma cadeia de caracteres direto de algumas cadeias de caracteres de componente.

`StringBuilder`

Em outros casos, você pode estar combinando cadeias de caracteres em um loop em que o número real de cadeias de caracteres de origem pode ser grande. A classe StringBuilder foi projetada para esses cenários. O código a seguir usa o método Append da classe StringBuilder para concatenar cadeias de caracteres.

C#

```
// Use StringBuilder for concatenation in tight loops.
var sb = new StringBuilder();
for (int i = 0; i < 20; i++)
{
    sb.AppendLine(i.ToString());
}
Console.WriteLine(sb.ToString());
```

Você pode ler mais sobre os motivos para escolher a concatenação de cadeia de caracteres ou a classe StringBuilder.

`String.Concat` ou `String.Join`

Outra opção para unir cadeias de caracteres de uma coleção é usar o método String.Concat. Utilize o método String.Join se um delimitador deve separar strings de origem. O código a seguir combina uma matriz de palavras usando os dois métodos:

```
C#

string[] words = ["The", "quick", "brown", "fox", "jumps", "over", "the", "lazy",
"dog."];

var unreadablePhrase = string.Concat(words);
Console.WriteLine(unreadablePhrase);

var readablePhrase = string.Join(" ", words);
Console.WriteLine(readablePhrase);
```

LINQ e `Enumerable.Aggregate`

Por fim, você pode usar LINQ e o método Enumerable.Aggregate para unir cadeias de caracteres de uma coleção. Esse método combina as cadeias de caracteres de origem usando uma expressão lambda. A expressão lambda faz o trabalho de adicionar cada cadeia de caracteres ao acúmulo existente. O exemplo a seguir combina uma matriz de palavras, adicionando um espaço entre cada palavra na matriz:

```
C#

string[] words = ["The", "quick", "brown", "fox", "jumps", "over", "the", "lazy",
"dog."];

var phrase = words.Aggregate((partialPhrase, word) =>$"{partialPhrase} {word}");
Console.WriteLine(phrase);
```

Essa opção pode causar mais alocações do que outros métodos para concatenar coleções, pois cria uma cadeia de caracteres intermediária para cada iteração. Se a otimização do desempenho for crítica, considere a classe StringBuilder ou o String.Concat ou o método String.Join para concatenar uma coleção, em vez de `Enumerable.Aggregate`.

Usar o GitHub Copilot para concatenar cadeias de caracteres

Você pode usar o GitHub Copilot em seu IDE para gerar código C# para concatenar várias cadeias de caracteres. Você pode personalizar o prompt para especificar cadeias de caracteres e o método a ser usado de acordo com seus requisitos.

O texto a seguir mostra um exemplo de prompt para o Copilot Chat:

Prompt do Copilot

```
Generate C# code to use String.Format to build an output string "Hi x, today's
date is y. You are z years old." where x is "John",  y is today's date and z is
the birthdate January 1, 2000. The final string should show date in the full
format mm/dd/yyyy. Show output.
```

O GitHub Copilot é alimentado por IA, portanto é passível de surpresas e erros. Para obter mais informações, confira Perguntas frequentes sobre o Copilot.

Saiba mais sobre GitHub Copilot no Visual Studio e GitHub Copilot no VS Code.

Confira também

- String
- StringBuilder
- Cadeias de caracteres

Como pesquisar em cadeias de caracteres

Artigo • 10/05/2023

Você pode usar duas estratégias principais para pesquisar texto em cadeias de caracteres. Os métodos da classe String pesquisam por um texto específico. Expressões regulares pesquisam por padrões no texto.

> **Observação**
>
> Os exemplos de C# neste artigo são executados no executador de código embutido Try.NET e no playground. Clique no botão **Executar** para executar um exemplo em uma janela interativa. Ao executar o código, é possível modificá-lo e executar o código modificado clicando em **Executar** novamente. O código modificado será executado na janela interativa ou, se a compilação falhar, a janela interativa exibirá todos as mensagens de erro do compilador C#.

O tipo string, que é um alias para a classe System.String, fornece uma série de métodos úteis para pesquisar o conteúdo de uma cadeia de caracteres. Entre elas estão Contains, StartsWith, EndsWith, IndexOf e LastIndexOf. A classe System.Text.RegularExpressions.Regex fornece um vocabulário avançado para pesquisar por padrões de texto. Neste artigo, você aprenderá essas técnicas e como escolher o melhor método para suas necessidades.

Uma cadeia de caracteres contém texto?

Os métodos String.Contains, String.StartsWith e String.EndsWith pesquisam uma cadeia de caracteres em busca de um texto específico. O seguinte exemplo mostra cada um desses métodos e uma variação que usa uma pesquisa que não diferencia maiúsculas de minúsculas:

C#

```
string factMessage = "Extension methods have all the capabilities of regular
static methods.";

// Write the string and include the quotation marks.
Console.WriteLine($"\"{factMessage}\"");

// Simple comparisons are always case sensitive!
bool containsSearchResult = factMessage.Contains("extension");
```

```
Console.WriteLine($"Contains \"extension\"? {containsSearchResult}");

// For user input and strings that will be displayed to the end user,
// use the StringComparison parameter on methods that have it to specify how
to match strings.
bool ignoreCaseSearchResult = factMessage.StartsWith("extension",
System.StringComparison.CurrentCultureIgnoreCase);
Console.WriteLine($"Starts with \"extension\"? {ignoreCaseSearchResult}
(ignoring case)");

bool endsWithSearchResult = factMessage.EndsWith(".",
System.StringComparison.CurrentCultureIgnoreCase);
Console.WriteLine($"Ends with '.'? {endsWithSearchResult}");
```

O exemplo anterior demonstra um ponto importante para usar esses métodos. As pesquisas de texto **diferenciam maiúsculas e minúsculas** por padrão. Use o valor de enumeração StringComparison.CurrentCultureIgnoreCase para especificar uma pesquisa que não diferencia maiúsculas de minúsculas.

Em que local de uma cadeia de caracteres o texto procurado ocorre?

Os métodos IndexOf e LastIndexOf também pesquisam texto em cadeias de caracteres. Esses métodos retornam o local do texto que está sendo procurado. Se o texto não for encontrado, elas retornarão `-1`. O exemplo a seguir mostra uma pesquisa para a primeira e a última ocorrência da palavra "métodos" e exibe o texto entre elas.

C#

```
string factMessage = "Extension methods have all the capabilities of regular
static methods.";

// Write the string and include the quotation marks.
Console.WriteLine($"\"{factMessage}\"");

// This search returns the substring between two strings, so
// the first index is moved to the character just after the first string.
int first = factMessage.IndexOf("methods") + "methods".Length;
int last = factMessage.LastIndexOf("methods");
string str2 = factMessage.Substring(first, last - first);
Console.WriteLine($"Substring between \"methods\" and \"methods\":
'{str2}'");
```

Localizar texto específico usando expressões regulares

A classe System.Text.RegularExpressions.Regex pode ser usada para pesquisar cadeias de caracteres. Essas pesquisas podem variar em complexidade, de padrões de texto simples até os complicados.

O exemplo de código a seguir procura a palavra "the" ou "their" em uma oração, sem diferenciar maiúsculas e minúsculas. O método estático Regex.IsMatch realiza a pesquisa. Você fornece a ele a cadeia de caracteres a pesquisar e um padrão de pesquisa. Nesse caso, um terceiro argumento especifica que a pesquisa não diferencia maiúsculas de minúsculas. Para obter mais informações, consulte System.Text.RegularExpressions.RegexOptions.

O padrão de pesquisa descreve o texto pelo qual procurar. A tabela a seguir descreve cada elemento desse padrão de pesquisa. (A tabela abaixo usa a única `\`, que deve ser escapada como `\\` em uma cadeia de caracteres C#).

Padrão	Significado
`the`	corresponder ao texto "the"
`(eir)?`	corresponder a 0 ou 1 ocorrência de "eir"
`\s`	corresponder a um caractere de espaço em branco

C#

```
string[] sentences =
{
    "Put the water over there.",
    "They're quite thirsty.",
    "Their water bottles broke."
};

string sPattern = "the(ir)?\\s";

foreach (string s in sentences)
{
    Console.Write($"{s,24}");

    if (System.Text.RegularExpressions.Regex.IsMatch(s, sPattern,
System.Text.RegularExpressions.RegexOptions.IgnoreCase))
    {
        Console.WriteLine($"  (match for '{sPattern}' found)");
    }
    else
    {
        Console.WriteLine();
    }
}
```

Dica

Os métodos `string` são geralmente melhores opções quando você está procurando por uma cadeia de caracteres exata. Expressões regulares são melhores quando você está procurando por algum padrão em uma cadeia de caracteres de origem.

Uma cadeia de caracteres segue um padrão?

O código a seguir usa expressões regulares para validar o formato de cada cadeia de caracteres em uma matriz. A validação requer que cada cadeia de caracteres tenha a forma de um número de telefone no qual os três grupos de dígitos são separados por traços, os dois primeiros grupos contêm três dígitos e o terceiro grupo contém quatro dígitos. O padrão de pesquisa usa a expressão regular `^\\d{3}-\\d{3}-\\d{4}$`. Para obter mais informações, consulte Linguagem de expressões regulares – referência rápida.

Padrão	Significado
`^`	corresponde ao início da cadeia de caracteres
`\d{3}`	corresponde a exatamente 3 caracteres de dígitos
`-`	corresponde ao caractere '-'
`\d{4}`	corresponde a exatamente 4 caracteres de dígitos
`$`	corresponde ao final da cadeia de caracteres

C#

```
string[] numbers =
{
    "123-555-0190",
    "444-234-22450",
    "690-555-0178",
    "146-893-232",
    "146-555-0122",
    "4007-555-0111",
    "407-555-0111",
    "407-2-5555",
    "407-555-8974",
    "407-2ab-5555",
    "690-555-8148",
    "146-893-232-"
};
```

```
string sPattern = "^\\d{3}-\\d{3}-\\d{4}$";

foreach (string s in numbers)
{
    Console.Write($"{s,14}");

    if (System.Text.RegularExpressions.Regex.IsMatch(s, sPattern))
    {
        Console.WriteLine(" - valid");
    }
    else
    {
        Console.WriteLine(" - invalid");
    }
}
```

Este padrão de pesquisa único corresponde a várias cadeias de caracteres válidas. Expressões regulares são melhores para pesquisar por ou validar mediante um padrão, em vez de uma única cadeia de caracteres de texto.

Confira também

- Guia de programação em C#
- Cadeias de caracteres
- LINQ e cadeias de caracteres
- System.Text.RegularExpressions.Regex
- Expressões regulares do .NET
- Linguagem de expressão regular – referência rápida
- Práticas recomendadas para o uso de cadeias de caracteres no .NET

Como modificar o conteúdo de uma cadeia de caracteres em C#

Artigo • 09/05/2023

Este artigo demonstra várias técnicas para produzir um `string` modificando um `string` existente. Todas as técnicas demonstradas retornam o resultado das modificações como um novo objeto `string`. Para demonstrar que as cadeias de caracteres originais e modificadas são instâncias distintas, os exemplos armazenam o resultado em uma nova variável. Você pode examinar o original `string` e o `string` novo e modificado ao executar cada exemplo.

> **Observação**
>
> Os exemplos de C# neste artigo são executados no executador de código embutido Try.NET e no playground. Clique no botão **Executar** para executar um exemplo em uma janela interativa. Ao executar o código, é possível modificá-lo e executar o código modificado clicando em **Executar** novamente. O código modificado será executado na janela interativa ou, se a compilação falhar, a janela interativa exibirá todos as mensagens de erro do compilador C#.

Há várias técnicas demonstradas neste artigo. Você pode substituir o texto existente. Você pode procurar padrões e substituir o texto correspondente com outro texto. Você pode tratar uma cadeia de caracteres como uma sequência de caracteres. Você também pode usar métodos de conveniência que removem o espaço em branco. Escolha as técnicas que mais se aproximam do seu cenário.

Substituir texto

O código a seguir cria uma nova cadeia de caracteres, substituindo o texto existente por um substituto.

C#

```
string source = "The mountains are behind the clouds today.";

// Replace one substring with another with String.Replace.
// Only exact matches are supported.
var replacement = source.Replace("mountains", "peaks");
Console.WriteLine($"The source string is <{source}>");
Console.WriteLine($"The updated string is <{replacement}>");
```

O código anterior demonstra essa propriedade *immutable* de cadeias de caracteres. Você pode ver no exemplo anterior que a cadeia de caracteres original, `source`, não é modificada. O método String.Replace cria uma nova `string` contendo as modificações.

O método Replace pode substituir cadeias de caracteres ou caracteres únicos. Em ambos os casos, todas as ocorrências do texto pesquisado são substituídas. O exemplo a seguir substitui todos os caracteres ' ' com '_':

C#

```
string source = "The mountains are behind the clouds today.";

// Replace all occurrences of one char with another.
var replacement = source.Replace(' ', '_');
Console.WriteLine(source);
Console.WriteLine(replacement);
```

A cadeia de caracteres de origem não é alterada e uma nova cadeia de caracteres é retornada com a substituição.

Cortar espaço em branco

Você pode usar os métodos String.Trim, String.TrimStart e String.TrimEnd para remover espaços em branco à esquerda ou à direita. O código a seguir mostra um exemplo de cada um desses casos. A cadeia de caracteres de origem não é alterada; esses métodos retornam uma nova cadeia de caracteres com o conteúdo modificado.

C#

```
// Remove trailing and leading white space.
string source = "    I'm wider than I need to be.      ";
// Store the results in a new string variable.
var trimmedResult = source.Trim();
var trimLeading = source.TrimStart();
var trimTrailing = source.TrimEnd();
Console.WriteLine($"<{source}>");
Console.WriteLine($"<{trimmedResult}>");
Console.WriteLine($"<{trimLeading}>");
Console.WriteLine($"<{trimTrailing}>");
```

Remover texto

Você pode remover texto de uma cadeia de caracteres usando o método String.Remove. Esse método remove um número de caracteres começando em um índice específico. O

exemplo a seguir mostra como usar String.IndexOf seguido por Remove para remover texto de uma cadeia de caracteres:

C#

```
string source = "Many mountains are behind many clouds today.";
// Remove a substring from the middle of the string.
string toRemove = "many ";
string result = string.Empty;
int i = source.IndexOf(toRemove);
if (i >= 0)
{
    result= source.Remove(i, toRemove.Length);
}
Console.WriteLine(source);
Console.WriteLine(result);
```

Substituir padrões correspondentes

Você pode usar expressões regulares para substituir padrões correspondentes de texto com um novo texto, possivelmente definido por um padrão. O exemplo a seguir usa a classe System.Text.RegularExpressions.Regex para localizar um padrão em uma cadeia de caracteres de origem e substituí-lo pela capitalização correta. O método Regex.Replace(String, String, MatchEvaluator, RegexOptions) usa uma função que fornece a lógica de substituição como um de seus argumentos. Neste exemplo, essa função `LocalReplaceMatchCase` é uma **função local** declarada dentro do método de exemplo. `LocalReplaceMatchCase` usa a classe System.Text.StringBuilder para criar a cadeia de caracteres de substituição com a capitalização correta.

Expressões regulares são mais úteis para localizar e substituir texto que segue um padrão, em vez de texto conhecido. Saiba mais em Como pesquisar cadeias de caracteres. O padrão de pesquisa "the\s" procura a palavra "the" seguida por um caractere de espaço em branco. Essa parte do padrão garante que isso não corresponda a "there" na cadeia de caracteres de origem. Para obter mais informações sobre elementos de linguagem de expressão regular, consulte Linguagem de expressão regular – referência rápida.

C#

```
string source = "The mountains are still there behind the clouds today.";

// Use Regex.Replace for more flexibility.
// Replace "the" or "The" with "many" or "Many".
// using System.Text.RegularExpressions
string replaceWith = "many ";
```

```
source = System.Text.RegularExpressions.Regex.Replace(source, "the\\s",
LocalReplaceMatchCase,
    System.Text.RegularExpressions.RegexOptions.IgnoreCase);
Console.WriteLine(source);

string LocalReplaceMatchCase(System.Text.RegularExpressions.Match
matchExpression)
{
    // Test whether the match is capitalized
    if (Char.IsUpper(matchExpression.Value[0]))
    {
        // Capitalize the replacement string
        System.Text.StringBuilder replacementBuilder = new
System.Text.StringBuilder(replaceWith);
        replacementBuilder[0] = Char.ToUpper(replacementBuilder[0]);
        return replacementBuilder.ToString();
    }
    else
    {
        return replaceWith;
    }
}
```

O método StringBuilder.ToString retorna uma cadeia de caracteres imutável com o conteúdo no objeto StringBuilder.

Modificar caracteres individuais

Você pode produzir uma matriz de caracteres de uma cadeia de caracteres, modificar o conteúdo da matriz e, em seguida, criar uma nova cadeia de caracteres com base no conteúdo modificado da matriz.

O exemplo a seguir mostra como substituir um conjunto de caracteres em uma cadeia de caracteres. Primeiro, ele usa o método String.ToCharArray() para criar uma matriz de caracteres. Ele usa o método IndexOf para encontrar o índice inicial da palavra "fox". Os próximos três caracteres são substituídos por uma palavra diferente. Por fim, uma nova cadeia de caracteres é construída com a matriz de caracteres atualizada.

C#

```
string phrase = "The quick brown fox jumps over the fence";
Console.WriteLine(phrase);

char[] phraseAsChars = phrase.ToCharArray();
int animalIndex = phrase.IndexOf("fox");
if (animalIndex != -1)
{
    phraseAsChars[animalIndex++] = 'c';
    phraseAsChars[animalIndex++] = 'a';
```

```
    phraseAsChars[animalIndex] = 't';
}

string updatedPhrase = new string(phraseAsChars);
Console.WriteLine(updatedPhrase);
```

Compilar programaticamente o conteúdo de cadeia de caracteres

Como as cadeias de caracteres são imutáveis, todos os exemplos anteriores criam cadeias de caracteres temporárias ou matrizes de caracteres. Em cenários de alto desempenho, pode ser desejável evitar essas alocações de heap. O .NET Core fornece um método String.Create que permite preencher programaticamente o conteúdo de caracteres de uma cadeia de caracteres por meio de um retorno de chamada, evitando as alocações de cadeias de caracteres temporárias intermediárias.

C#

```
// constructing a string from a char array, prefix it with some additional
characters
char[] chars = { 'a', 'b', 'c', 'd', '\0' };
int length = chars.Length + 2;
string result = string.Create(length, chars, (Span<char> strContent, char[]
charArray) =>
{
    strContent[0] = '0';
    strContent[1] = '1';
    for (int i = 0; i < charArray.Length; i++)
    {
        strContent[i + 2] = charArray[i];
    }
});

Console.WriteLine(result);
```

Você pode modificar uma cadeia de caracteres em um bloco fixo com código não seguro, mas é **altamente** desaconselhável modificar o conteúdo da cadeia de caracteres depois que uma cadeia de caracteres é criada. Fazer isso pode quebrar as coisas de maneiras imprevisíveis. Por exemplo, se alguém internalizar uma cadeia de caracteres com o mesmo conteúdo que o seu, ele receberá a sua cópia e não esperará que você esteja modificando sua cadeia de caracteres.

Confira também

- Expressões regulares do .NET
- Linguagem de expressão regular – referência rápida

Como comparar cadeias de caracteres no C#

Artigo • 20/03/2024

Você compara cadeias de caracteres para responder a uma das duas perguntas: "Essas duas cadeias de caracteres são iguais?" ou "Em que ordem essas cadeias de caracteres devem ser colocadas ao classificá-las?"

Essas duas perguntas são complicadas devido a fatores que afetam as comparações de cadeia de caracteres:

- Você pode escolher uma comparação ordinal ou linguística.
- Você pode escolher se o uso de maiúsculas faz diferença.
- Você pode escolher comparações específicas de cultura.
- As comparações linguísticas dependem da plataforma e da cultura.

Os campos de enumeração System.StringComparison representam estas opções:

- **CurrentCulture**: Compare strings usando regras de classificação sensíveis à cultura e a cultura atual.
- **CurrentCultureIgnoreCase**: Compare strings usando regras de classificação sensíveis à cultura, a cultura atual e ignorando o caso das strings que estão sendo comparadas.
- **InvariantCulture**: Compare strings usando regras de classificação sensíveis à cultura e a cultura invariável.
- **InvariantCultureIgnoreCase**: Compare strings usando regras de classificação sensíveis à cultura, a cultura invariável e ignorando o caso das strings que estão sendo comparadas.
- **Ordinal**: Compare strings usando regras de classificação ordinais (binárias).
- **OrdinalIgnoreCase**: Compare strings usando regras de classificação ordinais (binárias) e ignorando o caso das strings que estão sendo comparadas.

> **Observação**
>
> Os exemplos de C# neste artigo são executados no executador de código embutido Try.NET e no playground. Clique no botão **Executar** para executar um exemplo em uma janela interativa. Ao executar o código, é possível modificá-lo e executar o código modificado clicando em **Executar** novamente. O código modificado será executado na janela interativa ou, se a compilação falhar, a janela interativa exibirá todos as mensagens de erro do compilador C#.

Ao comparar cadeias de caracteres, você pode definir uma ordem entre elas. As comparações são usadas para classificar uma sequência de cadeias de caracteres. Uma vez que a sequência esteja em uma ordem conhecida, fica mais fácil pesquisar, tanto por software quanto por humanos. Outras comparações podem verificar se as strings são iguais. Essas verificações de semelhança são semelhantes à igualdade, mas algumas diferenças, como diferenças de maiúsculas e minúsculas, podem ser ignoradas.

Comparações ordinárias padrão

Por padrão, as operações mais comuns:

- String.Equals
- String.Equality e String.Inequality, ou seja, operadores de igualdade == e !=, respectivamente, executam uma comparação ordinal que diferencia maiúsculas de minúsculas. String.Equals tem uma sobrecarga onde um argumento StringComparison pode ser fornecido para alterar suas regras de classificação. O exemplo a seguir demonstra que:

C#

```
string root = @"C:\users";
string root2 = @"C:\Users";

bool result = root.Equals(root2);
Console.WriteLine($"Ordinal comparison: <{root}> and <{root2}> are {(result
? "equal." : "not equal.")}");

result = root.Equals(root2, StringComparison.Ordinal);
Console.WriteLine($"Ordinal comparison: <{root}> and <{root2}> are {(result
? "equal." : "not equal.")}");

Console.WriteLine($"Using == says that <{root}> and <{root2}> are {(root ==
root2 ? "equal" : "not equal")}");
```

A comparação ordinal padrão não leva em conta regras linguísticas ao comparar cadeias de caracteres. Ela compara o valor binário de cada objeto Char em duas cadeias de caracteres. Como resultado, a comparação ordinal padrão também diferencia maiúsculas de minúsculas.

Observe que o teste de igualdade com String.Equals e os operadores `==` e `!=` difere da comparação de cadeias de caracteres usando os métodos String.CompareTo e Compare(String, String). Todos eles executam uma comparação que diferencia maiúsculas de minúsculas. No entanto, enquanto os testes de igualdade executam uma comparação ordinal, os métodos `CompareTo` e `Compare` executam uma comparação

linguística com reconhecimento de cultura usando a cultura atual. Deixe clara a intenção do seu código chamando uma sobrecarga que especifica explicitamente o tipo de comparação a ser executada.

Comparações ordinais que não diferenciam maiúsculas de minúsculas

O método String.Equals(String, StringComparison) permite que você especifique um valor StringComparison de StringComparison.OrdinalIgnoreCase para especificar uma comparação que não diferencia maiúsculas de minúsculas. Há também um método estático String.Compare(String, String, StringComparison) que executa uma comparação ordinal que não diferencia maiúsculas de minúsculas se você especificar um valor StringComparison.OrdinalIgnoreCase para o argumento StringComparison. Essas comparações são mostradas no código a seguir:

C#

```
string root = @"C:\users";
string root2 = @"C:\Users";

bool result = root.Equals(root2, StringComparison.OrdinalIgnoreCase);
bool areEqual = String.Equals(root, root2,
StringComparison.OrdinalIgnoreCase);
int comparison = String.Compare(root, root2, comparisonType:
StringComparison.OrdinalIgnoreCase);

Console.WriteLine($"Ordinal ignore case: <{root}> and <{root2}> are {(result
? "equal." : "not equal.")}");
Console.WriteLine($"Ordinal static ignore case: <{root}> and <{root2}> are
{(areEqual ? "equal." : "not equal.")}");
if (comparison < 0)
    Console.WriteLine($"<{root}> is less than <{root2}>");
else if (comparison > 0)
    Console.WriteLine($"<{root}> is greater than <{root2}>");
else
    Console.WriteLine($"<{root}> and <{root2}> are equivalent in order");
```

Esses métodos usam as convenções de maiúsculas e minúsculas da cultura invariável ao realizar uma comparação ordinal sem distinção entre maiúsculas e minúsculas.

Comparações linguísticas

Muitos métodos de comparação de cadeias de caracteres (como String.StartsWith) usam regras linguísticas para a *cultura atual* por padrão para ordenar suas entradas. Essa

comparação linguística às vezes é chamada de "ordem de classificação de palavras." Quando você executa uma comparação linguística, alguns caracteres Unicode não alfanuméricos podem ter pesos especiais atribuídos. Por exemplo, o hífen "-" pode ter um peso pequeno atribuído a ele para que "co-op" e "coop" apareçam um ao lado do outro na ordem de classificação. Alguns caracteres de controle não imprimíveis podem ser ignorados. Além disso, alguns caracteres Unicode podem ser equivalentes a uma sequência de instâncias Char. O exemplo a seguir usa a frase "Eles dançam na rua" em alemão com os "ss" (U+0073 U+0073) em uma cadeia de caracteres e 'ß' (U+00DF) em outra. Linguisticamente (no Windows), "ss" é igual ao caractere 'ß' Esszet alemão nas culturas "en-US" e "de-DE".

C#

```
string first = "Sie tanzen auf der Straße.";
string second = "Sie tanzen auf der Strasse.";

Console.WriteLine($"First sentence is <{first}>");
Console.WriteLine($"Second sentence is <{second}>");

bool equal = String.Equals(first, second,
StringComparison.InvariantCulture);
Console.WriteLine($"The two strings {(equal == true ? "are" : "are not")}
equal.");
showComparison(first, second);

string word = "coop";
string words = "co-op";
string other = "cop";

showComparison(word, words);
showComparison(word, other);
showComparison(words, other);
void showComparison(string one, string two)
{
    int compareLinguistic = String.Compare(one, two,
StringComparison.InvariantCulture);
    int compareOrdinal = String.Compare(one, two, StringComparison.Ordinal);
    if (compareLinguistic < 0)
        Console.WriteLine($"<{one}> is less than <{two}> using invariant
culture");
    else if (compareLinguistic > 0)
        Console.WriteLine($"<{one}> is greater than <{two}> using invariant
culture");
    else
        Console.WriteLine($"<{one}> and <{two}> are equivalent in order
using invariant culture");
    if (compareOrdinal < 0)
        Console.WriteLine($"<{one}> is less than <{two}> using ordinal
comparison");
    else if (compareOrdinal > 0)
        Console.WriteLine($"<{one}> is greater than <{two}> using ordinal
```

```
comparison");
    else
        Console.WriteLine($"<{one}> and <{two}> are equivalent in order
using ordinal comparison");
}
```

No Windows, antes do .NET 5, a ordem de classificação de "cop", "coop" e "co-op" muda quando você muda de uma comparação linguística para uma comparação ordinal. As duas frases em alemão também são comparadas de forma diferente ao usar os tipos diferentes de comparação. Antes do .NET 5, as APIs de globalização do .NET usavam bibliotecas National Language Support (NLS). No .NET 5 e versões posteriores, as APIs de globalização do .NET usam bibliotecas ICU (Componentes Internacionais para Unicode), que unificam o comportamento de globalização da NET em todos os sistemas operacionais com suporte.

Comparações usando culturas específicas

O exemplo a seguir armazena objetos CultureInfo para as culturas en-US e de-DE. As comparações são feitas usando um objeto CultureInfo para garantir uma comparação específica da cultura. A cultura usada afeta as comparações linguísticas. O exemplo a seguir mostra os resultados da comparação das duas frases em alemão usando a cultura "en-US" e a cultura "de-DE":

C#

```
string first = "Sie tanzen auf der Straße.";
string second = "Sie tanzen auf der Strasse.";

Console.WriteLine($"First sentence is <{first}>");
Console.WriteLine($"Second sentence is <{second}>");

var en = new System.Globalization.CultureInfo("en-US");

// For culture-sensitive comparisons, use the String.Compare
// overload that takes a StringComparison value.
int i = String.Compare(first, second, en,
System.Globalization.CompareOptions.None);
Console.WriteLine($"Comparing in {en.Name} returns {i}.");

var de = new System.Globalization.CultureInfo("de-DE");
i = String.Compare(first, second, de,
System.Globalization.CompareOptions.None);
Console.WriteLine($"Comparing in {de.Name} returns {i}.");

bool b = String.Equals(first, second, StringComparison.CurrentCulture);
Console.WriteLine($"The two strings {(b ? "are" : "are not")} equal.");

string word = "coop";
```

```
string words = "co-op";
string other = "cop";

showComparison(word, words, en);
showComparison(word, other, en);
showComparison(words, other, en);
void showComparison(string one, string two, System.Globalization.CultureInfo
culture)
{
    int compareLinguistic = String.Compare(one, two, en,
System.Globalization.CompareOptions.None);
    int compareOrdinal = String.Compare(one, two, StringComparison.Ordinal);
    if (compareLinguistic < 0)
        Console.WriteLine($"<{one}> is less than <{two}> using en-US
culture");
    else if (compareLinguistic > 0)
        Console.WriteLine($"<{one}> is greater than <{two}> using en-US
culture");
    else
        Console.WriteLine($"<{one}> and <{two}> are equivalent in order
using en-US culture");
    if (compareOrdinal < 0)
        Console.WriteLine($"<{one}> is less than <{two}> using ordinal
comparison");
    else if (compareOrdinal > 0)
        Console.WriteLine($"<{one}> is greater than <{two}> using ordinal
comparison");
    else
        Console.WriteLine($"<{one}> and <{two}> are equivalent in order
using ordinal comparison");
}
```

As comparações que diferenciam cultura normalmente são usadas para comparar e classificar cadeias de caracteres inseridas por usuários com outras cadeias de caracteres inseridas por usuários. Os caracteres e as convenções de classificação dessas cadeias de caracteres podem variar de acordo com a localidade do computador do usuário. Até mesmo cadeias de caracteres que contêm caracteres idênticos podem ser classificadas de formas diferentes dependendo da cultura do thread atual.

Classificação linguística e cadeias de caracteres de pesquisa em matrizes

Os exemplos a seguir mostram como classificar e pesquisar cadeias de caracteres em uma matriz usando uma comparação linguística que depende da cultura atual. Use os métodos Array estáticos que aceitam um parâmetro System.StringComparer.

O exemplo a seguir mostra como classificar um array de strings usando a cultura atual:

C#

```
string[] lines = new string[]
{
    @"c:\public\textfile.txt",
    @"c:\public\textFile.TXT",
    @"c:\public\Text.txt",
    @"c:\public\testfile2.txt"
};

Console.WriteLine("Non-sorted order:");
foreach (string s in lines)
{
    Console.WriteLine($"   {s}");
}

Console.WriteLine("\n\rSorted order:");

// Specify Ordinal to demonstrate the different behavior.
Array.Sort(lines, StringComparer.CurrentCulture);

foreach (string s in lines)
{
    Console.WriteLine($"   {s}");
}
```

Depois que a matriz é classificada, você pode procurar as entradas usando uma pesquisa binária. Uma pesquisa binária é iniciada no meio da coleção para determinar qual metade da coleção contém a cadeia de caracteres procurada. Cada comparação subsequente subdivide a parte restante da coleção na metade. A matriz é classificada usando o StringComparer.CurrentCulture. A função local `ShowWhere` exibe informações sobre o local em que a cadeia de caracteres foi encontrada. Se a cadeia de caracteres não for encontrada, o valor retornado indicará onde ela estaria se fosse encontrada.

C#

```
string[] lines = new string[]
{
    @"c:\public\textfile.txt",
    @"c:\public\textFile.TXT",
    @"c:\public\Text.txt",
    @"c:\public\testfile2.txt"
};
Array.Sort(lines, StringComparer.CurrentCulture);

string searchString = @"c:\public\TEXTFILE.TXT";
Console.WriteLine($"Binary search for <{searchString}>");
int result = Array.BinarySearch(lines, searchString,
StringComparer.CurrentCulture);
ShowWhere<string>(lines, result);
```

```
Console.WriteLine($"{(result > 0 ? "Found" : "Did not find")}
{searchString}");

void ShowWhere<T>(T[] array, int index)
{
    if (index < 0)
    {
        index = ~index;

        Console.Write("Not found. Sorts between: ");

        if (index == 0)
            Console.Write("beginning of sequence and ");
        else
            Console.Write($"{array[index - 1]} and ");

        if (index == array.Length)
            Console.WriteLine("end of sequence.");
        else
            Console.WriteLine($"{array[index]}.");
    }
    else
    {
        Console.WriteLine($"Found at index {index}.");
    }
}
```

Classificação ordinal e pesquisa em coleções

O código a seguir usa a classe de coleção System.Collections.Generic.List<T> para armazenar cadeias de caracteres. As cadeias de caracteres são classificadas usando o método List<T>.Sort. Esse método precisa de um delegado que compara e ordena as duas cadeias de caracteres. O método String.CompareTo fornece essa função de comparação. Execute o exemplo e observe a ordem. Essa operação de classificação usa uma classificação ordinal que diferencia maiúsculas e minúsculas. Você usaria os métodos String.Compare estáticos para especificar regras de comparação diferentes.

C#

```
List<string> lines = new List<string>
{
    @"c:\public\textfile.txt",
    @"c:\public\textFile.TXT",
    @"c:\public\Text.txt",
    @"c:\public\testfile2.txt"
};

Console.WriteLine("Non-sorted order:");
foreach (string s in lines)
```

```
{
    Console.WriteLine($"   {s}");
}

Console.WriteLine("\n\rSorted order:");

lines.Sort((left, right) => left.CompareTo(right));
foreach (string s in lines)
{
    Console.WriteLine($"   {s}");
}
```

Uma vez classificada, a lista de cadeias de caracteres pode ser pesquisada usando uma pesquisa binária. O exemplo a seguir mostra como pesquisar a lista classificada usando a mesma função de comparação. A função local `ShowWhere` mostra o local em que o texto procurado está ou deveria estar:

C#

```
List<string> lines = new List<string>
{
    @"c:\public\textfile.txt",
    @"c:\public\textFile.TXT",
    @"c:\public\Text.txt",
    @"c:\public\testfile2.txt"
};
lines.Sort((left, right) => left.CompareTo(right));

string searchString = @"c:\public\TEXTFILE.TXT";
Console.WriteLine($"Binary search for <{searchString}>");
int result = lines.BinarySearch(searchString);
ShowWhere<string>(lines, result);

Console.WriteLine($"{(result > 0 ? "Found" : "Did not find")}
{searchString}");

void ShowWhere<T>(IList<T> collection, int index)
{
    if (index < 0)
    {
        index = ~index;

        Console.Write("Not found. Sorts between: ");

        if (index == 0)
            Console.Write("beginning of sequence and ");
        else
            Console.Write($"{collection[index - 1]} and ");

        if (index == collection.Count)
            Console.WriteLine("end of sequence.");
        else
```

```
            Console.WriteLine($"{collection[index]}.");
    }
    else
    {
        Console.WriteLine($"Found at index {index}.");
    }
}
```

Use sempre o mesmo tipo de comparação para classificação e pesquisa. O uso de tipos diferentes de comparação para classificação e pesquisa produz resultados inesperados.

Classes de coleção como System.Collections.Hashtable, System.Collections.Generic.Dictionary<TKey,TValue> e System.Collections.Generic.List<T> têm construtores que usam um parâmetro System.StringComparer quando o tipo dos elementos ou chaves é `string`. Em geral, você deve usar esses construtores sempre que possível e especificar StringComparer.Ordinal ou StringComparer.OrdinalIgnoreCase.

Confira também

- System.Globalization.CultureInfo
- System.StringComparer
- Cadeias de caracteres
- Comparando cadeias de caracteres
- Globalizando e localizando aplicativos

Colaborar conosco no GitHub

A fonte deste conteúdo pode ser encontrada no GitHub, onde você também pode criar e revisar problemas e solicitações de pull. Para obter mais informações, confira o nosso guia para colaboradores.

.NET

Comentários do .NET

O .NET é um projeto código aberto. Selecione um link para fornecer comentários:

Abrir um problema de documentação

Fornecer comentários sobre o produto

Como capturar uma exceção não compatível com CLS

Artigo • 10/05/2023

Algumas linguagens .NET, incluindo o C++/CLI, permite que os objetos lancem exceções que não derivam de Exception. Essas exceções são chamadas de *exceções não CLS* ou *não exceções*. Em C#, não é possível gerar exceções que não sejam do CLS, mas você pode capturá-las de duas maneiras:

- Em um bloco `catch (RuntimeWrappedException e)`.

  Por padrão, um assembly do Visual C# captura exceções não CLS como exceções encapsuladas. Use este método se você precisar de acesso à exceção original, que pode ser acessada por meio da propriedade RuntimeWrappedException.WrappedException. O procedimento, mais adiante neste tópico, explica como capturar exceções dessa maneira.

- Em um bloco de captura geral (um bloco de captura sem um tipo de exceção especificado), que é colocado após todos os outros blocos `catch`.

  Use esse método quando desejar realizar alguma ação (como gravar em um arquivo de log) em resposta a exceções não CLS e você não precisa de acesso às informações de exceção. Por padrão, o Common Language Runtime encapsula todas as exceções. Para desabilitar esse comportamento, adicione esse atributo de nível de assembly em seu código, geralmente no arquivo AssemblyInfo.cs: `[assembly: RuntimeCompatibilityAttribute(WrapNonExceptionThrows = false)]`.

Para capturar uma exceção não CLS

Em um bloco `catch(RuntimeWrappedException e)`, acesse a exceção original por meio da propriedade RuntimeWrappedException.WrappedException.

Exemplo

O exemplo a seguir mostra como capturar uma exceção que não é do CLS, que foi gerada por uma biblioteca de classes escrita em C++/CLI. Observe que, neste exemplo, o código cliente C# sabe com antecedência que o tipo da exceção que está sendo gerada é um System.String. Você pode converter a propriedade RuntimeWrappedException.WrappedException de volta a seu tipo original, desde que o tipo seja acessível por meio do código.

C#

```
// Class library written in C++/CLI.
var myClass = new ThrowNonCLS.Class1();

try
{
    // throws gcnew System::String(
    // "I do not derive from System.Exception!");
    myClass.TestThrow();
}
catch (RuntimeWrappedException e)
{
    String s = e.WrappedException as String;
    if (s != null)
    {
        Console.WriteLine(s);
    }
}
```

Confira também

- RuntimeWrappedException
- Exceções e manipulação de exceções

Atributos

19/06/2025

Os atributos fornecem uma maneira poderosa de associar metadados ou informações declarativas ao código (assemblies, tipos, métodos, propriedades e assim por diante). Depois de associar um atributo a uma entidade de programa, você pode consultar o atributo em tempo de execução usando uma técnica chamada *reflexão*.

Os atributos têm as seguintes propriedades:

- Os atributos adicionam metadados ao seu programa. *Metadado* é informação sobre os tipos definidos em um programa. Todos os assemblies .NET contêm um conjunto de metadados especificado que descreve os tipos e os membros de tipo definidos no assembly. Você pode adicionar atributos personalizados para especificar quaisquer outras informações necessárias.
- Atributos podem ser aplicados a assemblies inteiros, módulos ou elementos de programa menores, como classes e propriedades.
- Os atributos podem aceitar argumentos da mesma forma que métodos e propriedades.
- Os atributos permitem que um programa examine seus próprios metadados ou metadados em outros programas usando reflexão.

Trabalhar com reflexão

APIs de Reflexão fornecidas por Type descrevem assemblies, módulos e tipos. Você pode usar a reflexão para criar dinamicamente uma instância de um tipo, associar o tipo a um objeto existente ou obter o tipo de um objeto existente e invocar seus métodos ou acessar seus campos e propriedades. Quando você usa atributos em seu código, a reflexão permite acessá-los. Para obter mais informações, consulte Attributes.

Aqui está um exemplo simples de reflexão com o GetType() método. Todos os tipos da `Object` classe base herdam esse método, que é usado para obter o tipo de uma variável:

> **Observação**
>
> Certifique-se de adicionar as instruções `using System;` e `using System.Reflection;` na parte superior do seu arquivo de código C# (*.cs*).

C#

```
// Using GetType to obtain type information:
int i = 42;
```

```
Type type = i.GetType();
Console.WriteLine(type);
```

A saída mostra o tipo:

Saída

```
System.Int32
```

O exemplo a seguir usa a reflexão para obter o nome completo do assembly carregado.

C#

```
// Using Reflection to get information of an Assembly:
Assembly info = typeof(int).Assembly;
Console.WriteLine(info);
```

A saída deverá ser semelhante ao seguinte exemplo:

Saída

```
System.Private.CoreLib, Version=7.0.0.0, Culture=neutral,
PublicKeyToken=7cec85d7bea7798e
```

Diferenças de palavra-chave para IL

As palavras-chave `protected` C# e `internal` não têm nenhum significado em IL (Linguagem Intermediária) e não são usadas nas APIs de reflexão. Os termos correspondentes na IL são *Família* e *Assembly*. Aqui estão algumas maneiras pelas quais você pode usar esses termos:

- Para identificar um `internal` método usando reflexão, use a IsAssembly propriedade.
- Para identificar um método `protected internal`, use o IsFamilyOrAssembly.

Trabalhar com atributos

Os atributos podem ser colocados em quase qualquer declaração, embora um atributo específico possa restringir os tipos de declarações nas quais ele é válido. Em C#, especifique um atributo colocando o nome do atributo entre colchetes (`[]`) acima da declaração da entidade à qual ela se aplica.

Neste exemplo, você usa o SerializableAttribute atributo para aplicar uma característica específica a uma classe:

```C#
[Serializable]
public class SampleClass
{
    // Objects of this type can be serialized.
}
```

Você pode declarar um método com o DllImportAttribute atributo:

```C#
[System.Runtime.InteropServices.DllImport("user32.dll")]
extern static void SampleMethod();
```

Você pode colocar vários atributos em uma declaração:

```C#
void MethodA([In][Out] ref double x) { }
void MethodB([Out][In] ref double x) { }
void MethodC([In, Out] ref double x) { }
```

Alguns atributos podem ser especificados mais de uma vez para uma determinada entidade. O exemplo a seguir mostra a multiutilização do ConditionalAttribute atributo:

```C#
[Conditional("DEBUG"), Conditional("TEST1")]
void TraceMethod()
{
    // ...
}
```

> **Observação**
>
> Por convenção, todos os nomes de atributo terminam com o sufixo "Attribute" para distingui-los de outros tipos nas bibliotecas do .NET. No entanto, você não precisa especificar o sufixo de atributo ao usar atributos no código. Por exemplo, uma `[DllImport]` declaração é equivalente a uma `[DllImportAttribute]` declaração, mas `DllImportAttribute` é o nome real da classe na Biblioteca de Classes do .NET.

Parâmetros de atributo

Muitos atributos têm parâmetros, que podem ser *posicionais*, *sem nome* ou *nomeados*. A tabela a seguir descreve como trabalhar com atributos nomeados e posicionais:

Parâmetros posicionais

Parâmetros do construtor de atributo:

Parâmetros nomeados

Propriedades ou campos do atributo:

- Deve especificar, não pode omitir
- Sempre especifique primeiro
- Especificar em **determinada** ordem
- Sempre opcional, omita quando falso
- Especifique após os parâmetros posicionais
- Especificar em qualquer ordem

Por exemplo, o código a seguir mostra três atributos equivalentes `DllImport` :

C#

```
[DllImport("user32.dll")]
[DllImport("user32.dll", SetLastError=false, ExactSpelling=false)]
[DllImport("user32.dll", ExactSpelling=false, SetLastError=false)]
```

O primeiro parâmetro, o nome da DLL, é posicional e sempre vem primeiro. As outras instâncias são parâmetros nomeados. Nesse cenário, ambos os parâmetros nomeados têm o valor padrão de false, então podem ser omitidos. Consulte a documentação do atributo individual para obter informações sobre valores de parâmetro padrão. Para obter mais informações sobre tipos de parâmetro permitidos, consulte a seção atributos da especificação da linguagem C# .

Destinos do atributo

O *destino* de um atributo é a entidade à qual o atributo se aplica. Por exemplo, um atributo pode se aplicar a uma classe, um método ou um assembly. Por padrão, um atributo se aplica ao elemento que o segue. Mas você também pode identificar explicitamente o elemento a ser associado, como um método, um parâmetro ou o valor retornado.

Para identificar explicitamente um destino de atributo, use a seguinte sintaxe:

C#

```
[target : attribute-list]
```

A tabela a seguir mostra a lista de valores possíveis `target` .

Expandir a tabela

Valor-alvo	Aplica-se a
`assembly`	Montagem completa
`module`	Módulo de montagem atual
`field`	Campo em uma classe ou estrutura
`event`	Acontecimento
`method`	Método ou acessadores de propriedade `get` e `set`
`param`	Parâmetros de método ou parâmetros de acessador de propriedade `set`
`property`	Propriedade
`return`	Valor retornado de um método, indexador de propriedade ou acessador de propriedade `get`
`type`	Struct, classe, interface, enum ou delegado

Você pode especificar o `field` valor de destino para aplicar um atributo ao campo de backup criado para uma propriedade implementada automaticamente.

O exemplo a seguir mostra como aplicar atributos a assemblies e módulos. Para obter mais informações, consulte Atributos comuns (C#).

C#

```
using System;
using System.Reflection;
[assembly: AssemblyTitleAttribute("Production assembly 4")]
[module: CLSCompliant(true)]
```

O exemplo a seguir mostra como aplicar atributos a métodos, parâmetros de método e valores de retorno de método em C#.

C#

```
// default: applies to method
[ValidatedContract]
int Method1() { return 0; }

// applies to method
[method: ValidatedContract]
int Method2() { return 0; }

// applies to parameter
int Method3([ValidatedContract] string contract) { return 0; }

// applies to return value
[return: ValidatedContract]
int Method4() { return 0; }
```

> ⓘ **Observação**
>
> Independentemente dos destinos nos quais o `ValidatedContract` atributo é definido como válido, o `return` destino deve ser especificado, mesmo que o `ValidatedContract` atributo seja definido para aplicar somente aos valores retornados. Em outras palavras, o compilador não usa as `AttributeUsage` informações para resolver destinos de atributo ambíguos. Para obter mais informações, consulte **AttributeUsage**.

Examinar maneiras de usar atributos

Aqui estão algumas maneiras comuns de usar atributos no código:

- Marque os métodos do controlador que respondem às mensagens POST usando o `HttpPost` atributo. Para obter mais informações, consulte a classe HttpPostAttribute.
- Descreva como realizar marshaling de parâmetros de método ao interoperar com código nativo. Para obter mais informações, consulte a classe MarshalAsAttribute.
- Descreva as propriedades COM (Component Object Model) para classes, métodos e interfaces.
- Chame o código não gerenciado usando a DllImportAttribute classe.
- Descreva o assembly em termos de versão, título, descrição ou marca.
- Descreva quais membros de uma classe devem ser serializados para persistência.
- Descreva como fazer mapeamento entre nós XML e membros de classe para serialização de XML.
- Descreva os requisitos de segurança para métodos.
- Especifique as características usadas para impor a segurança.
- Controle otimizações com compilador JIT (Just-In-Time) para que o código permaneça fácil de depurar.

- Obtenha informações sobre o chamador de um método.

Examinar cenários de reflexão

A reflexão é útil nos seguintes cenários:

- Acessar atributos nos metadados do programa. Para obter mais informações, consulte Recuperar informações armazenadas em atributos.
- Examine e instancie tipos em um assembly.
- Crie novos tipos em tempo de execução usando classes no System.Reflection.Emit namespace.
- Execute métodos de acesso e vinculação tardia em tipos criados em tempo de execução. Para obter mais informações, consulte Carregar e usar tipos dinamicamente.

Links relacionados

- Atributos comuns (C#)
- Informações do chamador (C#)
- Atributos
- Reflexão
- Exibir informações de tipo
- Reflexão e tipos genéricos
- System.Reflection.Emit
- Recuperando informações armazenadas em atributos

Criar atributos personalizados

19/06/2025

Você pode criar seus próprios atributos personalizados definindo uma classe de atributo, uma classe que deriva direta ou indiretamente de Attribute. Isso torna a identificação de definições de atributos em metadados rápida e fácil. Suponha que você queira marcar tipos com o nome do programador que escreveu o tipo. Você pode definir uma classe de atributo personalizada `Author` :

C#

```
[System.AttributeUsage(System.AttributeTargets.Class |
                       System.AttributeTargets.Struct)
]
public class AuthorAttribute : System.Attribute
{
    private string Name;
    public double Version;

    public AuthorAttribute(string name)
    {
        Name = name;
        Version = 1.0;
    }
}
```

O nome `AuthorAttribute` da classe é o nome do atributo, `Author` mais o `Attribute` sufixo. Ela é derivada de `System.Attribute`, portanto, é uma classe de atributo personalizada. Os parâmetros do construtor são os parâmetros posicionais do atributo personalizado. Neste exemplo, `name` é um parâmetro posicional. Quaisquer propriedades ou campos públicos de leitura/gravação são chamados de parâmetros. Nesse caso, `version` é o único parâmetro nomeado. Observe o uso do `AttributeUsage` atributo para tornar o `Author` atributo válido somente em classes e `struct` declarações.

Você pode usar esse novo atributo da seguinte maneira:

C#

```
[Author("P. Ackerman", Version = 1.1)]
class SampleClass
{
    // P. Ackerman's code goes here...
}
```

`AttributeUsage` tem um parâmetro nomeado, `AllowMultiple` com o qual você pode fazer um atributo personalizado de uso único ou multiutilização. No exemplo de código a seguir, um atributo multiuso é criado.

C#

```
[System.AttributeUsage(System.AttributeTargets.Class |
                       System.AttributeTargets.Struct,
                       AllowMultiple = true)  // Multiuse attribute.
]
public class AuthorAttribute : System.Attribute
{
    string Name;
    public double Version;

    public AuthorAttribute(string name)
    {
        Name = name;

        // Default value.
        Version = 1.0;
    }

    public string GetName() => Name;
}
```

No exemplo de código a seguir, vários atributos do mesmo tipo são aplicados a uma classe.

C#

```
[Author("P. Ackerman"), Author("R. Koch", Version = 2.0)]
public class ThirdClass
{
    // ...
}
```

Consulte também

- System.Reflection
- Escrevendo atributos personalizados
- AttributeUsage (C#)

Acessar atributos usando reflexão

17/06/2025

O fato de que você pode definir atributos personalizados e colocá-los em seu código-fonte seria de pouco valor sem alguma maneira de recuperar essas informações e agir sobre elas. Usando reflexão, você pode recuperar as informações que foram definidas com atributos personalizados. O método de chave é `GetCustomAttributes`, que retorna uma matriz de objetos que são os equivalentes em tempo de execução dos atributos de código-fonte. Esse método tem muitas versões sobrecarregadas. Para obter mais informações, consulte Attribute.

Uma especificação de atributo como:

C#

```
[Author("P. Ackerman", Version = 1.1)]
class SampleClass { }
```

é conceitualmente equivalente ao seguinte código:

C#

```
var anonymousAuthorObject = new Author("P. Ackerman")
{
    Version = 1.1
};
```

No entanto, o código não é executado até que `SampleClass` seja consultado quanto a atributos. Chamar `GetCustomAttributes` em `SampleClass` faz com que um objeto `Author` seja construído e inicializado. Se a classe tiver outros atributos, outros objetos de atributo serão construídos da mesma forma. `GetCustomAttributes` em seguida, retorna o `Author` objeto e quaisquer outros objetos de atributo em uma matriz. Em seguida, você pode iterar sobre essa matriz, determinar quais atributos foram aplicados com base no tipo de cada elemento de matriz e extrair informações dos objetos de atributo.

Aqui está um exemplo completo. Um atributo personalizado é definido, aplicado a várias entidades e recuperado por meio de reflexão.

C#

```
// Multiuse attribute.
[System.AttributeUsage(System.AttributeTargets.Class |
                       System.AttributeTargets.Struct,
                       AllowMultiple = true)  // Multiuse attribute.
]
```

```
public class AuthorAttribute : System.Attribute
{
    string Name;
    public double Version;

    public AuthorAttribute(string name)
    {
        Name = name;

        // Default value.
        Version = 1.0;
    }

    public string GetName() => Name;
}

// Class with the Author attribute.
[Author("P. Ackerman")]
public class FirstClass
{
    // ...
}

// Class without the Author attribute.
public class SecondClass
{
    // ...
}

// Class with multiple Author attributes.
[Author("P. Ackerman"), Author("R. Koch", Version = 2.0)]
public class ThirdClass
{
    // ...
}

class TestAuthorAttribute
{
    public static void Test()
    {
        PrintAuthorInfo(typeof(FirstClass));
        PrintAuthorInfo(typeof(SecondClass));
        PrintAuthorInfo(typeof(ThirdClass));
    }

    private static void PrintAuthorInfo(System.Type t)
    {
        System.Console.WriteLine($"Author information for {t}");

        // Using reflection.
        System.Attribute[] attrs = System.Attribute.GetCustomAttributes(t);  //
Reflection.

        // Displaying output.
        foreach (System.Attribute attr in attrs)
```

```
        {
            if (attr is AuthorAttribute a)
            {
                System.Console.WriteLine($"   {a.GetName()}, version
{a.Version:f}");
            }
        }
    }
}
/* Output:
    Author information for FirstClass
       P. Ackerman, version 1.00
    Author information for SecondClass
    Author information for ThirdClass
       R. Koch, version 2.00
       P. Ackerman, version 1.00
*/
```

Consulte também

- System.Reflection
- Attribute
- Recuperando informações armazenadas em atributos

Como criar uma união do C/C++ usando atributos em C#

Artigo • 03/04/2023

Usando atributos, você pode personalizar como os structs são dispostos na memória. Por exemplo, você pode criar o que é conhecido como uma união no C/C++ usando os atributos `StructLayout(LayoutKind.Explicit)` e `FieldOffset`.

Neste segmento de código, todos os campos de `TestUnion` são iniciados no mesmo local na memória.

C#

```
[System.Runtime.InteropServices.StructLayout(LayoutKind.Explicit)]
struct TestUnion
{
    [System.Runtime.InteropServices.FieldOffset(0)]
    public int i;

    [System.Runtime.InteropServices.FieldOffset(0)]
    public double d;

    [System.Runtime.InteropServices.FieldOffset(0)]
    public char c;

    [System.Runtime.InteropServices.FieldOffset(0)]
    public byte b;
}
```

O código a seguir é outro exemplo em que os campos são iniciados em locais diferentes definidos explicitamente.

C#

```
[System.Runtime.InteropServices.StructLayout(LayoutKind.Explicit)]
struct TestExplicit
{
    [System.Runtime.InteropServices.FieldOffset(0)]
    public long lg;

    [System.Runtime.InteropServices.FieldOffset(0)]
    public int i1;

    [System.Runtime.InteropServices.FieldOffset(4)]
    public int i2;

    [System.Runtime.InteropServices.FieldOffset(8)]
```

```
    public double d;

    [System.Runtime.InteropServices.FieldOffset(12)]
    public char c;

    [System.Runtime.InteropServices.FieldOffset(14)]
    public byte b;
}
```

Os dois campos inteiros, `i1` e `i2` combinados, compartilham os mesmos locais de memória que `lg`. `lg` usa os primeiros oito bytes, ou então `i1` usa os quatro primeiros bytes e `i2` usa os próximos quatro bytes. Esse tipo de controle sobre o layout do struct é útil ao usar a invocação de plataforma.

Confira também

- System.Reflection
- Attribute
- Atributos

Collaborate with us on GitHub

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

.NET

.NET feedback

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

Open a documentation issue

Provide product feedback

Genéricos e atributos

Artigo • 21/11/2024

Atributos podem ser aplicados a tipos genéricos da mesma forma que a tipos não genéricos. No entanto, você pode aplicar atributos somente em *tipos genéricos abertos* e *tipos genéricos construídos fechados*, não em *tipos genéricos parcialmente construídos*. Um *tipo genérico aberto* é aquele em que nenhum dos argumentos de tipo é especificado, como `Dictionary<TKey, TValue>` Um *tipo genérico construído fechado* especifica todos os argumentos de tipo, como `Dictionary<string, object>`. Um *tipo genérico parcialmente construído* especifica alguns argumentos de tipo, mas não todos. Um exemplo é `Dictionary<string, TValue>`. Um *tipo* genérico não associado é aquele em que os argumentos de tipo são omitidos, como `Dictionary<,>`.

Os exemplos a seguir usam esse atributo personalizado:

C#

```
class CustomAttribute : Attribute
{
    public object? info;
}
```

Um atributo pode fazer referência a um tipo genérico não associado:

C#

```
public class GenericClass1<T> { }

[CustomAttribute(info = typeof(GenericClass1<>))]
class ClassA { }
```

Especifica vários parâmetros de tipo usando a quantidade apropriada de vírgulas. Neste exemplo, `GenericClass2` tem dois parâmetros de tipo:

C#

```
public class GenericClass2<T, U> { }

[CustomAttribute(info = typeof(GenericClass2<,>))]
class ClassB { }
```

Um atributo pode referenciar um tipo genérico construído fechado:

C#

```
public class GenericClass3<T, U, V> { }

[CustomAttribute(info = typeof(GenericClass3<int, double, string>))]
class ClassC { }
```

Um atributo que referencia um parâmetro de tipo genérico causa um erro em tempo de compilação:

C#

```
[CustomAttribute(info = typeof(GenericClass3<int, T, string>))]  //Error
CS0416
class ClassD<T> { }
```

A partir do C# 11, um tipo genérico pode herdar de Attribute:

C#

```
public class CustomGenericAttribute<T> : Attribute { }  //Requires C# 11
```

Para obter informações sobre um tipo genérico ou um parâmetro de tipo em tempo de execução, é possível usar os métodos do System.Reflection. Para obter mais informações, confira Genéricos e Reflexão.

Confira também

- Genéricos
- Atributos

Como consultar metadados de um assembly com reflexão (LINQ)

Artigo • 02/04/2025

Use as APIs de reflexão do .NET para examinar os metadados em um assembly do .NET e criar coleções de tipos, membros de tipo e parâmetros que estão nesse assembly. Como essas coleções dão suporte à interface genérica IEnumerable<T> , elas podem ser consultadas usando LINQ.

O exemplo a seguir mostra como o LINQ pode ser usado com reflexão para recuperar metadados específicos sobre métodos que correspondem a um critério de pesquisa especificado. Nesse caso, a consulta localiza os nomes de todos os métodos no assembly que retornam tipos enumeráveis, como matrizes.

C#

```
Assembly assembly = Assembly.Load("System.Private.CoreLib, Version=7.0.0.0,
Culture=neutral, PublicKeyToken=7cec85d7bea7798e");
var pubTypesQuery = from type in assembly.GetTypes()
                    where type.IsPublic
                    from method in type.GetMethods()
                    where method.ReturnType.IsArray == true
                        || (method.ReturnType.GetInterface(

typeof(System.Collections.Generic.IEnumerable<>).FullName!) != null
                        && method.ReturnType.FullName != "System.String")
                    group method.ToString() by type.ToString();

foreach (var groupOfMethods in pubTypesQuery)
{
    Console.WriteLine($"Type: {groupOfMethods.Key}");
    foreach (var method in groupOfMethods)
    {
        Console.WriteLine($"  {method}");
    }
}
```

O exemplo usa o Assembly.GetTypes método para retornar uma matriz de tipos no assembly especificado. O filtro onde é aplicado para que somente tipos públicos sejam retornados. Para cada tipo público, uma subconsulta é gerada usando a matriz MethodInfo que é retornada da chamada Type.GetMethods. Esses resultados são filtrados para retornar apenas os métodos cujo tipo de retorno é uma matriz ou então um tipo que implementa IEnumerable<T>. Por fim, esses resultados são agrupados usando o nome do tipo como uma chave.

Genéricos e reflexão

19/06/2025

Como o CLR (Common Language Runtime) tem acesso a informações de tipo genérico em tempo de execução, você pode usar a reflexão para obter informações sobre tipos genéricos da mesma forma que para tipos não genéricos. Para obter mais informações, consulte Genéricos no Runtime.

O System.Reflection.Emit namespace também contém novos membros que dão suporte a genéricos. Veja como definir um tipo genérico com a emissão de reflexão.

Para obter uma lista das condições invariáveis para termos usados na reflexão genérica, consulte os comentários da propriedade IsGenericType:

- IsGenericType: retornará true se um tipo for genérico.
- GetGenericArguments: retorna uma matriz de `Type` objetos que representam os argumentos de tipo fornecidos para um tipo construído ou os parâmetros de tipo de uma definição de tipo genérico.
- GetGenericTypeDefinition: retorna a definição de tipo genérico subjacente para o tipo construído atual.
- GetGenericParameterConstraints: retorna uma matriz de `Type` objetos que representam as restrições no parâmetro de tipo genérico atual.
- ContainsGenericParameters: retorna verdadeiro se o tipo ou qualquer um dos seus tipos ou métodos de delimitação contêm parâmetros de tipo para os quais não foram fornecidos tipos específicos.
- GenericParameterAttributes: obtém uma combinação de sinalizadores `GenericParameterAttributes` que descrevem as restrições especiais do parâmetro de tipo genérico atual.
- GenericParameterPositionPara um objeto `Type` que representa um parâmetro de tipo, obtém a posição do parâmetro de tipo na lista de parâmetros de tipo da definição de tipo genérico ou da definição de método genérico que declarou o parâmetro de tipo.
- IsGenericParameter: obtém um valor que indica se a corrente `Type` representa um parâmetro de tipo de um tipo genérico ou definição de método.
- IsGenericTypeDefinition: obtém um valor que indica se a atual Type representa uma definição de tipo genérico, da qual outros tipos genéricos podem ser construídos. Retornará true se o tipo representar a definição de um tipo genérico.
- DeclaringMethod: retorna o método genérico que definiu o parâmetro de tipo genérico atual ou nulo se o parâmetro de tipo não foi definido por um método genérico.
- MakeGenericType: substitui os elementos de uma matriz de tipos para os parâmetros de tipo da definição de tipo genérico atual e retorna um Type objeto que representa o tipo construído resultante.

Além disso, os membros da MethodInfo classe habilitam informações de tempo de execução para métodos genéricos. Consulte as IsGenericMethod observações de propriedade para obter uma lista de condições invariáveis para termos usados para refletir sobre métodos genéricos:

- IsGenericMethod: Retorna verdadeiro se um método for genérico.
- GetGenericArguments: retorna uma matriz de objetos Type que representam os argumentos de tipo de um método genérico construído ou os parâmetros de tipo de uma definição de método genérico.
- GetGenericMethodDefinition: retorna a definição de método genérico subjacente para o método construído atual.
- ContainsGenericParameters: retorna true se o método ou qualquer um dos seus tipos de delimitação contêm parâmetros de tipo para os quais não foram fornecidos tipos específicos.
- IsGenericMethodDefinition: retorna true se a corrente MethodInfo representa a definição de um método genérico.
- MakeGenericMethod: substitui os elementos de uma matriz de tipos para os parâmetros de tipo da definição de método genérico atual e retorna um MethodInfo objeto que representa o método construído resultante.

Consulte também

- Genéricos
- Reflexão e tipos genéricos
- Genéricos

Definir e ler atributos personalizados

19/06/2025

Os atributos fornecem uma maneira de associar informações ao código de forma declarativa. Eles também podem fornecer um elemento reutilizável que pode ser aplicado a vários destinos. Considere o ObsoleteAttribute. Ele pode ser aplicado a classes, structs, métodos, construtores e muito mais. Ele *declara* que o elemento é obsoleto. Em seguida, cabe ao compilador C# procurar esse atributo e executar alguma ação em resposta.

Neste tutorial, você aprenderá a adicionar atributos ao seu código, como criar e usar seus próprios atributos e como usar alguns atributos integrados ao .NET.

Pré-requisitos

Você precisa configurar seu computador para executar o .NET. Você pode encontrar as instruções de instalação na página Downloads do .NET . Você pode executar esse aplicativo no Windows, Ubuntu Linux, macOS ou em um contêiner do Docker. Você precisa instalar seu editor de código favorito. As descrições a seguir usam o Visual Studio Code, que é um editor multiplataforma de software livre. No entanto, você pode usar as ferramentas com as quais se sente confortável.

Criar o aplicativo

Agora que você instalou todas as ferramentas, crie um novo aplicativo de console do .NET. Para usar o gerador de linha de comando, execute o seguinte comando no shell favorito:

CLI do .NET

```
dotnet new console
```

Esse comando cria arquivos de projeto .NET básicos. Você executa `dotnet restore` para restaurar as dependências necessárias para compilar este projeto.

Não é necessário executar dotnet restore, pois ele é executado implicitamente por todos os comandos que exigem uma restauração, como `dotnet new`, `dotnet build`, `dotnet run`, `dotnet test`, `dotnet publish` e `dotnet pack`. Para desabilitar a restauração implícita, use a opção `--no-restore`.

O comando `dotnet restore` ainda é útil em determinados cenários em que realizar uma restauração explícita faz sentido, como compilações de integração contínua no Azure DevOps

Services ou em sistemas de compilação que precisam controlar explicitamente quando a restauração ocorrerá.

Para obter informações sobre como gerenciar feeds do NuGet, confira a documentação do dotnet restore.

Para executar o programa, use `dotnet run`. Você deve ver a saída do "Olá, Mundo" no console.

Adicionar atributos ao código

Em C#, os atributos são classes que herdam da `Attribute` classe base. Qualquer classe que herda de `Attribute` pode ser usada como uma espécie de "marca" em outras partes do código. Por exemplo, há um atributo chamado `ObsoleteAttribute`. Esse atributo sinaliza que o código está obsoleto e não deve mais ser usado. Você coloca esse atributo em uma classe, por exemplo, usando colchetes.

C#

```
[Obsolete]
public class MyClass
{
}
```

Embora a classe seja chamada `ObsoleteAttribute`, é necessário usar `[Obsolete]` apenas no código. A maioria dos códigos C# segue esta convenção. Você pode usar o nome `[ObsoleteAttribute]` completo se quiser.

Ao marcar uma classe obsoleta, é uma boa ideia fornecer algumas informações sobre *por que* ela é obsoleta e/ou *o que* usar em vez disso. Você inclui um parâmetro de cadeia de caracteres para o atributo Obsoleto para fornecer essa explicação.

C#

```
[Obsolete("ThisClass is obsolete. Use ThisClass2 instead.")]
public class ThisClass
{
}
```

A cadeia de caracteres está sendo passada como um argumento para um `ObsoleteAttribute` construtor, como se você estivesse escrevendo `var attr = new ObsoleteAttribute("some string")`.

Os parâmetros para um construtor de atributo são limitados a tipos/literais simples: `bool, int, double, string, Type, enums, etc` e matrizes desses tipos. Você não pode usar uma expressão ou uma variável. Você é livre para usar parâmetros posicionais ou nomeados.

Criar seu próprio atributo

Você cria um atributo definindo uma nova classe que herda da `Attribute` classe base.

C#

```
public class MySpecialAttribute : Attribute
{
}
```

Com o código anterior, você pode usar `[MySpecial]` (ou `[MySpecialAttribute]`) como um atributo em outro lugar na base de código.

C#

```
[MySpecial]
public class SomeOtherClass
{
}
```

Atributos na biblioteca de classes base do .NET, como `ObsoleteAttribute`, disparam determinados comportamentos dentro do compilador. No entanto, qualquer atributo criado atua apenas como metadados e não resulta na execução de nenhum código dentro da classe de atributo. Cabe a você agir sobre esses metadados em outro lugar em seu código.

Há uma pegadinha aqui. Conforme mencionado anteriormente, somente determinados tipos podem ser passados como argumentos ao usar atributos. No entanto, ao criar um tipo de atributo, o compilador C# não impede que você crie esses parâmetros. No exemplo a seguir, você criou um atributo com um construtor que é compilado corretamente.

C#

```
public class GotchaAttribute : Attribute
{
    public GotchaAttribute(Foo myClass, string str)
    {
    }
}
```

No entanto, não é possível usar esse construtor com sintaxe de atributo.

C#

```
[Gotcha(new Foo(), "test")] // does not compile
public class AttributeFail
{
}
```

O código anterior causa um erro de compilador como `Attribute constructor parameter 'myClass' has type 'Foo', which is not a valid attribute parameter type`

Como restringir o uso de atributos

Os atributos podem ser usados nos seguintes "destinos". Os exemplos anteriores os mostram em classes, mas também podem ser usados em:

- Assembléia
- Classe
- Construtor
- Delegar
- Enumeração
- Acontecimento
- Campo
- GenericParameter
- Interfase
- Método
- Módulo
- Parâmetro
- Propriedade
- ReturnValue
- Estrutura

Quando você cria uma classe de atributo, por padrão, o C# permite que você use esse atributo em qualquer um dos possíveis destinos de atributo. Se você quiser restringir seu atributo a determinados destinos, poderá fazer isso usando a `AttributeUsageAttribute` classe de atributo. Isso mesmo, um atributo em cima de um atributo!

C#

```
[AttributeUsage(AttributeTargets.Class | AttributeTargets.Struct)]
public class MyAttributeForClassAndStructOnly : Attribute
{
}
```

Se você tentar colocar o atributo acima em algo que não seja uma classe ou um struct, você receberá um erro do compilador como `Attribute 'MyAttributeForClassAndStructOnly' is not valid on this declaration type. It is only valid on 'class, struct' declarations`

```
C#

public class Foo
{
    // if the below attribute was uncommented, it would cause a compiler error
    // [MyAttributeForClassAndStructOnly]
    public Foo()
    { }
}
```

Como usar atributos anexados a um elemento de código

Os atributos atuam como metadados. Sem alguma força externa, eles não fazem nada.

Para localizar e agir sobre atributos, a reflexão é necessária. A reflexão permite que você escreva código em C# que examine outro código. Por exemplo, você pode usar Reflexão para obter informações sobre uma classe (adicionar `using System.Reflection;` à cabeça do código):

```
C#

TypeInfo typeInfo = typeof(MyClass).GetTypeInfo();
Console.WriteLine("The assembly qualified name of MyClass is " +
typeInfo.AssemblyQualifiedName);
```

Isso imprime algo como: `The assembly qualified name of MyClass is ConsoleApplication.MyClass, attributes, Version=1.0.0.0, Culture=neutral, PublicKeyToken=null`

Depois de ter um `TypeInfo` objeto (ou um `MemberInfo`, `FieldInfo` ou outro objeto), você poderá usar o `GetCustomAttributes` método. Esse método retorna uma coleção de `Attribute` objetos. Você também pode usar `GetCustomAttribute` e especificar um tipo de atributo.

Aqui está um exemplo de uso de `GetCustomAttributes` em uma instância `MemberInfo` de `MyClass` (que vimos anteriormente, tem um atributo `[Obsolete]` nele).

```
C#
```

```
var attrs = typeInfo.GetCustomAttributes();
foreach(var attr in attrs)
    Console.WriteLine("Attribute on MyClass: " + attr.GetType().Name);
```

Isso imprime no console: `Attribute on MyClass: ObsoleteAttribute`. Tente adicionar outros atributos a `MyClass`.

É importante observar que esses `Attribute` objetos são instanciados de forma preguiçosa. Ou seja, eles não serão instanciados até que você use `GetCustomAttribute` ou `GetCustomAttributes`. Eles também são instanciados a cada vez. Chamar `GetCustomAttributes` duas vezes em uma linha retorna duas instâncias diferentes de `ObsoleteAttribute`.

Atributos comuns em tempo de execução

Os atributos são usados por muitas ferramentas e estruturas. O NUnit usa atributos como `[Test]` e `[TestFixture]` que são usados pelo executor de teste NUnit. ASP.NET MVC usa atributos como `[Authorize]` e fornece um framework de filtros de ação para lidar com aspectos transversais nas ações do MVC. O PostSharp usa a sintaxe de atributo para permitir programação orientada a aspectos em C#.

Aqui estão alguns atributos notáveis integrados às bibliotecas de classes base do .NET Core:

- `[Obsolete]`. Este foi usado nos exemplos acima e reside no `System` namespace. É útil fornecer documentação declarativa sobre uma alteração da base de código. Uma mensagem pode ser fornecida na forma de uma cadeia de caracteres e outro parâmetro booliano pode ser usado para escalonar de um aviso do compilador para um erro do compilador.
- `[Conditional]`. Esse atributo está no `System.Diagnostics` namespace. Esse atributo pode ser aplicado a métodos (ou classes de atributo). Você deve passar uma cadeia de caracteres para o construtor. Se essa cadeia de caracteres não corresponder a uma `#define` diretiva, o compilador C# removerá todas as chamadas para esse método (mas não o método em si). Normalmente, você usa essa técnica para fins de depuração (diagnóstico).
- `[CallerMemberName]`. Esse atributo pode ser usado em parâmetros e reside no `System.Runtime.CompilerServices` namespace. `CallerMemberName` é um atributo usado para injetar o nome do método que está chamando outro método. É uma forma de eliminar 'cadeias de caracteres mágicas' ao implementar INotifyPropertyChanged em diversas estruturas de interface do usuário. Como exemplo:

```
C#
```

```
public class MyUIClass : INotifyPropertyChanged
{
    public event PropertyChangedEventHandler? PropertyChanged;

    public void RaisePropertyChanged([CallerMemberName] string propertyName =
default!)
    {
        PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(propertyName));
    }

    private string? _name;
    public string? Name
    {
        get { return _name;}
        set
        {
            if (value != _name)
            {
                _name = value;
                RaisePropertyChanged();   // notice that "Name" is not needed here
explicitly
            }
        }
    }
}
```

No código acima, você não precisa ter uma cadeia de caracteres literal `"Name"` . Usar `CallerMemberName` evita erros relacionados à digitação e também possibilita a refatoração/renomeação mais suave. Os atributos trazem poder declarativo para C#, mas são uma forma de código de metadados e não agem sozinhos.

Tutorial: Atualizar interfaces com métodos de interface padrão

19/06/2025

Você pode definir uma implementação ao declarar um membro de uma interface. O cenário mais comum é adicionar membros com segurança a uma interface já lançada e usada por inúmeros clientes.

Neste tutorial, você aprenderá a:

- ✓ Estenda as interfaces com segurança adicionando métodos com implementações.
- ✓ Crie implementações parametrizadas para fornecer maior flexibilidade.
- ✓ Permitir que os implementadores forneçam uma implementação mais específica na forma de uma substituição.

Pré-requisitos

Você precisa configurar seu computador para executar o .NET, incluindo o compilador C#. O compilador C# está disponível com o Visual Studio 2022 ou o SDK do .NET.

Visão geral do cenário

Este tutorial começa com a versão 1 de uma biblioteca de relacionamento com o cliente. Você pode obter o aplicativo inicial em nosso repositório de exemplos no GitHub. A empresa que criou essa biblioteca pretendia que os clientes com aplicativos existentes adotassem sua biblioteca. Eles forneceram definições mínimas de interface para os usuários de sua biblioteca implementarem. Esta é a definição de interface para um cliente:

C#

```
public interface ICustomer
{
    IEnumerable<IOrder> PreviousOrders { get; }

    DateTime DateJoined { get; }
    DateTime? LastOrder { get; }
    string Name { get; }
    IDictionary<DateTime, string> Reminders { get; }
}
```

Eles definiram uma segunda interface que representa uma ordem:

```C#
public interface IOrder
{
    DateTime Purchased { get; }
    decimal Cost { get; }
}
```

A partir dessas interfaces, a equipe pode criar uma biblioteca para que seus usuários criem uma experiência melhor para seus clientes. Seu objetivo era criar uma relação mais profunda com os clientes existentes e melhorar suas relações com novos clientes.

Agora, é hora de atualizar a biblioteca para a próxima versão. Um dos recursos solicitados permite um desconto de fidelidade para clientes que têm muitos pedidos. Esse novo desconto de fidelidade é aplicado sempre que um cliente faz um pedido. O desconto específico é uma propriedade de cada cliente individual. Cada implementação de `ICustomer` pode definir regras diferentes para o desconto de fidelidade.

A maneira mais natural de adicionar essa funcionalidade é aprimorar a `ICustomer` interface com um método para aplicar qualquer desconto de fidelidade. Essa sugestão de design causou preocupação entre desenvolvedores experientes: "As interfaces são imutáveis depois que são lançadas! Não faça uma alteração significativa! Você deve usar implementações de interface padrão para atualizar interfaces. Os autores da biblioteca podem adicionar novos membros à interface e fornecer uma implementação padrão para esses membros.

As implementações de interface padrão permitem que os desenvolvedores atualizem uma interface enquanto ainda permitem que os implementadores substituam essa implementação. Os usuários da biblioteca podem aceitar a implementação padrão como uma alteração da falha. Se as regras de negócio forem diferentes, elas poderão substituir a implementação.

Atualizar com métodos de interface padrão

A equipe concordou com a implementação padrão mais provável: um desconto de fidelidade para os clientes.

A atualização deve fornecer a funcionalidade para definir duas propriedades: o número de pedidos necessários para ser qualificado para o desconto e o percentual do desconto. Esses recursos o tornam um cenário perfeito para métodos de interface padrão. Você pode adicionar um método à `ICustomer` interface e fornecer a implementação mais provável. Todas as implementações existentes e novas podem usar a implementação padrão ou fornecer suas próprias.

Primeiro, adicione o novo método à interface, incluindo o corpo do método:

```C#
// Version 1:
public decimal ComputeLoyaltyDiscount()
{
    DateTime TwoYearsAgo = DateTime.Now.AddYears(-2);
    if ((DateJoined < TwoYearsAgo) && (PreviousOrders.Count() > 10))
    {
        return 0.10m;
    }
    return 0;
}
```

O autor da biblioteca escreveu um primeiro teste para verificar a implementação:

```C#
SampleCustomer c = new SampleCustomer("customer one", new DateTime(2010, 5, 31))
{
    Reminders =
    {
        { new DateTime(2010, 08, 12), "childs's birthday" },
        { new DateTime(1012, 11, 15), "anniversary" }
    }
};

SampleOrder o = new SampleOrder(new DateTime(2012, 6, 1), 5m);
c.AddOrder(o);

o = new SampleOrder(new DateTime(2103, 7, 4), 25m);
c.AddOrder(o);

// Check the discount:
ICustomer theCustomer = c;
Console.WriteLine($"Current discount: {theCustomer.ComputeLoyaltyDiscount()}");
```

Observe a seguinte parte do teste:

```C#
// Check the discount:
ICustomer theCustomer = c;
Console.WriteLine($"Current discount: {theCustomer.ComputeLoyaltyDiscount()}");
```

Essa conversão de `SampleCustomer` até `ICustomer` é necessária. A `SampleCustomer` classe não precisa fornecer uma implementação para `ComputeLoyaltyDiscount`; que é fornecida pela `ICustomer` interface. No entanto, a `SampleCustomer` classe não herda membros de suas

interfaces. Essa regra não mudou. Para chamar qualquer método declarado e implementado na interface, a variável deve ser o tipo da interface, `ICustomer` neste exemplo.

Fornecer parametrização

A implementação padrão é muito restritiva. Muitos consumidores desse sistema podem escolher limites diferentes para o número de compras, um tamanho diferente de associação ou um desconto percentual diferente. Você pode fornecer uma melhor experiência de atualização para mais clientes fornecendo uma maneira de definir esses parâmetros. Vamos adicionar um método estático que define esses três parâmetros que controlam a implementação padrão:

C#

```
// Version 2:
public static void SetLoyaltyThresholds(
    TimeSpan ago,
    int minimumOrders = 10,
    decimal percentageDiscount = 0.10m)
{
    length = ago;
    orderCount = minimumOrders;
    discountPercent = percentageDiscount;
}
private static TimeSpan length = new TimeSpan(365 * 2, 0,0,0); // two years
private static int orderCount = 10;
private static decimal discountPercent = 0.10m;

public decimal ComputeLoyaltyDiscount()
{
    DateTime start = DateTime.Now - length;

    if ((DateJoined < start) && (PreviousOrders.Count() > orderCount))
    {
        return discountPercent;
    }
    return 0;
}
```

Há muitos novos recursos de linguagem mostrados nesse pequeno fragmento de código. As interfaces agora podem incluir membros estáticos, incluindo campos e métodos. Modificadores de acesso diferentes também estão habilitados. Os outros campos são privados, o novo método é público. Qualquer dos modificadores são permitidos em membros de interface.

Os aplicativos que usam a fórmula geral para calcular o desconto de fidelidade, mas parâmetros diferentes, não precisam fornecer uma implementação personalizada; eles podem definir os argumentos por meio de um método estático. Por exemplo, o código a seguir define

uma "apreciação do cliente" que recompensa qualquer cliente com mais de um mês de associação:

```
C#

ICustomer.SetLoyaltyThresholds(new TimeSpan(30, 0, 0, 0), 1, 0.25m);
Console.WriteLine($"Current discount: {theCustomer.ComputeLoyaltyDiscount()}");
```

Estender a implementação padrão

O código que você adicionou até agora forneceu uma implementação conveniente para os cenários em que os usuários querem algo como a implementação padrão ou para fornecer um conjunto de regras não relacionado. Como última funcionalidade, vamos refatorar o código um pouco para habilitar cenários em que os usuários possam querer expandir sobre a implementação padrão.

Considere uma startup que deseja atrair novos clientes. Eles oferecem um desconto de 50% no primeiro pedido de um novo cliente. Caso contrário, os clientes existentes obterão o desconto padrão. O autor da biblioteca precisa mover a implementação padrão para um `protected static` método para que qualquer classe que implemente essa interface possa reutilizar o código em sua implementação. A implementação padrão do membro da interface também chama esse método compartilhado:

```
C#

public decimal ComputeLoyaltyDiscount() => DefaultLoyaltyDiscount(this);
protected static decimal DefaultLoyaltyDiscount(ICustomer c)
{
    DateTime start = DateTime.Now - length;

    if ((c.DateJoined < start) && (c.PreviousOrders.Count() > orderCount))
    {
        return discountPercent;
    }
    return 0;
}
```

Em uma implementação de uma classe que implementa essa interface, a substituição pode chamar o método auxiliar estático e estender essa lógica para fornecer o desconto de "novo cliente":

```
C#

public decimal ComputeLoyaltyDiscount()
{
```

```
    if (PreviousOrders.Any() == false)
        return 0.50m;
    else
        return ICustomer.DefaultLoyaltyDiscount(this);
}
```

Você pode ver todo o código concluído em nosso repositório de exemplos no GitHub. Você pode obter o aplicativo inicial em nosso repositório de exemplos no GitHub.

Esses novos recursos significam que as interfaces podem ser atualizadas com segurança quando há uma implementação padrão razoável para esses novos membros. Crie cuidadosamente interfaces para expressar ideias funcionais simples implementadas por várias classes. Isso facilita a atualização dessas definições de interface quando novos requisitos são descobertos para essa mesma ideia funcional.

Tutorial: Funcionalidade mix ao criar classes usando interfaces com métodos de interface padrão

Artigo • 04/08/2024

Você pode definir uma implementação ao declarar um membro de uma interface. Essa funcionalidade fornece novos recursos em que você pode definir implementações padrão para recursos declarados em interfaces. As classes podem escolher quando substituir a funcionalidade, quando usar a funcionalidade padrão e quando não declarar suporte a recursos discretos.

Neste tutorial, você aprenderá a:

- ✓ Crie interfaces com implementações que descrevem recursos discretos.
- ✓ Crie classes que usam as implementações padrão.
- ✓ Crie classes que substituam algumas ou todas as implementações padrão.

Pré-requisitos

Você precisa configurar o computador para executar o .NET, incluindo o compilador C#. O compilador C# está disponível com o Visual Studio 2022 ou o SDK do .NET.

Limitações de métodos de extensão

Uma maneira de implementar o comportamento que aparece como parte de uma interface é definindo métodos de extensão que fornecem o comportamento padrão. As interfaces declaram um conjunto mínimo de membros, fornecendo uma área de superfície maior para qualquer classe que implemente essa interface. Por exemplo, os métodos de extensão em Enumerable fornecem a implementação para qualquer sequência ser a fonte de uma consulta LINQ.

Os métodos de extensão são resolvidos no tempo de compilação, usando o tipo declarado da variável. As classes que implementam a interface podem fornecer uma implementação melhor para qualquer método de extensão. As declarações de variável devem corresponder ao tipo de implementação para permitir que o compilador escolha essa implementação. Quando o tipo de tempo de compilação corresponder à interface, as chamadas de método são resolvidas para o método de extensão. Outra preocupação com os métodos de extensão é que eles podem ser acessados pelo mesmo local que a

classe contendo os métodos de extensão. As classes não poderão declarar se devem ou não fornecer recursos declarados nos métodos de extensão.

Você pode declarar as implementações padrão como métodos de interface. Em seguida, cada classe usa automaticamente a implementação padrão. Qualquer classe que possa fornecer uma implementação melhor pode substituir a definição do método de interface por um algoritmo melhor. De certa forma, essa técnica é semelhante à forma de utilização dos métodos de extensão.

Neste artigo, você aprende como as implementações de interface padrão habilitam novos cenários.

Criar o aplicativo

Considere um aplicativo de automação residencial. Você provavelmente tem muitos tipos de luzes e indicadores que podem ser usados em toda a casa. Cada luz deve dar suporte a APIs para ativá-las, desativá-las e relatar o estado atual. Algumas luzes e indicadores podem dar suporte a outros recursos, como:

- Ativar a luz e desligá-la após um temporizador.
- Piscar a luz por um período de tempo.

Alguns desses recursos estendidos podem ser emulados em dispositivos que dão suporte ao conjunto mínimo. Isso indica o fornecimento de uma implementação padrão. Para os dispositivos com mais funcionalidades internas, o software do dispositivo usaria os recursos nativos. Para outras luzes, eles podem optar por implementar a interface e usar a implementação padrão.

Os membros de interface padrão são uma solução melhor para esse cenário do que os métodos de extensão. Os autores de classe podem controlar quais interfaces eles escolhem implementar. Essas interfaces escolhidas estão disponíveis como métodos. Além disso, como os métodos de interface padrão são virtuais por padrão, a expedição de método sempre escolhe a implementação na classe.

Vamos criar o código para demonstrar essas diferenças.

Criar interfaces

Comece criando a interface que define o comportamento de todas as luzes:

```
C#
```

```
public interface ILight
{
    void SwitchOn();
    void SwitchOff();
    bool IsOn();
}
```

Uma luminária suspensa básica pode implementar essa interface, conforme mostrado no seguinte código:

```C#
public class OverheadLight : ILight
{
    private bool isOn;
    public bool IsOn() => isOn;
    public void SwitchOff() => isOn = false;
    public void SwitchOn() => isOn = true;

    public override string ToString() => $"The light is {(isOn ? "on" :
"off")}";
}
```

Neste tutorial, o código não conduz os dispositivos IoT, mas emula essas atividades gravando mensagens no console. É possível explorar o código sem automatizar sua casa.

Em seguida, vamos definir a interface para uma luz que pode ser desativada automaticamente após um tempo limite:

```C#
public interface ITimerLight : ILight
{
    Task TurnOnFor(int duration);
}
```

Você pode adicionar uma implementação básica à luz suspensa, mas uma solução melhor é modificar essa definição de interface para fornecer uma implementação padrão `virtual`:

```C#
public interface ITimerLight : ILight
{
    public async Task TurnOnFor(int duration)
    {
```

```
        Console.WriteLine("Using the default interface method for the
ITimerLight.TurnOnFor.");
        SwitchOn();
        await Task.Delay(duration);
        SwitchOff();
        Console.WriteLine("Completed ITimerLight.TurnOnFor sequence.");
    }
}
```

A classe `OverheadLight` pode implementar a função de temporizador declarando suporte à interface:

```
C#

public class OverheadLight : ITimerLight { }
```

Um tipo de luz diferente pode dar suporte a um protocolo mais sofisticado. Ele pode fornecer sua própria implementação para `TurnOnFor`, conforme mostrado no seguinte código:

```
C#

public class HalogenLight : ITimerLight
{
    private enum HalogenLightState
    {
        Off,
        On,
        TimerModeOn
    }

    private HalogenLightState state;
    public void SwitchOn() => state = HalogenLightState.On;
    public void SwitchOff() => state = HalogenLightState.Off;
    public bool IsOn() => state != HalogenLightState.Off;
    public async Task TurnOnFor(int duration)
    {
        Console.WriteLine("Halogen light starting timer function.");
        state = HalogenLightState.TimerModeOn;
        await Task.Delay(duration);
        state = HalogenLightState.Off;
        Console.WriteLine("Halogen light finished custom timer function");
    }

    public override string ToString() => $"The light is {state}";
}
```

Ao contrário da substituição de métodos de classe virtual, a declaração de `TurnOnFor` na classe `HalogenLight` não usa a palavra-chave `override`.

Recursos combinados

As vantagens dos métodos de interface padrão ficam mais claras à medida que você introduz recursos mais avançados. O uso de interfaces permite combinar recursos. Também permite que cada autor de classe escolha entre uma implementação padrão ou uma implementação personalizada. Vamos adicionar uma interface com uma implementação padrão para uma luz piscando:

C#

```
public interface IBlinkingLight : ILight
{
    public async Task Blink(int duration, int repeatCount)
    {
        Console.WriteLine("Using the default interface method for
IBlinkingLight.Blink.");
        for (int count = 0; count < repeatCount; count++)
        {
            SwitchOn();
            await Task.Delay(duration);
            SwitchOff();
            await Task.Delay(duration);
        }
        Console.WriteLine("Done with the default interface method for
IBlinkingLight.Blink.");
    }
}
```

A implementação padrão permite que qualquer luz pisque. A luz suspensa pode adicionar recursos de temporizador e piscar usando a implementação padrão:

C#

```
public class OverheadLight : ILight, ITimerLight, IBlinkingLight
{
    private bool isOn;
    public bool IsOn() => isOn;
    public void SwitchOff() => isOn = false;
    public void SwitchOn() => isOn = true;

    public override string ToString() => $"The light is {(isOn ? "on" :
"off")}";
}
```

Um novo tipo de luz `LEDLight`, dá suporte à função de temporizador e à função de piscar diretamente. Esse estilo de luz implementa as interfaces `ITimerLight` e `IBlinkingLight` e substitui o método `Blink`:

C#

```
public class LEDLight : IBlinkingLight, ITimerLight, ILight
{
    private bool isOn;
    public void SwitchOn() => isOn = true;
    public void SwitchOff() => isOn = false;
    public bool IsOn() => isOn;
    public async Task Blink(int duration, int repeatCount)
    {
        Console.WriteLine("LED Light starting the Blink function.");
        await Task.Delay(duration * repeatCount);
        Console.WriteLine("LED Light has finished the Blink function.");
    }

    public override string ToString() => $"The light is {(isOn ? "on" :
"off")}";
}
```

Um `ExtraFancyLight` pode dar suporte diretamente a funções de piscar e temporizador:

C#

```
public class ExtraFancyLight : IBlinkingLight, ITimerLight, ILight
{
    private bool isOn;
    public void SwitchOn() => isOn = true;
    public void SwitchOff() => isOn = false;
    public bool IsOn() => isOn;
    public async Task Blink(int duration, int repeatCount)
    {
        Console.WriteLine("Extra Fancy Light starting the Blink function.");
        await Task.Delay(duration * repeatCount);
        Console.WriteLine("Extra Fancy Light has finished the Blink
function.");
    }
    public async Task TurnOnFor(int duration)
    {
        Console.WriteLine("Extra Fancy light starting timer function.");
        await Task.Delay(duration);
        Console.WriteLine("Extra Fancy light finished custom timer
function");
    }

    public override string ToString() => $"The light is {(isOn ? "on" :
"off")}";
}
```

O `HalogenLight` criado anteriormente não dá suporte a piscar. Portanto, não adicione `IBlinkingLight` à lista de interfaces com suporte.

Detectar os tipos de luz usando a correspondência de padrões

Em seguida, vamos gravar um código de teste. Você pode usar o recurso de correspondência de padrões do C# para determinar as funcionalidades de uma luz examinando quais interfaces ele dá suporte. O método a seguir exercita as funcionalidades com suporte de cada luz:

C#

```
private static async Task TestLightCapabilities(ILight light)
{
    // Perform basic tests:
    light.SwitchOn();
    Console.WriteLine($"\tAfter switching on, the light is {(light.IsOn() ?
"on" : "off")}");
    light.SwitchOff();
    Console.WriteLine($"\tAfter switching off, the light is {(light.IsOn() ?
"on" : "off")}");

    if (light is ITimerLight timer)
    {
        Console.WriteLine("\tTesting timer function");
        await timer.TurnOnFor(1000);
        Console.WriteLine("\tTimer function completed");
    }
    else
    {
        Console.WriteLine("\tTimer function not supported.");
    }

    if (light is IBlinkingLight blinker)
    {
        Console.WriteLine("\tTesting blinking function");
        await blinker.Blink(500, 5);
        Console.WriteLine("\tBlink function completed");
    }
    else
    {
        Console.WriteLine("\tBlink function not supported.");
    }
}
```

O código a seguir no método `Main` cria cada tipo de luz na sequência e testa essa luz:

C#

```
static async Task Main(string[] args)
{
    Console.WriteLine("Testing the overhead light");
```

```
    var overhead = new OverheadLight();
    await TestLightCapabilities(overhead);
    Console.WriteLine();

    Console.WriteLine("Testing the halogen light");
    var halogen = new HalogenLight();
    await TestLightCapabilities(halogen);
    Console.WriteLine();

    Console.WriteLine("Testing the LED light");
    var led = new LEDLight();
    await TestLightCapabilities(led);
    Console.WriteLine();

    Console.WriteLine("Testing the fancy light");
    var fancy = new ExtraFancyLight();
    await TestLightCapabilities(fancy);
    Console.WriteLine();
}
```

Como o compilador determina a melhor implementação

Esse cenário mostra uma interface base sem implementações. A adição de um método à interface `ILight`, introduz novas complexidades. As regras de linguagem que regem os métodos de interface padrão minimizam o efeito nas classes concretas que implementam várias interfaces derivadas. Vamos aprimorar a interface original com um novo método para demonstrar como isso altera seu uso. Cada luz do indicador pode relatar seu status de energia como um valor enumerado:

C#

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

A implementação padrão pressupõe que não há energia:

C#

```
public interface ILight
{
```

```
    void SwitchOn();
    void SwitchOff();
    bool IsOn();
    public PowerStatus Power() => PowerStatus.NoPower;
}
```

Essas alterações são compiladas de forma limpa, embora `ExtraFancyLight` declare suporte para a interface `ILight` e as interfaces derivadas `ITimerLight` e `IBlinkingLight`. Há apenas uma implementação "mais próxima" declarada na interface `ILight`. Qualquer classe que declarasse uma substituição se tornaria a única implementação "mais próxima". Você viu exemplos nas classes anteriores que substituíram os membros de outras interfaces derivadas.

Evite substituir o mesmo método em várias interfaces derivadas. Isso cria uma chamada de método ambíguo sempre que uma classe implementa ambas as interfaces derivadas. O compilador não pode escolher um único método, logo, um erro é gerado. Por exemplo, se `IBlinkingLight` e `ITimerLight` implementaram uma substituição de `PowerStatus`, `OverheadLight` precisaria fornecer uma substituição mais específica. Caso contrário, o compilador não poderá escolher entre as implementações nas duas interfaces derivadas. Este cenário é mostrado no seguinte diagrama:

O diagrama anterior ilustra a ambiguidade. `OverheadLight` não fornece uma implementação de `ILight.PowerStatus`. Ambos `IBlinkingLight` e `ITimerLight` forneçam substituições mais específicas. Uma chamada para `ILight.PowerStatus` em uma instância de `OverheadLight` é ambígua. Você precisa adicionar uma nova substituição em `OverheadLight` para resolver a ambiguidade.

Normalmente, você pode evitar essa situação mantendo as definições de interface pequenas e focadas em um recurso. Nesse cenário, cada funcionalidade de uma luz é a própria interface, e várias interfaces são herdadas apenas por classes.

Este exemplo mostra um cenário em que você pode definir recursos discretos que podem ser misturados em classes. Você declara qualquer conjunto de funcionalidades com suporte declarando a quais interfaces uma classe dá suporte. O uso de métodos de interface padrão virtual permite que as classes usem ou definam uma implementação diferente para um ou todos os métodos de interface. Essa funcionalidade de linguagem fornece novas maneiras de modelar os sistemas do mundo real sendo criados. Os métodos de interface padrão fornecem uma maneira mais clara de expressar classes relacionadas que podem combinar diferentes recursos usando implementações virtuais dessas funcionalidades.

Colaborar conosco no GitHub

A fonte deste conteúdo pode ser encontrada no GitHub, onde você também pode criar e revisar problemas e solicitações de pull. Para obter mais informações, confira o nosso guia para colaboradores.

.NET

Comentários do .NET

O .NET é um projeto código aberto. Selecione um link para fornecer comentários:

Abrir um problema de documentação

Fornecer comentários sobre o produto

Arvores de expressão

Artigo • 16/04/2025

As árvores de expressão representam código em uma estrutura de dados semelhante a uma árvore, em que cada nó é uma expressão, por exemplo, uma chamada de método ou uma operação binária, como `x < y`.

Se você usou LINQ, terá experiência com uma biblioteca avançada em que os `Func` tipos fazem parte do conjunto de API. (Se você não estiver familiarizado com LINQ, provavelmente deseja ler o tutorial linq e o artigo sobre expressões lambda antes desta.) As Árvores de Expressão fornecem uma interação mais avançada com os argumentos que são funções.

Você escreve argumentos de função, normalmente usando expressões Lambda, ao criar consultas LINQ. Em uma consulta LINQ típica, esses argumentos de função são transformados em um delegado que o compilador cria.

Você já escreve código que usa árvores de expressão. As APIs LINQ do Entity Framework aceitam árvores de expressão como argumentos para o padrão de expressão de consulta LINQ. Isso permite que o Entity Framework traduza a consulta que você escreveu em C# para o SQL que é executado no mecanismo de banco de dados. Outro exemplo é Moq, que é uma estrutura de simulação popular para .NET.

Quando você deseja ter uma interação mais avançada, precisa usar *árvores de expressão*. As Árvores de Expressão representam o código como uma estrutura que você examina, modifica ou executa. Essas ferramentas fornecem o poder de manipular o código durante o tempo de execução. Você escreve um código que examina algoritmos em execução ou injeta novos recursos. Em cenários mais avançados, você modifica algoritmos em execução e até converte expressões C# em outro formulário para execução em outro ambiente.

Você compila e executa o código representado por árvores de expressão. A criação e a execução de árvores de expressão permitem a modificação dinâmica do código executável, a execução de consultas LINQ em vários bancos de dados e a criação de consultas dinâmicas. Para obter mais informações sobre árvores de expressão no LINQ, consulte Como usar árvores de expressão para criar consultas dinâmicas.

As árvores de expressão também são usadas no DLR (Dynamic Language Runtime) para fornecer interoperabilidade entre linguagens dinâmicas e .NET e para permitir que os gravadores de compilador emitam árvores de expressão em vez de CIL (linguagem intermediária da Microsoft). Para obter mais informações sobre a DLR, consulte Visão geral do Dynamic Language Runtime.

Você pode fazer com que o compilador C# ou Visual Basic crie uma árvore de expressão para você com base em uma expressão lambda anônima ou crie árvores de expressão manualmente

usando o System.Linq.Expressions namespace.

Quando uma expressão lambda é atribuída a uma variável de tipo Expression<TDelegate>, o compilador emite o código para criar uma árvore de expressão que representa a expressão lambda.

Os exemplos de código a seguir demonstram como fazer com que o compilador C# crie uma árvore de expressão que represente a expressão `num => num < 5` lambda.

```
C#

Expression<Func<int, bool>> lambda = num => num < 5;
```

Você cria árvores de expressão em seu código. Você cria a árvore criando cada nó e anexando os nós a uma estrutura de árvore. Você aprenderá a criar expressões no artigo sobre como criar árvores de expressão.

Árvores de expressão são imutáveis. Se você quiser modificar uma árvore de expressão, será necessário construir uma nova árvore de expressão copiando a existente e substituindo nós nela. Você usa um visitante de árvore expressão para percorrer a árvore de expressão existente. Para obter mais informações, consulte o artigo sobre como traduzir árvores de expressão.

Depois de criar uma árvore de expressão, execute o código representado pela árvore de expressão.

Limitações

O compilador C# gera árvores de expressão somente a partir de lambdas de expressão (ou lambdas de linha única). Ele não consegue analisar instruções lambdas (ou lambdas de várias linhas). Para obter mais informações sobre expressões lambda em C#, consulte Expressões Lambda.

Há alguns elementos de linguagem C# mais recentes que não se traduzem bem em árvores de expressão. Árvores de expressão não podem conter `await` expressões ou `async` expressões lambda. Muitos dos recursos adicionados em C# 6 e posterior não aparecem exatamente como escritos em árvores de expressão. Em vez disso, os recursos mais recentes são expostos em árvores de expressão na sintaxe equivalente e anterior, sempre que possível. Outros constructos não estão disponíveis. Isso significa que o código que interpreta árvores de expressão funciona da mesma forma quando novos recursos de linguagem são introduzidos. No entanto, mesmo com essas limitações, as árvores de expressão permitem que você crie algoritmos dinâmicos que dependem da interpretação e modificação de código que é

representado como uma estrutura de dados. Ele permite que bibliotecas avançadas, como o Entity Framework, realizem o que fazem.

As árvores de expressão não dão suporte a novos tipos de nó de expressão. Seria uma mudança significativa para todas as bibliotecas que interpretam árvores de expressão a introdução de novos tipos de nó. A lista a seguir inclui a maioria dos elementos de linguagem C# que não podem ser usados:

- Métodos condicionais removidos da saída
- base acesso
- Expressões de grupo de métodos, incluindo um grupo de métodos *address-of* (&) e expressões de método anônimo
- Referências a funções locais
- Instruções, incluindo atribuição (`=`) e expressões com corpo de instrução
- Métodos parciais com apenas uma declaração de definição
- Operações de ponteiro não seguras
- dynamic Operações
- Operadores de coalescing com null ou default literal à esquerda, atribuição de coalescência nula e o operador de propagação nulo (?.)
- Inicializadores de matriz multidimensional, propriedades indexadas e inicializadores de dicionário
- Expressões de coleção.
- throw Expressões
- Acessando static virtual ou os membros da interface abstract
- Expressões lambda que têm atributos
- Cadeias de caracteres interpoladas
- Conversões de cadeia de caracteres UTF-8 ou literais de cadeia de caracteres UTF-8
- Invocações de método usando argumentos variáveis, argumentos nomeados ou argumentos opcionais
- Expressões que usam System.Index ou o operador System.Range, index "from end" (^ ou expressões de intervalo (..)
- async expressões lambda ou await expressões, incluindo await foreach e await using
- Literais de tupla, conversões de tupla, tupla == ou != ou with expressões
- Descarta (_), desconstrução de atribuição, o operador de correspondência de padrões is ou a expressão de switch correspondência de padrões
- Chamada COM com `ref` omitida nos argumentos
- ref, in ou out parâmetros, `ref` valores retornados, `out` argumentos ou quaisquer valores do ref struct tipo

Arvores de expressão – dados que definem código

18/06/2025

Uma árvore de expressão é uma estrutura de dados que define o código. As árvores de expressão são baseadas nas mesmas estruturas que um compilador usa para analisar o código e gerar a saída compilada. Ao ler este artigo, você observará um pouco de similaridade entre árvores de expressão e os tipos usados nas APIs Roslyn para criar Analisadores e CodeFixes. (Analisadores e CodeFixes são pacotes NuGet que executam análise estática no código e sugerem possíveis correções para um desenvolvedor.) Os conceitos são semelhantes e o resultado final é uma estrutura de dados que permite examinar o código-fonte de forma significativa. No entanto, as árvores de expressão são baseadas em um conjunto diferente de classes e APIs que as APIs Roslyn. Aqui está uma linha de código:

```C#
var sum = 1 + 2;
```

Se você analisar o código anterior como uma árvore de expressão, a árvore conterá vários nós. O nó mais externo é uma instrução de declaração de variável com atribuição (`var sum = 1 + 2;`). Esse nó mais externo contém vários nós filho: uma declaração de variável, um operador de atribuição e uma expressão que representa o lado direito do sinal de igual. Essa expressão é ainda subdividida em expressões que representam a operação de adição e os operandos esquerdo e direito da adição.

Vamos detalhar um pouco mais as expressões que compõem o lado direito do sinal de igual. A expressão é `1 + 2` uma expressão binária. Mais especificamente, é uma expressão de adição binária. Uma expressão de adição binária tem dois filhos, representando os nós esquerdo e direito da expressão de adição. Aqui, ambos os nós são expressões constantes: o operando esquerdo é o valor `1` e o operando à direita é o valor `2`.

Visualmente, a instrução inteira é uma árvore: você pode começar no nó raiz e viajar para cada nó na árvore para ver o código que compõe a instrução:

- Instrução de declaração de variável com atribuição (`var sum = 1 + 2;`)
  - Declaração de tipo de variável implícita (`var sum`)
    - Palavra-chave var implícita (`var`)
    - Declaração de nome de variável (`sum`)
  - Operador de atribuição (`=`)
  - Expressão de adição binária (`1 + 2`)

- Operando esquerdo (`1`)
- Operador de adição (`+`)
- Operando direito (`2`)

A árvore anterior pode parecer complicada, mas é muito poderosa. Após o mesmo processo, você decompõe expressões muito mais complicadas. Considere esta expressão:

C#

```
var finalAnswer = this.SecretSauceFunction(
    currentState.createInterimResult(), currentState.createSecondValue(1, 2),
    decisionServer.considerFinalOptions("hello")) +
    MoreSecretSauce('A', DateTime.Now, true);
```

A expressão anterior também é uma declaração de variável com uma atribuição. Neste exemplo, o lado direito da atribuição é uma árvore muito mais complicada. Você não vai decompor essa expressão, mas considere quais podem ser os diferentes componentes. Há chamadas de método que utilizam o objeto atual como receptor: algumas possuem um receptor explícito `this`, enquanto outras não. Há chamadas de método usando outros objetos receptores, há argumentos constantes de diferentes tipos. Por fim, há um operador de adição binária. Dependendo do tipo de retorno de `SecretSauceFunction()` ou `MoreSecretSauce()`, esse operador de adição binária pode ser uma chamada de método para um operador de adição substituído, resolvendo em uma chamada de método estático ao operador de adição binária definido para uma classe.

Apesar dessa complexidade percebida, a expressão anterior cria uma estrutura de árvore que pode ser navegada tão facilmente quanto o primeiro exemplo. Você continua percorrendo os nós filho para encontrar os nós folha na expressão. Os nós pai terão referências aos filhos, sendo que cada nó tem uma propriedade que descreve o tipo de nó.

A estrutura de uma árvore de expressão é muito consistente. Depois de aprender as noções básicas, você entenderá até mesmo o código mais complexo quando ele é representado como uma árvore de expressão. A elegância na estrutura de dados explica como o compilador C# analisa os programas C# mais complexos e cria uma saída adequada desse código-fonte complicado.

Depois de se familiarizar com a estrutura das árvores de expressão, você descobrirá que o conhecimento adquirido rapidamente permite trabalhar com muitos cenários mais avançados. O potencial das árvores de expressão é incrível.

Além de traduzir algoritmos a serem executados em outros ambientes, as árvores de expressão facilitam a gravação de algoritmos que inspecionam o código antes de executá-lo. Você escreve um método cujos argumentos são expressões e, em seguida, examina essas expressões

antes de executar o código. A Árvore de Expressão é uma representação completa do código: você vê valores de qualquer subexpressão. Você vê os nomes dos métodos e das propriedades. Você vê o valor de qualquer expressão constante. Você converte uma árvore de expressão em um delegado executável e executa o código.

As APIs para Árvores de Expressão permitem criar árvores que representam quase qualquer construção de código válida. No entanto, para manter as coisas o mais simples possível, alguns idiomas C# não podem ser criados em uma árvore de expressão. Um exemplo é expressões assíncronas (usando as palavras-chave `async` e `await`). Se suas necessidades exigirem algoritmos assíncronos, você precisará manipular os `Task` objetos diretamente, em vez de contar com o suporte do compilador. Outro exemplo é na criação de loops. Normalmente, você cria esses loops usando `for`, `foreach` `while` ou `do` loops. Como você vê posteriormente nesta série, as APIs para árvores de expressão dão suporte a uma única expressão de loop, com `break` e `continue` expressões que controlam a repetição do loop.

A única coisa que você não pode fazer é modificar uma árvore de expressão. Árvores de expressão são estruturas de dados imutáveis. Se você quiser alterar (alterar) uma árvore de expressão, deverá criar uma nova árvore que seja uma cópia do original, mas com as alterações desejadas.

Suporte ao Runtime do .NET para árvores de expressão

19/06/2025

Há uma grande lista de classes no runtime do .NET que funcionam com árvores de expressão. Você pode ver a lista completa em System.Linq.Expressions. Em vez de enumerar a lista completa, vamos entender como as classes de runtime foram projetadas.

No design de linguagem, uma expressão é um corpo de código que avalia e retorna um valor. As expressões podem ser simples: a expressão `1` constante retorna o valor constante de 1. Eles podem ser mais complicados: a expressão `(-B + Math.Sqrt(B*B - 4 * A * C)) / (2 * A)` retorna uma raiz para uma equação quadrática (no caso em que a equação tem uma solução).

System.Linq.Expression e tipos derivados

Uma das complexidades de trabalhar com árvores de expressão é que muitos tipos diferentes de expressões são válidos em muitos locais em programas. Considere uma expressão de atribuição. O lado direito de uma atribuição pode ser um valor constante, uma variável, uma expressão de chamada de método ou outros elementos. Essa flexibilidade de linguagem significa que você poderá encontrar muitos tipos diferentes de expressão em qualquer lugar nos nós de uma árvore ao percorrer uma árvore de expressão. Portanto, quando você trabalha com o tipo de expressão base, essa é a maneira mais simples de trabalhar. No entanto, às vezes você precisa saber mais. A classe expressão base contém uma `NodeType` propriedade para essa finalidade. Retorna um `ExpressionType`, que é uma enumeração de tipos de expressão possíveis. Quando você sabe qual é o tipo do nó, pode convertê-la para esse tipo e executar ações específicas, conhecendo o tipo do nó de expressão. Você pode buscar por tipos específicos de nós e, em seguida, trabalhar com as propriedades específicas desse tipo de expressão.

Por exemplo, esse código imprime o nome de uma variável para uma expressão de acesso variável. O código a seguir mostra a prática de verificar o tipo de nó e, em seguida, converter em uma expressão de acesso variável e verificar as propriedades do tipo de expressão específico:

C#

```
Expression<Func<int, int>> addFive = (num) => num + 5;

if (addFive is LambdaExpression lambdaExp)
{
    var parameter = lambdaExp.Parameters[0];  -- first
```

```
    Console.WriteLine(parameter.Name);
    Console.WriteLine(parameter.Type);
}
```

Criar árvores de expressão

A `System.Linq.Expression` classe também contém muitos métodos estáticos para criar expressões. Esses métodos criam um nó de expressão usando os argumentos fornecidos para seus filhos. Dessa forma, você cria uma expressão com os nós folha. Por exemplo, esse código cria uma expressão Add:

C#

```
// Addition is an add expression for "1 + 2"
var one = Expression.Constant(1, typeof(int));
var two = Expression.Constant(2, typeof(int));
var addition = Expression.Add(one, two);
```

Você pode ver neste exemplo simples que muitos tipos estão envolvidos na criação e no trabalho com árvores de expressão. Essa complexidade é necessária para fornecer os recursos do vocabulário avançado fornecido pela linguagem C#.

Navegar pelas APIs

Há tipos de nó de expressão que correspondem a quase todos os elementos de sintaxe da linguagem C#. Cada tipo tem métodos específicos para esse tipo de elemento de linguagem. É muita coisa para guardar na sua cabeça ao mesmo tempo. Em vez de tentar memorizar tudo, aqui estão as técnicas que você usa para trabalhar com árvores de expressão:

1. Observar os membros da enumeração `ExpressionType` para determinar possíveis nós que devem ser examinados. Essa lista ajuda quando você deseja percorrer e entender uma árvore de expressão.
2. Examine os membros estáticos da `Expression` classe para criar uma expressão. Esses métodos podem compilar qualquer tipo de expressão em um conjunto de seu nós filho.
3. Examine a `ExpressionVisitor` classe para criar uma árvore de expressão modificada.

Você encontra mais à medida que olha para cada uma dessas três áreas. Invariavelmente, você encontra o que precisa quando começa com uma dessas três etapas.

Executar árvores de expressão

19/06/2025

Uma *árvore de expressão* é uma estrutura de dados que representa algum código. Não é um código compilado e executável. Se você quiser executar o código .NET representado por uma árvore de expressão, deverá convertê-lo em instruções IL executáveis. A execução de uma árvore de expressão pode retornar um valor ou pode apenas executar uma ação, como chamar um método.

Somente árvores de expressão que representam expressões lambda podem ser executadas. As árvores de expressão que representam expressões lambda são do tipo LambdaExpression ou Expression<TDelegate>. Para executar essas árvores de expressão, chame o Compile método para criar um delegado executável e, em seguida, invoque o delegado.

> ⓘ **Observação**
>
> Se o tipo do delegado não for conhecido, ou seja, a expressão lambda é do tipo **LambdaExpression** e não **Expression<TDelegate>**, chame o **DynamicInvoke** método no delegado em vez de invocá-lo diretamente.

Se uma árvore de expressão não representar uma expressão lambda, você poderá criar uma nova expressão lambda que tenha a árvore de expressão original como seu corpo, chamando o Lambda<TDelegate>(Expression, IEnumerable<ParameterExpression>) método. Em seguida, você pode executar a expressão lambda, conforme descrito anteriormente nesta seção.

Expressões lambda para funções

Você pode converter qualquer LambdaExpression ou qualquer tipo derivado de LambdaExpression em IL executável. Outros tipos de expressão não podem ser convertidos diretamente em código. Essa restrição tem pouco efeito na prática. Expressões Lambda são os únicos tipos de expressões que você deseja executar convertendo em IL (linguagem intermediária executável). (Pense no que significaria executar diretamente um System.Linq.Expressions.ConstantExpression. Isso significaria algo útil?) Qualquer árvore de expressão que seja um System.Linq.Expressions.LambdaExpression ou um tipo derivado de `LambdaExpression` pode ser convertida em IL. O tipo System.Linq.Expressions.Expression<TDelegate> de expressão é o único exemplo concreto nas bibliotecas do .NET Core. Ele é usado para representar uma expressão que corresponde a qualquer tipo de delegado. Como esse tipo corresponde a um tipo delegado, o .NET pode examinar a expressão e gerar IL para um delegado adequado que atenda à assinatura da expressão lambda. O tipo de delegado é baseado no tipo de expressão. Você deve conhecer o

tipo de retorno e a lista de argumentos se quiser usar o objeto delegado de maneira fortemente tipada. O `LambdaExpression.Compile()` método retorna o `Delegate` tipo. Você precisará convertê-lo no tipo delegado correto para fazer com que as ferramentas de tempo de compilação verifiquem a lista de argumentos ou o tipo de retorno.

Na maioria dos casos, existe um mapeamento simples entre uma expressão e seu representante correspondente. Por exemplo, uma árvore de expressão representada por `Expression<Func<int>>` seria convertida em um delegado do tipo `Func<int>`. Para uma expressão lambda com qualquer tipo de retorno e lista de argumentos, existe um tipo delegado que é o tipo de destino para o código executável representado por essa expressão lambda.

O System.Linq.Expressions.LambdaExpression tipo contém LambdaExpression.Compile e LambdaExpression.CompileToMethod os membros que você usaria para converter uma árvore de expressão em código executável. O `Compile` método cria um delegado. O `CompileToMethod` método atualiza um System.Reflection.Emit.MethodBuilder objeto com o IL que representa a saída compilada da árvore de expressão.

> **ⓘ Importante**
>
> `CompileToMethod` só está disponível no .NET Framework, não no .NET Core ou no .NET 5 e posterior.

Como opção, você também pode fornecer um System.Runtime.CompilerServices.DebugInfoGenerator que receberá as informações de depuração de símbolo para o objeto delegado gerado. O `DebugInfoGenerator` fornece informações completas de depuração sobre o delegado gerado.

Você converteria uma expressão em um delegado usando o seguinte código:

C#

```
Expression<Func<int>> add = () => 1 + 2;
var func = add.Compile(); // Create Delegate
var answer = func(); // Invoke Delegate
Console.WriteLine(answer);
```

O exemplo de código a seguir demonstra os tipos concretos usados ao compilar e executar uma árvore de expressão.

C#

```
Expression<Func<int, bool>> expr = num => num < 5;

// Compiling the expression tree into a delegate.
Func<int, bool> result = expr.Compile();

// Invoking the delegate and writing the result to the console.
Console.WriteLine(result(4));

// Prints True.

// You can also use simplified syntax
// to compile and run an expression tree.
// The following line can replace two previous statements.
Console.WriteLine(expr.Compile()(4));

// Also prints True.
```

O exemplo de código a seguir demonstra como executar uma árvore de expressão que representa elevar um número a uma potência criando uma expressão lambda e executando-a. O resultado, representado pelo número elevado à potência, é exibido.

C#

```
// The expression tree to execute.
BinaryExpression be = Expression.Power(Expression.Constant(2d),
Expression.Constant(3d));

// Create a lambda expression.
Expression<Func<double>> le = Expression.Lambda<Func<double>>(be);

// Compile the lambda expression.
Func<double> compiledExpression = le.Compile();

// Execute the lambda expression.
double result = compiledExpression();

// Display the result.
Console.WriteLine(result);

// This code produces the following output:
// 8
```

Execução e tempo de vida

Execute o código invocando o delegado criado quando você chamou `LambdaExpression.Compile()`. O código anterior, `add.Compile()`, retorna um delegado. Você invoca esse delegado chamando `func()`, que executa o código.

Esse delegado representa o código na árvore de expressão. Você pode reter o identificador para esse delegado e invocá-lo mais tarde. Você não precisa compilar a árvore de expressão sempre que quiser executar o código que ela representa. (Lembre-se de que as árvores de expressão são imutáveis e compilar a mesma árvore de expressão mais tarde cria um delegado que executa o mesmo código.)

> ⊗ **Cuidado**
>
> Não crie mecanismos de cache mais sofisticados para aumentar o desempenho evitando chamadas de compilação desnecessárias. Comparar duas árvores de expressão arbitrária para determinar se elas representam o mesmo algoritmo é uma operação demorada. O tempo de computação que você economiza evitando chamadas `LambdaExpression.Compile()` extras provavelmente é mais do que consumido pelo tempo de execução do código que determina se duas árvores de expressão diferentes resultam no mesmo código executável.

Advertências

Compilar uma expressão lambda para um delegado e invocar esse delegado é uma das operações mais simples que você pode executar com uma árvore de expressão. No entanto, mesmo com essa operação simples, há ressalvas que você deve estar ciente.

As Expressões Lambda criam fechamentos em todas as variáveis locais que são referenciadas na expressão. Você deve assegurar que todas as variáveis que farão parte do delegado são utilizáveis no local em que você chamar `Compile` e no momento em que você executar o delegado resultante. O compilador garante que as variáveis estejam no escopo. No entanto, se sua expressão acessar uma variável que implementa `IDisposable`, é possível que seu código possa descartar o objeto enquanto ele ainda é mantido pela árvore de expressão.

Por exemplo, esse código funciona bem, porque `int` não implementa `IDisposable`:

C#

```
private static Func<int, int> CreateBoundFunc()
{
    var constant = 5; // constant is captured by the expression tree
    Expression<Func<int, int>> expression = (b) => constant + b;
    var rVal = expression.Compile();
    return rVal;
}
```

O delegado capturou uma referência à variável `constant` local. Essa variável é acessada a qualquer momento no futuro, quando a função retornada por `CreateBoundFunc` é executada.

No entanto, considere a seguinte classe (bastante inventada) que implementa System.IDisposable:

C#

```
public class Resource : IDisposable
{
    private bool _isDisposed = false;
    public int Argument
    {
        get
        {
            if (!_isDisposed)
                return 5;
            else throw new ObjectDisposedException("Resource");
        }
    }

    public void Dispose()
    {
        _isDisposed = true;
    }
}
```

Se você usar isto em uma expressão, como mostrado no código a seguir, você obterá um System.ObjectDisposedException quando executar o código referenciado pela propriedade `Resource.Argument`.

C#

```
private static Func<int, int> CreateBoundResource()
{
    using (var constant = new Resource()) // constant is captured by the
expression tree
    {
        Expression<Func<int, int>> expression = (b) => constant.Argument + b;
        var rVal = expression.Compile();
        return rVal;
    }
}
```

O delegado retornado desse método fechou sobre o objeto `constant`, que foi descartado. (Foi descartado, porque foi declarado em uma instrução `using`).

Agora, ao executar o delegado retornado por esse método, uma `ObjectDisposedException` será gerada no ponto de execução.

Parece estranho ter um erro de tempo de execução representando uma estrutura de tempo de compilação, mas esse é o mundo que você adentra quando trabalha com árvores de expressão.

Há inúmeras permutações desse problema, por isso é difícil oferecer orientação geral para evitá-lo. Tenha cuidado ao acessar variáveis locais ao definir expressões e tenha cuidado ao acessar o estado no objeto atual (representado por `this`) ao criar uma árvore de expressão retornada por meio de uma API pública.

O código em sua expressão pode fazer referência a métodos ou propriedades em outros assemblies. Esse assembly deve estar acessível quando a expressão for definida, quando ela for compilada e quando o delegado resultante for invocado. Você é recebido com um `ReferencedAssemblyNotFoundException` quando ele não está presente.

Resumo

Árvores de expressão que representam expressões lambda podem ser compiladas para criar um delegado que você pode executar. As árvores de expressão fornecem um mecanismo para executar o código representado por uma árvore de expressão.

A Árvore de Expressão representa o código que seria executado para qualquer construção que você criar. Desde que o ambiente em que você compila e execute o código corresponda ao ambiente em que você cria a expressão, tudo funciona conforme o esperado. Quando isso não acontece, os erros são previsíveis e são capturados em seus primeiros testes de qualquer código usando as árvores de expressão.

Interpretar expressões

Artigo • 27/03/2025

O exemplo de código a seguir demonstra como a árvore de expressão que representa a expressão lambda `num => num < 5` pode ser decomposta em suas partes.

C#

```
// Add the following using directive to your code file:
// using System.Linq.Expressions;

// Create an expression tree.
Expression<Func<int, bool>> exprTree = num => num < 5;

// Decompose the expression tree.
ParameterExpression param = (ParameterExpression)exprTree.Parameters[0];
BinaryExpression operation = (BinaryExpression)exprTree.Body;
ParameterExpression left = (ParameterExpression)operation.Left;
ConstantExpression right = (ConstantExpression)operation.Right;

Console.WriteLine($"Decomposed expression: {param.Name} => {left.Name}
{operation.NodeType} {right.Value}");

// This code produces the following output:

// Decomposed expression: num => num LessThan 5
```

Agora, vamos escrever um código para examinar a estrutura de uma árvore de expressão . Cada nó em uma árvore de expressão é um objeto de uma classe derivada de `Expression`.

Esse design faz com que visitar todos os nós em uma árvore de expressão seja uma operação recursiva relativamente simples. A estratégia geral é iniciar no nó raiz e determine que tipo de nó ele é.

Se o tipo de nó tiver filhos, visite os filhos recursivamente. Em cada nó filho, repita o processo usado no nó raiz: determine o tipo e, se o tipo tiver filhos, visite cada um dos filhos.

Examinar uma expressão sem filhos

Vamos começar visitando cada nó em uma árvore de expressão simples. Aqui está o código que cria uma expressão constante e, em seguida, examina suas propriedades:

C#

```
var constant = Expression.Constant(24, typeof(int));

Console.WriteLine($"This is a/an {constant.NodeType} expression type");
Console.WriteLine($"The type of the constant value is {constant.Type}");
Console.WriteLine($"The value of the constant value is {constant.Value}");
```

O código anterior imprime a seguinte saída:

Saída

```
This is a/an Constant expression type
The type of the constant value is System.Int32
The value of the constant value is 24
```

Agora, vamos escrever o código que examinaria essa expressão e escreverá algumas propriedades importantes sobre ela.

Expressão de adição

Vamos começar com o exemplo de adição da introdução a esta seção.

C#

```
Expression<Func<int>> sum = () => 1 + 2;
```

> ⓘ **Observação**
>
> Não use `var` para declarar essa árvore de expressão, pois o tipo natural do delegado é `Func<int>`, não `Expression<Func<int>>`.

O nó raiz é um `LambdaExpression`. Para obter o código interessante no lado direito do operador `=>`, você precisa encontrar um dos filhos de `LambdaExpression`. Você fará isso com todas as expressões nesta seção. O nó pai nos ajudar a localizar o tipo de retorno do `LambdaExpression`.

Para examinar cada nó nesta expressão, você precisa visitar de maneira recorrente muitos nós. Aqui está uma primeira implementação simples:

C#

```
Expression<Func<int, int, int>> addition = (a, b) => a + b;
```

```
Console.WriteLine($"This expression is a {addition.NodeType} expression
type");
Console.WriteLine($"The name of the lambda is {((addition.Name == null) ? "
<null>" : addition.Name)}");
Console.WriteLine($"The return type is {addition.ReturnType.ToString()}");
Console.WriteLine($"The expression has {addition.Parameters.Count}
arguments. They are:");
foreach (var argumentExpression in addition.Parameters)
{
    Console.WriteLine($"\tParameter Type:
{argumentExpression.Type.ToString()}, Name: {argumentExpression.Name}");
}

var additionBody = (BinaryExpression)addition.Body;
Console.WriteLine($"The body is a {additionBody.NodeType} expression");
Console.WriteLine($"The left side is a {additionBody.Left.NodeType}
expression");
var left = (ParameterExpression)additionBody.Left;
Console.WriteLine($"\tParameter Type: {left.Type.ToString()}, Name:
{left.Name}");
Console.WriteLine($"The right side is a {additionBody.Right.NodeType}
expression");
var right = (ParameterExpression)additionBody.Right;
Console.WriteLine($"\tParameter Type: {right.Type.ToString()}, Name:
{right.Name}");
```

Este exemplo imprime a seguinte saída:

Saída

```
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
```

Você percebe muita repetição no exemplo de código anterior. Vamos limpar tudo isso e criar um visitante de nós de expressão com uma finalidade mais geral. Isso exigirá que escrevamos um algoritmo recursivo. Qualquer nó poderia ser de um tipo que pode ter filhos. Qualquer nó que tem filhos exige que nós visitemos esses filhos e determinemos o que é esse nó. Aqui está a versão otimizada que utiliza a recursão para executar as operações de adição:

C#

```
using System.Linq.Expressions;

namespace Visitors;
// Base Visitor class:
public abstract class Visitor
{
    private readonly Expression node;

    protected Visitor(Expression node) => this.node = node;

    public abstract void Visit(string prefix);

    public ExpressionType NodeType => node.NodeType;
    public static Visitor CreateFromExpression(Expression node) =>
        node.NodeType switch
        {
            ExpressionType.Constant => new
ConstantVisitor((ConstantExpression)node),
            ExpressionType.Lambda => new
LambdaVisitor((LambdaExpression)node),
            ExpressionType.Parameter => new
ParameterVisitor((ParameterExpression)node),
            ExpressionType.Add => new BinaryVisitor((BinaryExpression)node),
            _ => throw new NotImplementedException($"Node not processed yet:
{node.NodeType}"),
        };
}

// Lambda Visitor
public class LambdaVisitor : Visitor
{
    private readonly LambdaExpression node;
    public LambdaVisitor(LambdaExpression node) : base(node) => this.node =
node;

    public override void Visit(string prefix)
    {
        Console.WriteLine($"{prefix}This expression is a {NodeType}
expression type");
        Console.WriteLine($"{prefix}The name of the lambda is {((node.Name
== null) ? "<null>" : node.Name)}");
        Console.WriteLine($"{prefix}The return type is {node.ReturnType}");
        Console.WriteLine($"{prefix}The expression has
{node.Parameters.Count} argument(s). They are:");
        // Visit each parameter:
        foreach (var argumentExpression in node.Parameters)
        {
            var argumentVisitor = CreateFromExpression(argumentExpression);
            argumentVisitor.Visit(prefix + "\t");
        }
        Console.WriteLine($"{prefix}The expression body is:");
        // Visit the body:
        var bodyVisitor = CreateFromExpression(node.Body);
        bodyVisitor.Visit(prefix + "\t");
```

```
    }
}

// Binary Expression Visitor:
public class BinaryVisitor : Visitor
{
    private readonly BinaryExpression node;
    public BinaryVisitor(BinaryExpression node) : base(node) => this.node =
node;

    public override void Visit(string prefix)
    {
        Console.WriteLine($"{prefix}This binary expression is a {NodeType}
expression");
        var left = CreateFromExpression(node.Left);
        Console.WriteLine($"{prefix}The Left argument is:");
        left.Visit(prefix + "\t");
        var right = CreateFromExpression(node.Right);
        Console.WriteLine($"{prefix}The Right argument is:");
        right.Visit(prefix + "\t");
    }
}

// Parameter visitor:
public class ParameterVisitor : Visitor
{
    private readonly ParameterExpression node;
    public ParameterVisitor(ParameterExpression node) : base(node)
    {
        this.node = node;
    }

    public override void Visit(string prefix)
    {
        Console.WriteLine($"{prefix}This is an {NodeType} expression type");
        Console.WriteLine($"{prefix}Type: {node.Type}, Name: {node.Name},
ByRef: {node.IsByRef}");
    }
}

// Constant visitor:
public class ConstantVisitor : Visitor
{
    private readonly ConstantExpression node;
    public ConstantVisitor(ConstantExpression node) : base(node) =>
this.node = node;

    public override void Visit(string prefix)
    {
        Console.WriteLine($"{prefix}This is an {NodeType} expression type");
        Console.WriteLine($"{prefix}The type of the constant value is
{node.Type}");
        Console.WriteLine($"{prefix}The value of the constant value is
{node.Value}");
```

```
    }
}
```

Esse algoritmo é a base de um algoritmo que visita qualquer `LambdaExpression` arbitrário. O código que você criou procura apenas uma pequena amostra dos possíveis conjuntos de nós da árvore de expressão que pode encontrar. No entanto, você ainda pode aprender bastante com o que ele produz. (O caso padrão no método `Visitor.CreateFromExpression` imprime uma mensagem no console de erro quando um novo tipo de nó é encontrado. Dessa forma, você sabe adicionar um novo tipo de expressão.)

Ao executar esse visitante na expressão de adição anterior, você obtém a seguinte saída:

Saída

```
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
```

Agora que criou uma implementação de visitante mais geral, você pode visitar e processar muitos tipos diferentes de expressões.

Expressão de adição com mais operandos

Vamos testar um exemplo mais complicado, mas ainda limitar os tipos de nó somente à adição:

C#

```
Expression<Func<int>> sum = () => 1 + 2 + 3 + 4;
```

Antes de executar esses exemplos no algoritmo de visitante, tente um exercício de pensamento para descobrir qual seria a saída. Lembre-se de que o operador `+` é um operador binário : ele deve ter dois filhos, representando os operandos esquerdo e direito. Há várias maneiras possíveis de construir uma árvore que podem estar corretas:

C#

```
Expression<Func<int>> sum1 = () => 1 + (2 + (3 + 4));
Expression<Func<int>> sum2 = () => ((1 + 2) + 3) + 4;

Expression<Func<int>> sum3 = () => (1 + 2) + (3 + 4);
Expression<Func<int>> sum4 = () => 1 + ((2 + 3) + 4);
Expression<Func<int>> sum5 = () => (1 + (2 + 3)) + 4;
```

Você pode ver a separação em duas respostas possíveis para destacar a mais promissora. A primeira representa expressões *associativas à direita*. A segunda representa expressões *associativas à esquerda*. A vantagem desses dois formatos é que o formato é dimensionado para qualquer número arbitrário de expressões de adição.

Se você executar essa expressão por meio do visitante, verá essa saída, verificando se a expressão de adição simples é *associativa à esquerda*.

Para executar este exemplo e ver a árvore de expressão completa, você faz uma alteração na árvore de expressão de origem. Quando a árvore de expressão contém todas as constantes, a árvore resultante simplesmente contém o valor constante de `10`. O compilador executa toda a adição e reduz a expressão à sua forma mais simples. Basta adicionar uma variável na expressão é suficiente para ver a árvore original:

C#

```
Expression<Func<int, int>> sum = (a) => 1 + a + 3 + 4;
```

Crie um visitante para essa soma e execute o visitante. Você verá esta saída:

Saída

```
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
```

Você pode executar qualquer uma das outras amostras por meio do código de visitante e ver qual árvore ela representa. Aqui está um exemplo da expressão `sum3` anterior (com um parâmetro adicional para impedir que o compilador compute a constante):

C#

```
Expression<Func<int, int, int>> sum3 = (a, b) => (1 + a) + (3 + b);
```

Esta é a saída do visitante:

Saída

```
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
```

Observe que os parênteses não fazem parte da saída. Não há nenhum nó na árvore de expressão que representa os parênteses na expressão de entrada. A estrutura da árvore de expressão contém todas as informações necessárias para comunicar a precedência.

Estendendo este exemplo

O exemplo lida apenas com as árvores de expressão mais rudimentares. O código que você viu nesta seção trata apenas inteiros constantes e o operador de `+` binário. Como um exemplo final, vamos atualizar o visitante para lidar com uma expressão mais complicada. Vamos fazer com que funcione para a seguinte expressão fatorial:

C#

```
Expression<Func<int, int>> factorial = (n) =>
    n == 0 ?
    1 :
    Enumerable.Range(1, n).Aggregate((product, factor) => product * factor);
```

Este código representa uma possível implementação da função *fatorial* matemática. A maneira como você escreveu esse código destaca duas limitações da criação de árvores de expressão atribuindo expressões lambda a Expressões. Primeiro, lambdas de instrução não são permitidos. Isso significa que você não pode usar loops, blocos, instruções if/else e outras estruturas de controle comuns em C#. Você está limitado a usar expressões. Em segundo lugar, você não pode chamar recursivamente a mesma expressão. Você poderia se ela já fosse um delegado, mas não pode chamá-la em sua forma de árvore de expressão. Na seção criando árvores de expressão, você aprenderá técnicas para superar essas limitações.

Nesta expressão, você encontrará todos estes tipos de nós:

1. Igual (expressão binária)
2. Multiplicar (expressão binária)
3. Condicional (a expressão `? :`)
4. Expressão de chamada de método (chamar `Range()` e `Aggregate()`)

Uma maneira de modificar o algoritmo do visitante é continuar executando-o e escrever o tipo de nó toda vez que você atingir sua cláusula `default`. Após algumas iterações, você verá cada um dos nós em potencial. Então, você tem tudo o que precisa. O resultado seria algo assim:

C#

```
public static Visitor CreateFromExpression(Expression node) =>
    node.NodeType switch
    {
        ExpressionType.Constant    => new
ConstantVisitor((ConstantExpression)node),
        ExpressionType.Lambda      => new
LambdaVisitor((LambdaExpression)node),
        ExpressionType.Parameter   => new
ParameterVisitor((ParameterExpression)node),
        ExpressionType.Add         => new
BinaryVisitor((BinaryExpression)node),
        ExpressionType.Equal       => new
BinaryVisitor((BinaryExpression)node),
        ExpressionType.Multiply    => new BinaryVisitor((BinaryExpression)
node),
        ExpressionType.Conditional => new
ConditionalVisitor((ConditionalExpression) node),
        ExpressionType.Call        => new
MethodCallVisitor((MethodCallExpression) node),
        _ => throw new NotImplementedException($"Node not processed yet:
{node.NodeType}"),
    };
```

Os `ConditionalVisitor` e `MethodCallVisitor` processam esses dois nós:

C#

```
public class ConditionalVisitor : Visitor
{
    private readonly ConditionalExpression node;
    public ConditionalVisitor(ConditionalExpression node) : base(node)
    {
        this.node = node;
    }

    public override void Visit(string prefix)
    {
        Console.WriteLine($"{prefix}This expression is a {NodeType}
expression");
        var testVisitor = Visitor.CreateFromExpression(node.Test);
        Console.WriteLine($"{prefix}The Test for this expression is:");
        testVisitor.Visit(prefix + "\t");
        var trueVisitor = Visitor.CreateFromExpression(node.IfTrue);
        Console.WriteLine($"{prefix}The True clause for this expression
```

```
is:");
        trueVisitor.Visit(prefix + "\t");
        var falseVisitor = Visitor.CreateFromExpression(node.IfFalse);
        Console.WriteLine($"{prefix}The False clause for this expression
is:");
        falseVisitor.Visit(prefix + "\t");
    }
}

public class MethodCallVisitor : Visitor
{
    private readonly MethodCallExpression node;
    public MethodCallVisitor(MethodCallExpression node) : base(node)
    {
        this.node = node;
    }

    public override void Visit(string prefix)
    {
        Console.WriteLine($"{prefix}This expression is a {NodeType}
expression");
        if (node.Object == null)
            Console.WriteLine($"{prefix}This is a static method call");
        else
        {
            Console.WriteLine($"{prefix}The receiver (this) is:");
            var receiverVisitor = Visitor.CreateFromExpression(node.Object);
            receiverVisitor.Visit(prefix + "\t");
        }

        var methodInfo = node.Method;
        Console.WriteLine($"{prefix}The method name is
{methodInfo.DeclaringType}.{methodInfo.Name}");
        // There is more here, like generic arguments, and so on.
        Console.WriteLine($"{prefix}The Arguments are:");
        foreach (var arg in node.Arguments)
        {
            var argVisitor = Visitor.CreateFromExpression(arg);
            argVisitor.Visit(prefix + "\t");
        }
    }
}
```

E a saída da árvore de expressão seria:

Saída

```
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
```

Expandir a biblioteca de amostras

Os exemplos nesta seção mostram as técnicas principais para visitar e examinar nós em uma árvore de expressão. Ela simplifica os tipos de nós que você encontrará para se concentrar nas tarefas principais de visitar e acessar nós em uma árvore de expressão.

Primeiro, os visitantes só lidam com constantes que são números inteiros. Valores constantes podem ser qualquer outro tipo numérico, e a linguagem C# dá suporte a conversões e promoções entre esses tipos. Uma versão mais robusta desse código espelharia todas essas funcionalidades.

Até o último exemplo reconhece um subconjunto dos tipos de nó possíveis. Você ainda pode alimentá-lo com muitas expressões que o fariam falhar. Uma implementação completa está incluída no .NET Standard com o nome ExpressionVisitor e pode lidar com todos os tipos de nó possíveis.

Por fim, a biblioteca usada neste artigo foi criada para demonstração e aprendizado. Ela não está otimizada. Isso torna as estruturas claras e destaca as técnicas usadas para visitar os nós e analisar o que está presente.

Mesmo com essas limitações, você deve estar bem no caminho para escrever algoritmos que leem e entendem árvores de expressão.

Criar árvores de expressão

19/06/2025

O compilador C# criou todas as árvores de expressão que você viu até agora. Você criou uma expressão lambda atribuída a uma variável digitada como um `Expression<Func<T>>` ou algum tipo semelhante. Para muitos cenários, você cria uma expressão na memória durante o tempo de execução.

Árvores de expressão são imutáveis. Ser imutável significa que você deve construir a árvore a partir das folhas até a raiz. As APIs que você usa para criar as árvores de expressão refletem esse fato: os métodos que você usa para criar um nó usam todos os filhos como argumentos. Vamos percorrer alguns exemplos para mostrar as técnicas.

Criar nós

Comece com a expressão de adição com a qual você tem trabalhado ao longo destas seções:

```C#
Expression<Func<int>> sum = () => 1 + 2;
```

Para construir essa árvore de expressão, você precisará criar os nós folha. Os nós folha são constantes. Use o Constant método para criar os nós:

```C#
var one = Expression.Constant(1, typeof(int));
var two = Expression.Constant(2, typeof(int));
```

Em seguida, crie a expressão de adição:

```C#
var addition = Expression.Add(one, two);
```

Depois de criar a expressão de adição, crie a expressão lambda:

```C#
var lambda = Expression.Lambda(addition);
```

Essa expressão lambda não contém argumentos. Mais adiante nesta seção, você verá como mapear argumentos para parâmetros e criar expressões mais complicadas.

Para expressões como essa, você pode combinar todas as chamadas em uma só instrução:

```
C#

var lambda2 = Expression.Lambda(
    Expression.Add(
        Expression.Constant(1, typeof(int)),
        Expression.Constant(2, typeof(int))
    )
);
```

Criar uma árvore

A seção anterior mostrava as noções básicas da criação de uma árvore de expressão na memória. Árvores mais complexas geralmente significam mais tipos de nós e um número maior de nós na árvore. Vamos percorrer mais um exemplo e mostrar dois outros tipos de nó que você normalmente criará quando criar árvores de expressão: os nós de argumento e os nós de chamada de método. Vamos criar uma árvore de expressão para criar esta expressão:

```
C#

Expression<Func<double, double, double>> distanceCalc =
    (x, y) => Math.Sqrt(x * x + y * y);
```

Você começa criando expressões de parâmetro para `x` e `y`.

```
C#

var xParameter = Expression.Parameter(typeof(double), "x");
var yParameter = Expression.Parameter(typeof(double), "y");
```

A criação das expressões de multiplicação e adição segue o padrão que você já viu:

```
C#

var xSquared = Expression.Multiply(xParameter, xParameter);
var ySquared = Expression.Multiply(yParameter, yParameter);
var sum = Expression.Add(xSquared, ySquared);
```

Em seguida, você precisa criar uma expressão de chamada de método para a chamada para `Math.Sqrt`.

```C#
var sqrtMethod = typeof(Math).GetMethod("Sqrt", new[] { typeof(double) }) ?? throw
new InvalidOperationException("Math.Sqrt not found!");
var distance = Expression.Call(sqrtMethod, sum);
```

A `GetMethod` chamada poderá ser retornada `null` se o método não for encontrado. Provavelmente isso ocorre porque você errou a ortografia do nome do método. Caso contrário, isso pode significar que o assembly necessário não está carregado. Por fim, você coloca a chamada de método em uma expressão lambda e define os argumentos para a expressão lambda:

```C#
var distanceLambda = Expression.Lambda(
    distance,
    xParameter,
    yParameter);
```

Neste exemplo mais complicado, você verá mais algumas técnicas que geralmente precisam para criar árvores de expressão.

Primeiro, você precisa criar os objetos que representam parâmetros ou variáveis locais antes de usá-los. Depois de criar esses objetos, você pode usá-los na árvore de expressão onde precisar.

Em segundo lugar, você precisa usar um subconjunto das APIs de Reflexão para criar um System.Reflection.MethodInfo objeto para que você possa criar uma árvore de expressão para acessar esse método. Você deve limitar-se ao subconjunto das APIs de Reflexão que estão disponíveis na plataforma .NET Core. Novamente, essas técnicas se estendem a outras árvores de expressão.

Criar código em profundidade

Você não está limitado no que pode criar usando essas APIs. No entanto, quanto mais complicada a árvore de expressão que você deseja construir, mais difícil é gerenciar e ler o código.

Vamos criar uma árvore de expressão equivalente a este código:

```C#
Func<int, int> factorialFunc = (n) =>
{
    var res = 1;
```

```
    while (n > 1)
    {
        res = res * n;
        n--;
    }
    return res;
};
```

O código anterior não constrói a árvore de expressão, mas simplesmente o delegado. Usando a classe `Expression`, não é possível criar lambdas de instrução. Este é o código necessário para criar a mesma funcionalidade. Não há uma API para criar um `while` loop, em vez disso, você precisa criar um loop que contenha um teste condicional e um destino de rótulo para sair do loop.

C#

```
var nArgument = Expression.Parameter(typeof(int), "n");
var result = Expression.Variable(typeof(int), "result");

// Creating a label that represents the return value
LabelTarget label = Expression.Label(typeof(int));

var initializeResult = Expression.Assign(result, Expression.Constant(1));

// This is the inner block that performs the multiplication,
// and decrements the value of 'n'
var block = Expression.Block(
    Expression.Assign(result,
        Expression.Multiply(result, nArgument)),
    Expression.PostDecrementAssign(nArgument)
);

// Creating a method body.
BlockExpression body = Expression.Block(
    new[] { result },
    initializeResult,
    Expression.Loop(
        Expression.IfThenElse(
            Expression.GreaterThan(nArgument, Expression.Constant(1)),
            block,
            Expression.Break(label, result)
        ),
        label
    )
);
```

O código para criar a árvore de expressão para a função fatorial é um pouco mais longo, mais complicado e está cheio de rótulos e instruções de quebra e outros elementos que você gostaria de evitar em nossas tarefas de codificação diárias.

Para esta seção, você escreveu código para visitar todos os nós nesta árvore de expressão e registrar informações sobre os nós que são gerados neste exemplo. Você pode exibir ou baixar o código de exemplo no repositório GitHub do dotnet/docs. Experimente por si mesmo criando e executando os exemplos.

Mapear construções de código para expressões

O exemplo de código a seguir demonstra uma árvore de expressão que representa a expressão `num => num < 5` lambda usando a API.

C#

```
// Manually build the expression tree for
// the lambda expression num => num < 5.
ParameterExpression numParam = Expression.Parameter(typeof(int), "num");
ConstantExpression five = Expression.Constant(5, typeof(int));
BinaryExpression numLessThanFive = Expression.LessThan(numParam, five);
Expression<Func<int, bool>> lambda1 =
    Expression.Lambda<Func<int, bool>>(
        numLessThanFive,
        new ParameterExpression[] { numParam });
```

A API de árvores de expressão também dá suporte a atribuições e expressões de fluxo de controle, como loops, blocos condicionais e `try-catch` blocos. Usando a API, você pode criar árvores de expressão mais complexas do que aquelas que podem ser criadas a partir de expressões lambda pelo compilador C#. O exemplo a seguir demonstra como criar uma árvore de expressão que calcula o fatorial de um número.

C#

```
// Creating a parameter expression.
ParameterExpression value = Expression.Parameter(typeof(int), "value");

// Creating an expression to hold a local variable.
ParameterExpression result = Expression.Parameter(typeof(int), "result");

// Creating a label to jump to from a loop.
LabelTarget label = Expression.Label(typeof(int));

// Creating a method body.
BlockExpression block = Expression.Block(
    // Adding a local variable.
    new[] { result },
    // Assigning a constant to a local variable: result = 1
    Expression.Assign(result, Expression.Constant(1)),
        // Adding a loop.
        Expression.Loop(
           // Adding a conditional block into the loop.
```

```
            Expression.IfThenElse(
                // Condition: value > 1
                Expression.GreaterThan(value, Expression.Constant(1)),
                // If true: result *= value --
                Expression.MultiplyAssign(result,
                    Expression.PostDecrementAssign(value)),
                // If false, exit the loop and go to the label.
                Expression.Break(label, result)
            ),
        // Label to jump to.
        label
    )
);

// Compile and execute an expression tree.
int factorial = Expression.Lambda<Func<int, int>>(block, value).Compile()(5);

Console.WriteLine(factorial);
// Prints 120.
```

Para obter mais informações, consulte Gerando métodos dinâmicos com árvores de expressão no Visual Studio 2010, que também se aplica a versões posteriores do Visual Studio.

Mover árvores de expressão

19/06/2025

Neste artigo, você aprenderá a visitar cada nó em uma árvore de expressão ao criar uma cópia modificada dessa árvore de expressão. Você traduz árvores de expressão para entender os algoritmos e para que possam ser adaptados a outro ambiente. Você altera o algoritmo que foi criado. Você poderá adicionar registro em log, interceptar chamadas de método e monitorá-las ou realizar outras ações.

O código que você compila para mover uma árvore de expressão é uma extensão do que você já viu para visitar todos os nós em uma árvore. Quando você move uma árvore de expressão, visita todos os nós e ao visitá-los, cria a nova árvore. A nova árvore pode conter referências aos nós originais ou novos nós que você colocou na árvore.

Vamos acessar uma árvore de expressão e criar uma árvore com alguns nós de substituição. Neste exemplo, vamos substituir qualquer constante por uma constante 10 vezes maior. Caso contrário, deixe a árvore de expressão intacta. Em vez de ler o valor da constante e substituí-la por uma nova constante, você faz essa substituição substituindo o nó constante por um novo nó que executa a multiplicação.

Aqui, depois de encontrar um nó constante, você cria um novo nó de multiplicação cujos filhos são a constante original e a constante `10`:

C#

```
private static Expression ReplaceNodes(Expression original)
{
    if (original.NodeType == ExpressionType.Constant)
    {
        return Expression.Multiply(original, Expression.Constant(10));
    }
    else if (original.NodeType == ExpressionType.Add)
    {
        var binaryExpression = (BinaryExpression)original;
        return Expression.Add(
            ReplaceNodes(binaryExpression.Left),
            ReplaceNodes(binaryExpression.Right));
    }
    return original;
}
```

Crie uma árvore substituindo o nó original pelo substituto. Verifique as alterações compilando e executando a árvore substituída.

C#

```
var one = Expression.Constant(1, typeof(int));
var two = Expression.Constant(2, typeof(int));
var addition = Expression.Add(one, two);
var sum = ReplaceNodes(addition);
var executableFunc = Expression.Lambda(sum);

var func = (Func<int>)executableFunc.Compile();
var answer = func();
Console.WriteLine(answer);
```

A criação de uma nova árvore é uma combinação de visitar os nós na árvore existente e criar novos nós e inseri-los na árvore. O exemplo anterior mostra a importância das árvores de expressão serem imutáveis. Observe que a nova árvore criada no código anterior contém uma mistura de nós recém-criados e nós da árvore existente. Os nós podem ser usados em ambas as árvores porque os nós na árvore existente não podem ser modificados. Reutilização de nós resulta em eficiências significativas de memória. Os mesmos nós podem ser usados ao longo de uma árvore ou em várias árvores de expressão. Como os nós não podem ser modificados, o mesmo nó pode ser reutilizado sempre que necessário.

Percorrer e executar uma adição

Vamos verificar a nova árvore criando um segundo visitante que percorre a árvore de nós de adição e calcula o resultado. Faça algumas modificações no visitante que você viu até agora. Nesta nova versão, o visitante retorna a soma parcial da operação de adição até este ponto. Para uma expressão constante, é simplesmente o valor da expressão constante. Para uma expressão de adição, o resultado será a soma dos operandos esquerdos e direitos, uma vez que essas árvores forem percorridas.

C#

```
var one = Expression.Constant(1, typeof(int));
var two = Expression.Constant(2, typeof(int));
var three = Expression.Constant(3, typeof(int));
var four = Expression.Constant(4, typeof(int));
var addition = Expression.Add(one, two);
var add2 = Expression.Add(three, four);
var sum = Expression.Add(addition, add2);

// Declare the delegate, so you can call it
// from itself recursively:
Func<Expression, int> aggregate = null!;
// Aggregate, return constants, or the sum of the left and right operand.
// Major simplification: Assume every binary expression is an addition.
aggregate = (exp) =>
    exp.NodeType == ExpressionType.Constant ?
    (int)((ConstantExpression)exp).Value :
```

```
    aggregate(((BinaryExpression)exp).Left) +
aggregate(((BinaryExpression)exp).Right);

var theSum = aggregate(sum);
Console.WriteLine(theSum);
```

Há um pouco de código aqui, mas os conceitos são acessíveis. Esse código visita filhos em uma pesquisa de profundidade inicial. Ao encontrar um nó constante, o visitante retorna o valor da constante. Depois de visitar ambos os filhos, o visitante terá a soma calculada para essa subárvore. Agora o nó de adição poderá computar sua soma. Depois de todos os nós na árvore de expressão terem sido visitados, a soma foi computada. Você pode monitorar a execução rodando o exemplo no depurador e acompanhando o processo.

Vamos facilitar o rastreamento de como os nós são analisados e como a soma é computada atravessando a árvore. Aqui está uma versão atualizada do método Aggregate que inclui um pouco de informações de rastreamento:

C#

```
private static int Aggregate(Expression exp)
{
    if (exp.NodeType == ExpressionType.Constant)
    {
        var constantExp = (ConstantExpression)exp;
        Console.Error.WriteLine($"Found Constant: {constantExp.Value}");
        if (constantExp.Value is int value)
        {
            return value;
        }
        else
        {
            return 0;
        }
    }
    else if (exp.NodeType == ExpressionType.Add)
    {
        var addExp = (BinaryExpression)exp;
        Console.Error.WriteLine("Found Addition Expression");
        Console.Error.WriteLine("Computing Left node");
        var leftOperand = Aggregate(addExp.Left);
        Console.Error.WriteLine($"Left is: {leftOperand}");
        Console.Error.WriteLine("Computing Right node");
        var rightOperand = Aggregate(addExp.Right);
        Console.Error.WriteLine($"Right is: {rightOperand}");
        var sum = leftOperand + rightOperand;
        Console.Error.WriteLine($"Computed sum: {sum}");
        return sum;
    }
    else throw new NotSupportedException("Haven't written this yet");
}
```

Executar isto na expressão `sum` resulta na seguinte saída:

Saída

```
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
```

Rastreie a saída e acompanhe no código anterior. Você será capaz de entender como o código visita cada nó e calcula a soma, à medida que percorre a árvore e localiza a soma.

Agora, vamos examinar uma execução diferente, com a expressão fornecida por `sum1`:

C#

```
Expression<Func<int>> sum1 = () => 1 + (2 + (3 + 4));
```

Aqui está a saída da análise desta expressão:

Saída

```
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
```

Embora a resposta final seja a mesma, o percurso da árvore é diferente. Os nós são percorridos em uma ordem diferente, porque a árvore foi construída com operações diferentes ocorrendo primeiro.

Criar uma cópia modificada

Crie um novo projeto **de Aplicativo de Console** . Adicione uma `using` diretiva ao arquivo para o `System.Linq.Expressions` namespace. Adicione a `AndAlsoModifier` classe ao seu projeto.

C#

```
public class AndAlsoModifier : ExpressionVisitor
{
    public Expression Modify(Expression expression)
    {
        return Visit(expression);
    }

    protected override Expression VisitBinary(BinaryExpression b)
    {
        if (b.NodeType == ExpressionType.AndAlso)
        {
            Expression left = this.Visit(b.Left);
            Expression right = this.Visit(b.Right);

            // Make this binary expression an OrElse operation instead of an
AndAlso operation.
            return Expression.MakeBinary(ExpressionType.OrElse, left, right,
b.IsLiftedToNull, b.Method);
        }

        return base.VisitBinary(b);
    }
}
```

Essa classe herda a ExpressionVisitor classe e é especializada para modificar expressões que representam operações condicionais `AND` . Altera essas operações de uma condicional `AND` para uma condicional `OR`. A classe substitui o VisitBinary método do tipo base, pois expressões condicionais `AND` são representadas como expressões binárias. `VisitBinary` No método, se a expressão passada para ela representar uma operação condicional `AND`, o código construirá uma nova expressão que contenha o operador condicional `OR` em vez do operador condicional `AND`. Se a expressão passada para `VisitBinary` não representar uma operação condicional `AND`, o método reverterá para a implementação da classe base. Os métodos da classe base constroem nós semelhantes às árvores de expressão passadas, mas as subárvores dos nós são substituídas pelas árvores de expressão produzidas de maneira recorrente pelo visitante.

Adicione uma `using` diretiva ao arquivo para o `System.Linq.Expressions` namespace. Adicione código ao `Main` método no arquivo Program.cs para criar uma árvore de expressão e passá-la para o método que o modifica.

C#

```
Expression<Func<string, bool>> expr = name => name.Length > 10 &&
name.StartsWith("G");
Console.WriteLine(expr);

AndAlsoModifier treeModifier = new AndAlsoModifier();
Expression modifiedExpr = treeModifier.Modify((Expression)expr);

Console.WriteLine(modifiedExpr);

/*  This code produces the following output:

    name => ((name.Length > 10) && name.StartsWith("G"))
    name => ((name.Length > 10) || name.StartsWith("G"))
*/
```

O código cria uma expressão que contém uma operação condicional `AND` . Em seguida, ele cria uma instância da `AndAlsoModifier` classe e passa a expressão para o `Modify` método dessa classe. As árvores de expressão original e modificada são geradas para mostrar a alteração. Compile e execute o aplicativo.

Saiba Mais

Este exemplo mostra um pequeno subconjunto do código que você compilaria para percorrer e interpretar os algoritmos representados por uma árvore de expressão. Para obter informações sobre como criar uma biblioteca de finalidade geral que traduz árvores de

expressão em outra linguagem, leia esta série por Matt Warren. Ele entra em detalhes sobre como traduzir qualquer um dos códigos que você pode encontrar em uma árvore de expressão.

Espero que você tenha visto o verdadeiro potencial das árvores de expressão. Você examina um conjunto de códigos, faz todas as alterações desejadas nesse código e executa a versão alterada. Como as árvores de expressão são imutáveis, você cria novas árvores usando os componentes das árvores existentes. Reutilizar nós minimiza a quantidade de memória necessária para criar árvores de expressão modificadas.

Depurando árvores de expressão no Visual Studio

22/06/2025

Você pode analisar a estrutura e o conteúdo das árvores de expressão ao depurar seus aplicativos. Para obter uma visão geral rápida da estrutura da árvore de expressão, você pode usar a `DebugView` propriedade, que representa árvores de expressão usando uma sintaxe especial. `DebugView` está disponível apenas no modo de depuração.

Como `DebugView` é uma cadeia de caracteres, você pode usar o Visualizador de Texto interno para exibi-lo em várias linhas, selecionando **Visualizador de Texto** no ícone de lupa ao lado do `DebugView` rótulo.

Como alternativa, você pode instalar e usar um visualizador personalizado para árvores de expressão, como:

- Expressões legíveis (licença mit, disponível no Visual Studio Marketplace), renderiza a árvore de expressão como código C# com tema, com várias opções de renderização:

- O Visualizador de Árvore de Expressão (licença MIT) fornece uma exibição de árvore da árvore de expressão e seus nós individuais:

Abrir um visualizador para uma árvore de expressão

Selecione o ícone de lupa que aparece ao lado da árvore de expressão em **DataTips**, uma janela **Inspeção**, a janela **Autos** ou a janela **Locais**. Uma lista de visualizadores disponíveis é exibida:

Selecione o visualizador que você deseja usar.

Consulte também

- Depuração no Visual Studio
- Criar visualizadores personalizados
- DebugView sintaxe

Sintaxe de DebugView

18/06/2025

A propriedade **DebugView** (disponível somente durante a depuração) fornece uma renderização em forma de cadeia de caracteres das árvores de expressões. A maior parte da sintaxe é bastante simples de entender; os casos especiais são descritos nas seções a seguir.

Cada exemplo é seguido por um comentário em bloco, contendo o **DebugView**.

ParameterExpression

ParameterExpression os nomes de variáveis são exibidos com um `$` símbolo no início.

Se um parâmetro não tiver um nome, ele será atribuído a um nome gerado automaticamente, como `$var1` ou `$var2`.

C#

```
ParameterExpression numParam =  Expression.Parameter(typeof(int), "num");
/*
    $num
*/

ParameterExpression numParam =  Expression.Parameter(typeof(int));
/*
    $var1
*/
```

ConstantExpression

Para ConstantExpression objetos que representam valores inteiros, cadeias de caracteres e `null`, o valor da constante é exibido.

Para tipos numéricos que têm sufixos padrão como literais C#, o sufixo é adicionado ao valor. A tabela a seguir mostra os sufixos associados a vários tipos numéricos.

Expandir a tabela

Tipo	Palavra-chave	Sufixo
System.UInt32	de tonalidade	U
System.Int64	longo	L

Tipo	Palavra-chave	Sufixo
System.UInt64	ulong	UL
System.Double	duplo	D
System.Single	flutuar	F
System.Decimal	decimal	M

C#

```
int num = 10;
ConstantExpression expr = Expression.Constant(num);
/*
    10
*/

double num = 10;
ConstantExpression expr = Expression.Constant(num);
/*
    10D
*/
```

BlockExpression

Se o tipo de um BlockExpression objeto for diferente do tipo da última expressão no bloco, o tipo será exibido entre colchetes angulares (`<` e `>`). Caso contrário, o BlockExpression tipo do objeto não será exibido.

C#

```
BlockExpression block = Expression.Block(Expression.Constant("test"));
/*
    .Block() {
        "test"
    }
*/

BlockExpression block =  Expression.Block(typeof(Object),
Expression.Constant("test"));
/*
    .Block<System.Object>() {
        "test"
    }
*/
```

LambdaExpression

LambdaExpression os objetos são exibidos junto com seus tipos delegados.

Se uma expressão lambda não tiver um nome, será atribuído um nome gerado automaticamente, como `#Lambda1` ou `#Lambda2`.

C#

```
LambdaExpression lambda =  Expression.Lambda<Func<int>>(Expression.Constant(1));
/*
    .Lambda #Lambda1<System.Func'1[System.Int32]>() {
        1
    }
*/

LambdaExpression lambda =  Expression.Lambda<Func<int>>(Expression.Constant(1),
"SampleLambda", null);
/*
    .Lambda #SampleLambda<System.Func'1[System.Int32]>() {
        1
    }
*/
```

LabelExpression

Se você especificar um valor padrão para o LabelExpression objeto, esse valor será exibido antes do LabelTarget objeto.

O `.Label` token indica o início do rótulo. O token `.LabelTarget` indica o destino para o qual ir.

Se um rótulo não tiver um nome, ele será atribuído a um nome gerado automaticamente, como `#Label1` ou `#Label2`.

C#

```
LabelTarget target = Expression.Label(typeof(int), "SampleLabel");
BlockExpression block = Expression.Block(
    Expression.Goto(target, Expression.Constant(0)),
    Expression.Label(target, Expression.Constant(-1))
);
/*
    .Block() {
        .Goto SampleLabel { 0 };
        .Label
            -1
        .LabelTarget SampleLabel:
    }
```

```
*/

LabelTarget target = Expression.Label();
BlockExpression block = Expression.Block(
    Expression.Goto(target),
    Expression.Label(target)
);
/*
    .Block() {
        .Goto #Label1 { };
        .Label
        .LabelTarget #Label1:
    }
*/
```

Operadores verificados

Os operadores verificados são exibidos com o `#` símbolo na frente do operador. Por exemplo, o operador de adição verificado é exibido como `#+`.

C#

```
Expression expr = Expression.AddChecked( Expression.Constant(1),
Expression.Constant(2));
/*
    1 #+ 2
*/

Expression expr = Expression.ConvertChecked( Expression.Constant(10.0),
typeof(int));
/*
    #(System.Int32)10D
*/
```

Métodos System.Linq.Expressions.Expression.Add

20/06/2025

Este artigo fornece comentários complementares à documentação de referência para esta API.

O Add método retorna um BinaryExpression que tem a Method propriedade definida para o método de implementação. A propriedade Type é definida como o tipo do nó. Se o nó for levantado, as propriedades IsLifted e IsLiftedToNull serão ambas `true`. Caso contrário, elas serão `false`. A propriedade Conversion é `null`.

As informações a seguir descrevem o método de implementação, o tipo de nó e se um nó tem comparação de precisão.

Implementando método

As regras a seguir determinam o método de implementação selecionado para a operação:

- Se a Type propriedade de um `left` ou `right` representa um tipo definido pelo usuário que sobrecarrega o operador de adição, o MethodInfo que representa esse método é o método de implementação.
- Caso contrário, se `left`.Tipo e `right`.Tipo são tipos numéricos, o método de implementação é `null`.

Tipo de nó e versão com comparação de precisão x sem comparação de precisão

Se o método de implementação não for `null`:

- Se `left`.Type e `right`.Type forem atribuíveis aos tipos de argumentos correspondentes do método de implementação, o nó não terá comparação de precisão. O tipo do nó é o tipo de retorno do método de implementação.

- Se as duas condições a seguir forem atendidas, o nó terá comparação de precisão e o tipo do nó será o tipo anulável que corresponde ao tipo de retorno do método de implementação:
  - `left`.tipo e `right`.tipo são ambos tipos de valor dos quais pelo menos um é nulo, e os tipos não nulos correspondentes são iguais aos tipos de argumento correspondentes do método de implementação.

    - O tipo de retorno do método de implementação é um tipo de valor não anulável.

Se o método de implementação for `null`:

- Se `left`.Type e `right`.Type forem não anuláveis, o nó não terá comparação de precisão. O tipo do nó é o tipo de resultado do operador de adição pré-definido.
- Se `left`.Type e `right`.Type forem anuláveis, o nó terá comparação de precisão. O tipo do nó é o tipo anulável correspondente ao tipo de resultado do operador de adição pré-definido.

Classe System.Linq.Expressions.BinaryExpression

19/06/2025

Este artigo fornece comentários complementares à documentação de referência para esta API.

A BinaryExpression classe representa uma expressão que tem um operador binário.

As tabelas a seguir resumem os métodos de fábrica que podem ser usados para criar um BinaryExpression que tem um tipo de nó específico, representado pela propriedade NodeType. Cada tabela contém informações para uma classe específica de operações, como aritmética ou bit a bit.

Operações aritméticas binárias

Expandir a tabela

Tipo de nó	Método de Fábrica
Add	Add
AddChecked	AddChecked
Divide	Divide
Modulo	Modulo
Multiply	Multiply
MultiplyChecked	MultiplyChecked
Power	Power
Subtract	Subtract
SubtractChecked	SubtractChecked

Operações Bit a Bit

Expandir a tabela

Tipo de nó	Método de Fábrica
And	And

Tipo de nó	Método de Fábrica
Or	Or
ExclusiveOr	ExclusiveOr

Operações de turno

Expandir a tabela

Tipo de nó	Método de Fábrica
LeftShift	LeftShift
RightShift	RightShift

Operações boolianas condicionais

Expandir a tabela

Tipo de nó	Método de Fábrica
AndAlso	AndAlso
OrElse	OrElse

Operações de comparação

Expandir a tabela

Tipo de nó	Método de Fábrica
Equal	Equal
NotEqual	NotEqual
GreaterThanOrEqual	GreaterThanOrEqual
GreaterThan	GreaterThan
LessThan	LessThan
LessThanOrEqual	LessThanOrEqual

Operações de coalescing

Expandir a tabela

Tipo de nó	Método de Fábrica
Coalesce	Coalesce

Operações de indexação de matriz

Expandir a tabela

Tipo de nó	Método de Fábrica
ArrayIndex	ArrayIndex

Além disso, os MakeBinary métodos também podem ser usados para criar um BinaryExpression. Esses métodos de fábrica podem ser usados para criar um BinaryExpression de qualquer tipo de nó que representa uma operação binária. O parâmetro desses métodos, que é do tipo NodeType, especifica o tipo de nó desejado.

Visão geral da interoperabilidade

19/06/2025

A interoperabilidade permite que você preserve e aproveite os investimentos existentes em código não gerenciado. O código executado sob o controle do CLR (Common Language Runtime) é *o código gerenciado* e o código executado fora do CLR não é *gerenciado*. Componentes COM, COM+, C++, componentes ActiveX e API do Microsoft Windows são exemplos de código não gerenciado.

O .NET permite a interoperabilidade com código não gerenciado por meio de serviços de invocação de plataforma, namespace System.Runtime.InteropServices , interoperabilidade C++ e interoperabilidade COM (interoperabilidade COM).

Invocação de plataforma

A invocação de plataforma é um serviço que permite que o código gerenciado chame funções não gerenciadas implementadas em DLLs (bibliotecas de link dinâmico), como a API do Microsoft Windows. Ele localiza e invoca uma função exportada e gerencia seus argumentos (inteiros, cadeias de caracteres, matrizes, estruturas e assim por diante) através da fronteira de interoperação conforme necessário.

Para obter mais informações, consulte Consumindo Funções de DLL Não Gerenciadas e Como usar o invocador de plataforma para reproduzir um arquivo WAV.

> ⓘ **Observação**
>
> O CLR (**Common Language Runtime**) gerencia o acesso aos recursos do sistema. Chamar código não gerenciado que está fora do CLR ignora esse mecanismo de segurança e, portanto, apresenta um risco à segurança. Por exemplo, código não gerenciado pode chamar recursos em código não gerenciado diretamente, ignorando mecanismos de segurança CLR. Para obter mais informações, consulte **Segurança no .NET**.

Interoperabilidade C++

Você pode usar a interoperabilidade C++, também conhecida como IJW (It Just Works), para encapsular uma classe C++ nativa. A interoperabilidade do C++ permite que o código criado em C# ou em outro idioma do .NET acesse-o. Você escreve código C++ para encapsular um componente DLL ou COM nativo. Ao contrário de outras linguagens .NET, o Visual C++ tem suporte de interoperabilidade que permite código gerenciado e não gerenciado no mesmo

aplicativo e até mesmo no mesmo arquivo. Em seguida, você cria o código C++ usando a opção **do compilador /clr** para produzir um assembly gerenciado. Por fim, você adiciona uma referência ao assembly em seu projeto em C# e usa os objetos encapsulados da mesma forma que usaria outras classes gerenciadas.

Expondo componentes COM a C#

Você pode consumir um componente COM de um projeto em C#. As etapas gerais são as seguintes:

1. Localize um componente COM para usá-lo e registrá-lo. Use regsvr32.exe para registrar ou cancelar o registro de uma DLL COM.
2. Adicione ao projeto uma referência ao componente COM ou à biblioteca de tipos. Quando você adiciona a referência, o Visual Studio usa o Tlbimp.exe (Importador da Biblioteca de Tipos), que usa uma biblioteca de tipos como entrada, para gerar um assembly de interoperabilidade do .NET. O assembly, também chamado de RCW (Runtime Callable Wrapper), contém classes gerenciadas e interfaces que encapsulam as classes COM e as interfaces que estão na biblioteca de tipos. O Visual Studio adiciona ao projeto uma referência ao assembly gerado.
3. Crie uma instância de uma classe definida no RCW. A criação de uma instância dessa classe cria uma instância do objeto COM.
4. Use o objeto assim como você usa outros objetos gerenciados. Quando o objeto é recuperado pela coleta de lixo, a instância do objeto COM também é liberada da memória.

Para obter mais informações, consulte Expondo componentes COM ao .NET Framework.

Expondo C# ao COM

Os clientes COM podem consumir tipos do C# que foram expostos corretamente. As etapas básicas para expor tipos C# são as seguintes:

1. Adicione atributos de interoperabilidade no projeto C#. Você pode tornar um assembly COM visível modificando as propriedades do projeto em C#. Para obter mais informações, consulte Caixa de diálogo de informações do assembly.
2. Gere uma biblioteca de tipos COM e registre-a para uso em COM. Você pode modificar as propriedades do projeto C# para registrar automaticamente o assembly do C# para a interoperabilidade COM. O Visual Studio usa a Regasm.exe (ferramenta Assembly Registration), com a opção de linha de comando `/tlb`, que usa um assembly gerenciado como entrada para gerar uma biblioteca de tipos. Essa biblioteca de tipos descreve os

`public` tipos no assembly e adiciona entradas de registro para que os clientes COM possam criar classes gerenciadas.

Para obter mais informações, consulte Expondo componentes do .NET Framework ao COM e Classe COM de exemplo.

Consulte também

- Melhorando o desempenho da interoperabilidade
- Introdução à interoperabilidade entre COM e .NET
- Introdução à interoperabilidade COM no Visual Basic
- Marshaling entre código gerenciado e não gerenciado
- Interoperação com código não gerenciado
- Registrando assemblies usando COM

Exemplo de classe COM

Artigo • 10/05/2023

O código a seguir é um exemplo de uma classe que você poderia expor como um objeto COM. Depois de posicionar o código em um arquivo .cs adicionado a seu projeto, defina a propriedade **Registrar para interoperabilidade COM** como **True**. Para obter mais informações, consulte Como registrar um componente para interoperabilidade COM.

A exposição de objetos do C# para COM requer a declaração de uma interface de classe, de uma "interface de eventos" se necessário e da própria classe. Os membros de classe devem seguir estas regras para ficarem visíveis ao COM:

- A classe deve ser pública.
- As propriedades, os métodos e os eventos devem ser públicos.
- As propriedades e os métodos devem ser declarados na interface de classe.
- Os eventos devem ser declarados na interface de eventos.

Outros membros públicos da classe que não são declarados nessas interfaces não estarão visíveis para COM, mas eles ficarão visíveis para outros objetos .NET. Para expor propriedades e métodos ao COM, você deve declará-los na interface de classe e marcá-los com um atributo `DispId` e implementá-los na classe. A ordem de declaração dos membros na interface é a ordem usada para a vtable do COM. Para expor eventos de sua classe, você deve declará-los na interface de eventos e marcá-los com um atributo `DispId`. A classe não deve implementar essa interface.

A classe implementa a interface de classe. Ela pode implementar mais de uma interface, mas a primeira implementação é a interface de classe padrão. Implemente os métodos e propriedades expostos ao COM aqui. Ela deve ser marcada como pública e deve corresponder às declarações na interface de classe. Além disso, declare aqui os eventos acionados pela classe. Ela deve ser pública e deve corresponder às declarações na interface de eventos.

Exemplo

C#

```
using System.Runtime.InteropServices;

namespace project_name
{
    [Guid("EAA4976A-45C3-4BC5-BC0B-E474F4C3C83F")]
```

```
    public interface ComClass1Interface
    {
    }

    [Guid("7BD20046-DF8C-44A6-8F6B-687FAA26FA71"),
        InterfaceType(ComInterfaceType.InterfaceIsIDispatch)]
    public interface ComClass1Events
    {
    }

    [Guid("0D53A3E8-E51A-49C7-944E-E72A2064F938"),
        ClassInterface(ClassInterfaceType.None),
        ComSourceInterfaces(typeof(ComClass1Events))]
    public class ComClass1 : ComClass1Interface
    {
    }
}
```

Confira também

- Interoperabilidade
- Página de Build, Designer de Projeto (C#)

Collaborate with us on GitHub

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

.NET

.NET feedback

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

Open a documentation issue

Provide product feedback

Passo a passo: programação do Office em C#

Artigo • 09/03/2023

O C# oferece recursos que melhoram a programação do Microsoft Office. As funcionalidades úteis do C# incluem argumentos nomeados e opcionais e valores retornados do tipo `dynamic`. Na programação COM, você pode omitir a palavra-chave `ref` e obter acesso a propriedades indexadas.

Ambas as linguagens permitem incorporar as informações de tipo, que permitem a implantação de assemblies que interagem com componentes COM sem implantar assemblies de interoperabilidade primários (PIAs) no computador do usuário. Para obter mais informações, consulte Instruções passo a passo: Inserindo tipos de assemblies gerenciados.

Este passo a passo demonstra essas funcionalidades no contexto de programação do Office, mas muitos deles também são úteis na programação em geral. No passo a passo, você usa um aplicativo Suplemento do Excel para criar uma pasta de trabalho do Excel. Em seguida, você cria um documento do Word que contém um link para a pasta de trabalho. Por fim, você vê como habilitar e desabilitar a dependência de PIA.

> ⓘ **Importante**
>
> O VSTO se baseia no **.NET Framework**. Os suplementos COM também podem ser gravados com o .NET Framework. Os suplementos do Office não podem ser criados com o **.NET Core e o .NET 5+**, as versões mais recentes do .NET. Isso ocorre porque o .NET Core/.NET 5+ não pode trabalhar em conjunto com o .NET Framework no mesmo processo e pode levar a falhas de carga de suplemento. Você pode continuar a usar o .NET Framework para escrever suplementos VSTO e COM para o Office. A Microsoft não atualizará o VSTO ou a plataforma de suplemento COM para usar o .NET Core ou o .NET 5+. Você pode aproveitar o .NET Core e o .NET 5+, incluindo o ASP.NET Core, para criar o lado do servidor dos **Suplementos Web do Office**.

Pré-requisitos

Você deve ter o Microsoft Office Excel e o Microsoft Office Word instalados no computador para concluir esse passo a passo.

> **Observação**
>
> Seu computador pode mostrar diferentes nomes ou locais para alguns dos elementos de interface do usuário do Visual Studio nas instruções a seguir. A edição do Visual Studio que você possui e as configurações que você usa determinam esses elementos. Para obter mais informações, consulte **Personalizando o IDE**.

Configurar um aplicativo Suplemento do Excel

1. Inicie o Visual Studio.
2. No menu **Arquivo** , aponte para **Novo**e selecione **Projeto**.
3. No painel **Modelos Instalados**, expanda **C#**, expanda **Office** e, em seguida, selecione o ano da versão do produto do Office.
4. No painel **Modelos**, selecione **Suplemento do <Excel> versão**.
5. Observe a parte superior do painel **Modelos** para se certificar de que **.NET Framework 4** ou uma versão posterior, é exibido na caixa **Estrutura de Destino**.
6. Digite um nome para seu projeto na caixa **Nome**, se desejar.
7. Selecione **OK**.
8. O novo projeto aparece no **Gerenciador de Soluções**.

Adicionar referências

1. No **Gerenciador de Soluções**, clique com o botão direito do mouse no nome do projeto e, em seguida, selecione **Adicionar Referência**. A caixa de diálogo **Adicionar Referência** é exibida.
2. Na guia **Assemblies**, selecione **Microsoft.Office.Interop.Excel**, versão `<version>.0.0.0` (para uma chave para os números de versão de produto do Office, consulte Versões da Microsoft), na lista **Nome do componente** e mantenha a tecla CTRL pressionada e selecione **Microsoft.Office.Interop.Word**, `version <version>.0.0.0`. Se você não vir os assemblies, talvez seja necessário verificar se eles estão instalados (confira Como: Instalar assemblies de interoperabilidade primária do Office).
3. Selecione **OK**.

Adicionar instruções Imports necessárias ou diretivas de uso

No **Gerenciador de Soluções**, clique com o botão direito do mouse no arquivo **ThisAddIn.cs** e, em seguida, selecione **Exibir Código**. Adicione as seguintes diretivas `using` (C#) na parte superior do arquivo de código se elas ainda não estiverem presentes.

```C#
using System.Collections.Generic;
using Excel = Microsoft.Office.Interop.Excel;
using Word = Microsoft.Office.Interop.Word;
```

Criar uma lista de contas bancárias

No **Gerenciador de Soluções**, clique com o botão direito do mouse no nome do projeto e, em seguida, selecione **Adicionar** e **Classe**. Nomeie a classe Account.cs. Selecione **Adicionar**. Substitua a definição da classe `Account` pelo código a seguir. As definições de classe usam *propriedades autoimplementadas*.

```C#
class Account
{
    public int ID { get; set; }
    public double Balance { get; set; }
}
```

Para criar uma lista `bankAccounts` que contém duas contas, adicione o seguinte código ao método `ThisAddIn_Startup` em *ThisAddIn.cs*. As declarações de lista usam *inicializadores de coleção*.

```C#
var bankAccounts = new List<Account>
{
    new Account
    {
        ID = 345,
        Balance = 541.27
    },
    new Account
    {
        ID = 123,
        Balance = -127.44
    }
};
```

Exportar dados para o Excel

No mesmo arquivo, adicione o método a seguir para a classe `ThisAddIn`. O método configura uma planilha do Excel e exporta dados para ela.

C#

```
void DisplayInExcel(IEnumerable<Account> accounts,
           Action<Account, Excel.Range> DisplayFunc)
{
    var excelApp = this.Application;
    // Add a new Excel workbook.
    excelApp.Workbooks.Add();
    excelApp.Visible = true;
    excelApp.Range["A1"].Value = "ID";
    excelApp.Range["B1"].Value = "Balance";
    excelApp.Range["A2"].Select();

    foreach (var ac in accounts)
    {
        DisplayFunc(ac, excelApp.ActiveCell);
        excelApp.ActiveCell.Offset[1, 0].Select();
    }
    // Copy the results to the Clipboard.
    excelApp.Range["A1:B3"].Copy();
}
```

- O método Add tem um *parâmetro opcional* para especificar um modelo específico. Os parâmetros opcionais permitem omitir o argumento para esse parâmetro, se você deseja usar o valor padrão do parâmetro. Como o exemplo anterior não tem argumentos, `Add` usa o modelo padrão e cria uma pasta de trabalho. A instrução equivalente em versões anteriores do C# requer um argumento de espaço reservado: `excelApp.Workbooks.Add(Type.Missing)`. Para obter mais informações, consulte Argumentos nomeados e opcionais.
- As propriedades `Range` e `Offset` do objeto Range usam o recurso de *propriedades indexadas*. Este recurso permite consumir essas propriedades de tipos COM usando a sintaxe típica do C# a seguir. Propriedades indexadas também permitem que você use a propriedade `Value` do objeto `Range`, eliminando a necessidade de usar a propriedade `Value2`. A propriedade `Value` é indexada, mas o índice é opcional. Argumentos opcionais e propriedades indexadas trabalham juntos no exemplo a seguir.

C#

```
// Visual C# 2010 provides indexed properties for COM programming.
excelApp.Range["A1"].Value = "ID";
```

```
excelApp.ActiveCell.Offset[1, 0].Select();
```

Não é possível criar propriedades indexadas de sua preferência. O recurso dá suporte apenas ao consumo de propriedades indexadas existentes.

Adicione o seguinte código no final de `DisplayInExcel` para ajustar as larguras das colunas para adequar o conteúdo.

```
C#

excelApp.Columns[1].AutoFit();
excelApp.Columns[2].AutoFit();
```

Essas adições demonstram outro recurso no C#: tratar valores `Object` retornados de hosts COM como o Office, como se eles tivessem o tipo dinâmico. Os objetos COM são tratados como `dynamic` automaticamente quando os **Inserir Tipos de Interoperabilidade** têm seu valor padrão, `True`, ou, de modo equivalente, quando você faz referência ao assembly com a opção do compilador **EmbedInteropTypes**. Para obter mais informações sobre como inserir tipos de interoperabilidade, consulte os procedimentos "Localizar a referência de PIA" e "Restaurar a dependência de PIA" posteriormente neste artigo. Para obter mais informações sobre `dynamic`, consulte dynamic ou Usando o tipo dynamic.

Invocar DisplayInExcel

Adicione o código a seguir no final do método `ThisAddIn_StartUp`. A chamada para `DisplayInExcel` contém dois argumentos. O primeiro argumento é o nome da lista de contas a ser processada. O segundo argumento é uma expressão lambda de várias linhas que define como processar os dados. Os valores `ID` e `balance` de cada conta serão exibidos em células adjacentes e a linha será exibida em vermelho se o equilíbrio for menor do que zero. Para obter mais informações, consulte Expressões Lambda.

```
C#

DisplayInExcel(bankAccounts, (account, cell) =>
// This multiline lambda expression sets custom processing rules
// for the bankAccounts.
{
    cell.Value = account.ID;
    cell.Offset[0, 1].Value = account.Balance;
    if (account.Balance < 0)
    {
        cell.Interior.Color = 255;
        cell.Offset[0, 1].Interior.Color = 255;
```

```
    }
});
```

Para executar o programa, pressione F5. Uma planilha do Excel é exibida contendo os dados das contas.

Adicionar um documento do Word

Adicione o código a seguir ao final do método `ThisAddIn_StartUp` para criar um documento do Word que contém um link para a pasta de trabalho do Excel.

C#

```
var wordApp = new Word.Application();
wordApp.Visible = true;
wordApp.Documents.Add();
wordApp.Selection.PasteSpecial(Link: true, DisplayAsIcon: true);
```

Esse código demonstra vários dos recursos em C#: a capacidade de omitir a palavra-chave `ref` na programação COM, argumentos nomeados e argumentos opcionais. O método PasteSpecial tem sete parâmetros, todos os quais são parâmetros de referência opcionais. Argumentos opcionais e nomeados permitem designar os parâmetros que você deseja acessar pelo nome e enviar argumentos apenas para esses parâmetros. Neste exemplo, os argumentos indicam que deve ser criado um link para a pasta de trabalho na área de transferência (parâmetro `Link`), e que o link será exibido no documento do Word como um ícone (parâmetro `DisplayAsIcon`). O C# permite também que você omita a palavra-chave `ref` nesses argumentos.

Executar o aplicativo

Pressione F5 para executar o aplicativo. O Excel é iniciado e exibe uma tabela que contém as informações das duas contas em `bankAccounts`. Em seguida, um documento do Word é exibido contendo um link para a tabela do Excel.

Limpar o projeto concluído

No Visual Studio, clique em **Limpar Solução** no menu **Compilar**. Caso contrário, o suplemento será executado toda vez que você abrir o Excel no seu computador.

Localizar a referência de PIA

1. Execute o aplicativo novamente, mas não selecione **Limpar Solução**.
2. Selecione **Iniciar**. Localize **Microsoft Visual Studio <versão>>** e abra o prompt de comando do desenvolvedor.
3. Digite `ildasm` na janela Prompt de Comando do Desenvolvedor para Visual Studio e pressione Enter. A janela IL DASM é exibida.
4. No menu **Arquivo** na janela IL DASM, selecione **Arquivo**>**Abrir**. Clique duas vezes em **Visual Studio <versão>** e clique duas vezes em **Projetos**. Abra a pasta do seu projeto e procure na pasta bin/Debug por *nome do projeto*.dll. Clique duas vezes em *nome do projeto*.dll. Uma nova janela exibe os atributos do projeto, além das referências a outros módulos e assemblies. O assembly inclui os namespaces `Microsoft.Office.Interop.Excel` e `Microsoft.Office.Interop.Word`. Por padrão, no Visual Studio, o compilador importa os tipos que você precisa de um PIA referenciado para o seu assembly. Para obter mais informações, consulte Como exibir o conteúdo de um assembly.
5. Clique duas vezes no ícone **MANIFEST**. Uma janela será exibida contendo uma lista de assemblies que contêm itens referenciados pelo projeto. `Microsoft.Office.Interop.Excel` e `Microsoft.Office.Interop.Word` não estão na lista. Como você importou os tipos de que seu projeto precisa para o assembly, não é necessário instalar referências a um PIA. Importar os tipos no assembly facilita a implantação. Os PIAs não precisam estar presentes no computador do usuário. Um aplicativo não requer a implantação de uma versão específica de um PIA. Os aplicativos podem funcionar com várias versões do Office, desde que as APIs necessárias existam em todas as versões. Como a implantação de PIAs não é mais necessária, você pode criar um aplicativo em cenários avançados que funcione com várias versões do Office, incluindo versões anteriores. Seu código não pode usar nenhuma API que não esteja disponível na versão do Office com a qual você está trabalhando. Nem sempre fica claro se uma API específica estava disponível em uma versão anterior. Não é recomendável trabalhar com versões anteriores do Office.
6. Feche a janela do manifesto e a janela do assembly.

Restaurar a dependência de PIA

1. No **Gerenciador de Soluções**, selecione o botão **Mostrar Todos os Arquivos**. Expanda a pasta **Referências** e selecione **Microsoft.Office.Interop.Excel**. Pressione F4 para exibir a janela **Propriedades**.
2. Na janela **Propriedades**, altere a propriedade **Inserir Tipos de Interoperabilidade** de **True** para **False**.
3. Repita as etapas 1 e 2 deste procedimento para `Microsoft.Office.Interop.Word`.

4. Em C#, comente as duas chamadas para `Autofit` no final do método `DisplayInExcel`.
5. Pressione F5 para verificar se o projeto ainda é executado corretamente.
6. Repita as etapas de 1 a 3 do procedimento anterior para abrir a janela do assembly. Observe que `Microsoft.Office.Interop.Word` e `Microsoft.Office.Interop.Excel` não estão mais na lista de assemblies inseridos.
7. Clique duas vezes no ícone **MANIFEST** e role a lista de assemblies referenciados. Ambos `Microsoft.Office.Interop.Word` e `Microsoft.Office.Interop.Excel` estão na lista. Como o aplicativo faz referência aos PIAs do Excel e do Word e a propriedade **Inserir Tipos de Interoperabilidade** está definida como **False**, ambos os assemblies devem existir no computador do usuário final.
8. No Visual Studio, selecione **Limpar Solução** no menu **Compilar** para limpar o projeto concluído.

Confira também

- Propriedades autoimplementadas (C#)
- Inicializadores de objeto e coleção
- Argumentos nomeados e opcionais
- dinâmico
- Usando o tipo dynamic
- Expressões lambda [C#]
- Passo a passo: inserindo informações de tipo dos Microsoft Office Assemblies no Visual Studio
- Instruções passo a passo: Inserindo tipos de assemblies gerenciados
- Passo a passo: criando o primeiro suplemento do VSTO para Excel

Como usar invocação de plataforma para executar um arquivo WAV

Artigo • 07/04/2023

O exemplo de código C# a seguir ilustra como usar os serviços de invocação de plataforma para reproduzir um arquivo de som WAV no sistema operacional Windows.

Exemplo

Esse código de exemplo usa DllImportAttribute para importar o ponto de entrada de método `PlaySound` da `winmm.dll` como `Form1 PlaySound()`. O exemplo tem um Windows Form simples com um botão. Ao clicar no botão, abre-se uma caixa de diálogo padrão OpenFileDialog do Windows para que você possa abrir o arquivo para reprodução. Quando um arquivo wave é selecionado, ele é executado usando o método `PlaySound()` da biblioteca *winmm.dll*. Para obter mais informações sobre esse método, consulte Usando a função PlaySound com arquivos de áudio Waveform. Procure e selecione um arquivo que tenha uma extensão .wav e, em seguida, selecione **Abrir** para reproduzir o arquivo wave usando a invocação de plataforma. Uma caixa de texto exibe o caminho completo do arquivo selecionado.

C#

```
using System.Runtime.InteropServices;

namespace WinSound;

public partial class Form1 : Form
{
    private TextBox textBox1;
    private Button button1;

    public Form1()  // Constructor.
    {
        InitializeComponent();
    }

    [DllImport("winmm.DLL", EntryPoint = "PlaySound", SetLastError = true,
CharSet = CharSet.Unicode, ThrowOnUnmappableChar = true)]
    private static extern bool PlaySound(string szSound, System.IntPtr hMod,
PlaySoundFlags flags);

    [System.Flags]
    public enum PlaySoundFlags : int
    {
        SND_SYNC = 0x0000,
```

```
        SND_ASYNC = 0x0001,
        SND_NODEFAULT = 0x0002,
        SND_LOOP = 0x0008,
        SND_NOSTOP = 0x0010,
        SND_NOWAIT = 0x00002000,
        SND_FILENAME = 0x00020000,
        SND_RESOURCE = 0x00040004
    }

    private void button1_Click(object sender, System.EventArgs e)
    {
        var dialog1 = new OpenFileDialog();

        dialog1.Title = "Browse to find sound file to play";
        dialog1.InitialDirectory = @"c:\";
        //<Snippet5>
        dialog1.Filter = "Wav Files (*.wav)|*.wav";
        //</Snippet5>
        dialog1.FilterIndex = 2;
        dialog1.RestoreDirectory = true;

        if (dialog1.ShowDialog() == DialogResult.OK)
        {
            textBox1.Text = dialog1.FileName;
            PlaySound(dialog1.FileName, new System.IntPtr(),
PlaySoundFlags.SND_SYNC);
        }
    }

    private void Form1_Load(object sender, EventArgs e)
    {
        // Including this empty method in the sample because in the IDE,
        // when users click on the form, generates code that looks for a
default method
        // with this name. We add it here to prevent confusion for those
using the samples.
    }
}
```

A caixa de diálogo **Abrir Arquivos** é filtrada por meio das configurações de filtro para mostrar somente os arquivos que têm uma extensão .wav.

Compilando o código

Crie um novo projeto de aplicativos do Windows Forms em C# no Visual Studio e nomeie-o como **WinSound**. Copie o código anterior e cole-o sobre o conteúdo do arquivo *Form1.cs*. Copie o código a seguir e cole-o no arquivo *Form1.Designer.cs*, no método `InitializeComponent()`, após qualquer código existente.

```
C#
```

```
this.button1 = new System.Windows.Forms.Button();
this.textBox1 = new System.Windows.Forms.TextBox();
this.SuspendLayout();
//
// button1
//
this.button1.Location = new System.Drawing.Point(192, 40);
this.button1.Name = "button1";
this.button1.Size = new System.Drawing.Size(88, 24);
this.button1.TabIndex = 0;
this.button1.Text = "Browse";
this.button1.Click += new System.EventHandler(this.button1_Click);
//
// textBox1
//
this.textBox1.Location = new System.Drawing.Point(8, 40);
this.textBox1.Name = "textBox1";
this.textBox1.Size = new System.Drawing.Size(168, 20);
this.textBox1.TabIndex = 1;
this.textBox1.Text = "File path";
//
// Form1
//
this.AutoScaleDimensions = new System.Drawing.SizeF(5, 13);
this.ClientSize = new System.Drawing.Size(292, 266);
this.Controls.Add(this.textBox1);
this.Controls.Add(this.button1);
this.Name = "Form1";
this.Text = "Platform Invoke WinSound C#";
this.ResumeLayout(false);
this.PerformLayout();
```

Compile e execute o código.

Confira também

- Um olhar detalhado sobre invocação de plataforma
- Marshaling de dados com invocação de plataforma

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For

.NET

.NET feedback

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

Open a documentation issue

more information, see our contributor guide.

Provide product feedback

Como usar propriedades indexadas na programação para interoperabilidade COM

Artigo • 22/09/2023

As propriedades indexadas trabalham juntamente com outras funcionalidades no C#, como argumentos nomeados e opcionais, um novo tipo (dinâmico) e informações de tipo inseridas para melhorar a programação do Microsoft Office.

> ⓘ **Importante**
>
> **VSTO (Ferramentas do Visual Studio para Office)** depende do **.NET Framework**. Os suplementos COM também podem ser gravados com o .NET Framework. Os suplementos do Office não podem ser criados com o **.NET Core e o .NET 5+**, as versões mais recentes do .NET. Isso ocorre porque o .NET Core/.NET 5+ não pode trabalhar em conjunto com o .NET Framework no mesmo processo e pode levar a falhas de carga de suplemento. Você pode continuar a usar o .NET Framework para escrever suplementos VSTO e COM para o Office. A Microsoft não atualizará o VSTO ou a plataforma de suplemento COM para usar o .NET Core ou o .NET 5+. Você pode aproveitar o .NET Core e o .NET 5+, incluindo o ASP.NET Core, para criar o lado do servidor dos **Suplementos Web do Office**.

Nas versões anteriores do C#, os métodos são acessíveis como propriedades apenas se o método `get` não tem parâmetros e o método `set` tem apenas um parâmetro de valor. No entanto, nem todas as propriedades COM atendem a essas restrições. Por exemplo, a propriedade Range[] do Excel tem um acessador `get` que requer um parâmetro para o nome do intervalo. No passado, como não era possível acessar a propriedade `Range` diretamente, era necessário usar o método `get_Range` em vez disso, conforme mostrado no exemplo a seguir.

{language}

```
// Visual C# 2008 and earlier.
var excelApp = new Excel.Application();
// . . .
Excel.Range targetRange = excelApp.get_Range("A1", Type.Missing);
```

Em vez disso, as propriedades indexadas permitem escrever o seguinte:

{language}

```
// Visual C# 2010.
var excelApp = new Excel.Application();
// . . .
Excel.Range targetRange = excelApp.Range["A1"];
```

O exemplo anterior também usa o recurso argumentos opcionais, que permite que você omita `Type.Missing`.

As propriedades indexadas permitem escrever o código a seguir.

{language}

```
// Visual C# 2010.
targetRange.Value = "Name";
```

Não é possível criar propriedades indexadas de sua preferência. O recurso dá suporte apenas ao consumo de propriedades indexadas existentes.

Exemplo

O código a seguir mostra um exemplo completo. Para obter mais informações sobre como configurar um projeto que acessa a API do Office, consulte Como acessar objetos de interoperabilidade do Office usando recursos do Visual C#.

{language}

```
// You must add a reference to Microsoft.Office.Interop.Excel to run
// this example.
using System;
using Excel = Microsoft.Office.Interop.Excel;

namespace IndexedProperties
{
    class Program
    {
        static void Main(string[] args)
        {
            CSharp2010();
        }

        static void CSharp2010()
        {
            var excelApp = new Excel.Application();
            excelApp.Workbooks.Add();
            excelApp.Visible = true;
```

```
            Excel.Range targetRange = excelApp.Range["A1"];
            targetRange.Value = "Name";
        }

        static void CSharp2008()
        {
            var excelApp = new Excel.Application();
            excelApp.Workbooks.Add(Type.Missing);
            excelApp.Visible = true;

            Excel.Range targetRange = excelApp.get_Range("A1",
Type.Missing);
            targetRange.set_Value(Type.Missing, "Name");
            // Or
            //targetRange.Value2 = "Name";
        }
    }
}
```

Confira também

- Argumentos nomeados e opcionais
- dinâmico
- Usando o tipo dynamic

Collaborate with us on GitHub

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

.NET feedback

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

Open a documentation issue

Provide product feedback

Como acessar objetos de interoperabilidade do Office

Artigo • 09/03/2023

O C# tem recursos que simplificam o acesso a objetos de API do Office. Os novos recursos incluem argumentos nomeados e opcionais, um novo tipo chamado `dynamic` e a capacidade de passar argumentos para parâmetros de referência em métodos COM como se fossem parâmetros de valor.

Neste artigo, você usará os novos recursos para escrever código que cria e exibe uma planilha do Microsoft Office Excel. Escreva o código para adicionar um documento do Office Word que contenha um ícone que esteja vinculado à planilha do Excel.

Para concluir este passo a passo, você deve ter o Microsoft Office Excel 2007 e o Microsoft Office Word 2007 ou versões posteriores, instaladas no computador.

> **Observação**
>
> Seu computador pode mostrar diferentes nomes ou locais para alguns dos elementos de interface do usuário do Visual Studio nas instruções a seguir. A edição do Visual Studio que você possui e as configurações que você usa determinam esses elementos. Para obter mais informações, consulte **Personalizando o IDE**.

> **Importante**
>
> O VSTO se baseia no **.NET Framework**. Os suplementos COM também podem ser gravados com o .NET Framework. Os suplementos do Office não podem ser criados com o **.NET Core e o .NET 5+**, as versões mais recentes do .NET. Isso ocorre porque o .NET Core/.NET 5+ não pode trabalhar em conjunto com o .NET Framework no mesmo processo e pode levar a falhas de carga de suplemento. Você pode continuar a usar o .NET Framework para escrever suplementos VSTO e COM para o Office. A Microsoft não atualizará o VSTO ou a plataforma de suplemento COM para usar o .NET Core ou o .NET 5+. Você pode aproveitar o .NET Core e o .NET 5+, incluindo o ASP.NET Core, para criar o lado do servidor dos **Suplementos Web do Office**.

Para criar um novo aplicativo de console

1. Inicie o Visual Studio.
2. No menu **Arquivo** , aponte para **Novo**e selecione **Projeto**. A caixa de diálogo **Novo Projeto** aparecerá.
3. No painel **Modelos Instalados**, expanda **C#** e selecione **Windows**.
4. Observe a parte superior da caixa de diálogo **Novo Projeto** para selecionar **.NET Framework 4** (ou versão posterior) como uma estrutura de destino.
5. No painel **Modelos**, selecione **Aplicativo de Console**.
6. Digite um nome para o projeto no campo **Nome**.
7. Selecione **OK**.

O novo projeto aparece no **Gerenciador de Soluções**.

Para adicionar referências

1. No **Gerenciador de Soluções**, clique com o botão direito do mouse no nome do projeto e, em seguida, selecione **Adicionar Referência**. A caixa de diálogo **Adicionar Referência** é exibida.
2. Na página **Assemblies**, selecione **Microsoft.Office.Interop.Word** na lista **Nome do Componente** e, mantendo a tecla CTRL pressionada, selecione **Microsoft.Office.Interop.Excel**. Se você não vir os assemblies, talvez seja necessário instalá-los. Confira Como instalar assemblies de interoperabilidade primários do Office.
3. Selecione **OK**.

Para adicionar as diretivas using necessárias

No **Gerenciador de Soluções**, clique com o botão direito do mouse no arquivo *Program.cs* e, em seguida, selecione **Exibir Código**. Adicione as seguintes diretivas `using` na parte superior do arquivo de código:

C#

```
using Excel = Microsoft.Office.Interop.Excel;
using Word = Microsoft.Office.Interop.Word;
```

Para criar uma lista de contas bancárias

Cole a seguinte definição de classe em **Program.cs**, na classe `Program`.

C#

```
public class Account
{
    public int ID { get; set; }
    public double Balance { get; set; }
}
```

Adicione o seguinte código ao método `Main` para criar uma lista `bankAccounts` que contenha duas contas.

C#

```
// Create a list of accounts.
var bankAccounts = new List<Account> {
    new Account {
                  ID = 345678,
                  Balance = 541.27
                },
    new Account {
                  ID = 1230221,
                  Balance = -127.44
                }
};
```

Para declarar um método que exporta as informações de conta para o Excel

1. Adicione o método a seguir à classe `Program` para configurar uma planilha do Excel. O método Add tem um parâmetro opcional para especificar um modelo específico. Os parâmetros opcionais permitem omitir o argumento para esse parâmetro, se você deseja usar o valor padrão do parâmetro. Como você não forneceu um argumento, `Add` usará o modelo padrão e criará uma nova pasta de trabalho. A instrução equivalente em versões anteriores do C# requer um argumento de espaço reservado: `ExcelApp.Workbooks.Add(Type.Missing)`.

C#

```
static void DisplayInExcel(IEnumerable<Account> accounts)
{
    var excelApp = new Excel.Application();
    // Make the object visible.
    excelApp.Visible = true;

    // Create a new, empty workbook and add it to the collection returned
    // by property Workbooks. The new workbook becomes the active workbook.
    // Add has an optional parameter for specifying a particular template.
```

```
    // Because no argument is sent in this example, Add creates a new
workbook.
    excelApp.Workbooks.Add();

    // This example uses a single workSheet. The explicit type casting is
    // removed in a later procedure.
    Excel._Worksheet workSheet = (Excel.Worksheet)excelApp.ActiveSheet;
}
```

Adicione o código a seguir no final de `DisplayInExcel`. O código insere valores nas duas primeiras colunas da primeira linha da planilha.

C#

```
// Establish column headings in cells A1 and B1.
workSheet.Cells[1, "A"] = "ID Number";
workSheet.Cells[1, "B"] = "Current Balance";
```

Adicione o código a seguir no final de `DisplayInExcel`. O loop `foreach` coloca as informações da lista de contas nas duas primeiras colunas de sucessivas linhas da planilha.

C#

```
var row = 1;
foreach (var acct in accounts)
{
    row++;
    workSheet.Cells[row, "A"] = acct.ID;
    workSheet.Cells[row, "B"] = acct.Balance;
}
```

Adicione o seguinte código no final de `DisplayInExcel` para ajustar as larguras das colunas para adequar o conteúdo.

C#

```
workSheet.Columns[1].AutoFit();
workSheet.Columns[2].AutoFit();
```

As versões anteriores do C# exigem a conversão explícita para essas operações, porque `ExcelApp.Columns[1]` retorna um `Object`, e `AutoFit` é um método Range do Excel. As linhas a seguir mostram a conversão.

C#

```
((Excel.Range)workSheet.Columns[1]).AutoFit();
((Excel.Range)workSheet.Columns[2]).AutoFit();
```

O C# converte o `Object` retornado em `dynamic` automaticamente, se o assembly for referenciado pela opção do compilador **EmbedInteropTypes** ou, de maneira equivalente, se a propriedade **Inserir Tipos de Interoperabilidade** do Excel for true. True é o valor padrão para essa propriedade.

Para executar o projeto

Adicione a seguinte linha no final de `Main`.

```
C#

// Display the list in an Excel spreadsheet.
DisplayInExcel(bankAccounts);
```

Pressione CTRL+F5. Uma planilha do Excel é exibida contendo os dados das duas contas.

Para adicionar um documento do Word

O código a seguir abre um aplicativo do Word e cria um ícone que é vinculado à planilha do Excel. Cole o método `CreateIconInWordDoc`, fornecido posteriormente nesta etapa, na classe `Program`. O `CreateIconInWordDoc` usa argumentos nomeados e opcionais para reduzir a complexidade das chamadas de método a Add e PasteSpecial. Essas chamadas incorporam dois recursos que simplificam as chamadas para métodos COM que possuem parâmetros de referência. Primeiro, você pode enviar argumentos para os parâmetros de referência como se fossem parâmetros de valor. Ou seja, você pode enviar valores diretamente, sem criar uma variável para cada parâmetro de referência. O compilador gera variáveis temporárias para conter os valores de argumento e descarta as variáveis quando você retornar da chamada. Em segundo lugar, você pode omitir a palavra-chave `ref` na lista de argumentos.

O método `Add` tem quatro parâmetros de referência que são opcionais. Você poderá omitir argumentos para alguns ou todos os parâmetros se desejar usar os valores padrão.

O método `PasteSpecial` insere o conteúdo da área de transferência. O método tem sete parâmetros de referência que são opcionais. O código a seguir especifica argumentos

para dois deles: `Link`, para criar um link para a origem do conteúdo da área de transferência, e `DisplayAsIcon`, para exibir o link como um ícone. Você pode usar argumentos nomeados para esses dois argumentos e omitir os outros. Embora esses argumentos sejam parâmetros de referência, você não precisa usar a palavra-chave `ref` ou criar variáveis para os enviar como argumentos. Você pode enviar os valores diretamente.

C#

```
static void CreateIconInWordDoc()
{
    var wordApp = new Word.Application();
    wordApp.Visible = true;

    // The Add method has four reference parameters, all of which are
    // optional. Visual C# allows you to omit arguments for them if
    // the default values are what you want.
    wordApp.Documents.Add();

    // PasteSpecial has seven reference parameters, all of which are
    // optional. This example uses named arguments to specify values
    // for two of the parameters. Although these are reference
    // parameters, you do not need to use the ref keyword, or to create
    // variables to send in as arguments. You can send the values directly.
    wordApp.Selection.PasteSpecial( Link: true, DisplayAsIcon: true);
}
```

Adicione a instrução a seguir no final de `Main`.

C#

```
// Create a Word document that contains an icon that links to
// the spreadsheet.
CreateIconInWordDoc();
```

Adicione a instrução a seguir no final de `DisplayInExcel`. O método `Copy` adiciona a planilha na área de transferência.

C#

```
// Put the spreadsheet contents on the clipboard. The Copy method has one
// optional parameter for specifying a destination. Because no argument
// is sent, the destination is the Clipboard.
workSheet.Range["A1:B3"].Copy();
```

Pressione CTRL+F5. Um documento do Word é exibido contendo um ícone. Clique duas vezes no ícone para colocar a planilha no primeiro plano.

Para definir a propriedade Inserir Tipos Interop

Outras melhorias são possíveis quando você chama um tipo COM que não requer um assembly de interoperabilidade primário (PIA) no tempo de execução. A remoção da dependência nos PIAs resulta na independência de versão e em uma implantação mais fácil. Para obter mais informações sobre as vantagens da programação sem PIAs, consulte Passo a passo: inserindo tipos de assemblies gerenciados.

Além disso, a programação é mais fácil porque o tipo `dynamic` representa os tipos necessários e retornados declarados em métodos COM. Variáveis com o tipo `dynamic` não são avaliadas até o tempo de execução, o que elimina a necessidade de conversão explícita. Para obter mais informações, veja Usando o tipo dynamic.

A inserção de informações de tipo, em vez do uso de PIAs, é o comportamento padrão. Devido a esse padrão, vários dos exemplos anteriores são simplificados. Você não precisa de nenhuma conversão explícita. Por exemplo, a declaração de `worksheet` em `DisplayInExcel` é escrita como `Excel._Worksheet workSheet = excelApp.ActiveSheet`, em vez de `Excel._Worksheet workSheet = (Excel.Worksheet)excelApp.ActiveSheet`. As chamadas para `AutoFit` no mesmo método também exigem conversão explícita sem o padrão, pois `ExcelApp.Columns[1]` retorna um `Object`, e `AutoFit` é um método do Excel. O código a seguir mostra a conversão.

C#

```
((Excel.Range)workSheet.Columns[1]).AutoFit();
((Excel.Range)workSheet.Columns[2]).AutoFit();
```

Para alterar o padrão e usar PIAs em vez de inserir informações de tipo, expanda o nó Referências no Gerenciador de Soluções e, em seguida, selecione **Microsoft.Office.Interop.Excel** ou **Microsoft.Office.Interop.Word**. Se você não conseguir ver a janela **Propriedades**, pressione **F4**. Localize **Inserir Tipos Interop** na lista de propriedades e altere seu valor para **False**. De maneira equivalente, você pode compilar usando a opção do compilador **Referências**, em vez de **EmbedInteropTypes** em um prompt de comando.

Para adicionar formatação adicional à tabela

Substitua as duas chamadas para `AutoFit` em `DisplayInExcel` pela instrução a seguir.

C#

```
// Call to AutoFormat in Visual C# 2010.
workSheet.Range["A1", "B3"].AutoFormat(
    Excel.XlRangeAutoFormat.xlRangeAutoFormatClassic2);
```

O método AutoFormat tem sete parâmetros de valor, que são opcionais. Argumentos nomeados e opcionais permitem que você forneça argumentos para nenhum, alguns ou todos eles. Na instrução anterior, você fornece um argumento para apenas um dos parâmetros, `Format`. Como `Format` é o primeiro parâmetro na lista de parâmetros, você não precisará fornecer o nome do parâmetro. No entanto, poderá ser mais fácil entender a instrução se você incluir o nome do parâmetro, conforme mostrado no código a seguir.

C#

```
// Call to AutoFormat in Visual C# 2010.
workSheet.Range["A1", "B3"].AutoFormat(Format:
    Excel.XlRangeAutoFormat.xlRangeAutoFormatClassic2);
```

Pressione CTRL + F5 para ver o resultado. Você pode encontrar outros formatos na lista na enumeração XlRangeAutoFormat.

Exemplo

O código a seguir mostra um exemplo completo.

C#

```
using System.Collections.Generic;
using Excel = Microsoft.Office.Interop.Excel;
using Word = Microsoft.Office.Interop.Word;

namespace OfficeProgrammingWalkthruComplete
{
    class Walkthrough
    {
        static void Main(string[] args)
        {
            // Create a list of accounts.
            var bankAccounts = new List<Account>
            {
                new Account {
                              ID = 345678,
                              Balance = 541.27
                            },
                new Account {
                              ID = 1230221,
```

```
                                Balance = -127.44
                            }
            };

            // Display the list in an Excel spreadsheet.
            DisplayInExcel(bankAccounts);

            // Create a Word document that contains an icon that links to
            // the spreadsheet.
            CreateIconInWordDoc();
        }

        static void DisplayInExcel(IEnumerable<Account> accounts)
        {
            var excelApp = new Excel.Application();
            // Make the object visible.
            excelApp.Visible = true;

            // Create a new, empty workbook and add it to the collection
returned
            // by property Workbooks. The new workbook becomes the active
workbook.
            // Add has an optional parameter for specifying a particular
template.
            // Because no argument is sent in this example, Add creates a
new workbook.
            excelApp.Workbooks.Add();

            // This example uses a single workSheet.
            Excel._Worksheet workSheet = excelApp.ActiveSheet;

            // Earlier versions of C# require explicit casting.
            //Excel._Worksheet workSheet =
(Excel.Worksheet)excelApp.ActiveSheet;

            // Establish column headings in cells A1 and B1.
            workSheet.Cells[1, "A"] = "ID Number";
            workSheet.Cells[1, "B"] = "Current Balance";

            var row = 1;
            foreach (var acct in accounts)
            {
                row++;
                workSheet.Cells[row, "A"] = acct.ID;
                workSheet.Cells[row, "B"] = acct.Balance;
            }

            workSheet.Columns[1].AutoFit();
            workSheet.Columns[2].AutoFit();

            // Call to AutoFormat in Visual C#. This statement replaces the
            // two calls to AutoFit.
            workSheet.Range["A1", "B3"].AutoFormat(
                Excel.XlRangeAutoFormat.xlRangeAutoFormatClassic2);
```

```
            // Put the spreadsheet contents on the clipboard. The Copy
method has one
            // optional parameter for specifying a destination. Because no
argument
            // is sent, the destination is the Clipboard.
            workSheet.Range["A1:B3"].Copy();
        }

        static void CreateIconInWordDoc()
        {
            var wordApp = new Word.Application();
            wordApp.Visible = true;

            // The Add method has four reference parameters, all of which
are
            // optional. Visual C# allows you to omit arguments for them if
            // the default values are what you want.
            wordApp.Documents.Add();

            // PasteSpecial has seven reference parameters, all of which are
            // optional. This example uses named arguments to specify values
            // for two of the parameters. Although these are reference
            // parameters, you do not need to use the ref keyword, or to
create
            // variables to send in as arguments. You can send the values
directly.
            wordApp.Selection.PasteSpecial(Link: true, DisplayAsIcon: true);
        }
    }

    public class Account
    {
        public int ID { get; set; }
        public double Balance { get; set; }
    }
}
```

Confira também

- Type.Missing
- dinâmico
- Argumentos nomeados e opcionais
- Como usar argumentos nomeados e opcionais na programação do Office

Como usar argumentos nomeados e opcionais na programação do Office

Artigo • 07/06/2024

Os argumentos nomeados e opcionais aprimoram a conveniência, a flexibilidade e a legibilidade na programação em C#. Além disso, esses recursos facilitam bastante o acesso a interfaces COM, como as APIs de Automação do Microsoft Office.

> ⓘ **Importante**
>
> O Visual Studio Tools para Office (VSTO) depende do .NET Framework. Os suplementos COM também podem ser gravados com o .NET Framework. Os suplementos do Office não podem ser criados com o .NET Core e o .NET 5+, as versões mais recentes do .NET. Isso ocorre porque o .NET Core/.NET 5+ não pode trabalhar em conjunto com o .NET Framework no mesmo processo e pode levar a falhas de carga de suplemento. Você pode continuar a usar o .NET Framework para escrever suplementos VSTO e COM para o Office. A Microsoft não atualizará o VSTO ou a plataforma de suplemento COM para usar o .NET Core ou o .NET 5+. Você pode aproveitar o .NET Core e o .NET 5+, incluindo o ASP.NET Core, para criar o lado do servidor dos Suplementos Web do Office.

No exemplo a seguir, o método ConvertToTable tem 16 parâmetros que representam as características de uma tabela, como o número de colunas e linhas, formatação, bordas, fontes e cores. Todos os 16 parâmetros são opcionais, pois na maioria das vezes você não deseja especificar determinados valores para todos eles. No entanto, sem argumentos nomeados e opcionais, é preciso fornecer um valor ou um valor de espaço reservado. Com argumentos nomeados e opcionais, você especifica somente os valores para os parâmetros necessários para seu projeto.

Você deve ter o Microsoft Office Word instalado em seu computador para concluir esses procedimentos.

> ⓘ **Observação**
>
> Seu computador pode mostrar diferentes nomes ou locais para alguns dos elementos de interface do usuário do Visual Studio nas instruções a seguir. A edição do Visual Studio que você possui e as configurações que você usa

determinam esses elementos. Para obter mais informações, consulte Personalizando o IDE.

Crie um novo aplicativo de console

Inicie o Visual Studio. No menu **Arquivo** , aponte para **Novo**e selecione **Projeto**. No painel **Categorias de Modelos**, expanda **C#** e selecione **Windows**. Observe a parte superior do painel **Modelos** para se certificar de que **.NET Framework 4** é exibido na caixa **Estrutura de Destino**. No painel **Modelos**, selecione **Aplicativo de Console**. Digite um nome para o projeto no campo **Nome**. Selecione **OK**. O novo projeto aparece no **Gerenciador de Soluções**.

Adicionar uma referência

No **Gerenciador de Soluções**, clique com o botão direito do mouse no nome do projeto e, em seguida, selecione **Adicionar Referência**. A caixa de diálogo **Adicionar Referência** é exibida. Na página **.NET**, selecione **Microsoft.Office.Interop.Word** na lista **Nome do Componente**. Selecione **OK**.

Adicionar as diretivas using necessárias

No **Gerenciador de Soluções**, clique com o botão direito do mouse no arquivo *Program.cs* e, em seguida, selecione **Exibir Código**. Adicione as seguintes diretivas `using` na parte superior do arquivo de código:

C#

```
using Word = Microsoft.Office.Interop.Word;
```

Exibir texto em um documento do Word

Na classe `Program` em *Program.cs*, adicione o seguinte método para criar um aplicativo do Word e um documento do Word. O método Add tem quatro parâmetros opcionais. Este exemplo usa os valores padrão. Portanto, nenhum argumento é necessário na instrução de chamada.

C#

```
static void DisplayInWord()
{
    var wordApp = new Word.Application();
    wordApp.Visible = true;
    // docs is a collection of all the Document objects currently
    // open in Word.
    Word.Documents docs = wordApp.Documents;

    // Add a document to the collection and name it doc.
    Word.Document doc = docs.Add();
}
```

Adicione o código a seguir ao final do método para definir onde exibir o texto no documento e o texto a ser exibido:

C#

```
// Define a range, a contiguous area in the document, by specifying
// a starting and ending character position. Currently, the document
// is empty.
Word.Range range = doc.Range(0, 0);

// Use the InsertAfter method to insert a string at the end of the
// current range.
range.InsertAfter("Testing, testing, testing. . .");
```

Executar o aplicativo

Adicione a seguinte instrução a Main:

C#

```
DisplayInWord();
```

Pressione CTRL+F5 para executar o projeto. É exibido um documento do Word contendo o texto especificado.

Alterar o texto para uma tabela

Use o método `ConvertToTable` para colocar o texto em uma tabela. O método tem 16 parâmetros opcionais. O IntelliSense coloca os parâmetros opcionais entre colchetes, como mostrado na ilustração a seguir. Os valores padrão de `Type.Missing` são o nome simples para `System.Type.Missing`.

```
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])
```

Os argumentos nomeados e opcionais permitem que você especifique valores apenas para os parâmetros que deseja alterar. Adicione o seguinte código ao final do método `DisplayInWord` para criar uma tabela. O argumento especifica que as vírgulas na cadeia de caracteres de texto em `range` separam as células da tabela.

C#

```
// Convert to a simple table. The table will have a single row with
// three columns.
range.ConvertToTable(Separator: ",");
```

Pressione CTRL+F5 para executar o projeto.

Experimentar outros parâmetros

Altere a tabela para que ela tenha uma coluna e três linhas, substitua a última linha em `DisplayInWord` pela instrução a seguir e digite CTRL+F5.

C#

```
range.ConvertToTable(Separator: ",", AutoFit: true, NumColumns: 1);
```

Especifique um formato predefinido para a tabela, substitua a última linha em `DisplayInWord` pela instrução a seguir e digite CTRL+F5. O formato pode ser qualquer uma das constantes WdTableFormat.

C#

```
range.ConvertToTable(Separator: ",", AutoFit: true, NumColumns: 1,
    Format: Word.WdTableFormat.wdTableFormatElegant);
```

Exemplo

O código a seguir inclui o exemplo completo:

C#

```
using System;
using Word = Microsoft.Office.Interop.Word;

namespace OfficeHowTo
{
    class WordProgram
    {
        static void Main(string[] args)
        {
            DisplayInWord();
        }

        static void DisplayInWord()
        {
            var wordApp = new Word.Application();
            wordApp.Visible = true;
            // docs is a collection of all the Document objects currently
            // open in Word.
            Word.Documents docs = wordApp.Documents;

            // Add a document to the collection and name it doc.
            Word.Document doc = docs.Add();

            // Define a range, a contiguous area in the document, by
specifying
            // a starting and ending character position. Currently, the
document
            // is empty.
            Word.Range range = doc.Range(0, 0);

            // Use the InsertAfter method to insert a string at the end of
the
            // current range.
            range.InsertAfter("Testing, testing, testing. . .");

            // You can comment out any or all of the following statements to
            // see the effect of each one in the Word document.

            // Next, use the ConvertToTable method to put the text into a
table.
            // The method has 16 optional parameters. You only have to
specify
            // values for those you want to change.

            // Convert to a simple table. The table will have a single row
with
            // three columns.
            range.ConvertToTable(Separator: ",");

            // Change to a single column with three rows..
            range.ConvertToTable(Separator: ",", AutoFit: true, NumColumns:
1);
```

```
            // Format the table.
            range.ConvertToTable(Separator: ",", AutoFit: true, NumColumns:
1,
                Format: Word.WdTableFormat.wdTableFormatElegant);
        }
    }
}
```

Colaborar conosco no GitHub

A fonte deste conteúdo pode ser encontrada no GitHub, onde você também pode criar e revisar problemas e solicitações de pull. Para obter mais informações, confira o nosso guia para colaboradores.

Comentários do .NET

O .NET é um projeto código aberto. Selecione um link para fornecer comentários:

- Abrir um problema de documentação
- Fornecer comentários sobre o produto

Usar o tipo dinâmico

Artigo • 16/06/2023

O tipo `dynamic` é estático, mas um objeto do tipo `dynamic` ignora a verificação de tipo estático. Na maioria dos casos, ele funciona como se tivesse o tipo `object`. O compilador pressupõe que um elemento `dynamic` dá suporte a qualquer operação. Portanto, você não precisa determinar se o objeto obtém seu valor de uma API COM, de uma linguagem dinâmica como o IronPython, do HTML DOM (Modelo de Objeto do Documento), a reflexão ou de algum outro lugar no programa. No entanto, se o código não for válido, os erros aparecerão em tempo de execução.

Por exemplo, se método de instância `exampleMethod1` no código a seguir tiver apenas um parâmetro, o compilador reconhecerá que a primeira chamada para o método, `ec.exampleMethod1(10, 4)`, não será válido porque ele contém dois argumentos. Essa chamada causa um erro do compilador. O compilador não verifica a segunda chamada para o método, `dynamic_ec.exampleMethod1(10, 4)`, porque o tipo `dynamic_ec` é `dynamic`. Portanto, nenhum erro de compilador é relatado. No entanto, o erro não passa despercebido indefinidamente. Ele aparece em tempo de execução e causa uma exceção em tempo de execução.

C#

```
static void Main(string[] args)
{
    ExampleClass ec = new ExampleClass();
    // The following call to exampleMethod1 causes a compiler error
    // if exampleMethod1 has only one parameter. Uncomment the line
    // to see the error.
    //ec.exampleMethod1(10, 4);

    dynamic dynamic_ec = new ExampleClass();
    // The following line is not identified as an error by the
    // compiler, but it causes a run-time exception.
    dynamic_ec.exampleMethod1(10, 4);

    // The following calls also do not cause compiler errors, whether
    // appropriate methods exist or not.
    dynamic_ec.someMethod("some argument", 7, null);
    dynamic_ec.nonexistentMethod();
}
```

C#

```
class ExampleClass
{
```

```
    public ExampleClass() { }
    public ExampleClass(int v) { }

    public void exampleMethod1(int i) { }

    public void exampleMethod2(string str) { }
}
```

A função do compilador nesses exemplos é reunir informações sobre o que cada instrução está propondo fazer para o objeto ou expressão que tem o tipo `dynamic`. O runtime examina as informações armazenadas e qualquer instrução que não seja válida causa uma exceção em tempo de execução.

O resultado de operações mais dinâmicas é `dynamic`. Por exemplo, se você passar o ponteiro do mouse sobre o uso de `testSum` no exemplo a seguir, o IntelliSense exibirá o tipo **(variável local) testSum dinâmico**.

C#

```
dynamic d = 1;
var testSum = d + 3;
// Rest the mouse pointer over testSum in the following statement.
System.Console.WriteLine(testSum);
```

As operações em que o resultado não é `dynamic` incluem:

- Conversões de `dynamic` em outro tipo.
- Chamadas de construtor que incluem argumentos do tipo `dynamic`.

Por exemplo, o tipo de `testInstance` na seguinte declaração é `ExampleClass` e não `dynamic`:

C#

```
var testInstance = new ExampleClass(d);
```

Conversões

As conversões entre objetos dinâmicos e outros tipos são fáceis. As conversões permitem que o desenvolvedor alterne entre o comportamento dinâmico e o não dinâmico.

Você pode converter qualquer um para `dynamic` implicitamente, conforme mostrado nos exemplos a seguir.

```C#
dynamic d1 = 7;
dynamic d2 = "a string";
dynamic d3 = System.DateTime.Today;
dynamic d4 = System.Diagnostics.Process.GetProcesses();
```

Por outro lado, você pode aplicar dinamicamente qualquer conversão implícita a qualquer expressão do tipo `dynamic`.

```C#
int i = d1;
string str = d2;
DateTime dt = d3;
System.Diagnostics.Process[] procs = d4;
```

Resolução de sobrecarga com argumentos de tipo dynamic

A resolução de sobrecarga ocorre em tempo de execução em vez de em tempo de compilação se um ou mais dos argumentos em uma chamada de método tem o tipo `dynamic` ou se o receptor da chamada do método é do tipo `dynamic`. No exemplo a seguir, se o único método `exampleMethod2` acessível for definido para obter um argumento de cadeia de caracteres, enviar `d1` como o argumento não causará um erro de compilador, mas causará uma exceção de tempo de execução. A resolução de sobrecarga falha em tempo de execução porque o tipo de tempo de execução de `d1` é `int` e `exampleMethod2` requer uma cadeia de caracteres.

```C#
// Valid.
ec.exampleMethod2("a string");

// The following statement does not cause a compiler error, even though ec
is not
// dynamic. A run-time exception is raised because the run-time type of d1
is int.
ec.exampleMethod2(d1);
// The following statement does cause a compiler error.
//ec.exampleMethod2(7);
```

runtime de linguagem dinâmico

O DLR (tempo de execução de linguagem dinâmica) fornece a infraestrutura que dá suporte ao tipo `dynamic` em C# e também à implementação das linguagens de programação dinâmicas como IronPython e IronRuby. Para obter mais informações sobre o DLR, consulte Visão geral do Dynamic Language Runtime.

interoperabilidade COM

Muitos métodos COM permitem variação nos tipos de argumento e tipo de retorno, especificando os tipos como `object`. A interoperabilidade COM exigiu a conversão explícita dos valores para coordenar com variáveis fortemente tipadas no C#. Se você compilar usando a opção **EmbedInteropTypes** (opções do compilador C#), a introdução do tipo `dynamic` permitirá tratar as ocorrências de `object` em assinaturas COM, como se fossem do tipo `dynamic` e, portanto, evitar grande parte da conversão. Para obter mais informações sobre como usar o tipo `dynamic` com objetos COM, confira o artigo sobre Como acessar objetos de interoperabilidade do Office usando recursos C#.

Artigos relacionados

Título	Descrição
dinâmico	Descreve o uso da palavra-chave `dynamic`.
Visão geral do Dynamic Language Runtime	Fornece uma visão geral do DLR, que é um ambiente de tempo de execução que adiciona um conjunto de serviços para as linguagens dinâmicas para o CLR (Common Language Runtime).
Passo a passo: Criando e usando objetos dinâmicos	Fornece instruções passo a passo para criar um objeto dinâmico personalizado e para criar um projeto que acessa uma biblioteca `IronPython`.

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For

.NET

.NET feedback

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

Open a documentation issue

Provide product feedback

more information, see our contributor guide.

Passo a passo: criando e usando objetos dinâmicos em C#

19/06/2025

Objetos dinâmicos expõem membros como propriedades e métodos em tempo de execução, em vez de em tempo de compilação. Objetos dinâmicos permitem que você crie objetos para trabalhar com estruturas que não correspondem a um tipo ou formato estático. Por exemplo, você pode usar um objeto dinâmico para referenciar o DOM (Modelo de Objeto de Documento HTML), que pode conter qualquer combinação de elementos e atributos de marcação HTML válidos. Como cada documento HTML é exclusivo, os membros de um documento HTML específico são determinados em tempo de execução. Um método comum para fazer referência a um atributo de um elemento HTML é passar o nome do atributo para o `GetProperty` método do elemento. Para fazer referência ao `id` atributo do elemento `<div id="Div1">`HTML, primeiro você obtém uma referência ao `<div>` elemento e, em seguida, usa `divElement.GetProperty("id")`. Se você usar um objeto dinâmico, poderá referenciar o `id` atributo como `divElement.id`.

Objetos dinâmicos também fornecem acesso conveniente a linguagens dinâmicas como IronPython e IronRuby. Você pode usar um objeto dinâmico para se referir a um script dinâmico interpretado em tempo de execução.

Você faz referência a um objeto dinâmico usando associação tardia. Especifique o tipo de um objeto com associação tardia como `dynamic`. Para obter mais informações, confira dynamic.

Você pode criar objetos dinâmicos personalizados usando as classes no System.Dynamic namespace. Por exemplo, você pode criar um ExpandoObject e especificar os membros desse objeto em tempo de execução. Você também pode criar seu próprio tipo que herda a DynamicObject classe. Em seguida, você pode sobrescrever os membros da classe DynamicObject para fornecer funcionalidade dinâmica em tempo de execução.

Este artigo contém dois passo a passo independentes:

- Crie um objeto personalizado que exponha dinamicamente o conteúdo de um arquivo de texto como propriedades de um objeto.
- Crie um projeto que use uma `IronPython` biblioteca.

Pré-requisitos

- Visual Studio 2022 versão 17.3 ou uma versão posterior com o **pacote de desenvolvimento de desktop do .NET** instalado. O SDK do .NET 7 é incluído quando você

seleciona essa carga de trabalho.

> **Observação**
>
> Seu computador pode mostrar nomes ou locais diferentes para alguns dos elementos de interface do usuário do Visual Studio nas instruções a seguir. A edição do Visual Studio que você tem e as configurações que você usa determinam esses elementos. Para obter mais informações, consulte Personalizando o IDE.

- Para o segundo passo a passo, instale o IronPython para .NET. Vá para a página Baixar para obter a versão mais recente.

Criar um objeto dinâmico personalizado

O primeiro passo a passo define um objeto dinâmico personalizado que pesquisa o conteúdo de um arquivo de texto. Uma propriedade dinâmica especifica o texto a ser pesquisado. Por exemplo, se o código de chamada especificar `dynamicFile.Sample`, a classe dinâmica retornará uma lista genérica de cadeias de caracteres que contém todas as linhas do arquivo que começam com "Amostra". A pesquisa diferencia maiúsculas de minúsculas. A classe dinâmica também dá suporte a dois argumentos opcionais. O primeiro argumento é um valor de enumeração de opções de pesquisa que especifica que a classe dinâmica deve pesquisar correspondências no início da linha, no final da linha ou em qualquer ponto da linha. O segundo argumento especifica que a classe dinâmica deve cortar espaços à esquerda e à direita de cada linha antes da pesquisa. Por exemplo, se o código de chamada especificar `dynamicFile.Sample(StringSearchOption.Contains)`, a classe dinâmica procurará "Amostra" em qualquer lugar de uma linha. Se o código de chamada especificar `dynamicFile.Sample(StringSearchOption.StartsWith, false)`, a classe dinâmica procurará "Amostra" no início de cada linha e não removerá espaços à esquerda e à direita. O comportamento padrão da classe dinâmica é pesquisar uma correspondência no início de cada linha e remover espaços à esquerda e à direita.

Criar uma classe dinâmica personalizada

Inicie o Visual Studio. Selecione **Criar um novo projeto**. Na caixa de diálogo **Criar um novo projeto** , selecione C#, selecione **Aplicativo de Console** e, em seguida, selecione **Avançar**. Na caixa de diálogo **Configurar seu novo projeto** , insira `DynamicSample` o **nome do Projeto** e selecione **Avançar**. Na caixa de **diálogo Informações adicionais** , selecione **.NET 7.0 (Atual)** para a **Estrutura de Destino** e, em seguida, selecione **Criar**. No **Gerenciador de Soluções**, clique com o botão direito do mouse no projeto DynamicSample e selecione **Adicionar**>**Classe**. Na caixa **Nome** , digite `ReadOnlyFile` e selecione **Adicionar**. Na parte

superior do *arquivo ReadOnlyFile.cs* ou *ReadOnlyFile.vb*, adicione o código a seguir para importar os namespaces System.IO e System.Dynamic.

```
C#

using System.IO;
using System.Dynamic;
```

O objeto dinâmico personalizado usa uma enumeração para determinar os critérios de pesquisa. Antes da instrução de classe, adicione a seguinte definição de enum.

```
C#

public enum StringSearchOption
{
    StartsWith,
    Contains,
    EndsWith
}
```

Atualize a instrução de classe para herdar a `DynamicObject` classe, conforme mostrado no exemplo de código a seguir.

```
C#

class ReadOnlyFile : DynamicObject
```

Adicione o seguinte código à classe `ReadOnlyFile` para definir um campo privado para o caminho do arquivo e um construtor para a classe `ReadOnlyFile`.

```
C#

// Store the path to the file and the initial line count value.
private string p_filePath;

// Public constructor. Verify that file exists and store the path in
// the private variable.
public ReadOnlyFile(string filePath)
{
    if (!File.Exists(filePath))
    {
        throw new Exception("File path does not exist.");
    }

    p_filePath = filePath;
}
```

1. Adicione o método a seguir `GetPropertyValue` à `ReadOnlyFile` classe. O `GetPropertyValue` método usa, como entrada, critérios de pesquisa e retorna as linhas de um arquivo de texto que correspondem a esses critérios de pesquisa. Os métodos dinâmicos fornecidos pela `ReadOnlyFile` classe chamam o `GetPropertyValue` método para recuperar seus respectivos resultados.

C#

```
public List<string> GetPropertyValue(string propertyName,
                                     StringSearchOption StringSearchOption =
StringSearchOption.StartsWith,
                                     bool trimSpaces = true)
{
    StreamReader sr = null;
    List<string> results = new List<string>();
    string line = "";
    string testLine = "";

    try
    {
        sr = new StreamReader(p_filePath);

        while (!sr.EndOfStream)
        {
            line = sr.ReadLine();

            // Perform a case-insensitive search by using the specified search
options.
            testLine = line.ToUpper();
            if (trimSpaces) { testLine = testLine.Trim(); }

            switch (StringSearchOption)
            {
                case StringSearchOption.StartsWith:
                    if (testLine.StartsWith(propertyName.ToUpper())) {
results.Add(line); }
                    break;
                case StringSearchOption.Contains:
                    if (testLine.Contains(propertyName.ToUpper())) {
results.Add(line); }
                    break;
                case StringSearchOption.EndsWith:
                    if (testLine.EndsWith(propertyName.ToUpper())) {
results.Add(line); }
                    break;
            }
        }
    }
    catch
    {
        // Trap any exception that occurs in reading the file and return null.
        results = null;
```

```
    }
    finally
    {
        if (sr != null) {sr.Close();}
    }

    return results;
}
```

Após o `GetPropertyValue` método, adicione o código a seguir para substituir o TryGetMember método da DynamicObject classe. O TryGetMember método é chamado quando um membro de uma classe dinâmica é solicitado e nenhum argumento é especificado. O `binder` argumento contém informações sobre o membro referenciado e o `result` argumento faz referência ao resultado retornado para o membro especificado. O TryGetMember método retorna um valor booliano que retorna `true` se o membro solicitado existir; caso contrário, retorna `false`.

C#

```
// Implement the TryGetMember method of the DynamicObject class for dynamic member
calls.
public override bool TryGetMember(GetMemberBinder binder,
                                  out object result)
{
    result = GetPropertyValue(binder.Name);
    return result == null ? false : true;
}
```

Após o `TryGetMember` método, adicione o código a seguir para substituir o TryInvokeMember método da DynamicObject classe. O TryInvokeMember método é chamado quando um membro de uma classe dinâmica é solicitado com argumentos. O `binder` argumento contém informações sobre o membro referenciado e o `result` argumento faz referência ao resultado retornado para o membro especificado. O `args` argumento contém uma matriz dos argumentos que são passados para o membro. O TryInvokeMember método retorna um valor booliano que retorna `true` se o membro solicitado existir; caso contrário, retorna `false`.

A versão personalizada do `TryInvokeMember` método espera que o primeiro argumento seja um valor da `StringSearchOption` enumeração que você definiu em uma etapa anterior. O `TryInvokeMember` método espera que o segundo argumento seja um valor booliano. Se um ou ambos os argumentos forem valores válidos, eles serão passados para o `GetPropertyValue` método para recuperar os resultados.

C#

```
// Implement the TryInvokeMember method of the DynamicObject class for
// dynamic member calls that have arguments.
```

```
public override bool TryInvokeMember(InvokeMemberBinder binder,
                                     object[] args,
                                     out object result)
{
    StringSearchOption StringSearchOption = StringSearchOption.StartsWith;
    bool trimSpaces = true;

    try
    {
        if (args.Length > 0) { StringSearchOption = (StringSearchOption)args[0]; }
    }
    catch
    {
        throw new ArgumentException("StringSearchOption argument must be a
StringSearchOption enum value.");
    }

    try
    {
        if (args.Length > 1) { trimSpaces = (bool)args[1]; }
    }
    catch
    {
        throw new ArgumentException("trimSpaces argument must be a Boolean
value.");
    }

    result = GetPropertyValue(binder.Name, StringSearchOption, trimSpaces);

    return result == null ? false : true;
}
```

Salve e feche o arquivo.

Criar um arquivo de texto de exemplo

No **Gerenciador de Soluções**, clique com o botão direito do mouse no projeto DynamicSample e selecione **Adicionar**>**Novo Item**. No painel **Modelos Instalados** , selecione **Geral** e, em seguida, selecione o modelo **arquivo de texto** . Deixe o nome padrão de *TextFile1.txt* na caixa **Nome** e selecione **Adicionar**. Copie o texto a seguir para o arquivo *TextFile1.txt* .

text

```
List of customers and suppliers

Supplier: Lucerne Publishing (https://www.lucernepublishing.com/)
Customer: Preston, Chris
Customer: Hines, Patrick
Customer: Cameron, Maria
Supplier: Graphic Design Institute (https://www.graphicdesigninstitute.com/)
Supplier: Fabrikam, Inc. (https://www.fabrikam.com/)
```

```
Customer: Seubert, Roxanne
Supplier: Proseware, Inc. (http://www.proseware.com/)
Customer: Adolphi, Stephan
Customer: Koch, Paul
```

Salve e feche o arquivo.

Criar um aplicativo de exemplo que usa o objeto dinâmico personalizado

No **Gerenciador de Soluções**, clique duas vezes no arquivo *Program.cs* . Adicione o código a seguir ao `Main` procedimento para criar uma instância da `ReadOnlyFile` classe para o arquivo *TextFile1.txt* . O código usa associação tardia para chamar membros dinâmicos e recuperar linhas de texto que contêm a string "Cliente".

C#

```
dynamic rFile = new ReadOnlyFile(@"..\..\..\TextFile1.txt");
foreach (string line in rFile.Customer)
{
    Console.WriteLine(line);
}
Console.WriteLine("----------------------------");
foreach (string line in rFile.Customer(StringSearchOption.Contains, true))
{
    Console.WriteLine(line);
}
```

Salve o arquivo e pressione `Ctrl+F5 para compilar e executar o aplicativo.`

Chamar uma biblioteca de linguagem dinâmica

O passo a passo a seguir cria um projeto que acessa uma biblioteca escrita na linguagem dinâmica IronPython.

Para criar uma classe dinâmica personalizada

No Visual Studio, selecione **Arquivo**>**Novo**>**Projeto**. Na caixa de diálogo **Criar um novo projeto** , selecione C#, selecione **Aplicativo de Console** e, em seguida, selecione **Avançar**. Na caixa de diálogo **Configurar seu novo projeto** , insira `DynamicIronPythonSample` o **nome do Projeto** e selecione **Avançar**. Na caixa de **diálogo Informações adicionais** , selecione **.NET 7.0 (Atual)** para a **Estrutura de Destino** e, em seguida, selecione **Criar**. Instale o pacote NuGet IronPython . Edite o arquivo *de Program.cs* . Na parte superior do arquivo, adicione o código

a seguir para importar os namespaces `Microsoft.Scripting.Hosting` e `IronPython.Hosting` das bibliotecas IronPython e o namespace `System.Linq`.

C#

```
using System.Linq;
using Microsoft.Scripting.Hosting;
using IronPython.Hosting;
```

No método Main, adicione o código a seguir para criar um novo `Microsoft.Scripting.Hosting.ScriptRuntime` objeto para hospedar as bibliotecas IronPython. O `ScriptRuntime` objeto carrega o módulo da biblioteca IronPython random.py.

C#

```
// Set the current directory to the IronPython libraries.
System.IO.Directory.SetCurrentDirectory(
   Environment.GetFolderPath(Environment.SpecialFolder.ProgramFiles) +
   @"\IronPython 2.7\Lib");

// Create an instance of the random.py IronPython library.
Console.WriteLine("Loading random.py");
ScriptRuntime py = Python.CreateRuntime();
dynamic random = py.UseFile("random.py");
Console.WriteLine("random.py loaded.");
```

Após o código para carregar o módulo random.py, adicione o código a seguir para criar uma matriz de inteiros. A matriz é passada para o `shuffle` método do módulo random.py, que classifica aleatoriamente os valores na matriz.

C#

```
// Initialize an enumerable set of integers.
int[] items = Enumerable.Range(1, 7).ToArray();

// Randomly shuffle the array of integers by using IronPython.
for (int i = 0; i < 5; i++)
{
    random.shuffle(items);
    foreach (int item in items)
    {
        Console.WriteLine(item);
    }
    Console.WriteLine("-------------------");
}
```

Salve o arquivo e pressione `Ctrl+F5` para compilar e executar o aplicativo.

Consulte também

- System.Dynamic
- System.Dynamic.DynamicObject
- Usando o tipo dynamic
- dinâmico
- Implementando interfaces dinâmicas (PDF para download do Microsoft TechNet)

Reduzir alocações de memória usando novos recursos do C#

19/06/2025

> ⓘ **Importante**
>
> As técnicas descritas nesta seção melhoram o desempenho quando *aplicadas a caminhos quentes* em seu código. *Os caminhos frequentes* são as seções da base de código executadas com frequência e repetidamente em operações normais. A aplicação dessas técnicas ao código que não é executado com frequência terá impacto mínimo. Antes de fazer alterações para melhorar o desempenho, é essencial medir uma linha de base. Em seguida, analise essa linha de base para determinar onde ocorrem gargalos de memória. Você pode aprender sobre muitas ferramentas multiplataforma para medir o desempenho do aplicativo na seção sobre Diagnóstico e instrumentação. Você pode praticar uma sessão de criação de perfil no tutorial para medir o uso da memória na documentação do Visual Studio.
>
> Depois de medir o uso da memória e determinar que você pode reduzir as alocações, use as técnicas nesta seção para reduzir as alocações. Após cada alteração sucessiva, meça o uso da memória novamente. Verifique se cada alteração tem um impacto positivo no uso da memória em seu aplicativo.

O trabalho de otimização de desempenho no .NET geralmente significa remover alocações do seu código. Cada bloco de memória alocado deve eventualmente ser liberado. Menos alocações reduzem o tempo gasto na coleta de lixo. Permite um tempo de execução mais previsível ao remover coletas de lixo de caminhos de código específicos.

Para reduzir alocações, uma tática comum é alterar as estruturas de dados críticas de tipos `class` para tipos `struct`. Essa alteração afeta a semântica de usar esses tipos. Parâmetros e retornos agora são passados por valor em vez de por referência. O custo de copiar um valor será insignificante se os tipos forem pequenos, três palavras ou menos (considerando que uma palavra é de tamanho natural de um inteiro). É mensurável e pode ter um impacto real no desempenho para tipos maiores. Para combater o efeito da cópia, os desenvolvedores podem passar esses tipos por meio de `ref` para recuperar a semântica pretendida.

Os recursos do C# `ref` oferecem a capacidade de expressar a semântica desejada para `struct` tipos sem afetar negativamente sua usabilidade geral. Antes desses aprimoramentos, os desenvolvedores precisavam recorrer a estruturas com ponteiros e memória não processada para alcançar o mesmo impacto no desempenho. O compilador gera *um código*

verificávelmente seguro para os novos `ref` recursos relacionados. *Código verificávelmente seguro* significa que o compilador detecta possíveis sobrecargas de buffer ou acesso à memória não alocada ou liberada. O compilador detecta e impede alguns erros.

Transmitir e retornar por referência

Variáveis em C# armazenam *valores*. Em `struct` tipos, o valor é o conteúdo de uma instância do tipo. Em `class` tipos, o valor é uma referência a um bloco de memória que armazena uma instância do tipo. Adicionar o `ref` modificador significa que a variável armazena a *referência* ao valor. Em tipos `struct`, a referência aponta para o armazenamento que contém o valor. Em tipos `class`, a referência aponta para o armazenamento que contém a referência ao bloco de memória.

Em C#, os parâmetros para métodos são *passados por valor* e os valores *retornados são retornados por valor*. O *valor* do argumento é passado para o método. O *valor* do argumento de retorno é o valor retornado.

O `ref`, `in`, `ref readonly` ou `out` modificador indica que o argumento é *passado por referência*. Uma *referência* ao local de armazenamento é passada para o método. Adicionar `ref` à assinatura do método significa que o valor *retornado é retornado por referência*. Uma *referência* ao local de armazenamento é o valor retornado.

Você também pode usar a *atribuição ref* para que uma variável se refira a outra variável. Uma atribuição típica copia o *valor* do lado direito para a variável no lado esquerdo da atribuição. Uma *atribuição ref* copia o local de memória da variável no lado direito para a variável no lado esquerdo. O `ref` agora refere-se à variável original:

C#

```
int anInteger = 42; // assignment.
ref int location = ref anInteger; // ref assignment.
ref int sameLocation = ref location; // ref assignment

Console.WriteLine(location); // output: 42

sameLocation = 19; // assignment

Console.WriteLine(anInteger); // output: 19
```

Ao *atribuir* uma variável, você altera seu *valor*. Ao *atribuir referências* a uma variável, você altera aquilo a que ela se refere.

Você pode trabalhar diretamente com o armazenamento para valores usando variáveis `ref`, transmissão por referência e atribuição de ref. As regras de escopo impostas pelo compilador garantem a segurança ao trabalhar diretamente com o armazenamento.

Os modificadores `ref readonly` e `in` indicam que o argumento deve ser passado por referência e não pode ser reatribuído no método. A diferença é que `ref readonly` indica que o método usa o parâmetro como uma variável. O método pode capturar o parâmetro ou pode retornar o parâmetro como referência de somente leitura. Nesses casos, você deve usar o `ref readonly` modificador. Caso contrário, o `in` modificador oferecerá mais flexibilidade. Você não precisa adicionar o `in` modificador a um argumento para um `in` parâmetro, portanto, é possível atualizar as assinaturas de API existentes com segurança usando o `in` modificador. O compilador emitirá um aviso se você não adicionar o modificador `ref` ou `in` a um argumento para um parâmetro `ref readonly`.

Contexto ref seguro

O C# inclui regras para `ref` expressões para garantir que uma `ref` expressão não possa ser acessada quando o armazenamento ao qual se refere não é mais válido. Considere o seguinte exemplo:

C#

```
public ref int CantEscape()
{
    int index = 42;
    return ref index; // Error: index's ref safe context is the body of CantEscape
}
```

O compilador relata um erro porque você não pode retornar uma referência a uma variável local de um método. O chamador não pode acessar o armazenamento que está sendo referenciado. O *contexto de ref safe* define o escopo no qual uma `ref` expressão é segura para acessar ou modificar. A tabela a seguir lista os *contextos de ref safe* para tipos de variáveis. Os campos `ref` não podem ser declarados em um `class` ou em uma `struct` não ref, portanto, essas linhas não estão na tabela:

Expandir a tabela

Declaração	***contexto ref seguro***
local não ref	bloco onde o local é declarado
parâmetro não referencial	método atual

Declaração	*contexto ref seguro*
Parâmetro `ref`, `ref readonly`, `in`	método de chamada
Parâmetro `out`	método atual
Campo de `class`	método de chamada
campo `struct` não ref	método atual
Campo `ref` de `ref struct`	método de chamada

Uma variável poderá ser `ref` retornada se o *contexto ref seguro* for o método chamador. Se o *contexto ref seguro* for o método atual ou um bloco, o retorno de `ref` não será permitido. O snippet a seguir mostra dois exemplos. Um campo membro pode ser acessado no escopo que chama um método, portanto, o *contexto ref seguro* de um campo de classe ou estrutura é o método de chamada. O *contexto seguro de referência* para um parâmetro com os modificadores `ref` ou `in` abrange todo o método. Ambos podem ser `ref` retornados de um método membro:

C#

```
private int anIndex;

public ref int RetrieveIndexRef()
{
    return ref anIndex;
}

public ref int RefMin(ref int left, ref int right)
{
    if (left < right)
        return ref left;
    else
        return ref right;
}
```

Observação

Quando o modificador `ref readonly` ou o modificador `in` é aplicado a um parâmetro, esse parâmetro pode ser retornado por `ref readonly`, não por `ref`.

O compilador garante que uma referência não possa escapar de seu *contexto ref safe*. Você pode usar os parâmetros `ref`, as variáveis locais `ref return` e `ref` com segurança porque o

compilador detecta caso você tenha escrito acidentalmente um código onde uma expressão `ref` poderia ser acessada quando seu armazenamento não é válido.

Contexto seguro e estruturas de referência

`ref struct` os tipos exigem mais regras para garantir que eles possam ser usados com segurança. Um `ref struct` tipo pode incluir `ref` campos. Isso requer a introdução de um *contexto seguro*. Para a maioria dos tipos, o *contexto seguro* é o método de chamada. Em outras palavras, um valor que não é um `ref struct` sempre pode ser retornado de um método.

Informalmente, o *contexto seguro* para um `ref struct` é o escopo em que todos os seus `ref` campos podem ser acessados. Em outras palavras, é a interseção do *contexto ref seguro* de todos os campos `ref`. O método a seguir retorna um `ReadOnlySpan<char>` para um campo membro, portanto, o *contexto seguro* é o método:

C#

```
private string longMessage = "This is a long message";

public ReadOnlySpan<char> Safe()
{
    var span = longMessage.AsSpan();
    return span;
}
```

Por outro lado, o código a seguir emite um erro porque o membro `ref field` do `Span<int>` refere-se a uma matriz de inteiros alocados na pilha. Ele não pode escapar do método:

C#

```
public Span<int> M()
{
    int length = 3;
    Span<int> numbers = stackalloc int[length];
    for (var i = 0; i < length; i++)
    {
        numbers[i] = i;
    }
    return numbers; // Error! numbers can't escape this method.
}
```

Unificar tipos de memória

A introdução de System.Span<T> e System.Memory<T> fornece um modelo unificado para trabalhar com memória. System.ReadOnlySpan<T> e System.ReadOnlyMemory<T> fornecem versões somente leitura para acessar a memória. Todos eles fornecem uma abstração em um bloco de memória armazenando uma matriz de elementos semelhantes. A diferença é que `Span<T>` e `ReadOnlySpan<T>` são `ref struct` tipos, enquanto `Memory<T>` e `ReadOnlyMemory<T>` são `struct` tipos. Os intervalos contêm um `ref field`. Portanto, instâncias de um intervalo não podem deixar o *contexto seguro*. O *contexto seguro* de um `ref struct` é o *contexto seguro de referência* de sua `ref field`. A implementação de `Memory<T>` e `ReadOnlyMemory<T>` remove essa restrição. Você usa esses tipos para acessar diretamente buffers de memória.

Melhorar o desempenho com segurança de ref

Usar esses recursos para melhorar o desempenho envolve estas tarefas:

- *Evite alocações*: quando você altera um tipo de um `class` para um `struct`, você altera como ele é armazenado. As variáveis locais são armazenadas na pilha. Os membros são armazenados de forma integrada quando o objeto de contêiner é alocado. Essa alteração significa menos alocações e isso diminui o trabalho que o coletor de lixo faz. Também pode diminuir a pressão de memória, fazendo com que o coletor de lixo seja executado com menos frequência.
- *Preservar semântica de referência*: alterar um tipo de um `class` para um `struct` altera a semântica de passar uma variável para um método. O código que modificou o estado de seus parâmetros precisa de modificação. Agora que o parâmetro é um `struct`, o método está modificando uma cópia do objeto original. Você pode restaurar a semântica original passando esse parâmetro como um `ref` parâmetro. Após essa alteração, o método modifica o original `struct` novamente.
- *Evite copiar dados*: copiar tipos maiores `struct` pode afetar o desempenho em alguns caminhos de código. Você também pode adicionar o `ref` modificador para passar estruturas de dados maiores aos métodos por referência, em vez de por valor.
- *Restringir modificações*: quando um `struct` tipo é passado por referência, o método chamado pode modificar o estado do struct. Você pode substituir o modificador `ref` pelos modificadores `ref readonly` ou `in` para indicar que o argumento não pode ser modificado. Prefira `ref readonly` quando o método capta o parâmetro ou o retorna por referência somente leitura. Você também pode criar tipos `readonly struct` ou `struct` com membros `readonly` para fornecer mais controle sobre quais membros de um `struct` podem ser modificados.
- *Manipular diretamente a memória*: alguns algoritmos são mais eficientes ao tratar estruturas de dados como um bloco de memória que contém uma sequência de elementos. Os tipos `Span` e `Memory` fornecem acesso seguro a blocos de memória.

Nenhuma dessas técnicas exige `unsafe` código. Usado com sabedoria, você pode obter características de desempenho do código seguro que antes só era possível usando técnicas não seguras. Você pode experimentar as técnicas por conta própria no tutorial sobre como reduzir as alocações de memória.

Tutorial: como reduzir alocações de memória com segurança `ref`

19/06/2025

Geralmente, o ajuste de desempenho para um aplicativo .NET envolve duas técnicas. Primeiro, reduza o número e o tamanho das alocações de heap. Em segundo lugar, reduza a frequência com que os dados são copiados. O Visual Studio fornece ótimas ferramentas que ajudam a analisar como seu aplicativo está usando memória. Depois de determinar onde seu aplicativo faz alocações desnecessárias, você faz alterações para minimizar essas alocações. Você converte os tipos `class` em tipos `struct`. Você usa `ref` de segurança para preservar a semântica e minimizar a cópia extra.

Use o Visual Studio 17.5 para obter a melhor experiência com este tutorial. A ferramenta de alocação de objeto .NET usada para analisar o uso da memória faz parte do Visual Studio. Você pode usar o Visual Studio Code e a linha de comando para executar o aplicativo e fazer todas as alterações. No entanto, você não poderá ver os resultados da análise de suas alterações.

O aplicativo que você usará é uma simulação de um aplicativo IoT que monitora vários sensores para determinar se um intruso entrou em uma galeria secreta com objetos de valor. Os sensores de IoT estão constantemente enviando dados que medem a mistura de Oxigênio (O2) e Dióxido de Carbono (CO2) no ar. Eles também relatam a temperatura e a umidade relativa. Cada um desses valores está flutuando ligeiramente o tempo todo. No entanto, quando uma pessoa entra na sala, a mudança torna-se um pouco mais acentuada, e sempre na mesma direção: Oxigênio diminui, Dióxido de Carbono aumenta, temperatura aumenta, tal como a umidade relativa. Quando os sensores se combinam para mostrar aumentos, o alarme de intruso é disparado.

Neste tutorial, você executará o aplicativo, fará medidas sobre alocações de memória e, em seguida, melhorará o desempenho reduzindo o número de alocações. O código-fonte está disponível no navegador de exemplos.

Explorar o aplicativo inicial

Baixe o aplicativo e execute o exemplo inicial. O aplicativo inicial funciona corretamente, mas como aloca muitos objetos pequenos com cada ciclo de medição, seu desempenho diminui lentamente à medida que é executado ao longo do tempo.

```
Console
```

```
Press <return> to start simulation

Debounced measurements:
    Temp:      67.332
    Humidity:  41.077%
    Oxygen:    21.097%
    CO2 (ppm): 404.906
Average measurements:
    Temp:      67.332
    Humidity:  41.077%
    Oxygen:    21.097%
    CO2 (ppm): 404.906

Debounced measurements:
    Temp:      67.349
    Humidity:  46.605%
    Oxygen:    20.998%
    CO2 (ppm): 408.707
Average measurements:
    Temp:      67.349
    Humidity:  46.605%
    Oxygen:    20.998%
    CO2 (ppm): 408.707
```

Muitas linhas removidas.

Console

```
Debounced measurements:
    Temp:      67.597
    Humidity:  46.543%
    Oxygen:    19.021%
    CO2 (ppm): 429.149
Average measurements:
    Temp:      67.568
    Humidity:  45.684%
    Oxygen:    19.631%
    CO2 (ppm): 423.498
Current intruders: 3
Calculated intruder risk: High

Debounced measurements:
    Temp:      67.602
    Humidity:  46.835%
    Oxygen:    19.003%
    CO2 (ppm): 429.393
Average measurements:
    Temp:      67.568
    Humidity:  45.684%
    Oxygen:    19.631%
    CO2 (ppm): 423.498
```

```
Current intruders: 3
Calculated intruder risk: High
```

Você pode explorar o código para saber como o aplicativo funciona. O programa principal executa a simulação. Depois de pressionar `<Enter>`, ele cria uma sala e coleta alguns dados de linha de base iniciais:

C#

```
Console.WriteLine("Press <return> to start simulation");
Console.ReadLine();
var room = new Room("gallery");
var r = new Random();

int counter = 0;

room.TakeMeasurements(
    m =>
    {
        Console.WriteLine(room.Debounce);
        Console.WriteLine(room.Average);
        Console.WriteLine();
        counter++;
        return counter < 20000;
    });
```

Depois que os dados de linha de base tiverem sido estabelecidos, ele executará a simulação na sala, em que um gerador de número aleatório determina se um intruso entrou na sala:

C#

```
counter = 0;
room.TakeMeasurements(
    m =>
    {
        Console.WriteLine(room.Debounce);
        Console.WriteLine(room.Average);
        room.Intruders += (room.Intruders, r.Next(5)) switch
        {
            ( > 0, 0) => -1,
            ( < 3, 1) => 1,
            _ => 0
        };

        Console.WriteLine($"Current intruders: {room.Intruders}");
        Console.WriteLine($"Calculated intruder risk: {room.RiskStatus}");
        Console.WriteLine();
        counter++;
        return counter < 200000;
    });
```

Outros tipos contêm as medidas, uma medida esporádica que é a média das últimas 50 medidas e a média de todas as medidas realizadas.

Em seguida, execute o aplicativo usando a ferramenta de alocação de objeto .NET. Verifique se você está usando o `Release` build, não o `Debug` build. No menu *Depuração*, abra o *Analisador de Desempenho*. Verifique a opção *Acompanhamento de Alocação de Objetos do .NET*, mas nada mais. Execute seu aplicativo até a conclusão. O criador de perfil mede as alocações de objetos e relata as alocações e os ciclos de coleta de lixo. Você deverá ver um grafo semelhante à seguinte imagem:

O grafo anterior mostra que trabalhar para minimizar alocações fornecerá benefícios de desempenho. Você vê um padrão de serrilhado no grafo de objetos dinâmicos. Isso informa que vários objetos são criados que rapidamente se tornam lixo. Eles são coletados depois, como é mostrado no grafo delta de objeto. As barras vermelhas para baixo indicam um ciclo de coleta de lixo.

Em seguida, examine a guia *Alocações* abaixo dos grafos. Esta tabela mostra quais tipos são mais alocados:

Insights | Allocations | Call Tree | Functions | Collections

Type	Allocations
Small Object Heap	
System.String	842,451
IntruderAlert.SensorMeasurement	220,000
IntruderAlert.IntruderRisk	200,000
System.Char[]	80
System.Diagnostics.Tracing.EventSource.EventMetadata[]	7
System.Byte[]	339

O System.String tipo representa a maioria das alocações. A tarefa mais importante é minimizar a frequência das alocações de strings. Este aplicativo imprime várias saídas formatadas no console constantemente. Para esta simulação, queremos manter as mensagens; portanto, iremos nos concentrar nos dois próximos tipos de linha: o tipo `SensorMeasurement` e o tipo `IntruderRisk`.

Clique duas vezes na `SensorMeasurement` linha. Você pode ver que todas as alocações ocorrem no `static` método `SensorMeasurement.TakeMeasurement`. Você pode ver o método no seguinte snippet:

```
C#

public static SensorMeasurement TakeMeasurement(string room, int intruders)
{
    return new SensorMeasurement
    {
        CO2 = (CO2Concentration + intruders * 10) + (20 * generator.NextDouble() -
10.0),
        O2 = (O2Concentration - intruders * 0.01) + (0.005 *
generator.NextDouble() - 0.0025),
        Temperature = (TemperatureSetting + intruders * 0.05) + (0.5 *
generator.NextDouble() - 0.25),
        Humidity = (HumiditySetting + intruders * 0.005) + (0.20 *
generator.NextDouble() - 0.10),
        Room = room,
        TimeRecorded = DateTime.Now
    };
}
```

Cada medida aloca um novo `SensorMeasurement` objeto, que é um `class` tipo. Cada `SensorMeasurement` criado causa uma alocação de heap.

Alterar classes para structs

O código a seguir mostra a declaração inicial de `SensorMeasurement`:

```
C#

public class SensorMeasurement
{
    private static readonly Random generator = new Random();

    public static SensorMeasurement TakeMeasurement(string room, int intruders)
    {
        return new SensorMeasurement
        {
            CO2 = (CO2Concentration + intruders * 10) + (20 *
generator.NextDouble() - 10.0),
            O2 = (O2Concentration - intruders * 0.01) + (0.005 *
generator.NextDouble() - 0.0025),
            Temperature = (TemperatureSetting + intruders * 0.05) + (0.5 *
generator.NextDouble() - 0.25),
            Humidity = (HumiditySetting + intruders * 0.005) + (0.20 *
generator.NextDouble() - 0.10),
            Room = room,
```

```
            TimeRecorded = DateTime.Now
        };
    }

    private const double CO2Concentration = 409.8; // increases with people.
    private const double O2Concentration = 0.2100; // decreases
    private const double TemperatureSetting = 67.5; // increases
    private const double HumiditySetting = 0.4500; // increases

    public required double CO2 { get; init; }
    public required double O2 { get; init; }
    public required double Temperature { get; init; }
    public required double Humidity { get; init; }
    public required string Room { get; init; }
    public required DateTime TimeRecorded { get; init; }

    public override string ToString() => $"""
            Room: {Room} at {TimeRecorded}:
                Temp:      {Temperature:F3}
                Humidity:  {Humidity:P3}
                Oxygen:    {O2:P3}
                CO2 (ppm): {CO2:F3}
            """;
}
```

O tipo foi originalmente criado como um `class` porque contém várias `double` medidas. Ele é maior do que o ideal para ser copiado em caminhos críticos. No entanto, essa decisão significou um grande número de alocações. Altere o tipo de um `class` para um `struct`.

A alteração de um `class` para `struct` introduz alguns erros de compilação porque o código original usava verificações de referência `null` em alguns pontos. O primeiro está na `DebounceMeasurement` classe, no `AddMeasurement` método:

C#

```
public void AddMeasurement(SensorMeasurement datum)
{
    int index = totalMeasurements % debounceSize;
    recentMeasurements[index] = datum;
    totalMeasurements++;
    double sumCO2 = 0;
    double sumO2 = 0;
    double sumTemp = 0;
    double sumHumidity = 0;
    for (int i = 0; i < debounceSize; i++)
    {
        if (recentMeasurements[i] is not null)
        {
            sumCO2 += recentMeasurements[i].CO2;
            sumO2+= recentMeasurements[i].O2;
            sumTemp+= recentMeasurements[i].Temperature;
```

```
            sumHumidity += recentMeasurements[i].Humidity;
        }
    }
    O2 = sumO2 / ((totalMeasurements > debounceSize) ? debounceSize :
totalMeasurements);
    CO2 = sumCO2 / ((totalMeasurements > debounceSize) ? debounceSize :
totalMeasurements);
    Temperature = sumTemp / ((totalMeasurements > debounceSize) ? debounceSize :
totalMeasurements);
    Humidity = sumHumidity / ((totalMeasurements > debounceSize) ? debounceSize :
totalMeasurements);
}
```

O `DebounceMeasurement` tipo contém uma matriz de 50 medidas. As leituras de um sensor são relatadas como a média das últimas 50 medidas. Isso reduz o ruído nas leituras. Antes de 50 leituras completas serem realizadas, esses valores são `null`. O código verifica a `null` referência para relatar a média correta na inicialização do sistema. Depois de alterar o `SensorMeasurement` tipo para um struct, você deve usar um teste diferente. O tipo `SensorMeasurement` inclui um `string` para o identificador de sala, de forma que você possa usar esse teste no lugar desse.

C#

```
if (recentMeasurements[i].Room is not null)
```

Os outros três erros do compilador estão todos no método que faz medidas repetidamente em uma sala:

C#

```
public void TakeMeasurements(Func<SensorMeasurement, bool> MeasurementHandler)
{
    SensorMeasurement? measure = default;
    do {
        measure = SensorMeasurement.TakeMeasurement(Name, Intruders);
        Average.AddMeasurement(measure);
        Debounce.AddMeasurement(measure);
    } while (MeasurementHandler(measure));
}
```

No método inicializador, a variável local para a `SensorMeasurement` é uma *referência nula*:

C#

```
SensorMeasurement? measure = default;
```

Agora que o `SensorMeasurement` é um `struct`, em vez de um `class`, o *nullable* é um tipo de valor anulável. Você pode alterar a declaração para um tipo de valor para corrigir os erros restantes do compilador:

```
C#

SensorMeasurement measure = default;
```

Agora que os erros do compilador foram resolvidos, você deve examinar o código para garantir que a semântica não tenha sido alterada. Como `struct` os tipos são passados por valor, as modificações feitas nos parâmetros do método não são visíveis após o retorno do método.

> ⓘ **Importante**
>
> Alterar um tipo de um `class` para um `struct` pode alterar a semântica do seu programa. Quando um `class` tipo é passado para um método, todas as mutações feitas no método são feitas no argumento. Quando um tipo `struct` é passado a um método, as mutações feitas no método são feitas em uma *cópia* do argumento. Isso significa que qualquer método que, por padrão, modifique seus argumentos deve ser atualizado para usar o modificador `ref` em qualquer tipo de argumento que você tenha alterado de `class` para `struct`.

O `SensorMeasurement` tipo não inclui nenhum método que altere o estado, portanto, isso não é uma preocupação neste exemplo. Você pode provar isso adicionando o `readonly` modificador ao `SensorMeasurement` struct:

```
C#

public readonly struct SensorMeasurement
```

O compilador impõe a natureza `readonly` do struct `SensorMeasurement`. Se a inspeção do código tiver perdido algum método que modificou o estado, o compilador informará. Seu aplicativo ainda é compilado sem erros, portanto, esse tipo é `readonly`. Adicionar o modificador `readonly` quando você altera um tipo de `class` para `struct` pode ajudá-lo a encontrar membros que modificam o estado de `struct`.

Evite fazer cópias

Você removeu um grande número de alocações desnecessárias do seu aplicativo. O `SensorMeasurement` tipo não aparece na tabela em nenhum lugar.

Agora, ele está fazendo trabalho extra copiando a `SensorMeasurement` estrutura sempre que ela é usada como um parâmetro ou um valor retornado. O "struct" `SensorMeasurement` contém quatro "doubles", um DateTime e um `string`. Essa estrutura é mensuravelmente maior que uma referência. Vamos adicionar os modificadores `ref` ou `in` aos locais onde o tipo `SensorMeasurement` é usado.

A próxima etapa é localizar métodos que retornam uma medida ou que façam uma medida como argumento e usem referências sempre que possível. Comece no struct `SensorMeasurement`. O método estático `TakeMeasurement` cria e retorna um novo `SensorMeasurement`:

C#

```
public static SensorMeasurement TakeMeasurement(string room, int intruders)
{
    return new SensorMeasurement
    {
        CO2 = (CO2Concentration + intruders * 10) + (20 * generator.NextDouble() -
10.0),
        O2 = (O2Concentration - intruders * 0.01) + (0.005 *
generator.NextDouble() - 0.0025),
        Temperature = (TemperatureSetting + intruders * 0.05) + (0.5 *
generator.NextDouble() - 0.25),
        Humidity = (HumiditySetting + intruders * 0.005) + (0.20 *
generator.NextDouble() - 0.10),
        Room = room,
        TimeRecorded = DateTime.Now
    };
}
```

Deixaremos este como está, retornando por valor. Se você tentasse retornar por `ref`, obteria um erro de compilação. Não é possível retornar um `ref` a uma estrutura criada localmente no método. O design do struct imutável significa que você só pode definir os valores da medida na construção. Esse método deve criar um novo struct de medida.

Vamos analisar `DebounceMeasurement.AddMeasurement` novamente. Você deve adicionar o `in` modificador ao `measurement` parâmetro:

C#

```
public void AddMeasurement(in SensorMeasurement datum)
{
    int index = totalMeasurements % debounceSize;
```

```
    recentMeasurements[index] = datum;
    totalMeasurements++;
    double sumCO2 = 0;
    double sumO2 = 0;
    double sumTemp = 0;
    double sumHumidity = 0;
    for (int i = 0; i < debounceSize; i++)
    {
        if (recentMeasurements[i].Room is not null)
        {
            sumCO2 += recentMeasurements[i].CO2;
            sumO2+= recentMeasurements[i].O2;
            sumTemp+= recentMeasurements[i].Temperature;
            sumHumidity += recentMeasurements[i].Humidity;
        }
    }
    O2 = sumO2 / ((totalMeasurements > debounceSize) ? debounceSize :
totalMeasurements);
    CO2 = sumCO2 / ((totalMeasurements > debounceSize) ? debounceSize :
totalMeasurements);
    Temperature = sumTemp / ((totalMeasurements > debounceSize) ? debounceSize :
totalMeasurements);
    Humidity = sumHumidity / ((totalMeasurements > debounceSize) ? debounceSize :
totalMeasurements);
}
```

Isso salva uma operação de cópia. O `in` parâmetro é uma referência à cópia já criada pelo chamador. Você também pode salvar uma cópia com o `TakeMeasurement` método no `Room` tipo. Este método ilustra como o compilador fornece segurança quando você passa argumentos por `ref`. O método inicial `TakeMeasurement` no `Room` tipo usa um argumento de `Func<SensorMeasurement, bool>`. Se você tentar adicionar o `in` modificador ou `ref` a essa declaração, o compilador relatará um erro. Você não pode passar um `ref` argumento para uma expressão lambda. O compilador não pode garantir que a expressão chamada não copie a referência. Se a expressão lambda *capturar* a referência, a referência poderá ter um tempo de vida maior do que o valor ao qual se refere. Acessá-lo fora de seu *contexto ref safe* resultaria em corrupção de memória. As `ref` regras de segurança não permitem isso. Saiba mais na visão geral de recursos de segurança de referência.

Preservar semântica

Os conjuntos finais de alterações não terão um grande impacto no desempenho desse aplicativo porque os tipos não são criados em caminhos críticos. Essas alterações ilustram algumas das outras técnicas que você usaria no ajuste de desempenho. Vamos dar uma olhada na classe inicial `Room` :

```
C#
```

```
public class Room
{
    public AverageMeasurement Average { get; } = new ();
    public DebounceMeasurement Debounce { get; } = new ();
    public string Name { get; }

    public IntruderRisk RiskStatus
    {
        get
        {
            var CO2Variance = (Debounce.CO2 - Average.CO2) > 10.0 / 4;
            var O2Variance = (Average.O2 - Debounce.O2) > 0.005 / 4.0;
            var TempVariance = (Debounce.Temperature - Average.Temperature) > 0.05
/ 4.0;
            var HumidityVariance = (Debounce.Humidity - Average.Humidity) > 0.20 /
4;
            IntruderRisk risk = IntruderRisk.None;
            if (CO2Variance) { risk++; }
            if (O2Variance) { risk++; }
            if (TempVariance) { risk++; }
            if (HumidityVariance) { risk++; }
            return risk;
        }
    }

    public int Intruders { get; set; }


    public Room(string name)
    {
        Name = name;
    }

    public void TakeMeasurements(Func<SensorMeasurement, bool> MeasurementHandler)
    {
        SensorMeasurement? measure = default;
        do {
            measure = SensorMeasurement.TakeMeasurement(Name, Intruders);
            Average.AddMeasurement(measure);
            Debounce.AddMeasurement(measure);
        } while (MeasurementHandler(measure));
    }
}
```

Esse tipo contém várias propriedades. Alguns são tipos `class`. A criação de um `Room` objeto envolve várias alocações. Uma para o `Room` em si e outra para cada um dos membros de um tipo `class` que ele contém. Você pode converter duas dessas propriedades de tipos `class` em tipos `struct`: os tipos `DebounceMeasurement` e `AverageMeasurement`. Vamos trabalhar nessa transformação com ambos os tipos.

Altere o tipo `DebounceMeasurement` de um `class` para `struct`. Isso apresenta um erro `CS8983: A 'struct' with field initializers must include an explicitly declared constructor` do compilador. Você pode corrigir isso adicionando um construtor sem parâmetros vazio:

```
C#

public DebounceMeasurement() { }
```

Você pode aprender mais sobre este requisito no artigo de referência linguística sobre structs.

A Object.ToString() substituição não modifica nenhum dos valores do struct. Você pode adicionar o `readonly` modificador a essa declaração de método. O `DebounceMeasurement` tipo é *mutável*, portanto, você precisará tomar cuidado para que as modificações não afetem cópias descartadas. O `AddMeasurement` método modifica o estado do objeto. Ele é chamado da classe `Room`, no método `TakeMeasurements`. Você deseja que essas alterações persistam depois de chamar o método. Você pode alterar a `Room.Debounce` propriedade para retornar uma *referência* a uma única instância do `DebounceMeasurement` tipo:

```
C#

private DebounceMeasurement debounce = new();
public ref readonly DebounceMeasurement Debounce { get { return ref debounce; } }
```

Há algumas alterações no exemplo anterior. Primeiro, a *propriedade* é uma propriedade somente leitura que retorna uma referência somente leitura à instância pertencente a essa sala. Agora ele é apoiado por um campo declarado que é inicializado quando o `Room` objeto é instanciado. Depois de fazer essas alterações, você atualizará a implementação do `AddMeasurement` método. Ela usa o campo de suporte privado, `debounce`, não a propriedade somente leitura `Debounce`. Dessa forma, as alterações ocorrem na única instância criada durante a inicialização.

A mesma técnica funciona com a `Average` propriedade. Primeiro, modifique o `AverageMeasurement` tipo de um `class` para um `struct` e adicione o `readonly` modificador ao `ToString` método:

```
C#

namespace IntruderAlert;

public struct AverageMeasurement
{
    private double sumCO2 = 0;
    private double sumO2 = 0;
    private double sumTemperature = 0;
```

```
    private double sumHumidity = 0;
    private int totalMeasurements = 0;

    public AverageMeasurement() { }

    public readonly double CO2 => sumCO2 / totalMeasurements;
    public readonly double O2 => sumO2 / totalMeasurements;
    public readonly double Temperature => sumTemperature / totalMeasurements;
    public readonly double Humidity => sumHumidity / totalMeasurements;

    public void AddMeasurement(in SensorMeasurement datum)
    {
        totalMeasurements++;
        sumCO2 += datum.CO2;
        sumO2 += datum.O2;
        sumTemperature += datum.Temperature;
        sumHumidity+= datum.Humidity;
    }

    public readonly override string ToString() => $"""
        Average measurements:
            Temp:      {Temperature:F3}
            Humidity:  {Humidity:P3}
            Oxygen:    {O2:P3}
            CO2 (ppm): {CO2:F3}
        """;
}
```

Em seguida, modifique a `Room` classe seguindo a mesma técnica usada para a `Debounce` propriedade. A propriedade `Average` retorna um `readonly ref` ao campo privado para a medição média. O `AddMeasurement` método modifica os campos internos.

C#

```
private AverageMeasurement average = new();
public  ref readonly AverageMeasurement Average { get { return ref average; } }
```

Evitar boxe

Há uma última mudança para melhorar o desempenho. O programa principal é imprimir estatísticas para a sala, incluindo a avaliação de risco:

C#

```
Console.WriteLine($"Current intruders: {room.Intruders}");
Console.WriteLine($"Calculated intruder risk: {room.RiskStatus}");
```

A chamada ao `ToString` gerado faz a conversão boxing para o valor de enumeração. Você pode evitar isso escrevendo uma substituição na `Room` classe que formata a cadeia de caracteres com base no valor do risco estimado:

C#

```
public override string ToString() =>
    $"Calculated intruder risk: {RiskStatus switch
    {
        IntruderRisk.None => "None",
        IntruderRisk.Low => "Low",
        IntruderRisk.Medium => "Medium",
        IntruderRisk.High => "High",
        IntruderRisk.Extreme => "Extreme",
        _ => "Error!"
    }}, Current intruders: {Intruders.ToString()}";
```

Em seguida, modifique o código no programa principal para chamar este novo `ToString` método:

C#

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

Execute o aplicativo usando a ferramenta de análise de desempenho e examine a tabela atualizada de alocações.

Insights | Allocations | Call Tree | Functions | Collections

Type	Allocations
Small Object Heap	
System.String	642,271
System.Reflection.RuntimeParameterInfo	676
System.Object[]	493
System.Byte[]	339
System.Reflection.ParameterInfo[]	295
System.Reflection.RuntimeMethodInfo	255
System.Signature	237

Você removeu várias alocações e forneceu ao aplicativo um aumento de desempenho.

Como usar a segurança de referência no aplicativo

Essas técnicas são um ajuste de desempenho de nível baixo. Elas podem aumentar o desempenho no aplicativo quando aplicadas a caminhos críticos e quando o impacto é medido antes e depois das alterações. Na maioria dos casos, o ciclo que você seguirá é:

- *Alocações de medidas*: determine quais tipos estão sendo mais alocados e quando você pode reduzir as alocações de heap.
- *Converter classe em struct*: muitas vezes, os tipos podem ser convertidos de um `class` para um `struct`. O aplicativo usa espaço de pilha em vez de fazer alocações de heap.
- *Preservar semântica*: converter um `class` em um `struct` pode afetar a semântica para parâmetros e valores retornados. Qualquer método que modifique seus parâmetros agora deve marcar esses parâmetros com o `ref` modificador. Isso garante que as modificações sejam feitas no objeto correto. Da mesma forma, se um valor retornado de propriedade ou método deve ser modificado pelo chamador, esse retorno deve ser marcado com o `ref` modificador.
- *Evite cópias*: Ao passar uma estrutura grande como parâmetro, você pode marcar o parâmetro com o modificador `in`. Você pode passar uma referência em menos bytes e garantir que o método não modifique o valor original. Você também pode retornar valores `readonly ref` para retornar uma referência que não pode ser modificada.

Usando essas técnicas, você pode melhorar o desempenho em caminhos frequentes do seu código.

Tutorial: Escrever um manipulador de interpolação de cadeia de caracteres personalizado

Artigo • 04/01/2025

Neste tutorial, você aprenderá a:

- ✓ Implementar o padrão de manipulador de interpolação de cadeia de caracteres
- ✓ Interaja com o receptor em uma operação de interpolação de cadeia de caracteres.
- ✓ Adicionar argumentos ao manipulador de interpolação de cadeia de caracteres
- ✓ Entender os novos recursos de biblioteca para interpolação de cadeia de caracteres

Pré-requisitos

Você precisa configurar seu computador para executar o .NET. O compilador C# está disponível com do Visual Studio 2022 ou o SDK do .NET .

Este tutorial pressupõe que você esteja familiarizado com C# e .NET, incluindo o Visual Studio ou a CLI do .NET.

Você pode escrever um manipulador personalizado de cadeia de caracteres interpolada . Um manipulador de cadeia de caracteres interpolado é um tipo que processa a expressão de espaço reservado em uma cadeia de caracteres interpolada. Sem um manipulador personalizado, os marcadores de posição são processados de forma semelhante a String.Format. Cada espaço reservado é formatado como texto e, em seguida, os componentes são concatenados para formar a cadeia de caracteres resultante.

Você pode escrever um manipulador para qualquer cenário em que use informações sobre a cadeia de caracteres resultante. Será usado? Quais restrições estão no formato? Alguns exemplos incluem:

- Você pode exigir que nenhuma das cadeias de caracteres resultantes seja maior que algum limite, como 80 caracteres. Você pode processar as cadeias de caracteres interpoladas para preencher um buffer de comprimento fixo e interromper o processamento depois que o comprimento do buffer for atingido.
- Você pode ter um formato tabular e cada espaço reservado deve ter um comprimento fixo. Um manipulador personalizado pode impor isso, em vez de forçar que todo o código do cliente esteja em conformidade.

Neste tutorial, você criará um manipulador de interpolação de cadeia de caracteres para um dos principais cenários de desempenho: bibliotecas de log. Dependendo do nível de log configurado, o trabalho para construir uma mensagem de log não é necessário. Se o registro em log estiver desativado, o trabalho para construir uma cadeia de caracteres a partir de uma expressão de cadeia de caracteres interpolada não será necessário. A mensagem nunca é impressa, portanto, qualquer concatenação de cadeia de caracteres pode ser ignorada. Além disso, todas as expressões usadas nos espaços reservados, incluindo a geração de rastreamentos de pilha, não precisam ser feitas.

Um manipulador de cadeia de caracteres interpolado pode determinar se a cadeia de caracteres formatada será usada e só executará o trabalho necessário, se necessário.

Implementação inicial

Vamos começar com uma classe de `Logger` básica que dá suporte a diferentes níveis:

C#

```
public enum LogLevel
{
    Off,
    Critical,
    Error,
    Warning,
    Information,
    Trace
}

public class Logger
{
    public LogLevel EnabledLevel { get; init; } = LogLevel.Error;

    public void LogMessage(LogLevel level, string msg)
    {
        if (EnabledLevel < level) return;
        Console.WriteLine(msg);
    }
}
```

Esse `Logger` dá suporte a seis níveis diferentes. Quando uma mensagem não passa pelo filtro do nível de log, não há saída de dados. A API pública para o logger aceita uma string (totalmente formatada) como mensagem. Todo o trabalho para criar a string já está concluído.

Implementar o padrão de manipulador

Esta etapa é criar um manipulador de cadeia de caracteres *interpolado* que recria o comportamento atual. Um manipulador de cadeia de caracteres interpolado é um tipo que deve ter as seguintes características:

- O System.Runtime.CompilerServices.InterpolatedStringHandlerAttribute aplicado ao tipo.
- Um construtor que tem dois parâmetros `int`, `literalLength` e `formattedCount`. (Mais parâmetros são permitidos).
- Um método público `AppendLiteral` com a assinatura: `public void AppendLiteral(string s)`.
- Um método público `AppendFormatted` genérico com a assinatura: `public void AppendFormatted<T>(T t)`.

Internamente, o construtor cria a cadeia de caracteres formatada e fornece um membro para um cliente recuperar essa cadeia de caracteres. O código a seguir mostra um tipo de `LogInterpolatedStringHandler` que atende a esses requisitos:

C#

```
[InterpolatedStringHandler]
public ref struct LogInterpolatedStringHandler
{
    // Storage for the built-up string
    StringBuilder builder;

    public LogInterpolatedStringHandler(int literalLength, int
formattedCount)
    {
        builder = new StringBuilder(literalLength);
        Console.WriteLine($"\tliteral length: {literalLength},
formattedCount: {formattedCount}");
    }

    public void AppendLiteral(string s)
    {
        Console.WriteLine($"\tAppendLiteral called: {{{s}}}");

        builder.Append(s);
        Console.WriteLine($"\tAppended the literal string");
    }

    public void AppendFormatted<T>(T t)
    {
        Console.WriteLine($"\tAppendFormatted called: {{{t}}} is of type
{typeof(T)}");

        builder.Append(t?.ToString());
        Console.WriteLine($"\tAppended the formatted object");
    }
```

```
    internal string GetFormattedText() => builder.ToString();
}
```

Agora você pode adicionar uma sobrecarga para `LogMessage` na classe `Logger` para experimentar o novo manipulador de cadeia de caracteres interpolado:

C#

```
public void LogMessage(LogLevel level, LogInterpolatedStringHandler builder)
{
    if (EnabledLevel < level) return;
    Console.WriteLine(builder.GetFormattedText());
}
```

Você não precisa remover o método de `LogMessage` original, o compilador prefere um método com um parâmetro de manipulador interpolado em vez de um método com um parâmetro `string` quando o argumento é uma expressão de cadeia de caracteres interpolada.

Você pode verificar se o novo manipulador é invocado usando o seguinte código como o programa principal:

C#

```
var logger = new Logger() { EnabledLevel = LogLevel.Warning };
var time = DateTime.Now;

logger.LogMessage(LogLevel.Error, $"Error Level. CurrentTime: {time}. This
is an error. It will be printed.");
logger.LogMessage(LogLevel.Trace, $"Trace Level. CurrentTime: {time}. This
won't be printed.");
logger.LogMessage(LogLevel.Warning, "Warning Level. This warning is a
string, not an interpolated string expression.");
```

A execução do aplicativo produz uma saída semelhante ao seguinte texto:

PowerShell

```
        literal length: 65, formattedCount: 1
        AppendLiteral called: {Error Level. CurrentTime: }
        Appended the literal string
        AppendFormatted called: {10/20/2021 12:19:10 PM} is of type
System.DateTime
        Appended the formatted object
        AppendLiteral called: {. This is an error. It will be printed.}
        Appended the literal string
Error Level. CurrentTime: 10/20/2021 12:19:10 PM. This is an error. It will
```

```
be printed.
        literal length: 50, formattedCount: 1
        AppendLiteral called: {Trace Level. CurrentTime: }
        Appended the literal string
        AppendFormatted called: {10/20/2021 12:19:10 PM} is of type
System.DateTime
        Appended the formatted object
        AppendLiteral called: {. This won't be printed.}
        Appended the literal string
Warning Level. This warning is a string, not an interpolated string
expression.
```

Rastreando a saída, você pode ver como o compilador adiciona código para chamar o manipulador e criar a cadeia de caracteres:

- O compilador adiciona uma chamada para construir o manipulador, passando o comprimento total do texto literal na cadeia de caracteres de formato e o número de espaços reservados.
- O compilador adiciona chamadas a `AppendLiteral` e `AppendFormatted` para cada seção da cadeia de caracteres literal e para cada espaço reservado.
- O compilador invoca o método `LogMessage` usando o `CoreInterpolatedStringHandler` como argumento.

Por fim, observe que o último aviso não invoca o manipulador de cadeia de caracteres interpolada. O argumento é um `string`, de modo que a chamada invoca a outra sobrecarga com um parâmetro de cadeia de caracteres.

> **ⓘ Importante**
>
> A versão do `Logger` desta seção é um `ref struct`. Uma `ref struct` minimiza as alocações porque ela deve ser armazenada na pilha. No entanto, tipos `ref struct`, em geral, não podem implementar interfaces. Isso pode causar problemas de compatibilidade para frameworks de teste de unidade e tipos de simulação que usam tipos de `ref struct` para implementação.

Adicionar mais recursos ao manipulador

A versão anterior do manipulador de cadeia de caracteres interpolada implementa o padrão. Para evitar o processamento de cada expressão de marcador de posição, você precisa de mais informações no gerenciador. Nesta seção, você aprimora o manipulador para que ele funcione menos quando a cadeia de caracteres construída não for gravada no log. Você usa

System.Runtime.CompilerServices.InterpolatedStringHandlerArgumentAttribute para especificar um mapeamento entre parâmetros para uma API pública e parâmetros para o construtor de um manipulador. Isso fornece ao manipulador as informações necessárias para determinar se a cadeia de caracteres interpolada deve ser avaliada.

Vamos começar com alterações no Manipulador. Primeiro, adicione um campo para acompanhar se o manipulador estiver habilitado. Adicione dois parâmetros ao construtor: um para especificar o nível de log dessa mensagem e o outro uma referência ao objeto de log:

C#

```
private readonly bool enabled;

public LogInterpolatedStringHandler(int literalLength, int formattedCount,
Logger logger, LogLevel logLevel)
{
    enabled = logger.EnabledLevel >= logLevel;
    builder = new StringBuilder(literalLength);
    Console.WriteLine($"\tliteral length: {literalLength}, formattedCount:
{formattedCount}");
}
```

Em seguida, use o campo para que o manipulador só acrescente literais ou objetos formatados quando a cadeia de caracteres final for usada:

C#

```
public void AppendLiteral(string s)
{
    Console.WriteLine($"\tAppendLiteral called: {{{s}}}");
    if (!enabled) return;

    builder.Append(s);
    Console.WriteLine($"\tAppended the literal string");
}

public void AppendFormatted<T>(T t)
{
    Console.WriteLine($"\tAppendFormatted called: {{{t}}} is of type
{typeof(T)}");
    if (!enabled) return;

    builder.Append(t?.ToString());
    Console.WriteLine($"\tAppended the formatted object");
}
```

Em seguida, você precisa atualizar a declaração `LogMessage` para que o compilador passe os parâmetros adicionais para o construtor do manipulador. Isso é tratado usando o System.Runtime.CompilerServices.InterpolatedStringHandlerArgumentAttribute no argumento do manipulador:

C#

```
public void LogMessage(LogLevel level,
[InterpolatedStringHandlerArgument("", "level")]
LogInterpolatedStringHandler builder)
{
    if (EnabledLevel < level) return;
    Console.WriteLine(builder.GetFormattedText());
}
```

Esse atributo especifica a lista de argumentos para `LogMessage` que são mapeados para os parâmetros que seguem os parâmetros obrigatórios `literalLength` e `formattedCount`. A cadeia de caracteres vazia ("), especifica o receptor. O compilador substitui o valor do objeto `Logger` representado pelo `this` pelo argumento seguinte para o construtor do manipulador. O compilador substitui o valor de `level` para o argumento a seguir. Você pode fornecer qualquer número de argumentos para qualquer manipulador que escrever. Os argumentos que você adiciona são argumentos de cadeia de caracteres.

Você pode executar essa versão usando o mesmo código de teste. Desta vez, você verá os seguintes resultados:

PowerShell

```
        literal length: 65, formattedCount: 1
        AppendLiteral called: {Error Level. CurrentTime: }
        Appended the literal string
        AppendFormatted called: {10/20/2021 12:19:10 PM} is of type
System.DateTime
        Appended the formatted object
        AppendLiteral called: {. This is an error. It will be printed.}
        Appended the literal string
Error Level. CurrentTime: 10/20/2021 12:19:10 PM. This is an error. It will
be printed.
        literal length: 50, formattedCount: 1
        AppendLiteral called: {Trace Level. CurrentTime: }
        AppendFormatted called: {10/20/2021 12:19:10 PM} is of type
System.DateTime
        AppendLiteral called: {. This won't be printed.}
Warning Level. This warning is a string, not an interpolated string
expression.
```

Você pode ver que os métodos `AppendLiteral` e `AppendFormat` estão sendo chamados, mas eles não estão realizando nenhuma tarefa. O manipulador determinou que a cadeia de caracteres final não é necessária, portanto, o manipulador não a cria. Ainda há algumas melhorias a serem feitos.

Primeiro, você pode adicionar uma sobrecarga do `AppendFormatted` para restringir o argumento a um tipo que implementa System.IFormattable. Essa sobrecarga permite que os chamadores adicionem cadeias de caracteres de formato nos espaços reservados. Ao fazer essa alteração, também vamos alterar o tipo de retorno dos outros métodos `AppendFormatted` e `AppendLiteral`, de `void` para `bool` (se algum desses métodos tiver tipos de retorno diferentes, você receberá um erro de compilação). Essa alteração permite *curto-circuito*. Os métodos retornam `false` para indicar que o processamento da expressão de cadeia de caracteres interpolada deve ser interrompido. Retornar `true` indica que ele deve continuar. Neste exemplo, você está usando-o para interromper o processamento quando a cadeia de caracteres resultante não for necessária. O curto-circuito dá suporte a ações mais refinadas. Você pode parar de processar a expressão quando ela atingir um determinado comprimento, para dar suporte a buffers de comprimento fixo. Ou alguma condição pode indicar que os elementos restantes não são necessários.

C#

```
public void AppendFormatted<T>(T t, string format) where T : IFormattable
{
    Console.WriteLine($"\tAppendFormatted (IFormattable version) called: {t}
with format {{{format}}} is of type {typeof(T)},");

    builder.Append(t?.ToString(format, null));
    Console.WriteLine($"\tAppended the formatted object");
}
```

Com essa adição, você pode especificar cadeias de caracteres de formato em sua expressão de cadeia de caracteres interpolada:

C#

```
var time = DateTime.Now;

logger.LogMessage(LogLevel.Error, $"Error Level. CurrentTime: {time}. The
time doesn't use formatting.");
logger.LogMessage(LogLevel.Error, $"Error Level. CurrentTime: {time:t}. This
is an error. It will be printed.");
logger.LogMessage(LogLevel.Trace, $"Trace Level. CurrentTime: {time:t}. This
won't be printed.");
```

O `:t` na primeira mensagem especifica o "formato de hora curto" para a hora atual. O exemplo anterior mostrou uma das sobrecargas para o método `AppendFormatted` que você pode criar para o manipulador. Você não precisa especificar um argumento genérico para o objeto que está sendo formatado. Você pode ter maneiras mais eficientes de converter tipos que você cria em string. Você pode escrever sobrecargas de `AppendFormatted` que adota esses tipos em vez de um argumento genérico. O compilador escolhe a melhor sobrecarga. O runtime usa essa técnica para converter System.Span<T> em saída de cadeia de caracteres. Você pode adicionar um parâmetro inteiro para especificar o alinhamento da saída, com ou sem um IFormattable. O System.Runtime.CompilerServices.DefaultInterpolatedStringHandler que é fornecido com o .NET 6 contém nove sobrecargas de AppendFormatted para usos diferentes. Você pode usá-lo como referência ao criar um manipulador para suas finalidades.

Execute o exemplo agora e você verá que, para a mensagem `Trace`, apenas o primeiro `AppendLiteral` é chamado:

PowerShell

```
        literal length: 60, formattedCount: 1
        AppendLiteral called: Error Level. CurrentTime:
        Appended the literal string
        AppendFormatted called: 10/20/2021 12:18:29 PM is of type
System.DateTime
        Appended the formatted object
        AppendLiteral called: . The time doesn't use formatting.
        Appended the literal string
Error Level. CurrentTime: 10/20/2021 12:18:29 PM. The time doesn't use
formatting.
        literal length: 65, formattedCount: 1
        AppendLiteral called: Error Level. CurrentTime:
        Appended the literal string
        AppendFormatted (IFormattable version) called: 10/20/2021 12:18:29
PM with format {t} is of type System.DateTime,
        Appended the formatted object
        AppendLiteral called: . This is an error. It will be printed.
        Appended the literal string
Error Level. CurrentTime: 12:18 PM. This is an error. It will be printed.
        literal length: 50, formattedCount: 1
        AppendLiteral called: Trace Level. CurrentTime:
Warning Level. This warning is a string, not an interpolated string
expression.
```

Você pode fazer uma atualização final para o construtor do manipulador que melhora a eficiência. O manipulador pode adicionar um parâmetro de `out bool` final. Definir esse parâmetro como `false` indica que o manipulador não deve ser chamado para processar a expressão de cadeia de caracteres interpolada:

C#

```
public LogInterpolatedStringHandler(int literalLength, int formattedCount,
Logger logger, LogLevel level, out bool isEnabled)
{
    isEnabled = logger.EnabledLevel >= level;
    Console.WriteLine($"\tliteral length: {literalLength}, formattedCount:
{formattedCount}");
    builder = isEnabled ? new StringBuilder(literalLength) : default!;
}
```

Essa alteração significa que você pode remover o campo `enabled`. Em seguida, você pode alterar o tipo de retorno de `AppendLiteral` e `AppendFormatted` para `void`. Agora, ao executar o exemplo, você verá a seguinte saída:

PowerShell

```
        literal length: 60, formattedCount: 1
        AppendLiteral called: Error Level. CurrentTime:
        Appended the literal string
        AppendFormatted called: 10/20/2021 12:19:10 PM is of type
System.DateTime
        Appended the formatted object
        AppendLiteral called: . The time doesn't use formatting.
        Appended the literal string
Error Level. CurrentTime: 10/20/2021 12:19:10 PM. The time doesn't use
formatting.
        literal length: 65, formattedCount: 1
        AppendLiteral called: Error Level. CurrentTime:
        Appended the literal string
        AppendFormatted (IFormattable version) called: 10/20/2021 12:19:10
PM with format {t} is of type System.DateTime,
        Appended the formatted object
        AppendLiteral called: . This is an error. It will be printed.
        Appended the literal string
Error Level. CurrentTime: 12:19 PM. This is an error. It will be printed.
        literal length: 50, formattedCount: 1
Warning Level. This warning is a string, not an interpolated string
expression.
```

A única saída quando `LogLevel.Trace` foi especificado é a saída do construtor. O manipulador indicou que ele não está habilitado, portanto, nenhum dos métodos de `Append` foi invocado.

Este exemplo ilustra um ponto importante para manipuladores de cadeias de caracteres interpoladas, especialmente quando bibliotecas de log são usadas. Quaisquer efeitos colaterais nos espaços reservados podem não ocorrer. Adicione o seguinte código ao seu programa principal e veja esse comportamento em ação:

C#

```
int index = 0;
int numberOfIncrements = 0;
for (var level = LogLevel.Critical; level <= LogLevel.Trace; level++)
{
    Console.WriteLine(level);
    logger.LogMessage(level, $"{level}: Increment index a few times
{index++}, {index++}, {index++}, {index++}, {index++}");
    numberOfIncrements += 5;
}
Console.WriteLine($"Value of index {index}, value of numberOfIncrements:
{numberOfIncrements}");
```

Você pode ver que a variável `index` é incrementada cinco vezes cada iteração do loop. Como os espaços reservados são avaliados apenas para `Critical`, `Error` e `Warning` níveis, não para `Information` e `Trace`, o valor final de `index` não corresponde à expectativa:

PowerShell

```
Critical
Critical: Increment index a few times 0, 1, 2, 3, 4
Error
Error: Increment index a few times 5, 6, 7, 8, 9
Warning
Warning: Increment index a few times 10, 11, 12, 13, 14
Information
Trace
Value of index 15, value of numberOfIncrements: 25
```

Manipuladores de cadeia de caracteres interpolados fornecem maior controle sobre como uma expressão de cadeia de caracteres interpolada é convertida em uma cadeia de caracteres. A equipe de runtime do .NET usou esse recurso para melhorar o desempenho em várias áreas. Você pode usar a mesma funcionalidade em suas próprias bibliotecas. Para explorar mais, confira System.Runtime.CompilerServices.DefaultInterpolatedStringHandler. Ele fornece uma implementação mais completa do que você criou aqui. Você verá muitas outras sobrecargas possíveis para os métodos `Append`.

O SDK da Plataforma do Compilador do .NET

19/06/2025

Os compiladores criam um modelo detalhado de código de aplicativo à medida que validam a sintaxe e a semântica desse código. Eles usam esse modelo para criar a saída executável a partir do código-fonte. O SDK do .NET Compiler Platform fornece acesso a esse modelo. Cada vez mais, contamos com recursos de IDE (ambiente de desenvolvimento integrado), como IntelliSense, refatoração, renomeação inteligente, "Localizar todas as referências" e "Ir para definição" para aumentar nossa produtividade. Contamos com ferramentas de análise de código para melhorar nossa qualidade de código e geradores de código para ajudar na construção do aplicativo. À medida que essas ferramentas ficam mais inteligentes, elas precisam de acesso a cada vez mais do modelo que só os compiladores criam à medida que processam o código do aplicativo. Esta é a principal missão das APIs roslyn: abrir as caixas opacas e permitir que ferramentas e usuários finais compartilhem a riqueza de informações que os compiladores têm sobre nosso código. Em vez de serem tradutores opacos de código-fonte para código-objeto, por meio do Roslyn, os compiladores se tornam plataformas: APIs que podem ser utilizadas para tarefas relacionadas ao código em suas ferramentas e aplicativos.

Conceitos do SDK do .NET Compiler Platform

O SDK do .NET Compiler Platform reduz drasticamente a barreira à entrada para a criação de ferramentas e aplicativos focados em código. Ele cria muitas oportunidades de inovação em áreas como meta-programação, geração e transformação de código, uso interativo das linguagens C# e Visual Basic e inserção de C# e Visual Basic em linguagens específicas do domínio.

O SDK do .NET Compiler Platform permite que você crie ***analisadores*** e ***correções de código*** que encontrem e corrija erros de codificação. ***Os analisadores*** entendem a sintaxe (estrutura de código) e a semântica para detectar práticas que devem ser corrigidas. ***As correções de*** código fornecem uma ou mais correções sugeridas para lidar com erros de codificação encontrados por analisadores ou diagnóstico de compilador. Normalmente, um analisador e as correções de código associadas são empacotadas em um único projeto.

Analisadores e correções de código usam a análise estática para entender o código. Eles não executam o código ou fornecem outros benefícios de teste. Eles podem, no entanto, apontar práticas que geralmente levam a bugs, código inintencável ou violação de diretriz padrão.

Além de analisadores e correções de código, o SDK da Plataforma do Compilador .NET também permite criar ***refatorações de código***. Ele também fornece um único conjunto de APIs que permitem examinar e entender uma base de código C# ou Visual Basic. Como você pode usar essa base de código única, você pode escrever analisadores e correções de código com mais facilidade aproveitando as APIs de análise sintactica e semântica fornecidas pelo SDK da Plataforma do Compilador .NET. Liberado da tarefa grande de replicar a análise feita pelo compilador, você pode se concentrar na tarefa mais focada de localizar e corrigir erros comuns de codificação para seu projeto ou biblioteca.

Um benefício menor é que seus analisadores e correções de código são menores e usam muito menos memória quando carregados no Visual Studio do que se você escrevesse sua própria base de código para entender o código em um projeto. Aproveitando as mesmas classes usadas pelo compilador e pelo Visual Studio, você pode criar suas próprias ferramentas de análise estática. Isso significa que sua equipe pode usar analisadores e correções de código sem um impacto perceptível no desempenho do IDE.

Há três cenários principais para escrever analisadores e correções de código:

1. *Impor padrões de codificação de equipe*
2. *Fornecer diretrizes sobre pacotes de biblioteca*
3. *Fornecer diretrizes gerais*

Impor padrões de codificação de equipe

Muitas equipes têm padrões de codificação que são impostos por meio de revisões de código com outros membros da equipe. Analisadores e correções de código podem tornar esse processo muito mais eficiente. As revisões de código ocorrem quando um desenvolvedor compartilha seu trabalho com outras pessoas na equipe. O desenvolvedor terá investido todo o tempo necessário para concluir um novo recurso antes de receber comentários. Semanas podem passar enquanto o desenvolvedor reforça hábitos que não correspondem às práticas da equipe.

Os analisadores são executados à medida que um desenvolvedor escreve o código. O desenvolvedor recebe comentários imediatos que incentivam a seguir as diretrizes imediatamente. O desenvolvedor cria hábitos para escrever código em conformidade assim que começar a criar protótipos. Quando o recurso estiver pronto para revisão pelos seres humanos, todas as diretrizes padrão terão sido implementadas.

As equipes podem criar analisadores e correções de código que procurem as práticas mais comuns que violem as práticas de codificação em equipe. Elas podem ser instaladas no computador de cada desenvolvedor para impor os padrões.

> **Dica**
>
> Antes de criar seu próprio analisador, confira os analisadores integrados. Para obter mais informações, consulte regras de estilo de código.

Fornecer diretrizes com pacotes de biblioteca

Há uma grande variedade de bibliotecas disponíveis para desenvolvedores do .NET no NuGet. Alguns deles vêm da Microsoft, alguns de empresas terceirizadas e outros de membros da comunidade e voluntários. Essas bibliotecas recebem mais adoção e avaliações mais altas quando os desenvolvedores podem ter êxito com essas bibliotecas.

Além de fornecer documentação, você pode fornecer analisadores e correções de código que encontram e corrigem usos incorretos comuns de sua biblioteca. Essas correções imediatas ajudarão os desenvolvedores a ter êxito mais rapidamente.

Você pode empacotar analisadores e correções de código com sua biblioteca no NuGet. Nesse cenário, todos os desenvolvedores que instalam seu pacote NuGet também instalarão o pacote do analisador. Todos os desenvolvedores que usam sua biblioteca receberão imediatamente orientações de sua equipe na forma de comentários imediatos sobre erros e correções sugeridas.

Fornecer diretrizes gerais

A comunidade de desenvolvedores do .NET descobriu, por meio da experiência, padrões que funcionam bem e padrões que são melhor evitados. Vários membros da comunidade criaram analisadores que impõem esses padrões recomendados. À medida que aprendemos mais, sempre há espaço para novas ideias.

Esses analisadores podem ser carregados no Visual Studio Marketplace e baixados por desenvolvedores usando o Visual Studio. Os recém-chegados ao idioma e à plataforma aprendem rapidamente as práticas aceitas e tornam-se produtivos no início de sua jornada do .NET. À medida que elas se tornam mais amplamente utilizadas, a comunidade adota essas práticas.

Geradores de origem

Os geradores de origem visam permitir a *metaprogramação em tempo de compilação*, ou seja, código que pode ser criado em tempo de compilação e adicionado à compilação. Os geradores de origem podem ler o conteúdo da compilação antes de serem executados, bem

como acessar *arquivos adicionais*. Essa capacidade permite que eles introspectem o código C# do usuário e os arquivos específicos do gerador. Você pode aprender a criar geradores de origem incrementais usando o livro de receitas do gerador de origem.

Próximas etapas

O SDK do .NET Compiler Platform inclui os modelos de objeto de linguagem mais recentes para geração, análise e refatoração de código. Esta seção fornece uma visão geral conceitual do SDK do .NET Compiler Platform. Mais detalhes podem ser encontrados nas seções de guias de início rápido, exemplos e tutoriais.

Você pode saber mais sobre os conceitos no SDK do .NET Compiler Platform nestes cinco tópicos:

- Explorar o código com o visualizador de sintaxe
- Entender o modelo de API do compilador
- Trabalhar com sintaxe
- Trabalhar com semântica
- Trabalhar com um espaço de trabalho

Para começar, você precisará instalar o **SDK do .NET Compiler Platform**:

Instruções de instalação – Instalador do Visual Studio

Há duas maneiras diferentes de encontrar o **SDK do .NET Compiler Platform** no **Instalador do Visual Studio**:

Instalar usando o instalador do Visual Studio – exibição de cargas de trabalho

O SDK do .NET Compiler Platform não é selecionado automaticamente como parte da carga de trabalho de desenvolvimento de extensão do Visual Studio. Você deve selecioná-lo como um componente opcional.

1. Executar **o Instalador do Visual Studio**
2. Escolha **Mudar**
3. Verifique a carga **de trabalho de desenvolvimento de extensão do Visual Studio** .
4. Abra o nó **Desenvolvimento de extensão do Visual Studio** na árvore de resumo.
5. Marque a caixa do **SDK da .NET Compiler Platform**. Você o encontrará por último nos componentes opcionais.

Opcionalmente, você também desejará que o **editor DGML** exiba grafos no visualizador:

1. Abra o nó **Componentes individuais** na árvore de resumo.
2. Marque a caixa do **Editor DGML**

Instalar usando a guia Instalador do Visual Studio – Componentes individuais

1. Executar **o Instalador do Visual Studio**
2. Escolha **Mudar**
3. Selecione a guia **Componentes individuais**
4. Marque a caixa do **SDK da .NET Compiler Platform**. Você o encontrará na parte superior na seção **Compiladores, ferramentas de build e runtimes** .

Opcionalmente, você também desejará que o **editor DGML** exiba grafos no visualizador:

1. Marque a caixa do **Editor DGML**. Você o encontrará na seção **Ferramentas de código** .

Entender o modelo do SDK do .NET Compiler Platform

19/06/2025

Os compiladores processam o código que você escreve seguindo regras estruturadas que geralmente diferem da maneira como os humanos leem e entendem o código. Uma compreensão básica do modelo usado pelos compiladores é essencial para entender as APIs que você usa ao criar ferramentas baseadas em Roslyn.

Áreas funcionais do pipeline do compilador

O SDK do .NET Compiler Platform expõe a análise de código dos compiladores C# e Visual Basic para você como consumidor, fornecendo uma camada de API que espelha um pipeline de compilador tradicional.

Cada fase desse fluxo de trabalho é um componente separado. Primeiro, a fase de análise cria tokens do texto de origem e o analisa na sintaxe que segue a gramática da linguagem. Em segundo lugar, a fase de declaração analisa a origem e os metadados importados para formar símbolos nomeados. Em seguida, a fase de associação corresponde os identificadores no código aos símbolos. Por fim, a fase de emissão emite um assembly com todas as informações criadas pelo compilador.

Correspondente a cada uma dessas fases, o SDK da Plataforma do Compilador .NET expõe um modelo de objeto que permite o acesso às informações nessa fase. A fase de análise expõe uma árvore de sintaxe, a fase de declaração expõe uma tabela de símbolos hierárquica, a fase

de associação expõe o resultado da análise semântica do compilador e a fase de emissão é uma API que produz códigos de bytes IL.

Cada compilador combina esses componentes como um único todo de ponta a ponta.

Essas APIs são as mesmas usadas pelo Visual Studio. Por exemplo, os recursos de estrutura e formatação de código usam as árvores de sintaxe, o **Pesquisador de Objetos** e os recursos de navegação usam a tabela de símbolos, as refatorações e **Ir para Definição** usam o modelo semântico, e **Editar e Continuar** usa todos esses, incluindo a Emit API.

Camadas de API

O SDK do compilador .NET consiste em várias camadas de APIs: APIs do compilador, APIs de diagnóstico, APIs de script e APIs de workspaces.

APIs de Compilador

A camada do compilador contém os modelos de objeto que correspondem às informações expostas em cada fase do pipeline do compilador, sintactica e semântica. A camada do compilador também contém um instantâneo imutável de uma única invocação de um compilador, incluindo referências de assembly, opções do compilador e arquivos de código-fonte. Há duas APIs distintas que representam a linguagem C# e a linguagem Visual Basic. As duas APIs são semelhantes em forma, mas personalizadas para garantir alta fidelidade a cada idioma. Essa camada não tem dependências em componentes do Visual Studio.

APIs de diagnóstico

Como parte de sua análise, o compilador pode produzir um conjunto de diagnósticos que abrange tudo, desde sintaxe, semântica e erros de atribuição definidos até vários avisos e diagnóstico informativo. A camada de API do Compilador expõe o diagnóstico por meio de uma API extensível que permite que analisadores definidos pelo usuário sejam conectados ao processo de compilação. Ele permite que diagnósticos definidos pelo usuário, como aqueles produzidos por ferramentas como o StyleCop, sejam produzidos junto com o diagnóstico definido pelo compilador. Essa forma de produção de diagnóstico tem o benefício da integração natural a ferramentas como o MSBuild e o Visual Studio, que dependem do diagnóstico para experiências como interrupção de um build com base na política, exibição de textos sublinhados em tempo real no editor e sugestão de correções de código.

APIs de script

As APIs de hospedagem e script são criadas sobre a camada do compilador. Você pode usar as APIs de script para executar snippets de código e acumular um contexto de execução de runtime. O REPL (Loop de Leitura-Avaliação-Impressão) interativo do C# usa essas APIs. O REPL permite que você use C# como uma linguagem de script, executando o código interativamente enquanto você o escreve.

APIs dos workspaces

A camada Workspaces contém a API do Workspace, que é o ponto de partida para fazer análise de código e refatoração em soluções inteiras. Ele ajuda você a organizar todas as informações sobre os projetos em uma solução em um único modelo de objeto, oferecendo acesso direto aos modelos de objeto de camada do compilador sem a necessidade de analisar arquivos, configurar opções ou gerenciar dependências projeto a projeto.

Além disso, a camada Workspaces apresenta um conjunto de APIs usadas ao implementar ferramentas de análise de código e refatoração que funcionam em um ambiente de host como o IDE do Visual Studio. Os exemplos incluem as APIs Localizar Todas as Referências, Formatação e Geração de Código.

Essa camada não tem dependências em componentes do Visual Studio.

Trabalhar com sintaxe

19/06/2025

A *árvore de sintaxe* é uma estrutura de dados imutável fundamental exposta pelas APIs do compilador. Essas árvores representam a estrutura lexical e sintactica do código-fonte. Eles servem a dois propósitos importantes:

- Para permitir que ferramentas - como um IDE, suplementos, ferramentas de análise de código e refatorações - vejam e processem a estrutura sintactica do código-fonte no projeto de um usuário.
- Para permitir que ferramentas, como uma IDE e processos de refatoração, criem, modifiquem e reorganizem o código-fonte de maneira natural, sem a necessidade de usar edições diretas de texto. Ao criar e manipular árvores, as ferramentas podem criar e reorganizar facilmente o código-fonte.

Árvores de sintaxe

As árvores de sintaxe são a estrutura primária usada para compilação, análise de código, associação, refatoração, recursos de IDE e geração de código. Nenhuma parte do código-fonte é compreendida sem que primeiro seja identificada e categorizada em um dos muitos elementos de linguagem estrutural conhecidos.

> **Observação**
>
> RoslynQuoter é uma ferramenta de software livre que mostra as chamadas à API de fábrica de sintaxe usadas para construir uma árvore de sintaxe de um programa. Para experimentá-lo ao vivo, veja http://roslynquoter.azurewebsites.net .

As árvores de sintaxe têm três atributos principais:

- Eles mantêm todas as informações de origem em total fidelidade. A fidelidade total significa que a árvore de sintaxe contém todas as informações encontradas no texto de origem, cada construção gramatical, cada token léxico e tudo mais no meio, incluindo espaço em branco, comentários e diretivas de pré-processador. Por exemplo, cada literal mencionado na origem é representado exatamente como foi digitado. As árvores de sintaxe também capturam erros no código-fonte quando o programa está incompleto ou malformado, representando tokens ignorados ou ausentes.
- Elas podem produzir o texto exato do qual foram analisados. Em qualquer nó de sintaxe, é possível obter a representação de texto da subárvore enraizada nesse nó. Essa capacidade significa que as árvores de sintaxe podem ser usadas como uma maneira de

construir e editar o texto de origem. Ao criar uma árvore, você criou, por implicação, o texto equivalente e, ao fazer uma nova árvore com alterações em uma árvore existente, você editou efetivamente o texto.
- Eles são imutáveis e thread-safe. Depois que uma árvore é obtida, ela é um instantâneo do estado atual do código e nunca é alterada. Isso permite que vários usuários interajam com a mesma árvore de sintaxe ao mesmo tempo em threads diferentes sem bloqueio ou duplicação. Como as árvores são imutáveis e nenhuma modificação pode ser feita diretamente em uma árvore, os métodos de fábrica ajudam a criar e modificar árvores de sintaxe criando instantâneos adicionais da árvore. As árvores são eficientes na maneira como reutilizam nós subjacentes, de modo que uma nova versão pode ser reconstruída rapidamente e com pouca memória extra.

Uma árvore de sintaxe é literalmente uma estrutura de dados de árvore, em que elementos estruturais não terminais pairam outros elementos. Cada árvore de sintaxe é composta por nós, tokens e desafios.

Nós de sintaxe

Os nós de sintaxe são um dos principais elementos das árvores de sintaxe. Esses nós representam construções sintáticas, como declarações, comandos, cláusulas e expressões. Cada categoria de nós de sintaxe é representada por uma classe separada derivada de Microsoft.CodeAnalysis.SyntaxNode. O conjunto de classes de nó não é extensível.

Todos os nós de sintaxe são nós não terminais na árvore de sintaxe, o que significa que eles sempre têm outros nós e tokens como filhos. Como filho de outro nó, cada nó tem um nó pai que pode ser acessado por meio da propriedade SyntaxNode.Parent. Como os nós e as árvores são imutáveis, o pai de um nó nunca é alterado. A raiz da árvore tem um pai nulo.

Cada nó tem um método SyntaxNode.ChildNodes(), que retorna uma lista de nós filho em ordem sequencial com base em sua posição no texto de origem. Esta lista não contém tokens. Cada nó também tem métodos para examinar os Descendentes, como DescendantNodes, DescendantTokens ou DescendantTrivia – que representam uma lista de todos os nós, tokens ou desafios, que existem na subárvore com raiz nesse nó.

Além disso, cada subclasse de nó de sintaxe expõe os mesmos filhos por meio de propriedades fortemente tipadas. Por exemplo, uma classe de nó BinaryExpressionSyntax tem três propriedades adicionais específicas aos operadores binários: Left, OperatorToken e Right. O tipo de Left e Right é ExpressionSyntax, e o tipo de OperatorToken é SyntaxToken.

Alguns nós de sintaxe têm filhos opcionais. Por exemplo, um IfStatementSyntax tem um opcional ElseClauseSyntax. Se o filho não estiver presente, a propriedade retornará nulo.

Tokens de sintaxe

Tokens de sintaxe são os terminais da gramática de linguagem, representando os menores fragmentos sintacticos do código. Eles nunca são os pais de outros nós ou tokens. Os tokens de sintaxe consistem em palavras-chave, identificadores, literais e pontuação.

Para fins de eficiência, o SyntaxToken tipo é um tipo de valor CLR. Portanto, ao contrário dos nós de sintaxe, há apenas uma estrutura para todos os tipos de tokens com uma combinação de propriedades que têm significado dependendo do tipo de token que está sendo representado.

Por exemplo, um token literal inteiro representa um valor numérico. Além do texto de origem não processado abrangido pelo token, o token literal tem uma propriedade Value que informa o valor inteiro decodificado exato. Essa propriedade é tipada como Object porque pode ser um dos muitos tipos primitivos.

A ValueText propriedade informa as mesmas informações que a Value propriedade; no entanto, essa propriedade sempre é digitada como String. Um identificador no texto de origem em C# pode incluir caracteres de escape Unicode, mas a sintaxe da sequência de escape em si não é considerada parte do nome do identificador. Portanto, embora o texto bruto abrangido pelo token inclua a sequência de escape, a propriedade ValueText não. Em vez disso, ela inclui os caracteres Unicode identificados pelo escape. Por exemplo, se o texto de origem contiver um identificador escrito como `\u03C0`, a ValueText propriedade desse token retornará `π`.

Curiosidades de sintaxe

As curiosidades de sintaxe representam as partes do texto de origem que são em grande parte insignificantes para a compreensão normal do código, como espaço em branco, comentários e diretivas de pré-processador. Assim como os tokens de sintaxe, as curiosidades são tipos de valor. O tipo único Microsoft.CodeAnalysis.SyntaxTrivia é usado para descrever todos os tipos de curiosidades.

Como os desafios não fazem parte da sintaxe de linguagem normal e podem aparecer em qualquer lugar entre dois tokens quaisquer, eles não são incluídos na árvore de sintaxe como um filho de um nó. No entanto, como eles são importantes ao implementar um recurso como refatoração e manter a fidelidade total com o texto de origem, eles existem como parte da árvore de sintaxe.

Você pode acessar curiosidades inspecionando as coleções SyntaxToken.LeadingTrivia ou SyntaxToken.TrailingTrivia de um token. Quando o texto de origem é analisado, sequências de curiosidades são associadas a tokens. Em geral, um token possui qualquer desafio após ele na mesma linha até o próximo token. Todas as curiosidades após essa linha são associadas ao

token a seguir. O primeiro token no arquivo de origem obtém todos as desafios iniciais e a última sequência de desafios no arquivo é anexada ao token de fim do arquivo, que, de outro modo, tem largura zero.

Ao contrário dos nós e tokens de sintaxe, os desafios de sintaxe não têm pais. No entanto, como eles fazem parte da árvore e cada um está associado a um único token, você pode acessar o token ao qual ele está associado usando a SyntaxTrivia.Token propriedade.

Intervalos

Cada nó, token ou desafio conhece sua posição dentro do texto de origem e o número de caracteres no qual ele consiste. Uma posição de texto é representada como um inteiro de 32 bits, que é um índice `char` baseado em zero. Um TextSpan objeto é a posição inicial e uma contagem de caracteres, ambos representados como inteiros. Se TextSpan tiver um comprimento zero, ele se referirá a um local entre dois caracteres.

Cada nó tem duas propriedades TextSpan: Span e FullSpan.

A propriedade Span é o intervalo de texto do início do primeiro token na subárvore do nó ao final do último token. Esse intervalo não inclui nenhum desafio à esquerda ou à direita.

A propriedade FullSpan é o intervalo de texto que inclui o intervalo normal do nó mais o intervalo de qualquer desafio à esquerda ou à direita.

Por exemplo:

C#

```
      if (x > 3)
      {
||        // this is bad
          |throw new Exception("Not right.");|  // better exception?||
      }
```

O nó de instrução dentro do bloco tem um intervalo indicado pelas barras verticais simples (|). Ele inclui os caracteres `throw new Exception("Not right.");`. O intervalo completo é indicado pelas barras verticais duplas (||). Ele inclui os mesmos caracteres do intervalo e os caracteres associados ao desafio à esquerda e à direita.

Tipos

Cada nó, token ou desafio tem uma propriedade SyntaxNode.RawKind, do tipo System.Int32, que identifica o elemento de sintaxe exato representado. Esse valor pode ser convertido em

uma enumeração específica do idioma. Cada linguagem, C# ou Visual Basic, tem uma única `SyntaxKind` enumeração (Microsoft.CodeAnalysis.CSharp.SyntaxKind e Microsoft.CodeAnalysis.VisualBasic.SyntaxKind, respectivamente) que lista todos os possíveis nós, tokens e elementos triviais na gramática. Essa conversão pode ser feita automaticamente acessando os métodos de extensão CSharpExtensions.Kind ou VisualBasicExtensions.Kind.

A propriedade RawKind permite a desambiguidade fácil de tipos de nó de sintaxe que compartilham a mesma classe de nó. Para tokens e curiosidades, essa propriedade é a única maneira de distinguir um tipo de elemento de outro.

Por exemplo, uma única BinaryExpressionSyntax classe tem Left, OperatorTokene Right como crianças. A propriedade Kind distingue se ela é um tipo AddExpression, SubtractExpression ou MultiplyExpression de nó de sintaxe.

> **Dica**
>
> É recomendável verificar os tipos usando métodos de extensão **IsKind** (para C#) ou **IsKind** (para VB).

Erros

Mesmo quando o texto de origem contém erros de sintaxe, uma árvore de sintaxe completa que é reversível para a origem é revelada. Quando o analisador encontra um código que não está em conformidade com a sintaxe definida da linguagem, ele usa uma das duas técnicas para criar uma árvore de sintaxe:

- Se o analisador espera um tipo específico de token, mas não o encontra, ele pode inserir um token ausente na árvore de sintaxe no local que o token era esperado. Um token ausente representa o token real esperado, mas ele tem um intervalo vazio e sua SyntaxNode.IsMissing propriedade retorna `true`.

- O analisador pode ignorar tokens até encontrar um em que possa continuar analisando. Nesse caso, os tokens ignorados são anexados como um nó de desafio com o tipo SkippedTokensTrivia.

Trabalhar com semântica

19/06/2025

As árvores de sintaxe representam a estrutura lexical e sintactica do código-fonte. Embora essas informações sejam suficientes para descrever todas as declarações e lógicas na origem, não são informações suficientes para identificar o que está sendo referenciado. Um nome pode representar:

- um tipo
- um campo
- um método
- uma variável local

Embora cada um deles seja exclusivamente diferente, determinar a qual um identificador realmente se refere geralmente requer uma compreensão profunda das regras de linguagem.

Há elementos de programa representados no código-fonte e os programas também podem se referir a bibliotecas compiladas anteriormente, empacotadas em arquivos de assembly. Embora nenhum código-fonte e, portanto, nenhum nó ou árvore de sintaxe, esteja disponível para assemblies, os programas ainda podem se referir a elementos contidos neles.

Para essas tarefas, você precisa do **modelo semântico**.

Além de um modelo sintactico do código-fonte, um modelo semântico encapsula as regras de linguagem, oferecendo uma maneira fácil de corresponder corretamente os identificadores com o elemento de programa correto que está sendo referenciado.

Compilação

Uma compilação é uma representação de tudo o que é necessário para compilar um programa C# ou Visual Basic, que inclui todas as referências de assembly, opções do compilador e arquivos de origem.

Como todas essas informações estão em um só lugar, os elementos contidos no código-fonte podem ser descritos com mais detalhes. A compilação representa cada tipo declarado, membro ou variável como um símbolo. A compilação contém uma variedade de métodos que ajudam a localizar e relacionar os símbolos que foram declarados no código-fonte ou importados como metadados de um assembly.

Semelhantes às árvores de sintaxe, as compilações são imutáveis. Depois de criar uma compilação, ela não pode ser alterada por você ou por qualquer outra pessoa com quem você possa compartilhá-la. No entanto, você pode criar uma nova compilação a partir de uma

compilação existente, especificando uma alteração conforme você faz isso. Por exemplo, você pode criar uma compilação que seja a mesma em todos os sentidos que uma compilação existente, exceto que ela pode incluir um arquivo de origem adicional ou uma referência de assembly.

Símbolos

Um símbolo representa um elemento distinto declarado pelo código-fonte ou importado de um assembly como metadados. Cada namespace, tipo, método, propriedade, campo, evento, parâmetro ou variável local é representado por um símbolo.

Uma variedade de métodos e propriedades no Compilation tipo ajudam você a encontrar símbolos. Por exemplo, você pode encontrar um símbolo para um tipo declarado por seu nome de metadados comum. Você também pode acessar a tabela de símbolos inteira como uma árvore de símbolos com raiz pelo namespace global.

Os símbolos também contêm informações adicionais que o compilador determina da origem ou dos metadados, como outros símbolos referenciados. Cada tipo de símbolo é representado por uma interface separada derivada de ISymbol, cada uma com seus próprios métodos e propriedades detalhando as informações coletadas pelo compilador. Muitas dessas propriedades fazem referência direta a outros símbolos. Por exemplo, a propriedade IMethodSymbol.ReturnType indica o símbolo de tipo real que o método retorna.

Os símbolos apresentam uma representação comum de namespaces, tipos e membros, entre o código-fonte e metadados. Por exemplo, um método que foi declarado no código-fonte e um método que foi importado de metadados são representados por um IMethodSymbol com as mesmas propriedades.

Os símbolos são semelhantes no conceito ao sistema de tipos CLR, conforme representado pela System.Reflection API, mas são mais ricos, pois modelam mais do que apenas tipos. Namespaces, variáveis locais e rótulos são todos símbolos. Além disso, os símbolos são uma representação de conceitos de linguagem, não conceitos clr. Há muita sobreposição, mas há muitas distinções significativas também. Por exemplo, um método iterador em C# ou Visual Basic é um único símbolo. No entanto, quando o método iterador é convertido em metadados CLR, ele é um tipo e vários métodos.

Modelo semântico

Um modelo semântico representa todas as informações semânticas de um único arquivo de origem. Você pode usá-lo para descobrir o seguinte:

- Os símbolos referenciados em uma posição específica na fonte.

- O tipo resultante de qualquer expressão.
- Todo o diagnóstico, que são erros e avisos.
- Como as variáveis fluem dentro e fora de regiões de origem.
- As respostas para perguntas mais especulativas.

Trabalhar com um workspace

19/06/2025

A camada **workspaces** é o ponto de partida para fazer análise de código e refatoração em soluções inteiras. Nessa camada, a API do Workspace ajuda você a organizar todas as informações sobre os projetos em uma solução em um único modelo de objeto, oferecendo acesso direto a modelos de objeto de camada do compilador, como texto de origem, árvores de sintaxe, modelos semânticos e compilações sem a necessidade de analisar arquivos, configurar opções ou gerenciar dependências entre projetos.

Ambientes de host, como uma IDE, fornecem um espaço de trabalho para você, correspondente à solução aberta. Também é possível usar esse modelo fora de um IDE simplesmente carregando um arquivo de solução.

Espaço de trabalho

Um workspace é uma representação ativa da sua solução como uma coleção de projetos, cada um com uma coleção de documentos. Um workspace normalmente está vinculado a um ambiente anfitrião que está mudando constantemente à medida que um usuário digita ou manipula propriedades.

O Workspace fornece acesso ao modelo atual da solução. Quando ocorre uma alteração no ambiente de host, o workspace aciona eventos correspondentes e a propriedade Workspace.CurrentSolution é atualizada. Por exemplo, quando o usuário digita em um editor de texto correspondente a um dos documentos de origem, o workspace usa um evento para sinalizar que o modelo geral da solução foi alterado e qual documento foi modificado. Em seguida, você pode reagir a essas alterações analisando o novo modelo para verificar a correção, destacando áreas significativas ou fazendo uma sugestão para uma modificação de código.

Você também pode criar workspaces independentes desconectados do ambiente de hospedagem ou usados em um aplicativo que não tem ambiente de hospedagem.

Soluções, projetos e documentos

Embora um ambiente de trabalho possa ser alterado sempre que uma tecla é pressionada, você pode trabalhar com o modelo da solução completamente isolado.

Uma solução é um modelo imutável dos projetos e documentos. Isso significa que o modelo pode ser compartilhado sem bloqueio ou duplicação. Depois de obter uma instância da solução da propriedade Workspace.CurrentSolution, essa instância nunca será alterada. No

entanto, como com árvores de sintaxe e compilações, você pode modificar soluções construindo novas instâncias com base em soluções existentes e alterações específicas. Para que o espaço de trabalho reflita suas alterações, você deve aplicar explicitamente a solução modificada de volta ao espaço de trabalho.

Um projeto faz parte do modelo de solução imutável geral. Ele representa todos os documentos do código-fonte, opções de análise e compilação e referências de assembly e projeto a projeto. Em um projeto, você pode acessar a compilação correspondente sem precisar determinar as dependências do projeto ou analisar os arquivos de origem.

Um documento também faz parte do modelo de solução imutável geral. Um documento representa um único arquivo de origem do qual você pode acessar o texto do arquivo, a árvore de sintaxe e o modelo semântico.

O diagrama a seguir é uma representação de como o Workspace se relaciona com o ambiente do host, as ferramentas e como as edições são feitas.

Resumo

Roslyn expõe um conjunto de APIs do compilador e APIs de Workspaces que fornece informações avançadas sobre seu código-fonte e que tem total fidelidade com as linguagens C# e Visual Basic. O SDK do .NET Compiler Platform reduz drasticamente a barreira à entrada para a criação de ferramentas e aplicativos focados em código. Ele cria muitas oportunidades de inovação em áreas como meta-programação, geração e transformação de código, uso interativo das linguagens C# e Visual Basic e inserção de C# e Visual Basic em linguagens específicas do domínio.

Explorar código com o visualizador de sintaxe Roslyn no Visual Studio

Artigo • 10/05/2023

Este artigo oferece uma visão geral sobre a ferramenta de Visualizador de sintaxe que é fornecida como parte do SDK do .NET Compiler Platform ("Roslyn"). O Visualizador de sintaxe é uma janela de ferramentas que ajuda a inspecionar e explorar árvores de sintaxe. É uma ferramenta essencial para compreender os modelos do código que você deseja analisar. Também é um auxílio para depuração ao desenvolver seus próprios aplicativos usando o SDK do.NET Compiler Platform ("Roslyn"). Abra essa ferramenta ao criar seus primeiros analisadores. O visualizador ajuda você a entender os modelos usados pelas APIs. Você também pode usar ferramentas como SharpLab ou LINQPad para inspecionar o código e entender as árvores de sintaxe.

Instruções de instalação – Instalador do Visual Studio

Há duas maneiras diferentes de encontrar o **SDK da .NET Compiler Platform** no **Instalador do Visual Studio**:

Instalar usando o Instalador do Visual Studio – exibição de cargas de trabalho

O SDK da .NET Compiler Platform não é selecionado automaticamente como parte da carga de trabalho de desenvolvimento da extensão do Visual Studio. É necessário selecioná-lo como um componente opcional.

1. Execute o **Instalador do Visual Studio**
2. Escolha **Mudar**
3. Marque a carga de trabalho de **Desenvolvimento de extensão do Visual Studio**.
4. Abra o nó **Desenvolvimento de extensão do Visual Studio** na árvore de resumo.
5. Marque a caixa do **SDK da .NET Compiler Platform**. Você a encontrará por último nos componentes opcionais.

Opcionalmente, você também poderá fazer o **Editor DGML** exibir gráficos no visualizador:

1. Abra o nó **Componentes individuais** na árvore de resumo.
2. Marque a caixa do **Editor DGML**

Instalar usando o Instalador do Visual Studio – guia Componentes individuais

1. Execute o **Instalador do Visual Studio**
2. Escolha **Mudar**
3. Selecione a guia **Componentes individuais**
4. Marque a caixa do **SDK da .NET Compiler Platform**. Você a encontrará na parte superior, na seção **Compiladores, ferramentas de compilação e runtimes**.

Opcionalmente, você também poderá fazer o **Editor DGML** exibir gráficos no visualizador:

1. Marque a caixa do **Editor DGML**. Você a encontrará na seção **Ferramentas de código**.

Familiarize-se com os conceitos usados no SDK do.NET Compiler Platform lendo o artigo de visão geral. Ele fornece uma introdução a árvores de sintaxe, nós, tokens e trívia.

Visualizador de sintaxe

O **Visualizador de sintaxe** permite a inspeção da árvore de sintaxe de arquivo de código C# ou Visual Basic na janela do editor atualmente ativa dentro do IDE do Visual Studio. O visualizador pode ser iniciado ao clicar em **Exibir**>**Outras Janelas**>**Visualizador de sintaxe**. Você também pode usar a barra de ferramentas de **Início Rápido** no canto superior direito. Digite "sintaxe" e o comando para abrir o **Visualizador de sintaxe** deverá aparecer.

Este comando abre o Visualizador de sintaxe como uma janela de ferramentas flutuante. Se você não tiver uma janela de editor de código aberta, a exibição ficará em branco, conforme mostrado na figura a seguir.

Encaixe esta janela de ferramentas em um local conveniente dentro do Visual Studio, como o lado esquerdo. O Visualizador mostra informações sobre o arquivo de código atual.

Crie um novo projeto usando o comando **Arquivo**>**Novo Projeto**. Você pode criar um projeto Visual Basic ou C#. Quando o Visual Studio abre o arquivo de código principal deste projeto, o visualizador exibe a árvore de sintaxe dele. Você pode abrir qualquer arquivo de C# ou Visual Basic existente nesta instância do Visual Studio e o visualizador exibirá a árvore de sintaxe do arquivo correspondente. Se você tiver vários arquivos de código abertos no Visual Studio, o visualizador exibirá a árvore de sintaxe do arquivo de código atualmente ativo, (o arquivo de código que tem o foco do teclado).

C#

Conforme mostrado nas imagens acima, a janela de ferramentas do visualizador exibe a árvore de sintaxe na parte superior e uma grade de propriedade na parte inferior. A grade de propriedade exibe as propriedades do item atualmente selecionado na árvore, incluindo *Type* e *Kind* (SyntaxKind) do .NET do item.

As árvores de sintaxe incluem três tipos de itens: *nós*, *tokens* e *trívia*. Você pode ler mais sobre esses tipos no artigo Trabalhar com sintaxe. Os itens de cada tipo estão representados com cores diferentes. Clique no botão "Legenda" para uma visão geral das cores usadas.

Cada item da árvore também exibe sua própria **extensão**. A **extensão** é composta pelos índices (a posição inicial e final) do nó no arquivo de texto. No exemplo de C# anterior, o token “UsingKeyword [0..5)” selecionado tem uma **abrangência** de cinco caracteres de largura, [0..5). A notação "[..)" significa que o índice inicial faz parte da extensão, mas o índice final não.

Há duas maneiras de navegar na árvore:

- Expandir ou clicar em itens na árvore. O visualizador seleciona automaticamente o texto correspondente à extensão do item no editor de código.
- Clicar ou selecionar texto no editor de código. No exemplo anterior do Visual Basic, se você seleciona a linha que contém "Module Module1" no editor de código, o visualizador navega automaticamente até o nó ModuleStatement correspondente na árvore.

O visualizador realça o item da árvore cuja extensão melhor corresponda com a extensão do texto selecionado no editor.

O visualizador atualiza a árvore para corresponder às modificações no arquivo de código ativo. Adicione uma chamada a `Console.WriteLine()` dentro de `Main()`. Conforme você digita, o visualizador atualiza a árvore.

Pare de digitar quando já tiver digitado `Console.`. A árvore tem alguns itens coloridos em rosa. Neste ponto, há erros (também conhecidos como "Diagnóstico") no código digitado. Esses erros são anexados a nós, tokens e trívia na árvore de sintaxe. O visualizador mostra quais itens têm erros anexados a eles, realçando a tela de fundo em rosa. Você pode inspecionar os erros em qualquer item colorido em rosa passando o mouse sobre o item. O visualizador exibe somente erros sintáticos (os erros relacionados à sintaxe do código digitado); erros semânticos não são exibidos.

Gráficos de sintaxe

Clique com o botão direito do mouse em qualquer item da árvore e clique em **Exibir gráfico de sintaxe direcionado**.

C#

O visualizador exibe uma representação gráfica da subárvore com raiz no item selecionado. Repita essas etapas para o nó **MethodDeclaration** correspondente ao método `Main()` no exemplo de C#. O visualizador exibe um gráfico de sintaxe que tem a seguinte aparência:

O visualizador de gráfico de sintaxe tem uma opção para exibir uma legenda para seu esquema de cores. Você também pode passar o mouse sobre itens específicos no gráfico de sintaxe para exibir as respectivas propriedades.

Você pode exibir gráficos de sintaxe de itens diferentes da árvore repetidamente e os gráficos serão sempre exibidos na mesma janela no Visual Studio. Você pode encaixar essa janela em um local conveniente no Visual Studio para não precisar alternar entre as guias para exibir um novo gráfico de sintaxe. A parte inferior, abaixo das janelas do editor de código, geralmente é conveniente.

Veja o layout de encaixe para usar com a janela de ferramentas do visualizador e a janela do gráfico de sintaxe:

Outra opção é colocar a janela de gráfico de sintaxe em um segundo monitor, em uma configuração de dois monitores.

Inspecionando semântica

O Visualizador de sintaxe possibilita uma inspeção rudimentar de símbolos e informações semânticas. Digite `double x = 1 + 1;` dentro de Main() no exemplo de C#. Em seguida, selecione a expressão `1 + 1` na janela do editor de código. O visualizador realça o nó **AddExpression** no visualizador. Clique com o botão direito do mouse nesse **AddExpression** e clique em **Exibir Symbol (se houver)**. Observe que a maioria dos itens de menu tem o qualificador "se houver". O Visualizador de sintaxe inspeciona as propriedades de um Nó, incluindo propriedades que podem não estar presentes em todos os nós.

A grade de propriedade do visualizador é atualizada conforme mostrado na figura a seguir: o símbolo da expressão é um **SynthesizedIntrinsicOperatorSymbol** com **Kind = Method**.

Experimente **Exibir TypeSymbol (se houver)** para o mesmo nó **AddExpression**. A grade de propriedade no visualizador é atualizada, conforme mostrado na figura a seguir, indicando que o tipo da expressão selecionada é `Int32`.

Experimente **Exibir TypeSymbol Convertido (se houver)** para o mesmo nó **AddExpression**. A grade de propriedade é atualizada, indicando que, embora o tipo da

expressão seja `Int32`, o tipo convertido da expressão é `Double`, conforme mostrado na figura a seguir. Esse nó inclui informações de símbolo de tipo convertido porque a expressão `Int32` ocorre em um contexto em que deve ser convertida em um `Double`. Essa conversão satisfaz o tipo `Double` especificado para a variável `x` no lado esquerdo do operador de atribuição.

Por fim, experimente **Exibir Valor Constante (se houver)** para o mesmo nó **AddExpression**. A grade de propriedade mostra que o valor da expressão é uma constante de tempo de compilação com o valor `2`.

O exemplo anterior também pode ser replicado no Visual Basic. Digite `Dim x As Double = 1 + 1` em um arquivo do Visual Basic. Selecione a expressão `1 + 1` na janela do editor de código. O visualizador realça o nó **AddExpression** correspondente no visualizador. Repita as etapas anteriores para esta **AddExpression** e você verá resultados idênticos.

Examine mais código no Visual Basic. Atualize seu arquivo principal do Visual Basic com o seguinte código:

VB

```
Imports C = System.Console

Module Program
    Sub Main(args As String())
        C.WriteLine()
    End Sub
End Module
```

Esse código introduz um alias chamado `C` que mapeia para o tipo `System.Console` na parte superior do arquivo e usa esse alias em `Main()`. Selecione o uso desse alias, o `C` em `C.WriteLine()`, dentro do método `Main()`. O visualizador seleciona o nó **IdentifierName** correspondente no visualizador. Clique com botão direito do mouse nesse nó e clique em **Exibir Symbol (se houver)**. A grade de propriedade mostra que esse identificador é associado com o tipo `System.Console`, conforme mostrado na figura a seguir:

Experimente **Exibir AliasSymbol (se houver)** para o mesmo nó **IdentifierName**. A grade de propriedade mostra que o identificador é um alias com nome `C`, que é associado com o destino `System.Console`. Em outras palavras, a grade de propriedade fornece informações sobre o **AliasSymbol** correspondente ao identificador `C`.

Inspecione o símbolo correspondente a qualquer tipo, método e propriedade declarados. Selecione o nó correspondente no visualizador e clique em **Exibir Symbol**

(se houver). Selecione o método `Sub Main()`, incluindo o corpo do método. Clique em **Exibir Symbol (se houver)** para o nó **SubBlock** correspondente no visualizador. A grade de propriedade mostra que o **MethodSymbol** deste **SubBlock** tem nome `Main` com o tipo de retorno `Void`.

Os exemplos de Visual Basic acima podem ser facilmente replicados em C#. Digite `using C = System.Console;` no lugar de `Imports C = System.Console` para o alias. As etapas anteriores em C# geram resultados idênticos na janela do visualizador.

As operações de inspeção semântica estão disponíveis somente em nós. Elas não estão disponíveis em tokens nem trívia. Nem todos os nós têm informações semânticas interessantes para inspecionar. Quando um nó não tem informações semânticas interessantes, clicar em **Exibir * Symbol (se houver)** mostrará uma grade de propriedade em branco.

Você pode ler mais sobre as APIs para executar análise semântica no documento de visão geral Trabalhar com semântica.

Fechando o visualizador de sintaxe

Você pode fechar a janela do visualizador quando não a estiver usando para examinar o código-fonte. O visualizador de sintaxe atualiza a exibição enquanto você navega pelo código, editando e alterando a origem. Ele poderá se tornar uma distração quando você não estiver usando.

Escolha os IDs de diagnóstico

19/06/2025

Uma ID de diagnóstico é a cadeia de caracteres associada a um determinado diagnóstico, como um erro do compilador ou um diagnóstico produzido por um analisador.

As IDs são exibidas de várias APIs, como:

- DiagnosticDescriptor.Id
- ObsoleteAttribute.DiagnosticId
- ExperimentalAttribute.DiagnosticId

As IDs de diagnóstico também são usadas como identificadores na origem, por exemplo, em #pragma warning disable ou arquivos .editorconfig.

Considerações

- As IDs de diagnóstico devem ser exclusivas
- As IDs de diagnóstico devem ser identificadores legais em C#
- As IDs de diagnóstico devem ter menos de 15 caracteres
- As IDs de diagnóstico devem ser no formato `<PREFIX><number>`
  - O prefixo é específico para seu projeto
  - O número representa o diagnóstico específico

> **Observação**
>
> É uma alteração de falha de origem para alterar as IDs de diagnóstico, pois as supressões existentes seriam ignoradas se a ID fosse alterada.

Não limite seu prefixo a dois caracteres (como `CSXXX` e `CAXXXX`). Em vez disso, use um prefixo mais longo para evitar conflitos. Por exemplo, o diagnóstico `System.*` usa `SYSLIB` como prefixo.

Introdução à análise de sintaxe

Artigo • 09/05/2023

Neste tutorial, você explorará a **API de sintaxe**. A API de sintaxe fornece acesso às estruturas de dados que descrevem um programa C# ou Visual Basic. Essas estruturas de dados têm detalhes suficientes para que possam representar qualquer programa, de qualquer tamanho. Essas estruturas podem descrever programas completos que compilam e executam corretamente. Elas também podem descrever programas incompletos, enquanto você os escreve no editor.

Para habilitar essa expressão avançada, as estruturas de dados e as APIs que compõem a API de sintaxe são necessariamente complexas. Começaremos com a aparência da estrutura de dados para o programa "Olá, Mundo" típico:

C#

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

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

Veja o texto do programa anterior. Você reconhece elementos familiares. O texto inteiro representa um único arquivo de origem, ou uma **unidade de compilação**. As três primeiras linhas do arquivo de origem são **diretivas de uso**. A origem restante está contida em uma **declaração de namespace**. A declaração de namespace contém uma **declaração de classe** filha. A declaração de classe contém uma **declaração de método**.

A API de sintaxe cria uma estrutura de árvore com a raiz que representa a unidade de compilação. Nós da árvore representam as diretivas using, a declaração de namespace e todos os outros elementos do programa. A estrutura da árvore continua até os níveis mais baixos: a cadeia de caracteres "Olá, Mundo!" é um **token literal de cadeia de caracteres** descendente de um **argumento**. A API de sintaxe fornece acesso à estrutura do programa. Você pode consultar as práticas recomendadas de código específico,

percorrer a árvore inteira para entender o código e criar novas árvores ao modificar a árvore existente.

Essa breve descrição fornece uma visão geral do tipo de informações acessíveis usando a API de sintaxe. A API de sintaxe não é nada mais de uma API formal que descreve os constructos de código familiares que você conhece do C#. As funcionalidades completas incluem informações sobre como o código é formatado, incluindo quebras de linha, espaço em branco e recuo. Usando essas informações, você pode representar totalmente o código como escrito e lido por programadores humanos ou pelo compilador. Usar essa estrutura permite que você interaja com o código-fonte em um nível muito significativo. Não se trata mais de cadeias de caracteres de texto, mas de dados que representam a estrutura de um programa C#.

Para começar, será necessário instalar o **SDK do .NET Compiler Platform**:

Instruções de instalação – Instalador do Visual Studio

Há duas maneiras diferentes de encontrar o **SDK da .NET Compiler Platform** no **Instalador do Visual Studio**:

Instalar usando o Instalador do Visual Studio – exibição de cargas de trabalho

O SDK da .NET Compiler Platform não é selecionado automaticamente como parte da carga de trabalho de desenvolvimento da extensão do Visual Studio. É necessário selecioná-lo como um componente opcional.

1. Execute o **Instalador do Visual Studio**
2. Escolha **Mudar**
3. Marque a carga de trabalho de **Desenvolvimento de extensão do Visual Studio**.
4. Abra o nó **Desenvolvimento de extensão do Visual Studio** na árvore de resumo.
5. Marque a caixa do **SDK da .NET Compiler Platform**. Você a encontrará por último nos componentes opcionais.

Opcionalmente, você também poderá fazer o **Editor DGML** exibir gráficos no visualizador:

1. Abra o nó **Componentes individuais** na árvore de resumo.
2. Marque a caixa do **Editor DGML**

Instalar usando o Instalador do Visual Studio – guia Componentes individuais

1. Execute o **Instalador do Visual Studio**
2. Escolha **Mudar**
3. Selecione a guia **Componentes individuais**
4. Marque a caixa do **SDK da .NET Compiler Platform**. Você a encontrará na parte superior, na seção **Compiladores, ferramentas de compilação e runtimes**.

Opcionalmente, você também poderá fazer o **Editor DGML** exibir gráficos no visualizador:

1. Marque a caixa do **Editor DGML**. Você a encontrará na seção **Ferramentas de código**.

Noções básicas sobre árvores de sintaxe

Você pode usar a API de sintaxe para uma análise da estrutura do código C#. A **API de sintaxe** expõe os analisadores, as árvores de sintaxe e os utilitários para analisar e criar árvores de sintaxe. Trata-se do modo como você pesquisa o código em busca de elementos de sintaxe específicos ou lê o código para um programa.

Uma árvore de sintaxe é uma estrutura de dados usada pelos compiladores C# e Visual Basic para entender programas nessas linguagens. Árvores de sintaxe são produzidas pelo mesmo analisador que é executado quando um projeto é compilado ou quando um desenvolvedor pressiona F5. As árvores de sintaxe têm fidelidade total com à linguagem de programação; cada bit de informações em um arquivo de código é representado na árvore. Gravar uma árvore de sintaxe em texto reproduz o texto original exato que foi analisado. As árvores de sintaxe também são **imutáveis**; uma vez criada, uma árvore de sintaxe nunca pode ser alterada. Os consumidores de árvores podem analisar as árvores de vários threads, sem bloqueios ou outras medidas de simultaneidade, sabendo que os dados nunca são alterados. Você pode usar APIs para criar novas árvores que são o resultado da modificação de uma árvore existente.

Os quatro principais blocos de construção de árvores de sintaxe são:

- A classe Microsoft.CodeAnalysis.SyntaxTree, uma instância da qual representa uma árvore de análise inteira. SyntaxTree é uma classe abstrata que tem derivativos específicos a um idioma. Você usa os métodos de análise da classe Microsoft.CodeAnalysis.CSharp.CSharpSyntaxTree (ou Microsoft.CodeAnalysis.VisualBasic.VisualBasicSyntaxTree) para analisar o texto em C# (ou em Visual Basic).

- A classe Microsoft.CodeAnalysis.SyntaxNode, instâncias da qual representam constructos sintáticos como declarações, instruções, cláusulas e expressões.
- A estrutura Microsoft.CodeAnalysis.SyntaxToken, que representa uma pontuação, operador, identificador ou palavra-chave individual.
- Finalmente, a estrutura Microsoft.CodeAnalysis.SyntaxTrivia, que representa os bits de informação sem significância sintática, tais como o espaço em branco entre tokens, diretivas de pré-processamento e comentários.

Trívia, tokens e nós são compostos hierarquicamente para formar uma árvore que representa completamente tudo em um fragmento de código do Visual Basic ou do C#. Você pode ver essa estrutura usando a janela **Visualizador de Sintaxe**. No Visual Studio, escolha **Exibição**>**Outras Janelas**>**Visualizador de Sintaxe**. Por exemplo, o arquivo de origem C# anterior examinado usando o **Visualizador de Sintaxe** se parecerá com a figura a seguir:

SyntaxNode: Azul | **SyntaxToken**: Verde | **SyntaxTrivia**: Vermelho

Ao navegar nessa estrutura de árvore, você pode encontrar qualquer instrução, expressão, token ou bit de espaço em branco em um arquivo de código.

Embora você possa encontrar tudo em um arquivo de código usando as APIs de sintaxe, a maioria dos cenários envolvem o exame de pequenos snippets de código ou a pesquisa por instruções ou fragmentos específicos. Os dois exemplos a seguir mostram usos típicos para navegar pela estrutura de códigos ou pesquisar por instruções individuais.

Percorrendo árvores

Você pode examinar os nós em uma árvore de sintaxe de duas maneiras. Você pode percorrer a árvore para examinar cada nó, ou então consultar elementos ou nós específicos.

Passagem manual

Você pode ver o código concluído para essa amostra no nosso repositório do GitHub.

> **Observação**
>
> Os tipos de árvore de sintaxe usam a herança para descrever os elementos de sintaxe diferentes que são válidos em locais diferentes no programa. Usar essas APIs geralmente significa converter propriedades ou membros da coleção em tipos derivados específicos. Nos exemplos a seguir, a atribuição e as conversões são instruções separadas, usando variáveis explicitamente tipadas. Você pode ler o código para ver os tipos de retorno da API e o tipo de runtime dos objetos retornados. Na prática, é mais comum usar variáveis implicitamente tipadas e depender de nomes de API para descrever o tipo de objeto que está sendo examinado.

Criar um novo projeto de **Ferramenta de Análise de Código Autônoma** do C#:

- No Visual Studio, escolha **Arquivo**>**Novo**>**Projeto** para exibir a caixa de diálogo Novo Projeto.
- Em **Visual C#**>**Extensibilidade**, escolha **Ferramenta de Análise de Código Autônoma**.
- Nomeie o projeto "**SyntaxTreeManualTraversal**" e clique em OK.

Você vai analisar o programa básico "Olá, Mundo!" mostrado anteriormente. Adicione o texto ao programa Olá, Mundo como uma constante em sua classe `Program`:

C#

```
        const string programText =
@"using System;
using System.Collections;
using System.Linq;
using System.Text;

namespace HelloWorld
{
    class Program
    {
        static void Main(string[] args)
```

```
        {
            Console.WriteLine(""Hello, World!"");
        }
    }
}";
```

Em seguida, adicione o código a seguir para criar a **árvore de sintaxe** para o texto do código na constante `programText`. Adicione a seguinte linha ao seu método `Main`:

C#

```
SyntaxTree tree = CSharpSyntaxTree.ParseText(programText);
CompilationUnitSyntax root = tree.GetCompilationUnitRoot();
```

Essas duas linhas criam a árvore e recuperam o nó raiz dessa árvore. Agora você pode examinar os nós na árvore. Adicione essas linhas ao seu método `Main` para exibir algumas das propriedades do nó raiz na árvore:

C#

```
WriteLine($"The tree is a {root.Kind()} node.");
WriteLine($"The tree has {root.Members.Count} elements in it.");
WriteLine($"The tree has {root.Usings.Count} using statements. They are:");
foreach (UsingDirectiveSyntax element in root.Usings)
    WriteLine($"\t{element.Name}");
```

Execute o aplicativo para ver o que seu código descobriu sobre o nó raiz nessa árvore.

Normalmente, percorreria a árvore para saber mais sobre o código. Neste exemplo, você está analisando código que você conhece para explorar as APIs. Adicione o código a seguir para examinar o primeiro membro do nó `root`:

C#

```
MemberDeclarationSyntax firstMember = root.Members[0];
WriteLine($"The first member is a {firstMember.Kind()}.");
var helloWorldDeclaration = (NamespaceDeclarationSyntax)firstMember;
```

Esse membro é um Microsoft.CodeAnalysis.CSharp.Syntax.NamespaceDeclarationSyntax. Ele representa tudo no escopo da declaração `namespace HelloWorld`. Adicione o seguinte código para examinar quais nós são declarados dentro do namespace `HelloWorld`:

C#

```
WriteLine($"There are {helloWorldDeclaration.Members.Count} members declared
in this namespace.");
WriteLine($"The first member is a
{helloWorldDeclaration.Members[0].Kind()}.");
```

Execute o programa para ver o que você aprendeu.

Agora que você sabe que a declaração é um Microsoft.CodeAnalysis.CSharp.Syntax.ClassDeclarationSyntax, declare uma nova variável de tipo para examinar a declaração de classe. Essa classe contém somente um membro: o método `Main`. Adicione o código a seguir para localizar o método `Main` e convertê-lo em um Microsoft.CodeAnalysis.CSharp.Syntax.MethodDeclarationSyntax.

C#

```
var programDeclaration =
(ClassDeclarationSyntax)helloWorldDeclaration.Members[0];
WriteLine($"There are {programDeclaration.Members.Count} members declared in
the {programDeclaration.Identifier} class.");
WriteLine($"The first member is a {programDeclaration.Members[0].Kind()}.");
var mainDeclaration =
(MethodDeclarationSyntax)programDeclaration.Members[0];
```

O nó de declaração do método contém todas as informações de sintaxe sobre o método. Permite exibir o tipo de retorno do método `Main`, o número e os tipos dos argumentos e o texto do corpo do método. Adicione os códigos a seguir:

C#

```
WriteLine($"The return type of the {mainDeclaration.Identifier} method is
{mainDeclaration.ReturnType}.");
WriteLine($"The method has {mainDeclaration.ParameterList.Parameters.Count}
parameters.");
foreach (ParameterSyntax item in mainDeclaration.ParameterList.Parameters)
    WriteLine($"The type of the {item.Identifier} parameter is
{item.Type}.");
WriteLine($"The body text of the {mainDeclaration.Identifier} method
follows:");
WriteLine(mainDeclaration.Body?.ToFullString());

var argsParameter = mainDeclaration.ParameterList.Parameters[0];
```

Execute o programa para ver todas as informações que você descobriu sobre este programa:

text

```
The tree is a CompilationUnit node.
The tree has 1 elements in it.
The tree has 4 using statements. They are:
        System
        System.Collections
        System.Linq
        System.Text
The first member is a NamespaceDeclaration.
There are 1 members declared in this namespace.
The first member is a ClassDeclaration.
There are 1 members declared in the Program class.
The first member is a MethodDeclaration.
The return type of the Main method is void.
The method has 1 parameters.
The type of the args parameter is string[].
The body text of the Main method follows:
        {
            Console.WriteLine("Hello, World!");
        }
```

Métodos de consulta

Além de percorrer árvores, você também pode explorar a árvore de sintaxe usando os métodos de consulta definidos em Microsoft.CodeAnalysis.SyntaxNode. Esses métodos devem ser imediatamente familiares a qualquer pessoa familiarizada com o XPath. Você pode usar esses métodos com o LINQ para localizar itens rapidamente em uma árvore. O SyntaxNode tem métodos de consulta como DescendantNodes, AncestorsAndSelf e ChildNodes.

Você pode usar esses métodos de consulta para localizar o argumento para o método `Main` como uma alternativa a navegar pela árvore. Adicione o seguinte código à parte inferior do método `Main`:

C#

```
var firstParameters = from methodDeclaration in root.DescendantNodes()
                                        .OfType<MethodDeclarationSyntax>()
                      where methodDeclaration.Identifier.ValueText == "Main"
                      select
methodDeclaration.ParameterList.Parameters.First();

var argsParameter2 = firstParameters.Single();

WriteLine(argsParameter == argsParameter2);
```

A primeira instrução usa uma expressão LINQ e o método DescendantNodes para localizar o mesmo parâmetro do exemplo anterior.

Execute o programa e você poderá ver que a expressão LINQ encontrou o mesmo parâmetro encontrado ao navegar manualmente pela árvore.

O exemplo usa instruções `WriteLine` para exibir informações sobre as árvores de sintaxe conforme elas são percorridas. Você também pode aprender mais executando o programa concluído no depurador. Você pode examinar mais das propriedades e métodos que fazem parte da árvore de sintaxe criada para o programa Olá, Mundo.

Caminhadores de sintaxe

Muitas vezes, você deseja localizar todos os nós de um tipo específico em uma árvore de sintaxe, por exemplo, cada declaração de propriedade em um arquivo. Ao estender a classe Microsoft.CodeAnalysis.CSharp.CSharpSyntaxWalker e substituir o método VisitPropertyDeclaration(PropertyDeclarationSyntax), você processa cada declaração de propriedade em uma árvore de sintaxe sem conhecer a estrutura dele com antecedência. CSharpSyntaxWalker é um tipo específico de CSharpSyntaxVisitor, que visita recursivamente um nó e cada um dos filhos desse nó.

Este exemplo implementa um CSharpSyntaxWalker que examina uma árvore de sintaxe. Ele coleta diretivas `using` que ele constata que não estão importando um namespace `System`.

Crie um novo projeto de **Ferramenta de Análise de Código Autônoma** do C#; nomeie-o "**SyntaxWalker**".

Você pode ver o código concluído para essa amostra no nosso repositório do GitHub. A amostra no GitHub contém os dois projetos descritos neste tutorial.

Assim como no exemplo anterior, você pode definir uma constante de cadeia de caracteres para conter o texto do programa que você pretende analisar:

C#

```
        const string programText =
@"using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using Microsoft.CodeAnalysis;
using Microsoft.CodeAnalysis.CSharp;

namespace TopLevel
{
    using Microsoft;
    using System.ComponentModel;
```

```
    namespace Child1
    {
        using Microsoft.Win32;
        using System.Runtime.InteropServices;

        class Foo { }
    }

    namespace Child2
    {
        using System.CodeDom;
        using Microsoft.CSharp;

        class Bar { }
    }
}";
```

Este texto de origem contém diretivas `using` espalhadas em quatro locais diferentes: o nível de arquivo, no namespace de nível superior e nos dois namespaces aninhados. Este exemplo destaca um cenário principal para usar a classe CSharpSyntaxWalker para consultar código. Seria complicado visitar cada nó na árvore de sintaxe de raiz para encontrar declarações using. Em vez disso, você pode criar uma classe derivada e substituir o método chamado apenas quando o nó atual na árvore é uma diretiva using. O visitante não realiza nenhum trabalho em nenhum outro tipo de nó. Esse método único examina cada uma das instruções `using` e compila uma coleção de namespaces que não estão no namespace `System`. Você compila um CSharpSyntaxWalker que examina todas as instruções `using`, mas apenas as instruções `using`.

Agora que você definiu o texto do programa, você precisa criar um `SyntaxTree` e obter a raiz dessa árvore:

C#

```
SyntaxTree tree = CSharpSyntaxTree.ParseText(programText);
CompilationUnitSyntax root = tree.GetCompilationUnitRoot();
```

Em seguida, crie uma nova classe. No Visual Studio, escolha **Projeto**>**Adicionar Novo Item**. Na caixa de diálogo **Adicionar Novo Item**, digite *UsingCollector.cs* como o nome do arquivo.

Você implementa a funcionalidade de visitante `using` na classe `UsingCollector`. Para começar, crie a classe `UsingCollector` derivada de CSharpSyntaxWalker.

C#

```
class UsingCollector : CSharpSyntaxWalker
```

Você precisa de armazenamento para conter os nós de namespace que você está coletando. Declare uma propriedade pública somente leitura na classe `UsingCollector`; use essa variável para armazenar os nós UsingDirectiveSyntax que você encontrar:

```
C#

public ICollection<UsingDirectiveSyntax> Usings { get; } = new
List<UsingDirectiveSyntax>();
```

A classe base CSharpSyntaxWalker implementa a lógica para visitar cada nó na árvore de sintaxe. A classe derivada substitui os métodos chamados para os nós específicos nos quais você está interessado. Nesse caso, você está interessado em qualquer diretiva `using`. Isso significa que você deve substituir o método VisitUsingDirective(UsingDirectiveSyntax). Um argumento para esse método é um objeto Microsoft.CodeAnalysis.CSharp.Syntax.UsingDirectiveSyntax. Essa é uma importante vantagem de se usar os visitantes: eles chamam os métodos substituídos com argumentos que já foram convertidos para o tipo de nó específico. A classe Microsoft.CodeAnalysis.CSharp.Syntax.UsingDirectiveSyntax tem uma propriedade Name que armazena o nome do namespace que está sendo importado. É um Microsoft.CodeAnalysis.CSharp.Syntax.NameSyntax. Adicione o código a seguir na substituição VisitUsingDirective(UsingDirectiveSyntax):

```
C#

public override void VisitUsingDirective(UsingDirectiveSyntax node)
{
    WriteLine($"\tVisitUsingDirective called with {node.Name}.");
    if (node.Name.ToString() != "System" &&
        !node.Name.ToString().StartsWith("System."))
    {
        WriteLine($"\t\tSuccess. Adding {node.Name}.");
        this.Usings.Add(node);
    }
}
```

Assim como no exemplo anterior, você adicionou uma variedade de instruções `WriteLine` para ajudar na compreensão do método. Você pode ver quando ele é chamado e quais argumentos são passados para ele a cada vez.

Por fim, você precisa adicionar duas linhas de código para criar o `UsingCollector` e fazer com que ele acesse o nó raiz, coletando todas as instruções `using`. Em seguida, adicione um loop `foreach` para exibir todas as instruções `using` encontradas pelo seu coletor:

C#

```
var collector = new UsingCollector();
collector.Visit(root);
foreach (var directive in collector.Usings)
{
    WriteLine(directive.Name);
}
```

Compile e execute o programa. Você deve ver o seguinte resultado:

Console

```
        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 . . .
```

Parabéns! Você usou a **API de sintaxe** para localizar tipos específicos de instruções C# e declarações em código-fonte C#.

Introdução à análise semântica

Artigo • 10/05/2023

Este tutorial presume que você está familiarizado com a API de sintaxe. O artigo Introdução à a análise de sintaxe fornece uma introdução suficiente.

Neste tutorial, você explora as APIs de **Símbolo** e de **Associação**. Essas APIs fornecem informações sobre o *significado semântico* de um programa. Elas permitem fazer e responder perguntas sobre os tipos representado por qualquer símbolo em seu programa.

Você deverá instalar o **SDK do .NET Compiler Platform**:

Instruções de instalação – Instalador do Visual Studio

Há duas maneiras diferentes de encontrar o **SDK da .NET Compiler Platform** no **Instalador do Visual Studio**:

Instalar usando o Instalador do Visual Studio – exibição de cargas de trabalho

O SDK da .NET Compiler Platform não é selecionado automaticamente como parte da carga de trabalho de desenvolvimento da extensão do Visual Studio. É necessário selecioná-lo como um componente opcional.

1. Execute o **Instalador do Visual Studio**
2. Escolha **Mudar**
3. Marque a carga de trabalho de **Desenvolvimento de extensão do Visual Studio**.
4. Abra o nó **Desenvolvimento de extensão do Visual Studio** na árvore de resumo.
5. Marque a caixa do **SDK da .NET Compiler Platform**. Você a encontrará por último nos componentes opcionais.

Opcionalmente, você também poderá fazer o **Editor DGML** exibir gráficos no visualizador:

1. Abra o nó **Componentes individuais** na árvore de resumo.
2. Marque a caixa do **Editor DGML**

Instalar usando o Instalador do Visual Studio – guia Componentes individuais

1. Execute o **Instalador do Visual Studio**
2. Escolha **Mudar**
3. Selecione a guia **Componentes individuais**
4. Marque a caixa do **SDK da .NET Compiler Platform**. Você a encontrará na parte superior, na seção **Compiladores, ferramentas de compilação e runtimes**.

Opcionalmente, você também poderá fazer o **Editor DGML** exibir gráficos no visualizador:

1. Marque a caixa do **Editor DGML**. Você a encontrará na seção **Ferramentas de código**.

Noções básicas sobre compilações e símbolos

Conforme você trabalha mais com o SDK do .NET Compiler, você se familiariza com as distinções entre a API de Sintaxe e a API de Semântica. A **API de Sintaxe** permite que você examine a *estrutura* de um programa. Muitas vezes, no entanto, você deseja as informações sobre a semântica ou *significado* de um programa. Embora um snippet ou arquivo de código livre do Visual Basic ou C# possa ser analisado sintaticamente de modo isolado, não faz sentido fazer a esmo perguntas como "qual é o tipo dessa variável". O significado de um nome de tipo pode ser dependente de referências de assembly, importações de namespace ou outros arquivos de código. Essas perguntas são respondidas usando-se a **API de Semântica**, especificamente a classe Microsoft.CodeAnalysis.Compilation.

Uma instância de Compilation é análoga a um único projeto conforme visto pelo compilador e representa tudo o que é necessário para compilar um programa Visual Basic ou C#. A **compilação** inclui o conjunto de arquivos de origem a serem compilados, referências de assembly e opções de compilador. Você pode avaliar o significado do código usando todas as outras informações neste contexto. Um Compilation permite que você encontre **símbolos** – entidades como tipos, namespaces, membros e variáveis aos quais os nomes e outras expressões se referem. O processo de associar nomes e expressões com **símbolos** é chamado de **associação**.

Assim como Microsoft.CodeAnalysis.SyntaxTree, Compilation é uma classe abstrata com derivativos específicos a um idioma. Ao criar uma instância de compilação, você deve invocar um método de fábrica na classe Microsoft.CodeAnalysis.CSharp.CSharpCompilation (ou Microsoft.CodeAnalysis.VisualBasic.VisualBasicCompilation).

Consultar símbolos

Neste tutorial, você analisa novamente o programa "Olá, Mundo". Dessa vez, você consulta os símbolos no programa para compreender quais tipos esses símbolos representam. Você consulta os tipos em um namespace e aprende a localizar os métodos disponíveis em um tipo.

Você pode ver o código concluído para essa amostra no nosso repositório do GitHub.

> **Observação**
>
> Os tipos de árvore de sintaxe usam a herança para descrever os elementos de sintaxe diferentes que são válidos em locais diferentes no programa. Usar essas APIs geralmente significa converter propriedades ou membros da coleção em tipos derivados específicos. Nos exemplos a seguir, a atribuição e as conversões são instruções separadas, usando variáveis explicitamente tipadas. Você pode ler o código para ver os tipos de retorno da API e o tipo de runtime dos objetos retornados. Na prática, é mais comum usar variáveis implicitamente tipadas e depender de nomes de API para descrever o tipo de objeto que está sendo examinado.

Criar um novo projeto de **Ferramenta de Análise de Código Autônoma** do C#:

- No Visual Studio, escolha **Arquivo**>**Novo**>**Projeto** para exibir a caixa de diálogo Novo Projeto.
- Em **Visual C#**>**Extensibilidade**, escolha **Ferramenta de Análise de Código Autônoma**.
- Nomeie o projeto "**SemanticQuickStart**" e clique em OK.

Você vai analisar o programa básico "Olá, Mundo!" mostrado anteriormente. Adicione o texto ao programa Olá, Mundo como uma constante em sua classe `Program`:

C#

```
        const string programText =
@"using System;
using System.Collections.Generic;
using System.Text;

namespace HelloWorld
{
    class Program
    {
        static void Main(string[] args)
        {
```

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

Em seguida, adicione o código a seguir para criar a árvore de sintaxe para o texto do código na constante `programText`. Adicione a seguinte linha ao seu método `Main`:

```C#
SyntaxTree tree = CSharpSyntaxTree.ParseText(programText);

CompilationUnitSyntax root = tree.GetCompilationUnitRoot();
```

Em seguida, compile uma CSharpCompilation da árvore que você já criou. A amostra "Olá, Mundo" depende dos tipos String e Console. Você precisa fazer referência ao assembly que declara esses dois tipos em sua compilação. Adicione a seguinte linha ao seu método `Main` para criar uma compilação de sua árvore de sintaxe, incluindo a referência ao assembly apropriado:

```C#
var compilation = CSharpCompilation.Create("HelloWorld")
    .AddReferences(MetadataReference.CreateFromFile(
        typeof(string).Assembly.Location))
    .AddSyntaxTrees(tree);
```

O método CSharpCompilation.AddReferences adiciona referências à compilação. O método MetadataReference.CreateFromFile carrega um assembly como uma referência.

Consultar o modelo semântico

Assim que você tiver uma Compilation, você poderá solicitar a ela um SemanticModel para qualquer SyntaxTree contida nessa Compilation. Você pode pensar no modelo semântico como a origem de todas as informações normalmente obtidas do IntelliSense. Um SemanticModel pode responder a perguntas como "O que são nomes no escopo nesse local?", "Quais membros são acessíveis deste método?", "Quais variáveis são usadas neste bloco de texto?" e "A que este nome/expressão se refere?" Adicione esta instrução para criar o modelo semântico:

```C#
SemanticModel model = compilation.GetSemanticModel(tree);
```

Associar um nome

A Compilation cria o SemanticModel da SyntaxTree. Depois de criar o modelo, você pode consultar para localizar a primeira diretiva `using` e recuperar as informações de símbolo para o namespace `System`. Adicione estas duas linhas a seu método `Main` para criar o modelo semântico e recuperar o símbolo para a primeira instrução using:

C#

```
// Use the syntax tree to find "using System;"
UsingDirectiveSyntax usingSystem = root.Usings[0];
NameSyntax systemName = usingSystem.Name;

// Use the semantic model for symbol information:
SymbolInfo nameInfo = model.GetSymbolInfo(systemName);
```

O código anterior mostra como associar o nome na primeira diretiva `using` para recuperar um Microsoft.CodeAnalysis.SymbolInfo para o namespace `System`. O código anterior também ilustra o uso da **sintaxe de modelo** para localizar a estrutura do código; você usa o **modelo semântico** para entender seu significado. A **sintaxe de modelo** localiza a cadeia de caracteres `System` na instrução using. O **modelo semântico** tem todas as informações sobre os tipos definidos no namespace `System`.

Do objeto SymbolInfo, você pode obter o Microsoft.CodeAnalysis.ISymbol usando a propriedade SymbolInfo.Symbol. Essa propriedade retorna o símbolo a que essa expressão se refere. Para expressões que não se referem a nada (como literais numéricos), essa propriedade é `null`. Quando o SymbolInfo.Symbol não for null, o ISymbol.Kind denotará o tipo do símbolo. Nesse exemplo, a propriedade ISymbol.Kind é um SymbolKind.Namespace. Adicione o código a seguir ao método `Main`. Ele recupera o símbolo para o namespace `System` e, em seguida, exibe todos os namespaces filho declarados no namespace `System`:

C#

```
var systemSymbol = (INamespaceSymbol?)nameInfo.Symbol;
if (systemSymbol?.GetNamespaceMembers() is not null)
{
    foreach (INamespaceSymbol ns in systemSymbol?.GetNamespaceMembers()!)
    {
        Console.WriteLine(ns);
    }
}
```

Execute o programa e você deverá ver a seguinte saída:

Saída

```
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 . . .
```

Observação

A saída não inclui todos os namespaces que são namespaces filhos do namespace `System`. El exibe cada namespace presente nessa compilação, que só referencia o assembly em que `System.String` é declarada. Quaisquer outros namespaces declarados em outros assemblies não são conhecidos desta compilação

Associar uma expressão

O código anterior mostra como encontrar um símbolo associando-o a um nome. Há outras expressões em um programa C# que podem ser associadas que não são nomes. Para demonstrar essa capacidade, acessaremos a associação a um único literal de cadeia de caracteres.

O programa "Olá, Mundo" contém Microsoft.CodeAnalysis.CSharp.Syntax.LiteralExpressionSyntax, uma cadeia de caracteres "Olá, Mundo!" exibida no console.

Você encontra a cadeia de caracteres "Olá, Mundo!" localizando o único literal de cadeia de caracteres no programa. Em seguida, depois de localizar o nó de sintaxe, você obtém as informações de tipo para esse nó do modelo semântico. Adicione o código a seguir ao método `Main`:

C#

```
// Use the syntax model to find the literal string:
LiteralExpressionSyntax helloWorldString = root.DescendantNodes()
```

```
.OfType<LiteralExpressionSyntax>()
.Single();

// Use the semantic model for type information:
TypeInfo literalInfo = model.GetTypeInfo(helloWorldString);
```

O struct Microsoft.CodeAnalysis.TypeInfo inclui uma propriedade TypeInfo.Type que permite o acesso às informações semânticas sobre o tipo do literal. Neste exemplo, ele é do tipo `string`. Adicione uma declaração que atribui essa propriedade a uma variável local:

```
C#

var stringTypeSymbol = (INamedTypeSymbol?)literalInfo.Type;
```

Para concluir este tutorial, criaremos uma consulta LINQ que criará uma sequência de todos os métodos públicos declarados no tipo `string` que retorna um `string`. Essa consulta torna-se complexa, então a compilaremos linha a linha e então a reconstruiremos como uma única consulta. A ordem desta consulta é a sequência de todos os membros declarados no tipo `string`:

```
C#

var allMembers = stringTypeSymbol?.GetMembers();
```

Essa sequência de origem contém todos os membros, incluindo propriedades e campos, portanto, filtre-a usando o método ImmutableArray<T>.OfType para localizar elementos que são objetos Microsoft.CodeAnalysis.IMethodSymbol:

```
C#

var methods = allMembers?.OfType<IMethodSymbol>();
```

Em seguida, adicione outro filtro para retornar somente os métodos que são públicos e retornam um `string`:

```
C#

var publicStringReturningMethods = methods?
    .Where(m => SymbolEqualityComparer.Default.Equals(m.ReturnType,
stringTypeSymbol) &&
    m.DeclaredAccessibility == Accessibility.Public);
```

Selecione apenas a propriedade de nome e somente os nomes distintos, removendo quaisquer sobrecargas:

C#

```
var distinctMethods = publicStringReturningMethods?.Select(m =>
m.Name).Distinct();
```

Você pode também compilar a consulta completa usando a sintaxe de consulta LINQ e, em seguida, exibir todos os nomes de método no console:

C#

```
foreach (string name in (from method in stringTypeSymbol?
                         .GetMembers().OfType<IMethodSymbol>()
                         where
SymbolEqualityComparer.Default.Equals(method.ReturnType, stringTypeSymbol)
&&
                         method.DeclaredAccessibility ==
Accessibility.Public
                         select method.Name).Distinct())
{
    Console.WriteLine(name);
}
```

Compile e execute o programa. Você deve ver o seguinte resultado:

Saída

```
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 . . .
```

Você usou a API de semântica para localizar e exibir informações sobre os símbolos que fazem parte deste programa.

Introdução à transformação de sintaxe

Artigo • 09/05/2023

Este tutorial baseia-se nos conceitos e técnicas explorados nos guias de início rápido Introdução à análise de sintaxe e Introdução à análise semântica. Se ainda não o fez, você deve concluir as etapas rápidas antes de começar esta.

Neste início rápido, você explora técnicas para criar e transformar árvores de sintaxe. Em combinação com as técnicas que você aprendeu em guias de início rápido anteriores, você cria sua primeira refatoração de linha de comando!

Instruções de instalação – Instalador do Visual Studio

Há duas maneiras diferentes de encontrar o **SDK da .NET Compiler Platform** no **Instalador do Visual Studio**:

Instalar usando o Instalador do Visual Studio – exibição de cargas de trabalho

O SDK da .NET Compiler Platform não é selecionado automaticamente como parte da carga de trabalho de desenvolvimento da extensão do Visual Studio. É necessário selecioná-lo como um componente opcional.

1. Execute o **Instalador do Visual Studio**
2. Escolha **Mudar**
3. Marque a carga de trabalho de **Desenvolvimento de extensão do Visual Studio**.
4. Abra o nó **Desenvolvimento de extensão do Visual Studio** na árvore de resumo.
5. Marque a caixa do **SDK da .NET Compiler Platform**. Você a encontrará por último nos componentes opcionais.

Opcionalmente, você também poderá fazer o **Editor DGML** exibir gráficos no visualizador:

1. Abra o nó **Componentes individuais** na árvore de resumo.
2. Marque a caixa do **Editor DGML**

Instalar usando o Instalador do Visual Studio – guia Componentes individuais

1. Execute o **Instalador do Visual Studio**
2. Escolha **Mudar**
3. Selecione a guia **Componentes individuais**
4. Marque a caixa do **SDK da .NET Compiler Platform**. Você a encontrará na parte superior, na seção **Compiladores, ferramentas de compilação e runtimes**.

Opcionalmente, você também poderá fazer o **Editor DGML** exibir gráficos no visualizador:

1. Marque a caixa do **Editor DGML**. Você a encontrará na seção **Ferramentas de código**.

Imutabilidade e a plataforma de compiladores .NET

Imutabilidade é um princípio fundamental da plataforma de compiladores .NET. Estruturas de dados imutáveis não podem ser alteradas depois de criadas. Estruturas de dados imutáveis podem ser compartilhadas com segurança e analisadas por vários consumidores simultaneamente. Não há perigo de que um consumidor afete o outro de maneiras imprevisíveis. Seu analisador não precisa de bloqueios ou outras medidas de simultaneidade. Essa regra se aplica a árvores de sintaxe, compilações, símbolos, modelos semânticos e todas as outras estruturas de dados que você encontrar. Em vez de modificar as estruturas existentes, as APIs criam novos objetos com base nas diferenças especificadas para os antigos. Você aplica esse conceito a árvores de sintaxe para criar novas árvores usando transformações.

Criar e transformar árvores

Você escolhe uma das duas estratégias para transformações de sintaxe. Os **métodos de fábrica** são melhor usados quando você está procurando por nós específicos para substituir ou locais específicos onde deseja inserir um novo código. **Regravadores** são a melhor opção quando você deseja examinar um projeto inteiro em busca dos padrões de código que deseja substituir.

Criar nós com métodos de fábrica

A primeira transformação de sintaxe demonstra os métodos de fábrica. Substitua uma instrução `using System.Collections;` por uma instrução `using System.Collections.Generic;`. Este exemplo demonstra como você cria objetos Microsoft.CodeAnalysis.CSharp.CSharpSyntaxNode usando os métodos de fábrica

Microsoft.CodeAnalysis.CSharp.SyntaxFactory. Para cada tipo de **nó**, **tokenou trívia**, há um método de fábrica que cria uma instância desse tipo. Você cria árvores de sintaxe compondo os nós hierarquicamente de baixa para cima. Em seguida, você transformará o programa existente substituindo nós existentes pela nova árvore criada.

Inicie o Visual Studio e crie um novo projeto de **Ferramenta de Análise de Código Autônoma** do C#. No Visual Studio, escolha **Arquivo**>**Novo**>**Projeto** para exibir a caixa de diálogo Novo Projeto. Em **Visual C#**>**Extensibilidade**, escolha uma **Ferramenta de Análise de Código Autônoma**. Este guia de início rápido tem dois projetos de exemplo, portanto, nomeie a solução **SyntaxTransformationQuickStart** e nomeie o projeto **ConstructionCS**. Clique em **OK**.

Este projeto usa os métodos de classe Microsoft.CodeAnalysis.CSharp.SyntaxFactory para construir um Microsoft.CodeAnalysis.CSharp.Syntax.NameSyntax representando o namespace `System.Collections.Generic`.

Adicione a seguinte diretiva de uso à parte superior do `Program.cs`.

C#

```
using static Microsoft.CodeAnalysis.CSharp.SyntaxFactory;
using static System.Console;
```

Você criará **nós de sintaxe de nome** para construir a árvore que representa a instrução `using System.Collections.Generic;`. NameSyntax é a classe base para quatro tipos de nomes que aparecem no C#. Você compõe esses quatro tipos de nomes para criar qualquer nome que possa aparecer na linguagem C#:

- Microsoft.CodeAnalysis.CSharp.Syntax.NameSyntax, que representa nomes simples de identificadores únicos como `System` e `Microsoft`.
- Microsoft.CodeAnalysis.CSharp.Syntax.GenericNameSyntax, que representa um tipo genérico ou nome de método, como `List<int>`.
- Microsoft.CodeAnalysis.CSharp.Syntax.QualifiedNameSyntax,que representa um nome qualificado do formulário `<left-name>.<right-identifier-or-generic-name>`, como `System.IO`.
- Microsoft.CodeAnalysis.CSharp.Syntax.AliasQualifiedNameSyntax, que representa um nome usando um alias externo de assembly como `LibraryV2::Foo`.

Você usa o método IdentifierName(String) para criar um nó NameSyntax. Adicione o seguinte código no seu método `Main` no `Program.cs`:

C#

```
NameSyntax name = IdentifierName("System");
WriteLine($"\tCreated the identifier {name}");
```

O código anterior cria um objeto IdentifierNameSyntax e o atribui à variável `name`. Muitas das APIs Roslyn retornam classes básicas para facilitar o trabalho com tipos relacionados. A variável `name`, um NameSyntax, pode ser reutilizada conforme você constrói o QualifiedNameSyntax. Não use inferência de tipo ao criar a amostra. Você automatizará essa etapa neste projeto.

Você criou o nome. Agora, é hora de criar mais nós na árvore criando um QualifiedNameSyntax. A nova árvore usa `name` como a esquerda do nome e um novo IdentifierNameSyntax para o namespace `Collections` como o lado direito do QualifiedNameSyntax. Adicione o seguinte código a `program.cs`:

C#

```
name = QualifiedName(name, IdentifierName("Collections"));
WriteLine(name.ToString());
```

Execute o código novamente e confira os resultados. Você está construindo uma árvore de nós que representa o código. Você continuará este padrão para construir o QualifiedNameSyntax para o namespace `System.Collections.Generic`. Adicione o seguinte código a `Program.cs`:

C#

```
name = QualifiedName(name, IdentifierName("Generic"));
WriteLine(name.ToString());
```

Execute o programa novamente para ver se você construiu a árvore para o código a ser adicionado.

Criar uma árvore modificada

Você criou uma pequena árvore de sintaxe que contém uma instrução. As APIs para criar novos nós são a escolha certa para criar instruções únicas ou outros pequenos blocos de código. No entanto, para construir blocos maiores de código, você deve usar métodos que substituem nós ou inserem nós em uma árvore existente. Lembre-se de que as árvores de sintaxe são imutáveis. A **API de Sintaxe** não fornece nenhum mecanismo para modificar uma árvore de sintaxe existente após a construção. Em vez disso, fornece métodos que produzem novas árvores com base nas alterações

existentes. Os métodos `With*` são definidos em classes concretas que derivam de SyntaxNode ou em métodos de extensão declarados na classe SyntaxNodeExtensions. Esses métodos criam um novo nó aplicando alterações nas propriedades filho de um nó existente. Além disso, o método de extensão ReplaceNode pode ser usado para substituir um nó descendente em uma subárvore. Esse método também atualiza o pai para apontar para o filho recém-criado e repete esse processo até a árvore inteira — um processo conhecido como *re-spinning* da árvore.

O próximo passo é criar uma árvore que represente um programa inteiro (pequeno) e depois modificá-la. Adicione o seguinte código ao início da classe `Program`:

C#

```
        private const string sampleCode =
@"using System;
using System.Collections;
using System.Linq;
using System.Text;

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

Observação

O código de exemplo usa o namespace `System.Collections` e não o namespace `System.Collections.Generic`.

Em seguida, adicione o seguinte código à parte inferior do método `Main` para analisar o texto e criar uma árvore:

C#

```
SyntaxTree tree = CSharpSyntaxTree.ParseText(sampleCode);
var root = (CompilationUnitSyntax)tree.GetRoot();
```

Este exemplo usa o método WithName(NameSyntax) para substituir o nome em um nó UsingDirectiveSyntax pelo que foi construído no código anterior.

Crie um novo nó UsingDirectiveSyntax usando o método WithName(NameSyntax) para atualizar o nome `System.Collections` com o nome criado no código anterior. Adicione o seguinte código à parte inferior do método `Main`:

```
C#

var oldUsing = root.Usings[1];
var newUsing = oldUsing.WithName(name);
WriteLine(root.ToString());
```

Execute o programa e observe atentamente a saída. O `newUsing` não foi colocado na árvore raiz. A árvore original não foi alterada.

Adicione o seguinte código usando o método de extensão ReplaceNode para criar uma nova árvore. A nova árvore é o resultado da substituição da importação existente pelo nó `newUsing` atualizado. Você atribui essa nova árvore à variável `root` existente:

```
C#

root = root.ReplaceNode(oldUsing, newUsing);
WriteLine(root.ToString());
```

Execute o programa novamente. Desta vez, a árvore importa corretamente o namespace `System.Collections.Generic`.

Transformar árvores usando `SyntaxRewriters`

Os métodos `With*` e ReplaceNode fornecem meios convenientes para transformar ramos individuais de uma árvore de sintaxe. A classe Microsoft.CodeAnalysis.CSharp.CSharpSyntaxRewriter realiza várias transformações em uma árvore de sintaxe. A classe Microsoft.CodeAnalysis.CSharp.CSharpSyntaxRewriter é uma subclasse de Microsoft.CodeAnalysis.CSharp.CSharpSyntaxVisitor<TResult>. O CSharpSyntaxRewriter aplica uma transformação a um tipo específico de SyntaxNode. Você pode aplicar transformações a vários tipos de objetos SyntaxNode sempre que eles aparecerem em uma árvore de sintaxe. O segundo projeto neste guia de início rápido cria uma refatoração de linha de comando que remove tipos explícitos em declarações de variáveis locais em qualquer lugar em que a inferência de tipo possa ser usada.

Crie um novo projeto de **Ferramenta de Análise de Código Autônoma** do C#. No Visual Studio, clique com o botão direito do mouse no nó da solução `SyntaxTransformationQuickStart`. Escolha **Adicionar**>**Novo Projeto** para exibir o diálogo

Novo Projeto. Em **Visual C#**>**Extensibilidade**, escolha **Ferramenta de Análise de Código Autônoma**. Nomeie seu projeto como `TransformationCS` e clique em OK.

A primeira etapa é criar uma classe que deriva de CSharpSyntaxRewriter para executar as transformações. Adicione um novo arquivo de classe ao projeto. No Visual Studio, escolha **Projeto**>**Adicionar Classes...**. Na caixa de diálogo **Adicionar Novo Item**, digite `TypeInferenceRewriter.cs` como nome do arquivo.

Adicione o seguinte usando diretivas ao arquivo `TypeInferenceRewriter.cs`:

```C#
using Microsoft.CodeAnalysis;
using Microsoft.CodeAnalysis.CSharp;
using Microsoft.CodeAnalysis.CSharp.Syntax;
```

Em seguida, faça a classe `TypeInferenceRewriter` se estender à classe CSharpSyntaxRewriter:

```C#
public class TypeInferenceRewriter : CSharpSyntaxRewriter
```

Adicione o seguinte código para declarar um campo somente leitura privado para conter um SemanticModel e inicializá-lo no construtor. Você precisará deste campo posteriormente para determinar onde a inferência de tipos pode ser usada:

```C#
private readonly SemanticModel SemanticModel;

public TypeInferenceRewriter(SemanticModel semanticModel) => SemanticModel =
semanticModel;
```

Substitua o método VisitLocalDeclarationStatement(LocalDeclarationStatementSyntax):

```C#
public override SyntaxNode
VisitLocalDeclarationStatement(LocalDeclarationStatementSyntax node)
{

}
```

> ⓘ **Observação**
>
> Muitas das APIs Roslyn declaram os tipos de retorno que são classes base dos tipos de runtime reais retornados. Em muitos cenários, um tipo de nó pode ser substituído inteiramente por outro tipo de nó, ou até mesmo removido. Neste exemplo, o método **VisitLocalDeclarationStatement(LocalDeclarationStatementSyntax)** retorna **SyntaxNode**, em vez do tipo derivado de **LocalDeclarationStatementSyntax**. Este regravador retorna um novo nó **LocalDeclarationStatementSyntax** baseado no existente.

Este guia de início rápido lida com declarações de variáveis locais. Você poderia estendê-lo para outras declarações, como loops `foreach`, loops `for`, expressões LINQ e expressões lambda. Além disso, este regravador só irá transformar as declarações da forma mais simples:

C#

```
Type variable = expression;
```

Se você quiser explorar por conta própria, considere estender a amostra finalizada para esses tipos de declarações de variáveis:

C#

```
// Multiple variables in a single declaration.
Type variable1 = expression1,
     variable2 = expression2;
// No initializer.
Type variable;
```

Adicione o seguinte código ao corpo do método `VisitLocalDeclarationStatement` para ignorar a reescrita dessas formas de declaração:

C#

```
if (node.Declaration.Variables.Count > 1)
{
    return node;
}
if (node.Declaration.Variables[0].Initializer == null)
{
    return node;
}
```

O método indica que nenhuma reescrita ocorre retornando o parâmetro `node` não modificado. Se nenhuma dessas expressões `if` for verdadeira, o nó representa uma possível declaração com inicialização. Adicione estas instruções para extrair o nome do tipo especificado na declaração e vinculá-lo usando o campo SemanticModel para obter um símbolo de tipo:

```
C#

var declarator = node.Declaration.Variables.First();
var variableTypeName = node.Declaration.Type;

var variableType = (ITypeSymbol)SemanticModel
    .GetSymbolInfo(variableTypeName)
    .Symbol;
```

Agora, adicione esta instrução para associar a expressão inicializadora:

```
C#

var initializerInfo =
SemanticModel.GetTypeInfo(declarator.Initializer.Value);
```

Por fim, inclua a seguinte instrução `if` para substituir o nome do tipo existente pela palavra-chave `var`, se o tipo de expressão do inicializador corresponder ao tipo especificado:

```
C#

if (SymbolEqualityComparer.Default.Equals(variableType,
initializerInfo.Type))
{
    TypeSyntax varTypeName = SyntaxFactory.IdentifierName("var")
        .WithLeadingTrivia(variableTypeName.GetLeadingTrivia())
        .WithTrailingTrivia(variableTypeName.GetTrailingTrivia());

    return node.ReplaceNode(variableTypeName, varTypeName);
}
else
{
    return node;
}
```

A condicional é necessária porque a declaração pode converter a expressão inicializadora em uma classe ou interface base. Se este for o caso, os tipos à esquerda e à direita da atribuição não coincidem. Remover o tipo explícito nesses casos alteraria a

semântica de um programa. `var` é especificado como um identificador em vez de uma palavra-chave porque `var` é uma palavra-chave contextual. As trivialidades inicial e final (espaço em branco) são transferidas do nome do tipo antigo para a palavra-chave `var` para manter espaço em branco vertical e recuo. É mais simples usar `ReplaceNode` em vez de `With*` para transformar o LocalDeclarationStatementSyntax, pois o nome do tipo é na verdade o neto da declaração.

Você concluiu o `TypeInferenceRewriter`. Agora, retorne ao arquivo `Program.cs` para finalizar o exemplo. Crie um teste Compilation e obtenha o SemanticModel dele. Use esse SemanticModel para testar seu `TypeInferenceRewriter`. Você realizará esta etapa por último. Enquanto isso, declare uma variável de espaço reservado representando sua compilação de teste:

```
C#

Compilation test = CreateTestCompilation();
```

Após uma pausa, um erro será exibido informando que não existe método `CreateTestCompilation`. Pressione **Ctrl + Ponto** para abrir a lâmpada e pressione Enter para invocar o comando **Gerar Stub de Método**. Este comando irá gerar um stub de método para o método `CreateTestCompilation` na classe `Program`. Você voltará a preencher este método mais tarde:

Grave o seguinte código para iterar sobre cada SyntaxTree no teste Compilation. Para cada um, inicialize um novo `TypeInferenceRewriter` com o SemanticModel para essa árvore:

C#

```
foreach (SyntaxTree sourceTree in test.SyntaxTrees)
{
    SemanticModel model = test.GetSemanticModel(sourceTree);

    TypeInferenceRewriter rewriter = new TypeInferenceRewriter(model);

    SyntaxNode newSource = rewriter.Visit(sourceTree.GetRoot());

    if (newSource != sourceTree.GetRoot())
    {
        File.WriteAllText(sourceTree.FilePath, newSource.ToFullString());
    }
}
```

Dentro da instrução `foreach` criada, adicione o seguinte código para executar a transformação em cada árvore de origem. Este código registra condicionalmente a nova árvore transformada se alguma edição tiver sido feita. Seu regravador só deve modificar uma árvore se encontrar uma ou mais declarações de variáveis locais que poderiam ser simplificadas usando a inferência de tipos:

C#

```
SyntaxNode newSource = rewriter.Visit(sourceTree.GetRoot());

if (newSource != sourceTree.GetRoot())
{
    File.WriteAllText(sourceTree.FilePath, newSource.ToFullString());
}
```

Você deve ver rabiscos abaixo do código `File.WriteAllText`. Selecione a lâmpada e adicione a instrução `using System.IO;` necessária.

Você está quase lá! Resta somente uma etapa: criar um teste Compilation. Como você não usou a inferência de tipos durante este guia de início rápido, este seria um caso de teste perfeito. Infelizmente, criar uma compilação a partir de um arquivo de projeto C# está além do escopo desta explicação passo a passo. Mas, felizmente, se você está seguindo as instruções cuidadosamente, há esperança. Substitua o conteúdo do método `CreateTestCompilation` pelo código a seguir. Ele cria uma compilação de teste que combina coincidentemente com o projeto descrito neste guia de início rápido:

C#

```
String programPath = @"..\..\..\Program.cs";
String programText = File.ReadAllText(programPath);
SyntaxTree programTree =
```

```
                CSharpSyntaxTree.ParseText(programText)
                                .WithFilePath(programPath);

String rewriterPath = @"..\..\..\TypeInferenceRewriter.cs";
String rewriterText = File.ReadAllText(rewriterPath);
SyntaxTree rewriterTree =
                CSharpSyntaxTree.ParseText(rewriterText)
                                .WithFilePath(rewriterPath);

SyntaxTree[] sourceTrees = { programTree, rewriterTree };

MetadataReference mscorlib =
        MetadataReference.CreateFromFile(typeof(object).Assembly.Location);
MetadataReference codeAnalysis =

MetadataReference.CreateFromFile(typeof(SyntaxTree).Assembly.Location);
MetadataReference csharpCodeAnalysis =

MetadataReference.CreateFromFile(typeof(CSharpSyntaxTree).Assembly.Location)
;

MetadataReference[] references = { mscorlib, codeAnalysis,
csharpCodeAnalysis };

return CSharpCompilation.Create("TransformationCS",
    sourceTrees,
    references,
    new CSharpCompilationOptions(OutputKind.ConsoleApplication));
```

Cruze os dedos e execute o projeto. No Visual Studio, escolha **Depurar**>**Iniciar Depuração**. O Visual Studio exibirá um aviso dizendo que os arquivos em seu projeto foram alterados. Clique em "**Sim para Todos**" para recarregar os arquivos modificados. Examine-os para observar sua grandiosidade. Observe como o código parece mais limpo sem todos os especificadores de tipo explícitos e redundantes.

Parabéns! Você usou as **APIs do compilador** para criar sua própria refatoração que pesquisa todos os arquivos em um projeto C# para determinados padrões sintáticos, analisa a semântica do código-fonte que corresponde a esses padrões e os transforma. Agora você é oficialmente um autor de refatoração!

Tutorial: escrever seu primeiro analisador e correção de código

Artigo • 10/05/2023

O SDK do .NET Compiler Platform fornece as ferramentas necessárias para criar diagnósticos personalizados (analisadores), correções de código, refatoração de código e supressores de diagnóstico destinados a códigos C# ou Visual Basic. Um **analisador** contém código que reconhece violações às suas regras. Sua **correção de código** contém o código que corrige a violação. As regras que você implementar podem ser qualquer coisa, incluindo estrutura do código, estilo de codificação, convenções de nomenclatura e muito mais. O .NET Compiler Platform fornece a estrutura para executar análise conforme os desenvolvedores escrevem o código, bem como todos os recursos de interface do usuário do Visual Studio para corrigir o código: mostrar rabiscos no editor, popular a Lista de Erros do Visual Studio, criar as sugestões da "lâmpada" e mostrar a visualização avançada das correções sugeridas.

Neste tutorial, você explorará a criação de um **analisador** e uma **correção de código** que o acompanha, usando as APIs do Roslyn. Um analisador é uma maneira de executar a análise de código-fonte e relatar um problema para o usuário. Opcionalmente, uma correção de código pode ser associada ao analisador para representar uma modificação no código-fonte do usuário. Este tutorial cria um analisador que localiza as declarações de variável local que poderiam ser declaradas usando o modificador `const`, mas não o são. A correção de código anexa modifica essas declarações para adicionar o modificador `const`.

Pré-requisitos

- Visual Studio 2019, versão 16.8 ou posterior

Será necessário instalar o **SDK do .NET Compiler Platform** por meio do Instalador do Visual Studio:

Instruções de instalação – Instalador do Visual Studio

Há duas maneiras diferentes de encontrar o **SDK da .NET Compiler Platform** no **Instalador do Visual Studio**:

Instalar usando o Instalador do Visual Studio – exibição de cargas de trabalho

O SDK da .NET Compiler Platform não é selecionado automaticamente como parte da carga de trabalho de desenvolvimento da extensão do Visual Studio. É necessário selecioná-lo como um componente opcional.

1. Execute o **Instalador do Visual Studio**
2. Escolha **Mudar**
3. Marque a carga de trabalho de **Desenvolvimento de extensão do Visual Studio**.
4. Abra o nó **Desenvolvimento de extensão do Visual Studio** na árvore de resumo.
5. Marque a caixa do **SDK da .NET Compiler Platform**. Você a encontrará por último nos componentes opcionais.

Opcionalmente, você também poderá fazer o **Editor DGML** exibir gráficos no visualizador:

1. Abra o nó **Componentes individuais** na árvore de resumo.
2. Marque a caixa do **Editor DGML**

Instalar usando o Instalador do Visual Studio – guia Componentes individuais

1. Execute o **Instalador do Visual Studio**
2. Escolha **Mudar**
3. Selecione a guia **Componentes individuais**
4. Marque a caixa do **SDK da .NET Compiler Platform**. Você a encontrará na parte superior, na seção **Compiladores, ferramentas de compilação e runtimes**.

Opcionalmente, você também poderá fazer o **Editor DGML** exibir gráficos no visualizador:

1. Marque a caixa do **Editor DGML**. Você a encontrará na seção **Ferramentas de código**.

Há várias etapas para criar e validar o analisador:

1. Crie a solução.
2. Registre o nome e a descrição do analisador.
3. Relate os avisos e recomendações do analisador.
4. Implemente a correção de código para aceitar as recomendações.
5. Melhore a análise por meio de testes de unidade.

Criar a solução

- No Visual Studio, escolha **Arquivo** > **Novo** > **Projeto...** para exibir o diálogo Novo Projeto.
- Em **Visual C#** > **Extensibilidade**, escolha **Analisador com correção de código (.NET Standard)**.
- Nomeie seu projeto como "**MakeConst**" e clique em OK.

> **Observação**
>
> Você pode receber um erro de compilação (*MSB4062: não foi possível carregar a tarefa "CompareBuildTaskVersion"*). Para corrigir isso, atualize os pacotes NuGet na solução com o Gerenciador de Pacotes NuGet ou use `Update-Package` na janela do console do Gerenciador de Pacotes.

Explorar o modelo do analisador

O analisador com o modelo de correção de código cria cinco projetos:

- **MakeConst**, que contém o analisador.
- **MakeConst.CodeFixes**, que contém a correção de código.
- **MakeConst.Package**, que é usado para produzir o pacote NuGet para o analisador e a correção de código.
- **MakeConst.Test**, que é um projeto de teste de unidade.
- **MakeConst.Vsix**, que é o projeto de inicialização padrão que carrega uma segunda instância do Visual Studio com seu novo analisador. Pressione F5 para iniciar o projeto VSIX.

> **Observação**
>
> Os analisadores devem ter como destino o .NET Standard 2.0 porque podem ser executados nos ambientes do .NET Core (builds de linha de comando) e do .NET Framework (Visual Studio).

> **Dica**
>
> Quando você executa seu analisador, você pode iniciar uma segunda cópia do Visual Studio. Essa segunda cópia usa um hive do Registro diferente para armazenar configurações. Isso lhe permite diferenciar as configurações visuais em

duas cópias do Visual Studio. Você pode escolher um tema diferente para a execução experimental do Visual Studio. Além disso, não use perfil móvel de suas configurações nem faça logon na conta do Visual Studio usando a execução experimental do Visual Studio. Isso mantém as diferenças entre as configurações.

O hive inclui não apenas o analisador em desenvolvimento, mas também todos os analisadores anteriores abertos. Para redefinir o hive da Roslyn, você precisa excluí-lo manualmente de *%LocalAppData%\Microsoft\VisualStudio*. O nome da pasta do hive da Roslyn terminará em `Roslyn`, por exemplo, `16.0_9ae182f9Roslyn`. Observe que talvez seja necessário limpar a solução e compilar novamente após excluir o hive.

Na segunda instância do Visual Studio que você acabou de iniciar, crie um projeto de Aplicativo de Console do C# (qualquer estrutura de destino funcionará – os analisadores funcionam no nível de origem). Passe o mouse sobre o token com um sublinhado ondulado e o texto de aviso fornecido por um analisador será exibido.

O modelo cria um analisador que relata um aviso em cada declaração de tipo em que o nome do tipo contém letras minúsculas, conforme mostrado na figura a seguir:

```
namespace ConsoleApp1
{
    0 references
    class Program
    {
        0 references
        static void Main(string[] args)
        {
            Console.WriteLine("Hello World!");
        }
    }
}
```

O modelo também fornece uma correção de código que altera qualquer nome de tipo que contenha caracteres de letras minúsculas, deixando-o com todas as letras maiúsculas. Você pode clicar na lâmpada exibida com o aviso para ver as alterações sugeridas. Aceitar as alterações sugeridas atualiza o nome do tipo e todas as referências para esse tipo na solução. Agora que você já viu o analisador inicial em ação, feche a segunda instância do Visual Studio e retorne ao projeto do analisador.

Você não precisa iniciar uma segunda cópia do Visual Studio e criar um novo código para testar cada alteração em seu analisador. O modelo também cria um projeto de teste de unidade para você. Esse projeto contém dois testes. `TestMethod1` mostra o formato típico de um teste que analisa o código sem disparar um diagnóstico. `TestMethod2` mostra o formato de um teste que dispara um diagnóstico e, em seguida,

aplica uma correção de código sugerida. Conforme você cria o analisador e a correção de código, você escreve testes para estruturas de código diferentes para verificar seu trabalho. Testes de unidade para os analisadores são muito mais rápidos do que testá-los de forma interativa com o Visual Studio.

> **Dica**
>
> Testes de unidade de analisador são uma excelente ferramenta quando você sabe quais constructos de código devem e não devem disparar seu analisador. Carregar o analisador em outra cópia do Visual Studio é uma excelente ferramenta para explorar e encontrar constructos nos quais você talvez não tenha pensado ainda.

Neste tutorial, você grava um analisador que relata ao usuário qualquer declaração de variável local que possa ser convertida em constante local. Por exemplo, considere o seguinte código:

```C#
int x = 0;
Console.WriteLine(x);
```

No código acima, um valor constante é atribuído a `x`, que nunca é modificado. Ele pode ser declarado usando o modificador `const`:

```C#
const int x = 0;
Console.WriteLine(x);
```

A análise para determinar se uma variável pode ser tornada constante está envolvida, exigindo análise sintática, análise constante da expressão de inicializador e também análise de fluxo de dados, para garantir que nunca ocorram gravações na variável. O .NET Compiler Platform fornece APIs que facilitam essa análise.

Criar registros do analisador

O modelo cria a classe inicial `DiagnosticAnalyzer`, no arquivo *MakeConstAnalyzer.cs*. Esse analisador inicial mostra duas propriedades importantes de cada analisador.

- Cada analisador de diagnóstico deve fornecer um atributo `[DiagnosticAnalyzer]` que descreve a linguagem em que opera.

- Cada analisador de diagnóstico deve derivar (direta ou indiretamente) da classe DiagnosticAnalyzer.

O modelo também mostra os recursos básicos que fazem parte de qualquer analisador:

1. Registrar ações. As ações representam alterações de código que devem disparar o analisador para examinar se há violações de código. Quando o Visual Studio detecta as edições de código que correspondem a uma ação registrada, ele chama o método registrado do analisador.
2. Criar diagnósticos. Quando o analisador detecta uma violação, ele cria um objeto de diagnóstico que o Visual Studio usa para notificar o usuário sobre a violação.

Registrar ações na substituição do método DiagnosticAnalyzer.Initialize(AnalysisContext). Neste tutorial, você visitará **nós de sintaxe** em busca de declarações locais e verá quais delas têm valores constantes. Se houver possibilidade de uma declaração ser constante, seu analisador criará e relatará um diagnóstico.

A primeira etapa é atualizar as constantes de registro e o método `Initialize`, de modo que essas constantes indiquem seu analisador "Make Const". A maioria das constantes de cadeia de caracteres é definida no arquivo de recurso de cadeia de caracteres. Você deve seguir essa prática para uma localização mais fácil. Abra o arquivo *Resources.resx* para o projeto do analisador **MakeConst**. Isso exibe o editor de recursos. Atualize os recursos de cadeia de caracteres da seguinte maneira:

- Altere `AnalyzerDescription` para "Variables that are not modified should be made constants.".
- Altere `AnalyzerMessageFormat` para "Variable '{0}' can be made constant".
- Altere `AnalyzerTitle` para "Variable can be made constant".

Quando você terminar, o editor de recursos deverá aparecer conforme mostrado na seguinte figura:

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.

As alterações restantes estão no arquivo do analisador. Abra *MakeConstAnalyzer.cs* no Visual Studio. Altere a ação registrada de uma que age em símbolos para uma que age sobre a sintaxe. No método `MakeConstAnalyzerAnalyzer.Initialize`, localize a linha que registra a ação em símbolos:

```
C#
```

```
context.RegisterSymbolAction(AnalyzeSymbol, SymbolKind.NamedType);
```

Substitua-a com a seguinte linha:

```C#
context.RegisterSyntaxNodeAction(AnalyzeNode,
SyntaxKind.LocalDeclarationStatement);
```

Após essa alteração, você poderá excluir o método `AnalyzeSymbol`. Este analisador examina SyntaxKind.LocalDeclarationStatement, e não instruções SymbolKind.NamedType. Observe que `AnalyzeNode` tem rabiscos vermelhos sob ele. O código apenas que você acaba de adicionar referencia um método `AnalyzeNode` que não foi declarado. Declare esse método usando o seguinte código:

```C#
private void AnalyzeNode(SyntaxNodeAnalysisContext context)
{
}
```

Altere `Category` para "Usage" em *MakeConstAnalyzer.cs*, conforme mostrado no seguinte código:

```C#
private const string Category = "Usage";
```

Localize as declarações locais que podem ser constantes

É hora de escrever a primeira versão do método `AnalyzeNode`. Ele deve procurar uma única declaração local que poderia ser `const` mas não é, algo semelhante ao seguinte código:

```C#
int x = 0;
Console.WriteLine(x);
```

A primeira etapa é encontrar declarações locais. Adicione o seguinte código a `AnalyzeNode` em *MakeConstAnalyzer.cs*:

```
C#

var localDeclaration = (LocalDeclarationStatementSyntax)context.Node;
```

Essa conversão sempre terá êxito porque seu analisador fez o registro para alterações unicamente a declarações locais. Nenhum outro tipo de nó dispara uma chamada para seu método `AnalyzeNode`. Em seguida, verifique a declaração para quaisquer modificadores `const`. Se você encontrá-los, retorne imediatamente. O código a seguir procura por quaisquer modificadores `const` na declaração local:

```
C#

// make sure the declaration isn't already const:
if (localDeclaration.Modifiers.Any(SyntaxKind.ConstKeyword))
{
    return;
}
```

Por fim, você precisa verificar que a variável pode ser `const`. Isso significa assegurar que ela nunca seja atribuída após ser inicializada.

Você executará alguma análise semântica usando o SyntaxNodeAnalysisContext. Você usa o argumento `context` para determinar se a declaração de variável local pode ser tornada `const`. Um Microsoft.CodeAnalysis.SemanticModel representa todas as informações semânticas em apenas um arquivo de origem. Você pode aprender mais no artigo que aborda modelos semânticos. Você usará o Microsoft.CodeAnalysis.SemanticModel para realizar a análise de fluxo de dados na instrução de declaração local. Em seguida, você usa os resultados dessa análise de fluxo de dados para garantir que a variável local não seja escrita com um novo valor em nenhum outro lugar. Chame o método de extensão GetDeclaredSymbol para recuperar o ILocalSymbol para a variável e verifique se ele não está contido na coleção DataFlowAnalysis.WrittenOutside da análise de fluxo de dados. Adicione o seguinte código ao final do método `AnalyzeNode`:

```
C#

// Perform data flow analysis on the local declaration.
DataFlowAnalysis dataFlowAnalysis =
context.SemanticModel.AnalyzeDataFlow(localDeclaration);

// Retrieve the local symbol for each variable in the local declaration
```

```
// and ensure that it is not written outside of the data flow analysis
region.
VariableDeclaratorSyntax variable =
localDeclaration.Declaration.Variables.Single();
ISymbol variableSymbol = context.SemanticModel.GetDeclaredSymbol(variable,
context.CancellationToken);
if (dataFlowAnalysis.WrittenOutside.Contains(variableSymbol))
{
    return;
}
```

O código recém-adicionado garante que a variável não seja modificada e pode, portanto, ser tornada `const`. É hora de gerar o diagnóstico. Adicione o código a seguir como a última linha em `AnalyzeNode`:

```
C#

context.ReportDiagnostic(Diagnostic.Create(Rule, context.Node.GetLocation(),
localDeclaration.Declaration.Variables.First().Identifier.ValueText));
```

Você pode verificar seu andamento pressionando F5 para executar o analisador. Você pode carregar o aplicativo de console que você criou anteriormente e, em seguida, adicionar o seguinte código de teste:

```
C#

int x = 0;
Console.WriteLine(x);
```

A lâmpada deve aparecer e o analisador deve relatar um diagnóstico. No entanto, dependendo da sua versão do Visual Studio, você verá:

- A lâmpada, que ainda usa a correção de código gerada por modelo e dirá a você que ela pode ser colocada em letras maiúsculas.
- Uma mensagem na barra de notificações, na parte superior do editor, informando que o "MakeConstCodeFixProvider" encontrou um erro e foi desabilitado. Isso ocorre porque o provedor de correção de código ainda não foi alterado e espera encontrar `TypeDeclarationSyntax` elementos em vez de `LocalDeclarationStatementSyntax`.

A próxima seção explica como escrever a correção de código.

Escrever a correção de código

Um analisador pode fornecer uma ou mais correções de código. Uma correção de código define uma edição que resolve o problema relatado. Para o analisador que você criou, você pode fornecer uma correção de código que insere a palavra-chave const:

```diff
- int x = 0;
+ const int x = 0;
Console.WriteLine(x);
```

O usuário escolhe-a da lâmpada da interface do usuário no editor e do Visual Studio altera o código.

Abra o arquivo *CodeFixResources.resx* e altere `CodeFixTitle` para "Make constant".

Abra o arquivo *MakeConstCodeFixProvider.cs* adicionado pelo modelo. Essa correção de código já está conectada à ID de Diagnóstico produzida pelo analisador de diagnóstico, mas ela ainda não implementa a transformação de código correta.

Em seguida, exclua o método `MakeUppercaseAsync`. Ele não se aplica mais.

Todos os provedores de correção de código derivam de CodeFixProvider. Todos eles substituem CodeFixProvider.RegisterCodeFixesAsync(CodeFixContext) para relatar as correções de código disponíveis. Em `RegisterCodeFixesAsync`, altere o tipo de nó ancestral pelo qual você está pesquisando para um LocalDeclarationStatementSyntax para corresponder ao diagnóstico:

```C#
var declaration =
root.FindToken(diagnosticSpan.Start).Parent.AncestorsAndSelf().OfType<LocalD
eclarationStatementSyntax>().First();
```

Em seguida, altere a última linha para registrar uma correção de código. A correção criará um novo documento resultante da adição do modificador `const` para uma declaração existente:

```C#
// Register a code action that will invoke the fix.
context.RegisterCodeFix(
    CodeAction.Create(
        title: CodeFixResources.CodeFixTitle,
        createChangedDocument: c => MakeConstAsync(context.Document,
declaration, c),
```

```
        equivalenceKey: nameof(CodeFixResources.CodeFixTitle)),
    diagnostic);
```

Você observará rabiscos vermelhos no código que você acabou de adicionar no símbolo `MakeConstAsync`. Adicione uma declaração para `MakeConstAsync` semelhante ao seguinte código:

```
C#

private static async Task<Document> MakeConstAsync(Document document,
    LocalDeclarationStatementSyntax localDeclaration,
    CancellationToken cancellationToken)
{
}
```

Seu novo método `MakeConstAsync` transformará o Document que representa o arquivo de origem do usuário em um novo Document que agora contém uma declaração `const`.

Você cria um novo token de palavra-chave `const` a ser inserido no início da instrução de declaração. Tenha cuidado para remover qualquer desafio à esquerda do primeiro token de instrução de declaração e anexe-o ao token `const`. Adicione o seguinte código ao método `MakeConstAsync`:

```
C#

// Remove the leading trivia from the local declaration.
SyntaxToken firstToken = localDeclaration.GetFirstToken();
SyntaxTriviaList leadingTrivia = firstToken.LeadingTrivia;
LocalDeclarationStatementSyntax trimmedLocal =
localDeclaration.ReplaceToken(
    firstToken, firstToken.WithLeadingTrivia(SyntaxTriviaList.Empty));

// Create a const token with the leading trivia.
SyntaxToken constToken = SyntaxFactory.Token(leadingTrivia,
SyntaxKind.ConstKeyword,
SyntaxFactory.TriviaList(SyntaxFactory.ElasticMarker));
```

Em seguida, adicione o token `const` à declaração usando o seguinte código:

```
C#

// Insert the const token into the modifiers list, creating a new modifiers
list.
SyntaxTokenList newModifiers = trimmedLocal.Modifiers.Insert(0, constToken);
// Produce the new local declaration.
LocalDeclarationStatementSyntax newLocal = trimmedLocal
```

```
    .WithModifiers(newModifiers)
    .WithDeclaration(localDeclaration.Declaration);
```

Em seguida, formate a nova declaração de acordo com regras de formatação de C#. Formatar de suas alterações para corresponderem ao código existente cria uma experiência melhor. Adicione a instrução a seguir imediatamente após o código existente:

```C#
// Add an annotation to format the new local declaration.
LocalDeclarationStatementSyntax formattedLocal =
newLocal.WithAdditionalAnnotations(Formatter.Annotation);
```

Um novo namespace é necessário para esse código. Adicione a seguinte diretiva `using` para a parte superior do arquivo:

```C#
using Microsoft.CodeAnalysis.Formatting;
```

A etapa final é fazer a edição. Há três etapas para esse processo:

1. Obter um identificador para o documento existente.
2. Criar um novo documento, substituindo a declaração existente pela nova declaração.
3. Retornar o novo documento.

Adicione o seguinte código ao final do método `MakeConstAsync`:

```C#
// Replace the old local declaration with the new local declaration.
SyntaxNode oldRoot = await
document.GetSyntaxRootAsync(cancellationToken).ConfigureAwait(false);
SyntaxNode newRoot = oldRoot.ReplaceNode(localDeclaration, formattedLocal);

// Return document with transformed tree.
return document.WithSyntaxRoot(newRoot);
```

A correção de código está pronta para ser experimentada. Pressione F5 para executar o projeto do analisador em uma segunda instância do Visual Studio. Na segunda instância do Visual Studio, crie um novo projeto de Aplicativo de Console em C# e adicione algumas declarações de variável local inicializadas com valores constantes para o

método Main. Você verá que elas são relatadas como avisos, conforme mostrado a seguir.

```
static void Main(string[] args)
{
    int i = 1;
    int j = 2;
    int k = i + j;
}
```

Você fez muito progresso. Há rabiscos sob as declarações que podem ser tornados `const`. Mas ainda há trabalho a fazer. Isso funciona bem se você adicionar `const` às declarações começando com `i`, depois `j` e, por fim, `k`. Mas se você adicionar o modificador `const` em uma ordem diferente, começando com `k`, seu analisador criará erros: `k` não pode ser declarado como `const`, a menos que `i` e `j` já sejam ambos `const`. Você tem que fazer mais análise para assegurar que lida com as diferentes maneiras em que variáveis podem ser declaradas e inicializadas.

Criar testes de unidade

Seu analisador e correção de código trabalham em um caso simples de uma única declaração que pode ser tornada const. Há várias instruções de declaração possíveis em que essa implementação comete erros. Você tratará desses casos trabalhando com a biblioteca de teste de unidade gravada pelo modelo. Isso é muito mais rápido do que abrir repetidamente uma segunda cópia do Visual Studio.

Abra o arquivo *MakeConstUnitTests.cs* no projeto de teste de unidade. O modelo criou dois testes que seguem os dois padrões comuns para um analisador e o teste de unidade de correção de código. `TestMethod1` mostra o padrão para um teste que garante que o analisador não relata um diagnóstico quando não deve fazê-lo. `TestMethod2` mostra o padrão para relatar um diagnóstico e executar a correção de código.

O modelo usa pacotes Microsoft.CodeAnalysis.Testing para testes de unidade.

> **Dica**
>
> A biblioteca de testes dá suporte a uma sintaxe de marcação especial, incluindo o seguinte:
>
> - `[|text|]`: indica que um diagnóstico é relatado para `text`. Por padrão, esse formulário só pode ser usado para testar analisadores com exatamente um

`DiagnosticDescriptor` fornecido por `DiagnosticAnalyzer.SupportedDiagnostics`.

- `{|ExpectedDiagnosticId:text|}`: indica que um diagnóstico com Id `ExpectedDiagnosticId` é relatado para `text`.

Substitua os testes de modelo na classe `MakeConstUnitTest` pelo seguinte método de teste:

C#

```
        [TestMethod]
        public async Task LocalIntCouldBeConstant_Diagnostic()
        {
            await VerifyCS.VerifyCodeFixAsync(@"
using System;

class Program
{
    static void Main()
    {
        [|int i = 0;|]
        Console.WriteLine(i);
    }
}
", @"
using System;

class Program
{
    static void Main()
    {
        const int i = 0;
        Console.WriteLine(i);
    }
}
");
        }
```

Execute este teste para garantir que ele passa. No Visual Studio, abra o **Gerenciador de Testes** selecionando **Teste**>**Windows**>**Gerenciador de Testes**. Então selecione **Executar tudo**.

Criar testes para declarações válidas

Como regra geral, os analisadores devem sair assim que possível, fazendo o mínimo de trabalho. O Visual Studio chama analisadores registrados conforme o usuário edita o

código. A capacidade de resposta é um requisito fundamental. Há vários casos de teste para o código que não deverão gerar o diagnóstico. O analisador já gerencia um desses testes, o caso em que uma variável é atribuída após ser inicializada. Adicione o seguinte método de teste para representar esse caso:

C#

```
        [TestMethod]
        public async Task VariableIsAssigned_NoDiagnostic()
        {
            await VerifyCS.VerifyAnalyzerAsync(@"
using System;

class Program
{
    static void Main()
    {
        int i = 0;
        Console.WriteLine(i++);
    }
}
");
        }
```

Esse teste é aprovado também. Em seguida, adicione métodos de teste para condições que você ainda não gerenciou:

- Declarações que já são `const`, porque elas já são const:

  C#

  ```
          [TestMethod]
          public async Task VariableIsAlreadyConst_NoDiagnostic()
          {
              await VerifyCS.VerifyAnalyzerAsync(@"
  using System;

  class Program
  {
      static void Main()
      {
          const int i = 0;
          Console.WriteLine(i);
      }
  }
  ");
          }
  ```

- Declarações que não têm nenhum inicializador, porque não há nenhum valor a ser usado:

C#

```
        [TestMethod]
        public async Task NoInitializer_NoDiagnostic()
        {
            await VerifyCS.VerifyAnalyzerAsync(@"
using System;

class Program
{
    static void Main()
    {
        int i;
        i = 0;
        Console.WriteLine(i);
    }
}
");
        }
```

- Declarações em que o inicializador não é uma constante, porque elas não podem ser constantes de tempo de compilação:

C#

```
        [TestMethod]
        public async Task InitializerIsNotConstant_NoDiagnostic()
        {
            await VerifyCS.VerifyAnalyzerAsync(@"
using System;

class Program
{
    static void Main()
    {
        int i = DateTime.Now.DayOfYear;
        Console.WriteLine(i);
    }
}
");
        }
```

Isso pode ser ainda mais complicado, porque o C# permite várias declarações como uma instrução. Considere a seguinte constante de cadeia de caracteres de caso de teste:

C#

```
        [TestMethod]
        public async Task MultipleInitializers_NoDiagnostic()
        {
            await VerifyCS.VerifyAnalyzerAsync(@"
using System;

class Program
{
    static void Main()
    {
        int i = 0, j = DateTime.Now.DayOfYear;
        Console.WriteLine(i);
        Console.WriteLine(j);
    }
}
");
        }
```

A variável `i` pode ser tornada constante, mas o mesmo não se aplica à variável `j`. Portanto, essa instrução não pode ser tornada uma declaração const.

Execute os testes novamente e você verá esses novos casos de teste falharem.

Atualize seu analisador para ignorar as declarações corretas

Você precisa de algumas melhorias no método `AnalyzeNode` do analisador para filtrar o código que corresponde a essas condições. Elas são todas condições relacionadas, portanto, alterações semelhantes corrigirão todas essas condições. Faça as alterações a seguir em `AnalyzeNode`:

- A análise semântica examinou uma única declaração de variável. Esse código deve estar em um loop de `foreach` que examina todas as variáveis declaradas na mesma instrução.
- Cada variável declarada precisa ter um inicializador.
- O inicializador de cada variável declarada precisa ser uma constante de tempo de compilação.

No seu método `AnalyzeNode`, substitua a análise semântica original:

C#

```
// Perform data flow analysis on the local declaration.
DataFlowAnalysis dataFlowAnalysis =
context.SemanticModel.AnalyzeDataFlow(localDeclaration);
```

```
// Retrieve the local symbol for each variable in the local declaration
// and ensure that it is not written outside of the data flow analysis
region.
VariableDeclaratorSyntax variable =
localDeclaration.Declaration.Variables.Single();
ISymbol variableSymbol = context.SemanticModel.GetDeclaredSymbol(variable,
context.CancellationToken);
if (dataFlowAnalysis.WrittenOutside.Contains(variableSymbol))
{
    return;
}
```

com o snippet de código a seguir:

```C#
// Ensure that all variables in the local declaration have initializers that
// are assigned with constant values.
foreach (VariableDeclaratorSyntax variable in
localDeclaration.Declaration.Variables)
{
    EqualsValueClauseSyntax initializer = variable.Initializer;
    if (initializer == null)
    {
        return;
    }

    Optional<object> constantValue =
context.SemanticModel.GetConstantValue(initializer.Value,
context.CancellationToken);
    if (!constantValue.HasValue)
    {
        return;
    }
}

// Perform data flow analysis on the local declaration.
DataFlowAnalysis dataFlowAnalysis =
context.SemanticModel.AnalyzeDataFlow(localDeclaration);

foreach (VariableDeclaratorSyntax variable in
localDeclaration.Declaration.Variables)
{
    // Retrieve the local symbol for each variable in the local declaration
    // and ensure that it is not written outside of the data flow analysis
region.
    ISymbol variableSymbol =
context.SemanticModel.GetDeclaredSymbol(variable,
context.CancellationToken);
    if (dataFlowAnalysis.WrittenOutside.Contains(variableSymbol))
    {
        return;
```

```
    }
}
```

O primeiro loop de `foreach` examina cada declaração de variável usando a análise sintática. A primeira verificação garante que a variável tenha um inicializador. A segunda verificação garante que o inicializador seja uma constante. O segundo loop tem a análise semântica original. As verificações semânticas estão em um loop separado porque ele tem um impacto maior no desempenho. Execute os testes novamente e você deverá ver todos eles serem aprovados.

Adicionar o final polonês

Você está quase lá. Há mais algumas condições com as quais o seu analisador deve lidar. Enquanto o usuário está escrevendo código, o Visual Studio chama os analisadores. Muitas vezes o analisador será chamado para código que não é compilado. O método `AnalyzeNode` do analisador de diagnóstico não verifica para ver se o valor da constante é conversível para o tipo de variável. Assim, a implementação atual converterá facilmente uma declaração incorreta, tal como `int i = "abc"`, em uma constante local. Adicione um método de teste para este caso:

C#

```
        [TestMethod]
        public async Task DeclarationIsInvalid_NoDiagnostic()
        {
            await VerifyCS.VerifyAnalyzerAsync(@"
using System;

class Program
{
    static void Main()
    {
        int x = {|CS0029:""abc""|};
    }
}
");
        }
```

Além disso, os tipos de referência não são tratados corretamente. O único valor de constante permitido para um tipo de referência é `null`, exceto no caso de System.String, que permite literais de cadeia de caracteres. Em outras palavras, `const string s = "abc"` é legal, mas `const object s = "abc"` não é. Este snippet de código verifica essa condição:

C#

```
        [TestMethod]
        public async Task DeclarationIsNotString_NoDiagnostic()
        {
            await VerifyCS.VerifyAnalyzerAsync(@"
using System;

class Program
{
    static void Main()
    {
        object s = ""abc"";
    }
}
");
        }
```

Para ser criterioso, você precisará adicionar outro teste para verificar se pode criar uma declaração de constante para uma cadeia de caracteres. O snippet de código a seguir define o código que gera o diagnóstico e o código após a aplicação da correção:

C#

```
        [TestMethod]
        public async Task StringCouldBeConstant_Diagnostic()
        {
            await VerifyCS.VerifyCodeFixAsync(@"
using System;

class Program
{
    static void Main()
    {
        [|string s = ""abc"";|]
    }
}
", @"
using System;

class Program
{
    static void Main()
    {
        const string s = ""abc"";
    }
}
");
        }
```

Por fim, se uma variável é declarada com a palavra-chave `var`, a correção de código faz a coisa errada e gera uma declaração `const var`, que não é compatível com a linguagem C#. Para corrigir esse bug, a correção de código deve substituir a palavra-chave `var` pelo nome do tipo inferido:

C#

```
        [TestMethod]
        public async Task VarIntDeclarationCouldBeConstant_Diagnostic()
        {
            await VerifyCS.VerifyCodeFixAsync(@"
using System;

class Program
{
    static void Main()
    {
        [|var item = 4;|]
    }
}
", @"
using System;

class Program
{
    static void Main()
    {
        const int item = 4;
    }
}
");
        }

        [TestMethod]
        public async Task VarStringDeclarationCouldBeConstant_Diagnostic()
        {
            await VerifyCS.VerifyCodeFixAsync(@"
using System;

class Program
{
    static void Main()
    {
        [|var item = ""abc"";|]
    }
}
", @"
using System;

class Program
{
    static void Main()
    {
```

```
        const string item = ""abc"";
    }
}
");
        }
```

Felizmente, todos os erros acima podem ser resolvidos usando as mesmas técnicas que você acabou de aprender.

Para corrigir o primeiro bug, primeiro abra *DiagnosticAnalyzer.cs* e localize o loop foreach em que cada um dos inicializadores de declaração local é verificado para garantir que valores constantes sejam atribuídos a eles. Imediatamente *antes* do primeiro loop foreach, chame `context.SemanticModel.GetTypeInfo()` para recuperar informações detalhadas sobre o tipo declarado da declaração local:

```C#
TypeSyntax variableTypeName = localDeclaration.Declaration.Type;
ITypeSymbol variableType =
context.SemanticModel.GetTypeInfo(variableTypeName,
context.CancellationToken).ConvertedType;
```

Em seguida, dentro do loop `foreach`, verifique cada inicializador para garantir que ele pode ser convertido no tipo de variável. Adicione a seguinte verificação depois de garantir que o inicializador é uma constante:

```C#
// Ensure that the initializer value can be converted to the type of the
// local declaration without a user-defined conversion.
Conversion conversion =
context.SemanticModel.ClassifyConversion(initializer.Value, variableType);
if (!conversion.Exists || conversion.IsUserDefined)
{
    return;
}
```

A próxima alteração é realizada com base na última. Antes da chave de fechamento do primeiro loop foreach, adicione o código a seguir para verificar o tipo da declaração de local quando a constante é uma cadeia de caracteres ou valor nulo.

```C#
// Special cases:
//  * If the constant value is a string, the type of the local declaration
//    must be System.String.
//  * If the constant value is null, the type of the local declaration must
```

```
//    be a reference type.
if (constantValue.Value is string)
{
    if (variableType.SpecialType != SpecialType.System_String)
    {
        return;
    }
}
else if (variableType.IsReferenceType && constantValue.Value != null)
{
    return;
}
```

Você precisa escrever um pouco mais de código no seu provedor de correção de código para substituir a palavra-chave `var` pelo nome do tipo correto. Retorne para *MakeConstCodeFixProvider.cs*. O código que você adicionará realizará as seguintes etapas:

- Verifique se a declaração é uma declaração `var` e se afirmativo:
- Crie um novo tipo para o tipo inferido.
- Certifique-se de que a declaração de tipo não é um alias. Em caso afirmativo, é legal declarar `const var`.
- Certifique-se de que `var` não é um nome de tipo neste programa. (Em caso afirmativo, `const var` é legal).
- Simplificar o nome completo do tipo

Isso soa como muito código. Mas não é. Substitua a linha que declara e inicializa `newLocal` com o código a seguir. Ele é colocado imediatamente após a inicialização de `newModifiers`:

C#

```
// If the type of the declaration is 'var', create a new type name
// for the inferred type.
VariableDeclarationSyntax variableDeclaration =
localDeclaration.Declaration;
TypeSyntax variableTypeName = variableDeclaration.Type;
if (variableTypeName.IsVar)
{
    SemanticModel semanticModel = await
document.GetSemanticModelAsync(cancellationToken).ConfigureAwait(false);

    // Special case: Ensure that 'var' isn't actually an alias to another
type
    // (e.g. using var = System.String).
    IAliasSymbol aliasInfo = semanticModel.GetAliasInfo(variableTypeName,
cancellationToken);
    if (aliasInfo == null)
```

```
    {
        // Retrieve the type inferred for var.
        ITypeSymbol type = semanticModel.GetTypeInfo(variableTypeName,
cancellationToken).ConvertedType;

        // Special case: Ensure that 'var' isn't actually a type named
'var'.
        if (type.Name != "var")
        {
            // Create a new TypeSyntax for the inferred type. Be careful
            // to keep any leading and trailing trivia from the var keyword.
            TypeSyntax typeName =
SyntaxFactory.ParseTypeName(type.ToDisplayString())
                .WithLeadingTrivia(variableTypeName.GetLeadingTrivia())
                .WithTrailingTrivia(variableTypeName.GetTrailingTrivia());

            // Add an annotation to simplify the type name.
            TypeSyntax simplifiedTypeName =
typeName.WithAdditionalAnnotations(Simplifier.Annotation);

            // Replace the type in the variable declaration.
            variableDeclaration =
variableDeclaration.WithType(simplifiedTypeName);
        }
    }
}
// Produce the new local declaration.
LocalDeclarationStatementSyntax newLocal =
trimmedLocal.WithModifiers(newModifiers)
                            .WithDeclaration(variableDeclaration);
```

Você precisará adicionar uma diretiva `using` para usar o tipo Simplifier:

```
C#

using Microsoft.CodeAnalysis.Simplification;
```

Execute seus testes, que devem todos ser aprovados. Dê parabéns a si mesmo, executando seu analisador concluído. Pressione `Ctrl`+`F5` para executar o projeto do analisador em uma segunda instância do Visual Studio com a extensão de versão prévia da Roslyn carregada.

- Na segunda instância do Visual Studio, crie um novo projeto de Aplicativo de Console de C# e adicione `int x = "abc";` ao método Main. Graças à primeira correção de bug, nenhum aviso deve ser relatado para esta declaração de variável local (embora haja um erro do compilador, conforme esperado).
- Em seguida, adicione `object s = "abc";` ao método Main. Devido à segunda correção de bug, nenhum aviso deve ser relatado.

- Por fim, adicione outra variável local que usa a palavra-chave `var`. Você verá que um aviso é relatado e uma sugestão é exibida abaixo e a esquerda.
- Mova o cursor do editor sobre o sublinhado ondulado e pressione `Ctrl`+`.`. para exibir a correção de código sugerida. Ao selecionar a correção de código, observe que a palavra-chave `var` agora é gerenciada corretamente.

Por fim, adicione o seguinte código:

```
C#

int i = 2;
int j = 32;
int k = i + j;
```

Após essas alterações, você obtém linhas onduladas vermelhas apenas nas duas primeiras variáveis. Adicione `const` para ambos `i` e `j`, e você receberá um novo aviso em `k` porque ele agora pode ser `const`.

Parabéns! Você criou sua primeira extensão do .NET Compiler Platform que executa análise de código com o sistema em funcionamento para detectar um problema e fornece uma correção rápida para corrigi-lo. Ao longo do caminho, você aprendeu muitas das APIs de código que fazem parte do SDK do .NET Compiler Platform (APIs do Roslyn). Você pode verificar seu trabalho comparando-o à amostra concluída em nosso repositório GitHub de exemplos.

Outros recursos

- Introdução à análise de sintaxe
- Introdução à análise semântica

Conceitos de programação (C#)

17/06/2025

Esta seção explica os conceitos de programação na linguagem C#.

Nesta seção

Expandir a tabela

Título	Descrição
Covariância e contravariância (C#)	Mostra como habilitar a conversão implícita de parâmetros de tipo genérico em interfaces e delegados.
Iteradores (C#)	Descreve os iteradores, que são usados para percorrer coleções e retornar elementos um de cada vez.

Seções relacionadas

- Dicas de desempenho

  Discute várias regras básicas que podem ajudá-lo a aumentar o desempenho do aplicativo.

Covariância e contravariância (C#)

20/06/2025

No C#, a covariância e a contravariância habilitam a conversão de referência implícita para tipos de matriz, tipos de delegados e argumentos de tipo genérico. A covariância preserva a compatibilidade de atribuição e a contravariância a reverte.

O código a seguir demonstra a diferença entre compatibilidade de atribuição, covariância e contravariância.

C#

```
// Assignment compatibility.
string str = "test";
// An object of a more derived type is assigned to an object of a less derived
type.
object obj = str;

// Covariance.
IEnumerable<string> strings = new List<string>();
// An object that is instantiated with a more derived type argument
// is assigned to an object instantiated with a less derived type argument.
// Assignment compatibility is preserved.
IEnumerable<object> objects = strings;

// Contravariance.
// Assume that the following method is in the class:
static void SetObject(object o) { }
Action<object> actObject = SetObject;
// An object that is instantiated with a less derived type argument
// is assigned to an object instantiated with a more derived type argument.
// Assignment compatibility is reversed.
Action<string> actString = actObject;
```

A covariância para matrizes permite a conversão implícita de uma matriz de um tipo mais derivado em uma matriz de um tipo menos derivado. Mas essa operação não é tipo seguro, conforme mostrado no exemplo de código a seguir.

C#

```
object[] array = new String[10];
// The following statement produces a run-time exception.
// array[0] = 10;
```

O suporte de covariância e contravariância aos grupos de método permite a correspondência de assinaturas de método com tipos de delegados. Isso permite atribuir aos delegados não apenas métodos que têm assinaturas correspondentes, mas também métodos que retornam

tipos mais derivados (covariância) ou que aceitam parâmetros que têm tipos menos derivados (contravariância) do que os especificados pelo tipo delegado. Para obter mais informações, consulte Variação em Delegados (C#) e Uso da Variação em Delegados (C#).

O exemplo de código a seguir mostra o suporte a covariância e contravariância para grupos de métodos.

C#

```
static object GetObject() { return null; }
static void SetObject(object obj) { }

static string GetString() { return ""; }
static void SetString(string str) { }

static void Test()
{
    // Covariance. A delegate specifies a return type as object,
    // but you can assign a method that returns a string.
    Func<object> del = GetString;

    // Contravariance. A delegate specifies a parameter type as string,
    // but you can assign a method that takes an object.
    Action<string> del2 = SetObject;
}
```

No .NET Framework 4 e versões posteriores, o C# dá suporte a covariância e contravariância em interfaces e delegados genéricos e permite a conversão implícita de parâmetros de tipo genérico. Para obter mais informações, consulte Variação em Interfaces Genéricas (C#) e Variação em Delegados (C#).

O exemplo de código a seguir mostra a conversão de referência implícita para interfaces genéricas.

C#

```
IEnumerable<String> strings = new List<String>();
IEnumerable<Object> objects = strings;
```

Uma interface ou delegado genérico é chamado *de variante* se seus parâmetros genéricos forem declarados covariantes ou contravariantes. O C# permite que você crie suas próprias interfaces e delegados personalizados. Para obter mais informações, consulte Criando interfaces genéricas variantes (C#) e Variação em delegados (C#).

Tópicos relacionados

Expandir a tabela

Título	Descrição
Variação em interfaces genéricas (C#)	Discute covariância e contravariância em interfaces genéricas e fornece uma lista de interfaces genéricas variantes no .NET.
Criando interfaces genéricas variantes (C#)	Mostra como criar interfaces variantes personalizadas.
Usando variação em interfaces para coleções genéricas (C#)	Mostra como o suporte à covariância e à contravariância nas interfaces IEnumerable<T> e IComparable<T> pode ajudá-lo a reutilizar o código.
Variação em delegados (C#)	Discute a covariância e a contravariância em interfaces genéricas e não genéricas, e fornece uma lista de delegados genéricos variáveis no .NET.
Usando variação em delegados (C#)	Mostra como usar o suporte de covariância e contravariância em delegados não genéricos para corresponder às assinaturas de método com tipos delegados.
Usar variação para delegados genéricos Func e Action (C#)	Mostra como o suporte de covariância e contravariância nos delegados `Func` e `Action` pode ajudar na reutilização do código.

Variação em interfaces genéricas (C#)

19/06/2025

O .NET Framework 4 introduziu suporte de variação para várias interfaces genéricas existentes. O suporte à variação permite a conversão implícita de classes que implementam essas interfaces.

A partir do .NET Framework 4, as seguintes interfaces são variantes:

- IEnumerable<T> (T é covariante)
- IEnumerator<T> (T é covariante)
- IQueryable<T> (T é covariante)
- IGrouping<TKey,TElement> (`TKey` e `TElement` são covariantes)
- IComparer<T> (T é contravariante)
- IEqualityComparer<T> (T é contravariante)
- IComparable<T> (T é contravariante)

A partir do .NET Framework 4.5, as seguintes interfaces são variantes:

- IReadOnlyList<T> (T é covariante)
- IReadOnlyCollection<T> (T é covariante)

A covariância permite que um método tenha um tipo de retorno mais derivado do que o definido pelo parâmetro de tipo genérico da interface. Para ilustrar o recurso de covariância, considere essas interfaces genéricas: `IEnumerable<Object>` e `IEnumerable<String>`. A `IEnumerable<String>` interface não herda a `IEnumerable<Object>` interface. No entanto, o `String` tipo herda o `Object` tipo e, em alguns casos, talvez você queira atribuir objetos dessas interfaces uns aos outros. Isso é mostrado no exemplo de código a seguir.

C#

```
IEnumerable<String> strings = new List<String>();
IEnumerable<Object> objects = strings;
```

Em versões anteriores do .NET Framework, esse código causa um erro de compilação em C# e, se `Option Strict` estiver ativado, no Visual Basic. Mas agora você pode usar `strings` em vez de `objects`, conforme mostrado no exemplo anterior, porque a IEnumerable<T> interface é covariante.

A contravariância permite que um método tenha tipos de argumento menos derivados do que o especificado pelo parâmetro genérico da interface. Para ilustrar a contravariância, suponha que você tenha criado uma `BaseComparer` classe para comparar instâncias da `BaseClass` classe. A classe `BaseComparer` implementa a interface `IEqualityComparer<BaseClass>`. Como a IEqualityComparer<T> interface agora é contravariante, você pode usar `BaseComparer` para comparar instâncias de classes que herdam a `BaseClass` classe. Isso é mostrado no exemplo de código a seguir.

C#

```
// Simple hierarchy of classes.
class BaseClass { }
class DerivedClass : BaseClass { }

// Comparer class.
class BaseComparer : IEqualityComparer<BaseClass>
{
    public int GetHashCode(BaseClass baseInstance)
    {
        return baseInstance.GetHashCode();
    }
    public bool Equals(BaseClass x, BaseClass y)
    {
        return x == y;
    }
}
class Program
{
    static void Test()
    {
        IEqualityComparer<BaseClass> baseComparer = new BaseComparer();

        // Implicit conversion of IEqualityComparer<BaseClass> to
        // IEqualityComparer<DerivedClass>.
        IEqualityComparer<DerivedClass> childComparer = baseComparer;
    }
}
```

Para obter mais exemplos, consulte Como usar variação em interfaces para coleções genéricas (C#).

A variação em interfaces genéricas tem suporte somente para tipos de referência. Os tipos de valor não dão suporte à variação. Por exemplo, `IEnumerable<int>` não pode ser convertido `IEnumerable<object>` implicitamente em , porque os inteiros são representados por um tipo de valor.

C#

```
IEnumerable<int> integers = new List<int>();
// The following statement generates a compiler error,
// because int is a value type.
// IEnumerable<Object> objects = integers;
```

Também é importante lembrar que as classes que implementam interfaces variantes ainda são invariáveis. Por exemplo, embora List<T> implemente a interface IEnumerable<T>covariante, você não pode converter `List<String>` implicitamente em `List<Object>`. Isso é ilustrado no exemplo de código a seguir.

C#

```
// The following line generates a compiler error
// because classes are invariant.
// List<Object> list = new List<String>();

// You can use the interface object instead.
IEnumerable<Object> listObjects = new List<String>();
```

Consulte também

- Usando variação em interfaces para coleções genéricas (C#)
- Criando interfaces genéricas variantes (C#)
- Interfaces genéricas
- Variação em delegados (C#)

Criando interfaces genéricas variantes (C#)

20/06/2025

Você pode declarar parâmetros de tipo genérico em interfaces como covariantes ou contravariantes. *A covariância* permite que os métodos de interface tenham tipos de retorno mais derivados do que os definidos pelos parâmetros de tipo genérico. *A contravariância* permite que os métodos de interface tenham tipos de argumento menos derivados do que os especificados pelos parâmetros genéricos. Uma interface genérica que tem parâmetros de tipo genérico covariantes ou contravariantes é chamada *de variante*.

> ⓘ **Observação**
>
> O .NET Framework 4 introduziu suporte de variação para várias interfaces genéricas existentes. Para obter a lista das interfaces variantes no .NET, consulte Variação em Interfaces Genéricas (C#).

Declarando interfaces genéricas variantes

Você pode declarar interfaces genéricas variantes usando as palavras-chave `in` e `out` para parâmetros de tipo genérico.

> ⓘ **Importante**
>
> `ref`, `in` e `out` os parâmetros em C# não podem ser variantes. Os tipos de valor também não dão suporte à variação.

Você pode declarar um covariante de parâmetro de tipo genérico usando a `out` palavra-chave. O tipo covariante deve atender às seguintes condições:

- O tipo é usado apenas como um tipo de retorno de métodos de interface e não é usado como um tipo de argumentos de método. Isso é ilustrado no exemplo a seguir, no qual o tipo `R` é declarado covariante.

```C#
interface ICovariant<out R>
{
    R GetSomething();
    // The following statement generates a compiler error.
    // void SetSomething(R sampleArg);
```

```
}
```

Há uma exceção a essa regra. Se você tiver um delegado genérico contravariante como um parâmetro de método, você poderá usar o tipo como um parâmetro de tipo genérico para o delegado. Isso é ilustrado pelo tipo `R` no exemplo a seguir. Para obter mais informações, consulte Variância em delegados (C#) e Usando variância para delegados genéricos Func e Action (C#).

C#

```
interface ICovariant<out R>
{
    void DoSomething(Action<R> callback);
}
```

- O tipo não é usado como uma restrição genérica para os métodos de interface. Isso é ilustrado no código a seguir.

C#

```
interface ICovariant<out R>
{
    // The following statement generates a compiler error
    // because you can use only contravariant or invariant types
    // in generic constraints.
    // void DoSomething<T>() where T : R;
}
```

Você pode declarar um parâmetro de tipo genérico contravariante usando a `in` palavra-chave. O tipo contravariante pode ser usado apenas como um tipo de argumentos de método e não como um tipo de retorno dos métodos de interface. O tipo contravariante também pode ser usado para restrições genéricas. O código a seguir mostra como declarar uma interface contravariante e usar uma restrição genérica para um de seus métodos.

C#

```
interface IContravariant<in A>
{
    void SetSomething(A sampleArg);
    void DoSomething<T>() where T : A;
    // The following statement generates a compiler error.
    // A GetSomething();
}
```

Também é possível dar suporte à covariância e à contravariância na mesma interface, mas para parâmetros de tipo diferentes, conforme mostrado no exemplo de código a seguir.

C#

```
interface IVariant<out R, in A>
{
    R GetSomething();
    void SetSomething(A sampleArg);
    R GetSetSomethings(A sampleArg);
}
```

Implementando interfaces genéricas variantes

Você implementa interfaces genéricas variantes em classes usando a mesma sintaxe usada para interfaces invariáveis. O exemplo de código a seguir mostra como implementar uma interface covariante em uma classe genérica.

C#

```
interface ICovariant<out R>
{
    R GetSomething();
}
class SampleImplementation<R> : ICovariant<R>
{
    public R GetSomething()
    {
        // Some code.
        return default(R);
    }
}
```

Classes que implementam interfaces variantes são invariáveis. Por exemplo, considere o código a seguir.

C#

```
// The interface is covariant.
ICovariant<Button> ibutton = new SampleImplementation<Button>();
ICovariant<Object> iobj = ibutton;

// The class is invariant.
SampleImplementation<Button> button = new SampleImplementation<Button>();
// The following statement generates a compiler error
// because classes are invariant.
// SampleImplementation<Object> obj = button;
```

Estendendo interfaces genéricas variantes

Ao estender uma interface genérica variante, você precisa usar as palavras-chave `in` e `out` para especificar explicitamente se a interface derivada suporta variação. O compilador não infere a variação da interface que está sendo estendida. Por exemplo, considere as interfaces a seguir.

C#

```
interface ICovariant<out T> { }
interface IInvariant<T> : ICovariant<T> { }
interface IExtCovariant<out T> : ICovariant<T> { }
```

Na interface `IInvariant<T>`, o parâmetro de tipo genérico `T` é invariável, enquanto na `IExtCovariant<out T>`, o parâmetro de tipo é covariante, embora ambas estendam a mesma interface. A mesma regra é aplicada a parâmetros de tipo genérico contravariantes.

Você pode criar uma interface que estenda tanto a interface em que o parâmetro `T` de tipo genérico é covariante quanto a interface em que ele é contravariante, desde que, na interface resultante, o parâmetro `T` de tipo genérico seja invariável. Isso é ilustrado no exemplo de código a seguir.

C#

```
interface ICovariant<out T> { }
interface IContravariant<in T> { }
interface IInvariant<T> : ICovariant<T>, IContravariant<T> { }
```

No entanto, se um parâmetro `T` de tipo genérico for declarado covariante em uma interface, você não poderá declará-lo contravariante na interface de extensão ou vice-versa. Isso é ilustrado no exemplo de código a seguir.

C#

```
interface ICovariant<out T> { }
// The following statement generates a compiler error.
// interface ICoContraVariant<in T> : ICovariant<T> { }
```

Evitando ambiguidade

Quando você implementa interfaces genéricas variantes, a variação às vezes pode levar à ambiguidade. Essa ambiguidade deve ser evitada.

Por exemplo, se você implementar explicitamente a mesma interface genérica variante com diferentes parâmetros de tipo genérico em uma classe, ela poderá criar ambiguidade. O compilador não produz um erro nesse caso, mas não é especificado qual implementação de interface será escolhida em tempo de execução. Essa ambiguidade pode levar a bugs sutis em seu código. Considere o exemplo de código a seguir.

C#

```
// Simple class hierarchy.
class Animal { }
class Cat : Animal { }
class Dog : Animal { }

// This class introduces ambiguity
// because IEnumerable<out T> is covariant.
class Pets : IEnumerable<Cat>, IEnumerable<Dog>
{
    IEnumerator<Cat> IEnumerable<Cat>.GetEnumerator()
    {
        Console.WriteLine("Cat");
        // Some code.
        return null;
    }

    IEnumerator IEnumerable.GetEnumerator()
    {
        // Some code.
        return null;
    }

    IEnumerator<Dog> IEnumerable<Dog>.GetEnumerator()
    {
        Console.WriteLine("Dog");
        // Some code.
        return null;
    }
}
class Program
{
    public static void Test()
    {
        IEnumerable<Animal> pets = new Pets();
        pets.GetEnumerator();
    }
}
```

Neste exemplo, não é especificado como o `pets.GetEnumerator` método escolhe entre `Cat` e `Dog`. Isso pode causar problemas em seu código.

Consulte também

- Variação em interfaces genéricas (C#)
- Usar variação para delegados genéricos Func e Action (C#)

Usando variação em interfaces para coleções genéricas (C#)

17/06/2025

Uma interface covariante permite que seus métodos retornem tipos mais derivados do que os especificados na interface. Uma interface contravariante permite que seus métodos aceitem parâmetros de tipos menos derivados do que os especificados na interface.

No .NET Framework 4, várias interfaces existentes tornaram-se covariantes e contravariantes. Elas incluem IEnumerable<T> e IComparable<T>. Isso permite reutilizar métodos que operam com coleções genéricas de tipos base para coleções de tipos derivados.

Para obter uma lista de interfaces variantes no .NET, consulte Variação em Interfaces Genéricas (C#).

Convertendo coleções genéricas

O exemplo a seguir ilustra os benefícios do suporte à covariância na IEnumerable<T> interface. O `PrintFullName` método aceita uma coleção do `IEnumerable<Person>` tipo como um parâmetro. No entanto, você pode reutilizá-lo para uma coleção do `IEnumerable<Employee>` tipo porque `Employee` herda `Person`.

C#

```
// Simple hierarchy of classes.
public class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
}

public class Employee : Person { }

class Program
{
    // The method has a parameter of the IEnumerable<Person> type.
    public static void PrintFullName(IEnumerable<Person> persons)
    {
        foreach (Person person in persons)
        {
            Console.WriteLine("Name: {0} {1}",
            person.FirstName, person.LastName);
        }
    }
```

```
    public static void Test()
    {
        IEnumerable<Employee> employees = new List<Employee>();

        // You can pass IEnumerable<Employee>,
        // although the method expects IEnumerable<Person>.

        PrintFullName(employees);

    }
}
```

Comparando coleções genéricas

O exemplo a seguir ilustra os benefícios do suporte à contravariância na IEqualityComparer<T> interface. A classe `PersonComparer` implementa a interface `IEqualityComparer<Person>`. No entanto, você pode reutilizar essa classe para comparar uma sequência de objetos do `Employee` tipo porque `Employee` herda `Person`.

C#

```
// Simple hierarchy of classes.
public class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
}

public class Employee : Person { }

// The custom comparer for the Person type
// with standard implementations of Equals()
// and GetHashCode() methods.
class PersonComparer : IEqualityComparer<Person>
{
    public bool Equals(Person x, Person y)
    {
        if (Object.ReferenceEquals(x, y)) return true;
        if (Object.ReferenceEquals(x, null) ||
            Object.ReferenceEquals(y, null))
            return false;
        return x.FirstName == y.FirstName && x.LastName == y.LastName;
    }
    public int GetHashCode(Person person)
    {
        if (Object.ReferenceEquals(person, null)) return 0;
        int hashFirstName = person.FirstName == null
            ? 0 : person.FirstName.GetHashCode();
        int hashLastName = person.LastName.GetHashCode();
        return hashFirstName ^ hashLastName;
```

```
    }
}

class Program
{

    public static void Test()
    {
        List<Employee> employees = new List<Employee> {
               new Employee() {FirstName = "Michael", LastName = "Alexander"},
               new Employee() {FirstName = "Jeff", LastName = "Price"}
            };

        // You can pass PersonComparer,
        // which implements IEqualityComparer<Person>,
        // although the method expects IEqualityComparer<Employee>.

        IEnumerable<Employee> noduplicates =
            employees.Distinct<Employee>(new PersonComparer());

        foreach (var employee in noduplicates)
            Console.WriteLine(employee.FirstName + " " + employee.LastName);
    }
}
```

Consulte também

- Variação em interfaces genéricas (C#)

Variação em delegados (C#)

19/06/2025

O .NET Framework 3.5 introduziu o suporte a variação para assinaturas de método correspondentes com tipos de delegados em todos os delegados do C#. Isso significa que você pode atribuir aos delegados não apenas métodos que têm assinaturas correspondentes, mas também métodos que retornam tipos mais derivados (covariância) ou que aceitam parâmetros que têm tipos menos derivados (contravariância) do que os especificados pelo tipo delegado. Isso inclui delegados genéricos e não genéricos.

Por exemplo, considere o código a seguir, que tem duas classes e dois delegados: genérico e não genérico.

C#

```
public class First { }
public class Second : First { }
public delegate First SampleDelegate(Second a);
public delegate R SampleGenericDelegate<A, R>(A a);
```

Ao criar delegados dos tipos `SampleDelegate` ou `SampleGenericDelegate<A, R>`, você pode atribuir qualquer um dos métodos a seguir a esses delegados.

C#

```
// Matching signature.
public static First ASecondRFirst(Second second)
{ return new First(); }

// The return type is more derived.
public static Second ASecondRSecond(Second second)
{ return new Second(); }

// The argument type is less derived.
public static First AFirstRFirst(First first)
{ return new First(); }

// The return type is more derived
// and the argument type is less derived.
public static Second AFirstRSecond(First first)
{ return new Second(); }
```

O exemplo de código a seguir ilustra a conversão implícita entre a assinatura do método e o tipo delegado.

C#

```
// Assigning a method with a matching signature
// to a non-generic delegate. No conversion is necessary.
SampleDelegate dNonGeneric = ASecondRFirst;
// Assigning a method with a more derived return type
// and less derived argument type to a non-generic delegate.
// The implicit conversion is used.
SampleDelegate dNonGenericConversion = AFirstRSecond;

// Assigning a method with a matching signature to a generic delegate.
// No conversion is necessary.
SampleGenericDelegate<Second, First> dGeneric = ASecondRFirst;
// Assigning a method with a more derived return type
// and less derived argument type to a generic delegate.
// The implicit conversion is used.
SampleGenericDelegate<Second, First> dGenericConversion = AFirstRSecond;
```

Para obter mais informações, consulte Usando variação em delegados (C#) e Usando variação para os delegados genéricos Func e Action (C#).

Variação em parâmetros de tipo genérico

No .NET Framework 4 ou posterior, você pode habilitar a conversão implícita entre delegados, para que delegados genéricos que tenham tipos diferentes especificados por parâmetros de tipo genérico possam ser atribuídos uns aos outros, se os tipos forem herdados uns dos outros, conforme exigido pela variação.

Para habilitar a conversão implícita, você deve declarar explicitamente os parâmetros genéricos em um delegado como covariantes ou contravariantes usando a palavra-chave `in` ou a palavra-chave `out`.

O exemplo de código a seguir mostra como você pode criar um delegado que tenha um parâmetro de tipo genérico covariante.

C#

```
// Type T is declared covariant by using the out keyword.
public delegate T SampleGenericDelegate <out T>();

public static void Test()
{
    SampleGenericDelegate <String> dString = () => " ";

    // You can assign delegates to each other,
    // because the type T is declared covariant.
    SampleGenericDelegate <Object> dObject = dString;
}
```

Se você usar apenas o suporte à variação para corresponder assinaturas de método com tipos delegados e não usar as palavras-chave `in` e `out`, você pode achar que às vezes pode instanciar delegados com métodos ou expressões lambda idênticas, mas não pode atribuir um delegado a outro.

No exemplo de código a seguir, `SampleGenericDelegate<String>` não é possível converter `SampleGenericDelegate<Object>` explicitamente, embora `String` herda `Object`. Você pode corrigir esse problema marcando o parâmetro `T` genérico com a `out` palavra-chave.

C#

```
public delegate T SampleGenericDelegate<T>();

public static void Test()
{
    SampleGenericDelegate<String> dString = () => " ";

    // You can assign the dObject delegate
    // to the same lambda expression as dString delegate
    // because of the variance support for
    // matching method signatures with delegate types.
    SampleGenericDelegate<Object> dObject = () => " ";

    // The following statement generates a compiler error
    // because the generic type T is not marked as covariant.
    // SampleGenericDelegate <Object> dObject = dString;

}
```

Delegados genéricos que têm parâmetros de tipo variante no .NET

O .NET Framework 4 introduziu suporte de variação para parâmetros de tipo genérico em vários delegados genéricos existentes:

- `Action` delega do namespace System, por exemplo, Action<T> e Action<T1,T2>
- `Func` delega do namespace System, por exemplo, Func<TResult> e Func<T,TResult>
- O delegado Predicate<T>
- O delegado Comparison<T>
- O delegado Converter<TInput,TOutput>

Para obter mais informações e exemplos, consulte Usando variação para delegados genéricos Func e Action (C#).

Declarando parâmetros de tipo variante em delegados genéricos

Se um delegado genérico tiver parâmetros de tipo genérico covariantes ou contravariantes, ele poderá ser chamado de *delegado genérico variante*.

Você pode declarar um covariante de parâmetro de tipo genérico em um delegado genérico usando a `out` palavra-chave. O tipo covariante pode ser usado apenas como um tipo de retorno de método e não como um tipo de argumentos de método. O exemplo de código a seguir mostra como declarar um delegado genérico covariante.

```
C#

public delegate R DCovariant<out R>();
```

Você pode declarar um parâmetro de tipo genérico contravariante em um delegado genérico usando a palavra-chave `in`. O tipo contravariante pode ser usado apenas como um tipo de argumentos de método e não como um tipo de retorno de método. O exemplo de código a seguir mostra como declarar um delegado genérico contravariante.

```
C#

public delegate void DContravariant<in A>(A a);
```

> **Importante**
>
> `ref`, `in` e `out` os parâmetros em C# não podem ser marcados como variante.

Também é possível dar suporte à variância e à covariância no mesmo delegado, mas para parâmetros de tipo diferente. Isso é mostrado no exemplo a seguir.

```
C#

public delegate R DVariant<in A, out R>(A a);
```

Instanciando e invocando delegados genéricos variantes

Você pode instanciar e invocar delegados variantes assim como instanciar e invocar delegados invariáveis. No exemplo a seguir, o delegado é instanciado por uma expressão lambda.

```
C#
```

```
DVariant<String, String> dvariant = (String str) => str + " ";
dvariant("test");
```

Combinando delegados genéricos variantes

Não combine delegados variantes. O método Combine não dá suporte à conversão de delegados variantes e espera que os delegados sejam exatamente do mesmo tipo. Isso pode levar a uma exceção em tempo de execução quando você combina delegados, seja pelo método Combine ou pelo operador `+`, conforme mostrado no exemplo de código a seguir.

C#

```
Action<object> actObj = x => Console.WriteLine("object: {0}", x);
Action<string> actStr = x => Console.WriteLine("string: {0}", x);
// All of the following statements throw exceptions at run time.
// Action<string> actCombine = actStr + actObj;
// actStr += actObj;
// Delegate.Combine(actStr, actObj);
```

Variação em parâmetros de tipo genérico para tipos de valor e referência

Há suporte para variação para parâmetros de tipo genérico somente para tipos de referência. Por exemplo, `DVariant<int>` não pode ser convertido implicitamente em `DVariant<Object>` ou `DVariant<long>`, porque inteiro é um tipo de valor.

O exemplo a seguir demonstra que não há suporte para variação em parâmetros de tipo genérico para tipos de valor.

C#

```
// The type T is covariant.
public delegate T DVariant<out T>();

// The type T is invariant.
public delegate T DInvariant<T>();

public static void Test()
{
    int i = 0;
    DInvariant<int> dInt = () => i;
    DVariant<int> dVariantInt = () => i;

    // All of the following statements generate a compiler error
```

```
    // because type variance in generic parameters is not supported
    // for value types, even if generic type parameters are declared variant.
    // DInvariant<Object> dObject = dInt;
    // DInvariant<long> dLong = dInt;
    // DVariant<Object> dVariantObject = dVariantInt;
    // DVariant<long> dVariantLong = dVariantInt;
}
```

Consulte também

- Genéricos
- Usar variação para delegados genéricos Func e Action (C#)
- Como combinar delegados (delegados multicast)

Usando variação em delegações (C#)

19/06/2025

Quando você atribui um método a um delegado, *a covariância* e *a contravariância* fornecem flexibilidade para a correspondência de um tipo delegado com uma assinatura de método. A covariância permite que um método tenha um tipo de retorno mais derivado do que o definido no delegado. A contravariância permite um método que tem tipos de parâmetro que são menos derivados do que aqueles no tipo delegado.

Exemplo 1: Covariância

Descrição

Este exemplo demonstra como os delegados podem ser usados com métodos que têm tipos de retorno derivados do tipo de retorno na assinatura do delegado. O tipo de dados retornado por `DogsHandler` é do tipo `Dogs`, que deriva do `Mammals` tipo definido no delegado.

Código

C#

```
class Mammals {}
class Dogs : Mammals {}

class Program
{
    // Define the delegate.
    public delegate Mammals HandlerMethod();

    public static Mammals MammalsHandler()
    {
        return null;
    }

    public static Dogs DogsHandler()
    {
        return null;
    }

    static void Test()
    {
        HandlerMethod handlerMammals = MammalsHandler;

        // Covariance enables this assignment.
        HandlerMethod handlerDogs = DogsHandler;
```

```
    }
}
```

Exemplo 2: contravariância

Descrição

Este exemplo demonstra como os delegados podem ser usados com métodos que têm parâmetros cujos tipos são tipos base do tipo de parâmetro de assinatura delegado. Com o conceito de contravariância, você pode utilizar um único manipulador de eventos em vez de handlers separados. O exemplo a seguir usa dois delegados:

- Um representante KeyEventHandler que define a assinatura do evento Button.KeyDown. Sua assinatura é:

  C#

  ```
  public delegate void KeyEventHandler(object sender, KeyEventArgs e)
  ```

- Um representante MouseEventHandler que define a assinatura do evento Button.MouseClick. Sua assinatura é:

  C#

  ```
  public delegate void MouseEventHandler(object sender, MouseEventArgs e)
  ```

O exemplo define um manipulador de eventos com um parâmetro EventArgs e o usa para manipular tanto os eventos `Button.KeyDown` quanto os eventos `Button.MouseClick`. Ele pode fazer isso porque EventArgs é um tipo base de ambos KeyEventArgs e MouseEventArgs.

Código

C#

```
// Event handler that accepts a parameter of the EventArgs type.
private void MultiHandler(object sender, System.EventArgs e)
{
    label1.Text = System.DateTime.Now.ToString();
}

public Form1()
{
    InitializeComponent();
```

```
    // You can use a method that has an EventArgs parameter,
    // although the event expects the KeyEventArgs parameter.
    this.button1.KeyDown += this.MultiHandler;

    // You can use the same method
    // for an event that expects the MouseEventArgs parameter.
    this.button1.MouseClick += this.MultiHandler;

}
```

Consulte também

- Variação em delegados (C#)
- Usar variação para delegados genéricos Func e Action (C#)

Usando variância para delegados genéricos Func e Action (C#)

19/06/2025

Esses exemplos demonstram como usar covariância e contravariância nos delegados `Func` e `Action` genéricos para permitir a reutilização de métodos e fornecer mais flexibilidade em seu código.

Para obter mais informações sobre covariância e contravariância, consulte Variância em Delegados (C#).

Usando delegados com parâmetros de tipo covariante

O exemplo a seguir ilustra os benefícios do suporte à covariância nos delegados genéricos `Func`. O `FindByTitle` método usa um parâmetro do `String` tipo e retorna um objeto do `Employee` tipo. No entanto, você pode atribuir esse método ao `Func<String, Person>` delegado porque `Employee` herda `Person`.

C#

```
// Simple hierarchy of classes.
public class Person { }
public class Employee : Person { }
class Program
{
    static Employee FindByTitle(String title)
    {
        // This is a stub for a method that returns
        // an employee that has the specified title.
        return new Employee();
    }

    static void Test()
    {
        // Create an instance of the delegate without using variance.
        Func<String, Employee> findEmployee = FindByTitle;

        // The delegate expects a method to return Person,
        // but you can assign it a method that returns Employee.
        Func<String, Person> findPerson = FindByTitle;

        // You can also assign a delegate
        // that returns a more derived type
        // to a delegate that returns a less derived type.
```

```
        findPerson = findEmployee;

    }
}
```

Usando delegados com parâmetros de tipo contravariantes

O exemplo a seguir ilustra os benefícios do suporte à contravariância nos delegados genéricos `Action`. O `AddToContacts` método usa um parâmetro do `Person` tipo. No entanto, você pode atribuir esse método ao `Action<Employee>` delegado porque `Employee` herda `Person`.

C#

```
public class Person { }
public class Employee : Person { }
class Program
{
    static void AddToContacts(Person person)
    {
        // This method adds a Person object
        // to a contact list.
    }

    static void Test()
    {
        // Create an instance of the delegate without using variance.
        Action<Person> addPersonToContacts = AddToContacts;

        // The Action delegate expects
        // a method that has an Employee parameter,
        // but you can assign it a method that has a Person parameter
        // because Employee derives from Person.
        Action<Employee> addEmployeeToContacts = AddToContacts;

        // You can also assign a delegate
        // that accepts a less derived parameter to a delegate
        // that accepts a more derived parameter.
        addEmployeeToContacts = addPersonToContacts;
    }
}
```

Consulte também

- Covariância e contravariância (C#)
- Genéricos

Iteradores (C#)

17/06/2025

Um *iterador* pode ser usado para percorrer coleções como listas e matrizes.

Um método iterador ou um acessador `get` executa uma iteração personalizada em uma coleção. Um método iterador usa a instrução de retorno de rendimento para retornar cada elemento um de cada vez. Quando uma instrução `yield return` for atingida, o local atual no código será lembrado. Na próxima vez que a função de iterador for chamada, a execução é reiniciada desse local.

Um iterador é consumido no código cliente, usando uma instrução foreach ou usando uma consulta LINQ.

No exemplo a seguir, a primeira iteração do `foreach` loop faz com que a execução prossiga no `SomeNumbers` método iterador até que a primeira `yield return` instrução seja atingida. Essa iteração retorna um valor de 3 e o local atual no método iterador é mantido. Na próxima iteração do loop, a execução no método iterador continua de onde parou, parando novamente quando atingir uma `yield return` declaração. Essa iteração retorna um valor de 5 e a localização atual no método iterador é retida novamente. O loop é concluído quando o final do método iterador é atingido.

C#

```
static void Main()
{
    foreach (int number in SomeNumbers())
    {
        Console.Write(number.ToString() + " ");
    }
    // Output: 3 5 8
    Console.ReadKey();
}

public static System.Collections.IEnumerable SomeNumbers()
{
    yield return 3;
    yield return 5;
    yield return 8;
}
```

O tipo de retorno de um método iterador ou acessador `get` pode ser IEnumerable, IEnumerable<T>, IEnumerator ou IEnumerator<T>.

Você pode usar uma `yield break` instrução para encerrar a iteração.

> ⓘ **Observação**
>
> Todos os exemplos neste tópico, exceto o exemplo Iterador Simples, incluem diretivas using para os namespaces `System.Collections` e `System.Collections.Generic`.

Iterador simples

O exemplo a seguir tem uma única `yield return` instrução que está dentro de um loop for . Em `Main`, cada iteração do corpo da instrução `foreach` cria uma chamada para a função iteradora, que avança para a próxima instrução `yield return`.

C#

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

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

Criando uma classe de coleção

No exemplo a seguir, a `DaysOfTheWeek` classe implementa a IEnumerable interface, que requer um GetEnumerator método. O compilador chama implicitamente o `GetEnumerator` método, que retorna um IEnumerator.

O `GetEnumerator` método retorna cada cadeia de caracteres uma de cada vez usando a `yield return` instrução.

C#

```
static void Main()
{
    DaysOfTheWeek days = new DaysOfTheWeek();

    foreach (string day in days)
    {
        Console.Write(day + " ");
    }
    // Output: Sun Mon Tue Wed Thu Fri Sat
    Console.ReadKey();
}

public class DaysOfTheWeek : IEnumerable
{
    private string[] days = ["Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat"];

    public IEnumerator GetEnumerator()
    {
        for (int index = 0; index < days.Length; index++)
        {
            // Yield each day of the week.
            yield return days[index];
        }
    }
}
```

O exemplo a seguir cria uma `Zoo` classe que contém uma coleção de animais.

A instrução `foreach` que se refere à instância de classe `theZoo` implicitamente chama o método `GetEnumerator`. As instruções `foreach` que se referem às propriedades `Birds` e `Mammals` utilizam o método iterador nomeado `AnimalsForType`.

C#

```
static void Main()
{
    Zoo theZoo = new Zoo();

    theZoo.AddMammal("Whale");
    theZoo.AddMammal("Rhinoceros");
    theZoo.AddBird("Penguin");
    theZoo.AddBird("Warbler");

    foreach (string name in theZoo)
    {
        Console.Write(name + " ");
    }
    Console.WriteLine();
    // Output: Whale Rhinoceros Penguin Warbler
```

```
    foreach (string name in theZoo.Birds)
    {
        Console.Write(name + " ");
    }
    Console.WriteLine();
    // Output: Penguin Warbler

    foreach (string name in theZoo.Mammals)
    {
        Console.Write(name + " ");
    }
    Console.WriteLine();
    // Output: Whale Rhinoceros

    Console.ReadKey();
}

public class Zoo : IEnumerable
{
    // Private members.
    private List<Animal> animals = new List<Animal>();

    // Public methods.
    public void AddMammal(string name)
    {
        animals.Add(new Animal { Name = name, Type = Animal.TypeEnum.Mammal });
    }

    public void AddBird(string name)
    {
        animals.Add(new Animal { Name = name, Type = Animal.TypeEnum.Bird });
    }

    public IEnumerator GetEnumerator()
    {
        foreach (Animal theAnimal in animals)
        {
            yield return theAnimal.Name;
        }
    }

    // Public members.
    public IEnumerable Mammals
    {
        get { return AnimalsForType(Animal.TypeEnum.Mammal); }
    }

    public IEnumerable Birds
    {
        get { return AnimalsForType(Animal.TypeEnum.Bird); }
    }

    // Private methods.
    private IEnumerable AnimalsForType(Animal.TypeEnum type)
```

```
    {
        foreach (Animal theAnimal in animals)
        {
            if (theAnimal.Type == type)
            {
                yield return theAnimal.Name;
            }
        }
    }

    // Private class.
    private class Animal
    {
        public enum TypeEnum { Bird, Mammal }

        public string Name { get; set; }
        public TypeEnum Type { get; set; }
    }
}
```

Usando iteradores com uma lista genérica

No exemplo a seguir, a Stack<T> classe genérica implementa a IEnumerable<T> interface genérica. O Push método atribui valores a uma matriz do tipo `T`. O GetEnumerator método retorna os valores da matriz usando a `yield return` instrução.

Além do método genérico GetEnumerator , o método não genérico GetEnumerator também deve ser implementado. Isso ocorre porque IEnumerable<T> herda de IEnumerable. A implementação não genérica adia a implementação genérica.

O exemplo usa iteradores nomeados para dar suporte a várias maneiras de iterar na mesma coleção de dados. Esses iteradores nomeados são as propriedades `TopToBottom` e `BottomToTop`, e o método `TopN`.

A propriedade `BottomToTop` usa um iterador em um acessador `get`.

C#

```
static void Main()
{
    Stack<int> theStack = new Stack<int>();

    //  Add items to the stack.
    for (int number = 0; number <= 9; number++)
    {
        theStack.Push(number);
    }
```

```
    // Retrieve items from the stack.
    // foreach is allowed because theStack implements IEnumerable<int>.
    foreach (int number in theStack)
    {
        Console.Write("{0} ", number);
    }
    Console.WriteLine();
    // Output: 9 8 7 6 5 4 3 2 1 0

    // foreach is allowed, because theStack.TopToBottom returns IEnumerable(Of
Integer).
    foreach (int number in theStack.TopToBottom)
    {
        Console.Write("{0} ", number);
    }
    Console.WriteLine();
    // Output: 9 8 7 6 5 4 3 2 1 0

    foreach (int number in theStack.BottomToTop)
    {
        Console.Write("{0} ", number);
    }
    Console.WriteLine();
    // Output: 0 1 2 3 4 5 6 7 8 9

    foreach (int number in theStack.TopN(7))
    {
        Console.Write("{0} ", number);
    }
    Console.WriteLine();
    // Output: 9 8 7 6 5 4 3

    Console.ReadKey();
}

public class Stack<T> : IEnumerable<T>
{
    private T[] values = new T[100];
    private int top = 0;

    public void Push(T t)
    {
        values[top] = t;
        top++;
    }
    public T Pop()
    {
        top--;
        return values[top];
    }

    // This method implements the GetEnumerator method. It allows
    // an instance of the class to be used in a foreach statement.
    public IEnumerator<T> GetEnumerator()
    {
```

```
        for (int index = top - 1; index >= 0; index--)
        {
            yield return values[index];
        }
    }

    IEnumerator IEnumerable.GetEnumerator()
    {
        return GetEnumerator();
    }

    public IEnumerable<T> TopToBottom
    {
        get { return this; }
    }

    public IEnumerable<T> BottomToTop
    {
        get
        {
            for (int index = 0; index <= top - 1; index++)
            {
                yield return values[index];
            }
        }
    }

    public IEnumerable<T> TopN(int itemsFromTop)
    {
        // Return less than itemsFromTop if necessary.
        int startIndex = itemsFromTop >= top ? 0 : top - itemsFromTop;

        for (int index = top - 1; index >= startIndex; index--)
        {
            yield return values[index];
        }
    }

}
```

Informações de sintaxe

Um iterador pode ocorrer como um método ou como um acessador `get`. Um iterador não pode ser utilizado em um evento, em um construtor de instância, em um construtor estático ou em um finalizador estático.

Uma conversão implícita deve existir do tipo de expressão na declaração `yield return` para o argumento de tipo para o elemento `IEnumerable<T>` retornado pelo iterador.

Em C#, um método de iterador não pode ter nenhum parâmetro `in`, `ref` ou `out`.

Em C#, `yield` não é uma palavra reservada e tem um significado especial somente quando é usada antes de uma palavra-chave `return` ou `break`.

Implementação técnica

Embora você escreva um iterador como um método, o compilador o converte em uma classe aninhada que é, de fato, uma máquina de estado. Essa classe mantém o controle da posição do iterador enquanto o loop `foreach` no código cliente continuar.

Para ver o que o compilador faz, você pode usar a ferramenta Ildasm.exe para exibir o código de idioma intermediário comum gerado para um método iterador.

Ao criar um iterador para uma classe ou struct, você não precisa implementar toda IEnumerator a interface. Quando o compilador detecta o iterador, ele gera automaticamente os métodos `Current`, `MoveNext` e `Dispose` da interface IEnumerator ou IEnumerator<T>.

A cada iteração sucessiva do loop `foreach` (ou a chamada direta ao `IEnumerator.MoveNext`), o próximo corpo de código do iterador continua, depois da instrução `yield return` anterior. Em seguida, ele continuará até a próxima instrução `yield return`, até que o final do corpo do iterador seja alcançado ou até que uma instrução `yield break` seja encontrada.

Os iteradores não dão suporte ao IEnumerator.Reset método. Para iterar novamente desde o início, você deve obter um novo iterador. Chamar Reset no iterador retornado por um método iterador lança um NotSupportedException.

Para obter informações adicionais, consulte a Especificação da Linguagem C#.

Uso de Iteradores

Os iteradores permitem manter a simplicidade de um `foreach` loop quando você precisa usar código complexo para preencher uma sequência de lista. Isso pode ser útil quando você deseja fazer o seguinte:

- Modificar a sequência de lista após a primeira iteração de loop `foreach`.
- Evite carregar totalmente uma lista grande antes da primeira iteração de um `foreach` loop. Um exemplo é uma busca paginada para carregar um lote de linhas da tabela. Outro exemplo é o EnumerateFiles método, que implementa iteradores no .NET.
- Encapsular a criação da lista no iterador. No método iterador, você pode criar a lista e, em seguida, produzir cada resultado em um loop.

Consulte também

- System.Collections.Generic
- IEnumerable<T>
- foreach, in
- Genéricos

Instruções (Guia de Programação em C#)

19/06/2025

As ações executadas por um programa são expressas em instruções. As ações comuns incluem declarar variáveis, atribuir valores, chamar métodos, fazer loop por meio de coleções e ramificar para um ou outro bloco de código, dependendo de uma determinada condição. A ordem na qual as instruções são executadas em um programa é chamada de fluxo de controle ou fluxo de execução. O fluxo de controle pode variar sempre que um programa é executado, dependendo de como o programa reage à entrada que recebe em tempo de execução.

Uma instrução pode consistir em uma única linha de código que termina em um ponto-e-vírgula ou uma série de instruções de linha única em um bloco. Um bloco de instrução é colocado entre colchetes {} e pode conter blocos aninhados. O código a seguir mostra dois exemplos de instruções de linha única e um bloco de instrução de várias linhas:

C#

```
    public static void Main()
    {
        // Declaration statement.
        int counter;

        // Assignment statement.
        counter = 1;

        // Error! This is an expression, not an expression statement.
        // counter + 1;

        // Declaration statements with initializers are functionally
        // equivalent to  declaration statement followed by assignment statement:
        int[] radii = [15, 32, 108, 74, 9]; // Declare and initialize an array.
        const double pi = 3.14159; // Declare and initialize  constant.

        // foreach statement block that contains multiple statements.
        foreach (int radius in radii)
        {
            // Declaration statement with initializer.
            double circumference = pi * (2 * radius);

            // Expression statement (method invocation). A single-line
            // statement can span multiple text lines because line breaks
            // are treated as white space, which is ignored by the compiler.
            System.Console.WriteLine($"Radius of circle #{counter} is {radius}.
Circumference = {circumference:N2}");

            // 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
*/
```

Tipos de declarações

A tabela a seguir lista os vários tipos de instruções em C# e suas palavras-chave associadas, com links para tópicos que incluem mais informações:

Expandir a tabela

Categoria	Palavras-chave/anotações em C#
Declarações	Uma instrução de declaração apresenta uma nova variável ou constante. Uma declaração de variável pode, opcionalmente, atribuir um valor à variável. Uma declaração constante, a atribuição é obrigatória.
Declarações de expressão	As instruções de expressão que calculam um valor devem armazenar o valor em uma variável.
Instruções de seleção	As instruções de seleção permitem ramificar-se em diferentes seções de código, dependendo de uma ou mais condições especificadas. Para obter mais informações, consulte os seguintes tópicos: • se • alternar
Instruções de iteração	As instruções de iteração permitem que você faça loop por meio de coleções como matrizes ou execute o mesmo conjunto de instruções repetidamente até que uma condição especificada seja atendida. Para obter mais informações, consulte os seguintes tópicos: • fazer • para • foreach • enquanto
Instruções de atalho	Instruções de salto transferem o controle para outra seção do código. Para obter mais informações, consulte os seguintes tópicos: • de interrupção • continuar • Goto • retornar

Categoria	Palavras-chave/anotações em C#
	• rendimento
Instruções para manipulação de exceções	Instruções para tratamento de exceções permitem que você se recupere normalmente de condições excepcionais que ocorrem em tempo de execução. Para obter mais informações, consulte os seguintes tópicos: • jogar • try-catch • try-finally • try-catch-finally
checked e unchecked	As instruções `checked` e `unchecked` permitem especificar se as operações numéricas de tipo integral podem causar um estouro quando o resultado é armazenado em uma variável muito pequena para conter o valor resultante.
A instrução `await`	Se você marcar um método com o modificador assíncrono , poderá usar o operador await no método. Quando o controle atinge uma `await` expressão no método assíncrono, o controle retorna ao chamador e o progresso no método é suspenso até que a tarefa aguardada seja concluída. Quando a tarefa for concluída, a execução poderá ser retomada no método. Para obter um exemplo simples, consulte a seção "Métodos Assíncronos" de Métodos. Para obter mais informações, consulte Programação assíncrona com async e await.
A instrução `yield return`	Um iterador realiza uma iteração personalizada em uma coleção, como uma lista ou uma matriz. Um iterador usa a instrução de retorno de rendimento para retornar cada elemento um de cada vez. Quando uma instrução `yield return` for atingida, o local atual no código será lembrado. A execução é reiniciada desse local quando o iterador é chamado na próxima vez. Para obter mais informações, consulte Iteradores.
A instrução `fixed`	A instrução fixa impede que o coletor de lixo realoque uma variável móvel. Para obter mais informações, consulte fixed.
A instrução `lock`	A instrução de bloqueio permite limitar o acesso a blocos de código a apenas um thread de cada vez. Para obter mais informações, consulte lock.
Declarações rotuladas	Você pode dar um rótulo a uma instrução e, em seguida, usar a palavra-chave goto para ir para a instrução rotulada. (Consulte o exemplo na linha a seguir.)
A instrução vazia	A instrução vazia consiste em um único ponto e vírgula. Ele não faz nada e pode ser usado em situações onde uma declaração é necessária, mas nenhuma ação precisa ser executada.

Instruções de declaração

O código a seguir mostra exemplos de declarações variáveis com e sem uma atribuição inicial e uma declaração constante com a inicialização necessária.

```C#
// Variable declaration statements.
double area;
double radius = 2;

// Constant declaration statement.
const double pi = 3.14159;
```

Instruções de expressão

O código a seguir mostra exemplos de instruções de expressão, incluindo atribuição, criação de objeto com atribuição e invocação de método.

```C#
// Expression statement (assignment).
area = 3.14 * (radius * radius);

// Expression statement (result discarded).
int x = 0;
x++;

// Expression statement (method invocation).
System.Console.WriteLine();

// Expression statement (new object creation).
System.Collections.Generic.List<string> strings =
    new System.Collections.Generic.List<string>();
```

A instrução vazia

Os exemplos a seguir mostram dois usos para uma declaração vazia:

```C#
void ProcessMessages()
{
    while (ProcessMessage())
        ; // Statement needed here.
}

void F()
{
```

```
    //...
    if (done) goto exit;
//...
exit:
    ; // Statement needed here.
}
```

Declarações incorporadas

Algumas instruções, por exemplo, instruções de iteração, sempre têm uma instrução inserida que as segue. Essa declaração inserida pode ser uma única declaração ou várias declarações delimitadas por colchetes angulares {} em um bloco de declaração. Até mesmo instruções inseridas de uma única linha podem ser colocadas entre colchetes {}, conforme mostrado no seguinte exemplo:

C#

```
// Recommended style. Embedded statement in  block.
foreach (string s in System.IO.Directory.GetDirectories(
                        System.Environment.CurrentDirectory))
{
    System.Console.WriteLine(s);
}

// Not recommended.
foreach (string s in System.IO.Directory.GetDirectories(
                        System.Environment.CurrentDirectory))
    System.Console.WriteLine(s);
```

Uma declaração inserida que não está entre colchetes {} não pode ser uma declaração ou uma declaração rotulada. Isso é mostrado no exemplo a seguir:

C#

```
if(pointB == true)
    //Error CS1023:
    int radius = 5;
```

Coloque a instrução inserida em um bloco para corrigir o erro:

C#

```
if (b == true)
{
    // OK:
    System.DateTime d = System.DateTime.Now;
```

```
    System.Console.WriteLine(d.ToLongDateString());
}
```

Blocos de instrução aninhados

Blocos de instrução podem ser aninhados, conforme mostrado no código a seguir:

C#

```
foreach (string s in System.IO.Directory.GetDirectories(
    System.Environment.CurrentDirectory))
{
    if (s.StartsWith("CSharp"))
    {
        if (s.EndsWith("TempFolder"))
        {
            return s;
        }
    }
}
return "Not found.";
```

Declarações inalcançáveis

Se o compilador determinar que o fluxo de controle nunca poderá alcançar uma instrução específica em nenhuma circunstância, ele produzirá um aviso CS0162, conforme mostrado no exemplo a seguir:

C#

```
// An over-simplified example of unreachable code.
const int val = 5;
if (val < 4)
{
    System.Console.WriteLine("I'll never write anything."); //CS0162
}
```

Especificação da linguagem C#

Para obter mais informações, consulte a seção Instruções da especificação da linguagem C#.

Consulte também

- Palavras-chave de declaração
- Operadores e expressões C#

Membros aptos para expressão (Guia de Programação em C#)

Artigo • 10/05/2023

As definições de corpo da expressão permitem que você forneça uma implementação de um membro em uma forma bastante concisa e legível. Você pode usar uma definição de corpo da expressão sempre que a lógica para qualquer membro com suporte, como um método ou propriedade, consiste em uma única expressão. Uma definição de corpo da expressão tem a seguinte sintaxe geral:

```
C#

member => expression;
```

em que *expression* é uma expressão válida.

As definições do corpo da expressão podem ser usadas com os seguintes membros de tipo:

- Método
- Propriedade somente leitura
- Propriedade
- Construtor
- Finalizer
- Indexador

Métodos

Um método apto para expressão consiste em uma única expressão que retorna um valor cujo tipo corresponde ao tipo de retorno do método, ou, para métodos que retornam `void`, que executam uma operação. Por exemplo, os tipos que substituem o método ToString normalmente incluem uma única expressão que retorna a representação da cadeia de caracteres do objeto atual.

O exemplo a seguir define uma classe `Person` que substitui o método ToString por uma definição de corpo da expressão. Ele também define um método `DisplayName` que exibe um nome para o console. Observe que a palavra-chave `return` não é usada na definição de corpo da expressão `ToString`.

```
C#
```

```
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();
   }
}
```

Para obter mais informações, consulte Métodos (Guia de Programação em C#).

Propriedades somente leitura

Você pode usar a definição de corpo da expressão para implementar uma propriedade somente leitura. Para isso, use a seguinte sintaxe:

C#

```
PropertyType PropertyName => expression;
```

O exemplo a seguir define uma classe `Location` cuja propriedade somente leitura `Name` é implementada como uma definição de corpo da expressão que retorna o valor do campo `locationName` particular:

C#

```
public class Location
{
   private string locationName;
```

```
    public Location(string name)
    {
        locationName = name;
    }

    public string Name => locationName;
}
```

Para obter mais informações sobre as propriedades, confira Propriedades (Guia de Programação em C#).

Propriedades

Você pode usar as definições de corpo da expressão para implementar a propriedade `get` e os acessadores `set`. O exemplo a seguir demonstra como fazer isso:

```
C#

public class Location
{
    private string locationName;

    public Location(string name) => Name = name;

    public string Name
    {
        get => locationName;
        set => locationName = value;
    }
}
```

Para obter mais informações sobre as propriedades, confira Propriedades (Guia de Programação em C#).

Construtores

Uma definição de corpo da expressão para um construtor normalmente consiste em uma expressão de atribuição simples ou uma chamada de método que manipula os argumentos do construtor ou inicializa o estado da instância.

O exemplo a seguir define uma classe `Location` cujo construtor tem um único parâmetro de cadeia de caracteres chamado *nome*. A definição de corpo da expressão atribui o argumento à propriedade `Name`.

```
C#
```

```
public class Location
{
   private string locationName;

   public Location(string name) => Name = name;

   public string Name
   {
      get => locationName;
      set => locationName = value;
   }
}
```

Para obter mais informações, consulte Construtores (Guia de Programação em C#).

Finalizadores

Uma definição de corpo da expressão para um finalizador normalmente contém instruções de limpeza, como instruções que liberam recursos não gerenciados.

O exemplo a seguir define um finalizador que usa uma definição de corpo da expressão para indicar que o finalizador foi chamado.

C#

```
public class Destroyer
{
   public override string ToString() => GetType().Name;

   ~Destroyer() => Console.WriteLine($"The {ToString()} finalizer is
executing.");
}
```

Para obter mais informações, consulte Finalizadores (Guia de Programação em C#).

Indexadores

Como as propriedades, os acessadores `get` e `set` do indexador consistirão em definições de corpo da expressão se o acessador `get` consistir em uma única expressão que retorna um valor ou o acessador `set` executar uma atribuição simples.

O exemplo a seguir define uma classe chamada `Sports` que inclui uma matriz String interna que contém os nomes de vários esportes. Os acessadores `get` e `set` do indexador são implementados como definições de corpo da expressão.

```C#
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;
   }
}
```

Para obter mais informações, consulte Indexadores (Guia de Programação em C#).

Confira também

- Regras de estilo de código do .NET para membros do corpo da expressão

Comparações de igualdade (Guia de Programação em C#)

19/06/2025

Às vezes, é necessário comparar dois valores para igualdade. Em alguns casos, você está testando a *igualdade de valor*, também conhecida como *equivalência*, o que significa que os valores contidos pelas duas variáveis são iguais. Em outros casos, você precisa determinar se duas variáveis se referem ao mesmo objeto subjacente na memória. Esse tipo de igualdade é chamado de *igualdade de referência* ou *identidade*. Este tópico descreve esses dois tipos de igualdade e fornece links para outros tópicos para obter mais informações.

Igualdade de referência

A igualdade de referência significa que duas referências de objeto se referem ao mesmo objeto subjacente. Isso pode ocorrer por meio de uma atribuição simples, conforme mostrado no exemplo a seguir.

C#

```
using System;
class Test
{
    public int Num { get; set; }
    public string Str { get; set; }

    public static void Main()
    {
        Test a = new Test() { Num = 1, Str = "Hi" };
        Test b = new Test() { Num = 1, Str = "Hi" };

        bool areEqual = System.Object.ReferenceEquals(a, b);
        // False:
        System.Console.WriteLine($"ReferenceEquals(a, b) = {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) = {areEqual}");
    }
}
```

Nesse código, dois objetos são criados, mas após a instrução de atribuição, ambas as referências se referem ao mesmo objeto. Portanto, eles têm igualdade de referência. Use o ReferenceEquals método para determinar se duas referências se referem ao mesmo objeto.

O conceito de igualdade de referência se aplica apenas aos tipos de referência. Objetos de tipo de valor não podem ter igualdade de referência porque quando uma instância de um tipo de valor é atribuída a uma variável, uma cópia do valor é feita. Portanto, você nunca pode ter dois structs não embalados que apontam para o mesmo local na memória. Além disso, se você usar ReferenceEquals para comparar dois tipos de valor, o resultado sempre será `false`, mesmo que os valores contidos nos objetos sejam todos idênticos. Isso ocorre porque cada variável é encaixada em uma instância de objeto separada. Para obter mais informações, consulte Como testar a igualdade de referência (Identidade).

Igualdade de valores

Igualdade de valor significa que dois objetos contêm o mesmo valor ou valores. Para tipos de valor primitivo, como int ou bool, os testes de igualdade de valor são simples. Você pode usar o == operador, conforme mostrado no exemplo a seguir.

C#

```
int a = GetOriginalValue();
int b = GetCurrentValue();

// Test for value equality.
if (b == a)
{
    // The two integers are equal.
}
```

Para a maioria dos outros tipos, o teste de igualdade de valor é mais complexo porque requer que você entenda como o tipo o define. Para classes e structs que têm vários campos ou propriedades, a igualdade de valor geralmente é definida para significar que todos os campos ou propriedades têm o mesmo valor. Por exemplo, dois objetos `Point` podem ser definidos para serem equivalentes se pointA.X for igual a pointB.X e pointA.Y for igual a pointB.Y. Para registros, a igualdade de valor significa que duas variáveis de um tipo de registro são iguais se os tipos corresponderem e todos os valores de propriedade e campo corresponderem.

No entanto, não há nenhum requisito de que a equivalência seja baseada em todos os campos de um tipo. Ele pode ser baseado em um subconjunto. Ao comparar tipos que você não possui, você deve entender especificamente como a equivalência é definida para esse tipo. Para obter mais informações sobre como definir a igualdade de valor em suas próprias classes e structs, consulte Como definir a igualdade de valor para um tipo.

Igualdade de valor para valores de ponto flutuante

Comparações de igualdade de valores de ponto flutuante (duplo e flutuante) são problemáticas devido à imprecisão da aritmética de ponto flutuante em computadores binários. Para obter mais informações, consulte as observações no tópico System.Double.

Tópicos relacionados

Expandir a tabela

Título	Descrição
Como testar a igualdade de referência (Identidade)	Descreve como determinar se duas variáveis têm igualdade de referência.
Como definir a igualdade de valor para um tipo	Descreve como fornecer uma definição personalizada de igualdade de valor para um tipo.
tipos de	Fornece informações sobre o sistema de tipos C# e links para informações adicionais.
Registros	Fornece informações sobre tipos de registro, que testam a igualdade de valor por padrão.

Como definir a igualdade de valor para uma classe ou struct (Guia de programação em C#)

Artigo • 13/03/2024

Os registros implementam automaticamente a igualdade de valor. Considere definir um `record` em vez de um `class` quando seu tipo modelar dados e tiver que implementar a igualdade de valor.

Quando você define uma classe ou struct, decide se faz sentido criar uma definição personalizada de igualdade de valor (ou equivalência) para o tipo. Normalmente, você implementa igualdade de valor quando espera adicionar objetos do tipo a uma coleção, ou quando seu objetivo principal for armazenar um conjunto de campos ou propriedades. Você pode basear sua definição de igualdade de valor em uma comparação de todos os campos e propriedades no tipo ou pode basear a definição em um subconjunto.

Em ambos os casos e em classes e structs, sua implementação deve seguir as cinco garantias de equivalência (para as seguintes regras, suponha que `x`, `y` e `z` não sejam nulos):

1. Propriedade reflexiva: `x.Equals(x)` retorna `true`.
2. Propriedade simétrica: `x.Equals(y)` retorna o mesmo valor que `y.Equals(x)`.
3. Propriedade transitiva: se `(x.Equals(y) && y.Equals(z))` retorna `true`, então `x.Equals(z)` retorna `true`.
4. Invocações sucessivas de `x.Equals(y)` retornam o mesmo valor, contanto que os objetos referenciados por x e y não sejam modificados.
5. Qualquer valor não nulo não é igual a nulo. No entanto, `x.Equals(y)` gera uma exceção quando `x` é nulo. Isso quebra as regras 1 ou 2, a depender do argumento para `Equals`.

Qualquer struct que você define já tem uma implementação padrão de igualdade de valor que ele herda da substituição System.ValueType do método Object.Equals(Object). Essa implementação usa a reflexão para examinar todos os campos e propriedades no tipo. Embora essa implementação produza resultados corretos, ela é relativamente lenta

em comparação com uma implementação personalizada escrita especificamente para o tipo.

Os detalhes de implementação para a igualdade de valor são diferentes para classes e struct. No entanto, as classes e structs exigem as mesmas etapas básicas para implementar a igualdade:

1. Substitua o método virtualObject.Equals(Object). Na maioria dos casos, sua implementação de `bool Equals( object obj )` deve apenas chamar o método `Equals` específico do tipo que é a implementação da interface System.IEquatable<T>. (Consulte a etapa 2.)

2. Implemente a interface System.IEquatable<T> fornecendo um método `Equals` específico do tipo. Isso é o local em que a comparação de equivalência de fato é realizada. Por exemplo, você pode decidir definir a igualdade comparando apenas um ou dois campos em seu tipo. Não lance exceções a partir de `Equals`. Para classes relacionadas por herança:

   - esse método deve examinar somente os campos que são declarados na classe. Ele deve chamar `base.Equals` para examinar os campos que estão na classe base. (Não chame `base.Equals` se o tipo herdar diretamente de Object, pois a implementação Object de Object.Equals(Object) executa uma verificação de igualdade de referência.)

   - Duas variáveis devem ser consideradas iguais somente se os tipos de tempo de execução das variáveis que estão sendo comparadas forem os mesmos. Além disso, verifique se a implementação `IEquatable` do método `Equals` para o tipo de tempo de execução será usada se os tipos de tempo de execução e tempo de compilação de uma variável forem diferentes. Uma estratégia para garantir que os tipos de tempo de execução sejam sempre comparados corretamente é implementar `IEquatable` somente em classes `sealed`. Para obter mais informações, consulte o exemplo de classe mais adiante neste artigo.

3. Opcional, mas recomendado: sobrecarregue os operadores `==` e `!=`.

4. Substitua Object.GetHashCode para que os dois objetos que têm a igualdade de valor produzam o mesmo código hash.

5. Opcional: para dar suporte às definições para “maior que” ou “menor que”, implemente a interface IComparable<T> para seu tipo e também sobrecarregue os operadores `<=` e `>=`.

ⓘ **Observação**

Use registros para obter a semântica de igualdade de valor sem nenhum código clichê desnecessário.

Exemplo de classe

O exemplo a seguir mostra como implementar a igualdade de valor em uma classe (tipo de referência).

C#

```
namespace ValueEqualityClass;

class TwoDPoint : IEquatable<TwoDPoint>
{
    public int X { get; private set; }
    public int Y { get; private set; }

    public TwoDPoint(int x, int y)
    {
        if (x is (< 1 or > 2000) || y is (< 1 or > 2000))
        {
            throw new ArgumentException("Point must be in range 1 - 2000");
        }
        this.X = x;
        this.Y = y;
    }

    public override bool Equals(object obj) => this.Equals(obj as
TwoDPoint);

    public bool Equals(TwoDPoint p)
    {
        if (p is null)
        {
            return false;
        }

        // Optimization for a common success case.
        if (Object.ReferenceEquals(this, p))
        {
            return true;
        }

        // If run-time types are not exactly the same, return false.
        if (this.GetType() != p.GetType())
        {
            return false;
```

```
        }

        // Return true if the fields match.
        // Note that the base class is not invoked because it is
        // System.Object, which defines Equals as reference equality.
        return (X == p.X) && (Y == p.Y);
    }

    public override int GetHashCode() => (X, Y).GetHashCode();

    public static bool operator ==(TwoDPoint lhs, TwoDPoint rhs)
    {
        if (lhs is null)
        {
            if (rhs is null)
            {
                return true;
            }

            // Only the left side is null.
            return false;
        }
        // Equals handles case of null on right side.
        return lhs.Equals(rhs);
    }

    public static bool operator !=(TwoDPoint lhs, TwoDPoint rhs) => !(lhs ==
rhs);
}

// For the sake of simplicity, assume a ThreeDPoint IS a TwoDPoint.
class ThreeDPoint : TwoDPoint, IEquatable<ThreeDPoint>
{
    public int Z { get; private set; }

    public ThreeDPoint(int x, int y, int z)
        : base(x, y)
    {
        if ((z < 1) || (z > 2000))
        {
            throw new ArgumentException("Point must be in range 1 - 2000");
        }
        this.Z = z;
    }

    public override bool Equals(object obj) => this.Equals(obj as
ThreeDPoint);

    public bool Equals(ThreeDPoint p)
    {
        if (p is null)
        {
            return false;
        }
```

```
        // Optimization for a common success case.
        if (Object.ReferenceEquals(this, p))
        {
            return true;
        }

        // Check properties that this class declares.
        if (Z == p.Z)
        {
            // Let base class check its own fields
            // and do the run-time type comparison.
            return base.Equals((TwoDPoint)p);
        }
        else
        {
            return false;
        }
    }

    public override int GetHashCode() => (X, Y, Z).GetHashCode();

    public static bool operator ==(ThreeDPoint lhs, ThreeDPoint rhs)
    {
        if (lhs is null)
        {
            if (rhs is null)
            {
                // null == null = true.
                return true;
            }

            // Only the left side is null.
            return false;
        }
        // Equals handles the case of null on right side.
        return lhs.Equals(rhs);
    }

    public static bool operator !=(ThreeDPoint lhs, ThreeDPoint rhs) => !
(lhs == rhs);
}

class Program
{
    static void Main(string[] args)
    {
        ThreeDPoint pointA = new ThreeDPoint(3, 4, 5);
        ThreeDPoint pointB = new ThreeDPoint(3, 4, 5);
        ThreeDPoint pointC = null;
        int i = 5;

        Console.WriteLine("pointA.Equals(pointB) = {0}",
pointA.Equals(pointB));
        Console.WriteLine("pointA == pointB = {0}", pointA == pointB);
        Console.WriteLine("null comparison = {0}", pointA.Equals(pointC));
```

```
        Console.WriteLine("Compare to some other type = {0}",
pointA.Equals(i));

        TwoDPoint pointD = null;
        TwoDPoint pointE = null;

        Console.WriteLine("Two null TwoDPoints are equal: {0}", pointD ==
pointE);

        pointE = new TwoDPoint(3, 4);
        Console.WriteLine("(pointE == pointA) = {0}", pointE == pointA);
        Console.WriteLine("(pointA == pointE) = {0}", pointA == pointE);
        Console.WriteLine("(pointA != pointE) = {0}", pointA != pointE);

        System.Collections.ArrayList list = new
System.Collections.ArrayList();
        list.Add(new ThreeDPoint(3, 4, 5));
        Console.WriteLine("pointE.Equals(list[0]): {0}",
pointE.Equals(list[0]));

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}

/* Output:
    pointA.Equals(pointB) = True
    pointA == pointB = True
    null comparison = False
    Compare to some other type = False
    Two null TwoDPoints are equal: True
    (pointE == pointA) = False
    (pointA == pointE) = False
    (pointA != pointE) = True
    pointE.Equals(list[0]): False
*/
```

Em classes (tipo de referência), a implementação padrão de ambos os métodos Object.Equals(Object) executa uma comparação de igualdade de referência, não uma verificação de igualdade de valor. Quando um implementador substitui o método virtual, o objetivo é fornecer semântica de igualdade de valor.

Os operadores `==` e `!=` podem ser usados com classes, mesmo se a classe não sobrecarregá-los. No entanto, o comportamento padrão é executar uma verificação de igualdade de referência. Em uma classe, se você sobrecarregar o método `Equals`, você deverá sobrecarregar os operadores `==` e `!=`, mas isso não é necessário.

> ⓘ **Importante**

O código de exemplo anterior pode não lidar com todos os cenários de herança da maneira esperada. Considere o seguinte código:

C#

```
TwoDPoint p1 = new ThreeDPoint(1, 2, 3);
TwoDPoint p2 = new ThreeDPoint(1, 2, 4);
Console.WriteLine(p1.Equals(p2)); // output: True
```

Esse código relata que `p1` é igual a `p2` apesar da diferença de valores `z`. A diferença é ignorada porque o compilador escolhe a implementação `TwoDPoint` de `IEquatable` com base no tipo de tempo de compilação.

A igualdade de valores interna de tipos `record` lida com cenários como este corretamente. Se `TwoDPoint` e `ThreeDPoint` fossem tipos `record`, o resultado de `p1.Equals(p2)` seria `False`. Para obter mais informações, consulte [Igualdade nas hierarquias de herança de tipo record](#).

Exemplo de struct

O exemplo a seguir mostra como implementar a igualdade de valor em um struct (tipo de valor):

C#

```
namespace ValueEqualityStruct
{
    struct TwoDPoint : IEquatable<TwoDPoint>
    {
        public int X { get; private set; }
        public int Y { get; private set; }

        public TwoDPoint(int x, int y)
            : this()
        {
            if (x is (< 1 or > 2000) || y is (< 1 or > 2000))
            {
                throw new ArgumentException("Point must be in range 1 -
2000");
            }
            X = x;
            Y = y;
        }

        public override bool Equals(object? obj) => obj is TwoDPoint other
&& this.Equals(other);
```

```
        public bool Equals(TwoDPoint p) => X == p.X && Y == p.Y;

        public override int GetHashCode() => (X, Y).GetHashCode();

        public static bool operator ==(TwoDPoint lhs, TwoDPoint rhs) =>
lhs.Equals(rhs);

        public static bool operator !=(TwoDPoint lhs, TwoDPoint rhs) => !
(lhs == rhs);
    }

    class Program
    {
        static void Main(string[] args)
        {
            TwoDPoint pointA = new TwoDPoint(3, 4);
            TwoDPoint pointB = new TwoDPoint(3, 4);
            int i = 5;

            // True:
            Console.WriteLine("pointA.Equals(pointB) = {0}",
pointA.Equals(pointB));
            // True:
            Console.WriteLine("pointA == pointB = {0}", pointA == pointB);
            // True:
            Console.WriteLine("object.Equals(pointA, pointB) = {0}",
object.Equals(pointA, pointB));
            // False:
            Console.WriteLine("pointA.Equals(null) = {0}",
pointA.Equals(null));
            // False:
            Console.WriteLine("(pointA == null) = {0}", pointA == null);
            // True:
            Console.WriteLine("(pointA != null) = {0}", pointA != null);
            // False:
            Console.WriteLine("pointA.Equals(i) = {0}", pointA.Equals(i));
            // CS0019:
            // Console.WriteLine("pointA == i = {0}", pointA == i);

            // Compare unboxed to boxed.
            System.Collections.ArrayList list = new
System.Collections.ArrayList();
            list.Add(new TwoDPoint(3, 4));
            // True:
            Console.WriteLine("pointA.Equals(list[0]): {0}",
pointA.Equals(list[0]));

            // Compare nullable to nullable and to non-nullable.
            TwoDPoint? pointC = null;
            TwoDPoint? pointD = null;
            // False:
            Console.WriteLine("pointA == (pointC = null) = {0}", pointA ==
pointC);
            // True:
```

```
            Console.WriteLine("pointC == pointD = {0}", pointC == pointD);

            TwoDPoint temp = new TwoDPoint(3, 4);
            pointC = temp;
            // True:
            Console.WriteLine("pointA == (pointC = 3,4) = {0}", pointA ==
pointC);

            pointD = temp;
            // True:
            Console.WriteLine("pointD == (pointC = 3,4) = {0}", pointD ==
pointC);

            Console.WriteLine("Press any key to exit.");
            Console.ReadKey();
        }
    }

    /* Output:
        pointA.Equals(pointB) = True
        pointA == pointB = True
        Object.Equals(pointA, pointB) = True
        pointA.Equals(null) = False
        (pointA == null) = False
        (pointA != null) = True
        pointA.Equals(i) = False
        pointE.Equals(list[0]): True
        pointA == (pointC = null) = False
        pointC == pointD = True
        pointA == (pointC = 3,4) = True
        pointD == (pointC = 3,4) = True
    */
}
```

Para estruturas, a implementação padrão de Object.Equals(Object) (que é a versão substituída em System.ValueType) executa uma verificação de igualdade de valor por meio de reflexão para comparar os valores de cada campo no tipo. Quando um implementador substitui o método virtual `Equals` em um struct, a finalidade é fornecer uma maneira mais eficiente de executar a verificação de igualdade de valor e, opcionalmente, basear a comparação em algum subconjunto dos campos ou propriedades do struct.

Os operadores `==` e `!=` não podem operar em um struct a menos que o struct os sobrecarregue explicitamente.

Confira também

- Comparações de igualdade

Colaborar conosco no GitHub

A fonte deste conteúdo pode ser encontrada no GitHub, onde você também pode criar e revisar problemas e solicitações de pull. Para obter mais informações, confira o nosso guia para colaboradores.

Comentários do .NET

O .NET é um projeto código aberto. Selecione um link para fornecer comentários:

- Abrir um problema de documentação
- Fornecer comentários sobre o produto

Como testar a igualdade de referência (Identidade) (Guia de Programação em C#)

Artigo • 03/02/2025

Você não precisa implementar nenhuma lógica personalizada para dar suporte a comparações de igualdade de referência em seus tipos. Essa funcionalidade é fornecida para todos os tipos pelo método de Object.ReferenceEquals estático.

O exemplo a seguir mostra como determinar se duas variáveis têm de igualdade de referência, o que significa que elas se referem ao mesmo objeto na memória.

O exemplo também mostra por que Object.ReferenceEquals sempre retorna `false` para tipos de valor. Isso ocorre devido ao **boxing**, que cria instâncias de objetos separadas para cada argumento de tipo de valor. Além disso, você não deve usar ReferenceEquals para determinar a igualdade de cadeia de caracteres.

Exemplo

C#

```
using System.Text;

namespace TestReferenceEquality
{
    struct TestStruct
    {
        public int Num { get; private set; }
        public string Name { get; private set; }

        public TestStruct(int i, string s) : this()
        {
            Num = i;
            Name = s;
        }
    }

    class TestClass
    {
        public int Num { get; set; }
        public string? Name { get; set; }
    }

    class Program
    {
```

```
        static void Main()
        {
            // Demonstrate reference equality with reference types.
            #region ReferenceTypes

            // Create two reference type instances that have identical
values.
            TestClass tcA = new TestClass() { Num = 1, Name = "New
TestClass" };
            TestClass tcB = new TestClass() { Num = 1, Name = "New
TestClass" };

            Console.WriteLine("ReferenceEquals(tcA, tcB) = {0}",
                                Object.ReferenceEquals(tcA, tcB)); // false

            // After assignment, tcB and tcA refer to the same object.
            // They now have reference equality.
            tcB = tcA;
            Console.WriteLine("After assignment: ReferenceEquals(tcA, tcB) =
{0}",
                                Object.ReferenceEquals(tcA, tcB)); // true

            // Changes made to tcA are reflected in tcB. Therefore, objects
            // that have reference equality also have value equality.
            tcA.Num = 42;
            tcA.Name = "TestClass 42";
            Console.WriteLine("tcB.Name = {0} tcB.Num: {1}", tcB.Name,
tcB.Num);
            #endregion

            // Demonstrate that two value type instances never have
reference equality.
            #region ValueTypes

            TestStruct tsC = new TestStruct( 1, "TestStruct 1");

            // Value types are boxed into separate objects when passed to
ReferenceEquals.
            // Even if the same variable is used twice, boxing ensures they
are different instances.
            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
*/
```

A implementação de `Equals` na classe base universal System.Object também executa uma verificação de igualdade de referência, mas é melhor não usá-la porque, se uma classe substituir o método, os resultados podem não ser o esperado. O mesmo vale para os operadores `==` e `!=`. Quando eles estão operando em tipos de referência, o comportamento padrão de `==` e `!=` é executar uma verificação de igualdade de referência. No entanto, classes derivadas podem sobrecarregar o operador para

executar uma verificação de igualdade de valor. Para minimizar o potencial de erro, é melhor sempre usar ReferenceEquals quando você precisa determinar se dois objetos têm igualdade de referência.

Cadeias de caracteres constantes dentro do mesmo assembly sempre são internalizadas pelo runtime. Ou seja, apenas uma instância de cada cadeia de caracteres literal única é mantida. No entanto, o runtime não garante que as cadeias de caracteres criadas em tempo de execução sejam internadas, nem garante que duas cadeias de caracteres constantes iguais em assemblies diferentes sejam internadas.

> ⓘ **Observação**
>
> `ReferenceEquals` retorna `false` para tipos de valores devidos ao **boxing**, já que cada argumento é reservado individualmente em um objeto separado.

Consulte também

- Comparações de Igualdade

Coerções e conversões de tipo (Guia de Programação em C#)

19/06/2025

Como o C# é tipado estaticamente no momento da compilação, depois que uma variável é declarada, ela não pode ser declarada novamente ou atribuída a um valor de outro tipo, a menos que esse tipo seja implicitamente conversível para o tipo da variável. Por exemplo, `string` não pode ser convertido implicitamente para `int`. Portanto, depois de declarar `i` como um `int`, não é possível atribuir a cadeia de caracteres "Olá" a ela, como mostra o código a seguir:

C#

```
int i;

// error CS0029: can't implicitly convert type 'string' to 'int'
i = "Hello";
```

No entanto, às vezes, talvez seja necessário copiar um valor para um parâmetro de variável ou método de outro tipo. Por exemplo, você pode ter uma variável de inteiro que precisa passar para um método cujo parâmetro é digitado como `double`. Ou talvez seja necessário atribuir uma variável de classe a uma variável de um tipo de interface. Esses tipos de operações são *chamados de conversões de tipo*. No C#, você pode executar os seguintes tipos de conversões:

- **Conversões implícitas**: nenhuma sintaxe especial é necessária porque a conversão sempre é bem-sucedida e nenhum dado é perdido. Os exemplos incluem conversões de tipos integrais menores para maiores, conversões de classes derivadas para classes base e conversões de intervalo.

- **Conversões explícitas (casts)**: as conversões explícitas exigem uma expressão cast. O casting é necessário quando as informações podem ser perdidas no processo de conversão ou quando a conversão pode não ter êxito por outros motivos. Exemplos típicos incluem conversão numérica em um tipo que tem menos precisão ou um intervalo menor e conversão de uma instância de classe base em uma classe derivada.

- **Conversões definidas pelo usuário**: as conversões definidas pelo usuário usam métodos especiais que você pode definir para habilitar conversões explícitas e implícitas entre tipos personalizados que não têm uma relação de classe base derivada de classe. Para saber mais, confira Operadores de conversão definidos pelo usuário.

- **Conversões com classes auxiliares**: para converter entre tipos não compatíveis, como inteiros e System.DateTime objetos, ou cadeias de caracteres hexadecimal e matrizes de

bytes, você pode usar a System.BitConverter classe, a System.Convert classe e os `Parse` métodos dos tipos numéricos internos, como Int32.Parse. Para obter mais informações, consulte os seguintes artigos:

- Como converter uma matriz de bytes em um int
- Como converter uma cadeia de caracteres em um número
- Como converter entre cadeias de caracteres hexadecimal e tipos numéricos

Conversões implícitas

Para tipos numéricos internos, uma conversão implícita pode ser feita quando o valor a ser armazenado pode caber na variável sem ser truncado ou arredondado. Para tipos integrais, essa restrição significa que o intervalo do tipo de origem é um subconjunto adequado do intervalo para o tipo de destino. Por exemplo, uma variável do tipo long (inteiro de 64 bits) pode armazenar qualquer valor que um int (inteiro de 32 bits) possa armazenar. No exemplo a seguir, o compilador converte implicitamente o valor da `num` direita em um tipo `long` antes de atribuí-lo a `bigNum`.

C#

```
// Implicit conversion. A long can
// hold any value an int can hold, and more!
int num = 2147483647;
long bigNum = num;
```

Para obter uma lista completa de todas as conversões numéricas implícitas, consulte a seção conversões numéricas implícitas do artigo conversões numéricas internas .

Para tipos de referência, sempre existe uma conversão implícita de uma classe para qualquer uma de suas classes ou interfaces base diretas ou indiretas. Nenhuma sintaxe especial é necessária porque uma classe derivada sempre contém todos os membros de uma classe base.

C#

```
Derived d = new Derived();

// Always OK.
Base b = d;
```

Conversões explícitas

No entanto, se uma conversão não puder ser realizada sem o risco de perda de informações, o compilador exigirá que você execute uma conversão explícita, que é chamada de *cast*. Um cast é uma forma de fazer a conversão explicitamente. Isso indica que você está ciente de que a perda de dados pode ocorrer ou a conversão pode falhar em tempo de execução. Para executar uma conversão, especifique o tipo de destino entre parênteses antes da expressão que você deseja converter. O programa a seguir converte um double em um int. O programa não compila sem a conversão.

C#

```
double x = 1234.7;
int a;
// Cast double to int.
a = (int)x;
Console.WriteLine(a);
// Output: 1234
```

Para uma lista completa de conversões numéricas explícitas com suporte, consulte a seção Conversões numéricas explícitas do artigo Conversões numéricas nativas.

Para tipos de referência, uma conversão explícita será necessária se você precisar converter de um tipo base para um tipo derivado:

Uma operação de conversão entre tipos de referência não altera o tipo de execução em tempo real do objeto subjacente; ela apenas altera o tipo do valor que está sendo usado como referência para esse objeto. Para obter mais informações, consulte Polymorphism.

Exceções de conversão de tipo durante a execução

Em algumas conversões de tipo de referência, o compilador não pode determinar se uma conversão é válida. É possível que uma operação de conversão que é compilada corretamente falhe em tempo de execução. Conforme mostrado no exemplo a seguir, um tipo de conversão que falha em tempo de execução faz com que uma InvalidCastException seja lançada.

C#

```
Animal a = new Mammal();
Reptile r = (Reptile)a; // InvalidCastException at run time
```

Converter explicitamente o argumento `a` em um `Reptile` faz uma suposição perigosa. É mais seguro não fazer suposições, mas sim verificar o tipo. O C# fornece o is operador para permitir que você teste a compatibilidade antes de realmente executar uma conversão. Para saber mais,

confira Como converter com segurança usando a correspondência de padrões e os operadores "is" e "as".

Especificação da linguagem C#

Para obter mais informações, consulte a seção Conversões da especificação da linguagem C#.

Consulte também

- tipos de
- Expressão de conversão
- Operadores de conversão definidos pelo usuário
- Conversão generalizada de tipo
- Como converter uma cadeia de caracteres em um número

Conversões boxing e unboxing (Guia de Programação em C#)

19/06/2025

Conversão boxing é o processo de conversão de um tipo de valor para o tipo `object` ou para qualquer tipo de interface implementada por esse tipo de valor. Quando o CLR (Common Language Runtime) realiza a conversão de um tipo de valor, ele encapsula o valor dentro de uma instância System.Object e a armazena no heap gerenciado. A conversão unboxing extrai o tipo de valor do objeto. A conversão boxing é implícita, a conversão unboxing é explícita. O conceito de conversões boxing e unboxing serve como base para a exibição unificada de C# do sistema de tipos em que um valor de qualquer tipo pode ser tratado como um objeto.

No exemplo a seguir, a variável `i` de inteiro é *boxada* e atribuída ao objeto `o`.

C#

```
int i = 123;
// The following line boxes i.
object o = i;
```

Em seguida, o objeto `o` pode ser desencaixado e atribuído à variável inteira `i`.

C#

```
o = 123;
i = (int)o;  // unboxing
```

Os exemplos a seguir ilustram como o boxe é usado em C#.

C#

```
// String.Concat example.
// String.Concat has many versions. Rest the mouse pointer on
// Concat in the following statement to verify that the version
// that is used here takes three object arguments. Both 42 and
// true must be boxed.
Console.WriteLine(String.Concat("Answer", 42, true));

// List example.
// Create a list of objects to hold a heterogeneous collection
// of elements.
List<object> mixedList = new List<object>();

// Add a string element to the list.
mixedList.Add("First Group:");
```

```
// Add some integers to the list.
for (int j = 1; j < 5; j++)
{
    // Rest the mouse pointer over j to verify that you are adding
    // an int to a list of objects. Each element j is boxed when
    // you add j to mixedList.
    mixedList.Add(j);
}

// Add another string and more integers.
mixedList.Add("Second Group:");
for (int j = 5; j < 10; j++)
{
    mixedList.Add(j);
}

// Display the elements in the list. Declare the loop variable by
// using var, so that the compiler assigns its type.
foreach (var item in mixedList)
{
    // Rest the mouse pointer over item to verify that the elements
    // of mixedList are objects.
    Console.WriteLine(item);
}

// The following loop sums the squares of the first group of boxed
// integers in mixedList. The list elements are objects, and cannot
// be multiplied or added to the sum until they are unboxed. The
// unboxing must be done explicitly.
var sum = 0;
for (var j = 1; j < 5; j++)
{
    // The following statement causes a compiler error: Operator
    // '*' cannot be applied to operands of type 'object' and
    // 'object'.
    //sum += mixedList[j] * mixedList[j];

    // After the list elements are unboxed, the computation does
    // not cause a compiler error.
    sum += (int)mixedList[j] * (int)mixedList[j];
}

// The sum displayed is 30, the sum of 1 + 4 + 9 + 16.
Console.WriteLine("Sum: " + sum);

// Output:
// Answer42True
// First Group:
// 1
// 2
// 3
// 4
// Second Group:
// 5
```

```
// 6
// 7
// 8
// 9
// Sum: 30
```

Desempenho

Em relação às atribuições simples, as conversões boxing e unboxing são processos computacionalmente dispendiosos. Quando um tipo de valor é submetido à conversão boxing, um novo objeto deve ser alocado e construído. A um grau menor, a conversão necessária para a conversão unboxing também é computacionalmente dispendiosa. Para obter mais informações, consulte Desempenho.

Boxe

A conversão boxing é usada para armazenar tipos de valor no heap coletado como lixo. A conversão boxing é uma conversão implícita de um tipo de valor para o tipo `object` ou para qualquer tipo de interface implementada por esse tipo de valor. A conversão boxing de um tipo de valor aloca uma instância de objeto no heap e copia o valor no novo objeto.

Considere a seguinte declaração de uma variável de tipo de valor:

C#

```
int i = 123;
```

A instrução a seguir aplica implicitamente a operação de conversão boxing na variável `i`:

C#

```
// Boxing copies the value of i into object o.
object o = i;
```

O resultado dessa instrução é a criação de uma referência de objeto `o`, na pilha, que faz referência a um valor do tipo `int`, no heap. Esse valor é uma cópia do valor de tipo de valor atribuído à variável `i`. A diferença entre as duas variáveis, `i` e `o`, é ilustrada na figura de conversão boxing a seguir:

Também é possível executar o boxe explicitamente como no exemplo a seguir, mas o boxe explícito nunca é necessário:

C#

```
int i = 123;
object o = (object)i;  // explicit boxing
```

Exemplo

Este exemplo converte uma variável `i` de inteiro em um objeto `o` usando boxing. Em seguida, o valor armazenado na variável `i` é alterado de `123` para `456`. O exemplo mostra que o tipo de valor original e o objeto boxed usam locais de memória separados e, portanto, podem armazenar valores diferentes.

C#

```
        // Create an int variable
        int i = 123;

        // Box the value type into an object reference
        object o = i;  // boxing

        // Display the initial values
        Console.WriteLine($"Value of i: {i}");
        Console.WriteLine($"Value of boxed object o: {o}");

        // Modify the original value type
        i = 456;

        // Display the values after modification
        Console.WriteLine("\nAfter changing i to 456:");
        Console.WriteLine($"Value of i: {i}");
        Console.WriteLine($"Value of boxed object o: {o}");

        // Output:
        // Value of i: 123
        // Value of boxed object o: 123

        // After changing i to 456:
```

```
        // Value of i: 456
        // Value of boxed object o: 123
```

Conversão unboxing

A conversão unboxing é uma conversão explícita do tipo `object` para um tipo de valor ou de um tipo de interface para um tipo de valor que implementa a interface. Uma operação de unboxing consiste em:

- Verificar a instância do objeto para garantir que ele é um valor da conversão boxing de um determinado tipo de valor.
- Copiando o valor da instância para a variável de tipo valor.

As instruções a seguir demonstram operações conversão boxing e unboxing:

C#

```
int i = 123;      // a value type
object o = i;     // boxing
int j = (int)o;   // unboxing
```

A figura a seguir demonstra o resultado das instruções anteriores:

Para a conversão unboxing de tipos de valor ter êxito em tempo de execução, o item sendo submetido à conversão unboxing deve ser uma referência para um objeto que foi criado anteriormente ao realizar a conversão boxing de uma instância desse tipo de valor. Tentar realizar a conversão unboxing de `null` causa uma NullReferenceException. Tentar realizar a conversão unboxing de uma referência para um tipo de valor incompatível causa uma InvalidCastException.

Exemplo

O exemplo a seguir demonstra um caso de conversão unboxing inválida e o `InvalidCastException` resultante. Usando `try` e `catch`, uma mensagem de erro é exibida quando o erro ocorre.

```C#
class TestUnboxing
{
    static void Main()
    {
        int i = 123;
        object o = i;  // implicit boxing

        try
        {
            int j = (short)o;  // attempt to unbox

            System.Console.WriteLine("Unboxing OK.");
        }
        catch (System.InvalidCastException e)
        {
            System.Console.WriteLine($"{e.Message} Error: Incorrect unboxing.");
        }
    }
}
```

Este programa gera:

`Specified cast is not valid. Error: Incorrect unboxing.`

Se você alterar a instrução:

```C#
int j = (short)o;
```

para:

```C#
int j = (int)o;
```

a conversão será executada e você obterá a saída:

`Unboxing OK.`

Especificação da linguagem C#

Para obter mais informações, consulte a Especificação da Linguagem C# . A especificação de idioma é a fonte definitiva para a sintaxe e o uso de C#.

Consulte também

- Tipos de referência
- tipos de valor

Como converter uma matriz de bytes em um int (Guia de Programação em C#)

Artigo • 07/04/2023

Este exemplo mostra como usar a classe BitConverter para converter uma matriz de bytes em um int e de volta em uma matriz de bytes. Talvez você precise converter bytes em um tipo de dados interno depois de ler bytes da rede, por exemplo. Além do método ToInt32(Byte[], Int32) no exemplo, a tabela a seguir lista métodos na classe BitConverter que convertem bytes (de uma matriz de bytes) em outros tipos internos.

Tipo retornado	Método
`bool`	ToBoolean(Byte[], Int32)
`char`	ToChar(Byte[], Int32)
`double`	ToDouble(Byte[], Int32)
`short`	ToInt16(Byte[], Int32)
`int`	ToInt32(Byte[], Int32)
`long`	ToInt64(Byte[], Int32)
`float`	ToSingle(Byte[], Int32)
`ushort`	ToUInt16(Byte[], Int32)
`uint`	ToUInt32(Byte[], Int32)
`ulong`	ToUInt64(Byte[], Int32)

Exemplos

Este exemplo inicializa uma matriz de bytes, reverte a matriz se a arquitetura do computador for little-endian (ou seja, se o byte menos significativo for armazenado primeiro) e, em seguida, chama o método ToInt32(Byte[], Int32) para converter quatro bytes da matriz em um `int`. O segundo argumento para ToInt32(Byte[], Int32) especifica o índice de início da matriz de bytes.

Observação

A saída pode ser diferente dependendo da extremidade (ordenação dos bytes) da arquitetura do computador.

C#

```
byte[] bytes = { 0, 0, 0, 25 };

// If the system architecture is little-endian (that is, little end first),
// reverse the byte array.
if (BitConverter.IsLittleEndian)
    Array.Reverse(bytes);

int i = BitConverter.ToInt32(bytes, 0);
Console.WriteLine("int: {0}", i);
// Output: int: 25
```

Neste exemplo, o método GetBytes(Int32) da classe BitConverter é chamado para converter um `int` em uma matriz de bytes.

> **Observação**
>
> A saída pode ser diferente dependendo da extremidade (ordenação dos bytes) da arquitetura do computador.

C#

```
byte[] bytes = BitConverter.GetBytes(201805978);
Console.WriteLine("byte array: " + BitConverter.ToString(bytes));
// Output: byte array: 9A-50-07-0C
```

Confira também

- BitConverter
- IsLittleEndian
- Types

Como converter uma cadeia de caracteres em um número (Guia de Programação em C#)

Artigo • 07/11/2024

Converta um `string` em um número chamando o método `Parse` ou `TryParse` encontrado em tipos numéricos (`int`, `long`, `double` e assim por diante) ou usando métodos na classe System.Convert.

Será um pouco mais eficiente e simples chamar um método `TryParse` (por exemplo, int.TryParse("11", out number)) ou o método `Parse` (por exemplo, var number = int.Parse("11")). Usar um método Convert é mais útil para objetos gerais que implementam IConvertible.

É possível usar métodos `Parse` ou `TryParse` no tipo numérico que se espera que a cadeia de caracteres contenha, como o tipo System.Int32. O método Convert.ToInt32 usa Parse internamente. O método `Parse` retorna o número convertido; o método `TryParse` retorna um valor booliano que indica se a conversão foi bem-sucedida e retorna o número convertido em um parâmetro `out`. Se a cadeia de caracteres não estiver em um formato válido, `Parse` lançará uma exceção, mas `TryParse` retornará `false`. Ao chamar um método `Parse`, você sempre deve usar o tratamento de exceções para capturar um FormatException quando a operação de análise falhar.

Chamar métodos Parse ou TryParse

Os métodos `Parse` e `TryParse` ignoram o espaço em branco no início e no final da cadeia de caracteres; porém, todos os outros caracteres devem formar o tipo numérico correto (`int`, `long`, `ulong`, `float`, `decimal` e assim por diante). Qualquer espaço em branco na cadeia de caracteres que forma o número causa um erro. Por exemplo, é possível usar `decimal.TryParse` para analisar "10", "10,3" ou " 10 ", mas não é possível usar esse método para analisar 10 de "10X", "1 0" (observe o espaço inserido), "10 .3" (observe o espaço inserido), "10e1" (`float.TryParse` funcionará neste caso) e assim por diante. Uma cadeia de caracteres cujo valor seja `null` ou String.Empty falhará ao ser analisada com êxito. Você pode verificar uma cadeia de caracteres nula ou vazia antes de tentar analisá-la chamando o método String.IsNullOrEmpty.

O exemplo a seguir demonstra chamadas com e sem êxito para `Parse` e `TryParse`.

C#

```
using System;

public static class StringConversion
{
    public static void Main()
    {
        string input = String.Empty;
        try
        {
            int result = Int32.Parse(input);
            Console.WriteLine(result);
        }
        catch (FormatException)
        {
            Console.WriteLine($"Unable to parse '{input}'");
        }
        // Output: Unable to parse ''

        try
        {
            int numVal = Int32.Parse("-105");
            Console.WriteLine(numVal);
        }
        catch (FormatException e)
        {
            Console.WriteLine(e.Message);
        }
        // Output: -105

        if (Int32.TryParse("-105", out int j))
        {
            Console.WriteLine(j);
        }
        else
        {
            Console.WriteLine("String could not be parsed.");
        }
        // Output: -105

        try
        {
            int m = Int32.Parse("abc");
        }
        catch (FormatException e)
        {
            Console.WriteLine(e.Message);
        }
        // Output: Input string was not in a correct format.

        const string inputString = "abc";
        if (Int32.TryParse(inputString, out int numValue))
        {
```

```
            Console.WriteLine(numValue);
        }
        else
        {
            Console.WriteLine($"Int32.TryParse could not parse
'{inputString}' to an int.");
        }
        // Output: Int32.TryParse could not parse 'abc' to an int.
    }
}
```

O exemplo a seguir ilustra uma abordagem para analisar uma cadeia de caracteres que deve incluir caracteres numéricos à esquerda (incluindo caracteres hexadecimais) e caracteres não numéricos à direita. Ele atribui caracteres válidos do início de uma cadeia de caracteres até uma nova cadeia de caracteres antes de chamar o método TryParse. Como as cadeias de caracteres a ser analisadas contêm poucos caracteres, o exemplo chama o método String.Concat para atribuir os caracteres válidos a uma nova cadeia de caracteres. Para cadeias de caracteres maiores, pode ser usada a classe StringBuilder.

C#

```
using System;

public static class StringConversion
{
    public static void Main()
    {
        var str = "  10FFxxx";
        string numericString = string.Empty;
        foreach (var c in str)
        {
            // Check for numeric characters (hex in this case) or leading or
trailing spaces.
            if ((c >= '0' && c <= '9') || (char.ToUpperInvariant(c) >= 'A'
&& char.ToUpperInvariant(c) <= 'F') || c == ' ')
            {
                numericString = string.Concat(numericString, c.ToString());
            }
            else
            {
                break;
            }
        }

        if (int.TryParse(numericString,
System.Globalization.NumberStyles.HexNumber, null, out int i))
        {
            Console.WriteLine($"'{str}' --> '{numericString}' --> {i}");
        }
        // Output: '  10FFxxx' --> '  10FF' --> 4351
```

```
        str = "   -10FFXXX";
        numericString = "";
        foreach (char c in str)
        {
            // Check for numeric characters (0-9), a negative sign, or
leading or trailing spaces.
            if ((c >= '0' && c <= '9') || c == ' ' || c == '-')
            {
                numericString = string.Concat(numericString, c);
            }
            else
            {
                break;
            }
        }

        if (int.TryParse(numericString, out int j))
        {
            Console.WriteLine($"'{str}' --> '{numericString}' --> {j}");
        }
        // Output: '   -10FFXXX' --> '   -10' --> -10
    }
}
```

Chamar métodos Convert

A tabela a seguir lista alguns dos métodos da classe Convert que podem ser usados para converter uma cadeia de caracteres em um número.

Expandir a tabela

Tipo numérico	Método
`decimal`	ToDecimal(String)
`float`	ToSingle(String)
`double`	ToDouble(String)
`short`	ToInt16(String)
`int`	ToInt32(String)
`long`	ToInt64(String)
`ushort`	ToUInt16(String)
`uint`	ToUInt32(String)
`ulong`	ToUInt64(String)

O exemplo a seguir chama o método Convert.ToInt32(String) para converter uma cadeia de caracteres de entrada em um int. O exemplo captura as duas exceções mais comuns geradas por esse método: FormatException e OverflowException. Se o número resultante puder ser incrementado sem exceder Int32.MaxValue, o exemplo adicionará 1 ao resultado e exibirá a saída.

C#

```
using System;

public class ConvertStringExample1
{
    static void Main(string[] args)
    {
        int numVal = -1;
        bool repeat = true;

        while (repeat)
        {
            Console.Write("Enter a number between -2,147,483,648 and
+2,147,483,647 (inclusive): ");

            string? input = Console.ReadLine();

            // ToInt32 can throw FormatException or OverflowException.
            try
            {
                numVal = Convert.ToInt32(input);
                if (numVal < Int32.MaxValue)
                {
                    Console.WriteLine("The new value is {0}", ++numVal);
                }
                else
                {
                    Console.WriteLine("numVal cannot be incremented beyond
its current value");
                }
            }
            catch (FormatException)
            {
                Console.WriteLine("Input string is not a sequence of
digits.");
            }
            catch (OverflowException)
            {
                Console.WriteLine("The number cannot fit in an Int32.");
            }

            Console.Write("Go again? Y/N: ");
            string? go = Console.ReadLine();
            if (go?.ToUpper() != "Y")
            {
                repeat = false;
```

```
            }
        }
    }
}
// Sample Output:
//   Enter a number between -2,147,483,648 and +2,147,483,647 (inclusive):
473
//   The new value is 474
//   Go again? Y/N: y
//   Enter a number between -2,147,483,648 and +2,147,483,647 (inclusive):
2147483647
//   numVal cannot be incremented beyond its current value
//   Go again? Y/N: y
//   Enter a number between -2,147,483,648 and +2,147,483,647 (inclusive):
-1000
//   The new value is -999
//   Go again? Y/N: n
```

Usar o GitHub Copilot para converter uma string em um número

Você pode usar o GitHub Copilot no IDE para gerar código para converter uma string em um número em C#.

Se você estiver usando o Visual Studio 2022 versão 17.8 ou posterior, poderá experimentar o GitHub Copilot controlado por IA no Visual Studio para gerar código para converter uma cadeia de caracteres em um número. Envie sua pergunta como um prompt na janela de bate-papo do Copilot, como no exemplo a seguir. Você também pode enviar prompts usando o bate-papo em linha na própria janela do editor.

> **Observação**
>
> O GitHub Copilot é alimentado por IA, portanto é passível de surpresas e erros. Verifique qualquer código gerado ou sugestões. Para obter mais informações sobre o uso geral do GitHub Copilot, o impacto do produto, a supervisão humana e a privacidade, consulte Perguntas frequentes sobre o GitHub Copilot.

O texto a seguir mostra um exemplo de prompt para o Copilot Chat:

> Gere código C# para converter a cadeia de caracteres "123" em um número usando TryParse. Forneça um exemplo de saída.

Você pode personalizar o prompt para usar uma cadeia de caracteres de acordo com seus requisitos.

Você pode usar recursos de bate-papo, como comandos de barra, referências a arquivos, métodos ou classes e threads, para definir a intenção e obter melhores respostas com o contexto com escopo. Para um arquivo existente que está aberto no IDE, você pode solicitar o GitHub Copilot usando o chat embutido com `/generate code to convert the string string1 in #Filename to a number` o .

A saída a seguir mostra um exemplo de resposta do Copilot Chat:

Saída

```
Here is a C# code snippet to convert the string "123" to a number using
TryParse:

using System;

class Program
{
    static void Main()
    {
        string numberString = "123";
        int number;
        bool success = int.TryParse(numberString, out number);

        if (success)
        {
            // Example output
            Console.WriteLine("The converted number is: " + number);
        }
        else
        {
            Console.WriteLine("Conversion failed.");
        }
    }
}

When you run this code, the output will be:
The converted number is: 123
```

Quando o Copilot retorna um bloco de código, a resposta inclui opções para copiar o código, inserir o código em um novo arquivo ou visualizar a saída do código.

Observação

Seus resultados podem ser diferentes do que é mostrado nas respostas de exemplo. Os modelos de IA não são determinísticos, ou seja, eles podem retornar respostas diferentes quando fazem a mesma pergunta. Isso pode ser devido a

aprendizado e adaptação adicionais ao longo do tempo, variação de idioma, mudanças no contexto, como seu histórico de bate-papo e muito mais.

GitHub Copilot Chat

Generate C# code to convert the string "123" to a number. Provide an example output.

generate c# code to convert the string "123" to a number using TryParse. Provide example output

GitHub Copilot

Here is a C# code snippet to convert the string "123" to a number using TryParse:

```
using System;

class Program
{
    static void Main()
    {
        string numberString = "123";
        int number;
        bool success = int.TryParse(numberString, out number);

        if (success)
        {
            // Example output
            Console.WriteLine("The converted number is: " + number);
        }
        else
        {
            Console.WriteLine("Conversion failed.");
        }
    }
}
```

Insert in new file Preview

When you run this code, the output will be:

```
The converted number is: 123
```

Preview

How can I convert a string to a floating-point number in C#?

References Was this helpful?

AI suggestions might be inaccurate.

Para saber mais, veja:

- Central de Confiabilidade do GitHub Copilot
- GitHub Copilot no Visual Studio
- GitHub Copilot no VS Code

Como converter entre cadeias de caracteres hexadecimais e tipos numéricos (Guia de Programação em C#)

Artigo • 10/05/2023

Estes exemplos mostram como realizar as seguintes tarefas:

- Obter o valor hexadecimal de cada caractere em uma cadeia de caracteres.
- Obter o char que corresponde a cada valor em uma cadeia de caracteres hexadecimal.
- Converter um `string` hexadecimal em um int.
- Converter um `string` hexadecimal em um float.
- Como converter uma matriz de bytes em um `string` hexadecimal.

Exemplos

Este exemplo gera o valor hexadecimal de cada caractere em um `string`. Primeiro, ele analisa o `string` como uma matriz de caracteres. Em seguida, ele chama ToInt32(Char) em cada caractere para obter seu valor numérico. Por fim, ele formata o número como sua representação hexadecimal em um `string`.

C#

```
string input = "Hello World!";
char[] values = input.ToCharArray();
foreach (char letter in values)
{
    // Get the integral value of the character.
    int value = Convert.ToInt32(letter);
    // Convert the integer value to a hexadecimal value in string form.
    Console.WriteLine($"Hexadecimal value of {letter} is {value:X}");
}
/* Output:
    Hexadecimal value of H is 48
    Hexadecimal value of e is 65
    Hexadecimal value of l is 6C
    Hexadecimal value of l is 6C
    Hexadecimal value of o is 6F
```

```
    Hexadecimal value of   is 20
    Hexadecimal value of W is 57
    Hexadecimal value of o is 6F
    Hexadecimal value of r is 72
    Hexadecimal value of l is 6C
    Hexadecimal value of d is 64
    Hexadecimal value of ! is 21
 */
```

Este exemplo analisa um `string` de valores hexadecimais e gera o caractere correspondente a cada valor hexadecimal. Primeiro, ele chama o método Split(Char[]) para obter cada valor hexadecimal como um `string` individual em uma matriz. Em seguida, ele chama ToInt32(String, Int32) para converter o valor hexadecimal em um valor decimal representado como um int. Ele mostra duas maneiras diferentes de obter o caractere correspondente a esse código de caractere. A primeira técnica usa ConvertFromUtf32(Int32), que retorna o caractere correspondente ao argumento de inteiro como um `string`. A segunda técnica converte explicitamente o `int` em um char.

C#

```
string hexValues = "48 65 6C 6C 6F 20 57 6F 72 6C 64 21";
string[] hexValuesSplit = hexValues.Split(' ');
foreach (string hex in hexValuesSplit)
{
    // Convert the number expressed in base-16 to an integer.
    int value = Convert.ToInt32(hex, 16);
    // Get the character corresponding to the integral value.
    string stringValue = Char.ConvertFromUtf32(value);
    char charValue = (char)value;
    Console.WriteLine("hexadecimal value = {0}, int value = {1}, char value
= {2} or {3}",
                      hex, value, stringValue, charValue);
}
/* Output:
    hexadecimal value = 48, int value = 72, char value = H or H
    hexadecimal value = 65, int value = 101, char value = e or e
    hexadecimal value = 6C, int value = 108, char value = l or l
    hexadecimal value = 6C, int value = 108, char value = l or l
    hexadecimal value = 6F, int value = 111, char value = o or o
    hexadecimal value = 20, int value = 32, char value =   or
    hexadecimal value = 57, int value = 87, char value = W or W
    hexadecimal value = 6F, int value = 111, char value = o or o
    hexadecimal value = 72, int value = 114, char value = r or r
    hexadecimal value = 6C, int value = 108, char value = l or l
    hexadecimal value = 64, int value = 100, char value = d or d
    hexadecimal value = 21, int value = 33, char value = ! or !
*/
```

Este exemplo mostra outra maneira de converter um hexadecimal `string` em um inteiro, chamando o método Parse(String, NumberStyles).

C#

```
string hexString = "8E2";
int num = Int32.Parse(hexString,
System.Globalization.NumberStyles.HexNumber);
Console.WriteLine(num);
//Output: 2274
```

O exemplo a seguir mostra como converter um hexadecimal `string` para um float usando a classe System.BitConverter e o método UInt32.Parse.

C#

```
string hexString = "43480170";
uint num = uint.Parse(hexString,
System.Globalization.NumberStyles.AllowHexSpecifier);

byte[] floatVals = BitConverter.GetBytes(num);
float f = BitConverter.ToSingle(floatVals, 0);
Console.WriteLine("float convert = {0}", f);

// Output: 200.0056
```

O exemplo a seguir mostra como converter uma matriz de bytes em uma cadeia de caracteres hexadecimal usando a classe System.BitConverter.

C#

```
byte[] vals = { 0x01, 0xAA, 0xB1, 0xDC, 0x10, 0xDD };

string str = BitConverter.ToString(vals);
Console.WriteLine(str);

str = BitConverter.ToString(vals).Replace("-", "");
Console.WriteLine(str);

/*Output:
  01-AA-B1-DC-10-DD
  01AAB1DC10DD
 */
```

O exemplo a seguir mostra como converter uma matriz de byte em uma cadeia de caracteres hexadecimal chamando o método Convert.ToHexString, introduzido no .NET 5.0.

C#

```
byte[] array = { 0x64, 0x6f, 0x74, 0x63, 0x65, 0x74 };

string hexValue = Convert.ToHexString(array);
Console.WriteLine(hexValue);

/*Output:
  646F74636574
 */
```

Confira também

- Cadeias de Caracteres de Formato Numérico Padrão
- Types
- Como determinar se uma cadeia de caracteres representa um valor numérico

Controle de versão com as palavras-chave Override e New (Guia de Programação em C#)

19/06/2025

A linguagem C# foi projetada para que o controle de versão entre classes base e derivadas em bibliotecas diferentes possa evoluir e manter a compatibilidade com versões anteriores. Isso significa, por exemplo, que a introdução de um novo membro em uma classe base com o mesmo nome de um membro em uma classe derivada é completamente suportada por C# e não leva a um comportamento inesperado. Isso também significa que uma classe deve declarar explicitamente se um método se destina a substituir um método herdado ou se um método é um novo método que oculta um método herdado de nome semelhante.

Em C#, classes derivadas podem conter métodos com o mesmo nome que métodos de classe base.

- Se o método na classe derivada não for precedido por palavras-chave novas ou de substituição , o compilador emitirá um aviso e o método se comportará como se a `new` palavra-chave estivesse presente.
- Se o método na classe derivada for precedido com a `new` palavra-chave, o método será definido como independente do método na classe base.
- Se o método na classe derivada for precedido com a `override` palavra-chave, os objetos da classe derivada chamarão esse método em vez do método de classe base.
- Para aplicar a `override` palavra-chave ao método na classe derivada, o método de classe base deve ser definido como virtual.
- O método de classe base pode ser chamado de dentro da classe derivada usando a `base` palavra-chave.
- As `override` palavras-chave , `virtual` e `new` também podem ser aplicadas a propriedades, indexadores e eventos.

Por padrão, os métodos C# não são virtuais. Se um método for declarado como virtual, qualquer classe que herda o método poderá implementar sua própria versão. Para tornar um método virtual, o `virtual` modificador é usado na declaração de método da classe base. A classe derivada pode substituir o método virtual base usando a `override` palavra-chave ou ocultar o método virtual na classe base usando a `new` palavra-chave. Se nem a `override`

palavra-chave nem a `new` palavra-chave forem especificadas, o compilador emitirá um aviso e o método na classe derivada ocultará o método na classe base.

Para demonstrar isso na prática, suponha por um momento que a Empresa A tenha criado uma classe chamada `GraphicsClass`, que seu programa usa. A seguir está `GraphicsClass`:

C#

```
class GraphicsClass
{
    public virtual void DrawLine() { }
    public virtual void DrawPoint() { }
}
```

Sua empresa usa essa classe e você a usa para derivar sua própria classe, adicionando um novo método:

C#

```
class YourDerivedGraphicsClass : GraphicsClass
{
    public void DrawRectangle() { }
}
```

Seu aplicativo é usado sem problemas até que a Empresa A libere uma nova versão, `GraphicsClass` que se assemelha ao seguinte código:

C#

```
class GraphicsClass
{
    public virtual void DrawLine() { }
    public virtual void DrawPoint() { }
    public virtual void DrawRectangle() { }
}
```

A nova versão de `GraphicsClass` agora contém um método chamado `DrawRectangle`. Inicialmente, nada ocorre. A nova versão ainda é compatível em termos binários com a versão antiga. Qualquer software implantado continuará funcionando, mesmo que a nova classe esteja instalada nesses sistemas de computador. Todas as chamadas existentes para o método `DrawRectangle` continuarão a referenciar sua versão, em sua classe derivada.

No entanto, assim que você recompilar seu aplicativo usando a nova versão, `GraphicsClass` receberá um aviso do compilador CS0108. Este aviso informa que você precisa considerar como deseja que seu `DrawRectangle` método se comporte em seu aplicativo.

Se você quiser que seu método substitua o novo método de classe base, use a `override` palavra-chave:

```
C#

class YourDerivedGraphicsClass : GraphicsClass
{
    public override void DrawRectangle() { }
}
```

A `override` palavra-chave garante que todos os objetos derivados `YourDerivedGraphicsClass` usem a versão de classe derivada de `DrawRectangle`. Objetos derivados de `YourDerivedGraphicsClass` ainda podem acessar a versão da classe base de `DrawRectangle` usando a palavra-chave base:

```
C#

base.DrawRectangle();
```

Se você não quiser que seu método substitua o novo método de classe base, as considerações a seguir se aplicam. Para evitar confusão entre os dois métodos, você pode renomear seu método. Isso pode ser demorado e propenso a erros, e apenas não prático em alguns casos. No entanto, se o projeto for relativamente pequeno, você poderá usar as opções de Refatoração do Visual Studio para renomear o método. Para obter mais informações, consulte Refatorando classes e tipos (Designer de Classe).

Como alternativa, você pode impedir o aviso usando a palavra-chave `new` em sua definição de classe derivada:

```
C#

class YourDerivedGraphicsClass : GraphicsClass
{
    public new void DrawRectangle() { }
}
```

Usar a `new` palavra-chave informa ao compilador que sua definição oculta a definição contida na classe base. Esse é o comportamento padrão.

Seleção de método e substituição

Quando um método é nomeado em uma classe, o compilador C# seleciona o melhor método para chamar se mais de um método for compatível com a chamada, como quando há dois

métodos com o mesmo nome e parâmetros compatíveis com o parâmetro passado. Os seguintes métodos seriam compatíveis:

```C#
public class Derived : Base
{
    public override void DoWork(int param) { }
    public void DoWork(double param) { }
}
```

Quando `DoWork` for chamado em uma instância de `Derived`, o compilador C# primeiro tentará tornar a chamada compatível com as versões do `DoWork` declarado originalmente em `Derived`. Os métodos de substituição não são considerados como declarados em uma classe, são novas implementações de um método declarado em uma classe base. Somente se o compilador C# não puder corresponder à chamada de método a um método `Derived` original, ele tentará corresponder a chamada a um método substituído com o mesmo nome e parâmetros compatíveis. Por exemplo:

```C#
int val = 5;
Derived d = new();
d.DoWork(val);  // Calls DoWork(double).
```

Como a variável `val` pode ser convertida em um duplo implicitamente, o compilador C# chama `DoWork(double)` em vez de `DoWork(int)`. Há duas maneiras de evitar isso. Primeiro, evite declarar novos métodos com o mesmo nome que métodos virtuais. Em segundo lugar, você pode instruir o compilador C# a chamar o método virtual para que ele pesquise na lista de métodos da classe base, convertendo a instância de `Derived` para `Base`. Como o método é virtual, a implementação de `DoWork(int)` em `Derived` será chamada. Por exemplo:

```C#
((Base)d).DoWork(val);  // Calls DoWork(int) on Derived.
```

Para obter mais exemplos de `new` e `override`, consulte Saber quando usar substituição e novas palavras-chave.

Consulte também

- Sistema de tipos do C#

- Métodos
- Herança

Saber quando usar substituição e novas palavras-chave (Guia de Programação em C#)

19/06/2025

Em C#, um método em uma classe derivada pode ter o mesmo nome de um método na classe base. Você pode especificar como os métodos interagem usando as palavras-chave novas e substituídas . O `override` modificador *estende* o método de classe `virtual` base e o `new` modificador *oculta* um método de classe base acessível. A diferença é ilustrada nos exemplos deste tópico.

Em um aplicativo de console, declare as duas classes `BaseClass` a seguir e `DerivedClass`. `DerivedClass` herda de `BaseClass`.

C#

```
class BaseClass
{
    public void Method1()
    {
        Console.WriteLine("Base - Method1");
    }
}

class DerivedClass : BaseClass
{
    public void Method2()
    {
        Console.WriteLine("Derived - Method2");
    }
}
```

`Main` No método, declare variáveis `bc` e `dc` `bcdc`.

- `bc` é do tipo `BaseClass`, e seu valor é do tipo `BaseClass`.
- `dc` é do tipo `DerivedClass`, e seu valor é do tipo `DerivedClass`.
- `bcdc` é do tipo `BaseClass`, e seu valor é do tipo `DerivedClass`. Essa é a variável à qual prestar atenção.

Como `bc` e `bcdc` têm o tipo `BaseClass`, eles podem ter acesso direto a `Method1`, a menos que você usa a conversão. A variável `dc` pode acessar ambos `Method1` e `Method2`. Essas relações são mostradas no código a seguir.

C#

```
class Program
{
    static void Main(string[] args)
    {
        BaseClass bc = new BaseClass();
        DerivedClass dc = new DerivedClass();
        BaseClass bcdc = new DerivedClass();

        bc.Method1();
        dc.Method1();
        dc.Method2();
        bcdc.Method1();
    }
    // Output:
    // Base - Method1
    // Base - Method1
    // Derived - Method2
    // Base - Method1
}
```

Em seguida, adicione o método a seguir `Method2` a `BaseClass`. A assinatura desse método corresponde à assinatura do `Method2` método em `DerivedClass`.

C#

```
public void Method2()
{
    Console.WriteLine("Base - Method2");
}
```

Como `BaseClass` agora tem um `Method2` método, uma segunda instrução de chamada pode ser adicionada para `BaseClass` variáveis `bc` e `bcdc`, conforme mostrado no código a seguir.

C#

```
bc.Method1();
bc.Method2();
dc.Method1();
dc.Method2();
bcdc.Method1();
bcdc.Method2();
```

Ao compilar o projeto, você verá que a adição do método `Method2` em `BaseClass` causa um aviso. O aviso diz que o `Method2` método em `DerivedClass` oculta o `Method2` método em `BaseClass`. Recomenda-se usar a palavra-chave `new` na definição `Method2` se você pretende

causar esse resultado. Como alternativa, você pode renomear um dos `Method2` métodos para resolver o aviso, mas isso nem sempre é prático.

Antes de adicionar `new`, execute o programa para ver a saída produzida pelas instruções de chamada adicionais. Os resultados a seguir são exibidos.

C#

```
// Output:
// Base - Method1
// Base - Method2
// Base - Method1
// Derived - Method2
// Base - Method1
// Base - Method2
```

A `new` palavra-chave preserva as relações que produzem essa saída, mas suprime o aviso. As variáveis que têm tipo `BaseClass` continuam a acessar os membros de `BaseClass`, e a variável que tem tipo `DerivedClass` continua a acessar os membros em `DerivedClass` primeiro, e então considerar os membros herdados de `BaseClass`.

Para suprimir o aviso, adicione o modificador `new` à definição de `Method2` em `DerivedClass`, conforme mostrado no código a seguir. O modificador pode ser adicionado antes ou depois `public`.

C#

```
public new void Method2()
{
    Console.WriteLine("Derived - Method2");
}
```

Execute o programa novamente para verificar se a saída não foi alterada. Verifique também se o aviso não é mais exibido. Ao usar `new`, você está afirmando que está ciente de que o membro que ele modifica oculta um membro herdado da classe base. Para obter mais informações sobre o ocultamento de nome por meio da herança, consulte o novo Modificador.

Para contrastar esse comportamento com os efeitos do uso `override`, adicione o método a seguir a `DerivedClass`. O `override` modificador pode ser adicionado antes ou depois `public`.

C#

```
public override void Method1()
{
```

```
    Console.WriteLine("Derived - Method1");
}
```

Adicione o `virtual` modificador à definição de `Method1` em `BaseClass`. O `virtual` modificador pode ser adicionado antes ou depois `public`.

C#

```
public virtual void Method1()
{
    Console.WriteLine("Base - Method1");
}
```

Execute o projeto novamente. Observe principalmente as duas últimas linhas da saída a seguir.

C#

```
// Output:
// Base - Method1
// Base - Method2
// Derived - Method1
// Derived - Method2
// Derived - Method1
// Base - Method2
```

O uso do `override` modificador permite `bcdc` acessar o `Method1` método definido em `DerivedClass`. Normalmente, esse é o comportamento desejado nas hierarquias de herança. Você deseja que os objetos que têm valores criados a partir da classe derivada usem os métodos definidos na classe derivada. Você obtém esse comportamento usando `override` para estender o método de classe base.

O código a seguir contém o exemplo completo.

C#

```
using System;
using System.Text;

namespace OverrideAndNew
{
    class Program
    {
        static void Main(string[] args)
        {
            BaseClass bc = new BaseClass();
            DerivedClass dc = new DerivedClass();
            BaseClass bcdc = new DerivedClass();
```

```
            // The following two calls do what you would expect. They call
            // the methods that are defined in BaseClass.
            bc.Method1();
            bc.Method2();
            // Output:
            // Base - Method1
            // Base - Method2

            // The following two calls do what you would expect. They call
            // the methods that are defined in DerivedClass.
            dc.Method1();
            dc.Method2();
            // Output:
            // Derived - Method1
            // Derived - Method2

            // The following two calls produce different results, depending
            // on whether override (Method1) or new (Method2) is used.
            bcdc.Method1();
            bcdc.Method2();
            // Output:
            // Derived - Method1
            // Base - Method2
        }
    }

    class BaseClass
    {
        public virtual void Method1()
        {
            Console.WriteLine("Base - Method1");
        }

        public virtual void Method2()
        {
            Console.WriteLine("Base - Method2");
        }
    }

    class DerivedClass : BaseClass
    {
        public override void Method1()
        {
            Console.WriteLine("Derived - Method1");
        }

        public new void Method2()
        {
            Console.WriteLine("Derived - Method2");
        }
    }
}
```

O exemplo a seguir ilustra um comportamento semelhante em um contexto diferente. O exemplo define três classes: uma classe base nomeada `Car` e duas classes derivadas dela e `ConvertibleCar` `Minivan`. A classe base contém um `DescribeCar` método. O método exibe uma descrição básica de um carro e, em seguida, chama `ShowDetails` para fornecer informações adicionais. Cada uma das três classes define um `ShowDetails` método. O `new` modificador é usado para definir `ShowDetails` na `ConvertibleCar` classe. O `override` modificador é usado para definir `ShowDetails` na `Minivan` classe.

C#

```
// Define the base class, Car. The class defines two methods,
// DescribeCar and ShowDetails. DescribeCar calls ShowDetails, and each derived
// class also defines a ShowDetails method. The example tests which version of
// ShowDetails is selected, the base class method or the derived class method.
class Car
{
    public void DescribeCar()
    {
        System.Console.WriteLine("Four wheels and an engine.");
        ShowDetails();
    }

    public virtual void ShowDetails()
    {
        System.Console.WriteLine("Standard transportation.");
    }
}

// Define the derived classes.

// Class ConvertibleCar uses the new modifier to acknowledge that ShowDetails
// hides the base class method.
class ConvertibleCar : Car
{
    public new void ShowDetails()
    {
        System.Console.WriteLine("A roof that opens up.");
    }
}

// Class Minivan uses the override modifier to specify that ShowDetails
// extends the base class method.
class Minivan : Car
{
    public override void ShowDetails()
    {
        System.Console.WriteLine("Carries seven people.");
    }
}
```

O exemplo testa qual versão de `ShowDetails` é chamada. O método a seguir, `TestCars1`, declara uma instância de cada classe e, em seguida, chama `DescribeCar` em cada instância.

C#

```
public static void TestCars1()
{
    System.Console.WriteLine("\nTestCars1");
    System.Console.WriteLine("----------");

    Car car1 = new Car();
    car1.DescribeCar();
    System.Console.WriteLine("----------");

    // Notice the output from this test case. The new modifier is
    // used in the definition of ShowDetails in the ConvertibleCar
    // class.

    ConvertibleCar car2 = new ConvertibleCar();
    car2.DescribeCar();
    System.Console.WriteLine("----------");

    Minivan car3 = new Minivan();
    car3.DescribeCar();
    System.Console.WriteLine("----------");
}
```

`TestCars1` produz a saída a seguir. Observe especialmente os resultados para `car2`, que provavelmente não são o que você esperava. O tipo do objeto é `ConvertibleCar`, mas `DescribeCar` não acessa a versão do `ShowDetails` que é definido na `ConvertibleCar` classe porque esse método é declarado com o `new` modificador, não com o `override` modificador. Como resultado, um `ConvertibleCar` objeto exibe a mesma descrição de um `Car` objeto. Contraste os resultados para `car3`, que é um objeto `Minivan`. Nesse caso, o `ShowDetails` método declarado na `Minivan` classe substitui o `ShowDetails` método que é declarado na `Car` classe e a descrição exibida descreve uma minivan.

C#

```
// TestCars1
// ----------
// Four wheels and an engine.
// Standard transportation.
// ----------
// Four wheels and an engine.
// Standard transportation.
// ----------
// Four wheels and an engine.
```

```
// Carries seven people.
// ----------
```

`TestCars2` cria uma lista de objetos que têm o tipo `Car`. Os valores dos objetos são instanciados das classes `Car`, `ConvertibleCar`, e `Minivan`. `DescribeCar` é aplicado a cada elemento da lista. O código a seguir mostra a definição de `TestCars2`.

C#

```
public static void TestCars2()
{
    System.Console.WriteLine("\nTestCars2");
    System.Console.WriteLine("----------");

    var cars = new List<Car> { new Car(), new ConvertibleCar(),
        new Minivan() };

    foreach (var car in cars)
    {
        car.DescribeCar();
        System.Console.WriteLine("----------");
    }
}
```

O resultado a seguir é exibido. Observe que é o mesmo que a saída exibida por `TestCars1`. O `ShowDetails` método da `ConvertibleCar` classe não é chamado, independentemente de o tipo do objeto ser `ConvertibleCar`, como em `TestCars1`, ou `Car`, como em `TestCars2`. Por outro lado, `car3` chama o método `ShowDetails` da classe `Minivan` em ambos os casos, independentemente de ser do tipo `Minivan` ou do tipo `Car`.

C#

```
// TestCars2
// ----------
// Four wheels and an engine.
// Standard transportation.
// ----------
// Four wheels and an engine.
// Standard transportation.
// ----------
// Four wheels and an engine.
// Carries seven people.
// ----------
```

Métodos `TestCars3` e `TestCars4` concluem o exemplo. Esses métodos chamam `ShowDetails` diretamente, primeiro de objetos declarados para ter tipo `ConvertibleCar` e `Minivan`

(`TestCars3`), em seguida, de objetos declarados para ter tipo `Car` (`TestCars4`). O código a seguir define esses dois métodos.

C#

```
public static void TestCars3()
{
    System.Console.WriteLine("\nTestCars3");
    System.Console.WriteLine("----------");
    ConvertibleCar car2 = new ConvertibleCar();
    Minivan car3 = new Minivan();
    car2.ShowDetails();
    car3.ShowDetails();
}

public static void TestCars4()
{
    System.Console.WriteLine("\nTestCars4");
    System.Console.WriteLine("----------");
    Car car2 = new ConvertibleCar();
    Car car3 = new Minivan();
    car2.ShowDetails();
    car3.ShowDetails();
}
```

Os métodos produzem a saída a seguir, que corresponde aos resultados do primeiro exemplo neste tópico.

C#

```
// TestCars3
// ----------
// A roof that opens up.
// Carries seven people.

// TestCars4
// ----------
// Standard transportation.
// Carries seven people.
```

O código a seguir mostra o projeto completo e sua saída.

C#

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace OverrideAndNew2
```

```
{
    class Program
    {
        static void Main(string[] args)
        {
            // Declare objects of the derived classes and test which version
            // of ShowDetails is run, base or derived.
            TestCars1();

            // Declare objects of the base class, instantiated with the
            // derived classes, and repeat the tests.
            TestCars2();

            // Declare objects of the derived classes and call ShowDetails
            // directly.
            TestCars3();

            // Declare objects of the base class, instantiated with the
            // derived classes, and repeat the tests.
            TestCars4();
        }

        public static void TestCars1()
        {
            System.Console.WriteLine("\nTestCars1");
            System.Console.WriteLine("----------");

            Car car1 = new Car();
            car1.DescribeCar();
            System.Console.WriteLine("----------");

            // Notice the output from this test case. The new modifier is
            // used in the definition of ShowDetails in the ConvertibleCar
            // class.
            ConvertibleCar car2 = new ConvertibleCar();
            car2.DescribeCar();
            System.Console.WriteLine("----------");

            Minivan car3 = new Minivan();
            car3.DescribeCar();
            System.Console.WriteLine("----------");
        }
        // Output:
        // TestCars1
        // ----------
        // Four wheels and an engine.
        // Standard transportation.
        // ----------
        // Four wheels and an engine.
        // Standard transportation.
        // ----------
        // Four wheels and an engine.
        // Carries seven people.
        // ----------
```

```
        public static void TestCars2()
        {
            System.Console.WriteLine("\nTestCars2");
            System.Console.WriteLine("----------");

            var cars = new List<Car> { new Car(), new ConvertibleCar(),
                new Minivan() };

            foreach (var car in cars)
            {
                car.DescribeCar();
                System.Console.WriteLine("----------");
            }
        }
        // Output:
        // TestCars2
        // ----------
        // Four wheels and an engine.
        // Standard transportation.
        // ----------
        // Four wheels and an engine.
        // Standard transportation.
        // ----------
        // Four wheels and an engine.
        // Carries seven people.
        // ----------

        public static void TestCars3()
        {
            System.Console.WriteLine("\nTestCars3");
            System.Console.WriteLine("----------");
            ConvertibleCar car2 = new ConvertibleCar();
            Minivan car3 = new Minivan();
            car2.ShowDetails();
            car3.ShowDetails();
        }
        // Output:
        // TestCars3
        // ----------
        // A roof that opens up.
        // Carries seven people.

        public static void TestCars4()
        {
            System.Console.WriteLine("\nTestCars4");
            System.Console.WriteLine("----------");
            Car car2 = new ConvertibleCar();
            Car car3 = new Minivan();
            car2.ShowDetails();
            car3.ShowDetails();
        }
        // Output:
        // TestCars4
        // ----------
        // Standard transportation.
```

```
        // Carries seven people.
    }

    // Define the base class, Car. The class defines two virtual methods,
    // DescribeCar and ShowDetails. DescribeCar calls ShowDetails, and each
derived
    // class also defines a ShowDetails method. The example tests which version of
    // ShowDetails is used, the base class method or the derived class method.
    class Car
    {
        public virtual void DescribeCar()
        {
            System.Console.WriteLine("Four wheels and an engine.");
            ShowDetails();
        }

        public virtual void ShowDetails()
        {
            System.Console.WriteLine("Standard transportation.");
        }
    }

    // Define the derived classes.

    // Class ConvertibleCar uses the new modifier to acknowledge that ShowDetails
    // hides the base class method.
    class ConvertibleCar : Car
    {
        public new void ShowDetails()
        {
            System.Console.WriteLine("A roof that opens up.");
        }
    }

    // Class Minivan uses the override modifier to specify that ShowDetails
    // extends the base class method.
    class Minivan : Car
    {
        public override void ShowDetails()
        {
            System.Console.WriteLine("Carries seven people.");
        }
    }

}
```

Consulte também

- Sistema de tipos do C#
- Controle de versão com as palavras-chave override e new
- base

- resumo

Como substituir o método ToString (Guia de Programação em C#)

Artigo • 07/04/2023

Cada classe ou struct no C# herda implicitamente a classe Object. Portanto, cada objeto no C# obtém o método ToString, que retorna uma representação de cadeia de caracteres desse objeto. Por exemplo, todas as variáveis do tipo `int` tem um método `ToString`, que permite retornar seus conteúdos como uma cadeia de caracteres:

```
C#

int x = 42;
string strx = x.ToString();
Console.WriteLine(strx);
// Output:
// 42
```

Ao criar uma classe ou struct personalizada, é necessário substituir o método ToString a fim de fornecer informações sobre o tipo ao código cliente.

Para obter informações sobre como usar cadeias de caracteres de formato e outros tipos de formatação personalizada com o método `ToString`, consulte Tipos de Formatação.

> **ⓘ Importante**
>
> Ao decidir quais informações devem ser fornecidas por meio desse método, considere se a classe ou struct será utilizado por código não confiável. Assegure-se de que nenhuma informação que possa ser explorada por código mal-intencionado seja fornecida.

Substituir o método `ToString` na classe ou struct:

1. Declare um método `ToString` com os seguintes modificadores e tipo retornado:

   ```
   C#

   public override string ToString(){}
   ```

2. Implemente o método para que ele retorne uma cadeia de caracteres.

O exemplo a seguir retorna o nome da classe, além dos dados específicos de uma instância particular da classe.

```C#
class Person
{
    public string Name { get; set; }
    public int Age { get; set; }

    public override string ToString()
    {
        return "Person: " + Name + " " + Age;
    }
}
```

É possível testar o método `ToString`, conforme mostrado no exemplo de código a seguir:

```C#
Person person = new Person { Name = "John", Age = 12 };
Console.WriteLine(person);
// Output:
// Person: John 12
```

Confira também

- IFormattable
- Guia de Programação em C#
- O sistema de tipos C#
- Cadeias de caracteres
- cadeia de caracteres
- override
- virtual
- Formatar tipos

Membros (Guia de Programação em C#)

17/06/2025

Classes e structs têm membros que representam seus dados e comportamento. Os membros de uma classe incluem todos os membros declarados na classe, juntamente com todos os membros (exceto construtores e finalizadores) declarados em todas as classes em sua hierarquia de herança. Os membros privados em classes base são herdados, mas não podem ser acessados de classes derivadas.

A tabela a seguir lista os tipos de membros que uma classe ou struct pode conter:

Expandir a tabela

Membro	Descrição
Campos	Os campos são variáveis declaradas no escopo da classe. Um campo pode ser um tipo numérico interno ou uma instância de outra classe. Por exemplo, uma classe de calendário pode ter um campo que contém a data atual.
constantes	Constantes são campos cujo valor é definido em tempo de compilação e não podem ser alterados.
Propriedades	As propriedades são métodos em uma classe que são acessadas como se fossem campos nessa classe. Uma propriedade pode fornecer proteção para um campo de classe para evitar que ele seja alterado sem o conhecimento do objeto.
Métodos	Os métodos definem as ações que uma classe pode executar. Os métodos podem usar parâmetros que fornecem dados de entrada e podem retornar dados de saída por meio de parâmetros. Os métodos também podem retornar um valor diretamente, sem usar um parâmetro.
Eventos	Os eventos fornecem notificações sobre ocorrências, como cliques de botão ou a conclusão bem-sucedida de um método, para outros objetos. Os eventos são definidos e disparados usando delegados.
Operadores	Operadores sobrecarregados são considerados membros do tipo. Ao sobrecarregar um operador, você o define como um método público em um tipo. Para obter mais informações, consulte Sobrecarga de operador.
indexadores	Os indexadores permitem que um objeto seja indexado de maneira semelhante às matrizes.
Construtores	Construtores são métodos que são chamados quando o objeto é criado pela primeira vez. Eles geralmente são usados para inicializar os dados de um objeto.
Finalizadores	Os finalizadores são usados muito raramente em C#. São métodos chamados pelo mecanismo de execução de runtime quando o objeto está prestes a ser removido da

Membro	Descrição
	memória. Eles geralmente são usados para garantir que todos os recursos que devem ser liberados sejam tratados adequadamente.
Tipos aninhados	Tipos aninhados são tipos declarados em outro tipo. Tipos aninhados geralmente são usados para descrever objetos que são usados apenas pelos tipos que os contêm.

Consulte também

- Classes

Classes abstratas e lacradas e membros de classe (Guia de Programação em C#)

19/06/2025

A palavra-chave abstrata permite que você crie classes e membros de classe incompletos e que devem ser implementados em uma classe derivada.

A palavra-chave selada permite evitar a herança de uma classe ou de membros específicos da classe que foram marcados anteriormente como virtuais.

Classes abstratas e membros de classe

As classes podem ser declaradas como abstratas colocando a palavra-chave `abstract` antes da definição da classe. Por exemplo:

C#

```
public abstract class A
{
    // Class members here.
}
```

Uma classe abstrata não pode ser instanciada. A finalidade de uma classe abstrata é fornecer uma definição comum de uma classe base que várias classes derivadas podem compartilhar. Por exemplo, uma biblioteca de classes pode definir uma classe abstrata que é usada como um parâmetro para muitas de suas funções e exigir que os programadores usem essa biblioteca para fornecer sua própria implementação da classe criando uma classe derivada.

Classes abstratas também podem definir métodos abstratos. Isso é feito adicionando a palavra-chave `abstract` antes do tipo de retorno do método. Por exemplo:

C#

```
public abstract class A
{
    public abstract void DoWork(int i);
}
```

Métodos abstratos não têm implementação, portanto, a definição do método é seguida por um ponto-e-vírgula em vez de um bloco de método normal. Classes derivadas da classe abstrata devem implementar todos os métodos abstratos. Quando uma classe abstrata herda

um método virtual de uma classe base, a classe abstrata pode substituir o método virtual por um método abstrato. Por exemplo:

```C#
// compile with: -target:library
public class D
{
    public virtual void DoWork(int i)
    {
        // Original implementation.
    }
}

public abstract class E : D
{
    public abstract override void DoWork(int i);
}

public class F : E
{
    public override void DoWork(int i)
    {
        // New implementation.
    }
}
```

Se um método `virtual` for declarado `abstract`, ele ainda será virtual para qualquer classe que herdar da classe abstrata. Uma classe que herda um método abstrato não pode acessar a implementação original do método, no exemplo anterior, `DoWork` na classe F não pode chamar `DoWork` na classe D. Dessa forma, uma classe abstrata pode forçar classes derivadas a fornecer novas implementações de método para métodos virtuais.

Classes seladas e membros de classe

As classes podem ser declaradas como seladas colocando a palavra-chave `sealed` antes da definição da classe. Por exemplo:

```C#
public sealed class D
{
    // Class members here.
}
```

Uma classe lacrada não pode ser usada como uma classe base. Por esse motivo, também não pode ser uma classe abstrata. Classes seladas impedem a derivação. Como eles nunca podem

ser usados como uma classe base, algumas otimizações em tempo de execução podem tornar a chamada de membros de classe selada um pouco mais rápida.

Um método, indexador, propriedade ou evento, em uma classe derivada que está substituindo um membro virtual da classe base pode declarar esse membro como lacrado. Isso nega o aspecto virtual do membro para qualquer classe derivada adicional. Isso é feito colocando a `sealed` palavra-chave antes da palavra-chave override na declaração de membro da classe. Por exemplo:

```
C#

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

Consulte também

- Sistema de tipos do C#
- Herança
- Métodos
- Campos
- Como definir propriedades abstratas

Classes estáticas e membros de classe estática (Guia de Programação em C#)

19/06/2025

Uma classe estática é basicamente a mesma que uma classe não estática, mas há uma diferença: uma classe estática não pode ser instanciada. Em outras palavras, você não pode usar o novo operador para criar uma variável do tipo de classe. Como não há nenhuma variável de instância, você acessa os membros de uma classe estática usando o próprio nome de classe. Por exemplo, se você tiver uma classe estática nomeada `UtilityClass` que tenha um método estático público chamado `MethodA`, chame o método, conforme mostrado no exemplo a seguir:

C#

```
UtilityClass.MethodA();
```

Uma classe estática pode ser usada como um contêiner conveniente para conjuntos de métodos que operam apenas em parâmetros de entrada e não precisam obter ou definir nenhum campo de instância interna. Por exemplo, na Biblioteca de Classes do .NET, a classe estática System.Math contém métodos que executam operações matemáticas, sem nenhum requisito para armazenar ou recuperar dados exclusivos de uma instância específica da Math classe. Ou seja, você aplica os membros da classe especificando o nome da classe e o nome do método, conforme mostrado no exemplo a seguir.

C#

```
double dub = -3.14;
Console.WriteLine(Math.Abs(dub));
Console.WriteLine(Math.Floor(dub));
Console.WriteLine(Math.Round(Math.Abs(dub)));

// Output:
// 3.14
// -4
// 3
```

Como é o caso de todos os tipos de classe, o runtime do .NET carrega as informações de tipo para uma classe estática quando o programa que faz referência à classe é carregado. O programa não pode especificar exatamente quando a classe é carregada. No entanto, é garantido carregar e ter seus campos inicializados e seu construtor estático chamado antes que a classe seja referenciada pela primeira vez em seu programa. Um construtor estático é chamado apenas uma vez e uma classe estática permanece na memória durante o tempo de vida do domínio do aplicativo no qual o programa reside.

> ⓘ **Observação**
>
> Para criar uma classe não estática que permita que apenas uma instância de si mesma seja criada, consulte Implementando Singleton em C#.

A lista a seguir fornece os principais recursos de uma classe estática:

- Contém apenas membros estáticos.
- Não pode ser instanciado.
- Está lacrado.
- Não pode conter construtores de instâncias.

A criação de uma classe estática é, portanto, basicamente a mesma que a criação de uma classe que contém apenas membros estáticos e um construtor privado. Um construtor privado impede que a classe seja instanciada. A vantagem de usar uma classe estática é que o compilador pode verificar se nenhum membro de instância foi adicionado acidentalmente. O compilador garante que as instâncias dessa classe não possam ser criadas.

As classes estáticas são lacradas e, portanto, não podem ser herdadas. Eles não podem herdar de nenhuma classe ou interface, exceto Object. Classes estáticas não podem conter um construtor de instância. No entanto, eles podem conter um construtor estático. Classes não estáticas também devem definir um construtor estático se a classe contiver membros estáticos que exigem inicialização não trivial. Para obter mais informações, consulte Construtores Estáticos.

Exemplo

Aqui está um exemplo de uma classe estática que contém dois métodos que convertem a temperatura de Celsius em Fahrenheit e de Fahrenheit para Celsius:

C#

```
public static class TemperatureConverter
{
    public static double CelsiusToFahrenheit(string temperatureCelsius)
    {
        // Convert argument to double for calculations.
        double celsius = Double.Parse(temperatureCelsius);

        // Convert Celsius to Fahrenheit.
        double fahrenheit = (celsius * 9 / 5) + 32;
```

```
        return fahrenheit;
    }

    public static double FahrenheitToCelsius(string temperatureFahrenheit)
    {
        // Convert argument to double for calculations.
        double fahrenheit = Double.Parse(temperatureFahrenheit);

        // Convert Fahrenheit to Celsius.
        double celsius = (fahrenheit - 32) * 5 / 9;

        return celsius;
    }
}

class TestTemperatureConverter
{
    static void Main()
    {
        Console.WriteLine("Please select the convertor direction");
        Console.WriteLine("1. From Celsius to Fahrenheit.");
        Console.WriteLine("2. From Fahrenheit to Celsius.");
        Console.Write(":");

        string? selection = Console.ReadLine();
        double F, C = 0;

        switch (selection)
        {
            case "1":
                Console.Write("Please enter the Celsius temperature: ");
                F = TemperatureConverter.CelsiusToFahrenheit(Console.ReadLine() ??
"0");
                Console.WriteLine($"Temperature in Fahrenheit: {F:F2}");
                break;

            case "2":
                Console.Write("Please enter the Fahrenheit temperature: ");
                C = TemperatureConverter.FahrenheitToCelsius(Console.ReadLine() ??
"0");
                Console.WriteLine($"Temperature in Celsius: {C:F2}");
                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.
 */
```

Membros estáticos

Uma classe não estática pode conter métodos estáticos, campos, propriedades ou eventos. O membro estático pode ser chamado em uma classe mesmo quando nenhuma instância da classe existe. O membro estático sempre é acessado pelo nome da classe, não pelo nome da instância. Existe apenas uma cópia de um membro estático, independentemente de quantas instâncias da classe são criadas. Métodos e propriedades estáticos não podem acessar campos e eventos não estáticos em seu tipo de contenção e não podem acessar uma variável de instância de qualquer objeto, a menos que seja explicitamente passado em um parâmetro de método.

É mais comum declarar uma classe não estática com alguns membros estáticos do que declarar uma classe inteira como estática. Dois usos comuns de campos estáticos são manter uma contagem do número de objetos instanciados ou armazenar um valor que deve ser compartilhado entre todas as instâncias.

Métodos estáticos podem ser sobrecarregados, mas não substituídos, porque pertencem à classe e não a nenhuma instância da classe.

Embora um campo não possa ser declarado como `static const`, um campo const é essencialmente estático em seu comportamento. Ele pertence ao tipo, não a instâncias do tipo. Portanto, `const` campos podem ser acessados usando a mesma notação de `ClassName.MemberName` usada para campos estáticos. Nenhuma instância de objeto é necessária.

O C# não dá suporte a variáveis locais estáticas (ou seja, variáveis que são declaradas no escopo do método).

Você declara membros de classe estáticos usando a palavra-chave `static` antes do tipo de retorno do membro, conforme mostrado no exemplo a seguir:

C#

```
public class Automobile
{
    public static int NumberOfWheels = 4;

    public static int SizeOfGasTank
```

```
    {
        get
        {
            return 15;
        }
    }

    public static void Drive() { }

    public static event EventType? RunOutOfGas;

    // Other non-static fields and properties...
}
```

Os membros estáticos são inicializados antes que o membro estático seja acessado pela primeira vez e antes que o construtor estático, se houver um, seja chamado. Para acessar um membro de classe estático, use o nome da classe em vez de um nome de variável para especificar o local do membro, conforme mostrado no exemplo a seguir:

C#

```
Automobile.Drive();
int i = Automobile.NumberOfWheels;
```

Se a classe contiver campos estáticos, forneça um construtor estático que os inicialize quando a classe for carregada.

Uma chamada para um método estático gera uma instrução de chamada em CIL (linguagem intermediária comum), enquanto uma chamada para um método de instância gera uma instrução `callvirt`, que também verifica referências de objeto nulos. No entanto, na maioria das vezes, a diferença de desempenho entre os dois não é significativa.

Especificação da linguagem C#

Para obter mais informações, consulte classes estáticas, membros estáticos e de instância e construtores estáticos na Especificação da Linguagem C#. A especificação de idioma é a fonte definitiva para a sintaxe e o uso de C#.

Consulte também

- estático
- Classes
- classe
- Construtores estáticos

- Construtores de instância

Modificadores de acesso (Guia de Programação em C#)

Artigo • 20/08/2024

Todos os tipos e membros de tipo têm um nível de acessibilidade. O nível de acessibilidade controla se eles podem ser usados em outro código no assembly ou em outros assemblies. Um assembly é um *.dll* ou *.exe* criado ao selecionar um ou mais arquivos *.cs* em uma única compilação. Use os modificadores de acesso a seguir para especificar a acessibilidade de um tipo ou membro quando você o declarar:

- público: O código em qualquer assembly pode acessar esse tipo ou membro. O nível de acessibilidade do tipo que contém controla o nível de acessibilidade dos membros públicos do tipo.
- privado: Somente o código declarado no mesmo `class` ou `struct` pode acessar este membro.
- protegido: Somente código no mesmo `class` ou em um `class` derivado pode acessar esse tipo ou membro.
- interno: Somente código no mesmo assembly pode acessar esse tipo ou membro.
- interno protegido: Somente código no mesmo assembly *ou* em uma classe derivada em outro assembly pode acessar esse tipo ou membro.
- privado protegido: Somente código no mesmo assembly *e* na mesma classe ou em uma classe derivada pode acessar o tipo ou membro.
- arquivo: somente o código no mesmo arquivo pode acessar o tipo ou membro.

O modificador record em um tipo faz com que o compilador sintetize membros extras. O modificador `record` não afeta a acessibilidade padrão para um `record class` ou um `record struct`.

Tabela de resumo

Expandir a tabela

Local do chamador	`public`	`protected internal`	`protected`	`internal`	`private protected`	`private`	`file`
Dentro do arquivo	✔	✔	✔	✔	✔	✔	✔
Dentro da classe	✔	✔	✔	✔	✔	✔	✘

Local do chamador	`public`	`protected internal`	`protected`	`internal`	`private protected`	`private`	`file`
Classe derivada (mesmo assembly)	✔	✔	✔	✔	✔	✘	✘
Classe não derivada (mesmo assembly)	✔	✔	✘	✔	✘	✘	✘
Classe derivada (assembly diferente)	✔	✔	✔	✘	✘	✘	✘
Classe não derivada (assembly diferente)	✔	✘	✘	✘	✘	✘	✘

Os exemplos a seguir demonstram como especificar modificadores de acesso em um tipo e membro:

C#

```
public class Bicycle
{
    public void Pedal() { }
}
```

Nem todos os modificadores de acesso são válidos para todos os tipos ou membros em todos os contextos. Em alguns casos, a acessibilidade do tipo que contém restringe a acessibilidade dos seus membros.

Várias declarações de uma classe parcial ou membro parcial devem ter a mesma acessibilidade. Se uma declaração da classe ou membro parcial não incluir um modificador de acesso, as outras declarações não poderão declarar um modificador de acesso. O compilador gera um erro se múltiplas declarações para a classe ou método parcial declararem acessibilidades diferentes.

Acessibilidade de classe e estrutura

Classes e estruturas declaradas diretamente em um namespace (não estão aninhadas em outras classes ou structs) podem ter acesso `public`, `internal` ou `file`. `internal` é o

padrão, se nenhum modificador de acesso for especificado.

Os membros de struct, incluindo classes e structs aninhados, podem ser declarados como `public`, `internal` ou `private`. Os membros de classe, incluindo classes e structs aninhados, podem ser `public`, `protected internal`, `protected`, `internal`, `private protected` ou `private`. Os membros de classe e struct, incluindo classes e structs aninhados, têm acesso `private` por padrão.

As classes derivadas não podem ter maior acessibilidade do que seus tipos base. Você não pode declarar uma classe pública `B` derivada de uma classe interna `A`. Se permitido, teria o efeito de tornar `A` público, pois todos os membros `protected` ou `internal` de `A` são acessíveis na classe derivada.

Você pode permitir que outros assemblies específicos acessem os tipos internos usando o `InternalsVisibleToAttribute`. Para obter mais informações, consulte Assemblies amigáveis.

Outros tipos

As interfaces declaradas diretamente em um namespace podem ser `public` ou `internal` e, assim como classes e structs, o padrão das interfaces é o acesso `internal`. Membros de interface são `public` por padrão, pois a finalidade de uma interface é permitir que outros tipos acessem uma classe ou um struct. As declarações de membros da interface podem incluir qualquer modificador de acesso. Você usa modificadores de acesso em `interface` membros para fornecer uma implementação comum necessária para todos os implementadores de uma interface.

Um tipo delegate declarado diretamente em um namespace tem acesso `internal` por padrão.

Para obter mais informações sobre modificadores de acesso, consulte a página Níveis de acessibilidade.

Acessibilidade de membro

Membros de um `class` ou `struct` (incluindo classes e estruturas aninhadas) podem ser declarados com qualquer um dos seis tipos de acesso. Os membros de struct não podem ser declarados como `protected`, `protected internal` ou `private protected`, pois os structs não permitem herança.

Normalmente, a acessibilidade de um membro não é maior que a acessibilidade do tipo que o contém. No entanto, um membro `public` de uma classe `internal` poderá ser acessível de fora do assembly se o membro implementar métodos de interface ou substituir métodos virtuais definidos em uma classe base pública.

O tipo de qualquer campo, propriedade ou evento do membro deve ser pelo menos tão acessível quanto o próprio membro. Da mesma forma, o tipo de retorno e os tipos de parâmetro de qualquer método, indexador ou delegado devem ser pelo menos tão acessíveis quanto o próprio membro. Por exemplo, você não pode ter um método `public`, `M`, que retorna uma classe `C`, a menos que `C` também seja `public`. Da mesma forma, você não pode ter uma propriedade `protected` do tipo `A`, se `A` for declarada como `private`.

Os operadores definidos pelo usuário sempre devem ser declarados como `public` e `static`. Para obter mais informações, consulte Sobrecarga de operador.

Para definir o nível de acesso para um membro `class` ou `struct`, adicione a palavra-chave apropriada à declaração do membro, conforme mostrado no exemplo a seguir.

C#

```
// public class:
public class Tricycle
{
    // protected method:
    protected void Pedal() { }

    // private field:
    private int _wheels = 3;

    // protected internal property:
    protected internal int Wheels
    {
        get { return _wheels; }
    }
}
```

Os finalizadores não podem ter modificadores de acessibilidade. Os membros de um `enum` tipo são sempre `public` e nenhum modificador de acesso pode ser aplicado.

O modificador de acesso `file` é permitido apenas em declarações de tipo de nível superior (não aninhados).

Especificação da linguagem C#

Para obter mais informações, consulte a Especificação da linguagem C#. A especificação da linguagem é a fonte definitiva para a sintaxe e o uso de C#.

Confira também

- Especificar a ordem do modificador (regra de estilo IDE0036)
- Sistema de tipos do C#
- Interfaces
- Níveis de acessibilidade
- private
- público
- interno
- protected
- internos protegidos
- privado protegido
- sealed
- class
- struct
- interface
- Tipos anônimos

Campos (Guia de Programação em C#)

19/06/2025

Um *campo* é uma variável de qualquer tipo que é declarado diretamente em uma classe ou struct. Os campos são *membros* do tipo que os contém.

Uma classe ou struct pode ter campos de instância, campos estáticos ou ambos. Os campos de instância são específicos para uma instância de um tipo. Se você tiver uma classe `T`, com um campo `F` de instância, poderá criar dois objetos do tipo `T` e modificar o valor de `F` cada objeto sem afetar o valor no outro objeto. Por outro lado, um campo estático pertence ao próprio tipo e é compartilhado entre todas as instâncias desse tipo. Você só pode acessar o campo estático usando o nome do tipo. Se você acessar o campo estático por um nome de instância, receberá um erro de tempo de compilação CS0176 .

Em geral, você deve declarar `private` ou `protected` acessibilidade para campos. Os dados que seu tipo expõe ao código do cliente devem ser fornecidos por meio de métodos, propriedades e indexadores. Usando esses constructos para acesso indireto a campos internos, você pode se proteger contra valores de entrada inválidos. Um campo privado que armazena os dados expostos por uma propriedade pública é chamado de repositório de *backup* ou *campo de backup*. Você pode declarar `public` campos, mas não pode impedir que o código que usa seu tipo configure esse campo como um valor inválido ou altere os dados de um objeto.

Os campos normalmente armazenam os dados que devem ser acessíveis a mais de um método de tipo e devem ser armazenados por mais tempo do que o tempo de vida de qualquer método único. Por exemplo, um tipo que representa uma data de calendário pode ter três campos inteiros: um para o mês, um para o dia e outro para o ano. Variáveis que não são usadas fora do escopo de um único método devem ser *declaradas como variáveis locais* dentro do próprio corpo do método.

Os campos são declarados no bloco de classe ou struct especificando o nível de acesso, seguido pelo tipo, seguido pelo nome do campo. Por exemplo:

C#

```
public class CalendarEntry
{

    // private field (Located near wrapping "Date" property).
    private DateTime _date;

    // Public property exposes _date field safely.
    public DateTime Date
    {
        get
```

```
        {
            return _date;
        }
        set
        {
            // Set some reasonable boundaries for likely birth dates.
            if (value.Year > 1900 && value.Year <= DateTime.Today.Year)
            {
                _date = value;
            }
            else
            {
                throw new ArgumentOutOfRangeException("Date");
            }
        }
    }

    // public field (Generally not recommended).
    public string? Day;

    // Public method also exposes _date field safely.
    // Example call: birthday.SetDate("1975, 6, 30");
    public void SetDate(string dateString)
    {
        DateTime dt = Convert.ToDateTime(dateString);

        // Set some reasonable boundaries for likely birth dates.
        if (dt.Year > 1900 && dt.Year <= DateTime.Today.Year)
        {
            _date = dt;
        }
        else
        {
            throw new ArgumentOutOfRangeException("dateString");
        }
    }

    public TimeSpan GetTimeSpan(string dateString)
    {
        DateTime dt = Convert.ToDateTime(dateString);

        if (dt.Ticks < _date.Ticks)
        {
            return _date - dt;
        }
        else
        {
            throw new ArgumentOutOfRangeException("dateString");
        }
    }
}
```

Para acessar um campo em uma instância, adicione um período após o nome da instância, seguido pelo nome do campo, como em `instancename._fieldName`. Por exemplo:

```
C#

CalendarEntry birthday = new CalendarEntry();
birthday.Day = "Saturday";
```

Um campo pode receber um valor inicial usando o operador de atribuição quando o campo é declarado. Para atribuir automaticamente o `Day` campo a `"Monday"`, por exemplo, você declararia `Day` como no exemplo a seguir:

```
C#

public class CalendarDateWithInitialization
{
    public string Day = "Monday";
    //...
}
```

Os campos são inicializados imediatamente antes que o construtor da instância de objeto seja chamado. Se o construtor atribuir um valor a um campo, isso substituirá qualquer valor anteriormente definido na declaração do campo. Para obter mais informações, veja Usando construtores.

> ⓘ **Observação**
>
> Um inicializador de campo não pode se referir a outros campos de instância.

Os campos podem ser marcados como public, , private, protected, internal, protected internalou private protected. Esses modificadores de acesso definem como os usuários do tipo podem acessar os campos. Para obter mais informações, consulte Modificadores de Acesso.

Opcionalmente, um campo pode ser declarado static. Os campos estáticos estão disponíveis para chamadores a qualquer momento, mesmo que nenhuma instância do tipo exista. Para obter mais informações, consulte Classes estáticas e membros de classe estática.

Um campo pode ser declarado readonly. Um campo somente leitura só pode ser atribuído a um valor durante a inicialização ou em um construtor. Um `static readonly` campo é semelhante a uma constante, exceto que o compilador C# não tem acesso ao valor de um campo somente leitura estático no momento da compilação, apenas no momento da execução. Para obter mais informações, consulte Constantes.

Um campo pode ser declarado required. Um campo necessário deve ser inicializado pelo construtor ou por um inicializador de objeto quando um objeto é criado. Adicione o System.Diagnostics.CodeAnalysis.SetsRequiredMembersAttribute atributo a qualquer declaração de construtor que inicialize todos os membros necessários.

O `required` modificador não pode ser combinado com o `readonly` modificador no mesmo campo. No entanto, a propriedade pode ser somente `required` e `init`.

A partir do C# 12, os parâmetros do construtor primário são uma alternativa para declarar campos. Quando seu tipo tiver dependências que devem ser fornecidas na inicialização, você poderá criar um construtor primário que forneça essas dependências. Esses parâmetros podem ser capturados e usados no lugar de campos declarados em seus tipos. No caso de tipos record, os parâmetros do construtor primário são exibidos como propriedades públicas.

Especificação da linguagem C#

Para obter mais informações, consulte a Especificação da Linguagem C# . A especificação de idioma é a fonte definitiva para a sintaxe e o uso de C#.

Consulte também

- Sistema de tipos do C#
- Usando construtores
- Herança
- Modificadores de acesso
- Classes abstratas e seladas e membros da classe

Constantes (Guia de Programação em C#)

19/06/2025

Constantes são valores imutáveis que são conhecidos em tempo de compilação e não mudam durante a vida útil do programa. As constantes são declaradas com o modificador const . Somente os tipos internos do C# podem ser declarados como `const`. Constantes de tipo de referência, exceto String, só podem ser inicializadas com um valor nulo. Tipos definidos pelo usuário, incluindo classes, structs e matrizes, não podem ser `const`. Use o modificador readonly para criar uma classe, struct ou matriz que é inicializada uma vez em tempo de execução (por exemplo, em um construtor) e depois disso não pode ser alterada.

O C# não dá suporte `const` a métodos, propriedades ou eventos.

O tipo de enumeração permite que você defina constantes nomeadas para tipos internos integrais (por exemplo `int`, `uint`, `long` e assim por diante). Para obter mais informações, consulte enum.

As constantes devem ser inicializadas conforme são declaradas. Por exemplo:

C#

```
class Calendar1
{
    public const int Months = 12;
}
```

Neste exemplo, a constante `Months` é sempre 12 e não pode ser alterada até mesmo pela própria classe. Na verdade, quando o compilador encontra um identificador constante no código-fonte C# (por exemplo, `Months`), ele substitui o valor literal diretamente no código il (linguagem intermediária) que ele produz. Como não há nenhum endereço variável associado a uma constante em tempo de execução, `const` os campos não podem ser passados por referência e não podem aparecer como um valor l em uma expressão.

> **Observação**
>
> Tenha cuidado ao se referir a valores constantes definidos em outro código, como DLLs. Se uma nova versão da DLL definir um novo valor para a constante, seu programa ainda manterá o valor literal antigo até que ele seja recompilado em relação à nova versão.

Várias constantes do mesmo tipo podem ser declaradas ao mesmo tempo, por exemplo:

C#

```
class Calendar2
{
    public const int Months = 12, Weeks = 52, Days = 365;
}
```

A expressão usada para inicializar uma constante pode referir-se a outra constante se ela não criar uma referência circular. Por exemplo:

```
C#

class Calendar3
{
    public const int Months = 12;
    public const int Weeks = 52;
    public const int Days = 365;

    public const double DaysPerWeek = (double) Days / (double) Weeks;
    public const double DaysPerMonth = (double) Days / (double) Months;
}
```

As constantes podem ser marcadas como públicas, privadas, protegidas, internas, protegidas internas ou privadas protegidas. Esses modificadores de acesso definem como os usuários da classe podem acessar a constante. Para obter mais informações, consulte Modificadores de Acesso.

As constantes são acessadas como se fossem campos estáticos porque o valor da constante é o mesmo para todas as instâncias do tipo. Você não usa a palavra-chave `static` para declará-las. Expressões que não estão na classe que define a constante devem usar o nome da classe, um período e o nome da constante para acessar a constante. Por exemplo:

```
C#

int birthstones = Calendar.Months;
```

Especificação da linguagem C#

Para obter mais informações, consulte a Especificação da Linguagem C# . A especificação de idioma é a fonte definitiva para a sintaxe e o uso de C#.

Consulte também

- Propriedades
- tipos de

- readonly
- Imutabilidade em C#: Parte Um - Tipos de Imutabilidade

Como definir propriedades abstract (Guia de programação em C#)

Artigo • 07/04/2023

O exemplo a seguir mostra como definir propriedades abstract. Uma declaração de propriedade abstract não fornece uma implementação dos acessadores da propriedade – ela declara que a classe dá suporte às propriedades, mas deixa a implementação do acessador para classes derivadas. O exemplo a seguir demonstra como implementar as propriedades abstract herdadas de uma classe base.

Esse exemplo consiste em três arquivos, cada um deles é compilado individualmente e seu assembly resultante é referenciado pela próxima compilação:

- abstractshape.cs: a classe `Shape` que contém uma propriedade abstract `Area`.
- shapes.cs: as subclasses da classe `Shape`.
- shapetest.cs: um programa de teste para exibir as áreas de alguns objetos derivados de `Shape`.

Para compilar o exemplo, use o comando a seguir:

```
csc abstractshape.cs shapes.cs shapetest.cs
```

Isso criará o arquivo executável shapetest.exe.

Exemplos

Esse arquivo declara a classe `Shape` que contém a propriedade `Area` do tipo `double`.

C#

```
// compile with: csc -target:library abstractshape.cs
public abstract class Shape
{
    private string name;

    public Shape(string s)
    {
        // calling the set accessor of the Id property.
        Id = s;
    }

    public string Id
    {
```

```
        get
        {
            return name;
        }

        set
        {
            name = value;
        }
    }

    // Area is a read-only property - only a get accessor is needed:
    public abstract double Area
    {
        get;
    }

    public override string ToString()
    {
        return $"{Id} Area = {Area:F2}";
    }
}
```

- Os modificadores da propriedade são colocados na própria declaração de propriedade. Por exemplo:

  ```
  C#

  public abstract double Area
  ```

- Ao declarar uma propriedade abstract (como `Area` neste exemplo), você simplesmente indica quais acessadores de propriedade estão disponíveis, mas não os implementa. Neste exemplo, apenas um acessador get está disponível, assim, a propriedade é somente leitura.

O código a seguir mostra três subclasses de `Shape` e como elas substituem a propriedade `Area` para fornecer sua própria implementação.

```
C#

// compile with: csc -target:library -reference:abstractshape.dll shapes.cs
public class Square : Shape
{
    private int side;

    public Square(int side, string id)
        : base(id)
    {
        this.side = side;
```

```
    }

    public override double Area
    {
        get
        {
            // Given the side, return the area of a square:
            return side * side;
        }
    }
}

public class Circle : Shape
{
    private int radius;

    public Circle(int radius, string id)
        : base(id)
    {
        this.radius = radius;
    }

    public override double Area
    {
        get
        {
            // Given the radius, return the area of a circle:
            return radius * radius * System.Math.PI;
        }
    }
}

public class Rectangle : Shape
{
    private int width;
    private int height;

    public Rectangle(int width, int height, string id)
        : base(id)
    {
        this.width = width;
        this.height = height;
    }

    public override double Area
    {
        get
        {
            // Given the width and height, return the area of a rectangle:
            return width * height;
        }
    }
}
```

O código a seguir mostra um programa de teste que cria uma quantidade de objetos derivados de `Shape` e imprime suas áreas.

C#

```
// compile with: csc -reference:abstractshape.dll;shapes.dll shapetest.cs
class TestClass
{
    static void Main()
    {
        Shape[] shapes =
        {
            new Square(5, "Square #1"),
            new Circle(3, "Circle #1"),
            new Rectangle( 4, 5, "Rectangle #1")
        };

        System.Console.WriteLine("Shapes Collection");
        foreach (Shape s in shapes)
        {
            System.Console.WriteLine(s);
        }
    }
}
/* Output:
    Shapes Collection
    Square #1 Area = 25.00
    Circle #1 Area = 28.27
    Rectangle #1 Area = 20.00
*/
```

Confira também

- Guia de Programação em C#
- O sistema do tipo C#
- Classes e membros de classes abstract e sealed
- Propriedades

Como definir constantes em C#

Artigo • 07/04/2023

As constantes são campos cujos valores são definidos em tempo de compilação e nunca podem ser alterados. Use constantes para fornecer nomes significativos em vez de literais numéricos ("números mágicos") a valores especiais.

> ⓘ **Observação**
>
> No C#, a diretiva de pré-processador **#define** não pode ser utilizada para definir constantes da mesma maneira que é normalmente usada no C e no C++.

Para definir valores de constantes de tipos integrais (`int`, `byte` e assim por diante), use um tipo enumerado. Para obter mais informações, consulte enum.

Para definir constantes não integrais, uma abordagem é agrupá-las em uma única classe estática de nome `Constants`. Isso exigirá que todas as referências às constantes sejam precedidas com o nome de classe, conforme mostrado no exemplo a seguir.

Exemplo

C#

```
static class Constants
{
    public const double Pi = 3.14159;
    public const int SpeedOfLight = 300000; // km per sec.
}

class Program
{
    static void Main()
    {
        double radius = 5.3;
        double area = Constants.Pi * (radius * radius);
        int secsFromSun = 149476000 / Constants.SpeedOfLight; // in km
        Console.WriteLine(secsFromSun);
    }
}
```

O uso do qualificador de nome de classe ajuda a garantir que você e outras pessoas que usam a constante entendam que ele é constante e não pode ser modificado.

Confira também

- Sistema de tipos do C#

Propriedades (Guia de Programação em C#)

Artigo • 14/11/2024

Uma *propriedade* é um membro que oferece um mecanismo flexível para ler, gravar ou calcular o valor de um campo de dados. As propriedades aparecem como membros de dados públicos, mas são implementadas como métodos especiais chamados *acessadores*. Esse recurso permite que os chamadores acessem dados facilmente e ainda ajuda a promover a segurança e a flexibilidade dos dados. A sintaxe para propriedades é uma extensão natural para os campos. Um campo define um local de armazenamento:

C#

```
public class Person
{
    public string? FirstName;

    // Omitted for brevity.
}
```

Propriedades implementadas automaticamente

Uma definição de propriedade contém declarações para um acessador `get` e `set` que recupera e atribui o valor dessa propriedade:

C#

```
public class Person
{
    public string? FirstName { get; set; }

    // Omitted for brevity.
}
```

O exemplo anterior mostra uma propriedade *implementada* automaticamente. O compilador gera um campo de suporte oculto para a propriedade. O compilador também implementa o corpo dos acessadores `get` e `set`. Todos os atributos são aplicados à propriedade implementada automaticamente. Você pode aplicar o atributo ao campo de suporte gerado pelo compilador especificando a tag `field:` no atributo.

Você pode inicializar uma propriedade para um valor diferente do padrão definindo um valor após a chave de fechamento da propriedade. Talvez você prefira que o valor inicial

para a propriedade `FirstName` seja a cadeia de caracteres vazia em vez de `null`. Você especificaria isso conforme mostrado no código a seguir:

C#

```
public class Person
{
    public string FirstName { get; set; } = string.Empty;

    // Omitted for brevity.
}
```

Propriedades com suporte de campo

No C# 13, você pode adicionar validação ou outra lógica no acessador de uma propriedade usando o recurso de visualização de field palavra-chave. A `field` palavra-chave acessa o campo de suporte sintetizado do compilador para uma propriedade. Ele permite que você escreva um acessador de propriedade sem declarar explicitamente um campo de suporte separado.

C#

```
public class Person
{
    public string? FirstName
    {
        get;
        set => field = value.Trim();
    }

    // Omitted for brevity.
}
```

> ⓘ **Importante**
>
> A `field` palavra-chave é um recurso de visualização no C# 13. Você deve estar usando o .NET 9 e definir seu `<LangVersion>` elemento como `preview` no arquivo de projeto para usar a `field` palavra-chave contextual.
>
> Você deve ter cuidado ao usar o recurso de `field` palavra-chave em uma classe que tenha um campo chamado `field`. A nova `field` palavra-chave sombreia um campo nomeado `field` no escopo de um acessador de propriedade. Você pode alterar o nome da `field` variável ou usar o `@` token para fazer referência ao `field`

identificador como `@field`. Você pode saber mais lendo a especificação do recurso para a field palavra-chave.

Propriedades obrigatórias

O exemplo anterior permite que um chamador crie um `Person` usando o construtor padrão, sem definir a propriedade `FirstName`. A propriedade alterou o tipo para uma cadeia de caracteres *que permite valor nulo*. A partir do C# 11, você pode *exigir* que os chamadores definam uma propriedade:

C#

```
public class Person
{
    public Person() { }

    [SetsRequiredMembers]
    public Person(string firstName) => FirstName = firstName;

    public required string FirstName { get; init; }

    // Omitted for brevity.
}
```

O código anterior faz duas alterações na classe `Person`. Primeiro, a declaração de propriedade `FirstName` inclui o modificador `required`. Isso significa que qualquer código que crie um novo `Person` deve definir essa propriedade usando um inicializador de objeto. Em segundo lugar, o construtor que usa um parâmetro `firstName` tem o atributo System.Diagnostics.CodeAnalysis.SetsRequiredMembersAttribute. Esse atributo informa ao compilador que esse construtor define *todos* `required` os membros. Os chamadores que usam esse construtor não precisam definir propriedades `required` com um inicializador de objeto.

> **ⓘ Importante**
>
> Não confunda `required` com *não anulável*. É válido definir uma propriedade `required` como `null` ou `default`. Se o tipo for não anulável, como `string` nesses exemplos, o compilador emitirá um aviso.

C#

```
var aPerson = new Person("John");
aPerson = new Person{ FirstName = "John"};
// Error CS9035: Required member `Person.FirstName` must be set:
//aPerson2 = new Person();
```

Definições de corpo de expressão

Os acessadores de propriedade geralmente consistem em instruções de linha única. Os acessadores atribuem ou retornam o resultado de uma expressão. Você pode implementar essas propriedades como membros aptos para expressão. As definições de corpo da expressão consistem no token `=>` seguido pela expressão à qual atribuir ou recuperar da propriedade.

Propriedades somente leitura podem implementar o acessador `get` como um membro apto para expressão. O exemplo a seguir implementa a propriedade `Name` somente leitura como um membro apto para expressão:

C#

```
public class Person
{
    public Person() { }

    [SetsRequiredMembers]
    public Person(string firstName, string lastName)
    {
        FirstName = firstName;
        LastName = lastName;
    }

    public required string FirstName { get; init; }
    public required string LastName { get; init; }

    public string Name => $"{FirstName} {LastName}";

    // Omitted for brevity.
}
```

A propriedade `Name` é uma propriedade computada. Não há campo de apoio para `Name`. A propriedade calcula isso toda vez.

Controle de acesso

Os exemplos anteriores mostraram propriedades de leitura/gravação. Você também pode criar propriedades somente leitura ou dar acessibilidade diferente aos acessadores get e set. Suponha que sua `Person` classe só deva `FirstName` habilitar a alteração do valor da propriedade de outros métodos na classe. Você pode dar acessibilidade ao acessador `private` definido em vez de `internal` ou `public`:

```
C#

public class Person
{
    public string? FirstName { get; private set; }

    // Omitted for brevity.
}
```

A propriedade `FirstName` pode ser lida em qualquer código, mas só pode ser atribuída do código na classe `Person`.

Você pode adicionar qualquer modificador de acesso restritivo aos acessadores get ou set. Um modificador de acesso em um acessador individual deve ser mais restritivo do que o acesso da propriedade. O código anterior é legal porque a propriedade `FirstName` é `public`, mas o acessador set é `private`. Você não poderia declarar uma propriedade `private` com um acessador `public`. As declarações de propriedade também podem ser declaradas `protected`, `internal`, `protected internal` ou até mesmo `private`.

Há dois modificadores de acesso especiais para acessadores `set`:

- Um acessador `set` pode ter `init` como seu modificador de acesso. Esse acessador `set` pode ser chamado somente de um inicializador de objeto ou dos construtores do tipo. É mais restritivo do que `private` no acessador `set`.
- Uma propriedade implementada automaticamente pode declarar um `get` acessador sem um `set` acessador. Nesse caso, o compilador permite que o acessador `set` seja chamado somente dos construtores do tipo. É mais restritivo do que o acessador `init` no acessador `set`.

Modifique a classe `Person` da seguinte maneira:

```
C#

public class Person
{
    public Person(string firstName) => FirstName = firstName;
```

```
    public string FirstName { get; }

    // Omitted for brevity.
}
```

O exemplo anterior requer que os chamadores usem o construtor que inclui o parâmetro `FirstName`. Os chamadores não podem usar inicializadores de objeto para atribuir um valor à propriedade. Para dar suporte a inicializadores, você pode transformar o `set` em um `init`, conforme mostrado no seguinte código:

C#

```
public class Person
{
    public Person() { }
    public Person(string firstName) => FirstName = firstName;

    public string? FirstName { get; init; }

    // Omitted for brevity.
}
```

Esses modificadores são frequentemente usados com o modificador `required` para forçar a inicialização adequada.

Propriedades com campos de suporte

Combine o conceito de uma propriedade computada com um campo privado e crie uma *propriedade avaliada armazenada em cache*. Por exemplo, atualize a propriedade `FullName` para que a formatação da string ocorra no primeiro acesso:

C#

```
public class Person
{
    public Person() { }

    [SetsRequiredMembers]
    public Person(string firstName, string lastName)
    {
        FirstName = firstName;
        LastName = lastName;
    }

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

```
    private string? _fullName;
    public string FullName
    {
        get
        {
            if (_fullName is null)
                _fullName = $"{FirstName} {LastName}";
            return _fullName;
        }
    }
}
```

Essa implementação funciona porque as propriedades `FirstName` e `LastName` são somente leitura. As pessoas podem mudar o nome. A atualização das propriedades `FirstName` e `LastName` para permitir acessadores `set` exige que você invalide qualquer valor armazenado em cache para `fullName`. Modifique os acessadores `set` das propriedades `FirstName` e `LastName` para que o campo `fullName` seja calculado novamente:

C#

```
public class Person
{
    private string? _firstName;
    public string? FirstName
    {
        get => _firstName;
        set
        {
            _firstName = value;
            _fullName = null;
        }
    }

    private string? _lastName;
    public string? LastName
    {
        get => _lastName;
        set
        {
            _lastName = value;
            _fullName = null;
        }
    }

    private string? _fullName;
    public string FullName
    {
        get
        {
            if (_fullName is null)
```

```
                _fullName = $"{FirstName} {LastName}";
            return _fullName;
        }
    }
}
```

Esta versão final avalia a propriedade `FullName` apenas quando necessário. Se a versão calculada anteriormente é válida, ela é usada. Caso contrário, o cálculo atualizará o valor armazenado em cache. Os desenvolvedores que usam essa classe não precisam saber dos detalhes da implementação. Nenhuma dessas alterações internas afetam o uso do objeto Person.

A partir do C# 13, você pode criar partial propriedades em classes `partial`. A declaração de implementação de uma `partial` propriedade não pode ser uma propriedade implementada automaticamente. Uma propriedade implementada automaticamente usa a mesma sintaxe que uma declaração de propriedade parcial de declaração.

Propriedades

As propriedades são uma forma de campos inteligentes em uma classe ou objeto. De fora do objeto, elas parecem como campos no objeto. No entanto, as propriedades podem ser implementadas usando a paleta completa de funcionalidades do C#. Você pode fornecer validação, acessibilidade diferente, avaliação lenta ou quaisquer requisitos necessários aos seus cenários.

- Propriedades simples que não exigem nenhum código de acesso personalizado podem ser implementadas como definições de corpo de expressão ou como propriedades implementadas automaticamente.
- As propriedades permitem que uma classe exponha uma forma pública de obter e definir valores, enquanto oculta o código de implementação ou de verificação.
- Um acessador de propriedade get é usado para retornar o valor da propriedade e um acessador de propriedade set é usado para atribuir um novo valor. Um acessador de propriedade init é usado para atribuir um novo valor somente durante a construção de objeto. Esses acessadores podem ter diferentes níveis de acesso. Para obter mais informações, consulte Restringindo a acessibilidade aos acessadores.
- A palavra-chave value é usada para definir o valor que o acessor `set` ou `init` está atribuindo.
- As propriedades podem ser de *leitura/gravação* (elas têm um acessador `get` e `set`), *somente leitura* (elas têm um acessador `get`, mas nenhum `set`) ou *somente*

gravação (elas têm um acessador `set`, mas nenhum `get`). As propriedades somente gravação são raras.

Especificação da Linguagem C#

Para obter mais informações, veja Propriedades na Especificação da Linguagem C#. A especificação da linguagem é a fonte definitiva para a sintaxe e o uso de C#.

Confira também

- Indexadores
- Palavra-chave init
- Palavra-chave get
- Palavra-chave set

Usando propriedades (Guia de Programação em C#)

Artigo • 14/11/2024

As propriedades combinam aspectos de métodos e campos. Para o usuário de um objeto, uma propriedade parece como um campo; o acesso à propriedade exige a mesma sintaxe. Para o implementador de uma classe, uma propriedade é um ou dois blocos de código, representando um acessador get e/ou um ou acessador set OU init. O bloco de código do acessador `get` é executado quando a propriedade é lida; o bloco de código do acessador `set` ou `init` é executado quando a propriedade recebe um valor. Uma propriedade sem um acessador `set` é considerada como somente leitura. Uma propriedade sem um acessador `get` é considerada como somente gravação. Uma propriedade que tem os dois acessadores é leitura/gravação. Você pode usar um acessador `init` em vez de um acessador `set` para permitir que a propriedade seja definida como parte da inicialização do objeto, mas de outra forma torná-la somente leitura.

Diferentemente dos campos, as propriedades não são classificadas como variáveis. Portanto, você não pode passar uma propriedade como um parâmetro ref ou out.

As propriedades têm muitos usos:

- Elas podem validar dados antes de permitir uma alteração.
- Elas podem expor dados de forma transparente em uma classe em que esses dados são recuperados de alguma outra fonte, como um banco de dados.
- Elas podem executar uma ação quando os dados são alterados, como gerar um evento ou alterar o valor de outros campos.

As propriedades são declaradas no bloco de classe, especificando o nível de acesso do campo, seguido pelo tipo da propriedade, pelo nome da propriedade e por um bloco de código que declara um acessador `get` e/ou um acessador `set`. Por exemplo:

C#

```
public class Date
{
    private int _month = 7;  // Backing store

    public int Month
    {
        get => _month;
        set
        {
```

```
            if ((value > 0) && (value < 13))
            {
                _month = value;
            }
        }
    }
}
```

Neste exemplo, `Month` é declarado como uma propriedade de maneira que o acessador `set` possa garantir que o valor `Month` esteja definido entre 1 e 12. A propriedade `Month` usa um campo particular para rastrear o valor real. O local real dos dados de uma propriedade é frequentemente chamado de "repositório de backup" da propriedade. É comum que as propriedades usem campos privados como repositório de backup. O campo é marcado como particular para garantir que ele só pode ser alterado ao chamar a propriedade. Para obter mais informações sobre as restrições de acesso público e particular, consulte Modificadores de acesso. As propriedades implementadas automaticamente fornecem sintaxe simplificada para declarações de propriedade simples. Para obter mais informações, consulte Propriedades implementadas automaticamente.

A partir do C# 13, você pode usar propriedades com suporte de campo para adicionar validação ao `set` acessador de uma propriedade implementada automaticamente, conforme mostrado no exemplo a seguir:

C#

```
public class DateExample
{
    public int Month
    {
        get;
        set
        {
            if ((value > 0) && (value < 13))
            {
                field = value;
            }
        }
    }
}
```

ⓘ Importante

A `field` palavra-chave é um recurso de visualização no C# 13. Você deve estar usando o .NET 9 e definir seu `<LangVersion>` elemento como `preview` no arquivo

de projeto para usar a `field` palavra-chave contextual.

Você deve ter cuidado ao usar o recurso de `field` palavra-chave em uma classe que tenha um campo chamado `field`. A nova `field` palavra-chave sombreia um campo nomeado `field` no escopo de um acessador de propriedade. Você pode alterar o nome da `field` variável ou usar o `@` token para fazer referência ao `field` identificador como `@field`. Você pode saber mais lendo a especificação do recurso para a field palavra-chave.

O acessador get

O corpo do acessador `get` assemelha-se ao de um método. Ele deve retornar um valor do tipo de propriedade. O compilador C# e o compilador just-in-time (JIT) detectam padrões comuns para implementar o acessador `get`, e otimiza esses padrões. Por exemplo, um acessador `get` que retorna um campo sem executar qualquer computação provavelmente é otimizado para uma leitura de memória desse campo. As propriedades implementadas automaticamente seguem esse padrão e se beneficiam dessas otimizações. No entanto, um método acessador virtual `get` não pode ser embutido porque o compilador não sabe em tempo de compilação qual método pode realmente ser chamado em tempo de execução. O seguinte exemplo mostra um acessador `get` que retorna o valor de um campo particular `_name`:

C#

```
    class Employee
{
    private string _name;  // the name field
    public string Name => _name;     // the Name property
}
```

Quando você referencia a propriedade, exceto como o destino de uma atribuição, o acessador `get` é invocado para ler o valor da propriedade. Por exemplo:

C#

```
var employee= new Employee();
//...

System.Console.Write(employee.Name);  // the get accessor is invoked here
```

O acessador `get` deve ser um membro com corpo de expressão ou terminar em uma instrução return ou throw, e o controle não pode fluir para fora do corpo do acessador.

> ⚠ **Aviso**
>
> Geralmente, é um estilo de programação ruim alterar o estado do objeto usando o `get` acessador. Uma exceção a essa regra é uma *propriedade avaliada* lentamente, em que o valor de uma propriedade é calculado somente quando ela é acessada pela primeira vez.

O acessador `get` pode ser usado para retornar o valor do campo ou para calculá-lo e retorná-lo. Por exemplo:

```C#
class Manager
{
    private string _name;
    public string Name => _name != null ? _name : "NA";
}
```

No exemplo anterior, se você não atribuir um valor à propriedade `Name`, ela retornará o valor `NA`.

O acessador set

O acessador `set` é semelhante a um método cujo tipo de retorno é void. Ele usa uma parâmetro implícito chamado `value`, cujo tipo é o tipo da propriedade. O compilador e o compilador JIT também reconhecem padrões comuns para um acessador `set` ou `init`. Esses padrões comuns são otimizados, gravando diretamente a memória para o campo de backup. No exemplo a seguir, uma acessador `set` é adicionado à propriedade `Name`:

```C#
class Student
{
    private string _name;  // the name field
    public string Name    // the Name property
    {
        get => _name;
        set => _name = value;
```

```
    }
}
```

Quando você atribui um valor à propriedade, o acessador `set` é invocado por meio do uso de um argumento que fornece o novo valor. Por exemplo:

```C#
var student = new Student();
student.Name = "Joe";  // the set accessor is invoked here

System.Console.Write(student.Name);  // the get accessor is invoked here
```

É um erro usar o nome de parâmetro implícito `value`, para uma declaração de variável local em um acessador `set`.

O acessador init

O código para criar um acessador `init` é o mesmo que o código para criar um acessador `set`, exceto que você usa a palavra-chave `init` em vez de `set`. A diferença é que o acessador `init` só pode ser empregado no construtor ou usando um inicializador de objeto.

Comentários

As propriedades podem ser marcadas como `public`, `private`, `protected`, `internal`, `protected internal` ou `private protected`. Esses modificadores de acesso definem como os usuários da classe podem acessar a propriedade. Os acessadores `get` e `set` da mesma propriedade podem ter modificadores de acesso diferentes. Por exemplo, `get` pode ser `public` para permitir o acesso somente leitura de fora do tipo, e `set` pode ser `private` ou `protected`. Para obter mais informações, consulte Modificadores de Acesso.

Uma propriedade pode ser declarada como uma propriedade estática usando a palavra-chave `static`. As propriedades estáticas estão disponíveis para chamadores a qualquer momento, mesmo se não existir nenhuma instância da classe. Para obter mais informações, consulte Classes estáticas e membros de classes estáticas.

Uma propriedade pode ser marcada como uma propriedade virtual usando a palavra-chave virtual. As propriedades virtuais permitem que classes derivadas substituam o comportamento da propriedade, usando a palavra-chave override. Para obter mais informações sobre essas opções, consulte Herança.

Uma propriedade que substitui uma propriedade virtual também pode ser sealed, especificando que ela não é mais virtual para classes derivadas. Por fim, uma propriedade pode ser declarada abstract. As propriedades abstratas não definem nenhuma implementação na classe e as classes derivadas devem escrever sua própria implementação. Para obter mais informações sobre essas opções, consulte Classes e membros de classes abstract e sealed.

> ⓘ **Observação**
>
> É um erro usar um modificador virtual, abstract ou override em um acessador de uma propriedade static.

Exemplos

Este exemplo demonstra as propriedades instância, estática e somente leitura. Ele aceita o nome do funcionário digitado no teclado, incrementa `NumberOfEmployees` em 1 e exibe o nome e o número do funcionário.

C#

```
public class Employee
{
    public static int NumberOfEmployees;
    private static int _counter;
    private string _name;

    // A read-write instance property:
    public string Name
    {
        get => _name;
        set => _name = value;
    }

    // A read-only static property:
    public static int Counter => _counter;

    // A Constructor:
    public Employee() => _counter = ++NumberOfEmployees; // Calculate the
employee's number:
}
```

Exemplo de propriedade oculta

Este exemplo demonstra como acessar uma propriedade em uma classe base que está oculta por outra propriedade que tem o mesmo nome em uma classe derivada:

C#

```
public class Employee
{
    private string _name;
    public string Name
    {
        get => _name;
        set => _name = value;
    }
}

public class Manager : Employee
{
    private string _name;

    // Notice the use of the new modifier:
    public new string Name
    {
        get => _name;
        set => _name = value + ", Manager";
    }
}

class TestHiding
{
    public static void Test()
    {
        Manager m1 = new Manager();

        // Derived class property.
        m1.Name = "John";

        // Base class property.
        ((Employee)m1).Name = "Mary";

        System.Console.WriteLine("Name in the derived class is: {0}",
m1.Name);
        System.Console.WriteLine("Name in the base class is: {0}",
((Employee)m1).Name);
    }
}
/* Output:
    Name in the derived class is: John, Manager
    Name in the base class is: Mary
*/
```

A seguir estão os pontos importantes do exemplo anterior:

- A propriedade `Name` na classe derivada oculta a propriedade `Name` na classe base. Nesse caso, o modificador `new` é usado na declaração da propriedade na classe derivada:

  ```
  C#
  public new string Name
  ```

- A conversão `(Employee)` é usada para acessar a propriedade oculta na classe base:

  ```
  C#
  ((Employee)m1).Name = "Mary";
  ```

Para obter mais informações sobre como ocultar membros, consulte o Modificador new.

Exemplo de substituição de propriedade

Neste exemplo, duas classes, `Cube` e `Square`, implementam uma classe abstrata `Shape` e substituem sua propriedade `Area` abstrata. Observe o uso do modificador override nas propriedades. O programa aceita o lado como uma entrada e calcula as áreas para o cubo e o quadrado. Ele também aceita a área como uma entrada e calcula o lado correspondente para o cubo e o quadrado.

```
C#
abstract class Shape
{
    public abstract double Area
    {
        get;
        set;
    }
}

class Square : Shape
{
    public double side;

    //constructor
    public Square(double s) => side = s;

    public override double Area
    {
        get => side * side;
        set => side = System.Math.Sqrt(value);
```

```
    }
}

class Cube : Shape
{
    public double side;

    //constructor
    public Cube(double s) => side = s;

    public override double Area
    {
        get => 6 * side * side;
        set => side = System.Math.Sqrt(value / 6);
    }
}

class TestShapes
{
    static void Main()
    {
        // Input the side:
        System.Console.Write("Enter the side: ");
        double side = double.Parse(System.Console.ReadLine());

        // Compute the areas:
        Square s = new Square(side);
        Cube c = new Cube(side);

        // Display the results:
        System.Console.WriteLine("Area of the square = {0:F2}", s.Area);
        System.Console.WriteLine("Area of the cube = {0:F2}", c.Area);
        System.Console.WriteLine();

        // Input the area:
        System.Console.Write("Enter the area: ");
        double area = double.Parse(System.Console.ReadLine());

        // Compute the sides:
        s.Area = area;
        c.Area = area;

        // Display the results:
        System.Console.WriteLine("Side of the square = {0:F2}", s.side);
        System.Console.WriteLine("Side of the cube = {0:F2}", c.side);
    }
}
/* Example Output:
    Enter the side: 4
    Area of the square = 16.00
    Area of the cube = 96.00

    Enter the area: 24
    Side of the square = 4.90
```

```
    Side of the cube = 2.00
*/
```

Confira também

- Propriedades
- Propriedades de interface
- Propriedades implementadas automaticamente
- Propriedades parciais

Propriedades de interface (Guia de Programação em C#)

Artigo • 15/10/2024

As propriedades podem ser declaradas em uma interface. O exemplo a seguir declara um acessador de propriedade de interface:

C#

```
public interface ISampleInterface
{
    // Property declaration:
    string Name
    {
        get;
        set;
    }
}
```

As propriedades da interface normalmente não têm um corpo. Os acessadores indicam se a propriedade é leitura/gravação, somente leitura ou somente gravação. Ao contrário de classes e structs, declarar os acessadores sem um corpo não declara uma propriedade implementada automaticamente. Uma interface pode definir uma implementação padrão para membros, incluindo propriedades. Definir uma implementação padrão para uma propriedade em uma interface é raro porque as interfaces não podem definir campos de dados de instância.

Exemplo

Neste exemplo, a interface `IEmployee` tem uma propriedade de leitura/gravação, `Name` e uma propriedade somente leitura, `Counter`. A classe `Employee` implementa a interface `IEmployee` e usa essas duas propriedades. O programa lê o nome de um novo funcionário e o número atual de funcionários e exibe o nome do funcionário e o número do funcionário computado.

Seria possível usar o nome totalmente qualificado da propriedade, que referencia a interface na qual o membro é declarado. Por exemplo:

C#

```
string IEmployee.Name
{
```

```
    get { return "Employee Name"; }
    set { }
}
```

O exemplo anterior demonstra a Implementação explícita da interface. Por exemplo, se a classe `Employee` está implementando duas interfaces `ICitizen` e `IEmployee` e as duas interfaces têm a propriedade `Name`, é preciso haver a implementação explícita de membro da interface. Ou seja, a seguinte declaração de propriedade:

C#

```
string IEmployee.Name
{
    get { return "Employee Name"; }
    set { }
}
```

Implementa a propriedade `Name` na interface `IEmployee`, enquanto a seguinte declaração:

C#

```
string ICitizen.Name
{
    get { return "Citizen Name"; }
    set { }
}
```

Implementa a propriedade `Name` na interface `ICitizen`.

C#

```
interface IEmployee
{
    string Name
    {
        get;
        set;
    }

    int Counter
    {
        get;
    }
}

public class Employee : IEmployee
{
```

```
    public static int numberOfEmployees;

    private string _name;
    public string Name  // read-write instance property
    {
        get => _name;
        set => _name = value;
    }

    private int _counter;
    public int Counter  // read-only instance property
    {
        get => _counter;
    }

    // constructor
    public Employee() => _counter = ++numberOfEmployees;
}
```

C#

```
System.Console.Write("Enter number of employees: ");
Employee.numberOfEmployees = int.Parse(System.Console.ReadLine());

Employee e1 = new Employee();
System.Console.Write("Enter the name of the new employee: ");
e1.Name = System.Console.ReadLine();

System.Console.WriteLine("The employee information:");
System.Console.WriteLine("Employee number: {0}", e1.Counter);
System.Console.WriteLine("Employee name: {0}", e1.Name);
```

Saída de exemplo

Console

```
Enter number of employees: 210
Enter the name of the new employee: Hazem Abolrous
The employee information:
Employee number: 211
Employee name: Hazem Abolrous
```

Confira também

- Propriedades
- Usando propriedades
- Comparação entre propriedades e indexadores

- Indexadores
- Interfaces

Restringindo a acessibilidade ao acessador (Guia de Programação em C#)

Artigo • 30/10/2024

As partes get e set de uma propriedade ou de um indexador são chamadas *acessadores*. Por padrão, esses acessadores têm a mesma visibilidade ou nível de acesso da propriedade ou do indexador aos quais pertencem. Para obter mais informações, consulte níveis de acessibilidade. No entanto, às vezes é útil restringir o acesso a um desses acessadores. Normalmente, você restringe a acessibilidade do acessador `set`, mantendo o acessador `get` publicamente acessível. Por exemplo:

C#

```
private string _name = "Hello";

public string Name
{
    get
    {
        return _name;
    }
    protected set
    {
        _name = value;
    }
}
```

Neste exemplo, uma propriedade chamada `Name` define um acessador `get` e `set`. O acessador `get` recebe o nível de acessibilidade da propriedade em si, `public` nesse caso, embora o `set` acessador esteja restrito explicitamente ao aplicar o modificador de acesso protegido ao acessador em si.

> **Observação**
>
> Os exemplos neste artigo não usam propriedades implementadas automaticamente. *As propriedades* implementadas automaticamente fornecem uma sintaxe concisa para declarar propriedades quando um campo de suporte personalizado não é necessário.

Restrições em modificadores de acesso nos acessadores

O uso dos modificadores de acesso em propriedades ou indexadores está sujeito a estas condições:

- Não é possível usar modificadores de acessador em uma interface nem uma implementação explícita de membro de interface.
- É possível usar os modificadores de acessador somente se a propriedade ou o indexador tiver os acessadores `set` e `get`. Nesse caso, o modificador é permitido em apenas um dos dois acessadores.
- Se a propriedade ou o indexador tiver um modificador substituir, o modificador de acessador deverá corresponder o acessador do acessador substituído, se houver.
- O nível de acessibilidade do acessador deve ser mais restritivo do que o nível de acessibilidade na propriedade ou no indexador em si.

Modificadores de acesso em acessadores de substituição

Quando você substitui uma propriedade ou indexador, os acessadores substituídos devem estar acessíveis ao código de substituição. Além disso, a acessibilidade da propriedade/indexador, e seus acessadores, devem corresponder à propriedade/indexador substituído e seus acessadores. Por exemplo:

C#

```
public class Parent
{
    public virtual int TestProperty
    {
        // Notice the accessor accessibility level.
        protected set { }

        // No access modifier is used here.
        get { return 0; }
    }
}
public class Kid : Parent
{
    public override int TestProperty
    {
        // Use the same accessibility level as in the overridden accessor.
        protected set { }

        // Cannot use access modifier here.
```

```
        get { return 0; }
    }
}
```

Implementando interfaces

Quando você usa um acessador para implementar uma interface, o acessador pode não ter um modificador de acesso. No entanto, se você implementar a interface usando um acessador, como `get`, o outro acessador poderá ter um modificador de acesso, como no exemplo a seguir:

C#

```
public interface ISomeInterface
{
    int TestProperty
    {
        // No access modifier allowed here
        // because this is an interface.
        get;
    }
}

public class TestClass : ISomeInterface
{
    public int TestProperty
    {
        // Cannot use access modifier here because
        // this is an interface implementation.
        get { return 10; }

        // Interface property does not have set accessor,
        // so access modifier is allowed.
        protected set { }
    }
}
```

Domínio de acessibilidade do acessador

Se você usar um modificador de acesso no acessador, o domínio de acessibilidade do acessador será determinado por esse modificador.

Se você não usou um modificador de acesso no acessador, o domínio de acessibilidade do acessador será determinado pelo nível de acessibilidade da propriedade ou do indexador.

Exemplo

O exemplo a seguir contém três classes, `BaseClass`, `DerivedClass` e `MainClass`. Há duas propriedades no `BaseClass`, `Name` e `Id` em ambas as classes. O exemplo demonstra como a propriedade `Id` no `DerivedClass` pode ser oculta pela propriedade `Id` no `BaseClass` quando você usa um modificador de acesso restritivo como protegido ou privado. Portanto, em vez disso, quando você atribui valores a essa propriedade, a propriedade na classe `BaseClass` é chamada. Substituindo o modificador de acesso por público tornará a propriedade acessível.

O exemplo também demonstra que um modificador de acesso restritivo, como `private` ou `protected`, no acessador `set` da propriedade `Name` em `DerivedClass` impede o acesso ao acessador na classe derivada. Ele gera um erro quando você faz uma atribuição a ele ou acessa a propriedade de classe base com o mesmo nome, quando acessível.

C#

```
public class BaseClass
{
    private string _name = "Name-BaseClass";
    private string _id = "ID-BaseClass";

    public string Name
    {
        get { return _name; }
        set { }
    }

    public string Id
    {
        get { return _id; }
        set { }
    }
}

public class DerivedClass : BaseClass
{
    private string _name = "Name-DerivedClass";
    private string _id = "ID-DerivedClass";

    new public string Name
    {
        get
        {
            return _name;
        }

        // Using "protected" would make the set accessor not accessible.
```

```
        set
        {
            _name = value;
        }
    }

    // Using private on the following property hides it in the Main Class.
    // Any assignment to the property will use Id in BaseClass.
    new private string Id
    {
        get
        {
            return _id;
        }
        set
        {
            _id = value;
        }
    }
}

class MainClass
{
    static void Main()
    {
        BaseClass b1 = new BaseClass();
        DerivedClass d1 = new DerivedClass();

        b1.Name = "Mary";
        d1.Name = "John";

        b1.Id = "Mary123";
        d1.Id = "John123";  // The BaseClass.Id property is called.

        System.Console.WriteLine("Base: {0}, {1}", b1.Name, b1.Id);
        System.Console.WriteLine("Derived: {0}, {1}", d1.Name, d1.Id);

        // Keep the console window open in debug mode.
        System.Console.WriteLine("Press any key to exit.");
        System.Console.ReadKey();
    }
}
/* Output:
    Base: Name-BaseClass, ID-BaseClass
    Derived: John, ID-BaseClass
*/
```

Comentários

Observe que, se você substituir a declaração `new private string Id` por `new public string Id`, você obterá a saída:

`Name and ID in the base class: Name-BaseClass, ID-BaseClass` `Name and ID in the derived class: John, John123`

Confira também

- Propriedades
- Indexadores
- Modificadores de acesso
- Somente propriedades de inicialização
- Propriedades obrigatórias

Como declarar e usar propriedades de leitura e gravação (Guia de Programação em C#)

Artigo • 07/04/2023

As propriedades oferecem a conveniência de membros de dados públicos sem os riscos associados ao acesso sem proteção, sem controle e não verificado aos dados de um objeto. Propriedades declaram *acessadores*: métodos especiais que atribuem e recuperam valores do membro de dados subjacente. O acessador set habilita a atribuição de membros de dados e o acessador get recupera valores do membro de dados.

Este exemplo mostra uma classe `Person` que tem duas propriedades: `Name` (string) e `Age` (int). Ambas as propriedades fornecem acessadores `get` e `set`, portanto, são consideradas propriedades de leitura/gravação.

Exemplo

C#

```
class Person
{
    private string _name = "N/A";
    private int _age = 0;

    // Declare a Name property of type string:
    public string Name
    {
        get
        {
            return _name;
        }
        set
        {
            _name = value;
        }
    }

    // Declare an Age property of type int:
    public int Age
    {
        get
        {
            return _age;
```

```
        }

        set
        {
            _age = value;
        }
    }

    public override string ToString()
    {
        return "Name = " + Name + ", Age = " + Age;
    }
}

public class Wrapper
{
    private string _name = "N/A";
    public string Name
    {
        get
        {
            return _name;
        }
        private set
        {
            _name = value;
        }
    }

}

class TestPerson
{
    static void Main()
    {
        // Create a new Person object:
        Person person = new Person();

        // Print out the name and the age associated with the person:
        Console.WriteLine("Person details - {0}", person);

        // Set some values on the person object:
        person.Name = "Joe";
        person.Age = 99;
        Console.WriteLine("Person details - {0}", person);

        // Increment the Age property:
        person.Age += 1;
        Console.WriteLine("Person details - {0}", person);

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
```

```
/* Output:
    Person details - Name = N/A, Age = 0
    Person details - Name = Joe, Age = 99
    Person details - Name = Joe, Age = 100
*/
```

Programação robusta

No exemplo anterior, as propriedades `Name` e `Age` são públicas e incluem os acessadores `get` e `set`. Os acessadores públicos permitem que qualquer objeto leia e grave essas propriedades. No entanto, às vezes é desejável excluir um os acessadores. Você pode omitir o acessador `set` para tornar a propriedade somente leitura:

C#

```
public string Name
{
    get
    {
        return _name;
    }
    private set
    {
        _name = value;
    }
}
```

Como alternativa, é possível expor um acessador publicamente, porém, tornando o outro privado ou protegido. Para obter mais informações, consulte Acessibilidade do Acessador Assimétrico.

Depois de serem declaradas, as propriedades podem ser usadas como campos da classe. As propriedades permitem uma sintaxe natural na obtenção e configuração do valor de uma propriedade, conforme as instruções a seguir:

C#

```
person.Name = "Joe";
person.Age = 99;
```

Em um método de propriedade `set`, uma variável especial `value` está disponível. Essa variável contém o valor que o usuário especificou, por exemplo:

C#

```
_name = value;
```

Observe a sintaxe normal para incrementar a propriedade `Age` em um objeto `Person`:

```C#
person.Age += 1;
```

Se métodos `set` e `get` separados fossem usados para modelar propriedades, o código equivalente se pareceria com isto:

```C#
person.SetAge(person.GetAge() + 1);
```

O método `ToString` é substituído neste exemplo:

```C#
public override string ToString()
{
    return "Name = " + Name + ", Age = " + Age;
}
```

Observe que `ToString` não é usado explicitamente no programa. Ele é invocado por padrão pelas chamadas `WriteLine`.

Confira também

- Propriedades
- Sistema de tipos do C#

Propriedades implementadas automaticamente

Artigo • 14/11/2024

As propriedades implementadas automaticamente tornam a declaração de propriedade mais concisa quando nenhuma outra lógica é necessária nos acessadores de propriedade. Elas também habilitam o código do cliente a criar objetos. Ao declarar uma propriedade, como mostrado no exemplo a seguir, o compilador cria um campo de suporte privado e anônimo que pode ser acessado somente por meio dos acessadores `get` e `set` da propriedade. `init` Os acessadores também podem ser declarados como propriedades implementadas automaticamente.

O exemplo a seguir mostra uma classe simples que tem algumas propriedades implementadas automaticamente:

C#

```
// This class is mutable. Its data can be modified from
// outside the class.
public class Customer
{
    // Auto-implemented properties for trivial get and set
    public double TotalPurchases { get; set; }
    public string Name { get; set; }
    public int CustomerId { get; set; }

    // Constructor
    public Customer(double purchases, string name, int id)
    {
        TotalPurchases = purchases;
        Name = name;
        CustomerId = id;
    }

    // Methods
    public string GetContactInfo() { return "ContactInfo"; }
    public string GetTransactionHistory() { return "History"; }

    // .. Additional methods, events, etc.
}

class Program
{
    static void Main()
    {
        // Initialize a new object.
        Customer cust1 = new Customer(4987.63, "Northwind", 90108);
```

```
        // Modify a property.
        cust1.TotalPurchases += 499.99;
    }
}
```

Você não pode declarar propriedades implementadas automaticamente em interfaces. As propriedades implementadas automaticamente e com suporte de campo declaram um campo de suporte de instância privada e as interfaces não podem declarar campos de instância. Declarar uma propriedade em uma interface sem definir um corpo declara uma propriedade com acessadores. Cada tipo que implementa essa interface deve implementar essa propriedade.

Você pode inicializar propriedades implementadas automaticamente de forma semelhante aos campos:

C#

```
public string FirstName { get; set; } = "Jane";
```

A classe mostrada no exemplo anterior é mutável. O código cliente pode alterar os valores nos objetos após a criação. Em classes complexas que contêm comportamento significativo (métodos) e dados, geralmente é necessário ter propriedades públicas. No entanto, para classes pequenas ou structs que encapsulam apenas um conjunto de valores (dados) e têm pouco ou nenhum comportamento, você deve usar uma das seguintes opções para tornar os objetos imutáveis:

- Declare apenas um acessador `get` (imutável em todos os lugares, exceto o construtor).
- Declare um acessador `get` e um acessador `init` (imutável em todos os lugares, exceto durante a construção do objeto).
- Declare o acessador `set` como privado (imutável para os consumidores).

Para obter mais informações, consulte Como implementar uma classe leve com propriedades implementadas automaticamente.

Talvez seja necessário adicionar validação a uma propriedade implementada automaticamente. O C# 13 adiciona propriedades com suporte de campo como um recurso de visualização. Use a `field` palavra-chave para acessar o campo de suporte sintetizado do compilador de uma propriedade implementada automaticamente. Por exemplo, você pode garantir que a `FirstName` propriedade no exemplo anterior não possa ser definida como `null` ou a cadeia de caracteres vazia:

C#

```
public string FirstName
{
    get;
    set
    {
        field = (string.IsNullOrWhiteSpace(value) is false
            ? value
            : throw new ArgumentException(nameof(value), "First name can't
be whitespace or null"));
    }
} = "Jane";
```

Esse recurso permite que você adicione lógica aos acessadores sem exigir que você declare explicitamente o campo de suporte. Use a `field` palavra-chave para acessar o campo de suporte gerado pelo compilador.

> ⓘ **Importante**
>
> A `field` palavra-chave é um recurso de visualização no C# 13. Você deve estar usando o .NET 9 e definir seu `<LangVersion>` elemento como `preview` no arquivo de projeto para usar a `field` palavra-chave contextual.
>
> Você deve ter cuidado ao usar o recurso de `field` palavra-chave em uma classe que tenha um campo chamado `field`. A nova `field` palavra-chave sombreia um campo nomeado `field` no escopo de um acessador de propriedade. Você pode alterar o nome da `field` variável ou usar o `@` token para fazer referência ao `field` identificador como `@field`. Você pode saber mais lendo a especificação do recurso para **a field palavra-chave**.

Confira também

- Usar propriedades implementadas automaticamente (regra de estilo IDE0032)
- Propriedades
- Modificadores

Como implementar uma classe leve com propriedades implementadas automaticamente

Artigo • 26/10/2024

Este exemplo mostra como criar uma classe leve imutável que serve apenas para encapsular um conjunto de propriedades implementadas automaticamente. Use esse tipo de constructo em vez de um struct quando for necessário usar a semântica do tipo de referência.

Você pode tornar uma propriedade imutável das seguintes maneiras:

- Declare somente o acessador get, o que torna a propriedade imutável em todos os lugares, exceto no construtor do tipo.
- Declare um acessador init em vez de `set`, o que torna a propriedade configurável somente no construtor ou com o uso de um inicializador de objeto.
- Declare o acessador set como private. A propriedade será configurável somente dentro do tipo, mas imutável para os consumidores.

Você pode adicionar o modificador required à declaração de propriedade para forçar os chamadores a definir a propriedade como parte da inicialização de um novo objeto.

O exemplo a seguir mostra como uma propriedade somente com o acessador get difere de outra com get e conjunto privado.

C#

```
class Contact
{
    public string Name { get; }
    public string Address { get; private set; }

    public Contact(string contactName, string contactAddress)
    {
        // Both properties are accessible in the constructor.
        Name = contactName;
        Address = contactAddress;
    }

    // Name isn't assignable here. This will generate a compile error.
    //public void ChangeName(string newName) => Name = newName;

    // Address is assignable here.
```

```
    public void ChangeAddress(string newAddress) => Address = newAddress;
}
```

Exemplo

O exemplo a seguir mostra duas maneiras de implementar uma classe imutável que implementou propriedades automaticamente. Entre essas maneiras, uma declara uma das propriedades com um `set` privado e outra declara uma das propriedades somente com um `get`. A primeira classe usa um construtor somente para inicializar as propriedades e a segunda classe usa um método de fábrica estático que chama um construtor.

C#

```
// This class is immutable. After an object is created,
// it cannot be modified from outside the class. It uses a
// constructor to initialize its properties.
class Contact
{
    // Read-only property.
    public string Name { get; }

    // Read-write property with a private set accessor.
    public string Address { get; private set; }

    // Public constructor.
    public Contact(string contactName, string contactAddress)
    {
        Name = contactName;
        Address = contactAddress;
    }
}

// This class is immutable. After an object is created,
// it cannot be modified from outside the class. It uses a
// static method and private constructor to initialize its properties.
public class Contact2
{
    // Read-write property with a private set accessor.
    public string Name { get; private set; }

    // Read-only property.
    public string Address { get; }

    // Private constructor.
    private Contact2(string contactName, string contactAddress)
    {
        Name = contactName;
        Address = contactAddress;
    }
```

```
    // Public factory method.
    public static Contact2 CreateContact(string name, string address)
    {
        return new Contact2(name, address);
    }
}

public class Program
{
    static void Main()
    {
        // Some simple data sources.
        string[] names = ["Terry Adams","Fadi Fakhouri", "Hanying Feng",
                          "Cesar Garcia", "Debra Garcia"];
        string[] addresses = ["123 Main St.", "345 Cypress Ave.", "678 1st
Ave",
                              "12 108th St.", "89 E. 42nd St."];

        // Simple query to demonstrate object creation in select clause.
        // Create Contact objects by using a constructor.
        var query1 = from i in Enumerable.Range(0, 5)
                     select new Contact(names[i], addresses[i]);

        // List elements cannot be modified by client code.
        var list = query1.ToList();
        foreach (var contact in list)
        {
            Console.WriteLine("{0}, {1}", contact.Name, contact.Address);
        }

        // Create Contact2 objects by using a static factory method.
        var query2 = from i in Enumerable.Range(0, 5)
                     select Contact2.CreateContact(names[i],
addresses[i]);

        // Console output is identical to query1.
        var list2 = query2.ToList();

        // List elements cannot be modified by client code.
        // CS0272:
        // list2[0].Name = "Eugene Zabokritski";
    }
}

/* Output:
    Terry Adams, 123 Main St.
    Fadi Fakhouri, 345 Cypress Ave.
    Hanying Feng, 678 1st Ave
    Cesar Garcia, 12 108th St.
    Debra Garcia, 89 E. 42nd St.
*/
```

O compilador cria campos de suporte para cada propriedade implementada automaticamente. Os campos não são acessíveis diretamente do código-fonte.

Confira também

- Propriedades
- struct
- Inicializadores de objeto e coleção

Métodos (Guia de Programação em C#)

19/06/2025

Um método é um bloco de código que contém uma série de instruções. Um programa faz com que as instruções sejam executadas chamando o método e especificando os argumentos de método necessários. No C#, todas as instruções executadas são realizadas no contexto de um método.

O `Main` método é o ponto de entrada para cada aplicativo C# e é chamado pelo CLR (Common Language Runtime) quando o programa é iniciado. Em um aplicativo que usa instruções de nível superior, o `Main` método é gerado pelo compilador e contém todas as instruções de nível superior.

> **Observação**
>
> Este artigo discute métodos nomeados. Para obter informações sobre funções anônimas, consulte expressões lambda.

Assinaturas de método

Os métodos são declarados em uma classe, struct ou interface especificando o nível de acesso, como `public` ou `private`, modificadores opcionais, como `abstract` ou `sealed`, o valor retornado, o nome do método e quaisquer parâmetros de método. Essas partes juntas são a assinatura do método.

> **Importante**
>
> Um tipo de retorno de um método não faz parte da assinatura do método para fins de sobrecarga de método. No entanto, ele faz parte da assinatura do método ao determinar a compatibilidade entre um delegado e o método para o qual ele aponta.

Os parâmetros de método estão entre parênteses e separados por vírgulas. Parênteses vazios indicam que o método não requer parâmetros. Esta classe contém quatro métodos:

C#

```
abstract class Motorcycle
{
    // Anyone can call this.
    public void StartEngine() {/* Method statements here */ }
```

```
    // Only derived classes can call this.
    protected void AddGas(int gallons) { /* Method statements here */ }

    // Derived classes can override the base class implementation.
    public virtual int Drive(int miles, int speed) { /* Method statements here */
return 1; }

    // Derived classes must implement this.
    public abstract double GetTopSpeed();
}
```

Acesso a método

Chamar um método em um objeto é como acessar um campo. Após o nome do objeto, adicione um período, o nome do método e parênteses. Os argumentos são listados dentro dos parênteses e separados por vírgulas. Os métodos da `Motorcycle` classe podem, portanto, ser chamados como no exemplo a seguir:

C#

```
class TestMotorcycle : Motorcycle
{
    public override double GetTopSpeed()
    {
        return 108.4;
    }

    static void Main()
    {
        TestMotorcycle moto = new TestMotorcycle();

        moto.StartEngine();
        moto.AddGas(15);
        moto.Drive(5, 20);
        double speed = moto.GetTopSpeed();
        Console.WriteLine($"My top speed is {speed}");
    }
}
```

Parâmetros de método versus argumentos

A definição do método especifica os nomes e tipos de quaisquer parâmetros obrigatórios. Quando o código de chamada invoca o método, ele fornece valores concretos chamados argumentos para cada parâmetro. Os argumentos devem ser compatíveis com o tipo de parâmetro, mas o nome do argumento (se houver) usado no código de chamada não precisa ser o mesmo que o parâmetro nomeado definido no método. Por exemplo:

C#

```
public void Caller()
{
    int numA = 4;
    // Call with an int variable.
    int productA = Square(numA);

    int numB = 32;
    // Call with another int variable.
    int productB = Square(numB);

    // Call with an integer literal.
    int productC = Square(12);

    // Call with an expression that evaluates to int.
    productC = Square(productA * 3);
}

int Square(int i)
{
    // Store input argument in a local variable.
    int input = i;
    return input * input;
}
```

Passando por referência versus passando por valor

Por padrão, quando uma instância de um tipo de valor é passada para um método, sua cópia é passada em vez da própria instância. Portanto, as alterações no argumento não têm efeito sobre a instância original no método de chamada. Para passar uma instância de tipo de valor por referência, use a palavra-chave `ref`. Para obter mais informações, consulte Como passar parâmetros de Value-Type.

Quando um objeto de um tipo de referência é passado para um método, uma referência ao objeto é passada. Ou seja, o método não recebe o objeto em si, mas um argumento que indica o local do objeto. Se você alterar um membro do objeto usando essa referência, a alteração será refletida no argumento no método de chamada, ainda que você passe o objeto por valor.

Você cria um tipo de referência usando a `class` palavra-chave, como mostra o exemplo a seguir:

C#

```
public class SampleRefType
{
```

```
    public int value;
}
```

Agora, se você passar um objeto baseado nesse tipo para um método, uma referência ao objeto será passada. O exemplo a seguir passa um objeto de tipo `SampleRefType` para o método `ModifyObject`:

C#

```
public static void TestRefType()
{
    SampleRefType rt = new SampleRefType();
    rt.value = 44;
    ModifyObject(rt);
    Console.WriteLine(rt.value);
}

static void ModifyObject(SampleRefType obj)
{
    obj.value = 33;
}
```

O exemplo faz essencialmente a mesma coisa que o exemplo anterior em que passa um argumento por valor para um método. Mas, como um tipo de referência é usado, o resultado é diferente. A modificação feita no campo `ModifyObject` do parâmetro, `value`, também altera o campo `obj` do argumento, `value`, no método `rt`. O `TestRefType` método exibe 33 como saída.

Para obter mais informações sobre como passar tipos de referência por referência e por valor, consulte Passando parâmetros de tipo de referência e Tipos de referência.

Valores retornados

Os métodos podem retornar um valor para o chamador. Se o tipo de retorno (o tipo listado antes do nome do método) não for `void`, o método poderá retornar o valor usando a instrução return. Uma instrução com a `return` palavra-chave seguida por um valor que corresponde ao tipo de retorno retornará esse valor para o chamador do método.

O valor pode ser retornado ao chamador por valor ou por referência. Valores são retornados ao chamador por referência se a `ref` palavra-chave é usada na assinatura do método e segue cada palavra-chave `return`. Por exemplo, a assinatura do método a seguir e a instrução return indicam que o método retorna uma variável nomeada `estDistance` por referência ao chamador.

C#

```
public ref double GetEstimatedDistance()
{
    return ref estDistance;
}
```

A palavra-chave `return` também interrompe a execução do método. Se o tipo de retorno for `void`, uma instrução `return` sem um valor ainda será útil para interromper a execução do método. Sem a `return` palavra-chave, o método interromperá a execução quando chegar ao final do bloco de código. Métodos com um tipo de retorno não nulo são necessários para usar a `return` palavra-chave para retornar um valor. Por exemplo, esses dois métodos usam a palavra-chave `return` para retornar inteiros:

C#

```
class SimpleMath
{
    public int AddTwoNumbers(int number1, int number2)
    {
        return number1 + number2;
    }

    public int SquareANumber(int number)
    {
        return number * number;
    }
}
```

Para usar um valor retornado de um método, o método de chamada pode usar o método chamado em qualquer lugar em que um valor do mesmo tipo seria suficiente. Você também pode atribuir o valor retornado a uma variável. Por exemplo, os dois exemplos de código a seguir atingem a mesma meta:

C#

```
int result = obj.AddTwoNumbers(1, 2);
result = obj.SquareANumber(result);
// The result is 9.
Console.WriteLine(result);
```

C#

```
result = obj.SquareANumber(obj.AddTwoNumbers(1, 2));
// The result is 9.
Console.WriteLine(result);
```

O uso de uma variável local, nesse caso, `result` para armazenar um valor é opcional. Ele pode ajudar na legibilidade do código ou pode ser necessário se você precisar armazenar o valor original do argumento para todo o escopo do método.

Para usar um valor retornado por referência de um método, você deve declarar uma variável local ref se pretende modificar seu valor. Por exemplo, se o `Planet.GetEstimatedDistance` método retornar um Double valor por referência, você poderá defini-lo como uma variável local ref com código como o seguinte:

```
C#

ref double distance = ref Planet.GetEstimatedDistance();
```

Retornar uma matriz multidimensional de um método, `M` que modifica o conteúdo da matriz não é necessário se a função de chamada passou a matriz para `M`. Você pode retornar a matriz resultante de `M` para melhorar o estilo ou o fluxo funcional dos valores, mas isso não é necessário porque o C# passa todos os tipos de referência por valor, e o valor de uma referência de matriz é o ponteiro para a matriz. No método `M`, todas as alterações no conteúdo da matriz são observáveis por qualquer código que tenha uma referência à matriz, conforme mostrado no exemplo a seguir:

```
C#

static void Main(string[] args)
{
    int[,] matrix = new int[2, 2];
    FillMatrix(matrix);
    // matrix is now full of -1
}

public static void FillMatrix(int[,] matrix)
{
    for (int i = 0; i < matrix.GetLength(0); i++)
    {
        for (int j = 0; j < matrix.GetLength(1); j++)
        {
            matrix[i, j] = -1;
        }
    }
}
```

Métodos assíncronos

Usando o recurso async, você pode invocar métodos assíncronos sem usar retornos de chamada explícitos ou dividir manualmente seu código entre vários métodos ou expressões

lambda.

Se você marcar um método com o modificador assíncrono , poderá usar o operador await no método. Quando o controle atinge uma expressão await no método assíncrono, ele retorna para o chamador e o progresso no método é suspenso até a tarefa aguardada ser concluída. Quando a tarefa for concluída, a execução poderá ser retomada no método.

> ⓘ **Observação**
>
> Um método assíncrono retorna para o chamador quando encontra o primeiro objeto esperado que ainda não está completo ou chega ao final do método assíncrono, o que ocorrer primeiro.

Um método assíncrono normalmente tem um tipo de retorno de Task<TResult>, TaskIAsyncEnumerable<T>ou `void`. O tipo de retorno `void` é usado principalmente para definir manipuladores de eventos, nos quais o tipo de retorno `void` é necessário. Um método assíncrono que retorna `void` não pode ser aguardado e o chamador de um método de retorno nulo não pode capturar as exceções que esse método gera. Um método assíncrono pode ter qualquer tipo de retorno semelhante a uma tarefa.

No exemplo a seguir, `DelayAsync` é um método assíncrono que tem um tipo de retorno de Task<TResult>. `DelayAsync` tem uma instrução `return` que retorna um número inteiro. Portanto, a declaração do método `DelayAsync` deve ter um tipo de retorno `Task<int>`. Como o tipo de retorno é `Task<int>`, a avaliação da `await` expressão em `DoSomethingAsync` produz um inteiro, como demonstra a instrução a seguir: `int result = await delayTask`.

O `Main` método é um exemplo de um método assíncrono que tem um tipo de retorno de Task. Ele vai para o método `DoSomethingAsync` e, como ele é expresso com uma única linha, pode omitir as palavras-chave `async` e `await`. Como `DoSomethingAsync` é um método assíncrono, a tarefa para a chamada `DoSomethingAsync` deve ser aguardada, como mostra a seguinte instrução: `await DoSomethingAsync();`.

C#

```
class Program
{
    static Task Main() => DoSomethingAsync();

    static async Task DoSomethingAsync()
    {
        Task<int> delayTask = DelayAsync();
        int result = await delayTask;
```

```
        // The previous two statements may be combined into
        // the following statement.
        //int result = await DelayAsync();

        Console.WriteLine($"Result: {result}");
    }

    static async Task<int> DelayAsync()
    {
        await Task.Delay(100);
        return 5;
    }
}
// Example output:
//   Result: 5
```

Um método assíncrono não pode declarar parâmetros ref ou out , mas pode chamar métodos que têm esses parâmetros.

Para obter mais informações sobre métodos assíncronos, consulte Programação assíncrona com async e await e Tipos de retorno assíncronos.

Definições do corpo da expressão

É comum ter definições de método que simplesmente retornam imediatamente com o resultado de uma expressão ou que tenham uma única instrução como o corpo do método. Há um atalho de sintaxe para definir esses métodos usando `=>`:

C#

```
public Point Move(int dx, int dy) => new Point(x + dx, y + dy);
public void Print() => Console.WriteLine(First + " " + Last);
// Works with operators, properties, and indexers too.
public static Complex operator +(Complex a, Complex b) => a.Add(b);
public string Name => First + " " + Last;
public Customer this[long id] => store.LookupCustomer(id);
```

Se o método retornar `void` ou for um método assíncrono, o corpo do método deverá ser uma expressão de instrução (igual a lambdas). Para propriedades e indexadores, eles devem ser somente leitura e você não usa a palavra-chave do acessador `get`.

Iteradores

Um iterador realiza uma iteração personalizada em uma coleção, como uma lista ou uma matriz. Um iterador usa a instrução de retorno de rendimento para retornar cada elemento um

de cada vez. Quando uma instrução `yield return` for atingida, o local atual no código será lembrado. A execução é reiniciada desse local quando o iterador é chamado na próxima vez.

Você chama um iterador do código do cliente usando uma instrução foreach.

O tipo de retorno de um iterador pode ser IEnumerable, IEnumerable<T>, IAsyncEnumerable<T>, IEnumerator ou IEnumerator<T>.

Para obter mais informações, consulte Iteradores.

Especificação da linguagem C#

Para obter mais informações, consulte a Especificação da Linguagem C# . A especificação de idioma é a fonte definitiva para a sintaxe e o uso de C#.

Consulte também

- Sistema de tipos do C#
- Modificadores de acesso
- Classes estáticas e membros de classes estáticas
- Herança
- Classes abstratas e seladas e membros da classe
- params
-
- Ref
- Parâmetros do método

Funções locais (Guia de Programação em C#)

Artigo • 31/12/2024

Funções locais são métodos de um tipo que estão aninhados em outro membro. Eles só podem ser chamados do membro que os contém. Funções locais podem ser declaradas em e chamadas de:

- Métodos, especialmente os métodos iteradores e os métodos assíncronos
- Construtores
- Acessadores de propriedades
- Acessadores de eventos
- Métodos anônimos
- Expressões lambda
- Finalizadores
- Outras funções locais

No entanto, as funções locais não podem ser declaradas dentro de um membro apto para expressão.

> ⓘ **Observação**
>
> Em alguns casos, você pode usar uma expressão lambda para implementar uma funcionalidade que também tem suporte por uma função local. Para obter uma comparação, confira **Funções locais em comparação a expressões Lambda**.

Funções locais tornam a intenção do seu código clara. Qualquer pessoa que leia seu código pode ver que o método não pode ser chamado, exceto pelo método que contém. Para projetos de equipe, elas também impossibilitam que outro desenvolvedor chame o método por engano diretamente de qualquer outro lugar na classe ou no struct.

Sintaxe de função local

Uma função local é definida como um método aninhado dentro de um membro recipiente. Sua definição tem a seguinte sintaxe:

```
C#
```

```
<modifiers> <return-type> <method-name> <parameter-list>
```

> ⓘ **Observação**
>
> O `<parameter-list>` não deve conter os parâmetros nomeados com palavra-chave contextual `value`. O compilador cria a variável temporária "value", que contém as variáveis externas referenciadas, que posteriormente causam ambiguidade e também podem causar um comportamento inesperado.

Você pode usar os seguintes modificadores com uma função local:

- async
- unsafe
- static Uma função local estática não pode capturar variáveis locais nem o estado da instância.
- extern Uma função local externa deve ser `static`.

Todas as variáveis locais definidas no membro relativo, incluindo os parâmetros do método, são acessíveis em uma função local não estática.

Ao contrário de uma definição de método, uma definição de função local não pode incluir o modificador de acesso de membro. Como todas as funções locais são privadas, incluindo um modificador de acesso, como a palavra-chave `private`, gera o erro do compilador CS0106, "O modificador 'private' não é válido para este item.".

O exemplo a seguir define uma função local chamada `AppendPathSeparator` que é privada para um método chamado `GetText`:

C#

```
private static string GetText(string path, string filename)
{
     var reader = File.OpenText($"{AppendPathSeparator(path)}{filename}");
     var text = reader.ReadToEnd();
     return text;

     string AppendPathSeparator(string filepath)
     {
        return filepath.EndsWith(@"\") ? filepath : filepath + @"\";
     }
}
```

Você pode aplicar atributos a uma função local, seus parâmetros e parâmetros de tipo, como mostra o exemplo a seguir:

C#

```
#nullable enable
private static void Process(string?[] lines, string mark)
{
    foreach (var line in lines)
    {
        if (IsValid(line))
        {
            // Processing logic...
        }
    }

    bool IsValid([NotNullWhen(true)] string? line)
    {
        return !string.IsNullOrEmpty(line) && line.Length >= mark.Length;
    }
}
```

O exemplo anterior usa um atributo especial para ajudar o compilador na análise estática em um contexto anulável.

Funções locais e exceções

Um dos recursos úteis de funções locais é que elas podem permitir que exceções sejam apresentadas imediatamente. Para métodos de iteradores, as exceções são apresentadas somente quando a sequência retornada é enumerada e não quando o iterador é recuperado. Para métodos assíncronos, as exceções geradas em um método assíncrono são observadas quando a tarefa retornada é esperada.

O exemplo a seguir define um método `OddSequence` que enumera números ímpares em um intervalo especificado. Já que ele passa um número maior que 100 para o método enumerador `OddSequence`, o método gera uma ArgumentOutOfRangeException. Assim como demonstrado pela saída do exemplo, a exceção é apresentada somente quando você itera os números e não quando você recupera o enumerador.

C#

```
public class IteratorWithoutLocalExample
{
   public static void Main()
   {
      IEnumerable<int> xs = OddSequence(50, 110);
      Console.WriteLine("Retrieved enumerator...");
```

```
      foreach (var x in xs)  // line 11
      {
         Console.Write($"{x} ");
      }
   }

   public static IEnumerable<int> OddSequence(int start, int end)
   {
      if (start < 0 || start > 99)
         throw new ArgumentOutOfRangeException(nameof(start), "start must be
between 0 and 99.");
      if (end > 100)
         throw new ArgumentOutOfRangeException(nameof(end), "end must be
less than or equal to 100.");
      if (start >= end)
         throw new ArgumentException("start must be less than end.");

      for (int i = start; i <= end; i++)
      {
         if (i % 2 == 1)
            yield return i;
      }
   }
}
// The example displays the output like this:
//
//    Retrieved enumerator...
//    Unhandled exception. System.ArgumentOutOfRangeException: end must be
less than or equal to 100. (Parameter 'end')
//    at IteratorWithoutLocalExample.OddSequence(Int32 start, Int32
end)+MoveNext() in IteratorWithoutLocal.cs:line 22
//    at IteratorWithoutLocalExample.Main() in IteratorWithoutLocal.cs:line
11
```

Se você colocar a lógica do iterador em uma função local, as exceções de validação de argumento serão geradas quando você recuperar o enumerador, como mostra o exemplo a seguir:

C#

```
public class IteratorWithLocalExample
{
   public static void Main()
   {
      IEnumerable<int> xs = OddSequence(50, 110);  // line 8
      Console.WriteLine("Retrieved enumerator...");

      foreach (var x in xs)
      {
         Console.Write($"{x} ");
      }
   }
```

```
   public static IEnumerable<int> OddSequence(int start, int end)
   {
      if (start < 0 || start > 99)
         throw new ArgumentOutOfRangeException(nameof(start), "start must be
between 0 and 99.");
      if (end > 100)
         throw new ArgumentOutOfRangeException(nameof(end), "end must be
less than or equal to 100.");
      if (start >= end)
         throw new ArgumentException("start must be less than end.");

      return GetOddSequenceEnumerator();

      IEnumerable<int> GetOddSequenceEnumerator()
      {
         for (int i = start; i <= end; i++)
         {
            if (i % 2 == 1)
               yield return i;
         }
      }
   }
}
// The example displays the output like this:
//
//    Unhandled exception. System.ArgumentOutOfRangeException: end must be
less than or equal to 100. (Parameter 'end')
//    at IteratorWithLocalExample.OddSequence(Int32 start, Int32 end) in
IteratorWithLocal.cs:line 22
//    at IteratorWithLocalExample.Main() in IteratorWithLocal.cs:line 8
```

Funções locais vs. expressões lambda

À primeira vista, as funções locais e as expressões lambda são semelhantes. Em muitos casos, a escolha entre usar expressões lambda e funções locais é uma questão de estilo e preferência pessoal. No entanto, há diferenças reais nos casos em que você pode usar uma ou outra, e é importante conhecer essas diferenças.

Examinaremos as diferenças entre a função local e as implementações de expressão lambda do algoritmo fatorial. Esta é a versão que usa uma função local:

C#

```
public static int LocalFunctionFactorial(int n)
{
    return nthFactorial(n);

    int nthFactorial(int number) => number < 2
        ? 1
```

```
        : number * nthFactorial(number - 1);
}
```

Esta versão usa expressões lambda:

```C#
public static int LambdaFactorial(int n)
{
    Func<int, int> nthFactorial = default(Func<int, int>);

    nthFactorial = number => number < 2
        ? 1
        : number * nthFactorial(number - 1);

    return nthFactorial(n);
}
```

Nomenclatura

As funções locais são explicitamente nomeadas como métodos. As expressões lambda são métodos anônimos e precisam ser atribuídas a variáveis de um tipo `delegate`, normalmente os tipos `Action` ou `Func`. Quando você declara uma função local, o processo é como gravar um método normal. Você declara um tipo de retorno e uma assinatura de função.

Assinaturas de função e tipos de expressão lambda

As expressões lambda dependem do tipo da variável `Action`/`Func` atribuída para determinar o argumento e os tipos de retorno. Em funções locais, como a sintaxe é muito parecida com gravar um método normal, os tipos de argumento e o tipo de retorno já fazem parte da declaração de função.

Algumas expressões lambda têm um tipo natural , o que permite que o compilador infera o tipo de retorno e os tipos de parâmetro da expressão lambda.

Atribuição definida

As expressões lambda são objetos declarados e atribuídos em tempo de execução. Para que uma expressão lambda seja usada, ela precisa ser definitivamente atribuída: a variável `Action`/`Func` à qual ela foi atribuída deve ser declarada e a expressão lambda atribuída a ela. Observe que `LambdaFactorial` deve declarar e inicializar a expressão

lambda `nthFactorial` antes de defini-la. Não fazer isso resulta em um erro em tempo de compilação para referenciar `nthFactorial` antes de atribuí-lo.

As funções locais são definidas em tempo de compilação. Como elas não são atribuídas a variáveis, elas podem ser referenciadas de qualquer local de código **em que ela está no escopo**; no primeiro exemplo `LocalFunctionFactorial`, você pode declarar a função local antes ou depois da instrução `return` e não disparar erros do compilador.

Essas diferenças significam que os algoritmos recursivos são mais fáceis de criar usando funções locais. Você pode declarar e definir uma função local que chame a si mesma. As expressões Lambda devem ser declaradas e atribuídas a um valor padrão antes de serem reatribuídas a um corpo que faça referência à mesma expressão lambda.

Implementação como delegado

As expressões lambda são convertidas em delegados quando declaradas. As funções locais são mais flexíveis, pois podem ser gravadas como método tradicional *ou* como delegado. As funções locais só são convertidas em delegados quando ***usadas*** como delegado.

Se você declarar uma função local e apenas referenciá-la chamando-a como se fosse um método, ela não será convertida em um delegado.

Captura de variável

As regras de atribuição definitiva também afetam as variáveis capturadas pela função local ou pela expressão lambda. O compilador pode executar uma análise estática, que permite que as funções locais atribuam de maneira definitiva as variáveis capturadas no escopo delimitador. Considere este exemplo:

C#

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

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

O compilador pode determinar que `LocalFunction` definitivamente atribua `y` quando chamada. Como a `LocalFunction` é chamada antes da instrução `return`, `y` é atribuído

definitivamente na instrução `return`.

Quando uma função local captura variáveis no escopo delimitador, ela é implementada usando um fechamento, assim como são os tipos delegados.

Alocações de heap

Dependendo do uso, as funções locais podem evitar as alocações de heap que são sempre necessárias nas expressões lambda. Se uma função local nunca é convertida em um delegado, e nenhuma das variáveis capturadas pela função local é capturada por outras lambdas ou funções locais convertidas em delegados, o compilador pode evitar alocações de heap.

Considere este exemplo assíncrono:

C#

```
public async Task<string> PerformLongRunningWorkLambda(string address, int
index, string name)
{
    if (string.IsNullOrWhiteSpace(address))
        throw new ArgumentException(message: "An address is required",
paramName: nameof(address));
    if (index < 0)
        throw new ArgumentOutOfRangeException(paramName: nameof(index),
message: "The index must be non-negative");
    if (string.IsNullOrWhiteSpace(name))
        throw new ArgumentException(message: "You must supply a name",
paramName: nameof(name));

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

    return await longRunningWorkImplementation();
}
```

O fechamento dessa expressão lambda contém as variáveis `address`, `index` e `name`. Para funções locais, o objeto que implementa o fechamento pode ser um tipo `struct`. Esse tipo de struct seria passado por referência à função local. Essa diferença na implementação poderia economizar em uma alocação.

A instanciação necessária para expressões lambda significa alocações de memória extras, que podem ser um fator de desempenho em caminhos de código críticos. As funções locais não incorrem nessa sobrecarga.

Se você sabe que a função local não será convertida em um delegado e nenhuma das variáveis capturadas por ela será capturada por outras lambdas ou funções locais convertidas em delegados, você pode garantir que a função local evite ser alocada no heap, declarando-a como função local `static`.

Dica

Habilite a regra de estilo de código .NET **IDE0062**, para garantir que as funções locais sejam sempre marcadas como `static`.

Observação

A função local equivalente desse método também usa uma classe para o fechamento. O fechamento de uma função local ser implementado como um `class` ou como um `struct`, trata-se de um detalhe de implementação. Uma função local pode usar um `struct`, enquanto uma lambda sempre usará um `class`.

C#

```
public async Task<string> PerformLongRunningWork(string address, int index,
string name)
{
    if (string.IsNullOrWhiteSpace(address))
        throw new ArgumentException(message: "An address is required",
paramName: nameof(address));
    if (index < 0)
        throw new ArgumentOutOfRangeException(paramName: nameof(index),
message: "The index must be non-negative");
    if (string.IsNullOrWhiteSpace(name))
        throw new ArgumentException(message: "You must supply a name",
paramName: nameof(name));

    return await longRunningWorkImplementation();

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

```
    }
}
```

Uso da palavra-chave `yield`

Uma vantagem final não demonstrada neste exemplo é que as funções locais podem ser implementadas como iteradores, usando a sintaxe `yield return` para produzir uma sequência de valores.

```C#
public IEnumerable<string> SequenceToLowercase(IEnumerable<string> input)
{
    if (!input.Any())
    {
        throw new ArgumentException("There are no items to convert to
lowercase.");
    }

    return LowercaseIterator();

    IEnumerable<string> LowercaseIterator()
    {
        foreach (var output in input.Select(item => item.ToLower()))
        {
            yield return output;
        }
    }
}
```

A instrução `yield return` não é permitida em expressões lambda. Para obter mais informações, confira Erro do compilador CS1621.

Embora as funções locais possam parecer redundantes para expressões lambda, elas realmente servem a diferentes finalidades e têm usos diferentes. As funções locais são mais eficientes para quando você deseja escrever uma função que é chamada apenas do contexto de outro método.

Especificação da linguagem C#

Para obter mais informações, confira a seção Declarações de função local da Especificação da linguagem C#.

Confira também

- Usar função local em vez do lambda (regra de estilo IDE0039)
- Métodos

Variáveis locais de tipo implícito (Guia de Programação em C#)

Artigo • 14/03/2023

Variáveis locais podem ser declaradas sem fornecer um tipo explícito. A palavra-chave `var` instrui o compilador a inferir o tipo da variável da expressão no lado direito da instrução de inicialização. O tipo inferido pode ser um tipo interno, um tipo anônimo, um tipo definido pelo usuário ou um tipo definido na biblioteca de classes .NET. Para obter mais informações sobre como inicializar matrizes com `var`, consulte Matrizes de tipo implícito.

Os exemplos a seguir mostram várias maneiras em que as variáveis locais podem ser declaradas com `var`:

C#

```
// i is compiled as an int
var i = 5;

// s is compiled as a string
var s = "Hello";

// a is compiled as int[]
var a = new[] { 0, 1, 2 };

// expr is compiled as IEnumerable<Customer>
// or perhaps IQueryable<Customer>
var expr =
    from c in customers
    where c.City == "London"
    select c;

// anon is compiled as an anonymous type
var anon = new { Name = "Terry", Age = 34 };

// list is compiled as List<int>
var list = new List<int>();
```

É importante entender que a palavra-chave `var` não significa "variante" e não indica que a variável é vagamente tipada ou de associação tardia. Isso apenas significa que o compilador determina e atribui o tipo mais apropriado.

A palavra-chave `var` pode ser usada nos seguintes contextos:

- Em variáveis locais (variáveis declaradas no escopo do método) conforme mostrado no exemplo anterior.
- Em uma instrução de inicialização for.

```
C#

for (var x = 1; x < 10; x++)
```

- Em uma instrução de inicialização foreach.

```
C#

foreach (var item in list) {...}
```

- Em uma instrução using.

```
C#

using (var file = new StreamReader("C:\\myfile.txt")) {...}
```

Para obter mais informações, consulte Como usar matrizes e variáveis locais de tipo implícito em uma expressão de consulta.

Tipos var e anônimos

Em muitos casos, o uso de `var` é opcional e é apenas uma conveniência sintática. No entanto, quando uma variável é inicializada com um tipo anônimo você deve declarar a variável como `var` se precisar acessar as propriedades do objeto em um momento posterior. Esse é um cenário comum em expressões de consulta LINQ. Para obter mais informações, consulte Tipos Anônimos.

Da perspectiva do código-fonte, um tipo anônimo não tem nome. Portanto, se uma variável de consulta tiver sido inicializada com `var`, a única maneira de acessar as propriedades na sequência retornada será usar `var` como o tipo da variável de iteração na instrução `foreach`.

```
C#

class ImplicitlyTypedLocals2
{
    static void Main()
    {
```

```
        string[] words = { "aPPLE", "BlUeBeRrY", "cHeRry" };

        // If a query produces a sequence of anonymous types,
        // then use var in the foreach statement to access the properties.
        var upperLowerWords =
             from w in words
             select new { Upper = w.ToUpper(), Lower = w.ToLower() };

        // Execute the query
        foreach (var ul in upperLowerWords)
        {
            Console.WriteLine("Uppercase: {0}, Lowercase: {1}", ul.Upper,
ul.Lower);
        }
    }
}
/* Outputs:
    Uppercase: APPLE, Lowercase: apple
    Uppercase: BLUEBERRY, Lowercase: blueberry
    Uppercase: CHERRY, Lowercase: cherry
 */
```

Comentários

As seguintes restrições se aplicam às declarações de variável de tipo implícito:

- `var` pode ser usado apenas quando uma variável local é declarada e inicializada na mesma instrução, a variável não pode ser inicializada como nula, um grupo de métodos ou uma função anônima.
- `var` não pode ser usado em campos no escopo da classe.
- Variáveis declaradas usando `var` não podem ser usadas na expressão de inicialização. Em outras palavras, essa expressão é válida: `int i = (i = 20);`, mas essa expressão gera um erro em tempo de compilação: `var i = (i = 20);`
- Diversas variáveis de tipo implícito não podem ser inicializadas na mesma instrução.
- Se um tipo nomeado `var` estiver no escopo, a palavra-chave `var` será resolvida para esse nome de tipo e não será tratada como parte de uma declaração de variável local de tipo implícito.

Tipagem implícita com a palavra-chave `var` só pode ser aplicada às variáveis no escopo do método local. Digitação implícita não está disponível para os campos de classe, uma vez que o compilador C# encontraria um paradoxo lógico ao processar o código: o compilador precisa saber o tipo do campo, mas não é possível determinar o tipo até

que a expressão de atribuição seja analisada. A expressão não pode ser avaliada sem saber o tipo. Considere o seguinte código:

```
C#

private var bookTitles;
```

`bookTitles` é um campo de classe dado o tipo `var`. Como o campo não tem nenhuma expressão para avaliar, é impossível para o compilador inferir que tipo `bookTitles` deveria ser. Além disso, também é insuficiente adicionar uma expressão ao campo (como você faria para uma variável local):

```
C#

private var bookTitles = new List<string>();
```

Quando o compilador encontra campos durante a compilação de código, ele registra cada tipo de campo antes de processar quaisquer expressões associadas. O compilador encontra o mesmo paradoxo ao tentar analisar `bookTitles`: ele precisa saber o tipo do campo, mas o compilador normalmente determinaria o tipo de `var` analisando a expressão, o que não possível sem saber o tipo com antecedência.

Você pode descobrir que `var` também pode ser útil com expressões de consulta em que o tipo construído exato da variável de consulta é difícil de ser determinado. Isso pode ocorrer com operações de agrupamento e classificação.

A palavra-chave `var` também pode ser útil quando o tipo específico da variável é enfadonho de digitar no teclado, é óbvio ou não acrescenta à legibilidade do código. Um exemplo em que `var` é útil dessa maneira é com os tipos genéricos aninhados, como os usados com operações de grupo. Na consulta a seguir, o tipo da variável de consulta é `IEnumerable<IGrouping<string, Student>>`. Contanto que você e as outras pessoas que devem manter o código entendam isso, não há problema em usar a tipagem implícita por questões de conveniência e brevidade.

```
C#

// Same as previous example except we use the entire last name as a key.
// Query variable is an IEnumerable<IGrouping<string, Student>>
var studentQuery3 =
    from student in students
    group student by student.Last;
```

O uso de `var` ajuda a simplificar o código, mas seu uso deve ser restrito a casos em que ele é necessário ou facilita a leitura do código. Para obter mais informações sobre quando usar `var` corretamente, confira a seção Variáveis locais digitados implicitamente no artigo Diretrizes de Codificação de C#.

Confira também

- Referência de C#
- Matrizes de tipo implícito
- Como usar matrizes e variáveis locais de tipo implícito em uma expressão de consulta
- Tipos anônimos
- Inicializadores de objeto e coleção
- var
- LINQ em C#
- LINQ (Consulta Integrada à Linguagem)
- Instruções de iteração
- Instrução using

Como usar variáveis locais de tipo implícito e matrizes em uma expressão de consulta (Guia de Programação em C#)

Artigo • 07/04/2023

Será possível usar variáveis locais de tipo implícito sempre que você desejar que o compilador determine o tipo de uma variável local. É necessário usar variáveis locais de tipo implícito para armazenar tipos anônimos, usados frequentemente em expressões de consulta. Os exemplos a seguir ilustram usos obrigatórios e opcionais de variáveis locais de tipo implícito em consultas.

As variáveis locais de tipo implícito são declaradas usando a palavra-chave contextual var. Para obter mais informações, consulte Variáveis locais de tipo implícito e Matrizes de tipo implícito.

Exemplos

O exemplo a seguir mostra um cenário comum em que a palavra-chave `var` é necessária: uma expressão de consulta que produz uma sequência de tipos anônimos. Nesse cenário, a variável de consulta e a variável de iteração na instrução `foreach` devem ser tipadas implicitamente usando `var`, porque você não tem acesso a um nome de tipo para o tipo anônimo. Para obter mais informações sobre tipos anônimos, consulte Tipos anônimos.

C#

```
private static void QueryNames(char firstLetter)
{
    // Create the query. Use of var is required because
    // the query produces a sequence of anonymous types:
    // System.Collections.Generic.IEnumerable<????>.
    var studentQuery =
        from student in students
        where student.FirstName[0] == firstLetter
        select new { student.FirstName, student.LastName };

    // Execute the query and display the results.
    foreach (var anonType in studentQuery)
    {
        Console.WriteLine("First = {0}, Last = {1}", anonType.FirstName,
```

```
anonType.LastName);
    }
}
```

O exemplo a seguir usa a palavra-chave `var` em uma situação semelhante, mas na qual o uso de `var` é opcional. Como `student.LastName` é uma cadeia de caracteres, a execução da consulta retorna uma sequência de cadeias de caracteres. Portanto, o tipo de `queryId` poderia ser declarado como `System.Collections.Generic.IEnumerable<string>` em vez de `var`. A palavra-chave `var` é usada por conveniência. No exemplo, a variável de iteração na instrução `foreach` tem tipo explícito como uma cadeia de caracteres, mas, em vez disso, poderia ser declarada usando `var`. Como o tipo da variável de iteração não é um tipo anônimo, o uso de `var` é opcional, não obrigatório. Lembre-se de que `var` por si só não é um tipo, mas uma instrução para o compilador inferir e atribuir o tipo.

C#

```
// Variable queryId could be declared by using
// System.Collections.Generic.IEnumerable<string>
// instead of var.
var queryId =
    from student in students
    where student.Id > 111
    select student.LastName;

// Variable str could be declared by using var instead of string.
foreach (string str in queryId)
{
    Console.WriteLine("Last name: {0}", str);
}
```

Confira também

- Guia de Programação em C#
- Métodos de Extensão
- LINQ (Consulta Integrada à Linguagem)
- LINQ em C#

Membros de Extensão (Guia de Programação em C#)

Artigo • 01/05/2025

Os membros da extensão permitem que você "adicione" métodos aos tipos existentes sem criar um novo tipo derivado, recompilar ou modificar o tipo original.

A partir do C# 14, há duas sintaxes que você usa para definir métodos de extensão. O C# 14 adiciona extension contêineres, em que você define vários membros de extensão para um tipo ou uma instância de um tipo. Antes do C# 14, você adiciona o this modificador ao primeiro parâmetro de um método estático para indicar que o método aparece como um membro de uma instância do tipo de parâmetro.

Métodos de extensão são métodos estáticos, mas são chamados como se fossem métodos de instância no tipo estendido. Para o código do cliente escrito em C#, F# e Visual Basic, não há nenhuma diferença aparente entre chamar um método de extensão e os métodos definidos em um tipo. Ambas as formas de métodos de extensão são compiladas para o mesmo IL (Linguagem Intermediária). Os consumidores de membros da extensão não precisam saber qual sintaxe foi usada para definir métodos de extensão.

Os membros de extensão mais comuns são os operadores de consulta padrão LINQ que adicionam funcionalidade de consulta aos tipos System.Collections.IEnumerable e System.Collections.Generic.IEnumerable<T> existentes. Para usar os operadores de consulta padrão, primeiro coloque-os no escopo com uma diretiva `using System.Linq`. Em seguida, qualquer tipo que implementar IEnumerable<T> parece ter métodos de instância, como GroupBy, OrderBy, Average, e assim por diante. Você pode ver esses métodos extras no complemento de instrução do IntelliSense ao digitar "ponto" após uma instância de um tipo IEnumerable<T>, como List<T> ou Array.

Exemplo de OrderBy

O exemplo a seguir mostra como chamar o método de operador `OrderBy` de consulta padrão em uma matriz de inteiros. A expressão entre parênteses é uma expressão lambda. Muitos operadores de consulta padrão tomam expressões lambda como parâmetros. Para obter mais informações, consulte Expressões Lambda.

C#

```
int[] numbers = [10, 45, 15, 39, 21, 26];
IOrderedEnumerable<int> result = numbers.OrderBy(g => g);
foreach (int i in result)
{
```

```
    Console.Write(i + " ");
}
//Output: 10 15 21 26 39 45
```

Os métodos de extensão são definidos como métodos estáticos, mas são chamados usando a sintaxe do método de instância. Seu primeiro parâmetro especifica em qual tipo o método opera. O parâmetro segue este modificador. Os métodos de extensão só estão no escopo quando você importa explicitamente o namespace para o código-fonte com a diretiva `using`.

Declarar membros da extensão

A partir do C# 14, você pode declarar *blocos de extensão*. Um bloco de extensão é um bloco em uma classe estática não aninhada, não genérica que contém membros de extensão para um tipo ou uma instância desse tipo. O exemplo de código a seguir define um bloco de extensão para o `string` tipo. O bloco de extensão contém um membro: um método que conta as palavras na cadeia de caracteres:

C#

```
namespace CustomExtensionMembers;

public static class MyExtensions
{
    extension(string str)
    {
        public int WordCount() =>
            str.Split([' ', '.', '?'],
StringSplitOptions.RemoveEmptyEntries).Length;
    }
}
```

Antes do C# 14, você declara um método de extensão adicionando o `this` modificador ao primeiro parâmetro:

C#

```
namespace CustomExtensionMethods;

public static class MyExtensions
{
    public static int WordCount(this string str) =>
        str.Split([' ', '.', '?'], StringSplitOptions.RemoveEmptyEntries).Length;
}
```

Ambas as formas de extensões devem ser definidas dentro de uma classe estática não aninhada e não genérica.

E ele pode ser chamado de um aplicativo usando a sintaxe para acessar membros da instância:

```
C#

string s = "Hello Extension Methods";
int i = s.WordCount();
```

Embora os membros da extensão adicionem novos recursos a um tipo existente, os membros da extensão não violam o princípio do encapsulamento. As declarações de acesso para todos os membros do tipo estendido se aplicam aos membros da extensão.

A classe `MyExtensions` e o método `WordCount` são `static`, e podem ser acessados como todos os outros membros `static`. O `WordCount` método pode ser invocado como outros `static` métodos da seguinte maneira:

```
C#

string s = "Hello Extension Methods";
int i = MyExtensions.WordCount(s);
```

O código C# anterior se aplica tanto ao bloco de extensão quanto à sintaxe `this` para membros de extensão. O código anterior:

- Declara e atribui um novo `string` nome `s` com um valor de `"Hello Extension Methods"`.
- Chama `MyExtensions.WordCount` com o argumento dado `s`.

Para obter mais informações, consulte Como implementar e chamar um método de extensão personalizado.

Em geral, você provavelmente chama os membros da extensão com muito mais frequência do que os implementa. Como os membros da extensão são invocados como se fossem declarados como membros da classe estendida, nenhum conhecimento especial é necessário para usá-los pelo código cliente. Para habilitar membros de extensão para um tipo específico, basta adicionar uma `using` diretiva para o namespace no qual os métodos são definidos. Por exemplo, para usar os operadores de consulta padrão, adicione esta `using` diretiva ao seu código:

```
C#

using System.Linq;
```

Vincular membros de extensão durante a compilação

Você pode usar membros de extensão para estender uma classe ou interface, mas não para substituir o comportamento definido em uma classe. Um membro de extensão com o mesmo nome e assinatura que os de uma interface ou de uma classe nunca é chamado. No tempo de compilação, os membros de extensão sempre têm menor prioridade do que os membros de instância (ou estáticos) definidos no próprio tipo. Em outras palavras, se um tipo tiver um método chamado `Process(int i)` e você tiver um método de extensão com a mesma assinatura, o compilador sempre se associará ao método de membro. Quando o compilador encontra uma invocação de membro, ele primeiro procura um correspondente nos membros do tipo. Se nenhuma correspondência for encontrada, ela procurará todos os membros da extensão definidos para o tipo. Ele se associa ao primeiro membro de extensão encontrado. O exemplo a seguir demonstra as regras que o compilador C# segue para determinar se deve ser associado a um membro de instância no tipo ou a um membro de extensão. A classe `Extensions` estática contém membros de extensão definidos para qualquer tipo que implemente `IMyInterface`:

C#

```
public interface IMyInterface
{
    void MethodB();
}

// Define extension methods for IMyInterface.

// The following extension methods can be accessed by instances of any
// class that implements IMyInterface.
public static class Extension
{
    public static void MethodA(this IMyInterface myInterface, int i) =>
        Console.WriteLine("Extension.MethodA(this IMyInterface myInterface, int
i)");

    public static void MethodA(this IMyInterface myInterface, string s) =>
        Console.WriteLine("Extension.MethodA(this IMyInterface myInterface, string
s)");

    // This method is never called in ExtensionMethodsDemo1, because each
    // of the three classes A, B, and C implements a method named MethodB
    // that has a matching signature.
    public static void MethodB(this IMyInterface myInterface) =>
        Console.WriteLine("Extension.MethodB(this IMyInterface myInterface)");
}
```

As extensões equivalentes podem ser declaradas usando a sintaxe do membro da extensão C# 14:

C#

```
public static class Extension
{
    extension(IMyInterface myInterface)
    {
        public void MethodA(int i) =>
            Console.WriteLine("Extension.MethodA(this IMyInterface myInterface,
int i)");

        public void MethodA(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 void MethodB() =>
            Console.WriteLine("Extension.MethodB(this IMyInterface myInterface)");
    }
}
```

Classes `A` e `B` `C` todos implementam a interface:

C#

```
// Define three classes that implement IMyInterface, and then use them to test
// the extension methods.
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)");
    }
}
```

O `MethodB` método de extensão nunca é chamado porque seu nome e assinatura correspondem exatamente aos métodos já implementados pelas classes. Quando o compilador não consegue encontrar um método de instância com uma assinatura correspondente, ele se associa a um método de extensão correspondente se houver.

C#

```
// 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()
 */
```

Padrões de uso comuns

Funcionalidade de Coleção

No passado, era comum criar "Classes de Coleção" que implementavam a System.Collections.Generic.IEnumerable<T> interface para um determinado tipo e continham funcionalidades que atuavam em coleções desse tipo. Embora não haja nada de errado com a criação desse tipo de objeto de coleção, a mesma funcionalidade pode ser obtida usando uma extensão no System.Collections.Generic.IEnumerable<T>. As extensões têm a vantagem de permitir que a funcionalidade seja chamada de qualquer coleção, como uma System.Array ou System.Collections.Generic.List<T> que implementa System.Collections.Generic.IEnumerable<T> nesse tipo. Um exemplo disso usando uma Matriz de Int32 pode ser encontrado anteriormente neste artigo.

Funcionalidade do Layer-Specific

Ao usar uma Arquitetura onion ou outro design de aplicativo em camadas, é comum ter um conjunto de entidades de domínio ou objetos de transferência de dados que podem ser usados para se comunicar entre os limites do aplicativo. Esses objetos geralmente não contêm nenhuma funcionalidade ou apenas funcionalidade mínima que se aplica a todas as camadas do aplicativo. Os métodos de extensão podem ser usados para adicionar funcionalidades específicas a cada camada de aplicativo.

C#

```
public class DomainEntity
{
    public int Id { get; set; }
    public required string FirstName { get; set; }
    public required string LastName { get; set; }
}

static class DomainEntityExtensions
{
    static string FullName(this DomainEntity value)
        => $"{value.FirstName} {value.LastName}";
}
```

Você pode declarar uma propriedade equivalente `FullName` em C# 14 e posterior usando a nova sintaxe de bloco de extensão:

C#

```
static class DomainEntityExtensions
{
    extension(DomainEntity value)
    {
        string FullName => $"{value.FirstName} {value.LastName}";
    }
}
```

Estendendo tipos predefinidos

Em vez de criar novos objetos quando a funcionalidade reutilizável precisa ser criada, muitas vezes você pode estender um tipo existente, como um tipo .NET ou CLR. Por exemplo, se você não usar métodos de extensão, poderá criar uma `Engine` ou `Query` classe para executar uma consulta em um SQL Server que pode ser chamada de vários locais em nosso código. No entanto, você pode estender a System.Data.SqlClient.SqlConnection classe usando métodos de extensão para executar essa consulta de qualquer lugar que você tenha uma conexão com um SQL Server. Outros exemplos podem ser adicionar funcionalidades comuns à System.String classe, estender os recursos de processamento de dados do System.IO.Stream objeto e System.Exception objetos para funcionalidades específicas de tratamento de erros. Esses tipos de casos de uso são limitados apenas por sua imaginação e bom senso.

Estender tipos predefinidos pode ser difícil com `struct` tipos porque eles são passados por valor aos métodos. Isso significa que todas as alterações no struct são feitas em uma cópia do struct. Essas alterações não serão visíveis quando o método de extensão for encerrado. Você pode adicionar o `ref` modificador ao primeiro argumento tornando-o um `ref` método de extensão. A `ref` palavra-chave pode aparecer antes ou depois da `this` palavra-chave sem diferenças semânticas. Adicionar o `ref` modificador indica que o primeiro argumento é passado por referência. Essa técnica permite que você escreva métodos de extensão que alterem o estado do struct que está sendo estendido (observe que os membros privados não estão acessíveis). Somente tipos de valor ou tipos genéricos restritos ao struct (para obter mais informações sobre essas regras, consulte struct a restrição para obter mais informações) são permitidos como o primeiro parâmetro de um `ref` método de extensão ou como o receptor de um bloco de extensão. O exemplo a seguir mostra como usar um `ref` método de extensão para modificar diretamente um tipo interno sem a necessidade de reatribuir o resultado ou passá-lo por uma função com a `ref` palavra-chave:

C#

```
public static class IntExtensions
{
    public static void Increment(this int number)
        => number++;
```

```
    // Take note of the extra ref keyword here
    public static void RefIncrement(this ref int number)
        => number++;
}
```

Os blocos de extensão equivalentes são mostrados no seguinte código:

```C#
public static class IntExtensions
{
    extension(int number)
    {
        public void Increment()
            => number++;
    }

    // Take note of the extra ref keyword here
    extension(ref int number)
    {
        public void RefIncrement()
            => number++;
    }
}
```

Diferentes blocos de extensão são necessários para distinguir os modos de parâmetro "by-value" e "by-ref" para o receptor.

Você pode ver a diferença que a aplicação `ref` ao receptor tem no exemplo a seguir:

```C#
int x = 1;

// Takes x by value leading to the extension method
// Increment modifying its own copy, leaving x unchanged
x.Increment();
Console.WriteLine($"x is now {x}"); // x is now 1

// Takes x by reference leading to the extension method
// RefIncrement changing the value of x directly
x.RefIncrement();
Console.WriteLine($"x is now {x}"); // x is now 2
```

Você pode aplicar a mesma técnica adicionando `ref` membros de extensão aos tipos de estrutura definidos por usuários.

```C#
```

```
public struct Account
{
    public uint id;
    public float balance;

    private int secret;
}

public static class AccountExtensions
{
    // ref keyword can also appear before the this keyword
    public static void Deposit(ref this Account account, float amount)
    {
        account.balance += amount;

        // The following line results in an error as an extension
        // method is not allowed to access private members
        // account.secret = 1; // CS0122
    }
}
```

O exemplo anterior também pode ser criado usando blocos de extensão no C# 14:

C#

```
public static class AccountExtensions
{
    extension(ref Account account)
    {
        // ref keyword can also appear before the this keyword
        public void Deposit(float amount)
        {
            account.balance += amount;

            // The following line results in an error as an extension
            // method is not allowed to access private members
            // account.secret = 1; // CS0122
        }
    }
}
```

Você pode acessar esses métodos de extensão da seguinte maneira:

C#

```
Account account = new()
{
    id = 1,
    balance = 100f
};
```

```
Console.WriteLine($"I have ${account.balance}"); // I have $100

account.Deposit(50f);
Console.WriteLine($"I have ${account.balance}"); // I have $150
```

Diretrizes gerais

É preferível adicionar funcionalidade modificando o código de um objeto ou derivando um novo tipo sempre que for razoável e possível fazer isso. Os métodos de extensão são uma opção crucial para criar funcionalidade reutilizável em todo o ecossistema do .NET. Os membros da extensão são preferíveis quando a origem original não está sob seu controle, quando um objeto derivado é inadequado ou impossível, ou quando a funcionalidade tem escopo limitado.

Para obter mais informações sobre tipos derivados, consulte Herança.

Se você implementar métodos de extensão para um determinado tipo, lembre-se dos seguintes pontos:

- Um método de extensão não será chamado se ele tiver a mesma assinatura que um método definido no tipo.
- Os métodos de extensão são colocados no escopo no nível do namespace. Por exemplo, se você tiver várias classes estáticas que contêm métodos de extensão em um único namespace nomeado `Extensions`, todas elas serão colocadas no escopo pela `using Extensions;` diretiva.

Para uma biblioteca de classes que você implementou, você não deve usar métodos de extensão para evitar incrementar o número de versão de um assembly. Se você quiser adicionar uma funcionalidade significativa a uma biblioteca para a qual possui o código-fonte, siga as diretrizes do .NET para controle de versão do assembly. Para obter mais informações, consulte o Controle de Versão do Assembly.

Consulte também

- Exemplos de programação paralela (muitos exemplos demonstram métodos de extensão)
- Expressões Lambda
- Visão geral dos operadores de consulta padrão
- Regras de conversão para parâmetros de instância e seu impacto
- Interoperabilidade de métodos de extensão entre idiomas
- Métodos de extensão e delegados curried
- Método de extensão: associação e relatório de erros

Como implementar e chamar um método de extensão personalizado (Guia de Programação em C#)

Artigo • 04/10/2024

Este artigo mostra como implementar seus próprios métodos de extensão para qualquer tipo do .NET. O código do cliente pode usar seus métodos de extensão. Os projetos de cliente devem fazer referência ao assembly (montagem) nos quais estão contidos. Os projetos cliente devem adicionar uma diretriz de using (utilização) que especifica o namespace no qual os métodos de extensão são definidos.

Para definir e chamar o método de extensão;

1. Defina uma classe estática para conter o método de extensão. A classe não pode ser aninhada dentro de outro tipo e deve estar visível para o código do cliente. Para obter mais informações sobre regras de acessibilidade, consulte Modificadores de acesso.
2. Implemente o método de extensão como um método estático com, pelo menos, a mesma visibilidade da classe que a contém.
3. O primeiro parâmetro do método especifica o tipo no qual o método opera. Ele deve ser precedido pelo modificador this.
4. No código de chamada, adicione uma diretiva `using` para especificar o namespace que contém a classe do método de extensão.
5. Chame os métodos como métodos de instância no tipo.

> **Observação**
>
> O primeiro parâmetro não é especificado pelo código de chamada porque ele representa o tipo no qual o operador está sendo aplicado e o compilador já conhece o tipo do objeto. Você só precisa fornecer argumentos para os parâmetros de 2 até o `n`.

O exemplo a seguir implementa um método de extensão chamado `WordCount` na classe `CustomExtensions.StringExtension`. O método funciona na classe String, que é especificada como o primeiro parâmetro do método. O namespace `CustomExtensions` é importado para o namespace do aplicativo e o método é chamado dentro do método `Main`.

C#

```
using CustomExtensions;

string s = "The quick brown fox jumped over the lazy dog.";
// Call the method as if it were an
// instance method on the type. Note that the first
// parameter is not specified by the calling code.
int i = s.WordCount();
System.Console.WriteLine("Word count of s is {0}", i);


namespace CustomExtensions
{
    // Extension methods must be defined in a static class.
    public static class StringExtension
    {
        // This is the extension method.
        // The first parameter takes the "this" modifier
        // and specifies the type for which the method is defined.
        public static int WordCount(this string str)
        {
            return str.Split(new char[] {' ', '.','?'},
StringSplitOptions.RemoveEmptyEntries).Length;
        }
    }
}
```

A resolução de sobrecarga dá preferência ao método de instância ou estático definido pelo próprio tipo antes que métodos de extensão. Os métodos de extensão não podem acessar nenhum dado particular na classe estendida.

Confira também

- Métodos de Extensão
- LINQ (Consulta Integrada à Linguagem)
- Classes static e membros de classes static
- protected
- interno
- público
- this
- namespace

Como criar um novo método para uma enumeração (Guia de Programação em C#)

Artigo • 30/04/2025

Você pode usar métodos de extensão para adicionar funcionalidades específicas a um tipo de enumeração específico. No exemplo a seguir, a enumeração `Grades` representa as notas em letras possíveis que um aluno pode receber em uma classe. Um método de extensão chamado `Passing` é adicionado ao tipo `Grades`, de modo que cada instância desse tipo agora saiba se representa uma nota de aprovação ou não.

C#

```
public enum Grades
{
    F = 0,
    D = 1,
    C = 2,
    B = 3,
    A = 4
};

// Define an extension method in a non-nested static class.
public static class Extensions
{
    public static bool Passing(this Grades grade, Grades minPassing = Grades.D) =>
        grade >= minPassing;
}
```

Você pode chamar o método de extensão como se ele fosse declarado no `enum` tipo:

C#

```
Grades g1 = Grades.D;
Grades g2 = Grades.F;
Console.WriteLine($"First {(g1.Passing() ? "is" : "is not")} a passing grade.");
Console.WriteLine($"Second {(g2.Passing() ? "is" : "is not")} a passing grade.");

Console.WriteLine("\r\nRaising the bar!\r\n");
Console.WriteLine($"First {(g1.Passing(Grades.C) ? "is" : "is not")} a passing
grade.");
Console.WriteLine($"Second {(g2.Passing(Grades.C) ? "is" : "is not")} a passing
grade.");
/* 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.
*/
```

A partir do C# 14, você pode declarar *membros de extensão* em um bloco de extensão. A nova sintaxe permite que você adicione propriedades de *extensão*. Você também pode adicionar membros de extensão que parecem ser novos métodos estáticos ou propriedades. Você não está mais limitado a extensões que parecem ser métodos de instância. O exemplo a seguir mostra um bloco de extensão que adiciona uma propriedade de extensão de instância para `Passing` e uma propriedade de extensão estática para `MinimumPassingGrade`:

C#

```
public static class EnumExtensions
{
    private static Grades minimumPassingGrade = Grades.D;

    extension(Grades grade)
    {
        public static Grades MinimumPassingGrade
        {
            get => minimumPassingGrade;
            set => minimumPassingGrade = value;
        }

        public bool Passing => grade >= minimumPassingGrade;
    }
}
```

Você chama essas novas propriedades de extensão como se elas fossem declaradas no tipo estendido:

C#

```
public static class EnumExtensions
{
    private static Grades minimumPassingGrade = Grades.D;

    extension(Grades grade)
    {
        public static Grades MinimumPassingGrade
        {
            get => minimumPassingGrade;
            set => minimumPassingGrade = value;
        }

        public bool Passing => grade >= minimumPassingGrade;
    }
}
```

Você pode saber mais sobre os novos membros da extensão no artigo sobre membros da extensão e no artigo de referência da linguagem na palavra-chave "extensão".

Consulte também

- Métodos de Extensão

Argumentos nomeados e opcionais (Guia de Programação em C#)

Artigo • 29/01/2025

Os argumentos nomeados permitem que você especifique um argumento para um parâmetro correspondendo o argumento com seu nome em vez de com sua posição na lista de parâmetros. *Argumentos opcionais* permitem omitir argumentos para alguns parâmetros. Ambas as técnicas podem ser usadas com os métodos, indexadores, construtores e delegados.

Quando você usa argumentos nomeados e opcionais, os argumentos são avaliados na ordem em que aparecem na lista de argumentos e não na lista de parâmetros.

Parâmetros nomeados e opcionais permitem que você forneça argumentos para parâmetros selecionados. Essa capacidade facilita bastante chamadas para interfaces COM como as APIs de Automação do Microsoft Office.

Argumentos nomeados

Os argumentos nomeados liberam você da necessidade de combinar a ordem dos argumentos com a ordem dos parâmetros nas listas de parâmetros de métodos chamados. O argumento para cada parâmetro pode ser especificado pelo nome do parâmetro. Por exemplo, uma função que imprime detalhes do pedido (como nome do vendedor, número do pedido e nome do produto) pode ser chamada enviando argumentos por posição, na ordem definida pela função.

C#

```
PrintOrderDetails("Gift Shop", 31, "Red Mug");
```

Se não se lembrar da ordem dos parâmetros, mas souber os nomes, você poderá enviar os argumentos em qualquer ordem.

C#

```
PrintOrderDetails(orderNum: 31, productName: "Red Mug", sellerName: "Gift
Shop");
PrintOrderDetails(productName: "Red Mug", sellerName: "Gift Shop", orderNum:
31);
```

Os argumentos nomeados também melhoram a legibilidade do código identificando o que cada argumento representa. No método de exemplo abaixo, o `sellerName` não pode ser nulo ou um espaço em branco. Como `sellerName` e `productName` são tipos de cadeia de caracteres, em vez de enviar argumentos por posição, é melhor usar argumentos nomeados para remover a ambiguidade dos dois e reduzir a confusão para qualquer pessoa que leia o código.

Os argumentos nomeados, quando usados com argumentos posicionais, são válidos, desde que

- não sejam seguidos por argumentos posicionais ou,

  C#
  ```
  PrintOrderDetails("Gift Shop", 31, productName: "Red Mug");
  ```

- são usados na posição correta. No exemplo a seguir, o parâmetro `orderNum` está na posição correta, mas não está explicitamente nomeado.

  C#
  ```
  PrintOrderDetails(sellerName: "Gift Shop", 31, productName: "Red Mug");
  ```

Argumentos posicionais que seguem argumentos nomeados fora de ordem são inválidos.

C#
```
// This generates CS1738: Named argument specifications must appear after
all fixed arguments have been specified.
PrintOrderDetails(productName: "Red Mug", 31, "Gift Shop");
```

Exemplo

O código a seguir implementa os exemplos desta seção, juntamente com outros exemplos.

C#
```
class NamedExample
{
    static void Main(string[] args)
    {
        // The method can be called in the normal way, by using positional
```

```
arguments.
        PrintOrderDetails("Gift Shop", 31, "Red Mug");

        // Named arguments can be supplied for the parameters in any order.
        PrintOrderDetails(orderNum: 31, productName: "Red Mug", sellerName:
"Gift Shop");
        PrintOrderDetails(productName: "Red Mug", sellerName: "Gift Shop",
orderNum: 31);

        // Named arguments mixed with positional arguments are valid
        // as long as they are used in their correct position.
        PrintOrderDetails("Gift Shop", 31, productName: "Red Mug");
        PrintOrderDetails(sellerName: "Gift Shop", 31, productName: "Red
Mug");
        PrintOrderDetails("Gift Shop", orderNum: 31, "Red Mug");

        // However, mixed arguments are invalid if used out-of-order.
        // The following statements will cause a compiler error.
        // PrintOrderDetails(productName: "Red Mug", 31, "Gift Shop");
        // PrintOrderDetails(31, sellerName: "Gift Shop", "Red Mug");
        // PrintOrderDetails(31, "Red Mug", sellerName: "Gift Shop");
    }

    static void PrintOrderDetails(string sellerName, int orderNum, string
productName)
    {
        if (string.IsNullOrWhiteSpace(sellerName))
        {
            throw new ArgumentException(message: "Seller name cannot be null
or empty.", paramName: nameof(sellerName));
        }

        Console.WriteLine($"Seller: {sellerName}, Order #: {orderNum},
Product: {productName}");
    }
}
```

Argumentos opcionais

A definição de um método, construtor, indexador ou delegado pode especificar que seus parâmetros são obrigatórios ou opcionais. Qualquer chamada deve fornecer argumentos para todos os parâmetros necessários, mas pode omitir argumentos para parâmetros opcionais. Um tipo de referência anulável (`T?`) permite que os argumentos sejam explicitamente `null`, mas não torna inerentemente um parâmetro opcional.

Cada parâmetro opcional tem um valor padrão como parte de sua definição. Se nenhum argumento é enviado para esse parâmetro, o valor padrão é usado. Um valor padrão deve ser um dos seguintes tipos de expressões:

- uma expressão de constante;
- uma expressão da forma `new ValType()`, em que `ValType` é um tipo de valor, como um enum ou um struct;
- uma expressão da forma default(ValType), em que `ValType` é um tipo de valor.

Os parâmetros opcionais são definidos no final da lista de parâmetros, depois de todos os parâmetros obrigatórios. O chamador deve fornecer argumentos para todos os parâmetros necessários e quaisquer parâmetros opcionais anteriores aos especificados por ele. Não há suporte para lacunas separadas por vírgulas na lista de argumentos. Por exemplo, no código a seguir, o método de instância `ExampleMethod` é definido com um parâmetro obrigatório e dois opcionais.

C#

```
public void ExampleMethod(int required, string optionalstr = "default
string",
    int optionalint = 10)
```

A chamada para `ExampleMethod` a seguir causa um erro do compilador, porque um argumento é fornecido para o terceiro parâmetro, mas não para o segundo.

C#

```
// anExample.ExampleMethod(3, ,4);
```

No entanto, se você souber o nome do terceiro parâmetro, poderá usar um argumento nomeado para realizar a tarefa.

C#

```
anExample.ExampleMethod(3, optionalint: 4);
```

O IntelliSense usa colchetes para indicar parâmetros opcionais, conforme mostrado na seguinte ilustração:

```
anExample.ExampleMethod(
  void ExampleClass.ExampleMethod(int required,
                                  [string optionalstr = "default string"],
                                  [int optionalint = 10])
```

Observação

Você também pode declarar parâmetros opcionais usando a classe OptionalAttribute do .NET. Os parâmetros `OptionalAttribute` não exigem um valor padrão. No entanto, se um valor padrão for desejado, dê uma olhada na classe DefaultParameterValueAttribute.

Exemplo

No exemplo a seguir, o construtor para `ExampleClass` tem um parâmetro, que é opcional. O método de instância `ExampleMethod` tem um parâmetro obrigatório, `required` e dois parâmetros opcionais, `optionalstr` e `optionalint`. O código em `Main` mostra as diferentes maneiras em que o construtor e o método podem ser invocados.

C#

```
namespace OptionalNamespace
{
    class OptionalExample
    {
        static void Main(string[] args)
        {
            // Instance anExample does not send an argument for the
constructor's
            // optional parameter.
            ExampleClass anExample = new ExampleClass();
            anExample.ExampleMethod(1, "One", 1);
            anExample.ExampleMethod(2, "Two");
            anExample.ExampleMethod(3);

            // Instance anotherExample sends an argument for the
constructor's
            // optional parameter.
            ExampleClass anotherExample = new ExampleClass("Provided name");
            anotherExample.ExampleMethod(1, "One", 1);
            anotherExample.ExampleMethod(2, "Two");
            anotherExample.ExampleMethod(3);

            // The following statements produce compiler errors.

            // An argument must be supplied for the first parameter, and it
            // must be an integer.
            //anExample.ExampleMethod("One", 1);
            //anExample.ExampleMethod();

            // You cannot leave a gap in the provided arguments.
            //anExample.ExampleMethod(3, ,4);
            //anExample.ExampleMethod(3, 4);

            // You can use a named parameter to make the previous
            // statement work.
```

```
            anExample.ExampleMethod(3, optionalint: 4);
        }
    }

    class ExampleClass
    {
        private string _name;

        // Because the parameter for the constructor, name, has a default
        // value assigned to it, it is optional.
        public ExampleClass(string name = "Default name")
        {
            _name = name;
        }

        // The first parameter, required, has no default value assigned
        // to it. Therefore, it is not optional. Both optionalstr and
        // optionalint have default values assigned to them. They are
optional.
        public void ExampleMethod(int required, string optionalstr =
"default string",
            int optionalint = 10)
        {
            Console.WriteLine(
                $"{_name}: {required}, {optionalstr}, and {optionalint}.");
        }
    }

    // The output from this example is the following:
    // Default name: 1, One, and 1.
    // Default name: 2, Two, and 10.
    // Default name: 3, default string, and 10.
    // Provided name: 1, One, and 1.
    // Provided name: 2, Two, and 10.
    // Provided name: 3, default string, and 10.
    // Default name: 3, default string, and 4.
}
```

O código anterior ilustra vários casos em que os parâmetros opcionais são usados corretamente e incorretamente. Os argumentos devem ser fornecidos para todos os parâmetros necessários. As lacunas em argumentos opcionais devem ser preenchidas com parâmetros nomeados.

Atributos de informações do chamador

Atributosde informações do chamador, como CallerFilePathAttribute, CallerLineNumberAttribute, CallerMemberNameAttribute e CallerArgumentExpressionAttribute, são usados para obter informações sobre o

chamador de um método. Esses atributos são especialmente úteis quando você está depurando ou quando precisa registrar informações sobre chamadas de método.

Esses atributos são parâmetros opcionais com valores padrão fornecidos pelo compilador. O chamador não deve fornecer explicitamente um valor para esses parâmetros.

Interfaces COM

Os argumentos nomeados e opcionais, juntamente com suporte para objetos dinâmicos e outros aprimoramentos, aprimoram enormemente a interoperabilidade com APIs COM, como APIs de Automação do Office.

Por exemplo, o método AutoFormat na interface Range do Microsoft Office Excel tem sete parâmetros, todos opcionais. Esses parâmetros são mostrados na seguinte ilustração:

```
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])
```

No entanto, você pode simplificar muito a chamada para `AutoFormat` usando argumentos nomeados e opcionais. Os argumentos nomeados e opcionais permitem que você omita o argumento para um parâmetro opcional se não desejar alterar o valor padrão do parâmetro. Na chamada a seguir, um valor é especificado para apenas um dos sete parâmetros.

C#

```
var excelApp = new Microsoft.Office.Interop.Excel.Application();
excelApp.Workbooks.Add();
excelApp.Visible = true;

var myFormat =

Microsoft.Office.Interop.Excel.XlRangeAutoFormat.xlRangeAutoFormatAccounting
1;

excelApp.Range["A1", "B4"].AutoFormat( Format: myFormat );
```

Para obter mais informações e exemplos, consulte Como usar argumentos nomeados e opcionais na programação do Office e Como acessar objetos de interoperabilidade do Office usando recursos do Visual C#.

Resolução de sobrecarga

O uso de argumentos nomeados e opcionais afeta a resolução de sobrecarga das seguintes maneiras:

- Um método, indexador ou construtor é um candidato para a execução se cada um dos parâmetros é opcional ou corresponde, por nome ou posição, a um único argumento na instrução de chamada e esse argumento pode ser convertido para o tipo do parâmetro.
- Se mais de um candidato for encontrado, as regras de resolução de sobrecarga de conversões preferenciais serão aplicadas aos argumentos que são especificados explicitamente. Os argumentos omitidos para parâmetros opcionais são ignorados.
- Se dois candidatos são considerados igualmente bons, a preferência vai para um candidato que não tenha parâmetros opcionais para os quais argumentos foram omitidos na chamada. A resolução de sobrecarga geralmente prefere candidatos com menos parâmetros.

Especificação da linguagem C#

Para obter mais informações, consulte a Especificação da linguagem C#. A especificação da linguagem é a fonte definitiva para a sintaxe e o uso de C#.

Construtores (guia de programação em C#)

19/06/2025

Um *construtor* é um método chamado pelo runtime quando é criada uma instância de uma classe ou de uma struct. Uma classe ou struct pode ter vários construtores que têm argumentos diferentes. Os construtores permitem que você garanta que as instâncias do tipo sejam válidas quando criadas. Para obter mais informações e exemplos, consulte Construtores de Instância e Usando Construtores.

Há várias ações que fazem parte da inicialização de uma nova instância. As seguintes ações ocorrem na seguinte ordem:

1. *Campos de instância são definidos como 0*. Essa inicialização normalmente é feita pelo runtime.
2. *Inicializadores de campo são executados*. Os inicializadores de campo na execução de tipo mais derivada.
3. *Inicializadores de campo de tipo base são executados*. Inicializadores de campo começando com a base direta por meio de cada tipo base para System.Object.
4. *Construtores de instância base são executados*. Todos os construtores de instância, começando por Object.Object, em cada classe base para a classe base direta.
5. *O construtor de instância é executado*. O construtor de instância para o tipo é executado.
6. *Inicializadores de objeto são executados*. Se a expressão incluir os inicializadores de objeto, eles são executados após a execução do construtor de instância. Inicializadores de objeto são executados na ordem textual.

As ações anteriores ocorrem quando uma instância é criada usando o operador new. Se uma nova instância de um `struct` for definida como seu valor `default`, todos os campos de instância serão definidos como 0. Os elementos de uma matriz são definidos como o valor padrão de 0 ou `null` quando uma matriz é criada.

O construtor estático, se houver, é executado antes que qualquer ação do construtor de instância ocorra para qualquer instância do tipo. O construtor estático é executado no máximo uma vez.

A partir do C# 14, você pode declarar construtores de instância como membros parciais. Construtores parciais devem ter uma declaração de definição e uma declaração de implementação.

Sintaxe do construtor

Um construtor é um método com o mesmo nome que seu tipo. A assinatura do método pode incluir um modificador de acesso opcional , o nome do método e sua lista de parâmetros; ela não inclui um tipo de retorno. O exemplo a seguir mostra o construtor de uma classe chamada `Person`.

C#

```
public class Person
{
   private string last;
   private string first;

   public Person(string lastName, string firstName)
   {
      last = lastName;
      first = firstName;
   }

   // Remaining implementation of Person class.
}
```

Se um construtor puder ser implementado como uma única instrução, você poderá usar uma membro do corpo da expressão. O exemplo a seguir define uma `Location` classe cujo construtor tem um único parâmetro de cadeia de caracteres. `name` A definição do corpo da expressão atribui o argumento ao campo `locationName`.

C#

```
public class Location
{
   private string locationName;

   public Location(string name) => Name = name;

   public string Name
   {
      get => locationName;
      set => locationName = value;
   }
}
```

Se um tipo exigir um parâmetro para criar uma instância, você poderá usar um construtor primário para indicar que um ou mais parâmetros são necessários para instanciar o tipo, conforme mostrado no exemplo a seguir:

C#

```
public class LabelledContainer<T>(string label)
{
   public string Label { get; } = label;
   public required T Contents
   {
      get;
      init;
   }
}
```

Você pode declarar um construtor primário em um `partial` tipo. Somente uma declaração de construtor primário é permitida. Em outras palavras, apenas uma declaração do `partial` tipo pode incluir os parâmetros para o construtor primário.

Construtores estáticos

Os exemplos anteriores mostram construtores de instância, que inicializam um novo objeto. Uma classe ou struct também pode declarar um construtor estático, que inicializa membros estáticos do tipo. Construtores estáticos não têm parâmetros. Se você não fornecer um construtor estático para inicializar campos estáticos, o compilador C# inicializará campos estáticos para o valor padrão, conforme listado no artigo Valores padrão dos tipos C#.

O exemplo a seguir usa um construtor estático para inicializar um campo estático.

C#

```
public class Adult : Person
{
   private static int minimumAge;

   public Adult(string lastName, string firstName) : base(lastName, firstName)
   { }

   static Adult() => minimumAge = 18;

   // Remaining implementation of Adult class.
}
```

Você também pode definir um construtor estático com uma definição de corpo de expressão, como mostra o exemplo a seguir.

C#

```
public class Child : Person
{
   private static int maximumAge;
```

```
    public Child(string lastName, string firstName) : base(lastName, firstName)
    { }

    static Child() => maximumAge = 18;

    // Remaining implementation of Child class.
}
```

Para obter mais informações e exemplos, consulte Construtores Estáticos.

Construtores parciais

A partir do C# 14, você pode declarar *construtores parciais* em uma classe ou struct parcial. Qualquer construtor parcial deve ter uma *declaração de definição* e uma *declaração de implementação*. As assinaturas dos construtores parciais que declaram e implementam devem corresponder de acordo com as regras dos membros parciais. A declaração de definição do construtor parcial não pode usar o inicializador `: base()` ou o construtor `: this()`. Qualquer inicializador de construtor deve ser adicionado na declaração de implementação.

Consulte também

- o sistema de tipo C#
- static
- Por que os inicializadores são executados na ordem oposta, como construtores? Parte Um

Usar construtores (guia de programação em C#)

Artigo • 20/01/2025

Quando uma classe ou struct é instanciada, o runtime chama seu construtor. Os construtores têm o mesmo nome que a classe ou struct e eles geralmente inicializam os membros de dados do novo objeto.

No exemplo a seguir, uma classe chamada `Taxi` é definida usando um construtor simples. Essa classe é então instanciada com o operador new. O runtime invoca o construtor `Taxi` imediatamente depois que a memória é alocada para o novo objeto.

C#

```
public class Taxi
{
    private string taxiTag;

    public Taxi(string tag) => taxiTag = tag;

    public override string ToString() => $"Taxi: {taxiTag}";
}

class TestTaxi
{
    static void Main()
    {
        Taxi t = new Taxi("Tag1345");
        Console.WriteLine(t);
    }
}
```

Um construtor que não tem nenhum parâmetro é chamado de *construtor sem parâmetros*. O runtime invoca o construtor sem parâmetros quando um objeto é instanciado usando o operador `new` e nenhum argumento é fornecido para `new`. O C# 12 introduziu *construtores primários*. Um construtor primário especifica parâmetros que devem ser fornecidos para inicializar um novo objeto. Para obter mais informações, consulte Construtores de instâncias.

A menos que a classe seja static, as classes sem construtores recebem um construtor sem parâmetros público pelo compilador C# para habilitar a instanciação de classe. Para obter mais informações, consulte Classes estáticas e membros de classes estáticas.

Você pode impedir que uma classe seja instanciada tornando o construtor privado, da seguinte maneira:

```
C#

class NLog
{
    // Private Constructor:
    private NLog() { }

    public static double e = Math.E;  //2.71828...
}
```

Para obter mais informações, consulte Construtores particulares.

Construtores para tipos struct se assemelham a construtores de classe. Quando um tipo de struct é instanciado com `new`, o runtime invoca um construtor. Quando um `struct` é definido como seu valor `default`, o runtime inicializa toda a memória no struct como 0. Se declarar os inicializadores de campo em um tipo de `struct`, você deverá fornecer um construtor sem parâmetros. Para obter mais informações, consulte a seção Inicialização de struct e valores padrão do artigo Tipos de estrutura.

O código a seguir usa o construtor sem parâmetros para Int32, para que você tenha certeza de que o inteiro é inicializado:

```
C#

int i = new int();
Console.WriteLine(i);
```

O código a seguir, no entanto, causa um erro do compilador porque ele não usa `new` e porque tenta usar um objeto que não é inicializado:

```
C#

int i;
Console.WriteLine(i);
```

Como alternativa, alguns tipos de `struct` (incluindo todos os tipos numéricos internos) podem ser inicializados ou atribuídos e, em seguida, usados como no exemplo a seguir:

```
C#

int a = 44;  // Initialize the value type...
int b;
```

```
b = 33;      // Or assign it before using it.
Console.WriteLine("{0}, {1}", a, b);
```

Classes e structs podem definir construtores que recebem parâmetros, incluindo construtores primários. Os construtores que usam parâmetros devem ser chamados por meio de uma instrução `new` ou uma instrução base. As classes e structs também podem definir vários construtores e nenhum deles precisa definir um construtor sem parâmetros. Por exemplo:

C#

```
public class Employee
{
    public int Salary;

    public Employee() { }

    public Employee(int annualSalary) => Salary = annualSalary;

    public Employee(int weeklySalary, int numberOfWeeks) => Salary =
weeklySalary * numberOfWeeks;
}
```

Essa classe pode ser criada usando qualquer uma das instruções a seguir:

C#

```
Employee e1 = new Employee(30000);
Employee e2 = new Employee(500, 52);
```

Um construtor pode usar a palavra-chave `base` para chamar o construtor de uma classe base. Por exemplo:

C#

```
public class Manager : Employee
{
    public Manager(int annualSalary)
        : base(annualSalary)
    {
        //Add further instructions here.
    }
}
```

Neste exemplo, o construtor da classe base é chamado antes que o bloco do construtor seja executado. A palavra-chave `base` pode ser usada com ou sem parâmetros. Os

parâmetros para o construtor podem ser usados como parâmetros para `base` ou como parte de uma expressão. Para obter mais informações, consulte base.

Em uma classe derivada, se um construtor de classe base não for chamado explicitamente usando a palavra-chave `base`, o construtor sem parâmetros, se houver, será chamado implicitamente. As seguintes declarações de construtor são efetivamente iguais:

C#

```
public Manager(int initialData)
{
    //Add further instructions here.
}
```

C#

```
public Manager(int initialData)
    : base()
{
    //Add further instructions here.
}
```

Se uma classe base não oferecer um construtor sem parâmetros, a classe derivada deverá fazer uma chamada explícita para um construtor base usando `base`.

Um construtor pode invocar outro construtor no mesmo objeto usando a palavra-chave this. Como `base`, `this` pode ser usado com ou sem parâmetros e todos os parâmetros no construtor estão disponíveis como parâmetros para `this` ou como parte de uma expressão. Por exemplo, o segundo construtor no exemplo anterior pode ser reescrito usando `this`:

C#

```
public Employee(int weeklySalary, int numberOfWeeks)
    : this(weeklySalary * numberOfWeeks)
{
}
```

O uso da palavra-chave `this` no exemplo anterior faz com que o seguinte construtor seja chamado:

C#

```
public Employee(int annualSalary) => Salary = annualSalary;
```

Os construtores podem ser marcados como public, private, protected, interno, protegido interno ou private protected. Esses modificadores de acesso definem como os usuários da classe podem construir a classe. Para obter mais informações, consulte Modificadores de Acesso.

Um construtor pode ser declarado estático usando a palavra-chave static. Construtores estáticos são chamados automaticamente, antes que quaisquer campos estáticos sejam acessados e sejam usados para inicializar membros de classe estática. Para obter mais informações, consulte Construtores estáticos.

Especificação da Linguagem C#

Para obter mais informações, veja Construtores de instância e Construtores estáticos na Especificação de Linguagem C#. A especificação da linguagem é a fonte definitiva para a sintaxe e o uso de C#.

Confira também

- Sistema do tipo C#
- Construtores
- Finalizadores

Construtores de instância (guia de programação em C#)

19/06/2025

Você declara um construtor de instância para especificar o código executado ao criar uma nova instância de um tipo com a new expressão. Para inicializar uma classe estática ou variáveis estáticas em uma classe não estática, você pode definir um construtor estático.

Como mostra o exemplo a seguir, você pode declarar vários construtores de instância em um tipo:

C#

```
class Coords
{
    public Coords()
        : this(0, 0)
    {  }

    public Coords(int x, int y)
    {
        X = x;
        Y = y;
    }

    public int X { get; set; }
    public int Y { get; set; }

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

class Example
{
    static void Main()
    {
        var p1 = new Coords();
        Console.WriteLine($"Coords #1 at {p1}");
        // Output: Coords #1 at (0,0)

        var p2 = new Coords(5, 3);
        Console.WriteLine($"Coords #2 at {p2}");
        // Output: Coords #2 at (5,3)
    }
}
```

No exemplo anterior, o primeiro construtor, sem parâmetros, chama o segundo construtor com ambos os argumentos iguais `0`. Para fazer isso, use a `this` palavra-chave.

Ao declarar um construtor de instância em uma classe derivada, você pode chamar um construtor de uma classe base. Para fazer isso, use a `base` palavra-chave, como mostra o exemplo a seguir:

C#

```
abstract class Shape
{
    public const double pi = Math.PI;
    protected double x, y;

    public Shape(double x, double y)
    {
        this.x = x;
        this.y = y;
    }

    public abstract double Area();
}

class Circle : Shape
{
    public Circle(double radius)
        : base(radius, 0)
    {  }

    public override double Area() => pi * x * x;
}

class Cylinder : Circle
{
    public Cylinder(double radius, double height)
        : base(radius)
    {
        y = height;
    }

    public override double Area() => (2 * base.Area()) + (2 * pi * x * y);
}

class Example
{
    static void Main()
    {
        double radius = 2.5;
        double height = 3.0;

        var ring = new Circle(radius);
        Console.WriteLine($"Area of the circle = {ring.Area():F2}");
        // Output: Area of the circle = 19.63

        var tube = new Cylinder(radius, height);
        Console.WriteLine($"Area of the cylinder = {tube.Area():F2}");
```

```
        // Output: Area of the cylinder = 86.39
    }
}
```

Construtores sem parâmetros

Se uma *classe* não tiver construtores de instância explícitos, o C# fornecerá um construtor sem parâmetros que você pode usar para instanciar uma instância dessa classe, como mostra o exemplo a seguir:

C#

```
public class Person
{
    public int age;
    public string name = "unknown";
}

class Example
{
    static void Main()
    {
        var person = new Person();
        Console.WriteLine($"Name: {person.name}, Age: {person.age}");
        // Output:  Name: unknown, Age: 0
    }
}
```

Esse construtor inicializa os campos e as propriedades da instância de acordo com os inicializadores correspondentes. Se um campo ou propriedade não tiver um inicializador, seu valor será definido como o valor padrão do tipo do campo ou da propriedade. Se você declarar pelo menos um construtor de instância em uma classe, o C# não fornecerá um construtor sem parâmetros.

Um tipo *de estrutura* sempre fornece um construtor sem parâmetros. O construtor sem parâmetros é um construtor implícito sem parâmetros que produz o valor padrão de um tipo ou um construtor explicitamente declarado sem parâmetros. Para obter mais informações, consulte a seção inicialização do Struct e valores padrão do artigo Tipos de estrutura .

Construtores primários

A partir do C# 12, você pode declarar um *construtor primário* em classes e structs. Você coloca todos os parâmetros entre parênteses seguindo o nome do tipo:

C#

```
public class NamedItem(string name)
{
    public string Name => name;
}
```

Os parâmetros para um construtor primário estão no escopo em todo o corpo do tipo de declaração. Eles podem inicializar propriedades ou campos. Elas podem ser usadas como variáveis em métodos ou funções locais. Eles podem ser passados para um construtor base.

Um construtor primário indica que esses parâmetros são necessários para qualquer instância do tipo. Qualquer construtor escrito explicitamente deve usar a sintaxe do `this(...)` inicializador para invocar o construtor primário. Isso garante que os parâmetros do construtor primário sejam definitivamente atribuídos por todos os construtores. Para qualquer `class` tipo, incluindo `record class` tipos, o construtor implícito sem parâmetros não é emitido quando um construtor primário está presente. Para qualquer tipo `struct`, incluindo tipos `record struct`, o construtor implícito sem parâmetros é sempre emitido e inicializa todos os campos, incluindo os parâmetros do construtor primário, com o padrão de bits 0. Se você escrever um construtor sem parâmetros explícito, ele deverá invocar o construtor primário. Nesse caso, você pode especificar um valor diferente para os parâmetros do construtor primário. O código a seguir mostra exemplos de construtores primários.

C#

```
// name isn't captured in Widget.
// width, height, and depth are captured as private fields
public class Widget(string name, int width, int height, int depth) :
NamedItem(name)
{
    public Widget() : this("N/A", 1,1,1) {} // unnamed unit cube

    public int WidthInCM => width;
    public int HeightInCM => height;
    public int DepthInCM => depth;

    public int Volume => width * height * depth;
}
```

Você pode adicionar atributos ao método de construtor primário sintetizado especificando o `method:` destino no atributo:

C#

```
[method: MyAttribute]
public class TaggedWidget(string name)
{
```

```
    // details elided
}
```

Se você não especificar o `method` destino, o atributo será colocado na classe em vez do método.

Nos tipos `class` e `struct`, os parâmetros do construtor principal estão disponíveis em qualquer lugar no corpo do tipo. O parâmetro pode ser implementado como um campo privado capturado. Se as únicas referências a um parâmetro forem inicializadores e chamadas de construtor, esse parâmetro não será capturado em um campo privado. Os usos em outros membros do tipo fazem com que o compilador capture o parâmetro em um campo privado.

Se o tipo incluir o modificador `record`, o compilador sintetizará uma propriedade pública com o mesmo nome que o parâmetro do construtor primário. Para tipos `record class`, se um parâmetro de construtor primário tiver o mesmo nome de um construtor primário base, essa propriedade é pública no tipo base `record class`. Ele não é duplicado no tipo derivado `record class`. Essas propriedades não são geradas para tipos que não sejam `record`.

Consulte também

- Classes, estruturas e recordes
- Construtores
- Finalizadores
- base
- este
- Especificação de recurso de construtores primários

Construtores privados (Guia de Programação em C#)

19/06/2025

Um construtor privado é um construtor de instância especial. Geralmente, ele é usado em classes que contêm apenas membros estáticos. Se uma classe tiver um ou mais construtores privados e nenhum construtor público, outras classes (exceto classes aninhadas) não poderão criar instâncias dessa classe. Por exemplo:

C#

```
class NLog
{
    // Private Constructor:
    private NLog() { }

    public static double e = Math.E;  //2.71828...
}
```

A declaração do construtor vazio impede a geração automática de um construtor sem parâmetros. Observe que, se você não usar um modificador de acesso com o construtor, ele ainda será privado por padrão. No entanto, o modificador privado geralmente é usado explicitamente para deixar claro que a classe não pode ser instanciada.

Construtores privados são usados para impedir a criação de instâncias de uma classe quando não há campos ou métodos de instância, como a Math classe, ou quando um método é chamado para obter uma instância de uma classe. Se todos os métodos da classe forem estáticos, considere tornar a classe completa estática. Para obter mais informações, consulte Classes estáticas e membros de classe estática.

Exemplo

Veja a seguir um exemplo de uma classe usando um construtor privado.

C#

```
public class Counter
{
    private Counter() { }

    public static int currentCount;

    public static int IncrementCount()
    {
```

```
        return ++currentCount;
    }
}

class TestCounter
{
    static void Main()
    {
        // If you uncomment the following statement, it will generate
        // an error because the constructor is inaccessible:
        // Counter aCounter = new Counter();   // Error

        Counter.currentCount = 100;
        Counter.IncrementCount();
        Console.WriteLine($"New count: {Counter.currentCount}");

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
// Output: New count: 101
```

Observe que se você remover a marca de comentário da seguinte instrução do exemplo, ela gerará um erro porque o construtor está inacessível devido a seu nível de proteção:

C#

```
// Counter aCounter = new Counter();   // Error
```

Consulte também

- Sistema de tipos do C#
- Construtores
- Finalizadores
- privado
- público

Construtores estáticos (Guia de Programação em C#)

Artigo • 03/08/2024

Um construtor estático é usado para inicializar quaisquer dados estáticos ou para executar uma ação específica que precisa ser executada apenas uma vez. Ele é chamado automaticamente antes que a primeira instância seja criada ou os membros estáticos sejam referenciados. Um construtor estático é chamado no máximo uma vez.

C#

```
class SimpleClass
{
    // Static variable that must be initialized at run time.
    static readonly long baseline;

    // Static constructor is called at most one time, before any
    // instance constructor is invoked or member is accessed.
    static SimpleClass()
    {
        baseline = DateTime.Now.Ticks;
    }
}
```

Há várias ações que fazem parte da inicialização estática. Essas ações ocorrem na seguinte ordem:

1. *Os campos estáticos são definidos como 0*. O runtime normalmente faz essa inicialização.
2. *Inicializadores de campo estático são executados*. Os inicializadores de campo estático na execução de tipo mais derivada.
3. *Inicializadores de campo estático do tipo base são executados*. Inicializadores de campo estático começando com a base direta por meio de cada tipo base para System.Object.
4. *Qualquer construtor estático é executado*. Todos os construtores estáticos, desde a classe base final de Object.Object até cada classe base até a execução de tipo. A ordem de execução do construtor estático não é especificada. No entanto, todos os construtores estáticos na hierarquia são executados antes de qualquer instância ser criada.

ⓘ **Importante**

Há uma exceção importante à regra em que um construtor estático é executado antes de qualquer instância ser criada. Se um inicializador de campo estático criar uma instância do tipo, esse inicializador será executado (incluindo qualquer chamada para um construtor de instância) antes da execução do construtor estático. Isso é mais comum no *padrão singleton*, conforme mostrado no exemplo a seguir:

C#

```
public class Singleton
{
    // Static field initializer calls instance constructor.
    private static Singleton instance = new Singleton();

    private Singleton()
    {
        Console.WriteLine("Executes before static constructor.");
    }

    static Singleton()
    {
        Console.WriteLine("Executes after instance constructor.");
    }

    public static Singleton Instance => instance;
}
```

Um inicializador de módulo pode ser uma alternativa a um construtor estático. Para obter mais informações, consulte a especificação para inicializadores de módulo.

Comentários

Construtores estáticos têm as seguintes propriedades:

- Um construtor estático não usa modificadores de acesso nem tem parâmetros.
- Uma classe ou struct só pode ter um construtor estático.
- Construtores estáticos não podem ser herdados ou sobrecarregados.
- Um construtor estático não pode ser chamado diretamente e deve ser chamado apenas pelo CLR (Common Language Runtime). Ele é invocado automaticamente.
- O usuário não tem controle sobre quando o construtor estático é executado no programa.
- Um construtor estático é chamado automaticamente. Ele inicializa a classe antes de a primeira instância ser criada ou de os membros estáticos declarados nessa classe (não nas classes base) serem referenciados. Um construtor estático será

executado antes de um construtor de instância. Se os inicializadores de variável de campo estático estiverem presentes na classe do construtor estático, eles serão executados na ordem textual na qual aparecem na declaração de classe. Os inicializadores são executados imediatamente antes do construtor estático.

- Se você não fornecer um construtor estático para inicializar campos estáticos, todos os campos estáticos serão inicializados com seu valor padrão, conforme listado nos Valores padrão de tipos C#.
- Se um construtor estático gerar uma exceção, o runtime não a invocará uma segunda vez e o tipo permanecerá não inicializado durante o tempo de vida do domínio do aplicativo. Normalmente, uma exceção TypeInitializationException é lançada quando um construtor estático não consegue instanciar um tipo ou uma exceção sem tratamento que ocorre em um construtor estático. Para construtores estáticos que não estão explicitamente definidos no código-fonte, a solução de problemas pode exigir a inspeção do código IL (linguagem intermediária).
- A presença de um construtor estático impede a adição do atributo do tipo BeforeFieldInit. Isso limita a otimização do runtime.
- Um campo declarado como `static readonly` só pode ser atribuído como parte de sua declaração ou em um construtor estático. Quando um construtor estático explícito não for necessário, inicialize os campos estáticos na declaração, em vez de usar um construtor estático para melhorar a otimização do runtime.
- O runtime chama um construtor estático não mais do que uma vez em um único domínio do aplicativo. Essa chamada é feita em uma região bloqueada com base no tipo específico da classe. Nenhum mecanismo de bloqueio extra é necessário no corpo de um construtor estático. Para evitar o risco de deadlocks, não bloqueie o thread atual em inicializadores e construtores estáticos. Por exemplo, não aguarde tarefas, threads, identificadores de espera ou eventos, não adquira bloqueios e não execute o bloqueio de operações paralelas, como loops paralelos `Parallel.Invoke` e consultas LINQ paralelas.

> **Observação**
>
> Embora não seja diretamente acessível, a presença de um construtor estático explícito deve ser documentada para auxiliar na solução de problemas de exceções de inicialização.

Uso

- Um uso típico de construtores estáticos é quando a classe está usando um arquivo de log e o construtor é usado para gravar entradas nesse arquivo.

- Construtores estáticos também são úteis ao criar classes wrapper para código não gerenciado quando o construtor pode chamar o método `LoadLibrary`.
- Construtores estáticos também são um local conveniente para impor verificações em tempo de execução no parâmetro de tipo que não podem ser verificadas no tempo de compilação por meio de restrições de parâmetro de tipo.

Exemplo

Nesse exemplo, a classe `Bus` tem um construtor estático. Quando a primeira instância do `Bus` for criada (`bus1`), o construtor estático será invocado para inicializar a classe. O exemplo de saída verifica se o construtor estático é executado somente uma vez, mesmo se duas instâncias de `Bus` forem criadas e se é executado antes que o construtor da instância seja executado.

C#

```
public class Bus
{
    // Static variable used by all Bus instances.
    // Represents the time the first bus of the day starts its route.
    protected static readonly DateTime globalStartTime;

    // Property for the number of each bus.
    protected int RouteNumber { get; set; }

    // Static constructor to initialize the static variable.
    // It is invoked before the first instance constructor is run.
    static Bus()
    {
        globalStartTime = DateTime.Now;

        // The following statement produces the first line of output,
        // and the line occurs only once.
        Console.WriteLine("Static constructor sets global start time to
{0}",
            globalStartTime.ToLongTimeString());
    }

    // Instance constructor.
    public Bus(int routeNum)
    {
        RouteNumber = routeNum;
        Console.WriteLine("Bus #{0} is created.", RouteNumber);
    }

    // Instance method.
    public void Drive()
    {
        TimeSpan elapsedTime = DateTime.Now - globalStartTime;
```

```
        // For demonstration purposes we treat milliseconds as minutes to
simulate
        // actual bus times. Do not do this in your actual bus schedule
program!
        Console.WriteLine("{0} is starting its route {1:N2} minutes after
global start time {2}.",
                                this.RouteNumber,
                                elapsedTime.Milliseconds,
                                globalStartTime.ToShortTimeString());
    }
}

class TestBus
{
    static void Main()
    {
        // The creation of this instance activates the static constructor.
        Bus bus1 = new Bus(71);

        // Create a second bus.
        Bus bus2 = new Bus(72);

        // Send bus1 on its way.
        bus1.Drive();

        // Wait for bus2 to warm up.
        System.Threading.Thread.Sleep(25);

        // Send bus2 on its way.
        bus2.Drive();

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
/* Sample output:
    Static constructor sets global start time to 3:57:08 PM.
    Bus #71 is created.
    Bus #72 is created.
    71 is starting its route 6.00 minutes after global start time 3:57 PM.
    72 is starting its route 31.00 minutes after global start time 3:57 PM.
*/
```

Especificação da linguagem C#

Para saber mais, confira a seção Construtores estáticos da Especificação da linguagem C#.

Confira também

- Sistema de tipos do C#
- Construtores
- Classes static e membros de classes static
- Finalizadores
- Diretrizes de design de construtor
- Aviso de segurança – CA2121: os construtores estáticos devem ser privados
- Inicializadores de módulo

Colaborar conosco no GitHub

A fonte deste conteúdo pode ser encontrada no GitHub, onde você também pode criar e revisar problemas e solicitações de pull. Para obter mais informações, confira o nosso guia para colaboradores.

.NET

Comentários do .NET

O .NET é um projeto código aberto. Selecione um link para fornecer comentários:

Abrir um problema de documentação

Fornecer comentários sobre o produto

Como escrever um construtor de cópia (Guia de Programação em C#)

Artigo • 04/07/2024

Os registros em C# fornecem um construtor de cópia para objetos, mas para classes você precisa escrever um por conta própria.

> ⓘ **Importante**
>
> Escrever construtores de cópia que funcionam para todos os tipos derivados em uma hierarquia de classe pode ser difícil. Se a sua classe não for `sealed`, você deverá criar uma hierarquia de tipos `record class` para usar o construtor de cópia sintetizado pelo compilador.

Exemplo

No exemplo a seguir, a classe `Person` define um construtor de cópia que usa, como seu argumento, uma instância de `Person`. Os valores das propriedades do argumento são atribuídos às propriedades da nova instância de `Person`. O código contém um construtor de cópia alternativa que envia as propriedades `Name` e `Age` da instância que você deseja copiar para o construtor de instância da classe. A classe `Person` é `sealed`, portanto, nenhum tipo derivado pode ser declarado que possa introduzir erros copiando apenas a classe base.

C#

```
public sealed class Person
{
    // Copy constructor.
    public Person(Person previousPerson)
    {
        Name = previousPerson.Name;
        Age = previousPerson.Age;
    }

    //// Alternate copy constructor calls the instance constructor.
    //public Person(Person previousPerson)
    //    : this(previousPerson.Name, previousPerson.Age)
    //{
    //}

    // Instance constructor.
```

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

    public int Age { get; set; }

    public string Name { get; set; }

    public string Details()
    {
        return Name + " is " + Age.ToString();
    }
}

class TestPerson
{
    static void Main()
    {
        // Create a Person object by using the instance constructor.
        Person person1 = new Person("George", 40);

        // Create another Person object, copying person1.
        Person person2 = new Person(person1);

        // Change each person's age.
        person1.Age = 39;
        person2.Age = 41;

        // Change person2's name.
        person2.Name = "Charles";

        // Show details to verify that the name and age fields are distinct.
        Console.WriteLine(person1.Details());
        Console.WriteLine(person2.Details());

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
// Output:
// George is 39
// Charles is 41
```

Confira também

- ICloneable
- Registros
- O sistema de tipos C#

- Construtores
- Finalizadores

Colaborar conosco no GitHub

A fonte deste conteúdo pode ser encontrada no GitHub, onde você também pode criar e revisar problemas e solicitações de pull. Para obter mais informações, confira o nosso guia para colaboradores.

Comentários do .NET

O .NET é um projeto código aberto. Selecione um link para fornecer comentários:

Abrir um problema de documentação

Fornecer comentários sobre o produto

Finalizadores (Guia de Programação em C#)

Artigo • 14/03/2023

Os finalizadores (historicamente chamados de **destruidores**) são usados para executar a limpeza final necessária quando uma instância da classe está sendo coletada pelo coletor de lixo. Na maioria dos casos, você pode evitar escrever um finalizador usando System.Runtime.InteropServices.SafeHandle ou as classes derivadas para encapsular qualquer identificador não gerenciado.

Comentários

- Os finalizadores não podem ser definidos em structs. Eles são usados somente com classes.
- Uma classe pode ter somente um finalizador.
- Os finalizadores não podem ser herdados ou sobrecarregados.
- Os finalizadores não podem ser chamados. Eles são invocados automaticamente.
- Um finalizador não usa modificadores ou não tem parâmetros.

Por exemplo, o seguinte é uma declaração de um finalizador para a classe `Car`.

C#

```
class Car
{
    ~Car()  // finalizer
    {
        // cleanup statements...
    }
}
```

Um finalizador também pode ser implementado como uma definição do corpo da expressão, como mostra o exemplo a seguir.

C#

```
public class Destroyer
{
   public override string ToString() => GetType().Name;

   ~Destroyer() => Console.WriteLine($"The {ToString()} finalizer is
executing.");
}
```

O finalizador chama implicitamente Finalize na classe base do objeto. Portanto, uma chamada para um finalizador é convertida implicitamente para o código a seguir:

C#

```
protected override void Finalize()
{
    try
    {
        // Cleanup statements...
    }
    finally
    {
        base.Finalize();
    }
}
```

Esse arranjo significa que o método `Finalize` é chamado de forma recursiva para todas as instâncias da cadeia de herança, da mais derivada à menos derivada.

> ⓘ **Observação**
>
> Finalizadores vazios não devem ser usados. Quando uma classe contém um finalizador, uma entrada é criada na fila `Finalize`. Essa fila é processada pelo coletor de lixo. Quando o coletor de lixo processa a fila, ele chama cada finalizador. Finalizadores desnecessários, incluindo finalizadores vazios, finalizadores que chamam apenas o finalizador de classe base ou finalizadores que chamam apenas métodos emitidos condicionalmente, causam uma perda desnecessária de desempenho.

O programador não tem controle sobre quando o finalizador é chamado porque isso é determinado pelo coletor de lixo. O coletor de lixo procura objetos que não estão mais sendo usados pelo aplicativo. Se considerar um objeto qualificado para finalização, ele chamará o finalizador (se houver) e recuperará a memória usada para armazenar o objeto. É possível forçar a coleta de lixo chamando Collect, mas na maioria das vezes essa chamada deve ser evitada, porque pode criar problemas de desempenho.

> ⓘ **Observação**
>
> Executar ou não os finalizadores como parte do encerramento do aplicativo é uma situação específica para cada **implementação do .NET**. Quando um aplicativo é encerrado, o .NET Framework faz todos os esforços razoáveis para chamar

finalizadores para objetos que ainda não foram coletados, a menos que essa limpeza tenha sido suprimida (por uma chamada ao método de biblioteca `GC.SuppressFinalize`, por exemplo). O .NET 5 (incluindo o .NET Core) e versões posteriores não chamam finalizadores como parte do encerramento do aplicativo. Para obter mais informações, consulte o problema do GitHub **dotnet/csharpstandard #291**.

Se você precisar executar a limpeza de forma confiável quando um aplicativo for encerrado, registre um manipulador para o evento System.AppDomain.ProcessExit. Esse manipulador garantiria que IDisposable.Dispose() (ou) IAsyncDisposable.DisposeAsync() fosse chamado para todos os objetos que exigem limpeza antes da saída do aplicativo. Como você não pode chamar *Finalizar* diretamente e não pode garantir que o coletor de lixo chame todos os finalizadores antes de sair, você deve usar `Dispose` ou `DisposeAsync` garantir que os recursos sejam liberados.

Usar finalizadores para liberar recursos

Em geral, o C# não requer tanto gerenciamento de memória por parte do desenvolvedor quanto linguagens que não têm como destino um runtime com coleta de lixo. Isso ocorre porque o coletor de lixo do .NET gerencia implicitamente a alocação e a liberação de memória para seus objetos. No entanto, quando seu aplicativo encapsula recursos não gerenciados, como janelas, arquivos e conexões de rede, você deve usar finalizadores para liberar esses recursos. Quando o objeto está qualificado para finalização, o coletor de lixo executa o método `Finalize` do objeto.

Liberação explícita de recursos

Se seu aplicativo estiver usando um recurso externo caro, também será recomendável fornecer uma maneira de liberar explicitamente o recurso antes que o coletor de lixo libere o objeto. Para liberar o recurso, implemente um método `Dispose` da interface IDisposable que executa a limpeza necessária para o objeto. Isso pode melhorar consideravelmente o desempenho do aplicativo. Mesmo com esse controle explícito sobre os recursos, o finalizador se tornará uma proteção usada para limpar os recursos se a chamada para o método `Dispose` falhar.

Para obter mais informações sobre como limpar recursos, consulte os seguintes artigos:

- Limpando recursos não gerenciados
- Implementando um método Dispose
- Implementar um método DisposeAsync

- Instrução using

Exemplo

O exemplo a seguir cria três classes que compõem uma cadeia de herança. A classe `First` é a classe base, `Second` é derivado de `First` e `Third` é derivado de `Second`. Todas as três têm finalizadores. Em `Main`, uma instância da classe mais derivada é criada. A saída desse código depende de qual implementação do .NET o aplicativo tem como destino:

- .NET Framework: a saída mostra que os finalizadores das três classes são chamados automaticamente quando o aplicativo é encerrado, na ordem do mais derivado para o menos derivado.
- .NET 5 (incluindo o .NET Core) ou uma versão posterior: não há saída, porque essa implementação do .NET não chama finalizadores quando o aplicativo termina.

C#

```
class First
{
    ~First()
    {
        System.Diagnostics.Trace.WriteLine("First's finalizer is called.");
    }
}

class Second : First
{
    ~Second()
    {
        System.Diagnostics.Trace.WriteLine("Second's finalizer is called.");
    }
}

class Third : Second
{
    ~Third()
    {
        System.Diagnostics.Trace.WriteLine("Third's finalizer is called.");
    }
}

/*
Test with code like the following:
    Third t = new Third();
    t = null;

When objects are finalized, the output would be:
Third's finalizer is called.
```

```
Second's finalizer is called.
First's finalizer is called.
*/
```

Especificação da linguagem C#

Para obter mais informações, confira a seção Finalizadores da Especificação da linguagem C#.

Confira também

- IDisposable
- Guia de Programação em C#
- Construtores
- Coleta de lixo

Inicializadores de objeto e coleção (Guia de Programação em C#)

Artigo • 27/05/2024

O C# permite criar uma instância de um objeto ou uma coleção e executar as atribuições de membro em uma única instrução.

Inicializadores de objeto

Os inicializadores de objeto permitem atribuir valores a quaisquer campos ou propriedades acessíveis de um objeto na hora de criação sem que seja necessário invocar um construtor seguido por linhas de instruções de atribuição. A sintaxe do inicializador de objeto permite especificar argumentos para um construtor ou omitir os argumentos (e a sintaxe de parênteses). O exemplo a seguir mostra como usar um inicializador de objeto com um tipo nomeado, `Cat`, e como invocar o construtor sem parâmetros. Observe o uso de propriedades autoimplementadas na classe `Cat`. Para obter mais informações, consulte Propriedades autoimplementadas.

C#

```
public class Cat
{
    // Auto-implemented properties.
    public int Age { get; set; }
    public string? Name { get; set; }

    public Cat()
    {
    }

    public Cat(string name)
    {
        this.Name = name;
    }
}
```

C#

```
Cat cat = new Cat { Age = 10, Name = "Fluffy" };
Cat sameCat = new Cat("Fluffy"){ Age = 10 };
```

A sintaxe dos inicializadores de objetos permite que você crie uma instância, e depois atribui o objeto recém-criado, com suas propriedades atribuídas, à variável na

atribuição.

Os inicializadores de objeto podem definir indexadores, além de atribuir campos e propriedades. Considere esta classe `Matrix` básica:

C#

```
public class Matrix
{
    private double[,] storage = new double[3, 3];

    public double this[int row, int column]
    {
        // The embedded array will throw out of range exceptions as
appropriate.
        get { return storage[row, column]; }
        set { storage[row, column] = value; }
    }
}
```

Você poderia inicializar a matriz de identidade com o código a seguir:

C#

```
var identity = new Matrix
{
    [0, 0] = 1.0,
    [0, 1] = 0.0,
    [0, 2] = 0.0,

    [1, 0] = 0.0,
    [1, 1] = 1.0,
    [1, 2] = 0.0,

    [2, 0] = 0.0,
    [2, 1] = 0.0,
    [2, 2] = 1.0,
};
```

Nenhum indexador acessível que contenha um setter acessível pode ser usado como uma das expressões no inicializador de objeto, independentemente do número ou dos tipos de argumentos. Os argumentos de índice formam o lado esquerdo da atribuição e o valor é o lado direito da expressão. Por exemplo, todos os seguintes inicializadores são válidos se `IndexersExample` tiver os indexadores apropriados:

C#

```
var thing = new IndexersExample
{
```

```
    name = "object one",
    [1] = '1',
    [2] = '4',
    [3] = '9',
    Size = Math.PI,
    ['C',4] = "Middle C"
}
```

Para que o código anterior seja compilado, o tipo `IndexersExample` precisará ter os seguintes membros:

```C#
public string name;
public double Size { set { ... }; }
public char this[int i] { set { ... }; }
public string this[char c, int i] { set { ... }; }
```

Inicializadores de objeto com tipos anônimos

Embora inicializadores de objetos possam ser usados em qualquer contexto, eles são especialmente úteis em expressões de consulta LINQ. Expressões de consulta fazem uso frequente de tipos anônimos, que podem ser inicializados somente usando um inicializador de objeto, como mostrado na declaração a seguir.

```C#
var pet = new { Age = 10, Name = "Fluffy" };
```

Os tipos anônimos habilitam a cláusula `select` em uma expressão de consulta LINQ a transformar objetos da sequência original em objetos cujo valor e forma podem diferir dos originais. Talvez convenha armazenar apenas uma parte das informações de cada objeto em uma sequência. No exemplo a seguir, suponha que um objeto de produto (`p`) contenha muitos campos e métodos e que você só está interessado em criar uma sequência de objetos que contêm o nome do produto e o preço unitário.

```C#
var productInfos =
    from p in products
    select new { p.ProductName, p.UnitPrice };
```

Quando essa consulta é executada, a variável `productInfos` contém uma sequência de objetos que podem ser acessados em uma instrução `foreach` como mostrado neste exemplo:

```C#
foreach(var p in productInfos){...}
```

Cada objeto no novo tipo anônimo tem duas propriedades públicas que recebem os mesmos nomes que as propriedades ou os campos no objeto original. Você também poderá renomear um campo quando estiver criando um tipo anônimo; o exemplo a seguir renomeia o campo `UnitPrice` como `Price`.

```C#
select new {p.ProductName, Price = p.UnitPrice};
```

Inicializadores de objeto com o modificador `required`

Use a palavra-chave `required` para forçar os chamadores a definir o valor de uma propriedade ou campo usando um inicializador de objeto. As propriedades necessárias não precisam ser definidas como parâmetros de construtor. O compilador garante que todos os chamadores inicializem esses valores.

```C#
public class Pet
{
    public required int Age;
    public string Name;
}

// `Age` field is necessary to be initialized.
// You don't need to initialize `Name` property
var pet = new Pet() { Age = 10};

// Compiler error:
// Error CS9035 Required member 'Pet.Age' must be set in the object
initializer or attribute constructor.
// var pet = new Pet();
```

É uma prática típica garantir que seu objeto seja inicializado corretamente, especialmente quando você tem vários campos ou propriedades para gerenciar e não

deseja incluir todos no construtor.

Inicializadores de objeto com o acessador `init`

Garantir que ninguém altere o objeto projetado pode ser limitado usando um acessador `init`. Ele ajuda a restringir a configuração do valor da propriedade.

C#

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

// The `LastName` property can be set only during initialization. It CAN'T
be modified afterwards.
// The `FirstName` property can be modified after initialization.
var pet = new Person() { FirstName = "Joe", LastName = "Doe"};

// You can assign the FirstName property to a different value.
pet.FirstName = "Jane";

// Compiler error:
// Error CS8852  Init - only property or indexer 'Person.LastName' can only
be assigned in an object initializer,
//               or on 'this' or 'base' in an instance constructor or an
'init' accessor.
// pet.LastName = "Kowalski";
```

As propriedades somente inicialização necessárias dão suporte a estruturas imutáveis, permitindo a sintaxe natural para os usuários do tipo.

Inicializadores de objeto com propriedades tipadas por classe

É fundamental considerar as implicações para propriedades de tipo de classe ao inicializar um objeto:

C#

```
public class HowToClassTypedInitializer
{
    public class EmbeddedClassTypeA
    {
        public int I { get; set; }
        public bool B { get; set; }
```

```
        public string S { get; set; }
        public EmbeddedClassTypeB ClassB { get; set; }

        public override string ToString() => $"{I}|{B}|{S}|||{ClassB}";

        public EmbeddedClassTypeA()
        {
            Console.WriteLine($"Entering EmbeddedClassTypeA constructor.
Values are: {this}");
            I = 3;
            B = true;
            S = "abc";
            ClassB = new() { BB = true, BI = 43 };
            Console.WriteLine($"Exiting EmbeddedClassTypeA constructor.
Values are: {this})");
        }
    }

    public class EmbeddedClassTypeB
    {
        public int BI { get; set; }
        public bool BB { get; set; }
        public string BS { get; set; }

        public override string ToString() => $"{BI}|{BB}|{BS}";

        public EmbeddedClassTypeB()
        {
            Console.WriteLine($"Entering EmbeddedClassTypeB constructor.
Values are: {this}");
            BI = 23;
            BB = false;
            BS = "BBBabc";
            Console.WriteLine($"Exiting EmbeddedClassTypeB constructor.
Values are: {this})");
        }
    }

    public static void Main()
    {
        var a = new EmbeddedClassTypeA
        {
            I = 103,
            B = false,
            ClassB = { BI = 100003 }
        };
        Console.WriteLine($"After initializing EmbeddedClassTypeA: {a}");

        var a2 = new EmbeddedClassTypeA
        {
            I = 103,
            B = false,
            ClassB = new() { BI = 100003 } //New instance
        };
        Console.WriteLine($"After initializing EmbeddedClassTypeA a2:
```

```
{a2}");
    }

    // Output:
    //Entering EmbeddedClassTypeA constructor Values are: 0|False||||
    //Entering EmbeddedClassTypeB constructor Values are: 0|False|
    //Exiting EmbeddedClassTypeB constructor Values are: 23|False|BBBabc)
    //Exiting EmbeddedClassTypeA constructor Values are:
3|True|abc|||43|True|BBBabc)
    //After initializing EmbeddedClassTypeA:
103|False|abc|||100003|True|BBBabc
    //Entering EmbeddedClassTypeA constructor Values are: 0|False||||
    //Entering EmbeddedClassTypeB constructor Values are: 0|False|
    //Exiting EmbeddedClassTypeB constructor Values are: 23|False|BBBabc)
    //Exiting EmbeddedClassTypeA constructor Values are:
3|True|abc|||43|True|BBBabc)
    //Entering EmbeddedClassTypeB constructor Values are: 0|False|
    //Exiting EmbeddedClassTypeB constructor Values are: 23|False|BBBabc)
    //After initializing EmbeddedClassTypeA a2:
103|False|abc|||100003|False|BBBabc
}
```

O exemplo a seguir mostra como, para o ClassB, o processo de inicialização envolve a atualização de valores específicos, mantendo outros da instância original. O inicializador reutiliza a instância atual. Os valores de ClassB são: `100003` (novo valor que atribuímos aqui), `true` (mantido da inicialização de EmbeddedClassTypeA), `BBBabc` (padrão inalterado de EmbeddedClassTypeB).

Inicializadores de coleção

Os inicializadores de coleção permitem especificar um ou mais inicializadores de elemento quando você inicializa um tipo de coleção que implementa IEnumerable e tem `Add` com a assinatura apropriada como um método de instância ou um método de extensão. Os inicializadores de elemento podem ser um valor, uma expressão ou um inicializador de objeto. Ao usar um inicializador de coleção, você não precisa especificar várias chamadas. O compilador adiciona as chamadas automaticamente.

O exemplo a seguir mostra dois inicializadores de coleção simples:

C#

```
List<int> digits = new List<int> { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
List<int> digits2 = new List<int> { 0 + 1, 12 % 3, MakeInt() };
```

O inicializador de coleção a seguir usa inicializadores de objeto para inicializar objetos da classe `Cat` definida em um exemplo anterior. Os inicializadores de objeto individuais

são envolvidos por chaves e separados por vírgulas.

C#

```
List<Cat> cats = new List<Cat>
{
    new Cat{ Name = "Sylvester", Age=8 },
    new Cat{ Name = "Whiskers", Age=2 },
    new Cat{ Name = "Sasha", Age=14 }
};
```

Você poderá especificar nulo como um elemento em um inicializador de coleção se o método `Add` da coleção permitir.

C#

```
List<Cat?> moreCats = new List<Cat?>
{
    new Cat{ Name = "Furrytail", Age=5 },
    new Cat{ Name = "Peaches", Age=4 },
    null
};
```

É possível especificar elementos indexados quando a coleção é compatível com indexação de leitura/gravação.

C#

```
var numbers = new Dictionary<int, string>
{
    [7] = "seven",
    [9] = "nine",
    [13] = "thirteen"
};
```

O exemplo anterior gera o código que chama o Item[TKey] para definir os valores. Você também poderia inicializar dicionários e outros contêineres associativos usando a seguinte sintaxe. Observe que, em vez da sintaxe do indexador, com parênteses e uma atribuição, ele usa um objeto com vários valores:

C#

```
var moreNumbers = new Dictionary<int, string>
{
    {19, "nineteen" },
    {23, "twenty-three" },
```

```
    {42, "forty-two" }
};
```

Este exemplo de inicializador chama Add(TKey, TValue) para adicionar os três itens no dicionário. Essas duas maneiras diferentes para inicializar coleções associativas tem um comportamento um pouco diferente devido às chamadas de método que o compilador gera. As duas variantes trabalham com a classe `Dictionary`. Outros tipos podem ser compatíveis apenas com uma ou com outra, dependendo da API pública deles.

Inicializadores de objeto com inicialização de propriedade somente leitura da coleção

Algumas classes podem ter propriedades de coleção em que a propriedade é somente leitura, como a propriedade `Cats` de `CatOwner` no seguinte caso:

C#

```
public class CatOwner
{
    public IList<Cat> Cats { get; } = new List<Cat>();
}
```

Você não pode usar a sintaxe do inicializador de coleção discutida até o momento, pois a propriedade não pode ser atribuída a uma nova lista:

C#

```
CatOwner owner = new CatOwner
{
    Cats = new List<Cat>
    {
        new Cat{ Name = "Sylvester", Age=8 },
        new Cat{ Name = "Whiskers", Age=2 },
        new Cat{ Name = "Sasha", Age=14 }
    }
};
```

No entanto, novas entradas podem ser adicionadas a `Cats` usando a sintaxe de inicialização omitindo a criação da lista (`new List<Cat>`), conforme mostrado a seguir:

C#

```
CatOwner owner = new CatOwner
{
```

```
    Cats =
    {
        new Cat{ Name = "Sylvester", Age=8 },
        new Cat{ Name = "Whiskers", Age=2 },
        new Cat{ Name = "Sasha", Age=14 }
    }
};
```

O conjunto de entradas a serem adicionadas é exibido entre parênteses. O código anterior é idêntico à gravação:

C#

```
CatOwner owner = new ();
owner.Cats.Add(new Cat{ Name = "Sylvester", Age=8 });
owner.Cats.Add(new Cat{ Name = "Whiskers", Age=2 });
owner.Cats.Add(new Cat{ Name = "Sasha", Age=14 });
```

Exemplos

O exemplo a seguir combina os conceitos de inicializadores de coleção e objeto.

C#

```
public class InitializationSample
{
    public class Cat
    {
        // Auto-implemented properties.
        public int Age { get; set; }
        public string? Name { get; set; }

        public Cat() { }

        public Cat(string name)
        {
            Name = name;
        }
    }

    public static void Main()
    {
        Cat cat = new Cat { Age = 10, Name = "Fluffy" };
        Cat sameCat = new Cat("Fluffy"){ Age = 10 };

        List<Cat> cats = new List<Cat>
        {
            new Cat { Name = "Sylvester", Age = 8 },
            new Cat { Name = "Whiskers", Age = 2 },
            new Cat { Name = "Sasha", Age = 14 }
```

```
        };

        List<Cat?> moreCats = new List<Cat?>
        {
            new Cat { Name = "Furrytail", Age = 5 },
            new Cat { Name = "Peaches", Age = 4 },
            null
        };

        // Display results.
        System.Console.WriteLine(cat.Name);

        foreach (Cat c in cats)
        {
            System.Console.WriteLine(c.Name);
        }

        foreach (Cat? c in moreCats)
        {
            if (c != null)
            {
                System.Console.WriteLine(c.Name);
            }
            else
            {
                System.Console.WriteLine("List element has null value.");
            }
        }
    }
    // Output:
    //Fluffy
    //Sylvester
    //Whiskers
    //Sasha
    //Furrytail
    //Peaches
    //List element has null value.
}
```

O exemplo a seguir mostra um objeto que implementa IEnumerable e contém um método `Add` com vários parâmetros. Ele usa um inicializador de coleção com vários elementos por item na lista que correspondem à assinatura do método `Add`.

C#

```
public class FullExample
{
    class FormattedAddresses : IEnumerable<string>
    {
        private List<string> internalList = new List<string>();
        public IEnumerator<string> GetEnumerator() =>
internalList.GetEnumerator();
```

```
        System.Collections.IEnumerator
System.Collections.IEnumerable.GetEnumerator() =>
internalList.GetEnumerator();

        public void Add(string firstname, string lastname,
            string street, string city,
            string state, string zipcode) => internalList.Add($"""
            {firstname} {lastname}
            {street}
            {city}, {state} {zipcode}
            """
            );
    }

    public static void Main()
    {
        FormattedAddresses addresses = new FormattedAddresses()
        {
            {"John", "Doe", "123 Street", "Topeka", "KS", "00000" },
            {"Jane", "Smith", "456 Street", "Topeka", "KS", "00000" }
        };

        Console.WriteLine("Address Entries:");

        foreach (string addressEntry in addresses)
        {
            Console.WriteLine("\r\n" + addressEntry);
        }
    }

    /*
        * Prints:

        Address Entries:

        John Doe
        123 Street
        Topeka, KS 00000

        Jane Smith
        456 Street
        Topeka, KS 00000
        */
}
```

Os métodos `Add` podem usar a palavra-chave `params` para obter um número variável de argumentos, como mostrado no seguinte exemplo. Este exemplo também demonstra a implementação personalizada de um indexador para inicializar uma coleção usando índices. A partir do C# 13, o parâmetro `params` não é restrito a uma matriz. Pode ser um tipo de coleção ou interface.

C#

```
public class DictionaryExample
{
    class RudimentaryMultiValuedDictionary<TKey, TValue> :
IEnumerable<KeyValuePair<TKey, List<TValue>>> where TKey : notnull
    {
        private Dictionary<TKey, List<TValue>> internalDictionary = new
Dictionary<TKey, List<TValue>>();

        public IEnumerator<KeyValuePair<TKey, List<TValue>>> GetEnumerator()
=> internalDictionary.GetEnumerator();

        System.Collections.IEnumerator
System.Collections.IEnumerable.GetEnumerator() =>
internalDictionary.GetEnumerator();

        public List<TValue> this[TKey key]
        {
            get => internalDictionary[key];
            set => Add(key, value);
        }

        public void Add(TKey key, params TValue[] values) => Add(key,
(IEnumerable<TValue>)values);

        public void Add(TKey key, IEnumerable<TValue> values)
        {
            if (!internalDictionary.TryGetValue(key, out List<TValue>?
storedValues))
            {
                internalDictionary.Add(key, storedValues = new List<TValue>
());
            }
            storedValues.AddRange(values);
        }
    }

    public static void Main()
    {
        RudimentaryMultiValuedDictionary<string, string>
rudimentaryMultiValuedDictionary1
            = new RudimentaryMultiValuedDictionary<string, string>()
            {
                {"Group1", "Bob", "John", "Mary" },
                {"Group2", "Eric", "Emily", "Debbie", "Jesse" }
            };
        RudimentaryMultiValuedDictionary<string, string>
rudimentaryMultiValuedDictionary2
            = new RudimentaryMultiValuedDictionary<string, string>()
            {
                ["Group1"] = new List<string>() { "Bob", "John", "Mary" },
                ["Group2"] = new List<string>() { "Eric", "Emily", "Debbie",
"Jesse" }
```

```
            };
        RudimentaryMultiValuedDictionary<string, string>
rudimentaryMultiValuedDictionary3
            = new RudimentaryMultiValuedDictionary<string, string>()
            {
                {"Group1", new string []{ "Bob", "John", "Mary" } },
                { "Group2", new string[]{ "Eric", "Emily", "Debbie", "Jesse"
} }
            };

        Console.WriteLine("Using first multi-valued dictionary created with
a collection initializer:");

        foreach (KeyValuePair<string, List<string>> group in
rudimentaryMultiValuedDictionary1)
        {
            Console.WriteLine($"\r\nMembers of group {group.Key}: ");

            foreach (string member in group.Value)
            {
                Console.WriteLine(member);
            }
        }

        Console.WriteLine("\r\nUsing second multi-valued dictionary created
with a collection initializer using indexing:");

        foreach (KeyValuePair<string, List<string>> group in
rudimentaryMultiValuedDictionary2)
        {
            Console.WriteLine($"\r\nMembers of group {group.Key}: ");

            foreach (string member in group.Value)
            {
                Console.WriteLine(member);
            }
        }
        Console.WriteLine("\r\nUsing third multi-valued dictionary created
with a collection initializer using indexing:");

        foreach (KeyValuePair<string, List<string>> group in
rudimentaryMultiValuedDictionary3)
        {
            Console.WriteLine($"\r\nMembers of group {group.Key}: ");

            foreach (string member in group.Value)
            {
                Console.WriteLine(member);
            }
        }
    }

    /*
        * Prints:
```

```
        Using first multi-valued dictionary created with a collection
initializer:

        Members of group Group1:
        Bob
        John
        Mary

        Members of group Group2:
        Eric
        Emily
        Debbie
        Jesse

        Using second multi-valued dictionary created with a collection
initializer using indexing:

        Members of group Group1:
        Bob
        John
        Mary

        Members of group Group2:
        Eric
        Emily
        Debbie
        Jesse

        Using third multi-valued dictionary created with a collection
initializer using indexing:

        Members of group Group1:
        Bob
        John
        Mary

        Members of group Group2:
        Eric
        Emily
        Debbie
        Jesse
        */
}
```

Confira também

- Usar inicializadores de objeto (regra de estilo IDE0017)
- Usar inicializadores de coleção (regra de estilo IDE0028)
- LINQ em C#
- Tipos anônimos

Colaborar conosco no GitHub

A fonte deste conteúdo pode ser encontrada no GitHub, onde você também pode criar e revisar problemas e solicitações de pull. Para obter mais informações, confira o nosso guia para colaboradores.

Comentários do .NET

O .NET é um projeto código aberto. Selecione um link para fornecer comentários:

- Abrir um problema de documentação
- Fornecer comentários sobre o produto

Como inicializar objetos usando um inicializador de objeto (Guia de Programação em C#)

Artigo • 15/05/2024

Você pode usar os inicializadores de objeto para inicializar objetos de tipo de maneira declarativa, sem invocar explicitamente um construtor para o tipo.

Os exemplos a seguir mostram como usar os inicializadores de objeto com objetos nomeados. O compilador processa os inicializadores de objetos ao acessar, primeiro, o construtor de instância sem parâmetro e, em seguida, processar as inicializações de membro. Portanto, se o construtor sem parâmetros for declarado como `private` na classe, os inicializadores de objeto que exigem acesso público falharão.

Se você estiver definindo um tipo anônimo, é necessário usar um inicializador de objeto. Para obter mais informações, consulte Como retornar subconjuntos de propriedades de elementos em uma consulta.

Exemplo

O exemplo a seguir mostra como inicializar um novo tipo `StudentName`, usando inicializadores de objeto. Este exemplo define as propriedades de `StudentName` tipo:

C#

```
public class HowToObjectInitializers
{
    public static void Main()
    {
        // Declare a StudentName by using the constructor that has two
parameters.
        StudentName student1 = new StudentName("Craig", "Playstead");

        // Make the same declaration by using an object initializer and
sending
        // arguments for the first and last names. The parameterless
constructor is
        // invoked in processing this declaration, not the constructor that
has
        // two parameters.
        StudentName student2 = new StudentName
        {
            FirstName = "Craig",
            LastName = "Playstead"
```

```
        };

        // Declare a StudentName by using an object initializer and sending
        // an argument for only the ID property. No corresponding
constructor is
        // necessary. Only the parameterless constructor is used to process
object
        // initializers.
        StudentName student3 = new StudentName
        {
            ID = 183
        };

        // Declare a StudentName by using an object initializer and sending
        // arguments for all three properties. No corresponding constructor
is
        // defined in the class.
        StudentName student4 = new StudentName
        {
            FirstName = "Craig",
            LastName = "Playstead",
            ID = 116
        };

        Console.WriteLine(student1.ToString());
        Console.WriteLine(student2.ToString());
        Console.WriteLine(student3.ToString());
        Console.WriteLine(student4.ToString());
    }
    // Output:
    // Craig  0
    // Craig  0
    //   183
    // Craig  116

    public class StudentName
    {
        // This constructor has no parameters. The parameterless constructor
        // is invoked in the processing of object initializers.
        // You can test this by changing the access modifier from public to
        // private. The declarations in Main that use object initializers
will
        // fail.
        public StudentName() { }

        // The following constructor has parameters for two of the three
        // properties.
        public StudentName(string first, string last)
        {
            FirstName = first;
            LastName = last;
        }

        // Properties.
        public string? FirstName { get; set; }
```

```
        public string? LastName { get; set; }
        public int ID { get; set; }

        public override string ToString() => FirstName + "  " + ID;
    }
}
```

Os inicializadores de objeto podem ser usados para definir indexadores em um objeto. O exemplo a seguir define uma classe `BaseballTeam` que usa um indexador para obter e definir jogadores em posições diferentes. O inicializador pode atribuir jogadores com base na abreviação da posição ou no número usado para cada scorecard de beisebol de posição:

C#

```
public class HowToIndexInitializer
{
    public class BaseballTeam
    {
        private string[] players = new string[9];
        private readonly List<string> positionAbbreviations = new
List<string>
        {
            "P", "C", "1B", "2B", "3B", "SS", "LF", "CF", "RF"
        };

        public string this[int position]
        {
            // Baseball positions are 1 - 9.
            get { return players[position-1]; }
            set { players[position-1] = value; }
        }
        public string this[string position]
        {
            get { return players[positionAbbreviations.IndexOf(position)]; }
            set { players[positionAbbreviations.IndexOf(position)] = value;
}
        }
    }

    public static void Main()
    {
        var team = new BaseballTeam
        {
            ["RF"] = "Mookie Betts",
            [4] = "Jose Altuve",
            ["CF"] = "Mike Trout"
        };

        Console.WriteLine(team["2B"]);
```

```
    }
}
```

O próximo exemplo mostra a ordem de execução das inicializações de construtores e membros usando construtor com e sem parâmetro:

```
C#
```

Confira também

- Inicializadores de objeto e coleção

Colaborar conosco no GitHub

A fonte deste conteúdo pode ser encontrada no GitHub, onde você também pode criar e revisar problemas e solicitações de pull. Para obter mais informações, confira o nosso guia para colaboradores.

.NET

Comentários do .NET

O .NET é um projeto código aberto. Selecione um link para fornecer comentários:

Abrir um problema de documentação

Fornecer comentários sobre o produto

Como inicializar um dicionário com um inicializador de coleção (Guia de Programação em C#)

Artigo • 31/01/2025

Um Dictionary<TKey,TValue> contém uma coleção de pares de chave-valor. Seu método Add recebe dois parâmetros, um para a chave e outro para o valor. Uma maneira de inicializar um Dictionary<TKey,TValue> ou qualquer coleção cujo método `Add` use vários parâmetros, é colocar cada conjunto de parâmetros entre chaves, conforme mostrado no exemplo a seguir. Outra opção é usar um inicializador de índice, também mostrado no exemplo a seguir.

> **Observação**
>
> A principal diferença entre essas duas maneiras de inicializar a coleção é que, no caso de ter chaves duplicadas, por exemplo:
>
> ```C#
> { 111, new StudentName { FirstName="Sachin", LastName="Karnik", ID=211
> } },
> { 111, new StudentName { FirstName="Dina", LastName="Salimzianova",
> ID=317 } },
> ```
>
> O método Add gerará ArgumentException: `'An item with the same key has already been added. Key: 111'`, enquanto a segunda parte do exemplo, o método de indexador público de leitura/gravação, substituirá silenciosamente a entrada já existente com a mesma chave.

> **Dica**
>
> Você pode usar a assistência de IA para inicializar um dicionário usando o GitHub Copilot.

Exemplo

No exemplo de código a seguir, um Dictionary<TKey,TValue> é inicializado com instâncias do tipo `StudentName`. A primeira inicialização usa o método `Add` com dois argumentos. O compilador gera uma chamada para `Add` para cada um dos pares de chaves `int` e valores `StudentName`. A segunda usa um método de indexador público de leitura/gravação da classe `Dictionary`:

C#

```
public class HowToDictionaryInitializer
{
    class StudentName
    {
        public string? FirstName { get; set; }
        public string? LastName { get; set; }
        public int ID { get; set; }
    }

    public static void Main()
    {
        var students = new Dictionary<int, StudentName>()
        {
            { 111, new StudentName { FirstName="Sachin", LastName="Karnik",
ID=211 } },
            { 112, new StudentName { FirstName="Dina",
LastName="Salimzianova", ID=317 } },
            { 113, new StudentName { FirstName="Andy", LastName="Ruth",
ID=198 } }
        };

        foreach(var index in Enumerable.Range(111, 3))
        {
            Console.WriteLine($"Student {index} is
{students[index].FirstName} {students[index].LastName}");
        }
        Console.WriteLine();

        var students2 = new Dictionary<int, StudentName>()
        {
            [111] = new StudentName { FirstName="Sachin", LastName="Karnik",
ID=211 },
            [112] = new StudentName { FirstName="Dina",
LastName="Salimzianova", ID=317 } ,
            [113] = new StudentName { FirstName="Andy", LastName="Ruth",
ID=198 }
        };

        foreach (var index in Enumerable.Range(111, 3))
        {
            Console.WriteLine($"Student {index} is
{students2[index].FirstName} {students2[index].LastName}");
        }
```

```
    }
}
```

Observe os dois pares de chaves em cada elemento da coleção na primeira declaração. As chaves mais internas incluem o inicializador do objeto para `StudentName` e as chaves mais externas incluem o inicializador do par chave/valor a ser adicionado ao `students` Dictionary<TKey,TValue>. Por fim, todo o inicializador de coleção do dicionário é colocado entre chaves. Na segunda inicialização, o lado esquerdo da atribuição é a chave e o lado direito é o valor, usando um inicializador de objeto para `StudentName`.

Usar o GitHub Copilot para inicializar um dicionário

Você pode usar o GitHub Copilot em seu IDE para gerar código C# para inicializar um dicionário com um inicializador de coleção. Você pode personalizar o prompt para adicionar detalhes de acordo com seus requisitos.

O texto a seguir mostra um exemplo de prompt para o Copilot Chat:

Prompt do Copilot

```
Generate C# code to initialize Dictionary<int, Employee> using key-value
pairs within the collection initializer. The employee class is a record
class with two properties: Name and Age.
```

O GitHub Copilot é alimentado pela IA, portanto, surpresas e erros são possíveis. Para obter mais informações, consulte Perguntas frequentes sobre o Copilot.

Saiba mais sobre GitHub Copilot no Visual Studio e GitHub Copilot no VS Code.

Confira também

- Inicializadores de objeto e coleção

Tipos aninhados (Guia de Programação em C#)

19/06/2025

Um tipo definido em uma classe, struct ou interface é chamado de tipo aninhado. Por exemplo

C#

```
public class Container
{
    class Nested
    {
        Nested() { }
    }
}
```

Independentemente de o tipo externo ser uma classe, uma interface ou uma struct, os tipos aninhados são privados por padrão, acessíveis somente por meio do tipo que contêm. No exemplo anterior, a classe `Nested` é inacessível para tipos externos.

Você também pode especificar um modificador de acesso para definir a acessibilidade de um tipo aninhado, da seguinte maneira:

- Os tipos aninhados de uma **classe** podem ser públicos, protegidos, internos, protegidos internos, privados ou privados protegidos.

  No entanto, definir uma classe aninhada `protected`, `protected internal`, ou `private protected` dentro de uma classe selada gera o aviso do compilador CS0628, "novo membro protegido declarado na classe selada".

  Lembre-se também de que tornar um tipo aninhado visível externamente viola a regra de qualidade de código CA1034 "Tipos aninhados não devem estar visíveis".

- Tipos aninhados de um **struct** podem ser públicos, internos ou particulares.

O exemplo a seguir torna a classe `Nested` pública:

C#

```
public class Container
{
    public class Nested
    {
        Nested() { }
```

```
    }
}
```

O tipo aninhado ou interno pode acessar o tipo recipiente ou externo. Para acessar o tipo recipiente, passe-o como um argumento ao construtor do tipo aninhado. Por exemplo:

```C#
public class Container
{
    public class Nested
    {
        private Container? parent;

        public Nested()
        {
        }
        public Nested(Container parent)
        {
            this.parent = parent;
        }
    }
}
```

Um tipo aninhado tem acesso a todos os membros acessíveis ao seu tipo recipiente. Ele pode acessar membros privados e protegidos do tipo recipiente, incluindo quaisquer membros protegidos herdados.

Na declaração anterior, o nome completo da classe `Nested` é `Container.Nested`. Esse é o nome usado para criar uma nova instância da classe aninhada, da seguinte maneira:

```C#
Container.Nested nest = new Container.Nested();
```

Consulte também

- Sistema de tipos do C#
- Modificadores de acesso
- Construtores
- Regra CA1034

Classes e membros parciais (Guia de Programação em C#)

17/06/2025

É possível dividir a definição de uma classe, um struct, uma interface ou um membro em dois ou mais arquivos de origem. Cada arquivo de origem contém uma seção do tipo ou definição de membro e todas as partes são combinadas quando o aplicativo é compilado.

Classes parciais

Há várias situações em que a divisão de uma definição de classe é desejável:

- Declarar uma classe em arquivos separados permite que vários programadores trabalhem nela ao mesmo tempo.
- Você pode adicionar código à classe sem precisar recriar o arquivo de origem que inclui a origem gerada automaticamente. O Visual Studio usa essa abordagem quando cria Windows Forms, código de wrapper de serviço Web e assim por diante. Você pode criar código que usa essas classes sem precisar modificar o arquivo que o Visual Studio cria.
- Os geradores de origem podem gerar funcionalidade extra em uma classe.

Para dividir uma definição de classe, use o modificador de palavra-chave parcial . Na prática, cada classe parcial é normalmente definida em um arquivo separado, facilitando o gerenciamento e a expansão da classe ao longo do tempo.

O exemplo a seguir `Employee` demonstra como a classe pode ser dividida em dois arquivos: Employee_Part1.cs e Employee_Part2.cs.

C#

```
// This is in Employee_Part1.cs
public partial class Employee
{
    public void DoWork()
    {
        Console.WriteLine("Employee is working.");
    }
}

// This is in Employee_Part2.cs
public partial class Employee
{
    public void GoToLunch()
    {
        Console.WriteLine("Employee is at lunch.");
```

```
    }
}

//Main program demonstrating the Employee class usage
public class Program
{
    public static void Main()
    {
        Employee emp = new Employee();
        emp.DoWork();
        emp.GoToLunch();
    }
}

// Expected Output:
// Employee is working.
// Employee is at lunch.
```

A palavra-chave `partial` indica que outras partes da classe, struct ou interface podem ser definidas no namespace. Todas as partes devem usar a palavra-chave `partial`. Todas as partes devem estar disponíveis em tempo de compilação para formar o tipo final. Todas as partes devem ter a mesma acessibilidade, tais como `public`, `private` e assim por diante.

Se alguma parte for declarada como abstrata, o tipo inteiro será considerado abstrato. Se alguma parte for declarada como lacrada, o tipo inteiro será considerado lacrado. Se alguma parte declarar um tipo base, o tipo inteiro herda dessa classe.

Todas as partes que especificam uma classe base devem concordar, mas partes que omitem uma classe base ainda herdam o tipo base. As partes podem especificar diferentes interfaces base e o tipo final implementa todas as interfaces listadas por todas as declarações parciais. Qualquer membro de classe, struct ou interface declarado em uma definição parcial está disponível para todas as outras partes. O tipo final é a combinação de todas as partes em tempo de compilação.

> ⓘ **Observação**
>
> O modificador de `partial` não está disponível em declarações de representante ou enumeração.

O exemplo a seguir mostra que os tipos aninhados podem ser parciais, mesmo que o tipo em que estão aninhados não seja parcial.

C#

```
class Container
{
```

```
    partial class Nested
    {
        void Test() { }
    }

    partial class Nested
    {
        void Test2() { }
    }
}
```

Em tempo de compilação, atributos de definições de tipo parcial são mesclados. Por exemplo, considere as declarações a seguir:

C#

```
[SerializableAttribute]
partial class Moon { }

[ObsoleteAttribute]
partial class Moon { }
```

Elas são equivalentes às seguintes declarações:

C#

```
[SerializableAttribute]
[ObsoleteAttribute]
class Moon { }
```

Os itens a seguir são mesclados de todas as definições de tipo parcial:

- Comentários XML. No entanto, se ambas as declarações de um membro parcial incluírem comentários, apenas os comentários do membro de implementação serão incluídos.
- Interfaces
- atributos de parâmetro de tipo genérico
- atributos class
- membros

Por exemplo, considere as declarações a seguir:

C#

```
partial class Earth : Planet, IRotate { }
partial class Earth : IRevolve { }
```

Elas são equivalentes às seguintes declarações:

```C#
class Earth : Planet, IRotate, IRevolve { }
```

Restrições

Há várias regras a seguir quando você está trabalhando com definições parciais de classe:

- Todas as definições de tipo parcial que devem ser partes do mesmo tipo devem ser modificadas com `partial`. Por exemplo, as seguintes declarações de classe geram um erro:

  ```C#
  public partial class A { }
  //public class A { }  // Error, must also be marked partial
  ```

- O modificador `partial` só pode aparecer imediatamente antes da palavra-chave `class`, `struct` ou `interface`.
- Tipos parciais aninhados são permitidos em definições de tipo parcial, conforme ilustrado no exemplo a seguir:

  ```C#
  partial class ClassWithNestedClass
  {
      partial class NestedClass { }
  }

  partial class ClassWithNestedClass
  {
      partial class NestedClass { }
  }
  ```

- Todas as definições de tipo parcial que devem ser partes do mesmo tipo devem ser definidas no mesmo assembly e no mesmo módulo (arquivo .dll ou .exe). Definições parciais não podem abranger vários módulos.
- O nome de classe e os parâmetros de tipo genérico devem corresponder em todas as definições de tipo parcial. Tipos genéricos podem ser parciais. Cada declaração parcial deve usar os mesmos nomes de parâmetro na mesma ordem.
- As seguintes palavras-chave em uma definição de tipo parcial são opcionais, mas, se estiverem presentes em uma definição de tipo parcial, é preciso que esteja especificada em outra definição parcial para o mesmo tipo:

- público
- privado
- Protegido
- interno
- resumo
- selado
- classe base
- modificador new (partes aninhadas)
- restrições genéricas

Para obter mais informações, consulte Restrições a parâmetros de tipo.

Exemplos

No exemplo a seguir, os campos e o `Coords` construtor da classe são declarados em uma definição de classe parcial (`Coords_Part1.cs`) e o método é declarado `PrintCoords` em outra definição de classe parcial (`Coords_Part2.cs`). Essa separação demonstra como as classes parciais podem ser divididas em vários arquivos para facilitar a manutenção.

C#

```
// This is in Coords_Part1.cs
public partial class Coords
{
    private int x;
    private int y;

    public Coords(int x, int y)
    {
        this.x = x;
        this.y = y;
    }
}

// This is in Coords_Part2.cs
public partial class Coords
{
    public void PrintCoords()
    {
        Console.WriteLine("Coords: {0},{1}", x, y);
    }
}

// Main program demonstrating the Coords class usage
class TestCoords
{
    static void Main()
    {
```

```
        Coords myCoords = new Coords(10, 15);
        myCoords.PrintCoords();

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
// Output: Coords: 10,15
```

O exemplo a seguir mostra que você também pode desenvolver interfaces e structs parciais.

C#

```
partial interface ITest
{
    void Interface_Test();
}

partial interface ITest
{
    void Interface_Test2();
}

partial struct S1
{
    void Struct_Test() { }
}

partial struct S1
{
    void Struct_Test2() { }
}
```

Membros parciais

Uma classe ou struct parcial pode conter um membro parcial. Uma parte da classe contém a assinatura do membro. Uma implementação pode ser definida na mesma parte ou em outra parte.

Uma implementação não é necessária para um método parcial quando a assinatura obedece às seguintes regras:

- A declaração não inclui modificadores de acesso. O método tem acesso private por padrão.
- O tipo de retorno é void.
- Nenhum dos parâmetros tem o modificador out.
- A declaração do método não pode incluir nenhum dos seguintes modificadores:

- virtual
- substituir
- selado
- novo
- Extern

O método e todas as chamadas para o método são removidos no momento da compilação quando não há nenhuma implementação.

Qualquer membro que não esteja em conformidade com todas essas restrições, incluindo construtores, propriedades, indexadores e eventos, deve fornecer uma implementação. Essa implementação pode ser fornecida por um *gerador de origem*. Propriedades parciais não podem ser implementadas usando propriedades implementadas automaticamente. O compilador não pode distinguir entre uma propriedade implementada automaticamente e a declaração de declaração de uma propriedade parcial.

A partir do C# 13, a declaração de implementação de uma propriedade parcial pode usar propriedades com suporte de campo para definir a declaração de implementação. Uma propriedade com suporte de campo fornece uma sintaxe concisa em que a palavra-chave acessa `field` o campo de suporte sintetizado do compilador para a propriedade. Por exemplo, você pode escrever o seguinte código:

C#

```
// in file1.cs
public partial class PropertyBag
{
    // Defining declaration
    public partial int MyProperty { get; set; }
}

// In file2.cs
public partial class PropertyBag
{
    // Defining declaration
    public partial int MyProperty { get => field; set; }
}
```

Você pode usar `field` no `get` acessador ou `set` em ambos.

> **Importante**
>
> A `field` palavra-chave é um recurso de visualização no C# 13. Você deve estar usando o .NET 9 e definir seu `<LangVersion>` elemento como `preview` no arquivo de projeto para usar a `field` palavra-chave contextual.

Você deve ter cuidado ao usar o recurso de `field` palavra-chave em uma classe que tenha um campo chamado `field`. A nova `field` palavra-chave sombreia um campo nomeado `field` no escopo de um acessador de propriedade. Você pode alterar o nome da `field` variável ou usar o `@` token para fazer referência ao `field` identificador como `@field`. Você pode saber mais lendo a especificação do recurso para [a field palavra-chave](#).

Membros parciais permitem que o implementador de uma parte de uma classe declare um membro. O implementador de outra parte da classe pode definir esse membro. Há dois cenários em que essa separação é útil: modelos que geram código clichê e geradores de origem.

- **Código do modelo**: o modelo reserva um nome de método e uma assinatura para que o código gerado possa chamar o método. Esses métodos seguem as restrições que permitem a um desenvolvedor decidir se quer implementar o método. Se o método não for implementado, o compilador removerá a assinatura do método e todas as chamadas para o método. As chamadas para o método, incluindo qualquer resultado que ocorreria da avaliação de argumentos nas chamadas, não têm efeito em tempo de execução. Portanto, qualquer código na classe parcial pode usar livremente um método parcial, mesmo que a implementação não seja fornecida. Nenhum erro de tempo de compilação ou em tempo de execução resulta se o método for chamado, mas não implementado.
- **Geradores de origem**: os geradores de origem fornecem uma implementação para membros. O desenvolvedor humano pode adicionar a declaração de membro (geralmente com atributos lidos pelo gerador de origem). O desenvolvedor pode gravar um código que chama esses membros. O gerador de origem é executado durante a compilação e fornece a implementação. Nesse cenário, as restrições para membros parciais que nem sempre são implementadas geralmente não são seguidas.

C#

```
// Definition in file1.cs
partial void OnNameChanged();

// Implementation in file2.cs
partial void OnNameChanged()
{
  // method body
}
```

- As declarações de membro parcial devem começar com a palavra-chave contextual partial.

- As assinaturas de membro parcial em ambas as partes do tipo parcial devem ser correspondentes.
- Métodos de membros podem ter modificadores static e unsafe.
- Membros parciais podem ser genéricos. As restrições devem ser as mesmas na definição e implementação da declaração de membro. Os nomes de parâmetro e de parâmetro de tipo não precisam ser os mesmos na declaração de implementação e de definição.
- Você pode criar um delegado para um método parcial definido e implementado, mas não para um método parcial que não tenha uma implementação.

Especificação da Linguagem C#

Para obter mais informações, confira Tipos parciais e Métodos parciais na Especificação da Linguagem C#. A especificação da linguagem é a fonte definitiva para a sintaxe e o uso de C#. Os novos recursos para membros parciais são definidos nas especificações de recurso para estender métodos parciais, propriedades parciais e indexadores e eventos parciais e construtores.

Confira também

- Classes
- Tipos de estrutura
- Interfaces
- parcial (tipo)

Como retornar subconjuntos de propriedades de elementos em uma consulta (Guia de Programação em C#)

Artigo • 13/03/2024

Use um tipo anônimo em uma expressão de consulta quando essas duas condições se aplicarem:

- Você deseja retornar apenas algumas das propriedades de cada elemento de origem.
- Você não precisa armazenar os resultados da consulta fora do escopo do método em que a consulta é executada.

Se você deseja retornar apenas uma propriedade ou campo de cada elemento de origem, use somente o operador de ponto na cláusula `select`. Por exemplo, para retornar somente a `ID` de cada `student`, escreva a cláusula `select` da seguinte maneira:

C#

```
select student.ID;
```

Exemplo

O exemplo a seguir mostra como usar um tipo anônimo para retornar apenas um subconjunto das propriedades de cada elemento de origem que corresponda à condição especificada.

C#

```
private static void QueryByScore()
{
    // Create the query. var is required because
    // the query produces a sequence of anonymous types.
    var queryHighScores =
        from student in students
        where student.ExamScores[0] > 95
        select new { student.FirstName, student.LastName };

    // Execute the query.
    foreach (var obj in queryHighScores)
    {
        // The anonymous type's properties were not named. Therefore
```

```
        // they have the same names as the Student properties.
        Console.WriteLine(obj.FirstName + ", " + obj.LastName);
    }
}
/* Output:
Adams, Terry
Fakhouri, Fadi
Garcia, Cesar
Omelchenko, Svetlana
Zabokritski, Eugene
*/
```

Observe que o tipo anônimo usa nomes do elemento de origem para suas propriedades, se nenhum nome for especificado. Para fornecer novos nomes para as propriedades no tipo anônimo, escreva a instrução `select` da seguinte maneira:

C#

```
select new { First = student.FirstName, Last = student.LastName };
```

Se você tentar fazer isso no exemplo anterior, a instrução `Console.WriteLine` também deve ser alterada:

C#

```
Console.WriteLine(student.First + " " + student.Last);
```

Compilando o código

Para executar esse código, copie e cole a classe em um aplicativo de console em C# com uma diretiva `using` para System.Linq.

Confira também

- Tipos anônimos
- LINQ em C#

Colaborar conosco no GitHub

A fonte deste conteúdo pode ser encontrada no GitHub, onde

.NET

Comentários do .NET

O .NET é um projeto código aberto. Selecione um link para fornecer comentários:

você também pode criar e revisar problemas e solicitações de pull. Para obter mais informações, confira o nosso guia para colaboradores.

Abrir um problema de documentação

Fornecer comentários sobre o produto

Implementação de interface explícita (guia de programação em C#)

19/06/2025

Se uma classe implementar duas interfaces que contêm um membro com a mesma assinatura, a implementação desse membro na classe fará com que ambas as interfaces usem esse membro como implementação. No exemplo a seguir, todas as chamadas para `Paint` invocar o mesmo método. Este primeiro exemplo define os tipos:

C#

```
public interface IControl
{
    void Paint();
}
public interface ISurface
{
    void Paint();
}
public class SampleClass : IControl, ISurface
{
    // Both ISurface.Paint and IControl.Paint call this method.
    public void Paint()
    {
        Console.WriteLine("Paint method in SampleClass");
    }
}
```

O exemplo a seguir chama os métodos:

C#

```
SampleClass sample = new SampleClass();
IControl control = sample;
ISurface surface = sample;

// The following lines all call the same method.
sample.Paint();
control.Paint();
surface.Paint();

// Output:
// Paint method in SampleClass
// Paint method in SampleClass
// Paint method in SampleClass
```

Mas talvez você não queira que a mesma implementação seja chamada para ambas as interfaces. Para chamar uma implementação diferente dependendo de qual interface está em uso, você pode implementar um membro de interface explicitamente. Uma implementação de interface explícita é um membro de classe que só é chamado por meio da interface especificada. Nomeie o membro da classe prefixando-o com o nome da interface e um período. Por exemplo:

C#

```
public class SampleClass : IControl, ISurface
{
    void IControl.Paint()
    {
        System.Console.WriteLine("IControl.Paint");
    }
    void ISurface.Paint()
    {
        System.Console.WriteLine("ISurface.Paint");
    }
}
```

O membro da `IControl.Paint` classe só está disponível por meio da `IControl` interface e `ISurface.Paint` só está disponível por meio de `ISurface`. Ambas as implementações de método são separadas e nenhuma delas está disponível diretamente na classe. Por exemplo:

C#

```
SampleClass sample = new SampleClass();
IControl control = sample;
ISurface surface = sample;

// The following lines all call the same method.
//sample.Paint(); // Compiler error.
control.Paint();  // Calls IControl.Paint on SampleClass.
surface.Paint();  // Calls ISurface.Paint on SampleClass.

// Output:
// IControl.Paint
// ISurface.Paint
```

A implementação explícita também é usada para resolver casos em que duas interfaces declaram membros diferentes com o mesmo nome, como uma propriedade e um método. Para implementar ambas as interfaces, uma classe precisa usar a implementação explícita para a propriedade `P` ou o método `P`, ou ambos, para evitar um erro do compilador. Por exemplo:

C#

```
interface ILeft
{
    int P { get;}
}
interface IRight
{
    int P();
}

class Middle : ILeft, IRight
{
    public int P() { return 0; }
    int ILeft.P { get { return 0; } }
}
```

Uma implementação de interface explícita não tem um modificador de acesso, pois não é acessível como um membro do tipo em que está definido. Em vez disso, ele só é acessível quando chamado por meio de uma instância da interface. Se você especificar um modificador de acesso para uma implementação de interface explícita, receberá o erro do compilador CS0106. Para obter mais informações, consulte interface (Referência de C#).

Você pode definir uma implementação para membros declarados em uma interface. Se uma classe herdar uma implementação de método de uma interface, esse método só será acessível por meio de uma referência do tipo de interface. O membro herdado não aparece como parte da interface pública. O exemplo a seguir define uma implementação padrão para um método de interface:

C#

```
public interface IControl
{
    void Paint() => Console.WriteLine("Default Paint method");
}
public class SampleClass : IControl
{
    // Paint() is inherited from IControl.
}
```

O exemplo a seguir invoca a implementação padrão:

C#

```
var sample = new SampleClass();
//sample.Paint();// "Paint" isn't accessible.
var control = sample as IControl;
control.Paint();
```

Qualquer classe que implemente a `IControl` interface pode substituir o método padrão `Paint` , seja como um método público ou como uma implementação de interface explícita.

Consulte também

- Programação orientada a objetos
- Interfaces
- Herança

Como implementar os membros da interface explicitamente (Guia de Programação em C#)

Artigo • 13/03/2024

Este exemplo declara uma interface, `IDimensions` e uma classe, `Box`, que implementa explicitamente os membros de interface `GetLength` e `GetWidth`. Os membros são acessados por meio da instância `dimensions` da interface.

Exemplo

C#

```
interface IDimensions
{
    float GetLength();
    float GetWidth();
}

class Box : IDimensions
{
    float lengthInches;
    float widthInches;

    Box(float length, float width)
    {
        lengthInches = length;
        widthInches = width;
    }
    // Explicit interface member implementation:
    float IDimensions.GetLength()
    {
        return lengthInches;
    }
    // Explicit interface member implementation:
    float IDimensions.GetWidth()
    {
        return widthInches;
    }

    static void Main()
    {
        // Declare a class instance box1:
        Box box1 = new Box(30.0f, 20.0f);

        // Declare an interface instance dimensions:
```

```
        IDimensions dimensions = box1;

        // The following commented lines would produce compilation
        // errors because they try to access an explicitly implemented
        // interface member from a class instance:
        //System.Console.WriteLine("Length: {0}", box1.GetLength());
        //System.Console.WriteLine("Width: {0}", box1.GetWidth());

        // Print out the dimensions of the box by calling the methods
        // from an instance of the interface:
        System.Console.WriteLine("Length: {0}", dimensions.GetLength());
        System.Console.WriteLine("Width: {0}", dimensions.GetWidth());
    }
}
/* Output:
    Length: 30
    Width: 20
*/
```

Programação robusta

- Observe que as seguintes linhas, no método `Main`, foram comentadas, pois produziriam erros de compilação. Um membro de interface implementado explicitamente não pode ser acessado de uma instância de classe:

  ```
  C#

  //System.Console.WriteLine("Length: {0}", box1.GetLength());
  //System.Console.WriteLine("Width: {0}", box1.GetWidth());
  ```

- Observe também que as linhas a seguir, no método `Main`, imprimem com êxito as dimensões da caixa, pois os métodos estão sendo chamados de uma instância da interface:

  ```
  C#

  System.Console.WriteLine("Length: {0}", dimensions.GetLength());
  System.Console.WriteLine("Width: {0}", dimensions.GetWidth());
  ```

Confira também

- Programação orientada a objetos
- Interfaces
- Como implementar membros de duas interfaces explicitamente

Colaborar conosco no GitHub

A fonte deste conteúdo pode ser encontrada no GitHub, onde você também pode criar e revisar problemas e solicitações de pull. Para obter mais informações, confira o nosso guia para colaboradores.

Comentários do .NET

O .NET é um projeto código aberto. Selecione um link para fornecer comentários:

- Abrir um problema de documentação
- Fornecer comentários sobre o produto

Como implementar explicitamente membros de duas interfaces (Guia de Programação em C#)

Artigo • 13/03/2024

A implementação explícita da interface também permite ao programador implementar duas interfaces que têm os mesmos nomes de membro e implementar separadamente cada membro de interface. Este exemplo exibe as dimensões de uma caixa em unidades inglesas e no sistema métrico. A Caixa classe implementa duas interfaces, IEnglishDimensions e IMetricDimensions, que representam os diferentes sistemas de medida. As duas interfaces têm nomes de membro idênticos, Comprimento e Largura.

Exemplo

C#

```
// Declare the English units interface:
interface IEnglishDimensions
{
    float Length();
    float Width();
}

// Declare the metric units interface:
interface IMetricDimensions
{
    float Length();
    float Width();
}

// Declare the Box class that implements the two interfaces:
// IEnglishDimensions and IMetricDimensions:
class Box : IEnglishDimensions, IMetricDimensions
{
    float lengthInches;
    float widthInches;

    public Box(float lengthInches, float widthInches)
    {
        this.lengthInches = lengthInches;
        this.widthInches = widthInches;
    }

    // Explicitly implement the members of IEnglishDimensions:
    float IEnglishDimensions.Length() => lengthInches;
```

```
    float IEnglishDimensions.Width() => widthInches;

    // Explicitly implement the members of IMetricDimensions:
    float IMetricDimensions.Length() => lengthInches * 2.54f;

    float IMetricDimensions.Width() => widthInches * 2.54f;

    static void Main()
    {
        // Declare a class instance box1:
        Box box1 = new Box(30.0f, 20.0f);

        // Declare an instance of the English units interface:
        IEnglishDimensions eDimensions = box1;

        // Declare an instance of the metric units interface:
        IMetricDimensions mDimensions = box1;

        // Print dimensions in English units:
        System.Console.WriteLine("Length(in): {0}", eDimensions.Length());
        System.Console.WriteLine("Width (in): {0}", eDimensions.Width());

        // Print dimensions in metric units:
        System.Console.WriteLine("Length(cm): {0}", mDimensions.Length());
        System.Console.WriteLine("Width (cm): {0}", mDimensions.Width());
    }
}
/* Output:
    Length(in): 30
    Width (in): 20
    Length(cm): 76.2
    Width (cm): 50.8
*/
```

Programação robusta

Caso deseje que as medidas padrão estejam unidades inglesas, implemente os métodos Comprimento e Largura normalmente e implemente explicitamente os métodos Comprimento e Largura da interface IMetricDimensions:

C#

```
// Normal implementation:
public float Length() => lengthInches;
public float Width() => widthInches;

// Explicit implementation:
float IMetricDimensions.Length() => lengthInches * 2.54f;
float IMetricDimensions.Width() => widthInches * 2.54f;
```

Nesse caso, é possível acessar as unidades inglesas da instância de classe e acessar as unidades métricas da instância da interface:

C#

```
public static void Test()
{
    Box box1 = new Box(30.0f, 20.0f);
    IMetricDimensions mDimensions = box1;

    System.Console.WriteLine("Length(in): {0}", box1.Length());
    System.Console.WriteLine("Width (in): {0}", box1.Width());
    System.Console.WriteLine("Length(cm): {0}", mDimensions.Length());
    System.Console.WriteLine("Width (cm): {0}", mDimensions.Width());
}
```

Confira também

- Programação orientada a objetos
- Interfaces
- Como implementar membros de interface explicitamente

Colaborar conosco no GitHub

A fonte deste conteúdo pode ser encontrada no GitHub, onde você também pode criar e revisar problemas e solicitações de pull. Para obter mais informações, confira o nosso guia para colaboradores.

.NET

Comentários do .NET

O .NET é um projeto código aberto. Selecione um link para fornecer comentários:

Abrir um problema de documentação

Fornecer comentários sobre o produto

Delegados (Guia de Programação em C#)

19/06/2025

Um delegado é um tipo que representa referências aos métodos com lista de parâmetros e tipo de retorno específicos. Ao instanciar um delegado, você pode associar a instância de delegado a qualquer método com assinatura e tipo de retorno compatíveis. Você pode invocar (ou chamar) o método por meio da instância delegada.

Delegados são usados para passar métodos como argumentos a outros métodos. Os manipuladores de eventos são essencialmente métodos que você invoca por meio de delegados. Quando você cria um método personalizado, uma classe como um controle do Windows pode chamar seu método quando um determinado evento ocorre.

O exemplo a seguir mostra uma declaração de delegado:

C#

```
public delegate int PerformCalculation(int x, int y);
```

Você pode atribuir qualquer método de qualquer classe ou struct acessível que corresponda ao tipo de delegado ao delegado. O método pode ser estático ou um método de instância. A flexibilidade permite que você altere programaticamente as chamadas de método ou conecte um novo código às classes existentes.

> ⓘ **Observação**
>
> No contexto de sobrecarga de método, a assinatura de um método não inclui o valor retornado. No entanto, no contexto de delegados, a assinatura inclui o valor retornado. Em outras palavras, um método deve ter um tipo de retorno compatível como o tipo de retorno declarado pelo delegado.

A capacidade de referir-se a um método como um parâmetro torna os delegados ideais para definir métodos de callback. Você pode escrever um método que compara dois objetos em seu aplicativo. O método pode, então, ser usado em um delegado para um algoritmo de classificação. Como o código de comparação é separado da biblioteca, o método de classificação pode ser mais geral.

Os ponteiros de função dão suporte a cenários semelhantes, em que você precisa de mais controle sobre a convenção de chamada. O código associado a um delegado é invocado usando um método virtual adicionado a um tipo de delegado. Ao trabalhar com ponteiros de função, você pode especificar convenções diferentes.

Explorar características de delegados

Os delegados têm as seguintes características:

- Os delegados permitem que os métodos sejam passados como parâmetros.
- Os delegados podem ser usados para definir métodos de retorno de chamada.
- Os delegados podem ser encadeados, como chamar vários métodos em um único evento.
- Os métodos não precisam corresponder exatamente ao tipo delegado. Para obter mais informações, confira Usando a variação delegados.
- Expressões lambda são uma maneira mais concisa de escrever blocos de código diretamente. Expressões lambda (em determinados contextos) são compiladas para delegar tipos. Para obter mais informações sobre expressões lambda, consulte expressões Lambda.

Revisar artigos relacionados

Para obter mais informações sobre delegados, consulte os seguintes artigos:

- Usar delegados
- Delegados com métodos nomeados vs. anônimos
- Usar a variação em delegados
- Como combinar delegados (delegados multicast)
- Como declarar, instanciar e usar um delegado

Acessar a especificação da linguagem C#

A especificação de idioma é a fonte definitiva para a sintaxe e o uso de C#. Para obter mais informações, consulte Delegados na Especificação da Linguagem C#.

Links relacionados

- Delegate
- Eventos

Usando delegados (Guia de Programação em C#)

19/06/2025

Um delegado é um tipo que encapsula com segurança um método, semelhante a um ponteiro de função em C e C++. No entanto, ao contrário dos ponteiros de função de C, delegados são orientados a objeto, fortemente tipados e seguros. O exemplo a seguir declara um delegado nomeado `Callback` que pode encapsular um método que usa uma cadeia de caracteres como um argumento e retorna nulo:

C#

```
public delegate void Callback(string message);
```

Um objeto delegado normalmente é construído fornecendo o nome do método que o delegado encapsula ou com uma expressão lambda. Um delegado pode ser invocado uma vez instanciado dessa maneira. A invocação de um representante chama o método anexado à instância delegada. Os parâmetros passados para o delegado pelo chamador são passados para o método. O delegado retorna o valor de retorno, se houver, do método. Por exemplo:

C#

```
// Create a method for a delegate.
public static void DelegateMethod(string message)
{
    Console.WriteLine(message);
}
```

C#

```
// Instantiate the delegate.
Callback handler = DelegateMethod;

// Call the delegate.
handler("Hello World");
```

Os tipos de delegado são derivados da Delegate classe no .NET. Os tipos de delegado são selados; eles não podem ser derivados, e não é possível derivar classes personalizadas de Delegate. Como o delegado instanciado é um objeto, ele pode ser passado como um argumento ou atribuído a uma propriedade. Um método pode aceitar um delegado como um parâmetro e chamar o delegado posteriormente. Isso é conhecido como um retorno de chamada assíncrono e é um método comum de notificação de um chamador quando um

processo longo é concluído. Quando um delegado é usado dessa forma, o código que usa o delegado não precisa de nenhum conhecimento da implementação do método que está sendo usado. A funcionalidade é semelhante à oferecida pelas interfaces de encapsulamento.

Outro uso comum de chamadas de retorno é definir um método de comparação personalizada e passar esse delegado para um método de classificação. Ele permite que o código do chamador se torne parte do algoritmo de classificação. O seguinte método de exemplo usa o `Del` tipo como um parâmetro:

C#

```
public static void MethodWithCallback(int param1, int param2, Callback callback)
{
    callback("The number is: " + (param1 + param2).ToString());
}
```

Em seguida, você pode passar o delegado criado no exemplo anterior para esse método:

C#

```
MethodWithCallback(1, 2, handler);
```

E receber a seguinte saída para o console:

Console

```
The number is: 3
```

`MethodWithCallback` não precisa acessar o console diretamente, nem ser projetado pensando em um console. O que `MethodWithCallback` faz é preparar uma cadeia de caracteres e passar a cadeia de caracteres para outro método. Um método delegado pode usar qualquer número de parâmetros.

Quando um delegado é construído para encapsular um método de instância, o delegado faz referência à instância e ao método. Um delegado não tem conhecimento do tipo de instância, exceto pelo método que envolve. Um delegado pode se referir a qualquer tipo de objeto, desde que haja um método nesse objeto que corresponda à assinatura do delegado. Quando um delegado é construído para encapsular um método estático, ele faz referência apenas ao método. Considere as seguintes declarações:

C#

```
public class MethodClass
{
```

```
    public void Method1(string message) { }
    public void Method2(string message) { }
}
```

Juntamente com a estática `DelegateMethod` mostrada anteriormente, agora temos três métodos que você pode encapsular em uma `Del` instância.

Um delegado pode chamar mais de um método quando invocado, conhecido como multicasting. Para adicionar um método extra à lista de métodos do delegado — a lista de invocação — basta adicionar dois delegados usando os operadores de adição ou de atribuição de adição ('+' ou '+ ='). Por exemplo:

C#

```
var obj = new MethodClass();
Callback d1 = obj.Method1;
Callback d2 = obj.Method2;
Callback d3 = DelegateMethod;

//Both types of assignment are valid.
Callback allMethodsDelegate = d1 + d2;
allMethodsDelegate += d3;
```

O `allMethodsDelegate` contém três métodos em sua lista de invocação — `Method1` `Method2` e `DelegateMethod`. Os três delegados originais, `d1` e `d2` `d3`, permanecem inalterados. Quando `allMethodsDelegate` é invocado, todos os três métodos são chamados em ordem. Se o delegado usar parâmetros de referência, a referência será passada sequencialmente para cada um dos três métodos, e quaisquer alterações por um método ficarão visíveis para o próximo método. Quando qualquer um dos métodos gera uma exceção que não é capturada dentro do método, essa exceção é passada para o chamador do delegado. Nenhum método subsequente na lista de invocação é chamado. Se o delegado tiver um valor retornado e/ou parâmetros externos, ele retornará o valor retornado e os parâmetros do último método invocado. Para remover um método da lista de invocação, use os operadores de subtração ou atribuição de subtração (`-` ou `-=`). Por exemplo:

C#

```
//remove Method1
allMethodsDelegate -= d1;

// copy AllMethodsDelegate while removing d2
Callback oneMethodDelegate = (allMethodsDelegate - d2)!;
```

Como os tipos de delegado são derivados de `System.Delegate`, os métodos e as propriedades definidos por essa classe podem ser invocados pelo delegado. Por exemplo, para localizar o número de métodos na lista de invocação de um delegado, você pode escrever:

C#

```
int invocationCount = d1.GetInvocationList().GetLength(0);
```

Delegados com mais de um método em sua lista de invocação derivam de MulticastDelegate, que é uma subclasse de `System.Delegate`. O código anterior funciona em ambos os casos porque ambas as classes dão suporte `GetInvocationList`.

Delegados multicast são amplamente usados na manipulação de eventos. Os objetos de origem do evento enviam notificações de evento para objetos de destinatário que se registraram para receber esse evento. Para se registrar em um evento, o destinatário cria um método projetado para manipular o evento e, em seguida, cria um delegado para esse método e passa o delegado para a origem do evento. A origem chama o delegado quando o evento ocorre. O delegado chama então o método de manipulação de eventos no destinatário, fornecendo os dados do evento. A origem do evento define o tipo de delegado para um determinado evento. Para obter mais informações, consulte Eventos.

A comparação de delegados de dois tipos diferentes atribuídos no tempo de compilação resulta em um erro de compilação. Se as instâncias delegadas forem estaticamente do tipo `System.Delegate`, a comparação será permitida, mas retornará false em tempo de execução. Por exemplo:

C#

```
delegate void Callback1();
delegate void Callback2();

static void method(Callback1 d, Callback2 e, System.Delegate f)
{
    // Compile-time error.
    Console.WriteLine(d == e);

    // OK at compile-time. False if the run-time type of f
    // is not the same as that of d.
    Console.WriteLine(d == f);
}
```

Consulte também

- Representantes

- Usar a variação em delegados
- Variação dos Delegados
- Usar variância para delegados genéricos de Func e Action
- Eventos

Delegados com métodos nomeados versus anônimos (Guia de Programação em C#)

17/06/2025

Um delegado pode ser associado a um método nomeado. Ao instanciar um delegado usando um método nomeado, o método é passado como um parâmetro, por exemplo:

C#

```
// Declare a delegate.
delegate void WorkCallback(int x);

// Define a named method.
void DoWork(int k) { /* ... */ }

// Instantiate the delegate using the method as a parameter.
WorkCallback d = obj.DoWork;
```

O exemplo anterior usa um método nomeado. Os delegados construídos com um método nomeado podem encapsular um método estático ou um método de instância. Métodos nomeados são a única maneira de instanciar um delegado nas versões anteriores do C#. O C# permite criar uma instância de um delegado e especificar imediatamente um bloco de código que o delegado processa quando chamado. O bloco pode conter uma expressão lambda ou um método anônimo, conforme mostrado no exemplo a seguir:

C#

```
// Declare a delegate.
delegate void WorkCallback(int x);

// Instantiate the delegate using an anonymous method.
WorkCallback d = (int k) => { /* ... */ };
```

O método passado como parâmetro delegado deve ter a mesma assinatura da declaração delegada. Uma instância delegada pode encapsular métodos estáticos ou de instância.

> **Observação**
>
> Embora o delegado possa usar um parâmetro **out**, não é recomendável utilizá-lo com delegados de evento multicast, pois não é possível saber qual delegado será chamado.

Os grupos de métodos com uma única sobrecarga têm um *tipo natural*. O compilador pode inferir o tipo de retorno e os tipos de parâmetro para o tipo delegado:

C#

```
var read = Console.Read; // Just one overload; Func<int> inferred
var write = Console.Write; // ERROR: Multiple overloads, can't choose
```

Exemplos

O exemplo a seguir é um exemplo simples de declarar e usar um delegado. Observe que tanto o delegado, `MultiplyCallback` e o método associado, `MultiplyNumbers`, têm a mesma assinatura

C#

```
// Declare a delegate
delegate void MultiplyCallback(int i, double j);

class MathClass
{
    static void Main()
    {
        MathClass m = new MathClass();

        // Delegate instantiation using "MultiplyNumbers"
        MultiplyCallback d = m.MultiplyNumbers;

        // Invoke the delegate object.
        Console.WriteLine("Invoking the delegate using 'MultiplyNumbers':");
        for (int i = 1; i <= 5; i++)
        {
            d(i, 2);
        }

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }

    // Declare the associated method.
    void MultiplyNumbers(int m, double n)
    {
        Console.Write(m * n + " ");
    }
}
/* Output:
    Invoking the delegate using 'MultiplyNumbers':
    2 4 6 8 10
*/
```

No exemplo a seguir, um delegado é mapeado para métodos estáticos e de instância e retorna informações específicas sobre cada um.

C#

```
// Declare a delegate
delegate void Callback();

class SampleClass
{
    public void InstanceMethod()
    {
        Console.WriteLine("A message from the instance method.");
    }

    static public void StaticMethod()
    {
        Console.WriteLine("A message from the static method.");
    }
}

class TestSampleClass
{
    static void Main()
    {
        var sc = new SampleClass();

        // Map the delegate to the instance method:
        Callback d = sc.InstanceMethod;
        d();

        // Map to the static method:
        d = SampleClass.StaticMethod;
        d();
    }
}
/* Output:
    A message from the instance method.
    A message from the static method.
*/
```

Confira também

- Representantes
- Como combinar delegados (delegados multicast)
- Eventos

Como combinar delegados (delegados multicast) (Guia de Programação em C#)

Artigo • 13/03/2024

Este exemplo demonstra como criar delegados multicast. Uma propriedade útil de objetos delegados é que vários objetos podem ser atribuídos a uma instância delegada usando o operador `+`. O delegado multicast contém uma lista dos delegados atribuídos. Quando o delegado multicast é chamado, ele invoca os delegados da lista, em ordem. Apenas os delegados do mesmo tipo podem ser combinados.

O operador `-` pode ser usado para remover um delegado de componente de um delegado multicast.

Exemplo

C#

```
using System;

// Define a custom delegate that has a string parameter and returns void.
delegate void CustomCallback(string s);

class TestClass
{
    // Define two methods that have the same signature as CustomCallback.
    static void Hello(string s)
    {
        Console.WriteLine($"  Hello, {s}!");
    }

    static void Goodbye(string s)
    {
        Console.WriteLine($"  Goodbye, {s}!");
    }

    static void Main()
    {
        // Declare instances of the custom delegate.
        CustomCallback hiDel, byeDel, multiDel, multiMinusHiDel;

        // In this example, you can omit the custom delegate if you
        // want to and use Action<string> instead.
        //Action<string> hiDel, byeDel, multiDel, multiMinusHiDel;

        // Initialize the delegate object hiDel that references the
        // method Hello.
```

```
        hiDel = Hello;

        // Initialize the delegate object byeDel that references the
        // method Goodbye.
        byeDel = Goodbye;

        // The two delegates, hiDel and byeDel, are combined to
        // form multiDel.
        multiDel = hiDel + byeDel;

        // Remove hiDel from the multicast delegate, leaving byeDel,
        // which calls only the method Goodbye.
        multiMinusHiDel = multiDel - hiDel;

        Console.WriteLine("Invoking delegate hiDel:");
        hiDel("A");
        Console.WriteLine("Invoking delegate byeDel:");
        byeDel("B");
        Console.WriteLine("Invoking delegate multiDel:");
        multiDel("C");
        Console.WriteLine("Invoking delegate multiMinusHiDel:");
        multiMinusHiDel("D");
    }
}
/* Output:
Invoking delegate hiDel:
  Hello, A!
Invoking delegate byeDel:
  Goodbye, B!
Invoking delegate multiDel:
  Hello, C!
  Goodbye, C!
Invoking delegate multiMinusHiDel:
  Goodbye, D!
*/
```

Confira também

- MulticastDelegate
- Eventos

Colaborar conosco no GitHub

A fonte deste conteúdo pode ser encontrada no GitHub, onde você também pode criar e revisar problemas e solicitações

.NET

Comentários do .NET

O .NET é um projeto código aberto. Selecione um link para fornecer comentários:

de pull. Para obter mais informações, confira o nosso guia para colaboradores.

Abrir um problema de documentação

Fornecer comentários sobre o produto

Como declarar, instanciar e usar um delegado (Guia de Programação em C#)

Artigo • 13/03/2024

Você pode declarar delegados usando qualquer um dos seguintes métodos:

- Declarar um tipo de delegado e declarar um método com uma assinatura correspondente:

C#

```
// Declare a delegate.
delegate void NotifyCallback(string str);

// Declare a method with the same signature as the delegate.
static void Notify(string name)
{
    Console.WriteLine($"Notification received for: {name}");
}
```

C#

```
// Create an instance of the delegate.
NotifyCallback del1 = new NotifyCallback(Notify);
```

- Atribuir um grupo de métodos a um tipo de delegado:

C#

```
// C# 2.0 provides a simpler way to declare an instance of NotifyCallback.
NotifyCallback del2 = Notify;
```

- Declarar um método anônimo:

C#

```
// Instantiate NotifyCallback by using an anonymous method.
NotifyCallback del3 = delegate(string name)
    { Console.WriteLine($"Notification received for: {name}"); };
```

- Usar uma expressão lambda:

C#

```
// Instantiate NotifyCallback by using a lambda expression.
NotifyCallback del4 = name =>  { Console.WriteLine($"Notification received
for: {name}"); };
```

Para obter mais informações, consulte Expressões Lambda.

O exemplo a seguir ilustra a declaração, instanciação e o uso de um delegado. A classe `BookDB` encapsula um banco de dados de uma livraria que mantém um banco de dados de livros. Ela expõe um método, `ProcessPaperbackBooks`, que localiza todos os livros de bolso no banco de dados e chama um delegado para cada um. O tipo `delegate` usado tem o nome `ProcessBookCallback`. A classe `Test` usa essa classe para imprimir os títulos e o preço médio dos livros de bolso.

O uso de delegados promove uma boa separação de funcionalidade entre o banco de dados da livraria e o código de cliente. O código de cliente não tem conhecimento de como os livros são armazenados ou como o código da livraria localiza os livros de bolso. O código da livraria não tem conhecimento do processamento executado nos livros de bolso após a localização.

Exemplo

C#

```
// A set of classes for handling a bookstore:
namespace Bookstore
{
    using System.Collections;

    // Describes a book in the book list:
    public struct Book
    {
        public string Title;        // Title of the book.
        public string Author;       // Author of the book.
        public decimal Price;       // Price of the book.
        public bool Paperback;      // Is it paperback?

        public Book(string title, string author, decimal price, bool
paperBack)
        {
            Title = title;
            Author = author;
            Price = price;
            Paperback = paperBack;
        }
    }

    // Declare a delegate type for processing a book:
```

```
    public delegate void ProcessBookCallback(Book book);

    // Maintains a book database.
    public class BookDB
    {
        // List of all books in the database:
        ArrayList list = new ArrayList();

        // Add a book to the database:
        public void AddBook(string title, string author, decimal price, bool
paperBack)
        {
            list.Add(new Book(title, author, price, paperBack));
        }

        // Call a passed-in delegate on each paperback book to process it:
        public void ProcessPaperbackBooks(ProcessBookCallback processBook)
        {
            foreach (Book b in list)
            {
                if (b.Paperback)
                    // Calling the delegate:
                    processBook(b);
            }
        }
    }
}

// Using the Bookstore classes:
namespace BookTestClient
{
    using Bookstore;

    // Class to total and average prices of books:
    class PriceTotaller
    {
        int countBooks = 0;
        decimal priceBooks = 0.0m;

        internal void AddBookToTotal(Book book)
        {
            countBooks += 1;
            priceBooks += book.Price;
        }

        internal decimal AveragePrice()
        {
            return priceBooks / countBooks;
        }
    }

    // Class to test the book database:
    class Test
    {
        // Print the title of the book.
```

```
        static void PrintTitle(Book b)
        {
            Console.WriteLine($"   {b.Title}");
        }

        // Execution starts here.
        static void Main()
        {
            BookDB bookDB = new BookDB();

            // Initialize the database with some books:
            AddBooks(bookDB);

            // Print all the titles of paperbacks:
            Console.WriteLine("Paperback Book Titles:");

            // Create a new delegate object associated with the static
            // method Test.PrintTitle:
            bookDB.ProcessPaperbackBooks(PrintTitle);

            // Get the average price of a paperback by using
            // a PriceTotaller object:
            PriceTotaller totaller = new PriceTotaller();

            // Create a new delegate object associated with the nonstatic
            // method AddBookToTotal on the object totaller:
            bookDB.ProcessPaperbackBooks(totaller.AddBookToTotal);

            Console.WriteLine("Average Paperback Book Price: ${0:#.##}",
                    totaller.AveragePrice());
        }

        // Initialize the book database with some test books:
        static void AddBooks(BookDB bookDB)
        {
            bookDB.AddBook("The C Programming Language", "Brian W. Kernighan
and Dennis M. Ritchie", 19.95m, true);
            bookDB.AddBook("The Unicode Standard 2.0", "The Unicode
Consortium", 39.95m, true);
            bookDB.AddBook("The MS-DOS Encyclopedia", "Ray Duncan", 129.95m,
false);
            bookDB.AddBook("Dogbert's Clues for the Clueless", "Scott
Adams", 12.00m, true);
        }
    }
}
/* Output:
Paperback Book Titles:
   The C Programming Language
   The Unicode Standard 2.0
   Dogbert's Clues for the Clueless
Average Paperback Book Price: $23.97
*/
```

Programação robusta

- Declarando um delegado.

  A instrução a seguir declara um novo tipo de delegado.

  ```
  C#
  public delegate void ProcessBookCallback(Book book);
  ```

  Cada tipo de delegado descreve o número e os tipos dos argumentos e o tipo do valor retornado dos métodos que pode encapsular. Sempre que um novo conjunto de tipos de argumento ou tipo de valor retornado for necessário, um novo tipo de delegado deverá ser declarado.

- Instanciando um delegado.

  Após a declaração do tipo de delegado, um objeto delegado deve ser criado e associado a um método específico. No exemplo anterior, faça isso passando o método `PrintTitle` para o método `ProcessPaperbackBooks`, como no exemplo a seguir:

  ```
  C#
  bookDB.ProcessPaperbackBooks(PrintTitle);
  ```

  Isso cria um novo objeto delegado associado ao método estático `Test.PrintTitle`. Da mesma forma, o método não estático `AddBookToTotal` no objeto `totaller` é passado como no exemplo a seguir:

  ```
  C#
  bookDB.ProcessPaperbackBooks(totaller.AddBookToTotal);
  ```

  Em ambos os casos, um novo objeto delegado é passado para o método `ProcessPaperbackBooks`.

  Após a criação de um delegado, o método ao qual ele está associado nunca se altera; objetos delegados são imutáveis.

- Chamando um delegado.

  Normalmente, o objeto delegado, após sua criação, é passado para outro código que chamará o delegado. Um objeto delegado é chamado usando seu nome

seguido dos argumentos entre parênteses a serem passados para o delegado. A seguir, veja um exemplo de uma chamada de delegado:

```
C#

processBook(b);
```

Um delegado pode ser chamado de forma síncrona, como neste exemplo ou de forma assíncrona, usando os métodos `BeginInvoke` e `EndInvoke`.

Confira também

- Eventos
- Representantes

Colaborar conosco no GitHub

A fonte deste conteúdo pode ser encontrada no GitHub, onde você também pode criar e revisar problemas e solicitações de pull. Para obter mais informações, confira o nosso guia para colaboradores.

.NET

Comentários do .NET

O .NET é um projeto código aberto. Selecione um link para fornecer comentários:

Abrir um problema de documentação

Fornecer comentários sobre o produto

Cadeias de caracteres e literais de cadeia de caracteres

Artigo • 22/11/2024

Uma cadeia de caracteres é um objeto do tipo String cujo valor é texto. Internamente, o texto é armazenado como uma coleção sequencial somente leitura de objetos Char. A propriedade Length de uma cadeia de caracteres representa o número de objetos `Char` que ela contém, não o número de caracteres Unicode. Para acessar os pontos de código Unicode individuais em uma cadeia de caracteres, use o objeto StringInfo.

string vs. System.String

Em C#, a palavra-chave `string` é um alias para String; portanto, `String` e `string` são equivalentes. Use o alias fornecido `string` como ele funciona mesmo sem `using System;`. A classe `String` fornece muitos métodos para criar, manipular e comparar cadeias de caracteres com segurança. Além disso, a linguagem C# sobrecarrega alguns operadores para simplificar operações comuns de cadeia de caracteres. Para obter mais informações sobre a palavra-chave, consulte string. Para obter mais informações sobre o tipo e seus métodos, consulte String.

Declarando e inicializando cadeias de caracteres

Você pode declarar e inicializar cadeias de caracteres de várias maneiras, conforme mostrado no exemplo a seguir:

C#

```
// Declare without initializing.
string message1;

// Initialize to null.
string? message2 = null;

// Initialize as an empty string.
// Use the Empty constant instead of the literal "".
string message3 = System.String.Empty;

// Initialize with a regular string literal.
string oldPath = "c:\\Program Files\\Microsoft Visual Studio 8.0";
```

```
// Initialize with a verbatim string literal.
string newPath = @"c:\Program Files\Microsoft Visual Studio 9.0";

// Use System.String if you prefer.
System.String greeting = "Hello World!";

// In local variables (i.e. within a method body)
// you can use implicit typing.
var temp = "I'm still a strongly-typed System.String!";

// Use a const string to prevent 'message4' from
// being used to store another string value.
const string message4 = "You can't get rid of me!";

// Use the String constructor only when creating
// a string from a char*, char[], or sbyte*. See
// System.String documentation for details.
char[] letters = { 'A', 'B', 'C' };
string alphabet = new string(letters);
```

Você não usa o operador novo para criar um objeto de cadeia de caracteres, exceto ao inicializar a cadeia de caracteres com uma matriz de caracteres.

Inicialize uma cadeia de caracteres com o valor Empty constante para criar um novo objeto String cuja cadeia de caracteres é de comprimento zero. O literal de cadeia de caracteres de uma cadeia de caracteres de comprimento zero é `""`. Ao inicializar cadeias de caracteres com o valor Empty em vez de nulo, você poderá reduzir as chances de uma NullReferenceException ocorrer. Use o método de IsNullOrEmpty(String) estático para verificar o valor de uma cadeia de caracteres antes de tentar acessá-la.

Imutabilidade de cadeias de caracteres

Objetos de cadeia de caracteres são imutáveis: eles não podem ser alterados depois de criados. Todos os métodos String e operadores C# que parecem modificar uma cadeia de caracteres realmente retornam os resultados em um novo objeto de cadeia de caracteres. No exemplo a seguir, quando o conteúdo de `s1` e `s2` são concatenados para formar uma única cadeia de caracteres, as duas cadeias de caracteres originais não são modificadas. O operador `+=` cria uma nova cadeia de caracteres que contém o conteúdo combinado. Esse novo objeto é atribuído à variável `s1` e o objeto original atribuído a `s1` é liberado para coleta de lixo porque nenhuma outra variável contém uma referência a ele.

C#

```
string s1 = "A string is more ";
string s2 = "than the sum of its chars.";
```

```
// Concatenate s1 and s2. This actually creates a new
// string object and stores it in s1, releasing the
// reference to the original object.
s1 += s2;

System.Console.WriteLine(s1);
// Output: A string is more than the sum of its chars.
```

Como uma "modificação" de cadeia de caracteres é, na verdade, uma nova criação de cadeia de caracteres, você deve ter cuidado ao criar referências a cadeias de caracteres. Se você criar uma referência a uma cadeia de caracteres e depois "modificar" a cadeia de caracteres original, a referência continuará apontando para o objeto original em vez do novo objeto que foi criado quando a cadeia de caracteres foi modificada. O código a seguir ilustra esse comportamento:

C#

```
string str1 = "Hello ";
string str2 = str1;
str1 += "World";

System.Console.WriteLine(str2);
//Output: Hello
```

Para obter mais informações sobre como criar novas cadeias de caracteres baseadas em modificações como operações de pesquisa e substituição na cadeia de caracteres original, consulte Como modificar o conteúdo da cadeia de caracteres.

Literais de cadeia de caracteres entre aspas

Os literais de cadeia de caracteres entre aspas iniciam e terminam com um único caractere de aspas duplas (`"`) na mesma linha. Literais de cadeia de caracteres entre aspas são mais adequados para cadeias de caracteres que se encaixam em uma única linha e não incluem sequências de escape. Um literal de cadeia de caracteres entre aspas deve inserir caracteres de escape, conforme demonstrado no exemplo a seguir:

C#

```
string columns = "Column 1\tColumn 2\tColumn 3";
//Output: Column 1        Column 2        Column 3

string rows = "Row 1\r\nRow 2\r\nRow 3";
/* Output:
    Row 1
    Row 2
```

```
    Row 3
*/

string title = "\"The \u00C6olean Harp\", by Samuel Taylor Coleridge";
//Output: "The Æolean Harp", by Samuel Taylor Coleridge
```

Literais de cadeia de caracteres textuais

Literais de cadeia de caracteres verbatim são mais convenientes para cadeias de caracteres de várias linhas, cadeias de caracteres que contêm caracteres de barra invertida ou aspas duplas inseridas. Cadeias de caracteres verbatim preservam caracteres de nova linha como parte do texto da cadeia de caracteres. Use aspas duplas para inserir uma marca de aspas simples dentro de uma cadeia de caracteres textual. O exemplo a seguir mostra alguns usos comuns para cadeias de caracteres verbatim:

C#

```
string title = "\"The \u00C6olean Harp\", by Samuel Taylor Coleridge";
//Output: "The Æolean Harp", by Samuel Taylor Coleridge

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."
```

Literais brutos de cadeia de caracteres

A partir do C# 11, é possível usar *literais de cadeia de caracteres bruta* para criar cadeias de caracteres multilinha com mais facilidade ou usar quaisquer caracteres que exijam sequências de escape. *Literais de cadeia de caracteres bruta* removem a necessidade de usar sequências de escape. Você pode escrever a cadeia de caracteres, incluindo a formatação de espaço em branco, como deseja que ela apareça na saída. Um *literal de cadeia de caracteres bruta*:

- Inicia e termina com uma sequência de pelo menos três caracteres de aspas duplas (`"""`). Você pode usar mais de três caracteres consecutivos iniciem e terminem a sequência para dar suporte a literais de cadeia de caracteres que contêm três (ou mais) caracteres de aspas repetidos.
- Literais de cadeia de caracteres bruta de linha única exigem os caracteres de aspas de abertura e fechamento na mesma linha.
- Literais de cadeia de caracteres bruta multilinha exigem caracteres de aspas de abertura e fechamento em suas próprias linhas.
- Em literais de cadeia de caracteres brutos de várias linhas, qualquer espaço em branco à esquerda das aspas de fechamento é removido de todas as linhas do literal da cadeia de caracteres bruta.
- Em literais de cadeia de caracteres brutos de várias linhas, o espaço em branco após a citação de abertura na mesma linha é ignorado.
- Em literais de cadeia de caracteres brutos de várias linhas, somente as linhas de espaço em branco após a cotação de abertura são incluídas no literal da cadeia de caracteres.

Os exemplos a seguir demonstram estas regras:

C#

```
string singleLine = """Friends say "hello" as they pass by.""";
string multiLine = """
    "Hello World!" is typically the first program someone writes.
    """;
string embeddedXML = """
       <element attr = "content">
           <body style="normal">
               Here is the main text
           </body>
           <footer>
               Excerpts from "An amazing story"
           </footer>
       </element >
       """;
// The line "<element attr = "content">" starts in the first column.
// All whitespace left of that column is removed from the string.

string rawStringLiteralDelimiter = """"
    Raw string literals are delimited
    by a string of at least three double quotes,
    like this: """
    """";
```

Os exemplos a seguir demonstram os erros do compilador relatados com base nessas regras:

C#

```
// CS8997: Unterminated raw string literal.
var multiLineStart = """This
    is the beginning of a string
    """;

// CS9000: Raw string literal delimiter must be on its own line.
var multiLineEnd = """
    This is the beginning of a string """;

// CS8999: Line does not start with the same whitespace as the closing line
// of the raw string literal
var noOutdenting = """
    A line of text.
Trying to outdent the second line.
    """;
```

Os dois primeiros exemplos são inválidos porque literais de cadeia de caracteres bruta multilinha exigem a sequência de aspas de abertura e fechamento em sua própria linha. O terceiro exemplo é inválido porque o texto é retirado da sequência de aspas de fechamento.

Você deve considerar literais de cadeia de caracteres bruta ao gerar texto que inclua caracteres que exigem sequências de escape ao usar literais de cadeia de caracteres entre aspas ou literais de cadeia de caracteres verbatim. Os literais de cadeia de caracteres bruta são lidos com mais facilidade por você e outras pessoas, pois se parecem com o texto de saída. Por exemplo, considere o seguinte código que inclui uma cadeia de caracteres de JSON formatado:

C#

```
string jsonString = """
{
  "Date": "2019-08-01T00:00:00-07:00",
  "TemperatureCelsius": 25,
  "Summary": "Hot",
  "DatesAvailable": [
    "2019-08-01T00:00:00-07:00",
    "2019-08-02T00:00:00-07:00"
  ],
  "TemperatureRanges": {
    "Cold": {
      "High": 20,
      "Low": -10
    },
    "Hot": {
      "High": 60,
      "Low": 20
    }
```

```
            },
  "SummaryWords": [
    "Cool",
    "Windy",
    "Humid"
  ]
}
""";
```

Sequências de escape de cadeia de caracteres

Expandir a tabela

Sequência de escape	Nome do caractere	Codificação Unicode
`\'`	Aspas simples	0x0027
`\"`	Aspas duplas	0x0022
`\\`	Barra invertida	0x005C
`\0`	Zero	0x0000
`\a`	Alerta	0x0007
`\b`	Backspace	0x0008
`\e`	Escape	0x001B
`\f`	Avanço de formulário	0x000C
`\n`	Nova linha	0x000A
`\r`	Retorno de carro	0x000D
`\t`	Guia horizontal	0x0009
`\v`	Guia vertical	0x000B
`\u`	Sequência de escape Unicode (UTF-16)	`\uHHHH` (intervalo: 0000 – FFFF; exemplo: `\u00E7` = "ç")
`\U`	Sequência de escape Unicode (UTF-32)	`\U00HHHHHH` (intervalo: 000000 - 10FFFF; exemplo: `\U0001F47D` = "👽")
`\x`	Sequência de escape Unicode semelhante a "\u", exceto com comprimento variável	`\xH[H][H][H]` (intervalo: 0 – FFFF; exemplo: `\x00E7` ou `\x0E7` ou `\xE7` = "ç")

⚠ **Aviso**

Ao usar a sequência de escape `\x` e especificar menos de 4 dígitos hexadecimais, se os caracteres que seguem imediatamente a sequência de escape forem dígitos hexadecimais válidos (ou seja, 0-9, A-F e a-f), eles serão interpretados como sendo parte da sequência de escape. Por exemplo, `\xA1` produz "\", que é o ponto de código U+00A1. No entanto, se o próximo caractere for "A" ou "a", a sequência de escape será interpretada como sendo `\xA1A` e produzirá "ঢ", que é o ponto de código U+0A1A. Nesses casos, especificar todos os quatro dígitos hexadecimais (por exemplo, `\x00A1`) impede qualquer possível interpretação incorreta.

ⓘ **Observação**

No tempo de compilação, as cadeias de caracteres brutas e verbatim são convertidas em comuns com todas as mesmas sequências de escape. Portanto, se você exibir uma cadeia de caracteres literal ou bruta na janela de inspeção do depurador, você verá os caracteres de escape que foram adicionados pelo compilador, não a versão literal ou bruta do seu código-fonte. Por exemplo, a cadeia de caracteres textual `@"C:\files.txt"` será exibida na janela de inspeção como `"C:\\files.txt"`.

Formatar cadeias de caracteres

Uma cadeia de caracteres de formato é uma cadeia de caracteres cujo conteúdo é determinado dinamicamente em tempo de execução. Cadeias de caracteres de formato são criadas incorporando *expressões interpoladas* ou espaços reservados dentro de chaves dentro em uma cadeia de caracteres. Tudo dentro das chaves (`{...}`) é resolvido para um valor e uma saída como uma cadeia de caracteres formatada no runtime. Há dois métodos para criar cadeias de caracteres de formato: interpolação de cadeia de caracteres e formatação composta.

Interpolação de cadeia de caracteres

Você declara *cadeias de caracteres interpoladas* com o caractere especial `$`. Uma cadeia de caracteres interpolada inclui expressões interpoladas em chaves. Se você não estiver familiarizado com a interpolação de cadeia de caracteres, consulte a interpolação de cadeia de caracteres – tutorial interativo em C# para obter uma visão geral rápida.

Use a interpolação de cadeia de caracteres para melhorar a legibilidade e a manutenção do código. A interpolação de strings obtém os mesmos resultados que o método `String.Format`, mas melhora a facilidade de uso e a clareza no código.

C#

```
var jh = (firstName: "Jupiter", lastName: "Hammon", born: 1711, published:
1761);
Console.WriteLine($"{jh.firstName} {jh.lastName} was an African American
poet born in {jh.born}.");
Console.WriteLine($"He was first published in {jh.published} at the age of
{jh.published - jh.born}.");
Console.WriteLine($"He'd be over {Math.Round((2018d - jh.born) / 100d) *
100d} years old today.");

// Output:
// Jupiter Hammon was an African American poet born in 1711.
// He was first published in 1761 at the age of 50.
// He'd be over 300 years old today.
```

Você pode usar a interpolação de cadeia de caracteres para inicializar uma cadeia de caracteres constante quando todas as expressões usadas para espaços reservados também são cadeias de caracteres constantes.

A partir do C# 11, você pode combinar *literais de cadeia de caracteres bruta* com interpolações de cadeia de caracteres. Você inicia e termina a cadeia de caracteres de formato com três ou mais aspas duplas sucessivas. Se a cadeia de caracteres de saída deve conter o caractere `{` ou `}`, você poderá usar caracteres de `$` extras para especificar quantos caracteres `{` e `}` iniciam e terminam uma interpolação. Qualquer sequência com menos de `{` ou `}` caracteres é incluída na saída. O exemplo a seguir mostra como você pode usar esse recurso para mostrar a distância de um ponto em relação à origem e colocar o ponto entre chaves.

C#

```
int X = 2;
int Y = 3;

var pointMessage = $$"""The point {{{X}}, {{Y}}} is {{Math.Sqrt(X * X + Y *
Y)}} from the origin.""";

Console.WriteLine(pointMessage);
// Output:
// The point {2, 3} is 3.605551275463989 from the origin.
```

Interpolação da cadeia de caracteres textual

Por exemplo, o C# também permite a interpolação de strings verbatim em várias linhas, usando a sintaxe `$@` ou `@$`.

Para interpretar sequências de escape literalmente, use um literal de cadeia de caracteres textual. Uma cadeia de caracteres verbatim interpolada começa com o caractere `$` seguido pelo caractere `@`. Você pode usar os tokens `$` e `@` em qualquer ordem: tanto `$@"..."` quanto `@$"..."` são strings verbatim interpoladas válidas.

C#

```
var jh = (firstName: "Jupiter", lastName: "Hammon", born: 1711, published:
1761);
Console.WriteLine($@"{jh.firstName} {jh.lastName}
    was an African American poet born in {jh.born}.");
Console.WriteLine(@$"He was first published in {jh.published}
at the age of {jh.published - jh.born}.");

// Output:
// Jupiter Hammon
//     was an African American poet born in 1711.
// He was first published in 1761
// at the age of 50.
```

Formatação composta

O String.Format utiliza os espaços reservados entre chaves para criar uma cadeia de caracteres de formato. Este exemplo resulta em saída semelhante ao método de interpolação de cadeia de caracteres usado no exemplo anterior.

C#

```
var pw = (firstName: "Phillis", lastName: "Wheatley", born: 1753, published:
1773);
Console.WriteLine("{0} {1} was an African American poet born in {2}.",
pw.firstName, pw.lastName, pw.born);
Console.WriteLine("She was first published in {0} at the age of {1}.",
pw.published, pw.published - pw.born);
Console.WriteLine("She'd be over {0} years old today.", Math.Round((2018d -
pw.born) / 100d) * 100d);

// Output:
// Phillis Wheatley was an African American poet born in 1753.
// She was first published in 1773 at the age of 20.
// She'd be over 300 years old today.
```

Para obter mais informações sobre a formatação de tipos .NET, consulte Tipos de Formatação no .NET.

Subcadeias de caracteres

Uma subcadeia de caracteres é qualquer sequência de caracteres contida em uma cadeia de caracteres. Use o método Substring para criar uma nova cadeia de caracteres de uma parte da cadeia de caracteres original. Você pode pesquisar uma ou mais ocorrências de uma subcadeia de caracteres usando o método IndexOf. Use o método Replace para substituir todas as ocorrências de uma subcadeia de caracteres especificada por uma nova cadeia de caracteres. Assim como o método Substring, Replace realmente retorna uma nova cadeia de caracteres e não modifica a cadeia de caracteres original. Para obter mais informações, consulte Como pesquisar cadeias de caracteres e Como modificar o conteúdo da cadeia de caracteres.

C#

```
string s3 = "Visual C# Express";
System.Console.WriteLine(s3.Substring(7, 2));
// Output: "C#"

System.Console.WriteLine(s3.Replace("C#", "Basic"));
// Output: "Visual Basic Express"

// Index values are zero-based
int index = s3.IndexOf("C");
// index = 7
```

Acesso a caracteres individuais

Você pode usar a notação de matriz com um valor de índice para adquirir acesso somente leitura a caracteres individuais, como no seguinte exemplo:

C#

```
string s5 = "Printing backwards";

for (int i = 0; i < s5.Length; i++)
{
    System.Console.Write(s5[s5.Length - i - 1]);
}
// Output: "sdrawkcab gnitnirP"
```

Caso os métodos String não forneçam a funcionalidade que você precisa para modificar caracteres individuais em uma cadeia de caracteres, você pode usar um objeto StringBuilder para modificar os caracteres individuais diretamente e, em seguida, criar uma nova cadeia de caracteres para armazenar os resultados usando os métodos

StringBuilder. No exemplo a seguir, suponha que você deve modificar a cadeia de caracteres original de uma maneira específica e armazenar os resultados para uso futuro:

C#

```
string question = "hOW DOES mICROSOFT wORD DEAL WITH THE cAPS lOCK KEY?";
System.Text.StringBuilder sb = new System.Text.StringBuilder(question);

for (int j = 0; j < sb.Length; j++)
{
    if (System.Char.IsLower(sb[j]) == true)
        sb[j] = System.Char.ToUpper(sb[j]);
    else if (System.Char.IsUpper(sb[j]) == true)
        sb[j] = System.Char.ToLower(sb[j]);
}
// Store the new string.
string corrected = sb.ToString();
System.Console.WriteLine(corrected);
// Output: How does Microsoft Word deal with the Caps Lock key?
```

Cadeias de caracteres nulas e cadeias de caracteres vazias

Uma cadeia de caracteres vazia é uma instância de um objeto System.String que contém zero caracteres. Cadeias de caracteres vazias são usadas frequentemente em vários cenários de programação para representar um campo de texto em branco. Você pode chamar métodos em strings vazias porque elas são objetos System.String válidos. Cadeias de caracteres vazias são inicializadas da seguinte maneira:

C#

```
string s = String.Empty;
```

Por outro lado, uma cadeia de caracteres nula não se refere a uma instância de um objeto System.String e qualquer tentativa de chamar um método em uma cadeia de caracteres nula causa um NullReferenceException. No entanto, você pode usar cadeias de caracteres nulas em operações de concatenação e comparação com outras cadeias de caracteres. Os exemplos a seguir ilustram alguns casos em que uma referência a uma cadeia de caracteres nula faz e não faz com que uma exceção seja gerada:

C#

```
string str = "hello";
string? nullStr = null;
string emptyStr = String.Empty;

string tempStr = str + nullStr;
// Output of the following line: hello
Console.WriteLine(tempStr);

bool b = (emptyStr == nullStr);
// Output of the following line: False
Console.WriteLine(b);

// The following line creates a new empty string.
string newStr = emptyStr + nullStr;

// Null strings and empty strings behave differently. The following
// two lines display 0.
Console.WriteLine(emptyStr.Length);
Console.WriteLine(newStr.Length);
// The following line raises a NullReferenceException.
//Console.WriteLine(nullStr.Length);

// The null character can be displayed and counted, like other chars.
string s1 = "\x0" + "abc";
string s2 = "abc" + "\x0";
// Output of the following line: * abc*
Console.WriteLine("*" + s1 + "*");
// Output of the following line: *abc *
Console.WriteLine("*" + s2 + "*");
// Output of the following line: 4
Console.WriteLine(s2.Length);
```

Usando StringBuilder para criação rápida de cadeia de caracteres

As operações de cadeia de caracteres no .NET são altamente otimizadas e, na maioria dos casos, não afetam significativamente o desempenho. No entanto, em alguns cenários, como loops rígidos que são executados centenas ou milhares de vezes, as operações de cadeia de caracteres podem afetar o desempenho. A classe StringBuilder cria um buffer de cadeia de caracteres que oferece melhor desempenho se o programa executar muitas manipulações de cadeia de caracteres. A string StringBuilder também permite que você reatribua caracteres individuais, algo que o tipo de dados de string padrão não suporta. Esse código, por exemplo, altera o conteúdo de uma cadeia de caracteres sem criar uma nova cadeia de caracteres:

```
C#
```

```
System.Text.StringBuilder sb = new System.Text.StringBuilder("Rat: the ideal
pet");
sb[0] = 'C';
System.Console.WriteLine(sb.ToString());
//Outputs Cat: the ideal pet
```

Neste exemplo, um objeto StringBuilder é usado para criar uma cadeia de caracteres de um conjunto de tipos numéricos:

C#

```
var sb = new StringBuilder();

// Create a string composed of numbers 0 - 9
for (int i = 0; i < 10; i++)
{
    sb.Append(i.ToString());
}
Console.WriteLine(sb);  // displays 0123456789

// Copy one character of the string (not possible with a System.String)
sb[0] = sb[9];

Console.WriteLine(sb);  // displays 9123456789
```

Cadeias de caracteres, métodos de extensão e LINQ

Como o tipo String implementa IEnumerable<T>, você pode usar os métodos de extensão definidos na classe Enumerable em cadeias de caracteres. Para evitar a desordem visual, esses métodos são excluídos do IntelliSense para o tipo de String, mas estão disponíveis no entanto. Você também pode usar expressões de consulta LINQ em cadeias de caracteres. Para obter mais informações, consulte LINQ e strings.

Artigos relacionados

- Como modificar o conteúdo da cadeia de caracteres: ilustra técnicas para transformar cadeias de caracteres e modificar o conteúdo das cadeias de caracteres.
- Como comparar cadeias de caracteres: mostra como executar comparações ordinais e específicas de cultura de cadeias de caracteres.
- Como concatenar várias cadeias de caracteres: demonstra várias maneiras de unir várias cadeias de caracteres em uma.

- Como analisar cadeias de caracteres usando String.Split: contém exemplos de código que ilustram como usar o método String.Split para analisar cadeias de caracteres.
- Como pesquisar cadeias de caracteres: explica como usar a pesquisa de texto ou padrões específicos em cadeias de caracteres.
- Como determinar se uma cadeia de caracteres representa um valor numérico: mostra como analisar com segurança uma cadeia de caracteres para ver se ela tem um valor numérico válido.
- Interpolação de cadeias de caracteres: descreve o recurso de interpolação de cadeia de caracteres que fornece uma sintaxe prática para cadeias de caracteres de formato.
- Operações Básicas de String: fornece links para artigos que usam métodos System.String e System.Text.StringBuilder para executar operações básicas de string.
- Análise de cadeias de caracteres: descreve como converter representações de cadeia de caracteres de tipos base do .NET em instâncias de tipos correspondentes.
- analisar cadeias de caracteres de data e hora no .NET: mostra como converter uma cadeia de caracteres como "24/01/2008" em um objeto System.DateTime.
- Comparando cadeias de caracteres: inclui informações sobre como comparar cadeias de caracteres e fornece exemplos no C# e no Visual Basic.
- Usando a classe StringBuilder: descreve como criar e modificar objetos de cadeia de caracteres dinâmicos usando a classe StringBuilder.
- LINQ e cadeias de caracteres: Fornece informações sobre como executar várias manipulações de strings usando consultas LINQ.

Como determinar se uma cadeia de caracteres representa um valor numérico (Guia de Programação em C#)

Artigo • 08/04/2025

Para determinar se uma cadeia de caracteres é uma representação válida de um tipo numérico especificado, use o método estático `TryParse` implementado por todos os tipos numéricos primitivos e também por tipos como DateTime e IPAddress. O exemplo a seguir mostra como determinar se "108" é um int válido.

C#

```
int i = 0;
string s = "108";
bool result = int.TryParse(s, out i); //i now = 108
```

Se a cadeia de caracteres contiver caracteres não numéricos ou o valor numérico for muito grande ou muito pequeno para o tipo específico especificado, `TryParse` retornará false e definirá o parâmetro out como zero. Caso contrário, ele retornará true e definirá o parâmetro out como o valor numérico da cadeia de caracteres.

> **Observação**
>
> Uma cadeia de caracteres pode conter apenas caracteres numéricos e ainda não ser válida para o tipo cujo `TryParse` método você usa. Por exemplo, "256" não é um valor válido, `byte` mas é válido para `int`. "98.6" não é um valor válido para `int`, mas é um valor válido para `decimal`.

Exemplo

Os exemplos a seguir mostram como usar `TryParse` com representações de cadeia de caracteres de `long`, `byte` e `decimal` valores.

C#

```
string numString = "1287543"; //"1287543.0" will return false for a long
long number1 = 0;
bool canConvert = long.TryParse(numString, out number1);
if (canConvert == true)
```

```
Console.WriteLine($"number1 now = {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 = {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 = {number3}");
else
Console.WriteLine("number3 is not a valid decimal");
```

Programação robusta

Tipos numéricos primitivos também implementam o `Parse` método estático, que gera uma exceção se a cadeia de caracteres não for um número válido. `TryParse` geralmente é mais eficiente porque apenas retorna false se o número não for válido.

Segurança do .NET

Sempre use os métodos `TryParse` ou `Parse` para validar a entrada do usuário de controles como caixas de texto e caixas de combinação.

Consulte também

- Como converter uma matriz de bytes em um int
- Como converter uma cadeia de caracteres em um número
- Como converter entre cadeias de caracteres hexadecimal e tipos numéricos
- Analisando cadeias de caracteres numéricas
- Tipos de Formatação

Indexadores

19/06/2025

Você define *indexadores* quando instâncias de uma classe ou struct podem ser indexadas como uma matriz ou outra coleção. O valor indexado pode ser definido ou recuperado sem especificar explicitamente um tipo ou membro de instância. Os indexadores se assemelham a propriedades, exceto pelo fato de que seus acessadores recebem parâmetros.

O exemplo a seguir define uma classe genérica com métodos de get e set simples para atribuir e recuperar valores.

C#

```
namespace Indexers;

public 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;
   }
}
```

O exemplo anterior mostra um indexador de leitura/gravação. Ele contém os acessadores `get` e `set`. Você pode definir indexadores somente leitura como um membro de expressão encorpado, conforme mostrado nos seguintes exemplos:

C#

```
namespace Indexers;

public class ReadOnlySampleCollection<T>(params IEnumerable<T> items)
{
   // Declare an array to store the data elements.
   private T[] arr = [.. items];

   public T this[int i] => arr[i];

}
```

A `get` palavra-chave não é usada; `=>` introduz o corpo da expressão.

Os indexadores habilitam propriedades *indexadas* : propriedades referenciadas usando um ou mais argumentos. Esses argumentos fornecem um índice em algumas coleções de valores.

- Os indexadores permitem que os objetos sejam indexados de forma semelhante às matrizes.
- Um `get` acessador retorna um valor. Um `set` acessador atribui um valor.
- A this palavra-chave define o indexador.
- A value palavra-chave é o argumento para o `set` acessador.
- Os indexadores não exigem um valor de índice inteiro; cabe a você definir o mecanismo de pesquisa específico.
- Os indexadores podem ser sobrecarregados.
- Os indexadores podem ter um ou mais parâmetros formais, por exemplo, ao acessar uma matriz bidimensional.
- Você pode declarar partial indexadores em partial tipos.

Você pode aplicar quase tudo o que aprendeu trabalhando com propriedades aos indexadores. A única exceção a essa regra é *as propriedades implementadas automaticamente*. Nem sempre o compilador pode gerar o armazenamento correto para um indexador. Você pode definir vários indexadores em um tipo, desde que as listas de argumentos para cada indexador sejam exclusivas.

Usos de indexadores

Você define *indexadores* em seu tipo quando sua API modela alguma coleção. Seu indexador não é obrigatório para mapear diretamente para os tipos de coleção que fazem parte da estrutura principal do .NET. Os indexadores permitem que você forneça a API que corresponde à abstração do tipo sem expor os detalhes internos de como os valores dessa abstração são armazenados ou computados.

Matrizes e vetores

Seu tipo pode modelar uma matriz ou um vetor. A vantagem de criar seu próprio indexador é que você pode definir o armazenamento dessa coleção para atender às suas necessidades. Imagine um cenário em que seu tipo modela dados históricos grandes demais para serem carregados de uma só vez na memória. Você precisa carregar e descarregar seções da coleção com base no uso. O exemplo a seguir modela esse comportamento. Ele relata quantos pontos de dados existem. Ele cria páginas para armazenar seções dos dados sob demanda. Ele remove páginas da memória para abrir espaço para páginas necessárias para solicitações mais recentes.

```
C#
```

```
namespace Indexers;

public record Measurements(double HiTemp, double LoTemp, double AirPressure);

public class DataSamples
{
    private class Page
    {
        private readonly List<Measurements> pageData = new ();
        private readonly int _startingIndex;
        private readonly int _length;

        public Page(int startingIndex, int length)
        {
            _startingIndex = startingIndex;
            _length = length;

            // 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 { get; private set; } = false;
        public DateTime LastAccess { get; set; } = DateTime.Now;
    }

    private readonly int _totalSize;
    private readonly List<Page> pagesInMemory = new ();
```

```
    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 newPage = new Page(startingIndex, 1000);
        addPageToCache(newPage);
        return newPage;
    }

    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);
    }
}
```

Você pode seguir esse idioma de design para modelar qualquer tipo de coleção em que haja bons motivos para não carregar todo o conjunto de dados em uma coleção na memória. Observe que a `Page` classe é uma classe aninhada privada que não faz parte da interface pública. Esses detalhes estão ocultos dos usuários dessa classe.

Dicionários

Outro cenário comum é quando você precisa modelar um dicionário ou um mapa. Este cenário ocorre quando seu tipo armazena valores com base em chave, possivelmente em chaves de texto. Este exemplo cria um dicionário que mapeia argumentos de linha de comando para expressões lambda que gerenciam essas opções. O exemplo a seguir mostra duas classes: uma `ArgsActions` classe que mapeia uma opção de linha de comando para um System.Action delegado, e uma `ArgsProcessor` que usa a `ArgsActions` para executar cada Action quando encontra essa opção.

C#

```
namespace Indexers;
public class ArgsProcessor
{
    private readonly ArgsActions _actions;

    public ArgsProcessor(ArgsActions actions)
    {
        _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();

    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;
    }
}
```

Neste exemplo, a coleção `ArgsAction` mapeia próximo à coleção subjacente. `get` determina se uma opção determinada está configurada. Em caso afirmativo, ele retorna a opção Action associada. Caso contrário, retorna um Action que não faz nada. O acessador público não inclui um acessador `set`. Em vez disso, o design está usando um método público para definir opções.

Mapas Multidimensionais

Você pode criar indexadores que usam vários argumentos. Além disso, esses argumentos não são restritos a serem do mesmo tipo.

O exemplo a seguir mostra uma classe que gera valores para um conjunto mandelbrot. Para obter mais informações sobre a matemática por trás do conjunto, leia este artigo. O indexador usa dois duplos para definir um ponto no plano X e Y. O `get` acessador calcula o número de iterações até que um ponto seja determinado como não no conjunto. Quando o número máximo de iterações é atingido, o ponto está no conjunto e o valor maxIterations da classe é retornado. (As imagens geradas pelo computador popularizadas para o conjunto Mandelbrot definem cores para o número de iterações necessárias para determinar que um ponto está fora do conjunto.)

C#

```
namespace Indexers;
public class Mandelbrot(int 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))
            {
                (x, y) = (x * x - y * y + x0, 2 * x * y + y0);
                iterations++;
            }
            return iterations;
        }
    }
}
```

O Conjunto mandelbrot define valores em cada coordenada (x,y) para valores de número real. Isso define um dicionário que pode conter um número infinito de valores. Portanto, não há armazenamento por trás desse conjunto. Em vez disso, essa classe calcula o valor para cada ponto quando o código chama o `get` acessador. Não há nenhum armazenamento subjacente usado.

Resumindo

Você cria indexadores sempre que tiver um elemento semelhante a uma propriedade em sua classe em que essa propriedade representa não um único valor, mas sim um conjunto de valores. Um ou mais argumentos identificam cada item individual. Esses argumentos podem identificar exclusivamente qual item no conjunto deve ser referenciado. Os indexadores estendem o conceito de propriedades, em que um membro é tratado como um item de dados externamente à classe, mas como um método internamente. Os indexadores permitem que os argumentos encontrem um único item em uma propriedade que representa um conjunto de itens.

Você pode acessar a pasta de exemplo para indexadores. Para obter instruções de download, consulte Exemplos e Tutoriais.

Especificação da linguagem C#

Para obter mais informações, consulte Indexadores na Especificação da Linguagem C#. A especificação de idioma é a fonte definitiva para a sintaxe e o uso de C#.

Usando indexadores (Guia de Programação em C#)

Artigo • 20/08/2024

Os indexadores são uma conveniência sintática que permite criar uma classe, um struct ou uma interface que os aplicativos clientes podem acessar como uma matriz. O compilador gera uma propriedade `Item` (ou uma propriedade de nome alternativo, se IndexerNameAttribute estiver presente) e os métodos acessadores apropriados. Os indexadores são implementados em tipos cuja principal finalidade é encapsular uma coleção ou matriz interna. Por exemplo, suponha que você tenha uma classe `TempRecord` que representa a temperatura em Fahrenheit, conforme registrada em 10 momentos diferentes durante um período de 24 horas. A classe contém uma matriz `temps` do tipo `float[]` para armazenar os valores de temperatura. Ao implementar um indexador nessa classe, os clientes podem acessar as temperaturas em uma instância `TempRecord` como `float temp = tempRecord[4]`, e não como `float temp = tempRecord.temps[4]`. A notação do indexador não apenas simplifica a sintaxe para aplicativos clientes, mas também torna a classe e a finalidade dela mais intuitivas para que os outros desenvolvedores entendam.

Para declarar um indexador em uma classe ou struct, use a palavra-chave this, como mostra o seguinte exemplo:

C#

```
// Indexer declaration
public int this[int index]
{
    // get and set accessors
}
```

> **Importante**
>
> Declarar um indexador vai gerar automaticamente uma propriedade chamada `Item` no objeto. A propriedade `Item` não pode ser acessada diretamente por meio da expressão de acesso a membro da instância. Além disso, se você adicionar a sua propriedade `Item` a um objeto com indexador, será gerado um erro do compilador CS0102. Para evitar esse erro, use a opção de IndexerNameAttribute renomear o indexador, conforme detalhado mais para frente neste artigo.

Comentários

O tipo de um indexador e o tipo dos seus parâmetros devem ser pelo menos tão acessíveis quanto o próprio indexador. Para obter mais informações sobre níveis de acessibilidade, consulte Modificadores de acesso.

Para obter mais informações sobre como usar indexadores com uma interface, consulte Indexadores de Interface.

A assinatura de um indexador consiste do número e dos tipos de seus parâmetros formais. Ela não inclui o tipo de indexador nem os nomes dos parâmetros formais. Se você declarar mais de um indexador na mesma classe, eles terão diferentes assinaturas.

Um indexador não é classificado como uma variável; portanto, um valor de indexador não pode ser passado por referência (como um parâmetro ref ou out), a menos que seu valor seja uma referência (ou seja, ele retorna por referência).

Para fornecer o indexador com um nome que outras linguagens possam usar, use System.Runtime.CompilerServices.IndexerNameAttribute, como mostra o seguinte exemplo:

```
C#

// Indexer declaration
[System.Runtime.CompilerServices.IndexerName("TheItem")]
public int this[int index]
{
    // get and set accessors
}
```

Esse indexador tem o nome `TheItem`, pois ele é substituído pelo atributo de nome do indexador. Por padrão, o nome do indexador é `Item`.

Exemplo 1

O exemplo a seguir mostra como declarar um campo de matriz privada, `temps` e um indexador. O indexador permite acesso direto à instância `tempRecord[i]`. A alternativa ao uso do indexador é declarar a matriz como um membro público e acessar seus membros, `tempRecord.temps[i]`, diretamente.

```
C#

public class TempRecord
{
```

```
    // Array of temperature values
    float[] temps =
    [
        56.2F, 56.7F, 56.5F, 56.9F, 58.8F,
        61.3F, 65.9F, 62.1F, 59.2F, 57.5F
    ];

    // To enable client code to validate input
    // when accessing your indexer.
    public int Length => temps.Length;

    // Indexer declaration.
    // If index is out of range, the temps array will throw the exception.
    public float this[int index]
    {
        get => temps[index];
        set => temps[index] = value;
    }
}
```

Observe que, quando o acesso de um indexador é avaliado, por exemplo, em uma instrução `Console.Write`, o acessador get é invocado. Portanto, se não existir nenhum acessador `get`, ocorrerá um erro em tempo de compilação.

C#

```
var tempRecord = new TempRecord();

// Use the indexer's set accessor
tempRecord[3] = 58.3F;
tempRecord[5] = 60.1F;

// Use the indexer's get accessor
for (int i = 0; i < 10; i++)
{
    Console.WriteLine($"Element #{i} = {tempRecord[i]}");
}
```

Indexando usando outros valores

O C# não limita o tipo de parâmetro do indexador ao inteiro. Por exemplo, pode ser útil usar uma cadeia de caracteres com um indexador. Esse indexador pode ser implementado pesquisando a cadeia de caracteres na coleção e retornando o valor adequado. Como os acessadores podem ser sobrecarregados, as versões do inteiro e da cadeia de caracteres podem coexistir.

Exemplo 2

O exemplo a seguir declara uma classe que armazena os dias da semana. Um acessador `get` aceita uma cadeia de caracteres, o nome de um dia e retorna o inteiro correspondente. Por exemplo, "Sunday" retorna 0, "Monday" retorna 1 e assim por diante.

C#

```
// Using a string as an indexer value
class DayCollection
{
    string[] days = ["Sun", "Mon", "Tues", "Wed", "Thurs", "Fri", "Sat"];

    // Indexer with only a get accessor with the expression-bodied
definition:
    public int this[string day] => FindDayIndex(day);

    private int FindDayIndex(string day)
    {
        for (int j = 0; j < days.Length; j++)
        {
            if (days[j] == day)
            {
                return j;
            }
        }

        throw new ArgumentOutOfRangeException(
            nameof(day),
            $"Day {day} is not supported.\nDay input must be in the form
\"Sun\", \"Mon\", etc");
    }
}
```

Exemplo de consumo 2

C#

```
var week = new DayCollection();
Console.WriteLine(week["Fri"]);

try
{
    Console.WriteLine(week["Made-up day"]);
}
catch (ArgumentOutOfRangeException e)
{
```

```
    Console.WriteLine($"Not supported input: {e.Message}");
}
```

Exemplo 3

O exemplo a seguir declara uma classe que armazena os dias da semana usando a enumeração System.DayOfWeek. Um acessador `get` aceita `DayOfWeek`, o valor de um dia, e retorna o inteiro correspondente. Por exemplo, `DayOfWeek.Sunday` retorna 0, `DayOfWeek.Monday` retorna 1 e assim por diante.

C#

```
using Day = System.DayOfWeek;

class DayOfWeekCollection
{
    Day[] days =
    [
        Day.Sunday, Day.Monday, Day.Tuesday, Day.Wednesday,
        Day.Thursday, Day.Friday, Day.Saturday
    ];

    // Indexer with only a get accessor with the expression-bodied
definition:
    public int this[Day day] => FindDayIndex(day);

    private int FindDayIndex(Day day)
    {
        for (int j = 0; j < days.Length; j++)
        {
            if (days[j] == day)
            {
                return j;
            }
        }
        throw new ArgumentOutOfRangeException(
            nameof(day),
            $"Day {day} is not supported.\nDay input must be a defined
System.DayOfWeek value.");
    }
}
```

Exemplo de consumo 3

C#

```
var week = new DayOfWeekCollection();
Console.WriteLine(week[DayOfWeek.Friday]);
```

```
try
{
    Console.WriteLine(week[(DayOfWeek)43]);
}
catch (ArgumentOutOfRangeException e)
{
    Console.WriteLine($"Not supported input: {e.Message}");
}
```

Programação robusta

Há duas maneiras principais nas quais a segurança e a confiabilidade de indexadores podem ser melhoradas:

- Certifique-se de incorporar algum tipo de estratégia de tratamento de erros para manipular a chance de passagem de código cliente em um valor de índice inválido. Anteriormente, no primeiro exemplo neste artigo, a classe TempRecord oferece uma propriedade Length que permite que o código cliente verifique a saída antes de passá-la para o indexador. Também é possível colocador o código de tratamento de erro dentro do próprio indexador. Certifique-se documentar para os usuários as exceções que você gera dentro de um acessador do indexador.

- Defina a acessibilidade dos acessadores get e set para que ela seja mais restritiva possível. Isso é importante para o acessador `set` em particular. Para obter mais informações, consulte Restringindo a acessibilidade aos acessadores.

Confira também

- Indexadores
- Propriedades

Indexadores em interfaces (Guia de Programação em C#)

Artigo • 20/08/2024

Os indexadores podem ser declarados em uma interface. Acessadores de indexadores de interface diferem dos acessadores de indexadores de classe das seguintes maneiras:

- Os acessadores de interface não usam modificadores.
- Um acessador de interface normalmente não tem corpo.

Portanto, a finalidade do acessador é indicar se o indexador é do tipo leitura/gravação, somente leitura ou somente gravação. Você pode fornecer uma implementação para um indexador definido em uma interface, mas isso é raro. Normalmente, os indexadores definem uma API para acessar campos de dados, que não podem ser definidos em uma interface.

Este é um exemplo de um acessador de indexador de interface:

C#

```
public interface ISomeInterface
{
    //...

    // Indexer declaration:
    string this[int index]
    {
        get;
        set;
    }
}
```

A assinatura de um indexador deve ser diferente das assinaturas de todos os outros indexadores declarados na mesma interface.

Exemplo

O exemplo a seguir mostra como implementar indexadores de interface.

C#

```
// Indexer on an interface:
public interface IIndexInterface
{
```

```
    // Indexer declaration:
    int this[int index]
    {
        get;
        set;
    }
}

// Implementing the interface.
class IndexerClass : IIndexInterface
{
    private int[] arr = new int[100];
    public int this[int index]   // indexer declaration
    {
        // The arr object will throw IndexOutOfRange exception.
        get => arr[index];
        set => arr[index] = value;
    }
}
```

C#

```
IndexerClass test = new IndexerClass();
System.Random rand = System.Random.Shared;
// Call the indexer to initialize its elements.
for (int i = 0; i < 10; i++)
{
    test[i] = rand.Next();
}
for (int i = 0; i < 10; i++)
{
    System.Console.WriteLine($"Element #{i} = {test[i]}");
}

/* Sample output:
    Element #0 = 360877544
    Element #1 = 327058047
    Element #2 = 1913480832
    Element #3 = 1519039937
    Element #4 = 601472233
    Element #5 = 323352310
    Element #6 = 1422639981
    Element #7 = 1797892494
    Element #8 = 875761049
    Element #9 = 393083859
*/
```

No exemplo anterior, é possível usar a implementação de membro de interface explícita usando o nome totalmente qualificado do membro de interface. Por exemplo

C#

```
string IIndexInterface.this[int index]
{
}
```

No entanto, o nome totalmente qualificado só será necessário para evitar ambiguidade quando a classe estiver implementando mais de uma interface com a mesma assinatura do indexador. Por exemplo, se uma classe `Employee` estiver implementando dois interfaces, `ICitizen` e `IEmployee`, e as duas interfaces tiverem a mesma assinatura de indexador, a implementação de membro de interface explícita é necessária. Ou seja, a seguinte declaração de indexador:

```C#
string IEmployee.this[int index]
{
}
```

implementa o indexador na interface `IEmployee`, enquanto a seguinte declaração:

```C#
string ICitizen.this[int index]
{
}
```

implementa o indexador na interface `ICitizen`.

Confira também

- Indexadores
- Propriedades
- Interfaces

Comparação entre propriedades e indexadores (Guia de Programação em C#)

Artigo • 26/10/2024

Os indexadores são como propriedades. Com exceção das diferenças mostradas na tabela a seguir, todas as regras definidas para acessadores de propriedade também se aplicam a acessadores de indexador.

Expandir a tabela

Propriedade	Indexador
Permite que os métodos sejam chamados como se fossem membros de dados públicos.	Permite que elementos de uma coleção interna de um objeto sejam acessados usando uma notação de matriz no próprio objeto.
Acessado por meio de um nome simples.	Acessado por meio de um índice.
Pode ser estático ou um membro de instância.	Deve ser um membro da instância.
Um acessador get de uma propriedade não tem parâmetros.	Um acessador `get` de um indexador tem a mesma lista de parâmetro formal que o indexador.
Um acessador set de uma propriedade contém o parâmetro implícito `value`.	Um acessador `set` de um indexador tem a mesma lista de parâmetro formal que o indexador, bem como o mesmo parâmetro de valor.
Suporta sintaxe abreviada com propriedades implementadas automaticamente.	Dá suporte a membros aptos para expressão a fim de obter somente indexadores.

Confira também

- Indexadores
- Propriedades

Eventos (Guia de Programação em C#)

Artigo • 01/04/2025

Os eventos permitem que uma classe ou objeto notifique outras classes ou objetos quando algo de interesse ocorrer. A classe que envia (ou *gera*) o evento é chamada de *publicadora* e as classes que recebem (ou *manipulam*) o evento são chamadas de *assinantes*.

Em um aplicativo Web ou Windows Forms em C# típico, você assina eventos gerados por controles como botões e caixas de listagem. Você pode usar o IDE (ambiente de desenvolvimento integrado) do Visual C# para navegar pelos eventos que um controle publica e selecionar os que deseja manipular. O IDE fornece uma maneira fácil de adicionar automaticamente um método de manipulador de eventos vazio e o código para assinar o evento. Para obter mais informações, consulte Como inscrever-se em eventos e cancelar a inscrição.

Visão geral de eventos

Os eventos têm as seguintes propriedades:

- O editor determina quando um evento é gerado; os assinantes determinam qual ação é tomada em resposta ao evento.
- Um evento pode ter vários assinantes. Um assinante pode manipular vários eventos de vários publicadores.
- Eventos que não têm assinantes nunca são acionados.
- Os eventos normalmente são usados para sinalizar ações do usuário, como cliques de botão ou seleções de menu em interfaces gráficas do usuário.
- Quando um evento tem vários assinantes, os manipuladores de eventos são invocados de forma síncrona quando um evento é acionado. Para invocar eventos de forma assíncrona, consulte chamando métodos síncronos de forma assíncrona.
- Na biblioteca de classes do .NET, os eventos são baseados no delegado EventHandler e na classe base EventArgs.

Seções relacionadas

Para obter mais informações, consulte:

- Como assinar e cancelar a assinatura de eventos
- Como publicar eventos em conformidade com as diretrizes do .NET
- Como gerar eventos de classe base em classes derivadas

- Como implementar eventos de interface
- Como implementar acessadores de eventos personalizados

Especificação da linguagem C#

Para obter mais informações, veja Eventos na Especificação da linguagem C#. A especificação de idioma é a fonte definitiva para a sintaxe e o uso de C#.

Consulte também

- EventHandler
- Delegados
- Criando manipuladores de eventos no Windows Forms

Como realizar e cancelar a assinatura de eventos (Guia de Programação em C#)

Artigo • 10/03/2023

Você assina um evento publicado por outra classe quando quer escrever um código personalizado que é chamado quando esse evento é gerado. Por exemplo, você pode assinar o evento `click` de um botão para fazer com que seu aplicativo faça algo útil quando o usuário clicar no botão.

Para assinar eventos usando o IDE do Visual Studio

1. Se você não vir a janela **Propriedades**, no modo de exibição de **Design**, clique com o botão direito do mouse no formulário ou controle para o qual deseja criar um manipulador de eventos e selecione **Propriedades**.

2. Na parte superior da janela **Propriedades**, clique no ícone **Eventos**.

3. Clique duas vezes no evento que deseja criar, por exemplo, o evento `Load`.

   O Visual C# cria um método de manipulador de eventos vazio e adiciona-o ao código. Como alternativa, você pode adicionar o código manualmente no modo de exibição **Código**. Por exemplo, as linhas de código a seguir declaram um método de manipulador de eventos que será chamado quando a classe `Form` gerar o evento `Load`.

   C#

   ```
   private void Form1_Load(object sender, System.EventArgs e)
   {
       // Add your form load event handling code here.
   }
   ```

   A linha de código que é necessária para assinar o evento também é gerada automaticamente no método `InitializeComponent` no arquivo Form1.Designer.cs em seu projeto. Ele é semelhante a isto:

   C#

   ```
   this.Load += new System.EventHandler(this.Form1_Load);
   ```

Para assinar eventos de forma programática

1. Defina um método de manipulador de eventos cuja assinatura corresponda à assinatura do delegado do evento. Por exemplo, se o evento se basear no tipo de delegado EventHandler, o código a seguir representará o stub do método:

   ```C#
   void HandleCustomEvent(object sender, CustomEventArgs a)
   {
      // Do something useful here.
   }
   ```

2. Use o operador de atribuição de adição (`+=`) para anexar um manipulador de eventos ao evento. No exemplo a seguir, suponha que um objeto chamado `publisher` tem um evento chamado `RaiseCustomEvent`. Observe que a classe do assinante precisa de uma referência à classe do editor para assinar seus eventos.

   ```C#
   publisher.RaiseCustomEvent += HandleCustomEvent;
   ```

   Você também pode usar uma expressão lambda para especificar um manipulador de eventos:

   ```C#
   public Form1()
   {
       InitializeComponent();
       this.Click += (s,e) =>
           {
               MessageBox.Show(((MouseEventArgs)e).Location.ToString());
           };
   }
   ```

Para assinar eventos usando uma função anônima

Se não tiver que cancelar a assinatura de um evento posteriormente, você poderá usar o operador de atribuição de adição (`+=`) para anexar uma função anônima como um manipulador de eventos. No exemplo a seguir, suponha que um objeto chamado `publisher` tenha um evento chamado `RaiseCustomEvent` e que uma classe `CustomEventArgs` também tenha sido definida para conter algum tipo de informação de

evento específico. Observe que a classe do assinante precisa de uma referência a `publisher` para assinar seus eventos.

```
C#

publisher.RaiseCustomEvent += (object o, CustomEventArgs e) =>
{
  string s = o.ToString() + " " + e.ToString();
  Console.WriteLine(s);
};
```

Você não poderá cancelar facilmente a assinatura de um evento se tiver usado uma função anônima para assiná-lo. Para cancelar a assinatura nesse cenário, volte para o código em que você assina o evento, armazene a função anônima em uma variável de delegado e adicione o delegado ao evento. Recomendamos que você não use funções anônimas para assinar eventos se tiver que cancelar a assinatura do evento em algum momento posterior em seu código. Para obter mais informações sobre expressões lambda, consulte Expressão lambda.

Cancelando a assinatura

Para impedir que o manipulador de eventos seja invocado quando o evento for gerado, cancele a assinatura do evento. Para evitar perda de recursos, cancele a assinatura de eventos antes de descartar um objeto de assinante. Até que você cancele a assinatura de um evento, o delegado multicast subjacente ao evento no objeto de publicação terá uma referência ao delegado que encapsula o manipulador de eventos do assinante. Desde que o objeto de publicação contenha essa referência, a coleta de lixo não excluirá seu objeto de assinante.

Para cancelar a assinatura de um evento

- Use o operador de atribuição de subtração (`-=`) para cancelar a assinatura de um evento:

  ```
  C#

  publisher.RaiseCustomEvent -= HandleCustomEvent;
  ```

  Quando todos os assinantes tiverem cancelado a assinatura de um evento, a instância do evento na classe do publicador será definida como `null`.

Confira também

- Eventos
- event
- Como publicar eventos em conformidade com as diretrizes do .NET
- Operadores - e -=
- Operadores + e +=

Como acionar eventos de classe base em classes derivadas (Guia de Programação em C#)

Artigo • 10/05/2023

O exemplo simples a seguir mostra o modo padrão para declarar os eventos em uma classe base para que eles também possam ser gerados das classes derivadas. Esse padrão é amplamente usado em classes do Windows Forms nas bibliotecas de classes do .NET.

Quando você cria uma classe que pode ser usada como uma classe base para outras classes, deve considerar o fato de que os eventos são um tipo especial de delegado que pode ser invocado apenas de dentro da classe que os declarou. As classes derivadas não podem invocar diretamente eventos declarados dentro da classe base. Embora, às vezes, você possa desejar um evento que possa ser gerado apenas pela classe base, na maioria das vezes você deve habilitar a classe derivada para invocar os eventos de classe base. Para fazer isso, você pode criar um método de invocação protegido na classe base que encapsula o evento. Chamando ou substituindo esse método de invocação, as classes derivadas podem invocar o evento diretamente.

> ⓘ **Observação**
>
> Não declare eventos virtuais em uma classe base e substitua-os em uma classe derivada. O compilador C# não lida com eles corretamente e é imprevisível se um assinante do evento derivado realmente estará assinando o evento de classe base.

Exemplo

C#

```
namespace BaseClassEvents
{
    // Special EventArgs class to hold info about Shapes.
    public class ShapeEventArgs : EventArgs
    {
        public ShapeEventArgs(double area)
        {
            NewArea = area;
        }
```

```
        public double NewArea { get; }
    }

    // Base class event publisher
    public abstract class Shape
    {
        protected double _area;

        public double Area
        {
            get => _area;
            set => _area = value;
        }

        // The event. Note that by using the generic EventHandler<T> event
type
        // we do not need to declare a separate delegate type.
        public event EventHandler<ShapeEventArgs> ShapeChanged;

        public abstract void Draw();

        //The event-invoking method that derived classes can override.
        protected virtual void OnShapeChanged(ShapeEventArgs e)
        {
            // Safely raise the event for all subscribers
            ShapeChanged?.Invoke(this, e);
        }
    }

    public class Circle : Shape
    {
        private double _radius;

        public Circle(double radius)
        {
            _radius = radius;
            _area = 3.14 * _radius * _radius;
        }

        public void Update(double d)
        {
            _radius = d;
            _area = 3.14 * _radius * _radius;
            OnShapeChanged(new ShapeEventArgs(_area));
        }

        protected override void OnShapeChanged(ShapeEventArgs e)
        {
            // Do any circle-specific processing here.

            // Call the base class event invocation method.
            base.OnShapeChanged(e);
        }
```

```
        public override void Draw()
        {
            Console.WriteLine("Drawing a circle");
        }
    }

    public class Rectangle : Shape
    {
        private double _length;
        private double _width;

        public Rectangle(double length, double width)
        {
            _length = length;
            _width = width;
            _area = _length * _width;
        }

        public void Update(double length, double width)
        {
            _length = length;
            _width = width;
            _area = _length * _width;
            OnShapeChanged(new ShapeEventArgs(_area));
        }

        protected override void OnShapeChanged(ShapeEventArgs e)
        {
            // Do any rectangle-specific processing here.

            // Call the base class event invocation method.
            base.OnShapeChanged(e);
        }

        public override void Draw()
        {
            Console.WriteLine("Drawing a rectangle");
        }
    }

    // Represents the surface on which the shapes are drawn
    // Subscribes to shape events so that it knows
    // when to redraw a shape.
    public class ShapeContainer
    {
        private readonly List<Shape> _list;

        public ShapeContainer()
        {
            _list = new List<Shape>();
        }

        public void AddShape(Shape shape)
        {
            _list.Add(shape);
```

```
            // Subscribe to the base class event.
            shape.ShapeChanged += HandleShapeChanged;
        }

        // ...Other methods to draw, resize, etc.

        private void HandleShapeChanged(object sender, ShapeEventArgs e)
        {
            if (sender is Shape shape)
            {
                // Diagnostic message for demonstration purposes.
                Console.WriteLine($"Received event. Shape area is now
{e.NewArea}");

                // Redraw the shape here.
                shape.Draw();
            }
        }
    }

    class Test
    {
        static void Main()
        {
            //Create the event publishers and subscriber
            var circle = new Circle(54);
            var rectangle = new Rectangle(12, 9);
            var container = new ShapeContainer();

            // Add the shapes to the container.
            container.AddShape(circle);
            container.AddShape(rectangle);

            // Cause some events to be raised.
            circle.Update(57);
            rectangle.Update(7, 7);

            // Keep the console window open in debug mode.
            Console.WriteLine("Press any key to continue...");
            Console.ReadKey();
        }
    }
}
/* Output:
        Received event. Shape area is now 10201.86
        Drawing a circle
        Received event. Shape area is now 49
        Drawing a rectangle
 */
```

Confira também

- Guia de Programação em C#
- Eventos
- Representantes
- Modificadores de acesso
- Criando manipuladores de eventos no Windows Forms

Como implementar eventos de interface (Guia de Programação em C#)

Artigo • 13/03/2024

Um interface pode declarar uma evento. O exemplo a seguir mostra como implementar eventos de interface em uma classe. Basicamente, as regras são as mesmas aplicadas à implementação de qualquer método ou propriedade de interface.

Implementar eventos de interface em uma classe

Declare o evento na classe e, em seguida, invoque-o nas áreas apropriadas.

C#

```
namespace ImplementInterfaceEvents
{
    public interface IDrawingObject
    {
        event EventHandler ShapeChanged;
    }
    public class MyEventArgs : EventArgs
    {
        // class members
    }
    public class Shape : IDrawingObject
    {
        public event EventHandler ShapeChanged;
        void ChangeShape()
        {
            // Do something here before the event…

            OnShapeChanged(new MyEventArgs(/*arguments*/));

            // or do something here after the event.
        }
        protected virtual void OnShapeChanged(MyEventArgs e)
        {
            ShapeChanged?.Invoke(this, e);
        }
    }

}
```

Exemplo

O exemplo a seguir mostra como lidar com a situação menos comum, na qual a classe herda de duas ou mais interfaces e cada interface tem um evento com o mesmo nome. Nessa situação, é necessário fornecer uma implementação explícita da interface para pelo menos um dos eventos. Ao gravar uma implementação explícita da interface de um evento, também é necessário gravar os acessadores de evento `add` e `remove`. Normalmente, eles são fornecidos pelo compilador, mas nesse caso o compilador não pode fornecê-los.

Ao fornecer acessadores próprios, é possível especificar se os dois eventos são representados pelo mesmo evento na classe ou por eventos diferentes. Por exemplo, se os eventos forem gerados em horários diferentes, de acordo com as especificações da interface, será possível associar cada evento a uma implementação separada na classe. No exemplo a seguir, os assinantes determinam qual evento `OnDraw` receberão ao converter a referência de forma para um `IShape` ou um `IDrawingObject`.

C#

```
namespace WrapTwoInterfaceEvents
{
    using System;

    public interface IDrawingObject
    {
        // Raise this event before drawing
        // the object.
        event EventHandler OnDraw;
    }
    public interface IShape
    {
        // Raise this event after drawing
        // the shape.
        event EventHandler OnDraw;
    }

    // Base class event publisher inherits two
    // interfaces, each with an OnDraw event
    public class Shape : IDrawingObject, IShape
    {
        // Create an event for each interface event
        event EventHandler PreDrawEvent;
        event EventHandler PostDrawEvent;

        object objectLock = new Object();

        // Explicit interface implementation required.
        // Associate IDrawingObject's event with
        // PreDrawEvent
```

```
        #region IDrawingObjectOnDraw
        event EventHandler IDrawingObject.OnDraw
        {
            add
            {
                lock (objectLock)
                {
                    PreDrawEvent += value;
                }
            }
            remove
            {
                lock (objectLock)
                {
                    PreDrawEvent -= value;
                }
            }
        }
        #endregion
        // Explicit interface implementation required.
        // Associate IShape's event with
        // PostDrawEvent
        event EventHandler IShape.OnDraw
        {
            add
            {
                lock (objectLock)
                {
                    PostDrawEvent += value;
                }
            }
            remove
            {
                lock (objectLock)
                {
                    PostDrawEvent -= value;
                }
            }
        }

        // For the sake of simplicity this one method
        // implements both interfaces.
        public void Draw()
        {
            // Raise IDrawingObject's event before the object is drawn.
            PreDrawEvent?.Invoke(this, EventArgs.Empty);

            Console.WriteLine("Drawing a shape.");

            // Raise IShape's event after the object is drawn.
            PostDrawEvent?.Invoke(this, EventArgs.Empty);
        }
    }
    public class Subscriber1
    {
```

```
        // References the shape object as an IDrawingObject
        public Subscriber1(Shape shape)
        {
            IDrawingObject d = (IDrawingObject)shape;
            d.OnDraw += d_OnDraw;
        }

        void d_OnDraw(object sender, EventArgs e)
        {
            Console.WriteLine("Sub1 receives the IDrawingObject event.");
        }
    }
    // References the shape object as an IShape
    public class Subscriber2
    {
        public Subscriber2(Shape shape)
        {
            IShape d = (IShape)shape;
            d.OnDraw += d_OnDraw;
        }

        void d_OnDraw(object sender, EventArgs e)
        {
            Console.WriteLine("Sub2 receives the IShape event.");
        }
    }

    public class Program
    {
        static void Main(string[] args)
        {
            Shape shape = new Shape();
            Subscriber1 sub = new Subscriber1(shape);
            Subscriber2 sub2 = new Subscriber2(shape);
            shape.Draw();

            // Keep the console window open in debug mode.
            System.Console.WriteLine("Press any key to exit.");
            System.Console.ReadKey();
        }
    }
}
/* Output:
    Sub1 receives the IDrawingObject event.
    Drawing a shape.
    Sub2 receives the IShape event.
*/
```

Confira também

- Eventos
- Representantes

- Implementação de interface explícita
- Como acionar eventos de classe base em classes derivadas

Colaborar conosco no GitHub

A fonte deste conteúdo pode ser encontrada no GitHub, onde você também pode criar e revisar problemas e solicitações de pull. Para obter mais informações, confira o nosso guia para colaboradores.

Comentários do .NET

O .NET é um projeto código aberto. Selecione um link para fornecer comentários:

Abrir um problema de documentação

Fornecer comentários sobre o produto

Como implementar acessadores de eventos personalizados (Guia de Programação em C#)

Artigo • 10/03/2023

Um evento é um tipo especial de delegado multicast que só pode ser invocado de dentro da classe em que ele está declarado. O código cliente assina o evento ao fornecer uma referência a um método que deve ser invocado quando o evento for disparado. Esses métodos são adicionados à lista de invocação do delegado por meio de acessadores de evento, que se assemelham aos acessadores de propriedade, com a exceção de que os acessadores de eventos são nomeados `add` e `remove`. Na maioria dos casos, não é necessário fornecer acessadores de eventos personalizados. Quando nenhum acessador de evento personalizado for fornecido no código, o compilador o adicionará automaticamente. No entanto, em alguns casos será necessário fornecer um comportamento personalizado. Um caso desse tipo é mostrado no tópico Como implementar eventos de interface.

Exemplo

O exemplo a seguir mostra como implementar os acessadores de eventos personalizados adicionar e remover. Embora seja possível substituir qualquer código dentro dos acessadores, é recomendável que você bloqueie o evento antes de adicionar ou remover um novo método de manipulador de eventos.

C#

```
event EventHandler IDrawingObject.OnDraw
{
    add
    {
        lock (objectLock)
        {
            PreDrawEvent += value;
        }
    }
    remove
    {
        lock (objectLock)
        {
            PreDrawEvent -= value;
        }
```

```
    }
}
```

Confira também

- Eventos
- event

Parâmetros de tipo genérico (Guia de Programação em C#)

17/06/2025

Em uma definição de tipo genérico ou método, um parâmetro de tipo é um espaço reservado para um tipo específico que um cliente especifica ao criar uma instância do tipo genérico. Uma classe genérica, como `GenericList<T>` listada em Introdução aos Genéricos, não pode ser usada as-is porque não é realmente um tipo; é mais como um blueprint para um tipo. Para usar `GenericList<T>`, o código do cliente deve declarar e criar uma instância de um tipo construído especificando um argumento de tipo dentro dos colchetes angulares. O argumento de tipo para essa classe específica pode ser qualquer tipo reconhecido pelo compilador. Qualquer número de instâncias de tipo construído pode ser criado, cada uma usando um argumento de tipo diferente, da seguinte maneira:

C#

```
GenericList<float> list1 = new GenericList<float>();
GenericList<ExampleClass> list2 = new GenericList<ExampleClass>();
GenericList<ExampleStruct> list3 = new GenericList<ExampleStruct>();
```

Em cada uma dessas instâncias de `GenericList<T>`, todas as ocorrências de `T` na classe são substituídas em tempo de execução pelo argumento de tipo. Por meio dessa substituição, criamos três objetos separados de tipo seguro e eficiente usando uma única definição de classe. Para obter mais informações sobre como essa substituição é executada pelo CLR, consulte Genéricos no Runtime.

Você pode aprender as convenções de nomenclatura para parâmetros de tipo genérico no artigo sobre convenções de nomenclatura.

Consulte também

- System.Collections.Generic
- Genéricos
- Diferenças entre modelos C++ e genéricos em C#

Restrições a parâmetros de tipo (Guia de Programação em C#)

Artigo • 05/08/2024

Restrições informam o compilador sobre as funcionalidades que um argumento de tipo deve ter. Sem nenhuma restrição, o argumento de tipo poderia ser qualquer tipo. O compilador pode assumir somente os membros de System.Object, que é a classe base definitiva para qualquer tipo .NET. Para obter mais informações, consulte Por que usar restrições. Se o código do cliente usa um tipo que não satisfaz uma restrição, o compilador emite um erro. Restrições são especificadas usando a palavra-chave contextual `where`. A tabela a seguir lista os sete tipos de restrições:

Expandir a tabela

Constraint	Descrição
`where T : struct`	O argumento de tipo deve ser um tipo de valor não anulável, que inclui os tipos `record struct`. Para obter informações sobre tipos que permitem valor nulo, consulte Tipos que permitem valor nulo. Como todos os tipos de valor têm um construtor sem parâmetros acessível, declarado ou implícito, a restrição `struct` implica a restrição `new()` e não pode ser combinada com a restrição `new()`. Você não pode combinar a restrição `struct` com a restrição `unmanaged`.
`where T : class`	O argumento de tipo deve ser um tipo de referência. Essa restrição se aplica também a qualquer classe, interface, delegado ou tipo de matriz. Em um contexto anulável, `T` deve ser um tipo de referência não anulável.
`where T : class?`	O argumento de tipo deve ser um tipo de referência, anulável ou não anulável. Essa restrição se aplica também a qualquer classe, interface, delegado ou tipo de matriz, incluindo registros.
`where T : notnull`	O argumento de tipo deve ser um tipo não anulável. O argumento pode ser um tipo de referência não anulável ou um tipo de valor não anulável.
`where T : unmanaged`	O argumento de tipo deve ser um tipo não gerenciado não anulável. A restrição `unmanaged` implica a restrição `struct` e não pode ser combinada com as restrições `struct` ou `new()`.
`where T : new()`	O argumento de tipo deve ter um construtor público sem parâmetros. Quando usado em conjunto com outras restrições, a restrição `new()` deve ser a última a ser especificada. A restrição `new()` não pode ser combinada com as restrições restrições `struct` e `unmanaged`.
`where T :` *<nome da*	O argumento de tipo deve ser ou derivar da classe base especificada. Em um contexto anulável, `T` deve ser um tipo de referência não anulável derivado da

Constraint	Descrição
classe base>	classe base especificada.
`where T :` *<nome da classe base>?*	O argumento de tipo deve ser ou derivar da classe base especificada. Em um contexto anulável, `T` pode ser um tipo anulável ou não anulável derivado da classe base especificada.
`where T :` *<nome da interface>*	O argumento de tipo deve ser ou implementar a interface especificada. Várias restrições de interface podem ser especificadas. A interface de restrição também pode ser genérica. Em um contexto anulável, `T` deve ser um tipo não anulável que implementa a interface especificada.
`where T :` *<nome da interface>?*	O argumento de tipo deve ser ou implementar a interface especificada. Várias restrições de interface podem ser especificadas. A interface de restrição também pode ser genérica. Em um contexto anulável, `T` pode ser um tipo de referência anulável, um tipo de referência não anulável ou um tipo de valor. `T` não pode ser um tipo de valor anulável.
`where T : U`	O argumento de tipo fornecido para `T` deve ser ou derivar do argumento fornecido para `U`. Em um contexto anulável, se `U` for um tipo de referência não anulável, `T` deve ser um tipo de referência não anulável. Se `U` for um tipo de referência anulável, `T` pode ser anulável ou não anulável.
`where T : default`	Essa restrição resolve a ambiguidade quando você precisa especificar um parâmetro de tipo não treinado ao substituir um método ou fornecer uma implementação de interface explícita. A restrição `default` implica o método base sem a restrição `class` a `struct`. Para obter mais informações, consulte a proposta de especificação de restrição default.
`where T : allows ref struct`	Essa anti-restrição declara que o argumento de tipo para `T` pode ser um tipo de `ref struct`. O tipo ou método genérico deve obedecer às regras de segurança de ref para qualquer instância de `T` porque pode ser um `ref struct`.

Algumas restrições são mutuamente exclusivas e algumas restrições devem estar em uma ordem especificada:

- Você pode aplicar no máximo uma das restrições `struct`, `class`, `class?`, `notnull` e `unmanaged`. Se você fornecer qualquer uma dessas restrições, ela deverá ser a primeira restrição especificada para esse parâmetro de tipo.
- A restrição de classe base (`where T : Base` ou `where T : Base?`) não pode ser combinada com nenhuma das restrições `struct`, `class`, `class?`, `notnull` ou `unmanaged`.
- Você pode aplicar no máximo uma restrição de classe base, em qualquer um dos formulários. Se você quiser dar suporte ao tipo base anulável, use `Base?`.
- Você não pode nomear a forma não anulável e anulável de uma interface como uma restrição.

- A restrição `new()` não pode ser combinada à restrição `struct` ou `unmanaged`. Se você especificar a restrição `new()`, ela deverá ser a última restrição para esse parâmetro de tipo. As antirrestrições, se aplicáveis, podem seguir a restrição `new()`.
- A restrição `default` pode ser aplicada somente em implementações de interface explícitas ou de substituição. Ela não pode ser combinada com as restrições `struct` ou `class`.
- A antirrestrição `allows ref struct` não pode ser combinada à restrição `class` ou `class?`.
- A antirrestrição `allows ref struct` deve seguir todas as restrições para esse argumento de tipo.

Por que usar restrições

As restrições especificam os recursos e as expectativas de um parâmetro de tipo. Declarar essas restrições significa que você pode usar as operações e as chamadas de método do tipo de restrição. Você aplica restrições ao parâmetro de tipo quando sua classe ou método genérico usa qualquer operação nos membros genéricos além da atribuição simples, o que inclui chamar quaisquer métodos sem suporte por System.Object. Por exemplo, a restrição de classe base informa ao compilador que somente os objetos desse tipo ou derivados desse tipo podem substituir este argumento de tipo. Uma vez que o compilador tiver essa garantia, ele poderá permitir que métodos desse tipo sejam chamados na classe genérica. O exemplo de código a seguir demonstra a funcionalidade que pode ser adicionada à classe `GenericList<T>` (em Introdução aos Genéricos) ao aplicar uma restrição de classe base.

C#

```
public class Employee
{
    public Employee(string name, int id) => (Name, ID) = (name, id);
    public string Name { get; set; }
    public int ID { get; set; }
}

public class GenericList<T> where T : Employee
{
    private class Node
    {
        public Node(T t) => (Next, Data) = (null, t);

        public Node? Next { get; set; }
        public T Data { get; set; }
    }

    private Node? head;
```

```
    public void AddHead(T t)
    {
        Node n = new Node(t) { Next = head };
        head = n;
    }

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

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

    public T? FindFirstOccurrence(string s)
    {
        Node? current = head;
        T? t = null;

        while (current != null)
        {
            //The constraint enables access to the Name property.
            if (current.Data.Name == s)
            {
                t = current.Data;
                break;
            }
            else
            {
                current = current.Next;
            }
        }
        return t;
    }
}
```

A restrição permite que a classe genérica use a propriedade `Employee.Name`. A restrição especifica que todos os itens do tipo `T` são um objeto `Employee` ou um objeto que herda de `Employee`.

Várias restrições podem ser aplicadas ao mesmo parâmetro de tipo e as restrições em si podem ser tipos genéricos, da seguinte maneira:

```
C#

class EmployeeList<T> where T : notnull, Employee, IComparable<T>, new()
{
    // ...
```

```
    public void AddDefault()
    {
        T t = new T();
        // ...
    }
}
```

Ao aplicar a restrição `where T : class`, evite os operadores `==` e `!=` no parâmetro de tipo, pois esses operadores testam somente a identidade de referência e não a igualdade de valor. Esse comportamento ocorrerá mesmo se esses operadores forem sobrecarregados em um tipo usado como argumento. O código a seguir ilustra esse ponto; a saída é false, muito embora a classe String sobrecarregue o operador `==`.

C#

```
public static void OpEqualsTest<T>(T s, T t) where T : class
{
    System.Console.WriteLine(s == t);
}

private static void TestStringEquality()
{
    string s1 = "target";
    System.Text.StringBuilder sb = new System.Text.StringBuilder("target");
    string s2 = sb.ToString();
    OpEqualsTest<string>(s1, s2);
}
```

O compilador sabe apenas que `T` é um tipo de referência no tempo de compilação e deve usar os operadores padrão válidos para todos os tipos de referência. Caso seja necessário testar a igualdade de valor, aplique a restrição `where T : IEquatable<T>` ou `where T : IComparable<T>` e implemente a interface em qualquer classe usada para construir a classe genérica.

Restringindo vários parâmetros

É possível aplicar restrições a vários parâmetros e várias restrições a um único parâmetro, conforme mostrado no exemplo a seguir:

C#

```
class Base { }
class Test<T, U>
    where U : struct
```

```
    where T : Base, new()
{ }
```

Parâmetros de tipo não associado

Os parâmetros de tipo que não têm restrições, como o T na classe pública `SampleClass<T>{}`, são denominados "parâmetros de tipo não associado". Os parâmetros de tipo não associado têm as seguintes regras:

- Os operadores `!=` e `==` não podem ser usados, pois não há garantia de que o argumento de tipo concreto oferecerá suporte a eles.
- Eles podem ser convertidos para e de `System.Object` ou explicitamente convertidos para qualquer tipo de interface.
- Você pode compará-los com nulo. Se um parâmetro não associado for comparado a `null`, a comparação sempre retornará false se o argumento de tipo for um tipo de valor.

Parâmetros de tipo como restrições

O uso de um parâmetro de tipo genérico como uma restrição será útil quando uma função membro com parâmetro de tipo próprio tiver que restringir esse parâmetro para o parâmetro de tipo do tipo recipiente, conforme mostrado no exemplo a seguir:

C#

```
public class List<T>
{
    public void Add<U>(List<U> items) where U : T {/*...*/}
}
```

No exemplo anterior, `T` é uma restrição de tipo no contexto do método `Add` e um parâmetro de tipo não associado no contexto da classe `List`.

Parâmetros de tipo também podem ser usados como restrições em definições de classe genérica. O parâmetro de tipo deve ser declarado entre colchetes angulares junto com quaisquer outros parâmetros de tipo:

C#

```
//Type parameter V is used as a type constraint.
public class SampleClass<T, U, V> where T : V { }
```

A utilidade dos parâmetros de tipo como restrições com classes genéricas é limitada, pois o compilador não pode presumir nada sobre o parâmetro de tipo, exceto que ele deriva de `System.Object`. Use parâmetros de tipo como restrições em classes genéricas em cenários nos quais deseja impor uma relação de herança entre dois parâmetros de tipo.

restrição de `notnull`

Você pode usar a restrição `notnull` para especificar que o argumento de tipo deve ser um tipo de valor não anulável ou um tipo de referência não anulável. Ao contrário da maioria das outras restrições, se um argumento de tipo violar a restrição `notnull`, o compilador gerará um aviso em vez de um erro.

A restrição `notnull` só tem efeito quando usada em um contexto anulável. Se você adicionar a restrição `notnull` em um contexto alheio anulável, o compilador não gerará avisos ou erros para violações da restrição.

restrição de `class`

A restrição `class` em um contexto anulável especifica que o argumento de tipo deve ser um tipo de referência não anulável. Em um contexto anulável, quando um argumento de tipo é um tipo de referência anulável, o compilador gera um aviso.

restrição de `default`

A adição de tipos de referência anuláveis complica o uso de `T?` em um tipo ou método genérico. `T?` pode ser usado com a restrição `struct` ou `class`, mas uma deve estar presente. Quando a restrição `class` era usada, `T?` referia-se ao tipo de referência anulável para `T`. `T?` pode ser usado quando nenhuma das restrições é aplicada. Nesse caso, `T?` é interpretado como `T?` para tipos de valor e tipos de referência. No entanto, se `T` for uma instância de Nullable<T>, `T?` será o mesmo que `T`. Em outras palavras, ele não se torna `T??`.

Como `T?` agora pode ser usado sem a restrição `class` ou `struct`, as ambiguidades podem surgir em substituições ou implementações de interface explícitas. Em ambos os casos, a substituição não inclui as restrições, mas as herda da classe base. Quando a classe base não aplica a restrição `class` ou `struct`, as classes derivadas precisam especificar de alguma forma uma substituição aplicada ao método base sem qualquer

restrição. O método derivado aplica a restrição `default`. A restrição `default` não esclarece a restrição *nem* a restrição `class` nem `struct`.

Restrição não gerenciada

Você pode usar a restrição `unmanaged` para especificar que o parâmetro de tipo deve ser um tipo não gerenciado não anulável. A restrição `unmanaged` permite que você escreva rotinas reutilizáveis para trabalhar com tipos que podem ser manipulados como blocos de memória, conforme mostrado no exemplo a seguir:

C#

```
unsafe public static byte[] ToByteArray<T>(this T argument) where T :
unmanaged
{
    var size = sizeof(T);
    var result = new Byte[size];
    Byte* p = (byte*)&argument;
    for (var i = 0; i < size; i++)
        result[i] = *p++;
    return result;
}
```

O método anterior deve ser compilado em um contexto `unsafe` porque ele usa o operador `sizeof` em um tipo não conhecido como um tipo interno. Sem a restrição `unmanaged`, o operador `sizeof` não está disponível.

A restrição `unmanaged` implica a restrição `struct` e não pode ser combinada ela. Como a restrição `struct` implica a restrição `new()`, a restrição `unmanaged` não pode ser combinada com a restrição `new()` também.

Restrições de delegado

Você pode usar System.Delegate ou System.MulticastDelegate como uma restrição de classe base. O CLR sempre permitia essa restrição, mas a linguagem C# não a permite. A restrição `System.Delegate` permite que você escreva código que funcione com delegados de uma maneira fortemente tipada. O código a seguir define um método de extensão que combina dois delegados fornecidos que são do mesmo tipo:

C#

```
public static TDelegate? TypeSafeCombine<TDelegate>(this TDelegate source,
TDelegate target)
```

```
    where TDelegate : System.Delegate
    => Delegate.Combine(source, target) as TDelegate;
```

Você pode usar o método anterior para combinar delegados que são do mesmo tipo:

```C#
Action first = () => Console.WriteLine("this");
Action second = () => Console.WriteLine("that");

var combined = first.TypeSafeCombine(second);
combined!();

Func<bool> test = () => true;
// Combine signature ensures combined delegates must
// have the same type.
//var badCombined = first.TypeSafeCombine(test);
```

Se você remover a marca de comentário na última linha, ela não será compilada. Tanto `first` quanto `test` são tipos de representante, mas são tipos diferentes de representantes.

Restrições de enum

Você também pode especificar o tipo System.Enum como uma restrição de classe base. O CLR sempre permitia essa restrição, mas a linguagem C# não a permite. Genéricos usando `System.Enum` fornecem programação fortemente tipada para armazenar em cache os resultados do uso de métodos estáticos em `System.Enum`. O exemplo a seguir localiza todos os valores válidos para um tipo enum e, em seguida, cria um dicionário que mapeia esses valores para sua representação de cadeia de caracteres.

```C#
public static Dictionary<int, string> EnumNamedValues<T>() where T :
System.Enum
{
    var result = new Dictionary<int, string>();
    var values = Enum.GetValues(typeof(T));

    foreach (int item in values)
        result.Add(item, Enum.GetName(typeof(T), item)!);
    return result;
}
```

`Enum.GetValues` e `Enum.GetName` usam reflexão, que tem implicações de desempenho. Você pode chamar `EnumNamedValues` para criar uma coleção que é armazenada em cache e reutilizada, em vez de repetir as chamadas que exigem reflexão.

Você pode usá-lo conforme mostrado no exemplo a seguir para criar uma enum e compilar um dicionário de seus nomes e valores:

C#

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

C#

```
var map = EnumNamedValues<Rainbow>();

foreach (var pair in map)
    Console.WriteLine($"{pair.Key}:\t{pair.Value}");
```

Argumentos de tipo implementam interface declarada

Alguns cenários exigem que um argumento fornecido para um parâmetro de tipo implemente essa interface. Por exemplo:

C#

```
public interface IAdditionSubtraction<T> where T : IAdditionSubtraction<T>
{
    static abstract T operator +(T left, T right);
    static abstract T operator -(T left, T right);
}
```

Esse padrão permite que o compilador C# determine o tipo que contém os operadores sobrecarregados ou qualquer método `static virtual` ou `static abstract`. Ele fornece a sintaxe para que os operadores de adição e subtração possam ser definidos em um

tipo que contém. Sem essa restrição, os parâmetros e argumentos seriam necessários para serem declarados como a interface, em vez do parâmetro de tipo:

```C#
public interface IAdditionSubtraction<T> where T : IAdditionSubtraction<T>
{
    static abstract IAdditionSubtraction<T> operator +(
        IAdditionSubtraction<T> left,
        IAdditionSubtraction<T> right);

    static abstract IAdditionSubtraction<T> operator -(
        IAdditionSubtraction<T> left,
        IAdditionSubtraction<T> right);
}
```

A sintaxe anterior exigiria que os implementadores usassem a implementação de interface explícita para esses métodos. Fornecer a restrição extra permite que a interface defina os operadores em termos dos parâmetros de tipo. Os tipos que implementam a interface podem implementar implicitamente os métodos de interface.

Permite ref struct

A antirrestrição `allows ref struct` declara que o argumento de tipo correspondente para pode ser um tipo de ref struct. As instâncias desse parâmetro de tipo devem obedecer às seguintes regras:

- Não pode ser demarcado.
- Ele participa regras de segurança de ref.
- Não é possível usar instâncias em que um tipo de `ref struct` não é permitido, como campos `static`.
- As instâncias podem ser marcadas com o modificador `scoped`.

A cláusula `allows ref struct` não é herdada. No seguinte código:

```C#
class SomeClass<T, S>
    where T : allows ref struct
    where S : T
{
    // etc
}
```

O argumento para `S` não pode ser um `ref struct` porque `S` não tem a cláusula `allows ref struct`.

Um parâmetro de tipo que tem a cláusula `allows ref struct` não pode ser usado como um argumento de tipo, a menos que o parâmetro de tipo correspondente também tenha a cláusula `allows ref struct`. Essa regra é demonstrada no exemplo a seguir:

C#

```
public class Allow<T> where T : allows ref struct
{

}

public class Disallow<T>
{
}

public class Example<T> where T : allows ref struct
{
    private Allow<T> fieldOne; // Allowed. T is allowed to be a ref struct

    private Disallow<T> fieldTwo; // Error. T is not allowed to be a ref
struct
}
```

O exemplo anterior mostra que um argumento de tipo que pode ser um tipo de `ref struct` não pode ser substituído por um parâmetro de tipo que não pode ser um tipo de `ref struct`.

Confira também

- System.Collections.Generic
- Introdução aos genéricos
- Classes genéricas
- Restrição new

Colaborar conosco no GitHub

A fonte deste conteúdo pode ser encontrada no GitHub, onde você também pode criar e revisar problemas e solicitações

.NET

Comentários do .NET

O .NET é um projeto código aberto. Selecione um link para fornecer comentários:

de pull. Para obter mais informações, confira o nosso guia para colaboradores.

Abrir um problema de documentação

Fornecer comentários sobre o produto

Classes genéricas (Guia de Programação em C#)

Artigo • 10/05/2023

As classes genéricas encapsulam operações que não são específicas de um determinado tipo de dados. O uso mais comum das classes genéricas é com coleções, como listas vinculadas, tabelas de hash, pilhas, filas, árvores e assim por diante. As operações como adicionar e remover itens da coleção são realizadas basicamente da mesma maneira, independentemente do tipo de dados que estão sendo armazenados.

Na maioria dos cenários que exigem classes de coleção, a abordagem recomendada é usar as que são fornecidas na biblioteca de classes do .NET. Para obter mais informações sobre o uso dessas classes, consulte Coleções genéricas no .NET.

Em geral, você cria classes genéricas iniciando com uma classe concreta existente e alterando os tipos para parâmetros de tipo, um por vez, até alcançar o equilíbrio ideal de generalização e usabilidade. Ao criar suas próprias classes genéricas, observe as seguintes considerações importantes:

- Quais tipos generalizar em parâmetros de tipo.

  Como uma regra, quanto mais tipos você puder parametrizar, mais flexível e reutilizável seu código se tornará. No entanto, generalização em excesso poderá criar um código que seja difícil de ser lido ou entendido por outros desenvolvedores.

- Quais restrições, se houver, aplicar aos parâmetros de tipo (consulte Restrições a parâmetros de tipo).

  Uma boa regra é aplicar o máximo de restrições, de maneira que ainda seja possível manipular os tipos que você precisa manipular. Por exemplo, se você souber que a classe genérica é destinada a ser usada apenas com tipos de referência, aplique a restrição da classe. Isso impedirá o uso não intencional de sua classe com tipos de valor e permitirá que você use o operador `as` em `T` e verificar se há valores nulos.

- Se deve-se levar em consideração o comportamento genérico em subclasses e classes base.

  Como as classes genéricas podem servir como classes base, as mesmas considerações de design aplicam-se nesse caso, como com as classes não

genéricas. Consulte as regras sobre heranças de classes base genéricas mais adiante neste tópico.

- Se implementar uma ou mais interfaces genéricas.

  Por exemplo, se você estiver projetando uma classe que será usada para criar itens em uma coleção com base em classes genéricas, poderá ser necessário implementar uma interface como a IComparable<T>, em que `T` é o tipo de sua classe.

Para obter um exemplo de uma classe genérica simples, consulte Introdução aos genéricos.

As regras para parâmetros de tipo e restrições têm várias implicações para o comportamento de classes genéricas, especialmente em relação à acessibilidade de membro e herança. Antes de prosseguir, você deve compreender alguns termos. Para um `Node<T>,` que é de classe genérica, o código cliente pode fazer referência à classe especificando um argumento de tipo – para criar um tipo construído fechado (`Node<int>`); ou deixando o parâmetro de tipo não especificado – por exemplo, quando você especifica uma classe base genérica, para criar um tipo construído aberto (`Node<T>`). As classes genéricas podem herdar de classes base construídas concretas, fechadas ou abertas:

C#

```
class BaseNode { }
class BaseNodeGeneric<T> { }

// concrete type
class NodeConcrete<T> : BaseNode { }

//closed constructed type
class NodeClosed<T> : BaseNodeGeneric<int> { }

//open constructed type
class NodeOpen<T> : BaseNodeGeneric<T> { }
```

As classes não genéricas ou em outras palavras, classes concretas, podem herdar de classes base construídas fechadas, mas não de classes construídas abertas ou de parâmetros de tipo, porque não há maneiras de o código cliente fornecer o argumento de tipo necessário para instanciar a classe base em tempo de execução.

C#

```
//No error
class Node1 : BaseNodeGeneric<int> { }
```

```
//Generates an error
//class Node2 : BaseNodeGeneric<T> {}

//Generates an error
//class Node3 : T {}
```

As classes genéricas que herdam de tipos construídos abertos devem fornecer argumentos de tipo para qualquer parâmetro de tipo de classe base que não é compartilhado pela classe herdeira, conforme demonstrado no código a seguir:

C#

```
class BaseNodeMultiple<T, U> { }

//No error
class Node4<T> : BaseNodeMultiple<T, int> { }

//No error
class Node5<T, U> : BaseNodeMultiple<T, U> { }

//Generates an error
//class Node6<T> : BaseNodeMultiple<T, U> {}
```

As classes genéricas que herdam de tipos construídos abertos devem especificar restrições que são um superconjunto ou sugerem, as restrições no tipo base:

C#

```
class NodeItem<T> where T : System.IComparable<T>, new() { }
class SpecialNodeItem<T> : NodeItem<T> where T : System.IComparable<T>,
new() { }
```

Os tipos genéricos podem usar vários parâmetros de tipo e restrições, da seguinte maneira:

C#

```
class SuperKeyType<K, V, U>
    where U : System.IComparable<U>
    where V : new()
{ }
```

Tipos construídos abertos e construídos fechados podem ser usados como parâmetros de método:

C#

```
void Swap<T>(List<T> list1, List<T> list2)
{
    //code to swap items
}

void Swap(List<int> list1, List<int> list2)
{
    //code to swap items
}
```

Se uma classe genérica implementa uma interface, todas as instâncias dessa classe podem ser convertidas nessa interface.

As classes genéricas são invariáveis. Em outras palavras, se um parâmetro de entrada especifica um `List<BaseClass>`, você receberá um erro em tempo de compilação se tentar fornecer um `List<DerivedClass>`.

Confira também

- System.Collections.Generic
- Guia de Programação em C#
- Genéricos
- Salvar o estado de enumeradores
- Um enigma de herança, parte 1

Interfaces genéricas (Guia de Programação em C#)

19/06/2025

Geralmente, é útil definir interfaces para classes de coleção genéricas ou para as classes genéricas que representam itens na coleção. Para evitar operações de conversão boxe e unboxing em tipos de valor, é melhor usar interfaces genéricas, como IComparable<T>, em classes genéricas. A biblioteca de classes do .NET define várias interfaces genéricas para uso com as classes de coleção no System.Collections.Generic namespace. Para obter mais informações sobre essas interfaces, consulte interfaces genéricas.

Quando uma interface é especificada como uma restrição em um parâmetro de tipo, somente tipos que implementam a interface podem ser usados. O exemplo de código a seguir mostra uma `SortedList<T>` classe derivada da `GenericList<T>` classe. Para obter mais informações, consulte Introdução aos genéricos. `SortedList<T>` adiciona a restrição `where T : IComparable<T>`. Essa restrição permite que o método `BubbleSort` em `SortedList<T>` use o método genérico CompareTo nos elementos da lista. Neste exemplo, os elementos de lista são uma classe simples, `Person` que implementa `IComparable<Person>`.

C#

```
//Type parameter T in angle brackets.
public class GenericList<T> : System.Collections.Generic.IEnumerable<T>
{
    protected Node head;
    protected Node current = null;

    // Nested class is also generic on T
    protected class Node
    {
        public Node next;
        private T data;  //T as private member datatype

        public Node(T t)  //T used in non-generic constructor
        {
            next = null;
            data = t;
        }

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

        public T Data  //T as return type of property
```

```
        {
            get { return data; }
            set { data = value; }
        }
    }

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

    public void AddHead(T t)  //T as method parameter type
    {
        Node n = new Node(t);
        n.Next = head;
        head = n;
    }

    // Implementation of the iterator
    public System.Collections.Generic.IEnumerator<T> GetEnumerator()
    {
        Node current = head;
        while (current != null)
        {
            yield return current.Data;
            current = current.Next;
        }
    }

    // IEnumerable<T> inherits from IEnumerable, therefore this class
    // must implement both the generic and non-generic versions of
    // GetEnumerator. In most cases, the non-generic method can
    // simply call the generic method.
    System.Collections.IEnumerator System.Collections.IEnumerable.GetEnumerator()
    {
        return GetEnumerator();
    }
}

public class SortedList<T> : GenericList<T> where T : System.IComparable<T>
{
    // A simple, unoptimized sort algorithm that
    // orders list elements from lowest to highest:

    public void BubbleSort()
    {
        if (null == head || null == head.Next)
        {
            return;
        }
        bool swapped;

        do
        {
            Node previous = null;
```

```
            Node current = head;
            swapped = false;

            while (current.next != null)
            {
                //  Because we need to call this method, the SortedList
                //  class is constrained on IComparable<T>
                if (current.Data.CompareTo(current.next.Data) > 0)
                {
                    Node tmp = current.next;
                    current.next = current.next.next;
                    tmp.next = current;

                    if (previous == null)
                    {
                        head = tmp;
                    }
                    else
                    {
                        previous.next = tmp;
                    }
                    previous = tmp;
                    swapped = true;
                }
                else
                {
                    previous = current;
                    current = current.next;
                }
            }
        } while (swapped);
    }
}

// A simple class that implements IComparable<T> using itself as the
// type argument. This is a common design pattern in objects that
// are stored in generic lists.
public class Person : System.IComparable<Person>
{
    string name;
    int age;

    public Person(string s, int i)
    {
        name = s;
        age = i;
    }

    // This will cause list elements to be sorted on age values.
    public int CompareTo(Person p)
    {
        return age - p.age;
    }

    public override string ToString()
```

```
    {
        return name + ":" + age;
    }

    // Must implement Equals.
    public bool Equals(Person p)
    {
        return (this.age == p.age);
    }
}

public class Program
{
    public static void Main()
    {
        //Declare and instantiate a new generic SortedList class.
        //Person is the type argument.
        SortedList<Person> list = new SortedList<Person>();

        //Create name and age values to initialize Person objects.
        string[] names =
        [
            "Franscoise",
            "Bill",
            "Li",
            "Sandra",
            "Gunnar",
            "Alok",
            "Hiroyuki",
            "Maria",
            "Alessandro",
            "Raul"
        ];

        int[] ages = [45, 19, 28, 23, 18, 9, 108, 72, 30, 35];

        //Populate the list.
        for (int x = 0; x < 10; x++)
        {
            list.AddHead(new Person(names[x], ages[x]));
        }

        //Print out unsorted list.
        foreach (Person p in list)
        {
            System.Console.WriteLine(p.ToString());
        }
        System.Console.WriteLine("Done with unsorted list");

        //Sort the list.
        list.BubbleSort();

        //Print out sorted list.
        foreach (Person p in list)
        {
```

```
            System.Console.WriteLine(p.ToString());
        }
        System.Console.WriteLine("Done with sorted list");
    }
}
```

Várias interfaces podem ser especificadas como restrições em um único tipo, da seguinte maneira:

```C#
class Stack<T> where T : System.IComparable<T>, IEnumerable<T>
{
}
```

Uma interface pode definir mais de um parâmetro de tipo, da seguinte maneira:

```C#
interface IDictionary<K, V>
{
}
```

As regras de herança que se aplicam às classes também se aplicam às interfaces:

```C#
interface IMonth<T> { }

interface IJanuary : IMonth<int> { }  //No error
interface IFebruary<T> : IMonth<int> { }  //No error
interface IMarch<T> : IMonth<T> { }    //No error
                                       //interface IApril<T>  : IMonth<T, U> {}
//Error
```

Interfaces genéricas podem herdar de interfaces não genéricas se a interface genérica for covariante, o que significa que ela usa apenas seu parâmetro de tipo como um valor retornado. Na biblioteca de classes do .NET, IEnumerable<T> herda de IEnumerable porque IEnumerable<T> usa apenas `T` no valor retornado de GetEnumerator e no getter de propriedade Current.

Classes concretas podem implementar interfaces construídas fechadas, da seguinte maneira:

```C#
interface IBaseInterface<T> { }
```

```
class SampleClass : IBaseInterface<string> { }
```

Classes genéricas podem implementar interfaces genéricas ou interfaces construídas fechadas, desde que a lista de parâmetros de classe forneça todos os argumentos exigidos pela interface, da seguinte maneira:

```C#
interface IBaseInterface1<T> { }
interface IBaseInterface2<T, U> { }

class SampleClass1<T> : IBaseInterface1<T> { }          //No error
class SampleClass2<T> : IBaseInterface2<T, string> { }  //No error
```

As regras que controlam a sobrecarga de método são as mesmas para métodos dentro de classes genéricas, structs genéricos ou interfaces genéricas. Para obter mais informações, consulte Métodos Genéricos.

A partir do C# 11, as interfaces podem declarar `static abstract` ou `static virtual` membros. Interfaces que declaram membros `static abstract` ou `static virtual` são quase sempre interfaces genéricas. O compilador deve resolver chamadas para métodos `static virtual` e `static abstract` no tempo de compilação. Os métodos `static virtual` e `static abstract` declarados em interfaces não têm um mecanismo de expedição de runtime análogo a métodos `virtual` ou `abstract` declarados em classes. Em vez disso, o compilador usa informações de tipo disponíveis no tempo de compilação. Esses membros normalmente são declarados em interfaces genéricas. Além disso, a maioria das interfaces que declaram os métodos `static virtual` ou `static abstract` declaram que um dos parâmetros de tipo deve implementar a interface declarada. Em seguida, o compilador usa os argumentos de tipo fornecidos para resolver o tipo do membro declarado.

Consulte também

- Introdução aos genéricos
- interface
- Genéricos

Métodos genéricos (guia de programação em C#)

19/06/2025

Um método genérico é um método que é declarado com parâmetros de tipo, da seguinte maneira:

```C#
static void Swap<T>(ref T lhs, ref T rhs)
{
    T temp;
    temp = lhs;
    lhs = rhs;
    rhs = temp;
}
```

O exemplo de código a seguir mostra uma maneira de chamar o método usando `int` o argumento type:

```C#
public static void TestSwap()
{
    int a = 1;
    int b = 2;

    Swap<int>(ref a, ref b);
    System.Console.WriteLine(a + " " + b);
}
```

Você também pode omitir o argumento de tipo e o compilador o inferirá. Esta chamada para `Swap` é equivalente à chamada anterior:

```C#
Swap(ref a, ref b);
```

As mesmas regras para inferência de tipo se aplicam a métodos estáticos e métodos de instância. O compilador pode inferir os parâmetros de tipo com base nos argumentos de método que você passa; ele não pode inferir os parâmetros de tipo apenas de uma restrição ou valor retornado. Portanto, a inferência de tipo não funciona com métodos que não têm parâmetros. A inferência de tipo ocorre no tempo de compilação antes que o compilador tente resolver assinaturas de método sobrecarregadas. O compilador aplica a lógica de inferência de

tipo a todos os métodos genéricos que compartilham o mesmo nome. Na etapa de resolução de sobrecarga, o compilador inclui apenas os métodos genéricos nos quais a inferência de tipo foi bem-sucedida.

Em uma classe genérica, métodos não genéricos podem acessar os parâmetros de tipo de nível de classe, da seguinte maneira:

C#

```
class SampleClass<T>
{
    void Swap(ref T lhs, ref T rhs) { }
}
```

Se você definir um método genérico que usa os mesmos parâmetros de tipo que a classe de contenção, o compilador gerará um aviso CS0693 porque, dentro do escopo do método, o argumento fornecido para o interior `T` ocultará o argumento fornecido para o exterior `T`. Se você precisar da flexibilidade de chamar um método de classe genérico com argumentos de tipo diferentes daqueles fornecidos quando a classe foi instanciada, considere fornecer outro identificador para o parâmetro de tipo do método, conforme mostrado no `GenericList2<T>` exemplo a seguir.

C#

```
class GenericList<T>
{
    // CS0693.
    void SampleMethod<T>() { }
}

class GenericList2<T>
{
    // No warning.
    void SampleMethod<U>() { }
}
```

Use restrições para habilitar operações mais especializadas em parâmetros de tipo em métodos. Esta versão de `Swap<T>`, agora nomeada `SwapIfGreater<T>`, só pode ser usada com argumentos de tipo que implementam IComparable<T>.

C#

```
void SwapIfGreater<T>(ref T lhs, ref T rhs) where T : System.IComparable<T>
{
    T temp;
    if (lhs.CompareTo(rhs) > 0)
    {
```

```
        temp = lhs;
        lhs = rhs;
        rhs = temp;
    }
}
```

Métodos genéricos podem ser sobrecarregados em vários parâmetros de tipo. Por exemplo, todos os métodos a seguir podem estar localizados na mesma classe:

```C#
void DoWork() { }
void DoWork<T>() { }
void DoWork<T, U>() { }
```

Você também pode usar o parâmetro de tipo como o tipo de retorno de um método. O exemplo de código a seguir mostra um método que retorna uma matriz de tipo `T`:

```C#
T[] Swap<T>(T a, T b)
{
    return [b, a];
}
```

Especificação da linguagem C#

Para obter mais informações, consulte a Especificação da Linguagem C# .

Consulte também

- System.Collections.Generic
- Introdução aos genéricos
- Métodos

Genéricos e matrizes (Guia de Programação em C#)

19/06/2025

Matrizes unidimensionais que têm um limite inferior de zero são implementadas IList<T>automaticamente. Isso permite a criação de métodos genéricos que podem usar o mesmo código para iterar por meio de matrizes e outros tipos de coleção. Essa técnica é útil principalmente para ler dados em coleções. A IList<T> interface não pode ser usada para adicionar ou remover elementos de uma matriz. Uma exceção será gerada se você tentar chamar um IList<T> método como RemoveAt em uma matriz neste contexto.

O exemplo de código a seguir demonstra como um único método genérico que usa um IList<T> parâmetro de entrada pode iterar por meio de uma lista e uma matriz, nesse caso, uma matriz de inteiros.

C#

```
class Program
{
    static void Main()
    {
        int[] arr = [0, 1, 2, 3, 4];
        List<int> list = new List<int>();

        for (int x = 5; x < 10; x++)
        {
            list.Add(x);
        }

        ProcessItems<int>(arr);
        ProcessItems<int>(list);
    }

    static void ProcessItems<T>(IList<T> coll)
    {
        // IsReadOnly returns True for the array and False for the List.
        System.Console.WriteLine
            ("IsReadOnly returns {0} for this collection.",
            coll.IsReadOnly);

        // The following statement causes a run-time exception for the
        // array, but not for the List.
        //coll.RemoveAt(4);

        foreach (T item in coll)
        {
            System.Console.Write(item?.ToString() + " ");
        }
```

```
        System.Console.WriteLine();
    }
}
```

Consulte também

- System.Collections.Generic
- Genéricos
- Matrizes
- Genéricos

Delegados Genéricos (Guia de Programação em C#)

19/06/2025

Um delegado pode definir seus próprios parâmetros de tipo. O código que faz referência ao delegado genérico pode especificar o argumento de tipo para criar um tipo construído fechado, assim como ao instanciar uma classe genérica ou chamar um método genérico, conforme mostrado no exemplo a seguir:

C#

```
public delegate void Del<T>(T item);
public static void Notify(int i) { }

Del<int> m1 = new Del<int>(Notify);
```

O C# versão 2.0 tem um novo recurso chamado conversão de grupo de métodos, que se aplica a tipos de delegados concretos e genéricos e permite que você escreva a linha anterior com essa sintaxe simplificada:

C#

```
Del<int> m2 = Notify;
```

Os delegados definidos em uma classe genérica podem usar os parâmetros de tipo de classe genéricos da mesma forma que os métodos de classe.

C#

```
class Stack<T>
{
    public delegate void StackDelegate(T[] items);
}
```

O código que faz referência ao delegado deve especificar o argumento de tipo da classe que contém, da seguinte maneira:

C#

```
private static void DoWork(float[] items) { }

public static void TestStack()
{
    Stack<float> s = new Stack<float>();
```

```
    Stack<float>.StackDelegate d = DoWork;
}
```

Os delegados genéricos são especialmente úteis na definição de eventos com base no padrão de design comum, pois o argumento do remetente pode ser fortemente tipado e não precisa ser convertido de e para Object.

C#

```
delegate void StackEventHandler<T, U>(T sender, U eventArgs);

class Stack<T>
{
    public class StackEventArgs : System.EventArgs { }
    public event StackEventHandler<Stack<T>, StackEventArgs>? StackEvent;

    protected virtual void OnStackChanged(StackEventArgs a)
    {
        if (StackEvent is not null)
            StackEvent(this, a);
    }
}

class SampleClass
{
    public void HandleStackChange<T>(Stack<T> stack, Stack<T>.StackEventArgs args)
{ }
}

public static void Test()
{
    Stack<double> s = new Stack<double>();
    SampleClass o = new SampleClass();
    s.StackEvent += o.HandleStackChange;
}
```

Consulte também

- System.Collections.Generic
- Introdução aos genéricos
- Métodos genéricos
- Classes genéricas
- Interfaces genéricas
- Representantes
- Genéricos

Diferenças entre modelos C++ e genéricos em C# (Guia de Programação em C#)

17/06/2025

Os modelos C# Generics e C++ são recursos de linguagem que dão suporte a tipos parametrizados. No entanto, há muitas diferenças entre os dois. No nível da sintaxe, os genéricos C# são uma abordagem mais simples para tipos parametrizados sem a complexidade dos modelos C++. Além disso, o C# não tenta fornecer todas as funcionalidades que os modelos C++ fornecem. No nível de implementação, a principal diferença é que as substituições de tipo genérico C# são executadas em tempo de execução e as informações de tipo genérico são preservadas para objetos instanciados. Para obter mais informações, consulte Genéricos no Runtime.

Veja a seguir as principais diferenças entre os modelos C# Generics e C++:

- Os genéricos de C# não fornecem a mesma flexibilidade que os modelos C++. Por exemplo, não é possível chamar operadores aritméticos em uma classe genérica C#, embora seja possível chamar operadores definidos pelo usuário.
- O C# não permite parâmetros de modelo sem tipo, como `template C<int i> {}`.
- O C# não dá suporte à especialização explícita; ou seja, uma implementação personalizada de um modelo para um tipo específico.
- O C# não dá suporte à especialização parcial: uma implementação personalizada para um subconjunto dos argumentos de tipo.
- O C# não permite que o parâmetro de tipo seja usado como a classe base para o tipo genérico.
- O C# não permite que parâmetros de tipo tenham tipos padrão.
- No C#, um parâmetro de tipo genérico não pode ser genérico, embora os tipos construídos possam ser usados como genéricos. O C++ permite parâmetros de modelo.
- O C++ permite código que pode não ser válido para todos os parâmetros de tipo no modelo, que é então verificado para o tipo específico usado como o parâmetro de tipo. O C# requer que o código em uma classe seja escrito de forma que funcione com qualquer tipo que atenda às restrições. Por exemplo, no C++ é possível escrever uma função que usa os operadores aritméticos `+` e `-` objetos do parâmetro de tipo, o que produzirá um erro no momento da instanciação do modelo com um tipo que não dá suporte a esses

operadores. O C# não permite isso; os únicos constructos de idioma permitidos são aqueles que podem ser deduzidos das restrições.

Consulte também

- Introdução aos genéricos
- Modelos

Genéricos no runtime (guia de programação em C#)

20/06/2025

Quando um tipo ou método genérico é compilado em CIL (linguagem intermediária comum), ele contém metadados que o identificam como tendo parâmetros de tipo. Como o CIL para um tipo genérico é usado difere com base em se o parâmetro de tipo fornecido é um tipo de valor ou tipo de referência.

Quando um tipo genérico é construído pela primeira vez com um tipo de valor como parâmetro, o runtime cria um tipo genérico especializado com o parâmetro ou parâmetros fornecidos substituídos nos locais apropriados no CIL. Tipos genéricos especializados são criados uma vez para cada tipo de valor exclusivo que é usado como um parâmetro.

Por exemplo, caso o código do programa declare uma pilha construída de inteiros:

```
C#

Stack<int>? stack;
```

Neste ponto, o runtime gerará uma versão especializada da classe Stack<T> com o inteiro substituído corretamente, de acordo com seu parâmetro. Agora, sempre que o código do programa utilizar uma pilha de inteiros, o runtime reutilizará a classe especializada Stack<T> gerada. No exemplo a seguir, duas instâncias de uma pilha de inteiros são criadas e compartilham uma única instância do `Stack<int>` código:

```
C#

Stack<int> stackOne = new Stack<int>();
Stack<int> stackTwo = new Stack<int>();
```

No entanto, suponha que outra Stack<T> classe com um tipo de valor diferente, como um `long` ou uma estrutura definida pelo usuário como parâmetro, seja criada em outro ponto do código. Como resultado, o runtime gerará outra versão do tipo genérico e substituirá um `long` nos locais apropriados no CIL. As conversões não são mais necessárias porque cada classe genérica especializada contém nativamente o tipo de valor.

Os genéricos funcionam de forma um pouco diferente para tipos de referência. Na primeira vez que um tipo genérico é construído com qualquer tipo de referência, o runtime cria um tipo genérico especializado com referências de objeto substituídas pelos parâmetros no CIL. Em seguida, sempre que um tipo construído é instanciado com um tipo de referência como seu

parâmetro, independentemente do tipo que ele é, o runtime reutiliza a versão especializada criada anteriormente do tipo genérico. Isso é possível porque todas as referências têm o mesmo tamanho.

Por exemplo, suponha que você tenha dois tipos de referência, uma `Customer` classe e uma `Order` classe, e também suponha que você criou uma pilha de `Customer` tipos:

```
C#

class Customer { }
class Order { }
```

```
C#

Stack<Customer> customers;
```

Neste ponto, o runtime gera uma versão especializada da Stack<T> classe que armazena referências de objeto que serão preenchidas posteriormente em vez de armazenar dados. Suponha que a próxima linha de código crie uma pilha de outro tipo de referência, com o nome `Order`:

```
C#

Stack<Order> orders = new Stack<Order>();
```

Ao contrário dos tipos de valor, outra versão especializada da Stack<T> classe não é criada para o `Order` tipo. Em vez disso, uma instância da versão especializada da Stack<T> classe é criada e a `orders` variável é definida para referenciá-la. Imagine que uma linha de código foi encontrada para criar uma pilha de um tipo `Customer`:

```
C#

customers = new Stack<Customer>();
```

Assim como acontece com o uso anterior da Stack<T> classe criada usando o `Order` tipo, outra instância da classe especializada Stack<T> é criada. Os ponteiros contidos nela são definidos para referenciar uma área de memória do tamanho de um `Customer` tipo. Como o número de tipos de referência pode variar descontroladamente de programa para programa, a implementação em C# de genéricos reduz consideravelmente a quantidade de código reduzindo para um o número de classes especializadas criadas pelo compilador para classes genéricas de tipos de referência.

Além disso, quando uma classe C# genérica é instanciada usando um tipo de valor ou parâmetro de tipo de referência, a reflexão pode consultá-la em tempo de execução e seu tipo real e seu parâmetro de tipo podem ser apurados.

Consulte também

- System.Collections.Generic
- Introdução aos genéricos
- Genéricos

Referência de linguagem C#

A referência de linguagem fornece uma referência informal à sintaxe e expressões idiomáticas do C# para iniciantes e desenvolvedores experientes de C# e .NET.

Referência de linguagem C#

VISÃO GERAL

Estratégia de linguagem C#

REFERÊNCIA

Palavras-chave de C#

Operadores e expressões C#

Configurar versão da linguagem

Especificação da linguagem C# – rascunho do C# 8 em andamento

Novidades

NOVIDADES

Novidades no C# 14

O que há de novo no C# 13

Novidades do C# 12

Novidades do C# 11

REFERÊNCIA

Alterações de falha no compilador C#

Compatibilidade de versões

Mantenha contato

REFERÊNCIA

.NET Developer Community

YouTube

Twitter

Especificações

Leia as especificações detalhadas da linguagem C# e as especificações detalhadas dos recursos mais recentes.

Especificações ECMA e recursos mais recentes

VISÃO GERAL

Processo de especificação

Conteúdo detalhado da especificação ECMA

REFERÊNCIA

Especificações de recursos mais recentes

Rascunho da especificação C# ECMA – material introdutório

REFERÊNCIA

Prefácio

Introdução

REFERÊNCIA

Escopo

Referências normativas

Termos e definições

Descrição geral

Compatibilidade

Rascunho da especificação C# ECMA – Especificação da linguagem

REFERÊNCIA

Estrutura lexical

Conceitos básicos

Tipos

Variáveis

Conversões

Padrões

Expressões

Instruções

REFERÊNCIA

Namespaces

Classes

Estruturas

matrizes

Interfaces

Enums

Delegados

Exceções

Atributos

Código não seguro

Rascunho da especificação C# ECMA - Anexos

REFERÊNCIA

Gramática

Problemas de portabilidade

Biblioteca padrão

Comentários de documentação

Bibliografia