Contents

Guia do .NET

06/12/2019 • 3 minutes to read • Edit Online

O guia do .NET fornece uma grande quantidade de informações sobre o .NET. Dependendo da sua familiaridade com o .NET, talvez você queira explorar as diferentes seções deste guia e outras seções da documentação do .NET.

Novo no .NET?

Se você desejar uma visão geral de alto nível sobre o .NET, confira O que é .NET?.

Se você é novo no .NET, confira o artigo Introdução.

Se você preferir um tour guiado por meio dos recursos principais do .NET, confira o Tour no .NET.

Você também pode ler sobre os Componentes de arquitetura do .NET para obter uma visão geral das várias "partes" do .NET e como elas se encaixam.

Novo no .NET Core?

Se você é novo no .NET Core, confira Introdução ao .NET Core.

Novo no .NET Standard?

Se você é novo no .NET Standard, confira .NET Standard.

Portabilidade do Código do .NET Framework para o .NET Core

Se você estiver procurando portabilidade para um aplicativo, serviço ou algum componente de um sistema para o .NET Core, confira Portabilidade para .NET Core do .NET Framework.

Portabilidade de um pacote do NuGet do .NET Framework para .NET Standard ou .NET Core

Se você estiver procurando portabilidade de um pacote NuGet para o .NET Standard, confira Portabilidade para .NET Core do .NET Framework. As ferramentas para o .NET Standard e o .NET Core são compartilhadas, então o conteúdo será relevante para portabilidade para o NET Standard, bem como para .NET Core.

Interessado nos Principais Conceitos do .NET?

Se você estiver interessado em alguns dos principais conceitos do .NET, consulte:

- Componentes de arquitetura do .NET
- .NET Standard
- Interoperabilidade Nativa
- Coleta de lixo
- Tipos Base no .NET
- Coleções
- Datas, horas e fusos horários
- Programação Assíncrona

Além disso, confira cada guia de linguagem para saber mais sobre as três principais linguagens do .NET:

- Guia do C#
- Guia do F#
- Guia do Visual Basic

Referência API

Confira a Referência da API .NET para ver a variedade de APIs disponíveis.

Introdução

04/11/2019 • 2 minutes to read • Edit Online

Há várias formas para começar a trabalhar com o .NET. Como .NET é uma plataforma enorme, há vários artigos nesta documentação que mostram como você pode começar a trabalhar com o .NET, cada um com uma perspectiva diferente.

Introdução ao uso de linguagens .NET

- Os artigos de Introdução ao C# e os Tutoriais do C# fornecem diversas maneiras de começar a trabalhar de modo centrado no C#.
- Os tutoriais de Introdução ao F# fornecem três maneiras principais de usar o F#: com o Visual Studio, Visual Studio Code ou ferramentas de linha de comando.
- Os artigos de Introdução ao Visual Basic fornecem guias para usar o Visual Basic no Visual Studio.

Introdução ao uso do .NET Core

- Introdução ao .NET Core fornece uma visão geral de artigos que mostram como começar a trabalhar com o .NET Core em diferentes sistemas operacionais e usando ferramentas diferentes.
- Os Tutoriais do .NET Core detalham diversas maneiras de começar a trabalhar com o .NET Core usando o sistema operacional e as ferramentas de sua escolha.

Introdução ao uso do .NET Core no Docker

Introdução ao .NET e o Docker mostra como você pode usar o .NET Core em contêineres do Docker do Windows.

Tour do .NET

29/11/2019 • 17 minutes to read • Edit Online

O .NET é uma plataforma de desenvolvimento de uso geral. Ele tem vários recursos importantes, como suporte a várias linguagens de programação, modelos de programação assíncronos e simultâneos, além de interoperabilidade nativa, o que possibilita uma ampla variedade de cenários em várias plataformas.

Este artigo oferece um tour guiado por alguns dos principais recursos do .NET. Consulte o tópico Componentes de arquitetura do .NET para saber mais sobre as partes de arquitetura do .NET e para que elas são usadas.

Como executar os exemplos de código

Para saber como configurar um ambiente de desenvolvimento para executar os exemplos de código, consulte o tópico Introdução. Copie e cole os exemplos de código desta página no seu ambiente para executá-los.

Linguagens de programação

O .NET dá suporte a várias linguagens de programação. As implementações do .NET implementam o CLI (Common Language Infrastructure), que, entre outras coisas, especifica um runtime independente de linguagem e interoperabilidade de linguagem. Isso significa que você escolhe qualquer linguagem .NET para criar aplicativos e serviços no .NET.

A Microsoft desenvolve ativamente e dá suporte a três linguagens de .NET: C#, F# e VB (Visual Basic).

- A C# é simples, poderosa, fortemente tipada e orientada a objeto, mantendo a expressividade e elegância das linguagens de estilo C. Qualquer pessoa familiarizada com C e linguagens semelhantes encontra poucos problemas para adaptar-se à C#. Confira o Guia de C# para saber mais sobre o C#.
- F# é uma linguagem de programação de plataforma cruzada com prioridade para a parte funcional e que também dá suporte à programação imperativa e orientada a objeto tradicional. Confira o Guia de F# para saber mais sobre o F#.
- A Visual Basic é uma linguagem fácil de aprender, que você usa para criar uma variedade de aplicativos executados no .NET. Entre as linguagens de .NET, a sintaxe da VB é a mais próxima da linguagem humana comum, geralmente sendo mais fácil para pessoas novas no desenvolvimento de software.

Gerenciamento automático de memória

O .NET usa a (GC) coleta de lixo para fornecer gerenciamento automático de memória para os programas. A GC opera em uma abordagem lenta no gerenciamento de memória, dando prioridade à taxa de transferência do aplicativo em relação à coleta imediata da memória. Para saber mais sobre o GC do .NET, confira Noções básicas da coleta de lixo (GC).

As duas linhas a seguir alocam memória:

```
var title = ".NET Primer";
var list = new List<string>();
```

Não há nenhuma palavra-chave análoga para desalocar memória, pois a desalocação ocorre automaticamente quando o coletor de lixo recupera a memória por meio de sua execução agendada.

O coletor de lixo é um dos serviços que ajudam a garantir a *segurança da memória*. Um programa é considerado

de memória segura se ele acessa somente a memória alocada. Por exemplo, o runtime garante que um aplicativo não acesse memória não alocada além dos limites de uma matriz.

No exemplo a seguir, o runtime aciona uma exceção `InvalidIndexException` para impor segurança da memória:

```
int[] numbers = new int[42];
int number = numbers[42]; // Will throw an exception (indexes are 0-based)
```

Trabalhando com recursos não gerenciados

Alguns objetos fazem referência a *recursos não gerenciados*. Os recursos não gerenciados são recursos que não são mantidos automaticamente pelo runtime do .NET. Por exemplo, um identificador de arquivo é um recurso não gerenciado. Um objeto FileStream é um objeto gerenciado, mas ele faz referência a um identificador de arquivo, que não é gerenciado. Quando você termina de usar o FileStream, é necessário liberar o identificador de arquivo.

No .NET, objetos que fazem referência a recursos não gerenciados implementam a interface IDisposable. Quando você termina de usar o objeto, você chama o método Dispose() do objeto, responsável por liberar quaisquer recursos não gerenciados. As linguagens .NET fornecem uma instrução `using` conveniente para tais objetos, conforme mostrado no exemplo a seguir:

```
using System.IO;

using (FileStream stream = GetFileStream(context))
{
    // Operations on the stream
}
```

Uma vez que o bloco `using` é concluído, o runtime do .NET automaticamente chama o método Dispose() do objeto `stream`, que libera o identificador de arquivo. O runtime também faz isso se uma exceção faz com que o controle deixe o bloco.

Para obter mais detalhes, consulte os seguintes tópicos:

- Para C#, consulte o tópico Instrução Using (referência de C#).
- Para F#, consulte Gerenciamento de recursos: a palavra-chave Use.
- Para VB, consulte o tópico Instrução Using (Visual Basic).

Segurança de tipos

Um objeto é uma instância de um tipo específico. As únicas operações permitidas para um determinado objeto são aquelas do seu tipo. Um tipo `Dog` pode ter os métodos `Jump` e `WagTail`, mas não um método `SumTotal`. Um programa só chama os métodos que pertencem a um determinado tipo. Todas as outras chamadas resultam em um erro em tempo de compilação ou uma exceção de tempo de execução (no caso de usar recursos dinâmicos ou `object`).

As linguagens do .NET são orientadas a objeto, com hierarquias de classes base e classes derivadas. O runtime do .NET só permite conversões de objeto e chamadas alinhadas à hierarquia do objeto. Lembre-se de que todos os tipos definidos em qualquer linguagem .NET derivam do tipo base Object.

```
Dog dog = AnimalShelter.AdoptDog(); // Returns a Dog type.
Pet pet = (Pet)dog; // Dog derives from Pet.
pet.ActCute();
Car car = (Car)dog; // Will throw - no relationship between Car and Dog.
object temp = (object)dog; // Legal - a Dog is an object.
```

A segurança de tipos também é usada para ajudar a forçar o encapsulamento, garantindo a fidelidade das palavras-chave acessadoras. Palavras-chave acessadoras são artefatos que controlam o acesso a membros de um determinado tipo por outro código. Elas geralmente são usadas para vários tipos de dados dentro de um tipo que são usados para gerenciar seu comportamento.

```
private Dog _nextDogToBeAdopted = AnimalShelter.AdoptDog()
```

C#, VB e F# dão suporte à *inferência de tipo* de variável local. Inferência de tipos significa que o compilador deduz o tipo da expressão no lado esquerdo da expressão com base na expressão do lado direito. Isso não significa que a segurança de tipos é interrompida ou evitada. O tipo resultante realmente tem um tipo forte com tudo o que a expressão implica. No exemplo anterior, `dog` é reescrito para introduzir a inferência de tipos e o restante do exemplo permanece inalterado:

```
var dog = AnimalShelter.AdoptDog();
var pet = (Pet)dog;
pet.ActCute();
Car car = (Car)dog; // will throw - no relationship between Car and Dog
object temp = (object)dog; // legal - a Dog is an object
car = (Car)temp; // will throw - the runtime isn't fooled
car.Accelerate() // the dog won't like this, nor will the program get this far
```

O F# tem ainda mais recursos de inferência de tipos do que inferência de tipo de variável local de método encontrada no C# e no VB. Para obter mais informações, consulte Inferência de tipos.

Delegados e lambdas

Um delegado é representado por uma assinatura de método. Qualquer método com essa assinatura pode ser atribuído ao delegado e é executado quando o delegado é invocado.

Os delegados são como ponteiros de função do C++, exceto pelo fato de que eles são fortemente tipados. Eles são um tipo de método desconectado dentro do sistema de tipos CLR. Os métodos regulares são anexados a uma classe e só podem ser chamados diretamente por meio de convenções de chamada estáticas ou de instância.

No .NET, os delegados são comumente usados em manipuladores de eventos, na definição de operações assíncronas e em expressões lambda, que são a base da LINQ. Saiba mais no tópico Delegados e lambdas.

Genéricos

Os genéricos permitem que o programador introduza um *parâmetro de tipo* ao criar suas classes, o que permite que o código do cliente (os usuários do tipo) especifiquem o tipo exato a ser usado no lugar do parâmetro de tipo.

Os genéricos foram adicionados para ajudar os programadores a implementar estruturas de dados genéricos. Antes de sua adição, para que um tipo como o `List` fosse genérico, seria necessário que ele trabalhasse com elementos do tipo `object`. Isso ocasionava vários problemas de desempenho e de semântica, junto com a possibilidade de erros sutis de runtime. O mais notório deles é quando uma estrutura de dados contém, por exemplo, inteiros e cadeias de caracteres e uma `InvalidCastException` é gerada ao trabalhar com os membros da lista.

O exemplo a seguir mostra a execução de um programa básico usando uma instância de tipos List<T>:

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

namespace GenericsSampleShort
{
    public static void Main(string[] args)
    {
        // List<string> is the client way of specifying the actual type for the type parameter T
        List<string> listOfStrings = new List<string> { "First", "Second", "Third" };

        // listOfStrings can accept only strings, both on read and write.
        listOfStrings.Add("Fourth");

        // Below will throw a compile-time error, since the type parameter
        // specifies this list as containing only strings.
        listOfStrings.Add(1);
    }
}
```

Para obter mais informações, veja o tópico Visão geral de tipos genéricos (Genéricos).

Programação assíncrona

Programação assíncrona é um conceito de primeira classe no .NET, com suporte assíncrono no runtime, bibliotecas de estrutura e constructos da linguagem do .NET. Internamente, elas são baseadas em objetos (como `Task`) que aproveitam o sistema operacional para realizar trabalhos associados a E/S de modo tão eficiente quanto possível.

Para saber mais sobre programação assíncrona no .NET, comece com o tópico Visão geral da assincronia.

Consulta Integrada à Linguagem (LINQ)

O LINQ é um poderoso conjunto de recursos para C# e VB que permite que você escreva código simples e declarativo para operar em dados. Os dados podem ser de várias formas (como objetos na memória, um banco de dados SQL ou um documento XML), mas o código LINQ que você escreve não difere para cada fonte de dados.

Para obter mais informações e ver alguns exemplos, consulte o tópico LINQ (Consulta integrada à linguagem).

Interoperabilidade nativa

Todos os sistemas operacionais incluem uma API (interface de programação de aplicativo) que fornece serviços de sistema. O .NET fornece várias maneiras de chamar essas APIs.

A principal maneira de fazer interoperabilidade nativa é via "invocação de plataforma" ou P/Invoke, de forma abreviada, que tem suporte em plataformas Linux e Windows. A maneira somente para Windows de se fazer interoperabilidade nativa é conhecida como "interoperabilidade COM", que é usada para trabalhar com componentes COM em código gerenciado. Ela foi desenvolvida com base na infraestrutura de P/Invoke, mas funciona de maneiras levemente diferentes.

A maioria do suporte de interoperabilidade do Mono (e, portanto, do Xamarin) para Java e Objective-C é criado da mesma forma, ou seja, eles usam os mesmos princípios.

Para obter mais informações sobre a interoperabilidade nativa, confira o artigo Native interoperability (Interoperabilidade nativa).

Código não seguro

Dependendo do suporte da linguagem, o CLR permite acessar a memória nativa e realizar aritmética de ponteiro

por meio de código `unsafe`. Essas operações são necessárias para certos algoritmos e interoperabilidade do sistema. Embora poderoso, o uso de código não seguro não é recomendado a menos que seja necessário para fornecer interoperabilidade com APIs do sistema ou implementar um algoritmo mais eficiente. O código não seguro pode não ser executado da mesma maneira em ambientes diferentes e também perde os benefícios de um coletor de lixo e da segurança de tipos. Recomenda-se restringir e centralizar ao máximo o código não seguro, além de testar esse código detalhadamente.

O exemplo a seguir é uma versão modificada do `ToString()` método da classe `StringBuilder`. Ele ilustra como o uso do código `unsafe` pode implementar um algoritmo com eficiência movendo blocos de memória diretamente:

```
public override String ToString()
{
    if (Length == 0)
        return String.Empty;

    string ret = string.FastAllocateString(Length);
    StringBuilder chunk = this;
    unsafe
    {
        fixed (char* destinationPtr = ret)
        {
            do
            {
                if (chunk.m_ChunkLength > 0)
                {
                    // Copy these into local variables so that they are stable even in the presence of ----s
(hackers might do this)
                    char[] sourceArray = chunk.m_ChunkChars;
                    int chunkOffset = chunk.m_ChunkOffset;
                    int chunkLength = chunk.m_ChunkLength;

                    // Check that we will not overrun our boundaries.
                    if ((uint)(chunkLength + chunkOffset) <= ret.Length && (uint)chunkLength <=
(uint)sourceArray.Length)
                    {
                        fixed (char* sourcePtr = sourceArray)
                            string.wstrcpy(destinationPtr + chunkOffset, sourcePtr, chunkLength);
                    }
                    else
                    {
                        throw new ArgumentOutOfRangeException("chunkLength",
Environment.GetResourceString("ArgumentOutOfRange_Index"));
                    }
                }
                chunk = chunk.m_ChunkPrevious;
            } while (chunk != null);
        }
    }
    return ret;
}
```

{1>{2>Próximas etapas<2}<1}

Se você estiver interessado em um tour pelos recursos do C#, confira Tour do C#.

Se você estiver interessado em um tour pelos recursos do F#, veja o Tour do F#.

Se você deseja começar a escrever seu próprio código, viste Introdução.

Para saber mais sobre componentes importantes do .NET, confira Componentes de Arquitetura do .NET.

Componentes de arquitetura do .NET

06/12/2019 • 11 minutes to read • Edit Online

Um aplicativo .NET é desenvolvido para e é executado em uma ou mais *implementações do .NET*. Implementações do .NET incluem o .NET Framework, o .NET Core e o Mono. Há uma especificação de API comum a todas as implementações de .NET que é chamada de .NET Standard. Este artigo fornece uma breve introdução a cada um desses conceitos.

.NET Standard

O .NET Standard é um conjunto de APIs que são implementadas pela Biblioteca de classes base de uma implementação do .NET. De maneira mais formal, é uma especificação das APIs do .NET que compõem um conjunto uniforme de contratos nos quais você compila seu código. Esses contratos são implementados em cada implementação do .NET. Isso permite a portabilidade entre diferentes implementações de .NET, possibilitando efetivamente que seu código seja executado em qualquer lugar.

O .NET Standard também é uma estrutura de destino. Se seu código se destina a uma versão do .NET Standard, ele pode ser executado em qualquer implementação do .NET que dê suporte a essa versão do .NET Standard.

Para saber mais sobre o NET Standard e como destinar a ele, consulte o tópico .NET Standard.

Implementações do .NET

Cada implementação do .NET inclui os seguintes componentes:

- Um ou mais runtimes. Exemplos: o CLR para o .NET Framework, o CoreCLR e o CoreRT para o .NET Core.
- Uma biblioteca de classes que implemente o .NET Standard e possa implementar APIs adicionais. Exemplos: biblioteca de classes base do NET Framework, biblioteca de classes base do .NET Core.
- Opcionalmente, uma ou mais estruturas de aplicativo. Exemplos: ASP.net, Windows Formse Windows Presentation Foundation (WPF) estão incluídos no .NET Framework e no .NET Core.
- Opcionalmente, ferramentas de desenvolvimento. Algumas ferramentas de desenvolvimento são compartilhadas entre várias implementações.

Há quatro implementações principais de .NET que a Microsoft desenvolve e mantém ativamente: .NET Core, .NET Framework, Mono e UWP.

.NET Core

O .NET Core é uma implementação multiplataforma do .NET, projetado para lidar com cargas de trabalho de servidor e na nuvem em escala. Ele é executado no Windows, no Linux e no macOS. Ele implementa o .NET Standard, portanto o código direcionado para o .NET Standard pode ser executados no .NET Core. O ASP.NET Core, o Windows Forms e o WPF (Windows Presentation Foundation) são executados no .NET Core.

Para saber mais sobre o .NET Core, consulte a Guia .NET Core e Escolhendo entre o .NET Core e .NET Framework para aplicativos de servidor.

.NET Framework

O.NET Framework é a implementação original do .NET que existe desde 2002. É o mesmo .NET Framework que os desenvolvedores do .NET sempre usaram. As versões 4.5 e posteriores implementam o .NET Standard, assim, o código que se destina ao .NET Standard pode ser executado nessas versões do .NET Framework. Ele contém APIs adicionais específicas do Windows, como APIs para desenvolvimento de área de trabalho do Windows com o Windows Forms e o WPF. O .NET Framework é otimizado para a compilação de aplicativos da área de

trabalho do Windows.

Para saber mais sobre o .NET Framework, consulte o Guia do .NET Framework.

Mono

O Mono é uma implementação do .NET que é usada principalmente quando um pequeno runtime é necessário. É o runtime que impulsiona aplicativos Xamarin no Android, Mac, iOS, tvOS e watchOS e concentra-se principalmente em um impacto pequeno. O Mono também é plataforma para jogos criados com o mecanismo Unity.

Ele dá suporte a todas as versões do .NET Standard publicadas atualmente.

Historicamente, o Mono implementava a maior API do .NET Framework e emulava alguns dos recursos mais populares do Unix. Às vezes, ele é usado para executar aplicativos .NET que dependem desses recursos no Unix.

O Mono normalmente é usado com um compilador Just-In-Time, mas ele também apresenta um compilador estático completo (compilação Ahead Of Time) que é usado em plataformas como iOS.

Para saber mais sobre o Mono, consulte a Documentação do Mono.

UWP (Plataforma Universal do Windows)

A UWP é uma implementação do .NET que é usada para criar aplicativos do Windows modernos e sensíveis ao toque, bem como software para a IoT (Internet das Coisas). Ela foi projetada para unificar os diferentes tipos de dispositivos que você talvez tenha como destinho, incluindo PCs, tablets, phablets, telefones e até mesmo ao Xbox. A UWP fornece muitos serviços, como um repositório centralizado de aplicativos, um ambiente de execução (AppContainer) e um conjunto de APIs do Windows para usar em vez das APIS do Win32 (WinRT). Os aplicativos podem ser escritos em C++, C#, VB.NET e JavaScript. Ao usar o C# e VB.NET, as APIs do .NET são fornecidas pelo .NET Core.

Para saber mais sobre a UWP, consulte Introdução à Plataforma Universal do Windows.

Runtimes do .NET

Um runtime é o ambiente de execução de um programa gerenciado. O SO faz parte do ambiente do runtime, mas não faz parte do runtime do .NET. Aqui estão alguns exemplos de runtimes do .NET:

- CLR (Common Language Runtime) para o .NET Framework
- Core Common Language Runtime (CoreCLR) para o .NET Core
- .NET Native para a Plataforma Universal do Windows
- O runtime Mono para Xamarin.iOS, Xamarin.Android, Xamarin.Mac e a estrutura de área de trabalho do Mono

Ferramentas do .NET e infraestrutura comum

Você tem acesso a um amplo conjunto de ferramentas e componentes de infraestrutura que funcionam com todas as implementações do .NET. Elas incluem, mas não são limitadas a, o seguinte:

- As linguagens do .NET e seus compiladores
- O sistema de projetos do .NET (com base em arquivos *.csproj*, *.vbproj* e *.fsproj*)
- MSBuild, o mecanismo de build usado para compilar projetos
- NuGet, o gerenciador de pacotes da Microsoft para o .NET
- Ferramentas de orquestração de build em software livre, como CAKE e FAKE

Padrões aplicáveis

A C# linguagem e as especificações de Common Language Infrastructure (CLI) são padronizadas por meio de ®

ECMA International. As primeiras edições desses padrões foram publicadas pela ECMA em dezembro de 2001.

As revisões subsequentes para os padrões foram desenvolvidas pelos grupos de tarefas TC49C#-tg2 () e TC49-TG3 (CLI) no comitê técnico de linguagens de programação (TC49) e adotadas pelo assembly geral da ECMA e subsequentemente por ISO/IEC JTC 1 por meio do processo de controle rápido ISO.

Padrões mais recentes

Os documentos ECMA oficiais a seguir estão disponíveis C# para o e a CLI (TR-84):

- **O C# idioma padrão (versão 5,0)** : ECMA-334. pdf
- **O Common Language Infrastructure**: isso está disponível em formato PDF e formato zip .
- **Informações derivadas do arquivo XML da partição IV**: isso está disponível em formatos PDF e zip .

Os documentos ISO/IEC oficiais estão disponíveis na página de padrões do ISO/IEC publicamente disponível . Esses links são diretos dessa página:

- **Tecnologia da informação-linguagens C#de programação-** : ISO/IEC 23270:2018
- **Tecnologia da informação — partições de Common Language Infrastructure (CLI) I para vi**: ISO/IEC 23271:2012
- **Tecnologia da informação — Common Language Infrastructure (CLI) — relatório técnico sobre informações derivadas do arquivo XML da partição IV**: ISO/IEC TR 23272:2011

Consulte também

- Escolhendo entre o .NET Core e .NET Framework para aplicativos de servidor
- .NET Standard
- Guia do .NET Core
- Guia do .NET Framework
- Guia do C#
- Guia do F#
- Guia do VB.NET

.NET Standard

31/10/2019 • 21 minutes to read • Edit Online

O .NET Standard é uma especificação formal de APIs do .NET que devem estar disponíveis em todas as implementações do .NET. A motivação por trás do .NET Standard é estabelecer maior uniformidade no ecossistema do .NET. A ECMA 335 continua estabelecendo a uniformidade de comportamento da implementação do .NET, mas não há especificação semelhante para as BCLs (Bibliotecas de Classe Base) do .NET para implementações da biblioteca do .NET.

O .NET Standard permite os seguintes principais cenários:

- Define um conjunto uniforme de APIs de BCL para todas as implementações do .NET a serem implementadas, independentemente da carga de trabalho.
- Permite que os desenvolvedores criem bibliotecas portáteis, utilizáveis entre implementações do .NET, usando esse mesmo conjunto de APIs.
- Reduz ou até elimina a compilação condicional de origem compartilhada em razão das APIs do .NET, apenas para APIs do sistema operacional.

As diversas implementações do .NET se destinam a versões específicas do .NET Standard. Cada versão de implementação do .NET anuncia a versão mais alta do .NET Standard a qual ela dá suporte, uma afirmação que significa que também há suporte para versões anteriores. Por exemplo, o .NET Framework 4.6 implementa o .NET Standard 1.3, o que significa que ele expõe todas as APIs definidas nas versões 1.0 a 1.3 do .NET Standard. Da mesma forma, o .NET Framework 4.6.1 implementa o .NET Standard 1.4, enquanto o .NET Core 1.0 implementa o .NET Standard 1.6.

Suporte à implementação do .NET

A seguinte tabela lista as versões de plataforma **mínimas** que dão suporte a cada versão do .NET Standard. Isso significa que versões posteriores de uma plataforma listada também dão suporte para a versão correspondente do .NET Standard. Por exemplo, o .NET Core 2.2 dá suporte ao .NET Standard 2.0 e anteriores.

.NET STANDARD	1.0	1.1	1.2	1.3	1.4	1.5	1.6	2.0	2.1
.NET Core	1.0	1.0	1.0	1.0	1.0	1.0	1.0	2.0	3.0
.NET Framework [1]	4.5	4.5	4.5.1	4.6	4.6.1	4.6.1 [2]	4.6.1 [2]	4.6.1 [2]	N/A[3]
Mono	4.6	4.6	4.6	4.6	4.6	4.6	4.6	5.4	6.4
Xamarin.iOS	10.0	10.0	10.0	10.0	10.0	10.0	10.0	10.14	12.16
Xamarin.Mac	3.0	3.0	3.0	3.0	3.0	3.0	3.0	3.8	5.16

.NET STANDARD	1.0	1.1	1.2	1.3	1.4	1.5	1.6	2.0	2.1
Xamarin.Android	7.0	7.0	7.0	7.0	7.0	7.0	7.0	8.0	10.0
Plataforma Universal do Windows	10.0	10.0	10.0	10.0	10.0	10.0.16299	10.0.16299	10.0.16299	TBD
Unity	2018.1	2018.1	2018.1	2018.1	2018.1	2018.1	2018.1	2018.1	TBD

1 as versões listadas para .NET Framework se aplicam ao SDK do .NET Core 2,0 e às versões posteriores das ferramentas. As versões mais antigas usaram um mapeamento diferente para .NET Standard 1,5 e superior. Você pode baixar ferramentas para o .NET Core Tools para visual studio 2015 se não for possível atualizar para o visual Studio 2017 ou uma versão posterior.

2 as versões listadas aqui representam as regras que o NuGet usa para determinar se uma determinada biblioteca de .NET Standard é aplicável. Embora o NuGet considere .NET Framework 4.6.1 como suporte a .NET Standard 1,5 até 2,0, há vários problemas com o consumo de .NET Standard bibliotecas que foram criadas para essas versões de projetos .NET Framework 4.6.1. Para projetos .NET Framework que precisam usar essas bibliotecas, recomendamos que você atualize o projeto para o destino .NET Framework 4.7.2 ou superior.

3 .NET Framework não oferecerá suporte a .NET Standard 2,1 ou versões posteriores. Para obter mais detalhes, consulte o anúncio do .NET Standard 2,1.

- As colunas representam versões do .NET Standard. Cada célula de cabeçalho é um link para um documento que mostra quais APIs foram adicionadas a essa versão do .NET Standard.
- As linhas representam as diferentes implementações do .NET.
- O número de versão em cada célula indica a versão *mínima* da implementação de que você precisará para direcionar essa versão do .NET Standard.
- Para obter uma tabela interativa, consulte Versões do .NET Standard.

Para localizar a versão mais recente do .NET Standard para a qual você pode direcionar, siga estas etapas:

1. Localize a linha que indica a implementação do .NET na qual você deseja executar.
2. Localize a coluna nessa linha que indica sua versão, da direita para a esquerda.
3. O cabeçalho da coluna indica a versão do .NET Standard compatível com o destino. Você também pode ter como destino qualquer versão inferior do .NET Standard. Versões superiores do .NET Standard também darão suporte à implementação.
4. Repita esse processo para cada plataforma de destino. Se você tiver mais de uma plataforma de destino, escolha a versão menos recente entre elas. Por exemplo, se você quiser executar no .NET Framework 4.5 e no .NET Core 1.0, a versão mais recente do .NET Standard que você pode usar é o .NET Standard 1.1.

Para qual versão do .NET Standard direcionar

Ao escolher uma versão do .NET Standard, você deve considerar a seguinte compensação:

- Quanto mais recente a versão, mais APIs estarão disponíveis para você.
- Quanto menos recente a versão, mais plataformas a implementarão.

Em geral, recomendamos que você direcione para a versão *menos recente* possível do .NET Standard. Portanto, depois de localizar a versão mais recente do .NET Standard para a qual você pode direcionar, execute estas etapas:

1. Direcione para a próxima versão menos recente do .NET Standard e compile seu projeto.
2. Se seu projeto for compilado com êxito, repita a etapa 1. Caso contrário, redirecione para a próxima versão mais recente, e essa será a versão que você deve usar.

No entanto, ter como destino versões do .NET Standard inferiores introduz diversas dependências de suporte. Se o projeto for destinado ao .NET Standard 1.x, recomendamos que você *também* tenha como destino o .NET Standard 2.0. Isso simplifica o grafo de dependência para os usuários da sua biblioteca executados em estruturas compatíveis do .NET Standard 2.0, além de reduzir o número de pacotes que eles precisam baixar.

Regras de controle de versão do .NET Standard

Há duas regras principais de controle de versão:

- Aditivo: as versões do .NET Standard são círculos logicamente concêntricos: versões mais recentes incorporam todas as APIs das versões anteriores. Não há alterações significativas entre as versões.
- Imutável: após o envio, as versões do .NET Standard serão congeladas. As novas APIs ficarão disponíveis primeiro em implementações específicas do .NET, como .NET Core. Se a banca examinadora do .NET Standard achar que as novas APIs devem estar disponíveis para todas as implementações do .NET, elas serão adicionadas em uma nova versão do .NET Standard.

Especificação

A especificação do .NET Standard é um conjunto padronizado de APIs. A especificação é mantida por implementadores do .NET, especificamente Microsoft (inclui o .NET Framework, .NET Core e Mono) e Unity. Um processo de comentários público é usado como parte do estabelecimento de novas versões do .NET Standard por meio do GitHub.

Artefatos oficiais

A especificação oficial é um conjunto de arquivos .cs que definem as APIs que fazem parte do padrão. O diretório ref em dotnet/standard repository define as APIs do .NET Standard.

O metapacote NETStandard.Library (de origem) descreve o conjunto de bibliotecas que define (parcialmente) uma ou mais versões do .NET Standard.

Um componente específico, como o `System.Runtime`, descreve:

- Parte do .NET Standard (apenas seu escopo).
- Várias versões do .NET Standard, para esse escopo.

Artefatos derivados são fornecidos para habilitar leitura mais conveniente e permitir determinados cenários do desenvolvedor (por exemplo, usando um compilador).

- Lista de APIs em markdown
- Assemblies de referência, distribuídos como pacotes NuGet e referenciados pelo metapacote NETStandard.Library.

Representação de pacote

O meio de distribuição principal dos assemblies de referência do .NET Standard são os pacotes NuGet. As implementações são entregues de várias formas, apropriadas para cada implementação do .NET.

Pacotes NuGet são direcionados a uma ou mais estruturas. Os pacotes do .NET Standard são direcionados à estrutura do ".NET Standard". É possível direcionar a estrutura do .NET Standard usando o `netstandard` TFM compacto (por exemplo, `netstandard1.4`). Bibliotecas destinadas a execução em vários runtimes devem ter essa estrutura como alvo. Para obter o mais amplo conjunto de APIs, direcione `netstandard2.0`, pois o número de APIs disponíveis mais do que dobrou entre o .NET Standard 1.6 e 2.0.

O metapacote `NETStandard.Library` referencia o conjunto completo de pacotes NuGet que definem o .NET

Standard. A maneira mais comum de apontar `netstandard` é fazer referência a esse metapacote. Ele descreve e fornece acesso às ~40 bibliotecas .NET e APIs associadas, que definem a .NET Standard. Você pode referenciar pacotes adicionais destinados a `netstandard` para obter acesso a APIs adicionais.

Controle de versão

A especificação não é única, mas um conjunto de versões de APIs de crescimento incremental e linear. A primeira versão do padrão estabelece um conjunto de linhas de base de APIs. As versões subsequentes adicionam APIs e herdam APIs definidas por versões anteriores. Não há nenhuma provisão estabelecido para remoção de APIs do padrão.

O .NET Standard não é específico a nenhuma implementação do .NET, nem corresponde ao esquema de controle de versão de nenhum desses runtimes.

APIs adicionadas a qualquer implementação (por exemplo, .NET Framework, .NET Core e Mono) podem ser consideradas como candidatas a serem adicionadas à especificação, especialmente se forem consideradas fundamentais por natureza. As novas versões do .NET Standard são criadas com base em versões de implementação do .NET, permitindo que você destine a novas APIs de uma PCL do .NET Standard. Os mecanismos de controle de versão são descritos mais detalhadamente em Controle de versão do .NET Core.

O controle de versão do .NET Standard é importante para uso. Com uma versão do .NET Standard você pode usar bibliotecas direcionadas a essa mesma versão ou inferior. A abordagem a seguir descreve o fluxo de trabalho para uso de PCLs do .NET Standard, específico ao direcionamento do .NET Standard.

- Selecione uma versão do .NET Standard a ser usada para a PCL.
- Use bibliotecas que dependem da mesma versão do .NET Standard ou inferior.
- Se você encontrar uma biblioteca que depende de uma versão mais recente do .NET Standard, deverá adotar essa mesma versão ou decidir não usar essa biblioteca.

Direcionamento do .NET Standard

Você pode criar .NET Standard Libraries usando uma combinação de `netstandard` estrutura e metapacote NETStandard.Library. Você pode ver exemplos de direcionamento do .NET Standard com as ferramentas do .NET Core.

Modo de compatibilidade do .NET framework

O modo de compatibilidade do .NET Framework foi introduzido a partir do .NET Standard 2.0. Esse modo de compatibilidade permite que os projetos do .NET Standard referenciem as bibliotecas do .NET Framework como se elas fossem compiladas para o .NET Standard. Fazer referência a bibliotecas do .NET Framework não funciona para todos os projetos, como as bibliotecas que usam APIs do WPF (Windows Presentation Foundation).

Para obter mais informações, veja .NET Framework compatibility mode (Modo de compatibilidade do .NET Framework).

Bibliotecas do .NET standard e Visual Studio

Para criar bibliotecas do .NET Standard no Visual Studio, verifique se tem o Visual Studio 2017 versão 15.3 ou posterior instalado no Windows ou o Visual Studio para Mac versão 7.1 ou posterior instalado no macOS.

Se você precisar apenas consumir as bibliotecas do .NET Standard 2.0 em seus projetos, também será possível fazer isso no Visual Studio 2015. No entanto, é necessário ter o NuGet cliente 3.6 ou posterior instalado. É possível baixar o cliente do NuGet para Visual Studio 2015 na página downloads do NuGet.

Comparação com bibliotecas de classes portáteis

O .NET Standard é o substituto das PCLs (Bibliotecas de classe portáteis). O .NET Standard aprimora a experiência de criar bibliotecas portáteis, selecionando um BCL padrão e estabelecendo maior uniformidade entre as implementações do .NET como resultado. Uma biblioteca direcionada ao .NET Standard é uma PCL ou uma "PCL baseada no .NET Standard". PCLs existentes são "PCLs baseadas em perfil".

Os perfis do .NET Standard e da PCL foram criados para finalidades semelhantes, mas também diferem de maneiras básicas.

Semelhanças:

- Definem APIs que podem ser usadas para compartilhamento de código binário.

Diferenças:

- O .NET Standard é um conjunto estruturado de APIs, enquanto os perfis de PCL são definidos por interseções de plataformas existentes.
- O .NET Standard tem versões lineares, ao passo que os perfis de PCL não.
- Os perfis de PCL representam plataformas da Microsoft, enquanto o .NET Standard é independente de plataforma.

Compatibilidade com PCL

O .NET Standard é compatível com um subconjunto de perfis de PCL. O .NET Standard 1.0, 1.1 e 1.2 se sobrepõem, cada um, com um conjunto de perfis de PCL. Essa sobreposição foi criada por dois motivos:

- Habilite PCLs baseadas no .NET Standard para referenciar PCLs baseadas em perfil.
- Permita que PCLs baseadas em perfil sejam empacotadas como PCLs baseadas no .NET Standard.

Compatibilidade de PCL baseada em perfil é fornecida pelo pacote NuGet. Microsoft.NETCore.Portable.Compatibility. Essa dependência é necessária ao referenciar pacotes NuGet que contêm PCLs baseadas em perfil.

PCLs baseadas em perfil e empacotadas como `netstandard` são mais fáceis de serem consumidas do que PCLs baseadas em perfil tipicamente empacotadas. `netstandard` o empacotamento é compatível com os usuários existentes.

Você pode ver o conjunto de perfis PCL que são compatíveis com o .NET Standard:

PERFIL DO PCL	.NET STANDARD	PLATAFORMAS PCL
Profile7	1.1	.NET Framework 4.5, Windows 8
Profile31	1.0	Windows 8.1, Windows Phone Silverlight 8.1
Profile32	1.2	Windows 8.1, Windows Phone 8.1
Profile44	1.2	.NET Framework 4.5.1, Windows 8.1
Profile49	1.0	.NET Framework 4.5, Windows Phone Silverlight 8
Profile78	1.0	.NET Framework 4.5, Windows 8, Windows Phone Silverlight 8
Profile84	1.0	Windows Phone 8.1, Windows Phone Silverlight 8.1

PERFIL DO PCL	.NET STANDARD	PLATAFORMAS PCL
Profile111	1.1	.NET Framework 4.5, Windows 8, Windows Phone 8.1
Profile151	1.2	.NET Framework 4.5.1, Windows 8.1, Windows Phone 8.1
Profile157	1.0	Windows 8.1, Windows Phone 8.1, Windows Phone Silverlight 8.1
Profile259	1.0	.NET Framework 4.5, Windows 8, Windows Phone 8.1, Windows Phone Silverlight 8

Consulte também

- Versões do .NET Standard
- Criar uma biblioteca de .NET Standard
- Direcionamento de plataforma cruzada

Novidades no .NET Standard

31/10/2019 • 7 minutes to read • Edit Online

O .NET Standard é uma especificação formal que define um conjunto com versões das APIs que devem estar disponíveis em implementações .NET que estejam de acordo com a versão standard. O .NET Standard é destinado a desenvolvedores de bibliotecas. Uma biblioteca direcionada a uma versão do .NET Standard pode ser usada em qualquer implementação do .NET Framework, do .NET Core ou do Xamarin que dê suporte a essa versão do Standard.

A versão mais recente do .NET Standard é a 2.0. Ele está incluído no SDK do .NET Core 2,0, bem como no Visual Studio 2017 versão 15,3 com a carga de trabalho do .NET Core instalada.

Implementações .NET com suporte

O .NET Standard 2.0 tem suporte das seguintes implementações do .NET:

- .NET Core 2.0 ou posterior
- .NET Framework 4.6.1 ou posterior
- Mono 5.4 ou posterior
- Xamarin.iOS 10.14 ou posterior
- Xamarin.Mac 3.8 ou posterior
- Xamarin.Android 8.0 ou posterior
- Plataforma Universal do Windows 10.0.16299 ou posterior

Novidades no .NET Standard 2.0

O .NET Standard 2.0 inclui estes recursos novos:

Um conjunto muito maior de APIs

Na versão 1.6, o .NET Standard incluía um subconjunto de APIs comparativamente pequeno. Entre os excluídos estavam várias APIs usadas normalmente no .NET Framework ou no Xamarin. Isso complica o desenvolvimento, pois exige que os desenvolvedores encontrem substituições adequadas para APIs conhecidas quando desenvolvem aplicativos e bibliotecas direcionadas a várias implementações do .NET. O .NET Standard 2.0 resolve essa limitação adicionando mais de 20.000 APIs que estavam disponíveis no .NET Standard 1.6, a versão anterior do Standard. Para obter uma lista com as APIs que foram adicionadas ao .NET Standard 2.0, confira .NET Standard 2.0 versus 1.6.

Algumas das adições ao namespace System no .NET Standard 2.0 incluem:

- Suporte para a classe AppDomain.
- Aprimoramento do suporte para trabalhar com matrizes de membros adicionais na classe Array.
- Aprimoramento do suporte para trabalhar com atributos de membros adicionais na classe Attribute.
- Aprimoramento do suporte ao calendário e opções de formatação adicionais para valores DateTime.
- Funcionalidade de arredondamento Decimal adicional.
- Funcionalidade adicional na classe Environment.
- Aprimoramento do controle sobre o coletor de lixo por meio da classe GC.
- Aprimoramento do suporte para comparação, enumeração e normalização de cadeia de caracteres na classe String.
- Suporte para ajustes de horário de verão e tempos de transição nas classes TimeZoneInfo.AdjustmentRule e

TimeZoneInfo.TransitionTime.

- Aprimoramento considerável da funcionalidade na classe Type.
- Aprimoramento do suporte para desserialização de objetos de exceção com a adição de um construtor de exceção com os parâmetros SerializationInfo e StreamingContext.

Suporte a bibliotecas do .NET Framework

A grande maioria das bibliotecas são direcionadas ao .NET Framework em vez do .NET Standard. No entanto, a maioria das chamadas nessas bibliotecas são para APIs incluídas no .NET Standard 2.0. A partir do .NET Standard 2.0, você pode acessar bibliotecas do .NET Framework de uma biblioteca do .NET Standard usando uma shim de compatibilidade. Essa camada de compatibilidade é transparente para os desenvolvedores; você não precisa fazer nada para tirar proveito das bibliotecas do .NET Framework.

O único requisito é que as APIs chamadas pela biblioteca de classes .NET Framework estejam incluídas no .NET Standard 2.0.

Suporte para Visual Basic

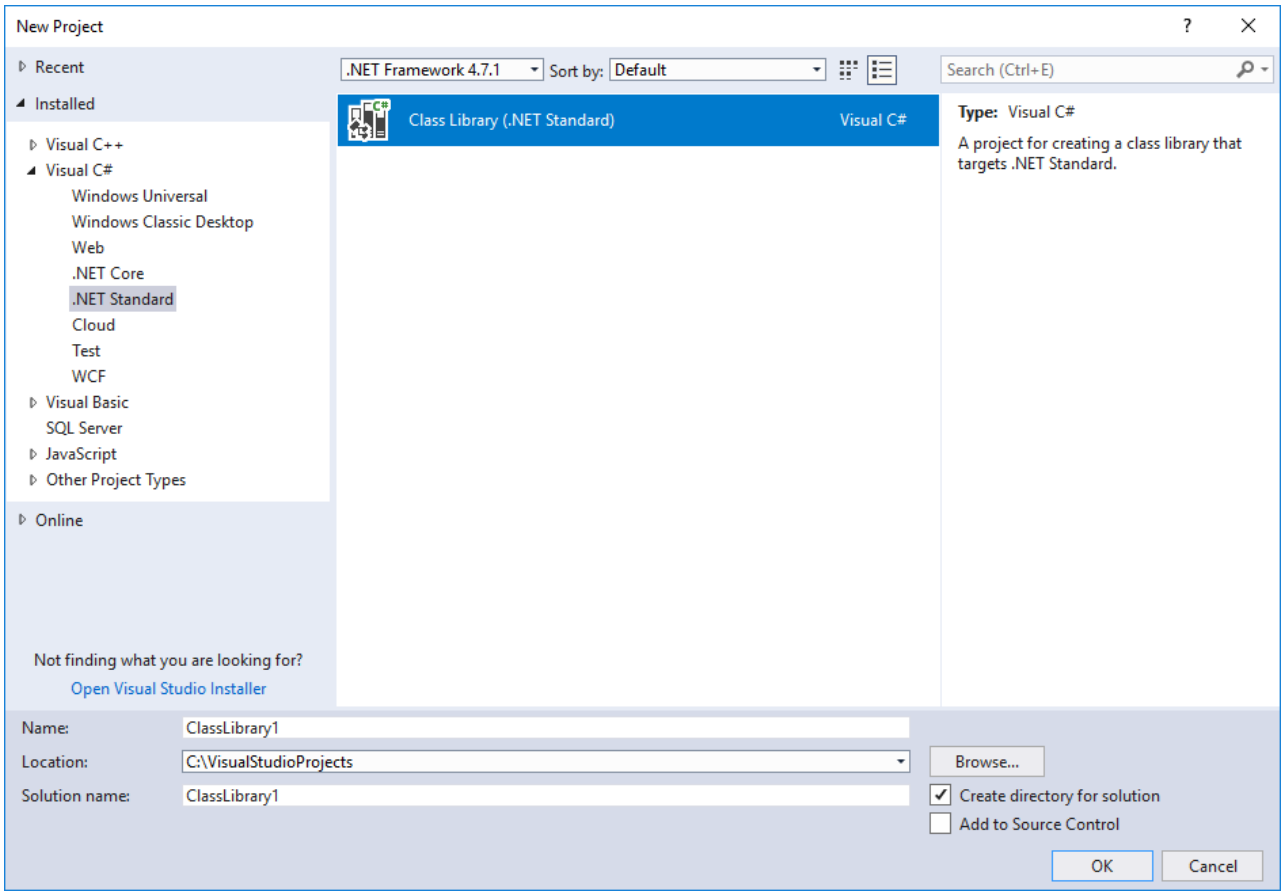
Agora, você pode desenvolver bibliotecas .NET Standard no Visual Basic. Para Visual Basic desenvolvedores que usam o Visual Studio 2017 versão 15,3 ou posterior com a carga de trabalho do .NET Core instalada, o Visual Studio agora inclui um modelo de biblioteca de classes .NET Standard. Para desenvolvedores em Visual Basic que usam outros ambientes e ferramentas de desenvolvimento, use o comando dotnet new para criar um projeto de biblioteca do .NET Standard. Para saber mais, confira o Suporte a ferramentas para bibliotecas .NET Standard.

Suporte a ferramentas para bibliotecas .NET Standard

Com o lançamento do .NET Core 2.0 e do .NET Standard 2.0, o Visual Studio 2017 e a CLI (Interface de linha de comando) do .NET Core incluem o suporte a ferramentas para criação de bibliotecas .NET Standard.

Se instalar o Visual Studio com a carga de trabalho **desenvolvimento de plataforma cruzada do .NET Core**, você poderá criar um projeto de biblioteca .NET Standard 2.0 usando um modelo de projeto, como mostra a figura a seguir:

- C#
- Visual Basic



Se você estiver usando a CLI do .NET Core, o seguinte comando dotnet new criará um projeto de biblioteca de classes direcionado ao .NET Standard 2.0:

```
dotnet new classlib
```

Consulte também

- .NET Standard
- Apresentando o .NET Standard

Estruturas de destino em projetos no estilo SDK

06/12/2019 • 8 minutes to read • Edit Online

Ao destinar a uma estrutura em um aplicativo ou uma biblioteca, você está especificando o conjunto de APIs que deseja disponibilizar para o aplicativo ou a biblioteca. Você especifica a estrutura de destino em seu arquivo de projeto usando os TFMs (Monikers da Estrutura de Destino).

Um aplicativo ou uma biblioteca pode ser destinada a uma versão do .NET Standard. As versões do .NET Standard representam conjuntos de APIs padronizadas entre todas as implementações do .NET. Por exemplo, uma biblioteca pode se destinar ao NET Standard 1.6 e obter acesso a APIs que funcionam no .NET Core e .NET Framework usando a mesma base de código.

Um aplicativo ou uma biblioteca também pode se destinar a uma implementação específica do .NET para obter acesso a APIs específicas de implementação. Por exemplo, um aplicativo que tem como destino o Xamarin.iOS (por exemplo, `Xamarin.iOS10`) obtém acesso aos wrappers da API fornecidos para Xamarin iOS do iOS 10 ou um aplicativo que tem como destino a UWP (Plataforma Universal do Windows, `uap10.0`) tem acesso a APIs que são compiladas para dispositivos que executam o Windows 10.

Para algumas estruturas de destino (por exemplo, o .NET Framework), as APIs são definidas pelos assemblies que a estrutura instala em um sistema e pode incluir as APIs da estrutura do aplicativo (por exemplo, ASP.NET).

Para estruturas de destino com base em pacote (por exemplo, .NET Standard e .NET Core), as APIs são definidas pelos pacotes incluídos no aplicativo ou na biblioteca. Um *metapacote* é um pacote NuGet que não tem nenhum conteúdo próprio, mas é uma lista de dependências (outros pacotes). Uma estrutura de destino com base em pacote NuGet especifica implicitamente um metapacote que faz referência a todos os pacotes que, juntos, compõem a estrutura.

Versões mais recentes de estrutura de destino

A tabela a seguir define as estruturas de destino mais comuns, como elas são referenciadas e qual versão do .NET Standard elas implementam. Estas versões de estrutura de destino são as versões estáveis mais recentes. As versões de pré-lançamento não são mostradas. Um TFM (Moniker da Estrutura de Destino) é um formato de token padronizado para especificar a estrutura de destino de um aplicativo ou uma biblioteca do .NET.

ESTRUTURA DE DESTINO	ÚLTIMA VERSÃO ESTÁVEL	TFM (MONIKER DE ESTRUTURA DE DESTINO)	IMPLEMENTADO VERSÃO DO .NET STANDARD
.NET Standard	2.1	netstandard 2.1	{1>N/A<1}
.NET Core	3,1	netcoreapp 3.1	2.1
.NET Framework	4.8	net48	2.0

Versões de estrutura de destino com suporte

Normalmente, uma estrutura de destino é referenciada por um TFM. A tabela a seguir mostra as estruturas de destino com suporte pelo SDK do .NET Core e o cliente do NuGet. Equivalentes estão mostrados entre colchetes. Por exemplo, `win81` é um TFM equivalente ao `netcore451`.

ESTRUTURA DE DESTINO	TFM
.NET Standard	netstandard1.0 netstandard1.1 netstandard1.2 netstandard1.3 netstandard1.4 netstandard1.5 netstandard1.6 netstandard2.0 netstandard 2.1
.NET Core	netcoreapp1.0 netcoreapp1.1 netcoreapp2.0 netcoreapp2.1 netcoreapp2.2 netcoreapp 3.0 netcoreapp 3.1
.NET Framework	net11 net20 net35 net40 net403 net45 net451 net452 net46 net461 net462 net47 net471 net472 net48
Windows Store	netcore [netcore45] netcore45 [win] [win8] netcore451 [win81]
.NET Micro Framework	netmf
Silverlight	sl4 sl5
Windows Phone	wp [wp7] wp7 wp75 wp8 wp81 wpa81
Plataforma Universal do Windows	uap [uap10.0] uap10.0 [win10] [netcore50]

Como especificar estruturas de destino

As estruturas de destino são especificadas no arquivo de projeto. Quando uma única estrutura de destino é especificada, use o elemento **TargetFramework**. O arquivo de projeto de aplicativo de console a seguir

demonstra como direcionar o .NET Core 3,0:

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>netcoreapp3.0</TargetFramework>
  </PropertyGroup>

</Project>
```

Ao especificar várias estruturas de destino, você pode fazer referência a assemblies para cada estrutura de destino de forma condicional. Em seu código, você pode condicionalmente compilar em relação a esses assemblies usando símbolos de pré-processador com a lógica *if-then-else*.

O seguinte arquivo de projeto de biblioteca tem como destino as APIs do .NET Standard (`netstandard1.4`) e APIs do .NET Framework (`net40` e `net45`). Use o elemento **TargetFrameworks** plural com várias estruturas de destino. Observe como os atributos `Condition` incluem pacotes específicos de implementação quando a biblioteca é compilada para os dois TFMs de .NET Framework:

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <TargetFrameworks>netstandard1.4;net40;net45</TargetFrameworks>
  </PropertyGroup>

  <!-- Conditionally obtain references for the .NET Framework 4.0 target -->
  <ItemGroup Condition=" '$(TargetFramework)' == 'net40' ">
    <Reference Include="System.Net" />
  </ItemGroup>

  <!-- Conditionally obtain references for the .NET Framework 4.5 target -->
  <ItemGroup Condition=" '$(TargetFramework)' == 'net45' ">
    <Reference Include="System.Net.Http" />
    <Reference Include="System.Threading.Tasks" />
  </ItemGroup>

</Project>
```

No aplicativo ou na biblioteca, você escreve código condicional para compilar para cada estrutura de destino:

```
public class MyClass
{
    static void Main()
    {
#if NET40
        Console.WriteLine("Target framework: .NET Framework 4.0");
#elif NET45
        Console.WriteLine("Target framework: .NET Framework 4.5");
#else
        Console.WriteLine("Target framework: .NET Standard 1.4");
#endif
    }
}
```

O sistema de compilação reconhece os símbolos de pré-processador que representam as estruturas de destino mostradas na tabela de versões do Framework de destino com suporte quando você está usando projetos em estilo SDK. Ao usar um símbolo que representa um TFM do .NET Standard ou .NET Core, substitua o ponto por um sublinhado e altere as letras minúsculas por maiúsculas (por exemplo, o símbolo de `netstandard1.4` é `NETSTANDARD1_4`).

A lista completa de símbolos de pré-processador para estruturas de destino do .NET Core é:

FRAMEWORKS DE DESTINO	SÍMBOLOS
.NET Framework	`NETFRAMEWORK`, `NET20`, `NET35`, `NET40`, `NET45`, `NET451`, `NET452`, `NET46`, `NET461`, `NET462`, `NET47`, `NET471`, `NET472`, `NET48`
.NET Standard	`NETSTANDARD`, `NETSTANDARD1_0`, `NETSTANDARD1_1`, `NETSTANDARD1_2`, `NETSTANDARD1_3`, `NETSTANDARD1_4`, `NETSTANDARD1_5`, `NETSTANDARD1_6`, `NETSTANDARD2_0`, `NETSTANDARD2_1`
.NET Core	`NETCOREAPP`, `NETCOREAPP1_0`, `NETCOREAPP1_1`, `NETCOREAPP2_0`, `NETCOREAPP2_1`, `NETCOREAPP2_2`, `NETCOREAPP3_0`, `NETCOREAPP3_1`

Estruturas de destino preteridas

As seguintes estruturas de destino estão preteridas. Os pacotes direcionados a essas estruturas de destino devem ser migrados para as substituições indicadas.

TFM PRETERIDO	SUBSTITUIÇÃO
aspnet50 aspnetcore50 dnxcore50 dnx dnx45 dnx451 dnx452	netcoreapp
dotnet dotnet50 dotnet51 dotnet52 dotnet53 dotnet54 dotnet55 dotnet56	netstandard
netcore50	uap10.0
win	netcore45
win8	netcore45
win81	netcore451
win10	uap10.0
winrt	netcore45

Consulte também

- Pacotes, Metapacotes e Estruturas
- Desenvolvendo Bibliotecas com as Ferramentas de Plataforma Cruzada
- .NET Standard
- Controle de versão do .NET Core
- Repositório GitHub dotnet/standard
- Repositório GitHub de Ferramentas NuGet
- Perfis de estrutura no .NET

Glossário .NET

23/10/2019 • 24 minutes to read • Edit Online

A principal meta deste glossário é esclarecer os significados de termos e acrônimos selecionados que aparecem com frequência na documentação do .NET sem definições.

AOT

Compilador Ahead-of-Time.

Semelhante ao JIT, esse compilador também converte IL em código de máquina. Diferentemente da compilação JIT, a compilação AOT acontece antes que o aplicativo seja executado e normalmente é executada em um computador diferente. Como as cadeias da ferramenta da AOT não compilam em tempo de execução, elas não têm que minimizar o tempo gasto na compilação. Isso significa que elas podem gastar mais tempo em otimização. Como o contexto da AOT é o aplicativo inteiro, o compilador AOT também executa a vinculação de módulo cruzado e a análise de programa inteiro, o que significa que todas as referências são seguidas e um único executável é produzido.

Confira CoreRT e .NET Native.

ASP.NET

A implementação do ASP.NET original que é fornecida com o .NET Framework.

Às vezes, o ASP.NET é um termo abrangente que se refere a ambas as implementações de ASP.NET, incluindo o ASP.NET Core. O significado que o termo carrega em uma determinada instância é determinado pelo contexto. Consulte ASP.NET 4.x quando quiser esclarecer que você não está usando ASP.NET para as duas implementações.

Confira Documentação do ASP.NET.

ASP.NET Core

Uma implementação multiplataforma de alto desempenho e software livre do ASP.NET com base no .NET Core.

Confira Documentação do ASP.NET Core.

assembly

Um arquivo *.dll/ .exe* que contém uma coleção de APIs que podem ser chamadas por aplicativos ou outros assemblies.

Um assembly pode incluir tipos como interfaces, classes, estruturas, enumerações e delegados. Às vezes, os assemblies em uma pasta *bin* de projeto são chamados de *binários*. Consulte também biblioteca.

CLR

Common Language Runtime.

O significado exato depende do contexto, mas geralmente se refere ao tempo de execução do .NET Framework. O CLR manipula a alocação e o gerenciamento de memória. O CLR também é uma máquina virtual que não só executa aplicativos, mas também gera e compila código dinamicamente usando um compilador JIT. A implementação atual do CLR da Microsoft é somente para Windows.

CoreCLR

Common Language Runtime do .NET Core.

Esse CLR é criado com a mesma base de código que o CLR. Originalmente, o CoreCLR era o tempo de execução do Silverlight e foi projetado para ser executado em várias plataformas, especificamente o Windows e OS X. Agora o CoreCLR faz parte do .NET Core e representa uma versão simplificada do CLR. Ainda é um tempo de execução multiplataforma, incluindo também suporte para várias distribuições do Linux. O CoreCLR também é uma máquina virtual com recursos JIT e de execução de código.

CoreFX

BCL (biblioteca de classes base) do .NET Core

Um conjunto de bibliotecas que compõem os namespaces System.* e, até certo limite, Microsoft.*. A BCL é uma estrutura de nível inferior e de uso geral, base para a criação de estruturas de aplicativo de nível mais alto, como o ASP.NET Core. O código-fonte da BCL do .NET Core está contido no repositório CoreFX. No entanto, a maioria das APIs do .NET Core também estão disponíveis no .NET Framework, portanto você pode pensar no CoreFX como um fork da BCL do .NET Framework.

CoreRT

Tempo de execução do .NET Core.

Ao contrário do CLR/CoreCLR, o CoreRT não é uma máquina virtual, o que significa que ele não inclui os recursos para gerar e executar código dinamicamente, já que não inclui um JIT. No entanto, ele inclui a GC e a capacidade de RTTI (identificação de tipo de tempo de execução) e reflexão. Contudo, seu sistema de tipos é projetado para que os metadados para reflexão não sejam necessários. Isso permite ter uma cadeia de ferramentas AOT que possa desvincular metadados supérfluos e, mais importante, identificar código que o aplicativo não usa. O CoreRT está em desenvolvimento.

Consulte Introdução ao .NET Native e ao CoreRT

várias plataformas

A capacidade de desenvolver e executar um aplicativo que pode ser usado em vários sistemas operacionais diferentes, como Linux, Windows e iOS, sem que seja preciso reescrever especificamente para cada um deles. Isso permite a reutilização de código e a consistência entre os aplicativos em diferentes plataformas.

ecossistema

Todos os softwares de tempo de execução, as ferramentas de desenvolvimento e os recursos da comunidade que são usados para compilar e executar aplicativos de uma determinada tecnologia.

O termo "Ecossistema do .NET" difere de termos semelhantes, como "Pilha do .NET", em relação à inclusão de bibliotecas e aplicativos de terceiros. Veja um exemplo em uma frase:

- "A motivação por trás do .NET Standard é estabelecer maior uniformidade no ecossistema do .NET."

estrutura

Em geral, uma coleção abrangente de APIs que facilita o desenvolvimento e a implantação de aplicativos que são baseados em uma tecnologia específica. Nesse sentido geral, o ASP.NET Core e o Windows Forms são exemplos de estruturas de aplicativo. Consulte também biblioteca.

A palavra "estrutura" tem um significado técnico mais específico nos seguintes termos:

- .NET Framework
- estrutura de destino
- TFM (Moniker de Estrutura de Destino)

Na documentação existente, "estrutura" às vezes se refere a uma implementação do .NET. Por exemplo, um artigo pode chamar o .NET Core de uma estrutura. Planejamos eliminar da documentação esse uso confuso da palavra.

GC

Coletor de lixo.

O coletor de lixo é uma implementação do gerenciamento automático de memória. O GC libera a memória ocupada por objetos que não estejam mais em uso.

Consulte Coleta de lixo.

IL

Linguagem intermediária.

As linguagens .NET de nível mais alto , como C#, são compiladas em um conjunto de instruções independente de hardware, que é chamado de IL (linguagem intermediária). A IL às vezes é conhecida como MSIL (Microsoft Intermediate Language) ou CIL (Common Intermediate Language).

JIT

Compilador Just-In-Time.

Semelhante ao AOT, esse compilador converte a IL em um código de máquina que o processador entenda. Ao contrário da AOT, a compilação JIT acontece sob demanda e é executada no mesmo computador em que o código precisa ser executado. Como a compilação JIT ocorre durante a execução do aplicativo, o tempo de compilação faz parte do tempo de execução. Assim, os compiladores JIT precisam equilibrar o tempo gasto com a otimização do código em relação à economia que o código resultante pode produzir. Mas um JIT reconhece o hardware e pode evitar que os desenvolvedores tenham que fornecer implementações diferentes.

implementação do .NET

Uma implementação do .NET inclui o seguinte:

- Um ou mais tempos de execução. Exemplos: CLR, CoreCLR e CoreRT.
- Uma biblioteca de classes que implemente uma versão do .NET Standard, podendo incluir APIs adicionais. Exemplos: biblioteca de classes base do NET Framework, biblioteca de classes base do .NET Core.
- Opcionalmente, uma ou mais estruturas de aplicativo. Exemplos: ASP.NET, Windows Forms e WPF estão incluídos no .NET Framework.
- Opcionalmente, ferramentas de desenvolvimento. Algumas ferramentas de desenvolvimento são compartilhadas entre várias implementações.

Exemplos de implementações do .NET:

- .NET Framework
- .NET Core
- UWP (Plataforma Universal do Windows)

biblioteca

Uma coleção de APIs que podem ser chamadas por aplicativos ou outras bibliotecas. Uma biblioteca do .NET é composta de um ou mais assemblies.

A biblioteca de palavras e a estrutura geralmente são usadas como sinônimos.

metapacote

Um pacote NuGet que não tem nenhuma biblioteca própria, mas é apenas uma lista de dependências. Os pacotes incluídos podem, opcionalmente, estabelecer a API para uma estrutura de destino.

Consulte Pacotes, metapacotes e estruturas

Mono

Mono é uma implementação do .NET multiplataforma de software livre usada principalmente quando é necessário um pequeno tempo de execução. É o tempo de execução que impulsiona aplicativos Xamarin no Android, Mac, iOS, tvOS e watchOS e se concentra, principalmente, em aplicativos que exigem superfície reduzida.

Ele dá suporte a todas as versões do .NET Standard publicadas atualmente.

Historicamente, o Mono implementava a maior API do .NET Framework e emulava alguns dos recursos mais populares do Unix. Às vezes, ele é usado para executar aplicativos .NET que dependem desses recursos no Unix.

O Mono normalmente é usado com um compilador Just-In-Time, mas ele também apresenta um compilador estático completo (compilação Ahead Of Time) que é usado em plataformas como iOS.

Para saber mais sobre o Mono, consulte a Documentação do Mono.

.NET

O termo coletivo para .NET Standard e todas as implementações de .NET, bem como as cargas de trabalho. Sempre em maiúsculas, nunca ".Net".

Consulte o guia do .NET

.NET Core

Uma implementação multiplataforma de alto desempenho e software livre do .NET. Inclui o CoreCLR (Core Common Language Runtime), o CoreRT (tempo de execução Core AOT, em desenvolvimento), a biblioteca de classes base do Core e o SDK do Core.

Consulte .NET Core.

CLI do .NET Core

Uma cadeia de ferramentas multiplataforma para o desenvolvimento de aplicativos .NET Core.

Consulte Ferramentas da CLI (interface de linha de comando) do .NET Core.

SDK do .NET Core

Um conjunto de bibliotecas e ferramentas que permitem aos desenvolvedores criar bibliotecas e aplicativos do .NET Core. Inclui a CLI do .NET Core para a criação de aplicativos, bibliotecas e tempo de execução do .NET Core para criar e executar aplicativos e o executável do dotnet (*dotnet.exe*) que executa comandos de CLI e executa aplicativos.

Consulte a Visão geral do SDK do .NET Core.

.NET Framework

Uma implementação do .NET que é executado somente no Windows. Inclui o CLR (Common Language Runtime), a Biblioteca de classes base e as bibliotecas de estrutura do aplicativo, como ASP.NET, Windows Forms e WPF.

Consulte o Guia do .NET Framework.

.NET Nativo

Uma cadeia de ferramentas de compilador que gera código nativo AOT (Ahead Of Time), em vez de JIT (Just-In-Time).

A compilação acontece no computador do desenvolvedor, semelhante à maneira como um compilador e vinculador C++ funciona. Ela remove código não utilizado e gasta mais tempo otimizando-o. Ele extrai o código de bibliotecas e os mescla no executável. O resultado é um módulo único que representa o aplicativo inteiro.

A UWP foi a primeira estrutura de aplicativo com suporte pelo .NET Native. Agora, há suporte para criação de aplicativos de console nativo para macOS, Windows e Linux.

Consulte Introdução ao .NET Native e ao CoreRT

.NET Standard

Uma especificação formal das APIs do .NET que estão disponíveis em cada implementação do .NET.

Às vezes, a especificação do .NET Standard também é chamada de biblioteca na documentação. Como uma biblioteca inclui implementações de API e não apenas especificações (interfaces), é um equívoco chamar .NET Standard de "biblioteca". Planejamos eliminar esse uso da documentação, exceto em referência ao nome do metapacote do .NET Standard (`NETStandard.Library`).

Consulte .NET Standard.

NGEN

Geração (de imagem) nativa.

Você pode pensar nessa tecnologia como um compilador JIT persistente. Ela geralmente compila o código no computador em que o código é executado, mas a compilação normalmente ocorre no momento da instalação.

pacote

Um pacote NuGet — ou apenas um pacote — é um arquivo *.zip* com um ou mais assemblies de mesmo nome, junto com metadados adicionais, como o nome do autor.

O arquivo *.zip* tem uma extensão *.nupkg* e pode conter ativos, como arquivos *.dll* e arquivos *.xml*, para uso com várias versões e estruturas de destino. Quando instalado em um aplicativo ou uma biblioteca, os ativos apropriados são selecionados com base na estrutura de destino especificada pelo aplicativo ou pela biblioteca. Os ativos que definem a interface estão na pasta *ref* e os ativos que definem a implementação estão na pasta *lib*.

platform

Um sistema operacional e o hardware em que ele é executado, como macOS, Windows, Linux, iOS e Android.

Veja alguns exemplos de uso nessas frases:

- "O .NET Core é uma implementação multiplataforma do .NET."
- "Os perfis de PCL representam plataformas da Microsoft enquanto que o .NET Standard é independente de

plataforma."

A documentação do .NET frequentemente usa "plataforma .NET" para significar uma implementação do .NET ou a pilha do .NET, incluindo todas as implementações. Os dois usos tendem a ser confundidos com o significado (SO/hardware) principal, portanto planejamos eliminar esses usos da documentação.

tempo de execução

O ambiente de execução de um programa gerenciado.

O SO faz parte do ambiente do tempo de execução, mas não faz parte do tempo de execução do .NET. Aqui estão alguns exemplos de tempos de execução do .NET:

- CLR (Common Language Runtime)
- Core Common Language Runtime (CoreCLR)
- .NET Native (para UWP)
- tempo de execução Mono

Às vezes, a documentação do .NET usa "tempo de execução" para se referir a uma implementação do .NET. Por exemplo, nas seguintes sentenças "tempo de execução" deve ser substituído por "implementação":

- "Os diversos tempos de execução do .NET implementam versões específicas do .NET Standard."
- "Bibliotecas destinadas à execução em vários tempos de execução devem ter essa estrutura como destino." (referindo-se ao .NET Standard)
- "Os diversos tempos de execução do .NET implementam versões específicas do .NET Standard. ... Cada versão de tempo de execução do .NET anuncia a última versão do .NET Standard à qual ele dá suporte..."

Planejamos eliminar esse uso inconsistente.

stack

Um conjunto de tecnologias de programação que são usadas para compilar e executar aplicativos.

"A pilha do .NET" refere-se ao .NET Standard e a todas as implementações do .NET. A frase "uma pilha do .NET" pode se referir a uma implementação do .NET.

estrutura de destino

A coleção de APIs da qual um aplicativo ou biblioteca do .NET depende.

Um aplicativo ou uma biblioteca pode se destinar a uma versão do .NET Standard (por exemplo, .NET Standard 2.0), que é a especificação de um conjunto padronizado de APIs entre todas as implementações de .NET. Um aplicativo ou uma biblioteca também pode se destinar a uma versão de uma implementação específica do .NET, obtendo acesso a APIs específicas da implementação. Por exemplo, um aplicativo que tem como destino o Xamarin.iOS obtém acesso aos wrappers da API fornecidos para Xamarin iOS.

Para algumas estruturas de destino (por exemplo, o .NET Framework) as APIs disponíveis são definidas pelos assemblies que uma implementação do .NET instala em um sistema, que podem incluir as APIs de estrutura do aplicativo (por exemplo, ASP.NET, WinForms). Para estruturas de destino baseadas em pacote (como .NET Standard e .NET Core), a APIs de estrutura são definidas por pacotes instalados no aplicativo ou na biblioteca. Nesse caso, a estrutura de destino especifica implicitamente um metapacote que faz referência a todos os pacotes que, juntos, compõem a estrutura.

Consulte Estruturas de destino.

TFM

Moniker da estrutura de destino.

Um formato de token padronizado para especificar a estrutura de destino de um aplicativo ou uma biblioteca do .NET. As estruturas de destino são geralmente referenciadas por um nome curto, como `net462`. Os TFMs de formato longo (como. .NETFramework,Version=4.6.2) existem, mas não são geralmente usados para especificar uma estrutura de destino.

Consulte Estruturas de destino.

UWP

Plataforma Universal do Windows.

Uma implementação do .NET que é usada para criar aplicativos do Windows modernos e sensíveis ao toque, bem como software para a IoT (Internet das Coisas). Ela foi projetada para unificar os diferentes tipos de dispositivos que você talvez tenha como destinho, incluindo PCs, tablets, phablets, telefones e até mesmo ao Xbox. A UWP fornece muitos serviços, como um repositório centralizado de aplicativos, um ambiente de execução (AppContainer) e um conjunto de APIs do Windows para usar em vez das APIS do Win32 (WinRT). Os aplicativos podem ser escritos em C++, C#, VB.NET e JavaScript. Ao usar o C# e VB.NET, as APIs do .NET são fornecidas pelo .NET Core.

Consulte também

- Guia do .NET
- Guia do .NET Framework
- .NET Core
- Visão geral do ASP.NET
- Visão geral do ASP.NET Core

Diretrizes da biblioteca de software livre

23/10/2019 • 2 minutes to read • Edit Online

Estas diretrizes fornecem recomendações para que os desenvolvedores criem bibliotecas .NET de alta qualidade. Esta documentação concentra-se no *quê* e no *porquê* ao criar uma biblioteca .NET, não no *como*.

Aspectos das bibliotecas .NET de software livre de alta qualidade:

- **Inclusivas** – as bibliotecas .NET de qualidade se esforçam para dar suporte a várias plataformas, linguagens de programação e aplicativos.
- **Estáveis** – as bibliotecas .NET de qualidade coexistem no ecossistema do .NET, em execução em aplicativos criados com várias bibliotecas.
- **Projetadas para evoluir** – as bibliotecas .NET devem melhorar e evoluir ao longo do tempo, continuando a dar suporte aos usuários existentes.
- **Depuráveis** – as bibliotecas .NET devem usar as ferramentas mais recentes para criar uma ótima experiência de depuração para os usuários.
- **Confiáveis** – as bibliotecas .NET têm a confiança dos desenvolvedores por publicarem no NuGet usando práticas recomendadas de segurança.

Introdução

Tipos de recomendações

Cada artigo apresenta quatro tipos de recomendações: **Fazer**, **Considerar**, **Evitar**, e **Não Fazer**. O tipo de recomendação indica a intensidade em que ela deve ser seguida.

Procure quase sempre seguir a recomendação **Fazer**. Por exemplo:

✔ **FAZER** a distribuição de sua biblioteca usando um pacote NuGet.

Por outro lado, as recomendações **Considerar** geralmente devem ser seguidas, mas há exceções à regra legítimas e você não precisará se preocupar caso não possa seguir as diretrizes:

✔ **CONSIDERAR** o uso do SemVer 2.0.0 para criar a versão do seu pacote NuGet.

As recomendações **Evitar** mencionam coisas que em geral não são ideais, mas há casos em que, às vezes, quebrar a regra faz sentido:

EVITAR referências do pacote NuGet que demandam uma versão exata.

Por fim, as recomendações **Não fazer** indicam algo que você quase nunca deve fazer:

NÃO FAZER a publicação de versões de nome forte e sem nome forte da biblioteca. Por exemplo, `Contoso.Api` e `Contoso.Api.StrongNamed`.

AVANÇAR

Escolhendo entre o .NET Core e .NET Framework para aplicativos de servidor

23/10/2019 • 14 minutes to read • Edit Online

Há duas implementações com suporte para a compilação de aplicativos de servidor com o .NET: .NET Framework e .NET Core. Ambas compartilham muitos dos mesmos componentes e você pode compartilhar código entre as duas. No entanto, há diferenças fundamentais entre as duas e sua escolha depende do que você deseja realizar. Este artigo fornece diretrizes sobre quando usar cada uma.

Use o .NET Core para o aplicativo para servidores se:

- Você tiver necessidades de plataforma cruzada.
- Você estiver direcionando microsserviços.
- Você estiver usando contêineres do Docker.
- Você precisar de alto desempenho e sistemas escalonáveis.
- Você precisar de versões do .NET correspondentes a cada aplicativo.

Use o .NET Framework para o aplicativo para servidores se:

- Seu aplicativo usar o .NET Framework atualmente (a recomendação é estender em vez de migrar).
- Seu aplicativo usa bibliotecas .NET de terceiros ou pacotes NuGet não disponíveis para o .NET Core.
- Seu aplicativo usa tecnologias .NET que não estão disponíveis para o .NET Core.
- Seu aplicativo usa uma plataforma que não oferece suporte ao .NET Core. Windows, macOS e Linux dão suporte ao .NET Core.

Quando escolher o .NET Core

As seguintes seções oferecem uma explicação mais detalhada sobre os motivos para escolher o .NET Core mencionados anteriormente.

Necessidades de plataforma cruzada

Se seu aplicativo (web/serviço) precisa ser executado em várias plataformas (Windows, Linux e macOS), use o .NET Core.

O .NET Core dá suporte aos sistemas operacionais mencionados anteriormente como sua estação de trabalho de desenvolvimento. O Visual Studio fornece um IDE (ambiente de desenvolvimento integrado) para Windows e macOS. Você também pode usar o Visual Studio Code, que é executado no Windows, Linux e macOS. O Visual Studio Code dá suporte ao .NET Core, incluindo IntelliSense e depuração. A maioria dos editores de terceiros, como Sublime, Emacs e VI, trabalham com o .NET Core. Esses editores de terceiros obtém o IntelliSense do editor usando o Omnisharp. Também é possível evitar o uso de editores de código e usar diretamente as ferramentas de CLI do .NET Core, disponíveis para todas as plataformas com suporte.

Arquitetura de microsserviços

Uma arquitetura de microsserviços possibilita uma combinação de tecnologias em um limite de serviço. Essa combinação de tecnologias permite uma adoção gradual do .NET Core para novos microsserviços que funcionam com outros serviços ou microsserviços. Por exemplo, você pode combinar microsserviços ou serviços desenvolvidos com .NET Framework, Java, Ruby ou outras tecnologias monolíticas.

Há muitas plataformas de infraestrutura disponíveis. O Azure Service Fabric é criado para sistemas de microsserviço grandes e complexos. O Serviço de Aplicativo do Azure é uma boa escolha para microsserviços

sem monitoração de estado. Alternativas de microsserviços baseadas em Docker se adaptam a qualquer tipo de abordagem de microsserviços, conforme explicado na seção Contêineres. Todas essas plataformas oferecem suporte ao .NET Core e são ideais para hospedar microsserviços.

Para obter mais informações sobre a arquitetura de microsserviços, consulte Microsserviços do .NET. Arquitetura para aplicativos .NET em contêineres.

Contêineres

Os contêineres são frequentemente usados em conjunto com uma arquitetura de microsserviços. Os contêineres também podem ser usados para colocar em contêiner os aplicativos ou serviços Web que seguem qualquer padrão de arquitetura. O .NET Framework pode ser usado em contêineres do Windows, mas a modularidade e a natureza leve do .NET Core o tornam uma opção melhor para contêineres. Ao criar e implantar um contêiner, o tamanho de sua imagem será muito menor com o .NET Core que com o .NET Framework. Como ele é multiplataforma, é possível implantar aplicativos para servidores em contêineres do Docker do Linux, por exemplo.

Os contêineres do Docker podem ser hospedados em sua própria infraestrutura do Windows ou do Linux ou em um serviço de nuvem, como o Serviço de Kubernetes do Azure. O Serviço de Kubernetes do Azure pode gerenciar, orquestrar e dimensionar aplicativos baseados em contêiner na nuvem.

Uma necessidade de alto desempenho e sistemas escalonáveis

Quando o seu sistema precisa do melhor desempenho e escalabilidade possíveis, o .NET Core e o ASP.NET Core são as melhores opções. O tempo de execução do servidor de alto desempenho para Windows Server e Linux tornam o .NET uma estrutura da Web de desempenho superior nas avaliações da TechEmpower.

O desempenho e a escalabilidade são especialmente relevantes para arquiteturas de microsserviços, nas quais centenas de microsserviços podem estar em execução. Com o ASP.NET Core, os sistemas são executados com um número bem menor de servidores/VMs (Máquinas Virtuais). A redução em servidores/VMs gera economia em infraestrutura e hospedagem.

Necessidade de lado a lado de versões do .NET por nível de aplicativo

Para instalar aplicativos com dependências em diferentes versões do .NET, é recomendável o .NET Core. O .NET core oferece instalação lado a lado de versões diferentes do tempo de execução do .NET Core no mesmo computador. Essa instalação lado a lado permite vários serviços no mesmo servidor, cada um em sua própria versão do .NET Core. Ela também reduz os riscos e gera economia financeira nas operações de TI e atualizações de aplicativo.

Quando escolher o .NET Framework

O .NET Core oferece benefícios significativos para novos aplicativos e padrões de aplicativo. No entanto, o .NET Framework continua sendo a escolha natural para muitos cenários existentes e, portanto, não é substituído pelo .NET Core em todos os aplicativos para servidores.

Aplicativos .NET Framework atuais

Na maioria dos casos, não é necessário migrar aplicativos existentes para o .NET Core. Em vez disso, uma abordagem recomendada é usar o .NET Core ao estender um aplicativo existente, por exemplo, para gravar um novo serviço Web no ASP.NET Core.

Necessidade de usar bibliotecas .NET de terceiros ou pacotes NuGet não disponíveis para o .NET Core

As bibliotecas estão rapidamente adotando o .NET Standard. O .NET Standard permite o compartilhamento de código entre todas as implementações do .NET, incluindo o .NET Core. Com o .NET Standard 2.0, isso é ainda mais fácil:

- A superfície de API se tornou muito maior.
- Foi introduzido um modo de compatibilidade do .NET Framework. Este modo de compatibilidade permite que

os projetos do .NET Standard/.NET Core referenciem bibliotecas do .NET Framework. Para saber mais sobre o modo de compatibilidade, consulte o Comunicado do .NET Standard 2.0.

Portanto, apenas nos casos em que as bibliotecas ou pacotes NuGet usarem tecnologias que não estão disponíveis no .NET Standard/.NET Core, você precisará usar o .NET Framework.

Necessidade de usar as tecnologias do .NET que não estão disponíveis para .NET Core

Algumas tecnologias do .NET Framework não estão disponíveis no .NET Core. Algumas delas poderão ser disponibilizadas em versões posteriores do .NET Core. Outras não se aplicam aos novos padrões de aplicativo direcionados pelo .NET Core e podem não ser mais disponibilizadas. A lista a seguir mostra as tecnologias mais comuns não encontradas no .NET Core:

- Aplicativos Web Forms do ASP.NET: O Web Forms do ASP.NET só está disponível no .NET Framework. O ASP.NET Core não pode ser usado para Web Forms do ASP.NET. Não há planos para trazer os Web Forms do ASP.NET para o .NET Core.
- Aplicativos de Páginas da Web do ASP.NET: As Páginas da Web do ASP.NET não estão incluídas no ASP.NET Core.
- Implementação de serviços do WCF. Mesmo que haja uma Biblioteca de Cliente WCF para consumir serviços WCF no .NET Core, a implementação de servidor do WCF só está disponível no .NET Framework no momento. Esse cenário não é parte do plano atual para o .NET Core, mas ele está sendo considerado para o futuro.
- Serviços relacionados ao fluxo de trabalho: O Windows WF (Workflow Foundation), os Serviços de Fluxo de Trabalho (WCF + WF em um único serviço) e o WCF Data Services (anteriormente conhecido como "ADO.NET Data Services") só estão disponíveis no .NET Framework. Não há planos para trazer WF/WCF+WF/WCF Data Services para o .NET Core.
- Suporte de linguagem: Atualmente, há suporte para Visual Basic e F# no .NET Core, mas não para todos os tipos de projeto. Para obter uma lista de modelos de projeto com suporte, consulte Opções de modelo para o dotnet new.

Além do roteiro oficial, há outras estruturas a serem transferidas para o .NET Core. Para obter uma lista completa, consulte os problemas de CoreFX marcados como port-to-core. Essa lista não representa um compromisso da Microsoft para levar esses componentes para o .NET Core. Ela simplesmente captura o desejo da comunidade para que isso seja feito. Se você se importa com qualquer um dos componentes marcados como `port-to-core`, participe das discussões no GitHub. E se você acha que algo está faltando, envie uma nova questão no repositório do CoreFX.

Necessidade de usar uma plataforma que não oferece suporte a .NET Core

Algumas plataformas de terceiros ou da Microsoft não oferecem suporte a .NET Core. Alguns serviços do Azure fornecem um SDK que ainda não está disponível para ser consumido no .NET Core. Esta é uma circunstância transitória, pois todos os serviços do Azure usam o .NET Core. Enquanto isso, você sempre pode usar a API REST equivalente em vez do SDK do cliente.

Consulte também

- Escolher entre o ASP.NET e o ASP.NET Core
- ASP.NET Core direcionado para o .NET Framework
- Estruturas de destino
- Guia do .NET Core
- Portabilidade do .NET Framework para .NET Core
- Introdução ao .NET e ao Docker
- Visão Geral dos Componentes .NET

- Microsserviços .NET. Arquitetura para aplicativos .NET em contêineres

O que é "código gerenciado"?

23/10/2019 • 5 minutes to read • Edit Online

Ao trabalhar com o .NET Framework, geralmente você encontrará o termo "código gerenciado". Este documento explicará o que esse termo significa e obterá informações adicionais sobre ele.

Em resumo, código gerenciado é exatamente isso: código cuja execução é gerenciada por um tempo de execução. Nesse caso, o tempo de execução em questão é chamado de **Common Language Runtime** ou CLR, independentemente da implementação (Mono ou do .NET Framework ou .NET Core). CLR é responsável por pegar o código gerenciado, compilá-lo em código de computador e, em seguida, executá-lo. Além disso, o tempo de execução fornece vários serviços importantes, como gerenciamento automático de memória, limites de segurança, segurança de digitação etc.

Compare isso à maneira pela qual você executaria um programa C/C++, também chamado de "código não gerenciado". No mundo não gerenciado, o programador é responsável por quase tudo. O programa real é, essencialmente, um binário que o sistema operacional (SO) carrega na memória e inicia. Tudo, desde o gerenciamento da memória até as considerações de segurança, é responsabilidade do programador.

O código gerenciado é escrito em uma das linguagens de alto nível que podem ser executadas sobre o .NET, como C#, Visual Basic, F# e outras. Quando você compila o código gravado nessas linguagens com seu respectivo compilador, não obtém o código do computador. Você obtém o código de **Linguagem intermediária** que o tempo de execução compila e executa. C++ é a única exceção a essa regra, já que também pode produzir binários nativos e não gerenciados que são executados no Windows.

Linguagem intermediária e execução

O que é "Linguagem Intermediária" (ou IL)? É um produto de compilação de código gravado em linguagens .NET de alto nível. Quando você compila o código gravado em uma dessas linguagens, obtém um binário composto de IL. É importante observar que a IL é independente de qualquer linguagem específica executada em tempo de execução; há até mesmo uma especificação separada para que você pode ler se desejar.

Depois de gerar a IL do código de alto nível, execute-o. Aqui, a CLR assume e inicia o processo de compilação **Just-In-Time** ou **colocando em compilação JIT** seu código da IL no código do computador, que pode ser executado em uma CPU. Dessa forma, a CLR sabe exatamente o que o código está fazendo e pode, efetivamente, *gerenciá*-lo.

A Linguagem intermediária também é chamada CIL (Common Intermediate Language) ou MSIL (Microsoft Intermediate Language).

Interoperabilidade de código não gerenciado

Naturalmente, a CLR permite passar os limites entre o mundo gerenciado e o não gerenciado e há muitos códigos que fazem isso, mesmo nas Bibliotecas de classes base. Isso é chamado de **interoperabilidade** ou apenas **interop**. Essas condições permitiriam, por exemplo, resumir uma biblioteca não gerenciada e chamá-la. No entanto, é importante observar que quando você faz isso, quando o código passa os limites do tempo de execução, o gerenciamento real da execução está novamente no código não gerenciado e, portanto, se enquadra nas mesmas restrições.

Semelhante a isso, C# é uma linguagem que permite que você use construções não gerenciadas como ponteiros diretamente no código, utilizando o que é conhecido como **contexto inseguro** que designa um trecho de código para o qual a execução não é gerenciada pela CLR.

Mais recursos

- Visão geral do .NET Framework
- Código não seguro e ponteiros
- Interoperabilidade nativa

Gerenciamento automático de memória

31/10/2019 • 14 minutes to read • Edit Online

O gerenciamento automático de memória é um dos serviços que o Common Language Runtime fornece durante a Execução gerenciada. O coletor de lixo do Common Language Runtime gerencia a alocação e a liberação de memória para um aplicativo. Para desenvolvedores, isso significa que você não tem que escrever código para executar tarefas de gerenciamento de memória quando desenvolver aplicativos gerenciados. O gerenciamento automático de memória pode eliminar problemas comuns, como esquecer de liberar um objeto e causar um vazamento de memória ou tentar acessar a memória de um objeto que já tinha sido liberado. Esta seção descreve como o coletor de lixo aloca e libera memória.

Alocando memória

Quando você inicializa um novo processo, o runtime reserva uma região contígua de espaço de endereço para o processo. Esse espaço de endereço reservado é chamado de heap gerenciado. O heap gerenciado mantém um ponteiro para o endereço no qual o próximo objeto do heap será alocado. Inicialmente, esse ponteiro é definido como o endereço básico do heap gerenciado. Todos os tipos de referência são alocados no heap gerenciado. Quando um aplicativo cria o primeiro tipo de referência, a memória é alocada para o tipo no endereço base do heap gerenciado. Quando o aplicativo cria o próximo objeto, o coletor de lixo aloca memória para ele no espaço de endereço logo depois do primeiro objeto. Desde que exista espaço de endereço disponível, o coletor de lixo continua alocando espaço para novos objetos dessa maneira.

A alocação memória com base no heap gerenciado é mais rápida do que a alocação de memória não gerenciada. Como o runtime aloca memória para um objeto adicionando um valor a um ponteiro, ele é quase tão rápido quanto a alocação de memória com base na pilha. Além disso, como novos objetos que são alocados consecutivamente são armazenados contiguamente no heap gerenciado, um aplicativo pode acessar os objetos muito rapidamente.

Liberando memória

O mecanismo de otimização do coletor de lixo determina o melhor momento para executar uma coleta com base nas alocações que estão sendo feitas. Quando o coletor de lixo executa uma coleta, ele libera a memória dos objetos que não estão mais sendo usados pelo aplicativo. Ele determina quais objetos não estão mais sendo usados examinando as raízes do aplicativo. Cada aplicativo tem um conjunto de raízes. Cada raiz refere-se a um objeto no heap gerenciado ou é definida como nula. As raízes de um aplicativo incluem campos estáticos, variáveis locais e parâmetros na pilha de um thread, além de registros de CPU. O coletor de lixo tem acesso à lista de raízes ativas mantidas pelo tempo de execução e pelo compilador JIT (Just-In-Time). Usando essa lista, ele examina as raízes de um aplicativo e, no processo, cria um gráfico que contém todos os objetos que possam ser alcançados nas raízes.

Objetos que não estão no gráfico são inacessíveis a partir das raízes do aplicativo. O coletor de lixo considera lixo os objetos inacessíveis e liberará a memória alocada para eles. Durante uma coleta, o coletor de lixo examina o heap gerenciado, procurando os blocos de espaço de endereço ocupados por objetos inacessíveis. Na medida em que descobre cada objeto inacessível, ele usa uma função de cópia de memória para compactar os objetos acessíveis na memória, liberando os blocos de espaços de endereço alocados para objetos inacessíveis. Uma vez que a memória dos objetos acessíveis tenha sido compactada, o coletor de lixo faz as correções necessárias no ponteiro de forma que as raízes do aplicativo apontem para os objetos em seus novos locais. Ele também posiciona o ponteiro do heap gerenciado após o último objeto acessível. Observe que memória é compactada somente se uma coleta descobre um número significativo de objetos inacessíveis. Se todos os objetos no heap gerenciado sobrevivem a uma coleta, não há necessidade de compactação de memória.

Para melhorar o desempenho, o runtime aloca memória para objetos grandes em um heap separado. O coletor de lixo automaticamente libera a memória para objetos grandes. No entanto, para evitar mover objetos grandes na memória, essa memória não é compactada.

Gerações e desempenho

Para otimizar o desempenho do coletor de lixo, o heap gerenciado é dividido em três gerações: 0, 1 e 2. O algoritmo da coleta de lixo do runtime é baseado em várias generalizações que a indústria de software de computador descobriu serem verdadeiras experimentando os esquemas da coleta de lixo. Primeiro, é mais rápido compactar a memória para uma parte do heap gerenciado do que para o heap gerenciado inteiro. Em segundo lugar, objetos mais recentes terão vidas úteis menores e objetos mais antigos objetos terão vidas úteis maiores. Finalmente, objetos mais recentes tendem a se relacionar e serem acessados pelo aplicativo aproximadamente ao mesmo tempo.

O coletor de lixo do runtime armazena novos objetos na geração 0. Os objetos criados no início no tempo de vida do aplicativo que sobrevivem a coleções são promovidos e armazenados nas gerações 1 e 2. O processo de promoção do objeto será descrito mais adiante neste tópico. Como é mais rápido compactar uma parte do heap gerenciado do que o heap inteiro, esse esquema permite que o coletor de lixo libere a memória em uma geração específica em vez liberar a memória para toda a memória gerenciada a cada vez que ele executa uma coleta.

Na verdade, o coletor de lixo executa uma coleta quando a geração 0 está cheia. Se um aplicativo tentar criar um novo objeto quando a geração 0 está cheia, o coletor de lixo descobre que não existe nenhum espaço de endereço restante na geração 0 para alocar para o objeto. O coletor de lixo executa uma coleta em uma tentativa de liberar espaço de endereço na geração 0 para o objeto. O coletor de lixo inicia examinando os objetos na geração 0 em vez de todos os objetos no heap gerenciado. Isso é a abordagem mais eficiente, porque novos objetos costumam ter tempos de vida curtos e é esperado que muitos dos objetos na geração 0 não estejam mais em uso pelo aplicativo quando uma coleta é executada. Além disso, uma única coleta de geração 0 normalmente recupera memória suficiente para permitir ao aplicativo continuar criando novos objetos.

Após o coletor de lixo executar uma coleta de geração 0, ele compacta a memória para os objetos acessíveis conforme explicado anteriormente neste tópico em Liberando memória. Assim, o coletor de lixo promove esses objetos e considera isso parte da geração 1 do heap gerenciado. Como os objetos que sobrevivem a coleções tendem a ter tempos de vida mais longos, faz sentido promovê-los a uma geração mais alta. Como resultado, o coletor de lixo não tem que reexaminar os objetos em gerações 1 e 2 sempre que executa uma coleta de geração 0.

Depois que o coletor de lixo executa sua primeira coleta de geração 0 e promove os objetos atingíveis para a geração 1, ele considera o restante do heap gerenciado a geração 0. Ele continua alocando a memória para novos objetos na geração 0 até que a geração 0 esteja completa e seja necessário executar outra coleta. Nesse ponto, o mecanismo de otimização do coletor de lixo determina se é necessário examinar os objetos em gerações mais antigas. Por exemplo, se uma coleta de geração 0 não recupera memória suficiente para que o aplicativo conclua com êxito a sua tentativa de criar um novo objeto, o coletor de lixo pode executar uma coleta de geração 1 e, então, de geração 2. Se isso não recuperar memória suficiente, o coletor de lixo poderá executar uma coleta de gerações 2, 1 e 0. Depois de cada coleta, o coletor de lixo compactará os objetos atingíveis na geração 0 e os promoverá para a geração 1. Os objetos na geração 1 que sobrevivem a coleções são promovidos para a geração 2. Como o coletor de lixo só dá suporte a três gerações, os objetos na geração 2 que sobrevivem a uma coleta permanecem na geração 2 até serem considerados inacessíveis em uma coleta futura.

Liberando memória para recursos não gerenciados

Para a maioria dos objetos que seu aplicativo cria, você pode confiar no coletor de lixo para executar automaticamente as tarefas de gerenciamento de memória necessárias. Entretanto, recursos não gerenciados requerem limpeza explícita. O tipo mais comum de recursos não gerenciados é um objeto que encapsula um recurso do sistema operacional, como um identificador de arquivo, um identificador de janela ou uma conexão de rede. Embora o coletor de lixo seja capaz de acompanhar o tempo de vida de um objeto gerenciado que encapsule

um recurso não gerenciado, ele não tem conhecimento específico sobre como limpar o recurso. Quando você cria um objeto que encapsula um recurso não gerenciado, é recomendável fornecer o código necessário para limpar o recurso não gerenciado em um método público **Dispose**. Ao fornecer um método **Dispose**, você permite que usuários do seu objeto liberem, explicitamente, sua memória quando terminarem com o objeto. Quando usa um objeto que encapsula um recurso não gerenciado, você deve estar ciente de **Dispose** e chamá-lo conforme necessário. Para obter mais informações sobre a limpeza de recursos não gerenciados e um exemplo de um padrão de design para implementar **Dispose**, consulte Coleta de lixo.

Consulte também

- GC
- Coleta de lixo
- Processo de execução gerenciada

Visão geral do CLR (Common Language Runtime)

06/12/2019 • 8 minutes to read • Edit Online

O .NET Framework fornece um ambiente de tempo de execução chamado Common Language Runtime, que executa o código e fornece serviços que facilitam o processo de desenvolvimento.

Compiladores e ferramentas expõem as funcionalidades do Common Language Runtime e permitem que você grave um código que se beneficia desse ambiente de execução gerenciado. O código que você desenvolve com um compilador de linguagem que visa o runtime é chamado de código gerenciado. Ele aproveita recursos como integração em qualquer idioma, tratamento de exceções em qualquer idioma, segurança aprimorada, suporte a controle de versão e implantação, um modelo simplificado para interação entre componentes, além de serviços de depuração e criação de perfil.

> **NOTE**
> Compiladores e ferramentas podem produzir saída que o Common Language Runtime pode consumir porque o sistema de tipos, o formato dos metadados e o ambiente de tempo de execução (o sistema virtual de execução) são todos definidos por um padrão público, a especificação Common Language Infrastructure ECMA. Para obter mais informações, consulte Especificações ECMA C# e Common Language Infrastructure.

Para habilitar o runtime para fornecer serviços de código gerenciado, os compiladores de linguagem devem emitir metadados que descrevam os tipos, os membros e as referências em seu código. Os metadados são armazenados com o código. Todo arquivo PE (Portable Executable) do Common Language Runtime carregável contém metadados. O runtime usa metadados para localizar e carregar classes, dispor instâncias na memória, resolver invocações de método, gerar código nativo, impor segurança e definir limites de contexto do runtime.

O runtime identifica automaticamente o layout de objeto e gerencia referências a objetos, liberando-os quando eles não estão sendo mais usados. Objetos cujos tempos de vida são gerenciados dessa forma são chamados de dados gerenciados. A coleta de lixo elimina perdas de memória, bem como alguns outros erros de programação comuns. Se o código for gerenciado, você poderá usar dados gerenciados, dados não gerenciados ou ambos no seu aplicativo do .NET Framework. Como os compiladores de linguagem fornecem seus próprios tipos, como tipos primitivos, você talvez sem sempre saiba (ou precise saber) se os dados estão sendo gerenciados.

O Common Language Runtime facilita a criação de componentes e aplicativos cujos objetos interagem entre linguagens. Objetos gravados em linguagens diferentes podem se comunicar entre si e seus comportamentos podem estar totalmente integrados. Por exemplo, você pode definir uma classe e usar uma linguagem diferente para derivar uma classe da classe original ou chamar um método na classe original. Você também pode passar uma instância de uma classe para um método de uma classe gravado em uma linguagem diferente. Essa integração em qualquer idioma é possível porque os compiladores de linguagens e ferramentas que segmentam o runtime usam um Common Type System definido pelo runtime e seguem as regras do runtime para definir novos tipos, bem como para criar, usar, manter e associar a tipos.

Como parte de seus metadados, todos os componentes gerenciados transportam informações sobre os componentes e os recursos com base nos quais eles foram compilados. O runtime usa essas informações para garantir que o componente ou o aplicativo tenha as versões especificadas de tudo o que precisa, o que torna seu código menos suscetível à interrupção devido a alguma dependência não atendida. As informações de registro e os dados de estado não são mais armazenados no registro onde pode ser difícil estabelecer e mantê-los. Em vez de isso, as informações sobre os tipos que você define (e suas dependências) são armazenadas com o código como metadados, fazendo com que as tarefas de replicação do componente e remoção sejam muito menos complicadas.

Compiladores de linguagens e ferramentas expõem a funcionalidade do runtime da maneira como seriam úteis e

intuitivas para desenvolvedores. Isso significa que alguns recursos do runtime devem ser mais perceptíveis em um ambiente do que em outro. Como você usa o runtime depende de quais compiladores de linguagem ou ferramentas você usa. Por exemplo, se for um desenvolvedor do Visual Basic, você poderá observar que, com o Common Language Runtime, a linguagem do Visual Basic tem mais recursos orientados a objeto do que antes. O runtime oferece os seguintes benefícios:

- Melhorias de desempenho.
- A possibilidade de usar facilmente componentes desenvolvidos em outras linguagens.
- Tipos extensíveis fornecidos por uma biblioteca de classes.
- Recursos de linguagem como herança, interfaces e sobrecarga para programação orientada ao objeto.
- Suporte a threading livre explícito que permite a criação de aplicativos multithread, escalonáveis.
- Suporte ao tratamento de exceções estruturado.
- Suporte a atributos personalizados.
- Coleta de lixo.
- Uso de delegados em vez de ponteiros de função para maior segurança e segurança de tipos. Para saber mais sobre representantes, veja Common Type System.

Versões do CLR

O número de versão de .NET Framework não corresponde necessariamente ao número de versão do CLR que ele inclui. Para obter uma lista de versões de .NET Framework e suas versões correspondentes do CLR, consulte .NET Framework versões e dependências. As versões do .NET Core têm uma única versão do produto, ou seja, não há nenhuma versão CLR separada. Para obter uma lista das versões do .NET Core, consulte baixar o .NET Core.

Tópicos relacionados

CARGO	DESCRIÇÃO
Processo de execução gerenciada	Descreve as etapas obrigatórias para usufruir o Common Language Runtime.
Gerenciamento Automático de Memória	Descreve como o coletor de lixo aloca e libera memória.
Visão geral do .NET Framework	Descreve os conceitos-chave do .NET Framework, como Common Type System, interoperabilidade entre linguagens, execução gerenciada, domínios de aplicativos e assemblies.
Common Type System	Descreve como os tipos são declarados, usados e gerenciados no runtime para dar suporte à integração entre linguagens.

Componentes de independência de linguagem e componentes independentes da linguagem

04/11/2019 • 106 minutes to read • Edit Online

O .NET é independente de linguagem. Isso significa que, como desenvolvedor, você pode desenvolver em uma das muitas linguagens direcionadas às implementações do .NET, como C#, F# e Visual Basic. É possível acessar tipos e membros de bibliotecas de classes desenvolvidas para implementações do .NET sem que seja necessário conhecer a linguagem em que foram originalmente escritas e sem precisar seguir as convenções da linguagem original. Se você for um desenvolvedor de componentes, o componente poderá ser acessado por qualquer aplicativo .NET, independentemente da linguagem.

> **NOTE**
> A primeira parte deste artigo descreve como criar componentes independentes de linguagem, ou seja, componentes que podem ser consumidos por aplicativos gravados em qualquer linguagem. Você também pode criar um único componente ou aplicativo de código-fonte gravado em várias linguagens; consulte Interoperabilidade em qualquer idioma na segunda parte deste artigo.

Para interagir completamente com outros objetos gravados em qualquer linguagem, os objetos devem expor aos chamadores somente os recursos comuns a todas as linguagens. Esse conjunto comum de recursos é definido pela CLS (Common Language Specification), que é um conjunto de regras que se aplicam aos assemblies gerados. A Common Language Specification é definida na Partição I, cláusulas 7 a 11 do Padrão ECMA-335: Common Language Infrastructure.

Se o componente estiver de acordo com a Common Language Specification, ele será compatível com a CLS e poderá ser acessado pelo código em assemblies gravados em qualquer linguagem de programação que dê suporte a CLS. É possível determinar se o componente está de acordo com a Common Language Specification no tempo de compilação aplicando o atributo CLSCompliantAttribute ao código-fonte. Para obter mais informações, consulte o atributo CLSCompliantAttribute.

Neste artigo:

Regras de conformidade com CLS

Esta seção discute as regras para criar um componente compatível com CLS. Para obter uma lista completa de regras, consulte Partição I, Cláusula 11 do Padrão ECMA-335: Common Language Infrastructure.

> **NOTE**
> A Common Language Specification aborda cada regra de conformidade com CLS à medida que se aplica a consumidores (desenvolvedores que estão acessando programaticamente um componente compatível com CLS), estruturas (desenvolvedores que estão usando um compilador de linguagem para criar bibliotecas compatíveis com CLS) e extensores (desenvolvedores que estão criando uma ferramenta, como um compilador de linguagem ou um analisador de código que cria componentes compatíveis com CLS). Este artigo enfoca as regras que se aplicam às estruturas. Entretanto, algumas das regras que se aplicam a extensores também podem ser aplicadas a assemblies criados usando Reflection.Emit.

Para criar um componente independente de linguagem, você só precisa aplicar as regras de compatibilidade com CLS à interface pública do componente. A implementação privada não precisa estar de acordo com a especificação.

> **IMPORTANT**
> As regras de conformidade com CLS só se aplicam à interface pública de um componente e não à implementação privada.

Por exemplo, inteiros sem sinal que não sejam Byte não são compatíveis com CLS. Como a classe `Person` no exemplo a seguir expõe uma propriedade `Age` de tipo UInt16, o código a seguir exibe um aviso do compilador.

```
using System;

[assembly: CLSCompliant(true)]

public class Person
{
   private UInt16 personAge = 0;

   public UInt16 Age
   { get { return personAge; } }
}
// The attempt to compile the example displays the following compiler warning:
//    Public1.cs(10,18): warning CS3003: Type of 'Person.Age' is not CLS-compliant
```

```
<Assembly: CLSCompliant(True)>

Public Class Person
   Private personAge As UInt16

   Public ReadOnly Property Age As UInt16
      Get
         Return personAge
      End Get
   End Property
End Class
' The attempt to compile the example displays the following compiler warning:
'    Public1.vb(9) : warning BC40027: Return type of function 'Age' is not CLS-compliant.
'
'       Public ReadOnly Property Age As UInt16
'                                ~~~
```

É possível tornar a classe Pessoa compatível com CLS alterando o tipo de propriedade `Age` de `UInt16` para Int16, que é um inteiro com sinal de 16 bits compatível com CLS. Não é necessário alterar o tipo do campo `personAge` privado.

```
using System;

[assembly: CLSCompliant(true)]

public class Person
{
   private Int16 personAge = 0;

   public Int16 Age
   { get { return personAge; } }
}
```

```
<Assembly: CLSCompliant(True)>

Public Class Person
   Private personAge As UInt16

   Public ReadOnly Property Age As Int16
      Get
         Return CType(personAge, Int16)
      End Get
   End Property
End Class
```

A interface pública de uma biblioteca consiste no seguinte:

- Definições de classes públicas.
- Definições dos membros públicos de classes públicas e definições de membros acessíveis para classes derivadas (ou seja, membros protegidos).
- Parâmetros e tipos de retorno de métodos públicos de classes públicas e parâmetros e tipos de retorno de métodos acessíveis para classes derivadas.

As regras de conformidade com CLS estão listadas na tabela a seguir. O texto das regras é tirado literalmente do Padrão ECMA-335: Common Language Infrastructure, com direitos autorais de 2012 da Ecma International. Informações mais detalhadas sobre essas regras são encontradas nas seções a seguir.

CATEGORIA	CONSULTE	REGRA	NÚMERO DA REGRA
Acessibilidade	Acessibilidade de membro	A acessibilidade não deverá ser alterada ao substituir métodos herdados, exceto na substituição de um método herdado de um assembly diferente com acessibilidade `family-or-assembly`. Nesse caso, a substituição deverá ter a acessibilidade `family`.	10
Acessibilidade	Acessibilidade de membro	A visibilidade e a acessibilidade de tipos e membros deverão ser de tal forma que os tipos na assinatura de qualquer membro sejam visíveis e acessíveis sempre que o próprio membro estiver visível e acessível. Por exemplo, um método público visível fora do assembly não deve ter um argumento cujo tipo seja visível somente dentro do assembly. A visibilidade e a acessibilidade dos tipos que compõem um tipo genérico instanciado usado na assinatura de qualquer membro deverão estar visíveis e acessíveis sempre que o próprio membro estiver visível e acessível. Por exemplo, um tipo genérico instanciado presente na assinatura de um membro visível fora do assembly não deverá ter um argumento genérico cujo tipo seja visível somente dentro do assembly.	12
Matrizes	Matrizes	As matrizes deverão ter elementos com um tipo compatível com CLS e todas as dimensões da matriz deverão ter limites inferiores iguais a zero. Se o item for uma matriz, o tipo do elemento da matriz será necessário para diferenciar as sobrecargas. Quando a sobrecarga é baseada em dois ou mais tipos de matriz, os tipos de elemento deverão ser chamados de tipos.	16

CATEGORIA	CONSULTE	REGRA	NÚMERO DA REGRA
Atributos	Atributos	Os atributos deverão ser do tipo System.Attribute ou de um tipo herdado dele.	41
Atributos	Atributos	A CLS só permite um subconjunto das codificações de atributos personalizados. Os únicos tipos que devem ser exibidos nessas codificações são (consulte a Partição IV) System.Type, System.String, System.Char, System.Boolean, System.Byte, System.Int16, System.Int32, System.Int64, System.Single, System.Double e qualquer tipo de enumeração com base em um tipo de inteiro básico compatível com CLS.	34
Atributos	Atributos	A CLS não permite modificadores obrigatórios visíveis publicamente (`modreq` , consulte a Partição II), mas permite modificadores opcionais (`modopt` , consulte a Partição II) que ela não entende.	35
Construtores	Construtores	Um construtor de objeto deverá chamar um construtor de instância de sua classe base antes de qualquer acesso aos dados da instância herdados. (Isso não se aplica a tipos de valor, que não precisam ter construtores.)	21
Construtores	Construtores	Um construtor de objeto não deverá ser chamado, exceto como parte da criação de um objeto e um objeto não deve ser inicializado duas vezes.	22
Enumerações	Enumerações	O tipo subjacente de um enum deverá ser um tipo de inteiro CLS interno, o nome do campo deverá ser "value__", e esse campo deverá ser marcado como `RTSpecialName` .	7

CATEGORIA	CONSULTE	REGRA	NÚMERO DA REGRA
Enumerações	Enumerações	Há dois tipos diferentes de enums, indicados pela presença ou pela ausência do atributo personalizado System.FlagsAttribute (consulte a Biblioteca da Partição IV). Um representa valores de inteiro nomeados; o outro representa sinalizadores de bit nomeados que podem ser combinados para gerar um valor sem nome. O valor de um `enum` não está limitado aos valores especificados.	8
Enumerações	Enumerações	Campos estáticos de literais de um enum deverão ter o tipo do próprio enum.	9
Eventos	Eventos	Os métodos que implementam um evento deverão ser marcados como `SpecialName` nos metadados.	29
Eventos	Eventos	A acessibilidade de um evento e de seus acessadores deverá ser idêntica.	30
Eventos	Eventos	Os métodos `add` e `remove` de um evento deverão estar presentes ou ausentes.	31
Eventos	Eventos	Os métodos `add` e `remove` de um evento deverão utilizar um parâmetro cada um, cujo tipo defina o tipo do evento e ele deverá ser derivado de System.Delegate.	32
Eventos	Eventos	Os eventos deverão respeitar um padrão de nomenclatura específico. O atributo SpecialName mencionado na regra 29 da CLS deverá ser ignorado em comparações de nome apropriadas e respeitar as regras do identificador.	33

CATEGORIA	CONSULTE	REGRA	NÚMERO DA REGRA
Exceções	Exceções	Os atributos acionados deverão ser do tipo System.Exception ou de um tipo herdado dele. Mesmo assim, os métodos compatíveis com CLS não precisam bloquear a propagação de outros tipos de exceção.	40
Geral	Regras de conformidade da CLS	As regras CLS só se aplicam a essas partes de um tipo acessíveis ou visíveis fora do assembly de definição.	1
Geral	Regras de conformidade da CLS	Membros de tipos incompatíveis com CLS não deverão ser marcados como compatíveis com CLS.	2
Genéricos	Tipos e membros genéricos	Os tipos aninhados deverão ter, pelo menos, tantos parâmetros genéricos quanto o tipo delimitador. Os parâmetros genéricos em um tipo aninhado correspondem, por posição, aos parâmetros genéricos no tipo delimitador.	42
Genéricos	Tipos e membros genéricos	O nome de um tipo genérico deverá codificar o número de parâmetros de tipo declarados no tipo não aninhado ou recém-introduzidos no tipo, se aninhado, de acordo com as regras definidas anteriormente.	43
Genéricos	Tipos e membros genéricos	Um tipo genérico deverá redeclarar restrições suficientes para assegurar que todas as restrições no tipo base ou nas interfaces sejam atendidas pelas restrições de tipo genérico.	44
Genéricos	Tipos e membros genéricos	Tipos usados como restrições em parâmetros genéricos deverão ser compatíveis com CLS.	45

CATEGORIA	CONSULTE	REGRA	NÚMERO DA REGRA
Genéricos	Tipos e membros genéricos	A visibilidade e a acessibilidade de membros (incluindo tipos aninhados) em um tipo genérico instanciado deverão ser consideradas no escopo da instanciação específica, em vez da declaração de tipo genérico como um todo. Supondo isso, as regras de visibilidade e acessibilidade da regra 12 da CLS continuam sendo aplicáveis.	46
Genéricos	Tipos e membros genéricos	Para cada método genérico abstrato ou virtual, deverá haver uma implementação concreta (não abstrata) padrão	47
Interfaces	Interfaces	As interfaces em conformidade com CLS não deverão exigir a definição de métodos incompatíveis com CLS para implementá-los.	18
Interfaces	Interfaces	As interfaces compatíveis com CLS não deverão definir métodos estáticos, nem devem definir campos.	19
Membros	Membros de tipo em geral	Campos e métodos estáticos globais não são compatíveis com CLS.	36
Membros	--	O valor de um estático literal é especificado por meio do uso de metadados de inicialização do campo. Um literal compatível com CLS deve ter um valor especificado em metadados de inicialização de campo que sejam exatamente do mesmo tipo que o literal (ou do tipo subjacente, se esse literal for um `enum`).	13
Membros	Membros de tipo em geral	A restrição vararg não faz parte da CLS e a única convenção de chamada com suporte pela CLS é a convenção de chamada gerenciada padrão.	15

CATEGORIA	CONSULTE	REGRA	NÚMERO DA REGRA
Convenções de nomenclatura	Convenções de nomenclatura	Os assemblies deverão seguir o Anexo 7 do Relatório Técnico 15 do Padrão Unicode 3.0 que controla o conjunto de caracteres permitidos para iniciar e serem incluídos em identificadores, disponíveis online em Formulários de Normalização de Unicode. Os identificadores deverão estar no formato canônico definido pelo Formulário C de Normalização de Unicode. Para fins de CLS, dois identificadores serão iguais se os mapeamentos em minúsculas (conforme especificado pelos mapeamentos em minúsculas um para um, insensíveis a localidade Unicode) forem os mesmos. Ou seja, para dois identificadores serem considerados diferentes na CLS, eles deverão ser diferentes além de apenas maiúsculas e minúsculas. No entanto, para substituir uma definição herdada, a CLI exige que a codificação precisa da declaração original seja usada.	4
Sobrecarga	Convenções de nomenclatura	Todos os nomes introduzidos em um escopo compatível com CLS deverão ser independentes e distintos do tipo, exceto quando os nomes forem idênticos e resolvidos por meio da sobrecarga. Ou seja, embora o CTS permita que um tipo single use o mesmo nome para um método e um campo, a CLS não permite.	5

CATEGORIA	CONSULTE	REGRA	NÚMERO DA REGRA
Sobrecarga	Convenções de nomenclatura	Campos e tipos aninhados deverão ser diferenciados apenas por comparação de identificador, mesmo que o CTS permita que assinaturas diferentes sejam distinguidas. Métodos, propriedades e eventos com o mesmo nome (por comparação de identificador) deverão ser diferentes além apenas do tipo de retorno, exceto conforme especificado na Regra 39 da CLS	6
Sobrecarga	Sobrecargas	Somente propriedades e métodos podem ser sobrecarregados.	37
Sobrecarga	Sobrecargas	As propriedades e os métodos só podem ser sobrecarregados com base no número e nos tipos de seus parâmetros, exceto os operadores de conversão chamados `op_Implicit` e `op_Explicit`, que também podem ser sobrecarregados com base no tipo de retorno.	38
Sobrecarga	--	Se dois ou mais métodos em conformidade com CLS declarados em um tipo tiverem o mesmo nome e, para um conjunto específico de instanciações de tipo, tiverem os mesmos tipos de parâmetro e retorno, esses métodos deverão ser semanticamente equivalentes nessas instanciações de tipo.	48
Propriedades	Propriedades	Os métodos que implementam os métodos getter e setter de uma propriedade deverão ser marcados como `SpecialName` nos metadados.	24
Propriedades	Propriedades	Os acessadores de uma propriedade deverão ser todos estáticos, virtuais ou de instância.	26

CATEGORIA	CONSULTE	REGRA	NÚMERO DA REGRA
Propriedades	Propriedades	O tipo de uma propriedade deverá ser o tipo de retorno do getter e o tipo do último argumento do setter. Os tipos dos parâmetros da propriedade deverão ser os tipos dos parâmetros do getter e os tipos de todos os parâmetros, menos o parâmetro final do setter. Todos esses tipos deverão ser compatíveis com CLS e não deverão ser ponteiros gerenciados (ou seja, não deverão ser passados por referência).	27
Propriedades	Propriedades	As propriedades deverão seguir um padrão de nomenclatura específico. O atributo `SpecialName` mencionado na regra 24 da CLS deverá ser ignorado em comparações de nome apropriadas e respeitar as regras do identificador. Uma propriedade deverá ter um método getter, um método setter ou ambos.	28
Conversão de tipos	Conversão de tipos	Se op_Implicit ou op_Explicit for fornecido, um meio alternativo de coerção deverá ser fornecido.	39
Tipos	Tipos e assinaturas de membro de tipo	Tipos de valor demarcado não são compatíveis com CLS.	3
Tipos	Tipos e assinaturas de membro de tipo	Todos os tipos exibidos em uma assinatura deverão ser compatíveis com CLS. Todos os tipos que compõem um tipo genérico instanciado deverão ser compatíveis com CLS.	11
Tipos	Tipos e assinaturas de membro de tipo	Referências com tipo não são compatíveis com CLS.	14
Tipos	Tipos e assinaturas de membro de tipo	Tipos de ponteiro não gerenciados não são compatíveis com CLS.	17

CATEGORIA	CONSULTE	REGRA	NÚMERO DA REGRA
Tipos	Tipos e assinaturas de membro de tipo	Classes compatíveis com CLS, tipos de valor e interfaces não deverão exigir a implementação de membros incompatíveis com CLS	20
Tipos	Tipos e assinaturas de membro de tipo	System. Object é compatível com CLS. Qualquer outra classe compatível com CLS deverá herdar de uma classe compatível com CLS.	23

Tipos e assinaturas de membro de tipo

O tipo System.Object é compatível com CLS e é o tipo base de todos os tipos de objeto no sistema de tipos do .NET Framework. A herança no .NET Framework é implícita (por exemplo, a classe String herda implicitamente da classe `Object`) ou explícita (por exemplo, a classe CultureNotFoundException herda explicitamente da classe ArgumentException, que herda explicitamente da classe Exception). Para que um tipo derivado seja compatível com CLS, seu tipo base também deverá ser compatível com CLS.

O exemplo a seguir mostra um tipo derivado cujo tipo de base não é compatível com CLS. Ele define uma classe `Counter` base que usa um inteiro de 32 bits sem sinal como um contador. Como a classe fornece funcionalidade de contador encapsulando um inteiro sem sinal, a classe é marcada como não compatível com CLS. Assim, uma classe derivada, `NonZeroCounter`, também não é compatível com CLS.

```
using System;

[assembly: CLSCompliant(true)]

[CLSCompliant(false)]
public class Counter
{
   UInt32 ctr;

   public Counter()
   {
      ctr = 0;
   }

   protected Counter(UInt32 ctr)
   {
      this.ctr = ctr;
   }

   public override string ToString()
   {
      return $"{ctr}). ";
   }

   public UInt32 Value
   {
      get { return ctr; }
   }

   public void Increment()
   {
      ctr += (uint) 1;
   }
}

public class NonZeroCounter : Counter
{
   public NonZeroCounter(int startIndex) : this((uint) startIndex)
   {
   }

   private NonZeroCounter(UInt32 startIndex) : base(startIndex)
   {
   }
}
// Compilation produces a compiler warning like the following:
//    Type3.cs(37,14): warning CS3009: 'NonZeroCounter': base type 'Counter' is not
//            CLS-compliant
//    Type3.cs(7,14): (Location of symbol related to previous warning)
```

```
<Assembly: CLSCompliant(True)>

<CLSCompliant(False)> _
Public Class Counter
   Dim ctr As UInt32

   Public Sub New
      ctr = 0
   End Sub

   Protected Sub New(ctr As UInt32)
      ctr = ctr
   End Sub

   Public Overrides Function ToString() As String
      Return $"{ctr}). "
   End Function

   Public ReadOnly Property Value As UInt32
      Get
         Return ctr
      End Get
   End Property

   Public Sub Increment()
      ctr += CType(1, UInt32)
   End Sub
End Class

Public Class NonZeroCounter : Inherits Counter
   Public Sub New(startIndex As Integer)
      MyClass.New(CType(startIndex, UInt32))
   End Sub

   Private Sub New(startIndex As UInt32)
      MyBase.New(CType(startIndex, UInt32))
   End Sub
End Class
' Compilation produces a compiler warning like the following:
'    Type3.vb(34) : warning BC40026: 'NonZeroCounter' is not CLS-compliant
'    because it derives from 'Counter', which is not CLS-compliant.
'
'    Public Class NonZeroCounter : Inherits Counter
'                 ~~~~~~~~~~~~~~
```

Todos os tipos exibidos em assinaturas de membro, incluindo um tipo de retorno de método ou um tipo de propriedade, devem ser compatíveis com CLS. Além disso, para tipos genéricos:

- Todos os tipos que compõe um tipo genérico instanciado devem ser compatíveis com CLS.
- Todos os tipos usados como restrições em parâmetros genéricos devem ser compatíveis com CLS.

O Common Type System do .NET inclui vários tipos internos com suporte diretamente com o Common Language Runtime e codificados especialmente nos metadados de um assembly. Desses tipos intrínsecos, os tipos listados na tabela a seguir são compatíveis com CLS.

TIPO COMPATÍVEL COM CLS	DESCRIÇÃO
Byte	Inteiro sem sinal de 8 bits
Int16	Inteiro com sinal de 16 bits

TIPO COMPATÍVEL COM CLS	DESCRIÇÃO
Int32	Inteiro com sinal de 32 bits
Int64	Inteiro com sinal de 64 bits
Simples	Valor do ponto flutuante de precisão simples
Duplo	Valor do ponto flutuante de precisão dupla
Booliano	tipo de valor verdadeiro ou falso
Char	unidade de código codificado UTF-16
Decimal	Número decimal de ponto não flutuante
IntPtr	Ponteiro ou identificador de um tamanho definido por plataforma
Cadeia de caracteres	Coleção de zero, um ou mais objetos Char

Os tipos intrínsecos listados na tabela a seguir não são compatíveis com CLS.

TIPO NÃO COMPATÍVEL	DESCRIÇÃO	ALTERNATIVA COMPATÍVEL COM CLS
SByte	Tipo de dados inteiro com sinal de 8 bits	Int16
UInt16	Inteiro sem sinal de 16 bits	Int32
UInt32	Inteiro sem sinal de 32 bits	Int64
UInt64	Inteiro sem sinal de 64 bits	Int64 (pode estourar), BigInteger, ouDouble
UIntPtr	Ponteiro ou identificador sem sinal	IntPtr

A biblioteca de classes .NET Framework ou qualquer outra biblioteca de classes pode incluir outros tipos que não sejam compatíveis com CLS; por exemplo:

- Tipos de valor demarcado. O exemplo de C# a seguir cria uma classe que tem uma propriedade pública do tipo `int*` named `Value`. Como um `int*` é um tipo de valor demarcado, o compilador o sinaliza como sem conformidade com CLS.

```
using System;

[assembly:CLSCompliant(true)]

public unsafe class TestClass
{
   private int* val;

   public TestClass(int number)
   {
      val = (int*) number;
   }

   public int* Value {
      get { return val; }
   }
}
// The compiler generates the following output when compiling this example:
//        warning CS3003: Type of 'TestClass.Value' is not CLS-compliant
```

- Referências de tipo, que são constructos especiais que contêm referência a um objeto e referência a um tipo.

Se um tipo não for compatível com CLS, você deverá aplicar o atributo CLSCompliantAttribute com um parâmetro *isCompliant* com um valor de `false` a ele. Para obter mais informações, consulte a seção CLSCompliantAttribute attribute.

O exemplo a seguir ilustra o problema de conformidade com CLS em uma assinatura do método e em uma instanciação de tipo genérico. Ele define uma classe `InvoiceItem` com uma propriedade do tipo UInt32, uma propriedade do tipo Nullable(Of UInt32) e um construtor com parâmetros do tipo `UInt32` e `Nullable(Of UInt32)`. Você recebe quatro avisos do compilador ao tentar de compilar esse exemplo.

```
using System;

[assembly: CLSCompliant(true)]

public class InvoiceItem
{
   private uint invId = 0;
   private uint itemId = 0;
   private Nullable<uint> qty;

   public InvoiceItem(uint sku, Nullable<uint> quantity)
   {
      itemId = sku;
      qty = quantity;
   }

   public Nullable<uint> Quantity
   {
      get { return qty; }
      set { qty = value; }
   }

   public uint InvoiceId
   {
      get { return invId; }
      set { invId = value; }
   }
}
// The attempt to compile the example displays the following output:
//    Type1.cs(13,23): warning CS3001: Argument type 'uint' is not CLS-compliant
//    Type1.cs(13,33): warning CS3001: Argument type 'uint?' is not CLS-compliant
//    Type1.cs(19,26): warning CS3003: Type of 'InvoiceItem.Quantity' is not CLS-compliant
//    Type1.cs(25,16): warning CS3003: Type of 'InvoiceItem.InvoiceId' is not CLS-compliant
```

```
<Assembly: CLSCompliant(True)>

Public Class InvoiceItem

   Private invId As UInteger = 0
   Private itemId As UInteger = 0
   Private qty AS Nullable(Of UInteger)

   Public Sub New(sku As UInteger, quantity As Nullable(Of UInteger))
      itemId = sku
      qty = quantity
   End Sub

   Public Property Quantity As Nullable(Of UInteger)
      Get
         Return qty
      End Get
      Set
         qty = value
      End Set
   End Property

   Public Property InvoiceId As UInteger
      Get
         Return invId
      End Get
      Set
         invId = value
      End Set
   End Property
End Class
' The attempt to compile the example displays output similar to the following:
'    Type1.vb(13) : warning BC40028: Type of parameter 'sku' is not CLS-compliant.
'
'       Public Sub New(sku As UInteger, quantity As Nullable(Of UInteger))
'                      ~~~
'    Type1.vb(13) : warning BC40041: Type 'UInteger' is not CLS-compliant.
'
'       Public Sub New(sku As UInteger, quantity As Nullable(Of UInteger))
'                                                             ~~~~~~~~
'    Type1.vb(18) : warning BC40041: Type 'UInteger' is not CLS-compliant.
'
'       Public Property Quantity As Nullable(Of UInteger)
'                                               ~~~~~~~~
'    Type1.vb(27) : warning BC40027: Return type of function 'InvoiceId' is not CLS-compliant.
'
'       Public Property InvoiceId As UInteger
```

Para eliminar os avisos do compilador, substitua os tipos não compatíveis com CLS na interface pública `InvoiceItem` por tipos compatíveis:

```
using System;

[assembly: CLSCompliant(true)]

public class InvoiceItem
{
   private uint invId = 0;
   private uint itemId = 0;
   private Nullable<int> qty;

   public InvoiceItem(int sku, Nullable<int> quantity)
   {
      if (sku <= 0)
         throw new ArgumentOutOfRangeException("The item number is zero or negative.");
      itemId = (uint) sku;

      qty = quantity;
   }

   public Nullable<int> Quantity
   {
      get { return qty; }
      set { qty = value; }
   }

   public int InvoiceId
   {
      get { return (int) invId; }
      set {
         if (value <= 0)
            throw new ArgumentOutOfRangeException("The invoice number is zero or negative.");
         invId = (uint) value; }
   }
}
```

```
Assembly: CLSCompliant(True)>

Public Class InvoiceItem

   Private invId As UInteger = 0
   Private itemId As UInteger = 0
   Private qty AS Nullable(Of Integer)

   Public Sub New(sku As Integer, quantity As Nullable(Of Integer))
      If sku <= 0 Then
         Throw New ArgumentOutOfRangeException("The item number is zero or negative.")
      End If
      itemId = CUInt(sku)
      qty = quantity
   End Sub

   Public Property Quantity As Nullable(Of Integer)
      Get
         Return qty
      End Get
      Set
         qty = value
      End Set
   End Property

   Public Property InvoiceId As Integer
      Get
         Return CInt(invId)
      End Get
      Set
         invId = CUInt(value)
      End Set
   End Property
End Class
```

Além dos tipos específicos listados, algumas categorias de tipos não são compatíveis com CLS. Entre eles estão tipos de ponteiro não gerenciados e tipos de ponteiro de função. O exemplo a seguir gera um aviso do compilador porque ele usa um ponteiro para um inteiro a fim de criar uma matriz de inteiros.

```
using System;

[assembly: CLSCompliant(true)]

public class ArrayHelper
{
   unsafe public static Array CreateInstance(Type type, int* ptr, int items)
   {
      Array arr = Array.CreateInstance(type, items);
      int* addr = ptr;
      for (int ctr = 0; ctr < items; ctr++) {
          int value = *addr;
          arr.SetValue(value, ctr);
          addr++;
      }
      return arr;
   }
}
// The attempt to compile this example displays the following output:
//    UnmanagedPtr1.cs(8,57): warning CS3001: Argument type 'int*' is not CLS-compliant
```

```
using System;

[assembly: CLSCompliant(true)]

public class ArrayHelper
{
   unsafe public static Array CreateInstance(Type type, int* ptr, int items)
   {
      Array arr = Array.CreateInstance(type, items);
      int* addr = ptr;
      for (int ctr = 0; ctr < items; ctr++) {
          int value = *addr;
          arr.SetValue(value, ctr);
          addr++;
      }
      return arr;
   }
}
// The attempt to compile this example displays the following output:
//    UnmanagedPtr1.cs(8,57): warning CS3001: Argument type 'int*' is not CLS-compliant
```

Para classes abstratas compatíveis com CLS (ou seja, classes marcadas como `abstract` no C#), todos os membros da classe também devem ser compatíveis com CLS.

Convenções de nomenclatura

Como algumas linguagens de programação não diferenciam maiúsculas de minúsculas, os identificadores (como nomes de namespaces, tipos e membros) devem se diferenciar além de maiúsculas e minúsculas. Dois identificadores serão considerados equivalentes se seus mapeamentos em minúsculas forem os mesmos. O exemplo do C# a seguir define duas classes públicas, `Person` e `person`. Como elas são diferentes apenas em maiúsculas e minúsculas, o compilador do C# as sinaliza como não compatíveis com CLS.

```
using System;

[assembly: CLSCompliant(true)]

public class Person : person
{

}

public class person
{

}
// Compilation produces a compiler warning like the following:
//    Naming1.cs(11,14): warning CS3005: Identifier 'person' differing
//                       only in case is not CLS-compliant
//    Naming1.cs(6,14): (Location of symbol related to previous warning)
```

Identificadores de linguagem de programação, como nomes de namespaces, tipos e membros, devem estar de acordo com o Padrão Unicode 3.0, Relatório Técnico 15, Anexo 7. Isso significa que:

- O primeiro caractere de um identificador pode ser qualquer letra maiúscula Unicode, letra minúscula, letra maiúscula do título, letra modificadora, outra letra ou o número da letra. Para obter informações sobre categorias de caractere Unicode, consulte a enumeração System.Globalization.UnicodeCategory.
- Os caracteres subsequentes podem ser de qualquer uma das categorias, como o primeiro caractere e também podem incluir marcas sem espaçamento, marcas que combinam espaçamento, números decimais, pontuações de conector e códigos de formatação.

Antes de comparar identificadores, você deve filtrar códigos de formatação e converter os identificadores em Formulário C de Normalização de Unicode, porque um caractere único pode ser representado por várias unidades de código codificadas em UTF-16. As sequências de caracteres que produzem as mesmas unidades de código no Formulário C de Normalização de Unicode não são compatíveis com CLS. O exemplo a seguir define uma propriedade chamada `Å`, que consiste no caractere ANGSTROM SIGN (U+212B) e uma segunda propriedade chamada `Å`, que consiste no caractere LATIN CAPITAL LETTER A WITH RING ABOVE (U+00C5). O compilador do C# sinaliza o código-fonte como não compatível com CLS.

```
public class Size
{
   private double a1;
   private double a2;

   public double Å
   {
       get { return a1; }
       set { a1 = value; }
   }

   public double Å
   {
       get { return a2; }
       set { a2 = value; }
   }
}
// Compilation produces a compiler warning like the following:
//    Naming2a.cs(16,18): warning CS3005: Identifier 'Size.Å' differing only in case is not
//            CLS-compliant
//    Naming2a.cs(10,18): (Location of symbol related to previous warning)
//    Naming2a.cs(18,8): warning CS3005: Identifier 'Size.Å.get' differing only in case is not
//            CLS-compliant
//    Naming2a.cs(12,8): (Location of symbol related to previous warning)
//    Naming2a.cs(19,8): warning CS3005: Identifier 'Size.Å.set' differing only in case is not
//            CLS-compliant
//    Naming2a.cs(13,8): (Location of symbol related to previous warning)
```

```
<Assembly: CLSCompliant(True)>
Public Class Size
   Private a1 As Double
   Private a2 As Double

   Public Property Å As Double
       Get
          Return a1
       End Get
       Set
          a1 = value
       End Set
   End Property

   Public Property Å As Double
       Get
          Return a2
       End Get
       Set
          a2 = value
       End Set
   End Property
End Class
' Compilation produces a compiler warning like the following:
'    Naming1.vb(9) : error BC30269: 'Public Property Å As Double' has multiple definitions
'     with identical signatures.
'
'       Public Property Å As Double
'                       ~
```

Os nomes de membro em um escopo específico (como os namespaces em um assembly, os tipos em um namespace ou os membros em um tipo) devem ser exclusivos, exceto os nomes resolvidos por meio de sobrecarga. Esse requisito é mais rígido do que o do Common Type System, que permite que vários membros em um escopo tenham nomes idênticos desde que sejam tipos diferentes de membros (por exemplo, um é um método e outro é um campo). Em particular, para membros de tipo:

- Campos e tipos aninhados são diferenciados apenas por nome.
- Métodos, propriedades e eventos que tenham o mesmo nome devem ser diferentes além apenas do tipo de retorno.

O exemplo a seguir ilustra o requisito de que nomes de membros devem ser exclusivos dentro de seu escopo. Ele define uma classe chamada `Converter` que inclui quatro membros chamados `Conversion`. Três são métodos e um é uma propriedade. O método que inclui um parâmetro `Int64` tem um nome exclusivo, mas os dois métodos com um parâmetro `Int32` não têm, porque o valor retornado não é considerado parte da assinatura de um membro. A propriedade `Conversion` também viola esse requisito porque as propriedades não podem ter o mesmo nome dos métodos sobrecarregados.

```
using System;

[assembly: CLSCompliant(true)]

public class Converter
{
   public double Conversion(int number)
   {
      return (double) number;
   }

   public float Conversion(int number)
   {
      return (float) number;
   }

   public double Conversion(long number)
   {
      return (double) number;
   }

   public bool Conversion
   {
      get { return true; }
   }
}
// Compilation produces a compiler error like the following:
//    Naming3.cs(13,17): error CS0111: Type 'Converter' already defines a member called
//             'Conversion' with the same parameter types
//    Naming3.cs(8,18): (Location of symbol related to previous error)
//    Naming3.cs(23,16): error CS0102: The type 'Converter' already contains a definition for
//             'Conversion'
//    Naming3.cs(8,18): (Location of symbol related to previous error)
```

```
<Assembly: CLSCompliant(True)>

Public Class Converter
   Public Function Conversion(number As Integer) As Double
      Return CDbl(number)
   End Function

   Public Function Conversion(number As Integer) As Single
      Return CSng(number)
   End Function

   Public Function Conversion(number As Long) As Double
      Return CDbl(number)
   End Function

   Public ReadOnly Property Conversion As Boolean
      Get
         Return True
      End Get
   End Property
End Class
' Compilation produces a compiler error like the following:
'    Naming3.vb(8) : error BC30301: 'Public Function Conversion(number As Integer) As Double'
'                    and 'Public Function Conversion(number As Integer) As Single' cannot
'                    overload each other because they differ only by return types.
'
'       Public Function Conversion(number As Integer) As Double
'                       ~~~~~~~~~~
'    Naming3.vb(20) : error BC30260: 'Conversion' is already declared as 'Public Function
'                     Conversion(number As Integer) As Single' in this class.
'
'       Public ReadOnly Property Conversion As Boolean
'                                ~~~~~~~~~~
```

Linguagens individuais incluem palavras-chave exclusivas, portanto, linguagens que apontam para o Common Language Runtime também devem oferecer algum mecanismo para fazer referência a identificadores (como nomes de tipo) que coincidam com palavras-chave. Por exemplo, `case` é uma palavra-chave no C# e no Visual Basic. No entanto, o exemplo do Visual Basic a seguir pode remover a ambiguidade de uma classe chamada `case` da palavra-chave `case`, usando chaves de abertura e fechamento. Caso contrário, o exemplo produziria a mensagem de erro "A palavra-chave não é válida como um identificador", e não seria compilado.

```
Public Class [case]
   Private _id As Guid
   Private name As String

   Public Sub New(name As String)
      _id = Guid.NewGuid()
      Me.name = name
   End Sub

   Public ReadOnly Property ClientName As String
      Get
         Return name
      End Get
   End Property
End Class
```

O exemplo do C# a seguir pode criar uma instância da classe `case` usando o símbolo @ para remover a ambiguidade do identificador da palavra-chave da linguagem. Sem ele, o compilador do C# exibiria duas mensagens de erro, "Tipo esperado" e "'Maiúsculas e minúsculas' do termo de expressão inválido".

```
using System;

public class Example
{
   public static void Main()
   {
      @case c = new @case("John");
      Console.WriteLine(c.ClientName);
   }
}
```

Conversão de tipos

A Common Language Specification define dois operadores de conversão:

- `op_Implicit`, que é usado para conversões de ampliação que não resultam em perda de dados ou precisão. Por exemplo, a estrutura Decimal inclui um operador `op_Implicit` sobrecarregado para converter valores de tipos integrais e valores Char em valores `Decimal`.
- `op_Explicit`, que é usado para conversões de redução que possam resultar em perda de magnitude (um valor é convertido em um valor com um intervalo menor) ou precisão. Por exemplo, a estrutura `Decimal` inclui um operador `op_Explicit` sobrecarregado para converter valores Double e Single em `Decimal` e para converter valores `Decimal` em valores inteiros, `Double`, `Single` e `Char`.

No entanto, nem todas as linguagens dão suporte à sobrecarga de operador ou à definição de operadores personalizados. Se optar por implementar esses operadores de conversão, você também deverá fornecer uma maneira alternativa para realizar a conversão. Recomendamos que você forneça métodos `From`Xxx e `To`Xxx.

O exemplo a seguir define conversões explícitas e implícitas compatíveis com CLS. Ele cria uma classe `UDouble` que representa um número de ponto flutuante de precisão dupla com sinal. Ele fornece conversões implícitas de `UDouble` em `Double` e conversões explícitas de `UDouble` em `Single`, de `Double` em `UDouble` e de `Single` em `UDouble`. Ele também define um método `ToDouble` como uma alternativa ao operador de conversão implícita e os métodos `ToSingle`, `FromDouble` e `FromSingle` como alternativas aos operadores de conversão explícita.

```
using System;

public struct UDouble
{
   private double number;

   public UDouble(double value)
   {
      if (value < 0)
         throw new InvalidCastException("A negative value cannot be converted to a UDouble.");

      number = value;
   }

   public UDouble(float value)
   {
      if (value < 0)
         throw new InvalidCastException("A negative value cannot be converted to a UDouble.");

      number = value;
   }

   public static readonly UDouble MinValue = (UDouble) 0.0;
   public static readonly UDouble MaxValue = (UDouble) Double.MaxValue;

   public static explicit operator Double(UDouble value)
   {
      return value.number;
```

```
    }

    public static implicit operator Single(UDouble value)
    {
       if (value.number > (double) Single.MaxValue)
          throw new InvalidCastException("A UDouble value is out of range of the Single type.");

       return (float) value.number;
    }

    public static explicit operator UDouble(double value)
    {
       if (value < 0)
          throw new InvalidCastException("A negative value cannot be converted to a UDouble.");

       return new UDouble(value);
    }

    public static implicit operator UDouble(float value)
    {
       if (value < 0)
          throw new InvalidCastException("A negative value cannot be converted to a UDouble.");

       return new UDouble(value);
    }

    public static Double ToDouble(UDouble value)
    {
       return (Double) value;
    }

    public static float ToSingle(UDouble value)
    {
       return (float) value;
    }

    public static UDouble FromDouble(double value)
    {
       return new UDouble(value);
    }

    public static UDouble FromSingle(float value)
    {
       return new UDouble(value);
    }
 }
```

```
Public Structure UDouble
   Private number As Double

   Public Sub New(value As Double)
      If value < 0 Then
         Throw New InvalidCastException("A negative value cannot be converted to a UDouble.")
      End If
      number = value
   End Sub

   Public Sub New(value As Single)
      If value < 0 Then
         Throw New InvalidCastException("A negative value cannot be converted to a UDouble.")
      End If
      number = value
   End Sub

   Public Shared ReadOnly MinValue As UDouble = CType(0.0, UDouble)
   Public Shared ReadOnly MaxValue As UDouble = Double.MaxValue

   Public Shared Widening Operator CType(value As UDouble) As Double
      Return value.number
   End Operator

   Public Shared Narrowing Operator CType(value As UDouble) As Single
      If value.number > CDbl(Single.MaxValue) Then
         Throw New InvalidCastException("A UDouble value is out of range of the Single type.")
      End If
      Return CSng(value.number)
   End Operator

   Public Shared Narrowing Operator CType(value As Double) As UDouble
      If value < 0 Then
         Throw New InvalidCastException("A negative value cannot be converted to a UDouble.")
      End If
      Return New UDouble(value)
   End Operator

   Public Shared Narrowing Operator CType(value As Single) As UDouble
      If value < 0 Then
         Throw New InvalidCastException("A negative value cannot be converted to a UDouble.")
      End If
      Return New UDouble(value)
   End Operator

   Public Shared Function ToDouble(value As UDouble) As Double
      Return CType(value, Double)
   End Function

   Public Shared Function ToSingle(value As UDouble) As Single
      Return CType(value, Single)
   End Function

   Public Shared Function FromDouble(value As Double) As UDouble
      Return New UDouble(value)
   End Function

   Public Shared Function FromSingle(value As Single) As UDouble
      Return New UDouble(value)
   End Function
End Structure
```

Matrizes

As matrizes compatíveis com CLS estão em conformidade com as seguintes regras:

- Todas as dimensões de uma matriz devem ter um limite inferior igual a zero. O exemplo a seguir cria uma

matriz não compatível com CLS com um limite inferior de um. Independentemente da presença do atributo CLSCompliantAttribute, o compilador não detecta se a matriz retornada pelo método `Numbers.GetTenPrimes` não é compatível com CLS.

```
[assembly: CLSCompliant(true)]

public class Numbers
{
  public static Array GetTenPrimes()
  {
      Array arr = Array.CreateInstance(typeof(Int32), new int[] {10}, new int[] {1});
      arr.SetValue(1, 1);
      arr.SetValue(2, 2);
      arr.SetValue(3, 3);
      arr.SetValue(5, 4);
      arr.SetValue(7, 5);
      arr.SetValue(11, 6);
      arr.SetValue(13, 7);
      arr.SetValue(17, 8);
      arr.SetValue(19, 9);
      arr.SetValue(23, 10);

      return arr;
  }
}
```

```
<Assembly: CLSCompliant(True)>

Public Class Numbers
   Public Shared Function GetTenPrimes() As Array
      Dim arr As Array = Array.CreateInstance(GetType(Int32), {10}, {1})
      arr.SetValue(1, 1)
      arr.SetValue(2, 2)
      arr.SetValue(3, 3)
      arr.SetValue(5, 4)
      arr.SetValue(7, 5)
      arr.SetValue(11, 6)
      arr.SetValue(13, 7)
      arr.SetValue(17, 8)
      arr.SetValue(19, 9)
      arr.SetValue(23, 10)
      Return arr
   End Function
End Class
```

- Todos os elementos de matriz devem consistir em tipos compatíveis com CLS. O exemplo a seguir define dois métodos que retornam matrizes não compatíveis com CLS. O primeiro retorna uma matriz de valores UInt32. O segundo retorna uma matriz Object que inclui valores Int32 e `UInt32`. Embora o compilador identifique a primeira matriz como não compatível devido ao seu tipo `UInt32`, ele não reconhece que a segunda matriz inclui elementos não compatíveis com CLS.

```
using System;

[assembly: CLSCompliant(true)]

public class Numbers
{
  public static UInt32[] GetTenPrimes()
  {
      uint[] arr = { 1u, 2u, 3u, 5u, 7u, 11u, 13u, 17u, 19u };
      return arr;
  }

  public static Object[] GetFivePrimes()
  {
      Object[] arr = { 1, 2, 3, 5u, 7u };
      return arr;
  }
}
// Compilation produces a compiler warning like the following:
//    Array2.cs(8,27): warning CS3002: Return type of 'Numbers.GetTenPrimes()' is not
//            CLS-compliant
```

```
<Assembly: CLSCompliant(True)>

Public Class Numbers
   Public Shared Function GetTenPrimes() As UInt32()
      Return { 1ui, 2ui, 3ui, 5ui, 7ui, 11ui, 13ui, 17ui, 19ui }
   End Function
   Public Shared Function GetFivePrimes() As Object()
      Dim arr() As Object = { 1, 2, 3, 5ui, 7ui }
      Return arr
   End Function
End Class
' Compilation produces a compiler warning like the following:
'    warning BC40027: Return type of function 'GetTenPrimes' is not CLS-compliant.
```

- A resolução de sobrecarga para métodos que tenham parâmetros de matriz se baseia no fato de que são matrizes e em seu tipo de elemento. Por esse motivo, a seguinte definição de um método `GetSquares` sobrecarregado é compatível com CLS.

```
using System;
using System.Numerics;

[assembly: CLSCompliant(true)]

public class Numbers
{
  public static byte[] GetSquares(byte[] numbers)
  {
      byte[] numbersOut = new byte[numbers.Length];
      for (int ctr = 0; ctr < numbers.Length; ctr++) {
          int square = ((int) numbers[ctr]) * ((int) numbers[ctr]);
          if (square <= Byte.MaxValue)
              numbersOut[ctr] = (byte) square;
          // If there's an overflow, assign MaxValue to the corresponding
          // element.
          else
              numbersOut[ctr] = Byte.MaxValue;

      }
      return numbersOut;
  }

  public static BigInteger[] GetSquares(BigInteger[] numbers)
{
      BigInteger[] numbersOut = new BigInteger[numbers.Length];
      for (int ctr = 0; ctr < numbers.Length; ctr++)
          numbersOut[ctr] = numbers[ctr] * numbers[ctr];

     return numbersOut;
  }
}
```

```
Imports System.Numerics

<Assembly: CLSCompliant(True)>

Public Module Numbers
   Public Function GetSquares(numbers As Byte()) As Byte()
      Dim numbersOut(numbers.Length - 1) As Byte
      For ctr As Integer = 0 To numbers.Length - 1
         Dim square As Integer = (CInt(numbers(ctr)) * CInt(numbers(ctr)))
         If square <= Byte.MaxValue Then
            numbersOut(ctr) = CByte(square)
         ' If there's an overflow, assign MaxValue to the corresponding
         ' element.
         Else
            numbersOut(ctr) = Byte.MaxValue
         End If
      Next
      Return numbersOut
   End Function

   Public Function GetSquares(numbers As BigInteger()) As BigInteger()
       Dim numbersOut(numbers.Length - 1) As BigInteger
       For ctr As Integer = 0 To numbers.Length - 1
          numbersOut(ctr) = numbers(ctr) * numbers(ctr)
       Next
       Return numbersOut
   End Function
End Module
```

Interfaces

Interfaces compatíveis com CLS podem definir propriedades, eventos e métodos virtuais (métodos sem

implementação). Uma interface compatível com CLS não pode ter nenhum dos seguintes itens:

- Métodos estáticos ou campos estáticos. O compilador do C# gerará erros de compilador se você definir um membro estático em uma interface.
- Campos. O compilador do C# gerará erros de compilador se você definir um campo em uma interface.
- Métodos que não são compatíveis com CLS. Por exemplo, a definição a seguir da interface inclui um método, `INumber.GetUnsigned`, que está marcado como não compatível com CLS. Este exemplo gera um aviso do compilador.

```
using System;

[assembly:CLSCompliant(true)]

public interface INumber
{
    int Length();
    [CLSCompliant(false)] ulong GetUnsigned();
}
// Attempting to compile the example displays output like the following:
//    Interface2.cs(8,32): warning CS3010: 'INumber.GetUnsigned()': CLS-compliant interfaces
//            must have only CLS-compliant members
```

```
<Assembly: CLSCompliant(True)>

Public Interface INumber
  Function Length As Integer
    <CLSCompliant(False)> Function GetUnsigned As ULong
  End Interface
  ' Attempting to compile the example displays output like the following:
  '    Interface2.vb(9) : warning BC40033: Non CLS-compliant 'function' is not allowed in a
  '    CLS-compliant interface.
  '
  '       <CLSCompliant(False)> Function GetUnsigned As ULong
  '                                      ~~~~~~~~~~~
```

  Devido a essa regra, os tipos compatíveis com CLS não são necessários para implementar membros não compatíveis com CLS. Se uma estrutura compatível com CLS expuser uma classe que implementa uma interface não compatível com CLS, ela também deverá fornecer implementações concretas de todos os membros não compatíveis com CLS.

Compiladores de linguagem compatíveis com CLS também devem permitir que uma classe forneça implementações separadas dos membros com o mesmo nome e a assinatura em várias interfaces. O C# dá suporte a implementações explícitas de interface para fornecer implementações diferentes de métodos com nomes idênticos. O exemplo a seguir ilustra esse cenário, definindo uma classe `Temperature` que implementa as interfaces `ICelsius` e `IFahrenheit` como implementações explícitas de interface.

```
using System;

[assembly: CLSCompliant(true)]

public interface IFahrenheit
{
   decimal GetTemperature();
}

public interface ICelsius
{
   decimal GetTemperature();
}

public class Temperature : ICelsius, IFahrenheit
{
   private decimal _value;

   public Temperature(decimal value)
   {
      // We assume that this is the Celsius value.
      _value = value;
   }

   decimal IFahrenheit.GetTemperature()
   {
      return _value * 9 / 5 + 32;
   }

   decimal ICelsius.GetTemperature()
   {
      return _value;
   }
}
public class Example
{
   public static void Main()
   {
      Temperature temp = new Temperature(100.0m);
      ICelsius cTemp = temp;
      IFahrenheit fTemp = temp;
      Console.WriteLine("Temperature in Celsius: {0} degrees",
                        cTemp.GetTemperature());
      Console.WriteLine("Temperature in Fahrenheit: {0} degrees",
                        fTemp.GetTemperature());
   }
}
// The example displays the following output:
//       Temperature in Celsius: 100.0 degrees
//       Temperature in Fahrenheit: 212.0 degrees
```

```
Assembly: CLSCompliant(True)>

Public Interface IFahrenheit
   Function GetTemperature() As Decimal
End Interface

Public Interface ICelsius
   Function GetTemperature() As Decimal
End Interface

Public Class Temperature : Implements ICelsius, IFahrenheit
   Private _value As Decimal

   Public Sub New(value As Decimal)
      ' We assume that this is the Celsius value.
      _value = value
   End Sub

   Public Function GetFahrenheit() As Decimal _
          Implements IFahrenheit.GetTemperature
      Return _value * 9 / 5 + 32
   End Function

   Public Function GetCelsius() As Decimal _
          Implements ICelsius.GetTemperature
      Return _value
   End Function
End Class

Module Example
   Public Sub Main()
      Dim temp As New Temperature(100.0d)
      Console.WriteLine("Temperature in Celsius: {0} degrees",
                        temp.GetCelsius())
      Console.WriteLine("Temperature in Fahrenheit: {0} degrees",
                        temp.GetFahrenheit())
   End Sub
End Module
' The example displays the following output:
'       Temperature in Celsius: 100.0 degrees
'       Temperature in Fahrenheit: 212.0 degrees
```

Enumerações

Enumerações compatíveis com CLS devem seguir estas regras:

- O tipo subjacente da enumeração deve ser um inteiro intrínseco compatível com CLS (Byte, Int16, Int32 ou Int64). Por exemplo, o código a seguir tenta definir uma enumeração cujo tipo subjacente é UInt32 e gera um aviso do compilador.

```
using System;

[assembly: CLSCompliant(true)]

public enum Size : uint {
    Unspecified = 0,
    XSmall = 1,
    Small = 2,
    Medium = 3,
    Large = 4,
    XLarge = 5
};

public class Clothing
{
    public string Name;
    public string Type;
    public string Size;
}
// The attempt to compile the example displays a compiler warning like the following:
//    Enum3.cs(6,13): warning CS3009: 'Size': base type 'uint' is not CLS-compliant
```

```
<Assembly: CLSCompliant(True)>

Public Enum Size As UInt32
   Unspecified = 0
   XSmall = 1
   Small = 2
   Medium = 3
   Large = 4
   XLarge = 5
End Enum

Public Class Clothing
   Public Name As String
   Public Type As String
   Public Size As Size
End Class
' The attempt to compile the example displays a compiler warning like the following:
'    Enum3.vb(6) : warning BC40032: Underlying type 'UInt32' of Enum is not CLS-compliant.
'
'    Public Enum Size As UInt32
'                ~~~~
```

- Um tipo de enumeração deve ter um campo de instância única chamado `Value__` que foi marcado com o atributo `FieldAttributes.RTSpecialName`. Isso permite que você referencie o valor do campo implicitamente.
- Uma enumeração inclui campos estáticos literais, cujos tipos correspondem ao tipo da própria enumeração. Por exemplo, se você definir uma enumeração `State` com valores de `State.On` e `State.Off`, `State.On` e `State.Off` serão campos literais estáticos cujo tipo será `State`.
- Há dois tipos de enumeração:
  - Uma enumeração que representa um conjunto de valores mutuamente excludentes, valores inteiros nomeados. Esse tipo de enumeração é indicado pela ausência do atributo personalizado System.FlagsAttribute.
  - Uma enumeração que representa um conjunto de sinalizadores de bit que podem ser combinados para produzir um valor sem nome. Esse tipo de enumeração é indicado pela presença do atributo personalizado System.FlagsAttribute.

Para obter mais informações, consulte a documentação da estrutura Enum.

- O valor de uma enumeração não está limitado ao intervalo de seus valores especificados. Em outras palavras, o intervalo de valores em uma enumeração é o intervalo de seu valor subjacente. Você pode usar o método `Enum.IsDefined` para determinar se um valor especificado é membro de uma enumeração.

Membros de tipo em geral

O Common Language Specification requer que todos os campos e métodos sejam acessados como membros de uma classe específica. Portanto, campos e métodos estáticos globais (ou seja, campos ou métodos estáticos que são definidos independentemente de um tipo) não são compatíveis com CLS. Se você tentar incluir um campo ou método global no código-fonte, os compiladores do C# gerarão um erro de compilador.

A Common Language Specification dá suporte somente à convenção de chamada gerenciada padrão. Ela não dá suporte a convenções e métodos de chamada não gerenciados com listas de argumentos de variável marcadas com a palavra-chave `varargs`. Para listas de argumentos de variável que são compatíveis com a convenção de chamada gerenciada padrão, use o atributo ParamArrayAttribute ou a implementação da linguagem individual, como a palavra-chave `params` em C# e a palavra-chave `ParamArray` em Visual Basic.

Acessibilidade de membro

A substituição de um membro herdado não pode alterar a acessibilidade desse membro. Por exemplo, um método público em uma classe base não pode ser substituído por um método privado em uma classe derivada. Há uma exceção: um membro `protected internal` (em C#) ou `Protected Friend` (em Visual Basic) em um assembly que é substituído por um tipo em um assembly diferente. Nesse caso, a acessibilidade da substituição é `Protected`.

O exemplo a seguir ilustra o erro que é gerado quando o atributo CLSCompliantAttribute é definido como `true` e `Person`, que é uma classe derivada de `Animal`, tenta alterar a acessibilidade da propriedade `Species` de pública para privada. O exemplo será compilado com êxito se sua acessibilidade for alterada para pública.

```
using System;

[assembly: CLSCompliant(true)]

public class Animal
{
   private string _species;

   public Animal(string species)
   {
      _species = species;
   }

   public virtual string Species
   {
      get { return _species; }
   }

   public override string ToString()
   {
      return _species;
   }
}

public class Human : Animal
{
   private string _name;

   public Human(string name) : base("Homo Sapiens")
   {
      _name = name;
   }

   public string Name
   {
      get { return _name; }
   }

   private override string Species
   {
      get { return base.Species; }
   }

   public override string ToString()
   {
      return _name;
   }
}

public class Example
{
   public static void Main()
   {
      Human p = new Human("John");
      Console.WriteLine(p.Species);
      Console.WriteLine(p.ToString());
   }
}
// The example displays the following output:
//    error CS0621: 'Human.Species': virtual or abstract members cannot be private
```

```
<Assembly: CLSCompliant(True)>

Public Class Animal
   Private _species As String

   Public Sub New(species As String)
      _species = species
   End Sub

   Public Overridable ReadOnly Property Species As String
      Get
         Return _species
      End Get
   End Property

   Public Overrides Function ToString() As String
      Return _species
   End Function
End Class

Public Class Human : Inherits Animal
   Private _name As String

   Public Sub New(name As String)
      MyBase.New("Homo Sapiens")
      _name = name
   End Sub

   Public ReadOnly Property Name As String
      Get
         Return _name
      End Get
   End Property

   Private Overrides ReadOnly Property Species As String
      Get
         Return MyBase.Species
      End Get
   End Property

   Public Overrides Function ToString() As String
      Return _name
   End Function
End Class

Public Module Example
   Public Sub Main()
      Dim p As New Human("John")
      Console.WriteLine(p.Species)
      Console.WriteLine(p.ToString())
   End Sub
End Module
' The example displays the following output:
'     'Private Overrides ReadOnly Property Species As String' cannot override
'     'Public Overridable ReadOnly Property Species As String' because
'      they have different access levels.
'
'         Private Overrides ReadOnly Property Species As String
```

Os tipos na assinatura de um membro deverão ser acessíveis sempre que o membro for acessível. Por exemplo, isso significa que um membro público não pode incluir um parâmetro cujo tipo seja privado, protegido ou interno. O exemplo a seguir ilustra o erro do compilador que resulta quando um construtor de classe `StringWrapper` expõe um valor de enumeração `StringOperationType` interno que determina como um valor de cadeia de caracteres deve ser encapsulado.

```
using System;
using System.Text;

public class StringWrapper
{
   string internalString;
   StringBuilder internalSB = null;
   bool useSB = false;

   public StringWrapper(StringOperationType type)
   {
      if (type == StringOperationType.Normal) {
         useSB = false;
      }
      else {
         useSB = true;
         internalSB = new StringBuilder();
      }
   }

   // The remaining source code...
}

internal enum StringOperationType { Normal, Dynamic }
// The attempt to compile the example displays the following output:
//    error CS0051: Inconsistent accessibility: parameter type
//            'StringOperationType' is less accessible than method
//            'StringWrapper.StringWrapper(StringOperationType)'
```

```
Imports System.Text

<Assembly:CLSCompliant(True)>

Public Class StringWrapper

   Dim internalString As String
   Dim internalSB As StringBuilder = Nothing
   Dim useSB As Boolean = False

   Public Sub New(type As StringOperationType)
      If type = StringOperationType.Normal Then
         useSB = False
      Else
         internalSB = New StringBuilder()
         useSB = True
      End If
   End Sub

   ' The remaining source code...
End Class

Friend Enum StringOperationType As Integer
   Normal = 0
   Dynamic = 1
End Enum
' The attempt to compile the example displays the following output:
'    error BC30909: 'type' cannot expose type 'StringOperationType'
'     outside the project through class 'StringWrapper'.
'
'       Public Sub New(type As StringOperationType)
'                              ~~~~~~~~~~~~~~~~~~~~
```

Tipos e membros genéricos

Tipos aninhados sempre têm pelo menos o mesmo número de parâmetros genéricos do tipo delimitador. Eles

correspondem por posição aos parâmetros genéricos no tipo delimitador. O tipo genérico também pode incluir novos parâmetros genéricos.

A relação entre os parâmetros de tipo genérico de um tipo de contenção e seus tipos aninhados pode ser ocultada pela sintaxe de linguagens individuais. No exemplo a seguir, um tipo genérico `Outer<T>` contém duas classes aninhadas, `Inner1A` e `Inner1B<U>`. As chamadas para o método `ToString`, que cada classe herda de `Object.ToString`, mostram que cada classe aninhada inclui parâmetros de tipo de sua classe de contenção.

```
using System;

[assembly:CLSCompliant(true)]

public class Outer<T>
{
   T value;

   public Outer(T value)
   {
      this.value = value;
   }

   public class Inner1A : Outer<T>
   {
      public Inner1A(T value) : base(value)
      {  }
   }

   public class Inner1B<U> : Outer<T>
   {
      U value2;

      public Inner1B(T value1, U value2) : base(value1)
      {
         this.value2 = value2;
      }
   }
}

public class Example
{
   public static void Main()
   {
      var inst1 = new Outer<String>("This");
      Console.WriteLine(inst1);

      var inst2 = new Outer<String>.Inner1A("Another");
      Console.WriteLine(inst2);

      var inst3 = new Outer<String>.Inner1B<int>("That", 2);
      Console.WriteLine(inst3);
   }
}
// The example displays the following output:
//       Outer`1[System.String]
//       Outer`1+Inner1A[System.String]
//       Outer`1+Inner1B`1[System.String,System.Int32]
```

```
<Assembly:CLSCompliant(True)>

Public Class Outer(Of T)
   Dim value As T

   Public Sub New(value As T)
      Me.value = value
   End Sub

   Public Class Inner1A : Inherits Outer(Of T)
      Public Sub New(value As T)
         MyBase.New(value)
      End Sub
   End Class

   Public Class Inner1B(Of U) : Inherits Outer(Of T)
      Dim value2 As U

      Public Sub New(value1 As T, value2 As U)
         MyBase.New(value1)
         Me.value2 = value2
      End Sub
   End Class
End Class

Public Module Example
   Public Sub Main()
      Dim inst1 As New Outer(Of String)("This")
      Console.WriteLine(inst1)

      Dim inst2 As New Outer(Of String).Inner1A("Another")
      Console.WriteLine(inst2)

      Dim inst3 As New Outer(Of String).Inner1B(Of Integer)("That", 2)
      Console.WriteLine(inst3)
   End Sub
End Module
' The example displays the following output:
'       Outer`1[System.String]
'       Outer`1+Inner1A[System.String]
'       Outer`1+Inner1B`1[System.String,System.Int32]
```

Os nomes de tipo genérico são codificados no formato *name*'*n*, em que *name* é o nome do tipo, ` é um literal de caractere e *n* é o número de parâmetros declarados no tipo ou, para tipos genéricos aninhados, o número de parâmetros de tipo recém-introduzidos. Essa codificação de nomes de tipo genéricos é principalmente de interesse de desenvolvedores que usam a reflexão para acessar tipos genéricos compatíveis com CLS em uma biblioteca.

Se as restrições forem aplicadas a um tipo genérico, qualquer tipo usado como restrição também deverá ser compatível com CLS. O exemplo a seguir define uma classe chamada `BaseClass` que não é compatível com CLS e uma classe genérica chamada `BaseCollection` cujo parâmetro de tipo deve derivar de `BaseClass`. Mas como `BaseClass` não é compatível com CLS, o compilador emite um aviso.

```
using System;

[assembly:CLSCompliant(true)]

[CLSCompliant(false)] public class BaseClass
{}

public class BaseCollection<T> where T : BaseClass
{}
// Attempting to compile the example displays the following output:
//    warning CS3024: Constraint type 'BaseClass' is not CLS-compliant
```

```
Assembly: CLSCompliant(True)>

<CLSCompliant(False)> Public Class BaseClass
End Class

Public Class BaseCollection(Of T As BaseClass)
End Class
' Attempting to compile the example displays the following output:
'    warning BC40040: Generic parameter constraint type 'BaseClass' is not
'    CLS-compliant.
'
'    Public Class BaseCollection(Of T As BaseClass)
'                                       ~~~~~~~~~
```

Se um tipo genérico for derivado de um tipo de base genérico, ele deverá redeclarar todas as restrições, para assegurar que as restrições no tipo de base também sejam atendidas. O exemplo a seguir define um `Number<T>` que pode representar qualquer tipo numérico. Ele também define uma classe `FloatingPoint<T>` que representa um valor de ponto flutuante. No entanto, o código-fonte falha na compilação porque não aplica a restrição em `Number<T>` (esse T deve ser um tipo de valor) a `FloatingPoint<T>`.

```
using System;

[assembly:CLSCompliant(true)]

public class Number<T> where T : struct
{
   // use Double as the underlying type, since its range is a superset of
   // the ranges of all numeric types except BigInteger.
   protected double number;

   public Number(T value)
   {
      try {
         this.number = Convert.ToDouble(value);
      }
      catch (OverflowException e) {
         throw new ArgumentException("value is too large.", e);
      }
      catch (InvalidCastException e) {
         throw new ArgumentException("The value parameter is not numeric.", e);
      }
   }

   public T Add(T value)
   {
      return (T) Convert.ChangeType(number + Convert.ToDouble(value), typeof(T));
   }

   public T Subtract(T value)
   {
      return (T) Convert.ChangeType(number - Convert.ToDouble(value), typeof(T));
   }
}

public class FloatingPoint<T> : Number<T>
{
   public FloatingPoint(T number) : base(number)
   {
      if (typeof(float) == number.GetType() ||
          typeof(double) == number.GetType() ||
          typeof(decimal) == number.GetType())
         this.number = Convert.ToDouble(number);
      else
         throw new ArgumentException("The number parameter is not a floating-point number.");
   }
}
// The attempt to compile the example displays the following output:
//       error CS0453: The type 'T' must be a non-nullable value type in
//               order to use it as parameter 'T' in the generic type or method 'Number<T>'
```

```
<Assembly:CLSCompliant(True)>

Public Class Number(Of T As Structure)
   ' Use Double as the underlying type, since its range is a superset of
   ' the ranges of all numeric types except BigInteger.
   Protected number As Double

   Public Sub New(value As T)
      Try
         Me.number = Convert.ToDouble(value)
      Catch e As OverflowException
         Throw New ArgumentException("value is too large.", e)
      Catch e As InvalidCastException
         Throw New ArgumentException("The value parameter is not numeric.", e)
      End Try
   End Sub

   Public Function Add(value As T) As T
      Return CType(Convert.ChangeType(number + Convert.ToDouble(value), GetType(T)), T)
   End Function

   Public Function Subtract(value As T) As T
      Return CType(Convert.ChangeType(number - Convert.ToDouble(value), GetType(T)), T)
   End Function
End Class

Public Class FloatingPoint(Of T) : Inherits Number(Of T)
   Public Sub New(number As T)
      MyBase.New(number)
      If TypeOf number Is Single Or
               TypeOf number Is Double Or
               TypeOf number Is Decimal Then
         Me.number = Convert.ToDouble(number)
      Else
         throw new ArgumentException("The number parameter is not a floating-point number.")
      End If
   End Sub
End Class
' The attempt to compile the example displays the following output:
'    error BC32105: Type argument 'T' does not satisfy the 'Structure'
'    constraint for type parameter 'T'.
'
'    Public Class FloatingPoint(Of T) : Inherits Number(Of T)
'                                                    ~
```

O exemplo será compilado com êxito se a restrição for adicionada à classe `FloatingPoint<T>`.

```
using System;

[assembly:CLSCompliant(true)]

public class Number<T> where T : struct
{
   // use Double as the underlying type, since its range is a superset of
   // the ranges of all numeric types except BigInteger.
   protected double number;

   public Number(T value)
   {
      try {
         this.number = Convert.ToDouble(value);
      }
      catch (OverflowException e) {
         throw new ArgumentException("value is too large.", e);
      }
      catch (InvalidCastException e) {
         throw new ArgumentException("The value parameter is not numeric.", e);
      }
   }

   public T Add(T value)
   {
      return (T) Convert.ChangeType(number + Convert.ToDouble(value), typeof(T));
   }

   public T Subtract(T value)
   {
      return (T) Convert.ChangeType(number - Convert.ToDouble(value), typeof(T));
   }
}

public class FloatingPoint<T> : Number<T> where T : struct
{
   public FloatingPoint(T number) : base(number)
   {
      if (typeof(float) == number.GetType() ||
          typeof(double) == number.GetType() ||
          typeof(decimal) == number.GetType())
         this.number = Convert.ToDouble(number);
      else
         throw new ArgumentException("The number parameter is not a floating-point number.");
   }
}
```

```
<Assembly:CLSCompliant(True)>

Public Class Number(Of T As Structure)
   ' Use Double as the underlying type, since its range is a superset of
   ' the ranges of all numeric types except BigInteger.
   Protected number As Double

   Public Sub New(value As T)
      Try
         Me.number = Convert.ToDouble(value)
      Catch e As OverflowException
         Throw New ArgumentException("value is too large.", e)
      Catch e As InvalidCastException
         Throw New ArgumentException("The value parameter is not numeric.", e)
      End Try
   End Sub

   Public Function Add(value As T) As T
      Return CType(Convert.ChangeType(number + Convert.ToDouble(value), GetType(T)), T)
   End Function

   Public Function Subtract(value As T) As T
      Return CType(Convert.ChangeType(number - Convert.ToDouble(value), GetType(T)), T)
   End Function
End Class

Public Class FloatingPoint(Of T As Structure) : Inherits Number(Of T)
   Public Sub New(number As T)
      MyBase.New(number)
      If TypeOf number Is Single Or
               TypeOf number Is Double Or
               TypeOf number Is Decimal Then
         Me.number = Convert.ToDouble(number)
      Else
         throw new ArgumentException("The number parameter is not a floating-point number.")
      End If
   End Sub
End Class
```

A Common Language Specification impõe um modelo por instanciação conservador para tipos aninhados e membros protegidos. Tipos genéricos abertos não podem expor campos ou membros com assinaturas que contenham uma instanciação específica de um tipo genérico aninhado, protegido. Tipos não genéricos que estendam uma instanciação específica de uma interface ou classe base genérica não podem expor campos ou membros com assinaturas que contenham uma instanciação diferente de um tipo genérico aninhado e protegido.

O exemplo a seguir define um tipo genérico, `C1<T>`, e uma classe protegida, `C1<T>.N`. `C1<T>` possui dois métodos, `M1` e `M2`. No entanto, `M1` não é compatível com CLS porque tenta retornar um objeto `C1<int>.N` de `C1<T>`. Uma segunda classe, `C2`, é derivada de `C1<long>`. Tem dois métodos, `M3` e `M4`. `M3` não é compatível com CLS porque tenta retornar um `C1<int>.N` objeto de uma subclasse de `C1<long>`. Os compiladores de linguagens podem ser ainda mais restritivos. Neste exemplo, o Visual Basic exibe um erro ao tentar compilar `M4`.

```
using System;

[assembly:CLSCompliant(true)]

public class C1<T>
{
   protected class N { }

   protected void M1(C1<int>.N n) { } // Not CLS-compliant - C1<int>.N not
                                     // accessible from within C1<T> in all
                                     // languages
   protected void M2(C1<T>.N n) { }   // CLS-compliant – C1<T>.N accessible
                                     // inside C1<T>
}

public class C2 : C1<long>
{
   protected void M3(C1<int>.N n) { }  // Not CLS-compliant – C1<int>.N is not
                                       // accessible in C2 (extends C1<long>)

   protected void M4(C1<long>.N n) { } // CLS-compliant, C1<long>.N is
                                       // accessible in C2 (extends C1<long>)
}
// Attempting to compile the example displays output like the following:
//       Generics4.cs(9,22): warning CS3001: Argument type 'C1<int>.N' is not CLS-compliant
//       Generics4.cs(18,22): warning CS3001: Argument type 'C1<int>.N' is not CLS-compliant
```

```
<Assembly:CLSCompliant(True)>

Public Class C1(Of T)
   Protected Class N
   End Class

   Protected Sub M1(n As C1(Of Integer).N)   ' Not CLS-compliant - C1<int>.N not
                                             ' accessible from within C1(Of T) in all
   End Sub                                   ' languages

   Protected Sub M2(n As C1(Of T).N)     ' CLS-compliant – C1(Of T).N accessible
   End Sub                               ' inside C1(Of T)
End Class

Public Class C2 : Inherits C1(Of Long)
   Protected Sub M3(n As C1(Of Integer).N)   ' Not CLS-compliant – C1(Of Integer).N is not
   End Sub                                   ' accessible in C2 (extends C1(Of Long))

   Protected Sub M4(n As C1(Of Long).N)
   End Sub
End Class
' Attempting to compile the example displays output like the following:
'    error BC30508: 'n' cannot expose type 'C1(Of Integer).N' in namespace
'    '<Default>' through class 'C1'.
'
'       Protected Sub M1(n As C1(Of Integer).N)   ' Not CLS-compliant - C1<int>.N not
'                                 ~~~~~~~~~~~~~~~~
'    error BC30389: 'C1(Of T).N' is not accessible in this context because
'    it is 'Protected'.
'
'       Protected Sub M3(n As C1(Of Integer).N)   ' Not CLS-compliant - C1(Of Integer).N is not
'
'                                 ~~~~~~~~~~~~~~~~
'
'    error BC30389: 'C1(Of T).N' is not accessible in this context because it is 'Protected'.
'
'       Protected Sub M4(n As C1(Of Long).N)
'                                 ~~~~~~~~~~~~~
```

Construtores

Os construtores em classes compatíveis com CLS e em estruturas devem seguir estas regras:

- Um construtor de uma classe derivada deve chamar o construtor de instância de sua classe base antes de acessar quaisquer dados de instância herdados. Esse requisito deve-se ao fato de que construtores de classe base não são herdados por suas classes derivadas. Essa regra não se aplica a estruturas que não dão suporte a herança direta.

  Normalmente, os compiladores aplicam essa regra independentemente da conformidade com CLS, conforme mostrado no exemplo a seguir. Ele cria uma classe `Doctor` que é derivada de uma classe `Person`, mas a classe `Doctor` falha ao chamar o construtor de classe `Person` para inicializar campos herdados da instância.

```
using System;

[assembly: CLSCompliant(true)]

public class Person
{
private string fName, lName, _id;

public Person(string firstName, string lastName, string id)
{
    if (String.IsNullOrEmpty(firstName + lastName))
        throw new ArgumentNullException("Either a first name or a last name must be provided.");

    fName = firstName;
    lName = lastName;
    _id = id;
}

public string FirstName
{
    get { return fName; }
}

public string LastName
{
    get { return lName; }
}

public string Id
{
    get { return _id; }
}

public override string ToString()
{
    return String.Format("{0}{1}{2}", fName,
                         String.IsNullOrEmpty(fName) ?  "" : " ",
                         lName);
}
}

public class Doctor : Person
{
public Doctor(string firstName, string lastName, string id)
{
}

public override string ToString()
{
    return "Dr. " + base.ToString();
}
}
// Attempting to compile the example displays output like the following:
//    ctor1.cs(45,11): error CS1729: 'Person' does not contain a constructor that takes 0
//            arguments
//    ctor1.cs(10,11): (Location of symbol related to previous error)
```

```
<Assembly: CLSCompliant(True)>

Public Class Person
   Private fName, lName, _id As String

   Public Sub New(firstName As String, lastName As String, id As String)
      If String.IsNullOrEmpty(firstName + lastName) Then
         Throw New ArgumentNullException("Either a first name or a last name must be provided.")
      End If

      fName = firstName
      lName = lastName
      _id = id
   End Sub

   Public ReadOnly Property FirstName As String
      Get
         Return fName
      End Get
   End Property

   Public ReadOnly Property LastName As String
      Get
         Return lName
      End Get
   End Property

   Public ReadOnly Property Id As String
      Get
         Return _id
      End Get
   End Property

   Public Overrides Function ToString() As String
      Return String.Format("{0}{1}{2}", fName,
                           If(String.IsNullOrEmpty(fName), "", " "),
                           lName)
   End Function
End Class

Public Class Doctor : Inherits Person
   Public Sub New(firstName As String, lastName As String, id As String)
   End Sub

   Public Overrides Function ToString() As String
      Return "Dr. " + MyBase.ToString()
   End Function
End Class
' Attempting to compile the example displays output like the following:
'    Ctor1.vb(46) : error BC30148: First statement of this 'Sub New' must be a call
'    to 'MyBase.New' or 'MyClass.New' because base class 'Person' of 'Doctor' does
'    not have an accessible 'Sub New' that can be called with no arguments.
'
'       Public Sub New()
'                  ~~~
```

- Um construtor de objeto não pode ser chamado, exceto para criar um objeto. Além disso, um objeto não pode ser inicializado duas vezes. Por exemplo, isso significa que `Object.MemberwiseClone` não deve chamar construtores.

Propriedades

As propriedades em tipos compatíveis com CLS devem seguir estas regras:

- Uma propriedade deve ter um setter, um getter ou ambos. Em um assembly, eles são implementados como métodos especiais, o que significa que aparecerão como métodos separados (o getter é chamado de `get`

_propertyname e o setter é `set` *_propertyname*) marcados como `SpecialName` nos metadados do assembly. Os compiladores do C# aplicam automaticamente essa regra, sem a necessidade de aplicar o atributo CLSCompliantAttribute.

- Um tipo de propriedade é o tipo de retorno do getter da propriedade e o último argumento do setter. Esses tipos devem ser compatíveis com CLS e os argumentos não podem ser atribuídos à propriedade por referência (ou seja, não podem ser ponteiros gerenciados).
- Se uma propriedade tiver um getter e um setter, ambos deverão ser virtuais, estáticos ou instâncias. O compilador do C# impõe automaticamente essa regra por meio de sintaxe da definição de propriedade.

Eventos

Um evento é definido por seu nome e tipo. O tipo de evento é um delegado que é usado para indicar o evento. Por exemplo, o evento `DbConnection.StateChange` é do tipo `StateChangeEventHandler`. Além do evento em si, três métodos com nomes com base no nome do evento fornecem a implementação do evento e estão marcados como `SpecialName` nos metadados do assembly:

- Um método para adicionar um manipulador de eventos, chamado `add` *EventName*. Por exemplo, o método de assinatura do evento para o evento `DbConnection.StateChange` é chamado `add_StateChange`.
- Um método para remover um manipulador de eventos, chamado `remove` *EventName*. Por exemplo, o método de remoção para o evento `DbConnection.StateChange` é chamado `remove_StateChange`.
- Um método para indicar que o evento ocorreu, chamado `raise` *_EventName*.

> **NOTE**
> A maioria das regras da Common Language Specification em relação a eventos é implementada por compiladores de linguagem e é transparente para desenvolvedores de componente.

Os métodos para adicionar, remover e acionar o evento devem ter a mesma acessibilidade. Eles também devem ser todos estáticos, instâncias ou virtuais. Os métodos para adicionar e remover um evento têm um parâmetro cujo tipo é o tipo de delegado do evento. Os métodos para adicionar e remover devem estar ambos presentes ou ausentes.

O exemplo a seguir define uma classe compatível com CLS chamada `Temperature` que acionará um evento `TemperatureChanged` se a mudança de temperatura entre as duas leituras for igual ou exceder o valor de limite. A classe `Temperature` define explicitamente um método `raise_TemperatureChanged` para executar seletivamente os manipuladores de eventos.

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

[assembly: CLSCompliant(true)]

public class TemperatureChangedEventArgs : EventArgs
{
   private Decimal originalTemp;
   private Decimal newTemp;
   private DateTimeOffset when;

   public TemperatureChangedEventArgs(Decimal original, Decimal @new, DateTimeOffset time)
   {
      originalTemp = original;
      newTemp = @new;
      when = time;
   }
```

```
   public Decimal OldTemperature
   {
      get { return originalTemp; }
   }

   public Decimal CurrentTemperature
   {
      get { return newTemp; }
   }

   public DateTimeOffset Time
   {
      get { return when; }
   }
}

public delegate void TemperatureChanged(Object sender, TemperatureChangedEventArgs e);

public class Temperature
{
   private struct TemperatureInfo
   {
      public Decimal Temperature;
      public DateTimeOffset Recorded;
   }

   public event TemperatureChanged TemperatureChanged;

   private Decimal previous;
   private Decimal current;
   private Decimal tolerance;
   private List<TemperatureInfo> tis = new List<TemperatureInfo>();

   public Temperature(Decimal temperature, Decimal tolerance)
   {
      current = temperature;
      TemperatureInfo ti = new TemperatureInfo();
      ti.Temperature = temperature;
      tis.Add(ti);
      ti.Recorded = DateTimeOffset.UtcNow;
      this.tolerance = tolerance;
   }

   public Decimal CurrentTemperature
   {
      get { return current; }
      set {
         TemperatureInfo ti = new TemperatureInfo();
         ti.Temperature = value;
         ti.Recorded = DateTimeOffset.UtcNow;
         previous = current;
         current = value;
         if (Math.Abs(current - previous) >= tolerance)
            raise_TemperatureChanged(new TemperatureChangedEventArgs(previous, current, ti.Recorded));
      }
   }

   public void raise_TemperatureChanged(TemperatureChangedEventArgs eventArgs)
   {
      if (TemperatureChanged == null)
         return;

      foreach (TemperatureChanged d in TemperatureChanged.GetInvocationList()) {
         if (d.Method.Name.Contains("Duplicate"))
            Console.WriteLine("Duplicate event handler; event handler not executed.");
         else
            d.Invoke(this, eventArgs);
      }
   }
```

```
}

public class Example
{
   public Temperature temp;

   public static void Main()
   {
      Example ex = new Example();
   }

   public Example()
   {
      temp = new Temperature(65, 3);
      temp.TemperatureChanged += this.TemperatureNotification;
      RecordTemperatures();
      Example ex = new Example(temp);
      ex.RecordTemperatures();
   }

   public Example(Temperature t)
   {
      temp = t;
      RecordTemperatures();
   }

   public void RecordTemperatures()
   {
      temp.TemperatureChanged += this.DuplicateTemperatureNotification;
      temp.CurrentTemperature = 66;
      temp.CurrentTemperature = 63;
   }

   internal void TemperatureNotification(Object sender, TemperatureChangedEventArgs e)
   {
      Console.WriteLine("Notification 1: The temperature changed from {0} to {1}", e.OldTemperature,
e.CurrentTemperature);
   }

   public void DuplicateTemperatureNotification(Object sender, TemperatureChangedEventArgs e)
   {
      Console.WriteLine("Notification 2: The temperature changed from {0} to {1}", e.OldTemperature,
e.CurrentTemperature);
   }
}
```

```
Imports System.Collections
Imports System.Collections.Generic

<Assembly: CLSCompliant(True)>

Public Class TemperatureChangedEventArgs   : Inherits EventArgs
   Private originalTemp As Decimal
   Private newTemp As Decimal
   Private [when] As DateTimeOffset

   Public Sub New(original As Decimal, [new] As Decimal, [time] As DateTimeOffset)
      originalTemp = original
      newTemp = [new]
      [when] = [time]
   End Sub

   Public ReadOnly Property OldTemperature As Decimal
      Get
         Return originalTemp
      End Get
   End Property
```

```
    Public ReadOnly Property CurrentTemperature As Decimal
       Get
          Return newTemp
       End Get
    End Property

    Public ReadOnly Property [Time] As DateTimeOffset
       Get
          Return [when]
       End Get
    End Property
End Class

Public Delegate Sub TemperatureChanged(sender As Object, e As TemperatureChangedEventArgs)

Public Class Temperature
    Private Structure TemperatureInfo
       Dim Temperature As Decimal
       Dim Recorded As DateTimeOffset
    End Structure

    Public Event TemperatureChanged As TemperatureChanged

    Private previous As Decimal
    Private current As Decimal
    Private tolerance As Decimal
    Private tis As New List(Of TemperatureInfo)

    Public Sub New(temperature As Decimal, tolerance As Decimal)
       current = temperature
       Dim ti As New TemperatureInfo()
       ti.Temperature = temperature
       ti.Recorded = DateTimeOffset.UtcNow
       tis.Add(ti)
       Me.tolerance = tolerance
    End Sub

    Public Property CurrentTemperature As Decimal
       Get
          Return current
       End Get
       Set
          Dim ti As New TemperatureInfo
          ti.Temperature = value
          ti.Recorded = DateTimeOffset.UtcNow
          previous = current
          current = value
          If Math.Abs(current - previous) >= tolerance Then
             raise_TemperatureChanged(New TemperatureChangedEventArgs(previous, current, ti.Recorded))
          End If
       End Set
    End Property

    Public Sub raise_TemperatureChanged(eventArgs As TemperatureChangedEventArgs)
       If TemperatureChangedEvent Is Nothing Then Exit Sub

       Dim ListenerList() As System.Delegate = TemperatureChangedEvent.GetInvocationList()
       For Each d As TemperatureChanged In TemperatureChangedEvent.GetInvocationList()
          If d.Method.Name.Contains("Duplicate") Then
             Console.WriteLine("Duplicate event handler; event handler not executed.")
          Else
             d.Invoke(Me, eventArgs)
          End If
       Next
    End Sub
End Class

Public Class Example
```

```
Public Class Example
   Public WithEvents temp As Temperature

   Public Shared Sub Main()
      Dim ex As New Example()
   End Sub

   Public Sub New()
      temp = New Temperature(65, 3)
      RecordTemperatures()
      Dim ex As New Example(temp)
      ex.RecordTemperatures()
   End Sub

   Public Sub New(t As Temperature)
      temp = t
      RecordTemperatures()
   End Sub

   Public Sub RecordTemperatures()
      temp.CurrentTemperature = 66
      temp.CurrentTemperature = 63

   End Sub

   Friend Shared Sub TemperatureNotification(sender As Object, e As TemperatureChangedEventArgs) _
          Handles temp.TemperatureChanged
      Console.WriteLine("Notification 1: The temperature changed from {0} to {1}", e.OldTemperature,
e.CurrentTemperature)
   End Sub

   Friend Shared Sub DuplicateTemperatureNotification(sender As Object, e As TemperatureChangedEventArgs) _
          Handles temp.TemperatureChanged
      Console.WriteLine("Notification 2: The temperature changed from {0} to {1}", e.OldTemperature,
e.CurrentTemperature)
   End Sub
End Class
```

Sobrecargas

A Common Language Specification impõe os seguintes requisitos em membros sobrecarregados:

- Os membros podem ser sobrecarregados com base no número de parâmetros e no tipo de qualquer parâmetro. A convenção de chamada, o tipo de retorno, os modificadores personalizados aplicados ao método ou seus parâmetros e se os parâmetros são passados por valor ou por referência não são considerados na diferenciação entre sobrecargas. Para obter um exemplo, consulte o código para o requisito de que os nomes devem ser exclusivos em um escopo na seção Convenções de nomenclatura.
- Somente propriedades e métodos podem ser sobrecarregados. Campos e eventos não podem ser sobrecarregados.
- Métodos genéricos podem ser sobrecarregados com base no número de seus parâmetros genéricos.

> **NOTE**
> Os operadores `op_Explicit` e `op_Implicit` são exceções à regra de que o valor retornado não é considerado parte de uma assinatura de método para resolução de sobrecarga. Esses dois operadores podem ser sobrecarregados com base nos parâmetros e valor retornado.

Exceções

Objetos de exceção devem derivar de System.Exception ou de outro tipo derivado de `System.Exception`. O exemplo a seguir ilustra o erro do compilador que resulta quando uma classe personalizada chamada `ErrorClass` é usada para tratamento de exceções.

```
using System;

[assembly: CLSCompliant(true)]

public class ErrorClass
{
   string msg;

   public ErrorClass(string errorMessage)
   {
      msg = errorMessage;
   }

   public string Message
   {
      get { return msg; }
   }
}

public static class StringUtilities
{
   public static string[] SplitString(this string value, int index)
   {
      if (index < 0 | index > value.Length) {
         ErrorClass badIndex = new ErrorClass("The index is not within the string.");
         throw badIndex;
      }
      string[] retVal = { value.Substring(0, index - 1),
                          value.Substring(index) };
      return retVal;
   }
}
// Compilation produces a compiler error like the following:
//    Exceptions1.cs(26,16): error CS0155: The type caught or thrown must be derived from
//            System.Exception
```

```
Imports System.Runtime.CompilerServices

<Assembly: CLSCompliant(True)>

Public Class ErrorClass
   Dim msg As String

   Public Sub New(errorMessage As String)
      msg = errorMessage
   End Sub

   Public ReadOnly Property Message As String
      Get
         Return msg
      End Get
   End Property
End Class

Public Module StringUtilities
   <Extension()> Public Function SplitString(value As String, index As Integer) As String()
      If index < 0 Or index > value.Length Then
         Dim BadIndex As New ErrorClass("The index is not within the string.")
         Throw BadIndex
      End If
      Dim retVal() As String = { value.Substring(0, index - 1),
                                 value.Substring(index) }
      Return retVal
   End Function
End Module
' Compilation produces a compiler error like the following:
'    Exceptions1.vb(27) : error BC30665: 'Throw' operand must derive from 'System.Exception'.
'
'             Throw BadIndex
'             ~~~~~~~~~~~~~~
```

Para corrigir este erro, a classe `ErrorClass` deve herdar de `System.Exception`. Além disso, a propriedade Message deve ser substituída. O exemplo a seguir corrige esses erros para definir uma classe `ErrorClass` que seja compatível com CLS.

```
using System;

[assembly: CLSCompliant(true)]

public class ErrorClass : Exception
{
   string msg;

   public ErrorClass(string errorMessage)
   {
      msg = errorMessage;
   }

   public override string Message
   {
      get { return msg; }
   }
}

public static class StringUtilities
{
   public static string[] SplitString(this string value, int index)
   {
      if (index < 0 | index > value.Length) {
         ErrorClass badIndex = new ErrorClass("The index is not within the string.");
         throw badIndex;
      }
      string[] retVal = { value.Substring(0, index - 1),
                          value.Substring(index) };
      return retVal;
   }
}
```

```
Imports System.Runtime.CompilerServices

<Assembly: CLSCompliant(True)>

Public Class ErrorClass : Inherits Exception
   Dim msg As String

   Public Sub New(errorMessage As String)
      msg = errorMessage
   End Sub

   Public Overrides ReadOnly Property Message As String
      Get
         Return msg
      End Get
   End Property
End Class

Public Module StringUtilities
   <Extension()> Public Function SplitString(value As String, index As Integer) As String()
      If index < 0 Or index > value.Length Then
         Dim BadIndex As New ErrorClass("The index is not within the string.")
         Throw BadIndex
      End If
      Dim retVal() As String = { value.Substring(0, index - 1),
                                 value.Substring(index) }
      Return retVal
   End Function
End Module
```

Atributos

Em assemblies do .NET Framework, atributos personalizados fornecem um mecanismo extensível para armazenamento de atributos personalizados e recuperação de metadados sobre objetos de programação, como assemblies, tipos, membros e parâmetros de método. Atributos personalizados devem derivar de System.Attribute ou de um tipo derivado de `System.Attribute`.

O exemplo a seguir viola essa regra. Ele define uma classe `NumericAttribute` que não deriva de `System.Attribute`. Observe que um erro de compilador só acontece quando o atributo não compatível com CLS é aplicado e não quando a classe é definida.

```
using System;

[assembly: CLSCompliant(true)]

[AttributeUsageAttribute(AttributeTargets.Class | AttributeTargets.Struct)]
public class NumericAttribute
{
   private bool _isNumeric;

   public NumericAttribute(bool isNumeric)
   {
      _isNumeric = isNumeric;
   }

   public bool IsNumeric
   {
      get { return _isNumeric; }
   }
}

[Numeric(true)] public struct UDouble
{
   double Value;
}
// Compilation produces a compiler error like the following:
//    Attribute1.cs(22,2): error CS0616: 'NumericAttribute' is not an attribute class
//    Attribute1.cs(7,14): (Location of symbol related to previous error)
```

```
<Assembly: CLSCompliant(True)>

<AttributeUsageAttribute(AttributeTargets.Class Or AttributeTargets.Struct)> _
Public Class NumericAttribute
   Private _isNumeric As Boolean

   Public Sub New(isNumeric As Boolean)
      _isNumeric = isNumeric
   End Sub

   Public ReadOnly Property IsNumeric As Boolean
      Get
         Return _isNumeric
      End Get
   End Property
End Class

<Numeric(True)> Public Structure UDouble
   Dim Value As Double
End Structure
' Compilation produces a compiler error like the following:
'    error BC31504: 'NumericAttribute' cannot be used as an attribute because it
'    does not inherit from 'System.Attribute'.
'
'    <Numeric(True)> Public Structure UDouble
'     ~~~~~~~~~~~~~
```

O construtor ou as propriedades de um atributo compatível com CLS podem expor somente os seguintes tipos:

- Booliano
- Byte
- Char
- Duplo
- Int16
- Int32
- Int64
- Simples
- Cadeia de caracteres
- Tipo
- Qualquer tipo de enumeração cujo tipo subjacente seja `Byte`, `Int16`, `Int32` ou `Int64`.

O exemplo a seguir define uma classe `DescriptionAttribute` que deriva de Atributo. O construtor de classe tem um parâmetro do tipo `Descriptor`, portanto a classe não é compatível com CLS. Observe que o compilador do C# emite um aviso, mas compila com êxito.

```
using System;

[assembly:CLSCompliantAttribute(true)]

public enum DescriptorType { type, member };

public class Descriptor
{
   public DescriptorType Type;
   public String Description;
}

[AttributeUsage(AttributeTargets.All)]
public class DescriptionAttribute : Attribute
{
   private Descriptor desc;

   public DescriptionAttribute(Descriptor d)
   {
      desc = d;
   }

   public Descriptor Descriptor
   { get { return desc; } }
}
// Attempting to compile the example displays output like the following:
//       warning CS3015: 'DescriptionAttribute' has no accessible
//               constructors which use only CLS-compliant types
```

```
<Assembly:CLSCompliantAttribute(True)>

Public Enum DescriptorType As Integer
   Type = 0
   Member = 1
End Enum

Public Class Descriptor
   Public Type As DescriptorType
   Public Description As String
End Class

<AttributeUsage(AttributeTargets.All)> _
Public Class DescriptionAttribute : Inherits Attribute
   Private desc As Descriptor

   Public Sub New(d As Descriptor)
      desc = d
   End Sub

   Public ReadOnly Property Descriptor As Descriptor
      Get
         Return desc
      End Get
   End Property
End Class
```

O atributo CLSCompliantAttribute

O atributo CLSCompliantAttribute é usado para indicar se um elemento do programa está em conformidade com a Common Language Specification. O construtor `CLSCompliantAttribute.CLSCompliantAttribute(Boolean)` inclui um único parâmetro necessário, *isCompliant*, que indica se o elemento de programa é compatível com CLS.

No tempo de compilação, o compilador detecta os elementos não compatíveis que provavelmente são compatíveis com CLS e emite um aviso. O compilador não emite avisos para tipos ou membros explicitamente declarados como não compatíveis.

Os desenvolvedores de componentes podem usar o atributo `CLSCompliantAttribute` de duas maneiras:

- Para definir as partes da interface pública exposta por um componente que são compatíveis com CLS e as partes que não são compatíveis com CLS. Quando o atributo é usado para marcar elementos de programa específicos como compatíveis com CLS, seu uso garante que os elementos sejam acessíveis em todas as linguagens e ferramentas direcionadas ao .NET Framework.
- Para garantir que a interface pública da biblioteca de componentes exponha apenas elementos de programa que são compatíveis com CLS. Se os elementos não forem compatíveis com CLS, os compiladores geralmente emitirão um aviso.

WARNING

Em alguns casos, os compiladores de linguagem aplicam as regras de compatibilidade com CLS independentemente do uso ou não do atributo `CLSCompliantAttribute`. Por exemplo, a definição de um membro `*static` em uma interface viola uma regra CLS. No entanto, se você definir um membro `*static` em uma interface, o compilador do C# exibirá uma mensagem de erro e falhará ao compilar o aplicativo.

O atributo `CLSCompliantAttribute` é marcado com um atributo AttributeUsageAttribute que tem um valor de `AttributeTargets.All`. Esse valor permite que você aplique o atributo `CLSCompliantAttribute` a qualquer elemento de programa, incluindo assemblies, módulos, tipos (classes, estruturas, interfaces, enumerações e delegados),

membros de tipo (construtores, métodos, propriedades, campos e eventos), parâmetros, parâmetros genéricos e valores de retorno. No entanto, na prática, você deve aplicar o atributo somente a assemblies, tipos e membros de tipo. Caso contrário, os compiladores ignoram o atributo e continuam gerando avisos do compilador sempre que encontrarem um parâmetro não compatível, parâmetro genérico ou valor retornado na interface pública da biblioteca.

O valor do atributo `CLSCompliantAttribute` é herdado pelos elementos contidos no programa. Por exemplo, se um assembly for marcado como compatível com CLS, seus tipos também serão compatíveis com CLS. Se um tipo for marcado como compatível com CLS, seus membros e tipos aninhados também serão compatíveis com CLS.

Você pode substituir explicitamente a compatibilidade herdada aplicando o atributo `CLSCompliantAttribute` a um elemento contido no programa. Por exemplo, é possível usar o atributo `CLSCompliantAttribute` com um valor *isCompliant* de `false` para definir um tipo não compatível em um assembly compatível e é possível usar o atributo com um valor *isComplian* de `true` para definir um tipo compatível em um assembly não compatível. Você também pode definir membros não compatíveis em um tipo compatível. No entanto, um tipo não compatível não pode ter membros compatíveis. Portanto, você não pode usar o atributo com um valor *isCompliant* de `true` para substituir a herança de um tipo não compatível.

Ao desenvolver componentes, você sempre deve usar o atributo `CLSCompliantAttribute` para indicar se o assembly, seus tipos e membros são compatíveis com CLS.

Para criar componentes compatíveis com CLS:

1. Use `CLSCompliantAttribute` para marcar o assembly como compatível com CLS.
2. Marque qualquer tipo exposto publicamente no assembly que não seja compatível com CLS como não compatível.
3. Marque qualquer membro publicamente exposto em tipos compatíveis com CLS como não compatíveis.
4. Forneça uma alternativa compatível com CLS para membros não compatíveis com CLS.

Se você marcou com êxito todos os tipos e membros não compatíveis, o compilador não deverá emitir avisos de não conformidade. Entretanto, você deve indicar quais membros não são compatíveis com CLS e listar suas alternativas compatíveis com CLS na documentação do produto.

O exemplo a seguir usa o atributo `CLSCompliantAttribute` para definir um assembly compatível com CLS e um tipo, `CharacterUtilities`, que tem dois membros não compatíveis com CLS. Como ambos os membros são marcados com o atributo `CLSCompliant(false)`, o compilador não produz avisos. A classe também fornece uma alternativa compatível com CLS para ambos os métodos. Normalmente, nós adicionaríamos apenas duas sobrecargas ao método `ToUTF16` para fornecer alternativas compatíveis com CLS. Entretanto, como os métodos não podem ser sobrecarregados com base no valor retornado, os nomes dos métodos compatíveis com CLS são diferentes dos nomes dos métodos não compatíveis.

```
using System;
using System.Text;

[assembly:CLSCompliant(true)]

public class CharacterUtilities
{
   [CLSCompliant(false)] public static ushort ToUTF16(String s)
   {
      s = s.Normalize(NormalizationForm.FormC);
      return Convert.ToUInt16(s[0]);
   }

   [CLSCompliant(false)] public static ushort ToUTF16(Char ch)
   {
      return Convert.ToUInt16(ch);
   }

   // CLS-compliant alternative for ToUTF16(String).
   public static int ToUTF16CodeUnit(String s)
   {
      s = s.Normalize(NormalizationForm.FormC);
      return (int) Convert.ToUInt16(s[0]);
   }

   // CLS-compliant alternative for ToUTF16(Char).
   public static int ToUTF16CodeUnit(Char ch)
   {
      return Convert.ToInt32(ch);
   }

   public bool HasMultipleRepresentations(String s)
   {
      String s1 = s.Normalize(NormalizationForm.FormC);
      return s.Equals(s1);
   }

   public int GetUnicodeCodePoint(Char ch)
   {
      if (Char.IsSurrogate(ch))
         throw new ArgumentException("ch cannot be a high or low surrogate.");

      return Char.ConvertToUtf32(ch.ToString(), 0);
   }

   public int GetUnicodeCodePoint(Char[] chars)
   {
      if (chars.Length > 2)
         throw new ArgumentException("The array has too many characters.");

      if (chars.Length == 2) {
         if (! Char.IsSurrogatePair(chars[0], chars[1]))
            throw new ArgumentException("The array must contain a low and a high surrogate.");
         else
            return Char.ConvertToUtf32(chars[0], chars[1]);
      }
      else {
         return Char.ConvertToUtf32(chars.ToString(), 0);
      }
   }
}
```

```
Imports System.Text

<Assembly:CLSCompliant(True)>

Public Class CharacterUtilities
   <CLSCompliant(False)> Public Shared Function ToUTF16(s As String) As UShort
      s = s.Normalize(NormalizationForm.FormC)
      Return Convert.ToUInt16(s(0))
   End Function

   <CLSCompliant(False)> Public Shared Function ToUTF16(ch As Char) As UShort
      Return Convert.ToUInt16(ch)
   End Function

   ' CLS-compliant alternative for ToUTF16(String).
   Public Shared Function ToUTF16CodeUnit(s As String) As Integer
      s = s.Normalize(NormalizationForm.FormC)
      Return CInt(Convert.ToInt16(s(0)))
   End Function

   ' CLS-compliant alternative for ToUTF16(Char).
   Public Shared Function ToUTF16CodeUnit(ch As Char) As Integer
      Return Convert.ToInt32(ch)
   End Function

   Public Function HasMultipleRepresentations(s As String) As Boolean
      Dim s1 As String = s.Normalize(NormalizationForm.FormC)
      Return s.Equals(s1)
   End Function

   Public Function GetUnicodeCodePoint(ch As Char) As Integer
      If Char.IsSurrogate(ch) Then
         Throw New ArgumentException("ch cannot be a high or low surrogate.")
      End If
      Return Char.ConvertToUtf32(ch.ToString(), 0)
   End Function

   Public Function GetUnicodeCodePoint(chars() As Char) As Integer
      If chars.Length > 2 Then
         Throw New ArgumentException("The array has too many characters.")
      End If
      If chars.Length = 2 Then
         If Not Char.IsSurrogatePair(chars(0), chars(1)) Then
            Throw New ArgumentException("The array must contain a low and a high surrogate.")
         Else
            Return Char.ConvertToUtf32(chars(0), chars(1))
         End If
      Else
         Return Char.ConvertToUtf32(chars.ToString(), 0)
      End If
   End Function
End Class
```

Se você estiver desenvolvendo um aplicativo em vez de uma biblioteca (ou seja, se não estiver expondo tipos ou membros que possam ser consumidos por outros desenvolvedores de aplicativos), a conformidade com CLS dos elementos do programa que seu aplicativo consome só serão de interesse se sua linguagem não der suporte a eles. Nesse caso, seu compilador de linguagem gerará um erro quando você tentar usar um elemento não compatível com CLS.

Interoperabilidade em qualquer idioma

A independência de linguagem tem vários significados possíveis. Um significado envolve o consumo contínuo de tipos gravados em uma linguagem de um aplicativo gravado em outra linguagem. Um segundo significado, que é o enfoque deste artigo, envolve combinar o código gravado em várias linguagens em um único assembly do .NET

Framework.

O exemplo a seguir ilustra a interoperabilidade em qualquer idioma com a criação de uma biblioteca de classes chamada Utilities.dll que inclui duas classes, `NumericLib` e `StringLib`. A classe `NumericLib` é gravada em C#, e a classe `StringLib` é gravada em Visual Basic. Aqui está o código-fonte de `StringUtil.vb`, que inclui um único membro, `ToTitleCase`, em sua classe `StringLib`.

```
Imports System.Collections.Generic
Imports System.Runtime.CompilerServices

Public Module StringLib
   Private exclusions As List(Of String)

   Sub New()
      Dim words() As String = { "a", "an", "and", "of", "the" }
      exclusions = New List(Of String)
      exclusions.AddRange(words)
   End Sub

   <Extension()> _
   Public Function ToTitleCase(title As String) As String
      Dim words() As String = title.Split()
      Dim result As String = String.Empty

      For ctr As Integer = 0 To words.Length - 1
         Dim word As String = words(ctr)
         If ctr = 0 OrElse Not exclusions.Contains(word.ToLower()) Then
            result += word.Substring(0, 1).ToUpper() + _
                      word.Substring(1).ToLower()
         Else
            result += word.ToLower()
         End If
         If ctr <= words.Length - 1 Then
            result += " "
         End If
      Next
      Return result
   End Function
End Module
```

Aqui está o código-fonte de NumberUtil.cs, que define uma classe `NumericLib` com dois membros, `IsEven` e `NearZero`.

```
using System;

public static class NumericLib
{
   public static bool IsEven(this IConvertible number)
   {
      if (number is Byte ||
          number is SByte ||
          number is Int16 ||
          number is UInt16 ||
          number is Int32 ||
          number is UInt32 ||
          number is Int64)
         return ((long) number) % 2 == 0;
      else if (number is UInt64)
         return ((ulong) number) %2 == 0;
      else
         throw new NotSupportedException("IsEven called for a non-integer value.");
   }

   public static bool NearZero(double number)
   {
      return number < .00001;
   }
}
```

Para empacotar as duas classes em um único assembly, você deve compilá-las em módulos. Para compilar o arquivo de código-fonte do Visual Basic em um módulo, use este comando:

```
vbc /t:module StringUtil.vb
```

Para compilar o arquivo de código-fonte do C# em um módulo, use este comando:

```
csc /t:module NumberUtil.cs
```

Então você usa a ferramenta Link (Link.exe) para compilar os dois módulos em um assembly:

```
link numberutil.netmodule stringutil.netmodule /out:UtilityLib.dll /dll
```

O exemplo a seguir chama os métodos `NumericLib.NearZero` e `StringLib.ToTitleCase`. Observe que o código do Visual Basic e o código do C# podem acessar os métodos em ambas as classes.

```
using System;

public class Example
{
   public static void Main()
   {
      Double dbl = 0.0 - Double.Epsilon;
      Console.WriteLine(NumericLib.NearZero(dbl));

      string s = "war and peace";
      Console.WriteLine(s.ToTitleCase());
   }
}
// The example displays the following output:
//       True
//       War and Peace
```

```
Module Example
   Public Sub Main()
      Dim dbl As Double = 0.0 - Double.Epsilon
      Console.WriteLine(NumericLib.NearZero(dbl))

      Dim s As String = "war and peace"
      Console.WriteLine(s.ToTitleCase())
   End Sub
End Module
' The example displays the following output:
'       True
'       War and Peace
```

Para compilar o código do Visual Basic, use este comando:

```
vbc example.vb /r:UtilityLib.dll
```

Para compilar usando C#, altere o nome do compilador de vbc para csc e altere a extensão do arquivo de .vb para .cs:

```
csc example.cs /r:UtilityLib.dll
```

Independência da linguagem e componentes independentes da linguagem

31/10/2019 • 106 minutes to read • Edit Online

O .NET Framework independe da linguagem. Isso significa que, como desenvolvedor, você pode desenvolver em uma das muitas linguagens que segmentam o .NET Framework, como C#, C++/CLI, Eiffel, F#, IronPython, IronRuby, PowerBuilder, Visual Basic, Visual COBOL e Windows PowerShell. É possível acessar os tipos e os membros das bibliotecas de classes desenvolvidas para o .NET Framework sem que seja necessário conhecer a linguagem em que foram originalmente gravados e sem precisar seguir as convenções da linguagem original. Se você for um desenvolvedor de componentes, o componente poderá ser acessado por qualquer aplicativo do .NET Framework, independentemente da linguagem.

> **NOTE**
> A primeira parte deste artigo discorre sobre a criação de componentes independentes de linguagem, ou seja, componentes que podem ser consumidos por aplicativos escritos em qualquer linguagem. Você também pode criar um único componente ou aplicativo de código-fonte gravado em várias linguagens; consulte Interoperabilidade em qualquer idioma na segunda parte deste artigo.

Para interagir completamente com outros objetos gravados em qualquer linguagem, os objetos devem expor aos chamadores somente os recursos comuns a todas as linguagens. Esse conjunto comum de recursos é definido pela CLS (Common Language Specification), que é um conjunto de regras que se aplicam aos assemblies gerados. A Common Language Specification é definida na Partição I, cláusulas 7 a 11 do Padrão ECMA-335: Common Language Infrastructure.

Se o componente estiver de acordo com a Common Language Specification, ele será compatível com a CLS e poderá ser acessado pelo código em assemblies gravados em qualquer linguagem de programação que dê suporte a CLS. É possível determinar se o componente está de acordo com a Common Language Specification no tempo de compilação aplicando-se o atributo CLSCompliantAttribute ao código-fonte. Para obter mais informações, consulte O atributo CLSCompliantAttribute.

Neste artigo:

    - Construtores
    - Propriedades
    - Eventos
    - Sobrecargas
    - Exceções
    - Atributos
- O atributo CLSCompliantAttribute
- Interoperabilidade em qualquer idioma

Regras de conformidade com CLS

Esta seção discute as regras para criar um componente compatível com CLS. Para obter uma lista completa de regras, consulte Partição I, Cláusula 11 do Padrão ECMA-335: Common Language Infrastructure.

> **NOTE**
> A Common Language Specification aborda cada regra de conformidade com CLS à medida que se aplica a consumidores (desenvolvedores que estão acessando programaticamente um componente compatível com CLS), estruturas (desenvolvedores que estão usando um compilador de linguagem para criar bibliotecas compatíveis com CLS) e extensores (desenvolvedores que estão criando uma ferramenta, como um compilador de linguagem ou um analisador de código que cria componentes compatíveis com CLS). Este artigo enfoca as regras que se aplicam às estruturas. Entretanto, algumas das regras que se aplicam a extensores também podem ser aplicadas a assemblies criados usando-se Reflection.Emit.

Para criar um componente independente de linguagem, você só precisa aplicar as regras de compatibilidade com CLS à interface pública do componente. A implementação privada não precisa estar de acordo com a especificação.

> **IMPORTANT**
> As regras de conformidade com CLS só se aplicam à interface pública de um componente e não à implementação privada.

Por exemplo, inteiros sem sinal que não sejam Byte não são compatíveis com CLS. Como a classe `Person` no exemplo a seguir expõe uma propriedade `Age` de tipo UInt16, o código a seguir exibe um aviso do compilador.

```
using System;

[assembly: CLSCompliant(true)]

public class Person
{
   private UInt16 personAge = 0;

   public UInt16 Age
   { get { return personAge; } }
}
// The attempt to compile the example displays the following compiler warning:
//    Public1.cs(10,18): warning CS3003: Type of 'Person.Age' is not CLS-compliant
```

```
<Assembly: CLSCompliant(True)>

Public Class Person
   Private personAge As UInt16

   Public ReadOnly Property Age As UInt16
      Get
         Return personAge
      End Get
   End Property
End Class
' The attempt to compile the example displays the following compiler warning:
'    Public1.vb(9) : warning BC40027: Return type of function 'Age' is not CLS-compliant.
'
'       Public ReadOnly Property Age As UInt16
'                                ~~~
```

É possível tornar a classe `Person` compatível com CLS alternado-se o tipo de propriedade `Age` de UInt16 para Int16, que é um inteiro com sinal 16 bits compatível com CLS. Não é necessário alterar o tipo do campo `personAge` privado.

```
using System;

[assembly: CLSCompliant(true)]

public class Person
{
   private Int16 personAge = 0;

   public Int16 Age
   { get { return personAge; } }
}
```

```
<Assembly: CLSCompliant(True)>

Public Class Person
   Private personAge As UInt16

   Public ReadOnly Property Age As Int16
      Get
         Return CType(personAge, Int16)
      End Get
   End Property
End Class
```

A interface pública de uma biblioteca consiste no seguinte:

- Definições de classes públicas.
- Definições dos membros públicos de classes públicas e definições de membros acessíveis para classes derivadas (ou seja, membros protegidos).
- Parâmetros e tipos de retorno de métodos públicos de classes públicas e parâmetros e tipos de retorno de métodos acessíveis para classes derivadas.

As regras de conformidade com CLS estão listadas na tabela a seguir. O texto das regras é tirado literalmente do Padrão ECMA-335: Common Language Infrastructure, com direitos autorais de 2012 da Ecma International. Informações mais detalhadas sobre essas regras são encontradas nas seções a seguir.

CATEGORIA	CONSULTE	REGRA	NÚMERO DA REGRA
Acessibilidade	Acessibilidade de membro	A acessibilidade não deverá ser alterada ao substituir métodos herdados, exceto na substituição de um método herdado de um assembly diferente com acessibilidade `family-or-assembly`. Nesse caso, a substituição deverá ter a acessibilidade `family`.	10
Acessibilidade	Acessibilidade de membro	A visibilidade e a acessibilidade de tipos e membros deverão ser de tal forma que os tipos na assinatura de qualquer membro sejam visíveis e acessíveis sempre que o próprio membro estiver visível e acessível. Por exemplo, um método público visível fora do assembly não deve ter um argumento cujo tipo seja visível somente dentro do assembly. A visibilidade e a acessibilidade dos tipos que compõem um tipo genérico instanciado usado na assinatura de qualquer membro deverão estar visíveis e acessíveis sempre que o próprio membro estiver visível e acessível. Por exemplo, um tipo genérico instanciado presente na assinatura de um membro visível fora do assembly não deverá ter um argumento genérico cujo tipo seja visível somente dentro do assembly.	12
Matrizes	Matrizes	As matrizes deverão ter elementos com um tipo compatível com CLS e todas as dimensões da matriz deverão ter limites inferiores iguais a zero. Se o item for uma matriz, o tipo do elemento da matriz será necessário para diferenciar as sobrecargas. Quando a sobrecarga é baseada em dois ou mais tipos de matriz, os tipos de elemento deverão ser chamados de tipos.	16

CATEGORIA	CONSULTE	REGRA	NÚMERO DA REGRA
Atributos	Atributos	Os atributos deverão ser do tipo System.Attribute ou de um tipo que o herde.	41
Atributos	Atributos	A CLS só permite um subconjunto das codificações de atributos personalizados. Os únicos tipos que deverão ser exibidos nessas codificações são (consulte a Partição IV): System.Type, System.String, System.Char, System.Boolean, System.Byte, System.Int16, System.Int32, System.Int64, System.Single, System.Double e qualquer tipo de enumeração baseado em um tipo inteiro de base compatível com CLS.	34
Atributos	Atributos	A CLS não permite modificadores obrigatórios visíveis publicamente (`modreq` , consulte a Partição II), mas permite modificadores opcionais (`modopt` , consulte a Partição II) que ela não entende.	35
Construtores	Construtores	Um construtor de objeto deverá chamar um construtor de instância de sua classe base antes de qualquer acesso aos dados da instância herdados. (Isso não se aplica a tipos de valor, que não precisam ter construtores.)	21
Construtores	Construtores	Um construtor de objeto não deverá ser chamado, exceto como parte da criação de um objeto e um objeto não deve ser inicializado duas vezes.	22
Enumerações	Enumerações	O tipo subjacente de um enum deverá ser um tipo de inteiro CLS interno, o nome do campo deverá ser "value__", e esse campo deverá ser marcado como `RTSpecialName` .	7

CATEGORIA	CONSULTE	REGRA	NÚMERO DA REGRA
Enumerações	Enumerações	Há dois tipos diferentes de enums, indicados pela presença ou pela ausência do atributo personalizado System.FlagsAttribute (consulte a Biblioteca da Partição IV). Um representa valores de inteiro nomeados; o outro representa sinalizadores de bit nomeados que podem ser combinados para gerar um valor sem nome. O valor de um `enum` não está limitado aos valores especificados.	8
Enumerações	Enumerações	Campos estáticos de literais de um enum deverão ter o tipo do próprio enum.	9
Eventos	Eventos	Os métodos que implementam um evento deverão ser marcados como `SpecialName` nos metadados.	29
Eventos	Eventos	A acessibilidade de um evento e de seus acessadores deverá ser idêntica.	30
Eventos	Eventos	Os métodos `add` e `remove` de um evento deverão estar presentes ou ausentes.	31
Eventos	Eventos	Os métodos `add` e `remove` de um evento deverão utilizar um parâmetro cada um cujo tipo define o tipo do evento, e ele deverá ser derivado de System.Delegate.	32
Eventos	Eventos	Os eventos deverão respeitar um padrão de nomenclatura específico. O atributo `SpecialName` mencionado na regra 29 da CLS deverá ser ignorado em comparações de nome apropriadas e respeitar as regras do identificador.	33

CATEGORIA	CONSULTE	REGRA	NÚMERO DA REGRA
Exceções	Exceções	Os atributos acionados deverão ser do tipo System.Exception ou de um tipo herdado dele. Mesmo assim, os métodos compatíveis com CLS não precisam bloquear a propagação de outros tipos de exceção.	40
Geral	Conformidade com CLS: as Regras	As regras CLS só se aplicam a essas partes de um tipo acessíveis ou visíveis fora do assembly de definição.	1
Geral	Conformidade com CLS: as Regras	Membros de tipos incompatíveis com CLS não deverão ser marcados como compatíveis com CLS.	2
Genéricos	Tipos e membros genéricos	Os tipos aninhados deverão ter, pelo menos, tantos parâmetros genéricos quanto o tipo delimitador. Os parâmetros genéricos em um tipo aninhado correspondem, por posição, aos parâmetros genéricos no tipo delimitador.	42
Genéricos	Tipos e membros genéricos	O nome de um tipo genérico deverá codificar o número de parâmetros de tipo declarados no tipo não aninhado ou recém-introduzidos no tipo, se aninhado, de acordo com as regras definidas anteriormente.	43
Genéricos	Tipos e membros genéricos	Um tipo genérico deverá redeclarar restrições suficientes para assegurar que todas as restrições no tipo base ou nas interfaces sejam atendidas pelas restrições de tipo genérico.	4444
Genéricos	Tipos e membros genéricos	Tipos usados como restrições em parâmetros genéricos deverão ser compatíveis com CLS.	45

CATEGORIA	CONSULTE	REGRA	NÚMERO DA REGRA
Genéricos	Tipos e membros genéricos	A visibilidade e a acessibilidade de membros (incluindo tipos aninhados) em um tipo genérico instanciado deverão ser consideradas no escopo da instanciação específica, em vez da declaração de tipo genérico como um todo. Supondo isso, as regras de visibilidade e acessibilidade da regra 12 da CLS continuam sendo aplicáveis.	46
Genéricos	Tipos e membros genéricos	Para cada método genérico abstrato ou virtual, deverá haver uma implementação concreta (não abstrata) padrão.	47
Interfaces	Interfaces	As interfaces em conformidade com CLS não deverão exigir a definição de métodos incompatíveis com CLS para implementá-los.	18
Interfaces	Interfaces	As interfaces compatíveis com CLS não deverão definir métodos estáticos, nem devem definir campos.	19
Membros	Membros de tipo em geral	Campos e métodos estáticos globais não são compatíveis com CLS.	36
Membros	--	O valor de um estático literal é especificado por meio do uso de metadados de inicialização do campo. Um literal compatível com CLS deve ter um valor especificado em metadados de inicialização de campo que sejam exatamente do mesmo tipo que o literal (ou do tipo subjacente, se esse literal for um `enum`).	13
Membros	Membros de tipo em geral	A restrição vararg não faz parte da CLS e a única convenção de chamada com suporte pela CLS é a convenção de chamada gerenciada padrão.	15

CATEGORIA	CONSULTE	REGRA	NÚMERO DA REGRA
Convenções de nomenclatura	Convenções de nomenclatura	Os assemblies deverão seguir o Anexo 7 do Relatório Técnico 15 do Padrão Unicode 3.0 que controla o conjunto de caracteres que podem iniciar e ser incluídos em identificadores, disponíveis online em https://www.unicode.org/unicode/reports/tr15/tr15-18.html. Os identificadores deverão estar no formato canônico definido pelo Formulário C de Normalização de Unicode. Para fins de CLS, dois identificadores serão iguais se os mapeamentos em minúsculas (conforme especificado pelos mapeamentos em minúsculas um para um, insensíveis a localidade Unicode) forem os mesmos. Ou seja, para dois identificadores serem considerados diferentes na CLS, eles deverão ser diferentes além de apenas maiúsculas e minúsculas. No entanto, para substituir uma definição herdada, a CLI exige que a codificação precisa da declaração original seja usada.	4
Sobrecarga	Convenções de nomenclatura	Todos os nomes introduzidos em um escopo compatível com CLS deverão ser independentes e distintos do tipo, exceto quando os nomes forem idênticos e resolvidos por meio da sobrecarga. Ou seja, embora o CTS permita que um único tipo use o mesmo nome para um método e um campo, a CLS não permite.	5

CATEGORIA	CONSULTE	REGRA	NÚMERO DA REGRA
Sobrecarga	Convenções de nomenclatura	Campos e tipos aninhados deverão ser diferenciados apenas por comparação de identificador, mesmo que o CTS permita que assinaturas diferentes sejam distinguidas. Métodos, propriedades e eventos com o mesmo nome (por comparação de identificador) deverão ser diferentes além apenas do tipo de retorno, exceto conforme especificado na Regra 39 da CLS.	6
Sobrecarga	Sobrecargas	Somente propriedades e métodos podem ser sobrecarregados.	37
Sobrecarga	Sobrecargas	As propriedades e os métodos só podem ser sobrecarregados com base no número e nos tipos de seus parâmetros, exceto os operadores de conversão chamados `op_Implicit` e `op_Explicit`, que também podem ser sobrecarregados com base no tipo de retorno.	38
Sobrecarga	--	Se dois ou mais métodos em conformidade com CLS declarados em um tipo tiverem o mesmo nome e, para um conjunto específico de instanciações de tipo, tiverem os mesmos tipos de parâmetro e retorno, esses métodos deverão ser semanticamente equivalentes nessas instanciações de tipo.	48
Tipos	Tipo e assinaturas de membro de tipo	System.Object é compatível com CLS. Qualquer outra classe compatível com CLS deverá herdar de uma classe compatível com CLS.	23
Propriedades	Propriedades	Os métodos que implementam os métodos getter e setter de uma propriedade deverão ser marcados como `SpecialName` nos metadados.	24

CATEGORIA	CONSULTE	REGRA	NÚMERO DA REGRA
Propriedades	Propriedades	Os acessadores de uma propriedade deverão ser todos estáticos, virtuais ou de instância.	26
Propriedades	Propriedades	O tipo de uma propriedade deverá ser o tipo de retorno do getter e o tipo do último argumento do setter. Os tipos dos parâmetros da propriedade deverão ser os tipos dos parâmetros do getter e os tipos de todos os parâmetros, menos o parâmetro final do setter. Todos esses tipos deverão ser compatíveis com CLS e não deverão ser ponteiros gerenciados (por exemplo, não deverão ser passados por referência).	27
Propriedades	Propriedades	As propriedades deverão seguir um padrão de nomenclatura específico. O atributo `SpecialName` mencionado na regra 24 da CLS deverá ser ignorado em comparações de nome apropriadas e respeitar as regras do identificador. Uma propriedade deverá ter um método getter, um método setter ou ambos.	28
Conversão de tipos	Conversão de tipos	Se `op_Implicit` ou `op_Explicit` for fornecido, um meio alternativo de fornecimento da coerção deverá ser fornecido.	39
Tipos	Tipo e assinaturas de membro de tipo	Tipos de valor demarcado não são compatíveis com CLS.	3
Tipos	Tipo e assinaturas de membro de tipo	Todos os tipos exibidos em uma assinatura deverão ser compatíveis com CLS. Todos os tipos que compõem um tipo genérico instanciado deverão ser compatíveis com CLS.	11
Tipos	Tipo e assinaturas de membro de tipo	Referências com tipo não são compatíveis com CLS.	14

CATEGORIA	CONSULTE	REGRA	NÚMERO DA REGRA
Tipos	Tipo e assinaturas de membro de tipo	Tipos de ponteiro não gerenciados não são compatíveis com CLS.	17
Tipos	Tipo e assinaturas de membro de tipo	Classes compatíveis com CLS, tipos de valor e interfaces não deverão exigir a implementação de membros incompatíveis com CLS.	20

Tipos e assinaturas de membro de tipo

O tipo System.Object é compatível com CLS e é o tipo base de todos os tipos de objeto no sistema de tipos do .NET Framework. A herança no .NET Framework é implícita (por exemplo, a classe String herda implicitamente da classe Object) ou explícita (por exemplo, a classe CultureNotFoundException herda explicitamente da classe ArgumentException, que herda explicitamente da classe SystemException, que herda explicitamente da classe Exception). Para que um tipo derivado seja compatível com CLS, seu tipo base também deverá ser compatível com CLS.

O exemplo a seguir mostra um tipo derivado cujo tipo de base não é compatível com CLS. Ele define uma classe `Counter` base que usa um inteiro de 32 bits sem sinal como um contador. Como a classe fornece funcionalidade de contador encapsulando um inteiro sem sinal, a classe é marcada como não compatível com CLS. Assim, uma classe derivada, `NonZeroCounter`, também não é compatível com CLS.

```
using System;

[assembly: CLSCompliant(true)]

[CLSCompliant(false)]
public class Counter
{
   UInt32 ctr;

   public Counter()
   {
      ctr = 0;
   }

   protected Counter(UInt32 ctr)
   {
      this.ctr = ctr;
   }

   public override string ToString()
   {
      return String.Format("{0}). ", ctr);
   }

   public UInt32 Value
   {
      get { return ctr; }
   }

   public void Increment()
   {
      ctr += (uint) 1;
   }
}

public class NonZeroCounter : Counter
{
   public NonZeroCounter(int startIndex) : this((uint) startIndex)
   {
   }

   private NonZeroCounter(UInt32 startIndex) : base(startIndex)
   {
   }
}
// Compilation produces a compiler warning like the following:
//    Type3.cs(37,14): warning CS3009: 'NonZeroCounter': base type 'Counter' is not
//            CLS-compliant
//    Type3.cs(7,14): (Location of symbol related to previous warning)
```

```
<Assembly: CLSCompliant(True)>

<CLSCompliant(False)> _
Public Class Counter
   Dim ctr As UInt32

   Public Sub New
      ctr = 0
   End Sub

   Protected Sub New(ctr As UInt32)
      ctr = ctr
   End Sub

   Public Overrides Function ToString() As String
      Return String.Format("{0}). ", ctr)
   End Function

   Public ReadOnly Property Value As UInt32
      Get
         Return ctr
      End Get
   End Property

   Public Sub Increment()
      ctr += CType(1, UInt32)
   End Sub
End Class

Public Class NonZeroCounter : Inherits Counter
   Public Sub New(startIndex As Integer)
      MyClass.New(CType(startIndex, UInt32))
   End Sub

   Private Sub New(startIndex As UInt32)
      MyBase.New(CType(startIndex, UInt32))
   End Sub
End Class
' Compilation produces a compiler warning like the following:
'    Type3.vb(34) : warning BC40026: 'NonZeroCounter' is not CLS-compliant
'    because it derives from 'Counter', which is not CLS-compliant.
'
'    Public Class NonZeroCounter : Inherits Counter
'                 ~~~~~~~~~~~~~~
```

Todos os tipos exibidos em assinaturas de membro, incluindo um tipo de retorno de método ou um tipo de propriedade, devem ser compatíveis com CLS. Além disso, para tipos genéricos:

- Todos os tipos que compõe um tipo genérico instanciado devem ser compatíveis com CLS.
- Todos os tipos usados como restrições em parâmetros genéricos devem ser compatíveis com CLS.

O Common Type System do .NET Framework inclui vários tipos internos que têm suporte direto do Common Language Runtime e são codificados especialmente nos metadados de um assembly. Desses tipos intrínsecos, os tipos listados na tabela a seguir são compatíveis com CLS.

TIPO COMPATÍVEL COM CLS	DESCRIÇÃO
Byte	Inteiro sem sinal de 8 bits
Int16	Inteiro com sinal de 16 bits

TIPO COMPATÍVEL COM CLS	DESCRIÇÃO
Int32	Inteiro com sinal de 32 bits
Int64	Inteiro com sinal de 64 bits
Single	Valor do ponto flutuante de precisão simples
Double	Valor do ponto flutuante de precisão dupla
Boolean	Tipo de valor `true` ou `false`
Char	unidade de código codificado UTF-16
Decimal	Número decimal de ponto não flutuante
IntPtr	Ponteiro ou identificador de um tamanho definido por plataforma
String	Coleção de zero, um ou mais objetos Char

Os tipos intrínsecos listados na tabela a seguir não são compatíveis com CLS.

TIPO NÃO COMPATÍVEL	DESCRIÇÃO	ALTERNATIVA COMPATÍVEL COM CLS
SByte	Tipo de dados inteiro com sinal de 8 bits	Int16
TypedReference	Ponteiro para um objeto e seu tipo de runtime	Nenhum
UInt16	Inteiro sem sinal de 16 bits	Int32
UInt32	Inteiro sem sinal de 32 bits	Int64
UInt64	Inteiro sem sinal de 64 bits	Int64 (pode estourar), BigInteger ou Double
UIntPtr	Ponteiro ou identificador sem sinal	IntPtr

A biblioteca de classes .NET Framework ou qualquer outra biblioteca de classes pode incluir outros tipos que não sejam compatíveis com CLS; por exemplo:

- Tipos de valor demarcado. O exemplo do C# a seguir cria uma classe que tem uma propriedade pública do tipo `int*` chamada `Value`. Como um `int*` é um tipo de valor demarcado, o compilador o sinaliza como incompatível com CLS.

```
using System;

[assembly:CLSCompliant(true)]

public unsafe class TestClass
{
   private int* val;

   public TestClass(int number)
   {
      val = (int*) number;
   }

   public int* Value {
      get { return val; }
   }
}
// The compiler generates the following output when compiling this example:
//        warning CS3003: Type of 'TestClass.Value' is not CLS-compliant
```

- Referências de tipo, que são constructos especiais que contêm referência a um objeto e referência a um tipo. As referências tipadas são representadas no .NET Framework pela classe TypedReference.

Se um tipo não for compatível com CLS, você deverá aplicar o atributo CLSCompliantAttribute com um valor `isCompliant` de `false` a ele. Para obter mais informações, consulte a seção O atributo CLSCompliantAttribute.

O exemplo a seguir ilustra o problema de conformidade com CLS em uma assinatura do método e em uma instanciação de tipo genérico. Ele define uma classe `InvoiceItem` com uma propriedade do tipo UInt32, uma propriedade do tipo `Nullable(Of UInt32)` e um construtor com parâmetros de tipo UInt32 e `Nullable(Of UInt32)`. Você recebe quatro avisos do compilador ao tentar de compilar esse exemplo.

```
using System;

[assembly: CLSCompliant(true)]

public class InvoiceItem
{
   private uint invId = 0;
   private uint itemId = 0;
   private Nullable<uint> qty;

   public InvoiceItem(uint sku, Nullable<uint> quantity)
   {
      itemId = sku;
      qty = quantity;
   }

   public Nullable<uint> Quantity
   {
      get { return qty; }
      set { qty = value; }
   }

   public uint InvoiceId
   {
      get { return invId; }
      set { invId = value; }
   }
}
// The attempt to compile the example displays the following output:
//    Type1.cs(13,23): warning CS3001: Argument type 'uint' is not CLS-compliant
//    Type1.cs(13,33): warning CS3001: Argument type 'uint?' is not CLS-compliant
//    Type1.cs(19,26): warning CS3003: Type of 'InvoiceItem.Quantity' is not CLS-compliant
//    Type1.cs(25,16): warning CS3003: Type of 'InvoiceItem.InvoiceId' is not CLS-compliant
```

```
<Assembly: CLSCompliant(True)>

Public Class InvoiceItem

   Private invId As UInteger = 0
   Private itemId As UInteger = 0
   Private qty AS Nullable(Of UInteger)

   Public Sub New(sku As UInteger, quantity As Nullable(Of UInteger))
      itemId = sku
      qty = quantity
   End Sub

   Public Property Quantity As Nullable(Of UInteger)
      Get
         Return qty
      End Get
      Set
         qty = value
      End Set
   End Property

   Public Property InvoiceId As UInteger
      Get
         Return invId
      End Get
      Set
         invId = value
      End Set
   End Property
End Class
' The attempt to compile the example displays output similar to the following:
'    Type1.vb(13) : warning BC40028: Type of parameter 'sku' is not CLS-compliant.
'
'       Public Sub New(sku As UInteger, quantity As Nullable(Of UInteger))
'                      ~~~
'    Type1.vb(13) : warning BC40041: Type 'UInteger' is not CLS-compliant.
'
'       Public Sub New(sku As UInteger, quantity As Nullable(Of UInteger))
'                                                             ~~~~~~~~
'    Type1.vb(18) : warning BC40041: Type 'UInteger' is not CLS-compliant.
'
'       Public Property Quantity As Nullable(Of UInteger)
'                                               ~~~~~~~~
'    Type1.vb(27) : warning BC40027: Return type of function 'InvoiceId' is not CLS-compliant.
'
'       Public Property InvoiceId As UInteger
'                       ~~~~~~~~~
```

Para eliminar os avisos do compilador, substitua os tipos não compatíveis com CLS na interface pública `InvoiceItem` por tipos compatíveis:

```
using System;

[assembly: CLSCompliant(true)]

public class InvoiceItem
{
   private uint invId = 0;
   private uint itemId = 0;
   private Nullable<int> qty;

   public InvoiceItem(int sku, Nullable<int> quantity)
   {
      if (sku <= 0)
         throw new ArgumentOutOfRangeException("The item number is zero or negative.");
      itemId = (uint) sku;

      qty = quantity;
   }

   public Nullable<int> Quantity
   {
      get { return qty; }
      set { qty = value; }
   }

   public int InvoiceId
   {
      get { return (int) invId; }
      set {
         if (value <= 0)
            throw new ArgumentOutOfRangeException("The invoice number is zero or negative.");
         invId = (uint) value; }
   }
}
```

```
<Assembly: CLSCompliant(True)>

Public Class InvoiceItem

   Private invId As UInteger = 0
   Private itemId As UInteger = 0
   Private qty AS Nullable(Of Integer)

   Public Sub New(sku As Integer, quantity As Nullable(Of Integer))
      If sku <= 0 Then
         Throw New ArgumentOutOfRangeException("The item number is zero or negative.")
      End If
      itemId = CUInt(sku)
      qty = quantity
   End Sub

   Public Property Quantity As Nullable(Of Integer)
      Get
         Return qty
      End Get
      Set
         qty = value
      End Set
   End Property

   Public Property InvoiceId As Integer
      Get
         Return CInt(invId)
      End Get
      Set
         invId = CUInt(value)
      End Set
   End Property
End Class
```

Além dos tipos específicos listados, algumas categorias de tipos não são compatíveis com CLS. Entre eles estão tipos de ponteiro não gerenciados e tipos de ponteiro de função. O exemplo a seguir gera um aviso do compilador porque ele usa um ponteiro para um inteiro a fim de criar uma matriz de inteiros.

```
using System;

[assembly: CLSCompliant(true)]

public class ArrayHelper
{
   unsafe public static Array CreateInstance(Type type, int* ptr, int items)
   {
      Array arr = Array.CreateInstance(type, items);
      int* addr = ptr;
      for (int ctr = 0; ctr < items; ctr++) {
          int value = *addr;
          arr.SetValue(value, ctr);
          addr++;
      }
      return arr;
   }
}
// The attempt to compile this example displays the following output:
//    UnmanagedPtr1.cs(8,57): warning CS3001: Argument type 'int*' is not CLS-compliant
```

Para classes abstratas compatíveis com CLS (ou seja, classes marcadas como `abstract` no C# ou como `MustInherit` no Visual Basic), todos os membros da classe também devem ser compatíveis com CLS.

Convenções de nomenclatura

Como algumas linguagens de programação não diferenciam maiúsculas de minúsculas, os identificadores (como nomes de namespaces, tipos e membros) devem se diferenciar além de maiúsculas e minúsculas. Dois identificadores serão considerados equivalentes se seus mapeamentos em minúsculas forem os mesmos. O exemplo do C# a seguir define duas classes públicas, `Person` e `person`. Como elas são diferentes apenas em maiúsculas e minúsculas, o compilador do C# as sinaliza como não compatíveis com CLS.

```
using System;

[assembly: CLSCompliant(true)]

public class Person : person
{

}

public class person
{

}
// Compilation produces a compiler warning like the following:
//    Naming1.cs(11,14): warning CS3005: Identifier 'person' differing
//                       only in case is not CLS-compliant
//    Naming1.cs(6,14): (Location of symbol related to previous warning)
```

Identificadores de linguagem de programação, como nomes de namespaces, tipos e membros, devem estar de acordo com o Padrão Unicode 3.0, Relatório Técnico 15, Anexo 7. Isso significa que:

- O primeiro caractere de um identificador pode ser qualquer letra maiúscula Unicode, letra minúscula, letra maiúscula do título, letra modificadora, outra letra ou o número da letra. Para obter informações sobre categorias de caracteres Unicode, consulte a enumeração System.Globalization.UnicodeCategory.
- Os caracteres subsequentes podem ser de qualquer uma das categorias, como o primeiro caractere e também podem incluir marcas sem espaçamento, marcas que combinam espaçamento, números decimais, pontuações de conector e códigos de formatação.

Antes de comparar identificadores, você deve filtrar códigos de formatação e converter os identificadores em Formulário C de Normalização de Unicode, porque um caractere único pode ser representado por várias unidades de código codificadas em UTF-16. As sequências de caracteres que produzem as mesmas unidades de código no Formulário C de Normalização de Unicode não são compatíveis com CLS. O exemplo a seguir define uma propriedade chamada `Å`, que consiste no caractere SÍMBOLO DE ANGSTROM (U+212B) e uma segunda propriedade chamada `Å`, que consiste no caractere LETRA LATINA MAIÚSCULA A COM ANEL SUPERIOR (U+00C5). Os compiladores do C# e do Visual Basic sinalizam o código-fonte como incompatível com CLS.

```
public class Size
{
   private double a1;
   private double a2;

   public double Å
   {
       get { return a1; }
       set { a1 = value; }
   }

   public double Å
   {
       get { return a2; }
       set { a2 = value; }
   }
}
// Compilation produces a compiler warning like the following:
//    Naming2a.cs(16,18): warning CS3005: Identifier 'Size.Å' differing only in case is not
//            CLS-compliant
//    Naming2a.cs(10,18): (Location of symbol related to previous warning)
//    Naming2a.cs(18,8): warning CS3005: Identifier 'Size.Å.get' differing only in case is not
//            CLS-compliant
//    Naming2a.cs(12,8): (Location of symbol related to previous warning)
//    Naming2a.cs(19,8): warning CS3005: Identifier 'Size.Å.set' differing only in case is not
//            CLS-compliant
//    Naming2a.cs(13,8): (Location of symbol related to previous warning)
```

```
<Assembly: CLSCompliant(True)>
Public Class Size
   Private a1 As Double
   Private a2 As Double

   Public Property Å As Double
       Get
          Return a1
       End Get
       Set
          a1 = value
       End Set
   End Property

   Public Property Å As Double
       Get
          Return a2
       End Get
       Set
          a2 = value
       End Set
   End Property
End Class
' Compilation produces a compiler warning like the following:
'    Naming1.vb(9) : error BC30269: 'Public Property Å As Double' has multiple definitions
'     with identical signatures.
'
'       Public Property Å As Double
'                       ~
```

Os nomes de membro em um escopo específico (como os namespaces em um assembly, os tipos em um namespace ou os membros em um tipo) devem ser exclusivos, exceto os nomes resolvidos por meio de sobrecarga. Esse requisito é mais rígido do que o do Common Type System, que permite que vários membros em um escopo tenham nomes idênticos desde que sejam tipos diferentes de membros (por exemplo, um é um método e outro é um campo). Em particular, para membros de tipo:

- Campos e tipos aninhados são diferenciados apenas por nome.
- Métodos, propriedades e eventos que tenham o mesmo nome devem ser diferentes além apenas do tipo de retorno.

O exemplo a seguir ilustra o requisito de que nomes de membros devem ser exclusivos dentro de seu escopo. Ele define uma classe chamada `Converter` que inclui quatro membros chamados `Conversion`. Três são métodos e um é uma propriedade. O método que inclui um parâmetro Int64 tem um nome exclusivo, mas os dois métodos com um parâmetro Int32 não têm, porque o valor retornado não é considerado parte da assinatura de um membro. A propriedade `Conversion` também viola esse requisito porque as propriedades não podem ter o mesmo nome dos métodos sobrecarregados.

```
using System;

[assembly: CLSCompliant(true)]

public class Converter
{
   public double Conversion(int number)
   {
      return (double) number;
   }

   public float Conversion(int number)
   {
      return (float) number;
   }

   public double Conversion(long number)
   {
      return (double) number;
   }

   public bool Conversion
   {
      get { return true; }
   }
}
// Compilation produces a compiler error like the following:
//    Naming3.cs(13,17): error CS0111: Type 'Converter' already defines a member called
//             'Conversion' with the same parameter types
//    Naming3.cs(8,18): (Location of symbol related to previous error)
//    Naming3.cs(23,16): error CS0102: The type 'Converter' already contains a definition for
//             'Conversion'
//    Naming3.cs(8,18): (Location of symbol related to previous error)
```

```
<Assembly: CLSCompliant(True)>

Public Class Converter
   Public Function Conversion(number As Integer) As Double
      Return CDbl(number)
   End Function

   Public Function Conversion(number As Integer) As Single
      Return CSng(number)
   End Function

   Public Function Conversion(number As Long) As Double
      Return CDbl(number)
   End Function

   Public ReadOnly Property Conversion As Boolean
      Get
         Return True
      End Get
   End Property
End Class
' Compilation produces a compiler error like the following:
'    Naming3.vb(8) : error BC30301: 'Public Function Conversion(number As Integer) As Double'
'                    and 'Public Function Conversion(number As Integer) As Single' cannot
'                    overload each other because they differ only by return types.
'
'       Public Function Conversion(number As Integer) As Double
'                       ~~~~~~~~~~
'    Naming3.vb(20) : error BC30260: 'Conversion' is already declared as 'Public Function
'                     Conversion(number As Integer) As Single' in this class.
'
'       Public ReadOnly Property Conversion As Boolean
'                                ~~~~~~~~~~
```

Linguagens individuais incluem palavras-chave exclusivas, portanto, linguagens que apontam para o Common Language Runtime também devem oferecer algum mecanismo para fazer referência a identificadores (como nomes de tipo) que coincidam com palavras-chave. Por exemplo, `case` é uma palavra-chave no C# e no Visual Basic. No entanto, o exemplo do Visual Basic a seguir pode remover a ambiguidade de uma classe chamada `case` da palavra-chave `case`, usando chaves de abertura e fechamento. Caso contrário, o exemplo produziria a mensagem de erro "A palavra-chave não é válida como um identificador", e não seria compilado.

```
Public Class [case]
   Private _id As Guid
   Private name As String

   Public Sub New(name As String)
      _id = Guid.NewGuid()
      Me.name = name
   End Sub

   Public ReadOnly Property ClientName As String
      Get
         Return name
      End Get
   End Property
End Class
```

O exemplo do C# a seguir pode criar uma instância da classe `case` usando o símbolo `@` para remover a ambiguidade do identificador da palavras-chave da linguagem. Sem ele, o compilador do C# exibiria duas mensagens de erro, "Tipo esperado" e "'Maiúsculas e minúsculas' do termo de expressão inválido".

```
using System;

public class Example
{
   public static void Main()
   {
      @case c = new @case("John");
      Console.WriteLine(c.ClientName);
   }
}
```

Conversão de tipos

A Common Language Specification define dois operadores de conversão:

- `op_Implicit`, que é usado para conversões de ampliação que não resultam em perda de dados ou precisão. Por exemplo, a estrutura Decimal inclui um operador `op_Implicit` sobrecarregado para converter valores de tipos integrais e valores Char em valores Decimal.
- `op_Explicit`, que é usado para conversões de redução que possam resultar em perda de magnitude (um valor é convertido em um valor com um intervalo menor) ou precisão. Por exemplo, a estrutura Decimal inclui um operador `op_Explicit` sobrecarregado para converter Double e valores Single em Decimal e para converter valores Decimal em valores inteiros, o Double, Single e Char.

No entanto, nem todas as linguagens dão suporte à sobrecarga de operador ou à definição de operadores personalizados. Se optar por implementar esses operadores de conversão, você também deverá fornecer uma maneira alternativa para realizar a conversão. Recomendamos que você forneça os métodos `From`*Xxx* e `To`*Xxx*.

O exemplo a seguir define conversões explícitas e implícitas compatíveis com CLS. Ele cria uma classe `UDouble` que representa um número de ponto flutuante de precisão dupla com sinal. Ele fornece conversões implícitas de `UDouble` em Double e conversões explícitas de `UDouble` em Single, de Double em `UDouble` e de Single em `UDouble`. Ele também define um método `ToDouble` como uma alternativa ao operador de conversão implícita e os métodos `ToSingle`, `FromDouble` e `FromSingle` como alternativas aos operadores de conversão explícita.

```
using System;

public struct UDouble
{
   private double number;

   public UDouble(double value)
   {
      if (value < 0)
         throw new InvalidCastException("A negative value cannot be converted to a UDouble.");

      number = value;
   }

   public UDouble(float value)
   {
      if (value < 0)
         throw new InvalidCastException("A negative value cannot be converted to a UDouble.");

      number = value;
   }

   public static readonly UDouble MinValue = (UDouble) 0.0;
   public static readonly UDouble MaxValue = (UDouble) Double.MaxValue;

   public static explicit operator Double(UDouble value)
   {
      return value.number;
```

```
    }

    public static implicit operator Single(UDouble value)
    {
       if (value.number > (double) Single.MaxValue)
          throw new InvalidCastException("A UDouble value is out of range of the Single type.");

       return (float) value.number;
    }

    public static explicit operator UDouble(double value)
    {
       if (value < 0)
          throw new InvalidCastException("A negative value cannot be converted to a UDouble.");

       return new UDouble(value);
    }

    public static implicit operator UDouble(float value)
    {
       if (value < 0)
          throw new InvalidCastException("A negative value cannot be converted to a UDouble.");

       return new UDouble(value);
    }

    public static Double ToDouble(UDouble value)
    {
       return (Double) value;
    }

    public static float ToSingle(UDouble value)
    {
       return (float) value;
    }

    public static UDouble FromDouble(double value)
    {
       return new UDouble(value);
    }

    public static UDouble FromSingle(float value)
    {
       return new UDouble(value);
    }
 }
```

```
Public Structure UDouble
   Private number As Double

   Public Sub New(value As Double)
      If value < 0 Then
         Throw New InvalidCastException("A negative value cannot be converted to a UDouble.")
      End If
      number = value
   End Sub

   Public Sub New(value As Single)
      If value < 0 Then
         Throw New InvalidCastException("A negative value cannot be converted to a UDouble.")
      End If
      number = value
   End Sub

   Public Shared ReadOnly MinValue As UDouble = CType(0.0, UDouble)
   Public Shared ReadOnly MaxValue As UDouble = Double.MaxValue

   Public Shared Widening Operator CType(value As UDouble) As Double
      Return value.number
   End Operator

   Public Shared Narrowing Operator CType(value As UDouble) As Single
      If value.number > CDbl(Single.MaxValue) Then
         Throw New InvalidCastException("A UDouble value is out of range of the Single type.")
      End If
      Return CSng(value.number)
   End Operator

   Public Shared Narrowing Operator CType(value As Double) As UDouble
      If value < 0 Then
         Throw New InvalidCastException("A negative value cannot be converted to a UDouble.")
      End If
      Return New UDouble(value)
   End Operator

   Public Shared Narrowing Operator CType(value As Single) As UDouble
      If value < 0 Then
         Throw New InvalidCastException("A negative value cannot be converted to a UDouble.")
      End If
      Return New UDouble(value)
   End Operator

   Public Shared Function ToDouble(value As UDouble) As Double
      Return CType(value, Double)
   End Function

   Public Shared Function ToSingle(value As UDouble) As Single
      Return CType(value, Single)
   End Function

   Public Shared Function FromDouble(value As Double) As UDouble
      Return New UDouble(value)
   End Function

   Public Shared Function FromSingle(value As Single) As UDouble
      Return New UDouble(value)
   End Function
End Structure
```

Matrizes

As matrizes compatíveis com CLS estão em conformidade com as seguintes regras:

- Todas as dimensões de uma matriz devem ter um limite inferior igual a zero. O exemplo a seguir cria uma

matriz não compatível com CLS com um limite inferior de um. Independentemente da presença do atributo CLSCompliantAttribute, o compilador não detecta se a matriz retornada pelo método `Numbers.GetTenPrimes` não é compatível com CLS.

```
[assembly: CLSCompliant(true)]

public class Numbers
{
   public static Array GetTenPrimes()
   {
      Array arr = Array.CreateInstance(typeof(Int32), new int[] {10}, new int[] {1});
      arr.SetValue(1, 1);
      arr.SetValue(2, 2);
      arr.SetValue(3, 3);
      arr.SetValue(5, 4);
      arr.SetValue(7, 5);
      arr.SetValue(11, 6);
      arr.SetValue(13, 7);
      arr.SetValue(17, 8);
      arr.SetValue(19, 9);
      arr.SetValue(23, 10);

      return arr;
   }
}
```

```
<Assembly: CLSCompliant(True)>

Public Class Numbers
   Public Shared Function GetTenPrimes() As Array
      Dim arr As Array = Array.CreateInstance(GetType(Int32), {10}, {1})
      arr.SetValue(1, 1)
      arr.SetValue(2, 2)
      arr.SetValue(3, 3)
      arr.SetValue(5, 4)
      arr.SetValue(7, 5)
      arr.SetValue(11, 6)
      arr.SetValue(13, 7)
      arr.SetValue(17, 8)
      arr.SetValue(19, 9)
      arr.SetValue(23, 10)

      Return arr
   End Function
End Class
```

- Todos os elementos de matriz devem consistir em tipos compatíveis com CLS. O exemplo a seguir define dois métodos que retornam matrizes não compatíveis com CLS. O primeiro retorna uma matriz de valores UInt32. O segundo retorna uma matriz Object que inclui valores Int32 e UInt32. Embora o compilador identifique a primeira matriz como não compatível devido ao seu tipo UInt32, ele não reconhece que a segunda matriz inclui elementos não compatíveis com CLS.

```
using System;

[assembly: CLSCompliant(true)]

public class Numbers
{
   public static UInt32[] GetTenPrimes()
   {
      uint[] arr = { 1u, 2u, 3u, 5u, 7u, 11u, 13u, 17u, 19u };
      return arr;
   }

   public static Object[] GetFivePrimes()
   {
      Object[] arr = { 1, 2, 3, 5u, 7u };
      return arr;
   }
}
// Compilation produces a compiler warning like the following:
//    Array2.cs(8,27): warning CS3002: Return type of 'Numbers.GetTenPrimes()' is not
//            CLS-compliant
```

```
<Assembly: CLSCompliant(True)>

Public Class Numbers
   Public Shared Function GetTenPrimes() As UInt32()
      Return { 1ui, 2ui, 3ui, 5ui, 7ui, 11ui, 13ui, 17ui, 19ui }
   End Function

   Public Shared Function GetFivePrimes() As Object()
      Dim arr() As Object = { 1, 2, 3, 5ui, 7ui }
      Return arr
   End Function
End Class
' Compilation produces a compiler warning like the following:
'    warning BC40027: Return type of function 'GetTenPrimes' is not CLS-compliant.
'
'       Public Shared Function GetTenPrimes() As UInt32()
'                              ~~~~~~~~~~~~
```

- A resolução de sobrecarga para métodos que tenham parâmetros de matriz se baseia no fato de que são matrizes e em seu tipo de elemento. Por esse motivo, a seguinte definição de um método `GetSquares` sobrecarregado é compatível com CLS.

```
using System;
using System.Numerics;

[assembly: CLSCompliant(true)]

public class Numbers
{
   public static byte[] GetSquares(byte[] numbers)
   {
      byte[] numbersOut = new byte[numbers.Length];
      for (int ctr = 0; ctr < numbers.Length; ctr++) {
         int square = ((int) numbers[ctr]) * ((int) numbers[ctr]);
         if (square <= Byte.MaxValue)
            numbersOut[ctr] = (byte) square;
         // If there's an overflow, assign MaxValue to the corresponding
         // element.
         else
            numbersOut[ctr] = Byte.MaxValue;

      }
      return numbersOut;
   }

   public static BigInteger[] GetSquares(BigInteger[] numbers)
   {
      BigInteger[] numbersOut = new BigInteger[numbers.Length];
      for (int ctr = 0; ctr < numbers.Length; ctr++)
         numbersOut[ctr] = numbers[ctr] * numbers[ctr];

      return numbersOut;
   }
}
```

```
Imports System.Numerics

<Assembly: CLSCompliant(True)>

Public Module Numbers
   Public Function GetSquares(numbers As Byte()) As Byte()
      Dim numbersOut(numbers.Length - 1) As Byte
      For ctr As Integer = 0 To numbers.Length - 1
         Dim square As Integer = (CInt(numbers(ctr)) * CInt(numbers(ctr)))
         If square <= Byte.MaxValue Then
            numbersOut(ctr) = CByte(square)
         ' If there's an overflow, assign MaxValue to the corresponding
         ' element.
         Else
            numbersOut(ctr) = Byte.MaxValue
         End If
      Next
      Return numbersOut
   End Function

   Public Function GetSquares(numbers As BigInteger()) As BigInteger()
      Dim numbersOut(numbers.Length - 1) As BigInteger
      For ctr As Integer = 0 To numbers.Length - 1
         numbersOut(ctr) = numbers(ctr) * numbers(ctr)
      Next
      Return numbersOut
   End Function
End Module
```

Interfaces

Interfaces compatíveis com CLS podem definir propriedades, eventos e métodos virtuais (métodos sem

implementação). Uma interface compatível com CLS não pode ter nenhum dos seguintes itens:

- Métodos estáticos ou campos estáticos. Os compiladores do C# e do Visual Basic gerarão erros de compilador se você definir um membro estático em uma interface.
- Campos. Os compiladores do C# e do Visual Basic gerarão erros de compilador se você definir um campo em uma interface.
- Métodos que não são compatíveis com CLS. Por exemplo, a definição a seguir da interface inclui um método, `INumber.GetUnsigned`, que está marcado como não compatível com CLS. Este exemplo gera um aviso do compilador.

```
using System;

[assembly:CLSCompliant(true)]

public interface INumber
{
   int Length();
   [CLSCompliant(false)] ulong GetUnsigned();
}
// Attempting to compile the example displays output like the following:
//    Interface2.cs(8,32): warning CS3010: 'INumber.GetUnsigned()': CLS-compliant interfaces
//            must have only CLS-compliant members
```

```
<Assembly: CLSCompliant(True)>

Public Interface INumber
   Function Length As Integer

   <CLSCompliant(False)> Function GetUnsigned As ULong
End Interface
' Attempting to compile the example displays output like the following:
'    Interface2.vb(9) : warning BC40033: Non CLS-compliant 'function' is not allowed in a
'    CLS-compliant interface.
'
'       <CLSCompliant(False)> Function GetUnsigned As ULong
'                                      ~~~~~~~~~~~
```

  Devido a essa regra, os tipos compatíveis com CLS não são necessários para implementar membros não compatíveis com CLS. Se uma estrutura compatível com CLS expuser uma classe que implementa uma interface não compatível com CLS, ela também deverá fornecer implementações concretas de todos os membros não compatíveis com CLS.

Compiladores de linguagem compatíveis com CLS também devem permitir que uma classe forneça implementações separadas dos membros com o mesmo nome e a assinatura em várias interfaces. O C# e o Visual Basic dão suporte a implementações explícitas de interface para fornecer implementações diferentes de métodos com nomes idênticos. O Visual Basic também dá suporte à palavra-chave `Implements`, que permite que você designe explicitamente qual interface e membro um determinado membro implementa. O exemplo a seguir ilustra esse cenário, definindo uma classe `Temperature` que implementa as interfaces `ICelsius` e `IFahrenheit` como implementações explícitas de interface.

```
using System;

[assembly: CLSCompliant(true)]

public interface IFahrenheit
{
   decimal GetTemperature();
}

public interface ICelsius
{
   decimal GetTemperature();
}

public class Temperature : ICelsius, IFahrenheit
{
   private decimal _value;

   public Temperature(decimal value)
   {
      // We assume that this is the Celsius value.
      _value = value;
   }

   decimal IFahrenheit.GetTemperature()
   {
      return _value * 9 / 5 + 32;
   }

   decimal ICelsius.GetTemperature()
   {
      return _value;
   }
}
public class Example
{
   public static void Main()
   {
      Temperature temp = new Temperature(100.0m);
      ICelsius cTemp = temp;
      IFahrenheit fTemp = temp;
      Console.WriteLine("Temperature in Celsius: {0} degrees",
                        cTemp.GetTemperature());
      Console.WriteLine("Temperature in Fahrenheit: {0} degrees",
                        fTemp.GetTemperature());
   }
}
// The example displays the following output:
//       Temperature in Celsius: 100.0 degrees
//       Temperature in Fahrenheit: 212.0 degrees
```

```
<Assembly: CLSCompliant(True)>

Public Interface IFahrenheit
   Function GetTemperature() As Decimal
End Interface

Public Interface ICelsius
   Function GetTemperature() As Decimal
End Interface

Public Class Temperature : Implements ICelsius, IFahrenheit
   Private _value As Decimal

   Public Sub New(value As Decimal)
      ' We assume that this is the Celsius value.
      _value = value
   End Sub

   Public Function GetFahrenheit() As Decimal _
          Implements IFahrenheit.GetTemperature
      Return _value * 9 / 5 + 32
   End Function

   Public Function GetCelsius() As Decimal _
          Implements ICelsius.GetTemperature
      Return _value
   End Function
End Class

Module Example
   Public Sub Main()
      Dim temp As New Temperature(100.0d)
      Console.WriteLine("Temperature in Celsius: {0} degrees",
                        temp.GetCelsius())
      Console.WriteLine("Temperature in Fahrenheit: {0} degrees",
                        temp.GetFahrenheit())
   End Sub
End Module
' The example displays the following output:
'       Temperature in Celsius: 100.0 degrees
'       Temperature in Fahrenheit: 212.0 degrees
```

Enumerações

Enumerações compatíveis com CLS devem seguir estas regras:

- O tipo subjacente da enumeração deve ser um inteiro intrínseco compatível com CLS (Byte, Int16, Int32 ou Int64). Por exemplo, o código a seguir tenta definir uma enumeração cujo tipo subjacente é UInt32 e gera um aviso do compilador.

```
using System;

[assembly: CLSCompliant(true)]

public enum Size : uint {
   Unspecified = 0,
   XSmall = 1,
   Small = 2,
   Medium = 3,
   Large = 4,
   XLarge = 5
};

public class Clothing
{
   public string Name;
   public string Type;
   public string Size;
}
// The attempt to compile the example displays a compiler warning like the following:
//    Enum3.cs(6,13): warning CS3009: 'Size': base type 'uint' is not CLS-compliant
```

```
<Assembly: CLSCompliant(True)>

Public Enum Size As UInt32
   Unspecified = 0
   XSmall = 1
   Small = 2
   Medium = 3
   Large = 4
   XLarge = 5
End Enum

Public Class Clothing
   Public Name As String
   Public Type As String
   Public Size As Size
End Class
' The attempt to compile the example displays a compiler warning like the following:
'    Enum3.vb(6) : warning BC40032: Underlying type 'UInt32' of Enum is not CLS-compliant.
'
'    Public Enum Size As UInt32
'                ~~~~
```

- Um tipo de enumeração deve ter um campo de instância única chamado `Value__` que foi marcado com o atributo FieldAttributes.RTSpecialName. Isso permite que você referencie o valor do campo implicitamente.
- Uma enumeração inclui campos estáticos literais, cujos tipos correspondem ao tipo da própria enumeração. Por exemplo, se você definir uma enumeração `State` com valores de `State.On` e `State.Off`, `State.On` e `State.Off` serão campos literais estáticos cujo tipo será `State`.
- Há dois tipos de enumeração:
  - Uma enumeração que representa um conjunto de valores mutuamente excludentes, valores inteiros nomeados. Esse tipo de enumeração é indicado pela ausência do atributo personalizado System.FlagsAttribute.
  - Uma enumeração que representa um conjunto de sinalizadores de bit que podem ser combinados para produzir um valor sem nome. Esse tipo de enumeração é indicado pela presença do atributo personalizado System.FlagsAttribute.

Para obter mais informações, consulte a documentação da estrutura Enum.

- O valor de uma enumeração não está limitado ao intervalo de seus valores especificados. Em outras palavras, o intervalo de valores em uma enumeração é o intervalo de seu valor subjacente. Você pode usar o método Enum.IsDefined para determinar se um valor especificado é membro de uma enumeração.

Membros de tipo em geral

O Common Language Specification requer que todos os campos e métodos sejam acessados como membros de uma classe específica. Portanto, campos e métodos estáticos globais (ou seja, campos ou métodos estáticos que são definidos independentemente de um tipo) não são compatíveis com CLS. Se você tentar incluir um campo ou um método global em seu código fonte, os compiladores do C# e do Visual Basic gerarão um erro de compilador.

A Common Language Specification dá suporte somente à convenção de chamada gerenciada padrão. Ela não dá suporte a convenções e métodos de chamada não gerenciados com listas de argumentos de variável marcadas com a palavra-chave `varargs`. Para as listas de argumentos variáveis que são compatíveis com a convenção de chamada gerenciada padrão, use o atributo ParamArrayAttribute ou a implementação da linguagem individual, como a palavra-chave `params` no C# e a palavra-chave `ParamArray` no Visual Basic.

Acessibilidade de membro

A substituição de um membro herdado não pode alterar a acessibilidade desse membro. Por exemplo, um método público em uma classe base não pode ser substituído por um método privado em uma classe derivada. Há uma exceção: um membro `protected internal` (em C#) ou `Protected Friend` (em Visual Basic) em um assembly que é substituído por um tipo em um assembly diferente. Nesse caso, a acessibilidade da substituição é `Protected`.

O exemplo a seguir ilustra o erro que é gerado quando o atributo CLSCompliantAttribute é definido como `true` e `Human`, que é uma classe derivada de `Animal` tenta alterar a acessibilidade da propriedade `Species` de chave pública para privada. O exemplo será compilado com êxito se sua acessibilidade for alterada para pública.

```
using System;

[assembly: CLSCompliant(true)]

public class Animal
{
   private string _species;

   public Animal(string species)
   {
      _species = species;
   }

   public virtual string Species
   {
      get { return _species; }
   }

   public override string ToString()
   {
      return _species;
   }
}

public class Human : Animal
{
   private string _name;

   public Human(string name) : base("Homo Sapiens")
   {
      _name = name;
   }

   public string Name
   {
      get { return _name; }
   }

   private override string Species
   {
      get { return base.Species; }
   }

   public override string ToString()
   {
      return _name;
   }
}

public class Example
{
   public static void Main()
   {
      Human p = new Human("John");
      Console.WriteLine(p.Species);
      Console.WriteLine(p.ToString());
   }
}
// The example displays the following output:
//    error CS0621: 'Human.Species': virtual or abstract members cannot be private
```

```
<Assembly: CLSCompliant(True)>

Public Class Animal
   Private _species As String

   Public Sub New(species As String)
      _species = species
   End Sub

   Public Overridable ReadOnly Property Species As String
      Get
         Return _species
      End Get
   End Property

   Public Overrides Function ToString() As String
      Return _species
   End Function
End Class

Public Class Human : Inherits Animal
   Private _name As String

   Public Sub New(name As String)
      MyBase.New("Homo Sapiens")
      _name = name
   End Sub

   Public ReadOnly Property Name As String
      Get
         Return _name
      End Get
   End Property

   Private Overrides ReadOnly Property Species As String
      Get
         Return MyBase.Species
      End Get
   End Property

   Public Overrides Function ToString() As String
      Return _name
   End Function
End Class

Public Module Example
   Public Sub Main()
      Dim p As New Human("John")
      Console.WriteLine(p.Species)
      Console.WriteLine(p.ToString())
   End Sub
End Module
' The example displays the following output:
'     'Private Overrides ReadOnly Property Species As String' cannot override
'     'Public Overridable ReadOnly Property Species As String' because
'      they have different access levels.
'
'         Private Overrides ReadOnly Property Species As String
```

Os tipos na assinatura de um membro deverão ser acessíveis sempre que o membro for acessível. Por exemplo, isso significa que um membro público não pode incluir um parâmetro cujo tipo seja privado, protegido ou interno. O exemplo a seguir ilustra o erro do compilador que resulta quando um construtor de classe `StringWrapper` expõe um valor de enumeração `StringOperationType` interno que determina como um valor de cadeia de caracteres deve ser encapsulado.

```
using System;
using System.Text;

public class StringWrapper
{
   string internalString;
   StringBuilder internalSB = null;
   bool useSB = false;

   public StringWrapper(StringOperationType type)
   {
      if (type == StringOperationType.Normal) {
         useSB = false;
      }
      else {
         useSB = true;
         internalSB = new StringBuilder();
      }
   }

   // The remaining source code...
}

internal enum StringOperationType { Normal, Dynamic }
// The attempt to compile the example displays the following output:
//    error CS0051: Inconsistent accessibility: parameter type
//            'StringOperationType' is less accessible than method
//            'StringWrapper.StringWrapper(StringOperationType)'
```

```
Imports System.Text

<Assembly:CLSCompliant(True)>

Public Class StringWrapper

   Dim internalString As String
   Dim internalSB As StringBuilder = Nothing
   Dim useSB As Boolean = False

   Public Sub New(type As StringOperationType)
      If type = StringOperationType.Normal Then
         useSB = False
      Else
         internalSB = New StringBuilder()
         useSB = True
      End If
   End Sub

   ' The remaining source code...
End Class

Friend Enum StringOperationType As Integer
   Normal = 0
   Dynamic = 1
End Enum
' The attempt to compile the example displays the following output:
'    error BC30909: 'type' cannot expose type 'StringOperationType'
'     outside the project through class 'StringWrapper'.
'
'       Public Sub New(type As StringOperationType)
'                                ~~~~~~~~~~~~~~~~~~~
```

Tipos e membros genéricos

Tipos aninhados sempre têm pelo menos o mesmo número de parâmetros genéricos do tipo delimitador. Eles

correspondem por posição aos parâmetros genéricos no tipo delimitador. O tipo genérico também pode incluir novos parâmetros genéricos.

A relação entre os parâmetros de tipo genérico de um tipo de contenção e seus tipos aninhados pode ser ocultada pela sintaxe de linguagens individuais. No exemplo a seguir, um tipo genérico `Outer<T>` contém duas classes aninhadas, `Inner1A` e `Inner1B<U>`. As chamadas para o método `ToString`, que cada classe herda de Object.ToString, mostra que cada classe aninhada inclui parâmetros de tipo de sua classe de contenção.

```
using System;

[assembly:CLSCompliant(true)]

public class Outer<T>
{
   T value;

   public Outer(T value)
   {
      this.value = value;
   }

   public class Inner1A : Outer<T>
   {
      public Inner1A(T value) : base(value)
      {  }
   }

   public class Inner1B<U> : Outer<T>
   {
      U value2;

      public Inner1B(T value1, U value2) : base(value1)
      {
         this.value2 = value2;
      }
   }
}

public class Example
{
   public static void Main()
   {
      var inst1 = new Outer<String>("This");
      Console.WriteLine(inst1);

      var inst2 = new Outer<String>.Inner1A("Another");
      Console.WriteLine(inst2);

      var inst3 = new Outer<String>.Inner1B<int>("That", 2);
      Console.WriteLine(inst3);
   }
}
// The example displays the following output:
//       Outer`1[System.String]
//       Outer`1+Inner1A[System.String]
//       Outer`1+Inner1B`1[System.String,System.Int32]
```

```
<Assembly:CLSCompliant(True)>

Public Class Outer(Of T)
   Dim value As T

   Public Sub New(value As T)
      Me.value = value
   End Sub

   Public Class Inner1A : Inherits Outer(Of T)
      Public Sub New(value As T)
         MyBase.New(value)
      End Sub
   End Class

   Public Class Inner1B(Of U) : Inherits Outer(Of T)
      Dim value2 As U

      Public Sub New(value1 As T, value2 As U)
         MyBase.New(value1)
         Me.value2 = value2
      End Sub
   End Class
End Class

Public Module Example
   Public Sub Main()
      Dim inst1 As New Outer(Of String)("This")
      Console.WriteLine(inst1)

      Dim inst2 As New Outer(Of String).Inner1A("Another")
      Console.WriteLine(inst2)

      Dim inst3 As New Outer(Of String).Inner1B(Of Integer)("That", 2)
      Console.WriteLine(inst3)
   End Sub
End Module
' The example displays the following output:
'       Outer`1[System.String]
'       Outer`1+Inner1A[System.String]
'       Outer`1+Inner1B`1[System.String,System.Int32]
```

Os nomes de tipo genérico são codificados no formato *nome*`*n*, em que *nome* é o nome do tipo, ` é um literal de caractere e *n* é o número de parâmetros declarados no tipo ou, para tipos genéricos aninhados, o número de parâmetros de tipo recém-introduzidos. Essa codificação de nomes de tipo genéricos é principalmente de interesse de desenvolvedores que usam a reflexão para acessar tipos genéricos compatíveis com CLS em uma biblioteca.

Se as restrições forem aplicadas a um tipo genérico, qualquer tipo usado como restrição também deverá ser compatível com CLS. O exemplo a seguir define uma classe chamada `BaseClass` que não é compatível com CLS e uma classe genérica chamada `BaseCollection` cujo parâmetro de tipo deve derivar de `BaseClass`. Mas como `BaseClass` não é compatível com CLS, o compilador emite um aviso.

```
using System;

[assembly:CLSCompliant(true)]

[CLSCompliant(false)] public class BaseClass
{}


public class BaseCollection<T> where T : BaseClass
{}
// Attempting to compile the example displays the following output:
//    warning CS3024: Constraint type 'BaseClass' is not CLS-compliant
```

```
<Assembly: CLSCompliant(True)>

<CLSCompliant(False)> Public Class BaseClass
End Class


Public Class BaseCollection(Of T As BaseClass)
End Class
' Attempting to compile the example displays the following output:
'    warning BC40040: Generic parameter constraint type 'BaseClass' is not
'    CLS-compliant.
'
'    Public Class BaseCollection(Of T As BaseClass)
'                                       ~~~~~~~~~
```

Se um tipo genérico for derivado de um tipo de base genérico, ele deverá redeclarar todas as restrições, para assegurar que as restrições no tipo de base também sejam atendidas. O exemplo a seguir define um `Number<T>` que pode representar qualquer tipo numérico. Ele também define uma classe `FloatingPoint<T>` que representa um valor de ponto flutuante. No entanto, o código-fonte falha na compilação porque não aplica a restrição em `Number<T>` (esse T deve ser um tipo de valor) a `FloatingPoint<T>`.

```
using System;

[assembly:CLSCompliant(true)]

public class Number<T> where T : struct
{
   // use Double as the underlying type, since its range is a superset of
   // the ranges of all numeric types except BigInteger.
   protected double number;

   public Number(T value)
   {
      try {
         this.number = Convert.ToDouble(value);
      }
      catch (OverflowException e) {
         throw new ArgumentException("value is too large.", e);
      }
      catch (InvalidCastException e) {
         throw new ArgumentException("The value parameter is not numeric.", e);
      }
   }

   public T Add(T value)
   {
      return (T) Convert.ChangeType(number + Convert.ToDouble(value), typeof(T));
   }

   public T Subtract(T value)
   {
      return (T) Convert.ChangeType(number - Convert.ToDouble(value), typeof(T));
   }
}

public class FloatingPoint<T> : Number<T>
{
   public FloatingPoint(T number) : base(number)
   {
      if (typeof(float) == number.GetType() ||
          typeof(double) == number.GetType() ||
          typeof(decimal) == number.GetType())
         this.number = Convert.ToDouble(number);
      else
         throw new ArgumentException("The number parameter is not a floating-point number.");
   }
}
// The attempt to comple the example displays the following output:
//       error CS0453: The type 'T' must be a non-nullable value type in
//               order to use it as parameter 'T' in the generic type or method 'Number<T>'
```

```
<Assembly:CLSCompliant(True)>

Public Class Number(Of T As Structure)
   ' Use Double as the underlying type, since its range is a superset of
   ' the ranges of all numeric types except BigInteger.
   Protected number As Double

   Public Sub New(value As T)
      Try
         Me.number = Convert.ToDouble(value)
      Catch e As OverflowException
         Throw New ArgumentException("value is too large.", e)
      Catch e As InvalidCastException
         Throw New ArgumentException("The value parameter is not numeric.", e)
      End Try
   End Sub

   Public Function Add(value As T) As T
      Return CType(Convert.ChangeType(number + Convert.ToDouble(value), GetType(T)), T)
   End Function

   Public Function Subtract(value As T) As T
      Return CType(Convert.ChangeType(number - Convert.ToDouble(value), GetType(T)), T)
   End Function
End Class

Public Class FloatingPoint(Of T) : Inherits Number(Of T)
   Public Sub New(number As T)
      MyBase.New(number)
      If TypeOf number Is Single Or
               TypeOf number Is Double Or
               TypeOf number Is Decimal Then
         Me.number = Convert.ToDouble(number)
      Else
         throw new ArgumentException("The number parameter is not a floating-point number.")
      End If
   End Sub
End Class
' The attempt to comple the example displays the following output:
'    error BC32105: Type argument 'T' does not satisfy the 'Structure'
'    constraint for type parameter 'T'.
'
'    Public Class FloatingPoint(Of T) : Inherits Number(Of T)
'                                                        ~
```

O exemplo será compilado com êxito se a restrição for adicionada à classe `FloatingPoint<T>`.

```
using System;

[assembly:CLSCompliant(true)]


public class Number<T> where T : struct
{
   // use Double as the underlying type, since its range is a superset of
   // the ranges of all numeric types except BigInteger.
   protected double number;

   public Number(T value)
   {
      try {
         this.number = Convert.ToDouble(value);
      }
      catch (OverflowException e) {
         throw new ArgumentException("value is too large.", e);
      }
      catch (InvalidCastException e) {
         throw new ArgumentException("The value parameter is not numeric.", e);
      }
   }

   public T Add(T value)
   {
      return (T) Convert.ChangeType(number + Convert.ToDouble(value), typeof(T));
   }

   public T Subtract(T value)
   {
      return (T) Convert.ChangeType(number - Convert.ToDouble(value), typeof(T));
   }
}

public class FloatingPoint<T> : Number<T> where T : struct
{
   public FloatingPoint(T number) : base(number)
   {
      if (typeof(float) == number.GetType() ||
          typeof(double) == number.GetType() ||
          typeof(decimal) == number.GetType())
         this.number = Convert.ToDouble(number);
      else
         throw new ArgumentException("The number parameter is not a floating-point number.");
   }
}
```

```
<Assembly:CLSCompliant(True)>

Public Class Number(Of T As Structure)
   ' Use Double as the underlying type, since its range is a superset of
   ' the ranges of all numeric types except BigInteger.
   Protected number As Double

   Public Sub New(value As T)
      Try
         Me.number = Convert.ToDouble(value)
      Catch e As OverflowException
         Throw New ArgumentException("value is too large.", e)
      Catch e As InvalidCastException
         Throw New ArgumentException("The value parameter is not numeric.", e)
      End Try
   End Sub

   Public Function Add(value As T) As T
      Return CType(Convert.ChangeType(number + Convert.ToDouble(value), GetType(T)), T)
   End Function

   Public Function Subtract(value As T) As T
      Return CType(Convert.ChangeType(number - Convert.ToDouble(value), GetType(T)), T)
   End Function
End Class

Public Class FloatingPoint(Of T As Structure) : Inherits Number(Of T)
   Public Sub New(number As T)
      MyBase.New(number)
      If TypeOf number Is Single Or
               TypeOf number Is Double Or
               TypeOf number Is Decimal Then
         Me.number = Convert.ToDouble(number)
      Else
         throw new ArgumentException("The number parameter is not a floating-point number.")
      End If
   End Sub
End Class
```

A Common Language Specification impõe um modelo por instanciação conservador para tipos aninhados e membros protegidos. Tipos genéricos abertos não podem expor campos ou membros com assinaturas que contenham uma instanciação específica de um tipo genérico aninhado, protegido. Tipos não genéricos que estendam uma instanciação específica de uma interface ou classe base genérica não podem expor campos ou membros com assinaturas que contenham uma instanciação diferente de um tipo genérico aninhado e protegido.

O exemplo a seguir define um tipo genérico, `C1<T>` (ou `C1(Of T)` no Visual Basic) e uma classe protegida, `C1<T>.N` (ou `C1(Of T).N` no Visual Basic). `C1<T>` possui dois métodos, `M1` e `M2`. No entanto, `M1` é sem conformidade com CLS porque tenta retornar um objeto `C1<int>.N` (ou `C1(Of Integer).N`) de C1<T> (ou `C1(Of T)`). Uma segunda classe, `C2`, é derivada de `C1<long>` (ou de `C1(Of Long)`). Tem dois métodos, `M3` e `M4`. No entanto, `M3` não é compatível com CLS porque tenta retornar um objeto `C1<int>.N` (ou `C1(Of Integer).N`) de uma subclasse de `C1<long>`. Os compiladores de linguagens podem ser ainda mais restritivos. Neste exemplo, o Visual Basic exibe um erro ao tentar compilar `M4`.

```
using System;

[assembly:CLSCompliant(true)]

public class C1<T>
{
   protected class N { }

   protected void M1(C1<int>.N n) { } // Not CLS-compliant - C1<int>.N not
                                     // accessible from within C1<T> in all
                                     // languages
   protected void M2(C1<T>.N n) { }   // CLS-compliant – C1<T>.N accessible
                                     // inside C1<T>
}

public class C2 : C1<long>
{
   protected void M3(C1<int>.N n) { }  // Not CLS-compliant – C1<int>.N is not
                                       // accessible in C2 (extends C1<long>)

   protected void M4(C1<long>.N n) { } // CLS-compliant, C1<long>.N is
                                       // accessible in C2 (extends C1<long>)
}
// Attempting to compile the example displays output like the following:
//       Generics4.cs(9,22): warning CS3001: Argument type 'C1<int>.N' is not CLS-compliant
//       Generics4.cs(18,22): warning CS3001: Argument type 'C1<int>.N' is not CLS-compliant
```

```
<Assembly:CLSCompliant(True)>

Public Class C1(Of T)
   Protected Class N
   End Class

   Protected Sub M1(n As C1(Of Integer).N)   ' Not CLS-compliant - C1<int>.N not
                                             ' accessible from within C1(Of T) in all
   End Sub                                   ' languages


   Protected Sub M2(n As C1(Of T).N)     ' CLS-compliant – C1(Of T).N accessible
   End Sub                               ' inside C1(Of T)
End Class

Public Class C2 : Inherits C1(Of Long)
   Protected Sub M3(n As C1(Of Integer).N)   ' Not CLS-compliant – C1(Of Integer).N is not
   End Sub                                   ' accessible in C2 (extends C1(Of Long))

   Protected Sub M4(n As C1(Of Long).N)
   End Sub
End Class
' Attempting to compile the example displays output like the following:
'    error BC30508: 'n' cannot expose type 'C1(Of Integer).N' in namespace
'    '<Default>' through class 'C1'.
'
'       Protected Sub M1(n As C1(Of Integer).N)   ' Not CLS-compliant - C1<int>.N not
'                             ~~~~~~~~~~~~~~~~~
'    error BC30389: 'C1(Of T).N' is not accessible in this context because
'    it is 'Protected'.
'
'       Protected Sub M3(n As C1(Of Integer).N)   ' Not CLS-compliant - C1(Of Integer).N is not
'
'                             ~~~~~~~~~~~~~~~~~
'
'    error BC30389: 'C1(Of T).N' is not accessible in this context because it is 'Protected'.
'
'       Protected Sub M4(n As C1(Of Long).N)
'                             ~~~~~~~~~~~~~
```

Construtores

Os construtores em classes compatíveis com CLS e em estruturas devem seguir estas regras:

- Um construtor de uma classe derivada deve chamar o construtor de instância de sua classe base antes de acessar quaisquer dados de instância herdados. Esse requisito deve-se ao fato de que construtores de classe base não são herdados por suas classes derivadas. Essa regra não se aplica a estruturas que não dão suporte a herança direta.

  Normalmente, os compiladores aplicam essa regra independentemente da conformidade com CLS, conforme mostrado no exemplo a seguir. Ele cria uma classe `Doctor` que é derivada de uma classe `Person`, mas a classe `Doctor` falha ao chamar o construtor de classe `Person` para inicializar campos herdados da instância.

```
using System;

[assembly: CLSCompliant(true)]

public class Person
{
   private string fName, lName, _id;

   public Person(string firstName, string lastName, string id)
   {
      if (String.IsNullOrEmpty(firstName + lastName))
         throw new ArgumentNullException("Either a first name or a last name must be provided.");

      fName = firstName;
      lName = lastName;
      _id = id;
   }

   public string FirstName
   {
      get { return fName; }
   }

   public string LastName
   {
      get { return lName; }
   }

   public string Id
   {
      get { return _id; }
   }

   public override string ToString()
   {
      return String.Format("{0}{1}{2}", fName,
                           String.IsNullOrEmpty(fName) ?  "" : " ",
                           lName);
   }
}

public class Doctor : Person
{
   public Doctor(string firstName, string lastName, string id)
   {
   }

   public override string ToString()
   {
      return "Dr. " + base.ToString();
   }
}
// Attempting to compile the example displays output like the following:
//    ctor1.cs(45,11): error CS1729: 'Person' does not contain a constructor that takes 0
//            arguments
//    ctor1.cs(10,11): (Location of symbol related to previous error)
```

```
<Assembly: CLSCompliant(True)>

Public Class Person
   Private fName, lName, _id As String

   Public Sub New(firstName As String, lastName As String, id As String)
      If String.IsNullOrEmpty(firstName + lastName) Then
         Throw New ArgumentNullException("Either a first name or a last name must be provided.")
      End If

      fName = firstName
      lName = lastName
      _id = id
   End Sub

   Public ReadOnly Property FirstName As String
      Get
         Return fName
      End Get
   End Property

   Public ReadOnly Property LastName As String
      Get
         Return lName
      End Get
   End Property

   Public ReadOnly Property Id As String
      Get
         Return _id
      End Get
   End Property

   Public Overrides Function ToString() As String
      Return String.Format("{0}{1}{2}", fName,
                           If(String.IsNullOrEmpty(fName), "", " "),
                           lName)
   End Function
End Class

Public Class Doctor : Inherits Person
   Public Sub New(firstName As String, lastName As String, id As String)
   End Sub

   Public Overrides Function ToString() As String
      Return "Dr. " + MyBase.ToString()
   End Function
End Class
' Attempting to compile the example displays output like the following:
'    Ctor1.vb(46) : error BC30148: First statement of this 'Sub New' must be a call
'    to 'MyBase.New' or 'MyClass.New' because base class 'Person' of 'Doctor' does
'    not have an accessible 'Sub New' that can be called with no arguments.
'
'       Public Sub New()
'                  ~~~
```

- Um construtor de objeto não pode ser chamado, exceto para criar um objeto. Além disso, um objeto não pode ser inicializado duas vezes. Por exemplo, isso significa que Object.MemberwiseClone e métodos de desserialização como BinaryFormatter.Deserialize não devem chamar construtores.

Propriedades

As propriedades em tipos compatíveis com CLS devem seguir estas regras:

- Uma propriedade deve ter um setter, um getter ou ambos. Em um assembly, eles são implementados como métodos especiais, o que significa que aparecerão como métodos separados (o getter é chamado de `get_`

propertyname e o setter é `set_` *propertyname*) marcado como `SpecialName` nos metadados do assembly. Os compiladores do C# e do Visual Basic aplicam automaticamente essa regra, sem a necessidade de aplicar o atributo CLSCompliantAttribute.

- Um tipo de propriedade é o tipo de retorno do getter da propriedade e o último argumento do setter. Esses tipos devem ser compatíveis com CLS e os argumentos não podem ser atribuídos à propriedade por referência (ou seja, não podem ser ponteiros gerenciados).

- Se uma propriedade tiver um getter e um setter, ambos deverão ser virtuais, estáticos ou instâncias. Os compiladores do C# e do Visual Basic aplicam automaticamente essa regra por meio de sua sintaxe de definição da propriedade.

Eventos

Um evento é definido por seu nome e tipo. O tipo de evento é um delegado que é usado para indicar o evento. Por exemplo, o evento AppDomain.AssemblyResolve é do tipo ResolveEventHandler. Além do evento em si, três métodos com nomes com base no nome do evento fornecem a implementação do evento e estão marcados como `SpecialName` nos metadados do assembly:

- Um método para adicionar um manipulador de eventos, chamado `add_` *EventName*. Por exemplo, o método de assinatura do evento para o evento AppDomain.AssemblyResolve é chamado `add_AssemblyResolve`.

- Um método para remover um manipulador de eventos, chamado `remove_` *EventName*. Por exemplo, o método de remoção para o evento AppDomain.AssemblyResolve é chamado `remove_AssemblyResolve`.

- Um método para indicar que o evento ocorreu, chamado `raise_` *EventName*.

> **NOTE**
> A maioria das regras da Common Language Specification em relação a eventos é implementada por compiladores de linguagem e é transparente para desenvolvedores de componente.

Os métodos para adicionar, remover e acionar o evento devem ter a mesma acessibilidade. Eles também devem ser todos estáticos, instâncias ou virtuais. Os métodos para adicionar e remover um evento têm um parâmetro cujo tipo é o tipo de delegado do evento. Os métodos para adicionar e remover devem estar ambos presentes ou ausentes.

O exemplo a seguir define uma classe compatível com CLS chamada `Temperature` que acionará um evento `TemperatureChanged` se a mudança de temperatura entre as duas leituras for igual ou exceder o valor de limite. A classe `Temperature` define explicitamente um método `raise_TemperatureChanged` para executar seletivamente os manipuladores de eventos.

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

[assembly: CLSCompliant(true)]

public class TemperatureChangedEventArgs : EventArgs
{
   private Decimal originalTemp;
   private Decimal newTemp;
   private DateTimeOffset when;

   public TemperatureChangedEventArgs(Decimal original, Decimal @new, DateTimeOffset time)
   {
      originalTemp = original;
      newTemp = @new;
```

```
      when = time;
   }

   public Decimal OldTemperature
   {
      get { return originalTemp; }
   }

   public Decimal CurrentTemperature
   {
      get { return newTemp; }
   }

   public DateTimeOffset Time
   {
      get { return when; }
   }
}

public delegate void TemperatureChanged(Object sender, TemperatureChangedEventArgs e);

public class Temperature
{
   private struct TemperatureInfo
   {
      public Decimal Temperature;
      public DateTimeOffset Recorded;
   }

   public event TemperatureChanged TemperatureChanged;

   private Decimal previous;
   private Decimal current;
   private Decimal tolerance;
   private List<TemperatureInfo> tis = new List<TemperatureInfo>();

   public Temperature(Decimal temperature, Decimal tolerance)
   {
      current = temperature;
      TemperatureInfo ti = new TemperatureInfo();
      ti.Temperature = temperature;
      tis.Add(ti);
      ti.Recorded = DateTimeOffset.UtcNow;
      this.tolerance = tolerance;
   }

   public Decimal CurrentTemperature
   {
      get { return current; }
      set {
         TemperatureInfo ti = new TemperatureInfo();
         ti.Temperature = value;
         ti.Recorded = DateTimeOffset.UtcNow;
         previous = current;
         current = value;
         if (Math.Abs(current - previous) >= tolerance)
            raise_TemperatureChanged(new TemperatureChangedEventArgs(previous, current, ti.Recorded));
      }
   }

   public void raise_TemperatureChanged(TemperatureChangedEventArgs eventArgs)
   {
      if (TemperatureChanged == null)
         return;

      foreach (TemperatureChanged d in TemperatureChanged.GetInvocationList()) {
         if (d.Method.Name.Contains("Duplicate"))
            Console.WriteLine("Duplicate event handler; event handler not executed.");
         else
```

```
            d.Invoke(this, eventArgs);
      }
   }
}

public class Example
{
   public Temperature temp;

   public static void Main()
   {
      Example ex = new Example();
   }

   public Example()
   {
      temp = new Temperature(65, 3);
      temp.TemperatureChanged += this.TemperatureNotification;
      RecordTemperatures();
      Example ex = new Example(temp);
      ex.RecordTemperatures();
   }

   public Example(Temperature t)
   {
      temp = t;
      RecordTemperatures();
   }

   public void RecordTemperatures()
   {
      temp.TemperatureChanged += this.DuplicateTemperatureNotification;
      temp.CurrentTemperature = 66;
      temp.CurrentTemperature = 63;
   }

   internal void TemperatureNotification(Object sender, TemperatureChangedEventArgs e)
   {
      Console.WriteLine("Notification 1: The temperature changed from {0} to {1}", e.OldTemperature,
e.CurrentTemperature);
   }

   public void DuplicateTemperatureNotification(Object sender, TemperatureChangedEventArgs e)
   {
      Console.WriteLine("Notification 2: The temperature changed from {0} to {1}", e.OldTemperature,
e.CurrentTemperature);
   }
}
```

```
Imports System.Collections
Imports System.Collections.Generic

<Assembly: CLSCompliant(True)>

Public Class TemperatureChangedEventArgs   : Inherits EventArgs
   Private originalTemp As Decimal
   Private newTemp As Decimal
   Private [when] As DateTimeOffset

   Public Sub New(original As Decimal, [new] As Decimal, [time] As DateTimeOffset)
      originalTemp = original
      newTemp = [new]
      [when] = [time]
   End Sub

   Public ReadOnly Property OldTemperature As Decimal
      Get
```

```
            Return originalTemp
        End Get
    End Property

    Public ReadOnly Property CurrentTemperature As Decimal
        Get
            Return newTemp
        End Get
    End Property

    Public ReadOnly Property [Time] As DateTimeOffset
        Get
            Return [when]
        End Get
    End Property
End Class

Public Delegate Sub TemperatureChanged(sender As Object, e As TemperatureChangedEventArgs)

Public Class Temperature
    Private Structure TemperatureInfo
        Dim Temperature As Decimal
        Dim Recorded As DateTimeOffset
    End Structure

    Public Event TemperatureChanged As TemperatureChanged

    Private previous As Decimal
    Private current As Decimal
    Private tolerance As Decimal
    Private tis As New List(Of TemperatureInfo)

    Public Sub New(temperature As Decimal, tolerance As Decimal)
        current = temperature
        Dim ti As New TemperatureInfo()
        ti.Temperature = temperature
        ti.Recorded = DateTimeOffset.UtcNow
        tis.Add(ti)
        Me.tolerance = tolerance
    End Sub

    Public Property CurrentTemperature As Decimal
        Get
            Return current
        End Get
        Set
            Dim ti As New TemperatureInfo
            ti.Temperature = value
            ti.Recorded = DateTimeOffset.UtcNow
            previous = current
            current = value
            If Math.Abs(current - previous) >= tolerance Then
                raise_TemperatureChanged(New TemperatureChangedEventArgs(previous, current, ti.Recorded))
            End If
        End Set
    End Property

    Public Sub raise_TemperatureChanged(eventArgs As TemperatureChangedEventArgs)
        If TemperatureChangedEvent Is Nothing Then Exit Sub

        Dim ListenerList() As System.Delegate = TemperatureChangedEvent.GetInvocationList()
        For Each d As TemperatureChanged In TemperatureChangedEvent.GetInvocationList()
            If d.Method.Name.Contains("Duplicate") Then
                Console.WriteLine("Duplicate event handler; event handler not executed.")
            Else
                d.Invoke(Me, eventArgs)
            End If
        Next
    End Sub
```

```
End Class

Public Class Example
   Public WithEvents temp As Temperature

   Public Shared Sub Main()
      Dim ex As New Example()
   End Sub

   Public Sub New()
      temp = New Temperature(65, 3)
      RecordTemperatures()
      Dim ex As New Example(temp)
      ex.RecordTemperatures()
   End Sub

   Public Sub New(t As Temperature)
      temp = t
      RecordTemperatures()
   End Sub

   Public Sub RecordTemperatures()
      temp.CurrentTemperature = 66
      temp.CurrentTemperature = 63

   End Sub

   Friend Shared Sub TemperatureNotification(sender As Object, e As TemperatureChangedEventArgs) _
          Handles temp.TemperatureChanged
      Console.WriteLine("Notification 1: The temperature changed from {0} to {1}", e.OldTemperature,
e.CurrentTemperature)
   End Sub

   Friend Shared Sub DuplicateTemperatureNotification(sender As Object, e As TemperatureChangedEventArgs) _
          Handles temp.TemperatureChanged
      Console.WriteLine("Notification 2: The temperature changed from {0} to {1}", e.OldTemperature,
e.CurrentTemperature)
   End Sub
End Class
```

Sobrecargas

A Common Language Specification impõe os seguintes requisitos em membros sobrecarregados:

- Os membros podem ser sobrecarregados com base no número de parâmetros e no tipo de qualquer parâmetro. A convenção de chamada, o tipo de retorno, os modificadores personalizados aplicados ao método ou seus parâmetros e se os parâmetros são passados por valor ou por referência não são considerados na diferenciação entre sobrecargas. Para obter um exemplo, consulte o código para o requisito de que os nomes devem ser exclusivos em um escopo na seção Convenções de nomenclatura.
- Somente propriedades e métodos podem ser sobrecarregados. Campos e eventos não podem ser sobrecarregados.
- Métodos genéricos podem ser sobrecarregados com base no número de seus parâmetros genéricos.

> **NOTE**
> Os operadores `op_Explicit` e `op_Implicit` são exceções à regra de que o valor retornado não é considerado parte de uma assinatura de método para resolução de sobrecarga. Esses dois operadores podem ser sobrecarregados com base nos parâmetros e valor retornado.

Exceções

Objetos de exceção devem derivar de System.Exception ou de outro tipo derivado de System.Exception. O

exemplo a seguir ilustra o erro do compilador que resulta quando uma classe personalizada chamada `ErrorClass` é usada para tratamento de exceções.

```
using System;

[assembly: CLSCompliant(true)]

public class ErrorClass
{
   string msg;

   public ErrorClass(string errorMessage)
   {
      msg = errorMessage;
   }

   public string Message
   {
      get { return msg; }
   }
}

public static class StringUtilities
{
   public static string[] SplitString(this string value, int index)
   {
      if (index < 0 | index > value.Length) {
         ErrorClass badIndex = new ErrorClass("The index is not within the string.");
         throw badIndex;
      }
      string[] retVal = { value.Substring(0, index - 1),
                          value.Substring(index) };
      return retVal;
   }
}
// Compilation produces a compiler error like the following:
//    Exceptions1.cs(26,16): error CS0155: The type caught or thrown must be derived from
//            System.Exception
```

```
Imports System.Runtime.CompilerServices

<Assembly: CLSCompliant(True)>

Public Class ErrorClass
   Dim msg As String

   Public Sub New(errorMessage As String)
      msg = errorMessage
   End Sub

   Public ReadOnly Property Message As String
      Get
         Return msg
      End Get
   End Property
End Class

Public Module StringUtilities
   <Extension()> Public Function SplitString(value As String, index As Integer) As String()
      If index < 0 Or index > value.Length Then
         Dim BadIndex As New ErrorClass("The index is not within the string.")
         Throw BadIndex
      End If
      Dim retVal() As String = { value.Substring(0, index - 1),
                                 value.Substring(index) }
      Return retVal
   End Function
End Module
' Compilation produces a compiler error like the following:
'    Exceptions1.vb(27) : error BC30665: 'Throw' operand must derive from 'System.Exception'.
'
'             Throw BadIndex
'             ~~~~~~~~~~~~~~
```

Para corrigir este erro, a classe `ErrorClass` deve herdar de System.Exception. Além disso, a propriedade `Message` deve ser substituída. O exemplo a seguir corrige esses erros para definir uma classe `ErrorClass` que seja compatível com CLS.

```
using System;

[assembly: CLSCompliant(true)]

public class ErrorClass : Exception
{
   string msg;

   public ErrorClass(string errorMessage)
   {
      msg = errorMessage;
   }

   public override string Message
   {
      get { return msg; }
   }
}

public static class StringUtilities
{
   public static string[] SplitString(this string value, int index)
   {
      if (index < 0 | index > value.Length) {
         ErrorClass badIndex = new ErrorClass("The index is not within the string.");
         throw badIndex;
      }
      string[] retVal = { value.Substring(0, index - 1),
                          value.Substring(index) };
      return retVal;
   }
}
```

```
Imports System.Runtime.CompilerServices

<Assembly: CLSCompliant(True)>

Public Class ErrorClass : Inherits Exception
   Dim msg As String

   Public Sub New(errorMessage As String)
      msg = errorMessage
   End Sub

   Public Overrides ReadOnly Property Message As String
      Get
         Return msg
      End Get
   End Property
End Class

Public Module StringUtilities
   <Extension()> Public Function SplitString(value As String, index As Integer) As String()
      If index < 0 Or index > value.Length Then
         Dim BadIndex As New ErrorClass("The index is not within the string.")
         Throw BadIndex
      End If
      Dim retVal() As String = { value.Substring(0, index - 1),
                                 value.Substring(index) }
      Return retVal
   End Function
End Module
```

Atributos

Em assemblies do .NET Framework, atributos personalizados fornecem um mecanismo extensível para armazenamento de atributos personalizados e recuperação de metadados sobre objetos de programação, como assemblies, tipos, membros e parâmetros de método. Atributos personalizados devem derivar de System.Attribute ou de um tipo derivado de System.Attribute.

O exemplo a seguir viola essa regra. Ele define uma classe `NumericAttribute` que não deriva de System.Attribute. Observe que um erro de compilador só acontece quando o atributo não compatível com CLS é aplicado e não quando a classe é definida.

```
using System;

[assembly: CLSCompliant(true)]

[AttributeUsageAttribute(AttributeTargets.Class | AttributeTargets.Struct)]
public class NumericAttribute
{
   private bool _isNumeric;

   public NumericAttribute(bool isNumeric)
   {
      _isNumeric = isNumeric;
   }

   public bool IsNumeric
   {
      get { return _isNumeric; }
   }
}

[Numeric(true)] public struct UDouble
{
   double Value;
}
// Compilation produces a compiler error like the following:
//    Attribute1.cs(22,2): error CS0616: 'NumericAttribute' is not an attribute class
//    Attribute1.cs(7,14): (Location of symbol related to previous error)
```

```
<Assembly: CLSCompliant(True)>

<AttributeUsageAttribute(AttributeTargets.Class Or AttributeTargets.Struct)> _
Public Class NumericAttribute
   Private _isNumeric As Boolean

   Public Sub New(isNumeric As Boolean)
      _isNumeric = isNumeric
   End Sub

   Public ReadOnly Property IsNumeric As Boolean
      Get
         Return _isNumeric
      End Get
   End Property
End Class

<Numeric(True)> Public Structure UDouble
   Dim Value As Double
End Structure
' Compilation produces a compiler error like the following:
'    error BC31504: 'NumericAttribute' cannot be used as an attribute because it
'    does not inherit from 'System.Attribute'.
'
'    <Numeric(True)> Public Structure UDouble
'     ~~~~~~~~~~~~~
```

O construtor ou as propriedades de um atributo compatível com CLS podem expor somente os seguintes tipos:

- Boolean
- Byte
- Char
- Double
- Int16
- Int32
- Int64
- Single
- String
- Type
- Qualquer tipo de enumeração cujo tipo subjacente seja Byte, Int16, Int32 ou Int64.

O exemplo a seguir define uma classe `DescriptionAttribute` que é derivada de Attribute. O construtor de classe tem um parâmetro do tipo `Descriptor`, portanto a classe não é compatível com CLS. Observe que o compilador do C# emite um aviso, mas compila com êxito, enquanto o compilador do Visual Basic nem emite um aviso ou um erro.

```
using System;

[assembly:CLSCompliantAttribute(true)]

public enum DescriptorType { type, member };

public class Descriptor
{
   public DescriptorType Type;
   public String Description;
}

[AttributeUsage(AttributeTargets.All)]
public class DescriptionAttribute : Attribute
{
   private Descriptor desc;

   public DescriptionAttribute(Descriptor d)
   {
      desc = d;
   }

   public Descriptor Descriptor
   { get { return desc; } }
}
// Attempting to compile the example displays output like the following:
//       warning CS3015: 'DescriptionAttribute' has no accessible
//               constructors which use only CLS-compliant types
```

```
<Assembly:CLSCompliantAttribute(True)>

Public Enum DescriptorType As Integer
   Type = 0
   Member = 1
End Enum

Public Class Descriptor
   Public Type As DescriptorType
   Public Description As String
End Class

<AttributeUsage(AttributeTargets.All)> _
Public Class DescriptionAttribute : Inherits Attribute
   Private desc As Descriptor

   Public Sub New(d As Descriptor)
      desc = d
   End Sub

   Public ReadOnly Property Descriptor As Descriptor
      Get
         Return desc
      End Get
   End Property
End Class
```

O atributo CLSCompliantAttribute

O atributo CLSCompliantAttribute é usado para indicar se um elemento do programa está em conformidade com a Common Language Specification. O construtor CLSCompliantAttribute.CLSCompliantAttribute(Boolean) inclui um único parâmetro necessário, `isCompliant`, que indica se o elemento de programa é compatível com CLS.

No tempo de compilação, o compilador detecta os elementos não compatíveis que provavelmente são compatíveis com CLS e emite um aviso. O compilador não emite avisos para tipos ou membros explicitamente declarados como não compatíveis.

Os desenvolvedores de componentes podem usar o atributo CLSCompliantAttribute de duas maneiras:

- Para definir as partes da interface pública exposta por um componente que são compatíveis com CLS e as partes que não são compatíveis com CLS. Quando o atributo é usado para marcar elementos de programa específicos como compatíveis com CLS, seu uso garante que os elementos sejam acessíveis em todas as linguagens e ferramentas direcionadas ao .NET Framework.
- Para garantir que a interface pública da biblioteca de componentes exponha apenas elementos de programa que são compatíveis com CLS. Se os elementos não forem compatíveis com CLS, os compiladores geralmente emitirão um aviso.

> **WARNING**
> Em alguns casos, os compiladores de linguagem aplicam as regras de compatibilidade com CLS independentemente do uso ou não do atributo CLSCompliantAttribute. Por exemplo, definir um membro estático em uma interface viola uma regra CLS. Neste sentido, se você definir um membro `static` (no C#) ou `Shared` (no Visual Basic) em uma interface, os compiladores do C# e do Visual Basic exibirão uma mensagem de erro e não compilarão o aplicativo.

O atributo CLSCompliantAttribute é marcado com um atributo AttributeUsageAttribute que tem um valor de AttributeTargets.All. Esse valor permite que você aplique o atributo CLSCompliantAttribute a qualquer elemento de programa, incluindo assemblies, módulos, tipos (classes, estruturas, interfaces, enumerações e delegados),

membros de tipo (construtores, métodos, propriedades, campos e eventos), parâmetros, parâmetros genéricos e valores de retorno. No entanto, na prática, você deve aplicar o atributo somente a assemblies, tipos e membros de tipo. Caso contrário, os compiladores ignoram o atributo e continuam gerando avisos do compilador sempre que encontrarem um parâmetro não compatível, parâmetro genérico ou valor retornado na interface pública da biblioteca.

O valor do atributo CLSCompliantAttribute é herdado pelos elementos contidos no programa. Por exemplo, se um assembly for marcado como compatível com CLS, seus tipos também serão compatíveis com CLS. Se um tipo for marcado como compatível com CLS, seus membros e tipos aninhados também serão compatíveis com CLS.

Você pode substituir explicitamente a compatibilidade herdada aplicando o atributo CLSCompliantAttribute a um elemento contido no programa. Por exemplo, é possível usar o atributo CLSCompliantAttribute com um valor `isCompliant` de `false` para definir um tipo não compatível em um assembly compatível, e é possível usar o atributo com um valor `isCompliant` de `true` para definir um tipo compatível em um assembly não compatível. Você também pode definir membros não compatíveis em um tipo compatível. No entanto, um tipo não compatível não pode ter membros compatíveis. Portanto, você não pode usar o atributo com um valor `isCompliant` de `true` para substituir a herança de um tipo não compatível.

Ao desenvolver componentes, você sempre deve usar o atributo CLSCompliantAttribute para indicar se o assembly, seus tipos e membros são compatíveis com CLS.

Para criar componentes compatíveis com CLS:

1. Use CLSCompliantAttribute para marcar o assembly como compatível com CLS.
2. Marque qualquer tipo exposto publicamente no assembly que não seja compatível com CLS como não compatível.
3. Marque qualquer membro publicamente exposto em tipos compatíveis com CLS como não compatíveis.
4. Forneça uma alternativa compatível com CLS para membros não compatíveis com CLS.

Se você marcou com êxito todos os tipos e membros não compatíveis, o compilador não deverá emitir avisos de não conformidade. Entretanto, você deve indicar quais membros não são compatíveis com CLS e listar suas alternativas compatíveis com CLS na documentação do produto.

O exemplo a seguir usa o atributo CLSCompliantAttribute para definir um assembly compatível com CLS e um tipo, `CharacterUtilities`, que tem dois membros não compatíveis com CLS. Como ambos os membros são marcados com o atributo `CLSCompliant(false)`, o compilador não produz avisos. A classe também fornece uma alternativa compatível com CLS para ambos os métodos. Normalmente, nós adicionaríamos apenas duas sobrecargas ao método `ToUTF16` para fornecer alternativas compatíveis com CLS. Entretanto, como os métodos não podem ser sobrecarregados com base no valor retornado, os nomes dos métodos compatíveis com CLS são diferentes dos nomes dos métodos não compatíveis.

```
using System;
using System.Text;

[assembly:CLSCompliant(true)]

public class CharacterUtilities
{
   [CLSCompliant(false)] public static ushort ToUTF16(String s)
   {
      s = s.Normalize(NormalizationForm.FormC);
      return Convert.ToUInt16(s[0]);
   }

   [CLSCompliant(false)] public static ushort ToUTF16(Char ch)
   {
      return Convert.ToUInt16(ch);
   }

   // CLS-compliant alternative for ToUTF16(String).
   public static int ToUTF16CodeUnit(String s)
   {
      s = s.Normalize(NormalizationForm.FormC);
      return (int) Convert.ToUInt16(s[0]);
   }

   // CLS-compliant alternative for ToUTF16(Char).
   public static int ToUTF16CodeUnit(Char ch)
   {
      return Convert.ToInt32(ch);
   }

   public bool HasMultipleRepresentations(String s)
   {
      String s1 = s.Normalize(NormalizationForm.FormC);
      return s.Equals(s1);
   }

   public int GetUnicodeCodePoint(Char ch)
   {
      if (Char.IsSurrogate(ch))
         throw new ArgumentException("ch cannot be a high or low surrogate.");

      return Char.ConvertToUtf32(ch.ToString(), 0);
   }

   public int GetUnicodeCodePoint(Char[] chars)
   {
      if (chars.Length > 2)
         throw new ArgumentException("The array has too many characters.");

      if (chars.Length == 2) {
         if (! Char.IsSurrogatePair(chars[0], chars[1]))
            throw new ArgumentException("The array must contain a low and a high surrogate.");
         else
            return Char.ConvertToUtf32(chars[0], chars[1]);
      }
      else {
         return Char.ConvertToUtf32(chars.ToString(), 0);
      }
   }
}
```

```
Imports System.Text

<Assembly:CLSCompliant(True)>

Public Class CharacterUtilities
   <CLSCompliant(False)> Public Shared Function ToUTF16(s As String) As UShort
      s = s.Normalize(NormalizationForm.FormC)
      Return Convert.ToUInt16(s(0))
   End Function

   <CLSCompliant(False)> Public Shared Function ToUTF16(ch As Char) As UShort
      Return Convert.ToUInt16(ch)
   End Function

   ' CLS-compliant alternative for ToUTF16(String).
   Public Shared Function ToUTF16CodeUnit(s As String) As Integer
      s = s.Normalize(NormalizationForm.FormC)
      Return CInt(Convert.ToInt16(s(0)))
   End Function

   ' CLS-compliant alternative for ToUTF16(Char).
   Public Shared Function ToUTF16CodeUnit(ch As Char) As Integer
      Return Convert.ToInt32(ch)
   End Function

   Public Function HasMultipleRepresentations(s As String) As Boolean
      Dim s1 As String = s.Normalize(NormalizationForm.FormC)
      Return s.Equals(s1)
   End Function

   Public Function GetUnicodeCodePoint(ch As Char) As Integer
      If Char.IsSurrogate(ch) Then
         Throw New ArgumentException("ch cannot be a high or low surrogate.")
      End If
      Return Char.ConvertToUtf32(ch.ToString(), 0)
   End Function

   Public Function GetUnicodeCodePoint(chars() As Char) As Integer
      If chars.Length > 2 Then
         Throw New ArgumentException("The array has too many characters.")
      End If
      If chars.Length = 2 Then
         If Not Char.IsSurrogatePair(chars(0), chars(1)) Then
            Throw New ArgumentException("The array must contain a low and a high surrogate.")
         Else
            Return Char.ConvertToUtf32(chars(0), chars(1))
         End If
      Else
         Return Char.ConvertToUtf32(chars.ToString(), 0)
      End If
   End Function
End Class
```

Se você estiver desenvolvendo um aplicativo em vez de uma biblioteca (ou seja, se não estiver expondo tipos ou membros que possam ser consumidos por outros desenvolvedores de aplicativos), a conformidade com CLS dos elementos do programa que seu aplicativo consome só serão de interesse se sua linguagem não der suporte a eles. Nesse caso, seu compilador de linguagem gerará um erro quando você tentar usar um elemento não compatível com CLS.

Interoperabilidade em qualquer idioma

A independência de linguagem tem vários significados possíveis. Um significado, discutido no artigo Independência de linguagem e componentes independentes de linguagem, envolve o consumo pleno de tipos escritos em uma linguagem de um aplicativo escrito em outra linguagem. Um segundo significado, que é o

enfoque deste artigo, envolve combinar o código gravado em várias linguagens em um único assembly do .NET Framework.

O exemplo a seguir ilustra a interoperabilidade em qualquer idioma com a criação de uma biblioteca de classes chamada Utilities.dll que inclui duas classes, `NumericLib` e `StringLib`. A classe `NumericLib` é gravada em C#, e a classe `StringLib` é gravada em Visual Basic. Aqui está o código-fonte de StringUtil.vb, que inclui um único membro, `ToTitleCase`, em sua classe `StringLib`.

```
Imports System.Collections.Generic
Imports System.Runtime.CompilerServices

Public Module StringLib
   Private exclusions As List(Of String)

   Sub New()
      Dim words() As String = { "a", "an", "and", "of", "the" }
      exclusions = New List(Of String)
      exclusions.AddRange(words)
   End Sub

   <Extension()> _
   Public Function ToTitleCase(title As String) As String
      Dim words() As String = title.Split()
      Dim result As String = String.Empty

      For ctr As Integer = 0 To words.Length - 1
         Dim word As String = words(ctr)
         If ctr = 0 OrElse Not exclusions.Contains(word.ToLower()) Then
            result += word.Substring(0, 1).ToUpper() + _
                      word.Substring(1).ToLower()
         Else
            result += word.ToLower()
         End If
         If ctr <= words.Length - 1 Then
            result += " "
         End If
      Next
      Return result
   End Function
End Module
```

Aqui está o código-fonte de NumberUtil.cs, que define uma classe `NumericLib` com dois membros, `IsEven` e `NearZero`.

```
using System;

public static class NumericLib
{
   public static bool IsEven(this IConvertible number)
   {
      if (number is Byte ||
          number is SByte ||
          number is Int16 ||
          number is UInt16 ||
          number is Int32 ||
          number is UInt32 ||
          number is Int64)
         return Convert.ToInt64(number) % 2 == 0;
      else if (number is UInt64)
         return ((ulong) number) % 2 == 0;
      else
         throw new NotSupportedException("IsEven called for a non-integer value.");
   }

   public static bool NearZero(double number)
   {
      return Math.Abs(number) < .00001;
   }
}
```

Para empacotar as duas classes em um único assembly, você deve compilá-las em módulos. Para compilar o arquivo de código-fonte do Visual Basic em um módulo, use este comando:

```
vbc /t:module StringUtil.vb
```

Para obter mais informações sobre a sintaxe de linha de comando do compilador do Visual Basic, consulte Compilando através da linha de comando.

Para compilar o arquivo de código-fonte do C# em um módulo, use este comando:

```
csc /t:module NumberUtil.cs
```

Para obter mais informações sobre a sintaxe de linha de comando do compilador do C#, consulte Compilando na linha de comando com csc.exe.

Então você usa as Opções do vinculador para compilar os dois módulos em um assembly:

```
link numberutil.netmodule stringutil.netmodule /out:UtilityLib.dll /dll
```

O exemplo a seguir chama os métodos `NumericLib.NearZero` e `StringLib.ToTitleCase`. Observe que o código do Visual Basic e o código do C# podem acessar os métodos em ambas as classes.

```
using System;

public class Example
{
   public static void Main()
   {
      Double dbl = 0.0 - Double.Epsilon;
      Console.WriteLine(NumericLib.NearZero(dbl));

      string s = "war and peace";
      Console.WriteLine(s.ToTitleCase());
   }
}
// The example displays the following output:
//       True
//       War and Peace
```

```
Module Example
   Public Sub Main()
      Dim dbl As Double = 0.0 - Double.Epsilon
      Console.WriteLine(NumericLib.NearZero(dbl))

      Dim s As String = "war and peace"
      Console.WriteLine(s.ToTitleCase())
   End Sub
End Module
' The example displays the following output:
'       True
'       War and Peace
```

Para compilar o código do Visual Basic, use este comando:

```
vbc example.vb /r:UtilityLib.dll
```

Para compilar usando C#, altere o nome do compilador de **vbc** para **csc** e altere a extensão do arquivo de .vb para .cs:

```
csc example.cs /r:UtilityLib.dll
```

Consulte também

- CLSCompliantAttribute

Bibliotecas do Framework

29/11/2019 • 6 minutes to read • Edit Online

O .NET tem um conjunto padrão expansivo de bibliotecas de classes, conhecido como as bibliotecas de classes base (conjunto principal) ou as bibliotecas de classes .NET Framework (conjunto completo). Essas bibliotecas fornecem implementações para muitos algoritmos, funcionalidades do utilitário e tipos gerais e específicos do aplicativo. Tanto bibliotecas comerciais quanto comunitárias são criadas com base nas bibliotecas de classes .NET Framework, fornecendo bibliotecas off-the-shelf fáceis de usar para uma amplo conjunto de tarefas de computação.

Um subconjunto dessas bibliotecas é fornecido com cada implementação do .NET. APIs de BCL (bibliotecas de classes base) são esperadas com qualquer implementação do .NET, porque os desenvolvedores desejarão tê-las e porque bibliotecas populares precisarão delas para serem executadas. Bibliotecas específicas de aplicativo acima da BCL, como ASP.NET, não estarão disponíveis em todas as implementações do .NET.

Bibliotecas de classes base

As BCL fornecem a funcionalidade de utilitário e tipos mais básicos e são a base de todas as outras bibliotecas de classes do .NET. Elas têm o objetivo de fornecer implementações de caráter bastante geral, sem nenhum desvio para nenhuma carga de trabalho. O desempenho é sempre uma consideração importante, já que os aplicativos podem preferir uma política específica como baixa latência para alta taxa de transferência ou memória insuficiente para baixo uso de CPU. Essas bibliotecas estão destinadas a, em termos gerais, serem de alto desempenho e adotarem uma abordagem intermediária acordo com essas preocupações de desempenho diversas. Para a maioria dos aplicativos, essa abordagem tem sido muito bem-sucedida.

Tipos primitivos

O .NET inclui um conjunto de tipos primitivos, que são usados (em graus variáveis) em todos os programas. Esses tipos contêm dados, como números, cadeias de caracteres, bytes e objetos arbitrários. A linguagem C# inclui palavras-chave para esses tipos. Um conjunto de amostra desses tipos é listado abaixo, com palavras-chave do C# correspondentes.

- System.Object (object) – a classe base ultimate no sistema de tipos CLR. É a raiz da hierarquia de tipos.
- System.Int16 (short) – tipo inteiro com sinal de 16 bits. O UInt16 sem sinal também existe.
- System.Int32 (int) – um tipo inteiro com sinal de 32 bits. O UInt32 sem sinal também existe.
- System.Single (float) – um tipo de ponto flutuante de 32 bits.
- System.Decimal (decimal) – um tipo decimal de 128 bits.
- System.Byte (byte) – um inteiro de 8 bits sem sinal que representa um byte de memória.
- System.Boolean (bool)-um tipo booliano que representa `true` ou `false`.
- System.Char (char) – um tipo numérico de 16 bits que representa um caractere Unicode.
- System.String (string) – representa uma série de caracteres. Diferente de um `char[]`, mas permite a indexação em cada `char` individual em `string`.

Estruturas de dados

O .NET inclui um conjunto de estruturas de dados que são fundamentais para quase todos os aplicativos .NET. Elas são em sua maioria coleções, mas também incluem outros tipos.

- Array – representa uma matriz de objetos fortemente tipados que podem ser acessados por índice. Tem um

tamanho fixo, de acordo com sua construção.

- List<T> – representa uma lista fortemente tipada de objetos que podem ser acessados por índice. É redimensionado automaticamente conforme necessário.
- Dictionary<TKey,TValue> – representa uma coleção de valores que são indexados por uma chave. Os valores podem ser acessados via chave. É redimensionado automaticamente conforme necessário.
- Uri – fornece uma representação de objeto de um URI (Uniform Resource Identifier) e fácil acesso às partes do URI.
- DateTime – representa um momento no tempo, geralmente expresso como uma data e hora do dia.

APIs utilitárias

O .NET inclui um conjunto de APIs utilitárias que fornecem funcionalidade para várias tarefas importantes.

- HttpClient – uma API para enviar solicitações HTTP e receber respostas HTTP de um recurso identificado por um URI.
- XDocument – uma API para carregar e consultar documentos XML com o LINQ.
- StreamReader – uma API para ler arquivos.
- StreamWriter – uma API para gravar arquivos.

APIs do modelo de aplicativo

Há muitos modelos de aplicativo que podem ser usados com o .NET, fornecidos por várias empresas.

- ASP.NET – fornece uma estrutura da Web para a criação de sites e serviços. Suporte para Windows, Linux e macOS (depende de versão do ASP.NET).

Visão geral da biblioteca de classes do .NET

31/10/2019 • 9 minutes to read • Edit Online

As implementações do .NET incluem classes, interfaces, delegados e tipos de valor que agilizam e otimizam o processo de desenvolvimento e dão acesso à funcionalidade do sistema. Para facilitar a interoperabilidade entre linguagens, os tipos do .NET têm conformidade com CLS e, assim, podem ser usados por qualquer linguagem de programação cujo compilador esteja de acordo com a CLS (Common Language Specification).

Os tipos do .NET são a base na qual os aplicativos, os componentes e os controles do .NET são criados. As implementações do .NET incluem tipos que realizam as seguintes funções:

- Representam tipos de dados base e exceções.
- Encapsulam estruturas de dados.
- Executam E/S.
- Acessam informações sobre tipos carregados.
- Invocam verificações de segurança do .NET Framework.
- Fornecem acesso a dados, uma GUI detalhada do lado do cliente e uma GUI do lado do cliente controlada pelo servidor.

O .NET fornece um conjunto avançado de interfaces, bem como classes abstratas e concretas (não abstratas). Você pode usar as classes concretas como estão ou, em muitos casos, pode derivar suas próprias classes a partir delas. Para usar a funcionalidade de uma interface, você pode criar uma classe que implementa a interface ou derivar uma classe de uma das classes do .NET que implementa a interface.

Convenções de nomenclatura

Os tipos do .NET usam um esquema de nomenclatura de sintaxe por ponto que denota uma hierarquia. Essa técnica agrupa tipos relacionados em namespaces para que eles possam ser pesquisados e referenciados com mais facilidade. A primeira parte do nome completo — até o ponto mais à direita — é o nome do namespace. A última parte do nome é o nome do tipo. Por exemplo, `System.Collections.Generic.List<T>` representa o tipo `List<T>`, que pertence ao namespace `System.Collections.Generic`. Os tipos em System.Collections.Generic podem ser usados para trabalhar com coleções genéricas.

Esse esquema de nomenclatura facilita para desenvolvedores de bibliotecas estender o .NET Framework para criar grupos hierárquicos de tipos e nomeá-los de maneira consistente e informativa. Ele também permite que tipos sejam identificados sem ambiguidades por seu nome completo (ou seja, pelo namespace e pelo nome de tipo), o que impede conflitos de nomes de tipo. Os desenvolvedores de bibliotecas devem usar a seguinte convenção para criar nomes para seus namespaces:

CompanyName.TechnologyName

Por exemplo, o namespace `Microsoft.Word` segue esta diretriz.

O uso de padrões de nomenclatura para agrupar tipos relacionados em namespaces é uma forma muito útil de compilar e documentar bibliotecas de classes. No entanto, esse esquema de nomenclatura não afeta visibilidade, acesso a membro, herança, segurança ou associação. Um namespace pode ser particionado em vários assemblies e um único assembly pode conter tipos de vários namespaces. O assembly fornece a estrutura formal para criação de versão, implantação, segurança, carregamento e visibilidade no Common Language Runtime.

Para obter mais informações sobre namespaces e nomes de tipos, consulte Common Type System.

namespace System

O System é o namespace raiz para tipos fundamentais no.NET. Esse namespace contém classes que representam os tipos de dados base usados por todos os aplicativos: Object (a raiz da hierarquia de herança), Byte, Char, Array, Int32, String etc. Muitos desses tipos correspondem aos tipos de dados primitivos que sua linguagem de programação usa. Ao gravar códigos usando tipos .NET Framework, você pode usar a palavra-chave correspondente da sua linguagem quando um tipo de dados base do .NET Framework é esperado.

A tabela a seguir lista os tipos base que o .NET fornece, descreve resumidamente cada tipo e indica o tipo correspondente em Visual Basic, C#, C++ e F#.

CATEGORIA	NOME DA CLASSE	DESCRIÇÃO	TIPO DE DADOS EM VISUAL BASIC	TIPO DE DADOS EM C#	TIPO DE DADOS DE C++/CLI	TIPO DE DADOS DE F#
Inteiro	Byte	Um inteiro de 8 bits sem sinal.	**Byte**	**byte**	**unsigned char**	**byte**
	SByte	Um inteiro de 8 bits com sinal. Não compatível com CLS.	**SByte**	**sbyte**	**char** - ou - **signed char**	**sbyte**
	Int16	Um inteiro de 16 bits com sinal.	**Short**	**short**	**short**	**int16**
	Int32	Um inteiro com sinal de 32 bits.	**Integer**	**int**	**int** - ou - **long**	**int**
	Int64	Um inteiro com sinal de 64 bits.	**Long**	**long**	**__int64**	**int64**
	UInt16	Um inteiro de 16 bits sem sinal. Não compatível com CLS.	**UShort**	**ushort**	**unsigned short**	**uint16**
	UInt32	Um inteiro sem sinal de 32 bits. Não compatível com CLS.	**UInteger**	**uint**	**unsigned int** - ou - **unsigned long**	**uint32**

CATEGORIA	NOME DA CLASSE	DESCRIÇÃO	TIPO DE DADOS EM VISUAL BASIC	TIPO DE DADOS EM C#	TIPO DE DADOS DE C++/CLI	TIPO DE DADOS DE F#
	UInt64	Um inteiro sem sinal de 64 bits. Não compatível com CLS.	**ULong**	**ulong**	**unsigned __int64**	**uint64**
Ponto flutuante	Single	Um número de ponto flutuante de precisão simples (32 bits).	**Simples**	**float**	**float**	**float32** ou **single**
	Double	Um número de ponto flutuante de precisão dupla (64 bits).	**Duplo**	**double**	**double**	**float** ou **double**
Lógico	Boolean	Um valor booliano (verdadeiro ou falso).	**Booliano**	**bool**	**bool**	**bool**
Outros	Char	Um caractere Unicode (16 bits).	**Char**	**char**	**wchar_t**	**char**
	Decimal	Um valor decimal (128 bits).	**Decimal**	**decimal**	**Decimal**	**decimal**
	IntPtr	Um inteiro com sinal cujo tamanho dependa da plataforma subjacente (um valor de 32 bits em uma plataforma de 32 bits e um valor de 64 bits em uma plataforma de 64 bits).	**IntPtr** Nenhum tipo interno.	**IntPtr** Nenhum tipo interno.	**IntPtr** Nenhum tipo interno.	**unativeint**

CATEGORIA	NOME DA CLASSE	DESCRIÇÃO	TIPO DE DADOS EM VISUAL BASIC	TIPO DE DADOS EM C#	TIPO DE DADOS DE C++/CLI	TIPO DE DADOS DE F#
	UIntPtr	Um inteiro sem sinal cujo tamanho dependa da plataforma subjacente (um valor de 32 bits em uma plataforma de 32 bits e um valor de 64 bits em uma plataforma de 64 bits). Não compatível com CLS.	**UIntPtr** Nenhum tipo interno.	**UIntPtr** Nenhum tipo interno.	**UIntPtr** Nenhum tipo interno.	**unativeint**
	Object	A raiz da hierarquia do objeto.	**Object**	**object**	**Object^**	**obj**
	String	Uma cadeia de caracteres Unicode imutável e de comprimento fixo.	**Cadeia de caracteres**	**string**	**String^**	**string**

Além dos tipos de dados base, o namespace System contém mais de 100 classes, que vão desde classes que identificam exceções até classes que lidam com os conceitos de tempo de execução, como domínios de aplicativo e o coletor de lixo. O namespace System também contém vários namespaces de segundo nível.

Para obter mais informações sobre namespaces, use o Navegador de API do .NET para procurar a Biblioteca de classes do .NET. A documentação de referência de API fornece informações sobre cada namespace, seus tipos e cada um dos seus membros.

Consulte também

- Common Type System
- Navegador de API do .NET
- Visão Geral

Trabalhando com tipos base no .NET

31/10/2019 • 2 minutes to read • Edit Online

Esta seção descreve as operações de tipo base .NET, incluindo formatação, conversão e operações comuns.

Nesta seção

Conversão de tipos no .NET Framework
Descreve como converter de um tipo em outro.

Formatando Tipos
Descreve como formatar cadeias de caracteres usando especificadores de formato de cadeia de caracteres.

Manipulando cadeias de caracteres
Descreve como manipular e formatar cadeias de caracteres.

Análise de cadeias de caracteres
Descreve como converter cadeias de caracteres em tipos do .NET Framework.

Seções relacionadas

Common Type System
Descreve os tipos usados pelo .NET Framework.

Datas, horas e fusos horários
Descreve como trabalhar com fusos horários e conversões de fusos horários em aplicativos com distinção de fusos horários.

Bibliotecas de classes do .NET

31/10/2019 • 7 minutes to read • Edit Online

As bibliotecas de classe são o conceito de biblioteca compartilhada para .NET. Elas permitem que você componentize funcionalidades úteis em módulos que podem ser usados por vários aplicativos. Elas também podem ser usadas como um meio de carregar a funcionalidade não necessária ou não conhecida na inicialização do aplicativo. Bibliotecas de classe são descritas usando o formato de arquivo do Assembly .NET.

Há três tipos de bibliotecas de classes que você pode usar:

- Bibliotecas de classe **específicas da plataforma** têm acesso a todas as APIs em uma determinada plataforma (por exemplo, .NET Framework, Xamarin iOS), mas só podem ser usadas por aplicativos e bibliotecas que direcionam essa plataforma.
- Bibliotecas de classe **portáteis** têm acesso a um subconjunto de APIs e podem ser usadas por aplicativos e bibliotecas voltados para várias plataformas.
- Bibliotecas de classe **.NET Standard** são uma fusão do conceito de biblioteca portátil e específica da plataforma em um único modelo que fornece o melhor dos dois mundos.

Bibliotecas de classes específicas da plataforma

Bibliotecas específicas da plataforma estão associadas a uma única implementação do .NET (por exemplo, .NET Framework no Windows) e, portanto, podem assumir dependências significativas em um ambiente de execução conhecido. Esse ambiente exporá um conjunto conhecido de APIs (.NET e APIs de SO) e manterá e exporá o estado esperado (por exemplo, o Registro do Windows).

Os desenvolvedores que criam bibliotecas específicas de plataforma podem aproveitar ao máximo a plataforma subjacente. As bibliotecas só serão executadas na plataforma específica, fazendo verificações de plataforma ou outras formas de código condicional desnecessários (módulo de códigos de fornecimento únicos para várias plataformas).

Bibliotecas específicas da plataforma são o tipo de biblioteca de classes principal para o .NET Framework. Mesmo que outras implementações do .NET tenham surgido, as bibliotecas específicas da plataforma permaneceram o tipo de biblioteca dominante.

Bibliotecas de classes portáteis

Há suporte para bibliotecas portáteis em várias implementações do .NET. Elas ainda podem assumir dependências em um ambiente de execução conhecido, no entanto, o ambiente é sintético e é gerado pela intersecção de um conjunto de implementações de .NET concretas. Isso significa que suposições de plataforma e APIs expostas são um subconjunto que estaria disponível em uma biblioteca específica da plataforma.

Você escolhe uma configuração de plataforma ao criar uma biblioteca portátil. Esses são os conjuntos de plataformas que você precisa dar suporte (por exemplo, .NET Framework 4.5+, Windows Phone 8.0+). Quanto mais plataformas você escolher dar suporte, menos suposições de plataforma terá que fazer e menor será o denominador comum. Essa característica pode ser confusa a princípio, já que as pessoas geralmente pensam que "mais é melhor", mas descobrem que mais plataformas com suporte resultam em menos APIs disponíveis.

Muitos desenvolvedores de biblioteca mudam de produzir várias bibliotecas específicas de plataforma de uma origem (usando as diretivas de compilação condicional) para bibliotecas portáteis. Há várias abordagens para acessar a funcionalidade específica da plataforma nas bibliotecas portáteis com a técnica bait-and-switch, a mais amplamente aceita neste momento.

Bibliotecas de classe do .NET Standard

As bibliotecas do .NET Standard são uma substituição dos conceitos de bibliotecas específicas da plataforma e portáteis. Elas são específicas da plataforma no sentido de que expõem toda a funcionalidade da plataforma subjacente (sem plataformas sintéticas ou interseções de plataforma). Elas são portáteis no sentido de que funcionam em todas as plataformas de suporte.

O .NET Standard expõe um conjunto de *contratos* de biblioteca. As implementações do .NET devem dar suporte a cada contrato, completamente ou não dar nenhum suporte. Cada implementação, portanto, dá suporte a um conjunto de contratos do .NET Standard. O resultado é que há suporte para cada biblioteca de classes do .NET Standard nas plataformas compatíveis com suas dependências de contrato.

O .NET Standard não expõe toda a funcionalidade do .NET Framework (isso nem é uma meta), no entanto, ele expõe mais APIs que as Bibliotecas de classes portáteis. Mais APIs serão adicionadas ao longo do tempo.

As seguintes plataformas dão suporte às bibliotecas de classes do .NET Standard:

- .NET Core
- .NET Framework
- Mono
- Xamarin.iOS, Xamarin.Mac, Xamarin.Android
- UWP (Plataforma Universal do Windows)
- Windows
- Windows Phone
- Windows Phone Silverlight

Para obter mais informações, consulte o tópico .NET Standard.

Bibliotecas de classes do Mono

Há suporte para as bibliotecas de classe no Mono, incluindo os três tipos de bibliotecas descritas acima. Mono era geralmente visto (corretamente) como uma implementação de plataforma cruzada do Microsoft .NET Framework. Em parte, isso acontecia porque bibliotecas do .NET Framework específicas da plataforma podiam ser executadas no tempo de execução do Mono sem modificação ou recompilação. Essa característica estava em vigor antes da criação de bibliotecas de classes portáteis, portanto, era uma opção óbvia para habilitar a portabilidade binária entre o .NET Framework e o Mono (embora ele funcionasse apenas em uma direção).

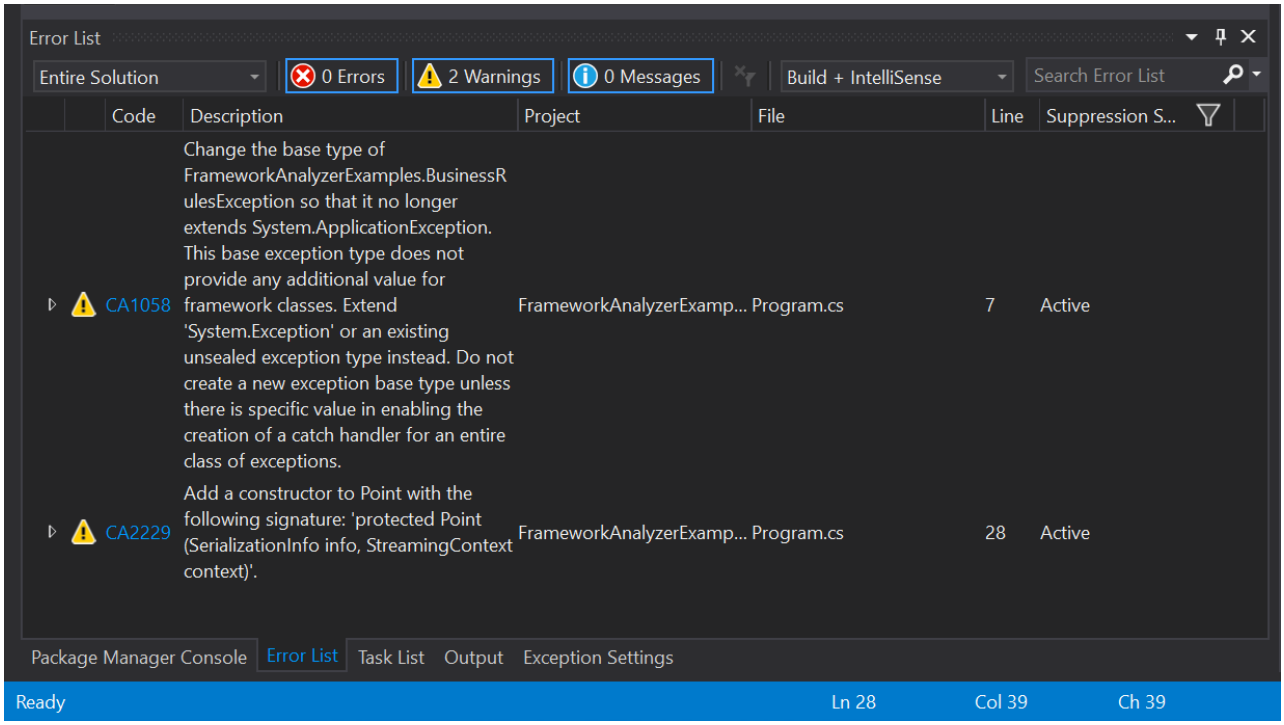
Os analisadores do Roslyn

23/10/2019 • 3 minutes to read • Edit Online

Os analisadores do Roslyn usam o .NET Compiler SDK (APIs do Roslyn) para analisar o código-fonte do seu projeto para localizar problemas e sugerir correções. Analisadores diferentes procuram diferentes classes de problemas, variando de práticas que podem causar erros a problemas de segurança à compatibilidade da API.

Os analisadores do Roslyn trabalham tanto interativamente quanto durante as compilações. O analisador fornece uma orientações diferentes no Visual Studio ou em compilações da linha de comando.

Embora você edite códigos no Visual Studio, os analisadores são executados enquanto você faz alterações, capturando possíveis problemas assim que você cria um código que aciona problemas. Os problemas são destacados com linhas onduladas abaixo deles. O Visual Studio exibe uma lâmpada. Quando você clica nela, o analisador sugere soluções possíveis para o problema. Quando você compila o projeto, no Visual Studio ou na linha de comando, o código-fonte é analisado, e o analisador fornece uma lista completa de possíveis problemas. A imagem a seguir mostra um exemplo.



Os analisadores do Roslyn relatam problemas potenciais como erros, avisos ou informações com base na gravidade do problema.

É possível instalar os analisadores do Roslyn como pacotes NuGet no projeto. Os analisadores configurados e qualquer configuração para cada analisador são restaurados e executados no computador de qualquer desenvolvedor para o projeto.

> **NOTE**
>
> A experiência do usuário com os analisadores do Roslyn é diferente da experiência com as bibliotecas de análise de código, como as versões mais antigas do FxCop e as ferramentas de análise de segurança. Você não precisa executar explicitamente os analisadores do Roslyn. Não é necessário usar os itens de menu "Executar análise de código" no menu "Analisar" do Visual Studio. Os analisadores do Roslyn executam de forma assíncrona enquanto você trabalha.

Mais informações sobre analisadores específicos

Os analisadores a seguir são abordados nesta seção:

- Analisador de API: este analisador verifica se há em seu código possíveis riscos de compatibilidade ou usos de APIs preteridas.
- Analisador de estrutura: este analisador examina seu código para verificar se ele segue as diretrizes dos aplicativos do .NET Framework. Essas regras incluem várias recomendações com base em segurança.
- .NET Portability Analyzer: esse analisador examina seu código para ver a quantidade de trabalho necessário para tornar o aplicativo compatível com outras implementações do .NET e perfis, incluindo .NET Core, .NET Standard, UWP e Xamarin para iOS, Android e Mac.

Analisador de API do .NET

23/10/2019 • 9 minutes to read • Edit Online

O analisador do API .NET é um analisador Roslyn que descobre riscos potenciais de compatibilidade para APIs em C# em diferentes plataformas e detecta chamadas a APIs preteridas. Pode ser útil para todos os desenvolvedores de C# em qualquer estágio do desenvolvimento.

O Analisador de API é fornecido como um pacote NuGet Microsoft.DotNet.Analyzers.Compatibility. Depois que você faz referência a ele em um projeto, ele automaticamente monitora o código e indica um problema de uso de API. Você também pode obter sugestões sobre possíveis correções clicando na lâmpada. O menu suspenso inclui uma opção para suprimir os avisos.

> **NOTE**
> O analisador do .NET API ainda é uma versão de pré-lançamento.

Pré-requisitos

- Visual Studio 2017 e versões posteriores ou Visual Studio para Mac (todas as versões).

Descobrindo APIs preteridas

Quais são as APIs preteridas?

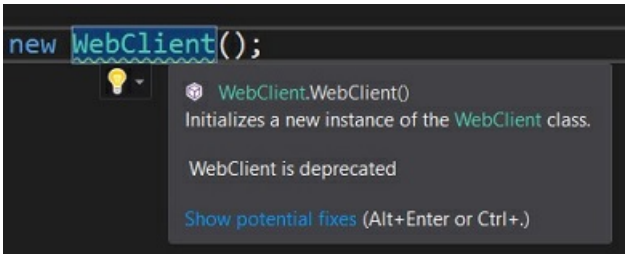
A família .NET é um conjunto de produtos grandes que são atualizados constantemente para atender melhor às necessidades dos clientes. É natural substituir algumas APIs e substituí-las por novas. Uma API é considerada preterida quando existe uma alternativa melhor. Uma maneira de informar que uma API foi preterida e não deve ser usada é marcá-la com o atributo ObsoleteAttribute. A desvantagem dessa abordagem é que há apenas uma ID de diagnóstico para todas as APIs preteridas (para C#, CS0612). Isso significa que:

- É impossível ter documentos dedicados para cada caso.
- É impossível suprimir determinada categoria de avisos. Você pode suprimir todos ou nenhum deles.
- Para informar os usuários de uma nova preterição, um assembly referenciado ou um pacote de direcionamento deve ser atualizado.

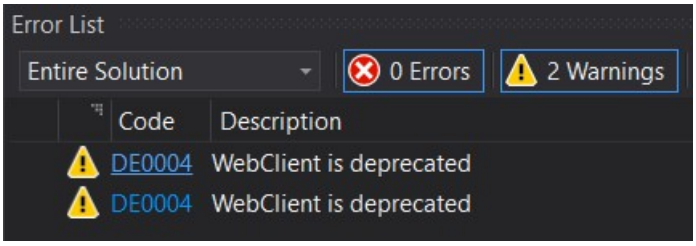
O Analisador de API usa códigos de erro específicos de API que começam com DE (que representa Erro de Preterição), o que permite controlar a exibição de avisos individuais. As APIs preteridas identificadas pelo analisador são definidas no repositório dotnet/platform-compat.

Como usar o analisador de API

Quando uma API preterida, como WebClient, é usada em um código, o Analisador de API a realça com uma linha irregular verde. Quando você focaliza a chamada de API, uma lâmpada é exibida com informações sobre a substituição de API, como no seguinte exemplo:



A janela **Lista de Erros** contém avisos com uma ID exclusiva por API preterida, conforme mostrado no seguinte exemplo (`DE004`):



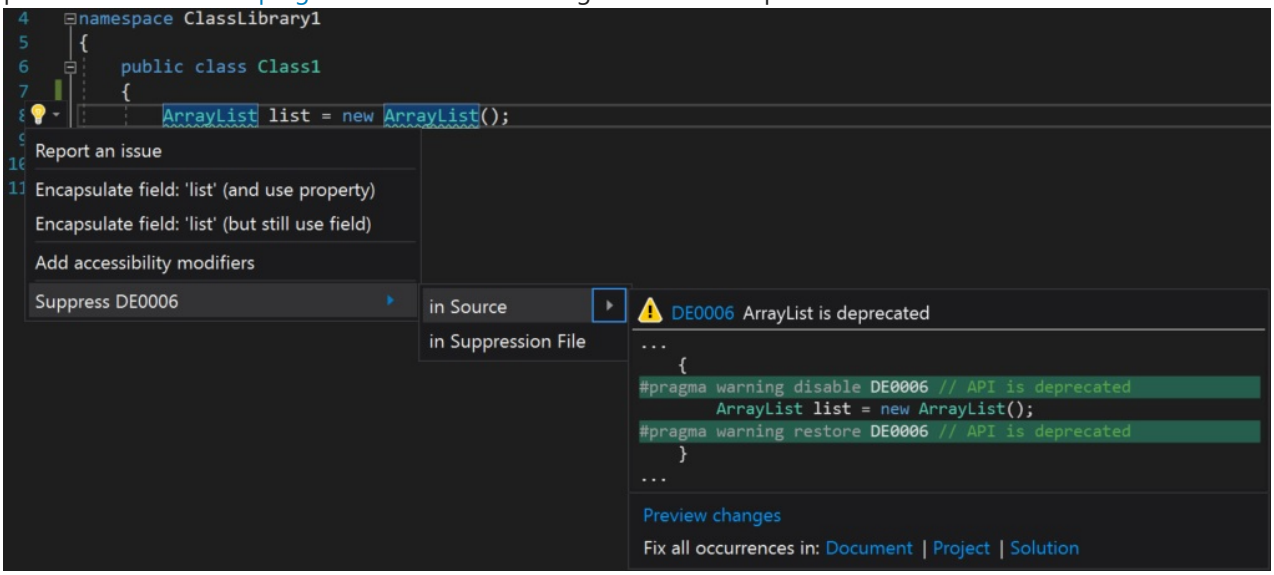
Clicando na ID, você vai para uma página da Web com informações detalhadas sobre por que a API foi preterida e sugestões sobre APIs alternativas que podem ser usadas.

Os avisos podem ser suprimidos clicando com o botão direito do mouse no membro realçado e selecionando **Suprimir <ID de diagnóstico>** . Há duas maneiras de suprimir avisos:

- localmente (no código-fonte)
- globalmente (em um arquivo de supressão) – recomendado

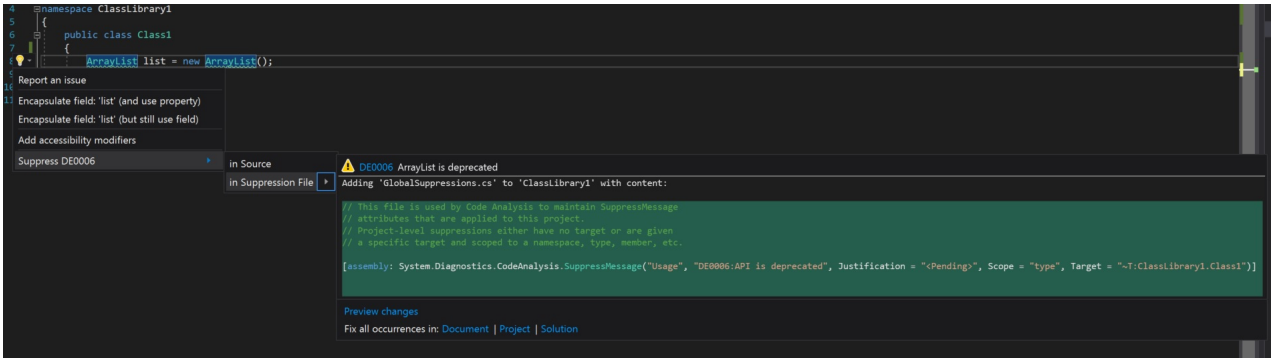
Como suprimir avisos localmente

Para suprimir avisos localmente, clique no membro para o qual você deseja suprimir avisos e selecione **Ações Rápidas e Refatorações** > **Suprimir *ID de diagnóstico* <ID de diagnóstico >** > **na Fonte**. A diretiva de pré-processador de aviso #pragma é adicionada ao código-fonte no escopo definido:

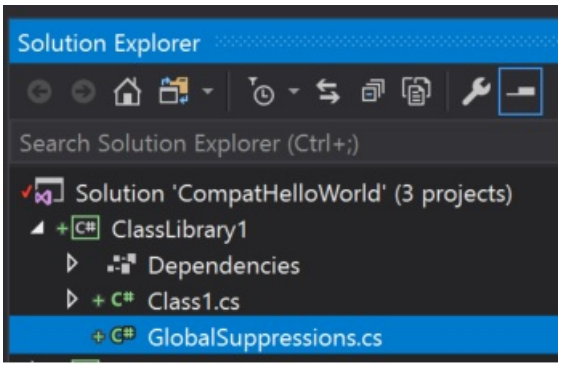


Como suprimir avisos globalmente

Para suprimir avisos globalmente, clique no membro para o qual você deseja suprimir avisos e selecione **Ações Rápidas e Refatorações** > **Suprimir *ID de diagnóstico*<ID de diagnóstico >** > **no Arquivo de Supressão**.



Um arquivo *GlobalSuppressions.cs* é adicionado ao projeto após a primeira supressão. Novo supressões globais são anexadas a este arquivo.



A supressão global é a maneira recomendada de garantir a consistência do uso da API em projetos.

Detectando problemas entre plataformas

De forma semelhante a APIs preteridas, o analisador identifica todas as APIs que não são entre plataformas. Por exemplo, Console.WindowWidth funciona no Windows, mas não no Linux ou no macOS. A ID de diagnóstico é mostrada na janela **Lista de Erros**. Você pode suprimir esse aviso clicando e selecionando **Ações Rápidas e Refatorações**. Diferentemente de casos de preterição em que você tem duas opções (continuar usando o membro preterido e suprimir avisos ou não o utilizar), aqui, se estiver desenvolvendo o código apenas para algumas plataformas, você poderá suprimir todos os avisos para todas as outras plataformas em que não planejar executar o código. Para fazer isso, basta editar o arquivo de projeto e adicionar a propriedade `PlatformCompatIgnore` que lista todas as plataformas a serem ignoradas. Os valores aceitos são: `Linux`, `macOS` e `Windows`.

```
<PropertyGroup>
    <PlatformCompatIgnore>Linux;macOS</PlatformCompatIgnore>
</PropertyGroup>
```

Se o código for voltado para várias plataformas e você desejar aproveitar uma API que não tem suporte em algumas delas, poderá proteger essa parte do código com uma instrução `if`:

```
if (RuntimeInformation.IsOSPlatform(OSPlatform.Windows))
{
     var w = Console.WindowWidth;
     // More code
}
```

Você também pode compilar condicionalmente por estrutura/sistema operacional de destino, mas, no momento, precisa fazer isso manualmente.

Diagnóstico com suporte

Atualmente, o analisador trata dos seguintes casos:

- Uso de uma API padrão do .NET que gera PlatformNotSupportedException (PC001).
- Uso de uma API padrão do .NET que não está disponível no .NET Framework 4.6.1 (PC002).
- Uso de uma API nativa que não existe no UWP (PC003).
- Uso das APIs Delegate.BeginInvoke e EndInvoke (PC004).
- Uso de uma API que está marcada como preterida (DEXXXX).

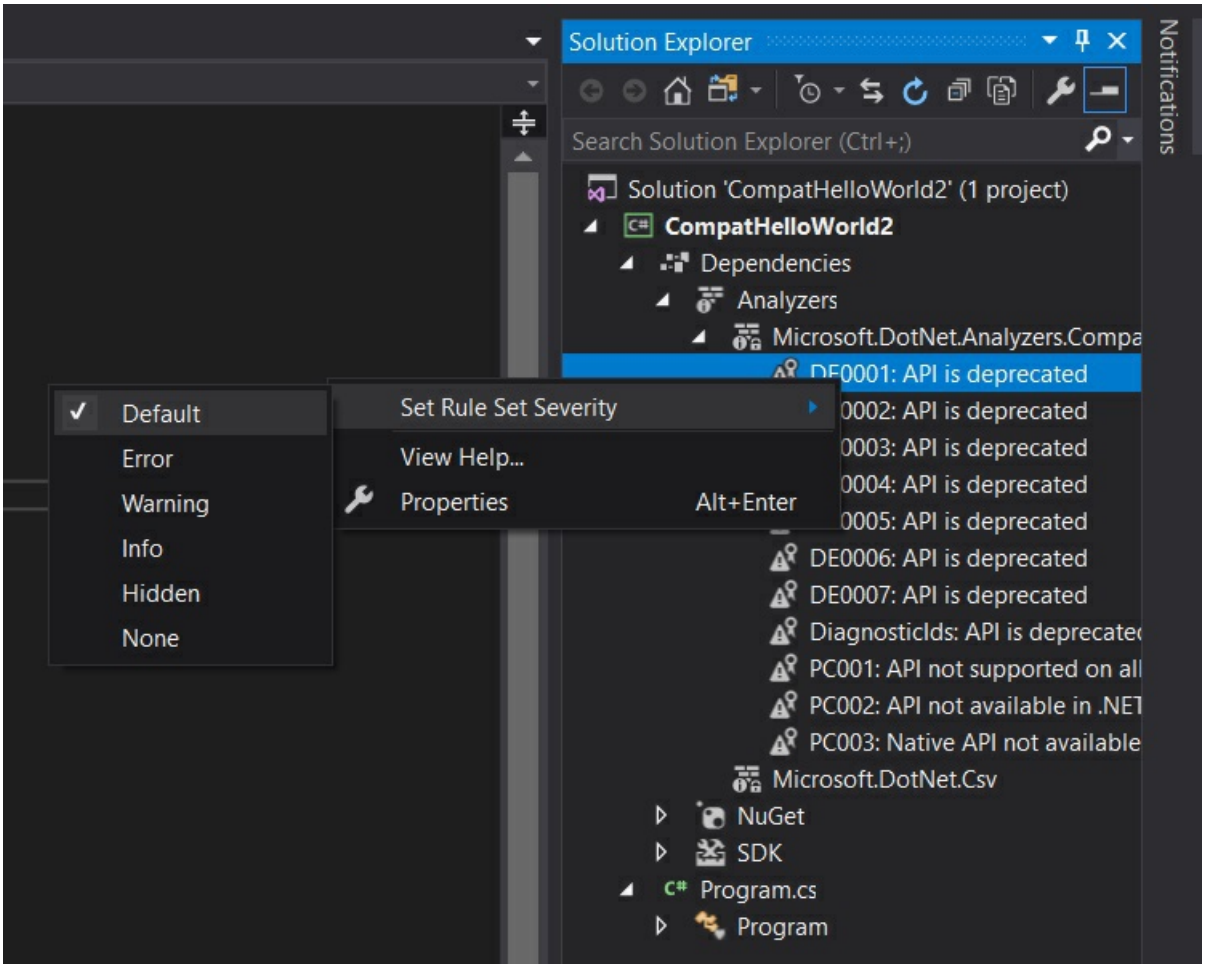
Máquina de CI

Todos esses diagnósticos estão disponíveis não só no IDE, mas também na linha de comando, como parte da

criação do projeto, o que inclui o servidor de CI.

Configuração

O usuário decide como o diagnóstico deve ser tratado: como avisos, erros, sugestões ou ser desligado. Por exemplo, como arquiteto, você pode decidir que problemas de compatibilidade devem ser tratados como erros, chamadas a algumas APIs preteridas geram avisos, enquanto outras só geram sugestões. Você pode configurar isso separadamente por ID de diagnóstico e por projeto. Para fazer isso no **Gerenciador de Soluções**, navegue até o nó **Dependências** em seu projeto. Expanda os nós**Dependencies** > **Analyzers** > **Microsoft.DotNet.Analyzers.Compatibility**. Clique com o botão direito do mouse na ID de diagnóstico, selecione **Definir Severidade de Conjunto de Regras** e selecione a opção desejada.



Consulte também

- Postagem de blog Introduzindo o analisador de API.
- Vídeo de demonstração no YouTube Analisador de API.

O .NET Portability Analyzer

24/10/2019 • 10 minutes to read • Edit Online

Deseja fazer suas bibliotecas serem compatíveis com multiplataforma? Deseja ver quanto trabalho é necessário para fazer seu .NET Framework aplicativo ser executado no .NET Core? O .net Portability Analyzer é uma ferramenta que analisa os assemblies e fornece um relatório detalhado sobre as APIs do .NET que estão faltando para que os aplicativos ou bibliotecas sejam portáteis em suas plataformas .net de destino especificadas. O analisador de portabilidade é oferecido como uma extensão do Visual Studio, que analisa um assembly por projeto e como um aplicativo de console ApiPort, que analisa os assemblies por arquivos ou diretórios especificados.

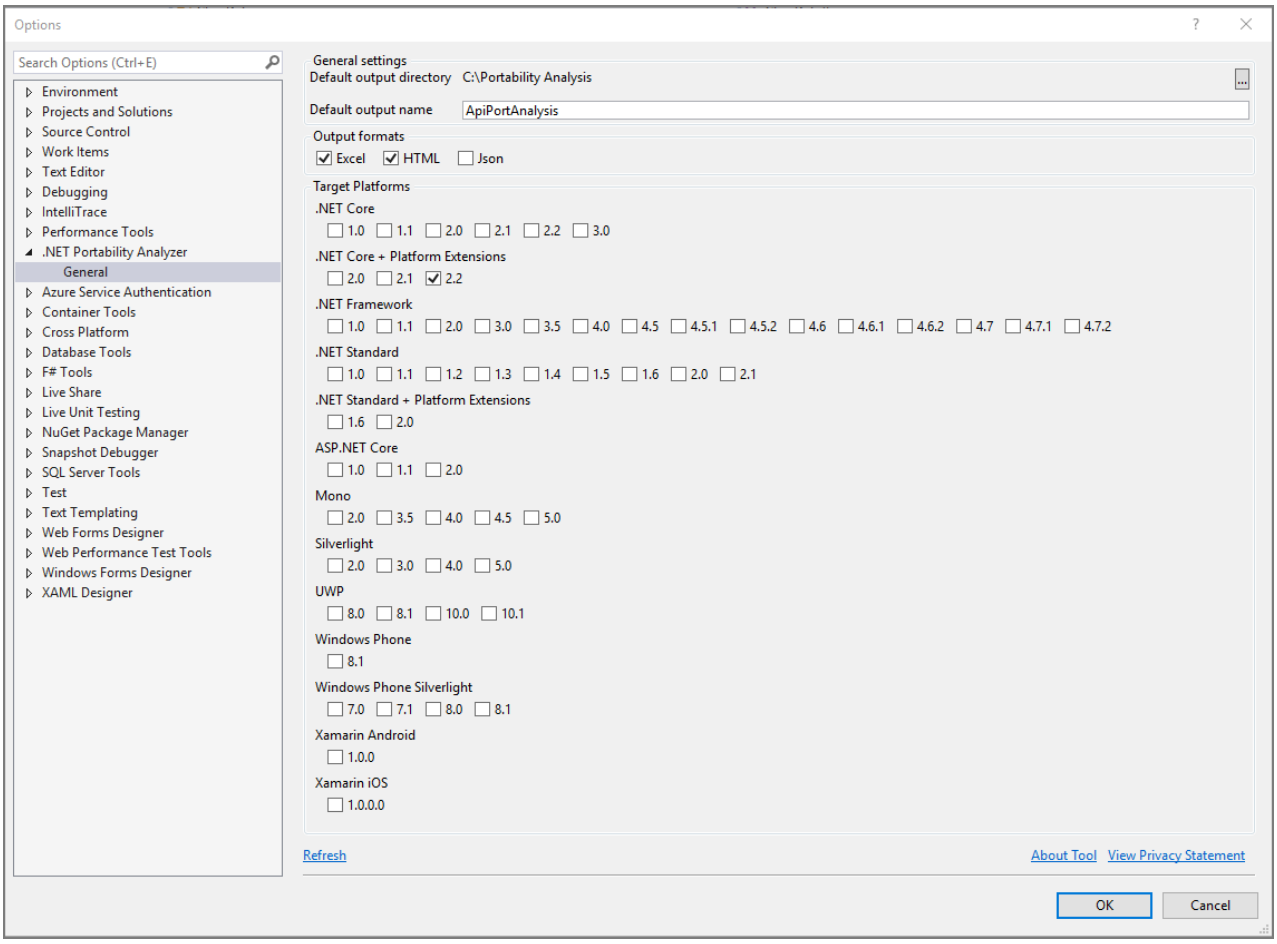
Depois de converter seu projeto para direcionar a nova plataforma, como o .NET Core, você pode usar a ferramenta Analisador de API baseada em Roslyn para identificar as APIs que geram PlatformNotSupportedException exceções e outros problemas de compatibilidade.

Destinos comuns

- .NET Core: tem um design modular, utiliza lado a lado e está direcionado a cenários de plataforma cruzada. O lado a lado permite adotar novas versões do .NET Core sem interromper outros aplicativos. Se sua meta é portar seu aplicativo para as plataformas cruzadas compatíveis com o .NET Core, esse é o destino recomendado.
- . Net Standard: inclui as APIs de .net Standard disponíveis em todas as implementações do .net. Se sua meta é fazer sua biblioteca ser executada em todas as plataformas compatíveis com o .NET, esse é o destino recomendado.
- ASP.NET Core: uma estrutura da Web moderna criada no .NET Core. Se sua meta é portar seu aplicativo Web para o .NET Core para dar suporte a várias plataformas, esse é o destino recomendado.
- Extensões de plataforma.NET Core +: inclui as APIs do .NET Core, além do pacote de compatibilidade do Windows, que fornece muitas das .NET Framework tecnologias disponíveis. Esse é um destino recomendado para portar seu aplicativo do .NET Framework para o .NET Core no Windows.
- .NET Standard + extensões de plataforma: inclui as APIs de .net Standard além do pacote de compatibilidade do Windows, que fornece muitas das .NET Framework tecnologias disponíveis. Esse é um destino recomendado para portar sua biblioteca do .NET Framework para o .NET Core no Windows.

Como usar o Analisador de Portabilidade do .NET

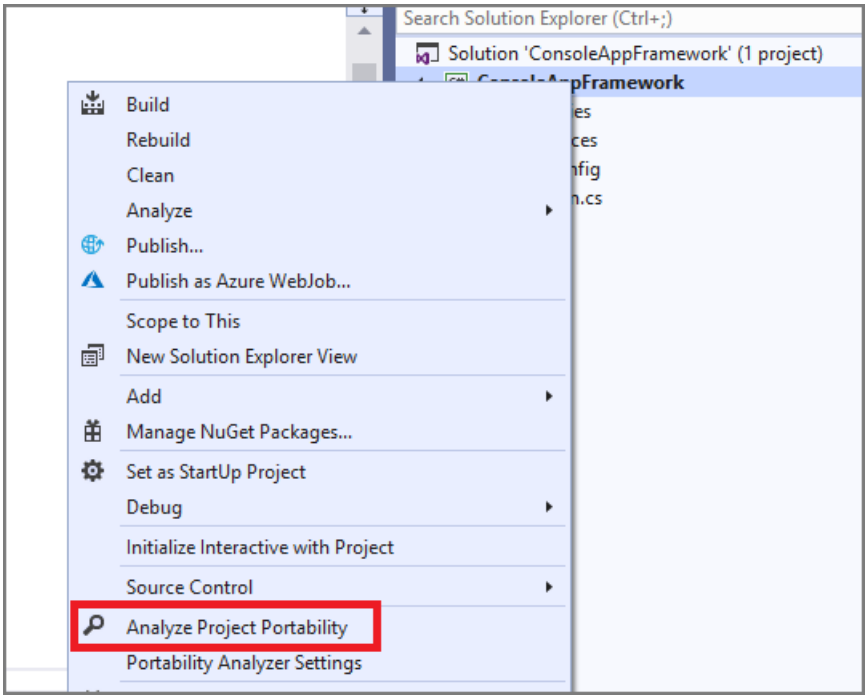
Para começar a usar o Analisador de Portabilidade do .NET no Visual Studio, primeiro é necessário baixar e instalar a extensão do Visual Studio Marketplace. Ele funciona no Visual Studio 2017 e versões posteriores. É possível configurá-la no Visual Studio por meio de **Analisar** > **Configurações do Analisador de Portabilidade** e selecionar as Plataformas de Destino, que são as versões/plataformas do .NET que você deseja avaliar as lacunas de portabilidade, comparando com a plataforma/versão com a qual seu assembly atual foi criado.



Também é possível usar o aplicativo de console ApiPort, baixando-o do repositório ApiPort. É possível usar a opção de comando `listTargets` para exibir a lista de destino disponível e escolher as plataformas de destino especificando a opção de comando `-t` ou `--target`.

Analisar portabilidade

Para analisar todo o projeto no Visual Studio, clique com o botão direito do mouse no projeto no **Gerenciador de Soluções** e selecione **Analisar Portabilidade do Assembly**. Caso contrário, acesse o menu **Analyze** e selecione **Analisar Portabilidade do Assembly**. Em seguida, selecione o executável do seu projeto ou DLL.



Também é possível usar o aplicativo de console ApiPort.

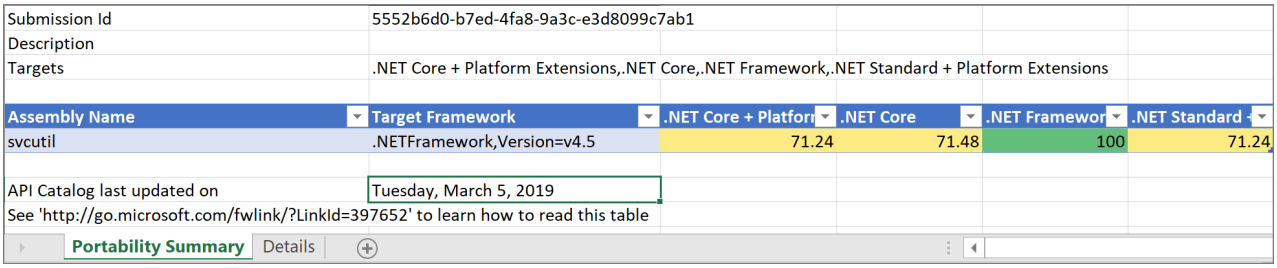
- Digite o seguinte comando para analisar o diretório atual: `ApiPort.exe analyze -f .`
- Para analisar uma lista específica de arquivos. dll, digite o seguinte comando:
  `ApiPort.exe analyze -f first.dll -f second.dll -f third.dll`
- Execute `ApiPort.exe -?` para obter mais ajuda

É recomendável que você inclua todos os arquivos exe e dll relacionados que você tem e que deseje portar e exclua os arquivos do qual seu aplicativo depende, mas que você não tem e não pode portar. Isso fará com que o relatório de portabilidade seja o mais relevante.

Exibir e interpretar o resultado da portabilidade

Apenas as APIs que não são compatíveis com uma Plataforma de Destino são exibidas no relatório. Após executar a análise no Visual Studio, você verá o link do arquivo do relatório de portabilidade do .NET aparecer. Se você usou o aplicativo de console ApiPort, o relatório de portabilidade do .NET é salvo como um arquivo no formato especificado. O padrão está em um arquivo Excel (*.xlsx*) em seu diretório atual.

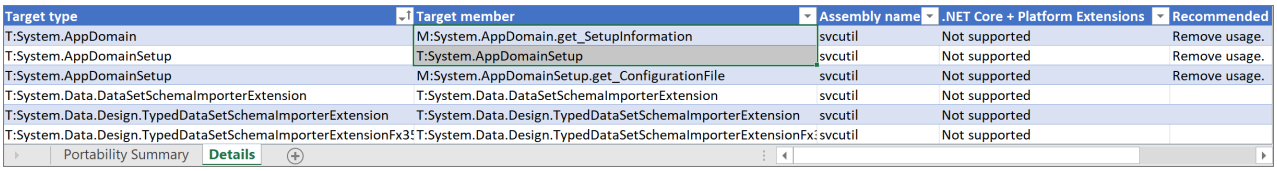
Resumo de Portabilidade



Submission Id	5552b6d0-b7ed-4fa8-9a3c-e3d8099c7ab1				
Description					
Targets	.NET Core + Platform Extensions,.NET Core,.NET Framework,.NET Standard + Platform Extensions				
Assembly Name	**Target Framework**	**.NET Core + Platfor**	**.NET Core**	**.NET Framewor**	**.NET Standard**
svcutil	.NETFramework,Version=v4.5	71.24	71.48	100	71.24
API Catalog last updated on	Tuesday, March 5, 2019				
See 'http://go.microsoft.com/fwlink/?LinkId=397652' to learn how to read this table					

A seção Resumo de Portabilidade do relatório mostra o percentual de portabilidade para cada assembly incluído na execução. No exemplo anterior, 71,24% das APIs do .NET Framework usadas no aplicativo `svcutil` estão disponíveis no .NET Core + Extensões de plataforma. Se você executar a ferramenta Analisador de Portabilidade do .NET em vários assemblies, cada um deles deverá ter uma linha no relatório Resumo de Portabilidade.

Detalhes

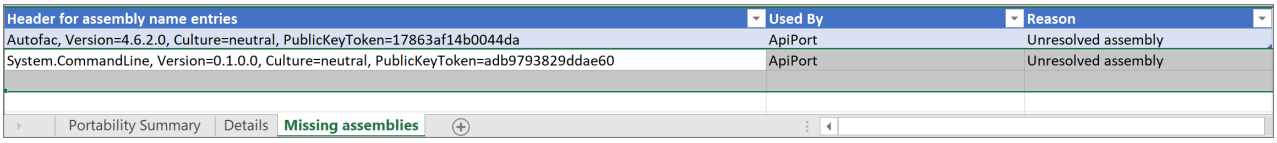


Target type	Target member	Assembly name	.NET Core + Platform Extensions	Recommended
T:System.AppDomain	M:System.AppDomain.get_SetupInformation	svcutil	Not supported	Remove usage.
T:System.AppDomainSetup	T:System.AppDomainSetup	svcutil	Not supported	Remove usage.
T:System.AppDomainSetup	M:System.AppDomainSetup.get_ConfigurationFile	svcutil	Not supported	Remove usage.
T:System.Data.DataSetSchemaImporterExtension	T:System.Data.DataSetSchemaImporterExtension	svcutil	Not supported	
T:System.Data.Design.TypedDataSetSchemaImporterExtension	T:System.Data.Design.TypedDataSetSchemaImporterExtension	svcutil	Not supported	
T:System.Data.Design.TypedDataSetSchemaImporterExtensionFx3	T:System.Data.Design.TypedDataSetSchemaImporterExtensionFx	svcutil	Not supported	

A seção de **detalhes** do relatório lista as APIs ausentes em qualquer uma das **plataformas de destino**selecionadas.

- Tipo de destino: o tipo tem uma API ausente de uma Plataforma de Destino
- Membro de destino: o método está ausente de uma Plataforma de Destino
- Nome do assembly: o assembly do .NET Framework no qual a API ausente reside.
- Cada uma das plataformas de destino selecionadas é uma coluna, como ".NET Core": o valor "sem suporte" significa que a API não tem suporte nesta plataforma de destino.
- Alterações recomendadas: a API ou a tecnologia recomendada para a qual alterar. No momento, esse campo está vazio ou desatualizado para muitas APIs. Devido ao grande número de APIs, é um grande desafio mantê-las em funcionamento. Estamos examinando soluções alternativas para fornecer informações úteis aos clientes.

Assemblies Ausentes



Header for assembly name entries	Used By	Reason
Autofac, Version=4.6.2.0, Culture=neutral, PublicKeyToken=17863af14b0044da	ApiPort	Unresolved assembly
System.CommandLine, Version=0.1.0.0, Culture=neutral, PublicKeyToken=adb9793829ddae60	ApiPort	Unresolved assembly

Você pode encontrar uma seção Assemblies Ausentes em seu relatório. Ele informa que essa lista de assemblies é referenciada por seus assemblies analisados e não foi analisada. Se for um assembly de sua propriedade, inclua-o

na execução do analisador de portabilidade de Api para que você possa obter um relatório de portabilidade detalhado no nível da API para ele. Se for uma biblioteca de terceiros, pesquise se eles têm uma versão mais recente compatível com sua plataforma de destino. Em caso afirmativo, considere migrar para a versão mais recente. Por fim, seria esperado que essa lista incluísse todos os assemblies de terceiros do qual seu aplicativo depende e confirmasse que eles têm uma versão compatível com sua plataforma de destino.

Para obter mais informações sobre o Analisador de Portabilidade do .NET, acesse a documentação do GitHub e assista ao vídeo A Brief Look at the .NET Portability Analyzer (Uma análise breve do Analisador de Portabilidade do .NET) do Channel 9.

O Analisador do .NET Framework

27/11/2019 • 9 minutes to read • Edit Online

Você pode usar o Analisador do .NET Framework para localizar problemas potenciais no seu código de aplicativo baseado no .NET Framework. Este analisador localiza potenciais problemas e sugere correções para eles.

O analisador é executado interativamente no Visual Studio enquanto você escreve seu código ou como parte de um build de CI. Você deve adicionar o analisador ao seu projeto tão cedo quando possível no desenvolvimento. Quanto antes você encontrar potenciais problemas no seu código, mais fácil será corrigi-los. No entanto, você pode adicioná-lo a qualquer momento no ciclo de desenvolvimento. Ele encontra eventuais problemas com o código existente e avisa sobre novos problemas conforme você dá sequência ao desenvolvimento.

Instalar e configurar o Analisador do .NET Framework

Os analisadores de .NET Framework devem ser instalados como um pacote NuGet em todos os projetos em que você deseja executá-los. Somente um desenvolvedor precisa adicioná-los ao projeto. O pacote de analisador é uma dependência de projeto e será executado no computador de cada um dos desenvolvedores assim que ele tiver a solução atualizada.

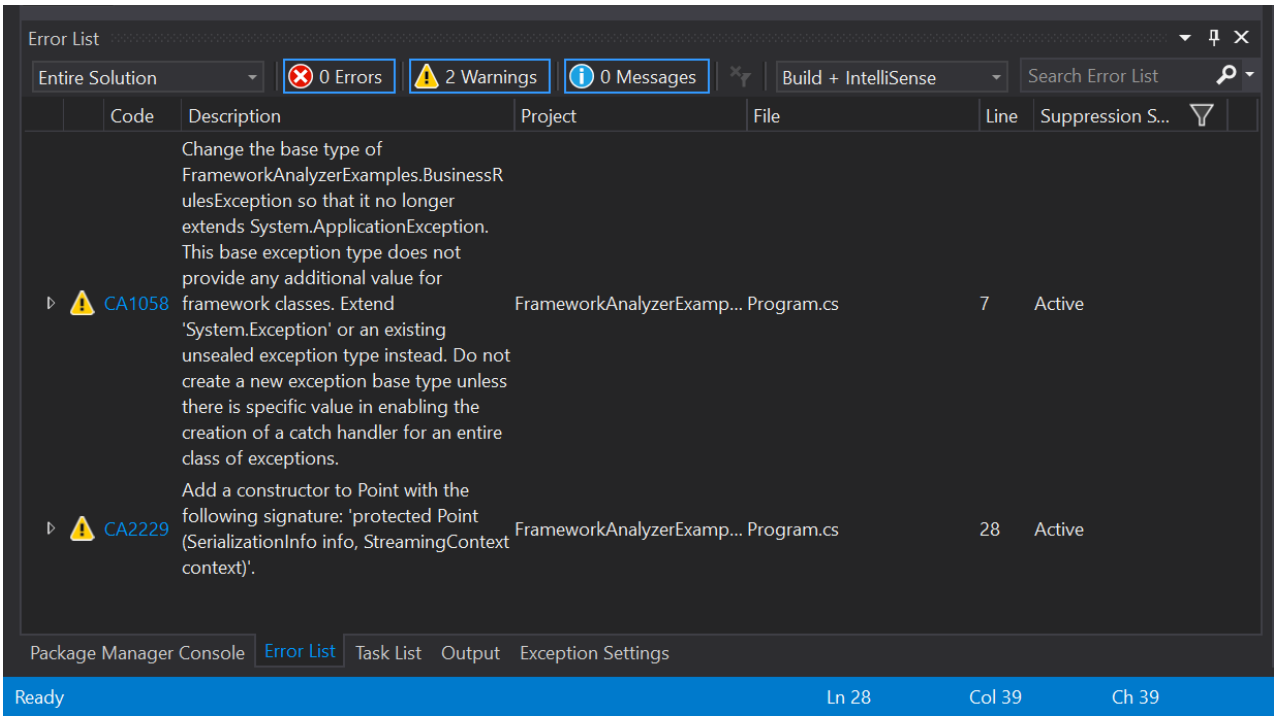
O Analisador do .NET Framework é fornecido no pacote do NuGet Microsoft.NetFramework.Analyzers. Esse pacote fornece apenas os analisadores específicos para o .NET Framework, que inclui os analisadores de segurança. Na maioria dos casos, você desejará o pacote do NuGet Microsoft.CodeAnalysis.FxCopAnalyzers. O pacote de agregação FxCopAnalyzers contém todos os analisadores de estrutura incluídos no pacote Framework.Analyzers, bem como os analisadores a seguir:

- Microsoft.CodeQuality.Analyzers: fornece diretrizes gerais e diretrizes para APIs do .NET Standard
- Microsoft.NetCore.Analyzers: fornece analisadores específicos para as APIs do .NET Core.
- Text.Analyzers: fornece diretrizes para texto incluído como código, incluindo comentários.

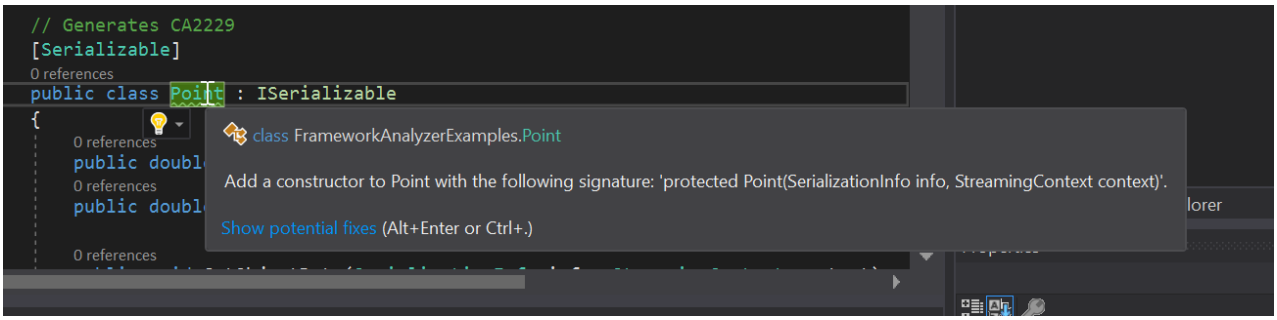
Para instalá-lo, clique com o botão direito do mouse no projeto e selecione "Gerenciar Dependências". Do explorador do NuGet, pesquise por "NetFramework Analyzer" ou, se você preferir, "Fx Cop Analyzer". Instale a versão estável mais recente em todos os projetos na solução.

Usar o Analisador do .NET Framework

Depois de instalar o pacote do NuGet, compile a solução. O analisador relatará eventuais problemas que ele localize na base de código. Os problemas são relatados como avisos na janela Lista de Erros do Visual Studio, conforme mostrado na imagem a seguir:



Ao escrever código, você vê linhas onduladas sob qualquer problema potencial existente nele. Passe o mouse sobre qualquer problema e você verá detalhes sobre o problema e sugestões de eventuais correções possíveis, conforme mostrado na imagem a seguir:



Os analisadores examinam o código em sua solução e fornecem a você uma lista de avisos para qualquer um desses problemas:

CA1058: os tipos não devem estender determinados tipos base

Há um pequeno número de tipos do .NET Framework dos quais você não deve derivar diretamente.

Categoria: design

Severidade: Aviso

Informações adicionais: CA:1058: os tipos não devem estender determinados tipos base

CA2153: não capturar exceções de estado corrompido

A captura de exceções de estado corrompido pode mascarar erros (como violações de acesso), resultando em um estado inconsistente de execução ou tornando mais fácil para invasores comprometerem um sistema. Em vez disso, capture e manipule um conjunto mais específico de tipos de exceção ou gere novamente a exceção

Categoria: segurança

Severidade: Aviso

Informações adicionais: ## CA2153: não capturar exceções de estado corrompido

{1>{2>CA2229: Implementar construtores de serialização<2}<1}

O analisador gera este aviso quando você cria um tipo que implementa a interface ISerializable, mas não define o

construtor de serialização necessário. Para corrigir uma violação dessa regra, implemente o construtor de serialização. Para uma classe lacrada, torne o construtor particular; do contrário, deixe-o protegido. O construtor de serialização tem a seguinte assinatura:

```
public class MyItemType
{
    // The special constructor is used to deserialize values.
    public MyItemType(SerializationInfo info, StreamingContext context)
    {
        // implementation removed.
    }
}
```

Categoria: uso

Severidade: Aviso

Informações adicionais: CA2229: implementar construtores de serialização

CA2235: marcar todos os campos não serializáveis

Um campo de instância de um tipo que não seja serializável é declarado em um tipo que é serializável. Você deve marcar explicitamente esse campo com o NonSerializedAttribute para corrigir este aviso.

Categoria: uso

Severidade: Aviso

Informações adicionais: CA2235: marcar todos os campos não serializáveis

CA2237: marcar tipos ISerializable com serializable

Para serem reconhecidos pelo Common Language Runtime como serializáveis, os tipos devem ser marcados usando-se o atributo SerializableAttribute, mesmo quando o tipo usa uma rotina de serialização personalizada implementando a interface ISerializable.

Categoria: uso

Severidade: Aviso

Informações adicionais: CA2237: marcar tipos ISerializable com serializable

CA3075: processamento de DTD inseguro em XML

Se você usar instâncias de DtdProcessing inseguras ou referenciar origens de entidade externa, o analisador poderá aceitar a entrada não confiável e divulgar as informações confidenciais para invasores.

Categoria: segurança

Severidade: Aviso

Informações adicionais: A3075: processamento de DTD inseguro em XML

CA5350: não usar algoritmos de criptografia fracos

Algoritmos de criptografia degradam-se ao longo do tempo, conforme os ataques se tornam mais avançados. Dependendo do tipo e do aplicativo desse algoritmo de criptografia, uma maior degradação de sua intensidade criptográfica poderá permitir que invasores leiam mensagens criptografadas, adulterem mensagens criptografadas, forjem assinaturas digitais, violem o conteúdo de hash ou comprometam qualquer sistema criptográfico baseado neste algoritmo. Para criptografia, use um algoritmo AES (AES-256, AES-192 e AES-128 são aceitáveis) com um comprimento de chave maior ou igual a 128 bits. Para o hash, use uma função de hash da família SHA-2, tal como SHA-2 512, SHA-2 384 ou SHA-2 256.

Categoria: segurança

Severidade: Aviso

Informações adicionais: CA5350: não usar algoritmos de criptografia fracos

CA5351: não usar algoritmos de criptografia violados

Existe um ataque que torna a quebra desse algoritmo computacionalmente viável. Isso permite que os invasores violem as garantias criptográficas que ele foi projetado para fornecer. Dependendo do tipo e do aplicativo desse algoritmo de criptografia, isso poderá permitir que invasores leiam e adulterem mensagens criptografadas, forjem assinaturas digitais, violem o conteúdo de hash ou comprometam qualquer sistema de criptografia baseado neste algoritmo. Para criptografia, use um algoritmo AES (AES-256, AES-192 e AES-128 são aceitáveis) com um comprimento de chave maior ou igual a 128 bits. Para o hash, use uma função de hash da família SHA-2, tal como SHA512, SHA384 ou SHA256. Para assinaturas digitais, use RSA com um comprimento de chave maior ou igual a 2.048 bits, ou então ECDSA com um comprimento de chave maior ou igual a 256 bits.

Categoria: segurança

Severidade: Aviso

Informações adicionais: CA5351: não usar algoritmos de criptografia violados

Tratando e gerando exceções no .NET

31/10/2019 • 5 minutes to read • Edit Online

Aplicativos devem ser capazes de tratar de erros que ocorrem durante a execução de uma maneira consistente. O .NET fornece um modelo para notificar aplicativos sobre erros de maneira uniforme: operações do .NET indicam falhas por meio da geração de exceções.

Exceções

Uma exceção é qualquer condição de erro ou comportamento inesperado encontrado por um programa em execução. Exceções podem ser geradas devido a uma falha em seu código ou no código que você chama (como uma biblioteca compartilhada), recursos do sistema operacional não disponíveis, condições inesperadas encontradas pelo tempo de execução (como código que não pode ser verificado) e assim por diante. Seu aplicativo pode se recuperar de algumas dessas condições, mas não de outras. Embora você possa se recuperar da maioria das exceções de aplicativo, não é possível recuperar-se da maioria das exceções de tempo de execução.

No .NET, uma exceção é um objeto herdado da classe System.Exception. Uma exceção é lançada de uma área do código em que ocorreu um problema. A exceção é passada pilha acima até que o aplicativo trate dela ou o programa seja encerrado.

Métodos de tratamento de exceção vs. tratamento de erro tradicional

Tradicionalmente, o modelo de tratamento de erro da linguagem confiava na forma exclusiva da linguagem de detectar erros e localizar manipuladores para eles ou no mecanismo de tratamento de erro fornecido pelo sistema operacional. A maneira como o .NET implementa o tratamento de exceção oferece as seguintes vantagens:

- O lançamento e tratamento de exceção funciona da mesma maneira para linguagens de programação .NET.
- Não requer nenhuma sintaxe de linguagem específica para tratamento de exceção, mas permite que cada linguagem defina sua própria sintaxe.
- Exceções podem ser geradas pelos limites de processo e até mesmo de computador.
- O código de tratamento de exceção pode ser adicionado a um aplicativo para aumentar a confiabilidade do programa.

As exceções oferecem vantagens sobre outros métodos de notificação de erro, como códigos de retorno. Falhas não passam despercebidas porque se uma exceção for lançada e você não tratar dela, o tempo de execução encerra o aplicativo. Valores inválidos não continuam a se propagar através do sistema como resultado do código que não consegue verificar se há um código de retorno de falha.

Exceções comuns

A tabela a seguir lista algumas exceções comuns com exemplos do que pode causá-las.

TIPO DE EXCEÇÃO	DESCRIÇÃO	EXEMPLO
Exception	A classe base para todas as exceções.	Nenhuma (use uma classe derivada dessa exceção).

TIPO DE EXCEÇÃO	DESCRIÇÃO	EXEMPLO
IndexOutOfRangeException	Gerada pelo tempo de execução somente quando uma matriz é indexada incorretamente.	Indexar uma matriz fora do intervalo válido: `arr[arr.Length+1]`
NullReferenceException	Gerada pelo tempo de execução somente quando um objeto nulo é referenciado.	`object o = null;` `o.ToString();`
InvalidOperationException	Gerada por métodos quando em um estado inválido.	Chamar `Enumerator.MoveNext()` após a remoção de um item da coleção subjacente.
ArgumentException	A classe base para todas as exceções de argumento.	Nenhuma (use uma classe derivada dessa exceção).
ArgumentNullException	Gerada por métodos que não permitem que um argumento seja nulo.	`String s = null;` `"Calculate".IndexOf(s);`
ArgumentOutOfRangeException	Gerada por métodos que verificam se os argumentos estão em um determinado intervalo.	`String s = "string";` `s.Substring(s.Length+1);`

Consulte também

- Classe e propriedades da exceção
- Como: usar o bloco try-catch para capturar exceções
- Como: usar exceções específicas em um bloco catch
- Como: Gerar exceções explicitamente
- Como: criar exceções definidas pelo usuário
- Usando manipuladores de exceção filtrados por usuário
- Como: usar blocos Finally
- Manipulando exceções de interoperabilidade COM
- Práticas recomendadas para exceções
- O que todo desenvolvedor precisa saber sobre exceções no tempo de execução

Assemblies no .NET

08/11/2019 • 13 minutes to read • Edit Online

Os assemblies formam as unidades fundamentais de implantação, controle de versão, reutilização, escopo de ativação e permissões de segurança para o. Aplicativos baseados em rede. Um assembly é uma coleção de tipos e recursos compilados para funcionar juntos e formar uma unidade lógica de funcionalidade. Os assemblies assumem a forma de arquivos executáveis (. *exe*) ou Dynamic Link Library (. *dll*) e são os blocos de construção de aplicativos .net. Eles oferecem ao Common Language Runtime as informações de que ele precisa para estar ciente das implementações de tipo.

No .NET Core e .NET Framework, você pode criar um assembly a partir de um ou mais arquivos de código-fonte. No .NET Framework, os assemblies podem conter um ou mais módulos. Isso permite que projetos maiores sejam planejados para que vários desenvolvedores possam trabalhar em módulos ou arquivos de código-fonte separados, que são combinados para criar um único assembly. Para obter mais informações sobre módulos, consulte como compilar um assembly de multiarquivos.

Os assemblies têm as seguintes propriedades:

- Os assemblies são implementados como arquivos . *exe* ou . *dll* .
- Para bibliotecas direcionadas ao .NET Framework, você pode compartilhar assemblies entre aplicativos, colocando-os no cache de assembly global (GAC). Você deve obter os assemblies de nome forte antes de poder incluí-los no GAC. Para obter mais informações, consulte assemblies com nomes fortes.
- Os assemblies serão carregados na memória somente se forem necessários. Se eles não forem usados, eles não serão carregados. Isso significa que os assemblies podem ser uma maneira eficiente de gerenciar recursos em projetos grandes.
- Você pode obter informações programaticamente sobre um assembly usando reflexão. Para obter mais informações, confira Reflexão (C#) ou Reflexão (Visual Basic).
- Você pode carregar um assembly apenas para inspecioná-lo usando a classe MetadataLoadContext no .NET Core e os métodos Assembly.ReflectionOnlyLoad ou Assembly.ReflectionOnlyLoadFrom no .NET Core e .NET Framework.

Assemblies no Common Language Runtime

Os assemblies fornecem o Common Language Runtime com as informações de que ele precisa para estar ciente das implementações de tipo. Para o runtime, um tipo não existe fora do contexto de um assembly.

Um assembly define as seguintes informações:

- Código que o Common Language Runtime é executado. Observe que cada assembly pode ter apenas um ponto de entrada: `DllMain` , `WinMain` ou `Main` .
- Limite de segurança. Um assembly é a unidade na qual as permissões são solicitadas e concedidas. Para obter mais informações sobre limites de segurança em assemblies, consulte considerações de segurança do assembly.
- Limite de tipo. A identidade de cada tipo inclui o nome do assembly no qual ele reside. Um tipo chamado `MyType` , carregado no escopo de um assembly, não é o mesmo de um tipo chamado `MyType` , carregado no escopo de outro assembly.
- Limite de escopo de referência. O manifesto do assembly tem metadados que são usados para resolver

tipos e satisfazer solicitações de recursos. O manifesto especifica os tipos e recursos a serem expostos fora do assembly e enumera outros assemblies dos quais ele depende. O código MSIL (Microsoft Intermediate Language) em um arquivo executável portátil (PE) não será executado, a menos que tenha um manifesto de assemblyassociado.

- Limite de versão. O assembly é a menor unidade com controle de versão no Common Language Runtime. Todos os tipos e recursos no mesmo assembly têm controle de versão como uma unidade. O manifesto do assembly descreve as dependências de versão que você especifica para quaisquer assemblies dependentes. Para obter mais informações sobre o controle de versão, consulte controle de versão do assembly.
- Unidade de implantação. Quando um aplicativo é iniciado, somente os assemblies que o aplicativo chama inicialmente devem estar presentes. Outros assemblies, como assemblies que contêm recursos de localização ou classes utilitárias, podem ser recuperados sob demanda. Isso permite que os aplicativos sejam simples e finos quando baixados pela primeira vez. Para obter mais informações sobre a implantação de assemblies, consulte implantar aplicativos.
- Unidade de execução lado a lado. Para obter mais informações sobre como executar várias versões de um assembly, consulte assemblies e execução lado a lado.

Criar um assembly

Assemblies podem ser estáticos ou dinâmicos. Assemblies estáticos são armazenados em disco em arquivos PE. Os assemblies estáticos podem incluir interfaces, classes e recursos como bitmaps, arquivos JPEG e outros arquivos de recurso. Você também pode criar assemblies dinâmicos, que são executados diretamente da memória e não são salvos em disco antes da execução. Você pode salvar assemblies dinâmicos em disco após sua execução.

Existem várias maneiras de criar assemblies. Você pode usar as ferramentas de desenvolvimento, como o Visual Studio, que podem criar arquivos . *dll* ou . *exe* . Você pode usar ferramentas no SDK do Windows para criar assemblies com módulos de outros ambientes de desenvolvimento. Você também pode usar APIs do Common Language Runtime, como System.Reflection.Emit, a fim de criar assemblies dinâmicos.

Compile assemblies Criando-os no Visual Studio, criando-os com ferramentas de interface de linha de comando do .NET Core ou compilando assemblies de .NET Framework com um compilador de linha de comando. Para obter mais informações sobre como criar assemblies usando as ferramentas de interface de linha de comando do .NET Core, consulte ferramentas de interface de linha de comando do .NET Core. Para compilar assemblies com os compiladores de linha de comando, consulte compilação de linha de comando com CSC. exe para C#, ou Build da linha de comando para Visual Basic.

> **NOTE**
> Para criar um assembly no Visual Studio, no menu **Compilar** , selecione **Compilar**.

Manifesto do assembly

Cada assembly tem um arquivo de *manifesto do assembly* . Semelhante a um sumário, o manifesto do assembly contém:

- A identidade do assembly (seu nome e versão).
- Uma tabela de arquivos que descreve todos os outros arquivos que compõem o assembly, como outros assemblies que você criou que o arquivo . *exe* ou . *dll* depende, arquivos de bitmap ou arquivos Leiame.
- Uma *lista de referências de assembly*, que é uma lista de todas as dependências externas, como . *dll*s ou outros arquivos. As referências de assembly contêm referências a objetos globais e privados. Os objetos globais estão disponíveis para todos os outros aplicativos. No .NET Core, os objetos globais são acoplados a

um tempo de execução específico do .NET Core. No .NET Framework, os objetos globais residem no GAC (cache de assembly global). *System. IO. dll* é um exemplo de um assembly no GAC. Os objetos privados devem estar em um nível de diretório no ou abaixo do diretório no qual seu aplicativo está instalado.

Como os assemblies contêm informações sobre conteúdo, controle de versão e dependências, os aplicativos que os usam não precisam depender de fontes externas, como o registro em sistemas Windows, para funcionar corretamente. Os assemblies reduzem conflitos . *dll* e tornam seus aplicativos mais confiáveis e fáceis de implantar. Em muitos casos, você pode instalar um aplicativo baseado em .NET simplesmente copiando seus arquivos para o computador de destino. Para obter mais informações, consulte manifesto do assembly.

Adicionar uma referência a um assembly

Para usar um assembly em um aplicativo, você deve adicionar uma referência a ele. Depois que um assembly é referenciado, todos os tipos, propriedades, métodos e outros membros acessíveis de seus namespaces estão disponíveis para seu aplicativo como se o código fosse parte de seu arquivo de origem.

> **NOTE**
> A maioria dos assemblies da Biblioteca de Classes .NET é referenciada automaticamente. Se um assembly do sistema não for referenciado automaticamente, para o .NET Core, você poderá adicionar uma referência ao pacote NuGet que contém o assembly. Use o Gerenciador de pacotes NuGet no Visual Studio ou adicione um <PackageReference elemento > para o assembly para o projeto . *csproj* ou . *vbproj* . No .NET Framework, você pode adicionar uma referência ao assembly usando a caixa de diálogo **Adicionar referência** no Visual Studio ou usando a opção de linha de comando `-reference` para os C# compiladores do ou do Visual Basic .

No C#, você pode usar duas versões do mesmo assembly em um único aplicativo. Para obter mais informações, consulte alias externo.

Conteúdo relacionado

TÍTULO	DESCRIÇÃO
Conteúdo do assembly	Elementos que compõem um assembly.
Manifesto do assembly	Dados no manifesto do assembly e como eles são armazenados em assemblies.
Cache de assembly global	Como o GAC armazena e usa assemblies.
Assemblies de nome forte	Características de assemblies de nome forte.
Considerações de segurança do assembly	Como funciona a segurança com assemblies.
Controle de versão do assembly	Visão geral da política de controle de versão do .NET Framework.
Posicionamento do assembly	Onde localizar os assemblies.
Assemblies e execução lado a lado	Use várias versões do tempo de execução ou um assembly simultaneamente.
Emitir métodos e assemblies dinâmicos	Como criar assemblies dinâmicos.

TÍTULO	DESCRIÇÃO
Como o runtime localiza assemblies	Como o .NET Framework resolve referências de assembly em tempo de execução.

Referência

System.Reflection.Assembly

Consulte também

- Formato de arquivo do assembly .NET
- Assemblies no .NET
- Assemblies Friend
- Assemblies de referência
- Como carregar e descarregar assemblies
- Como usar e depurar a descargabilidade do assembly no .NET Core
- Como determinar se um arquivo é um assembly

Gerenciamento de memória e coleta de lixo no .NET

31/10/2019 • 2 minutes to read • Edit Online

Esta seção da documentação fornece informações sobre o gerenciamento de memória do .NET.

Nesta seção

Limpando recursos não gerenciados
Descreve como gerenciar e limpar corretamente os recursos não gerenciados.

Coleta de lixo
Fornece informações sobre o coletor de lixo do .NET.

Seções relacionadas

Guia de desenvolvimento

Visão geral de tipos genéricos

23/10/2019 • 5 minutes to read • Edit Online

Desenvolvedores usam genéricos o tempo todo no .NET, seja implícita ou explicitamente. Ao usar o LINQ no .NET, você já percebeu que você está trabalhando com IEnumerable<T>? Ou, no caso de você já ter visto uma amostra online de um "repositório genérico" para conversar com bancos de dados usando o Entity Framework, você já viu que a maioria dos métodos retorna IQueryable<T>? Talvez você tenha se perguntado o que é o **T** nesses exemplos e por que ele está lá.

Introduzidos pela primeira vez no .NET Framework 2.0, os **genéricos** são essencialmente um "modelo de código" que permite aos desenvolvedores definir estruturas de dados fortemente tipadas sem se comprometer com um tipo de dados real. Por exemplo, List<T> é uma coleção de genéricos que pode ser declarada e usada com qualquer tipo, como `List<int>`, `List<string>` ou `List<Person>`.

Para entender por que os genéricos são úteis, vamos dar uma olhada em uma classe específica antes e depois adicionar os genéricos: ArrayList. No .NET Framework 1.0, os elementos `ArrayList` eram do tipo Object. Isso significava que qualquer elemento adicionado silenciosamente era convertido em um `Object`. O mesmo aconteceria ao ler os elementos da lista. Esse processo é conhecido como conversão boxing e unboxing e afeta o desempenho. Mais do que isso, no entanto, não é possível determinar o tipo de dados na lista no tempo de compilação. Isso resulta em um código frágil. Genéricos resolvem esse problema definindo o tipo de dados que cada instância de lista conterá. Resumindo, você só pode adicionar inteiros a `List<int>` e só pode adicionar Pessoas a `List<Person>`.

Genéricos também estão disponíveis em tempo de execução. Isso significa que o tempo de execução sabe que tipo de estrutura de dados você está usando e pode armazená-la na memória com mais eficiência.

O exemplo a seguir é um pequeno programa que ilustra a eficiência de saber o tipo da estrutura de dados em tempo de execução:

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

  namespace GenericsExample {
    class Program {
      static void Main(string[] args) {
        //generic list
        List<int> ListGeneric = new List<int> { 5, 9, 1, 4 };
        //non-generic list
        ArrayList ListNonGeneric = new ArrayList { 5, 9, 1, 4 };
        // timer for generic list sort
        Stopwatch s = Stopwatch.StartNew();
        ListGeneric.Sort();
        s.Stop();
        Console.WriteLine($"Generic Sort: {ListGeneric}  \n Time taken: {s.Elapsed.TotalMilliseconds}ms");

        //timer for non-generic list sort
        Stopwatch s2 = Stopwatch.StartNew();
        ListNonGeneric.Sort();
        s2.Stop();
        Console.WriteLine($"Non-Generic Sort: {ListNonGeneric}  \n Time taken:
{s2.Elapsed.TotalMilliseconds}ms");
        Console.ReadLine();
      }
    }
  }
```

Este programa produz uma saída semelhante à seguinte:

```
Generic Sort: System.Collections.Generic.List`1[System.Int32]
 Time taken: 0.0034ms
Non-Generic Sort: System.Collections.ArrayList
 Time taken: 0.2592ms
```

A primeira coisa que você pode observar aqui é que classificar a lista genérica é significativamente mais rápido do que classificar a lista não genérica. Você também observará que o tipo de lista genérico é distinto ([System.Int32]) enquanto o tipo da lista não genérico é generalizado. Como o tempo de execução sabe que o genérico `List<int>` é do tipo Int32, ele pode armazenar elementos de lista em uma matriz de inteiros subjacente na memória, enquanto o `ArrayList` não genérico tem que converter cada elemento da lista em um objeto. Como este exemplo mostra, as conversões extras levam tempo e reduzem a velocidade da classificação da lista.

Uma vantagem adicional de o tempo de execução saber o tipo de seu genérico é uma melhor experiência de depuração. Quando você está depurando um genérico em C#, você sabe que tipo de cada elemento está na sua estrutura de dados. Sem os genéricos, você não faria ideia de qual o tipo de cada elemento.

Consulte também

- Guia de Programação em C# – Genéricos

Delegados e lambdas

23/10/2019 • 7 minutes to read • Edit Online

Os delegados definem um tipo, que especifica uma assinatura de método específica. Um método (estático ou instância) que satisfaz essa assinatura pode ser atribuído a uma variável desse tipo, que então é chamada diretamente (com os argumentos apropriados) ou passada como um argumento para outro método e, então, chamada. O exemplo a seguir demonstra o uso de um delegado.

```
using System;
using System.Linq;

public class Program
{
    public delegate string Reverse(string s);

    static string ReverseString(string s)
    {
        return new string(s.Reverse().ToArray());
    }

    static void Main(string[] args)
    {
        Reverse rev = ReverseString;

        Console.WriteLine(rev("a string"));
    }
}
```

- A linha `public delegate string Reverse(string s);` cria um tipo de delegado de determinada assinatura, nesse caso, um método que usa um parâmetro de cadeia de caracteres e, em seguida, retorna um parâmetro de cadeia de caracteres.
- O método `static string ReverseString(string s)`, que tem exatamente a mesma assinatura do tipo de delegado definido, implementa o delegado.
- A linha `Reverse rev = ReverseString;` mostra que você pode atribuir um método a uma variável do tipo de delegado correspondente.
- A linha `Console.WriteLine(rev("a string"));` demonstra como usar uma variável de um tipo de delegado para invocar o delegado.

Para simplificar o processo de desenvolvimento, o .NET inclui um conjunto de tipos de delegados que os programadores podem reutilizar, sem precisar criar novos tipos. Eles são `Func<>`, `Action<>` e `Predicate<>`, e podem ser usados em vários locais das APIs .NET sem a necessidade de definir novos tipos de delegado. Obviamente, há algumas diferenças entre os três, que você verá em suas assinaturas e que geralmente têm a ver com a maneira como eles deveriam ser usados:

- `Action<>` é usado quando é necessário executar uma ação usando os argumentos do delegado.
- `Func<>` é normalmente é usado quando você tem uma transformação à mão, ou seja, quando é necessário transformar os argumentos do delegado em um resultado diferente. As projeções são um excelente exemplo disso.
- `Predicate<>` é usado quando você precisa determinar se o argumento satisfaz a condição do delegado. Ele também pode ser escrito como um `Func<T, bool>`.

Agora, podemos pegar nosso exemplo acima e reescrevê-lo usando o delegado `Func<>` em vez de um tipo

personalizado. O programa continuará sendo executado exatamente da mesma forma.

```
using System;
using System.Linq;

public class Program
{
    static string ReverseString(string s)
    {
        return new string(s.Reverse().ToArray());
    }

    static void Main(string[] args)
    {
        Func<string, string> rev = ReverseString;

        Console.WriteLine(rev("a string"));
    }
}
```

Para este exemplo simples, ter um método definido fora do método `Main` parece um pouco supérfluo. É por isso que o .NET Framework 2.0 introduziu o conceito de **delegados anônimos**. Com suporte deles, você pode criar delegados "embutidos" sem precisar especificar tipos ou métodos adicionais. Você simplesmente embute a definição do delegado onde precisar dela.

Para dar um exemplo, vamos trocá-lo e usar nosso delegado anônimo para filtrar uma lista com apenas os números pares e, em seguida, imprimi-los no console.

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

public class Program
{
    public static void Main(string[] args)
    {
        List<int> list = new List<int>();

        for (int i = 1; i <= 100; i++)
        {
            list.Add(i);
        }

        List<int> result = list.FindAll(
          delegate (int no)
          {
              return (no % 2 == 0);
          }
        );

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

Como você pode ver, o corpo do delegado é apenas um conjunto de expressões, como aconteceria com qualquer outro delegado. Mas em vez de ele ser uma definição separada, nós o introduzimos *ad hoc* em nossa chamada ao método List<T>.FindAll.

No entanto, mesmo com essa abordagem, ainda há muito código que podemos descartar. É aí que as **expressões lambda** entram em cena.

As expressões lambda ou apenas "lambdas" para abreviar, foram introduzidas pela primeira vez no C# 3.0 como um dos principais elementos de construção da LINQ (Consulta integrada à linguagem). Elas são apenas uma sintaxe mais conveniente para usar delegados. Elas declaram uma assinatura e um corpo de método, mas não têm uma identidade formal própria, a menos que sejam atribuídas a um delegado. Ao contrário dos representantes, elas podem ser atribuídas diretamente como o lado esquerdo do registro de eventos ou em várias cláusulas e métodos LINQ.

Como uma expressão lambda é apenas outra maneira de especificar um delegado, nós podemos reescrever o exemplo acima para usar uma expressão lambda em vez de um delegado anônimo.

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

public class Program
{
    public static void Main(string[] args)
    {
        List<int> list = new List<int>();

        for (int i = 1; i <= 100; i++)
        {
            list.Add(i);
        }

        List<int> result = list.FindAll(i => i % 2 == 0);

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

No exemplo anterior, a expressão lambda usada é `i => i % 2 == 0`. Mais uma vez, trata-se apenas de uma sintaxe **muito** conveniente para usar delegados, de modo que o que acontece nos bastidores é semelhante ao que acontece com o delegado anônimo.

Novamente, as lambdas são apenas delegados, o que significa que podem ser usadas como um manipulador de eventos sem problemas, como o snippet de código a seguir ilustra.

```
public MainWindow()
{
    InitializeComponent();

    Loaded += (o, e) =>
    {
        this.Title = "Loaded";
    };
}
```

O operador `+=` nesse contexto é usado para assinar um evento. Para obter mais informações, confira Como: realizar e cancelar a assinatura de eventos.

Recursos e leituras adicionais

- Delegados
- Funções Anônimas
- Expressões lambda

LINQ (Consulta Integrada à Linguagem)

23/10/2019 • 12 minutes to read • Edit Online

O que é?

A LINQ fornece funcionalidades de consulta no nível da linguagem e uma API de função de ordem superior para C# e VB como uma maneira de escrever um código expressivo e declarativo.

Sintaxe de consulta de nível de linguagem:

```
var linqExperts = from p in programmers
                  where p.IsNewToLINQ
                  select new LINQExpert(p);
```

```
Dim linqExperts = From p in programmers
                  Where p.IsNewToLINQ
                  Select New LINQExpert(p)
```

Mesmo exemplo usando a API `IEnumerable<T>`:

```
var linqExperts = programmers.Where(p => p.IsNewToLINQ)
                             .Select(p => new LINQExpert(p));
```

```
Dim linqExperts = programmers.Where(Function(p) p.IsNewToLINQ).
                             Select(Function(p) New LINQExpert(p))
```

A LINQ é expressiva

Imagine que você tem uma lista de animais de estimação, mas deseja convertê-la em um dicionário em que você pode acessar um animal de estimação diretamente por seu valor `RFID`.

Código imperativo tradicional:

```
var petLookup = new Dictionary<int, Pet>();

foreach (var pet in pets)
{
    petLookup.Add(pet.RFID, pet);
}
```

```
Dim petLookup = New Dictionary(Of Integer, Pet)()

For Each pet in pets
    petLookup.Add(pet.RFID, pet)
Next
```

A intenção por trás do código não é criar um novo `Dictionary<int, Pet>` e adicionar a ele por meio de um loop, é converter uma lista existente em um dicionário. A LINQ preserva a intenção enquanto o código imperativo não.

Expressão LINQ equivalente:

```
var petLookup = pets.ToDictionary(pet => pet.RFID);
```

```
Dim petLookup = pets.ToDictionary(Function(pet) pet.RFID)
```

O código usando a LINQ é importante porque ele nivela a área de atuação entre a intenção e o código ao raciocinar como um programador. Outro bônus é a brevidade do código. Imagine reduzir grandes partes de uma base de código em 1/3 como feito acima. Excelente, certo?

Provedores LINQ simplificam o acesso a dados

Para uma parte significativa do software em uso, tudo gira em torno de lidar com os dados de alguma origem (bancos de dados, JSON, XML etc.). Geralmente isso envolve aprender uma nova API para cada fonte de dados, o que pode ser inconveniente. A LINQ simplifica isso abstraindo os elementos comuns de acesso a dados em uma sintaxe de consulta que tem a mesma aparência, independentemente da fonte de dados escolhida.

Considere o seguinte: localizar todos os elementos XML com um valor de atributo específico.

```
public static IEnumerable<XElement> FindAllElementsWithAttribute(XElement documentRoot, string elementName,
                                        string attributeName, string value)
{
    return from el in documentRoot.Elements(elementName)
           where (string)el.Element(attributeName) == value
           select el;
}
```

```
Public Shared Function FindAllElementsWithAttribute(documentRoot As XElement, elementName As String,
                                        attributeName As String, value As String) As IEnumerable(Of
XElement)
    Return From el In documentRoot.Elements(elementName)
           Where el.Element(attributeName).ToString() = value
           Select el
End Function
```

Escrever o código para percorrer manualmente o documento XML para realizar essa tarefa seria muito mais desafiador.

Interagir com o XML não é a única coisa que você pode fazer com provedores LINQ. O LINQ to SQL é um ORM (Mapeador de Objeto Relacional) de funções bastante básicas para um banco de dados MSSQL. A biblioteca JSON.NET fornece uma passagem do documento JSON eficiente por meio da LINQ. Além disso, se não existir uma biblioteca que faça o que você precisa, também é possível escrever seu próprio provedor LINQ.

Por que usar a sintaxe de consulta?

Essa é uma pergunta que surge bastante. Afinal, isso

```
var filteredItems = myItems.Where(item => item.Foo);
```

```
Dim filteredItems = myItems.Where(Function(item) item.Foo)
```

é muito mais conciso que isso:

```
var filteredItems = from item in myItems
                    where item.Foo
                    select item;
```

```
Dim filteredItems = From item In myItems
                    Where item.Foo
                    Select item
```

A sintaxe da API não é apenas uma maneira mais concisa de fazer a sintaxe de consulta?

Nº A sintaxe de consulta permite o uso da cláusula **let**, que permite que você introduza e associe uma variável no escopo da expressão, usando-a em partes subsequentes da expressão. É possível reproduzir o mesmo código com apenas a sintaxe da API, mas mais provavelmente levará a um código que é difícil de ler.

Portanto, isso levanta a questão, **você deve usar apenas a sintaxe de consulta?**

A resposta para essa pergunta será **sim** se...

- Sua base de código já usar a sintaxe de consulta
- Você precisar definir o escopo de variáveis em suas consultas devido à complexidade
- Você preferir a sintaxe de consulta e ela não for distraí-lo da base de código

A resposta para essa pergunta será **não** se...

- Sua base de código já usar a sintaxe de API
- Você não precisar definir o escopo de variáveis em suas consultas
- Você preferir a sintaxe de API e ela não for distraí-lo da base de código

Exemplos essenciais

Para obter uma lista realmente abrangente de amostras de LINQ, visite 101 LINQ Samples (101 exemplos da LINQ).

A seguir está uma rápida demonstração de algumas das partes essenciais da LINQ. De nenhuma forma isso é abrangente, uma vez que a LINQ fornece significativamente mais funcionalidades do que o que é apresentado aqui.

- O básico: `Where`, `Select` e `Aggregate`:

```
// Filtering a list.
var germanShepards = dogs.Where(dog => dog.Breed == DogBreed.GermanShepard);

// Using the query syntax.
var queryGermanShepards = from dog in dogs
                          where dog.Breed == DogBreed.GermanShepard
                          select dog;

// Mapping a list from type A to type B.
var cats = dogs.Select(dog => dog.TurnIntoACat());

// Using the query syntax.
var queryCats = from dog in dogs
                select dog.TurnIntoACat();

// Summing the lengths of a set of strings.
int seed = 0;
int sumOfStrings = strings.Aggregate(seed, (s1, s2) => s1.Length + s2.Length);
```

```
' Filtering a list.
Dim germanShepards = dogs.Where(Function(dog) dog.Breed = DogBreed.GermanShepard)

' Using the query syntax.
Dim queryGermanShepards = From dog In dogs
                          Where dog.Breed = DogBreed.GermanShepard
                          Select dog

' Mapping a list from type A to type B.
Dim cats = dogs.Select(Function(dog) dog.TurnIntoACat())

' Using the query syntax.
Dim queryCats = From dog In dogs
                Select dog.TurnIntoACat()

' Summing the lengths of a set of strings.
Dim seed As Integer = 0
Dim sumOfStrings As Integer = strings.Aggregate(seed, Function(s1, s2) s1.Length + s2.Length)
```

- Nivelar uma lista de listas:

```
// Transforms the list of kennels into a list of all their dogs.
var allDogsFromKennels = kennels.SelectMany(kennel => kennel.Dogs);
```

```
' Transforms the list of kennels into a list of all their dogs.
Dim allDogsFromKennels = kennels.SelectMany(Function(kennel) kennel.Dogs)
```

- União entre dois conjuntos (com o comparador personalizado):

```
public class DogHairLengthComparer : IEqualityComparer<Dog>
{
    public bool Equals(Dog a, Dog b)
    {
        if (a == null && b == null)
        {
            return true;
        }
        else if ((a == null && b != null) ||
                 (a != null && b == null))
        {
            return false;
        }
        else
        {
            return a.HairLengthType == b.HairLengthType;
        }
    }

    public int GetHashCode(Dog d)
    {
        // Default hashcode is enough here, as these are simple objects.
        return d.GetHashCode();
    }
}

...

// Gets all the short-haired dogs between two different kennels.
var allShortHairedDogs = kennel1.Dogs.Union(kennel2.Dogs, new DogHairLengthComparer());
```

```
Public Class DogHairLengthComparer
  Inherits IEqualityComparer(Of Dog)

  Public Function Equals(a As Dog,b As Dog) As Boolean
      If a Is Nothing AndAlso b Is Nothing Then
          Return True
      ElseIf (a Is Nothing AndAlso b IsNot Nothing) OrElse (a IsNot Nothing AndAlso b Is Nothing) Then
          Return False
      Else
          Return a.HairLengthType = b.HairLengthType
      End If
  End Function

  Public Function GetHashCode(d As Dog) As Integer
      ' Default hashcode is enough here, as these are simple objects.
      Return d.GetHashCode()
  End Function
End Class

...

' Gets all the short-haired dogs between two different kennels.
Dim allShortHairedDogs = kennel1.Dogs.Union(kennel2.Dogs, New DogHairLengthComparer())
```

- Interseção entre dois conjuntos:

```
// Gets the volunteers who spend share time with two humane societies.
var volunteers = humaneSociety1.Volunteers.Intersect(humaneSociety2.Volunteers,
                                                     new VolunteerTimeComparer());
```

```
' Gets the volunteers who spend share time with two humane societies.
Dim volunteers = humaneSociety1.Volunteers.Intersect(humaneSociety2.Volunteers,
                                                     New VolunteerTimeComparer())
```

- Ordenar:

```
// Get driving directions, ordering by if it's toll-free before estimated driving time.
var results = DirectionsProcessor.GetDirections(start, end)
              .OrderBy(direction => direction.HasNoTolls)
              .ThenBy(direction => direction.EstimatedTime);
```

```
' Get driving directions, ordering by if it's toll-free before estimated driving time.
Dim results = DirectionsProcessor.GetDirections(start, end).
                OrderBy(Function(direction) direction.HasNoTolls).
                ThenBy(Function(direction) direction.EstimatedTime)
```

- Por fim, um exemplo mais avançado: determinar se os valores das propriedades de duas instâncias do mesmo tipo são iguais (emprestados e modificados dessa postagem do StackOverflow):

```
public static bool PublicInstancePropertiesEqual<T>(this T self, T to, params string[] ignore) where T : class
{
    if (self == null || to == null)
    {
        return self == to;
    }

    // Selects the properties which have unequal values into a sequence of those properties.
    var unequalProperties = from property in typeof(T).GetProperties(BindingFlags.Public |
BindingFlags.Instance)
                            where !ignore.Contains(property.Name)
                            let selfValue = property.GetValue(self, null)
                            let toValue = property.GetValue(to, null)
                            where !Equals(selfValue, toValue)
                            select property;
    return !unequalProperties.Any();
}
```

```
<System.Runtime.CompilerServices.Extension()>
Public Function PublicInstancePropertiesEqual(Of T As Class)(self As T, [to] As T, ParamArray ignore As
String()) As Boolean
    If self Is Nothing OrElse [to] Is Nothing Then
        Return self Is [to]
    End If

    ' Selects the properties which have unequal values into a sequence of those properties.
    Dim unequalProperties = From [property] In GetType(T).GetProperties(BindingFlags.Public Or
BindingFlags.Instance)
                            Where Not ignore.Contains([property].Name)
                            Let selfValue = [property].GetValue(self, Nothing)
                            Let toValue = [property].GetValue([to], Nothing)
                            Where Not Equals(selfValue, toValue) Select [property]
    Return Not unequalProperties.Any()
End Function
```

PLINQ

PLINQ, ou LINQ Paralela, é um mecanismo de execução paralelo para expressões de LINQ. Em outras palavras, expressões regulares de LINQ podem ser paralelizadas trivialmente em qualquer número de threads. Isso é feito por meio de uma chamada para `AsParallel()` precedendo a expressão.

Considere o seguinte:

```
public static string GetAllFacebookUserLikesMessage(IEnumerable<FacebookUser> facebookUsers)
{
    var seed = default(UInt64);

    Func<UInt64, UInt64, UInt64> threadAccumulator = (t1, t2) => t1 + t2;
    Func<UInt64, UInt64, UInt64> threadResultAccumulator = (t1, t2) => t1 + t2;
    Func<Uint64, string> resultSelector = total => $"Facebook has {total} likes!";

    return facebookUsers.AsParallel()
                        .Aggregate(seed, threadAccumulator, threadResultAccumulator, resultSelector);
}
```

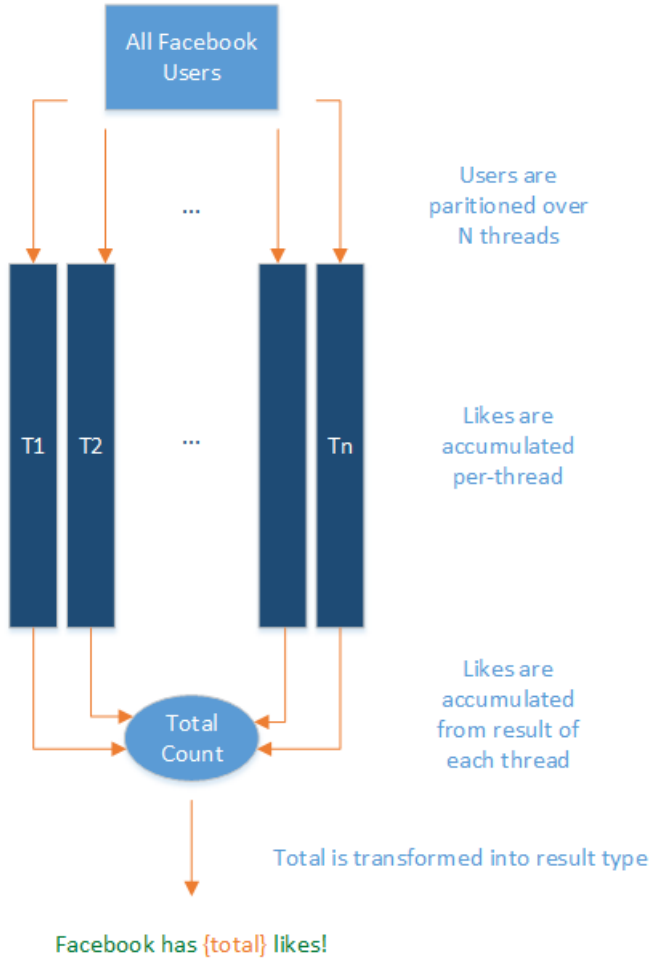
```
Public Shared GetAllFacebookUserLikesMessage(facebookUsers As IEnumerable(Of FacebookUser)) As String
{
    Dim seed As UInt64 = 0

    Dim threadAccumulator As Func(Of UInt64, UInt64, UInt64) = Function(t1, t2) t1 + t2
    Dim threadResultAccumulator As Func(Of UInt64, UInt64, UInt64) = Function(t1, t2) t1 + t2
    Dim resultSelector As Func(Of Uint64, string) = Function(total) $"Facebook has {total} likes!"

    Return facebookUsers.AsParallel().
                        Aggregate(seed, threadAccumulator, threadResultAccumulator, resultSelector)
}
```

Esse código particionará `facebookUsers` entre threads do sistema conforme necessário, somará o total de semelhantes em cada thread em paralelo, somará os resultados calculados por cada thread e projetará esse resultado em uma bela cadeia de caracteres.

Na forma de diagrama:



Trabalhos vinculados à CPU paralelizáveis que podem ser facilmente expressos por meio da LINQ (em outras palavras, que são funções puras e não têm efeitos colaterais) são ótimos candidatos para PLINQ. Para trabalhos que *têm* um efeito colateral, considere usar a Biblioteca de paralelismo de tarefas.

Recursos adicionais:

- 101 exemplos do LINQ
- LINQPad, um mecanismo de consulta de banco de dados e ambiente de playground para C#/F#/VB
- EduLinq, um livro eletrônico para aprender como o LINQ to Objects é implementado

Common Type System e Common Language Specification

23/10/2019 • 6 minutes to read • Edit Online

Novamente, dois termos que são usados livremente no mundo do .NET, eles realmente são essenciais para compreender como uma implementação do .NET possibilita o desenvolvimento para várias linguagens e para entender como ela funciona.

Common Type System

Para começar do zero, lembre-se que uma implementação do .NET é *independente de linguagem*. Isso não significa apenas que um programador pode escrever seu código em qualquer idioma que pode ser compilada para IL. Isso também significa que ela precisa ser capaz de interagir com o código escrito em outras linguagens que podem ser usadas em uma implementação do .NET.

Para fazer isso de forma transparente, deve haver uma maneira comum de descrever todos os tipos com suporte. Isso é o que o CTS (Common Type System) é responsável por fazer. Ele foi feito de várias maneiras:

- Estabelecer uma estrutura para a execução em qualquer idioma.
- Fornecer um modelo orientado a objetos para dar suporte à implementação de várias linguagens em uma implementação do .NET.
- Definir um conjunto de regras que todas as linguagens devem seguir quando se trata de trabalhar com tipos.
- Fornecer uma biblioteca que contém os tipos primitivos básicos que são usados no desenvolvimento de aplicativos (por exemplo, `Boolean`, `Byte`, `Char` etc.)

O CTS define dois tipos principais que devem ter suporte: tipos de referência e valor. Seus nomes apontam para suas definições.

Os objetos de tipos de referência são representados por uma referência ao valor real do objeto. Uma referência aqui é similar a um ponteiro no C/C++. Ele simplesmente se refere a um local da memória no qual estão os valores dos objetos. Isso tem um profundo impacto sobre como esses tipos são usados. Se você atribuir um tipo de referência a uma variável e, em seguida, passar essa variável em um método, por exemplo, qualquer alteração feita no objeto será refletida no objeto principal. Não há nenhuma cópia.

Tipos de valor são o oposto, no qual os objetos são representados por seus valores. Se você atribuir um tipo de valor a uma variável, você estará, essencialmente, copiando um valor do objeto.

O CTS define várias categorias de tipos, cada um com sua semântica específica e o uso:

- Classes
- Estruturas
- Enums
- Interfaces
- Delegados

O CTS também define todas as outras propriedades de tipos, como modificadores de acesso, o que são membros de tipo válidos, como a herança e a sobrecarga funcionam e assim por diante. Infelizmente, se aprofundar em qualquer um deles está além do escopo de um artigo introdutório como este, mas você pode consultar a seção Mais recursos no final para obter links para mais conteúdos detalhados que abordam esses tópicos.

Common Language Specification

Para habilitar cenários de interoperabilidade completa, todos os objetos que são criados em código devem se basear em alguns pontos em comum nos idiomas que estão consumindo eles (são seus *chamadores*). Como há várias linguagens diferentes, o .NET especificou essas semelhanças em algo chamado de CLS **(Common Language Specification)**. A CLS define um conjunto de recursos que são necessários para muitos aplicativos comuns. Ela também fornece uma espécie de receita para qualquer linguagem que é implementada no .NET, no que ela precisar dar suporte.

A CLS é um subconjunto do CTS. Isso significa que todas as regras no CTS também se aplicam à CLS, a menos que as regras da CLS sejam mais estritas. Se um componente é criado usando apenas as regras na CLS, ou seja, ele expõe apenas os recursos da CLS em sua API, ele é chamado de **compatível com CLS**. Por exemplo, as `<framework-librares>` estão em conformidade com CLS precisamente porque precisam funcionar em todas as linguagens com suporte no .NET.

Você pode consultar os documentos na seção Mais recursos abaixo para obter uma visão geral de todos os recursos na CLS.

Mais recursos

- Common Type System
- Common Language Specification

Processamento paralelo, simultaneidade e programação assíncrona no .NET

31/10/2019 • 2 minutes to read • Edit Online

O .NET fornece várias maneiras de escrever código assíncrono para tornar seu aplicativo mais ágil para o usuário e a escrever código paralelo que usa vários threads de execução para maximizar o desempenho do computador do usuário.

Nesta seção

Programação Assíncrona

Descreve mecanismos para programação assíncrona fornecida pelo .NET.

Programação paralela

Descreve um modelo de programação baseado em tarefa que simplifica o desenvolvimento paralelo, permitindo escrever código paralelo refinado, eficiente e escalonável em uma linguagem natural sem a necessidade de trabalhar diretamente com threads ou o pool de threads.

Threading

Descreve os mecanismos básicos de sincronização e simultaneidade fornecidos pelo .NET.

Visão geral da assincronia

23/10/2019 • 3 minutes to read • Edit Online

Não muito tempo atrás, os aplicativos ficavam mais rápidos simplesmente comprando um computador ou servidor mais novo e então essa tendência parou. Na verdade, ela foi invertida. Surgiram telefones celulares com chips ARM de núcleo único de 1 GHz e as cargas de trabalho do servidor passaram para VMs. Os usuários ainda querem uma interface do usuário responsiva e os proprietários de negócios querem servidores que ajustem a escala com seus negócios. A transição para celular e nuvem e uma população conectada à Internet de >3B usuários resultaram em um novo conjunto de padrões de software.

- Os aplicativos cliente devem estar sempre ativos, sempre conectados e constantemente responsivos à interação do usuário (por exemplo, toque) com classificações altas na loja de aplicativos!
- Os serviços devem lidar com picos de tráfego escalando e reduzindo horizontalmente sem problemas.

A programação assíncrona é uma técnica chave que torna fácil de lidar com o bloqueio de E/S e operações simultâneas em vários núcleos. O .NET fornece a capacidade para os aplicativos e serviços serem responsivos e elásticos com modelos de programação assíncrona fáceis de usar e em nível de linguagem em C#, VB e F#.

Por que escrever código assíncrono?

Os aplicativos modernos fazem amplo uso de E/S de arquivos e rede. As APIs de E/S tradicionalmente bloqueiam por padrão, resultando em experiências de usuário e utilização de hardware ruins, a menos que você deseje aprender e usar padrões desafiadores. O modelo de programação assíncrona em nível de linguagem e as APIs assíncronas baseadas em tarefa invertem esse modelo, tornando a execução assíncrona o padrão com poucos novos conceitos para aprender.

O código assíncrono tem as seguintes características:

- Lida com mais solicitações de servidor gerando threads para lidar com mais solicitações enquanto espera as solicitações de E/S retornarem.
- Permite que as interfaces do usuário sejam mais responsivas gerando threads para a interação da interface do usuário enquanto espera as solicitações de E/S e fazendo a transição do trabalho de longa execução para outros núcleos de CPU.
- Muitas das APIs do .NET mais novas são assíncronas.
- É fácil escrever código assíncrono no .NET!

O que vem a seguir?

Para obter mais informações, consulte os comentários no tópico Assincronia detalhada.

O tópico Padrões de programação assíncrona fornece uma visão geral dos três padrões de programação assíncronos com suporte no .NET Framework:

- APM (Modelo assíncrono de programação) (herdado)
- EAP (Padrão assíncrono baseado em evento) (herdado)
- TAP (Padrão assíncrono baseado em tarefa) (recomendado para novos desenvolvimentos)

Para obter mais informações sobre o modelo de programação baseado em tarefa recomendado, consulte o tópico Programação assíncrona baseada em tarefas.

Assincronia detalhada

31/10/2019 • 19 minutes to read • Edit Online

Escrever código assíncrono vinculado à CPU ou à E/S é simples usando o modelo assíncrono baseado em Tarefas do .NET. O modelo é exposto pelos tipos `Task` e `Task<T>` e pelas palavras-chave `async` e `await` no C# e no Visual Basic. (Recursos específicos a um idioma podem ser encontrados na seção Consulte também). Este artigo explica como usar a assincronia do .NET e fornece informações sobre a estrutura de assincronia usada nos bastidores.

Task e Task<T>

Tarefas são constructos usados para implementar o que é conhecido como o modelo de promessa de simultaneidade. Em resumo, elas oferecem a você uma "promessa" de que o trabalho será concluído em um momento posterior, permitindo que você coordene a promessa com uma API limpa.

- `Task` representa uma única operação que não retorna um valor.
- `Task<T>` representa uma única operação que retorna um valor do tipo `T`.

É importante pensar nas tarefas como abstrações do trabalho ocorrendo de maneira assíncrona e *não* uma abstração de threading. Por padrão, as tarefas são executadas no thread atual e delegam o trabalho para o sistema operacional, conforme apropriado. Opcionalmente, as tarefas podem ser solicitadas explicitamente para serem executadas em um thread separado por meio da API `Task.Run`.

As tarefas expõem um protocolo de API para monitoramento, aguardando e acessando o valor do resultado (no caso de `Task<T>`) de uma tarefa. A integração de linguagem, com a palavra-chave `await`, fornece uma abstração de nível mais alto para o uso de tarefas.

O uso de `await` permite que seu aplicativo ou serviço realize um trabalho útil enquanto uma tarefa estiver em execução gerando o controle para seu chamador até que a tarefa seja concluída. Seu código não precisa contar com retornos de chamada ou eventos para continuar a execução após a tarefa ter sido concluída. A integração da API da tarefa e da linguagem faz isso para você. Se você estiver usando `Task<T>`, a palavra-chave `await` "desencapsulará" adicionalmente o valor retornado quando a Tarefa for concluída. Os detalhes de como isso funciona são explicados mais abaixo.

Você pode aprender mais sobre as tarefas e as diferentes maneiras de interagir com elas no tópico Padrão assíncrono baseado em tarefa (TAP).

Aprofundamento em tarefas para uma operação vinculada à E/S

A seção a seguir descreve uma exibição de 10.000 pés do que acontece com uma chamada de E/S assíncrona típica. Vamos começar com alguns exemplos.

O primeiro exemplo chama um método assíncrono e retorna uma tarefa ativa, provavelmente ainda para ser concluída.

```
public Task<string> GetHtmlAsync()
{
    // Execution is synchronous here
    var client = new HttpClient();

    return client.GetStringAsync("https://www.dotnetfoundation.org");
}
```

O segundo exemplo adiciona o uso das palavras-chave `async` e `await` para operar na tarefa.

```
public async Task<string> GetFirstCharactersCountAsync(string url, int count)
{
    // Execution is synchronous here
    var client = new HttpClient();

    // Execution of GetFirstCharactersCountAsync() is yielded to the caller here
    // GetStringAsync returns a Task<string>, which is *awaited*
    var page = await client.GetStringAsync("https://www.dotnetfoundation.org");

    // Execution resumes when the client.GetStringAsync task completes,
    // becoming synchronous again.

    if (count > page.Length)
    {
        return page;
    }
    else
    {
        return page.Substring(0, count);
    }
}
```

A chamada para `GetStringAsync()` realiza a chamada por bibliotecas .NET de níveis inferiores (talvez chamando outros métodos assíncronos) até atingir uma chamada de interoperabilidade P/Invoke em uma biblioteca de rede nativa. A biblioteca nativa pode chamar subsequentemente uma chamada à API do sistema (como `write()` para um soquete no Linux). Um objeto de tarefa será criado no limite nativo/gerenciado, possivelmente usando TaskCompletionSource. O objeto de tarefa será passado pelas camadas, possivelmente operado ou retornado diretamente, finalmente retornado ao chamador inicial.

No segundo exemplo acima, um objeto `Task<T>` será retornado de `GetStringAsync`. O uso da palavra-chave `await` faz com que o método retorne um objeto de tarefa recém-criado. O controle retorna para o chamador desse local no método `GetFirstCharactersCountAsync`. Os métodos e propriedades do objeto Task<T> permitem que os chamadores monitorem o progresso da tarefa, que será concluída quando o código restante em GetFirstCharactersCountAsync for executado.

Após a chamada à API do sistema, a solicitação está no espaço de kernel, trilhando seu caminho para o subsistema de rede do sistema operacional (como `/net` no Kernel do Linux). Aqui, o sistema operacional tratará a solicitação de rede *assincronamente*. Os detalhes podem ser diferentes dependendo do sistema operacional usado (a chamada de driver de dispositivo pode ser agendada como um sinal enviado de volta para o tempo de execução ou pode ser feita uma chamada de driver de dispositivo e *em seguida,* um sinal enviado de volta), mas, no fim, o tempo de execução será informado de que a solicitação de rede está em andamento. Neste momento, o trabalho para o driver de dispositivo estará agendado, em andamento ou já finalizado (a solicitação já está "durante a transmissão"), mas como tudo isso está ocorrendo assincronamente, o driver do dispositivo pode manipular outra coisa imediatamente.

Por exemplo, no Windows, um thread de sistema operacional faz uma chamada para o driver de dispositivo de rede e solicita que ele realize a operação de rede por meio de um IRP (Pacote de Solicitação de Interrupção), que representa a operação. O driver de dispositivo recebe o IRP, faz a chamada para a rede, marca o IRP como

"pendente" e retorna para o sistema operacional. Como o thread do sistema operacional agora sabe que o IRP está "pendente", ele não tem mais nenhum trabalho para fazer para esse trabalho e "retorna" para que possa ser usado para realizar outro trabalho.

Quando a solicitação é atendida e os dados voltam por meio do driver de dispositivo, ele notifica a CPU sobre os novos dados recebidos por meio de uma interrupção. Como essa interrupção é tratada variará dependendo do sistema operacional, mas, no fim, os dados serão passados pelo sistema operacional até atingirem uma chamada de interoperabilidade do sistema (por exemplo, no Linux um manipulador de interrupção agendará a metade inferior da IRQ para passar os dados pelo sistema operacional assincronamente). Observe que isso *também* ocorre assincronamente. O resultado é colocado na fila até que o próximo thread disponível possa executar o método assíncrono e "desencapsular" o resultado da tarefa concluída.

Durante todo esse processo, uma consideração importante é que **nenhum thread é dedicado a executar a tarefa**. Embora o trabalho seja executado em algum contexto (ou seja, o sistema operacional tem de passar os dados para um driver de dispositivo e responder a uma interrupção), nenhum thread é dedicado a *esperar* os dados da solicitação voltarem. Isso permite que o sistema lide com um volume muito maior de trabalho em vez de esperar que alguma chamada de E/S seja concluída.

Embora o descrito acima possa parecer muito trabalho a ser feito, quando medido em termos de tempo total, ele é minúsculo em comparação com o tempo necessário para realizar o trabalho real de E/S. Embora não seja precisa, uma linha do tempo possível para essa chamada teria esta aparência:

0-1
——2-
3

- O tempo gasto dos pontos `0` até `1` é tudo até um método assíncrono gerar o controle para seu chamador.
- O tempo gasto dos pontos `1` até `2` é o tempo gasto na E/S, sem custo de CPU.
- Por fim, o tempo gasto dos pontos `2` até `3` está passando o controle de volta (e potencialmente um valor) para o método assíncrono, ponto no qual está sendo executado novamente.

O que isso significa para um cenário de servidor?

Esse modelo funciona bem com uma carga de trabalho de cenário de servidor típica. Como não há nenhum thread dedicado para bloquear tarefas não concluídas, o pool de threads do servidor pode atender a um volume muito maior de solicitações da Web.

Considere dois servidores: um que executa o código assíncrono e que não faz isso. Para esse exemplo, cada servidor tem apenas cinco threads disponíveis para solicitações de serviço. Observe que esses números são imaginariamente pequenos e servem apenas em um contexto de demonstração.

Suponha que ambos os servidores recebem seis solicitações simultâneas. Cada solicitação executa uma operação de E/S. O servidor *sem* o código assíncrono precisa enfileirar a sexta solicitação até que um dos cinco threads tenham concluído o trabalho vinculado à E/S e escrito uma resposta. No ponto em que a 20ª solicitação chega, o servidor pode começar a ficar lento, pois a fila está ficando muito longa.

O servidor *com* o código assíncrono em execução ainda enfileira a sexta solicitação, mas como ele usa `async` e `await`, cada um de seus threads é liberado quando o trabalho vinculado à E/S começa, em vez de quando ele é concluído. No momento em que a 20ª solicitação chega, a fila para as solicitações recebidas é muito menor (se ela tiver algo em absoluto) e o servidor não fica lento.

Embora esse seja um exemplo inventado, ele funciona de maneira muito semelhante no mundo real. Na verdade, você pode esperar que um servidor seja capaz de lidar com uma ordem de grandeza de mais solicitações usando `async` e `await` do que se ele estivesse dedicando um thread para cada solicitação recebida.

O que isso significa para um cenário de cliente?

O maior ganho do uso do `async` e `await` para um aplicativo cliente é um aumento na capacidade de resposta.

Embora você possa fazer um aplicativo responsivo gerando threads manualmente, o ato de gerar um thread é uma operação cara em relação a apenas usar o `async` e `await`. Especialmente para algo como um jogo para dispositivos móveis, afetar o thread da interface do usuário o mínimo possível no que concerne à E/S é crucial.

Mais importante, como o trabalho vinculado à E/S passa praticamente nenhum tempo na CPU, dedicar um thread de CPU inteiro para realizar quase nenhum trabalho útil seria um uso inadequado dos recursos.

Além disso, distribuir o trabalho para o thread de interface do usuário (como atualizar uma interface do usuário) é muito simples com os métodos `async` e não requer trabalho extra (como chamar um delegado thread-safe).

Aprofundamento em Task e Task<T> para uma operação vinculada à CPU

O código `async` vinculado à CPU é um pouco diferente do código `async` vinculado à E/S. Como o trabalho é feito na CPU, não há como contornar a dedicação de um thread à computação. O uso de `async` e `await` fornece uma maneira simples de interagir com thread em segundo plano e manter o chamador do método assíncrono responsivo. Observe que isso não fornece nenhuma proteção para dados compartilhados. Se você estiver usando dados compartilhados, ainda precisará aplicar uma estratégia de sincronização apropriada.

Esta é uma exibição de 10.000 pés de uma chamada assíncrona vinculada à CPU:

```
public async Task<int> CalculateResult(InputData data)
{
    // This queues up the work on the threadpool.
    var expensiveResultTask = Task.Run(() => DoExpensiveCalculation(data));

    // Note that at this point, you can do some other work concurrently,
    // as CalculateResult() is still executing!

    // Execution of CalculateResult is yielded here!
    var result = await expensiveResultTask;

    return result;
}
```

`CalculateResult()` executa no thread no qual foi chamado. Quando ele chama `Task.Run`, coloca na fila uma operação cara vinculada à CPU, `DoExpensiveCalculation()`, no pool de threads e recebe um identificador `Task<int>`. `DoExpensiveCalculation()` é finalmente executado simultaneamente no próximo thread disponível, provavelmente em outro núcleo da CPU. É possível fazer o trabalho simultâneo enquanto `DoExpensiveCalculation()` está ocupado em outro thread, pois o thread que chamou `CalculateResult()` ainda está em execução.

Uma vez que `await` é encontrado, a execução de `CalculateResult()` é gerada para seu chamador, permitindo que outro trabalho seja realizado com o thread atual enquanto `DoExpensiveCalculation()` está produzindo um resultado. Ao terminar, o resultado é enfileirado para ser executado no thread principal. No fim, o thread principal retornará para a execução de `CalculateResult()`, ponto em que ele terá o resultado de `DoExpensiveCalculation()`.

Por que a assincronia ajuda aqui?

`async` e `await` são a prática recomendada para gerenciar o trabalho vinculado à CPU quando você precisar de capacidade de resposta. Existem vários padrões para usar a assincronia com o trabalho vinculado à CPU. É importante observar que há um pequeno custo para usar a assincronia e não é recomendado para loops estreitos. Cabe a você determinar como escrever seu código em torno dessa nova capacidade.

Consulte também

- Programação assíncrona em C#

- Programação assíncrona com async e await (C#)
- Programação assíncrona em F#
- Programação assíncrona com Async e Await (Visual Basic)

Padrões de programação assíncrona

31/10/2019 • 3 minutes to read • Edit Online

O .NET fornece três padrões para a execução de operações assíncronas:

- **TAP (padrão assíncrono baseado em tarefa)** , que usa um único método para representar o início e a conclusão de uma operação assíncrona. O TAP foi introduzido no .NET Framework 4. **É a abordagem recomendada para a programação assíncrona no .NET.** As palavras-chave async e await no C# e os operadores Async e Await no Visual Basic adicionam suporte à linguagem para TAP. Para saber mais, confira Padrão assíncrono baseado em tarefa (TAP).
- **Padrão assíncrono baseado em evento (EAP)** , que é o modelo herdado baseado em evento para fornecimento do comportamento assíncrono. Ele requer um método que tem o sufixo `Async` e um ou mais eventos, tipos delegados de manipulador de eventos e tipos derivados de `EventArg`. O EAP foi introduzido no .NET Framework 2.0. Não é mais recomendado para novo desenvolvimento. Para saber mais, confira EAP (Padrão Assíncrono baseado em Evento).
- Padrão de **Modelo de programação assíncrona (APM)** (também chamado de padrão IAsyncResult), que é o modelo herdado que usa a interface IAsyncResult para fornecimento do comportamento assíncrono. Nesse padrão, as operações síncronas exigem os métodos `Begin` e `End` (por exemplo, `BeginWrite` e `EndWrite` para implementar uma operação de gravação assíncrona). Esse padrão não é mais recomendado para implantação nova. Para saber mais, veja APM (Modelo Assíncrono de Programação).

Comparação de padrões

Para obter uma comparação rápida de como os três padrões modelam as operações assíncronas, considere um método `Read` que lê uma quantidade especificada de dados em um buffer fornecido começando em um deslocamento especificado:

```
public class MyClass
{
    public int Read(byte [] buffer, int offset, int count);
}
```

O equivalente do TAP para este método poderia expor o método único `ReadAsync` a seguir:

```
public class MyClass
{
    public Task<int> ReadAsync(byte [] buffer, int offset, int count);
}
```

O equivalente do EAP poderia expor o seguinte conjunto de tipos e membros:

```
public class MyClass
{
    public void ReadAsync(byte [] buffer, int offset, int count);
    public event ReadCompletedEventHandler ReadCompleted;
}
```

O equivalente do APM poderia expor os métodos `BeginRead` e `EndRead`:

```
public class MyClass
{
    public IAsyncResult BeginRead(
        byte [] buffer, int offset, int count,
        AsyncCallback callback, object state);
    public int EndRead(IAsyncResult asyncResult);
}
```

Consulte também

- Assincronia detalhada
- Programação assíncrona em C#
- Programação assíncrona em F#
- Programação assíncrona com Async e Await (Visual Basic)

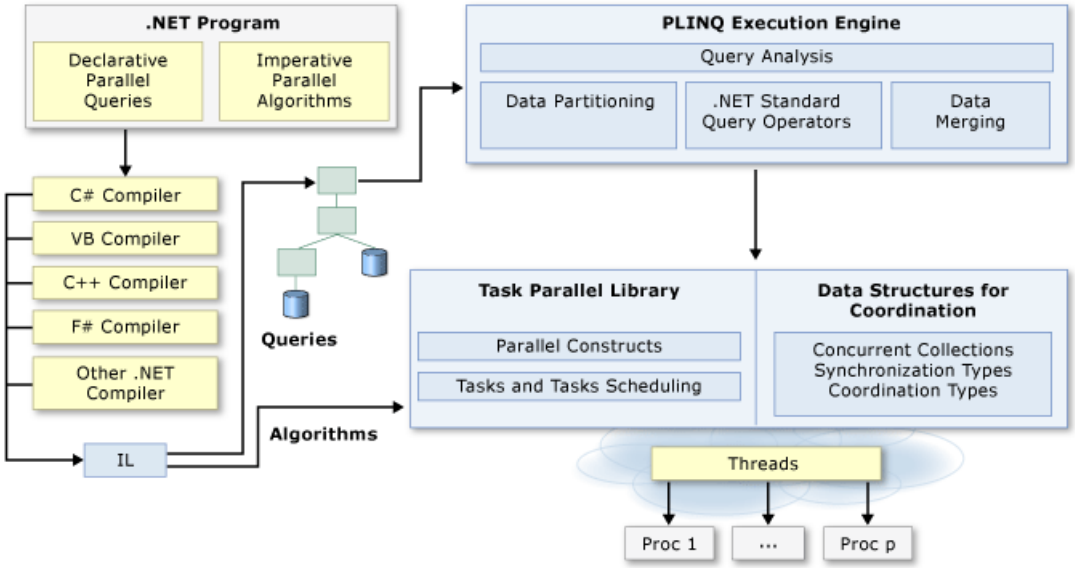
Parallel Programming in .NET (Programação paralela no .NET)

31/10/2019 • 3 minutes to read • Edit Online

Muitos computadores pessoais e estações de trabalho têm vários núcleos de CPU, o que permite a execução simultânea de vários threads. Para tirar proveito do hardware, é possível paralelizar seu código para distribuir o trabalho entre vários processadores.

No passado, a paralelização exigia a manipulação em baixo nível de threads e de bloqueios. O Visual Studio e o .NET Framework aprimoram o suporte à programação paralela ao fornecer um runtime, tipos de biblioteca de classes e ferramentas de diagnóstico. Esses recursos, que foram introduzidos com o .NET Framework 4, simplificam o desenvolvimento paralelo. É possível escrever código paralelo eficiente, refinado e dimensionável em uma linguagem natural sem precisar trabalhar diretamente com threads ou o pool de threads.

A ilustração a seguir oferece uma visão geral de alto nível da arquitetura de programação paralela do .NET Framework:



Tópicos relacionados

TECNOLOGIA	DESCRIÇÃO
TPL (Biblioteca de Paralelismo de Tarefas)	Fornece documentação para a classe System.Threading.Tasks.Parallel, a qual inclui versões paralelas de `For` e loops `ForEach`, e também para a classe System.Threading.Tasks.Task, a qual representa a forma preferencial de expressar operações assíncronas.
PLINQ (LINQ paralelo)	Uma implementação paralela do LINQ em Objects que melhora significativamente o desempenho em muitos cenários.
Estruturas de dados para programação paralela	Fornece links para a documentação de classes de coleta com threads seguros, tipos de sincronização leves e tipos para inicialização lenta.

TECNOLOGIA	DESCRIÇÃO
Ferramentas de diagnóstico paralelo	Fornece links para a documentação de janelas do depurador para tarefas e pilhas paralelas, e para a Visualização Simultânea do Visual Studio.
Particionadores personalizados para PLINQ e TPL	Descreve como os particionadores funcionam e como configurar os particionadores padrão ou criar um novo particionador.
Agendadores de tarefas	Descreve como os agendadores funcionam e como os agendadores padrão podem ser configurados.
Expressões lambda em PLINQ e TPL	Fornece uma visão geral das expressões lambda em C# e Visual Basic e mostra como elas são usadas em PLINQ e na Biblioteca Paralela de Tarefas.
Para leitura adicional	Fornece links para informações adicionais e recursos de exemplo para a programação paralela no .NET.

Consulte também

- Visão Geral da Assincronia
- Threading gerenciado

Threading gerenciado

31/10/2019 • 3 minutes to read • Edit Online

Quer você esteja desenvolvendo para computadores com um processador, quer para com vários, é conveniente que seu aplicativo forneça uma interação mais dinâmica com o usuário, mesmo se o aplicativo, no momento, estiver realizando outro trabalho. Usar vários threads de execução é uma das maneiras mais eficazes de manter seu aplicativo responsivo ao usuário e, ao mesmo tempo, usar o processador entre ou, até mesmo, durante os eventos do usuário. Embora esta seção introduza os conceitos básicos de threading, ele se concentra nos conceitos de threading gerenciado e em como usá-lo.

> **NOTE**
>
> Começando no .NET Framework 4, a programação multi-threaded ficou bastante simplificada com as classes System.Threading.Tasks.Parallel e System.Threading.Tasks.Task, o PLINQ (LINQ paralelo), as novas classes de coleção simultânea no namespace System.Collections.Concurrent e um novo modelo de programação que se baseia no conceito de tarefas e não em threads. Para obter mais informações, consulte Programação paralela.

Nesta seção

Noções básicas de threading gerenciado
Fornece uma visão geral de threading gerenciado e descreve quando usar vários threads.

Usando threads e threading
Explica como criar, iniciar, pausar, retomar e abortar threads.

Práticas recomendadas de threading gerenciado
Aborda os níveis de sincronização, como evitar deadlocks e condições de corrida, além de outros problemas de threading.

Objetos e recursos de threading
Descreve as classes gerenciadas que você pode usar para sincronizar as atividades de threads e os dados de objetos acessados em threads diferentes, bem como fornece uma visão geral dos threads de pool do thread.

Referência

System.Threading
Contém classes para uso e sincronização de threads gerenciados.

System.Collections.Concurrent
Contém classes de coleção que são seguras para uso com vários threads.

System.Threading.Tasks
Contém classes para criação e agendamento de tarefas de processamento simultâneo.

Seções relacionadas

Domínios do aplicativo
Fornece uma visão geral de domínios do aplicativo e seu uso pelo Common Language Infrastructure.

E/S de arquivo assíncrona
Descreve as vantagens de desempenho e a operação básica da E/S assíncrona.

TAP (Padrão Assíncrono Baseado em Tarefa)

Fornece uma visão geral do padrão recomendado para programação assíncrona no .NET.

Chamando métodos síncronos de forma assíncrona

Explica como chamar métodos nos threads de pool do thread usando recursos internos de representantes.

Programação paralela

Descreve as bibliotecas de programação paralela, que simplificam o uso de vários threads em aplicativos.

PLINQ (LINQ paralelo)

Descreve um sistema para executar consultas paralelamente, de modo a aproveitar vários processadores.

Tipos relacionados a memória e extensão

31/10/2019 • 3 minutes to read • Edit Online

A partir do .NET Core 2.1, o .NET inclui uma variedade de tipos interrelacionados que representam uma região contígua, fortemente tipada de memória arbitrária. Elas incluem:

- System.Span<T>, um tipo usado para acessar uma região contígua da memória. Uma instância Span<T> pode ser sustentada por uma matriz do tipo `T`, uma String, um buffer alocado com stackalloc ou um ponteiro para memória não gerenciada. Como ela deve ser alocada na pilha, tem várias restrições. Por exemplo, um campo em uma classe não pode ser do tipo Span<T>, nem a extensão pode ser usada em operações assíncronas.
- System.ReadOnlySpan<T>, uma versão imutável da estrutura Span<T>.
- System.Memory<T>, uma região contígua da memória alocada no heap gerenciado, em vez da pilha. Uma instância Memory<T> pode ser sustentada por uma matriz do tipo `T` ou uma String. Como ela pode ser armazenada no heap gerenciado, Memory<T> não tem nenhuma das limitações de Span<T>.
- System.ReadOnlyMemory<T>, uma versão imutável da estrutura Memory<T>.
- System.Buffers.MemoryPool<T>, que aloca blocos de memória fortemente tipados de um pool de memória para um proprietário. Instâncias IMemoryOwner<T> podem ser alocadas do pool chamando MemoryPool<T>.Rent e lançadas de volta ao pool chamando MemoryPool<T>.Dispose().
- System.Buffers.IMemoryOwner<T>, que representa o proprietário de um bloco de memória e controla o gerenciamento do tempo de vida.
- MemoryManager<T>, uma classe base abstrata que pode ser usada para substituir a implementação de Memory<T>, de modo que Memory<T> possa ser sustentada por tipos adicionais, como identificadores seguros. MemoryManager<T> destina-se a cenários avançados.
- ArraySegment<T>, um wrapper para determinado número de elementos de matriz, começando em um índice específico.
- System.MemoryExtensions, uma coleção de métodos de extensão para converter cadeias de caracteres, matrizes e segmentos de matriz para blocos Memory<T>.

> **NOTE**
> Para estruturas anteriores, Span<T> e Memory<T> estão disponíveis no pacote do System.Memory NuGet.

Para obter mais informações, consulte o namespace de System.Buffers.

Como trabalhando com memória e extensão

Como os tipos relacionados a memória e extensão normalmente são usados para armazenar dados em um pipeline de processamento, é importante que os desenvolvedores sigam um conjunto de melhores práticas ao usar Span<T>, Memory<T> e tipos relacionados. Essas melhores práticas estão documentadas em Diretrizes de uso de Memória<T > e Extensão<T >.

Consulte também

- System.Memory<T>
- System.ReadOnlyMemory<T>
- System.Span<T>
- System.ReadOnlySpan<T>
- System.Buffers

Diretrizes de uso de Memory<T> e Span<T>

31/10/2019 • 26 minutes to read • Edit Online

O .NET Core inclui vários tipos que representam uma região contígua arbitrária de memória. O .NET Core 2.0 introduziu Span<T> e ReadOnlySpan<T>, que são buffers de memória leves que podem ter suporte de memória gerenciada ou não gerenciada. Como esses tipos só podem ser armazenados na pilha, eles são inadequados para vários cenários, incluindo chamadas de método assíncronas. O .NET Core 2.1 adiciona vários outros tipos, inclusive Memory<T>, ReadOnlyMemory<T>, IMemoryOwner<T> e MemoryPool<T>. Assim como Span<T>, Memory<T> e os tipos relacionados podem ter suporte de memória gerenciada e não gerenciada. Diferentemente de Span<T>, Memory<T> pode ser armazenado no heap gerenciado.

Tanto Span<T> como Memory<T> representam buffers de dados estruturados que podem ser usados em pipelines. Ou seja, eles são projetados para que alguns ou todos os dados possam ser passados com eficiência para os componentes no pipeline que pode processá-los e, opcionalmente, modificar o buffer. Como Memory<T> e os tipos relacionados podem ser acessados por vários componentes ou threads, é importante que os desenvolvedores sigam algumas diretrizes de uso padrão para criar um código robusto.

Gerenciamento de proprietários, consumidores e vida útil

Como os buffers podem ser passados entre APIs e, às vezes, ser acessados de vários threads, é importante levar em consideração o gerenciamento da vida útil. Os três principais conceitos são:

- **Propriedade**. O proprietário de uma instância de buffer é responsável pelo gerenciamento da vida útil, inclusive pela destruição do buffer quando ele não está mais em uso. Todos os buffers pertencem a um único proprietário. Geralmente, o proprietário é o componente que criou o buffer ou que recebeu o buffer de fábrica. É possível também transferir a propriedade. O **Componente-A** poderá ceder o controle do buffer ao **Componente-B**, passando o **Componente-A** a não poder mais usar o buffer. A partir daí, o **Componente-B** se tornará responsável por destruir o buffer quando ele não estiver mais em uso.
- **Consumo**. O consumidor de uma instância do buffer pode usar essa instância, lendo e possivelmente gravando nela. Os buffers podem ter um consumidor por vez, a menos que algum mecanismo de sincronização externo seja fornecido. O consumidor ativo de um buffer não é necessariamente o proprietário do buffer.
- **Concessão**. A concessão é o período de tempo em que um componente específico pode ser o consumidor do buffer.

O exemplo de pseudocódigo a seguir ilustra esses três conceitos. Ele inclui um método `Main` que cria uma instância para o buffer Memory<T> de tipo Char, chama o método `WriteInt32ToBuffer` para gravar a representação da cadeia de caracteres de um número inteiro no buffer e, em seguida, chama o método `DisplayBufferToConsole` para exibir o valor do buffer.

```
using System;

class Program
{
    // Write 'value' as a human-readable string to the output buffer.
    void WriteInt32ToBuffer(int value, Buffer buffer);

    // Display the contents of the buffer to the console.
    void DisplayBufferToConsole(Buffer buffer);

    // Application code
    static void Main()
    {
        var buffer = CreateBuffer();
        try
        {
            int value = Int32.Parse(Console.ReadLine());
            WriteInt32ToBuffer(value, buffer);
            DisplayBufferToConsole(buffer);
        }
        finally
        {
            buffer.Destroy();
        }
    }
}
```

O método `Main` cria o buffer (neste caso, uma instância de Span<T>), portanto, ele é o respectivo proprietário. Dessa forma, `Main` é responsável por destruir o buffer quando ele não está mais em uso. Ele faz isso chamando o método Span<T>.Clear() do buffer. O método Clear() realmente limpa a memória do buffer. A estrutura de Span<T> não tem um método que destrua o buffer.

O buffer tem dois consumidores: `WriteInt32ToBuffer` e `DisplayBufferToConsole`. Há apenas um consumidor por vez (primeiro `WriteInt32ToBuffer`, depois `DisplayBufferToConsole`), e nenhum dos consumidores é proprietário do buffer. Ainda, "consumidor" neste contexto não implica uma exibição somente leitura do buffer. Os consumidores podem modificar o conteúdo do buffer, assim como `WriteInt32ToBuffer` o faz, se for fornecida uma exibição de leitura/gravação do buffer.

O método `WriteInt32ToBuffer` tem uma concessão (pode consumir) o buffer entre o início da chamada do método e a hora em que o método é retornado. Da mesma maneira, `DisplayBufferToConsole` tem uma concessão no buffer enquanto ele está em execução, e a concessão é liberada quando o método é desenrolado. Não há APIs para gerenciamento de concessão. Uma "concessão" é uma questão conceitual.

Memory<T> e o modelo proprietário/consumidor

Conforme a seção Gerenciamento de proprietários, consumidores e vida útil descreve, um buffer tem sempre um proprietário. O .NET Core tem suporte para dois modelos de propriedade:

- Um modelo com suporte para propriedade única. Um buffer tem um único proprietário por toda a vida útil.
- Um modelo com suporte para transferência de propriedade. É possível transferir a propriedade de um buffer do proprietário original (o respectivo criador) para outro componente que se torna responsável pelo gerenciamento da vida útil do buffer. Esse proprietário pode, por sua vez, transferir a propriedade para outro componente, e assim por diante.

Use a interface System.Buffers.IMemoryOwner<T> para gerenciar explicitamente a propriedade de um buffer. IMemoryOwner<T> tem suporte para os dois modelos de propriedade. O componente que possui uma referência de IMemoryOwner<T> possui o buffer. O exemplo a seguir usa uma instância de IMemoryOwner<T> para refletir a propriedade de um buffer de Memory<T>.

```
using System;
using System.Buffers;

class Example
{
    static void Main()
    {
        IMemoryOwner<char> owner = MemoryPool<char>.Shared.Rent();

        Console.Write("Enter a number: ");
        try {
            var value = Int32.Parse(Console.ReadLine());

            var memory = owner.Memory;

            WriteInt32ToBuffer(value, memory);

            DisplayBufferToConsole(owner.Memory.Slice(0, value.ToString().Length));
        }
        catch (FormatException) {
            Console.WriteLine("You did not enter a valid number.");
        }
        catch (OverflowException) {
            Console.WriteLine($"You entered a number less than {Int32.MinValue:N0} or greater than
{Int32.MaxValue:N0}.");
        }
        finally {
            owner?.Dispose();
        }
    }

    static void WriteInt32ToBuffer(int value, Memory<char> buffer)
    {
        var strValue = value.ToString();

        var span = buffer.Span;
        for (int ctr = 0; ctr < strValue.Length; ctr++)
            span[ctr] = strValue[ctr];
    }

    static void DisplayBufferToConsole(Memory<char> buffer) =>
        Console.WriteLine($"Contents of the buffer: '{buffer}'");

}
```

Podemos também escrever este exemplo com `using`:

```
using System;
using System.Buffers;

class Example
{
    static void Main()
    {
        using (IMemoryOwner<char> owner = MemoryPool<char>.Shared.Rent())
        {
            Console.Write("Enter a number: ");
            try {
                var value = Int32.Parse(Console.ReadLine());

                var memory = owner.Memory;
                WriteInt32ToBuffer(value, memory);
                DisplayBufferToConsole(memory.Slice(0, value.ToString().Length));
            }
            catch (FormatException) {
                Console.WriteLine("You did not enter a valid number.");
            }
            catch (OverflowException) {
                Console.WriteLine($"You entered a number less than {Int32.MinValue:N0} or greater than
{Int32.MaxValue:N0}.");
            }
        }
    }

    static void WriteInt32ToBuffer(int value, Memory<char> buffer)
    {
        var strValue = value.ToString();

        var span = buffer.Slice(0, strValue.Length).Span;
        strValue.AsSpan().CopyTo(span);
    }

    static void DisplayBufferToConsole(Memory<char> buffer) =>
        Console.WriteLine($"Contents of the buffer: '{buffer}'");
}
```

Neste código:

- O método `Main` contém a referência à instância de IMemoryOwner<T>, portanto, o método `Main` é o proprietário do buffer.
- Os métodos `WriteInt32ToBuffer` e `DisplayBufferToConsole` aceitam Memory<T> como uma API pública. Portanto, eles são consumidores do buffer, e o consomem somente um de cada vez.

Embora o método `WriteInt32ToBuffer` seja destinado a gravar um valor no buffer, isso não se aplica ao método `DisplayBufferToConsole`. Para refletir isso, ele poderia aceitar um argumento de tipo ReadOnlyMemory<T>. Para obter informações adicionais sobre ReadOnlyMemory<T>, consulte #2 de regra: Use ReadOnlySpan<t > ou ReadOnlyMemory<t > se o buffer deve ser somente leitura.

Instâncias de Memory<T> "sem proprietário"

É possível criar uma instância de Memory<T> sem usar IMemoryOwner<T>. Nesse caso, a propriedade do buffer é implícita, em vez de explícita, e tem suporte apenas para o modelo de proprietário único. É possível fazer isso da seguinte maneira:

- Chamar um dos construtores de Memory<T> diretamente, passando um `T[]`, assim como no exemplo a seguir.
- Chamar o método de extensão String.AsMemory para criar uma instância de `ReadOnlyMemory<char>`.

```
using System;

class Example
{
    static void Main()
    {
        Memory<char> memory = new char[64];

        Console.Write("Enter a number: ");
        var value = Int32.Parse(Console.ReadLine());

        WriteInt32ToBuffer(value, memory);
        DisplayBufferToConsole(memory);
    }

    static void WriteInt32ToBuffer(int value, Memory<char> buffer)
    {
        var strValue = value.ToString();
        strValue.AsSpan().CopyTo(buffer.Slice(0, strValue.Length).Span);
    }

    static void DisplayBufferToConsole(Memory<char> buffer) =>
        Console.WriteLine($"Contents of the buffer: '{buffer}'");
}
```

O método que inicialmente cria a instância de Memory<T> é o proprietário implícito do buffer. Não é possível transferir a propriedade para nenhum outro componente porque não há instâncias de IMemoryOwner<T> para facilitar a transferência. Como alternativa, imagine que o coletor de lixo do runtime possui o buffer, e que todos os métodos apenas o consomem.

Diretrizes de uso

Como um bloco de memória pertence, mas se destina a ser passado para vários componentes, alguns dos quais podem operar em um determinado bloco de memória simultaneamente, é importante estabelecer diretrizes para o uso de Memory<T> e Span<T>. Diretrizes são necessárias porque:

- É possível que um componente retenha uma referência a um bloco de memória depois que o respectivo proprietário o liberar.
- É possível que um componente opere em um buffer, ao mesmo tempo em que outro componente esteja operando nele, corrompendo os dados no buffer durante o processo.
- Embora a natureza alocada na pilha do Span<T> otimize o desempenho e torne Span<T> o tipo preferencial para a operação em um bloco de memória, ele também submete Span<T> para algumas restrições importantes. É importante saber quando usar Span<T> e quando usar Memory<T>.

A seguir estão nossas recomendações para usar com êxito o Memory<T> e os tipos relacionados. A orientação que se aplica a Memory<T> e a Span<T> também se aplica a ReadOnlyMemory<T> e a ReadOnlySpan<T>, a menos que seja explicitamente definido de outra forma.

#1 de regra: para uma API síncrona, use<T > de span em vez de memória<T > como um parâmetro, se possível.

Span<T> é mais versátil do que Memory<T> e pode representar uma variedade maior de buffers de memória contíguos. Span<T> também oferece um melhor desempenho que Memory<T>. Por fim, é possível usar a propriedade Memory<T>.Span para converter uma instância de Memory<T> em Span<T>, embora a conversão de Span<T> em Memory<T> seja inviável. Portanto, se os chamadores tiverem uma instância de Memory<T>, eles poderão chamar seus métodos com os parâmetros Span<T> de qualquer maneira.

Usando um parâmetro de tipo Span<T> em vez de o tipo Memory<T>, você pode também escrever uma

implementação correta do método de consumo. Você receberá automaticamente verificações de tempo de compilação para garantir que não esteja tentando acessar o buffer, além da concessão do método (saiba mais sobre isso a seguir).

Às vezes, você terá que usar um parâmetro Memory<T> em vez de um parâmetro Span<T>, mesmo que esteja totalmente síncrono. Talvez uma API da qual você dependa aceite apenas argumentos Memory<T>. Isso é possível, mas conheça as vantagens e desvantagens envolvidas com o uso de Memory<T> de maneira síncrona.

#2 de regra: Use ReadOnlySpan<T > ou ReadOnlyMemory<T > se o buffer deve ser somente leitura.

Nos exemplos anteriores, o método `DisplayBufferToConsole` apenas lê no buffer, sem modificar o respectivo conteúdo. A assinatura do método deve ser alterada para a seguinte.

```
void DisplayBufferToConsole(ReadOnlyMemory<char> buffer);
```

Na verdade, se combinarmos esta regra com a Regra 1, poderemos fazer ainda melhor e reescrever a assinatura do método da seguinte forma:

```
void DisplayBufferToConsole(ReadOnlySpan<char> buffer);
```

O método `DisplayBufferToConsole` agora funciona praticamente com todos os tipos de buffer possíveis: `T[]`, armazenamento alocado com stackalloc, e assim por diante. Você pode, inclusive, passar um String diretamente nele!

#3 de regra: se o método aceitar memória<T > e retornar `void`, você não deverá usar a instância de<de memória > T depois que o método retornar.

Isso está relacionado ao conceito de "concessão" mencionado anteriormente. A concessão de um retorno de void do método na instância de Memory<T> começa quando o método é inserido e termina quando o método é encerrado. Considere o exemplo a seguir, que chama `Log` em um loop com base na entrada do console.

```
using System;
using System.Buffers;

public class Example
{
    // implementation provided by third party
    static extern void Log(ReadOnlyMemory<char> message);

    // user code
    public static void Main()
    {
        using (var owner = MemoryPool<char>.Shared.Rent())
        {
            var memory = owner.Memory;
            var span = memory.Span;
            while (true)
            {
                int value = Int32.Parse(Console.ReadLine());
                if (value < 0)
                    return;

                int numCharsWritten = ToBuffer(value, span);
                Log(memory.Slice(0, numCharsWritten));
            }
        }
    }

    private static int ToBuffer(int value, Span<char> span)
    {
        string strValue = value.ToString();
        int length = strValue.Length;
        strValue.AsSpan().CopyTo(span.Slice(0, length));
        return length;
    }
}
```

Se `Log` for um método totalmente síncrono, esse código se comportará como esperado porque há apenas um consumidor ativo da instância de memória a qualquer momento. Mas imagine que `Log` tenha essa implementação.

```
// !!! INCORRECT IMPLEMENTATION !!!
static void Log(ReadOnlyMemory<char> message)
{
    // Run in background so that we don't block the main thread while performing IO.
    Task.Run(() =>
    {
        StreamWriter sw = File.AppendText(@".\input-numbers.dat");
        sw.WriteLine(message);
    });
}
```

Nesta implementação, `Log` viola a respectiva concessão porque ela ainda tenta usar a instância de Memory<T> em segundo plano, após o método original ter retornado. O método `Main` pode alterar o buffer enquanto `Log` tenta ler nele, o que pode resultar em dados corrompidos.

Há várias maneiras de resolver isso:

- O método `Log` pode retornar uma classe Task em vez de `void`, assim como faz a seguinte implementação do método `Log`.

```
// An acceptable implementation.
static Task Log(ReadOnlyMemory<char> message)
{
    // Run in the background so that we don't block the main thread while performing IO.
    return Task.Run(() => {
                StreamWriter sw = File.AppendText(@".\input-numbers.dat");
        sw.WriteLine(message);
        sw.Flush();
    });
}
```

- É possível implementar `Log` da seguinte maneira:

```
// An acceptable implementation.
static void Log(ReadOnlyMemory<char> message)
{
    string defensiveCopy = message.ToString();
    // Run in the background so that we don't block the main thread while performing IO.
    Task.Run(() => {
        StreamWriter sw = File.AppendText(@".\input-numbers.dat");
        sw.WriteLine(defensiveCopy);
        sw.Flush();
    });
}
```

#4 de regra: se o seu método aceitar uma memória<T > e retornar uma tarefa, você não deverá usar a instância da memória<T > depois que a tarefa for transferida para um estado terminal.

Esta é apenas a variante assíncrona da Regra 3. O método `Log` do exemplo anterior pode ser escrito da seguinte forma para cumprir esta regra:

```
// An acceptable implementation.
static void Log(ReadOnlyMemory<char> message)
{
    // Run in the background so that we don't block the main thread while performing IO.
    Task.Run(() => {
        string defensiveCopy = message.ToString();
        StreamWriter sw = File.AppendText(@".\input-numbers.dat");
        sw.WriteLine(defensiveCopy);
        sw.Flush();
    });
}
```

Nesse caso, "estado terminal" significa que a tarefa passa para um estado concluído, com falha ou cancelado. Em outras palavras, "estado terminal" significa "qualquer coisa que cause atraso de lançamento ou continuação de execução".

Essa orientação se aplica a métodos que retornam Task, Task<TResult>, ValueTask<TResult> ou outros tipos semelhantes.

#5 de regra: se o Construtor aceitar memória<T > como um parâmetro, os métodos de instância no objeto construído serão considerados consumidores da instância de memória<T >.

Considere o exemplo a seguir:

```
class OddValueExtractor
{
    public OddValueExtractor(ReadOnlyMemory<int> input);
    public bool TryReadNextOddValue(out int value);
}

void PrintAllOddValues(ReadOnlyMemory<int> input)
{
    var extractor = new OddValueExtractor(input);
    while (extractor.TryReadNextOddValue(out int value))
    {
      Console.WriteLine(value);
    }
}
```

Nesse caso, o construtor `OddValueExtractor` aceita um `ReadOnlyMemory<int>` como parâmetro, para que o próprio construtor seja um consumidor da instância de `ReadOnlyMemory<int>`, e todos os métodos da instância no valor retornado também sejam consumidores da instância original de `ReadOnlyMemory<int>`. Isso significa que `TryReadNextOddValue` consome a instância de `ReadOnlyMemory<int>`, mesmo que a instância não seja passada diretamente para o método `TryReadNextOddValue`.

#6 de regra: se você tiver uma propriedade de tipo de memória<T > tipada (ou um método de instância equivalente) em seus tipos, os métodos de instância nesse objeto serão considerados consumidores da instância de > de memória<T.

Na verdade, esta é apenas uma variante da Regra 5. Esta regra existe porque presume-se que os setters de propriedade ou métodos equivalentes devam capturar e persistir as respectivas entradas, de modo que os métodos da instância no mesmo objeto possam usar o estado capturado.

O exemplo a seguir aciona esta regra:

```
class Person
{
    // Settable property.
    public Memory<char> FirstName { get; set; }

    // alternatively, equivalent "setter" method
    public SetFirstName(Memory<char> value);

    // alternatively, a public settable field
    public Memory<char> FirstName;
}
```

#7 de regra: se você tiver uma referência de > IMemoryOwner<T, você deverá, em algum momento, descartar ou transferir sua propriedade (mas não ambas).

Como uma instância de Memory<T> pode ter suporte de memória gerenciada e não gerenciada, o proprietário deve chamar MemoryPool<T>.Dispose quando concluir o trabalho realizado na instância de Memory<T>. Como alternativa, o proprietário pode transferir a propriedade da instância de IMemoryOwner<T> para um componente diferente, até que o componente receptor se torne responsável por chamar MemoryPool<T>.Dispose no tempo adequado (saiba mais sobre isso a seguir).

Uma falha ao chamar o método Dispose pode causar perdas de memória não gerenciada ou outra degradação do desempenho.

Esta regra também se aplica ao código que chama métodos de fábrica, como MemoryPool<T>.Rent. O chamador se torna proprietário do IMemoryOwner<T> retornado e fica responsável pelo descarte da instância após concluída.

Regra #8: se você tiver um parâmetro IMemoryOwner<T > na superfície da API, você estará aceitando a propriedade dessa instância.

Aceitar uma instância desse tipo indica que o componente pretende se apropriar dessa instância. O componente se torna responsável pelo descarte adequado de acordo com a Regra 7.

Os componentes que transferem a propriedade da instância de IMemoryOwner<T> para um componente diferente não devem mais usar essa instância após a conclusão da chamada do método.

> **IMPORTANT**
> Se o construtor aceitar IMemoryOwner<T> como parâmetro, o respectivo tipo deverá implementar IDisposable, e o método Dispose deverá chamar MemoryPool<T>.Dispose.

#9 de regra: se você estiver encapsulando um método p/invoke síncrono, sua API deverá aceitar a extensão<T > como um parâmetro.

De acordo com a Regra 1, Span<T> geralmente é o tipo correto a ser usado para APIs síncronas. Você pode fixar instâncias do Span<T> por meio da palavra-chave `fixed`, como no exemplo a seguir.

```
using System.Runtime.InteropServices;

[DllImport(...)]
private static extern unsafe int ExportedMethod(byte* pbData, int cbData);

public unsafe int ManagedWrapper(Span<byte> data)
{
    fixed (byte* pbData = &MemoryMarshal.GetReference(data))
    {
        int retVal = ExportedMethod(pbData, data.Length);

        /* error checking retVal goes here */

        return retVal;
    }
}
```

No exemplo anterior, `pbData` poderá ser nulo, se o alcance da entrada estiver vazio. Se o método exportado exigir que `pbData` seja não nulo, mesmo que `cbData` seja 0, o método poderá ser implementado da seguinte maneira:

```
public unsafe int ManagedWrapper(Span<byte> data)
{
    fixed (byte* pbData = &MemoryMarshal.GetReference(data))
    {
        byte dummy = 0;
        int retVal = ExportedMethod((pbData != null) ? pbData : &dummy, data.Length);

        /* error checking retVal goes here */

        return retVal;
    }
}
```

Regra #10: se você estiver encapsulando um método p/invoke assíncrono, sua API deverá aceitar memória<T > como um parâmetro.

Como não é possível usar a palavra-chave `fixed` em operações assíncronas, use o método Memory<T>.Pin para fixar instâncias de Memory<T>, independentemente do tipo de memória contígua que a instância representar. O exemplo a seguir mostra como usar esta API para executar uma chamada de P/Invoke assíncrona.

```
using System.Runtime.InteropServices;

[UnmanagedFunctionPointer(...)]
private delegate void OnCompletedCallback(IntPtr state, int result);

[DllImport(...)]
private static extern unsafe int ExportedAsyncMethod(byte* pbData, int cbData, IntPtr pState, IntPtr
lpfnOnCompletedCallback);

private static readonly IntPtr _callbackPtr = GetCompletionCallbackPointer();

public unsafe Task<int> ManagedWrapperAsync(Memory<byte> data)
{
    // setup
    var tcs = new TaskCompletionSource<int>();
    var state = new MyCompletedCallbackState
    {
        Tcs = tcs
    };
    var pState = (IntPtr)GCHandle.Alloc(state);

    var memoryHandle = data.Pin();
    state.MemoryHandle = memoryHandle;

    // make the call
    int result;
    try
    {
        result = ExportedAsyncMethod((byte*)memoryHandle.Pointer, data.Length, pState, _callbackPtr);
    }
    catch
    {
        ((GCHandle)pState).Free(); // cleanup since callback won't be invoked
        memoryHandle.Dispose();
        throw;
    }

    if (result != PENDING)
    {
        // Operation completed synchronously; invoke callback manually
        // for result processing and cleanup.
        MyCompletedCallbackImplementation(pState, result);
    }

    return tcs.Task;
}

private static void MyCompletedCallbackImplementation(IntPtr state, int result)
{
    GCHandle handle = (GCHandle)state;
    var actualState = (MyCompletedCallbackState)state;
    handle.Free();
    actualState.MemoryHandle.Dispose();

    /* error checking result goes here */

    if (error)
    {
        actualState.Tcs.SetException(...);
    }
    else
    {
        actualState.Tcs.SetResult(result);
    }
}

private static IntPtr GetCompletionCallbackPointer()
{
```

```
    OnCompletedCallback callback = MyCompletedCallbackImplementation;
    GCHandle.Alloc(callback); // keep alive for lifetime of application
    return Marshal.GetFunctionPointerForDelegate(callback);
}

private class MyCompletedCallbackState
{
    public TaskCompletionSource<int> Tcs;
    public MemoryHandle MemoryHandle;
}
```

Consulte também

- System.Memory<T>
- System.Buffers.IMemoryOwner<T>
- System.Span<T>

Interoperabilidade nativa

23/10/2019 • 2 minutes to read • Edit Online

Os artigos a seguir mostram várias maneiras de fazer "interoperabilidade nativa" no .NET.

Existem alguns motivos para chamar em código nativo:

- Os sistemas operacionais vêm com um grande volume de APIs que não estão presentes nas bibliotecas de classes gerenciadas. Um exemplo perfeito para esse cenário seria o acesso a funções de gerenciamento de hardware ou do sistema operacional.
- Comunicação com outros componentes que têm ou podem produzir ABIs de estilo C (ABIs nativas), como o código Java que é exposto via JNI (Interface Nativa do Java) ou qualquer outra linguagem gerenciada que pode produzir um componente nativo.
- No Windows, a maioria dos softwares que são instalados, como o pacote Microsoft Office, registra os componentes COM que representam seus programas e permitem que sejam automatizados ou usados pelos desenvolvedores. Isso também requer interoperabilidade nativa.

A lista acima não abrange todas as possíveis situações e cenários em que o desenvolvedor desejaria/gostaria de/precisaria fazer interface com componentes nativos. A biblioteca de classes do .NET, por exemplo, usa o suporte para interoperabilidade nativa para implementar um número razoável de APIs, como suporte ao console e manipulação, acesso ao sistema de arquivos e outras. No entanto, é importante observar que há uma opção, se necessário.

> **NOTE**
> A maioria dos exemplos nesta seção será apresentada para todas as três plataformas com suporte para .NET Core (Windows, Linux e macOS). No entanto, para alguns exemplos ilustrativos e curtos, é exibida apenas uma amostra que usa nomes de arquivo e extensões do Windows (ou seja, "dll" para bibliotecas). Isso não significa que tais recursos não estão disponíveis em Linux ou macOS; foi feito simplesmente para maior conveniência.

Consulte também

- Invocação de plataforma (P/Invoke)
- Marshaling de tipo
- Melhores práticas de interoperabilidade nativa

Invocação de plataforma (P/Invoke)

31/10/2019 • 11 minutes to read • Edit Online

P/Invoke é uma tecnologia que permite acessar structs, retornos de chamada e funções em bibliotecas não gerenciadas de um código gerenciado. A maior parte da API do P/Invoke está contida em dois namespaces: `System` e `System.Runtime.InteropServices`. O uso desses dois namespaces fornece as ferramentas que descrevem como você deseja se comunicar com o componente nativo.

Vamos começar com exemplo mais comum, que é chamar funções não gerenciadas no seu código gerenciado. Vamos mostrar uma caixa de mensagem de um aplicativo de linha de comando:

```
using System;
using System.Runtime.InteropServices;

public class Program
{
    // Import user32.dll (containing the function we need) and define
    // the method corresponding to the native function.
    [DllImport("user32.dll", CharSet = CharSet.Unicode, SetLastError = true)]
    private static extern int MessageBox(IntPtr hWnd, string lpText, string lpCaption, uint uType);

    public static void Main(string[] args)
    {
        // Invoke the function as a regular managed method.
        MessageBox(IntPtr.Zero, "Command-line message box", "Attention!", 0);
    }
}
```

O exemplo anterior é simples, mas mostra o que é necessário para invocar funções não gerenciadas de um código gerenciado. Vamos analisar o exemplo:

- A linha 1 mostra a instrução de uso para o namespace `System.Runtime.InteropServices` que contém todos os itens necessários.
- A linha 7 apresenta o atributo `DllImport`. Esse atributo é crucial, pois informa ao tempo de execução que deve carregar a DLL não gerenciada. A cadeia de caracteres passada é a DLL na qual nossa função de destino está incluída. Além disso, especifica qual conjunto de caracteres deve ser usado para realizar marshaling de cadeias de caracteres. Por fim, especifica que essa função chama SetLastError e que o tempo de execução deve capturar esse código de erro para que o usuário possa recuperá-lo via Marshal.GetLastWin32Error().
- A linha 8 é o ponto crucial do trabalho do P/Invoke. Define um método gerenciado que tem **exatamente a mesma assinatura** que o não gerenciado. A declaração tem uma nova palavra-chave que você pode observar, `extern`, que informa ao tempo de execução que é um método externo; quando invocado, o tempo de execução deve encontrá-la na DLL especificada no atributo `DllImport`.

O restante do exemplo é simplesmente chamar o método como você faria com qualquer outro método gerenciado.

A amostra é semelhante para macOS. O nome da biblioteca deve ser alterado no atributo `DllImport`, pois o macOS tem um esquema diferente de nomenclatura para bibliotecas dinâmicas. O exemplo a seguir usa a função `getpid(2)` para obter a ID do processo do aplicativo e imprimi-la para o console:

```
using System;
using System.Runtime.InteropServices;

namespace PInvokeSamples
{
    public static class Program
    {
        // Import the libSystem shared library and define the method
        // corresponding to the native function.
        [DllImport("libSystem.dylib")]
        private static extern int getpid();

        public static void Main(string[] args)
        {
            // Invoke the function and get the process ID.
            int pid = getpid();
            Console.WriteLine(pid);
        }
    }
}
```

Também é semelhante no Linux. O nome da função é o mesmo, já que `getpid(2)` é uma chamada do sistema POSIX padrão.

```
using System;
using System.Runtime.InteropServices;

namespace PInvokeSamples
{
    public static class Program
    {
        // Import the libc shared library and define the method
        // corresponding to the native function.
        [DllImport("libc.so.6")]
        private static extern int getpid();

        public static void Main(string[] args)
        {
            // Invoke the function and get the process ID.
            int pid = getpid();
            Console.WriteLine(pid);
        }
    }
}
```

Chamando código gerenciado do código não gerenciado

O tempo de execução viabiliza o fluxo da comunicação nas duas direções, o que permite retornar a chamada no código gerenciado das funções nativas usando ponteiros de função. A coisa mais próxima a um ponteiro de função no código gerenciado é um **delegado**; portanto, isso é usado para permitir retornos de chamada do código nativo para o código gerenciado.

A maneira de usar esse recurso é semelhante ao processo gerenciado para nativo, conforme descrito anteriormente. Para um retorno de chamada específico, você define um delegado que corresponda à assinatura e o passa para o método externo. O tempo de execução cuidará do resto.

```
using System;
using System.Runtime.InteropServices;

namespace ConsoleApplication1
{
    public static class Program
    {
        // Define a delegate that corresponds to the unmanaged function.
        private delegate bool EnumWC(IntPtr hwnd, IntPtr lParam);

        // Import user32.dll (containing the function we need) and define
        // the method corresponding to the native function.
        [DllImport("user32.dll")]
        private static extern int EnumWindows(EnumWC lpEnumFunc, IntPtr lParam);

        // Define the implementation of the delegate; here, we simply output the window handle.
        private static bool OutputWindow(IntPtr hwnd, IntPtr lParam)
        {
            Console.WriteLine(hwnd.ToInt64());
            return true;
        }

        public static void Main(string[] args)
        {
            // Invoke the method; note the delegate as a first parameter.
            EnumWindows(OutputWindow, IntPtr.Zero);
        }
    }
}
```

Antes de analisar o exemplo, é uma boa ideia examinar as assinaturas das funções não gerenciadas com as quais você precisa trabalhar. A função a ser chamada para enumerar todas as janelas tem esta assinatura:
`BOOL EnumWindows (WNDENUMPROC lpEnumFunc, LPARAM lParam);`

O primeiro parâmetro é um retorno de chamada. Esse retorno de chamada tem a seguinte assinatura:
`BOOL CALLBACK EnumWindowsProc (HWND hwnd, LPARAM lParam);`

Agora, vamos analisar o exemplo:

- A linha 9 no exemplo define um delegado que corresponde à assinatura do retorno de chamada do código não gerenciado. Observe como os tipos LPARAM e HWND são representados usando `IntPtr` no código gerenciado.
- As linhas 13 e 14 introduzem a função `EnumWindows` da biblioteca user32.dll.
- As linhas de 17 a 20 implementam o delegado. Neste exemplo simples, queremos apenas produzir o identificador para o console.
- Por fim, na linha 24, chamamos o método externo e passamos o delegado.

Os exemplos de Linux e macOS são mostrados abaixo. Para eles, usamos a função `ftw` que pode ser encontrada em `libc`, a biblioteca C. Essa função é usada para percorrer as hierarquias de diretório e leva um ponteiro para uma função como um dos seus parâmetros. Essa função tem a seguinte assinatura:
`int (*fn) (const char *fpath, const struct stat *sb, int typeflag)`.

```
using System;
using System.Runtime.InteropServices;

namespace PInvokeSamples
{
    public static class Program
    {
        // Define a delegate that has the same signature as the native function.
        private delegate int DirClbk(string fName, StatClass stat, int typeFlag);

        // Import the libc and define the method to represent the native function.
        [DllImport("libc.so.6")]
        private static extern int ftw(string dirpath, DirClbk cl, int descriptors);

        // Implement the above DirClbk delegate;
        // this one just prints out the filename that is passed to it.
        private static int DisplayEntry(string fName, StatClass stat, int typeFlag)
        {
            Console.WriteLine(fName);
            return 0;
        }

        public static void Main(string[] args)
        {
            // Call the native function.
            // Note the second parameter which represents the delegate (callback).
            ftw(".", DisplayEntry, 10);
        }
    }

    // The native callback takes a pointer to a struct. The below class
    // represents that struct in managed code. You can find more information
    // about this in the section on marshalling below.
    [StructLayout(LayoutKind.Sequential)]
    public class StatClass
    {
        public uint DeviceID;
        public uint InodeNumber;
        public uint Mode;
        public uint HardLinks;
        public uint UserID;
        public uint GroupID;
        public uint SpecialDeviceID;
        public ulong Size;
        public ulong BlockSize;
        public uint Blocks;
        public long TimeLastAccess;
        public long TimeLastModification;
        public long TimeLastStatusChange;
    }
}
```

O exemplo do macOS usa a mesma função; a única diferença é o argumento para o atributo `DllImport`, pois o macOS mantém `libc` em um local diferente.

```
using System;
using System.Runtime.InteropServices;

namespace PInvokeSamples
{
    public static class Program
    {
        // Define a delegate that has the same signature as the native function.
        private delegate int DirClbk(string fName, StatClass stat, int typeFlag);

        // Import the libc and define the method to represent the native function.
        [DllImport("libSystem.dylib")]
        private static extern int ftw(string dirpath, DirClbk cl, int descriptors);

        // Implement the above DirClbk delegate;
        // this one just prints out the filename that is passed to it.
        private static int DisplayEntry(string fName, StatClass stat, int typeFlag)
        {
            Console.WriteLine(fName);
            return 0;
        }

        public static void Main(string[] args)
        {
            // Call the native function.
            // Note the second parameter which represents the delegate (callback).
            ftw(".", DisplayEntry, 10);
        }
    }

    // The native callback takes a pointer to a struct. The below class
    // represents that struct in managed code.
    [StructLayout(LayoutKind.Sequential)]
    public class StatClass
    {
        public uint DeviceID;
        public uint InodeNumber;
        public uint Mode;
        public uint HardLinks;
        public uint UserID;
        public uint GroupID;
        public uint SpecialDeviceID;
        public ulong Size;
        public ulong BlockSize;
        public uint Blocks;
        public long TimeLastAccess;
        public long TimeLastModification;
        public long TimeLastStatusChange;
    }
}
```

Os exemplos anteriores dependem de parâmetros e, em ambos os casos, os parâmetros são fornecidos como tipos gerenciados. O tempo de execução faz a "coisa certa" e processa esses parâmetros em seus equivalentes no outro lado. Saiba mais sobre é realizado o marshaling de tipos para código nativo em nossa página sobre Marshaling de tipo.

Mais recursos

- Wiki do PInvoke.net uma wiki excelente com informações sobre as APIs comuns do Windows e como chamá-las.
- P/Invoke em C++/CLI
- Documentação do Mono no P/Invoke

Marshaling de tipo

23/10/2019 • 8 minutes to read • Edit Online

Marshaling é o processo de transformar tipos quando precisam atravessar entre código nativo e gerenciado.

O marshaling é necessário porque os tipos são diferentes, no código gerenciado e não gerenciado. No código gerenciado, por exemplo, você tem uma `String`; no mundo não gerenciado, as cadeias de caracteres podem ser Unicode ("larga"), não Unicode, terminada em nulo, ASCII, etc. Por padrão, o subsistema do P/Invoke tenta fazer a coisa certa com base no comportamento padrão, descrito neste artigo. Contudo, nas situações em que você precisa de controle extra, pode utilizar o atributo MarshalAs para especificar qual é o tipo esperado no lado não gerenciado. Por exemplo, se você quiser que a cadeia de caracteres seja enviada como uma cadeia de caracteres ANSI terminada em nulo, faça o seguinte:

```
[DllImport("somenativelibrary.dll")]
static extern int MethodA([MarshalAs(UnmanagedType.LPStr)] string parameter);
```

Regras padrão para tipos comuns de marshaling

Geralmente, o tempo de execução tenta fazer a "coisa certa" ao realizar marshaling para exigir a menor quantidade de trabalho de você. As tabelas a seguir descrevem como cada tipo tem o marshaling realizado por padrão quando usado em um parâmetro ou campo. Os tipos de caracteres e números inteiros de largura fixa C99/C++11 são usados para garantir que a tabela a seguir esteja correta para todas as plataformas. Use qualquer tipo nativo que tenha os mesmos requisitos de alinhamento e tamanho que esses tipos.

Esta primeira tabela descreve os mapeamentos para vários tipos para os quais o marshaling é o mesmo para ambos P/Invoke e o marshaling do campo.

TIPO .NET	TIPO NATIVO
`byte`	`uint8_t`
`sbyte`	`int8_t`
`short`	`int16_t`
`ushort`	`uint16_t`
`int`	`int32_t`
`uint`	`uint32_t`
`long`	`int64_t`
`ulong`	`uint64_t`
`char`	`char` ou `char16_t` dependendo do `CharSet` do P/Invoke ou da estrutura. Confira a documentação do conjunto de caracteres.

TIPO .NET	TIPO NATIVO
`string`	`char*` ou `char16_t*` dependendo do `CharSet` do P/Invoke ou da estrutura. Confira a documentação do conjunto de caracteres.
`System.IntPtr`	`intptr_t`
`System.UIntPtr`	`uintptr_t`
Tipos de ponteiro do .NET (por exemplo: `void*`)	`void*`
Tipo derivado de `System.Runtime.InteropServices.SafeHandle`	`void*`
Tipo derivado de `System.Runtime.InteropServices.CriticalHandle`	`void*`
`bool`	Tipo `BOOL` Win32
`decimal`	Struct `DECIMAL` COM
Representante do .NET	Ponteiro de função nativo
`System.DateTime`	Tipo `DATE` Win32
`System.Guid`	Tipo `GUID` Win32

Algumas categorias de marshaling terão padrões diferentes se você estiver realizando marshaling como um parâmetro ou uma estrutura.

TIPO .NET	TIPO NATIVO (PARÂMETRO)	TIPO NATIVO (CAMPO)
Matriz .NET	Um ponteiro para o início de uma matriz de representações nativas dos elementos da matriz.	Não é permitido sem um atributo `[MarshalAs]`
Uma classe com um `LayoutKind` de `Sequential` ou `Explicit`	Um ponteiro para a representação nativa da classe	A representação nativa da classe

A tabela a seguir inclui as regras de marshaling padrão que são somente do Windows. Em plataformas que não são Windows, você não pode realizar marshal desses tipos.

TIPO .NET	TIPO NATIVO (PARÂMETRO)	TIPO NATIVO (CAMPO)	
`object`	`VARIANT`	`IUnknown*`	
`System.Array`	Interface COM	Não é permitida sem um atributo `[MarshalAs]`	
`System.ArgIterator`	`va_list`	Não permitida	

TIPO .NET	TIPO NATIVO (PARÂMETRO)	TIPO NATIVO (CAMPO)	
`System.Collections.IEnumerat`	`IEnumVARIANT*`	Não permitida	
`System.Collections.IEnumerab`	`IDispatch*`	Não permitida	
`System.DateTimeOffset`	`int64_t` representando o número de tiques desde a meia-noite de 1º de janeiro de 1601		`int64_t` representando o número de tiques desde a meia-noite de 1º de janeiro de 1601

Alguns tipos só podem ter o marshaling realizado como parâmetros e não como campos. Esses tipos estão listados na tabela a seguir:

TIPO .NET	TIPO NATIVO (PARÂMETRO SOMENTE)
`System.Text.StringBuilder`	`char*` ou `char16_t*` dependendo do `CharSet` do P/Invoke. Confira a documentação do conjunto de caracteres.
`System.ArgIterator`	`va_list` (no Windows x86/x64/arm64 somente)
`System.Runtime.InteropServices.ArrayWithOffset`	`void*`
`System.Runtime.InteropServices.HandleRef`	`void*`

Se esses padrões não fizerem exatamente o que você deseja, personalize como os parâmetros têm o marshaling realizado. O artigo sobre marshaling de parâmetro explica como personalizar a forma como os tipos de parâmetros diferentes têm o marshaling realizado.

Marshaling padrão em cenários COM

Quando você chama métodos em objetos COM no .NET, o tempo de execução do .NET altera a regras de marshaling padrão para corresponder à semântica de COM. A tabela a seguir lista as regras que os tempos de execução do .NET usam em cenários COM:

TIPO .NET	TIPO NATIVO (CHAMADAS DE MÉTODO COM)
`bool`	`VARIANT_BOOL`
`StringBuilder`	`LPWSTR`
`string`	`BSTR`
Tipos delegados	`_Delegate*` no .NET Framework. Não permitido no .NET Core.
`System.Drawing.Color`	`OLECOLOR`
Matriz .NET	`SAFEARRAY`
`string[]`	`SAFEARRAY` de `BSTR`s

Marshaling de classes e structs

Outro aspecto do marshaling de tipos é como passar um struct para um método não gerenciado. Por exemplo, alguns dos métodos não gerenciados requerem um struct como parâmetro. Nesses casos, você precisa criar um struct correspondente ou uma classe na parte gerenciada do mundo para usar como parâmetro. No entanto, definir a classe não é suficiente; também é necessário ensinar o marshaler como mapear campos na classe para o struct não gerenciado. Aqui, o atributo `StructLayout` se torna útil.

```
[DllImport("kernel32.dll")]
static extern void GetSystemTime(SystemTime systemTime);

[StructLayout(LayoutKind.Sequential)]
class SystemTime {
    public ushort Year;
    public ushort Month;
    public ushort DayOfWeek;
    public ushort Day;
    public ushort Hour;
    public ushort Minute;
    public ushort Second;
    public ushort Milsecond;
}

public static void Main(string[] args) {
    SystemTime st = new SystemTime();
    GetSystemTime(st);
    Console.WriteLine(st.Year);
}
```

O código anterior mostra um exemplo simples de chamar para a função `GetSystemTime()`. A parte interessante está na linha 4. O atributo especifica que os campos da classe devem ser mapeados em sequência até o struct no outro lado (não gerenciado). Isso significa que os nomes dos campos não são importantes, apenas sua ordem é importante, já que precisa corresponder ao struct não gerenciado, mostrado no exemplo abaixo:

```
typedef struct _SYSTEMTIME {
  WORD wYear;
  WORD wMonth;
  WORD wDayOfWeek;
  WORD wDay;
  WORD wHour;
  WORD wMinute;
  WORD wSecond;
  WORD wMilliseconds;
} SYSTEMTIME, *PSYSTEMTIME;
```

Às vezes, o marshaling padrão para sua estrutura não faz o que você precisa. O artigo Personalizando o marshaling de estrutura ensina como personalizar a forma como sua estrutura tem o marshaling realizado.

Personalização do marshaling de estrutura

04/11/2019 • 11 minutes to read • Edit Online

Às vezes, as regras de marshaling padrão para estruturas não são exatamente o que você precisa. Os tempos de execução do .NET fornecem alguns pontos de extensão para você personalizar o layout de sua estrutura e como os campos têm o marshaling realizado.

Personalização do layout de estrutura

O .NET fornece o atributo System.Runtime.InteropServices.StructLayoutAttribute e a enumeração System.Runtime.InteropServices.LayoutKind para permitir que você personalize como os campos são colocados na memória. As diretrizes a seguir ajudarão a evitar problemas comuns.

✔ **CONSIDERE** usar `LayoutKind.Sequential` sempre que possível.

✔ **USE** somente `LayoutKind.Explicit` no marshaling quando seu struct nativo também tiver um layout explícito, como uma união.

Evite usar `LayoutKind.Explicit` ao realizar o marshaling de estruturas em plataformas não Windows se você precisar direcionar os tempos de execução antes do .net Core 3,0. O tempo de execução do .NET Core antes de 3,0 não dá suporte à passagem de estruturas explícitas por valor para funções nativas em sistemas Intel ou AMD de 64 bits que não sejam Windows. No entanto, o tempo de execução dá suporte à passagem de estruturas explícitas por referência em todas as plataformas.

Personalização do marshaling de campo booliano

O código nativo tem muitas representações boolianas diferentes. Somente no Windows, há três formas de representar valores boolianos. O tempo de execução não conhece a definição nativa de sua estrutura, portanto, o melhor que ele pode fazer é estimar como realizar marshal de seus valores boolianos. O tempo de execução do .NET fornece uma maneira de indicar como realizar marshal de seu campo booliano. Os exemplos a seguir mostram como realizar marshal do .NET `bool` para diferentes tipos boolianos nativos.

Valores boolianos padrão para realizar marshaling como um valor `BOOL` Win32 nativo de 4 bytes conforme mostrado no exemplo a seguir:

```
public struct WinBool
{
    public bool b;
}
```

```
struct WinBool
{
    public BOOL b;
};
```

Se você quiser ser explícito, use o valor UnmanagedType.Bool para obter o mesmo comportamento descrito acima:

```
public struct WinBool
{
    [MarshalAs(UnmanagedType.Bool)]
    public bool b;
}
```

```
struct WinBool
{
    public BOOL b;
};
```

Usando os valores `UnmanagedType.U1` ou `UnmanagedType.I1` abaixo, você pode informar o tempo de execução para realizar marshal do campo `b` como um tipo `bool` nativo de 1 byte.

```
public struct CBool
{
    [MarshalAs(UnmanagedType.U1)]
    public bool b;
}
```

```
struct CBool
{
    public bool b;
};
```

No Windows, use o valor UnmanagedType.VariantBool para informar o tempo de execução para realizar marshal de seu valor booliano em um valor `VARIANT_BOOL` de 2 bytes:

```
public struct VariantBool
{
    [MarshalAs(UnmanagedType.VariantBool)]
    public bool b;
}
```

```
struct VariantBool
{
    public VARIANT_BOOL b;
};
```

> **NOTE**
>
> `VARIANT_BOOL` é diferente da maioria dos tipos de bool em que `VARIANT_TRUE = -1` e `VARIANT_FALSE = 0`. Além disso, todos os valores que não são iguais a `VARIANT_TRUE` são considerados falsos.

Personalização do marshaling de campo de matriz

O .NET também inclui algumas formas de personalizar o marshaling de matriz.

Por padrão, o .NET realiza marshal de matrizes como um ponteiro para uma lista contígua dos elementos:

```
public struct DefaultArray
{
    public int[] values;
}
```

```
struct DefaultArray
{
    int* values;
};
```

Se você estiver interagindo com APIs COM, talvez seja necessário realizar marshal de matrizes como objetos `SAFEARRAY*`. Use o valor System.Runtime.InteropServices.MarshalAsAttribute e UnmanagedType.SafeArray para informar o tempo de execução para realizar marshal de uma matriz como uma `SAFEARRAY*`:

```
public struct SafeArrayExample
{
    [MarshalAs(UnmanagedType.SafeArray)]
    public int[] values;
}
```

```
struct SafeArrayExample
{
    SAFEARRAY* values;
};
```

Se for necessário personalizar o tipo de elemento que está na `SAFEARRAY`, use os campos MarshalAsAttribute.SafeArraySubType e MarshalAsAttribute.SafeArrayUserDefinedSubType para personalizar o tipo de elemento exato da `SAFEARRAY`.

Se você precisar realizar marshal da matriz in-loco, use o valor UnmanagedType.ByValArray para informar o marshaler para realizar marshal da matriz in-loco. Ao usar esse marshaling, você também precisa fornecer um valor ao campo MarshalAsAttribute.SizeConst para o número de elementos na matriz, para que o tempo de execução possa alocar espaço corretamente para a estrutura.

```
public struct InPlaceArray
{
    [MarshalAs(UnmanagedType.ByValArray, SizeConst = 4)]
    public int[] values;
}
```

```
struct InPlaceArray
{
    int values[4];
};
```

> **NOTE**
> O .NET não dá suporte à realização de marshaling de um campo de matriz de comprimento variável como um Membro de Matriz Flexível C99.

Personalização do marshaling de campo de cadeia de caracteres

O .NET também fornece uma ampla variedade de personalizações para realizar marshaling de campos de cadeia de caracteres.

Por padrão, o .NET realiza marshal de uma cadeia de caracteres como um ponteiro para uma cadeia de caracteres terminada em nulo. A codificação depende do valor do campo StructLayoutAttribute.CharSet no System.Runtime.InteropServices.StructLayoutAttribute. Se nenhum atributo for especificado, a codificação será padronizada para uma codificação ANSI.

```
[StructLayout(LayoutKind.Sequential, CharSet = CharSet.Ansi)]
public struct DefaultString
{
    public string str;
}
```

```
struct DefaultString
{
    char* str;
};
```

```
[StructLayout(LayoutKind.Sequential, CharSet = CharSet.Unicode)]
public struct DefaultString
{
    public string str;
}
```

```
struct DefaultString
{
    char16_t* str; // Could also be wchar_t* on Windows.
};
```

Se você precisar usar codificações diferentes para campos distintos ou apenas prefere ser mais explícito em sua definição de struct, use os valores UnmanagedType.LPStr ou UnmanagedType.LPWStr em um atributo System.Runtime.InteropServices.MarshalAsAttribute.

```
public struct AnsiString
{
    [MarshalAs(UnmanagedType.LPStr)]
    public string str;
}
```

```
struct AnsiString
{
    char* str;
};
```

```
public struct UnicodeString
{
    [MarshalAs(UnmanagedType.LPWStr)]
    public string str;
}
```

```
struct UnicodeString
{
    char16_t* str; // Could also be wchar_t* on Windows.
};
```

Se quiser realizar marshal de suas cadeias de caracteres usando a codificação UTF-8, use o valor UnmanagedType.LPUTF8Str em seu MarshalAsAttribute.

```
public struct UTF8String
{
    [MarshalAs(UnmanagedType.LPUTF8Str)]
    public string str;
}
```

```
struct UTF8String
{
    char* str;
};
```

> **NOTE**
> O uso de UnmanagedType.LPUTF8Str requer o .NET Framework 4.7 (ou versões posteriores) ou o .NET Core 1.1 (ou versões posteriores). Não está disponível no .NET Standard 2.0.

Se você estiver trabalhando com APIs COM, talvez seja necessário realizar marshal de uma cadeia de caracteres como um `BSTR`. Usando o valor UnmanagedType.BStr, você pode realizar marshal de uma cadeia de caracteres como um `BSTR`.

```
public struct BString
{
    [MarshalAs(UnmanagedType.BStr)]
    public string str;
}
```

```
struct BString
{
    BSTR str;
};
```

Ao usar uma API baseada no WinRT, talvez seja necessário realizar marshal de uma cadeia de caracteres como um `HSTRING`. Usando o valor UnmanagedType.HString, você pode realizar marshal de uma cadeia de caracteres como um `HSTRING`.

```
public struct HString
{
    [MarshalAs(UnmanagedType.HString)]
    public string str;
}
```

```
struct BString
{
    HSTRING str;
};
```

Se sua API requer que você passe a cadeia de caracteres in-loco na estrutura, use o valor UnmanagedType.ByValTStr. A codificação de uma cadeia de caracteres com marshaling realizado por `ByValTStr` é determinada por meio do atributo `CharSet`. Além disso, ela requer que o comprimento da cadeia de caracteres seja passado pelo campo MarshalAsAttribute.SizeConst.

```
[StructLayout(LayoutKind.Sequential, CharSet = CharSet.Ansi)]
public struct DefaultString
{
    [MarshalAs(UnmanagedType.ByValTStr, SizeConst = 4)]
    public string str;
}
```

```
struct DefaultString
{
    char str[4];
};
```

```
[StructLayout(LayoutKind.Sequential, CharSet = CharSet.Unicode)]
public struct DefaultString
{
    [MarshalAs(UnmanagedType.ByValTStr, SizeConst = 4)]
    public string str;
}
```

```
struct DefaultString
{
    char16_t str[4]; // Could also be wchar_t[4] on Windows.
};
```

Personalização do marshaling de campo decimal

Se você estiver trabalhando no Windows, poderá encontrar algumas APIs que usam a estrutura `CY` ou `CURRENCY` nativa. Por padrão, o tipo .NET `decimal` realizar marshal da estrutura nativa `DECIMAL`. No entanto, você pode usar um MarshalAsAttribute com o valor UnmanagedType.Currency para instruir o marshaler para converter um valor `decimal` para um valor nativo `CY`.

```
public struct Currency
{
    [MarshalAs(UnmanagedType.Currency)]
    public decimal dec;
}
```

```
struct Currency
{
    CY dec;
};
```

`System.Object`s de marshaling

No Windows, você pode realizar marshal dos campos tipados de `object` para o código nativo. Você pode realizar marshal desses campos para um dos três tipos:

- `VARIANT`
- `IUnknown*`
- `IDispatch*`

Por padrão, um campo tipado de `object` terá o marshaling realizado para um `IUnknown*` que encapsula o objeto.

```
public struct ObjectDefault
{
    public object obj;
}
```

```
struct ObjectDefault
{
    IUnknown* obj;
};
```

Se você quiser realizar marshal de um campo de objeto para um `IDispatch*`, adicione um MarshalAsAttribute com o valor UnmanagedType.IDispatch.

```
public struct ObjectDispatch
{
    [MarshalAs(UnmanagedType.IDispatch)]
    public object obj;
}
```

```
struct ObjectDispatch
{
    IDispatch* obj;
};
```

Se você quiser realizar marshal dele como um `VARIANT`, adicione um MarshalAsAttribute com o valor UnmanagedType.Struct.

```
public struct ObjectVariant
{
    [MarshalAs(UnmanagedType.Struct)]
    public object obj;
}
```

```
struct ObjectVariant
{
    VARIANT obj;
};
```

A tabela a seguir descreve como diferentes tipos de tempo de execução do campo `obj` são mapeados para os vários tipos armazenados em um `VARIANT`:

TIPO .NET	TIPO DE VARIANTE		TIPO .NET	TIPO DE VARIANTE
`byte`	`VT_UI1`		`System.Runtime.InteropS`	`VT_BSTR` `BStrWrapper`
`sbyte`	`VT_I1`		`object`	`VT_DISPATCH`
`short`	`VT_I2`		`System.Runtime.InteropS`	`VT_UNKNOWN` `nownWrapper`
`ushort`	`VT_UI2`		`System.Runtime.InteropS`	`VT_DISPATCH` `atchWrapper`
`int`	`VT_I4`		`System.Reflection.Missi`	`VT_ERROR`
`uint`	`VT_UI4`		`(object)null`	`VT_EMPTY`
`long`	`VT_I8`		`bool`	`VT_BOOL`
`ulong`	`VT_UI8`		`System.DateTime`	`VT_DATE`
`float`	`VT_R4`		`decimal`	`VT_DECIMAL`
`double`	`VT_R8`		`System.Runtime.InteropS`	`VT_CURRENCY` `encyWrapper`
`char`	`VT_UI2`		`System.DBNull`	`VT_NULL`
`string`	`VT_BSTR`			

Como personalizar o marshaling de parâmetro

31/10/2019 • 5 minutes to read • Edit Online

Quando o comportamento de marshaling de parâmetro padrão do tempo de execução do .NET não fizer o que você deseja, use o atributo System.Runtime.InteropServices.MarshalAsAttribute para personalizar o modo como seus parâmetros passam pelo processo de marshaling.

Personalizar parâmetros de cadeias de caracteres

O .NET tem uma variedade de formatos para a realização de marshaling de cadeias de caracteres. Esses métodos são divididos em seções distintas em strings no C-style e em formatos de cadeias de caracteres centrados no Windows.

Cadeias de caracteres C-style

Cada um desses formatos passa uma cadeia de caracteres terminada em nulo para o código nativo. Eles diferem pela codificação da cadeia de caracteres nativa.

VALOR `SYSTEM.RUNTIME.INTEROPSERVICES.UNMANAGEDTYPE`	CODIFICANDO
LPStr	ANSI
LPUTF8Str	UTF-8
LPWStr	UTF-16
LPTStr	UTF-16

O formato UnmanagedType.VBByRefStr é um pouco diferente. Como `LPWStr`, ele realiza marshalling da cadeia de caracteres para uma cadeia de caracteres C-style nativa codificada em UTF-16. No entanto, a assinatura gerenciada passa a cadeia de caracteres por referência e a assinatura nativa correspondente usa a cadeia de caracteres por valor. Essa distinção permite que você use uma API nativa que recebe uma cadeia de caracteres por valor e a modifica no local sem precisar usar um `StringBuilder`. Recomendamos que você não use esse formato manualmente, pois ele pode causar confusão com as assinaturas nativas e gerenciadas incompatíveis.

Formatos de cadeias de caracteres centradas no Windows

Ao interagir com interfaces COM ou OLE, você provavelmente descobrirá que as funções nativas tomam cadeias de caracteres como argumentos `BSTR`. Você pode usar o tipo não gerenciado UnmanagedType.BStr para realizar marshalling de uma cadeia de caracteres como um `BSTR`.

Se você estiver interagindo com as APIs do WinRT, poderá usar o formato UnmanagedType.HString para realizar marshalling de uma cadeia de caracteres como `HSTRING`.

Personalizar parâmetros de matriz

.NET também fornece várias maneiras de realizar marshalling de parâmetros de matriz. Se você estiver chamando uma API que usa uma matriz C-style, use o tipo não gerenciado UnmanagedType.LPArray. Se os valores na matriz precisarem de marshaling personalizado, você poderá usar o campo ArraySubType no atributo `[MarshalAs]` para isso.

Se você estiver usando APIs COM, provavelmente terá que organizar seus parâmetros de matriz como

`SAFEARRAY*`s. Para fazer isso, você pode usar o tipo não gerenciado UnmanagedType.SafeArray. O tipo padrão dos elementos do `SAFEARRAY` pode ser visto na tabela sobre como personalizar campos `object`. Você pode usar os campos MarshalAsAttribute.SafeArraySubType e MarshalAsAttribute.SafeArrayUserDefinedSubType para personalizar o tipo de elemento exato do `SAFEARRAY`.

Personalizar parâmetros boolianos ou decimais

Para saber mais sobre como realizar marshaling de parâmetros boolianos ou decimais, consulte Como personalizar o marshaling de estrutura.

Personalizar parâmetros de objeto (somente Windows)

No Windows, o tempo de execução do .NET fornece várias maneiras diferentes de realizar marshalling de parâmetros de objeto para código nativo.

Marshaling como interfaces COM específicas

Se sua API leva um ponteiro para um objeto COM, você pode usar qualquer um dos seguintes formatos `UnmanagedType` em um parâmetro de tipo `object` para informar o .NET para realizar marshalling como essas interfaces específicas:

- `IUnknown`
- `IDispatch`
- `IInspectable`

Além disso, se o seu tipo estiver marcado com `[ComVisible(true)]` ou se você estiver organizando o tipo `object`, poderá usar o formato UnmanagedType.Interface para realizar marshaling do seu objeto como um COM callable wrapper para a exibição COM do seu tipo.

Realizar marshaling para um `VARIANT`

Se sua API nativa tiver uma `VARIANT` Win32, você poderá usar o formato UnmanagedType.Struct no seu `object` para organizar seus objetos como `VARIANT`s. Consulte a documentação sobre como personalizar campos `object` para um mapeamento entre tipos .NET e tipos `VARIANT`.

Empacotadores personalizados

Se você desejar projetar uma interface COM nativa em um tipo gerenciado diferente, poderá usar o formato `UnmanagedType.CustomMarshaler` e uma implementação de ICustomMarshaler para fornecer seu próprio código de marshaling personalizado.

Práticas recomendadas de interoperabilidade nativa

23/10/2019 • 19 minutes to read • Edit Online

.NET oferece uma variedade maneiras de personalizar seu código de interoperabilidade nativa. Este artigo inclui as diretrizes que as equipes .NET da Microsoft seguem para a interoperabilidade nativa.

Diretrizes gerais

As diretrizes nesta seção se aplicam a todos os cenários de interoperabilidade.

- ✔️ **USE** a mesma nomenclatura e uso de maiúsculas para seus métodos e parâmetros como o método nativo que você deseja chamar.
- ✔️ **CONSIDERE** usar a mesma nomenclatura e uso de maiúsculas para valores constantes.
- ✔️ **USE** tipos .NET com mapeamento mais próximo do tipo nativo. Por exemplo, no caso de C#, use `uint` quando o tipo nativo for `unsigned int`.
- ✔️ **USE** os atributos `[In]` e `[Out]` somente quando o comportamento desejado for diferente do comportamento padrão.
- ✔️ **CONSIDERE** usar System.Buffers.ArrayPool<T> para agrupar seus buffers de matriz nativos.
- ✔️ **CONSIDERE** encapsular suas declarações P/Invoke em uma classe com o mesmo nome e letras maiúsculas como sua biblioteca nativa.
  - Isso permite que seus atributos `[DllImport]` usem o recurso de linguagem C# `nameof` para passar o nome da biblioteca nativa e garantir que você não tenha digitado errado o nome da biblioteca nativa.

Configurações de atributo DllImport

CONFIGURAÇÃO	PADRÃO	RECOMENDAÇÃO	DETALHES
PreserveSig	`true`	manter padrão	Quando esta configuração é definida como false, valores de retorno HRESULT com falha serão considerados exceções (e o valor de retorno na definição torna-se nulo).
SetLastError	`false`	depende da API	Defina esta configuração como true se a API usa GetLastError e usa Marshal.GetLastWin32Error para obter o valor. Se a API definir uma condição que informa um erro, obtenha o erro antes de fazer outras chamadas para evitar que ele seja sobrescrito inadvertidamente.

CONFIGURAÇÃO	PADRÃO	RECOMENDAÇÃO	DETALHES
CharSet	`CharSet.None`, que reverte para o comportamento `CharSet.Ansi`	Use explicitamente `CharSet.Unicode` ou `CharSet.Ansi` quando os caracteres ou cadeias de caracteres estiverem presentes na definição	Isso especifica o comportamento de marshaling de cadeias de caracteres e o que `ExactSpelling` faz quando `false`. Note que `CharSet.Ansi` é na verdade UTF8 no Unix. O Windows usa Unicode a *maior parte* do tempo, enquanto o Unix usa UTF8. Veja mais informações na documentação sobre conjuntos de caracteres.
ExactSpelling	`false`	`true`	Defina como true e obtenha um pequeno benefício de desempenho: o tempo de execução não irá buscar por nomes de função alternativos com o sufixo "A" ou "W" dependendo do valor da configuração `CharSet` ("A" para `CharSet.Ansi` e "W" para `CharSet.Unicode`).

Parâmetros de cadeia de caracteres

Quando o CharSet é Unicode ou o argumento é explicitamente marcado como `[MarshalAs(UnmanagedType.LPWSTR)]` *e* a cadeia de caracteres é passada por valor (não `ref` ou `out`), a cadeia de caracteres será fixada e usada diretamente pelo código nativo (em vez de copiado).

Lembre-se de marcar `[DllImport]` como `Charset.Unicode`, a menos que você queira explicitamente o tratamento ANSI de suas cadeias de caracteres.

 NÃO use parâmetros `[Out] string`. Os parâmetros de cadeia de caracteres passados por valor com o atributo `[Out]` podem desestabilizar o tempo de execução se a cadeia de caracteres for uma cadeia de caracteres internada. Veja mais informações sobre a centralização da cadeia de caracteres na documentação do String.Intern.

 EVITE parâmetros `StringBuilder`. Marshaling de `StringBuilder` *sempre* cria uma cópia do buffer nativo. Dessa forma, ele pode ser extremamente ineficiente. Veja o cenário típico da chamada de uma API do Windows que usa uma cadeia de caracteres:

1. Criar um SB da capacidade desejada (aloca capacidade gerenciada) **{1}**
2. Chamar
   a. Aloca um buffer nativo **{2}**
   b. Copia o conteúdo se `[In]` *(o padrão para um parâmetro* `StringBuilder`*)*
   c. Copia o buffer nativo em uma matriz gerenciada recém-alocada se `[Out]` **{3}** *(também é o padrão para* `StringBuilder`*)*
3. `ToString()` aloca outra matriz gerenciada **{4}**

Ou seja, *{4}* alocações para obter uma cadeia de caracteres fora do código nativo. O melhor que você pode fazer para limitar isso é reutilizar o `StringBuilder` em outra chamada, mas isso economiza apenas *1* alocação. É muito melhor usar e armazenar em cache um buffer de caractere de `ArrayPool` - você pode então reduzir para apenas a alocação para `ToString()` nas chamadas subsequentes.

O outro problema com `StringBuilder` é que esta configuração sempre copia o buffer de retorno de volta para o

primeiro nulo. Se a cadeia de caracteres transmitida não estiver terminada, ou terminar por dois caracteres nulos, na melhor das hipóteses, o recurso P/Invoke estará incorreto.

Se você *usar* o `StringBuilder`, uma última pegadinha é que a capacidade **não** inclui um nulo oculto, que é sempre contabilizado na interoperabilidade. É comum as pessoas entenderem errado, já que a maioria das APIs deseja o tamanho do buffer, *incluindo* o valor nulo. Isso pode resultar em alocações desnecessárias/desperdiçadas. Além disso, esse problema impede que o tempo de execução otimize o marshaling de `StringBuilder` para minimizar as cópias.

✔ **CONSIDERE** usar `char[]`s de um `ArrayPool`.

Para obter mais informações sobre o marshaling de cadeia de caracteres, veja Marshaling padrão para cadeias de caracteres e Personalizando marshaling de cadeia de caracteres.

> **Específico do Windows**
> Para cadeias de caracteres `[Out]`, a CLR usará `CoTaskMemFree` por padrão para liberar cadeias de caracteres, ou `SysStringFree` para cadeias de caracteres que são marcadas como `UnmanagedType.BSTR`.
> **Para a maioria das APIs com um buffer de cadeia de caracteres de saída:**
> A contagem de caracteres transmitidos deve incluir o nulo. Se o valor retornado for menor que a contagem de caracteres transmitidos, a chamada foi bem-sucedida e o valor consiste no número de caracteres *sem* o nulo à direita. Caso contrário, a contagem consiste no tamanho necessário do buffer *incluindo* o caractere nulo.
>
> - Passe cinco, obtenha quatro: A cadeia de caracteres tem quatro caracteres e um nulo à direita.
> - Passe cinco, obtenha seis: A cadeia de caracteres tem cinco caracteres, precisa de um buffer de seis caracteres para manter o valor nulo.
>
> Tipos de dados do Windows para cadeias de caracteres

Parâmetros e campos boolianos

É fácil cometer erros com boolianos. Por padrão, o marshaling de um `bool` .NET é realizado para um Windows `BOOL`, em que é um valor de 4 bytes. No entanto, os tipos `_Bool` e `bool` em C e C++ são um byte *único*. Isso pode dificultar o rastreamento de bugs, já que metade do valor de retorno será descartado, o que só *potencialmente* alterará o resultado. Para obter mais informações sobre o marshaling de valores do `bool` .NET para tipos `bool` C ou C++, veja a documentação sobre como personalizar marshaling de campo booliano.

GUIDs

Os GUIDs podem ser usados diretamente em assinaturas. Muitas APIs do Windows usam aliases do tipo `GUID&` como `REFIID`. Quando passadas por ref, elas podem ser passadas por `ref` ou pelo atributo `[MarshalAs(UnmanagedType.LPStruct)]`.

GUID	GUID BY-REF
`KNOWNFOLDERID`	`REFKNOWNFOLDERID`

NÃO use `[MarshalAs(UnmanagedType.LPStruct)]` para qualquer coisa diferente de parâmetros GUID `ref`.

Tipos blittable

Os tipos blittable são tipos que têm a mesma representação em nível de bits no código gerenciado e nativo. Como tal, eles não precisam ser convertidos em outro formato para serem empacotados para e do código nativo e, uma vez que isso melhora o desempenho, eles devem ter preferência.

Tipos blittable:

- `byte`, `sbyte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `single`, `double`
- matrizes unidimensionais não aninhadas de tipos blittable (por exemplo, `int[]`)

- estruturas e classes com layout fixo que só têm tipos de valor blittable para campos de instância
  - layout fixo requer `[StructLayout(LayoutKind.Sequential)]` ou `[StructLayout(LayoutKind.Explicit)]`
  - structs são `LayoutKind.Sequential` por padrão, classes são `LayoutKind.Auto`

NÃO blittable:

- `bool`

ÀS VEZES blittable:

- `char`, `string`

Quando tipos blittable são passados por referência, eles são simplesmente fixados pelo marshaller em vez de serem copiados para um buffer intermediário. (As classes são inerentemente passadas por referência, structs são passadas por referência quando usadas com `ref` ou `out`).

`char` é blittable em uma matriz unidimensional **ou** se for parte de um tipo que é explicitamente marcado com `[StructLayout]` com `CharSet = CharSet.Unicode`.

```
[StructLayout(LayoutKind.Sequential, CharSet = CharSet.Unicode)]
public struct UnicodeCharStruct
{
    public char c;
}
```

`string` é blittable se não está contido em outro tipo e está sendo passado como um argumento marcado com `[MarshalAs(UnmanagedType.LPWStr)]` ou o `[DllImport]` tem `CharSet = CharSet.Unicode` definido.

Você pode verificar se um tipo é blittable pela tentativa de criar um `GCHandle` fixado. Se o tipo não for uma cadeia de caracteres ou considerado blittable, `GCHandle.Alloc` lançará um `ArgumentException`.

✔ **TORNE** suas estruturas mais blittable quando possível.

Para obter mais informações, consulte:

- Tipos blittable e não blittable
- Marshaling de Tipo

Manter objetos gerenciados ativos

`GC.KeepAlive()` garantirá que um objeto permaneça no escopo até que o método KeepAlive seja alcançado.

`HandleRef` permite ao marshaller manter um objeto ativo pela duração de um P/Invoke. Ele pode ser usado em vez de `IntPtr` em assinaturas de métodos. `SafeHandle` substitui efetivamente essa classe e deve ser usado em seu lugar.

`GCHandle` permite fixar um objeto gerenciado e obter o ponteiro nativo para ele. O padrão básico é:

```
GCHandle handle = GCHandle.Alloc(obj, GCHandleType.Pinned);
IntPtr ptr = handle.AddrOfPinnedObject();
handle.Free();
```

Fixar não é o padrão para `GCHandle`. O outro padrão principal é passar uma referência a um objeto gerenciado por meio do código nativo e voltar ao código gerenciado, geralmente com um retorno de chamada. Aqui está o padrão:

```
GCHandle handle = GCHandle.Alloc(obj);
SomeNativeEnumerator(callbackDelegate, GCHandle.ToIntPtr(handle));

// In the callback
GCHandle handle = GCHandle.FromIntPtr(param);
object managedObject = handle.Target;

// After the last callback
handle.Free();
```

Não se esqueça que `GCHandle` precisa ser explicitamente liberado para evitar perda de memória.

Tipos de dados comuns do Windows

Veja a seguir uma lista dos tipos de dados comumente usados em APIs do Windows e quais tipos C# devem ser usados ao chamar o código do Windows.

Os tipos a seguir são do mesmo tamanho no Windows de 32 e 64 bits, apesar de seus nomes.

LARGURA	WINDOWS	C (WINDOWS)	C#	ALTERNATIVA
32	`BOOL`	`int`	`int`	`bool`
8	`BOOLEAN`	`unsigned char`	`byte`	`[MarshalAs(UnmanagedType.U1` `bool`
8	`BYTE`	`unsigned char`	`byte`	
8	`CHAR`	`char`	`sbyte`	
8	`UCHAR`	`unsigned char`	`byte`	
16	`SHORT`	`short`	`short`	
16	`CSHORT`	`short`	`short`	
16	`USHORT`	`unsigned short`	`ushort`	
16	`WORD`	`unsigned short`	`ushort`	
16	`ATOM`	`unsigned short`	`ushort`	
32	`INT`	`int`	`int`	
32	`LONG`	`long`	`int`	
32	`ULONG`	`unsigned long`	`uint`	
32	`DWORD`	`unsigned long`	`uint`	
64	`QWORD`	`long long`	`long`	
64	`LARGE_INTEGER`	`long long`	`long`	
64	`LONGLONG`	`long long`	`long`	

LARGURA	WINDOWS	C (WINDOWS)	C#	ALTERNATIVA
64	`ULONGLONG`	`unsigned long long`	`ulong`	
64	`ULARGE_INTEGER`	`unsigned long long`	`ulong`	
32	`HRESULT`	`long`	`int`	
32	`NTSTATUS`	`long`	`int`	

Os tipos a seguir, sendo ponteiros, seguem a largura da plataforma. Use `IntPtr` / `UIntPtr` para eles.

TIPOS DE PONTEIROS ASSINADOS (USE `INTPTR`)	TIPOS DE PONTEIROS NÃO ASSINADOS (USE `UINTPTR`)
`HANDLE`	`WPARAM`
`HWND`	`UINT_PTR`
`HINSTANCE`	`ULONG_PTR`
`LPARAM`	`SIZE_T`
`LRESULT`	
`LONG_PTR`	
`INT_PTR`	

Um `PVOID` Windows que é um `void*` C pode passar por marshalling como `IntPtr` ou `UIntPtr`, mas prefere `void*` quando possível.

Tipos de dados do Windows

Intervalos de tipos de dados

Structs

As structs gerenciadas são criadas e não são removidas até o método retornar. Por definição, elas são "fixadas" (não serão movidas pelo GC). Você também pode simplesmente pegar o endereço em blocos de código não seguros se o código nativo não usar o ponteiro após o final do método atual.

Structs blittable são muito mais eficazes, pois simplesmente podem ser usados de modo direto pela camada de marshaling. Tente tornar structs blittable (por exemplo, evite `bool`). Para saber mais, veja a seção Tipos blittable.

Se a struct é blittable, use `sizeof()` em vez de `Marshal.SizeOf<MyStruct>()` para melhor desempenho. Como mencionado acima, você pode validar que o tipo é blittable ao tentar criar um `GCHandle` fixado. Se o tipo não for uma cadeia de caracteres ou considerado blittable, `GCHandle.Alloc` lançara `ArgumentException`.

Os ponteiros para structs nas definições devem ser transmitidos por `ref` ou usar `unsafe` e `*`.

✔️ **BUSQUE** a struct gerenciada o mais próximo possível da forma e dos nomes usados na documentação ou no cabeçalho da plataforma oficial.

✔️ **USE** `sizeof()` C# em vez de `Marshal.SizeOf<MyStruct>()` para estruturas blittable a fim de melhorar o desempenho.

Uma matriz como `INT_PTR Reserved1[2]` precisa ser empacotada para dois campos `IntPtr`, `Reserved1a` e `Reserved1b`. Quando a matriz nativa é um tipo primitivo, podemos usar a palavra-chave `fixed` para escrevê-la um pouco mais limpa. Por exemplo, `SYSTEM_PROCESS_INFORMATION` se parece com isso no cabeçalho nativo:

```
typedef struct _SYSTEM_PROCESS_INFORMATION {
    ULONG NextEntryOffset;
    ULONG NumberOfThreads;
    BYTE Reserved1[48];
    UNICODE_STRING ImageName;
...
} SYSTEM_PROCESS_INFORMATION
```

Em C#, podemos escrevê-lo assim:

```
internal unsafe struct SYSTEM_PROCESS_INFORMATION
{
    internal uint NextEntryOffset;
    internal uint NumberOfThreads;
    private fixed byte Reserved1[48];
    internal Interop.UNICODE_STRING ImageName;
    ...
}
```

No entanto, existem algumas armadilhas com buffers fixos. Buffers fixos de tipos não blittable não serão empacotados corretamente, portanto, a matriz no local precisa ser expandida para vários campos individuais. Além disso, no .NET Framework e no .NET Core antes da versão 3.0, se um struct contendo um campo de buffer fixo for aninhado dentro de um struct não blittable, não será realizado o marshaling correto do campo de buffer fixo para código nativo.

Conjuntos de caracteres e marshaling

23/10/2019 • 2 minutes to read • Edit Online

A maneira como valores `char`, objetos `string` e objetos `System.Text.StringBuilder` são empacotados depende do valor do campo `CharSet` em P/Invoke ou na estrutura. Para definir o `CharSet` de um P/Invoke, configure o campo DllImportAttribute.CharSet quando declarar o P/Invoke. Para definir o `CharSet` para um tipo, defina o campo StructLayoutAttribute.CharSet na sua declaração de classe ou struct. Se esses campos do atributo não estiverem definidos, caberá ao compilador de linguagem determinar qual `CharSet` a ser usado. O C# e o Visual Basic usam o conjunto de caracteres Ansi por padrão.

A tabela a seguir mostra um mapeamento entre cada conjunto de caracteres e como um caractere ou uma cadeia de caracteres é representado quando o é realizado seu marshaling com esse conjunto de caracteres:

VALOR `CHARSET`	WINDOWS	.NET CORE 2.2 E ANTERIORES NO UNIX	.NET CORE 3.0 E POSTERIORES E MONO NO UNIX
Ansi	`char` (a página de códigos do Windows (ANSI) padrão do sistema)	`char` (UTF-8)	`char` (UTF-8)
Unicode	`wchar_t` (UTF-16)	`char16_t` (UTF-16)	`char16_t` (UTF-16)
Automático	`wchar_t` (UTF-16)	`char16_t` (UTF-16)	`char` (UTF-8)

Quando escolher o conjunto de caracteres, saiba qual será a representação esperada por sua representação nativa.

Interoperabilidade COM no .NET

23/10/2019 • 2 minutes to read • Edit Online

O COM (Component Object Model) permite que um objeto exponha sua funcionalidade a outros componentes e aplicativos host em plataformas Windows. Para ajudar a permitir que os usuários executem a interoperação com as bases de código existentes, o .NET Framework sempre forneceu um forte suporte para interoperação com bibliotecas COM. No .NET Core 3.0, uma grande parte desse suporte foi adicionada ao .NET Core no Windows. Esta documentação explicará como as tecnologias comuns de interoperabilidade COM funcionam e como você pode utilizá-las para executar a interoperação com as bibliotecas COM existentes.

- Wrappers COM
- COM Callable Wrappers
- RCWs (Runtime Callable Wrappers)
- Tipos .NET qualificados para interoperação COM

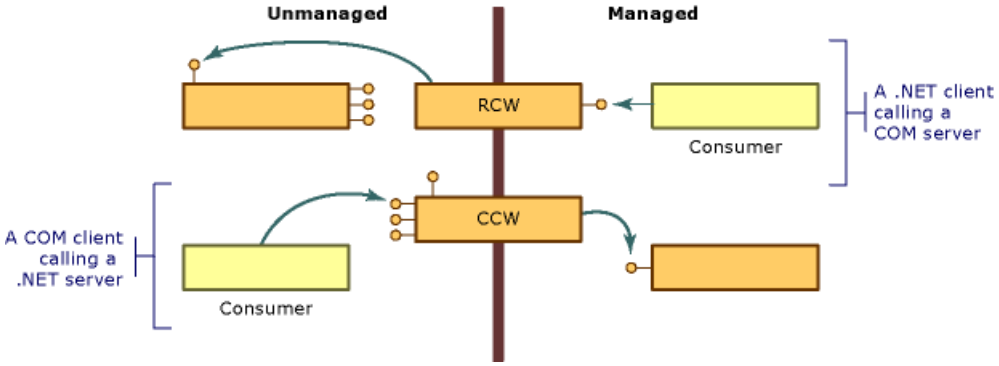
Wrappers COM

31/10/2019 • 3 minutes to read • Edit Online

O COM difere do modelo de objeto de runtime do .NET de várias maneiras importantes:

- Os clientes de objetos COM devem gerenciar o tempo de vida desses objetos; o Common Language Runtime gerencia o tempo de vida de objetos em seu ambiente.
- Os clientes de objetos COM descobrem se um serviço está disponível solicitando uma interface que fornece esse serviço e recebendo um ponteiro de interface ou não. Os clientes de objetos .NET podem obter uma descrição da funcionalidade de um objeto usando a reflexão.
- Os objetos .NET residem na memória gerenciada pelo ambiente de execução do runtime do .NET. O ambiente de execução pode mover objetos na memória por motivos de desempenho e atualizar todas as referências nos objetos movidos por ele. Os clientes não gerenciados, depois de obter um ponteiro para um objeto, dependem do objeto para permanecerem no mesmo local. Esses clientes não têm nenhum mecanismo para lidar com um objeto cujo local não é fixo.

Para superar essas diferenças, o runtime fornece classes wrapper para fazer com que os clientes gerenciados e não gerenciados acreditem que estão chamando objetos em seus respectivos ambientes. Sempre que o cliente gerenciado chama um método em um objeto COM, o tempo de execução cria um RCW (Runtime Callable Wrapper). Os RCWs eliminam as diferenças entre os mecanismos de referência gerenciada e não gerenciada, entre outras coisas. O tempo de execução também cria um CCW (COM Callable Wrapper) para reverter o processo, permitindo que um cliente COM chame um método em um objeto .NET diretamente. Como mostra a ilustração a seguir, a perspectiva do código de chamada determina qual classe wrapper é criada pelo runtime.



Na maioria dos casos, o RCW (Runtime Callable Wrapper) padrão ou o CCW gerado pelo tempo de execução fornece o marshaling adequado para chamadas que cruzam o limite entre o COM e o tempo de execução do .NET. Usando atributos personalizados, opcionalmente, você pode ajustar a maneira como o runtime representa o código gerenciado e não gerenciado.

Consulte também

- Interoperabilidade COM avançada no .NET Framework
- RCW (Runtime Callable Wrapper)
- COM Callable Wrapper
- Como personalizar wrappers padrão no .NET Framework
- Como personalizar wrappers callable em tempo de execução no .NET Framework

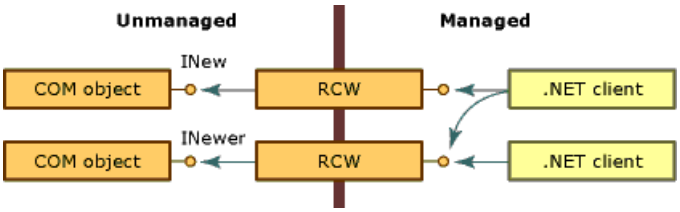
RCW (Runtime Callable Wrapper)

31/10/2019 • 7 minutes to read • Edit Online

O common language runtime expõe objetos COM através de um proxy chamado RCW (Runtime Callable Wrapper). Embora o RCW pareça ser um objeto comum para clientes .NET, a função principal dele é realizar marshaling de chamadas entre um cliente .NET e um objeto COM.

O runtime cria exatamente um RCW para cada objeto COM, independentemente do número de referências que existem nesse objeto. O runtime mantém um único RCW por processo para cada objeto. Se você criar um RCW em um domínio de aplicativo ou apartment e, em seguida, passar uma referência a outro domínio de aplicativo ou apartment, um proxy para o primeiro objeto será usado. Conforme mostra a ilustração a seguir, qualquer número de clientes pode conter uma referência a objetos COM que expõem interfaces INew e INewer.

A seguinte imagem mostra o processo para acessar objetos COM por meio do RCW (Runtime Callable Wrapper):



Usando metadados derivados de uma biblioteca de tipos, o runtime cria o objeto COM que está sendo chamado e um wrapper para esse objeto. Cada RCW mantém um cache de ponteiros de interface no objeto COM que ele encapsula e, além disso, libera sua referência no objeto COM quando o RCW não é mais necessário. O runtime executa a coleta de lixo no RCW.

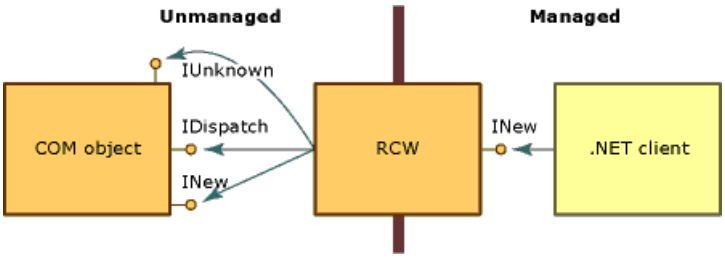
Entre outras atividades, o RCW realiza marshaling de dados entre código gerenciado e não gerenciado, em nome do objeto encapsulado. Especificamente, o RCW fornece marshaling para argumentos de método e valores retornados do método sempre que o cliente e o servidor têm representações diferentes dos dados transmitidos entre eles.

O wrapper padrão impõe regras de marshaling internas. Por exemplo, quando um cliente .NET passa um tipo de cadeia de caracteres como parte de um argumento para um objeto não gerenciado, o wrapper converte a cadeia de caracteres em um tipo BSTR. Se o objeto COM retornar um BSTR ao chamador gerenciado, o chamador receberá uma cadeia de caracteres. Tanto o cliente quanto o servidor enviam e recebem dados com os quais estão familiarizados. Outros tipos não exigem conversão. Por exemplo, um wrapper padrão sempre passará um inteiro de 4 bytes entre código gerenciado e não gerenciado sem converter o tipo.

Marshaling de interfaces selecionadas

A meta principal do RCW (Runtime Callable Wrapper) é ocultar as diferenças entre os modelos de programação gerenciado e não gerenciado. Para criar uma transição suave, o RCW consome interfaces COM selecionadas sem expô-las ao cliente .NET, conforme mostrado na ilustração a seguir.

A seguinte imagem mostra as interfaces COM e o RCW (Runtime Callable Wrapper):



Quando criado como um objeto associado precocemente, o RCW é um tipo específico. Ele implementa as interfaces que o objeto COM implementa e expõe os métodos, propriedades e eventos das interfaces do objeto. Na ilustração, o RCW expõe a interface INew mas consome as interfaces **IUnknown** e **IDispatch**. Além disso, o RCW expõe todos os membros da interface INew para o cliente .NET.

O RCW consome as interfaces listadas na tabela a seguir, as quais são expostas pelo objeto que ele encapsula.

INTERFACE	DESCRIÇÃO
IDispatch	Para associação tardia a objetos COM por meio de reflexão.
IErrorInfo	Fornece uma descrição textual do erro, sua origem, um arquivo de Ajuda, um contexto de Ajuda e o GUID da interface que definiu o erro (sempre **GUID_NULL** para classes do .NET).
IProvideClassInfo	Se o objeto COM sendo empacotado implementa **IProvideClassInfo**, o RCW extrai as informações de tipo dessa interface para fornecer uma melhor identidade de tipo.
IUnknown	Para gerenciamento do tempo de vida, coerção de tipo e identidade de objeto: – Identidade de objeto O tempo de execução faz distinção entre os objetos COM, comparando o valor da interface **IUnknown** para cada objeto. – Coerção de tipo O RCW reconhece a descoberta de tipo dinâmico executada pelo método **QueryInterface**. – Gerenciamento do tempo de vida Usando o **QueryInterface** método RCW obtém e mantém uma referência a um objeto não gerenciado até o tempo de execução executar a coleta de lixo no wrapper, que libera o objeto não gerenciado.

O RCW, opcionalmente, consome as interfaces listadas na tabela a seguir, as quais são expostas pelo objeto que ele encapsula.

INTERFACE	DESCRIÇÃO
IConnectionPoint e **IConnectionPointContainer**	O RCW converte objetos que expõem o estilo do evento de ponto de conexão para eventos com base em delegado.
IDispatchEx (somente .NET Framework)	Se a classe implementa **IDispatchEx**, o RCW implementa **IExpando**. A interface **IDispatchEx** é uma extensão da interface **IDispatch** que, ao contrário de **IDispatch**, permite a enumeração, adição, exclusão e chamada de membros que diferencia maiúsculas de minúsculas.

INTERFACE	DESCRIÇÃO
IEnumVARIANT	Permite que os tipos COM que dão suporte a enumerações sejam tratados como coleções.

Consulte também

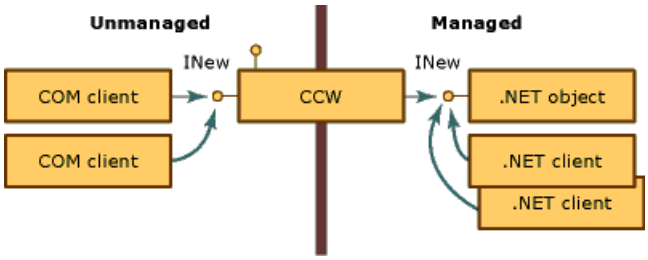
- Wrappers COM
- COM Callable Wrapper
- Resumo da conversão de bibliotecas de tipos em assemblies
- Importando uma biblioteca de tipos como um assembly

COM Callable Wrapper

31/10/2019 • 17 minutes to read • Edit Online

Quando um cliente COM chama um objeto .NET, o Common Language Runtime cria o objeto gerenciado e um CCW (COM Callable Wrapper) para o objeto. Se não é possível referenciar um objeto .NET diretamente, os clientes COM usam o CCW como um proxy do objeto gerenciado.

O runtime cria exatamente um CCW para um objeto gerenciado, independentemente do número de clientes COM que solicita seus serviços. Como mostra a ilustração a seguir, vários clientes COM podem conter uma referência ao CCW que expõe a interface INew. O CCW, por sua vez, contém uma única referência ao objeto gerenciado que implementa a interface e é coletada como lixo. Os clientes COM e .NET podem fazer solicitações no mesmo objeto gerenciado simultaneamente.



Os COM Callable Wrappers são invisíveis para outras classes em execução no runtime do .NET. Sua finalidade principal é realizar marshaling de chamadas entre o código gerenciado e não gerenciado; no entanto, os CCWs também gerenciam a identidade e o tempo de vida dos objetos gerenciados encapsulados por eles.

Identidade do objeto

O runtime aloca memória para o objeto .NET em seu heap coletado como lixo, que permite ao runtime mover o objeto na memória, conforme necessário. Por outro lado, o runtime aloca memória para o CCW em um heap não coletado, possibilitando que os clientes COM referenciem o wrapper diretamente.

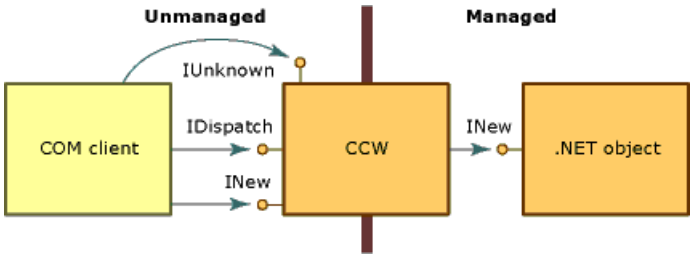
Tempo de vida do objeto

Ao contrário do cliente .NET encapsulado por ele, o CCW é contado por referência no modo tradicional do COM. Quando a contagem de referência do CCW chega a zero, o wrapper libera sua referência no objeto gerenciado. Um objeto gerenciado sem referências restantes é coletado durante o próximo ciclo de coleta de lixo.

Simulando Interfaces COM

O CCW expõe todas as interfaces públicas visíveis para o COM, os tipos de dados e os valores retornados para clientes COM de uma maneira consistente com a exigência do COM em relação à interação baseada em interface. Para um cliente COM, a invocação de métodos em um objeto .NET é idêntica à invocação de métodos em um objeto COM.

Para criar essa abordagem direta, o CCW fabrica interfaces COM tradicionais, como **IUnknown** e **IDispatch**. Como mostra a ilustração a seguir, o CCW mantém uma única referência no objeto .NET encapsulado por ele. O cliente COM e o objeto .NET interagem mutuamente por meio do proxy e da construção de stub do CCW.



Além de expor as interfaces que são implementadas explicitamente por uma classe no ambiente gerenciado, o runtime do .NET fornece implementações das interfaces COM listadas na tabela a seguir em nome do objeto. Uma classe .NET pode substituir o comportamento padrão fornecendo sua própria implementação dessas interfaces. No entanto, o tempo de execução sempre fornece a implementação para as interfaces **IUnknown** e **IDispatch**.

INTERFACE	DESCRIÇÃO
IDispatch	Fornece um mecanismo de associação tardia ao tipo.
IErrorInfo	Fornece uma descrição textual do erro, sua origem, um arquivo de Ajuda, um contexto de Ajuda e o GUID da interface que definiu o erro (sempre **GUID_NULL** para classes do .NET).
IProvideClassInfo	Permite aos clientes COM obter acesso à interface **ITypeInfo** implementada por uma classe gerenciada. Retorna `COR_E_NOTSUPPORTED` no .NET Core para tipos não importados do COM.
ISupportErrorInfo	Permite a um cliente COM determinar se o objeto gerenciado dá suporte à interface **IErrorInfo**. Nesse caso, permite ao cliente obter um ponteiro para o último objeto de exceção. Todos os tipos gerenciados dão suporte à interface **IErrorInfo**.
ITypeInfo (somente .NET Framework)	Fornece informações de tipo de uma classe que são exatamente iguais às informações de tipo produzidas pelo Tlbexp.exe.
IUnknown	Fornece a implementação padrão da interface **IUnknown** com a qual o cliente COM gerencia o tempo de vida do CCW e fornece a coerção de tipo.

Uma classe gerenciada também pode fornecer as interfaces COM descritas na tabela a seguir.

INTERFACE	DESCRIÇÃO
A interface de classe (*_classname*)	Interface, exposta pelo runtime e não definida explicitamente, que expõe todas as interfaces públicas, métodos, propriedades e campos que são expostos explicitamente em um objeto gerenciado.
IConnectionPoint e **IConnectionPointContainer**	Interface para objetos que dão origem a eventos baseados em representante (uma interface para o registro de assinantes do evento).

INTERFACE	DESCRIÇÃO
IDispatchEx (somente .NET Framework)	Interface fornecida pelo tempo de execução se a classe implementa **IExpando**. A interface **IDispatchEx** é uma extensão da interface **IDispatch** que, ao contrário de **IDispatch**, permite a enumeração, adição, exclusão e chamada de membros que diferencia maiúsculas de minúsculas.
IEnumVARIANT	Interface para classes de tipo de coleção, que enumera os objetos na coleção, se a classe implementa **IEnumerable**.

Introdução à interface de classe

A interface de classe, que não é definida explicitamente no código gerenciado, é uma interface que expõe todos os métodos públicos, propriedades, campos e eventos que são expostos explicitamente no objeto .NET. Essa interface pode ser uma interface dupla ou somente de expedição. A interface de classe recebe o nome da própria classe do .NET, precedido por um sublinhado. Por exemplo, para a classe Mammal, a interface de classe é _Mammal.

Para classes derivadas, a interface de classe também expõe todos os métodos públicos, propriedades e campos da classe base. A classe derivada também expõe uma interface de classe para cada classe base. Por exemplo, se a classe Mammal estende a classe MammalSuperclass, que, por sua vez, estende System.Object, o objeto do .NET expõe para os clientes COM três interfaces de classe chamadas _Mammal, _MammalSuperclass e _Object.

Por exemplo, considere a seguinte classe do .NET:

```
' Applies the ClassInterfaceAttribute to set the interface to dual.
<ClassInterface(ClassInterfaceType.AutoDual)> _
' Implicitly extends System.Object.
Public Class Mammal
    Sub Eat()
    Sub Breathe()
    Sub Sleep()
End Class
```

```
// Applies the ClassInterfaceAttribute to set the interface to dual.
[ClassInterface(ClassInterfaceType.AutoDual)]
// Implicitly extends System.Object.
public class Mammal
{
    public void Eat() {}
    public void Breathe() {}
    public void Sleep() {}
}
```

O cliente COM pode obter um ponteiro para uma interface de classe chamada `_Mammal`. No .NET Framework, você pode usar a ferramenta Exportador da Biblioteca de Tipos (Tlbexp.exe) para gerar uma biblioteca de tipos que contém a definição de interface `_Mammal`. Não há suporte para o Exportador da Biblioteca de Tipos no .NET Core. Se a classe `Mammal` tiver implementado uma ou mais interfaces, as interfaces aparecerão na coclass.

```
[odl, uuid(…), hidden, dual, nonextensible, oleautomation]
interface _Mammal : IDispatch
{
    [id(0x00000000), propget] HRESULT ToString([out, retval] BSTR*
        pRetVal);
    [id(0x60020001)] HRESULT Equals([in] VARIANT obj, [out, retval]
        VARIANT_BOOL* pRetVal);
    [id(0x60020002)] HRESULT GetHashCode([out, retval] short* pRetVal);
    [id(0x60020003)] HRESULT GetType([out, retval] _Type** pRetVal);
    [id(0x6002000d)] HRESULT Eat();
    [id(0x6002000e)] HRESULT Breathe();
    [id(0x6002000f)] HRESULT Sleep();
}
[uuid(…)]
coclass Mammal
{
    [default] interface _Mammal;
}
```

A geração da interface de classe é opcional. Por padrão, a interoperabilidade COM gera uma interface somente de expedição para cada classe exportada para uma biblioteca de tipos. É possível impedir ou modificar a criação automática dessa interface aplicando o ClassInterfaceAttribute à classe. Embora a interface de classe possa facilitar a tarefa de exposição das classes gerenciadas ao COM, seus usos são limitados.

Caution

O uso da interface de classe, em vez da definição explícita de sua própria, pode complicar o controle futuro de versão da classe gerenciada. Leia as diretrizes a seguir antes de usar a interface de classe.

Defina uma interface explícita para uso dos clientes COM, em vez de gerar a interface de classe.

Como a interoperabilidade COM gera uma interface de classe automaticamente, alterações posteriores à versão na classe podem alterar o layout da interface de classe exposta pelo Common Language Runtime. Já que os clientes COM não estão normalmente preparados para manipular as alterações no layout de uma interface, eles são interrompidos se o layout de membro da classe é alterado.

Esta diretriz reforça a noção de que as interfaces expostas para os clientes COM devem permanecer inalteráveis. Para reduzir o risco de interromper os clientes COM reordenando inadvertidamente o layout da interface, isole todas as alterações na classe do layout da interface definindo interfaces explicitamente.

Use o **ClassInterfaceAttribute** para desativar a geração automática da interface de classe e implementar uma interface explícita para a classe, como mostra o seguinte fragmento de código:

```
<ClassInterface(ClassInterfaceType.None)>Public Class LoanApp
    Implements IExplicit
    Sub M() Implements IExplicit.M
…
End Class
```

```
[ClassInterface(ClassInterfaceType.None)]
public class LoanApp : IExplicit
{
    int IExplicit.M() { return 0; }
}
```

O valor **ClassInterfaceType.None** impede que a interface de classe seja gerada quando os metadados da classe são exportados para uma biblioteca de tipos. No exemplo anterior, os clientes COM podem acessar a classe `LoanApp` somente pela interface `IExplicit`.

Evitar o cache de identificadores de expedição (DispIds)

O uso da interface de classe é uma opção aceitável para os clientes com script, os clientes do Microsoft Visual Basic 6.0 ou qualquer cliente de associação tardia que não armazena em cache os DispIds dos membros da interface. Os DispIds identificam os membros da interface para habilitar a associação tardia.

Para a interface de classe, a geração dos DispIds se baseia na posição do membro na interface. Se você alterar a ordem do membro e exportar a classe para uma biblioteca de tipos, você alterará os DispIds gerados na interface de classe.

Para evitar a interrupção de clientes COM de associação tardia ao usar a interface de classe, aplique o **ClassInterfaceAttribute** ao valor **ClassInterfaceType.AutoDispatch**. Esse valor implementa uma interface de classe somente de expedição, mas omite a descrição da interface na biblioteca de tipos. Sem uma descrição da interface, os clientes não conseguem armazenar em cache os DispIds em tempo de compilação. Embora esse seja o tipo de interface padrão para a interface de classe, é possível aplicar o valor do atributo explicitamente.

```
<ClassInterface(ClassInterfaceType.AutoDispatch)> Public Class LoanApp
    Implements IAnother
    Sub M() Implements IAnother.M
…
End Class
```

```
[ClassInterface(ClassInterfaceType.AutoDispatch)]
public class LoanApp
{
    public int M() { return 0; }
}
```

Para obter o DispId de um membro da interface em tempo de execução, os clientes COM podem chamar **IDispatch.GetIdsOfNames**. Para invocar um método na interface, passe o DispId retornado como um argumento para **IDispatch.Invoke**.

Restrinja o uso da opção de interface dupla para a interface de classe.

Interfaces duplas permitem a associação inicial e tardia a membros da interface por clientes COM. Em tempo de design e durante o teste, talvez seja útil definir a interface de classe como dupla. Para uma classe gerenciada (e suas classes base) que nunca serão modificadas, essa opção também é aceitável. Em todos os outros casos, evite definir a interface de classe como dupla.

Uma interface dupla gerada automaticamente pode ser apropriada em casos raros. No entanto, com mais frequência, ela cria complexidade relacionada à versão. Por exemplo, os clientes COM que usam a interface de classe de uma classe derivada podem ser facilmente interrompidos com alterações na classe base. Quando um terceiro fornece a classe base, o layout da interface de classe fica fora de seu controle. Além disso, ao contrário de uma interface somente de expedição, uma interface dupla (**ClassInterfaceType.AutoDual**) fornece uma descrição da interface de classe na biblioteca de tipos exportada. Uma descrição como essa incentiva os clientes de associação tardia a armazenarem em cache os DispIds em tempo de compilação.

Verifique se todas as notificações de evento COM têm associação tardia.

Por padrão, informações de tipo COM são incorporadas diretamente em assemblies gerenciados, o que elimina a necessidade de PIAS (assemblies de interoperabilidade primários). No entanto, uma das limitações das informações de tipo inseridas é que elas não são compatíveis com a entrega de notificações de eventos COM por chamadas early-bound (vtable), mas apenas com chamadas `IDispatch::Invoke` de associação tardia.

Se o seu aplicativo exigir chamadas early-bound para métodos de interface de eventos COM, defina a propriedade **Embed Interop Types** no Visual Studio como `true`, ou inclua o seguinte elemento no arquivo de projeto:

```
<EmbedInteropTypes>True</EmbedInteropTypes>
```

Consulte também

- ClassInterfaceAttribute
- Wrappers COM
- Expondo componentes do .NET Framework ao COM
- Como expor componentes do .NET Core ao COM
- Qualificando tipos .NET para interoperação
- RCW (Runtime Callable Wrapper)

Tipos .NET qualificados para interoperação COM

31/10/2019 • 3 minutes to read • Edit Online

Se você pretende expor os tipos em um assembly para aplicativos COM, considere os requisitos de interoperabilidade COM em tempo de design. Tipos gerenciados (classe, interface, estrutura e enumeração) se integram perfeitamente com tipos COM quando você obedece às seguintes diretrizes:

- As classes devem implementar interfaces explicitamente.

  Embora a interoperabilidade COM forneça um mecanismo para gerar automaticamente uma interface contendo todos os membros da classe e os membros da respectiva classe base, é muito melhor fornecer interfaces explícitas. A interface gerada automaticamente é chamada de interface de classe. Para obter diretrizes, confira Apresentando a interface de classe.

  Você pode usar Visual Basic, C# e C++ para incorporar as definições de interface no código, em vez de usar a linguagem IDL ou uma equivalente. Para obter detalhes da sintaxe, consulte a documentação da linguagem.

- Os tipos gerenciados devem ser públicos.

  Somente os tipos públicos em um assembly são registrados e exportados para a biblioteca de tipos. Como resultado, somente os tipos públicos são visíveis para COM.

  Tipos gerenciados expõem recursos para outro código gerenciado que pode não ser exposto a COM. Por exemplo, campos de constantes, construtores com parâmetros e métodos estáticos não são expostos a clientes COM. Além disso, como o runtime realiza marshaling de dados dentro e fora de um tipo, os dados podem ser copiados ou transformados.

- Propriedades, métodos, campos e eventos devem ser públicos.

  Membros de tipos públicos também devem ser públicos para ser visíveis para COM. Você pode restringir a visibilidade de um assembly, um tipo público ou membros públicos de um tipo público aplicando o ComVisibleAttribute. Por padrão, todos os membros e tipos públicos são visíveis.

- Os tipos devem ter um construtor sem parâmetros público para ser ativado no COM.

  Tipos públicos gerenciados são visíveis para o COM. No entanto, sem um construtor sem parâmetros público (um construtor sem argumentos), clientes COM não podem criar o tipo. Clientes COM ainda podem usar o tipo se ele é ativado por outros meios.

- Os tipos não podem ser abstratos.

  Nem clientes COM, tampouco clientes .NET podem criar tipos abstratos.

Quando exportados para COM, a hierarquia de herança de um tipo gerenciado é nivelada. O controle de versão também difere entre ambientes gerenciados e não gerenciados. Os tipos expostos ao COM não têm as mesmas características de controle de versão de outros tipos gerenciados.

Consulte também

- ComVisibleAttribute
- Expondo componentes do .NET Framework ao COM
- Apresentando a interface de classe
- Aplicando atributos de interoperabilidade

- Como empacotar um assembly .NET Framework para o COM

Aplicando atributos de interoperabilidade

31/10/2019 • 8 minutes to read • Edit Online

O namespace System.Runtime.InteropServices fornece três categorias de atributos específicos à interoperabilidade: aquelas aplicadas por você em tempo de design, aquelas aplicadas pelas ferramentas de interoperabilidade COM e as APIs durante o processo de conversão e aquelas aplicadas por você ou pela interoperabilidade COM.

Se você não estiver familiarizado com a tarefa de aplicação de atributos ao código gerenciado, consulte Estendendo metadados usando atributos. Assim como outros atributos personalizados, você pode aplicar atributos específicos à interoperabilidade a tipos, métodos, propriedades, parâmetros, campos e outros membros.

Atributos em tempo de design

Ajuste o resultado do processo de conversão executado pelas APIs e pelas ferramentas de interoperabilidade COM usando atributos em tempo de design. A tabela a seguir descreve os atributos que podem ser aplicados ao código-fonte gerenciado. Às vezes, as ferramentas de interoperabilidade COM também podem aplicar os atributos descritos nesta tabela.

ATRIBUTO	DESCRIÇÃO
AutomationProxyAttribute	Especifica se o tipo deve ter o marshaling realizado usando o marshaler de Automação ou um proxy e stub personalizados.
ClassInterfaceAttribute	Controla o tipo de interface gerado para uma classe.
CoClassAttribute	Identifica o CLSID da coclass original importado de uma biblioteca de tipos. As ferramentas de interoperabilidade COM geralmente aplicam esse atributo.
ComImportAttribute	Indica que uma definição de coclass ou interface foi importada de uma biblioteca de tipos COM. O runtime usa esse sinalizador para saber como ativar e realizar marshaling do tipo. Esse atributo proíbe o tipo que está sendo exportado novamente para uma biblioteca de tipos. As ferramentas de interoperabilidade COM geralmente aplicam esse atributo.
ComRegisterFunctionAttribute	Indica que um método deve ser chamado quando o assembly é registrado para uso por meio do COM, de modo que o código escrito pelo usuário possa ser executado durante o processo de registro.
ComSourceInterfacesAttribute	Identifica as interfaces que são fontes de eventos para a classe. Ferramentas de interoperabilidade COM podem aplicar esse atributo.

ATRIBUTO	DESCRIÇÃO
ComUnregisterFunctionAttribute	Indica que um método deve ser chamado quando o assembly tem o registro cancelado por meio do COM, de modo que o código escrito pelo usuário possa ser executado durante o processo.
ComVisibleAttribute	Renderiza tipos invisíveis para o COM quando o valor do atributo é igual a **false**. Esse atributo pode ser aplicado a um tipo individual ou a um assembly inteiro para controlar a visibilidade COM. Por padrão, todos os tipos gerenciados e públicos são visíveis; o atributo não é necessário para torná-los visíveis.
DispIdAttribute	Especifica o DISPID (identificador de expedição) COM de um método ou campo. Este atributo contém o DISPID do método, do campo ou da propriedade que ele descreve. Ferramentas de interoperabilidade COM podem aplicar esse atributo.
ComDefaultInterfaceAttribute	Indica a interface padrão para uma classe COM implementada no .NET. Ferramentas de interoperabilidade COM podem aplicar esse atributo.
FieldOffsetAttribute	Indica a posição física de cada campo dentro de uma classe quando usado com o **StructLayoutAttribute** e o **LayoutKind** é definido como Explicit.
GuidAttribute	Especifica o GUID (identificador global exclusivo) de uma classe, uma interface ou toda uma biblioteca de tipos. A cadeia de caracteres passada para o atributo deve ter um formato que seja um argumento de construtor aceitável para o tipo **System.Guid**. Ferramentas de interoperabilidade COM podem aplicar esse atributo.
IDispatchImplAttribute	Indica qual implementação da interface **IDispatch** o Common Language Runtime usa ao expor interfaces duplas e dispinterfaces ao COM.
InAttribute	Indica se os dados devem ter o marshaling realizado para o chamador. Pode ser usado para parâmetros de atributo.
InterfaceTypeAttribute	Controla como uma interface gerenciada é exposta aos clientes COM (Dupla, derivada de IUnknown ou somente IDispatch). Ferramentas de interoperabilidade COM podem aplicar esse atributo.
LCIDConversionAttribute	Indica que uma assinatura de método não gerenciado espera um parâmetro LCID. Ferramentas de interoperabilidade COM podem aplicar esse atributo.

ATRIBUTO	DESCRIÇÃO
MarshalAsAttribute	Indica como os dados em campos ou parâmetros devem ter o marshaling realizado entre o código gerenciado e não gerenciado. O atributo sempre é opcional, porque cada tipo de dados tem um comportamento de marshaling padrão. Ferramentas de interoperabilidade COM podem aplicar esse atributo.
OptionalAttribute	Indica que um parâmetro é opcional. Ferramentas de interoperabilidade COM podem aplicar esse atributo.
OutAttribute	Indica se os dados em um campo ou parâmetro devem ter o marshaling realizado de um objeto chamado novamente para seu chamador.
PreserveSigAttribute	Suprime a transformação de assinatura HRESULT ou retval que normalmente ocorre durante chamadas de interoperação. O atributo afeta o marshaling, bem como a exportação da biblioteca de tipos. Ferramentas de interoperabilidade COM podem aplicar esse atributo.
ProgIdAttribute	Especifica o ProgID de uma classe do .NET Framework. Pode ser usado para classes de atributos.
StructLayoutAttribute	Controla o layout físico dos campos de uma classe. Ferramentas de interoperabilidade COM podem aplicar esse atributo.

Atributos da ferramenta de conversão

A tabela a seguir descreve os atributos que as ferramentas de interoperabilidade COM aplicam durante o processo de conversão. Esses atributos não são aplicados em tempo de design.

ATRIBUTO	DESCRIÇÃO
ComAliasNameAttribute	Indica o alias COM de um parâmetro ou tipo de campo. Pode ser usado para parâmetros de atributo, campos ou valores retornados.
ComConversionLossAttribute	Indica que as informações sobre uma classe ou interface foram perdidas quando foram importadas de uma biblioteca de tipos para um assembly.
ComEventInterfaceAttribute	Identifica a interface de origem e a classe que implementa os métodos da interface do evento.
ImportedFromTypeLibAttribute	Indica que o assembly foi originalmente importado de uma biblioteca de tipos COM. Este atributo contém a definição de biblioteca de tipos da biblioteca de tipos original.

ATRIBUTO	DESCRIÇÃO
TypeLibFuncAttribute	Contém o **FUNCFLAGS** que foi originalmente importado para essa função da biblioteca de tipos COM.
TypeLibTypeAttribute	Contém o **TYPEFLAGS** que foi originalmente importado para esse tipo da biblioteca de tipos COM.
TypeLibVarAttribute	Contém o **VARFLAGS** que foi originalmente importado para essa variável da biblioteca de tipos COM.

Consulte também

- System.Runtime.InteropServices
- Expondo componentes do .NET Framework ao COM
- Atributos
- Qualificando tipos .NET para interoperação
- Como empacotar um assembly .NET Framework para o COM

Coleções e estruturas de dados

27/11/2019 • 10 minutes to read • Edit Online

Dados semelhantes podem normalmente ser tratados com mais eficiência quando armazenados e manipulados como uma coleção. Você pode usar a classe ou as classes System.Array nos namespaces System.Collections, System.Collections.Generic, System.Collections.Concurrent, System.Collections.Immutable para adicionar, remover e modificar elementos individuais ou um intervalo de elementos em uma coleção.

Há dois tipos principais de coleções; coleções genéricas e coleções não genéricas. Coleções genéricas foram adicionadas ao .NET Framework 2.0 e fornecem coleções que são fortemente tipadas no tempo de compilação. Por isso, coleções genéricas normalmente oferecem melhor desempenho. Coleções genéricas aceitam um parâmetro de tipo quando são criadas e não exigem que você converta de e para o tipo Object ao adicionar ou remover itens da coleção. Além disso, há suporte para a maioria das coleções genéricas em aplicativos da Windows Store. As coleções não genéricas armazenam itens como Object, exigem a conversão e a maioria não tem suporte para o desenvolvimento de aplicativos da Windows Store. No entanto, você pode ver as coleções não genéricas no código mais antigo.

Começando com o .NET Framework 4, as coleções no namespace System.Collections.Concurrent fornecem operações thread-safe eficientes para acessar itens da coleção de vários threads. As classes de coleção imutáveis no namespace System.Collections.Immutable (NuGet package) são inerentemente thread-safe, pois as operações são executadas em uma cópia da coleção original e a coleção original não pode ser modificada.

Recursos comuns de coleção

Todas as coleções fornecem métodos para adicionar, remover ou localizar itens na coleção. Além disso, todas as coleções que direta ou indiretamente implementam a interface ICollection ou a interface ICollection<T> compartilham estes recursos:

- **A capacidade de enumerar a coleção**

  Coleções do .NET Framework implementam System.Collections.IEnumerable ou System.Collections.Generic.IEnumerable<T> para permitir a iteração da coleção por meio dela. Um enumerador pode ser considerado um ponteiro móvel para qualquer elemento da coleção. A instrução foreach, in e a Instrução For Each...Next usam o enumerador exposto pelo método GetEnumerator e ocultam a complexidade de manipulação do enumerador. Além disso, qualquer coleção que implementa System.Collections.Generic.IEnumerable<T> é considerada um *tipo passível de consulta* e pode ser consultada com LINQ. Consultas LINQ fornecem um padrão comum para o acesso de dados. Elas são geralmente mais concisas e legíveis que loops `foreach` padrão e fornecem filtragem, classificação e agrupamento de recursos. Consultas LINQ também podem melhorar o desempenho. Para obter mais informações, consulte LINQ to Objects (C#), LINQ to Objects (Visual Basic), Parallel LINQ (PLINQ), Introdução a consultas LINQ (C#) e Operações básicas de consulta (Visual Basic).

- **A capacidade de copiar o conteúdo da coleção para uma matriz**

  Todas as coleções podem ser copiadas para uma matriz usando o método **CopyTo**; no entanto, a ordem dos elementos na nova matriz se baseia na sequência na qual o enumerador os retorna. A matriz resultante é sempre unidimensional com um limite inferior de zero.

Além disso, muitas classes de coleção contêm os seguintes recursos:

- **Propriedades de capacidade e contagem**

A capacidade de uma coleção é o número de elementos que ela pode conter. A contagem de uma coleção é o número de elementos que ela realmente contém. Algumas coleções ocultam a capacidade ou a contagem ou ambas.

A maioria das coleções se expande automaticamente em capacidade quando a capacidade atual é atingida. A memória é realocada e os elementos são copiados da coleção antiga para a nova. Isso reduz o código necessário para usar a coleção; no entanto, o desempenho da coleção pode ser afetado de forma negativa. Por exemplo, para List<T>, se Count for inferior a Capacity, adicionar um item será uma operação O(1). Se a capacidade precisar ser aumentada para acomodar o novo elemento, adicionar um item se torna uma operação O(n), em que n é Count. A melhor maneira de evitar um desempenho ruim causado por várias realocações é definir a capacidade inicial para que seja o tamanho estimado da coleção.

Uma BitArray é um caso especial; sua capacidade é a mesma que seu comprimento, que é o mesmo de sua contagem.

- **Um limite inferior consistente**

  O limite inferior de uma coleção é o índice do seu primeiro elemento. Todas as coleções indexadas nos namespaces System.Collections têm um limite inferior de zero, indicando que são indexados em 0. Array tem um limite inferior de zero por padrão, mas um limite inferior diferente pode ser definido ao criar uma instância da classe **Matriz** usando Array.CreateInstance.

- **Sincronização para acesso de vários threads** (System.Collections somente classes).

  Tipos de coleção não genérica no namespace System.Collections oferecem algum acesso thread-safe com sincronização; geralmente exposto por meio dos membros SyncRoot e IsSynchronized. Essas coleções são não thread-safe por padrão. Se você precisar de acesso com multithread escalável e eficiente para uma coleção, use uma das classes no namespace System.Collections.Concurrent ou considere usar uma coleção imutável. Para obter mais informações, veja Coleções thread-safe.

Escolhendo uma coleção

Em geral, você deve usar coleções genéricas. A tabela a seguir descreve alguns cenários comuns de coleção e as classes de coleção que você pode usar para esses cenários. Se você for inexperiente com coleções genéricas, esta tabela o ajudará a escolher a coleção genérica adequada para a tarefa.

EU QUERO...	OPÇÕES DE COLEÇÃO GENÉRICA	OPÇÕES DE COLEÇÃO NÃO GENÉRICA	OPÇÕES DE COLEÇÃO THREAD-SAFE OU IMUTÁVEL
Armazenar itens como pares chave/valor para consulta rápida por chave	Dictionary<TKey,TValue>	Hashtable (Um conjunto de pares chave/valor que são organizados com base no código hash da chave.)	ConcurrentDictionary<TKey, TValue> ReadOnlyDictionary<TKey,TValue> ImmutableDictionary<TKey,TValue>
Itens de acesso por índice	List<T>	Array ArrayList	ImmutableList<T> ImmutableArray
Usar itens primeiro a entrar, primeiro a sair (PEPS)	Queue<T>	Queue	ConcurrentQueue<T> ImmutableQueue<T>

EU QUERO...	OPÇÕES DE COLEÇÃO GENÉRICA	OPÇÕES DE COLEÇÃO NÃO GENÉRICA	OPÇÕES DE COLEÇÃO THREAD-SAFE OU IMUTÁVEL
Usar dados último a entrar, primeiro a sair (UEPS)	Stack<T>	Stack	ConcurrentStack<T> ImmutableStack<T>
Acessar itens em sequência	LinkedList<T>	Nenhuma recomendação	Nenhuma recomendação
Receba notificações quando itens forem removidos da coleção ou adicionados a ela. (implementa INotifyPropertyChanged e INotifyCollectionChanged)	ObservableCollection<T>	Nenhuma recomendação	Nenhuma recomendação
Uma coleção classificada	SortedList<TKey,TValue>	SortedList	ImmutableSortedDictionary<TKey,TValue> ImmutableSortedSet<T>
Um conjunto de funções matemáticas	HashSet<T> SortedSet<T>	Nenhuma recomendação	ImmutableHashSet<T> ImmutableSortedSet<T>

Tópicos relacionados

TÍTULO	DESCRIÇÃO
Selecionando uma Classe de Coleção	Descreve as diferentes coleções e ajuda a selecionar uma para o seu cenário.
Tipos de Coleção de Uso Comum	Descreve os tipos de coleção genérica e não genérica normalmente usadas, tais como System.Array, System.Collections.Generic.List<T>, e System.Collections.Generic.Dictionary<TKey,TValue>.
Quando Usar Coleções Genéricas	Descreve o uso de tipos de coleção genérica.
Comparações e Classificações Dentro de Coleções	Discute o uso de comparações de igualdade e comparações de classificação em coleções.
Tipos de Coleção Sorted	Descreve as características e o desempenho de coleções classificadas
Tipos de Coleção de Tabela de Hash e Dicionário	Descreve os recursos de tipos de dicionário baseado em hash genérico e não genérico.
Coleções Thread-Safe	Descreve os tipos de coleção, tais como System.Collections.Concurrent.BlockingCollection<T> e System.Collections.Concurrent.ConcurrentBag<T> que dão suporte a acesso simultâneo seguro e eficiente de vários threads.
System.Collections.Immutable	Apresenta as coleções imutáveis e fornece links para os tipos de coleção.

Referência

System.Array
System.Collections
System.Collections.Concurrent
System.Collections.Generic
System.Collections.Specialized
System.Linq
System.Collections.Immutable

Numéricos no .NET

31/10/2019 • 7 minutes to read • Edit Online

O .NET fornece uma variedade de inteiros numéricos e primitivos de ponto flutuante, bem como System.Numerics.BigInteger, que é um tipo integral sem limite teórico superior ou inferior, System.Numerics.Complex, que representa números complexos e um conjunto de tipos habilitados para SIMD no namespace System.Numerics.

Tipos de inteiro

O .NET dá suporte a tipos inteiros tanto com sinal quanto sem sinal de 8, 16, 32 e 64 bits, que estão listados na tabela a seguir:

DIGITE	ASSINADO/NÃO ASSINADO	TAMANHO (EM BYTES)	VALOR MÍNIMO	VALOR MÁXIMO
System.Byte	Não assinado	1	0	255
System.Int16	Assinado	2	-32,768	32,767
System.Int32	Assinado	4	-2,147,483,648	2,147,483,647
System.Int64	Assinado	8	-9,223,372,036,854,775,808	9,223,372,036,854,775,807
System.SByte	Assinado	1	-128	127
System.UInt16	Não assinado	2	0	65,535
System.UInt32	Não assinado	4	0	4,294,967,295
System.UInt64	Não assinado	8	0	18,446,744,073,709,551,615

Cada tipo inteiro dá suporte a um conjunto de operadores aritméticos padrão. A classe System.Math fornece métodos para um conjunto mais amplo de funções matemáticas.

Você também pode trabalhar com os bits individuais nos valores usando a classe System.BitConverter.

> **NOTE**
> Os tipos de inteiro sem sinal não estão em conformidade com CLS. Para saber mais, veja Componentes de independência de linguagem e componentes independentes da linguagem.

BigInteger

A estrutura System.Numerics.BigInteger é um tipo imutável que representa um inteiro arbitrariamente grande cujo valor, em teoria, não tem limites superiores ou inferiores. Os métodos do tipo BigInteger são muito semelhantes aos dos outros tipos integrais.

Tipos de ponto flutuante

O .NET inclui três primitivos tipos de ponto flutuante, que estão listados na tabela a seguir:

DIGITE	TAMANHO (EM BYTES)	INTERVALO APROXIMADO	PRECISÃO
System.Single	4	$\pm1,5 \times 10^{-45}$ para $\pm3,4 \times 10^{38}$	~6 a 9 dígitos
System.Double	8	$\pm5.0 \times 10^{-324}$ to $\pm1.7 \times 10^{308}$	~15 a 17 dígitos
System.Decimal	16	$\pm1,0 \times 10^{-28}$ para $\pm7,9228 \times 10^{28}$	28 a 29 dígitos

Os tipos Single e Double dão suporte a valores especiais que representam não é um número e infinito. Por exemplo, o tipo Double fornece os seguintes valores: Double.NaN, Double.NegativeInfinity e Double.PositiveInfinity. Você usa os métodos Double.IsNaN, Double.IsInfinity, Double.IsPositiveInfinity e Double.IsNegativeInfinity para testar esses valores especiais.

Cada tipo de ponto flutuante dá suporte a um conjunto de operadores aritméticos padrão. A classe System.Math fornece métodos para um conjunto mais amplo de funções matemáticas. O .NET Core 2.0 e posteriores incluem a classe System.MathF que fornece métodos que aceitam argumentos do tipo Single.

Você também pode trabalhar com os bits individuais nos valores Double e Single usando a classe System.BitConverter. A estrutura System.Decimal tem seus próprios métodos Decimal.GetBits e Decimal.Decimal(Int32[]), para trabalhar com os bits individuais de um valor decimal, assim como seu próprio conjunto de métodos para executar algumas operações matemáticas adicionais.

Os tipos Double e Single destinam-se a ser usados para valores que, por sua natureza, são imprecisos (por exemplo, a distância entre duas estrelas) e para aplicativos em que um alto grau de precisão e erro de arredondamento pequeno não são necessários. Você deve usar o tipo System.Decimal para casos em que uma maior precisão é necessária e erros de arredondamento devem ser minimizados.

> **NOTE**
> O tipo Decimal não elimina a necessidade de arredondamento. Em vez disso, ele minimiza erros devido a arredondamento.

Complexo

A estrutura System.Numerics.Complex representa um número complexo, ou seja, um número com uma parte de número real e uma parte de número imaginário. Dá suporte a um conjunto padrão de aritmética, de comparação, de igualdade, de conversões explícita e implícita, bem como a métodos matemáticos, algébricos e trigonométricos.

Tipos habilitados para SIMD

O namespace System.Numerics inclui um conjunto de tipos habilitados para SIMD do .NET. Operações SIMD (Single Instruction Multiple Data) podem ser paralelizadas no nível de hardware. Isso aumenta a taxa de transferência dos cálculos vetorizadas, que são comuns em aplicativos matemáticos, científicos e gráficos.

Os tipos habilitados para SIMD do .NET incluem o seguinte:

- Os tipos Vector2, Vector3 e Vector4, que representam vetores com 2, 3 e 4 valores de Single.
- Dois tipos de matriz, Matrix3x2, que representa uma matriz 3x2, e Matrix4x4, que representa uma matriz 4x4.

- O tipo Plane representa um plano no espaço tridimensional.
- O tipo Quaternion, que representa um vetor usado para codificar rotações físicas tridimensionais.
- O tipo Vector<T>, que representa um vetor de um tipo numérico especificado e fornece um amplo conjunto de operadores que se beneficiam de suporte a SIMD. A contagem de uma instância Vector<T> é corrigida, mas seu valor Vector<T>.Count depende da CPU do computador em que o código é executado.

  > **NOTE**
  > O tipo Vector<T> não está incluído no .NET Framework. Você deve instalar o pacote System.Numerics.Vectors do NuGet para obter acesso a esse tipo.

Os tipos habilitados para SIMD são implementados de modo que possam ser usados com hardware não habilitados para SIMD ou compiladores JIT. Para aproveitar instruções SIMD, seus aplicativos de 64 bits devem ser executados pelo runtime que usa o compilador RyuJIT, que está incluído no .NET Core e no .NET Framework 4.6 e versões posteriores. Ele adiciona suporte a SIMD quando tem processadores de 64 bits como destino.

Consulte também

- Fundamentos do aplicativo
- Cadeias de Caracteres de Formato Numérico Padrão

Datas, horas e fusos horários

31/10/2019 • 6 minutes to read • Edit Online

Além da estrutura DateTime básica, o .NET fornece as seguintes classes que dão suporte para trabalhar com fusos horários:

- TimeZone

  Use esta classe para trabalhar com o fuso horário local do sistema e a zona UTC (Tempo Universal Coordenado). A funcionalidade da classe TimeZone é amplamente substituída pela classe TimeZoneInfo.

- TimeZoneInfo

  Use essa classe para trabalhar com qualquer fuso horário que esteja predefinido em um sistema, para criar novos fusos horários e para converter facilmente datas e horas de um fuso horário para outro. Para novos desenvolvimentos, use a classe TimeZoneInfo em vez da classe TimeZone.

- DateTimeOffset

  Use essa estrutura para trabalhar com datas e horas cujo deslocamento (ou diferença) em relação ao horário UTC é conhecido. A estrutura DateTimeOffset combina um valor de data e hora com o deslocamento desse horário em relação ao UTC. Devido à sua relação com o UTC, um valor individual de data e hora identifica, sem ambiguidade um único ponto no tempo. Isso aumenta a portabilidade de um computador para outro de um valor de DateTimeOffset em relação a um valor de DateTime.

Esta seção da documentação fornece as informações de que você precisa para trabalhar com fusos horários e para criar aplicativos com reconhecimento de fuso horário que podem converter datas e horas de um fuso horário para outro.

Nesta seção

Visão geral sobre fuso horário Discute a terminologia, os conceitos e os problemas envolvidos na criação de aplicativos com reconhecimento de fuso horário.

Escolhendo entre DateTime, DateTimeOffset, TimeSpan e TimeZoneInfo Discute quando usar os tipos DateTime, DateTimeOffset e TimeZoneInfo ao trabalhar com os dados de data e hora.

Encontrando os fusos horários definidos em um sistema local Descreve como enumerar os fusos horários encontrados em um sistema local.

Como enumerar os fusos horários presentes em um computador Fornece exemplos que enumeram os fusos horários definidos no Registro de um computador e que permitem que os usuários selecionem um fuso horário predefinido em uma lista.

Como acessar os objetos de fuso horário UTC e local predefinidos Descreve como acessar o Tempo Universal Coordenado e o fuso horário local.

Como criar uma instância de um objeto TimeZoneInfo Descreve como criar uma instância de um objeto TimeZoneInfo do Registro do sistema local.

Criando uma instância de um objeto DateTimeOffset Discute as maneiras de criar uma instância de um objeto DateTimeOffset e maneiras de converter um valor DateTime em um valor DateTimeOffset.

Como criar fusos horários sem regras de ajuste Descreve como criar um fuso horário personalizado sem suporte para transição de/para o horário de verão.

Como criar fusos horários com regras de ajuste Descreve como criar um fuso horário personalizado com suporte a uma ou mais transições de/para o horário de verão.

Salvando e restaurando fusos horários Descreve o suporte de TimeZoneInfo para serialização e desserialização dos dados de fuso horário e ilustra alguns dos cenários nos quais esses recursos podem ser usados.

Como salvar fusos horários em um recurso inserido Descreve como criar um fuso horário personalizado e salvar suas informações em um arquivo de recurso.

Como restaurar fusos horários de um recurso inserido Descreve como criar instâncias de fusos horários personalizados que foram salvos em um arquivo de recurso inserido.

Executando operações aritméticas com datas e horas Discute os problemas envolvidos ao adicionar, subtrair e comparar valores DateTime e DateTimeOffset.

Como usar fusos horários em aritmética de data e hora Descreve como realizar uma aritmética de data e hora que reflita as regras de ajuste de um fuso horário.

Convertendo entre DateTime e DateTimeOffset Descreve como converter entre valores DateTime e DateTimeOffset.

Convertendo horários entre fusos horários Descreve como converter horários de um fuso horário para outro.

Como resolver horários ambíguos Descreve como resolver um horário ambíguo, mapeando-o para o horário padrão do fuso horário.

Como permitir que os usuários resolvam horários ambíguos Descreve como permitir que um usuário determine o mapeamento entre um horário local ambíguo e o Tempo Universal Coordenado.

Referência

System.TimeZoneInfo

Manipulando e acionando eventos

04/11/2019 • 15 minutes to read • Edit Online

Os eventos no .NET são baseados no modelo de representante. O modelo de representante segue o padrão de design do observador, que permite a um assinante se registrar em um provedor e receber notificações dele. Um remetente de eventos envia uma notificação por push de que um evento ocorreu e um receptor de eventos recebe essa notificação e define uma resposta. Este artigo descreve os principais componentes do modelo de representante, como consumir eventos em aplicativos e como implementar eventos no código.

Para saber mais sobre como manipular eventos em aplicativos da Windows 8.x Store, confira Visão geral de eventos e eventos roteados.

Eventos

Um evento é uma mensagem enviada por um objeto para sinalizar a ocorrência de uma ação. A ação pode ser causada pela interação do usuário, como o clique em um botão, ou ser resultado de alguma outra lógica de programa, como a alteração do valor de uma propriedade. O objeto que aciona o evento é chamado de *remetente do evento*. O remetente do evento não sabe qual objeto ou método receberá (identificador) os eventos que ele aciona. O evento normalmente é membro do remetente do evento. Por exemplo, o evento Click é membro da classe Button e o evento PropertyChanged é membro da classe que implementa a interface INotifyPropertyChanged.

Para definir um evento, use a palavra-chave `event` no C# ou `Event` no Visual Basic na assinatura da sua classe de evento e especifique o tipo de representante para o evento. Os representantes são descritos na próxima seção.

Normalmente, para acionar um evento, você adiciona um método que é marcado como `protected` e `virtual` (em C#) ou `Protected` e `Overridable` (no Visual Basic). Dê a esse método o nome `On` *EventName*; por exemplo, `OnDataReceived`. O método deve usar um parâmetro que especifica um objeto de dados de evento, que é um objeto do tipo EventArgs ou um tipo derivado. Você fornece esse método para permitir que as classes derivadas substituam a lógica para acionamento do evento. Uma classe derivada sempre deve chamar o método `On` *EventName* da classe base a fim de garantir que os representantes registrados recebam o evento.

O exemplo de código a seguir mostra como declarar um evento denominado `ThresholdReached`. O evento está associado ao representante EventHandler e é gerado em um método chamado `OnThresholdReached`.

```
class Counter
{
    public event EventHandler ThresholdReached;

    protected virtual void OnThresholdReached(EventArgs e)
    {
        EventHandler handler = ThresholdReached;
        handler?.Invoke(this, e);
    }

    // provide remaining implementation for the class
}
```

```
Public Class Counter
    Public Event ThresholdReached As EventHandler

    Protected Overridable Sub OnThresholdReached(e As EventArgs)
        RaiseEvent ThresholdReached(Me, e)
    End Sub

    ' provide remaining implementation for the class
End Class
```

Delegados

Um representante é um tipo que contém uma referência a um método. Um representante é declarado com uma assinatura que mostra o tipo de retorno e os parâmetros para os métodos aos quais faz referência, e pode conter referências apenas aos métodos que correspondem à sua assinatura. Portanto, um representante é equivalente a um ponteiro de função fortemente tipado ou um retorno de chamada. Uma declaração de representante é suficiente para definir uma classe de representante.

Representantes têm muitos usos no .NET. No contexto de eventos, um representante é um intermediário (ou mecanismo do tipo ponteiro) entre a origem do evento e o código que manipula o evento. Associe um representante a um evento incluindo o tipo de representante na declaração do evento, como mostrado no exemplo da seção anterior. Para obter mais informações sobre representantes, consulte a classe Delegate.

O .NET fornece os representantes EventHandler e EventHandler<TEventArgs> para dar suporte à maioria dos cenários de evento. Use o representante EventHandler para todos os eventos que não incluem dados de evento. Use o representante EventHandler<TEventArgs> para eventos que incluem dados sobre o evento. Esses representantes não têm valor de tipo de retorno e usam dois parâmetros (um objeto para a origem do evento e um objeto para dados do evento).

Os representantes são multicast, o que significa que eles podem manter referências a mais de um método de manipulação de eventos. Para obter detalhes, consulte a página de referência Delegate. Os representantes proporcionam flexibilidade e controle refinado na manipulação de eventos. Um representante atua como um dispatcher de evento para a classe que aciona o evento ao manter uma lista de manipuladores de eventos registrados para o evento.

Para cenários em que os representantes EventHandler e EventHandler<TEventArgs> não funcionam, você pode definir um representante. Os cenários que exigem que você defina um representante são muito raros; por exemplo, quando você deve trabalhar com código que não reconhece genéricos. Marque um representante com a palavra-chave `delegate` no C# e `Delegate` no Visual Basic na declaração. O exemplo a seguir mostra como declarar um representante chamado `ThresholdReachedEventHandler`.

```
public delegate void ThresholdReachedEventHandler(object sender, ThresholdReachedEventArgs e);
```

```
Public Delegate Sub ThresholdReachedEventHandler(sender As Object, e As ThresholdReachedEventArgs)
```

Dados de evento

Os dados associados a um evento podem ser fornecidos por meio de uma classe de dados do evento. O .NET fornece muitas classes de dados de evento que podem ser usadas em seus aplicativos. Por exemplo, a classe SerialDataReceivedEventArgs é a classe de dados de evento do evento SerialPort.DataReceived. O .NET segue um padrão de nomenclatura de terminação para todas as classes de dados de evento com `EventArgs`. Determine qual classe de dados de evento está associada a um evento observando o representante do evento. Por exemplo, o representante SerialDataReceivedEventHandler inclui a classe SerialDataReceivedEventArgs como um de seus

parâmetros.

A classe EventArgs é o tipo base para todas as classes de dados de evento. EventArgs também é a classe usada quando um evento não tem nenhum dado associado. Quando você criar um evento cuja finalidade seja apenas notificar outras classes de que algo aconteceu e que não precise passar nenhum dado, inclua a classe EventArgs como o segundo parâmetro no representante. Você poderá passar o valor EventArgs.Empty quando nenhum dado for fornecido. O representante EventHandler inclui a classe EventArgs como um parâmetro.

Quando quiser criar uma classe de dados de evento personalizada, crie uma classe derivada de EventArgs e forneça todos os membros necessários para passar dados que estejam relacionados ao evento. Normalmente, você deve usar o mesmo padrão de nomenclatura do .NET e terminar o nome da classe de dados de evento com `EventArgs`.

O exemplo a seguir mostra uma classe de dados de evento chamada `ThresholdReachedEventArgs`. Ele contém propriedades que são específicas ao evento que está sendo acionado.

```
public class ThresholdReachedEventArgs : EventArgs
{
    public int Threshold { get; set; }
    public DateTime TimeReached { get; set; }
}
```

```
Public Class ThresholdReachedEventArgs
    Inherits EventArgs

    Public Property Threshold As Integer
    Public Property TimeReached As DateTime
End Class
```

Manipuladores de eventos

Para responder a um evento, você pode definir um método de manipulador de eventos no receptor do evento. Esse método deve corresponder à assinatura do representante para o evento que está sendo manipulado. No manipulador de eventos, execute as ações que são necessárias quando o evento é acionado, como coletar a entrada do usuário depois que ele clica em um botão. Para receber notificações de ocorrência de eventos, o método de manipulador de eventos deve estar inscrito no evento.

O exemplo a seguir mostra um método de manipulador de eventos chamado `c_ThresholdReached` que corresponde à assinatura para o representante EventHandler. O método está inscrito no evento `ThresholdReached`.

```
class Program
{
    static void Main()
    {
        var c = new Counter();
        c.ThresholdReached += c_ThresholdReached;

        // provide remaining implementation for the class
    }

    static void c_ThresholdReached(object sender, EventArgs e)
    {
        Console.WriteLine("The threshold was reached.");
    }
}
```

```
Module Module1

    Sub Main()
        Dim c As New Counter()
        AddHandler c.ThresholdReached, AddressOf c_ThresholdReached

        ' provide remaining implementation for the class
    End Sub

    Sub c_ThresholdReached(sender As Object, e As EventArgs)
        Console.WriteLine("The threshold was reached.")
    End Sub
End Module
```

Manipuladores de eventos estáticos e dinâmicos

O .NET permite que os assinantes se registrem para receber notificações de eventos de modo estático ou dinâmico. Os manipuladores de eventos estáticos permanecem em vigor por toda a vida da classe cujos eventos eles manipulam. Os manipuladores de eventos dinâmicos são ativados e desativados explicitamente durante a execução do programa, geralmente em resposta a alguma lógica de programa condicional. Por exemplo, eles podem ser usados se as notificações de eventos forem necessárias apenas sob determinadas condições, ou se um aplicativo fornecer vários manipuladores de eventos e as condições de tempo de execução definirem o apropriado para uso. O exemplo na seção anterior mostra como adicionar dinamicamente um manipulador de eventos. Para obter mais informações, veja Eventos (no Visual Basic) e Eventos (em C#).

Acionando vários eventos

Se sua classe acionar vários eventos, o compilador vai gerar um campo por instância de representante de evento. Se o número de eventos for grande, o custo de armazenamento de um campo por representante pode não ser aceitável. Para esses casos, o .NET fornece propriedades de evento que você pode usar com outra estrutura de dados de sua escolha para armazenar representantes de eventos.

As propriedades de evento consistem em declarações de evento acompanhadas por acessadores de evento. Os acessadores de evento são métodos que você define para adicionar ou remover instâncias de representante de evento da estrutura de dados de armazenamento. Observe que as propriedades de evento são mais lentas que os campos de evento, pois cada representante de evento deve ser recuperado para que possa ser invocado. A compensação está entre a memória e a velocidade. Se sua classe define muitos eventos que raramente são acionados, você desejará implementar as propriedades de evento. Para saber mais, confira Como manipular vários eventos usando propriedades de evento.

Tópicos relacionados

TÍTULO	DESCRIÇÃO
Como acionar e consumir eventos	Contém exemplos de como acionar e consumir eventos.
Como manipular vários eventos usando propriedades de evento	Mostrar como usar propriedades de evento para manipular vários eventos.
Padrão de design do observador	Descreve o padrão de design que permite a um assinante se registrar em um provedor e receber notificações dele.
Como consumir eventos em um aplicativo Web Forms	Mostra como manipular um evento acionado por um controle do Web Forms.

Consulte também

- EventHandler
- EventHandler<TEventArgs>
- EventArgs
- Delegate
- Eventos (Visual Basic)
- Eventos (guia de programação em C#)
- Visão geral de eventos e eventos roteados (aplicativos UWP)

Processo de execução gerenciada

31/10/2019 • 16 minutes to read • Edit Online

O processo de execução gerenciada inclui as seguintes etapas, que serão discutidas em detalhes mais adiante neste tópico:

1. Escolhendo um compilador.

   Para obter as vantagens fornecidas pelo Common Language Runtime, você deve usar um ou mais compiladores de linguagem que selecionam o tempo de execução.

2. Compilando seu código em MSIL.

   A compilação converte seu código-fonte em MSIL (Microsoft Intermediate Language) e gera os metadados necessários.

3. Compilando MSIL em código nativo.

   No tempo de execução, uma compilação JIT (just-in-time) converte o MSIL em código nativo. Durante essa compilação, o código deve passar por um processo de verificação que examina o MSIL e os metadados para descobrir se o código pode ser considerado fortemente tipado.

4. Executando o código.

   O Common Language Runtime fornece a infraestrutura que permite que a execução ocorra e os serviços que podem ser usados durante a execução.

Escolhendo um compilador

Para obter as vantagens fornecidas pelo CLR (Common Language Runtime), você deve usar um ou mais compiladores de linguagem que direcionam o tempo de execução, como Visual Basic, C#, Visual C++, F# ou um dos muitos compiladores de terceiros como um Eiffel, um Perl ou um compilador COBOL.

Por ser um ambiente de execução multilíngue, o runtime dá suporte a uma grande variedade de tipos de dados e recursos de linguagem. O compilador de linguagem que você usa determina quais recursos do runtime estão disponíveis, e você cria seu código usando esses recursos. O seu compilador, e não o runtime, estabelece a sintaxe que seu código deve usar. Se o componente deve ser completamente utilizável por componentes escritos em outras linguagens, os tipos exportados do seu componente devem expor somente recursos de linguagem incluídos na Independência de linguagem e componentes independentes de linguagem (CLS). Você pode usar o atributo CLSCompliantAttribute para garantir que seu código seja compatível com CLS. Para saber mais, veja Componentes de independência de linguagem e componentes independentes da linguagem.

Voltar ao início

Compilando para MSIL

Quando compila para o código gerenciado, o compilador converte seu código fonte em MSIL, que é um conjunto de instruções independente de CPU que pode ser convertido em código nativo com eficiência. O MSIL inclui instruções para carregamento, armazenamento, inicialização e chamada de métodos em objetos, bem como instruções para operações aritméticas e lógicas, fluxo de controle, acesso direto à memória, tratamento de exceções e outras operações. Para o código ser executado, a MSIL deve ser convertida em código específico de CPU, geralmente por um Compilador JIT (Just-In-Time). Como o Common Language Runtime fornece um ou mais compiladores JIT para cada arquitetura de computador para a qual dá suporte, o mesmo conjunto de MSIL

pode ser compilado por JIT e executado em qualquer arquitetura compatível.

Quando um compilador produz MSIL, ele também produz metadados. Metadados descrevem os tipos no seu código, incluindo a definição de cada tipo, as assinaturas de membros de cada tipo, os membros que seu código referencia e outros dados que o runtime usa no runtime. O MSIL e os metadados estão contidos em um arquivo PE (Portable Executable) com base no e que estende o arquivo PE Microsoft publicado e o COFF (Common Object File Format) usados historicamente no conteúdo executável. Esse formato de arquivo, que acomoda MSIL ou código nativo, bem como metadados, permite ao sistema operacional reconhecer imagens do Common Language Runtime. A presença de metadados no arquivo com MSIL permite que seu código se descreva, o que significa que não há necessidade de bibliotecas de tipos ou IDL (Interface Definition Language). O runtime localiza e extrai os metadados do arquivo conforme necessário durante a execução.

Voltar ao início

Compilando MSIL para código nativo

Para executar o MSIL, ele deve ser compilado no CLR para código nativo da arquitetura do computador de destino. O .NET Framework fornece duas maneiras para realizar essa conversão:

- Um compilador JIT (just-in-time) do .NET Framework.
- O Ngen.exe (Gerador de Imagem Nativa) do .NET Framework.

Compilação pelo compilador JIT

A compilação JIT converte MSIL para código nativo sob demanda no tempo de execução do aplicativo, quando o conteúdo de um assembly é carregado e executado. Como o CLR fornece um compilador JIT para cada arquitetura de CPU compatível, os desenvolvedores podem compilar um conjunto de assemblies MSIL que pode ser compilado pelo JIT e executado em diferentes computadores com arquiteturas diferentes. No entanto, se seu código gerenciado chama APIs nativas específicas da plataforma ou uma biblioteca de classes específica da plataforma, ele será executado somente nesse sistema operacional.

A compilação JIT leva em conta a possibilidade de um código jamais ser chamado durante a execução. Em vez de usar o tempo e a memória para converter todo o MSIL em um arquivo PE para código nativo, ele converte o MSIL conforme necessário durante a execução e armazena o código nativo resultante na memória de forma que seja acessível para chamadas subsequentes no contexto desse processo. O carregador cria e anexa um stub para cada método em um tipo quando o tipo é carregado e inicializado. Quando um método é chamado pela primeira vez, o stub passa o controle para o compilador JIT, que converte o MSIL para esse método em código nativo e modifica o stub para apontar diretamente para o código nativo gerado. Portanto, as chamadas subsequentes para o método compilado por JIT vão diretamente para o código nativo.

Geração de código do tempo de instalação usando NGen.exe

Como o compilador JIT converte o MSIL do assembly para código nativo quando métodos individuais definidos nesse assembly são chamados, ele afeta o desempenho negativamente no tempo de execução. Na maioria dos casos, esse desempenho diminuído é aceitável. Mais importante, o código gerado pelo compilador JIT é associado ao processo que disparou a compilação. Ele não pode ser compartilhado por vários processos. Para permitir que o código gerado seja compartilhado por várias invocações de um aplicativo ou por vários processos que compartilham um conjunto de assemblies, o CLR dá suporte a um modo de compilação antecipado. Esse modo de compilação Ahead Of Time usa o Ngen.exe (Gerador de Imagem Nativa) para converter assemblies MSIL em código nativo de maneira semelhante ao compilador JIT. No entanto, a operação de Ngen.exe é diferente da operação do compilador JIT de três maneiras:

- Ele executa a conversão do MSIL em código nativo antes de executar o aplicativo, e não durante a execução do aplicativo.
- Ele cria um assembly inteiro por vez, em vez de um método de cada vez.

- Ele mantém o código gerado no Cache de Imagem Nativa como um arquivo no disco.

Verificação de código

Como parte de sua compilação para código nativo, o código MSIL deve passar por um processo de verificação, a menos que um administrador estabeleça uma política de segurança permitindo que o código ignore a verificação. A verificação examina o MSIL e os metadados para descobrir se o código é fortemente tipado, o que significa que ele acessa apenas os locais de memória que está autorizado a acessar. A segurança de tipo ajuda a isolar objetos uns dos outros e a protegê-los de danos não intencionais ou corrupção mal-intencionada. Ela também dá garantia de que restrições de segurança no código possam ser aplicadas confiavelmente.

O runtime depende das seguintes declarações serem verdadeiras para o código que é comprovadamente fortemente tipado:

- Uma referência a um tipo é estritamente compatível com o tipo que está sendo referenciado.
- Somente operações definidas adequadamente são invocadas em um objeto.
- Identidades são o que elas afirmam ser.

Durante o processo de verificação, o código MSIL é examinado em uma tentativa de confirmar que o código possa acessar locais de memória e chamar métodos apenas por tipos corretamente definidos. Por exemplo, o código não pode permitir que campos de um objeto sejam acessados de uma maneira que permita que locais de memória sejam saturados. Além disso, a verificação inspeciona o código para determinar se o MSIL foi corretamente gerado, porque MSIL incorreto pode levar a uma violação das regras de segurança de tipos. O processo de verificação passa um conjunto bem definido de código fortemente tipado, e ele passa apenas código fortemente tipado. Entretanto, alguns códigos fortemente tipados podem não passar nessa verificação devido às limitações do processo de verificação, e algumas linguagens, por projeto, não produzem código fortemente tipado verificável. Se o código fortemente tipado for exigido pela política de segurança mas o código não passar pela verificação, uma exceção será lançada quando o código for executado.

Voltar ao início

Executando o código

O CLR fornece a infraestrutura que permite que a execução gerenciada ocorra e os serviços que podem ser usados durante a execução. Para um método ser executado, ele deve ser compilado para o código específico do processador. Cada método para MSIL gerado é compilado por JIT quando é chamado pela primeira vez e executado. Na próxima vez em que o método for executado, o código nativo compilado por JIT existente será executado. O processo de compilação por JIT e a execução do código é repetido até a execução ser concluída.

Durante a execução, o código gerenciado recebe serviços como coleta de lixo, segurança, interoperabilidade com código não gerenciado, suporte à depuração entre linguagens e suporte avançado à implantação e ao controle de versão.

No Microsoft Windows XP e Windows Vista, o carregador do sistema operacional verifica módulos gerenciados examinando um bit no cabeçalho COFF. Se o bit estiver definido, o módulo será gerenciado. Se o carregador detecta módulos gerenciados, ele carrega mscoree.dll e `_CorValidateImage`, e `_CorImageUnloading` notifica o carregador quando as imagens do módulo gerenciado são carregadas e descarregadas. `_CorValidateImage` executa as seguintes ações:

1. Garante que o código seja um código gerenciado válido.
2. Altera o ponto de entrada na imagem para um ponto de entrada no ambiente de execução.

No Windows 64 bits, `_CorValidateImage` modifica a imagem que está na memória transformando-a do formato PE32 para o formato PE32+.

Voltar ao início

Consulte também

- Visão Geral
- Componentes de independência de linguagem e componentes independentes da linguagem
- Metadados e componentes autodescritivos
- Ilasm.exe (IL Assembler)
- Security
- Interoperação com código não gerenciado
- Implantação
- Assemblies no .NET
- Domínios do aplicativo

Metadados e componentes autodescritivos

31/10/2019 • 16 minutes to read • Edit Online

No passado, um componente de software (.exe ou .dll) escrito em uma linguagem não podia usar facilmente um componente de software escrito em outra linguagem. COM foi um passo para a solução desse problema. O .NET Framework facilita ainda mais a interoperação entre componentes permitindo que compiladores emitam informações declarativas adicionais sobre todos os módulos e assemblies. Essas informações, chamadas de metadados, ajudam os componentes a interagirem perfeitamente.

Metadados são informações binárias que descrevem o seu programa, armazenadas em um arquivo PE (Portable Executable) do Common Language Runtime ou na memória. Quando você compila seu código em um arquivo PE, os metadados são inseridos em uma parte do arquivo, e o código é convertido em MSIL (Microsoft Intermediate Language) e inserido em outra parte do arquivo. Cada tipo e membro definido e referenciado em um módulo ou assembly é descrito em metadados. Quando o código é executado, o runtime carrega os metadados na memória e os referencia para descobrir informações sobre suas classes de código, membros, herança etc.

Os metadados descrevem cada tipo e membro definido no seu código de maneira neutra em relação à linguagem. Os metadados armazenam as seguintes informações:

- Descrição do assembly.
  - Identidade (nome, versão, cultura, chave pública).
  - Os tipos que são exportados.
  - Outros assemblies dos quais esse assembly dependa.
  - Permissões de segurança necessárias à execução.
- Descrição dos tipos.
  - Nome, visibilidade, classe base e interfaces implementadas.
  - Membros (métodos, campos, propriedades, eventos, tipos aninhados).
- Atributos.
  - Elementos descritivos adicionais que modificam tipos e membros.

Benefícios dos metadados

Os metadados são a chave para um modelo de programação mais simples e eliminam a necessidade de arquivos IDL (Interface Definition Language), arquivos de cabeçalho ou qualquer método externo de referência a componente. Os metadados permitem que linguagens .NET Framework se descrevam automaticamente de maneira neutra em relação à linguagem, sem que o desenvolvedor e o usuário vejam. Além disso, metadados são extensíveis pelo uso de atributos. Os metadados oferecem os seguintes benefícios principais:

- Arquivos autodescritivos.

  Módulos e assemblies do Common Language Runtime são autodescritivos. Os metadados de um módulo contêm tudo de que ele precisa para interagir com outro módulo. Como os metadados fornecem automaticamente a funcionalidade de IDL em COM, você pode usar um arquivo para definição e implementação. Módulos e assemblies do runtime sequer exigem o registro no sistema operacional. Como resultado, as descrições usadas pelo runtime sempre refletem o código real no arquivo compilado, o que aumenta a confiabilidade do aplicativo.

- Interoperabilidade de linguagem e facilidade de design com base em componentes.

  Os metadados fornecem todas as informações necessárias sobre código compilado para você herdar uma classe de um arquivo PE escrita em uma linguagem diferente. Você pode criar uma instância de qualquer classe escrita em qualquer linguagem gerenciada (qualquer linguagem que segmente o Common Language Runtime), sem se preocupar com o marshaling explícito ou com o uso de código de interoperabilidade personalizado.

- Atributos.

  O .NET Framework permite declarar tipos específicos de metadados, chamados atributos, no seu arquivo compilado. Os atributos podem ser encontrados em todo o .NET Framework e são usados para controlar mais detalhadamente como o seu programa se comporta no tempo de execução. Além disso, você pode emitir seus próprios metadados personalizados em arquivos do .NET Framework por meio de atributos definidos pelo usuário. Para obter mais informações, consulte Atributos.

Metadados e a estrutura de arquivos PE

Os metadados são armazenados em uma seção de um arquivo PE (Portable Executable) do .NET Framework, e o MSIL (Microsoft Intermediate Language) é armazenado em outra seção do arquivo PE. A parte de metadados do arquivo contém uma série de estruturas de tabela e de dados do heap. A parte MSIL contém tokens MSIL e de metadados que referenciam a parte de metadados do arquivo PE. Você pode encontrar tokens de metadados ao usar ferramentas como o MSIL Disassembler (Ildasm.exe) para exibir o MSIL de seu código, por exemplo.

Tabelas e heaps de metadados

Cada tabela de metadados contém informações sobre os elementos do seu programa. Por exemplo, uma tabela de metadados descreve as classes em seu código, outra descreve os campos e assim por diante. Se existirem dez classes em seu código, a tabela de classes terá dez linhas, uma para cada classe. Tabelas de metadados referenciam outras tabelas e heaps. Por exemplo, a tabela de metadados de classes referencia a tabela de métodos.

Metadados também armazenam informações em quatro estruturas de heap: cadeia de caracteres, blob, cadeia de caracteres de usuário e GUID. Todas as cadeias de caracteres usadas para nomear tipos e membros são armazenadas no heap da cadeia de caracteres. Por exemplo, uma tabela de métodos não armazena diretamente o nome de um método específico, mas aponta para o nome do método armazenado no heap da cadeia de caracteres.

Tokens de metadados

Cada linha de cada tabela de metadados é identificada com exclusividade na parte MSIL do arquivo PE por um token de metadados. Tokens de metadados são conceitualmente semelhantes a ponteiros, persistentes no MSIL, e referenciam uma tabela de metadados específica.

Um token de metadados é um número de quatro bytes. O byte superior denota a tabela de metadados a que um token específico se refere (método, tipo etc.). Os três bytes restantes especificam a linha na tabela de metadados que corresponde ao elemento de programação descrito. Se você definir um método em C# e compilá-lo em um arquivo PE, o seguinte token de metadados poderá existir na parte MSIL do arquivo PE:

```
0x06000004
```

O byte superior (`0x06`) indica que esse é um token **MethodDef**. Os três bytes inferiores (`000004`) pedem para o Common Language Runtime examinar na quarta linha da tabela **MethodDef** em busca das informações que descrevem essa definição de método.

Metadados em um arquivo PE

Quando um programa é compilado para o Common Language Runtime, ele é convertido em um arquivo PE que consiste em três partes. A tabela a seguir descreve o conteúdo de cada parte.

SEÇÃO PE	CONTEÚDO DA SEÇÃO PE
Cabeçalho PE	O índice das seções principais do arquivo PE e o endereço do ponto de entrada. O ambiente de execução usa essas informações para identificar o arquivo como um arquivo PE e determinar onde a execução começa ao carregar o programa na memória.
Instruções MSIL	As instruções MSIL que compõem o seu código. Muitas instruções MSIL são acompanhadas de tokens de metadados.
Metadados	Tabelas e heaps de metadados. O runtime usa esta seção para registrar informações todos os tipos e membros em seu código. Esta seção também inclui atributos personalizados e informações de segurança.

Uso de metadados em tempo de execução

Para compreender melhor os metadados e sua função no Common Language Runtime, pode ser útil criar um programa simples e ilustrar como os metadados afetam sua própria vida de tempo de execução. O exemplo de código a seguir mostra dois métodos dentro de uma classe chamada `MyApp`. O método `Main` é o ponto de entrada do programa, e o método `Add` apenas retorna a soma dos dois argumentos inteiros.

```
Public Class MyApp
   Public Shared Sub Main()
      Dim ValueOne As Integer = 10
      Dim ValueTwo As Integer = 20
      Console.WriteLine("The Value is: {0}", Add(ValueOne, ValueTwo))
   End Sub

   Public Shared Function Add(One As Integer, Two As Integer) As Integer
      Return (One + Two)
   End Function
End Class
```

```
using System;
public class MyApp
{
   public static int Main()
   {
      int ValueOne = 10;
      int ValueTwo = 20;
      Console.WriteLine("The Value is: {0}", Add(ValueOne, ValueTwo));
      return 0;
   }
   public static int Add(int One, int Two)
   {
      return (One + Two);
   }
}
```

Quando o código é executado, o runtime carrega o módulo na memória e consulta os metadados dessa classe. Quando carregado, o runtime executa uma análise abrangente do fluxo MSIL (Microsoft Intermediate Language) do método para convertê-lo em instruções de máquina rápidas nativas. O runtime usa um compilador JIT (just-in-time) para converter as instruções MSIL em código de máquina nativo, um método por vez, conforme necessário.

O exemplo a seguir mostra parte do MSIL produzido a partir da função `Main` do código anterior. Você pode exibir

o MSIL e os metadados de qualquer aplicativo .NET Framework usando o MSIL Disassembler (Ildasm.exe).

```
.entrypoint
.maxstack  3
.locals ([0] int32 ValueOne,
         [1] int32 ValueTwo,
         [2] int32 V_2,
         [3] int32 V_3)
IL_0000:  ldc.i4.s   10
IL_0002:  stloc.0
IL_0003:  ldc.i4.s   20
IL_0005:  stloc.1
IL_0006:  ldstr      "The Value is: {0}"
IL_000b:  ldloc.0
IL_000c:  ldloc.1
IL_000d:  call int32 ConsoleApplication.MyApp::Add(int32,int32) /* 06000003 */
```

O compilador JIT lê o MSIL para o método inteiro o analisa-o por completo, além de gerar instruções nativas eficientes para o método. No `IL_000d`, um token de metadados do método `Add` (`/*` `06000003 */`) é encontrado e o tempo de execução usa o token para consultar a terceira linha da tabela **MethodDef**.

A tabela a seguir mostra parte da tabela **MethodDef** referenciada pelo token de metadados que descreve o método `Add`. Embora haja outras tabelas de metadados nesse assembly e elas tenham seus próprios valores exclusivos, somente essa tabela é examinada.

LINHA	RVA (ENDEREÇO VIRTUAL RELATIVO)	IMPLFLAGS	SINALIZADORES	NAME (APONTA PARA O HEAP DA CADEIA DE CARACTERES.)	ASSINATURA (APONTA PARA O HEAP DE BLOB.)
1	0x00002050	IL Gerenciado	Público ReuseSlot SpecialName RTSpecialName .ctor	.ctor (construtor)	
2	0x00002058	IL Gerenciado	Público Estático ReuseSlot	Principal	Cadeia de Caracteres
3	0x0000208c	IL Gerenciado	Público Estático ReuseSlot	Adicionar	int, int, int

Cada coluna da tabela contém informações importantes sobre seu código. A coluna **RVA** permite que o tempo de execução calcule o endereço de memória inicial do MSIL que define esse método. As colunas **ImplFlags** e **Flags** contêm bitmasks que descrevem o método (por exemplo, se o método é público ou particular). A coluna **Nome** indexa o nome do método com base no heap da cadeia de caracteres. A coluna **Assinatura** indexa a definição da assinatura do método no heap de blob.

O tempo de execução calcula o endereço de deslocamento desejado com base na coluna **RVA** na terceira linha e

retorna esse endereço para o compilador JIT, que depois passa para o novo endereço. O compilador JIT continua processando o MSIL no novo endereço até encontrar outro token de metadados, e o processo é repetido.

Usando metadados, o ambiente de runtime tem acesso a todas as informações necessárias para carregar seu código e processá-lo em instruções de máquina nativas. Dessa maneira, os metadados permitem arquivos autodescritivos e, com o CTS, a herança entre linguagens.

Tópicos relacionados

TÍTULO	DESCRIÇÃO
Atributos	Descreve como aplicar atributos, escrever atributos personalizados e recuperar informações armazenadas em atributos.

Compilando aplicativos de console no .NET Framework

31/10/2019 • 3 minutes to read • Edit Online

Os aplicativos do .NET Framework podem usar a classe System.Console para ler e gravar caracteres no console. Os dados do console são lidos a partir do fluxo de entrada padrão, os dados para o console são gravados no fluxo de saída padrão e os dados de erro do console são gravados no fluxo de saída de erro padrão. Esses fluxos são automaticamente associados ao console quando o aplicativo é iniciado e são apresentados como as propriedades In, Out e Error, respectivamente.

O valor da propriedade Console.In é um objeto System.IO.TextReader, ao passo que os valores das propriedades Console.Out e Console.Error são objetos System.IO.TextWriter. Você pode associar essas propriedades com fluxos que não representam o console, tornando possível apontar para o fluxo em um local diferente de entrada ou saída. Por exemplo, você pode redirecionar a saída para um arquivo ao definir a propriedade Console.Out para um System.IO.StreamWriter que encapsula um System.IO.FileStream pelo método Console.SetOut. As propriedades Console.In e Console.Out não precisam fazer referência ao mesmo fluxo.

> **NOTE**
> Para mais informações sobre a compilação de aplicativos de console, incluindo exemplos em C #, Visual Basic e C++, consulte a documentação da classe Console.

Se o console não existir, como em um aplicativo baseado no Windows, a gravação de saída no fluxo de saída padrão não será visível, uma vez que não haverá nenhum console para gravar as informações. A gravação de informações em um console inacessível não gera uma exceção.

Como alternativa, para permitir que o console leia e grave em um aplicativo baseado no Windows que foi desenvolvido usando o Visual Studio, abra a caixa de diálogo **Propriedades** do projeto, clique na guia **Aplicativo** e defina **Tipo de aplicativo** como **Aplicativo de Console**.

Nos aplicativos de console falta uma bomba de mensagem iniciada por padrão. Portanto, as chamadas do console para os temporizadores do Microsoft Win32 podem falhar.

A classe **System.Console** tem métodos que podem ler caracteres individuais ou linhas inteiras do console. Outros métodos convertem dados e cadeias de formato e gravam as cadeias formatadas no console. Para mais informações sobre cadeias de caracteres de formatação, consulte Tipos de formatação.

Consulte também

- System.Console
- Formatando Tipos

Fundamentos do aplicativo .NET Framework

31/10/2019 • 4 minutes to read • Edit Online

Esta seção da documentação do .NET Framework oferece informações sobre tarefas básicas de desenvolvimento do aplicativo no .NET Framework.

Nesta seção

Tipos base
Discute os tipos de bancos de dados de formatação e análise e o uso de expressões regulares para processar o texto.

Coleções e Estruturas de Dados
Discute os vários tipos de coleção disponíveis no .NET Framework, incluindo pilhas, filas, listas, matrizes e structs.

Genéricos
Descreve o recurso Genéricos, incluindo coleções, interfaces e representantes genéricos fornecidos pelo .NET Framework. Fornece links à documentação de recursos para C#, Visual Basic e Visual C++ e a tecnologias de suporte como Reflection.

Numéricos
Descreve os tipos numéricos no .NET Framework.

Eventos
Apresenta uma visão geral do modelo de evento no .NET Framework.

Exceções
Descreve o tratamento de erro oferecido pelo .NET Framework e os fundamentos do gerenciamento de exceções.

E/S de arquivo e de fluxo
Explica como você pode realizar o acesso a fluxo de arquivos e dados de maneira síncrona e assíncrona, e como usar armazenamento isolado.

Datas, horas e fusos horários
Descreve como trabalhar com fusos horários e conversões de fusos horários em aplicativos com distinção de fusos horários.

Domínios do aplicativo e assemblies
Descreve como criar e trabalhar com assemblies e domínios de aplicativos.

Serialização
Discute o processo de conversão do estado de um objeto em formato que possa ser persistido ou transportado.

Recursos em aplicativos de área de trabalho
Descreve o suporte do .NET Framework para a criação e o armazenamento de recursos. Esta seção também descreve o suporte a recursos localizados e o modelo de recursos de assembly satélite para embalar e implantar esses recursos localizados.

Globalização e localização
Oferece informações para ajudar você a projetar e desenvolver aplicativos prontos para uso.

Acessibilidade Fornece informações sobre a automação da interface do usuário da Microsoft, que é uma estrutura de acessibilidade que aborda as necessidades de produtos de tecnologia assistencial e estruturas de teste automatizadas, fornecendo acesso programático a informações sobre a interface do usuário.

Atributos
Descreve como usar atributos para personalizar metadados.

Aplicativos de 64 bits
Discute questões relevantes ao desenvolvimento de aplicativos que serão executados no sistema operacional Windows de 64 bits.

Seções relacionadas

Guia de desenvolvimento
Fornece um guia para todas as principais áreas de tecnologia e tarefas para o desenvolvimento de aplicativos, incluindo a criação, a configuração, a depuração, a proteção e a implantação de seu aplicativo, bem como informações sobre programação dinâmica, interoperabilidade, extensibilidade, gerenciamento de memória e threading.

Security
Oferece informações sobre classes e serviços no Command Language Runtime e no .NET Framework que facilitam um desenvolvimento seguro do aplicativo.

E/S de arquivo e de fluxo

04/12/2019 • 16 minutes to read • Edit Online

E/S (entrada/saída) de arquivos e fluxos refere-se à transferência de dados de ou para uma mídia de armazenamento. No .NET Framework, os namespaces `System.IO` contêm tipos que permitem a leitura e a gravação, de forma síncrona e assíncrona, em fluxos de dados e arquivos. Esses namespaces também contêm tipos que executam compactação e descompactação em arquivos e tipos que possibilitam a comunicação por meio de pipes e portas seriais.

Um arquivo é uma coleção ordenada e nomeada de bytes com armazenamento persistente. Ao trabalhar com arquivos, você trabalha com caminhos de diretórios, armazenamento em disco e nomes de arquivos e diretórios. Por outro lado, um fluxo é uma sequência de bytes que você pode usar para ler e gravar em um repositório, o qual pode ser uma entre vários tipos de mídia de armazenamento (por exemplo, discos ou memória). Assim como há vários repositórios diferentes de discos, há vários tipos diferentes de fluxos diferentes de fluxos de arquivos, como os fluxos de rede, memória e pipes.

Arquivos e diretórios

Você pode usar os tipos no namespace System.IO para interagir com arquivos e diretórios. Por exemplo, você pode obter e definir propriedades para arquivos e diretórios e recuperar coleções de arquivos e diretórios com base em critérios de pesquisa.

Para convenções de nomenclatura de caminhos e os modos de expressar um caminho de arquivo para sistemas Windows, incluindo a sintaxe de dispositivo DOS compatível com o .NET Core 1.1 e posterior e o .NET Framework 4.6.2 e posterior, veja Formatos de caminho de arquivo em sistemas Windows.

Aqui estão algumas classes de arquivos e diretórios comumente usadas:

- File – Fornece métodos estáticos para criar, copiar, excluir, mover e abrir arquivos, além de ajudar na criação de um objeto FileStream.
- FileInfo – Fornece métodos de instâncias para criar, copiar, excluir, mover e abrir arquivos, além de ajudar na criação de um objeto FileStream.
- Directory – Fornece métodos estáticos para criar, mover e enumerar ao longo de diretórios e subdiretórios.
- DirectoryInfo – Fornece métodos de instância para criar, mover e enumerar ao longo de diretórios e subdiretórios.
- Path – Fornece métodos e propriedades para processar cadeias de caracteres de diretório de uma maneira compatível com várias plataformas.

Você sempre deve fornecer tratamento de exceção robusto ao chamar métodos de sistema de arquivos. Para obter mais informações, veja Tratamento de erros de E/S.

Além de usar essas classes, os usuários do Visual Basic podem usar os métodos e as propriedades fornecidas pela classe Microsoft.VisualBasic.FileIO.FileSystem para E/S de arquivo.

Confira Como copiar diretórios, Como criar uma listagem de diretórios e Como enumerar diretórios e arquivos.

Fluxos

A classe base abstrata Stream oferece suporte a leitura e gravação de bytes. Todas as classes que representam

fluxos herdam da classe Stream. A classe Stream e suas classes derivadas fornecem uma visão comum de fontes e repositórios de dados, isolando o programador de detalhes específicos do sistema operacional e dispositivos subjacentes.

Fluxos envolvem estas três operações fundamentais:

- Leitura – Transferência de dados de um fluxo para uma estrutura de dados, como uma matriz de bytes.
- Gravação – Transferência de dados para um fluxo a partir de uma fonte de dados.
- Busca – Consulta e modificação da posição atual em um fluxo.

Dependendo do repositório ou da fonte de dados subjacente, os fluxos podem oferecer suporte somente algumas dessas capacidades. Por exemplo, a classe PipeStream não oferece suporte à operação de busca. As propriedades CanRead, CanWrite e CanSeek de um fluxo especificam as operações às quais o fluxo oferece suporte.

Algumas classes de fluxo comumente usadas são:

- FileStream – Para leitura e gravação em um arquivo.
- IsolatedStorageFileStream – Para leitura e gravação em um arquivo no armazenamento isolado.
- MemoryStream – Para leitura e gravação na memória como o repositório de backup.
- BufferedStream – Para melhorar o desempenho das operações de leitura e gravação.
- NetworkStream – Para leitura e gravação via soquetes de rede.
- PipeStream – Para leitura e gravação sobre pipes anônimos e nomeados.
- CryptoStream – Para vincular fluxos de dados a transformações criptográficas.

Para um exemplo de como trabalhar com fluxos de forma assíncrona, confira E/S de arquivo assíncrona.

Leitores e gravadores

O namespace System.IO também fornece tipos usados para ler caracteres codificados de fluxos e gravá-los em fluxos. Normalmente, os fluxos são criados para a entrada e a saída de bytes. Os tipos de leitor e de gravador tratam a conversão dos caracteres codificados de/para bytes para que o fluxo possa concluir a operação. Cada classe de leitor e gravador é associada a um fluxo, o qual pode ser recuperado pela propriedade `BaseStream` da classe.

Algumas classes de leitores e gravadores comumente usadas são:

- BinaryReader e BinaryWriter – Para leitura e gravação de tipos de dados primitivos como valores binários.
- StreamReader e StreamWriter – Para leitura e gravação de caracteres usando um valor de codificação para converter os caracteres para/de bytes.
- StringReader e StringWriter – Para leitura e gravação de caracteres e cadeias de caracteres.
- TextReader e TextWriter – Funcionam como as classes base abstratas para outros leitores e gravadores que leem e gravam caracteres e cadeias de caracteres, mas não dados binários.

Confira Como ler texto de um arquivo, Como gravar texto em um arquivo, Como ler caracteres em uma cadeia de caracteres e Como gravar caracteres em uma cadeia de caracteres.

Operações de E/S assíncronas

A leitura ou gravação de uma grande quantidade de dados pode consumir muitos recursos. Você deve executar essas tarefas de forma assíncrona se seu aplicativo precisar continuar respondendo ao usuário. Com as operações

de E/S síncronas, o thread de interface do usuário é bloqueado até que a operação de uso intensivo seja concluída. Use operações de e/s assíncronas ao desenvolver aplicativos da loja do Windows 8. x para evitar a criação da impressão de que seu aplicativo parou de funcionar.

Os membros assíncronas contêm `Async` em seus nomes, como os métodos CopyToAsync, FlushAsync, ReadAsync e WriteAsync. Você usa esses métodos com as palavras-chave `async` e `await`.

Para saber mais, confira E/S de arquivo assíncrona.

Compactação

A compactação refere-se ao processo de reduzir o tamanho de um arquivo para fins de armazenamento. A descompactação é o processo de extrair o conteúdo de um arquivo compactado para um formato utilizável. O namespace System.IO.Compression contém tipos para compactar e descompactar arquivos e fluxos.

As classes a seguir são frequentemente usadas para compactar e descompactar arquivos e fluxos:

- ZipArchive – Para criar e recuperar entradas em arquivos ZIP.
- ZipArchiveEntry – Para representar um arquivo compactado.
- ZipFile – Para criar, extrair e abrir um pacote compactado.
- ZipFileExtensions – Para criar e extrair entradas em um pacote compactado.
- DeflateStream – Para compactar e descompactar fluxos usando o algoritmo de Deflate.
- GZipStream – Para compactar e descompactar fluxos no formato de dados GZIP.

Confira How to: Compress and Extract Files (Como compactar e extrair arquivos).

Armazenamento isolado

Um armazenamento isolado é um mecanismo de armazenamento de dados que fornece isolamento e segurança ao definir maneiras padronizadas de associar códigos a dados salvos. O armazenamento fornece um sistema de arquivos virtual que é isolado por usuário, assembly e (opcionalmente) domínio. O armazenamento isolado é particularmente útil quando o aplicativo não tem permissão para acessar arquivos de usuários. Você pode salvar configurações ou arquivos para seu aplicativo de modo que ele seja controlado pela política de segurança do computador.

O armazenamento isolado não está disponível para aplicativos da loja do Windows 8. x; em vez disso, use classes de dados de aplicativo no namespace Windows.Storage. Para saber mais, veja Dados de aplicativo.

As classes a seguir são usadas com frequência na implementação do armazenamento isolado:

- IsolatedStorage – Fornece a classe base para implementações de armazenamento isolado.
- IsolatedStorageFile – Fornece uma área de armazenamento isolado que contém arquivos e diretórios.
- IsolatedStorageFileStream – Expõe um arquivo no armazenamento isolado.

Confira Armazenamentos isolado.

Operações de E/S em aplicativos da Windows Store

O .NET para aplicativos da loja do Windows 8. x contém muitos dos tipos de leitura e gravação em fluxos; no entanto, esse conjunto não inclui todos os tipos de e/s de .NET Framework.

Algumas diferenças importantes a serem observadas ao usar operações de e/s em aplicativos da loja do Windows 8. x:

- Tipos especificamente relacionados a operações de arquivo, como File, FileInfo, Directory e DirectoryInfo, não estão incluídos no .NET para aplicativos da loja do Windows 8. x. Em vez disso, use os tipos no namespace Windows.Storage do Windows Runtime, como StorageFile e StorageFolder.
- O armazenamento isolado não está disponível. Use dados de aplicativo.
- Use métodos assíncronos, como ReadAsync e WriteAsync, para evitar o bloqueio do thread da interface do usuário.
- Os tipos de compactação com base em caminhos ZipFile e ZipFileExtensions não estão disponíveis. Em vez disso, use os tipos no namespace Windows.Storage.Compression.

É possível converter entre fluxos do .NET Framework e fluxos do Windows Runtime, se necessário. Para obter mais informações, consulte como converter entre fluxos de .NET Framework e fluxos de Windows Runtime ou WindowsRuntimeStreamExtensions.

Para obter mais informações sobre operações de e/s em um aplicativo da loja do Windows 8. x, consulte início rápido: lendo e gravando arquivos.

E/S e segurança

Ao usar as classes no namespace System.IO, você deve atender aos requisitos de segurança do sistema operacional, como ACLs (listas de controle de acesso) para controlar o acesso a arquivos e diretórios. Esse é um requisito adicional aos requisitos de FileIOPermission. As ACLs podem ser gerenciadas por meio de programação. Para saber mais, confira Como adicionar ou remover entradas da lista de controle de acesso.

As políticas de segurança padrão impedem que aplicativos da Internet ou intranet acessem arquivos no computador do usuário. Consequentemente, não use classes de E/S que exijam um caminho para um arquivo físico ao escrever código que será baixado via Internet ou intranet. Em vez disso, use o armazenamento isolado para aplicativos .NET Framework tradicionais ou use dados de aplicativos para aplicativos da loja do Windows 8. x.

Uma verificação de segurança é executada somente quando o fluxo é construído. Consequentemente, não abra um fluxo para depois passá-lo para código ou domínios de aplicativos menos confiáveis.

Tópicos relacionados

- Tarefas comuns de E/S
  Fornece uma lista das tarefas de E/S associadas a arquivos, diretórios e fluxos, além de links para conteúdo e exemplos relevantes para cada tarefa.
- E/S de arquivo assíncrona
  Descreve as vantagens de desempenho e a operação básica da E/S assíncrona.
- Armazenamento isolado
  Descreve um mecanismo de armazenamento isolado que fornece isolamento e segurança ao definir maneiras padronizadas de associar códigos aos dados salvos.
- Pipes
  Descreve operações de pipes anônimos e nomeados no .NET Framework.
- Arquivos mapeados em memória
  Descreve arquivos mapeados na memória, os quais armazenam o conteúdo de arquivos do disco na memória virtual. Você pode usar arquivos mapeados na memória para editar arquivos muito grandes e para criar memória compartilhada para a comunicação entre processos.

Globalizando e localizando aplicativos do .NET

31/10/2019 • 5 minutes to read • Edit Online

O desenvolvimento de um aplicativo que possa ser usado em todo o mundo, incluindo ser localizado em um ou mais idiomas, envolve três etapas: a globalização, a análise da possibilidade de localização e a localização em si.

Globalização

Esta etapa envolve a criação e a codificação de um aplicativo que seja independente de cultura e idioma, bem como que ofereça suporte a interfaces de usuário localizadas e dados regionais para todos os usuários. Ela envolve a tomada de decisões de design e de programação que não sejam baseadas em suposições para culturas específicas. Mesmo quando um aplicativo globalizado não está localizado, ele foi criado e escrito para que possa ser localizado posteriormente em um ou mais idiomas com relativa facilidade.

Análise de possibilidade de localização

Essa etapa envolve a verificação do código e do design de um aplicativo para garantir que ele possa ser facilmente localizado e para identificar potenciais obstáculos à localização, bem como verificar se o código executável do aplicativo está separado de seus recursos. Se a fase de globalização foi eficaz, a análise de capacidade de localização confirmará as opções de design e codificação feitas durante a globalização. O estágio de localizabilidade também pode identificar todos os problemas restantes para que o código-fonte do aplicativo não precise ser modificado durante o estágio de localização.

Localização

Essa etapa envolve a personalização de um aplicativo para culturas ou regiões específicas. Se as etapas de globalização e localizabilidade tiverem sido feitas corretamente, a localização consistirá principalmente na tradução da interface de usuário.

Seguir estas três etapas oferece duas vantagens:

- Libera você de ter que readaptar um aplicativo projetado para oferecer suporte a uma única cultura, como inglês dos EUA, para oferecer suporte a culturas adicionais.
- Isso resulta em aplicativos localizados que são mais estáveis e possuem menos bugs.

O .NET fornece suporte abrangente ao desenvolvimento de aplicativos preparados para o mundo e localizados. Em particular, muitos membros de tipos na biblioteca de classes do .NET auxiliam na globalização ao retornarem valores que refletem as convenções da cultura do usuário atual ou de uma cultura específica. Além disso, o .NET Framework dá suporte a assemblies satélites que facilitam o processo de localizar um aplicativo.

Confira mais informações na Documentação de globalização.

Nesta seção

Globalização

Discute o primeiro estágio da criação de um aplicativo pronto para o mundo, o que envolve o projeto e a codificação de um aplicativo independente de cultura e idioma.

Análise de possibilidade de localização

Discute o segundo estágio da criação de um aplicativo localizado, o que envolve a identificação de obstáculos potenciais à localização.

Localização

Discute o estágio final da criação de um aplicativo localizado, o que envolve a personalização da interface de usuário de um aplicativo para regiões ou culturas específicas.

Operações de cadeia de caracteres sem diferenciação de cultura

Descreve como usar métodos e classes do .NET sensíveis a culturas por padrão para obter resultados sem diferenciação de cultura.

Práticas recomendadas para o desenvolvimento de aplicativos prontos para o mundo

Descreve as práticas recomendadas a serem seguidas para globalização, localização e desenvolvimento de aplicativos ASP.NET prontos para o mundo.

Referência

- Namespace System.Globalization

  Contém classes que definem informações relacionadas à cultura, incluindo idioma, país/região, calendários em uso, padrões de formato para datas, moeda, números e ordem de classificação para cadeias de caracteres.

- Namespace System.Resources

  Fornece classes para a criação, manipulação e utilização de recursos.

- Namespace System.Text

  Contém classes que representam ASCII, ANSI, Unicode e outras codificações de caracteres.

- Resgen.exe (Gerador de Arquivo de Recurso)

  Descreve como usar Resgen.exe para converter arquivos .txt e XML (.resx) para arquivos .resources binários do Common Language Runtime.

- Winres.exe (Editor de Recursos do Windows Forms)

  Descreve como usar Winres.exe para localizar formulários do Windows Forms.

Estendendo metadados por meio de atributos

31/10/2019 • 2 minutes to read • Edit Online

O Common Language Runtime permite adicionar declarações descritivas parecidas com palavras, chamadas atributos, para anotar elementos de programação como tipos, campos, métodos e propriedades. Quando você compila seu código para o runtime, ele é convertido em MSIL (Microsoft Intermediate Language) e colocado dentro de um arquivo PE (executável portátil) com metadados gerados pelo compilador. Os atributos permitem colocar informações descritivas extras em metadados que podem ser extraídos usando serviços de reflexão de runtime. O compilador cria atributos quando você declara instâncias de classes especiais que derivam de System.Attribute.

O .NET Framework usa atributos por vários motivos e para solucionar vários problemas. Os atributos descrevem como serializar dados, especificar características que são usadas para impor segurança e limitar otimizações pelo compilador JIT (Just-In-Time) para que o código permaneça fácil de depurar. Os atributos também podem registrar o nome de um arquivo ou o autor do código, ou controlar a visibilidade de controles e membros durante o desenvolvimento de formulários.

Tópicos relacionados

TÍTULO	DESCRIÇÃO
Aplicando atributos	Descreve como aplicar um atributo a um elemento do código.
Escrevendo atributos personalizados	Descreve como criar classes de atributos personalizados.
Recuperando informações armazenadas em atributos	Descreve como recuperar atributos personalizados para o código que é carregado no contexto de execução.
Metadados e componentes autodescritivos	Fornece uma visão geral dos metadados e descreve como eles são implementados em um arquivo executável PE (executável portátil) do .NET Framework.
Como carregar assemblies no contexto somente reflexão	Explica como recuperar informações de atributos personalizados no contexto somente reflexão.

Referência

System.Attribute

Diretrizes de design de estrutura

23/10/2019 • 3 minutes to read

Esta seção fornece diretrizes para a criação de bibliotecas que estendem e interagem com o .NET Framework. O objetivo é ajudar os designers de bibliotecas a garantir a consistência de API e facilidade de uso, fornecendo um modelo de programação unificado que é independente da linguagem de programação usada para o desenvolvimento. É recomendável que você siga estas diretrizes de design ao desenvolver classes e componentes que estendam o .NET Framework. Design de bibliotecas inconsistente negativamente afeta a produtividade do desenvolvedor e desencoraja a adoção.

As diretrizes são organizadas como recomendações simples prefixadas com os termos `Do`, `Consider`, `Avoid`, e `Do not`. Estas diretrizes destinam-se a ajudar os designers de biblioteca de classes a entender as compensações entre as diferentes soluções. Pode haver situações em que o design de boa biblioteca requer que você viole essas diretrizes de design. Nesses casos, devem ser raros, e é importante que você tenha um motivo claro e convincente para sua decisão.

Estas diretrizes foram extraídas do livro *as diretrizes de Design do Framework: As convenções, linguagens e padrões para bibliotecas do .NET reutilizável, 2nd Edition*, por Krzysztof Cwalina e Brad Abrams.

Nesta seção

Diretrizes de nomenclatura
Fornece diretrizes para a nomeação de assemblies, namespaces, tipos e membros em bibliotecas de classes.

Diretrizes de Design de tipo
Fornece diretrizes para usar classes abstratas e estáticas, interfaces, enumerações, estruturas e outros tipos.

Diretrizes de design de membro
Fornece diretrizes para criação e uso de propriedades, métodos, construtores, campos, eventos, operadores e parâmetros.

Designer voltado para extensibilidade
Discute os mecanismos de extensibilidade, como a criação de subclasses, usando eventos, os membros virtuais e retornos de chamada e explica como escolher os mecanismos que melhor atendem aos requisitos da sua estrutura.

Diretrizes de design para exceções
Descreve as diretrizes de design para projetar, lançar e capturar exceções.

Diretrizes de uso
Descreve diretrizes para usando tipos comuns, como matrizes, atributos e coleções, suporte à serialização e sobrecarga de operadores de igualdade.

Padrões comuns de Design
Fornece diretrizes para escolher e implementar as propriedades de dependência.

Consulte também

- Visão geral
- Guia de desenvolvimento

Documentos e dados XML

04/11/2019 • 6 minutes to read • Edit Online

O .NET Framework fornece um conjunto de classes abrangente e integrado que permite que você crie aplicativos com reconhecimento de XML facilmente. As classes nos namespaces a seguir suportam a análise e gravação de XML, edição de dados XML na memória, validação de dados e transformação de XSLT.

- System.Xml
- System.Xml.XPath
- System.Xml.Xsl
- System.Xml.Schema
- System.Xml.Linq

Para obter uma lista completa, pesquise "System.Xml" no navegador da API .NET.

As classes nesses namespaces suportam recomendações World Wide Web Consortium (W3C). Por exemplo:

- A classe System.Xml.XmlDocument implementa as recomendações do DOM (Modelo de Objeto do Documento) Core do W3C nível 1 e do DOM Core nível 2.
- As classes System.Xml.XmlReader e System.Xml.XmlWriter dão suporte para as recomendações W3C XML 1.0 e Namespaces em XML.
- Os esquemas na classe System.Xml.Schema.XmlSchemaSet dão suporte para as recomendações do W3C XML Schema Part 1: Structures (Esquema XML do W3C Parte 1: estruturas) e do XML Schema Part 2: Datatypes (Esquema XML Parte 2: tipos de dados).
- As classes no namespace System.Xml.Xsl dão suporte para as transformações XSLT que estão em conformidade com a recomendação W3C XSLT 1.0.

As classes XML do .NET Framework fornecem esses benefícios:

- **Produtividade.** O LINQ to XML (C#) e o LINQ to XML (Visual Basic) facilitam a programação com XML e fornecem uma experiência de consulta que é semelhante ao SQL.
- **Extensibilidade.** As classes XML no .NET Framework são extensíveis pelo uso de classes base abstratas e métodos virtuais. Por exemplo, você pode criar uma classe derivada da classe de XmlUrlResolver que armazena o fluxo de cache no disco local.
- **Arquitetura conectável.** O .NET Framework fornece uma arquitetura na qual os componentes podem se utilizar uns aos outros e os dados podem ser transmitidos entre os componentes. Por exemplo, um armazenamento de dados, tal como um objeto XPathDocument ou XmlDocument, pode ser transformado com a classe XslCompiledTransform e a saída pode então ser transmitida para outro armazenamento ou retornados como um fluxo de um serviço da web.
- **Desempenho.** Para melhorar o desempenho do aplicativo, algumas das classes XML do .NET Framework dão suporte a um modelo baseado em streaming com as seguintes características:
  - Armazenamento em cache mínimo para somente encaminhamento, análise de recepção modelo (XmlReader).
  - Validação somente de encaminhamento (XmlReader).

- Navegação do cursor que minimiza a criação do nó com um único nó virtual no entanto fornece acesso aleatório ao documento (XPathNavigator).

  Para obter um melhor desempenho sempre que o processamento de XSLT for necessário, você pode usar a classe XPathDocument, que é um repositório otimizado, somente de leitura para consultas do XPath projetado para trabalhar com eficiência com a classe XslCompiledTransform.

- **Integração com o ADO.NET.** As classes XML e o ADO.NET são bem integrados para reunir dados relacionais e XML. A classe de DataSet é um cache de memória dos dados recuperados de uma base de dados. A classe DataSet tem a capacidade de ler e escrever XML usando as classes de XmlReader e de XmlWriter , para manter sua estrutura de esquema relacional como esquemas XML (XSD), e para interpretar a estrutura do esquema de um documento XML.

Nesta seção

Opções de processamento XML Discute opções para processar dados XML.

Processando dados XML na memória Discute os três modelos para processamento de dados XML na memória: LINQ to XMLC#() e LINQ to XML (Visual Basic), a classe XmlDocument (com base na modelo de objeto do documento W3C) e a classe XPathDocument (com base no modelo de dados XPath).

XSLT Transformations
Descreve como usar o processador XSLT.

SOM (modelo de objeto de esquema) XML
Descreve as classes usadas para criar e manipular esquemas XML (XSD), fornecendo uma classe XmlSchema para carregar e editar um esquema.

Integração XML com dados relacionais e o ADO.NET
Descreve como o .NET Framework habilita o acesso síncrono, em tempo real, às representações de dados relacionais e hierárquicas através dos objetos DataSet e XmlDataDocument.

Gerenciando namespaces em um documento XML
Descreve como a classe XmlNamespaceManager classe é usada para armazenar e manter as informações do namespace.

Digite suporte nas classes de System.Xml
Descreve como mapa de tipos de dados XML para tipos de CLR, como converter tipos de dados XML e outros recursos de suporte de tipo nas classes System.Xml.

Seções relacionadas

ADO.NET
Fornece informações sobre como acessar dados usando ADO.NET.

Segurança
Fornece uma visão geral do sistema de segurança do .NET Framework.

Segurança no .NET

24/10/2019 • 2 minutes to read • Edit Online

O Common Language Runtime e o .NET fornecem muitas classes e serviços úteis que permitem aos desenvolvedores escrever facilmente código seguro e permitir que os administradores do sistema personalizem as permissões concedidas ao código para que ele possa acessar recursos protegidos. Além disso, o tempo de execução e o .NET fornecem classes e serviços úteis que facilitam o uso da criptografia e da segurança baseada em funções.

Nesta seção

- Principais conceitos de segurança
  Fornece uma visão geral dos recursos de segurança do Common Language Runtime. Esta seção é de interesse dos desenvolvedores e administradores de sistema.
- Segurança baseada em Função
  Descreve como interagir com a segurança baseada em função no seu código. Esta seção é de interesse dos desenvolvedores.
- Modelo de criptografia
  Fornece uma visão geral dos serviços de criptografia fornecidos pelo .NET. Esta seção é de interesse dos desenvolvedores.
- Diretrizes de codificação segura
  Descreve algumas das práticas recomendadas para a criação de aplicativos .NET confiáveis. Esta seção é de interesse dos desenvolvedores.

Seções relacionadas

Guia de desenvolvimento
Fornece um guia para todas as principais áreas de tecnologia e tarefas para o desenvolvimento de aplicativos, incluindo a criação, a configuração, a depuração, a proteção e a implantação de seu aplicativo, bem como informações sobre programação dinâmica, interoperabilidade, extensibilidade, gerenciamento de memória e threading.

Serialização no .NET

23/10/2019 • 2 minutes to read • Edit Online

A serialização é o processo de conversão do estado de um objeto em um formulário que possa ser persistido ou transportado. O complemento de serialização é desserialização, que converte um fluxo em um objeto. Juntos, esses processos permitem que os dados sejam armazenados e transferidos.

O .NET apresenta as seguintes tecnologias de serialização:

- A serialização binária preserva a fidelidade do tipo, que é útil para preservar o estado de um objeto entre diferentes invocações de um aplicativo. Por exemplo, você pode compartilhar um objeto entre diferentes aplicativos serializando-o para a área de transferência. Você pode serializar um objeto para um fluxo, um disco, a memória, pela rede, e assim por diante. O acesso remoto usa a serialização para passar objetos "por valor" de um computador ou domínio de aplicativo para outro.
- A serialização de XML e SOAP serializa apenas campos e propriedades públicas e não preserva a fidelidade do tipo. Isso é útil quando você deseja fornecer ou consumir dados sem restringir o aplicativo que usa os dados. Como o XML é um padrão aberto, é uma opção atrativa para compartilhar dados pela Web. SOAP é, da mesma forma, um padrão aberto, uma opção atrativa.
- A serialização JSON serializa apenas as propriedades públicas e não preserva a fidelidade do tipo. JSON é um padrão aberto que é uma opção atraente para compartilhar dados na Web.

Referência

System.Runtime.Serialization
Contém classes que podem ser usadas para serialização e desserialização de objetos.

System.Xml.Serialization
Contém classes que podem ser usadas para serializar objetos em documentos ou fluxos de formato XML.

System.Text.Json
Contém classes que podem ser usadas para serializar objetos em fluxos ou documentos de formato JSON.

Desenvolvendo para várias plataformas com o .NET Framework

23/10/2019 • 6 minutes to read • Edit Online

Você pode desenvolver aplicativos para plataformas Microsoft e não Microsoft usando o .NET Framework e o Visual Studio.

Opções para o desenvolvimento de plataforma cruzada

> **IMPORTANT**
> Porque os projetos de biblioteca de classes portátil destinam-se apenas um subconjunto muito específico de implementações do .NET, nós não recomendamos seu uso em desenvolvimento de novos aplicativos. A substituição recomendada é uma biblioteca .NET Standard, que se destina a todas as implementações do .NET que dão suporte a uma versão específica do .NET Standard. Para obter mais informações, confira .NET Standard.

Para desenvolver várias plataformas, é possível compartilhar código-fonte ou binários, além de fazer chamadas entre o código do .NET Framework e as APIs do Tempo de Execução do Windows.

SE DESEJAR...	USE...
Compartilhar código-fonte entre os aplicativos do Windows Phone 8.1 e do Windows 8.1	**Projetos compartilhados** (modelo de aplicativos universais no Visual Studio 2013, atualização 2). -No momento, não há suporte do Visual Basic. -Você pode separar o código específico da plataforma usando # `if` instruções. Para obter detalhes, consulte: - Comece a codificar - Usando o Visual Studio para compilar aplicativos universais do XAML (postagem de blog) - Usando o Visual Studio para compilar aplicativos convergentes do XAML (vídeo)

SE DESEJAR...	USE...
Compartilhar binários entre aplicativos destinados a plataformas diferentes	**Projetos de biblioteca de classes portátil** para o código que é independente de plataforma. -Essa abordagem geralmente é usada para o código que implementa a lógica de negócios. -Você pode usar o Visual Basic ou c#. -Suporte API varia de acordo com a plataforma. -Projetos de biblioteca de classe portátil destinados ao Windows 8.1 e Windows Phone 8.1 dão suporte a APIs do Windows Runtime e XAML. Esses recursos não estão disponíveis em versões anteriores da Biblioteca de Classes Portátil. -Se necessário, você poderá abstrair o código específico da plataforma usando interfaces ou classes abstratas. Para obter detalhes, consulte: - Biblioteca de classes portátil - Como fazer portáteis trabalharem de bibliotecas de classe para você (postagem de blog) - Usando a biblioteca de classes portátil com o MVVM - Recursos do aplicativo para bibliotecas direcionadas a várias plataformas - .NET portability Analyzer (extensão do Visual Studio)
Compartilhar código-fonte entre aplicativos para plataformas diferentes do Windows 8.1 e do Windows Phone 8.1	**Adicionar como vínculo** recurso. -Essa abordagem é adequada para a lógica de aplicativo que é comum aos dois aplicativos, mas não portáteis, por algum motivo. É possível usar esse recurso para o código do C# ou do Visual Basic. Por exemplo, o Windows Phone 8 e o Windows 8 compartilham as APIs do Tempo de Execução do Windows, mas as Bibliotecas de Classes Portáteis não oferecem suporte ao Tempo de Execução do Windows para essas plataformas. É possível usar `Add as link` para compartilhar o código do Tempo de Execução do Windows entre um aplicativo do Windows Phone 8 e um aplicativo da Windows Store destinado ao Windows 8. Para obter detalhes, consulte: - Compartilhar código com Adicionar como Link - Como: Adicionar itens existentes a um projeto
Gravar aplicativos da Windows Store usando o .NET Framework ou chamar APIs do Tempo de Execução do Windows do código do .NET Framework	**APIs do Windows Runtime** do seu código .NET Framework c# ou Visual Basic e usar o .NET Framework para criar aplicativos da Windows Store. Você deve saber as diferenças de API entre as duas plataformas. Porém, existem classes que ajudam a trabalhar com essas diferenças. Para obter detalhes, consulte: - Suporte do .NET framework para aplicativos da Windows Store e o tempo de execução do Windows - Passando um URI para o tempo de execução do Windows - WindowsRuntimeStreamExtensions - WindowsRuntimeSystemExtensions

SE DESEJAR...	USE...
Compilar aplicativos do .NET Framework para plataformas que não sejam da Microsoft	**Assemblies de referência de biblioteca de classes portátil** no .NET Framework e uma ferramenta de terceiros ou extensão do Visual Studio, como Xamarin. Para obter detalhes, consulte: - Biblioteca de classes portátil agora disponível em todas as plataformas. (publicação de blog) - Documentação do Xamarin
Usar JavaScript e HTML para desenvolvimento de plataforma cruzada	**Modelos de aplicativo universal** no Visual Studio 2013, atualização 2 para desenvolver com APIs do Windows Runtime para Windows 8.1 e Windows Phone 8.1. Atualmente, não é possível usar JavaScript e HTML com APIs do .NET Framework para desenvolver aplicativos de plataforma cruzada. Para obter detalhes, consulte: - Modelos de projeto de JavaScript - Portabilidade de um aplicativo de tempo de execução do Windows usando JavaScript para Windows Phone