**IMPLEMENTAR ACESSO AOS DADOS (25–30%)**

Realizar operações de E/S

* Ler e escrever arquivos e fluxos; ler e escrever a partir da rede usando classes no namespace System.Net; implementar operações de E/S assíncronas

Operações E/S de Arquivo e Diretórios

Interagir com arquivos é uma tarefa comum ao desenvolver um aplicativo, as operações de E/S se referem à leitura e gravação de arquivos de e para armazenamento. Os arquivos são armazenados em diretórios e o .NET Framework fornece um conjunto de classes para copiar, mover, excluir ou verificar a existência de arquivos ou diretórios. Um arquivo é uma coleção de bytes ordenada e nomeada que foi salva no armazenamento. Ao trabalhar com arquivos, você usa um fluxo (Stream). Um fluxo é um objeto na memória usado para representar a sequência de bytes em um arquivo. As classes especiais de File Reader e File Writer permitem trabalhar com fluxos codificados que são úteis ao armazenar algum tipo de informação em arquivos, armazenar essas informações em um formato diferente (por exemplo, binário ou texto), ou quando você precisa enviar ou acessar algum tipo de dados pela rede.

Esta sessão abordará os principais conceitos usados para interagir com o sistema de arquivos e fornecerá um entendimento do funcionamento dos seguintes itens:

1. Drives e diretórios
2. Arquivos e Streams(fluxos)
3. Interação com arquivos remotos
4. E/S de arquivo assíncrono

O .NET Framework fornece as classes para interagir com uma E/S de arquivo que pode ser encontrada no namespace System.IO. Este namespace é a coleção das classes base dedicadas a serviços de entrada e saída baseados em arquivos e memória.

Trabalhando com o Drive

DriveInfo é uma classe fornecida pelo .NET no System.IO, usada para interagir com uma mídia de armazenamento, que pode ser um disco rígido ou qualquer outro armazenamento (ou seja, disco removível). Esta classe fornece informações sobre as unidades, como nome, tamanho e espaço livre da unidade. Você também pode conhecer quais unidades estão disponíveis e que tipo elas são.

Dentro do construtor do DriveInfo, o nome do Drive é passado e você pode acessar suas informações. Você também pode obter todas as unidades e buscar seus detalhes usando o método GetDrives () (método estático do DriveInfo classe).

DriveInfo d\_info = new DriveInfo(@"C:\");

Console.WriteLine("O nome é: " + d\_info.Name);

Console.WriteLine("Tipo de unidade é: " + d\_info.DriveType);

Console.WriteLine("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");

// Obtenha toda a unidade

DriveInfo[] driveInfo = DriveInfo.GetDrives();

foreach (DriveInfo info in driveInfo)

{

Console.WriteLine("Drive {0}", info.Name);

Console.WriteLine("File type: {0}", info.DriveType);

if (info.IsReady == true)

{

Console.WriteLine("Volume label: {0}", info.VolumeLabel);

Console.WriteLine("File system: {0}", info.DriveFormat);

Console.WriteLine("Available space to current user:{0,15} bytes", info.AvailableFreeSpace);

Console.WriteLine("Total available space: {0,15} bytes", info.TotalFreeSpace);

Console.WriteLine("Total size of drive: {0,15} bytes", info.TotalSize);

}

}

Trabalhando com Diretórios e Arquivos

O drive contém diretórios e arquivos. Para trabalhar com eles, DirectoryInfo ou Directory (Static Class) é usado. Ambas as classes podem ser usadas para acessar a estrutura de diretórios. Você pode acessar todos os arquivos das pastas (ou a subpasta), bem como o arquivo específico na pasta ou subpasta usando essas classes. Você também pode criar e executar outras operações relacionadas à pasta em uma nova pasta ou diretório usando essas classes.

O C# fornece as seguintes classes para trabalhar com o sistema de arquivos. Eles podem ser usados ​​para acessar diretórios, acessar arquivos, abrir arquivos para leitura ou gravação, criar um novo arquivo ou mover arquivos existentes de um local para outro, etc.

|  |  |
| --- | --- |
| **Classe** | **Uso** |
| [File](https://www.tutorialsteacher.com/csharp/csharp-file#file) | É uma classe estática que fornece diferentes funcionalidades, como copiar, criar, mover, excluir, abrir para leitura ou gravação, criptografar ou descriptografar, verificar se existe um arquivo, anexar linhas ou texto ao conteúdo de um arquivo, obter a hora do último acesso, etc. |
| [FileInfo](https://www.tutorialsteacher.com/csharp/csharp-fileinfo) | Fornece a mesma funcionalidade que uma classe File estática. Você tem mais controle sobre como executar operações de leitura/gravação em um arquivo escrevendo código manualmente para ler ou gravar bytes de um arquivo. |
| [Directory](https://msdn.microsoft.com/en-us/library/system.io.directory(v=vs.110).aspx) | Euma classe estática que fornece funcionalidade para criar, mover, excluir e acessar um diretório ou subdiretórios. |
| [DirectoryInfo](https://msdn.microsoft.com/en-us/library/system.io.directoryinfo(v=vs.110).aspx) | Fornece métodos de instância para criar, mover, excluir e acessar um diretório ou subdiretórios. |
| [Path](https://msdn.microsoft.com/en-us/library/system.io.path(v=vs.110).aspx) | É uma classe estática que fornece funcionalidades como recuperar a extensão de um arquivo, alterar a extensão de um arquivo, recuperar o caminho físico absoluto e outras funcionalidades relacionadas ao caminho usando uma variável string. |

DIRECTORY E DIRECTORYINFO

A classe Directory é uma classe estática que executa uma única operação. Geralmente, é preferível usar em casos como ao executar uma única tarefa,como apenas criar uma pasta.

DirectoryInfo é uma classe não estática que executa várias operações. Geralmente é usado ao executar várias operações/tarefas, como criar uma pasta, criar subpastas ou movê-las ou obter arquivos dessa pasta. O código a seguir mostra como você pode verificar a existencia do diretorio e criar uma nova pasta usando essas duas classes.

A classe DirectoryInfo herda do objeto System.IO.FileSystemInfo e, assim como o objeto FileInfo, contém as mesmas propriedades adequadas para os atributos, hora da criação, extensão, nome completo, última hora de acesso e última. hora de gravação do diretório. As propriedades para a classe DirectoryInfo estão listadas na Tabela 9-22.

Propriedades e métodos importantes de DirectoryInfo:

|  |  |
| --- | --- |
| **Propriedade** | **Descrição** |
| Exists | Retorna um booleano indicando se o diretório existe |
| Name | Obtém o nome da instância DirectoryInfo |
| Parent | Retorna um objeto DirectoryInfo do diretório pai |
| Root | Retorna um objeto DirectoryInfo do diretório raiz |

O objeto Directory e DirectoryInfo possui métodos semelhantes que executam a mesma operação; a única diferença é que, como o objeto Directory é estático, os métodos levam parâmetros para os diretórios a serem manipulados, e o objeto DirectoryInfo manipula o diretório da instância. A tabela abaixo lista alguns dos métodos comuns entre o objeto Directory e DirectoryInfo.

|  |  |
| --- | --- |
| **Método** | **Descrição** |
| Create (DirectoryInfo) | Cria o diretório |
| CreateDirectory (Directory) |
| Delete | Exclui o diretório |
| GetAccessControl | Retorna um objeto DirectorySecurity que encapsula as entradas da lista de controle de acesso para o diretório atual |
| GetDirectories | Retorna uma matriz DirectoryInfo dos subdiretórios no diretório atual |
| GetFiles | Retorna uma matriz FileInfo dos arquivos no diretório atual |
| GetFileSystemInfos | Retorna uma matriz FileSystemInfo dos arquivos e diretórios no diretório atual |
| MoveTo (DirectoryInfo) | Move um diretório |
| Move (Directory) |
| SetAccessControl | Aplica entradas da lista de controle de acesso descritas por um objeto DirectorySecurity ao diretório atual |

var path\_Directory = @"C:\Users\x\_kat\Desktop\Code\_Estudos\01\_Certificado\_70-483\Udemy\_Exam-70-483\Chapter 11\path\_Directory";

var path\_DirectoryInfo = @"C:\Users\x\_kat\Desktop\Code\_Estudos\01\_Certificado\_70-483\Udemy\_Exam-70-483\Chapter 11\path\_DirectoryInfo";

// Verificar a existência do diretório / pasta criado usando a classe de diretório

if (Directory.Exists(path\_Directory))

{

Console.WriteLine("Pasta de Diretório Existe");

Console.WriteLine("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");

}

else

{

// Crie um novo diretório / pasta usando a classe Directory

DirectoryInfo directory = Directory.CreateDirectory(path\_Directory);

Console.WriteLine("O nome do directory é: " + directory.Name);

Console.WriteLine("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");

}

// Crie um novo diretório / pasta usando a classe DirectoryInfo

DirectoryInfo directoryInfo = new DirectoryInfo(path\_DirectoryInfo);

// Verificar a existência do diretório / pasta criado usando a classe DirectoryInfo

if (directoryInfo.Exists)

{

Console.WriteLine("A pasta DirectoryInfo existe");

Console.WriteLine("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");

}

else

{

directoryInfo.Create();

Console.WriteLine("O nome do directoryInfo é: " + directoryInfo.Name);

Console.WriteLine("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");

}

Conforme mostrado no código, você só precisa fornecer um caminho completo, juntamente com o nome de uma pasta na qual deseja criar a pasta como um parâmetro no método CreateDirectory() da classe Directory. Um novo diretório ou pasta será criado. Quando o caminho não é fornecido, por padrão, a pasta será criada no diretório atual em que você está trabalhando.

O código também mostra a criação de uma pasta pela classe DirectoryInfo no qual você pode executar outras operações fornecidas pela classe DirectoryInfo em uma pasta recém-criada, como o método Exist() (para verificar a existência de uma pasta), Delete() (para excluir a pasta) ou o método CreateSubdirectory() (para criar um subdiretório). Você pode tentar criar um novo diretório em um local em que não tenha permissões suficientes. Nesse caso, será lançada a seguinte exceção:

**UnauthorizedAccessException**

Você também pode excluir a pasta usando o método Delete(), mas se o diretório não for encontrado, você encontrará a seguinte exceção:

**DirectoryNotFoundException.**

Uma coisa importante a lembrar ao trabalhar com diretórios e arquivos é que o sistema operacional controla o acesso a todos os elementos no computador local ou em uma unidade de rede compartilhada. O acesso às pastas pode ser organizado usando a classe DirectorySecurity do sistema. Namespace Security.AccessControl. A Listagem 4-4 mostra como você pode usá-los para permitir que todos acessem uma pasta. Isso, é claro, requer que o programa em execução tenha os direitos para fazer essa modificação.

DirectoryInfo directoryInfo = new DirectoryInfo(path\_DirectoryInfo);

directoryInfo.Create();

DirectorySecurity directorySecurity = directoryInfo.GetAccessControl();

directorySecurity.AddAccessRule(new FileSystemAccessRule("everyone",

FileSystemRights.ReadAndExecute, AccessControlType.Allow));

directoryInfo.SetAccessControl(directorySecurity);

Além dos métodos para criar e remover diretórios, você também pode consultar um método para atributos, subdiretórios ou arquivos. As classes Directory e DirectoryInfo possuem um método para recuperar todos os subdiretórios de um determinado diretório. Isso retorna uma matriz de objetos DirectoryInfo.

Você pode especificar um padrão de pesquisa e uma enumeração do tipo SearchOption que permita pesquisar automaticamente apenas a pasta superior ou todas as subpastas. Ao percorrer todas as pastas, pode ser que você esteja tentando acessar uma pasta sem os direitos de acesso necessários. Isso lança um UnauthorizedAccessException. Ao usar o SearchOption.AllDirectories e uma exceção é lançada, você não obtém resultados.

O exmplo a seguir mostra um exemplo de percorrer a árvore de diretórios manualmente e manipular qualquer exceção que possa ocorrer. Também mostra como usar um padrão de pesquisa para limitar os resultados a todas as pastas que contêm o caractere a. Ele limita a profundidade recursiva da função para garantir que você possa controlar quanto tempo o método leva para executar.

private static void ListDirectories(DirectoryInfo directoryInfo, string searchPattern, int maxLevel, int currentLevel)

{

if (currentLevel >= maxLevel)

{

return;

}

string indent = new string('-', currentLevel);

try

{

DirectoryInfo[] subDirectories = directoryInfo.GetDirectories(searchPattern);

foreach (DirectoryInfo subDirectory in subDirectories)

{

Console.WriteLine(indent + subDirectory.Name);

ListDirectories(subDirectory, searchPattern, maxLevel, currentLevel + 1);

}

}

catch (UnauthorizedAccessException)

{

// You don’t have access to this folder.

Console.WriteLine(indent + "Can’t access: " + directoryInfo.Name);

return;

}

catch (DirectoryNotFoundException)

{

// The folder is removed while iterating

Console.WriteLine(indent + "Can’t find: " + directoryInfo.Name);

return;

}

}

DirectoryInfo directoryInfo = new DirectoryInfo(@"C:\Program Files");

ListDirectories(directoryInfo, "\*a\*", 5, 0);

Um padrão de pesquisa pode consistir em vários caracteres curinga que formam um padrão de pesquisa como você pode ver na tabela abaixo:

|  |  |  |
| --- | --- | --- |
| **Caractere** | **Descrição** | **Exemplo** |
| \* | Zero ou mais caracteres | \*m\* corresponde a Arquivos e mídia comuns, mas não ao Windows |
| ? | Exatamente um caractere | ?edia corresponde à mídia, mas não ao Windows Media Player |

Ao trabalhar com a enorme estrutura de diretórios, use o método EnumerateDirectories()em vez do método GetDirectories() para buscar os diretórios, pois EnumerateDirectories() começa a enumerar diretórios antes que eles tenham sido completamente recuperados (execução tardia/deferred execution); enquanto que no caso GetDirectories(), o código não avançaria até que toda a lista de diretórios fosse recuperada (execução imediata/ immediate execution)

DriveInfo d\_info = new DriveInfo(@"C:\");

var diretorios\_get = d\_info.RootDirectory.GetDirectories();

var diretorios\_info = d\_info.RootDirectory.EnumerateDirectories();

foreach (var dirinfo in diretorios\_info)

{

Console.WriteLine("O nome é:" + dirinfo.Name);

}

Outro método que pode ser útil é o método MoveTo em DirectoryInfo ou Move nas classes de diretório quando você deseja mover um diretório existente para um novo local.

//Using Directory Class

Directory.Move(@"C:\source", @"c:\destination");

//Using DirectoryInfo Class

DirectoryInfo directorymove = new DirectoryInfo(@"C:\Source");

directoryInfo.MoveTo(@"C:\destination");

O método Move() é usado com a classe Directory (classe estática), enquanto o método MoveTo() é usado com a classe DirectoryInfo. O método Move () requer que você conheça o caminho do diretório de origem e o caminho do diretório de destino, enquanto o método MoveTo() exige apenas o caminho do diretório de destino porque o objeto DirectoryInfo já restringe a referência do caminho do diretório de origem.

Função GetFiles

A classe Directory ou DirectoryInfo pode também ser usada para buscar todos os arquivos em uma pasta específica ou sua subpasta ou arquivos com tipos específicos (como imagens), já a classe File ou FileInfo é usada para acessar as informações desses arquivos ou para executar operações nesses arquivos. Estas classes permitem interagir com arquivos para, por exemplo, criar um arquivo ou excluir um arquivo ou verificar sua existência

path\_Directory = @"C:\Windows";

path\_DirectoryInfo = @"C:\Inetpub";

// Obter arquivo de um diretório específico usando a classe de diretório

string[] fileNames = Directory.GetFiles(path\_Directory);

foreach (var name in fileNames)

{

Console.WriteLine("O nome é: {0}", name);

}

// Obter arquivos de um diretório específico usando a classe DirectoryInfo

DirectoryInfo di = new DirectoryInfo(path\_DirectoryInfo);

FileInfo[] files = di.GetFiles();

foreach (var file in files)

{

Console.WriteLine("O nome é: {0}", file.Name);

}

A classe Directory fornecerá apenas nomes de arquivos no diretório fornecido, enquanto DirectoryInfo retornará um objeto FileInfo(Class) no qual você pode executar operações relacionadas a arquivos.

FILE AND FILEINFO

A classe File e FileInfo é semelhante, exceto que a classe File é estática e contém apenas métodos nos quais a classe FileInfo permite criar uma instância que represente um arquivo, portanto ela possui propriedades e métodos. Observe que a classe File não possui propriedades porque é uma classe estática.

Os métodos para a classe File e FileInfo também são semelhantes. Esteja ciente de que os parâmetros são diferentes com base no uso de um objeto File ou FileInfo. Os métodos para a classe File usam parâmetros a um caminho de arquivo já os métodos da classe FileInfo usam uma instância em vez de parâmetros. Alguns dos métodos comuns entre a classe File e FileInfo são: AppendAllText, CopyTo/Copy, Create, Decrypt, Delete, Encrypt, MoveTo, Open, Replace, SetAccessControl.

File

O C# inclui a classe File estática para executar operações de E/S no sistema de arquivos físico. A classe estática File inclui vários métodos utilitários para interagir com arquivos físicos de qualquer tipo, por exemplo binário, texto etc. Use esta classe estática File para executar alguma operação rápida no arquivo físico. Não é recomendável usar a classe File para várias operações em vários arquivos ao mesmo tempo devido a razões de desempenho. Use a classe FileInfo nesse cenário.

Métodos importantes da classe de arquivo estático

|  |  |
| --- | --- |
| **Método** | **Descrição** |
| AppendAllLines | Anexa linhas a um arquivo e, em seguida, fecha o arquivo. Se o arquivo especificado não existir, esse método cria um arquivo, grava as linhas especificadas no arquivo e fecha o arquivo. |
| AppendAllText | Abre um arquivo, anexa a sequência especificada ao arquivo e, em seguida, fecha o arquivo. Se o arquivo não existir, esse método cria um arquivo, grava a sequência especificada no arquivo e fecha o arquivo. |
| AppendText | Cria um StreamWriter que anexa o texto codificado em UTF-8 a um arquivo existente ou a um novo arquivo se o arquivo especificado não existir. |
| Copy | Copia um arquivo existente para um novo arquivo. Não é permitido substituir um arquivo com o mesmo nome. |
| Create | Cria ou substitui um arquivo no caminho especificado. |
| CreateText | Cria ou abre um arquivo para escrever texto codificado em UTF-8. |
| Decrypt | Descriptografa um arquivo que foi criptografado pela conta atual usando o método Criptografar. |
| Delete | Exclui o arquivo especificado. |
| Encrypt | Criptografa um arquivo para que somente a conta usada para criptografar o arquivo possa descriptografá-lo. |
| Exists | Determina se o arquivo especificado existe. |
| GetAccessControl | Obtém um objeto FileSecurity que encapsula as entradas da lista de controle de acesso (ACL) para um arquivo especificado. |
| Move | Move um arquivo especificado para um novo local, fornecendo a opção de especificar um novo nome de arquivo. |
| Open | Abre um FileStream no caminho especificado com acesso de leitura / gravação. |
| ReadAllBytes | Abre um arquivo binário, lê o conteúdo do arquivo em uma matriz de bytes e, em seguida, fecha o arquivo. |
| ReadAllLines | Abre um arquivo de texto, lê todas as linhas do arquivo e depois o fecha. |
| ReadAllText | Abre um arquivo de texto, lê todas as linhas do arquivo e depois o fecha. |
| Replace | Substitui o conteúdo de um arquivo especificado pelo conteúdo de outro arquivo, excluindo o arquivo original e criando um backup do arquivo substituído. |
| WriteAllBytes | Cria um novo arquivo, grava a matriz de bytes especificada no arquivo e fecha o arquivo. Se o arquivo de destino já existir, ele será substituído. |
| WriteAllLines | Cria um novo arquivo, grava uma coleção de seqüências de caracteres no arquivo e fecha o arquivo. |
| WriteAllText | Cria um novo arquivo, grava a sequência especificada no arquivo e, em seguida, fecha o arquivo. Se o arquivo de destino já existir, ele será substituído. |

// Para criar um arquivo no local atual chamado "Arquivo" usando Arquivo (Classe estática)

File.Create("Arquivo.txt").Close();

// Para escrever conteúdo em um arquivo chamado "Arquivo"

File.WriteAllText("Arquivo.txt", "Este é um arquivo criado pela File Class");

// Para ler o arquivo chamado "Arquivo"

string fileContent = File.ReadAllText("Arquivo.txt");

Console.WriteLine(fileContent);

//Se exixtir arquivo, apagar e criar..

if (File.Exists("../Copied Arquivo.txt"))

{

File.Delete("../Copied Arquivo.txt");

// Para copiar "Arquivo" do local atual para um novo (pasta anterior)

File.Copy("Arquivo.txt", "../Copied Arquivo.txt");

}

else

{

File.Copy("Arquivo.txt", "../Copied Arquivo.txt");

}

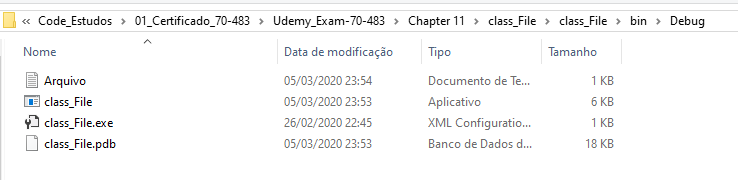
// Para criar um arquivo no local atual chamado "ArquivoInfo" usando a classe FileInfo

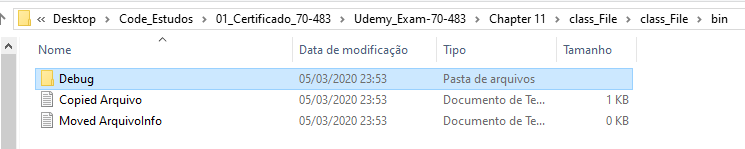
FileInfo info = new FileInfo("ArquivoInfo.txt");

info.Create().Close();

// Para mover "FileInfo" do local atual para um novo (Pasta anterior)

info.MoveTo("../Moved ArquivoInfo.txt");





Como você notou na primeira linha de código, o método Create() é precedido pelo método Close(). Isso se deve ao arquivo(Classe estática) executando uma única operação que é, nesse caso, a criação de um arquivo e, após a criação de um arquivo, você deve fechá-lo antes de executar outra operação nele. É por isso que Close() é chamado após a criação: para que o arquivo possa ser gravado.

Essas operações executadas em um arquivo são muito semelhantes às realizadas no Directory.Copy/CopyTo (métodos de File/FileInfo) deve ser usado onde você deseja deixar uma cópia em um local anterior e outra/outras em um novo local; considerando que Move/MoveTo (métodos de File/FileInfo) deve ser usado onde você não deseja deixar um copie em um local anterior, mas copie-o para um novo local. É como um comportamento de recortar e colar. Quando você precisar atualizar o conteúdo de um arquivo ou alterar o conteúdo de um arquivo já criado, você pode usar o método AppendText () fornecido pelas classes File e FileInfo. Basicamente, existem dois métodos diferentes para criar um arquivo. Estes são dados abaixo com detalhes:

1. Create ():Crie ou substitua um arquivo especificado em um parâmetro como Path e retorne o objeto do FileStream.
2. CreateText ():Crie ou abra um arquivo para escrever e retornar o objeto do StreamWriter.

Repare que qunado a instrução File.Exists retorna false, você pode não pode assumir com segurança que o arquivo não existe, pois você pode não ser o único usuário acessando o sistema de arquivos. Enquanto você trabalha com o sistema de arquivos, outros usuários estão fazendo exatamente a mesma coisa. Talvez eles removem a pasta que você deseja usar para criar um novo arquivo. Ou eles alteram repentinamente as permissões em um arquivo para que você não possa mais acessá-lo.

Normalmente, ao lidar com uma situação em que vários usuários acessam recursos compartilhados, começamos a usar um mecanismo de bloqueio para sincronizar o uso de recursos. O C# possui um mecanismo de bloqueio que você pode usar para sincronizar o acesso ao código quando vários segmentos estiverem envolvidos. Isso garante que um determinado pedaço de código não possa ser executado simultaneamente no mesmo momento. No entanto, o sistema de arquivos não possui esses mecanismos de bloqueio. É um sistema multithread, mas sem nenhum dos regulamentos de segurança que você deseja ver. Veja o código que determina se um arquivo existe e depois lê todo o texto nele.

string path = @"C:\temp\test.txt";

if (File.Exists(path))

File.ReadAllText(path);

Outro usuário pode remover o arquivo entre a chamada para Exists e ReadAllText, o que causaria uma exceção. No entanto, você pode antecipar as exceções que podem ser lançadas e garantir que seu aplicativo saiba como lidar com elas. Veja a Listagem 4-21.

string path = @"C:\temp\test.txt";

try

{

File.ReadAllText(path);

}

catch (DirectoryNotFoundException) { }

catch (FileNotFoundException) { }

Ao trabalhar com o sistema de arquivos, lembre-se de que as exceções podem e ocorrerão. Um bom tratamento de exceções é importante para um aplicativo robusto que funciona com arquivos.

Anexar linhas de texto

Use o método AppendAllLines () para anexar várias linhas de texto ao arquivo especificado, como mostrado abaixo.

string dummyLines = "Esta é a primeira linha." + Environment.NewLine +

"Esta é a segunda linha." + Environment.NewLine +

"Esta é a terceira linha.";

// Abre DummyFile.txt e acrescenta linhas. Se o arquivo não existir, crie e abra.

File.AppendAllLines(@"Arquivo.txt", dummyLines.Split(Environment.NewLine.ToCharArray()));

## 

Anexar String

## Use o método File.AppendAllText () para anexar uma string a um arquivo em uma única linha de código, como mostrado abaixo.

File.AppendAllText(@"Arquivo.txt", Environment.NewLine + "Adicionar uma string ao arquivo");

Substituir texto

Use o método File.WriteAllText () para escrever textos no arquivo. Observe que ele não acrescentará texto, mas substituirá os textos existentes.

File.WriteAllText(@"Arquivo.txt", "Texto subtituido");

O exemplo a seguir mostra como executar operações diferentes usando a classe estática File.

// Verifique se o arquivo existe ou não em um local específico

bool isFileExists = File.Exists(@"C:\DummyFile.txt"); // retorna false

// Copia DummyFile.txt como novo arquivo DummyFileNew.txt

File.Copy(@"C:\DummyFile.txt", @"D:\NewDummyFile.txt");

// Obter quando o arquivo foi acessado pela última vez

DateTime lastAccessTime = File.GetLastAccessTime(@"C:\DummyFile.txt");

// obtém quando o arquivo foi gravado pela última vez

DateTime lastWriteTime = File.GetLastWriteTime(@"C:\DummyFile.txt");

// Mover arquivo para o novo local

File.Move(@"C:\DummyFile.txt", @"D:\DummyFile.txt");

// Abre o arquivo e retorna o FileStream para ler bytes do arquivo

FileStream fs = File.Open(@"D:\DummyFile.txt", FileMode.OpenOrCreate);

// Abra o arquivo e retorne o StreamReader para ler a string do arquivo

StreamReader sr = File.OpenText(@"D:\DummyFile.txt");

// Excluir arquivo

File.Delete(@"C:\DummyFile.txt");

Assim, é fácil trabalhar com arquivos físicos usando a classe estática File. No entanto, se você quiser mais flexibilidade, use a classe FileInfo. Da mesma maneira, use a classe Directory estática para trabalhar com diretórios físicos.

FileInfo

Você aprendeu a executar tarefas diferentes em arquivos físicos usando a classe estática File na seção anterior. Aqui, usaremos a classe FileInfo para executar operações de leitura/gravação em arquivos físicos. A classe FileInfo fornece a mesma funcionalidade que a classe estática File, mas você tem mais controle sobre operações de leitura/gravação em arquivos escrevendo código manualmente para ler ou gravar bytes de um arquivo.

O objeto FileInfo, diferentemente do objeto File, herda do objeto System.IO.FileSystemInfo, que contém propriedades para os atributos, hora da criação, extensão, nome completo, última hora de acesso e última hora de gravação do arquivo.

Propriedades e métodos importantes de FileInfo:

|  |  |
| --- | --- |
| **Propriedade** | **Descrição** |
| Directory | Obtém uma instância do diretório pai. |
| DirectoryName | Obtém uma string representando o caminho completo do diretório. |
| Exists | Obtém um valor indicando se existe um arquivo. |
| Extension | Obtém a string que representa a parte da extensão do arquivo. |
| FullName | Obtém o caminho completo do diretório ou arquivo. |
| IsReadOnly | Obtém ou define um valor que determina se o arquivo atual é somente leitura. |
| LastAccessTime | Obtém ou define a hora em que o arquivo ou diretório atual foi acessado pela última vez |
| LastWriteTime | Obtém ou define o horário em que o arquivo ou diretório atual foi gravado pela última vez. |
| Length | Obtém o tamanho, em bytes, do arquivo atual. |
| Name | Obtém o nome do arquivo. |

…

|  |  |
| --- | --- |
| **Método** | **Descrição** |
| AppendText | Cria um StreamWriter que acrescenta texto ao arquivo representado por esta instância do FileInfo. |
| CopyTo | Copia um arquivo existente para um novo arquivo, impedindo a substituição de um arquivo existente. |
| Create | Cria um arquivo. |
| CreateText | Cria um StreamWriter que grava um novo arquivo de texto. |
| Decrypt | Descriptografa um arquivo que foi criptografado pela conta atual usando o método Criptografar. |
| Delete | Exclui o arquivo especificado. |
| Encrypt | Criptografa um arquivo para que somente a conta usada para criptografar o arquivo possa descriptografá-lo. |
| GetAccessControl | Obtém um objeto FileSecurity que encapsula as entradas da lista de controle de acesso (ACL) para um arquivo especificado. |
| MoveTo | Move um arquivo especificado para um novo local, fornecendo a opção de especificar um novo nome de arquivo. |
| Open | Abre um no FileMode especificado. |
| OpenRead | Cria um FileStream somente leitura. |
| OpenText | Cria um StreamReader com codificação UTF8 que lê de um arquivo de texto existente. |
| OpenWrite | Cria um FileStream somente para gravação. |
| Replace | Substitui o conteúdo de um arquivo especificado pelo arquivo descrito pelo objeto FileInfo atual, excluindo o arquivo original e criando um backup do arquivo substituído. |
| ToString | Retorna um caminho como string. |

O construtor da classe FileInfo usa um parâmetro de string que contém o caminho e o nome do arquivo. Este é o único construtor para o objeto FileInfo. O exemplo de código a seguir cria uma instância de um objeto FileInfo e grava o nome do arquivo na janela Saída:

// Cria o objeto FileInfo para o caminho especificado

FileInfo fi = new FileInfo(@"DummyFile.txt");

Console.WriteLine(fi.Name);

Repare que o arquivo não foi "aberto"; a instância está simplesmente permitindo que você obtenha informações sobre o arquivo. Existem métodos que permitem abrir e alterar o conteúdo do arquivo, mas simplesmente criar uma instância de um objeto FileInfo não abre o arquivo. O exemplo a seguir mostra como ler bytes de um arquivo manualmente e depois convertê-los em uma string usando a codificação UTF8:

// Cria o objeto FileInfo para o caminho especificado

FileInfo fi = new FileInfo(@"DummyFile.txt");

// Abrir arquivo para leitura \ gravação

FileStream fs = fi.Open(FileMode.OpenOrCreate, FileAccess.ReadWrite, FileShare.ReadWrite);

// cria uma matriz de bytes do mesmo tamanho que o comprimento do FileStream

byte[] fileBytes = newbyte[fs.Length];

// define contador para verificar quantos bytes ler. Diminua o contador enquanto lê cada byte

int numBytesToRead = (int)fileBytes.Length;

// Contador para indicar o número de bytes já lidos

int numBytesRead = 0;

// itera até que todos os bytes sejam lidos no FileStream

while (numBytesToRead > 0)

{

int n = fs.Read(fileBytes, numBytesRead, numBytesToRead);

if (n == 0)

break;

numBytesRead += n;

numBytesToRead -= n;

}

// Depois de ler todos os bytes do FileStream, você pode convertê-lo em string em codificação UTF8

string filestring = Encoding.UTF8.GetString(fileBytes);

Console.WriteLine(filestring);

Como você viu no código acima, você deve escrever muito código para ler/escrever uma string de um FileSream. A mesma operação de leitura/gravação pode ser feita facilmente usando o StreamReader e o StreamWriter. O exemplo a seguir mostra como o StreamReader facilita a leitura de strings de um arquivo:

// Cria o objeto FileInfo para o caminho especificado

FileInfo fi = new FileInfo(@"DummyFile.txt");

Console.WriteLine("======Com StreamReader e ===========");

fs = fi.Open(FileMode.OpenOrCreate, FileAccess.ReadWrite, FileShare.ReadWrite);

// Crie o objeto StreamReader passando o objeto FileStream no qual ele precisa operar

StreamReader sr = new StreamReader(fs);

// Use o método ReadToEnd para ler todo o conteúdo do arquivo

string fileContent = sr.ReadToEnd();

Console.WriteLine("StreamReader: " + fileContent);

fs.Close();

Console.WriteLine("======Com using StreamReader e ===========");

using (StreamReader reader = new StreamReader(@"DummyFile.txt"))

{

// Read entire text file with ReadToEnd.

string contents = reader.ReadToEnd();

Console.WriteLine("using StreamReader: " + contents);

}

Observe que fi.Open() possui três parâmetros: O primeiro parâmetro é FileMode para criar e abrir um arquivo, se ele não existir; o segundo parâmetro, FileAccess, é indicar uma operação de leitura; e o terceiro parâmetro é compartilhar o arquivo para leitura com outros usuários enquanto o arquivo estiver aberto. O exemplo a seguir mostra como o StreamWriter facilita a gravação de strings em um arquivo:

using (StreamWriter writer = new StreamWriter(fs))

{

// Crie o objeto StreamWriter para gravar a string no FileSream

writer.WriteLine(Environment.NewLine + "Outra linha do streamwriter");

}

Operações de leitura e gravação não são possíveis no mesmo objeto FileStream simultaneamente.

**System.IO.IOException:** 'O processo não pode acessar o arquivo ‘DummyFile.txt' porque ele está sendo usado por outro processo.'

Se você já estiver lendo um arquivo, crie um objeto FileStream separado para gravar no mesmo arquivo. Assim, você pode usar as classes FileInfo, StreamReader e StreamWriter para ler/gravar o conteúdo do arquivo físico.

FileInfo fi = new FileInfo(@"D:\DummyFile.txt");

FileStream fsToRead = fi.Open(FileMode.OpenOrCreate, FileAccess.ReadWrite, FileShare.ReadWrite);

FileStream fsToWrite = fi.Open(FileMode.OpenOrCreate, FileAccess.ReadWrite, FileShare.ReadWrite);

Trabalhando com caminhos (path)

Ao acessar arquivos, você geralmente precisa combinar um diretório e um nome de arquivo. Concatená-los manualmente usando a adição simples de cadeias pode funcionar, mas é propenso a erros. Nunca tente adicionar manualmente seqüências de caracteres para formar um caminho. Sempre use a classe Path ao combinar várias seqüências para formar um caminho legal. Por exemplo, adicionar as seqüências de caracteres mostradas abaixo resulta em um caminho inválido.

string folder = @"C:\temp";

string fileName = "test.dat";

string fullPath = folder + fileName; // Results in C:\temptest.dat

A classe estática Path que pode ser encontrada no System.IO possui alguns métodos auxiliares para lidar com esses tipos de situações. Um desses métodos é o método estático Combine, que possui várias sobrecargas que aceitam vários parâmetros de seqüência de caracteres. O código abaixo mostra como o método Combine resulta em um caminho correto.

string fullPath = Path.Combine(folder, fileName); // Results in C:\\temp\\test.dat

A classe Path oferece alguns outros métodos úteis: GetDirectoryName, GetExtensions, GetFileName e GetPathRoot. A Listagem 4-13 mostra como você pode usar esses métodos em uma sequência que contém um caminho completo.

string path = @"C:\temp\subdir\file.txt";

Console.WriteLine(Path.GetDirectoryName(path)); // Displays C:\temp\subdir

Console.WriteLine(Path.GetExtension(path)); // Displays .txt

Console.WriteLine(Path.GetFileName(path)); // Displays file.txt

Console.WriteLine(Path.GetPathRoot(path)); // Displays C:\

A classe Path também pode ajudá-lo quando você precisar armazenar temporariamente alguns dados. Você pode usar GetRandomFileName para criar um arquivo ou nome de diretório aleatório. GetTempPath retorna o local da pasta temporária do usuário atual e GetTempFileName cria um arquivo temporário que você pode usar para armazenar alguns dados.

Sumário

1. A classe DriveInfo oferece suporte para interagir com Drives.
2. O C# fornece as classes Directory e DirectoryInfo para interagir com os diretórios. A classe Directory é estática e fornece funcionalidades como criar, copiar, mover, excluir etc. para diretórios físicos com menos codificação e preferível para operações únicas, enquanto DirectoryInfo é preferível para várias operações.
3. File e FileInfo: as duas classes são usadas para interagir com arquivos. A classe File é uma classe estática para leitura/gravação de arquivo físico com menos codificação e preferível para executar uma única operação em um arquivo, enquanto FileInfo é para várias operações.
4. A classe FileInfo e DirectoryInfo fornece a mesma funcionalidade da classe estática File and Directory.

STREAM

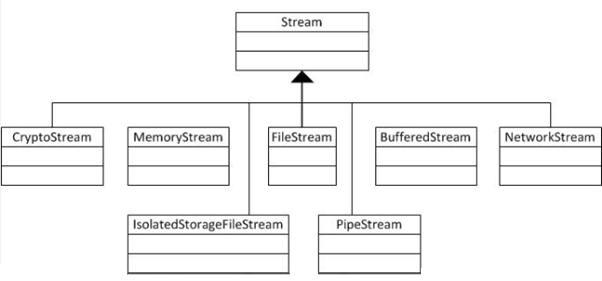
Stream é uma classe abstrata usada para escrever e ler bytes. É para operações de E/S de arquivo. Ao trabalhar com arquivos, é importante saber sobre o Stream, porque um arquivo é armazenado no disco rígido ou no DVD do computador em uma sequência de bytes. Ao se comunicar pela rede, um arquivo é transferido na forma de uma sequência de bytes. Além disso, ele é armazenado na memória na forma de uma sequência de bytes.

O Stream tem três tarefas principais:

1. **Escrita**>> Escrever significa converter o objeto ou dados em bytes e armazená-lo na memória ou em um arquivo, ou pode ser enviado pela rede.
2. **Leitura**>> Ler significa ler os bytes e convertê-los em algo significativos, como Texto, ou desserializá-los em um objeto.
3. **Procurar(Seeking)**>> É o conceito de consulta para a posição atual de um cursor e movendo-o. A procura não é suportada por todos os fluxos, ou seja, você não pode avançar ou retroceder em um fluxo de bytes que está sendo enviado por uma rede.

Stream: System.IO.Stream é uma classe abstrata que fornece métodos padrão para transferir bytes (leitura, gravação, etc.) para a fonte. É como uma classe de wrapper para transferir bytes. As classes que precisam ler/gravar bytes de uma fonte específica devem implementar a classe Stream.

As seguintes classes herdam a classe Stream para fornecer funcionalidade aos bytes de leitura / gravação de uma fonte específica:



|  |  |
| --- | --- |
| **Classe** | **Descrição** |
| FileStream | Lê ou grava bytes de / para um arquivo físico, seja um arquivo .txt, .exe, .jpg ou qualquer outro arquivo. |
| IsolatedStorageFileStream | Lê e grava arquivos em armazenamento isolado |
| MemoryStream | Lê ou grava bytes armazenados na memória. |
| BufferedStream | Lê ou grava bytes de outros Streams para melhorar o desempenho de certas operações de E / S. |
| NetworkStream | Lê ou grava bytes de um soquete de rede. |
| PipeStream | Lê ou grava bytes de diferentes processos. |
| CryptoStream | É para vincular fluxos de dados a transformações criptográficas. |

Codificação e decodificação

O processo de conversão de caracteres em bytes (e vice-versa) é chamado de codificação e decodificação. O Unicode Consortium é responsável por manter um padrão que descreva como isso deve acontecer. Um char, que é o tipo de caractere mais básico, é equivalente a um único caractere Unicode que ocupa 2 bytes de memória. Uma string é simplesmente uma sequência de caracteres. System.Text.Encoding é a classe que ajuda a converter entre bytes e seqüências de caracteres.

O .NET Framework oferece vários padrões de codificação que você pode usar. UTF-8 é aquele que é suficiente para uso geral. Ele pode representar todos os caracteres Unicode e é usado como a codificação padrão em muitas classes do .NET Framework. Outras codificações são:

|  |  |  |
| --- | --- | --- |
| **Esquema Encoding** | **Classe** | **Utilização** |
| ASCII | ASCIIEnconding | GetEnconding(20127) ou propriedade ASCII |
| Default | Enconding | GetEnconding(0) ou propriedade Default |
| UTF-7 | UTF7Enconding | GetEnconding(65000) ou propriedade UTF7 |
| UTF-8 | UTF8Enconding | GetEnconding(65001) ou propriedade UTF8 |
| UTF-16 | UnicodeEnconding | GetEnconding(1201) ou propriedade BigEndianUnicode |
| UTF-16 (little-endian) | UnicodeEnconding | GetEnconding(1200) ou propriedade Unicode |
| UTF32 | UTF32Enconding | GetEnconding(12000) ou propriedade UTF32 |
| Windows OS | Enconding | GetEnconding(1252) |

Para obter o código de página para passer como parâmetro no método GetEncoding utilize o seguinte xódigo que lista todos os códigos de página:

// Print the header.

Console.Write("CodePage identifier and name ");

Console.Write("BrDisp BrSave ");

Console.Write("MNDisp MNSave ");

Console.WriteLine("1-Byte ReadOnly ");

// For every encoding, get the property values.

foreach (EncodingInfo ei in Encoding.GetEncodings())

{

Encoding e = ei.GetEncoding();

Console.Write("{0,-6} {1,-25} ", ei.CodePage, ei.Name);

Console.Write("{0,-8} {1,-8} ", e.IsBrowserDisplay, e.IsBrowserSave);

Console.Write("{0,-8} {1,-8} ", e.IsMailNewsDisplay, e.IsMailNewsSave);

Console.WriteLine("{0,-8} {1,-8} ", e.IsSingleByte, e.IsReadOnly);

}

Para tornar isso um pouco mais fácil ao trabalhar com texto, a classe File também suporta um método CreateText que cria um arquivo com uma codificação UTF-8 para você. CreateText retorna um StreamWriter, uma classe que herda de TextWriter e permite que você escreva caracteres diretamente em um Stream com uma codificação específica, como mostra a Listagem 4-15.

string path = @"c:\temp\test.dat";

using (StreamWriter streamWriter = File.CreateText(path))

{

string myValue = "MyValue";

streamWriter.Write(myValue);

}

Obviamente, você também desejará ler dados de um arquivo. Você pode fazer isso usando diretamente um objeto FileStream, lendo os bytes e convertendo-os novamente em uma sequência com a codificação correta. A Listagem 4-16 mostra como ler os bytes de um arquivo e convertê-los em uma string com codificação UTF-8.

using (FileStream fileStream = File.OpenRead(path))

{

byte[] data = new byte[fileStream.Length];

for (int index = 0; index < fileStream.Length; index++)

{

data[index] = (byte)fileStream.ReadByte();

}

Console.WriteLine(Encoding.UTF8.GetString(data)); // Displays: MyValue

using (StreamReader streamWriter = File.OpenText(path))

{

Console.WriteLine(streamWriter.ReadLine()); // Displays: MyValue

}

}

Se você souber que está analisando um arquivo de texto, também poderá usar um StreamReader (como o oposto do StreamWriter) para ler um arquivo de texto.

FileStream

FileStream provem da classe abstrata Stream, usada principalmente para escrever e ler bytes no arquivo. O próximo exemplo mostra como você pode usar o FileStream quando pode gravar conteúdo em um arquivo ao interagir com a classe File.

FileStream fileStream = File.Create("File.txt");

string content = "This is file content";

byte[] contentInBytes = Encoding.UTF8.GetBytes(content);

fileStream.Write(contentInBytes, 0, contentInBytes.Length);

fileStream.Close();

Basicamente, quando você cria o arquivo usando as classes File ou FileInfo, ele retorna um objeto do tipo FileStream ou StreamWriter ou outro Stream(sequência de bytes), pois o arquivo é armazenado ou transferido na forma de bytes. Após obter o fluxo (stream) de um arquivo, você pode executar as respectivas operações em um arquivo, dependendo do tipo de fluxo. Como neste caso, File.Create() retorna um objeto FileStream para que você possa executar ainda mais as operações do FileStream no arquivo criado.

Como mencionado, o Stream funciona em bytes; portanto, para escrever algo no arquivo, você precisa converter o conteúdo na forma de bytes e depois gravar no arquivo usando o método Write() do FileStream. O método Write() utiliza três parâmetros que contêm os bytes de conteúdo a serem gravados, a posição inicial e final dos bytes a serem gravados.

Ao lidar com arquivos, é importante liberar o recurso, conforme mostrado no código para o métodoFile.Close() deve ser chamado para liberar os recursos do arquivo para que ele possa ser usado para operações posteriores a serem executadas em Files. Se você não liberar o recurso, receberá a exceção "Um arquivo está aberto/está sendo usado em outro processo" ou algo assim. Você também pode usar o bloco using para liberar o recurso. O FileStream tem alguns parâmetros para explicar, os seguintes detalhes ilustram parâmetros que o FileStream aceita:

|  |  |
| --- | --- |
| **Parâmetro** | **Descrição** |
| File\_Name | File\_Name é o nome do arquivo no qual uma operação será executada. |
| FileMode | FileMode determina se você cria, abre ou trunca um arquivo.  1. **Append**: Isto cria o arquivo caso o arquivo não existir ou, se o arquivo existir, coloca o cursor no final do arquivo.  2. **Create**: cria um novo arquivo e, se o arquivo já existir, o substituirá.  3. **CreateNew**: Cria um novo arquivo e, se o arquivo já existir, lançará uma exceção.  4. **Open**: abre o arquivo.  5. **OpenOrCreate**: abre o arquivo existente; se não for encontrado, cria um novo.  6. **Truncate**: abre o arquivo existente e trunca seu tamanho para zero bytes. |
| FileAcces | FileAcces determina o que você pode fazer com o fluxo depois que ele é criado.  1. **Read**: informa ao arquivo que tem acesso apenas a leitura.  2. **ReadWrite**: informa ao arquivo que tem acesso de leitura e gravação do arquivo.  3. **Write**: informa ao arquivo que tem acesso apenas a gravação. |
| FileShare | FileShare determina o tipo de acesso que outros fluxos podem ter nesse arquivo ao mesmo tempo em que você o abre  1. **Delete**: permite a exclusão subsequente de um arquivo.  2. **Iheritable**: permite que o arquivo manipule o processo de herança do filho.  3. **None**: pára de compartilhar o arquivo. O arquivo deve ser fechado antes do acesso por outro processo.  4. **Read**: Permite arquivo para leitura.  5. **ReadWrlte**: Permite arquivo para leitura e gravação.  6. **Write**: Permite que o arquivo seja gravado. |

Preste atenção a essas opções, pois é provável que no exame surja uma pergunta sobre as enumerações FileMode, FileAccess ou FileShare.

FileIOPermissionAccess (Veja mais sobre isso, na parte 4-Depuração e Seguranca, Criptografia)

Ao criar ou abrir um arquivo, o processo deve ter as permissões corretas para o arquivo ou diretório para executar a operação especificada. A enumeração System.Security.Permissions.FileIOPermissionAccess contém os tipos de permissões para um arquivo ou diretório.

|  |  |
| --- | --- |
| **Valor** | **Descrição** |
| NoAccess | Sem acesso a um arquivo ou diretório |
| Read | Acesso de leitura a um arquivo ou diretório |
| Write | Acesso de gravação a um arquivo ou diretório |
| Append | Acesso para anexar dados a um arquivo ou diretório. O acesso anexado também inclui a capacidade de criar um novo arquivo ou diretório |
| PathDiscovery | Acesso às informações sobre o caminho |
| AllAccess | Append, Read, Write, e PathDiscovery fornecem acesso ao arquivo ou diretório |

IPermission perm1 = new FileIOPermission(FileIOPermissionAccess.Read, @"C:\Windows");

IPermission perm2 = new FileIOPermission(FileIOPermissionAccess.Write, @"C:\Windows");

IPermission perm3 = new FileIOPermission(FileIOPermissionAccess.Write, @"C:\Windows");

IPermission perm4 = new FileIOPermission(FileIOPermissionAccess.AllAccess, @"C:\Windows");

IPermission all = new FileIOPermission(PermissionState.Unrestricted);

IPermission none = new FileIOPermission(PermissionState.None);

Console.WriteLine(perm1.Union(perm2)); // IPermission Read="C:\Windows" e Write="C:\Windows"

Console.WriteLine(perm2.Union(perm3)); // IPermission Write="C:\Windows"

Console.WriteLine(perm1.Intersect(perm4)); // IPermission Read="C:\Windows"

Console.WriteLine(perm1.Union(all)); // IPermission Unrestricted="true"

Console.WriteLine(perm1.Intersect(all)); // IPermission Read="C:\Windows"

Console.WriteLine(perm1.Union(none)); // IPermission Read="C:\Windows"

IsolatedStorageFileStream

Para ler ou gravar de um arquivo em um armazenamento isolado, use um objeto IsolatedStorageFileStream com um leitor de fluxo (objeto StreamReader) ou o gravador do fluxo (objeto StreamWriter). V ocê pode usar IsolatedStorageFile e IsolatedStorageFileStream para gravar dados em um arquivo de armazenamento isolado.

* IsolatedStorageFile é essencialmente um ponteiro para o arquivo de armazenamento isolado (área) no disco.

var isolated = IsolatedStorageFile.GetUserStoreForApplication();

using (var writer = new StreamWriter(isolated.CreateFile("TestStore.txt")))

{

writer.WriteLine("Text");

}

* IsolatedStorageFileStream é uma representação na memória dos dados em um arquivo dentro da área de armazenamento isolada.

var isolated\_stream = IsolatedStorageFile.GetUserStoreForApplication();

using (StreamWriter writer = new StreamWriter(new IsolatedStorageFileStream("TestStore.txt", FileMode.Create, FileAccess.Write, isolated\_stream)))

{

writer.WriteLine("Text");

}

O exemplo de código a seguir obtém um repositório isolado e verifica se existe um arquivo chamado TestStore.txt no repositório. Se não existir, ele cria o arquivo e grava "Armazenamento Isolado do Hello" no arquivo. Se TestStore.txt já existir, o código de exemplo será lido a partir do arquivo.

IsolatedStorageFile isoStore = IsolatedStorageFile.GetStore(IsolatedStorageScope.User | IsolatedStorageScope.Assembly, null, null);

if (isoStore.FileExists("TestStore.txt"))

{

Console.WriteLine("The file already exists!");

using (IsolatedStorageFileStream isoStream = new IsolatedStorageFileStream("TestStore.txt", FileMode.Open, isoStore))

{

using (StreamReader reader = new StreamReader(isoStream))

{

Console.WriteLine("Reading contents:");

Console.WriteLine(reader.ReadToEnd());

}

}

}

else

{

using (IsolatedStorageFileStream isoStream = new IsolatedStorageFileStream("TestStore.txt", FileMode.CreateNew, isoStore))

{

using (StreamWriter writer = new StreamWriter(isoStream))

{

writer.WriteLine("Hello Isolated Storage");

Console.WriteLine("You have written to the file.");

}

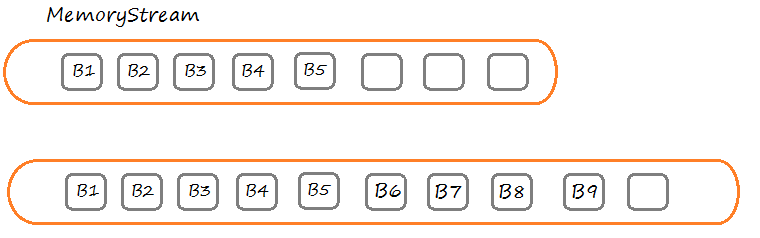
}

}

MemoryStream

O MemoryStream provém da classe abstrata Stream; é usado principalmente para escrever e ler bytes da memória.Se seu trabalho estiver apenas na memória, é melhor usar o MemoryStream. Caso contrário, FileStream com BufferedStream.

Essencialmente, MemoryStream é um objeto que gerencia um buffer é uma matriz de bytes, enquanto os bytes gravados nesse fluxo serão automaticamente atribuídos à próxima posição a partir da posição atual do cursor na matriz. Quando o buffer estiver cheio, uma nova matriz com um tamanho maior a ser criado e copie os dados da matriz antiga.



// Crie um objeto MemoryStream com capacidade de 100 bytes.

MemoryStream memoryStream = new MemoryStream(10);

byte[] javaBytes = Encoding.UTF8.GetBytes("Java ");

byte[] csharpBytes = Encoding.UTF8.GetBytes("CSharp Comprimento");

// Grava bytes no fluxo de memória.

memoryStream.Write(javaBytes, 0, javaBytes.Length);

memoryStream.Write(csharpBytes, 0, csharpBytes.Length);

// Escreva capacidade e comprimento.

// ==> Capacidade: 10, Comprimento: 23. Aumenta automatico para Capacidade: 256

Console.WriteLine("Capacidade: {0}, Comprimento: {1}",

memoryStream.Capacity.ToString(),

memoryStream.Length.ToString());

// Neste momento, a posição do cursor está em pé após o caractere '0'.

Console.WriteLine("Position:" + memoryStream.Position); // ==> 23.

// Mova o cursor para trás 6 bytes, comparado à posição atual.

memoryStream.Seek(-6, SeekOrigin.Current);

// Agora, posicione o cursor após o caractere 'i' e antes de 'm'.

Console.WriteLine("Position:" + memoryStream.Position); // ==> 17

byte[] vsBytes = Encoding.UTF8.GetBytes("vs");

// Grava no fluxo de memória. Substituindo "im" por "vs"

memoryStream.Write(vsBytes, 0, vsBytes.Length);

byte[] allBytes = memoryStream.GetBuffer();

string data = Encoding.UTF8.GetString(allBytes);

// ==> Java vs rp

Console.WriteLine("Conteudo lido por GetBuffer: " + data);

// Define a posição para o início do fluxo

memoryStream.Seek(0, SeekOrigin.Begin);

// Ler do arquivo

byte[] readContent = new byte[memoryStream.Length];

int count = memoryStream.Read(readContent, 0, readContent.Length);

for (int i = count; i < memoryStream.Length; i++)

{

readContent[i] = Convert.ToByte(memoryStream.ReadByte());

}

var resultado = Encoding.UTF8.GetString(readContent);

Console.WriteLine("Conteudo lido por Read: " + resultado);

BufferedStream

Buffer é um bloco de bytes na memória usado para armazenar em cache os dados.O BufferedStream precisa que o fluxo (stream) seja armazenado em buffer e ajuda a melhorar a eficiência da leitura e gravação de dados.

BufferedStream apenas dois construtores, que envolvem fluxos diferentes.

|  |  |
| --- | --- |
| **Constructor** | **Description** |
| BufferedStream(Stream) | Initializes a new instance of the **BufferedStream** class with a default buffer size of 4096 bytes. |
| BufferedStream(Stream, Int32) | Initializes a new instance of the **BufferedStream** class with the specified buffer size. |

Os dados gravados no buffer de fluxo serão aramazenados temporariamente na memória e, quando o buffer estiver cheio, os dados serão automaticamente liberados no arquivo, você poderá liberar proativamente os dados no arquivo usando o método Flush(). O uso do BufferedStream nesse caso reduz o número de vezes ao gravar na unidade e, portanto, aumenta a eficiência do programa.

String fileName = @"File.txt";

FileInfo file = new FileInfo(fileName);

// Verifique se o diretório existe.

file.Directory.Create();

// Crie um novo arquivo, se existir, será sobrescrito.

// Retorna o objeto FileStream.

using (FileStream fileStream = file.Create())

{

// Create BufferedStream quebra o FileStream.

// (Especifique que o buffer é 10000 bytes).

using (BufferedStream bs = new BufferedStream(fileStream, 10000))

{

int índice = 0;

for(índice = 1; índice < 2000; índice++)

{

String s = "Esta é a linha" + índice + "\n";

byte[] bytes = Encoding.UTF8.GetBytes(s);

// Grava no buffer, quando o buffer estiver cheio, ele

// automaticamente pressiona o arquivo.

bs.Write(bytes, 0, bytes.Length);

}

//Set the position to the begninig of stream

bs.Seek(0, SeekOrigin.Begin);

//Read from file

byte[] readContent\_Buff = new byte[bs.Length];

int count\_Buff = bs.Read(readContent\_Buff, 0, readContent\_Buff.Length);

for (int i = count\_Buff; i < bs.Length; i++)

{

readContent\_Buff[i] = Convert.ToByte(bs.ReadByte());

}

string result = Encoding.UTF8.GetString(readContent\_Buff);

Console.WriteLine(result);

// Liberando os dados restantes no buffer para o arquivo.

bs.Flush();

}

}

Os discos rígidos são otimizados para a leitura de blocos maiores de dados. Ler um arquivo byte a byte pode ser mais lento do que ler grandes pedaços de dados e processá-los byte a byte. Assim como no GZipStream, o BufferedStream usa outro Stream em seu construtor. O BufferedStream ajuda você a verificar se é possível ler ou gravar grandes pedaços de dados de uma só vez. O exemplo abaixo mostra como gravar alguns dados em um BufferedStream que agrupa um FileStream.

string path = @"c:\temp\bufferedStream.txt";

using (FileStream fileStream = File.Create(path))

{

using (BufferedStream bufferedStream = new BufferedStream(fileStream))

{

using (StreamWriter streamWriter = new StreamWriter(bufferedStream))

{

streamWriter.WriteLine("A line of text.");

}

}

}

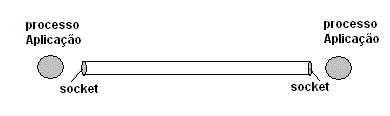
NetworkStream

**NetworkStream não está no exame 70-483.**

A classe NetworkStream fornece métodos para enviar e receber dados sobre Stream soquetes no modo de bloqueio. Você pode usar a classe NetworkStream para transferência de dados síncrona e assíncrona. O .NET fornece um conjunto de classes em System.Net.sockets que torna a nossa vida mais fácil, pois elas fornecem as classes TcpListener, TcpClient, NetworkStream que abstraem muitas tarefas que antes exigiam a criação de código extra.

O que é um socket ?

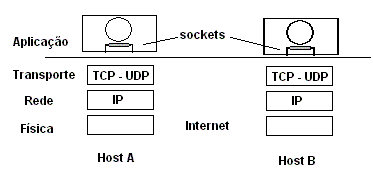
Um socket pode ser entendido como uma porta de um canal de comunicação que permite a um processo executando em um computador enviar/receber mensagens para/de outro processo que pode estar sendo executado no mesmo computador ou num computador remoto.



Os sockets permitem então a comunicação processo a processo da seguinte forma:

* Comunicação local: processos locais usando sockets locais
* Comunicação remota: processos remotos usando sockets em rede (TCP/IP)

Abaixo temos uma figura com que representa a comunicação de sockets e a pilha TCP/IP



Tipos de serviço de transporte:

* Datagrama - transporte não orientado a conexão e sem controle de erros (protocolo UDP)
* DataStream - transporte orientado a conexão com controle de erros (protocolo TCP)

[NetworkStream.Write vs. Socket.Send](https://stackoverflow.com/questions/6555011/networkstream-write-vs-socket-send)

A vantagem de um NetworkStream deriva principalmente do fato de ser um Stream. A desvantagem de um soquete é que o código comum que lê e grava de fontes abstratas de E/S, como um Stream, não pode manipular um soquete. Digamos, por exemplo, que você tinha uma biblioteca de comunicações e que suportava serializar mensagens de arquivos, pipes nomeados e TCP/IP. A escolha ideal para a classe de E/S seria Stream, pois os métodos de serialização poderiam aceitar um FileStream, um PipeStream, um NetworkStream e até um MemoryStream. Esse é o benefício da abstração, porque depois que criamos o Stream, um método pode interagir com ele sem saber que tipo de Stream é.

A seguir a seqüência de ações realizadas no paradigma cliente/servidor (modo orientado a conexão)

|  |  |
| --- | --- |
| **Servidor** | **Cliente** |
| Cria um socket e atribui-lhe um endereço. Este endereço deve ser conhecido pelo cliente. | Cria um socket e atribui-lhe um endereço |
| Aguarda a conexão de um cliente | Solicita a conexão do seu socket ao socket do servidor (conhece o endereço ) |
| Aceita a conexão e cria um novo socket para comunicar com o cliente em causa | Aguarda que a conexão seja estabelecida |
| Recebe a mensagem no novo socket | Envia uma mensagem ( request ) |
| Envia mensagem de resposta (reply) | Recebe a mensage de resposta (reply) |
| fecha a conexão com o cliente | Fecha a conexão com o servidor |

Para colocar em prática a teoria vou criar dois projetos: TCPServidor e TCPCliente. Como não vou usar threads vamos precisar executar cada projeto separadamente. O projeto TCPServidor deverá ser executado primeiro e a seguir o projeto TCPCliente.

Para criar um NetworkStream, você deve fornecer um Socketconectado. Você também pode especificar o que FileAccess permissão que o NetworkStream tem sobre o Socketfornecido. Por padrão, fechar o NetworkStream não fecha o Socketfornecido. Se você quiser que o NetworkStream tenha permissão para fechar o Socketfornecido, deverá especificar true para o valor do parâmetro ownsSocket.

Código do servidor:

IPAddress[] ips = Dns.GetHostAddresses("localhost");

var adress = ips.Length == 0 ? System.Net.IPAddress.Any : ips[1];

const int numeroPorta = 8000;

TcpListener tcpListener = new TcpListener(adress, numeroPorta);

tcpListener.Start();

try

{

Console.WriteLine("Aguardando uma conexão...");

// aceita a conexao do cliente e retorna uma inicializacao TcpClient

using (TcpClient tcpClient = tcpListener.AcceptTcpClient())

{

Console.WriteLine("Conexão aceita.");

// obtem o stream

NetworkStream networkStream\_read = tcpClient.GetStream();

NetworkStream networkStream\_write = tcpClient.GetStream();

// qualquer comunicacao com o cliente remoto usando o TcpClient pode comecar aqui

string responseString = "Conectado ao servidor";

int contagem = 0;

bool terminado = false;

while (!terminado)

{

Byte[] sendBytes = Encoding.ASCII.GetBytes(responseString + " - " + contagem);

networkStream\_write.Write(sendBytes, 0, sendBytes.Length);

// le o stream em um array de bytes

byte[] bytes = new byte[tcpClient.ReceiveBufferSize + 1];

networkStream\_read.Read(bytes, 0, System.Convert.ToInt32(tcpClient.ReceiveBufferSize));

// Retorna os dados recebidos do cliente para o console

string clientdata = Encoding.ASCII.GetString(bytes);

clientdata = clientdata.Replace("\0", "");

Console.WriteLine(clientdata);

responseString = "";

contagem++;

if (clientdata == "tchau" || clientdata == "quit"

|| clientdata == "fim" || clientdata == "exit") terminado = true;

}

}

tcpListener.Stop();

}

catch (Exception e)

{

Console.WriteLine(e.ToString());

Console.ReadLine();

}

Código do cliente:

TcpClient tcpClient = new TcpClient();

Console.WriteLine("Estabelecendo conexão.");

tcpClient.Connect("127.0.0.1", 8000);

NetworkStream networkStream\_read = tcpClient.GetStream();

NetworkStream networkStream\_write = tcpClient.GetStream();

if (networkStream\_write.CanWrite & networkStream\_read.CanRead)

{

bool completed = false;

while (!completed)

{

// Le o NetworkStream em um buffer

byte[] bytes = new byte[tcpClient.ReceiveBufferSize + 1];

string input = Console.ReadLine();

if (input == "tchau" || input == "quit"

|| input == "fim" || input == "exit") completed = true;

networkStream\_read.Read(bytes, 0, System.Convert.ToInt32(tcpClient.ReceiveBufferSize));

// exibe os dados recebidos do host no console

string returndata = Encoding.ASCII.GetString(bytes);

returndata = returndata.Replace("\0", "");

Console.WriteLine(("Host retornou : " + returndata));

byte[] sendBytes = Encoding.UTF8.GetBytes(input);

networkStream\_write.Write(sendBytes, 0, sendBytes.Length);

}

}

else if (!networkStream\_read.CanRead)

{

Console.WriteLine("Não é possível enviar dados para este stream");

tcpClient.Close();

}

else if (!networkStream\_write.CanWrite)

{

Console.WriteLine("Não é possivel ler dados deste stream");

tcpClient.Close();

}

Use os métodos Write e Read para E/S de bloqueio síncrono de thread único simples. Se você quiser processar a E/S usando threads separados, considere o uso dos métodos BeginWrite e EndWrite ou os métodos BeginRead e EndRead para comunicação. O NetworkStream não dá suporte ao acesso aleatório ao fluxo de dados de rede. O valor da propriedade CanSeek, que indica se o fluxo dá suporte à busca, é sempre false; ler a propriedade Position, ler a propriedade Length ou chamar o método Seek gerará um NotSupportedException.

As operações de leitura e gravação podem ser executadas simultaneamente em uma instância da classe NetworkStream sem a necessidade de sincronização. Desde que haja um thread exclusivo para as operações de gravação e um thread exclusivo para as operações de leitura, não haverá nenhuma interferência cruzada entre threads de leitura e gravação e nenhuma sincronização será necessária.

NetworkStream networkStream\_read = tcpClient.GetStream();

NetworkStream networkStream\_write = tcpClient.GetStream();

PipeStream

**PipeStream não está no exame 70-483.**

A classe PipeStream fornece a classe base para operações de pipes nomeados e anônimos no .NET Framework.

Os Pipes, ou canais, são usados ​​para comunicação entre processos e threads. Normalmente, há um servidor de pipe único ao qual um ou mais clientes podem se conectar e trocar mensagens.

Na plataforma .NET os Pipes são implementados como streams e assim você tem uma opção para não somente enviar bytes em um pipe, mas você pode usar todos os recursos dos streams como os leitores(readers) e os escritores (writers).

Pipes são representados pelo namespace System.IO.Pipes na plataforma .NET. Estes são os principais objetos que você precisará, dependendo do tipo de Pipe com o qual deseja trabalhar:

* Nomeados: NamedPipeServerStream e NamedPipeClientStream;
* Anônimos: AnonymousPipeServerStream e AnonymousPipeClientStream;

Os pipes anônimos vêm com algumas limitações em comparação com os pipes nomeados:

* Eles são unidirecionais, ou seja, o servidor e o cliente não podem trocar mensagens. A comunicação em que o servidor e o cliente têm permissão para enviar mensagens é chamada de duplex;
* Só permitem a comunicação com um único cliente. Pipes nomeados fornecem um cenário de vários clientes;
* Não podem funcionar pela rede, são limitados à mesma máquina. Pipes nomeados não têm tal limitação;

Abaixo implementação do servidor Pipe, nomeado e anônimo:

class Program

{

//Variaveis globais para Conexão Anonima

private string \_pipeHandle;

private ManualResetEventSlim \_pipeHandleSet;

static void Main(string[] args)

{

var p = new Program();

p.Run(args);

}

public void Run(string[] args)

{

\_pipeHandleSet = new ManualResetEventSlim(initialState: false);

string nomePipe = args.Length == 1 ? args[0] : "PipeReaderDemo";

PipesLeitor(nomePipe);

}

private void PipesLeitor(string nomePipe)

{

Console.WriteLine($"### Servidor - {nomePipe} ####");

try

{

using (var pipeReader = new NamedPipeServerStream(nomePipe, PipeDirection.In))

{

Console.WriteLine("Aguardando conexão de clientes....");

pipeReader.WaitForConnection();

Console.WriteLine("Cliente(writer) conectado !!");

const int BUFFERSIZE = 256;

bool terminado = false;

while (!terminado)

{

byte[] buffer = new byte[BUFFERSIZE];

int nRead = pipeReader.Read(buffer, 0, BUFFERSIZE);

string input = Encoding.UTF8.GetString(buffer, 0, nRead);

Console.WriteLine(input);

if (input == "tchau" || input == "quit"

|| input == "fim" || input == "exit") terminado = true;

}

}

Console.WriteLine("Agora... pipe anônimo - reader");

var pipeReader\_Anon = new AnonymousPipeServerStream(PipeDirection.In, HandleInheritability.None);

using (var reader = new StreamReader(pipeReader\_Anon))

{

//Seta valor do canal

\_pipeHandle = pipeReader\_Anon.GetClientHandleAsString();

Console.WriteLine($"pipe handle: {\_pipeHandle}");

//Controle que avisa que servidor esta pronto

\_pipeHandleSet.Set();

//iniciando cliente anonimo

Anonimo\_Writer();

bool fim = false;

while (!fim)

{

string line = reader.ReadLine();

Console.WriteLine(line);

if (line == "fim") fim = true;

}

Console.WriteLine("concluindo a leitura...");

}

Console.WriteLine("Leitura completa.");

Console.ReadLine();

}

catch (Exception ex)

{

Console.WriteLine(ex.Message);

}

}

private void Anonimo\_Writer()

{

Console.WriteLine("pipe anônimo - writer");

//Controle que aguarda o servidor estar pronto

\_pipeHandleSet.Wait();

//Aqui é inidicado qual canal será utilizado \_pipeHandle

var pipeWriter = new AnonymousPipeClientStream(PipeDirection.Out, \_pipeHandle);

using (var writer = new StreamWriter(pipeWriter))

{

writer.AutoFlush = true;

Console.WriteLine("iniciando o writer");

for (int i = 0; i < 5; i++)

{

writer.WriteLine($"Mensagem {i}");

Task.Delay(500).Wait();

}

writer.WriteLine("fim");

}

}

}

Abaixo implementação do cliente(writer) Pipe nomeado:

private static void PipesWriter(string nomeServidor, string nomePipe)

{

Console.WriteLine($"### Servidor - {nomePipe} ####");

try

{

// Caso o nomeServidor estiver incorreto: Erro "O caminho da rede não foi encontrado.\r\n"

// Caso o nomePipe estiver incorreto: Não há conexão e fica aguandando....

using (var pipeWriter = new NamedPipeClientStream(nomeServidor, nomePipe, PipeDirection.Out))

{

Console.WriteLine("Tentando conexão com servidor " + nomePipe + "....");

pipeWriter.Connect();

Console.WriteLine("Servidor(reader) conectado");

Console.WriteLine("Iniciando o writer, comece a digitar...");

bool completed = false;

while (!completed)

{

string input = Console.ReadLine();

if (input == "tchau" || input == "quit"

|| input == "fim" || input == "exit") completed = true;

byte[] buffer = Encoding.UTF8.GetBytes(input);

pipeWriter.Write(buffer, 0, buffer.Length);

}

}

Console.WriteLine("Escrita terminada...");

Console.ReadLine();

}

catch (Exception ex)

{

Console.WriteLine(ex.Message);

}

}

string nomePipe = args.Length >= 1 ? args[0] : "PipeReaderDemo";

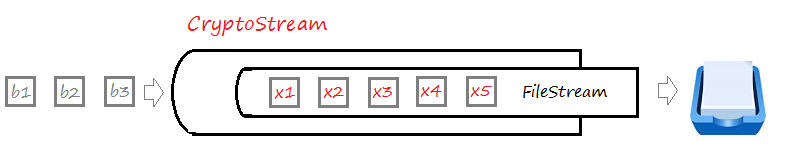
string nomeServidor = args.Length >= 2 ? args[1] : "localhost";

PipesWriter(nomeServidor, nomePipe);

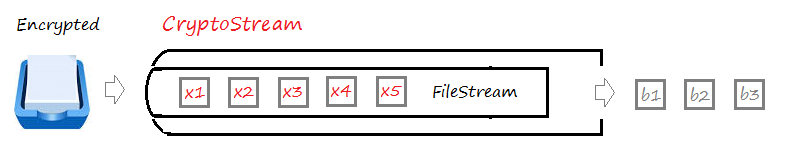
CryptoStream

**CryptoStream não está no exame 70-483.**

CryptoStream é uma classe usada para criptografia de fluxo de dados.Quando você escreve dados de bytes no CryptoStream, os bytes são criptografados em outros bytes antes de serem liberados no fluxo que está gravado no arquivo. A imagem abaixo ilustra o CryptoStream encapsular outro fluxo (como gravar arquivo de fluxo).Observe que você pode escolher um algoritmo de criptografia ao criar o objeto CryptoStream.



Em uma situação inversa, o CryptoStream quebra um arquivo de fluxo de leitura (arquivo cujo conteúdo foi criptografado acima), o byte no FileStream foi criptografado e será descriptografado pelo CryptoStream. Outra coisa importante a ser lembrada é que nem todos os algoritmos de criptografia têm soluções de criptografia e descriptografia bidirecional.



A criptografia dos dados que passam por streams para privacidade e integridade também é importante.  
O C # fornece uma classe, CryptoStream, para a criptografia de dados que trafega por streams. A implementação é similar a da criptografia simétrica, porém, como se trata de streams, se não for usar using é necessário finalizar o objeto para não carregar a memória.

class Criptografia\_Stream

{

public byte[] EncryptStream(SymmetricAlgorithm symmetricAlgo)

{

// especifique os dados

string plainData = "Mensagem Secreta STREAM";

byte[] cipherDataInBytes;

ICryptoTransform encryptor = symmetricAlgo.CreateEncryptor(symmetricAlgo.Key, symmetricAlgo.IV);

// Create the streams used for encryption.

using (MemoryStream memoryStream = new MemoryStream())

{

// crptoStream conhece o criptografador e o stream no qual os dados serão gravados

using (CryptoStream crptoStream = new CryptoStream(memoryStream, encryptor, CryptoStreamMode.Write))

{

// writer tem referência de cryptoStream (o que criptografar e onde)

using (StreamWriter streamWriter = new StreamWriter(crptoStream))

{

//Write all data to the stream.

streamWriter.Write(plainData);

}

cipherDataInBytes = memoryStream.ToArray();

// coloca os bytes dos dados criptografados na string

string cipherData = Encoding.UTF8.GetString(cipherDataInBytes);

// Simétrica criptografia:o???w????r?\_\_7v?l?Zf?zzi?\_J?c

Console.WriteLine("Stream criptografado:" + cipherData);

}

}

return cipherDataInBytes;

}

public void DecryptStream(SymmetricAlgorithm symmetricAlgo, byte[] cipherDataInBytes)

{

ICryptoTransform decryptor = symmetricAlgo.CreateDecryptor(symmetricAlgo.Key, symmetricAlgo.IV);

using (MemoryStream msDecrypt = new MemoryStream(cipherDataInBytes))

{

using (CryptoStream csDecrypt = new CryptoStream(msDecrypt, decryptor, CryptoStreamMode.Read))

{

using (StreamReader srDecrypt = new StreamReader(csDecrypt))

{

// Read the decrypted bytes from the decrypting stream

// and place them in a string.

var plaintext = srDecrypt.ReadToEnd();

Console.WriteLine("Stream Descriptografado..." + plaintext);

}

}

}

}

}

using (SymmetricAlgorithm symmetricAlgo = SymmetricAlgorithm.Create())

{

var stream\_cripto = new Criptografia\_Stream();

var cipherDataInBytes = stream\_cripto.EncryptStream(symmetricAlgo);

stream\_cripto.DecryptStream(symmetricAlgo, cipherDataInBytes);

}

GZipStream - Compactar dados

Devido à maneira como os Streams são projetados, você pode acoplar vários objetos de Stream para executar uma operação mais complexa. Este princípio é chamado de padrão decorador. Por exemplo, quando você deseja compactar alguns dados, pode usar um GZipStream., Que utiliza outro objeto Stream em seu construtor. O segundo fluxo é usado como entrada ou saída do algoritmo de compactação. Ao atribuir a ele um objeto FileStream, você pode compactar facilmente alguns dados (como outro arquivo) e armazená-los em disco. Você também pode usar um MemoryStream como entrada ou saída de um GZipStream. Você pode encontrar o GZipStream no espaço para nome System.IO.Compression. A Listagem 4-18 cria alguns dados, grava-os em um arquivo e compacta o arquivo. Como você pode ver, o arquivo compactado é muito menor que o arquivo original.

string folder = @"c:\temp";

string uncompressedFilePath = Path.Combine(folder, "uncompressed.dat");

string compressedFilePath = Path.Combine(folder, "compressed.gz");

byte[] dataToCompress = Enumerable.Repeat((byte)'a', 1024 \* 1024).ToArray();

using (FileStream uncompressedFileStream = File.Create(uncompressedFilePath))

{

uncompressedFileStream.Write(dataToCompress, 0, dataToCompress.Length);

}

using (FileStream compressedFileStream = File.Create(compressedFilePath))

{

using (GZipStream compressionStream = new GZipStream(

compressedFileStream, CompressionMode.Compress))

{

compressionStream.Write(dataToCompress, 0, dataToCompress.Length);

}

}

FileInfo uncompressedFile = new FileInfo(uncompressedFilePath);

FileInfo compressedFile = new FileInfo(compressedFilePath);

Console.WriteLine(uncompressedFile.Length); // Displays 1048576

Console.WriteLine(compressedFile.Length); // Displays 1052

Como mostra a Lista 4-18, você pode passar outro Stream para o construtor de um GZipStream. Ao gravar dados no GZipStream, ele os comprime e os encaminha imediatamente para o FileStream.

WebRequest e WebResponse

WebRequest e WebResponse formam um par de classes que você pode usar em conjunto para enviar uma solicitação de informações e, em seguida, receber a resposta com os dados solicitados. Um WebRequest é criado usando um método estático Create na classe WebRequest. O método Create inspeciona o endereço que você passa para ele e, em seguida, seleciona a implementação correta do protocolo. Se você passasse o endereço http://www.microsoft.com para ele, veria que você está trabalhando com o protocolo HTTP e retornaria um HttpWebRequest. Depois de criar o WebRequest correto, você pode definir outras propriedades, como instruções de autenticação ou cache. Ao terminar de compor sua solicitação, chame o método GetResponse para executar a solicitação e recuperar a resposta.

WebRequest request = WebRequest.Create(@"http://www.microsoft.com");

using (WebResponse resp = request.GetResponse())

{

using (StreamReader responseStream = new StreamReader(resp.GetResponseStream()))

{

string responseText = responseStream.ReadToEnd();

Console.WriteLine(responseText); // Displays the HTML of the website

}

}

FILE READER AND WRITER

Para converter bytes em formato legível ou gravar ou ler valores como bytes ou como sequência, o .NET oferece o seguinteclasses nesse caso. Reader e Writer são classes no namespace System.IO que lêem ou gravam caracteres codificados de e para fluxos. A tabela abaixo lista os tipos comuns de leitores e gravadores no namespace System.IO.

|  |  |
| --- | --- |
| **Classe** | **Descrição** |
| BinaryReader, BinaryWriter | Usado para ler e escrever valores binários |
| StreamReader, StreamWriter | Usado para ler e escrever caracteres usando um valor codificado para converter os caracteres de e para bytes |
| StringReader, StringWriter | Usado para ler e escrever caracteres de e para strings |
| TextReader, TextWriter | Classes abstratas para outros leitores e escritores que lêem e escrevem caracteres ou seqüências |

BinaryReader and BinaryWriter

Essas classes são usadas para ler e escrever valores como Valores Binários.

* BinaryReader é uma classe auxiliar para ler o tipo de dados primitivo a partir de bytes.
* BinaryWriter grava tipos primitivos em binário.

O BinaryReader possui métodos para ler o valor de um tipo de dados específico. Por exemplo, se houver um valor de sequência em forma de binário, use o método ReadString () e assim por diante, mas se não houver valor escrito como binário e você desejar ler, a exceção será lançada. Além disso, é importante ler ordinalmente à medida que os valores são gravados.

//Write Data Types values as Binary Values in Sample.dat file

FileStream file = File.Create("Sample.dat");

BinaryWriter binaryWriter = new BinaryWriter(file);

binaryWriter.Write("String Value");

binaryWriter.Write('A');

binaryWriter.Write(true);

binaryWriter.Close();

//Read Binary values as respective data type's values from Sample.dat

FileStream fileToOpen = File.Open("Sample.dat", FileMode.Open);

BinaryReader binaryReader = new BinaryReader(fileToOpen);

Console.WriteLine(binaryReader.ReadString());

Console.WriteLine(binaryReader.ReadChar());

Console.WriteLine(binaryReader.ReadBoolean());

Console.ReadKey();

StreamReader and StreamWriter

O StreamReader e o StringReader herdam da classe abstrata TextReader, é uma classe auxiliar para ler caracteres de um fluxo convertendo bytes em caracteres usando um valor codificado. Pode ser usado para ler strings (caracteres) de diferentes fluxos, como FileStream, MemoryStream, etc. A classe StreamReader é usada para ler a entrada de caracteres em uma codificação específica. A codificação padrão é UTF-8. Você pode usar um StreamReader para ler um arquivo de texto padrão. A tabela abaixo lista alguns dos métodos para a classe StreamReader.

|  |  |
| --- | --- |
| **Método** | **Descrição** |
| Close | Fecha o leitor de fluxo e o fluxo subjacente |
| Peek | Retorna o próximo caractere no fluxo, mas não move a posição do caractere |
| Read() | Retorna o próximo caractere no fluxo e move a posição do caractere em um |
| Read(Char[], Int32, Int32) | Lê o número especificado de caracteres na matriz de bytes |
| ReadBlock(Char[], Int32, Int32) |
| ReadLine | Lê uma linha de caracteres e retorna uma string |
| ReadToEnd | Lê todos os caracteres da posição atual até o final do arquivo e retorna uma string |

O StreamReader fornece métodos para leitura de caractere por caractere, linha por linha ou um arquivo inteiro em uma chamada. Os métodos Read e ReadBlock retornam caracteres e matrizes de caracteres; os métodos ReadLine e ReadToEnd retornam seqüências de caracteres. O código a seguir abre um arquivo e primeiro lê o conteúdo caractere por caractere, depois linha por linha e, em seguida, lê todo o conteúdo.

string projectDirectory = Directory.GetParent(workingDirectory).Parent.FullName;

using (StreamReader streamReader = new StreamReader(projectDirectory + "\\Numbers.txt"))

{

Console.WriteLine("Char by Char");

while (!streamReader.EndOfStream)

Console.WriteLine((char)streamReader.Read());

}

using (StreamReader streamReader = new StreamReader(projectDirectory + "\\Numbers.txt"))

{

Console.WriteLine("Line by line");

while (!streamReader.EndOfStream)

Console.WriteLine(streamReader.ReadLine());

}

using (StreamReader streamReader = new StreamReader(projectDirectory + "\\Numbers.txt"))

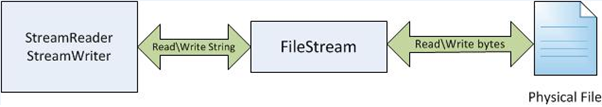
{

Console.WriteLine("Entire file");

Console.WriteLine(streamReader.ReadToEnd());

}

O StreamWriter e o StringWriter herdam da classe TextWriter, é uma classe auxiliar para escrever uma string em um Stream convertendo caracteres em bytes. Ele pode ser usado para gravar strings em diferentes Streams, como FileStream, MemoryStream, etc.



A imagem acima mostra que o FileStream lê bytes de um arquivo físico e, em seguida, o StreamReader lê as strings convertendo esses bytes em strings. Da mesma forma, o StreamWriter pega uma string e a converte em bytes e grava no FileStream e, em seguida, o FileStream grava os bytes em um arquivo físico. Portanto, o FileStream lida com bytes, como StreamReader e StreamWriter lida com seqüências de caracteres.

using (StreamWriter streamWriter = new StreamWriter(projectDirectory + "\\Numbers.txt"))

{

streamWriter.Write('A');

}

using (StreamReader streamReader = new StreamReader(projectDirectory + "\\Numbers.txt"))

{

Console.WriteLine(streamReader.ReadLine());

}

StringReader e StringWriter

Essas classes são usadas para ler e escrever caracteres de e para a string. Lembre-se que estas duas classes aceitam em seus construtures a classe StringBuilder como parâmetro, o que, geralmente é feito para que se conecte a entrada e leitura da string em partes diferentes do código.

// Grava string ou caracteres

using (StringWriter stringWriter = new StringWriter())

{

stringWriter.Write("Exemplo do String Writer");

stringWriter.Write("Anexar texto");

Console.WriteLine(stringWriter.ToString());

}

using (StringReader stringReader = new StringReader("Exemplo de leitor de string"))

{

Console.WriteLine(stringReader.ReadLine());

}

Comunicação através da rede

O namespace System.Net fornece suporte para seus aplicativos se comunicarem em uma rede. Geralmente, os membros desse namespace que você usa são as classes WebRequest e WebResponse. Ambas classes são abstratas e usadas para se comunicar pela rede. O namespace System.Net também fornece classes implementadas específicas que dependem de qual protocolo será usado para comunicação. Por exemplo, a classe HttpWebRequest e a classe HttpWebResponse são usadas quando você está usando o protocolo Http. Em geral, usamos a classe WebRequest para enviar a solicitação de informações e a classe WebResponse para receber a resposta das informações solicitadas.

WebRequest request = WebRequest.Create("http://www.apress.com");

using (WebResponse response = request.GetResponse())

{

StreamReader reader = new StreamReader(response.GetResponseStream());

string result = reader.ReadToEnd();

Console.WriteLine(result);

}

O WebRequest é criado usando o método Create() (método estático da classe WebRequest), que pega o endereço da solicitação em uma string ou no formato Uri. O WebResponse está vinculado ao WebRequest, então ele obtem a resposta das informações ou dados solicitados usando seu método GetResponse().

O método Create() do WebRequest inspeciona o endereço e escolhe a implementação correta do protocolo. No código, passamos o http://www.apress.com, para que ele escolhesse o protocolo Http e retornasse o HttpWebRequest. Você também pode usar o método ou as propriedades do WebRequest para executar outras operações nele. Após obter a resposta, o StreamReader é usado para obter a resposta no fluxo para que possa ser lida.

ARQUIVO ASSÍNCRONO: ASYNC E AWAIT IN FILE I/O

A leitura e gravação do arquivo pode ser uma tarefa demorada e é necessário aguardar muito tempo para concluir a operação. No código síncrono é executado linha por linha e frequentemente aguarda até que a tarefa termine. Essa espera pode ser longa o suficiente para incomodar o usuário e causar um grande impacto na experiência do usuário. Por exemplo, em aplicativos de desktop, você tem um thread que é o thread principal e que é responsável por todas as tarefas, ou seja, atualizar a interface do usuário e processar outras tarefas também. Se você tem uma tarefa longa a ser processada (ou seja, aguardar a resposta do fluxo da rede ou ler um arquivo da Internet), o thread principal ficará ocupado no processamento dessa tarefa e, enquanto isso, a interface do usuário do aplicativo ficará bloqueada e não sendo possível responder, o que será uma experiência ruim para o usuário.

Quando você cria um aplicativo da área de trabalho, como um aplicativo Windows Presentation Foundation (WPF), WinForm ou Windows Store, seu aplicativo possui um thread principal responsável pela atualização da interface do usuário. Esse encadeamento é responsável por processar todas as atividades do usuário. Se esse segmento estiver ocupado com outra coisa, o aplicativo parece não responder. A maioria dos usuários tentará fechar esse aplicativo porque eles acham que algo está errado. É importante executar operações de execução longa em outro encadeamento, o que garante que seu aplicativo permaneça responsivo enquanto executa a operação de execução longa em segundo plano.

Com um aplicativo de servidor, as coisas são um pouco diferentes. Quando o usuário acessa o servidor Web do ASP.NET, ele precisa aguardar antes que a solicitação seja concluída. Se a página demorar muito para carregar, o usuário perceberá isso como um desempenho ruim. O que está acontecendo no servidor é que um encadeamento atende a solicitação e começa a criar a resposta para o usuário. O servidor começa a coletar dados, produzir HTML e executar todos os tipos de outras ações. Um servidor possui um número limitado de threads para lidar com solicitações recebidas. Quando um encadeamento aguarda a E/S, não pode funcionar com outras solicitações. Quando todos os encadeamentos estão ocupados e uma nova solicitação é recebida, a nova solicitação é colocada na fila. Esse problema é chamado de "falta de pool de threads". Todos os encadeamentos aguardam que algumas E/S ocorram com uma carga de CPU de 0%, e os tempos de resposta para novas solicitações passarão do teto.

Em tais cenários, as classes Stream, Reader e Writer oferecem a capacidade de ler ou gravar arquivos de forma assíncrona. Isso pode ser útil quando você deseja retornar o processamento de volta à interface do usuário enquanto realiza uma operação demorada em um arquivo grande. No C#, existem duas novas palavras-chave usadas ao lidar com o processamento assíncrono: async e await. Async e Await são as palavras-chave fornecidas pelo .NET Framework no C# 5.0. Eles dizem ao compilador para executar o código de maneira assíncrona. A palavra-chave async é um modificador de método que permite que o método use a palavra-chave wait e também habilita um método de chamada para essa função usando a palavra-chave wait. A palavra-chave wait é usada ao chamar um método para suspender a execução nesse método até que a tarefa esperada seja concluída.

Um método assíncrono contém async em seu nome, como ReadAsync, WriteAsync, ReadLineAsync e ReadToEndAsync, etc. Esses métodos assíncronos são implementados em classes de fluxo, como Stream, FileStream, MemoryStream e em classes usadas para ler ou gravar em fluxos como TextReader e TextWriter.

Por exemplo, suponha que você tenha um aplicativo de formulários do Windows que tenha um botão que execute uma tarefa de longa execução. Você pode modificar a assinatura do evento de clique no botão com o modificador assíncrono e, em seguida, chamar o método de longa duração usando a palavra-chave wait. Isso permite que o processo de longa execução seja executado, mas também permite que o usuário navegue para diferentes partes do seu aplicativo, para que o sistema não pareça estar bloqueado.

private async void button1\_Click(object sender, EventArgs e)

{

button1.Text = "Searching...";

// This will get the current WORKING directory (i.e. \bin\Debug)

string workingDirectory = Environment.CurrentDirectory;

// This will get the current PROJECT directory

string projectDirectory = Directory.GetParent(workingDirectory).Parent.FullName;

string outputFileName = projectDirectory + "\\FoundFiles.txt";

await SearchDirectory(workingDirectory, "A", outputFileName);

button1.Text = "Finished";

Process.Start(outputFileName);

}

No exemplo, a legenda do formulário é alterada para "Searching" quando o botão é clicado. Quando o método MyLongRunningProcess é chamado, o método é executado, mas o processamento é retornado ao thread principal até que o método SearchDirectory termine a execução. Depois disso, a legenda do formulário é alterada para "Concluído". Você pode usar os métodos assíncronos para o arquivo de E/S de maneira semelhante. O exemplo de código a seguir implementa o método SearchDirectory que pesquisa todos os arquivos em uma determinada pasta e procura uma sequência específica no arquivo. Se o arquivo contiver a string, seu nome será gravado em um arquivo de saída. Quando o processo é concluído, o arquivo de saída é mostrado no visualizador de texto padrão.

private static async Task SearchDirectory(string searchPath, string searchString, string outputFileName)

{

StreamWriter streamWriter = File.CreateText(outputFileName);

string[] fileNames = Directory.GetFiles(searchPath);

//para Testar funcionalidade do async

await Task.Run(() =>

{

for (int i = 0; i < 8; i++)

{

Thread.Sleep(4000);

Debug.Write("Processando... ");

}

});

//await FindTextInFilesAsync(fileNames, searchString, streamWriter);

streamWriter.Close();

}

private static async Task FindTextInFilesAsync(string[] fileNames, string searchString, StreamWriter outputFile)

{

foreach (string fileName in fileNames)

{

if (fileName.ToLower().EndsWith(".txt"))

{

StreamReader streamReader = new StreamReader(fileName);

string textOfFile = await streamReader.ReadToEndAsync();

streamReader.Close();

if (textOfFile.Contains(searchString))

{

await outputFile.WriteLineAsync(fileName);

}

}

}

}



É importante saber que a classe File estática não suporta E/S assíncrona real. Se você chamar métodos assíncronos, isso será falsificado usando outro thread do pool de threads para trabalhar com o arquivo. Para E / S assíncrona real, você precisa usar o objeto FileStream e transmitir um valor verdadeiro para o parâmetro useAsync.

O exemplo abaixo mostra um exemplo de como gravar assincronamente em um arquivo. O método Write é substituído por WriteAsync, o método é marcado com o modificador assíncrono para sinalizar ao compilador que você precisa de ajuda para transformar seu código e, finalmente, você usa a palavra-chave await na tarefa retornada.

public async Task CreateAndWriteAsyncToFile()

{

using (FileStream stream = new FileStream("test.dat", FileMode.Create,

FileAccess.Write, FileShare.None, 4096, true))

{

byte[] data = new byte[100000];

new Random().NextBytes(data);

await stream.WriteAsync(data, 0, data.Length);

}

}

O objeto Task retornado representa algum trabalho em andamento, encapsulando o estado da operação assíncrona. Eventualmente, o objeto Task retorna o resultado da operação ou exceções que foram geradas de forma assíncrona. No caso acima, o método não possui nenhum valor retornado ao chamador. É por isso que o WriteAsync retorna um objeto Task comum. Quando um método tem um valor de retorno, ele retorna a Tarefa<T>, onde T é o tipo do valor retornado. No próximo exemplo, o método GetStringAsync é usado. Esse método retorna Task<string>, o que significa que, eventualmente, quando o processo for concluído, um valor string estará disponível (ou uma exceção).

public async Task ReadAsyncHttpRequest()

{

HttpClient client = new HttpClient();

string result = await client.GetStringAsync("http://www.microsoft.com");

Debug.WriteLine("Fim ReadAsyncHttpRequest... ");

}

Sempre que o .NET Framework oferece um equivalente assíncrono de um método síncrono, é melhor usá-lo. Cria uma melhor experiência do usuário e um aplicativo mais escalável. No entanto, quando você está trabalhando em um aplicativo crítico para o desempenho, vale a pena saber o que o compilador faz por você. Se não tiver certeza, use um criador de perfil para realmente medir a diferença, para que você possa tomar uma decisão informada.

Executando operações de E/S em paralelo

Operações E/S assíncrona reais garantem que seu thread possa executar outro trabalho até que o sistema operacional notifique seu aplicativo de que a E/S esteje pronta. Várias operações de E/S assíncronas ainda podem ser executadas uma após a outra. Quando você olha para o código abaixo, pode ver que várias esperas são usadas em um método. Com o código atual, cada solicitação da web inicia quando a anterior é concluída. O thread não será bloqueado durante a execução dessas solicitações, mas o tempo que o método leva é a soma das três solicitações da Web.

public async Task ExecuteMultipleRequests()

{

HttpClient client = new HttpClient();

string microsoft = await client.GetStringAsync("http://www.microsoft.com");

string msdn = await client.GetStringAsync("http://msdn.microsoft.com");

string blogs = await client.GetStringAsync("http://blogs.msdn.com/");

Debug.WriteLine("Fim ExecuteMultipleRequests... ");

}

Você também pode escrever um código que executará essas operações em paralelo. Se você executasse essas solicitações em paralelo, teria apenas que esperar o tempo que a solicitação mais longa demorar (as outras duas já estarão concluídas). Você pode fazer isso usando o método estático Task.WhenAll. Assim que você chama GetStringAsync, a operação assíncrona é iniciada. No entanto, você não espera imediatamente o resultado. Em vez disso, você deixa todos os três pedidos iniciarem e depois espera que eles terminem. Agora, todas as três operações são executadas em paralelo, o que pode economizar muito tempo.

public async Task ExecuteMultipleRequestsInParallel()

{

HttpClient client = new HttpClient();

Task microsoft = client.GetStringAsync("http://www.microsoft.com");

Task msdn = client.GetStringAsync("http://msdn.microsoft.com");

Task blogs = client.GetStringAsync("http://blogs.msdn.com/");

await Task.WhenAll(microsoft, msdn, blogs);

Debug.WriteLine("Fim ExecuteMultipleRequestsInParallel... ");

}

Sumário

1. Stream é uma classe abstratabasepara todas as outras classes que tem três tarefas principais: leitura, escrita e busca bytes em fontes diferentes.
2. FileStream deriva da classe abstrata Stream; é usado para escrever e ler bytes no arquivo físico.
3. As classes Reader & Writer fornecem funcionalidade para ler bytes das classes Stream (FileStream, MemoryStream etc.) e converte bytes em codificação apropriada.
4. StreamReader fornece um método auxiliar para ler a string do FileStream convertendo bytes em strings. O StreamWriter fornece um método auxiliar para gravar a seqüência de caracteres no FileStream, convertendo as seqüências em bytes.
5. MemoryStream deriva da classe abstrata Stream; é usado para escrever e ler bytes da memória.
6. Buffer é um bloco de bytes na memória usado para armazenar em cache os dados. O BufferedStream precisa que um fluxo de bytes na memória para armazenar em cache os dados.
7. StringReader e StringWriter são usadas para ler e escrever strings.
8. As classes BinaryReader e BinaryWriter são usadas para ler e escrever valores como Valores Binários.
9. StreamWriter deriva de TextWriter e é usado para escrever caracteres / caracteres no fluxo. StreamReader deriva de TextReader; é usado para ler bytes ou string.
10. Para comunicação em rede, usamos a classe WebRequest para enviar a solicitação de informações e a classe WebResponse para receber a resposta das informações solicitadas.

Dados de consumo

* Recuperar dados de um banco de dados; atualizar dados em um banco de dados; consumir dados JSON e XML; recuperar dados usando serviços Web

O gerenciamento de dados é um dos aspectos mais importantes de um aplicativo. Imagine que você pode criar apenas aplicativos que armazenam seus dados na memória. Assim que o usuário sai, todos os dados são perdidos e os lançamentos subsequentes exigem a reinserção de todos os dados necessários. Obviamente, seria uma situação impossível de se trabalhar, e é por isso que o .NET Framework ajuda a armazenar seus dados de maneira persistente. Isso pode ser feito em um banco de dados usando diretamente o ADO.NET ou o Entity Framework. Você também pode armazenar e solicitar dados de um Web service externo e recuperar a resposta em JavaScript Object Notation (JSON) ou Extensible Markup Language (XML). O .NET ajuda a executar essas solicitações e analisar os dados retornados.

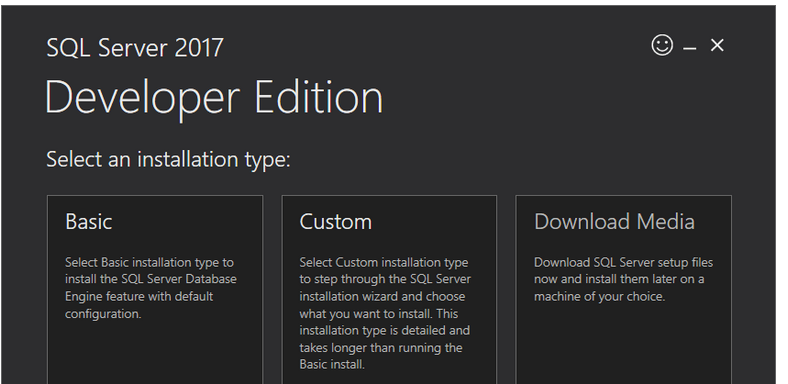
INSTALAR SQL SERVER E O SQL SERVER MANAGEMENT STUDIO

Antes de começarmos a explorar os tópicos dessa seção, como muitos exercícios serão realizados utilizando como base o provedor SQL, será necessário instalar duas ferramentas: SQL Server e o SQL Server Management Studio. Para instalar o SQL Server, você precisa baixá-lo no site Microsoft.com através do seguinte link:



Após a conclusão do download, clique duas vezes no arquivo SQL2019-SSEI-Dev para iniciar o instalador.

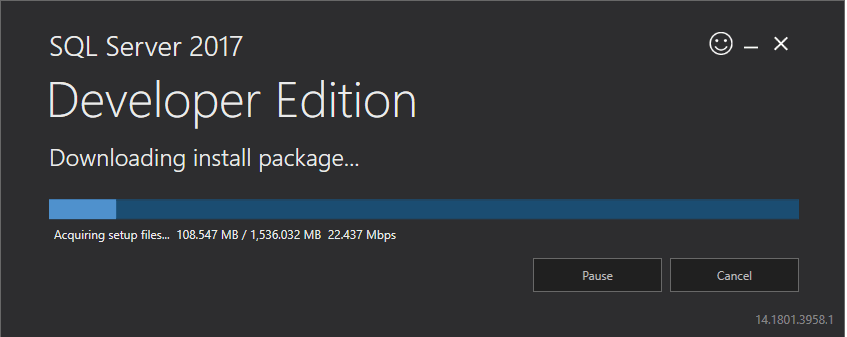
1. O instalador solicita que você selecione o tipo de instalação, escolha o tipo de instalação personalizada, que permite percorrer o assistente de instalação do SQL Server e selecionar os recursos que você deseja instalar.



2. Especifique a pasta para armazenar os arquivos de instalação do download e clique no botão Instalar.



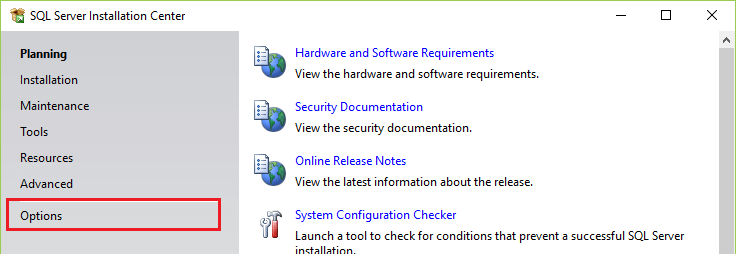
3. O instalador começa a baixar o pacote de instalação por um tempo.



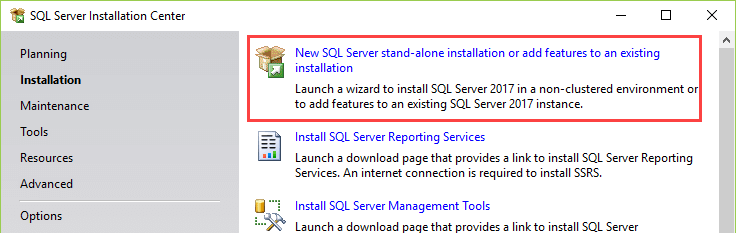
4. Após o download, abra a pasta que armazena o pacote de instalação e clique duas vezes no arquivo SETUP.exe.



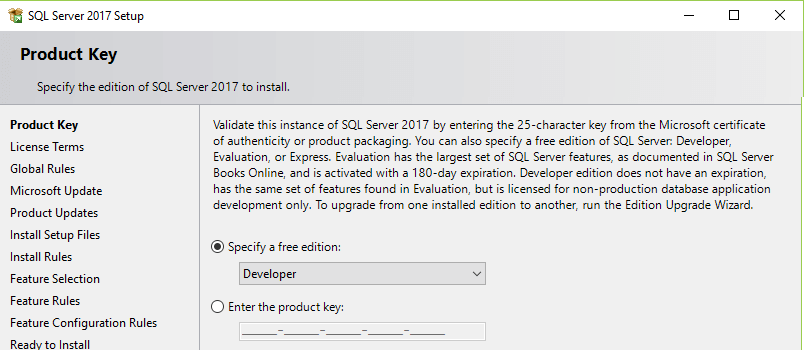
5. A seguinte janela é exibida; selecione a opção de instalação à esquerda.



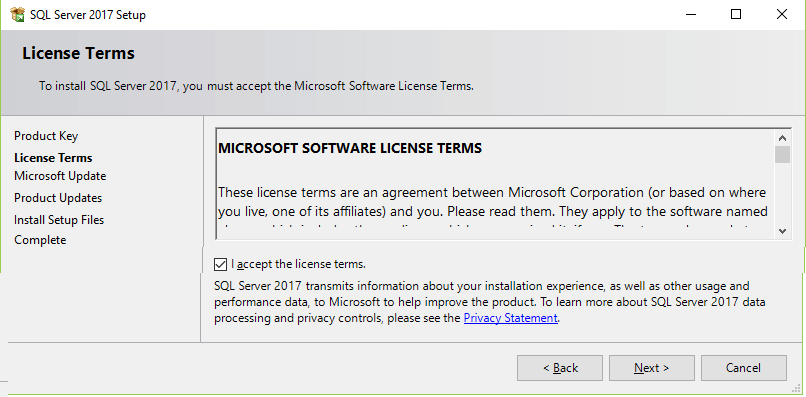
6. Clique no primeiro link para iniciar um assistente para instalar o SQL Server 2017.



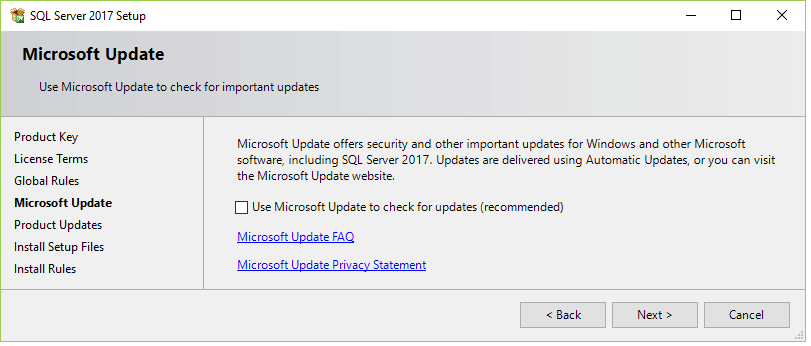
7. Especifique a edição que deseja instalar, selecione Developer edition e clique no botão Next.



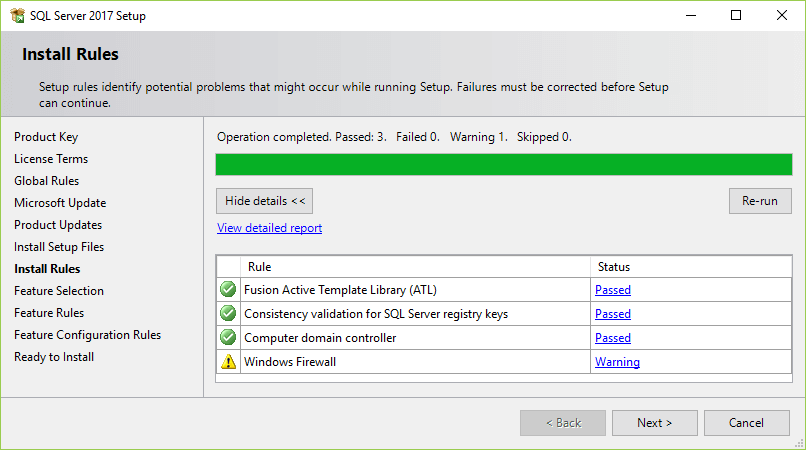
8. Selecione "Aceito os termos da licença" e clique no botão Avançar.



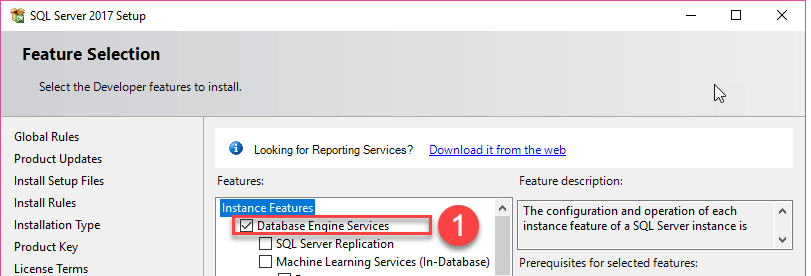
9. Desmarque a opção “Use o Microsoft Update para verificar se há atualizações (recomendadas)” para não receber informações de segurança, atualizações, etc do SQL Server e clique no botão Avançar.



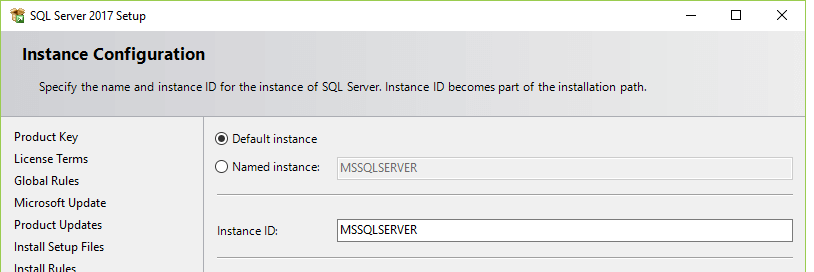
10. A instalação verifica os pré-requisitos antes da instalação. Se nenhum erro for encontrado, clique no botão Avançar.



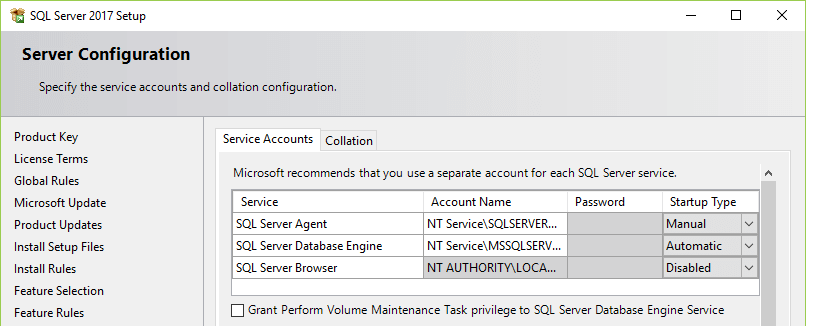
11. Selecione os recursos que você deseja instalar. Por enquanto, você só precisa dos Serviços do Mecanismo de Banco de Dados, basta marcar a caixa de seleção e clicar no botão Avançar para continuar



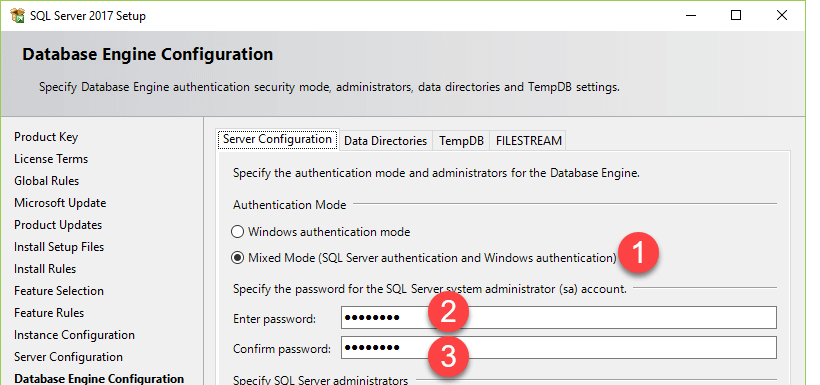
12. Especifique o nome e o ID de instalação da instância do SQL Server e clique no botão Avançar.



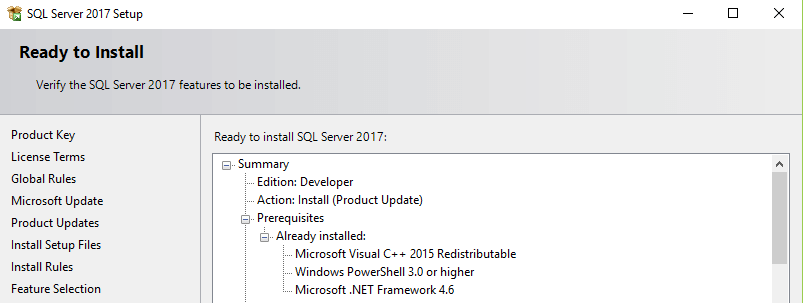
13. Especifique a conta de serviço e a configuração do agrupamento. Basta usar a configuração padrão e clique no botão Avançar.



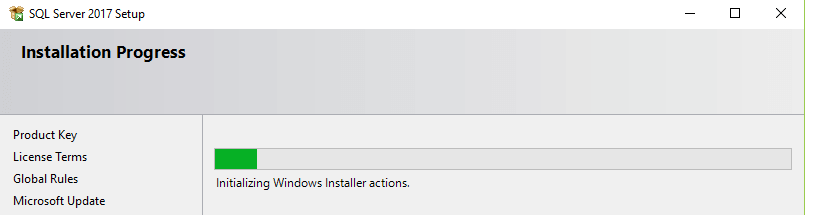
14. Especifique o modo de segurança do mecanismo de banco de dados, para simplificar coloque Autenticação Windows. Senão, primeiro, escolha Modo Misto. Em seguida, insira a senha da conta de administrador do sistema do SQL Server (sa). Em seguida, digite novamente a mesma senha para confirmá-la. Depois disso, clique no botão Adicionar usuário atual. Por fim, clique no botão Avançar.



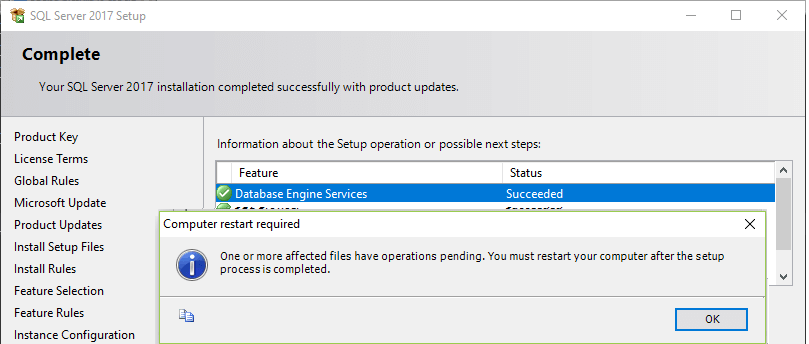
15. Verifique os recursos do SQL Server 2017 a serem instalados:



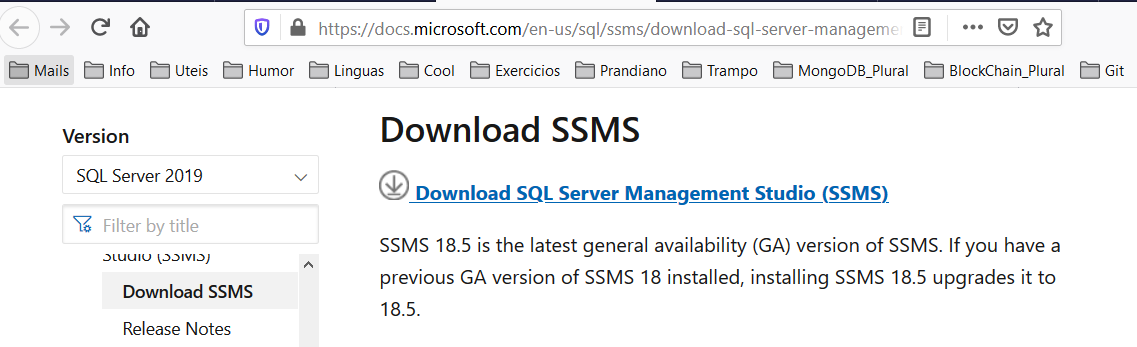
16. O instalador inicia o processo de instalação



17. Depois de concluída, a seguinte janela é exibida. Clique no botão OK.

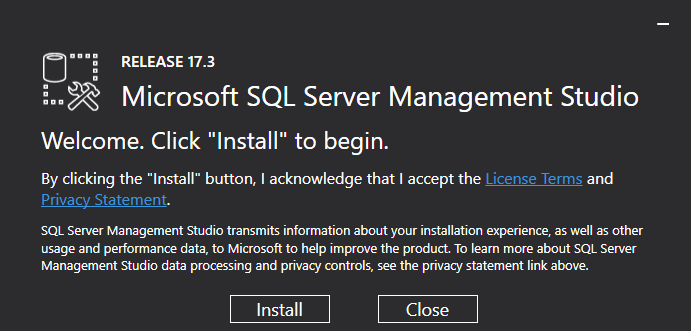


Para interagir com servidores SQL, você precisa instalar o SQL Server Management Studio (SSMS). O SQL Server Management Studio é um software para consultar, projetar e gerenciar o SQL Server no computador local ou na nuvem. Ele fornece ferramentas para configurar, monitorar e administrar instâncias do SQL Server. Primeiro, baixe o SSMS no site da Microsoft através do seguinte link:

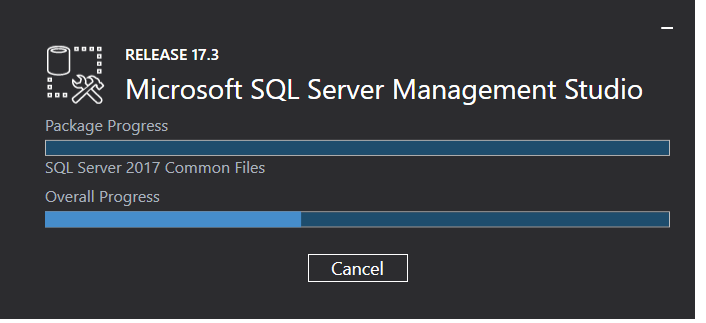


Segundo, clique duas vezes no arquivo de instalação SSMS-Setup-ENU.exe para iniciar a instalação. O processo de instalação do SMSS é direto, basta seguir a sequência da tela.

1. Clique no botão Instalar



2. Aguarde alguns minutos enquanto o instalador configura o software.



3. Quando a instalação estiver concluída, clique no botão Fechar



Agora, você deve ter o SQL Server 2017 e o SQL Server Management Studio instalados no seu computador. Em seguida, para conectar-se ao SQL Server usando o Microsoft SQL Server Management Studio, siga estas etapas:

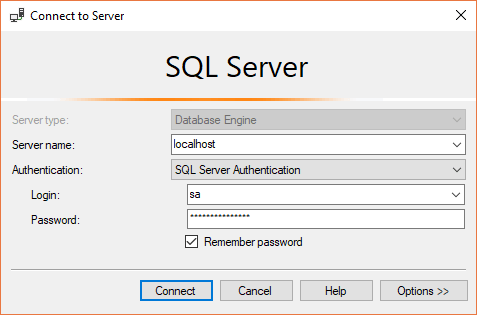
Primeiro, inicie o Microsoft SQL Server Management Studio no menu Iniciar:



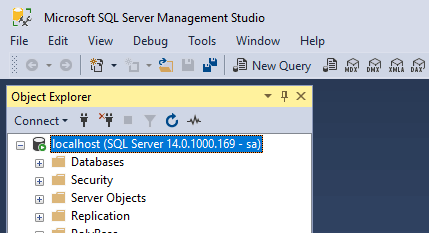
Em seguida, no menu Conectar no Pesquisador de Objetos, escolha o Mecanismo de Banco de Dados…



Em seguida, se você não optou por Autenticação Windows, insira as informações para o nome do servidor (host local), autenticação (autenticação do SQL Server) e senha do usuário sa e clique no botão Conectar para conectar-se ao SQL Server. Observe que você deve usar o usuário sa e a senha inseridos durante a instalação.



Se a conexão for estabelecida com sucesso, você verá o seguinte painel do Pesquisador de Objetos:



TRABALHANDO COM UM BANCO DE DADOS

No .NET Framework, você pode encontrar vários tipos de classes relacionadas a dados no System.Data.dll. Existem inúmeras classes base e interfaces definidas neste namespace que um provedor de dados deve implementar para permitir que o ADO.NET acesse um banco de dados. Quase todos os aplicativos usam bancos de dados, portanto você deve entender os conceitos que envolvem o ADO.NET, na qual existem três partes conceituais:

1. **Camada Conectada:** você se conecta explicitamente a um banco de dados e usa isso como armazenamento de dados subjacente. Você executa consultas usando o SQL (Structured Query Language) para criar, ler, atualizar e excluir dados (normalmente conhecidos como operações CRUD). Essas consultas são usadas pela infraestrutura do ADO.NET e encaminhadas para o banco de dados de sua escolha. Ao trabalhar da maneira conectada, você usa objetos de Conexão, objetos de Comando, objetos DataReader, entre outros.
2. **Camada Desconectada:** você trabalha com DataSets e DataTables que imitam a estrutura de um banco de dados relacional na memória. Você pode usá-los para trabalhar com dados offline e posteriormente sincronizá-los com um banco de dados quando estiver online. Um DataSet criado após a execução de uma consulta em um banco de dados conectado pode ser manipulado na memória e as alterações podem ser enviadas de volta ao armazenamento de dados usando um DataAdapter.
3. **Entity Framework (ORM)**

CAMADA CONECTADA

Em uma camada conectada, você se conecta a um banco de dados como fonte de dados e executa consultas escrevendo SQL. Essas consultas são usadas pelo ADO.NET e encaminhadas para o banco de dados de sua escolha. Para interagir com um banco de dados, você normalmente usa objetos Connection, Command e DataReader.

Provedoresde dados

Um provedor de dados do .NET Framework é usado para conectar-se a um banco de dados, executar comandos e trabalhar os dados resultantes. Os provedores de dados do .NET Framework fornecem uma camada fina que se integra a um banco de dados específico, para que você possa criar programas que possam trabalhar com diferentes tipos de bancos de dados sem precisar alterar nenhum código.

Existem muitos tipos diferentes de bancos de dados como o Microsoft SQL Server, Oracle e MySQL, entre outros. O .NET Framework oferece recursos para trabalhar com dados em todos esses tipos de bancos de dados de maneira padrão. Oracle, MySQL e outros principais sistemas de banco de dados têm seu próprio namespace e classes que implementam as classes e interfaces básicas do ADO.NET. Isso permite que você use sintaxe consistente em todas as bases de dados. Por exemplo, o namespace System.Data.SqlClient e o namespace System.Data.OleDb que contém os tipos que implementam as classes base e interfaces do ADO.NET para conectar-se a um banco de dados do SQL Server.

Conectando-se a um Banco de Dados

Depois de decidir o provedor de dados, é importante trabalhar com a conexão com um banco de dados específico. Você deve estabelecer uma conexão com seu banco de dados para interação adicional. O ADO.NET fornece classes para estabelecer conexão com um banco de dados específico.

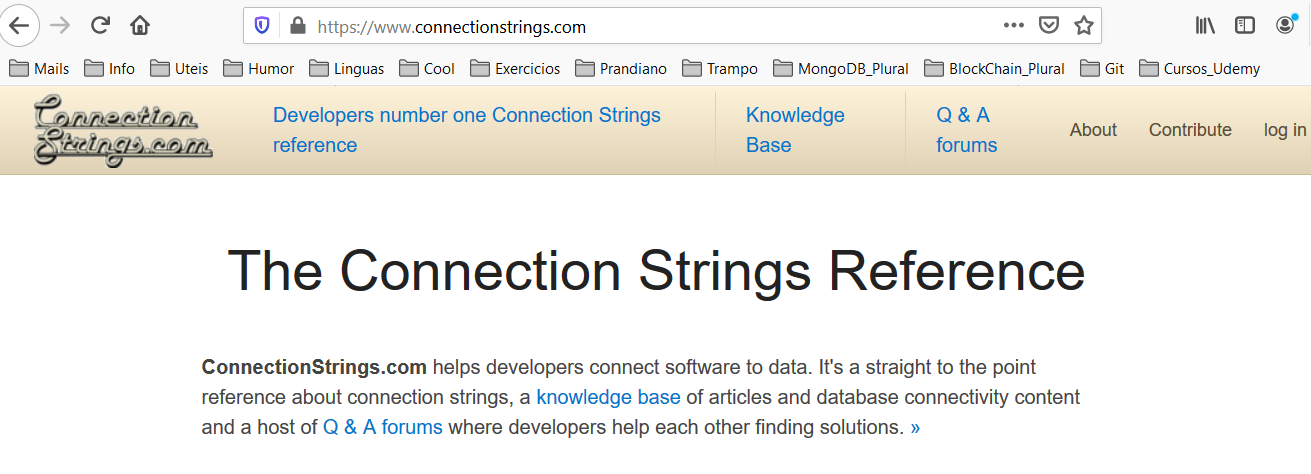
Um objeto de conexão é usado para abrir uma linha de comunicação com um banco de dados. O objeto SqlConnection é usado para conectar-se a um banco de dados do SQL Server. Essa classe, juntamente com a classe de conexão de qualquer provedor, herda da classe System.Data.Common.DBConnection. A Tabela a seguir lista as propriedades mais comuns para a classe DBConnection.

|  |  |
| --- | --- |
| **Propriedade** | **Descrição** |
| ConnectionString | Obtém ou define a sequência usada para abrir uma conexão com um banco de dados |
| ConnectionTimeout | Obtém o tempo em segundos que o sistema deve esperar ao estabelecer uma conexão com o banco de dados antes de gerar um erro |
| Database | Obtém o nome do banco de dados |
| DataSource | Obtém o nome do servidor de banco de dados |
| ServerVersion | Obtém a versão do servidor para o banco de dados |
| State | Obtém uma sequência que representa o estado da conexão, como Aberta ou Fechada |

A propriedade mais importante a ser observada é a propriedade ConnectionStringque contém todas as informações que o .NET Framework precisa saber:

* Local dos dados
* Nome do banco de dados
* Provedor/tipo de banco de dados
* Autenticação no banco de dados

Cada provedor tem pequenas variações nas configurações na cadeia de conexão, um ótimo recurso para consultar a sintaxe das cadeias de conexão para diferentes de bancos de dados é: <http://www.ConnectionStrings.com>



Para o SQL Server, uma cadeia de conexão possui a seguinte sintaxe.

connectionString = "Server=myServerAddress;Database=myDataBase; User Id=myUsername;Password=myPassword;";

Essa cadeia de conexão descreve onde o banco de dados está localizado, como é nomeado e alguns detalhes sobre como se autenticar no banco de dados. O formato básico de uma cadeia de conexão é uma série de pares de chave/valor conectados por um sinal de igual (=), todos separados por ponto e vírgula (;).

A classe DbConnection é uma classe base que fornece a funcionalidade relacionada à conexão. SqlConnection é derivado da classe DbConnection e pode ser usado para conexão com um banco de dados do SQL Server. A seguir, uma lista os métodos mais importantes para a classe DBConnection.

|  |  |
| --- | --- |
| **Método** | **Descrição** |
| BeginTransaction | Inicia uma transação de banco de dados |
| Close | Fecha a conexão com o banco de dados |
| GetSchema | Retorna um DataTable que contém as informações do esquema para a fonte de dados |
| Open | Abre a conexão com o banco de dados usando a cadeia de conexão |

O método Open é usado para estabelecer uma conexão com o banco de dados. Depois de ter uma conexão, você pode usar esse objeto junto com os outros objetos do ADO.NET para executar comandos no banco de dados. O código a seguir cria uma instância do objeto SqlConnection, define a propriedade ConnectionString e abre uma conexão com o banco de dados:

var connectionString = @"Persist Security Info = False; Integrated Security = true; Initial Catalog = Northwind; server = (local)";

using (SqlConnection connection = new SqlConnection(connectionString))

{

connection.Open();

// Execute operations against the database

} // Connection is automatically closed.

Como a classe DbConnection usa uma conexão de banco de dados real e não gerenciada, é importante garantir que você feche corretamente a conexão quando terminar. Ter conexões abertas por muito tempo é um problema, pois outros usuários não podem se conectar, pois o servidor permite um número específico de conexões e, se o limite for excedido, não permitirá que nenhum outro usuário se conecte devido à das conexões já atribuídas (que ainda estão abertas).Por esse motivo, a classe DbConnection implementa IDisposable, para que você possa determinar deterministicamente a conexão e liberar qualquer objeto não gerenciado associado.

Quando você precisar criar uma cadeia de conexão dinamicamente, poderá usar uma das várias classes DbConnectionStringBuilder. Por exemplo, você pode usar OracleConnectionStringBuilder para criar uma cadeia de conexão para um banco de dados Oracle e SqlConnectionStringBuilder para um banco de dados Microsoft SQL Server. O código abaixo mostra como usar o SqlConnectionStringBuilder para criar uma nova cadeia de conexão programaticamente.

var sqlConnectionStringBuilder = new SqlConnectionStringBuilder();

sqlConnectionStringBuilder.DataSource = @"localhost";

sqlConnectionStringBuilder.InitialCatalog = "School";

sqlConnectionStringBuilder.IntegratedSecurity = true;

string connectionString = sqlConnectionStringBuilder.ToString();

A propriedade IntegratedSecurity obtém ou define um valor booliano que indica se a ID de Usuário e a Senha são especificadas na conexão(quando false) ou se as atuais credenciais da conta do Windows são usadas para autenticação(quando true).Codificar uma string de conexão geralmente não é a melhor maneira de configurar um aplicativo. Quando você implanta um aplicativo em um ambiente de teste, preparo ou produção, a cadeia de conexão geralmente precisa ser alterada e, na maioria das vezes, isso é feito por um profissional de TI que não possui (e não quer ter!) qualquer experiência de programação.

Por esse motivo, você pode armazenar facilmente cadeias de conexão em um arquivo de configuração externo. Dependendo do tipo de aplicativo, você usa o arquivo app.config ou web.config. Esses arquivos podem ser usados para armazenar todos os tipos de definições de configuração para um aplicativo. Você também pode usar outros arquivos de configuração, que você faz referência a partir desses arquivos.Um exemplo de uma cadeia de conexão no arquivo app.config de um aplicativo é mostrado abaixo.

<?xmlversion="1.0"encoding="utf-8" ?>

<configuration>

<configSections>

</configSections>

<connectionStrings>

<addname="PeopleConnection"

connectionString="Data Source=localhost;Initial Catalog=School;Integrated Security=True"

providerName="System.Data.SqlClient" />

</connectionStrings>

<startup>

<supportedRuntimeversion="v4.0"sku=".NETFramework,Version=v4.6.1" />

</startup>

</configuration>

Se você deseja usar a cadeia de conexão no seu aplicativo, pode usar a propriedade ConfigurationManager.ConnectionStrings do System.Configuration.dll. Você pode acessar cadeias de conexão por índice e por nome. A exemplo a seguir mostra como acessar uma cadeia de conexão por nome e usá-la para abrir uma conexão com o banco de dados.

using System.Configuration;

string connectionString = ConfigurationManager.ConnectionStrings["ProgConnection"].ConnectionString;

Conectar-se a um banco de dados é uma operação demorada. Ter uma conexão aberta por muito tempo também é um problema, pois pode levar outros usuários a não conseguirem se conectar. Para minimizar os custos de abrir e fechar repetidamente conexões, o ADO.NET aplica uma otimização chamada de pool de conexões. Ao usar o SQL Server, um pool de conexões é mantido pelo seu aplicativo. Quando uma nova conexão é solicitada, o .NET Framework verifica se há uma conexão aberta no pool. Se houver, não será necessário abrir uma nova conexão e executar todas as etapas iniciais da configuração. Por padrão, o pool de conexões está ativado, o que pode proporcionar uma enorme melhoria de desempenho.

Comandos SqlCommand

ADO.NET fornece a classe SqlCommand, usado para executar instruções (comandos/consultas) no banco de dados. Ao usar esta classe, você pode executar comandos de inserção, exclusão, atualização ou procedimento armazenado.

SqlCommand command = new SqlCommand("SELECT \* FROM Student", connection);

SqlCommand requer um comando (instrução/consulta) para executar e uma conexão (cadeia de conexão) na qual um comando escrito será executado. Esse código sabe qual comando executar em qual conexão (em outras palavras, o que fazer usando qual caminho).

Método ExecuteReader - Lendo dados

Depois que a conexão for estabelecida, você poderá começar a enviar consultas ao banco de dados. As consultas são construídas usando SQL, que é uma sintaxe especial otimizada para trabalhar com bancos de dados. Digamos que você esteja trabalhando com uma tabela do banco de dados chamada People com a estrutura abaixo:

|  |  |  |
| --- | --- | --- |
| **Column name** | **Data type** | **Nullable** |
| id | int, Primary Key | False |
| FirstName | varchar(30) | False |
| MiddleName | varchar(30) | True |
| LastName | varchar(30) | False |

Se você deseja selecionar algumas linhas nesta tabela, primeiro precisa definir uma instrução SQL. Uma consulta SQL pode ser executada usando um objeto SqlCommand. Use o método ExecuteReader para recuperar resultados do banco de dados, como quando você usa uma instrução Select. Esse método retorna um objeto SqlDataReader que permanece conectado a um banco de dados o tempo todo em que o leitor está aberto. SqlDataReader é um conjunto de resultados somente para frente, o que significa que você não pode mover para nenhum registro anterior e pode ler somente um registro por vez. Você pode ler a coluna específica de uma tabela pelo número do índice ou nome da coluna. O exemplo abaixo mostra o uso do método ExecuteReader:

string connectionString = ConfigurationManager.ConnectionStrings["PeopleConnection"].ConnectionString;

using (SqlConnection connection = new SqlConnection(connectionString))

{

connection.Open();

SqlCommand cmd = new SqlCommand();

cmd.Connection = connection;

cmd.CommandType = CommandType.Text;

cmd.CommandText = "SELECT \* FROM People";

using (SqlDataReader dr = cmd.ExecuteReader())

{

if (dr.HasRows)

{

while (dr.Read())

{

Console.WriteLine(string.Format("Id: {0}", dr[0]));

Console.WriteLine(string.Format("First Name: {0}, Last Name: {1}", dr["FirstName"], dr["LastName"]));

}

}

}

}

O método abaixo mostra como você pode usá-lo para percorrer as linhas na tabela People e exibir os resultados. O ADO.NET também suporta as palavras-chave async/wait, para que você possa executar consultas de forma assíncrona.

publicstaticasync Task SelectDataFromTable()

{

string connectionString = ConfigurationManager.ConnectionStrings["PeopleConnection"].ConnectionString;

using (SqlConnection connection = new SqlConnection(connectionString))

{

SqlCommand command = new SqlCommand("SELECT \* FROM People", connection);

await connection.OpenAsync();

SqlDataReader dataReader = await command.ExecuteReaderAsync();

while (await dataReader.ReadAsync())

{

string formatStringWithMiddleName = "Person({0}) is named {1}{2}{3}";

if (!(dataReader["middlename"] == null))

{

Console.WriteLine(formatStringWithMiddleName, dataReader["id"],

dataReader["firstname"], dataReader["middlename"], dataReader["lastname"]);

}

}

dataReader.Close();

}

}

O SqlDataReader fornece algumas propriedades listadas abaixo, as duas últimas propriedades são indexadores que usamos nos exemplos acima. Eles são usados para buscar um valor de coluna específico com base em seu nome (string) ou número de índice (int).

|  |  |
| --- | --- |
| **Propriedade** | **Descrição** |
| FieldCount | Retorna o número de colunas na linha atual. |
| HasRows | Retorna um booleano indicando se o leitor possui alguma linha. |
| IsClosed | Retorna um booleano indicando se o leitor está fechado. |
| Item[Int32] | Este é um indexador que retorna a coluna com base no índice. |
| Item[String] | Este é um indexador que retorna a coluna com base no nome da coluna. |

Não há uma propriedade Count para o número de linhas no conjunto de resultados. A única maneira de obter a contagem é atravessar o datareader. Os indexadores, que são as propriedades do item, retornam um objeto. Eles permitem que você obtenha o valor de uma coluna pelo índice da coluna ou pelo nome. Cabe a você converter o objeto para o tipo certo ao usar os indexadores

O método Read lê o registro de um banco de dados e está pronto para ser lido para o próximo, se existir. Enquanto o loop itera e a execução ocorre para o próximo registro, Read retorna true, e assim por diante até que haja o último registro, Read retorna false, e o loop termina. Você deve fechar o objeto leitor e, em seguida, feche o objeto de conexão. Esquecer de fechar a conexão pode prejudicar o desempenho. Você pode usar o bloco "using" para evitar essas coisas.

O SqlDataReader oferece alguns métodos que podem mapear o valor de uma coluna para o tipo CLR correspondente. Por exemplo, você pode chamar métodos como GetInt32 (int index), GetGuid (int index), GetString (int index) e assim por diante. A seguir, uma lista métodos de comumente usado para a classe System.Data.Common.DBDataReader:

|  |  |
| --- | --- |
| **Método** | **Descrição** |
| Close | Fecha o objeto |
| GetBoolean | Retorna o valor da coluna especificada como um valor booleano |
| GetByte | Retorna o valor da coluna especificada como um byte |
| GetChar | Retorna o valor da coluna especificada como um caractere |
| GetDateTime | Retorna o valor da coluna especificada como um objeto DateTime |
| GetDecimal | Retorna o valor da coluna especificada como um objeto Decimal |
| GetDouble | Retorna o valor da coluna especificada como um objetoDouble |
| GetFieldType | Retorna o tipo de dados da coluna especiﬁcada |
| GetFieldValue<T> | Retorna o valor da coluna especificada como um tipo |
| GetFloat | Retorna o valor da coluna especificada como um objetoFloat |
| GetGuid | Retorna o valor da coluna especificada como um GUID |
| GetInt16 | Retorna o valor da coluna especificada como um número inteiro de 16 bits |
| GetInt32 | Retorna o valor da coluna especificada como um número inteiro de 32 bits |
| GetInt64 | Retorna o valor da coluna especificada como um número inteiro de 64 bits |
| GetName | Retorna o nome da coluna especi ﬁ cada, dada a posição ordinal |
| GetOrdinal | Retorna a posição ordinal de uma coluna, com o nome da coluna |
| GetSchemaTable | Retorna um DataTable que descreve os metadados da coluna |
| GetString | Retorna o valor da coluna especificada como uma string |
| GetValue | Retorna o valor da coluna especificada como um objeto |
| GetValues | Preenche uma matriz de objetos com os valores das colunas |
| NextResult | Move o cursor para o próximo conjunto de resultados no leitor |
| IsDBNull | Retorna um booleano para indicar se a coluna especificada contém um valor nulo |
| Read | Avança o cursor para o próximo registro |

Vários métodos GetTYPE permitem que você use um índice de coluna para obter o valor do leitor de dados e converter o valor no tipo especificado. Isso funciona apenas com índices de coluna e não com nomes. Se a ordem na sua cláusula SELECT mudar, seus índices também deverão mudar.

Ao executar uma consulta, você também pode agrupar várias instruções SQL juntas. O Sql-DataReader retornado contém vários conjuntos de resultados. Você pode avançar para o próximo conjunto de resultados chamando NextResult ou NextResultAsync no seu objeto SqlDataReader. O código abaixo mostra como executar uma consulta que executa dois conjuntos de resultados.

publicstaticasync Task SelectMultipleResultSets()

{

string connectionString = ConfigurationManager.

ConnectionStrings["PeopleConnection"].ConnectionString;

using (SqlConnection connection = new SqlConnection(connectionString))

{

SqlCommand command = new SqlCommand(@"SELECT \* FROM People;

SELECT TOP 1 \* FROM People ORDER BY LastName", connection);

await connection.OpenAsync();

SqlDataReader dataReader = await command.ExecuteReaderAsync();

await ReadQueryResults(dataReader);

await dataReader.NextResultAsync(); // Move to the next result set

await ReadQueryResults(dataReader);

dataReader.Close();

}

}

privatestaticasync Task ReadQueryResults(SqlDataReader dataReader)

{

while (await dataReader.ReadAsync())

{

string formatStringWithMiddleName = @"Person({0}) is named {1}{2}{3}";

if (!(dataReader["middlename"] == null))

{

Console.WriteLine(formatStringWithMiddleName, dataReader["id"],

dataReader["firstname"], dataReader["middlename"], dataReader["lastname"]);

}

}

}

Você pode usar a classe OleDbDataReader no lugar da classe SqlDataReader para recuperar dados do Microsoft Access (arquivo.accdb apresentado no Access 2007, formato de arquivo.mdb versões anteriores).

DBNull vs null

Esteja ciente de que se uma coluna contiver um valor nulo, o objeto retornado do indexador não será nulo; é DBNull.Value. Isso pode ser a causa de muitos erros, se você não entender a diferença entre os dois.DBNull é uma classe singleton, o que significa que somente essa instância dessa classe pode existir.

Se um campo de banco de dados estiver faltando, você poderá usar a propriedade DBNull.Value para atribuir explicitamente um valor de objeto de DBNull ao campo. No entanto, a maioria dos provedores de dados faz isso automaticamente.

Para avaliar os campos de banco de dados para determinar se seus valores são DBNull, você pode passar o valor do campo para o método DBNull.Value.Equals. No entanto, esse método raramente é usado porque há várias outras maneiras de avaliar um campo de banco de dados para os que estão faltando. Isso inclui a função Visual Basic IsDBNull, o método Convert.IsDBNull, o método DataTableReader.IsDBNull, o método IDataRecord.IsDBNull e vários outros métodos.

var middlename = AddFieldValue(dr, "MiddleName");

Console.WriteLine(string.Format("MiddleName: {0}", middlename));

privatestaticstring AddFieldValue(SqlDataReader row, string fieldName)

{

if (!DBNull.Value.Equals(row[fieldName]))

return (string)row[fieldName] + " ";

elseif (Convert.IsDBNull(row[fieldName]))

return"NA";

else

return String.Empty;

}

Método ExecuteScalar - Lendo uma linha

O método ExecuteScalar é usado quando você sabe que seu conjunto de resultados contém apenas uma única coluna com uma única linha. Isso é ótimo quando sua consulta retorna o resultado de uma função agregada, como SUM ou AVG. O código a seguir chama o método ExecuteScalar e retorna o Count of records na tabela People:

string connectionString = ConfigurationManager.ConnectionStrings["PeopleConnection"].ConnectionString;

using (SqlConnection connection = new SqlConnection(connectionString))

{

connection.Open();

SqlCommand cmd = new SqlCommand();

cmd.Connection = connection;

cmd.CommandType = CommandType.Text;

cmd.CommandText = "SELECT COUNT(\*) FROM People";

object obj = cmd.ExecuteScalar();

Console.WriteLine(string.Format("Count: {0}", obj.ToString()));

}

O método ExecuteScalar sempre retorna um objeto; portanto, cabe a você converter esse valor para o tipo certo quando desejar usá-lo.

Método ExecuteXmlReader

O método ExecuteXmlReader retorna um XMLReader, que permite representar os dados como XML. O código a seguir retorna os dados da tabela People em um objeto XmlReader:

string connectionString = ConfigurationManager.ConnectionStrings["PeopleConnection"].ConnectionString;

using (SqlConnection connection = new SqlConnection(connectionString))

{

connection.Open();

SqlCommand cmd = new SqlCommand();

cmd.Connection = connection;

cmd.CommandType = CommandType.Text;

cmd.CommandText = "SELECT \* FROM People FOR XML AUTO, XMLDATA";

System.Xml.XmlReader xmlreader = cmd.ExecuteXmlReader();

while (xmlreader.Read())

{

var xml = xmlreader.ReadOuterXml();

xml = xml.Replace(@"/>", @"/>" + Environment.NewLine);

Console.WriteLine(xml);

}

}

A instrução SQL foi alterada e incluiu a cláusula FOR XML AUTO, XMLDATA. Observe que o esquema é retornado junto com os dados neste exemplo. O resultado XML desta consulta é

<Schema name="Schema1" xmlns="urn:schemas-microsoft-com:xml-data" xmlns:dt="urn:schemas-microsoft-com:datatypes"><ElementType name="People" content="empty" model="closed"><AttributeType name="id" dt:type="i4" />

<AttributeType name="FirstName" dt:type="string" />

<AttributeType name="MiddleName" dt:type="string" />

<AttributeType name="LastName" dt:type="string" />

<attribute type="id" />

<attribute type="FirstName" />

<attribute type="MiddleName" />

<attribute type="LastName" />

</ElementType></Schema>

<People xmlns="x-schema:#Schema1" id="2" FirstName="Mary" LastName="Shelley" />

<People xmlns="x-schema:#Schema1" id="4" FirstName="Lady" LastName="Gaga" />

Método ExecuteNonQuery

O método ExecuteNonQuery é usado para executar instruções no banco de dados que não retornam conjuntos de resultados. O método ExecuteNonQuery() retorna somente o número de linhas afetadas.Por exemplo, uma instrução de inserção, atualização ou exclusão não retorna nenhum registro. Eles simplesmente executam a instrução em uma tabela. O código a seguir demonstra como executar uma instrução de inserção no banco de dados:

string connectionString = ConfigurationManager.ConnectionStrings["PeopleConnection"].ConnectionString;

using (SqlConnection connection = new SqlConnection(connectionString))

{

connection.Open();

SqlCommand cmd = new SqlCommand();

cmd.Connection = connection;

cmd.CommandType = CommandType.Text;

cmd.CommandText = "INSERT INTO People (FirstName, LastName) " + "VALUES ('Joe', 'Smith')";

int result = cmd.ExecuteNonQuery();

if (result > 0) Console.WriteLine("Data is Inserted");

else Console.WriteLine("Error while inserting");

}

Observe as três propriedades do objeto de comando que precisavam ser definidas antes de chamar o método ExecuteNonQuery. A primeira é a propriedade Connection. Isso deve ser definido como uma conexão aberta. Isso informa ao objeto de comando qual banco de dados usar ao executar o texto contido na propriedade CommandText. Neste exemplo, você usa o SQL embutido, e é por isso que a propriedade CommandType está definida como CommandType.Text. Se você usar um procedimento armazenado para inserir um registro People, seria necessário alterar o CommandType para CommandType.StoredProcedure e defina CommandText como nome do procedimento armazenado. Por exemplo, suponha que você tenha o seguinte procedimento armazenado que insere um registro na tabela People:

CREATEPROCEDURE [dbo].[PersonInsert]

@FirstName varchar(50),

@LastName varchar(50)

AS

BEGIN

INSERTINTO PEOPLE(FirstName, LastName)VALUES (@FirstName, @LastName)

END

O código a seguir executa o procedimento armazenado e passa nos parâmetros @FirstName e @LastName:

SqlCommand cmd\_sp = new SqlCommand();

cmd\_sp.Connection = connection;

cmd\_sp.CommandType = CommandType.StoredProcedure;

cmd\_sp.CommandText = "PersonInsert";

cmd\_sp.Parameters.Add(new SqlParameter("@FirstName", "Sasha"));

cmd\_sp.Parameters.Add(new SqlParameter("@LastName", "Gray"));

int result\_sp = cmd\_sp.ExecuteNonQuery();

O objeto Command possui uma propriedade Parameters que você usa para passar parâmetros para o procedimento armazenado. Observe também que o método ExecuteNonQuery retorna 1 como número de linhas afetadas pelo comando quando é executado uma Store Procedure.

Atualizando dados

Além de selecionar dados, você também pode atualizar os dados no banco de dados. Você pode criar novos registros, atualizar os existentes ou remover completamente um registro. Você pode fazer tudo isso usando uma conexão e um objeto de comando. Mas, em vez de retornar um leitor com as linhas resultantes, você recebe um valor inteiro de volta que mostra quantas linhas são afetadas pela sua última consulta. Isso significa que, se você executar um procedimento armazenado ou encadear várias instruções SQL juntas, obterá apenas o número de registros afetados pela última consulta, não em todas as consultas no seu procedimento. Ao verificar o resultado, você pode determinar se a quantidade correta de linhas foi afetada. Você usa o método ExecuteNonQuery ou ExecuteNonQueryAsync para fazer isso, conforme mostrado abaixo.

publicstaticasync Task UpdateRows()

{

string connectionString = ConfigurationManager.

ConnectionStrings["PeopleConnection"].ConnectionString;

using (SqlConnection connection = new SqlConnection(connectionString))

{

SqlCommand command = new SqlCommand("UPDATE People SET FirstName ='John' WHERE id = 3", connection);

await connection.OpenAsync();

int numberOfUpdatedRows = await command.ExecuteNonQueryAsync();

Console.WriteLine("Updated {0} rows", numberOfUpdatedRows);

}

}

Usando parâmetros

Até agora, você viu apenas consultas que já estavam completamente construídas em tempo de compilação. No mundo real, no entanto, muitas vezes você precisa usar valores dinâmicos ao executar uma consulta. Como uma consulta SQL nada mais é do que uma sequência simples, você pode ser tentado a concatenar várias seqüências para criar sua consulta. No entanto, esteja ciente de que este é um enorme risco de segurança. Por exemplo, digamos que você tenha um formulário no qual as pessoas possam preencher os nomes que você inserirá na tabela People. Sua consulta será mais ou menos assim:

SqlCommand command1 = new SqlCommand("INSERT INTO People VALUES('Diane', 'Birch', null)", connection);

Agora, você pode ler os valores que um usuário digita e depois construir manualmente essa sequência.Mas o que aconteceria se um usuário digitasse '); DELETE FROM People; -- invés de passar null para o útimo parâmetro, como no exemplo abaixo:

SqlCommand commanddel = new SqlCommand("INSERT INTO People VALUES('John', 'Birch', ''); DELETE FROM People; --'", connection);

Depois de executar esta consulta, todos os dados na sua tabela Peopleserão excluídos. Essa falha de segurança é conhecida como injeção SQL.

Para se proteger contra a injeção de SQL, você nunca deve usar diretamente a entrada do usuário em suas seqüências de caracteres SQL. Em vez de criar manualmente a consulta SQL correta, você pode usar instruções SQL com parâmetros.

publicstaticasync Task InsertRowWithParameterizedQuery()

{

string connectionString = ConfigurationManager.

ConnectionStrings["PeopleConnection"].ConnectionString;

using (SqlConnection connection = new SqlConnection(connectionString))

{

SqlCommand command =

new SqlCommand(@"INSERT INTO People([FirstName], [LastName], [MiddleName])

VALUES(@firstName, @lastName, @middleName)", connection);

await connection.OpenAsync();

command.Parameters.AddWithValue("@firstName", "John");

command.Parameters.AddWithValue("@lastName", "Doe");

command.Parameters.AddWithValue("@middleName", "Little");

int numberOfInsertedRows = await command.ExecuteNonQueryAsync();

Console.WriteLine("Inserted {0} rows", numberOfInsertedRows);

}

}

Essas consultas parametrizadas podem ser usadas ao executar consultas de seleção, atualização, inserção e exclusão. Além de serem muito mais seguros, eles também oferecem melhor desempenho. Como o banco de dados obtém uma consulta mais genérica, ele pode encontrar mais facilmente um plano de execução pré-compilado para executar sua consulta.

Usando transações

Ao trabalhar com um banco de dados, as coisas podem dar errado. Talvez você esteja executando várias consultas que devem ser agrupadas. Se o último falhar, os anteriores já serão executados e, de repente, seus dados poderão ficar corrompidos.Por isso, o .NET Framework ajuda você nas transações. Uma transação possui quatro propriedades principais que são chamadas de ACID:

* **Atomicidade**: Todas as operações são agrupadas. Se alguma falha, todas falham.
* **Consistência**: As transações levam o banco de dados de um estado válido para outro.
* **Isolamento**: As transações podem operar independentemente uma da outra. Várias transações simultâneas não se influenciam. Será como se eles fossem executados em série.
* **Durabilidade**: O resultado de uma transação confirmada é sempre armazenado permanentemente, mesmo se o banco de dados travar imediatamente depois.

Ao trabalhar com transações, é mais fácil usar a classe TransactionScope. O escopo da transação oferece uma maneira fácil de trabalhar com transações sem exigir que você interaja com a própria transação.

using System.Transactions;

publicasync Task InsertRowTransactionScope()

{

string connectionString = ConfigurationManager.

ConnectionStrings["ProgrammingInCSharpConnection"].ConnectionString;

using (TransactionScope transactionScope = new TransactionScope())

{

using (SqlConnection connection = new SqlConnection(connectionString))

{

SqlCommand command1 = new SqlCommand(@"INSERT INTO People([FirstName], [LastName], [MiddleInitial])

VALUES(‘John’, ‘Doe’, null)", connection);

SqlCommand command2 = new SqlCommand(@"INSERT INTO People([FirstName], [LastName], [MiddleInitial])

VALUES(‘Jane’, ‘Doe’, null)", connection);

await connection.OpenAsync();

command1.ExecuteNonQuery();

command2.ExecuteNonQuery();

}

transactionScope.Complete();

}

}

Parece inofensivo, mas gera um System.InvalidOperationException: um TransactionScope deve ser descartado no mesmo thread que foi criado.O motivo é que o TransactionScope não flui de um thread para outro por padrão. Para corrigir isso, precisamos usar TransactionScopeAsyncFlowOption.Enabled:

using (TransactionScope transactionScope = new TransactionScope(TransactionScopeAsyncFlowOption.Enabled))

{

using (SqlConnection connection = new SqlConnection(connectionString))

{

//....

}

//....

}

Se ocorrer uma exceção dentro do TransactionScope, a transação inteira será revertida.Se nada der errado, use TransactionScope.Complete para concluir a transação. É importante usar o TransactionScope dentro de uma instrução using para que seja automaticamente descartado quando não for mais necessário.

Um TransactionScope também pode ser construído passando uma enumeração TransactionScopeOption para ele. Usando essa enumeração, você pode definir o comportamento do TransactionScope.Um TransactionScope pode ser construído com três opções:

1. Necessário Participe da transação do ambiente ou crie uma nova, se ela não existir. Se houver uma transação ambiente, essa transação controlará quando a transação for concluída. Esse é o escopo padrão.
2. RequerNovo Inicie uma nova transação.
3. Suprimir Não participe de nenhuma transação.

Ao usar o TransactionScope, o .NET Framework gerencia automaticamente a transação para você. Se sua transação usar conexões aninhadas, vários bancos de dados ou vários tipos de recursos, sua transação será promovida para uma transação distribuída. A promoção para uma transação distribuída pode ter um enorme impacto no desempenho, se não for necessário. Se possível, tente evitar transações distribuídas. Se você precisar deles, o .NET Framework gerencia isso para você.

CAMADA DESCONECTADA

Em uma camada desconectada, você normalmente usa DataSets e DataTables que copiam a estrutura de um banco de dados relacional na memória. Um DataSet é criado no resultado de uma execução de uma consulta em um banco de dados conectado. Ele pode ser manipulado na memória e as alterações em um banco de dados ocorrem usando o DataAdapter. DataTable e DataSets são outra maneira de recuperar resultados de um banco de dados.

DataTable

DataTable é o mesmo que DataReader, exceto pelo fato de estar desconectado do banco de dados; você pode mover o cursor para frente e para trás; e você pode atualizar dados no DataTable, reconectar-se ao banco de dados e confirmar as alterações.

DataSet

Um DataSet é um contêiner para um ou mais DataTables. Você pode executar uma declaração SQL que retorna vários conjuntos de resultados e cada um pode estar contido no DataSet. Você pode filtrar, classificar ou atualizar os dados na memória.

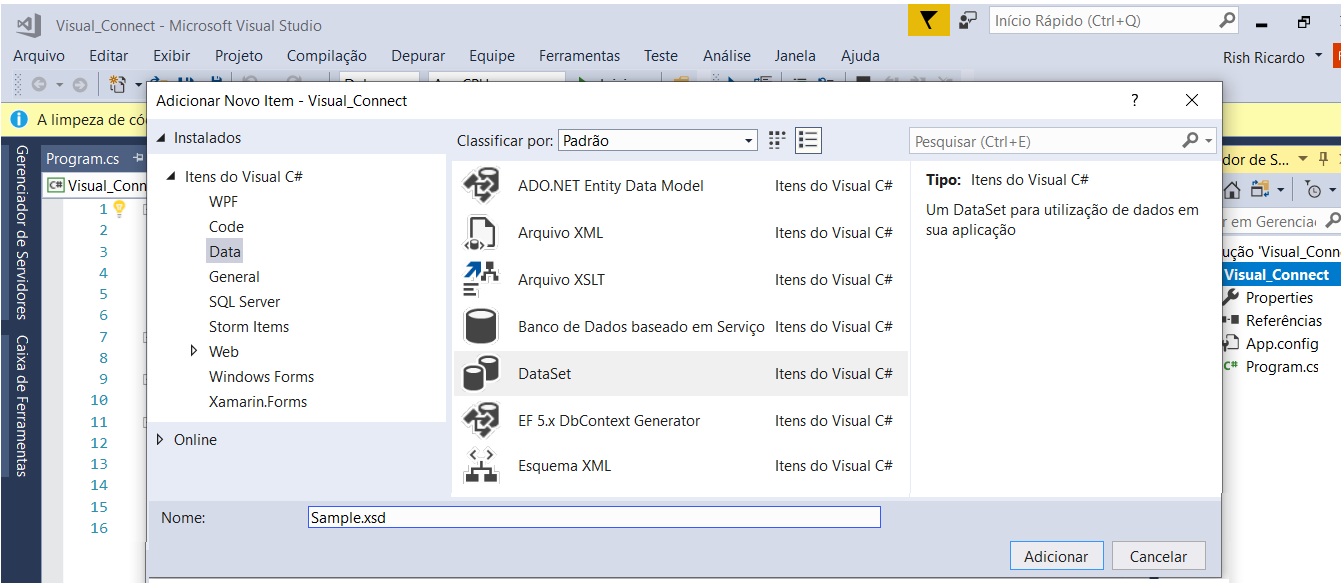
Os DataSets podem ser tipados ou não. Um DataSet tipado é um dataset que é derivado de uma classe DataSet e que usa a informação contida em um arquivo de esquema XML (.xsd) para gerar uma nova classe. Toda a informação do esquema (tabelas, colunas, linhas, etc.) é gerada e compilada neste nova classe DataSet. Como esta nova classe é derivada (herdar) da classe DataSet ela assuem toda a funcionalidade da classe DataSet.

Um DataSet não tipado não possui um corresponde arquivo de esquema, ele também possui tabelas, colunas, linhas, etc mas que são expostas somente como uma coleção. Você é quem deverá informar os nomes dos itens presentes nas coleções. Isto pode levar a que os erros sejam detectados somente na hora da compilação. Por exemplo, a linha de código abaixo tem um erro de digitação no nome da tabela que será detectado somente quando você for compilar o código: Ex: dim dt As Datatable = ds.Tables("Produtos"). Vejamos a seguir a comparação entre dois códigos que acessam dados. Um usa DataSets tipados e outro não tipado:

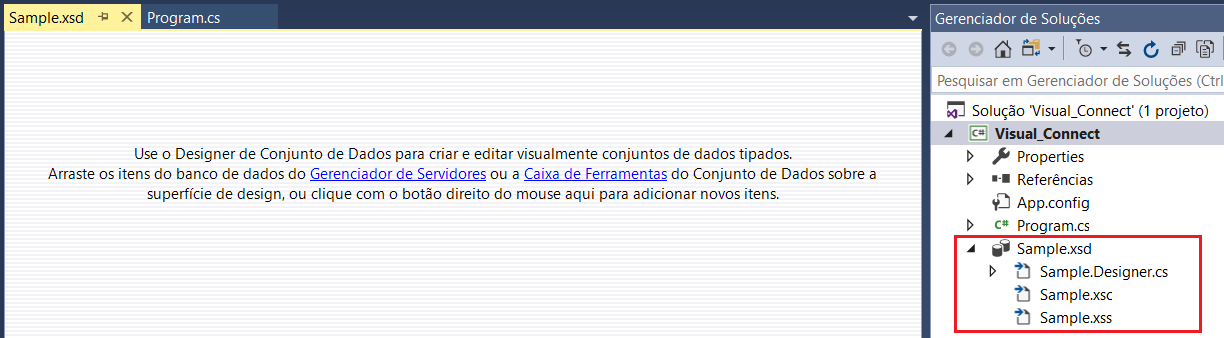
|  |  |
| --- | --- |
| **DataSet** | **Código** |
| Tipados | string s = dsCustomersOrders1.Customers(0).CustomerID; |
| Não Tipados | string s = (string)dsCustomersOrders1.Tables("Customers").Rows(0).Item("CustomerID"); |

Além de ser mais fácil de usar um DataSet tipado permite que você use o recurso da intellisense no seu código ao usar o editor do Visual Studio. É claro que haverá ocasiões que você vai ter que usar um DataSet não tipado. Vamos criar e usar um DataSet tipado para exibir o resultado de uma consulta SQL no banco de dados, mas poderia ser uma consulta a um banco de dados Northwind.mdf:

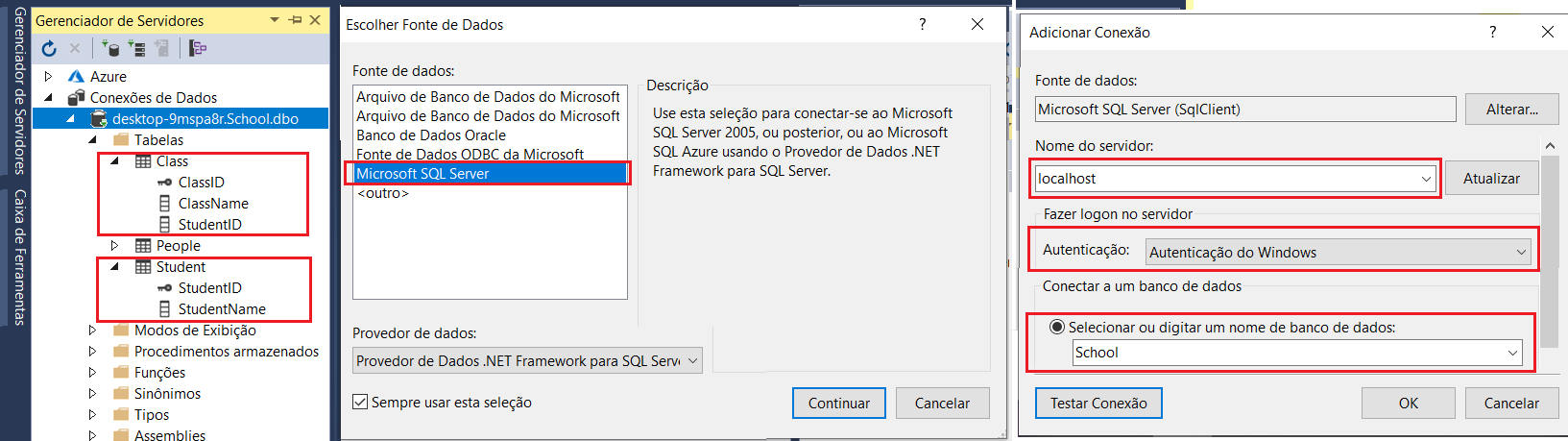
Etapa 1: Clique com o botão direito do mouse no seu projeto no Visual Studio e clique em Adicionar >> Novo Item. Selecione DataSet e nomeie-o como quiser (Sample.xsd neste exemplo).



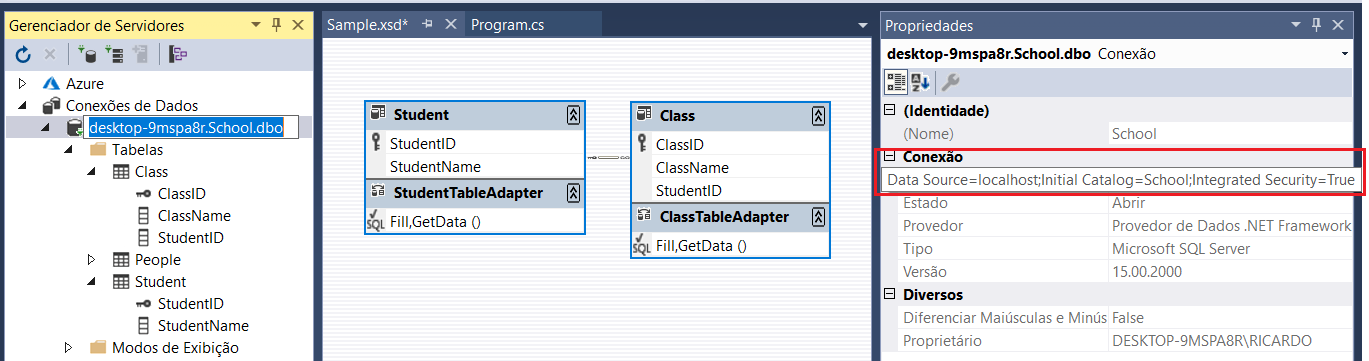
Depois disso, a janela Sample.xsd aparece na sua frente.



Etapa 2: adicione a conexão de dados acessando o Server Explorer. No Server Explorer, clique com o botão direito em Data Connections >> Add Connection; um assistente será aberto onde você precisará especificar o nome do servidor e selecionar um banco de dados criado no servidor SQL. Neste exemplo, foi adicionado o banco de dados School, com duas tabelas que irão ser usadas: Class e Student.



Etapa 3: arraste as tabelas da conexão adicionada (no Server Explorer) que for utilizar em Sample.xsd (a janela é aberta à sua frente) e salve o Sample.xsd. Para obter uma cadeia de conexão, clique com o botão direito do mouse em seu Connection (do server explorer) e clique em propriedades. Procure a propriedade String de conexão, onde você encontra sua string de conexão. Basta copiar e colar no seu código.



Passo 4: À medida que os componentes são adicionados, agora você pode interagir com seu banco de dados.

var connectionString = @"Data Source=localhost;Initial Catalog=School;Integrated Security=True";

Sample tds = new Sample();

using (SqlConnection connection = new SqlConnection(connectionString))

{

SqlDataAdapter da = new SqlDataAdapter("SELECT \* FROM Student", connection);

da.Fill(tds, tds.Student.TableName);

} // Connection is automatically closed.

foreach (DataRow item in tds.Student.Rows)

{

Console.WriteLine(string.Format("ID: {0} - Name: {1}", item[0], item[1]));

}

Use o método ReadXmlSchema para criar o esquema de um DataTable. O esquema inclui tabela, relação e definições de restrição. Para gravar um esquema em um documento XML, use o método WriteXmlSchema.

string projectDirectory = Directory.GetParent(workingDirectory).Parent.FullName;

var path = projectDirectory + "\\Sample.xsd";

DataSet dataSet = new DataSet();

dataSet.ReadXmlSchema(path);

foreach (DataTable table in dataSet.Tables)

{

Console.WriteLine(table.TableName);

foreach (DataColumn column in table.Columns)

{

Console.WriteLine("\t{0}: {1}", column.ColumnName,

column.DataType.Name);

}

}

É possível passar como parâmetro para o método ReadXmlSchema um StreamReader:

using (var xmlStream = new StreamReader(path))

{

dataSet.ReadXmlSchema(xmlStream);

foreach (DataTable table in dataSet.Tables)

{

// .....

}

};

Class

ClassID: Int32

ClassName: String

StudentID: Int32

Student

StudentID: Int32

StudentName: String

DataAdapter

DataAdapter é o objeto importante quando você trabalha com uma camada desconectada. Ele atua como uma ponte entre os dados na memória e um banco de dados. O DataAdapter preenche um DataTable ou DataSets e reconecta os dados na memória a um banco de dados. Você pode executar consultas de inserção, atualização, exclusão ou leitura enquanto os dados estiverem na memória e reconectar-se a um banco de dados para confirmar as alterações. O código abaixo mostra como executar operações orientadas a banco de dados usando uma camada desconectada:

string connectionString = ConfigurationManager.ConnectionStrings["PeopleConnection"].ConnectionString;

DataTable tbl = new DataTable();

using (SqlConnection connection = new SqlConnection(connectionString))

{

string command = "select \* from Student";

SqlDataAdapter ad = new SqlDataAdapter(command, connection);

ad.Fill(tbl);//Now the data in DataTable (memory)

}

foreach (DataRow item in tbl.Rows)

{

Console.WriteLine(string.Format("ID: {0} - Name: {1}", item[0], item[1]));

}

Quando o método de preenchimento do DataAdapter é chamado, uma consulta é executada e o método Fill() preenche o DataTable (obtém os dados e o mapa no DataTable). O DataTable não precisa manter aberta a conexão para preencher os dados, o que não é o caso do DataReader (em uma camada conectada). Essa é a beleza de uma camada desconectada e tem melhor desempenho do que uma camada conectada, pois lida com os dados presentes na memória, que são rapidamente acessíveis. Você também pode usar DataSet em vez de DataTable ao esperar vários conjuntos de resultados. O trabalho é o mesmo, exceto que ele pode retornar várias tabelas. DataSet possui a propriedade Table, pela qual você pode iterar sobre dados específicos da tabela.

Conforme indicado, você pode executar outras operações no DataTable ou no DataSet, como inserir, excluir etc. (Dados na memória), e essas operações têm desempenho rápido em comparação às operações executadas em uma camada Conectada. Como em uma camada conectada, o banco de dados é conectado e a arquitetura do ADO.NET leva a consulta e o mapa para um banco de dados (que consome tempo comparado ao mapa) ou executa alguma função nos dados presentes na memória, mas não em algum local externo (o caso de uma camada desconectada). O exemplo a seguir mostra a inserção de dados em uma DataTable e confirma as alterações em um banco de dados:

string connectionString = ConfigurationManager.ConnectionStrings["PeopleConnection"].ConnectionString;

DataTable tbl = new DataTable();

//New Record to add in DataTable

DataRow newRow = tbl.NewRow();

newRow["StudentName"] = "Mary Moon";

tbl.Rows.Add(newRow);

DataRow newRow2 = tbl.NewRow();

newRow2["StudentName"] = "Bob Cuspe";

tbl.Rows.Add(newRow2);

using (SqlConnection connection = new SqlConnection(connectionString))

{

//Now newRow has to add in Database(Pass newRow Parameters to this insert query)

string newCommand = @"Insert into Student(StudentName) Values(@StudentName)";

SqlDataAdapter ad = new SqlDataAdapter(newCommand, connection);

SqlCommand insertCommand = new SqlCommand(newCommand, connection);

//Create the parameters

insertCommand.Parameters.Add(new SqlParameter("@StudentName",SqlDbType.VarChar, 50, "StudentName"));

//Associate Insert Command to DataAdapter so that it could add into Database

ad.InsertCommand = insertCommand;

ad.Update(tbl);

}

Neste exemplo, é usada uma propriedade Parameter de um objeto Command, que recebe novos dados relacionados a parâmetros, como nome da coluna, tamanho da coluna e nome do parâmetro. "NewRow" adicionado no DataTable (novo registro na memória), mas não no banco de dados, mas mais tarde usou o método Update() do DataAdpater, que se reconecta a um banco de dados para fazer alterações (ou seja, DataTable atualizado mapeado para um banco de dados).

Este código abaixo demonstrará como usar a propriedade UpdateCommand de um objeto DbDataAdapter para atualizar um registro e também como usar a propriedade DeleteCommand de um DbDataAdapter para excluir um registro.

using (SqlConnection connection = new SqlConnection(connectionString))

{

SqlDataAdapter da = new SqlDataAdapter("SELECT \* FROM Student", connection);

//Create the update command

SqlCommand update = new SqlCommand();

update.Connection = connection;

update.CommandType = CommandType.Text;

update.CommandText = "UPDATE Student SET StudentName = @StudentName WHERE StudentID = @StudentID";

//Create the parameters

update.Parameters.Add(new SqlParameter("@StudentName", SqlDbType.VarChar, 50, "StudentName"));

update.Parameters.Add(new SqlParameter("@StudentID", SqlDbType.Int, 0, "StudentID"));

//Create the delete command

SqlCommand delete = new SqlCommand();

delete.Connection = connection;

delete.CommandType = CommandType.Text;

delete.CommandText = "DELETE FROM Student WHERE StudentID = @StudentID";

//Create the parameters

SqlParameter deleteParameter = new SqlParameter("@StudentID", SqlDbType.Int, 0, "StudentID");

deleteParameter.SourceVersion = DataRowVersion.Original; delete.Parameters.Add(deleteParameter);

//Associate the update and delete commands with the DataAdapter.

da.UpdateCommand = update;

da.DeleteCommand = delete;

//Get the data.

DataSet ds = new DataSet();

da.Fill(ds, "Student");

//Update the first row

ds.Tables[0].Rows[0]["StudentName"] = "Ricky Martin";

//Delete the second row.

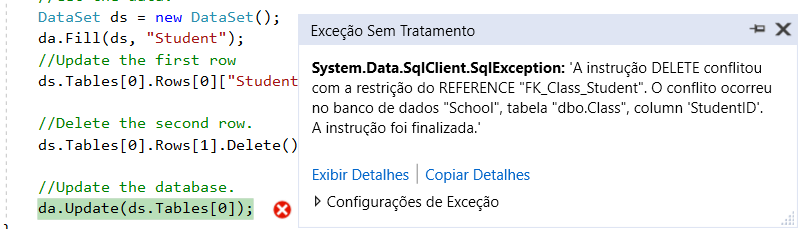
ds.Tables[0].Rows[5].Delete();

//Update the database.

da.Update(ds.Tables[0]);

}

Observe que as propriedades UpdateCommand e DeleteCommand estão definidas para diferentes objetos de comando que contêm a lógica para atualizar e excluir registros. A lógica DbDataAdapter executará automaticamente o objeto de comando de acordo com a função que ele deve executar. Certifique-se de colocar a lógica correta em seus objetos de comando, porque o DbDataAdapter apenas executará o comando; ele não verifica se realmente faz o trabalho correto.



System.Data.SqlClient.SqlException: 'A instrução DELETE conflitou com a restrição do REFERENCE "FK\_Class\_Student". O conflito ocorreu no banco de dados "School", tabela "dbo.Class", column 'StudentID'. A instrução foi finalizada.'

ENTITY FRAMEWORK

Usando um ORM (Object Relational Mapper)

As camadas conectadas e desconectadas forçam você a tratar dados da maneira de um esquema físico de um banco de dados. Essas camadas são fortemente acopladas ao banco de dados relacional, pois o usuário precisa usar o SQL para executar consultas e ter em mente a conexão, o comando, o DataReader, o DataSet e o DataAdapter, etc.

Um dos problemas com um banco de dados relacional é que ele é inerentemente diferente da estrutura orientada a objetos que você deseja usar em seus aplicativos. Você usa objetos que têm relação com outros objetos ou que herdam deles. O banco de dados usa tabelas com colunas e chaves estrangeiras para outras tabelas. Isso é chamado de **incompatibilidade de impedância objeto-relacional.**

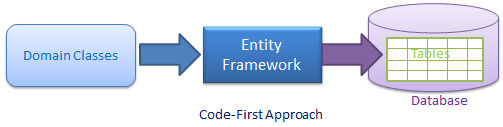
Você pode criar aplicativos complexos escrevendo manualmente suas consultas SQL e executando-as no banco de dados. Mas quando seu aplicativo cresce e você deseja usar um design orientado a objetos, começa a ter problemas. Obviamente, você pode mapear os resultados da sua consulta para objetos e criar consultas a partir das alterações nos seus objetos, mas essa é uma tarefa difícil.É aqui que entra o software Object Relational Mapping (ORM). Um desses ORM é o Entity Framework de código aberto da Microsoft, que diminui o fardo do trabalho tedioso de mapear seus objetos, construir consultas e acompanhar todas as alterações.

O uso de um ORM é um compromisso entre desempenho e velocidade de desenvolvimento. Diferentemente dessas camadas, o Entity Framework fornece a maneira orientada a objetos para interagir com um banco de dados. Usando essa camada conceitual do ADO.NET, você não precisa se preocupar com objetos de conexão ou de comando. Esse tipo de coisa é tratado automaticamente pelo Entity Framework. Você pode imaginar como o Entity Framework usa a reflexão para mapear dos resultados da consulta SQL para os objetos, o que é sempre mais lento do que fazê-lo manualmente. No entanto, o uso do Entity Framework aumentará imensamente sua velocidade de desenvolvimento. Ele permite que você crie seu aplicativo rapidamente, priorizando as partes que precisam de melhor desempenho. O Entity Framework possui uma visualização gráfica na qual é possível arrastar e soltar objetos do banco de dados e atualizar essa visualização sempre que houver uma alteração no base de dados. Ele geralmente atende a maioria das situações, caso contrário, você pode adicionar uma Store Procedure simples para as consultas problemáticas.

O LINQ é usado em vez do SQL e você pode usar um dos tipos de LINQ com sua fonte de dados fornecida pelo ADO.NET (banco de dados). Quando você executa consultas LInQ na fonte de dados ADO.Net, o tempo de execução da estrutura da entidade gera uma instrução SQL adequada em seu nome. Mas isso diminui o desempenho em comparação com a camada conectada e a desconectada.

O Entity Framework (EF) normalmente tem quatro abordagens para usar, a primeira é mais utilizada que é quando há um banco de dados existente, as outras são variações de abordagens para quando não houver um banco de dados definido e deseja iniciar o projeto e depois criar a base:

* **Database First (EF Designer from database):**Você deseja mapear um banco de dados existente para sua estrutura de objeto, com interface visual de mapeamento edmx.
* **Model First (Empty EF Designer model):** permite modelar nossa base de dados de forma visual através dos recursos do próprio Visual Studio, através da criação de um arquivo .edmx. A vantagens é de não haver necessidade de conhecimentos prévios do banco de dados ou do Entity Framework. Por outro lado, causa problemas em relação a manutenções e alterações nas regras de negócio da aplicação.
* **Empty Code First model:**O Entity Framework introduziu a abordagem Code-First com o Entity Framework 4.1. O Code-First é principalmente útil no Design Orientado a Domínios (Domain Driven Design). Na abordagem Code-First, você se concentra no domínio do seu aplicativo e começa a criar classes para a entidade do domínio, em vez de criar o banco de dados primeiro e depois criar as classes que correspondem ao design do banco de dados. A figura a seguir ilustra a abordagem do primeiro código.



Como você pode ver na figura acima, a EF API criará o banco de dados com base nas suas classes e configuração de domínio. Isso significa que você precisa começar a codificar primeiro em C# ou VB.NET e, em seguida, a EF criará o banco de dados a partir do seu código.

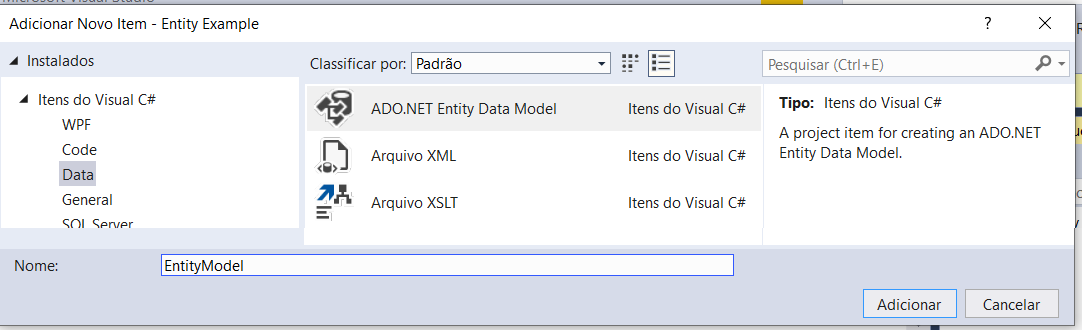
* **Code First(Code first from database):**Mesmo você tendo o banco pronto, pode adotar a abordagem Code First que vai escrever as classes para mapear o banco existente e a partir daí começa a trabalhar a partir das classes, sem interface visual de mapeamento edmx.

Essas abordagens podem ser usadas usando o Assistente de modelo de dados de entidade, mas a abordagem “Code first from database” também pode ser usada sem o Assistente de modelo de dados da entidade. Toda abordagem tem sua própria adequação. Temos duas principais frentes no que diz respeito à ordem e gestão dos processos durante o desenvolvimento de um software:

* **Modelo waterfall ou em cascata:** é a mais utilizada, onde o software passa por diversas etapas até sua entrega, e em teoria, o sistema possui todos os seus requisitos levantados no início e o projeto de software possui uma arquitetura, design, vários diagramas e estrutura da base de dados feita logo no início.
* **Modelo ágil:** onde o desenvolvimento de software é incremental, as necessidades são separadas em pequenas partes e o sistema não é imutável ocorrendo entregas periódicas com novas partes do sistema. Como deve imaginar, em um modelo ágil, como há releases incrementais, a base de dados e os dados necessários podem mudar muitas vezes antes de chegar a seu estado final. Nesses momentos a utilização do Entity Framework Migrations é fundamental, fornecendo uma forma prática para lidar com essas mudanças.

Database First (EF Designer from database)

Etapa 1: clique com o botão direito do mouse no seu projeto e clique em Adicionar>> Novo item. Selecione “ADO.NET Entity Data Model”, nomeie o que quiser (neste exemplo, eu o nomeei Sample) e clique no botão “Add”.



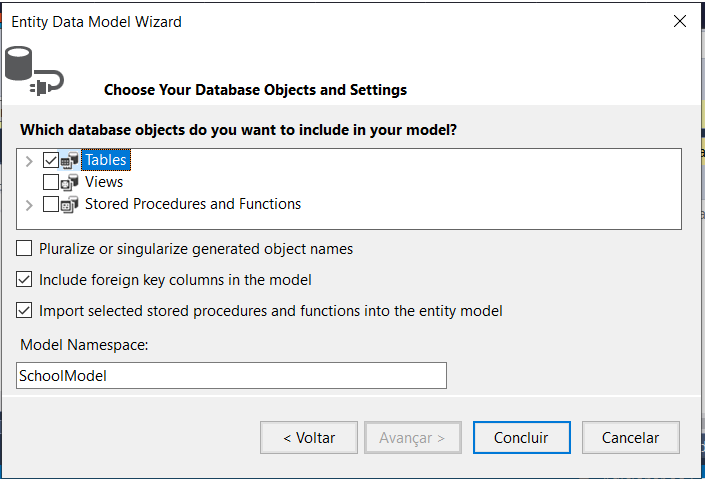
Passo 2: Depois de clicar no botão "Adicionar", o próximo passo é escolher o conteúdo do modelo. Escolha o primeiro, EF Designer do banco de dados, e clique no botão "Avançar".



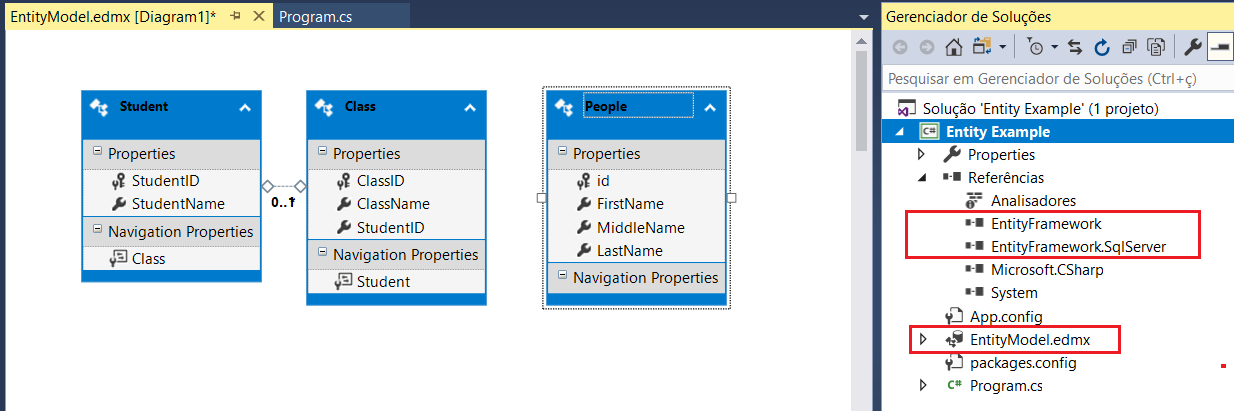
Etapa 3: Depois de clicar no botão "Avançar", o próximo passo é escolher Conexão de dados. Escolha na caixa suspensa (a Conexão de dados será escolhida e uma string de conexão será criada. Observe o nome abaixo no TextBox que seria o objeto do seu banco de dados. Você pode alterar esse nome. Eu o chamei de SchoolDB) e clique no botão " Avançar”, (vá para a Etapa 5); caso contrário, clique no botão “Nova conexão” na janela à sua frente (vá para a Etapa 4).

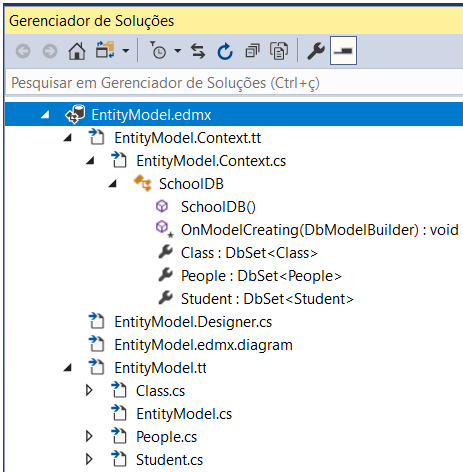
Passo 4: Se você clicou no botão "Nova conexão", especifique o nome do servidor e o banco de dados e clique no botão "OK". O painel "Conectar-se a um banco de dados" será ativado quando você especificar o nome do servidor. Você pode usar a autenticação do Windows ou uma autenticação do SQL Server para se conectar ao banco de dados. Se você selecionar Autenticação do SQL Server, precisará inserir um Logon e uma Senha válidos do SQL. A Conexão de Dados será selecionada e uma cadeia de conexão será estabelecida. Clique no botão "Next".

Etapa 5: Depois de clicar no botão "Avançar", escolha seus objetos e configurações de banco de dados. Existem caixas de seleção de "Tabelas", "View" e "Stored Procedures and Functions". Marque as caixas de seleção nos Objetos de Banco de Dados que você deseja adicionar.



Eu seleciono as "Tabelas" neste exemplo. Após a seleção, clique no botão "Concluir". O assistente está concluído. Referências necessárias e um arquivo chamado "Sample.edmx" serão adicionados aos seus projetos.



Clique na seta ao lado do arquivo NorthwindsModel.edmx no Solution Explore para exibir todos os arquivos criados para você. Observe o arquivo chamado EntityModel.tt. Este é um arquivo do Text Transformation Template Toolkit, também conhecido como modelo T4. Um arquivo de modelo T4 é usado para gerar automaticamente o código no Visual Studio. Os modelos T4 são uma mistura de blocos de texto e instruções de controle que permitem gerar um arquivo de código. Clique na seta ao lado do arquivo NorthwindsModel.tt para expandir a lista de arquivos gerados por este modelo. Um arquivo foi criado para cada tabela, exibição e procedimento armazenado que retorna um conjunto de resultados.

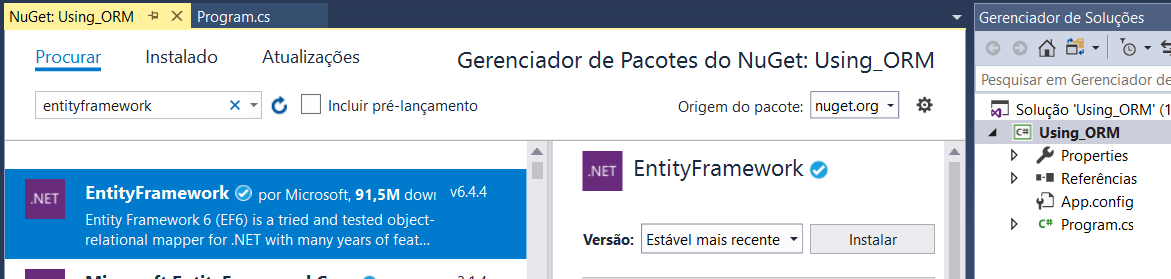
Agora clique no arquivo EntityModel.Context.tt no Solution Explorer. Este é o modelo T4 para o objeto de contexto. Pense no objeto Context como a classe que representa todo o banco de dados. Se você clicar na seta ao lado do arquivo EntityModel.Context.tt, poderá ver um arquivo, EntityModel.Context.cs. Esta é a classe que foi criada pelo modelo T4. Abra o arquivo EntityModel.Context.cs clicando nele. O nome da classe é SchoolDB e possui propriedades para cada tabela contida no banco de dados. As propriedades são tipos DbSet genéricos, que são uma coleção de cada tipo que representa uma tabela ou exibição. Algumas das propriedades que foram criadas estão listadas aqui:

publicvirtual DbSet<Class> Class { get; set; }

publicvirtual DbSet<People> People { get; set; }

publicvirtual DbSet<Student> Student { get; set; }

Agora você está pronto para interagir com seu banco de dados. Caso as referências as dll do EntityFramework não forem incluídas ao projeto, ou for adicionar o Entity Framework a qualquer outro projeto é possível utilizar o Gerenciador de Pacotes Nuget para essa tarefa:



O exemplo a seguir mostra como você pode obter todos os dados dos alunos usando o Entity Framework:

using (SchoolDB db = new SchoolDB())

{

var students = (from p in db.Student select p).ToList();

foreach (var student in students)

{

Console.WriteLine("ID is: " + student.StudentID);

Console.WriteLine("Name is: " + student.StudentName);

}

}

Como você pode ver, foram necessárias apenas algumas linhas de código para obter os registros e você não precisou escrever nenhuma consulta SQL. Tudo é tratado por você pelo Entity Framework. SchoolDB é um nome que representa seu banco de dados. Ele contém todas as coisas dentro do seu base de dados. Você pode usar seu objeto para acessar ou usar itens de banco de dados, como acessar tabelas, etc. Uma consulta LINQ é usada para recuperar os dados do banco de dados e recuperar os resultados. Observe que você escreve a consulta usando a sintaxe C#. As classes do Entity Framework sabem como converter isso em uma consulta SQL para você nos bastidores. Quando os dados são recuperados, você pode enumerar as categorias para gravar cada registro na janela Saída.

Anteriormente, foi apontado que a tabela Class tinha uma chave estrangeira para a tabela Student. A classe Class que é gerada pelo Assistente de Modelo de Dados da Entidade criou uma propriedade para representar esse relacionamento.

publicvirtual Student Student { get; set; }

O exemplo de código a seguir mostra como gravar uma consulta LINQ para unir as duas tabelas e gravar o nome do estudante e o nome da classe na janela Saída:

using (SchoolDB db = new SchoolDB())

{

var classes = from c in db.Student

join p in db.Class

on c.StudentID equals p.StudentID

select p;

foreach (var clas in classes)

{

Console.WriteLine(string.Format("StudentName: {0}, ClassName: {1}",

clas.Student.StudentName, clas.ClassName));

}

}

Neste exemplo, em vez de selecionar o objeto Student, ele seleciona o objeto Class. O Entity Framework recupera as colunas corretas e preenche as propriedades da classe Studente, como você pode ver, também preenche a propriedade Class.O código abaixo mostra como você pode adicionar um registro (novo aluno) a uma tabela de alunos usando EF:

using (SchoolDB db = new SchoolDB())

{

Student st = new Student();

st.StudentName = "Mubashar Rafique";

db.Student.Add(st);

db.SaveChanges();

}

É tão simples quanto adicionar um item em uma lista. "Db" é o objeto do banco de dados e para adicionar um novo objeto de aluno, chame o método Add () no db.Students; isso adicionará o aluno aos Student e, após o método SaveChanges, um novo aluno será adicionado ao seu banco de dados. Você pode executar outras operações, como atualizar, excluir e localizar usando o LINQ.

using (SchoolDB db = new SchoolDB())

{

//Find specific Studnet

var std = (from p in db.Student

where p.StudentName == "Mubashar Rafique"

select p).FirstOrDefault();

if (std != null)//if student is found

{

std.StudentName = "Updated Name";

db.SaveChanges();

}

if (std != null)//if student is found

{

db.Student.Remove(std);

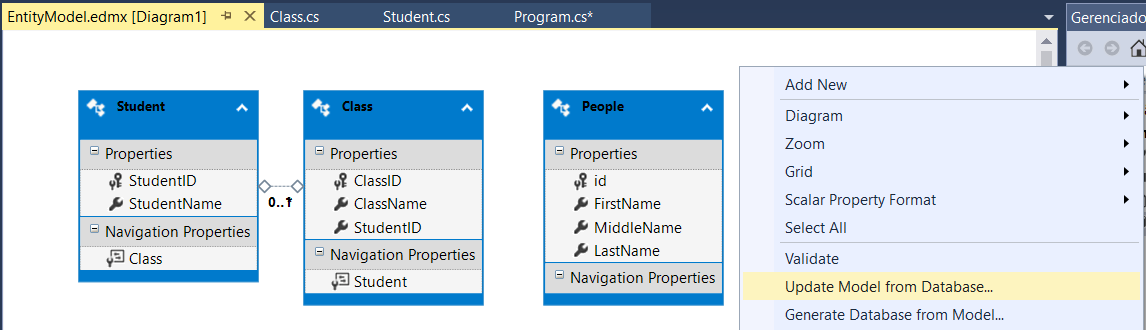
db.SaveChanges();

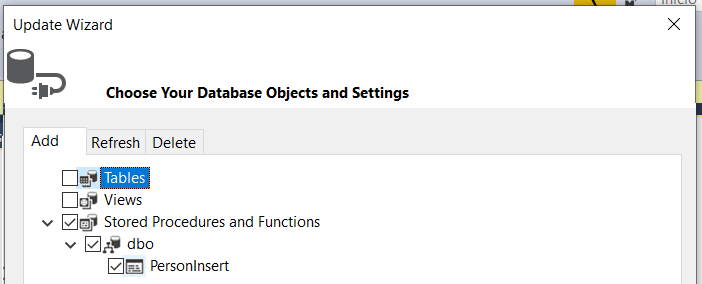
}

}

Chamar uma Stored Procedure

Como não selecionamos para incluir as Stored Procedure ao criar o modelo do Entity Framework, iremos fazer isso agora. Para isso, basta clicar com o botão direito do mouse na interface visual do arquivo .edmx e selecionar “Update Model from Database” conforme mostrado na figura abaixo.





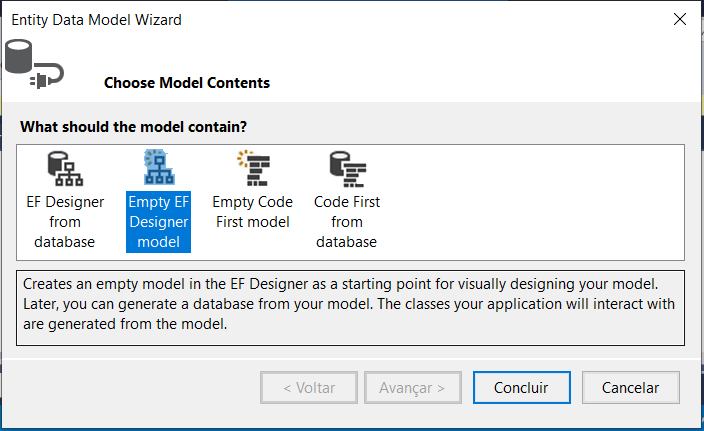
Para chamar um procedimento armazenado, basta chamar o método O exemplo de código a seguir chama o procedimento armazenado PersonInsert:

var retornosp = db.PersonInsert("Alceu", "Valenca");

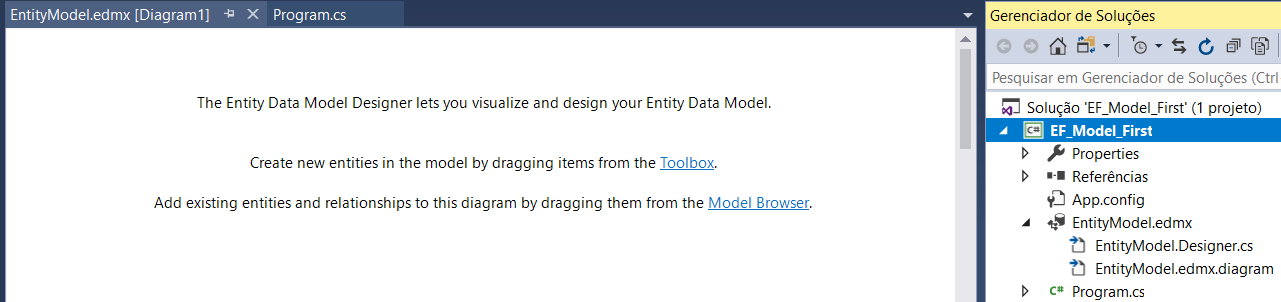
Console.WriteLine(retornosp); //-1

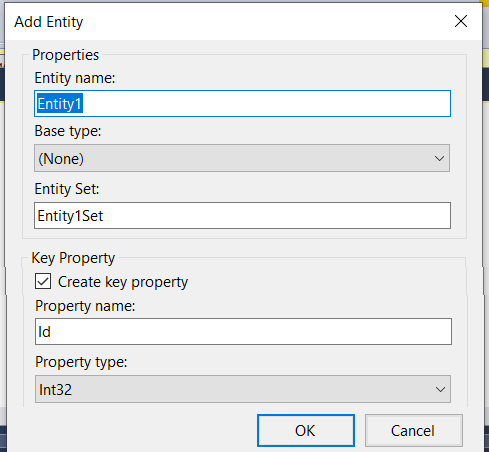
Model First (Empty EF Designer model):

Vamos construir agora nosso próprio modelo usando um novo designer de Context e EntityFramework, usando uma metodologia chamada Model-First, o que significa apenas que iremos construir o modelo do Entity Framework e depois usá-lo para criar o banco de dados.Nosso modelo representará um catálogo de cursos universitários muito simples. Criaremos entidades para alunos e cursos e mostraremos as relações entre eles.



O Visual Studio adicionará um novo arquivo EDMX ao local escolhido e exibirá uma tela de designer vazia mostrando o seguinte texto:

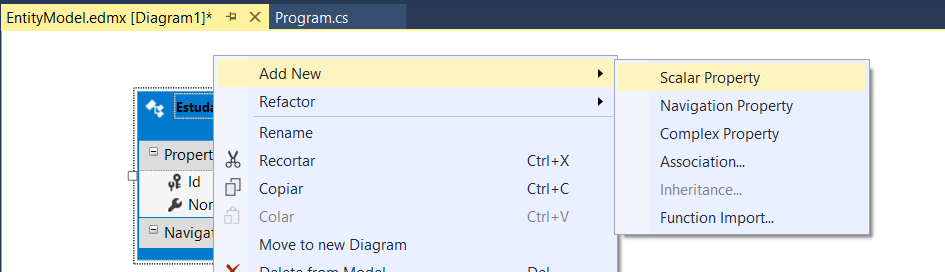


Vamos começar a configurar o nosso modelo! Primeiro, vamos criar uma entidade para representar um aluno. Clique com o botão direito do mouse no designer e selecione Adicionar novo -> Entidade:

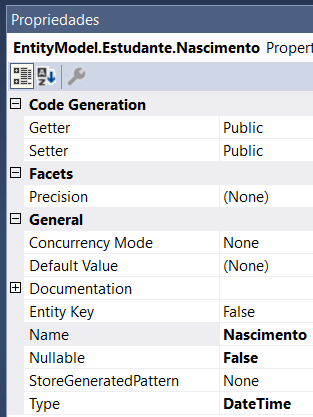
* **Entity Name:** exatamente o que parece. Estamos chamando o nosso de "Aluno". De um modo geral, essa será uma forma singular.
* **Base Type:** se a nova entidade herdar de outra entidade, você poderá usar este menu suspenso para selecionar a entidade de base.
* **Entity Set:** o nome da coleção desse tipo de entidade que o Contexto produzirá. Geralmente, essa será a forma plural do Nome da entidade.
* **Key Property**: os detalhes sobre uma coluna de chave que você deseja criar.

Nesse caso, estamos criando uma entidade Estudante com ID da chave. A entidade resultante é assim:

Agora temos um estudante, com um único ID de propriedade. Vamos adicionar mais algumas propriedades à entidade Student. Clique com o botão direito do mouse na entidade e selecione Adicionar novo -> Propriedade escalar



Propriedades escalares são propriedades da entidade que contêm um valor. No nosso caso, vamos adicionar uma propriedade para Name. Confira a janela Propriedades ao selecionar Nome para ver os detalhes da propriedade:



Algumas propriedades que você deve conhecer:

**Nullable**: se esta propriedade pode ser nula.

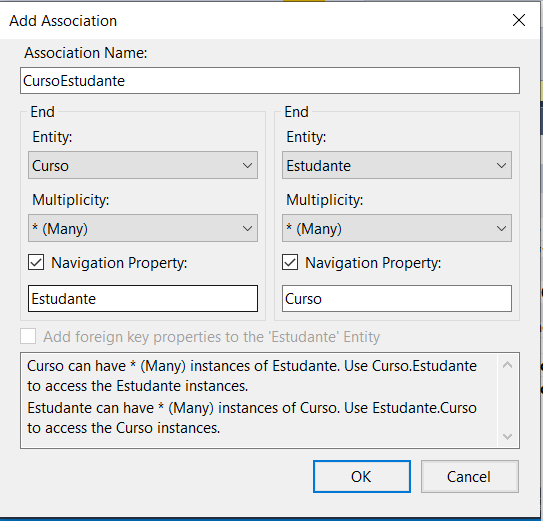
**Type:** O tipo .NET desta propriedade.

**Default Value**: se uma instância desta entidade for criada, o valor padrão usado para esta propriedade.

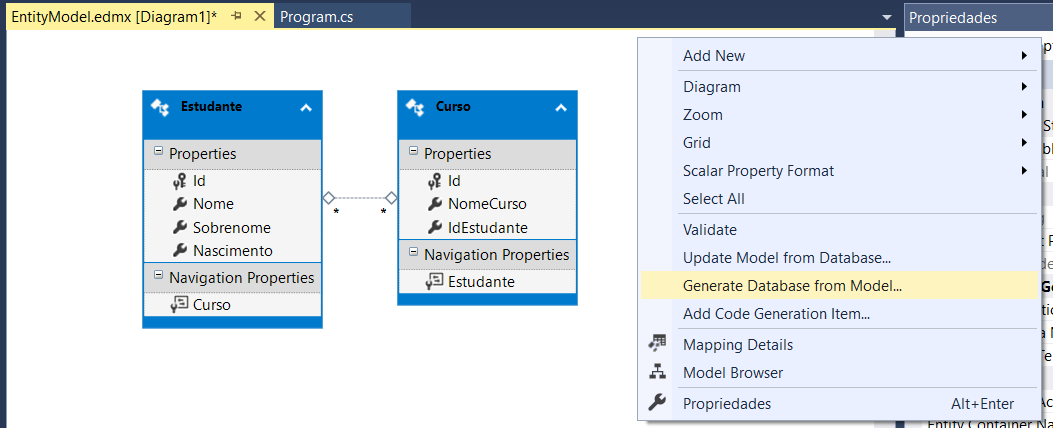
**Max Length**: o comprimento máximo dos dados. No caso de seqüências de caracteres, corresponde ao nvarchar (X) no SQL Server, onde X é o comprimento máximo.

No nosso caso, as propriedades já são o que queremos. Vamos adicionar mais duas propriedades, uma string LastName e um datetime DateOfBirth. Para DateOfBirth, lembre-se de alterar o Tipo na janela Propriedades.Agora vamos criar uma nova entidade para o Curso.

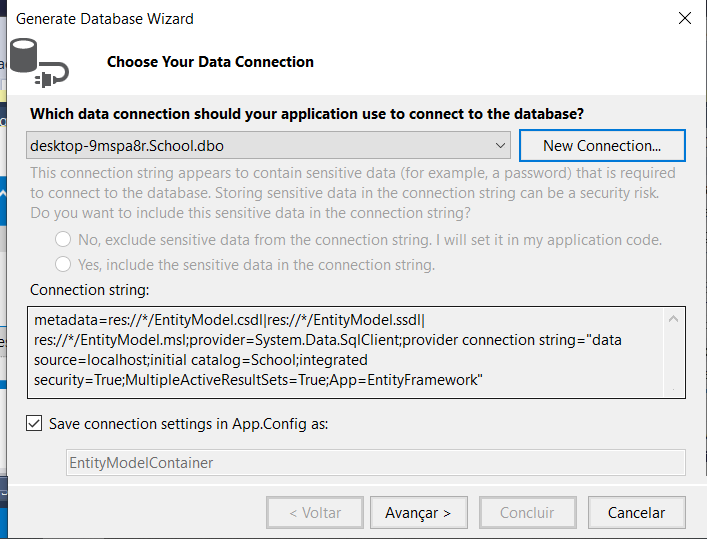
Agora que definimos todas as nossas entidades, podemos começar a criar os relacionamentos entre Estudante e Curso. Construímos esse relacionamento clicando com o botão direito do mouse em Estudante e selecionando Adicionar novo -> Associação. Isso abre o diálogo Associação. Um único aluno pode fazer muitos cursos, e um único curso (espero) terá muitos alunos. Portanto, esse é um relacionamento de muitos para muitos. Vamos criar esse relacionamento no designer:



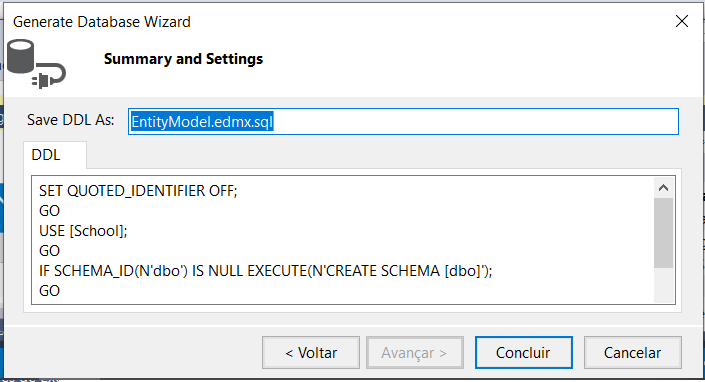
Agora que o modelo foi totalmente elaborado, podemos mostrar como o Entity Framework nos permite usar um modelo para criar um banco de dados.Primeiro, clique com o botão direito do mouse na tela do modelo e selecione Gerar banco de dados do modelo:



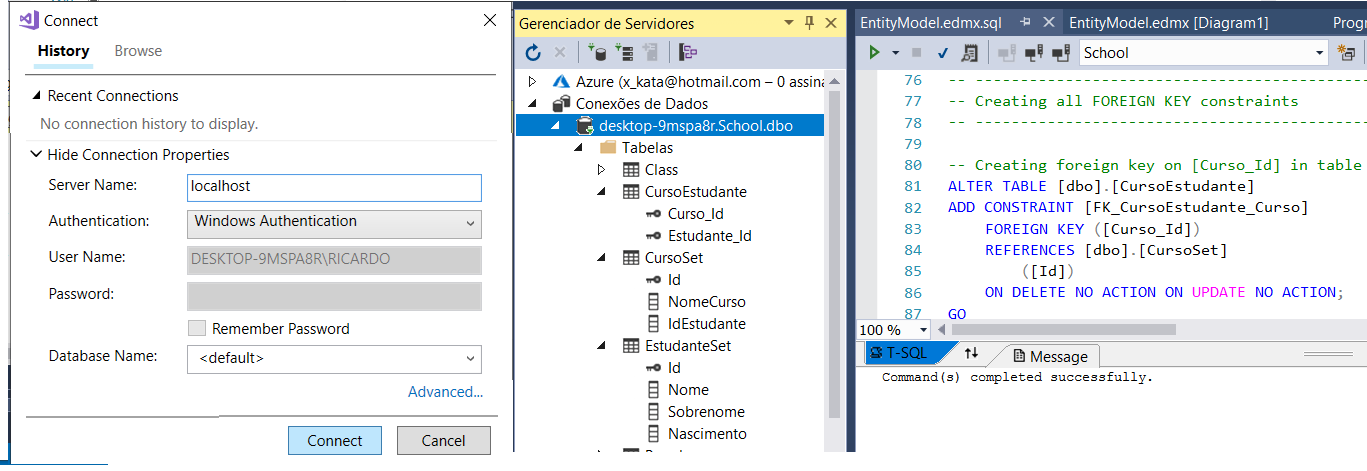
Em seguida, escolha a conexão que você deseja usar (estou usando um banco de dados local):



Clique em Avançar e você verá o script SQL necessário para criar as tabelas, colunas e relacionamentos para este banco de dados:

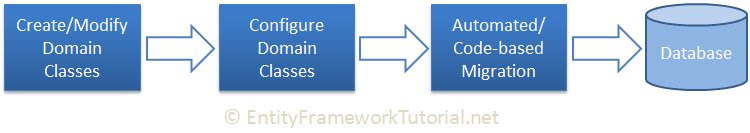


Clique em Concluir para criar o script necessário para criar o banco de dados. Para executar o script ainda é necessário fazer mais uma validação de conexão. Depois de executar o scriptseu banco de dados estará pronto. Aqui está meu banco de dados completo:



Empty Code First model

A figura a seguir ilustra o fluxo de trabalho de desenvolvimento com o Code-First.



O fluxo de trabalho de desenvolvimento na abordagem de primeiro código seria: Criar ou modificar classes de domínio -> configurar essas classes de domínio usando a API do Fluent ou atributos de anotação de dados -> Criar ou atualizar o esquema do banco de dados usando migração automatizada ou migração baseada em código.

O Entity Framework 4.3 incluiu um novo recurso Code First Migrations que permite evoluir de forma incremental o esquema do banco de dados conforme seu modelo muda com o tempo. Para a maioria dos desenvolvedores, essa é uma grande melhoria em relação às opções do inicializador de banco de dados das versões 4.1 e 4.2 que exigiam que você atualize manualmente o banco de dados ou descarte e recriá-lo quando o modelo foi alterado.

* Antes do Entity Framework 4.3, se você já possui dados (exceto dados de propagação) ou procedimentos armazenados, gatilhos etc. no seu banco de dados, essas estratégias eram usadas para descartar o banco de dados inteiro e recriá-lo, para que você perdesse os dados e outros bancos de dados objetos.
* Com a migração, ele atualizará automaticamente o esquema do banco de dados quando o modelo for alterado sem perder dados ou outros objetos de banco de dados existentes.
* Ele usa um novo inicializador de banco de dados chamado MigrateDatabaseToLatestVersion.

Existem dois tipos de migração:

1. **Migração automatizada:** modo totalmente automático, onde você não precisa se preocupar em atualizar o banco, pois ao executar o seu programa, isto será feito de maneira automatica.
2. **Migração baseada em código:** usa pontos de migração no banco de dados, onde podemos avançar, aplicando as últimas modificações, ou voltar na linha do tempo, em qualquer momento do banco de dados;

Migração automatizada

A migração automatizada foi introduzida pela primeira vez na estrutura 4.3 da entidade. Na migração automatizada, você não precisa processar a migração do banco de dados manualmente no arquivo de código. Por exemplo, para cada alteração, você também precisará alterar nas classes de domínio. Mas com a migração automatizada, basta executar um comando no Package Manager Console para fazer isso. Neste exemplo, iniciaremos nossas 3 aulas básicas, como Aluno, Curso e Inscrição, conforme mostrado no código a seguir.

publicclassAluno

{

publicint ID { get; set; }

publicstring LastName { get; set; }

publicstring FirstMidName { get; set; }

public DateTime EnrollmentDate { get; set; }

publicvirtual ICollection<Inscricao> Enrollments { get; set; }

}

publicclassCurso

{

publicint CourseID { get; set; }

publicstring Title { get; set; }

publicvirtual ICollection<Inscricao> Enrollments { get; set; }

}

publicclassInscricao

{

publicint EnrollmentID { get; set; }

publicint CourseID { get; set; }

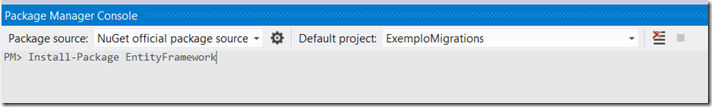
publicint StudentID { get; set; }

publicvirtual Curso Course { get; set; }

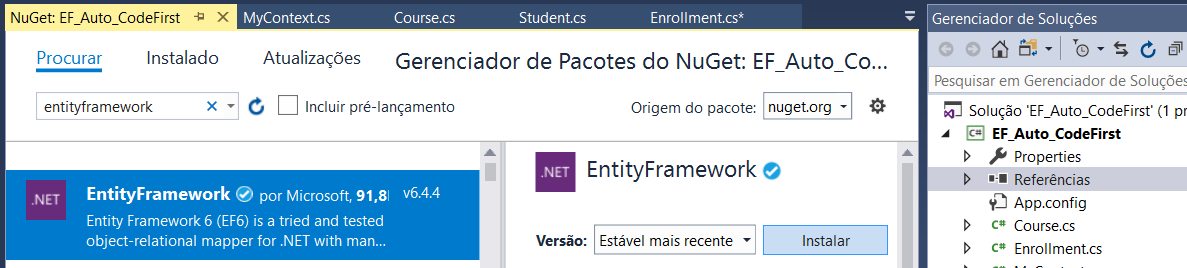
publicvirtual Aluno Student { get; set; }

}

Logo após criar as classes, vamos adicionar o Entity Framework CodeFirst usando o NuGet. Para isto abra o gerenciador do NuGet em Tools/Library Package Manager/Packager Manager Console e digite:



Ou instale pelo Gerenciador de Pacotes Nuget:



Após isto teremos o EF CodeFirst instalado em nosso projeto. Vamos agora criar um Contexto e uma classe para podermos trabalhar com o Migrations. A seguir está a classe de contexto.

publicclassMyContext : DbContext

{

public MyContext() : base("MyContextDB") { }

publicvirtual DbSet<Curso> Cursos { get; set; }

publicvirtual DbSet<Inscricao> Inscricoes { get; set; }

publicvirtual DbSet<Aluno> Alunos { get; set; }

}

O DbContext é um invólucro para ObjectContext e além disso esta classe contém:

* Um conjunto de APIs que são mais fáceis de usar do que a exposta pelo ObjectContext;
* As APIs que permitem utilizar o recurso do Code-First e as convenções;

Você pode definir no arquivo de configuração App.Config a string de conexão e indicar nome banco de dados, senão fizer isso, o padrão que o Entity Framework usa é o seguinte: A API DbContext por convenção cria o banco de dados para você em localhost\SQLEXPRESS (ou (localdb)\mssqllocaldb ou outras variações dependendo da instalação de seu serviço SQL) e o nome do banco de dados é derivado a partir do nome do projeto e do contexto, no nosso caso seria: EF\_Auto\_CodeFirst.MyContext. Porém, o nome pode ser customizado na classe contexto, como segue abaixo:

publicclassMyContext : DbContext

{

public MyContext() : base("SchoolDB")

Vamos tentar adicionando um arquivo app.config para identificarmos nosso servidor SQL e o nome do banco de dados que já temos chamado db\_code\_first. A tag configSections deve ser única e ser a primeira do arquivo de configurações, antes do connectionStrings:

<?xmlversion="1.0"encoding="utf-8"?>

<configuration>

<configSections>

<!-- For more information on Entity Framework configuration, visit http://go.microsoft.com/fwlink/?LinkID=237468 -->

<sectionname="entityFramework"

type="System.Data.Entity.Internal.ConfigFile.EntityFrameworkSection, EntityFramework, Version=6.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089"

requirePermission="false"/>

</configSections>

<startup>

<supportedRuntimeversion="v4.0"sku=".NETFramework,Version=v4.6.1"/>

</startup>

<connectionStrings>

<addname="MyContextDB"providerName="System.Data.SqlClient"connectionString="data source=localhost; initial catalog=db\_code\_first; integrated security=true;"/>

</connectionStrings>

<entityFramework>

<providers>

<providerinvariantName="System.Data.SqlClient"type="System.Data.Entity.SqlServer.SqlProviderServices, EntityFramework.SqlServer"/>

</providers>

</entityFramework>

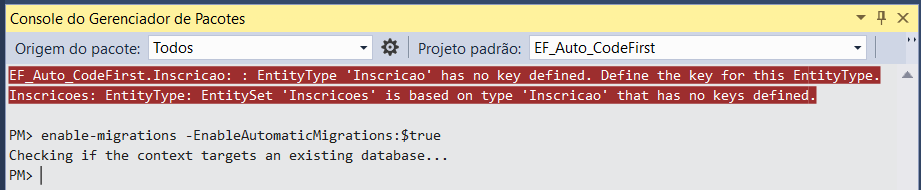
</configuration>

Vamos inicialmente adicionar o Migrations ao nosso projeto. Independente do método: manual ou automático, precisamos adicioná-lo ao nosso projeto. Faremos isto usando novamente a janela do Nuget através do comando Enable-Migrations:

Etapa 1 - Abra o Console do Gerenciador de Pacotes em Ferramentas → Gerenciador de Pacotes NuGet → Console do Gerenciador de Pacotes.

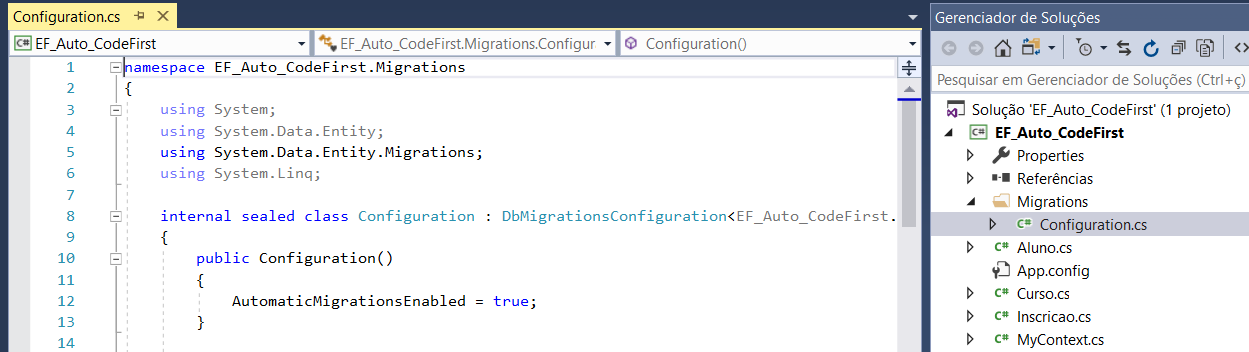
Etapa 2 - Para habilitar a migração automatizada, execute o seguinte comando no Package Manager Console.

**enable-migrations -EnableAutomaticMigrations:$true**



Repare que se os nomes das tabelas não forem iguais aos nomes dos campos chaves pode haver problemas de criação das chaves primárias.

Etapa 3 - Depois que o comando é executado com êxito, ele cria uma classe de Configuração selada interna derivada de DbMigrationConfigurationna pasta Migração do seu projeto, conforme mostrado no código a seguir.



Como você pode ver no construtor da classe Configuration, AutomaticMigrationsEnabled está definido true.

Etapa 4 - Defina o inicializador do banco de dados na classe de contexto com a nova estratégia de inicialização do banco de dados MigrateDatabaseToLatestVersion. Repare que o nome MyContextDB é o mesmo definido como o nome da configuração em App.config, conforme mostrado abaixo. Apesar de estar definindo aqui o nome do banco também como MyContextDB em base("MyContextDB"), esta configuração será ignorada uma vez que prevalence o que está definido no App.config que é initial catalog=db\_code\_first.

publicclassMyContext : DbContext

{

public MyContext() : base("MyContextDB")

{

Database.SetInitializer(new MigrateDatabaseToLatestVersion<MyContext,

EF\_Auto\_CodeFirst.Migrations.Configuration>("MyContextDB"));

}

publicvirtual DbSet<Curso> Cursos { get; set; }

publicvirtual DbSet<Inscricao> Inscricoes { get; set; }

publicvirtual DbSet<Aluno> Alunos { get; set; }

}

O que fizemos foi adicionar a chamada do DatabaseSetInitializer() com a opção MigrateDatabaseToLastVersion, o que faz com que nosso banco de dados seja sempre atualizado de acordo com as nossas classes.

Etapa 5 - Agora que temos o nosso modelo de entidades pronto e a migração automatizada configurada. Vamos definir o código no Modulo da solução para vermos o Code-First em ação, digite o código abaixo na aplicação console:

using (MyContext db = new MyContext())

{

Aluno st = new Aluno();

st.EnrollmentDate = DateTime.Now;

st.FirstMidName = "Mubashar Rafique";

st.LastName = "Silva";

db.Alunos.Add(st);

db.SaveChanges();

Console.WriteLine("Aluno Added!");

var alunos = db.Alunos;

foreach (var item in alunos)

{

Console.WriteLine(item.LastName);

}

}

Console.WriteLine(" Banco de dados e tabelas cridas com sucesso");

Quando você executa seu aplicativo, ele cuida automaticamente da migração quando você altera o modelo.A tabela foi definida com o nome Alunoes, Cursoes porque o Entity Framework pluraliza o nome da tabela usando a regra do idioma inglês, ou seja, acrescenta es ao final do nome.



Etapa 6 - Como você pode ver, uma tabela do sistema \_\_MigrationHistory também é criada no seu banco de dados com outras tabelas. No \_\_MigrationHistory, a migração automatizada mantém o histórico de alterações no banco de dados.

Etapa 7 - Quando você adiciona outra classe de entidade como sua classe de domínio e executa seu aplicativo, ele cria a tabela no seu banco de dados. Vamos adicionar a seguinte classe StudentLogIn.

publicclassAlunoLogin

{

[Key, ForeignKey("Aluno")]

publicint ID { get; set; }

publicstring EmailID { get; set; }

publicstring Password { get; set; }

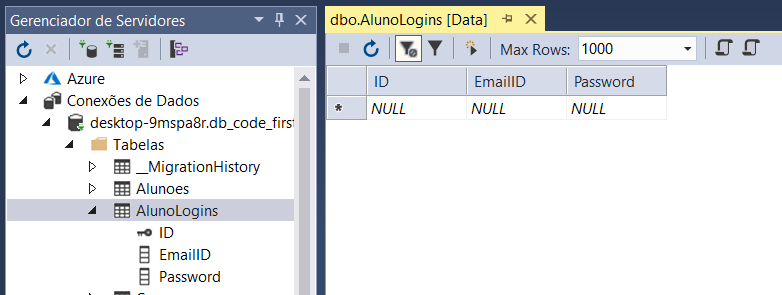
publicvirtual Aluno Aluno { get; set; }

}

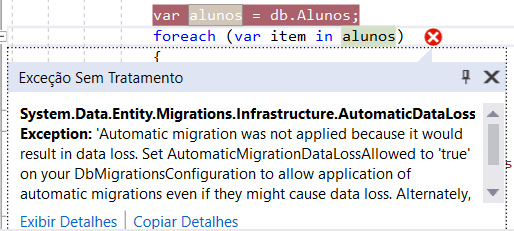
Etapa 8 - não se esqueça de adicionar o DBSet para a classe mencionada acima na sua classe de contexto, conforme mostrado no código a seguir.

publicvirtual DbSet<AlunoLogin> AlunoLogins { get; set; }

Etapa 9 - Execute seu aplicativo novamente e você verá que a tabela StudentsLogIn foi adicionada ao seu banco de dados.



As etapas acima mencionadas para migrações automatizadas funcionarão apenas para sua entidade. Por exemplo, para adicionar outra classe de entidade ou remover a classe existente, ela será migrada com êxito. Mas se você adicionar ou remover qualquer propriedade à sua classe de entidade, ela lançará uma exceção.



Etapa 10 - Para lidar com a migração de propriedades, você precisa definir AutomaticMigrationDataLossAllowed = true no construtor da classe de configuração.

internalsealedclassConfiguration : DbMigrationsConfiguration<EF\_Auto\_CodeFirst.MyContext>

{

public Configuration()

{

AutomaticMigrationsEnabled = true;

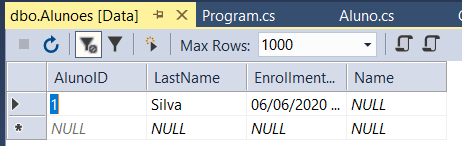
AutomaticMigrationDataLossAllowed = true;

//É possível definir o nome do context aqui

//ContextKey = "EFCodeFirstDemo.MyContext";

} //.....

Mas ao renomear a propriedade name de FirstMidName para Name na tabela Alunos, perdeu os valores gravados:



Migração Baseada em Código

A migração baseada em código é útil quando você deseja ter mais controle sobre a migração. Essa abordagem permite que o Code First atualize o esquema do banco de dados em vez de eliminar e recriar o banco de dados. Aqui está a regra básica para migrar alterações no banco de dados –

* **Enable Migrations:**Habilita a migração no seu projeto, criando uma classe de Configuração.
* **Add-Migrations “nome\_migrations”** – cria um alteração no banco de dados, onde o “nome\_migrations” é o nome que você irá dar para a atualização; Cria uma nova classe de migração conforme o nome especificado com os métodos Up() e Down().
* **Update Database:** aplica as alterações no banco de dados; Executa o último arquivo de migração criado pelo comando Add-Migration e aplica mudanças no esquema do banco de dados.
* **Update-DataBase – script** : gera um script com os comandos SQL para serem executados no banco de dados.

Neste exemplo, começaremos novamente com nossas 3 classes básicas, como Aluno, Curso e Inscrição.

A seguir está a classe de contexto. Vamos fazer algumas pequenas modificações do exemplo anterior para verificarmos os resultados. Primeiro, não utilizaremos o arquivo App.config para indicarmos a nossa conexão com o banco de dados. Segundo, os nomes atribuído aos DbSet abaixo estão no singular, anteriormente haviam sido colocados no plural e mesmo assim gerou as tabelas como Cursoes e Alunoes.

publicclassMyContext : DbContext

{

public MyContext() : base("MyContextDB")

{

//Database.SetInitializer(new MigrateDatabaseToLatestVersion<

// MyContext, EF\_CodeBasedFirst.Migrations.Configuration>("MyContextDB"));

}

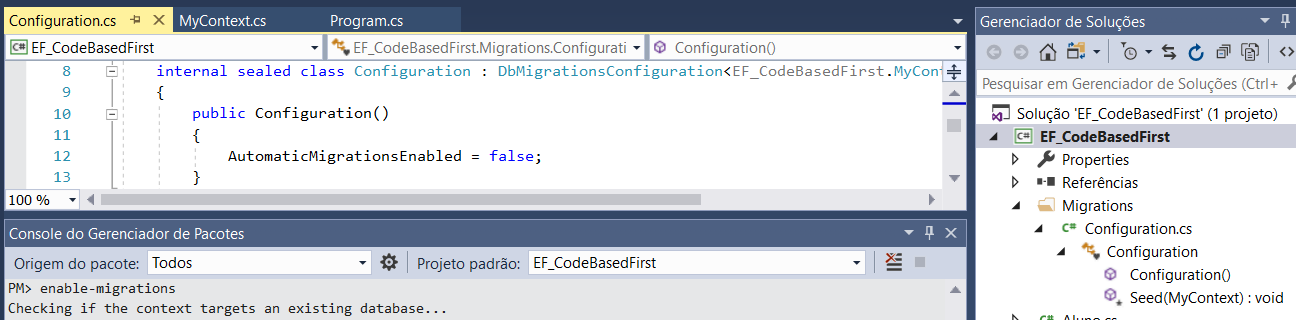
publicvirtual DbSet<Curso> Curso { get; set; }

publicvirtual DbSet<Inscricao> Inscricao { get; set; }

publicvirtual DbSet<Aluno> Aluno { get; set; }

}

Etapa 1 - Antes de executar o aplicativo, você precisa habilitar a migração, execute o comando enable-migrations no Console do Gerenciador de Pacotes.



O comando Enable-Migrations criará a classe Configuration derivada de DbMigrationsConfiguration com AutomaticMigrationsEnabled = false.

Etapa 2 - Agora, você precisa definir o inicializador do banco de dados MigrateDatabaseToLatestVersion na sua classe de contexto, conforme mostrado abaixo.

publicclassMyContext : DbContext

{

public MyContext() : base("SchoolDB")

{

Database.SetInitializer(new MigrateDatabaseToLatestVersion<

MyContext, EF\_CodeBasedFirst.Migrations.Configuration>());

}

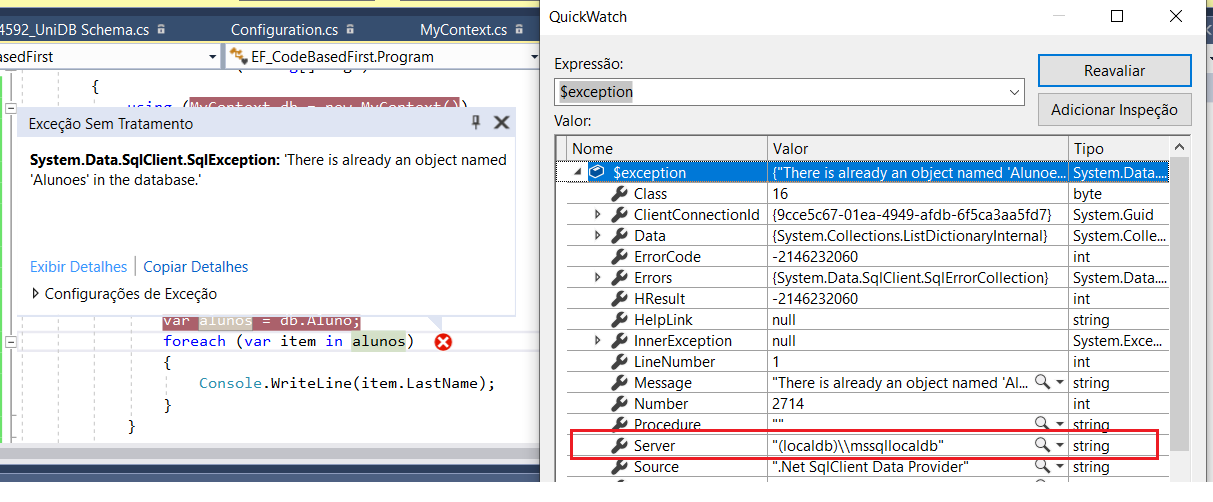
publicvirtual DbSet<Curso> Curso { get; set; }

publicvirtual DbSet<Inscricao> Inscricao { get; set; }

publicvirtual DbSet<Aluno> Aluno { get; set; }

}

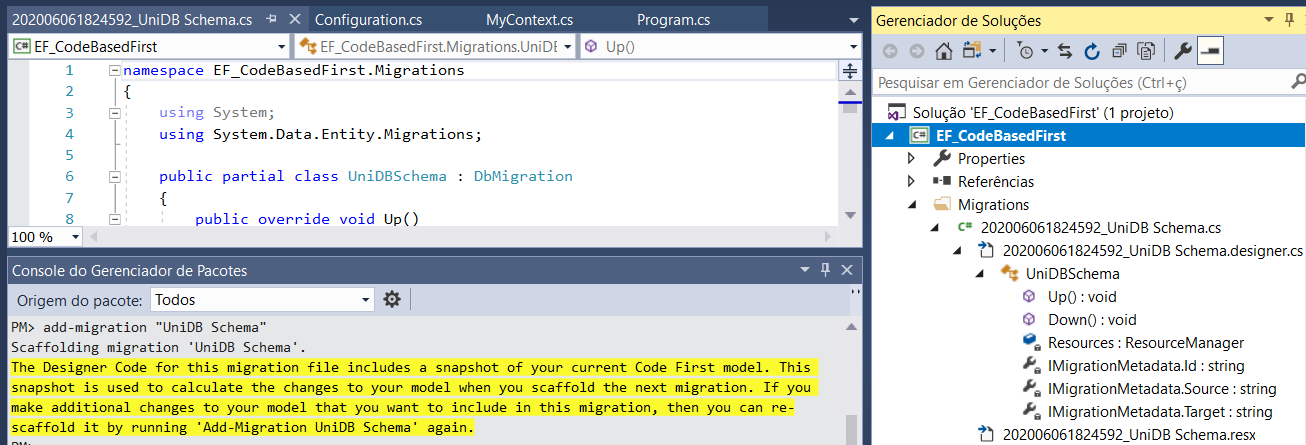
Repare que o Migrations.Configuration não carrega mais a indicação do nome da configuração no App.config. Um dos principais problemas dessa abordagem, que ocorreu no outro exemplo também é que, se você não souber aonde exatamente esta mapeado o seu serviço local do SQL, a migração gera o banco mas você não acaba sabendo exatamente onde. Em exemplos, indicavam que o banco estaria sendo gerado em localhost\SQLEXPRESS, porém após a geração do banco neste exemplo, descobrimos finalmente aonde estava sendo gerado nossas migrações locais (em (localdb)\mssqllocaldb ) apesar de estarmos sempre fazendo as conexões em banco com o servidor “localhost”....



Etapa 3 - A migração já está ativada, agora adicione a migração ao seu aplicativo executando o seguinte comando.

add-migration "UniDB Schema"

Etapa 4 - Quando o comando for executado com êxito, você verá que um novo arquivo foi criado na pasta Migração com o nome do parâmetro que você passou para o comando com um prefixo de carimbo de data/hora, conforme mostrado na imagem a seguir.



O comando acima criará um arquivo <timestamp> \_UniDB Schema.cs com os métodos Up() e Down(). Como você pode ver, o método Up() contém código para criar objetos de banco de dados e o método Down() contém código para descartar ou excluir objetos de banco de dados. Você também pode escrever seu próprio código personalizado para configurações adicionais. Essa é a vantagem sobre a migração automatizada.

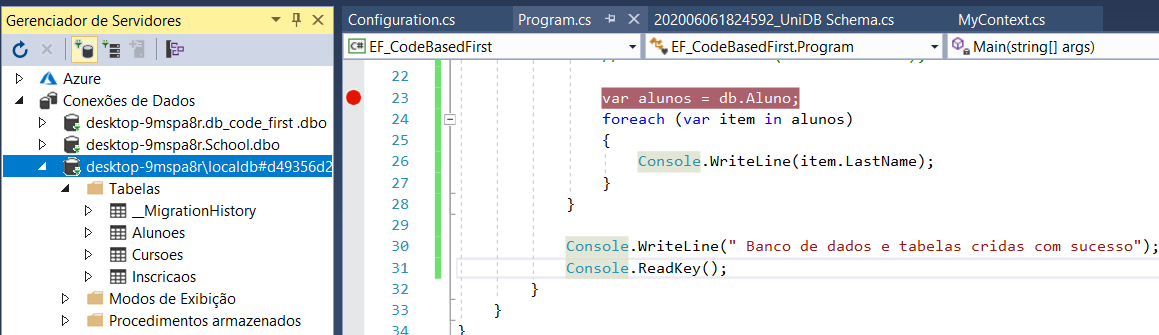
Para saber mais sobre os parâmetros do comando add-migration, execute os comandos get-help add-migration ou get-help add-migration -detailed no PMC, conforme mostrado abaixo.

**get-help add-migration**

Etapa 5 - Após criar um arquivo de migração usando o comando add-migration, é necessário atualizar o banco de dados. Execute o comando Update-Database para criar ou modificar um esquema de banco de dados. Use a opção –verbose para visualizar as instruções SQL que estão sendo aplicadas ao banco de dados de destino.

update-database -Verbose

Execute o comando get-help update-database ou get-help update-database -detailed no PMC para saber mais sobre o comando.Nesse ponto, o banco de dados será criado ou atualizado. Agora, sempre que você alterar as classes de domínio, execute Add-Migration com o parâmetro name para criar um novo arquivo de migração e execute o comando Update-Database para aplicar as alterações no esquema do banco de dados.



Observe que aqui nota-se a alteração dos nomes da classe contexto do plural para o singular, pois as chamadas que antes eram db.Alunos agora passaram a ser db.Aluno. Inserindo alguns dados.

Aluno st = new Aluno();

st.EnrollmentDate = DateTime.Now;

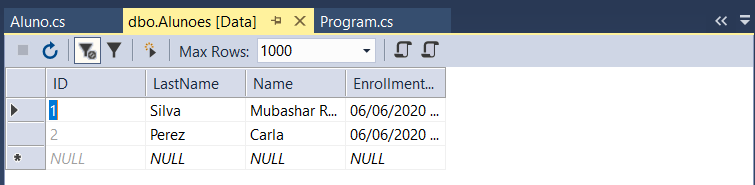
st.Name = "Carla";

st.LastName = "Perez";

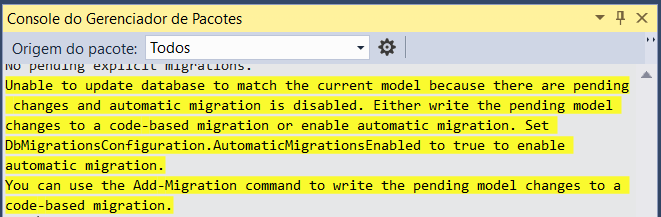
db.Aluno.Add(st);

db.SaveChanges();

Console.WriteLine("Aluno Added!");



Etapa 6 - adicione mais uma propriedade 'Idade' na turma do aluno. Se tentarmos apenas editar a propriedade e executarmos o comando update-database –Verbose dá o seguinte erro:

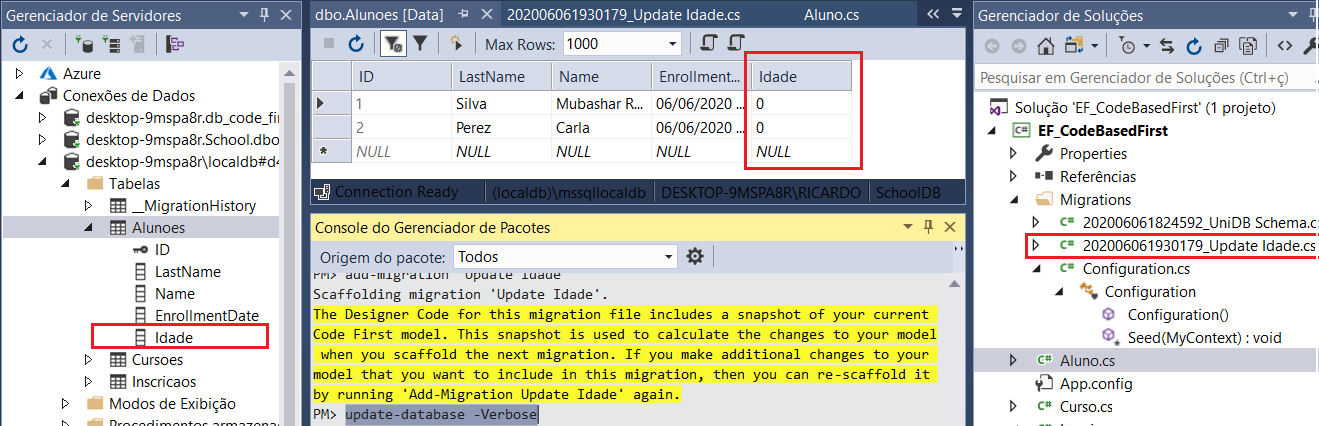


Mas se executaras instruções de adição e atualização.

add-migration "Update Idade"

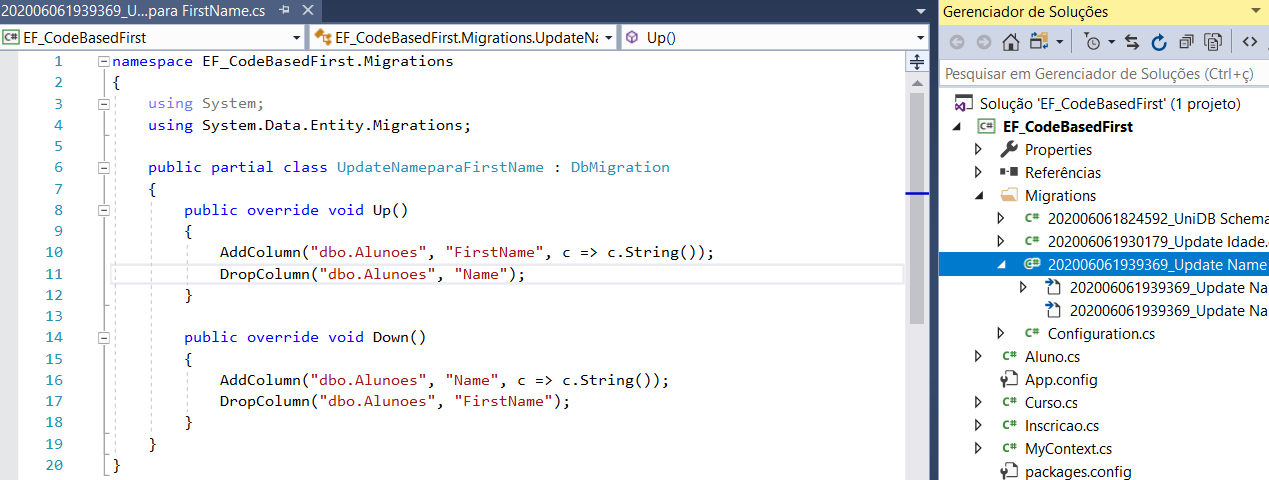
update-database -Verbose

Quando você executa PM → Atualizar banco de dados - detalhado, quando o comando é executado com êxito, você verá que a nova coluna Idade é adicionada ao seu banco de dados.



E se tentarmos editar uma propriedade que já haja dados gravados como, por exemple, o nome do aluno:

add-migration "Update Name para FirstName"



Repare nos métodos Up e Down gerados, indicando que irá adicionar a coluna FirstName e excluir a coluna Name.Mas antes de enviar isto para o banco, vamos imaginar que você queira setar um valor para o campo FirstName, por exemplo, todos os campos Ativos deverão estar true. Você pode fazer isto através do comando Sql() dentro do arquivo do migrations:

publicoverridevoid Up()

{

AddColumn("dbo.Alunoes", "FirstName", c => c.String());

Sql("update Alunoes set FirstName = 'Bob'");

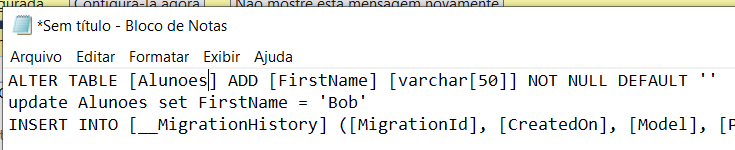
DropColumn("dbo.Alunoes", "Name");

}

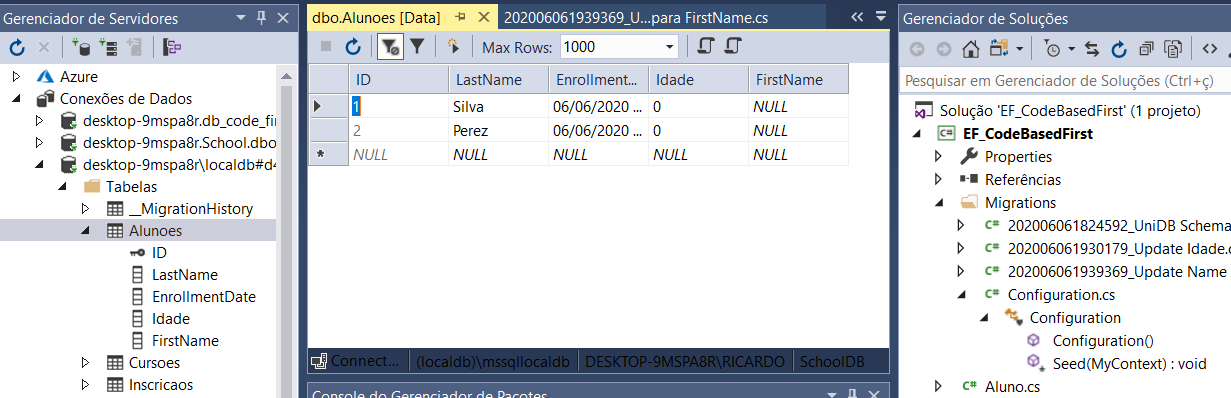
Agora vamos enviar o comando para o banco, mas de uma maneira um pouco diferente, vamos gerar um script SQL. Para isto execute o comando da seguinte maneira:

Update-DataBase –script

E o resultado será um arquivo de script SQL:



Basta executar este arquivo dentro do SQL que o campo será criado e atualizado, ou se preferir, execute novamente o Update-DataBase que isto será feito automaticamente.Este recurso do script pode ser util se você precisar atualizar também um ambiente e produção da sua aplicação.



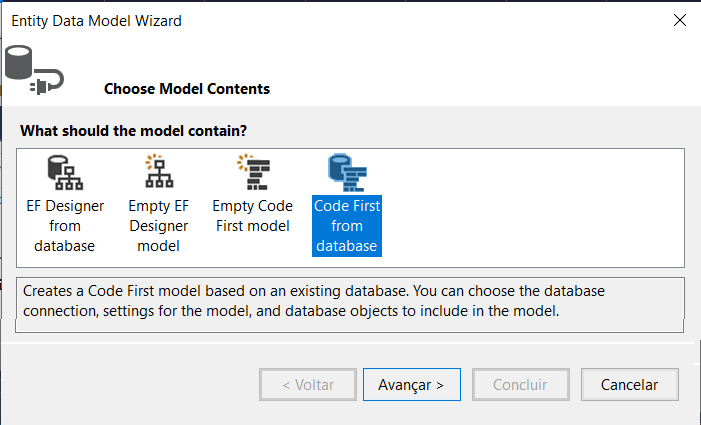
O mesmo problema da abordagem automática ocorreu, ou seja, excluiu a coluna com dados e anexou a nova coluna. Um recurso bem interessante do Migrations é que você pode voltar o seu banco de dados a qualquer ponto em que tenha executado um Add-Migrations. Imagine então que nós criamos o campo ativo no banco e agora você queira voltar ao ponto onde este campo ainda não existia, para isto vamos executar o seguinte comando do Migrations:

Update-DataBase –target “CriacaoBanco” ou update-database -TargetMigration:SchoolDB-v1

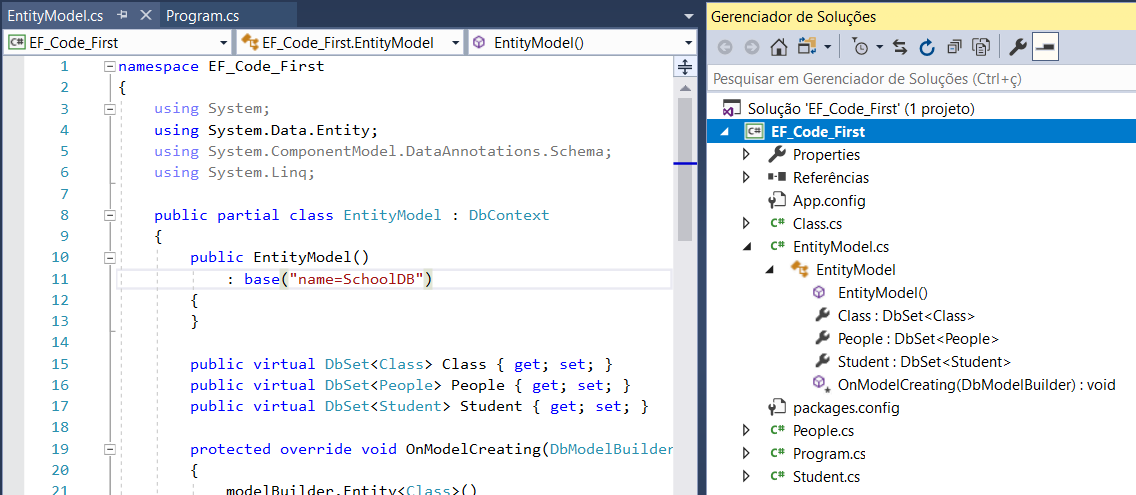
Isto volta nosso banco de dados a este ponto, que nosso caso foi a criação do banco, mas poderia ser qualquer outro ponto. Lembrando que este comando afeta apenas o banco de dados e não a nossa classe.

Code First (Code first from database)

Se você deseja usar o Code First para mapear essa entidade para um banco de dados, siga o mesmo procedimento para criação do .edmx em Database First (EF Designer from database)



Isso irá gerar as classes de entidade para suas tabelas e visualizações de banco de dados, como mostrado abaixo.



A parte mais interessante de tudo isso é que essas classes que a EF gerou para nós são as mesmas classes geradas pelas ferramentas de geração de código do EDMX. A única coisa que o Code-First realmente está fazendo de maneira diferente é permitir acesso direto a esses arquivos gerados, removendo o designer.

using (EntityModel ctx = new EntityModel())

{

var person = new People { FirstName = "John", LastName = "Doe" };

ctx.People.Add(person);

ctx.SaveChanges();

// Display all Blogs from the database

var query = from b in ctx.People

orderby b.FirstName

select b;

Console.WriteLine("All People in the database:");

foreach (var item in query)

{

Console.WriteLine(item.FirstName);

}

}

O assistente de Code First para banco de dados foi criado para gerar um conjunto de classes de ponto de partida que você pode ajustar e modificar. Se o esquema do banco de dados for alterado, você poderá editar manualmente as classes ou executar outra engenharia reversa para substituir as classes.

CONSUMINDO DADOS XML E JSON

XML e JSON são usados principalmente para comunicação na rede entre aplicativos diferentes ou plataformas diferentes. Discutiremos esses dois formatos de transmissão de mensagens / dados pela Internet com uma breve descrição.

Consumindo XML

XML (Extensible Markup Language) é uma linguagem de marcação que consiste em um conjunto de regras que definem como um documento deve ser formatado. O .NET Framework fornece classes presentes no System.Xml.dll para trabalhar com XML que foi basicamente projetado para armazenar e transportar dados. A vantagem do XML é que ele pode ser lido por humanos e programas de computador. Foi originalmente criado para ser a única solução de comunicação pela Internet entre diferentes aplicativos.

<?xmlversion="1.0"encoding="utf-8" ?>

<people>

<personfirstName="John"lastName="Doe">

<contactdetails>

<emailaddress>john@unknown.com</emailaddress>

</contactdetails>

</person>

<personfirstName="Jane"lastName="Doe">

<contactdetails>

<emailaddress>jane@unknown.com</emailaddress>

<phonenumber>001122334455</phonenumber>

</contactdetails>

</person>

</people>

Como você pode ver, um documento XML consiste em vários elementos. Primeiro, há uma linha opcional que especifica que você está olhando para XML. Essa linha é chamada de prólogo. Diz algo sobre a codificação usada e pode conter outros metadados. Após o prólogo, vem o conteúdo. Deve haver um único elemento raiz que contém o restante das informações. Dessa forma, você cria uma árvore hierárquica que define o relacionamento entre todos os elementos.

Abaixo do elemento raiz, há outros elementos filhos. Um elemento filho pode ser um único elemento que descreve alguma característica ou vários elementos que agrupam outros elementos. Os elementos podem conter atributos, que são pares nome-valor associados a um elemento. Um documento XML também pode conter comentários e instruções especiais de processamento. As instruções especiais de processamento geralmente estão contidas em um arquivo DTD (Document Type Definition) que é armazenado externamente no XML e referenciado no prólogo.

XML no .NET Framework

Analisar um arquivo XML como se fosse um arquivo de texto comum é muito trabalhoso. O .NET Framework ajuda você, fornecendo classes que podem ser usadas para analisar, criar e editar arquivos XML - na memória e no disco.

Essas classes são encontradas no namespace System.Xml. As classes importantes estão listadas na Tabela 4-3.

|  |  |
| --- | --- |
| **Name** | **Descrição** |
| XmlReader | Uma maneira rápida de ler um arquivo XML. Podesomente avançar através do arquivo e nada é armazenado em cache, por isso que é rápido e consome menos memória. |
| XmlWriter | Uma maneira rápida de criar um arquivo XML. Assim como no XmlReader, ele podesomente avançar e apenas e não armazenado em cache. |
| XmlDocument | Esta classe lê todo o XML na memória e permite navegar e editar o XML. |
| XPathNavigator | Ajuda na navegação por um documento XML para encontrar informações específicas. |

XmlReader

A classe XmlReader oferece a opção mais rápida de trabalhar com dados XML. É uma classe base abstrata que é herdada por classes como XmlTextReader, XmlNodeReader e XmlValidatingReader. Você cria uma nova instância do XmlReader usando o método estático Create. Você pode transmitir a esse método uma instância de XmlReaderSettings para configurar como o XML deve ser analisado. Dessa forma, você pode optar por ignorar dados do seu arquivo XML, como espaço em branco e comentários, ou iniciar em uma posição específica. O exemplo de como usar um XmlReader para analisar uma sequência que contém os dados XML em um arquivo xml ou de uma string.

staticvoid Parsing\_XmlReader(StringBuilder sb)

{

// This will get the current WORKING directory (i.e. \bin\Debug)

string workingDirectory = Environment.CurrentDirectory;

// or: Directory.GetCurrentDirectory() gives the same result

// This will get the current PROJECT directory

string projectDirectory = Directory.GetParent(workingDirectory).Parent.FullName;

string xmlString = System.IO.File.ReadAllText(projectDirectory + "\\XMLFile1.xml");

//Se for de um StringBuilder/StringWriter

var xml = sb.ToString();

using (StringReader stringReader = new StringReader(xmlString))

{

using (XmlReader xmlReader = XmlReader.Create(stringReader,

new XmlReaderSettings() { IgnoreWhitespace = true }))

{

xmlReader.MoveToContent();

xmlReader.ReadStartElement("people");

string firstName = xmlReader.GetAttribute("firstName");

string lastName = xmlReader.GetAttribute("lastName");

Console.WriteLine("Person: {0}{1}", firstName, lastName);

xmlReader.ReadStartElement("person");

Console.WriteLine("ContactDetails");

xmlReader.ReadStartElement("contactdetails");

string emailAddress = xmlReader.ReadString();

Console.WriteLine("Email address: {0}", emailAddress);

Console.WriteLine("=======xmlReader.Read()=======");

while (xmlReader.Read())// read the entire xml

{

Console.WriteLine(xmlReader.Value);

}

}

}

}

Como você pode ver, o XmlReader trata o XML como uma hierarquia de nós. Isso é mais fácil do que trabalhar com XML como se fosse um arquivo simples. Ao passar para nós, nós filhos e atributos, você pode analisar o documento para o conteúdo necessário.

XmlWriter

Quando você deseja criar um arquivo XML, pode usar a classe XmlWriter. Essa classe é criada usando o método estático Create e pode ser configurada usando uma instância da classe XmlWriterSettings. O código abaixo mostra como criar um novo arquivo XML. Como você pode ver, você tem métodos para escrever os elementos e atributos individuais.

static StringBuilder CreatingXM\_XmlWriter(StringBuilder sb)

{

using (StringWriter stream = new StringWriter(sb))

{

using (XmlWriter writer = XmlWriter.Create(stream, new XmlWriterSettings() { Indent = true }))

{

writer.WriteStartDocument();

writer.WriteStartElement("people");

writer.WriteStartElement("person");

writer.WriteAttributeString("firstName", "John");

writer.WriteAttributeString("lastName", "Doe");

writer.WriteStartElement("contactdetails");

writer.WriteElementString("EmailAddress", "john@unknown.com");

writer.WriteEndElement();

writer.WriteEndElement();

writer.Flush();

}

}

return sb;

}

Assim como na classe XmlReader, a classe XmlWriter está ciente da estrutura hierárquica do XML. No entanto, você precisa certificar-se de escrever a tag inicial e a final de cada elemento. Você pode optar por adicionar atributos ou adicionar elementos que tenham um valor de sequência como conteúdo.

XmlDocument

Embora XmlReader e XmlWriter sejam as opções mais rápidas, elas definitivamente não são as mais fáceis de usar. Quando você trabalha com documentos relativamente pequenos e o desempenho não é tão importante, você pode usar a classe XmlDocument. Sua função principal permite editar arquivos XML e representa o XML de forma hierárquica na memória e permite navegar facilmente no documento e editar elementos no local. Depois de terminar de editar o documento, você pode salvá-lo de volta a um arquivo ou fluxo.

Um XmlDocument usa um conjunto de objetos XmlNode para representar os vários elementos que compõem o documento. XmlDocument herda de XmlNode e adiciona recursos específicos para carregar e salvar documentos. O XmlNode ajuda na leitura de conteúdo e atributos e fornece métodos para adicionar nós filhos, para que você possa estruturar facilmente seu documento. O código a seguir mostra um exemplo no qual os recursos de leitura e edição do XmlDocument são usados ​​no XML.

XmlDocument doc = new XmlDocument();

// This will get the current WORKING directory (i.e. \bin\Debug)

string workingDirectory = Environment.CurrentDirectory;

// or: Directory.GetCurrentDirectory() gives the same result

// This will get the current PROJECT directory

string projectDirectory = Directory.GetParent(workingDirectory).Parent.FullName;

string xmlString = System.IO.File.ReadAllText(projectDirectory + "\\XMLFile1.xml");

Console.WriteLine("=====sReader====");

//to read xml string as a stream

StringReader sReader = new StringReader(xmlString);

doc.Load(sReader);

foreach (XmlNode item in doc.DocumentElement)

{

Console.WriteLine(item.InnerText);

}

doc.LoadXml(xmlString);

XmlNodeList nodes = doc.GetElementsByTagName("person");

// Output the names of the people in the document

foreach (XmlNode node in nodes)

{

string firstName = node.Attributes["firstName"].Value;

string lastName = node.Attributes["lastName"].Value;

Console.WriteLine("Name: {0}{1}", firstName, lastName);

}

// Start creating a new node

XmlNode newNode = doc.CreateNode(XmlNodeType.Element, "person", "");

XmlAttribute firstNameAttribute = doc.CreateAttribute("firstName");

firstNameAttribute.Value = "Foo";

XmlAttribute lastNameAttribute = doc.CreateAttribute("lastName");

lastNameAttribute.Value = "Bar";

newNode.Attributes.Append(firstNameAttribute);

newNode.Attributes.Append(lastNameAttribute);

doc.DocumentElement.AppendChild(newNode);

Console.WriteLine("Modified xml...");

doc.Save(Console.Out);

XPathNavigator

Um recurso bacana para navegar por um documento é o XPath, um tipo de linguagem de consulta para documentos XML. XmlDocument implementa IXPathNavigable para que você possa recuperar um objeto XPathNavigator a partir dele. O XPathNavigator oferece uma maneira fácil de navegar por um documento XML. Você pode usar métodos semelhantes aos de um XmlDocument para mover de um nó para outro ou pode usar uma consulta XPath. Isso permite selecionar elementos ou atributos com determinados valores, semelhante à maneira como o SQL seleciona dados em um banco de dados. A Listagem 4-46 mostra um exemplo de como usar uma consulta XPath para selecionar uma People pelo nome.

XmlDocument doc = new XmlDocument();

doc.LoadXml(xml); // Can be found in Listing 4-43

XPathNavigator nav = doc.CreateNavigator();

string query = "//people/person[@firstName='Jane']";

XPathNodeIterator nodes\_xpath = nav.Select("/people/person");

nodes\_xpath.MoveNext();

XPathNavigator nodesNavigator = nodes\_xpath.Current;

XPathNodeIterator nodesText = nodesNavigator.SelectDescendants(XPathNodeType.Text, false);

while (nodesText.MoveNext())

Console.WriteLine(nodesText.Current.Value);

XPathNodeIterator iterator = nav.Select(query);

Console.WriteLine(iterator.Count); // Displays 1

while (iterator.MoveNext())

{

string firstName = iterator.Current.GetAttribute("firstName","");

string lastName = iterator.Current.GetAttribute("lastName","");

Console.WriteLine("Name: {0}{1}", firstName, lastName);

}

Consumindo JSON

Outro formato popular usado por muitos serviços da web é o JavaScript Object Notation (JSON). Esses tipos de formatos (XML e JSON) são usados pelos Serviços da Web ou APIs da Web. Embora o XML seja útil, é detalhado e possui muitas regras em relação à estrutura de um documento. JSON é o que chamamos de alternativa "sem gordura" ao XML. Tem uma gramática mais fácil e geralmente carrega significativamente menos peso. O exemplo abaixo mostra como o XML People dos exemplos anteriores pode ser representado usando JSON.

{

"People": {

"Person": [

{

"firstName": "John",

"lastName": "Doe",

"ContactDetails": { "EmailAddress": "john@unknown.com" }

},

{

"firstName": "Jane",

"lastName": "Doe",

"ContactDetails": {

"EmailAddress": "jane@unknown.com",

"PhoneNumber": "001122334455"

}

}

]}

}

Ao trabalhar com XML, você usa classes como XmlWriter, XmlReader e XmlDocument; JSON não tem classes como elas. Normalmente, ao trabalhar com JSON, você usa uma biblioteca de serialização que ajuda a converter objetos em JSON e vice-versa. O .NET fornece uma classe JavaScriptSerializer para análise de dados JSON. Além disso, usamos a biblioteca Newtonsoft.Json para analisar dados JSON.

O JSON é usado com mais frequência em cenários assíncronos de JavaScript e XML (AJAX). Na realidade, o AJAX deve ser chamado de AJAJ porque o XML é trocado por JSON agora. AJAX é uma tecnologia que permite que um site execute chamadas em segundo plano para um servidor web. Essa é a técnica que permite que um site se atualize sem fazer uma atualização completa da página. JSON é particularmente útil para isso, porque todos os mecanismos JavaScript podem analisar uma cadeia JSON em um objeto por um comando relativamente simples.

USANDO WEB SERVICES

Outra opção além de armazenar dados em um banco de dados relacional ou em um arquivo é usar um serviço externo para armazenar dados. Pode ser um serviço da web que você criou ou pode ser de terceiros. Com esses serviços, você pode trocar dados entre aplicativos de uma maneira pouco acoplada. Você só precisa saber o endereço do serviço e como fazer uma solicitação para esse serviço. A implementação por trás da chamada está completamente oculta de seus consumidores, e você geralmente não precisa se preocupar com isso. O .NET Framework possui um suporte sólido para a criação de serviços da web. Você pode criar serviços da Web altamente flexíveis usando tecnologia .NET:

* **Serviço Web ASMX:** Esse recurso está incluído no .NET Framework 3.5 e abaixo, é uma abordagem antiga para criar serviços da Web que usa a classe “Visually Designed Class for web service". Mas a implementação continua válida até para Visual Studio 2017, .NET Framework 4.6.1.
* **WCF (Windows Communication Foundation):** pode enviar dados como mensagens assíncronas de um terminal em serviço para outro. Um terminal em serviço pode fazer parte de um serviço continuamente disponível hospedado pelo IIS ou pode ser um serviço hospedado em um aplicativo.

SERVIÇO WEB ASMX

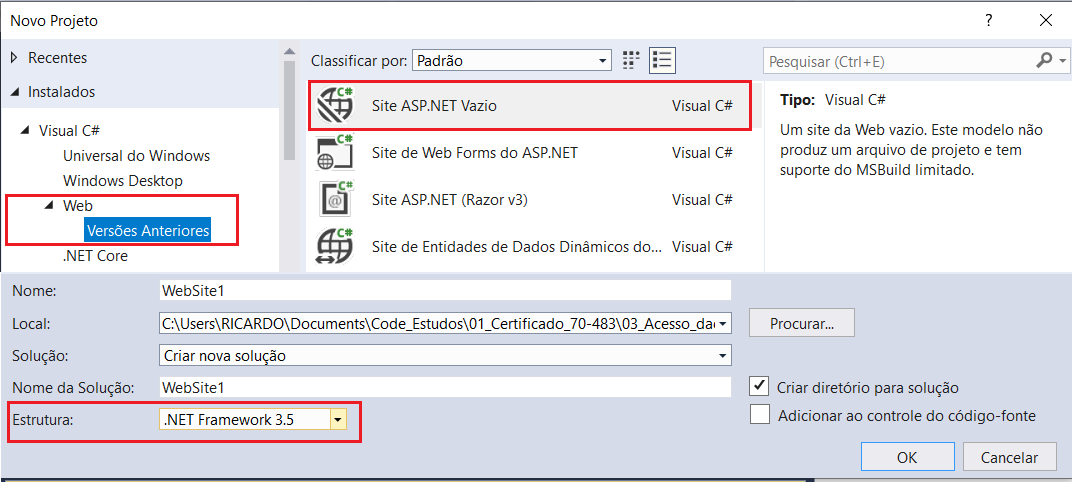
Há duas formas principais para criar um serviço da Web usando a "Visually Designed Class for web service":

1. Criando o serviço da Web
2. Criando proxy e consumindo o serviço da Web

Criando o serviço da Web

As etapas a seguir mostram o caminho para criar o serviço da web:

Etapa 1: Abra o VS, crie um novo projeto "Web">>“ASP.NET Empty Web Application”. Dê o nome que desejar (neste exemplo, eu o chamei de Web\_ASMX) e clique no botão "OK". O Framework selecionado foi ".NET Framework 3.5", mas testado com o Framework 4.6.1 sem nenhuma diferença.



Passo 2:Após a criação do projeto, clique com o botão direito do mouse em projeto >> Adicionar >> Novo Item. Selecione Serviço Web (ASMX) e nomeie o que quiser (neste exemplo, eu o nomeei SampleService) e clique no botão "Adicionar".

Etapa 3:Depois de adicionar o SampleService, o SampleService.asmx será adicionado ao seu projeto, que é uma classe Visually Designed, ou seja, seu serviço da web. Nesta classeos métodos são implementados pelo "WebMethod", você pode adicionar suas lógicas e métodos para expor para consumo esse serviço da Web no lado do cliente. Esses atributos são necessários para expor o serviço, caso contrário, se o atributo de serviço da Web estiver ausente, o serviço não será exposto ao cliente. Adicione outras duas funções no arquivo SampleService.asmx (na classe SampleService no método HelloWorld), como:

[WebMethod]

publicint Add(int a, int b)

{

return a + b;

}

[WebMethod]

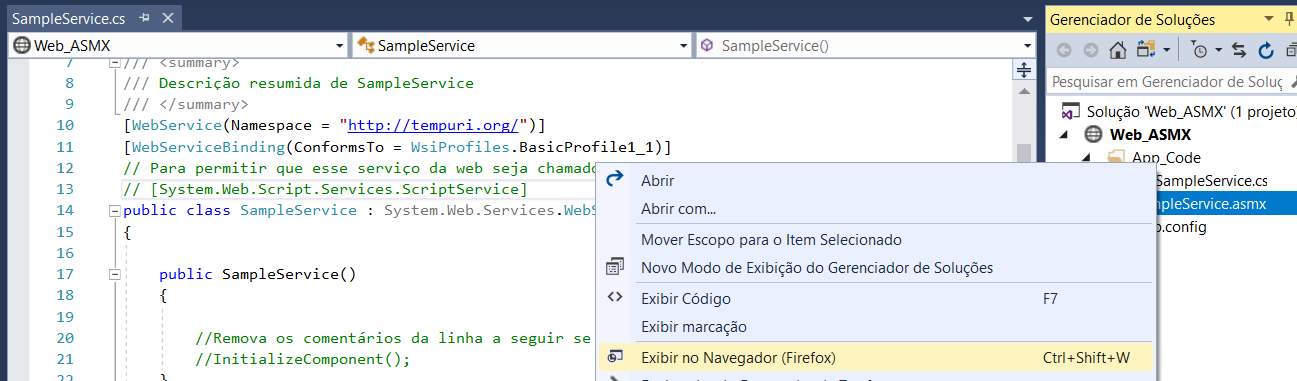
publicint Subtract(int a, int b)

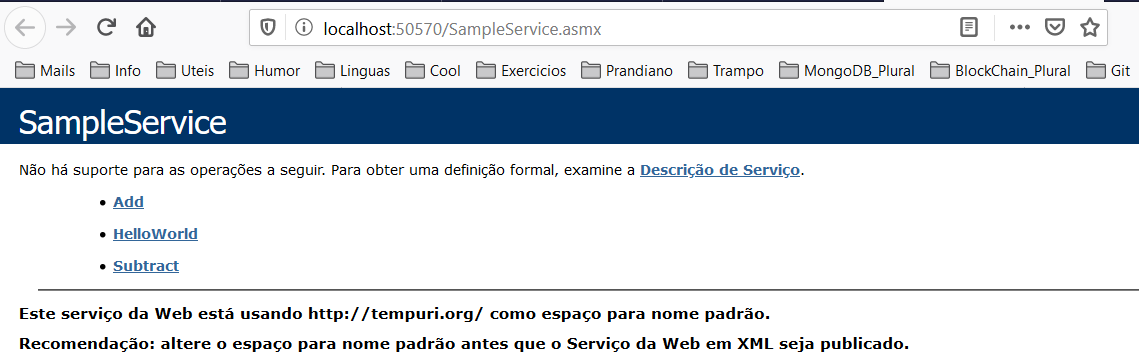
{

return a - b;

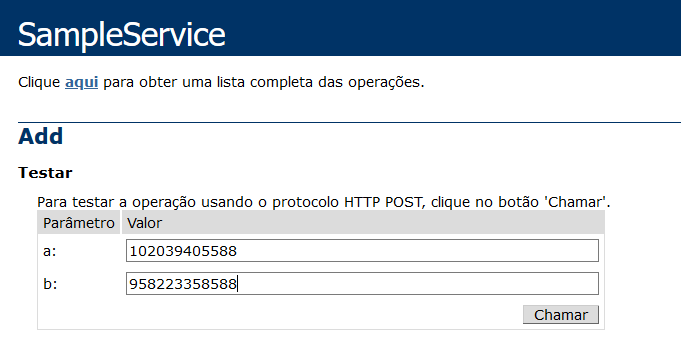
}

Etapa 4: Para testar este serviço, clique com o botão direito do mouse em SampleService.asmx >> Ver no navegador. A lista de métodos escritos em seu serviço está na sua frente. A visualização pode fornecer acesso para usá-lo como cliente (basicamente expõe os métodos do serviço da Web).

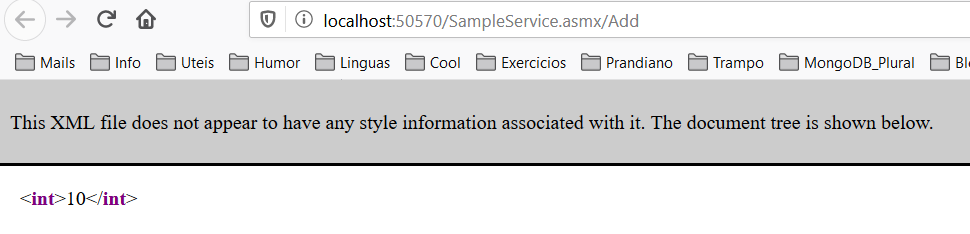




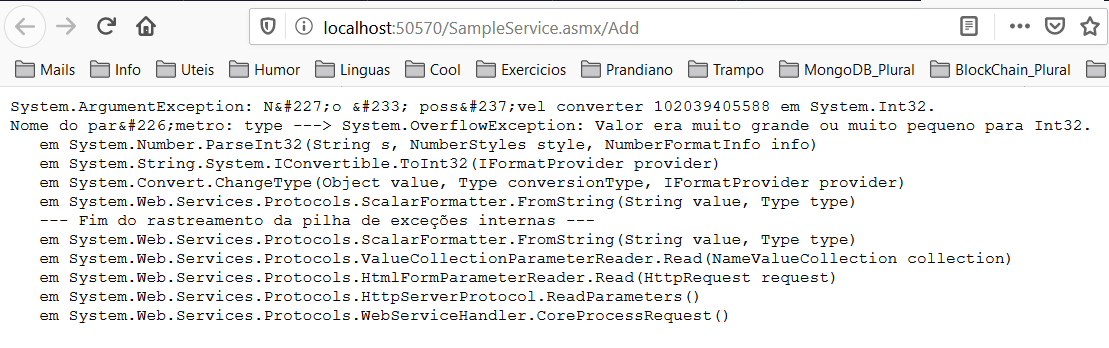
Passo 2:Clique em qualquer um para testar o método. (Clique em Adicionar.)



Etapa 3:Após executar o método, o resultado é exibido na forma de XML, conforme descrito, o formatos XML e JSON são usados para transportar dados pela rede e usados por um Serviço da Web ou APIs da Web.



Caso os dados forem inválidos, como for maior do que um Int32 ou uma string, ocorre uma exceção:



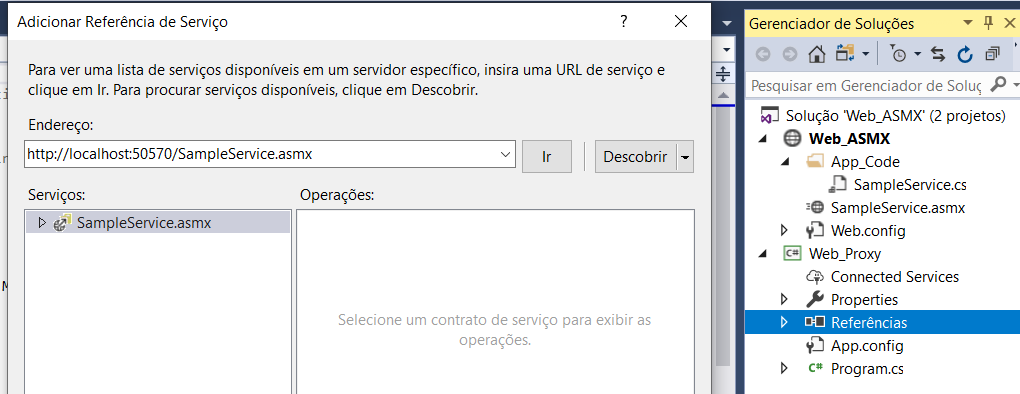
Para próxima etapa, lembre-se da Url utilizada:



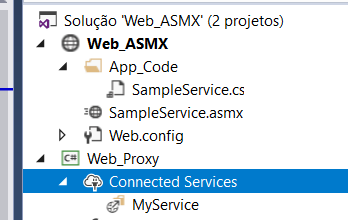
Criar proxy e consumir o serviço

A criação do Proxy é importante, pois é registrada no aplicativo cliente e permite que os métodos dos serviços da Web sejam usados como métodos locais no lado do cliente.Para isso, você deve seguir as seguintes etapas:

Passo 1: Crie outro projeto (Projeto C# Console) em sua solução e nomeie o que quiser. Clique com o botão direito do mouse em Referências >> Adicionar referência de serviço. Clique no botão “Discover”, pois ele buscará Web Serviceem sua solução. Você pode fornecer qualquer endereço externo para o Web Service que deseja consumir.



Você pode alterar o namespace da web descoberto. (Eu o chamei de MyService, que será usado posteriormente no código.)



Passo 2: Depois de clicar no botão "OK", uma pasta de referências de serviço é adicionada com as DLLs necessárias. Agora, no seu arquivo de código (arquivo de programa), basta escrever o seguinte trecho de código:

//Create the proxy for your service to use its methods

MyService.SampleServiceSoapClient proxy = new MyService.SampleServiceSoapClient();

int addResult = proxy.Add(5, 10);

int subtractResult = proxy.Subtract(100, 40);

Console.WriteLine("Addition Result is: " + addResult); // Addition Result is: 15

Console.WriteLine("Subtraction Result is: " + subtractResult); // Subtraction Result is: 60

MyService é o namespace que você adicionou ao adicionar SampleService neste projeto. Você pode criar um proxy com a classe escrita (acima no código).Quando houver uma alteração no Serviço, você só precisará atualizar sua referência adicionada a esse serviço. Isso é feito apenas expandindo a pasta "Referência de Serviço" no projeto do Cliente. Clique com o botão direito do mouse no seu serviço e clique em "Atualizar Referência de Serviço".

A classe do seu serviço é SampleService mas, no momento da criação de um proxy, você deve escrever SoapClient (sufixo) com o nome da classe SampleServiceSoapClient. Se o nome da classe de serviço for o MyService, a classe de proxy será MyServiceSoapClient.

SERVIÇO WEB WCF

No passado, você tinha muitas opções ao criar um serviço. Você poderia usar o .NET Remoting, XML Web Services (incluindo serviços baseados em SOAP) ou o Microsoft Message Queues, todos com suas próprias opções e recursos. O Windows Communication Foundation (WCF) substitui todas essas tecnologias por um modelo de programação único, unificado e extensível, que engloba as tecnologias do passado.

O WCF é uma estrutura para a criação de aplicativos orientados a serviços, permite acessar um banco de dados pela Web ou uma intranet usando um URI. Usando o WCF, você pode enviar dados como mensagens assíncronas de um terminal em serviço para outro. Um terminal em serviço pode fazer parte de um serviço continuamente disponível hospedado pelo IIS ou pode ser um serviço hospedado em um aplicativo.No serviço da Web WCF, a classe de serviço é implementada pelo atributo ServiceContract (como o serviço da Web no ASMX) e seus métodos são implementados pelo OperationContract (como o método da Web no ASMX).

|  |  |
| --- | --- |
| **Serviço WCF** | **Serviço ASMX** |
| O serviço WCF pode ser hospedado no IIS, WAS, Console, Host fornecido pelo WCF | O serviço ASMX pode apenas ser hospedado no IIS. |
| Ele suporta vários protocolos de comunicação, ou seja, HTTP, TCP, MSMQ e NamedPipes. | Ele suporta apenas HTTP. |
| Ele usa DataContractSerializer. | Ele usa XmlSerializer |

Nas versões anteriores do .NET, isso era chamado ADO.NET Data Services. Você pode selecionar, filtrar, adicionar, atualizar e até excluir dados usando um URI e parâmetros de string de consulta. O WCF Data Services usa o Open Data Protocol, OData, que é um protocolo da web que usa HTTP. Por exemplo, a solicitação a seguir pode ser feita para um Serviço de Dados WCF que expõe a tabela Categorias do banco de dados Northwinds.

http://localhost/WcfDataService1.svc/Categories? $ filter = CategoryName eq 'Beverages'

Neste exemplo, Categorias especifica a entidade a ser retornada, e o parâmetro filter (eq) na string de consulta é usado para encontrar a categoria com o nome 'Beverages'. Os espaços são permitidos na string de consulta. Você pode optar por retornar os dados como XML, caso em que segue o formato OData ATOM (a representação XML dos dados retornados de uma consulta OData) ou a JavaScript Object Notation, JSON (um formato leve de intercâmbio de dados). Por padrão, os dados são retornados como XML. O XML a seguir mostra a resposta para a chamada anterior ao Serviço de Dados do WCF:

O WCF é superior às tecnologias anteriores, pois a comunicação com um serviço WCF ocorre através de endpoints do serviço,não contendo nenhuma informação sobre como o serviço pode ser chamado - não especifica um endereço ou protocolo. Os endpoints fornecem aos clientes o acesso às funcionalidades oferecidas por um serviço WCF. O EndPoint do WCF é chamado também de modelo ABC:

* Address: define onde o serviço reside
* Binding: define como se comunicar com o serviço.
* Contract: define o que o serviço vai fazer

**Endpoint = Address + Binding + Contract**

Ao criar um serviço WCF, você geralmente começa com o contrato, que define quais operações seu serviço expõe. O contrato é o que o mundo exterior espera do seu serviço. Depois de especificar seu contrato, você especifica os bindings. Um bindingconfigura os protocolos e transportes que podem ser usados ​​para chamar seu serviço. Talvez seu serviço possa ser usado por HTTP, HTTPS e uma conexão de pipe nomeado. Em seguida, você precisa especificar o endereço, que é o ponto de extremidade que seu serviço expõe. Isso garante que haja um endereço de rede físico que possa ser usado para chamar seu serviço com um bindingespecífico.

Tipos de contratosWCF

A seguir temos um resumo dos 3 tipos de contratos WCF existentes:

Service Contracts

Contratos de serviços (Service Contracts) - Descrevem as operações que um serviço pode realizar. (Mapeia os tipos CLR para WSDL).

Data Contracts

Contratos de Dados (Data Contracts) - Descreve a estrutura de dados usada no serviço. (Mapeia tipos CLR para XSD). Um Data Contract é um acordo formal entre um serviço e um cliente que descreve os dados que serão trocados entre ambos. Para estabelecer a comunicação, o cliente e o serviço não necessitam trocar necessariamente os mesmos tipos de dados, devem trocar apenas os mesmos data contracts.

Um Data Contract especifica para cada parâmetro ou tipo de retorno qual informação será serializada (convertida em XML) para ser trocada. Os DataContracts são definidos através de classes e uma classe DataContract deverá ser decorada com o atributo DataContract e os campos e propriedades que o tipo possui devem ser decorados com o atributo DataMember. (A partir do service pack versão 3.5 da plataforma .NET isso não é mais obrigatório)

Message Contracts

Contratos de Mensagens (Message Contracts) - Define a estrutura de mensagens usadas no serviço. (Mapeia os tipos CLR para mensagens SOAP). Os contratos de mensagens permitem ao serviço interagir diretamente com mensagens e podem ser tipados ou não tipados. Eles são úteis em casos de interoperabilidade e quanto existe um formato de mensagem com o qual devemos dar conformidade e/ou customizar.

Por meio dos Contratos de Mensagens podemos ter um controle maior sobre a estrutura de mensagem SOAP usada pelo WCF efetuando uma personalização dos componentes da mensagem. Os Contratos de Mensagens são definidos em classes e as mesmas devem ser decoradas com o atributo MessageContract. Além disso uma classe definida com um contrato de Mensagem deve possuir um construtor sem parâmetros. Nas classes para especificar se um campo fará parte do header ou do body da mensagem devemos usar os atributos MessageHeader ou MessageBodyMember.

Aplicativo cliente que consome o serviço WCF

A seguir iremos criar um aplicativo cliente que consome o serviço WCF. Isso é semelhante à seção que descreveu como criar um modelo do ADO.NET Entity Framework e mostrou como selecionar, adicionar, atualizar e excluir registros. Esta seção cria um aplicativo de console que faz referência a um serviço WCF na mesma solução e executa todas as operações CRUD nos dados.

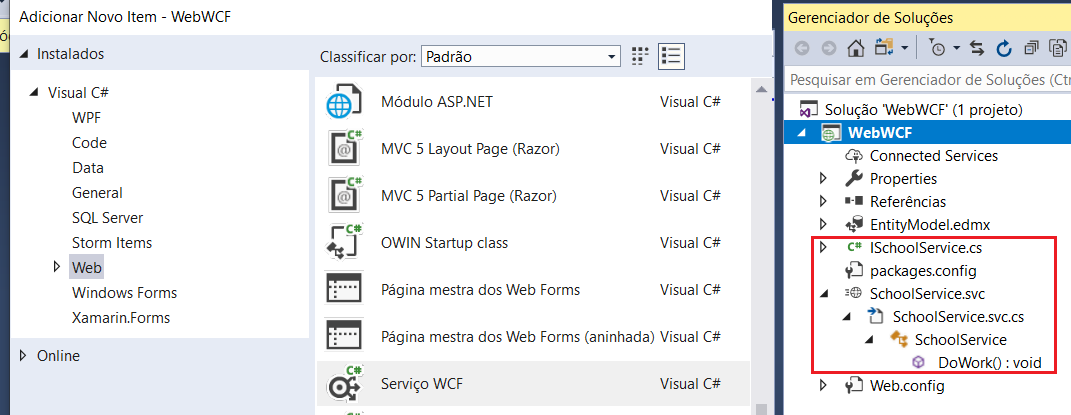
Criar um Serviço de Dados WCF envolve a criação de um aplicativo Web, a criação de um modelo ADO.NET Entity Framework para o banco de dados e, em seguida, a exposição do modelo, adicionando um arquivo WCF Data Service ao seu aplicativo Web. Para implementar um serviço WCF iniciamos pela definição do contrato para o serviço e então implementamos o contrato em um tipo de serviço. Um contrato para um serviço geralmente envolve a aplicação do atributo ServiceContract para uma interface e então a aplicação do atributo OperationContract para cada método exposto como parte do contrato do serviço. A seguir veja as regras para a definição de um contrato:

* O contrato deve ser uma interface contendo as assinaturas dos métodos do serviço;
* A interface deverá ser decorada com o atributo: ServiceContract de outra forma será lançada a exceção InvalidOperationException;
* Todas as operações que serão disponibilizadas devem ser decoradas com o atributo OperationContract;
* É obrigatória a definição de pelo menos uma operação com o atributo OperationContract;

Passo 1: Selecione Aplicativo da Web vazio do ASP.NET na lista de modelos C# instalados.

Passo 2: (Opcional) Especifique um número de porta específica para seu aplicativo Web. No Gerenciador de soluções, clique com o botão direito do mouse no projeto ASP.NET que você acabou de criar e escolha Propriedades. Selecione a guia Web e defina o valor da caixa de texto porta específica como 12345.

Passo 3: Clique com o botão direito do mouse no projeto no Gerenciador de Soluções, clique em Adicionar e selecione Novo Item no menu pop-up. Escolha WCF Data Service na lista de modelos da Web C# instalados. Caso não encontrar os templates WCF, você poderá adiciona-los através do instalador do Visual Studio na parte de intalação de componentes individuais. Nomeie o arquivo SchoolService.svc e clique no botão Adicionar.



Depois de criar o SchoolService, você pode ver dois arquivos criados,

* ISchoolService.cs: interface para declarar seus contratos de serviço e contratos de dados.
* SchoolService.scv.cs: é uma classe normal herdada pelo ISchoolService onde é possível definir todos os métodos e outros processos.

O SchoolService.svc contém instruções sobre como hospedar seu serviço no Internet Information Services (IIS), para que você possa colocar o arquivo.svc com o arquivo de código associado em um site hospedado pelo IIS para disponibilizá-lo aos usuários. O arquivo de código SchoolService.cs não contém nenhuma informação sobre como o serviço pode ser chamado - não especifica um endereço ou protocolo.

Passo 4: Você pode ver os dois arquivos tendo um tipo de coisa DoWork(). Você pode comentar ou remover ou deixar como está. Implementaremos um serviço da web Hello World. Altere DoWork() para retornar uma string "Hello World" na interface ISchoolService.cs:

[ServiceContract]

publicinterfaceISchoolService

{

[OperationContract]

string DoWork();

}

E na classe SchoolService.cs:

publicclassSchoolService : ISchoolService

{

publicstring DoWork()

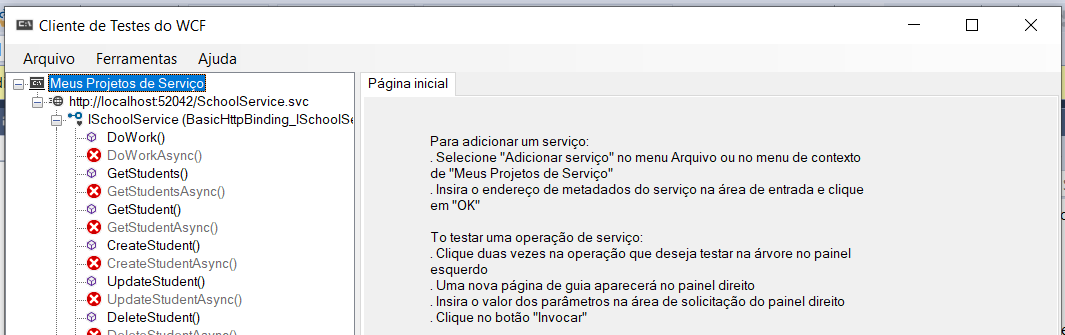
{

return"Hell World";

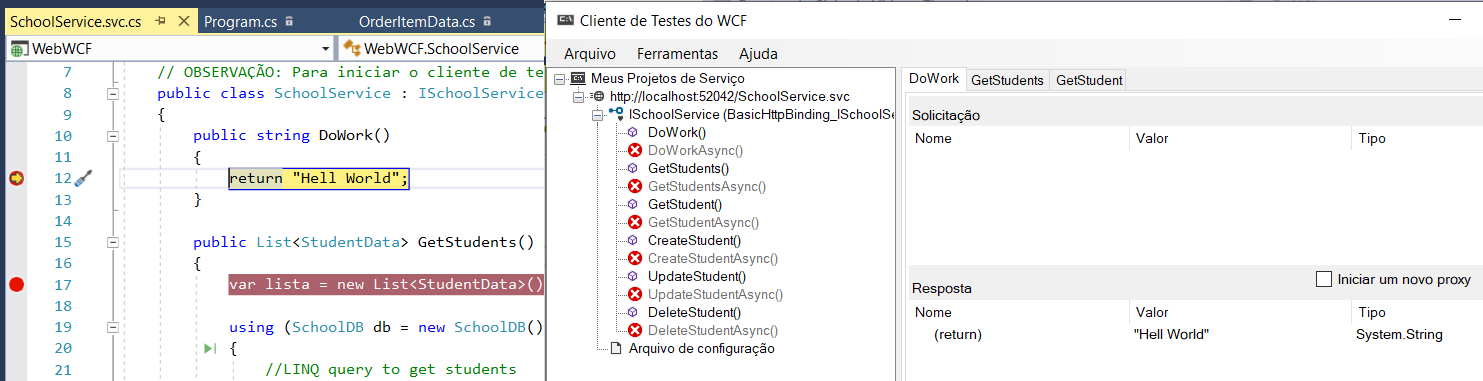
}

}

Passo 5: No Visual Studio, pressione a tecla F5 para iniciar a depuração do aplicativo. Se iniciar a depuração no arquivo SchoolService.scv.cs, o Visual Studio abre uma interface WCF Test Client de testes do serviço WFC.



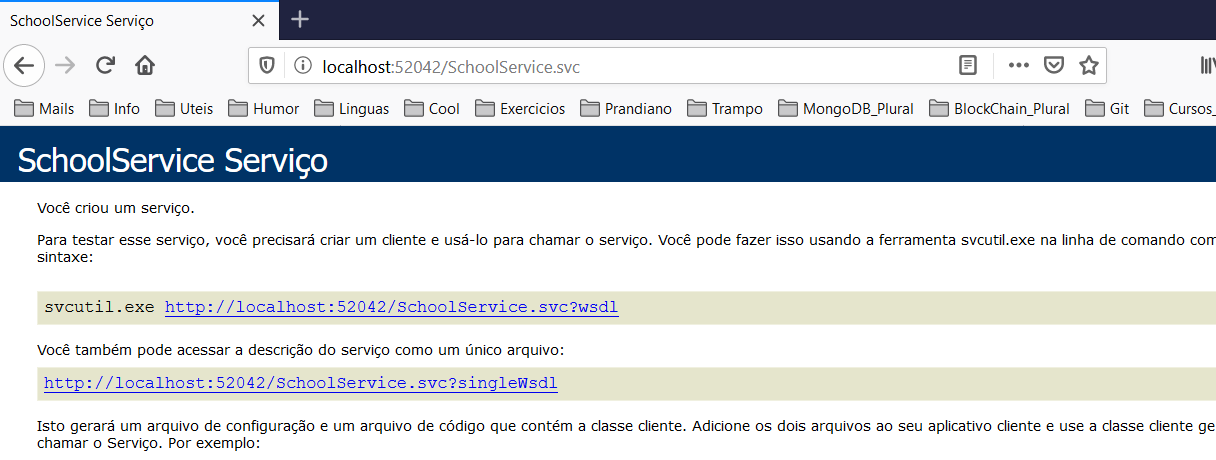
Inclusive é possível colocar breakpoints nos métodos e acompanhar passo-a-passo a execução dos mesmos.



Passo 6: Abra um navegador da Web no computador local. Na barra de endereços, digite a seguinte URL:

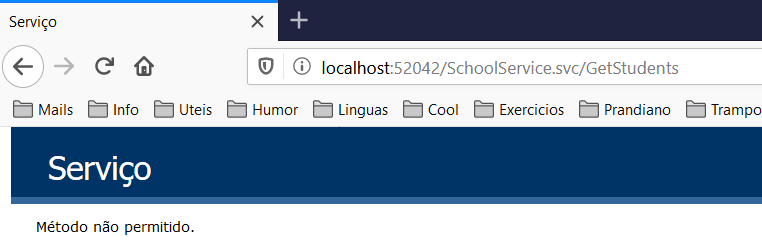
**http://localhost:52042/SchoolService.svc**

Isso retorna o documento padrão de serviço, que contém uma lista de conjuntos de entidades que são expostos por esse serviço de dados.



Passo 7: Para chamar métodos WCF a partir de um navegador, você precisa fazer duas coisas:

* Use os atributos [WebGet] e [WebInvoke] em seus métodos. Caso não colocar é emitido a seguinte mensagem:



* Configure o Web.config para usar um webHttpBinding para o terminal do seu serviço e ative o comportamento do webHttp.

<?xmlversion="1.0"encoding="utf-8"?>

<!--

Para obter mais informações sobre como configurar seu aplicativo ASP.NET, visite

https://go.microsoft.com/fwlink/?LinkId=169433

-->

<configuration>

<configSections>

<!-- .....

</configSections>

<connectionStrings>

<!-- .....

</connectionStrings>

<system.serviceModel>

<services>

<servicename="WebWCF.SchoolService"behaviorConfiguration="ServerBehaviorsManutencao">

<!-- Service Endpoints -->

<host>

<baseAddresses>

<addbaseAddress="http://localhost:52042/"/>

</baseAddresses>

</host>

<endpointaddress=""

binding="webHttpBinding"

behaviorConfiguration="EndPointBehaviorsManutencao"

contract="WebWCF.ISchoolService" />

</service>

</services>

<bindings>

<webHttpBinding>

<bindingname="default"/>

</webHttpBinding>

</bindings>

<behaviors>

<serviceBehaviors>

<behaviorname="ServerBehaviorsManutencao">

<serviceMetadatahttpGetEnabled="true"httpsGetEnabled="true" />

<serviceDebugincludeExceptionDetailInFaults="false" />

</behavior>

</serviceBehaviors>

<endpointBehaviors>

<behaviorname="EndPointBehaviorsManutencao">

<enableWebScript />

</behavior>

</endpointBehaviors>

</behaviors>

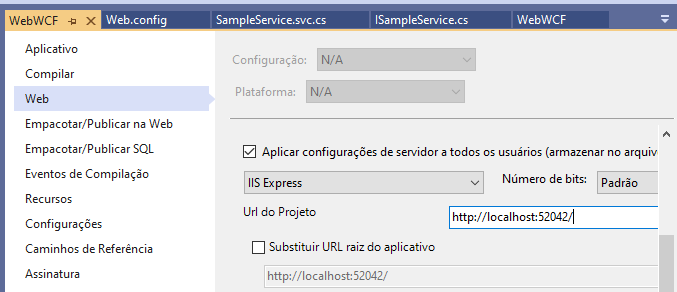
<serviceHostingEnvironmentaspNetCompatibilityEnabled="true"

multipleSiteBindingsEnabled="true" />

</system.serviceModel>

</configuration>

Observe que a url indicada em addbaseAddress deve a mesma que estive nas propriedades do projeto:



Passo 8: Crie 2 métodos para testar o retorno do WebGet e WebInvoke:

[ServiceContract]

publicinterfaceISchoolService

{

[OperationContract]

[WebGet]

string echoWithGet(string param1);

[OperationContract]

[WebInvoke(Method = "GET")]

string echoWithPost(string param1);

[OperationContract]

[WebGet]

string DoWork();

}

publicclassSchoolService : ISchoolService

{

publicstring echoWithGet(string s)

{

return"Get: " + s;

}

publicstring echoWithPost(string s)

{

return"Post: " + s;

}

publicstring DoWork()

{

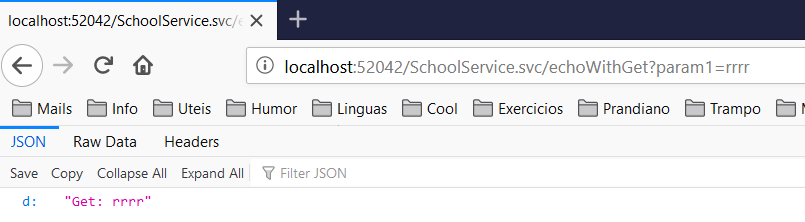
return"Hell World";

}

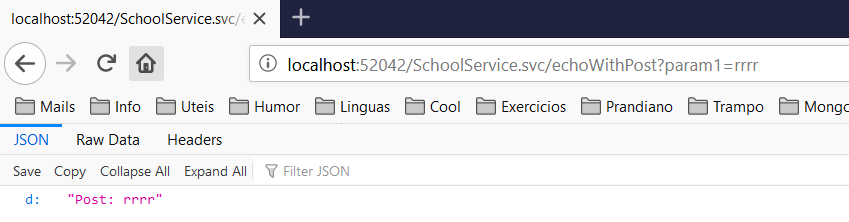
}

Passo 9: Chame os métodos, lembre-se que os parâmetros devem ser sempre tipo string:

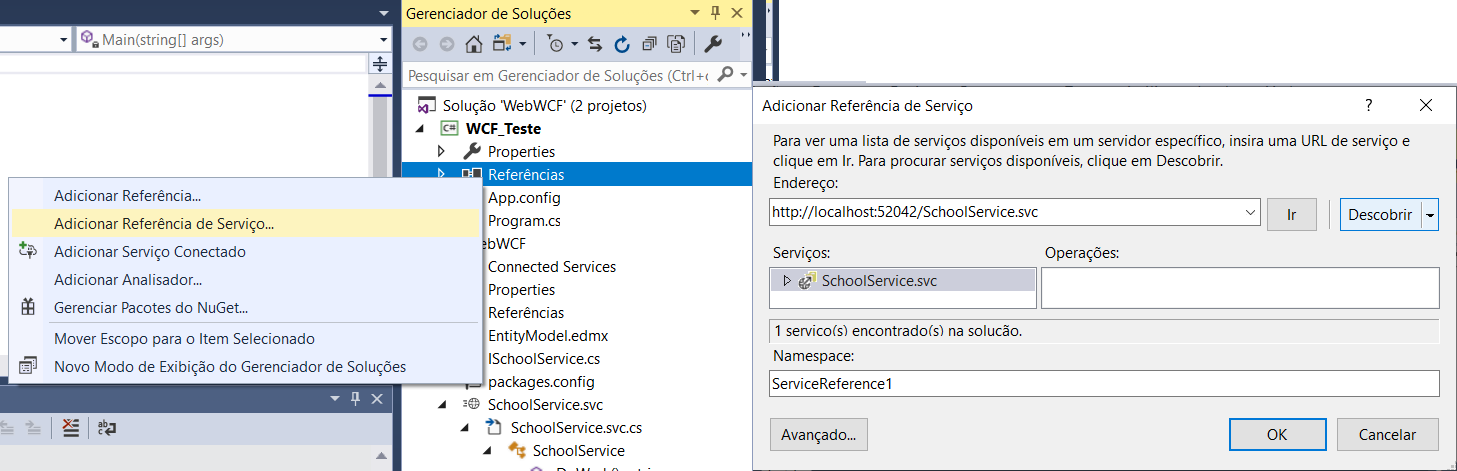
http://localhost:52042/SchoolService.svc/echoWithGet?param1=rrrr



<http://localhost:52042/SchoolService.svc/echoWithPost?param1=rrrr>



Passo 5: Para testar o consumo e a invocação desse código, vamos criar um novo aplicativo de console e adicioná-lo à solução. Selecione Adicionar referência de serviço e clique em Descobrir para selecionar o serviço criado:



using (SchoolServiceClient client = new SchoolServiceClient())

{

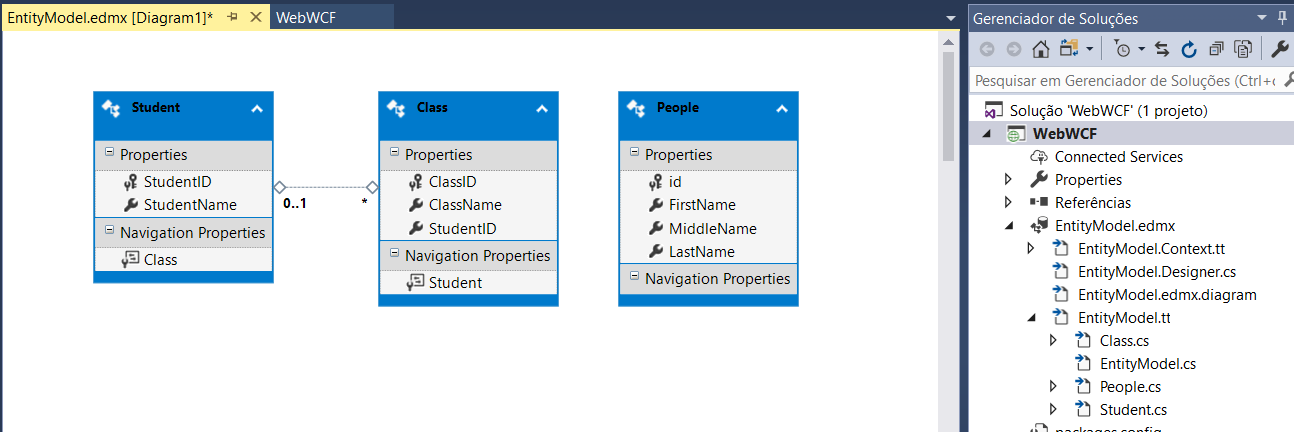
var retorno = client.DoWork();

Console.WriteLine(retorno); // Hell World

};

Agora que temos a aplicação funcionando e conectada ao serviço, podemos implementar o restante dos métodos que utilizará o modelo de dados do Entity Framework.

Passo 6: Clique com o botão direito do mouse no projeto no Gerenciador de Soluções, clique em Adicionar e selecione Novo Item no menu pop-up. Crie um modelo de dados de entidade ADO.NET para o banco de dados, assim como você fez em exercícios anteriores.



Passo 7: Crie uma classe que servirá para fazer o trafégo de objetos entre as aplicações utilizando serialização DataContract:

[DataContract]

publicclassStudentData

{

[DataMember]

publicint ID { get; set; }

[DataMember(Name = "Nome")]

publicstring Name { get; set; }

}

Passo 8: incluímos na interface do serviço ISchoolService.cs as chamadas dos métodos:

[ServiceContract]

publicinterfaceISchoolService

{

[OperationContract]

string DoWork();

[OperationContract]

List<StudentData> GetStudents();

[OperationContract]

StudentData GetStudent(int StudentID);

[OperationContract]

string CreateStudent(Student student);

[OperationContract]

string UpdateStudent(StudentData student);

[OperationContract]

string DeleteStudent(int StudentID);

}

Passo 9: Agora devemos criar os métodos na classes SchoolService.cs com as operações CRUD:

publicclassSchoolService : ISchoolService

{

publicstring DoWork()

{

return"Hell World";

}

public List<StudentData> GetStudents()

{

var lista = new List<StudentData>();

using (SchoolDB db = new SchoolDB())

{

//LINQ query to get students

var listaDB = (from p in db.Student

select p).ToList();

foreach (var item in listaDB)

{

lista.Add(new StudentData() { ID = item.StudentID, Name = item.StudentName });

}

}

return lista;

}

public StudentData GetStudent(int StudentID)

{

var student = new StudentData();

using (SchoolDB db = new SchoolDB())

{

var studentDB = db.Student.Find(StudentID);

student.ID = studentDB.StudentID;

student.Name = studentDB.StudentName;

}

return student;

}

publicstring CreateStudent(Student student)

{

using (SchoolDB db = new SchoolDB())

{

db.Student.Add(student);

db.SaveChanges();

}

return"Student Added!";

}

publicstring UpdateStudent(StudentData student)

{

using (SchoolDB db = new SchoolDB())

{

var EditedObj = db.Student.Find(student.ID);

if (EditedObj != null)//if student is found

{

EditedObj.StudentName = student.Name;

db.SaveChanges();

}

}

return"Student Updated!";

}

publicstring DeleteStudent(int StudentID)

{

using (SchoolDB db = new SchoolDB())

{

var EditedObj = db.Student.Find(StudentID);

if (EditedObj != null)

{

var classes = db.Class.Where(x => x.StudentID == StudentID);

foreach (var classe in classes)

{

classe.StudentID = null;

}

db.Student.Remove(EditedObj);

db.SaveChanges();

}

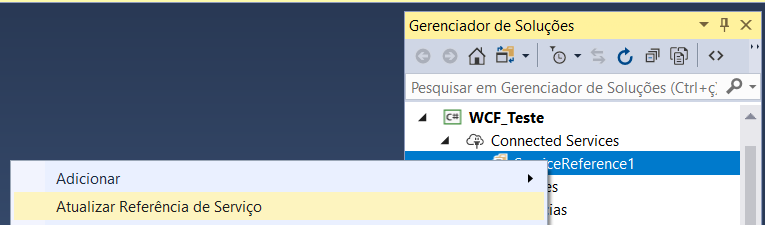
}

return"Student Removed!";

}

}

Passo 10: Em seguida, recompile o projeto WebWCF para registrar as mudanças e no projeto console WCF\_Teste, Clique para atualizar as referências do serviço:



Passo 11: Faça a referência ao serviço criado, instancie um objeto cliente que fará o uso dos métodos.

using WCF\_Teste.ServiceReference1;

using (SchoolServiceClient client = new SchoolServiceClient())

{

var students = client.GetStudents();

foreach (var student in students)

{

Console.WriteLine("ID is: " + student.StudentID);

Console.WriteLine("Name is: " + student.StudentName);

}

Student st = new Student();

st.StudentName = "Mubashar Rafique";

retorno = client.CreateStudent(st);

Console.WriteLine(retorno);

var std = client.GetStudent(1);

if (std != null)

{

std.StudentName = "Updated Name";

retorno = client.UpdateStudent(std);

Console.WriteLine(retorno);

retorno = client.DeleteStudent(3);

Console.WriteLine(retorno);

}

};

Faça os teste de operações na URL também:

Na barra de endereços do seu navegador da Web, digite a seguinte URL:

http://localhost:12345/northwind.svc/Customers

Isso retorna um conjunto de todos os clientes no banco de dados de exemplo Northwind.

Na barra de endereços do seu navegador da Web, digite a seguinte URL:

http://localhost:12345/northwind.svc/Customers('ALFKI')

Isso retorna uma instância de entidade para o cliente específico, ALFKI.

Na barra de endereços do seu navegador da Web, digite a seguinte URL:

http://localhost:12345/northwind.svc/Customers('ALFKI')/Orders

Isso percorre a relação entre clientes e pedidos para retornar um conjunto de todos os pedidos para o cliente específico ALFKI.

Na barra de endereços do seu navegador da Web, digite a seguinte URL:

http://localhost:12345/northwind.svc/Customers('ALFKI')/Orders?$filter=OrderID eq 10643

Isso filtra os pedidos que pertencem ao cliente específico ALFKI de modo que somente uma ordem específica seja retornada com base no valor OrderID fornecido.

Solicitar dados como JSON em um aplicativo cliente

Por padrão, os dados retornados de um Serviço de Dados do WCF estão no formato XML. No entanto, você pode enviar o cabeçalho "Accept: application/json;odata=verbose" na solicitação HTTP para retornar os dados no formato JSON. O código a seguir cria uma solicitação usando o objeto WebRequest no espaço para nome System.Net:

HttpWebRequest req = (HttpWebRequest)WebRequest.Create(@"http://localhost:8999/NorthwindsService.svc

Categories(1)?$select = CategoryID, CategoryName, Description");

req.Accept = "application / json; odata = verbose";

using (HttpWebResponse resp = (HttpWebResponse)req.GetResponse())

{

Stream s = resp.GetResponseStream();

StreamReader readStream = new StreamReader(s);

string jsonString = readStream.ReadToEnd();

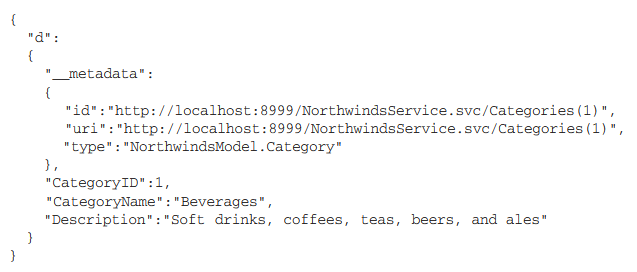
Debug.WriteLine(jsonString);

resp.Close();

readStream.Close();

}

Esse código cria uma solicitação que seleciona o registro de categoria com a chave primária 1 e seleciona as propriedades CategoryId, CategoryName e Description. A linha req.Accept = "application / json; odata = verbose" informa ao WCF Data Service para retornar os dados como JSON. Quando você executa esse código, ele imprime o seguinte na janela Saída. Quebras de linha são adicionadas para facilitar a leitura.



Usando LINQ

* Consultar dados usando operadores, incluindo projetar, juntar, agrupar, obter, ignorar, agregar; criar consultas sintaxe LINQ Method e sintaxe LINQ Query; selecionar dados usando tipos anônimos {select new}; forçar a execução de uma consulta; ler, filtrar, criar e modificar as estruturas de dados de usando LINQ to XML.

No .NET 3.5, um novo recurso foi adicionado ao C #: LINQ (Language Integrated Query). O LINQ é uma maneira de consultar diferentes tipos de fontes de dados que oferecem suporte a IEnumerable <T> ou IQueryable <T>. Ele oferece uma maneira fácil e elegante de acessar ou manipular dados de um objeto de banco de dados, documento XML e objetos na memória.

O LINQ geralmente é mais importante que outras estruturas de consulta devido à sua maneira de trabalhar com diferentes fontes de dados. As consultas geralmente são expressas em uma linguagem de consulta especializada. Diferentes idiomas foram desenvolvidos ao longo do tempo para os vários tipos de fontes de dados, por exemplo, SQL para bancos de dados relacionais e XQuery para XML. Portanto, os desenvolvedores tiveram que aprender uma nova linguagem de consulta para cada tipo de fonte ou formato de dados que eles devem suportar. O LINQ simplifica essa situação, oferecendo um modelo consistente para trabalhar com dados em vários tipos de fontes e formatos de dados. Em uma consulta LINQ, você está sempre trabalhando com objetos.

Você usa os mesmos padrões básicos de codificação para consultar e transformar dados em documentos XML, bancos de dados SQL, conjuntos de dados ADO.NET, coleções .NET e qualquer outro formato para o qual um provedor LINQ esteja disponível.

Tipos de LINQ

O LINQ opera com uma fonte de dados diferente e, devido ao trabalho com essas fontes de dados, é classificado nos seguintes tipos:

|  |  |
| --- | --- |
| **Tipo** | **Descrição** |
| LINQ to Object | O LINQ to Object fornece suporte para interação com objetos .NET na memória implementados por uma interface IEnumerable <T>. Usaremos o LINQ para objetar a explicação das consultas do LINQ. |
| LINQ para Entities | O LINQ to Entities fornece suporte para interação com um banco de dados relacional usando um ADO.NET Entity Framework. É mais flexível que o LINQ to SQL, mas complexo. Facilita diferentes provedores de dados, como Oracle, My SQL, MS SQL, etc. |
| LINQ para Dataset | O LINQ to Dataset fornece suporte para interação com um cache de dados na memória de forma fácil e maneira mais rápida. |
| LINQ para SQL | O LINQ to SQL, também conhecido como DLINQ, fornece suporte para interação com um banco de dados de relações como objetos. |
| LINQ para XML | O LINQ to XML, também conhecido como XLINQ, fornece suporte para interação com documentos XML, ou seja, para carregar documentos XML e para executar consultas como ler, filtrar, modificar, adicionar nó etc. nos dados XML. |
| Parallel LINQ | O LINQ paralelo, também conhecido como PLINQ, fornece suporte para o trabalho paralelo do LINQ. |

Usaremos o LINQ to Object para elaborar o tópico "Trabalhando com consultas LINQ" e para a elaboração explícita de "LINQ to XML" como um tópico.

SINTAXE DO LINQ

O LINQ fornece duas maneiras de interagir com fontes de dados para consultá-las:

1. LINQ Method
2. LINQ Query

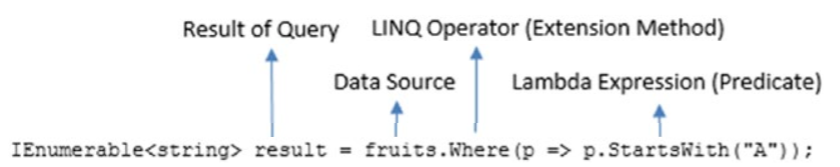
Essas duas sintaxes são semanticamente idênticas, isso facilita os desenvolvedores de SQL fornecendo a sintaxe LINQ Query e também facilita os desenvolvedores de C# oferecendo a sintaxe LINQ Method para consultar os dados,

Sintaxe do método

Sintaxe LINQ Method

O LINQ fornece a sintaxe do método para interagir com diferentes fontes de dados para consultá-las. É conhecido também como:

* Fluent Syntax, pois permite chamar uma série de métodos de extensão.
* Method Extension Syntax, pois usa, basicamente, métodos de extensão para acessar os dados.
* Lambda Syntax Query, pois método de extensão usa a sintaxe lambda como predicado.



Onde resultado deve ser de um tipo de dados retornados. Você também pode usar o tipo var quando não tiver certeza sobre o tipo de dados retornado. Vamos dar um exemplo de uma coleção de frutas como:

string[] fruits = newstring[]

{

"Apple","Mango","Strawberry","Date",

"Banana","Avocado","Cherry","Grape",

"Guava","Melon","Orange","Tomato"

};

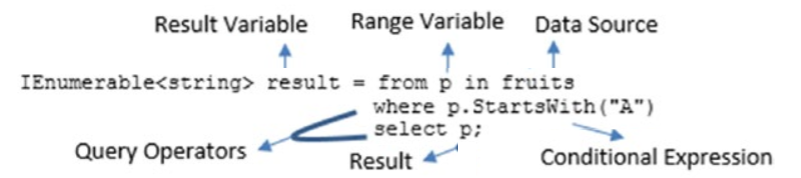
int fruitsLength = fruits.Where(p => p.StartsWith("A")).Count();

Console.WriteLine(fruitsLength); //2

Como a consulta retornará todas as frutas começando com "A", seria uma coleção de frutas, e as variáveis que recebem essas coleções devem ser do mesmo tipo que o tipo da coleção, que é uma coleção de seqüências de caracteres. Também podemos aplicar outro operador (método de extensão) Count() na mesma consulta para contar o número de frutas cujos nomes começaram com "A".

Sintaxe LINQ Query

O LINQ fornece outra maneira de executar uma consulta em diferentes fontes de dados, que é a sintaxe de consulta. É o mesmo que usar SQL para banco de dados relacional. Também é conhecido como Query Comprehension ou Query Expression. É o mesmo que consulta no SQL, com pouca diferença. A consulta nesta sintaxe sempre termina com um operador select ou group..by e começa com uma palavra-chave from.



Assim como fizemos na sintaxe do método para aplicar o operador (método de extensão) ainda mais para filtrar a consulta, também podemos fazer o mesmo nesse tipo de sintaxe. Tomando o mesmo exemplo de contagem do número de frutas cujo nome começa com "A", a consulta seria como:

int result = (from p in fruits

where p.StartsWith("A")

select p).Count();

Console.WriteLine(result); //2

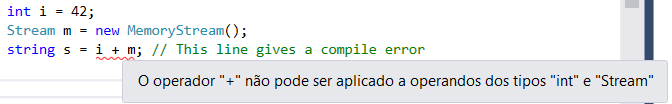
RECURSOS DE C# PARA SUPORTAR LINQ

Alguns recursos são adicionados ao C# que oferecem suporte ao LINQ. Alguns desses recursos de linguagem ajudam a criar consultas de uma maneira agradável e elegante, mas outros recursos às vezes são obrigatórios para que você possa criar uma consulta. Essas construções de linguagem são as seguintes:

1. **Variáveis ​​de tipo implícito**
2. **Inicializadores de Objetos**
3. **Expressões lambda**
4. **Métodos de extensão**
5. **Tipos anônimos**

Variáveis ​​de tipo implícito

Ao trabalhar com C#, na maioria das vezes você usa o sistema de tipos estáticos de C#. Isso significa que o compilador conhece o tipo de uma determinada variável e que vê se você a usa da maneira correta. Por exemplo, o código a seguir leva a um erro de compilação:



O que você está vendo aqui é chamado de digitação explícita. Você diz explicitamente ao compilador qual é o tipo de cada variável. No C# 3, um novo recurso foi adicionado: digitação implícita. Ao usar a digitação implícita, o compilador infere o tipo de variável para você e depois tipifica fortemente sua variável para esse tipo. Variáveis ​​implicitamente digitadas podem ser usadas apenas para variáveis ​​locais. Você pode usar a digitação implícita usando a palavra-chave var:

var it = 42;

var mt = new MemoryStream();

Nesse código, o compilador deduz para que o tipo de i é int e m é MemoryStream. Os tipos ainda são fortemente digitados, mas você não especifica o tipo.

Às vezes, ao usar uma consulta LINQ, o tipo de retorno é determinado no momento da compilação, para que você não possa especificar o retorno explicitamente porque não o conhece. Além de ser obrigatória nessas situações, a digitação implícita também pode melhorar a legibilidade do seu código. Se um usuário do seu código puder ver facilmente o tipo da sua variável, a digitação implícita poderá ser usada para evitar repetições. No caso a seguir, var pode melhorar a legibilidade:

Dictionary<string, IEnumerable<Tuple<Type, int>>> data = new Dictionary<string, IEnumerable<Tuple<Type, int>>>();

var implicitData = new Dictionary<string, IEnumerable<Tuple<Type, int>>>();

Nesse caso, você evita a repetição da declaração de tipo grande. Um leitor do seu código verá imediatamente que o tipo de implicitData é um dicionário. Quando o tipo da sua variável não é claro, é melhor evitar a digitação implícita:

var whatsMyType = GetData();

Nesse caso, um leitor do seu código teria que inspecionar o método GetData ou usar o IntelliSense no Visual Studio para ver qual é o tipo do whatsMyType.

Inicializadores de Objetos

Antes de o C# adicionar inicializadores de objetos, era necessário dividir a criação de um novo objeto e a configuração de suas propriedades, conforme mostrado abaixo:

publicclassPerson

{

publicstring FirstName { get; set; }

publicstring LastName { get; set; }

}

Person p = new Person();

p.FirstName = "John";

p.LastName = "Doe";

A nova sintaxe de inicialização do objeto permite combinar a criação de um novo objeto e a configuração de suas propriedades em uma instrução.

Person p2 = new Person

{

FirstName = "John",

LastName = "Doe"

};

Embora não seja estritamente necessário, ele pode melhorar a legibilidade do seu código quando você usa inicializadores de objetos (consulte a Lista 4-50). A mesma sintaxe pode ser usada ao criar coleções.

var people = new List<Person>{

new Person { FirstName = "João", LastName = "Corça"},

new Person { FirstName = "Jane", LastName = "Doe"}};

Na maioria das vezes, a sintaxe de inicialização de objeções é um bom açúcar sintático, mas, ao trabalhar com tipos anônimos, é realmente necessário. Usando a sintaxe de inicialização do objeto, você define as propriedades que um tipo anônimo possui.

Expressões lambda

Para entender o que é uma expressão lambda, é importante que você primeiro saiba o que é um método anônimo. Métodos anônimos foram introduzidos no C# 2.0 para permitir que você crie um método embutido em algum código, atribua-o a uma variável e repasse-o.

Func<int, int> myDelegate = delegate (int x) { return x \* 2; };

Console.WriteLine(myDelegate(21)); // Displays 42

O Func<T, T> é um dos tipos internos do .NET Framework. É uma notação abreviada para um delegado que recebe um int e retorna um int. Você também pode usar Action<...> para especificar um delegado que não possui um valor de retorno.Lambdas introduz uma notação abreviada para criar funções anônimas. O mesmo código pode ser escrito usando um lambda:

Func<int, int> myDelegate2 = x => x \* 2;

Console.WriteLine(myDelegate(21)); // Displays 42

Como você pode ver, o resultado é o mesmo, mas a notação é muito menor. Os lambdas usam a notação especial =>, que você pode ler como "se torna" ou "para qual". Ao trabalhar com, é definido em uma classe estática e ele usa a palavra-chave especial this para marcar-se como um método de extensão.

publicstaticclassIntExtensions

{

publicstaticint Multiply(thisint x, int y)

{

return x \* y;

}

}

int z = 2;

Console.WriteLine(z.Multiply(21)); // Displays 42

O LINQ é inteiramente baseado em métodos de extensão. Eles são definidos na classe System.Linq.Enumerables e permitem chamar funções LINQ em todos os tipos enumeráveis.

Tipos anônimos

Ao criar tipos anônimos, você usa inicializadores de objetos e digitação implícita. Um tipo anônimo é aquele que é modelado em tempo de compilação sem ter uma definição formal de classe.O compilador ajuda você a criar um tipo para você e fornece uma implementação padrão, substituindo os membros virtuais do System.Object.

Você cria um tipo anônimo usando a palavra-chave var como o tipo e usando o novo operador sem especificar um tipo, conforme mostrado abaixo.

var person = new

{

FirstName = "John",

LastName = "Doe"

};

Console.WriteLine(person.GetType().Name); // Displays “<>f\_\_AnonymousType0`2”

Aqui você vê como os inicializadores de objetos podem ser usados ​​para definir as propriedades de um tipo anônimo. Tipos anônimos são usados ​​no LINQ quando você cria uma chamada projeção. Isso significa que você seleciona certas propriedades de uma consulta e forma um tipo específico para ela.

USANDO CONSULTAS LINQ

Ao trabalhar com dados, seja na memória, de um banco de dados, um arquivo XML ou outro armazenamento, suas consultas sempre têm as três etapas a seguir:

1. **Obtenha os dados:**A fonte de dados pode ser diferente, ou seja, objetos na memória, objetos de banco de dados ou dados XML
2. **Crie uma consulta:** A consulta informa as informações para recuperar ou processar o que for necessário para processar a partir da fonte de dados. Diferentes tipos de operadores LINQ podem ser executados na consulta para filtrar, classificar, agrupar e modelar dados antes que eles retornem.
3. **Execute a consulta:** É importante saber que sempre que uma consulta é gravada, a execução da consulta não é feita. A execução da consulta varia dependendo da sua escolha. Por padrão, a execução da consulta é adiada até que você itere sobre a variável de consulta, mas você pode forçá-la a executar no momento de sua criação.

O exemplo a seguir mostra um exemplo de uma consulta LINQ simples que seleciona alguns números de uma matriz.

//1- First Step (Obtaining the Data Source)

int[] data = { 1, 2, 5, 8, 11 };

//2- Second Step (Creation of Query)

var result = from d in data

where d % 2 == 0

select d;

//3-Third Step (Execution of Query)

foreach (int i in result)

{

Console.WriteLine(i); // Displays 2 8

}

Como você pode ver, a consulta usa uma sintaxe especial: sintaxe da consulta. O CLR subjacente ao .NET Framework não tem noção do que fazer com uma sintaxe da consulta. O compilador converte a sintaxe da consulta em um conjunto de chamadas de método. A consulta anterior também pode ser escrita na sintaxe do método:

int[] data = { 1, 2, 5, 8, 11 };

var result2 = data.Where(d => d % 2 == 0);

Obviamente, não há método Where em uma matriz de números inteiros. É aqui que os métodos de extensão são usados. Os operadores LINQ são métodos de extensão em IEnumerable <T>. Você também vê como um lambda é usado como argumento para o método Where. Nesse caso, Where espera um argumento do tipo Func <int, bool>, o que significa que ele espera um bool para cada dado int.

Você pode escolher se deseja usar o método ou a sintaxe de consulta. Frequentemente, para consultas menores, a sintaxe da consulta é mais fácil de ler. No entanto, nem todos os operadores LINQ são suportados na sintaxe de consulta; portanto, às vezes você é forçado a usar a sintaxe baseada em método. Você também pode misturar as duas abordagens. O compilador sempre transforma sua sintaxe de consulta em sintaxe de método.

Execução diferida

A execução de uma consulta quando ela é gravada é adiada por padrão e você não pode obter o resultado de uma consulta até iterar sobre a variável de consulta ou executar métodos agregados (Max (), Min () e etc) ou métodos de extensão (ToList ( ), ToArray () e etc) para obter o resultado. Esse conceito é chamado de execução adiada da consulta. O exemplo abaixo mostra esse conceito:

List<int>def\_data = new List<int>(){ 1, 2, 5, 8, 11 };

var query = from p in def\_data

select p;

int count = 0;

count = query.Count();

Console.WriteLine(count); //Counts 5 records

def\_data.Add(12);

count = query.Count();

Console.WriteLine(count); //Count 6 records

//System.Linq.Enumerable+WhereSelectListIterator`2[System.Int32,System.Int32]

Console.WriteLine(query);

O código está apenas contando o número de registros. Depois que uma consulta é gravada, uma operação de Count() é executada para obter algum resultado e, nesse momento, retornará o número de registros, mas não no momento em que a consulta foi gravada. Como há adição em uma fonte de dados após a gravação de uma consulta, ela não deveria ter sido adicionada na variável count; mas isso não acontece na execução adiada, pois você não obterá resultados até que algum tipo de operação seja executada. Então, novamente, após a adição de um novo elemento na fonte de dados, quando Count () é chamado, ele obtém o resultado mais recente.

Execução imediata

Execução imediata de uma consulta é a execução no momento em que uma consulta é gravada. Força a consulta LINQ a ser executada e retorna os resultados imediatamente. Ao executar métodos/métodos agregados ou chamar ToList<T> ou ToArray<T> (métodos de extensão) em uma consulta, você pode forçá-lo a executar imediatamente. Execução imediata retorna os dados mais recentes (resultado). O código abaixo mostra este conceito:

List<int> imed\_data = new List<int>() { 1, 2, 5, 8, 11 };

var imed\_query = (from p in imed\_data

select p).ToList();

int imed\_count = 0;

imed\_count = imed\_query.Count();

Console.WriteLine(imed\_count); //Counts 5 records

imed\_data.Add(12);

imed\_count = imed\_query.Count();

Console.WriteLine(imed\_count); //Count 5 records

//System.Collections.Generic.List`1[System.Int32]

Console.WriteLine(imed\_query);

foreach (var item in imed\_query)

{

Console.WriteLine(item); // 1, 2, 5, 8, 11

}

Esse código não exibirá o últimoitem adicionado, pois há uma execução imediata da consulta executando o método de extensão (ToList ()) nela e, nesse momento, a consulta escrita realizada na variável imed\_query continha apenas os registros originais.

Observe que a execução do operador de agrupamento GroupBy () é adiada, enquanto a execução de outro operador de agrupamento ToLookup () é imediata

OPERADORES LINQPADRÃO

O LINQ possui alguns operadores de consulta padrão que você pode usar ao trabalhar com seus dados. Um provedor LINQ mapeia sua consulta para um armazenamento de dados específico, como LINQ to XML, LINQ to Entidades ou LINQ to Objects. Os operadores LINQ são na verdade um conjunto de métodos de extensão.Todo provedor LINQ é incentivado a implementar os operadores de consulta padrão para que você possa sempre usá-los. Isso significa que você pode usar esses operadores padrão em quase todas as fontes de dados, proporcionando uma experiência consistente.

Os operadores de consulta padrão para a sintaxe LINQ Query são: select, orderby, where, join e group by. Já para a sintaxe LINQ Method temos: All, Any, Average, Cast, Count, Distinct, GroupBy, Join, Max, Min, OrderBy, OrderByDescending, Select, SelectMany, Skip, SkipWhile, Sum, Take, Take-While, ThenBy, ThenByDescending, Where entre outros.Esses operadores oferecem flexibilidade para consultar dados, como filtragem de dados, classificação etc. Os operadores principais podem ser resumidos nas seguintes categorias:

|  |  |  |
| --- | --- | --- |
| **Operador** | **Descrição** | **Sintaxe** |
| Projeção | É usado quando um objeto é transformado em um novo formulário com base em alguma condição ou não. Selecione um resultado obtido de uma fonte de dados | Select |
| SelectMany |
| Filtragem | É usado para filtrar uma coleção ou sequência de dados com base no predicado ou em alguma condição específica. | Where |
| Ordenação | É usado para classificar os elementos de uma sequência em ordem crescente de acordo com uma chave. | Orderby, OrderByDescending |
| ThenBy, ThenByDescending |
| Junção | É usado para juntar duas ou mais seqüências ou coleções com base em alguma chave e produzir um | join..in..on.equals |
| Agrupamento | É usado para organizar elementos com base em uma determinada chave. Retorne uma sequência de itens em grupos como um IGroup <chave, elemento>. GroupBy e ToLookup são operadores de Agrupamento e são suportados pela sintaxe Query e sintaxe Method, exceto ToLookup(), que é suportado apenas na sintaxe do Method. | group.....by |
| group...by..into |
| Partição | É usado para dividir a coleção ou sequência em duas partes e retornar a parte restante (registro) deixada pela implicação desses operadores de partição. | Skip<T>(<count>) |
| Take<T>(<count>) |
| Agregação | Agregação significa aplicar funções agregadas no LINQ. Função agregada é uma função que calcula uma consulta e retorna um único valor. | group.....by |
| Faça a média de uma coleção numérica. | Average<T>(<param>) |
| Conte o número de elementos em uma coleção. | Count<T>(<param>) |
| Retorne o maior valor da coleção de valores numéricos. | Max<T>(<param>) |
| Retorne o menor valor da coleção de valores numéricos. | Min<T>(<param>) |
| Calcule a soma dos valores numéricos em uma coleção. | Sum<T>(<param> |

Quando você está trabalhando com alguns tipos mais complexos, o LINQ mostra seu poder. Para nossos testes dos operadores iremos deixar criado uma classe Order que possui alguns objetos OrderLine que apontam para um Produto. E iremos deixar preenchido com alguns itens.

publicclassProduct

{

publicstring Description { get; set; }

publicdecimal Price { get; set; }

}

publicclassOrderLine

{

publicint Amount { get; set; }

public Product Product { get; set; }

}

publicclassOrder

{

publicstring VendorName { get; set; }

public List<OrderLine> OrderLines { get; set; }

}

var orders = new List<Order>()

{

new Order()

{

VendorName = "João",

OrderLines = new List<OrderLine>()

{

new OrderLine(){ Amount= 3, Product = new Product(){ Description="Coca", Price=2 } },

new OrderLine(){ Amount= 42, Product = new Product(){ Description="Fanta", Price=15 } },

new OrderLine(){ Amount= 91, Product = new Product(){ Description="Bife", Price=54 } },

}

},

new Order()

{

VendorName = "Marcos",

OrderLines = new List<OrderLine>()

{

new OrderLine(){ Amount= 3, Product = new Product(){ Description="Coca", Price=2 } },

new OrderLine(){ Amount= 44, Product = new Product(){ Description="Lanche", Price=8 } },

new OrderLine(){ Amount= 91, Product = new Product(){ Description="Bife", Price=54 } },

}

},

};

Operador Projeção

O operador de projeção é usado para projetar uma fonte ou um elemento que não seja uma fonte com base na função de transformação. Existem basicamente dois operadores de projeção:Selecione e SelectMany.

int[] data = { 1, 2, 5, 8, 11 };

var result2 = from d in data select d;

Console.WriteLine(string.Join(", ", result2)); // Displays 1, 2, 5, 8, 11

Nesse caso, a consulta não está fazendo nada de especial. Você poderia ter usado a matriz de dados diretamente e produziria os mesmos resultados. No entanto, esta consulta mostra como o LINQ funciona. Primeiro, você começa com uma instrução from e depois uma fonte de dados. Isso ajuda o compilador, porque agora ele sabe imediatamente com o que está trabalhando. Por isso, ele pode fornecer o IntelliSense na próxima linha. Isso é diferente do que você está acostumado no SQL regular, no qual você começa com uma cláusula SELECT e, em seguida, uma cláusula FROM.

Você também pode combinar dados de várias fontes. Digamos que você tenha duas matrizes, ambas contendo números, e que deseja multiplicá-las. Na sua consulta LINQ, você usa várias instruções from para combinar os dados, conforme mostrado abaixo.

int[] data1 = { 1, 2, 5 };

int[] data2 = { 2, 4, 6 };

var result5 = from d1 in data1

from d2 in data2

select d1 \* d2;

Console.WriteLine(string.Join(", ", result5)); // Displays 2, 4, 6, 4, 8, 12, 10, 20, 30

Agora utilizando nossas classes, podemos ver melhor a diferença entre um Select e um SelectMany.Ao usar a projeção, você seleciona outro tipo ou um tipo anônimo como resultado da sua consulta. Você projeta seus resultados nele para se concentrar apenas nas propriedades realmente necessárias, com o Select teríamos:

IEnumerable<string> vendors = from p in orders

select p.VendorName;

foreach (var name in vendors)

{

Console.WriteLine(name);

}

Já com SelectMany seria:

var pedidos = (from p in orders

selectnew

{

VendorName = p.VendorName,

countorders = p.OrderLines.Count,

produto = p.OrderLines.FirstOrDefault().Product.Description,

preco = p.OrderLines.FirstOrDefault().Product.Price

});

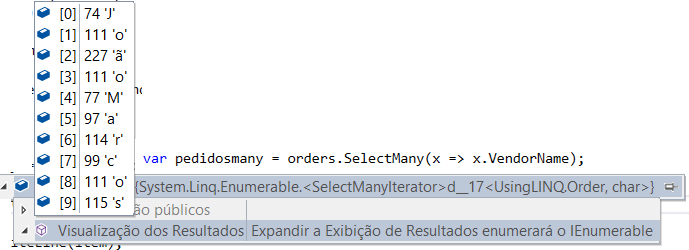
foreach (var item in pedidos)

{

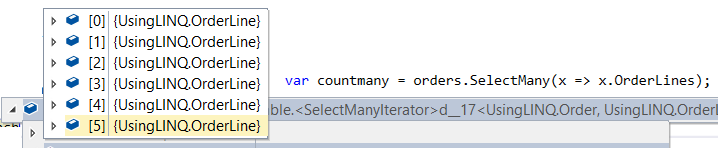
Console.WriteLine(item.VendorName + "\t" + item.produto + "\t" + item.countorders);

}

A consulta SelectMany inclui várias propriedades que não são definidas em nenhuma classe e podem recuperar o resultado de uma consulta acessando essas propriedades do tipo anônimo. Esse tipo de consulta é chamado Consulta de Tipo Anônimo.Um tipo anônimo é um objeto com propriedades somente leitura que não é declarado explicitamente. Se chamarmos o SelectMany pra o campo VendorName ele fragmenta a string em uma cadeia de caracteres:



E se quisermos listar todos os objetos filhos OrderLines dos objetos pai Order:



Mas tentar especificar uma propriedade ou um retorno único, retorna um erro de compilação:



Operador Filtragem

Agora você pode tornar a consulta um pouco mais interessante adicionando um filtro. Isso é feito adicionando uma instrução where. Essa instrução deve retornar um valor booleano que representa se o valor deve ser incluído no resultado final. Isso é chamado de predicado. O predicado é a instrução de comparação que é executada em relação a cada elemento na sequência.Como você está escrevendo em C #, deve usar os operadores e (&&) e ou (||) ao fazer declarações complexas..

var result3 = from d in data where d > 5 select d;

Console.WriteLine(string.Join(", ", result3)); // Displays 8, 11

var result2 = data.Where(d => d % 2 == 0);

Console.WriteLine(string.Join(", ", result2)); // Displays 2, 8

Você pode ter várias cláusulas where na sua expressão de consulta. É o mesmo que ter várias expressões na cláusula where.

var result3 = from d in data

where d % 2 == 0

where d > 5 select d;

Console.WriteLine(string.Join(", ", result3)); // Displays 8

Você normalmente não usaria várias cláusulas where no seu código. Em vez disso, basta separar suas cláusulas pelo operador &&. No entanto, para o teste, você poderá ver uma pergunta usando essa sintaxe que é válida; portanto, você deve estar ciente de que várias cláusulas where são equivalentes ao uso do operador &&.

Lembre-se também de que você pode chamar uma função em sua instrução para tornar seu código mais legível. O exemplo de código a seguir chama um método chamado IsEvenAndGT5 e passa no elemento atual enquanto enumera através da matriz:

staticbool IsEvenAndGT5(int i) { return (i % 2 == 0 && i > 5); }

var evenNumbers = from i in data

where IsEvenAndGT5(i)

select i;

foreach (int i in evenNumbers)

{

Console.WriteLine(i);

}

O último ponto a ser observado em relação à cláusula where é que ela pode aparecer em qualquer lugar da expressão de consulta, desde que não seja a primeira ou a última cláusula.

MétodoDistinct

O método Distinct retorna a lista distinta de valores na sequência retornada. Isso é útil quando você deseja remover duplicatas de uma sequência. O código a seguir retorna a lista distinta de descrição de um produto:

var distinctArray = orders.SelectMany(x => x.OrderLines).Select(x => x.Product.Description).Distinct();

foreach (var i in distinctArray)

{

Console.WriteLine(i);

}

Poré, o método Distinct se comporta de maneira diferente quando o objeto subjacente é uma classe personalizada. Como no seguinte código, onde o Distinct não parece funcionar:

var distinctobj = orders.SelectMany(x => x.OrderLines).Distinct();

foreach (var i in distinctobj)

{

Console.WriteLine(i.Product.Description);

}

Isso acontece pois a classe OrderLines deve implementar a interface IEquatable <T>, porque o método Distinct usa o comparador de igualdade padrão ao determinar se dois objetos são equivalentes. A interface IEquatable <T> possui um método que você deve substituir, chamado Equals. Neste exemplo, as propriedades Id e Description estão sendo usadas para determinar a equivalência. Você também deve substituir o método GetHashCode e retornar o código de hash com base nas propriedades do seu objeto.

publicclassOrderLine : IEquatable<OrderLine>

{

publicint Amount { get; set; }

public Product Product { get; set; }

publicbool Equals(OrderLine other)

{

if (Product.Description == other.Product.Description)

returntrue;

elsereturnfalse;

}

publicoverridebool Equals(object obj)

{

OrderLine other = (OrderLine)obj;

returnthis.Equals(other);

}

publicoverrideint GetHashCode()

{

//custom implementation of hashcode

string hash = this.Product.Description + this.Amount;

return hash.GetHashCode();

}

publicstaticbooloperator ==(OrderLine orderline1, OrderLine orderline2)

{

if (((object)orderline1) == null || ((object)orderline2) == null)

return Object.Equals(orderline1, orderline2);

return orderline1.Equals(orderline2);

}

publicstaticbooloperator !=(OrderLine orderline1, OrderLine orderline2)

{

if (((object)orderline1) == null || ((object)orderline2) == null)

return !Object.Equals(orderline1, orderline2);

return !(orderline1.Equals(orderline2));

}

}

OrderLine orderline1 = new OrderLine() { Product = new Product { Description = "Coca" } };

if (distinctobj.Contains(orderline1)) // True

Console.WriteLine("The list already contains this orderline.");

OperadorOrdenação

Outro operador útil é orderby, que pode ser usado para classificar sua coleção com um valor específico. Você pode usar orderby para classificar seus dados em ordem crescente ou decrescente. O código abaixo mostra como classificar os dados em ordem decrescente.

int[] data = { 1, 2, 5, 8, 11 };

var result4 = from d in data

where d > 5

orderby d descending

select d;

Console.WriteLine(string.Join(", ", result4)); // Displays 11, 8

Você também pode solicitar por mais de uma propriedade, separando suas condições por vírgula. O exemplo a seguir usa uma classe que contém uma propriedade Product.Descriptione Amount. A consulta retorna os elementos classificados primeiro por Descriptione depois por Amount.

var suborders0 = orders.SelectMany(x => x.OrderLines);

var orderna\_que = from d in suborders0

orderby d.Product.Description descending, d.Amount ascending

select d;

var orderna\_met = suborders0.OrderByDescending(d => d.Product.Description).ThenBy(d => d.Amount);

foreach (var item in orderna\_que)

{

Console.WriteLine(string.Join(", ", string.Format("Prod.: {0} - Amo.:{1}", item.Product.Description, item.Amount)));

}

Operador Junção (Join)

Outro operador padrão é o operador de junção. A junção pode ser usada para combinar dados de duas ou mais fontes. Ao usar o operador join, você deve especificar a propriedade que precisa ser igual, como você pode ver abaixo.

//Operador Junção

var table1 = new Dictionary<int, string>();

table1.Add(1, "Coca");

table1.Add(2, "Batata");

table1.Add(3, "Arroz");

table1.Add(5, "Feijão");

var table2 = new Dictionary<int, string>();

table2.Add(1, "Coca");

table2.Add(8, "Batata");

table2.Add(9, null);

table2.Add(10, null);

var popularProducts = from t1 in table1

join t2 in table2 on t1.Value equals t2.Value

selectnew { t1Key = t1.Key, t1Value = t1.Value, t2Key = t2.Key, t2Value = t2.Value };

var popularMethod = table1.Join(table2, t1 => t1.Value, t2 => t2.Value, (t1, t2) =>

new { t1Key = t1.Key, t1Value = t1.Value, t2Key = t2.Key, t2Value = t2.Value });

foreach (var item in popularProducts)

{

Console.WriteLine("Join t1:" + item.t1Key + ", " + item.t1Value + " -t2: " + item.t2Key + ", " + item.t2Value);

}

Join t1:1, Coca -t2: 1, Coca

Join t1:2, Batata -t2: 8, Batata

A cláusula join usa a palavra-chave igual ao invés de =. Isso ocorre porque você pode ingressar em campos apenas com base na equivalência, ao contrário do SQL, onde você pode usar> ou <sinais. O uso da palavra-chave igual deve deixar mais claro que a operação é equivalente.Você pode ver uma pergunta no exame que exibe quatro cláusulas de associação diferentes e será perguntado qual é o correto. Lembre-se de que uma cláusula de junção deve usar a palavra-chave equals e não o operador =.

Outer Join

Agora, suponha que você precise executar uma junção externa. Uma junção externa seleciona todos os elementos de uma sequência, mesmo se não houver um elemento correspondente na segunda sequência. No SQL, isso é chamado de RIGHT OUTTER JOIN ou LEFT OUTTER JOIN. No SQL, se você quiser todas as linhas da tabela no lado direito da cláusula JOIN, use um RIGHT OUTER JOIN; se você quiser todas as linhas da tabela no lado esquerdo da junção, use LEFT OUTER JOIN. Este cenário ocorre frequentemente ao escrever consultas no banco de dados. Por exemplo, se você tiver uma tabela que contenha uma chave estrangeira que seja anulável e desejar unir na tabela com a chave primária, use uma cláusula OUTER JOIN para garantir a seleção de todos os registros, mesmo que a coluna seja NULL. Para realizar essa mesma funcionalidade em uma expressão de consulta, você precisa usar a palavra-chave JOIN no grupo e o método DefaultIfEmpty. Uma junção de grupo permite combinar duas seqüências em um terceiro objeto.

//Operador Junção - OUTER JOIN

var tab1totab2 = from t1 in table1

join t2 in table2

on t1.Value equals t2.Value into outerGroup

from t2 in outerGroup.DefaultIfEmpty()

selectnew { t1Key = t1.Key, t1Value = t1.Value, t2Key = t2.Key, t2Value = t2.Value };

foreach (var item in tab1totab2)

{

Console.WriteLine("Left Outer t1:" + item.t1Key + ", " + item.t1Value + " -t2: " + item.t2Key + ", " + item.t2Value);

}

Left Outer t1:1, Coca -t2: 1, Coca

Left Outer t1:2, Batata -t2: 8, Batata

Left Outer t1:3, Arroz -t2: 0,

Left Outer t1:5, Feijão -t2: 0,

As listas são unidas em um objeto chamado outerGroup. Isso é feito usando a palavra-chave into. Há também uma segunda cláusula from que pode criar uma nova instância de um objeto t2. Se fosse um objeto poderia ser estipulado os valores padrões dentro do parêntesesquando não houver uma correspondência encontrada, assim:

from t2 in outerGroup.DefaultIfEmpty(new State {StateId = 0, StateName = ""})

Se você usar uma cláusula into, não poderá mais fazer referência à variável declarada no lado direito da instrução on na sua cláusula select (Repare que a tabela 2 não possui mais os campos Key nos itens sem correspondencia na saída com a tabela t1). Em vez disso, você usa a variável usada para enumerar os valores da nova sequência, neste exemplo, que é a variável do item. A cláusula select precisa ser alterada para usar a variável do item em vez da variável para obter o nome do estado.

Para expressões de consulta, você pode executar apenas junções esquerdas, portanto a ordem das sequências é importante na cláusula from.

var tab2totab1 = from t2 in table2

join t1 in table1

on t2.Key equals t1.Key into outerGroup //Não deixa colocar on s.Key

from t1 in outerGroup.DefaultIfEmpty()//Não deixa colocar from t2

selectnew { t2Key = t2.Key, t2Value = t2.Value, t1Key = t1.Key, t1Value = t1.Value };

foreach (var item in tab2totab1)

{

Console.WriteLine("'Right' Outer t2:" + item.t2Key + ", " + item.t2Value + " -t2: " + item.t1Key + ", " + item.t1Value);

}

'Right' Outer t2:1, Coca -t1: 1, Coca

'Right' Outer t2:8, Batata -t1: 0,

'Right' Outer t2:9, -t1: 0,

'Right' Outer t2:10, -t1: 0,

Repare aqui que o join foi realizado em cima do Key e não do Value, ficando assim até o Segundo item “Batata”, sem correnpondência da tabela t1.

Examplos de como realizer o Outer Join na sintaxe LINQ Method seguem abaixo:

var tab1totab2\_met1 = table1.GroupJoin(table2, t1 => t1.Value, t2 => t2.Value, (t1, outerGroup) => outerGroup.

DefaultIfEmpty(new KeyValuePair<int, string> { }).

Select(t2 =>new { t1Key = t1.Key, t1Value = t1.Value, t2Key = t2.Key, t2Value = t2.Value })).SelectMany(output => output);

foreach (var item in tab1totab2\_met1)

{

Console.WriteLine("Left Outer\_met1 t1:" + item.t1Key + ", " + item.t1Value + " -t2: " + item.t2Key + ", " + item.t2Value);

}

var tab1totab2\_met2 = table1.GroupJoin(table2, t1 => t1.Value, t2 => t2.Value, (t1, outerGroup) => outerGroup.DefaultIfEmpty().

Select(t2 =>new { t1Key = t1.Key, t1Value = t1.Value, t2Key = t2.Key, t2Value = t2.Value })).SelectMany(output => output);

foreach (var item in tab1totab2\_met2)

{

Console.WriteLine("Left Outer\_met2 t1:" + item.t1Key + ", " + item.t1Value + " -t2: " + item.t2Key + ", " + item.t2Value);

}

Chaves compostas

Pode haver casos em que você precise executar sua associação em uma chave composta. Uma chave composta contém várias propriedades necessárias para a finalidade de uma junção. Para fazer isso, crie dois tipos anônimos com as mesmas propriedades e compare os tipos anônimos. A consulta a seguir une os objetos table1 e table2 usando suas propriedades Key e Value:

// Operador Junção - CHAVE COMPOSTA

varchavecomposta = from t1 in table1

join t2 in table2

onnew { City = t1.Key, State = t1.Value } equals

new { City = t2.Key, State = t2.Value }

selectnew { t1Key = t1.Key, t1Value = t1.Value, t2Key = t2.Key, t2Value = t2.Value };

foreach (var item inchavecomposta)

{

Console.WriteLine("Composta: t1:" + item.t1Key + ", " + item.t1Value + " -t2: " + item.t2Key + ", " + item.t2Value);

}

var chavecomposta\_met = table1.Join(table2, t1 =>new { City = t1.Key, State = t1.Value },

t2 =>new { City = t2.Key, State = t2.Value },

(t1, t2) =>new { t1Key = t1.Key, t1Value = t1.Value, t2Key = t2.Key, t2Value = t2.Value });

Composta: t1:1, Coca -t2: 1, Coca

Operador Agrupamento

Ao usar o agrupamento, você agrupa seus dados por uma determinada propriedade e trabalha com esse resultado. Um exemplo em que isso é útil é quando você deseja saber quantos itens de cada produto você vendeu.

var result6 = from o in orders

from l in o.OrderLines

group l by l.Product into p

selectnew

{

Product = p.Key,

Amount = p.Sum(x => x.Amount)

};

foreach (var i in result6.ToList())

{

Console.WriteLine(string.Format("Product: {0}; Amount: {1}", i.Product.Description, i.Amount));

}

Um exemplo de agrupamento utilizando a sintaxe do método:

var suborders = orders.SelectMany(x => x.OrderLines).GroupBy(x => x.Product.Description);

foreach (var product in suborders)

{

Console.WriteLine("Product:" + product.Key + " Vendas:" + product.Sum(x => x.Product.Price));

}

Se você precisar agrupar por mais de um campo, crie um tipo anônimo como parâmetro para o método GroupBy. Por exemplo, o código a seguir será agrupado pela cidade e pelas propriedades do estado:

var employeesByState = orders.SelectMany(x => x.OrderLines).GroupBy(e =>new { e.Product.Description, e.Product.Price });

Você pode adicionar lógica ao seu grupo por cláusula para agrupar por qualquer coisa. O exemplo a seguir agrupa números pares e ímpares e, em seguida, imprime a contagem e a soma de cada grupo na janela Saída:

int[] myArray = newint[10] { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

var groupedNumbers = from i in myArray

group i by (i % 2 == 0 ? "Even" : "Odd");

foreach (var groupNumber in groupedNumbers)

{

Console.WriteLine(groupNumber.Key + ": " + groupNumber.Sum());

Console.WriteLine(string.Join(", ", groupNumber));

}

No exemplo anterior, a cláusula group by contém uma declaração condicional que retorna a cadeia de caracteres "Even" ou "Odd" e agrupa o número adequadamente. O código anterior produz o seguinte resultado.

Você pode usar uma cláusula select ao agrupar sequências, mas deve incluir uma cláusula into na cláusula de grupo. Suponha que no último exemplo você desejasse selecionar a chave e a soma dos números pares ou ímpares na consulta. O código a seguir pode fazer isso:

var selectgrouped = from i in myArray

group i by (i % 2 == 0 ? "Even" : "Odd") into grupo

selectnew { Key = grupo.Key, Soma = grupo.Sum(), Numbers = grupo };

foreach (var groupNumber in selectgrouped)

{

Console.WriteLine(groupNumber.Key + ": " + groupNumber.Soma);

Console.WriteLine(string.Join(", ", groupNumber.Numbers));

}

A variável grupo é do tipo IGrouping <TKey, TElement> e você pode usá-lo na sua cláusula select. A cláusula select cria um tipo anônimo com duas propriedades: Key,Soma e Numbers.

OperadorPartição

Quando você está trabalhando com um grande conjunto de dados, provavelmente deseja implementar a paginação. Ao usar a paginação, você não mostra todos os dados de uma vez para o usuário. Em vez disso, você carrega uma página de cada vez. Quando os dados vêm de um recurso externo, como um banco de dados, isso pode gerar um ganho de desempenho significativo. Para implementar a paginação, você pode usar os operadores Skip e Take.

var pageIndex = 2;

var pageSize = 10;

var pagedOrders = orders.Skip((pageIndex - 1) \* pageSize).Take(pageSize);

Existem duas outras funções que permitem encontrar o primeiro ou o último elemento na sua sequência. Eles também estão disponíveis apenas como métodos, mas podem ser usados por uma expressão de consulta. Essas funções podem ser úteis quando você deseja encontrar o primeiro ou o último elemento em uma sequência que atenda a uma condição específica, como o primeiro número par em uma matriz. A sintaxe do método Primeiro e Último é mostrada nos exemplos a seguir

var first = orders.Where(i => i.VendorName == "João").First();

var last = orders.Where(i => i.VendorName == "João").Last();

Operador Agregação

Agora, digamos que você queira saber o número médio, mínimo e máximo de OrderLines para um conjunto de pedidos. Você pode usar uma consulta LINQ para calcular facilmente esse valor:

var average = orders.Average(o => o.OrderLines.Count);

var sum = orders.Sum(o => o.OrderLines.Capacity);

var min = orders.Min(o => o.OrderLines.Count);

var max = orders.Max(o => o.OrderLines.Count);

Console.WriteLine(string.Format("Média: {0}; Min:{1}; Max: {2}", average, min, max));

Como você pode ver, não há sintaxe de consulta para o método Average, Min e Max, e é por isso que você precisa usar a sintaxe do método. Tente escrever esta consulta sem o LINQ. Não é difícil, mas definitivamente tem muito mais código.

Como o LINQ funciona

Digamos que você deseje implementar o operador de consulta Where sozinho. Se você olhar o código-fonte, verá que a definição do método Where é a seguinte:

publicstatic IEnumerable<TSource> Where(

this IEnumerable<TSource> source,

Func<TSource, bool> predicate)

Se você deseja criar sua própria implementação, é necessário criar um método de extensão com esta assinatura. Removendo a instrução using para System.Linq, você pode usar seu próprio método.Uma implementação básica para esse método é mostrada abaixo. Isso omite a verificação e o tratamento de erros.

publicstaticclassLinqExtensions

{

publicstatic IEnumerable<TSource> Where<TSource>(

this IEnumerable<TSource> source,

Func<TSource, bool> predicate)

{

foreach (TSource item in source)

{

if (predicate(item))

{

yieldreturn item;

}

}

}

}

A palavra-chave mágica neste exemplo de código é a utilização deyield no retorno. Como a declaração yield é uma implementação do padrão do iterador, o código não é executado até que a primeira chamada ao MoveNext seja feita. Isso é chamado de execução tardia. Uma consulta LINQ não é executada até que seja iterada, até esse momento a consulta não faz nada. Muitos erros ao trabalhar com consultas LINQ acontecem porque as pessoas esquecem quando a consulta é executada.

Isso é particularmente importante quando você está trabalhando com um dos outros provedores LINQ que trabalham com um banco de dados, como LINQ to Entities ou LINQ to SQL (provedores LINQ como LINQ to Entities analisam a consulta e a transformam em SQL). A consulta não será enviada ao banco de dado até que o resultado seja repetido. Isso também significa que a execução de uma consulta várias vezes atinge o banco de dados várias vezes. Por esse motivo, é melhor salvar os resultados de uma consulta em uma variável local e usá-la ao trabalhar com os dados.

A iteração pode ocorrer quando você chama um método como ToList ou quando itera os resultados em a para cada instrução (que chama o método MoveNext no iterador). Obviamente, outros métodos são mais complexos, mas todos seguem a mesma idéia.

USANDO LINQ TO XML

Um outro provedor que faz parte do .NET Framework é o LINQ to XML. Normalmente, você trabalha com arquivos XML usando as classes XmlWriter, XmlReader e XmlDocument. A vantagem do LINQ to XML é que você pode usar a mesma experiência de consulta usada no LINQ to Objects ou com outros provedores do LINQ.

O LINQ to XML ajuda a criar, editar e analisar arquivos XML. Se você apenas precisar obter algumas informações de um arquivo XML, o LINQ to XML oferecerá uma experiência de consulta fácil. Se você precisar de mais recursos, o LINQ o ajudará a escrever consultas poderosas que são mais compactas que outras classes XML.O namespaceSystem.Xml.Linq fornece as classes necessárias para interagir com dados/documento XML em C#. Algumas das classes são:

Consultando XML

Se você deseja consultar um arquivo XML com LINQ to XML, pode usar a classe XDocument para carregar um arquivo ou uma sequência que contém XML na memória. A classe XDocument trabalha com objetos do tipo XNode. XNode é uma classe abstrata que representa a ideia de algum segmento de conteúdo que um documento contém. Você pode usar XDocument.Nodes para acessar os nós que formam um arquivo XML ou pode usar XDocument.Descendants ou XDocument.Elements para procurar um conjunto específico de nós. Um resumo dos métodos de acesso aos nós e atributos se encontra na tabela abaixo:

|  |  |
| --- | --- |
| **Método** | **Objetivo** |
| Add | Adiciona um item no final da coleção filho do elemento. |
| AddAfterSelf | Adiciona um item à coleção filho do pai antes ouapós esse elemento. |
| AddBeforeSelf |
| AddFirst | Adiciona um item no início da coleção filho do elemento. |
| Ancestors | ​​Retorna uma coleção de objetos XElement que são ancestrais do elemento. Se você especificar um nome, o método retornará apenas elementos com esse nome. |
| Attribute | Retorna um atributo com um nome específico. |
| Attributes | Retorna uma coleção contendo os atributos desse elemento. Se você especificar um nome, a coleção incluirá apenas atributos com esse nome. |
| DescendantNodes | Retorna uma coleção de objetos XNode que são descendentes do elemento. |
| Descendants | Retorna uma coleção de objetos XElement que são descendentes do elemento. Se você especificar um nome, o método retornará apenas elementos com esse nome. |
| DescendantsAndSelf | Retorna uma coleção de objetos XElement que inclui esse elemento e seus descendentes. Se você especificar um nome, o método retornará apenas elementos com esse nome. |
| Element | Retorna o primeiro elemento filho com um nome especificado. |
| Elements | Retorna uma coleção contendo os filhos do elemento. Se você especificar um nome, o método retornará apenas elementos com esse nome. |
| ElementsAfterSelf | Retorna uma coleção que contém os irmãos do elemento que vêm antes ouapósesse elemento. Se você especificar um nome, o método retornará apenas esses elementos. |
| ElementsBeforeSelf |
| IsAfter | Retorna true se esse nó vierantes ou após outro nó especificado em um documento. |
| IsBefore |
| Load | Carrega o elemento de um nome de arquivo, fluxo ou leitor. |
| Nodes | Retorna uma coleção que contém os nós filhos deste elemento. |
| NodesAfterSelf | Retorna uma coleção que mantém os irmãos do nó que vêmantes ou após esse nó. |
| NodesBeforeSelf |
| Parse | Analisar Cria um XElement a partir de uma sequência XML. |
| Remove | Remove esse elemento de seu pai. |
| RemoveAll | Remove todos os nós e atributos desse elemento. |
| RemoveAttributes | Remove os atributos deste elemento. |
| RemoveNodes | Remove os nós filhos deste elemento. |
| ReplaceAll | Substitui os nós e atributos filhos do elemento pelosnovos especificados |
| ReplaceAttributes | Substitui os atributos do elemento pelos novos especificados. |
| ReplaceNodes | Substitui os nós filhos do elemento pelos novos especificados. |
| ReplaceWith | Substitui este nó pelo novo conteúdo especificado. |
| Save | Salva o elemento em um arquivo, Stream ou Writer. |
| SetAttributeValue | Define, adiciona ou remove um atributo. |
| SetElementValue | Define, adiciona ou remove um elemento filho. |
| SetValue | Define o valor do elemento. |
| ToString | Retorna o código XML recuado do elemento. |
| WriteTo | Grava o elemento em um xnawriter. |

Um dos recursos interessantes da classe XDocument é que ela representa o arquivo XML de maneira hierárquica. Por esse motivo, você pode passar de um nó para nós filhos e voltar para o nó pai. Atributos não são considerados nós; em vez disso, são pares de chave/valor que pertencem a um nó.

O LINQ to XML permite converter facilmente uma sequência em um documento XML. Para usar o LINQ to XML, você deve adicionar uma referência ao namespace System.Xml.Linq. Lembre-se, você pode usar o LINQ para consultar qualquer sequência, independentemente da fonte. Contanto que a sequência suporte a interface IEnumerable <T> ou IQueryable <T>, você pode usar uma expressão de consulta LINQ para converter a sequência em XML. Isso pode ser útil ao transferir dados entre dois sistemas. O exemplo a seguir converte uma lista de objetos Person em XML:

classPerson

{

publicint ID { get; set; }

publicstring firstName { get; set; }

publicstring lastName { get; set; }

publicstring emailaddress { get; set; }

publicstring phonenumber { get; set; }

}

List<Person> persons = new List<Person>()

{

new Person() { firstName = "John", lastName = "Smith", emailaddress = "john@unknown.com" },

new Person() { firstName = "Jane", lastName = "Doe", emailaddress = "jane@unknown.com" , phonenumber = "001122334455" },

};

var xmlpersons = new XElement("people",

from e in persons

selectnew XElement("person", new XElement("contactdetails", new XElement("emailaddress", e.emailaddress),

e.phonenumber == null ? null : new XElement("phonenumber", e.phonenumber)),

new XAttribute("firstName", e.firstName),

new XAttribute("lastName", e.lastName)

)

);

Console.WriteLine(xmlpersons);

O código acima cria uma saída em XML que contém um conjunto de pessoas que possuem alguns atributos e informações de contato, como abaixo:

<?xmlversion="1.0"encoding="utf-8" ?>

<people>

<personfirstName="John"lastName="Doe">

<contactdetails>

<emailaddress>john@unknown.com</emailaddress>

</contactdetails>

</person>

<personfirstName="Jane"lastName="Doe">

<contactdetails>

<emailaddress>jane@unknown.com</emailaddress>

<phonenumber>001122334455</phonenumber>

</contactdetails>

</person>

</people>

Você pode usar o LINQ to XML também para executar uma consulta que carrega os nomes das pessoas da cadeia de caracteres XML. O exemplo abaixo mostra como você pode usar o método Descendants e o método Attribute para carregar esses dados.

string projectDirectory = Directory.GetParent(workingDirectory).Parent.FullName;

string xmlString = System.IO.File.ReadAllText(projectDirectory + "\\XMLFile.xml");

XDocument doc = XDocument.Parse(xmlString);

IEnumerable<string> personNames = from p in doc.Descendants("person")

select (string)p.Attribute("firstName")

+ " " + (string)p.Attribute("lastName");

foreach (string s in personNames)

{

Console.WriteLine(s);

}

Uma coisa a observar é que o método Attribute retorna instâncias do XAttribute. O XAttribute possui uma propriedade Value do tipo string, mas também implementa operadores explícitos, para que você possa convertê-lo na maioria dos tipos básicos em C#.

Como o LINQ to XML oferece suporte aos operadores LINQ padrão, você pode usar facilmente operadores como Where e OrderBy em suas consultas XML. A Listagem 4-66 mostra como você pode filtrar todas as pessoas apenas para aquelas com um número de telefone e selecionar seus nomes completos em ordem alfabética. Você também vê com que facilidade é possível misturar sintaxe baseada em método e em consulta.

IEnumerable<string> personNames2 = from p in doc.Descendants("person")

where p.Descendants("phonenumber").Any()

let name = (string)p.Attribute("firstName")

+ " " + (string)p.Attribute("lastName")

orderby name

select name;

Criando XML

Além de consultar XML, o LINQ to XML também pode ajudá-lo a criar XML em uma sintaxe agradável e fluente. Você usa a classe XElement para criar seu próprio XML. Você pode usar o método Add para construir uma hierarquia XML ou o construtor XElement que utiliza uma matriz de objetos que formam o conteúdo. Essa sintaxe é agradável e elegante. A Listagem 4-67 mostra um exemplo de como criar algum XML.

XElement root = new XElement("Root",

new List<XElement>

{

new XElement("Child1"),

new XElement("Child2"),

new XElement("Child3")

},

new XAttribute("MyAttribute", 42));

root.Add(new XElement("Name", "Hamza Ali"));

root.Add(new XElement("Age", "21"));

root.Add(new XElement("Address", "Pakistan"));

root.Add(new XAttribute("AddAttribute", 3.21));

Console.WriteLine(root);

root.Save(projectDirectory + "\\test.xml");

O construtor do XElement está sobrecarregado. Leva o nome do elemento, seu valor, etc., e você pode ainda adicionar subelemento (conforme adicionado ao código) usando o objeto raiz ou elemento pai. O conteúdo do arquivo test.xml criado é:

<?xmlversion="1.0"encoding="utf-8"?>

<RootMyAttribute="42">

<Child1 />

<Child2 />

<Child3 />

</Root>

Usando inicializadores de objetos e inicializadores de coleção, você pode criar seus documentos XML de maneira fácil.

Atualizando XML

Quando você deseja modificar um trecho de XML, normalmente o carrega na memória, modifica o XML removendo e inserindo nós ou alterando o conteúdo dos nós existentes. Depois de terminar, salve o XML novamente no arquivo.

O LINQ to XML usa outra abordagem chamada construção funcional. A construção funcional trata a modificação de dados como um problema de transformação e não como uma manipulação detalhada de dados. A transformação pode consumir mais energia do processador, mas é mais fácil escrever e manter e, na maioria das situações, esses benefícios superam os custos.Digamos que você esteja trabalhando com o XML da Lista 4-64 e precise adicionar um número de telefone para Joe e um atributo IsMale a todas as pessoas.

Você pode usar código de procedimento para fazer isso, como mostra a Lista 4-68.

XElement root2 = XElement.Parse(xmlString);

foreach (XElement p in root2.Descendants("person"))

{

string name = (string)p.Attribute("firstName") + (string)p.Attribute("lastName");

p.Add(new XAttribute("IsMale", name.Contains("John")));

XElement contactDetails = p.Element("ContactDetails");

if (!contactDetails.Descendants("PhoneNumber").Any())

{

contactDetails.Add(new XElement("PhoneNumber", "001122334455"));

}

}

Você pode realizer a alteração partindo de um XDocument e listando os XElement para alterar somente o Valor de um Element. Pode utilizar a sintaxe LINQ Method.

var file = projectDirectory + "\\updatetest.xml";

string xmlData = System.IO.File.ReadAllText(file);

XDocument document = new XDocument();

document = XDocument.Parse(xmlData);

var contactDetails = document.Descendants("person")

.Where(e => e.Attribute("firstName").Value == "John")

.Select(e => e.Descendants("contactdetails")).FirstOrDefault();

foreach (XElement contacs in contactDetails)

{

if (contacs.Element("emailaddress") != null)

{

var mailnode = contacs.Element("emailaddress");

mailnode.Value = "john@unknown.com";

}

}

Você pode usar a construção funcional para transformar essa árvore em uma nova que inclua o novo atributo e elementos usando uma consulta LINQ to XML. A Listagem 4-69 mostra como você pode fazer isso.

XElement root3 = XElement.Parse(xmlString);

XElement newTree = new XElement("People",

from p in root3.Descendants("person")

let name = (string)p.Attribute("firstName") + (string)p.Attribute("lastName")

let contactDetails = p.Element("contactdetails")

selectnew XElement("person", new XAttribute("IsMale", name.Contains("John")),

p.Attributes(),

new XElement("contactdetails",

contactDetails.Element("emailaddress"),

contactDetails.Element("phonenumber") ?? new XElement("phonenumber", "112233455")

)));

Console.WriteLine(root3);

Depende da dificuldade da sua modificação, se você usa a maneira processual ou funcional regular. Especialmente quando a estrutura do documento XML muda, a maneira funcional pode trazer muitos benefícios.

Serializar e de-serializar dados

* Serializar e de-serializar dados usando serialização binária, serialização personalizada, Serializador XML, Serializador JSON e Serializador de Contrato de Dados

Serialização e Desserialização

Serialização é o processo de transformar um objeto em um formulário que pode ser persistido no armazenamento ou transferido de um domínio de aplicativo para outro. Serialização e desserialização de dados ou de um objeto são comumente usadas nos casos em que você frequentemente trocar dados com outros aplicativos, especioalmente, ao se comunicar com aplicativos remotos. Por exemplo, quando os dados serão enviados para um serviço web ou por um fluxo de rede, primeiro é necessário converter dados em um fluxo de bytes e, no lado do recebimento, você deve converta-o novamente de um fluxo de bytes para um objeto que é sua principal preocupação. Ou seja:

* **Serialização:** O processo de conversão de um objeto ou gráfico de objeto em um fluxo de bytes é chamado de serialização. É o processo de transformar um objeto em bytes ou texto para armazená-lo em qualquer tipo de armazenamento ou trocar o objeto pela rede.
* **Desserialização:** O processo de conversão de um fluxo inverso de bytes em um objeto ou gráfico de objeto é chamado desserialização

O objeto serializado carrega os dados de um objeto na forma de um fluxo, juntamente com as informações do tipo de objeto, ou seja, sua versão, cultura e nome do conjunto.Os métodos não são serializados porque a serialização apenas serializa os dados armazenados por um objeto.

O .NET Framework fornece classes para ajudá-lo a serializar e desserializar o objeto e também oferece maneiras de configurar seus próprios objetos. O C # fornece diferentestécnicas para executar processosde serialização e desserialização de dados.Por padrão, existem três mecanismos:

1. **BinaryFormatter**, é um serializador para serializar e desserializar os dados no formato binário.
2. **XmlSerializer,**é usado para serializar e desserializar o objeto em um documento XML. Este serializador permite controlar como os objetos são codificados em XML.
3. **DataContractSerializer,** também é usado para serializar o objeto em um fluxo XML usando um contrato de dados fornecido.

Também existem outros serializadores que são usados para serializar e desserializar os dados de acordo com seu uso, como:

1. **DataContractJsonSerializer**: serialize os objetos para a JavaScript Object Notation (JSON) e desserializar dados JSON para objetos.
2. **JavaScriptSerializer:** serialize e desserialize os objetos para o aplicativo habilitado para AJAX.

SERIALIZAÇÃO BINÁRIA

A serialização binária serializa um objeto ou dados ou gráfico de objeto em formato binário. A serialização binária usa codificação binária para serialização para produzir dados serializados compactos para usoscomo fluxos de rede de armazenamento ou baseado em soquete. Você também pode serializar dados que não são adequados para um formato XML, como uma imagem.

Um objeto esterilizado binário contém dados serializados, juntamente com as informações de tipo do objeto, incluindoversão, token público, cultura e nome do assembly. A serialização binária depende de uma plataforma .NET, ou seja, para trocar um objeto ou dados serializados binários de um aplicativo para outro aplicativo, e os dois aplicativos devem estar em uma plataforma .NET.

O serializador binário usa uma classe BinaryFormatter para implementar a serialização binária. É mais seguro que outras serializações. Para executar esse tipo de serialização, basta marcar um item com o SerializableAttribute. Depois disso, você precisa usar a instância do Binary Serializer para serializar o objeto ou gráfico de objetos. A seguir, os namespaces usados na serialização binária:

1. System.Runtime.Serialization
2. System.Runtime.Serialization.Formatters.Binary

Você pode serializar o objeto em um arquivo, memória ou banco de dados de acordo com sua necessidade.

O exemplo abaixo mostra como você pode configurar um objeto para serialização binária, serializá-lo em arquivo edepois desserialize-o em um objeto.Você pode usar o BinaryFormatter junto com um FileStream para ler e gravar seus objetos no disco. Lembre-se, o FileStream é um objeto usado para ler e gravar dados de e para o disco como matrizes de bytes.Para que uma classe seja serializada, você deve adicionar o atributo [Serializable] à parte superior da classe. O exemplo a seguir cria uma classe chamada Person e a torna serializável:

[Serializable]

publicclassTeacher

{

publicint ID { get; set; }

publicstring Name { get; set; }

publicdecimal Salary;

[NonSerialized]

privateint PrivateField;

publicvoid SetPrivate(int num) { PrivateField = num; }

}

classProgram

{

staticvoid Main(string[] args)

{

// Criou a instância e inicializou

Teacher professor = new Teacher()

{

ID = 1,

Name = "Raimundo Nonato",

Salary = 1000

};

professor.SetPrivate(666);

// Serializador binário

BinaryFormatter formatador = new BinaryFormatter();

//Sample.bin(Arquivo binário é criado) para armazenar dados serializados binários

using (FileStream file = new FileStream("Sample.bin", FileMode.Create))

{

// esta função serializa o "professor" (Objeto) em "arquivo" (Arquivo)

formatador.Serialize(file, professor);

}

Console.WriteLine("A serialização binária foi concluída com êxito!");

// Desserialização binária

using (FileStream file = new FileStream("Sample.bin", FileMode.Open))

{

Teacher dteacher = (Teacher)formatador.Deserialize(file);

}

Console.WriteLine("Desserialização binária concluída com êxito!");

Console.ReadKey();

}

}

Esteja ciente de que mesmo o campo privado será persistido no disco. Você pode restaurar o estado do objeto lendo o arquivo Person.bin e desserializando o arquivo.Se você executar o código e visualizar o objeto professor na janela Watch, poderá ver que todos os campos mantêm seu valor, até o campo PrivateField privado. Se você deseja impedir que o campo privado seja persistente, você pode adicionar o atributo [NonSerialized] antes da declaração do campo.

[Serializable]

publicclassTeacher

{

publicint ID { get; set; }

publicstring Name { get; set; }

publicdecimal Salary;

[NonSerialized]

privateint PrivateField;

publicvoid SetPrivate(int num) { PrivateField = num; }

}

A serialização binária é mais rígida que outras serializações. Quando o serializador binário não consegue encontrar umcampo, lança uma exceção. Você pode usar OptionalFieldAttribute para garantir que o serializador binário saiba que o campo foi adicionado em versões posteriores e que o objeto serializado atual não conterá esse campo.

No .Net 1.1, podemos definir o "BinaryFormatter.AssemblyFormat" como"FormatterAssemblyStyle.Simple" para permitir mais tolerância de versão. Dessa forma, nenhuma exceção será lançada se um campo específico não for encontrado no fluxo armazenado.

No .Net 2.0, a propriedade é por padrão "Simple", não "Full" (você podeverifique isso observando BinaryFormatter.AssemblyFormat no depurador), entãomesmo se não especificarmos "OptionalFieldAttribute", nenhuma exceção será lançada."OptionalFieldAttribute" é necessário apenas quando "FormatterAssemblyStyle.Full"  
precisa trabalhar com campos ausentes.

[Serializable]

publicclassTeacher

{

publicint ID { get; set; }

publicstring Name { get; set; }

publicstring Cargo { get; set; }

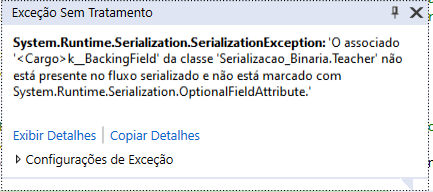
[NonSerialized]

publicdecimal Salary;

}

BinaryFormatter formatador = new BinaryFormatter();

formatador.AssemblyFormat = System.Runtime.Serialization.Formatters.FormatterAssemblyStyle.Full;



SERIALIZAÇÃO XML

A serialização XML serializa um objeto no formato XML que é legível tanto por humanos quanto por máquinas e é independente do ambiente em que é usado. Na serialização XML, apenas os campos ou propriedades públicos podem ser serializados. A serialização XML não converte métodos, indexadores, campos privados ou propriedades somente leitura(exceto coleções somente leitura). Para serializar todos os campos e propriedades de um objeto,público e privado, use o DataContractSerializer em vez da serialização XML.

Ao contrário da serialização binária, ele não inclui o tipo de um objeto serializadoem formação. Por exemplo, se você tiver um objeto serializado do tipo Professor, não há garantia de que eleseria desserializado em um objeto do tipo Professor. É por isso que a serialização XML não armazena as informações do tipo do objeto.

A serialização XML usa a classe XmlSerializer para implementar a serialização XML. XmlSerializer é menos rigoroso queBinarySerializer, mas não possui melhor desempenho. O XmlSerializer foi criado com a idéia das mensagens SOAP (Simple Object Access Protocol). SOAP é um protocolo para troca de informações com serviços da web. O XmlSerializer é fracamente acoplado aos seus objetos. Se você adicionar novas propriedades ou métodos aos seus objetos, o XmlSerializer não perceberá. Com algumas alterações simples na configuração, é possível mapear nós XML para propriedades em seus objetos para que ambos possam ser modificados independentemente.

Para executar a serialização XML, marque seu tipo com um atributo Serializable. A serialização XML pode ser feita sem especificar um atributo Serializable no tipo, mas é uma abordagem incorreta. É importante que você marque seus tipos com o atributo [Serializable] pois isso informa ao .NET Framework que seu tipo deve ser serializável. Ele verificará seu objeto e todos os objetos que ele referencia para garantir que ele possa serializar o gráfico inteiro. Se isso não for possível, você receberá uma exceção em tempo de execução.

[Serializable]

publicclassTeacher

{

publicint ID { get; set; }

publicstring Name { get; set; }

publiclong Salary { get; set; }

}

classProgram

{

staticvoid Main(string[] args)

{

// Criou a instância e inicializou

Teacher professor = new Teacher()

{

ID = 1,

Name = "Raimundo Nonato",

Salary = 1000

};

XmlSerializer xml = new XmlSerializer(typeof(Teacher));

using (var stream = new FileStream("Sample.xml", FileMode.Create))

{

xml.Serialize(stream, professor);

}

Console.WriteLine("A serialização XML foi concluída!");

using (var stream = new FileStream("Sample.xml", FileMode.Open))

{

professor = (Teacher)xml.Deserialize(stream);

}

Console.WriteLine(professor.ID);

Console.WriteLine(professor.Name);

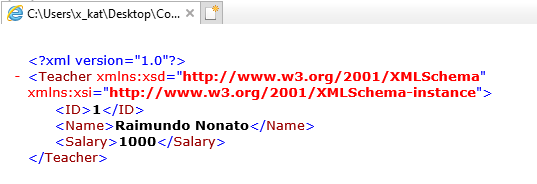
Console.WriteLine(professor.Salary);

Console.WriteLine("Desserialização XML concluída!");

Console.ReadKey();

}

}



A serialização XML pode ser configurada para obter mais controle sobre o tipo a ser serializado usando atributosfornecido pelo System.Xml.Serializationnamespace. A seguir, são atributos importantes (com seu uso) que são comumente usados:

|  |  |
| --- | --- |
| **Propriedade** | **Descrição** |
| XmlRoot | Aplicado no tipo, que informa ao compilador que este será o nó principal / pai de um objeto serializado em XML. |
| XmlAttribute | Aplicado em qualquer um dos campos públicos mapeados em um atributo em seu nó pai. |
| XmlElement | Aplicado em qualquer dos campos públicos mapeados em um elemento de um nó pai. |
| XmlIgnore | Aplicado em qualquer um dos campos públicos que não serão serializados. |
| XmlArray, XmlArrayItem | Esses dois (XmlArray e XmlArrayItem) podem ser aplicados em qualquer um dos campos públicos da coleção de tipos para serialização. |

Por padrão, cada campo público do seu tipo é serializado como XmlElement. Usando esses atributos mencionados acima, você pode mapear seu objeto no formato XML apropriado. Como no código abaixo, você também pode configurar os Atributos para obter mais controle sobre o objeto a ser serializado. E lembre-se, o tipo deve ser público para serialização XML, pois o XmlSerializer serializa apenas tipos ou membros públicos.

[Serializable]

[XmlRoot("Teacher")]

publicclassTeacher

{

[XmlAttribute("ID")]

publicint ID { get; set; }

[XmlElement("Name")]

publicstring Name { get; set; }

publiclong Salary { get; set; }

[XmlIgnore]

publicstring IgnoraCampo { get; set; }

[XmlElement("Students")]

public studentClass st { get; set; }

}

[Serializable]

publicclassstudentClass

{

[XmlAttribute("RollNo")]

publicint rollno { get; set; }

[XmlElement("Marks")]

publicint marks { get; set; }

}

SERIALIZAÇÃO COM DATACONTRACT

DataContractSerializer serializa um objeto em um formato XML usando um contrato de dados fornecido. Quando se trabalha com o WCF, seus tipos são serializados para que possam ser enviados para outros aplicativos. O WCF (Windows Communication Foundation) é uma estrutura para criar um aplicativo orientado a serviços.Essa serialização é feita por DataContractSerializer ou DataContractJsonSerializer. O WCF usa o DataContractSerializer como o serializador padrão.

As principais diferenças entre DataContractSerializer e XmlSerializer são:

1. Em vez de usar o atributo Serilizable, você usa o atributo DataContract.
2. Os membros não são serializados por padrão, como no XmlSerializer.
3. Todos os membros que você deseja serializar devem ser marcados explicitamente com um atributo DataMember.
4. Para ignorar um membro a ser serializado, use o atributo IgnoreDataMember em vez de XmlIgnore.
5. Os campos privados também são serializáveis pelo DataContractSerializer, o que não é possível no XmlSerializer.
6. No DataContractSerializer, você usa o método WriteObject () para serializar um objeto e o método ReadObject () para desserializar o fluxo em um objeto.

Você pode usar DataContractSerializer do namespace System.Runtime.Serialization da mesma maneira que usouXmlSerializer e BinarySerializer (BinaryFormatter) com a diferença de atributos ou métodos para serializar e desserializar.

[DataContract]

publicclassTeacher

{

[DataMember]

publicint ID { get; set; }

[DataMember(Name = "Nome")]

publicstring Name { get; set; }

[IgnoreDataMember]

publicstring IgnoraCampo { get; set; }

[DataMember]

publiclong Salary { get; set; }

}

classProgram

{

staticvoid Main(string[] args)

{

// Criou a instância e inicializou

Teacher professor = new Teacher()

{

ID = 1,

Name = "Raimundo Nonato",

IgnoraCampo = "kkkkk",

Salary = 1000,

};

DataContractSerializer dataContract = new DataContractSerializer(typeof(Teacher));

using (var stream = new FileStream("Sample.xml", FileMode.Create))

{

dataContract.WriteObject(stream, professor);

}

Console.WriteLine("A serialização DataContract foi concluída!");

using (var stream = new FileStream("Sample.xml", FileMode.Open))

{

professor = (Teacher)dataContract.ReadObject(stream);

}

Console.WriteLine(professor.ID);

Console.WriteLine(professor.Name);

Console.WriteLine(professor.Salary);

Console.WriteLine("Desserialização DataContract concluída!");

Console.ReadKey();

}

}

Serialização JSON com DataContractJsonSerializer

A serialização JSON serializa um objeto no formato JSON (JavaScript Object Notation), um formato de codificação eficiente que é especificamente útil ao enviar uma pequena quantidade de dados entre um cliente (navegador) e serviços da Web habilitados para AJAX.

A serialização JSON é manipulada automaticamente pelo WCF quando você usa tipos DataContract em operações de serviço expostas sobre pontos de extremidade habilitados para AJAX. No entanto, em alguns casos, pode ser necessário executar essa serialização manualmente com a serialização JSON, pois esse é um meio mais leve para armazenar dados em algum armazenamento ou enviar pela rede. DataContractJsonSerializer é usado para converter um objeto em dados JSON e converter novamente dados JSON em um objeto e é usado principalmente com o WCF.

DataContractJsonSerializer é uma classe fornecida pelo .NET no namespace System.Runtime.Serialization.Json. Como DataContractSerializer, DataContractJsonSerializer oferece suporte aos mesmos tipos que o DataContractSerializer tais como o método WriteObject() para serialização é um método ReadObject() para desserialização. O restante do procedimento para serialização JSON é o mesmo que os outros inclusive os membros privados também são serializados na serialização Json.

publicstatic Teacher SerializaJSONDataContract(Teacher professor)

{

//Serialization

DataContractJsonSerializer dataContract = new DataContractJsonSerializer(typeof(Teacher));

using (var stream = new FileStream("Sample.json", FileMode.Create))

{

dataContract.WriteObject(stream, professor);

}

Console.WriteLine("A serialização JSON DataContract foi concluída!");

using (var stream = new FileStream("Sample.json", FileMode.Open))

{

professor = (Teacher)dataContract.ReadObject(stream);

}

return professor;

}

Arquivo serializado Sample.json: {"ID":2,"Nome":"Prof Girafalles","Salary":55000}

Serialização JSON com JavaScriptSerializer

JavaScriptSerializer é uma classe fornecida pelo .NET no namespace System.Web.Script.Serialization encontrado no assembly System.Web.Extension usado para serializar e desserializar um objeto no formato Json para aplicativos habilitados para AJAX. Observe que não é necessário um atributo([Serializable] ou [DataContract]) para que o tipo de objeto seja serializado ao usar JavaScriptSerializer. Membros privados não podem ser serializados usando JavaScriptSerializer for Json Serialization

privateclassTeacher

{

privateint id { get; set; }

publicstring name { get; set; }

publiclong salary { get; set; }

}

staticvoid Main(string[] args)

{

// Criou a instância e inicializou

Teacher professor = new Teacher()

{

name = "Raimundo Nonato",

salary = 1000,

};

JavaScriptSerializer dataContract = new JavaScriptSerializer();

string serializedDataInStringFormat = dataContract.Serialize(professor);

Console.WriteLine("A serialização JavaScript foi concluída!");

professor = dataContract.Deserialize<Teacher>(serializedDataInStringFormat);

Console.WriteLine(professor.name);

Console.WriteLine(professor.salary);

Console.WriteLine("Desserialização JavaScript concluída!");

Console.ReadKey();

}

SERIALIZAÇÃO PERSONALIZADA (CUSTOM)

A serialização personalizada permite que um objeto controle sua própria serialização e desserialização. Existem dois métodos para personalizar os processos de serialização. O primeiro é adicionar um atributo antes de um método personalizado que manipule os dados do objeto durante e após a conclusão da serialização e desserialização. Você pode usar quatro atributos para fazer isso:

* OnDeserializedAttribute,
* OnDeserializingAttribute,
* OnSerializedAttribute e
* OnSerializingAttribute.

A adição desse atributo antes de uma declaração de método o reafirma durante ou após o processo de serialização ou desserialização. O código a seguir pode ser adicionado à classe Person para personalizar a lógica de serialização:

[OnSerializing()]

internalvoid OnSerializingMethod(StreamingContext context)

{

Console.WriteLine("OnSerializing.");

}

[OnSerialized()]

internalvoid OnSerializedMethod(StreamingContext context)

{

Console.WriteLine("OnSerialized.");

}

[OnDeserializing()]

internalvoid OnDeserializingMethod(StreamingContext context)

{

Console.WriteLine("OnDeserializing.");

}

[OnDeserialized()]

internalvoid OnDeserializedMethod(StreamingContext context)

{

Console.WriteLine("OnSerialized.");

}

Se você executar o código para qualquer um dos objetos serializadores e colocar pontos de interrupção em cada método, poderá ver quando cada método é chamado. Isso permite que você personalize a entrada ou a saída, caso você tenha aprimoramentos em seus objetos em versões posteriores e as propriedades estejam ausentes nos arquivos persistentes.

A segunda opção para personalizar o processo de serialização é implementar a interface ISerializable no tipo de um objeto.

ISerializable

ISerializable é uma interface que permite implementar serialização personalizada. Ao implementar a interface ISerializable, uma classe deve fornecer o método GetObjectData incluído na interface, além de um construtor especializado para aceitar dois parâmetros: uma instância de SerializationInfo e uma instância de StreamingContext sado quando o objeto é desserializado.

O método GetObjectData() é chamado durante a serialização e você precisa preencher o SerializationInfo fornecido com a chamada do método. Adicione a variável ou o valor a ser serializado com o nome associado a ela no método AddValue() do objeto SerializationInfo. Você pode usar qualquer texto como um nome associado a um valor ou variável. Você pode adicionar algumas ou algumas variáveis fornecidas com a chamada de método no objeto SerializationInfo. Essas variáveis ou valores fornecidos serão serializados. Com a desserialização, um construtor especial chamaria e os valores serializados desserializariam chamando o método Get do objeto SerializationInfo.

É importante lembrar que um objeto serializado pode expor dados privados sensíveis à segurança. Todo mundo que tem permissão para desserializar o arquivo pode acessar seus dados confidenciais. Se suas classes não exigirem um controle refinado do estado de seus objetos, você poderá usar o atributo [Serializable]. Classes que requerem mais controle sobre o processo de serialização podem implementar a interface ISerializable. Ao implementar ISerializable, você pode fornecer seu próprio mecanismo de serialização binária. Observe que o equivalente em xml disso é IXmlSerializable, conforme usado pelo XmlSerializer etc.

Para fins de DTO, BinaryFormatter deve ser evitado - coisas como xml (via XmlSerializer ou DataContractSerializer) ou json são boas, assim como formatos de plataforma cruzada, como buffers de protocolo.

[Serializable]

publicclassTeacher\_custom : ISerializable

{

publicint ID { get; set; }

publicstring Name { get; set; }

public Teacher\_custom()

{

}

protected Teacher\_custom(SerializationInfo info, StreamingContext context)

{

this.ID = info.GetInt32("IDKey");

this.Name = info.GetString("NameKey");

}

[SecurityPermissionAttribute(SecurityAction.Demand, SerializationFormatter = true)]

publicvoid GetObjectData(SerializationInfo info, StreamingContext context)

{

info.AddValue("IDKey", 9999);

info.AddValue("NameKey", "Novo Nome Prof");

}

}

staticvoid Main(string[] args)

{

Console.WriteLine("========Custom============");

Teacher\_custom professor\_cust = new Teacher\_custom()

{

ID = 11,

Name = "Prof Girafalles",

};

BinaryFormatter binaryFmt = new BinaryFormatter();

using (var stream = new FileStream(@"xmlCustom.xml", FileMode.OpenOrCreate))

{

binaryFmt.Serialize(stream, professor\_cust);

}

Console.WriteLine("A serialização Custom foi concluída!");

using (var stream = new FileStream(@"xmlCustom.xml", FileMode.Open))

{

professor\_cust = (Teacher\_custom)binaryFmt.Deserialize(stream);

}

Console.WriteLine(professor\_cust.ID);

Console.WriteLine(professor\_cust.Name);

Console.WriteLine("Desserialização Custom concluída!");

Console.ReadKey();

}

Usar atributos em vez de ISerializable

O uso dos quatro atributos é considerado a melhor prática, em vez de implementar a interface ISerializable. Também é mais fácil de implementar. Os métodos de atributo permitem manipular o objeto subjacente antes ou depois da serialização ou desserialização. A implementação da interface ISerializable intercepta o processo de serialização\desserialização e pode ter resultados inesperados ao trabalhar com objetos herdados de outro objeto que precisam ser serializados ou desserializados.

Comparação de desempenho de serialização

A tabela a seguir mostra umcomparativo de desempenho de diferentes técnicas de serialização por tamanho de dados (em bytes) e tempo (em milissegundos) necessário para serializar e desserializar um objeto ou gráfico de objeto:

|  |  |  |  |
| --- | --- | --- | --- |
| **Tamanhos** | **Binário** | **XML** | **Data Contract** |
| **Tamanho (Pequeno)** | **669** | **298** | **370** |
| Serialize | 0.0210 | 0.0218 | 0.004 |
| Deserialize | 0.0194 | 0.0159 | 0.0127 |
| **Tamanho (Grande)** | **204,793** | **323,981** | **364,299** |
| Serialize | 13.7000 | 5.5080 | 4.4438 |
| Deserialize | 19.3976 | 7.8893 | 11.4690 |

Sumário

1. O processo de conversão de um objeto ou gráfico de objetos em um fluxo de bytes é chamado de serialização e o processo inverso é chamado de desserialização.
2. A serialização binária é realizada usando um atributo serializável. É mais seguro que outras serializações, mas restrito a plataforma .NET.
3. A serialização XML serializa apenas membros públicos e não está restrita a uma plataforma .NET. Um objeto serializado XML é legível em comparação com um objeto serializado binário, que não é legível para humanos.
4. XmlSerializer e DataContractSerializer: as duas classes podem ser usadas para serialização de XML.
5. A serialização JSON é considerada uma abordagem rápida de serialização. É leve comparado com XML e serialização binária. Como na serialização XML, você pode apenas serializar membros públicos.
6. DataContractJsonSerializer e JavaScriptSerializer: ambas podem ser usadas para serialização JSON.
7. A serialização personalizada também pode ser realizada implementando uma interface ISerializable.

Armazenar dados e recuperar dados de coleções

* Armazenar e recuperar dados usando dicionários, matrizes, listas, conjuntos e filas; escolher um tipo de coleção; inicializar uma coleção; adicionar e remover itens de uma coleção; usar coleções tipificadas vs. coleções não tipificadas; implementar coleções personalizadas; implementar interfaces de coleções

Entender como manipular uma série de dados é fundamental para todos os tipos de desenvolvedores. Por exemplo, controles drop-downrequerem um conjunto de dados, a leitura de registros de um banco de dados requer um conjunto de dados e a leitura de um arquivo exige o armazenamento de um conjunto de dados na memória. Existem muitos termos diferentes que as pessoas usam para descrever uma série de dados, como matrizes (arrays), conjuntos (sets), coleções, listas, dicionários, pilhas (stacks) e filas (queues). Todos eles são usados para armazenar uma série de dados na memória e cada um oferece funcionalidade para anexar, pesquisar e classificar os dados. Esta seção explica matrizes e coleções e as diferenças entre os dois. Matrizes são o tipo mais primitivo em C#, com funcionalidade limitada, enquanto coleções é um termo geral que abrange listas, dicionários, filas e outros objetos.

Escolhendo uma coleção

Ao escolher um tipo de coleção, você deve pensar nos cenários que deseja oferecer suporte. As maiores diferenças entre as coleções são as maneiras pelas quais você acessa elementos.Os tipos de lista e dicionário oferecem acesso aleatório a todos os elementos. Um dicionário oferece recursos de leitura mais rápidos, mas não pode armazenar elementos duplicados.

Uma fila e uma pilha são usadas quando você deseja recuperar itens em uma ordem específica. O item é removido quando você o recupera. Coleções baseadas em conjuntos têm recursos especiais para comparar coleções. Eles não oferecem acesso aleatório a elementos individuais. Embora a Lista possa ser usada na maioria das situações, vale a pena ver se existe uma coleção mais especializada que pode facilitar sua vida.

Existem duas maneiras de criar e gerenciar grupos de objetos relacionados em C #

1. Array: armazenam várias variáveis ​​do mesmo tipo em uma estrutura de dados da matriz.
2. Collection: fornecem uma maneira flexível de trabalhar com grupos de objetos

* Ao contrário das matrizes, o grupo de objetos pode crescer e encolher dinamicamente
* Existem vários tipos diferentes de coleções

MATRIZES

Uma matriz é o tipo mais básico usado para armazenar um conjunto de dados. Matrizes são úteis quando você está trabalhando com um número fixo de objetos que todos têm o mesmo tipo.Você declara uma matriz usando uma sintaxe especial: tipo [] arrayName. Os colchetes denotam o tipo como sendo uma matriz. Todas as matrizes são herdadas da classe base System.Array. Esta classe contém propriedades e métodos que são úteis ao trabalhar com matrizes. Ao criar uma matriz, você deve especificar o número de itens que ela conterá. O exemplo a seguir cria uma única matriz dimensional de números de elementos é 4 inteiros:

int[] mySet\_1d = newint[4];

mySet\_1d[0] = 1;

mySet\_1d[1] = 2;

mySet\_1d[2] = 3;

mySet\_1d[3] = 4;

Você pode percorrer uma matriz usando a propriedade Length e um loop for.

int[] arrayOfInt = newint[10];

for (int x = 0; x < arrayOfInt.Length; x++)

{

arrayOfInt[x] = x;

}

Console.WriteLine(arrayOfInt.GetType()); //System.Int32[]

Matrizes são tipos de referência que herdam da classe Array. Uma matriz implementa IEnumerable, para que você possa usá-lo em um loop foreach.

var arrayRange = Enumerable.Range(0, 10);

//System.Linq.Enumerable +< RangeIterator > d\_\_113

Console.WriteLine(arrayRange.GetType());

Console.WriteLine(arrayRange.ToArray().GetType()); //System.Int32[]

Uma matriz também pode ser inicializada diretamente. O compilador verificará o comprimento para você e criará uma matriz apropriada.

int[] arrayInit = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };

Console.WriteLine(arrayInit.GetType()); //System.Int32[]

Além de declarar matrizes unidimensionais, você também pode criar matrizes multidimensionais e irregulares (jagged).Uma matriz bidimensional, por exemplo, significa que a matriz possui um certo número de linhas e colunas. Você o declara usando uma vírgula (,) na sua declaração de matriz. O número de linhas e colunas pode ser diferente. Você pode criar 3, 4 ou 5 ou até 2.147.483.647 dimensões. Você simplesmente precisa declarar a variável com o número de dimensões e inicializar o tamanho de cada dimensão. Paradeclarar uma matriz bidimensionaluse a seguinte sintaxe:

int[,] mySet\_2d = newint[2, 2];

mySet\_2d[0, 0] = 1;

mySet\_2d[0, 1] = 2;

mySet\_2d[1, 0] = 3;

mySet\_2d[1, 1] = 4;

Matrizes em C# são baseadas em zero. Portanto, se você possui uma matriz com dois elementos, o primeiro índice é 0 e a segunda é 1. Você pode ver uma pergunta sobre isso no exame. O código anterior criou uma matriz bidimensional com 2 elementos na primeira dimensão e 2 elementos na segunda dimensão. Conceitualmente, é como ter uma tabela com linhas e colunas.

Uma matriz irregular é uma matriz cujos elementos são matrizes. Como matrizes são tipos de referência, os valores de uma matriz irregular têm um valor padrão nulo. O exmplo abaixo mostra como você pode criar uma matriz irregular usando a sintaxe de inicialização.

int[][] jaggedArray =

{

newint[] {1,3,5,7,9},

newint[] {0,2,4,6},

newint[] {42,21}

};

foreach (var vector in jaggedArray)

{

foreach (var i in vector)

{

Console.Write(i);

}

Console.WriteLine();

}

Console.WriteLine(jaggedArray[0][0]); // 1

Console.WriteLine(jaggedArray[0][1]); // 3

As duas propriedades mais usadas de uma matriz sãoLength e Rank. A propriedade Length indica o número total de elementos em todas as dimensões da matriz. A propriedade Rank indica o número de dimensões na matriz. Essas propriedades são úteis ao determinar os limites de uma matriz ao executar loops for ou while. O método Clone é usado para fazer uma cópia superficial da matriz, enquanto o método CopyTo copia os elementos da matriz para outra matriz.

O maior problema com matrizes é que elas são de tamanho fixo. Ao trabalhar com grupos de objetos, você geralmente deseja adicionar ou remover itens da coleção. É por isso que o .NET Framework possui outros tipos de coleção.

Noções básicas sobre cópias rasas

É importante entender o conceito de uma cópia superficial. Ao clonar uma matriz com tipos de referência, você pode alterar inadvertidamente a matriz original se não entender o conceito de cópia superficial. Considere o seguinte exemplo:

Person[] orginal = new Person[1];

orginal[0] = new Person()

{

Name = "John"

};

Person[] clone = (Person[])orginal.Clone();

clone[0].Name = "Mary";

Person[] clonereplace = (Person[])orginal.Clone();

clonereplace[0] = new Person() { Name = "Bob" };

Console.WriteLine("Original name " + orginal[0].Name); //Original name Mary

Console.WriteLine("Clone name " + clone[0].Name); //Clone name Mary

Console.WriteLine("Replaced Clone name " + clonereplace[0].Name); //Replaced Clone name Bob

Console.ReadKey();

Neste exemplo, a propriedade Name do primeiro elemento no clone é alterada para "Mary" alterou també o Name da matriz orginal. Uma cópia superficial contém a referência ao elemento original na matriz original. Em clonereplace os nomes são diferentes porque na matriz clonada a referência no primeiro elemento foi substituída.

SYSTEM.COLLECTIONS

A coleção ajuda a gerenciar um grupo de objetos relacionados. No C#, coleções são estruturas de dados que fornecem uma maneira flexível de armazenar e recuperar objetos dinamicamente. Diferentemente das matrizes, um grupo de objetos em uma coleção pode aumentar e diminuir a qualquer momento. Coleções são classes instanciadas para gerenciar um grupo de objetos relacionados. No C#, existem três tipos de coleções:

1. **System.Collections**
2. **System.Collections.Generic**
3. **System.Collections.Concurrent**

System.Collections é um namespace que contém classes e interfaces que gerencia um grupo de dados. Ele armazena cada dado na forma de um tipo system.object. Portanto, um grupo de dados do tipo de valor sempre é colocado na boxed/unboxed. Ele define várias estruturas de dados para armazenar e recuperar dados como lista, fila e hashtable.

|  |  |
| --- | --- |
| **Classe** | **Explicação** |
| ArrayList | Matriz de objetos cujo tamanho pode aumentar e diminuir dinamicamente |
| Hashtable | Coleção de pares chave/valor, organizada com base no código hash |
| SortedList | Coleção de pares chave/valor cujos elementos são classificados pelo valor da chave |
| Queue | Gerencia o grupo de dados na ordem FIFO (primeiro a entrar, primeiro a sair) |
| Stack | Gerencia o grupo de dados na ordem LIFO (Ultimo a entrar primeiro a sair) |

A seguir, é apresentada uma lista de algumas propriedades e métodos usados com freqüência, definidos em System.Collections:

|  |  |  |
| --- | --- | --- |
| **Método e Propriedade** | **Classes** | **Explicação** |
| Add()  AddRange() | ArrayList, SortedList, Hashtable | Adicionar um objeto à Collection. |
| List<T>, Dictionary<K,V>, HashSet<T> |
| Enqueue() | Queue, Queue<T> |
| Push() | Stack, Stack<T> |
| Contains() | ArrayList, SortedList, Queue, Stack | Retorne true se o objeto específico estiver em Collection |
| List<T>, Queue<T>, Stack<T>, HashSet<T> |
| Sort() | ArrayList, List<T> | Classifique todos os objetos da Lista <T> usando o comparador |
| Clone() | ArrayList, SortedList, Hashtable, Queue, Stack | Crie uma cópia superficial de Collection |
| Remove() | ArrayList, SortedList, Hashtable | Remova a primeira ocorrência de objeto específico naCollection |
| List<T>, Dictionary<K,V>  HashSet<T> |
| Dequeue() | Queue, Queue<T> |
| Pop() | Stack, Stack<T> |
| RemoveAt() | ArrayList, SortedList, List<T> | Remova o objeto do índice específico de Collection |
| Clear() | Todas Classes | Remova todos os objetos da Collection |
| Find() | List<T> | Pesquise o objeto usando o predicado especificado |
| ContainsKey() | Hashtable, Dictionary<K,V> | Retorne true se a chave específica estiver naCollection |
| Peek() | Queue, Stack | Retornar o objeto no início da fila sem removê-lo |
| Queue<T>, Stack<T> |
| ToArray() | Queue, Stack | Copie os elementos da fila para uma nova matriz |
| Queue<T>, Stack<T> |
| Count | Todas Classes | Obter o número real de objetos armazenados em Collection |
| Capacity | ArrayList, SortedList | Obter ou definir o número de objetos que Collection pode conter |
| Item | ArrayList, SortedList | Obter ou define o elemento no índice especificado |
| ContainsValue | Hashtable, Dictionary<K,V> | Retorno true se um valor específico estiver naCollection |
| Keys | Hashtable, SortedList, Dictionary<K,V> | Obter lista de chaves contidas naCollection |
| Values | Hashtable, SortedList, Dictionary<K,V> | Obter lista de valores contidos naCollection |

ArrayList

É uma variedade de objetos que podem aumentar e diminuir seu tamanho dinamicamente. Ao contrário de matrizes, um ArrayList pode armazenar dados de vários tipos de dados. Pode ser acessado por seu índice. Inserindo e excluindo um elemento no meiode um ArrayList é mais caro do que inserir ou excluir um elemento no final de um ArrayList.

Um ArrayList contém muitos métodos e propriedades que ajudam a gerenciar um grupo de objetos.

ArrayList arraylist = new ArrayList();

//add objects in arraylist

arraylist.Add(22);

arraylist.Add("Ali");

arraylist.Add(true);

//Iterate over each index of arraylist

for (int i = 0; i < arraylist.Count; i++)

{

Console.WriteLine(arraylist[i]);

}

arraylist.Remove(22);

Console.WriteLine();

foreach (var item in arraylist)

{

Console.WriteLine(item);

}

Hashtable

O Hashtable armazena cada elemento de uma coleção em um par de chaves / valores. Otimiza as pesquisas computando a chave de hash e a armazena para acessar o valor correspondente.

Hashtable owner = new Hashtable();

//There are no keys but value can be duplicate

owner.Add("Bill", "Microsoft");

owner.Add("Paul", "Microsoft");

owner.Add("Steve", "Apple");

owner.Add("Mark", "Facebook");

//Display value against key

Console.WriteLine("Bill is the owner of {0}", owner["Bill"]);

//ContainsKey can be use to test key before inserting

if (!owner.ContainsKey("Trump"))

{

owner.Add("Trump", "The Trump Organization");

}

// When you use foreach to enumerate hash table elements,

// the elements are retrieved as KeyValuePair objects.

//DictionaryEntry is the pair of key & value

foreach (DictionaryEntry item in owner)

{

Console.WriteLine("{0} is owner of {1}", item.Key, item.Value);

}

var allValues = owner.Values;

foreach (var item in allValues)

{

Console.WriteLine("Company: {0}", item);

}

SortedList

Uma SortedList é uma coleção que contém pares chave \ valor, mas é diferente de uma Hashtable porque pode ser referenciada pela chave ou pelo índice e porque está classificada. Os elementos na SortedList são classificados pela implementação IComparable da chave ou pela implementação IComparer quando theSortedList é criado. O código a seguir cria uma SortedList, adiciona três elementos à lista e imprime os elementos na janela Saída:

SortedList mySortedList = new SortedList();

mySortedList.Add(3, "three");

mySortedList.Add(2, "second");

mySortedList.Add(1, "first");

foreach (DictionaryEntry item in mySortedList)

{

Console.WriteLine(item.Value);

}

Observe que a ordem dos elementos foi impressa com base na ordem da chave, não na ordem em que foram adicionados à lista. O tipo de variáveis passadas para o parâmetro-chave deve ser comparável entre si. Se você tentar adicionar um elemento com um número inteiro para uma chave e, em seguida, adicionar um segundo elemento com uma sequência de caracteres para uma chave, você receberá um erro porque os dois não podem ser comparados. Se sua lista contiver elementos com tipos diferentes para a chave, use um Hashtable

Queue

Fila é uma classe do namespace System.Collections. Ele armazena e recupera objetos na ordem FIFO (primeiro a entrar, primeiro a sair). Em outras palavras, ele gerencia uma coleção de objetos por ordem de chegada.Você acessa elementos na mesma ordem em que os adicionou. Ao obter um item, você também o remove da fila. É por isso que uma fila oferece armazenamento temporário.

Você pode usar uma fila, por exemplo, quando precisar processar mensagens recebidas. Cada nova mensagem é adicionada ao final da fila; Quando você terminar de processar uma mensagem, receberá uma nova desde o início da fila.A classe Queue possui três métodos importantes:

* Enqueue adiciona um elemento ao final da fila, equivalente ao final da linha.
* Dequeue remove o elemento mais antigo da fila, equivalente à frente da linha.
* Peek retorna o elemento mais antigo, mas não o remove imediatamente da fila.

Queue days = new Queue();

//Add(Enque) objects in queus

days.Enqueue("Mon");

days.Enqueue("Tue");

days.Enqueue("Wed");

days.Enqueue("Thu");

days.Enqueue("Fri");

days.Enqueue("Sat");

days.Enqueue("Sun");

// Displays the properties and values of the Queue.

Console.WriteLine("Total elements in queue are {0}", days.Count);

//Remove and return first element of the queue

Console.WriteLine("{0}", days.Dequeue());

//return first element of queue without removing it from queue

//return 'Tue'

Console.WriteLine("{0}", days.Peek());

//Iterate over each element of queue

Console.WriteLine();

Stack

Stack é uma classe do namespace System.Collections. Ele armazena e recupera objetos na ordem LIFO (Last In, First Out). Em outras palavras, os elementos pressionados no final aparecerão primeiro, por exemplo, uma pilha de pratos.Uma pilha possui os três métodos importantes a seguir:

* Push: Adicionar um novo item à pilha.
* Pop: Obtenha o item mais recente da pilha.
* Peek: Obtenha o item mais novo sem removê-lo.

Stack history = new Stack();

//Insert browser history in stack

history.Push("google.com");

history.Push("facebook.com/imaliasad");

history.Push("twitter.com/imaliasad");

history.Push("youtube.com");

// Displays the properties and values of the Stack.

Console.WriteLine("Total elements in stack are {0}", history.Count);

//Remove and return top element of the Stack

Console.WriteLine("{0}", history.Pop());

//return top element of Stack without removing it from Stack

//return 'twitter.com/imaliasad'

Console.WriteLine("{0}", history.Peek());

//Iterate over each element of Stack

foreach (var item in history)

{

Console.WriteLine(item);

}

SYSTEM.COLLECTIONS.GENERICS

Compreendendo genérico versus não genérico

A maioria dos tipos de coleção possui uma versão genérica e uma não genérica. Ao trabalhar com objetos de um tipo específico (ou tipo base), use a coleção genérica. Isso melhorará a segurança e o desempenho do tipo, porque não há necessidade de casting.As coleções não genéricas podem ser encontradas em System.Collections e coleções genéricas podem ser encontradas em System.Collections.Generic.

Se você usar um tipo de valor como parâmetro de tipo para uma coleção genérica, precisará eliminar todos os cenários nos quais o boxe pode ocorrer. Por exemplo, se seu tipo de valor não implementar IEquatable <T>, seu objeto precisará de um box para chamar Object.Equals (Object) para verificar a igualdade. O mesmo vale para a interface IComparable <T>.Ao usar tipos de referência, você não terá esses problemas.

System.Collections.Generics é um namespace que contém classes e interfaces para gerenciar uma coleção fortemente tipada. Em uma coleção genérica, os dados não podem ser colocados na caixa / fora da caixa porque os dados sempre são protegidos por tipo. É mais rápido e melhor que as classes e interfaces definidas no System.Collections. Ele também define várias estruturas de dados para armazenar e recuperar dados como List<T>, Queue<T>, Stack<T> e Dictionary<TKey, TValue>.

|  |  |
| --- | --- |
| **Classe** | **Explicação** |
| List<T> | Lista de objetos de tipo seguro que podem crescer e encolher dinamicamente |
| Dictionary<Tkey,Tvalue> | Representa a coleção de chaves e valores de segurança de tipo |
| SortedList <TKey, TValue> | Cria uma coleção de pares chave\valor que são classificados com base na chave e devem ser do mesmo tipo |
| Queue<T> | Representa a coleção First In, First Out de objetos de tipo seguro |
| Stack<T> | Representa a coleção Last In, First Out de objetos de tipo seguro |

List<T>

List<T> é uma coleção de objetos com segurança de tipo. A lista pode aumentar e diminuir seu tamanho dinamicamente. Com o suporte a genéricos, ele pode armazenar uma coleção de qualquer tipo de maneira segura. Portanto, é muito mais rápido e otimizado que ArrayList.

O tipo de lista garante que sempre haja espaço suficiente para armazenar itens adicionais. Se necessário, a implementação interna da classe List aumentará o tamanho da matriz usada para armazenar seus itens. A lista <T> pode armazenar tipos de referência e pode ter um valor nulo para um item. Também pode armazenar itens duplicados.

List<Person> people = new List<Person>();

//Add Person in list

people.Add(new Person { Name = "Ali", Age = 22 });

people.Add(new Person { Name = "Sundus", Age = 21 });

people.Add(new Person { Name = "Hogi", Age = 12 });

//Get total number of person in list

Console.WriteLine("Total person are: {0}", people.Count);

//Iterate over each person

foreach (var person in people)

{

Console.WriteLine("Name: {0} - Age: {1}", person.Name, person.Age);

}

//Instantiate and populate list of int with values

List<int> marks = new List<int> { 10, 25, 15, 23 };

//Remove '25' from the list

marks.Remove(25);

//Get each element by its index

Console.Write("Marks: ");

for (int i = 0; i < marks.Count; i++)

{

Console.Write(marks[i] + " ");

}

Dictionary<TKey, TValue>

O Dictionary<TKey, TValue> é uma classe de System.Collections.Generic. É uma coleção segura de tipos de pares de chave/valor. Cada chave no dicionário deve ser exclusiva e pode armazenar vários valores na mesma chave. O Dictionary<TKey, TValue> é muito mais rápido que o Hashtable.

//Initialize Dictionary (int for roll# and assign it to student)

Dictionary<int, Person> people = new Dictionary<int, Person>();

//Adding student against their roll#

people.Add(53, new Person { Name = "Ali Asad", Age = 22 });

people.Add(11, new Person { Name = "Sundus Naveed", Age = 21 });

people.Add(10, new Person { Name = "Hogi", Age = 12 });

//Display Name against key

Console.WriteLine("Roll# 11 is: {0}", people[11].Name);

//ContainsKey can be use to test key before inserting

if (!people.ContainsKey(13))

{

people.Add(13, new Person { Name = "Lakhtey", Age = 21 });

}

// When you use foreach to enumerate elements of dictionary,

// the elements are retrieved as KeyValuePairPair object.

//KeyValuePair<TKey, TValue> is the pair of key & value for dictionary

foreach (KeyValuePair<int, Person> student in people)

{

Console.WriteLine("Roll#: {0} - Name: {1} - Age: {2}",

student.Key, student.Value.Name, student.Value.Age);

}

//Get All values stored in Dictionary

var allValues = people.Values;

foreach (var student in allValues)

{

Console.WriteLine("Name: {0} - Age: {1}",

student.Name, student.Age);

}

HashSet<T>

Em algumas linguagens, como Java, há um tipo de conjunto especial. Em C#, um conjunto é uma palavra-chave reservada, mas você pode usar o HashSet<T> se precisar de uma. Um conjunto é uma coleção que não contém elementos duplicados e não possui uma ordem específica.

HashSet<int> oddSet = new HashSet<int>();

HashSet<int> evenSet = new HashSet<int>();

for (int x = 1; x <= 10; x++)

{

if (x % 2 == 0)

evenSet.Add(x);

else

oddSet.Add(x);

}

DisplaySet(oddSet);

DisplaySet(evenSet);

oddSet.UnionWith(evenSet);

DisplaySet(oddSet);

privatestaticvoid DisplaySet(HashSet<int> set)

{

Console.Write("{");

foreach (int i in set)

{

Console.Write("{0}", i);

}

Console.WriteLine("}");

}

Queue<T>

A Queue<T> é uma classe de tipo seguro do namespace System.Collections.Generic. Ele armazena e recupera dados na ordem FIFO (primeiro a entrar, primeiro a sair). Em outras palavras, ele gerencia uma coleta de dados por ordem de chegada. É muito mais rápido que a Queue definida em System.Collections porque o tipo de valor é colocado na caixa/fora da caixa na Queue, enquanto a Queue<T> sempre o protege com segurança.

Queue<string> days = new Queue<string>();

//Add(Enque) string object in days

days.Enqueue("Mon");

days.Enqueue("Tue");

days.Enqueue("Wed");

days.Enqueue("Thu");

days.Enqueue("Fri");

days.Enqueue("Sat");

days.Enqueue("Sun");

// Displays the properties and values of the Queue.

Console.WriteLine("Total elements in queue<string> are {0}",

days.Count);

//Remove and return first element of the queue<string>

Console.WriteLine("{0}", days.Dequeue());

//return first element of queue without removing it from queue

//return 'Tue'

Console.WriteLine("{0}", days.Peek());

//Iterate over each element of queue

foreach (var item in days)

{

Console.WriteLine(item);

}

Stack<T>

A Stack<T> é uma classe do namespace System.Collections.Generic. Ele armazena e recupera elementos na ordem LIFO (Last In, First Out). Em outras palavras, os elementos pressionados no final aparecerão primeiro, por exemplo, uma pilha de pratos. É muito mais rápido que a pilha definida no System.Collections porque o tipo de valor é colocado na boxed/unboxed na pilha, enquanto a Stack<T> sempre o protege com segurança.

Stack<string> history = new Stack<string>();

//Insert browser history in stack<string>

history.Push("google.com");

history.Push("facebook.com/imaliasad");

history.Push("twitter.com/imaliasad");

history.Push("youtube.com");

// Displays the properties and values of the Stack<string>.

Console.WriteLine("Total elements in stack<string> are {0}",

history.Count);

//Remove and return top element of the Stack<string>

Console.WriteLine("{0}", history.Pop());

//return top element of Stack<string> without removing it from Stack

//return 'twitter.com/imaliasad'

Console.WriteLine("{0}", history.Peek());

//Iterate over each element of Stack<string>

foreach (var item in history)

{

Console.WriteLine(item);

}

SYSTEM.COLLECTIONS.CONCURRENT

O namespace System.Collections.Concurrent foi introduzido na estrutura do .NET 4. Ele fornece várias classes de coleções seguras para encadeamento que protegem uma coleção de serem manipuladas por vários encadeamentos. As classes de coleção definidas em um System.Collections.Concurrent podem ser manipuladas apenas por um único thread.O .NET 4.0 Framework apresenta várias coleções seguras de thread no namespace System.Collections.Concurrent.

|  |  |
| --- | --- |
| **Classe** | **Explicação** |
| ConcurrentBag<T> | Representa uma coleção de objetos não ordenada e segura para threads. |
| ConcurrentDictionary<T, V> | Representa uma coleção segura de segmentos de pares de valor-chave. |
| ConcurrentQueue<T> | Representa uma coleção FIFO (Primeiro a entrar, Primeiro a sair). |
| ConcurrentStack<T> | Representa uma coleção LIFO (Last In, First Out) |

Discutiremos mais sobre System.Collections.Concurrent no capítulo 8: multithread, assíncrona e Programação paralela.

COLEÇÃO CUSTOMIZADA

Além das coleções padrão fornecidas pelo .NET, você pode criar suas próprias coleções fortemente tipadas e personalizadas. Coleções fortemente tipadas são úteis porque não incorrem no impacto do desempenho devido ao boxe e ao unboxing. Para criar sua própria coleção personalizada, você pode herdar da classe CollectionBase. As Tabelas 9-7 e 9-8 listam as propriedades e o método comumente usados da classe CollectionBase.

|  |  |
| --- | --- |
| **Propriedade** | **Descrição** |
| Capacity | Obtém ou define o número de elementos que a coleção pode conter |
| Count | Retorna o número de elementos no dicionário |
| InnerList | Obtém um ArrayList contendo os elementos na coleção |
| List | Obter um IList contendo os elementos na coleção |

Métodos System.Collections.CollectionBase (Lista parcial

|  |  |
| --- | --- |
| **Método** | **Descrição** |
| Clear | Limpa os elementos da coleção |
| OnInsert | Permite executar o processamento personalizado antes de inserir um novo elemento |
| OnRemove | Permite executar um processamento personalizado antes de remover um elemento |
| OnSet | Permite executar o processamento personalizado antes de definir um valor na coleção |
| RemoveAt | Remove o elemento no índice especificado |

Não há métodos Add, Insert, Sort, or Search na classe base. Ao implementar sua classe, você precisa implementar os métodos que deseja adicionar, inserir, classificar ou pesquisar itens na coleção.

publicclassPerson

{

publicint ID { get; set; }

publicstring FirstName { get; set; }

publicstring LastName { get; set; }

publicint Age { get; set; }

}

Você pode criar uma classe de coleção de pessoas que herda da classe CollectionBase. O código a seguir cria uma classe de coleção personalizada para a classe Person e cria os métodos Add, Insert e Remove e cria um indexador fortemente tipado. O indexador é usado quando você faz referência à coleção por índice, como myCollection [index].

classPersonCollection : CollectionBase

{

publicvoid Add(Person person)

{

List.Add(person);

}

publicvoid Insert(int index, Person person)

{

List.Insert(index, person);

}

publicvoid Remove(Person person)

{

List.Remove(person);

}

public Person this[int index]

{

get { return (Person)List[index]; }

set { List[index] = value; }

}

}

Agora que você tem uma classe PersonCollection fortemente tipada, pode usá-la em seu código:

PersonCollection persons = new PersonCollection();

persons.Add(new Person() { ID = 1, FirstName = "John", LastName = "Smith" });

persons.Add(new Person() { ID = 2, FirstName = "Jane", LastName = "Doe" });

persons.Add(new Person() { ID = 3, FirstName = "Bill Jones", LastName = "Smith" });

foreach (Person person in persons)

{

Console.WriteLine(person.FirstName);

}

O código anterior cria uma instância da classe PersonCollection, adiciona três objetos à classe e depois enumera a coleção e imprime o valor do elemento na janela Saída.Obviamente, você também pode optar por herdar diretamente de uma coleção existente. A Listagem 4-90 mostra um exemplo de como você pode herdar da Lista <T> para adicionar alguma funcionalidade específica.

publicclassPeopleCollection : List<Person>

{

publicvoid RemoveByAge(int age)

{

for (int index = this.Count - 1; index >= 0; index--)

{

if (this[index].Age == age)

{

this.RemoveAt(index);

}

}

}

publicoverridestring ToString()

{

StringBuilder sb = new StringBuilder();

foreach (Person p inthis)

{

sb.AppendFormat("{0} {1} is {2}", p.FirstName, p.LastName, p.Age);

}

return sb.ToString();

}

}

PeopleCollection people = new PeopleCollection {

new Person{ FirstName ="John", LastName = "Doe", Age = 42},

new Person{ FirstName = "Jane", LastName = "Doe", Age = 21 }};

people.RemoveByAge(42);

Console.WriteLine(people.Count); // Displays: 1

Console.ReadKey();

INTERFACES DE COLEÇÕES

Para podermos tratar uma coleção de informações temos que escrever e executar consultas e realizar iterações sobre o resultado. A interface IEnumerable e IEnumerator no .NET ajuda você a implementar o padrão do iterador, que permite acessar todos os elementos em uma coleção sem se preocupar com como ela é exatamente implementada. Você pode encontrar essas interfaces nos namespaces System.Collection e System.Collections.Generic. Ao usar o padrão do iterador, você pode iterar facilmente os elementos em uma matriz, lista ou coleção personalizada. É muito usado no LINQ, que pode acessar todos os tipos de coleções de maneira genérica sem realmente se importar com o tipo de coleção.

IEnumerable & IEnumerable<T>

O .NET definiu duas bibliotecas de classe base. Há uma interface IEnumerable não genérica para criar uma coleção não genérica personalizada e uma interface IEnumerable<T> segura para tipos genéricos para criar uma coleção segura para tipos.

IEnumerable

A interface IEnumerable é definida no namespace System.Collections. Ajuda a criar uma coleção não genérica personalizada. É a base direta ou indiretamente para algumas outras Interfaces e todas as coleções a implementam. A interface contém um único método GetEnumerator que retorna um IEnumerator. A definição de interface se parece com isso:

public interface IEnumerable

{

IEnumerator GetEnumerator();

}

O método GetEnumerator deve retornar uma instância de um objeto de uma classe que implementa a interface IEnumerator que itera sobre a coleção. IEnumerable<T> é sua tabela e IEnumerator é um cursor que só pode passar para a próxima linha e controla onde você está na tabela.



Antes do C # 2, implementar o IEnumerable em seus próprios tipos era bastante complicado. Você precisa acompanhar o estado atual e implementar outras funcionalidades, como verificar se a coleção foi modificada enquanto você a enumerava.Discutiremos o IEnumerator em muitos detalhes mais abaixo. Mas, por enquanto, o IEnumerator é usado para iterar sobre uma coleção, armazena as informações de um índice atual, seu valor e se uma iteração de coleção foi concluída ou não.

classmyArrayList : IEnumerable

{

object[] array = newobject[4];

int index = -1;

publicvoid Add(object o)

{

if (++index < array.Length)

{

array[index] = o;

}

}

public IEnumerator GetEnumerator()

{

for (int i = 0; i < array.Length; i++)

{

yieldreturn array[i];

}

}

}

myArrayList list = new myArrayList();

//stores object data in myArraylist

list.Add("Ali");

list.Add(22);

list.Add("Sundus");

list.Add(21);

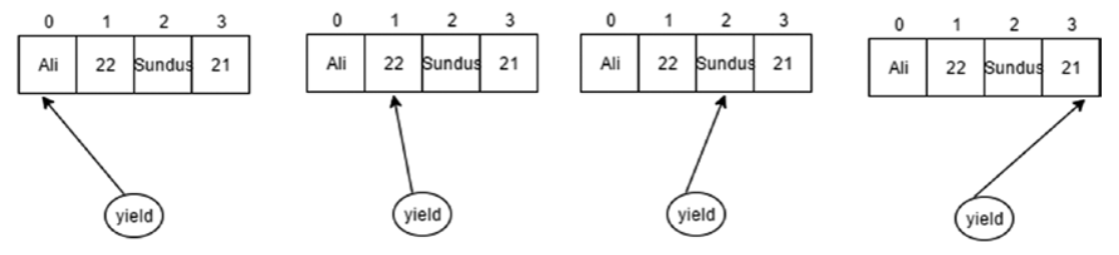
foreach (var item in list)

{

Console.WriteLine(item);

}

É importante saber que o método foreach da linguagem C# trabalha com todos os tipos que implementam a interface IEnumerable, ou seja, ArrayList e Queue.O loop foreach chama o método GetEnumerator da "lista", que gera um valor de retorno do índice de cada matriz em cada iteração. Portanto, myArrayList agora se tornou uma coleção personalizada devido ao IEnumerable. Na imagem a seguir, você pode ver um valoryield de retorno do índice de cada matriz em cada iteração.



// Let's create a class with an indexer

publicclassIndex\_ComYield : IEnumerable

{

string[] car = newstring[3];

publicstringthis[int carNum]

{

get { return car[carNum]; }

set { car[carNum] = value; }

}

// This is our enumerator It implments the IEnumerable interface

public IEnumerator GetEnumerator()

{

foreach (string c in car)

{

yieldreturn c;

}

}

}

Index\_ComYield c = new Index\_ComYield();

c[0] = "Chevrolet";

c[1] = "Mercedes";

c[2] = "Dodge";

// Se a classe Index\_ComYield não implemntar IEnumerable

// com o método GetEnumerator Dá erro no foreach

foreach (string car in c)

{

Console.WriteLine(car);

}

IEnumerable<T>

A versão genérica e de tipo ‘seguro’ chamado IEnumerable<T>, que está localizado no namespace System.Collections.Generic, define um único método GetEnumerator que retorna uma instância de um objeto que implementa a interface IEnumerator<T>.

public interface IEnumerable<out T> : IEnumerable

{

IEnumerator<T> GetEnumerator();

}

Como você pode ver, a interface IEnumerable<T> herda a interface IEnumerable. Portanto, um tipo que implementa IEnumerable<T> também tem de implementar os membros de IEnumerable.

classPerson

{

publicstring Name { get; set; }

publicint Age { get; set; }

}

Implementação com tipo mais seguro:

classPeople : IEnumerable<Person>

{

Person[] people;

public People(Person[] people)

{

this.people = people;

}

public IEnumerator<Person> GetEnumerator()

{

for (int index = 0; index < people.Length; index++)

{

yieldreturn people[index];

}

}

IEnumerator IEnumerable.GetEnumerator()

{

return GetEnumerator();

}

}

Implementação mais genérica:

classmyGenericList<T> : IEnumerable<T>

{

List<T> list = new List<T>();

//Get length of list<T>

publicint Length

{

get { return list.Count; }

}

publicvoid Add(T data)

{

list.Add(data);

}

public IEnumerator<T> GetEnumerator()

{

foreach (var item in list)

{

yieldreturn item;

}

}

IEnumerator IEnumerable.GetEnumerator()

{

//return IEnumerator<T> GetEnumerator()

returnthis.GetEnumerator();

}

}

myGenericList<Person> people = new myGenericList<Person>();

people.Add(new Person { Name = "Ali", Age = 22 });

people.Add(new Person { Name = "Sundus", Age = 21 });

people.Add(new Person { Name = "Hogi", Age = 12 });

Console.WriteLine("Total person: {0} \n", people.Length);

foreach (Person person in people)

{

Console.WriteLine("Name:{0} Age:{1}", person.Name, person.Age);

}

E em C# podemos trabalhar com objetos que não explicitamente implementam IEnumerable ou IEnumerable<T>, podemos fazer foreach em qualquer coleção. Por exemplo, métodos como Range(), facilitam a criação de coleções IEnumerable.

IEnumerable<int> result = from value in Enumerable.Range(0, 5) select value;

foreach (int value in result)

{

//0 1 2 3 4

Console.WriteLine(value);

}

Os métodos de extensão podem converter IEnumerable<int> ou para obter algumas informações sobre o IEnumerable<int>

List<int> list = result.ToList();

int[] array = result.ToArray();

Console.WriteLine(result.Average()); // 2

Console.WriteLine(result.Sum()); // 10

Console.WriteLine(result.First()); // 0

Console.WriteLine(result.Last()); // 4

Console.WriteLine(result.Max()); // 4

Console.WriteLine(result.Min()); // 0

Por fim, o IEnumerable<int> pode servir até como retorno de método utilizando yield. O yield é uma palavra-chave especial que pode ser usada apenas no contexto de iteradores. Ele instrui o compilador para criar uma classe de enumerador não explícito, ou seja, converte o código regular em um estado de máquina. O código gerado controla onde você está na coleção e implementa métodos como MoveNext e Current.

publicstatic IEnumerable<int> Soma\_Matriz2D(int[,] matriz\_2D)

{

// Use yield return to return all 2D array elements.

for (int x = 0; x < 15; x++)

{

for (int y = 0; y < 15; y++)

{

yieldreturn matriz\_2D[x, y];

}

}

}

int[,] matriz\_2D = newint[15, 15];

matriz\_2D[0, 1] = 4;

matriz\_2D[4, 4] = 5;

matriz\_2D[14, 2] = 3;

int sum = 0;

foreach (int value in Soma\_Matriz2D(matriz\_2D))

{

sum += value;

}

//SUMMED 2D ELEMENTS: 12

Console.WriteLine("SUMMED 2D ELEMENTS: " + sum);

IEnumerator e IEnumerator<T>

O .NET definiu duas bibliotecas de classe base. Existem interfaces IEnumerator não genéricas e genéricas para definir a iteração de uma coleção.

IEnumerator

IEnumerator suporta uma iteração simples sobre uma coleção sendo a interface base para todos os enumeradores.É a base para todos os enumeradores. O objeto IEnumerator fornece uma propriedade Current que retorna o objeto atual na enumeração. Ele também fornece um método MoveNext que move o enumerador para o próximo objeto na enumeração e um método Reset que redefine o enumerador imediatamente antes do início da enumeração (index = -1). Por fim, o enumerador fornece um método Dispose que permite limpar todos os recursos que estiver usando quando não for mais necessário.

public interface IEnumerator

{

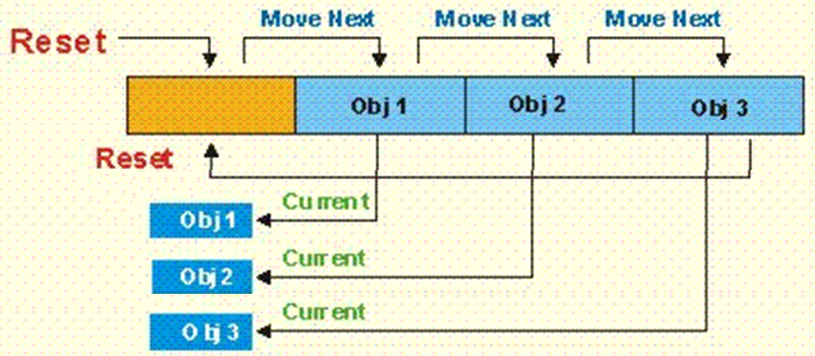
object Current { get; }

bool MoveNext();

void Reset();

}

Abaixo uma figura que mostra como funcionam os membros do IEnumerator:



Vejamos o funcionamento destes membros:

* Inicialmente, o enumerador é posicionado antes do primeiro elemento na coleção (index = -1). Nesta posição, se for chamada a propriedade Current ocorrerá uma exceção. O método Reset traz o enumerador de volta para essa posição.
* A propriedade Current retorna o mesmo objeto até que os métodos MoveNext ou Reset sejam chamados. Para ler o primeiro valor de Current, você deve chamar o método MoveNext para avançar o enumerador do primeiro elemento da coleção.Para voltar para o primeiro valor Current, deve-se chamar Reset seguido de um MoveNext;
* O método MoveNext define a propriedade Current para o próximo elemento. Se MoveNext passar o final da coleção, o enumerador estará posicionado após o último elemento na coleção e MoveNext retornará false. Chamadas subseqüentes para MoveNext também retornam false, se houver uma chamada de Current gera uma exceção;
* Os enumeradores podem ser usados para ler dados em uma coleção, mas não podem modificar estes dados.Se forem feitas alterações na coleção, como adicionar, modificar ou excluir elementos, o enumerador ficará invalidado e a próxima chamada para MoveNext ou Reset gera uma exceção InvalidOperationException. Se a coleção é modificada entre MoveNext e Current, Current retornará o elemento que é definido, mesmo se o enumerador está já invalidado.

A instrução foreach(C#)/for each(VB) oculta a complexidade dos enumeradores. Portanto é recomendável usar foreach/for each ao invés de tentar manipular diretamente o enumerador. Quando utilizamos um foreach, internamente o C# chama o método GetEnumerator da coleção e a cada condição do loop ele executa um MoveNext. Conforme o exemplo a seguir que percorre um array de inteiros:

List<int> numbers = new List<int> { 1, 2, 3, 5, 7, 9 };

foreach (int value in numbers)

{

Console.Write(value + " - "); // 1 - 2 - 3 - 5 - 7 - 9 -

}

O código anterior, é apenas um atalho sintático para a seguinte operação:

using (IEnumerator<int> enumerator = numbers.GetEnumerator())

{

// 1 - 2 - 3 - 5 - 7 - 9 -

while (enumerator.MoveNext()) Console.Write(enumerator.Current + " - ");

}

using (List<int>.Enumerator enumerator = numbers.GetEnumerator())

{

// 1 - 2 - 3 - 5 - 7 - 9 -

while (enumerator.MoveNext()) Console.Write(enumerator.Current + " - ");

}

Defina a iteração da sua coleção personalizada com IEnumerator.

classPerson

{

publicstring Name { get; set; }

publicint Age { get; set; }

}

classPeople : IEnumerable

{

Person[] people;

int index = -1;

publicvoid Add(Person per)

{

if (++index < people.Length)

{

people[index] = per;

}

}

public People(int size)

{

people = new Person[size];

}

public IEnumerator GetEnumerator()

{

returnnew PersonEnum(people);

}

}

//Implement IEnumerator

classPersonEnum : IEnumerator

{

Person[] \_people;

int index = -1;

public PersonEnum(Person[] people)

{

\_people = people;

}

//Check whether foreach can move to next iteration or not

publicbool MoveNext()

{

return (++index < \_people.Length);

}

//Reset the iteration

publicvoid Reset()

{

index = -1;

}

//Get current value

publicobject Current

{

get

{

return \_people[index];

}

}

}

People people = new People(3);

people.Add(new Person { Name = "Ali", Age = 22 });

people.Add(new Person { Name = "Sundus", Age = 21 });

people.Add(new Person { Name = "Hogi", Age = 12 });

foreach (var item in people)

{

//Cast from object to Person

Person person = (Person)item;

Console.WriteLine("Name:{0} Age:{1}", person.Name, person.Age);

}

IEnumerator <T>

IEnumerator <T> é uma interface genérica definida no namespace System.Collections.Generic. Possui métodos e propriedades que uma coleção de tipo seguro deve implementar para definir sua iteração.

public interface IEnumerator<out T> : IDisposable, IEnumerator

{

//element in the collection at the current position of the enumerator.

T Current { get; }

}

Como você pode ver, o IEnumerator <T> herda as interfaces IDisposable e IEnumerator. Portanto, um tipo que implementa IEnumerator<T> também deve implementar essas interfaces.

classPerson

{

publicstring Name { get; set; }

publicint Age { get; set; }

}

classmyGenericList<T> : IEnumerable<T>

{

T[] list;

int index = -1;

publicvoid Add(T obj)

{

if (++index < list.Length)

{

list[index] = obj;

}

}

public IEnumerator<T> GetEnumerator()

{

returnnew TEnum<T>(list);

}

IEnumerator IEnumerable.GetEnumerator()

{

returnthis.GetEnumerator();

}

public myGenericList(int size)

{

list = new T[size];

}

}

//Implement IEnumerator

classTEnum<T> : IEnumerator<T>

{

T[] \_list;

int index = -1;

public TEnum(T[] objs)

{

\_list = objs;

}

//Return if foreach can iterate to next index or not

publicbool MoveNext()

{

return (++index < \_list.Length);

}

publicvoid Reset()

{

index = -1;

}

//Get type-safe value of current array's index

//Its the Implementation of IEnumerator<T>

public T Current

{

get

{

return \_list[index];

}

}

//It's the implementation of 'IEnumerator'

object IEnumerator.Current

{

get

{

//return T Current

returnthis.Current;

}

}

// Example variable are used to implement Dispose()

bool m\_disposed = false;

IEnumerator<T> m\_reader;

bool m\_downcountEnumerators;

//It's the implementation of IDispose interface

publicvoid Dispose()

{

//Example of code to dispose un-needed resource

if (!m\_disposed)

{

// Only dispose the source enumerator if you are doing dynamic partitioning

if (!m\_downcountEnumerators)

{

m\_reader.Dispose();

}

m\_disposed = true;

}

}

}

myGenericList<Person> people = new myGenericList<Person>(3);

people.Add(new Person { Name = "Ali", Age = 22 });

people.Add(new Person { Name = "Sundus", Age = 21 });

people.Add(new Person { Name = "Hogi", Age = 12 });

foreach (var item in people)

{

//No need to cast

Console.WriteLine("Name:{0} Age:{1}", item.Name, item.Age);

}

O objetivo da seguinte implementação é vermos como os métodos do IEnumerator interagem com o foreach e a com a coleção. Primeiro vamos implementar o **IEnumerator** da nossa lista infinita:

publicclassInfinityNumbersListEnumerator : IEnumerator<int>

{

//Por que -1? O cursor começa antes da posição zero(Reset), então

//este valor deve ser zero após a primeira chamada do método MoveNext.

privateint \_current = -1;

publicint Current => \_current;

object IEnumerator.Current => \_current;

publicbool MoveNext()

{

\_current++;

returntrue;

}

publicvoid Reset()

{

\_current = -1;

}

publicvoid Dispose()

{

Reset();

}

}

Agora vamos para a implementação da lista que irá implementar IEnumerable<int>.

//Por enquanto vamos deixar o método Dispose assim mesmo.

//Agora vamos para a implementação da lista que irá implementar IEnumerable<int>.

publicclassInfinityNumbersList : IEnumerable<int>

{

privatereadonly InfinityNumbersListEnumerator \_enumerator;

public InfinityNumbersList()

{

\_enumerator = new InfinityNumbersListEnumerator();

}

public IEnumerator<int> GetEnumerator() => \_enumerator;

IEnumerator IEnumerable.GetEnumerator() => \_enumerator;

}

Com isso teremos um laço de repetição infinito em um foreach

InfinityNumbersList allNumbers = new InfinityNumbersList();

foreach (int number in allNumbers)

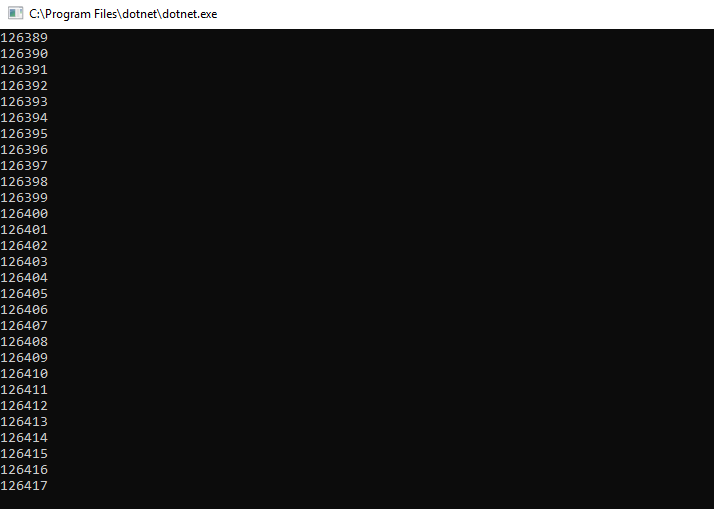
{

Console.WriteLine(number);

if (number >= 100)

break;

}



Três características básicas do IEnumerable/IEnumerator:

1. Ele não permite alteração nas coleções, funciona apenas como leitura (por isso que o LINQ gera novas coleções ao invés de causar efeitos colaterais);
2. Ele não fornece nenhuma informação sobre a coleção além do necessário para percorrê-la e dos oferecidos pelos métodos de extensão;
3. Ele não permite acesso à posições aleatórias da coleção, você só consegue percorrer de forma sequencial ou retornar ao início.

ICollection e ICollection <T>

ICollection e ICollection <T> são interfaces usadas para estender a definição de coleção personalizada.

ICollection

É a Interface base para todas as coleções que estão contidas dentro do namespace System.Collections. Define o tamanho, os enumeradores, e os métodos de sincronização para todas as coleções não genéricas.Direta ou indiretamente essas classes a implementa. Vamos dar uma olhada na definição do tipo ICollection interface que é composta por três propriedades Count, IsSynchronized e SyncRoot e um método CopyTo:

interfacepública ICollection: IEnumerable

{

// Obtém o número de elementos contidos no ICollection.

int Count { get; }

// Obtém um valor indicando se o acesso ao ICollection está sincronizado (thread safe).

bool IsSynchronized { get; }

// Obtém um objeto que pode ser usado para sincronizar o acesso ao ICollection.

object SyncRoot { get; }

// Copia os elementos da ICollection para uma matriz, iniciando em um índice System.Array específico.

void CopyTo(Array array, int index);

}

ICollection herda de IEnumerable. Portanto, você tem todos os membros da interface IEnumerable implementado em todas as classes que implementam a interface ICollection.

ICollection <T>

ICollection <T> é uma interface de segurança de tipo definida em System.Collections.Generic. Estende a funcionalidade de coleções genéricas.Quando olhamos para a versão genérica do ICollection, você vai perceber que ela não é exatamente o mesmo que o equivalente não-genérico, A nova definição de ICollection <T> tem algumasmétodos como Add, Remove, and Clear:

publicinterface ICollection<T>: IEnumerable<T>, IEnumerable

{

// Obtém o número de elementos contidos na `Generic.ICollection`.

int Count { get; }

// Obtém um valor indicando se o `Generic.ICollection` é somente leitura.

bool IsReadOnly { get; }

// Adiciona um item ao `System.Collections.Generic.ICollection`.

void Add(T item);

// Remove todos os itens da `Generic.ICollection`.

void Clear();

// Determina se o `System.Collections.Generic.ICollection` contém um valor específico.

bool Contains(T item);

// Copia os elementos da `Generic.ICollection` para uma matriz, iniciando no índice System.Array específico.

void CopyTo(Array T [], int arrayIndex);

// Remove a primeira ocorrência de um objeto da `Generic.ICollection`.

bool Remove(T item);

}

Sempre temos mais alguns métodos para adicionar, remover e/ou limpar uma coleção. A forma como a sincronização implementou também difere. Acredito que isso aconteceu porque a versão genérica dessa interface foi introduzida com o. NET 2.0 enquanto que a versão não genérica já foi introduzido em. NET 1.1.

IList e IList <T>

IList e IList<T> são interfaces que estendem a funcionalidade de um tipo de coleção personalizado.

IList

IList implementa ICollection e IEnumerable. Além disso, fornece definições de métodos para adicionar e remover elementos e para limpar a coleção. IList é uma interface definida em System.Collections. As implementações do IList se enquadram em três categorias:

1. Read-only: uma lista somente leitura não pode ser modificada.
2. Fixed-size: essa lista tem tamanho fixo, mas seus elementos podem ser editável
3. Variable-size: uma lista de tamanho variável permite adição, remoção e modificação de elementos.

Também fornece métodos para tratar o posicionamento dos elementos dentro do conjunto. Ela também fornece um indexador de objetos para permitir que o usuário acesse a coleção com colchetes como:

**lista[index].propriedades ou somente de forma generica como lista[index]**

publicinterfaceIList : ICollection, IEnumerable

{

// Obtém ou define o elemento no índice especificado.

objectthis[int index] { get; set; }

// Obtém um valor indicando se o IList tem um tamanho fixo.

IsFixedSize {get; }

// Obtém um valor indicando se o IList é somente leitura.

bool IsReadOnly { get; }

// Adiciona um item ao System.Collections.IList.

int Add(object value);

// Remove todos os itens do System.Collections.IList.

void Clear();

// Determina se o IList contém um valor específico.

bool Contains(object value);

// Determina o índice de um item específico no IList.

int IndexOf(object value);

// Insere um item na IList no índice especificado.

void Insert(int index, object value);

// Remove a primeira ocorrência de um objeto específico do IList.

void Remove(object value);

// Remove o item System.Collections.IList no índice especificado.

void RemoveAt(int index);

}

IList herda de ICollection e IEnumerable. Portanto, todos os membros das interfaces ICollection e IEnumerable devem ser implementados em todas as classes que implementam a interface IList.

IList<T>

IList<T> é uma interface definida em System.Collections.Generic. É usado para estender a coleção genérica personalizada. Não se parece exatamente com um IList não genérico. A nova definição de IList <T> é um pouco menor que o equivalente não genérico. Temos apenas alguns métodos novos para acessar uma coleção com posicionamento específico.

publicinterfaceIList<T> : ICollection<T>, IEnumerable<T>, IEnumerable

{

// Obtém ou define o elemento no índice especificado.

T this[int index] { get; set; }

// Determina o índice de um item específico no `Generic.IList`1.

int IndexOf(T item);

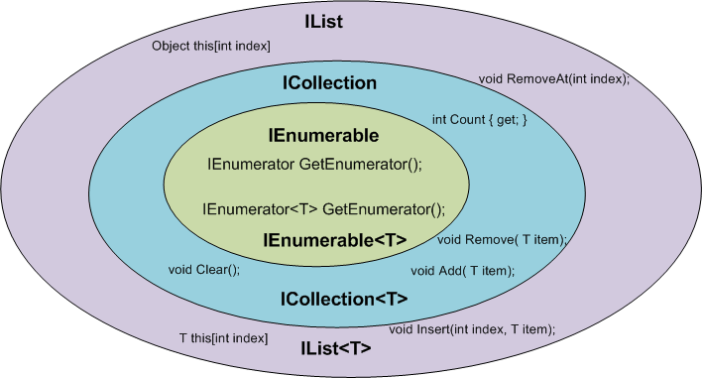
// Insere um item no `IGeneric.IList`1 no índice especificado.

void Insert(int index, T item);

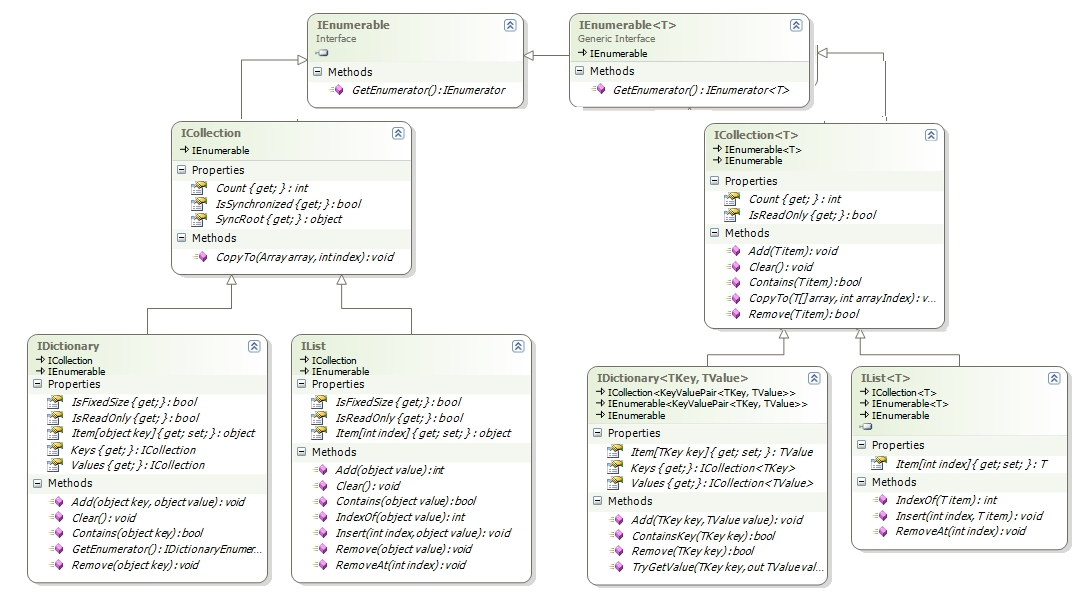
// Remove o item `SGeneric.IList` no índice especificado.

void RemoveAt(int index);

}



Como mencionamos antes, quando falamos sobre a interface ICollection<T>, existem alguns métodos a mais definidos na interface ICollection <T> do que na interface ICollection. Portanto a lista de membros da interface IList é um pouco menor do que o equivalente a não genérica. Então temos alguns novos métodos para acessar uma coleção com posicionamento específico.A hierarquia de interface deste namespace pode ser visto na figura abaixo:



Se você usar um tipo de interface mais “selada” como IEnumerable em vez de IList, você protege o seu código contra alterações significativas.  Se você usar o IEnumerable, o instanciador do método que você utiliza, pode fornecer qualquer objeto que implementa a interface IEnumerable.

Trata-se de quase todos os tipos de coleção da biblioteca de classes base e muitos tipos personalizados definidos. O código de chamada pode ser alterado no futuro, e seu código não vai quebrar tão facilmente como seria se você tivesse usado ICollection ou pior ainda IList.

Se você usar um tipo de interface mais amplo, como IList, você pode sofrer mais com os riscos de quebrar alterações no código. Se alguém quiser chamar o seu método com um objeto personalizado definido que apenas implementa IEnumerable ele simplesmente não vai funcionar e resultar em um erro de compilação.

ISet <T>

Em matemática, você costuma executar operações em um conjunto, como verificar se um conjunto é um subconjunto de outro conjunto, selecionar os elementos que dois conjuntos têm em comum ou que não têm em comum e combinar dois conjuntos. O HashSet implementa a interface ISet<T> que possui os membros que você pode encontrar abaixo.

publicinterfaceISet<T> : ICollection<T>, IEnumerable<T>, IEnumerable

{

// Adiciona um item ao ` System.Collections.Generic.HashSet<T>`

bool Add(T item);

// Remove todos os elementos da coleção especificada do objeto HashSet atual.`

void ExceptWith(IEnumerable<T> other);

// Modifica o atual objeto para conter os elementos nesse objeto e na coleção especificada.`

void IntersectWith(IEnumerable<T> other);

// Determina se um objeto HashSet é um subconjunto adequado da coleção especificada.`

bool IsProperSubsetOf(IEnumerable<T> other);

// Determina se um objeto HashSet é um superconjunto adequado da coleção especificada.`

bool IsProperSupersetOf(IEnumerable<T> other);

// Determina se um objeto HashSet é um subconjunto da coleção especificada.`

bool IsSubsetOf(IEnumerable<T> other);

// Determina se um objeto HashSet é um superconjunto da coleção especificada.`

bool IsSupersetOf(IEnumerable<T> other);

// Determina se o objeto HashSet atual e uma coleção especificada compartilham elementos comuns.`

bool Overlaps(IEnumerable<T> other);

// Determina se um objeto HashSet e a coleção especificada contêm os mesmos elementos.`

bool SetEquals(IEnumerable<T> other);

// Modifica o atual objeto HashSet para conter apenas os elementos presentes nesse objeto ou na coleção especificada, mas não os dois.`

void SymmetricExceptWith(IEnumerable<T> other);

// Modifica o objeto HashSet atual para conter todos os elementos presentes em si, a coleção especificada ou ambos.`

void UnionWith(IEnumerable<T> other);

}