Tags [Definidas pelo Editor]

Reflection

Construindo uma Engine de Serialização



Fone: (21) 93174772

MSN: juanplopes@gmail.com

Login DevMedia: juanplopes@gmail.com

**Juan Lopes**

me@juanlopes.net

Arquiteto de soluções pela Living Consultoria. Iniciou a carreira desenvolvendo aplicações biométricas em C++ e Java. Programa principalmente em C# desde 2007, mas desenvolve também em Ruby e Python. É entusiasta da comunidade Microsoft e participa de projetos open source.

*Sugestão de Ilustração: Espelhos, Estruturas (de preferência Wireframes)*

[A partir daqui inicia a parte **TEÓRICA** do artigo – Mantenha a Tag abaixo, será usada para contagem automática usando macros]

|  |
| --- |
| **Do que trata o artigo** |
| Este artigo fala sobre Reflection, uma técnica para observar e modificar a estrutura do código executável em tempo de execução. Explica a implementação no .NET, falando sobre Introspection, IL Emitting e ferramentas análogas, como Mono.Cecil e FxCop Introspection Model. |
| **Para que serve** |
| Em linguagens com tipagem estática, como C# e VB.NET, muitas vezes certas funcionalidades não são alcançáveis em tempo de compilação. Reflection consegue aumentar o dinamismo das linguagens por permitir a manipulação do código .NET após a compilação. |
| **Em que situação o tema é útil** |
| Reflection é bastante útil em situações onde é preciso conhecer e modificar o estado de um objeto dinamicamente. Perfeito para engines de serialização ou mecanismos de plugins. Além disso, a capacidade de IL Emitting permite que sejam criados métodos e classes em tempo de execução aumentando, em alguns casos, até performance do código. |

**Resumo do DevMan**

Reflection é a capacidade de um programa analisar e alterar sua estrutura ou o seu comportamento em tempo de execução. No .NET, temos suporte ao Reflection através das classes do namespace System.Reflection. Com elas, é possível obter informações sobre assemblies, classes e outras estruturas dinamicamente. Neste artigo, iremos mostrar como criar uma engine de serialização de arquivos com colunas pré-fixadas, usando reflection. Iremos também mostrar como resolver possíveis problemas de performance associados a essa técnica.

Você sabe o que acontece quando executa os seus programas? No C# e em diversas outras linguagens, o código-fonte precisa ser compilado para poder ser executado. Isto é, ele precisa ser transformado em uma representação mais amigável à máquina. Enquanto está executando, os programas compilados ficam carregados na memória, assim como os dados que possuem. Código e dados compartilham o mesmo espaço na memória. Sendo assim, pergunta-se: seria possível manipular o código em tempo de execução tal qual é possível com os dados?

A resposta é: sim! Por muito tempo, análise e modificação do código em tempo de execução tem sido utilizadas pelos programadores para dar características dinâmicas em seus programas. Entretanto, essas técnicas envolviam manipulação de endereços de memória de forma crua, o que geralmente podia levar a bugs indecifráveis e a sérias falhas de segurança.

Conforme as linguagens foram evoluindo, mecanismos foram sendo desenvolvidos para representar o código executável de forma estruturada, permitindo que um programa pudesse mais facilmente analisar sua própria representação através de estruturas de alto nível, reduzindo muito os possíveis bugs. Esses mecanísmos definem o que se conhece por Reflection (ou Reflexão). Para termos de nomenclatura, quando essa representação abrange o código executável, denomina-se reflexão computacional; quando abrange as estruturas definidas no programa (como classes e métodos), é chamada reflexão estrutural.

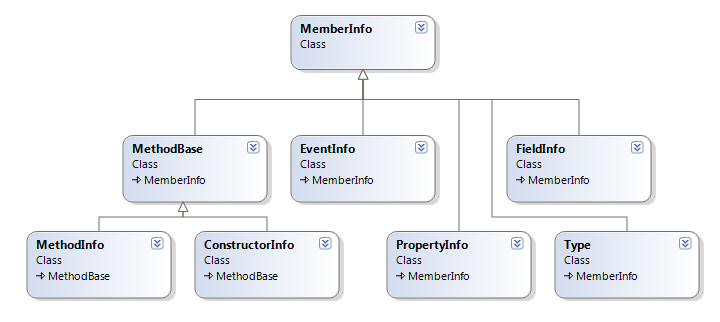
O .NET suporta reflexão estrutural amplamente através do namespace System.Reflection e, de uma forma mais tímida, também suporta reflexão computacional através de IL Emitting. Esse suporte é padronizado na plataforma para que seja possível analisar a estrutura de um programa independente da linguagem na qual ele foi escrito. Isto é, os metadados de um programa escrito em VB.NET, por exemplo, podem ser lidos por um programa escrito em C#, de forma transparente. Isso é crucial para permitir a interoperabilidade das linguagens.

Reflection está presente em muitos pontos da BCL e em muitas bibliotecas e frameworks do mercado. ORMs como NHibernate e Entity Framework usam reflection para alterar o estado de objetos em tempo de execução. Bibliotecas como Castle.DynamicProxy e a System.Xml.Serialization usam IL Emitting para gerar código em tempo de execução, seja com o objetivo de aumentar a performance ao longo tempo, ou até mesmo para estender a funcionalidade de certos objetos.

Reflection ou introspection?

Existe muita discussão com relação a nomenclatura no assunto. Algumas fontes dizem que Reflection é a capacidade de analisar e alterar o programa em tempo de execução, e que a análise somente é chamada de introspection. Outras fontes dizem que isso é irrelevante e que introspection, na verdade, tem a ver com a capacidade de um objeto conhecer seu próprio tipo dinamicamente. Iremos assumir a segunda definição para este artigo.

O .NET é capaz de introspection através do método GetType, presente na classe object e herdado por todos os outros objetos. Ele retorna uma instância da classe Type, porta de entrada para a maior parte das manipulações usando reflection. As estruturas principais do .NET estão representados por classes específicas no namespace System.Reflection. A **Figura 1** mostra algumas das principais classes desse namespace.



**Figura 1.** Hierarquia de classes que herdam de MemberInfo

Cada uma dessas classes permite obter informações sobre as estruturas de um programa. Uma instância de Type, por exemplo, possui o GetMethods, que é capaz de enumerar os métodos públicos de um certo tipo. A classe PropertyInfo permite que você obtenha ou altere o valor de uma propriedade em um objeto dinamicamente. Exemplos práticos serão discutidos a frente.

IL Emitting

Outra técnica importante de reflection é o uso das classes do namespace System.Reflection.Emit. Com ele, o programa é capaz de gerar classes e assemblies em tempo de execução, opcionalmente salvando em disco. O conjunto de classes desse namespace foi feito especialmente para ser usado por compiladores e script engines, mas tem grande utilidade em aplicações e bibliotecas comuns, pois por MSIL ser a linguagem mais básica da plataforma .NET, é possível alcançar grande flexibilidade ao escrever o código.

**Nota do DevMan**

IL é Intermediate Language, também conhecida como MSIL. É a linguagem de mais baixo nível na plataforma .NET. Análoga à linguagem Assembly nas plataformas x86 (não confundir com o conceito de assembly do .NET). As instruções em MSIL são bem mais cruas que em C#, por se tratar de uma linguagem mais próxima da máquina. Mesmo assim, é uma linguagem bem mais simples que Assembly puro, por ter conhecimento de estruturas de alto nível como métodos e objetos.

As principais classes usadas para este propósito são:

* AssemblyBuilder - usada para criar assemblies em runtime;
* DynamicMethod - permite criar um método descartável que pode ser recolhido pelo Garbage Collector;
* ILGenerator - usada para efetivamente emitir código executável;

Dentre essas, uma das classes que merece destaque é a DynamicMethod. Ela foi introduzida no .NET 2.0 e permite que métodos muito leves sejam criados em tempo de execução. Esses métodos podem ser chamados através de delegates com a mesma performance de um método normal. A principal vantagem desses métodos é a possibilidade de serem coletados pelo Garbage Collector.

Analogamente, o .NET Framework 4.0 introduziu o conceito de Collectible Assemblies. Até então, todo assembly carregado não podia ser descarregado até a finalização do AppDomain. Agora, um assembly criado a partir de IL Emitting pode ser marcado como "coletável" e ele poderá ser descartado após o uso. Isso evita que o consumo de memória da sua aplicação cresça indefinidamente ao utilizar bibliotecas que geram assemblies em runtime, como a System.Xml.Serialization.

Performance

Reflection é uma técnica que deve ser usada com cuidado, pois o ganho em flexibilidade e dinamismo na aplicação tem a um preço. A performance de algumas operações usando reflection é ruim. Chegando ao ponto de, por exemplo, uma invocação de método usando reflection custar cem vezes mais que se fosse escrita em tempo de compilação.

A notícia boa é que a maioria desses problemas tem solução, pois grande parte do custo está na construção dos objetos que são utilizados. Dessa forma, é bom fazer cache dos objetos com os quais se pretende trabalhar.

Para outros problemas, como o custo de invocação de métodos, existem soluções mais sofisticadas. Até o .NET 3.5, a única solução era criar DynamicMethods que invocassem os métodos a um custo quase igual (cerca de 50% maior) ao de uma invocação normal. É uma solução boa, e até hoje é a mais genérica possível

Com a introdução do conceito de lambda expressions, no .NET 3.5, foi possível a criação de métodos através das classes no namespace System.Linq.Expressions. A vantagem dessa solução é ser muito mais simples de implementar. Entretanto, para o ganho ser real, é preciso conhecer a assinatura do método em tempo de compilação, o que nem sempre é possível.

Alternativas

O namespace System.Reflection tem algumas limitações em sua implementação. Para sanar isso, algumas bibliotecas foram desenvolvidas, como a FxCop Introspection Model e a Mono.Cecil.

Uma das principais limitações é o fato do assembly precisar estar carregado na memória, para só então poder ser analisado. Junte isso com o fato de uma vez carregado, um assembly normal não poder ser descarregado, essa limitação causa diversos problemas com o ciclo de vida de algumas ferramentas.

Uma delas é o FxCop, que precisa analisar o código gerado para garantir que as políticas de construção foram seguidas. Entretanto, além de precisar de certas informações que a implementação padrão não tinha, ela ainda precisava ser capaz de analisar o assembly sem carregá-lo. Por esse motivo, existe o FxCop Introspection Model que visa justamente resolver esses problemas. Entretanto, ele não é amplamente utilizado principalmente pela falta de documentação da sua API pública.

A Mono.Cecil é uma biblioteca que faz parte do projeto Mono. Ela foi criada primariamente para dar suporte ao debugger da plataforma Mono, superando as limitações que a implementação padrão tinha. Essa biblioteca tornou-se tão poderosa que acabou ganhando o gosto popular e sendo usada em diversos outros projetos. Uma das grandes vantagens que ela trás é a capacidade de extrair o código executável bruto dos assemblies e carregar informações de debugging. Na plataforma .Net, essas informações não são extraíveis sem usar código não-gerenciado.

**Nota do DevMan**

Mono é um projeto patrocinado pela Novell, que busca desenvolver uma versão da Common Language Runtime para outras plataformas além das suportadas pela Microsoft. É, basicamente, uma versão do .NET para sistemas Unix-based. Por estar reimplementando a BCL por completo, o projeto poderia adicionar funcionalidades à System.Reflection, mas isto tornaria a biblioteca deles ligeiramente incompatível com a BCL da Microsoft. Por isso surgiu o Mono.Cecil.

[CHECKPOINT]

Conhecendo o namespace System.Reflection

Antes de partir para um exemplo mais pesado, vamos conhecer melhor as classes desse namespace. Considere, para os exemplos a seguir, a classe da **Listagem 1**.

**Listagem 1.** Classe de exemplo

public class Foo

{

public int FieldA;

[Description("propriedade A")]

public int PropertyA { get; set; }

public string Bar(int parameter)

{

return "abc";

}

}

Como dito anteriormente, a porta de entrada para grande parte das operações em reflection é a classe Type. Existem duas formas principais de obter uma instância desta classe. Uma é usando o operador typeof, a outra é invocando o método GetType, em qualquer objeto.

var foo = new Foo();

var type1 = foo.GetType();

var type2 = typeof(Foo);

Uma vez com esta instância em mãos, várias informações podem ser obtidas deste tipo. Todos os elementos dentro de uma class ou struct podem ser abstraídos na classe mais básica MemberInfo. Caso seja necessário retornar somente os métodos, por exemplo, GetMethods pode ser útil:

Console.WriteLine("Todos os métodos da classe Foo:");

foreach (var method in typeof(Foo).GetMethods())

Console.WriteLine(method.Name);

A maior parte das estruturas do .NET podem ser decoradas por atributos. Atributos são classes que herdam de Attribute e são tratadas de forma especial pelos compiladores C# e VB.NET. É possível listar os atributos de uma classe ou membro através do método GetCustomAttributes. Na **Listagem 2**, estamos obtendo uma propriedade pelo seu nome, e obtendo o primeiro atributo do tipo DescriptionAttribute.

**Listagem 2.** Listando os atributos de um membro

var propA = typeof(Foo).GetProperty("PropertyA");

var attribute = propA

.GetCustomAttributes(typeof(DescriptionAttribute), true)

.OfType<DescriptionAttribute>().First();

Console.WriteLine(attribute.Description);

Também é possível, tendo uma instância de MethodInfo em mãos, invocar o método que ela representa. Essa é um dos recursos mais utilizados em reflection, pois assim é possível interagir com os objetos de forma ativa.

**Listagem 3.** Invocando métodos dinamicamente

var type = typeof(Foo);

var method = type.GetMethod("Bar");

var foo = new Foo();

var resultado = method.Invoke(foo, new object[] { 3 });

Console.WriteLine(resultado);

Na **Listagem 3**, estamos obtendo uma instância do método Bar, que existe na classe Foo. Ao invocar esse método em cima de uma nova instância de Foo, passando o parâmetro 3, o resultado deve ser “abc”.

Medindo a performance do Invoke

Já sabemos como invocar métodos dinâmicamente. Podemos realizar um microbenchmark para verificar o quão lento é, de fato, usar invocações via reflection na aplicação.

**Listagem 4.** Microbenchmark de MethodInfo.Invoke

var foo = new Foo();

var bar = typeof(Foo).GetMethod("Bar");

var s = Stopwatch.StartNew();

for (int i = 0; i < 5000000; i++)

foo.Bar(i);

Console.WriteLine("Sem reflection: {0}", s.Elapsed);

s.Restart();

for (int i = 0; i < 5000000; i++)

bar.Invoke(foo, new object[] { i });

Console.WriteLine("Com reflection: {0}", s.Elapsed);

Na **Listagem 4**, estamos medindo o tempo de execução de 5 milhões de chamadas ao método Bar, da classe Foo. No primeiro loop, fazemos isso sem usar reflection, com invocação direta. No segundo loop, utilizamos MethodInfo.Invoke para realizar a chamada, passando o mesmo parâmetro. Considerando que o corpo do método Bar é desprezível (apenas um return), podemos analisar os resultados

Sem reflection: 00:00:00.0315970

Com reflection: 00:00:04.1397186

A diferença no tempo de execução é grande. Neste caso, foi mais de cem vezes mais lento utilizar Invoke em vez de chamadas estáticas. Felizmente existem formas de otimizar esse código. A mais simples delas é construindo lambda expressions em tempo de execução.

**Listagem 5.** Otimizando a chamada ao método com lambda expressions

var foo = new Foo();

var bar = typeof(Foo).GetMethod("Bar");

var param = Expression.Parameter(typeof(int));

var call = Expression.Call(Expression.Constant(foo), bar, param);

var lambda = Expression.Lambda<Func<int,string>>(call, param).Compile();

var s = Stopwatch.StartNew();

for (int i = 0; i < 5000000; i++)

foo.Bar(i);

Console.WriteLine("Sem reflection: {0}", s.Elapsed);

s.Restart();

for (int i = 0; i < 5000000; i++)

lambda.Invoke(i);

Console.WriteLine("Com reflection: {0}", s.Elapsed);

Na **Listagem 5**, o procedimento é o mesmo, porém estamos criando uma lambda expression para invocar o método Bar. O custo de invocação desta lambda expression é bem menor que uma chamada ao MethodInfo.Invoke, pois este método foi compilado dinamicamente. Por isso, os tempos de execução são bem menores.

Sem reflection: 00:00:00.0320835

Com reflection: 00:00:00.1428973

**Nota do DevMan**

O exemplo da Listagem 4 utiliza a construção de expression trees em tempo de execução. Este recurso permite que métodos sejam criados dinamicamente sem o uso de IL Emitting (o que seria bem mais complexo, apesar de mais flexível). Com elas, é possível representar, entre outras coisas, a invocação de um método através da função Expression.Call, que é utilizada no exemplo.

Como se pode perceber, o tempo ainda não é o mesmo, pois invocar uma lambda expression nunca será como invocar um método usando binding estático, mas é um valor muito mais plausível que o obtido no teste anterior.

Um fato importante a se observar é que atribuir valores a uma propriedade em uma classe conta como reflection invoke. Isso acontece porque propriedades são atribuidas através de chamada ao método set da mesma. Para otimizar a chamada a propriedades usando lambda expressions, existe código análogo ao anterior.

**Listagem 6.** Otimizando o método set em propriedades

var propA = typeof(Foo).GetProperty("PropertyA");

var foo = Expression.Parameter(typeof(Foo));

var value = Expression.Parameter(typeof(int));

var assign = Expression.Assign(Expression.Property(foo, propA), value);

var lambda = Expression.Lambda<Action<Foo, int>>(assign, foo, value).Compile();

var instance = new Foo();

lambda(instance, 42);

Console.WriteLine(instance.PropertyA);

Na **Listagem 6**, estamos criando uma lambda expression que atribui um valor a uma propriedade. Ela recebe uma instância de Foo e um inteiro como parâmetros e atribui o inteiro à propriedade PropertyA em Foo, fazendo isso com performance melhor que usando o método SetValue da classe PropertyInfo.

Um exemplo mais prático

Partindo dos exemplos conceituais para algo mais prático, iremos construir neste artigo uma engine de serialização personalizada, baseada nos antigos arquivos de colunas com tamanhos prefixados.

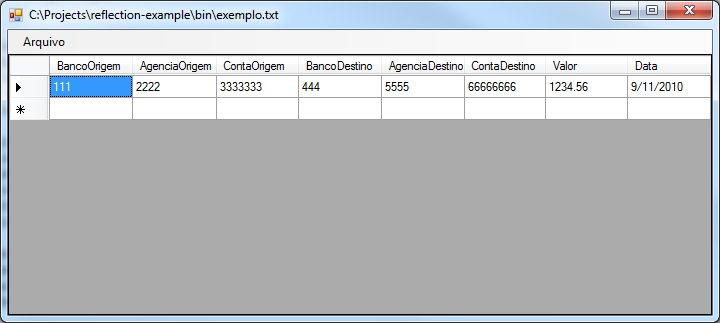
Esses arquivos geralmente possuem um documento chamdo layout, que define o que significa cada campo e quais colunas eles compreendem. Arquivos assim eram utilizados para fazer integração de sistemas, que geralmente eram trocados através de FTP ou qualquer outro protocolo de transferência.

Neste exemplo, iremos construir uma engine que seja capaz de escrever arquivos em qualquer layout e seja extensível. Para exemplificar, iremos tomar como base o layout (bastante simplificado) de uma remessa bancária definido pela estrutura na **Tabela 1**.

|  |  |  |
| --- | --- | --- |
| **Campo** | **Coluna inicial** | **Coluna final** |
| Banco de Origem | 1 | 5 |
| Agência de Origem | 6 | 10 |
| Conta de Origem | 11 | 20 |
| Banco de Destino | 21 | 25 |
| Agência de Destino | 26 | 30 |
| Conta de Destino | 31 | 40 |
| Valor | 41 | 50 |
| Data | 51 | 60 |

**Tabela 1.** Layout do arquivo de exemplo

Com esse layout, definimos um arquivo de texto, onde cada linha terá 60 caracteres, e cada caractere pertencerá a um campo em específico. Com isso em mãos, podemos escrever uma interface simples, capaz de editar o arquivo através do .NET. Essa interface está definida na **Figura 2**.



**Figura 2.** Interface de exemplo

Desenvolvendo a Biblioteca de Serialização

Para dar suporte a essa interface, é preciso desenvolver numa biblioteca que consiga serializar e desserializar objetos a partir de linhas do arquivo. E nesse ponto que reflection será de extrema importância. Pois o objetivo é que seja possível escrever uma classe como a da **Listagem 7** e uma biblioteca seja capaz de popular objetos dessa classe com valores vindos de um arquivo ou escrever um arquivo com valores que venham de objetos da classe.

**Listagem 7.** Classe Remessa

public class Remessa

{

[Field(1, 5)]

public int BancoOrigem { get; set; }

[Field(6, 10)]

public int AgenciaOrigem { get; set; }

[Field(11, 20)]

public int ContaOrigem { get; set; }

[Field(21, 25)]

public int BancoDestino { get; set; }

[Field(26, 30)]

public int AgenciaDestino { get; set; }

[Field(31, 40)]

public int ContaDestino { get; set; }

[Field(41, 50)]

public decimal Valor { get; set; }

[Field(51, 60)]

public DateTime Data { get; set; }

}

Cada propriedade na classe Remessa é decorada com um atributo Field, que define onde começa e onde termina cada campo. O problema é que a forma de transformar os valores desses campos em strings e de volta aos seus tipos originais pode variar de uso para uso. Essa possibilidade de customização define o primeiro ponto de extensão para a biblioteca.

Para permitir essa extensibilidade, vamos definir uma abstração que diga como modificar uma linha com o valor de uma propriedade e como ler para uma propriedade o valor em uma linha. Para isso, define-se a interface IField.

public interface IField

{

object ReadFrom(string line, Type type);

void WriteTo(StringBuilder line, object value);

}

Esta inteface é uma abstração de como uma classe que serializa um campo em específico deve se comportar. Ela pode, por exemplo, ser usada no futuro para definir um campo que deve ser serializado como Base64, por exemplo.

A primeira implementação da interface IField – a mais simples possível – é definida pela classe FieldAttribute.

**Listagem 8.** Classe FieldAttribute

[AttributeUsage(AttributeTargets.Property)]

public class FieldAttribute : Attribute, IField

{

int start, end;

int Size { get { return end - start; } }

public FieldAttribute(int start, int end)

{

this.start = start - 1;

this.end = end;

}

public virtual object ReadFrom(string line, Type type)

{

//Código na Listagem 9

}

public virtual void WriteTo(StringBuilder builder, object value)

{

//Código na Listagem 10

}

}

Na **Listagem 8**¸ definimos uma classe FieldAttribute, herdando de Attribute. Decorando a classe com AttributeUsage, definimos que só é possível utilizar este FieldAttribute em membros do tipo propriedade. Isso é verificado em tempo de compilação.

FieldAttribute ainda implementa a interface IField. Ela recebe como parâmetros a coluna inicial e final onde aquele campo será escrito em cada linha. Estamos assumindo que as colunas são numeradas começando com 1 e, para facilitar os cálculos, subtraimos 1 do parâmetro start no construtor.

**Listagem 9.** Método ReadFrom

public virtual object ReadFrom(string line, Type type)

{

var stringValue = line.Substring(start, Math.Min(line.Length - start, Size));

stringValue = stringValue.Trim('\0', ' ');

return ConvertFromString(stringValue, type);

}

protected virtual object ConvertFromString(string stringValue, Type type)

{

return Convert.ChangeType(stringValue, type, CultureInfo.InvariantCulture);

}

Na **Listagem 9**, implementamos os método ReadFrom, que recebe uma linha e um tipo de destino e retorna um objeto convertido do valor para esse tipo. Para extrair o valor, utilizamos o método Substring, partindo da primeira coluna do campo e obtendo no máximo o tamanho do campo, garantindo não passar do fim da string. É feito um Trim para remover os espaços e caracteres nulos a mais.

O método ConvertFromString serve de ponto de extensão para classes que queiram herdar de FieldAttribute, para não ter que reimplementar a lógica de extração da substring. Bastaria sobrescrever o método ConvertFromString.

**Listagem 10.** Método WriteTo

public virtual void WriteTo(StringBuilder builder, object value)

{

var stringValue = ConvertToString(value);

builder.Length = Math.Max(builder.Length, end);

WriteString(builder, stringValue);

WriteSpaces(builder, stringValue);

}

protected virtual string ConvertToString(object value)

{

return Convert.ToString(value, CultureInfo.InvariantCulture);

}

private void WriteString(StringBuilder builder, string stringValue)

{

for (int i = 0; i < Math.Min(stringValue.Length, Size); i++)

builder[start + i] = stringValue[i];

}

private void WriteSpaces(StringBuilder builder, string stringValue)

{

for (int i = start + stringValue.Length; i < end; i++)

builder[i] = ' ';

}

O método WriteTo, na **Listagem 10**, tem como objetivo receber um objeto, transformá-lo para sua representação em string e escrevê-lo no StringBuilder. Mais uma vez, deixamos um ponto de extensão que é o método ConvertToString, que transforma o objeto em sua representação textual. Após escrever a string, preenchemos as colunas que sobraram com espaços em branco.

Façades para compor a funcionalidade

**Nota do DevMan**

Façade é um design pattern que visa simplificar a interface de uma biblioteca complexa, aumentar sua legibilidade e encapsular algumas tarefas comuns. Nos exemplos do artigo, estamos encapsulando as classes do namespace System.Reflection para expor somente o que precisamos delas, combinadas com a funcionalidade do exemplo em si.

Até aqui, o exemplo não está utilizando reflection, apenas definindo como os campos serão serializados. Entretanto, agora é preciso criar uma classe que seja capaz de envolver a funcionalidade de um objeto PropertyInfo (otimizando-a), obter seu atributo de declaração, disponibilizando métodos para alterar valores em um objeto genérico.

**Listagem 11.** Façade para a classe PropertyInfo

public class PropertyWrapper

{

private PropertyInfo property;

private IField fieldDescriptor;

private Func<object, object> getFrom;

private Action<object, object> setTo;

public bool IsValid { get { return fieldDescriptor != null; } }

public string Name { get { return property.Name; } }

public PropertyWrapper(PropertyInfo property)

{

this.property = property;

this.fieldDescriptor = ExtractFieldAttributeFrom(property);

if (IsValid)

Optimize(property);

}

private static IField ExtractFieldAttributeFrom(PropertyInfo property)

{

return property.GetCustomAttributes(typeof(IField), true)

.OfType<IField>().FirstOrDefault();

}

private void Optimize(PropertyInfo property)

{

//Código na listagem 12

}

public void WriteTo(StringBuilder builder, object instance)

{

fieldDescriptor.WriteTo(builder, getFrom(instance));

}

public void ReadFrom(string line, object instance)

{

setTo(instance,

fieldDescriptor.ReadFrom(line, property.PropertyType));

}

}

Na **Listagem 11**, definimos a classe PropertyWrapper, que nada mais é que uma façade para simplificar o uso de um objeto de PropertyInfo. Ela provê métodos análogos aos da interface IField, exceto pelo fato de que ela já tem conhecimento sobre a propriedade em si, então ela pode atualizar o valor já na instância do objeto desejado. O método ExtractFieldAttribute faz o papel de obter a primeira instância que conseguir de um atributo que herde de IField, e esse é o principal ponto de extensibilidade, pois qualquer classe pode servir a esse propósito, desde que herde da interface correta. A implementação do método Optimize foi omitida pois é necessário que ele seja explicado com cuidado.

**Listagem 12.** Otimizador par Get e Set de propriedades

private void Optimize(PropertyInfo property)

{

var target = Expression.Parameter(typeof(object));

var value = Expression.Parameter(typeof(object));

var prop = Expression.Property

(Expression.Convert(target, property.DeclaringType), property);

this.getFrom = Expression.Lambda<Func<object, object>>(Expression.Convert(

prop, typeof(object)), target).Compile();

this.setTo = Expression.Lambda<Action<object, object>>(Expression.Assign(

prop, Expression.Convert(value, property.PropertyType)), target, value).Compile();

}

O código na **Listagem 12** cria os métodos otimizados para Get e Set da propriedade que estamos envolvendo. Ele utiliza o método de árvore de expressões, conforme apresentado anteriormente. Um cuidado que foi tomado é o de fazer expressões para os casts necessários, pois como os tipos dos parâmetros são sempre object, as expressions precisam fazer os casts para os tipos apropriados. Isto é feito através do método Expression.Convert.

Neste ponto, a biblioteca é capaz de receber uma instância de PropertyInfo e extrair todos os metadados necessários para tornar possível alterar valores em objetos de um determinado tipo. Iremos agora construtir um outro façade, desta vez sobre a classe Type.

**Listagem 13.** Façade para a classe Type

public class TypeWrapper

{

Type type;

IList<PropertyWrapper> properties;

Func<object> activator;

public TypeWrapper(Type type)

{

this.type = type;

this.properties = ExtractPropertiesFrom(type);

this.activator = Expression.Lambda<Func<object>>(

Expression.New(type)).Compile();

}

private static IList<PropertyWrapper> ExtractPropertiesFrom(Type type)

{

return type.GetProperties()

.Select(x => new PropertyWrapper(x))

.Where(x => x.IsValid).ToList();

}

public string WriteFrom(object instance)

{

var builder = new StringBuilder();

foreach (var property in properties)

property.WriteTo(builder, instance);

return builder.ToString();

}

public object ReadFrom(string line)

{

var instance = activator();

foreach (var property in properties)

property.ReadFrom(line, instance);

return instance;

}

}

A classe na **Listagem 13** encapsula a funcionalidade da classe Type. Já na sua construção, ela extrai todas as propriedades do tipo passado por parâmetro, envolvendo-as em instâncias de PropertyWrapper. Ainda no construtor, um método otimizado para construção do objeto via reflection também é criado, utilizando o mesmo conceito aplicado na classe PropertyWrapper.

O método WriteFrom recebe uma instância de um objeto preenchido e ordena a cada propriedade que escreva seu valor em um StringBuilder. Ele retorna a string construída no final.

O método ReadFrom faz o contrário. Recebe uma string e pede que cada propriedade leia o valor que lhe é devido em uma instância específica. E retorna a instância.

Escrevendo a API pública

Com essas abstrações, temos toda a funcionalidade necessária para transformar string em objeto e vice-versa. Iremos definir uma classe serializadora, simplesmente para facilitar os usos mais comuns, como a leitura de dados de arquivos, rede ou memória. O .NET já possui excelentes abstrações baseadas na classe Stream. Escolhemos, para este exemplo, as classes TextWriter e TextReader, que com seus métodos WriteLine e ReadLine, provêem a funcionalidade exata que precisamos para o exemplo (a escrita e a leitura de uma linha por vêz).

**Listagem 14.** Classe serializadora

public class FixedColumnsSerializer

{

TypeWrapper type;

public FixedColumnsSerializer(Type type)

{

this.type = new TypeWrapper(type);

}

public void WriteAll(TextWriter writer, IEnumerable objs)

{

foreach (var obj in objs)

Write(writer, obj);

}

public void Write(TextWriter writer, object obj)

{

writer.WriteLine(type.WriteFrom(obj));

}

public IEnumerable ReadAll(TextReader reader)

{

var list = new ArrayList();

for (var obj = Read(reader); obj != null; obj = Read(reader))

list.Add(obj);

return list;

}

public object Read(TextReader reader)

{

var line = reader.ReadLine();

if (line == null) return null;

return type.ReadFrom(line);

}

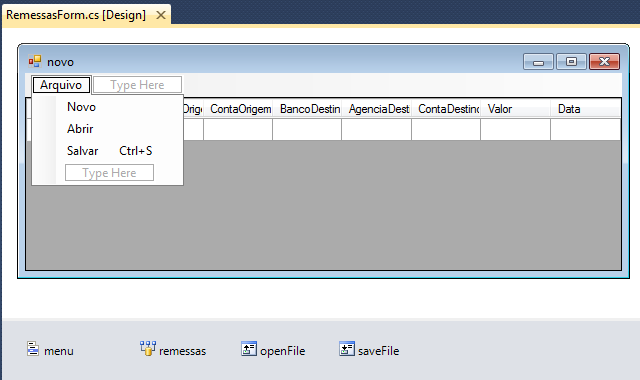
}

Com a classe na **Listagem 14**, é possível ler textos de arquivos, rede ou qualquer outro meio que suporte ser encapsulado uma instância da classe Stream. O método Read lê uma linha de um TextReader e retorna um objeto. O método Write, escreve o objeto passado em uma linha do TextWriter. Os métodos ReadAll e WriteAll são seus análogos para muitos registros.

Assim, a funcionalidade da biblioteca está completa. Já é possível ler e escrever dados somente utilizando instâncias de classes com propriedades decoradas com atributos IField.

Escrevendo a Interface Gráfica

Manteremos a interface de exemplo o mais simples possível. Definimos um DataGridView, um DataSource e alguns elementos de interface, como pode ser visto na **Figura 3**.

****

**Figura 3.** Design da interface de exemplo

Para carregar dados de um arquivo no DataGridView, como o da interface de exemplo, bastaria chamar o método ReadAll utilizando um StreamReader, obtido, por exemplo, com um File.OpenText. Analogamente, para salvar, é possível utilizar o método WriteAll, utilizando File.CreateText.

**Listagem 15.** Eventos para abrir e salvar arquivos

private void abrirMenu\_Click(object sender, EventArgs e)

{

if (openFile.ShowDialog() == DialogResult.OK)

Abrir(openFile.FileName);

}

private void Abrir(string fileName)

{

openFilePath = fileName;

this.Text = openFilePath;

using (var file = File.OpenText(openFilePath))

remessas.DataSource = serializer.ReadAll(file);

}

private void salvarMenu\_Click(object sender, EventArgs e)

{

grid.CommitEdit(DataGridViewDataErrorContexts.Commit);

if (openFilePath == null && saveFile.ShowDialog() == DialogResult.OK)

openFilePath = saveFile.FileName;

if (openFilePath != null)

Salvar(openFilePath);

}

private void Salvar(string fileName)

{

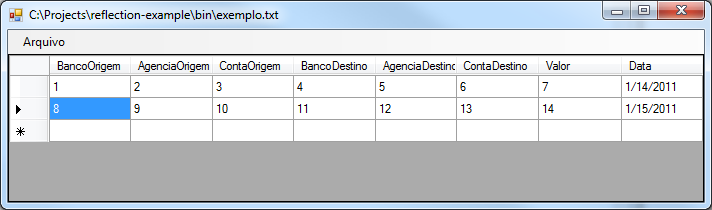
using (var file = File.CreateText(fileName))

serializer.WriteAll(file, remessas.DataSource as IEnumerable);

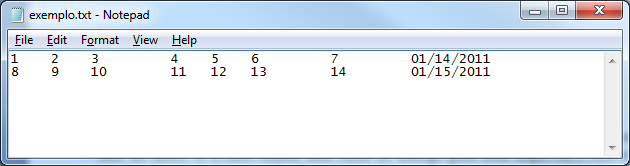
Abrir(fileName);

}

Na **Listagem 15**, os métodos abrirMenu\_Click e Abrir dizem respeito a como o arquivo vai ser lido do disco. O evento chama uma caixa de diálogo para abrir arquivos, e caso o usuário não cancele, é carregado o arquivo através do método Abrir. Analogamente, o menu salvar chama uma caixa de diálogo para salvar. Porém, antes disso, ele confirma as mudanças em curso na grid, para evitar valores meio-salvos. Se o salvamento for bem sucedido, ele abre o arquivo recém-salvo para edição.



**Figura 4.** Interface preenchida com valores de exemplo



**Figura 5.** Arquivo salvo com os valores preenchidos

Extensibilidade

Como definido anteriormente, o ponto principal de extensibilidade da biblioteca desenvolvida neste artigo é a interface IField. Entretanto, a classe FieldAttribute já implementa grande parte da funcionalidade necessária para obtenção de dados em colunas fixas. Geralmente uma classe só precisará implementar IField diretamente se desejar implementar uma regra muitíssimo específica que não tenha sido contemplada pela implementação padrão. Para maior parte dos casos, basta herdar de FieldAttribute e sobrescrever os métodos ConvertFromString e ConvertToString.

Suponha que precisemos desenvolver uma implementação de campo que leia e salve datas num formato específico (YYYYMMDD, por exemplo). Se a biblioteca não fosse extensível nesse ponto, precisaríamos alterar e recompilar a biblioteca. Além disso, deixamos propositalmente alguns pontos de extensão na classe FieldAttribute, para que a implementação referida pudesse ser simples.

**Listagem 16.** Atributo para formatação de datas

[AttributeUsage(AttributeTargets.Property)]

public class DateFieldAttribute : FieldAttribute

{

string format;

public DateFieldAttribute(int start, int end, string format)

: base(start, end)

{

this.format = format;

}

protected override object ConvertFromString(string stringValue, Type type)

{

return DateTime.ParseExact(stringValue, format, CultureInfo.InvariantCulture);

}

protected override string ConvertToString(object value)

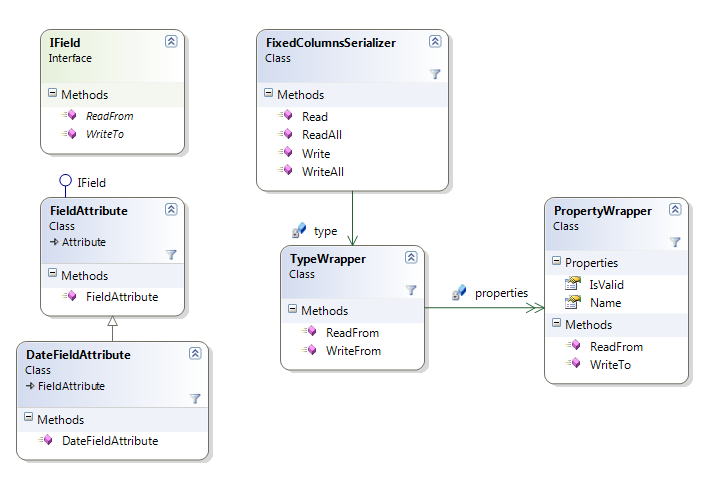
{

return ((DateTime)value).ToString(format, CultureInfo.InvariantCulture);

}

}

O código na **Listagem 16** foi possível pois os métodos ConvertFromString e ConvertToString na classe FieldAttribute foram marcados como virtuais e são os métodos mais apropriados para definir como transformar objetos em strings e vice-versa.



**Figura 6.** Diagrama de classes final da biblioteca

Conclusão

Reflection é uma ferramenta muito poderosa, que permite manipulações avançadas de objetos e tipos no .NET. Sem ela, muitas das grandes ferramentas que temos não seriam possíveis. Entretanto, é preciso ter parcimônia em recorrer a reflection. Quando dominada, é geralmente tentador usá-la para resolver todos os problemas. Mas tenha em mente que, apesar de poderoso, um código que usa reflection é mais propenso a erros e, caso não sejam tomados os cuidados necessários, pode ser muito mais lento que um código normal.

Dado esse conselho, sugerimos ao leitor que procure pontos de repetições no código que seriam beneficiados com o uso de reflection. Experimente eliminar essas repetições, e veja como reflection pode ser uma peça fundamental na escrita de códigos mais dinâmicos.

**Links**

Reflection

<http://msdn.microsoft.com/en-us/library/cxz4wk15.aspx>

IL / MSIL

<http://en.wikipedia.org/wiki/Common_Intermediate_Language>

Mono.Cecil

<http://www.mono-project.com/Cecil>

Customising FxCop by using Introspection

<http://www.codeproject.com/KB/cs/FxCop135.aspx>

Twitter do autor

<http://twitter.com/juanplopes>

**Saiba Mais**

Artigo – .Net Magazine 81 – Intermediate Language – Elemar JR

<http://www.devmedia.com.br/articles/viewcomp.asp?comp=18960>

Artigo – .Net Magazine 73 – Serialização – Vladimir Rech

<http://www.devmedia.com.br/articles/viewcomp.asp?comp=16847>

Artigo – .Net Magazine 78 – Mono

<http://www.devmedia.com.br/articles/viewcomp.asp?comp=18215>

Artigo – .Net Magazine 71 – Dynamic Types – Vinicius Quaiato

<http://www.devmedia.com.br/articles/viewcomp.asp?comp=16159>

Artigo – .Net Magazine 78 – Spring.Net – Fábio Augusto Falavinha

<http://www.devmedia.com.br/articles/viewcomp.asp?comp=18220>

Artigo – .Net Magazine 73 – Tipos no .Net – Carlos Alberto Lopes

<http://www.devmedia.com.br/articles/viewcomp.asp?comp=16843>

Artigo – .Net Magazine 76 – Design Patterns – Fabio Gouw

<http://www.devmedia.com.br/articles/viewcomp.asp?comp=17691>

Artigo – .Net Magazine 69 – AOP e Design Patterns na prática – Nelson Rodrigo Lombardi Basseto

<http://www.devmedia.com.br/articles/viewcomp.asp?comp=15239>

Artigo – .Net Magazine 77 – Criando um Sistema Plugável

<http://www.devmedia.com.br/articles/viewcomp.asp?comp=17992>

Artigo – .Net Magazine 73 – MEF Parte 1 – Marcello Palladino

<http://www.devmedia.com.br/articles/viewcomp.asp?comp=16846>