BOSCH
Invented for life

Docupedia Export

Author: Goncalves Donathan (CtP/ETS)
Date: 30-May-2023 14:59

Table of Contents

1 Restrições Genéricas

Em C# básico foram mostrados os tipos genéricos e suas capacidades. Agora vamos ir um pouco mais afundo nas capacidades desta feature do C#. Primeiramente, considere o seguinte código:

```
 1  public class A
 2  {
 3
 4  }
 5
 6  public class A1 : A
 7  {
 8
 9  }
10
11  public class A2 : A
12  {
13
14
15  }
16
17  public class B<T>
18      where T : A
19  {
20      public T Result { get; set; }
21  }
```

A classe genérica B contém agora uma restrição genérica definido com a palavra 'where'. Ela garante que o tipo T será/herdará de A. Assim, B<A>, B<A1> e B<A2> são tipos possíveis para B, enquanto isso, B<int> resultará em um erro. Além de um tipo você pode especificar vários tipos de outras restrições:

```
// C1<int> c1; Error
C1<C3<int>> c2; // Ok
C4<int> c3; // Ok
C5<int> c4; // Ok
C8<MemoryStream, Stream> c5; // Ok

public class C1<T> where T : class { }
public class C2<T> where T : class? { }
public class C3<T> where T : struct { }
public class C4<T> where T : notnull { } // Não deve ser anulável
public class C5<T> where T : unmanaged { } // Deve ser um tipo gerenciável, structs
puras
public class C6<T, U> where T : U { }
public class C7<T> where T : new()
{
    public T Get()
    {
        return new T();
    }
}
public class C8<T, U> where T : U, new() where U : class
{
    public U Value { get; set; } = new T();
}
```

Logo veremos alguns exemplos interessantes e complexos onde podemos aplicar restrições genéricas.

2 Variância Genérica

Sabemos que podemos colocar um objeto em uma variável do tipo da classe mãe. Isso é comum em Orientação a Objetos. Mas e com genéricos? Poderíamos por uma lista de int em uma variável de coleção de objetos? Afinal Lista é uma coleção e int é um objeto. Na verdade isso funciona em partes. A conversão entre object e int não é tão direta. Isso se dá graças ao boxing do int, mas de outros tipos de referência isso é totalmente possível. Observe:

```
object obj = 8; // Variância comum

var cobj = new C<Funcionario>();

A<Funcionario> a1 = cobj; // Esperado
B<Funcionario> b1 = cobj; // Esperado

B<Pessoa> b2 = cobj; // Covariância genérica
A<Aprendiz> a2 = cobj; // Contravariância genérica

a2.Value = new Aprendiz() { Nome = "Xipsita" };
// a2.Value = new Mae(); // Erro: a2 espera objeto do tipo filha
// Avo avo = a2.Value; // Erro: a2 não possui Get
Pessoa avo = b2.Value;
Console.WriteLine(avo.Nome);

var list = new List<Funcionario>();
IEnumerable<Pessoa> collection = list;

Func<Funcionario, Funcionario> func = x => x;
Func<Aprendiz, Pessoa> func2 = func;
Pessoa res = func2(new Aprendiz());

public class Pessoa
{
    public string Nome { get; set; }
}
public class Funcionario : Pessoa { }
public class Aprendiz : Funcionario { }

// in e out válidos a interfaces e delegados
public interface A<in T> // T só pode ser usado em parâmetros
{
    // T Get(); Erro
    T Value { set; }
}

public interface B<out T> // T só pode ser usado em retornos
{
    // void Set(T value); Erro
    T Value { get; }
}

public class C<T> : A<T>, B<T>
{
    public T Value { get; set; }
}
```

3 Reflexão

Reflexão é a capacidade das linguagens de programação de ler sua própria estrutura e até modificá-la. No C# a programação reflexiva é possível e extremamente poderosa.

```
1   using System.Reflection; // Necessário apenas para o GetRuntimeFields
2
3   var type = typeof(Funcionario);
4
5   Console.WriteLine($"Class {type.Name}:");
6   foreach (var prop in type.GetMembers()) // Obtém membros publicos!
7   {
8       Console.WriteLine($"\t{prop.MemberType} {prop.Name} :
    {prop.DeclaringType}");
9   }
10
11  foreach (var field in type.GetRuntimeFields()) // Obtém campos privados!
12  {
13      Console.WriteLine($"\t{field.MemberType} {field.Name} :
    {field.DeclaringType}");
14  }
15
16  Console.WriteLine();
17  Funcionario funcionario = new Funcionario("Xispita", "12345678", 10);
18  var nameProp = type.GetProperty("Nome");
19  var name = nameProp.GetValue(funcionario) as string;
20  nameProp.SetValue(funcionario, name + "!");
21  Console.WriteLine(funcionario.Nome);
22
23  var calcMethod = type.GetMethod("CalcularSalario");
24  var wage = calcMethod.Invoke(funcionario, new object[] { 200 });
25  Console.WriteLine(wage);
26
27  Console.WriteLine();
28  var assembly = Assembly.GetExecutingAssembly();
29  foreach (var x in assembly.GetTypes())
30      Console.WriteLine(x.Name);
31
32  /**
33  Saída:
34  Class Funcionario:
35          Method get_Nome : Funcionario
36          Method set_Nome : Funcionario
37          Method get_EDV : Funcionario
38          Method set_EDV : Funcionario
39          Method get_SalarioHora : Funcionario
40          Method CalcularSalario : Funcionario
41          Method GetType : System.Object
42          Method ToString : System.Object
43          Method Equals : System.Object
44          Method GetHashCode : System.Object
45          Constructor .ctor : Funcionario
46          Property Nome : Funcionario
47          Property EDV : Funcionario
48          Property SalarioHora : Funcionario
49          Field salarioHora : Funcionario
50          Field <Nome>k__BackingField : Funcionario
51          Field <EDV>k__BackingField : Funcionario
```

```
52
53  Xispita!
54  2000
55
56  EmbeddedAttribute
57  NullableAttribute
58  NullableContextAttribute
59  Program
60  Funcionario
61   **/
62
63  public class Funcionario
64  {
65      private double salarioHora;
66
67      public string Nome { get; set; }
68      public string EDV { get; set; }
69      public double SalarioHora => salarioHora;
70
71      public Funcionario(string nome, string edv, double salarioHora)
72      {
73          this.Nome = nome;
74          this.EDV = edv;
75          this.salarioHora = salarioHora;
76      }
77
78      public double CalcularSalario(double horas)
79          => horas * salarioHora;
80  }
```

4 Reflexão Genérica

Assim como podemos fazer reflexão sobre um tipo qualquer, podemos fazer reflexão sobre um tipo que não conhecemos:

```
1  Creator<int> creator = new Creator<int>();
2  int i = creator.Create();
3  int j = creator.Create();
4
5  public class Creator<T>
6  {
7      public T Create(params object[] arr)
8      {
9          var type = typeof(T);
10         foreach (var constructor in type.GetConstructors())
11         {
12             var parameters = constructor.GetParameters();
13             if (parameters.Length == arr.Length)
14                 return (T)constructor.Invoke(arr);
15         }
16         return default(T);
17     }
18 }
```

5 Expressions

Expressões são outra forma do C# de compreender o próprio código. Com elas o C# é capaz de ler uma função Lambda e gerar uma árvore de expressão e reconhecer sua estrutura.

```
1  using System;
2  using System.Linq;
3  using System.Linq.Expressions;
4  using System.Collections.Generic;
5
6  Expression<Func<float, float>> exp = x => MathF.Sqrt(x * x + 3) +
   MathF.Sin(MathF.PI * x) - 2;
7  analyzeNode(exp.Body);
8
9  /**
10     ((Sqrt(((x * x) + 3)) + Sin((3,1415927 * x))) - 2)
11     ├(Sqrt(((x * x) + 3)) + Sin((3,1415927 * x)))
12     │├Sqrt(((x * x) + 3))
13     ││└((x * x) + 3)
14     ││ ├(x * x)
15     ││ │├x
16     ││ │└x
17     ││ └3
18     │└Sin((3,1415927 * x))
19     │  └(3,1415927 * x)
20     │   ├3,1415927
21     │   └x
22     └2
23  **/
24
25 void analyzeNode(Expression exp)
26 {
27     Stack<string> stack = new Stack<string>();
28     _analyzeNode(exp);
29
30     void _analyzeNode(Expression exp, string newStr = null)
31     {
32         if (newStr != null)
33             stack.Push(newStr);
34
35         var levelInfo = string.Concat(stack.Reverse());
36         if (levelInfo.Length > 0)
37             levelInfo =
38                 levelInfo.Substring(0, levelInfo.Length - 1) +
39                 (levelInfo.Last() == ' ' ? '└' : '├');
40
41         switch (exp)
42         {
43             case MethodCallExpression call:
44                 Console.Write(levelInfo);
45                 Console.WriteLine(call);
46
47                 var args = call.Arguments;
48                 for (int i = 0; i < args.Count; i++)
49                     _analyzeNode(args[i], i == args.Count - 1 ? " " : "|")
   ;
50                 break;
51
```

```
            case BinaryExpression bin:
                Console.Write(levelInfo);
                Console.WriteLine(bin);

                _analyzeNode(bin.Left, "|");
                _analyzeNode(bin.Right," ");
                break;

            case ConstantExpression con:
                Console.Write(levelInfo);
                Console.WriteLine(con);
                break;

            case ParameterExpression par:
                Console.Write(levelInfo);
                Console.WriteLine(par);
                break;

            default:
                Console.WriteLine(exp.GetType());
                break;
        }

        if (newStr != null)
            stack.Pop();
    }
}
```

Expressions podem ser usadas para muitas coisas, em especial, representar código em diversas plataformas, visto que, é possível traduzir código C# para outras linguagens mais facilmente. Podemos criar nossas funções em tempo de execução, compila-las e executa-las:

```
using System;
using System.Linq.Expressions;

var parameter = Expression.Parameter(typeof(float));                    // x
var sqrt = typeof(MathF).GetMethod("Sqrt");                             // Sqrt
var exp = Expression.Call(sqrt, parameter);                             // Sqrt(x)
var lambda = Expression.Lambda<Func<float, float>>(exp, parameter);     // x => Sqrt(x)
var f = lambda.Compile();
var result = f(4);
Console.WriteLine(result);
```

As possibilidades são infinitas e você pode escrever qualquer função usando isso:

```

using System;
using System.Linq.Expressions;
using static System.Linq.Expressions.Expression;

int[] data = new int[] { 5, 2, 1, 3, 4 };

// int low = int.MaxValue;
// int value = 0;
```

```
// for (int i = 0; i < arr.Length; i++)
// {
//      value = arr[i];
//      if (value < low)
//          low = value;
// }

var arr = Parameter(typeof(int[]), "arr");
var len = typeof(int[]).GetProperty("Length");
var print = typeof(Console).GetMethod("WriteLine", 0, new Type[] { typeof(int) });
var low = Parameter(typeof(int), "low");
var value = Parameter(typeof(int), "value");
var i = Parameter(typeof(int), "i");
var label = Label(typeof(int));
var block = Block(
    new[] { low, i, value },
    Assign(low, Constant(int.MaxValue)),
    Assign(i, Constant(0)),
    Loop(
        Block(
            Assign(value, ArrayAccess(arr, i)),
            IfThen(LessThan(value, low),
                Assign(low, value)
            ),
            Assign(i, Add(i, Constant(1))),
            IfThen(GreaterThanOrEqual(i, Property(arr, len)),
                Break(label, low)
            )
        ), label
    )
);

var lambda = Expression.Lambda<Func<int[], int>>(block, arr);
var f = lambda.Compile();
var result = f(data);
Console.WriteLine(result);
```

Ainda sim, você não estará apto a modificar métodos aos quais você não criou inicialmente como uma Expression.

6 Atributos

Atributos são modificadores que podem ser criados e lidos com reflection. Eles são úteis nas mais diversas situações e é muito comum que entremos em contato com eles antes mesmo deles serem compreensíveis a nós:

```
1  using System;
2  using System.Reflection;
3
4  foreach (var type in Assembly.GetExecutingAssembly().GetTypes())
5  {
6      var att = type.GetCustomAttribute<MyAttribute>();
7
8      if (att == null)
9          continue;
10
11     if (att.Data < 15)
12         continue;
13
14     Console.WriteLine(type.Name);
15 }
16 // Output: B
17
18
19 public class MyAttribute : Attribute
20 {
21     public int Data { get; set; }
22
23     public MyAttribute(int data)
24         => this.Data = data;
25 }
26
27 [MyAttribute(10)]
28 public class A { }
29
30 [My(20)] // A palavra Attribute pode ser omotida
31 public class B { }
32
33 public class C { }
```

7 Exemplo: Removendo Prints de Objetos específicos

```
1  using System;
2  using System.Collections.Generic;
3  using System.Linq.Expressions;
4  using System.Reflection;
5  using static System.Linq.Expressions.Expression;
6
7  A[] arrA = new[] { new A(), new A(), new A(), new A() };
8  B[] arrB = new[] { new B(), new B(), new B(), new B() };
9
10 arrA.Act(x => Console.WriteLine(x));
11 arrB.Act(x => Console.WriteLine(x));
12 arrB.Act(x => Console.WriteLine("Xispita"));
13
14 public static class ShowData
15 {
16     public static void Act<T>(this IEnumerable<T> coll,
   Expression<Action<T>> action)
17         where T : InfoElement
18     {
19         var type = typeof(T);
20         var att = type.GetCustomAttribute<NotPrintableAttribute>();
21         if (att == null)
22         {
23             act(coll, action.Compile());
24             return;
25         }
26
27         action = removePrint(action);
28         act(coll, action.Compile());
29     }
30
31     private static void act<T>(IEnumerable<T> coll, Action<T> action)
32         where T : InfoElement
33     {
34         var it = coll.GetEnumerator();
35         while (it.MoveNext())
36             action(it.Current);
37     }
38
39     private static Expression<Action<T>>
   removePrint<T>(Expression<Action<T>> action)
40         where T : InfoElement
41     {
42         switch (action.Body)
43         {
44             case MethodCallExpression call:
45                 bool isWrite = call.Method.Name == "WriteLine" ||
46                     call.Method.Name == "Write";
47                 bool isConsole = call.Method.DeclaringType == typeof(Conso
   le);
48                 bool isPrint = isConsole && isWrite;
49
50                 if (!isPrint)
```

```
51                      return action;
52
53                  var obj = call.Arguments.Count > 0 ? call.Arguments[0] :
    null;
54                  bool isNotPrintable =
    obj.Type.GetCustomAttribute<NotPrintableAttribute>() != null;
55
56                  if (!isNotPrintable)
57                      return action;
58
59                  var t = Parameter(typeof(T));
60                  return Lambda<Action<T>>(Empty(), t);
61
62              default:
63                  return action;
64          }
65      }
66  }
67
68  public class NotPrintableAttribute : Attribute { }
69
70  public class InfoElement
71  {
72      public override string ToString()
73          => "Info: " + getInfo();
74
75      protected virtual string getInfo()
76          => null;
77  }
78
79  public class A : InfoElement
80  {
81      protected override string getInfo()
82          => "Olá Mundo!";
83  }
84
85  [NotPrintable]
86  public class B : InfoElement
87  {
88      protected override string getInfo()
89          => "Hello World!";
90  }
```

8 Exercícios

1. Printe todas as classes de um projeto que possuem métodos com um atributo chamado 'ImportantAttribute'.
2. Crie um método de extensão chamado Copy. Este método estende um tipo T qualquer com construtor vazio e copia suas propriedades e campos privados para um novo objeto, copiando-o.
3. Construa um conversor de código C# para Python usando Expressions. Considere que não existem vetores e que todas as variáveis são do tipo int.
4. Faça uma função que conte quantas vezes uma variável de entrada é usada numa função lambda recebida.