Aula 14 - Programação Funcional e LINQ

Docupedia Export

Author:Balem Luis (CtP/ETS)
Date:23-May-2023 18:41

Table of Contents

- Introdução a Programação Funcional
- Delegados
- Métodos Anônimos
- Exercícios Propostos
- Delegados Genéricos
- Select e Where
- System.Linq
- Exercícios Propostos

1 Introdução a Programação Funcional

Programação funciona é um mundo a parte dentro da programação. Ela é um paradigma de programação assim como Orientação a Objetos. Porém bem mais difícil e se acostumar. Na programação funcional a função é tratada como um objeto. Ela pode ser retornada, pode ser mandada como parâmetro e coisas desse tipo. Estruturamos o nosso pensamento em chamada de funções alto-nível que trabalham com dados imutáveis.
Vamos pegar leve olhando uma das programações funcionais mais simples e menos apegadas a uma programação funcional 'Hardcore' que é o JavaScirpt. Para você ter noção, enquanto JS tem loops e ifs, linguagens como Haskell não possuem loops nem estruturas condicionais. Implementar um merge-sort em F# usa recursão no lugar de loops:

```
let merge x y =
    let rec rMerge r x y =
        match x, y with
        | x, [] -> r @ x
        | [], y -> r @ y
        | x::xs, y::ys ->
            if x < y then rMerge (r @ [x]) xs (y::ys)
            else rMerge (r @ [y]) (x::xs) ys
    rMerge [] x y

let mergeSort x =
    let rec rMergeSort (x:'a list) =
        if x.Length < 2 then x
        else
            let split x =
                let rec rSplit (x:'a list) (y:'a list) =
                    if x.Length > y.Length
                    then rSplit x.Tail (x.Head::y)
                    else (x, y)
                rSplit x []
            let (a, b) = split x
            let sa = rMergeSort a
            let sb = rMergeSort b
            merge sa sb
    rMergeSort x

let x = [ 1; 3; 2; 5; 4; 6 ]
let r = mergeSort x
printfn "%A" r
```

A expressividade é poderosa mas complexa de se compreender. Vamos observar um pouco o JS que não é tão fortemente funcional:

```
1  // Declarar uma função
2  function f()
3  {
4      console.log("Olá mundo")
5  }
6
7  // Chama uma função 3 vezes
8  // A função é passada como parâmetro
9  function chamar3Vezes(func)
10 {
11     func()
12     func()
13     func()
14 }
15
16 // Printa 'Olá mundo' 3 vezes
17 chamar3Vezes(f)
```

Como você pode ver, a programação funcional permite que nós possamos usar as funções como dados, passando funções como parâmetro para outras funções. Não só isso, é possível retornar funções também:

```
1  function f()
2  {
3      console.log("Olá mundo")
4  }
5
6  function montarFuncao(texto)
7  {
8      return function func()
9      {
10         console.log(texto)
11     }
12 }
13
14 f = montarFuncao("Olá mundo")
15 f()
```

A função montar função retorna uma função que pode ser usada em outros lugares. A variável 'texto' é capturada pela função func que é atribuída a variável f. Assim f é uma função e pode ser posteriormente chamada. Podemos fazer mesclagens complexas de código:

```
function montarFuncao(texto)
{
    return function func()
    {
        console.log(texto)
    }
}

function chamar3Vezes(func)
{
    func()
    func()
    func()
}

f = montarFuncao("Olá mundo")
chamar3Vezes(f)
```

2 Delegados

Para representar esse comportamento, o C# nos trás os delegados. Estruturas que são declaradas no mesmo nível que classes, structs, enums, interfaces e namespaces:

```
1  using System;
2
3  MeuDelegate f = func;
4  f();
5
6  void func()
7  {
8      Console.WriteLine("Olá mundo");
9  }
10
11 public delegate void MeuDelegate();
```

O delegado define um novo tipo que podem armazenar qualquer função sem retorno (void) e sem parâmetros. Se tentarmos fazer essa operação com outro tipo de função teremos um erro. Abaixo alguns exemplos possíveis de utilização dos delegados:

```
1  using System;
2
3  MeuPrint print = Console.WriteLine;
4  print("Olá mundo");
5
6  public delegate void MeuPrint(string s);
7  using System;
8
9  // Delegados são apontadores de funções. Eles apontam para uma, nenhuma ou muitas funções.
10 // Abaixo um delegado começa não apontando para nada, após isso apontamos para count.
11 // Ao chamar f nós temos 'Count chamado' na tela.
12 MeuDelegate f = null;
13 f += count;
14 f("Olá mundo");
15
16 // Agora dicionamos parse ao ponteiro f. Ao chamar a função com a entrada "40" temos que tanto
17 // count quanto parse são chamados, na ordem que foram adicionados, ou seja, temos 'Count chamado'
18 // seguido de 'Parse chamado' e por fim a última saída, no caso do parse, é retornado e apresentado
19 // com o valor 40
```

```
f += parse;
int i = f("40");
Console.WriteLine(i);

int count(string s)
{
    Console.WriteLine("Count chamado");
    return s.Length;
}

int parse(string s)
{
    Console.WriteLine("Parse chamado");
    return int.Parse(s);
}

public delegate int MeuDelegate(string s);
using static System.Console;

executeNVezes(WriteLine, 10, "Xispita");

void executeNVezes(MeuDelegate func, int N, string param)
{
    for (int i = 0; i < N; i++)
        func(param);
}

public delegate void MeuDelegate(string s);
```

O próximo exemplo é muito interessante e mostra como usar delegados para representar funções matemáticas reais.

```
using static System.Console;

var f = linear(10, -5);
WriteLine(f(1));

FuncaoMatematica constante(float c)
{
    float funcaoConstante(float x)
```

```
9       {
10          return c;
11      }
12      return funcaoConstante;
13  }
14
15  FuncaoMatematica fx()
16  {
17      float funcaoX(float x)
18      {
19          return x;
20      }
21      return funcaoX;
22  }
23
24  FuncaoMatematica soma(FuncaoMatematica f, FuncaoMatematica g)
25  {
26      float funcaoSoma(float x)
27      {
28          return f(x) + g(x);
29      }
30      return funcaoSoma;
31  }
32
33  FuncaoMatematica produto(FuncaoMatematica f, FuncaoMatematica g)
34  {
35      void funcaoProduto(float x)
36      {
37          return f(x) * g(x);
38      }
39      return funcaoProduto;
40  }
41
42  // a * x + b
43  FuncaoMatematica linear(float a, float b)
44  {
45      var ax = produto(constante(a), fx());
46      return soma(ax, constante(b));
47  }
```

```
48
49  public delegate float FuncaoMatematica(float x);
```

3 Métodos Anônimos

Podemos também declarar funções sem precisar declarar uma função com nome e assinatura. O nome disso são funções anônimas e existem algumas formas de fazer, por exemplo, usando a palavra reservada delegate. Mas a mais utilizada é usando a 'arrow function'. A setinha que usamos para tornar umas métodos de uma linha também pode ser usada para isso. Veja alguns exemplos:

```
1  using System;
2
3  MeuPrint print1 = delegate(string s) // OK
4  {
5      Console.WriteLine(s);
6  };
7
8  MeuPrint print2 = (string s) => // OK
9  {
10     Console.WriteLine(s);
11 };
12
13 MeuPrint print3 = s => // OK, usando inferência
14 {
15     Console.WriteLine(s);
16 };
17
18 MeuPrint print4 = s => Console.WriteLine(s); // OK, de uma única linha
19
20 public delegate void MeuPrint(string s);
```

Veja o exemplo das funções matemáticas refatorado com funções anônimas:

```
1  // Poderiamos usar o código abaixo para função constante
2  // FuncaoMatematica constante(float c)
3  // {
4  //      return x => c;
5  // }
6  // Mas como ela pode ser escrita em uma linha usamos 2 setas:
7  // A primeira para dizer qual a implementação da função constante em uma única linha
8  // A segunda para definir a função f(x) = c (função que independente do número recebido retorna uma constante)
9  FuncaoMatematica constante(float c)
10     => x => c;
```

```
11
12  FuncaoMatematica fx()
13      => x => x;
14
15  FuncaoMatematica soma(FuncaoMatematica f, FuncaoMatematica g)
16      => x => f(x) + g(x);
17
18  FuncaoMatematica produto(FuncaoMatematica f, FuncaoMatematica g)
19      => x => f(x) * g(x);
20
21  FuncaoMatematica linear(float a, float b)
22      => x => a * x + b;
23
24  public delegate float FuncaoMatematica(float x);
```

4 Exercícios Propostos

1. Faça uma função que recebe um double x e retorna uma função que recebe um valor e retorna valor^x. Use System.Math.Pow para isso.
2. Faça uma função que receba um função F, número N e um número T. F recebe um int e retorna um int. Chame a função F repetidas vezes onde o parâmetro de F é o resultado da chamada anterior de F até que o resultado de F seja T. Inicie com parâmetro N. Dica: Teste a conjectura de collatz: Se um número for par, divida por 2, se for impar multiplique por 3 e some 1. Repetindo esse processo a conjectura aponta que qualquer N deve chegar a valer T = 1 em algum momento.
3. Faça uma função que receba duas funções, uma que leva uma int em um string e outra que leva um string em um int e faça a composição das duas funções retornando a função h(x) = f(g(x)).

5 Delegados Genéricos

Não comentamos ainda, mas sim, é possível fazer delegados genéricos:

```
Func<string, int> converter = int.Parse;
Func<int, int, int> somador = soma;

int soma(int i, int j) => i + j;

public delegate R Func<T, R>(T entrada);
public delegate R Func<T1, T2, R>(T1 entrada, T2 entrada2);

No namespace System temos delegados genéricos que você pode usar a vontade. Trata-se do Func que recebe vários parâmetros genéricos (até mais de 10) e retorna o último parâmetro (como o exemplo acima). E Action, que não retorna nada (void) e recebe vários parâmetros genéricos para determinar seus parâmetros de função. Existe ainda o Predicate que sempre retorna bool e em muitas siutações pode se substituído por Func.

using System;

Action<string> f = Console.WriteLine;
Func<double, double, double> g = Math.Pow;
Func<int, int, bool> h = (m, n) => m > n;

if (h(100, 10))
{
    var value = g(5, 2).ToString();
    f(value);
}
```

6 Select e Where

Agora que temos os poderosos recursos da Programação Funcional vamos entender como ele impacta o C# em sua principal função dentro da linguagem. Dentro das funções LINQ:

```
1  using System;
2  using System.Collections.Generic;
3
4  List<int> list = new List<int>();
5  list.Add(1);
6  list.Add(2);
7  list.Add(3);
8
9  // Retorna o Primeiro valor par, caso contrário retorna o valor default (0)
10 int value = list.FirstOrDefault(x => x % 2 == 0); // 2
11
12 public static class Enumerable
13 {
14     public static T FirstOrDefault<T>(IEnumerable<T> coll, Func<T, bool> func)
15     {
16         // Busca todos os objetos da coleção
17         foreach (var obj in coll)
18         {
19             // Se a condição enviada for verdadeira...
20             bool condition = func(obj);
21
22             // Retorna o Obejto
23             if (condition)
24                 return obj;
25         }
26         return default(T);
27     }
28 }
```

Este é o LINQ o mais próximo possível do seu poder máximo. Usando apenas funções anônimas podemos operar sobre dados de forma dinâmica e poderosa. Abaixo uma função de filtro chamada Where:

```
1  using System;
2  using System.Collections.Generic;
```

```
3
4   List<int> list = new List<int>();
5   list.Add(1);
6   list.Add(2);
7   list.Add(3);
8
9   // Retorna todos os valores impares da coleção
10  var values = list.Where(x => x % 2 == 1);
11
12  public static class Enumerable
13  {
14      public static IEnumerable<T> Where<T>(IEnumerable<T> coll, Func<T, bool> func)
15      {
16          // Busca todos os objetos da coleção
17          foreach (var obj in coll)
18          {
19              // Se a condição enviada for verdadeira...
20              bool condition = func(obj);
21
22              // Retorna o Obejto
23              if (condition)
24                  yield return obj;
25          }
26      }
27  }
```

O Where é usado para filtrar os dados, ou seja, ler apenas os dados que atendem uma determinada condição.
Mesmo não podendo alterar a lista original podemos criar novos dados a partir de uma lista anterior de forma imutável. Para isso utilizamos o Select.

```
1   using System;
2   using System.Collections.Generic;
3
4   List<int> list = new List<int>();
5   list.Add(1);
6   list.Add(2);
7   list.Add(3);
8
9   // De uma coleção de inteiros [1, 2, 3] retornamos uma lista de string dos quadrados
```

```
// ou seja, ["1", "4", "9"].
var values = list.Select(x => (x * x).ToString());

public static class Enumerable
{
    public static IEnumerable<R> Select<T, R>(IEnumerable<T> coll, Func<T, R> func)
    {
        // Transforma os objetos do tipo T no tipo R
        foreach (var obj in coll)
            yield return func(obj);
    }
}
```

O select por sua vez chama a a função anônima várias vezes e altera os objetos um por um. É um código incrível. Com poucas palavras em questão de sintaxe expressamos funções extremamente poderosas.

7 System.Linq

Evidentemente, tudo isso que você está utilizando já existe e está pronto. Basta importar System.Linq e você terá uma sequência infindável de funções e sobrecargas para utilizar. Você pode consultar a diversidade de implementações na páginas: https://learn.microsoft.com/pt-br/dotnet/api/system.linq.enumerable?view=net-7.0.

8 Exercícios Propostos

1. Considere o seguinte código de base para o exercício:

```
1  using System;
2  using System.Collections.Generic;
3
4  IEnumerable<int> get()
5  {
6      for (int i = 0; i < 1000; i++)
7          yield return i + 1;
8  }
```

Filtre os dados da função get para obter apenas os números múltiplos de 4.

1. Ainda considerando o exemplo do exercícios anterior, aplique a transformação da função collatz 3 vezes, ou seja, se o número for par, divida por 2, se for impar, multiplique por três e some 1. Depois disso, Conte (usando o Count da aula passada) todos os valores maiores que 1000. Repita o processo, só que dessa vez conte quantos valores são menores que 5.
2. Faça uma classe Pessoa com nome e idade e crie uma lista com várias pessoas. Apresente o nome de todos os maiores de idade.
3. Implemente o SkipWhile e TakeWhile. Funções como Skip e Take, mas que recebem uma função como o Where e pulam/pegam enquanto a condição for verdadeira.
4. Implemente a função Zip que pega duas coleções e opera elementos par-a-par em uma função dada.

```
1  using System;
2  using System.Collections.Generic;
3
4  int[] arrA = new int[] { 1, 3, 5, 7 };
5  int[] arrB = new int[] { 2, 3, 5, 9};
6
7  foreach (var k in arrA.Zip(arrB, (i, j) => j - i))
8      Console.Write(k);
9  // Result: 1002
10
11 public static class Enumerable
12 {
13     public static IEnumerable<T> TakeWhile<T>(IEnumerable<T> coll, Func<T, bool> func)
14     {
15         throw new NotImplementedException();
16     }
17
```

```
18      public static IEnumerable<T> SkipWhile<T>(IEnumerable<T> coll, Func<T, bool> func)
19      {
20          throw new NotImplementedException();
21      }
22
23      public static IEnumerable<R> Zip<T, U, R>(IEnumerable<T> coll, IEnumerable<U> second, Func<T, U, R> func)
24      {
25          throw new NotImplementedException();
26      }
27  }
```

Give feedback