Aula 13 - Coleções e Introdução a Language Integrated Query (LINQ)

Docupedia Export

Author:Balem Luis (CtP/ETS)
Date:18-May-2023 18:19

Table of Contents

1 Coleções, IEnumerable e IEnumerator 4
2 Métodos Iteradores 9
3 Introdução ao LINQ 13
4 Métodos de Extensão 17
5 Inferência 20
6 Observações Importantes 22
7 Exercícios Propostos 23

- Coleções, IEnumerable e IEnumerator
- Métodos Iteradores
- Introdução ao LINQ
- Métodos de Extensão
- Inferência
- Observações Importantes
- Exercícios Propostos

1 Coleções, IEnumerable e IEnumerator

Como você pode ter percebido, existem muitas coleções no C#. Vetores, Listas encadeadas, arrays dinâmicos, pilhas, filas, entre outras possibilidades. Existe uma certa dificuldade de padronizar a forma como acessamos coleções. Pensando nisso, o C# trouxe um padrão de projeto chamado iterador. Este é antigo e nasceu para aumentar a velocidade com que nós varriamos listas encadeadas. Se você lembra bem, na nosso exemplo de lista encadeada nós sempre pegavamos o primeiro elemento e olhávamos o próximo até encontrar o elemento que queríamos. Fazer esse processo toda vez é lento se queremos ler todos os elementos da lista. Pensando nisso a Microsoft criou a seguinte interface:

```
1  public interface IEnumerator<T>
2  {
3      T Current { get; }
4      bool MoveNext();
5      void Reset();
6  }
```

Vamos supor uma coleção aleatória. O V que você vê é a posição do iterador. Ele é isso, uma seta que aponta par algum lugar na coleção. Entende-se por coleção, qualquer conjunto de objetos, com ou sem ordem, com ou sem repetição. Ou seja, nem mesmo é um conjunto. Mesmo assim, podemos colocar os elementos enfileirados de qualquer forma que conseguirmos. O iterador começa em uma posição fora do vetor.

V										
-	7	9	3	4	8	0	4	6	2	1

Se verificarmos o valor de Current teremos então um erro. Ao usar a função MoveNext o iterador avança e a função retorna 'true', pois o avanço foi bem sucessido.

	V									
-	7	9	3	4	8	0	4	6	2	1

Agora Current tem o valor de 7. Chamando MoveNext 3 vezes teríamos então:

				V						
-	7	9	3	4	8	0	4	6	2	1

Se por acasos chamarmos Reset:

v										
-	7	9	3	4	8	0	4	6	2	1

Chamando MoveNext 1000 vezes o iterador fica na última posição possível, caso seja impossível avançar:

										v
-	7	9	3	4	8	0	4	6	2	1

Um iterador padrozina a forma que se lê uma coleção. Note que ele não permite que você altere os valores do iterador. Nem mesmo volte uma única casa. Apenas avançar, resetar e ler.

No C# toda coleção retorna um iterador, pois toda coleção implementa IEnumerable:

```
1  public interface IEnumerable<T>
2  {
3      IEnumerator<T> GetEnumerator();
4  }
```

Toda e qualquer coleção, vetor, pilhas, dicionário, lista de todas as formas, filas, hashs, tudo, implementa a interface IEnumerable e promete te retornar um iterador para que você possa ler ela.

```
1  using System;
2  using System.Collections;
3  using System.Collections.Generic;
4
5  List<int> lista = new List<int>();
6
7  lista.Add(1);
8  lista.Add(2);
9  lista.Add(3);
10
11 var it = lista.GetEnumerator();
12
13 while (it.MoveNext())
14     Console.WriteLine(it.Current);
15
16 public class ListIterator<T> : IEnumerator<T>
17 {
```

```
18      private int index = -1;
19      public List<T> List { get; set; }
20
21      public void Reset()
22      {
23          index = -1;
24      }
25
26      public bool MoveNext()
27      {
28          if (index >= List.Count)
29              return false;
30          index++;
31          return true;
32      }
33
34      public T Current
35      {
36          get
37          {
38              return List[index];
39          }
40      }
41
42      // IEnumerator genérico implementa o IEnumerator de object, por isso percisamos imlementar o Current que retorna um
    object
43      // Mas podemos chamar o nosso Current genérico como retorno
44      object IEnumerator.Current => Current;
45
46      // Libera recursos usados pelo iterador. Aqui não somos obrigados a fazer nada grandioso a não se que estejamos
    alocando recursos não gerenciados
47      public void Dispose() { }
48  }
49
50  public class List<T> : IEnumerable<T>
51  {
52      private int pos = 0;
53      private T[] vetor = new T[10];
54      public int Count => pos;
```

```
55
56      public T this[int index]
57      {
58          get => this.vetor[index];
59          set => this.vetor[index] = value;
60      }
61
62      public void Add(T value)
63      {
64          int len = vetor.Length;
65          if (pos == len)
66          {
67              T[] newVetor = new T[2 * len];
68              for (int i = 0; i < pos; i++)
69                  newVetor[i] = vetor[i];
70              vetor = newVetor;
71          }
72
73          vetor[pos] = value;
74          pos++;
75      }
76
77      // Simplismente retornamos o iterador que fizemos
78      public IEnumerator<T> GetEnumerator()
79      {
80          ListIterator<T> it = new ListIterator<T>();
81          it.List = this;
82          return it;
83      }
84
85      // Mesma condição que na implementação do iterador
86      IEnumerator IEnumerable.GetEnumerator()
87          => GetEnumerator();
88  }
```

Outra vantagem de implementar um IEnumerable é que ele é a base do **foreach**. O foreach é um for automático que chama o iterador e tem um funcionamento idêntico ao while com o iterador que fizemos acima. Certamente, não é possível alterar a lista original, assim como não é possível alterar os dados no iterador, enquanto você percorre um foreach:

```
1  List<int> lista = new List<int>();
2
3  lista.Add(1);
4  lista.Add(2);
5  lista.Add(3);
6
7  foreach(var n in lista)
8  {
9      Console.WriteLine(n);
10 }
11
12 // ...
```

2 Métodos Iteradores

Embora útil, a implementação pode ser um pouco tediosa. Pensando nisso o C# possui métodos iteradores. Qualquer função que deseja retornar um IEnumerable ou IEnumerator pode usa da palavra reservada yield para retornar os valores um por um. O código abaixo é equivalente ao código feito acima com a implementação completa do iterador:

```
1  using System;
2  using System.Collections;
3  using System.Collections.Generic;
4
5  List<int> lista = new List<int>();
6
7  lista.Add(1);
8  lista.Add(2);
9  lista.Add(3);
10
11 foreach(var n in lista)
12     Console.WriteLine(n);
13
14 public class List<T> : IEnumerable<T>
15 {
16     private int pos = 0;
17     private T[] vetor = new T[10];
18     public int Count => pos;
19
20     public T this[int index]
21     {
22         get => this.vetor[index];
23         set => this.vetor[index] = value;
24     }
25
26     public void Add(T value)
27     {
28         int len = vetor.Length;
29         if (pos == len)
30         {
31             T[] newVetor = new T[2 * len];
32             for (int i = 0; i < pos; i++)
33                 newVetor[i] = vetor[i];
```

```
34              vetor = newVetor;
35          }
36
37          vetor[pos] = value;
38          pos++;
39      }
40
41      public IEnumerator<T> GetEnumerator()
42      {
43          for (int i = 0; i < pos; i++)
44          {
45              yield return vetor[i];
46          }
47      }
48
49      IEnumerator IEnumerable.GetEnumerator()
50          => GetEnumerator();
51  }
```

O fluxo é muito interessante: Toda vez que o iterador é chamado, por um foreach por exemplo, o código do GetEnumerator roda junto de chamadas do MoveNext mas trava na linha yield return. Ou seja, o código não executa infinitamente mas literalmente congela na linha do yield return e só descongela quando chamamos o MoveNext novamente. Ao chamar o Current, obtemos o valor do último yield return visto. Observe o exemplo abaixo para entender este complexo fluxo de informação:

```
1   using System;
2   using System.Collections.Generic;
3
4   Console.WriteLine("Vou chamar a função get");
5   var it = get();
6   Console.WriteLine("Chamei a função get");
7
8   Console.WriteLine("Vou chamar a função MoveNext");
9   it.MoveNext();
10  Console.WriteLine("Congelei");
11  Console.WriteLine(it.Current);
12
13  it.MoveNext();
14  Console.WriteLine("Congelei");
15  Console.WriteLine(it.Current);
16
```

```
17  it.MoveNext();
18  Console.WriteLine("Congelei");
19  Console.WriteLine(it.Current);
20
21  it.MoveNext();
22  Console.WriteLine("Congelei");
23  Console.WriteLine(it.Current);
24
25  it.MoveNext();
26  Console.WriteLine("Congelei");
27  Console.WriteLine(it.Current);
28
29  IEnumerator<int> get()
30  {
31      Console.WriteLine("Entrei no get");
32      yield return 1;
33      Console.WriteLine("Descongelei a primeira vez");
34      yield return 2;
35      Console.WriteLine("Descongelei a segunda vez");
36      yield return 3;
37      Console.WriteLine("Descongelei a última vez");
38  }
```

Como saída desse programa temos:

- Vou chamar a funçao get
- Chamei a funçao get
- Vou chamar a funçao MoveNext
- Entrei no get
- Congelei
- 1
- Descongelei a primeira vez
- Congelei
- 2
- Descongelei a segunda vez
- Congelei
- 3
- Descongelei a última vez
- Congelei

- 3
- Congelei
- 3

3 Introdução ao LINQ

E se você usasse o iterador para fazer funções universais de processamento de coleções? E se usassemos métodos iteradores para fazer isso por demanda, apenas fazendo cálculos quando os valores são solicitados? Essa é a ideia genial por trás de uma das mais brilhantes features do C#, a Consulta Integrada à Linguagem, Language Integrated Query ou, simplismente, LINQ. A ideia é criar uma função estática que receba uma coleção qualquer (que herde de IEnumerable) e use o iterador para processá-la. Observe:

```
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  Stack<int> stack = new Stack<int>();
10 stack.Push(1);
11 stack.Push(2);
12 stack.Push(3);
13
14 int[] array = new int[] { 1, 2, 3 };
15
16 Console.WriteLine(Enumerable.Count(list)); // 3
17 Console.WriteLine(Enumerable.Count(stack)); // 3
18 Console.WriteLine(Enumerable.Count(array)); // 3
19
20 public static class Enumerable
21 {
22     public static int Count(IEnumerable<int> coll)
23     {
24         int count = 0;
25
26         var it = coll.GetEnumerator();
27         while (it.MoveNext())
28             count++;
29
30         return count;
31     }
```

```
32  }
```

A função Count conta quantos elementos existem em uma coleção, independete de qual seja. E é claro, melhoria 1: Count pode ser uma função genérica:

```
1   using System;
2   using System.Collections;
3   using System.Collections.Generic;
4
5   List<int> list = new List<int>();
6   list.Add(1);
7   list.Add(2);
8   list.Add(3);
9
10  Stack<string> stack = new Stack<string>();
11  stack.Push("1");
12  stack.Push("2");
13  stack.Push("3");
14
15  IEnumerable[] array = new IEnumerable[] { list, stack };
16
17  Console.WriteLine(Enumerable.Count<int>(list)); // 3
18  Console.WriteLine(Enumerable.Count<string>(stack)); // 3
19  Console.WriteLine(Enumerable.Count<IEnumerable>(array)); // 2
20
21  public static class Enumerable
22  {
23      public static int Count<T>(IEnumerable<T> coll)
24      {
25          int count = 0;
26
27          var it = coll.GetEnumerator();
28          while (it.MoveNext())
29              count++;
30
31          return count;
32      }
33  }
```

Atenção especial ao vetor de coleções que tem list e stack nele, perceba que funciona perfeitamente.

Outro exemplo: A função Take. Recebe a coleção e um número N, pega apenas os primeiros N valores:

```
1   public static class Enumerable
2   {
3       public static IEnumerable<T> Take<T>(IEnumerable<T> coll, int N)
4       {
5           List<T> list = new List<T>();
6
7           var it = coll.GetEnumerator();
8           for (int i = 0; i < N && it.MoveNext(); i++)
9               list.Add(it.Current);
10
11          return list;
12      }
13  }
```

Note dois fatos peculiáremos: Primeiro, a função retorna um IEnumerable, ou seja, podemos usar yield return na função Take, essa é a nossa melhoria 2:

```
1   public static class Enumerable
2   {
3       public static IEnumerable<T> Take<T>(IEnumerable<T> coll, int N)
4       {
5           var it = coll.GetEnumerator();
6           for (int i = 0; i < N && it.MoveNext(); i++)
7               yield return it.Current;
8       }
9   }
```

Segundo, como temos uma coleção de resultado podemos usar várias funções LINQ uma depois da outra:

```
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   Stack<string> stack = new Stack<string>();
```

```
10  stack.Push("1");
11  stack.Push("2");
12  stack.Push("3");
13
14  Console.WriteLine(Enumerable.Count<int>(Enumerable.Take<int>(list, 2))); // 2
15  Console.WriteLine(Enumerable.Count<string>(Enumerable.Take<string>(stack, 10))); // 3
16
17  public static class Enumerable
18  {
19      public static IEnumerable<T> Take<T>(IEnumerable<T> coll, int N)
20      {
21          var it = coll.GetEnumerator();
22          for (int i = 0; i < N && it.MoveNext(); i++)
23              yield return it.Current;
24      }
25
26      public static int Count<T>(IEnumerable<T> coll)
27      {
28          int count = 0;
29
30          var it = coll.GetEnumerator();
31          while (it.MoveNext())
32              count++;
33
34          return count;
35      }
36  }
```

4 Métodos de Extensão

Apesar de ser uma feramente legal, está longe do ideal. Mas para entender como tudo melhora, precisamos compreender os métodos de extensão. Primeiramente observe o seguinte exemplo:

```
1  using System;
2
3  MyExtensionMethods.Print("Oi");
4
5  public static class MyExtensionMethods
6  {
7      public static void Print(string text)
8      {
9          Console.WriteLine(text);
10     }
11 }
```

Fizemos um print mais extenso do que o necessário. Não seria legal se o Print já existisse dentro da classe String? Infelizmente não podemos abrir a classe String e adicionar nós mesmos, não é? E usar um 'Console.WriteLine(this)' (print você mesmo) lá de dentro. Isso não seria porquê veríamos coisas privadas dentro da classe que não gostaríamos de ver. Mas existe uma solução: Métodos de Extensão. Observe:

```
1  using System;
2
3  // Quando isso compila...
4  "Oi".Print();
5  // Vira isso...
6  // MyExtensionMethods.Print("Oi");
7
8  // objeto estranho.PrintObj() também funciona
9  new AppDomainUnloadedException().PrintObj();
10
11 public static class MyExtensionMethods
12 {
13     // Basta adicionar 'this' antes do primeiro parâmetro no método estático e adicionamos o método Print dentor da string
14     public static void Print(this string text)
15     {
16         Console.WriteLine(text);
17     }
18
```

```
    // Ou de QUALQUER objeto
    public static void PrintObj(this object obj)
    {
        Console.WriteLine(obj);
    }
}
```

Assim nossos métodos LINQ ganham a terceira melhoria, invertendo a ordem de chamada, deixando na ordem que nós usamos de fato:

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

List<int> list = new List<int>();
list.Add(1);
list.Add(2);
list.Add(3);

Stack<string> stack = new Stack<string>();
stack.Push("1");
stack.Push("2");
stack.Push("3");

Console.WriteLine(list.Take<int>(2).Count<int>()); // 2
Console.WriteLine(stack.Take<string>(10).Count<string>()); // 3

public static class Enumerable
{
    public static IEnumerable<T> Take<T>(this IEnumerable<T> coll, int N)
    {
        var it = coll.GetEnumerator();
        for (int i = 0; i < N && it.MoveNext(); i++)
            yield return it.Current;
    }

    public static int Count<T>(this IEnumerable<T> coll)
    {
        int count = 0;

        var it = coll.GetEnumerator();
```

```
31          while (it.MoveNext())
32              count++;
33
34          return count;
35      }
36  }
```

5 Inferência

Para nossa quarta e última melhoria nesta aula nós temos a inferência: A capacidade do C# de perceber o tipo que está sendo passado pelo contexto. Assim alguns parâmetros genéricos torna-se desnecessários:

```
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  Stack<string> stack = new Stack<string>();
10 stack.Push("1");
11 stack.Push("2");
12 stack.Push("3");
13
14 Console.WriteLine(list.Take(2).Count()); // 2
15 Console.WriteLine(stack.Take(10).Count()); // 3
16
17 public static class Enumerable
18 {
19     public static IEnumerable<T> Take<T>(this IEnumerable<T> coll, int N)
20     {
21         var it = coll.GetEnumerator();
22         for (int i = 0; i < N && it.MoveNext(); i++)
23             yield return it.Current;
24     }
25
26     public static int Count<T>(this IEnumerable<T> coll)
27     {
28         int count = 0;
29
30         var it = coll.GetEnumerator();
31         while (it.MoveNext())
32             count++;
33
34         return count;
```

```
35          }
36      }
```

6 Observações Importantes

É importante comentar que todas as coleções geradas por métodos LINQ são imutáveis. Isso significa que ao executar a função Take, você não modifica a coleção original, mas gera uma nova coleção. Você não está de fato mudando a coleção inicial. Tenha sempre isso em mente. A cada função LINQ você gera uma nova coleção 'virtual' ou não. Evidentemente, se os objetos da coleção forem por referência, alterá-los altera seus valores nas coleções originais.
Outro fator importante é que o método é executado por demanda (a cada loop de um foreach ou quando você usa MoveNext em um iterador), ou seja, ao executar a função Take, praticamente nenhum trabalho é realizado. Só será realizado trabalho na execução de outras funções não iteradoras, como Count por exemplo, ou ao consumir os dados. Por isso, se você tem uma carga alta de trabalho, é bom lembrar que se você não requisitar os dados o trabalho não é feito.

7 Exercícios Propostos

Implemente as seguintes funções LINQ:

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

List<int> list = new List<int>();
list.Add(1);
list.Add(2);
list.Add(3);

Stack<string> stack = new Stack<string>();
stack.Push("1");
stack.Push("2");
stack.Push("3");

Console.WriteLine(list.Take(2).Count()); // 2
Console.WriteLine(stack.Take(10).Count()); // 3

public static class Enumerable
{
    public static IEnumerable<T> Take<T>(this IEnumerable<T> coll, int N)
    {
        var it = coll.GetEnumerator();
        for (int i = 0; i < N && it.MoveNext(); i++)
            yield return it.Current;
    }

    public static int Count<T>(this IEnumerable<T> coll)
    {
        int count = 0;

        var it = coll.GetEnumerator();
        while (it.MoveNext())
            count++;

        return count;
    }
```

```
36
37      // Pula os primeiros N valores e retorna o resto da coleção
38      public static IEnumerable<T> Skip<T>(this IEnumerable<T> coll, int N)
39      {
40          throw new NotImplementedException();
41      }
42
43      // Retorna a coleção com um elemento a mais no final dela
44      public static IEnumerable<T> Append<T>(this IEnumerable<T> coll, T value)
45      {
46          throw new NotImplementedException();
47      }
48
49      // Retorna a coleção com um elemento a mais no inicio dela
50      public static IEnumerable<T> Prepend<T>(this IEnumerable<T> coll, T value)
51      {
52          throw new NotImplementedException();
53      }
54
55      // Cria um array e preenche os elementos da coleção, convertendo a coleção para um array
56      public static T[] ToArray<T>(this IEnumerable<T> coll)
57      {
58          throw new NotImplementedException();
59      }
60
61      // Cria uma lista e preenche os elementos da coleção, convertendo a coleção para uma lista
62      public static List<T> ToList<T>(this IEnumerable<T> coll)
63      {
64          throw new NotImplementedException();
65      }
66
67      // Divide a coleção inicial em vários vetores de tamanho size
68      public static IEnumerable<T[]> Chunk<T>(this IEnumerable<T> coll, int size)
69      {
70          throw new NotImplementedException();
71      }
72
73      // Concatena duas coleções retornando uma coleção maior
74      public static IEnumerable<T> Concat<T>(this IEnumerable<T> coll, IEnumerable<T> second)
```

```
75      {
76          throw new NotImplementedException();
77      }
78
79      // Retorna o primeiro elemento da coleção, caso a coleção esteja vazia estoure um erro
80      public static T First<T>(this IEnumerable<T> coll)
81      {
82          throw new NotImplementedException();
83      }
84
85      // Retorna o primeiro elemento da coleção, caso a coleção esteja vazia retorne o valor padrão default(T).
86      public static T FirstOrDefault<T>(this IEnumerable<T> coll)
87      {
88          throw new NotImplementedException();
89      }
90
91      // Retorna o último elemento da coleção, caso a coleção esteja vazia estoure um erro
92      public static T Last<T>(this IEnumerable<T> coll)
93      {
94          throw new NotImplementedException();
95      }
96
97      // Retorna o último elemento da coleção, caso a coleção esteja vazia retorne o valor padrão default(T).
98      public static T LastOrDefault<T>(this IEnumerable<T> coll)
99      {
100         throw new NotImplementedException();
101     }
102
103     // Retorna o único elemento da coleção, caso ela estja vazia ou tenha mais de um elemento estoure um erro
104     public static T Single<T>(this IEnumerable<T> coll)
105     {
106         throw new NotImplementedException();
107     }
108
109     // Retorna o único elemento da coleção, caso ela tenha mais de um elemento estoure um erro, esteja vazia retorne o
    valor padrão default(T)
110      public static T SingleOrDefault<T>(this IEnumerable<T> coll)
111     {
112         throw new NotImplementedException();
```

```
113         }
114
115         // Retorna uma coleção com a ordem dos elementos invertida
116         public static IEnumerable<T> Reverse<T>(this IEnumerable<T> coll)
117         {
118             throw new NotImplementedException();
119         }
120
121         // Dada duas coleções, retorna uma tupla juntando cada elemento par a par.
122         // Exemplo: 1,2,3 com 'a','b','c' retornaria (1,'a'),(2,'b'),(3,'c')
123         // Note que está função trabalha com 2 parâmetros genéricos
124         public static IEnumerable<((T, R)> Zip<T, R>(this IEnumerable<T> coll, IEnumerable<T> second)
125         {
126             throw new NotImplementedException();
127         }
128     }
```

Give feedback