

# Перебор элементов в foreach для своих типов (IEnumerable)



admin

December 2021   edited March 18

Принцип работы оператора foreach такой же, как в C#. Через GetEnumerator() получается перечислитель с функциями Current, MoveNext и Reset. Простой пример:

```
Foo f=new Foo();
foreach(int item in f) Write(item+" "); //1 2 3 4 5

class Foo{
    object GetEnumerator(){
        return Enumerator(new MyEnumerator()); //функция Enumerator делает из объекта перечислитель (IEnumerator)
    }
}

class MyEnumerator{
    int X;
    public bool MoveNext(){ //вызывается при каждой итерации (включая первую) в foreach. Должен возвращать true, если перечисление не закончено
        return If(++X<=5, true, false); //If(++X<=5, true, false) - то же самое, что ++X<=5 ? true : false. И так и так можно писать.
    }
    public object Current(){ //возвращает значение текущей итерации
        return X;
    }
    public Reset(){ //сбрасывает значение
        X=0;
    }
}
```

В этом примере перечислитель (enumerator) просто возвращает числа от 1 до 5. Перечислитель создаётся функцией Enumerator из любых объектов имеющих функции Current, MoveNext и Reset. MoveNext должна возвращать bool, а Current - object. У функции Enumerator есть перегрузка, которой подаётся не экземпляр перечислителя, а функции. Функцию GetEnumerator можно переписать так:

```
object GetEnumerator(){
    var e=new MyEnumerator();
    return Enumerator(e.MoveNext(), e.Current(), e.Reset()); //четвёртым аргументом можно ещё указать метод Dispose
    //return EnumeratorByFuncRefs(FuncRef(e.MoveNext()), FuncRef(e.Current()), FuncRef(e.Reset())); //можно так
}
```

## Про Dispose

В отличие от C#, в OverScript foreach не вызывает метод Dispose после завершения или прекращения перечисления. А вот функция ForEach вызывает. Посмотрим на более сложный пример создания списка (list):

```
var list=new List();
list.Add(10);
list.Add(20);
list.Add(30);

list.ForEach(object item, Write(item)); //Item-0: 10; Item-1: 20; Item-2: 30; Dispose!
WriteLine();

foreach(object $item in list) Write(item); //Item-0: 10; Item-1: 20; Item-2: 30; //как видим, foreach не вызвал Dispose
//символ $ перед item разрешает повторное декларирование переменной (в OverScript нет блочной области видимости)
WriteLine();

WriteLine(Join("", list)); //Dispose! Item-0: 10; Item-1: 20; Item-2: 30;
//Join, чтобы соединить строковые представления элементов, перебирает элементы используя перечислитель
//Dispose вывелся первым потому, что сначала завершилась Join, которая вызвала в конце Dispose перечислителя, и только потом WriteLine вывела

class List{
    int Capacity;
    public int Count;
```

```

object[] Items;

New(){
    Clear();
}
public Add(int v){
    if(Count >= Capacity-1){
        Capacity *= 2;
        Resize(Items, Capacity);
    }
    Items[Count]=v;
    Count++;
}

public Clear(){
    Capacity=10;
    Count=0;
    Items=new object[Capacity];
}

public Remove(int index){
    int c=Count-1;
    for(int i=index; i<c; i++) Items[i]=Items[i+1];
    Count=c;
}

object GetEnumerator(){
    var e=new MyEnumerator(Items, Count);
    return Enumerator(e);
}
}

class MyEnumerator{
    object[] Items;
    int Pos=-1, Count;

    New(object[] items, int length){
        Items=items;
        Count=length;
    }
}

```

```

    }
    public bool MoveNext(){
        return If(++Pos < Count, true, false);
    }
    public object Current(){
        return "Item-" + Pos + ": " + Items[Pos] + "; ";
    }
    public Reset(){
        Pos = -1;
    }
    public Dispose(){
        Write("Dispose! ");
    }
}

```

В этом примере перечислитель возвращает не просто значение элемента массива, а строку с этим значением. Если бы нужны были только сами значения, то можно было получить перечислитель так:

```

object GetEnumerator(){
    return Items->GetEnumerator(); //через Reflection получаем перечислитель массива
}

```

Но в таком случае выводились бы значения не в кол-ве Count, а вообще все (Capacity), что не нужно. Поэтому свой перечислитель тут очень уместен.

В некоторых случаях можно вообще обойтись без класса перечислителя:

```

Foo f = new Foo(5);
foreach(int x in f) Write(x + " "); //0 1 2 3 4

class Foo{
    int Pos = -1;
    int Count;
    New(int count){

```

```

        Count=count;
    }
    object GetEnumerator(){
        return Enumerator(GoNext(), GetCurrent(), ResetPos());
    }
    public bool GoNext(){
        return If(++Pos < Count, true, false);
    }
    public int GetCurrent(){
        return Pos;
    }
    public ResetPos(){
        Pos=-1;
    }
}

```

Обратите внимание, что функции Current, MoveNext и Reset названы по-другому. Так можно, когда в функцию Enumerator передаются функции. Тип GetCurrent тут int, а не object, что так же будет работать.

Объект из OverScript можно передать, например, в написанную на C# или VB.NET библиотеку, и этот внешний код сможет работать с ним так же, как с любым другим IEnumerable, вызывая при этом ваши OverScript-функции.

Код библиотеки (C#):

```

using System;
public class EnumerableTest
{
    public static void Start(System.Collections.IEnumerable e)
    {
        int n = 0;
        foreach(object item in e)
            Console.WriteLine($"Item-{n++}: " + item);

        Console.WriteLine("End!");
    }
}

```

```
}
```

Переделанный пример:

```
const object lib=typeof(@"C:\tests\TestLib.dll", "EnumerableTest");

Foo f=new Foo(5);
lib->Start(f); //передаём библиотечной функции, которая принимает System.Collections.IEnumerable, экземпляр Foo (формально все экземпляры в O
//Item-0: 0
//Item-1: 1
//Item-2: 2
//Item-3: 3
//Item-4: 4
//End!

class Foo{
    int Pos=-1;
    int Count;

    New(int count){Count=count;}
    object GetEnumerator(){return Enumerator(GoNext(), GetCurrent(), ResetPos());}
    bool GoNext(){return If(++Pos < Count, true, false);}
    int GetCurrent(){return Pos;}
    ResetPos(){Pos=-1;}
}
```

Код в библиотеке на каждой итерации foreach вызывает ваши GoNext и GetCurrent. Тут нет ничего особенного, ведь OverScript работает поверх .NET, а вызов ваших функций - это обычный вызов делегатов ( Func<bool>, Func<object>, Action ), в которых находятся функции интерпретатора, выполняющие код ваших функций.