Reflection с помощью стрелки

admin
February 3 edited April 2

В OverScript стрелка (->) при загрузке кода заменяется на вызов reflection-функций.

```
WriteLine("test"->ToUpper()); //TEST
//Посмотрим, как видит стрелку интерпретатор:
WriteLine(Expr("test"->ToUpper())); //@TInvoke(#System.String ToUpper()#,string,"test")
```

Напомню, что @ перед именем функции просто указывает, что нужно вызвать базовую (встроенную) функцию без поиска перегрузки в коде (можно сказать, запрещает перегрузку базовой функции).
Функция TInvoke вызывает MethodBase.Invoke. Первым аргументом идёт MethodInfo, т.е. объект с информацией о методе. В примере мы видим `#System.String ToUpper()#`. Это так выводится object-литерал с MethodInfo. Вторым аргументом идёт тип string для того, чтобы результат вызова MethodBase.Invoke приводился к типу string, и функция TInvoke возвращала string-значение. Есть обычная Invoke, которая возвращает object. Третий аргумент - литерал "test", на котором вызывать ToUpper.
Интерпретатор находит типы и их члены во время загрузки кода, если он сразу знает тип объекта, к которому применяется стрелка. Если тип неизвестен, то применяются общие функции, которые делают всё во время выполнения кода.

```
object obj="test";
WriteLine(obj->ToUpper()); //TEST
//Тут стрелка заменилась не на TInvoke, а на InvokeMethod:
WriteLine(Expr(obj->ToUpper())); //@InvokeMethod(obj,"ToUpper")
```

Функция InvokeMethod определяет тип объекта, находит и вызывает нужный метод. Это делается при каждом обращении, поэтому InvokeMethod работает медленнее, чем Invoke/TInvoke. К тому же результат будет object.

```
object obj="test";
//string s=obj->ToUpper(); //Can't put value of type 'object' into variable of type 'string'.
string s:=obj->ToUpper(); //присваивание с конвертированием
//Или так:
s=string\obj->ToUpper(); //приведение типа. Это то же, что и (string)obj->ToUpper(). На самом деле, это вызов функции: op_StringCasting(obj->ToUpper()).
```

Приведение типа само по себе работает быстрее конвертирования, но в целом получается медленнее из-за того, что дополнительно делается вызов функции (op_StringCasting).

Теперь пример вызова функции из System.Math:

```
const object Math=typeof("System.Math");
WriteLine(Math->Abs(-5)); //5
WriteLine(Expr(Math->Abs(-5))); //@TInvoke(#Int32 Abs(Int32)#,int,,-5)
```

Объект с типом Math - константа, а значит интерпретатор заранее знает тип и использует функцию TInvoke. Функция Abs статическая, поэтому объект, на котором её выполнять, пропущен. Уберём const:

```
object Math=typeof("System.Math");
WriteLine(Expr(Math->Abs(-5))); //@InvokeMethod(Math,"Abs",-5)
```

Теперь интерпретатор не знает тип при загрузке и использует InvokeMethod. Если интерпретатор знает тип, но среди аргументов есть object-ы, то всегда используется InvokeMethod, т.к. сразу найти перегрузку не получится, ведь в object-е может быть всё что угодно. Если параметр функции object, то для аргумента можно использовать подсказу типа, чтобы вызывалась TInvoke, а не InvokeMethod:

```
const object Object=typeof("System.Object");
const object ArrayList=typeof("System.Collections.ArrayList");
`ArrayList list=ArrayList.Create();
object x=123; //это обычная object-переменная
`Object y=123; //а это тоже object, но с подсказкой типа
WriteLine(Expr(list->Add(x))); //@InvokeMethod(list,"Add",x)
WriteLine(Expr(list->Add(y))); //@TInvoke(#Int32 Add(System.Object)#,int,list,y)
```

Казалось бы, и то и то object, но с x интерпретатор не знает, какую перегрузку метода искать, а с y будет искать метод с параметром, соответствующим типу подсказки.

Есть одно ограничение: не получится вызывать методы с передачей ссылки на переменную:

```
int x=0;
int->TryParse("5", x); //Ошибка! TryParse("5", ref x) тоже не сработает
```

От вызова функций перейдём к **обращению к полям и свойствам**.

```
WriteLine(int->MaxValue); //2147483647 //кстати, можно так: 5->MaxValue
WriteLine(Expr(int->MaxValue)); //2147483647

WriteLine("test"->Length); //4
WriteLine(Expr("test"->Length)); //@TGetValue(#Int32 Length#,int,"test")
```

Из примера видно, что int->MaxValue сразу заменяется в коде на литерал 2147483647, т.к. это константа. TGetValue - это как TInvoke, только для полей/свойств. Есть обычный GetValue, который возвращает object.

Эти функции вызывают FieldInfo.GetValue или PropertyInfo.GetValue.
Если делается присваивание, то используется функция SetValue, которая вызывает FieldInfo.SetValue или PropertyInfo.SetValue.

```
const object Environment=typeof("System.Environment");
Environment->CurrentDirectory=@"F:\docs\test";
WriteLine(Environment->CurrentDirectory); //F:\docs\test
//Посмотрим, как выглядит для интерпретатора присваивание:
WriteLine(Expr(Environment->CurrentDirectory=@"F:\docs\test")); //@SetValue(#System.String CurrentDirectory#,,"F:\\docs\\test")
```

Для событий нужно использовать операторы += и -=. Первый использует функцию AddEventHandler, а второй вызывает RemoveEventHandler.

```
const object Console=typeof("System.Console, System.Console");
object d=Delegate(OnConsoleCancel(object\, object\), Console->CancelKeyPress->EventHandlerType);

Console->CancelKeyPress += d; //нужно именно +=, а не =
WriteLine("Press Ctrl+C...");
Sleep(-1); //-1 - это Timeout.Infinite

OnConsoleCancel(object sender, object args){
    WriteLine("Canceled!");
    Sleep(1000);
    WriteLine("Bye-Bye!");
    Sleep(1500);
}
```

В этом примере перехватывается нажатие Ctrl+C. Если написать так:

```
Console->CancelKeyPress += d;
Console->CancelKeyPress += d;
```

то в событии будет два делегата. Если написать:

```
WriteLine(Console->CancelKeyPress); //System.ConsoleCancelEventHandler CancelKeyPress
```

то будет выведено строковое представление объекта EventInfo (`Console->CancelKeyPress` возвращает именно EventInfo). Если член не поле и не свойство, то возвращается MemberInfo, а если это тип, то возвращается Type.
А как, например, **очистить событие**? Для этого нужно знать поле, в котором хранятся делегаты события. В нашем случае это поле s_cancelCallbacks. Обнулить его можно так:

```
object field=Console->@GetField("s_cancelCallbacks", 40.ToEnum("System.Reflection.BindingFlags"));
field.SetValue(Console, null);
```

```
//Так можно получить делегат и потом сделать Delegate.GetInvocationList():
//foreach(object item in field.GetValue(Console)->GetInvocationList()) WriteLine(item);
//Передавать Console не обязательно, т.к. поле s_cancelCallbacks статическое:
//field.SetValue(, null);
//foreach(object item in field.GetValue()->GetInvocationList()) WriteLine(item);
```

Для поиска приватного поля используем Type.GetField. Кроме имени функции передаём ей BindingFlags. Здесь `40.ToEnum("System.Reflection.BindingFlags")` - это BindingFlags.NonPublic | BindingFlags.Static (32+8). Потом для обнуления поля используем SetValue, передавая третьим аргументом null (SetValue(field, Console, null)).
Далее про то, зачем символ @ перед GetField. При использовании стрелки @ указывает, что нужно искать член не у самого типа, а у типа типа (type.GetType()), т.е. у System.RuntimeType. Если в примере выше написать без @, то будет ошибка: "Member 'GetField' not found at 'System.Console'". А с @ будет поиск в typeof(System.Console).
Поясню на простом примере.
Есть класс в DLL (C#):

```
public class SimpleTest
{
    public static string ToString() => "Hello!";
}
```

Код OverScript:

```
const object SimpleTest=typeof("SimpleTestLib.dll", "SimpleTest");
WriteLine(SimpleTest->ToString()); //Hello!
WriteLine(SimpleTest->@ToString()); //SimpleTest
```

Третья строка в C# выглядела бы так: `typeof(SimpleTest).ToString()`.

Теперь посмотрим, как работает **обращение к элементам массивов**.
DLL (C#):

```
public class SimpleTest
{
    public static int[] Arr = { 100, 200, 300 };
}
```

Код OverScript:

```
const object SimpleTest=typeof("SimpleTestLib.dll", "SimpleTest");
WriteLine(SimpleTest->Arr[2]); //300
WriteLine(Expr(SimpleTest->Arr[2])); //@TGetElement(#Int32[] Arr#,int,,2)
WriteLine((SimpleTest->Arr)[2]); //можно и так
```

Видим, что для получения значения элемента массива используется функция TGetElement. В этом примере массив static, поэтому третий аргумент пропущен.

```
const object SimpleTest=typeof("SimpleTestLib.dll", "SimpleTest");
SimpleTest->Arr[2]=555;
WriteLine(SimpleTest->Arr[2]); //555
WriteLine(Expr(SimpleTest->Arr[2]=555)); //@SetElement(#Int32[] Arr#,,555,2)
```

При присваивании используется SetElement.
Для многомерных массивов (не путать с массивами массивов) нужно указывать несколько индексов: `SimpleTest->Arr[2, 3]`.
Пример для **массива массивов**.
DLL (C#):

```
public class SimpleTest
{
    public static int[][] JagArray = new int[][] { new int []{ 10, 20, 30 }, new int[] { 100, 200, 300 } };
}
```

Код OverScript:

```
int[] arr:=SimpleTest->JagArray[1]; // := потому, что используется @GetElement(#Int32[][] JagArray#,,1), возвращающая object. Перегрузок TGetElement для возврата массивов я не ст
WriteLine(arr[2]); //300
arr[2]=567;
WriteLine(arr[2]); //567
```

Можно ещё так написать:

```
WriteLine((int[]\SimpleTest->JagArray[1])[2]); //300
(int[]\SimpleTest->JagArray[1])[2]=567;
WriteLine((int[]\SimpleTest->JagArray[1])[2]); //567
```

А теперь про **обращение к элементам списка**.
DLL (C#):

```
public class SimpleTest
{
    public static List<int> List = new List<int>() {100, 200, 300};
}
```

Код OverScript:

```
const object SimpleTest=typeof("SimpleTestLib.dll", "SimpleTest");
WriteLine(SimpleTest->List->Item[2]); //300
WriteLine(Expr(SimpleTest->List->Item[2])); //@TGetValue(#Int32 Item [Int32]#,int,@GetValue(#System.Collections.Generic.List`1[System.Int32] List#),2)
```

Item - это свойство-индексатор. Если написать `SimpleTest->List[2]`, то будет ошибка. Давайте проверим со своим индексатором.
DLL (C#):

```
public class SimpleTest
{
    public int this[int x, int y]
    {
        get { return x + y; }
    }
}
```

Код OverScript:

```
const object SimpleTest=typeof("SimpleTestLib.dll", "SimpleTest");
object obj=SimpleTest.Create();
WriteLine(obj->Item[2, 5]); //7
```

Про **конвертирование значений** при присвоении.

```
const object Environment=typeof("System.Environment");
//Environment->ExitCode=777L; //ошибка. Свойство ExitCode типа int, а 777L - long.
Environment->ExitCode:=777L;
WriteLine(Environment->ExitCode);
WriteLine(Expr(Environment->ExitCode:=777L)); //@SetValue(#Int32 ExitCode#,,@ChangeType(777,#System.Int32#))
```

Присваиваемое значение оборачивается в функцию ChangeType, которая конвертирует его в тип свойства, который в нашем случае Int32 у ExitCode.

Shorthand-операторы (+=, *= и т.д.) при использовании стрелки не поддерживаются (только += и -= для событий).