Создание DLL и импорт функций из неё

admin
February 6 edited March 18

В OverScript есть базовые функции, которые вшиты в интерпретатор. Дополнительно к ним можно импортировать функции из своей DLL. Можно обходится без импортирования, вызывая функции из библиотек через рефлекшн, но такие вызовы в 7 раз медленнее вызова базовых функций.
Импорт функций из DLL делается директивой #import: `#import "MyLib.dll"` . Можно полный путь указывать: `#import @"D:\files\MyLib.dll"` . Собака указывает, что escape-последовательности в строке не обрабатываются. В пути можно использовать любые переменные среды, а также специальные:
1. %APPDIR% - путь к папке, в которой лежит выполняемый скрипт;
2. %MODDIR% - путь к папке модулей (modules в папке интерпретатора);
3. %OVSDIR% - путь к интерпретатору;
4. %CURDIR% - путь к текущей рабочей папке (Environment.CurrentDirectory).

Например: `#import @"%MODDIR%\MyLib.dll"` загружает MyLib.dll из папки modules. Из этой папки можно загружать и так: `#import <MyLib.dll>` .

Сразу покажу пример очень простой библиотеки:

```
using System;
using OverScript;

namespace OSAdditionalFuncs
{

    [Import] //помечаем атрибутом класс, в котором находится функция импорта
    public class MainClass
    {
        [Import] //помечаем атрибутом функцию импорта
        public static void AddFuncs(Action<string,BF> addFunc)
        {
            addFunc("Hello", new BF().Add(Hello_string, TypeID.String)); //добавление функции Hello к базовым функциям. new BF() - это создание новой базовой функции, а метод Add

        }
        private static string Hello_string(EvalUnit[] fnArgs, int scope, ClassInstance srcInst, ClassInstance inst, CallStack cstack, EvalUnit csrc)
        {
            string name = fnArgs[0].EvalString(scope, inst, cstack); //вычисляем первый аргумент (по индексу 0)

            return $"Hello, {name}. Today is {DateTime.Now}.";
        }

    }
```

```
}
```

Положили нашу OSAdditionalFuncs.dll в папку со скриптом и тестим:

```
#import "OSAdditionalFuncs.dll"

WriteLine(Hello("Jack")); //Hello Jack. Today is 06.02.2022 17:39:04.
```

Интерпретатор ищет в DLL тип с атрибутом [Import], а потом ищет у этого типа метод с атрибутом [Import], и найдя, выполняет этот метод. В метод импорта передаётся делегат addFunc, с помощью которого можно добавлять функции. Метод который мы добавляем - Hello_string. Все добавляемые методы должны иметь набор параметров как у него. Что это за параметры:
1. EvalUnit[] fnArgs - аргументы, передаваемые функции. Это не значения, а эвал-юниты (единицы вычисления: выражения или просто переменные). Не знаю, как их назвать по-другому, поэтому так и буду называть;
2. scope - номер области видимости, из которой вызывается функция;
3. ClassInstance srcInst - не используется, т.к. нужен только при вызове небазовых функций;
4. ClassInstance inst - экземпляр, из которого делается вызов;
5. CallStack cstack - стек вызовов
6. EvalUnit csrc - выражение, из которого делается вызов.

Чтобы получить значение аргумента нужно вычислить эвал-юнит. В примере это делается в строке: `string name = fnArgs[0].EvalString(scope, inst, cstack);` .

В директиве импорта можно задать что-то вроде пространства имён для доступа к функции:

```
#import "OSAdditionalFuncs.dll" as Lib

WriteLine(Lib@Hello("Jack")); //Hello Jack. Today is 06.02.2022 17:39:04.
```

Теперь имя функции - Lib@Hello. Как видим, используется не точка, как обычно, а собака. Это позволяет визуально сразу определить, что делается вызов импортированной функции. С точкой можно было бы принять за вызов метода экземпляра.
А можно так:

```
#import "OSAdditionalFuncs.dll" as Lib_*

WriteLine(Lib_Hello("Jack")); //Hello Jack. Today is 06.02.2022 17:39:04.
```

Т.е. так ко всем именам функций при импорте добавится Lib_.

В директиве импорта можно указать имя класса импорта:

```
#import "OSAdditionalFuncs.dll" as Lib, from=MainClass
```

Можно указать и имя метода импорта:

```
#import "OSAdditionalFuncs.dll" as Lib, from=MainClass, call=AddFuncs
```

Можно указывать несколько функций импорта:

```
#import "OSAdditionalFuncs.dll" as Lib, call=AddFuncs|AddFuncs2|AddFuncs3
```

Можно указать имена функций, которые импортировать:

```
#import "OSAdditionalFuncs.dll" as Lib, funcs=Hello|SomeFunc|Test
```

Можно разрешить замену уже существующих функций:

```
#import "OSAdditionalFuncs.dll" as Lib, override=true
```

Подробно про добавление функций

В примере выше есть строка:

```
addFunc("Hello", new BF().Add(Hello_string, TypeID.String));
```

Это самый простой вариант, в котором для функции с именем "Hello" добавляется одна перегрузка. Можно добавлять несколько перегрузок. Допишем в библиотеке новые перегрузки:

```
using System;
using OverScript;
using static OverScript.BasicFunctions; //чтобы можно было метод EvalArgs использовать

namespace OSAdditionalFuncs
{

    [Import]
    public class MainClass
    {
        [Import]
        public static void AddFuncs(Action<string,BF> addFunc)
```

```
        {
            addFunc("Hello", new BF().Add(Hello_string, TypeID.String)
                                     .Add(Hello_string_int, TypeID.String, TypeID.Int)
                                     .Add(Hello_string_long, TypeID.String, TypeID.Long)
                                     .Add(Hello_string_objectParams, TypeID.String).HasParams(TypeID.Object)
                    );
        }
        private static string Hello_string(EvalUnit[] fnArgs, int scope, ClassInstance srcInst, ClassInstance inst, CallStack cstack, EvalUnit csrc)
        {
            string name = fnArgs[0].EvalString(scope, inst, cstack);
            return $"Hello, {name}. Today is {DateTime.Now}.";
        }
        private static string Hello_string_int(EvalUnit[] fnArgs, int scope, ClassInstance srcInst, ClassInstance inst, CallStack cstack, EvalUnit csrc)
        {
            string name = fnArgs[0].EvalString(scope, inst, cstack); //вычисляем первый аргумент (по индексу 0)
            int num = fnArgs[1].EvalInt(scope, inst, cstack); //вычисляем второй аргумент (по индексу 1)
            return $"Hello, {name}. Int arg is {num}.";
        }
        private static string Hello_string_long(EvalUnit[] fnArgs, int scope, ClassInstance srcInst, ClassInstance inst, CallStack cstack, EvalUnit csrc)
        {
            string name = fnArgs[0].EvalString(scope, inst, cstack);
            long num = fnArgs[1].EvalLong(scope, inst, cstack);
            return $"Hello, {name}. Long arg is {num}.";
        }
        private static string Hello_string_objectParams(EvalUnit[] fnArgs, int scope, ClassInstance srcInst, ClassInstance inst, CallStack cstack, EvalUnit csrc)
        {
            string name = fnArgs[0].EvalString(scope, inst, cstack);
            object[] arr =EvalArgs(1, fnArgs, scope, inst, cstack); //EvalArgs возвращает массив значений аргументов начиная с заданного индекса (1)
            //int[] arr = EvalArgs<int>(1, fnArgs, scope, inst, cstack); //если парамсы, например, int, а не не object
            return $"Hello, {name}. Params: {string.Join(", ", arr)}.";
        }

    }
}
```

Тестим:

```
#import "OSAdditionalFuncs.dll"

WriteLine(Hello("Jack")); //Hello, Jack. Today is 21.02.2022 18:44:28.
WriteLine(Hello("Jack", 123)); //Hello, Jack. Int arg is 123.
WriteLine(Hello("Jack", 567L)); //Hello, Jack. Long arg is 567.
WriteLine(Hello("Jack", 777, "test", 4.9)); //Hello, Jack. Params: 777, test, 4,9.
```

У Hello есть 4 перегрузки. Перегрузка Hello_string_objectParams принимает params. Можно и так их передать:

```
WriteLine(Hello("Jack", new object[]{777, "test", 4.9}));
```

В AddFuncs для этой перегрузки params устанавливаются методом HasParams:

```
.Add(Hello_string_objectParams, TypeID.String).HasParams(TypeID.Object)
```

Перегрузки, Hello_string_int и Hello_string_long могут принимать не только int и long:

```
WriteLine(Hello("Jack", byte\123)); //Hello, Jack. Int arg is 123.
WriteLine(Hello("Jack", decimal\123)); //Hello, Jack. Long arg is 123.
```

Значение типа byte принимает Hello_string_int, a decimal - Hello_string_long (если передать число больше long.MaxValue, то будет ошибка при выполнении). В C# функция с long параметром не может принимать decimal.
Можно задать строгое совпадение типа параметра и аргумента:

```
.Add(Hello_string_int, TypeID.String, TypeID.Int).Strict(1) //1 - индекс аргумента (можно несколько указывать). Если вообще не указывать, то все параметры будут принимать строго
```

Теперь для Hello_string_int второй аргумент должен быть строго типа int.