FuncRef, FuncPureRef, Call и CallOn - ссылки на функции и их вызов

admin
December 2021 edited March 18

В OverScript функции нельзя использовать как объекты, но можно получать на них ссылки. Вызывать функции по ссылкам можно как на их родных объектах, так и на экземплярах любых своих типов.
Создаётся ссылка функцией FuncRef, а вызвать функцию по ссылке можно функцией Call.
Сначала простые примеры:

```
object fn=FuncRef(WriteLine("")); //ссылка на WriteLine(object), т.к. перегрузки для string нет
fn.Call("Hello, world!"); //Hello, world!

fn=FuncRef(Ceiling(0.0)); //ссылка на Ceiling(double), которая возвращает тип double
object x=fn.Call(4.9);
WriteLine(x+"; "+x.GetType()); //5; System.Double

fn=FuncRef(Ceiling(0.0F)); //ссылка на Ceiling(float), которая возвращает тип float (Single)
x=fn.Call(4.9);
WriteLine(x+"; "+x.GetType()); //5; System.Single

ReadKey();
```

Из примера видно, что функции для ссылок нужно указывать с аргументами, которые нужны для поиска нужной перегрузки. Имеют значение только их типы, т.е. можно написать `FuncRef(Ceiling(0.0))`, а можно `FuncRef(Ceiling(5.0))` - результат будет один и тот же.
При вызове функции нужно передать аргументы того типа, которые она принимает. Многие функции конвертируют значения в нужные типы, но некоторые функции предназначены строго для определённых типов и работать не будут. Поэтому нужно внимательно относиться к типам передаваемых аргументов.
Лучше всего, чтобы случайно не указать аргумент не того типа, задавать типы так:

```
object fn=FuncRef(Test(int\));
fn.Call(123); //int: 123
fn=FuncRef(Test(string\));
fn.Call(123); //string: 123 //число автоматически конвертировалось в строку

Test(int v){
    WriteLine("int: "+v);
}
Test(string v){
    WriteLine("string: "+v);
}
```

В этом примере есть `int\` и `string\`. Когда после типа идёт обратный слеш, то это значение по умолчанию для типа. Т.е. `int\` - это как `default(int)` в C#, а `string\` - как `default(string)`. А что будет, если указать тип без слеша (`FuncRef(Test(int)`)? Выскочит ошибка, т.к. интерпретатор будет искать перегрузку для object, ведь int это object содержащий тип.

```
object type=int; //в C# так не получится
WriteLine(type); //System.Int32
```

Теперь посмотрим, как ссылка на функцию захватывает ссылку на экземпляр:

```
Foo foo=new Foo("Hello!");
object fn=FuncRef(foo.Print());
fn.Call(); //Hello!
foo=new Foo("Test!");
fn.Call(); //Hello! //то же самое, хотя в foo новый экземпляр

class Foo{
    string Text;
    New(string text){
        Text=text;
    }
    public Print(){
        WriteLine(Text);
```

```
    }
}
```

FuncRef вернула ссылку на Print конкретного экземпляра. Если перезаписать foo новым экземпляром, вызываться будет Print на том экземпляре, который был в foo на момент вызова FuncRef.
Для получения ссылки на функцию без захвата конкретного экземпляра, есть функция FuncPureRef.

```
Foo foo=new Foo("Hello!");
object fn=FuncPureRef(foo.Print());
fn.CallOn(foo); //Hello!
foo=new Foo("Test!");
fn.CallOn(foo); //Test!

class Foo{
    string Text;
    New(string text){
        Text=text;
    }
    public Print(){
        WriteLine(Text);
    }
}
```

В этом примере вместо Call используем CallOn для вызова функции на указанном экземпляре. Кстати, CallOn можно использовать и для ссылок полученных через FuncRef, а не через FuncPureRef.
CallOn можно использовать с экземпляром любого класса.

```
Foo foo=new Foo();
object fn=FuncPureRef(foo.Test());
fn.CallOn(foo); //Instance of Foo
Bar bar=new Bar();
fn.CallOn(bar); //Instance of Bar

class Foo{
```

```
    public Test(){
        WriteLine(this);
    }
}
class Bar{}
```

Ещё пример:

```
object fn=FuncPureRef(Test());
fn.CallOn(new Foo()); //Instance of Foo; X=10
fn.CallOn(new Bar()); //Instance of Bar; X=20

int X;
public Test(){
    WriteLine(this+"; X="+X);
}

class Foo{
    int X=10;
}
class Bar{
    int X=20;
}
```

Динамический вызов функций и динамическое получение ссылок

Динамически вызвать метод объекта можно функцией DynCall. Для динамического получения ссылки на функцию нужно использовать перегрузки FuncRef и FuncPureRef, принимающие более одного аргумента.
Посмотрим на примере:

```
object obj=new Foo();
//вызовем функцию Test(567) экземпляра в obj через DynCall(объект, имя_функции, аргумент_0, аргумент_1, ...)
//DynCall каждый раз ищет функцию по имени и типам аргументов
WriteLine(DynCall(obj, "Test", 567)); //At Foo: 567
obj=new Bar();
```

```
WriteLine(DynCall(obj, "Test", 567)); //At Bar: 567
WriteLine(DynCall(obj, "Test", 567L)); //At Bar: 567 long
WriteLine();
//теперь динамически получим ссылки на функции и выполним их обычными Call и CallOn
object fn=FuncRef(obj, "Test", long\); //получаем ссылку на Test(long) в Bar
WriteLine(fn.Call(567)); //At Bar: 567 long
obj=new Foo();
fn=FuncRef(obj, "Test", long\); //получаем ссылку на Test(int) в Foo (int потому, что для long нет перегрузки)
WriteLine(fn.Call(567)); //At Foo: 567

fn=FuncPureRef(obj, "Test", long\); //получаем ссылку без привязки к экземпляру
WriteLine(fn.CallOn(obj, 567)); //At Foo: 567

class Foo{
    static string Test(int x){return "At Foo: "+x;}
}
class Bar{
    public string Test(int x){return "At Bar: "+x;}
    string Test(long x){return "At Bar: "+x+" long";}
}
```

Как видите, можно вызывать не только публичные, но и приватные методы. А ещё можно вызывать функции статических классов:

```
object obj=Foo;
WriteLine(DynCall(obj, "Test", 555)); //At Foo: 555
obj=Bar;
WriteLine(DynCall(obj, "Test", 555)); //At Bar: 555

static class Foo{
    static string Test(int x){return "At Foo: "+x;}
}
static class Bar{
    static string Test(int x){return "At Bar: "+x;}
}
```

Кстати, CallOn тоже можно вызывать на статических классах:

```
Foo.PrintX(); //555
object fn=FuncPureRef(new Bar().SetX(0));
fn.CallOn(Foo, 777); //вызываем SetX из Bar на статическом классе Foo
Foo.PrintX(); //777

static class Foo{
    static int X=555;
    public static PrintX(){WriteLine(X);}
}
class Bar{
    int X;
    public SetX(int x){X=x;} //функция и X не статические, а экземплярные, т.к. статические переменные жестко привязаны к своему классу, и пр
}
```

Этот пример может показаться очень странным, но на самом деле у каждого класса есть статический экземпляр (оксюморон, но, думаю, смысл понятен), в котором хранятся статические переменные. Этот экземпляр создаётся при первом обращении к его членам.

DynCall и динамические FuncRef и FuncPureRef работают медленно, т.к. при каждом вызове ищут нужную функцию. Обычные FuncRef и FuncPureRef (которые с одним аргументом-функцией) не ищут функцию, а просто получают её из выражения, в котором она уже найдена при загрузке кода.

Класс-интерфейс

В OverScript интерфейсов, как в C#, пока нет, но можно делать классы, имитирующие интерфейсы. Суть: два класса Bar и Baz наследуют от одного Foo, который у нас будет как бы интерфейсом. Мы сможем вызывать функцию Test не напрямую, а приводя объект к типу Foo.

```
Bar br=new Bar();
Baz bz=new Baz();
//проверим прямые вызовы:
br.Test(5); //At Bar: 5
bz.Test(5); //At Baz: 5
```

```
//а теперь вызовы через Test в Foo:
(Foo\br).Test(5); //At Bar: 5
(Foo\bz).Test(5); //At Baz: 5
//Напомню, что Foo\br - это то же самое, что (Foo)br. Можно и так и так писать:
((Foo)br).Test(5); //At Bar: 5
((Foo)bz).Test(5); //At Baz: 5

class Foo{
    object TestFn=FuncRef(this, "Test", int\);
    public Test(int x){
        TestFn.Call(x);
    }
}
class Bar:Foo{
    public Test(int x){
        WriteLine("At Bar: "+x);
    }
}
class Baz:Foo{
    public Test(int x){
        WriteLine("At Baz: "+x);
    }
}
```

Как интерпретатор видит классы Bar и Baz:

```
class Bar{
    object TestFn=FuncRef(this, "Test", int\);
    public Test(int x){
        TestFn.Call(x);
    }
    public Test(int x){
        WriteLine("At Bar: "+x);
    }
}
class Baz{
```

```
    object TestFn=FuncRef(this, "Test", int\);
    public Test(int x){
        TestFn.Call(x);
    }
    public Test(int x){
        WriteLine("At Baz: "+x);
    }
}
```

Получается по две Test в каждом. Выполняется всегда последняя. FuncRef тоже берёт ссылку на последнюю. В этом примере первые вообще не используются, а значит метод Test в Foo можно не наследовать, пометив его модификатором **exclusive**:

```
public exclusive Test(int x){
    TestFn.Call(x);
}
```

Так он будет только у Foo. Это сократит время загрузки кода и занимаемую им оперативную память.
