vJass 系列教程 10

函数对象和委托对象

Aeris ▸ NJU ▸ 2009/2/3

vJass 系列教程 10

函数对象和委托对象

本章介绍的函数对象是 vJass 中语法简单，但用处很大的特性之一。委托对象则在某些应用场合给我们提供了方便。

函数对象(function object)

函数作为对象

语法

对象的一个特征是，对象具有属性和方法，可以使用“.”操作符进行访问。函数对象则是把函数也化为了对象，可以用“.”操作，像对待对象一样对待函数。

函数唯一的用途就是调用，所以函数对象的语法也是**调用函数的语法**，很简单，就两条：

- execute 语法

```
函数名.execute(参数列表)
```

- evaluate 语法

```
函数名.evaluate(参数列表)
```

对比一下正常的函数调用语法：

```
函数名(参数列表)
```

这 3 种语法都是用来调用函数的，第 3 种语法是大家熟悉的。前面两种语法 execute 和 evaluate 的不同点在于，execute 语法是用来调用**无返回值函数**的，而 evaluate 语法是调用**有返回值函数**的。另外，如果使用 evaluate 语法调用函数，那么目标函数中**不能等待**。

例如，我定义了一个函数叫 Calculate 如下：

```
function Calculate takes integer a, integer b, integer c returns integer
    return a + b + c
endfunction
```

现在我要调用这个函数，那么我现在有两种选择：

1. 使用正常语法：`set result = Calculate(a, b, c)`
2. 使用 evaluate 语法：`set result = Calculate.evaluate(a, b, c)`

因为这个函数有返回值，我们不能用 execute 语法。

应用与注意点

为什么要用函数对象语法，函数对象的语法究竟有没有什么特别的优势呢？答案当然是有的，函数对象的优势是：

- **无视函数定义顺序，可以自由调用**
- **内部使用触发实现，可以规避单触发 300000 字节码限制**

关于这两个优势，我们分别来看两个例子：

第一个例子，我写了两个函数，A 和 B，函数 A 需要调用函数 B，可麻烦的是，函数 B 也要调用函数 A！也就是说，无论函数 A 和 B 的定义顺序如何，总会出现要么无法调用 A，要么无法调用 B 的情况。如果是普通 Jass，那么是很麻烦的。这时函数对象的优势就体现出来了，因为它无视函数定义顺序。例子代码如下，这个例子是**递归调用（自己调用自己）**和**互相调用**的结合。

```
function A takes real x returns real
    if (GetRandomInt(0, 1) == 0) then
        return B.evaluate(x * 0.02) // 这里写成 return B(x * 0.02)是不行的
    else
        return x
    endif
endfunction

function B takes real x returns real
    if (GetRandomInt(0, 1) == 1) then
        return A(x * 1000)
    else
        return x
    endif
endfunction
```

第二个例子，我写了一个函数 Func，这个函数里面有个 200 次的循环，而麻烦的是，循环中还要调用另一个函数 Calc，而这个函数里竟然又有一个 500 次的循环！这么一搞，这个触发肯定会超出 300000 字节码限制[1]而被强行终止。

怎么办呢？这时我们想到了函数对象，由于函数对象是**使用触发来实现**的，被调用函数的代码会**在另一个单独的触发中运行**。运行过程中对调用者的字节码计数器没有任何影响，这样就规避了 300000 字节码限制。

代码如下：

[1] 300000 字节码限制：魔兽脚本中，触发最多执行 300000 个字节码（不是语句），超过的话，触发会被强行终止，另外如果是地图的启动代码超过限制，魔兽很有可能会直接崩溃。与之对应的是，AI 脚本中，一个线程如果执行了 300000 个字节码，会被强制休眠 1 秒钟（而不像触发会被直接终止）。不过，如果触发中调用等待函数的话，那么触发的字节码计数器会被重置为 0。

```
function Func takes nothing returns integer
    local integer i = 0
    local integer sum = 0

    loop
        exitwhen (i >= 200)  // 会循环200次
        set sum = sum + Calc.evaluate(i)// 假设Calc函数中会循环500次，直接调用肯定会超出300000字节码限制
                                        // 使用evaluate之后，就没有问题了
        set i = i + 1
    endloop

    return sum
endfunction
```

- **注意：正因为函数对象是使用触发实现的，所以被调函数中不能调用 GetTriggeringTrigger()函数，否则结果肯定不正确。如果确实需要获取当前触发，那么应当在调用者函数中获取，然后放到函数参数里或者使用全局变量传递到被调函数里。**

使用函数对象的一个小缺点是：它比直接调用要略微慢一点，因为是通过触发来运行的。

函数接口(function interface)

语法

函数接口是什么？如果读者学过 C 语言的话，一定知道**函数指针**的概念，vJass 的函数接口实现的功能和 C 语言中的函数指针完全相同。如果读者没有学过 C 语言也没关系，看了本节就知道是怎么回事了。

函数接口的语法也很简单，以下是声明一个函数接口：

```
function interface 函数接口名 takes 参数 returns 返回值
```

细心的读者会发现，函数接口的语法和定义函数几乎完全相同，只是在 function 关键字后面加上了关键字 interface 而已。没错，正是这样。函数接口是一种数据类型，它的一个对象可以用来**代表和它具有相同参数和返回值的这一类函数中任何一个**。

例如，我声明一个函数接口如下：

```
function interface Action takes unit actor returns nothing
```

那么，Action 就是一个函数接口，我可以声明一个 Action 类型的对象 a：

```
local Action a
```

现在这个 a 变量就可以用来代表任何一个接受 1 个 unit 类型参数并且没有返回值的函数。

好，现在假设我有这样一个函数：

```
function KillTarget takes unit u returns nothing
    call KillUnit(u)
```

```
endfunction
```

很明显，这个函数 KillTarget 接受 1 个 unit 类型参数，没有返回值，和函数接口 Action 的规格说明一致（再看下 Action 的定义，它要求函数接受 1 个 unit 类型参数，无返回值），那么变量 a 就可以代表 KillTarget，也就是可以赋值为 KillTarget。

给函数接口对象赋值的语法是：

```
set(或 local) 变量名 = 函数接口名.函数名
```

比如对上面声明的变量 a，我们可以这样赋值：

```
set a = Action.KillTarget
```

这样，变量 a 就代表 KillTarget 函数了（学过 C 语言的读者可以想象为：函数指针 a 指向了函数 KillTarget）

好，现在变量 a 代表了一个函数，那么我们怎么通过变量 a 来调用（而不是直接调用）它代表的函数呢？调用一个函数接口所代表函数的语法如下：

```
变量名.execute(参数)    // 当这个函数接口代表无返回值函数时
或者
变量名.evaluate(参数)   // 当这个函数接口代表有返回值函数时
```

发现了么？调用函数接口的语法和**函数对象**的语法**完全相同**！仅仅是把函数名换成了变量名。假如我们要调用 a 代表的函数，就可以用如下语法：

```
call a.execute(某个单位)
```

因为 Action 所规定的函数是无返回值的，所以不能用 evaluate。

注意：和函数对象的注意点一样，1.目标函数中不能使用 GetTriggeringTrigger 函数；2.如果是 evaluate，那么目标函数不能等待。这里再提醒一遍。

到现在，读者会发现，函数接口和 **code（代码）类型变量**非常相似。的确如此，但是函数接口无论是使用方式还是适用范围都远远超过 code 类型的变量。比如，code 类型的变量只能代表无参数无返回值的函数，要想执行一个 code，必须使用一个触发来实现等。

应用

函数接口的引入，使得脚本的编写具有了更大的灵活性。在教程 8 的最后，我提到还有一种“更优雅”的方案可以代替接口，这就是函数接口。所以，**接口可以做的事情，函数接口也可以做（比如回调等）**，而且**推荐使用函数接口**。函数接口之所以“更优雅”，是因为函数接口类型的变量**不需要销毁**。因为它们总是代表某个函数的，而所代表的函数是不可能销毁的。

在一些系统的编写中，如果能很好地使用函数接口，那么编写出来的系统会具有最大的灵活性和适用性。

委托(delegate)

委托是一种提供方便的语法，首先我们来了解下什么是委托。

假如我写了一个结构，这个结构有 3 个方法：onCreate，onCollide，onDeath：

```
struct SomeStruct

    method onCollide takes unit u returns nothing
        call BJDebugMsg("Collision")
    endmethod

    method onCreate takes nothing returns nothing
        call BJDebugMsg("Create")
    endmethod

    method onDeath takes nothing returns nothing
        call BJDebugMsg("Death")
    endmethod

endstruct
```

现在我要写另外一个结构，“包装”一下这个结构：

```
struct Wrapper

    SomeStruct obj

    method onCollide takes unit u returns nothing
        call obj.onCollide()
    endmethod

    method onCreate takes nothing returns nothing
        call obj.onCreate()
    endmethod

    method onDeath takes nothing returns nothing
        call obj.onDeath()
    endmethod

endstruct
```

我们可以看到，结构 Wrapper 里也有这三个方法，但是 Wrapper 里这三个方法的实现只是简单的使用 obj 成员变量调用了 SomeStruct 的，这就是所谓“**委托**”。就是自己不做，让别人去做而已。

vJass 提供了一种简化的语法来方便实现委托，所以 Wrapper 还可以这么写：

```
struct Wrapper

    delegate SomeStruct obj

endstruct
```

就这么简单。这两种写法的效果是一样的。