vJass 系列教程 9

面向对象编程（四）操作符重载与虚拟属性

Aeris ▸ NJU ▸ 2009/2/3

vJass 系列教程 9

面向对象编程（四）操作符重载与虚拟属性

操作符重载和虚拟属性都属于“虚拟语法”，意思是说，它们可以使得一些原本不适用于这种类型对象的语法成为合法的。

操作符重载

背景

操作符重载有什么用？在教程 8 的最后，我向各位读者展示了 vJass 作者写的一个表格系统例子。我们首先关注一下这个表格系统怎么使用：

```
set t = Table.create()    // 创建一个新的表格对象

set i = t[1234567]        // 获取键1234567对应的值（如果不存在返回0）
set t[12341] = 32         // 给键12341赋值32

call t.flush(12341)       // 清除键12341对应的游戏缓存
t.exists(32)              // 32这个键有没有对应的值？
call t.reset()            // 清除表格的所有内容
call t.destroy()          // 销毁表格对象
                          // 注意销毁表格并不清除表格里的内容，要清除内容，调用reset
```

我们特别注意有灰色背景的那两行。第一行试图获取 t[1234567]的值，第 2 行给索引 12341 赋值为 32。t 是什么？数组吗？我们知道，[]（中括号）是访问数组元素的操作符。可是，再看看前面：

```
set t = Table.create()    // 创建一个新的表格对象
```

原来 t 不是数组，是一个表格对象！为什么表格对象可以使用[]操作符呢？因为在表格对象的定义中**重载了操作符**。表格结构重新实现了“[]”（访问数组元素）和“[]=”（数组元素赋值）操作符，所以，我们可以使用**和访问数组元素一样的方法**来访问表格的元素。

语法和用处

vJass 允许重载的操作符目前有 3 个（0.9.E.0 版）：

- []　数组元素读取
- []=　数组元素赋值
- <　比较操作符（不完善）

因为比较操作符的支持还不完善，故暂不介绍。重点介绍数组元素读取和赋值操作符。

要想使自己写的结构对象用起来“就像一个数组一样”，可以用[]去获取和赋值，就必须在结构里提供数组操作符的定义。也就是提供**操作符重载**。操作符重载的一般语法如下：

```
method operator 操作符 takes 参数 returns 返回值类型
    // 正常的方法体
endmethod
```

可以看到，重载操作符和方法定义很类似，因为**本质上，操作符也是一种函数**。

为什么这么说？请看下表：

正常的 Jass 语法	“函数化”的写法（非 Jass 语法）	等价的函数写法
set i = 1 + 1	set i = +(1, 1)	set i = add(1, 1)
set j = intArray[5]	set j = [](intArray, 5)	set j = get(intArray, 5)
if a > b then	if >(a, b) then	if greater(a, b) then

把第 2 列的＋、[]、>等符号替换成函数名，就是合法的写法了（第 2 行的数组操作只有在 vJass 中针对动态数组才合法）。

好，我们回到重载操作符的语法。前面说了重载操作符的语法和定义方法的语法类似，但是，重载操作符的**参数和返回值必须遵循该操作符的规律**，不能随便改动。比如[]操作符只能有一个参数，写两个参数就是非法，比较操作符必须返回布尔值，返回其他值也是非法。

下表是重载操作符的语法：

```
// 读取数组元素，必须接受1个参数，返回1个值
method operator [] takes 索引类型 参数名 returns 返回值类型
// 数组元素赋值，必须接受2个参数，没有返回值
method operator []= takes 索引类型 参数名, 值类型 参数名 returns nothing
// 比较操作符，必须接受1个参数，必须返回布尔类型
method operator < takes 另一个对象类型 参数名 returns boolean
```

以下是一些合法的定义：

```
method operator [] takes string index returns string
// 以后可以这样调用（假设obj是实现了这个操作符的结构对象）
set str = obj["Hello"]
// 如果这样调用的话，那么index就是字符串"Hello"，返回值赋给str

method operator []= takes string index, integer n returns nothing
// 以后可以这样调用（假设obj是实现了这个操作符的结构对象）
set obj["Hello"] = 5
// 如果这样调用的话，那么index就是字符串"Hello"，n就是整数5
```

完整的例子见教程 8 最后的表格系统。

操作符重载的目的是：**使我们能够以一种更加自然的方式去使用一些结构的对象**（例如表格也是“索引–值”对的形式，使用和访问数组一样的写法就很自然）。除此之外，**它和一般的函数没有任何区别**。

虚拟属性

背景

这是另一个使得结构使用起来“更加自然”的语法，和操作符重载一样，它的功能和一般函数相同。

我们知道，可以用.（点操作符）访问一个结构中的成员变量（属性），例如：

```
set goblin.goldCost = 120
set goblin.lumberCost = 20
```

这里我们设置 goblin 对象的 goldCost 属性为 120，lumberCost 属性为 20。为了使得语义正确，goblin 对象的所属结构（假设是 Goblin）必须定义两个属性：goldCost 和 lumberCost，就像这样：

```
struct Goblin
    integer goldCost
    integer lumberCost

    // 其他定义
endstruct
```

否则会报错：找不到某某成员。

那么，什么是虚拟属性呢？顾名思义，虚拟属性就是本来不存在，“虚拟”的一个属性。也就是说，**我们可以用访问属性的语法，去调用一个函数，“用起来就像是访问属性一样”**。

我们来看一个例子：现在我们需要一个结构来表示时间间隔，时间间隔的单位有时、分、秒（再大或者再小的不考虑），我们也需要计算这段时间间隔大概是多少小时、多少分、多少秒，一个很自然的设计如下：

```
// 时间间隔结构
struct Duration
    // 这段时间间隔的秒数，内部以秒来计算
    private integer secondCount

    // 获取秒数
    method getSec takes nothing returns integer
        return .secondCount
    endmethod

    // 设置秒数
    method setSec takes integer sec returns nothing
        set .secondCount = sec
    endmethod

    // 获取分钟数
```

```
    method getMin takes nothing returns integer
        return .secondCount / 60
    endmethod

    // 获取秒数
    method getHour takes nothing returns integer
        return .secondCount / 3600
    endmethod

endstruct
```

我们可以这样使用：

```
local Duration dur

set dur = Duration.create()

call dur.setSec(86400)

call BJDebugMsg("Sec: " + I2S(dur.getSec()))  // 打印出86400
call BJDebugMsg("Min: " + I2S(dur.getMin()))  // 打印出1440
call BJDebugMsg("Hour: " + I2S(dur.getHour()))    // 打印出24
```

这没什么问题，可是我们也可以换一种写法（注意不同之处，用灰色背景标出）：

```
// 时间间隔结构
struct Duration
    // 这段时间间隔的秒数，内部以秒来计算
    private integer secondCount

    // 获取秒数
    method operator sec takes nothing returns integer
        return .secondCount
    endmethod

    // 设置秒数
    method operator sec= takes integer sec returns nothing
        set .secondCount = sec
    endmethod

    // 获取分钟数
    method operator min takes nothing returns integer
        return .secondCount / 60
    endmethod

    // 获取秒数
    method operator hour takes nothing returns integer
        return .secondCount / 3600
    endmethod

endstruct
```

然后，我们可以这样访问：

```
local Duration dur

set dur = Duration.create()

set dur.sec = 86400

call BJDebugMsg("Sec: " + I2S(dur.sec))    // 打印出86400
call BJDebugMsg("Min: " + I2S(dur.min))    // 打印出1440
call BJDebugMsg("Hour: " + I2S(dur.hour)) // 打印出24
```

Duration 里并没有 sec、min 和 hour 这三个属性，我们却可以对它们进行读写。所以说，这三个属性是“虚拟”出来的。这就是虚拟属性。

语法和用途

虚拟属性有两个性质：**虚拟读和虚拟写**。所谓虚拟读，是说这个虚拟属性只能读而不能赋值，虚拟写则相反，只能给它赋值，却不能读出数据来。当然如果一个虚拟属性同时是虚拟读和虚拟写的，就可以像一般的属性（成员变量）一样自由访问了。定义虚拟属性的语法如下：

- 虚拟读

```
method operator 虚拟属性名 takes nothing returns 返回值类型
```

- 虚拟写

```
// 注意等号
method operator 虚拟属性名= takes 类型 变量名 returns nothing
```

定义虚拟属性如上面的例子一样，就像函数一样定义。**虚拟属性不得和真实属性同名**。

如果一个虚拟属性只定义了虚拟读方法，那么这个虚拟属性就是**只读**的，任何试图写入的操作都会出错，反之如果只定义了虚拟写方法，那么它就是**只写**的，任何试图读取的操作也会出错。如果要使这个属性既可以读也可以写，那么必须同时定义虚拟读方法和虚拟写方法。

虚拟读的例子见 Duration 结构的 min 和 hour 虚拟属性，可读可写的例子见 sec 属性。如果我们试图这样写就是非法的：

```
set dur.min = 1440    // 非法！min 没有定义虚拟写方法，所以不可写
```

而设置 sec 属性就是合法的，因为定义了虚拟写方法。

我们不能给 Duration 结构里添加名为 sec、min 或者 hour 的真实属性了，否则会报错。

从上面的例子可以看出，虚拟属性的功能和函数调用完全相同。如果一个操作使用访问属性的方法比较自然的话，可以考虑用虚拟属性。