vJass 系列教程 4

库（library）和域（scope）

Aeris ▸ NJU ▸ 2008-7-4

vJass 系列教程 4

库（library）和域（scope）

教程简介

从教程 2 和 3 里相信各位读者已经可以领略到 vJass 给地图开发所带来的便利，但那仅仅是 vJass 丰富功能的冰山一角。从这个教程开始，作者将逐步向各位读者介绍 vJass 的真正强大之处，从 vJass 提供的库和域开始。

库（library）

库，顾名思义，是一些实用例程的集合，日常开发中总会用到许多相似的任务，其解决办法大致都相同，把这些相似的程序整理出来，供以后碰到类似问题的时候可以复用（Reuse）。vJass 提供了实现库的语法特征，当然，这种语法特征并不仅仅可以使用在库的开发上。

语法概述

各位读者一定都知道 WE 中有个自定义脚本区（Custom Script Section），WE 的提示也告诉我们：在自定义脚本区写上自己需要的各种代码，然后这部分内容会被插入到全局变量定义区和触发函数区之间，也就是说，自定义脚本区的代码可以设置所有的全局变量，并且所有的触发都可以使用自定义代码区的函数。正是由于这种特性，使得这个区域成为了各种通用函数的安身之地。很多系统在移植的时候，都要求把大量的代码拷贝到自定义脚本区，目的是使这些函数能够在所有的触发函数中使用。但是，这样做带来的一个很明显的问题就是：自定义代码区成了一个大杂烩，里面存放的代码容易变得杂乱无章，很难查找到所需内容，特别是当引入了一个比较庞大的系统时。我们都希望这些系统函数能够像写触发那样，类别清晰，井井有条。

但遗憾的是，我们并不能把库函数放到触发里以期待能有同样的效果。如果我们这么做的话很可能会导致脚本出错。这是因为 Jass 里，变量和函数必须先定义后使用，而 WE 的一个特征就是：无法指定触发代码在最终脚本文件中的顺序。比如说，你定义了触发 a 和触发 b，在最终的 war3map.j 中，到底是 a 的代码在 b 前，还是 b 的代码在 a 前呢，我们是无从得知的（可能只有暴雪自己知道而我们不知道，所以说无从得知）。唯一可以确定的是，自定义脚本肯定在所有触发代码之前。这就导致如果我们把一些多个触发都要用到的库函数放到某个触发里而不是自定义代码区的话，会导致在最终生成的脚本里这些库函数不一定位于所有要用它们的触发之前，从而导致函数未定义错误。

vJass 的库（library）正是用来解决这个问题的。你可以把一段代码放到库里，而库的定义随便放到哪里都行，不需要放到自定义代码区。在编译阶段，这些库里的代码会被编译器自动整理到合适的地方去。另外，有了库，你可以利用库的依赖关系来指定触发代码的放置顺序，从而获得对脚本更大的控制力。

语法说明

库的定义

定义库的语法如下所示：

```
library 库名 [initializer 初始化函数] [requires 依赖库名[,...]]
[库内容]
endlibrary
```

库名：必选项，而且同一张地图中库名必须是唯一的，不能重名。库名可以和函数、变量还有文本宏同名，但不能和域同名。库的定义**不能**嵌套，就是说一个库中不能定义另一个库。

初始化函数：如果需要初始化，可以指定一个不带任何参数且无返回值的函数为初始化函数。初始化函数在**全局变量和触发初始化之前**调用。所以，库的初始化函数中不能调用触发，不能使用变量编辑器里定义的变量初始化值。库的初始化函数必须放在库内部，可以是公开、普通或者私有函数，但一般都用私有函数。关于公开、普通或者私有函数下文会提及。

依赖库名：指明本库依赖于某个/些其他库（使用了其他库的函数），依赖库可以不止一个，将库名之间用逗号隔开即可。这样，在编译的时候，这个库的代码可以保证放在被依赖的库代码之后。另外，**requires** 关键字可以换成 **needs** 或者 **uses**，但是同一地图中最好保持风格的一致。

库内容：库的内容主要是函数实现和全局变量声明。

编译时，库的所有内容都会被放到脚本文件全局变量的下方，所有的触发都可使用。

这是一个简单的例子：

```
library A initializer Init requires B
    private function Init takes nothing returns nothing
        // A初始化
    endfunction
endlibrary

library B initializer Init
    private function Init takes nothing returns nothing
        // B初始化
    endfunction
endlibrary
```

这里定义了库 A 和库 B，库 A 和 B 的初始化函数都是私有函数 Init，库 A 依赖于库 B，因此库 A 的函数可以调用库 B 中所有的普通函数和公共函数，而库 B 不可以调用库 A 的任何函数。我们还可以看到，A 和 B 的初始化函数是同名的，但是它们都是私有函数，作用域仅限于本库内部，因此不会冲突。

访问修饰符

从上面例子我们可以看到，库 A 和 B 的初始化函数前面都加了 private 关键字，这是所谓“访问修饰符”，是 vJass 引入的语法特征，通常和库、结构等配合使用，可以控制函数和变量的作用范围。而传统的 Jass 里，函数和变量从定义起，一直到脚本结尾都可以访问。引入访问控制符，使得传统 Jass 里容易发生函数名称冲突的现象得到一定程度的缓解，也使得一定程度的封装性成为可能。

访问修饰符的语法为：

```
[public] | [private] function 函数名 takes 参数 returns 返回值类型
[public] | [private] 类型 全局变量名
```

可以看到，访问修饰符的语法就是在函数或者全局变量名前面加上 public 或者 private 关键字，加上 public 的为公共符号，private 的为私有符号，什么都不加，和普通 Jass 写法相同的为一般符号。

公共符号：加上 public 修饰符的为公共符号。公共符号属于这个库所有，在库的内部和外部均可访问。在库的外部访问公共符号的方法是：**库名_变量名**，中间用下划线隔开。例如，对于库 Demo 中的公共函数 Function 使用名称 Demo_Function 来访问。而在库的内部，既可以使用和外部同样的方法，也可以直接使用原名来访问。多个库中可以定义同名的公共变量或者函数，不会冲突。实际上，公共符号的名称会被编译器修饰，修饰成“库名_变量名“的样子，正如访问的方法那样。因此，多个库中的同名公共符号不会冲突。

私有符号：加上 private 修饰符的为私有符号。私有符号供库内部使用，外部无法访问，注意私有全局变量在库外部也是无法访问的。当然，多个库可以有同名的私有成员，不会冲突。

一般符号：不加任何修饰符，和普通 Jass 写法相同的为一般符号。一般符号就是普通 Jass 的函数和全局变量，其作用域和公共符号相同，但是名称不会被修饰。所以整个脚本中不能有相同的一般符号。

下面是一个相对复杂的例子，但是它包括了库的大多数语法特征：

```
library A initializer Init requires B
    globals
        public integer money // 公共变量
        private integer internalVar // 私有变量
    endglobals

    private function Init takes nothing returns nothing
        // 初始化
        set money = 0
        set internalVar = 1
    endfunction

    public function SameName takes nothing returns nothing
        // A库中的某公共函数，和B库中的函数同名，不会冲突
    endfunction

    function FunctionA takes nothing returns nothing
        // A库中的一般函数，不能和任何函数同名

        // 依赖于B库，可以访问B库的公共变量life，注意变量名的修饰。不能访问B库的私有成员
        call BJDebugMsg("Life = " + I2S(B_life))
        // 访问本库的公共成员，既可以使用库名_成员名的形式，又可以直接访问
        call BJDebugMsg("Money = " + I2S(A_money))// 正确
        call BJDebugMsg("Money = " + I2S(money))  // 正确，作用同上
        call SameName()                           // 正确
        call A_SameName()                         // 正确，作用同上
    endfunction
endlibrary

library B initializer Init
    globals
```

```
        public integer life      // 公共变量
        private integer internalVar // 私有变量
    endglobals

    private function Init takes nothing returns nothing
        // 初始化
        set intenalVar = 1       // 和A库的变量不冲突
        set life = 1000
    endfunction

    public function SameName takes nothing returns nothing
        // B库中的某公共函数，和A库中的函数同名，不会冲突
    endfunction

    function FunctionB takes nothing returns nothing
        // B库中的其他函数
    endfunction
endlibrary

// 一般触发函数，可以使用所有库中的所有公共和普通成员
function Test takes nothing returns nothing
    //call A_Init()       // 错误！私有函数外部不能访问
    //set A_internalVar = 0 // 错误！不能访问私有变量
    call A_SameName() // 调用库A的SameName函数
    call B_SameName() // 调用库B的SameName函数
    call FunctionA()  // 调用A库中的普通函数
    call FunctionB()  // 调用B库中的普通函数
    call ExecuteFunc("A_SameName")  // 通过ExecuteFunc调用A的SameName函数
    call ExecuteFunc("B_SameName")  // 通过ExecuteFunc调用B的SameName函数

    call BJDebugMsg("Life = " + I2S(B_life))   // 访问公共成员
    call BJDebugMsg("Money = " + I2S(A_money))// 同上
endfunction
```

应用实例——缓存系统

库设计的目的就是抛弃那种把所有公共函数一股脑儿都放到自定义代码区的方法，鼓励我们像写触发那样把各种函数分类放到各个库中。由于库的代码会被编译器自动移到脚本前端，因此库的代码无论放到哪里都可以。平常写的时候，我们可以在触发里写库，然后利用触发可以分类的特点使得代码井井有条。本文用一个很简单的**缓存系统**为例来说明库的实际应用，这个系统综合运用了前面教程所介绍的知识，下面是系统的代码。

- **缓存系统**

缓存系统主要提供了游戏缓存类 API 的封装，使得一些常见操作更加简单。比如，可以用类似于下面的代码把值为 5 的 someAttribute 绑定到某个单位身上：

```
call CacheSystem_AttachInt(someUnit, "SomeAttribute", 5)
```

然后通过下面的代码可以取出存放的值：

```
set someAttribute = CacheSystem_GetAttachedInt(someUnit, "SomeAttribute")
```

这个系统的移植非常简单，把下面的代码拷贝到任意地方都可以用，当然，为清晰起见，最好为它创建一个单独的触发，转换成自定义代码，然后把系统代码放到里面。

这个系统里用到了文本宏（textmacro），但是并不是这个系统最重要的部分。文本宏将在下一章教程中讲述，这个系统中主要是用它生成一系列非常相似的代码，从而减少手工编写的工作量。

```
library CacheSystem initializer Init
    globals
        // 本系统使用的缓存，私有
        private gamecache systemCache
    endglobals

    // 常见的return bug函数，注意它是public的，不会和其他同名函数冲突
    public function H2I takes handle h returns integer
        return h
        return 0
    endfunction

    public function I2U takes integer i returns unit
        return i
        return null
    endfunction

    // 利用缓存把整数绑定到某句柄上
    public function AttachInt takes handle h, string label, integer x
returns nothing
        if x==0 then
            call FlushStoredInteger(systemCache,I2S(H2I(h)),label)
        else
            call StoreInteger(systemCache,I2S(H2I(h)),label,x)
        endif
    endfunction

    // 获取绑定到某句柄上的整数
    public function GetAttachedInt takes handle h, string label returns
integer
        return GetStoredInteger(systemCache, I2S(H2I(h)), label)
    endfunction

    // 利用缓存把实数绑定到某句柄上
    public function AttachReal takes handle h, string label, real x returns
nothing
        if x==0 then
            call FlushStoredReal(systemCache,I2S(H2I(h)),label)
        else
            call StoreReal(systemCache,I2S(H2I(h)),label,x)
        endif
    endfunction
```

```
    // 获取绑定到某句柄上的实数
    public function GetAttachedReal takes handle h, string label returns
real
        return GetStoredReal(systemCache,I2S(H2I(h)),label)
    endfunction

    // 利用缓存把布尔值绑定到某句柄上
    public function AttachBoolean takes handle h, string label, boolean x
returns nothing
        if not x then
            call FlushStoredBoolean(systemCache,I2S(H2I(h)),label)
        else
            call StoreBoolean(systemCache,I2S(H2I(h)),label,x)
        endif
    endfunction

    // 获取绑定到某句柄上的布尔值
    public function GetAttachedBoolean takes handle h, string label returns
boolean
        return GetStoredBoolean(systemCache,I2S(H2I(h)),label)
    endfunction

    // 利用缓存把字符串绑定到某句柄上
    public function AttachString takes handle h, string label, string x
returns nothing
        if ((x=="") or (x==null)) then
            call FlushStoredString(systemCache,I2S(H2I(h)),label)
        else
            call StoreString(systemCache,I2S(H2I(h)),label,x)
        endif
    endfunction

    // 获取绑定到某句柄上的字符串
    public function GetAttachedString takes handle h, string label returns
string
        return GetStoredString(systemCache,I2S(H2I(h)),label)
    endfunction

    // 利用缓存把另一个句柄绑定到某句柄上
    public function AttachObject takes handle h, string label, handle x
returns nothing
        if (x==null) then
            call FlushStoredInteger(systemCache,I2S(H2I(h)),label)
        else
            call StoreInteger(systemCache,I2S(H2I(h)),label,H2I(x))
        endif
    endfunction

    // 获取绑定到某句柄上的另一个句柄，使用文本宏
    //! textmacro GetAttachedHandle takes name, type
        public function GetAttached$name$ takes handle h, string label
returns $type$
```

```
            return GetStoredInteger(systemCache, I2S(H2I(h)), label)
            return null
        endfunction
    //! endtextmacro

    //! runtextmacro GetAttachedHandle("Object"       ,"handle")
    //! runtextmacro GetAttachedHandle("Widget"       ,"widget")
    //! runtextmacro GetAttachedHandle("Rect"         ,"rect")
    //! runtextmacro GetAttachedHandle("Region"       ,"region")
    //! runtextmacro GetAttachedHandle("TimerDialog"  ,"timerdialog")
    //! runtextmacro GetAttachedHandle("Unit"         ,"unit")
    //! runtextmacro GetAttachedHandle("Item"         ,"item")
    //! runtextmacro GetAttachedHandle("Effect"       ,"effect")
    //! runtextmacro GetAttachedHandle("Destructable" ,"destructable")
    //! runtextmacro GetAttachedHandle("Trigger"      ,"trigger")
    //! runtextmacro GetAttachedHandle("Timer"        ,"timer")
    //! runtextmacro GetAttachedHandle("Group"        ,"group")
    //! runtextmacro GetAttachedHandle("TriggerAction","triggeraction")
    //! runtextmacro GetAttachedHandle("Lightning"    ,"lightning")
    //! runtextmacro GetAttachedHandle("Image"        ,"image")
    //! runtextmacro GetAttachedHandle("Ubersplat"    ,"ubersplat")
    //! runtextmacro GetAttachedHandle("Sound"        ,"sound")
    //! runtextmacro GetAttachedHandle("Loc"          ,"location")

    // 清除绑定到句柄上的所有内容
    public function CleanAttachedVars takes handle h returns nothing
        call FlushStoredMission(systemCache,I2S(H2I(h)))
    endfunction

    // 初始化缓存
    private function Init takes nothing returns nothing
        call FlushGameCache(InitGameCache("CacheSystem.w3v"))
        set systemCache = InitGameCache("CacheSystem.w3v")
    endfunction
endlibrary
```

域（scope）

域是一个使用范围没有库广泛的语法特征。因为它仅仅是提供了控制函数和变量作用域的功能，可以看做是一个弱化版的库。但是在某些情况下，例如在复杂地图中控制函数的名称冲突现象，还是很有用处的。一般情况下，有库就足够我们用了。

语法概述

域很像一个库，不论是语法还是行为上，而且可以嵌套定义。但是它的内容会保留在原来的地方而不像库那样会放到脚本的最前端，因此，域不能有依赖关系，因为代码不会移动。

域的主要作用是控制和避免代码复杂时函数（包括变量）名称“撞车”的情况，因为绝大多数触发中定义的函数（比如初始化触发，触发条件，触发动作等函数）只有这个触发用到，其他触发根本不会使用，但是这些函数和变量却占用着名称，使得其他函数不能使用这个名称。比如单位组中常常用到匹配函数，

每次都不得不抓耳挠腮为这些函数起个不冲突的好名字，最后不耐烦了都只好用 Filter1，Filter2……依次编下去。如果把这个触发中的函数放到一个域中，然后设置它们为私有，就可以很好解决这个问题。因为和库类似，不同域的公共和私有函数允许同名。

语法说明

定义域的语法和库非常像，只是把关键字从 library 换成 scope

```
scope 域名 [initializer 初始化函数]
[域内容]
[嵌套域定义]
endscope
```

和库不同的是，域**可以**嵌套，但是**不能**有依赖关系。库和域的区别和联系是，库具有域的所有特征（虽然库中不能嵌套定义库，但可以嵌套定义域）而又有自己的特色，如内容会自动移动到脚本最前端使得任何触发都可以调用库的函数，普通的域不具有这种特征。

域中的函数和库一样可以使用访问修饰符，而且语义相同，这里不再赘述。

示例

这里给出 vJass 手册上的一个简单例子。

```
scope GetUnitDebugStr
    // 脚本中其他任何地方都可以定义H2I，不会引起冲突
    private function H2I takes handle h returns integer
        return h
        return 0
    endfunction

    function GetUnitDebugStr takes unit u returns string
        return GetUnitName(u) + "_" + I2S(H2I(u))
    endfunction
endscope
```

总结

库和域是作者向各位读者介绍的 vJass 的第一个高级特征，相信各位读者已经领略到 vJass 的强大了。

域在控制变量作用域的时候使用，库具有域的特征，但库常用来存放一些实用函数。利用好访问控制符，恼人的变量重名问题不再来。