## OAEScript 设计&实现

**一．目的**

现在各种AVG引擎都是要读取脚本的，而脚本的编写却总是一大问题。所以脚本编写困难成了很多引擎的诟病。但是脚本越贴近机器就越好编程，如果过于贴近人类语言脚本的解读就会产生困难。一开始我觉得预处理比较好，但是后来我又觉得脚本语言，一般都是没有编译和预处理的。就像PHP一样。所以，我决定把解释器全部放在引擎里。我给这个脚本起了一个名字，叫“OAE Script”。而且就像大多数语言一样，界面设计文件和代码是分离的（用过VC的应该知道，VB.NET亦是如此）。我就给这两个脚本各起了一个名字：界面设计的叫“OAE UI Design Script”（OAEUS），控制脚本叫“OAE Ctrl Script”（OAECS）。OAECS决定游戏运行了以后的一切动向。作为引擎，我们要先读取OAECS，然后再根据OAECS中指定的“LoadRes”标签加载OAEUS，再寻找Entry开始执行游戏。另外，我不想再用INI了。INI已经满足不了我们如此庞大的需求。

**二．OAE Script 语法**

首先我要严重地说一下：脚本不能区分大小写。我在调试的时候因为大小写问题整破了头皮。所以，这个脚本中大小写是不被区分的。在执行OAEUS&OAECS的过程中，一律转换成小写。先是代码DEMO：

# *Entry.OAECS*

#\*OAE Ctrl Script V2.0

<LoadRes>Main.OAEUS #加载名为Main的资源脚本

<Include>xxx.OAECS

<Action>Entry{ #一切游戏的入口点都在Entry

Do<Action>LoadScene:<DScene>Main #加载游戏窗口Main

}

<Action>Main\_Item1\_Click{ #Item1被点击时执行，名称随便起

Do<Action>LoadScene:<DScene>Main

}

<Action>Main\_Init{ #Main初始化时执行

Do<Action>SoundPlayLoop:<RSound>BgMusic

Do<Action>LoadItem:<Item>BgImage

}

“#”所跟的内容是注释。第一行是说明这是一个OAECS，适用的引擎版本为2.0。这不仅是一条注释，我们的引擎在读的时候会先读第一行，如果不是这个，或者版本号不对就会报错。这和Bash有点相似。1.0版留给了INI一切游戏的入口点都在Entry，和main()一个含义。Include就是把别的脚本展开进来。LoadScene是一个内部函数，Main可以算是参数吧。可以看到，在所有名称前面，都有一个“<>”标明类别。在这些类别中，只有Action是可执行的。而且为了与定义区分，在执行Action前必须有“Do”。用冒号让整个代码的风格变成了CSS。并且，LoadScene、LoadItem等都是内部函数。另外，可以使用Do<Action>{xxx}。这个样子调用Do会直接执行代码xxx，和JS一个样。至于OAEUS，就和现在我们使用的脚本元素差不多，但是结构有变。比INI 略微复杂。

下面是DEMO：

相比之下，这个OAEUS就略微显得冗长了。Attr是属性的意思。看到了Attr，就直接把冒号右边的内容写到Item的Struct的指定项中。Event是注册事件。Click是Event的一种。R开头的标签<RImage>是图像资源。R取自Resource。

# *Main.OAEUS*

#\*OAE UID Script V2.0

<DScene>Main{ #名为Main的场景的资源，“D”意味Design

<Attr>Width:800

<Attr>Height:600

<Event>Init:Do<Action>Main\_Init #在初始化的时候执行<Action>Main\_Init

}

<DItem>Item1{

<Attr>Type:Image

<Attr>LocX:100

<Attr>LocY:100

<Attr>Height:10

<Attr>Width:20

<Attr>NormalImage:<RImage>Item1\_Normal

<Attr>HoverImage:<RImage>Item1\_Hover

<Attr>ClickImage:<RImage>Item1\_Click

<Event>Click:Do<Action>Exit

}

<DItem>BgImage{

<Attr>Type:Image

<Attr>NormalImage:<RImage>BgImage

}

<RImage>BgImage{

<Attr>Path:"xx.jpg"

}

然而为了应对重复问题，我觉得有必要定义一个类似模板的东西。就像CSS的设计一样。在OAEUS中，一个Item应该可以继承另一个Item的内容。然后在其基础上加以更改。

下面是代码Demo：

# *Snippet in xx.OAEUS*

<DItem>Item1{

<Attr>Type:Image

<Attr>LocX:100

<Attr>LocY:100

<Attr>NormalImage:<RImage>Item1\_Normal

}

<DItem>Item2{

<BaseOn>:<DItem>Item1

<Attr>LocX:150

}

在这里，Item2通过BaseOn基于Item1建立了一个新Item。并且重定义了LocX。

还有关键的一点，就是用户自己的变量定义。面包告诉我别的游戏有什么好感度之类的，可以好感度可以改变游戏的进程和最终结局。所以看来我们还要增加If这类的判断语句。

# *Snippet in xx.OAECS*

<Var>Hello:0

<Action>Item1\_Click{

<Var>Hello:<Var>Hello+1

<If><Var>Hello<GT>2{

Do<Action>LoadScene:Scene1

}

<If><Var>Hello<LT>2{

Do<Action>LoadScene:Scene1

}

<If><Var>Hello<GE>0{

Do<Action>LoadScene:Scene1

}

}

首先把一个名为Hello的变量置0。任何变量不需要声明，解析器如果发现读取未赋值的变量，默认返回0。然后变量的值为它自身的值加一。之后进行三个判断。你学汇编的这些你该见过。GT大于，LT小于，GE大于等于，LE小于等于，EQ等于，NE不等于。我倒是没想做关于作用域的内容。一个变量一经声明全局有效。

最变态的不是这些，面包说还有个功能，就是如果长时间不点一个选项——就跟促销优惠机不可失失不再来一样——就会消失！我勒个去，这简直就是在挑战我的底线。这一点，我决定用特效实现。特效的相关内容，主要有淡出淡入，以及“推过来”（这是面包的原话，怎么不是推倒呢）。不过应该可以把后者概括为“向某个特定方向移动”的特效。

# *Snippet in xx.OAECS*

Do<Action>Ef\_FadeIn:<DItem>Item1,1

Do<Action>Sleep:10

Do<Action>Ef\_FadeOut:<DItem>Item,1

Do<Action>Ef\_Move:20,20,1

大概就是这样子。FadeIn，淡入；FadeOut，淡出；Sleep，延时10秒；Move。顺便，Ef\_FadeIn:<DItem>Item1,1那个最后的1是1秒的意思。意思是淡入一秒。

**三．OAEScript解析的实现**

很自然，我像往常一样不推崇一上来就用string.split。

我觉得我们得逐行读入。先把OAECS读入内存。然后调用一个OAECS读取函数，进行读取。OAECS读取函数会先寻找<Include>标签，如果有，那么就把Include的文件展开。<LoadRes>标签，再把每个<LoadRes>中的资源全部读入，再把Var，Action读入数组。后面附上了很完整的OAECS读取函数工作流程。

初始化函数先调用执行函数，执行“<Action>Entry”。

执行函数先调用读取Action函数，把函数名称传递过去，读取Action函数从Action数组中找到传过来Action名称的对应代码并返回，执行函数再从传来的代码第一行开始执行，逐字读入字符，检查到开头部分是Do就把后面对应的函数名称递归地交给自身来执行。至于参数的传递，再确认了开头是Do后面是Action标签，再后面是Action名称，再后面（而不是“{”）再用“，”分割剩余文本是一个可行的方案。遇到了<Var>Hello:0或<Var>Hello:<Var>Hello+1后，先用冒号分割文本。然后把冒号右边的表达式传给表达式计算函数，在把名称和计算结果传给变量赋值函数。

未完待续

关于整个类的一些信息：

1. Structure
   1. OAEAction，其中包含一个Action的名称和代码。
   2. OAEVar，包含每个脚本中定义的变量的名字和值。
2. 变量
   1. Action数组 OAEAction类型
   2. Var数组 OAEVar类型

各个函数的一些信息：

1. OAECS读取函数
   1. 参数：通过引用传递的当前操作文件流。
   2. 局部变量：文件流栈（数组）
   3. 基本动作：
      1. 开始逐行读入
      2. 读到Include标签，就把原来的文件流压入文件流栈，把Include的脚本文件读入当前操作文件流。
      3. 读到LoadRes标签，有的话交给OAEUS读取函数处理。
      4. 读到Var标签，有的话交给Action执行函数处理。（把该行代码传给Action执行函数）
      5. 读到Action标签，把Action的名称和花括号内的代码计入*Action数组。*
      6. 不停循环
2. Action执行函数
   1. 参数：Action名称，Action代码，Action参数
3. 读取Action函数
   1. 参数：Action名称
   2. 基本动作：先去*内部函数数组*中寻找指定Action，没找到就上*Action数组*里找。找到了就返回，没找到抛异常。
4. 变量读取函数
   1. 参数：变量 String （<Var>Hello）
   2. 基本动作：这有什么难的？
5. 表达式计算函数
   1. 参数：表达式 String
   2. 局部变量：值 数组 Integer、状态 String、是否为布尔表达式 Boolean、布尔表达式运算符 String
   3. 基本动作：首先检查*表达式*是否含有<Var>xx，有的话就把<Var>xx替换成指定的值。然后Trim后从头开始读，如果第一个是数字，就把数字记录在*值数组*中。遇到加减乘除后，把符号记录在*状态*变量中。接着读。如果检查*状态*变量，如果状态变量不为空，那么如果表达式的这一位不是数字就返回错误，如果上一位是除法运算这一位是0那么就Math Error，然后把*值*数组中的第一位取出来，与这一位进行*状态*变量中指定的计算，再存入值数组。然后如果一切执行完，返回。但是如果中途出现<，那么把状态置为“<”，*是否为布尔表达式*置为True，再读，如果*状态*为“<”且这一位就不是“>”就*布尔表达式运算符=布尔表达式运算符+这一位……*然后把布尔运算符右边的表达式的值存在*值*数组的第二位，读完后在进行入*布尔表达式运算符*中所要求的布尔运算。再返回。
6. 变量赋值函数
   1. 参数：变量 String 、 值 Integer

**四．结语**

这些东西谢某你一个人做多少有点让我不太放心，你也说过觉得我想得很奇怪，所以这篇东西写了，就是为了规范今后的脚本，省的出乱子。