

AOE2-XS 函数&规则篇

Part1 函数

在帝国时代 2：决定版中，XS 脚本对于场景功能的定制有着重要的作用。在本部分的 XS 教学内容中，我们来学习函数。

在数学上，函数是输入值 x 和输出值 y 的对应关系，可以表示为 $y = F(x)$, F 代表输入 x 和输出 y 的运算规则。在计算机程序里，函数是一段能实现特定功能的代码块。计算机程序里，函数的一般定义形式如下：

返回值类型 函数名称 (参数列表)
{ 函数体语句 }

AOE2-XS 是一种类 C/C++ 的语言，其函数定义规范大体遵循 C/C++ 的定义规范。下面举一个例子说明 XS 中的函数定义。

例：实现一个函数 `add()`，求 2 个实数相加的结果

```
float add (float x = 0.0, float y = 0.0)
{
    float result = x + y ;
    return (result) ;
}
```

函数说明：

float：返回值类型。在 XS 中函数的返回值类型有：`void`, `bool`, `int`, `float`, `string`, `vector`

add：函数名称，说明该函数的作用，由用户定义。（内置函数的名称由系统定义）自定义函数名称的命名规范应做到见名知意。

参数列表：函数参数列表是由小括号 () 包含的一系列参数，这些参数代表函数的输入。其定义规范和变量的定义规范一样，这里不再赘述。

【注意】：XS 函数的形参列表只支持**默认参数**，不支持**位置参数**。

以下是在 C++ 中，函数参数列表的 2 种定义形式：

```
void func(int a, int b) // 位置参数
{ 函数体 }
```

```
void func(int a = 0, int b = 0) // 默认参数 (XS 采用这种定义形式)
{ 函数体 }
```

简单的说，**默认参数即参数在定义时就给它赋初值。**

函数体：函数体语句，是由{ }包含起来的部分，是函数功能实现的核心代码块。在本例子中，函数体是：

```
{
    float result = x + y ;
    return (result) ;
}
```

return：函数返回值语句，返回函数的输出结果。此处的**返回值要用小括号()包含起来**。函数返回值的类型：除了 void 类型不需要 **return** 语句，其他类型的返回值，都需要 **return** 返回对应类型的数据。在本例中，`return (result);` 返回一个 float 类型的计算结果。

函数的调用：

函数在定义完成之后，需要调用它才能发挥效果。函数调用的一般形式为：

函数名(参数列表值)；

在本例中，如果我们想要计算求 $5.0 + (-3.0)$ 的和，可以这样调用函数：

```
add (5.0 , -3.0); // 输出结果：2
```

XS 函数的分类

介绍完函数定义之后，我们来看看**函数的分类**：

1. 按照是否有输入参数划分，可分为**无参函数**和**有参函数**：

无参函数：`void func() {函数体}`

有参函数：`void func(int a=0, float b=3.14) {函数体}`

2. 按照函数功能划分，可分为**内置函数**和**用户定义函数**。内置函数由系统定义，可直接调用；自定义函数由用户自己定义，遵循先定义后使用的原则。

XS 中的内置函数

在 XS 中有很多内置函数，它们都有其特定的功能，提供给用户使用。其中比较常用的有：

```
xsChartData( )  
xsEffectAmount( )  
xsGetRandomNumberLH( )  
xsGetObjectCount( )  
... ..
```

更多关于内置函数的内容，请参考 [UGC 文档](#) 和 [cly 的 XS 专栏](#) 这里不再展开讲解！ [书山有路勤为径，帝国无涯勇行舟]

用户定义函数(User Defined Function, UDF)

很多时候，仅凭内置函数并不能很好的完成场景功能的设计，这时就需要自己编写函数实现这些功能，自定义函数 UDF 就派上用场了。

顾名思义，UDF 函数的名称有用户自己定义，遵循标识符的命名规则，这里不再赘述。

主函数 main()

接下来，我们来看一个比较特殊的函数 `main()`。在一些编程语言里，主函数 `main()` 作为程序的入口，在执行阶段最先被调用。

在 AOE2-XS 里，若定义了 `main()` 函数，它会在 场景/战役 运行时自动执行 1 次。`main` 函数可以用来做一些初始化设置。

Main 函数的定义：

```
void main()  
{ 函数体 }
```

例：在场景/战役运行时打印欢迎信息 “Welcome to play AOE2”

```
void main() {xsChartData("Welcome to play AOE2");}
```

XS 函数扩展内容

1. 给函数起别名

某些函数的名称太长，写起来比较繁琐，且也不利于场景脚本的内置（内置 XS 脚本有 256 字符限制）。因此，需要根据情况给某些内置函数起别名以简化书写。

给函数起别名的模板如下：

```
返回值类型 别名函数 (形参列表)  
{ 原函数 (形参列表); }
```

注意，别名函数的形参列表，和原函数完全一样。此处通过调用别名函数，间接的调用了原函数。

下面举例例进行说明。

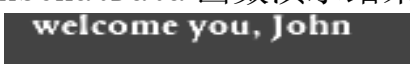
(1). 给 xsChatData() 函数起别名：

我们通过执行以下指令，在聊天窗口输出信息。

```
string player = "John";    // 玩家名称  
xsChatData("welcome you, " + player);    // 输出对话信息
```

xsChatData() 函数，通过将字符串与其他数据类型的拼接操作（“+” 拼接），生成对话信息，然后将信息输出到屏幕。关于 XS 中的数据类型转换规则，请参考 UGC 相关文档。

xsChatData 函数演示结果



如果我想给 xsChatData 函数起个简洁的别名叫 printf（C 语言中的标准输出函数），要怎么做呢？

定义函数 printf，在函数体中调用内置函数 xsChatData，并将 UDF 相关参数传递给 xsChatData() 即可。

以下是完整实现：

```
void printf(string desc="", float val_=-32768, int color_id=0)  
{  
    /* 参数说明：  
    desc: 文本描述信息  
    val_: 要在文本中显示的变量值信息  
    color_id: 颜色 id 编号，对应关系如下：
```

```

{ 1: <BLUE>蓝色; 2: <RED>红色; 3: <GREEN>绿色;
  4: <YELLOW>黄色; 5: <AQUA>青色; 6: <PURPLE>紫色;
  7: <GREY>灰色; 8: <ORANGE>橙色 }

*/
string color = "";
switch(color_id)
{ // 设置字体颜色
  case 1: color = "<BLUE>";
  case 2: color = "<RED>";
  case 3: color = "<GREEN>";
  case 4: color = "<YELLOW>";
  case 5: color = "<AQUA>";
  case 6: color = "<PURPLE>";
  case 7: color = "<GREY>";
  case 8: color = "<ORANGE>";
  default: color = "<WHITE>"; // 默认值: 白色
}
if ((desc != "") && (val_ != -32768)) {xsChatData(color+desc+val_);}
else if ((desc != "") && (val_ == -32768)) {xsChatData(color+desc);}
else if ((desc == "") && (val_ != -32768)) {xsChatData(color+val_);}
else if ((desc == "") && (val_ == -32768)) {xsChatData(color+"");}
else {}
}

```

给函数起完别名之后，我们想再次打印上面的欢迎信息，且调整字体颜色为黄色<YELLOW>，将代码进行修改如下：

```

string player = "John"; // 玩家名称
printf("welcome you, " + player, -32768, 4); // 输出对话信息，欢迎玩家 John (以黄色字体显示)

```

说明：

printf 函数有 3 个输入参数：desc(string)，val_(float)，color_id(文本颜色标号[1 ~ 8]，int)。分别用于接收输入的文本描述，变量数值，和文本颜色 ID 的信息。

这里我们不需要第 2 个参数 `val_`，但是此处必须传入其默认值（不传会产生错误，这是 XS 相比 C 类语言比较 Lower 的地方，吐槽！）

printf 函数输出对话信息：

```
welcome you, John
```

(2) . 给 `xsEffectAmount` 函数起别名 【内置 XS 中的运用推荐】

我们知道，触发效果的“XS 脚本调用”输入框，最多只能输入 256 个字符（包括换行和空格）。当我们需要修改的效果条目较多时，字符限制就成为瓶颈。

案例：

修改属性函数 `xsEffectAmount()` 名称较长，当修改条目较多时，可能超出 256 字符，这时就需要对它起别名。（这里将函数的别名起为 `effect`）

定义 `effect` 函数 代码如下：

```
void effect(int mod=0, int unit=1, int attr=0, float val=0.0, int p=-1)
{ xsEffectAmount(mod, unit, attr, val, p); }
```

参数说明：

mod：模式参数；

unit：单位 ID

attr：属性 ID

val：属性值

p：玩家编号

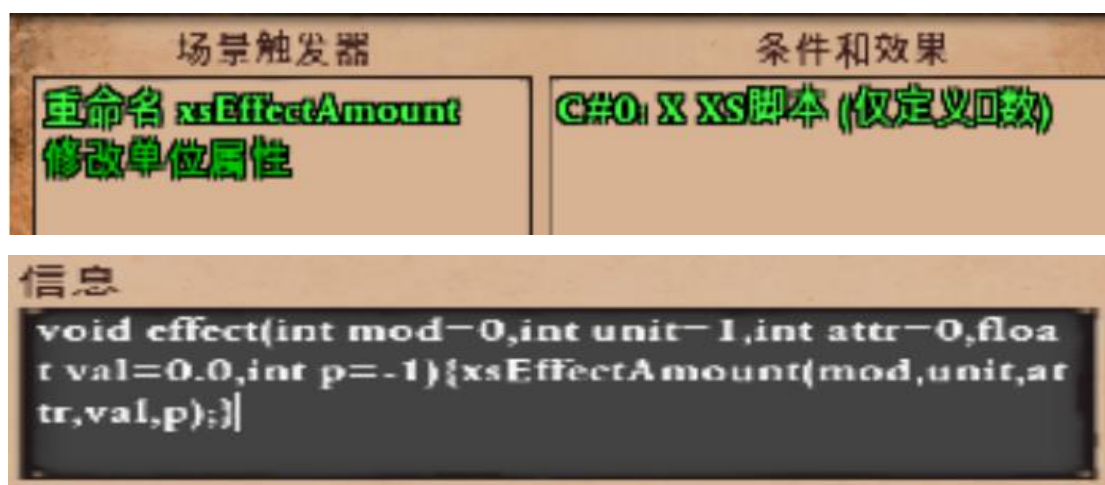
调用 `effect()` 函数

由于 `effect` 是别名函数，在场景内置脚本调用时，应当将它包含在其他函数体内部来调用。而 `main()` 函数在定义之后，会自动执行，这里将 `effect` 放在 `main()` 里进行调用，代码如下：

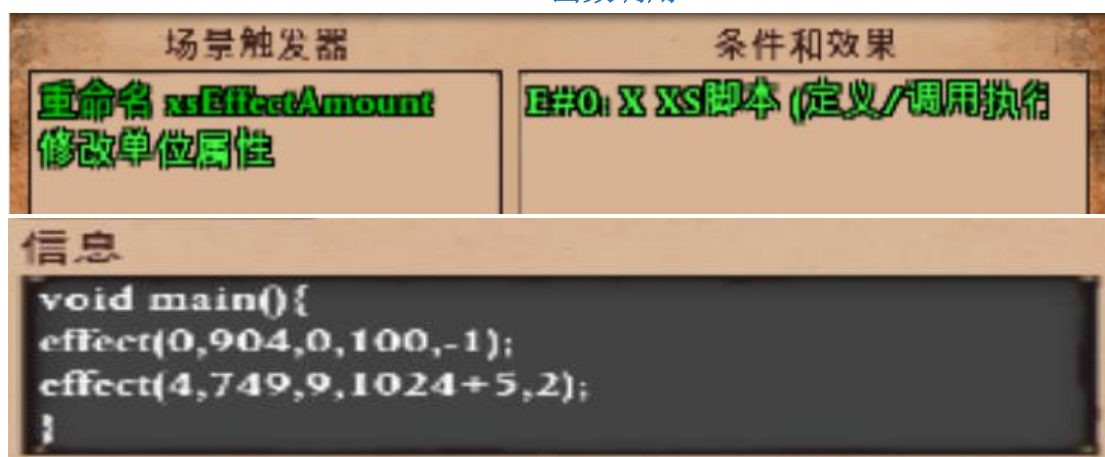
```
void main()
{ // 所有玩家 904 类（村民）生命值设置为 100
  effect(0, 904, 0, 100, -1);
  // 玩家 2 的 749 单位攻击力+5
  effect(4, 749, 9, 4*256+5, 2);
}
```

将以上代码写入场景文件的“条件：XS 脚本”和“效果 XS 脚本”：

effect 函数定义



effect 函数调用



经过以上设置，就成功的将所有玩家“村民”类以及玩家 2 的 ID=749 的单位相关属性进行了更改。

2. XS 函数的重写 (mutable)

函数重写，也叫函数覆盖。在 C++ 等语言中，重写是子类的函数与其继承的父类函数具有相同的函数名，参数列表，返回值类型从而实现子类函数的函数体覆盖父类函数，达到功能重写的效果。

在 AOE2-XS 中，由于 BUG 的原因，导致某些功能无法正常使用。要想让它们生效，对其进行重写是一种解决方案。

例如给变量赋初值的一些内置函数，在 XS 中不生效。下面举例说明：

XS 基本数据类型及其默认值对应关系：

```
{"int": -1, "float": 0.0, "bool": true,
"string": "", "vector": (0,0,0)}
```

由上面的默认值对应关系可知，布尔类型（bool）的默认值为 true，与我们的常规逻辑不合（我们通常认为数据类型的默认值是 0 值或者空值或假值），显然，我们需要将 bool 类型的默认值设置为 false，这就要用到函数重写。

XS 中函数重写的基本语法：

```
mutable 返回值类型 函数名(形参列表)
{ 函数体语句 }
```

说明：

1. 使用关键字 **mutable** 修饰的函数，可在后续对其进行函数体重写（这个函数也叫 基函数）；
2. 对函数进行重写时，要保证函数的 **返回值类型**，**函数名**，**参数列表** 都和基函数完全一样，区别在于二者函数体的不同；
3. 函数体语句，在重写时被用户重新定义，以覆盖基函数的函数体，实现不一样的效果。这也是函数重写的意义所在。

案例 1：重写 XS 中基本数据类型默认值的函数。

// 重写 Bool 函数

```
mutable
bool Bool(float val_=0.0) {
    if (val_ != 0) {return (true);}
    return (false);
}
```

调用： **bool** vb = **Bool**(); // 给 vb 变量赋初值 false

// 重写 Int 函数

```
mutable
int Int(float val= 0.0) { return (1*val_);}
```

调用： **int** vi = **Int**(); // 给 vi 变量赋初值 0

案例 2：现在我们要实现一个求和函数 `sum`，求 2 个实数相加的结果并返回。

函数实现如下：

```
mutable // 只有当该函数未来需要被重写时，才加关键字 mutable
float sum(float x=0.0, float y=0.0)
{
    float res = x+y;
    return (res);
}
sum(3.14, 2.75); // 调用函数，实现 3.14 和 2.75 相加；返回结果：5.89
```

【需求变更】：现在我想让 2 个实数相加后，只返回结果的整数部分。如果仍然用 `sum` 实现，该怎样做？

这里就需要对之前实现的 `sum` 进行重写，在之前的函数名称前用 `mutable` 关键字修饰，然后重写函数如下：

```
float sum(float x=0.0, float y=0.0)
{
    float res = 1*(x+y); // 相加后乘以 1，舍去小数部分
    return (res);
}
sum(3.14, 2.75); // 调用函数，实现 3.14 和 2.75 相加；返回结果：5.0
```

XS 函数重写的注意事项：

1. 函数重写的前提条件：

- (1) 函数名称相同
- (2) 函数参数列表相同
- (3) 返回值类型相同

2. 当一个 功能/机制 有多种实现方式时，根据运用场景的不同，采用不同方式实现，可以使用函数重写。

3. 函数重写一般只适用于 UDF 函数，不适用于 XS 提供的大多数内置函数（经测试，对某些内置函数重写，会造成游戏崩溃 ！）

XS 函数使用的经典误区

关于 UDF 函数的形参列表

通过之前的学习我们知道，函数分为无参函数和有参函数。关于有参函数的形参列表，与某些内置函数结合使用时，存在一些坑，需要特别注意。

1. UDF 函数的形式参数被屏蔽

案例：现在要给玩家 1 和玩家 2 的村民属性做如下设置：

- a. 工作效率 * 2
- b. 生命值设置 60
- c. 近战护甲+2（修改后为 3 甲）

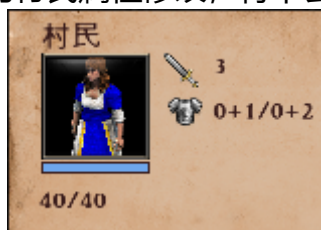
实现函数 villager_attrs 修改村民属性

方式一：

UDF 以函数的形式参数指定玩家序号（**错误的方式**）

```
void villager_attrs(int p1=1, int p2=2)
{
    xsEffectAmount(5, 904, 13, 2, p1);
    xsEffectAmount(0, 904, 0, 60, p1);
    xsEffectAmount(4, 904, 8, 4*256+2, p1);
    xsEffectAmount(5, 904, 13, 2, p2);
    xsEffectAmount(0, 904, 0, 60, p2);
    xsEffectAmount(4, 904, 8, 4*256+2, p2);
}
```

【效果预览】：用方式一做的村民属性修改，将不会生效！



原因在于**某些内置函数如 xsEffectAmount 屏蔽了 UDF 的形式参数**，使得参数 p1 和 p2 不能正确的传入 xsEffectAmount 里面。

方式二：在函数内定义局部变量，指定玩家序号（**正确的方式**）

```
void villager_attrs( )  
{  
    int p1=1 ; int p2=2 ;  
    xsEffectAmount(5, 904, 13, 2, p1);  
    xsEffectAmount(0, 904, 0, 60, p1);  
    xsEffectAmount(4, 904, 8, 4*256+2, p1);  
    xsEffectAmount(5, 904, 13, 2, p2);  
    xsEffectAmount(0, 904, 0, 60, p2);  
    xsEffectAmount(4, 904, 8, 4*256+2, p2);  
}
```

【效果预览】：用方式二做的属性设置，就能正确的生效：



以上是 2 种不同设置方式造成的差异，请特别注意 ！

你也许会问，为什么不直接在 xsEffectAmount() 函数里第 5 个参数写上 1, 2 代表玩家呢？这样当然可以。但如果设置的属性数量比较多，未来想变更玩家的设置时，需要将所有的 xsEffectAmount() 玩家参数都更改，相当麻烦。设置局部变量存储玩家序号等属性的好处是，更改玩家时只要改这个局部变量的值即可，做到一改全改。

Part2 特殊的函数：rule

rule 即规则，在帝国 2 决定版中，XS 脚本支持使用 rule 来执行某些周期循环的机制（如[日历机制](#)，[天气变化机制](#)，[单位属性的动态更新](#)，[资源投资复利机制](#)），这些机制都有其周期运行的特点，有的机制运行周期 T 恒定，有的周期 T 则是不断变化的。

Rule 的语法规则：

```
rule rule_name    // 规则名称
    active / inactive    // 规则初始状态: {active: 激活, inactive: 未激活}
    group group_name    // 规则组名称
    minInterval <int>    // 规则最小初始执行周期, 单位: 秒
    maxInterval <int>    // 规则最大初始执行周期, 单位: 秒
    runImmediately    // 若使用此参数, 则规则激活后第一次立即运行,
                        // 无需等待 minInterval/maxInterval 的时间间隔
    highFrequency    // 高频参数, 每物理秒循环规则 60 次 (与游戏速度
                        // 无关); 【注意】: 在 rule 中, 只能使用 highFrequency 或 minInterval /
                        // maxInterval 中的一个参数, 二者不能同时使用
    priority <int>    // 规则运行的优先级。取值范围在 [0, 255] 之间, 数
                        // 字越大优先级越高
    {
        规则体代码块 ...
        // xsDisableSelf();
    }
```

rule 的本质是一个函数，其特点是自带初始状态，执行间隔 (minInterval/maxInterval) 以及优先级 (priority, 取值范围 0~255)，它是一种特殊的函数。可以使用内置函数 xsGetFunctionID("rule_name") 获得其函数 ID 编号。

关于 rule 的定义，有以下几点注意事项：

- (1). 定义规则时，rule_name 和 group_name 不能同名；
- (2). 通过 active/inactive 参数指定规则的初始状态，该参数不能省略；
- (3). minInterval / maxInterval 是规则执行周期，在首次赋值时必须赋以一个整数常量值，不能赋予变量；若不指定，则默认值是 1。可通过

`xsSetRuleMinInterval()` 或 `xsSetRuleMaxInterval()` 函数修改 rule 的运行周期;

(4). 若干个 rule 被同时开启, 且初始运行周期相等时, priority 数值大的 rule 会优先被执行, 数值小的则靠后执行;

(5). rule 在执行过程中, 当满足一定条件需要关闭规则循环时, 可以通过在规则体中执行 `xsDisableSelf()` 语句, 或者在规则外通过 `xsDisableRule("rule_name")` 来禁用规则循环。

(6). 主函数 `main()` 的运行优先级高于激活 rule 的最大优先级 priority

255

Rule 综合运用案例

这里以资源投资复利机制为例, 进行讲解。

机制简介:

通过玩家的 gold 储备进行投资, 投资时扣除一定数量的 gold (gold 超出 300 的部分进行投资), 并分 10 期返还本金和利息, 各期的本金和利息单独结算; 根据各期所得利润, 按照一定比例设置“食物”, “木材”, “石头” 3 项贸易工厂资源的生成率, 作为额外回报。

该投资机制具有以下特点:

1. 波动利率(8%--12%)
2. 风险投资 (各期返还本金有 29%概率产生亏损)
3. 可多次循环投资, 使用 vector 数组存储每次投资的详细, 并周期更新待还本金和利息。当所有期数的本息返结算成后, 该次投资完成。

由于文档篇幅限制，该案例不在此处详细展开说明，案例详情请参考附件：

[invest_system.aoe2scenario](#)

[invest_system.xls](#)

如有疑问或者建议，请与我取得联系。您的反馈意见至关重要！

作者：babycat

联系方式：QQ:2855241645 微信号：babycat262