StarCraft II

Editor

Galaxy 教程

——疯人ℓ哀人

出自 Goblin Academy -- 地精研究院

http://bbs.islga.org/

目录

零．序言

[quote]

“教程之前要吐槽几句，这貌似是个惯例。”

“如果哪里你看不懂，请先：内事不明问谷哥，外事不明问度娘，啥也不知道问 GA。”

“因为我不是计算机专业的，很多定义与名词可能不符合编程习惯，请大家谅解。”

“教程中没有提到，但是 C 语言语法支持的功能大部分都不被 Galaxy 支持，我也许会有疏漏的地方，欢迎各位 SEer 指出，我将会一一添加到教程之中。”

“引用的部分，可能不是 Galaxy 语法，这点请大家注意。”

“Word 版教程中，引用部分使用[quote]和[/quote]标注，代码使用[codes=galaxy]和[/codes]标注。”

“这是一篇 Galaxy 教程，需要一部分基础知识，如果你什么也不知道的话，请看玻璃渣的官方教程，大体了解 SC2 和 SE 后再看此篇教程。”

“教程的语法说明部分，采用引用格式，加粗字体为相关说明，非加粗字体部分为固定格式。”

“如果你真的看不懂，可以参看一些 C 语言编程书籍作为参考。”

“感谢 AMO 与 ccctyll 对此教程的帮助。特别感谢 AMO 绘制封面，尽管还是使用自己用 Word 艺术字做的最土的一幅”

[/quote]

一. Galaxy 概述

Galaxy 是什么？谷哥的答案是银河，度娘的答案也是银河，Galaxy 的翻译确实是银河，除此之外还貌似是某款手机的名称。不过这些解释这跟我们没有什么关系。因为 GA 娘给出了正确答案，Galaxy 是玻璃渣（Blizzard）出品的游戏 SC2（StarCraftⅡ——星际争霸 2）的一种类似 C 的脚本语言。

《星际争霸 2》官方第 52 批 FAQ 中对 Galaxy 的描述如下：

[quote]
Q:星际 2 的地图编辑器还是使用魔兽争霸 3 的 JASS 程式语言吗，还是一种新版本的语言？
A:星际 2 的地图编辑器使用一种全新的脚本语言，我们把它叫做 Galaxy——银河。这是一种很接近于 C 语言的语言，任何熟悉 C 语言的人对 Galaxy 都可以很快上手。

Q:它是事件驱动的还是面向对象的？
A:虽然多数本地函数是基于对游戏对象操作的，但 Galaxy 语言本身不是面向对象的。
[/quote]

这些古老的信息大约只能用于提升考古学经验，对于我们来说 Galaxy 是我们在制作 SC2 地图中使用的一种脚本语言，我们使用它来描述如何实现我们需要的效果。你完全可以近似的认为 Galaxy 是一种面向对象的编程语言——一种需要 SC2 客户端支持的编程语言。

在制作 SC2 地图的过程中，Galaxy 起着不可替代的作用，尽管你有可能使用的是 GUI 界面下的触发编写，但是其实质上还是 Galaxy。Galaxy 可以说是 SE 中功能最强大的部分，它涉及到整个地图制作的方方面面，Galaxy 编写的地图脚本将地图的其他方面，如 Actor、单位数据等串联起来，才构成一张完整的地图。

不过，对比 War3（Warcraft III——魔兽争霸 3）的地图，SC2 的地图对于脚本支持的需求略小一些。很多内容已经不是必须用脚本来实现，使用 SE（StarCraft II Editor 的简称）的数据编辑器即可，不可否认，SE 的数据编辑器相当强大。当然这不是说你可完全脱离 Galaxy 来制作地图。当你为了实现需要的效果绞尽脑汁仍然毫无办法时，尝试使用 Galaxy 吧。

1.1 Galaxy 与 GUI 界面

GUI 是什么？

[quote]
图形用户界面（Graphical User Interface，简称 GUI，又称图形用户接口）是指采用图形方式显示的计算机操作用户界面。与早期计算机使用的命令行界面相比，图形界面对于

用户来说在视觉上更易于接受。

——度娘

[/quote]

注意：本教程中的 GUI 特指使用数据编辑器的 GUI 界面编写地图脚本。

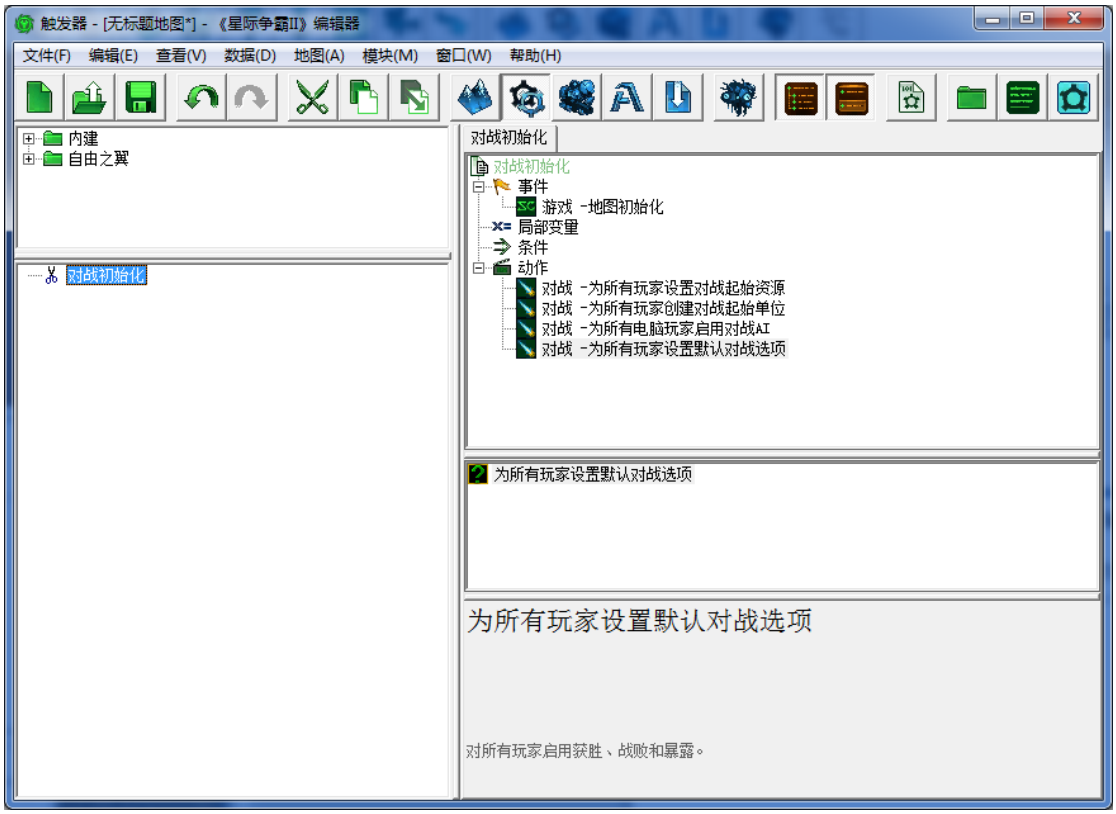


图 1-1 简体中文触发编辑器（GUI）界面

GUI 与 Galaxy 是互为表里关系，GUI 的本质不过是由触发编辑器自动生成 Galaxy 脚本而已。

让我们看一下一张新建地图的脚本：

（方法为在选择任意非库的触发、动作后，按 Ctrl +F11 键。）

代码 1-1 一张新建地图的原始 Galaxy 脚本（由英文语言的编辑器创建）

[codes=galaxy]

```
//==========================================================================
==============================
//
// Generated Map Script
//
// Name:   Just Another StarCraft II Map
// Author: Unknown Author
//
//==========================================================================
```

```
==============================
include "TriggerLibs/NativeLib"

//---------------------------------------------------------------------------
------------------------------
// Library Initialization
//---------------------------------------------------------------------------
------------------------------
void InitLibs () {
    libNtve_InitLib();
}

//---------------------------------------------------------------------------
------------------------------
// Trigger Variables
//---------------------------------------------------------------------------
------------------------------
trigger gt_MeleeInitialization;

//---------------------------------------------------------------------------
------------------------------
// Trigger: Melee Initialization
//---------------------------------------------------------------------------
------------------------------
bool gt_MeleeInitialization_Func (bool testConds, bool runActions) {
    // Actions
    if (!runActions) {
        return true;
    }

    MeleeInitResources();
    MeleeInitUnits();
    MeleeInitAI();
    MeleeInitOptions();
    return true;
}

//---------------------------------------------------------------------------
------------------------------
void gt_MeleeInitialization_Init () {
    gt_MeleeInitialization =
TriggerCreate("gt_MeleeInitialization_Func");
    TriggerAddEventMapInit(gt_MeleeInitialization);
}
```

```
//---------------------------------------------------------------------
------------------------------
// Trigger Initialization
//---------------------------------------------------------------------
------------------------------
void InitTriggers () {
    gt_MeleeInitialization_Init();
}

//---------------------------------------------------------------------
------------------------------
// Map Initialization
//---------------------------------------------------------------------
------------------------------
void InitMap () {
    InitLibs();
    InitTriggers();
}
```
[/codes]

代码 1-1 的内容就是图 1-1 显示的 GUI 界面下触发编辑中的内容。也就是我们存储在地图文件根目录下的 MapScript.galaxy 文件的内容。

从某些方面来讲，对比 War3 的地图编辑器 WE（World Editor），SE 在触发编辑器的功能方面有很多强化的地方。以至于我们完全可以只使用 GUI 界面来制作地图脚本，如果你有这样的打算，我还是建议你学习一部分 Galaxy，因为 GUI 中也有直接写 Galaxy 的地方（一般称为自定义脚本）。

使用 GUI 和直接写 Galaxy 脚本的差异在于：

使用 GUI 相对较为安全，出现语法问题的可能性较低；缺点是编写速度较慢，大部分情况下执行效率较直接写 Galaxy 略低。

使用 Galaxy 编写地图脚本比较快捷，执行效率高。缺点是很容易出现语法问题，Debug 难度较大，需要一部分英语水平。

具体选择哪种编写脚本方式，请自行选择。你也完全可以混合使用。需要明确说的一点，不同于 WE 下 J（Jass）和 T（Trigger，触发）在制作地图中实现能力上的差别，使用 GUI 完全可以实现 Galaxy 所能实现的内容。

1.2 编写 Galaxy 的工具

这里给出两种选择：

1、一般来说，完全可以直接使用 SE 来编写 Galaxy。

2、也可以使用 Galaxy++ Editor 来编写 Galaxy 脚本。

Galaxy++ Editor 支持类 C 的自定义语法、函数名提示、语法检查及其他功能，并有可见即可得的对话框设计工具。

官方说明地址（英文）：

http://www.sc2mapster.com/forums/resources/third-party-tools/19619-galaxy-editor/

http://www.sc2mapster.com/assets/galaxy-editor-beier/

下载地址：

ftp://46.163.69.112/Releases/

注：Galaxy++ Editor++ 需.net4.0 完整版本支持，使用课件及可得的对话框设计工具还需要 Microsoft XNA Framework Redistributable 3.1 的支持。

Microsoft .NET Framework 4（独立安装程序）下载地址：

http://www.microsoft.com/downloads/zh-cn/details.aspx?FamilyID=0A391ABD-25C1-4FC0-919F-B21F31AB88B7

Microsoft XNA Framework Redistributable 3.1 下载地址：

http://www.microsoft.com/download/en/details.aspx?id=15163

1.3 SE 的了解程度

Galaxy 是面向对象的脚本语言，它不能脱离 SE 的其他方面单独存在，更多的情况下，你在编写 Galaxy 的同时，还需要处理其他的一些内容，如通过数据编辑器设置单位、设置 Actor 等等。因此学 Galaxy 需要有一定的 SE 基础。

那么，我们就来看看你有怎样的水平。下面从各个方面描述了 SE 的基础知识，请实际的判断自己是否了解。

① SE 基础应用：

(1) 打开关闭 SE。

(2) 保存打开地图。

(3) SE 升级。

(4) 添加并切换语言包。

② 地图地形

(1) 修改地面纹理、地形高度。

(2) 水体的设置与添加。

(3) 添加单位与装饰物。

(4) 设置地图属性，包括地图大小，玩家属性等。

(5) 放置并使用点、区域、路径、镜头。

③ 数据编辑器

(1) 了解对战、战役中各个单位、技能的基本数据。

(2) 修改单位技能的数值属性。（伤害、血量）

(3) 完全新建一个单位。

(4) 新建或修改武器、技能、行为、效果。

(5) 能够设置足印、按钮等数据内容。

⑹ 了解并掌握武器、技能、行为、效果，能够熟练的构建彼此间联系以实现需要的效果。
⑺ 了解 Actor（动作者）的基本设置。
⑻ 简单的 Actor 事件处理。
⑼ 处理单位动作者。
⑽ 使用 Actor 部位运算。
⑾ 使用修改其他类别的 Actor。

④ 触发编辑器

⑴ 了解基本的事件、条件、动作。
⑵ 了解各种变量类型、及使用数组变量，知道如何进行变量类型间的转换。
⑶ 会使用 If 条件语句。
⑷ 会使用 while 循环语句。
⑸ 能够添加并控制单个、多个单位。
⑹ 处理镜头。
⑺ 对话框相关内容。
⑻ Datable（数据表）的存入读取。
⑼ Bank（数据集文件的使用）。
⑽ 能够在 GUI 下独立编写程序。
⑾ 了解 Galaxy（自定义脚本）的使用。
⑿ 自定义事件、条件、动作、函数、预设类型。

本人水平有限，不能面面俱到的写出关于 SE 的各个方面内容。但以上条目基本上列举了 SE 的各方面知识、应用。

我无法明确的告诉大家掌握了其中哪几条才能学习 Galaxy，我只能说，你对这些内容掌握的越多，学习 Galaxy 的难度越小。最好是先学会使用 GUI 界面的触发编辑器使用，了解变量、触发等名称的含义，然后在学习 Galaxy。当然，如果你有较高的编程水平，就当这句话我没说。

官方基础教程链接：http://www.battlenet.com.cn/sc2/zh/game/maps-and-mods/

二．变量

变量应该说是一个存储内容的箱子，我们可以把我们需要存储的内容放进去与取出来。你也可以将变量理解为一个数学表达式的未知数，如：X+1 中的 X。

那么我们为什么要使用变量呢？

变量实质是编程集成化的最基本体现。变量的作用是描述一个编写时还没有确定的值在执行或运算过程中出现的位置。这样我们就不需要为每种可能的值单独编写一段代码了。这点与数学中的未知量近视。

2.1、Galaxy 的变量类型

算法的处理对象就是数据，而数据有其特殊的存在方式。数据依据内容的不同区分成多种变量类型。变量类型其实就是变量能够存储的值的种类。Galaxy 中的变量类型如下：

表 2-1 Galaxy 中的变量类型。

序号	变量类型	GUI 界面下变量声明及初值	Galaxy 下变量声明
1	AI 筛选器	Variable 1 = (为 1 创建 AI 筛选器)〈AI 筛选器〉	aifilter gv_variable1;
2	标记	Variable 2 = 无标记〈标记〉	marker gv_variable2;
3	波次	Variable 3 = 无波次〈波次〉	wave gv_variable3;
4	波次目标	Variable 4 = 无波次目标〈波次目标〉	wavetarget gv_variable4;
5	波次信息	Variable 5 = 无波次信息〈波次信息〉	waveinfo gv_variable5;
6	布尔	Variable 6 = 假〈布尔〉	bool gv_variable6;
7	触发器	Variable 7 = 无触发器〈触发器〉	trigger gv_variable7;
8	传输信息	Variable 8 = 无传输信息〈传输信息〉	int gv_variable8;
9	传输信息源	Variable 9 = (无人)〈传输信息源〉	transmissionsource gv_variable9;
10	单位	Variable 10 = 无单位〈单位〉	unit gv_variable10;
11	单位类型	Variable 11 = 无游戏链接〈游戏连接 - 单位〉	string gv_variable11;
12	单位筛选器	Variable 13 = 无单位筛选器〈单位筛选器〉	unitfilter gv_variable13;
13	单位组	Variable 14 = (空的单位组)〈单位组〉	unitgroup gv_variable14;
14	点	Variable 15 = 无点〈点〉	point gv_variable15;
15	电影动画	Variable 16 = 无电影片段〈电影动画〉	int gv_variable16;
16	动画名称	Variable 17 = 站立〈动画名称〉	string gv_variable17;
17	动作者	Variable 18 = 无动作者〈动作者〉	actor gv_variable18;
18	动作者消息	Variable 19 = "SetShowing 1"〈动作者消息〉	string gv_variable19;

19	动作者作用域	Variable 20 = 无动作者作用域〈动作者作用域〉	actorscope gv_variable20;
20	对话	Variable 21 = 无对话〈对话〉	int gv_variable21;
21	对话回复	Variable 22 = 无对话回复〈对话回复〉	int gv_variable22;
22	对话框	Variable 23 = 无对话框〈对话框〉	int gv_variable23;
23	对话款项	Variable 24 = 无对话框项〈对话框项〉	int gv_variable24;
24	对话状态索引	Variable 25 = 无对话状态〈对话状态索引〉	string gv_variable25;
25	计分板	Variable 26 = 无计分板〈计分板〉	int gv_variable26;
26	计时器	Variable 27 = (新建计时器)〈计时器〉	timer gv_variable27;
27	计时器窗口	Variable 28 = 无计时器窗口〈计时器窗口〉	int gv_variable28;
28	技能命令	Variable 29 = 无技能命令〈技能命令〉	abilcmd gv_variable29;
29	交易类别	Variable 30 = 无交易类别〈交易类别〉	int gv_variable30;
30	交易项	Variable 31 = 无交易项〈交易项〉	int gv_variable31;
31	交易组	Variable 32 = 无交易组〈交易组〉	int gv_variable32;
32	镜头对象	Variable 33 = 无镜头对象〈镜头对象〉	camerainfo gv_variable33;
33	模型镜头	Variable 34= 无模型镜头〈模型镜头〉	string gv_variable34;
34	目标筛选	Variable 35 = 不包括: 发射物, 死亡的, 隐藏的〈目标筛选〉	unitfilter gv_variable35;
35	难度等级	Variable 36 = 普通〈难度等级〉	int gv_variable36;
36	区域	Variable 37 = 无区域〈区域〉	region gv_variable37;
37	任务档案	Variable 38 = 无任务档案〈任务档案〉	int gv_variable38;
38	任务目标	Variable 39 = 无任务目标〈任务目标〉	int gv_variable39;
39	时间段	Variable 40 = 00:00:00〈时间段〉	string gv_variable40;
40	实数	Variable 41 = 0.0〈实数〉	fixed gv_variable41;
41	属性 ID(玩家)	Variable 42 = 无属性 ID〈属性 ID(玩家)〉	string gv_variable42;
42	属性 ID(游戏)	Variable 43 = 无属性 ID〈属性 ID(游戏)〉	string gv_variable43;
43	属性集	Variable 44= 无属性值〈属性值〉	string gv_variable44;
44	数据集文件	Variable 45 = 无数据集文件〈数据集文件〉	bank gv_variable45;
45	水体	Variable 46 = 无水体〈水体〉	string gv_variable46;
46	顺序	Variable 47 = 无指令〈顺序〉	order gv_variable47;
47	头像	Variable 48 = 无头像〈头像〉	int gv_variable48;
48	玩家颜色	Variable 49= (玩家 01) 红色〈玩家颜色〉	int gv_variable49;
49	玩家组	Variable 50 = (空的玩家组)〈玩家组〉	playergroup gv_variable50;
50	微缩地图显示	Variable 51 = 没有微缩地图提示〈微缩地图提示〉	int gv_variable51;
51	文本	Variable 52 = 无文本〈文本〉	text gv_variable52;

52	文本标签	Variable 53 = 无文本标签〈文本标签〉	int gv_variable53;
53	显示器	Variable 54 = 无显示器〈显示器〉	revealer gv_variable54;
54	行星	Variable 55 = 无星球〈行星〉	int gv_variable55;
55	研究层级	Variable 56 = 无研究层级〈研究层级〉	int gv_variable56;
56	研究类别	Variable 57 = 无研究类别〈研究类别〉	int gv_variable57;
57	研究项	Variable 58 = 无研究项〈研究项〉	int gv_variable58;
58	颜色	Variable 59 = 黑色〈颜色〉	color gv_variable59;
59	音效	Variable 60 = 无音效〈音效〉	sound gv_variable60;
60	音效链接	Variable 61 = Editor Default Sound〈音效链接〉	soundlink gv_variable61;
61	佣兵	Variable 62 = 任意佣兵〈佣兵〉	int gv_variable62;
62	整数	Variable 63= 0〈整数〉	int gv_variable63;
63	装饰物	Variable 64 = 无装饰物〈装饰物〉	doodad gv_variable64;
64	字符串	Variable 65 = ""〈字符串〉	string gv_variable65;
65	字节	Variable 66 = 0〈字节〉	byte gv_variable65;

表 2-1 内容排序依据 SE 中的变量类型名称顺序排序。

大家仔细阅读的话，会发现有几种变量类型使用的是相同的变量类型，如传输信息、计时器窗口、行星、整数等都是采用了 int 变量类型；属性 ID、水体、字符串等都采用了 string 变量类型。具体原因与存储方式有关，这里不做说明。

变量类型有两类，一类是正常存储数据的变量类型，我称之为基础变量类型；一类是多个数据组合的复合类型的变量类型，我称之为复合变量类型。

关于这点大家可以先看 Native.galaxy 中的注释。

代码 2-1 Native.galaxy 中有关变量类型的注释

[codes=galaxy]

```
//--------------------------------------------------------------------
--------------------------------
// About Types
//--------------------------------------------------------------------
--------------------------------
//
// -- Complex types and automatic deletion --
//
// Many native types represent "complex" objects (i.e. larger than 4 bytes).
The script language
// automatically keeps track of these objects and deletes them from memory
when they are no longer
// used (that is, when nothing in the script references them any longer).
The types which benefit
// from automatic deletion are:
```

```
//
//     abilcmd, bank, camerainfo, marker, order, playergroup, point,
//     region, soundlink, string, text, timer, transmissionsource,
unitfilter, unitgroup, unitref,
//     waveinfo, wavetarget
//
// Other object types must be explicitly destroyed with the appropriate
native function when you
// are done using them.
//
//
// -- Complex types and equality --
//
// Normally, comparing two "complex" objects with the == or != operators
will only compare the
// object reference, not the contained object data.  However, a few types
will compare the contained
// data instead.  These types are:
//
//     abilcmd, point, string, unitfilter, unitref
//
// Examples:
//
//     Point(1, 2) == Point(1, 2)                          // True
//     "test string" == "test string"                     // True (note:
this is case sensitive)
//     AbilityCommand("move", 0) == AbilityCommand("move", 0)  // True
//     Order(abilCmd) == Order(abilCmd)                    // False
(two different order instances)
//     RegionEmpty() == RegionEmpty()                      // False
(two different region instances)
//
//
// -- Complex types and +/- operators --
//
// Besides numerical types (byte, int, fixed), a few complex types support
+ and/or - operators:
//
//     string, text    + operator will concatenate the strings or text
//     point          +/- operators will add or subtract the x and y
components of the points
```
[/codes]

对此注释的翻译（来源：http://bbs.islga.org/read-htm-tid-40285.html，由头目翻

译)：

[quote]
```
//--------------------------------------------------------------------------------
// 关于类型
//--------------------------------------------------------------------------------
//
// -- 复合类型与自动删除 --
//
// 许多原生类型代表“复合”对象(通常大于 4 字节)。银河脚本语言会自动追踪这些对象，当他们不再被使用时就将他们从
// 内存中移除(“不再被使用”的意思是没有任何指向他们的脚本引用)。受益于自动删除的类型有：
//
// abilcmd, bank, camerainfo, marker, order, playergroup, point,
// region, soundlink, string, text, timer, transmissionsource, unitfilter, unitgroup, unitref,
// waveinfo, wavetarget
//
// 而如果你想干掉其余对象类型则需要使用与之对应的显式销毁函数来进行销毁。
//
//
// -- 复合类型与等值比较 --
//
// 通常的，两个“复合”类型之间进行==操作或!=操作只会比较两者的引用，而非其引用的对象的值。然而，少数对象类型却会直接对其引用的对象的值进行比较。这些类型是：
//
// abilcmd, point, string, unitfilter, unitref
//
// 举例：
//
// Point(1, 2) == Point(1, 2) // 真
// "test string" == "test string" // 真 (注意：大小写敏感)
// AbilityCommand("move", 0) == AbilityCommand("move", 0) // 真
// Order(abilCmd) == Order(abilCmd) // 假 (两个不同指令实例)
// RegionEmpty() == RegionEmpty() // 假 (两个不同区域实例)
//
//
// -- 复合类型与 +/- 操作 --
//
// 除了数值类型(byte, int, fixed)以外，少数复合类型也支持+或-操作：
//
```

```
// string, text + 操作用于合并 string 或 text
// point +/- 操作将会对点的 x,y 坐标进行增减
[/quote]
```

通过等值比较的说明，我们可以看到。复合变量直接存储的内容并非是存储的值。而是这些值的引用。

如图 2-1 所示，如果不考虑从引用到存储数据之间的关系，复合变量可以看做存储内容为引用的基础变量，从内存上来说，复合变量的表也是与基础变量一样的。当调用复合变量时，多运行一步依照引用值获取实际存储内容的步骤。

这些复合变量类型与基础变量类型在使用中的差别。将在讲解相关内容时说明。

基础变量类型包括：byte，bool，int，fixed，string。其他基本都是复合变量类型。（此论断仅是个人臆断，没有对所有变量逐一测试。）

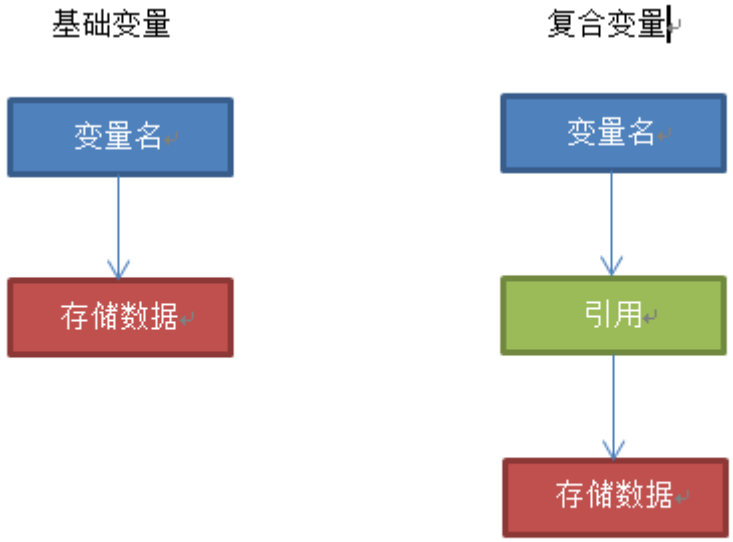


图 2-1 基础变量和复合变量

图 2-1 存储关系为个人猜测，我的水平无法做实际验证，仅作为理解基础变量和复合变量差异的辅助。

2.2 几个常用的变量类型

① bool 是布尔值的变量类型。
具体的值为“true”（真）与“false”（假）。bool 变量主要用于 if 语句。

② int 是整数的变量类型。
值范围为[-2147483648, 2147483648]，计算超出范围会导致值的异常，例如 2147483648 + 1 的计算结果显示为 -2147483648。

输入值超出范围在编译时会有错误提示。int 变量用处较多，主要常用于循环计数和数组变量的序号。
除了支持 10 进制，Galaxy 的整数变量还支持八进制与十六进制：
以 0 开头的整数是八进制整数，如 011 代表十进制的 9。
以 0x 开头的整数是十六进制整数，如 0x11 代表十进制的 17。
int 变量可以做 bool 使用，0 为“false”其他整数为“true”。

③ fixed 是浮点变量类型，简单来说就是带小数的数值——实数。
值范围为[-2147483648, 2147483648]，输入超出范围在编译时会有错误提示。
fixed 变量主要用于实体数据（如：单位）的详细数据值。

④ string 是字符串变量类型。
string 变量主要用于路径名、类别名和 datatable 的识别符。
string 变量的内容需要用”“括起来（注意是英文引号）。如”abc”
string 不支持中文及中文标点符号，但是采用其他方式可以实现中文效果。
之前提到的 Galaxy++ Editor 可以将中文转换成对应的 Unicode 内码，将需要转换的内容赋值给一个 string 变量即可，然后输出文件中查找对应的语句即可。

⑤ text 是文本变量类型。text 主要用于名称等显示的文本数据。
关于文本输出的相关帖子。
http://bbs.islga.org/read-htm-tid-39031.html
http://bbs.islga.org/read-htm-tid-39032.html
http://bbs.islga.org/read-htm-tid-54289.html
http://bbs.islga.org/read-htm-tid-76992.html

另外要特殊说明的是数组。

数组是什么？

数组是有序数据的集合。数组的每一个数据都属于同一个变量类型。用统一的一个数组名和若干下标类唯一的确定数组中的元素。

你可以将数组理解为批量处理的变量。在 Galaxy 脚本编写的过程中，数组的使用频率非常高。关于数组声明等内容在 2.4 中讲解。

2.3 变量的声明

变量在使用前需要声明。所谓变量声明就是向内存申请存储空间的过程，并用变量名标记这个存储空间。我们通过使用这个变量名来获取其存储的值。

变量声明其语法结构如下：

[quote]

变量类型 变量名;

[/quote]

变量声明的同时进行赋初值：

[quote]

变量类型 变量名 = 初始值;

[/quote]

如：

代码 2-2 变量声明举例

[codes=galaxy]

```
int a;
string b = "Yes";
```

[/codes]

变量类型参考 2.1、2.2 节中所述，变量名需要注意以下几点：

变量名只能使用字母、数字及下划线“_”。

变量名必须以字母开始。不能使用数字或者下划线“_”。

变量名不能重复。包括不同变量类型的变量。

变量名区分大小写，如 a 与 A 是两个变量。

变量不能使用关键词，如 int 等。

Galaxy 不支持多个函数同时声明，如“int a,b;”这种。

注意， GUI 下声明无此要求，原因是在 GUI 下声明的变量会自动转换成符合以上内容的变量名。如名为“无标题的变量 001”的变量会被自动转换转换成名为“gv_e697A0E6A087E9A298E79A84E58F98E9878F001”的变量。

Galaxy 下变量请在初始赋值后使用,GUI 界面下的变量声明一般会自动赋初始值。

代码 2-3 GUI 界面声明全局变量的 Galaxy 脚本

[codes=galaxy]

```
//--------------------------------------------------------------------------------
// Global Variables
//--------------------------------------------------------------------------------
string gv_s;

void InitGlobals () {
gv_s = "";
```

[/codes]

代码 2-4 GUI 界面声明局域变量的 Galaxy 脚本
[codes=galaxy]
```
// Variable Declarations
string lv_s;

// Variable Initialization
v_s = "";
}
```
[/codes]

在变量声明时还可以加入一个特殊的前置标识符——“const”。其意义是表达这里声明的是一个常量。

常量，顾名思义，就是保持不变的变量。常量多用于固定参数。如圆周率等。

常量的声明方式如下：

[quote]
const **变量类型 变量名** = **初始值**;
[/quote]

代码 2-5 常量声明举例
[codes=galaxy]
```
    const int x = 3;
```
[/codes]

2.4 数组的声明

一维数组的声明方式：

[quote]
变量类型［数组大小］变量名;
[/quote]

如:
代码 2-6 数组声明与赋值举例
[codes=galaxy]
```
    int [3] I;
    I[0] = 2;
    I[1] = 1;
    I[2] = 3;
```
[/codes]

数组的声明，就不是单一的申请存储一个数据的空间，而是一系列数据的空间。代码 2-6 中实际存储的值如表 2-2 所示。 申请的是三个连续的空间。

表 2-2 数组的存储结构

元素	I[0]	I[1]	I[2]
序号	0	1	2
值	2	1	3

声明数组基本上与声明正常变量类似，需要额外注意的共有两点：1、数组不能动态创建。每一个数组的长度在脚本编写时就必须确定。可以使用整数常量作为数组长度。数组序号从 0 开始，代码 2-6 中声明的数组，其可用的数组元素为 I[0],I[1],I[2],一共三个，如果你使用 I[3]，你会得到一个错误提示：（台服客户端）

[quote]
嘗試進入一個超出陣列的元素。
[/quote]

另外声明一个数组变量是无法赋初始值的。

多维数组的声明与一维数组近似：

[quote]
变量类型 [数组大小 1]……[数组大小 N] 变量名;
[/quote]

“[数组大小 1]……[数组大小 n]”是数组维度，说明此数组有 N 个存储维度。如声明一个二维整数数组 I，维度大小分别为 2、3，那么声明语句如下：

代码 2-7 多维数组声明举例：
[codes=galaxy]
```
int [2][3] I;
```
[/codes]

表 2-3 二维数组的存储结构

	行序号	I[][0]	I[][1]	I[][2]
列序号		0	1	2
I[0][]	0			
I[1][]	1			

如表 2-3 所示，二维数组 I 共有 2×3=6 个元素。依次为 I[0][0]、I[0][1]、I[0][2]、I[1][0]、I[1][1]、I[1][2]。三维及以上维度的的数组以此类推。

2.5 结构体

在 Galaxy 中，除了以上提到的变量类型，还可以讲多种不同类型的数据组合成一个新的整体，以便引用。这样的数据结构在 C 语言中被称为结构体。

结构体变量类型声明方式：

[quote]

struct **结构体变量类型名**

{

结构体成员 1;

结构体成员 2;

……

};

[/quote]

代码 2-8 结构体类型的声明实例：

```
[codes=galaxy]
struct variable
{
    int a;
    string [3] b;
    string [3] c;
};
[/codes]
```

代码 2-8 就是一个简单的结构体类型声明。注意这里声明的是结构体类型，而不是一个结构体本身。如果你需要使用结构体，你还需要如下语句定义结构体变量。

代码 2-9 结构体定义的实例：

```
[codes=galaxy]
    variable a;
    variable [2] b;
[/codes]
```

关于声明结构体变量类型，有如下需注意内容：

1、 结构体变量类型中的成员可以有若干个。这些成员可以是普通变量、数组、甚至是使用另外一个结构体变量类型声明的结构体。

2、 如果你声明的结构体变量类型的成员中有结构体。那么这个成员结构体的变量类型需要在当前声明的结构体变量类型声明语句之前声明。具体可参看代码 2-8、代码 2-9。如果顺序错误，会无法编译。

3、 结构体变量名的要求与普通变量相同。

4、 注意，不同于 if 结构、while 结构或自定义函数中使用的{}。定义结构体变量类型时必须要在}后面加上一个“;”。否则编译错误。

5、 结构体变量类型名不能与其成员名相同。否则编译错误。

6、 结构体只能在全局范围内声明，不能再函数范围内声明。

7、 结构体成员不能有初值。

8、 结构体变量类型声明时，不能同时定义结构体变量。

代码 2-10 正确的声明方式。

[codes=galaxy]
```
    struct variable_1
    {
        int a;
    };

    struct variable_2
    {
        int a;
        variable_1 b;
    };
```
[/codes]

代码 2-11 错误的声明方式。

[codes=galaxy]
```
    struct variable_2
    {
        int a;
        variable_1 b;
    };

    struct variable_1
    {
        int a;
    };
```
[/codes]

结构体变量的使用，其实跟正常的变量一样。比如我们使用代码 2-9 中的结构体变量类型定义一个结构体变量 x。若我们需要对其成员结构体 b 的成员 a 赋值 1，我们可以使用这样的语句。

[quote]
```
    x.b.a = 1;
```
[/quote]

使用结构体变量需注意以下几点。

1、 结构体不能作为自定义函数的参数类型或返回类型。

2、 如果结构体变量或其结构体成员、普通变量成员是数组，那么需要对应添加其数组序号。如：a[1].b[2].c[3]。

从某些意义上将，结构体的实用性不高。除非多层结构体（即结构体变量类型的成员中包含结构体），否则结构体的更大的作用是增加代码的可读性。

2.6 赋值、表达式和类型转换

赋值就是用变量保存对应类型值的过程。赋值符号为“=”。如“a = 1;”含义是将数值 1 赋值给变量 a。注意赋值过程是把等号右边的值赋给左边的变量。所以赋值的语法格式如下：

[quote]
变量 = **需要赋的值**;
[/quote]

注意赋值语句与声明语句一样需要用“;”作为结束标记。

“需要赋的值”可以是一个表达式。但等号左右的变量类型需相同或兼容。具体兼容的变量类型将在后面变量类型转换中说明。

关于表达式，请先看度娘说明：

[quote]
表达式

引表达式，是由数字、算符、数字分组符号(括号)、自由变量和约束变量等以能求得数值的有意义排列方法所得的组合。约束变量在表达式中已被指定数值，而自由变量则可以在表达式之外另行指定数值。

给与自由变量一些数值指定，可能可以给与一个表达式数值，即使对于一些自由变量的值，表示式或许没有定义。因此，一个表达式代表一个函数，其输入为自由变量的定值，而其输出则为表示式因之后所产生出的数值。

举例来说，表达式 *x* / *y*，分别使自由变量 *x* 和 *y* 定值为 10 和 5，其输出为数字 2；但在 *y* 值为 0 时则没有定义。

一个表达式的赋值和算符的定义以及数值的定义域是有关联的。

两个表达式若被说是等值的，表示对于自由变量任意的定值，两个表达式都会有相同的输出，即它们代表同一个函数。

一个表达式必须是合式的。亦即，其每个算符都必须有正确的输入数量，在正确的地方。如表达式 *2+3* 便是合式的；而表达式**2+*则不是合式的，至少不是算术的一般标记方式。

表达式和其赋值曾在一九三〇年代由阿隆佐・邱奇和 Stephen Kleene 在其 Λ 演算中被公式化。Λ 演算对现代数学和电脑编程语言的发展都曾有过重大的影响。

——度娘
[/quote]

Galaxy 中的表达式也是由操作数和运算符构成。

Galaxy 中支持的运算符包括：

[quote]

1.算术运算符：* - + / %

2.关系运算符： > < == != >= <=

3.逻辑运算符：！ && ||

4.位运算符：<< >> ~ | ^ &

5.赋值运算符：=及扩展赋值运算符

6.结构体运算符：.

7.下标运算符：[ ]

8.其他：如函数调用运算符:()

[/quote]

运算符的优先级和结合性如下：

[quote]

优先级【高到低】:

第一级：

圆括号【()】、下标运算符【[]】、结构体成员运算符【.】

第二级：

逻辑非运算符【!】、按位取反运算符【~】、负号运算符【-】

第三级：乘法运算符【*】、除法运算符【/】、取余运算符【%】

第四级：加法运算符【+】、减法运算符【-】

第五级：左移动运算符【<<】、右移动运算符【>>】

第六级：关系运算符【< > <= >= 】

第七级：等于运算符【==】、不等于运算符【!=】

第八级：按位与运算符【&】

第九级：按位异或运算符【^】

第十级：按位或运算符【|】

第十一级：逻辑与运算符【&&】

第十二级：逻辑或运算符【||】

第十三级：赋值运算符【= += -= *= /= %= >>= <<= &= |= ^=】

说明：

①G1 不要求运算对象的个数，G2 是单目运算符,，其他都是双目运算符。

②G2 条件运算符、G14 赋值运算符是自右向左的【也就是右结合性】，其他都是自左向右【左结合性】

归纳各类运算符【高到底】:

初等运算符【()、[]】 G1

单目运算符 G2

算术运算符(先乘除【取余】，后加减) G3,4

位运算符【<< >>】 G5

关系运算符 G6,7

位运算符【递减& ^ |】 G7,8,9

逻辑运算符(不包括！) G11,12

赋值运算符 G13

[/quote]

下面我们就来详细说明变量类型的兼容和转换。

能够兼容的变量类型只有 int 和 fixed 两种。确切的说，一个 int 类型的变量可以作为 fixed 类型的变量来处理。反之不行。同样也只有 int 变量能够与 fixed 在一个运算符两边，运算结果也是 fixed 类型的。

代码 2-12 变量类型兼容性

[codes=galaxy]

```
int a;
fixed b;
b = a + b;
a = a + b; ,
```

[/codes]

其中“b = a + b;”正确，“a = a + b;”错误。

除兼容性的自动转换，变量类型可以做强制转换，有如下几种：

代码 2-13 变量类型转换函数

[codes=galaxy]

```
//------------------------------------------------------------------------
-------------------------------
// Conversions
//------------------------------------------------------------------------
-------------------------------
native int     BoolToInt (bool f);

native fixed    IntToFixed (int x);
native string   IntToString (int x);
native text     IntToText (int x);

native int     FixedToInt (fixed x);
native string   FixedToString (fixed x, int precision);
native text     FixedToText (fixed x, int precision);
native text     FixedToTextAdvanced (fixed inNumber, int inStyle, bool
inGroup, int inMinDigits, int inMaxDigits);

native int     StringToInt (string x);
```

```
native fixed    StringToFixed (string x);
```
[/codes]

以上为Native.galaxy中类型转换函数声明。可以参考GUI界面对于的转换动作来使用。

三．简单的代码

3.1 基本语句

Galaxy 中一共有四种基本语句。分别为声明语句、赋值语句、执行语句、控制语句。
声明语句，顾名思义，就是声明变量、函数、引用的基本语句。
赋值语句，之前提到过，基本结构为“**变量 = 可以作为值的内容**;”。
执行语句，一般为函数调用。
控制语句，包括“if”选择结构和“while”循环结构的语句。
除此四种外，还有单独成行的“{”或“}”、空白行、注释。

在这些基本语句中，只有函数的定义语句（直接写函数体的），选择结构“if”或者循环结构“while”的“{”“}”，结构体声明的“{”，include 语句，空白行、以及注释不需要用英文的“;”作为结尾，其他全部需要。否则会在编译时报错。

3.2 注释

[quote]
注释就是对代码的解释和说明。目的是为了让别人和自己很容易看懂。为了让别人一看就知道这段代码是做什么用的。

——度娘
[/quote]

Galaxy 中只有行注释，并且 SE 的 Galaxy 注释不支持中文。
注释的语法格式为

[quote]
//**注释内容……**
[/quote]

注释符号“//”的有效范围只有当前行第一个注释符号“//”之后的全部内容。多行注释需要每行开始都添加“//”符号。

3.3 顺序结构

关于 Galaxy 下的代码，共有三种结构：顺序结构、选择结构、循环结构。下面我们就来依次介绍这三种结构的使用。

三种结构中，最为基本、最为常用的是顺序结构。所谓顺序结构，就是按照代码顺序依次执行的结构。

关于顺序结构，貌似没有什么好说的。根据想好的内容，依次描述即可。

3.4 选择结构

选择结构有两种，语法结构为：

[quote]

```
if (逻辑值或逻辑表达式)
{
    语句 1;
}
```

[/quote]

[quote]

```
if (逻辑值或逻辑表达式)
{
    语句 1;
}
else
{
    语句 2;
}
```

[/quote]

其含义为，如逻辑值或逻辑表达式的值为 true，那么执行语句 1，如果有 else 之后的部分，那么如果逻辑值或逻辑表达式的值为 false，那么执行语句 2。

多次选择可以使用：

[quote]

```
if (逻辑值或逻辑表达式)
{
    语句 1;
}
else if (逻辑值或逻辑表达式)
{
    语句 2;
}
……
else if (逻辑值或逻辑表达式)
{
    语句 n;
}

else
{
    语句 n+1;
```

```
    }
```
[/quote]

其中可以使用若干个“else if()”注意其中间需要有个空格。另外，如果使用“else”，则必须为最后一个，否则语法错误。

if 语句可以嵌套使用，详见代码 3-1。

代码 3-1 if 语句实例

[codes=galaxy]
```
if (X > Y)
{
    if (X > Z)
    {
        OutPut = X;
    }
    else
{
    OutPut = Z;
}
}
else
{
    if (Y > Z)
    {
        OutPut = Y;
    }
    else
    {
        OutPut = Z;
    }
}
```
[/codes]

代码 3-1 是一个 3 个变量 X、Y、Z 比较大小的代码。结构很简单，依次比较而已。先比较 X 和 Y，再用其中最大的一个与 Z 比较。如此获得最大的值。

在这里要重点说下逻辑表达式。

与逻辑表达式有关的运算符有两种，分别为关系运算符和逻辑运算符。

关系运算符是二目运算符，参与运算的两个操作数需要为同变量类型的变量，计算结果为布尔值（bool）。请参看 2.6 节中的表达式优先级等内容。关于复合类型与等值比较请参看 2.1 节中变量类型注释。

逻辑运算符不同于关系运算符。逻辑运算符中！为单目运算符，&&与||为二目运算符。

逻辑运算符真值表如下：

表 3-1 逻辑非运算符（Y = ！X）真值表

状态序号	输入值 X	输出值 Y
1	true	false
2	false	true

表 3-2 逻辑与运算符（Z = X && Y）真值表

状态序号	输入值 X	输入值 Y	输出值 Z
1	true	true	true
2	true	false	false
3	false	true	false
4	false	false	false

表 3-3 逻辑或运算符（Z = X || Y）真值表

状态序号	输入值 X	输入值 Y	输出值 Z
1	true	true	true
2	true	false	true
3	false	true	true
4	false	false	false

关于逻辑运算的详细内容，请自己询问度娘或者谷哥。

注意:逻辑运算采用短路模式，即当整个表达式计算到某一段时，已经确定结果，就不会进行额外的计算，这种现象出现在与（&&）和或（||）运算中。

如“false && (X > Y && X > Z)”与“true || (X > Y && X > Z)”中后面的“(X > Y && X > Z)”并不会实际参与运算。

测试代码如下：

代码 3-2 逻辑运算短路测试代码

[codes=galaxy]

```
void a()
{
    bool [2] b;
    if (true || b[2])
    {
    }
}
```

[/codes]

执行时没有错误提示。

奇怪么？实际上 b[2]已经超过数组上限，但是因为短路，所以 SC2 并没有调用这个数据，结果执行过程并没有错误提示。

3.5 循环结构

循环结构可以说是新手学习 Galaxy 的第一个难点，尽管循环并没有多少难度，但是确实有不少 WEer 不会使用（SE 现在的用户不多，不好下定论）。也许原因是循环结构的使用方式，更为抽象一些，并不像顺序结构与选择结构那么直观。事实上选择是否使用循环结构的判断条件就是你所要执行的代码之中是否存在一种规律，这种规律有两个条件：代码结构相同，部分值或结构有规律变化。

这样抽象的说也许会让你更加的糊涂。我们举例来说明。

比如，你购买东西时需要付款 10￥，你拿出两张 5￥的，共计 10￥。这个过程用顺序结构描述的话，就是：

[quote]
付款 5￥。
付款 5￥。
[/quote]

这样描述似乎比较麻烦，我们可以这样描述。

[quote]
付款 2 次，每次 5￥。
[/quote]

这就是将代码抽象成循环的过程。循环使代码更为简洁，少量的循环也许看不出来，如果你需要付款 1000￥，每次 5￥，共计 200 次，这样的情况下，循环的作用便清晰可见。那么能够抽象成循环的代码必须是完全相同吗？不是。你可以这样付款：

[quote]
付款 1￥。
付款 2￥。
付款 3￥。
付款 4￥。
[/quote]

抽象成循环：

[quote]
付款 4 次，第一次 1￥，每次多付 1￥。
[/quote]

这种才是最常使用的循环。相同结构，有规律的变化。然而，这“**有规律的变化**”也并非必备的。如此情况的付款：

[quote]

付款 1￥。
付款 3￥。
付款 3￥。
付款 2￥。
付款 1￥。
[/quote]

抽象成循环：

[quote]
付款 5 次：
第一次 1￥；
第二次 3￥；
第三次 3￥；
第四次 2￥；
第五次 1￥；
[/quote]

看到这里你会问一句：“这样岂不是更麻烦了吗？”确实是更加麻烦了。所以在实际应用中是否选择使用循环，我们需要考量。评判是否使用循环，依据是执行效率。这点就需要个人在使用 Galaxy 过程中慢慢体悟了。

Galaxy 中的循环语法只有一种：

[quote]
while （**逻辑值或逻辑表达式**）
{
循环代码;
continue;//依据需要使用。
break;//依据需要使用。
}
[/quote]

我们依照付款的实例来写个循环结构的代码。

代码 3-3 付款实例
[codes=galaxy]
```
int Cost = 0;
int I = 1;

//Type 1.
while (I <= 2)
{
    Cost += 5;
```

```
        I += 1;
    }

    //Type 2.
    int CostEachTime = 0
    while (I <= 4)
    {
        CostEachTime += 1;
        Cost += CostEachTime;
        I += 1;
    }

    //Type 3.
    while (I <= 5)
    {
        if (I == 1)
        {
            Cost += 1;
        }
        if (I == 2)
        {
            Cost += 3;
        }

        if (I == 3)
        {
            Cost += 3;
        }

        if (I == 4)
        {
            Cost += 2;
        }

        if (I == 5)
        {
            Cost += 1;
        }

        I += 1;
    }
[/codes]
```

注意到之前语法中的 continue 和 break？这两个语句有什么作用呢？先看以下代码。

代码 3-4 continue 和 break
[codes=galaxy]
```
    int i = 0;
    while (i < 2)
    {
        i += 1;
        TriggerDebugOutput(1,StringToText("a"),true);
        //continue;
        //break;
        TriggerDebugOutput(1,StringToText("b"),true);
    }
```
[/codes]

这是原始代码，输出结果是：

[quote]
a
b
a
b
[/quote]

不用说明原因了吧。如果我们去掉 continue 之前的注释符号//，输出结果是：

[quote]
a
a
[/quote]

原因很简单，continue 的作用是跳过当前循环 continue 语句后面的全部语句。

那么如果去掉 break 前面的注释符号//，输出是什么呢？

[quote]
a
[/quote]

原因是 break 的作用是跳出全部循环。

循环也可以嵌套使用，如：

代码 3-5 循环语句的嵌套使用。
[codes=galaxy]

```
    int I = 0;
    int J = 0;
    string [3][4] str;
    while (I < 3)
    {
        while (J < 4)
        {
            str[I][J] = IntToString(I) + "," + IntToString(J);
            J += 1;
        }
        I += 1;
    }
```
[/codes]

代码 3-5 的作用是为二维字符串数组 str 赋值。

使用循环结构时一定要注意，不要将语句写成死循环，即永远都满足循环条件并且没有 break 语句存在。很多时候我们会因为忘记写“I += 1;”这类步进代码而导致进入死循环。

代码 3-6 死循环实例

[codes=galaxy]
```
    int I = 0;
    int J = 0;
    string [3][4] str;
    while (I < 3)
    {
        while (J < 4)
        {
            str[I][J] = IntToString(I) + "," + IntToString(J);
        }
        I += 1;
    }
```
[/codes]

代码 3-6 会在 while（J < 4）中进入死循环，对比代码 3-5 可知，原因是忘记加入”j += 1;”语句，使 J 永远不会大于等于 4 而结束循环。

在 Galaxy 中死循环会在执行时得到错误提示：

[quote]
00:00:02.50 於’gt_E5AFB9E68898E5889DE5A78BE58C96_Func’ 的觸發器錯誤：執行時間太長
[/quote]

不过有些时候也许是你编写的脚本连续执行内容过多。才会获得此错误提示。

四．自定义函数与触发

4.1 GUI 下的地图脚本

尽管这是一篇 Galaxy 教程，我还是不得不加入 GUI 界面部分的内容。我在第一章也提到过，大家最好是学会 GUI 之后再尝试使用 Galaxy。之所以这样说，是因为 GUI 下你能够更直观的看到整个地图脚本构造，对于一些不复杂的脚本来说，甚至是一目了然。特别是如果你有 WE 的 T 的使用经验，学会 GUI 实际上比较简单的。

请不要认为我学会了使用 Galaxy 就可以完全脱离 GUI，实际上除非你使用导入 MapScript.galaxy 的方法来处理地图脚本，否则你必然要和 GUI 打交道。何况，库的制作不可能完全脱离 GUI。

这一节我会简要介绍下 GUI 下的地图脚本编写的基本知识，自定义事件、自定义动作、自定义函数以及库的制作等内容都会在后面逐步提到。

学习 Galaxy，首先要知道 SC2 脚本的基本结构。SC2 的脚本与 War3 的脚本结构基本相同，也都是触发引导函数执行的方式。

那么触发是什么？

触发器（Trigger）是 Galaxy 脚本的重要组成部分，它是玩家操作（包括电脑 AI 操作）与脚本执行之间的纽带。确切的说，整个脚本的全部函数，都是由触发引导执行的（实际上不完全是，具体在后面会提到）。当玩家或者 AI 操作满足某种状态时，触发就会执行对应预先写好的程序。

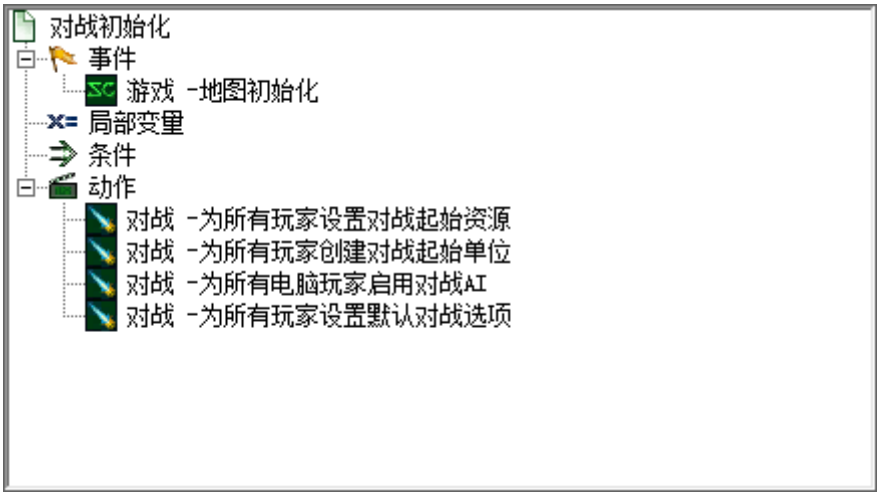


图 4-1 触发

从图 4-1，我们可以清楚的看到，触发由 4 个部分构成：事件、局部变量、条件和动作。其中局部变量比较特殊，实际上它并非触发真正的组成部分，而是属于动作部分。关于事件、条件与动作的说明，请看下面 Blizzard 官方教程中的叙述：

[quote]

“触发器模块”就是负责控制和疏导游戏玩法以及剧情走向的重要模块。创建游戏内剧

情动画、游戏中期暴兵、创建和跟踪任务目标等等游戏玩法的设计，都是通过“触发器模块”来进行的。

“触发器”可以视作为一组游戏指令的最基本构成单元。无论你何时想要在游戏中产生何种事件，你都可以使用一个“触发器”来指示游戏，游戏会准确地按照你的要求产生事件。一个触发器是由以下几种元素构成：

1. 事件 即导致“触发器”运行的原因。如果我们想创建一个“当英雄死亡时游戏即结束”的触发器时，那么这个触发器的事件就应该是“单位死亡”。

2. 条件 ——条件是指该“触发器”运行所需要满足的特定条件。比如在上面提到的例子中，当“触发器”因为某个单位死亡而运行时，我们可以设置一个“条件”以限制只有当死亡的单位是英雄时才运行。

3. 动作 ——动作即是当该“触发器”运行时游戏将会执行的指令。在我们的例子中，当英雄死亡时会执行的“动作”是“玩家 1 游戏结束”。

[/quote]

其实能够在 GUI 下用好事件、条件、和动作，你就可以算是一个入门的 SEer 了。你完全可以仅仅依靠使用这些，以及一些简单的数据设置，制作一张简单的 SC2 地图。当然，我们不会在此停步，我们还需要继续的学习下去。

在 GUI 界面下编写脚本，还要提到自定义脚本。

动作自定义脚本作，相当于在动作函数中插入一段自定义的 Galaxy 代码，注意不能包括函数或结构体等必须在全局范围内的内容，并且不能声明局部变量。

全局自定义脚本是在全局范围内编写（在触发器列表栏中）的，除去不能有其他范围（其他触发、自定义脚本等）重名的函数、变量外，没有特殊要求。但注意，这里面必须是函数构成的，不能有不在函数范围内的非声明语句。

4.2 Galaxy 下的函数

上一节提到了触发，那么，触发是由什么构成的呢？是由基本语句直接构成的么？

并非如此，虽然在 GUI 界面中你能够看到完整的触发，但是如果你打开地图的脚本文件，如代码 1-1，你会发现，Galaxy 脚本中，并没有直观触发的结构。你能看到的内容，除了声明就是函数。

函数是什么？

[quote]

函数过程中的这些语句用于完成某些有意义的工作——通常是处理文本，控制输入或计算数值。通过在程序代码中引入函数名称和所需的参数，可在该程序中执行（或称调用）该函数。

类似过程，不过函数一般都有一个返回值。它们都可在自己结构里面调用自己，称为递归。

——度娘

[/quote]

度娘的解释很渣，实际上编程语言中的函数较为类似数学的函数——给定参数返回结果。只是编程语言中的函数更为灵活，例如，可以没有输入的参数，可以不返回结果，而且在执行过程中还可以实现输出结果以外的功能。

总而言之，函数就是一段打包了的代码。这些代码往往需要重复使用，或者有着特殊的含义，我们把它包装起来，成为一个整体，就成为了函数。

那么，**定义**函数的语法结构是什么？

[quote]
类型标识符 函数名（形式参数表）
{
声明部分;
语句部分;
return **返回值**;
}
[/quote]

函数标识符就是 Galaxy 中的变量类型，另外还有一个特殊的无返回值函数标识符“void”。当函数的返回类型不为“void”时，必须有返回值，即有“return 返回值;”语句。

函数名的要求与变量名基本相同。

关于参数表，如果没有参数，留空即可，如果有多个参数，其格式如下：

[quote]
（**参数 1 的变量类型 参数 1 的变量名 ， 参数 2 的变量类型 参数 2 的变量名**，……）
[/quote]

注意多行参数以英文“,”分割。如果将参数的变量名留空，不会有语法错误，只是无法使用参数。

参数可以使用对应变量类型的常量、变量、表达式、函数。

返回值语句 return 返回的值要与函数的类型标识符对应，否则会产生语法错误。如果函数的类型为空（void）那么可以使用“Return;”语句返回，或者省略。如果类型非空，那么函数的最后一个语句必须为“return 返回值;”，否则语法错误。

我们可以看一个实际的函数。

代码 4-1 隐藏或显示单位动作
[codes=galaxy]
```
void libNtve_gf_ShowHideUnit (unit lp_unit, bool lp_showHide) {
    // Implementation
    if ((lp_showHide == true)) {
```

```
        UnitSetState(lp_unit, c_unitStateHidden, false);
    }
    else {
        UnitSetState(lp_unit, c_unitStateHidden, true);
    }
}
```
[/codes]

代码 4-1 就是**显示或隐藏单位**这个动作的实际函数。这是一个官方函数，出自 NativeLib.galaxy。这个函数的类型标识符为 void，代表这是一个没有返回值的函数。函数名为 libNtve_gf_ShowHideUnit，一共有两个参数，分别为单位类型的 lp_unit 和布尔值类型的 lp_showHide.

知道了函数结构，下一步我们就需要知道怎么调用函数。

执行函数的语法：

[quote]

函数名（参数表）

[/quote]

请注意：函数需要在调用前**定义**。否则会有语法错误。

那么，有没有办法在调用还没有**定义**的函数？

有，那就是函数的声明。**不知道大家注意了没有，这节我提到的都是函数的“定义”，并且都加粗字体。实际上就是为了区别函数的“定义”和函数的“声明”。**

函数的“定义”及语法都在前面说明过，函数的声明又是什么呢？我们先来看看函数声明的语法：

[quote]

类型标识符 函数名（参数表）；

[/quote]

注意最后有个英文“;”存在。

函数的声明类似变量声明——声明后即可使用。如你所见，函数的声明并没有写函数体，只是说明了函数的返回类型、函数的名称及函数的参数。具体应用请看代码 5-2.

代码 4-2 函数的声明与定义实例

[codes=galaxy]

```
    void a();
    void a()
    {
        return;
    }
```
[/codes]

此外，有一点需要说明一下，一个有返回值的函数可以在不使用其结果的条件下使用，如：

代码 4-3 不使用返回值时函数调用

[codes=galaxy]
```
int a()
{
    return 0;
}

a();
```
[/codes]

函数不能嵌套定义或声明，一个函数必须是全局的，但函数可以嵌套调用，甚至可以再一个语句中嵌套使用函数，如：

代码 4-4 函数的嵌套调用

[codes=galaxy]
```
    MaxI(RandomInt(0,100),RandomInt(0,100));
```
[/codes]

我们在调用比较整数大小函数 MaxI（）中使用随机整数函数 RandomInt（）的返回值作为参数。

另外还要说一下一种特殊的函数调用状态——递归。递归是指在调用一个函数的过程中，直接或间接的调用该函数本身的状态。如：

代码 4-5 递归示例

[codes=galaxy]
```
int a()
{
    return a();
}
```
[/codes]

当然不会有人使用代码 4-5，因为那明显是一个死循环。其实递归也可以看做循环，部分递归结构可以用循环结构替代。至于选择递归还是循环，请依据自己的代码内容决定。一般来说，因为递归涉及函数调用，在效率上及代码可读性上不如循环，所以一般来说，我们使用循环较多。

代码 4-5 为直接调用，间接嵌套调用需要函数的声明。如

代码 4-6 间接调用

[codes=galaxy]

```
    void a();
    void b();

    void a()
    {
        b();
    }
    void b()
    {
        a();
    }
[/codes]
```

4.3 全局变量与局域变量以及函数参数类别

看到这节的标题，你也许会问，不是在讲函数么，怎么又讲回变量？关于变量的内容，直接写到第二章不好吗？

全局变量和局域变量的差别在于变量的可用范围。全局变量的范围是整个脚本（static标志的略有差别，具体内容在后面叙述），而局域变量的范围是其声明的函数。至于参数，参数也是局部变量，其有效范围为其对应的函数。

代码 4-7 全局变量和局部变量的有效范围

```
[codes=galaxy]
int i = 0;

void a()
{
    TriggerDebugOutput(1, IntToText(i) , true);
}

void b()
{
    int i = 1;
    int j = 0;
    TriggerDebugOutput(1, IntToText(i) , true);
}

void c()
{
int j = 1;
    TriggerDebugOutput(1, IntToText(j) , true);
}
void d(int i)
{
```

```
    TriggerDebugOutput(1, IntToText(i) , true);
}

void e(int i)
{
int i = 1;
    TriggerDebugOutput(1, IntToText(i) , true);
}
```
[/codes]

TriggerDebugOutput()函数的作用是将文本输出到 Debug 窗口以及游戏。

那么当我们调用 a()，b()，c(),d(),e()。输出的结果是什么呢？有兴趣的话，你可以自己测试下。

函数 a()中的变量 i 使用的是全局变量，也就是 0；b()中使用了局部变量，尽管它与全局变量的变量名相同，但是在函数 b()中它还是 1，即局部变量如果跟全局变量重名，那么有效的值是局部变量，无论读取还是赋值，都不影响全局变量；c()中使用的是变量 j，这个变量的同名变量在 b()中已经声明，但是 c()的显示值是 1，即局部变量和不同函数的局部变量同名时，相互之间不影响，他们的有效值都是当前函数；函数 d()中的 i 是参数，函数的显示结果为输入的值，这点应该没有什么疑问吧；最后是函数 e()，你猜结果会是什么呢？结果是什么也没有，因为参数不能与局域变量重名，否则编译报错。

此外，要注意一下几点：

与函数调用相同，全局变量可以在任意处声明，但需要在声明后才可使用。局域变量必须在函数内最先声明，然后才是执行语句。

某函数内的局域变量或参数的变量名不能与所在函数的函数名相同。全局变量的变量名不能跟任意函数的函数名相同。否则编译时报错。

需要特殊说明前置标志“static”。这个标志只能在全局变量或者函数前使用，作用是表明声明、定义的变量与函数的有效范围为当前文件。如果你导入脚本文件，使用 include 加载，那么导入文件中的全局变量和函数都可以使用，加上“static”标志的除外。

这个标志多用于多人合作编写时，避免变量名、函数名冲突。一般情况下我们极少使用。

介绍变量的有效范围之后，就不得不说参数了。因为一个函数是封装的，其内部局域变量无法用于函数间传递数据，那么能够在函数间传递数据的就只有全局变量和参数（指使用变量范围，关于数据传递的其他方法会在后面介绍）。全局变量在某些情况下回出现一些问题，比如重复调用有等待的延时执行函数，会因为全局变量的值的变化产生 bug。所以调用数据一般都使用参数（这是必然的，不然要参数做什么）。

那么不知道大家考虑没有考虑过这样一个问题，当某个变量作为参数输入到某个函数中时，如果在这个函数内修改这个参数的值，那么原来参数的值是否会改变呢？

代码 4-8 实参与形参

[codes=galaxy]

```
void a(int b)
{
    b = 1;
}
int c()
{
    int d = 0;
    a(d);
    return d;
}
```
[/codes]

如代码 4-8 中，函数 c 的返回值是多少？0 还是 1？其真正的结果是 0。

我们称变量 d（调用函数时的参数变量）这样的参数是实际参数（实参）；称参数 b（函数声明的参数）为形式参数（形参）。

而我们调用函数时，被调用的函数的形参相当于一个新声明的局部变量。它与实际参数并无直接关系，无论你对其做任何赋值上的修改，都不会对实参有所影响。

但是，这里还有一点例外的情况。我们在节 2.1 的最后提到基础变量类型和复合变量类型，其中二者的差别就是这一点例外。

代码 4-9 复合变量的情况

[codes=galaxy]
```
void a(unit b)
{
UnitRemove(b);
b = null;
}
unit c()
{
    unit d = UnitFromId(1);
    a(d);
    return d;
}
```
[/codes]

类似代码 4-8，函数 c()的输出依旧是“UnitFromId(1);”，不过有一点，UnitRemove()函数的执行有效。即，函数 c()的输出值的单位被删除了。

这里你也许没看懂。但是实际上这并不复杂。当基础变量作为实参时，形参相当于一个实参的复制品。无论你对其值做任何修改，都无法影响到实参自身。但是如果使用复合变量做实参，对应的形参依旧是个复制品，但是复制的内容只有引用而不包括复合数据本身。所以当你对形参做部分改变（非引用改变，如赋值）将会影响到实参。

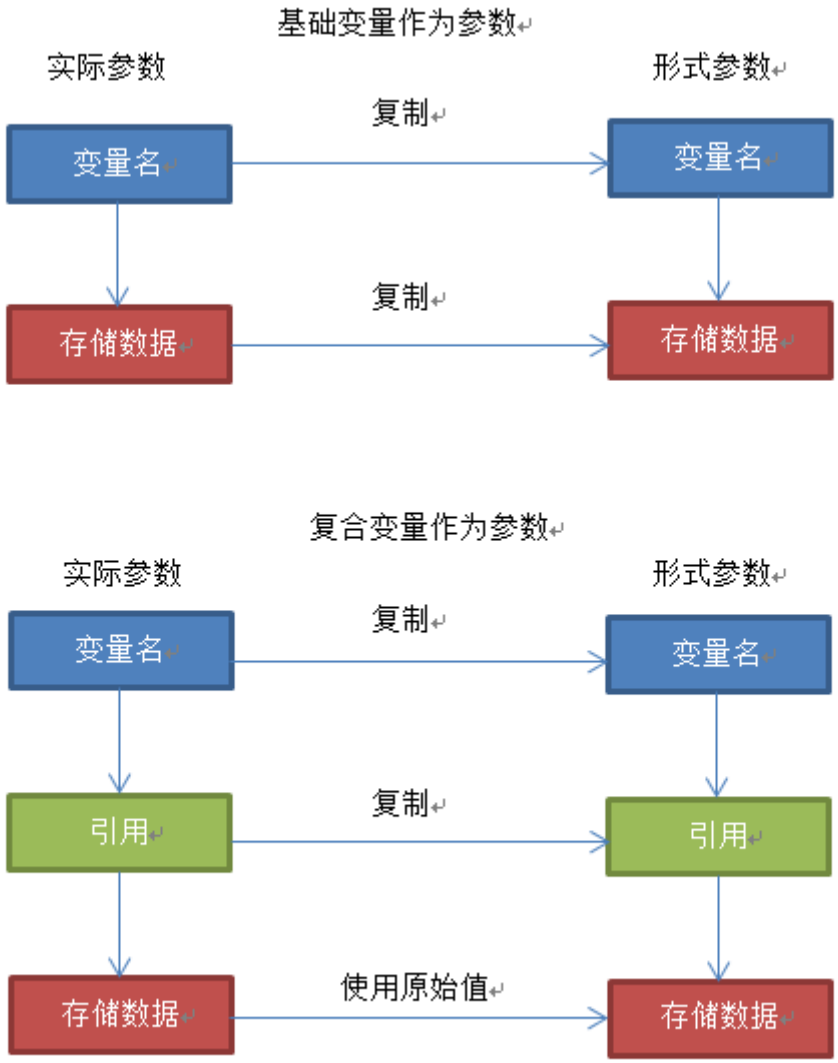


图 4-2 基础变量与复合变量作为参数

以上存储关系为个人猜测，未实际验证，仅作为理解实参形参关系的辅助。

4.4 Galaxy 下的触发与 SC2 地图的脚本结构

节 4.2 中我们说到，在地图脚本文件 MapScript.galaxy 中，无法明显的看到触发结构，那么 Galaxy 下并没有 GUI 中触发结构么？

非也。让我们来看代码 4-10.

代码 4-10 MapScript.galaxy

[codes=galaxy]

```
//==================================================================
================================
//
// Generated Map Script
```

```
//
// Name:   libGAWH_cf_CircumcircleCenterOfCircle
// Author: WhimsyDuke
//
//=========================================================================
================================
include "TriggerLibs/NativeLib"

//-------------------------------------------------------------------------
--------------------------------
// Library Initialization
//-------------------------------------------------------------------------
--------------------------------
void InitLibs () {
    libNtve_InitLib();
}

//-------------------------------------------------------------------------
--------------------------------
// Trigger Variables
//-------------------------------------------------------------------------
--------------------------------
trigger gt_Init;

//-------------------------------------------------------------------------
--------------------------------
// Custom Script: Script
//-------------------------------------------------------------------------
--------------------------------
point libGAWH_cf_CircumcircleCenterOfCircle(point lp_PointA, point
lp_PointB, point lp_PointC)
{
    fixed [2] lv_AngleOfPerpendicularBisector;
    fixed [2] lv_b;
    fixed [6] lv_X;
    fixed [6] lv_Y;

    lv_X[2] = PointGetX(lp_PointA);
    lv_X[3] = PointGetX(lp_PointB);
    lv_X[4] = PointGetX(lp_PointC);
    lv_X[0] = (lv_X[2] + lv_X[3]) / 2;
    lv_X[1] = (lv_X[2] + lv_X[4]) / 2;
    lv_Y[2] = PointGetY(lp_PointA);
    lv_Y[3] = PointGetY(lp_PointB);
```

```
    lv_Y[4] = PointGetY(lp_PointC);
    lv_Y[0] = (lv_Y[2] + lv_Y[3]) / 2;
    lv_Y[1] = (lv_Y[2] + lv_Y[4]) / 2;

    lv_AngleOfPerpendicularBisector[0] = -(lv_X[2] - lv_X[3]) / (lv_Y[2]
- lv_Y[3]);
    lv_AngleOfPerpendicularBisector[1] = -(lv_X[2] - lv_X[4]) / (lv_Y[2]
- lv_Y[4]);

    lv_b[0] = lv_Y[0] - lv_AngleOfPerpendicularBisector[0] * lv_X[0];
    lv_b[1] = lv_Y[1] - lv_AngleOfPerpendicularBisector[1] * lv_X[1];

    lv_X[5] = (lv_b[0] - lv_b[1]) / (lv_AngleOfPerpendicularBisector[1]
- lv_AngleOfPerpendicularBisector[0]);
    lv_Y[5] = lv_AngleOfPerpendicularBisector[0] * lv_X[5] + lv_b[0];
    MaxI(RandomInt(0,100),RandomInt(0,100));

    return Point(lv_X[5], lv_Y[5]);
}

//--------------------------------------------------------------------
--------------------------------
// Custom Script Initialization
//--------------------------------------------------------------------
--------------------------------
void InitCustomScript () {
}

//--------------------------------------------------------------------
--------------------------------
// Trigger: Init
//--------------------------------------------------------------------
--------------------------------
bool gt_Init_Func (bool testConds, bool runActions) {
    // Variable Declarations
    point lv_a;
    point lv_b;
    point lv_c;
    point lv_center;

    // Variable Initialization
    lv_a = RegionRandomPoint(RegionPlayableMap());
    lv_b = RegionRandomPoint(RegionPlayableMap());
    lv_c = RegionRandomPoint(RegionPlayableMap());
```

```
    lv_center = RegionRandomPoint(RegionPlayableMap());

    // Actions
    if (!runActions) {
        return true;
    }

    lv_center = libGAWH_cf_CircumcircleCenterOfCircle(lv_a,lv_b,lv_c);
    TriggerDebugOutput(1,
(FixedToTextAdvanced(DistanceBetweenPoints(lv_center, lv_a),
c_formatNumberStyleNormal, true, 2, 2)), true);
    TriggerDebugOutput(1,
(FixedToTextAdvanced(DistanceBetweenPoints(lv_center, lv_b),
c_formatNumberStyleNormal, true, 2, 2)), true);
    TriggerDebugOutput(1,
(FixedToTextAdvanced(DistanceBetweenPoints(lv_center, lv_c),
c_formatNumberStyleNormal, true, 2, 2)), true);
    return true;
}

//--------------------------------------------------------------------
--------------------------------
void gt_Init_Init () {
    gt_Init = TriggerCreate("gt_Init_Func");
    TriggerAddEventKeyPressed(gt_Init, c_playerAny, c_key1, true,
c_keyModifierStateIgnore, c_keyModifierStateIgnore,
c_keyModifierStateIgnore);
}

//--------------------------------------------------------------------
--------------------------------
// Trigger Initialization
//--------------------------------------------------------------------
--------------------------------
void InitTriggers () {
    gt_Init_Init();
}

//--------------------------------------------------------------------
--------------------------------
// Map Initialization
//--------------------------------------------------------------------
--------------------------------
void InitMap () {
```

```
    InitLibs();
    InitCustomScript();
    InitTriggers();
}
```
[/codes]

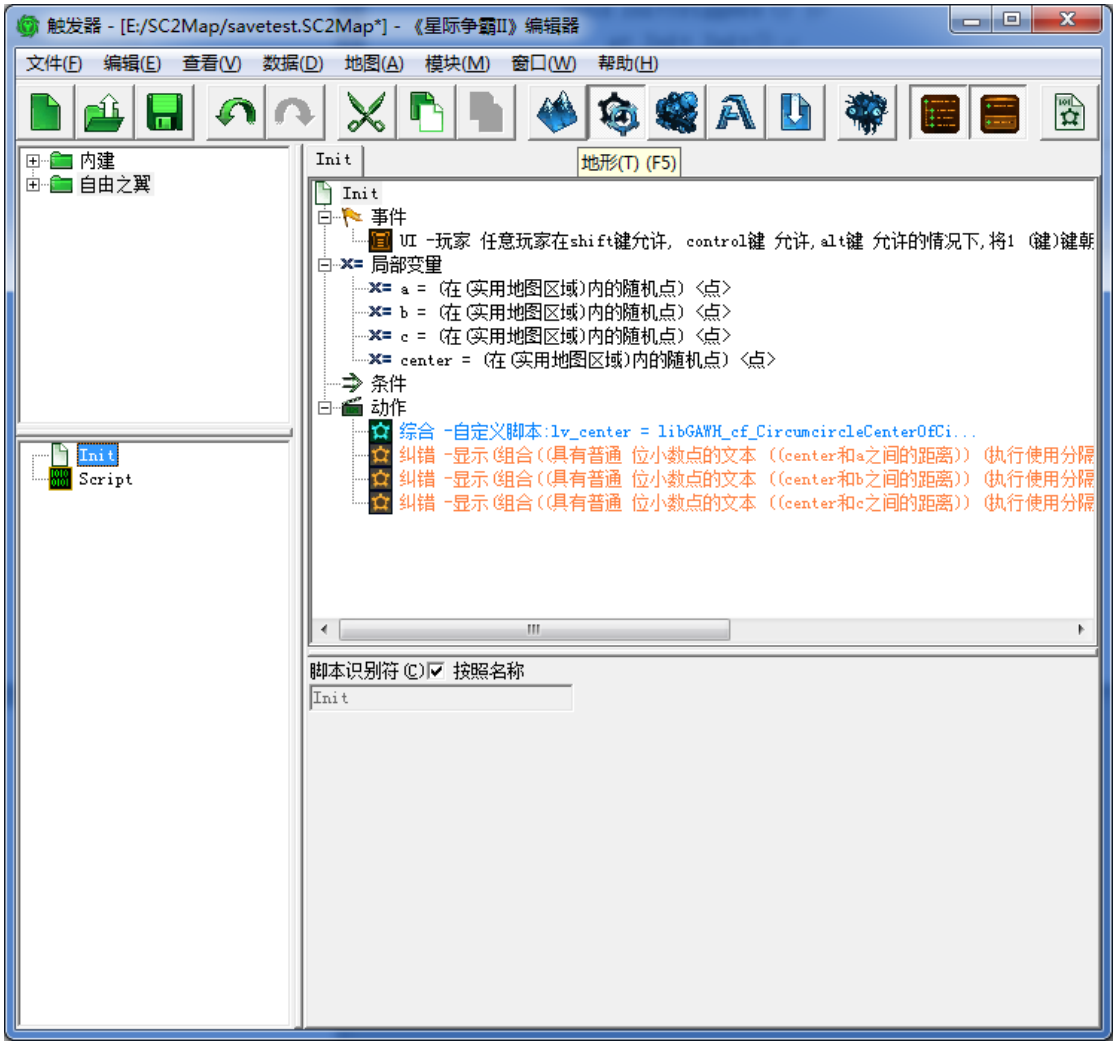


图 4-3 代码 4-10 的 GUI 截图

果然直接看不出来……怎么办呢？看一段代码，除非已经了解需要寻找的地方，否则都是按照执行、调用顺序来看。那么。代码 4-10 中的脚本入口在哪呢？（脚本入口就是初始执行的函数）。

很不错的一点，GUI 生成的脚本都有详细的注释。我们看最后一个函数前的注释：

[quote]
```
//--------------------------------------------------------------------------------
--------------------
// Map Initialization
//--------------------------------------------------------------------------------
--------------------
```
[/quote]

明显的写着地图初始化，确实，最后一个函数 InitMap()也就是整个脚本的主函数。所谓主函数，就是主要函数。在 Galaxy 下，指地图加载过程中首先执行的函数。其函数名为 InitMap ()固定不变，任何地图的 MapScript.galaxy 文件中都必须有这样一个函数，否则地图无法加载。

InitMap()函数调用了三个函数，分别为：

1、 InitLibs();这个函数是库的初始化函数，它调用了的 libNtve_InitLib()函数是库 NativeLib.galaxy 的初始化函数。这里要多提一句，大家注意到 include "TriggerLibs/NativeLib"这个语句了么？这句的作用就是预载 NativeLib.galaxy 作为头文件，然后我们就可以调用 NativeLib.galaxy 中的函数了。关于预载的详细内容，将会在后面详细讲解。

2、 InitCustomScript();这个是自定义脚本的预载。我们添加的全局自定义脚本（如图 4-3 中的 Script)是可以添加初始化函数的，添加位置在自定义脚本编写框下方，如图 4-3

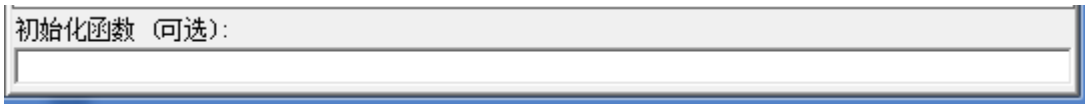


图 4-4 全局自定义脚本

3、 InitTriggers();这个函数就是触发器的初始化了。我们来看看其内容是什么：

[quote]
void InitTriggers () {
    gt_Init_Init();
}
[/quote]

现在我们只看到 gt_Init_Init()这一个函数，这个函数所属的注释范围的内容是：

代码 4-11 触发 Init

[codes=galaxy]
```
//--------------------------------------------------------------------------------
// Trigger: Init
//--------------------------------------------------------------------------------
bool gt_Init_Func (bool testConds, bool runActions) {
    // Variable Declarations
    point lv_a;
    point lv_b;
    point lv_c;
    point lv_center;
```

```
    // Variable Initialization
    lv_a = RegionRandomPoint(RegionPlayableMap());
    lv_b = RegionRandomPoint(RegionPlayableMap());
    lv_c = RegionRandomPoint(RegionPlayableMap());
    lv_center = RegionRandomPoint(RegionPlayableMap());

    // Actions
    if (!runActions) {
        return true;
    }

    lv_center = libGAWH_cf_CircumcircleCenterOfCircle(lv_a,lv_b,lv_c);
    TriggerDebugOutput(1,
(FixedToTextAdvanced(DistanceBetweenPoints(lv_center, lv_a),
c_formatNumberStyleNormal, true, 2, 2)), true);
    TriggerDebugOutput(1,
(FixedToTextAdvanced(DistanceBetweenPoints(lv_center, lv_b),
c_formatNumberStyleNormal, true, 2, 2)), true);
    TriggerDebugOutput(1,
(FixedToTextAdvanced(DistanceBetweenPoints(lv_center, lv_c),
c_formatNumberStyleNormal, true, 2, 2)), true);
    return true;
}

//--------------------------------------------------------------------
--------------------------------
void gt_Init_Init () {
    gt_Init = TriggerCreate("gt_Init_Func");
    TriggerAddEventKeyPressed(gt_Init, c_playerAny, c_key1, true,
c_keyModifierStateIgnore, c_keyModifierStateIgnore,
c_keyModifierStateIgnore);
}
```
[/codes]

看明白了么？如果你有 J 的经验，你能看出这里整个是一个触发，由注释可见，这个触发就是图 4-1 中的 Init 这个触发，而 void gt_Init_Init ()正是这个触发的注册函数。

确实如此。所谓触发，从代码上来看，由两个函数和一个变量构成。对应名称如下。记录触发的变量：

[quote]
trigger gt_**触发名**
[/quote]

触发注册函数：

[quote]
void gt_Init_**触发名**（）
[/quote]

触发条件动作函数：

[quote]
bool gt_**触发名**_Func (bool testConds, bool runActions)
[/quote]

当地图加载时，SC2 通过主函数 InitMap ()调用触发初始化函数 InitTriggers()。每一个有效的触发都在这里初始化——创建触发并注册事件（如果有）。当触发因某注册事件触发后，SC2 就会调用其对应触发条件动作函数。

当然，如果我们完全自主编写一个地图的全部脚本，除了脚本主函数需要固定函数名为 InitMap ()，不必以此顺序执行。并且也可以在一个触发注册函数注册全部触发。但是需要注意一点，作为触发条件动作函数的函数必须为bool 函数，并且使用指定的(bool testConds 和 bool runActions)参数，否则无法编译。

触发也可以写在全局自定义脚本中。请看代码 4-10，自定义代码 Script 也在文件中。注释“// Custom Script: Script”之后的就是。

4.5 函数的数据存储及函数间的数据传输

数据存储，一般来说使用变量即可，多个连续的数据，则采用数组。更为复杂的数据也可以采用多维数组存储，相比 J 下 8192 个元素上限的数组系统，Galaxy 的数组无疑强大很多，但是仍然有个致命的缺陷，那就是数组无法动态声明，即我们必须在编写脚本时确定一个函数的大小。

如我们需要将某个区域内的全部单位放入一个单位数组（不是单位组，是一个数组，变量类型为单位），那么我们要将这个数组的大小设置为多少呢？因为区域内的单位数量不确定，所以我们很有可能因超出数组范围而导致 bug 产生。当然你也可以使用一个很大的整数来声明数组，如“int[9999]这种”。但这会占用过多内存，导致游戏过慢。

那么，解决方案是什么呢？

解决方案是 DataTable。DataTable 是数据表（zhCN 下 SE 的翻译，对应的数据集文件指 bank）的名称。它很类似 WE 中的 HashTable（简称 HT）或 GameCache(简称 GC)。不过 DataTable 采用的 Key 是一个字符串，并非 HT 的两个整数或 GC 的两个字符串。相比之下，DataTable 的执行效率相当的高，完全不像 GC 那样缓慢。

因此，解决数组大小问题的方法就是用 DataTable 替代数组使用。

代码 4-12 DataTable 相关函数
[codes=galaxy]

```
//---------------------------------------------------------------------
------------------------------
// Data Table
// - Data tables provide named storage for any script type.
//   Table access may be either global or thread-local.
//---------------------------------------------------------------------
------------------------------
// Types
const int c_dataTypeUnknown           = -1;
const int c_dataTypeAbilCmd           =  0;
const int c_dataTypeActor             =  1;
const int c_dataTypeActorScope        =  2;
const int c_dataTypeAIFilter          =  3;
const int c_dataTypeBank              =  4;
const int c_dataTypeBool              =  5;
const int c_dataTypeByte              =  6;
const int c_dataTypeCameraInfo        =  7;
const int c_dataTypeCinematic         =  8;
const int c_dataTypeColor             =  9;
const int c_dataTypeControl           = 10;
const int c_dataTypeConversation      = 11;
const int c_dataTypeDialog            = 12;
const int c_dataTypeDoodad            = 13;
const int c_dataTypeFixed             = 14;
const int c_dataTypeInt               = 15;
const int c_dataTypeMarker            = 16;
const int c_dataTypeObjective         = 17;
const int c_dataTypeOrder             = 18;
const int c_dataTypePing              = 19;
const int c_dataTypePlanet            = 20;
const int c_dataTypePlayerGroup       = 21;
const int c_dataTypePoint             = 22;
const int c_dataTypePortrait          = 23;
const int c_dataTypeRegion            = 24;
const int c_dataTypeReply             = 25;
const int c_dataTypeRevealer          = 26;
const int c_dataTypeSound             = 27;
const int c_dataTypeSoundLink         = 28;
const int c_dataTypeString            = 29;
const int c_dataTypeText              = 30;
const int c_dataTypeTimer             = 31;
const int c_dataTypeTransmission      = 32;
const int c_dataTypeTransmissionSource  = 33;
const int c_dataTypeTrigger           = 34;
```

```
const int c_dataTypeUnit              = 35;
const int c_dataTypeUnitFilter         = 36;
const int c_dataTypeUnitGroup          = 37;
const int c_dataTypeUnitRef            = 38;
const int c_dataTypeWave              = 39;
const int c_dataTypeWaveInfo           = 40;
const int c_dataTypeWaveTarget         = 41;

// General functionality
native void    DataTableClear (bool global);
native int     DataTableValueCount (bool global);
native string   DataTableValueName (bool global, int index);
native bool    DataTableValueExists (bool global, string name);
native int     DataTableValueType (bool global, string name);
native void    DataTableValueRemove (bool global, string name);

// Type-specific value set/get
// - c_dataTypeAbilCmd
native void       DataTableSetAbilCmd (bool global, string name, abilcmd
val);
native abilcmd     DataTableGetAbilCmd (bool global, string name);

// - c_dataTypeActor
native void       DataTableSetActor (bool global, string name, actor
val);
native actor       DataTableGetActor (bool global, string name);

// - c_dataTypeActorScope
native void       DataTableSetActorScope (bool global, string name,
actorscope val);
native actorscope   DataTableGetActorScope (bool global, string name);

// - c_dataTypeAIFilter
native void       DataTableSetAIFilter (bool global, string name,
aifilter val);
native aifilter    DataTableGetAIFilter (bool global, string name);

// - c_dataTypeBank
native void       DataTableSetBank (bool global, string name, bank val);
native bank        DataTableGetBank (bool global, string name);

// - c_dataTypeBool
native void       DataTableSetBool (bool global, string name, bool val);
native bool        DataTableGetBool (bool global, string name);
```

```
// - c_dataTypeByte
native void        DataTableSetByte (bool global, string name, byte val);
native byte        DataTableGetByte (bool global, string name);

// - c_dataTypeCameraInfo
native void        DataTableSetCameraInfo (bool global, string name,
camerainfo val);
native camerainfo   DataTableGetCameraInfo (bool global, string name);

// - c_dataTypeCinematic
native void        DataTableSetCinematic (bool global, string name, int
val);
native int         DataTableGetCinematic (bool global, string name);

// - c_dataTypeColor
native void        DataTableSetColor (bool global, string name, color
val);
native color       DataTableGetColor (bool global, string name);

// - c_dataTypeControl
native void        DataTableSetControl (bool global, string name, int
val);
native int         DataTableGetControl (bool global, string name);

// - c_dataTypeConversation
native void        DataTableSetConversation (bool global, string name,
int val);
native int         DataTableGetConversation (bool global, string name);

// - c_dataTypeDialog
native void        DataTableSetDialog (bool global, string name, int
val);
native int         DataTableGetDialog (bool global, string name);

// - c_dataTypeDoodad
native void        DataTableSetDoodad (bool global, string name, doodad
val);
native doodad      DataTableGetDoodad (bool global, string name);

// - c_dataTypeFixed
native void        DataTableSetFixed (bool global, string name, fixed
val);
native fixed       DataTableGetFixed (bool global, string name);
```

```
// - c_dataTypeInt
native void        DataTableSetInt (bool global, string name, int val);
native int         DataTableGetInt (bool global, string name);

// - c_dataTypeMarker
native void        DataTableSetMarker (bool global, string name, marker
val);
native marker      DataTableGetMarker (bool global, string name);

// - c_dataTypeObjective
native void        DataTableSetObjective (bool global, string name, int
val);
native int         DataTableGetObjective (bool global, string name);

// - c_dataTypeOrder
native void        DataTableSetOrder (bool global, string name, order
val);
native order       DataTableGetOrder (bool global, string name);

// - c_dataTypePing
native void        DataTableSetPing (bool global, string name, int val);
native int         DataTableGetPing (bool global, string name);

// - c_dataTypePlanet
native void        DataTableSetPlanet (bool global, string name, int
val);
native int         DataTableGetPlanet (bool global, string name);

// - c_dataTypePlayerGroup
native void        DataTableSetPlayerGroup (bool global, string name,
playergroup val);
native playergroup  DataTableGetPlayerGroup (bool global, string name);

// - c_dataTypePoint
native void        DataTableSetPoint (bool global, string name, point
val);
native point       DataTableGetPoint (bool global, string name);

// - c_dataTypePortrait
native void        DataTableSetPortrait (bool global, string name, int
val);
native int         DataTableGetPortrait (bool global, string name);
```

```
// - c_dataTypeRegion
native void        DataTableSetRegion (bool global, string name, region
val);
native region      DataTableGetRegion (bool global, string name);

// - c_dataTypeReply
native void        DataTableSetReply (bool global, string name, int val);
native int         DataTableGetReply (bool global, string name);

// - c_dataTypeRevealer
native void        DataTableSetRevealer (bool global, string name,
revealer val);
native revealer    DataTableGetRevealer (bool global, string name);

// - c_dataTypeSound
native void        DataTableSetSound (bool global, string name, sound
val);
native sound       DataTableGetSound (bool global, string name);

// - c_dataTypeSoundLink
native void        DataTableSetSoundLink (bool global, string name,
soundlink val);
native soundlink   DataTableGetSoundLink (bool global, string name);

// - c_dataTypeString
native void        DataTableSetString (bool global, string name, string
val);
native string      DataTableGetString (bool global, string name);

// - c_dataTypeText
native void        DataTableSetText (bool global, string name, text val);
native text        DataTableGetText (bool global, string name);

// - c_dataTypeTimer
native void        DataTableSetTimer (bool global, string name, timer
val);
native timer       DataTableGetTimer (bool global, string name);

// - c_dataTypeTransmission
native void        DataTableSetTransmission (bool global, string name,
int val);
native int         DataTableGetTransmission (bool global, string name);

// - c_dataTypeTransmissionSource
```

```
native void              DataTableSetTransmissionSource (bool global,
string name, transmissionsource val);
native transmissionsource  DataTableGetTransmissionSource (bool global,
string name);

// - c_dataTypeTrigger
native void        DataTableSetTrigger (bool global, string name, trigger
val);
native trigger     DataTableGetTrigger (bool global, string name);

// - c_dataTypeUnit
native void        DataTableSetUnit (bool global, string name, unit val);
native unit        DataTableGetUnit (bool global, string name);

// - c_dataTypeUnitFilter
native void        DataTableSetUnitFilter (bool global, string name,
unitfilter val);
native unitfilter   DataTableGetUnitFilter (bool global, string name);

// - c_dataTypeUnitGroup
native void        DataTableSetUnitGroup (bool global, string name,
unitgroup val);
native unitgroup    DataTableGetUnitGroup (bool global, string name);

// - c_dataTypeUnitRef
native void        DataTableSetUnitRef (bool global, string name, unitref
val);
native unitref     DataTableGetUnitRef (bool global, string name);

// - c_dataTypeWave
native void        DataTableSetWave (bool global, string name, wave val);
native wave        DataTableGetWave (bool global, string name);

// - c_dataTypeWaveInfo
native void        DataTableSetWaveInfo (bool global, string name,
waveinfo val);
native waveinfo     DataTableGetWaveInfo (bool global, string name);

// - c_dataTypeWaveTarget
native void        DataTableSetWaveTarget (bool global, string name,
wavetarget val);
native wavetarget   DataTableGetWaveTarget (bool global, string name);
```
[/codes]

对应每一个变量类型，都有两个函数。分别为：

1、存储数据。

[quote]

native void DataTableSet **变量类型标识符** (bool global, string name, **对应变量类型 对应变量类型的存储值**);

[/quote]

2、读取数据。

[quote]

native void DataTableGet **变量类型标识符** (bool global, string name);

[/quote]

除此之外，还有六个通用函数：

[quote]

```
// General functionality
native void     DataTableClear (bool global);
native int      DataTableValueCount (bool global);
native string   DataTableValueName (bool global, int index);
native bool     DataTableValueExists (bool global, string name);
native int      DataTableValueType (bool global, string name);
native void     DataTableValueRemove (bool global, string name);
```

[/quote]

作用分别为：清理全部数据（分局域和全局范围）；存储总值计数；根据存储的顺序序号获取对应的 Key；判断对应 Key 的值是否存在，对应 Key 存储的变量类型；删除 Key 对应的值。

这里的参数 global 是控制这个 DataTable 存储数据的有效范围的，类似全局变量与局域变量。

如果我们需要使用一个数组名为 I 的整数数组，我们可以这样使用

代码 4-12 用 DataTable 实现数组

[codes=galaxy]

```
//Const
const string libGAWH_gv_cf_ArraySystemFirstName = "ArraySystem:";

//Function
void libGAWH_cf_ArraySystemSetInt(string lp_ArrayName, int lp_Index, int
lp_ArrayData)
{
```

```
    DataTableSetInt(true, libGAWH_gv_cf_ArraySystemFirstName +
lp_ArrayName + IntToString(lp_Index), lp_ArrayData);
}

int libGAWH_cf_ArraySystemGetInt(string lp_ArrayName, int lp_Index)
{
    return DataTableGetInt(true, libGAWH_gv_cf_ArraySystemFirstName +
lp_ArrayName + IntToString(lp_Index));
}
[/codes]
```

实际上用这样的函数来处理有点多此一举。直接使用 DataTable 函数处理也许更好些。

数据存储可以这样解决。下面就是数据传输。

函数间的数据传输，实际上并不难。简单来说，需要传入某函数的数据作为参数即可，需要返回到调用函数的，用返回值即可。

那么，会有什么问题？

问题很多，如函数的多值返回。

所谓多值返回，就是指一个函数返回一个以上的值。而因为函数的 return 语句只能带一个值，而函数本身会在执行到 return 语句处退出，所以使用返回值我们只能返回一个值。

当然，聪明的你会想出用复合变量来返回多个值的办法，如用 point 返回两个 fixed。这确实是不错的想法，但是实际中应用范围太窄。

那么用结构体？不幸的是，结构体不能作为函数返回类型。另外，Galaxy 也不支持数组作为参数——不是数组元素，而是一个数组本身作为参数——除非我们写一堆参数，然后把数组元素逐一作为实际参数调用。这样的话，仍然是不方便。

那么解决方案是什么？依旧是 DataTable，将需要返回的值存储到 DataTable 中，然后返回 DataTable 的 Key（string 变量类型的参数 name）即可。然后在调用函数中使用 Key 与数据序号即可。

Handle 值存储结构：

Galaxy 下并没有 Handle 这一类型变量。Handle 是 WAR3 的脚本语言中使用的一种综合变量类型。当然，我们没必要理解它是什么，我们只要知道它的存储方式即可，在实际应用中，我们会用到。

当然这种数据的存储结构也许有其正是名称。这里我暂时称其为 Handle 存储结构。

这种存储方式说来也简单，它只是一种可循环使用 Index（序号）分配方式，具体规则如下：

1、 有一个记录使用 Index 最大值的变量，从 0 开始递增，在一次游戏过程中，不减少或重置。
2、 有一个记录释放（不在使用的）Index 值的数组（栈结构），此数组内 Index 值无序，每当释放一个 Index 时入栈。
3、 当从系统获取一个 Index 时，优先判断释放 Index 数组（栈）是否为空（可用元素数量是否为 0）。若非空，读取数组最后一个 Index 值，并将数组元素数量记录减一

（相当于出栈）；若为空，给出记录使用 Index 最大值的变量的值，并将此值加一。

关于栈，这里做下说明：

[quote]

基本概念

栈，是硬件。主要作用表现为一种数据结构，是只能在某一端插入和删除的特殊线性表。它按照后进先出的原则存储数据，先进入的数据被压入栈底，最后的数据在栈顶，需要读数据的时候从栈顶开始弹出数据（最后一个数据被第一个读出来）。

栈是允许在同一端进行插入和删除操作的特殊线性表。允许进行插入和删除操作的一端称为栈顶(top)，另一端为栈底(bottom)；栈底固定，而栈顶浮动；栈中元素个数为零时称为空栈。插入一般称为进栈（PUSH），删除则称为退栈（POP）。栈也称为后进先出表。

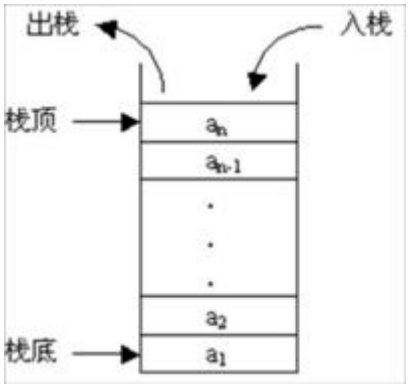
栈可以用来在函数调用的时候存储断点，做递归时要用到栈！

以上定义是在经典计算机科学中的解释。

在计算机系统中，栈则是一个具有以上属性的动态内存区域。程序可以将数据压入栈中，也可以将数据从栈顶弹出。在 i386 机器中，栈顶由称为 esp 的寄存器进行定位。压栈的操作使得栈顶的地址减小，弹出的操作使得栈顶的地址增大。

栈在程序的运行中有着举足轻重的作用。最重要的是栈保存了一个函数调用时所需要的维护信息，这常常称之为堆栈帧或者活动记录。堆栈帧一般包含如下几方面的信息：

1. 函数的返回地址和参数
2. 临时变量：包括函数的非静态局部变量以及编译器自动生成的其他临时变量。



栈的模型

基本算法

1．进栈（PUSH）算法

①若 TOP≥n 时，则给出溢出信息，作出错处理（进栈前首先检查栈是否已满，满则溢出；不满则作②）；

②置 TOP=TOP+1（栈指针加 1，指向进栈地址）；

③S(TOP)=X，结束（X 为新进栈的元素）；

2．退栈（POP）算法

①若 TOP≤0，则给出下溢信息，作出错处理(退栈前先检查是否已为空栈， 空则下溢；不空则作②)；

②X=S(TOP)，(退栈后的元素赋给 X)：

③TOP=TOP-1，结束（栈指针减 1，指向栈顶）。

——度娘

[/quote]

Handle 存储结构的实例代码这里也给出，大家自己参照规则分析吧，就不做解说了。

代码 4-13 Handle 存储结构

[codes=galaxy]

```
//=====================================================
//GAWH Index Get System
//=====================================================

//-----------------------------------------------------
//Const
//-----------------------------------------------------

const string libGAWH_gv_is_NameIndexSystemFirstName = "IndexSystem:";
const string libGAWH_gv_is_NameIndexSystemDeleteNum = ":DeleteNum:";
const string libGAWH_gv_is_NameIndexSystemDeleteIndex =
":DeleteIndex:";
const string libGAWH_gv_is_NameIndexSystemMaxNum = ":Max:";
const string libGAWH_gv_is_NameIndexSystemLastIndex = ":LastIndex:";
const string libGAWH_gv_is_NameIndexSystemUsed = ":Used:";

//-----------------------------------------------------
//Functions
//-----------------------------------------------------

int libGAWH_is_IndexSystemGetIndex(string lp_IndexName)
{
    int lv_DeleteNum = DataTableGetInt(true,
libGAWH_gv_is_NameIndexSystemFirstName + lp_IndexName +
libGAWH_gv_is_NameIndexSystemDeleteNum);
    int lv_MaxNum = 0;
    int lv_LastIndex = 0;

    if (lv_DeleteNum == 0)
    {
        lv_MaxNum = DataTableGetInt(true,
libGAWH_gv_is_NameIndexSystemFirstName + lp_IndexName +
libGAWH_gv_is_NameIndexSystemMaxNum);
        DataTableSetInt(true, libGAWH_gv_is_NameIndexSystemFirstName +
lp_IndexName + libGAWH_gv_is_NameIndexSystemMaxNum, lv_MaxNum + 1);
        DataTableSetInt(true, libGAWH_gv_is_NameIndexSystemFirstName +
lp_IndexName + libGAWH_gv_is_NameIndexSystemLastIndex, lv_MaxNum);
        DataTableSetBool(true, libGAWH_gv_is_NameIndexSystemFirstName +
```

```
lp_IndexName + IntToString(lv_MaxNum) +
libGAWH_gv_is_NameIndexSystemUsed, true);
       return lv_MaxNum;
   }
   else
   {
       lv_DeleteNum =lv_DeleteNum - 1;
       DataTableSetInt(true, libGAWH_gv_is_NameIndexSystemFirstName +
lp_IndexName + libGAWH_gv_is_NameIndexSystemDeleteNum, lv_DeleteNum);
       lv_LastIndex = DataTableGetInt(true,
libGAWH_gv_is_NameIndexSystemFirstName + lp_IndexName +
libGAWH_gv_is_NameIndexSystemDeleteIndex + IntToString(lv_DeleteNum));
       DataTableSetBool(true, libGAWH_gv_is_NameIndexSystemFirstName +
lp_IndexName + IntToString(lv_LastIndex) +
libGAWH_gv_is_NameIndexSystemUsed, true);
       DataTableSetInt(true, libGAWH_gv_is_NameIndexSystemFirstName +
lp_IndexName + libGAWH_gv_is_NameIndexSystemLastIndex, lv_LastIndex);
       return lv_LastIndex;
   }
}
void libGAWH_is_IndexSystemDeleteIndex(string lp_IndexName, int
lp_Index)
{
   int lv_DeleteNum = DataTableGetInt(true,
libGAWH_gv_is_NameIndexSystemFirstName + lp_IndexName +
libGAWH_gv_is_NameIndexSystemDeleteNum);
   bool lv_Used = DataTableGetBool(true,
libGAWH_gv_is_NameIndexSystemFirstName + lp_IndexName +
IntToString(lp_Index) + libGAWH_gv_is_NameIndexSystemUsed);
   if (lv_Used)
   {
       DataTableSetInt(true, libGAWH_gv_is_NameIndexSystemFirstName +
lp_IndexName + libGAWH_gv_is_NameIndexSystemDeleteNum, lv_DeleteNum +
1);
       DataTableSetInt(true, libGAWH_gv_is_NameIndexSystemFirstName +
lp_IndexName + libGAWH_gv_is_NameIndexSystemDeleteIndex +
IntToString(lv_DeleteNum), lp_Index);
       DataTableSetBool(true, libGAWH_gv_is_NameIndexSystemFirstName +
lp_IndexName + IntToString(lp_Index) +
libGAWH_gv_is_NameIndexSystemUsed, false);
   }
   else
   {
       TriggerDebugOutput(1,
```

```
StringToText("\xe6\x97\xa0\xe6\xb3\x95\xe9\x87\x8a\xe6\x94\xbe\xe6\x9
c\xaa\xe4\xbd\xbf\xe7\x94\xa8\xe5\xba\x8f\xe5\x8f\xb7\xef\xbc\x9a") +
IntToText(lp_Index), false);
    }
}
void libGAWH_is_IndexSystemFlushIndexRecord(string lp_IndexName)
{
    int lv_I = 0;
    int lv_MaxNum = DataTableGetInt(true,
libGAWH_gv_is_NameIndexSystemFirstName + lp_IndexName +
libGAWH_gv_is_NameIndexSystemMaxNum);
    DataTableSetInt(true, libGAWH_gv_is_NameIndexSystemFirstName +
lp_IndexName + libGAWH_gv_is_NameIndexSystemMaxNum, 0);
    DataTableSetInt(true, libGAWH_gv_is_NameIndexSystemFirstName +
lp_IndexName + libGAWH_gv_is_NameIndexSystemDeleteNum, 0);
    DataTableSetInt(true, libGAWH_gv_is_NameIndexSystemFirstName +
lp_IndexName + libGAWH_gv_is_NameIndexSystemLastIndex, 0);
    while (lv_I < lv_MaxNum)
    {
        DataTableSetBool(true, libGAWH_gv_is_NameIndexSystemFirstName +
lp_IndexName + IntToString(lv_I) + libGAWH_gv_is_NameIndexSystemUsed,
false);
        lv_I += 1;
    }
}
int libGAWH_is_IndexSystemLastUsedIndex(string lp_IndexName)
{
    return DataTableGetInt(true, libGAWH_gv_is_NameIndexSystemFirstName
+ lp_IndexName + libGAWH_gv_is_NameIndexSystemLastIndex);
}
[/codes]
```

至此，Galaxy 的基础知识讲解完毕。想必大家对于整个 Galaxy 有了大体了解。很有可能你看到这里还是什么都不会，那么请先记住大体的语法结构，变量类别即可，从下一章开始，请打开 SC2，我们准备开始实际动手了。

五．算法

算法是什么？

算法是描述如何在脚本中实现某种效果的过程说明。它的意义在于，它告知 SC2 如何执行，以此实现编写者所要达到的效果。它并不仅仅指脚本中的计算，还包括其他实现效果采用的步骤。

当然，为实现某种效果，我们有很多种算法可以采用，那么编写脚本时采用哪种算法和如何更好的用 Galaxy 描述算法就是 SEer 的首要问题。采用的算法的优劣以及实现算法的代码的执行效率是最直观的体现一个 SEer 水平的标志。而这种能力是需要在长期编写 Galaxy 脚本或使用其他编程语言的过程中培养的。

因此从某种角度来说，对于一个使用过其他编程语言的 SEer 来说，Galaxy 是较为简单的，如果你仅仅使用过 Jass，那么，也许你仍然需要学习很多内容。

下面是度娘对于算法的解说。

[quote]

算法（Algorithm）是指解题方案的准确而完整的描述，是一系列解决问题的清晰指令，算法代表着用系统的方法描述解决问题的策略机制。也就是说，能够对一定规范的输入，在有限时间内获得所要求的输出。如果一个算法有缺陷，或不适合于某个问题，执行这个算法将不会解决这个问题。不同的算法可能用不同的时间、空间或效率来完成同样的任务。一个算法的优劣可以用**空间复杂度**与**时间复杂度**来衡量。

一个算法应该具有以下七个重要的特征：

算法可以使用自然语言、伪代码、流程图等多种不同的方法来描述。

1、有穷性（Finiteness）

算法的有穷性是指算法必须能在执行有限个步骤之后终止

2、确切性(Definiteness)

算法的每一步骤必须有确切的定义；

3、输入项(Input)

一个算法有 0 个或多个输入，以刻画运算对象的初始情况，所谓 0 个输入是指算法本身定出了初始条件；

4、输出项(Output)

一个算法有一个或多个输出，以反映对输入数据加工后的结果。没有输出的算法是毫无意义的；

5、可行性(Effectiveness)

算法中执行的任何计算步都是可以被分解为基本的可执行的操作步，即每个计算步都可以在有限时间内完成（也称之为有效性）；

6、 高效性(High efficiency)

执行速度快，占用资源少；

7、 健壮性(Robustness)

对数据响应正确。

——度娘

[/quote]

我们使用 Galaxy 或者 GUI，不可避免的要和算法打交道。然而学习算法更多的是靠自己体悟和积淀。所以大家在看这篇教程时，不仅要记住语法格式，更要逐渐的掌握如何选择、编写、实现自己需要的算法。将算法与你需要制作的效果联系起来。

授人以鱼不如授人以渔，然而授人以渔比授人以鱼的难度大很多。我会尽可能的将代码编写的思路写出来，作为参考。也许我的思路不适合每一个人，希望能起到抛砖引玉的效果。

5.1 通过一个程序来讲解算法

代码 5-1 两条射线构成的角度

[codes=galaxy]

```
fixed libGAWH_cf_AngleRemainderDeal(fixed lp_AngleEnd, fixed
lp_AngleStart)
{
    fixed lv_AngleRemainder = lp_AngleEnd - lp_AngleStart;
    if (lv_AngleRemainder > 180)
    {
        return 360 - lv_AngleRemainder;
    }

    if (lv_AngleRemainder < -180)
    {
        return lv_AngleRemainder + 360;
    }

    return lv_AngleRemainder;
}
```

[/codes]

代码 5-1 的内容是通过两个射线的方向，取得两条射线构成的角度（小于 180 的角度）。请仔细思考下整个程序，你会发现，它其实很简单。

直接计算两条射线的角度差。

[quote]

fixed lv_AngleRemainder = lp_AngleEnd - lp_AngleStart;

//(注，这里隐含两个参数的取值范围为(-180,180])

[/quote]

对计算出的差值进行整理。

代码 5-2 整理计算结果

[codes=galaxy]
```
if (lv_AngleRemainder > 180)
    {
        return 360 - lv_AngleRemainder;
    }

    if (lv_AngleRemainder < -180)
    {
        return lv_AngleRemainder + 360;
    }
```
[/codes]

那么这个算法是怎么考虑出来的？这并不难，请大家跟随我的思路，考虑我们应该采用怎样的计算才能获得需要的结果。

首先，我们来考虑我们已知的参数和需要的结果。

我们现在知道的参数是两条射线的角度，lp_AngleEnd、lp_AngleStart。这两个值的取值范围是(-180,180]，因为一般来说，这个角度值是通过反三角函数 Atan 获取的，所以我们没必要做验证参数是否符合要求。

直接计算角度差 lp_AngleEnd-lp_AngleStart，得到的值范围是（-360，360）。但是这个范围明显不符合我们的需求，其主要原因是对应同一个结果，会有两个不同计算值。如-181°与 179° 在直角坐标系中完全是同一个角度。于是我们需要一个算法将结果范围整理到(-180,180]。

当 lp_AngleEnd-lp_AngleStart 的计算结果在范围(-180,180]时我们无需处理。需要处理的范围就分成了两段。

当 lp_AngleEnd-lp_AngleStart 的计算结果在范围(-360,-180]时，我们应该怎么处理呢？

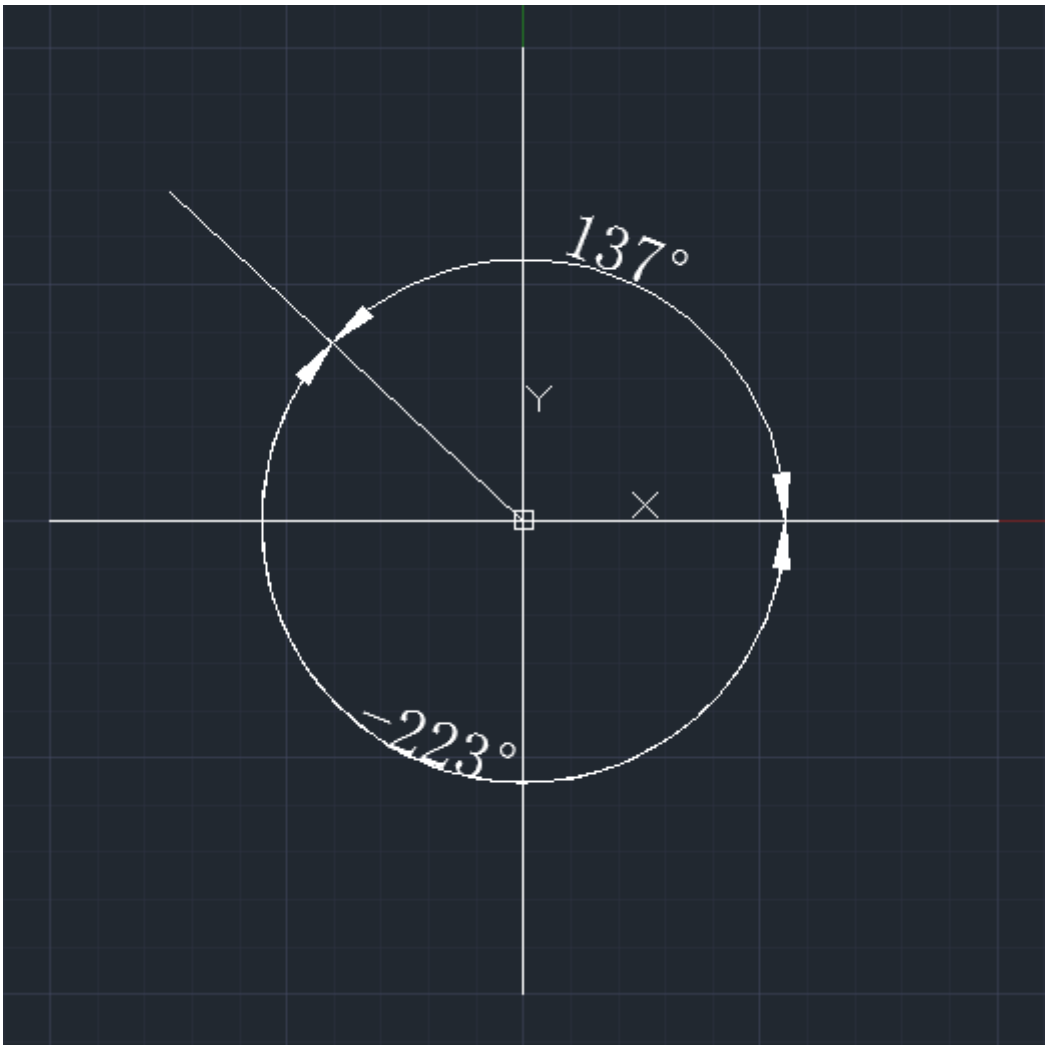


图 5-1

如图 5-1，-223° 可以换为 137° 。由此可知，若 lp_AngleEnd-lp_AngleStar 的值为 X，那么实际结果的计算方法为 360° +X。

当 lp_AngleEnd-lp_AngleStart 的计算结果在范围(180,360)时，我们应该怎么处理呢？

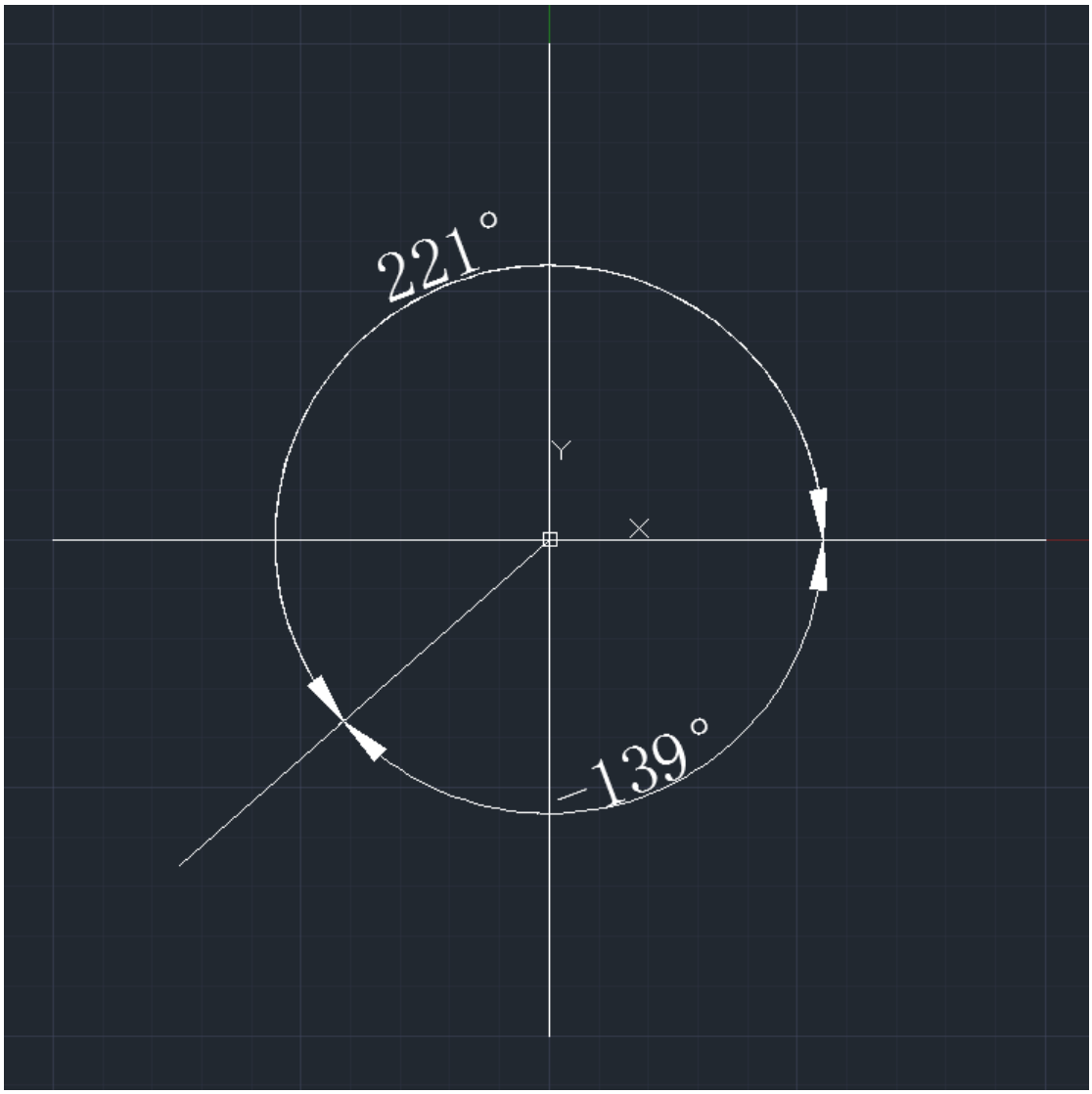


图 5-2

如图 2-2，221° 可以换为-139° 。由此可知，若 lp_AngleEnd-lp_AngleStar 的值为 X，那么实际结果的计算方法为 X-360° 。

依照以上内容，将想好的算法用 Galaxy 描述即可。

5.2 结构化的算法

算法不是简单就能说明的内容。并且不同的编程语言应用的算法基本上相同的。大家可以看一些其他编程语言的算法书，了解一些基本算法，如冒泡排序等等。

算法是实现你所需要效果的手段，我们可以采取的手段并不唯一。当我们想不出实现效果的算法时，也可将思路逆转，寻找能够曲线救国的方法。

要知道算法并没有“最好”的，只有“最符合要求”的，我们很多时候都要在效果和效率中抉择，这种抉择的依据是算法本身的复杂度，执行效率等等，每个人的选择方式不同，编程习惯不同，选择的算法也不同。

但是不论怎样的算法，都是由算法有三种基本结构组成的，算法的三种结构如下：

⑴ 顺序结构。

[quote]

顺序结构的程序设计是最简单的，只要按照解决问题的顺序写出相应的语句就行，它的执行顺序是自上而下，依次执行。

例如；a = 3，b = 5，现交换 a，b 的值，这个问题就好像交换两个杯子水，这当然要用到第三个杯子，假如第三个杯子是 c，那么正确的程序为： c = a； a = b； b = c； 执行结果是 a = 5，b = c = 3 如果改变其顺序，写成：a = b； c = a； b = c； 则执行结果就变成 a = b = c = 5，不能达到预期的目的，初学者最容易犯这种错误。 顺序结构可以独立使用构成一个简单的完整程序，常见的输入、计算，输出三部曲的程序就是顺序结构，例如计算圆的面积，其程序的语句顺序就是输入圆的半径 r，计算 s = 3.14159*r*r,输出圆的面积 s。不过大多数情况下顺序结构都是作为程序的一部分，与其它结构一起构成一个复杂的程序，例如分支结构中的复合语句、循环结构中的循环体等。

——度娘

[/quote]

⑵ 选择结构。

[quote]

选择程序结构用于判断给定的条件，根据判断的结果判断某些条件，根据判断的结果来控制程序的流程。使用选择结构语句时，要用逻辑表达式来描述条件 。

当然各种程序对选择结构语法不一样。例如：

C 语言的选择结构为：

```
if(条件表达式 1)
{语句块 1;}
else if(条件表达式 2)
{语句块 2;}
else if(条件表达式 3)
{语句块 3;}
.
.
.
else
{语句块 n;}
```

VB 中的选择结构为:

```
If(条件表达式 1) then
语句块 1
ElseIf(条件表达式 2) then
语句块 2
```

.
.
.
Else
语句块 n
End If
C 语言中 switch 语句为:
switch(变量或表达式)
{
case 常量表达式 1：
语句块 1；
break;
case 常量表达式 2：
语句块 2；
break;
…….
case 常量表达式 n：
语句块 n；
break;
default： 语句块 n+1
break;
}
VB 语言中 Select Case(相当于 C 的 switch)语句为:
Select Case 变量或表达式
Case 表达式列表 1
语句块 1
Case 常量表达式 2
语句块 2
…….
Case 常量表达式 n：
语句块 n
[Case Else
语句块 n+1]
End Select
}
条件表达式可以分为两类：
关系表达式和逻辑表达式
条件表达式的取值为逻辑值（也称布尔值）：

真(True) 和假(False)
C 用非 0 表示真,0 表示假

——度娘

[/quote]

⑶ 循环结构。

[quote]

循环结构可以减少源程序重复书写的工作量，用来描述重复执行某段算法的问题，这是程序设计中最能发挥计算机特长的程序结构 。

结构简介

循环结构可以看成是一个条件判断语句和一个向回转向语句的组合。另外，循环结构的三个要素：循环变量、循环体和循环终止条件. ，循环结构在程序框图中是利用判断框来表示，判断框内写上条件，两个出口分别对应着条件成立和条件不成立时所执行的不同指令，其中一个要指向循环体，然后再从循环体回到判断框的入口处.

——度娘

[/quote]

一个算法完全可以依据这三种基本结构分解开来，反过来，也就是说我们完全可以通过选择三种基本结构来构成我们需要的算法。节 3.3—3.5 就是依次对应顺序、选择、循环结构算法的语法结构。

在实际应用中，我们首先要决定的是使用否循环结构。因为循环结构涉及的方面较广，临时添加需要做大量修改。如果我们确定需要使用循环，那么我们优先考虑的是我们需要一个什么样的循环，采用什么样的循环条件来控制循环。

选择结构往往是在编写中添加，当我们需要根据不同的选择处理问题时，我们就添加选择结构的算法。

其他内容只要按照顺结构继续写下去即可。

如果我们实现的内容复杂,我们需要将其肢解,分步分块处理,特别是一些常用的功能，可以单独编写函数，这样我们能够更加明确的选择算法，编写出的算法也容易理解（可读性好）。

5.3 计算相关算法讲解

一张地图中 Galaxy 脚本只做两件事：计算和对象处理。这两者之间的联系也有两种：将计算结果赋值给对象的某个属性和依据计算结果不同选择如何处理对象。

一般来说，越复杂的代码，需要的计算越多，特别是做地图上的位置，轨迹有关处理时，需要复杂的解析几何运算，所以学会如何设计和应用运算相关的算法，是十分必要的。

既然是计算相关，那么首先你需要有足够的数学水平来推导公式。关于数学水平的相关问题，不在此教程范围之内，如果你的数学水平不足以完成你需要的算法，那么想办法降低条件或修改条件。编程也需要逆转思维，放开思维的。

这里，我以已知平面内三角形三顶点，求外接圆圆心为例，讲解如何实现运算相关的算法。

看到这个实例，你也许会问，这又不是做数学题，编写这个做什么？其实这是实际问题的抽象。例如如果我们做一个技能，需要确定与某三个单位距离相等的点，在那里创建效果之类的情况时，就需要设计这样一个函数了。

首先，这是一个解析几何的问题，我们需要画图并计算，这里给出示意图，具体计算过程就不一一写出。

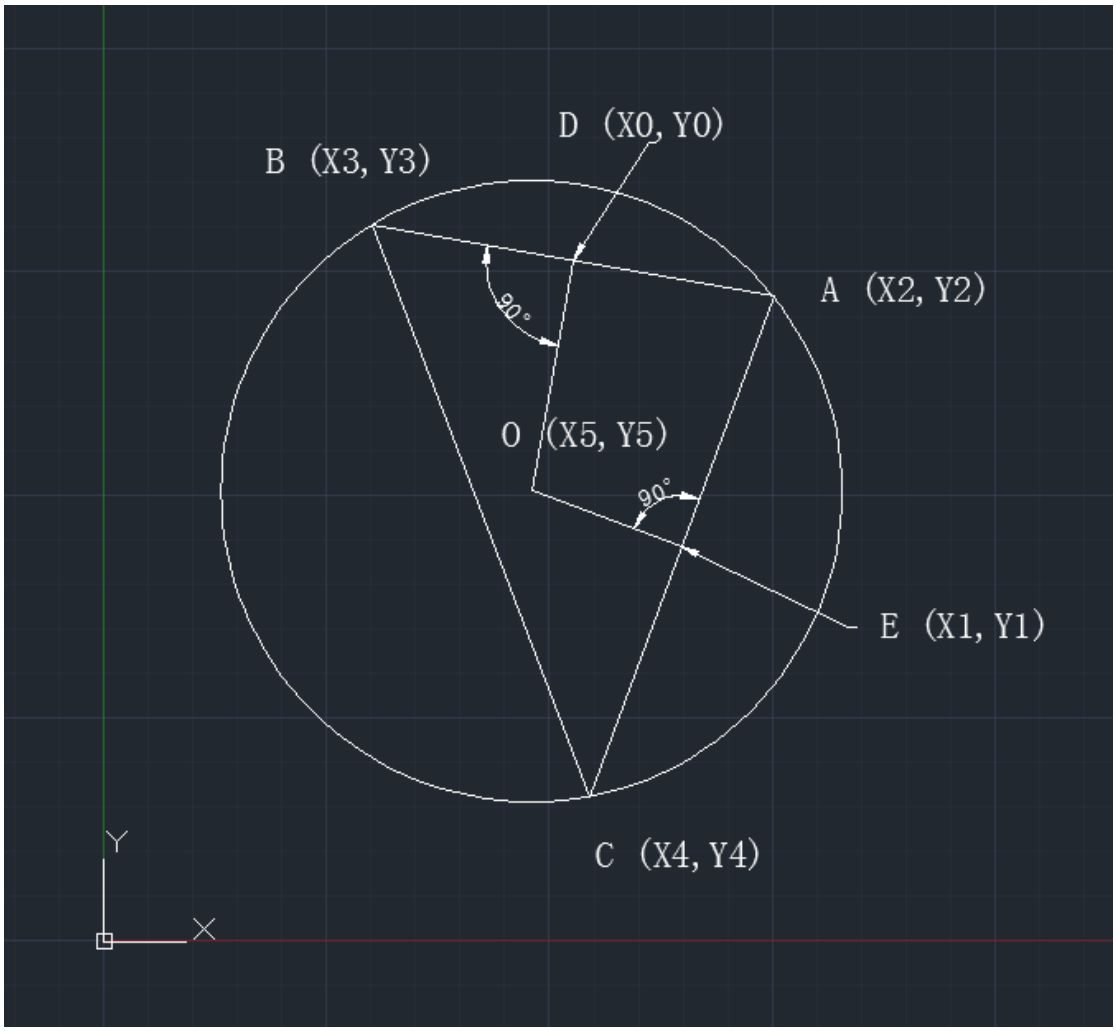


图 5-3 平面内三角形三顶点，求外接圆圆心示意图

已知点为 A、B、C，所求外接圆圆心为 O，OD 与 OE 分别为三角形 ABC 两边 AB 与 AC 的垂直平分线，外切圆圆心就是其交点。我们的算法也是依照通过求取 OD 与 OE 的直线方程，计算交点来编写的。

因为 Galaxy 中 point 变量的两个坐标值都是 fixed 类型的，所以我们可以直接声明两个 fixed 数组来存储对应点的 X 轴、Y 轴坐标。另外，因为我们确定 O 点坐标需要获取 OD 与 OE 的直线方程“Y = K × X + b”，所以还需要两个数组变量，分别存储 K（采用变量名为 AngleOfPerpendicularBisector）和 b。

代码 5-3 所需变量的声明

[codes=galaxy]
```
    fixed [2] lv_AngleOfPerpendicularBisector;
    fixed [2] lv_b;
    fixed [6] lv_X;
    fixed [6] lv_Y;
```
[/codes]

声明完变量后，我们就来写运算过程。(很多情况下，我们在编写过程中逐步添加修改变量，所以并非一次性的全部将变量声明完成。)

对于直线 OD，其直线方程我们可以通过点 D 坐标和 OD 直线方向来获取。D 点坐标很容易求：X0 = （X2 + X3） / 2，Y0 = （Y2 + Y3） / 2。

因为 OD 垂直 AB，所以其直线方程中的 K0 与 AB 的直线方程中的 K 的关系为 K0 = - （1 / K)。而 K 可以通过（Y2 - Y3） / (X2 - X3)来求。直线方程中的 b 通过带入 D 点坐标就可以求，b = Y0 - K0× X0。

至于 OE 也是如此。

综合以上内容我们可以编写出运算过程：

代码 5-4 运算过程

[codes=galaxy]
```
    lv_X[2] = PointGetX(lp_PointA);
    lv_X[3] = PointGetX(lp_PointB);
    lv_X[4] = PointGetX(lp_PointC);
    lv_X[0] = (lv_X[2] + lv_X[3]) / 2;
    lv_X[1] = (lv_X[2] + lv_X[4]) / 2;
    lv_Y[2] = PointGetY(lp_PointA);
    lv_Y[3] = PointGetY(lp_PointB);
    lv_Y[4] = PointGetY(lp_PointC);
    lv_Y[0] = (lv_Y[2] + lv_Y[3]) / 2;
    lv_Y[1] = (lv_Y[2] + lv_Y[4]) / 2;

    lv_AngleOfPerpendicularBisector[0] = -(lv_X[2] - lv_X[3]) / (lv_Y[2]
- lv_Y[3]);
    lv_AngleOfPerpendicularBisector[1] = -(lv_X[2] - lv_X[4])  / (lv_Y[2]
- lv_Y[4]);
    lv_b[0] = lv_Y[0] - lv_AngleOfPerpendicularBisector[0] * lv_X[0];
    lv_b[1] = lv_Y[1] - lv_AngleOfPerpendicularBisector[1] * lv_X[1];
```
[/codes]

其中因为输入的是 lp_PointA、lp_PointB、lp_PointC 三个点，所以为了减少运算消耗，我们将其赋值给 fixed 类型变量 X 和 Y（数组元素）。

最后就是解这个二元一次方程即可。分别得到 O 点坐标的 X5 和 Y5。

代码 5-5 解方程

[codes=galaxy]
```
    lv_X[5] = (lv_b[0] - lv_b[1]) / (lv_AngleOfPerpendicularBisector[1]
- lv_AngleOfPerpendicularBisector[0]);
    lv_Y[5] = lv_AngleOfPerpendicularBisector[0] * lv_X[5] + lv_b[0];
```
[/codes]

这样，计算步骤就写完了，不过因为我们的已知条件是三个 point 类型的点，返回也是。最终函数如代码 5-6。

代码 5-6 平面内三角形三顶点，求外接圆圆心函数

[codes=galaxy]
```
point libGAWH_cf_CircumcircleCenterOfCircle(point lp_PointA, point
lp_PointB, point lp_PointC)
{
    fixed [2] lv_AngleOfPerpendicularBisector;
    fixed [2] lv_b;
    fixed [6] lv_X;
    fixed [6] lv_Y;

    lv_X[2] = PointGetX(lp_PointA);
    lv_X[3] = PointGetX(lp_PointB);
    lv_X[4] = PointGetX(lp_PointC);
    lv_X[0] = (lv_X[2] + lv_X[3]) / 2;
    lv_X[1] = (lv_X[2] + lv_X[4]) / 2;
    lv_Y[2] = PointGetY(lp_PointA);
    lv_Y[3] = PointGetY(lp_PointB);
    lv_Y[4] = PointGetY(lp_PointC);
    lv_Y[0] = (lv_Y[2] + lv_Y[3]) / 2;
    lv_Y[1] = (lv_Y[2] + lv_Y[4]) / 2;

    lv_AngleOfPerpendicularBisector[0] = -(lv_X[2] - lv_X[3]) / (lv_Y[2]
- lv_Y[3]);
    lv_AngleOfPerpendicularBisector[1] = -(lv_X[2] - lv_X[4]) / (lv_Y[2]
- lv_Y[4]);

    lv_b[0] = lv_Y[0] - lv_AngleOfPerpendicularBisector[0] * lv_X[0];
    lv_b[1] = lv_Y[1] - lv_AngleOfPerpendicularBisector[1] * lv_X[1];

    lv_X[5] = (lv_b[0] - lv_b[1]) / (lv_AngleOfPerpendicularBisector[1]
- lv_AngleOfPerpendicularBisector[0]);
    lv_Y[5] = lv_AngleOfPerpendicularBisector[0] * lv_X[5] + lv_b[0];

    return Point(lv_X[5], lv_Y[5]);
}
```

[/codes]

注意到了么，代码5-6其实就是节4.4中代码4-10 MapScript.galaxy中的自定义脚本。对应的实际地图也就是测试这个函数的地图。测试方法很简单：随机生成3个点，输入到函数中去，求出外接圆圆心，然后分别计算这三点的距离，显示出来即可。

注意因为运算误差的关系，实际结果会有微小差别。

经过验证后，这个函数没有问题，那么我们的函数就编写好了。

5.4 面向对象算法讲解

这一节的范畴很大，就不举实例讲解了。需要做什么，大家在GUI下寻找对应动作即可。然后按 ctrl + F11，看对应的函数。这是很方便的寻找函数的方法。否则你也可以在natives.galaxy中寻找。

natives.galaxy 是需要看的，还有其他一些官方基础脚本文件。这些文件的位置在Mods\Core.SC2Mod\Base.SC2Data\TriggerLibs 或Mods\Core.SC2Mod\Base.SC2Data\TriggerLibs\GameData 文件夹中，当然最新版本文件会在Versions文件夹下的最新更新补丁mpq文件中。

（你需要使用MPQ Editor(下载地址:http://www.zezula.net/en/mpq/download.html）来打开MPQ文件。）

我们当然也可以通过打开官方战役地图等来学习如何编写面向对象的算法。

关于面向对象的算法，大家还是多用，慢慢积攒经验吧。

5.5 综合应用

现在，我们用一个实例来讲解整体应用——当你面对一个有着种种要求的系统，改如何着手编写脚本。因为需要完成的内容不同，具体思考方式也不同，编程习惯也因人而异，这里仅以我自己完成的一个系统来抛砖引玉。

我所要完成的内容是，任意多边形区域生成。有个内容可以看如下帖子：

http://bbs.islga.org/read-htm-tid-346762.html

http://bbs.islga.org/read-htm-tid-348761.html

http://bbs.islga.org/read-htm-tid-354509.html

http://bbs.islga.org/read-htm-tid-373495.html

注：此教程内函数不同于我已经发布那个版本，算是一个升级版本吧。在效率上还是已发布版本较高，但是需要预先设定最大顶点数。此外，因采用简化运算，此系统需要保证输入的顶点次序为边的连接次序，且任意两边不能有顶点外交点。

遇到这样一个需要编写的内容，首先要考虑的是输入输出的问题。按照之前的描述，我们可以确定，**我们的输入是一系列点，总数未知；输出为一个复合区域。此外，我们要求这个系统是局域化的（即，同时多次调用不会互相影响）。**

从输入条件来看，因为是总数未知的点，我们不能使用参数来输入点，参考节 4.5 中所述，我们可以采用 DataTable 来存储这一系列点。

从输出条件来看，我们的主要函数的类型需要为 region。

最后，因为此系统为局域化系统，我们需要将所有数据局域化，也就是将 DataTable 存储数据的 Key 在不同调用中区分开。这里我们可以使用节 4.5 中提到的 Handle 存储结构。

考虑到此，我们来依次编写整个系统。

1、 我们需要一个 Index 分配函数，用来获取多边形区域的序号，并有一个与之对应的销毁函数。暂定内容如下：

代码 5-7 Index 分配函数-1

[codes=galaxy]

```
int libGAWH_prc_PolygonInitialization()
{    string lv_PolygonName = libGAWH_gv_prc_PolygonRegion +
IntToString(lv_PolygonIndex);
     DataTableSetInt(true, lv_PolygonName +
libGAWH_gv_prc_PolygonVertex, 0);
     return lv_PolygonIndex;
}
void libGAWH_prc_PolygonRemove(int lp_PolygonIndex)
{
   libGAWH_is_IndexSystemDeleteIndex(libGAWH_gv_prc_PolygonIndexName,
lp_PolygonIndex);
}
```

[/codes]

其中存储在 Key 为“lv_PolygonName + libGAWH_gv_prc_PolygonVertex”的是此多边形的总点数，这里是将其数据清零。

2、 因为不能用参数依次输入顶点，我们还需要一个顶点输入函数:

代码 5-8 顶点输入函数-1

[codes=galaxy]

```
void libGAWH_prc_PolygonRegionAddVertex(int lp_PolygonIndex, point
lp_Vertex)
{
   string lv_PolygonName = libGAWH_gv_prc_PolygonRegion +
IntToString(lp_PolygonIndex);
   int lv_MaxVertexIndex = DataTableGetInt(true, lv_PolygonName +
libGAWH_gv_prc_PolygonVertex);
   fixed lv_X = PointGetX(lp_Vertex);
   fixed lv_Y = PointGetY(lp_Vertex);
   DataTableSetPoint(true, lv_PolygonName +
libGAWH_gv_prc_PolygonVertex + IntToString(lv_MaxVertexIndex),
```

```
lp_Vertex);
    DataTableSetFixed(true, lv_PolygonName +
libGAWH_gv_prc_PolygonVertex + IntToString(lv_MaxVertexIndex) +
libGAWH_gv_prc_PolygonVertexX, lv_X);
    DataTableSetFixed(true, lv_PolygonName +
libGAWH_gv_prc_PolygonVertex + IntToString(lv_MaxVertexIndex) +
libGAWH_gv_prc_PolygonVertexY, lv_Y);
}
```
[/codes]

这里我们分别存储了点和对应的 X、Y 轴坐标。其中点为备用。

3、 下面就是这个多边形创建函数了。在编写函数之前，我们需要先想好如何构成这个多边形区域。前面提到的几个主题中，我提到了相关算法，这里再解说一下：
这种算法的理论依据是**拓补几何**，相关数学原理请自己问度娘。这里直接给出答案：以平面内闭合曲线内部的任意一点为端点，向任意方向做射线，这条射线与闭合曲线的交点数必然为奇数。而多边形的边可以看做是某种闭合曲线，这样，我们可以很容易的区分多边形内部和外部的点。
另外，我们输出结果是一系列的区域集合，一般来说我们都是采用矩形区域。那么我们完全可以将整个多边形看作一幅图片，而这些多边形就是其中的像素。因此，我们可以知道，我们生成的区域与对应多边形的相似度完全由我们作为像素的矩形区域的大小决定，因而我们的多边形创建函数中，除了多边形序号这个参数外，还应该有一个构成区域大小的参数。那么我们可以确定，这个函数的整体框架。

代码 5-9 多边形区域生成函数框架
[codes=galaxy]
```
region libGAWH_prc_PolygonRegionCreate(int lp_PolygonIndex, fixed
lp_RegionPrecision)
{
    region lv_PolygonRegion = RegionEmpty();
    return lv_PolygonRegion;
}
```
[/codes]

4、 这些像素区域的位置是由一个个点确定的，我们采用区域中心来确定，这样，我们就把问题转换成为在多边形区域内找到一系列点的问题了。这些点不能在整个地图区域寻找，这样无效运算太大，因此我们需要用一个区域来确定多边形的范围。因此我们修改顶点输入函数为：

代码 5-10 顶点输入函数-2
[codes=galaxy]
```
void libGAWH_prc_PolygonRegionAddVertex(int lp_PolygonIndex, point
lp_Vertex)
{
```

```
    string lv_PolygonName = libGAWH_gv_prc_PolygonRegion +
IntToString(lp_PolygonIndex);
    int lv_MaxVertexIndex = DataTableGetInt(true, lv_PolygonName +
libGAWH_gv_prc_PolygonVertex);
    fixed lv_X = PointGetX(lp_Vertex);
    fixed lv_Y = PointGetY(lp_Vertex);
    fixed lv_MaxX = DataTableGetFixed(true, lv_PolygonName +
libGAWH_gv_prc_PolygonVertexMax + libGAWH_gv_prc_PolygonVertexX);
    fixed lv_MinX = DataTableGetFixed(true, lv_PolygonName +
libGAWH_gv_prc_PolygonVertexMin + libGAWH_gv_prc_PolygonVertexX);
    fixed lv_MaxY = DataTableGetFixed(true, lv_PolygonName +
libGAWH_gv_prc_PolygonVertexMax + libGAWH_gv_prc_PolygonVertexY);
    fixed lv_MinY = DataTableGetFixed(true, lv_PolygonName +
libGAWH_gv_prc_PolygonVertexMin + libGAWH_gv_prc_PolygonVertexY);

    DataTableSetPoint(true, lv_PolygonName +
libGAWH_gv_prc_PolygonVertex + IntToString(lv_MaxVertexIndex),
lp_Vertex);
    DataTableSetFixed(true, lv_PolygonName +
libGAWH_gv_prc_PolygonVertex + IntToString(lv_MaxVertexIndex) +
libGAWH_gv_prc_PolygonVertexX, lv_X);
    DataTableSetFixed(true, lv_PolygonName +
libGAWH_gv_prc_PolygonVertex + IntToString(lv_MaxVertexIndex) +
libGAWH_gv_prc_PolygonVertexY, lv_Y);

    if (lv_X > lv_MaxX)
    {
        DataTableSetFixed(true, lv_PolygonName +
libGAWH_gv_prc_PolygonVertexMax + libGAWH_gv_prc_PolygonVertexX,
lv_X);
    }
    if (lv_X < lv_MinX)
    {
        DataTableSetFixed(true, lv_PolygonName +
libGAWH_gv_prc_PolygonVertexMin + libGAWH_gv_prc_PolygonVertexX,
lv_X);
    }
    if (lv_Y > lv_MaxY)
    {
        DataTableSetFixed(true, lv_PolygonName +
libGAWH_gv_prc_PolygonVertexMax + libGAWH_gv_prc_PolygonVertexY,
lv_Y);
    }
    if (lv_Y < lv_MinY)
```

```
    {
        DataTableSetFixed(true, lv_PolygonName +
libGAWH_gv_prc_PolygonVertexMin + libGAWH_gv_prc_PolygonVertexY,
lv_Y);
    }

    DataTableSetInt(true, lv_PolygonName + libGAWH_gv_prc_PolygonVertex,
lv_MaxVertexIndex + 1);
}
```

算法很简单，每次添加顶点时比较当前最大最小 X、Y 轴坐标即可。

除此函数外，Index 分配函数也要做些修改，以保证重复使用相同 Index 时不会产生 bug。

代码 5-11 Index 分配函数-2

```
int libGAWH_prc_PolygonInitialization()
{
    int lv_PolygonIndex =
libGAWH_is_IndexSystemGetIndex(libGAWH_gv_prc_PolygonIndexName);
    string lv_PolygonName = libGAWH_gv_prc_PolygonRegion +
IntToString(lv_PolygonIndex);
    DataTableSetFixed(true, lv_PolygonName +
libGAWH_gv_prc_PolygonVertexMin + libGAWH_gv_prc_PolygonVertexX,
99999);
    DataTableSetFixed(true, lv_PolygonName +
libGAWH_gv_prc_PolygonVertexMin + libGAWH_gv_prc_PolygonVertexY,
99999);
    DataTableSetFixed(true, lv_PolygonName +
libGAWH_gv_prc_PolygonVertexMax + libGAWH_gv_prc_PolygonVertexX, -1);
    DataTableSetFixed(true, lv_PolygonName +
libGAWH_gv_prc_PolygonVertexMax + libGAWH_gv_prc_PolygonVertexY, -1);
    DataTableSetInt(true, lv_PolygonName + libGAWH_gv_prc_PolygonVertex,
0);
    return lv_PolygonIndex;
}
```

5、 如果给定顶点小于 3，那么我们返回的区域为空——请注意这一点。

6、 系统编写到这里，开始进入核心部分。现在的首要问题是，我们如何确定这条射线的方向，才能减少我们的计算量，然后，如何用这条射线来寻找多边形内部的点。关于射线方向，一般来说，采用垂直坐标轴的方向是最便于计算的。我们可以很方便的获取这条直线上的点。那么怎么利用这个射线呢？选一个点，然后用射线验证

么？

当然不能如此，这样的计算量太大。我们逆转思维，不通过射线验证，而是先确定一条直线（方向差 180° 的两条射线），然后在直线上选取复合条件的点呢？

我们来仔细考虑一下这个方法。我们首先做一条垂直 X 轴方向的直线，如果直线通过多边形，那么这个直线会与多边形的部分边相交。通过运算可以获得这些交点的坐标，然后以这些点的 Y 轴坐标排序，获得一系列点 P0、P1、P2……。那么，P0 和 P1 点之间的点就在多边形内部，P1 和 P2 之间的点在多边形外部。如图 5-2。依照这种方法，依次画出间隔为参数 lp_RegionPrecision（像素区域边长）的直线，我们就能确定每一个像素区域的位置。

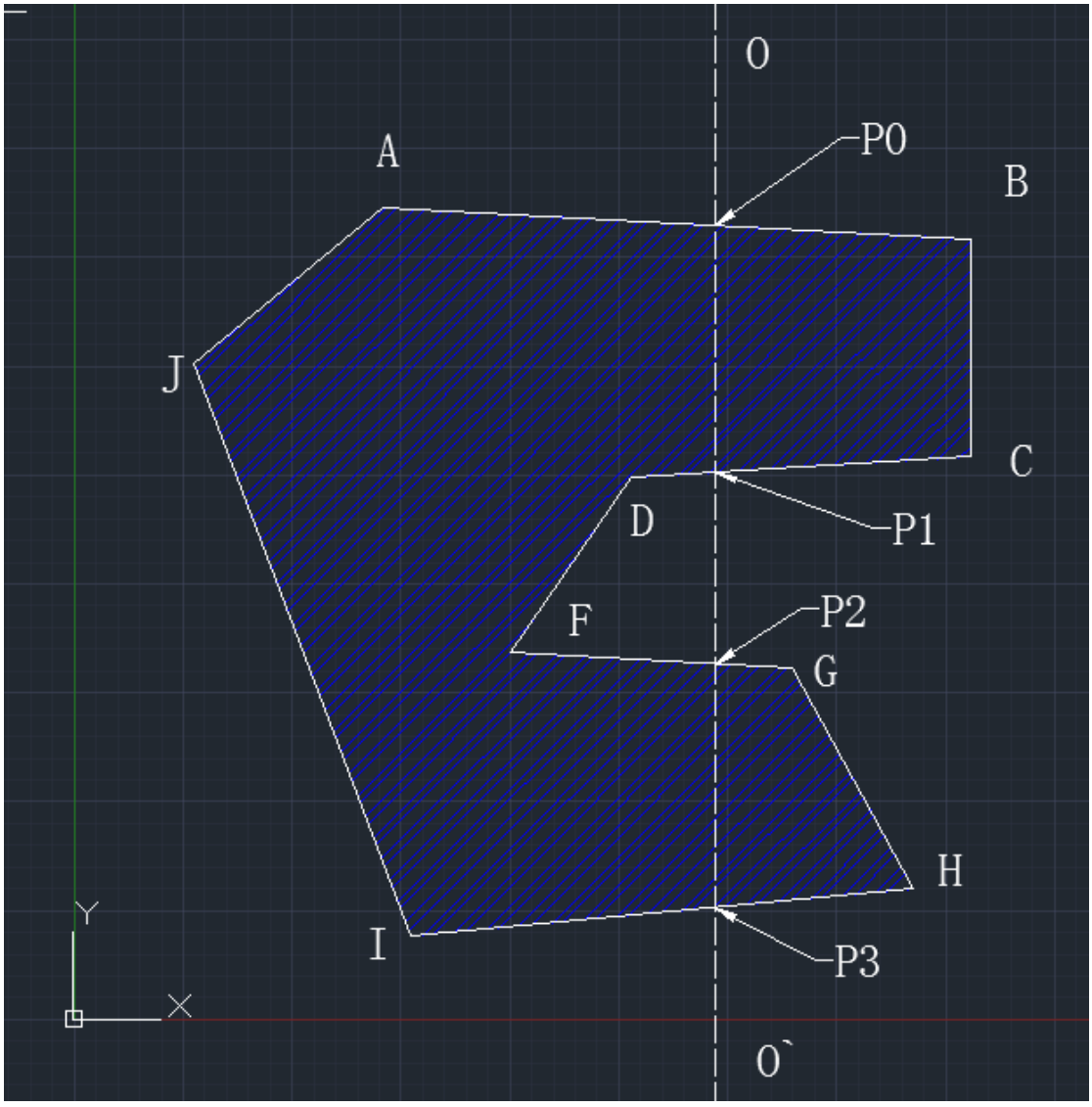


图 5-4 直线与多边形部分边交点

7、 算法的整体是一个循环，循环内容是没一条垂直于 X 轴直线，循环初始值是多边形所在区域的 X 坐标最小值，循环结束值是多边形所在区域的 X 坐标最大值，循环增量是参数 lp_RegionPrecision（像素区域边长）。

我们据此此写出主体循环结构：

代码 5-12 之前描述部分内容及主体循环结构

```
[codes=galaxy]
    string lv_PolygonName = libGAWH_gv_prc_PolygonRegion +
IntToString(lp_PolygonIndex);
    int lv_MaxVertexIndex = DataTableGetInt(true, lv_PolygonName +
libGAWH_gv_prc_PolygonVertex);
    fixed [4] lv_SideValue;
    fixed [2] lv_FirstPoint;
    fixed lv_HalfRegionPrecision = lp_RegionPrecision / 2;
    //Vertex number Check
    if (lv_MaxVertexIndex < 3)
    {
        return lv_PolygonRegion;
    }

    lv_SideValue[0] = DataTableGetFixed(true, lv_PolygonName +
libGAWH_gv_prc_PolygonVertexMax + libGAWH_gv_prc_PolygonVertexX);
    lv_SideValue[1] = DataTableGetFixed(true, lv_PolygonName +
libGAWH_gv_prc_PolygonVertexMin + libGAWH_gv_prc_PolygonVertexX);
    lv_SideValue[2] = DataTableGetFixed(true, lv_PolygonName +
libGAWH_gv_prc_PolygonVertexMax + libGAWH_gv_prc_PolygonVertexY);
    lv_SideValue[3] = DataTableGetFixed(true, lv_PolygonName +
libGAWH_gv_prc_PolygonVertexMin + libGAWH_gv_prc_PolygonVertexY);

    lv_LoopX = lv_SideValue[1];
    lv_LoopX += (((lv_SideValue[0] - lv_SideValue[1]) %
lp_RegionPrecision) / 2);

    while (lv_LoopX < lv_SideValue[0])
    {
        lv_LoopX += lv_HalfRegionPrecision;
        //Codes
        lv_LoopX += lv_HalfRegionPrecision;
    }
[/codes]
```

其中“lv_LoopX += (((lv_SideValue[0] - lv_SideValue[1]) % lp_RegionPrecision) / 2);”语句的作用是将像素区域居中。两次“lv_LoopX += lv_HalfRegionPrecision;”也是相同作用。

8、 接下来就是对于每一条画出的直线的处理。首先我们要找出每条直线与哪些边相交。看图 5-4，与直线 OO`相交的边有什么规律呢？

AB、CD、FG、HI，这是四条相交的边，发现没有，这四条边都满足同一个条件，即两个端点一个在直线左侧、一个在直线右侧。那么，我们用什么样的算法判断呢？

因为已知输入的顶点是有序的，相临两点构成一个边，那么我们对照当前判断的直线的 X 轴坐标值，比较给定参数中相临两个点的 X 坐标值与 lv_LoopX 的关系即可，即(lv_LoopX < MaxF(lv_Side[0], lv_Side[2]) && lv_LoopX > MinF(lv_Side[0], lv_Side[2]))。

然后，我们还要计算一下直线与边交点的坐标，并将 Y 轴坐标值存入 DataTable。其计算公式很简单：“(lv_LoopX - lv_Side[0]) / (lv_Side[0] - lv_Side[2]) * (lv_Side[1] - lv_Side[3]) + lv_Side[1]”

代码 5-13 求取交点

[codes=galaxy]

```
        lv_I = 1;
        lv_SideIntersectNum = 0;
        lv_Side[2] = lv_FirstPoint[0];
        lv_Side[3] = lv_FirstPoint[1];
        while (lv_I <= lv_MaxVertexIndex)
        {
            lv_Side[0] = lv_Side[2];
            lv_Side[1] = lv_Side[3];

            lv_Side[2] = DataTableGetFixed(true, lv_PolygonName +
libGAWH_gv_prc_PolygonVertex + IntToString(lv_I) +
libGAWH_gv_prc_PolygonVertexX);
            lv_Side[3] = DataTableGetFixed(true, lv_PolygonName +
libGAWH_gv_prc_PolygonVertex + IntToString(lv_I) +
libGAWH_gv_prc_PolygonVertexY);

            if (lv_LoopX < MaxF(lv_Side[0], lv_Side[2]) && lv_LoopX >
MinF(lv_Side[0], lv_Side[2]))
            {
                DataTableSetFixed(true, lv_PolygonName +
libGAWH_gv_prc_PolygonVertexY + IntToString(lv_SideIntersectNum),
(lv_LoopX - lv_Side[0]) / (lv_Side[0] - lv_Side[2]) * (lv_Side[1] -
lv_Side[3]) + lv_Side[1]);
                lv_SideIntersectNum += 1;
            }

            lv_I += 1;
        }
```

[/codes]

为了便于循环判断，我们在主循环前做一些补充赋值。

代码 5-14 求取交点的补充赋值

[codes=galaxy]
```
    lv_FirstPoint[0] = DataTableGetFixed(true, lv_PolygonName +
libGAWH_gv_prc_PolygonVertex + IntToString(0) +
libGAWH_gv_prc_PolygonVertexX);
    lv_FirstPoint[1] = DataTableGetFixed(true, lv_PolygonName +
libGAWH_gv_prc_PolygonVertex + IntToString(0) +
libGAWH_gv_prc_PolygonVertexY);
    DataTableSetFixed(true, lv_PolygonName +
libGAWH_gv_prc_PolygonVertex + IntToString(lv_MaxVertexIndex) +
libGAWH_gv_prc_PolygonVertexX, lv_FirstPoint[0]);
    DataTableSetFixed(true, lv_PolygonName +
libGAWH_gv_prc_PolygonVertex + IntToString(lv_MaxVertexIndex) +
libGAWH_gv_prc_PolygonVertexY, lv_FirstPoint[1]);
```
[/codes]

主要内容是将第一点存入最后一点之后。这样我们不需特殊处理最后一点。

9、 下一步的任务是将这些点按照 Y 轴坐标排序，为减少循环，采用交换最小值的方式，（如没有交点则不进行）：

代码 5-15 交点排序

[codes=galaxy]
```
           lv_I = 0;
          while (lv_I < lv_SideIntersectNum)
          {
              lv_J = lv_I;
              lv_TempMinY = 9999;
              while (lv_J < lv_SideIntersectNum)
              {
                  lv_TempY = DataTableGetFixed(true, lv_PolygonName +
libGAWH_gv_prc_PolygonVertexY + IntToString(lv_J));
                  if (lv_TempY < lv_TempMinY)
                  {
                      lv_TempMinY = lv_TempY;
                      lv_TempMinYIndex = lv_J;
                  }
                  lv_J += 1;
              }
              DataTableSetFixed(true, lv_PolygonName +
libGAWH_gv_prc_PolygonVertexY + IntToString(lv_TempMinYIndex),
DataTableGetFixed(true, lv_PolygonName + libGAWH_gv_prc_PolygonVertexY
+ IntToString(lv_I)));
              DataTableSetFixed(true, lv_PolygonName +
libGAWH_gv_prc_PolygonVertexY + IntToString(lv_I), lv_TempMinY);
```

```
            lv_I += 1;
        }
    }
[/codes]
```

10、最后就是按照次序，在序号为 n 和 n+1（n 为偶数）之间，依据参数 lp_RegionPrecision 添加区域了。

完整代码如下：

代码 5-16 任意多边形区域生成系统

```
[codes=galaxy]
//IndexGetSystem
//Const
const string libGAWH_gv_is_NameIndexSystemFirstName = "IndexSystem:";
const string libGAWH_gv_is_NameIndexSystemDeleteNum = ":DeleteNum:";
const string libGAWH_gv_is_NameIndexSystemDeleteIndex =
":DeleteIndex:";
const string libGAWH_gv_is_NameIndexSystemMaxNum = ":Max:";
const string libGAWH_gv_is_NameIndexSystemLastIndex = ":LastIndex:";

//Functions
int libGAWH_is_IndexSystemGetIndex(string lp_IndexName)
{
    int lv_DeleteNum = DataTableGetInt(true,
libGAWH_gv_is_NameIndexSystemFirstName + lp_IndexName +
libGAWH_gv_is_NameIndexSystemDeleteNum);
    int lv_MaxNum = 0;
    int lv_LastIndex = 0;

    if (lv_DeleteNum == 0)
    {
        lv_MaxNum = DataTableGetInt(true,
libGAWH_gv_is_NameIndexSystemFirstName + lp_IndexName +
libGAWH_gv_is_NameIndexSystemMaxNum);
        DataTableSetInt(true, libGAWH_gv_is_NameIndexSystemFirstName +
lp_IndexName + libGAWH_gv_is_NameIndexSystemMaxNum, lv_MaxNum + 1);
        DataTableSetInt(true, libGAWH_gv_is_NameIndexSystemFirstName +
lp_IndexName + libGAWH_gv_is_NameIndexSystemLastIndex, lv_MaxNum);
        return lv_MaxNum;
    }
    else
    {
        lv_DeleteNum =lv_DeleteNum - 1;
```

```
        DataTableSetInt(true, libGAWH_gv_is_NameIndexSystemFirstName +
lp_IndexName + libGAWH_gv_is_NameIndexSystemDeleteNum, lv_DeleteNum);
        lv_LastIndex = DataTableGetInt(true,
libGAWH_gv_is_NameIndexSystemFirstName + lp_IndexName +
libGAWH_gv_is_NameIndexSystemDeleteIndex + IntToString(lv_DeleteNum));
        DataTableSetInt(true, libGAWH_gv_is_NameIndexSystemFirstName +
lp_IndexName + libGAWH_gv_is_NameIndexSystemLastIndex, lv_LastIndex);
        return lv_LastIndex;
    }
}

void libGAWH_is_IndexSystemDeleteIndex(string lp_IndexName, int
lp_Index)
{
    int lv_DeleteNum = DataTableGetInt(true,
libGAWH_gv_is_NameIndexSystemFirstName + lp_IndexName +
libGAWH_gv_is_NameIndexSystemDeleteNum);

    DataTableSetInt(true, libGAWH_gv_is_NameIndexSystemFirstName +
lp_IndexName + libGAWH_gv_is_NameIndexSystemDeleteNum, lv_DeleteNum +
1);
    DataTableSetInt(true, libGAWH_gv_is_NameIndexSystemFirstName +
lp_IndexName + libGAWH_gv_is_NameIndexSystemDeleteIndex +
IntToString(lv_DeleteNum), lp_Index);
}

//Polygon Region Create
//Const
const string libGAWH_gv_prc_PolygonIndexName = "PolygonIndex:";
const string libGAWH_gv_prc_PolygonRegion = "PolygonRegion:";
const string libGAWH_gv_prc_PolygonVertex = ":VerTex:";
const string libGAWH_gv_prc_PolygonVertexX = ":X:";
const string libGAWH_gv_prc_PolygonVertexY = ":Y:";
const string libGAWH_gv_prc_PolygonVertexMax = ":Max:";
const string libGAWH_gv_prc_PolygonVertexMin = ":Min:";
const string libGAWH_gv_prc_PolygonSide = ":Side:";


//Function
int libGAWH_prc_PolygonInitialization()
{
    int lv_PolygonIndex =
libGAWH_is_IndexSystemGetIndex(libGAWH_gv_prc_PolygonIndexName);
    string lv_PolygonName = libGAWH_gv_prc_PolygonRegion +
```

```
IntToString(lv_PolygonIndex);
    DataTableSetFixed(true, lv_PolygonName +
libGAWH_gv_prc_PolygonVertexMin + libGAWH_gv_prc_PolygonVertexX,
99999);
    DataTableSetFixed(true, lv_PolygonName +
libGAWH_gv_prc_PolygonVertexMin + libGAWH_gv_prc_PolygonVertexY,
99999);
    DataTableSetFixed(true, lv_PolygonName +
libGAWH_gv_prc_PolygonVertexMax + libGAWH_gv_prc_PolygonVertexX, -1);
    DataTableSetFixed(true, lv_PolygonName +
libGAWH_gv_prc_PolygonVertexMax + libGAWH_gv_prc_PolygonVertexY, -1);
    DataTableSetInt(true, lv_PolygonName + libGAWH_gv_prc_PolygonVertex,
0);
    return lv_PolygonIndex;
}

void libGAWH_prc_PolygonRemove(int lp_PolygonIndex)
{
    libGAWH_is_IndexSystemDeleteIndex(libGAWH_gv_prc_PolygonIndexName,
lp_PolygonIndex);
}

void libGAWH_prc_PolygonRegionAddVertex(int lp_PolygonIndex, point
lp_Vertex)
{
    string lv_PolygonName = libGAWH_gv_prc_PolygonRegion +
IntToString(lp_PolygonIndex);
    int lv_MaxVertexIndex = DataTableGetInt(true, lv_PolygonName +
libGAWH_gv_prc_PolygonVertex);
    fixed lv_X = PointGetX(lp_Vertex);
    fixed lv_Y = PointGetY(lp_Vertex);
    fixed lv_MaxX = DataTableGetFixed(true, lv_PolygonName +
libGAWH_gv_prc_PolygonVertexMax + libGAWH_gv_prc_PolygonVertexX);
    fixed lv_MinX = DataTableGetFixed(true, lv_PolygonName +
libGAWH_gv_prc_PolygonVertexMin + libGAWH_gv_prc_PolygonVertexX);
    fixed lv_MaxY = DataTableGetFixed(true, lv_PolygonName +
libGAWH_gv_prc_PolygonVertexMax + libGAWH_gv_prc_PolygonVertexY);
    fixed lv_MinY = DataTableGetFixed(true, lv_PolygonName +
libGAWH_gv_prc_PolygonVertexMin + libGAWH_gv_prc_PolygonVertexY);

    DataTableSetPoint(true, lv_PolygonName +
libGAWH_gv_prc_PolygonVertex + IntToString(lv_MaxVertexIndex),
lp_Vertex);
    DataTableSetFixed(true, lv_PolygonName +
```

```
libGAWH_gv_prc_PolygonVertex + IntToString(lv_MaxVertexIndex) +
libGAWH_gv_prc_PolygonVertexX, lv_X);
   DataTableSetFixed(true, lv_PolygonName +
libGAWH_gv_prc_PolygonVertex + IntToString(lv_MaxVertexIndex) +
libGAWH_gv_prc_PolygonVertexY, lv_Y);

   if (lv_X > lv_MaxX)
   {
      DataTableSetFixed(true, lv_PolygonName +
libGAWH_gv_prc_PolygonVertexMax + libGAWH_gv_prc_PolygonVertexX,
lv_X);
   }
   if (lv_X < lv_MinX)
   {
      DataTableSetFixed(true, lv_PolygonName +
libGAWH_gv_prc_PolygonVertexMin + libGAWH_gv_prc_PolygonVertexX,
lv_X);
   }
   if (lv_Y > lv_MaxY)
   {
      DataTableSetFixed(true, lv_PolygonName +
libGAWH_gv_prc_PolygonVertexMax + libGAWH_gv_prc_PolygonVertexY,
lv_Y);
   }
   if (lv_Y < lv_MinY)
   {
      DataTableSetFixed(true, lv_PolygonName +
libGAWH_gv_prc_PolygonVertexMin + libGAWH_gv_prc_PolygonVertexY,
lv_Y);
   }

   DataTableSetInt(true, lv_PolygonName + libGAWH_gv_prc_PolygonVertex,
lv_MaxVertexIndex + 1);

region libGAWH_prc_PolygonRegionCreate(int lp_PolygonIndex, fixed
lp_RegionPrecision)
{
   string lv_PolygonName = libGAWH_gv_prc_PolygonRegion +
IntToString(lp_PolygonIndex);
   region lv_PolygonRegion = RegionEmpty();
   int lv_MaxVertexIndex = DataTableGetInt(true, lv_PolygonName +
libGAWH_gv_prc_PolygonVertex);
   int lv_I = 0;
   int lv_J = 0;
```

```
    int lv_SideIntersectNum = 0;
    fixed lv_LoopX = 0;
    fixed lv_LoopY = 0;
    fixed [4] lv_SideValue;
    fixed lv_HalfRegionPrecision = lp_RegionPrecision / 2;
    fixed [4] lv_Side;
    fixed [2] lv_FirstPoint;
    fixed lv_TempMinY = 0;
    int lv_TempMinYIndex = 0;
    fixed lv_TempY = 0;
    fixed lv_TempNextY = 0;

    //Vertex number Check
    if (lv_MaxVertexIndex < 3)
    {
        return lv_PolygonRegion;
    }

    lv_SideValue[0] = DataTableGetFixed(true, lv_PolygonName +
libGAWH_gv_prc_PolygonVertexMax + libGAWH_gv_prc_PolygonVertexX);
    lv_SideValue[1] = DataTableGetFixed(true, lv_PolygonName +
libGAWH_gv_prc_PolygonVertexMin + libGAWH_gv_prc_PolygonVertexX);
    lv_SideValue[2] = DataTableGetFixed(true, lv_PolygonName +
libGAWH_gv_prc_PolygonVertexMax + libGAWH_gv_prc_PolygonVertexY);
    lv_SideValue[3] = DataTableGetFixed(true, lv_PolygonName +
libGAWH_gv_prc_PolygonVertexMin + libGAWH_gv_prc_PolygonVertexY);

    lv_FirstPoint[0] = DataTableGetFixed(true, lv_PolygonName +
libGAWH_gv_prc_PolygonVertex + IntToString(0) +
libGAWH_gv_prc_PolygonVertexX);
    lv_FirstPoint[1] = DataTableGetFixed(true, lv_PolygonName +
libGAWH_gv_prc_PolygonVertex + IntToString(0) +
libGAWH_gv_prc_PolygonVertexY);
    DataTableSetFixed(true, lv_PolygonName +
libGAWH_gv_prc_PolygonVertex + IntToString(lv_MaxVertexIndex) +
libGAWH_gv_prc_PolygonVertexX, lv_FirstPoint[0]);
    DataTableSetFixed(true, lv_PolygonName +
libGAWH_gv_prc_PolygonVertex + IntToString(lv_MaxVertexIndex) +
libGAWH_gv_prc_PolygonVertexY, lv_FirstPoint[1]);

    lv_LoopX = lv_SideValue[1];
    lv_LoopX += (((lv_SideValue[0] - lv_SideValue[1]) %
lp_RegionPrecision) / 2);
```

```
    while (lv_LoopX < lv_SideValue[0])
    {
        lv_LoopX += lv_HalfRegionPrecision;

        lv_I = 1;
        lv_SideIntersectNum = 0;
        lv_Side[2] = lv_FirstPoint[0];
        lv_Side[3] = lv_FirstPoint[1];
        while (lv_I <= lv_MaxVertexIndex)
        {
            lv_Side[0] = lv_Side[2];
            lv_Side[1] = lv_Side[3];

            lv_Side[2] = DataTableGetFixed(true, lv_PolygonName +
libGAWH_gv_prc_PolygonVertex + IntToString(lv_I) +
libGAWH_gv_prc_PolygonVertexX);
            lv_Side[3] = DataTableGetFixed(true, lv_PolygonName +
libGAWH_gv_prc_PolygonVertex + IntToString(lv_I) +
libGAWH_gv_prc_PolygonVertexY);

            if (lv_LoopX < MaxF(lv_Side[0], lv_Side[2]) && lv_LoopX >
MinF(lv_Side[0], lv_Side[2]))
            {
                DataTableSetFixed(true, lv_PolygonName +
libGAWH_gv_prc_PolygonVertexY + IntToString(lv_SideIntersectNum),
(lv_LoopX - lv_Side[0]) / (lv_Side[0] - lv_Side[2]) * (lv_Side[1] -
lv_Side[3]) + lv_Side[1]);
                lv_SideIntersectNum += 1;
            }

            lv_I += 1;
        }

        if (lv_SideIntersectNum != 0)
        {
            lv_I = 0;
            while (lv_I < lv_SideIntersectNum)
            {
                lv_J = lv_I;
                lv_TempMinY = 9999;
                while (lv_J < lv_SideIntersectNum)
                {
                    lv_TempY = DataTableGetFixed(true, lv_PolygonName +
libGAWH_gv_prc_PolygonVertexY + IntToString(lv_J));
```

```
                if (lv_TempY < lv_TempMinY)
                {
                    lv_TempMinY = lv_TempY;
                    lv_TempMinYIndex = lv_J;
                }
                lv_J += 1;
            }
            DataTableSetFixed(true, lv_PolygonName +
libGAWH_gv_prc_PolygonVertexY + IntToString(lv_TempMinYIndex),
DataTableGetFixed(true, lv_PolygonName + libGAWH_gv_prc_PolygonVertexY
+ IntToString(lv_I)));
            DataTableSetFixed(true, lv_PolygonName +
libGAWH_gv_prc_PolygonVertexY + IntToString(lv_I), lv_TempMinY);
            lv_I += 1;
        }

        lv_LoopY = -1;
        lv_TempNextY = DataTableGetFixed(true, lv_PolygonName +
libGAWH_gv_prc_PolygonVertexY + IntToString(lv_SideIntersectNum));
        while (true)
        {
            if(lv_LoopY < lv_TempNextY)
            {
                if (lv_SideIntersectNum <= 0)
                {
                    break;
                }

                lv_SideIntersectNum -= 2;
                lv_LoopY = DataTableGetFixed(true, lv_PolygonName +
libGAWH_gv_prc_PolygonVertexY + IntToString(lv_SideIntersectNum + 1));
                lv_TempNextY = DataTableGetFixed(true, lv_PolygonName
+ libGAWH_gv_prc_PolygonVertexY + IntToString(lv_SideIntersectNum));
                lv_LoopY -= (((lv_LoopY - lv_TempNextY) %
lp_RegionPrecision) / 2);
            }

            RegionAddRect(lv_PolygonRegion, true, lv_LoopX -
lv_HalfRegionPrecision, lv_LoopX + lv_HalfRegionPrecision, lv_LoopY,
lv_LoopY + lp_RegionPrecision);
            lv_LoopY -= lp_RegionPrecision;
        }
    }
```

```
        lv_LoopX += lv_HalfRegionPrecision;
    }
    return lv_PolygonRegion;
}
```
[/codes]

PS:关于区域叠加，有如下注意事项：

- 区域叠加 256*256 测试通过，即 65536 块区域叠加。
- 负区域叠加后效果无法取消。
- 可以用添加区域到空区域的方式复制区域。
- 正负叠加的复杂区域内的随机点获取成功概率很高。共计 32 次测试，测试失败报错“經 32 嘗試，無法找到'未知地區'地區內的隨機點”。

此例中面向对象部分较少。在某些系统中则会需要很多面向对象的算法，该如何做呢？

不论是怎样的系统，或者单一函数，在编写之前，我们首先要想好输入输出条件。实际上很多时候，只要略微修改输入输出条件，便能将你的算法的执行效率提高很多。

一个系统也好，一个函数也好，其本身应该是封装的、独立的，输入输出更应该是标准化的。例如之前的例子里，所有的 DataTable 的存储 Key 都用常量存储，而且即使是不同的系统，Key 的结构也是近似的。这样，当你制作新的内容时，你可以方便的利用自己以前写好的函数。

运算和对象之间的联系其实要比想象中简单的多。当你确定输入输出条件后，下一步就是选择对应的对象处理函数，如上例中的添加区域函数“RegionAddRect()”，当你确定好这一步，剩下就是考虑对应每种情况应有的运算结果。

在编写过程中，如果不是确定必然只会单次使用，一般都尽量保证局域化。若是仅区别不同玩家局域化的系统，可以用长度 16 的数组来代替 DataTable，因为尽管 DataTable 的执行效率很高，但是过多次的连接字符串也会影响效率。

说起效率，如果你有足够水平，那么请在编写过程中就保证执行效率，从每一个细节做起。当你好不在意效率的将函数编写完，运行时突然发现执行起来卡的时候，你会相当郁闷的。要知道，优化一个函数比重写还要困难。

5.6 Debug 方法

Debug 的水平决定你的成品率和开发周期。关于这方面的讲解很难。原因同样是因人而异。

一般来说我们都是采用验证关键值的方式来 Debug。不在循环中的值可以赋值给全局变量，然后在 Debug 窗口中读取。通用的方法是用 TriggerDebugOutput()这个函数来输出基础变量类型及 Text 等类型的值。

SE 中不支持步进，但我们可以自己编写步进脚本。

代码 5-17 步进脚本

[codes=galaxy]
```
bool Next = true;
void BreakOut(string ShowText)
```

```
{
    TriggerDebugOutput(1, StringToText(ShowText), false);
    while(Next)
    {
        Wait(0.01,c_timeGame);
    }
    Next = true;
}
[/codes]
```

使用时在需要打断处调用函数“BreakOut();”即可设置断点，输入 Text 参数为打断时输出信息，用于输出 Debug 数据。

通过触发设置全局变量 Next = false;就可以使函数继续执行。

5.7 GUI 下的自定义函数、事件、条件、动作以及库的制作

SE 支持自定义函数、事件、条件、动作。如图 5-5、5-6。

图 5-5 可新建内容

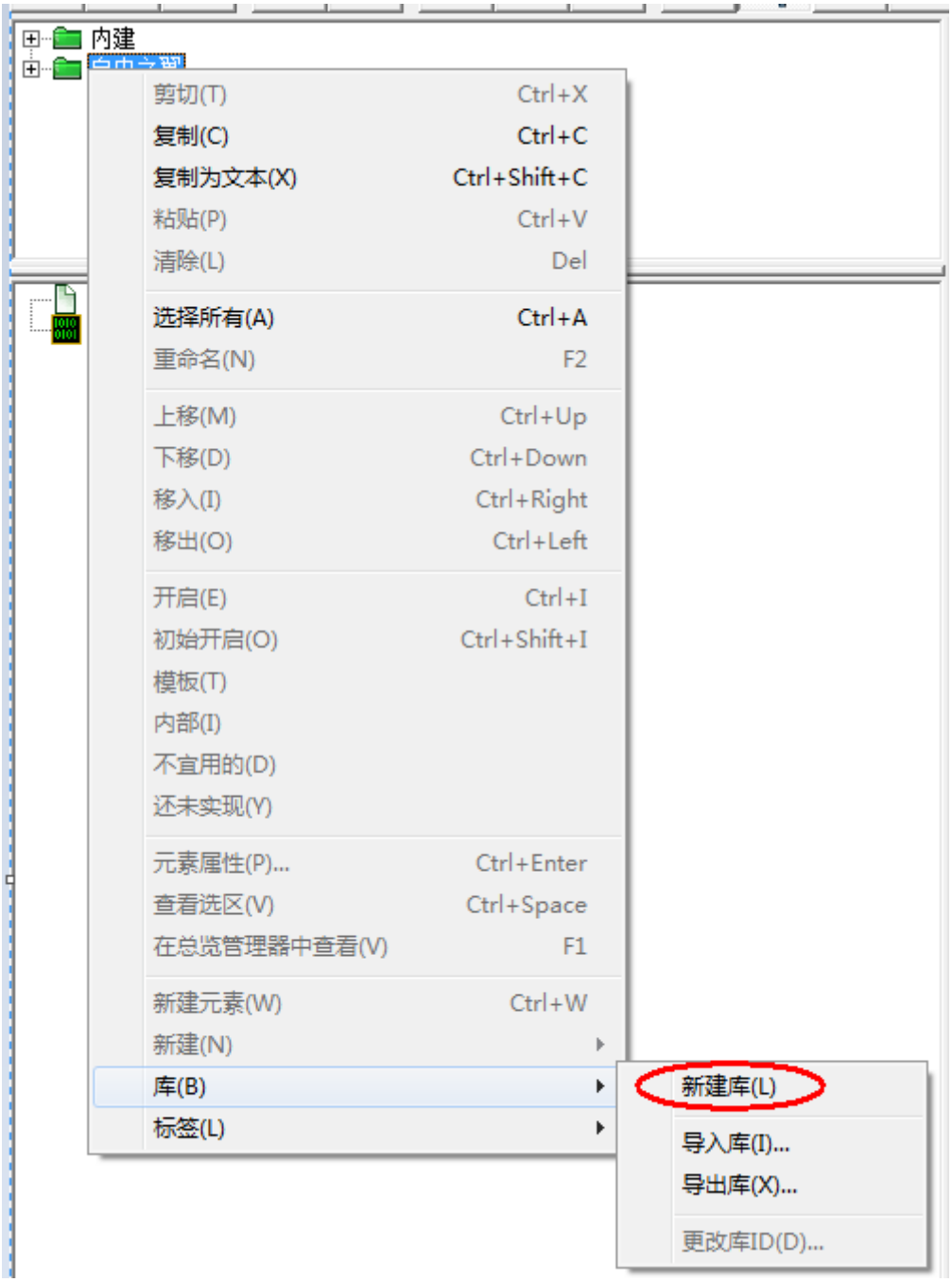


图 5-6 新建库

事件、条件、动作、函数的自定义实质上都是相同的内容，如图 5-7。

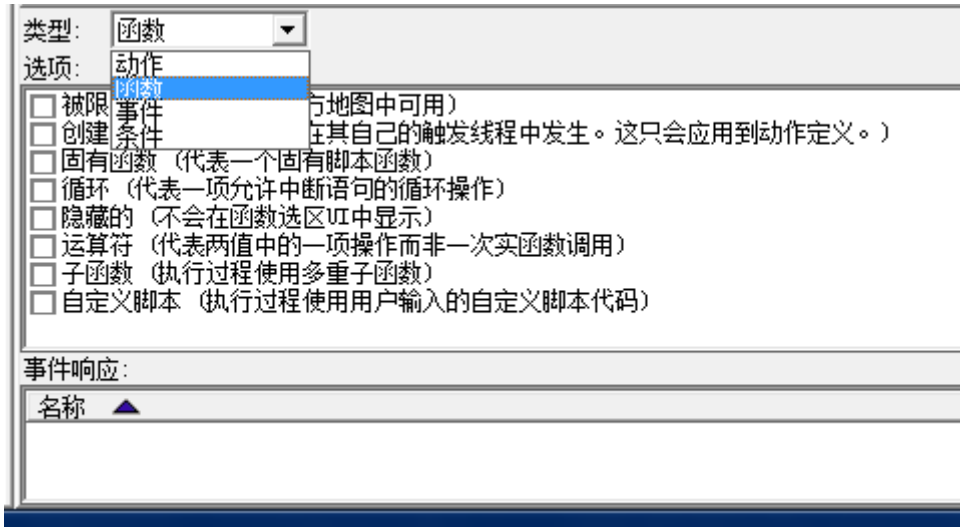


图 5-7 事件、条件、动作、函数

关于图 5-7 中每个选项选项的含义，大家参照基本库中的使用与自己测试来了解好了。个人能力有限，暂不做逐一说明。

使用自定义内容（不包括自定义脚本），可以使用几个宏，其内容如下：

[quote]
#AUTOVAR 用来创建随机不重名的变量。第二个参数代表类型，可以忽略。类型可以由别的东西决定，比如函数的参数。
#SUBFUNCS 用来代表 sub-function 里的内容。具体哪一行由关键字决定
#IFHAVESUBFUNCS 判断是否有填写 sub-function
#CUSTOM 用户所输入的自定义代码
#PARAM 用来取函数的参数
#SMARTBREAK 就是，当外部有循环结构的时候写 break，反之不写。
#DEFRETURN 代表外部函数的默认返回值
[/quote]

可参看：http://bbs.islga.org/read-htm-tid-177329.html 中头目的说明。详细使用方式及其他未说明宏可自行在基本库中寻找。

代码 5-18 一个使用宏的自定义脚本实例
[codes=galaxy]
```
    #AUTOVAR(I,int) = 0;
    #AUTOVAR(TotalSquaresNum,int) = 0;
    #AUTOVAR(NameLabyrinthNum,string) = "";
    #AUTOVAR(NameLabyrinthNum) = libGAWH_gv_lc_NameLabyrinthFirstName +
IntToString(#PARAM(LabyrinthNum));
    #AUTOVAR(TotalSquaresNum) = DataTableGetInt(true,
#AUTOVAR(NameLabyrinthNum) +
libGAWH_gv_lc_NameLabyrinthTotalSquaresNum);
```

```
    while(#AUTOVAR(I) <= #AUTOVAR(TotalSquaresNum))
    {
        #PARAM(SquarePoint) = DataTableGetPoint(true,
((#AUTOVAR(NameLabyrinthNum) +
libGAWH_gv_lc_NameLabyrinthTotalSquaresNum) + IntToString(#AUTOVAR(I)))
+ libGAWH_gv_lc_NameLabyrinthSquarePoint);
        #SUBFUNCS(LabyrinthSquaresCreateFunction)
        #AUTOVAR(I) = #AUTOVAR(I) + 1;
    }
```
[/codes]

对应的函数设置见图 5-8。

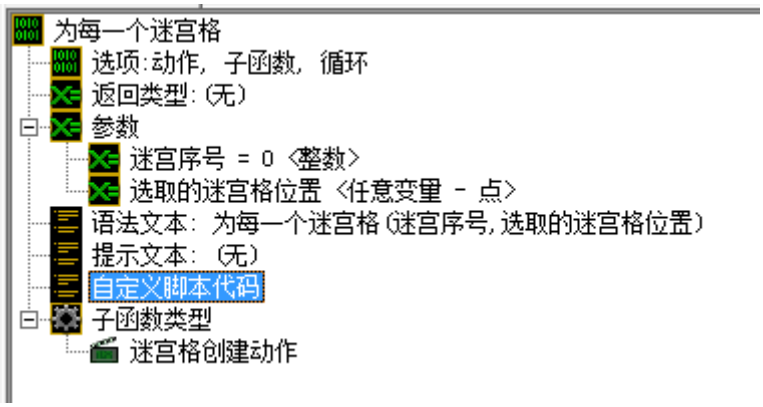


图 5-8 函数设置

下面来对代码 5-18 做一下解析。

[quote]

#AUTOVAR(I,int) = 0;是声明一个整数局域变量 I。

#AUTOVAR(TotalSquaresNum,int) = 0;是声明一个整数局域变量 TotalSquaresNum。

#AUTOVAR(NameLabyrinthNum,string) = "";是声明一个字符串局域变量 NameLabyrinthNum。

#AUTOVAR(NameLabyrinthNum) = libGAWH_gv_lc_NameLabyrinthFirstName + IntToString(#PARAM(LabyrinthNum));对字符串变量 NameLabyrinthNum 赋值，其值为 libGAWH_gv_lc_NameLabyrinthFirstName 加上整数参数 LabyrinthNum 转换为字符串后的值。

#AUTOVAR(TotalSquaresNum) = DataTableGetInt(true, #AUTOVAR(NameLabyrinthNum) + libGAWH_gv_lc_NameLabyrinthTotalSquaresNum); 对整数局域变量 TotalSquaresNum 赋值，其值为 DataTable 中 Key 为 NameLabyrinthNum 加上

libGAWH_gv_lc_NameLabyrinthTotalSquaresNum 对应项的整数存储值。

while(#AUTOVAR(I) <= #AUTOVAR(TotalSquaresNum))循环，循环条件为局域变量 I 小于等于整数局域变量 TotalSquaresNum。

#PARAM(SquarePoint) = DataTableGetPoint(true, ((#AUTOVAR(NameLabyrinthNum) + libGAWH_gv_lc_NameLabyrinthTotalSquaresNum) + IntToString(#AUTOVAR(I))) + libGAWH_gv_lc_NameLabyrinthSquarePoint);对参数 SquarePoint 赋值，值为 DataTable 中 Key 为指定值（略）对应的点（point）类型值。

#SUBFUNCS(LabyrinthSquaresCreateFunction) 执行循环动作中的代码。LabyrinthSquaresCreateFunction 为图 5-8“迷宫格创建动作”下添加的代码。此内容将会在使用此函数时添加。

#AUTOVAR(I) = #AUTOVAR(I) + 1;整数变量 I 自增 1.
[/quote]

大家可以看出，实际上宏的使用与 Galaxy 的原本语法完全相同，仅#SUBFUNCS 宏的使用方法较为特别。相信大家能够很轻松的学会宏的使用。

5.8 良好的编程习惯

这章内容仅供参考，你可以自己确立自己的编程习惯。

不论哪种编程语言的教程都会提到一点，那就是良好的编程习惯，这种好的习惯也许并不能影响你编写内容的执行效果，但是它会增加你的代码的可读性。使你在修改或应用以前编写的代码时能够更清晰、快捷的回忆起你当时的设计。如果你分享了自己的代码，同样也便于大家阅读。

良好的编程习惯包括三个方面：
1、注释。
2、规范的变量名及函数名。
3、明确的缩进及空行。
4、常量。

我们依次说明。

一段复杂的代码经常包含复杂的逻辑关系及处理步骤，当我们阅读这样一段代码时，往往会无从下手，或者对其中的内容不解。编程过程中，在函数关键点写下注释是相当好的习惯。这样不仅仅便于其他人阅读你的代码，也便于你自己对代码的修改、完善。
因为 Galaxy 不支持中文注释，所以大家还是采用英文注释吧。或者干脆拼音？

关于变量名及函数名，这个就是更为细节的内容。类似注释，它也是增加代码的可读性

的。如果你有加密脚本的意图，除非你是手动在编写的同时加密，否则还是建议你使用规范化的变量名及函数名。

这种规范的命名规则有那么几种，学编程的同学应该了解。具体大家可以自己问度娘，大家也可以根据自己的习惯来处理，或者参考 Blizzard 的官方脚本中的命名方法。

我自己的命名规则如下：

局域变量采用 **lv_变量名**，如 lv_I；

参数使用 **lp_参数名**，如 lp_Index；

全局变量采用 **库名_gv_功能名缩写_变量名** 的方式，如 libGAWH_gv_cf_ArraySystemFirstName；

函数名采用**库名_功能名缩写_函数名**的方式，如 libGAWH_cf_ArraySystemSetInt。

明确的缩进和空行便于你确定“{}”的配对，以及选择与循环语句嵌套时的逻辑关系。如果你使用 Galaxy++ Editor，软件会自动处理。如果你直接使用 SE 编程，那么请自己注意添加空格。这里我的语法格式不同于官方文件中的，个人感觉我这样的配对方式能容易体现出“{}”的配对。

一些重复使用的值，例如用于区分 DataTable 存储空间的 Key 的前置名等，最好使用常量，这样方便你在遇到冲突时的修改。