# 目录


一． Thyme 语言介绍.....2
二． Algodoo 控制台.....2
1. 控制台是什么.....2
2. 如何打开控制台.....2
3. 控制台操作按键.....2
三． Thyme 语法.....3
1. 基础部分.....3
(1) 变量.....3
(2) 运算符.....4
(3) If 条件判断.....5
(4) For 循环.....6
(5) 方法(函数).....7
(6) 类.....9
(7) 标识符.....11
(8) 结构.....11
(9) 数组.....12
(10) 指针.....14
(11) .phz 和.phn 文件.....14
(12) Thyme 额外知识点.....14
2. 进阶部分.....16
(1) 对象指针(此功能 BUG 太多故暂不包含例子).....16
(2) 扩展型 for.....16
(3) 更新链接(自身变量锁定).....17
(4) 数组元素追加.....17
(5) 局部变量声明.....17
(6) 对象评估.....18
例：三种读取文件的方式.....19
四． 物体的脚本选单.....19
(1) 只读属性.....19
(2) 可编辑属性.....19
(3) 自定义可编辑属性.....25
五． 物件的材质.....25
1. 材质贴图(.png 文件).....26
2. 属性文件(.cfg 文件，需要使用非 word 方式打开).....26
六． 场景的样式.....26
七． 启动自读取.....27
八． 脚本库的使用方法.....27
九． 如何书写一个自己的脚本库.....28
十. 总结.....28

# 1. Thyme 语言介绍

Thyme 语言是专门为 Algodoo 设计的内置编程语言。其本身具有的简单易用的特点给用户提供了极其方便的编程环境，可以让用户快速上手使用。同时，Thyme 语言也是非常强大的，在 Algodoo 中用好了 Thyme 语言，可以帮助您做出更多更有趣的场景，但它不仅仅局限于此，因 Algodoo 本身是由 C++制作的，如果可以找到并借用外部的接口，那么 Algodoo 将会拥有属于自己的额外内容包。

本教程主要针对 Thyme 语言进行讲解，不会涉及到 C++，如果想要学习 C++的话，可以去百度上查找 C++相关教程自行学习。

# 2. Algodoo 控制台

## 1. 控制台是什么

Algodoo 控制台是一个工作在后台的信息处理终端，在 Algodoo 所做的大部分操作都会显示在控制台中。同时，控制台也是一个可以进行代码编辑的地方，用户在场景中所编辑的所有代码都能在控制台中找到，当然，控制台中所做的大部分修改也能在场景中体现出来。

总之，控制台是一个编程学习的好地方。

## 2. 如何打开控制台

在 Algodoo 打开后我们会进到一个场景里，这时我们只要按下“ESC”键下面的“~”键或者 F10 键，就会从屏幕上方弹出一个黑色窗口，这代表着我们已经进入到控制台中了，同样的，再次按下会隐藏控制台。

## 3. 控制台操作按键

因为 Algodoo 的控制台不能像别的窗口一样随便查看，下面将简单介绍几个控制台使用的按键：

“←”“→”方向键：可以在控制台中最后一个“>”下方所在的可编辑输入框中调整光标位置

“↑”“↓”方向键：可以让执行过的代码再此出现到输入框中，以便于再次手动执行

“tab”键：当输入任意字符时，按下这个键即可自动补全完整代码

“home”“end”键：可以让光标快速定位到该行的最前和最后

“page up”“page down”键：可以让控制台的显示上翻和下翻

# 3. Thyme 语法

## 1. 基础部分

### (1) 变量

#### A. 变量声明

变量是编程语言中最重要的一部分，声明一个变量首先要有变量名，这里举几个例子：
Variable1 = 1
Variable2 = true
Variable3 = 2.41
Variable4 = ”Var”
上述四个均是变量声明，Variable1~Variable4 均是变量名，赋值符号后面的则是该变量的值，这样做完之后，一个变量就声明好了，接下来怎么用，就要看自己的需要了。

注意：Thyme 语言的变量命名规则：变量通常只能由英文、数字和下划线组成，且数字不能放在变量名的最前面（这个很重要！！！这个很重要！！！这个很重要！！！），如果是中文变量的话，需要在变量后方加一个空格再进行赋值，其余条件同上。

#### B. 变量类型

①整数变量
　　整数变量正如其名，这里的变量使用的就是数学上定义的整数，比如 Variable = 1。
②浮点数变量
　　浮点数又称作小数，数学上所涉及到的全部小数包括有理数和无理数都可以作为这个变量的值，比如 Variable = 61.6782。
③字符串变量
　　字符，顾名思义，就是我们平常用到的每个汉字，每个英文字母，甚至是数字，特殊符号。字符串则是将这些字符连接起来后的一个完整的词语，短语或句子，在这里，字符串变量的声明需要有一对双引号（“”）来引起一个字符或一个字符串，并将这个赋给变量，比如 Variable=”a 啊 12%……！”。
④布尔变量
　　布尔值是由英国的数学家乔治·布尔发明的，它只有两种状态，即真(true)和假(false)，在声明的时候只需要将 true 或 false 赋给变量即可，比如 Variable = false。
注：上述四种变量里只有整数变量和浮点数变量存在负值。

## C. 变量的使用

变量不同于常量，它是指变化的量、可变的量，将声明好的变量利用起来是非常重要的。
这里给出几种利用方式，以便于参考：
①手动声明(赋值)
例：Variable1 = true 或 Variable2 = 6221 或 Variable3 = ”ab2”或 Variable4 = 251.61
<变量名> = <变量值>
②变量之间的声明(赋值)
例：Variable2 = 251.1
Variable1 = Variable2
此时 Variable1 的类型会自动变成 Variable2 的类型并获取 Variable2 的值。
③变量的重复使用
例：Variable1 = 1 （此时 Variable1 的值是 1）
Variable1 = Variable1 + 1 （此时 Variable1 的值是 2）
例：Variable1 = 1 （此时 Variable1 的值是 1）
Variable1 = Variable1 + Variable2 （此时 Variable1 的值取决于 Variable2 的值）
例：Variable1 = 6 （此时 Variable1 的值是 6）
Variable2 = Variable1
Variable3 = Variable1
此时 Variable2 与 Variable3 的值等同于 Variable1 的值
D. 在何处可以声明和使用变量
①控制台
②物件的脚本选单中

## (2) 运算符

运算符的种类
在 Thyme 中的运算符有以下几类：

### ①计算运算符

a. + 加
b. - 减
c. * 乘
d. / 除
e. % 取余
f. ^ 幂运算

## ②逻辑运算符

a. &&逻辑与
b. || 逻辑或
c. ! 逻辑非

## ③条件运算符

a. > 大于
b. < 小于
c. == 等于
d. >= 大于等于
e. <= 小于等于
f. != 不等于

## ④其他运算符

a. = 赋值运算符
b. , 逗号
c. () 圆括号
d. ; 分号
e. {} 花括号
f. [] 方括号

## ⑤判断运算符

条件 ? {结果 A} : {结果 B}

## (3) If 条件判断

If 条件语句是程序执行的流程中最重要的一部分，在其他的编程语言中是这样，在 Thyme 中也是如此，不过相对的，Thyme 的语法与其他编程语言依旧有所区别。

表达式：条件 ? {条件为真后所执行的代码} : {条件为假后所执行的代码}

例子：
①假设当前有个变量 A 且它的值是 1，现在要求另一个变量 B 在 A 小于 1 时对外输出 2，在 A 大于等于 1 时对外输出 3，则书写方式如下：

```
A < 1 ? {
    B=2
```

```
} : {
    B=3
}
```

②假设当前物体不具有杀手特性，现在有要求需要该物体具有杀手特性时本身颜色会变成红色，则书写方式如下：

```
killer ? {
    color=[1,0,0,1]
}:{}
```

## (4) For 循环

For 循环语句作为 Thyme 中基本的流程语句，也是相当的重要，它一般在数组赋值和刷物体上具有十分重要的意义。

表达式： for(循环次数, (自变量) => {每次循环都要执行的代码})

例子：

①单纯利用 for 的循环次数来执行代码：

假设目前的场景中没有任何的圆形，此时要求需要刷出 5 个默认的圆，不更改其他设置，则书写方式如下：

```
for(5 , (x) => {
    Scene.addCircle({})
})
```

以上代码执行完毕后将会在原点刷处 5 个默认圆。

②利用 for 循环中的自变量来执行代码：

假设目前的场景中没有任何的圆形，此时要求需要刷出 5 个默认的圆并让它们的色相依次增加五度，则书写方式如下：

```
for(5 , (x) => {
    y=x*5;
    Scene.addCircle({
        colorHSVA:=[y,1,1,1]
    })
})
```

以上代码在执行之后的观察中可以发现，第一个球开始，每一个球都会逐渐增加 5°的值，直至循环结束。

在这里如果想要得到 for 中自变量的终值，则写法如下：

```
for(n , (x) => {
    x
})
```

或

```
for(n , (x) => {
    y=x
})
```

这块代码执行之后将会自动把自变量 x 的值输出到 for 的外面。

## (5) 方法(函数)

编程语言中经常会提到方法(函数)这个名词，而这个名词也无一例外地出现在了 Thyme 的语法中。

### ①无参函数

一个不存在参数的函数被称为无参函数，在 thyme 中它有一个标准形式，它被写成：

方法名(函数名) = {具有相关功能的代码}

例子:
A.现需要一个能够刷半径为 1m，生成位置为鼠标位置，颜色为白色的圆的套组代码，因控制台中一遍遍输入太过麻烦，现在需要将其压缩一下，变成：

```
AddSpecialCircle={
    Scene.addCircle({
        pos:=App.mousePos;
        radius:=1;
        color:=[1,1,1,1]
    })
}
```

输完后回车，即可得到这样一个具有特殊功能的代码 AddSpecialCircle，在控制台中输入 AddSpecialCircle 后回车，将会在鼠标指针的位置生成一个半径为 1m，颜色为白色的圆。

B.现在我有一个数字 A=1，我需要对它进行一个加 1 运算，但是因为一遍遍手动输入非常不方便，我可以将这个动作进行打包，变成：

```
AddOne={
    A=A+1
}
```

这样，一个对 A 进行加一的功能就做出来了，在输入 AddOne 回车后，我们不仅可以发现 A 加了 1 变成了 2，同时也可以发现控制台中也会出现一个与 A 相同的数字，这个数字可以被称作返回值，它取决于这个函数中最后一行输出的值。这里我再引入两个变量 B 和 C，并将它们赋值为 B=100 和 C=20，此时，将 AddOne 改成：

```
AddOne={
    A=A+1;
    B=B+2;
    C=C+5;
}
```

回车后输入 AddOne，我们会得到一个数 25，这个就是 AddOne 中 C=C+5 所得到的值，当然，如果想要输出 A 或 B 的值也是可行的，只需要在最后加上 A(或 B)的字母即可。

```
AddOne={
    A=A+1;
    B=B+2;
    C=C+5;
    A(或 B);
}
```

这个特性可以用来进行一些其他的运算，这里不再进行过多阐述。

## ②有参函数

有参函数顾名思义就是有参数的函数，他的书写规则与无参函数有些不同，它通常被写成：

方法名(函数名) = (参数 1,参数 2,⋯,参数 n) => {具有相关功能的代码}

例子：

A. 有参数但不使用参数（功能近似等同无参函数）
B. 有参数且使用参数

首先我们定义外部变量 A=100、B=300，现在我们需要有个函数能够让 A 与 B 的值自由变化，写法如下：

```
ChangeNumA=(c)=>{
    A = A+c;
    B = B+c
}
```

回车并输入 ChangeNumA(100)，我们可以得到它的值是 400，同时 A 的值也变成了 200，B 的值变成了 400，如果继续输入 ChangeNumA(100)，它的值会变成 500，同时同时 A 的值也变成了 300，B 的值变成了 500。在这里，有参函数的返回值与无参函数是相同的，最后一行的值即为输出的返回值，这个返回值在函数之间的调用中非常有用，在其他地方也很有用，在这里不做过多解释。

在这里，有参函数有一个特别的特性，那就是函数型参数。

函数型参数在 Thyme 并不能直接以函数形式直接出现在参数括弧中，它们通常会把自己藏在一个函数的内部，这里给大家举个例子：

```
fun=(function,x)=>{
    function(x);
};
```

注：这里的 function 已经构成了函数型参数

将上方代码输入到控制台中，我们便可以得到一个带有函数型参数的有参函数，这一类的函数使用有这样一个规则，即：

```
fun((函数型参数的虚拟自变量)=>{
    函数型参数的函数表达式
},影响该函数参数的实际自变量);
```

现在，我们将

“函数参数的虚拟自变量”改成 z；

“函数参数的函数表达式”改为 z^2；

“影响该函数参数的实际自变量”改为任意数字，这里我们输个 4；

现在我们可以得到这样一个函数

```
fun((z)=>{z^2},4)
```

回车之后，我们可以得到一个结果 16，这样，一个函数参数的使用就结束了。

顺便提醒一下，这里 Thyme 所写过的所有代码都是可以被新的代码覆盖的，使用起来非常方便，如果能做到同一变量的复用，则可以尽最大可能减少资源占用。

最后，注意使用的参数名不要与局部变量，方法名，类名等重复，否则会报错。

## (6) 类

类的概念在 Thyme 中比较模糊，它从某种意义上可以划为无参函数中的一部分，但是却不同于无参函数，同时，它又可以划为有参函数，同样的也不属于有参函数，它的存在比较特别，我们接下来将对它进行简单的解释：

### ①类的书写形式

```
类名={
//这是一个类变成实例的前提//
    类的代替名=alloc;

//这一块表示类的变量被初始化//
    类的代替名.变量名 1=数据;
    类的代替名.变量名 2=数据;
                •
                •
    类的代替名.变量名 N=数据;
    类的代替名.方法 1(函数 1)=null;
    类的代替名.方法 2(函数 2)=null;
                •
                •
    类的代替名.方法 N(函数 N)=null;

//这一块表示类中函数的定义//
类的代替名->{
        无参数方法(无参函数)=>{内容};
        有参数方法(有参函数)=(参数)=>{内容}
```

```
        };
        //将类所蕴含的所有东西全部输出//
        类的代替名;
    }
```

## ②类中的相关知识点

A. 一个类可以用来表示一个大的功能，也可以将一些函数归纳整理在这一个类中，具体用来干什么，全看个人。
B. Alloc 是 thyme 中用来实例化类的一个代码，如果没有它，编译器在进行实例化时将不会认为这个类是可用的。
C. 一个类在制作完毕后一定要在使用之前进行实例化，否则这个类将不会有任何作用。
D. 类中可以不包含变量，也可以不包含方法(函数)，如果有方法(函数)的化，一定要进行方法(函数)的制作。
E. 如果父类中有子类，则该子类不能使用父类的相关东西。(问题比较少见)
F. 如果要使用变量，请尽量避免其变量名与任何已有的类名重复

## ③例子

A. 类实际写法
假设现在需要一个能够计算加法的类，可以这样写：

```
    AddNeed={
        An=alloc;
        An.Add=null;
        An->{
            Add=(a,b)=>{
                A+B;
            }
        };
        An;
    }
```

写完后回车，这个类就完工了，
B. 类的实例化
类的实例化是指将制作好的类导入到外部变量中的过程，这里举个例子，就拿上面那个 AddNeed 类来说吧，这里起一个随便点的名字 test，让 test=AddNeed，然后回车，即可发现有 unnamed 出现，证明这个类没有问题，现在，这个类就被实例化了。
C. 类的使用
制作好的类在被实例化后，可以发现之前被赋值的变量后面多了个”.”，这说明这个类可以使用了，以上面那个为例子，我们输入 test.后按下 tab 键可以发现立即补全了一个 Add，此时立刻回车可以发现出现了(a, b)=>{A + B}，这是我们之前类中所写的算法，现在可以通过这种方式来直接查看。如果是有参函数的话，可以在函数名后面加”(参数)”后回车以得到返回的结果，如果是无参函数，则直接回车即可得到返回的结果。

## (7) 标识符

标识符在 Thyme 中处于一个很不起眼的位置上，看似没有什么用，实际用处非常大，接下来我们开始进行讲解。

### ①标识符是什么及标识符的作用

标识符是一种能够帮助区分“现在”跟“将来”的符号，它在 Thyme 中以“:=”出现，其形式像“=”，但却不同于“=”，带有“:=”的变量、函数、类将以一种“将来”的形式出现，而“=”是以“现在”的形式出现。

这里的“将来”是指该变量（函数、类）在以后会出现，而目前并不存在。

这里的“现在”是指该变量（函数、类）在当前已经存在，且以后也存在。

### ②例子

A. 假设我们目前没有 A 这个变量，现在我们需要对 A 进行声明，令 A:=100，回车，此时我们可以得到 A=100，并且发现原来没有的 A，现在已经有了。
B. 假设我们现在已经有了 A 这个变量并且 A=100，现在我们再对 A 进行声明，令 A:=200，回车，此时我们可以发现，系统给出了一个“Variable "A" already declared in this scope”的提示，它的意思是说“变量‘A’已在此范围中声明”，这意味着同样一个变量不能在同一个地方声明两次。同时我们可以发现在这句话后紧跟着一个数字 200，现在我们在来输入 A，回车，可以得到 A=200。

综上，标识符能够创建新的变量（函数、类），并且可以将新建对象的值覆盖到已有的对象上。

## (8) 结构

在 Thyme 中，结构是用来存储不同数据类型(数值，字符串，布尔值，函数，数组)的一个项，它可以存在于任何地方，通常情况下，自定义结构在关闭 Algodoo 后会被自动清除。

①自定义结构的书写形式

A.临时型(可以随时添加新的条目)

```
>结构名  = alloc
>结构名.变量 1 = true
>结构名.变量 2 = 14.1
>结构名.变量 3 = “BBAE”
```

B.类-型(可以在使用之前添加条目，在使用过程中依然可以添加新的条目(不建议))

```
结构名={
    结构名=alloc;
    结构名.变量 1 = true;
    结构名.变量 2 = 13;
```

结构名.变量 3 = "agc";
结构名;
}
②现有结构
A. Scene.my.变量名
B. 所有已存在类.变量名

由此我们可以看出，结构其实就是一个简单的类，通常情况下，结构的稳定性相较类而言，稳定性较差，在写代码时应尽量减少结构的出现。

## (9) 数组

数组的表现形式通常以方括号的形式出现在我们面前，它的形式如同集合，里面由一个一个的元素组成，然而里面的元素不同，带来的这个集合的性质也就不同，接下来将介绍几种 Thyme 中的数组形式。

### ①数组的几种形式

A.普通数组

普通数组，是一种由普通的数据(字符串，数值，布尔值)构成的数组。

数组 = [数据 1,数据 2,数据 3,……,数据 n]

例：

A = [1, 2, 3, 4];

B = [True, False, True, False];

C = [“abc”, ”bcd”, ”efg”]

D = [1, true, ”abcDefg”]

B.变量数组

变量数组，是一种由变量构成的数组，该数组元素的值取决于变量值，这种数组往往需要依靠一个持续执行的函数(update 或 postStep 类的函数)来进行实时更新。

数组 = [变量 1,变量 2,变量 3,……,变量 n]

例：

A = [Sim.frequency,Sim.windAngle]

C.方法数组

方法数组，指由方法(函数)构成的数组，该数组的元素既不是任何形式的数值，也不是变量，而是一个又一个的算法，在使用方法数组前需要将数组的元素提取出来然后才能使用该元素所包含的方法。

数组 = [{方法 1},{方法 2},{方法 3},……,{方法 n}]

例：

A = [{C=A+B},{C=A-B},{C=A*B},{C=A/B}]

D.交错数组

交错数组，指的是由数组构成的数组，因该数组的元素是由数组构成的，故在原始数组的基础上增加了数组的容量。

数组 = [[数组 1],[数组 2],[数组 3],……,[数组 4]]

例：

```
A = [
[1,2,3,4],
[“abc”,”def”,”ghi”,”jkl”],
[true,false],
[{c=a+b},{c=a-b}]
]
```

## ②数组元素的访问方式

A. 非交错数组

非交错数组的元素访问形式通常以“数组名”+“(序号)”的形式进行访问，在访问数组时，序号只能在 0~N-1 上进行取值，其中 N 代表元素个数。

例：

```
>A = [51,11,74,23,8457,1];
>B = A(0)
B = 51

>A = [“abc”,”def”,”ghi”,”jkl”];
>B = A(3)
B = jkl
```

B. 交错数组

交错数组的元素访问形式是以非交错数组为基础，在此情况上以多个序号相连来访问数组中数组的元素，一般情况下，交错数组中元素的访问容易产生混乱，使用时请务必注意。

例：

```
>A = [[[1,2,3],2,3,4], ["abc","def","ghi","jkl"], [true,false], [{a+b},{a-b}]]
>B = A(0)(0)(2)
B = 3
```

## ③方法数组的使用方式

方法数组在使用时不同于其他数组，方法数组在使用时需要先将元素提取出来，然后才能使用方法数组中的元素。

例子：

```
>D=10;B=20
>A = [{D+B},{D-B},{D*B},{D/B}]
>C=A(0)
C=30
```

## (10) 指针

在 Thyme 中，与一个变量相连的另外一个变量被称作该变量的指针，指针变量通常可以被用来读取另外一个变量的值，同样的，指针变量也可以将指向的变量进行值的修改。

指针变量={指向变量}

例：

```
>A=1
>B={A+1}
B=2
>A=2
B=3
```

## (11) .phz 和.phn 文件

### ①.phz 文件(可通过各种压缩软件打开查看)

Algodoo 通常以.phz 形式保存场景和元件，然而我们将它拆开来仔细查看，可以发现它是一个特别的压缩文件，我们所保存的所有 Algodoo 场景和元件在.phz 中一般以三个文件表示，它们是 thumb.png(存档缩率图)、scene.phn(存档本体)、checksums.txt(存档序号码)，这三个文件，除此之外，后来放入的其他文件，Algodoo 将不会自动读取。

### ②.phn 文件(可通过各种非 word 文本编辑软件进行编辑)

在打开的.phz 中，一定存在一个.phn 文件，该文件是这个存档的核心部分。我们用记事本打开来查看，可以发现这里面是用 Thyme 语言书写的，在这里面添加或删除相关数据，均会影响存档打开后的情况，具体修改还请遵守 Thyme 语言的相关语法规范。

## (12) Thyme 额外知识点

### ①什么是“->”

它是一个重定向符号，用于将当前值定向到另一个结构里。

例：

在控制台中输入 Scene->{my->{内容}}并回车，此时，重定向括号内内容的值将会覆盖掉 Scene.my 内已有内容的值。

现在键入以下例子变量并回车：

```
Scene.my.Variable1=1;
Scene.my.Variable2=0;
Scene.my.Variable3=3;
```

```
Scene->{
    my->{
        Variable1=6;
        Variable2=2;
        Variable3=4
    }
};
//得到的结果如下：
Scene.my.Variable1=6;
Scene.my.Variable2=2;
Scene.my.Variable3=4
```

重定向在需要进行大量修改时，可以节省大量的输入时间。

## ②注释

注释是一种特殊的文字，在 thyme 语言中，被注释的文字将不会被计算机所读取，此特性可以用来书写与代码有关的解释。

A. “//”可以注释一行字
B. “/**/”将文字放在两个“*”中间便可以注释相关文字。

## ③字符串扩展

我们都知道，一成不变的字符串变量是不可能满足我们所有需求的，当我们需要将原始字符串进行加长时，我们可以用“+”来将两个字符串合并到一起，这样，一个新的字符串将出现在我们面前（注意:Thyme 没有字符剔除功能）

例子：

```
>A = “aaa”
>A = A + “bbb”
A = “aaabbb”
```

## ④触发器和更新器

A. 触发器

触发器在 Algodoo 中只存在于实体上，它们往往在满足一定条件时才会执行它们所包含的代码。

OnKey 在按下按键时触发
OnCollide 在碰撞时触发
OnSpawn 在该实体生成于场景中时触发
OnHitByLaser 实体被镭射笔击中时触发(实体特有)
OnClick 在点击该实体时触发
OnDie 在该实体消失时触发
OnLaserHit 镭射笔击中实体后触发(镭射笔特有)

B. 更新器

更新器在 Algodoo 中只存在于实体上，它们往往在系统运行后就会执行它们所包含的代码。

postStep 按下运行按钮后的每一帧都在执行，与拟真运行按钮的状态有关

Update 程序启动后的每一帧都在执行，与拟真运行按钮的状态无关

## ⑤何时使用分号

Thyme 语言中分号的使用时机比较灵活，如果只有一行语句则该语句后可以不加分号，但若是有多行语句，则除了最后一行可以考虑加分号以外，其余行都必须添加分号。

# 2. 进阶部分

## (1) 对象指针(此功能 BUG 太多故暂不包含例子)

对象指针是一种特殊的指针，它指向的不是一个数值，而是一个 algodoo 场景中存在的，能够被选中的物体，我们利用指针变量可以从其他地方对该物体的相关属性值进行修改。

对象指针往往包含以下两部分：

实体绑定部分(将其放于实体的触发器或更新器中并替换“(e)=>{}”)：

(参数) => {充当该实体的变量 = {参数}}

需要注意的是，对象指针需要指向一个实体(像几何体，镭射笔等)，如果指向的是一个非实体则将会出现错误。

访问部分：

充当该实体的变量.this.

## (2) 扩展型 for

在基础部分我们提到了 for 循环，而在 Thyme 语言中，for 的循环是利用迭代原理实现的，在日常使用时往往会造成很多不便(如堆栈溢出)，所以在这我们可以利用扩展结构无限制增大循环的次数，以实现我们想要的任何结果。

用法举例：

```
Test= (a,r,c)=>{
        c < r ? {a(c)} : {};
    (c + 1) < r ? {a(c + 1)} : {};
    (c + 2) < r ? {a(c + 2)} : {};
            ...
            ...
```

```
                    ...
    (c + (N-1)) < r ? {a(c + (N-1))} : {};
        (c + N) < r ? {Test(a,r,(N))} : {}
}
```

例子中每增加一行代码意味着增加一倍的循环次数上限。

其中：

a 代表“(自变量) => {每次循环都要执行的代码}”；

r 代表循环次数上限；

c 代表循环次数下限；

在这个方法中，实际循环次数 n 需要用 r - c 来代替并且 r 要大于 c，否则该方法将不会执行。

## (3) 更新链接(自身变量锁定)

有些时候在 Algodoo 中保存的东西可能会因为某些原因而丢失掉数据，此时就需要引入更新链接对会丢失数据的变量进行锁定，以便于以后的再次使用，这对于弹簧、轴承和推进器十分有用。

用法举例：

```
物体上暂时不需要修改的属性(像 opaqueBorders、dampingFactor 等属性)={
    物体上想要保留的属性={
            内容
        };
    暂时不需要修改的属性的值
}
```

## (4) 数组元素追加

在前面的基础部分我们提到过数组，而数组中的元素个数为定值。一般情况下，固定元素个数的数组对我们来说基本够用，可是一旦出现特殊情况，那么这种数组将不会再满足我们的使用，这时候就可以使用“++”对数组元素个数进行追加，以改变数组的大小。

用法举例：

数组 A = 数组 A ++数组 B

例：

```
>A = [1,2,3]
>B = [4,5]
>A = A ++ B
A = [1,2,3,4,5]
```

## (5) 局部变量声明

局部变量在 Thyme 中非常有用，它的出现能够防止函数或实体中的值泄露到根域中，保证了数据的安全性。

①函数中的局部变量

在基础部分中我们讲到过函数，函数一般是由各种算法构成的，它们在执行的时候通常需要执行计算。一个函数中如果经常使用全局变量，将会导致这个函数的值泄漏到函数外面，造成数据的混乱，所以我们在此引入局部变量。

用法举例：

```
测试函数={
    局部变量:=15;
    全局变量 A=全局变量 A+局部变量;
}
```

这个例子就是运用了局部变量对全局变量的控制，使得全局变量 A 在每次执行该函数时都会自动增加 15。

②实体中的局部变量

实体中的局部变量如同实体的属性一样，此内容将会在“物体的脚本选单”部分进行讲解，在此我们不会进行讲解。

## (6) 对象评估

该系列比较特别，它们均有一个共性，那便是具有一个字符串形式的参数。

①Reflection.executeCode

该代码的作用是将已存在的代码进行执行并输出结果。

例：

A. 计算一个字符串中的数据

加：Reflection.executeCode(“50+2”)，结果：52

减：Reflection.executeCode(“50-2”)，结果：48

乘：Reflection.executeCode(“50*2”)，结果：100

除：Reflection.executeCode(“50/2”)，结果：25

取模：Reflection.executeCode(“50%2”)，结果：0

幂运算：Reflection.executeCode(“50^2”)，结果：2500

与：Reflection.executeCode(“true&&false”)，结果：false

或：Reflection.executeCode(“true||false”)，结果：true

非：Reflection.executeCode(“!true”)，结果：false

B. 计算多个字符串的相关联(只能用加号相关联)数据

例：Reflection.executeCode(“50+2”+”50+2”)，结果：50+250+2=302

Reflection.executeCode 的作用除了可以计算单一字符串的功能外，还可以计算多个字符串合并之后的值。当 Reflection.executeCode 的括号中有两个以上的字符串时，则需要将它们用“+”连接到一起。当两个字符串中均有数字需要运算时，Reflection.executeCode 将首先会将多个字符串合并，然后计算合并之后的值。

C. 执行一个字符串中所包含的代码

>A={b+c}

>b=1

>c=2

>Reflection.executeCode(“a”)，结果为 3

②Reflection.executeFile

该代码的作用是将已存在的文件进行读取执行并输出结果，特别的一点是它可以从外部读取外部文件中已存在的数据到 Algodoo 中。

例：三种读取文件的方式

A.从 Algodoo 根目录读取文件

写法：Reflection.executeFile(“文件名.扩展名”)

B.从 Algodoo 目录下的相对路径读取文件

写法：Reflection.executeFile(“\\文件夹名\\文件名.扩展名”)

C.从绝对路径读取文件

写法：Reflection.executeFile(“盘符:\\文件夹名\\文件名.扩展名”)

读取文件的类型：.txt .cfg .ini 等文本形式文件

③eval

该代码的作用是将已存在的代码进行执行并输出结果，同 Reflection.executeCode，但是作用范围不同，eval 的作用范围可大可小，而 Reflection.executeCode 的范围目前仅仅只能在[ROOT]域中使用。

# 4. 物体的脚本选单

物体的脚本选单位于物体右键后弹出的菜单底部，将鼠标移到上方单击，即可打开一个窗口，这个窗口里面包含着很多当前物体所具有的各种属性，在该物体编辑选单中找得到的与找不到的属性，从这里都能找到。

## (1) 只读属性

这些属性中有些是物体本身的身份信息，不能被修改，比如 Liquify、body、area、entityID、geomID。当我们把鼠标放上去点击时，我们会发现它们完全无反应，这就是只读属性。一般情况下，物体的只读属性无法被修改。

在这几个只读属性中，Liquify、body、area 这三个属性比较特别，它们可以被间接修改，在这里我们简单讲述一下它们是如何被修改的。

Liquify，表示液化，修改它就意味着需要将这个物体液化。

body，表示根物体，如果将两个或多个物体相互黏附时，它们的 body 会自动变成相同的，而且 body 数值大的将会变成 body 数值小的，这样就达到了修改 body 的效果。

area，表示面积，在一个矩形或圆形或多边形缩放或修改大小时，该值会自动修改。

## (2) 可编辑属性

①非动作属性(触发器与更新器以外的其他属性)

在把所有只读属性排除之后，得到的便是可编辑属性。它们均由数字、字符串、数组和布尔值组成，当我们把鼠标移上去后单击，我们可以发现该属性的黑框周围出现了一个绿圈，这表示当前值可编辑，这时我们在其中便可以直接修改各种属性值了。

adhesion，表示粘附力，默认值是 0，表示不会与其他物体有粘性。如果将此值调大，则数值越大该物体与其他物体的粘性越大。

airFrictionMult，表示该物体的空气摩擦力系数，默认值是 1，在相同的空气摩擦力环境下，airFrictionMult 越大的物体收到的空气阻力越大

angle，表示角度(弧度)，在 Algodoo 场景中除了圆和平面刚刚绘制出来度数不一定是 0 度外，其余的物体所处的度数均是 0 度。在旋转物体时，从左向右旋转表示负角度，从右向左旋转表示正角度，在旋转时，圆的旋转标记指向正右侧表示 0 度。

angvel，表示角速度，在物体不旋转时，angvel 是 0，当物体从左向右旋转时角 angvel 为负，当物体从右向左旋转时角 angvel 为正。

attraction，表示引力，它的值表示的就是物体编辑选单材质中的引力值。

attractionType，表示引力类型，默认为 2。它有两种类型，一种是线性，一种是平方。当类型为 1 时，表示当前 attraction 的类型为线性，当类型为 2 时，表示当前 attraction 的类型为平方。

collideSet，表示碰撞层，默认为 1。它对应着的是物体编辑选单碰撞作用层，在 collideSet 输入的数据都会在碰撞作用层栏里以二进制的形式进行打勾处理，collideSet 的取值范围在-1023~1023 之间，使用时请注意其范围。

collideWater，表示是否与液体碰撞，默认是 true。

color，表示物体的 RGBA 颜色，我们可以看到它是由包含四个元素的数组构成的，其中第一个值表示 R(红色)，第二个值表示 G(绿色)，第三个值表示 B(蓝色)，第四个值表示 A(透明度)，将它们的值组合起来便可以得到不同的颜色，在某些时候 color 表示的 RGBA 会十分好用，这里不再做过多的解释。

colorHSVA，表示物体的 HSVA 颜色，同样的，它也是包含四个元素，但不同的是，它们是由 H(色相：色彩向量)、S(彩度：颜色的纯净度)、V(明度：亮与暗)、A(透明度)组成的。因 colorHSVA 使用起来比 color 好用很多，所以我们通常都会使用 colorHSVA 来作为我们颜色的第一手处理办法。

controllerAcc，表示控制器加速度，默认值是 11。controllerAcc 出现于物体编辑选单的快捷控制件中，其值会显示在其中控制器加速度的黑色框里。在通常情况下，controllerAcc 几乎不会用到，所以在某些时候用 controllerAcc 来作为数据的传输途径和存储途径。

controllerInvertX，表示控制器的反置 X 轴，默认值是 false。在开启之后，X 轴方向的

控制方向将会与原来相反。

controllerInvertY，表示控制器的反置 Y 轴，默认值是 false。在开启之后，Y 轴方向的控制方向将会与原来相反。

controllerReverseXY，表示控制器的反置坐标轴，默认值是 false。在开启之后，X 轴与 Y 轴方向的控制方向将会与原来相反。

density，表示物体的密度，默认值是 2。

drawBorder，绘制物体的边框，默认值是 true。

drawCake，绘制旋转标记，默认值是 true。

edgeBlur，羽化边缘，默认值是 0。

friction，摩擦系数，默认值是 0.5，它的值表示的就是物体编辑选单材质中的摩擦力。

glued，表示是否已经黏附于其他物体，默认值 false。

heteroCollide，表示自身碰撞层碰撞，默认 false。如果相同碰撞层下的两个物体都开启 heteroCollide，则这两个物体将不会产生碰撞，只有在其中一个物体开启其他碰撞层后才会产生碰撞。

immortal，表示物体不灭，默认为 false。当开启后，该物体将不会被杀手物体所影响。

inertiaMultiplier，惯性常数，默认值为 1。数值越大惯性越大。

Killer，表示物体为杀手，默认为 false。开启后，除了不灭的物体以外，其余的物体均会在碰到该物体后删除。

materiaName，材质名称。可作为该物体的名字来使用。

materialVelocity，材质速度，默认值是 0。在一般情况下，可用于做匀加速——匀速的水平直线运动，正值向左运动，负值向右运动。

opaqueBorders，显示透明物件边框，默认值为 true。

pos，表示当前物体的位置。它由含有两个元素的数组组成，其中第一个值为 X，第二个值为 Y，它们的数据是根据 algodoo 的场景坐标系原点来定的

reflectiveness，反射率，默认值为 1(100%)。值越大反射的光强度越强

refractiveIndex，折光率，默认值为 1.5。值越大反射的光线越强。

resources， 未知属性，Algoryx 开发者专用。

restitution，弹性系数，默认值 0.5。数值越大弹性越大。

ruler，显示标尺，默认 false。开启后可于矩形上显示标尺。

showForceArrows，显示力的箭头，默认 false。

showMomentum，显示动量箭头，默认 false。

showVelocity，显示速度箭头，默认 false。

size，表示矩形的长与宽，它由含有两个元素的数组组成，其中第一个值表示矩形的长，第二个值表示矩形的宽。

text，矩形文本框，在两个双引号中输入文字便可显示在矩形内部。

textColor，表示文字的 RGBA 颜色，它是由包含四个元素的数组构成的，其中第一个值表示 R(红色)，第二个值表示 G(绿色)，第三个值表示 B(蓝色)，第四个值表示 A(透明度)，将它们的值组合起来便可以得到不同的文字颜色。

textConstrained，文字是否处于矩形框内部，默认值为 true。当关闭后，矩形内部的文字如果数量或字体大小太大将不会再被限制在矩形框中。

textFont，文字字体，修改双引号中的字体名可更改矩形框中的字体。

textFontSize，字体解析度，其中数值越大则矩形框中的文字越清晰。

textScale，文字大小，其中数值越大，文本框中的文本体积也就越大。

texture，材质贴图文件，其中的贴图路径读取有三种：
①从 material 文件夹下读取材质贴图：”文件名.扩展名”
②从相对路径下读取材质贴图：”\\文件夹名\\文件名.扩展名”
③从绝对路径下读取材质贴图：”盘符:\\文件夹名\\文件名.扩展名”

textureClamped，纹理固定，调整贴图的固定轴。

textureMatrix，贴图矩阵，调整贴图的缩放情况。

timetolive，存在时间，默认为∞。如果设置它的值为一个小的值后，按下拟真开始按钮后，它将会开始倒计时，当它的值为 0 后，该物体将会自动消失。

vel，物体运动速度，它是由含有两个元素的数组组成的，其中第一个元素表示物体的在 X 轴方向上的运动速度，第二个值表示物体在 Y 轴方向上的运动速度（正值表示物体向上向右运动，负值则相反）。

velocityDamping，速度阻尼，默认值为[0,0,0]，它们三个元素的值分别表示为 X 轴速度阻尼，Y 轴速度阻尼，角速度阻尼。其值在 0~ 1 之间取得

zDepth，物体之间的遮挡关系，数值大物体的在会显示在数值小的上方。

protractor, 绘制量角器，默认 false。开启后会在圆周围形成量角器条纹。

radius，圆的半径。

forceVertexDrawing，强制绘制多边形顶点，默认为 false。开启后无效果。

followGeometry，跟随几何体，默认为 true。

rotation，旋转过的弧度数，其值的正负和大小决定旋转的方向（正向增大是从右向左旋转，反向增大是从左向右旋转。）

allowDirectSolve，允许直接求解，已经失效。

constant，弹簧弹力 N/m。

dampingFactor，弹簧阻尼。

disableCollisions，禁用碰撞，默认为 false。未知用处。

legacyMode，传统模式，是 algodoo 求解器的版本。比如减小 1 就会让 algodoo 用老一版的算法来处理这个物体，在轴承上有一点用。默认是 1，改成 0 会让铰链常数为 NaN 的时候也可拉伸，能控制被拉伸之后 bug 一样乱飞的情况。

length，弹簧距离目标长度。

autoBend，自动弹出马达控制界面，默认 false。未知用处。

autoBrake，马达自动刹闸，默认 false。

bend，轴承马达控制界面，默认 false。

bendConstant，轴承常数，默认 NaN。输入数值后轴承会旋转，数值越大，旋转加速时间差越小。

bendTarget，轴承目标，默认 NaN。输入弧度数后，轴承会根据 bendConstant 的大小

来决定旋转该弧度所用的速度。

ccw，轴承反转，默认 false。

distanceLimit，轴承偏离轴心点距离限制，默认∞。当轴承距离中心点的距离大于这个限制值时，轴承将损坏消失。

hingeConstant，铰链常数，默认∞。当其数值较小时可用于制作减震和轴承动力传输。

ImpulseLimit，轴承破坏冲击力上限，默认∞。当其数值较小时，轴承容易被大的外力所破坏。

motor，启用马达，默认 false。开启后轴承将会变为马达，提供恒定动力。

motorSpeed，马达转速，其值为弧度数。

motorTorque，马达转矩，其值越大，马达提供的动力越大。

cutter，镭射切割器，默认为 false。开启后可用于切割实体。

fadeDist，激光笔消退距离。值越大激光笔的射程越长。

followGeometry，跟随几何体旋转，默认值为 true。关闭后即可实现物件不跟随几合体旋转，此时的物件旋转角度取决于 rotation。

maxCuts，最大切割距离乘数，默认 100。

maxRays，最大光路距离乘数，默认 1000。

showLaserBodyAttrib，显示镭射笔本体，默认 true。关闭后可以隐藏激光笔本体。

velocity，激光笔的 velocity 表示光速，默认∞。

fadeTime，轨迹追踪器消退时间。

②动作属性

onClick，点击触发器，在该属性框中的内容只有在用鼠标点击该物体时才会触发，否则不会主动执行。

onDie，消亡触发器，在该属性框中的内容只有在物体被删除时才会执行，否则不会主动执行。

onKey，按键触发器，在该属性框中的内容只有在按下键盘上任意按键时才会执行，否则不会主动执行，可以结合 Keys.类下的相关代码实现按键的筛选控制。

onSpawn，自身生成触发器，当物体在场景中出现时才会执行，否则不会主动执行。

onHitByLaser，激光照射触发器，物体被激光笔照射到时才会执行，否则不会主动执行。

激光照射触发器中可用来双向获取数据的代码有两种，分别为：

A. e.laser.[照射到该物体的激光笔的自身属性]（onHitByLaser 特有）

B. e.this.[被照射物体的自身属性]

使用时可以随意使用。

onCollide，碰撞触发器，当物体与其他物体发生碰撞时才会执行，否则不会主动执行。

碰撞触发器中的可用来双向获取数据代码有两种，分别为：

A. e.other.[碰撞该物体的其他物体的自身属性]（onCollide 特有）

B. e.this.[被碰撞物体的自身属性]

使用时可以随意使用。

onLaserHit，激光击中对象触发器，当一只激光笔照射到实体上时，这只激光笔的 onLaserHit 才会执行其中内容，否则不会主动执行。

激光碰撞触发器中可用来双向获取数据的代码有两种，分别为：

A. e.geom.[被照射物体的自身属性]（onLaserHit 特有）

B. e.this.[该激光笔的自身属性]

使用时可以随意使用。

postStep，按步更新器，在按下拟真按钮后才会执行其中的代码，否则不会主动执行。

update，系统更新器，在 algodoo 软件启动后即执行，无法主动停止执行。

以上动作属性中都可以应用 e.this.来获取自身数据，这个方法只可以应用于动作属性中，其它的属性中不可使用这种方式来获取数据。

## (3) 自定义可编辑属性

我们可以观察一下脚本选单的界面，属性表上方有一个黑色的输入框，这个输入框的作用就是来增加和修改属性的。

我们举个例子，假如我现在需要一个这个物体的温度值 T，并且让这个 T 获得一个 25 的值，这时我们便可以在输入框中输入一个“T=25;”，按下回车，我们发现物体的属性列表中多了一个 T=25 的黑框，这个便是我们所增加的可编辑属性。

在物件的属性表中，增加的可编辑属性有两种，一种是临时的，即关闭 algodoo 后再次打开便会消失，另外一种就是永久性的。刚刚讲的 T 便是一个临时性属性，如果我们想要一个永久性属性，我们只需要在输入的 T 之前加入下划线“_”，这样这个属性便会成为一个永久性属性。

在基础部分中，我们讲过各种各样的东西，现在如果想要将其应用在物体上，我们只

需要将他们从控制台中转移到这个属性表的黑框中即可。

# 5. 物件的材质

我们将目光转移到物件的材质栏中，我们发现 algodoo 给我们提供了几个原始的材质，点一下后我们发现物体的各项属性都会有微小的变动，这是为什么呢？

我们来到 algodoo 的安装文件夹下（X:\...\Algodoo），可以看见里面包含一个 materials 文件夹，我们打开它，映入眼帘的便是 algodoo 自身所包含的几种初始材质，接下来我们来学习如何新建自定义材质。

## 1. 材质贴图(.png 文件)

Algodoo 的每一种材质都拥有一个专属的贴图，在用户自己制作自定义材质时，可以选择为其添加自制的材质贴图，当然，也可以使用默认材质 default.png 而不为其添加自制贴图。

如果想要为这个新的材质添加贴图的话，需要做到的是尽量保证贴图名称与属性文件的名称相同，同时如果修改了贴图的位置的话，则需要在材质属性里面添加文件的路径(material 文件夹下的路径对应着 algodoo 中物体属性上 texture 里的路径，如果不换则以”文件.png”书写，如更换则请使用相对路径或绝对路径书写)，以便于 Algodoo 能够找到目标文件。

## 2. 属性文件(.cfg 文件，可以使用记事本方式打开)

属性文件是 Algodoo 物件材质的一个非常重要的部分，它里面的全部属性在材质应用时都会一一将所选物体中已有的、属性文件中提到的原始属性全部替换掉。于是我们可以利用这个特性来给物体添加我们需要的属性。如果利用好的话，可以最大限度减少代码工作量。接下来我们通过例子来介绍如何书写属性文件。

例：假设现在我们需要一个材质来让拥有该材质的物体在受到碰撞时变红，并且这里的变红并不仅仅是直接变红，而是慢慢变红，则我们可以这样书写：

```
onCollide=(e)=>{
	colorHSVA=[colorHSVA(0)>0?{colorHSVA(0)-1}:{0},1,1,1];
};
```

将这一部分写入到新建的材质属性文件中，假设这个材质的名字为 changeRed，则我们需要新建一个 changeRed 的.cfg 文件，和一个 changeRed 的.png 文件，随后将上述代码写入到 changeRed.cfg 文件中并保存。随后将这两个文件放入到 materials 文件夹下，此时启动 algodoo，创建一个物体右键打开编辑选单，找到材质栏后，可以看到 changeRed 的材质出现在了材质栏上方，点击它，此时这个物体将会拥有上述代码的特性，尝试将该物体碰撞几次后，可以发现物体变成红色了，这便是自定义材质所带来的特性。

# 6. 场景的样式

场景的样式又称为场景调色板，在 algodoo 新建场景时的窗口中有官方提供的十几种

调色板，它们的存在文件夹则是在 algodoo 根目录中的 palettes 文件夹下，如果想要制作自定义调色板，目前有两种形式，一是从 algodoo 中的右侧属性列中找到新物件，点开后下方就有一个储存调色板，点击后就能将自己选好的颜色制作成调色板，第二种方式便是从外部文件进行制作。

自定义场景样式的方式与自定义物体材质有所不同，物体材质修改完毕后对应的属性值均是某一物体的属性值，而场景样式修改完后影响的则是整个场景，甚至 algodoo 中原始的数据。

自定义场景的属性文件修改有局限性，它只能用于已存在值，而不能新建值，相对的，自定义的物体材质则能够新建值。接下来我们加将通过例子来介绍如何书写自定义场景文件。

例：假设我需要一个场景中拥有温度，并且这个场景中的温度是其他场景中温度的 2 倍，则需要这样书写：

其他场景：

```
……;
……;
……;
T = 25;
```

新加入场景：

```
……;
……;
……;
T=50;
```

此时将保存的自定义场景放到 palettes 文件夹中，打开 algodoo，我们可以发现，新建场景的窗口中多了一个刚刚定义好的模板，在打开他之前先点击其中任意一个模板新建一个场景，随后在控制台中输入 T 回车，可以得到一个数据 25，现在，我们创建一个刚刚的自定义场景，接着在控制台中输入 T 回车，可以得到一个数据 50。

通过这个例子，我们可以在接下来的使用中将任意已存在的东西通过场景的更换来修改。

# 7. 启动自读取

有些时候我们会发现在 algodoo 中输入过的变量或者函数等东西，再关闭 algodoo 后又打开时会消失，这个现象出现的原因则是因为输入过的东西是以临时值存在的，并不能长期保存，然而我们又需要将这些东西利用起来，所以这里可以使用 algodoo 提供的启动自读取来保证我们定义好的东西不会在关闭 algodoo 时消失，并且能够在下一次启动 algodoo 时将它们重新录入。

我们将目光转向 Algodoo 的根目录，观察后可以找到一个 autoexec.cfg 文件，打开它后可以在当中写入自己想要重复使用的功能和数值，以便于在每一次打开 algodoo 时都能够很方便地使用。

在向其中输入数据的时候需要注意的一点则是不要忘记在每个功能和数值后方加分号，以保证数据读取的成功。

# 8. 脚本库的使用方法

脚本库是由 Algodoo 用户们自己为了方便而整合出来的一中外挂式文件，它内部一般由各种类和各种方法组成，当在使用脚本库时需要将这个脚本库文件通过在启动自读取文件 autoexe.cfg 中输入“Reflection.executeFile “脚本库路径”;” 来进行外部挂载，同样的，在 Algodoo 的控制台中也可以进行临时挂载，在挂载成功后即可在 algodoo 中使用该脚本库中的内容。

# 9. 如何书写一个自己的脚本库

第一步：准备一个空的，可编辑的，并且可以用 Reflection.executeFile 读取的文件。

第二步：将自己可以复用的函数脚本或类脚本分条放入其中并在每一条的最后方加入分号。

第三步：将库中所引用的外部库用 Reflection.executeFile 进行关联（开头输入分号隔开）

第四步：将这些脚本统一放在一个大类的下方并对这个类中的每个小类进行实例化（如果需要）

第五步：将大类实例化

第六步：保存文件

通过以上几步便可以制作一个属于自己的脚本库，这个脚本库不仅可以给自己用，还可以将其分享出来，供他人使用。

# 10. 总结

通过上述学习，我们可以基本掌握 Algodoo 中脚本的基本使用方法，并且在未来的 Algodoo 创新道路上，我们也能够用脚本来改变我们使用 Algodoo 的思路和方法。从某种意义上来说，这个脚本教程尚不完全，还需要大家的一起努力，努力将它变得更加完善和便于阅读和使用。