Programming with NASAL

NASAL 编程指南

作者：Donald Young

目　录

第 1 章 序 言

NASAL 是 FlightGear 使用的一种非常强大的脚本语言系统，它与 FlightGear 关系十分密切，可以读取、修改 FlightGear 的属性树中的属性值，可以函数直接访问 FlightGear 的内部数据，可以创建 GUI 对话框，等等。要想深入的理解 FlightGear，并对其进行开发，NASAL 是一个强有力的工具。由于 FlightGear 在不断的更新，其中 NASAL 也有可能有部分升级变化，本文中内容如没有特殊注明，全部基于 FlightGear 2.10 版。

1.1 什么是 NASAL

NASAL 的全称是 Not Another Scripting Language。NASAL 借鉴了如 Javascript、Python、Perl 等流行的脚本语言系统的设计理念，实现了一个简单且完整的 OOP（面向对象程序设计）脚本语言系统，并且是不依赖于操作系统平台的。NASAL 内置了一个垃圾回收器来管理内存，因此，您不必像 C 程序员那样，时刻注意小心溢出可能导致的程序崩溃。

NASAL 的语法规则有点类似于 Javascript，因此，一个有经验的程序员，通常可以很快的上手。FlightGear 提供了很多用于库函数，这些函数都可以被 NASAL 直接调用。

和其他的脚本语言系统一样，NASAL 是平台无关的，并且是纯文本格式的。所有的 NASAL 代码在 NASAL 虚拟机中编译并执行。这意味着您可以使用任意一款文本编辑器来编写 NASAL 代码，并且这些代码在 Linux、Windows 等系统上执行的效果是一样的。

NASAL 代码可以在 FlightGear 的 aircraft 配置文件中调用执行，也可以嵌入到 XML 文件里面执行。FlightGear 中有各种 XML 文件用于描述对话框、指定物体动作、响应键盘游戏杆的操作、对应场景中的物体等等。这些 XML 文件中都可以嵌入 NASAL 代码并执行。

1.2 为什么用 NASAL

作为一种脚本语言，NASAL 具备多数脚本语言系统的基本特点。现在各种编程语言林立，有 C++，Java，C#，VB 等等，但脚本语言仍然有其自己的生存空间。脚本语言一般没有自己的编译器，相对复杂的 C++之类的高级语言。脚本语言系统简单的多，只具备基本的变量、函数、程序流程控制和面向对象等功能，但脚本语言是高级语言的扩充。

通常脚本语言不能独立运行，只能存在于宿主程序中，作为宿主程序的功能扩展。试想一下，一个游戏开发完成后，需要经过测试，发行，升级。在测试过程中如果发现了问题，需要对游戏中的一些属性值或是逻辑关系进行修改，重新修改再编译源代码无疑是个非常恐怖的工作。这时，脚本语言就可以发挥重要作用了，只需根据需要修改脚本代码，在源程序无需重新编译的情况下，原来的逻辑和属性就可以修改了。同时，脚本也为工作分工提供了便利。一个游戏只需少数的核心程序员完成游戏引擎的制作，游戏的关卡等制作完全可以由外围的程序员使用脚本来完成。

一般来说，一个程序或游戏会用一种特定的脚本语言。比如，现在很多游戏，如《魔兽世界》、《锁定：现代空战》、《鹰击长空》等使用了 LUA 脚本语言。FlightGear 使用了 NASAL 脚本语言，和其他流行的脚本不同的是，NASAL 对 FlightGear 的支持非常的好，只用少量的代码，可以十分方便的访问 FlightGear 中的各种属性，调用 FlightGear 的功能函数。最重要的是，虽然 FlightGear 是完全开源的，我们仍然可以在不修改 FlightGear 的源码情况下，仅利用 NASAL 脚本就可以实现很多新的功能。

1.3 NASAL 的应用

虽然 NASAL 是一个完备的脚本语言系统，但其应用的范围还是很窄，目前 NASAL 主要用于 FlightGear。如果需要在 FlightGear 下面开发一个新的系统或是设计实现一个新的功能，NASAL 是一个非常棒的工具。

例如，有一个 bombale 的脚本就是在没对 FlightGear 的源码进行任何修改下，仅用 NASAL 就实现了空战功能。也就是说，这个 bomable 脚本用 NASAL 实现了一个全新的 MOD 版的 FlightGear。

无论地境、飞机、AI 还是其他的内容，在 FlightGear 的设计之初，都是不支持的。现在，在 FlightGear 中有了如属性树，NASAL 脚本语言等高度灵活的子系统，这些功能都非常容易的实现了。

FlightGear 经过了近 10 年的发展，不断完善，内容越来越丰富，功能越来越强大。这其中有 C++程序员的不懈努力，但 FlightGear 提供的开放式的二次开发环境也作出了重要的贡献。

到 2009 年 3 月，FlightGear 中内置的 NASAL 代码达到了 170,000 行，相对 2006 年增长了 6 倍。到 2011 年 10 月，代码已经达到了 326,000 行。并且随着版本的更新，这个数字还在不断的增长。现在 NASAL 已经成了开发 FlightGear 的一个十分重要的工具。在 FlightGear 的系统中，除了十分影响运行效率的功能，大部分功能都使用 NASAL 来实现。

如果有 NASAL 方面的疑问，可以参考 NASAL FAQ，或是到官方网站（http://wiki.flightgear.org/Nasal_Modules）去查询帮助。

1.4 NASAL 的特点

1．NASAL 是一种可扩展的语言，非常适合嵌入到程序中运行。

2．NASAL 可以在返回前指定运行几次后中断，后面的调用可以继续以前的运行状态，这种功能类似某些游戏的存盘功能。

3．NASAL 有一个使用 ANSI C 编写的小巧的解释器，甚至比 LUA 还短小精悍。

4．NASAL 的虚拟机内部使用的是堆栈机模型。

5．NASAL 运行时不依赖于任何第三方库和工具。

6．NASAL 有一个简单易用的扩展 API，并且和其他的代码兼容非常好。

7．NASAL 是线程安全的。

8．NASAL 通过名字空间对代码进行模块化管理，并且可以实现很多编程技巧。

9．NASAL 支持 Unicode，这为 FlightGear 的国际化提供了支持。

10．NASAL 支持异常处置。

11．NASAL 支持标准 OOP（面向对象）编程方法。

1.5 赠与 C++程序员

C++代码一直以严谨高效著称，这也是 C++程序员得以自豪的一点。但 NASAL 作为脚本语言并不意味着低效。实际上 NASAL 是 FlightGear 中各个系统的熔接器，正是有了 NASAL，各系统间的沟通变得非常简单。并且，我们可以非常容易的增加 C++代码，然后在 NASAL 中调用，从而解决效率的问题。

在 FlightGear 的开发中，NASAL 非常方便且功能强大。我们无需安装庞大的 Visual Studio 或其他的 C++编译器，何况还要配置复杂的第三方支持库。只要您的机器可以运行 FlightGear，那么你就可以编写 NASAL 代码。

实际上，C++属于中级语言，介于高级语言和低级语言之间。语法比较复杂，学习起来难度较大。而 NASAL 属于高级抽象语言，更加注重对问题的描述，使得程序员可以更多的去考虑需要实现的功能，而不是在语法正确与否的问题上绕弯子。即使没有编程基础的人员也会发现，NASAL 是如此的简单，稍

微经过培训学习，就可以进行代码编写了。当然，基本的编程思想还是要通过不断的学习和工作进行积累，这一点没有任何捷径可走。

1．插　件

有经验的 C++程序员也许会找 SDK 来开发插件，但遗憾的是 FlightGear 没有 SDK，当然，也不需要，因为 FlightGear 是开源的。任何人都可以获取 FlightGear 的源代码并根据自己的需要进行二次开发。但是修改源代码通常不是一个好的选择。现在 FlightGear 的很多功能和系统的实现都是用 NASAL 来实现的，如果想要修改或增加某些功能的话，不妨查看一下 FlightGear 中的 NASAL 代码。

2．性　能

很显然，C++代码的运行效率要比 NASAL 高的多。但是，当 NASAL 如果不是特别的慢的话，那么对整体性能的影响就不大。早期的测试显示，NASAL 的垃圾回收器的效率比 perl 的要高，数字处理能力比 python 的效率要高。因此，在考虑到程序的运行效率和易用性之间的平衡，NASAL 还是很有使用价值。

3．垃圾回收器

NASAL 有一个实时的垃圾回收器，相信这也是 C++程序员羡慕的地方。我们再也不用直接和内存打交道，手动控制空间的分配与回收，不小心还会导致溢出。这些在 NASAL 中都不复存在，我们可以放心的使用变量，至于空间的分配，交给 NASAL 吧，它知道怎么做。

4．线程安全

和其他多数脚本语言不同，NASAL 是线程安全的。不需要 global lock 之类的指定，解析器会自动进行线程安排，唯一的限制就是垃圾回收器会阻断所有线程的运行。

5．异常处理

和其他语言类似，NASAL 支持异常处理。不需要用到“try”来进行异常处理，当用到 call( )来调用一个函数对象时，如果有异常，会自动调用 die( )函数来进行异常处理。

1.6　NASAL 资源

由于 NASAL 的使用范围狭窄，因此，很难获取丰富的资料来完成本书。编写本书的素材主要来自 wiki.flightgear.org。因此，如果对 NASAL 的使用有疑问，本书没有阐述清楚的，可以到网站上去获取相关的知识。

第 2 章　Hello NASAL world!

和多数程序语言一样，我们还是从最经典的 Hello world！程序入手，来了解这个语言系统。NASAL 语言设计的时候充分吸取了各种语言系统的成功经验。因此，如果您是一个经验丰富的程序员，您会高兴的发言，即使没有经过学习，您仍然可以轻松的阅读 NASAL 代码，甚至做一些修改。但是，如果需要编写自己的代码，那就要了解一下 NASAL 语言的基本内容，如语法规则，变量、函数的命名，程序流程的控制等等。各种编程语言的基本思想都是相通的，所以，具备基本的编程素质后，学习 NASAL 就会有事半功倍的效果。

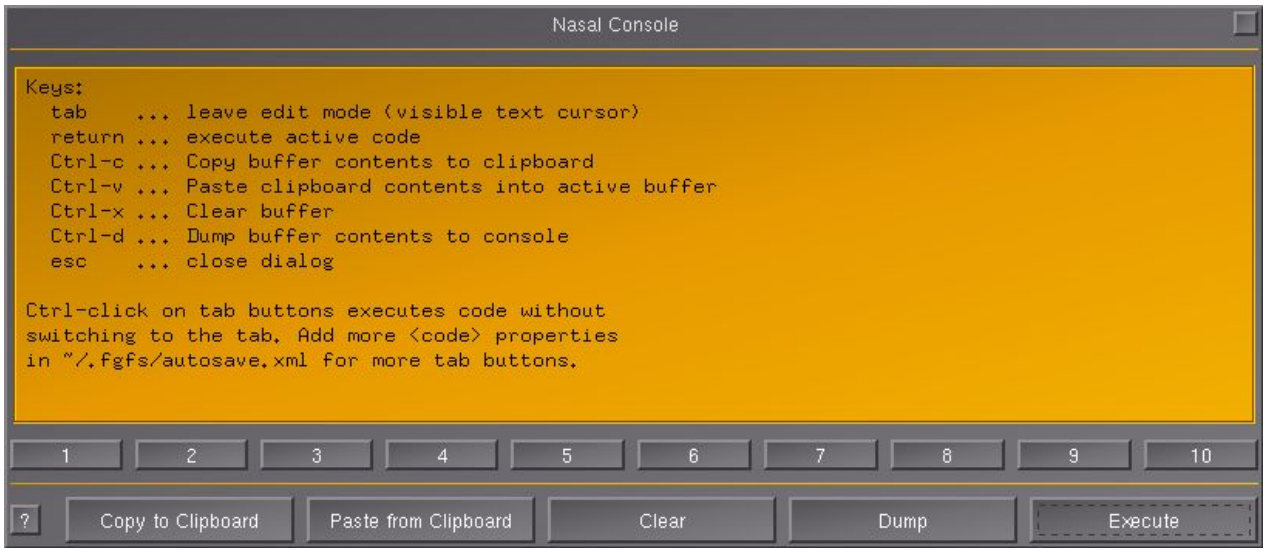
2.1　NASAL 的运行环境

每种编程语言都有自己的运行环境，比如在 VC 中，VC 提供了一个功能强大的 IDE，编辑代码后，点击按钮就可以编译执行，看到运行结果。但脚本语言不一样，因为一般脚本语言都是依赖于宿主程序运行的，少数的脚本系统有自己的运行环境，甚至有调试器。

非常不幸的是，NASAL 是一个冷门的语言，注定不会有大量的资源对其关注。因此，NASAL 没有独立的运行环境，没有专用的编辑器，甚至，在主流的编辑器中都没有 NASAL 的高亮显示模板。但选择了 FlightGear 就必须选择 NASAL，在现阶段没有其他选择的情况下，我们只能去适应它，学习他，然后灵活的使用它。Notepad++是一个非常不错的选择，它是一个开源的免费的编辑器，内置了很多种语言的高亮显示方案，可惜没有 NASAL 的。不过，我在对它的 cpp 模板文件做简单的修改后，就支持 NAS 文件了。并且可以实现注释、关键字、函数、字符串等的高亮显示，可以设置自动完成提示，可以折叠代码方便查看，还可以利用它本身强大的编辑功能，非常的方便。

NASAL 的宿主程序就是 FlightGear，因此，想要编制自己的代码，就必须使用 FlightGear 这个环境，我们可以把 FlightGear 当成 VC 的 IDE。NASAL 代码的执行有两种方式，一是嵌入到 XML 中，FlightGear 加载的时候会加载代码，然后根据我们的指令来执行相关的代码。还有一种方法就是在 FlightGear 的 NASAL Console 中输入代码并运行。两种方式各有优劣，第一种方式的优点是可以利用自己喜欢的编辑器，如 Editplus 或 Notepad++等编辑代码。这些专业的编辑器在编辑 XML 文件时提供了很多方便的功能，但是缺点就是，NASAL 代码必须预先编好，在 FlightGear 启动时加载到内存中，然后根据需要运行。修改代码后，虽然可以通过 NASAL 代码重新加载文件，但有些如初始化变量的代码不会执行，也就是说，我们修改了文件后，不一定能看到修改的结果。第二种方式的优点是，任何代码都可以立即执行，并马上看到结果，但缺点是 FlightGear 是一个运行环境，而不是编辑器。我们只能在窗口中一行一行的敲入自己的代码，代码的输入非常麻烦。在最新的 2.10 版的 FlightGear 中，Nasal Console 增加了对剪贴板的支持，这样就可以方便的和外部编辑器之间通过剪贴板来交换数据。在实际中，我们一般结合两种方法进行使用，一个小功能我们先在窗口中输入运行，查看结果，满意以后，再把代码补充的文件中去。有些程序运行逻辑方面的功能，可以直接在文件中进行修改，修改结束后，用一个命令重新加载文件，就可以看到修改的结果。

在 FlightGear 运行后，点击“Debug”菜单，然后选择“Nasal Console”，就可以打开 NASAL 窗口，如下图所示。窗口下部有 10 个按钮，相当于 10 个文件标签，我们可以在每个标签中编写自己的代码，不同标签中的代码不会互相影响。点击“?”按钮，可以查看帮助。这些标签都支持自动保存，编写的代码会自动保存在当前标签下。重启 FlightGear 后，对应的标签的代码会自动加载，对于我们调试来说，非常的方便。代码编写完成后，点击“Execute”按钮就会执行，执行结果在另外一个窗口中可以看到。启动 FlightGear 后会开启两个窗口，一个是程序运行窗口，也就是我们通常说的程序窗口。还有一个命令行窗口称为控制台，主要用于显示程序运行的信息、调试信息、还有根据需要显示的信息。其中 NASAL 的运行结果就这个控制台窗口中显示。

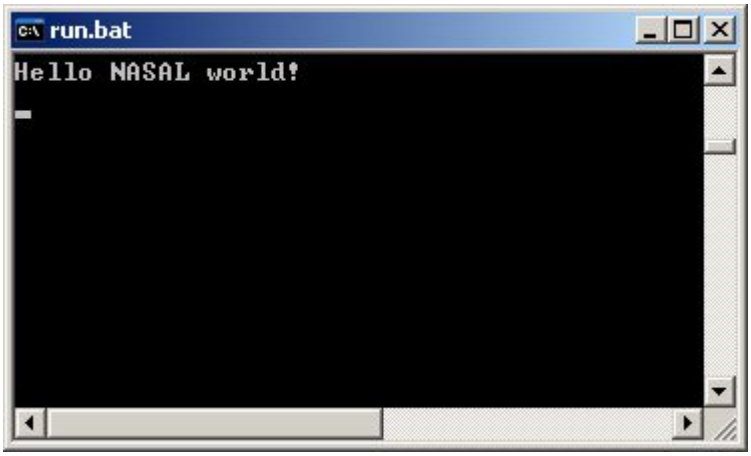


2.2 第一个程序

现在，开始我们的第一个程序。在 Nasal Console 窗口中输入：

```
print("Hello NASAL world! ");
```

点击“Execute”运行后，在命令行窗口中就可以看到运行结果：



是不是十分简单？虽然这个 Hello world 程序基本上不具备任何功能。但通过这个简单的程序我们可以发现：

1．NASAL 的程序运行非常简单，代码无需编译，点击直接就可以运行，马上就可以看到结果，没有 C++语言复杂的编译、链接过程。

2．语言风格和 C++语言类似，每行代码以“;”结束。

3．大小写是敏感的，如果输入“Print”就会出现错误提示。

NASAL 语言的注释使用“#”符号作为一行的开头。例如

```
#This is Hello world Program
print("Hello NASAL world!");
```

2.3 print

我们接触 NASAL 的第一个函数就是 print，之所以把 print 单独拿出来，是因为我实在对它是情有独钟。在 C++语言中，这个函数叫做 printf。printf 函数还有一个名字，称为穷人的调试器。在早期的 C 语言开发环境中，调试器不是很强大。因此，程序员在调试程序时，如果想要查看某一个变量的值，就会打印出来看看。这个习惯一直延续了下来，直至现在，仍然有很多程序员喜欢用 printf 函数查看程序运行过程中变量的值。甚至到了 windows 编程中，他们还是喜欢用 MessageBox 函数输出变量。由于 NASAL 没有专用的编辑器和调试器，因此 print 函数便显得尤为重要。灵活运用 print 会给 NASAL 代码的调试提供很好的帮助。NASAL 中的 print 功能更强，使用更简单。

print 函数的使用很简单，就是括号里加个参数就可以了，这个参数可以是字符串、变量等，他都会正常输出，如果有多个参数需要同时输出，用“,”隔开就可以了。

```
print("Hello"," NASAL ","world!");
```

在这行代码中，使用 print 函数连续输出了 3 个字符串，中间用“,”隔开，在输出时，会自动把 3 个字符串连接到一起。

运行的输出结果为：

```
Hello NASAL world!
```

如果参数中有变量，无需关心变量时数值型的还是字符串型的，直接输出即可。参考如下的代码

```
var x = 1;
```

```
var y = "string";
print(x,"\n",y);
```

运行的输出结果为：

```
1
string
```

在这个例子中使用了转义字符“\n”，这个也是从 C++语言借鉴过来的。目前 NASAL 支持的转义符有：

```
\n 换行
\t 水平制表位
\r 回车
\\ 反斜线
\' 单引号
\" 双引号
```

第 3 章　变　量

3.1　变量的声明

NASAL 使用关键字 var 来声明变量。和 C++语言不同，不用如 int、float、char 等关键字来指定变量的类型，NASAL 中只有一个关键字就是 var，无论变量是数值型的还是字符串型的，都用 var 来进行变量的声明。不用 var 声明而直接使用也可以，但这不是一个好的习惯。NASAL 的基本变量有 4 种，分别是数值、字符串、数组和哈希。变量名是区分大小写的，abc 和 Abc 是两个不同的变量。

声明变量时，必须为变量赋初值，也称为初始化，不能像 C++语言一样声明一个没有值的变量。

在一行中只能声明一个变量，不能在同一行声明多个变量。

```
var w = 100;      # w 是数值变量
var x = "hello";  # x 是字符串变量
var y = [];       # y 是数组
var z = {};       # z 是哈希
var min;          # 错误，没有赋初值
```

声明变量时应避免使用以下保留的关键字。

```
and       or        nil       if        else          elsif
for       foreach   while     return    break         continue
func      var       forindex
```

3.2　数　组

我们可以把同类的一些数据放置在一起，通过数组的方式组织这些数据，在需要的时候，通过下标来取得对应的内容。注意：和 C++语言一样，NASAL 的数组的下标也是从 0 开始的。

```
var array = ["A", "B", "C"];
print(array[0]); # 输出 A
print(array[1]); # 输出 B
```

```
print(array[2]); # 输出 C
```

NASAL 是一种十分灵活的语言，您甚至可以把不同类型的变量放到一个数组中，例如，把数值和字符串放到一个数组中：

```
var my_vector = ["A", 123, 25.12];
```

数组也可以作为一个元素保存在数组中，组成二维数组或是更多维的数组。

```
var arrayX = ["A", "B", "C"];
var arrayY = ["D",arrayX]
print(arrayY[0]);    # 输出 D
print(arrayY[1][0]); # 输出 A
```

NASAL 实现数组使用的向量的方式，与栈这种数据结构类似。而不是固定的一块连续的存储空间。这就意味着数组的大小是可以改变的。NASAL 有一些用于数组的库函数。

使用数组时要注意，数组的大小是固定的。如果在声明时，给出了数组元素，那么数组的大小就是元素的个数。如果没有给出元素，那么数组的大小是 0。但是数组的大小可以动态改变，使用 setsize，append 等函数，可以在运行时改变数组的大小，这样就可以根据实际情况灵活而方便的使用数组。

数组中的维数是一维的，也就是说数组中的元素是线性而不是矩阵方式排列的。但是我们可以把数组作为另外一个数组的元素来变相实现多维数组或矩阵。和其他语言不同，在其他语言中，例如二维数组类似于方阵，每一行和每一列元素个数是相同的。而 NASAL 中的二维数组，不同行的元素个数可以不同，完全取决于作为元素的数组的大小，这样可以实现其他语言难以实现的稀疏矩阵。

append(array, elements…)

append 函数用于向数组中增加元素，该函数有两个参数，第一个参数指定了用于操作的数组，第二个参数是要增加的变量。如果要增加多个元素，使用“,”将各个元素隔开即可。增加元素后，便可以用下标对新增加的元素进行访问，同时要注意使用下标时，不要超出数组的大小，如果不清楚数组的大小，可以用 size 函数获得。

```
var array = ["A", "B", "C"];
print(arry[3]);   # 错误，下标超出范围
append(array, "D");
print(arry[3]);   # 输出 D
append(array, "E", "F", "G");
print(arry[6]);   # 输出 G
```

setsize(array, size)

setsize 函数用于改变数组的大小，该函数有两个参数，第一个参数指定了用于操作的数组，第二个参数是新的数组大小。如果指定的大小大于原数组大小，原数组自动增加一些空元素以达到新的大小。如果指定的大小小于原数组大小，原数组会截取掉后面的元素以适应新的大小。

subvec(array, start, length=nil)

subvec 函数可以截取数组中的部分元素成立一个新数组。该函数有三个参数，第一个参数指定了用于操作的数组，第二个参数是新的数组起始位置，第三参数是新的数组的结束位置。第三个参数如果不指定的话，默认结束位置为原数组的末尾。

pop(array)

pop 函数和栈的弹出是一个概念，该函数返回数组的最后一个元素后，同时删除最后一个元素，数组的大小自动减 1。我们可以把数组当成一个栈，栈底是第一个元素，栈顶是最后一个元素，可以对栈进行 append 进行压栈，也可以通过 pop 函数弹出栈顶元素。

sort(vector, function)

sort 函数用于对数组里面的元素进行排序，该函数返回的是所有元素按照升序排列的一个新的数组。function 是指定排序的规则函数。规则函数有 2 个参数，并且该函数根据两个参数的关系返回一个比较结果。如果第一个参数小于第二个参数，规则函数返回一个负数；如果相等，则返回 0，如果大于，则

返回一个正数，sort 函数就是使用该规则函数对所有元素进行排序。通常规则函数的 2 个参数用 a 和 b，就像用于循环的变量，我们通常使用 i，j，k 一样。需要注意的是规则函数需要根据数组元素的类型来指定，如果数组的元素为数字型，规则函数就要按照数字方式处理；如果数组的元素为字符串型，那么规则函数就要针对字符串编写比较函数。

```
var list = [2,0,3,1];
var slist = sort(list, func(a, b) a-b);    # slist 为[0,1,2,3]
var m = ["2","0","3","1"];
var n = sort(m, func(a,b) cmp(a,b));       # slist 为["0","1","2","3"]
```

3.3 哈 希

NASAL 还有一种变量称为哈希（hash），在算法中 hash 表示一种单向混淆的算法算法。哈希这种变量可以作为类用于实现 OOP，也可以用于实现映射。数组是一种从数字下标到元素的映射方式，在某些情况下，这种数字下标不直观，不具备可读性。在 C++语言中我们可以用宏定义的方式用一串字符来代替数字，在标准 C++中有一种 MAP 映射的数据结构，MAP 这种数据结构就可以把两个元素建立联系。比如用一个字符串作为索引，找到对应的元素。NASAL 的哈希变量就可以实现这种功能。和数组不同的是，哈希使用的是“{ }”，而数组使用的是“[ ]”。例如，用如下方式声明含有 3 个元素的哈希。

```
var hash = {first:"A",second:"B",third:"C"};
```

访问数组中的元素使用的是数字下标，而访问哈希中的元素使用“.”，和 C++中的类的方法类似。

```
var hash = {first:"A",second:"B",third:"C"};
print(hash.first);   # 输出 A
print(hash.second);  # 输出 B
print(hash.third);   # 输出 C
```

如果给没有列表中的下标的赋值，则新的下标会自动增加到哈希中去，相当于增加了一个元素。

```
var hash = {first:"A",second:"B",third:"C"};
hash.forth = "D";
print(hash.forth);   # 输出 D
```

更为灵活的是，哈希和数组可以嵌套使用，即数组可以作为哈希的一个元素存在，哈希也可以成为数组的一个元素。需要注意的是，用于作为哈希使用的下标字符串必须是一个有效的字符串，而不能用一个字符串变量。例如：

```
var array = ["A", "B", "C"];
var hash = {first:"D",second:array};
var array2 = ["E",hash];
var strIndex = "first";
print(hash.first);          # 输出 D
print(hash.second[0]);      # 输出 A
print(array2[1].first);     # 输出 D
print(hash.strIndex);       # 错误，不能用字符串变量作为下标
```

同样，NASAL 内置一些处理哈希的库函数。

contains(hash, key)

contains 函数用于判断哈希中是否有一个下标。该函数查找的是哈希中的下标，而不是哈希所包含的元素，这一点需要注意。第二个参数指定的查找的下标，该参数需要用字符串方式指定。如果包含，返回 1，否则返回 0。

```
var hash = {first:"A",second:"B"};
```

```
print(contains(hash, "first"));     # 输出 1，表示包含下标 first
print(contains(hash, "third"));     # 输出 0，表示不包含下标 third
```

delete(hash, key)

delete 函数用于删除哈希中的一个元素。同样，第二个参数需要用字符串的形式指定下标。

```
var hash = {first:"A",second:"B"};
print(contains(hash, "first"));     # 输出 1，表示包含下标 first
delete(hash, "first")
print(contains(hash, "first"));     # 输出 0，下标 first 已删除
```

keys(hash)

keys 函数返回一个数组，该数组包含了哈希中的所有的下标字符串。

```
var hash = {first:"A",second:"B"};
keys(hash);       # keys 为["first","second"]
```

3.4 变量的高级使用

实际上，var 声明不是必须的，一个没有经过 var 声明的变量也可以使用，这和 C++语言有点区别。实际上 var 的作用是声明该变量时局部变量，而不是全局变量。不过仍建议在编程中使用 var 来声明变量，一是为了养成良好的编程习惯。二是为了变量的安全，因为一个没有经过 var 声明的变量，有可能在其他的文件中被改变，而这种改变可能并不是我们料想的，这种错误很难被发现。不同的文件具有名字空间的作用，如果想要调用其他文件中的变量，可以使用名字空间的方式调用。下面的例子是在两个不同的文件中对同一的变量的操作。

```
# hello.nas
var greeting = "Hello World";   # 在 hello 名字空间中定义的一个字符串
```

```
# greetme.nas
print(hello.greeting);          # 通过前缀调用其他名字空间中的变量。
```

使用 var 在进行变量的初始化时，其灵活性可以让 C++汗颜。作为一个 C++程序员，我第一次看到 NASAL 的这种怪异的赋值方式后，我的下巴差点掉了下来，半天没有合上。

注意使用多个变量赋值时，要用“( )”把变量放到一起，变量间用“,”隔开。

```
(var a, var b) = (1, 2);            # 同时声明多个变量
var (a, b) = (1, 2);                # 更简短的方式
(var a, v[0], obj.field) = (1,2,3) # 支持变量、数组、哈希多种表达方式
var color = [1, 1, 0.5];
var (r, g, b) = color;              #支持对数组的解析
(a, b) = (b, a);                    #这样就可以对两个变量交换，惊讶吧
```

NASAL 在操作数组时也是十分的灵活。

```
var v1 = ["a","b","c","d","e"]
var v2 = v1[3,2];     # v2 为 ["d","c"];
var v3 = v1[1:3];     # V3 为 v1 的 1 到 3 元素 ["b","c","d"];
var v4 = v1[1:];      # 没有第二个值表示到结尾 ["b","c","d","e"]
var i = 2;
var v5 = v1[i];       # 用变量作为下标 ["c"]
var v6 = v1[-2,-1];   # 负数下标表示从结尾开始，-1 表示最后一个元素 ["d","e"]
```

3.5 选择属性树还是变量

FlightGear 中有个十分重要的概念就是属性树。FlightGear 把运行中的所有的参数用树的形式组织到一起。如果您准备研究 NASAL，那么相信您已经对属性树有了比较深入的认识了。属性树和我们常用的 windows 的资源管理器一样。硬盘里的各种文件用目录和文件的形式进行组织，目录中可以包含文件，也可以包含子目录，文件就是最终的节点。因此，如果我们选择使用 NASAL 对 FlightGear 进行功能开发和扩展。在变量的使用上有两个选择，一是使用 NASAL 自己的变量，一是使用 FlightGear 目录树中的属性。这两种方式 NASAL 都可以很方便的调用。那么该如何选择呢？

NASAL 变量的特点是简单，而且速度快，如果一个变量只在 NASAL 代码中使用，那么 NASAL 变量是个不错的选择。

属性树中的属性是 FlightGear 内部各系统间通信的重要的工具，这个属性可以被 C++代码，XML 配置文件，NASAL 代码访问，如果一个值需要被多个系统，多个文件使用，那么把这个值放到属性树中作为一个属性十分合适。

还有一个重要的原因是 FlightGear 的运行效率，NASAL 变量使用起来运行速度快，据统计，使用 getprop 函数操作属性时，将比变量慢 50%。即使使用 node.getValue( )函数，依然会慢 10～20%，如果数据较多，那么这种效率的折扣便不可忽视。

第 4 章 运算符

NASAL 内置了用于数值、字符串型变量的各种运算符。

4.1 算术运算符

用于数值的运算符有：“＋”，“－”，“*”，“/”，“＋=”，“－=”，“*=”，“/=”。

这些运算符主要用于数学运算，使用方法和 C++语言一样。其他的如开方、幂、三角函数等运算使用库函数的方式实现。非常遗憾，NASAL 没有提供方便的自加（++）和自减（--）运算符。

4.2 关系运算符

关系运算符有：“<”，“<=”，“>”，“>=”，“==”，“!=”。

关系运算符用于对两个数值型的变量或数字进行比较。NASAL 中没有 true 和 false。如果比较正确则返回 1，否则返回 0。注意，这些关系运算符不能对字符串进行比较，如果需要比较字符串，可以使用 NASAL 用于处理字符串的库函数 cmp( )。

4.3 逻辑运算符

逻辑运算符有：“and”，“or”，“!”。

逻辑运算符认为 0 和 nil 是假（false），其他为真，负数也是 true。

和关系运算符不同，and 和 or 的运算结果不是 1 和 0，而是和它的两个操作数有关。

```
a and b        # 如果 a 为 false，则返回 a，否则返回 b
a or  b        # 如果 a 为 true，则返回 a，否则返回 b
```

例如：

```
print(4 and 5);              # 5
print(nil and 13);           # nil
print(4 or 5);               # 4
print(nil or 5);             # 5
```

C++语言中的三元运算符

```
a ? b : c
```

在 NASAL 中可以这样实现：

```
(a and b) or c
```

“!”的结果只返回 0（false）或者 1（true）。

```
print(!nil);          # 1
print(!5);            # 0
print(!0);            # 1
```

在 FlightGear 中，可以用 getprop 函数取得一个属性值，但如果系统中没有该属性值或是由于手误打错的该属性的名字，就会返回 nil，这样就有可能导致下面的程序运算错误，这时可以用 or 来避免这种情况在下面的例子中，获取高度值，当返回一个非 0 值的时候，altitude 就会等于该值，否则返回 nil 或 0 时，altitude 变量就等于 0。

```
var altitude = getprop("/position/altitude-ft") or 0.00;
```

4.4 连接运算符

连接运算符有：“~”。

连接运算符用于字符串的连接，如果操作数为数字，NASAL 会将数字转换成字符串。

```
print("Hello " ~ "World");      # Hello World
print(0 ~ 1);                   # 01
```

4.5 优先级

从高到低的顺序：

```
!       - (取负数)
*       /
+       -
~
<       >       <=      >=      ~=      ==
and
or
```

在各种编程语言中，对运算符的优先级的处理都是很头疼的问题。阅读他人的代码时，可能也会花费一些时间来理解到底表达式中的哪部分优先运算。因此，最好的处理方法就是，不考虑优先级的问题，使用括号“( )”来标识高优先级的部分，这样做的好处就是代码更容易阅读，而且不会出现歧义。在优先级明显的表达式如 a + b*c 中，可以不使用括号。如果对优先级不是很明确，那么最好使用括号来标识。

第 5 章　基本语法

和其他的编程语言一样，NASAL 程序中最基本的单位是语句，按照程序的组织，一条一条的执行语句，这就是面向过程的程序运行方式。在面向过程的程序设计中，程序员必须根据程序任务的要求，写出一条条语句，安排好它们的执行顺序。

5.1　赋　值

NASAL 中最简单的是赋值语句。赋值是改变一个变量值的最基本的方法。

```
w = 100;
x = "hello";
array[0] = 1;
```

NASAL 可以同时给多个变量赋值，多变量同时赋值时，需要用使用括号把变量括在一起。NASAL 首先计算赋值号右边的变量的值，然后一一对应赋值给左边的变量。在进行多变量赋值时，赋值号左右两边的变量的个数必须相等，否则会出错。

5.2　选择结构

选择结构是改变程序运行流程的一种方式，根据表达式的计算结果，来判断程序的下一步应该往哪儿走。选择结构使用 if 和 else 配套使用，如果有多个条件需要判断，还可以用 else if 或 elsif。

```
var foo=1;
 if (foo == 1)
   print("1\n");
 else
   print("0\n");
print("this is printed regardless\n");
```

在进行条件判断时，需要注意的是要使用“==”，而不是用“=”，这个错误是很多具有多年编程经验的老手也难以避免的。因为如果使用 if (foo = 1)的话，这是一条正确的语句，并不会报错。但却会导致程序运行结果错误，这类错误是很难发现的。因此，建议进行条件判断时，可以将表达式反写，如 if (1 == foo)，即使由于手误，写成了 if (1 = foo)，这时就会发生语法错误而被系统发现。

```
if (1==2) {
  print("wrong");
  } else if (1==3) {
    print("wronger");
  } else {
    print("don't know");
  }
```

NASAL 中没有 switch 语句，当需要进行多条件判断时，就用 else if 或 elsif 吧。

```
# weather_tile_management.nas
if (code == "altocumulus_sky"){weather_tiles.set_altocumulus_tile();}
else if (code == "broken_layers") {weather_tiles.set_broken_layers_tile();}
 else if (code == "stratus") {weather_tiles.set_overcast_stratus_tile();}
```

```
 else if (code == "cumulus_sky") {weather_tiles.set_fair_weather_tile();}
 else if (code == "gliders_sky") {weather_tiles.set_gliders_sky_tile();}
 else if (code == "summer_rain") {weather_tiles.set_summer_rain_tile();}
 else if (code == "low_pressure") {weather_tiles.set_low_pressure_tile();}
 else if (code == "cold_sector") {weather_tiles.set_cold_sector_tile();}
 else if (code == "warm_sector") {weather_tiles.set_warm_sector_tile();}
 else if (code == "test") {weather_tiles.set_4_8_stratus_tile();}
 else ...
```

5.3 循环结构

1．基本循环

当需要对一些内容重复使用某些操作时，可以使用循环结构。NASAL 有四种循环结构 for、while、foreach 和 forindex。

循环结构的语法和 C++语言类似

```
for(var i=0; i < 3; i = i+1) {
 # 循环体
 }
while (condition) {
# 循环体
}
```

和 C++语言不同的是，NASAL 没有 do…while 循环。

foreach 循环有两个参数，第一个参数是变量名，第二个参数是个数组。每一次循环会返回数组中的一个元素，然后依次返回下一个元素。

```
myhash= {first: 1000, second: 250, third: 25.2 };
foreach ( i; keys (myhash)) {
  myhash[i] *= 2;
  print (i, ": ", myhash[i]);
}
```

在上面的例子中，keys(myhash)功能是利用 keys 函数返回一个数组，该数组包含了 myhash 哈希中的所有下标["first", "second", "third"]，然后就可以利用访问数组的方式来访问 myhash 中的元素了。

forindex 循环和 foreach 循环类似，不同的是每一次返回的不是数组的元素，而是数组的下标：0,1,2,3…。foreach 和 forindex 循环主要用于不定大小数组的操作。由于数组的大小是可以动态改变的，根据程序运行的需要可以增加或删除元素从而改变数组的大小。当需要对数组中的每个元素进行操作时，不知道数组的大小，从而不能确定循环的次数，当然有函数可以获取数组的大小从而赋给循环变量。但更方便的方式是直接使用 foreach 或 forindex 循环来做。

2．循环的使用

合理使用循环可以大大减少代码的编写量，从 1 循环加到 100 这种例子没有实际意义，我们以设置 FlightGear 中的面板的按钮来看看循环是如何使用的。

下面的代码的功能是给 CDU 面板的几个参数设置为点击（click）属性。

```
setprop("/controls/cdu/L1-type", "click");
setprop("/controls/cdu/L2-type", "click");
setprop("/controls/cdu/L3-type", "click");
setprop("/controls/cdu/L4-type", "click");
setprop("/controls/cdu/L5-type", "click");
```

我们要编写 7 行类似的代码来实现这个功能，虽然可以通过复制、粘贴的方法减少代码输入，但还是显得笨重，如果有类似的工作，还要再粘贴一次。想象一下，如果一个文件中，充斥着这样的代码，我们日后来查找或修改一个功能时，在满篇的雷同的代码中找到需要的一行是多么的痛苦。

从上面的代码可以看出，每一行的功能是一样的，都是设置 click 属性。不同的是参数的名字：L1-type、L2-type...。用提取相似点的方法可以写成下面的循环结构。

```
var lines = ["L1-type", "L2-type", "L3-type", "L4-type", "L5-type"];
foreach(var line; lines) {
setprop("/controls/cdu/"~line, "click");}
```

还可以把上面的代码放到一个函数中去，这样在别的地方执行 init_lines 函数就可以达到同样的效果。

```
var init_lines= func {
  var lines = ["L1-type", "L2-type", "L3-type", "L4-type", "L5-type"];
  foreach(var line; lines) {
  setprop("/controls/cdu/"~line, "click");}
 }
```

这个函数功能比较单一，只能设置 L1～L5，如果还想设置 R1～R5，或是设置其中的几个呢？可以利用函数的参数功能。函数的使用在后面的章节中有详细的介绍，这里只给出实现代码。

```
var init_cdu_types= func(what) {
  foreach(var w; what) {
  setprop("/controls/cdu/"~w, "click");}
}
```

这样就可以利用该函数随意设置了。

```
var lines = ["L1-type", "L2-type", "L3-type", "L4-type", "L5-type"];
init_cdu_types(lines);

var lines = ["R1-type", "R2-type", "R3-type", "R4-type", "R5-type"];
init_cdu_types(lines);
```

我们还发现，这些参数有个特点就是，名称类似，序号从 1 到 5 递增。这也可以用循环实现。

```
var create_with_suffix = func(prefix,start,end,suffix) {
  var result = [];
  for (var i=start;i==end;i+=1) {
    var s= prefix ~ i ~ suffix;
    append(result,s);}
  return result;
}
var left = create_with_suffix("L",1,5,"-type");
var right = create_with_suffix("R",1,5,"-type");

init_cdu_types(left);
init_cdu_types(right);
```

3．定时器

一般来说，上面的 for、while、foreach 和 forindex 四种循环通常用于处理小型的任务，因为这四种循环会阻断程序的正常流程，如果循环体非常大，循环次数非常多，就会在循环这个阶段耽搁较长时间，从而影响其他任务的正常运行。或者有时候，我们需要一种函数一直执行，如变量监控，消息循环等，这些函数有时在整个程序运行周期内都运行。功能可能不是很复杂，比如只是监控一个变量的变化，

或是监控端口有没有消息等等，但是需要一直执行，并且不能阻断主程序的运行。这时，就可以使用 settimer 定时器。定时器和 Windows 系统的定时器功能类似，设定一个时间后，系统就会从从 settimer 函数开始运行时计时，到达设定时间后自动触发函数执行。定时器的优点在于不会阻断程序的正常流程，就好像多线程一样，自动开辟一个新的线程工作，和 windows 不同的是，windows 下的 settimer 函数是定时器启动，一直在执行，如果需要关闭还需 killtimer，而 NASAL 的 settimer 函数相当于延时执行，即延迟一段时间后再执行，而且只执行一次。我们可以让函数每隔一段时间后调用自己一次来实现这种特殊的定时器循环结构。

```
var loop = func {
    print("this line appears once every two seconds");
    settimer(loop, 2);
}
loop();        # 开始循环
```

settimer 函数的参数有两个，第一个参数是调用的函数名，第二个参数是时间间隔（单位为秒）。settimer 启动函数后只执行一次，由于每次函数执行时都调用了 settimer 函数，因此，该函数执行完毕后每隔一段时间，都会再执行一次，一直运行下去，如何使函数停止运行呢？我们可以在外部设定一个变量作为循环标志，如果该标志为 1，函数就继续执行，否则终止循环。让函数每次运行时检测这个变量，就可以了。改造的函数如下。

```
var running = 1;
var loop = func {
    if (running) {
        print("this line appears once every two seconds");
        settimer(loop, 2);
       }
}

loop();        # 开始循环
...
running = 0;   # 终止循环
```

但是上面的这个程序有严重的隐患。仔细看一下发现，我们设想通过给 running 变量赋值为 0，从而终止了程序的循环。假设循环体比较大，运行一次时间较长，当这次循环还没有结束时，我们在另外一个地方通过给 running 变量赋值为 0，然后马上再赋值为 1，启动一个新的循环。这时，由于前一个循环执行时间较长，没来的及反应 running 变量的变化，当循环结束以后，再次循环时，发现 running 已经为 1 了，于是，继续执行。也就是说，我们设想通过给标志变量赋值 0 来终止循环的想法失效了。失效的原因是我们没有预料到循环体循环一次到底用多长时间，如果我们知道循环体执行一次需要 1 秒钟，那么我们在终止循环时，就要保持标志变量为 0 持续 1 秒钟。循环体的运行时间和多种因素有关，例如机器硬件、操作系统、系统运行程序等等。所以，这个时间是无法预测的，我们只能再换一种方法，既能有效的结束前一个循环体，又能迅速的启动新的循环体。

方法总是有的，可以给每一个循环体分配一个 ID，然后让函数检查这个 ID 是否是系统要求的 ID，如果是，就继续执行，否则，自行终止。当需要启动新的循环体时，把 ID 加 1 即可。由于 ID 变了，因此前一个循环体的 ID 和现在的 ID 不一致，自行终止。而新的循环体可以顺利执行。这样，我们就不用费功夫来终止前面的循环体了，前面的循环体进入下一次循环时就会自己结束自己。修改后的代码如下。

```
var loopid = 0;
var loop = func(id) {
    id == loopid or return;          # ID 检测，如果对则继续，否则，函数终止。
    ...
    settimer(func { loop(id) }, 2);   # 用 ID 来调用自己
```

```
}

loop(loopid);          # 开始循环
...
loopid += 1;           # 结束其他的循环
loop(loopid);          # 开始一个新的循环，这两行可简写为 loop(loopid += 1);
```

第 6 章　函　数

6.1　基本概念

函数就是一段代码，这段代码通常具有通用性，并且会重复使用。

函数有两种用途：一是完成指定的任务，如初始化参数，打印一个变量等。这种情况下函数作为调用语句使用；二是计算并返回值，也就是数学意义上的函数，如 sin(x)等，这种情况下函数作为赋值语句的表达式使用。

函数是改变程序流程的一种方式。通常情况下，程序按顺序逐条执行，遇条件判断或循环结构时会根据条件改变程序流程。如果调用函数时，程序会中断正常流程，跳转到函数体执行代码，函数执行完毕结束后，再返回到原流程继续执行。

6.2　函数的声明

使用函数时，必须要首先声明定义函数，和 C++语言不同，C++语言中函数的声明和实现是分开的，而在 NASAL 中函数在声明时就要给出实现的代码。使用 func 关键字来声明函数。下面的代码就定义了名为 log_message 的函数，该函数为空函数，不实现任何功能。

```
var log_message = func {};
```

函数体在 func 后面的{　}中编写，函数体的代码可以遵循 NASAL 的基本语法规则任意组织。函数也可以嵌套使用，在一个函数体中可以调用另外一个函数。

6.3　函数的参数

用于完成任务的函数可以没有参数，但有时用于计算的函数就需要指定参数。比如计算 3 的正弦值，就需要写成 sin(3)，把 3 这个数值作为参数传送给 sin 函数。

函数是有默认参数的，当不显式指定参数名称时，NASAL 默认创建一个名为 arg 的数组来保存参数。使用下标来依次访问每一个参数。

```
var log_message = func {
  print(arg[0]);}
log_message("LOGOUT:");          # 输出 LOGOUT:
```

当函数需要参数时，一般要显式的声明参数，而不用默认参数。如上面的函数可以修改为：

```
var log_message = func(msg) {
  print(msg); };
```

函数的参数可以指定一个默认值，这样当调用该函数时，如果没有指定该参数，则该参数用默认值代替。

```
var log_message = func(msg="error") {
  print(msg); };
```

如果函数有多个参数，有的参数有默认值，有的没有，有默认值的参数必须放到后面。

```
var log_message = func(msg="error", line, object="ground"){};  # 错误
var log_message = func(line, msg="error", object="ground"){};  # 正确
```

NASAL 支持可变参数，当不确定参数个数时，就可以使用这种方式，其实，在前面已经遇到过可变参数了，函数的默认参数 arg 就是一个可变参数。arg 是一个数组，数组的大小由参数的个数决定。可以用参数名来代替 arg，只需在参数名后加一个“...”符号作为可变参数标志。

```
listify = func(elements...) { return elements; }
listify(1, 2, 3, 4); # 返回一个数组 [1, 2, 3, 4]
```

当传入函数的参数数目小于要求的个数时，程序会出错。但是允许传入的参数数目多于函数要求的个数。在一一对应的给参数赋完值后，多余的参数依次按顺序保存到 arg 数组中。

6.4 命名参数的函数调用

在编写代码过程中，有时我们会忘记参数的顺序。这时，我们可以使用这种调用方式，把值送给指定的参数，就像为变量赋值一样，即使顺序颠倒了，也无所谓。

```
var lookat =  func(heading=0, pitch=0, roll=0, x=nil, y=nil, z=nil)
{ #函数体 }
lookat(180, 20, 0, X0, Y0, Z0);     # 使用逗号分隔参数
lookat(heading:180, pitch:20, roll:0, x:X0, y:Y0, z:Z0); #使用参数名指定参数
lookat(x:X0, y:Y0, z:Z0, heading:180, pitch:20, roll:0); #顺序不一致也可以
```

需要注意的是，如果不使用参数名来传递参数值，那么所有的参数都不能指定参数名，必须按顺序用逗号分隔开。如果用参数名来传递参数值，那么所有的参数都必须指定参数名。不能只有几个参数用参数名，其他用逗号隔开，即使顺序一致也不行。

6.5 函数的返回值

函数体执行结束后要返回到原程序流程，函数都有一个返回值。当不显示指定返回值时，函数会返回最后一个表达式的值。空函数返回 nil，也可以用 return 来声明返回一个值，如上面的代码中返回 elements 数组。返回值根据需要可以是数值，字符串，数组等等。

return 还可用于中断函数的运行，提前返回到原程序流程中去。

6.6 函数的嵌套

这里说的嵌套并不是函数的嵌套调用，而是函数的嵌套定义。也就是在函数体中定义一个函数。

```
var calculate = func(param1,param2,operator) {
  var add = func(p1,p2) { return p1+p2; };
  var sub = func(p1,p2) { return p1-p2; };
  var mul = func(p1,p2) { return p1*p2; };
  var div = func(p1,p2) { return p1/p2; };
```

```
 if (operator == "+") return add(param1,param2);
 if (operator == "-") return sub(param1,param2);
 if (operator == "*") return mul(param1,param2);
 if (operator == "/") return div(param1,param2);
}
```

注意，这里的 add，sub，mul 和 div 四个函数是在函数体中定义和使用的，并且也只能在这个函数体内使用，因为根据规则，这些函数在函数体外是不可见的。

6.7 函数的重载

在 NASAL 中，函数是不能被重载的，同样，操作符也不能被重载。这时因为 NASAL 的变量是没有类型标志的。在声明变量时，不指定变量是数值、字符串还是数组，用一个 var 就可以了。函数声明时，参数也无法指定类型。所以不能通过指定参数类型来实现函数的重载。但我们可以用另外一个方法来变相的实现函数的重载。

NASAL 有个 typeof 系统函数，该函数可以返回变量的类型，返回值有 nil（空）、scalar（数值型）、vector（数组型）、hash（哈希型）、func（函数型）、和 ghost（句柄型）。这样，在函数体中，使用 typeof 函数来判断参数的类型，然后根据不同的类型指定对应的操作，就可以实现函数的重载了。利用下面的代码可以实现多种类型变量的乘法操作。在 3D 空间运算中，有大量的矢量或矩阵的运算，使用这种方式可以很轻松的实现。

```
var multiply2 = func (params) {
 if (typeof(params)=="scalar") return params*arg[0];
 if (typeof(params)=="vector") return params[0]*params[1];
 if (typeof(params)=="hash")   return params.x*params.y;
 die("cannot do what you want me to do");
}

multiply2( 2,6 );            # 两个数值相乘，参数为 2 个数值
multiply2( [5,7] );          # 数组中的两个数相乘，参数为 1 个数组
multiply2( {x:8, y:9} );     # 哈希中的两个元素相乘，参数为 1 个哈希
```

6.8 动态生成函数

由于返回值可以是一个函数，因此，我们可以利用函数来生成函数。就像工厂一样，一个函数是机器，可以根据实际情况，动态产生出所需的各种函数。

```
var i18n_hello = func(hello) {
   return func(name) { # 返回一个函数
     print(hello,name);
   }
  }

  # 创建 3 个函数
  var english_hello = i18n_hello("Good Day ");
  var spanish_hello = i18n_hello("Buenos Dias ");
  var italian_hello = i18n_hello("Buon giorno ");
```

```
# 调用刚创建的函数
english_hello("FlightGear");      # 输出: Good Day FlightGear
spanish_hello("FlightGear");      # 输出: Buenos Dias FlightGear
italian_hello("FlightGear");      # 输出: Buon giorno FlightGear
```

6.9 代码简化技巧

有时，我们可以利用函数来简化代码的输入。例如，在进行哈希赋值时，需要将哈希中的每个元素逐一赋值过去。通常这样做：

```
var l = thermalLift.new(ev.lat, ev.lon, ev.radius, ev.height, ev.cn, ev.sh,
ev.max_lift, ev.f_lift_radius);
```

当参数非常多时就很容易发生拼写错误。编写一个简单的函数来实现这个功能就可以在后续的工作中减少这种不必要的麻烦。利用函数简化后的代码更加简洁，清晰。

```
thermalLift.new_from_ev = func (ev) {
  thermalLift.new(ev.lat,  ev.lon,  ev.radius,  ev.height,  ev.cn,  ev.sh,
ev.max_lift, ev.f_lift_radius);}

var l = thermalLift.new_from_ev(ev);
```

下面的例子用于对两个节点下的参数进行复制。

```
t.getNode("latitude").setValue(f.getNode("latitude").getValue());
t.getNode("longitude").setValue(f.getNode("longitude").getValue());
t.getNode("altitude").setValue(f.getNode("altitude").getValue());
```

也可以用下面的代码来实现。

```
var copy = func(t,f,path){
 t.getNode(path).setValue(f.getNode(path).getValue());}

copyNode(t,f,"latitude");
copyNode(t,f,"longitude");
copyNode(t,f,"altitude");

#或者这样
foreach(var p; ["latitude", "longitude"," altitude"])
    copyNode(t,f,p);
```

第 7 章　面向对象程序设计

面向对象程序设计（OOP）是编程史上的一次重要的革新，具有里程碑的意义。OOP 真正把抽象的程序代码和现实世界统一了起来。这样，程序员就可以用正常理解生活中的事物一样来编程，而不是用晦涩的代码来试图表现现实世界。现在，很多种程序语言支持 OOP，如 C++，JAVA，C#等，就连传统的 C 语言也被苹果公司丰富出了一个 Objective-C，用于 iOS 平台的程序设计。

NASAL 并不是一个高级语言，只是一个简单的脚本引擎，但麻雀虽小，五脏俱全，NASAL 虽然不

像其他的语言完整而全面的支持 OOP，但也用自己短小精悍的代码实现了 OOP 的基本特性。

7.1 类

类是 OOP 的基本概念。类就是对一类事物的抽象，如人、车、动物等等。类具有属性，比如人有性别、身高、体重等属性。类还具有方法，也就是类函数。比如人可以走路，可以工作等。类是一个抽象的概念，使用类时，需要实例化，也就是需要派生出一个对象。如定义一个人的性别、身高、体重等属性后，这个人就被建立了，以后便可以让这个人走路、工作等。

实际上，类就是封装了一系列变量和函数的集合。变量就是类的属性，或称为成员变量；函数就是类的方法，或称为成员函数。

哈希这种类型的变量就可以用于表示类。

```
var position3D = {
 x:100.00,
 y:200.00,
 z:300.00,
 hello: func() { print("Hello world"); }
};
```

上面的代码就用哈希这种变量方式，定义了一个 position3D 的类，这个类有 x，y，z 三个属性，表示三维的空间坐标，还有一个 hello 函数用于输出一段文字信息。

7.2 类的自引用

在类内部的成员函数中，有时需要访问自己的成员变量。但如果类的成员变量的名字和外部全局变量的名字一致，就会出现歧义。在 C++中，用 this 指针来表示自己，在 NASAL 用 me 这个变量来表示自己。因此，在类内部的函数调用自己的成员变量时，最好都加上 me 这个标志，以表示使用的是类自己的变量。在调用自己的成员函数是也要加上 me，否则会出现找不到函数的错误。

```
var value = "test";

var data = {
   value: 23,                          # 成员变量 value 和全局变量同名
   write1: func { print(value); },     # 隐藏全局 value
   write2: func { print(me.value); }
};

data.write1();     # 输出 test
data.write2();     # 输出 23
```

7.3 类的构造

使用 OOP 的方法进行程序设计时，我们不直接使用类，而是把类实例化后再使用。在前面的例子中，虽然 position3D 是哈希变量，但通常不把 position3D 当做一个变量，而要当成一个类。使用这个类作为模板创造一个对象然后使用。在 C++中，我们用 new，而在 NASAL 中，我们用 parents 来从类中继承一个对象。下面的代码就利用 parents 生成了一个 test 对象，该对象从 position3D 继承而来，因此，拥有和 position3D 相同的成员变量和函数。

```
var test = { parents:[position3D] };
print(test.x);        # 输出 100.00
test.hello();         # 输出"Hello world"
```

为了和 OOP 统一，我们通常为类设定一个构造函数，我们借鉴了 C++，用 new 作为构造函数名。

```
var position3D = {
 x:100.00,
 y:200.00,
 z:300.00,
 new: func() { return { parents:[position3D] }; }
};

var test = position3D.new();
```

还可以创建一个带有参数的构造函数，例如，我们可以给构造函数中加入参数，用于初始化成员变量。

```
var position3D = {
 x:100.00,
 y:200.00,
 z:300.00,
 new: func(val) { return { parents:[position3D] x:val}; }
};

var test = position3D.new(400);
print(test.x);           # 输出 400
```

上面的例子中，创建 new 构造函数时，加了一个参数，用于对 x 这个成员变量进行初始化。使用这个带参数的构造函数创建新对象时，发现，新对象的 x 已经被初始化为 400 了。

7.4 类的实例

OOP 对类的使用的基本原则是：类是一个模板，不能直接使用，如果使用必须实例化。但是在 NASAL 中，类是使用哈希这种数据结构存在的。类一旦定义，就客观存在了，即使没做实例化也客观存在了，可以像哈希型变量一样任意操作，如果使用不当就会出现错误的结果。

```
var class = {
  value:100,
  write:func { print(me.value); },
  new:func(val=0) { return {parents:[class], value:val }; }
  };
```

上面定义了一个 class 的类，有一个成员变量 value，还有两个成员函数，一个是 write 用于输出 value 的值，一个是 new，作为构造函数出现。

使用 class 类正确的做法是，用 new 函数生成新的对象，然后对生成的对象操作，代码如下：

```
var instance1 = class.new();
var instance2 = class.new(123);

instance1.write();    # 输出 0
instance2.write();    # 输出 123
```

如果我们错误的使用赋值号，就会出现错误的结果。

```
var bad_instance1 = class;   # 错误的使用
var bad_instance2 = class;   # 错误的使用

bad_instance1.value = 123;   # class.value = 123
bad_instance2.value = 456;   # class.value = 456

bad_instance1.write();      # 输出 456, 而不是 123
```

在上面的例子中，我们本想生成两个实例，但用赋值号的结果是把 class 这个父类直接引用过来了。由于 class 这个对象是客观存在的，因此，这句话没有语法错误。并且这种赋值，不会把父类的所有成员都复制过去，而是指向了一个到父类的引用，相当于 C++中的指针。赋值后，bad_instance1，bad_instance2 和 class 三个变量都指向了同一个地方。因此对任何一个变量的修改，都会反应到 class 类上去。所以会得到错误的结果。因此在使用类进行程序设计时，一定要注意实例化问题。

不同的实例之间，他们的成员变量和函数是相互独立的，不会互相影响。我们继续上面 class 类的例子。如果我们修改了 class 类的 write 函数。

```
class.write = func { print("VALUE = " ~ me.value); }

var instance1 = class.new();
var instance2 = class.new(123);

instance1.write();    # 输出 VALUE = 0
instance2.write();    # 输出 VALUE = 123
```

我们对基类的修改，导致了所有派生类的函数都发生了变化。

如果我们只修改其中一个实例的成员函数，那么对另一个实例的成员函数没有影响。

```
instance1.write = func { print("VALUE = " ~ me.value); }

var instance1 = class.new();
var instance2 = class.new(123);

instance1.write();    # 输出 VALUE = 0
instance2.write();    # 输出 123
```

7.5 类的析构

NASAL 有自己的垃圾回收器，所以在声明变量时，不需要申请内存空间。变量不再使用了，内存空间自动回收再利用，不需要程序员做任何操作。在 C++中，程序员最头疼的就是空间的回收，C++中的指针虽然给各种算法提供了方便，但使用不当极易造成内存溢出。C++中的类析构函数主要就是做这些善后工作的，当不再使用时，要将申请的空间及时归还。并且还可以做一些其它的相关工作。

NASAL 不需要考虑空间回收问题，所以 NASAL 中的类是没有析构器的。但是在 FlightGear 中，虽然资源可以被自动回收，但 listener 不会被自动删除，还有一些循环体也不会自动终止，这些工作最好都由类的析构器来做。所以建议，手动给类增加一个析构函数，为了和 new 对应，建议使用 del 作为析构函数的函数名。析构函数中做一些相关的处理，然后当类结束后，手动调用 del 函数，以释放一些被占用的资源。

7.6 类的继承

类有个很重要的概念就是继承，我们可以从父类中继承一个子类，这个子类拥有父类所有的特性，并且子类还可以增加其他的特性。例如，我们可以从“人”这个类中，继承下来一个“男人”的类。“男人”这个类除了拥有人的特征外，还具备男性独有的特征。在 NASAL 中，可以使用 parents 进行类的继承。在前面的内容中，类的实例化的过程，其实就是类的继承。

NASAL 仅实现了简单的继承，并没有 OOP 中高级的封装特性，例如在 C++中，成员变量或函数可以设定为私有或公有的，继承也分 public、private 和 protect 等。在 NASAL 中，成员变量和函数是全部开放的。因此可以认为，NASAL 中的继承都是 public 的，并且成员变量和函数也都是 public 的。

```
var parent_object = { value: 123 };
var object = {
   parents: [parent_object],
   write: func { print(me.value); }
};

object.write();    # 输出 123
```

上面的例子中，object 从 parent_object 继承而来，从而拥有的 parent_object 的成员变量 value。而且 object 又增加了自己的 write 函数，write 函数中使用了成员变量 value。这是完全合法的，因为虽然 object 中没有定义 value 这个成员变量，但是由于 object 由 parent_object 继承而来，所以只要是 parent_object 的成员变量，object 也一并继承过来了。

这种继承是单继承，也就是说，object 只有一个父类。OOP 中还有一种多继承，子类可以多个父类中继承而来，从而拥有所有父类的所有特征。

```
var A = {                                        # class A
   new: func {
      return { parents: [A] };
   },
   alpha: func print("\tALPHA"),
   test:  func print("\tthis is A.test")
};

var B = {                                        # class B
   new: func(v) {                                # 带参数构造函数
      return { parents: [B], value: v };
   },
   bravo: func print("\tBRAVO"),
   test:  func print("\tthis is B.test"),
   write: func print("\tmy value is: ", me.value)
};

var C = {                                        # C 由 A 和 B 继承而来
   new: func(v) {
      return { parents: [C, A.new(), B.new(v)] };
   },
   charlie: func print("\tCHARLIE"),
   test:    func print("\tthis is C.test")
```

```
};

print("A instance");
var a = A.new();
a.alpha();            # 输出 ALPHA

print("B instance");
var b = B.new(123);
b.bravo();            # 输出 BRAVO
b.write();            # 输出 my value is: 123

print("C instance");
var c = C.new(456);
c.alpha();            # 输出 ALPHA
c.bravo();            # 输出 BRAVO
c.charlie();          # 输出 CHARLIE
c.test();             # 输出 this is C.test。A.test()和 B.test()被覆盖
c.write();            # 输出 my value is: 456
```

上面的例子展示了 NASAL 中的多继承。其中 C 类由 A 和 B 共同派生出来，C 拥有 A 和 B 所有的特性。包括所有的成员变量和成员函数。所以，当调用 c.alpha()函数时，调用的是继承自 A 类的函数。调用 c.bravo()时，调用的继承自 B 类的函数，调用 c.charlie()时，调用的是自己的函数。调用 c.test()时，由于 A 类和 B 类都有 test 函数，并且 C 类也有 test 函数，所以 C 的 test 函数会覆盖 A 和 B 的 test 函数，从而调用自己的 test 函数。类在进行成员变量或函数的调用时，首先使用自己的，如果自己没有，就用父类的。如果 C 类没有定义 test 函数，而父类 A 和 B 都有 test 函数。那么就按照继承的顺序 parents: [C, A.new(), B.new(v)]，优先调用 A 类的 test 函数。

7.7 类的虚变量

在 C++中，有一种成员函数叫虚函数。这种函数在类的定义中只是声明一下，没有具体的实现代码。具体的代码在派生类中实现。这样，使用基类的指针就可以访问派生类的函数。NASAL 不支持虚函数，但是在 NASAL 中，由于类是一种特殊的变量，而派生又完整复制了父类的特征。所以，我们可以在父类中直接访问子类的变量。

```
var class = {
  write:func { print(me.value); },
  new:func() { return { parents:[class]}; }
  };
var instance = {
  value:100,
  parents:[class]
  };

instance.write();     #输出 100
```

从上面的例子中可以看出。instance 从 class 类继承而来，然后又增加了一个 value 成员变量。然后调用父类的 write 函数来打印 value 变量的值。我们发现，其实在 class 类中是没有 value 这个成员变量的，然而 class 类的成员函数 write 仍然可以调用一个并不存在的变量，这就是虚变量。也就是说，父

类可以在没有任何声明的情况下，直接调用子类的成员变量。这很方便，当然也很危险，直接调用父类的 write 函数，如 class.write()就会出错，因为父类没有 value 这个变量。这得益于 NASAL 灵活的架构和内存管理方式。当构造一个类的时候，实际上就是开辟了一块空间来保存这个对象，因为类就是哈希变量。如果使用了继承，NASAL 首先在内存中开辟一段空间，把所有父类的特征放到这个空间里，然后把自己独有的特征也放到这个空间里，然后就和父类没有关系了，这时，虽然是调用父类的成员函数，但由于父类的特征已经完全复制过来，所以实际上调用自己内存空间里的成员函数，自己的成员函数访问自己的变量当然没有问题。然而用父类来调用 write()函数，就会在父类的内存空间里调用函数，由于父类的内存空间里没有 value 变量，所以，当然会出错。

NASAL 支持虚变量但不支持虚函数，在父类中调用子类的函数会出错。

7.8 类的封装

封装是一个形象的比喻。就是我们把类打包起来，只保留几个接口给外面，方便信息交换。其余不需要外面了解的东西全部“隐藏”起来。这样做的好处就是非常适合团队开发，每人开发一个模块，然后统一好接口函数及参数，具体的内部实现由个人自主完成，不需要考虑别人的影响。由于不直接访问成员变量，而是改由接口来访问成员变量，因此，模块的修改只要不涉及到接口函数，程序的其他部分的代码不需要做任何调整就能马上使用。就好像电脑的主板提供了 CPU，硬盘，内存等接口，这些接口有统一的规范，只要符合这个规范，无论是哪个厂商生产的产品，插到主板上马上就可以使用。

下面的代码中为 cloud 设置了经纬度属性。这样做当然可以，但如果后来经纬度的变量名改成了 longitude 和 latitude，那么这里的代码也要做相应改变，程序的其它地方呢？如果有的话，也要做一番调整，非常麻烦。

```
cloud.lon=43.22;
cloud.lat=10.22;
```

我们可以改用接口来实现，而不是直接修改成员变量，这样的代码更加容易维护。

```
cloud.setPos(lon, lat);
```

7.9 成员函数的回调调用

NASAL 中有 listener 和 timer 两种函数，这两种函数的特点就是可以根据特定条件去触发另一个函数。listener 用于属性监控，timer 用于定时触发。对于正常的调用没有问题。但是如果类中的成员函数使用了这种回调调用方式，就有可能出现问题。

```
var Manager = {
  new: func {
  return { parents: [Manager] };
  },
  start_timers: func {
    settimer(do_stuff, 5);       # 语法错误: do_stuff 不在可见范围内。
    settimer(me.do_stuff, 5);    # 逻辑错误: do_stuff 执行时，me 不知道是谁
    settimer(func me.do_stuff(), 5);    # 正确调用
    setlistener("/sim/foo", func me.do_stuff());  # 正确调用
    },
  do_stuff: func {
    print("doing stuff");
    },
```

```
};


var manager = Manager.new();
manager.start_timers();
```

上面的例子中，在 start_timers 成员函数中使用 settimer 调用了 do_stuff 成员函数。第一种调用是语法错误的，因为根据名字空间的可见范围，在 start_timers 函数空间中，没有 do_stuff 函数，因此属于语法错误。第二中调用语法没有问题，但运行时有可能出问题，我们来分析一下：该函数利用 settimer 函数定时在 5 秒钟后调用 do_stuff 函数，这就有了 5 秒的时间间隔，在这个时间间隔内如果 manage 出现了变化，假设修改了 do_stuff 函数，那么到预定时间后，开始执行 do_stuff 函数时，执行的是原来的函数还是修改后的函数呢？答案是原来的函数！我们可以做个实验，在 NASAL 的调试窗口中，第一窗口输入上面的代码，第二个窗口输入下面的代码：

```
manager.do_stuff = func { print("new doing stuff"); };
```

然后切换到第一个窗口，点击“Execute”按钮，执行代码，发现 5 秒后输出“doing stuff”。然后再点击按钮执行，5 秒内，马上切换到第二个窗口，执行第二个窗口的代码。等到 5 秒后，发现还是输出“doing stuff”，当然，如果再次调用 manager.start_timers()的话，会发现输出信息已经变成了“new doing stuff”了。这就导致我们得到了一次错误的输出，原因就在于我们在 settimer 设定的周期内对函数的修改没有反应出来。

为了避免这种情况发生，因此，建议使用 settimer(func me.do_stuff(), 5)这种调用方式，这种调用方式是安全的，不会出现第二种调用的问题。使用 setlistener 时也要使用这种安全的调用方式。

第 8 章　名字空间

当开发的项目规模大到一定程度的时候，变量命名危机的问题就凸显出来了。有时候，当我们想调用一个变量的时候，可能会忘记了该变量如何拼写。或是由于我们对变量的命名不规范，导致后来看到一个变量不清楚这个变量是干什么用的。还有当我们想声明一个新变量时，这个变量名可能已经被占用了，或是多人合作开发的时候，别人已经占用了这个名字。这时，对变量进行合理规划安排就显得十分必要。

8.1　可见范围

变量和函数一旦声明，就有它自己的可见范围。有时我们的代码执行时，系统提示没有定义的变量或函数时，就说明我们没有考虑到变量或函数的可见范围。

NASAL 代码的基本结构就是语句加函数。因此，在文件中，在最外层声明的变量和函数都是全局的。这些变量和函数可以在任何地方被访问。只是不使用 var 声明的变量和函数是真正的全局可见，而使用 var 来声明的话，只是在该文件内是可见的。并且，使用时，没有 C++语言先声明再使用的规矩。无论在代码的任何地方，只要有声明，就可以使用，真正做到了“一处声明，到处使用”，方便是方便，但也给不规范的程序员制造了不少麻烦。比如，我们有时想要知道某一个变量在何处声明的，如果代码不规范，就很麻烦。因此，建议还是保留 C++严谨的编程习惯，具有某些共性的变量在一起一并声明，或是放到文件头，或是放到一个单独的文件中，这样以后在代码维护时，非常方便。

在函数内部声明的变量和函数，只在这个函数内部可见，在函数外部是不可见的。但是，函数内部声明的变量和函数可以和全局的变量名字一样，会代替全局的变量和函数，不过只是暂时代替。在这个函数内部，如果调用了变量和函数的话，会调用函数自己定义的变量和函数，而不用全局的。但是其他的函数如果调用的话，还是调用全局的，没有影响。也就是说，函数内部定义的变量和函数如果和全局

的发生了命名冲突，会在函数内部隐藏掉全局的，而使用自己定义的，并且所做的任何操作，不会影响到全局。

```
var x = 1;
var show = func{
  var x = 2;       # 隐藏掉全局的 x = 1
  x += 1;
  print(x);
  };

show();            # 输出 3
print(x);          # 输出 1
```

8.2 名字空间

NASAL 中的名字空间其实只有全局空间和函数空间两种。全局的变量和函数在任何地方都可以调用，函数内部的变量和函数只能在函数内部使用。函数可以嵌套，所以函数空间也可以嵌套。

但有时，这些空间仍然不够用。例如，我们定义了一个类，这个类有一些属性，这些属性只有这个类才用的着。其他的地方不需要也最好不要随意访问，以免造成不必要的错误。这种情况下，把这些属性放到类空间中比较好。但是遗憾的是，连类都是用哈希这种特殊的变量来模拟的，又哪里来的类空间呢？所以只能变相来实现。

```
var wp1 = 0;
var wp1alt = 0;
var wp1dist = 0;
var wp1angle = 0;
var wp1id = "";

var wp2 = 0;
var wp2alt = 0;
var wp2dist = 0;
var wp2angle = 0;
var wp2id = "";
```

上面的代码中定义了两个航路点，其中每个航路点都有编号、高度、距离、角度、ID 信息。如果用这种方式来声明的话，每一个参数都是一个全局变量，不仅占用了全局空间，而且还暴露在全局中，其他的函数有可能因为拼写错误而意外修改这些参数，非常危险。如果把 wp1 做成一个哈希，所有的参数作为下标，就可以用下面的代码来代替。

```
var wp1 = {};
wp1.alt = 0;
wp1.dist = 0;
wp1.angle = 0;
wp1.id = "";

var wp2 = { alt:0, dist:0, angle:0, id: ""};
```

这样，这些属性就成了航路点的专用属性了，在全局空间也仅有 wp1 和 wp2 两个变量。当然还可以定义一个类，wp1 和 wp2 由这个类派生而来，这样就可以方便的处理更多的航路点。

8.3 模块空间

在 FlightGear 中使用 NASAL 时，需要在对应的飞机文件中加载进去，通常是在-set.xml 文件中进行加载。flightGear 在加载该 XML 文件时，为 nasal 区段的每个模块建立了一个模块空间，并分别把每个模块下的 nas 文件加载到对应的模块空间中去。

```
<nasal>
   ...
   <moduleA>
     <file>path/to/file1.nas</file>
     <file>path/to/file2.nas</file>
   </moduleA>
   <moduleB>
     <file>path/to/file3.nas</file>
   </moduleB>
 </nasal>
```

上面的例子中，我们在 XML 文件中加载了 3 个 nas 文件，并且分别加载到了 2 个模块中。其中 file1.nas 和 file2.nas 加载到 moduleA 模块中，file3.nas 加载到 moduleB 模块中。对于 3 个文件中的变量和函数，我们应该用下面的方式来调用。file1 和 file2 用 moduleA 调用，file3 用 moduleB 调用。

```
moduleA.varName;          # file1 和 file2 中的变量
moduleB.varName;          # file3 中的变量
```

第 9 章　异常处理

程序运行过程中可能会出现各种意外的情况，当碰到这些情况时就需要进行异常情况的处理。例如，当我们编制一个除法函数时，有可能会出现除以 0 的情况。这种情况就是一个异常，需要我们处理。再比如，我们使用网络发送数据，突然网络故障，网络通信终端，这也是一个异常。die 函数就可以退出函数，并抛出一个异常。

```
var divide = func(a, b) {
   if (b == 0)
      die("division by zero");
   return a / b;     # 如果 b == 0，这一行不执行
}
```

在 FlightGear 经常用到的 getprop 函数就进行了异常处理。如果这样使用 getprop("/4me")，就会产生一个异常："name must begin with alpha or '_'"。这样就可以防止函数接收非法的路径。

使用下面第一种方式正常调用函数时，如果产生异常，函数就会自动退出。第二种方式使用 call 函数来调用函数，当函数产生异常时，call 就会捕捉到异常，并接收 die 函数送出的异常信息，填充到 err 数组中，然后我们判断 err 数组的大小，就可以判断函数是正常执行完毕，还是异常退出了。

```
var value = getprop(property);
var value = call(func getprop(property), nil, var err = []);

if (size(err))
   print("ERROR: bad property ", property, " (", err[0], ")");   # 异常
```

```
else
    print("value of ", property, " is ", value);
```

call 函数有三个参数，第一个参数是要执行的函数；第二个参数是执行函数所需要的参数，可以是 nil；第三个参数是一个数组，用于接收异常。

并且，die 函数的输出不一定是一个字符串，也可以是一个变量，一个哈希等。下面的例子就定义了一个异常类，当函数执行出现异常时，利用这个类生成一个异常哈希并抛出。

```
var Error = {                    # 异常类
    new: func(msg, number) {
        return { parents: [Error], message: msg, number: number };
    },
};

var A = func(a) {
    if (a < 0)
        die(Error.new("negative argument to A", a));      # 抛出异常哈希
    return "A received " ~ a;
}

var B = func(val) {
    var result = A(val);
    print("B finished");      # 如果 A 抛出异常，这一行并不执行。
    return result;
}

var value = call(B, [-4], var err = []);

if (size(err)) {                          # 检查有无异常
    print("ERROR: ", err[0].message, "; bad value was ", err[0].number);
    die(err[0]);
} else {
    print("SUCCESS: ", value);
}
```

第 10 章　FlightGear 应用

NASAL 要发挥作用，最终还是要在程序中，虽然 NASAL 是开源的，各种软件都可以利用它，但目前，它只有一个客户，就是 FlightGear。所以，FlightGear 才是 NASAL 的舞台。当然，既然代码是给 FlightGear 用的，那么在编写代码时就需要注意一些 FlightGear 的相关问题。

10.1　创建脚本

创建脚本其实就是编写 NASAL 代码，用编辑器编辑好代码后，保存为*.nas 文件，然后放到合适的地方，并告诉 FlightGear 什么时候来执行。

按照 FlightGear 的约定，通常有以下几种脚本。

1．飞机脚本

这些脚本通常和某种特定的飞机相关，也就是每种飞机会有自己独特的一套系统，来控制飞机的各种动作、响应等等。这些脚本通常放置在对应飞机的文件夹中，然后通过该飞机对应的-set.xml 文件进行加载，在这个 XML 文件中有一个<nasal>区段，可以加载这些脚本。

2．通用器件脚本

这些脚本通常和某种通用的器件有关，比如高度表、速度表、GPS 等。这些文件通常放置在 $FG_ROOT/Aircraft/Generic 文件夹中。在对应的 XML 文件中加载。

3．XML 嵌入脚本

这些脚本不以 nas 文件的形式存在，直接作为代码嵌入到 XML 的区段中。这些脚本代码作为绑定对象一般出现在配置文件中（键盘、鼠标、游戏杆等等），也可以作为对物体的操作响应。

```
<binding>
  <command>nasal</command>
    <script>
      print("Binding Invoked!");
    </script>
</binding>
```

4．系统脚本

系统脚本作为一种通用的脚本存在，它和特定的飞机以及器件无关，例如一些基本的数学运算函数，基本的界面控制函数等都需要做成系统脚本。系统脚本存放在$FG_ROOT/Nasal 文件夹中。这个文件夹中的所有 nas 文件在 FlightGear 启动以后都会被自动加载。

5．模块脚本

有时后，我们开发一个模块时，可能会用到多个脚本文件。这些脚本文件可以编成一个组，放到一个文件夹下。然后把这个文件夹复制到$FG_ROOT/Nasal 文件夹中。这些模块脚本不会随 FlightGear 自动启动，可以通过改变一个属性使这个模块根据需要启动。

模块脚本有以下几个优点：

（1）相关的脚本文件可以组织到一个文件夹下，而不是和一堆无关的文件一起放到 $FG_ROOT/Nasal 文件夹中，脚本管理起来更加清晰。local_weather、canvas 就是一个很好的例子，还有很多种类似的模块。

（2）可以保证加载的顺序。模块脚本的加载在系统脚本之后，这样，就可以保证对系统模块的调用不会出现问题，否则可能会出现模块脚本调用了系统函数，而系统函数对应的脚本还没有加载进来的问题。

（3）模块脚本可以禁用。在运行过程中，我们可以根据需要禁用这个模块，只需把对应的属性值设置为 0 就可以了。

模块脚本和单文件脚本的不同之处就是名字空间。每个文件都有自己的名字空间，如果我们想调用 gui.nas 文件中的变量，应该这样调用 gui.foo。但所有的模块脚本共用一个名字空间，空间的名字就是通过 XML 文件加载的区段的名字。例如，我们在 preferences.xml 中这样加载 <nasal><local_weather>...</local_weather></nasal>，以后调用这个模块中的变量时就可以这样调用：local_weather.single_cloud_wrapper()。

在编写模块脚本时一定要注意设置一个启动条件。通常我们用 listener 来做。在 local_weather.nas 的最底部使用了这样一行代码_setlistener("/nasal/local_weather/enabled", updateMenu)。这样，在系统中，只需将 updateMenu 设置为 1，就可以启用模块；设置为 0，就可以禁用模块。

10.2 开发与调试

1．NASAL 控制台

进行系统功能开发时，其实编写代码只是一部分，还有很大一部分时间是代码的调试。有时，代码没有语法错误，但运行结果与预期不一致，就说明代码中有逻辑错误，这时就需要进行调试。像 VisualC++就有一个非常方便的开发调试环境。遗憾的是 NASAL 没有像微软这么强有力的后台支持，所以，连个像样点的编辑器都没有，更不用说功能强大的调试器了。NASAL 的调试只能通过 FlightGear 中的 Nasal Console 来进行。幸运的是，这个 Console 有好多个标签，每个标签都可以保存不同的代码，这样我们就可以方便的在不同的标签中放置不同的代码来调试。并且，如果嫌标签少，还可以通过修改 autosave.xml 文件中的<code>节点来增加。并且，关闭 FlightGear 后，这些标签中的代码是自动保存的。下次启动 FlightGear 时，会自动加载保存的代码，这给我们调试提供了一定的方便。

2．在不重启 FlightGear 的情况下加载或重新加载 NASAL 代码

在测试和调试代码时，最大的问题就是有时需要重启 FlightGear，但 FlightGear 的启动时间非常长，这不得不耽误我们大量的时间来等待 FlightGear 的启动过程。

FlightGear 提供了一个 io.load_nasal 系统函数。这个函数可以动态加载 nas 文件。利用这个功能，我们就可以不用重启 FlightGear，而只需测试、修改、重新加载、再测试、再修改，从而对所开发的功能不断完善。请看下面的例子：

我们在$FG_ROOT/foo 目录下创建一个名为 test.nas 的文件，文件内容为：

```
print("hi!");
var msg = "My message.";
var hello = func { print("I'm the test.hello() function"); };
```

然后打开 Nasal Console 输入下面的代码：

```
io.load_nasal(getprop("/sim/fg-root") ~ "/foo/test.nas", "example");
```

这行代码的意思是，按照我们设定的文件路径，加载 test.nas 文件，并且加载到 example 名字空间中。执行这行代码后，我们发现输出了信息“hi!”。并且 hello 函数也可用了，切换一个标签，然后输入代码：example.hello()。执行后，输出信息“I'm the test.hello() function”。也可以使用 print(example.msg)来输出变量 msg 的内容。如果需要修改，只需在外部修改 test.nas 文件，然后在 FlightGear 中重新加载一次，就可以看到修改后的结果了。

3．timer 和 listener

关于 listener 的使用，后面的内容有详细的介绍，这里只介绍 listener 对重新加载代码的影响。如果程序中设置了 timer 循环或 listener，那就比较麻烦了。因为，timer 循环和 listener 一旦设置，就永远存在了，即使重新载入了代码，也不会删除上次设置的 timer 循环和 listener。并且，新载入的代码又设置了一次 timer 循环和 listener，这样，就有两套 timer 循环和 listener 在同时运行，不出问题才怪。如果重新启动 FlightGear 的某个模块，系统其实也是重新加载了相关代码，也有可能出现这种问题。

所以，我们要想办法来解决这个问题。其实只要我们重启的时候把上次设置的 timer 循环和 listener 删掉不就可以了吗。timer 循环好办，在前面的内容中，我们已经了解到了，可以用设置 loopid 的方式来停掉无用的循环。对于 listener 只能手动删除了。我们通常的做法是在重新载入 nas 文件前，删除所有的 listener，也可以编个函数来做这些清理工作，然后再载入 nas 文件。这样就不会出问题了，这也是一个非常好的编程习惯。

对于包含很多 timer 和 listener 的复杂的脚本。比较好的做法是让系统重启的时候可以自动做清理工作。这个功能可以用 listener 实现，参见下面的代码：

```
var cleanup = func {
  removelistener(id1);
  removelistener(id2);
  removelistener(id3); };
```

```
setlistener("/sim/signals/reinit", cleanup);
```

利用 listener 来监控 reinit 这个属性，如果这个属性变化，就说明系统重启了，然后会自动触发 cleanup 函数，在 cleanup 函数中，我们可以做一些善后工作，比如删除所有 listener，将一些属性恢复成初始值等等。

这样做还是比较麻烦，就是每一次 setlistener 的时候，还需要一个变量记住这个 listener 的 id，如果代码太多了呢？就有可能会忘掉，有可能会出错。

还有一种方法是创建一个数组，每次 setlistener 的时候，自动把 id 增加到数组中去，然后用个 foreach 循环就可以遍历数组中所有的 id，然后逐个删除就可以了。改造后的代码如下：

```
var id_list=[];
var store_listener = func(id) append(id_list,id);

store_listener( setlistener("/sim/foo") );
store_listener( setlistener("/foo/bar") );

var cleanup = func(id_list) {
  foreach(var id; id_list)
  removelistener(id);
 };
```

在上面的代码中，当需要设置 listener 的时候，使用我们创建的 store_listener 函数来创建，这个函数在创建 listener 的同时，把 id 增加到了 id_list 数组中。cleanup 函数使用 foreach 循环来逐个删除每个 listener。这就相当于我们在创建 listener 的时候，同时在系统中注册了一下，当不用的时候，按照注册的 id 进行删除。即保证了 listener 的使用安全，又不会为编写代码增加过多的负担。

或是把 store_listener 函数改写成下面这个样子，以更方便使用。

```
var id_list=[];
var store_listener = func(property) append(id_list,setlistener(property) );

store_listener("/sim/foo/bar");
```

如果我们已经编好了大量的代码而又不想对代码做改动，我们可以利用函数的覆盖功能，即自定义的函数如果和系统函数重名，那么在这个空间内将使用自定义的函数，而隐藏掉系统的函数。

```
var original_settimer = settimer;
var original_setlistener = setlistener;
var cleanup_listeners = [];

var settimer = func(function, time, realtime=0) {
 original_settimer(function, time, realtime);
};

var setlistener = func(property, function, startup=0, runtime=1) {
  var handle = original_setlistener(property, function, startup, runtime);
  append(cleanup_listeners, handle);
};
```

首先定义两个函数变量来保存系统的 settimer 函数和 setlistener 函数。然后再编写自己的 settimer 函数和 setlistener 函数。在自定义的函数中除实现了原函数的功能外，还加入了注册 listener 的功能。这样以前编写的代码只要在头部加上这段代码就可以了。最后要做的就是加入下面一行代码：

```
_setlistener( "/sim/signals/reinit", remove_listeners );
```

首先定义两个函数变量来保存系统的 settimer 函数和 setlistener 函数。然后再编写自己的 settimer 函数和 setlistener 函数。在自定义的函数中除实现了原函数的功能外，还加入了注册 listener 的功能。这样以前编写的代码只要在头部加上这段代码就可以了。最后要做的就是加入下面一行代码：

```
_setlistener( "/sim/signals/reinit", remove_listeners );
```

创建一个 listener 来响应系统的重启操作，然后自动触发 remove_listeners 函数。注意这里使用的是底层的_setlistener 函数，以保证在 global.nas 文件加载前也可以正常设置 listener。这在编写系统脚本和模块脚本时非常重要。打开$FG_ROOT/Nasal 文件夹，我们发现很多系统脚本和模块脚本都用 _setlistener 函数来做清理和初始化操作。

4．调　试

系统目录中的 debug.nas 文件中提供很多非常实用的调试函数。这里只介绍几个函数，其它的函数的使用可以打开 debug.nas 文件自己查看。另外，debug 还使用了多种颜色来显示，但这些颜色代码是 Linux 平台专用的，所以在 Windows 下会输出乱码，我们可以用下面一行代码来关闭 debug 的彩色显示。

```
setprop ("/sim/startup/terminal-ansi-colors",0);
```

（1）debug.dump

dump 函数会输出变量的所有相关信息。

```
var as = props.globals.getNode("/velocities/airspeed-kt", 1);
debug.dump(as);
```

代码执行后会得到下面的输出信息：

```
</velocities/airspeed-kt=1.021376474393101 (DOUBLE; T)>
```

输出的信息包含了属性的路径，属性的值，以及属性的类型。上面显示了该属性是数值型变量。“T”表示该属性是“绑定”型的（属性的类型请参考 README.gui 中的关于属性的介绍）。

（2）debug.backtrace

该函数也可以简写为 debug.bt()，用于输出所有的变量。

（3）debug.benchmark

benchmark 可以按指定的次数运行某个函数，并记录运行的时间，这在测试函数的执行效率时非常有用。

（4）debug.exit

退出 FlightGear。

10.3　扩充函数

FlightGear 非常慷慨的提供了所有的源代码，这使得我们可以根据自己的需要对代码进行更改，当然，修改代码要遵循 GNU 的原则，主要就是避免相关的法律问题。如果只是个人研究，则没这么多限制，源代码都到手了，自主权不就在我们这儿了吗。

NASAL 的实现是在 SimGear 库中实现的，主要都在 NasalSys.cxx 文件中。NASAL 已经内置了许多函数，如果觉得不够用，可以增加自己的函数，并编写相应的处理代码，编译成功后就可以使用了。不过这要求你要具有 C++的编程经验。

其实，FlightGear 已经对 NASAL 进行了扩充了。NASAL 天生不是为 FlightGear 准备的，而 FlightGear 天生就是用 NASAL 的。所以 FlightGear 根据自己的情况增加了一些非常有用的核心库函数。如果真想编写自己的核心库函数，可以参考一下 FlightGear 的相关函数是如何做的。

下面介绍一些 FlightGear 增加的一些核心库函数，这些函数作为底层函数存在，无需加前缀，直接使用。

1．fgcommand

fgcommand 函数用于执行 FlightGear 内部的 command。例如 fgcommand("loadxml",arg)就可以利用 FlightGear 中的 loadxml 命令来加载文件。$FG_ROOT/Docs 目录下的 README.commands 文件中有 FlightGear 所有内置命令的介绍。

2．getprop

getprop 函数用于获取指定路径的属性的值，例如 getprop("/sim/frame-rate")，如果没有该节点，或没有初始化，函数返回 nil。

3．setprop

setprop 函数用于设置指定属性的值，该函数返回 nil。如果只有两个参数，那么第一个参数为属性的路径，第二个参数为属性值。如果有多个参数，那么最后一个参数前面所有的参数作为字符串连接在一起并加上分隔符作为属性的路径，最后一个参数作为属性值。

第一个路径为：/sim/current-view/view-number，第二个路径为：/controls/engines/engine[0]/reverser。

```
setprop("/sim/current-view/view-number", 2);
setprop("/controls", "engines/engine[0]", "reverser", 1);
```

4．interpolate

interpolate 用于在等待给定时间（单位是秒）后，把一个值或一个属性值复制到另一属性上。

```
interpolate("controls/switches/nav-lights-pos", 1, 0.25);
# 25ms 后 nav-lights-pos = 1

interpolate("controls/gear/brake-left-pos",
getprop("controls/gear/brake-left"), 1);
# 1s 后 brake-left-pos = brake-left
```

5．settimer

settime 函数用于在等待给定时间（单位是秒）后，执行一个函数。我们还可以利用函数执行自己的功能来实现循环，这个功能在前面的循环中已经介绍过了。

```
settimer ( func { print ( "My result"); }, 5);       # 5s 后执行 print
```

6．systime

systime 函数用于获取系统时间，这个时间是相对于 1972/01/01 00:00 逝去的时间，是一个高精度的浮点数。有时，我们可以利用它来做一个高精度计时器。

```
var start = systime();
how_fast_am_I(123);
var end = systime();
print("took ", end - start, " seconds");
```

7．carttogeod

carttogeod 函数用于将笛卡尔坐标系转换为地理坐标系。输入的是笛卡尔坐标中的 x，y 和 z，函数返回一个数组，包含了纬度、经度和高度，单位是度和米。

```
var geod = carttogeod(-2737504, -4264101, 3862172);
print("lat=", geod[0], " lon=", geod[1], " alt=", geod[2]);

# 输出信息为
lat=37.49999782141546 lon=-122.6999914632327 alt=998.6042055172776
```

8．geodtocart

geodtocart 和 carttogeod 函数的作用刚好相反，功能是把地理坐标系转换为笛卡尔坐标系。

```
var cart = geodtocart(37.5, -122.7, 1000); # lat/lon/alt(m)
print("x=", cart[0], " y=", cart[1], " z=", cart[2]);
```

```
# 输出
x=-2737504.667684828 y=-4264101.900993474 z=3862172.834656495
```

9. geodinfo

geodinfo 可以获取给定经纬度的点的地理属性。该函数有两个参数，分别是纬度和经度。返回一个数组，该数组有两个元素，第一个是该点的高度，第二个是一个哈希变量，包含了该点的属性。

```
debug.dump(geodinfo(lat, lon));

# 输出信息为
[ 106.9892101062052, { light_coverage : 0, bumpiness : 0.5999999999999999,
load_resistance : 1e+30, solid : 0,  names : [ "Lake", "Pond", "Reservoir",
"Stream", "Canal" ], friction_factor : 1, rolling_friction : 1.5 } ]
```

10. airportinfo

airportinfo 可以获取机场的参数，该函数被重载了，有三种使用方法：

```
var apt = airportinfo("KHAF");      # 代号为 KHAF 的机场信息
var apt = airportinfo(lat, lon);    # 离(lat,lon)点最近的机场信息
var apt = airportinfo();            # 离本机最近的机场信息
```

11. flightplan

flightplan 可以获取飞行计划，或是从文件中加载飞行计划。该函数有两种使用方法。

```
var fp = flightplan();
var fp = flightplan("/some/path/to/a/flightplan.xml");
```

12. props.Node

props.Node 类是一个封装了 SGPropertyNode 的对象，提供了对属性树的操作，拥有下面一些成员函数。

（1）getType()

返回节点的类型，返回值是一个字符串，标明了节点值的类型。 NONE，ALIAS，BOOL，INT，LONG，FLOAT，DOUBLE，STRING，UNSPECIFIED。

getName()

返回节点的名字。

getIndex()

返回节点的索引值。

getValue()

返回节点的值，如果没有就返回 nil。

setValue()

设置节点的值。

setIntValue()

设置节点的值，并强制把节点类型改为 INT。

setBoolValue()

设置节点的值，并强制把节点类型改为 BOOL。

setDoubleValue()

设置节点的值，并强制把节点类型改为 DOUBLE。

getParent()

返回上一级节点，如果没有就返回 nil。

getChild()

返回指定名字的子节点。如果没有就返回 nil，如果有多个相同名字的子节点，就返回第一个。

getChildren()

返回一个数组，包含所有的子节点。

remove()

删除本节点及所有子节点。

removeChild()

删除指定名字的子节点。

getNode()

返回用相对路径表示的相对于本节点的节点，如果没有就返回 nil。

setValues()

设置多个子节点的值，参数是一个包含了子节点名字和值的哈希变量。

10.4 系统库函数

FlightGear 提供的强大的系统库函数，这些库函数以 nas 文件的形式保存在$FG_ROOT/Nasal 文件夹下面，在系统启动的时候自动加载。提供了操作 FlightGear 的基本函数，可以把这些函数看做是 FlightGear 提供的 API 函数。其他的任何脚本文件都是利用了 NASAL 的核心库函数和这些系统库函数来完成所实现的功能的。NASAL 核心库函数直接使用即可，当使用这些系统库函数时，必须加上前缀，例如使用 math.nas 中的 abs 函数来获取绝对值时，就必须写为：math.abs(n)。只有 globals.nas 中的函数不需要加前缀，其余的系统库函数都必须加前缀。

系统库函数涵盖了飞机、发动机、控制系统、故障、界面、字符串、数学等多个方面，全部保存在$FG_ROOT/Nasal 文件夹下。根据 nas 的文件名就可以判断该文件提供的功能，该目录中还有以文件夹形式存在的几个模块，比如在 2.8 版提供支持的 canvas 模块。

在进行相关功能开发时，应参考已有飞机的类似功能模块，充分利用系统提供的现有库函数和模块，合理组织自己的代码，以实现所需的功能。

10.4 signal 和 listener

1．signal

signal 是一个信号，用于指示某种事件的发生。在属性树中的“/sim/signals”下面有很多信号，用于表示不同的事件。

exit：当退出 FlightGear 时为 true。

reinit：重新启动 FlightGear 时为 true，然后为 false。

click：鼠标点击地景时为 true，并且，点击点的地理坐标会保存在/sim/input/click 属性下。

screenshot：截取屏幕是为 true，然后为 false。

nasal-dir-initialized：$FG_ROOT/Nasal 目录下的所有的 nas 文件加载完成后为 true。主要用于加载完系统脚本后，给出一个信号，然后其他的脚本开始加载。

fdm-initialized：FDM 初始化完成后为 true。

reinit-gui：程序界面 GUI 重启完成后为 true。

frame：系统刷新每一帧画面的信号，当然我们可以用 settimer 函数并用参数 0 来保证函数的循环调用。但 settimer 的循环是要所有的代码执行一次以后再循环，这和系统模块的多少及加载顺序有关系。而 frame 信号是系统级的，时间间隔比较固定。

2．listener

listener 是 FlightGear 中的一个重要的概念，使用非常广泛。listener 的主要用途是监控属性树的改变。在 FlightGear 中，属性树维护着大量属性，这些属性和 windows 的文件、文件夹一样组织，就像一棵树一样，有节点，有叶子。而 listener 就是一个监控器，监控着属性树的改变。我们可以利用 listener

对一个特定的属性进行监控，当该属性值发生改变时，listener 会马上响应，并触发设置的响应函数。假如原属性为 1，我们又通过函数给该属性设置值为 1。即使前后的值一样，listener 也认为该属性发生了改变，也会触发响应函数。当子节点发生变化时，listener 也可以触发。

我们也可以用循环来不断的监控属性的改变，但循环非常占用系统资源，并且，如果属性前后两次改变的值相同，循环就无法发现。而 listener 就非常适合监控一些不经常变动的属性值，并且对系统效率影响很小。

非常遗憾的是 listener 不能监控 T 型的属性，因为这些属性往往不是通过设置属性的方法来设置的，而是和系统运行直接相关的。大多数 FDM 的属性都是 T 型的。例如：“/position/altitude-ft”和“orientation/heading-deg”属性都是由 FDM 直接修改的。listener 对这些属性不起作用。因此使用 listener 前，最好在属性树中查看一下需要监控的属性是不是 T 型的。在属性树的浏览窗口中，按住 Ctrl 键，用鼠标点击“.”，就会显示属性的详细信息，其中属性的类型在后面显示。

对于不能使用 listener 来监控的属性，只能使用 settimer 了。listener 适合监控那些不经常发生改变的属性，对于每一帧都会发生变化的属性，listener 每一帧都会触发，而把 settimer 循环时间设为 0 也可以保证每一帧都触发，但这时，listener 的效率就没有 settimer 效率高。这也是 FDM 的属性不能使用 listener 的原因，因为 FDM 是实时运算的，输出的信息每一帧都是不同的。

FlightGear 提供了_setlistener 和 setlistener 两个函数，_setlistener 是核心库提供的底层函数，而 setlistener 是 global.nas 提供的系统函数。setlistener 和_setlistener 的区别就是，setlistener 是对_setlistener 的封装。_setlistener 函数的可调参数更多，一般情况下我们不用_setlistener 函数，只有在编写系统脚本或模块脚本时才使用。因为，setlistener 是在 global.nas 中提供的，所以，只有当 FlightGear 加载完 global.nas 后，才提供 setlistener 函数。而有些系统脚本是在 global.nas 之前加载的，那时，setlistener 函数还不可用。所以，为了保证 listener 可以设置成功，必须用底层的_setlistener 函数。

3．setlistener()

该函数用于设置一个 listener 来监控属性的变化。函数的用法是：

```
var id = setlistener( prop, func, startup = 0, runtime = 1);
```

该函数有 4 个参数。

第一个参数需要监控的属性的路径或是属性节点，例如“/gear/launchbar/state”或 props.Node()；

第二个参数是响应函数，当监控的属性发生变化时，就是自动触发执行该函数；

第三个参数是响应函数是否初始化。默认为 0，表示设置 listener 的时候，响应函数不触发执行，只有当监控的属性发生变化时才触发执行。当设置为 1 时，当设置 listener 的时候，响应函数先执行一次作为初始化，然后当监控的属性发生变化时，再触发执行。这两种方式的区别就是在于设置 listener 的时候，函数是否执行。

第四个参数是执行方式，默认为 1，意思是只要这个属性值被写入，就马上触发。当设置为 0 时，只有新写入的值与原值不同时，才会触发，否则不触发。当设置为 2 时，子节点的创建、删除、修改也会触发响应函数。

setlistener 函数返回一个 id 值，该值主要用于删除 listener 时使用。用法是：

```
removelistener(id);
```

响应函数可以接收 4 个参数，但我们通常不使用参数或只用第一个参数。当设置了响应函数时，其实函数并未执行，只有 listener 触发时，才会执行响应函数。因此，如果响应函数调用了外部的某个变量，那么响应函数使用的是执行函数时变量的值，而不是设置 listener 时候的变量的值。

```
func([<changed_node> [, <listened_to_node> [, <operation> [,
<is_child_event>]]]])
```

下面的例子是调用没有参数的响应函数。

```
var say_bye = func { print("bye") };
setlistener("/sim/signals/exit", say_bye);
```

下面的例子是调用带 1 个参数响应函数，第一参数是监控的节点，这样，当响应函数被触发时，我们不仅知道了监控的属性发生了改变，同时，还可以使用 getValue 函数获得改变后的值。

```
var monitor = func(limit) {
   print("VALUE: ", limit.getValue());
};
var i = setlistener("instrumentation/radar/limit-deg", monitor);
```

下面的例子是调用带 2 个参数的响应函数，第一个参数是发生变化的节点，第二个参数是监控的节点。注意，在存在子节点的情况下，这两个参数是不一样的，下面的例子介绍了这种情况。通过 setlistener 函数设置对节点“instrumentation/radar”进行监控，但是“instrumentation/radar”不是一个属性，而是一个节点，下面还有很多子节点和属性，所以，我们在 setlistener 的函数中加入第 3 个和第 4 个函数。第 3 个参数为 0 表示响应函数在设置 listener 时不执行。第四个参数为 2 表示同时监控该节点下的子节点的变化。

```
var monitor = func(radarinfo, radar) {
   debug.dump(radarinfo);
   debug.dump(radar);
};
var i = setlistener("instrumentation/radar", monitor, 0, 2);
```

这时，当“instrumentation/radar/limit-deg”发生改变时，listener 监控到 radar 的子节点发生变化，立即触发响应函数。并且，radarinfo 保存了发生变化的子节点，也就是“instrumentation/radar/limit-deg”，而 radar 中保存着 listener 监控的节点，也就是“instrumentation/radar”。用这种方法来监控一个包含很多子节点的节点非常有效。

对同一个节点，可以使用多个 listener 来监控，并且当该节点发生变化时，可以按照对节点设置的 listener 分别触发不同的响应函数。使用属性树来查看时，节点后面的表示属性类型字符串中的“Ln”就表示对此节点监控的 listener 的数量。

10.5 例 1：创建属性 XML 文件

在 IO 系统库中，提供了读写 XML 文件的函数，使的我们可以方便的创建自己的 XML 文件。当然，XML 文件是用明文存储了，目前几乎所有的编辑器都支持对 XML 文件的编辑及高亮显示。但 NASAL 可以只用几行代码就能方便的建立一个和属性树对应的 XML 文件。

在 IO 库中有 read_properties 和 write_properties 函数，这两个函数实际上是对 fgcommand 中的“loadxml”和“savexml”命令的封装。

使用这些函数，可以很方便的把属性树的某个分支保存到 XML 文件中，这样，我们就可以用 setprop 函数来改变某个属性值，然后用 write_properties 函数来保存文件，对于某些情况下非常的方便。请看下面的代码：

```
var location= "/temp/test/foo";          # 属性树中的路径
var filename="test.xml";                 # 保存的 XML 文件名
setprop(location, "hello world");
io.write_properties(filename, location);
```

执行上面的代码后，就会创建 test.xml 文件，文件中保存了我们需要的属性树结构。注意，如果硬盘上相同文件夹下有一个同名文件，那么原文件会被删除！test.xml 文件的内容为：

```
<?xml version="1.0"?>
  <PropertyList>
    <temp>
      <test>
```

```
      <foo>hello world</foo>
    </test>
  </temp>
 </PropertyList>
```

在创建面板 XML 文件或其他文件时，有可能会出现很多类似的代码段，基本结构是一样的，只是部分字段的值不一样，如下面所示：

```
<path>vsd.ac</path>
 <animation>
  <type>select</type>
  <object-name>vsd</object-name>
  <condition>
   <greater-than-equals>
    <property>systems/electrical/outputs/efis</property>
    <value>9</value>
   </greater-than-equals>
  </condition>
 </animation>
```

这时我们就可以用下面的 NASAL 代码来创建这个 XML 文件。

```
var location = "/temp/test/";
var filename="xmltest.xml";
setprop(location~"path", "vsd.ac");
setprop(location~"animation[0]/type", "select");
setprop(location~"animation[0]/object-name", "vsd");
setprop(location~"animation[0]/condition/greather-than-equals/property",
"systems/electrical/outputs/efis");
setprop(location~"animation[0]/condition/greather-than-equals/value",
"9");
io.write_properties(filename, location);
```

10.6 例 2：地景取样

在进行地形开发时，有时，我们需要方便的获取任意点的地理参数，本例利用 NASAL 代码方便的实现了这一点。本例主要是仿照 FlightGear 中的仪表器件的实现方法，利用系统脚本和模块脚本来实现按照一定的速率来更新某些参数。

需要注意的是，用这种方式调用函数受限于系统的刷新率，如果 FPS 为 50，那么调用函数的次数最多 1 秒 50 次，因为，它是在每一帧的间隔来执行代码。

1．创建一个系统模块

在$FG_ROOT/Nasal 文件下面创建一个名为 terrainsample.nas 的文本文件。表示我们创建了一个系统脚本，这些脚本作为系统模块存在，在启动 FlightGear 时由系统自动加载。其中 terrainsample.nas 文件的内容为：

```
var sampler = func {
  print("running sampler() ");
  settimer(sampler, 3); # 以 3 秒为周期的定时器循环
 };
```

```
# 用 listener 设置为加载完系统 nas 后，自动加载本文件。注意用底层的_setlistener
_setlistener("/sim/signals/nasal-dir-initialized", sampler);
```

上面的代码我们就建立了一个系统模块，该模块在 FlightGear 启动后自动加载，并且，每隔 3 秒输出一段文字。这就是一个简单的系统模块。

这个 sampler 函数目前只是输出一段文字，没有实际的意思，现在我们给这个函数编写具体的工作代码。我们利用 getprop 函数获取当前飞机的经纬度信息，然后利用系统提供的 geodinfo 函数，获取该坐标的地理信息，其中返回的数组中第一个元素就是该点的高度，然后输出该高度，这样在飞行的过程中，就可以实时看到飞机所在位置处的地理高度。把飞机的当前的飞行高度减去此高度，就可以得到当前飞机相对于地面的真实高度。当然 geodinfo 函数还返回了该点的更多的信息，我们可以取出使用。

```
var sampler = func {
   print("running sampler() ");
   var lat=getprop("/position/latitude-deg");
   var lon=getprop("/position/longitude-deg");
   var result=geodinfo(lat,lon);
   var elevation_m = result[0];
   print("elevation is: ", elevation_m, " m");
   settimer(sampler, 3);
 };
 _setlistener("/sim/signals/nasal-dir-initialized", sampler);
```

上面的代码实现了显示飞机所在位置处的地理信息，但有时我们在设计地形时往往利用 UFO 进行飞行，然后用鼠标选择地面更加方便。FlightGear 在“/sim/input/click”节点下提供了鼠标点击位置处的地理信息，有“elevation-ft”、“elevation-m”、“latitude-deg”、“longitude-deg”四个属性。我们可以改造这个模块，实现鼠标选取点的地理信息显示。

```
var sampler = func {
   print("running sampler() ");
   var lat=getprop("/sim/input/click/latitude-deg");
   var lon=getprop("/sim/input/click/longitude-deg");
   var result=geodinfo(lat,lon);
   debug.dump(result);
};

var start_samper = func{
   setlistener("/sim/input/click/latitude-deg",sampler);
};
setprop ("/sim/startup/terminal-ansi-colors",0);  # 禁止颜色代码
_setlistener("/sim/signals/nasal-dir-initialized", start_sampler);
```

10.7 例 3：I/O 端口操作

NASAL 是不能直接操作外接设备的，但是 FlightGear 提供了强大的 I/O 功能，使得我们利用自定义的协议和外部设备进行数据交换。通过制作通讯协议，我们可以很方便的把外部的数据导入属性树中，或是把属性树中的参数传递到外面。试想一下，利用这种通讯协议，我们可以制作外接的模拟器面板或飞行摇杆，然后把面板的操作利用串口传递给 FlightGear，然后 FlightGear 再把接收到的信息更新到属性树中，从而实现了外部设备对 FlightGear 的控制。

我们还可以把 FDM 也放到外面，利用 MATLAB 强大的功能来做一个飞行模型，然后把计算的数据通过网络传递给 FlightGear，把 MATLAB 的计算结果用飞机来展示出来。还可以把飞行参数通过网络传递给另外一个飞参处理软件，这个软件可以在坐标轴上用曲线的形式展示实时的飞行参数，或是在 ATC 上显示整个空域的飞行情况。

FlightGear 支持几种协议通讯模式。利用这些协议，FlightGear 就可以方便的和外部进行数据交换。这里，我们使用普通协议（generic protocol）。普通协议的使用非常简单，只需要创建一个 XML 文件，在文件中指定需要发送或接收的属性，还可以指定格式。然后 FlightGear 就可以利用这个协议来通讯了，这里说的通讯范围很广，通讯的对象可以是硬盘上的一个文件，可以是网络终端或是一个连接到串口上的设备。

下面是一个简单的协议，指定了需要发送的参数，还可以进一步指定参数的格式以及变化系数。指定用该协议启动 FlightGear 后，FlightGear 就会按照协议规定的格式向外部发送数据。

```
<?xml version="1.0"?>
<PropertyList>
  <generic>
    <output>
      <line_separator>newline</line_separator>
      <var_separator>newline</var_separator>
      <binary_mode>false</binary_mode>
      <chunk>
        <name>speed</name>
        <format>V=%d</format>
        <node>/velocities/airspeed-kt</node>
      </chunk>
      <chunk>
        <name>heading (rad)</name>
        <format>H=%.6f</format>
        <type>float</type>
        <node>/orientation/heading-deg</node>
        <factor>0.0174532925199433</factor>  <!-- degrees to radians -->
      </chunk>
      <chunk>
        <name>pitch angle (deg)</name>
        <format>P=%03.2f</format>
        <node>/orientation/pitch-deg</node>
      </chunk>
    </output>
 </generic>
</PropertyList>
```

如果外部设备可以接收命令，例如外部利用串口连接的一个 LCD 显示设备，该设备可以利用指令来清屏，设置颜色，字体大小，字符位置等。我们就可以在属性树中创建一个 sendmessage 的属性节点，然后利用协议将此属性发送到串口。当需要发送指令时，只需要用 setprop 函数设置该属性值即可。

```
setprop("/sim/commu/sendmessage","0xFF,0xCA,0xFA");
```

反之一样，如果需要外部设备对 FlightGear 进行控制，可以利用协议发送命令到 recvmessage 中，然后，然后做一个 listener 监控这个属性，当有外部命令送入时，马上触发响应函数来处理这个命令。

```
var processMessage = func() {
  var msg = getprop("/sim/commu/recvmessage");
```

```
  print("Info: Message received! MSG:", msg);};
setlistener("/sim/commu/recvmessage", processMessage);
```