

kOS 中文教程

for KSP 1.7.3

Article by **Lookerksy** on 2019.8.8

1.

1. 概述

1.1

1.1 kOS 概述

===== 点击以返回目录 =====



```
PRINT "Hello World.".  
PRINT "These are the documents for Kerbal Operating System.".
```

(1) kOS 自动驾驶

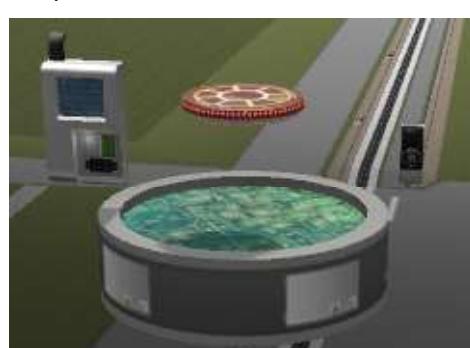
kOS 全称 Kerbal Operating System，是 PC 游戏Kerbal Space Program（坎巴拉太空计划）的一个 MOD，可以通过脚本化的编程来实现载具自动控制。



(2) kOS 内容简述

kOS 提供了几个新部件，每个部件都模拟了一台机载计算机，
他有自己的语言（KerboScript）。

玩家可以在上面运行用 KerboScript 语言编写的程序。



*注1: kOS 提供了如上4个新部件，本文管他们叫“kOS 部件”。
他们功能相同，区别仅在于大小、重量、文件存储容量。

(3) kOS 的特性

kOS 入门简单，功能丰富，用途广泛。具有以下特点：

1. 以数值形式获取精准数据。
2. 编程计算。
3. 精准执行各项操作。
4. 读写电脑上的文本文件。
5. 自定义 UI 窗口。（GUI）
6. 游戏视角控制。（可选项，需使用 KOS-StockCamera）

(4) kOS 安装与卸载

安装：把内容文件放到游戏目录\GameData 文件夹下。

卸载：删除游戏目录\GameData\kOS 文件夹，并删除游戏目录\Ships\Script 文件夹。

(5) KSP 的游戏语言

建议选择英文作为 KSP 的游戏语言。

因为编程的关键词/关键变量是英文的，可以直接照抄屏幕。

如果游戏语言选中文，则 kOS 学习/使用难度上一个台阶。

(6) kOS 的作者

kOS 起初由 Kevin Laity aka Nivekk 独立开发完成。现在则有一个团队持续进行开发与维护。

1.2

1.2 文章说明

===== 点击以返回目录 =====



(1) 文章定位

本文兼具教程与工具书的定位。

(2) 编写版本环境

本文在以下环境下编写：

全称	简称	版本号	说明
Kerbal Space Program	kSP	1.7 .3	游戏本体
Making History	DLC1	1.7 .1	DLC1
Breaking Ground	DLC2	1.2 .0	DLC2
Kerbal Operating System	kOS	1.1 .9 .0	编程控制 MOD

编写本文时搭配了以下 MOD。

全称	简称	版本号	说明
BDArmory	BDA	1.3 .1 .0	武器
Kerbal Joint Reinforcement	KJR	4.0 .14	节点加固
Critical Temperature Gauge	CTG	1.7 .0 .1	部件温度
Infernal Robotics	IR	3.0 .0 beta3 p7	转轴
Trajectories	TR	2.2 .1	落点预测

对应内容的下载地址请看 [这里](#)

(3) 适用版本环境

本文适用 KSP 1.7.3，对于其他版本KSP的适配性请自行测试。

(4) 文章的发布

本文使用 Mathematica 11.3.0 书写。发布有 .nb 和 .pdf 两种文件格式。

.nb 文件为 Mathematica 的笔记本文件，信息最完整；

.pdf 文件由 .nb 输出，有部分功能和格式丢失。

请悉知。

2.

2. 目录



1. 概述

1.1 kOS 概述

1.2 文章说明

2. 目录

3. 下载链接

4. 实例教程

4.1 kOS 载具操控入门

4.1.1 导读

4.1.2 实例1: Hello World

4.1.3 实例2: 控制火箭发射

4.2 编程指导

4.2.2 实例3: kOS 中的基本表达式

4.2.3 实例4: kOS 程序的优化

4.3 CSV 数据导出

4.3.1 导读

4.3.2 实例5: 用 Log 指令导出 CSV 数据

4.4 PID 控制教程

4.4.1 导读

4.4.2 PID 控制与 Ziegler-Nichols 方法

4.4.3 实例6: PID 控制火箭反推悬浮

4.5 变轨教程

4.5.1 导读

4.5.2 实例7: 控制火箭圆轨

4.6 边界测试

4.6.1 导读

4.6.2 实例8: 边界监测程序

4.7 GUI设计入门

4.7.1 导读

4.7.2 GUI 的构成

4.7.3 实例9：GUI——天体信息查询器

4.7.4 实例10：GUI——选项卡控件 TabWidget

5. kOS 与 KSP

5.1 系统预设变量

5.2 当前载具 (Ship)

5.3 kOS 的工作特点

5.3.1 显示帧、物理帧、逐帧宇宙

5.3.2 耗电

5.3.3 触发器

5.3.4 等待

5.4 kOS 部件

5.5 窗口设置

5.5.1 kOS 控制面板

5.5.2 kOS 难度设置

5.6 自定义 GUI 界面

5.7 文件和卷

5.7.1 脚本文件

5.7.2 文件存储位置

5.7.3 扩展 kOS 部件的存储容量

5.7.4 单个载具搭载多个 kOS 部件

5.7.5 数据中心(Archive)

5.7.6 自启动文件夹

5.8 机器语言

5.9 音频演奏 (SKID)

5.9.1 简明用法示例

5.9.2 kOS 中音频的构成

5.9.3 音频的代码实现

5.9.4 实例11: kOS 演奏乐曲

5.10 指定部件 (获取部件结构体Part)

5.10.1 部件的树状组织结构

5.10.2 通过部件列表指定部件

5.10.3 通过名称搜索并指定部件

5.10.4 通过动作组搜索并指定部件

5.10.5 通过正则表达式搜索并指定部件

5.11 部件操作 和 部件模组ParModules

5.11.1 部件信息和部件操作

5.11.2 部件模组 PartModules

5.11.3 实例12: 部件右键菜单使用集锦

5.12 生涯模式科技树限制

6. kOS 语言

6.1 kOS 语言特性

6.1.1 大小写不敏感

6.1.2 表达式

6.1.3 逻辑短路

6.1.4 自定义变量的自动类型

6.1.5 全局变量隐式声明

6.1.6 自定义函数

6.1.7 结构体

6.1.8 触发器

6.2 语法

6.2.1 一般规则

6.2.2 数值

6.2.3 代码块

6.2.4 内置函数

6.2.5 结构体函数(方法)

6.2.6 字典结构的后缀用法

6.2.7 自定义函数

6.2.8 预设的特殊变量

6.2.9 不存在自定义结构体

6.3 顺序控制

6.3.1 数值锁定和解锁

6.3.2 条件结构

6.3.3 选择表达式

6.3.4 循环结构

6.3.5 等待语句

6.3.6 逻辑操作符

6.3.7 函数声明

6.3.8 触发结构

6.4 变量和声明

6.4.1 变量声明

6.4.2 程序文件的参数

6.4.3 查询变量是否定义

6.4.4 Set 语句

6.4.5 Unset 语句

6.4.6 变量 Lock 赋值

6.4.7 逻辑量的操作

6.4.8 变量的有效范围

6.5 自定义函数

6.5.1 概览

6.5.2 声明函数

6.5.3 库函数文件

6.5.4 函数的调用

6.5.5 函数中的局部变量

6.5.6 函数的返回值

6.5.7 函数的传值调用和传址调用

6.5.8 函数的嵌套和递归

6.5.9 匿名函数

6.6 匿名函数

6.6.1 语法

6.6.2 匿名函数的用途

6.7 函数指针

6.7.1 函数指针语法: @符号

6.7.2 函数指针的用途

6.7.3 函数指针与匿名函数

7. 数学和基本几何

7.1 基本常数与数学函数

7.1.1 数学/物理常数

7.1.2 数学函数

7.2 数值 Scalar [结构体]

7.3 列表 List [结构体]

7.4 矢量 Vector [结构体]

7.5 朝向 Direction [结构体]

7.6 经纬坐标 GeoCoordinates [结构体]

7.7 坐标系

7.7.1 左手坐标系

7.7.2 三种坐标系

7.7.3 坐标系换算

8. kOS 指令

8.1 运行程序

8.1.1 概述

8.1.2 函数方法运行程序

8.1.3 关键词方法运行程序(不推荐)

8.1.4 程序运行的细节

8.2 飞行控制

8.2.1 概述

8.2.2 经典控制

8.2.3 底层控制 Control [结构体]

8.2.4 驾驶员输入 Control [结构体]

8.2.5 飞船系统

8.3 飞行路径预测

8.4 列表List (指令)

8.5 查询部件信息

8.6 文件 I/O

8.6.1 理解文件目录

8.6.2 路径

8.6.3 盘符

8.6.4 文件和目录

8.6.5 对 json 文件进行读写

8.6.6 用 Log 指令进行文件写入

8.6.7 文件读写总结

8.7 指令窗口和 GUI

8.7.1 指令窗口操作

8.7.2 屏幕显示工具

8.8 可序列化

8.9 通讯

8.9.1 通讯概述

8.9.2 报文

8.9.3 报文队列

8.9.4 通讯连接

8.9.5 载具间的通讯收发

8.9.6 载具内的通讯收发

8.9.7 实例13：用通讯报文(Message)指挥僚机

8.10 资源转移

9. 结构体

9.1 底层结构体

9.1.1 结构体 Structure [结构体]

9.2 集合

9.2.1 枚举 Enumerable [结构体]

9.2.2 迭代器 Iterator [结构体]

9.2.3 列表 List [结构体]

9.2.4 范围 Range [结构体]

9.2.5 序列 Queue [结构体]

9.2.6 堆栈 Stack [结构体]

9.2.7 唯一集 UniqueSet [结构体]

9.2.8 词典 Lexicon [结构体]

9.3 盘符与文件

9.3.1 盘符 Volume [结构体]

9.3.2 路径 Path [结构体]

9.3.3 盘符项目 VolumeItem [结构体]

9.3.4 盘符路径 VolumeDirectory [结构体]

9.3.5 盘符文件 VolumeFile [结构体]

9.3.6 文件内容 FileContent [结构体]

9.4 载具与部件

9.4.1 载具 Vessel [结构体]

9.4.2 乘员 CrewMember [结构体]

9.4.3 载具传感器 VesselSensors [结构体]

9.4.4 资源总量 AggregateResource [结构体]

9.4.5 对接单元 Element [结构体]

9.4.6 部件 Part [结构体]

9.4.7 边界 Bounds [结构体]

9.4.8 部件资源 Resource [结构体]

9.4.9 部件模组 PartModule [结构体]

9.4.10 kOS处理器 kOSProcessor [结构体]

9.4.11 内核 Core [结构体]

9.4.12 传感器 Sensor [结构体]

9.4.13 分离器 Decoupler [结构体]

9.4.14 对接口 DockingPort [结构体]

9.4.15 引擎 Engine [结构体]

9.4.16 引擎矢量喷口 Gimbal [结构体]

9.5 航行与操控

9.5.1 命令序列 Stage [结构体]

9.5.2 发射架 LaunchClamp [结构体]

9.5.3 高度 ALT [结构体]

9.5.4 等待时间 ETA [结构体]

9.5.5 调姿管理 SteeringManager [结构体]

9.5.6 变轨计划 ManeuverNode [结构体]

9.5.7 资源转移 ResourceTransfer [结构体]

9.5.8 飞行操控 Control [结构体]

9.5.9 矢量 Vector [结构体]

9.5.10 朝向 Direction [结构体]

9.5.11 经纬坐标 GeoCoordinates [结构体]

9.6 通讯与科研

9.6.1 通讯连接 Connection [结构体]

9.6.2 报文队列 MessageQueue [结构体]

9.6.3 报文 Message [结构体]

9.6.4 科研数据 ScienceData [结构体]

9.6.5 科研实验模组 ScienceExperimentModule [结构体]

9.7 轨道与天体

9.7.1 轨道物 Orbitable [结构体]

9.7.2 轨道 Orbit [结构体]

9.7.3 轨道速度 OrbitableVelocity [结构体]

9.7.4 天体 Body [结构体]

9.7.5 大气层 Atmosphere [结构体]

9.8 GUI 部件

9.8.1 GUI 设计技巧

9.8.2 控件 Widget [结构体]

- 9.8.3 窗体 GUI [结构体]
- 9.8.4 框体 Box [结构体]
- 9.8.5 滚动框体 ScrollBox [结构体]
- 9.8.6 文本 Label [结构体]
- 9.8.7 提示显示 TipDisplay [结构体]
- 9.8.8 按钮 Button [结构体]
- 9.8.9 下拉菜单 PopupMenu [结构体]
- 9.8.10 文本框 TextField [结构体]
- 9.8.11 滑块 Slider [结构体]
- 9.8.12 空档 Spacing [结构体]
- 9.8.13 样式 Style [结构体]
- 9.8.14 边距 StyleRectOffset [结构体]
- 9.8.15 状态样式 StyleState [结构体]
- 9.8.16 皮肤 Skin [结构体]

9.9 编程与配置

- 9.9.1 数值 Scalar [结构体]
- 9.9.2 逻辑值 Boolean [结构体]
- 9.9.3 字符串 String [结构体]
- 9.9.4 色彩 Colors [结构体]
- 9.9.5 函数指针 KOSDelegate [结构体]
- 9.9.6 PID循环 PIDLoop [结构体]
- 9.9.7 箭头绘制 VecDraw [结构体]
- 9.9.8 部件高亮 HighLight [结构体]
- 9.9.9 kOS 配置 Config [结构体]

9.10 杂项

- 9.10.1 第四墙 KUniverse [结构体]
- 9.10.2 载具存档 CraftTemplate [结构体]
- 9.10.3 载具载入距离 VesselLoadDistance [结构体]
- 9.10.4 场景载入距离 SituationLoadDistance [结构体]
- 9.10.5 时间 TimeSpan [结构体]
- 9.10.6 时间加速 TimeWarp [结构体]
- 9.10.7 指令窗口 Terminal [结构体]

9.10.8 指令窗口输入 TerminalInput [结构体]

9.10.9 音轨 Voice [结构体]

9.10.10 音符 Note [结构体]

9.10.11 版本信息 VersionInfo [结构体]

9.10.12 路标 WayPoint [结构体]

10. MOD 相关

10.1 MOD支持 Addons [结构体]

10.2 动作组扩展 Action Groups Extended (AGE)

10.3 通讯MOD支持 RTAddon [结构体]

10.4 闹钟 Kerbal Alarm Clock (KAC)

10.4.1 闹钟 KACAddon [结构体]

10.4.2 闹铃 KACAlarm [结构体]

10.4.3 使用坎巴拉闹钟

10.5 转轴 Infernal Robotics (IR)

10.5.1 转轴MOD支持 IRAddon [结构体]

10.5.2 转轴控制组IRControlGroup [结构体]

10.5.3 转轴单元 IRServo [结构体]

10.6 落点预测 Trajectories (TR)

10.6.1 落点 TRAddon [结构体]

11. kOS 的源码编辑环境

3.

3. 下载链接

===== 点击以返回目录 =====



本章罗列了本文攥写过程中，所有使用组件/工具的获取方式和链接。

(1) 获取 Mathematica 本体

全称	版本	说明	官网	国内代理	百度贴吧
Mathematica	11.3 .0	数学软件	点这里	点这里	点这里
.....					
说明: 无					



(2) 获取 KSP 本体及DLC

全称	版本	说明	Steam页
Kerbal Space Program	1.7 .3	游戏本体	点这里
Making History	1.7 .1	DLC1	点这里
Breaking Ground	1.2 .0	DLC2	点这里
.....			
说明: 无			



(3) 获取 kOS MOD本体

全称	版本	说明	Forum页	Github下载
Kerbal Operating System	1.1 .9 .0	编程MOD	点这里	点这里
.....				
说明: 英文教程: 在线阅读、离线包下载。				



(4) 获取 武器 MOD (BDArmory)

全称	版本	说明	Forum页	Github下载
BDArmory	1.3 .1 .0	武器MOD	点这里	点这里
.....				

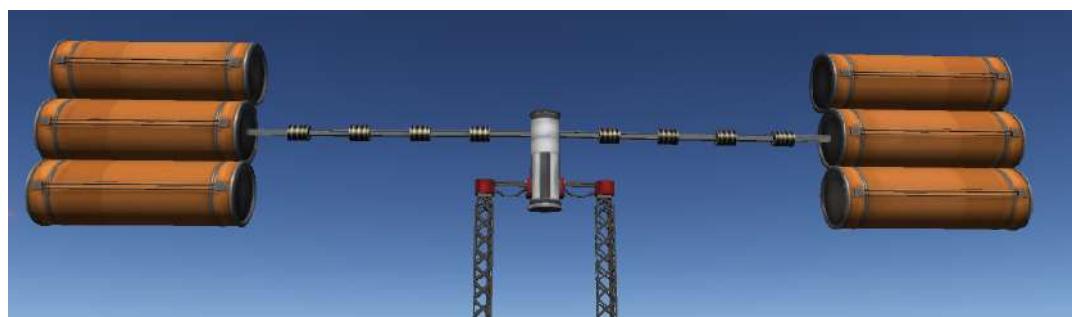
说明: 无



(5) 获取 节点加固 MOD (Kerbal Joint Reinforcement)

全称	版本	说明	Forum页	Github下载
Kerbal Joint Reinforcement	4.0 .14	节点加固MOD	点这里	点这里
.....				

说明: 强化部件之间的连接强度。 (对转轴部件的内部连接强度不起效)



(6) 获取 部件温度 MOD (Critical Temperature Gauge)

全称	版本	说明	Forum页	Github下载
Critical Temperature Gauge	1.7 .0 .1	部件温度MOD	点这里	点这里
.....				

说明: 此 MOD 在部件右键菜单中追加出的温度数据能被 kOS 读取。



(7) 获取 转轴 MOD (Infernal Robotics)

全称	版本	说明	Forum页	下载
Infernal Robotics	3.0 .0 beta3 p7	转轴MOD	点这里	点这里
说明: 无				



(8) 获取 落点预测 MOD (Trajectories)

全称	版本	说明	Forum页	Github 下载
Trajectories	2.2 .1	落点预测MOD	点这里	点这里
说明: 实时计算大气阻力作用下的载具轨迹及落点, 结果可被 kOS 读取。				



4.

4. 实例教程

4.1

4.1 kOS 载具操控入门

4.1.1

4.1.1 导读

===== 点击以返回目录 =====



建议玩家在初学 kOS 时必看。

本节是 kOS 的快速入门指南。

本节内容如下：

实例1：Hello World

- Step 1: 沙盒存档
- Step 2: 捏一个载具
- Step 3: 发射载具
- Step 4: 打开指令窗口
- Step 5: 使用指令窗口
- Step 6: kOS 中的文件系统

实例2：控制火箭发射

- Step 1: 捏一枚火箭
- Step 2: 显示发射倒数
- Step 3: 点火发射！
- Step 4: 控制火箭姿态
- Step 5: 火箭分级功能
- Step 6: 控制火箭转向
- Step Extra: 通过预设变量和结构体变量操控载具

4.1.2

4.1.2 实例1：Hello World

===== 点击以返回目录 =====

建议玩家在初学 kOS 时必看。

本实例中，我们将手把手做“Hello World”程序，借此了解 kOS 的使用流程。
另外我们还将介绍 kOS 中的文件系统。

Step 1：沙盒存档

首先游戏存档需要是沙盒模式，沙盒模式里没有科技树限制，
kOS 功能没有功能阉割，便于学习。

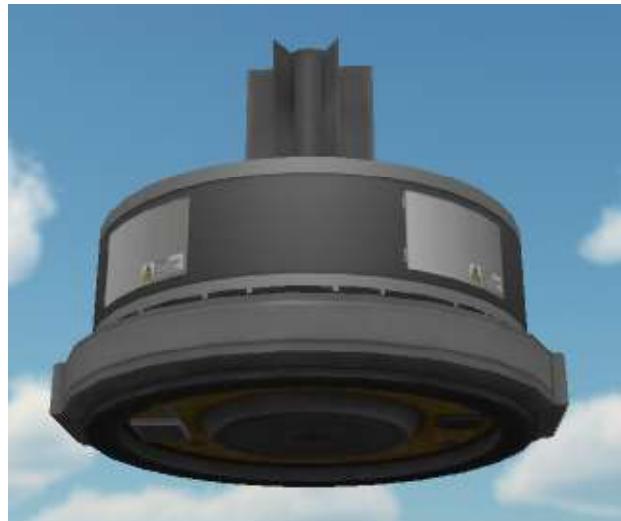
本文中所有实例均在沙盒模式中进行。

Step 2：捏一个载具

进 VAB/SPH 车间，用下面三个部件，捏一个载具。



载具里只要有指令部件，有 kOS 部件，有持续供电，kOS 就能工作了。



Step 3: 发射载具

点击 Launch 按钮，发射这个载具。



Step 4: 打开指令窗口

点击[红框1]叫出 kOS 控制面板，看到控制左侧列表里是刚才捏的未命名载具，该载具下属1个名为“CX-4181”的 kOS 部件。

我们再点击 [红框2] 来打开这个 kOS 部件对应的指令窗口。



打开后的指令窗口如下图：



*注1：kOS 的窗口和 KSP 的 DeBug 窗口、其他 MOD 的控制面板一样，都是要鼠标选中窗口后才能操作的，否则就只是对游戏中载具的操作。

一般的，窗口没被选中时会比选中时更加透明一些，玩家可以按此判断窗口状态。

*注2：“CX-4181”部件右键菜单里点“Open Terminal”也可以打开终端界面。如下图。



Step 5: 使用指令窗口

kOS 和其他可编程系统一样，都是靠执行指令语句来工作的。
kOS 的指令语句以句点 . 作为语句结束的标志。

选中指令窗口后，就可以键盘输入指令语句了。输好指令语句后按回车即可运行。

记住几个特点：

- kOS 对代码的大小写不敏感，即使是字符串也是部分大小写的。
- kOS 会无视多余的空格、制表符、换行等排版字符。
- 多条指令也可以输在同一行，一齐运行。

玩家可以在指令窗口中直接输入指令让 kOS 执行。
也可以将多个指令打包成代码文件，让 kOS 以运行这个程序文件。

((1)) 一个直接输入指令的例子

我们输入下面的代码并回车执行，能得到下图。
(ClearScreen 是清屏指令，Print 则是显示指令)

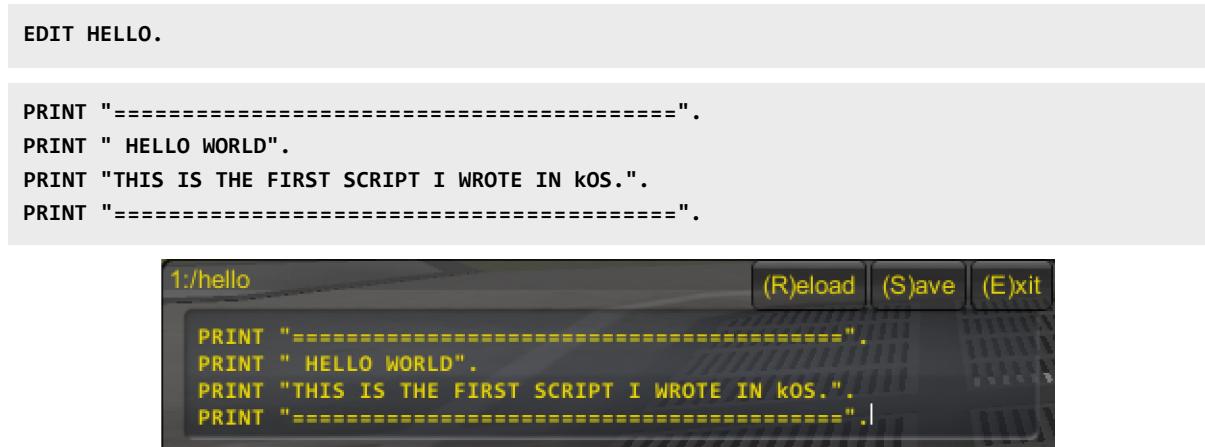
```
CLEARscreen. PrinT "Hello World".
```



((2)) 一个代码文件的例子

我们先输入并执行下面第一段命令（1行）。(HELLO 就是新建的代码文件的文件名)

此时 kOS 会弹出一个文本编辑窗让我们编辑新建文件的内容，我们输入/复制第二段命令（4行）进去，并点击 Save。



```

EDIT HELLO.

PRINT "=====".
PRINT " HELLO WORLD".
PRINT "THIS IS THE FIRST SCRIPT I WROTE IN kOS.".
PRINT "=====".

```

玩家可以在文本编辑窗里粘贴 Windows 剪贴板的内容，但是在指令窗口里不能。

然后我们选中指令窗口，输入以下代码并回车。

kOS 会执行刚才保存好的那个 HELLO 文件，结果如下图：



```

RUN HELLO.

Untitled Space Craft CPU: CX-4181 () 

Hello World
edit hello.
[New File]
[Saved changes to Archive:/hello]
run hello.
=====
HELLO WORLD
THIS IS THE FIRST SCRIPT I WROTE IN kOS.
=====
Program ended.

```

实际编程时建议使用 **RunPath("hello")**。, 不要用 Run hello.。

因为 RunPath()函数和 Run 指令相比功能更强大，详情可见 8.1 运行程序。

Step 6: kOS 中的文件系统

想知道刚才的 Hello 程序文件是存在哪的吗？

想要使用好 kOS，搞清楚 kOS 的文件系统是很重要的。

((1)) kOS 中的文件分类（按编码与用途分）

kOS 允许玩家像普通电脑一样管理和编辑文件，但是编辑仅限于文本文件。kOS 无法编辑二进制文件。

kOS 所管理的文件，按用编码和用途主要可以分为下面几类：

序号	编码	类别	用途说明
1	文本文件	程序代码文件(源文件)	能被 kOS 运行计算, 内容是指令语句的集合, 由玩家编写。用于保存玩家写的程序
2		例子:	火箭的发射入轨程序, 地面车辆的驾驶程序
3	json文件	自定义内容格式, 识别函数需要玩家自行在 kOS 中编写, 文件内容可编写成格式, 可用于和外部软件交换数据。	kOS 具有这种数据交换文件的识别功能, 由 kOS 生成。用于快速的从 kOS 导入 / 导出游戏数据
4		其他文本文件	由地图分析程序生成地面车辆驾驶路线的路径点列表。导出到 json 后, 由车辆驾驶程序读取并执行。
5	二进制文件	例子:	火箭发射时每秒采集一次飞行状态 (高度, 速度, 方向, 质量, 推力等) 以 csv 文件格式记录在
6		ksm文件	由程序代码文件编译成机器码的二进制文件, 是 kOS 的可执行文件
7	其他二进制文件	例子:	无
8		例子:	譬如, 你可以把一首mp3歌曲从外面拷进来

((2)) kOS 中的文件分类 (按存储位置分)

kOS 为了模拟真实的航天器/飞行器, 对文件的存储位置和寿命做了特别的设置。

kOS 允许玩家访问各种文本文件的内容, 根据存储位置不同主要分为两类, 如下表。

	第一类	第二类	
	自机机载电脑的文件	航天中心的文件	僚机机载电脑的文件
可访问性	可访问	与航天中心 (KSC) 有通讯信号时可访问	不可访问
游戏中位置	自机的某个 kOS 部件的存储空间内	航天中心 (KSC) 的数据库中 (云端)	僚机的某个 kOS 部件的存储空间内
文件大小	不超过 kOS 部件的存储容量	无限制 (因为 KSC 的数据库空间无限)	不超过 kOS 部件的存储容量
模拟真实的	载具机载计算机的硬盘	航天中心的数据库	载具机载计算机的硬盘
在玩家电脑中的位置	KSP.exe 的运行内存	KSP 游戏安装目录下的 Ships / Script 文件夹	KSP.exe 的运行内存
寿命	当 kOS 部件损毁时文件消灭	无限 (即使关闭游戏文件也依然存在)	当 kOS 部件损毁时文件消灭
例子:	载人舱在 Mun 背面无通讯信号时的自动着陆程序	记录火箭发射中空气阻力并导出成 csv 的程序, 之后要用 Excel 绘制其曲线	导弹的飞控程序

((3)) 卷 (Volume)

卷 (Volume) 是 kOS 中文件存储空间的称呼。

如果把 kOS 部件看作载具的机载计算机, 那么 Volume 就是这个机载计算机的硬盘空间。

每一个 kOS 部件 都有且仅有一个硬盘分区（Volume），
航天中心（KSC）的也对应一个 Volume，相当于一个云盘。

这点类似普通电脑上的本地硬盘和远程硬盘（云端），后者是其他机器共享出来让我们访问的。

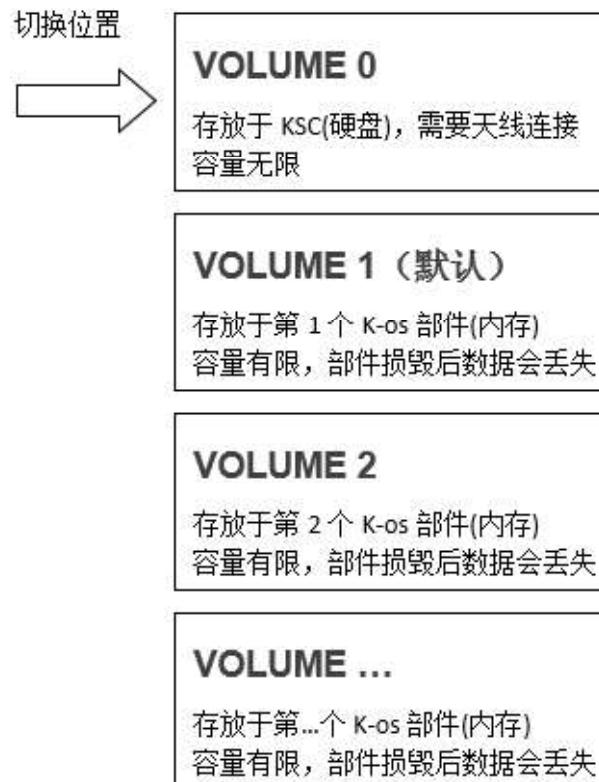


kOS 将这些 卷 Volume 按照编号命名，Volume 0 是指航天中心（KSC）的 Volume，
当前 kOS 部件 的则是 Volume 1，
自机载具上如果还有其他 kOS 部件，则从 Volume 2 开始命名。

一个 kOS 部件只能同时切换到一个 Volume，所有的文件操作都是以当前选择的 Volume 为对象进行的。

玩家如果要对其他卷进行文件操作必须先切换（Switch）到那个卷。

```
Switch to 0. //切换到 Volume 0, 这是位于 KSC 的无限存储空间
Switch to 1. //切换到 Volume 1, 这是当前 kOS 部件的存储空间
Switch to 2. //切换到 Volume 2, 这是自机上其他 kOS 部件 的存储空间
```



((4)) 刚才的 Hello 文件在哪?

回过来看刚才程序文件的编辑窗口, 左上角有 1:/hello 字样。

这代表这是一个名为 hello 的文件, 位于 Volume 1 的根目录下。

```
1:/hello
(R)eload (S)ave (E)xit
PRINT ======.
PRINT " HELLO WORLD".
PRINT "THIS IS THE FIRST SCRIPT I WROTE IN KOS.".
PRINT ======.
```

((5)) List Files 和 List Volumes 指令

我们可以用如下命令来查看当前卷 Volume 中的所有文件 File。结果如下:

LIST FILES.

```
list files.
/
Name          Size
-----
hello          176
Free space remaining: 9824
```

上图表示当前卷中只有一个 hello 文件, 他的大小是 176 字节。

我们还可以用如下命令来查看当前载具能访问到的所有卷 Volume。

LIST VOLUMES.

```
list volumes.

Volumes
ID      Name          Size
-----
0       Archive        -1
1*      Volume1       10000

```

关于上图的说明：

- ((1))** Volume 编号 1 右边的星号 * 表示当前选中的是这个卷。
- ((2))** 依赖于 kOS 部件的卷都有存储容量上限，例如 Volume1 至多能存 10000 字节的文件。而依赖于 KSC 的卷是无限存储容量的（写作 -1）。
- ((3))** 每个卷除了可以用编号进行指称，还可以用卷名（字符串）进行指称，Volume 0 的卷名是“Archive”。例子：SWITCH to 0. 和 SWITCH to Archive. 是一个意思。

((6)) 运行 Volume 0 中的程序

以实际例子作说明：

我们在指令窗口逐行输入并运行下面的代码。

先切到 ID=0 的卷；

接着看看里边内容，可以看到里边有文件 abc 和文件夹 boot；

再运行 abc 程序，可以看到 abc 程序显示了“www”三个字母。

```
SWITCH to 0.
LIST FILES.
run abc.
```

```
switch to 0.
list files.

/
Name          Size
-----
boot          <DIR>
abc           13
Free space remaining: infinite

run abc.
www
Program ended.
```

在 KSC 的设置菜单里边，整合有 kOS 的设置，里面直接有一项可以让你选择，
默认 Volume 是要设成 Volume 1（当前 kOS 部件），还是要设成 Volume 0（Archive）。

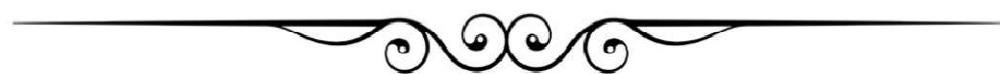
要访问 Volume 需要保持通讯信号，如果玩家不打算出大气层，或者只需在近地轨道活动。
推荐把这里的“Start on the archive”给勾上，这样程序直接放在 Ships/Script 文件夹里，
打开指令窗口就可以直接运行，不用考虑 volume 切换的问题了，省事。



4.1.3

4.1.3 实例2：控制火箭发射

===== 点击以返回目录 =====

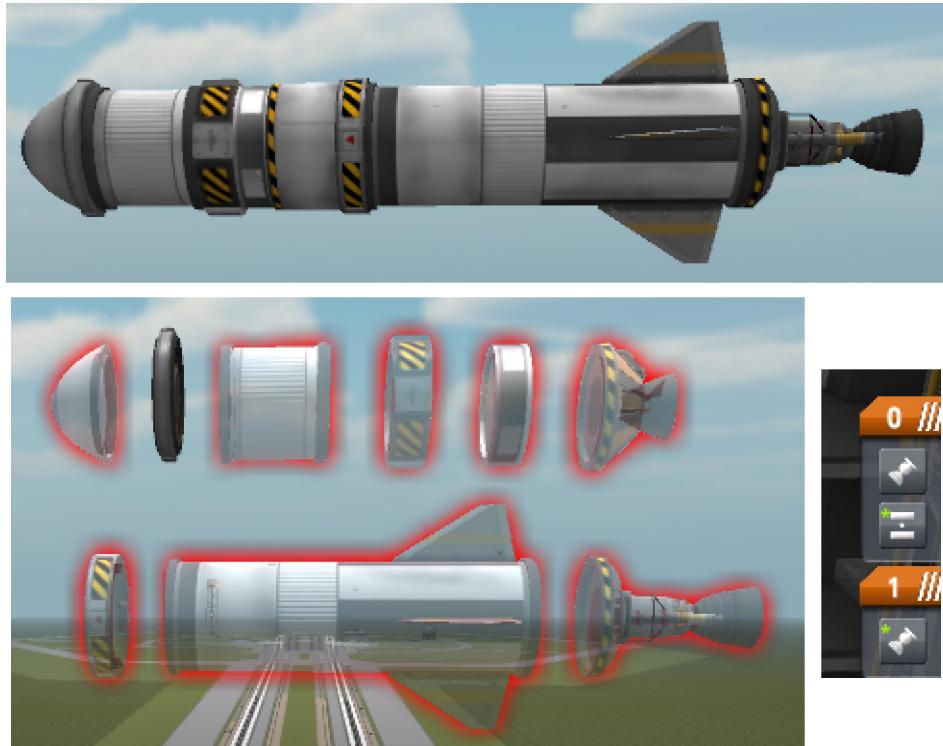


建议玩家在初学 kOS 时必看。

本实例中，我们将捏一个火箭，控制他大气中的发射飞行。
我们将从零开始，逐步为火箭完善一些控制功能。

Step 1: 捏一枚火箭

我们在 VAB 车间里捏一枚两级火箭，如下图。
注意一级火箭的引擎是有矢量的那个，二级火箭的姿态我们就要靠动量轮来撑了，
注意要把上一级火箭的分离和下一级火箭的点火放在同一个 Stage 里。



然后点击 Launch 转到发射台上。

Step 2: 显示发射倒数

我们来为火箭加上发射倒数功能。

我们在安装目录下的 Ships/Script 文件夹里，用 txt 新建一个文本文件，
输入/复制以下程序代码，然后保存为一个无后缀名的文件 test。

test 文件内容：

```
//我是注释

CLEARSCREEN. //首先，我们先清屏

PRINT "Counting down:".
//From结构是个循环结构，类似C语言里的for循环，做的是倒计时 10 ~ 1
FROM {local N is 10.} UNTIL N = 0 STEP {SET N to N - 1.} DO
{
    PRINT "... " + N. //在终端界面上显示 ...N
    WAIT 1. // 等待1秒钟。
}
```

C 系统(固态) (C:) ▶ Lookerksy_Program ▶ Kerbal Space Program v.1.3.1 Steam ▶ Ships ▶ Script ▶

工具(T) 帮助(H)

录 新建文件夹

名称	修改日期	类型	大小
boot	2017/12/14 20:03	文件夹	
abc	2017/12/20 22:07	文件	1 KB
test	2018/1/1 20:30	文件	1 KB

test - 记事本

文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)

```
//我是注释

CLEARSCREEN. //首先，我们先清屏

PRINT "Counting down:".
//From结构是个循环结构，类似C语言里的for循环，做的是倒计时 10 ~ 1
FROM {local N is 10.} UNTIL N = 0 STEP {SET N to N - 1.} DO
{
    PRINT "..." + N. //在终端界面上显示 ...N
    WAIT 1. // 等待1秒钟.
}
```

在 KSP 的指令窗口里，我们逐行输入并运行以下代码，
第一行表示切换到 Volume 0，第二行表示运行 Volume 0 下（Ships/Script 里）的 test 文件。

```
switch to 0.
run test.
```

运行结果截图：



现在这枚火箭能倒数了，我们接下来做下一步。

Step 3: 点火发射!

然后我们来为程序加上点火的功能。

然后我们返回到发射前，

我们要在上一步那个 test 代码文件里加上让火箭“点火升空，然后出大气层后引擎关机”的代码，然后重新再来一遍之前指令窗口里的操作。

test 文件内容：

```
//我是注释

//节流阀拉到最大，节流阀范围 0.0 是最小，1.0 是最大。
LOCK THROTTLE TO 1.0.

//首先，我们先清屏
CLEARSCREEN.

//以下的From结构是个循环，类似C语言里的for循环，做的是倒计时 10 ~ 1
PRINT "Counting down:".
FROM {local N is 10.} UNTIL N = 0 STEP {SET N to N - 1.} DO
{
    PRINT "... " + N. //在终端界面上显示 ...N
    WAIT 1. // 等待1秒钟.
}

Stage. //相当于按空格

//等到火箭到达 70000 m 高度时，程序结束
WAIT UNTIL SHIP:ALTITUDE > 70000.

//没有后续要运行的代码就代表程序结束，
//程序结束时，kOS会释放所有控制。
```

运行结果截图：



现在这枚火箭能上天了，但是飞行时无法控制方向。而且第一级火箭关机后，就没动作了。

Step 4：控制火箭姿态

然后我们来为程序加上姿态控制的功能。

然后我们返回到发射前，

我们在上一步那个 test 代码文件里加上让火箭“姿态稳定朝上”的代码，
然后重新再来一遍之前指令窗口里的操作。

test 文件内容：

```

//我是注释

//火箭朝向设置为竖直向上,
//UP是一个严格朝向值, 包含 滚转Roll 角度,
//如果车间里火箭方向没放对的话,
//发射时会滚转一定角度以符合 UP 朝向
LOCK STEERING TO UP.

//节流阀拉到最大, 节流阀范围 0.0 是最小, 1.0 是最大。
LOCK THROTTLE TO 1.0.

//首先, 我们先清屏
CLEARSCREEN.

//以下的From结构是个循环, 类似C语言里的for循环, 做的是倒计时 10 ~ 1
PRINT "Counting down:".
FROM {local N is 10.} UNTIL N = 0 STEP {SET N to N - 1.} DO
{
    PRINT "... " + N. //在终端界面上显示 ...N
    WAIT 1. // 等待1秒钟.
}

Stage. //相当于按空格

//等到火箭到达 70000 m 高度时, 程序结束
WAIT UNTIL SHIP:ALTITUDE > 70000.

//没有后续要运行的代码就代表程序结束,
//程序结束时, kOS会释放所有控制。

```

运行结果截图：



现在这枚火箭能没有偏斜的稳定向上飞了，但是第一级火箭关机后，仍旧会没动作。

Step 5：火箭分级功能

然后我们来为程序加上火箭分级的功能。

然后我们返回到发射前，

我们在上一步那个 test 代码文件里加上让火箭“一旦没油失去动力，就进行火箭分级”的代码，然后重新再来一遍之前指令窗口里的操作。

test 文件内容：

```

//我是注释

//火箭朝向设置为竖直向上
//UP是一个严格朝向值，包含 滚转Roll 角度，
//如果车间里火箭方向没放对的话，
//发射时会滚转一定角度以符合 UP 朝向
LOCK STEERING TO UP.

//节流阀拉到最大，节流阀范围 0.0 是最小，1.0 是最大。
LOCK THROTTLE TO 1.0.

//首先，我们先清屏
CLEARSCREEN.

//以下的From结构是个循环，类似C语言里的for循环，做的是倒计时 10 ~ 1
PRINT "Counting down:".
FROM {local N is 10.} UNTIL N = 0 STEP {SET N to N - 1.} DO
{
    PRINT "... " + N. //在指令窗口上显示 ...N
    WAIT 1. // 等待1秒钟.
}

//如果火箭的最大推力=0，那燃料肯定烧光了，那么就按空格，
//这种写法可以适应所有多级火箭，不管二级火箭、三级火箭，甚至更多级，都能正常工作。
WHEN MAXTHRUST = 0 THEN {
    PRINT "Staging".
    STAGE.
    //RETURN 返回的值代表是否要保留这个触发结构
    //true 的话就代表这个触发结构会继续工作
    //false 的话这个触发结构就寿命到了，消失了×
    RETURN true.
}.

//等到火箭到达 70000 m 高度时，程序结束
WAIT UNTIL SHIP:ALTITUDE > 70000.

//没有后续要运行的代码就代表程序结束，
//程序结束时，kOS会释放所有控制。

```

运行结果截图：



现在这枚火箭不仅能竖直上飞，所有分级点火都能顺利进行，火箭到达 70000m 高度后会自动关闭节流阀。

*注1：文中“WAIT UNTIL SHIP:MAXTHRUST = 0.” 也可以写成 “WAIT UNTIL STAGE:LIQUIDFUEL < 0.1”。这两句通常都可以用来判断当前分级的燃料是否烧完。不过各有局限。

SHIP:MAXTHRUST 的意思是“如果把节流阀开到最大，当前载具身上所有火箭的总推力是多少。但这种判断方式无法判断助推火箭是否燃料烧完。”

STAGE:LIQUIDFUEL 的意思是“当前级火箭能用到的燃料。”，是一个浮点值，火箭燃烧完之后，这个浮点值不会是完全的0，而是会有一些残留，所以需要用<0.1这种大约的条件来判断。

Step 6：控制火箭转向

然后我们在上一步的基础上加上一个函数结构，让火箭在不同速度时稳定在不同的姿态方向上。然后重新再来一遍之前指令窗口里的操作。

```

//我是注释

//声明一个自定义函数 head
function head {
    set v to Ship:AirSpeed. //设 v 为火箭的地面速度×
    set p to 90-sqrt(v)*1.9. //火箭的俯仰角 p 和 v 相关, 这是随便乱写的式子
    //Heading函数的第一个参数是指从正北方向顺时针转90°的水平方向（正东方）
    //第二个参数是指俯仰角, 90就是垂直, 0就是水平×
    return heading(90,p). //返回火箭的目标朝向, 其中heading
}

//火箭朝向设置为追随 head 函数返回的朝向值
LOCK STEERING TO head().

//节流阀拉到最大, 节流阀范围 0.0 是最小, 1.0 是最大。
LOCK THROTTLE TO 1.0.

//首先, 我们先清屏
CLEARSCREEN.

//以下的From结构是个循环, 类似C语言里的for循环, 做的是倒计时 10 ~ 1
PRINT "Counting down:".
FROM {local N is 10.} UNTIL N = 0 STEP {SET N to N - 1.} DO
{
    PRINT "... " + N. //在指令窗口上显示 ...N×
    WAIT 1. // 等待1秒钟.
}

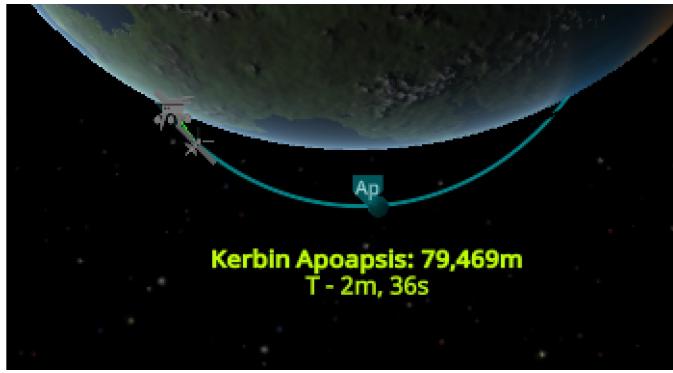
//如果火箭的最大推力=0, 那燃料肯定烧光了, 那么就按空格,
//这种写法可以适应所有多级火箭, 不管二级火箭、三级火箭, 甚至更多级, 都能正常工作。
WHEN MAXTHRUST = 0 THEN {
    PRINT "Staging".
    STAGE.
    //RETURN 返回的值代表是否要保留这个触发结构
    //true 的话就代表这个触发结构会继续工作×
    //false 的话这个触发结构就寿命到了, 消失了×
    RETURN true.
}.

//等到火箭的轨道的AP点到达 80000 m 高度时, 程序结束
WAIT UNTIL SHIP:OBT:APOAPSIS > 80000.
SAS ON. //把SAS打开, 这样kOS程序结束之后火箭不会乱转。

```

运行结果截图:





现在火箭发射时有了完整的转向和分级逻辑行为，
AP 点到达 80000m 高度后火箭会自动关闭节流阀。基本实现火箭上升段的控制功能。

Step Extra: 通过预设变量和结构体操控载具

我们注意到，本实例中有一些代码直接对应了载具的驾驶操作，例如：

```

LOCK STEERING TO UP. //载具朝向锁定向上
.....
LOCK THROTTLE TO 1.0. //节流阀拉到最大（0 最小，1 最大），相当于按了Z键
.....
Stage. //执行下一个发射序列，相当于按了空格键
.....
WHEN MAXTHRUST = 0 THEN {.....}. //如果最大推力为 0 了，那就.....
.....
//一直等待，直到火箭轨道的AP点到达 80000 m 高度时，再开始执行后续代码
WAIT UNTIL SHIP:OBT:APOAPSIS > 80000.
//Ship是一个结构体，Obt是Ship下属的一个结构体，Apoapsis是Obt下属的一个数值
//结构体和下属内容之间由冒号 : 分隔，结构体可以下属有结构体、函数或者变量
.....
SAS ON. //把SAS打开，相当于按了T键。

```

这个就是 kOS 的一个特色功能了：kOS 提供了一些预设的变量和结构体，
这些变量/结构体的值和游戏挂钩，对这些变量/结构体进行赋值等同于在游戏里驾驶操作。

这些预设的变量/结构体大致可以分为下面几类：

变量 / 结构体	例子	例句 / 备注
变量	Throttle 数值	Lock Throttle to 0. 节流阀拉到0 (关闭)
	SAS 布尔量	SAS off. 关闭SAS
	ShipName 字符串	Set ShipName to "Q". 把自机载具名设为 Q
函数	WarpTo ()	WarpTo (Time : Seconds + 60). 时间加速到当前时间的60秒后 * kOS 中可直接影响游戏的函数屈指可数
指令	Stage	Stage. 执行下一个发射序列，相当于按了空格键 * kOS 中可直接影响游戏的函数屈指可数
结构体	Ship 结构体	自机载具的 Vessel 类结构体
...下属变量	Ship : Mass 数值	Print Ship : Mass. 在指令窗口显示自机质量
	Ship : ShipName 字符串	Print ShipName. 在指令窗口显示自机载具名称
...下属函数	Ship : MaxThrustAt ()	Print Ship : MaxThrustAt (0). 在指令窗口显示自机在真空下的总推力 参数为数值，范围0 (真空) ~ 1 (标准大气压)
...下属结构体	Ship : Orbit	自机载具 (Vessel 类结构体) 下属的 Orbit 类 结构体，实际上就是把轨道类信息打包成 一个新的结构体，方便存放

4.2

4.2 编程指导

4.2.1

4.2.1 导读

===== 点击以返回目录 =====



本节是 kOS 的一些编程指导事项。

本节内容：

实例3： kOS 中的基本表达式

- (1) 任务目标
- (2) 赋值指令
- (3) Print 指令 和 Log 指令
- (4) 顺序结构 实现目标
- (5) 循环结构+条件结构 实现目标
- (6) 触发结构 实现目标
- (7) 函数结构

(8) 混合使用顺序、循环与触发结构

实例4：kOS 程序的优化

- (1) 优化的重要性
- (2) 准备一个有传感器的载具
- (3) 优化方法：使用计算量最小的表达式
- (4) 优化方法：减少条件语句的判断
- (5) 优化方法：精简循环语句的运行
- (6) 优化方法：不要在触发模块内放置复杂语句
- (7) 优化方法：减少触发器数量，简化触发条件

4.2.2

4.2.2 实例3：kOS 中的基本表达式

===== 点击以返回目录 =====



建议玩家在初学 kOS 时必看。

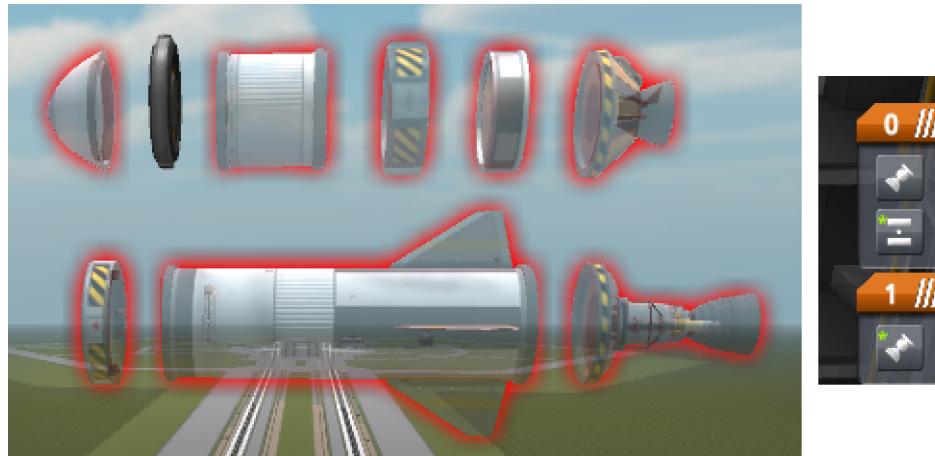
(1) 任务目标

接下来我们举例演示，通过不同的方式，来实现相同的事情：

- ((1)) 满推力垂直升空
- ((2)) 300m 高度改成距离垂直方向 10° 的俯仰角
- ((3)) 10000m 高度改成距离垂直方向 20° 的俯仰角
- ((4)) 20000m 高度改成距离垂直方向 30° 的俯仰角
- ((5)) 30000m 高度改成距离垂直方向 40° 的俯仰角
- ((6)) 40000m 高度时关闭引擎
- ((7)) 全程自动分级

我们用的载具还是和 实例2：控制火箭发射 里同样的火箭。





(2) 赋值指令

kOS 中有 Set 和 Lock 两种赋值指令。

Set是对变量直接赋值，就是通常我们认识的赋值。

```
Set a to 2. //此时 a=2
Set a to 3. //此时 a=3
```

Lock是对变量延迟赋值，每次调用变量时会重新计算Lock的表达式。

```
Set a to 2. //此时 a=2, b未定义, c未定义
Lock b to a. //此时 a=2, b=a=2, c未定义
Set c to b. //此时 a=2, b=a=2, c=2
Set a to 3. //此时 a=3, b=a=3, c=2
Set c to b. //此时 a=3, b=a=3, c=3
```

对于如 Throttle（节流阀）在内的，和驾驶直接相关的预设变量，
kOS会持续轮询其值，所以Lock指令用在这类变量上，就相当于持续更新表达式的值。

```
Set a to 0.2.
Lock Throttle to a. //此时 a=0.2, Throttle=a=0.2
.....
Set a to 0.4.      //此时 a=0.4, Throttle=a=0.4
.....
Set a to 0.6.      //此时 a=0.6, Throttle=a=0.6
```

(3) Print指令 和 Log指令

kOS中有两种常用的信息输出指令：Print指令 和 Log指令。

Print指令 是将信息输出到在kOS终端界面里，Print 后跟的内容会强制转换成字符串来显示。

```
Print "WOW". //终端界面显示: WOW
Set a to 123.
Print a+"WOW". //终端界面显示: 123WOW
```

Log指令 是将信息输出到文件里，通过Log指令 可实现信息记录。

Log指令会在每次执行的末尾，自动在文件里加上一个换行符。

```

Set a to 2.          //此时 a=2, b未定义, c未定义
Log a to "we.csv".  //此时 we.csv 文件内容为: 2
Log "WOW" to "we.csv".
//此时 we.csv 文件内容为:
2
WOW

```

(4) 顺序结构 实现目标

顺序结构就是那种从上往下逐条运行的。

```

LOCK STEERING TO HEADING (0,90). //朝向锁定向上
LOCK THROTTLE TO 1. //节流阀开到最大
WAIT 0.5. //等待0 .5秒
STAGE. //按下空格, 火箭发射, 第一级火箭点火

//300m 高度改成距离垂直方向10度的俯仰角
WAIT UNTIL SHIP:ALTITUDE > 300.
LOCK STEERING TO HEADING(0,90) + R (0,-10,0).

//1000m 高度改成距离垂直方向20度的俯仰角
WAIT UNTIL SHIP:ALTITUDE > 1000.
LOCK STEERING TO HEADING(0,90) + R (0,-20,0).

//2000m 高度改成距离垂直方向30度的俯仰角
WAIT UNTIL SHIP:ALTITUDE > 2000.
LOCK STEERING TO HEADING(0,90) + R (0,-30,0).

WAIT UNTIL SHIP:MAXTHRUST = 0. //等到第一级火箭烧完后
Stage. //按下空格, 分级, 一级火箭分离, 二级火箭启动

//3000m 高度改成距离垂直方向40度的俯仰角
WAIT UNTIL SHIP:ALTITUDE > 3000.
LOCK STEERING TO HEADING(0,90) + R (0,-40,0).

//4000m 高度时关闭引擎
WAIT UNTIL SHIP:ALTITUDE > 4000.
LOCK THROTTLE TO 0. //节流阀拉到0

WAIT UNTIL FALSE. //卡住程序, 不让程序结束, 要强行结束只能按 CTRL+C

```

*注1：这种方法有一个明显的弊端：要事先知道火箭分离是在什么高度。

(5) 循环结构+条件结构 实现目标

好，我们把同样的的内容改写成循环结构+条件结构的。

```

LOCK STEERING TO HEADING (0,90). //朝向锁定向上
LOCK THROTTLE TO 1. //节流阀开到最大
WAIT 0.5. //等待0 .5秒
STAGE. //按下空格，火箭发射，第一级火箭点火

UNTIL SHIP:ALTITUDE > 40000 { //在高度升到40000m高空之前都要不停循环×
    IF SHIP:ALTITUDE > 30000 { //如果高度升到超过30000m了×
        LOCK STEERING TO HEADING(0,90) + R (0,-40,0). //俯仰角往水平转到40°
    } ELSE IF SHIP:ALTITUDE > 20000 { //如果高度升到超过20000m了×
        LOCK STEERING TO HEADING(0,90) + R (0,-30,0). //俯仰角往水平转到30°
    } ELSE IF SHIP:ALTITUDE > 10000 { //如果高度升到超过10000m了×
        LOCK STEERING TO HEADING(0,90) + R (0,-20,0). //俯仰角往水平转到20°
    } ELSE IF SHIP:ALTITUDE > 300 { //如果高度升到超过300m了×
        LOCK STEERING TO HEADING(0,90) + R (0,-10,0). //俯仰角往水平转到10°
    }×
    IF SHIP:MAXTHRUST = 0 { //如果当前级火箭的燃料烧完了×
        STAGE. //按下空格切换到下一级火箭
    }×
    Wait 0.001. //等待到下一个物理帧
}

LOCK THROTTLE TO 0. //节流阀拉到0

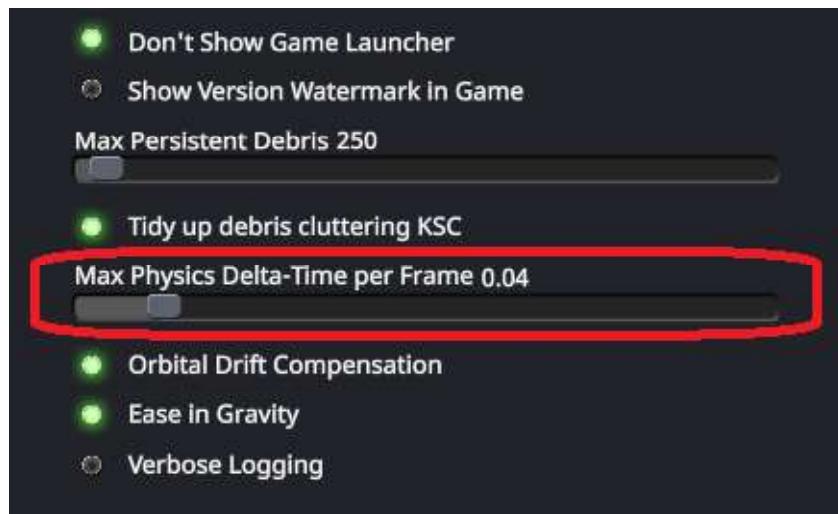
WAIT UNTIL FALSE. //卡住程序，不让程序结束，要强行结束只能按 CTRL+C

```

请注意上文中的 Wait 0.001. 语句，这里涉及到 KSP 和 kOS 中的一个重要概念：物理帧。

在 KSP 中时间并不是连续的，而是由一个个微小的时间间隔组成的。
 在微小时间间隔两头的，则是 KSP 中进行全宇宙物理演算的最小单位——物理帧。
 一个物理帧相当于宇宙的一张快照，包含了此静止时刻宇宙中的全部信息，比如星球运行的位置和速度，载具运行的位置和速度、资源量和热量，等等。
 KSP 和 kOS 在相邻两个物理帧之间进行演算，按照上一个物理帧的信息，计算下一个物理帧信息。例如这一帧飞船的姿态方向偏了，kOS 可以立即算出对应的调姿修正，飞船会在下一个物理帧执行这些操作。

我们可以在 KSP 的设置菜单中找到，设置物理帧之间间隔时间的地方，如下。



Wait 0.001. 语句的原意是执行长度为 0.001 秒的等待，但实际上这个时间长度肯定小于一个物理帧的时间长度（例如上图中的 0.04 秒），所以实际效果是，等到到下一个物理帧来到为止。此语句可用于避免在同一个物理帧里进行多次循环计算。

(6) 触发结构 实现目标

好，我们把同样的内容改写成触发结构。触发结构在代码里是最优先运行的。

```
//利用触发结构自动执行火箭分级
WHEN SHIP:MAXTHRUST = 0 THEN { //如果燃料用完了，就按空格×
    STAGE. //按空格，切换到下一级火箭×
    Return True. //这个触发结构是可重复触发的，不是一次性的
}

//利用触发结构的嵌套，自动执行在高度节点的转向
WHEN SHIP:ALTITUDE > 300 THEN { //高度超过300m后，运行一次以下指令
    LOCK STEERING TO HEADING(0,90) + R(0,-10,0). //俯仰角往水平转到10°×
    WHEN SHIP:ALTITUDE > 10000 THEN { //高度超过10000m后，运行一次以下指令
        LOCK STEERING TO HEADING(0,90) + R(0,-20,0). //俯仰角往水平转到20°×
        WHEN SHIP:ALTITUDE > 20000 THEN { //高度超过20000m后，运行一次以下指令
            LOCK STEERING TO HEADING(0,90) + R(0,-30,0). //俯仰角往水平转到30°×
            WHEN SHIP:ALTITUDE > 30000 THEN { //高度超过30000m后，运行一次以下指令
                LOCK STEERING TO HEADING(0,90) + R(0,-40,0). //俯仰角往水平转到40°
            }
        }
    }
}

LOCK STEERING TO HEADING(0,90). //朝向锁定向上
LOCK THROTTLE TO 1. //节流阀开到最大
WAIT 0.5. //等待0.5秒
STAGE. //按下空格，火箭发射，第一级火箭点火

WAIT UNTIL SHIP:ALTITUDE > 40000. //等到升到40000m高空后

LOCK THROTTLE TO 0. //节流阀拉到0

WAIT UNTIL FALSE. //卡住程序，不让程序结束，要强行结束只能按 CTRL+C
```

*注1：全局有效的触发结构应写在代码的最前。

*注2：本程序中火箭的转向是通过触发结构嵌套使用来实现的，每个触发结构从生效到触发之间，都在不停轮询自己的触发条件，这种嵌套的写法能推迟触发结构的生效时间，从而减少计算量，减少 kOS 的计算压力。

*注3：触发语句中的 Wait 语句是无效的！

(7) 函数结构 实现目标

好，我们把同样的内容改写成函数结构。LOCK STEERING TO 命令锁定到一个动态的值 —— head

函数上，而 head 函数随着飞船高度不同而不同，是个分段函数。

```
//声明一个自定义函数 head
function head {
    Parameter alt. //函数的自变量×
    If alt<300 { //一个分段函数。
        Set p to 0.
    } else if alt < 10000 {
        Set p to 10.
    } else if alt < 20000 {
        Set p to 20.
    } else if alt < 30000 {
        Set p to 30.
    } else {
        Set p to 40.
    }
    //第二个参数是指俯仰角，90就是垂直，0就是水平×
    return heading(90,90-p). //返回火箭的目标朝向，其中heading
}

WHEN SHIP:MAXTHRUST = 0 THEN { //如果燃料用完了，就按空格×
    STAGE. //按空格，切换到下一级火箭×
    Return True. //这个触发结构是可重复触发的，不是一次性的
}

LOCK STEERING TO HEAD(SHIP:ALTITUDE). //朝向锁定到以飞船高度为自变量的head函数上

LOCK THROTTLE TO 1. //节流阀开到最大
WAIT 0.5. //等待0.5秒
STAGE. //按下空格，火箭发射，第一级火箭点火

WAIT UNTIL SHIP:ALTITUDE > 40000. //等到升到40000m高空后

LOCK THROTTLE TO 0. //节流阀拉到0
WAIT UNTIL FALSE. //卡住程序，不让程序结束，要强行结束只能按 CTRL+C
```

*注1：函数的声明结构也应写在代码的最前。

(8) 混合使用顺序、循环与触发结构

以上几步展现了 kOS 在编程时常用的顺序控制结构，仅供演示。
实际应用时，玩家需将多种结构灵活组合，最有效率地实现目标。

4.2.3

4.2.3 实例4：kOS 程序的优化

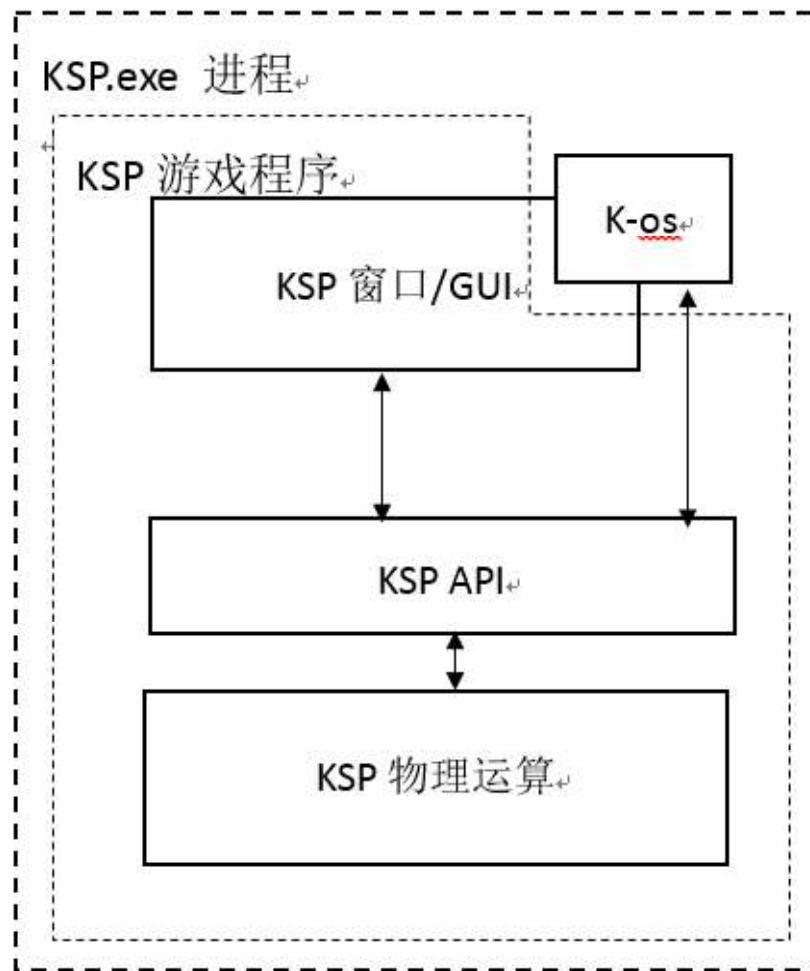
===== 点击以返回目录 =====



建议玩家在初学 kOS 时选看。

(1) 优化的重要性

kOS 是一个嵌入 KSP 游戏的插件，他通过对接游戏自身的载具操控 API 来实现载具控制。kOS 和游戏共同占用 KSP.exe 进程，使用的是玩家游戏时游戏本体用剩下的计算资源。



所以，留给 kOS 的计算资源本来就不多，所以在执行复杂程序的时候，kOS 很容易出现“严重影响效率”的情况。这更要求玩家在编程时的优化水平。

(2) 优化方法：使用计算量最小的表达式

比如说，二次函数有求根公式：

`Solve[a x2 + b x + c == 0, x]`

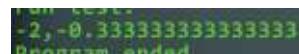
解方程

$$\left\{ \left\{ x \rightarrow \frac{-b - \sqrt{b^2 - 4ac}}{2a} \right\}, \left\{ x \rightarrow \frac{-b + \sqrt{b^2 - 4ac}}{2a} \right\} \right\}$$

其中的 $\frac{\sqrt{b^2 - 4ac}}{2a}$ 是 x_1, x_2 这两个根共有的东西，可以避免此项的重复计算，

相应的 kOS 程序级计算结果如下。（随便编几个数，假设是 $3x^2 + 7x + 2 = 0$ ）

```
//二次函数 a*x*x + b*x + c = 0
set a to 3. set b to 7. set c to 2. // 3*x^2 + 7*x + 2 = 0
set delta to b*b - 4*a*c.
if delta < 0 {
    print "None Root".
} else if delta=0 {
    set x to -b/2/a.
    print x. //一个解
} else {
    set xf to -b/2/a.
    set xb to sqrt(delta)/2/a.
    //分别计算x根式的两部分，然后拼起来，避免了一些重复计算×
    set x1 to xf-xb. set x2 to xf+xb.
    print x1+","+x2.
}
```



(3) 优化方法：减少条件语句的判断

当 if 结构中要判断的并列条件较多的时候，最好将它们拆分成多个 if 语句结构，然后嵌套在一起，就可以减少不必要的判断。

```
LOCK STEERING TO HEADING (0,90). //朝向锁定向上
LOCK THROTTLE TO 1. //节流阀开到最大
WAIT 0.5. //等待0 .5秒
STAGE. //按下空格，火箭发射，第一级火箭点火

UNTIL SHIP:ALTITUDE > 40000 { //在高度升到40000m高空之前都要不停循环×
    IF SHIP:ALTITUDE > 30000 { //如果高度升到超过30000m了×
        LOCK STEERING TO HEADING(0,90) + R (0,-40,0). //俯仰角往水平转到40°
    } ELSE IF SHIP:ALTITUDE > 20000 { //如果高度升到超过20000m了×
        LOCK STEERING TO HEADING(0,90) + R (0,-30,0). //俯仰角往水平转到30°
    } ELSE IF SHIP:ALTITUDE > 10000 { //如果高度升到超过10000m了×
        LOCK STEERING TO HEADING(0,90) + R (0,-20,0). //俯仰角往水平转到20°
    } ELSE IF SHIP:ALTITUDE > 300 { //如果高度升到超过300m了×
        LOCK STEERING TO HEADING(0,90) + R (0,-10,0). //俯仰角往水平转到10°
    }×
    IF SHIP:MAXTHRUST = 0 { //如果当前级火箭的燃料烧完了×
        STAGE. //按下空格切换到下一级火箭
    }
    //Wait 0.001. //这句话被注释掉了
}

LOCK THROTTLE TO 0. //节流阀拉到0

WAIT UNTIL FALSE. //卡住程序，不让程序结束，要强行结束只能按 CTRL+C
```

例如之前在 实例3： kOS 程序的顺序控制 的 循环结构+条件结构 部分，就在高度判断使用了嵌套的条件结构。

如果是每个条件语句都分开来独立写，如下，那么计算量肯定是很大的。

```

IF SHIP:ALTITUDE > 300 { //如果高度升到超过300m了
    LOCK STEERING TO HEADING(0,90) + R (0,-10,0). //俯仰角往水平转到10°
}
IF SHIP:ALTITUDE > 10000 { //如果高度升到超过10000m了
    LOCK STEERING TO HEADING(0,90) + R (0,-20,0). //俯仰角往水平转到20°
}
IF SHIP:ALTITUDE > 20000 { //如果高度升到超过20000m了
    LOCK STEERING TO HEADING(0,90) + R (0,-30,0). //俯仰角往水平转到30°
}
IF SHIP:ALTITUDE > 30000 { //如果高度升到超过30000m了
    LOCK STEERING TO HEADING(0,90) + R (0,-40,0). //俯仰角往水平转到40°
}

```

(4) 优化方法：精简循环语句的运行

在程序中，最影响代码速度的往往是循环结构，特别是多层嵌套的循环结构。

使用循环虽然简单，但是使用不当，往往可能带来很大的性能影响。循环结构的优化原则是将问题充分分解为小的循环，不在循环内的多余的工作（如赋值、常量计算等），避免死循环。还可以考虑将循环改为非循环来提高效率。

常用的循环优化手段如下：

降低循环层数。算法的事件复杂度主要是由循环嵌套的层数决定的，所以，算法中如果能够减少循环嵌套的层数，如将双重循环的内层循环展开，从而改写成单循环，则从事件复杂度上可达到降价的目的。

例如下面：计算一个3*3的二维数组内所有数累加的和。

```

//优化前
Set w1 to List(1,2,3). Set w2 to List(4,5,6). Set w3 to List(7,8,9). //三个一维数组
Set w to List(w1,w2,w3). //并成一个3*3二维数组，总和为 45
Set S to 0. //和初始化为 0
FROM {Local i is 0.} UNTIL i = 3 STEP {Set i to i+1.} DO {
    FROM {Local j is 0.} UNTIL j = 3 STEP {Set j to j+1.} DO {
        Set S to S + w[i][j].
    }
}
Print S.

```

//优化后

```

Set w1 to List(1,2,3). Set w2 to List(4,5,6). Set w3 to List(7,8,9). //三个一维数组
Set w to List(w1,w2,w3). //并成一个3*3二维数组，总和为 45
Set S to 0. //和初始化为 0
FROM {Local i is 0.} UNTIL i = 3 STEP {Set i to i+1.} DO {
    Set S to S + w[i][0].
    Set S to S + w[i][1].
    Set S to S + w[i][2].
}
Print S.

```

把循环内的重复运算提到循环外。这样可以避免每次循环过程中的重复计算。

这个很好理解，就不举例了。

合并循环。把两个或两个以上的循环合并放到一个循环里，这样会加快速度。

例如下面：计算一个3*3的二维数组内所有数累加的和，还有累乘的积。

```
//优化前
Set w1 to List(1,2,3). Set w2 to List(4,5,6). Set w3 to List(7,8,9). //三个一维数组
Set w to List(w1,w2,w3). //并成一个3*3二维数组，总和为 45
Set S to 0. //和初始化为 0
FROM {Local i is 0.} UNTIL i = 3 STEP {Set i to i+1.} DO {
    FROM {Local j is 0.} UNTIL j = 3 STEP {Set j to j+1.} DO {
        Set S to S + w[i][j].
    }
}
Set P to 1. //积初始化为
FROM {Local i is 0.} UNTIL i = 3 STEP {Set i to i+1.} DO {
    FROM {Local j is 0.} UNTIL j = 3 STEP {Set j to j+1.} DO {
        Set P to P * w[i][j].
    }
}
Print "S = "+S+" , P = "+P.
```

```
//优化后
Set w1 to List(1,2,3). Set w2 to List(4,5,6). Set w3 to List(7,8,9). //三个一维数组
Set w to List(w1,w2,w3). //并成一个3*3二维数组，总和为 45
Set S to 0. //和初始化为 0
Set P to 1. //积初始化为
FROM {Local i is 0.} UNTIL i = 3 STEP {Set i to i+1.} DO {
    FROM {Local j is 0.} UNTIL j = 3 STEP {Set j to j+1.} DO {
        Set S to S + w[i][j].
        Set P to P * w[i][j].
    }
}
Print "S = "+S+" , P = "+P.
```

(5) 优化方法：不要在触发模块内放置复杂语句

kOS 程序中对触发结构的优化很重要。

kOS 的触发模块是为了控制方便设计的，并且需要在一个物理帧之内全部运行完，本身并不适合进行复杂计算。

为此，在触发模块内的 wait 语句甚至是无效的。

另外，原则上不要在触发机构里安排进循环语句。除非是那种能立即完成的，计算量小的循环语句。

举个例子，我们准备好 实例2： 控制火箭发射 中的火箭，但是在上面多放了一个重力传感器和一个加速度传感器。这两个东西很小，不会影响火箭平衡。



我们来做一件事情：用这个载具垂直向上飞，但是要控制加速度在2个G左右，为此算法里需要用到一下反馈。

先看一下下面这个写法。

```

SET thr TO 1.
LOCK THROTTLE TO thr.//因为还没开始反馈，所以节流阀的反馈修正值先设到 0
LOCK STEERING TO UP.//朝向锁定到向上
WAIT 1.
STAGE.//等1秒再点火发射
WAIT 1.
WHEN SHIP:ALTITUDE > 100 THEN { //火箭高度超过100m后开始调节×
    SET g TO KERBIN:MU / KERBIN:RADIUS^2.//地面重力加速度 9.8
    //加速度矢量和重力矢量的差×
    LOCK accvec TO SHIP:SENSORS:ACC - SHIP:SENSORS:GRAV.
    LOCK gforce TO accvec:MAG / g.//计算加速度，Mag是指矢量的模
    //加速度设定一个目标值 2，和目标值的误差乘以一个系数，之后作为反馈×
    LOCK dthr TO 0.05 * (2 - gforce).
    UNTIL SHIP:ALTITUDE > 40000 { //火箭超过40000m之前都进行调节×
        WHEN STAGE:LIQUIDFUEL < 0.1 THEN { //“没油就按空格”的触发结构×
            STAGE.
        }×
        SET thr to thr + dthr.//将反馈修正值应用于节流阀(thr)
        //从300m高度就开始在屏幕上刷新加速度gforce×
        If SHIP:ALTITUDE > 300 {print gforce.}.
        WAIT 0.1.//每隔0.1秒调节一次
    }
}
wait until false. //卡住程序，不让程序结束，要强行结束只能按 CTRL+C

```

上面这种写法其实是无法运行的。when.....then.....的触发结构需要在当前物理帧之内运行完的，但是里头却有个 UNTIL SHIP:ALTITUDE > 40000 语句，还是到40km高度之后才结束的。这种情况下就很容易出错，程序运行的结果是：分级也不给你分，节流阀也不给你调，只有加速度

还在每秒10下刷新着.....

然后来看一下下面这个写法。这个写法把 when.....then.....的触发结构和 until 的循环结构分开来写了。

可以正常运行，加速度也可以控制在 2个G 附近。

```

SET thr TO 1.
SET dthr TO 0. //因为还没开始反馈，所以节流阀的反馈修正值先设到 0
LOCK THROTTLE TO thr. //节流阀挂钩到thr，初值1，之后要改thr的
LOCK STEERING TO UP. //朝向锁定到向上
Wait 1.
STAGE. //等1秒再点火发射
Wait 1.
//等起飞后1秒后再开始“没油就按空格”的触发结构
WHEN STAGE:LIQUIDFUEL < 0.1 THEN {
    STAGE.
}

WHEN SHIP:ALTITUDE > 100 THEN { //火箭高度超过100m后开始调节
    SET g TO KERBIN:MU / KERBIN:RADIUS^2. //地面重力加速度 9.81
    //加速度矢量和重力矢量的差
    LOCK accvec TO SHIP:SENSORS:ACC - SHIP:SENSORS:GRAV.
    LOCK gforce TO accvec:MAG / g. //计算加速度，Mag是指矢量的模
    //加速度设定一个目标值 2，和目标值的误差乘以一个系数，之后作为反馈
    LOCK dthr TO 0.05 * (2 - gforce).
}

UNTIL SHIP:ALTITUDE > 40000 { //火箭超过40000m之前都进行调节
    SET thr to thr + dthr. //将反馈修正值应用于节流阀(thr)
    //从300m高度就开始在屏幕上刷新加速度值gforce
    If SHIP:ALTITUDE > 300 {print gforce.}.
    WAIT 0.1. //每隔0.1秒调节一次
}

wait until false. //卡住程序，不让程序结束，要强行结束只能按 CTRL+C

```

下图结果我们可以看到，在飞行过程中，kOS根据情况减小了节流阀，控制火箭的加速度保持在2个G的样子。



*注1：本程序中的 `LOCK dthr TO 0.05 * (2 - gforce)`.，其实思路和 PID 控制相关。
有关 PID 控制的相关内容，请参考 实例6：PID 控制火箭反推悬浮。

(6) 优化方法：减少触发器数量，简化触发条件

触发结构是 kOS 的一大特色，目的是方便操控载具。
kOS 会在每个物理帧都轮询所有活动触发器的触发条件，
所以，减少活动触发器的数量，简化触发条件的表达式，是很重要的。

现在我们来改一下刚才的任务：
我们要用刚才的载具，垂直向上飞，但是要在不同高度把 G 里控制在不同程度。
100~10000米之间是加速度 2 个G，10000~20000米之间是加速度 2.5 个G，
20000~30000m之间是加速度 3 个G，30000~40000m之间是加速度 3.5 个G。
所以如果这些触发结构是并列书写的，就会这样：

```

SET thr TO 1.
SET dthr TO 0. //因为还没开始反馈，所以节流阀的反馈修正值先设到 0
LOCK THROTTLE TO thr. //节流阀挂钩到thr，初值1，之后要改thr的
LOCK STEERING TO UP. //朝向锁定到向上
Wait 1.
STAGE. //等1秒再点火发射
Wait 1.
//等起飞后1秒后再开始“没油就按空格”的触发结构
WHEN STAGE:LIQUIDFUEL < 0.1 THEN {
    STAGE.
}

WHEN SHIP:ALTITUDE > 100 THEN { //火箭高度超过100m后开始调节
    SET g TO KERBIN:MU / KERBIN:RADIUS^2. //地面重力加速度 9.81
    //加速度矢量和重力矢量的差
    LOCK accvec TO SHIP:SENSORS:ACC - SHIP:SENSORS:GRAV.
    LOCK gforce TO accvec:MAG / g. //计算加速度，Mag是指矢量的模
    //加速度设定一个目标值 2，和目标值的误差乘以一个系数，之后作为反馈
    LOCK dthr TO 0.05 * (2 - gforce).
}

WHEN SHIP:ALTITUDE > 1000 THEN {
    LOCK dthr TO 0.05 * (2.5 - gforce).
}
WHEN SHIP:ALTITUDE > 2000 THEN {
    LOCK dthr TO 0.05 * (3 - gforce).
}
WHEN SHIP:ALTITUDE > 3000 THEN {
    LOCK dthr TO 0.05 * (3.5 - gforce).
}

UNTIL SHIP:ALTITUDE > 4000 { //火箭超过4000m之前都进行调节
    SET thr to thr + dthr. //将反馈修正值应用于节流阀(thr)
    //从300m高度就开始在屏幕上刷新加速度值gforce
    If SHIP:ALTITUDE > 300 {print gforce.}.
    WAIT 0.1. //每隔0.1秒调节一次
}

wait until false. //卡住程序，不让程序结束，要强行结束只能按 CTRL+C

```

一共增加了三个 when.....then..... 触发结构，这东西虽然可行，不过很浪费计算资源。
因为时刻都有好几个触发结构在轮询触发条件，但他们不会同时动作。

把触发结构嵌套起来就可以解决这个问题，按照高度顺序依次激活内层的触发结构，这样就能减轻计算压力了。如下：

```

SET thr TO 1.
SET dthr TO 0. //因为还没开始反馈，所以节流阀的反馈修正值先设到 0
LOCK THROTTLE TO thr. //节流阀挂钩到thr，初值1，之后要改thr的
LOCK STEERING TO UP. //朝向锁定到向上
Wait 1.
STAGE. //等1秒再点火发射
Wait 1.
//等起飞后1秒后再开始“没油就按空格”的触发结构
WHEN STAGE:LIQUIDFUEL < 0.1 THEN {
    STAGE.
}

WHEN SHIP:ALTITUDE > 100 THEN { //火箭高度超过100m后开始调节
    SET g TO KERBIN:MU / KERBIN:RADIUS^2. //地面重力加速度 9.81
    //加速度矢量和重力矢量的差
    LOCK accvec TO SHIP:SENSORS:ACC - SHIP:SENSORS:GRAV.
    LOCK gforce TO accvec:MAG / g. //计算加速度，Mag是指矢量的模
    //加速度设定一个目标值 2，和目标值的误差乘以一个系数，之后作为反馈
    LOCK dthr TO 0.05 * (2 - gforce).
}

WHEN SHIP:ALTITUDE > 1000 THEN {
    LOCK dthr TO 0.05 * (2.5 - gforce).
    WHEN SHIP:ALTITUDE > 2000 THEN {
        LOCK dthr TO 0.05 * (3 - gforce).
        WHEN SHIP:ALTITUDE > 3000 THEN {
            LOCK dthr TO 0.05 * (3.5 - gforce).
        }
    }
}

UNTIL SHIP:ALTITUDE > 4000 { //火箭超过4000m之前都进行调节
    SET thr to thr + dthr. //将反馈修正值应用于节流阀(thr)
    //从300m高度就开始在屏幕上刷新加速度值gforce
    If SHIP:ALTITUDE > 300 {print gforce.}.
    WAIT 0.1. //每隔0.1秒调节一次
}

wait until false. //卡住程序，不让程序结束，要强行结束只能按 CTRL+C

```

4.3

4.3 CSV 数据导出

4.3.1

4.3.1 导读

===== 点击以返回目录 =====



本节提供了一种在游戏内监测并导出数据的方案。

建议玩家在初学 kOS 时选看。

本节内容：

实例5：用 Log 指令导出 csv 数据

- (1) Log 指令
- (2) CSV 数据文件
- (3) 用 kOS 记录飞行数据

4.3.2

4.3.2 实例5：用 Log 指令导出 CSV 数据

===== 点击以返回目录 =====



(1) Log 指令

Log 指令是 kOS 中重要的文件写入指令，用法简单，自由化程度高。

Log 指令用法如下：

```
Log [Data] to [FilePath].
```

*注1：[Data] 为自由表达式，可以是不同类型变量相加的形式。当相加的变量为不同类型时，变量自动转化为字符串进行拼接。表达式计算完成后会自动和一个换行符进行拼接，也就是说表达式计算完成后肯定是字符串形式的。

*注2：如果 [FilePath] 为未经声明的变量名，Log 指令会将 [Data] 写入变量名的同名文件，保存在 Volume 根目录下。如果 [FilePath] 为字符串，或字符串变量时，Log 指令会将 [Data] 写入以字符串命名的文件路径。

*注3：Log 指令工作时，会打开指定文件，如果没有此文件则新建一个。然后在文件末尾追加 [Data] 的字符串结果。注意 [Data] 的字符串结果的末尾会带一个换行符。

*注4：Log 指令工作时可能会产生大体积文件，建议将文件保存在 Volume 0 内。

Log 指令的例子：

```

Log "Hello_1" to mylog. //往 mylog 文件里写入字符串 "Hello_1"

Set fn to "mylog".
Log "Hello_2" to fn. //往 mylog 文件里写入字符串 "Hello_2"

Log 4+1 to mylog.txt. //往 mylog.txt 文件里写入字符串 "5"

Set a to V(1,2,3).
Log a to "mylog.txt" . //往 mylog.txt 文件里写入字符串 "V(1,2,3)"

//往Volume 0 下的 abc 文件下的 def 文件下的 mylog.f90 文件里写入
//写入字符串 "4 times 8 is: 32"
//表达式里用加号 + 来连接字符串, 表示字符串的合并连接
Set ch to "4 +"times".
LOG ch+"8 is: " + (4*8) to "0:/abc/def/mylog.f90".

```

(2) CSV 数据文件

CSV 是一种常用的逗号分隔的表格文件格式。

CSV 文件是一种文本文件, 内容是明文的。

CSV 文件可以用 Excel 打开和编辑, 也可以用记事本打开。

例子: 同一个文件分别用 Excel 和记事本打开, 如下图。

The screenshot shows a Microsoft Excel window with the title bar 'sample.csv - Microsoft Excel'. The ribbon menu is visible with tabs like '开始', '插入', '页面布局', '公式', '数据', '审阅', '视图', '负载测试', and '团队'. The '开始' tab is selected. The toolbar below includes icons for '粘贴', '剪贴板', '字体', '对齐方式', '数字', '条件格式', '套用表格格式', '单元格', and '编辑'. The formula bar shows 'F23'. The main worksheet area displays the following data:

	A	B	C	D	E	F	G	H
1	月份	全国CPI	城市CPI	农村CPI				
2	2015年09月	101.6	101.6	101.5				
3	2015年08月	102	102	101.8				
4	2015年07月	101.6	101.7	101.5				
5	2015年06月	101.4	101.4	101.2				
6	2015年05月	101.2	101.3	101				
7	2015年04月	101.5	101.6	101.3				
8	2015年03月	101.4	101.4	101.2				
9	2015年02月	101.4	101.5	101.2				
10	2015年01月	100.8	100.8	100.6				

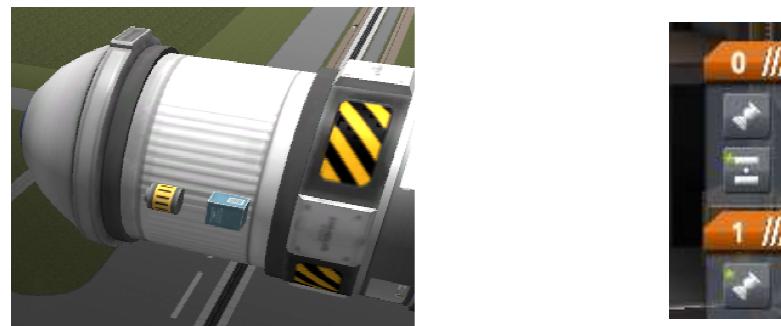
月份,全国CPI,城市CPI,农村CPI
2015年09月份,101.6,101.6,101.5
2015年08月份,102,102,101.8
2015年07月份,101.6,101.7,101.5
2015年06月份,101.4,101.4,101.2
2015年05月份,101.2,101.3,101
2015年04月份,101.5,101.6,101.3
2015年03月份,101.4,101.4,101.2
2015年02月份,101.4,101.5,101.2
2015年01月份,100.8,100.8,100.6

*注1: CSV 文件有类似 Excel 里单元格的概念, 如果单元格原文里本身就有逗号字符, CSV 则会自动加入一对双引号来区分哪个是正文, 哪个是分隔符。

"2015,年08月份", 102, 102, 101.8

(3) 用 kOS 记录飞行数据

接下来用 Log 指令做点实际的事情吧, 我们用 实例4: kOS 程序的优化 里同样的火箭。



我们要让火箭竖直向上飞的同时, 从100m高度开始, 每隔2秒记录一次飞行数据, 要记录的数据有: 时间、海拔高度、地面速度、加速度, 直到第一级火箭烧完为止。这些数据都要保存在 Volume 0 下的 CSV 文件里。

相关程序如下:

```
//运行此程序前先要 switch to 0. 来切到 Volume 0,  
  
Lock Steering to UP. //火箭稳定向上  
Lock Throttle to 1. //节流阀拉到最大  
Wait 3. //等待5秒钟  
Stage. //发射火箭  
  
//写入标题行，其中加速度为矢量，有 XYZ 三个方向的分量  
log "Time,Height,Speed,AccX,AccY,AccZ" to Data.csv.  
wait until Ship:Altitude>100. //先等到高度超过海拔 100 米，再开始后续的记录  
  
//直到燃料烧完推力为 0 开始，每隔 2 秒记录一次飞行数据。  
until Ship:MaxThrust = 0 {  
  
    //记录数据行到 Data.csv 文件，  
    //每行信息依次为：时间、海拔高度、地面速度、加速度  
    log Time:Seconds +", "+Ship:Altitude +", "+Ship:AirSpeed +", "  
        +Ship:Sensors:Acc to Data.csv.  
  
    wait 2. //等待 2 秒钟  
}
```

好了，记录到的数据用 Excel 打开如下。

这里有个小瑕疵就是，kOS 记录加速度时的字符串是 V(AccX,AccY,AccZ) 形式的矢量，玩家需要在 Excel 里自行删除替换多余的符号。

	A	B	C	D	E	F
1	Time	Height	Speed	AccX	AccY	AccZ
2	3467.54	100.2936779	20.94738192	V(-8.70675310439557)	-0.032621289	7.33887839558687)
3	3469.56	166.366675	44.3495268	V(-8.95771182574006)	-0.016811503	7.54940609463555)
4	3471.58	280.3637632	68.3818005	V(-9.18754978520072)	-0.019124912	7.7456294071577)
5	3473.6	443.5087635	92.99896473	V(-9.39638916914962)	-0.020102703	7.92368329521855)
6	3475.62	656.9179683	118.1446146	V(-9.60449963973256)	-0.021160325	8.09920209366351)
7	3477.64	921.7207214	143.8883478	V(-9.88825348920426)	-0.02229265	8.30167050301563)
8	3479.66	1239.213705	170.3234607	V(-10.1113009251591)	-0.023624626	8.53352505488437)
9	3481.68	1610.879318	197.5306822	V(-10.4235498824051)	-0.024912208	8.78949751741206)
10	3483.7	2038.326464	225.546149	V(-10.7262628806847)	-0.025979823	9.04524250185199)
11	3485.72	2523.183581	254.3666399	V(-11.0358468176204)	-0.026766194	9.30731189716249)
12	3487.74	3067.133741	284.0667695	V(-11.3876705215847)	-0.027334253	9.59920224397116)
13	3489.76	3671.95823	314.4783916	V(-11.4739951733886)	-0.027204125	9.67103349362355)
14	3491.78	4338.186473	345.074912	V(-11.8381349279715)	-0.027849957	9.96595628512017)
15	3493.8	5068.141865	377.6372845	V(-12.5816156159442)	-0.029071556	10.5957952243847)
16	3495.82	5865.707076	412.0757274	V(-13.3902405972085)	-0.030561924	11.2869960562929)
17	3497.84	6735.394055	449.1325646	V(-14.4704003772972)	-0.032786422	12.2043384267234)
18	3499.86	7683.05944	489.3430672	V(-15.7457470633429)	-0.035370014	13.2901158435379)
19	3501.88	8715.6584	533.2651747	V(-17.2023615265925)	-0.038450074	14.5313609410039)
20	3503.9	9840.88943	580.9653963	V(-18.5620484128495)	-0.041396644	15.6878322893997)
21	3505.92	11065.82202	631.8370311	V(-19.6638767099248)	-0.043954257	16.6242461203492)
22	3507.94	12396.47594	685.6562593	V(-20.8148931418043)	-0.046515811	17.6127177302693)
23	3509.96	13839.14473	742.787248	V(-22.1206667434652)	-0.049396584	18.7419695067001)
24	3511.98	15400.94031	803.6205264	V(-23.542190891638)	-0.052414978	19.9755052489832)
25	3514	17089.53616	868.3198542	V(-25.0038543188112)	-0.055621326	21.2459421798374)
26	3516.02	18912.84787	936.9827114	V(-26.4915817880165)	-0.058911733	22.5408315425493)
27	3518.04	20878.92006	1009.640709	V(-27.9852267119838)	-0.062106507	23.8424289122788)
28	3520.06	22995.84954	1086.32939	V(-29.4994325239733)	-0.065462862	25.1622590966365)
29	3522.08	25271.84285	1167.113011	V(-31.0359879877204)	-0.068945837	26.5012382923525)
30						

4.4

4.4 PID 控制教程

4.4.1

4.4.1 导读

本节介绍了 PID 控制的原理，并介绍了如何使用 kOS 中的 PID 结构体进行 PID 控制。

建议玩家在初学 kOS 时爱看不看。

本节内容：

PID 控制与 Ziegler-Nichols 方法

- (1) 引子
- (2) PID 控制器
- (3) 用 Ziegler-Nichols 方法来整定 PID

实例6：PID 控制火箭反推悬浮

- (1) 任务目标与条件
- (2) 使用 PID 结构体

- (3) 反推悬浮 with 普通 PID
- (4) 反推悬浮 with 双向调节的 PID
- (5) 反推悬浮 with 主动刹车 + 考虑平衡推力的 PID

4.4.2

4.4.2 PID 控制与 Ziegler-Nichols 方法



(1) 引子

在先前的实例4: kOS 程序的优化中，我们写过一个控制火箭的加速度在2G左右的程序。

当时我们把当前加速度 gforce 和目标加速度2之间的误差值乘以一个比例系数 0.05。

将 $0.05 * (2 - \text{gforce})$ 作为修正值施加在原来的节流阀杆量 thr 上。

最终实现了将火箭加速度稳定在 2G 的效果。

其实这里已经有我们接下来要介绍的 PID 控制的影子了。

(2) PID 控制器

PID 控制 是一种典型的反馈控制，广泛用于工业自动控制。

PID 三个字母分别指比例（Proportional）-积分（Integral）-微分（Derivative），
PID 控制就是不断对反馈的误差信号进行比例、积分与微分运算，将计算结果加入控制输出量的控制方式。

PID 控制的公式有很多种，下面是最典型的两种：

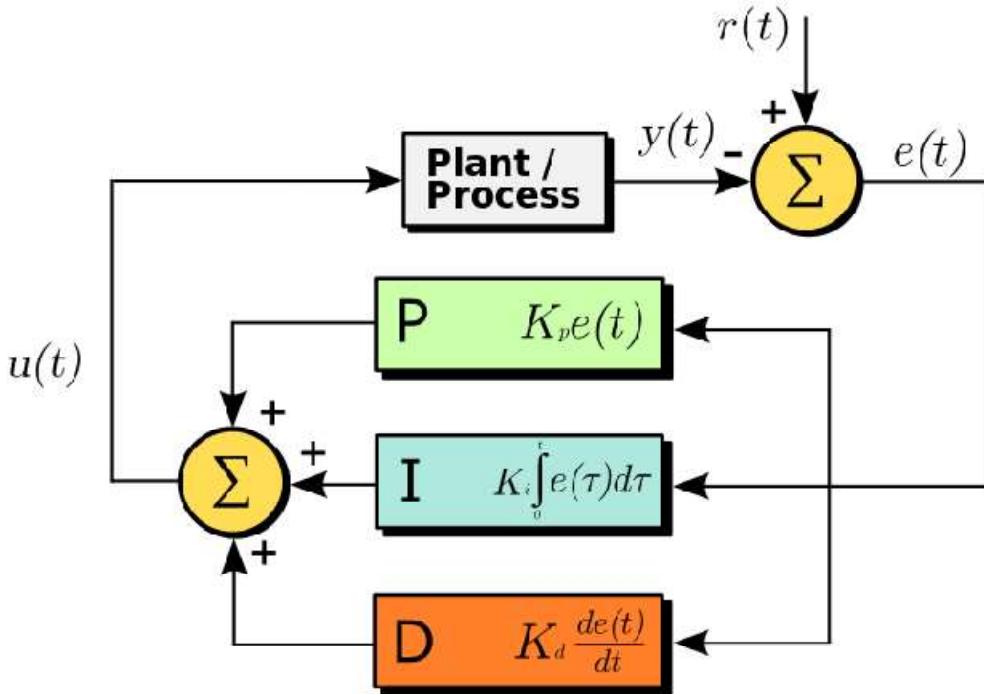
$$\begin{aligned} u[t] &= K_p * e[t] + K_i * \int_0^t e[t] dt + K_d * \frac{de[t]}{dt} \\ u[t] &= K_p * \left(e[t] + \frac{1}{Ti} * \int_0^t e[t] dt + Td * \frac{de[t]}{dt} \right) \end{aligned}$$

*注1：上图中 $u[t]$ 是 PID 控制的输出值， $e[t]$ 是目标状态和当前状态的差距；
第一项是比例项 P 项，是这个差距，乘一个系数 K_p ；
第二项是积分项 I 项，是这个差距的定积分，乘一个系数。
第三项是微分项 D 项，是这个差距的变化速率，乘一个系数。

*注2：两种公式的区别只是系数的写法不同。
其中 Ti 和 Td 又叫积分时间和微分时间，
注意，虽然这么叫，但是他们和算定积分时的区间，算导数时的 dt ，并没有关系。
积分区间是从 0 时间到当前时间，而 dt 是由计算时的刷新频率决定的；

*注3：PID 控制中三项各有用途，分别在不同场合对输出量 $u(t)$ 起主要作用。
比例项 P 在距离目标值较远时占主要贡献，作用是快速接近目标值。
积分项 I 在长时间处于目标值附近时占主要贡献，作用是稳定至目标值。
微分项 D 在稳定在目标值时占主要贡献，作用是抵抗外部微扰，起到阻尼作用。

下图为第一种典型公式的流程图，



图片来源：维基百科 PID controller 词条

PID 控制的功能非常强大，用途非常广泛。使用 PID 控制时，甚至不需要知道控制量和状态量之间的物理规律，只需要调节合适的 PID 参数就可以了。

(3) 用 Ziegler-Nichols 方法来整定 PID

使用 PID 温控器时要控制的好，就要更小的过冲和更快的稳定时间。

关键就是找到合适的 PID 参数。寻找合适 PID 参数的过冲叫 **调 PID 或整定**。

有一种 **Ziegler-Nichols 方法**，可以搞定 PID 整定这件事。

(Ziegler 和 Nichols 是两个人的名字，他们发表了这个方法)

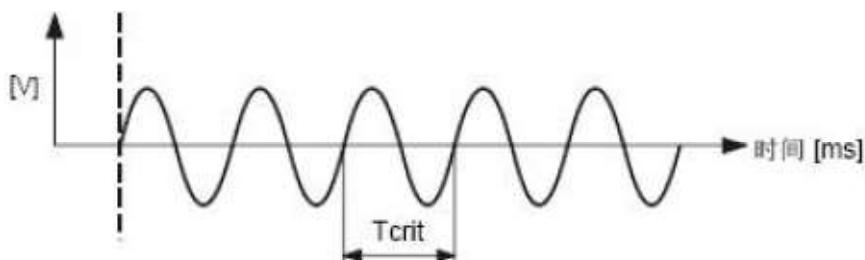
Step1. 只使用比例项 P，此时 PID 控制的状态变化是一条震荡曲线。

Step2. 调节比例项 P 的参数 K_p ，使状态变化曲线达到稳定震荡，如下图。

将此时的 K_p 参数记作 K_u ，将此时的震荡周期记作 T_u 。

Step3. 参照下表公式计算新的 PID 参数。

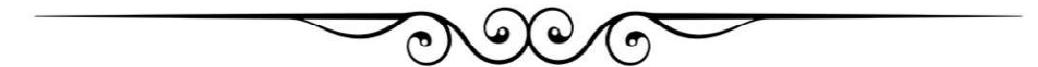
Step4. 根据测试情况微调 PID 参数。微调时尽可能只动 K_p 。



参数类型	K_p	K_i	K_d
参数值	$0.6 * K_u$	$2 * K_u / T_u$	$K_u * T_u / 8$
参数类型	K_p	T_i	T_d
参数值	$0.6 * K_u$	$T_u / 2$	$T_u / 8$

4.4.3

4.4.3 实例6：PID 控制火箭反推悬浮



(1) 任务目标与条件

任务目标：从发射台起飞竖飞，要稳定在500米高空。

载具条件：使用如下图的大推重比火箭，该火箭带有 kOS 部件以及多种传感器。

火箭顶部引擎 Tag 名叫 Up，火箭底部引擎 Tag 名叫 Down，都关闭了矢量。

火箭的 Stage 发射序列为，按一次空格启动底部引擎，按第二次启动顶部引擎。



(2) 使用 PID 结构体

kOS 提供有 PID 结构体，他把繁琐的 PID 控制算法打包成了一个黑箱，只留了几个关键参数作为外部接口。玩家只需要简单三步，就能方便的解决实际的 PID 控制问题：

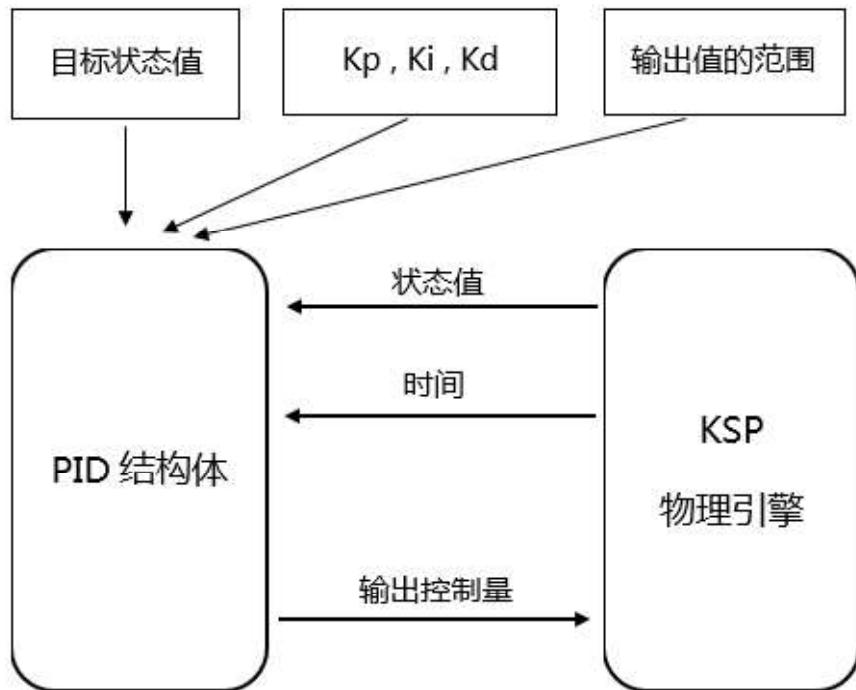
Step1. 实例化 PID 结构体，设定 PID 控制的参数：

(K_p, K_i, K_d) 、PID 输出值的范围、PID 的目标状态值；

Step2. 不断调用结构体内的 Update 函数进行 PID 状态刷新计算，

调用函数时输入时间、状态量的最新值来作为参数；

Step3. 将 Update 函数的返回值应用到输出控制量；



写成代码的话就是如下用法：

```

Set KP to .....
Set KI to .....
Set KD to ..... //设置PID参数
Set TargetV to ..... //目标值, 例如火箭悬停的话就是悬停的目标高度
//创建一个PIDLoop结构, 输入PID参数, 并约束输出值的范围
//例如火箭悬停的话就是节流阀百分比
SET PID TO PIDLOOP(KP, KI, KD, MINOUTPUT, MAXOUTPUT).
set PID:setpoint to TargetV //为这个PIDLoop设置目标值TargetV
until false {
    //每一物理帧都更新, 并提供当前时间和要控制的变量Status, 如火箭高度
    //PID会输出OUT, 例如火箭悬停的话就是节流阀百分比 ×
    SET OUT TO PID:UPDATE(TIME:SECONDS, Status).
}
  
```

(3) 反推悬浮 with 普通 PID

现在要求只使用火箭底部的引擎，要做实现反推悬浮，最容易想到的办法就是

$$\text{节流阀 (0, 1)} = \text{PID输出 (0, 1)}$$

其中(0,1)指的是这个值的范围。

根据这个思路，我们写出了下面的控制程序。

注意有红色的那行 Lock 语句，就是这个思路的体现。

```

//运行程序前先要 switch to 0.

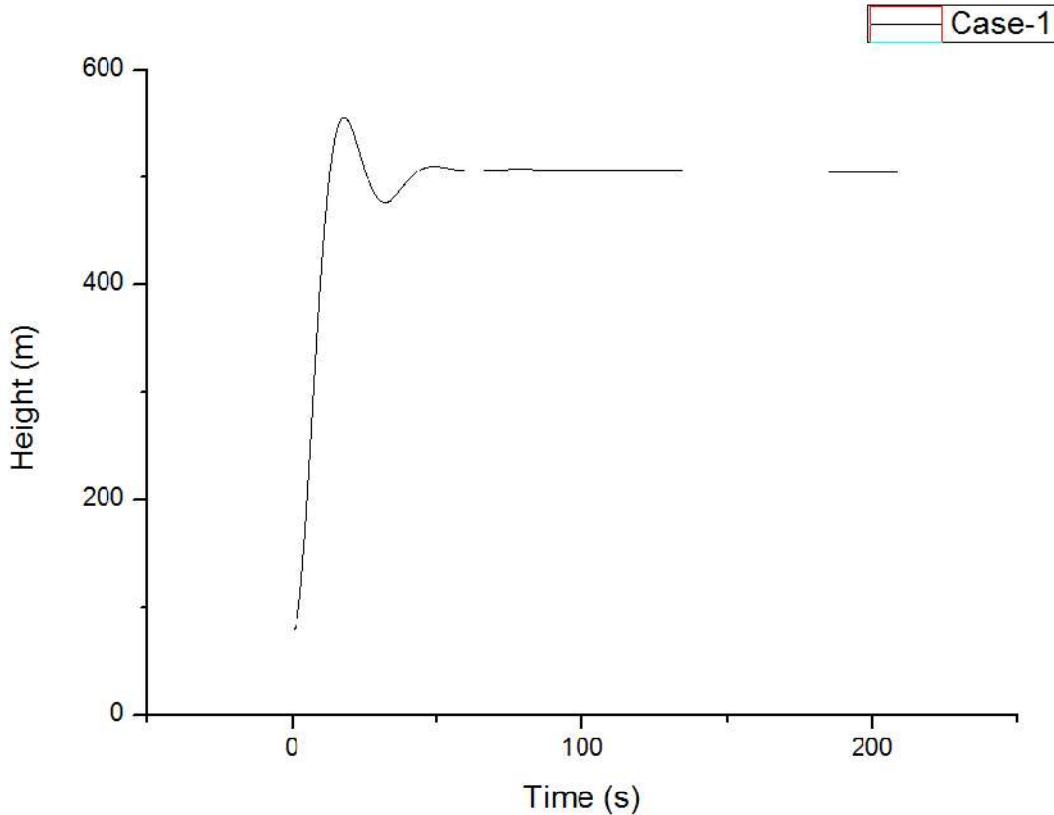
//设置PID参数 (按照经典PID计算)
Set KP to 0.6*0.0021. //0.6*Ku
Set KI to 2*KP/33. //2*Kp/Tu
Set KD to KP*33/8. //Kp*Tu/8
Set TargetH to 500. //设置目标悬浮高度
//=====
Lock Steering to UP. //火箭姿态稳定朝上。
//创建PID结构体，其中前3个是Kp,Ki,Kd参数，后2个是最小和最大输出
Set pid to PIDLoop(KP,KI,KD,0,1).
Set pid:SetPoint to TargetH. //设定PID的目标高度
Log "Time,Altitude" to "pidlog.csv". //记录标题行到数据文件pidlog.csv
Set value to 0. //赋节流阀初值
//=====
Lock Throttle to value. //value就是控制量
Wait 0.5. //给火箭点反应时间
Stage. //按下空格一次，点火的是火箭底部引擎。
Set StartTime to Time:Seconds. //纪录任务开始时间
Wait 0.5. //给火箭点反应时间

//直到火箭燃料用完之前，都始终用PID算法控制载具高度
Until Ship:MaxThrust = 0 {
    Set value to pid:Update(Time:Seconds,Ship:Altitude). //更新PID循环
    //记录数据曲线到文件，并同步在命令窗口里显示×
    Set MT to Time:Seconds-StartTime. //计算任务时间×
    Log MT+","+Ship:Altitude to "pidlog.csv".
    Print MT+","+Ship:Altitude.
    Wait 0.001. //等待下一个物理帧
}

//要结束程序的话请按 Ctrl + C

```

然后我们跑一下这个程序，根据输出的文件数据，绘制出如下曲线。



看上去效果已经有一点了，但是过冲很大，稳定的也慢。

而且 PID 没跟上火箭燃料消耗的速度，所以稳定位置约 506m，和目标值 500m 还有偏差。

(4) 反推悬浮 with 双向调节的 PID

这次我们同时使用火箭顶部和底部的引擎。要同时控制两个引擎输出不同的推力，我们不能用节流阀，我们要用引擎部件自身带的 ThrustLimit 字段。如下：

双引擎的推力限制 (-1, 1) = PID输出 (-1, 1)

其中(-1,1)指的是这个值的范围，

正值表示只有底部引擎工作，负值表示只有顶部引擎工作。

这种双向调节的方式理应比只有底部引擎的单项调节方式，表现更好。

根据这个思路，我们写出了下面的控制程序。

注意粗体字，就是我们修改的地方。注意有红色的那几行语句，就是这个思路的体现。

```

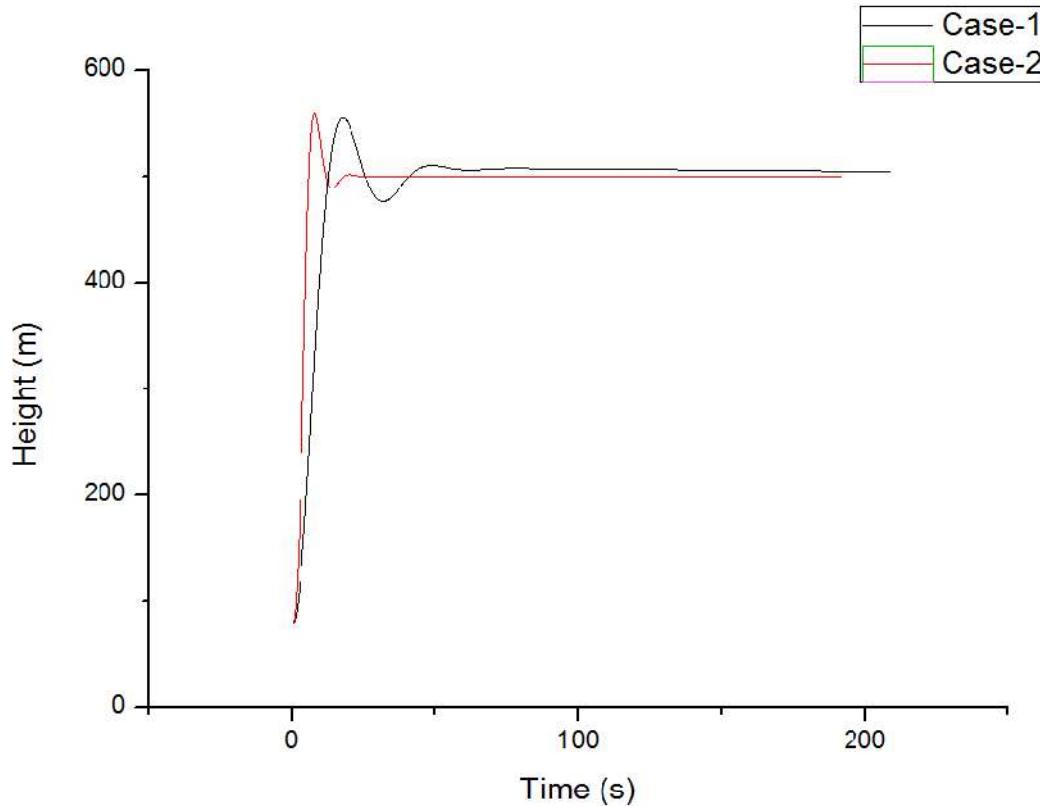
//运行程序前先要 switch to 0.

//设置PID参数
Set KP to 0.0036*2.5.
Set KI to 0.000435*2.5.
Set KD to 0.00744*2.
Set TargetH to 500. //设置目标悬浮高度
//=====
//分别找到上下两个引擎
//Ship:PartsTagged("Up")代表当前火箭上 标签标记为 Up 的所有部件的列表
//后面的[0]代表一个列表的第一个成员。
Set eu to Ship:PartsTagged("Up") [0].
Set ed to Ship:PartsTagged("Down") [0].
Lock Steering to UP. //火箭姿态稳定朝上。
//创建PID结构体，其中前3个是Kp,Ki,Kd参数，后2个是最小和最大输出
Set pid to PIDLoop(KP,KI,KD,-1,1).
Set pid:SetPoint to TargetH. //设定PID的目标高度
Log "Time,Altitude" to "pidlog.csv". //记录标题行到数据文件pidlog.csv
Lock Throttle to 1. //这次不调节节流阀，打算调节引擎的推力限制。
Set value to 0. //赋初值，0~1表示底部引擎工作，-1~0表示顶部引擎工作
Set eu:ThrustLimit to 0. //赋初值
Set ed:ThrustLimit to 0. //赋初值
//=====
Wait 0.5. //给火箭点反应时间
Stage. //按下空格一次，点火的是火箭底部引擎。
Wait 2.0. //给火箭点反应时间
Stage. //按下空格一次，点火的是火箭顶部引擎。
Set StartTime to Time:Seconds. //纪录任务开始时间
Wait 0.5. //给火箭点反应时间

//直到火箭燃料用完之前，都始终用PID算法控制载具高度
Until Ship:MaxThrust = 0 {
    Set value to pid:Update(Time:Seconds,Ship:Altitude). //更新PID循环
    //将 value 映射到 eu 和 ed 的推力限制
    //这个字段的有限范围是 0~100 (%), 超过范围会取就近值
    set eu:ThrustLimit to -value*100.
    set ed:ThrustLimit to value*100.
    //记录数据曲线到文件，并同步在命令窗口里显示
    Set MT to Time:Seconds-StartTime. //计算任务时间
    Log MT+","+Ship:Altitude to "pidlog.csv".
    Print MT+","+Ship:Altitude.
    Wait 0.001. //等待下一个物理帧
}
//要结束程序的话请按 Ctrl + C

```

我们跑一下这个程序，根据输出的文件数据，绘制出如下曲线，
其中 Case-2 红线就是这次的结果。。



看上去效果比之前好很多了。过冲差不多，但稳定的很快，稳定位置和目标值500m也很接近。

因为有两个引擎，可以正反调节，所以推力也很足，上升段更快，燃料也更早用完。

(5) 反推悬浮 with 主动刹车 + 考虑平衡推力的 PID

上一个假设看上去很好，但实际上基本没有火箭会装向上的引擎来反推。

更一般情况下，我们只用底部引擎，并且用其他方法来改善 PID 控制的效果。

方法一：在PID的输出值基础上再加上一项实时计算的平衡值。

平衡值是引擎推力等于火箭重力时的节流阀值。

$$\text{节流阀 } (\theta, 1) = \text{PID输出 } (\theta, 1) + \text{平衡值}$$

方法二：在火箭一开始快速上升的阶段增加一个“精准刹车”的逻辑：

如果火箭上升时高度低于 $500 - 5 = 495\text{m}$ ，但是抛物线顶点超过了这个值，那么这个状态下的火箭发动机停机，并且 I 部分的积分值重置为 0，直到火箭脱离此状态。

这就相当于让火箭在平衡位置释放，稳定起来肯定最快。

根据这个思路，我们写出了下面的控制程序。

注意有红色的那几行，就是这个思路的体现。

```

//运行程序前先要 switch to 0.

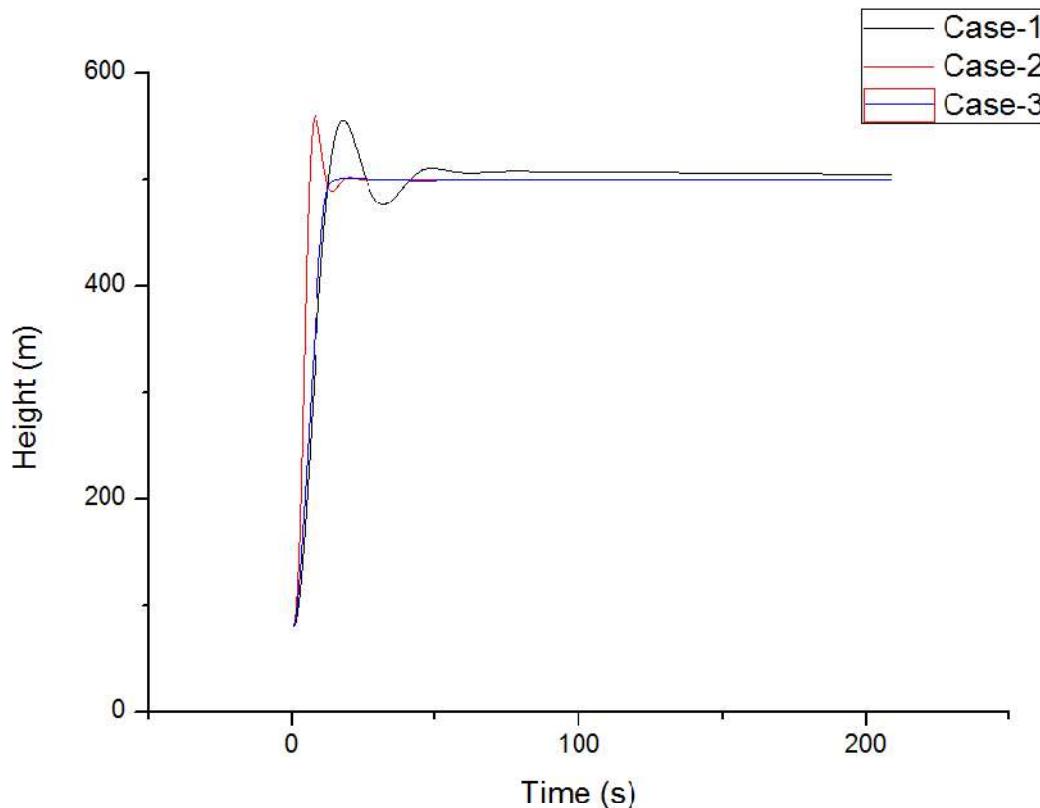
//设置PID参数
Set KP to 0.6*0.001*2.5. //0.6*Ku
Set KI to 2*KP/35*2.5. //2*Kp/Tu
Set KD to KP*35/8*2.5. //Kp*Tu/8
Set TargetH to 500. Set WarnErr to 5. //设置目标悬浮高度和警戒距离
//=====
Lock Steering to UP. //火箭姿态稳定朝上。
Log "Time,Altitude" to "pidlog.csv". //记录标题行到数据文件pidlog.csv
Set value to 0. //赋节流阀初值
//=====
Lock Throttle to 0. //节流阀先拉到0
Wait 0.5. //给火箭点反应时间
Stage. //按下空格一次，点火的是火箭底部引擎。
Wait 0.5. //给火箭点反应时间
Lock BT to Ship:Mass * Ship:Sensors:Grav:Mag / Ship:MaxThrust.
//创建PID结构体，其中前3个是Kp,Ki,Kd参数，后2个是最小和最大输出
Set pid to PIDLoop(KP,KI,KD,-BT,1-BT).
Set pid:SetPoint to TargetH. //设定PID的目标高度
//BT 是实时更新的平衡位置节流阀
Set Thre to BT + value.
Lock Throttle to Thre. //节流阀挂钩到Thre值,
Set StartTime to Time:Seconds. //纪录任务开始时间

//直到火箭燃料用完之前，都始终用PID算法控制载具高度
Until Ship:MaxThrust = 0 {
    //是否同时满足三个警戒条件(IsWarn)
    Set IsRise to (Ship:VerticalSpeed>0). //火箭是否在上升
    Set IsFar to (Ship:Altitude<TargetH-WarnErr). //火箭是否离得比较远
    Set TopH to Ship:VerticalSpeed*Ship:VerticalSpeed/2/9.81+Ship:Altitude.
    //是否火箭未来最大高度大约在500m
    Set IsSuit to (TopH>TargetH-WarnErr).
    Set IsWarn to (IsRise and IsFar and IsSuit).
    If IsWarn {
        Set Thre to 0. //首先是关闭节流阀×
        pid:Update(Time:Seconds,Ship:Altitude). //刷新PID，但悬空输出×
        pid:Reset(). //并且清空pid的积分数值
    } Else {×
        Set value to pid:Update(Time:Seconds,Ship:Altitude). //更新PID循环×
        Set Thre to BT + value. //应用节流阀=平衡推力+PID输出
    }
    //记录数据曲线到文件，并同步在命令窗口里显示
    Set MT to Time:Seconds-StartTime. //计算任务时间
    Log MT+","+Ship:Altitude to "pidlog.csv".
    Print MT+","+Ship:Altitude.
    Wait 0.001. //等待下一个物理帧
}

//要结束程序的话请按 Ctrl + C

```

我们跑一下这个程序，根据输出的文件数据，绘制出如下曲线，
其中 Case-3 蓝线就是这次的结果。可以看到结果非常好。



4.5

4.5 变轨教程

4.5.1

4.5.1 导读

===== 点击以返回目录 =====



本节介绍了在 kOS 中进行轨道机动的方法。

建议玩家在初学 kOS 时必看。

本节内容：

实例7：控制火箭圆轨

4.5.2

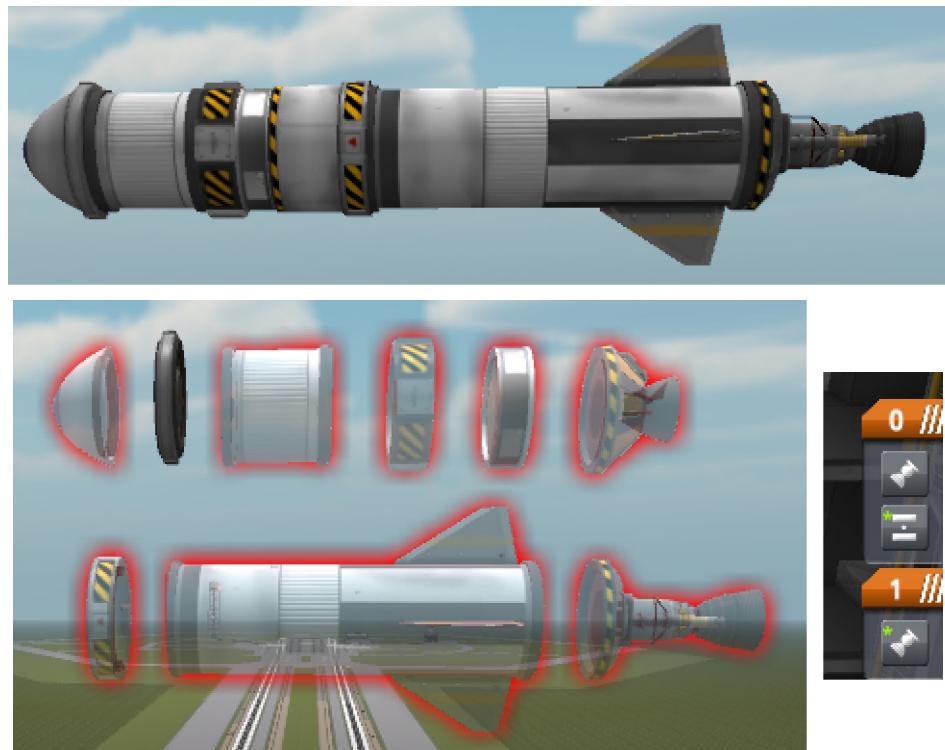
4.5.2 实例7：控制火箭圆轨

===== 点击以返回目录 =====



之前在 **实例2：控制火箭发射** 中，我们已经成功将火箭发射至 AP 点出大气的亚轨道。现在我们要准备接下去的入轨步骤。

我们用的火箭和之前一样，如下。



我们在之前程序基础上，加入出大气层后的变轨规划和执行变轨规划的内容。
操作步骤如下，其中后三条粗体字是我们这次新加的。

- Step1.** 垂直发射，始终满油门；
- Step2.** 到达一定高度的时候转到对应的方向；
- Step3.** 远点超过目标高度后节流阀关掉；
- Step4.** 进入太空中设计一个在远点的变轨到圆轨道的变轨规划，设计喷射时间长度为**dt**；
- Step5.** 调整好姿态不变，在到达远点之前**dt / 2**就开始喷射，喷射持续到近点到达**80km**；
- Step6.** 完成入轨。

按照上述思路写的程序如下，其中粗体字是这次修改的部分。

```

Set AimOrbitHeight to 100000. //入轨程序的目标轨道的高度

//声明一个自定义函数 head
function head {
    set v to Ship:AirSpeed. //设 v 为火箭的地面速度
    set p to 90-sqrt(v)*2.0. //火箭的俯仰角 p 和 v 相关，这是本人随便乱写的关系式
    //Heading函数的第一个参数是指从正北方向顺时针转90°的水平方向（正东方）
    //第二个参数是指俯仰角，90就是垂直，0就是水平
    return heading (90,p). //返回火箭的目标朝向，其中heading
}

//火箭朝向设置为追随 head 函数返回的朝向值
LOCK STEERING TO head().
//节流阀拉到最大，节流阀范围 0.0 是最小，1.0 是最大。
LOCK THROTTLE TO 1.0.

//首先，我们先清屏
CLEARSCREEN.
//以下的From结构是个循环，类似C语言里的for循环，做的是倒计时 10 ~ 1
PRINT "Counting down:".

```

```

FROM {local countdown is 10.} UNTIL countdown = 0 STEP {SET countdown to countdown - 1.} DO {
    PRINT "..." + countdown.
    WAIT 1. // 等待1秒钟.
}

//如果火箭的最大推力=0, 那燃料肯定烧光了, 那么就按空格,
WHEN MAXTHRUST = 0 THEN {
    PRINT "Staging".
    STAGE.
    //RETURN 返回的值代表是否要保留这个触发结构
    //true 的话就代表这个触发结构会继续工作
    //false 的话这个触发结构就寿命到了, 消失了×
    RETURN true.
}.

//等到火箭的轨道的AP点到达 100000 m 高度时
WAIT UNTIL SHIP:OBT:APOAPSIS > 100000.
Lock Throttle to 0. //关闭节流阀
Lock Steering to ship:PROGRADE. //火箭朝向保持速度方向
wait until Ship:Altitude>70001. //等进入太空之后再开始后面的变轨计算
Lock Throttle to 0.1. //慢慢推节流阀
wait until Ship:Apoapsis>AimOrbitHeight. //等远点重新到目标高度后
Lock Throttle to 0. //关掉节流阀
wait 1.//等1秒

//下面是设计变轨规划
Set EtaTime to ETA:Apoapsis. //要变轨的是远点, 采集到远点的时间
Set V1 to
(vcrs(Ship:Body:Position,Ship:Velocity:Orbit):Mag) / (Kerbin:Radius+Ship:Apoapsis).
//用角动量守恒来算远点速度
Set V2 to Sqrt (Constant():G*Kerbin:Mass / (Kerbin:Radius+Ship:Apoapsis)).
//用天体公式GM=V2R算出远点圆轨道的速度
Set MyNode to Node (Time:Seconds+EtaTime,0,0,V2-V1).
//生成一个变轨规划, 四个参数分别为: 几秒后、径向法向切向DV
add MyNode. //将变轨计划加入飞行计划

//下面是执行变轨规划
Set Nd to MyNode. //读取下一个变轨计划的信息
Set MaxAcc to Ship:MaxThrust/Ship:Mass.
//计算加速度, 这里忽略了质量会随着燃料燃烧而减少这件事
Set BurnDuration to Nd:Deltav:Mag/MaxAcc. //计算变轨需要的持续工作时间
wait until Nd:ETA<=(BurnDuration/2 + 40).
//等到工作时间的一半+40秒再执行后面的, 40秒是为了留时间调姿
Set Np to Nd:Deltav.
Lock Steering to Np.
//只读取刚计算好值, 因为机动过程中Nd的数据会变, 而只读取初值的话误差能接受
wait until Nd:ETA<=(BurnDuration/2).
//等到工作时间的一半, 这时候姿态对准喷射方向了, 就等点火了
Set Done to FALSE. //已完成变轨的标记, 用于跳出循环
Lock throttle to 1.//开始加速
wait until ship:PERIAPSIS>10000.//等到近点露出地面
until Done {//循环开始×
    set halfT to 0.5*ship:orbit:period. //轨道周期的一半×
    set halfTV to VelocityAt(ship, Time:seconds+halfT).
}

```

```

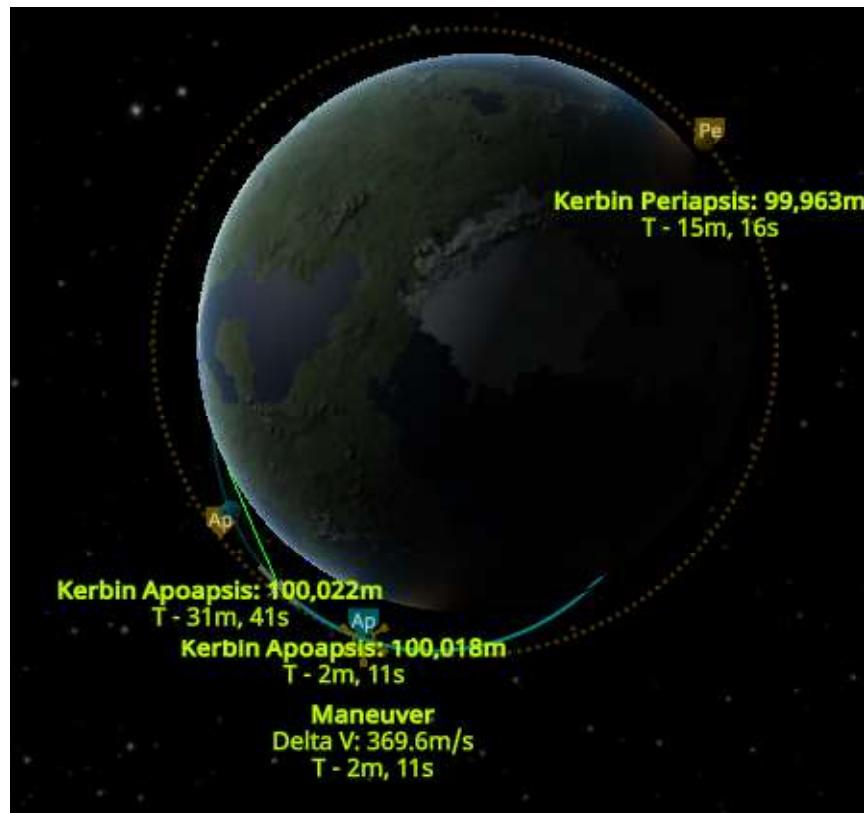
set halfTVV to halfTVV:orbit:Mag.
//一半周期后的速度，也就是轨道对过的速度
//圆轨开始时，自己是速度较小的远点，对过是速度较大的近点
//圆轨过程中，远点和近点差距缩小，直到圆轨道就相等了×
set delta to halfTVV-ship:Velocity:orbit:Mag. //飞船和对过的速度差×
set safe to 0.3*ship:Maxthrust/ship:mass. //速度差不大×
if delta < safe { lock throttle to 0.15. } //如果速度差不大就慢慢加速×
if delta < 0 {
    lock throttle to 0. ×
    Set Done to TRUE.}//如果速度大小倒错就加好了×
    wait 0.001.}

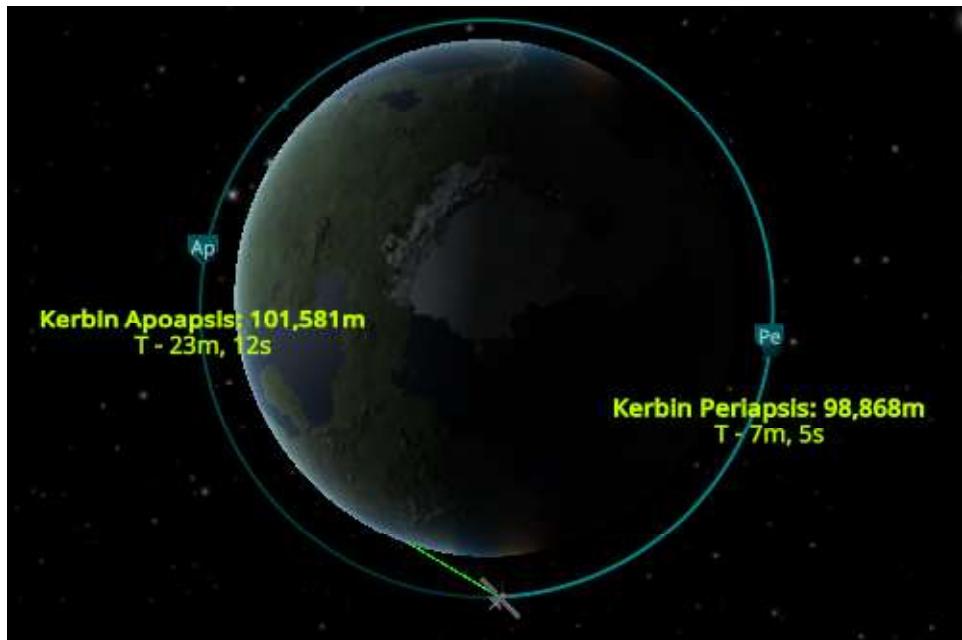
wait 1.

unLock Steering.
unLock Throttle. //解绑姿态和节流阀锁定
remove Nd. //移除变轨计划
//为了以防万一，将节流阀设为0，这可以避免放开节流阀后节流阀异常
Set Ship:Control:PilotMainThrottle to 0.
SAS ON. //把SAS打开，这样kOS程序结束之后火箭不会乱转。

```

运行结果如下，火箭成功入轨，精度还可以。





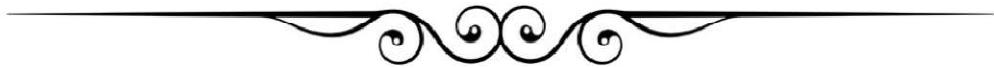
4.6

4.6 边界测试

4.6.1

4.6.1 导读

----- 点击以返回目录 -----



本节介绍了一个用于显示载具或部件的kOS工具程序。

本节使用大量还未介绍的特性和技巧，建议玩家在初学 kOS 时爱看不看。

本节内容：

实例8：边界监测程序

4.6.2

4.6.2 实例8：边界监测程序

===== 点击以返回目录 =====



```
@lazyglobal off.

//
//  display_bounds.ks
//  -----
//
// A small example program that will show you how to
// read the BOUNDS information of Vessels and Parts,
// and will display the box as a set of Vecdraws.
// You may iterate over the parts with the N and P
// keys. (The "-1 th" part is to show the whole
// vessel's box).
//

// Because this example uses lots of delegates
// in its VECDRAWs, it needs the time to keep running
// those delegates each update, or else it really
// bogs down:
if CONFIG:IPU < 500 {
    set CONFIG:IPU to 500.×
    HUDTEXT("NOTICE: EXAMPLE SCRIPT INCREASED CONFIG:IPU TO " + CONFIG:IPU,
        20, 2, 22, magenta, true).
}

// -----
// These are some utility functions for this
// example program to help display things:
// -----


function vector_tostring_rounded {
    // Same thing that vector:tostring() normally
    // does, but with more rounding so the display
    // doesn't get so big:
    parameter vec.

    return "V(" +
        round(vec:x,2) + ", " +
        round(vec:y,2) + ", " +
        round(vec:z,2) + ")".
```

```

}

local arrows is LIST().
function draw_abs_to_box {
    // Draws the vectors from origin TO the 2 opposite corners of the box:

    parameter B.

    // Wipe any old arrow draws off the screen.
    for arrow in arrows { set arrow:show to false. }×
    wait 0.

    ×
    arrows:CLEAR().
    arrows:ADD(Vecdraw(
        {return V(0,0,0).}, {return B:ABSMIN.}, RGB(1,0,0.75), "ABSMIN", 1, true)).
    arrows:ADD(Vecdraw(
        {return V(0,0,0).}, {return B:ABSMAX.}, RGB(1,0,0.75), "ABSMAX", 1, true)).
}

local edges is LIST().
function draw_box {
    // Draws a bounds box as a set of 12 non-pointy
    // vecdraws along the box edges:
    parameter B.

    // Wipe any old edge draws off the screen.
    for edge in edges { set edge:show to false. }×
    wait 0.

    // These need to calculate using relative coords to find all the box edges:
    local rel_x_size is B:RELMAX:X - B:RELMIN:X.
    local rel_y_size is B:RELMAX:Y - B:RELMIN:Y.
    local rel_z_size is B:RELMAX:Z - B:RELMIN:Z.

    edges:CLEAR().

    // The 4 edges parallel to the relative X axis:
    edges:ADD(Vecdraw(
        {return B:ABSORIGIN + B:FACING * V(B:RELMIN:X, B:RELMIN:Y, B:RELMIN:Z).},
        {return B:FACING * V(rel_x_size, 0, 0).},
        RGBA(1,0,1,0.75), "", 1, true, 0.02, false, false)).
    edges:ADD(Vecdraw(
        {return B:ABSORIGIN + B:FACING * V(B:RELMIN:X, B:RELMIN:Y, B:RELMAX:Z).},
        {return B:FACING * V(rel_x_size, 0, 0).},
        RGBA(1,0,1,0.75), "", 1, true, 0.02, false, false)).
    edges:ADD(Vecdraw(
        {return B:ABSORIGIN + B:FACING * V(B:RELMIN:X, B:RELMAX:Y, B:RELMAX:Z).},
        {return B:FACING * V(rel_x_size, 0, 0).},
        RGBA(1,0,1,0.75), "", 1, true, 0.02, false, false)).
    edges:ADD(Vecdraw(
        {return B:ABSORIGIN + B:FACING * V(B:RELMIN:X, B:RELMAX:Y, B:RELMIN:Z).},
        {return B:FACING * V(rel_x_size, 0, 0).},
        RGBA(1,0,1,0.75), "", 1, true, 0.02, false, false)).

    // The 4 edges parallel to the relative Y axis:

```

```

edges:ADD(Vecdraw(
    {return B:ABSORIGIN + B:FACING * V(B:RELMIN:X, B:RELMIN:Y, B:RELMIN:Z).},
    {return B:FACING * V(0, rel_y_size, 0).},
    RGBA(1,0,1,0.75), "", 1, true, 0.02, false, false)).
edges:ADD(Vecdraw(
    {return B:ABSORIGIN + B:FACING * V(B:RELMIN:X, B:RELMIN:Y, B:RELMAX:Z).},
    {return B:FACING * V(0, rel_y_size, 0).},
    RGBA(1,0,1,0.75), "", 1, true, 0.02, false, false)).
edges:ADD(Vecdraw(
    {return B:ABSORIGIN + B:FACING * V(B:RELMAX:X, B:RELMIN:Y, B:RELMAX:Z).},
    {return B:FACING * V(0, rel_y_size, 0).},
    RGBA(1,0,1,0.75), "", 1, true, 0.02, false, false)).
edges:ADD(Vecdraw(
    {return B:ABSORIGIN + B:FACING * V(B:RELMAX:X, B:RELMIN:Y, B:RELMIN:Z).},
    {return B:FACING * V(0, rel_y_size, 0).},
    RGBA(1,0,1,0.75), "", 1, true, 0.02, false, false).

// The 4 edges parallel to the relative Z axis:
edges:ADD(Vecdraw(
    {return B:ABSORIGIN + B:FACING * V(B:RELMIN:X, B:RELMIN:Y, B:RELMIN:Z).},
    {return B:FACING * V(0, 0, rel_z_size).},
    RGBA(1,0,1,0.75), "", 1, true, 0.02, false, false)).
edges:ADD(Vecdraw(
    {return B:ABSORIGIN + B:FACING * V(B:RELMIN:X, B:RELMAX:Y, B:RELMIN:Z).},
    {return B:FACING * V(0, 0, rel_z_size).},
    RGBA(1,0,1,0.75), "", 1, true, 0.02, false, false)).
edges:ADD(Vecdraw(
    {return B:ABSORIGIN + B:FACING * V(B:RELMAX:X, B:RELMAX:Y, B:RELMIN:Z).},
    {return B:FACING * V(0, 0, rel_z_size).},
    RGBA(1,0,1,0.75), "", 1, true, 0.02, false, false)).
edges:ADD(Vecdraw(
    {return B:ABSORIGIN + B:FACING * V(B:RELMAX:X, B:RELMIN:Y, B:RELMIN:Z).},
    {return B:FACING * V(0, 0, rel_z_size).},
    RGBA(1,0,1,0.75), "", 1, true, 0.02, false, false).
}

// =====
//      main program
// =====
//


local pNum is -1.
local keyPress is "".

until keyPress = "q" {

    local box is 0. // will get set to the bounds box in a moment.
    local description is "".

    clearscreen.

    if pNum = -1 {
        // PART NUMBER -1 will be a special flag this
        // example program uses to mean "entire vessel".
}

```

```

set box to ship:bounds.
set description to ship:TOSTRING().
} else {
local p is ship:parts[pNum].
set box to p:bounds.
set description to "Part[" + pNum + "]:" + p:TOSTRING().
}

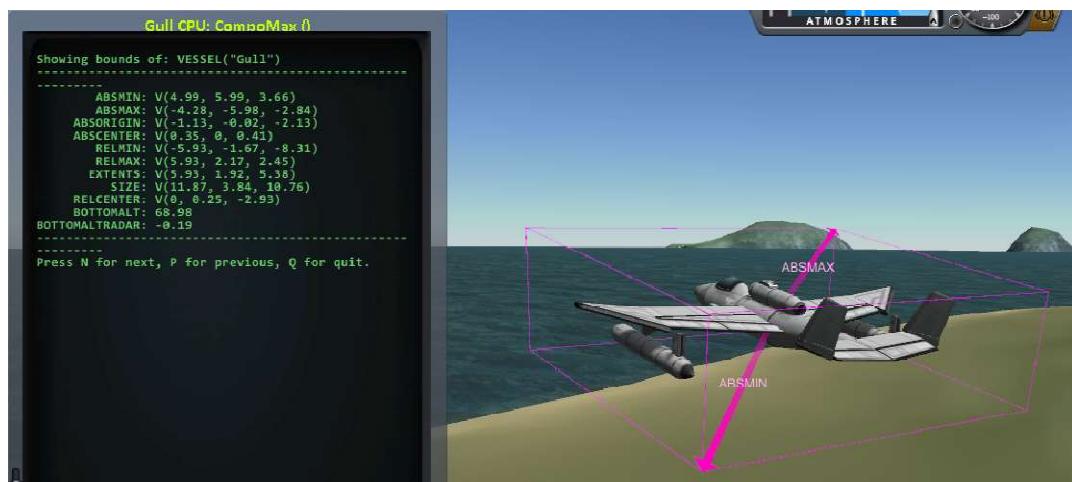
// These two functions do the actual drawing, and are defined
// below in this file. When trying to learn how this works,
// look at draw_abs_to_box() first - it's the simpler one to
// understand that just uses absolute coordinates. The other
// one, draw_box(), is more complex as it has to use the
// relative coords to get all the other corners of the box:
draw_abs_to_box(box).
draw_box(box).

print "Showing bounds of: " + description.
print "-----".
print "      ABSMIN: " + vector_tostring_rounded(box:ABSMIN).
print "      ABSMAX: " + vector_tostring_rounded(box:ABSMAX).
print "      ABSORIGIN: " + vector_tostring_rounded(box:ABSORIGIN).
print "      ABSCENTER: " + vector_tostring_rounded(box:ABSCENTER).
print "      RELMIN: " + vector_tostring_rounded(box:RELMIN).
print "      RELMAX: " + vector_tostring_rounded(box:RELMAX).
print "      EXTENTS: " + vector_tostring_rounded(box:EXTENTS).
print "      SIZE: " + vector_tostring_rounded(box:SIZE).
print "      RELCENTER: " + vector_tostring_rounded(box:RELCENTER).
print "      BOTTOMALT: " + round(box:BOTTOMALT,2).
print "      BOTTOMALTRADAR: " + round(box:BOTTOMALTRADAR,2).
print "-----".
print "Press N for next, P for previous, Q for quit.".
set keyPress to terminal:input:getchar().
if keyPress = "n" set pNum to min(ship:parts:length-1, pNum + 1).
if keyPress = "p" set pNum to max(-1, pNum - 1).

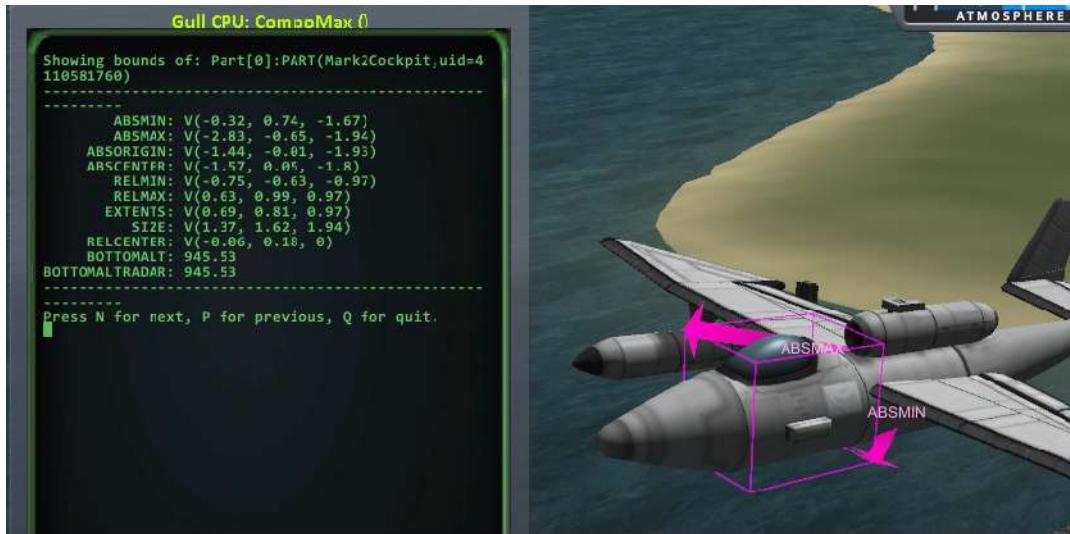
clearvecdraws().
}

```

运行程序会显示当前载具的边界框体，以及相应的尺寸信息。



在终端界面里输入 N，还能逐个部件地显示边界框体，以及相应的尺寸信息。



4.7

4.7 GUI 设计入门

4.7.1

4.7.1 导读

===== 点击以返回目录 =====



kOS 提供 GUI 设计的功能。GUI 指图形用户界面（Graphical User Interface）。

本节是 kOS 的 GUI 设计的入门教程，将介绍如何用 kOS 做出简单的控制面板窗口。

建议玩家在初学 kOS 时爱看不看。

本节内容：

GUI 的构成

实例9：GUI —— 天体信息查询器

- Step 0: 设计需求
- Step 1: 制作 GUI 界面
- Step 2: 为关闭键定义窗口关闭行为
- Step 3: 为查询键定义查询结果显示
- Step 4: 增加自启动功能
- Step 5: 最终效果

实例10：GUI —— 选项卡控件 TabWidget

- Step 0: 设计需求
- Step 1: 设计思路
- Step 2: 新建选项卡控件（Add TabWidget）
- Step 3: 添加选项卡（Add Tab）
- Step 4: 切换选项卡（Choose Tab）
- Step 5: 选项卡的切换显示逻辑

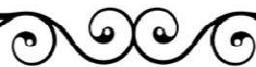
Step 6: 最终成品

Step 7: 使用选项卡控件

4.7.2

4.7.2 GUI 的构成

===== 点击以返回目录 =====

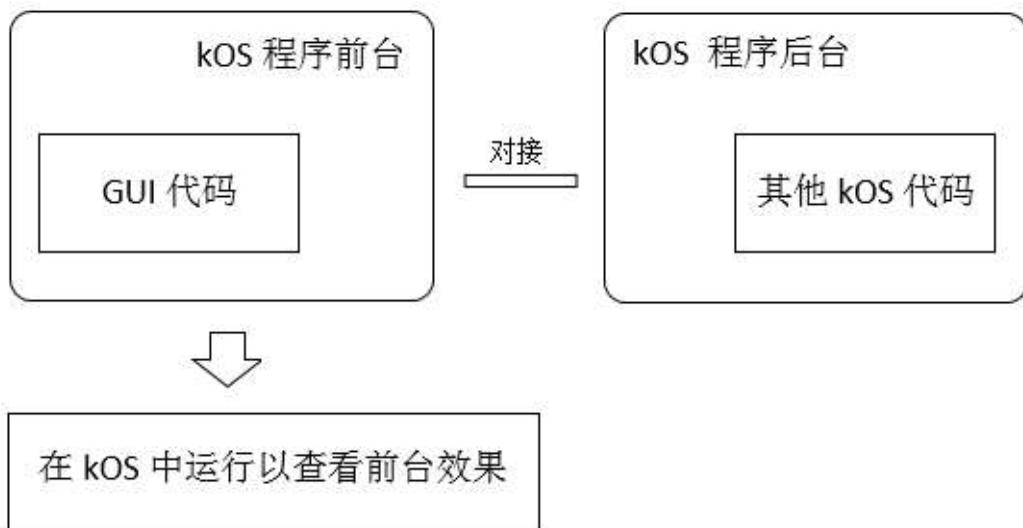


kOS 有个很有趣/有用的功能叫 GUI，能让玩家创造自己的对话窗口/控制面板，如下图：



kOS 只支持用代码进行这样的前台窗体设计。

kOS 提供了一套 GUI 相关的结构体和函数，用于设计前台窗体，就像普通编程里的前台代码一样，我们管他叫 GUI 代码。



kOS 中所有的前台元素以控件（Widget）形式存在的，kOS 专门为此设计了一套结构体。玩家可以用他们来设计属于自己的操作面板，就像在自创 MOD 一样。

	控件（结构体）	名称	说明
排版相关	Box	框体	用于其他控件的组合布局和排版，可作为选项卡的下属内容
	ScrollBar	滚动框体	一种特殊的框体（Box），可滚动以显示更多内容
	GUI	GUI	GUI 设计的起点，先要创建一个 GUI 结构体，之后才能在里边添加各种其他控件，GUI 对应窗体，一种特殊的框体（Box）
内容相关	Label	标注	可以显示文字或图片，没有交互功能
	TextField	文本框	一种特殊的标注（Label），用户可以编辑其中的文字
	Button	按钮	一种特殊的标注（Label），可以被按下以执行某种操作
	PopupMenu	下拉菜单	一种特殊的按钮（Button），带有一个列表给人从中选择
	Slider	滑块	可以通过拖拉来进行数值赋值
	Spacing	空档	用于排版时进行留白
样式相关	Style	样式	用于确定某一个控件的风格
	Skin	皮肤	一套样式（Style）的集合，用于确定GUI内所有控件的风格
	StyleRectOffset	样式边距	样式（Style）的子内容，用于确定控件的排版边距
	StyleState	样式状态	样式（Style）的子内容，用于确定控件显示方面的某些属性

4.7.3

4.7.3 实例9：GUI——天体信息查询器

===== 点击以返回目录 =====



Step 0：设计需求

我们来做一个天体信息查询器，

功能是显示我们所选择天体的信息，还有一个要求是自启动。

其实这程序并没有什么实际用处，就是后台简单，可以拿来画 GUI 练手的。

界面想了想，大概是要下面这个样子的：



Step 1: 制作 GUI 界面

```

Set gui to gui(300). //创建一个宽度是300像素的窗体

//窗口标题: 天体信息查询器
Set TitleLabel to gui:AddLabel("天体信息查询器").
Set TitleLabel:Style:FontSize to 16.
Set TitleLabel:Style:Align to "Center".

//查询条件的垂直框体
Set SearchBox to gui:AddVBox().
//垂直框体的第一行是文字“查询条件”，表示框体名称
Set SearchBoxLabel to SearchBox:AddLabel("查询条件").
Set SearchBoxLabel:Style:Align to "Center".
//垂直框体的第二行是下拉菜单和查询按钮
//要在同一行里显示多个控件时，需要先套一个水平框体
Set SearchBox1 to SearchBox:AddHBox().
Set SearchBox1:Style:BG to "".
//一个下拉菜单，让你选星球
Set BodyPopupMenu to SearchBox1:AddPopupMenu().
Set BodyList to List(
    "选择天体", "Sun", "Moho", "Eve", "Gilly", "Kerbin", "Mun", "Minmus",
    "Duna", "Ike", "Jool", "Laythe", "Vall", "Tylo", "Bop",
    "Pol", "Eeloo").
For Planet in BodyList {
    BodyPopupMenu:AddOption(Planet).

```

```

}

//一个查询的按钮
Set ButtonSearch to SearchBox1:AddButton("查询").

//留白10个像素
Set space1 to gui:AddSpacing(10).

//查询结果的垂直框体，分为好几行
Set ResultBox to gui:AddVBox().
//垂直框体的第一行是文字“查询结果”，表示框体名称
Set ResultBoxLabel to ResultBox:AddLabel("查询结果").
Set ResultBoxLabel:Style:Align to "Center".
//垂直框体的第二行开始是一个蚊子列表，显示查询结果
//要在同一行里显示多个控件时，需要先套一个水平框体
//水平框体的背景图像（BG）被取消了，否则嵌套着显示会很难看。
Set ResultBox1 to ResultBox:AddHBox(). Set ResultBox1:Style:BG to "".
//每行左侧是参数名
Set ResultLabel1 to ResultBox1:AddLabel("天体名称").
Set ResultLabel1:Style:Width to 100.
//每行右侧是参数值，默认是表示无数据的 NA,
Set BodyName to ResultBox1:AddLabel("NA").
Set ResultBox2 to ResultBox:AddHBox(). Set ResultBox2:Style:BG to "".
Set ResultLabel2 to ResultBox2:AddLabel("天体质量 (kg)").
Set ResultLabel2:Style:Width to 100.
Set RMass to ResultBox2:AddLabel("NA").
Set ResultBox3 to ResultBox:AddHBox(). Set ResultBox3:Style:BG to "".
Set ResultLabel3 to ResultBox3:AddLabel("天体半径 (m)").
Set ResultLabel3:Style:Width to 100.
Set RRadius to ResultBox3:AddLabel("NA").
Set ResultBox4 to ResultBox:AddHBox(). Set ResultBox4:Style:BG to "".
Set ResultLabel4 to ResultBox4:AddLabel("是否有大气层").
Set ResultLabel4:Style:Width to 100.
Set RHasAtm to ResultBox4:AddLabel("NA").
Set ResultBox5 to ResultBox:AddHBox(). Set ResultBox5:Style:BG to "".
Set ResultLabel5 to ResultBox5:AddLabel("公转周期 (s)").
Set ResultLabel5:Style:Width to 100.
Set RevoT to ResultBox5:AddLabel("NA").
Set ResultBox6 to ResultBox:AddHBox(). Set ResultBox6:Style:BG to "".
Set ResultLabel6 to ResultBox6:AddLabel("公转离心率").
Set ResultLabel6:Style:Width to 100.
Set RevoE to ResultBox6:AddLabel("NA").
Set ResultBox7 to ResultBox:AddHBox(). Set ResultBox7:Style:BG to "".
Set ResultLabel7 to ResultBox7:AddLabel("公转速度 (m/s)").
Set ResultLabel7:Style:Width to 100.
Set RevoV to ResultBox7:AddLabel("NA").
Set space2 to gui:AddSpacing(10).
Set CloseBox to gui:AddHBox(). Set CloseBox:Style:BG to "".
Set space3 to CloseBox:AddSpacing(145).
//一个窗口关闭的按钮
Set ButtonClose to CloseBox:AddButton("关闭").

gui>Show() //显示这个窗体

```

上面的代码扔到 kOS 里差不多是这样子的。到目前为止感觉不错。

我们发觉了，kOS 的 GUI 界面是支持中文文字的。

事实上，kOS 的指令窗口里不支持中文的显示，但是运算是支持的，GUI 也是支持显示的。





Step 2: 为关闭键定义窗口关闭行为

接下来我们要把“点击关闭按钮”的动作关联上“GUI 窗口的隐藏”行为。

为此需要在刚才末尾的“gui:Show ().”之后再添加相应的函数和关联代码。

我们还需要一个变量作为标记，来代表是不是需要关闭窗口。

```
//一个初值为假的值，用来标记是否要关闭窗口
Set isClosed to False.

//将点击关闭按钮的动作关联到一个把标记改为真值的函数上
Function CloseClick {
    Set isClosed to True.
}

Set ButtonClose:OnClick to CloseClick@.

Wait Until isClosed. //一旦这个标记为真了，就隐藏窗体
gui:Hide().
```

Step 3: 为查询键定义查询结果显示

然后我们要把“点击查询按钮”的动作关联上“显示查询结果”的行为。

为此需要在在代码里添加相应的函数和关联代码。

添加的位置不是很在意，在“gui:Show ().”之后和“Wait Until isClosed.”之前就可以了。

```

//搜索函数。用于在结果区域显示搜索结果的函数
Function SearchClick {
    //先排除选了第一个“选择天体”的情况，这种情况直接结束函数×
    If BodyPopupMenu:Value <> "选择天体" {
        //从天体名字符串获取天体结构体×
        Set SBody to Body(BodyPopupMenu:Value).
        //再从天体结构体获取各种信息×
        Set BodyName:Text to SBody:Name.
        Set RMass:Text to SBody:Mass:ToString.
        Set RRadius:Text to SBody:Radius:ToString.
        //根据所选天体是否有大气层来决定如何显示×
        If SBody:Atm:Exists {
            Set RHasAtm:Text to "有".
        } Else {
            Set RHasAtm:Text to "没有".
        }
        //如果天体有轨道（只有太阳没有自己的轨道）×
        If SBody:HasOrbit {
            Set RevoT:Text to SBody:Orbit:Period:ToString.
            Set RevoE:Text to SBody:Orbit:Eccentricity:ToString.
            //轨道速度被包装在多层结构体下，要一个个开进去，
            //还要用 Mag 后缀对矢量取模，还要用 Lock 来动态显示×
            Set RevoV:Text to SBody:Orbit:Velocity:Orbit:Mag:ToString.
            //如果天体是太阳，那么公转信息则显示为空
        } Else {
            Set RevoT:Text to "NA".
            Set RevoE:Text to "NA".
            Set RevoV:Text to "NA".
        }
    }
    //将点击搜索按钮的动作关联到搜索函数上
    Set ButtonSearch:OnClick to SearchClick@.
}

```

Step 4：增加自启动功能

kOS 可为程序设置自启动。步骤是这样的：

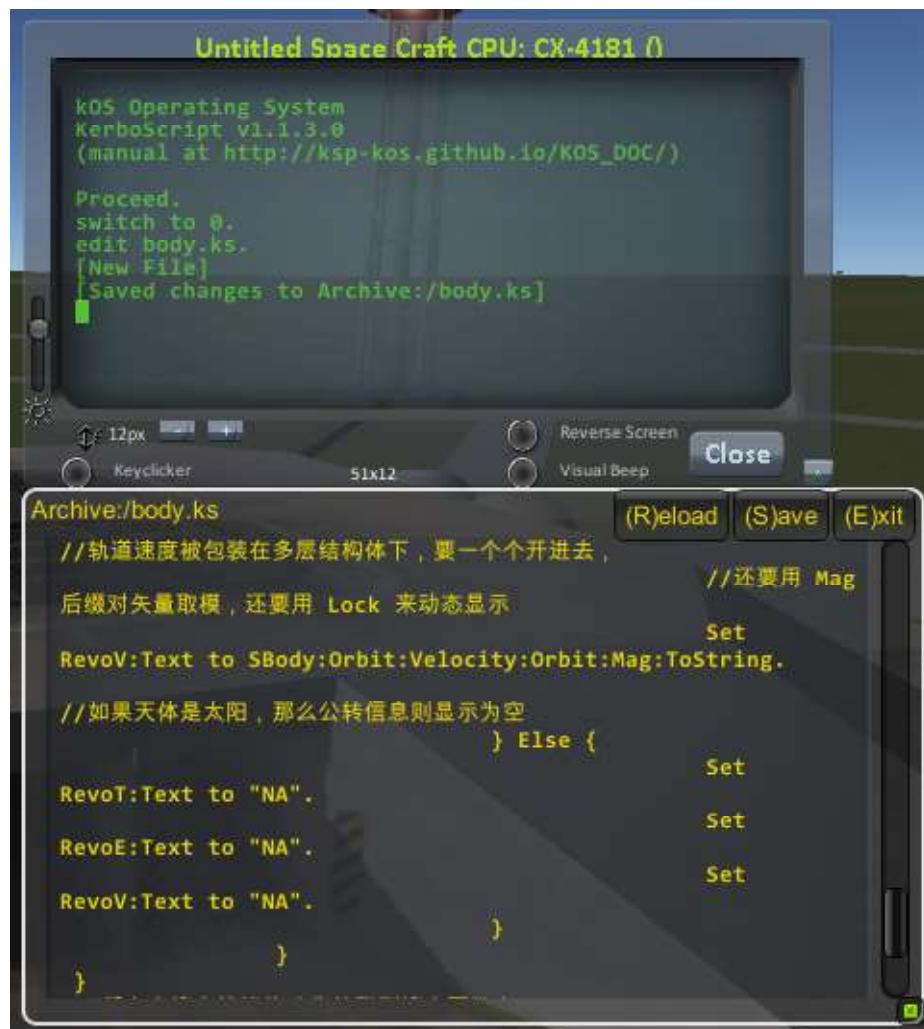
- 首先，在指令窗口里依次输入下述代码，其中的 body.ks 可以换成自己的文件名，但要是 .ks 后缀。

```

switch to 0.
edit body.ks.

```

- 然后可以看到新出现了弹窗让你编辑这个 body.ks 文件的内容，此时把自己的源码贴进去，保存。



3. 接下来我们在游戏目录下的 Ships/Script 文件夹下看到这个 body.ks 文件。我们需要把它放到同目录下的 boot 文件夹内。

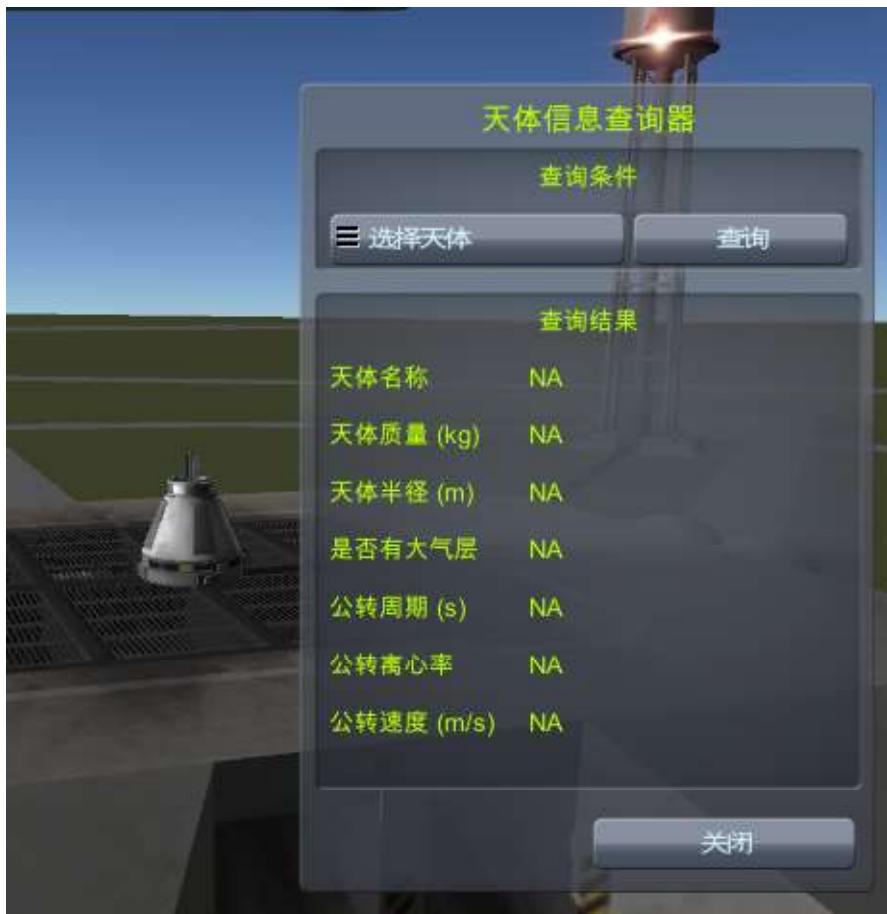


4. 下一步，我们进入车间，在载具的 kOS 部件上打开右键菜单，我们将启动文件选择为 Body.ks 文件。



4. 下一步，我们进入车间，在载具的 kOS 部件上打开右键菜单，我们将启动文件选择为 Body.ks 文件。

5. 最后我们点击 Lanch 进行载具发射，结果如下图所见，不需要我们操作，Body.ks 程序就自动运行了，并且跳出了查询器的窗口。



*注1：在这里我们利用了 kOS 里的文本编辑器来生成 .ks 源代码文件，而不是记事本或者其他东西，是因为 kOS 能识别的编码比较特殊，记事本或者其他东西生成的会导致 kOS 无法正确识别，后果轻则中文乱码，重则报错无法运行程序。

*注2：其实直接用记事本啥的，保存成 ANSI 编码的文本文件，也可以运行，就是运行时里边的中文会乱码显示。

如果文件只有注释里有中文，实际代码里边没有，那这样做也可以。

Step 5：最终效果

好了我们把之前几步做的所有代码连起来，如下。

并按照上一小节中说的自启动配置步骤来操作。我们得到了最后正常工作时候的样子。

```
Set gui to gui(300). //创建一个宽度是300像素的窗体

//窗口标题：天体信息查询器
Set TitleLabel to gui:AddLabel("天体信息查询器").
Set TitleLabel:Style:FontSize to 16.
Set TitleLabel:Style:Align to "Center".

//查询条件的垂直框体
Set SearchBox to gui:AddVBox().
//垂直框体的第一行是文字“查询条件”，表示框体名称
Set SearchBoxLabel to SearchBox:AddLabel("查询条件").
Set SearchBoxLabel:Style:Align to "Center".
//垂直框体的第二行是下拉菜单和查询按钮
//要在同一行里显示多个控件时，需要先套一个水平框体
Set SearchBox1 to SearchBox:AddHBox().
Set SearchBox1:Style:BG to "".
//一个下拉菜单，让你选星球
Set BodyPopupMenu to SearchBox1:AddPopupMenu().
Set BodyList to List(
    "选择天体", "Sun", "Moho", "Eve", "Gilly", "Kerbin", "Mun", "Minmus",
    "Duna", "Ike", "Jool", "Laythe", "Vall", "Tylo", "Bop",
    "Pol", "Eeloo").
For Planet in BodyList {
    BodyPopupMenu:AddOption(Planet).
}
//一个查询的按钮
Set ButtonSearch to SearchBox1:AddButton("查询").

//留白10个像素
Set space1 to gui:AddSpacing(10).

//查询结果的垂直框体，分为好几行
Set ResultBox to gui:AddVBox().
//垂直框体的第一行是文字“查询结果”，表示框体名称
Set ResultBoxLabel to ResultBox:AddLabel("查询结果").
Set ResultBoxLabel:Style:Align to "Center".
//垂直框体的第二行开始是一个蚊子列表，显示查询结果
//要在同一行里显示多个控件时，需要先套一个水平框体
//水平框体的背景图像（BG）被取消了，否则嵌套着显示会很难看。
Set ResultBox1 to ResultBox:AddHBox(). Set ResultBox1:Style:BG to "".
//每行左侧是参数名
Set ResultLabel1 to ResultBox1:AddLabel("天体名称").
Set ResultLabel1:Style:Width to 100.
```

```

//每行右侧是参数值， 默认是表示无数据的 NA,
Set BodyName to ResultBox1:AddLabel("NA").
Set ResultBox2 to ResultBox:AddHBox(). Set ResultBox2:Style:BG to "".
Set ResultLabel2 to ResultBox2:AddLabel("天体质量 (kg)").
Set ResultLabel2:Style:Width to 100.
Set RMass to ResultBox2:AddLabel("NA").
Set ResultBox3 to ResultBox:AddHBox(). Set ResultBox3:Style:BG to "".
Set ResultLabel3 to ResultBox3:AddLabel("天体半径 (m)").
Set ResultLabel3:Style:Width to 100.
Set RRadius to ResultBox3:AddLabel("NA").
Set ResultBox4 to ResultBox:AddHBox(). Set ResultBox4:Style:BG to "".
Set ResultLabel4 to ResultBox4:AddLabel("是否有大气层").
Set ResultLabel4:Style:Width to 100.
Set RHAsAtm to ResultBox4:AddLabel("NA").
Set ResultBox5 to ResultBox:AddHBox(). Set ResultBox5:Style:BG to "".
Set ResultLabel5 to ResultBox5:AddLabel("公转周期 (s)").
Set ResultLabel5:Style:Width to 100.
Set RevoT to ResultBox5:AddLabel("NA").
Set ResultBox6 to ResultBox:AddHBox(). Set ResultBox6:Style:BG to "".
Set ResultLabel6 to ResultBox6:AddLabel("公转离心率").
Set ResultLabel6:Style:Width to 100.
Set RevoE to ResultBox6:AddLabel("NA").
Set ResultBox7 to ResultBox:AddHBox(). Set ResultBox7:Style:BG to "".
Set ResultLabel7 to ResultBox7:AddLabel("公转速度 (m/s)").
Set ResultLabel7:Style:Width to 100.
Set RevoV to ResultBox7:AddLabel("NA").
Set space2 to gui:AddSpacing(10).
Set CloseBox to gui:AddHBox(). Set CloseBox:Style:BG to "".
Set space3 to CloseBox:AddSpacing(145).
//一个窗口关闭的按钮
Set ButtonClose to CloseBox:AddButton("关闭").

gui>Show() //显示这个窗体

//搜索函数。用于在结果区域显示搜索结果的函数
Function SearchClick {
    //先排除选了第一个“选择天体”的情况，这种情况直接结束函数×
    If BodyPopupMenu:Value <> "选择天体" {
        //从天体名字符串获取天体结构体×
        Set SBody to Body(BodyPopupMenu:Value).
        //再从天体结构体获取各种信息×
        Set BodyName:Text to SBody:Name.
        Set RMass:Text to SBody:Mass:ToString.
        Set RRadius:Text to SBody:Radius:ToString.
        //根据所选天体是否有大气层来决定如何显示×
        If SBody:Atm:Exists {
            Set RHAsAtm:Text to "有".
        } Else {
            Set RHAsAtm:Text to "没有".
        }
        //如果天体有轨道（只有太阳没有自己的轨道）×
        If SBody:HasOrbit {
            Set RevoT:Text to SBody:Orbit:Period:ToString.
        }
    }
}

```

```

Set RevoE:Text to SBody:Orbit:Eccentricity:ToString.
//轨道速度被包装在多层结构体下，要一个个开进去，
//还要用 Mag 后缀对矢量取模，还要用 Lock 来动态显示
Set RevoV:Text to SBody:Orbit:Velocity:Orbit:Mag:ToString.
//如果天体是太阳，那么公转信息则显示为空
} Else {
    Set RevoT:Text to "NA".
    Set RevoE:Text to "NA".
    Set RevoV:Text to "NA".
}
}

//将点击搜索按钮的动作关联到搜索函数上
Set ButtonSearch:OnClick to SearchClick@.

//一个初值为假的值，用来标记是否要关闭窗口
Set isClosed to False.

//将点击关闭按钮的动作关联到一个把标记改为真值的函数上
Function CloseClick {
    Set isClosed to True.
}
Set ButtonClose:OnClick to CloseClick@.

Wait Until isClosed. //一旦这个标记为真了，就隐藏窗体
gui:Hide().

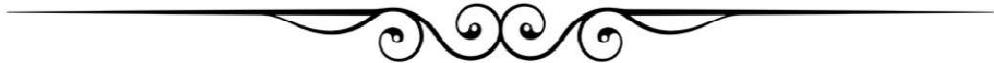
```



4.7.4

4.7.4 实例10：GUI——选项卡控件 TabWidget

===== 点击以返回目录 =====



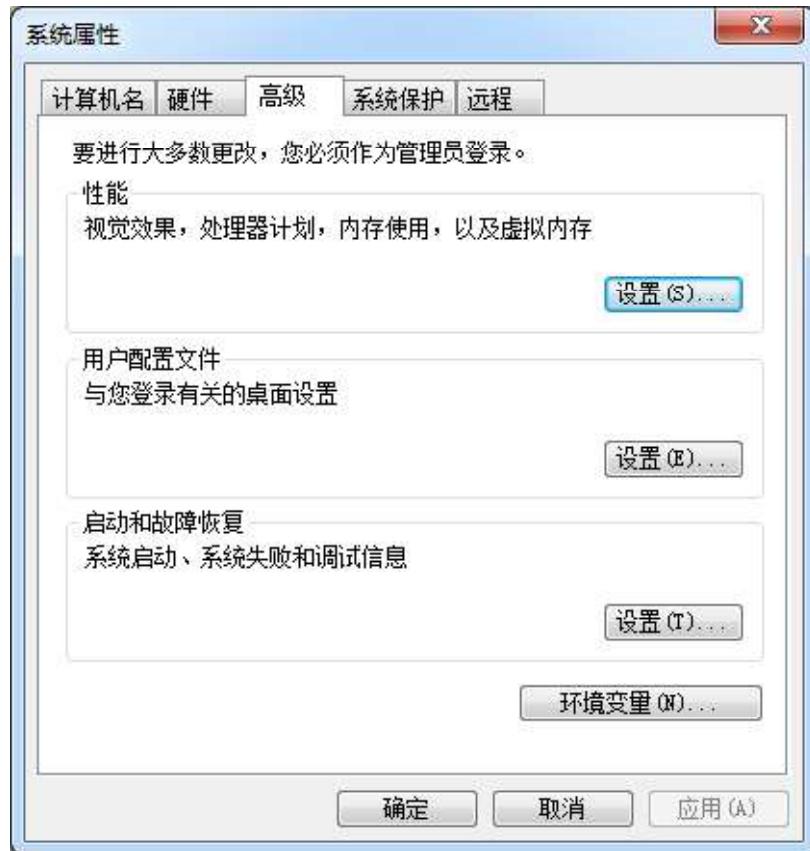
Step 0：设计需求

	控件（结构体）	名称	说明
排版相关	Box	框体	用于其他控件的组合布局和排版，可作为选项卡的下属内容
	ScrollBar	滚动框体	一种特殊的框体（Box），可滚动以显示更多内容
	GUI	GUI	GUI 设计的起点，先要创建一个 GUI 结构体，之后才能在里边添加各种其他控件，GUI 对应窗体，一种特殊的框体（Box）
内容相关	Label	标注	可以显示文字或图片，没有交互功能
	TextField	文本框	一种特殊的标注（Label），用户可以编辑其中的文字
	Button	按钮	一种特殊的标注（Label），可以被按下以执行某种操作
	PopupMenu	下拉菜单	一种特殊的按钮（Button），带有一个列表给人从中选择
	Slider	滑块	可以通过拖拉来进行数值赋值
	Spacing	空档	用于排版时进行留白
样式相关	Style	样式	用于确定某一个控件的风格
	Skin	皮肤	一套样式（Style）的集合，用于确定GUI内所有控件的风格
	StyleRectOffset	样式边距	样式（Style）的子内容，用于确定控件的排版边距
	StyleState	样式状态	样式（Style）的子内容，用于确定控件显示方面的某些属性

回来看一下上面这个表，这个表显示，kOS 提供了多种控件供玩家使用。

但里边少了一种很有用的：**选项卡控件（TabWidget）**。

选项卡平时随处可见，有利于在窗口中展现更多内容，他有利于窗口中内容的归类排版。



这里我们的目的就是，一来是通过已有的 kOS 控件，实现选项卡的效果；二来是要动手做一个库函数，一劳永逸，使得玩家可以像使用其他控件一样，方便的进行选项卡操作。

Step 1：设计思路

其实我们有办法做出来这种选项卡切换的效果的：

我们的选项卡控件（Box）下要放置两个子框体（Box），其中一个是横向排列的子框体（Box）负责显示选项卡标签（按钮，Button），另一个是纵向排列的子框体（Box）负责显示选项卡对应的内容（孙子框体，Box），而后，我们只需要写个逻辑，让某一个选项卡（按钮，Button）被选中后，非对应的选项卡内容（孙子框体，Box）都隐藏掉，只显示对应的，就可以了。如下图：



然后我们还要对两个子框体（Box）的样式进行调整，美化一下，使得选项卡控件（Box）下要的两个子框体（Box）看上去是连在一起的，并且各自的背景图案能连接的很好。就像下面那样：



所以总结一下，我们要将如下的行为做到函数化

新建选项卡控件（Add TabWidget）

添加选项卡（Add Tab）

选中选项卡（Choose Tab）

选项卡的切换显示逻辑

Step 2: 新建选项卡控件（Add TabWidget）

我们需要在游戏目录的 Ships\Script\TabWidget 文件夹下开一个 tabwidget 文件，里面贴上下面的话。

```

//使用时用法是这样 AddTabWidget(box). 会在母框体box里边添加一个选项卡控件TabWidget
//选项卡控件TabWidget本质是一个预设了一些函数声明的框体Box,
//调用函数可以使这个框体Box的行为模拟出选项卡控件的效果
DECLARE FUNCTION AddTabWidget
{
    //参数是指, 要在参数 (母框体Box) 内添加选项卡控件
    DECLARE PARAMETER box.

    //查询所在位置是否有应用了选项卡皮肤TabWidgetTab
    //如果没有应用, 则逐个定义式样Style
    //这种样式最终会被应用在选项卡 (按钮, Button) 上。
    IF NOT box:GUI:SKIN:HAS("TabWidgetTab") {

        //控件的式样可修改度非常大
        // 这里我们需要改控件 (按钮, Button) 的背景图片,
        //我们可以修改通常、被鼠标聚焦、被鼠标点击, 几种不同状态下
        //分别设置显示效果, 和挂钩的函数。
        LOCAL style IS box:GUI:SKIN:ADD("TabWidgetTab",box:GUI:SKIN:BUTTON).

        //我们要设置选项卡 (按钮, Button) 不同状态下的背景图片,
        //这些图片我们需要先准备好, 放到对应文件夹下
        //文件默认为.png图片, 所以不写文件后缀名是可以的
        SET style:BG TO "TabWidget/images/back".
        SET style:ON:BG to "TabWidget/images/front".
        //而且还要完善一下按钮上文字的样式定义
        SET style:TEXTCOLOR TO RGBA(0.7,0.75,0.7,1).
        SET style:HOVER:BG TO "".
        SET style:HOVER_ON:BG TO "".
        SET style:margin:H TO 0.
        SET style:margin:bottom TO 0.
    }

    //查询所在位置是否有应用了选项卡皮肤TabWidgetPanel
    //如果没有应用, 则逐个定义式样Style
    //这种样式最终会被应用在选项卡内容 (框体, Box) 上。
    IF NOT box:GUI:SKIN:HAS("TabWidgetPanel") {

        LOCAL style IS box:GUI:SKIN:ADD("TabWidgetPanel",box:GUI:SKIN:WINDOW).
        //这里我们也有一个背景图片要准备, 放到对应文件夹下
        //文件默认为.png图片, 所以不写文件后缀名是可以的
        SET style:BG TO "TabWidget/images/panel".
        SET style:padding:top TO 0.
    }

    //进行框体布置
    LOCAL vbox IS box:ADDVLayout. //ADDVLayout是指嵌套加入垂直排列的框体
    LOCAL tabs IS vbox:ADDHLAYOUT. //ADDHLAYOUT是指加入水平排列的框体
    LOCAL panels IS vbox:ADDSTACK. //ADDSTACK是指嵌套加入一个和自己一样的框体

    //返回所创建的最外层的框体
    RETURN vbox.
}

```

本段代码使用的背景图片如下。

从左至右分别是：选项卡被选中时、选项卡未被选中时、选项卡内容



再来比较一下，这三个背景图片做出来效果就是下面这样的。



Step 3: 添加选项卡 (Add Tab)

然后我们继续在刚才的 tabwidget 文件里，追加下面的话。

```

DECLARE FUNCTION AddTab
{
    DECLARE PARAMETER tabwidget. //第一个参数是框体，之前新建的选项卡控件
    DECLARE PARAMETER tabname. //第二个参数是字符串，要添加的选项卡的标题

    //从 tabwidgets 中拆出 选项卡 (tab) 部分，还有选项卡内容 (panel) 部分。
    LOCAL hboxes IS tabwidget:WIDGETS.
    LOCAL tabs IS hboxes[0].
    LOCAL panels IS hboxes[1].

    //在选项卡内容 (panel) 部分的框体 (Box) 里边增加一个垂直框体 (Box)
    //并且设置他的样式
    LOCAL panel IS panels:ADDVBOX.
    SET panel:STYLE TO panel:GUI:SKIN:GET("TabWidgetPanel").

    //在选项卡 (tab) 部分的框体 (Box) 里边增加一个选项卡 (按钮， Button)
    //并且设置他的样式
    LOCAL tab IS tabs:ADDBUTTON(tabname).
    SET tab:STYLE TO tab:GUI:SKIN:GET("TabWidgetTab").

    //切换到所添加的 tab选项卡 (按钮， Button)， 按钮设置为按下去不会自动弹起来
    SET tab:TOGGLE TO true.
    //选项卡 (按钮， Button) 是排他性的，按下去之后同Box内其他按钮都灰掉
    SET tab:EXCLUSIVE TO true.

    //如果添加了 tab 之后， tabwidget 里边有且仅有一个选项卡，
    //就显示这个选项卡，否则的话就隐藏整个 tabwidget,
    IF panels:WIDGETS:LENGTH = 1 {
        SET tab:PRESSED TO true.
        panels:SHOWONLY(panel).
    } else {
        panel:HIDE().
    }

    //把新增的 选项卡tab 和 选项卡内容panel 添加到全局变量列表里，之后要用到
    TabWidget_alltabs:ADD(tab).
    TabWidget_allpanels:ADD(panel).

    RETURN panel. //返回新添加的 选项卡内容panel
}

//分别对 选项卡tab 和 选项卡内容panel 创建全局变量的列表，之后要用到
GLOBAL TabWidget_alltabs TO LIST().
GLOBAL TabWidget_allpanels TO LIST().

```

Step 4: 切换选项卡 (Choose Tab)

然后我们继续在刚才的 tabwidget 文件里，追加下面的话。

```

DECLARE FUNCTION ChooseTab
{
    DECLARE PARAMETER tabwidget. //第一个参数是框体，之前新建的选项卡控件×
    DECLARE PARAMETER tabnum. //第二个参数是数字，这里边的第几个选项卡
    //从 tabwidget 里拆出 选项卡tab 的列表×
    LOCAL hboxes IS tabwidget:WIDGETS.
    LOCAL tabs IS hboxes[0].
    //将 选项卡tab 列表里第 tabnum 个选项卡设为选中×
    SET tabs:WIDGETS[tabnum]:PRESSED TO true.
}

```

Step 5: 选项卡的切换显示逻辑

然后我们继续在刚才的 tabwidget 文件里，追加下面的话。

```

//每个物理帧都查询一遍所有的选项卡,
//要让选项卡内容 (panel) 的变化跟上 ChooseTab函数的选择
WHEN True THEN {
    //遍历所有的 选项卡tab×
    FROM { LOCAL x IS 0. } UNTIL x >= TabWidget_alltabs:LENGTH STEP { SET x TO x+1. } DO×
    {
        //如果被按下按钮的选项卡内容 (panel) 不是正在显示 (VISIBLE) 状态的话,
        //那么就让这个选项卡内容 (panel) 进行独占显示 (showonly) ×
        IF TabWidget_alltabs[x]:PRESSED AND NOT TabWidget_allpanels[x]:VISIBLE {
            TabWidget_allpanels[x]:parent:showonly(TabWidget_allpanels[x]).}
        }
    }×
    PRESERVE. //为触发结构续命
}

```

Step 6: 最终成品

我们把刚才准备的东西连起来看一下。

```

//=====添加选项卡控件TabWidget 开始=====
//使用时用法是这样 AddTabWidget(box). 会在母框体box里边添加一个选项卡控件TabWidget
//选项卡控件TabWidget本质是一个预设了一些函数声明的框体Box,
//调用函数可以使这个框体Box的行为模拟出选项卡控件的效果
DECLARE FUNCTION AddTabWidget
{
    //参数是指，要在参数 (母框体Box) 内添加选项卡控件×
    DECLARE PARAMETER box.

    //查询所在位置是否有应用了选项卡皮肤TabWidgetTab
    //如果没有应用，则逐个定义式样Style
    //这种样式最终会被应用在选项卡 (按钮, Button) 上。
    IF NOT box:GUI:SKIN:HAS ("TabWidgetTab") {

        //控件的式样可修改度非常大
        // 这里我们需要改控件 (按钮, Button) 的背景图片,
        //我们可以修改通常、被鼠标聚焦、被鼠标点击，几种不同状态下
        //分别设置显示效果，和挂钩的函数。
    }
}

```

```

LOCAL style IS box:GUI:SKIN:ADD("TabWidgetTab",box:GUI:SKIN:BUTTON).

//我们要设置选项卡（按钮， Button）不同状态下的背景图片,
//这些图片我们需要先准备好，放到对应文件夹下
//文件默认为.png图片，所以不写文件后缀名是可以的
SET style:BG TO "TabWidget/images/back".
SET style:ON:BG TO "TabWidget/images/front".
//而且还要完善一下按钮上文字的样式定义
SET style:TEXTCOLOR TO RGBA(0.7,0.75,0.7,1).
SET style:HOVER:BG TO "".
SET style:HOVER_ON:BG TO "".
SET style:margin:H TO 0.
SET style:margin:bottom TO 0.
}

//查询所在位置是否有应用了选项卡皮肤TabWidgetPanel
//如果没有应用，则逐个定义式样Style
//这种样式最终会被应用在选项卡内容（框体， Box）上。
IF NOT box:GUI:SKIN:HAS("TabWidgetPanel") {
    LOCAL style IS box:GUI:SKIN:ADD("TabWidgetPanel",box:GUI:SKIN:WINDOW).
    //这里我们也有一个背景图片要准备，放到对应文件夹下
    //文件默认为.png图片，所以不写文件后缀名是可以的
    SET style:BG TO "TabWidget/images/panel".
    SET style:padding:top TO 0.
}

//进行框体布置
LOCAL vbox IS box:ADDVLayout. //ADDVLayout是指嵌套加入垂直排列的框体
LOCAL tabs IS vbox:ADDHLayout. //ADDHLayout是指加入水平排列的框体
LOCAL panels IS vbox:ADDSTACK. //ADDSTACK是指嵌套加入一个和自己一样的框体

//返回所创建的最外层的框体
RETURN vbox.
}

//=====添加选项卡控件TabWidget 结束=====
//=====添加选项卡Tab 开始=====

DECLARE FUNCTION AddTab
{
    DECLARE PARAMETER tabwidget. //第一个参数是框体，之前新建的选项卡控件
    DECLARE PARAMETER tabname. //第二个参数是字符串，要添加的选项课的标题

    //从 tabwidgets 中拆出 选项卡（tab）部分，还有选项卡内容（panel）部分。
    LOCAL hboxes IS tabwidget:WIDGETS.
    LOCAL tabs IS hboxes[0].
    LOCAL panels IS hboxes[1].

    //在选项卡内容（panel）部分的框体（Box）里边增加一个垂直框体（Box）
    //并且设置他的样式
    LOCAL panel IS panels:ADDVBOX.
    SET panel:STYLE TO panel:GUI:SKIN:GET("TabWidgetPanel").

    //在选项卡（tab）部分的框体（Box）里边增加一个选项卡（按钮， Button）
}

```

```

//并且设置他的样式
LOCAL tab IS tabs:ADDBUTTON(tabname).
SET tab:STYLE TO tab:GUI:SKIN:GET("TabWidgetTab").

//切换到所添加的 tab选项卡 (按钮, Button) , 按钮设置为按下去不会自动弹起来
SET tab:TOGGLE TO true.
//选项卡 (按钮, Button) 是排他性的, 按下去之后同Box内其他按钮都灰掉
SET tab:EXCLUSIVE TO true.

//如果添加了 tab 之后, tabwidget 里边有且仅有一个选项卡,
//就显示这个选项卡, 否则的话就隐藏整个 tabwidget,
IF panels:WIDGETS:LENGTH = 1 {
    SET tab:PRESSED TO true.
    panels:SHOWONLY(panel).
} else {
    panel:HIDE().
}

//把新增的 选项卡tab 和 选项卡内容panel 添加到全局变量列表里, 之后要用到
TabWidget_alltabs:ADD(tab).
TabWidget_allpanels:ADD(panel).

RETURN panel. //返回新添加的 选项卡内容panel
}

//分别对 选项卡tab 和 选项卡内容panel 创建全局变量的列表, 之后要用到
GLOBAL TabWidget_alltabs TO LIST().
GLOBAL TabWidget_allpanels TO LIST().

//=====添加选项卡Tab 开始=====
//=====选择选项卡ChooseTab 开始=====

DECLARE FUNCTION ChooseTab
{
    DECLARE PARAMETER tabwidget. //第一个参数是框体, 之前新建的选项卡控件
    DECLARE PARAMETER tabnum. //第二个参数是数字, 这里边的第几个选项卡
    //从 tabwidget 里拆出 选项卡tab 的列表
    LOCAL hboxes IS tabwidget:WIDGETS.
    LOCAL tabs IS hboxes[0].
    //将 选项卡tab 列表里第 tabnum 个选项卡设为选中
    SET tabs:WIDGETS[tabnum]:PRESSED TO true.
}

//=====选择选项卡ChooseTab 结束=====
//=====选项卡切换效果 开始=====

//每个物理帧都查询一遍所有的选项卡,
//要让选项卡内容 (panel) 的变化跟上 ChooseTab函数的选择
WHEN True THEN {
    //遍历所有的 选项卡tab
    FROM { LOCAL x IS 0. } UNTIL x >= TabWidget_alltabs:LENGTH STEP { SET x TO x+1. } DO
    {
        //如果被按下按钮的选项卡内容 (panel) 不是正在显示 (VISIBLE) 状态的话,
}

```

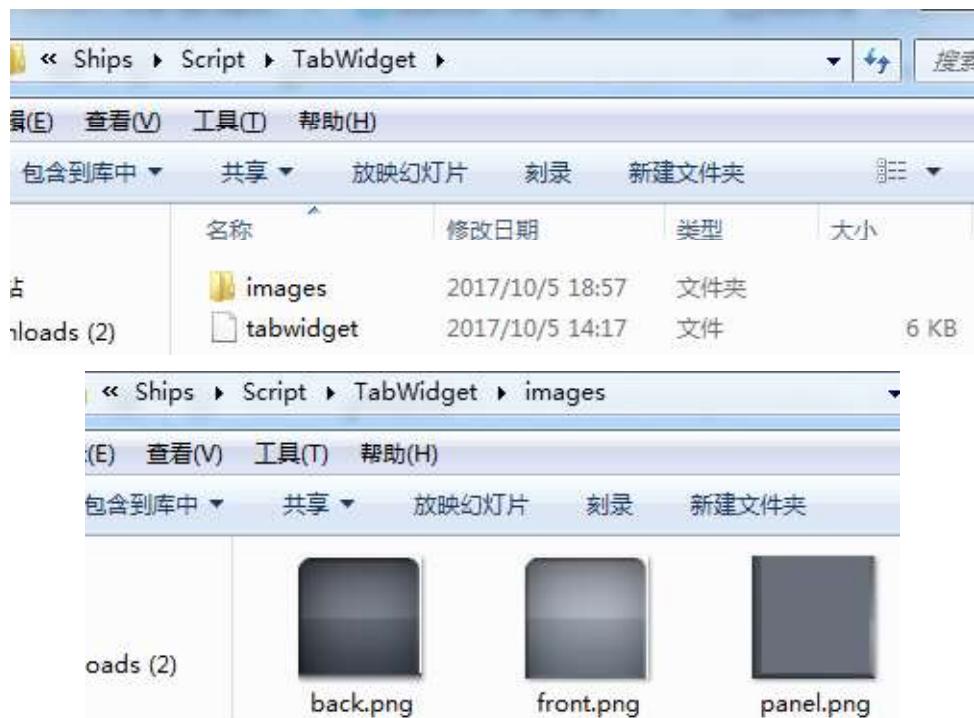
```

    //那么就让这个选项卡内容 (panel) 进行独占显示 (showonly) ×
    IF TabWidget_alltabs[x]:PRESSED AND NOT TabWidget_allpanels[x]:VISIBLE {
        TabWidget_allpanels[x]:parent:showonly(TabWidget_allpanels[x]).}
    }
}×
PRESERVE. //为触发结构续命
}

//=====选项卡切换效果 结束=====

```

我们把上面的代码存成名为 tabwidget 的文件，保存在 Ships\Script\TabWidget 文件夹下。至此，我们的选项卡控件就做好了。我们在电脑上能看到如下图的文件。



Step 7：使用选项卡控件

使用选项卡控件前，我们至少先要有一个 GUI 窗体。

```
LOCAL gui IS GUI(400). //一个400像素宽度的 GUI 窗体
```

使用选项卡控件前，还要先用下面的命令来调用库函数。

```
//RunOncePath 不会重复运行已经运行过的程序，可以用来避免重复调用库函数
RunOncePath ("TabWidget/tabwidget").
```

然后就可以进行选项卡的操作了，可共使用的操作如下

函数	参数	返回值	说明
AddTabWidget (gui)	gui - GUI 结构	框体Box	在 gui 下添加一个选项卡控件 (tabwidget)
AddTab (tabwidget, TabTitle)	tabwidget - 框体Box TabTitle - 字符串	框体Box	在选项卡控件 (tabwidget) 下添加一个选项卡 (Tab)
ChooseTab (tabwidget, n)	tabwidget - 框体Box n - 整数	无	在选项卡控件 (tabwidget) 中切换到第 n 个选项卡显示

好了，我们来实际操作一下：

先在 kOS 部件的指令窗口里键入以下命令，以切换到 Volume 0。

```
switch to 0.
```

然后新开一个程序，里边放下面的代码，然后运行。

```
LOCAL gui IS GUI (400). //一个400像素宽度的 GUI 窗体
//RunOncePath 不会重复运行已经运行过的程序，可以用来避免重复调用库函数
RunOncePath ("TabIndex/tabwidget").

Set tabs to AddTabWidget(gui). //新建一个选项卡控件
//一连添加三个选项卡
Set tab1 to AddTab (tabs,"First").
Set tab2 to AddTab (tabs,"Second").
Set tab3 to AddTab (tabs,"Third").

//每个选项卡都是一个框体Box，按照Box的用法随意添加一些东西
Set t11 to tab1:AddLabel("wwwwwww"). Set t12 to tab1:AddTextField("wwwwwww").
Set t21 to tab2:AddLabel("2333333"). Set t22 to tab2:AddTextField("2333333").
Set t31 to tab3:AddLabel("emmmmmm"). Set t32 to tab3:AddTextField("emmmmmm").

gui:show() //都搞完之后还要让这个 GUI 窗体显示出来。

Wait until false. //不让程序结束
```

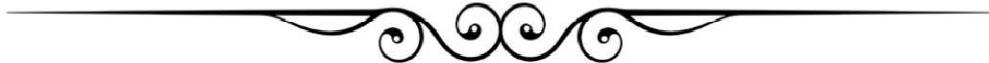
好的下面是我们实际运行的结果。可以看到选项卡在正常工作。



5. kOS 与 KSP

5.1 系统预设变量

----- 点击以返回目录 -----



本节介绍 kOS 中的预设变量/结构体。

他们已经在 kOS 里绑定了定义和类型，其值和 KSP 的游戏元素挂钩，对这些值的修改等同于在游戏里进行控制操作。

这些变量/结构体的值都是玩家能在 kOS 编程里用上的，玩家可从中一窥 kOS 的能力范围。

玩家在声明和使用变量的时候应该避开这些变量名，否则会出错。

(1) 载具和天体

变量	类型	读写性	说明
Ship	Vessel	只读	当前载具 (当前 kOS 程序所在的载具)
Target	Vessel 或 Body 或Part	读写	被设为目标的对象，为此变量赋值等同于在游戏里设定 Target
HasTarget	Boolean	只读	用来检查是不是设了 Target
星球名	Body	只读	Kerbin, Mum 之类的
.....			

例如：

```

Ship //kOS程序的当前载具
Target //目标载具/目标部件/目标星球，具体根据游戏里目标设的什么而定
Vessel("Untitled Space Craft") //载具 Untitled Space Craft

Kerbin //坎星
Body("Kerbin") //坎星（另一种写法）

```

(2) Ship 的快捷变量

Ship 表示当前 kOS 程序所在的载具，是一个很常用的预设变量。

kOS 为 Ship 下的常用变量额外准备了单独的变量名，使用这些变量就等同于使用 Ship 下对应的成员值。

变量	等同于变量名	类型	读写性
Heading	Ship : Heading	Scalar	只读
	当前载具朝向的罗盘指向，值为正北方开始的顺时针转角 范围 0~360，例如 0 表示北，90 表示西，225 表示正东南		
Prograde	Ship : Prograde	Direction	只读
	当前载具的速度方向的朝向		
Retrograde	Ship : Retrograde	Direction	只读
	当前载具的速度方向的反方向的朝向		
Facing	Ship : Facing	Direction	只读

	当前载具的朝向		
Maxthrust	Ship : MaxThrust	Scalar	只读
	当前载具的最大推力 (所有活动引擎的推力总和, 以最大节流阀、最大推力限制计算, 不考虑引擎之间方向关系)		
Velocity	Ship : Velocity	OrbitableVelocity	只读
	当前载具的速度		
Geoposition	Ship : Geoposition	GeoCoordinates	只读
	当前载具的经纬坐标		
Latitude	Ship : Latitude	Scalar	只读
	当前载具纬度坐标, 单位: °		
Longitude	Ship : Longitude	Scalar	只读
	当前载具纬度坐标, 单位: °		
Up	Ship : Up	Direction	只读
	当前载具头顶方向的朝向		
North	Ship : North	Direction	只读
	当前载具正北方向的朝向		
Body	Ship : Body	Body	只读
	当前载具所在星球		
AngularMomentum	Ship : AngularMomentum	Vector	只读
	当前载具的角动量, 单位: t * m^2 / s, (t = 吨)		
AngularVel	Ship : AngularVel	Vector	只读
	所在星球的角速度, 单位: rad / s		
AngularVelocity	Ship : AngularVel	Vector	只读
	所在星球的角速度, 单位: rad / s		
Mass	Ship : Mass	Scalar	只读
	当前载具的质量, 单位: t		
VerticalSpeed	Ship : VerticalSpeed	Scalar	只读
	当前载具的垂直速度, 单位: m / s		
GroundSpeed	Ship : GroundSpeed	Scalar	只读
	当前载具相对于地表 / 空气的速度, 单位: m / s		
Airspeed	Ship : AirSpeed	Scalar	只读
	当前载具相对于地表 / 空气的速度, 单位: m / s		
Altitude	Ship : Altitude	Scalar	只读
	当前载具的海拔高度, 单位: m		
Apoapsis	Ship : Apoapsis	Scalar	只读
	当前载具所在轨道的远地点海拔高度, 单位: m		
Periapsis	Ship : Periapsis	Scalar	只读
	当前载具所在轨道的近地点海拔高度, 单位: m		
Sensors	Ship : Sensors	VesselSensors	只读
	当前载具所搭载的传感器数据		
SrfPrograde	Ship : SrfPrograde	Direction	只读
	当前载具的地表速度的方向		
SrfRerograde	Ship : SrfRerograde	Direction	只读
	当前载具的地表速度的方向		
Obt	Ship : Obt	Orbit	只读
	当前载具的所处轨道		
Status	Ship : Status	String	只读

	当前载具的状态		
Shipname	Ship : Name	String	读写
	当前载具的载具名称		
.....			

(3) 预设列表

为了方便调用, kOS 提供了以下的预设列表。

这些预设列表不能直接使用, 需要先以类似 `list Bodies in B.` 的指令,
把这些列表存到列表变量B, 然后对B进行操作。

变量	类型	说明
Bodies	List of Body	天体的列表 (不包含小行星)
Targets	List of	可设为目标的载具列表 (包含小行星)
Fonts	List of String	可在 <code>Style : Font</code> 或 <code>Skin : Font</code> 中 <small>样式</small> 使用的字体的名称
Processors	List of Processor	当前载具 kOS 处理器 的列表
AggregateResources	list of AggregateResource	当前载具上资源总量的列表
Parts	List of Part	当前载具上部件的列表
Engines	List of Engine	当前载具上引擎类部件的列表
Seners	List of Sener	当前载具上传感器类部件的列表
Elements	List of Element	当前载具上对接元素的列表
DockingPorts	List of DockingPort	当前载具上对接口类部件的列表
Files	List of File	当前 卷Volume 上 文件File 的列表
Volumes	List of Volume	当前载具上可访问 卷Volume 的列表
.....		

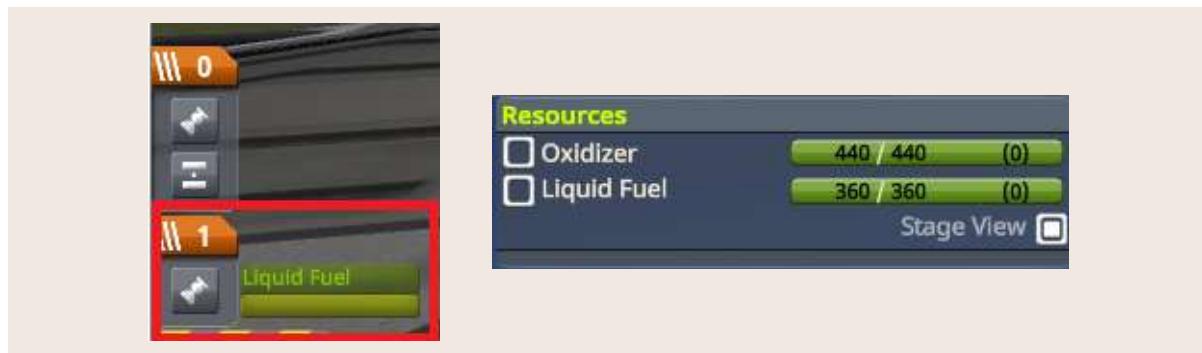
(4) 数学/物理常数

常数	写法	类型	说明
G	Constant : G	Scalar	万有引力常数 $6.67384 \times 10^{-11} \text{ N} \cdot \text{m}^2 / \text{kg}^2$
e	Constant : E	Scalar	自然对数的底 2.718281828459
π	Constant : Pi	Scalar	圆周率 3.141592654
c	Constant : C	Scalar	真空中光速 $299,792,458 \text{ m/s}$
AtmToPa	Constant : AtmToPa	Scalar	气压从 Atm 单位换算到 KPa 单位的系数 101.325
PaToAtm	Constant : PaToAtm	Scalar	气压从 KPa 单位换算到 Atm 单位的系数 0.00986923266716013
DegToRad	Constant : DegToRad	Scalar	从角度换算到弧度的系数 0.0174532925199433
RadToDeg	Constant : RadToDeg	Scalar	从弧度换算到角度的系数 57.2957795130823
.....

(5) 命令窗口、内核、KSC数据库、发射序列、变轨点

变量	类型	说明
Terminal	Terminal	命令窗口对应的结构体，主要为外观设置
Core	Core	运行当前 kOS 程序的内核信息
Archive	Volume	KSC 的公用数据库，对应 \Ships\Script 文件夹
Stage	Stage	发射序列中的当前信息
NextNode	ManeuverNode	下一个变轨点，如果没有设定变轨点则会出错
HasNode	Boolean	检查是否有变轨点，用来规避 NextNode 出错的
AllNodes	List of ManeuverNode	返回当前载具所有变轨点的列表
.....

*注1：Stage 对应的是发射序列里已经被激活的那一级（下左图红框处）。在 Stage 里能查询到这一级的可用资源的信息。（下右图，勾选 Stage View 状态下）



(6) 资源类型

kOS 提供有各种资源种类和信息以供查询，对应游戏中资源列表的内容（如下图）。



例子：

```
Print Ship:LiquidFuel. //显示当前载具上 所有液体燃料的数量
Print Stage:LiquidFuel. //显示发射序列里 当前级的 所有液体燃料的数量
Print Ship:LiquidFuel. //显示目标载具上 所有液体燃料的数量
```

*注1：只要把资源名拼写正确就能查询到，就算是其他MOD里新定义的资源也OK。

(7) 轨道飞行常用信息的快捷变量

KOS 将一些轨道飞行时常用的参考数据整理成了数值实时更新的变量。

这些变量里前几个都打包成了 ALT 结构体 和 ETA 结构体，以方便使用。

变量	类型	读写性	单位	说明
ALT : Apoapsis	Scalar	只读	m	当前载具所在轨道的远地点海拔高度
ALT : Periapsis	Scalar	只读	m	当前载具所在轨道的近地点海拔高度
ALT : Radar	Scalar	只读	m	当前载具的离地高度
ETA : Apoapsis	Scalar	只读	s	当前载具距离下一个远地点的时间
ETA : Periapsis	Scalar	只读	s	当前载具距离下一个远地点的时间
ETA : Transition	Scalar	只读	s	当前载具距离进入新轨道的时间，和 引力逃逸、引力捕获、变轨点有关
Encounter	-----	-----	-----	如果当前载具航线上会发生引力捕获 / 逃逸，则返回新轨道 Orbit 结构体 如果当前载具航线上不会发生引力捕 获 / 引力逃逸，则返回字符串 “None” [无]
.....

(8) 载具功能开关与地图视角

KSP 中有关飞船功能开关的要素，KOS 将他们做成了逻辑变量，并增加了一些自己的要素。玩家可以使用 Toggle、on、off 命令进行控制。

使用 Set 赋值语句也能起到相同效果，但是代码写起来太麻烦了所以这样设。

```

toggle AG1. //打开/关闭动作组AG1, 相当于按了一下按键1
toggle AG3. //打开/关闭动作组AG3, 相当于按了一下按键1
toggle AG50. //打开/关闭动作组AG50, 如果装了ActionGroupsExtended这个Mod的话才有效
toggle SAS. //打开/关闭SAS, 相当于按了一下按键T
toggle RCS. //打开/关闭RCS, 相当于按了一下按键R
SAS on. //打开SAS, 相当于如果SAS没有开启, 就按按键T
RCS off. //关闭RCS, 相当于如果RCS没有关闭, 就按按键R
Light off. //关所有灯, 相当于如果没开灯, 就按按键L
Brakes on. //刹所有能刹车的部件相当于如果没刹车, 就按按键B
Gear off. //拉起所有起落架和登陆架, 相当于按按键G
Legs on. //放下所有着陆架, 相当于按按键G, 不过只作用于登陆架
Chutes on. //展开所有降落伞(这条只能搭配on命令)
ChutesSafe on. //展开所有符合安全展开条件的降落伞(这条只能搭配on命令)
Panels on. //展开所有太阳能板
Radiators on. //展开所有天线
Ladders off. //放下所有着陆架
Bays on. //打开所有货仓
DeployDrills on. //展开所有钻头
Drills on. //所有钻头开始工作
FuelCells off. //停止所有燃料电池的工作
ISRU on. //启动所有资源转换器
Intakes off. //关闭进气道
ABORT on. //按下任务终止按钮

```

kOS 还将 M 键切换地图的功能做成了逻辑变量 MapView。

```

MapView on. //从载具视角进入地图视角
MapView off. //退出地图视角, 回到载具视角

```

*注1: 当使用 Toggle、on、off 命令对普通变量进行操作时, 如果该变量不是逻辑值, 会被强行变成逻辑值再进行运算。

(9) 航向控制

((1)) 航向控制 (快捷)

kOS 提供4种有关方便快捷地控制航向的变量, 他们必须使用 Lock 命令进行赋值。

变量	类型	读写性	说明
Throttle	Scalar	读写	节流阀, 范围0~1
Steering	Direction	读写	载具朝向。赋的值必须是 Direction 或 Vector
WheelThrottle	Scalar	读写	车轮的出力, 范围 -1~1. 负值是指倒车
WheelSteering	Scalar	读写	车轮转动朝向。此变量为罗盘指向, 定义为正北方开始的顺时针转角, 范围 0~360, 例如 0 表示北, 90 表示西, 225 表示正东南 所赋的值需为 GetCoordinates、Vessel、或 Scalar
.....			
说明: 以上航向控制变量必须使用 Lock 命令进行赋值			



((2)) 航向控制（精细）

kOS 还提供别航向控制（快捷）更精细的控制，包括：

- 载具转向时的力度（对应游戏中的 WASDQE 键）、
- 载具转向力度的偏移量（对应游戏中的 Alt+WASDQE 键）、
- 载具RCS平移时的力度（对应游戏中的IKJLHN 键）。

这些控制用的变量都打包在了 Ship 结构体下的 Control(底层控制)结构体里。



((3)) 航向控制（玩家输入）

kOS 还提供玩家键盘驾驶输入的接口，允许玩家对键盘上的驾驶操作进行监控和再编程。

这些变量都打包在了 Ship 结构体下的 Control(玩家输入) 结构体里。

(10) 时间信息

变量	类型	读写性	说明
MissionTime	Scalar	只读	从任务开始到当前时间的秒数
Time	TimeSpan	只读	时间结构体，包含当前时间的各种形式的表述
Time : Seconds	Scalar	只读	游戏里物理模拟的总时间秒数，常用的时间值
SessionTime	Scalar	只读	从物理载入到当前时间的秒数
.....			
说明： Time : Seconds 其本身的值无意义，使用时要采集两个不同时间的值并求差			

(11) 系统变量

用来返回 kOS 版本号的 Version 结构体。

```

PRINT VERSION.          // Returns operating system version number. e.g. 0.1.2.3
PRINT VERSION:MAJOR.    // Returns major version number. e.g. 0 if version is 0.1.2.3
PRINT VERSION:MINOR.    // Returns minor version number. e.g. 1 if version is 0.1.2.3
PRINT VERSION:PATCH.    // Returns patch version number. e.g. 2 if version is 0.1.2.3
PRINT VERSION:BUILD.    // Returns build version number. e.g. 3 if version is 0.1.2.3

```

用来读取和修改 kOS 设置的 Config 结构体。

Config 结构体对应的是 kOS 设置菜单里的内容



(12) 载具控制信号

有关载具通讯信号的问题，kOS 里可以让玩家选择三种方案：

- ((1)) 所有载具时刻都有通讯信号
- ((2)) 使用 KSP 原版的通讯信号规则
- ((3)) 使用 通讯限制MOD 的通讯信号规则（需要安装该MOD）

变量	类型	读写性	说明
HomeConnection	Connection	只读	从 KSC 到当前载具的控制信号
ControlConnection	Connection	只读	从控制源到当前载具的控制信号
.....			
说明： ControlConnection 的控制源可以是 KSC，也可以是附近的载人载具			

(13) 第四墙 Kuniverse

kOS 提供了 Kuniverse 结构体，Kuniverse 译作“第四墙”，指戏剧中用来分割镜框式舞台（剧内）与观众席（剧外）的平面。

Kuniverse 结构体 主要对应游戏暂停菜单里的操作，他能做到：

- ((1)) 返回发射台重新发射
- ((2)) 即时 SL，相当于按 F5 和 F9
- ((3)) 在不同载具之间切换，相当于按 [和]
- ((4)) 选择一个载具重新发射
- ((5)) 时间加速（Warp）

((6)) 等其他功能.....



(14) 时间加速控制

```
Set WARPmode TO "PHYSICS". //时间流速模式切换到物理加速（大气层中的那种）
Set WARPmode TO "RAILS". //时间流速模式切换到轨道加速（宇宙里的那种）
Set WARP to 5. //将时间流速设定到当前模式下的对应和倍数
```

kOS 还提供了 TimeWarp 结构体，整合了更强大的时间加速功能。

(15) 载入距离

KSP 为不同状态的载具设定了不同的载入距离（通常为 2.5 km），游戏只显示载入距离内的僚机，只让载入距离内的僚机参与物理演算。

kOS 为此特点提供了有关载入距离的结构体。

玩家甚至可以独立设置每个载具在每种状态下的载入距离。

(16) 程序报告 ProfileResult()

kOS 提供了 ProfileResult() 函数，用来监控 kOS 程序的运行。

该函数会返回一串字符串，描述了 kOS 程序运行的每一步花了多少时间。

使用 ProfileResult() 函数 前先要把 Config:STAT 设为 True。

使用 ProfileResult() 函数 会降低 kOS 的运行速度。

(17) 太阳本基准矢量 SolaPrimeVector

kOS 中的坐标系不考虑相对论效应，所有物理演算中的位置信息，都是在一套绝对静止的直角坐标系上进行。这套坐标系叫 **Raw-Raw 坐标系**，是惯性系。这个惯性系的基准就是太阳的本初子午线。

Raw-Raw 坐标系其实没啥实用性，有实用性的是另两个坐标系：Ship-Raw 和 SOI-Raw。

SOI-Raw 坐标系 以当前星球为中心，依照星球春分点朝向和北天极朝向决定的坐标系。坐标系不随星球自转，但是随星球平移，是一个惯性系。

Ship_Raw 坐标系 以当前载具质心为中心，和 SOI-Raw 保持相同的坐标轴朝向。坐标系随载具平动，但是不随载具转动，是一个非惯性系。

太阳本基准矢量 SolaPrimeVector 就是太阳本初子午线方向用矢量表示，然后在 Ship-Raw 坐标系下表示的值。

(18) MOD 扩展

kOS 具有很强的 MOD 适应性。

一般，不含 DLL 文件的 MOD 都能被 kOS 支持，kOS 都能进行该 MOD 的操作。

要是 MOD 里包含了 DLL 文件，则可能 kOS 无法进行该 MOD 的操作。

为了能控制一些原本不支持但很实用的 MOD 的操作，

kOS 专为几个 MOD 提供了拓展支持。他们包括：

- Action Groups Extended (动作组扩展)
- RemoteTech (通讯限制)
- Kerbal Alarm Clock (KAC闹钟)
- Infernal Robotics (转轴)
- DMagic Orbital Science (轨道科研)
- Trajectories (落点预测)

(19) 色彩

kOS 支持一定的图形绘制、部件高亮显示。

为此 kOS 提供了色彩结构体。

5.2

5.2 当前载具 (Ship)

----- 点击以返回目录 -----



需要明确一个重要概念。

本文中我们所说的“当前载具”、“自机”，
都是在指执行当前 kOS 程序的 kOS 部件所在的载具，也就是 Ship 变量。

与他相区别是“活动载具”这个概念，活动载具是指在游戏画面缩放中心的载具。

比如下图中，游戏画面聚焦的是右边的 Vessel_A 载具，高度表 67m 也是 Vessel_A 的海拔高度，这里 Vessel_A 就是活动载具。

而两个载具都有一个 kOS 部件，都能打开各自的命令窗口，并在里边运行程序。那么：

对于左边 Vessel_B 的命令窗口里的程序，Ship = 当前载具 = 自机 = Vessel_B 载具。

对于右边 Vessel_A 的命令窗口里的程序，Ship = 当前载具 = 自机 = Vessel_A 载具。



5.3

5.3 kOS 的工作特点

5.3.1

5.3.1 显示帧、物理帧、逐帧宇宙

===== 点击以返回目录 =====



在 KSP 中有两套帧数的概念，**物理帧**（Physics Ticks）和**显示帧**（Update Ticks）。

物理帧代表 KSP 进行物理演算间隔的物理帧，

对于 KSP 和 kOS 来说，一个物理帧就是宇宙运行的某一个瞬间的快照。

物理帧更新时，kOS 会根据上一个物理帧（快照），计算下一个物理帧时整个宇宙的状态。

这样**逐帧形式**，像是计算机求定积分数值解一样，模拟游戏中事物的变化。

KPS 游戏的**物理帧**会根据计算压力自动调节，性能好的电脑游戏里物理帧间隔就小，性能差的电脑游戏里物理帧间隔就大，但是有一个上限值（默认为 0.04 s），玩家可以在 Setting - General - MaxPhysics Delta-Time per Frame 变更此设置。如果物理帧间隔超过这个上限值，游戏就会卡到降低演算的时间流动速度。

而**显示帧**则和游戏的画面帧数一致。

显示帧默认为 120 fps，玩家可以在 Setting - Graphics - Frame Limit 里变更此设置。

kOS 为了做到实时控制，它是在**每两个物理帧的演算间隔里**进行计算的。

例如这一帧飞船的姿态方向偏了，kOS 可以马上命令 SAS 或 RCS 给出修正输出，飞船会在下一个物理帧执行这些操作。

5.3.2

5.3.2 耗电

===== 点击以返回目录 =====



kOS 部件需要电力来维持工作，一旦载具电量耗尽，kOS 就会没电关机，无法工作。

不仅 kOS 部件待机会消耗电量，kOS 程序每执行一条指令，都会额外消耗电量。

所幸的是，其实 kOS 部件的耗电量仍然是很小的，即使满负荷工作，

耗电速度也不会比一个车轮多。玩家只要注意别让 kOS 没电就行了。

5.3.3

5.3.3 触发器

===== 点击以返回目录 =====



((1)) 触发器的行为

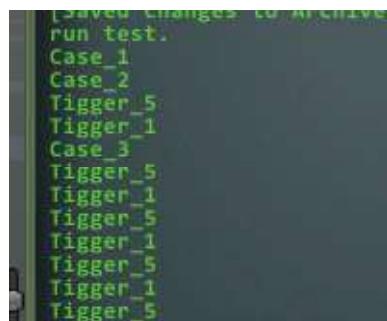
为了更好的载具控制，kOS 设计有触发器结构。

首先来看下面这样一个程序和运行结果。

```
//触发线
When true Then { //触发结构_1
    Print "Trigger_5".
    Wait 5. //等待5秒
    //返回 True值 表示触发结构不消失，下一个物理帧仍然有效
    Return true.
}

When true Then { //触发结构_2
    Print "Trigger_1".
    Wait 1. //等待1秒
    //返回 True值 表示触发结构不消失，下一个物理帧仍然有效
    Return true.
}

//主线
Print "Case_1".
Print "Case_2".
Wait 0.001. //等待下一个物理帧
Print "Case_3".
Wait Until false. //主线永不结束
```



kOS 的程序分为主线（Main-Line）和触发线（Triggers-Line）。

触发线有权中断主线，而主线不能中断触发线。

触发线中断主线之后，主线会挂起等待，直到触发线运行结束后，主线才会继续运行。

kOS中的触发器按触发模式可分为两类：**循环触发**和**单次触发**。

循环触发：

(1) Lock指令 + 航向控制 ()；

- (2) When触发指令 + Return True 语句;
- (3) On触发指令 + Return True 语句;
- (4) 具有持续轮询特点的其他结构体成员。

单次触发：

- (1) When触发指令；
- (2) On触发指令；
- (3) 具有单次触发特点的结构体成员（如 GUI 元素按钮的 OnClick 行为响应）。

((2)) 触发器的优先级

kOS 中的触发器有优先级先后的分别。优先级较高的触发器会如同中断主线一样中断优先级较低的触发器。

kOS 中的触发器按照优先级高低分的话如下：

优先级 30：

Lock 指令 + 航向控制变量（Throttle, Steering, WheelThrottle, WheelSteering）；

优先级 20：

When 触发指令，On 触发指令；

优先级 10：

具有单次触发特点的结构体成员（如 GUI 元素按钮的 OnClick 行为响应）；

优先级 0：

普通代码。

((3)) 解除触发器的优先级

在触发器里使用 DropPriority() 语句可以将当前触发器的优先级将为 0。

此举可用于允许其他低等级的触发器打断当前触发器。

DropPriority() .

((4)) 其他提示

最好不要在触发结构里使用 等待语句 Wait。

最好把触发结构设计为小计算量，可以在一个物理帧间隔内执行完毕的样子。

一般的，触发结构最好写在所有其他代码之前。

kOS 有物理帧之内最大执行指令数的限制（默认 200，在 kOS 设置菜单里设置），如果任意时刻执行的指令数已经足够多了，那么剩下还没执行的指令会顺延到下一个物理帧再执行。

5.3.4 等待

===== 点击以返回目录 =====



当 kOS 程序运行到 Wait 指令时，会立刻结束当前物理帧间隔的计算，并且开始计时。

在下一个物理帧间隔，kOS 会计算 Wait 指令的时间参数，并和计时时长作比较，如果计时时长>时间参数，则Wait指令结束，kOS 会继续执行后续语句。

Wait 指令也可以和 Until 关键词 联用，也可以和逻辑值搭配，具体请看如下例子：

```
//等待2两种的Wait指令
Wait 2.
Wait 1+1.

//会在第二个物理帧间隔就结束的 Wait指令
Wait 0.01.
Wait<0.
Wait True.
Wait Until True.

//永远都不会结束的 Wait指令
Wait Until False.
```

触发结构如果写在主线里的 Wait 语句面，那么直到这句 Wait 语句执行完之前，这个触发结构都不会被声明。

例如下面例子中，触发结构永远不会运行，声明都不会声明。

```
//这个程序最终结果只会显示 "Start"
Print "Start".
Wait Until False.

//这里之后的都不会被编译器执行到
Print "End".
When True Then {
    Print "Trigger".
}
```

5.4

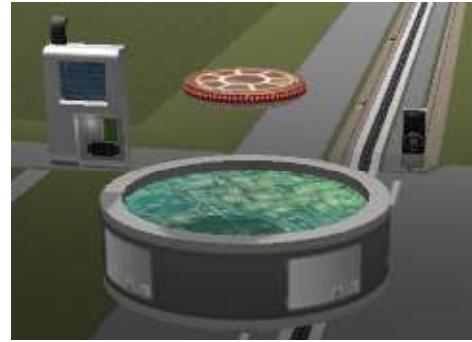
5.4 kOS 部件

===== 点击以返回目录 =====



(1) kOS 部件

安装完 kOS 之后，可以在车间里的 Control 面板 中找到如下图的四个 kOS 部件。他们的每一个都具有相同的 kOS 功能，区别只是模拟了不同时代的科技水平，体现在大小、质量、存储容量不同。



存储容量决定了玩家能在 kOS 部件上保存多少数据，
这些数据可以是 kOS 程序文件、数据文件、以及其他文件。

(2) CFG文件与kOS功能

每个 kOS 部件，在他的 CFG 文件中，都会有一段名为 KOSProcessor 的 Module 代码。比如下面那样。

```
MODULE
{
    name = kOSProcessor
    diskSpace = 5000
    ECPerBytePerSecond = 0
    ECPerInstruction = 0.000004
    # 注释：以下是可选项，以及他们的默认值
    # baseDiskSpace = 0
    # diskSpaceCostFactor = 0.0244140625
    # baseModuleCost = 0
    # diskSpaceMassFactor = 0.0000048829
    # baseModuleMass = 0
}
```

如果玩家想在一个部件里增加 kOS 功能，将这样的代码扔到部件的 CFG 文件 里就可以了。

下面介绍 Module 代码中每项字段的含义：

字段名	类型	默认值	说明
diskSpace	整数	1024	kOS 部件 右键菜单里会有 kOS DiskSpace 滑块来调节存储容量，滑块可以设置为 1 倍、2 倍、4 倍的 baseDiskSpace 值。 diskSpace 值则表示未操作滑块时的 存储容量
baseDiskSpace	整数	diskSpace 值	
ECPerInstruction	浮点数	0.000004	执行每条指令所要消耗的电量
ECPerBytePerSecond	浮点数	0.0	每个 Byte 存储容量每秒钟消耗的电量
diskSpaceCostFactor	浮点数	0.0244140625	每追加 1 Byte 存储容量所增加的造价
baseModuleCost	浮点数	0.0	kOS 功能 所对应的造价
diskSpaceMassFactor	浮点数	0.0000048829	每追加 1 Byte 存储容量所增加的质量
baseModuleMass	浮点数	0.0	kOS 功能 所对应的重量
.....



5.5

5.5 窗口设置

5.5.1

5.5.1 kOS 控制面板

===== 点击以返回目录 =====



和很多其他 MOD 一样，kOS 也有自己的控制面板。如下图。

控制面板里左边列表会显示物理加载范围内的所有带有 kOS 部件的载具，
每个部件右边都有个电脑图标，鼠标指向某一个电脑图标，
对应的 kOS 部件就会在游戏画面里高亮显示。

点击电脑图标可以打开这个部件（机载计算机）对应的指令窗口。



在相应 kOS 部件的右键菜单里点击 Open Terminal 也可打开指令窗口

5.5.2

5.5.2 kOS 难度设置

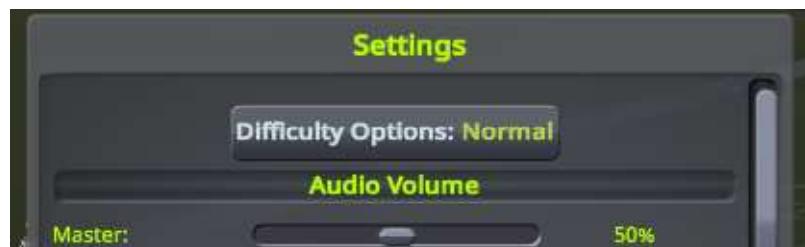
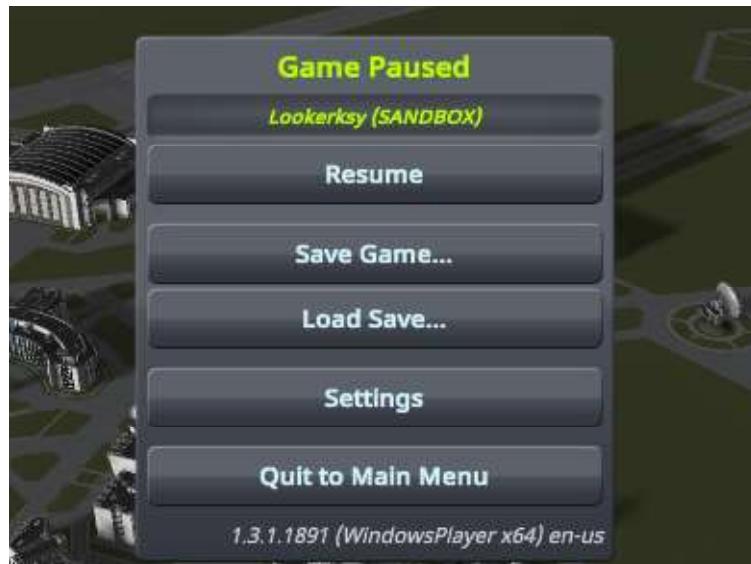
===== 点击以返回目录 =====



kOS 的设置菜单要在游戏难度设置菜单里找。

虽然是在难度设置菜单里，但设置的东西和 kOS 游戏难度无关。

以下为 kOS 设置菜单的进入方式。





上图选项依次为：

Instructions Per Update – 每个物理帧运行的最大命令数

Enable Compressed Persistence – 使用Base64算法压缩kOS部件中的文件

Show Statistics – 显示kOS程序的运行信息

Start On Archive – 载具发射时默认的存储空间是在KSC还是kOS部件

Obey Hide UI – 是否随KSP的UI隐藏键（F2）一起隐藏

Enable Safe Mode – 运行kOS程序时任何空值或者溢出值都会被报错

Audible Exceptions – 时报错会有蜂鸣声

Verbose Exceptions – 报错时会有详细说明

Only use Blizzy toolbar – 只在Blizzy工具栏MOD中显示kOS按钮

kOS指令窗口默认亮度

Debug each opcode – （针对Mod开发者）kOS的每一步运行都会生成报告，巨卡

另外右边的滑块是通讯规则的设置：

PermitAllConnectivityManager – 此选项代表不作任何通讯限制

CommNetConnectivityManager – 此选项代表使用 KSP 原版的通讯规则

如果玩家装了 通讯MOD，还会有第三个选项，对应 通讯MOD 的通讯规则。

kOS设置菜单里的设置均被打包至 **Config** 结构体，玩家可访问此结构体直接进行设置。

5.6

5.6 自定义 GUI 界面

===== 点击以返回目录 =====

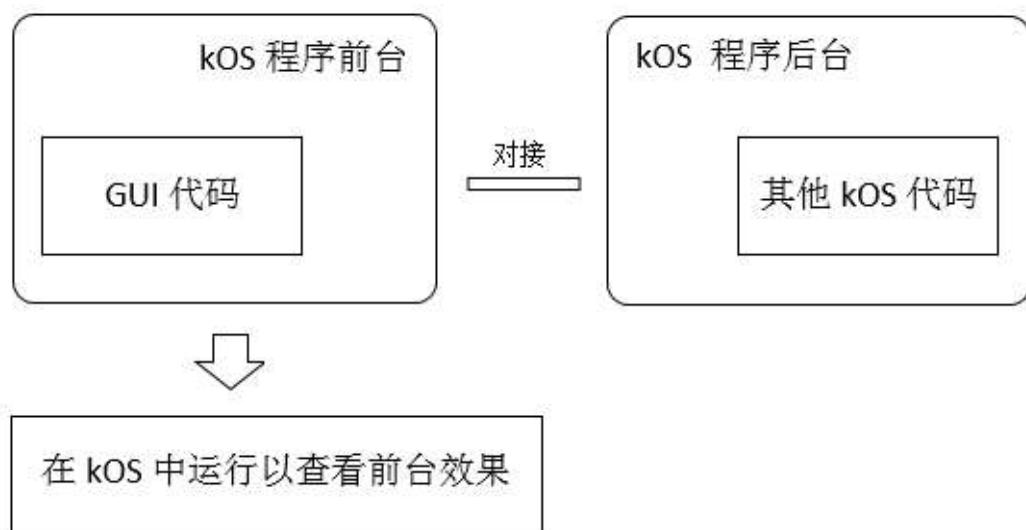


kOS 有个很有趣/有用的功能叫 GUI，能让玩家创造自己的对话窗口/控制面板，如下图：



kOS 只支持用代码进行这样的前台窗体设计。

kOS 提供了一套 GUI 相关的结构体和函数，用于设计前台窗体，
就像普通编程里的前台代码一样，我们管他叫 GUI 代码。



kOS 中所有的前台元素以控件（Widget）形式存在的，kOS 专门为这设计了一套结构体。
玩家可以用他们来设计属于自己的操作面板，就像在自创 MOD 一样。

	控件（结构体）	名称	说明
排版相关	Box	框体	用于其他控件的组合布局和排版，可作为选项卡的下属内容
	ScrollBar	滚动框体	一种特殊的框体（Box），可滚动以显示更多内容
	GUI	GUI	GUI 设计的起点，先要创建一个 GUI 结构体，之后才能在里边添加各种其他控件，GUI 对应窗体，一种特殊的框体（Box）
内容相关	Label	标注	可以显示文字或图片，没有交互功能
	TextField	文本框	一种特殊的标注（Label），用户可以编辑其中的文字
	Button	按钮	一种特殊的标注（Label），可以被按下以执行某种操作
	PopupMenu	下拉菜单	一种特殊的按钮（Button），带有一个列表给人从中选择
	Slider	滑块	可以通过拖拉来进行数值赋值
	Spacing	空档	用于排版时进行留白
样式相关	Style	样式	用于确定某一个控件的风格
	Skin	皮肤	一套样式（Style）的集合，用于确定GUI内所有控件的风格
	StyleRectOffset	样式边距	样式（Style）的子内容，用于确定控件的排版边距
	StyleState	样式状态	样式（Style）的子内容，用于确定控件显示方面的某些属性

有关 GUI 的详细说明，请见 GUI 部件

5.7

5.7 文件和卷

5.7.1

5.7.1 脚本文件

===== 点击以返回目录 =====



每个 kOS 程序 都有一个对应的脚本文件。

我们管这个脚本文件叫程序、源文件、代码文件，之类的称呼都可以。

脚本文件是一种文件，文件被存储在 卷Volume 里。

卷Volume 又叫盘符，他的存储容量限制了文件的大小。

5.7.2

5.7.2 文件存储位置

===== 点击以返回目录 =====

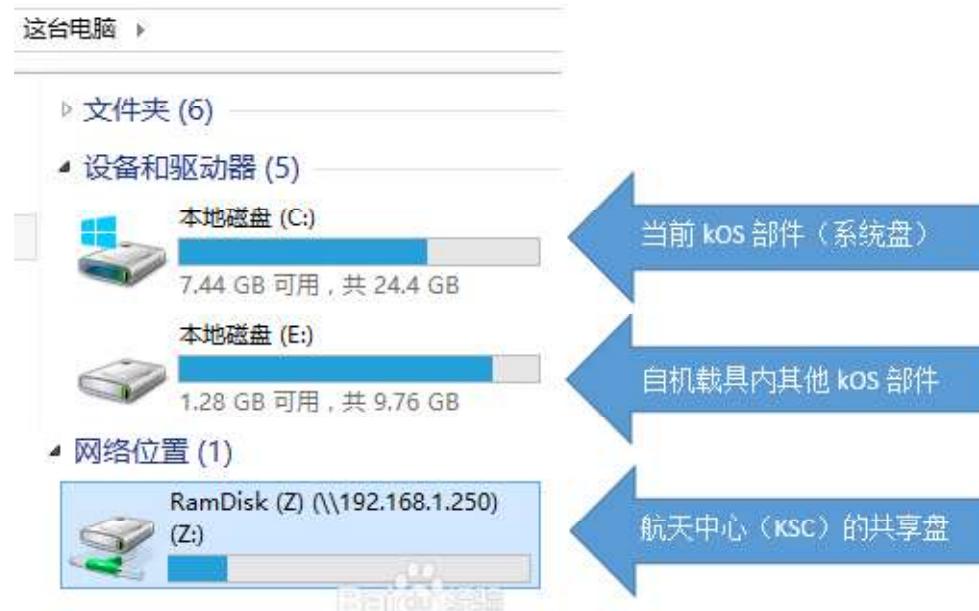


在真实世界中，

电脑文件或者保存在电脑自己有限容量的本地硬盘里，

或者保存在能随意拓展容量的云端里。

电脑如果坏了，硬盘里文件就没了，但是云端（网络硬盘）里的还有。



类似的，在 KSP 中，

文件或者保存在 kOS 部件（机载计算机）自己本地有限容量的卷 Volume 里，

或者保存在位于 KSC 航天中心 无限容量的卷 Volume 0 里。

kOS 部件如果损毁了，自己本地卷 Volume 里的文件就没了，但是 KSC 数据中心的还有。

文件存储位置	本地卷	云端	
	自机机载电脑的文件	航天中心电脑的文件	僚机机载电脑的文件
Volume 编号	1, 2, 3,	0	
Volume 名称	未设定	Archive	
可访问性	可访问	与航天中心 (KSC) 有通讯信号时可访问	不可访问
游戏中位置	自机的某个 kOS 部件 的存储空间内	航天中心 (KSC) 的数据库中	僚机的某个 kOS 部件 的存储空间内
文件大小	不超过 kOS 部件的 存储容量	无限制 (因为 KSC 的 数据库空间无限)	不超过 kOS 部件的 存储容量
模拟真实的	载具机载计算机的硬盘	航天中心的数据库	载具机载计算机的硬盘
在玩家电脑 中的位置	KSP.exe 的运行内存	KSP 游戏安装目录下的 Ships / Script 文件夹	KSP.exe 的运行内存
寿命	当 kOS 部件损毁时 文件消灭	无限 (即使关闭游戏 文件也依然存在)	当 kOS 部件损毁时 文件消灭
例子：	载人舱在 Mun 背面无通讯 信号时的自动着陆程序	记录火箭发射中空气阻力 并导出成 csv 的程序，之 后要用 Excel 绘制其曲线	导弹的飞控程序

在 KSC 的设置菜单里边，整合有 kOS 的设置，里面直接有一项可以让你选择，

默认 Volume 是要设成 Volume 1 (当前 kOS 部件)，还是要设成 Volume 0 (Archive)。



5.7.3

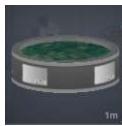
5.7.3 扩展 kOS 部件的存储容量

===== 点击以返回目录 =====

不同 kOS 部件的存储容量是不同的，他们有不同的基本的存储容量。
不过玩家可在部件的右键菜单的 kOS Disk Space 里选择将其拓展至原来的2倍或4倍，
代价是额外的造价、重量、耗电量。



下面是我们对不同 kOS 部件扩展前后的状态做的一个对比，
我们将部件按尺寸从大到小进行排列，并将 1x, 2x, 4x 三种选择命名为 基本、中配、高配

 CX - 4181	基本	中配	高配
质量 (t)	0.12	0.17	0.27
存储容量 (MByte)	10	20	40
每秒存储耗电 (EC / MByte)	0		
运行耗电 (EC / 200 条指令)	0.04		
造价	1200	1444	1932
质量 / 存储容量 (t / Mbyte)	0.012	0.0085	0.00675
造价 / 存储容量 (/ Mbyte)	120	72.2	48.3
.....			

 KR - 2402 b	基本	中配	高配
质量 (t)	0.08	0.1	0.15
存储容量 (MByte)	5	10	20
每秒存储耗电 (EC / MByte)	0		
运行耗电 (EC / 200 条指令)	0.04		
造价	1200	1322	1566
质量 / 存储容量 (t / Mbyte)	0.016	0.01	0.0075
造价 / 存储容量 (/ Mbyte)	240	132.2	78.3
.....			

 CompoMax	基本	中配	高配
质量 (t)	0.03	0.32	0.91
存储容量 (MByte)	60	120	240
每秒存储耗电 (EC / MByte)	0		
运行耗电 (EC / 200 条指令)	0.04		
造价	2200	3665	6595
质量 / 存储容量 (t / Mbyte)	0.0004	0.0027	0.0038
造价 / 存储容量 (/ Mbyte)	36.7	30.5	27.5
.....			

 KAL9000	基本	中配	高配
质量 (t)	0.0005	1.2505	3.7405
存储容量 (MByte)	25.5	51	102
每秒存储耗电 (EC / MByte)	0		
运行耗电 (EC / 200 条指令)	0.04		
造价	1200	7426	19877
质量 / 存储容量 (t / Mbyte)	0.00002	0.025	0.037
造价 / 存储容量 (/ Mbyte)	47.06	145.6	194.9
.....			

从上表中可以看出，增加存储容量时单位Byte对应的质量和造价有问题，
kOS 作者给代表不同科技水平的不同 kOS 部件设置了相同的因子。

所以根据我们算出来的结果，

对于质量较大的 **CX - 4181** 和 **KR-2402b** 这两个部件，增加存储容量是合算的；
对于质量较小的 **CompoMax** 和 **KAL9000** 这两个部件，增加存储容量完全不合算，
还不如再装一个新的。



5.7.4

5.7.4 单个载具搭载多个 kOS 部件

===== 点击以返回目录 =====



每个 kOS 部件就相当于一台机载计算机。

同一个载具上可以放置多个 kOS 部件，这种做法可以用于需要分离对接的载具。

另外，同一个载具上的多个 kOS 部件可以互相访问彼此的卷 Volume。

kOS 以编号来指代同一载具上的不同 kOS 部件，
但是要注意，同一个 Volume 在不同 kOS 部件中的编号不一样，
自己本地 Volume 的编号总是为 1。

例如，假设 载具 X 上有两个 kOS 部件，分别叫 A 和 B。

那么，

在 A 的指令系统里，Volume 0 就是指 KSC 航天中心里的云端存储空间，
Volume 1 就是指 A 的存储空间，Volume 2 就是指 B 的存储空间。

在 B 的指令系统里，Volume 0 就是指 KSC 航天中心里的云端存储空间，
Volume 1 就是指 B 的存储空间，Volume 2 就是指 A 的存储空间。

5.7.5

5.7.5 数据中心(Archive)

===== 点击以返回目录 =====



数据中心(Archive)是位于航天中心(KSC)的一个特殊的卷Volume，Volume 编号为 0。

不同于那些依存于 kOS 部件的卷Volume，数据中心(Archive)有如下特点：

1. 只要有到航天中心(KSC)的通讯信号，就能访问 数据中心(Archive)。
2. 其本体是游戏目录/Ships/Script 下的内容，不依存于具体的部件或载具，甚至不依存于游戏程序。即使载具坠毁、重飞，或者读档、游戏结束、游戏崩溃，也不会影响 数据中心(Archive)下的内容。
3. 无限存储空间。只要玩家的电脑硬盘有多大，数据中心(Archive)就能有多大。
4. 可以在 KSP 游戏外，直接在电脑里使用文本编辑软件编辑 数据中心(Archive) 中的程序。

5.7.6

5.7.6 自启动文件夹

===== 点击以返回目录 =====



kOS 允许载具在被物理加载时自启动程序。每个载具都可以设置一个自启动程序。

在游戏目录下的Ships\Script\boot文件夹内的就是自启动程序的存放位置，在里边的kOS程序文件都可以设为载具的自启动程序。



如果 Boot 文件夹内有 kOS 程序文件，则在车间里 kOS 部件的右键菜单里，会多出自启动程序的设置选项。



实际效果如下图，点击 Launch 后刚上发射台，我们还没做任何操作，SAS就被点亮了。



*注1：程序自启动的条件是被物理加载时，也就是说不止上发射台，因为自机移动而距离自己足够近的其它载具，也有程序自启动的效果。

*注2：有时自启动程序开始运行时，物理加载还没有全部完成。
所以在自启动程序里最好是先等待几秒钟，然后再执行具体操作。

5.8

5.8 机器语言

===== 点击以返回目录 =====



(1) ksm 文件

和其他编程系统一样，kOS 中除了有源代码(文本文件)，还会有可执行文件(机器语言文件)。这个机器语言的文件叫 .ksm 文件，只不过他不是经常要用到。

kOS 可以直接运行源码文件，
也可以先把源码文件编译成 .ksm 机器语言文件保存在 Volume 上，然后再执行 .ksm 文件。

kOS 直接运行源码文件时，其实也会有个编译过程，只不过不会在 Volume 上保存对应的 .ksm 文件。

(2) 使用 ksm 文件

玩家需用 Compile 指令进行 .ksm 文件的编译

```
Compile Path1. //将 路径Path1 的源文件 编译成同文件夹下同名的 .ksm 文件
Compile Path1 to Path2. //将 路径Path1 的源文件 编译成 路径Path2 的 .ksm 文件
```

//说明：

```
//Path1 和 Path2 都是 String 字符串格式,
//其内容可以是 "MyProgram01" 或是 "MyProgram01.txt" 这样的纯文件名字符串
//也可以是 "0:/MyProgram01" 或是 "0:/abc/MyProgram01" 这样复合了路径的字符串
//另外，上面的 "0://" 表示 Volume 0 下的路径。同理 Volume 1 下的则表示为"1://"
```

玩家可以使用 RunPath 函数 来运行编译好的 .ksm 文件。例如下面：

```
Compile abc to "def.ksm". //将 Volume 根目录下的 abc文件 编译成 def.ksm文件
Run def. //运行这个 def.ksm 文件
RunPath("def.ksm"). //和上一句效果一样
```

*注1： Run指令 和 RunPath函数 都能用于运行程序。不过后者功能更强大。

Run指令 只能运行 Volume 根目录下的文件， RunPath函数 则能运行子文件里的文件。

*注2： 当玩家在 Run指令 后写了没有后缀名的文件名时，（例如： Run qwert.）

kOS 会先试图查找 qwert.ksm 去运行，

如果没有 qwert.ksm 文件， kOS 才会 查找 qwert 文件 并运行，

如果 qwert 文件 也没有， kOS 最后才会查找 qwert.ks 并运行。

.ks 文件是 kOS 中源代码文件的默认后缀名，不过 kOS 支持直接运行其他后缀名甚至是无后缀名的源代码文件，所以 .ks 文件的存在高也不高。

(3) ksm 文件的优缺点

.ksm 文件有如下优点：

1. 运行时不需要再编译，工作响应快。

编译会消耗时间，编译过程中 KSP 中时间还是流动的。

.ksm 文件运行时能立即投入工作。但源文件运行时要先等编译工作完成。

2. .ksm 文件不含代码注释，体积可能更小。

.ksm 文件不一定体积就比源文件更小。

对于有大量注释文本的源文件，.ksm 文件 体积更小，时占用空间小。

如果源文件里没有注释文本，一般来说编译后的 .ksm 文件 是比源文件大的。

.ksm 文件有如下缺点：

1. .ksm 文件不可阅读，不可编辑。

2. .ksm 文件报错时不会提示报错代码信息。

.ksm 文件已经舍弃了代码，运行时如果出错，只会提供代码行号，

不会显示出错的代码是什么。

而在直接运行源文件时，如果遇到出错，报错信息里会有行号和代码，以及出错类型的具体提示。

*注1：kOS 中其实没啥地方必须要用到 .ksm 文件。
这个特性设定的目的，更多的是便于玩家理解 kOS 的工作方式。

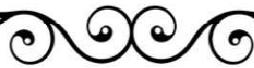
5.9

5.9 音频演奏（SKID）

5.9.1

5.9.1 简明用法示例

===== 点击以返回目录 =====



本小节是音频演奏（SKID）的简明用法示例。

```
Set n1 to Note(400,2). //创建音符 n1，频率 400Hz，音符持续 2 秒钟
Set n2 to Note(0,0.5). //创建音符 n2，无声音，音符持续 0.5 秒钟
Set n3 to Note(500,1). //创建音符 n2，频率 500Hz，音符持续 1 秒钟
Set s to List(n1,n2,n3). //将三个音符排成音符序列 s

Set V0 to GetVoice(0). //取第0音频轨道
V0:Play(s). //使用第01音频轨道演奏音符序列 s
```

5.9.2

5.9.2 kOS 中音频的构成

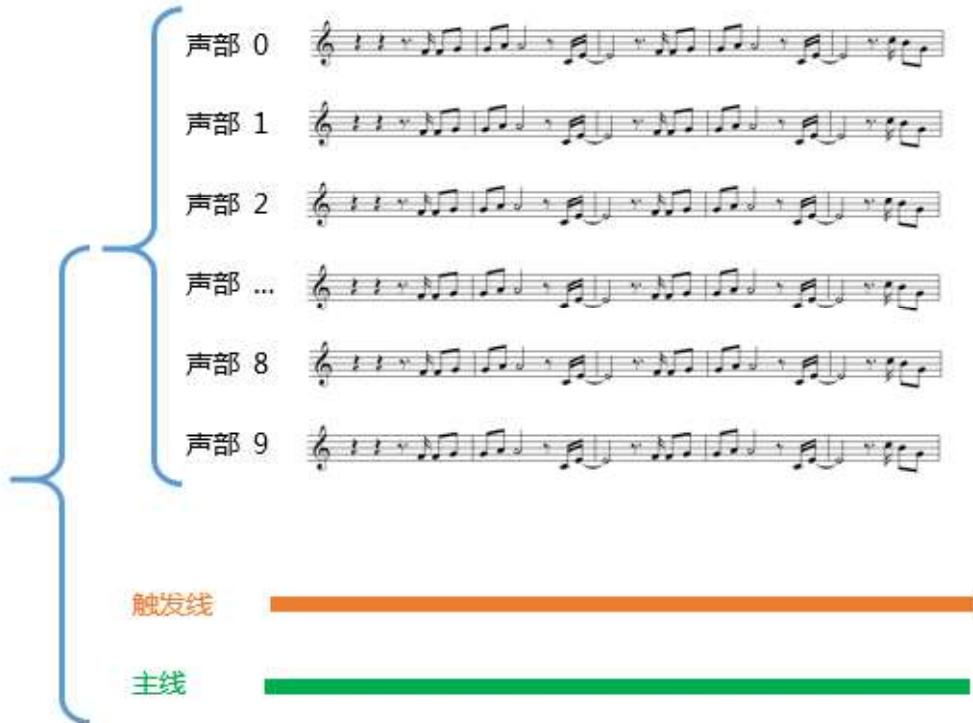
===== 点击以返回目录 =====



(1) 多音频轨道

kOS 的音频演奏（SKID）功能，支持用 10 根音频轨道同时演奏，编号 0~9，类似演奏时的多声部。

也就是说，可以进行和弦演奏。



音频演奏和 kOS 程序的主程序是独立的，
即使 kOS 程序遇到了 Wait 指令，音频演奏也不会被打断。

每一根音轨的内容都由一个音符序列组成。演奏音轨就是播放音符的 List。

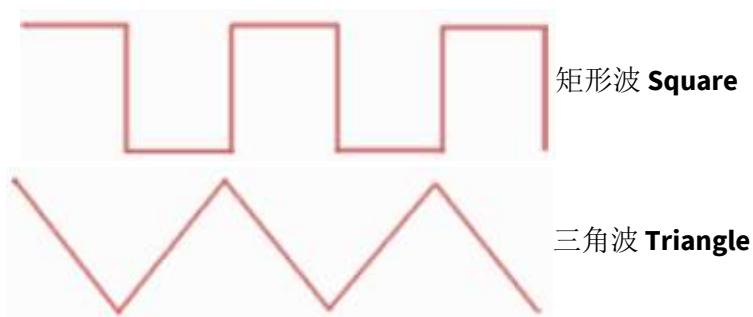
玩家可以对某一个音符设置音量（范围 0.0~1.0），
玩家也可以对某一条音频轨道设置音量（范围 0.0~1.0），
最终演奏出的效果是这两个音量值相乘。

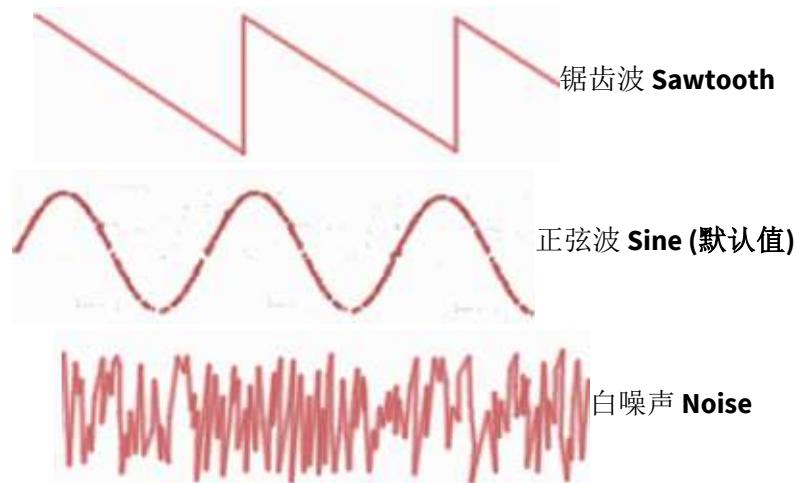
玩家还可以对某一条音频轨道设置独立的播放速度。还可以设置是否循环播放。

(2) 频率范围和基础波形

音频演奏（SKID）能发出 1Hz ~ 14kHz 频率范围的声音。

音频演奏（SKID）支持的音色，也就是声音基础波形，
有正弦波（默认）、方波、三角波、锯齿波、白噪音，这五种可供选择。
其中前三个听起来都有机械化/电子化的声音，正弦波听上去最自然。



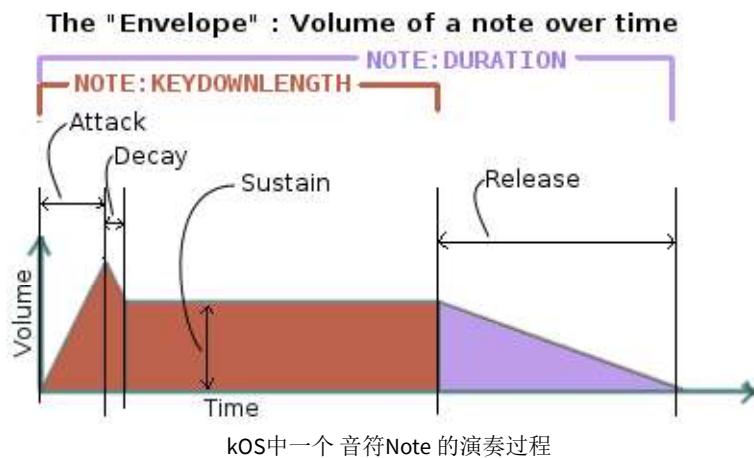


(3) 音符发音过程和音符参数

音频演奏（SKID）提供一个音符的淡入淡出模型，可以设置音符出现和结束时的响度。

音符Note 的音量有完整的 出现 - 维持 - 淡出 过程，模拟了自然的敲击音，且每个阶段的参数可调。

另外，音符Note 可以是恒定的频率，也可以是线性变化的频率



图中参数	特征	默认值	单位	结构体	备注
Attack	淡入时间	0	秒	Voice	淡入过程不受变速播放影响
Decay	衰减时间	0	秒	Voice	衰减过程不受变速播放影响
Sustain	稳定音量	1		Voice	稳定时音量是淡入峰值音量的多少倍。范围 0~1
Note : KeyDownlength	按键时长	1	秒	Note	用于计算 Sustain时长 = 按键时长 - Attack - Decay
Release	淡出时间	0.1	秒	Voice	淡出过程不受变速播放影响
Note : Duration	音符时长	1	秒	Note	\geq Note : KeyDownlength
	基础波形	正弦波		Voice	字符串。 五种基础波形供选： 正弦波、 方波、三角波、 锯齿波、白噪音
	开始频率		Hz	Note	0 Hz 的话就表示休止符
	结束频率	开始频率	Hz	Note	
Volume	音量	1		Note Voice	数值。Attack的峰值音量
.....

*注1:

结构体列为 Note 的参数，是和演奏技巧有关的，

所以安排在了 Note 结构体下，每个音符为单位进行独立调整；

结构体列为 Voice 的参数，是和乐器音色有关的，

所以安排在了 Voice 结构体下，只能以整个音频轨道为单位进行调整。

结构体列同时有 Note 和 Voice 的，在 Note 结构体 和 Voice 结构体 下，相关参数，共同作用。

*注2: KeyDownlengt 这个参数是用来计算 Sustain 时长 的。

Duration 不同，Duration 决定了下一个音符什么时候开始，

所以他影响音符怎样结束，具体有以下三种情况：

((1)) KeyDownlengt < Duration < KeyDownlengt + Release

音符还没全部淡出时就会被下一个音符的淡入打断

((2)) Duration = KeyDownlengt + Release

当音符淡出完成后立马就是下一个音符的淡入

((3)) Duration > KeyDownlengt + Release

音符淡出完成后，下一个音符淡入之前，会有一段空白

*注3: Sustain 时长 是根据 KeyDownlengt 按键时长 和 淡入 Attack 还有 衰减 Decay 算出来的。

Sustain 时长 = KeyDownlengt - Attack - Decay。

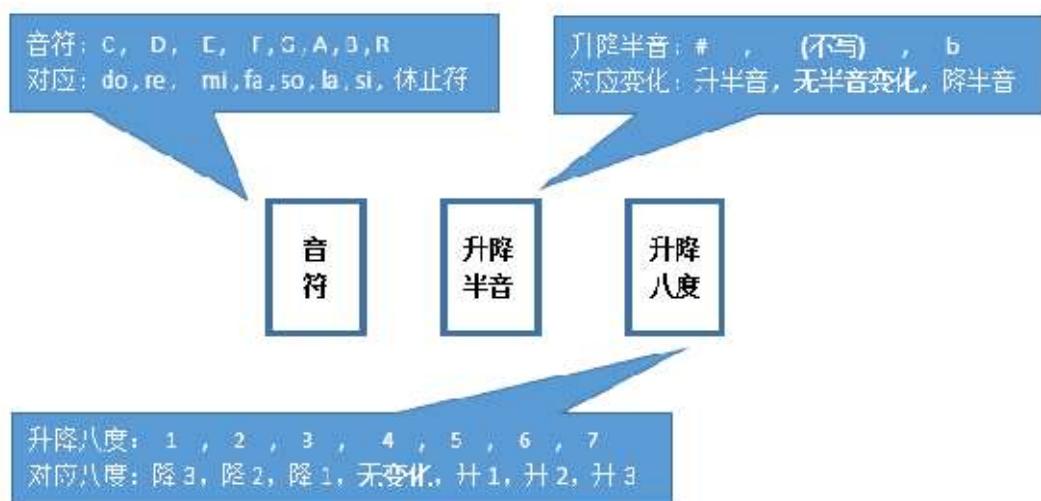
如果改了 音频轨道 Voice 的播放速度(Tempo)，Attack 和 Decay 还有 Release 的时间不会变，会变的只有 Sustain 时长。

(4) 音符的字母记法

在kOS中，玩家除了用数字表示频率之外，例如400Hz。

玩家还可以用字符来表示，字符表示的方法更适合音乐工作者。

kOS音符的字符表示一般由2或3个字符组成，如下图。



例子：

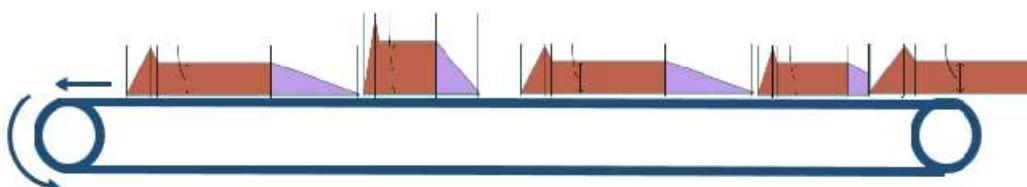
kOS音符	C4	D5	E3	F #4	Gb4	A6	B2	E #6
简谱音符	1	2	3	#4	b5	6	7	#3

(5) 音符构成音轨

确定好音符，照着曲谱做成一个音符的列表。

然后把列表送上音频轨道Voice（声部）这个传送带。整个旋律就能演奏了。

同一声部里的相邻音符，音符时长Duration 的前后边界是首尾紧贴着的。



一个Voice就像是一个匀速的传送带，放着各种音符

音频轨道Voice（声部）有很多设置选项，首先，在Voice里可以设置**演奏效果**，如下图。

从物理上，这些参数都和演奏的乐器有关。

参数	特征	默认值	单位	备注
Attack	淡入时间	0	秒	淡入过程不受变速播放影响
Decay	衰减时间	0	秒	衰减过程不受变速播放影响
Sustain	稳定音量	1		稳定时音量是淡入峰值音量的多少倍。范围 0~1
Release	淡出时间	0.1	秒	淡出过程不受变速播放影响
	基础波形	正弦波		字符串。 五种基础波形供选：正弦波、方波、三角波、锯齿波、白噪音
Volume	音量	1		数值。
Loop	循环播放	False		Boolean值
IsPlaying	播放状态			可以设 False 值来停止演奏
Tempo	播放速度	1		数值。
.....

好了最后使用 **Play** 函数就可以播放了。

5.9.3

5.9.3 音频的代码实现

===== 点击以返回目录 =====



(1) 获取音轨 GetVoice()

音频演奏 (SKID) 中一共有10个音频轨道，编号从0~9。

要用 GetVoice 函数，然后以赋值变量的形式把他们抓取之后才能使用。如下：

```
Set V0 to GetVoice(0).
```

(2) 音轨的参数设置

然后如果有必要，我们可以设置音频轨道的音色。音色参数默认值请见上一小节。

我们可以按照自己需求变更，比如：

```
Set V0:Volume to 0.9.
Set V0:Wave to "sawtooth".
Set V0:Attack to 0.1.
Set V0:Decay to 0.2.
Set V0:Sustain to 0.7.
Set V0:Release to 0.5.
```

(3) 构造音符 Note()

一段旋律是由一个个音符串联而成的，我们要构造每一个音符。

我们先来回顾一下 音符结构体 Note 的参数。

图中参数	特征	默认值	单位	备注
Note : KeyDownlength	按键时长	1	秒	用于计算 Sustain时长 = 按键时长 - Attack - Decay
Note : Duration	音符时长	1	秒	\geq Note : KeyDownlength
	开始频率		Hz	0 Hz 的话就表示休止符
	结束频率	开始频率	Hz	
Volume	音量	1		数值。Attack的峰值音量
.....

我们可以用 Note纯音符函数 和 SlideNote渐变音符函数。他们的用法如下：

```
//调用函数时如果不写全参数，没输入的参数会自动取默认值
//所以 Note函数 和 SlideNote函数 有以下几种用法：

Note(频率,音符时长,按键时长,音量) //纯音音符
Note(频率,音符时长,按键时长) //未赋值的参数会取默认值
Note(频率,音符时长) //未赋值的参数会取默认值

SlideNote(开始频率,结束频率,音符时长,按键时长,音量) //渐变音音符
SlideNote(开始频率,结束频率,音符时长,按键时长) //未赋值的参数会取默认值
SlideNote(开始频率,结束频率,音符时长) //未赋值的参数会取默认值
```

例子：

实例：

```
//标准音do(262Hz)，持续2s，之后是默认0.1秒的淡出，以及0.4秒的停顿，总共占时2.5秒，音量1
Set n1 to Note( C4 , 2.5 , 2 , 1 ).

//标准音do(262Hz)，持续2s，之后是默认0.1秒的淡出，以及0.4秒的停顿，总共占时2.5秒，音量1
Set n2 to Note( 262 , 2.5 , 2 , 1 ).

//标准音do(262Hz)，持续1.9s，之后是默认0.1秒的淡出，和下一个音符之间没有停顿，总共占时2秒，音量1
Set n3 to Note( C4 , 2 ).

Set n4 to Note( R , 4 ). //休止符，持续4s
```

(4) 演奏音轨 Voice:Play()

把每一个音符用 List函数 连成列表（数组），之后就可以用 Play函数 来演奏了。

```
Set song to List(n1,n4,n3,n2).
V0:Play(song).
```

5.9.4

5.9.4 实例11： kOS 演奏乐曲

----- 点击以返回目录 -----



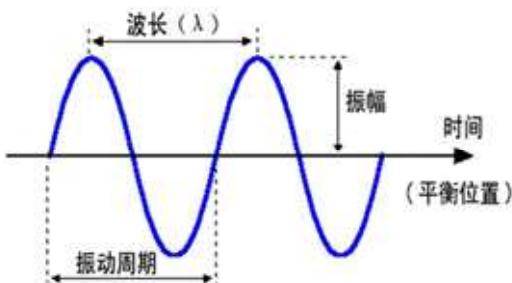
(1) 乐理知识：声音的波形函数

我们有一个典型的纯音的正弦函数如下：

$$y = A \sin[2\pi f(t - \phi_0)]$$

[正弦]

上式中振动y是关于时间变量t的函数，A为振幅，f为频率， ϕ_0 为初相位。这些可以在波形图上找到明显的几何对应。



(2) 乐理知识：声音的三要素

响度、音调、音色是声音的三要素。他们和波形函数里要素的对应关系如下。

声音要素	对应声音函数中	表现在生活中
响度	振幅 A	声音响，声音轻
音调	频率 f	声音尖，声音沉
音色	函数类型，例如 Sin正弦函数， 三角波，方波等	小提琴的声音、 二胡的声音、 费玉清的声音

*注1：正弦波的音色听起来更加自然，三角波或者方波的音色听起来更加像电子音。

然后音调与频率的对应关系如下图：

C调音符与频率对照表

音符	频率/Hz	音符	频率/Hz	音符	频率/Hz
低音1	262	中音1	523	高音1	1046
低音1#	277	中音1#	554	高音1#	1109
低音2	294	中音2	587	高音2	1175
低音2#	311	中音2#	622	高音2#	1245
低音3	330	中音3	659	高音3	1318
低音4	349	中音4	698	高音4	1397
低音4#	370	中音4#	740	高音4#	1480
低音5	392	中音5	784	高音5	1568
低音5#	415	中音5#	831	高音5#	1661
低音6	440	中音6	880	高音6	1760
低音6#	466	中音6#	932	高音6#	1865
低音7	494	中音7	988	高音7	1976

*注2：上表中数字1~7对应 do re mi fa so la si

高音(八度)是频率高一倍，低八度是频率低一半
表中数据整体呈现的是一个对数关系。

(3) 乐理知识：认识五线谱

我们知道，声音函数除了时间变量之外，还有振幅、频率、音色这三个元素。
而对于某一种乐器，音色一般是固定的。（还有管风琴这种一台顶一群的例外）
所以演奏乐器时要注意的一般就是什么时候演奏什么音，还有演奏的音要是多高，多响，这几件事。
记录什么时候演奏什么音，还有演奏的音要是多高，多响，这几类信息的东西，叫乐谱。
五线谱是一种最常见的，也是专为钢琴弹奏而发明的一种乐谱。

插画钢琴网 www.77music.com

乐谱编号:14748

少女さとり 3rd eye~

五线谱是一种横向卷轴式的标记记录。它可以分为多个**声部**。多个声部就像是多个音频轨道，同时播放演奏。

上图中有三行五线谱并列连在一起，就是有三个声部，其中第一声部不发声，第二第三声部在演奏。

有多个声同时演奏的音乐叫做**和弦**。

五线谱为了书写和演奏方便，在**音符**之外设有**谱号**、**调号**、**拍号**、**演奏速度**等影响全局的参数。

有关**音符**：音符是钢琴发声的标记，一个音符对应钢琴的一个琴键按下过一段之间之后再松开弹起的过程。

音符分为全音符、二分音符、四分音符、八分音符、十六分音符等等。

他们对应的是从琴键按下到弹起的时间长度（演奏时间），占用的是1、1/2、1/4、1/8、1/16等等个时间单位。

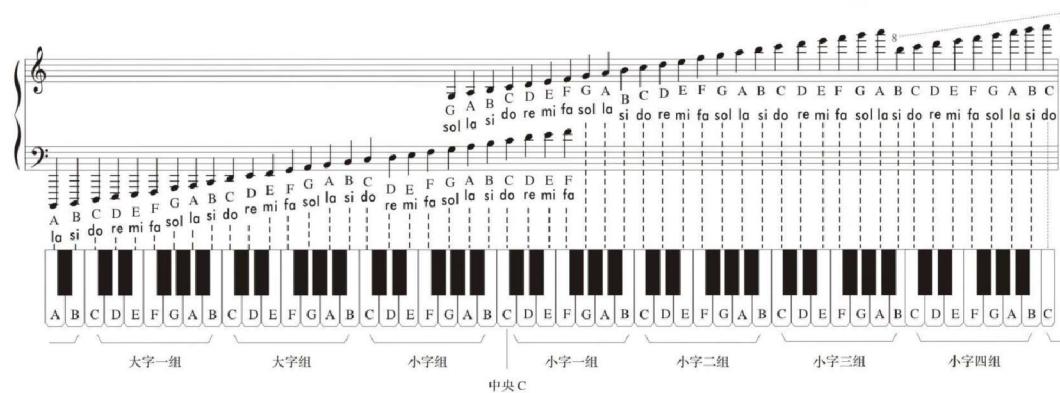
音符的右边可以加一个**附点**，用来表示这个音符的演奏时间是原来的1.5倍。

例如，总时间上，2个附点4分音符=3个4分音符。



然后我们该把钢琴键盘搬上来了：

大谱表与钢琴键盘对照表



* 图中标出的“组名”，是为了让学琴者对键盘结构有个基本的认知，而各组每个音的名称不在这里一一列出。



音符在五线谱上的高度位置代表音符的**音调**（频率），位置越高音调越高。

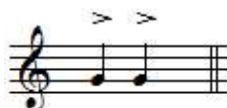
不同高度的音符对应不同的钢琴键位。

音符周围还可以写一些**变音记号**，用来微调这个音符的音调。

#	升号	升高半音
b	降号	降低半音
还原号		将升高或降低的音还原

变音记号

某些音符在演奏时需要加大声音，称为**重音**。重音用记号>表示



有关**谱号**：谱号决定音符书写高度的平移。但不改变音符的含义，所以也不改变对应的钢琴键位。

谱号写在五线谱每行行首。谱号有好多个，常见的是高音谱号（G谱号）和低音谱号（F谱号）。

谱号对定了本行五线谱里哪个高度位置的音符才算 do。

谱号对于钢琴演奏者的用处是，告诉你演奏时把手放在钢琴的左半边低音区还是右半边高音区，弹起来才顺手不累。



高音谱号 低音谱号

有关**休止符**：休止符也是一种音符，代表不演奏，用来给时间占位。

五线谱记谱	简谱记法	休止符名称
—	0000	全休止符
—	00	二分休止符
2	0	四分休止符
7	0	八分休止符
7	0	十六分休止符
7	0	三十二分休止符
7	0	六十四分休止符

休止符

有关**调号**：调号决定音符的起始音调。调号书写在谱号之后。

改变调号不会不改变音符在五线谱上的位置高度，但是会改变钢琴的对应键位。

调号是一种进阶标记，在简单的乐谱里很多都不会标记调号，这些乐谱默认为C大调。



C G D A E B #F #C



C F bB bE bA bD bG bC

五线谱的调号



不同调号在钢琴上键位的对比

有关**拍号**：五线谱被等时间的划分为一个个小节。拍号决定一个小节对应多少时间单位的音符。

拍号写成分数的样子。分母表示拍子的时值也就是说用几分音符来当一拍，分子代表每一小节有多少拍子。

如 $2/4$ 代表用四分音符代表一拍，每一小节有两拍。

有关**演奏速度**：演奏速度比如说60的话，就是一分钟演奏60个四分音符，正好1秒1个。

有关**延音线/连音线**：连接两个相邻的音符的线，像桥一样接在两个音符的头顶。

如果两个音符是相同音调的同一个琴键，在演奏时作为一个音符，从头按到尾就可以了；

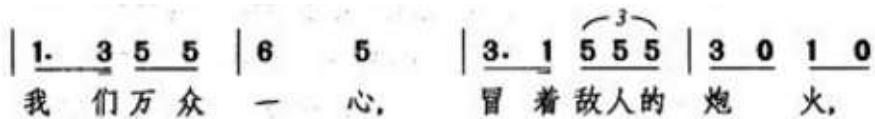
如果两个音符是不同音调的两个不同琴键，演奏的时候这两个音符要连起来演奏，声音不能断开。



延音线

有关**三连音**：连接3个相邻的相同音调音符的线，像桥一样接在两个音符的头顶，并且在弧线上写有数字3。

和延音线的区别是，延音线演奏时要当做弹1个音符，而三连音要紧挨着弹3下。



国歌中的三连音

有关**重复标记**和**跳房子**: 如果乐曲中有重读的节奏, 就可以用重复标记来偷懒。

如下图, 演奏顺序为: 121234546788

(4) 乐理知识: 认识简谱

简谱是另一种常见的乐谱。简谱和五线谱之间可以相互转换, 并且优点是书写方便。

简谱里没有谱号这个概念, 其他和五线谱照旧。简谱适合多种乐器。

简谱和五线谱的对照

简谱是一种横向卷轴式的标记记录。它可以分为多个**声部**。多个声部就像是多个音频轨道, 同时播放演奏。

下图中有两行简谱并列连在一起, 就是有两个声部。

有多个声同时演奏的音乐叫做**和弦**。

中国少年先锋队队歌

词：周郁辉
曲：寄明

The musical score consists of two staves of music. The first staff starts with a quarter note followed by a dash, then a sequence of notes: 5, 3, 1, 2, 3, 5, 6, 2, 1, 7, 6, 5, 5, 5, 3, 3, 2, 1, 1, 2, 1, 3, 5, 4, 2, 3, 4, 5, 5, 1, 1, 7, 6, 5, 3, 1, 2. The second staff continues with 6, 6, 1, 2, 3, 0, 5, 0, 0, 6, 4, 3, 2, 3, 2, 1, 2, 3, 5, 1, 6, 5, 6, 3, 2, 1, 1, 3, 1, 2, 3, 0, 5, 0, 0, 4, 2, 1, 2, 3, 2, 1, 2, 3, 5, 6, 4, 3, 4, 3, 2, 1, 1.

歌词如下：

我们是共产主义接班人，沿着继承革命先辈的光荣传统。
我们是共产主义接班人，民，鲜艳的红领巾飘扬在胸前称。
统，爱，爱，祖，国，爱，人，民，少先队员是我们骄傲的前胸。
程，爱，爱，祖，国，爱，人，民，少先队员是我们骄傲的前胸。

有两个声部的简谱

简谱为了书写和演奏方便，在音符之外设有调号、拍号、演奏速度等影响全局的参数。

有关**音符**：音符是乐器发声的标记，一个音符对应开始发声→结束发声的过程。

音符分为全音符、二分音符、四分音符、八分音符、十六分音符等等。

他们对应的是从琴键按下到弹起的时间长度（演奏时间），占用的是1、1/2、1/4、1/8、1/16等等个时间单位。

音符的右边可以加一个**附点**，用来表示这个音符的演奏时间是原来的1.5倍。

例如，总时间上，2个附点4分音符=3个4分音符。

音符名称	写法	时值
全音符	5 — — —	四拍
二分音符	5 —	二拍
四分音符	5	一拍
八分音符	5	半拍
十六分音符	5 =	四分之一拍
三十二分音符	5	八分之一拍

音符的长短

简谱中的音符以数字代表音符的**音调**（频率）。

1, 2, 3, 4, 5, 6, 7表示，对应do,re,mi,fa,so,la,si。数值越高音调就越高。

在数字上方或下方加点，表示这个音符的**高八度**或**低八度**音，上面加两个点就代表高两个八度，以此类推

音符周围还可以写一些**变音记号**，用来微调这个音符的音调。



将标准的音符升高或降低得来的音，就是变化音。将音符升高半音，叫升音。用“♯”（升号）表示。

#1 #2 #3 #4 #5 #6 #7

标准的音降低半音，用“♭”（降号）表示。

♭1 ♭2 ♭3 ♭4 ♭5 ♭6 ♭7

变音记号

某些音符在演奏时需要加大声音，称为**重音**。重音用记号>表示

||: > i i 0 |

有关**休止符**：简谱中休止符以数字0表示。**休止符代表不演奏，用来给时间占位。**



七子之歌 —— 澳门

I=C⁴₄ 电视片《澳门岁月》主题曲 闻一多词 李海鹰曲

5 3 5 3 5 | 6 5 3 6 5 - | 1 1 2 3 5 3 | 2 0 3 5 - |
 你可知 MA-CAU 不是我真姓，我离开你太久了，母亲。

5 5 6 5 3 5 5 | 6 5 1 6 5 - | 1 5 3 2 1 2 | 3 5 . 5 2 3 |
 但是他们掠去的是我的肉体，你依然保管我内心 的 灵

1 1 --- | 1 1 --- 0 5 | 3 2 . 1 6 5 5 | 6 6 5 6 . 1 3 1 | 2 2 - 0 5 |
 魂。 魂。 那三百年 来梦寐不忘的生母啊，请

3 2 . 1 6 5 5 | 6 6 5 6 3 2 | 2 2 - - - | 3 2 . 1 6 5 5 |
 叫儿的乳 名，叫我一声澳 门。 母亲啊母 亲。

1 6 5 3 2 | | 0 2 3 5 - 5 - 0 5 6 | 1 1 --- |
 我要 回 来 母 亲。 母 亲。

刘伟佳 抄写

带有休止符的简谱

有关调号：调号决定音符的起始音调。

简谱的调号直接写的是调号代表的字母（下图中红蓝字），调号一般书写在声部的开头，也可以写在小节的开头，表示从这里开始之后都用这个调。

改变调号不会不改变音符的音调，但是会改变钢琴的对应键位。

调号是一种进阶标记，在简单的乐谱里很多都不会标记调号，这些乐谱默认为C大调。

C G D A E B #F #C

C F bB bE bA bD bG

简谱的调号（你别看五线谱部分）



不同调号在钢琴上键位的对比

有关**拍号**：简谱被等时间的划分为一个个小节。**拍号决定一个小节对应多少时间单位的音符。**

拍号写成分数的样子。分母表示拍子的时值也就是说用几分音符来当一拍，分子代表每一小节有多少拍子。

如2/4代表用四分音符代表一拍，每一小节有两拍。

有关**演奏速度**：演奏速度比如说60的话，就是一分钟演奏60个四分音符，正好1秒1个。

有关**延音线/连音线**：连接两个不同时间的音符的线，像桥一样接在两个音符的头顶。

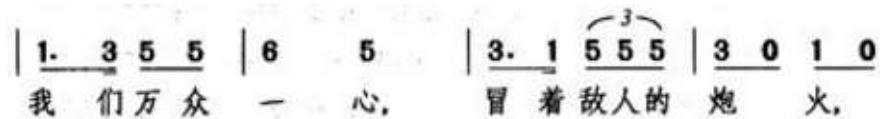
如果两个音符是相同音调的同一个琴键，在演奏时作为一个音符，从头按到尾就可以了；

如果两个音符是不同音调的两个不同琴键，演奏的时候这两个音符要连起来演奏，声音不能断开。



有关**三连音**：连接3个相邻的相同音调音符的线，像桥一样接在两个音符的头顶，并且在弧线上写有数字3。

和延音线的区别是，延音线演奏时要当做弹1个音符，而三连音要紧挨着弹3下。



国歌中的三连音

有关**重复标记**和**跳房子**：如果乐曲中有重读的节奏，就可以用重复标记来偷懒。

如下图，演奏顺序为：1 2 1 2 3 4 5 6 7 8 9 10

1.

1 | 2:|| 3 | 4 ||:5 | 6 | 7 | 8:|| 9 | 10 ||

(5) 演奏乐曲：两只老虎

好了本例音乐的简谱如下，

两 只 老 虎1=E $\frac{2}{4}$

快 名 词 曲

1 2 3 1 | 1 2 3 1 | 3 4 5 | 3 4 5 | 5 6 5 4 |
两只老虎，两只老虎，跑得快，跑得快。一只没有

3 1 | 5 6 5 4 | 3 1 | 2 5 | 1 0 | 2 5 | 1 0 |
眼睛，一只没有尾巴，真奇怪！真奇怪！

本曲谱上

本例中的音乐以及播放录像有在 <https://tieba.baidu.com/p/4968968340> 发表过。

```
// 《两只老虎》的简谱是：（以下是一种自创的 txt 里记录简谱的方法）
// 1=C 4/4
// | 1×2×3×1 | 1×2×3×1 | 3×4×5×0 | 3×4×5×0 |
// | 5_ 6_ 5_ 4_ 3×1 | 5_ 6_ 5_ 4_ 3×1 | 2×.5×1×0 | 2×.5×1×0 |

Set v to GetVoice (0). //取音频轨道
Set s to List (). //乐谱要求是note结构体的列表，准备一个空集，慢慢加
//四分音符设定为长度 0.5s
s:Add (Note("C4",0.5)).s:Add (Note("D4",0.5)).s:Add (Note("E4",0.5)).s:Add (Note("C4",0.5)).
s:Add (Note("C4",0.5)).s:Add (Note("D4",0.5)).s:Add (Note("E4",0.5)).s:Add (Note("C4",0.5)).
s:Add (Note("E4",0.5)).s:Add (Note("F4",0.5)).s:Add (Note("G4",0.5)).s:Add (Note("R",0.5)).
s:Add (Note("E4",0.5)).s:Add (Note("F4",0.5)).s:Add (Note("G4",0.5)).s:Add (Note("R",0.5)).
s:Add (Note("G4",0.25)).s:Add (Note("A4",0.25)).s:Add (Note("G4",0.25)).s:Add (Note("F4",0.25)).
s:Add (Note("E4",0.5)).s:Add (Note("C4",0.5)).
s:Add (Note("G4",0.25)).s:Add (Note("A4",0.25)).s:Add (Note("G4",0.25)).s:Add (Note("F4",0.25)).
s:Add (Note("E4",0.5)).s:Add (Note("C4",0.5)).
s:Add (Note("D4",0.5)).s:Add (Note("G3",0.5)).s:Add (Note("C4",0.5)).s:Add (Note("R",0.5)).
s:Add (Note("D4",0.5)).s:Add (Note("G3",0.5)).s:Add (Note("C4",0.5)).s:Add (Note("R",0.5)).

Set v:wave to "sawtooth". //波形设置为锯齿波，听起来像电子贺卡
v:Play (s). //播放乐曲
```

5.10

5.10 指定部件（获取部件结构体Part）

5.10.1

5.10.1 部件的树状组织结构

===== 点击以返回目录 =====

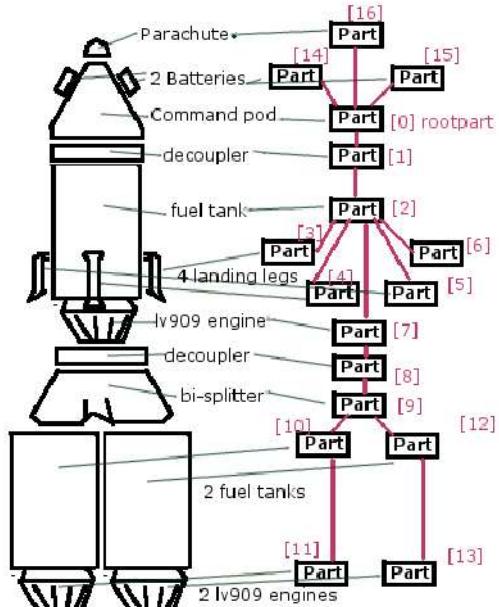


KSP 的飞船是由部件构成的，部件是游戏运算的最小单位。

kOS 以树状结构来存储飞船每个部件的数据。

树状结构图的根部件[0] (Vessel:RootPart) 是制作飞船时第一个添加的部件，吸附固定在根部件上的是第一级部件[1]，吸附固定在第一级部件上的是第二级部件[2].....以此类推。部件结构图切不可构成封闭环状。

以下为火箭的部件结构图示意。

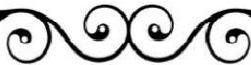


Demonstrating the tree layout of parts

5.10.2

5.10.2 通过部件列表指定部件

===== 点击以返回目录 =====



想要对部件进行操作，先要获得该部件的 部件结构体Part。（指定部件）

玩家可以通调用载具结构体Vessel 下属的 Parts 成员，
来得到这个载具上所有 部件结构体Part 的一个扁平的列表。

```
Ship:Parts. //当前载具上所有部件构成的列表
Set p3 to Ship:Parts[2]. //p3 是这个列表里的第3个 部件结构体Part 成员
```

*注1：虽然载具上的部件是有树状组织结构的，但是 Parts 得到的列表是压平后的，只有一层的。

5.10.3

5.10.3 通过名称搜索并指定部件

===== 点击以返回目录 =====



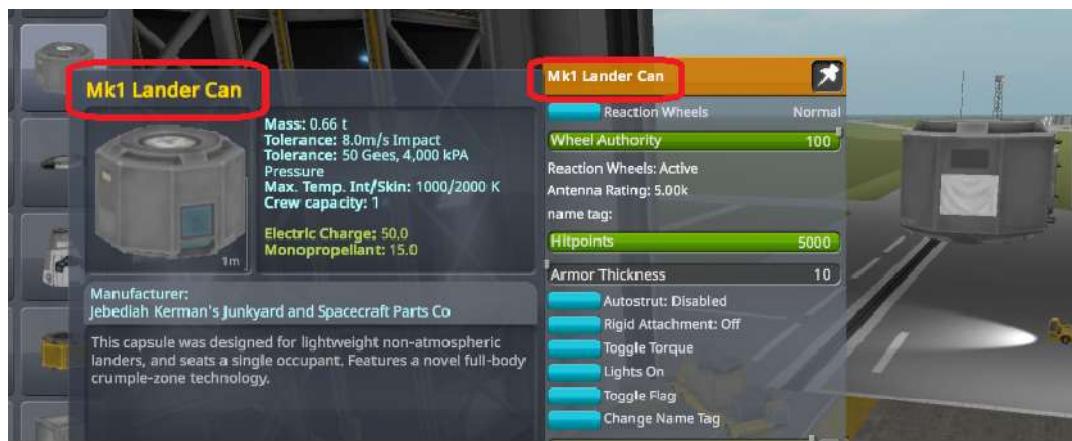
(1) 部件名、标题名、标签名

一个部件一共可以有三种名称：部件名Name、标题名Title、标签名NameTag

KSP 游戏中每一种部件都有自己的 部件名Name 和 标题名Title。

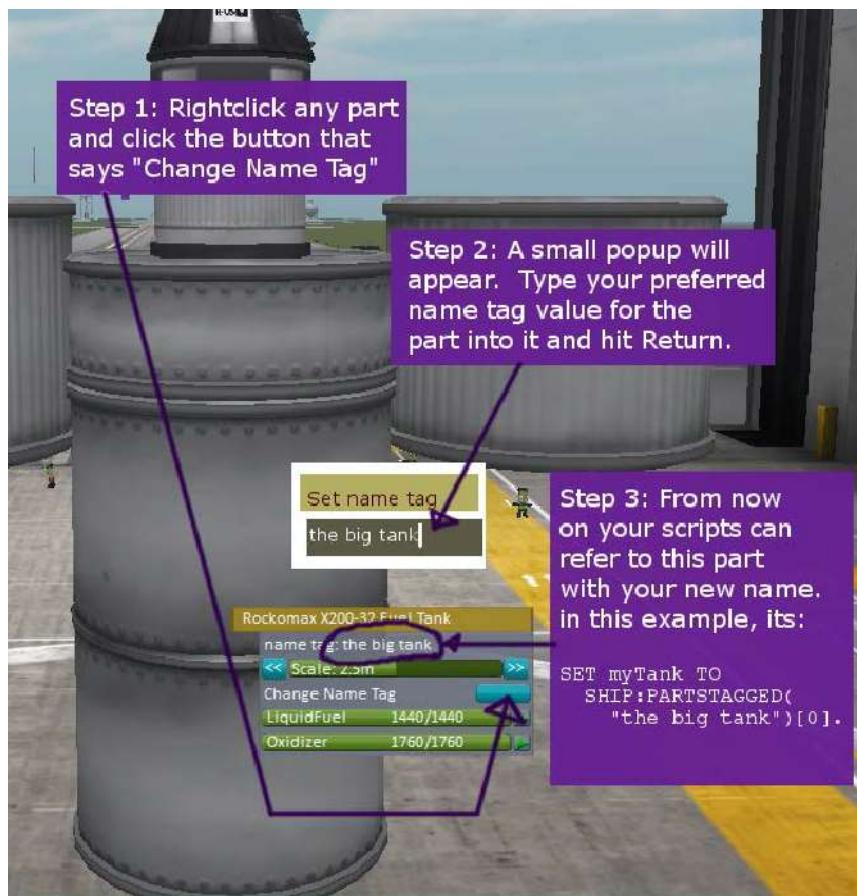
部件名Name 是一个不在游戏里显示的后台数据。

标题名Title 是平时玩家在车间里看到的部件名称。（下图红框处）



KSP 还提供一套可供玩家自行编辑的标签名系统 NameTag (Tag)。

Tag 值默认为空，玩家可以在车间和飞行时随时更改。如下图：



(2) 关于对称安装部件的标签名

把对称安装部件里的每一个都可以单独改 Tag 名。

但是把对称安装部件里的一个改 Tag 名后，移开再装回去，会波及其他同组里对称安装的部件。

(3) 通过名称指定部件

kOS 提供了几个函数，可以根据部件名Name、标题名Title、标签名Tag 来搜索并指定部件。

指定部件之后可以获得该部件的部件结构体Part，玩家可以据此对该部件做进一步操作，例如查看部件信息，操作部件的右键菜单。

```
Ship:PartsNamed(string) //返回当前载具上 部件名Name 里包含 字符串string 的部件构成的列表
Ship:PartsTitled(string) //返回当前载具上 标题名Title 里包含 字符串string 的部件构成的列表
Ship:PartsTagged(string) //返回当前载具上 标签名Tag 里包含 字符串string 的部件构成的列表
//返回当前载具上 部件名Name、标题名Title、标签名Tag 里包含字符串string的部件构成的列表
Ship:PartsDubbed(string)
```

以标签名Tag 为例，我们可以这么用：

```
Ship:PartsTagged("aaa"). //符合Tag名aaa的所有部件构成的列表
Set parts to Ship:PartsTagged("aaa"). //parts 是符合Tag名aaa的所有部件构成的列表
Set part to parts[0]. //part 是 parts 的第一个成员。

Set part to Ship:PartsTagged("aaa") [0]. //这句和前两句一个意思

//然后我们就可以对我们已经指定的部件 part 进行进一步操作了。
Print part:Mass. //比如说显示这个部件的质量
Print part:Thrust. //如果这是一个引擎部件，我们还可以得到这个引擎的推力。
part:Unlock(). //如果这是一个对接口部件，我们还可以让他解除对接
.....等等用途
```

推荐用 Tag 作为 kOS 中调用部件的手段。因为 Tag 可以随时编辑，泛用性高。

另外还有一种用法，是以“部件有Tag命名”作为搜索条件的

```
Ship:AllPartsTagged(). //有Tag名的所有部件构成的列表
```

5.10.4

5.10.4 通过动作组搜索并指定部件

===== 点击以返回目录 =====



和以名称指定部件的方式类似。

kOS 也支持以动作组作为共同点，来获得更精准的部件列表。

```
Ship:PartsInGroup(3). //动作组3 (Ag3) 对应的所有部件构成的列表
Set p2 to Ship:PartsInGroup(3) [1]. //p3 是这个列表里的第2个 部件结构体Part 成员
```

5.10.5

5.10.5 通过正则表达式搜索并指定部件

===== 点击以返回目录 =====



kOS 也支持使用正则表达式来搜索载具上的部件。

```
//部件名Name、标题名Title、标签名Tag 符合 正则表达式规则some 的所有部件构成的列表
Ship:PartsDubbedPattern(some).
```

有关上文 some 的地方如何填写，有关正则表达式的内容，
请查询 <https://docs.microsoft.com/en-us/dotnet/standard/base-types/regular-expression-language-quick-reference>。

5.11

5.11 部件操作 和 部件模组 ParModules

5.11.1

5.11.1 部件信息和部件操作

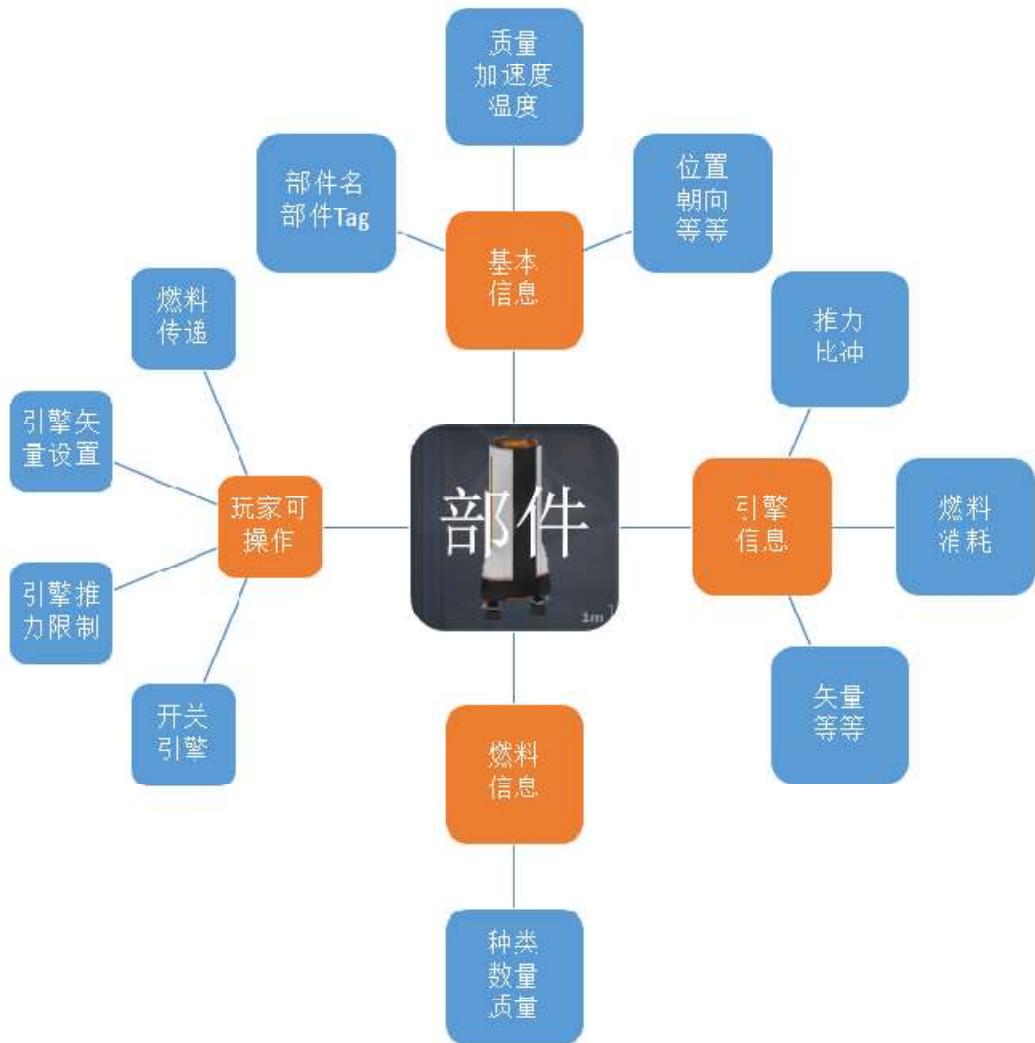
===== 点击以返回目录 =====



部件 (Part) 是游戏运算的最小对象。为便于游戏运算，部件上会集成各种的信息内容。游戏的后台计算就是根据设定的物理定律和游戏规则，对这些信息做加加减减算来算去，函数触发来触发去。



下图不太完善的列举了一下这个部件上承载的一些信息。



5.11.2

5.11.2 部件模组 PartModules

===== 点击以返回目录 =====



部件模组 PartModule 是 kOS 最有无限可能的一个功能。

部件 Part 下包含 部件模组 PartModule。

部件模组 PartModule 下又包含和右键菜单直接对应的内容成员。

这些内容成员可以分为三类：字段（**KSPFields**）、事件（**KSPEvents**）、动作组关联（**KSPAction**）。

玩家变更成员的值，或者触发成员的行为函数，就可以激发部件右键菜单里的操控。

只要是以 Module 形式融入游戏的内容，kOS 都能读取和调用。

因此，KOS 不仅能控制原版 KSP 中的部件右键菜单，

也兼容所有的部件类 MOD，并且也包含部分带 dll 文件的 MOD。

(1) 指定部件模组（获取部件模组结构体 PartModule）

玩家获取部件Part之后，可以通过部件Part下属的GetModule函数，来指定部件模组PartModule。

不过在使用GetModule函数之前，需要先知道所要的部件模组的Module名。

有两种方式可以知道Module名

((1)) 查看部件的 CFG文件

例如下图是某个着陆架的 CFG文件，
这个部件有一个名为 ModuleLandingLeg 的 部件模组PartModule，
这个部件模组里有一个名为 Deploy 的 KSPAction。
字面意思看，就能知道是展开着陆架的动作按钮。

```
angularDrag = 2
crashTolerance = 12
maxTemp = 2000 // = 2900
bulkheadProfiles = srf
MODULE
{
    name = ModuleLandingLeg
    animationName = Deploy
    wheelColliderName = wheelCollider
    suspensionTransformName = Piston
    orientFootToGround = true
    landingFootName = foot
    alignFootUp = true
    suspensionUpperLimit = 0.60
    impactTolerance = 300
    suspensionSpring = 1.25
    suspensionDamper = 1
    suspensionOffset = 0, 0.02,0
}
}
```

((2)) 查看部件的 Module名列表

一般更常用的方法是查看部件的Module名的字符串列表。

```
//假设变量 P 是我们要操作的 部件Part

//在命令窗口显示部件P上所有 PartModule 的 Module名
Print P:Modules. //这一步应该在测试的时候有，实际运行的时候删掉

//例如我们看到第3个是我们要的，那么取第3个 Module 的字符串名字
Set MName to P:Modules[2].
Set PModule to P:GetModule(MName). //PModule 就是我们要的 PartModule 了。
```

(2) 字段 (KSPFields)

KSP Field 对应的是，能在部件部件右键菜单里设置的数值，包含以按钮形式显示的逻辑值。

KSPField 是单个字段。

数值和字符串字段直接显示字段值。

布尔字段的 KSPField 以按钮形式显示，按钮按下后不会弹回来而是凹进去。

用法：

```
//假设 ModuleA 是我们要操作的 部件模组PartModule

//在命令窗口显示 部件模组ModuleA 上所有 字段KSPField 的 信息
Print ModuleA:AllFields. //这一步应该在测试的时候有，实际运行的时候删掉

//例如我们看到第3个字段 FName 是我们要操作的
ModuleA:GetField(FName). //返回部件模块moduleA的FName的值
ModuleA:SetField(FName,newFieldB). //将部件模块moduleA的FName的值设为newFieldB
```

(3) 事件 (KSPEvents)

KSP Event 对应的是，能在部件部件右键菜单里操作的，部件动作。

KSPEvent 能调用函数（方法），实现一些字段无法实现的功能（例如升起和降下起落架）。

KSPEvent 外形为按钮，按下后会弹回来。

用法：

```
//假设 ModuleA 是我们要操作的 部件模组PartModule

//在命令窗口显示 部件模组ModuleA 上所有 事件KSPEvent 的 信息
Print ModuleA:AllEvents. //这一步应该在测试的时候有，实际运行的时候删掉

//例如我们看到第3个事件 EName 是我们要操作的
ModuleA:DoEvent(EName). 运行部件模块moduleA的EName事件
```

注意，有些右键内容的名称是动态改变的，比如起落架降下和升起，降下后你再叫他降下是没有用的。

只要显示在右键菜单里的，kOS基本动能动。所以kOS能支持纯部件MOD。

(4) 动作组关联 (KSPActions)

KSP Action 对应的是，能在车间的动作组编辑里设置的，部件动作。

KSPAction 和 KSPEvent 类似，不过他是对应车间里可以设置的所有动作组。

例如引擎通常可以在车间里设置 toggle engine (开/关引擎) 的动作组，设置好之后，按对应数字键就可以触发动作。而 kOS 的 KSPAction，不需要设置动作组，就能在程序里触发这个行为。

用法：

```
//假设 ModuleA 是我们要操作的 部件模组PartModule

//在命令窗口显示 部件模组ModuleA 上所有 动作组关联KSPAction 的 信息
Print ModuleA:AllActions. //这一步应该在测试的时候有，实际运行的时候删掉

//例如我们看到第3个动作组关联 AName 是我们要操作的
ModuleA:DoAction(AName,boolan). //用布尔值触发部件模块moduleA中的AName动作组行为
```

注意，有些右键内容的名称是动态改变的，比如起落架降下和升起，降下后你再叫他降下是没有用的。

只要显示在车间动作组里的，kOS基本动能动。所以kOS能支持纯部件MOD。

5.11.3

5.11.3 实例12：部件右键菜单使用集锦

===== 点击以返回目录 =====



(1) 货仓的开合（调用动作组）



首先我们要明确一点：部件模组PartModule 是一种功能非常强大，但是使用起来也很麻烦的东西。

本节和上一节花了很多篇幅，只是为了做“在部件的右键菜单里点按钮”这样的事。

平时实践的时候，对于不复杂的载具，简单的部件操作，
用“预设动作组+调用动作组”的方法来进行操作，这样其实更方便。

比如，对于下图这样的载具（KSP自带载具），假设我们要用 kOS 控制飞机尾部货仓的开合，

一个简单的做法是把尾部货仓开合的动作 Ramp 挂钩到动作组 AG9，
然后在 kOS 程序里只需要启动动作组 AG9，就可以实现目的了。





```
//动作组变量 AGX 对应键盘上字母键 是按下还是弹起的状态
Ag9 Off. Wait 0.001. //先要确保按键 9 是谈起的
Ag9 On. Wait 0.001. Ag9 Off. //按下按键，然后立马弹起，此时货仓关闭
Wait 5. //等五秒钟
Ag9 On. Wait 0.001. Ag9 Off. //按下按键，然后立马弹起，此时货仓重新打开
```

结果图下图，我们成功的让尾部货仓关闭又打开了。



*注1：上文中“简单的部件操作”指的是在动作组设置里能找到的操作。
上文中“不复杂的载具”指的是能用10个数字键的动作组排得下操作的载具。

(2) BDA枪炮的射击 (KSP Event、KSP Action)

kOS 支持 BDA武器MOD 中机枪和火炮的射击动作，但无法控制这些武器的转动。所以使用这些武器时，要在车间里把转动范围调到 0。使用固定式枪炮时不需要此操作。



以上图机枪为例，假设该机枪的标签名Tag为 www，这样子设置是因为找起部件比较方便。

```
Set ps to ship:Parttagged("www"). //创建本载具上符合名称的部件列表ps
Set ps to ps[0]. //部件列表里只有一个部件ps，那就是我们要操作的机枪
set pm to ps:modules.
print pm. //我们不知道机枪的 Partmodule 叫啥名字
```

//屏幕显示下左图，我们知道机枪的 PartModule 叫 ModuleWeapon

```
LIST of 10 items:
[0] = "ModuleTurret"
[1] = "ModuleWeapon"
[2] = "BDALookConstraintUp"
[3] = "BDALookConstraintUp"
[4] = "BDALookConstraintUp"
[5] = "BDALookConstraintUp"
[6] = "BDAScaleByDistance"
[7] = "PartTemperatureState"
[8] = "KOSNameTag"
[9] = "HitpointTracker"
```

```
ModuleWeapon, containing:
LIST of 13 items:
[0] = "(get-only) barrage, is Boolean"
[1] = "(get-only) status, is String"
[2] = "(settable) engage range min, is Single"
[3] = "(settable) engage range max, is Single"
[4] = "(settable) engage air, is Boolean"
[5] = "(settable) engage missile, is Boolean"
[6] = "(settable) engage surface, is Boolean"
[7] = "(settable) engage slw, is Boolean"
[8] = "(callable) toggle, is KSPEvent"
[9] = "(callable) disable engage options, is KSPEvent"
[10] = "(callable) toggle weapon, is KSPAction"
[11] = "(callable) fire (toggle), is KSPAction"
[12] = "(callable) fire (hold), is KSPAction"
```

```
set pm to ps:getmodule(pm[1]).//按着 Partmodule 名字抓取Partmodule，存为pm
print pm.//然后再看看武器 Partmodule 的内容
```

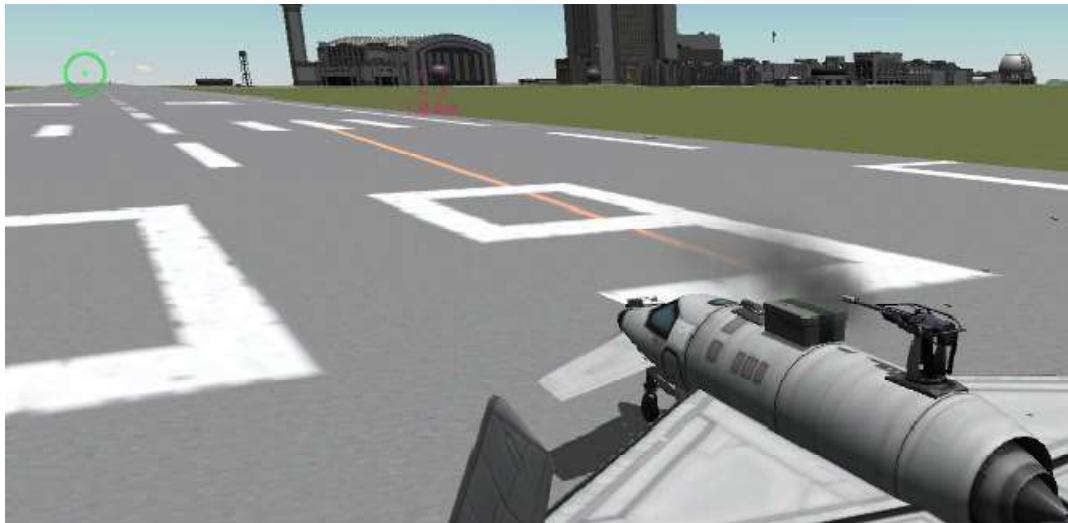
```
//屏幕显示上右图，对比游戏中的文字，
//我们知道唤醒机枪要用 toggle，连续射击要用 fire (toggle)，注意括号前有空格
```

然后我们就可以从头到尾连起来了。

```

Set ps to ship:Parttagged("www"). //创建本载具上符合名称的部件列表ps
Set ps to ps[0]. //部件列表里只有一个部件ps, 那就是我们要操作的机枪
set pm to ps:getmodule(ps:Modules[1]). //取此部件的 ModuleWeapon 部件模组
wait 1. //准备1秒钟后唤醒武器
pm:DoEvent("toggle").//唤醒武器, 这是个 KSP Event
wait 1. //准备1秒钟后射击
pm:DoAction("fire (toggle)",true).//开始射击, 这是个 KSP Action
wait 1. //连续射击1秒钟
pm:DoAction("fire (toggle)",false).//射击结束, 这是个 KSP Action
wait 1.

```



好。大功告成。

*注1：玩家可通过 KOS程序 控制的 转轴MOD 来代替枪炮的转动瞄准功能。
玩家可通过 KOS程序飞控 的自定义固推火箭（自定义组件MOD）来代替 BDA导弹。

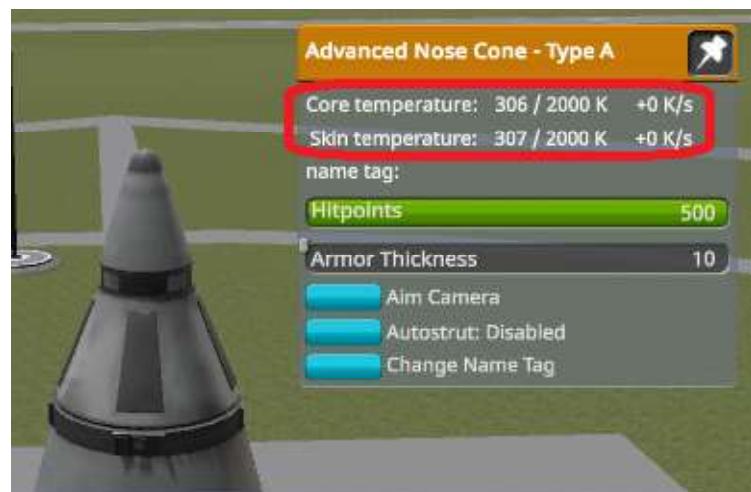
(3) 预防载具过热（KSP Field 读取）



((1)) 获取部件的温度数据

安装完 Critical Temperature Gauge（部件温度MOD）之后，
我们就可以在发射时，从部件的右键菜单，看到部件的内部温度、外部温度、还有他们的温度上限了。

KOS 能以字符串形式读到这些温度数据。



这为我们避免载具过热爆炸提供了可行性。

(2) KSP 中的热量交换

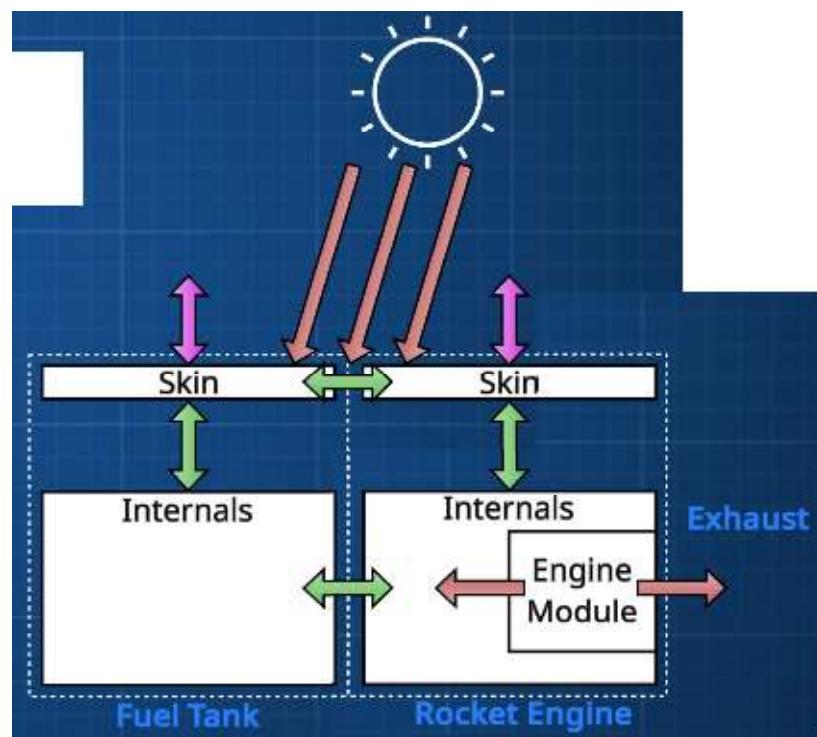
在 KSP 中计算热量时，每个部件都被分成表面和内部两部分，两部分有各自的温度。

相连部件的表面与表面之间相互交换热量，内部与内部之间相互交换热量，

同部件的表面与内部相互交换热量。

除此之外，太阳辐射、环境气流，只作用于部件的表面，影响表面温度。

而引擎部件、热核电池等部件的发热，只作用于部件内部，影响内部温度。



(3) 预防再聚过热

知道了这些以后，我们捏个推力过剩的火箭用来垂直向上飞。如下图。

另外请注意，此火箭只有中心的引擎打开了引擎矢量。

根据实测，如果火箭全推力工作，上升过程中顶端风帽部件会因为表面温度过热而爆炸。

我们的任务是写个 kOS 程序，在必要的时候减小节流阀 Throttle，

让此部件表面温度稳定在 极限温度-100开。我们给风帽部件取 标签名Tag 为qwert。



在实例4: kOS 程序的优化 中，我们用到了一个反馈控制来让火箭加速度维持在 2个G。这里我们也用类似的做法。

```

//自定义函数，从字符串 "xxxxxxxx" 中抽出表面温度温度数值
Function Treal {
    Parameter stro. //输入参数，例如 " 310 / 2000 K +0 K/s"×
    Set sn to stro:Find("/"). //找到分隔符 "/" 的位置×
    Set str to stro:SubString(0,sn). //取从头开始的长度为sn的 " 310 "×
    Set str to str:Trim. //去除头尾空格，得到 "310"×
    Set real to str:ToScalar(). //将字符串 "310" 转换成数值 310×
    Return real. //返回表面温度数值
}

//自定义函数，从字符串 "xxxxxxxx" 中抽出极限温度数值
Function Tlimit {
    Parameter stro. //输入参数，例如 " 310 / 2000 K +0 K/s"×
    Set sn1 to stro:Find("/"). //找到分隔符 "/" 的位置×
    Set sn2 to stro:Find("K"). //找到分隔符 "K" 的位置
    //取从 "/" 之后的空格开始的长度为sn2-sn1的 " 2000 "×
    Set str to stro:SubString(sn1+1,sn2-sn1-1).
    Set str to str:Trim. //去除头尾空格，得到 "2000"×
    Set real to str:ToScalar(). //将字符串 "2000" 转换成数值 2000×
    Return real. //返回取到的表面温度数值
}

Set pt to Ship:PartsTagged("qwerty") [0]. //pt 是风帽部件
//tp是温度Module。为了以后的泛用性就直接写具体字符串了
Set tp to pt:GetModule("PartTemperatureState").
//正常情况下内部温度是"core temperature"，表面温度是" skin temperature"
//但其实 skin 前面不是空格而是一个特殊字符，所以打不出来。
//幸好此部件的 tp>AllFieldNames[1] 可以当作 GetField 参数
//因为其内容正好和 Field 名对应的
Set sk to tp>AllFieldNames[1].
//获取表面温度字符串报文的用法就是 Tp:GetField(sk)
Set Tlim to Tlimit(Tp:GetField(sk)). //Tlim 就是风帽部件的极限温度

Wait 0.5. Set thr to 1. SAS On. Wait 0.5.
Lock Throttle to thr. //之后调整节流阀只需要给 thr 赋值
Wait 0.5. Stage. Wait 0.5. //火箭点火
Lock tc to Treal(Tp:GetField(sk)). //tc 是表面温度数值
Set tc0 to tc. Set t0 to Time:Seconds. //赋微分项前值
Wait 0.5. //等待一会儿

//引擎关机之前一直循环
Until Ship:MaxThrust = 0 {
    Set tc to Treal(Tp:GetField(sk)). //tc 是表面温度数值×
    Set Err to Tlim-100-tc. //实际温度和 极限温度-100开 之间的误差×
    Set Ditem to (tc-tc0)/(Time:Seconds-t0).
    Set thr to 0.02*Err-0.1*Ditem. //负反馈，用了 PID 中的 PD×
    Set thr to Max(Min(thr,1),0). //并且thr的范围应在 0.0~1.0 之间×
    Set tc0 to tc. Set t0 to Time:Seconds. //微分项前值刷新×
    Wait 0.001. //等待下一个物理帧
}

```

下图是程序运行时风帽温度维持在 1900 开的截图。

可以看到我们的程序起作用了，在速度和热量之间取到了平衡。



(4) 控制翼的翻动（KSP Field 写入）



KSP 中所有的控制翼都可以手动控制翻起，允许调整翻起角度。
只需要玩家在部件右键菜单里点开 Deploy 后拉动角度限制滑块。
这帮了我们的忙，我们只需要在部件右键菜单里点开 Deploy，
然后拉动角度限制滑块，即可完成单片机翼的控制。用 kOS。



首先我们要在车间里取消水平尾翼的部件对称性，让他们成为各自独立的部件。

左边的尾翼取标签名叫 LW，右边的尾翼取标签名叫 RW。

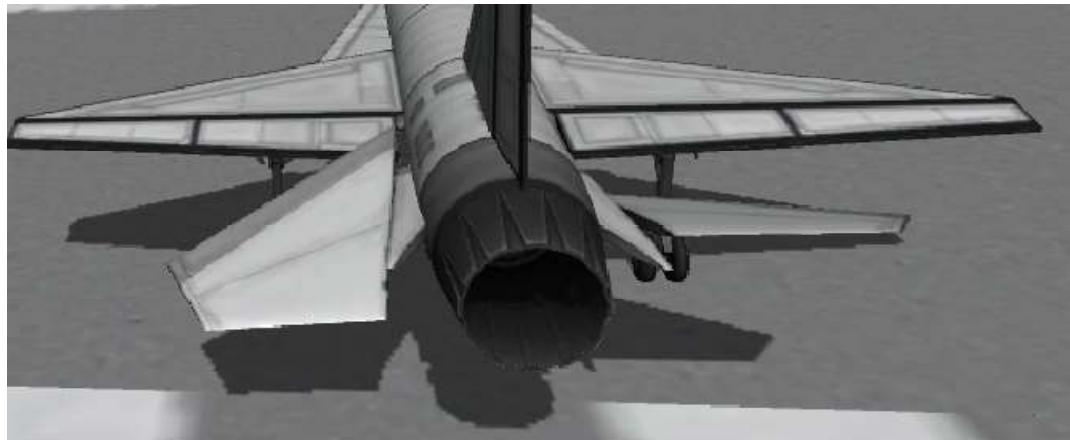
我们的任务是将左边的水平尾翼向上翻最大角度，右边的向下翻最大角度的一半。

```
Set L1 to Ship:PartsTagged("LW") [0]. //L1 是左边的水平尾翼
Set R1 to Ship:PartsTagged("RW") [0]. //R1 是右边的水平尾翼

Set L2 to L1:GetModule ("ModuleCOnrolSurface").//L2 是左边的机翼Module
Set R2 to R1:GetModule ("ModuleCOnrolSurface").//R2 是右边的机翼Module

//展开左右尾翼
L2:SetField("deploy",True). R2:SetField("deploy",True).
//设置左右尾翼展开角度
L2:SetField("authority limiter",100). R2:SetField("authority limiter",-50).
```

结果如下图。



*注1：本实例实现了对每片机翼的独立控制，玩家可以据此编写飞控程序。

5.12

5.12 生涯模式科技树限制

===== 点击以返回目录 =====



kOS 为生涯模式里的科技树限制设置了生涯结构体Career。

结构体类型：生涯结构体Career

结构体成员	类型	读写	说明
CanTrackObjects	Boolean	只读	是否解锁追踪小行星了
PatchLimit	number	只读	是否解锁路径预测了
CanMakeNodes	Boolean	只读	是否解锁轨道变轨点了
CanDoActions	Boolean	只读	是否解锁动作组了
.....

```
//用法示例:
```

```
Career () :PachLimit
Career () :CanDoActions
```

6.

6. kOS 语言

6.1

6.1 kOS 语言特性

6.1.1

6.1.1 大小写不敏感

===== 点击以返回目录 =====



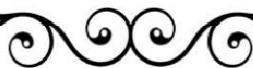
kOS 的编程语言叫 KerboScript。他对代码的大小写不敏感，
即使是文件路径、字符串、也是大小写视作等同。

```
"hello" = "HeLlo" //这个比较的返回结果是 True
```

6.1.2

6.1.2 表达式

===== 点击以返回目录 =====



kOS 支持用户自写表达式。

用户可以使用自定义变量和系统预设变量。支持的变量类型如下：

(1) 数值 Scalar

三个基本变量之一。

实数数值（标量）Scalar，分为整型和浮点型两种。

当整型和浮点型共同运算时，会全部转换成浮点型再运算。

```
Set x to 4 + 2.5.
Print x. //显示 6.5
```

(2) 字符串 String

三个基本变量之一。

字符串 String 可用于在指令窗口显示输出。

当字符串通过加号+与其他类型共同运算时，会全部转换成字符串再运算。

```

Set a to "qwert".
print a. //显示 qwert

print "Hello World". //显示 Hello World

Print "4 plus 3 is:"+ (4+3) +" DaZe". //显示 4 plus 3 is:7 DaZe

```

(3) 逻辑量 Boolean

三个基本变量之一。

只有真 (True) 假 (False) 两个值。

```

set myValue to (x >= 10 and x <= 99).
if myValue {
    print "x is a two digit number.".
}

```

(4) 结构体类型

除了数值 Scalar、字符串 String、逻辑量 Boolean 这三种基本量之外，还有很多其他的类型，都是结构体变量的形式。

他们通常包含多种变量的成员，可以是数值 Scalar、字符串 String、逻辑量 Boolean，也可以是其他结构体，还可以是结构体函数（方法）。

*注1：严格来说基本变量类型也算结构体，kOS 中所有变量类型都是结构体类型。
但玩家最好将这三个基本变量类型和其他结构体区分开来，

因为这三种变量直接携带信息，
而其他的结构体类型都是通过这三种变量间接携带信息的。

(5) 结构体成员

结构体下设多个成员，

大致可分为三种：基本变量、结构体变量、结构体函数（方法）。

结构体和结构体成员之间使用冒号 : 连接。

```

List Engines in En.
Set En1 to En[0]. //En1 是一个引擎结构体变量

Set En1:ThrustLimit to 50. //结构体下的 基本变量，引擎的推力限制在50%
Set b to En1:Gimbal. //结构体下的 矢量喷口 结构体变量,
print En1:ISPAt(50). //有参数的 结构体函数，引擎En1 在 50% 大气压下的比冲
En1:Activate(). //无参数的 结构体函数，启动引擎

```

6.1.3

6.1.3 逻辑短路

===== 点击以返回目录 =====



有一种叫 逻辑短路（Short-circuiting booleans）的技巧，可以用来降低计算量。

当 kOS 遇到诸如 If A or B、或 If A and B 的判断式时，
如果 表达式A 的值足够判断整个 If 条件的真假时，
那么 表达式B 就不会被执行。

```
set x to true.
if x or y+2 > 10 {
    print "yes".
} else {
    print "no".
}. //本例中 y+2 没有被执行。
```

6.1.4

6.1.4 自定义变量的自动类型

===== 点击以返回目录 =====



玩家在 kOS 中可以创建自定义变量。

自定义会根据所赋值的类型改变自身的类型。

```
Set a to 12345. //现在 a 是 数值类型 Scalar
Set a to True. //现在 a 是 逻辑类型 Boolean
Set a to "qwert". //现在 a 是 字符串类型 String
```

而对于预设变量、结构体下的变量成员，
他们的类型被 kOS 锁定，只能赋对应类型的值，否则会报错。

6.1.5

6.1.5 全局变量隐式声明

===== 点击以返回目录 =====



kOS 中自定义变量分全局变量和局部变量两种。kOS 支持全局变量的隐式声明。

这样自定义变量可以不事先声明，而是在首次使用时自动声明。

这样是为了体查新手玩家，让他们觉得 kOS 使用方便。

```
Set a to 5.4321.
```

有经验的程序不喜欢这样，因为在复杂程序中很容易因为写错变量名而程序出错，
而且这种出错不是语法错误，kOS 的语法检查功能并不能发现这种错误。

这时候我们可以用 @LAZYGLOBAL OFF. 指令，来关闭这项功能。

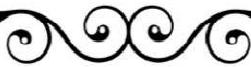
这样使用变量之前就强制需要变量声明指令了。

```
@LazyGlobal Off.
Global a is 5.4321.
```

6.1.6

6.1.6 自定义函数

===== 点击以返回目录 =====



kOS 支持用户自定义函数。

例如下面是一个将角度数值转成弧度数值的用户自定义函数：

```
DECLARE FUNCTION DEGREES_TO_RADIANS {
    DECLARE PARAMETER DEG.

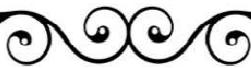
    RETURN CONSTANT():PI * DEG/180.
}.

SET ALPHA TO 45.
PRINT ALPHA + " degrees is " + DEGREES_TO_RADIANS(ALPHA) + " radians.".
```

6.1.7

6.1.7 结构体

===== 点击以返回目录 =====



结构体类型包含不止一段信息（成员），玩家可以用冒号：来访问结构体下的成员。

结构体的成员包含三种：基本变量、结构体变量、结构体函数（方法）。

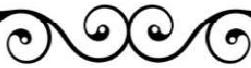
冒号之后的被称为后缀。例如下面：

```
Set myCraft To SHIP. //载具结构体Vessel
Set myMass To myCraft:MASS. //载具质量，数值Scaler
Set LD To Ship:LoadDistance. //载具的物理载入距离设置，载入距离结构体LoadDistance
Set myThr To myCraft:MaxThrustAT(0.5). //在0.5个大气压下的载具最大推力，结构体函数
```

6.1.8

6.1.8 触发器

===== 点击以返回目录 =====



kOS 有一个很有用的功能叫触发器。kOS 会在每个物理帧内检查触发条件，如果满足条件就会暂停主程序代码，运行触发结构内的代码，运行完之后控制权会交还给主程序代码。

```
WHEN ship:altitude > 50000 then {
    ag1 on.
} //当载具高度超过50km时，启动动作组1
```

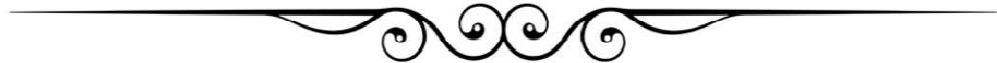
6.2

6.2 语法

6.2.1

6.2.1 一般规则

===== 点击以返回目录 =====



(1) 空格

kOS 编译时会把空格、制表符、换行符这三个视作同一种符号。
就算他们连续出现，在 kOS 眼里也只是一次空格。
玩家可以随意使用他们进行排版，这不影响 kOS 的运行效果。

(2) 句点

一句 kOS 语句以句点 . 作为结尾的标志。

(3) 算术运算符

```
+ - * / ^ e ( )
```

(4) 逻辑运算符

```
not and or true false <> >= <= = > <
```

(5) 指令和关键词

((1)) 变量声明和检查类

指令和关键词	说明	用法
declare	声明变量时的关键词	详见 变量和声明
local	声明局部变量时的关键词	
global	声明全局变量时的关键词	
defined	检查变量是否存在，返回逻辑值	
.....		

((2)) 变量赋值类

指令和关键词	说明	用法
set、to、is	赋值语句。 将 B 的值赋给 A。	<code>set A to B.</code> 或 <code>set A is B.</code>
unset	解除 set 赋值, 清空定义和内存	<code>unset A.</code>
lock、to、is	锁定语句。 每次需要 A 的值, 就重新计算一遍 B	<code>lock A to B.</code> 或 <code>lock A is B.</code>
unlock	解除 lock 赋值, 清空定义和内存	<code>unlock A.</code>
all	和 unset 或 unlock 联用, 解除所有变量的定义和内存	<code>unset all.</code> 或 <code>unlock all.</code>
on	把变量赋值成逻辑值 true	SAS on.
off	把变量赋值成逻辑值 false	SAS off.
toggle	翻转逻辑值变量的值 <code>true ↔ false</code>	toggle SAS.
.....

(3) 顺序控制类

指令和关键词	说明	用法
if、else	条件结构关键词，尖括号内可以是多句语句	<code>if A { B. } 或 if A1 { B1. } else A2 { B2. }</code>
choose	可作为表达式的条件指令结构	<code>choose B1 if A else B2</code>
until	until 循环结构关键词，	<code>until A { B. }</code>
for、in	for 循环结构关键词，专用于 list 结构体内的循环	<code>for A in listB { C. }</code>
from、until、step、do	from 循环结构关键词句，四个词后面的语句分别是赋初值、循环结束条件、更新循环标签、循环体	<code>from {A.} until {B.} step {B.} do { C.}</code>
break	跳出一层循环	<code>set x to 1. until 0 { set x to x + 1. × if x > 10 { break.}}</code>
wait、until	等待指令。等一定的描述时间，或者等到条件满足位置	<code>wait 4.5. 或 wait until A > B</code>
when、then	when 触发结构关键词。每个物理帧检查一次触发条件，如果满足则运行一次触发体，然后清除自身触发器	<code>when A then { B. }</code>
on	on 触发结构关键词。每个物理帧检查一次触发条件，如果结果从 false 翻转为 true，则运行一次触发体，然后清除自身触发器	<code>on AG1 { print "ActGruop_1". }</code>
perserve	[不推荐用] 用在触发体里，当触发体运行完之后，为触发器续命一次，阻止清除自身触发器	<code>when A then { B. perserve. }</code>
return	用在触发体里，遇到此指令即结束触发体的运行。如果返回 true 值，可以为触发器续命一次，阻止清除自身触发器。	<code>when A then { B. return true. }</code>
.....		

(4) 自定义函数类

指令和关键词	说明	用法
function	用于声明函数	<code>function A {</code>
parameter	用于声明函数 / 程序的参数	<code>parameter B.</code>
return	用于确认函数的返回值	<code>return B + 1. }</code>
.....		

(5) 指令和功能类

指令和关键词	说明	用法
stage	发射序列指令。相当于按空格键	stage.
add	把变轨规划添加到飞行轨道中	set A to node (A, B, C, D). add A. remove A.
remove	从飞行轨道上移除变轨规划	
print、at	在命令窗口显示文本信息， at 是用于行列数定位的可选项， print 的内容会强制转为字符串再显示	print 1 + 1. print (3 - 1) + " is two ". print " qwert " at (0, 10).
clearscreen	对指令窗口内容进行清屏	clearscreen.
reboot	重启kOS系统	reboot.
shutdown	关闭kOS系统	shutdown.
.....		

(6) 列表操作类

指令和关键词	说明	用法
list、file、volume	在指令窗口中显示所有文件的清单 在指令窗口中显示所有可访问卷的清单	list files. list volumes.
list	将当前载具上的所有PN的列表存到B 将载具A上所有PN的列表存到B PN可以是系统预设变量中任何预设列表	list parts in B. list parts from A to B.
.....		

(7) 文件操作类

指令和关键词	说明	用法
switch、volume	切换卷	switch volumt to 0. switch volumr to archive.
edit	打开文件编辑窗口， 如果没有文件则新建	edit A.
delete	[不推荐] 删除当前卷中的文件A	deltet A.
copy	[不推荐] 从 / 向另一个卷中拷贝文件A过来 / 过去	copy A to 1. copy A from 0.
rename	[不推荐] 将文件A重命名为B	rename file A to B.
compile	[不推荐] 编译源文件A 将源文件A编译成文件名B	compile A. compile A to B.
run、once	[不推荐] 运行文件A (不用once) 如果之前没运行过A才运A (用once)	run A. run once A.
log	将文本信息记录在文本文件上 内容会强制转为字符串再记录 每次运行结束后都会自动加回车号	log A + ", " + B to data.csv.
.....		
说明：不推荐使用 delete 、 copy 、 rename 、 compile 、 run 、 once 指令， 推荐使用对应他们功能的函数（详见文件I/O）		

(6) 其他符号

```
{ } [ ] , : // .
```

*注1: <>是指不等于; {}用于划出代码块; []用于括起数组List的下标;

冒号: 用于连接结构体和其成员; // 用于标记注释文本。

(7) 标识符

kOS 中自定义变量的标识符 (变量名) 要符合以下规则:

由字母、数字、下划线组成, 并且变量名的第一个字符必须是字母或下划线。

标识符 (变量名) 不区分字母的大小写。

(8) 不分大小写

kOS 对所有的代码都不分大小写, 无论是关键词、变量名、字符串, 还是其他的。

例如, “He|Lo” 和 “HeLL0” 会被 kOS 认为是相等的字符串。

(9) 结构体后缀

结构体变量和其成员之间用冒号:连接。

冒号表示调用结构体的成员, 冒号可以嵌套使用。

```
list parts in myList.  
print myList:length. // length is a suffix of myList  
  
print ship:velocity:orbit:x.
```

6.2.2

6.2.2 数值

===== 点击以返回目录 =====



kOS 中的数值都是实数标量 (Scalar)。

存储记数时全部存为32位整数或64位浮点数, 二者间自动切换。

64位浮点数时具有15~16位有效数字。

有关书写的规则:

书写时允许在首位以外的任意位置加下划线来表示位数分割,

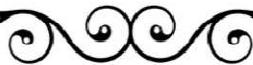
书写时允许使用工程记数法。如下写法全部都是合法的:

```
12345678  
12_345_678  
12345.6789  
12_345.6789  
-12345678  
12.123e12  
1.234e-12
```

6.2.3

6.2.3 代码块

===== 点击以返回目录 =====



使用尖括号{}扩起单句或复数语句句，这就构成了代码块。

代码块有两个作用，

第一是能在语法解释时，让多语句被当做一句来看待。

这在顺序控制语句和函数声明语句中很重要。

```
if x = 1
    print "it's 1".
```

```
if x = 1 { print "it's 1". print "yippieee.". }
```

```
if x = 1 {
    print "it's 1".
    print "yippieee.".
}
```

第二个是代码块限定了局部变量的有效范围。

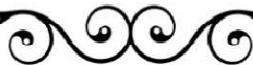
有关有效范围可参考 变量的有效范围

```
declare x to 3.
print "x here is " + x.
{
    declare x to 5.x
    print "x here is " + x.
    {
        declare x to 7.x
        print "x here is " + x.
    }
}
```

6.2.4

6.2.4 内置函数

===== 点击以返回目录 =====



kOS 中一共有三种形式的函数：内置函数、结构体函数、自定义函数。

kOS 的内置函数是预设的，从数学计算到载具操控范围都有涉及。

内置函数可以直接通过函数名来调用。

```
print MAX(12, 2).
print SIN(45).
```

另外有些内置函数是不要求参数的。书写时也可以直接连括号都省掉。

```
CLEARSCREEN.  
CLEARSCREEN().
```

6.2.5

6.2.5 结构体函数(方法)

===== 点击以返回目录 =====



kOS 中一共有三种形式的函数：内置函数、结构体函数、自定义函数。

结构体函数是结构体变量的成员。要以后缀的形式调用。

```
set x to ship:partsnamed("rtg").  
print x:length().  
x:remove(0).  
x:clear().
```

6.2.6

6.2.6 字典结构的后缀用法

===== 点击以返回目录 =====



有一种叫 Lexicon（字典结构）的特殊结构体。

允许玩家编辑字典内“词条序列”的内容，并支持形如结构体一样的冒号后成员形式调用。

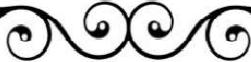
例子：

```
// Given this setup...  
set MyLex to Lexicon().  
MyLex:ADD( "key1", "value1").  
// ...these two lines have the same effect:  
print MyLex["key1"]. // key used in the usual way as an "index".  
print MyLex:key1. // key used in an alternate way as a "suffix".
```

6.2.7

6.2.7 自定义函数

===== 点击以返回目录 =====



kOS 中一共有三种形式的函数：内置函数、结构体函数、自定义函数。

自定义函数的内容行为都可以由玩家自行编写，调用方法和内置函数相同。

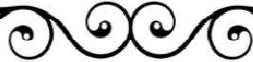
自定义函数比内置函数多了函数声明部分的结构。

有关自定义函数，详见 [自定义函数](#)。

6.2.8

6.2.8 预设的特殊变量

===== 点击以返回目录 =====



有一些特殊的标识符(变量名), 他们已经在 kOS 里绑定了定义和类型, 其值和 KSP 的游戏元素挂钩, 对这些值的修改等同于在游戏里进行控制操作。这些变量/结构体的值都是玩家能在 kOS 编程里用上的, 玩家在声明和使用变量的时候应该避开这些变量名, 否则会出错。有关这些变量, 请参阅 系统预设变量。

6.2.9

6.2.9 不存在自定义结构体

===== 点击以返回目录 =====



受机能限制, kOS 中所有结构体类型都是预设的, 固定的。kOS 无法自定义结构体类型。

6.3

6.3 顺序控制

6.3.1

6.3.1 数值锁定和解锁

===== 点击以返回目录 =====



锁定赋值 lock: 每次要调用变量值时, 都重新计算一遍表达式。

Lock 指令用法: Lock A to B.

(其中 A 为变量名、B 为表达式)

解除锁定赋值 unlock。

UnLock 指令用法: UnLock A.

(其中 A 为变量名)

*注1: Lock 指令所涉及的变量都必须为全局变量。

*注2: 对于 节流阀Throttle、把舵Steering 这些飞行关键变量。

kOS 会在每个物理帧都查询一遍他们的值, 对应的表达式会在每个物理帧都更新。

例子:

```

Set X TO 1.
Lock Y TO X + 2.
PRINT Y.          // Outputs 3
Set X TO 4.
PRINT Y.          // Outputs 6

Lock Y TO X + 2.
print "Y's Locked value is " + Y(). //因为Lock其实是个函数，所以也可以加上括号用的

UNLock X.
UNLock ALL. //这是个特殊的命令，指解锁全部Lock

```

6.3.2

6.3.2 条件结构

===== 点击以返回目录 =====



kOS 中条件结构的关键词是 If 和 Else。

用法:

```
IF A {
  B.
}
```

(其中 A 是判断式，B 是语句)

例子:

```

Set X TO 1.
IF TURE {print"piupiupiu"}.//条件语句最后的句号.可有可无
IF TURE
  print"piupiupiu". //和上面一句一个意思
IF X = 0 {
  PRINT "zero".
} ELSE IF X < 0 {
  PRINT "negative".
} ELSE {
  PRINT "positive".
}

```

*注1: kOS 的数值量转换成逻辑量时，0 值对应 False，非 0 值对应 True。

*注2: If 后面的 条件式不能用括号括起来，否则会报错。

比如这样是不行的 if (2<3)

*注3: 条件结构的尖括号内部算作一个代码块的内部。

6.3.3

6.3.3 选择表达式

===== 点击以返回目录 =====



Choose (选择表达式) 是一个根据条件二选一的指令，和 If语句结构相似。

Choose 结构用法:

```
CHOOSE expression1 IF condition ELSE expression2
```

和If的区别是， Choose 有返回值，可作为表达式使用。

例子：

```
Set y to CHOOSE x IF x>0 ELSE -x.           //绝对值 y=Abs(x)
PRINT CHOOSE "High" IF altitude > 20000 ELSE "Low". //判断高低
```

6.3.4

6.3.4 循环结构

===== 点击以返回目录 =====



kOS 中有三种循环结构： Until 循环、 For 循环、 From 循环。

*注1： 循环结构的尖括号内部算作一个代码块的内部。

(1) Until 循环

Until 循环 是一种常用的、 使用较简单的循环类型。

用法：

```
Until A {
  B.
}
```

(其中 A 是停止循环的条件式， B 是单据或多句执行语句)

例子：

```
Set X to 1.
UNTIL X > 10 { // Prints the numbers 1-10
  PRINT X.
  Set X to X + 1.
}
```

(2) For 循环

For 循环 专用于遍历列表List，并对列表List 里的成员做统一操作。

用法:

```
FOR A in B {
    c.
}
```

(其中 A 是形式变量名, B 是列表, c 是单据或多句执行语句)

例子:

```
PRINT "Counting flamed out engines:".
Set numOUT to 0.
LIST ENGINES IN myList.
FOR eng IN myList {
    IF eng:FLAMEOUT {
        Set numOUT to numOUT + 1.
    }
}
PRINT "There are " + numOUT + "Flamed out engines.".
```

(3) From 循环

From循环 是功能最为强大的一种循环类型。

用法:

```
FROM {A.} UNTIL B STEP {c.} DO {
    d.
}
```

(其中 A 是用于赋初值的语句, B 是停止循环的条件式,
c 是每次循环结束用来更新循环变量的语句, d 是循环体的执行语句)

例子:

```
FROM循环
FROM {Local x is 10.} UNTIL x = 0 STEP {Set x to x-1.} DO {
    print "T -" + x.
}
```

(4) 跳出循环 (Break)

Break 可以用在任何一种循环中, 用于跳出并结束循环。

例子:

```
Set X to 1.
UNTIL X > 10 { // Prints the numbers 1-10
    PRINT X.
    Set X to X + 1.
    if (x=4) {
        break. // 跳出当层循环
    }
}
```

*注2: Break 跳出的是最内层的循环。
例如对于两层循环嵌套而成的, Break 只能跳出里层的循环。

6.3.5

6.3.5 等待语句

===== 点击以返回目录 =====



等待Wait: 将主线暂停挂起, 直到经过了一段时间之后再恢复,
在此过程中触发线不受影响, 正常工作。

等待语句Wait 一般有几种用途:

- (1) 控制主线的进程, 确保主线程序不过早执行后续指令。
- (2) 用在主线的循环语句中, 防止循环频繁发生。
- (3) Wait 用在触发结构中也会有效, 但要注意他可能会阻滞程序运行。

例子:

```
Wait A. //程序暂停等待A秒钟
Wait 0.001. //程序等待一个物理帧
wait True. //程序等待一个物理帧

wait until Ship:Altitude<3000. //等到飞船高度低于 3km 时再执行后续语句
wait until false. //程序永不终止
```

*注1: kOS 遇到 Wait 语句时, 总会等到下一个物理帧再计算后面的等待时间。
所以无论 Wait 后面跟的是多久, 时间总是至少会等待一个物理帧。

*注2: 想要结束 wait until false. 的程序、或者其他想要结束的程序,
可以在命令窗口按快捷键 Ctrl + C 来强制结束当前程序。

6.3.6

6.3.6 逻辑操作符

===== 点击以返回目录 =====



逻辑操作符常用于条件语句的条件判断式中。
注意不等号的写法是 <>

```
= < > <= >= <>
AND OR NOT
```

例子:

```

IF X = 1 AND Y > 4 { PRINT "Both conditions are true". }
IF X = 1 OR Y > 4 { PRINT "At least one condition is true". }
IF NOT (X = 1 or Y > 4) { PRINT "Neither condition is true". }
IF X <> 1 { PRINT "X is not 1". }
SET MYCHECK TO NOT (X = 1 or Y > 4).
IF MYCHECK { PRINT "mycheck is true." }
LOCK CONTINUOUSCHECK TO X < 0.
WHEN CONTINUOUSCHECK THEN { PRINT "X has just become negative.". }
IF True { PRINT "This statement happens unconditionally." }
IF False { PRINT "This statement never happens." }
IF 1 { PRINT "This statement happens unconditionally." }
IF 0 { PRINT "This statement never happens." }
IF count { PRINT "count isn't zero.". }

```

6.3.7

6.3.7 函数声明

===== 点击以返回目录 =====



kOS 允许玩家创建并使用自定义函数，并自由设置返回值。

详情请见自定义函数

6.3.8

6.3.8 触发结构

===== 点击以返回目录 =====



为了实现对载具飞行控制很有帮助的“触发”功能和“挂起等待”功能。kOS程序具有双线运行的特点。

kOS 的运行分为主线（Main-Line）和触发线（Triggers-Line）两个阶段。触发线有权中断主线，而主线不能中断触发线。每个物理帧之间都运行一遍。

kOS在每个物理帧会：

- Step1. 先执行触发线。检查触发条件是否满足，若满足则执行触发体；
- Step2. 再执行主线。若遇到等待命令，且等待条件不满足，则中断挂起。

*注1：触发器里边别放局部变量，要用全局变量

*注2：编程中要精简触发器的使用，不要用触发器干太多事，否则计算量太大。
例如触发结构中最好不要放循环结构，循环结构中最好不要放触发结构。

*注3：kOS 中触发器的定义需要放在死循环语句的前面。
任何情况下，kOS都不会运行死循环语句后面的东西。

*注4：触发器每个物理帧都会轮询一遍触发条件，这会耗计算量，
如果可以请精简触发条件的计算次数。例如下面：

*注5：循环结构的尖括号内部算作一个代码块的内部。

使用触发器相互嵌套来减少轮询压力的实例：
根据高度阶段性调整飞船朝向，任何时候都只有一个触发语句查询着

```
Set head to heading(90,90).
Lock Steering to head.

when Ship:Altitude >1000 then
{
    Set head to heading(90,80).
    when Ship:Altitude >2000 then
    {
        Set head to heading(90,70).
        when Ship:Altitude >3000 then
        {
            Set head to heading(90,60).
            //.....依此类推
        }
    }
}
```

(1) When / Then 触发

When / Then 是最常用的触发结构。触发条件是判断式为 True。

用法：

```
when A then {
    B.
}
```

(其中 A 是判断式，B 是触发的执行语句)

例子：

```
WHEN STAGE:LIQUIDFUEL < 0.01 THEN { //写0.01是浮点值不可能刚好烧到0，总有剩余量的
    PRINT "No liquidfuel. Attempting to stage.".
    STAGE.
}
```

后面的句号都是可有可无的，干脆不加。

(2) On 触发

专为响应“按动作组按键”这类“逻辑值翻转”状态而设计的触发结构，
触发条件是判断式从 False 翻到 True 时。

例子：

```

ON AG1 {
    PRINT "Action Group 1 activated.".
    PRESERVE.
}

ON SAS PRINT "SAS system has been toggled".

```

*注5: On 触发用来监测功能开关很管用。

(3) 触发存续 Preserve. 和 Return True.

通常，触发结构的寿命是一次性的，触发执行语句运行完之后触发器就会被清除。
而触发存续命令 Preserve. 和 Return True. 则可为触发结构续命一次。
至于下次是否还要续命，取决于下次执行时是否还会碰到 Preserve. 或和 Return True.

建议使用 Return 而不要用 Preserve。

两者区别是：Return 是遇到 True 值则立即续命并退出触发体；遇到 False 值则立即退出触发结构（不续命）；Preserve则是等到触发体运行完之后再续命。
在顺序控制上 Return 更为灵活。

Preserve 和 Return 区别的例子：

```

WHEN STAGE:LIQUIDFUEL < 0.001 THEN {
    print "ABC".//此句运行×
    PRESERVE.
    PRINT "DEF".//此句运行
}

WHEN STAGE:LIQUIDFUEL < 0.001 THEN {
    print "ABC".//此句运行×
    return true.
    PRINT "DEF".//此句不运行
}

```

使用 Return 的例子：

```

//只要这一级火箭没燃料了，就显示说没燃料了，然后就按空格进行火箭分级
WHEN STAGE:LIQUIDFUEL < 0.01 THEN {
    PRINT "No liquidfuel. Attempting to stage.".
    STAGE.
    return true.
}

```

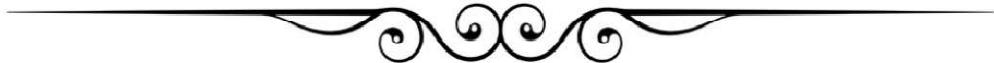
6.4

6.4 变量和声明

6.4.1

6.4.1 变量声明

===== 点击以返回目录 =====



玩家可以在 kOS 中声明自定义变量，声明出的变量有全局变量和局部变量之分。
除声明语句外，赋值命令 **Set** 和 **Lock** 也有声明隐式变量的功能；
但是当关闭变量隐式声明之后，就只有标准的变量声明语句可以用来声明变量了。

(1) 声明的语句

((1)) 声明局部变量（以下三句一个效果）

```
declare identifier to / is expression.  
declare local identifier to / is expression.  
local identifier to / is expression.
```

例子：

```
Declare a is 3.45.  
Declare Local c to "abc".  
Local b is False.
```

((2)) 声明全局变量（以下两句一个效果）

```
declare global identifier to / is expression.  
global identifier to / is expression.
```

例子：

```
Declare Global a is 3.45.  
Global b is False.
```

((3)) 用赋值命令来隐式声明（声明出的变量是全局变量）

```
set identifier to / is expression.  
lock identifier to / is expression.
```

例子：

```
Set a to 3.45.  
Lock b to a+3.
```

玩家也可以用 **@LazyGlobal off**. 来禁用变量隐式声明，
此句应写在程序开头，此句的效果是：
所有变量使用前都要明确声明；所有隐式声明的变量，全都会报错说变量是未定义的。

```
@LazyGlobal off
```

*注1：只写 **Declare** 不写是 **Local/Global** 的声明代表声明局部变量。

*注2: kOS 支持全局变量的数量是有限的, 请不要滥用。

(2) 有效范围

有关局部变量、全局变量、有效范围的概念, 详见 变量的有效范围

(3) 赋初值问题

kOS 声明变量的时候必须为变量赋初值。

6.4.2

6.4.2 程序文件的参数

===== 点击以返回目录 =====



(1) 声明程序的参数

kOS 允许玩家为程序文件设置参数（输入参数），
玩家可以像声明和调用自定义函数样调用外部程序。

test1 文件内容:

```
Declare Parameter x.  
Parameter y. //这两种参数声明方法都是可以的  
Print "X * Y = " + x*y.
```

test2 文件内容:

```
Set a to 3.  
Run test1(a,a+1).  
  
//把这两个文件都放在当前 Volume 下,  
//程序运行结果会输出: X * Y = 12
```

声明程序参数的时候也可以多个变量写在一句:

Parameter x,y.

*注1: 程序的参数都是局部变量, 作用范围是程序文件内部。

*注2: 原则上, 只要参数声明语句比使用语句靠前, 程序就能正常工作,
不过最好还是所有参数声明语句都写在程序头部。

(2) 为程序参数设置默认值

kOS 允许玩家调用外部程序文件时不给全部的程序参数赋值,
就是说, 对于有 N 个程序参数的程序文件,
调用他的时候, 可以只给出 M 个参数的值。 ($M \leq N$)

未给出值的程序参数，会自动设成默认值。

test1 文件内容：

```
Parameter P1.
Parameter P2, P3 is 0, P4 is "cheese".
print P1 + ", " + P2 + ", " + P3 + ", " + P4.
```

test2 文件内容：

```
Run test1(1,2).           // prints "1, 2, 0, cheese".
Run test1(1,2,3).         // prints "1, 2, 3, cheese".
Run test1(1,2,3,"hi").    // prints "1, 2, 3, hi".
RunPath(test1,1,2).       // prints "1, 2, 0, cheese".
RunPath(test1,1,2,3).     // prints "1, 2, 3, cheese".
RunPath(test1,1,2,3,"Hi"). // prints "1, 2, 3, hi".
```

声明程序参数默认值时，

没有默认值的程序参数要先声明，有默认值的程序参数要后声明。

所以像下面这种写法是不工作的：

```
DECLARE PARAMETER thisIsOptional is 0,
          thisIsOptionalToo is 0.
          thisIsMandatory.
```

6.4.3

6.4.3 查询变量是否定义

===== 点击以返回目录 =====



用法：

Defined A. // 查询变量 A (标识符) 是否有定义，返回 True 或 False 值
真 假

6.4.4

6.4.4 Set 语句

===== 点击以返回目录 =====



(1) Set 语句

Set语句用于给变量赋值，或者隐式的创建（声明）一个全局变量。

用法：

```
set A to / is B.  

(其中 A 是变量名, B是表达式)
```

例子：

```
Set x to 1.  
Set y to x+1.
```

Set 也可以为逻辑变量赋值。逻辑变量也可以用on, off, toggle命令进行赋值。

例子：

```
Set a To True.  
Set k To False.  
a Off.           //a=false  
k On.            //k=true  
Toggle k.        //k=false
```

(2) Set 赋值和声明赋初值的区别

举例说明，如下：

```
Set X To 1.  
Declare Local X To 1.  
Declare Global X To 1.
```

Set语句执行的动作是：

- Step1. 查找是否存在局部变量X，如果存在，将其赋值为1；
- Step2. 如果上一步里没找到，那么在当前层（代码块）的外层再试一次，
如果还没找到，在再外层一层试一次，依次……；
- Step3. 如果所有层都没有局部变量X，也没有全局变量X，那么，
新建（声明）一个全局变量X，并将它赋值为1.。

*注1：上面这段话建议在阅读完 XXXXXXXX 之后再回来看一遍。

Declare Local 语句执行的动作是：

- Step1. 在本层内创建（声明）一个局部变量X，将其赋初值为1。

Declare Global 语句执行的动作是：

- Step1. 创建一个全局变量X，并将其赋初值为1。（无论是否存在本地变量X）

6.4.5

6.4.5 Unset 语句

===== 点击以返回目录 =====



Unset语句用于删除已存在的用户自定义变量。（系统预设变量是无法删除的）

用法：

```
unset A.
```

无论要删除的变量是否存在，都不会影响Unset语句的运行，此时Unset语句不会给提示。

如果同时存在多个不同有效范围的同名变量，Unset会删除有效范围最小的那个变量。

6.4.6

6.4.6 变量 Lock 赋值

===== 点击以返回目录 =====



(1) Lock 赋值语句

锁定赋值 lock: 每次要调用变量值时，都重新计算一遍表达式。

Lock 指令用法: Lock A to B.

(其中 A 为变量名、B 为表达式)

解除锁定赋值 unlock。

UnLock 指令用法: UnLock A.

(其中 A 为变量名)

*注1: 由 Lock 指令赋值的变量都会强制变成全局变量。

*注2: Lock 语句里表达式所涉及的变量必须都是全局变量，否则会出错。

*注3: 对于节流阀Throttle、把舵Steering 这些飞行关键变量。

kOS 会在每个物理帧都查询一遍他们的值，对应的表达式会在每个物理帧都更新。

Lock 和 Unlock 指令的使用例子:

```

Set X TO 1.
Lock Y TO X + 2.
PRINT Y.          // Outputs 3
Set X TO 4.
PRINT Y.          // Outputs 6

Lock Y TO X + 2.
print "Y's Locked value is " + Y(). //因为Lock其实是个函数，所以也可以加上括号用的

UNLock X.
UNLock ALL. //这是个特殊的命令，指解锁全部Lock

```

(2) Lock 和 Set 的区别

举例说明 Lock 和 Set 指令的区别:

```

set a to 1.
set b to a+1.
set a to 2.
print "a="+a. //a=2
print "b="+b. //b=2

set a to 1.
lock b to a+1.
set a to 2.
print "a="+a. //a=2
print "b="+b. //b=3

```

(3) 跨文件调用

对全局变量的 Lock赋值 可以实现跨文件调用。如下：

test1 文件内容：

```

run File2.
print "x's locked value is " + x. //此语句不工作
print "x's locked value is " + x(). //此词语工作，以函数形式调用

```

test2 文件内容：

```
lock x to "this is x". //创建全局变量x
```

(4) 局部 Lock

有一种可以用于局部变量的 Lock 指令，
特点是变量和所有的行为，只能在某一层（代码块）内存在。

用法：

Local Lock A To B.
(其中 A 是局部变量的变量名， B 是表达式)

6.4.7

6.4.7 逻辑量的操作

===== 点击以返回目录 =====



(1) Toggle 指令

将 True 值变量翻转成 False 值，或者将 False 值变量翻转成 True 值。

```

Toggle Ag1.
Toggle SAS.

```

注意，对于数值量，0 意味着 False 值，非0 意味着 True 值。

(2) On 指令

将变量值设成逻辑值 True。

RCS On.

注意，当 On 指令作用在一个 数值或字符串变量 A 上，A 的值还是会变成 True

(3) Off 指令

将变量值设成逻辑值 False。

RCS Off.

注意，当 Off 指令作用在一个 数值或字符串变量 A 上，A 的值还是会变成 False

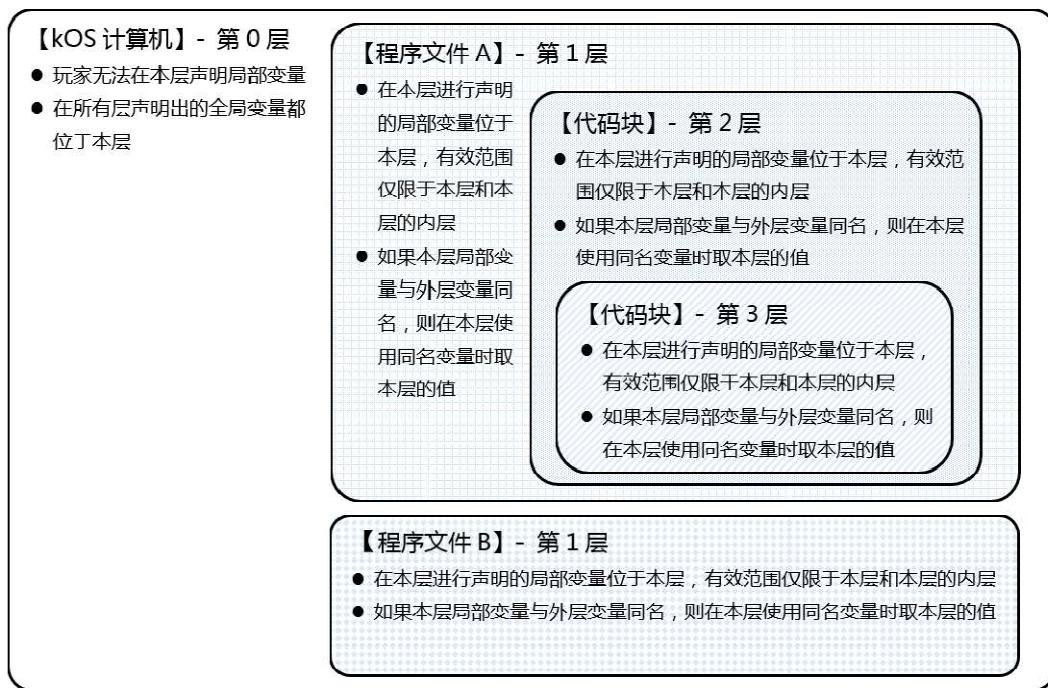
6.4.8

6.4.8 变量的有效范围

===== 点击以返回目录 =====



不管是哪种变量，他们的有效范围的规则都遵循下图：



上图总结下来就是以下四条规则。

kOS计算机、程序文件、代码块（代码块指尖括号括起的范围，包但不限于：

自定义函数、触发结构、循环结构、条件结构），

他们有对应范围。这些范围构成树状嵌套关系。

树状嵌套关系的顶层是 kOS 计算机，接下来是 程序文件，再之后是 代码块。

全局变量可以在任意位置声明，但视作位于树状嵌套关系的顶层。
某一层的局部变量只能在该层声明。

有效范围指该变量能在哪里使用。
位于某层的变量的有效范围为该层及其所有内层(包含嵌套的层)。
但是有例外，如果某层的变量与内层变量同名，
则在该内层，某层变量无法使用，而该内层变量可以使用。

所有系统预设变量都是全局变量。

来看几个例子。

例子1：

test1 文件内容：

```
Run test2.  
Print a. //这句会报错说变量未定义  
Print b. //这句会正常工作
```

test2 文件内容：

```
Local a is 1.  
Global b is 2.
```

例子2：

```
set A to 1.  
print"A="+A. //输出 A=1  
{  
    local A to 2.  
    print"A="+A. //输出 A=2  
    {  
        local A to 3.  
        print"A="+A. //输出 A=3  
    }  
    set A to 4. //将当前层级和所有层级的变量A都设为4，此时上面那个本来输出3的还是3  
    print"A="+A. //输出 A=4  
    lock A to 5. //将所有名叫A的变量，强制转换为同一个全局变量A，值是5，  
                //此时上面那个本来输出3的就是5了  
    print"A="+A. //输出 A=5  
}
```

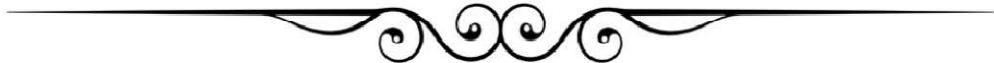
6.5

6.5 自定义函数

6.5.1

6.5.1 概览

===== 点击以返回目录 =====



kOS 支持玩家创建并使用自定义函数。玩家需要先声明函数再调用函数。

玩家可以为函数设置输入参数和返回值。

输入参数可以是任何类型的变量，甚至可以用函数指针来让函数也成为输入参数。

输入参数可以设默认值，如果调用函数时输入参数没写全，少写的参数就用默认值。

调用函数时，

数值量、字符串量、逻辑量的参数是传值调用，对参数进行修改并不会影响到外部；

结构体类型的参数是传址调用，修改参数下的成员，外部的结构体变量也会变化。

6.5.2

6.5.2 声明函数

===== 点击以返回目录 =====



(1) 函数声明语句

自定义函数声明格式

```
[declare] [local] function identifier {statements} [.]
// 声明函数时，[] 符号里的内容都是可选的
```

函数声明的例子：

```
Declare Local function plus { //Declare和Local都是可以省略的
    Declare parameter a,b,c. //参数是形参，参数可以连着写
    //如果函数需要返回值，可以写 Return + 表达式。
    //如果函数需要提前退出，也可以只写 Return.
    return a+b+c. //a+b+3.1415}.
Set a to 1.
Set b to 2.
print "a+b+Pi="+plus(a,b,3.1415).
//自定义函数的使用。系统预设函数用起来方式相同
```

函数的参数可以是任意类型，可以是数值、字符串、逻辑值、结构体、甚至是函数本身（通过函数指针实现）。

函数声明结构在程序中的位置随意，推荐放到文件头部。

*注1：函数参数是函数内的一个局部变量。

*注2：kOS 中函数的定义需要放在死循环语句的前面。

任何情况下，kOS都不会运行死循环语句后面的东西。

(2) 为函数参数设置默认值

函数的输入参数可以设默认值，这样的话，

如果调用函数时输入参数没写全，少写的参数就会用默认值。

```
Function plus { //Declare和Local都是可以省略的
    Parameter a.
    Parameter b,c is 3.1415.
    //参数是形参，参数可以连着写，也可以赋默认值或不赋默认值，
    //赋初值的参数要靠后写
    return a+b+c. //a+b+3.1415}.

Set a to 1.
Set b to 2.
print "a+b+Pi="+plus(a,b).
print "a+b+Pi="+plus(a,b,3.1415).
//自定义函数的使用。系统预设函数用起来方式相同
//调用的时候已经赋默认值的参数可以不写,
//不写的话就按照默认值，写了的话就按照赋的值
//所以以上两句plus函数结果相同
```

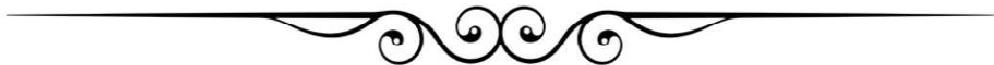
一个函数有多个输入参数的，没有默认值的参数要先写，有默认值的参数要后写。

*注2：函数参数的默认值可以是表达式，该表达式与函数调用时是否写足参数是逻辑短路的关系。亦即，如果调用函数时写了相应参数，该表达式就不执行。

6.5.3

6.5.3 库函数文件

===== 点击以返回目录 =====



玩家可以为 kOS 编写库函数文件。

kOS 专为此功能准备了指令 Run Once 和 函数 RunOncePath。

指令 Run Once 和 函数 RunOncePath 的特点是：同一个程序文件最多只运行一次，重复写无效。

例子：

test1 文件内容:

```
Golbal n To 0.      //用来统计函数运行过多少遍
Function hello {
    Print "Hello".
    Set n To n+1.  //函数运行次数记数
}
```

test2 文件内容:

```
Run Once test1. //运行库函数
hello().
Run Once test1. //test1程序已经运行过一遍了，这句不会运行了
Run Once test1. //test1程序已经运行过一遍了，这句不会运行了
hello().
hello().
Print n.    //显示函数运行次数
```

6.5.4

6.5.4 函数的调用

===== 点击以返回目录 =====



(1) 调用函数的一般方式

调用函数的一般方式遵循的格式是

函数名 (参数1, 参数2,)

参数的数量不能小于函数里未定义默认值的参数数量，
但是也不能大于函数里所有参数的总数量。

(2) 调用外部程序内的函数

ProgramA.ks 内容:

```
Function xyz {
    Declare Parameter x.
    Print "kOS"+x.
    return x*2.
}
```

ProgramB.ks 内容:

```
RunOncePath ("ProgramA").
Print xyz(123).
```

运行**ProgramB.ks**结果:

```
KOS123
246
```

(3) 调用外部程序（不带参数形式）

调用外部程序文件的遵循的格式是

方式一：（推荐）

```
RunPath ("外部程序名").
```

方式二：（不推荐）

```
Run 外部程序名.
```

例子：

ProgramA.ks 文件内容：

```
Print "kOS".
```

ProgramB.ks 文件内容：

```
RunPath ("ProgramA").
Print "KSP".
```

运行**ProgramB.ks**结果：

```
KOS
KSP
```

(4) 调用外部程序（带参数形式）

例子：

ProgramA.ks 文件内容：

```
Parameter a.
Print "kOS"+a.
```

ProgramB.ks 文件内容：

```
RunPath ("ProgramA",234).
Print "KSP".
```

运行**ProgramB.ks**结果：

```
KOS234
KSP
```

从以上来讲，函数（Function）和程序（Program）两个在用法上其实差不多。

(5) 调用其他卷内的程序文件的函数

调用其他卷Volume内函数其实没什么区别，就是先要切换到那个卷。

实例：

Volume 1 中 ProgramA.ks 文件内容:

```
Declare Parameter x.  
Print "kOS"+x.
```

Volume 0 中 ProgramB.ks 文件内容:

```
RunPath("ProgramA",123).  
Switch to 1.  
Print "KSP"+123.
```

在 Volume 0 中运行ProgramB.ks结果:

```
KOS123  
KSP123
```

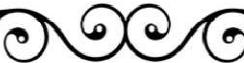
(6) 不要在指令窗口调用自定义函数

kOS 的自定义函数只能在程序文件内调用，不能直接在指令窗口内调用。

6.5.5

6.5.5 函数中的局部变量

===== 点击以返回目录 =====



函数定义的内部也算作一个代码块的内部。



6.5.6

6.5.6 函数的返回值

===== 点击以返回目录 =====



自定义函数使用 Return 指令来确定返回值。

Return 指令有两种用法。

用法一: **return** 加表达式, 函数运行到此时会停止运行并跳出,
并返回按表达式计算的值

return 表达式.

用法二: **return** 不加表达式, 函数运行到此时会停止运行并跳出
return.

```
Return a+b.
```

```
Return.
```

另外, Return可以分 if 的条件返回, 但是不同 return 语句返回的要保持是类型相同。

```

Function plus { //Declare和Local都是可以省略的
    Declare parameter a,b.
    Declare parameter c.
    If c>0 {
        Return a+b+c.
    } else {
        Return a+b.} //要保持不同条件分支里返回的类型相同
}

```

6.5.7

6.5.7 函数的传值调用和传址调用

===== 点击以返回目录 =====



(1) 简单量是传值调用

数值量、字符串、逻辑量，这些是简单的量，在函数调用中是传值调用。

函数内对这些变量赋新值，不影响函数外的变量值。

例子：

```

Function aaa {
    Parameter a.
    Set a to a+1.×
    Print a. // 显示 2
}

Set b to 1.
aaa(b).
Print b. // 显示 1

```

(2) 结构体是传址调用

除此以外的量都是结构体变量，在函数调用中是传地调用。

此时，函数参数传递的是函数外那个结构体变量的地址，

函数内对结构体变量的成员赋新值，等同于直接对函数外那个变量操作。

例子：

```

Function aaa {
    Parameter a.
    Set a:Name to "abcde".
    Print a. // 显示 abcde
}

Set Ship:Name to "qwerty".
aaa().
Print Ship:Name. // 显示 abcde

```

6.5.8

6.5.8 函数的嵌套和递归

===== 点击以返回目录 =====



(1) 函数的嵌套

函数可以嵌套定义的。但是函数内嵌套的函数定义，只有外层函数能调用，外层函数以外的指令无法调用内层函数。

例如下面 函数v2 是嵌套在 函数plus 内部。

但是在 函数plus 外无法调用 函数v2， 函数v2 只能由 函数plus 调用。

```
function plus {
    parameter a , b.
    function v2 {
        parameter v.
        return v*v.
    }×
    set c to a*a+v2(b).
    return c.
}

set z to plus(3,4).
print z. //能运行出结果25
set z to v2(9).
print z. //报错，找不到v2函数
```

(2) 函数的递归

另外，函数也可以递归的，例如下面是一个算阶乘的函数：

```
Function Fa {
    Parameter n.
    If n<1 {
        Set fn to 1.
    } Else {
        Set fn to n*Fa(n-1).
    }×
    Return fn.
}

Print "5! = " + Fa(5). //显示 5! = 120
```

6.5.9

6.5.9 匿名函数

===== 点击以返回目录 =====



kOS 中可以创建匿名函数，匿名函数是没有函数名的函数，只可以通过函数指针操作。

详见下文 匿名函数 和 函数指针。

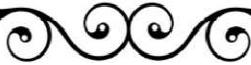
6.6

6.6 匿名函数

6.6.1

6.6.1 语法

===== 点击以返回目录 =====



kOS 提供匿名函数功能，玩家在声明匿名函数时无需规定函数名，而是用其他办法代替。

(1) 声明匿名函数

```
set some_variable to {
    // ---.
    // |
    // |--- 函数体
    // |
    // ---'
}. // 注意加上句点

//然后下面是两种调用方法，同样效果
some_variable().
some_variable:call().

//也可以像正常函数一样加上参数，详见下面的实例
```

(2) 常规的调用方法

例子：

```
set plus to {
    parameter a,b.
    return a + b.
}.

print plus(6, 4).
```

6.6.2

6.6.2 匿名函数的用途

===== 点击以返回目录 =====



匿名函数写起来字数少，无需大段的篇章，用完之后还可以清除函数定义。

适用于临时的小型函数，例如：

```
Set ShowResult to { Parameter n. Print "The Result is "+n.}.
ShowResult(5).
Unset ShowResult.
```

匿名函数还有一个典型的用途是用作函数名的字典。例子如下：

```
Function make_vessel_utilities {
    parameter ves.
    // 创建匿名函数的字典，对于不同问题给出回答。
    return LEXICON(
        "isSmall", {return ves:mass < 50.},
        "isBig", {return ves:mass > 150.},
        "circularEnough", {return ves:obt:eccentricity < 0.1.}
    ).
}

local that_ship_utils is make_vessel_utilities(Vessel("that ship")).

if that_ship_utils["isSmall"]() {
    print "that ship is small".
}

if that_ship_utils["circularEnough"]() {
    print "that ship is circularized".
}
```

*注1：匿名函数本质上是一个函数指针，有关函数指针的详情，请参考 [函数指针](#)

6.7

6.7 函数指针

6.7.1

6.7.1 函数指针语法：@符号

===== 点击以返回目录 =====



kOS 里允许使用函数指针。这个功能是依靠 KOSDelegate 类型结构体实现的。

KOSDelegate 还可以作为函数的参数来用，用法如下。

用法：

函数名 @

例子：

```

Function myfunc {
    parameter a,b.
    return a + b.
}

print myfunc(1, 2).
//函数名之后加个@符号就代表取这个函数的指针了
set aaa to myfunc@.
//调用函数的方式:
print aaa:call(1, 2).
print aaa(1,2). //最后两句一个意思

```

*注1: KOSDelegate 函数指针不适用于结构体内函数。

6.7.2

6.7.2 函数指针的用途

===== 点击以返回目录 =====



(1) 让函数成为其它函数的参数

函数指针能让函数作为其他函数的参数来使用。如下实例:

```

function plus {
    parameter a,b.
    return a+b.}
function times {
    parameter a,b,c,p.
    return p(a,b)*c.}

print times(2,3,5,plus@). //就相当于 (2+3)*5
//而times函数中2与3的加法则是在plus函数中定义

```

(2) 填写部分参数后打包成新函数

函数指针还有第三个用法，就是利用 KOSDelegate 的 bind 成员，从参数列表的左边往右赋值，然后打包成一个新的函数。

```

// 矢量V()有三个方向x,y,z的分量，正常使用时要 V(10,5,1)
local vecx is V@:bind(10). // 先设x=10，将剩下的还没设的参数y,z打包成一个新的函数
local vecxy is vecx:bind(5). // 再设y=5，将剩下的还没设的参数z打包成一个新的函数
local vecxyz is vecxy:bind(1). // 再设z=1，将剩下打包成一个新的函数
local vec is vecxyz:call(). // 等效于 V(10, 5, 1)，其实已经没参数好设了。

```

6.7.3

6.7.3 函数指针与匿名函数

===== 点击以返回目录 =====



匿名函数本质上是一个函数指针。

他们的用法相同，只是声明时形式有一些差异。

7.

7. 数学和基本几何

7.1

7.1 基本常数与数学函数

7.1.1

7.1.1 数学/物理常数

===== 点击以返回目录 =====



常数	写法	类型	说明
G	Constant : G	Scalar	万有引力常数 $6.67384 \times 10^{-11} \text{ N} \cdot \text{m}^2 / \text{kg}^2$
g0	Constant : g0	Scalar	地表重力加速度 9.80655 m/s^2
e	Constant : E	Scalar	自然对数的底 2.718281828459
π	Constant : Pi	Scalar	圆周率 3.141592654
c	Constant : C	Scalar	真空中光速 $299,792,458 \text{ m/s}$
AtmToPa	Constant : AtmToPa	Scalar	气压从 Atm 单位换算到 KPa 单位的系数 101.325
PaToAtm	Constant : PaToAtm	Scalar	气压从 KPa 单位换算到 Atm 单位的系数 0.00986923266716013
DegToRad	Constant : DegToRad	Scalar	从角度换算到弧度的系数 0.0174532925199433
RadToDeg	Constant : RadToDeg	Scalar	从弧度换算到角度的系数 57.2957795130823
Avogadro	Constant : Avogadro	Scalar	阿伏伽德罗常数 $6.02214129 \times 10^{23}$
Boltzmann	Constant : Boltzmann	Scalar	玻尔兹曼常数 $1.3806488 \times 10^{-23} \text{ J/K}$
IdealGas	Constant : IdealGas	Scalar	理想气体常数 8.31447 J/mol/K
.....

代码写法:

```
constant:G
constant:E
constant:Pi
```

7.1.2

7.1.2 数学函数

===== 点击以返回目录 =====



*注1：数学函数里描述角度大小用的数值是弧度制。

(1) 基本数学函数

函数	说明	例子
abs (a)	绝对值	PRINT ABS (-1). // prints 1
ceiling (a)	向上取整	PRINT CEILING (1.887). // prints 2
floor (a)	向下取整	PRINT FLOOR (1.887). // prints 1
ln (a)	自然对数lg	PRINT LN (2). // prints 0.6931471805599453
log10 (a)	以10为底的对数	PRINT LOG10 (2). // prints 0.30102999566398114
mod (a, b)	余数	PRINT MOD (21, 6). // prints 3 PRINT MOD (-21, 6). // prints -3
min (a, b)	最小值 (参数只能是两个)	PRINT MIN (0, 100). // prints 0
max (a, b)	最大值 (参数只能是两个)	PRINT MAX (0, 100). // prints 100
random ()	随机数 (范围0~1)	PRINT RANDOM (). // 0~1 随机数
round (a)	四舍五入到整数	PRINT ROUND (1.887). // prints 2
round (a, b)	将数字a 四舍五入到b位小数	PRINT ROUND (1.887, 2). // prints 1.89
sqrt (a)	平方根	PRINT SQRT (7.89). // prints 2.80891438103763
char (a)	按照unicode编码 将数值转换为字符	PRINT CHAR (65). // prints A
unchar (a)	按照unicode编码 将字符转换为数值	PRINT UNCHAR ("A"). // prints 65

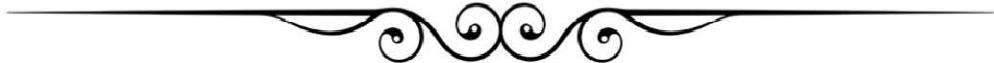
(2) 三角函数

函数	说明	例子
<code>sin (a)</code>	正弦	<code>PRINT SIN (6).</code> <code>// prints 0.10452846326</code>
<code>cos (a)</code>	余弦	<code>PRINT COS (6).</code> <code>// prints 0.99452189536</code>
<code>tan (a)</code>	正切	<code>PRINT TAN (6).</code> <code>// prints 0.10510423526</code>
<code>arcsin (x)</code>	反正弦	<code>PRINT ARCSIN (0.67).</code> <code>// prints 42.0670648</code>
<code>arccos (x)</code>	反余弦	<code>PRINT ARCCOS (0.67).</code> <code>// prints 47.9329352</code>
<code>arctan (x)</code>	反正切 (对边 / 临边 = x)	<code>PRINT ARCTAN (0.67).</code> <code>// prints 33.8220852</code>
<code>arctan2 (x, y)</code>	反正切 (对边 / 临边 = x / y)	<code>PRINT ARCTAN2 (0.67, 0.89).</code> <code>// prints 36.9727625</code>

7.2

7.2 数值 Scalar [结构体]

----- 点击以返回目录 -----



(1) kOS 中的数值

kOS 中的数值都是实数标量（Scalar），
存储记数时全部存为32位整数或64位浮点数，二者间自动切换。
64位浮点数时具有15~16位有效数字。

数值Scalar 是一种结构体。

kOS 中所有变量本质上都是结构体，都是由 最底层的 Structure 结构体 派生而来。
数值结构体Scalar 相比于 Structure 结构体，结构体成员完全相同，专为记数而设计。

虽说是这样，但是在 kOS 中，玩家只需要将 数值Scalar 当做一个普通的数来使用即可。

(2) 结构体成员

该结构体派生自 结构体 Structure 结构体。

玩家可以在 结构体 Structure [结构体] 查看 结构体 Structure 结构体的详细信息。

Scalar 数值结构体 成员列表：

结构体成员	成员类型	读写性	说明
Structure 结构的成员			派生自此类型，拥有其全部成员
.....			

(3) 数值的书写规则

有关数值的书写规则：

书写时允许在首位以外的任意位置加下划线来表示位数分割，
书写时允许使用工程记数法。如下写法全部都是合法的：

```
12345678
12_345_678
12345.6789
12_345.6789
-12345678
12.123e12
1.234e-12
```

(4) 数值的操作符

数值Scalar 和 数值Scalar 之间支持指数运算符和四则运算符：

a^b	a的b次方
$-a$	a的负数
$a * b$	a和b之间的乘除法
a / b	a和b之间的加减法

(5) 数值的局限

((1)) 数值Scalar是实数，不支持复数运算。

((2)) 数值Scalar以二进制数值形式存储，这是一个有理数。

kOS无法存储和计算无理数，只能给无理数取有理数的近似值之后再计算和存储。

((3)) 数值Scalar具有15~16位有效数字。有效数字后的误差会在逐次计算中放大，玩家实践中需要留意数值误差的放大情况，避免误差淹没有效数字的情况产生。

7.3

7.3 列表 List [结构体]

===== 点击以返回目录 =====



在kOS中，列表List关键词有两个角色：一种结构体类型、一种指令。

这里说的就是列表List作为结构体类型的特性和用法。

kOS中的列表结构体List就相当于其他编程语言里的数组。

(1) 创建结构体

在kOS中有许多结构体和函数都可以返回列表List。

如果要完全从零开始创建列表，就需要用List函数。

```
Set a to List().           //创建空列表
Set a to List(2,3,4,5,6).   //创建数值列表{2,3,4,5,6}
Set a to List("2","3a","4b"). //创建字符串列表{"2","3a","4b"}
Set a to List(List("a","b","c"),
              List(1,2,3)). //创建二维列表
```

另外，列表结构体还可以从其他列表或结构体抽取：

```
//使用List指令从预设列表创建列表的副本
List Parts in a. //为预设列表Parts创建副本a

//使用可以创建列表的结构体函数
Set a to Ship:PartsTagged(TagB). //当前载具上所有标签名为TagB的部件的列表

等等.....
```

有关kOS有哪些预设列表，请参考系统预设变量。

(2) 结构体成员

该结构体派生自 枚举Enumerable 结构体。

玩家可以在 枚举 Enumerable [结构体] 查看 枚举Enumerable 结构体的详细信息。

该结构体是可序列化 (Serializable) 的结构体。

List列表结构体 成员列表:

结构体成员	成员类型	读写性	说明
Enumerable 结构的成员			派生自此类型，拥有其全部成员
Add (item)	无返回值	函数	在末尾追加列表项目item
Insert (index, item)	无返回值	函数	在第index位插入列表项目item
Remove (index)	无返回值	函数	移除第index位的列表项目
Clear ()	无返回值	函数	清空列表
Copy	List	只读	当前列表的副本，用于复制列表
SubList (index, length)	List	只读	抽取列表的部分成为一个新的列表，取第index位开始的length长度的部分列表
Join (AAA)	String	只读	返回以AAA作为分隔符的列表内容
.....			

*注1: 可序列化是指,

该类型变量能通过 WriteJSON函数 和 ReadJSON函数 被读写至 json 文件,
指该类型变量能用于僚机之间的通信数据互传。

*注2: 列表List 的位置后缀是从0开始数的,

例如对于 Set a to List(4,5,6)., a[1] 的值是 5。

(3) 访问列表成员

列表成员的位置从0开始算。如果一个列表有N个成员，那么其成员位置就是 0 ~ N-1。

有四种方法可以访问列表成员：

方法1. **list[表达式]**

方法2. **list #整数**

```
Set a to List(2,3,4,5).
Print a[1+1]. //显示 4
Set b to 2.
Print a#2.    //显示 4
Print a#b.    //显示 4
```

*注3: 方法2的#号后跟的只能是数值量，而方法1的方括号内可以是表达式

方法3. **for a in list { ... }.**

```
Set a to List(2,3,4,5).
For b in a {
    Set b to b+1.
} //a={3,4,5,6}
```

*注4: For 循环是专为列表List 设计的, 只能用于对列表成员进行统一操作。

方法4. 使用列表迭代器 Iterator

有关迭代器Iterator 的使用, 请参考 迭代器 Iterator [结构体]。

(4) 多维数组

列表成员可以是任何变量, 也可以是另一个列表。

这样就构成了多维数组了。

```
Set a1 to List(1,2,3,4).
Set a2 to List(5,6,7,8).
Set a3 to List(9,10,11,12).
Set a to List(a1,a2,a3).
```

访问多维数组成员的方法和访问普通数组雷同, 区别只是需要一层一层的剥开。

同理, 方括号里可以放表达式, 而#后面就只能放数值量了。

```
Set b1 to a[1].      //b1={5,6,7,8}
Set b12 to a[1][1+1]. //b12=7
Set b1 to a#1.        //b1={5,6,7,8}
Set c to 2.
Set b12 to a#1#2. //b12=7
Set b12 to a#1#c. //b12=7
```

(5) 比较两个列表

假设 ListA 个 ListB 都是列表变量。

kOS 会对如下判断式给出 True 或 False 的返回值。

```
ListA = ListB
```

但是请注意, 此式比较的不是“是否两个列表的内容相同”,

而是“是否两个标识符指向的是同一段内容”。

二者之间的在下面这段代码里可见一斑。

```
Set a to List(1,2,3,4,5).
Set b to List(1,2,3,4,5).
Set c to a.
If a=b {Print "The Same List"}. //不会Print
If a=c {Print "The Same List"}. //会Print
```

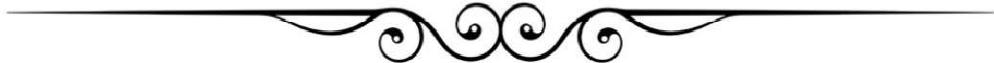
所以如果要比较“是否两个列表的内容相同”,

玩家还需要逐个比较列表成员, 不能偷懒。

7.4

7.4 矢量 Vector [结构体]

----- 点击以返回目录 -----



(1) 左手坐标系

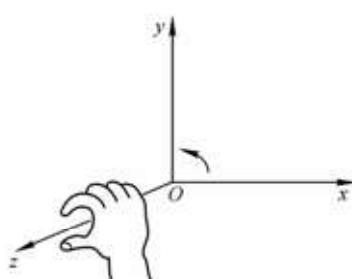
kOS 使用左手系直角坐标来描述游戏中各物体的方位。

和真实世界中基于右手系坐标建立起来的计算体系是镜像相反的。

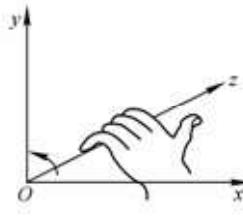
这个特点是由 Unity3D 游戏引擎决定的。

右手坐标系: $\hat{x} \times \hat{y} = \hat{z}$, 叉乘依照右手螺旋规则

左手坐标系: $\hat{x} \times \hat{y} = -\hat{z}$, 叉乘依照左手螺旋规则



(a) 右手系统



(b) 左手系统

(2) 创建结构体

使用 V 函数可以创建矢量结构体。kOS 中的矢量永远是三维矢量。

```
Set a to V(x,y,z).           //a是矢量结构体
Set b to V(2,3,2+2).)
```

另外，矢量结构体还可以由矢量计算得到，还可以从其他结构体抽取：

```
Set a1 to V(0,0,1).
Set a2 to V(0,2,0).
Set b1 to a1+a2. //b1=(0,2,1)

Set b2 to Target:Position. //b2为目标物在 Ship-Raw 坐标系中的坐标矢量

等等.....
```

有关 Ship-Raw 坐标系，请参考 坐标系。

(3) 结构体成员

该结构体是可序列化 (Serializable) 的结构体。

Vector 矢量结构体 成员列表：

结构体成员	成员类型	读写性	说明
X	Scalar	读写	在x轴的投影
Y	Scalar	读写	在y轴的投影
Z	Scalar	读写	在z轴的投影
Mag	Scalar	读写	矢量的模
Normalized	Vector	只读	单位矢量
Sqrmagnitude	Scalar	只读	模的平方
Direction	Direction	读写	对应的朝向
Vec	Vector	只读	矢量的副本
.....			

*注1：可序列化是指，该类型变量能通过 WriteJSON 函数 和 ReadJSON 函数 被读写至 json 文件，指该类型变量能用于僚机之间的通信数据互传。

(4) 矢量计算

函数 / 操作符	返回值类型	说明
*	Scalar or Vector	标量乘矢量 或 矢量点乘矢量
+	Vector	矢量相加
-	Vector	矢量相减
-	Vector	负质量。模不变，方向相反
VDot (v1, v2), *, VectorDotProduct (v1, v2)	Scalar	两矢量的数量积（点积）
VCrs (v1, v2), VectorCrossProduct (v1, v2)	Vector	两矢量的向量积（叉积）
VAng (v1, v2), Vectorangle (v1, v2)	Scalar	两矢量之间的夹角，单位：° 范围：0~180
VXcl (v1, v2), VectorExclude (v1, v2)	Vector	v2关于v1的正交分量，换句话说 v2在v1的法平面的投影矢量
.....		

矢量计算的例子：

向量与数的乘法，向量的点乘：

```
SET a TO 2.
SET vec1 TO V(1,2,3).
SET vec2 TO V(2,3,4).
PRINT a * vec1. // 显示 V(2,4,6)
PRINT vec1 * vec2. // 向量的点乘，显示 20
```

负向量：

```
PRINT -vec1. // 负向量
PRINT (-1)*vec1. // 负向量，同上
```

向量的加减法:

```
SET a TO 2.
SET vec1 TO V(1,2,3).
SET vec2 TO V(2,3,4).
PRINT vec1 + vec2. // 显示 V(3,5,7)
PRINT vec2 - vec1. // 显示 V(1,1,1)
```

向量的点乘:

```
SET vec1 TO V(1,2,3).
SET vec2 TO V(2,3,4).
// 以下三句命令全都是向量点乘, 显示 20
PRINT vec1 * vec2.
PRINT VDOT(vec1, vec2).
PRINT VECTORDOTPRODUCT(vec1, vec2).
```

向量的叉乘:

```
SET vec1 TO V(1,2,3).
SET vec2 TO V(2,3,4).
// 以下两句命令全都是叉乘, 显示 V(-1,2,-1)
PRINT VCRS(vec1, vec2).
PRINT VECTORCROSSPRODUCT(vec1, vec2).
```

```
SET varname TO V(100,5,0).
varname:X. // 向量的x分量 100.
V(100,5,0):Y. // 向量的y分量 5.
V(100,5,0):Z. // 向量的z分量 0.
```

varname:MAG. // 向量的模

```
SET varname:X TO 111. // 修改向量值, 修改x分量
SET varname:MAG to 10. // 修改向量值, 维持方向不变伸缩向量至模=10
```

7.5

7.5 朝向 Direction [结构体]

===== 点击以返回目录 =====



描述一个物体运动需要3个平动量，3个转动量，一共6个量。

朝向Direction就是其中用来描述转动的量。

(1) 左手坐标系

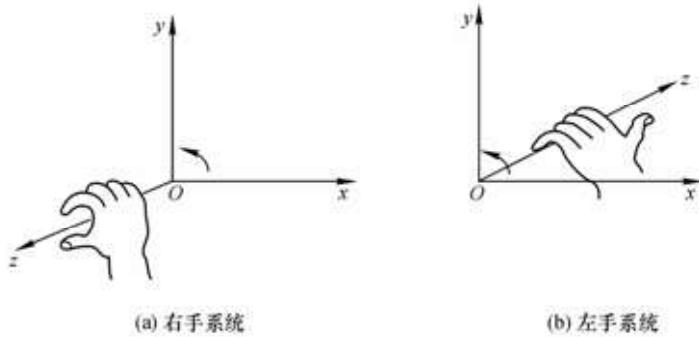
kOS 使用左手系直角坐标来描述游戏中各物体的方位。

和真实世界中基于右手系坐标建立起来的计算体系是镜像相反的。

这个特点是由 Unity3D 游戏引擎决定的。

右手坐标系: $\hat{x} \times \hat{y} = \hat{z}$, 叉乘依照右手螺旋规则

左手坐标系: $\hat{x} \times \hat{y} = -\hat{z}$, 叉乘依照左手螺旋规则



(2) 创建结构体

创建方式	说明
R (pitch, yaw, roll)	欧拉旋转。以载具姿态为基准，处理顺序是先转roll后转pitch最后转yaw 【常用】
Q (x, y, z, rot)	四元数旋转。比较复杂，如果不懂就别用
Heading (dir, pitch)	游戏界面陀螺仪上的指向，载具前方指向正北方向经线起顺时针dir角度，pitch俯仰角度，载具头顶指向正北方向 【常用】
LookDirUp (lookAt, lookUp)	载具前方同矢量lookAT方向，载具头顶是矢量lookUP方向
AngleAxis (degress, axisV)	绕轴旋转，沿矢量axisV方向旋转degress角度 degress 单位：°
RotateFromTo (fromV, toV)	能将矢量fromVec旋转到矢量toVec方向的朝向
Facing	其他结构体的成员。例如左边是载具的前方、上方、速度方向的朝向
Up	
Prograde	
等等	等等

*注1：四元数旋转是一种基于高纬复数的表达方法，不会引起万向锁问题，但是不容易理解。

(3) 结构体成员

Direction朝向结构体 成员列表:

结构体成员	成员类型	读写性	说明
Pitch	Scalar (°)	只读	绕着Pitch轴旋转的角度
Yaw	Scalar (°)	只读	绕着Yaw轴旋转的角度
Roll	Scalar (°)	只读	绕着Roll轴旋转的角度
ForeVector	Vector	只读	朝向的前方对应的单位矢量 是 Roll 的旋转轴方向
Vector	Vector	只读	同上
TopVector	Vector	只读	朝向的头顶上方向对应的单位矢量 是 Yaw 的旋转轴方向
UpVector	Vector	只读	同上
StarVector	Vector	只读	朝向的右舷方方向对应的单位矢量 是 Pitch 的旋转轴方向
RightVector	Vector	只读	同上
Inverse	Direction	只读	反向朝向
-	Direction	只读	前面加负号 -，也可以表示反向朝向
.....			

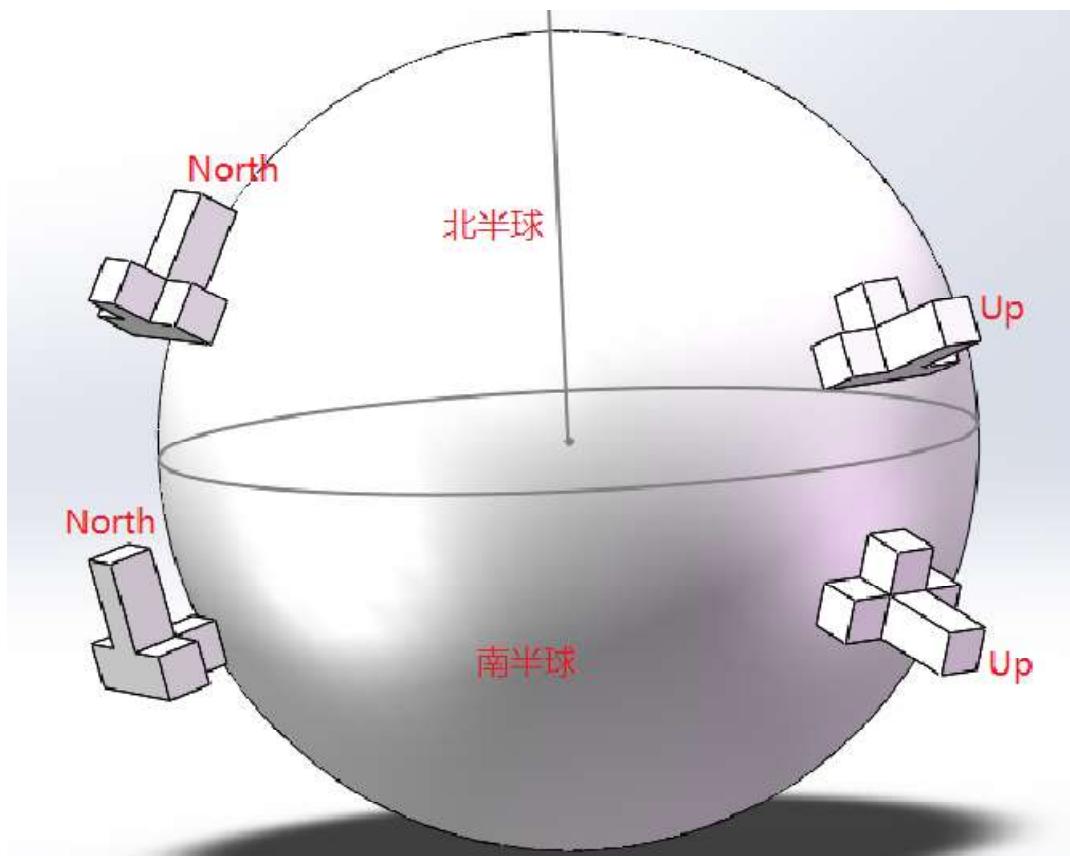
*注2：结构体创建后均以欧拉旋转表示。

(4) 方位的对应朝向

朝向Direction 比方向 多一个滚转维度Roll。

在 kOS 中，像 Up、North 这类根据方向生成的朝向值，其滚转Roll 值由 kOS 自动生成。
其规律为：使得载具头顶最靠近北天极。

在星球不同位置的方向对应的朝向：



(5) 欧拉旋转

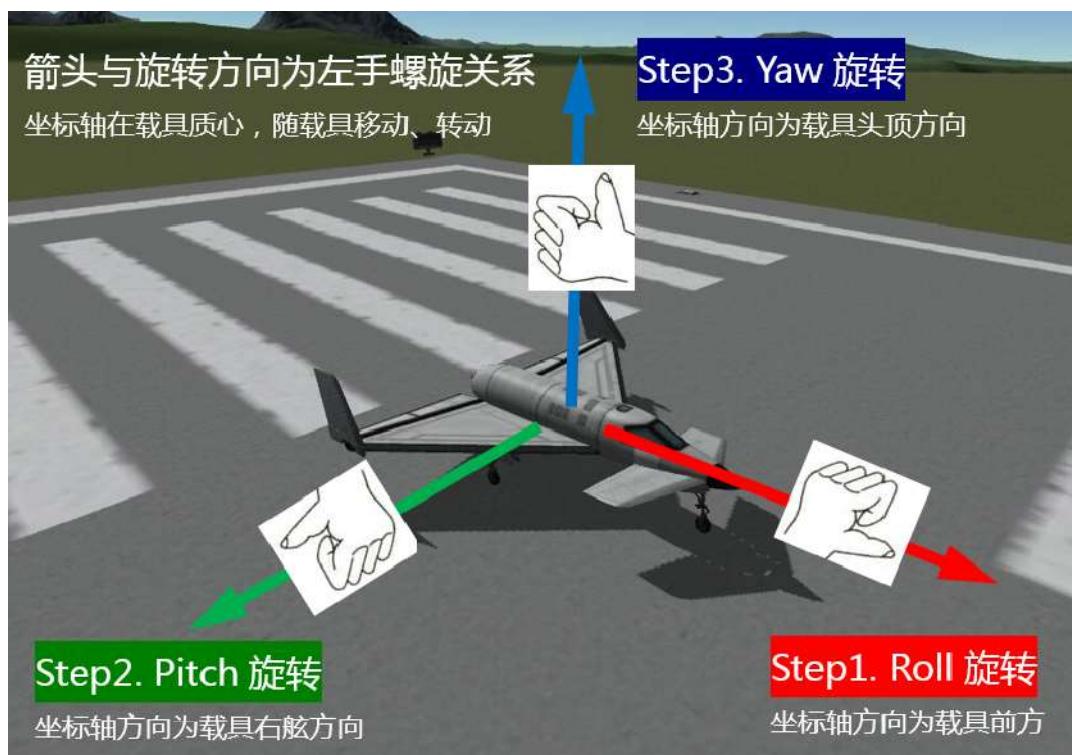
欧拉旋转是一种常用的易理解的表述方法。

他是一个旋转函数 $R(\text{Pitch}, \text{Yaw}, \text{Roll})$ 单独使用得不到有意义的结果。

要搭配 UP、North 或 Heading 函数 这些有明确指向的一起使用才有意义。

pitch, yaw, Roll 分别表示以左手螺旋，沿着飞船右舷方向，飞船头顶方向，飞船前方方向，转过的角度。

图解欧拉旋转角：

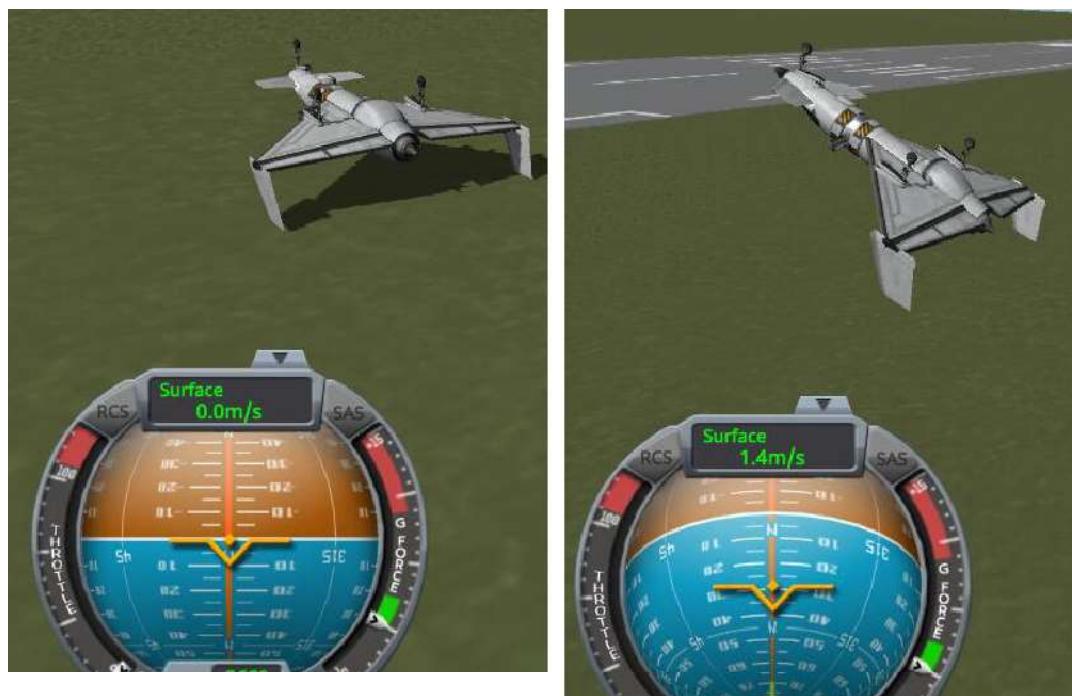


例子：在南半球分别转向至 North 和 $\text{North}^*\text{R}(30,0,0)$

Lock Steering to North.

Lock Steering to $\text{North}^*\text{R}(30,0,0)$.

通常使用 $\text{A}^*\text{R}(\text{Pitch}, \text{Yaw}, \text{Roll})$ 形式来表达使用欧拉旋转进行的旋转。



(6) 操作符和函数

朝向值可以使用数学符号计算，也可以使用函数计算。

Dir1 * Dir2 朝向 * 朝向

将 Dir2 按照 Dir1 进行旋转后得到的朝向。

例子：

```
//Heading的第一个参数是从罗盘正北方向顺时针转过而角度
//第二个参数是竖直俯仰，然后滚转至头顶向Up方向
//Heading的参数值没有范围限制，Heading(361,361)等同于Heading(1,1)
Lock Steering to Heading(90, 0).
Set a to AngleAxis(45,Up:Vector).
Wait 20.
Lock Steering to a*Heading(90, 0).
```



Dir * Vec 朝向 * 矢量

将 Vec 按照 Dir 进行旋转后得到的朝向。

Dir1 + Dir2 朝向 + 朝向 [不推荐]

这种写法可能会引起万向锁，不推荐用。

(7) 矢量和朝向的区别

矢量Vector 和 朝向Direction 都是一个三维数组，包含的信息量相同，但信息互有异同。

矢量值有方位信息和长度信息，但是没有滚转角度信息。

朝向值有方位信息和滚转角度信息，但是没有长度信息。

所以他们可以互相转换，但是转换的时候会丢失自己所独有的信息：

Dir → Vec 朝向转换为矢量

在 kOS 中，朝向值转换为的矢量值时，会丢失 滚转Roll 值。

新生成的矢量值为单位矢量，模为1。

Vec → Dir 矢量转换为朝向

在 kOS 中，矢量值转换为的朝向值时，其滚转Roll 值由 kOS 自动生成。

生成规律为：使得头顶方向最靠近北天极。

7.6

7.6 经纬坐标 GeoCoordinates [结构体]

===== 点击以返回目录 =====



kOS 的地理坐标系统和真实世界相同，使用经度，纬度，海拔高度描述地面的点。

(1) 创建结构体

使用 Latlng 函数可以创建经纬坐标结构体。

```
Set a to Latlng(37,-18).      //东经37°，南纬18°
Set b to Latlng(0,90).        //北极点
```

另外，经纬坐标结构体还可以从其他结构体抽取：

```
Set a1 to Ship:Geoposition. //当前载具的经纬坐标
```

等等.....

(2) 结构体成员

该结构体是可序列化（Serializable）的结构体。

GeoCoordinates 经纬坐标结构体 成员列表：

结构体成员	成员类型	读写性	说明
Lat	Scalar (°)	只读	纬度, -90 南极~+90 北极
Lng	Scalar (°)	只读	经度, -180 西经~+180 东经
Distance	Scalar (m)	只读	地表点到当前载具的距离
TerrainHeight	Scalar (m)	只读	地面的海平面高度, 负值表示地面在海平面之下
Heading	Scalar (°)	只读	从当前载具到地表点的罗盘角度, 是从正北方向顺时转过的角度
Bearing	Scalar (°)	只读	从当前载具到地表点的相对罗盘角度, 表示当前载具的几点钟方向, 在右侧为角度正值, 在左侧为角度负值
Position	Vector	只读	从当前载具质心指向地表点的矢量, Ship - Raw坐标系
AltitudePosition (h)	Vector	只读	从当前载具质心指向海拔高度h点的矢量, Ship - Raw坐标系
Velocity	OrbitableVelocity	只读	地表点的自转线速度
AltitudeVelocity (h)	OrbitableVelocity	只读	海拔高度h点的自转线速度
.....			

*注1: 可序列化是指,

该类型变量能通过 WriteJSON 函数 和 ReadJSON 函数 被读写至 json 文件,

指该类型变量能用于僚机之间的通信数据互传。

(3) 实际例子

```
Set spot To Latlng(37,-18). //东经37°, 南纬18°
Lock Steering To spot.      //载具方向朝向spot点
Set cft To SHIP:GEOPOSITION. //当前载具的经纬坐标
```

7.7

7.7 坐标系

7.7.1

7.7.1 左手坐标系

===== 点击以返回目录 =====



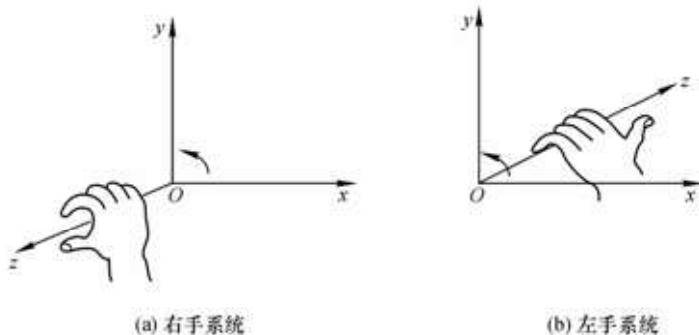
kOS 使用左手系直角坐标来描述游戏中各物体的方位。

和真实世界中基于右手系坐标建立起来的计算体系是镜像相反的。

这个特点是由 Unity3D 游戏引擎决定的。

右手坐标系: $\hat{x} \times \hat{y} = \hat{z}$, 叉乘依照右手螺旋规则

左手坐标系: $\hat{x} \times \hat{y} = -\hat{z}$, 叉乘依照左手螺旋规则



7.7.2

7.7.2 三种坐标系

----- 点击以返回目录 -----

kOS 中用来描述位置矢量时，有三种坐标系描述方式。

(1) 原生坐标 Raw-Raw

Raw-Raw坐标系是最没使用价值的坐标系，其数值没有物理意义。

他是依照绝对静止宇宙决定的惯性坐标系，是一个惯性系。

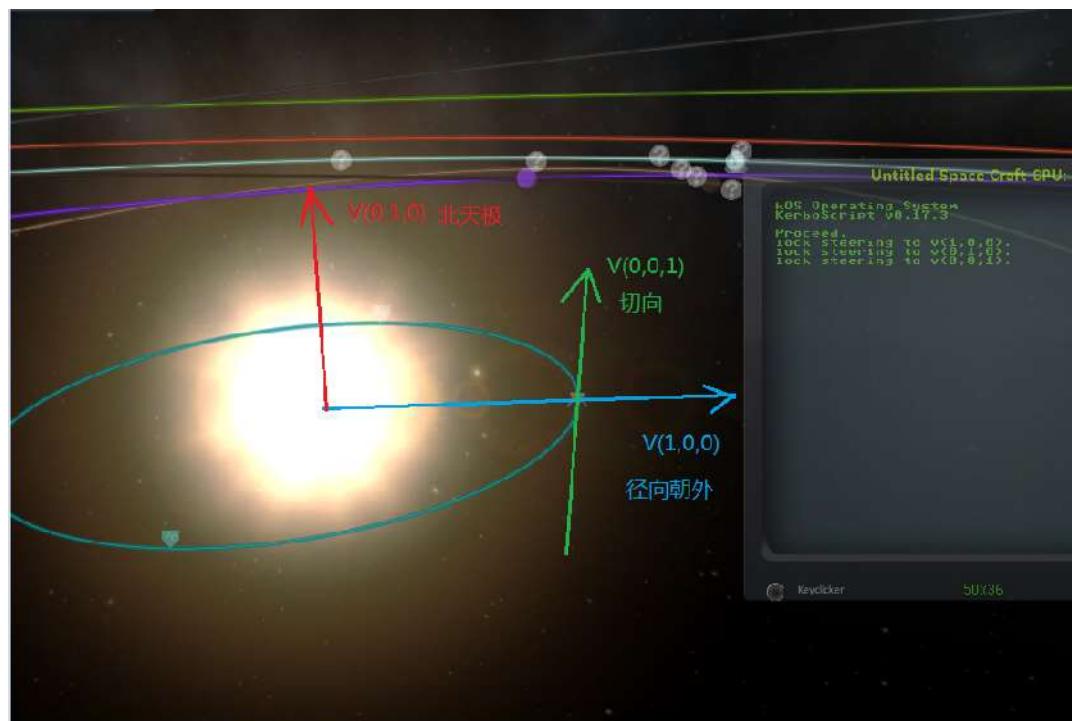
Raw-Raw坐标系仅用于在后台进行计算。

单纯的矢量 $V(x,y,z)$ 和朝向 $R(pitch,yaw,roll)$ 表示的就是基于这个坐标系的量。

由于此坐标系没有物理意义，所以这些函数的返回值也对玩家游戏起不到帮助。

下面是例子：Raw-Raw坐标系中的单位矢量。

Raw-Raw坐标系里 唯一的特征是： Y轴 $V(0,1,0)$ 永远指向北天极。

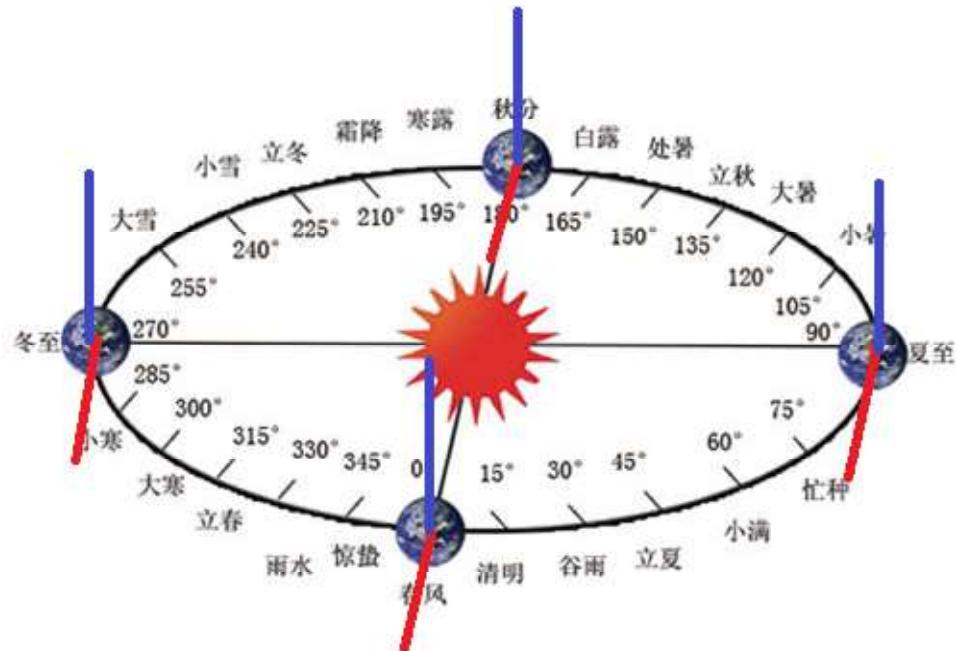


这些坐标轴在游戏时没啥价值

(2) 星球重心坐标 SOI-Raw

SOI-Raw坐标系是以当前星球为中心，依照星球春分点朝向和北天极朝向决定的坐标系。坐标系不随星球自转，但是随星球平移，是一个惯性系。

北天极方向为下图中蓝线，春分点方向为下图中红色。只画两个坐标轴是因为可以根据他们推出第三个坐标轴。



(3) 载具中心坐标 Ship-Raw

Ship-Raw坐标系是最常见的也是最实用的坐标系。

他是以当前载具质心为原点的坐标系，坐标轴的转动与SOI-Raw坐标系相同。

Ship-Raw坐标系不是惯性系，但是当扣除掉载具自身加速度后，可以看做惯性系。

7.7.3

7.7.3 坐标系换算

===== 点击以返回目录 =====



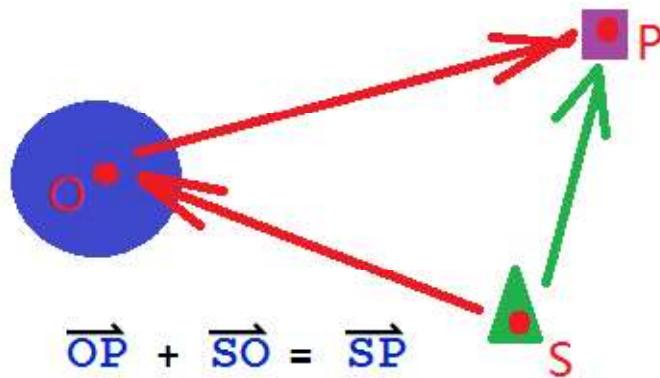
kOS 中的 Ship-Raw 坐标系 和 SOI-Raw 坐标系，他们能相互换算：

```
Ship_RAW - Ship : Body : POSITION = SOI_RAW
SOI_RAW + Ship : Body : POSITION = Ship_RAW
```

为了便于理解，假设坐标点为P，SOI中心点为O，飞船点为S，那么上面两个式子就可以写成

$$\vec{SP} - \vec{SO} = \vec{SP} + \vec{OS} = \vec{OP}$$

$$\vec{OP} + \vec{SO} = \vec{SP}$$



8.

8. kOS 指令

8.1

8.1 运行程序

8.1.1

8.1.1 概述

===== 点击以返回目录 =====



可供 kOS 进行程序运行的文件一共有两种：

源代码文件

机器语言文件

源代码文件是包含了程序代码的文本文件。

kOS 可以对其进行**编译、运行**。

源代码文件的优点：kOS 在编译和运行时能对代码进行**语法检查**，
显示报错信息和相关代码。

源代码文件的缺点：kOS 直接运行源代码文件时，
在开头会有个语法检查和编译过程的耗时，
程序会迟一些再开始运行。

机器语言文件是 kOS 对源代码文件进行编译后得到的文件，后缀名为 .ksm。

kOS 只能对 .ksm 文件进行**运行**。

机器语言文件的优点：.ksm文件 立即就能运行。

机器语言文件的缺点：.ksm文件 不含代码，kOS 无法进行**语法检查**和报错，
程序运行错误时也无法回溯原因。

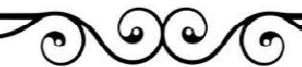
*注1：通常一个程序的机器语言文件会比源代码文件小很多。因为源代码文件里大量的注释、复杂的变量名占用了很多体积。但事情并不是绝对的，有时候机器语言文件会反而比源

代码文件大。

8.1.2

8.1.2 函数方法运行程序

===== 点击以返回目录 =====



kOS 提供以下两个函数来运行程序：

RunPath 函数。

常规运行

用法：

`RunPath (path, para1, para2, ...).`

(*path为字符串，是程序文件的路径，

只写文件名就表示文件在Volume根目录下*)

(*para1,para2,...表示该程序的程序参数，数量由程序定义决定*)

RunOncePath 函数。

非重复运行，如果该程序未运行过才会运行

用法：

`RunOncePath (path, para1, para2, ...).`

(*path为字符串，是程序文件的路径，

只写文件名就表示文件在Volume根目录下*)

(*para1,para2,...表示该程序的程序参数，数量由程序定义决定*)

例子：

0:/archive_lib/myfile.ks 文件内容:

```
Parameter a,b.  
Print a+b.
```

Volume 根目录下 myprogram 文件内容:

```
Print "www".
```

程序内容:

```
set filepath1 to "0:/archive_lib/myfile.ks".  
set filepath2 to "myprogram".
```

```
//RunPath 的作用是运行参数路径的程序，以下两种都可以  
RunPath(filepath1,2,3).  
RunPath("0:/archive_lib/myfile.ks",2,3).  
RunPath(filepath2).  
RunPath("myprogram").
```

//RunOncePath 和 RunPath 的区别是:

```
//RunOncePath 最多只会运行同样的程序一次，如果该程序之前运行过就不运行  
RunOncePath(filepath1,2,3).  
RunOncePath("0:/archive_lib/myfile.ks",2,3).  
RunOncePath(filepath2).  
RunOncePath("myprogram").
```

按快捷键 Ctrl+C 可强制中止正在运行的 kOS 程序。

8.1.3

8.1.3 关键词方法运行程序(不推荐)

===== 点击以返回目录 =====



使用关键词方法运行程序是旧版本 kOS 中的做法。不推荐玩家使用。

用法和函数方法运行程序的类似，

只不过程序参数要像函数参数一样写在括号里。

```
run (once) test. //如果有现成的test.ksm文件，运行它；  
//如果没有，则编译运行test.ks或test文件(不产生test.ksm文件)  
run (once) test.ksm. //运行已经编译好的test.ksm文件，  
//test.ksm可以是compile命令生成的，也可以是之前run命令留下的  
run (once) test (1,2). //以1,2作为参数运行，和函数类似
```

8.1.4

8.1.4 程序运行的细节

===== 点击以返回目录 =====



(1) 库函数的非重复运行

使用 RunOncePath 函数 和 Run Once 指令 可避免程序重复运行，因为他们如果碰到已经运行过的程序，就不会再运行了。

这个特点很适合用于专门存储多种函数声明的库函数，也适合某些变量的初始化程序。

(2) 路径的自动补全

程序文件名要以路径形式的字符串量表达。

例如“0:/archive_lib/myfile.ks”表示在 Volume 0 下的 archive_lib 下的没有 myfile.ks 文件。

当遇到一些路径不全的情况时，kOS 会自动补全路径：

情况一：例如“archive_lib/myfile.ks”这样缺少 Volume 编号的，
kOS 会默认在当前 Volume 下。

情况二：例如“myfile.ks”这样只有文件名的，
kOS 会默认在当前路径下。

情况三：例如“myfile”这样文件名缺少后缀的，
kOS 会：

- Step1. 在当前路径下寻找 myfile.ksm 文件，如果有那就是了；
- Step2. 如果上一步没找到，那么找当前路径下的 myfile 文件；
- Step3. 如果上一步还没找到，呢么找当前路径下的 myfile.ks 文件。

*注1：当前路径由 CD(path) 函数 确定，默认为当前 Volume 的根目录。

有关 CD(path) 函数的内容，请查看文件 I/O

(3) 程序的参数

kOS 中的程序文件可以像函数一样有参数，并且像函数一样使用。

ProgramA.ks 文件内容：

```
Parameter a.
Print "kOS"+a.
```

ProgramB.ks 文件内容：

```
RunPath ("ProgramA",234).
Run ProgramA(234).           //这句和上面一句一个意思
Print "KSP".
```

运行**ProgramB.ks**结果：

```
KOS234
KOS234
KSP
```

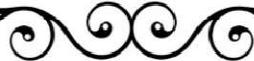
8.2

8.2 飞行控制

8.2.1

8.2.1 概述

===== 点击以返回目录 =====



kOS 中有三种载具驾驶操控的方法：

1. 只作用于载具朝向和引擎节流阀/车轮节流阀的经典控制；
2. 专注于载具每个方向精细操控的底层控制；
3. 专注于采集玩家键盘上驾驶操作的驾驶员输入。

8.2.2

8.2.2 经典控制

===== 点击以返回目录 =====



(1) 经典控制需用 Lock 操作

经典控制一共有四个量：

变量	可读写	说明
Throttle	读写	节流阀，范围 0~1，如果所赋值超出范围则就近取值
Steering	读写	载具姿态朝向。接受朝向 Direction 或 矢量 Vector 类型，还有字符串 "kill"，此字符串表示阻止载具旋转
WheelThrottle	读写	车轮的出力，范围 -1~1. 负值指倒车，如果所赋值超出范围则就近取值
WheelSteering	读写	车轮的转动朝向。可接受 经纬坐标 GetCoordinates 、矢量量 Vessel 、数值 Scalar 。其中 数值 Scalar 表示从正北方向顺时针转过的角度
.....

经典控制的这四个量：Throttle、Steering、WheelThrottle、WheelSteering，控制时必须使用 Lock 指令。

例子：

```

Lock Steering to UP. 飞船方向锁定至正上方
Lock Steering to Heading (90,45). 飞船方向锁定至偏东45°
Lock Steering to V(1,2,3). 飞船方向锁定至和变量V(1,2,3)同方向
Lock Steering to (-1)*Ship:Velocity:Surface. 飞船方向锁定至地标速度的反方向
Lock Steering TO VCRS (Ship:Velocity:Orbit,Body:POSITION). 飞船方向锁定至叉乘方向

```

*注1：kOS 的经典控制的转向操作使用了 PID 控制。

玩家可以查看 调姿管理 **SteeringManager** [结构体]，了解更多信息。

*注2：当 kOS 的运行结束时，控制权会交还给玩家。

这会造成节流阀杆量的一个问题：

如果运行程序前，玩家将节流阀设置在0.3，
那么控制权交还给玩家时，节流阀会恢复到0.3。
下面的语句可以将玩家的节流阀杆量恢复到0，避免这种情况发生。

```
Set Ship:Control:PilotMainThrottle to 0.
```

(2) 不要在控制循环中 Wait

下面这种将 Throttle 挂钩到一个带有 wait 指令的函数，这种事情，别做。
这种写法就相当于在触发结构里加Wait，会导致整个载具操控的延迟。

```
function get_throttle {
    wait 0.001. // this line is a bad idea.
    return 0.5.
}
lock throttle to get_throttle().
```

(3) 解锁控制

使用 Unlock 解锁指令可以取消基于锁定指令 Lock 的载具驾驶控制。

```
UNLock Steering. //解锁控制
UNLock Throttle.
UNLock ALL. //解除所有锁定
```

(4) 调解控制的 PID

使用 Lock Steering 命令进行姿态转向时，
kOS 会使用一个名叫 SteeringManager 的内置结构体来进行转向的PID控制，
其中PID的参数玩家可以更改。

例子：

```
SET STEERINGMANAGER:PITCHPID:KP TO 0.85.
SET STEERINGMANAGER:PITCHPID:KI TO 0.5.
SET STEERINGMANAGER:PITCHPID:KD TO 0.1.
```

kOS 中 SteeringManager 的参数设置是固定的。
如果玩家觉得载具的转向效果不好，可以自行调节 PID 参数。

*注3：使用 lock steering to ... 调姿时，kOS 会先处理 Pitch 和 Yaw 方向的转动，等到转到差不多的了才会开始 Roll 方向的转动。

这种操作模式适用于控制火箭这种缓慢均匀的转向。

如果要用在飞机或其他东西上，建议自己写飞控。

8.2.3 底层控制 Control [结构体]

===== 点击以返回目录 =====



(1) 控制结构体

底层控制专注于载具每个方向的精细操控。

能实现的功能和玩家手动操控相同。

底层控制基于载具结构体Vessel 下的控制结构体Control。

玩家可以使用 Control结构体 来写载具的飞控。

Control结构体 (底层控制部分)

结构体成员	类型和范围	读写性	说明	操作键
MainThrottle	Scalar [0, 1]	读写	节流阀	LEFTCTRL LEFTSHIFT
Yaw	Scalar [-1, 1]	读写	俯仰	D, A
Pitch	Scalar [-1, 1]	读写	偏航	W, S
Roll	Scalar [-1, 1]	读写	滚转	Q, E
Rotation	Vector	读写	旋转 (三者综合)	
YawTrim	Scalar [-1, 1]	读写	俯仰配平	ALT + d ALT + A
PitchTrim	Scalar [-1, 1]	读写	偏航配平	ALT + W ALT + S
RollTrim	Scalar [-1, 1]	读写	滚转配平	ALT + Q ALT + e
Fore	Scalar [-1, 1]	读写	前后平移	N, H
Starboard	Scalar [-1, 1]	读写	左右平移	L, J
Top	Scalar [-1, 1]	读写	上下平移	I, K
Translation	Vector	读写	平移 (三者综合)	
WheelSteer	Scalar [-1, 1]	读写	轮胎转向 (从北方起算顺时针角度)	A, D
WheelThrottle	Scalar [-1, 1]	读写	轮胎动力 前进 / 后退	W, S
WheelSteerTrim	Scalar [-1, 1]	读写	轮胎转向配平	ALT + A ALT + d
WheelThrottleTrim	Scalar [-1, 1]	读写	轮胎动力配平	ALT + W ALT + S
Neutral	Boolean	只读	是否在进行 Control控制	
Neutralize	Boolean	读写	设 True值 可解锁 Control控制	
.....

(2) 解锁控制

如果 kOS 使用过 Control结构体，那么当 kOS 的运行结束时，

大多数底层控制不会把控制权会交还给玩家，

因为一旦使用Control类命令，玩家键盘输入的手动操作就会被遮蔽。

这时候就需要下面的语句来解锁控制，交还控制权。

```
Set Ship:Control:Neutralize TO True. //解除control控制
```

(3) 实际例子

例子1:

三个转动量:

```
Set Ship:CONTROL:PITCH to 0.6.
Set Ship:CONTROL:YAW to 0.2.
Set Ship:CONTROL:ROLL to -0.5.
```

三个平动量:

```
Set Ship:CONTROL:FORE to 0.6.
Set Ship:CONTROL:STARBOARD to 0.2.
Set Ship:CONTROL:TOP to -0.5.
```

需要为 Control 结构体 的成员赋值了一个不为 0 的数,
(分辨率是 0.00001, 低于此数值会被认作 0)
kOS 才会认为 Control 正在控制飞船。

例子2: 遮蔽玩家键盘操作

```
Set Flip to 0.000011.

function DisablePilotControl {
    parameter Disable.
    if Disable=True {
        Set Flip to -Flip.
        Set Ship:Control:Pitch to Flip.
        Set Ship:Control:Yaw to Flip.
        Set Ship:Control:Roll to Flip.}}
```

8.2.4

8.2.4 驾驶员输入 Control [结构体]

===== 点击以返回目录 =====



kOS 可以读取玩家在键盘上的驾驶输入操作。

驾驶员输入并不会实际控制载具，他的作用是允许玩家对按键信号进行编程。

*注1: 驾驶员输入与底层控制的成员几乎完全相同，在成员名称前加 Pilot 即可。

Control 结构体 (驾驶员输入部分)

结构体成员	类型和范围	读写性	说明	操作键
PilotMainThrottle	Scalar [0, 1]	读写	节流阀	LEFTCTRL LEFTSHIFT
PilotYaw	Scalar [-1, 1]	读写	俯仰	D, A
PilotPitch	Scalar [-1, 1]	读写	偏航	W, S
PilotRoll	Scalar [-1, 1]	读写	滚转	Q, E
PilotRotation	Vector	读写	旋转 (三者综合)	
PilotYawTrim	Scalar [-1, 1]	读写	俯仰配平	ALT + d ALT + A
PilotPitchTrim	Scalar [-1, 1]	读写	偏航配平	ALT + W ALT + S
PilotRollTrim	Scalar [-1, 1]	读写	滚转配平	ALT + Q ALT + e
PilotFore	Scalar [-1, 1]	读写	前后平移	N, H
PilotStarboard	Scalar [-1, 1]	读写	左右平移	L, J
PilotTop	Scalar [-1, 1]	读写	上下平移	I, K
PilotTranslation	Vector	读写	平移 (三者综合)	
PilotWheelSteer	Scalar [-1, 1]	读写	轮胎转向 (从北方起算的顺时针角度)	A, D
PilotWheelThrottle	Scalar [-1, 1]	读写	轮胎动力 前进 / 后退	W, S
PilotWheelSteerTrim	Scalar [-1, 1]	读写	轮胎转向配平	ALT + A ALT + d
PilotWheelThrottleTrim	Scalar [-1, 1]	读写	轮胎动力配平	ALT + W ALT + S
PilotNeutral	Boolean	只读	是否在进行 Control 控制	
.....

例子：监控 WASD 的驾驶员输入

```
Lock Steering to up.
until false
{ print Ship:CONTROL:PILOTPitch.
  print Ship:CONTROL:PILOTYAW.
  print Ship:CONTROL:PILOTroll.
  wait 1.}
```

(1) 实际例子

一般在驾驶飞机时，W,S键控制俯仰Pitch，A,D键控制偏航Yaw，Q,E控制滚转Roll。

本实例将演示反转W,S的功能，将滚转控制嫁接到A,D键上。

首先我们需要一架飞机，我们把游戏自带的某一架原版飞机，机腹下挂的导弹拆掉，贴上kOS部件直接拿来用。



飞机上需要运行的程序如下：

```
//kOS有一个特点是“只要程序设定的转向操控值大于小分辨率0.00001,
//玩家的WASDQE输入就会无效，但还是能被PilotInput识别的。
function DisablePilotControl {
    parameter Disable.
    if Disable=True {
        Set Flip to -Flip.
        Set Ship:Control:Pitch to Flip.
        Set Ship:Control:Yaw to Flip.
        Set Ship:Control:Roll to Flip.
    }
    ×
    Set Flip to 0.000011. //初始化最小可识别操作量×
    until FALSE {
        DisablePilotControl (TRUE). //初始化先Disable掉玩家控制
        //将玩家W,S键Pitch反转嫁接回去×
        Set Ship:Control:Pitch to -Ship:Control:PilotPitch.
        //将玩家A,D键Yaw嫁接到Roll上×
        Set Ship:Control:Roll to Ship:Control:PilotYaw.
        wait 0.001. } //等待下一个物理帧
}
```

注：飞的时候别开SAS

8.2.5

8.2.5 飞船系统

===== 点击以返回目录 =====



除了之前那些对飞船的整体控制，kOS 还支持对飞船某种功能的统一控制。

(1) 功能开关

以下是原版 KSP 游戏里就有的功能开关

变量	类型	读写性	说明
RCS	Boolean	读写	控制 RCS功能 的开关 [R键]
SAS	Boolean	读写	控制 SAS功能 的开关 [T键]
SASMode	String	读写	为 SAS 功能设置工作模式, 有以下可选 "PROGRADE" "RETROGRADE" "NORMAL", "ANTINORMAL" "RADIALOUT" "RADIALIN", "TARGET" "ANTITARGET" "MANEUVER", "STABILITYASSIST" "STABILITY"
NavMode	String	读写	为导航球设置模式, 有以下可选 "ORBIT" "SURFACE" "TARGET"
Lights	Boolean	读写	控制所有 灯光 的开关 [U键]
Brakes	Boolean	读写	控制所有 刹车 的开关 [B键]
Gear	Boolean	读写	控制所有 起落架和着陆架展开 的开关 [G键]
Abort	Boolean	读写	控制 任务终止 的开关
Ag1	Boolean	读写	控制 动作组1 的开关 [大键盘1键]
.....	Boolean	读写	控制 动作组..... 的开关
Ag9	Boolean	读写	控制 动作组9 的开关 [大键盘9键]
Ag10	Boolean	读写	控制 动作组10 的开关 [大键盘0键]
.....			

以下是kOS 追加的功能开关

变量	类型	读写性	说明
Legs	Boolean	读写	控制所有 着陆架展开 的开关
Chutes	Boolean	读写	控制所有 降落伞展开 的开关 (只能用On指令)
ChutesSafe	Boolean	读写	控制所有 符合展开条件的降落伞展开 的开关 (只能用on指令)
Panels	Boolean	读写	控制所有 太阳能板展开 的开关
Radiators	Boolean	读写	控制所有 天线展开 的开关
Ladders	Boolean	读写	控制所有 着陆架展开 的开关
Bays	Boolean	读写	控制所有 货仓开合 的开关
DeployDrills	Boolean	读写	控制所有 钻头展开 的开关
Drills	Boolean	读写	控制所有 钻头工作 的开关
FuelCells	Boolean	读写	控制所有 燃料电池工作 的开关
ISRU	Boolean	读写	控制所有 资源转换器 的开关
Intakes	Boolean	读写	控制所有 进气道开启 / 关闭 的开关
.....			

(2) 功能开关例子

例子1：简单操控

```

toggle AG1. //打开/关闭动作组AG1, 相当于按了一下按键1
toggle AG3. //打开/关闭动作组AG3, 相当于按了一下按键1
toggle SAS. //打开/关闭SAS, 相当于按了一下按键T
toggle RCS. //打开/关闭RCS, 相当于按了一下按键R
SAS on. //打开SAS, 相当于如果SAS没有开启, 就按按键T
RCS off. //关闭RCS, 相当于如果RCS没有关闭, 就按按键R
Light off. //关所有灯, 相当于没开灯, 就按按键L
Brakes on. //刹所有能刹车的部件相当于如果没刹车, 就按按键B
Gear off. //拉起所有起落架和登陆架, 相当于按按键G
Legs on. //放下所有着陆架, 相当于按按键G, 不过只作用于登陆架
Chutes on. //展开所有降落伞 (这条只能搭配on命令)
ChutesSafe on. //展开所有符合安全展开条件的降落伞 (这条只能搭配on命令)
Panels on. //展开所有太阳能板
Radiators on. //展开所有天线
Ladders off. //放下所有着陆架
Bays on. //打开所有货仓
DeployDrills on. //展开所有钻头
Drills on. //所有钻头开始工作
FuelCells off. //停止所有燃料电池的工作
ISRU on. //启动所有资源转换器
Intakes off. //关闭进气道
ABORT on. //按下任务终止按钮

```

例子2：用触发结构对功能开关进行编程

```

ON ABORT {
    PRINT "Aborting!".
}

```

*注1：Ag1~Ag10 对应的动作组。

动作组是只对从 False 变到 True 的“上升沿瞬间”有反应。

使用动作组时，设 True 值之后别忘了再设回 False 值，不然下次没法用。

```

Ag1 On. //第一次激发动机组
Wait 1.
Ag1 On. //此句无效果
Wait 1.
Ag1 Off.
Wait 1.
Ag1 On. //第二次激发动机组

```

(3) 目标物

kOS 还支持设置目标物，目标物可以是 星球Body、载具Vessel、部件Part。

```

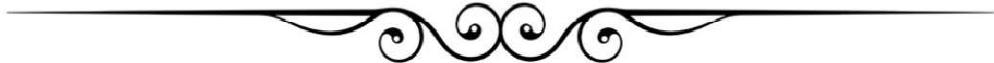
Set Target to Mun.
Set Target to Vessel("Untitled Space Craft").
Set Target to a. //a 是附近其它载具上的某个部件

```

8.3

8.3 飞行路径预测

----- 点击以返回目录 -----



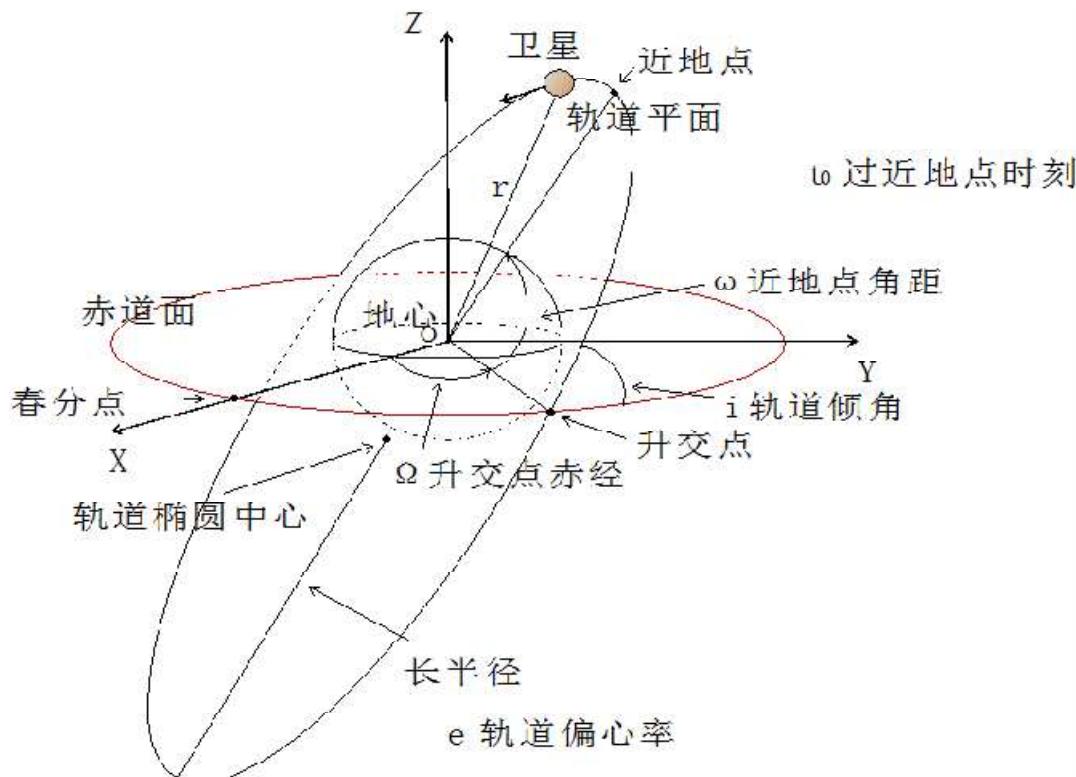
kOS 中能进行轨道预测的只能是静态轨道，
静态轨道是指载具受到的外力只有重力情况下的轨道。

*注1：安装了 Trajectories 落点预测MOD 之后，kOS 还能预测载具在大气层中，在重力和空气阻力共同作用下的轨道。该轨道数据仅有参考作用。
用法详见落点预测 Trajectories (TR)，在此不多述

(1) 轨道要素

轨道要素又称轨道根数，通过他们可以确定轨道平面在空间的方位，轨道在轨道平面中的方位，轨道的形状和航天器在轨道上的位置。知道了轨道要素之后，所有轨道信息都可以计算得出。轨道要素一共有六个，他们都是便于观测得出或直观理解的：

轨道长半轴 a 偏心率 e 平近角 θ
轨道倾角 i 升交点精度 Ω 近心点角距 ω
当六个轨道要素确定了之后，轨道也就能唯一确定了。



轨道长半轴 a —— 椭圆轨道的长轴*0.5

偏心率 e —— 椭圆轨道的离心率(偏心率)，可以通过公式算出半焦距 c ，

其中 $a+c = AP$ 点高度+星球半径, $a-c = PE$ 点高度+星球半径

平近点角 θ —— 椭圆轨道上航天器所在位置与升交点之间的夹角。

这个已经和轨道没关系了,

关系到的是航天器运行到轨道的一圈的哪个位置了

轨道倾角 i —— 椭圆轨道平面和赤道面的夹角, 范围 $0 \sim 180^\circ$

升交点赤经 Ω —— 椭圆轨道和赤道面有两个交点, 升交点是沿轨道移动时,

从南半球切换到北半球的那个交点。

近心点角距 ω —— 在轨道平面上, 从进心点 (PE点) 到升交点之间的夹角, 范围 $0 \sim 360^\circ$ 。

*注2: 使用 HyperEdit 这个 MOD, 通过拉动轨道参数的滑块, 能直观的理解每个参数对应意义。下图中轨道参数数据依次为:
轨道倾角, 偏心率, 远地点 (轨道长半轴-星球半径),
升交点赤经, 进心点角距, 平近点角。



(2) 获取载具的轨道要素

在 kOS 中可以从 Orbit 结构读取得到轨道参数。

```
Ship:Obt:SemiMajorAxis //轨道长半轴
Ship:Obt:Eccentricity //偏心率
Ship:Obt:SemiMajorAxis //平近点角
Ship:Obt:Inclination //轨道倾角
Ship:Obt:Lan //升交点赤经
Ship:Obt:ArgumentOfPeriapsis //近心点角距
```

除了以上信息 kOS 还有提供其他有用的信息。

```
Ship:Obt:Period //轨道周期
Ship:Obt:TrueAnomaly //真近点角
```

(3) 飞行路径预测

kOS 还能进行飞行预测。

```
OrbitAt(Orbitable,time) //所处轨道
PositionAt(Orbitable,time) //位置预测函数, 用于航天器的整个飞行
VelocityAt(Orbitable,time) //速度预测函数, 用于航天器的整个飞行
```

例子：

```
Print PositionAt(Ship,Time:Seconds+600) //当前载具600秒后的位置
Print VelocityAt(Ship,Time:Seconds+600) //当前载具600秒后的速度
```

(4) 变轨/逃逸/捕获 后的轨道信息和路径预测

变轨后的轨道，和逃逸/捕获后的轨道一样，他们都是航天器将要历经轨道中的一段，都是以轨道列表形式存储在Vessel:Patches中的。

```
Ship:Patches. //当前载具的轨道列表
Ship:Patches[0]. //当前载具的当前轨道
Ship:Patches[1]. //当前载具的下一段轨道
Ship:Patches[1]:HasNextPatch. //当前载具的下一段轨道他有没有再下一段的轨道
Ship:Patches[2]. //当前载具的第三段轨道
```

kOS 能对轨道列表里的这些新轨道进行路径预测。

```
//假设当前载具500秒后进入新轨道

Set a to Ship:Patches[1]. //当前飞船的下一段轨道
Print PositionAt(a,Time:Seconds+600) //当前载具600秒后的位置
Print VelocityAt(a,Time:Seconds+600) //当前载具600秒后的速度
```

8.4

8.4 列表List（指令）

===== 点击以返回目录 =====



在 kOS 中，列表List 关键词有两个角色：一种结构体类型、一种指令。

这里说的就是 列表List 作为类型的特性和用法。

列表List命令 仅用于对预设列表的操作。

一共有四种用法：

(1) List.

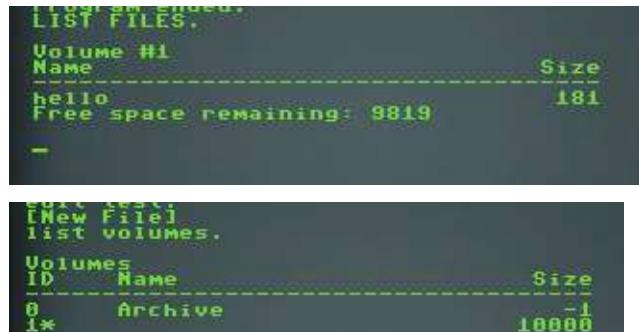
单独的 List 可以成句。List. 表示显示文件预设列表。

```
List.
List Files. //这两句一个意思
```

(2) List 预设列表.

List 后可以跟 kOS 预设列表，表示显示这些预设列表。

```
List Volumes. //在屏幕上显示飞船当前能访问到的所有Volume
List Files. //在屏幕上显示当前路径下所有的File
```



List 后可以跟的预设列表如下：

变量	类型	说明
Bodies	List of Body	天体的列表 (不包含小行星)
Targets	List of Vessel	可设为目标的载具列表 (包含小行星)
Fonts	List of String	可在 Style : Font 或 Skin : Font 中 [样式] 使用的字体的名称
Processors	List of Processor	当前载具 kOS 处理器 的列表
AggregateResources	list of AggregateResource	当前载具上资源总量的列表
Parts	List of Part	当前载具上部件的列表
Engines	List of Engine	当前载具上引擎类部件的列表
Sensors	List of Sensor	当前载具上传感器类部件的列表
Elements	List of Element	当前载具上对接元素的列表
DockingPorts	List of DockingPort	当前载具上对接口类部件的列表
Files	List of File	当前卷 Volume 上文件 File 的列表
Volumes	List of Volume	当前载具上可访问卷 Volume 的列表
.....

(3) List 预设列表 in A.

```
LIST ListKeyword IN YourVariable.
//将ListKeyword现成对象另存到YourVariable变量里
```

(4) List 预设列表 From 载具 in A.

```
ListKeyword FROM SomeVessel IN YourVariable.  
//将SomeVessel飞船上的ListKeyword现成对象另存到YourVariable变量里
```

(5) 实际例子

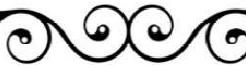
例子1:

```
List. //作用和在指令窗口使用List Files命令相同，在屏幕上显示当前Volume内所有的File  
list ENGINES. //表示当前飞船中的引擎的列表显示在控制台屏幕上。  
list ENGINES in MyList. //表示将当前飞船中的引擎的列表存为数组MyList  
//MyList[0]表示这个数组的第一个元素  
List ENGINES From PPS in MyList. //表示将PPS飞船中的引擎的列表存为数组MyList
```

8.5

8.5 查询部件信息

===== 点击以返回目录 =====



kOS 提供了几个函数，可以根据部件名Name、标题名Title、标签名Tag 来搜索并指定部件。指定部件之后可以获得该部件的部件结构体Part，玩家可以据此对该部件做进一步操作，例如查看部件信息，操作部件的右键菜单。

```
Ship:PartsNamed(string) //返回当前载具上 部件名Name 里包含 字符串string 的部件构成的列表  
Ship:PartsTitled(string) //返回当前载具上 标题名Title 里包含 字符串string 的部件构成的列表  
Ship:PartsTagged(string) //返回当前载具上 标签名Tag 里包含 字符串string 的部件构成的列表  
//返回当前载具上 部件名Name、标题名Title、标签名Tag 里包含字符串string的部件构成的列表  
Ship:PartsDubbed(string)
```

以标签名Tag 为例，我们可以这么用：

```
Ship:PartsTagged("aaa"). //符合Tag名aaa的所有部件构成的列表  
Set parts to Ship:PartsTagged("aaa"). //parts 是符合Tag名aaa的所有部件构成的列表  
Set part to parts[0]. //part 是 parts 的第一个成员。  
  
Set part to Ship:PartsTagged("aaa") [0]. //这句和前两句一个意思  
  
//然后我们就可以对我们已经指定的部件 part 进行进一步操作了。  
Print part:Mass. //比如说显示这个部件的质量  
Print part:Thrust. //如果这是一个引擎部件，我们还可以得到这个引擎的推力。  
part:Unlock(). //如果这是一个对接口部件，我们还可以让他解除对接  
.....等等用途
```

推荐用 Tag 作为 kOS 中调用部件的手段。因为 Tag 可以随时编辑，泛用性高。

另外还有一种用法，是以“部件有Tag命名”作为搜索条件的

```
Ship:AllPartsTagged(). //有Tag名的所有部件构成的列表
```

8.6

8.6 文件 I/O

8.6.1

8.6.1 理解文件目录

===== 点击以返回目录 =====



kOS 上的文件系统和现实世界里电脑的文件系统一样，有盘符（卷Volume），有文件夹，有子文件夹，有路径这些概念。这样子可以形成文件路径的一个树状结构。

8.6.2

8.6.2 路径

===== 点击以返回目录 =====



现实世界中 Windows 电脑上的一个完整路径也许是：

D:\Program Files (x86)\Steam\Steam.exe

现实世界中 Linux 电脑上的一个完整路径也许是：

/home/user/somefile

kOS 里的完整路径类似 Linux 系统的写法：

0:/acdc/new.txt

kOS 中的路径都是字符串，
kOS 中存在两种路径： **绝对路径** 和 **相对路径**。

(1) 绝对路径

```
"0:/lib/launch/base.ks" //文件 base.ks
"0:/lib/launch/base" //文件 base

"0:/lib/../base.ks" //这一句和下面一句是一个意思，两点..表示返回上一级子目录
"0:/base.ks"
"0:/.." //这句非法，卷的顶层已经没有上一级了
```

在路径中类似 .. 这样的写法，表示文件向上一级。

(2) 当前路径和相对路径

相对路径要有一个起始文件夹，CD 函数就是用来设定这个其实文件夹的

```
CD("0:/acdc"). //将当前文件夹路径设为 0:/acdc ,如果该文件不存在则创建文件夹
Print Path(). //显示当前文件夹路径
```

当前路径默认为当前 卷Volume 的根目录。

以设定的当前路径作为起始文件夹开始算，指代的路径

```
CD("0:/acdc"). //将当前文件夹路径设为 0:/acdc
"file.ks" //代表的是 0:/acdc/file.ks
"../file.ks" //和上面一样
"/top.ks" //注意这个和上面不一样， /代表从卷开始算的，这个代表的文件是 0:/top.ks
```

(3) 与路径有关的结构体

Path路径、Volume盘符（卷）、VolumeFile盘符文件、VolumeDirectory盘符路径。
玩家可以在盘符与文件查看具体信息。

(4) 与路径有关的部分函数

以下列举了几个本小节提到的与路径有关的函数。
更多相关函数请参考 路径 Path [结构体]。

((1)) 当前路径 Path()

Path函数 的返回值是当前路径。

```
Print Path().
```

((2)) 设定当前路径 CD(pathString)

CD函数 用于设定当前路径。

```
CD("0:/scripts").
```

((3)) 复制路径 CopyPath(FromPath,ToPath)

CopyPath函数 用于复制路径。

```
//把上一级文件夹里 launch.ks 文件夹复制到当前路径
CopyPath("../launch.ks", "").
```

例子：

```
COPYPATH(myfilename, "1:"). // This is an example of a bareword filename.
COPYPATH("myfilename", "1:"). // This is an example of an EXPRESSION filename.
COPYPATH(myfilename.ks, "1:"). // This is an example of a bareword filename.
COPYPATH(myfilename.txt, "1:"). // This is an example of a bareword filename.
COPYPATH("myfilename.ks", "1:"). // This is an example of an EXPRESSION filename
SET str TO "myfile" + "name" + ".ks".
COPYPATH(str, "1:"). // This is an example of an EXPRESSION filename
COPYPATH("myfile" + "name" + ".ks", "1:"). // This is an example of an EXPRE
```

((4)) 查询程序文件的路径 ScriptPath(pathString)

ScriptPath函数 的返回值是当前程序文件所在的路径。

```
Print ScriptPath().
```

8.6.3

8.6.3 盘符

===== 点击以返回目录 =====



玩家可以通过 Volume ID 或者 Volume Name 来调用盘符（卷）。

玩家可用过 Switch 命令来切换盘符

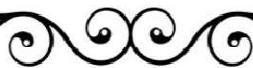
例子：

```
SWITCH TO 0.          // 切换到 Volume 0.
SET VOLUME(1):NAME TO "AwesomeDisk". // 将 Volume 1 重命名为 "AwesomeDisk".
SWITCH TO "AwesomeDisk".      // 切换到 Volume 1.
PRINT VOLUME(1):NAME.        // Prints "AwesomeDisk".
```

8.6.4

8.6.4 文件和目录

===== 点击以返回目录 =====



(1) List

List 指令可显示当前路径下所有的文件信息

```
List.
List Files. //这两句一个意思
```

(2) CD(PATH)

CD函数 用于设定当前路径。

```
CD("0:/scripts").
```

(3) CopyPath(FromPath,ToPath)

CopyPath函数 用于复制路径。

```
//把上一级文件夹里 launch.ks 文件夹复制到当前路径
CopyPath("../launch.ks", "").
```

例子：

```
COPYPATH(myfilename, "1:"). // This is an example of a bareword filename.
COPYPATH("myfilename", "1:"). // This is an example of an EXPRESSION filename.
COPYPATH(myfilename.ks, "1:"). // This is an example of a bareword filename.
COPYPATH(myfilename.txt, "1:"). // This is an example of a bareword filename.
COPYPATH("myfilename.ks", "1:"). // This is an example of an EXPRESSION filename
SET str TO "myfile" + "name" + ".ks".
COPYPATH(str, "1:"). // This is an example of an EXPRESSION filename
COPYPATH("myfile" + "name" + ".ks", "1:"). // This is an example of an EXPRE
```

(4) MovePath(FromPath,ToPath)

MouvePath函数 用于移动路径。

```
//把上一级文件夹里 launch.ks 文件夹移动到当前路径
CopyPath("../launch.ks", "").
```

(5) DeletePath(Path)

DeletePath函数 用于删除路径。

```
//删除上一级文件夹里 launch.ks 文件夹
DeletePath("../launch.ks").
```

(6) Exists(Path)

Exists函数 用于检查路径是否存在，返回的是逻辑值。

```
If Exists("boot/abc.txt") {
    Print "OK".
}
```

(7) Create(Path)

Create函数 用于创建路径（文件）。

```
Create("boot/abc.txt").
```

(8) CreateDir(Path)

CreateDir函数 用于创建路径（文件夹）。

```
CreateDir("boot/acdc").
```

(9) Open(Path)

Open函数 用于打开文件/文件夹，其返回值是 VolumeFile 或 VolumeDirectory 类型。

之后可以对文件内容经行读写，对文件夹信息进行获取。

```
Set file1 to Open("boot/abc.txt").
Print file1:ReadAll.      //显示这个文件的内容（仅文本文件有效）
file1:Write("www").      //在这个文件末尾加上 "www"
file1:WriteLn("qqq").     //在这个文件末尾加上 "qqq" 并回车

Set dir1 to CreateDir("boot/acdc").
Print dir1>List.         //显示这个文件夹下的文件/文件夹列表
```

8.6.5 对 json 文件进行读写

===== 点击以返回目录 =====



(1) json 文件

JSON文件 是一种轻量级的数据交换文件。

kOS 为 JSON文件 的读写操作提供了 ReadJson函数 和 WriteJSON函数。

JSON文件使用记事本打开是这种样子的：

```
{
    "items": [
        {
            "items": [
                11,
                12,
                13
            ],
            "$type": "kOS.Safe.Encapsulation.ListValue"
        },
        {
            "items": [
                21,
                22,
                23
            ],
            "$type": "kOS.Safe.Encapsulation.ListValue"
        },
        {
            "items": [
                31,
                32,
                33
            ],
            "$type": "kOS.Safe.Encapsulation.ListValue"
        }
    ],
    "$type": "kOS.Safe.Encapsulation.ListValue"
}
```

(2) 写入 json 文件

玩家可使用 WriteJSON函数 写入 json文件。

WriteJSON (object, path)

将 object内容 写入 路径path 对应的文件， object 必须是可序列化的类型。

使用此函数导出的文件的后缀是 .json。

例子：

```
SET L TO LEXICON().
SET NESTED TO QUEUE().
L:ADD("key1", "value1").
L:ADD("key2", NESTED).
NESTED:ADD("nestedvalue").
WRITEJSON(L, "output.json").
```

(3) 读取json文件

玩家可使用 **ReadJSON** 函数 读取 json 文件。

ReadJSON (path)

将路径 **path** 对应文件的内容读出，要读出的内容是可序列化的类型。

使用此函数读取的文件必须是 **.json** 文件。

例子：

```
SET L TO READJSON("output.json").
PRINT L["key1"].
```

(4) 实际例子

我们来尝试将一个二维数组传递到 json 文件，再读取回来。

这个二维数组是由两层 List 列表嵌套构造起来的

```
//导出部分
Set w1 to List(11,12,13).
Set w2 to List(21,22,23).
Set w3 to List(31,32,33).
Set w to List(w1,w2,w3). //创建一个二维数组
print w. //显示这个二维数组
WriteJSON(w,"data.json"). //导出为json文件

//读取部分
Set r to ReadJSON("data.json"). //读取保存为r
print r.
print r[1].
print r[1][1]. //依次显示整个二维数组，整行，某一个数
```

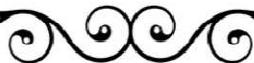
程序运行结果

```
runpath(test).
LIST of 3 items:
[0] = LIST of 3 items:
[0] = 11
[1] = 12
[2] = 13
[1] = LIST of 3 items:
[0] = 21
[1] = 22
[2] = 23
[2] = LIST of 3 items:
[0] = 31
[1] = 32
[2] = 33
LIST of 3 items:
[0] = LIST of 3 items:
[0] = 11
[1] = 12
[2] = 13
[1] = LIST of 3 items:
[0] = 21
[1] = 22
[2] = 23
[2] = LIST of 3 items:
[0] = 31
[1] = 32
[2] = 33
LIST of 3 items:
[0] = 21
[1] = 22
[2] = 23
22
Program ended.
```

8.6.6

8.6.6 用 Log 指令进行文件写入

===== 点击以返回目录 =====



Log 指令能以字符串形式写入文本文件。

若目标文件不存在则新建，若已存在末尾追加文本。

log str to filename.

例子：

```
Set a to 45.
log "A=" + a to logfile.
```

8.6.7

8.6.7 文件读写总结

===== 点击以返回目录 =====



在 kOS 中有三种方式可以读取文件，有三种方式可以写入文件，具体如下。

类型	方式	说明
写入文件	Log inf to filename.	将字符串inf添至filename文件名的文件，然后插入换行符
	VolumeFile : Write (inf) VolumeFile : WriteLn (inf)	将信息inf添至VolumeFile类型的量对应的文件内容。 Write函数的inf可为字符串，也可为VolumeFile类型。 WriteLn函数的inf只能为字符串，添加字符串之后会再加一个换行符
	WriteJSON (inf, filename)	向filename文件名的json文件内写入可序列化信息inf。
读取文件	VolumeFile : ReadAll	从VolumeFile类型的量对应的文件以字符串形式读取全部文件内容
	FileContent : String	从FileContent类型的量对应的文件以字符串形式读取全部文件内容
	ReadJSON (filename)	从filename文件名的json文件内读取可序列化信息inf。

有关 Log 指令，可以查看用 Log 指令进行文件写入；

有关 WriteJSON 函数 和 ReadJSON 函数，可以查看对 json 文件进行读写；

有关 VolumeFile 结构下的读写函数，可以查看 盘符文件 VolumeFile [结构体]；

有关 FileContent 结构下的读取函数，可以查看 文件内容 FileContent [结构体]。

8.7

8.7 指令窗口和 GUI

8.7.1

8.7.1 指令窗口操作

===== 点击以返回目录 =====



(1) ClearScreen 指令

ClearScreen指令 用于指令窗口的清屏。

```
ClearScreen.
```

(2) Print 指令

Print指令 用于在指令窗口中以字符串形式显示信息。

```
Print "abc".
Print "abc"+123.
```

(3) At(Col,Line)定位函数(Print 指令用)

At函数 用于搭配 Print指令 一起使用。

用于为 Print指令 显示的位置， 指定行数和列数。

```
PRINT "Hello" AT(0,10).
PRINT 4+1 AT(0,10).
PRINT "4 times 8 is: " + (4*8) AT(0,10).
```

(4) 设置指令窗口的尺寸

玩家可调整指令窗口的尺寸， 他是以显示字符的行数/列数为基准调节的。

```
Set Terminal:Width to 20.
Print Terminal:Width.
Set Terminal:Height to 15.
Print Terminal:Height.
```

(5) MapView 地图视角

MapView 是一个逻辑量， 用来开关小地图。

True值 表示处于小地图状态。 False值 表示处于载具驾驶状态。

```
MapView On.
MapView Off.
```

(6) Reboot 重启指令

重启当前程序所在的 kOS处理器。

```
Reboot.
```

(7) Shutdown 关机指令

关闭当前程序所在的 kOS 处理器。

```
Shutdown.
```

8.7.2

8.7.2 屏幕显示工具

===== 点击以返回目录 =====



(1) VecDraw / VecDrawAges 矢量箭头

下图是展示在屏幕上显示从当前载具动态指向目标载具箭头的画面。



矢量箭头还可以设置为隐藏箭头只显示文字，例如下图的锁定框。



像下面两段代码一样，通过 VecDraw / VecDrawAges 矢量箭头，可以实现这样的效果：

红色箭头的代码：

```

Set a to Vessel("www"). //选取载具名为 www 的载具
//创建矢量箭头，参数分别为 起点矢量、终点矢量、颜色、文本、
//、箭头长度比例(正数有效。实际箭头为从起点出发，朝终点矢量方向的，箭头长度为此比例值*起点终
//、是否显示箭头、箭头宽度(同时也影响字号)
Set b to VecDraw(V(0,0,0) , a:Position , RED , "Target" , 0.5 , True , 0.8).
Until False { //每个物理帧都刷新 www 载具的位置矢量
    Set b:Vec to a:Position.
    Wait 0.001.
}

```

绿色锁定框的代码：

```

set TargetCraft to vessel("www"). //选取载具名为 www 的载具
set colorT1 to RGBA(0,1,0,0). //完全透明的绿色，透明要素会在箭头有效，在文字无效
//创建矢量箭头，参数分别为 起点矢量、终点矢量、颜色、文本、
//、箭头长度比例(正数有效。实际箭头为从起点出发，朝终点矢量方向的，箭头长度为此比例值*起点终
//、是否显示箭头、箭头宽度(同时也影响字号)
set VDcraft to VecDraw(v(0,0,0), TargetCraft:Position, colorT1, "□",2,True,0.2).

until false { //每个物理帧都刷新 www 载具的位置矢量
    set VDcraft:Vec to TargetCraft:Position.
    wait 0.001.
}

```

*注1： kOS 矢量箭头是根据位置矢量画出的，
玩家获取载具的位置信息，还能获取载具上部件的位置信息。
使用起来很自由。

(2) HudText 屏显文字

下图是展示在屏幕的四个方位显示不同字号和颜色的文字的画面。



像下面的代码一样，通过 HudText 屏显文字，
可以实现这样的效果：

```
//参数依次为：文本、显示时长、对齐方式、字号、颜色、是否显示
//其中对齐方式只有如下四种可选.....
Set a1 to HudText("上左" , 10 , 1 , 32 , RGB(1.0,0,0) , True) .
Set a2 to HudText("上中" , 10 , 2 , 34 , RGB(0.8,0,0) , True) .
Set a3 to HudText("上右" , 10 , 3 , 36 , RGB(0.6,0,0) , True) .
Set b1 to HudText("中中" , 10 , 4 , 30 , RGB(0.4,0,0) , True) .
```

注2：HudText 屏显文字会在显示时效过后会自动清除数据。无需玩家担心。

(3) HighLight 部件高亮

在游戏中，任何时候，当玩家把鼠标移到某个部件上时，这个部件就会回绿色高亮显示，如下图。

当一个部件过热时，这个部件也会发红/发黄，这里也用到了部件高亮。



kOS 允许玩家以自己需求和喜好，对任意部件设置高亮显示。如下例子：

```
Set p to Ship:PartsTagged("qqq") [0]. //按照部件标签名选取部件
SET foo TO HIGHLIGHT( p, HSV(350,0.25,1) ). //创建 HighLight 结构
Wait 10. //等10秒钟
SET foo:ENABLED TO FALSE. //关闭部件高亮
Wait 10. //等10秒钟
SET foo:ENABLED TO TRUE. //重新开启部件高亮
```

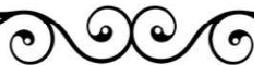
出来的效果是这样的：



8.8

8.8 可序列化

===== 点击以返回目录 =====



kOS 中的结构体类型，有一部分是 [可序列化](#) 的。

可序列化是指可以将数据转存成文件，之后读取这个文件还能无损地得到原数据。

在 kOS 中，可序列化的结构体类型有以下两个特征：

1. 能通过 WriteJSON 函数 和 ReadJSON 函数 读写数据到 JSON 文件。
2. 能被做成通讯报文，与其他 kOS 处理器 互传通信数据。

*注1：Lexicon, List, Queue, Range, Stack, UniqueSet 结构体的成员必须也是可序列化的，

不然会无法进行文件读写和通信传输。

8.9

8.9 通讯

8.9.1

8.9.1 通讯概述

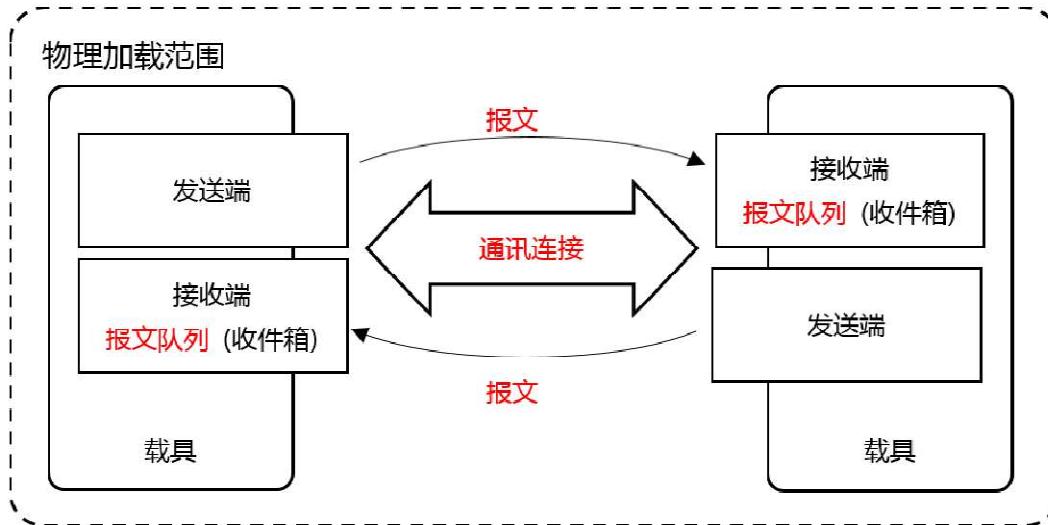
===== 点击以返回目录 =====



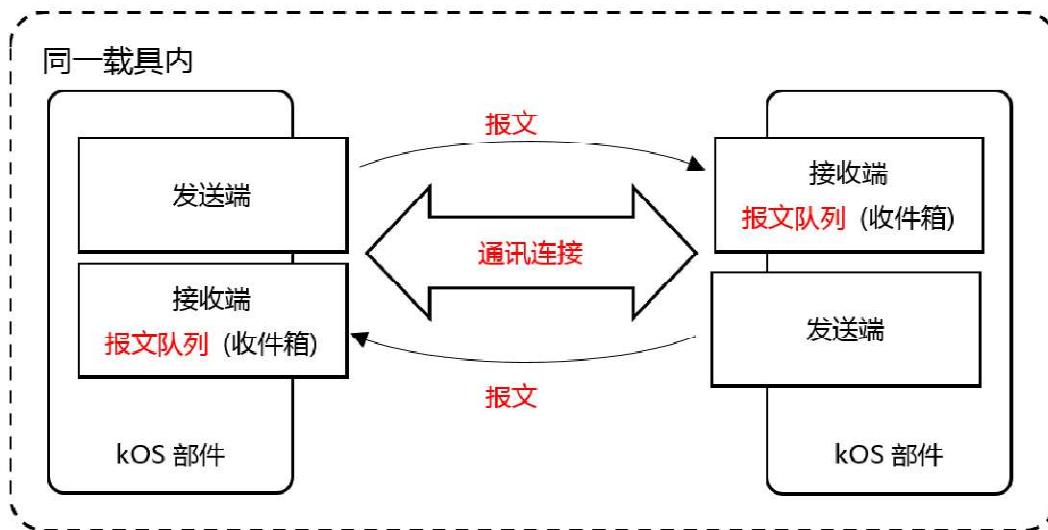
kOS 允许玩家通过 kOS 传输数据。

能被用来传输的数据都需要是可序列化的结构体类型。

有关载具间通讯的示意图如下：



有关载具内通讯的示意图如下：



*注1：载具间通讯，发送方和接收方都要在物理加载范围内，并且，发送方和接收方要有通讯信号。
如果两载具距离相隔较远，玩家需要调节那个载具的载入距离，以满足要求。

*注2：有关如何设置载具的载入距离，请参考
第四墙 KUniverse [结构体]
载具载入距离 VesselLoadDistance [结构体]
场景载入距离 SituationLoadDistance [结构体]

8.9.2 报文

===== 点击以返回目录 =====

进行通讯数据的传输时，被传输的数据被称为报文。

报文可以是 Scalar 数值、String 字符串、Boolean 逻辑量，
也可以是其他可序列化的结构体。

当玩家使用 Lexicon, List, Queue, Range, Stack, UniqueSet 这类结构体时，
其成员也必须是可序列化的结构体。

报文在发送时还会自动加上发送方的落款、接收方的抬头，以及时间戳。

8.9.3

8.9.3 报文队列

===== 点击以返回目录 =====



报文队列则相当于报文的收件箱。

每个 kOS 处理器 都有一个报文队列(CPU Queue)，载具本身也有一个报文队列(Vessel Queue)，他们都有各自的手贱抬头。

如果一个载具上有 N 个 kOS 处理器，那么一共就有 N+1 个报文队列。

玩家需要对队列进行“读件”才能得到报文。

“读件”时报文的获取顺序为序列式的“先进先出”。

被“读件”的报文会自动从报文队列中清除。

***注1：**当载具退出物理加载时，CPU Queue 中的消息会清空，Vessel Queue 中的消息会随存档一起被保存下来。

***注2：**进行超出物理加载范围的跨载具通信是不可能的，跨载具通信只能发生在同时被物理载入的不同载具之间。

8.9.4

8.9.4 通讯连接

===== 点击以返回目录 =====



通讯的主体可以是某个 kOS 部件，也可以是某个载具本身。

通讯的收发主体之间要有通讯连接才能进行报文传输。

如果通讯的收发主体在同一载具上，例如同一载具上的两个 kOS 部件，
那么他们 100% 能进行通讯。

如果通讯的收发主体在不同载具上，能否进行通讯要看他们之间是否有通讯连接。

通讯连接有三种模式，玩家初次使用 kOS 时会让玩家选择，

后期可在 Setting 菜单中变更设置：

1. PermitAllConnectivityManager

100% 能进行通讯；

2. CommNetConnectivityManager

按照原版 KSP 自身的通讯规则，如果有连接两者载具的通讯信号，不管多微弱，只要不是 No Connected，就可以通讯。

3. RemoteTechConnectivityManager

按照 通讯MOD 的通讯规则，如果有连接两者载具的通讯信号，不管多微弱，只要不是 No Connected，就可以通讯。（选择此项需要预先安装该 MOD）



*注1：通讯连接模式为 RemoteTechConnectivityManager 时，如果 通讯MOD 设置了通讯延迟，那么 报文的送达也会有相应延迟。

另外，kOS 还提供两个预设变量，
用于查询到 KSC 的通讯连接，以及到控制源的通讯连接。

变量名	类型	说明
HomeConnection	Connection	从当前载具到 KSC 的通讯连接，此变量可用于检查是否能从当前载具访问 KSC 的数据库Archive
ControlConnection	Connection	从当前载具到当前控制源的通讯连接，该控制源可能是 KSC，也可能是一个载人载具
.....

*注2：向 HomeConnection 或 ControlConnection 发送报文都会报错，
因为他们都在物理加载范围。

*注3：任何类型为 debris 的载具在首次加载时，kOS 均无法连接到该载具。需要玩家将载具类型改成 非debris，然后切到 tracking station 再切回来，才能实现到改载具的通讯连接。

8.9.5

8.9.5 载具间的通讯收发

===== 点击以返回目录 =====



载具间的通讯，其收发主体，是载具本身。

首先我们假设一有个发送方载具”www1”和一个接收方载具”www2”。

(1) 发送报文

在发送方载具 “www1” 上要运行的发送指令，如下：

```

Set m to "Hello".
//c 为载具 www2 到当前载具的通讯连接
Set c to Vessel("www2"):Connection.
//SendMessage 函数如果发送成功，会返回 True，如果发送失败，会返回 False
If c:SendMessage(m) {
    Print "Sent".
}

```

(2) 接收报文

在接受方载具“www2”上要运行的接受指令，如下：

```

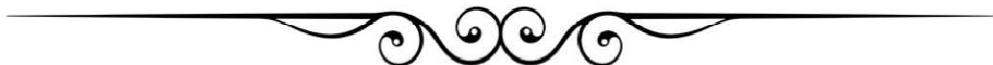
When Not Ship:Message:Empty {
    Set re to Ship:Message:Pop. //re 是一个 报文变量 Message ×
    Print re:Content. //Content 后缀表示报文内容的字符串
}

```

8.9.6

8.9.6 载具内的通讯收发

===== 点击以返回目录 =====



载具内的通讯，其收发主体，是都 kOS 处理器。

首先我们当前载具上有两个 kOS 部件，标签名分别为“part1”和“part2”。

在这里有一个很有用的函数：

Processor 函数，他可以抓取部件上和 kOS 相关的部件模组。

Processor 函数：

用法： **Processor (Para)**

返回值： 返回 **kOSProcessor** 类型

参数： **Para** 可为 **volume** 类型，也可为部件的标签名字符串 **String**

(1) 发送报文

在 part1 部件上要运行的发送指令，如下：

```

Set p2 to Processor("part2"). //p1 是 kOSProcessor 类型
Set c2 tp p2:Connection. //c2 为 part2 到 part1 的通讯连接
//SendMessage 函数如果发送成功，会返回 True，如果发送失败，会返回 False
If c2:SendMessage(m) {
    Print "Sent".
}

```

(2) 接收报文

在 part2 部件上要运行的接收指令，如下：

```
//Core 是一个预设变量，表示当前 kOS 处理器的 kOSProcessor 量
When Not Core:Message:Empty {
    Set re to Core:Message:Pop. //re 是一个 报文变量 Message ×
    Print re:Content. //Content 后缀表示报文内容的字符串
}
```

8.9.7

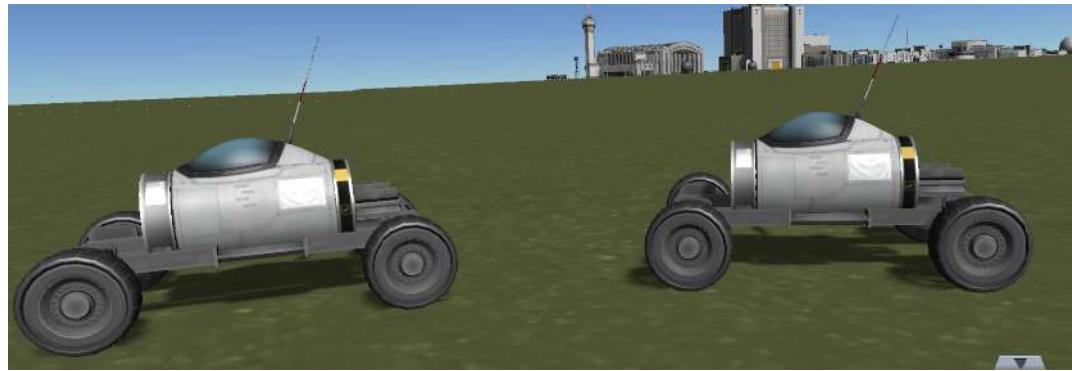
8.9.7 实例13：用通讯报文(Message)指挥僚机

===== 点击以返回目录 =====



(1) 任务目的

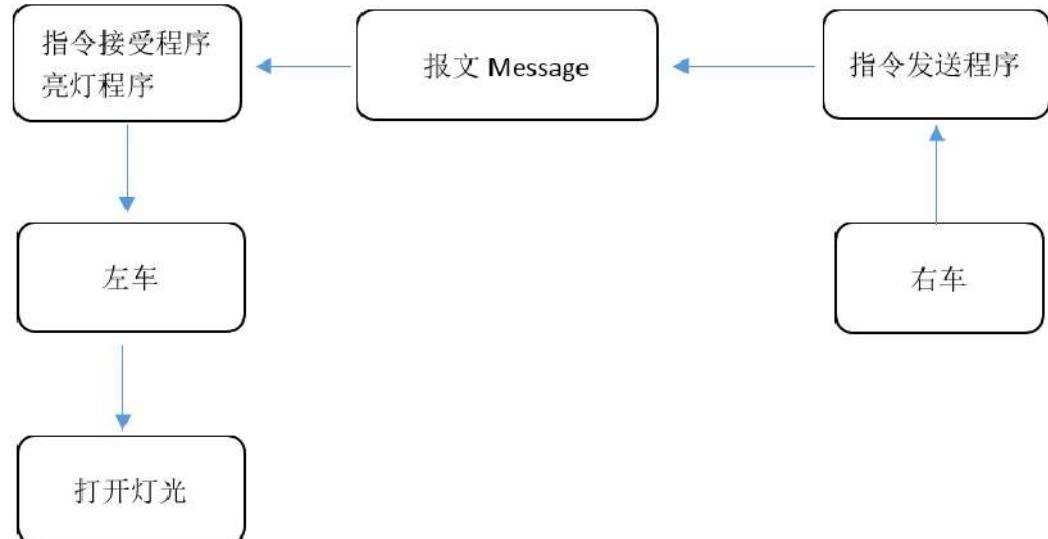
我们开来了两辆相同的 kOS 小车，左车命名为 `qwert`，驾驶舱部件的标签名 Tag 是 `www`，右车命名为 `asdfg`，驾驶舱部件的标签名 Tag 叫 `sss`。



我们要在右车上运行程序，遥控左车的驾驶舱亮灯。

(2) 方案思路

我们要让右车给左车发送 报文 Message，让左车收到命令后就打开灯光。如下图：



其中消息的内容我们设为字符串“OP”，僚机收到“OP”字符串之后，就会亮灯。

(3) 右车程序

```

Set Lship to Vessel("qwert"). //取左车载具
Set Lcn to Lship:Connection. //取从当前车（右车）到左车的通讯链路
If Lcn:IsConnected = False {
    print "Connect Failed". //如果没有连接，就报错
} else {
    If Lcn:SendMessage("OP") { //当前车（右车）向左车传递一个字符串"OP"
        Print "Order Sent". //如果消息发送成功就咬一声
    }.
}.

```

(4) 左车程序

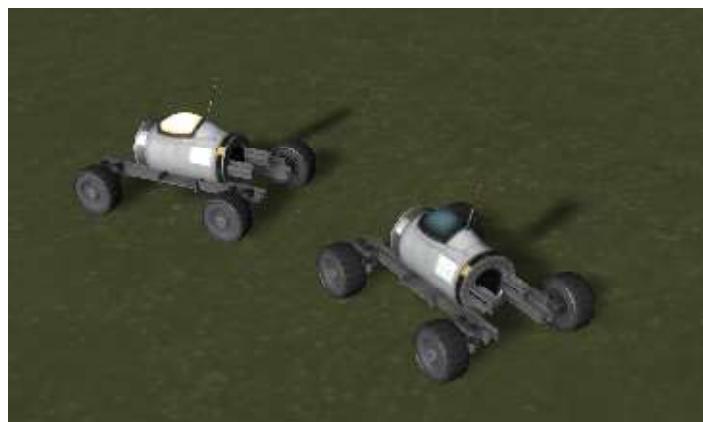
```

Set Lmsgs to Ship:Messages. //取当前车（左车）上的消息收件箱Message Queue
If Lmsgs:Empty = True {
    Print "No Message". //如果收件箱是空的，就报错
} else {
    Set msg to Lmsgs:Pop(). //取收件箱里最早的信息Message
    If msg:Content = "OP" { //如果是OP字符串，就点亮灯光
        Lights on.
    } else {
        Print "Order Error". //如果不是"OP"字符串，就报错
    }
}.

```

(5) 运行结果

我们先运行右车程序，给左车发送”OP”指令，然后再运行左车程序，检查收件箱，如果有”OP”指令，那么左车亮灯。结果如下



8.10

8.10 资源转移

===== 点击以返回目录 =====



(1) 资源转移函数

资源转移函数可以创建 资源转移结构体ResourceTransfer。

之后，玩家可以通过操作资源转移结构体，来实施控制资源转移进度。

资源转移：两个函数，负责转移部分和全部资源
Set transferFoo TO Transfer (资源名称,施放,受方,数量)
Set transferBar TO TransferAll(资源名称,施放,受方)

本小节只介绍如何启动资源转移，有关 资源转移结构体ResourceTransfer，请参考 **资源转移 ResourceTransfer [结构体]**。

(2) 施方与受方

施放和受方可以为以下三种之一：

1. 单个部件 Part
2. 部件的列表 List of Part
3. 单个对接元素 Element

(3) 实际例子

用法代码示例：

```
//将一个对接元素中的氧化剂转移到另一个对接元素中
LIST ELEMENTS IN elist.
Set foo TO TRANSFERALL("OXIDIZER", elist[0], elist[1]).
Set foo:ACTIVE to TRUE.
```

9.

9. 结构体

9.1

9.1 底层结构体

9.1.1

9.1.1 结构体 Structure [结构体]

----- 点击以返回目录 -----



(1) 概述

kOS 中所有变量本质上都是结构体，都是由 最底层的 Structure结构体 派生而来。

(2) 结构体成员

结构体成员	成员类型	读写性	说明
<code>ToString</code>	<code>String</code>	只读	执行 <code>Print</code> 指令时，屏幕上显示的文本
<code>HasSuffix (name)</code>	<code>Boolean</code>	函数	测试该结构体是否有指定后缀的值 其中 <code>name</code> 为 <code>String</code> 类型
<code>SuffixNames</code>	<code>String</code> 列表	只读	该结构体所有后缀的名称列表
<code>IsSerializable</code>	<code>Boolean</code>	只读	该结构体是否是可序列化的类型
<code>TypeName</code>	<code>String</code>	只读	该结构体的类型名称
<code>IsType (name)</code>	<code>Boolean</code>	函数	测试该结构体是否是从指定类型派生出的 或者是否就是这个指定类型本身 其中 <code>name</code> 为 <code>String</code> 类型
<code>Inheritance</code>	<code>String</code>	只读	描述该结构体的继承关系
.....			

(3) 实际例子

kOS 中的所有变量都是由 Structure 结构体 派生而来。

A 派生出 B 意味着 B 也拥有 A 的成员，可以用 A 的后缀。

```
print MuN:typeName().
Body // <--- system prints this

print ("hello"):typeName().
String // <--- system prints this

print (12345.678):typeName().
Scalar // <--- system prints this
```

SuffixNames 后缀的例子：

```
set v1 to V(12,41,0.1). // v1 is a vector
print v1:suffixnames.
List of 14 items:
[0] = DIRECTION
[1] = HASSUFFIX
[2] = ISSERIALIZABLE
[3] = ISTYPE
[4] = MAG
[5] = NORMALIZED
[6] = SQRMAGNITUDE
[7] = SUFFIXNAMES
[8] = TOSTRING
[9] = TYPENAME
[10] = VEC
[11] = X
[12] = Y
[13] = Z
```

Inheritance 后缀的例子：

```
set x to SHIP.
print x:inheritance.
Vessel derived from Orbitable derived from Structure
```

9.2

9.2 集合

9.2.1

9.2.1 枚举 Enumerable [结构体]

===== 点击以返回目录 =====



(1) 概述

枚举Enumerable 是集合类结构体的底层结构，玩家在 kOS 中不会遇到此结构体，但是其他集合类结构体都派生自此。

(2) 结构体成员

Enumerable 枚举结构体 成员列表：

结构体成员	成员类型	读写性	说明
Iterator	Iterator	只读	对应的序列的迭代器
ReverseIterator	Iterator	只读	对应的反向序列的迭代器
Length	Scalar	只读	结构里的成员数量
Contains (item)	Boolean	只读	检查该变量是否包含 item 项目
Empty	Boolean	只读	检查该变量是否内容为空
Dump	String	只读	将该变量内容全转换成字符串
.....			

*注1：迭代器结构体 Iterator 类似数组指针，是所有集合类结构体 共有的特性。

9.2.2

9.2.2 迭代器 Iterator [结构体]

===== 点击以返回目录 =====



(1) 概述

迭代器结构体 Iterator 类似数组指针，是所有集合类结构体 自带的成员。

(2) 用法示例

```
// Starting with a list that was built like this
Set myList To LIST( "Hello", "Aloha", "Bonjour").

// It could be looped over like this
Set MyCurrent TO myList:ITERATOR.
MyCurrent:RESet().
PRINT "After reSet, position = " + MyCurrent:INDEX.
UNTIL NOT MyCurrent:NEXT {
    PRINT "Item at position " + MyIter:INDEX + " is [" + MyIter:VALUE + "]".
}
```

上述代码会显示如下结果：

```
before the first NEXT, position = -1.
Item at position 0 is [Hello].
Item at position 1 is [Aloha].
Item at position 2 is [Bonjour].
```

(3) 结构体成员

Iterator 迭代器结构体 成员列表：

结构体成员	成员类型	读写性	说明
Next ()	Boolean	函数	移动到下一个集合元素，返回的值代表是否成功
Atend	Boolean	只读	是否当前位置是集合尾部位置
Index	Scalar	只读	当前的列表下标 (从0开始计)
Value	不定	只读	当前列表元素的值
.....			

*注2：迭代器结构体 **Iterator** 随集合类结构体一同创建，创建时默认在集合的 -1 位置。当玩家第一次调用 **Iterator:Next()** 之后，迭代器结构体 **Iterator** 才会跑到0。

9.2.3

9.2.3 列表 List [结构体]

===== 点击以返回目录 =====



详见 7.3 列表 List [结构体]。

9.2.4

9.2.4 范围 Range [结构体]

===== 点击以返回目录 =====



(1) 概述

范围结构其实是有固定步距的连续数值序列。

(2) 创建结构体

```
RANGE (START, STOP, STEP). // 一段有步距的实数数列
RANGE (START, STOP). // 一段步距为1的实数数列
RANGE (STOP). // 从1至stop的自然数列
```

(3) 用法示例

```
FOR I IN RANGE(5) {
    PRINT I.
}
// will print numbers 0,1,2,3,4
FOR I IN RANGE(2, 5) {
    PRINT I*I.
}
// will print 4, 9 and 16
```

(4) 结构体成员

该结构体派生自 枚举Enumerable 结构体。

玩家可以在 枚举 Enumerable [结构体] 查看 枚举Enumerable 结构体的详细信息。

该结构体是可序列化（Serializable）的结构体。

Range范围结构体 成员列表：

结构体成员	成员类型	读写性	说明
Enumerable结构的成员			派生自此类型，拥有其全部成员
Start	Scalar	只读	起始数值
Stop	Scalar	只读	终点数值
Step	Scalar	只读	布距，步长
.....			

*注1：可序列化是指，

该类型变量能通过 WriteJSON函数 和 ReadJSON函数 被读写至 json 文件，
指该类型变量能用于僚机之间的通信数据互传。

9.2.5

9.2.5 序列 Queue [结构体]

===== 点击以返回目录 =====



(1) 概述

序列是一种遵照“先进先出”原则的寄存机构。

(2) 用法示例

```
SET Q TO QUEUE(). // 创建一个空的序列
Q:PUSH("alice").
Q:PUSH("bob").
PRINT Q:POP. // will print 'alice'
PRINT Q:POP. // will print 'bob'
```

(3) 结构体成员

该结构体派生自 枚举Enumerable 结构体。

玩家可以在 枚举 Enumerable [结构体] 查看 枚举Enumerable 结构体的详细信息。

该结构体是可序列化（Serializable）的结构体。

Queue序列结构体 成员列表：

结构体成员	成员类型	读写性	说明
Enumerable结构的成员			派生自此类型，拥有其全部成员
Push (item)	无返回值	函数	将item数据加入序列
Pop ()	不定	函数	导出并移除序列最前的数据
Peek ()	不定	函数	导出但不移除序列最前的数据
Clear ()	无返回值	函数	清空序列
Copy	Queue	只读	该序列的副本
.....			

*注1：可序列化是指，

该类型变量能通过 WriteJSON 函数 和 ReadJSON 函数 被读写至 json 文件，
指该类型变量能用于僚机之间的通信数据互传。

9.2.6

9.2.6 堆栈 Stack [结构体]

===== 点击以返回目录 =====



(1) 概述

堆栈是一种遵照“先进后出”原则的寄存机构。

(2) 用法示例

```
SET S TO STACK(). // 创建一个空的堆栈
S:PUSH("alice").
S:PUSH("bob").
PRINT S:POP. // will print 'bob'
PRINT S:POP. // will print 'alice'
```

(3) 结构体成员

该结构体派生自 枚举Enumerable 结构体。

玩家可以在 枚举 Enumerable [结构体] 查看 枚举Enumerable 结构体的详细信息。

该结构体是可序列化 (Serializable) 的结构体。

Stack堆栈结构体 成员列表:

结构体成员	成员类型	读写性	说明
Enumerable结构的成员			派生自此类型，拥有其全部成员
Push (item)	无返回值	函数	将item数据加入堆栈
Pop ()	不定	函数	导出并移除序列最前的数据
Peek ()	不定	函数	导出但不移除序列最前的数据
Clear ()	无返回值	函数	清空序列
Copy	Queue	只读	该序列的副本
.....			

*注1: 可序列化是指,

该类型变量能通过 WriteJSON 函数 和 ReadJSON 函数 被读写至 json 文件,
指该类型变量能用于僚机之间的通信数据互传。

9.2.7

9.2.7 唯一集 UniqueSet [结构体]

===== 点击以返回目录 =====



(1) 概述

唯一集是一种集合类结构体，与 列表List 不同，
唯一集的成员不讲位置顺序，也禁止有重复的项。
唯一集可以用来查重复。

(2) 创用法示例

```
SET S TO UNIQUESET(1,2,3).
PRINT S:LENGTH. // will print 3
S:ADD(1). // 1 was already in the set so nothing happens
PRINT S:LENGTH. // will print 3 again
```

(3) 结构体成员

该结构体派生自 枚举Enumerable 结构体。

玩家可以在 枚举 Enumerable [结构体] 查看 枚举Enumerable 结构体的详细信息。

该结构体是可序列化 (Serializable) 的结构体。

UniqueSet唯一集结构体 成员列表:

结构体成员	成员类型	读写性	说明
Enumerable 结构的成员			派生自此类型，拥有其全部成员
Add (item)	无返回值	函数	在末尾追加项目 item
Remove (item)	无返回值	函数	移除 item 项目
Clear ()	无返回值	函数	清空唯一集
Copy	UniqueSet	只读	当前唯一集的副本
.....			

*注1：可序列化是指，
该类型变量能通过 WriteJSON 函数 和 ReadJSON 函数 被读写至 json 文件，
指该类型变量能用于僚机之间的通信数据互传。

9.2.8

9.2.8 词典 Lexicon [结构体]

===== 点击以返回目录 =====



(1) 概述

词典结构体是一种关联数组。是一种有特殊索引方式的多行二列数组。

(2) 创建结构体

以下两种创建方式效果相同：

```
set mylexicon to lexicon(). //创建空的词典结构体
mylexicon["key1"] = "value1". //为词典结构体增加内容
mylexicon["key2"] = "value2". //为词典结构体增加内容

//创建带有内容的词典结构体,
set mylexicon to lexicon("key1", "value1", "key2", "value2").
```

*注1：词典结构体成员的 key 值 和 value 值 可以是任何类型。

(3) 结构体成员

该结构体派生自 枚举 Enumerable 结构体。

玩家可以在 枚举 Enumerable [结构体] 查看 枚举 Enumerable 结构体的详细信息。

该结构体是可序列化 (Serializable) 的结构体。

Lexicon 词典结构体 成员列表：

结构体成员	成员类型	读写性	说明
Add (key, value)	无返回值		为词典结构添加一个成员
CaseSensitive	Boolean	读写	是否大小写敏感, 初始值为False
Case	Boolean	读写	同上
Clear ()	无返回值		清空结构中的键值
Copy ()	Lexicon	只读	该结构的副本, 用于结构复制
Dump	String	只读	将该结构的所有内容字符串导出
HasKey (key)	Boolean		该结构中是否包含对应的 key 值
HasValue (value)	Boolean		该结构中是否包含对应的 value 值
Keys	List	只读	该结构中的字符串的列表
Values	List	只读	该结构中的值的列表
Length	Scalar	只读	该结构的键值数量
Remove (key)	无返回值		移除结构中的指定键值
HasSuffix (name)	Boolean		是否有指定的后缀 (包含成员和Key值)
SuffixNames	List	只读	该结构体的后缀的字符串列表 (包含成员和Key值)
.....

*注1: 可序列化是指,
该类型变量能通过 WriteJSON 函数 和 ReadJSON 函数 被读写至 json 文件,
指该类型变量能用于僚机之间的通信数据互传。

(4) 访问词典成员

三种方法:

```
Lexicon[A]

For B in Lexicon:Keys {
    ....
}

Lexicon:Key
```

*注1: 第三种方法允许以冒号后缀形式调用对应key的value值。
此时key值应符合变量标识符的命名规则。
且key值不应与结构体原有成员名重复。 (如果重复则优先导出结构体原有成员)

(5) 隐式添加词典成员

```
s:add ("s2",10).
set s["s2"] to 10.//这两句话一个意思
```

(6) 当重复添加项时会报错

```
SET ARR TO LEXICON().
ARR:ADD ("somekey",100).
ARR:ADD ("somekey",200). //这句报错，因为somekey已存在

SET ARR TO LEXICON().
SET ARR["somekey"] to 100.
SET ARR["somekey"] to 200. //不报错，只是更新了somekey的值
```

(7) 实际例子

```
set s to LEXICON (). //创建空的词典结构体
set s1 to LEXICON () .
set s2 to LEXICON () .
s2:add ("T",True).
s2:add ("F",False).
print s2["T"]. //应该输出True
print s2:T. //应该输出True
print s2["F"]. //应该输出False
print s2:F. //应该输出False
s:add ("s1",s1).
s:add ("s2",s2).
s["s1"]:add ("a",3).
Print s1["a"]. //应该输出3
Print s1:a. //应该输出3
```

9.3

9.3 盘符与文件

9.3.1

9.3.1 盘符 Volume [结构体]

===== 点击以返回目录 =====



(1) 概述

盘符Volume (又叫卷)，表示 kOS处理器 或者 KSC的数据中心Archive 对应的存储空间。

(2) 结构体成员

Volume 盘符结构体 成员列表：

结构体成员	成员类型	读写性	说明
FreeSpace	Scalar (Byte)	只读	剩余存储空间
Capacity	Scalar (Byte)	只读	总存储空间
Name	String	读写	空间名称
Renameable	Boolean	只读	空间名称是否可修改
Root	VolumeDirectory	只读	盘符的根目录
Files	Lexicon	只读	空间上所有文件和文件夹的索引
PowerRequirement	Scalar	只读	该盘符的电力消耗 (由kOS部件而定)
Exists (path)	Boolean	函数	查询文件或文件夹是否存在
Create (path)	VolumeFile	函数	创建文件
CreateDir (path)	VolumeDirectory	函数	创建文件夹
Open (path)	VolumeFile or Boolean	函数	打开文件或文件夹, 如果失败则返回 False
Delete (path)	Boolean	函数	删除文件或文件夹
.....			

9.3.2

9.3.2 路径 Path [结构体]

===== 点击以返回目录 =====



(1) 概述

文件或文件夹路径。

(2) 结构体成员

Path路径结构体 成员列表:

结构体成员	成员类型	读写性	说明
Volume	Volume	只读	该路径所属的盘符
Segments	String 列表	只读	路径的分段
Length	Scalar	只读	路径分段的数量
Name	String	只读	路径所指的文件 / 文件夹名
HasExtension	Boolean	只读	路径是否包含后缀
Extension	String	只读	路径的后缀
Root	Path	只读	该路径所属盘符的根目录
Parent	Path	只读	该路径的上一级路径 对根目录使用此值会报错
ChangeName (name)	Path	函数	返回新的路径，区别是将原路径的最后一段改为了 name
ChangeExtension (name)	Path	函数	返回新的路径，区别是将原路径最后一段的后缀 改为了 name
IsParent (path)	Boolean	函数	检查 path 是不是该路径的 上级路径
Combine (name1, [name2, ...])	Path	函数	返回新的路径，新路径在该 路径后新增了子文件夹 / 文件
.....			

*注1：举例说明路径的分段：0:/directory/subdirectory/script.ks 包含了以下分段：
directory, subdirectory, script.ks

(3) 例子

Combine函数的例子：

```
set p to path("0:/home").
set p2 to p:combine("d1", "d2", "file.ks").
print p2

0:/home/d1/d2/file.ks
```

现实世界中 Windows 电脑上的一个完整路径也许是：

D:\Program Files (x86)\Steam\Steam.exe

现实世界中 Linux 电脑上的一个完整路径也许是：

/home/user/somefile

kOS 里的完整路径类似 Linux 系统的写法：

0:/acdc/new.txt

kOS 中的路径都是字符串，
kOS 中存在两种路径：绝对路径 和 相对路径。

(4) 绝对路径

```
"0:/lib/launch/base.ks" //文件 base.ks
"0:/lib/launch/base" //文件 base

"0:/lib/../base.ks" //这一句和下面一句是一个意思，两点..表示返回上一级子目录
"0:/base.ks"
"0:.." //这句非法，卷的顶层已经没有上一级了
```

在路径中类似 ./ 这样的写法，表示文件向上一级。

(5) 当前路径和相对路径

相对路径要有一个起始文件夹，CD函数就是用来设定这个其实文件夹的

```
CD("0:/acdc"). //将当前文件夹路径设为 0:/acdc ,如果该文件不存在则创建文件夹
Print Path(). //显示当前文件夹路径
```

当前路径默认为当前卷Volume 的根目录。

以设定的当前路径作为起始文件夹开始算，指代的路径

```
CD("0:/acdc"). //将当前文件夹路径设为 0:/acdc
"file.ks" //代表的是 0:/acdc/file.ks
"../file.ks" //和上面一样
"/top.ks" //注意这个和上面不一样， /代表从卷开始算的，这个代表的文件是 0:/top.ks
```

(6) 与路径有关的预设函数

((1)) 当前路径 Path()

Path函数的返回值是当前路径。

```
Print Path().
```

((2)) 设定当前路径 CD(pathString)

CD函数用于设定当前路径。

```
CD("0:/scripts").
```

((3)) 复制路径 CopyPath(FromPath,ToPath)

CopyPath函数用于复制路径。

```
//把上一级文件夹里 launch.ks 文件夹复制到当前路径
CopyPath("../launch.ks", "").
```

例子：

```
COPYPATH(myfilename, "1:"). // This is an example of a bareword filename.
COPYPATH("myfilename", "1:"). // This is an example of an EXPRESSION filename.
COPYPATH(myfilename.ks, "1:"). // This is an example of a bareword filename.
COPYPATH(myfilename.txt, "1:"). // This is an example of a bareword filename.
COPYPATH("myfilename.ks", "1:"). // This is an example of an EXPRESSION filename
SET str TO "myfile" + "name" + ".ks".
COPYPATH(str, "1:"). // This is an example of an EXPRESSION filename
COPYPATH("myfile" + "name" + ".ks", "1:"). // This is an example of an EXPRESSION filename
```

((4)) 移动路径 MovePath(FromPath,ToPath)

MovePath函数 用于移动路径。

```
//把上一级文件夹里 launch.ks 文件夹移动到当前路径
CopyPath("../launch.ks", "").
```

((5)) 删除路径 DeletePath(Path)

DeletePath函数 用于删除路径。

```
//删除上一级文件夹里 launch.ks 文件夹
CopyPath("../launch.ks").
```

((6)) 检查路径是否存在 Exists(Path)

Exists函数 用于检查路径是否存在，返回的是逻辑值。

```
If Exists ("boot/abc.txt") {
    Print "OK".
}
```

((7)) 创建文件路径 Create(Path)

Create函数 用于创建路径（文件）。

```
Create ("boot/abc.txt").
```

((8)) 创建文件夹路径 CreateDir(Path)

CreateDir函数 用于创建路径（文件夹）。

```
CreateDir ("boot/acdc").
```

((9)) 打开路径 Open(Path)

Open函数 用于打开文件/文件夹，其返回值是 VolumeFile 或 VolumeDirectory 类型。

之后可以对文件内容进行读写，对文件夹信息进行获取。

```

Set file1 to Open("boot/abc.txt").
Print file1:ReadAll.      //显示这个文件的内容（仅文本文件有效）
file1:Write("www").      //在这个文件末尾加上 "www"
file1:WriteLn("qqq").     //在这个文件末尾加上 "qqq" 并回车

Set dir1 to CreateDir("boot/acdc").
Print dir1>List.         //显示这个文件夹下的文件/文件夹列表

```

((10)) 查询程序文件的路径 ScriptPath(pathString)

ScriptPath函数 的返回值是当前程序文件所在的路径。

```
Print ScriptPath().
```

9.3.3

9.3.3 盘符项目 VolumeItem [结构体]

===== 点击以返回目录 =====



(1) 概述

盘符项目VolumeItem 包含了盘符下的文件和文件夹。

(2) 结构体成员

VolumeItem 盘符列表结构体 成员列表:

结构体成员	成员类型	读写性	说明
Name	String	只读	盘符项目的名称，含或追名
Extension	String	只读	盘符项目的后缀名
Size	Scalar (Byte)	只读	文件的大小，如果是文件夹则大小为0
IsFile	Boolean	只读	是文件则返回 True，是文件夹则返回 False
.....		 [真] [假]

9.3.4

9.3.4 盘符路径 VolumeDirectory [结构体]

===== 点击以返回目录 =====



(1) 概述

盘符路径VolumeDirectory 表示盘符下的文件夹。

(2) 结构体成员

该结构体派生自 盘符项目 VolumeItem 结构体。

玩家可以在 盘符项目 VolumeItem [结构体] 查看 盘符项目 VolumeItem 结构体的详细信息。

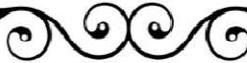
VolumeDirectory 盘符路径结构体 成员列表：

结构体成员	成员类型	读写性	说明
VolumeItem 结构的成员			派生自此类型，拥有其全部成员
List	VolumeFile 与 VolumeDirectory 的列表	函数	所有文件和文件夹的列表
.....			

9.3.5

9.3.5 盘符文件 VolumeFile [结构体]

===== 点击以返回目录 =====



(1) 概述

盘符文件 VolumeFile 表示盘符下的文件。

(2) 结构体成员

该结构体派生自 盘符项目 VolumeItem 结构体。

玩家可以在 盘符项目 VolumeItem [结构体] 查看 盘符项目 VolumeItem 结构体的详细信息。

VolumeFile 盘符文件结构体 成员列表：

结构体成员	成员类型	读写性	说明
VolumeItem 结构的成员			派生自此类型，拥有其全部成员
ReadAll ()	FileContent	函数	全部文件内容
Write (some)	Boolean	函数	写入文件，参数 some 需为 String 或 FileContent 类型
WriteLn (string)	Boolean	函数	写入文件然后换行，参数为 String
Clear ()	无返回值	函数	清空文件内容
.....			

9.3.6

9.3.6 文件内容 FileContent [结构体]

===== 点击以返回目录 =====



(1) 概述

表示文件内容。可以通过 VolumeFile:ReadAll 来获得该种结构体。

(2) 使用示例

复制文件

```
SET CONTENTS TO OPEN("filename") :READALL.
SET NEWFILE TO CREATE("newfile").
NEWFILE:WRITE(CONTENTS).
```

以字符串形式读取文件内容

```
SET CONTENTS_AS_STRING TO OPEN("filename") :READALL:STRING.
// do something with a string:
PRINT CONTENTS_AS_STRING:CONTAINS("test").
```

(3) 结构体成员

该结构体是可序列化（Serializable）的结构体。

FileContent文件内容结构体 成员列表：

结构体成员	成员类型	读写性	说明
Length	Scalar (Byte)	只读	文件大小
Empty	Boolean	只读	是否是空文件
Type	String	只读	文件类型
String	String	只读	UTF - 8 编码的文件内容
Iterator	Iterator	只读	文件索引 (按行记数)
.....			

*注1：可序列化是指，

该类型变量能通过 WriteJSON 函数 和 ReadJSON 函数 被读写至 json 文件，
指该类型变量能用于僚机之间的通信数据互传。

*注2：FileContent:Type 有四种返回结果：

1. TooShort 短文件，文件内容很短，无法判别
2. ASCII 包含 ascii 字符的文本文件
3. KSM 编译命令产生的 kOS 可执行文件.ksm
4. Binary 二进制文件

9.4

9.4 载具与部件

9.4.1

9.4.1 载具 Vessel [结构体]

===== 点击以返回目录 =====



(1) 获取结构体

游戏中所有的载具都有对应的 载具Vessel结构体。

玩家可以按照如下方法获取此结构体。

```

Set a to Vessel("Vessel Name"). //根据载具名称获取

Set a to Ship. //获取当前载具的 Vessel 结构体

Set a to Target. //获取目标载具的 Vessel 结构体

```

(2) 结构体成员

该结构体派生自轨道物 Orbital 结构体。

玩家可以在 轨道物 Orbital [结构体] 查看 轨道物 Orbital 结构体的详细信息。

该结构体是可序列化 (Serializable) 的结构体。

Vessel载具结构体 成员列表:

结构体成员	成员类型	读写性	说明
Orbital 结构的成员			派生自此类型，拥有其全部成员
Control	Control	只读	飞行控制
Bearing	Scalar (deg)	只读	该载具对于自机的相对罗盘方向，从自机机头方向顺时针转过的角度
Heading	Scalar (deg)	只读	该载具对于自机的绝对罗盘方向，从正北方顺时针转过的角度
MaxThrust	Scalar (kN)	只读	所有活动引擎的最大推力之和
MaxThrustAt (atm)	Scalar (kN)	函数	不同气压下 MaxThrust 值 的函数
AvailableThrust	Scalar (kN)	只读	所有活动引擎的推力和
AvailableThrustAt (atm)	Scalar (kN)	函数	不同气压下 AvailableThrust 值 的函数
Facing	Direction	只读	载具的前方朝向
Bounds	Bounds	只读	载具的边界框
Mass	Scalar (tons)	只读	载具的质量
WetMass	Scalar (tons)	只读	载具资源装满时的质量
DryMass	Scalar (tons)	只读	载具不算资源的质量
DynamicPressure	Scalar (atm)	只读	载具的空气动压
Q	Scalar (atm)	只读	同上
VerticalSpeed	Scalar (m / s)	只读	垂直速度，向上为正，向下为负
GroundSpeed	Scalar (m / s)	只读	水平速度
AirSpeed	Scalar (m / s)	只读	相对于地表 & 空气的速度
ShipName	String	读写	载具名称
Name	String	读写	同上
Status	String	只读	载具状态
Type	String	读写	载具类型
StartTracking ()	无返回值	函数	让KSC跟踪站把载具当做小行星一样跟踪
AngularMomentum	Vector	只读	Ship - Raw 坐标系下的角动量， $t * m^2 / (s * rad)$
AngularVel	Vector	只读	Ship - Raw 坐标系下的角速度， rad / s
Sensors	VesselSensors	只读	机载传感器数据
Loaded	Boolean	只读	该载具是否被物理引擎加载
UnPacked	Boolean	只读	该载具是否被物理引擎发现
LoadDistance	LoadDistance	只读	该载具的载入距离设置

IsDead	Boolean	只读	该载具尚否存在
Patches	Orbit 列表	只读	该载具飞行的轨道列表
RootPart	Part	只读	该载具的根部件
ControlPart	Part	只读	该载具的主控部件 (ControlFromHere 的部件)
Parts	Part 列表	只读	该载具上所有部件的列表
DockingPorts	DockingPort 列表	只读	该载具上所有对接口部件的列表
Elements	Element 列表	只读	该载具上所有对接单元的列表
Resources	AggrgateResource 列表	只读	该载具上所有资源概况的列表
PartsNamed (name)	Part 列表	函数	返回载具上符合 name 名称的部件列表
PartsNamedPattern (namepattern)	Part 列表	函数	同上, 区别是参数允许正则表达式
PartsTitled (title)	Part 列表	函数	返回载具上符合 title 标题的部件列表
PartsTitledPattern (titlepattern)	Part 列表	函数	同上, 区别是参数允许正则表达式
PartsTagged (tag)	Part 列表	函数	返回载具上符合 tag 标签名的部件列表
PartsTaggedPattern (tagpattern)	Part 列表	函数	同上, 区别是参数允许正则表达式
PartsDubbed (name)	Part 列表	函数	返回载具上符合 name 名称、 标题、标签的部件列表
PartsDubbedPattern (namepattern)	Part 列表	函数	同上, 区别是参数允许正则表达式
ModulesNamed (name)	PartModule 列表	函数	返回该载具上 name 名称的部件模组列表
PartsInGroup (groupNum)	Part 列表	函数	返回该载具上和给定动作组关联的部件 列表
ModulesInGroup (groupNum)	PartModule 列表	函数	返回该载具上和给定动作组关联的部件 模组列表
AllPartsTagged ()	Part 列表	函数	返回该载具上所有 tag 标签名的部件列表
CrewCapacity	Scalar	只读	乘员容量
Crew ()	CrewMember 列表	函数	乘员列表
Connection	Connection	只读	该载具的通讯连接
Messages	MessageQueue	只读	该载具的通讯的报文队列
.....			

*注1: 可序列化是指,

该类型变量能通过 WriteJSON 函数 和 ReadJSON 函数 被读写至 json 文件,
指该类型变量能用于僚机之间的通信数据互传。

*注2: :STATUS 的值可以是: “LANDED”、“SPLASHED”、“PRELAUNCH”、“FLYING”
、“SUB_OrbitAL”、“OrbitING”、“ESCAPING”、“DOCKED”

*注3: :TYPE 的值可以是: Base”、“Station”、“Ship”、“Lander”、“Rover”、“Probe”、“Debris”

(3) 实际例子

```
// Get a vessel by it's name.
// The name is Case Sensitive.
SET MY_VESS TO VESSEL("Some Ship Name").
// Save the current vessel in a variable,
// in case the current vessel changes.
SET MY_VESS TO SHIP.
// Save the target vessel in a variable,
// in case the target vessel changes.
SET MY_VESS TO TARGET.
```

```
print ship:Mass. //显示当前飞船的总质量
print ship:AirSpeed. //显示当前飞船的相对于大气的速度
print ship:Facing. //显示当前飞船的朝向，和所选的控制部件有关
print ship>Type. //显示当前飞船的标记类型
```

9.4.2

9.4.2 乘员 CrewMember [结构体]

===== 点击以返回目录 =====



(1) 概述

乘员CrewMember 对应载具中的某一个成员。
玩家可以从 Vessel:Crew() 获得载具上的成员列表。

(2) 结构体成员

CrewMember乘员结构体 成员列表:

结构体成员	成员类型	读写性	说明
Name	string	只读	乘员姓名
Gender	string	只读	性别, Male或Female
Experience	Scalar	只读	星级
Trait	string	只读	职业, 可以是 Pilot, Engineer or Scientist
Tourist	Boolean	只读	是不是旅客 (旅客无法进行驾驶)
Part	Part	只读	所处部件
.....			

9.4.3

9.4.3 载具传感器 VesselSensors [结构体]

===== 点击以返回目录 =====



(1) 概述

载具传感器 VesselSensors 综合了载具上传感器部件的读数。
玩家需要先在载具上安装传感器部件，然后才可以读取对应的读数。

(2) 结构体成员

VesselSensors 载具传感器结构体 成员列表：

结构体成员	成员类型	读写性	说明
Acc	Vector (m / s^2)	只读	加速度传感器
Pres	Scalar (kPa)	只读	气压传感器
Temp	Scalar (°C)	只读	温度传感器
Grav	Vector (m / s^2)	只读	重力加速度传感器
Light	Scalar	只读	太阳能板的光强传感器
.....			

9.4.4

9.4.4 资源总量 AggregateResource [结构体]

===== 点击以返回目录 =====



(1) 概述

资源总量 AggregateResource 对应当前载具上每种资源的总量。

(2) 获取结构体

玩家有两种方式可以获取 资源总量 AggregateResource

```
ST RESOURCES IN res. //res 是当前载具上 资源总量AggregateResource 的列表
```

```
//ses 是当前载具上某一个 Stage (发射序列) 可用的所有资源的 资源总量列表
Set ses to Stage:Resources.
```

(3) 结构体成员

List 列表结构体 成员列表：

结构体成员	成员类型	读写性	说明
Name	String	只读	资源名称
Amount	Scalar (unit)	只读	资源总量
Capacity	Scalar (unit)	只读	资源最大容量
Parts	Part列表	只读	含有此资源的部件列表
.....			

(4) 实际例子

例子1：

```

PRINT "THESE ARE ALL THE RESOURCES ON THE SHIP:".
LIST RESOURCES IN RESLIST.
FOR RES IN RESLIST {
    PRINT "Resource " + RES:NAME.
    PRINT " value = " + RES:AMOUNT.
    PRINT " which is "
    + ROUND(100*RES:AMOUNT/RES:CAPACITY)
    + "% full.".
}.

```

例子2:

```

PRINT "THESE ARE ALL THE RESOURCES active in this stage:".
SET RESLIST TO STAGE:RESOURCES.
FOR RES IN RESLIST {
    PRINT "Resource " + RES:NAME.
    PRINT " value = " + RES:AMOUNT.
    PRINT " which is "
    + ROUND(100*RES:AMOUNT/RES:CAPACITY)
    + "% full.".
}.

```

9.4.5

9.4.5 对接单元 Element [结构体]

===== 点击以返回目录 =====



(1) 概述

对接单元 Element 指的是载具上某些部分的部件集合，
他们现在已经通过对接成为载具的一部分，
但是原先他们是一个独立的载具。

*注1: 一定要有过对接行为才能算作对接单元。
在车间里两个用对接口的部分不算，
一定要发射之后，分离对接口再重新对接回去的才算。

(2) 获取对接单元

kOS 提供了有关对接单元的预设结构体，玩家可以自己从里面找。

List elements in el. //el是当前载具上所有 对接单元 的列表

*注2: 没有过对接行为的话，预设列表 Elements 就会是空集。

(3) 结构体成员

Element对接单元结构体 成员列表:

结构体成员	成员类型	读写性	说明
Name	String	读写	已对接载具的名字
Uid	String	只读	UID编号
Parts	Part列表	只读	该对接单元的部件列表
DockingPorts	DockingPort列表	只读	该对接单元的对接口列表
Vessel	Vessel	只读	该对接单元所属载具
Resources	AggregateResources列表	只读	该对接单元的资源概况的列表
.....

9.4.6

9.4.6 部件 Part [结构体]

===== 点击以返回目录 =====



(1) 获取结构体

玩家可以使用列表List（指令），或者查询部件信息，来获取部件结构体Part。

(2) 结构体成员

Part部件结构体 成员列表：

结构体成员	成员类型	读写性	说明
Name	String	只读	部件的名称
Title	String	只读	部件的标题名
Mass	Scalar (tons)	只读	部件的质量
DryMass	Scalar (tons)	只读	部件干重, 空资源质量
WetMass	Scalar (tons)	只读	部件适重, 满资源质量
Tag	String	读写	部件的标签名, 可自由设定
ControlFrom ()	无返回值	函数	从此控制。仅对当前载具有效
Stage	Scalar	只读	部件的命令序列
CID	String	只读	所在载具的CID号, 类似句柄号
UID	String	只读	部件的UID号, 类似句柄号
Rotation	Direction	只读	部件的x坐标轴的朝向
Position	Vector (m)	只读	部件的位置矢量
Facing	Direction	只读	部件的前方朝向
Bounds	Bounds	只读	部件的边界框
Resources	Resource列表	只读	部件的资源列表
Targetable	Boolean	只读	部件是否可被设为目标
Ship	Vessel	只读	部件的所在飞船
GetModule (name)	PartModule	函数	部件上名称为name的部件模组
GetModuleByIndex (n)	PartModule	函数	部件上第n个部件模组
Modules	String列表	只读	部件上的部件模组列表
AllModules	String列表	只读	同上
Parent	Part	只读	部件在载具上的父部件
HasParent	Boolean	只读	是否有父部件
Decoupler	Decoupler列表	只读	分离器列表
Separator	Decoupler列表	只读	同上
DecoupledIn	Scalar	只读	部件在载具的第几段Stage 为 -1 则载具不可分离
SeparatedIn	Scalar	只读	同上
HasPhysics	Boolean	只读	部件是否参加物理演算
Children	Part列表	只读	部件在载具上的子部件列表
.....			

(3) 实际例子

例子：将 tag 为 www 的飞船“从此操控”。

```
Set a to Ship:PartsTagged("www").
print a[0]:CONTROLFROM ().
```

顺便再举个例子说明一下，kOS内置PSRTS列表里的是所有物理引擎完整载入的部件名称，也就是说不光当前载具的部件，周围的其它载具的部件也会记录在内。

```

LIST PARTS FROM TARGET IN tParts.
PRINT "The target vessel has a".
PRINT "partcount of " + tParts:LENGTH.
SET totTargetable to 0.
FOR part in tParts {
IF part:TARGETABLE {
SET totTargetable TO totTargetable + 1.
}
}
PRINT "...and " + totTargetable.
PRINT " of them are targetable parts.".

```

9.4.7

9.4.7 边界 Bounds [结构体]

===== 点击以返回目录 =====



(1) 概述

边界 **Bounds** 为一个可显示的盒形边界范围，通常用于载具（**Vessel**）或部件（**Part**）的盒形边界范围。

玩家可以通过 **Vessel:Bounds** 或 **Part:Bounds** 获取该结构体，也可以通过 **Bounds** 函数自行创建。**Vessel:Bounds** 或 **Part:Bounds** 将 **Vessel/Part** 包起，从而有余量地标记碰撞体积。

*注1：边界 **Bounds** 结构体有助于玩家判断载具底端的离地距离，有助于判断相近的两载具是否即将碰撞。

*注2：一些事件会导致部件的边界**Bounds** 发生变化。例如操作转轴、展开太阳能板等。

一些事件会导致载具的边界**Bounds** 发生变化。例如部件爆炸、对接和分离等。

一些事件会导致边界**Bounds** 突然跳转朝向。例如为载具选择新朝向基准、进入舱内视角。

*注3：边界 **Bound** 结构体非常耗计算资源，没事少用。而且要掌握计算量小的用法。

(2) 创建结构体

```

local my_bounds is Bounds (AbsOrigin, Facing, RelMin, RelMax).
// 参数含义详见后文 结构体 成员列表 的说明。

```

例子：

```

// Makes a bounds that is centered around a flag,
// oriented in that flag's UP direction, which
// goes a lot further up into the sky than it does down
// into the ground (to demonstrate that the bounds box
// doesn't have to span equally far in all directions
// around the origin, and thus why the origin isn't always
// the center of the box):
local my_flag is vessel("that flag").
local my_bounds is BOUNDS(
    my_flag:position,
    my_flag:up, // In this facing, Z = up/down, X = north/south, and Y = east/west.

    // box is 20x20x502 meters, centered in east/west/north/south terms, but
    // extending higher up in the +Z direction than down in the -Z direction:
    V(-10, -10, -2),
    V(10, 10, 500)
).

```

(3) 结构体成员

Bounds 边界结构体 成员列表:

结构体成员	成员类型	读写性	说明
AbsOrigin	Vector	读写	边界盒的原点 (Ship - raw坐标系)
Facing	Direction	读写	边界盒自身坐标系的朝向 (同Part / Vessel自身的Facing)
RelMin	Vector	读写	边界盒想, 顶点的坐标 (自身坐标系) 该顶点选取规律为: x + y + z最小
RelMax	Vector	读写	RelMin对角顶点的坐标 (自身坐标系)
AbsMin	Vector	只读	边界盒顶点的坐标 (Ship - raw坐标系)
AbsMax	Vector	只读	AbsMin对角顶点的坐标 (自身坐标系)
RelCenter	Vector	只读	边界盒中心点的坐标 (自身坐标系)
AbsCenter	Vector	只读	边界盒中心点的坐标 (Ship - raw坐标系)
Extents	Vector	读写	从边界盒中心到最远顶点的向量 (自身坐标系)
Size	Vector	读写	边界盒的“直径”, 2倍的Extents
FurtherstCorner (Vector ray)	Vector	只读	边界盒最靠近参数方向的顶点坐标 (Ship - raw坐标系)
BottonAlt	Scalar	只读	边界盒海拔最低的顶点坐标
BottonAltRader	Scalar	只读	边界盒离地高度最低的顶点坐标
RelOrigin is missing	NA		V (0, 0, 0)
.....			

*注1: Bounds 边界结构体成员的写入只在“结构体是通过 Bounds函数创建的”的情况下有实际意义。

*注2: Part结构体的Bounds:AbsOrigin 坐标同 Part:POSITION。

Vessel结构体的Bounds:AbsOrigin 坐标同 Vessel:PARTS[0]:POSITION , 载具根部件的位置。

*注3: Bounds:RelMin 和 Bounds:RelMax 可通过如下方式转换为 Ship-raw坐标系

MyBounds:FACING * MyBounds:RELMIN , MyBounds:FACING * MyBounds:RELMAX

理论上说, ABSMIN = ABSORIGIN + (FACING * RELMIN), AbsMax值和AbsCenter值同理

9.4.8

9.4.8 部件资源 Resource [结构体]

===== 点击以返回目录 =====



(1) 概述

部件资源 Resource 是 部件Part 的一个成员。

玩家可以通过 Part:Resources 获取。

(2) 结构体成员

Resource部件资源结构体 成员列表:

结构体成员	成员类型	读写性	说明
Name	String	只读	资源名称
Amount	Scalar (unit)	只读	资源数量
Density	Scalar (T / unit)	只读	资源密度
Capacity	Scalar (unit)	只读	资源最大容量
Toggleable	Boolean	只读	资源是否能转移
Enabled	Boolean	只读	资源是否转移中
.....

9.4.9

9.4.9 部件模组 PartModule [结构体]

===== 点击以返回目录 =====



(1) 概述

部件模组 PartModule 对应部件的右键菜单。

玩家可以通过 Part:GetModule 函数来获取部件模组。

```
//假设变量 P 是我们要操作的 部件Part
//在命令窗口显示部件P上所有 PartModule 的 Module名
Print P:Modules. //这一步应该在测试的时候有，实际运行的时候删掉

//例如我们看到第3个是我们要的，那么取第3个 Module 的字符串名字
Set MName to P:Modules[2].
Set PModule to P:GetModule(MName). //PModule 就是我们要的 PartModule 了。
```

(2) 部件模组的三种成员：KSPField、KSPEvent、KSPAction

KSP Field 对应的是，能在部件部件右键菜单里设置的数值，包含以按钮形式显示的逻辑值。

KSPField 是单个字段。

数值和字符串字段直接显示字段值。

布尔字段的 KSPField 以按钮形式显示，按钮按下后不会弹回来而是凹进去。

用法：

```
//假设 ModuleA 是我们要操作的 部件模组PartModule
//在命令窗口显示 部件模组ModuleA 上所有 字段KSPField 的 信息
Print ModuleA:AllFields. //这一步应该在测试的时候有，实际运行的时候删掉

//例如我们看到第3个字段 FName 是我们要操作的
ModuleA:GetField(FName). //返回部件模块moduleA的FName的值
ModuleA:SetField(FName,newFieldB). //将部件模块moduleA的FName的值设为newFieldB
```

KSP Event 对应的是，能在部件部件右键菜单里操作的，部件动作。

KSPEvent 能调用函数（方法），实现一些字段无法实现的功能（例如升起和降下起落架）。

KSPEvent 外形为按钮，按下后会弹回来。

用法：

```
//假设 ModuleA 是我们要操作的 部件模组PartModule
//在命令窗口显示 部件模组ModuleA 上所有 事件KSPEvent 的 信息
Print ModuleA:AllEvents. //这一步应该在测试的时候有，实际运行的时候删掉

//例如我们看到第3个事件 EName 是我们要操作的
ModuleA:DoEvent (EName). 运行部件模块moduleA的EName事件
```

注意，有些右键内容的名称是动态改变的，比如起落架降下和升起，降下后你再叫他降下是没有用的。

只要显示在右键菜单里的，kOS基本动能动。所以kOS能支持纯部件MOD。

KSP Action 对应的是，能在车间的动作组编辑里设置的，部件动作。

KSPAction 和 KSPEvent 类似，不过他是对应车间里可以设置的所有动作组。

例如引擎通常可以在车间里设置 toggle engine (开/关引擎) 的动作组，设置好之后，按对应数字键就可以触发动作。而 kOS 的 KSPAction，不需要设置动作组，就能在程序里触发这个行为。

用法：

```
//假设 ModuleA 是我们要操作的 部件模组PartModule

//在命令窗口显示 部件模组ModuleA 上所有 动作组关联KSPAction 的 信息
Print ModuleA:AllActions. //这一步应该在测试的时候有，实际运行的时候删掉

//例如我们看到第3个动作组关联 AName 是我们要操作的
ModuleA:DoAction (AName,boolean). //用布尔值触发部件模块moduleA中的AName动作组行为
```

注意，有些右键内容的名称是动态改变的，比如起落架降下和升起，降下后你再叫他降下是没有用的。

只要显示在车间动作组里的，kOS基本动能动。所以kOS能支持纯部件MOD。

(3) 结构体成员

PartModule部件模组结构体 成员列表：

结构体成员	成员类型	读写性	说明
Name	String	只读	部件模组的名称
Part	Part	只读	所属部件
AllFields	String列表	只读	所有KSPField名，有格式字符串
AllFieldNames	String列表	只读	所有KSPField名，无格式字符串
AllEvents	String列表	只读	所有KSPEvent名，可直接用于DoEvent函数
AllEventNames	String列表	只读	所有KSPEvent名，不可直接用于DoEvent函数
AllActions	String列表	只读	所有KSPAction名，可直接用于DoAction函数
AllActionnames	String列表	只读	所有KSPAction名，不直接用于DoAction函数
GetField (name)	无返回值	函数	读取KSPField字段成员的值
SetField (name, value)	无返回值	函数	设置KSPField字段成员的值
DoEvent (name)	无返回值	函数	触发KSPEvent事件成员的行为
DoAction (name, bool)	无返回值	函数	触发KSPAction动作组关联操作
HasField (name)	Boolean	函数	部件模组里是否有KSPField
HadEvent (name)	Boolean	函数	部件模组里是否有KSPEvent
HasAction (name)	Boolean	函数	部件模组里是否有KSPAction
.....			

9.4.10

9.4.10 kOS处理器 kOSProcessor [结构体]

===== 点击以返回目录 =====



(1) 概述

kOS处理器 kOSProcessor 是一种特殊的 动作模组PartModule。

专指 kOS部件 里 kOS功能的动作模组。

(2) 获取结构体

玩家可以使用 Processor 函数 来方便的获取 kOS 处理器。

Processor 函数:

用法: Processor (Para)

返回值: 返回 kOSProcessor 类型

参数: Para 可为 volume 类型, 也可为部件的标签名字符串 String

例子:

```
Set p2 to Processor ("part2"). //p1 是kOSProcessor类型
```

(3) 结构体成员

该结构体派生自 动作模组 PartModule 结构体。

玩家可以在 部件模组 PartModule [结构体] 查看 动作模组 PartModule 结构体的详细信息。

kOSProcessor kOS 处理器 结构体 成员列表:

结构体成员	成员类型	读写性	说明
PartModule 结构的成员			派生自此类型, 拥有其全部成员
Mode	string	只读	kOS 处理器的状态
Activate ()	无返回值	函数	启用该 kOS 处理器
Deactivate ()	无返回值	函数	关闭该 kOS 处理器
Tag	string	只读	kOS 处理器的 Tag
Volume	volume	只读	kOS 处理器 本地盘符 Volume 1
BootFileName	string	读写	kOS 处理器中设置的自启动文件名
Connection	Connection	只读	到该 kOS 处理器 的通讯连接
.....			

*注1: :Mode 的值可以是: "OFF" 关闭, "Ready" 启用, "Starved" 没电

9.4.11

9.4.11 内核 Core [结构体]

===== 点击以返回目录 =====



(1) 概述

内核 Core 是一种特殊的 kOS 处理器 kOSProcessor, 专指当前 kOS 程序 所在的 kOS 处理器。对于一个 kOS 程序, 内核只有一个。玩家可以通过预设变量 Core 来使用内核。

(2) 结构体成员

该结构体派生自 kOS 处理器 kOSProcessor 结构体。

玩家可以在 kOS 处理器 kOSProcessor [结构体] 查看 kOS 处理器 kOSProcessor 结构体的详细信息。

Core内核结构体 成员列表:

结构体成员	成员类型	读写性	说明
Core 结构的成员			派生自此类型，拥有其全部成员
Vessel	Vessel	只读	运行当前 kOS 程序 的载具，同 Ship
Element	Element	只读	运行当前 kOS 程序 对接单元
Tag	String	读写	当前 Core 所在 part 的 Tag
Version	VersionInfo	只读	kOS 版本号
CurrentVolume	Volume	只读	当前 kOS 处理器 所选定的 volume 此项可用于防止搞错盘符误删文件
Messages	MessageQueue	只读	当前 kOS 处理器 的报文队列
.....			

9.4.12

9.4.12 传感器 Sensor [结构体]

===== 点击以返回目录 =====



(1) 概述

传感器 Sensor 是一种特殊的 部件 Part，是部件派生出的。

专用于传感器部件。玩家可以通过 预设列表 Sensors 来获取传感器。

LIST SENSORS IN ss. //ss是当前载具上 传感器Sensor 的列表

(2) 结构体成员

该结构体派生自 部件 Part 结构体。

玩家可以在 部件 Part [结构体] 查看 部件 Part 结构体的详细信息。

Sensor 传感器结构体 成员列表:

结构体成员	成员类型	读写性	说明
Part 结构的成员			派生自此类型，拥有其全部成员
Active	Boolean	只读	传感器是否打开
Type	String	只读	传感器类型
Display	String	只读	读数（含单位）
PowerConsumption	Scalar	只读	需求电量
Toggle ()	无返回值	函数	传感器的开关切换
.....			

9.4.13

9.4.13 分离器 Decoupler [结构体]

===== 点击以返回目录 =====



(1) 概述

分离器 Decoupler 是一种特殊的部件Part。是部件派生出的。
它是所有分离器、发射架、对接口部件的基础结构体。

(2) 结构体成员

分离器 Decoupler 成员列表：

结构体成员	成员类型	读写性	说明
Part结构的成员			派生自此类型，拥有其全部成员
Separator	Separator	只读	对接口特性
.....			

9.4.14

9.4.14 对接口 DockingPort [结构体]

===== 点击以返回目录 =====



(1) 概述

对接口 DockingPort 是一种特殊的部件Part。是部件派生出的。专指对接口部件。

(2) 获取结构体

以下两种方法都可以获取当前载具上的对接口DockingPort

```
List DockingPorts in dps. //dps 是当前载具上 对接口DockingPort 的列表
```

```
Set dps to Ship:DockingPorts.
```

(3) 结构体成员

该结构体派生自 分离器 Decoupler 结构体。
玩家可以在 部件 Part [结构体] 查看 部件Part 结构体的详细信息。

DockingPort对接口结构体 成员列表：

结构体成员	成员类型	读写性	说明
Decoupler 结构的成员			派生自此类型，拥有其全部成员
AquireRange	Scalar (m)	只读	对接口的吸附距离
AquireForce	Scalar (kN)	只读	对接口的吸附力
AquireTorque	Scalar	只读	对接口吸附时的扭矩
ReengagedDistance	Scalar (m)	只读	对接口分离后离开多少距离才可重新对接
DockedShipName	String	只读	对接口对接的飞船名
NodePosition	Vector	只读	对接口在自机 SHIP - RAW坐标系的位置
NodeType	String	只读	对接口类型, "size0", "size1", "size2" 三种, 从小到大
PortFacing	Direction	只读	对接口的正对朝向
State	String	只读	对接状态
Undock ()	无返回值	函数	解除对接
Targetable	Boolean	只读	此对接口是否可被设为目标
.....			

*注1: :State 的值可以是: “Ready”、“Docked (docker)”、“Docked (dockee)”、“Docked (same vessel)”、“Disabled”、“PreAttached”

9.4.15

9.4.15 引擎 Engine [结构体]

===== 点击以返回目录 =====



(1) 概述

引擎 Engine 是一种特殊的部件 Part。是部件派生出的。

专指对引擎部件。玩家可以从 预设列表 Engines 中获取该结构体:

List Engines in egs. //egs 是当前载具上 引擎Engine 的列表

(2) 结构体成员

该结构体派生自 部件 Part 结构体。

玩家可以在 部件 Part [结构体] 查看 部件 Part 结构体的详细信息。

Engine引擎结构体 成员列表:

结构体成员	成员类型	读写性	说明
Enumerable结构的成员			派生自此类型，拥有其全部成员
Activate ()	无返回值	函数	启动引擎
Shutdown ()	无返回值	函数	关闭引擎
ThrustLimit	Scalar (%)	读写	引擎推力限制
MaxThrust	Scalar (kN)	只读	引擎最大推力 (节流阀和推力限制都拉到最大时)
MaxThrustAt (atm)	Scalar (kN)	只读	引擎最大推力关于气压的函数 若参数atm < 0 则视作atm = 0
Thrust	Scalar (kN)	只读	引擎当前推力
AvailableThrust	Scalar (kN)	只读	引擎允许最大推力 (节流阀拉到最大时) (引擎Off时为0)
AvailableThrustAt (atm)	Scalar (kN)	只读	引擎允许最大推力关于气压的函数 若参数atm < 0 则视作atm = 0
PossibleThrust	Scalar (kN)	只读	引擎可达最大推力 (节流阀拉到最大时) (无关引擎状态)
PossibleThrustAt (atm)	Scalar (kN)	只读	引擎可达最大推力关于气压的函数 若参数atm < 0 则视作atm = 0
FuelFlow	Scalar	只读	燃料燃烧速度, (unit / s)
ISP	Scalar (s)	只读	当前比冲
ISPAAt (atm)	Scalar	函数	比冲关于气压的函数 若参数atm < 0 则视作atm = 0
VacuumIsp	Scalar (s)	只读	真空比冲
VIsp	Scalar	只读	同上
SeaLevelIsp	Scalar (s)	只读	海平面比冲
S1Isp	Scalar	只读	同上
FlameOut	Boolean	只读	是否因为燃料烧光熄火了
Ignition	Boolean	只读	引擎是否已点火
AllowRestart	Boolean	只读	引擎是否能重启 (固推不能)
AllowShutdown	Boolean	只读	引擎是否能关闭
ThrottleLock	Boolean	只读	引擎节流阀是否锁定 (固推锁定的)
MultMode	Boolean	只读	是否多种工作模式
Modes	List	只读	工作模式列表
Mode	String	只读	当前工作模式
ToggleMode ()	无返回值	函数	切换工作模式
PrimaryMode	Boolean	读写	主工作模式
AutoSwitch	Boolean	读写	是否工作模式自动切换
HasGimbal	Boolean	只读	是否有矢量喷口
Gimbal	Gimbal	只读	矢量喷口
.....			

9.4.16

9.4.16 引擎矢量喷口 Gimbal [结构体]

===== 点击以返回目录 =====



(1) 概述

引擎矢量喷口 Gimbal 可以从 引擎Engine 的 :Gimbal 后缀里获得。

(2) 结构体成员

该结构体派生自 部件模组PartModule 结构体。

玩家可以在 部件模组 PartModule [结构体] 查看 部件模组PartModule 结构体的详细信息。

Gimbal引擎矢量喷口结构体 成员列表:

结构体成员	成员类型	读写性	说明
PartModule 结构的成员			派生自此类型，拥有其全部成员
Lock	Boolean	读写	矢量喷口是否锁住
Pitch	Boolean	读写	是否响应 Pitch 方向 调节
Yaw	Boolean	读写	是否响应 Yaw 方向 调节
Roll	Boolean	读写	是否响应 Roll 方向 调节
Limit	Scalar (%)	读写	矢量喷口的最大转动角度设置
Range	Scalar (°)	只读	矢量喷口的最大转动角度
ResponseSpeed	Scalar	只读	矢量喷口的转动速度
PitchAngle	Scalar	只读	当前的 Pitch 方向 调节值, -1~1
YawAngle	Scalar	只读	当前的 Yaw 方向 调节值, -1~1
RollAngle	Scalar	只读	当前的 Roll 方向 调节值, -1~1
.....			

9.5

9.5 航行与操控

9.5.1

9.5.1 命令序列 Stage [结构体]

===== 点击以返回目录 =====



(1) 概述

命令序列 Stage 对应载具发射时，其火箭点火、分离器分离、降落伞展开，等行为的分级顺序。

玩家可以通过 预设变量Stage 获取该结构体。

(2) Stage. 指令

Stage 指令可以单独成句，对应一次命令序列的实施，相当于按了一下空格键。

Stage.

*注1: Stage 指令实施时会有一定的冷却时间，不能一下子连着执行。

需要先 Wait 一会儿才行。

(3) 结构体成员

Stage命令序列结构体 成员列表:

结构体成员	成员类型	读写性	说明
Ready	Boolean	只读	是否可以执行 Stage 指令
Number	Scalar	只读	当前是第几个命令命令序列
Resources	Resource列表	只读	当前命令序列能用到的燃料
ResourcesLex	Lexicon	只读	以词典结构存储的Resource列表
NextDecoupler	Decoupler或Separator	只读	要被激活的Decoupler或Separator
NextSeparator	Decoupler或Separator	只读	同上
.....			

(4) 实际例子

利用 Stage:NextDecoupler 实现火箭自动Stage。

```
STAGE.
IF stage:nextDecoupler:isType("LaunchClamp")
    STAGE.
IF stage:nextDecoupler <> "None" {
    WHEN availableThrust = 0 or (
        stage:resourcesLex["LiquidFuel"]:amount = 0 and
        stage:resourcesLex["SolidFuel"]:amount = 0) {
    THEN {
        STAGE.
        return stage:nextDecoupler <> "None".
    }
}
```

9.5.2

9.5.2 发射架 LaunchClamp [结构体]

===== 点击以返回目录 =====



(1) 概述

发射架 LaunchClamp 是一种特殊的 分离器 Decoupler。是分离器派生出的。
他可用于识别发射架部件。

(2) 结构体成员

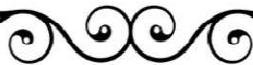
发射架 LaunchClamp 成员列表:

结构体成员	成员类型	读写性	说明
Decoupler结构的成员			派生自此类型，拥有其全部成员
.....			

9.5.3

9.5.3 高度 ALT [结构体]

===== 点击以返回目录 =====



(1) 概述

高度 ALT 是从其他地方映射过来打包而成的虚拟结构体。

作用是快速查询一些航行时重要的高度类数据。

玩家可以使用 预设变量ALT 获取。

(2) 结构体成员

ALCT高度结构体 成员列表：

结构体成员	成员类型	读写性	说明
Apoapsis	Scalar (m)	只读	当前载具的轨道远点高度
Periapsis	Scalar (m)	只读	当前载具的轨道近点高度
Rader	Scalar (m)	只读	当前载具的离地高度
.....			

9.5.4

9.5.4 等待时间 ETA [结构体]

===== 点击以返回目录 =====



(1) 概述

等待时间 ETA 是从其他地方映射过来打包而成的虚拟结构体。

作用是快速查询一些航行时重要的时间类数据。

玩家可以使用 预设变量ETA 获取。

(2) 结构体成员

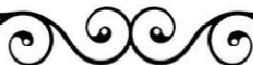
ETA等待时间结构体 成员列表：

结构体成员	成员类型	读写性	说明
Apoapsis	Scalar (s)	只读	距离下一个轨道远点的时间
Periapsis	Scalar (s)	只读	距离下一个轨道近点的时间
Periapsis	Scalar (s)	只读	距离下一个轨道的时间
.....			

9.5.5

9.5.5 调姿管理 SteeringManager [结构体]

===== 点击以返回目录 =====



(1) 概述

调姿管理 SteeringManager 对应使用 Lock Steering to 语句来控制载具转向时的 PID算法。玩家可以通过 预设变量SteeringManager 获取。

(2) 结构体成员

SteeringManager 调姿管理结构体 成员列表:

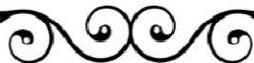
结构体成员	成员类型	读写性	说明
PitchPID	PIDLoop	只读	俯仰方向的PID
YawPID	PIDLoop	只读	偏航方向的PID
RollPID	PIDLoop	只读	滚转方向的PID
Enabled	Boolean	只读	驾驶管理是否在控制载具
Target	Direction	只读	若Enabled = True则返回目标朝向 如果是False，则返回载具朝向
ResetPIIDs ()	无返回值	函数	重置调姿相关的 PID 的积分项
ResetToDefault ()	无返回值	函数	重置调姿相关的 PID 的参数
ShowFacingVectors	Boolean	读写	是否箭头绘制VecDraw形式显示 载具前方、头顶、右舷的方向
ShowAngularVectors	Boolean	读写	是否箭头绘制VecDraw形式显示 载具Pitch、Yaw、Roll方向的 角速度的方向和大小
ShowSteeringStats	Boolean	读写	指令窗口是否要在每个物理帧 进行清屏 (用于配合Print 指令来实时显示数据)
WriteCsvFiles	Boolean	读写	是否实时记录调姿PID数据并 将他们导出到一个csv文件
PitchTS	Scalar (s)	读写	Pitch方向的PID内的稳定时间 和PID整定有关
YawTS	Scalar (s)	读写	Yaw方向的PID内的稳定时间 和PID整定有关
RollTS	Scalar (s)	读写	Roll方向的PID内的稳定时间 和PID整定有关
MaxStoppingTime	Scalar (s)	读写	最大刹车时间 用来限制最大角速度的
RollControlAngleRange	Scalar (deg)	读写	滚转调节阈值，滚转偏差超 过此阈值才会进行调节 范围1e (- 16) ~ 180
AngleError	Scalar (deg)	只读	误差角。载具朝向矢量和目标 朝向矢量的夹角。和 Pitch 和 Yaw 方向误差角有关
PitchError	Scalar (deg)	只读	Pitch方向 的误差角
YawError	Scalar (deg)	只读	Yaw方向 的误差角
RollError	Scalar (deg)	只读	Roll方向 的误差角
PitchTorqueAdjust	Scalar (kN)	读写	Pitch方向 的输出扭矩偏移
YawTorqueAdjust	Scalar (kN)	读写	Yaw方向 的输出扭矩偏移
RollTorqueAdjust	Scalar (kN)	读写	Roll方向 的输出扭矩偏移
PitchTorqueFactor	Scalar	读写	Pitch方向 的扭矩偏移因子
YawTorqueFactor	Scalar	读写	Yaw方向 的扭矩偏移因子
RollTorqueFactor	Scalar	读写	Roll方向 的扭矩偏移因子
.....			

*注1：在 Pitch、Yaw、Roool方向，
实际扭距 = (PID输出 + 输出扭矩偏移) * 扭距偏移因子。

9.5.6

9.5.6 变轨计划 ManeuverNode [结构体]

===== 点击以返回目录 =====



(1) 概述

变轨计划 ManeuverNode 对应游戏里玩家在地图界面上规划的轨道机动计划。

(2) 使用结构体

创建变轨计划结构

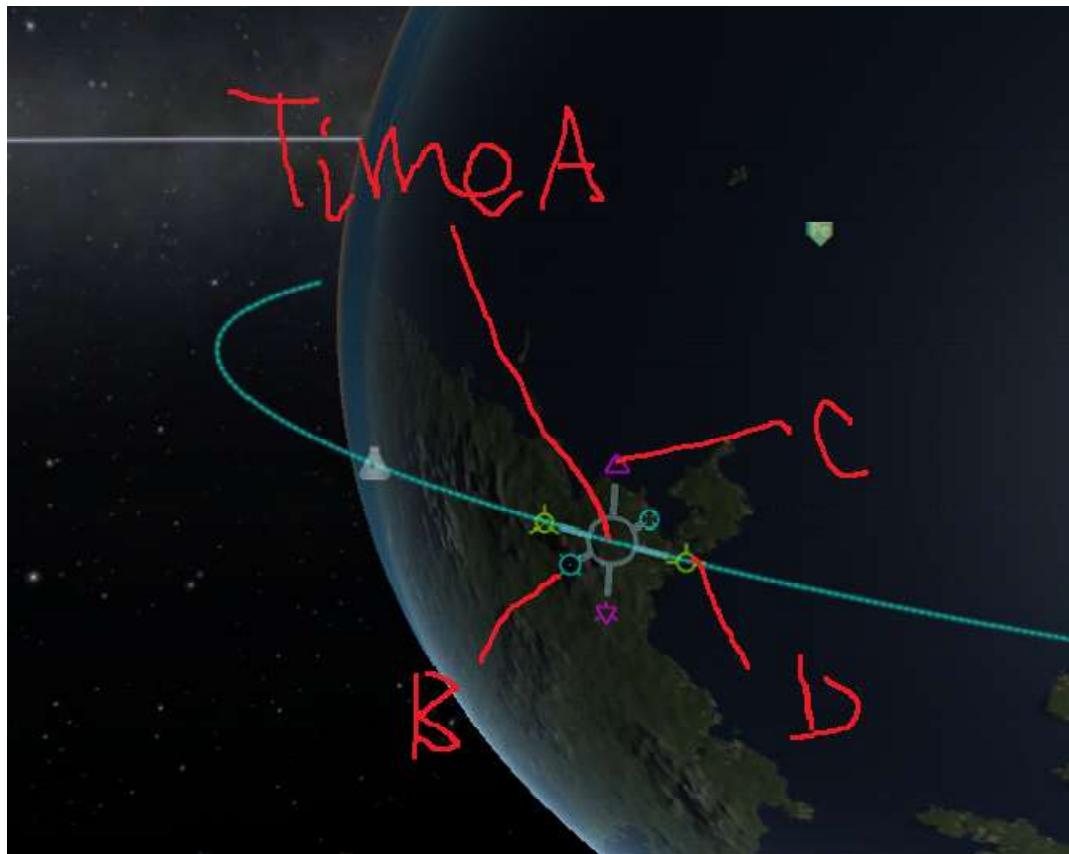
```
Set MyNode to Node(A,B,C,D). //生成一个位于时间A(s)的变轨计划,
//在径向(朝外)、法向(右手螺旋)、切向(朝前)的DV(m/s)分别是B, C, D

add MyNode. remove MyNode. //添加和移除轨道规划

//NextNode 是一个预设变量, 表示下一个已添加的变轨计划
Set MyNode to NextNode.

HasNode //这是个 Boolean类型 的预设变量, 表示当前载具是否有变轨计划

//这是个 ManeuverNode列表类型 的预设变量
//表示当前载具拥有的变轨计划
AllNodes
```



另外，下面的东西可能有助于玩家计算确定变轨计划ManeuverNode 的参数：

预测轨道信息的函数（如果设置了轨道规划，那么按照轨道规划的走）

位置：PositionAt(Orbitable,time)
速度：VelocityAt(Orbitable,time)
所处轨道：OrbitAT(Orbitable,time)

(3) 用法示例

```

SET myNode to NodeE( TIME:SECONDS+200, 0, 50, 10 ).  

//TIME:SECONDS+200 表示当前时间200秒后  

ADD myNode.

Print PositionAt(Ship,Time:Seconds+600) //当前载具600秒后的位置  

Print VelocityAt(Ship,Time:Seconds+600) //当前载具600秒后的速度

// creates a node 60 seconds from now with
// prograde = 100 m/s
SET X TO NODE(TIME:SECONDS+60, 0, 0, 100).

ADD X.           // adds maneuver to flight plan

PRINT X:PROGRADE. // prints 100.  

PRINT X:ETA.      // prints seconds till maneuver  

PRINT X:DELTAV    // prints delta-v vector

REMOVE X.         // remove node from flight plan

// Create a blank node
SET X TO NODE(0, 0, 0, 0).

ADD X.           // add Node to flight plan
SET X:PROGRADE to 500. // set prograde dV to 500 m/s
SET X:ETA to 30.    // Set to 30 sec from now

PRINT X:ORBIT:APOAPSIS. // apoapsis after maneuver
PRINT X:ORBIT:PERIAPSIS. // periapsis after maneuver

```

(4) 结构体成员

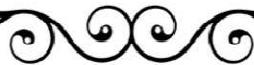
ManeuverNode变轨计划结构体 成员列表：

结构体成员	成员类型	读写性	说明
DeltaV	Vector (m / s)	只读	变轨需要付出的delta - V
BurnVector	Vector (m / s)	只读	同上
ETA	Scalar (s)	读写	距离变轨点的时间
Prograde	Scalar (m / s)	读写	Delta - V 在变轨点的速度方向上的分量
RadialOut	Scalar (m / s)	读写	Delta - V 在径向朝里方向上的分量
Normal	Scalar (m / s)	读写	Delta - V 在轨道法线向上的分量
Orbit	Orbit	只读	变轨后的理论轨道
.....

9.5.7

9.5.7 资源转移 ResourceTransfer [结构体]

===== 点击以返回目录 =====



(1) 概述

资源转移 ResourceTransfer 对应的是游戏里玩家在不同部件之间进行燃料转移的行为。

(2) 进行资源转移

创建资源转移函数

燃油转移：两个函数，负责转移部分和全部资源

Set transferFoo TO Transfer (资源名称,供油部件,受油部件,数量)

Set transferBar TO TransferAll (资源名称,供油部件,受油部件)

用法代码示例：

```
LIST ELEMENTS IN elist.
Set foo TO TRANSFERALL("OXIDIZER", elist[0], elist[1]).
Set foo:ACTIVE to TRUE.
```

(3) 结构体成员

ResourceTransfer 资源转移结构体 成员列表：

结构体成员	成员类型	读写性	说明
Status	String	只读	转移状态
Message	String	只读	针对转移状态的详细说明
Goal	Scalar (unit)	只读	设定的转移量，没有转移命令时值为 -1
Transferred	Scalar (unit / s)	只读	资源转移速度
Resource	String	只读	被转移的资源种类
Active	Boolean	读写	是否启动转移，默认False
.....			

*注1：:STATUS 的值可以是：“Inactive”、“Finished”、“Failed”、“Transferring”。

9.5.8

9.5.8 飞行操控 Control [结构体]

===== 点击以返回目录 =====

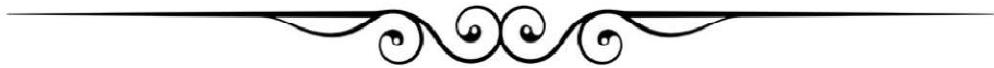


详见 底层控制 Control [结构体] 和 驾驶员输入 Control [结构体]。

9.5.9

9.5.9 矢量 Vector [结构体]

===== 点击以返回目录 =====

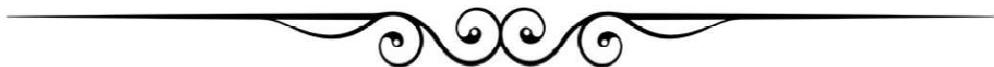


详见 矢量 Vector [结构体]。

9.5.10

9.5.10 朝向 Direction [结构体]

===== 点击以返回目录 =====



详见 朝向 Direction [结构体]。

9.5.11

9.5.11 经纬坐标 GeoCoordinates [结构体]

===== 点击以返回目录 =====



详见 经纬坐标 GeoCoordinates [结构体]。

9.6

9.6 通讯与科研

9.6.1

9.6.1 通讯连接 Connection [结构体]

===== 点击以返回目录 =====



(1) 概述

表明玩家与其他处理器/载具通讯的能力。玩家可以用 通讯连接Connection 来查明这种通讯是否存在，并发送消息。

(2) 获得通讯连接

有两种通讯方式：载具内通讯和载具间通讯。

载具内通讯用于与同一载具下的不同 kOS处理器通讯。

```
//使用 Parrocessor 函数获得指定 标签名Tag 的部件
SET MY_PROCESSOR TO PROCESSOR("second").
SET MY_CONNECTION TO MY_PROCESSOR:CONNECTION.
```

载具间通讯用于与同在物理加载范围内的其它载具通讯。

```
//使用 Vessel 函数获得指定 载具名 的载具
SET MY_VESSEL TO VESSEL("dunarover").
SET MY_CONNECTION TO MY_VESSEL:CONNECTION.
```

(3) 结构体成员

Connection 通讯连接结构体 成员列表:

结构体成员	成员类型	读写性	说明
IsConnected	Boolean	只读	检查是否有从该载具到自机的链接
Delay	Scalar (s)	只读	通讯延迟
Destination	Vessel or kOSProcessor	只读	链接的目的地
SendMessage (message)	Boolean	函数	发送报文, 返回值代表是否成功发送
发送信息			

9.6.2

9.6.2 报文队列 MessageQueue [结构体]

----- 点击以返回目录 -----



(1) 概述

按照先入先出原则的通讯数据收件箱。

(2) 访问报文队列

访问当前 kOS 处理器的报文队列。

```
SET QUEUE TO CORE:MESSAGES.
PRINT "Number of messages on the queue: " + QUEUE:LENGTH.
```

访问自机的报文队列。

```
SET QUEUE TO SHIP:MESSAGES.
```

(3) 结构体成员

MessageQueue 报文队列结构体 成员列表:

结构体成员	成员类型	读写性	说明
Empty	Boolean	只读	检查是否有报文
Length	Scalar	只读	有几条报文
Pop ()	Message	函数	返回并清除最早的一条报文
Peek ()	Message	函数	返回但不清除最早的一条报文
Clear ()	无返回值	函数	清空所有报文
Push (message)	无返回值	函数	在报文队列里添加一条报文
.			

9.6.3

9.6.3 报文 Message [结构体]

----- 点击以返回目录 -----



(1) 概述

报文是进行通讯时收发数据的最小单位。

(2) 用法示例

对报文进行收发：

```
// if there is a message in the ship's message queue
// we can forward it to a different CPU

// cpu1
SET CPU2 TO PROCESSOR("cpu2").
CPU2:CONNECTION:SENDMESSAGE(SHIP:MESSAGES:POP).

// cpu2
SET RECEIVED TO CORE:MESSAGES:POP.
PRINT "Original message sent at: " + RECEIVED:CONTENT:SENTAT.
```

(3) 结构体成员

该结构体是可序列化（Serializable）的结构体。

Message报文结构体 成员列表：

结构体成员	成员类型	读写性	说明
SentAt	Timespan	只读	报文的发送时间
ReceivedAt	Timespan	只读	报文的接收时间
Sender	Vessel or Boolean	只读	报文的发送者， 如果发送者已经不存在就返回False
HasSender	Boolean	只读	检查报文的发送者是否还存在
Content	Structure	只读	报文的内容
.....			

*注1：可序列化是指，
该类型变量能通过 WriteJSON 函数 和 ReadJSON 函数 被读写至 json 文件，
指该类型变量能用于僚机之间的通信数据互传。

9.6.4

9.6.4 科研数据 ScienceData [结构体]

===== 点击以返回目录 =====



(1) 概述

玩家可以从 科研实验模组 ScienceExperimentModule 里获取科研数据。

(2) 结构体成员

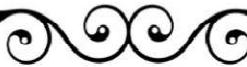
ScienceData科研数据结构体 成员列表:

结构体成员	成员类型	读写性	说明
Title	String	只读	科研实验名称
ScienceValue	Scalar	只读	回收数据能获得的科技点
TransmitValue	Scalar	只读	天线传输数据能获得科技点
DataAmount	Scalar	只读	数据量
.....			

9.6.5

9.6.5 科研实验模组 ScienceExperimentModule [结构体]

===== 点击以返回目录 =====



(1) 概述

科研实验模组 ScienceExperimentModule 是一种特殊的 部件模组PartModule。派生自部件模组。玩家可以按一下方式获取 科研实验模组。

```
//假设P是某科研部件，那么M就是科研实验模组
SET M TO P:GETMODULE("ModuleScienceExperiment").
```

(2) 结构体成员

该结构体派生自 部件模组PartModule 结构体。

玩家可以在 部件模组 PartModule [结构体] 查看 部件模组PartModule 结构体的详细信息。

ScienceExperimentModule科研试验模组结构体 成员列表:

结构体成员	成员类型	读写性	说明
ScienceExperimentModule 结构的成员			派生自此类型，拥有其全部成员
Deploy ()	无返回值	函数	展开并进行试验，如果有数据或者不能实验则会失败
Reset ()	无返回值	函数	重置实验，如果不能实验则会失败
Transmit ()	无返回值	函数	回传实验数据，没有实验数据则会失败
Dump ()	无返回值	函数	丢弃实验数据，没有实验数据则会失败
Inoperable ()	Boolean	只读	是否能进行实验
Rerunnable	Boolean	只读	是否能多次试验
Deployed	Boolean	只读	设备是否展开
HasData	Boolean	只读	是否有实验数据
Data	ScienceData列表	只读	科研数据的列表
.....			

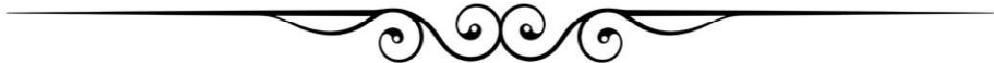
9.7

9.7 轨道与天体

9.7.1

9.7.1 轨道物 Orbital [结构体]

===== 点击以返回目录 =====



(1) 概述

轨道物Orbital 描述了沿稳定轨道运行的物体的信息。

通常可从 载具Vessel 或 天体Body 获得。

(2) 结构体成员

Orbital轨道物结构体 成员列表:

结构体成员	成员类型	读写性	说明
Name	String	只读	轨道物名称 (载具名称或天体名称)
Body	Body	只读	环绕天体
HasBody	Boolean	只读	是否有环绕天体 (除了Sun都有环绕天体)
HasOrbit	Boolean	只读	是否有轨道 (除了Sun都有轨道)
HasObt	Boolean	只读	同上
Obt	Orbit	只读	轨道物的当前轨道
Orbit	Orbit	只读	同上
Up	Direction	只读	轨道物背向环绕天体中心的方向
North	Direction	只读	平行于轨道物在环绕天体的海平面投影点的罗盘上的北方朝向
Prograde	Direction	只读	轨道物的轨道速度方向
SrfPrograde	Direction	只读	轨道物的轨道速度反方向
Retrograde	Direction	只读	轨道物关于环绕天体表面速度的方向
SrfRetrograde	Direction	只读	轨道物关于环绕天体表面速度的反方向
Position	Vector (m)	只读	轨道物的矢量位置 (Ship - RAW 坐标系)
Velocity	OrbitalVelocity	只读	轨道物的轨道速度 (Ship - RAW 坐标系)
Distance	Scalar (m)	只读	从当前载具到轨道物的距离
Direction	Direction	只读	从当前载具指向轨道物的朝向
Latitude	Scalar (deg)	只读	轨道物在环绕天体的海平面投影点纬度
Longitude	Scalar (deg)	只读	轨道物在环绕天体的海平面投影点经度
Altitude	Scalar (m)	只读	轨道物在环绕天体的海平面高度
Geoposition	GeoCoordinates	只读	轨道物在环绕天体的海平面投影点的地理坐标
Patches	Orbit列表	只读	轨道物的飞行轨道列表
.....			

9.7.2

9.7.2 轨道 Orbit [结构体]

===== 点击以返回目录 =====



(1) 概述

轨道Orbit是描述天体、载具、小行星在重力场作用下稳定运动的轨道。

对于天体，其轨道是固定不变的。

对于载具，考虑到载具可能会经历变轨点、逃逸、捕获的变化。在这些变化点之间的每一段都是一个轨道Orbit的一小段，载具的实际轨道是一系列轨道Orbit，取其片段拼接而成的。

玩家可从天体Body、载具Vessel、轨道物Orbitable结构体获得轨道Orbit结构体。

(2) 结构体成员

Orbit轨道结构体 成员列表：

结构体成员	成员类型	读写性	说明
Name	String	只读	轨道名称
Apoapsis	Scalar (m)	只读	轨道远点, AP点
Periapsis	Scalar (m)	只读	轨道近点, PE点
Body	Body	只读	环绕天体
Period	Scalar (s)	只读	轨道周期
Inclination	Scalar (deg)	只读	轨道倾角
Eccentricity	Scalar	只读	轨道离心率
SemiMajorAxis	Scalar (m)	只读	轨道长半轴
SemiMinorAxis	Scalar (m)	只读	轨道短半轴
Lan	Scalar (deg)	只读	升交点黄经 (纬度上升过程中轨道× 和赤道面的交点)
LongitudeOfAscendingNode	Scalar (deg)	只读	同上
ArgumentOfPeriapsis	Scalar (deg)	只读	近点幅角 (从升交点× 到近点扫过的角度)
TrueAnomaly	Scalar (deg)	只读	真近点角 (从近点到× 轨道物扫过的角度)
MeanAnomalyAtEpoch	Scalar (deg)	只读	平近点角 (从近点到× 轨道物在虚拟正圆× 上扫过的角度)
Epoch	Scalar	只读	轨道演算时间戳
Transition	String	只读	轨道衔接类型
Position	Vector (m)	只读	轨道物的当前位置
Velocity	OrbitableVelocity	只读	轨道物的当前速度
NextPatch	Orbit	只读	下一段轨道
NextPatchAt	Scalar	只读	几秒后进入到下一段轨道
HasNextPatch	Boolean	只读	是否有下一段轨道
.....

*注1: Orbit:Transition 的值可以是:

initial	初始轨道
final	最终轨道Ship
encounter	会被天体捕获的轨道
escape	会逃逸出当前天梯的轨道
maneuver	会遇到变轨点的轨道

9.7.3 轨道速度 OrbitableVelocity [结构体]

===== 点击以返回目录 =====



(1) 概述

轨道速度结构体，用于描述飞船和天体沿着轨道运行的速度。

轨道速度 OrbitableVelocity 通常通过 轨道物Orbitable 来获得的。

(2) 结构体成员

OrbitableVelocity 轨道速度结构体 成员列表:

结构体成员	成员类型	读写性	说明
Orbit	Vector (m / s)	只读	轨道速度
Surface	Vector (m / s)	只读	表面速度
.....

(3) 实际例子

代码示例:

```
Set VORB TO Ship:Velocity:Orbit.
Set VSDF TO Ship:Velocity:Surface.
Set MUNORB TO MUN:Velocity:Orbit.
Set MUNSRF TO MUN:Velocity:Surface.
```

9.7.4

9.7.4 天体 Body [结构体]

===== 点击以返回目录 =====



(1) 调用结构体

KSP 游戏中的天体和数量都是固定的。

玩家可以通过 Body 函数 获得天体结构体，也直接用天体名称同名的预设变量。

```
Set a to Body("Mun"). //这两句话一个效果
Set b to Mun.
```

原版 KSP 中有如下天体:

Kerbol	恒心Kerbol
Moho	行星Moho
Eve	行星Eve
Gilly	卫星Gilly
Kerbin	行星Kerbin
Mun	卫星Mun
Minmus	卫星Minmus
Duna	行星Duna
Ike	卫星Ike
Dres	行星Dres
Jool	行星Jool
Laythe	卫星Laythe
Vall	卫星Vall
Tylo	卫星Tylo
Bop	卫星Bop
Pol	卫星Pol
Eeloo	行星Eeloo

(2) 结构体成员

该结构体派生自 轨道物Orbitable 结构体。

玩家可以在 轨道物 Orbitable [结构体] 查看 轨道物Orbitable 结构体的详细信息。

该结构体是可序列化（Serializable）的结构体。

Body天体结构体 成员列表:

结构体成员	成员类型	读写性	说明
Orbitable结构的成员			派生自此类型，拥有其全部成员
Name	String	只读	天体名称
Description	String	只读	天体描述文字
Mass	Scalar (kg)	只读	天体质量
HasOcean	Boolean	只读	是否有海洋
HasSolidSurface	Boolean	只读	是否有固体表面
OrbitingChildren	Body列表	只读	环绕天体的列表
Altitude	Scalar (m)	只读	该天体中心距离其母星海平面的距离
RotationPeriod	Scalar (s)	只读	自转周期
Radius	Scalar (m)	只读	天体海平面半径
MU	Scalar ($N \cdot m^2 / kg$)	只读	天体的引力常数 $G * M$
Atm	Atmosphere (kPa)	只读	天体的大气压
AngularVel	Vector (Rad / s)	只读	天体的自转角速度 (Ship - Raw 坐标系)
GeopositionOf (Vector)	GeoCoordinates	函数	从当前载具为起点，画出给定矢量 Vector 后其终点所在的经纬坐标
GeopositionLatlng (la, lo)	GeoCoordinates	函数	在该天体 la 纬度，lo 经度的经纬坐标
AltitudeOf (Vector)	Scalar (m)	函数	从当前载具为起点，画出给定矢量 Vector 后其终点所在海平面高度
SOIRadius	Scalar (m)	只读	引力范围球半径
RotationAngle	Scalar (deg)	只读	零经零纬点和春分点夹角
.....			

*注1：可序列化是指，

该类型变量能通过 WriteJSON 函数 和 ReadJSON 函数 被读写至 json 文件，指该类型变量能用于僚机之间的通信数据互传。

*注2：注意区别 GeopositionLatlng(la,lo) 和
轨道物Orbitable 结构体的GEOPOSITION 后缀。
后者给出的是在其母星表面的经纬坐标。

(3) 相关例子

有关 AltitudeOf() 函数

Ship:Body:AltitudeOf(Ship:POSITION)

表示当前飞船位置，在所在天体的海平面高度。

有关 GEOPOSITIONOF() 函数

Ship:Body:GEOPOSITIONOF(Ship:POSITION + 1000*Ship:NORTH)

表示飞船以北1km的地理经纬坐标。

9.7.5

9.7.5 大气层 Atmosphere [结构体]

----- 点击以返回目录 -----



(1) 概述

大气层Atmosphere 描述了一个星球上的大气层状况。

通常可从天体Body结构体获得。

(2) 结构体成员

Atmosphere 大气层结构体 成员列表:

结构体成员	成员类型	读写性	说明
Body	String	只读	大气层的所属天体
Exists	Boolean	只读	所属天体是否有大气层
Oxygen	Boolean	只读	大气层中是否有氧气
SeaLevelPressure	Scalar (atm)	只读	海平面气压
AltitudePressure (alt)	Scalar (atm)	函数	在海拔alt高度的气压
Height	Scalar (m)	只读	大气层高度
MolarNass	Scalar (kg / mol)	只读	每摩尔大气分子质量
AdiabaticIndex	Scalar	只读	大气绝热指数
AdbIdx	Scalar	只读	同上
AltitudeTrmperature (alt)	Scalar	只读	给定高度的大气温度
AltTemp (alt)	Scalar	只读	同上
.....

(3) 气压随高度分布

在 kOS 中，大气层气压 P 随海拔高度 h 变化都是一个指数式的关系，如下：

$$P[h] = P_0 * e^{-\frac{h}{H}}$$

其中 P_0 为海平面大气压, H 为系数。每个星球的 P_0 和 H 都有所不同。

例如坎星上 $P_0 = 101.325 \text{ kPa}$, $H = 5600 \text{ m}$ 。

98

9.8 GUI 部件

981

981 GUI 设计技巧

[点击以返回目录](#)



GUI 允许玩家在屏幕中创建自己的会话窗口，

窗口里可以有文字、按钮等 UI 的常见元素。

玩家可以据此开发属于自己的控制系统，取代 MJ 等辅助控制类 MOD。



(1) GUI 的两种用法：调用(CallBacks)与轮询(Polling)

调用(CallBaxks) 就是将玩家对控件A 的操作行为关联上 函数B 的函数指针。

例如下面就是将点击按钮 关联上 隐藏窗体 的例子。

```
//一个初值为假的值，用来标记是否要关闭窗口
Set isClosed to False.

//将点击关闭按钮的动作关联到一个把标记改为真值的函数上
Function CloseClick {
    Set isClosed to True.
}

Set ButtonClose:OnClick to CloseClick@.

Wait Until isClosed. //一旦这个标记为真了，就隐藏窗体
gui:Hide().
```

轮询(Pooling) 则是指用循环结构或触发器不断查询空间状态，根据情况主动调用相应函数。

```
until thisButton:TAKEPRESS {
    wait 0.001.
} //此循环会一直运行到玩家按下按钮为止
```

(2) 创建一个窗体

GUI (宽度) // 不写高度则表示高度自适应
GUI (宽度, 高度)

玩家要创建窗体，先要使用 GUI函数 创建一个 GUI结构。

实际例子：

```
SET gui TO GUI(200).
SET button TO gui:ADDBUTTON("OK").
gui:SHOW().
UNTIL button:TAKEPRESS WAIT(0.1).
gui:HIDE().
```

之后要往这个窗体里添加控件，都要通过 `gui:Add` 函数来进行。

(3) 移除所有窗体

`ClearGUIs()`

此函数将调用所有窗体 `GUI` 的 `Hide()` 函数和 `Dispose()` 函数，一键实现所有窗体的清除。

```
ClearGUIs().
```

(4) 通讯延迟

正常情况下是没延迟的。但是如果使用了通讯限制 `MOD`，从 `GUI` 的操作到载具的响应，就会有相应延迟。

9.8.2

9.8.2 控件 Widget [结构体]

===== 点击以返回目录 =====



(1) 概述

控件 `Widget` 是所有控件的基础结构体，所有 `GUI` 相关的结构体都是由控件 `Widget` 派生而来的，所有 `GUI` 相关的结构体至少都有控件 `Widget` 的成员内容。

(2) 结构体成员

`Widget` 控件结构体 成员列表：

结构体成员	成员类型	读写性	说明
Show ()	无返回值	函数	显示该控件
Hide ()	无返回值	函数	隐藏该控件
Visible	Scalar	只读	该控件内部的其他可见控件的数量
Dispose ()	无返回值	函数	永久清除该控件。但 :Dispose () 没有隐藏作用，使用时要先 :Hide () 再 :Dispose ()
Enabled	Enabled	读写	是否允许用户交互，False值时控件就“灰掉”了
Style	Style	读写	控件的样式。kOS 为所有控件提供了默认样式，玩家也可以使用 :Style来自定义样式
GUI	GUI	只读	返回该控件所在的 GUI 窗体
Parent	Box	只读	返回该控件的上一级 Box 框体
HasParent	Boolean	只读	检查该控件是否有上一级控件
.....			

9.8.3

9.8.3 窗体 GUI [结构体]

===== 点击以返回目录 =====



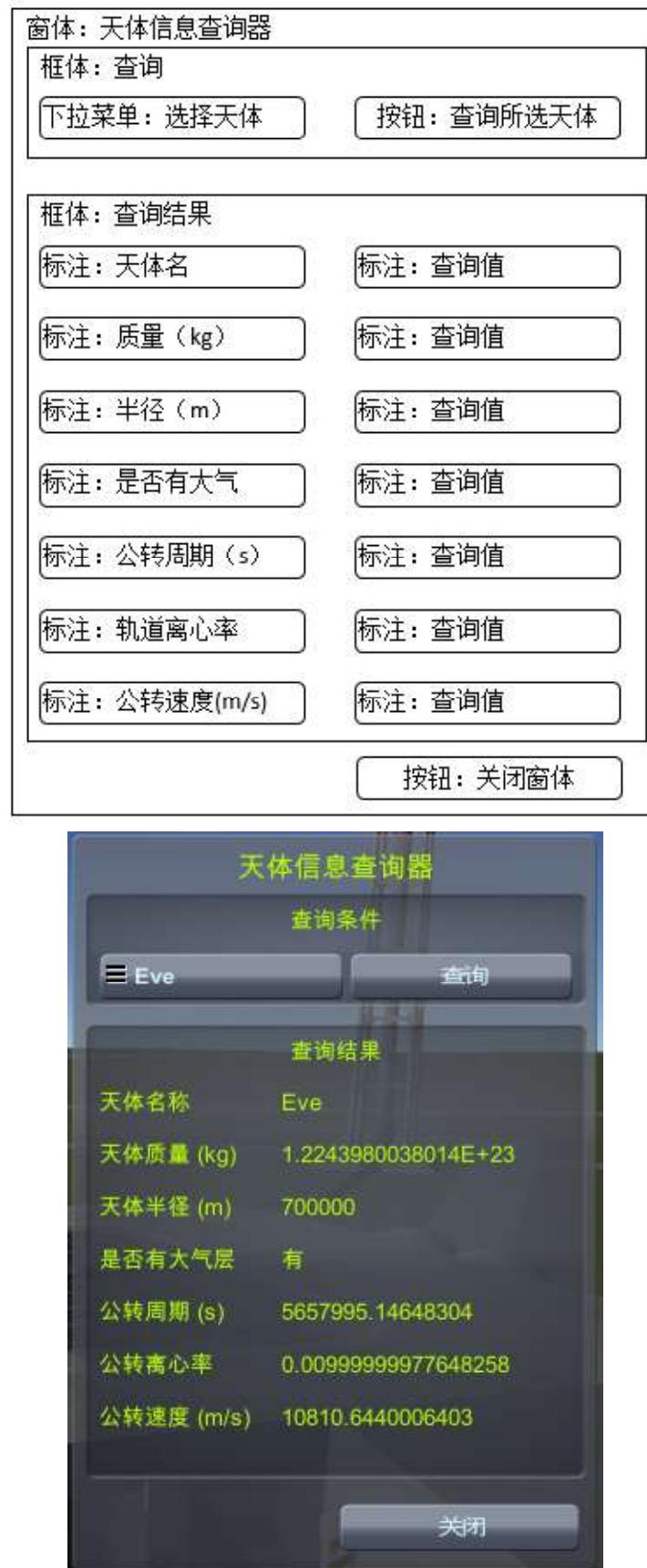
(1) 概述

窗体GUI 直接对应的是会话窗口，内容为会话窗口的属性。

一个会话窗口就有一个窗体GUI结构体。

kOS 中所有的控件都排版布置在一种叫 框体Box 的控件内部。

而位于顶层的窗体GUI，实际上就是框体Box 的强化版本，增加了窗口属性的内容。



上图为实例8: GUI——天体信息查询器 中窗体架构设计图与实物图的对比。

(2) 创建结构体

使用 GUI 函数可以创建该结构体。

```
Set a to GUI(400).      //横向 400 像素, 纵向自适应
Set a to GUI(400,300). //横向 400 像素, 纵向 300 像素
```

(3) 结构体成员

该结构体派生自 框体Box 结构体。

玩家可以在 框体 Box [结构体] 查看 框体Box 结构体的详细信息。

GUI窗体结构体 成员列表:

结构体成员	成员类型	读写性	说明
Box 结构的成员			派生自此类型, 拥有其全部成员
X	Scalar (pixels)	读写	该窗体左上角的 X 像素坐标
Y	Scalar (pixels)	读写	该窗体左上角的 Y 像素坐标
Draggable	Boolean	读写	是否允许用户拖动该窗体
ExtraDelay	Scalar (s)	读写	从该窗体操作到载具响应的延迟, 可以 用来模拟 通讯延迟
Skin	SKin	读写	该窗体的皮肤
ToolTip	String	读写	鼠标悬停时的标签文本
Show ()	无返回值	函数	显示该窗体
Hide ()	无返回值	函数	隐藏该窗体
.....			

9.8.4

9.8.4 框体 Box [结构体]

===== 点击以返回目录 =====



(1) 概述

框体Box 是一种辅助排版的控件, 框体Box 带有一种矩形范围,
使用 Box:Add 函数添加的其他控件都只能出现在这个矩形范围内。

(2) 添加结构体

使用 框体Box 的以下后缀函数可以在 框体 中添加 框体 Box。

AddHLayout ()	在框体中新建一个内部成员 排列顺序为水平的框体
AddVLayout ()	在框体中新建一个内部成员 排列顺序为垂直的框体
AddHBox ()	与 AddHLayout () 相同， 区别仅是框体有框线
AddVBox ()	与 AddVLayout () 相同， 区别仅是框体有框线
AddStack ()	在框体中新建一个可堆叠框体 多个此种框体共用同一位置 可用 ShowOnly 函数切换显示

(3) 结构体成员

该结构体派生自 控件 Widget 结构体。

玩家可以在 控件 Widget [结构体] 查看 控件 Widget 结构体的详细信息。

Box 框体结构体 成员列表：

结构体成员	成员类型	读写性	说明
Widget结构的成员			派生自此，拥有其全部成员
AddLabel (str)	Label	函数	在其中新建一个文本
AddButton (str)	Button	函数	在其中新建一个可点击按钮 按钮被点击后会立即弹回来
AddCheckBox (str, bool)	Button	函数	在其中新建一个可切换按钮 按钮有按下和弹起两种状态 对应 True 和 False 两种值 bool 参数对应按钮的初始值
AddTipDisplay ()	TipDisplay	函数	在框体中新建一个提示信息
AddRadioButton (str, bool)	Button	函数	在其中新建一个单选按钮 这是个排他的可点击按钮 当单选按钮按下时，框体 内其他按钮都会关闭。
AddTextField (str)	TextField	函数	在其中新建一个文本框
AddPopupMenu ()	PopupMenu	函数	在其中新建一个下拉菜单
AddHSlider (init, min, max)	Slider	函数	在其中新建一个水平滑块 (初值, 最小值, 最大值)
AddVSlider (init, min, max)	Slider	函数	在其中新建一个垂直滑块 (初值, 最小值, 最大值)
AddHLayout ()	Box	函数	在其中新建一个内部成员 排列顺序为水平的框体
AddVLayout ()	Box	函数	在其中新建一个内部成员 排列顺序为垂直的框体
AddHBox ()	Box	函数	与 AddHLayout () 相同， 区别仅是框体有框线
AddVBox ()	Box	函数	与 AddVLayout () 相同， 区别仅是框体有框线
AddStack ()	Box	函数	在其中新建一个可堆叠框体 多个此种框体共用同一位置 可用 ShowOnly 函数切换显示
AddScrollBox ()	ScorllBox	函数	在其中新建一个滚动框体
AddSpacing (PixelSize)	Spacing	函数	在其中新建一个空白区域
Widgets	Widget列表	只读	导出该框体内控件的列表
RadioValue	String	只读	返回打开的单选按钮的名称
OnRadioCHange	kOSDelegate	读写	当框体内有单选按钮状态发 生变化时，挂钩的函数指针
ShowOnly (widget)	无返回值	函数	仅显示某个控件
Clear ()	无返回值	函数	清空所有控件
.....			

9.8.5

9.8.5 滚动框体 ScrollBox [结构体]

===== 点击以返回目录 =====



(1) 概述

框体Box 的内容物添加太多时，在窗口里会显示不全。

滚动框体ScrollBox 比框体Box 多了滚动条，可以显示更多东西。

(2) 添加结构体

使用 框体Box 的以下后缀函数可以在 框体中添加 滚动框体ScrollBox。

AddScrollBox ()	在框体中新建一个滚动框体
-----------------	--------------

(3) 结构体成员

该结构体派生自 框体Box 结构体。

玩家可以在 框体 Box [结构体] 查看 框体Box 结构体的详细信息。

ScrollBox滚动框体结构体 成员列表:

结构体成员	成员类型	读写性	说明
Box 结构的成员			派生自此类型，拥有其全部成员
HAwlays	Boolean	读写	是否总是显示水平滚动条
VAwlays	Boolean	读写	是否总是显示垂直滚动条
Position	Vector	读写	滚动条定位，矢量的X值代表水平像素位置 Y值代表垂直像素位置，Z值无意义
.....			

9.8.6

9.8.6 文本 Label [结构体]

===== 点击以返回目录 =====



(1) 添加结构体

使用 框体Box 的以下后缀函数可以在 框体中添加 文本 Label。

AddLabel (str)	在框体中新建一个文本
----------------	------------

(2) 结构体成员

该结构体派生自 控件 Widget 结构体。

玩家可以在 控件 Widget [结构体] 查看 控件 Widget 结构体的详细信息。

Label文本结构体 成员列表:

结构体成员	成员类型	读写性	说明
Widget结构的成员			派生自此类型，拥有其全部成员
Text	String	读写	文本控件要显示的文字
Image	String	读写	文本可以以图像作为背景，Image为图像地址。图像必须存在 Volume0 中， 默认为 .png 格式，推荐用 png 图片
ToolTip	String	读写	鼠标移到文本上会出现的提示文字
.....			

(3) 富文本

kOS 支持如下富文本功能，

玩家只需在 Text 中添加一些类似 html 的标记即可实现：

```
//粗体
set mylabel1:text to "This is <b>important</b>".

//斜体
set mylabel2:text to "This is <i>important</i>".

//字号
set mylabel3:text to "This is <size=30>important</size>".

//预设颜色
set mylabel4:text to "This is <color=orange>important</color>".

//RGBA颜色
set mylabel5:text to "This is <color=#ffaa00FF>important</color>".
set mylabel6:text to "This is <color=#ffaa0080>important</color>".
```

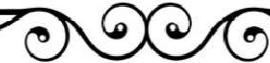
富文本功能是默认开启的，玩家可以用如下语句将其禁用。

```
set mylabel:style:richtext to false.
```

9.8.7

9.8.7 提示显示 TipDisplay [结构体]

===== 点击以返回目录 =====



(1) 添加结构体

使用 框体Box 的以下后缀函数可以在 框体中添加 提示显示 TipDisplay。

添加的位置最好是窗体内容的顶部或底部。

提示显示 TipDisplay 是一个有固定式样 Style 的文本 Label 结构体，专为其他 GUI 控件显示 ToolTip 文本提供空间。

AddTipDisplay ()	在框体中新建一个提示信息
------------------	--------------

(2) 结构体成员

该结构体派生自文本Label 结构体。

玩家可以在文本 Label [结构体] 查看文本Label 结构体的详细信息。

TipDisplay 提示显示 结构体 成员列表:

结构体成员	成员类型	读写性	说明
Label 结构的成员			派生自此类型，拥有其全部成员
.....			

(3) 实际例子

下文在窗口底部创建一个 提示显示 TipDisplay 区域，如果鼠标悬停到其他GUI控件上，提示显示 TipDisplay 区域就会显示该控件的提示文本（ToolTip）。

```
local done is false.
local g is gui(200).
local tiptext is g:addtipdisplay().

local buttonsBox is g:adddbbox().

local b1 is buttonsBox:adbutton("low").
set b1:tooltip to "Makes low pitch note.".
set b1:onclick to {getvoice(0):play(note(200,0.5))}.

local b2 is buttonsBox:adbutton("medium").
set b2:tooltip to "Makes medium pitch note.".
set b2:onclick to {getvoice(0):play(note(300,0.5))}.

local b3 is buttonsBox:adbutton("high").
set b3:tooltip to "Makes high pitch note.".
set b3:onclick to {getvoice(0):play(note(400,0.5))}.

local bClose is g:adbutton("Close").
set bClose:tooltip to "Ends program.".
set bClose:onclick to {set done to true.}.

g:show().
wait until done.
g:dispose().
```

9.8.8

9.8.8 按钮 Button [结构体]

===== 点击以返回目录 =====



(1) 概述

按钮Button 派生自 文本Label，在文本的基础上增加了按钮图像作为背景，增加了按钮弹起和按下两种状态，并为按钮的点击动作挂钩了函数指针。

(2) 添加结构体

使用 框体Box 的以下后缀函数可以在 框体中添加 按钮Button。

AddButton (str)	在框体中新建一个可点击按钮 按钮被点击后会立即弹回来
AddCheckBox (str, bool)	在框体中新建一个可切换按钮 按钮有按下和弹起两种状态 对应 True 和 False 两种值 bool参数对应按钮的初始值
AddRadioButton (str, bool)	在框体中新建一个单选按钮 这是个排他的可点击按钮 当单选按钮按下时，框体 内其他按钮都会关闭。

(3) 结构体成员

该结构体派生自 文本Label 结构体。

玩家可以在 文本 Label [结构体] 查看 文本Label 结构体的详细信息。

Button按钮结构体 成员列表：

结构体成员	成员类型	读写性	说明
Label结构的成员			派生自此类型，拥有其全部成员
Pressed	Boolean	读写	按钮是按下 (True) 还是弹起 (False)
TakePress	Boolean	只读	按钮按下 / 弹起状态值，当用户 读取其True值时，复位成False
Toggle	Boolean	读写	False值时，点击按钮后按钮立即弹起 True值时，每次点击按钮按钮都会使 按钮在 弹起 和 按下 状态之间切换
Exclusive	Boolean	读写	这个按钮是排他性的吗 (单选按钮)
OnClick ()	kOSDelegate	读写	点击按钮时所挂钩函数指针
OnToggle	kOSDelegate (Boolean)	读写	按钮状态改变时所挂钩的函数指针 OnToggle会向所调用函数传递一个 按钮状态的Boolean值，相应的，所 调用函数需以Boolean值作为参数
.....			

(4) 实际例子

OnClick 的例子：

```
set mybutton:ONCLICK to { print "Do something here.". }.
```

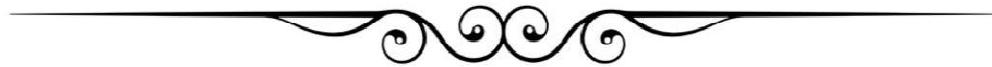
OnToggle 的例子：

```
mybutton:ONTOGGLE to { parameter val. print "Button value just became "
```

9.8.9

9.8.9 下拉菜单 PopupMenu [结构体]

===== 点击以返回目录 =====



(1) 添加结构体

使用 框体Box 的以下后缀函数可以在 框体中添加下拉菜单 PopupMenu。

AddPopupMenu ()	在框体中新建一个下拉菜单
------------------------	--------------

(2) 结构体成员

该结构体派生自 按钮Button 结构体。

玩家可以在 按钮 Button [结构体] 查看 按钮Button 结构体的详细信息。

PopupMenu下拉菜单结构体 成员列表:

结构体成员	成员类型	读写性	说明
Button 结构的成员			派生自此类型，拥有其全部成员
Options	List	读写	选项的列表，选项可以是任意结构
OptionSuffix	String	读写	在下拉菜单中显示选项的哪个或后缀 默认值为 "ToString" <small>转换为字符串</small>
AddOption (value)	无返回值	函数	在选项列表尾部添加一项value
Value	不定	读写	列表当前所选的项，如果未选择过则 为空字符串 ""
Index	Scalar	读写	当前所选是第几个选项
Changed	Boolean	读写	用户是否有选择过选项，当用户 读取其True值时，复位成False
OnChange	kOSDelegate (string)	读写	变更选项时所挂钩的函数指针 OnChange 会向所调用函数传递所选项 的String值，相应的，所 调用函数需以String值作为参数
Clear ()	无返回值	函数	清空所有选项
MaxVisible	Scalar	读写	下拉菜单中对多显示多少个选项 如果实际选项更多，就用上滚动条 默认值是 15
.			

(3) 使用示例

```
local popup is gui:addpopupmenu().

// Make the popup display the Body:NAME's instead of the Body:TOSTRING's:
set popup:OPTIONSUFFIX to "NAME".

list bodies in bodies.
for planet in bodies {
    if planet:hasbody and planet:body = Sun {
        popup:adoption(planet).
    }
}
set popup:value to body.
```

9.8.10

9.8.10 文本框 TextField [结构体]

===== 点击以返回目录 =====



(1) 概述

文本框TextField 也是文本Label 派生出的控件结构体。

文本框增加了允许用户修改文本的功能。

(2) 添加结构体

使用 框体Box 的以下后缀函数可以在 框体中添加 文本框TextField。

AddTextField (str)	在框体中新建一个文本框
--------------------	-------------

(3) 结构体成员

该结构体派生自 文本Label 结构体。

玩家可以在 文本 Label [结构体] 查看 文本Label 结构体的详细信息。

TextField文本框结构体 成员列表:

结构体成员	成员类型	读写性	说明
Label 结构的成员			派生自此类型，拥有其全部成员
Changed	Boolean	读写	用户是否修改过文本框，当用户读取其True值时，复位成False
OnChange	kOSDelegate<(string)>	读写	文本框修改时所挂钩的函数指针 OnChange会向所调用函数传递文本框内容的String值，相应的，所调用函数需以String值作为参数
Confirmed	Boolean	读写	用户是否回车确认过文本框，点击其他地方对该文本框失焦也算的当用户读取其True值时，复位成False
OnConfirmed	kOSDelegate<(string)>	读写	按回车确认文本框时挂钩的函数指针 OnConfirmed会向所调用函数传递文本框内容的String值，相应的，所调用函数需以String值作为参数
ToolTip	String	读写	文本框为空时才会出现的提示文字
.....			

(4) 实际例子

OnChange 的例子：

```
set myTextField:ONCHANGE to {parameter str. print "Value is now: " + str.}.
```

OnConfirmed 的例子：

```
set myTextField:ONCONFIRM to {parameter str. print "Value is now: " + str.}.
```

9.8.11

9.8.11 滑块 Slider [结构体]

----- 点击以返回目录 -----



(1) 概述

滑块Slider可以拉来拉去，来方便的设置实数值。

(2) 添加结构体

使用 框体Box 的以下后缀函数可以在 框体中添加 滑块Slider。

AddHSlider (init, min, max)	在框体中新建一个水平滑块 (初值, 最小值, 最大值)
AddVSlider (init, min, max)	在框体中新建一个垂直滑块 (初值, 最小值, 最大值)

(3) 结构体成员

该结构体派生自 控件 Widget 结构体。

玩家可以在 控件 Widget [结构体] 查看 控件 Widget 结构体的详细信息。

Slider滑块结构体 成员列表:

结构体成员	成员类型	读写性	说明
Widget结构的成员			派生自此类型，拥有其全部成员
Value	Scalar	读写	滑块的当前值
OnChange	kOSDelegate (Scalar)	读写	滑块修改时所挂钩的函数指针 OnChange会向所调用函数传递 滑块的Scalar值，相应的，所 调用函数需以Scalar值作为参数
Min	Scalar	读写	滑块拖拉的最小值
Max	Scalar	读写	滑块拖拉的最大值
.....			

9.8.12

9.8.12 空档 Spacing [结构体]

===== 点击以返回目录 =====



(1) 概述

空档Spacing是为了给排版留白而设计的。

(2) 添加结构体

使用框体Box 的以下后缀函数可以在框体中添加 空档Spacing。

AddSpacing (PixelSize)	在框体中新建一个空白区域
------------------------	--------------

(3) 结构体成员

该结构体派生自 控件 Widget 结构体。

玩家可以在 控件 Widget [结构体] 查看 控件 Widget 结构体的详细信息。

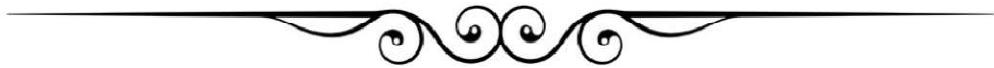
Spacing空档结构体 成员列表:

结构体成员	成员类型	读写性	说明
Widget结构的成员			派生自此类型，拥有其全部成员
Amount	Scalar	读写	在水平布局的框体Box中，表示要在 水平方向间隔多少像素； 在垂直布局的框体Box中，表示要在 垂直方向间隔多少像素；
.....			

9.8.13

9.8.13 样式 Style [结构体]

===== 点击以返回目录 =====



(1) 概述

样式Style是每一个控件Widget都带有的后缀成员，决定了控件的外观风格。

(2) 结构体成员

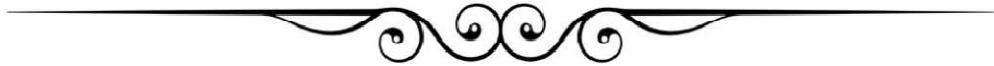
Style样式结构体 成员列表：

结构体成员	成员类型	读写性	说明
HStretch	Boolean	读写	控件是否水平拉伸
VStretch	Boolean	读写	控件是否垂直拉伸
Width	Scalar (pixels)	读写	控件宽度
Height	Scalar (pixels)	读写	控件高度
Margin	StyleRectOffset	读写	控件边距
Padding	StyleRectOffset	读写	控件内部和控件边框的间距
Border	StyleRectOffset	读写	控件的背景图片拉伸时的固定边距
Overflow	StyleRectOffset	读写	控件的背景图片超出控件边缘的举例
Align	String	读写	对齐方式, "Center", "Left", "Right" [居中] [左] [右] 仅对文本类和按钮类控件有效, 仅在设置了HStretch或Width时有效
Font	String	读写	字体。注意可通过预设 列表Fonts查看所有支持的字体
FontSize	Scalar	读写	字体大小
RichText	Boolean	读写	是否开启富文本
Normal	StyleState	读写	正常状态下的属性
On	StyleState	读写	按下(on)状态下的属性
Normal_On	StyleState	读写	同上
Hover	StyleState	读写	鼠标聚焦状态下的属性
Hover_On	StyleState	读写	鼠标聚焦并且按下(on)状态下的属性
Active	StyleState	读写	活动状态下的属性
Active_On	StyleState	读写	活动并且按下(on)状态下的属性
Focused	StyleState	读写	键盘聚焦状态下的属性
Focused_On	StyleState	读写	键盘聚焦并且按下(on)状态下的属性
BG	String	读写	背景图片, 和:Normal:BG相同
TextColor	Color	读写	文字颜色, 和:Normal:TextColor相同[转换为普通表达式].....

*注1：有关 Style:Border 与 控件背景图片的拉伸模式，请访问 状态样式 StyleState [结构体]

9.8.14 边距 StyleRectOffset [结构体]

===== 点击以返回目录 =====



(1) 概述

边距 StyleRectOffset 是 样式Style 的后缀成员，
将有关控件 Widget 的边缘留白的相关设置打包在了一起。

(2) 结构体成员

StyleRectOffset 边距结构体 成员列表：

结构体成员	成员类型	读写性	说明
Left	Scalar	读写	控件左侧的边距, 像素数
Right	Scalar	读写	控件右侧的边距, 像素数
Top	Scalar	读写	控件上侧的边距, 像素数
Bottom	Scalar	读写	控件下侧的边距, 像素数
H	Scalar	读写	同时设置控件左右的边距, 像素数
V	Scalar	读写	同时设置控件上下的边距, 像素数
.....			

9.8.15

9.8.15 状态样式 StyleState [结构体]

===== 点击以返回目录 =====



(1) 概述

状态样式 StyleState 是 样式Style 的后缀成员，
将有关控件 Widget 在不同操作状态下的背景和字体颜色设置打包在了一起。

(2) 结构体成员

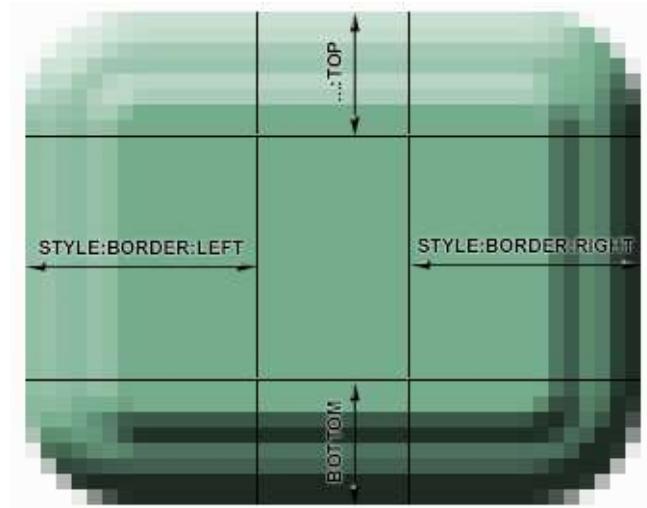
StyleState 状态样式结构体 成员列表：

结构体成员	成员类型	读写性	说明
BG	String	读写	控件的背景图片
TextColor	Color	读写	控件内部文字的颜色
.....			

(3) 控件背景图片

控件所用的背景图片，必须存储在 Volume 0 (Archive) 上。

控件的背景图片 StyleState:BG 采用一种叫 9-slice image 的显示模式。
如下图，他根据 Style:Border 的边距设置，将图片分成9个区域。
当控件进行拉伸时，只有最中间的区域进行完整的拉伸（垂直和水平方向），
而周围其他的区域则是配合中间进行不完全拉伸（不同时在垂直和水平方向）。
这样就能让一个具有边缘特征的图像随意拉伸而不显得突兀了。



如上图，箭头所指的边距由Style:Border 确定，在整个拉伸过程中尺寸不变。

9.8.16

9.8.16 皮肤 Skin [结构体]

===== 点击以返回目录 =====



(1) 概述

皮肤Skin 是窗体GUI 的后缀成员。

本质上是一个 样式Style 的打包方案，作用于窗体以内的所有 控件Widget。

(2) 结构体成员

Skin皮肤结构体 成员列表：

结构体成员	成员类型	读写性	说明
Box	Style	读写	所有框体的样式
Button	Style	读写	所有按钮的样式
HorizontalScrollbar	Style	读写	所有滚动框体里水平滚动条的样式
HorizontalScrollbarLeftButton	Style	读写	所有滚动框体里水平滚动条左侧按钮的样式
HorizontalScrollbarRightButton	Style	读写	所有滚动框体里水平滚动条右侧按钮的样式
HorizontalScrollbarThumb	Style	读写	所有滚动框体里水平滚动条的游标的样式
HorizontalSlider	Style	读写	所有水平滑块的样式
HorizontalSliderThumb	Style	读写	所有水平滑块里游标的样式
VerticalScrollbar	Style	读写	所有滚动框体里垂直滚动条的样式
VerticalScrollbarLeftButton	Style	读写	所有滚动框体里垂直滚动条左侧按钮的样式
VerticalScrollbarRightButton	Style	读写	所有滚动框体里垂直滚动条右侧按钮的样式
VerticalScrollbarThumb	Style	读写	所有滚动框体里垂直滚动条的游标的样式
VerticalSlider	Style	读写	所有垂直滑块的样式
VerticalSliderThumb	Style	读写	所有初值滑块里游标的样式
Label	Style	读写	所有文本控件的样式
ScrollView	Style	读写	所有滚动框体的样式
TextField	Style	读写	所有文本框的样式
Toggle	Style	读写	所有可切换按钮的样式 (用GUI下AddCheckButton和AddRadioButton添加)
FlatLayout	Style	读写	所有透明框体的样式 (用GUI下AddHLayout和AddVLayout添加)
PopupMenu	Style	读写	所有下拉菜单的样式
PopupMenuItem	Style	读写	所有下拉菜单选项的样式
LabelTipOverlay	Style	读写	所有控件提示文本的样式 (鼠标移上去弹出的文字)
Window	Style	读写	所有窗体的样式
Font	String	读写	文字的字体。注意可通过预设列表Fonts查看所有支持的字体
SelectionColor	Color	读写	选定文本时候的背景颜色
Add (name, style)	Style	函数	再皮肤里新建样式
Has (name)	Boolean	函数	皮肤里有没有这个样式
Get (name)	Style	函数	调取皮肤里的样式
.....			

9.9

9.9 编程与配置

9.9.1

9.9.1 数值 Scalar [结构体]

===== 点击以返回目录 =====



详见 数值 Scalar [结构体]。

9.9.2

9.9.2 逻辑值 Boolean [结构体]

===== 点击以返回目录 =====



(1) 结构体成员

该结构体派生自 结构体 Structure 结构体。

玩家可以在 结构体 Structure [结构体] 查看 结构体 Structure 结构体的详细信息。

Boolean逻辑值结构体 成员列表：

结构体成员	成员类型	读写性	说明
Structure 结构的成员			派生自此类型，拥有其全部成员
.....			

(2) 逻辑运算符

与 或 非
and or not

9.9.3

9.9.3 字符串 String [结构体]

===== 点击以返回目录 =====



(1) 创建结构体

从零开始创建字符串的方法如下。

```
// Create a new string
SET s TO "Hello, Strings!".
```

另外玩家还可以从其他结构体或者函数来获取/创建结构体。

(2) 访问字符串

字符串**String** 就是字符的数组，一切可以对列表**List** 用的操作都可以用在字符串。

例如下面：

```
LOCAL str is "abcde".
local index is 0.
until index = str:LENGTH {
    print str[index].
    set index to index + 1.
}
```

(3) 比较字符串

两个字符串可以用 =, <, >, <, >=, <= 符号进行相应比较。

比较的时候把两个字符串左端对齐，然后从左开始依次比较相同位置字符的 ASCII 码。

请注意，kOS 对字符串是不区分大小写的。

还有要注意一下数值和数值的字符串是不同的，如下：

```
print (1234 < 99). // prints "False"
print ("1234" < 99). // prints "True"
```

(4) 结构体成员

String 字符串结构体 成员列表：

结构体成员	成员类型	读写性	说明
<code>Contains (str)</code>	<code>Boolean</code>	函数	该字符串中是否包含 <code>str</code>
<code>StartSwith (str)</code>	<code>Boolean</code>	函数	该字符串是否以 <code>str</code> 开头
<code>EndSwith (str)</code>	<code>Boolean</code>	函数	该字符串是否以 <code>str</code> 结尾
<code>Find (str)</code>	<code>Scalar</code>	函数	<code>str</code> 字符串的首次出现位置
<code>IndexOf (str)</code>	<code>Scalar</code>	函数	同上
<code>FindAt (str, n)</code>	<code>Scalar</code>	函数	从第 <code>n</code> 位置开始往后数的 <code>str</code> 字符串的首次出现位置
<code>FindLast (str)</code>	<code>Scalar</code>	函数	<code>str</code> 字符串最后一次出现的位置
<code>LastIndexOf (str)</code>	<code>Scalar</code>	函数	同上
<code>FindLastAt (str, n)</code>	<code>Scalar</code>	函数	从第 <code>n</code> 位置开始往后数的 <code>str</code> 字符串的最后一次出现位置
<code>Insert (n, str)</code>	<code>String</code>	函数	在该字符串的 <code>n</code> 位置插入 <code>str</code>
<code>Iterator</code>	<code>Iterator</code>	只读	字符串的数组指针
<code>Length</code>	<code>Scalar</code>	只读	字符串的长度
<code>MatchSpattern (pattern)</code>	<code>Boolean</code>	函数	字符串是否符合正则表达式 <code>pattern</code> 的规则
<code>PadLeft (width)</code>	<code>String</code>	函数	将该字符串在 <code>width</code> 长度的空间内靠右对齐，不足的位置用空格填补
<code>PadRight (width)</code>	<code>String</code>	函数	将该字符串在 <code>width</code> 长度的空间内靠左对齐，不足的位置用空格填补
<code>Remove (index, count)</code>	<code>String</code>	函数	字符串剔除从 <code>index</code> 到 <code>index + count</code> 位置的字符，之后的样子
<code>Replace (oldstr, newstr)</code>	<code>String</code>	函数	把字符串中的 <code>oldstr</code> 替换为 <code>newstr</code>
<code>Split (separator)</code>	<code>String</code>列表	函数	以字符<code>separator</code>为边界裁分字符串返回<code>String</code>的List
<code>SubString (start, count)</code>	<code>String</code>	函数	该字符串中从 <code>index</code> 到 <code>index + count</code> 位置的字符，形成一个新字符串
<code>ToLower</code>	<code>String</code>	只读	该字符串的全小写形式
<code>ToUpper</code>	<code>String</code>	只读	该字符串的全大写形式
<code>Trim</code>	<code>String</code>	只读	去除字符串中空格后的样子
<code>TrimEnd</code>	<code>String</code>	只读	去除字符串尾部空格后的样子
<code>TrimStart</code>	<code>String</code>	只读	去除字符串头部空格后的样子
<code>ToNumber ()</code> <code>ToNumber (Err)</code>	<code>Scalar</code>	函数	把数值的字符串转换成数值，如果字符串内容不是数值则返回 <code>Err</code> 数值
<code>ToScalar ()</code> <code>ToScalar (Err)</code>	<code>Scalar</code>	函数	同上
<code>.....</code>			

*注1：这些成员函数的作用是产生新的字符串，而不是更改原字符串的值。

(5) 实际例子

```
set s to "abcdefg".
print s. //abcdefg
print s:Insert(3,"456"). //abc456defg
set s to "abc".
print s:PadLeft(5). //    abc
```

9.9.4

9.9.4 色彩 Colors [结构体]

===== 点击以返回目录 =====



kOS提供预设的颜色称呼，也提供带有Alpha通道的GBA色和HSV色函数供使用。

(1) 预设颜色

以下为 kOS 中的预设颜色，他们都是预设变量，他们都对应一种色彩的结构体。

```
RED //红色
GREEN //绿色
BLUE //蓝色
YELLOW //黄色
CYAN //蓝绿色
MAGENTA //品红
PURPLE //紫色
WHITE //白色
BLACK //黑色
```

(2) 创建结构体

除此之外，玩家还可以用颜色函数来创建色彩结构体。

创建RGB结构体

```
RGB(r,g,b) //创建不透明的RGBA颜色，参数范围0~1
RGB(r,g,b,a) //创建RGBA颜色，参数范围0~1
```

创建HSVA结构体

```
HSV(h,s,v) //创建不透明的HSVA颜色，参数范围0~1
HSVA(h,s,v,a) //创建HSVA颜色，参数范围0~1
```

(3) 结构体成员

RGB色彩结构体 成员列表：

结构体成员	成员类型	读写性	说明
: R or : Red	Scalar	只读	红原色, 范围 0~1
: G or : Green	Scalar	只读	绿原色, 范围 0~1
: B or : Blue	Scalar	只读	蓝原色, 范围 0~1
: A or : Alpha	Scalar	只读	不透明度, 范围 0~1
: Html or : Hex	String	只读	十六进制表示的不带不透明度的颜色 比如 # ff0000 表示纯红色
.....			

HSVA色彩结构体 成员列表:

结构体成员	成员类型	读写性	说明
: H or : Hue	Scalar	只读	色调, 范围 0~360
: S or : Saturation	Scalar	只读	饱和度, 范围 0~1
: V or : ValueALU	Scalar	只读	亮度, 范围 0~1
: A or : Alpha	Scalar	只读	不透明度, 范围 0~1
.....			

(4) 实际例子

RGBA颜色的例子:

```

SET myarrow TO VECDRAW.
SET myarrow:VEC to V(10,10,10).
SET myarrow:COLOR to YELLOW.
SET mycolor TO YELLOW.
SET myarrow:COLOR to mycolor.
SET myarrow:COLOR to RGB(1.0,1.0,0.0).

// COLOUR spelling works too
SET myarrow:COLOUR to RGB(1.0,1.0,0.0).

// half transparent yellow.
SET myarrow:COLOR to RGBA(1.0,1.0,0.0,0.5).

PRINT GREEN:HTML. // prints #00ff00

```

HSVA颜色的例子

```

SET myarrow TO VECDRAW.
SET myarrow:VEC to V(10,10,10).
SET myarrow:COLOR to HSV(60,1,1). // Yellow
SET myarrow:COLOR:S to 0.5. // Light yellow
SET myarrow:COLOR:H to 0. // pink

```

9.9.5 函数指针 KOSDelegate [结构体]

===== 点击以返回目录 =====



(1) 概述

函数指针KOSDelegate 可以用来挂钩函数，
挂钩了函数之后就能把变量名当函数名来用。
除此之外，函数指针还可以作为函数参数传递给函数。

(2) 使用函数指针

用法：

函数名@

例子1：

```
Function myfunc {
    parameter a,b.
    return a + b.
}

print myfunc(1, 2).
//函数名之后加个@符号就代表取这个函数的指针了
set aaa to myfunc@.
//调用函数的方式：
print aaa:call(1, 2).
print aaa(1,2). //最后两句一个意思
```

例子2：

```
function myfunc { print "hello, there". }
local print_a_thing is myfunc@.
print_a_thing().
print_a_thing:Call() //最后两句干的是同样的事
```

*注1： KOSDelegate 函数指针不适用于结构体内函数。

(3) 结构体成员

KOSDelegate函数指针结构体 成员列表：

结构体成员	成员类型	读写性	说明
Call (args)	不定	函数	调用函数指针所指的函数
Bind (args)	KOSDelegate	函数	为靠左的参数赋值，然后把剩下的 打包成一个新的函数指针
IsDead	Boolean	只读	检查函数指针是否失效
.....			

(4) DoNothing 关键词

DoNothing 表示一个空内容的函数。
玩家如果不需要用用函数指针了，就可以把函数指针挂钩到 DoNothing，
这样可以减小计算量。

```
//假设 a 是一个KOSDelegate结构
Set a to DoNothing.
```

9.9.6

9.9.6 PID循环 PIDLoop [结构体]

===== 点击以返回目录 =====



(1) 概述

PID循环 PIDLoop 是 kOS 提供的PID控制器。

玩家游戏时不需要自己写PIS控制算法，可以直接使用 PID循环。

并且使用 PID循环 还能节省指令数，对于复杂程序编程很有帮助。

(2) 使用PID循环结构体

```
Set KP to .....
Set KI to .....
Set KD to ..... //设置PID参数
Set TargetV to ..... //目标值，例如火箭悬停的话就是悬停的目标高度
SET PID TO PIDLOOP(KP, KI, KD, MINOUTPUT, MAXOUTPUT).
//创建一个PIDLoop结构，输入PID参数，并约束输出值的范围
//例如火箭悬停的话就是节流阀百分比
set PID:setpoint to TargetV //为这个PIDLoop设置目标值TargetV
until false {
    SET OUT TO PID:UPDATE(TIME:SECONDS, Status).
    //每一帧都更新，并提供当前时间和要控制的变量Status
    //PID会输出OUT，例如火箭悬停的话就是节流阀百分比
    Wait 0.001. //等待下一个物理帧
}
```

(3) 结构体成员

PIDLoop PID循环结构体 成员列表：

结构体成员	成员类型	读写性	说明
<code>LastSampleTime</code>	<code>Scalar</code>	只读	上次采样时间, 算微分项要用
<code>Kp</code>	<code>Scalar</code>	读写	比例项系数
<code>Ki</code>	<code>Scalar</code>	读写	积分项系数
<code>Kd</code>	<code>Scalar</code>	读写	微分项系数
<code>Input</code>	<code>Scalar</code>	只读	上次的输入值
<code>SetPoint</code>	<code>Scalar</code>	读写	当前目标值
<code>Error</code>	<code>Scalar</code>	只读	上次的误差值, 算比例项要用
<code>Output</code>	<code>Scalar</code>	只读	上次的输出值
<code>MaxOutput</code>	<code>Scalar</code>	读写	输出值上限
<code>MinOutput</code>	<code>Scalar</code>	读写	输出值下限
<code>ErrorSum</code>	<code>Scalar</code>	只读	误差的对时间积分, 算积分项要用
<code>PTerm</code>	<code>Scalar</code>	只读	比例项的输出
<code>ITerm</code>	<code>Scalar</code>	只读	积分项的输出
<code>DTerm</code>	<code>Scalar</code>	只读	微分项的输出
<code>ChangeRate</code>	<code>Scalar (次 / s)</code>	只读	上次的采样频率
<code>Reset ()</code>	无返回值	函数	重设积分项
<code>Update (time, input)</code> [更新]	<code>Scalar</code>	函数	更新一次PID循环 PIDLoop的计算要靠这个函数推动
.....			

(4) PID循环的内部算法

下面是 PID循环 在运行时等效的算法:

```

// assume that the terms LastSampleTime, Kp, Ki, Kd, Setpoint, MinOutput, and MaxOutput are p
defined
declare function Update {
    declare parameter sampleTime, input.
    set Error to Setpoint - input.
    set PTerm to error * Kp.
    set ITerm to 0.×
    set DTerm to 0.×
    if (LastSampleTime < sampleTime) {
        set dt to sampleTime - LastSampleTime.
        if dt < 1 {
            // only calculate integral and derivative if the time
            // difference is less than one second, and their gain is not 0.×
            if Ki <> 0 {
                set ITerm to (ErrorSum + Error) * dt * Ki.
            }×
            set ChangeRate to (input - LastInput) / dt.
            if Kd <> 0 {
                set DTerm to ChangeRate * Kd.
            }
        }
    }×
    set Output to pTerm + iTerm + dTerm.
    // if the output goes beyond the max/min limits, reset it and adjust ITerm.
    if Output > MaxOutput {
        set Output to MaxOutput.
        // adjust the value of ITerm as well to prevent the value
        // from winding up out of control.
        if (Ki <> 0) and (LastSampleTime < sampleTime) {
            set ITerm to Output - Pterm - DTerm.
        }
    }×
    else if Output < MinOutput {
        set Output to MinOutput.
        // adjust the value of ITerm as well to prevent the value
        // from winding up out of control.
        if (Ki <> 0) and (LastSampleTime < sampleTime) {
            set ITerm to Output - Pterm - DTerm.
        }
    }×
    set LastSampleTime to sampleTime.
    if Ki <> 0 set ErrorSum to ITerm / Ki.
    else set ErrorSum to 0.×
    return Output.
}

```

9.9.7 箭头绘制 VecDraw [结构体]

===== 点击以返回目录 =====



(1) 概述

kOS 允许玩家在游戏画面上画辅助箭头，箭头绘制VecDraw 就对应的是辅助箭头。

(2) 创建结构体

玩家可以使用如下函数来创建 箭头绘制VecDraw。

```
Set a to VecDraw(). //创建默认箭头
Set a to VecDrawargs(). //同上
//其中参数即为VecDraw结构成员名
Set a to VecDraw(start,vec,color,label,scale,show,width,pointy).
Set a to VecDrawargs(start,vec,color,label,scale,show,width,pointy). //同上
```

玩家可以通过如下函数指令清除 箭头绘制VecDraw。

```
ClearVecDraws(). //清除所有VecDraw结构体
```

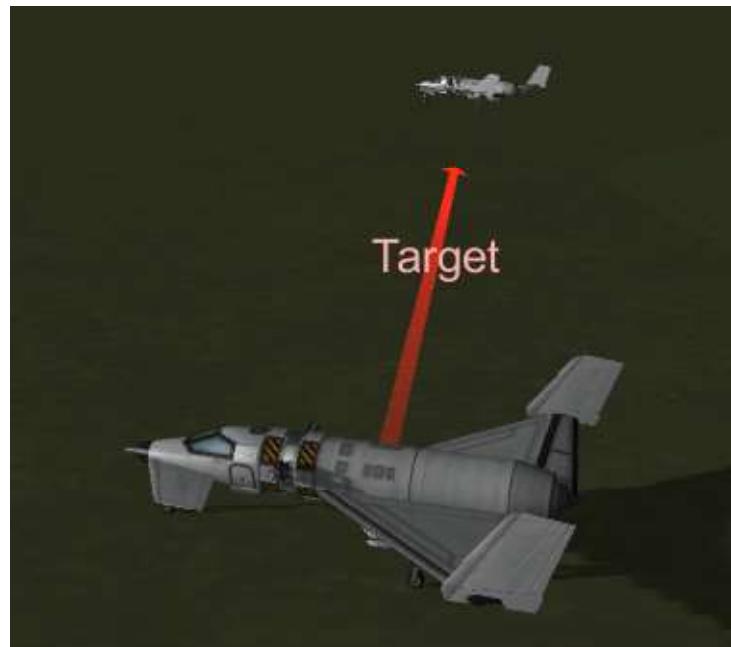
(3) 结构体成员

VecDraw箭头绘制结构体 成员列表：

结构体成员	成员类型	读写性	说明	默认值
Start	Vector	读写	箭头的起点 (Ship - Raw坐标系)	V (0, 0, 0)
Vec	Vector	读写	要画的箭头 (Ship - Raw坐标系)	V (0, 0, 0)
Color	Color	读写	箭头的颜色	White
Colour		读写	同上	
Label	String	读写	文字标记	空字符串
Scale	integer	读写	箭头大小	1.0
Show	Boolean	读写	是否显示	False
Width	数值	读写	箭头的宽度，连同文字字号一起缩放	0.2
Pointy	Boolean	读写	是否有箭头	True
Wiping	Boolean	读写	是否擦除	True
StartUpdater	kOSDelegate	读写	实时刷新Start矢量，所挂钩的函数必须返回Vector类型的值。如果不用了就设为DoNothing	
VecUpdater	kOSDelegate	读写	实时刷新Vec矢量所挂钩的函数必须返回Vector类型的值。如果不用了就设为DoNothing	
VectorUpdater	kOSDelegate	读写	同上	
ColorUpdater	kOSDelegate	读写	实时刷新Color矢量所挂钩的函数必须返回Color类型的值。如果不用了就设为DoNothing	
ColourUpdater	kOSDelegate	读写	同上	
.....				

(4) 实际例子

下图是展示在屏幕里显示从当前载具动态指向目标载具箭头的画面。



矢量箭头还可以设置为隐藏箭头只显示文字，例如下图的锁定框。



像下面两段代码一样，通过 VecDraw / VecDrawAges 矢量箭头，
可以实现这样的效果：

```
Set a to Vessel("www"). //a是www载具
Set b to VecDraw(V(0,0,0) , a:Position , RED , "Target" , 0.5 , True , 0.8).
Set b:VecUpdater to {Return a:Position.} //实时更新a的位置
Wait Until False. //程序不停止
```

```
Set a to Vessel("www"). //a是www载具
Set b to VecDraw(V(0,0,0) , a:Position , RGB(0,1,0) , "[+]" , 2 , True , 0).
Set b:VecUpdater to {Return a:Position}. //实时更新a的位置
Wait Until False. //程序不停止
```

另外，因为 VecDraw 下的函数指针默认值是 DoNothing，但是是实时更新的，
放着就会产生计算量，如果不需要用了就要设为 DoNothing

```
Set b:VecUpdater to DoNothing.
```

另外，创建矢量箭头时也可以实用匿名函数作动态参数。如下：

```
// Small dynamically moving vecdraw example:
SET anArrow TO VECDRAW(
  { return (6-4*cos(100*time:seconds)) * up:vector. },
  { return (4*sin(100*time:seconds)) * up:vector. },
  { return RGBA(1, 1, RANDOM(), 1). },
  "Jumping arrow!",
  1.0,
  TRUE,
  0.2,
  TRUE,
  TRUE
).
wait 20. // Give user time to see it in motion.
set anArrow:show to false. // Make it stop drawing.
```

9.9.8

9.9.8 部件高亮 HighLight [结构体]

===== 点击以返回目录 =====



(1) 概述

部件高亮 HighLight 可以控制载具上每个部件的高亮显示。

(2) 创建结构体

玩家可以使用 HighLight 函数创建结构体

```
HighLight(part,color) //参数为要高亮的部件和颜色
```

(3) 结构体成员

HighLight 部件高亮结构体 成员列表：

结构体成员	成员类型	读写性	说明
Color	Color	读写	高亮颜色
Enabled	Boolean	读写	是否启动部件高亮， 默认为 True
.....

(4) 实际例子

```
list elements in elist.
// Color the first element pink
SET foo TO HIGHLIGHT( elist[0], HSV(350,0.25,1) ).
// Turn the highlight off
SET foo:ENABLED TO FALSE
// Turn the highlight back on
SET foo:ENABLED TO TRUE
```

部件高亮的效果是这样的：



9.9.9

9.9.9 kOS 配置 Config [结构体]

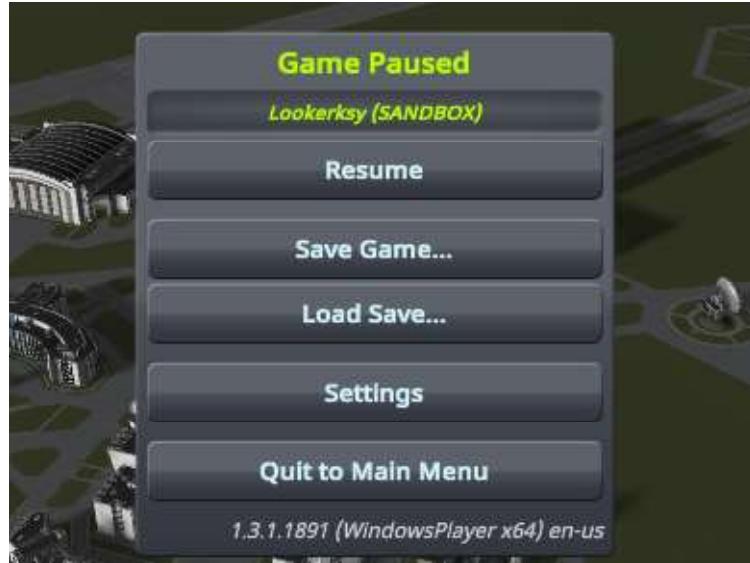
===== 点击以返回目录 =====



(1) 概述

kOS 配置 Config 对应里 kOS 的设置内容。

玩家可以直接在 Config 结构体里 查看和更改这些设置内容。





(2) 获取结构体

通过 预设变量Config即可获取。

```
Print Config:Ipu.
```

(3) 结构体成员

Config kOS配置列表结构体 成员列表:

结构体成员	成员类型	读写性	说明	默认值
Ipu	Scalar	读写	每个物理帧的最大指令数，范围 50~2000	150
Ucp	Boolean	读写	是否使用压缩Volume中的文件优点是文件体积会变小，缺点是会变慢	False
Stat	Boolean	读写	是否显示kOS程序运行的状态信息，如果设为True就可以用ProfileResult函数了	False
RT	Boolean	读写	是否与通讯MOD进行交互	False
Arch	Boolean	读写	是否将默认存储空间从Volume1改为Volume0	False
ObeyHideUi	Boolean	读写	指令窗口是否受F2键(隐藏UI)的控制	True
Safe	Boolean	读写	是否对空值Nan和溢出值Infinity提示报错	False
Audioerr	Boolean	读写	报错时是否使用蜂鸣器	False
Verbose	Boolean	读写	是否显示报错的详细信息	False
Telnet	Boolean	读写	是否启用远程服务器	False
Tport	integer	读写	远程服务器的端口	5410
IpAddress	String	读写	远程服务器的IP地址	"127.0.0.1"
Btightness	Scalar	读写	指令窗口亮度	0.7 ∈ [0, 1]
DefaultFontSize	Scalar	读写	指令窗口里的文字大小	12 ∈ [6, 20] 整数
DefaultWidth	Scalar	读写	指令窗口亮度的默认宽度	50 ∈ [15, 255] 整数
DefaultHeight	Scalar	读写	指令窗口亮度的默认高度	36 ∈ [3, 160] 整数
DebugEachOPCode	Boolean	读写	是否在运行时同步生成运行报告，会很卡	False
.....				

9.10

9.10 杂项

9.10.1

9.10.1 第四墙 KUniverse [结构体]

===== 点击以返回目录 =====



(1) 概述

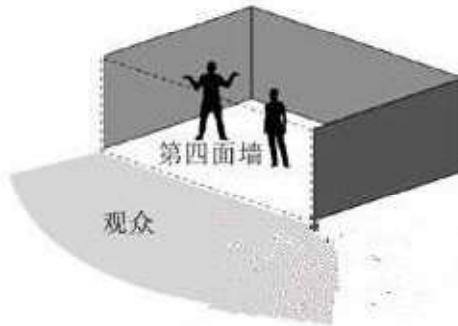
第四墙 KUniverse 全称是 KUniverse 4th wall methods,

对应的是游戏在 Esc 菜单里的操作。

玩家可以从 预设变量KUniverse 来获取该结构体。



4th wall 的字面翻译是第四墙，这是一个戏剧方面的术语，指一面在传统三壁镜框式舞台中虚构的“墙”。它可以让观众看见戏剧中的观众。



(2) 结构体成员

KUniverse第四墙结构体 成员列表:

结构体成员	成员类型	读写性	说明
<code>CanRevert</code>	<code>Boolean</code>	只读	是否能返回
<code>CanRevertToLaunch</code>	<code>Boolean</code>	只读	是否能返回到发射台
<code>CanRevertToEditor</code>	<code>Boolean</code>	只读	是否能返回到车间
<code>RevertToLaunch ()</code>	无返回值	函数	返回到发射台
<code>RevertToEditor ()</code>	无返回值	函数	返回到车间
<code>RevertTo (name)</code>	<code>String</code>	函数	返回到指定车间 ("VAB" 或 "SPH")
<code>OriginEditor</code>	<code>String</code>	只读	生产车间 ("VAB" 或 "SPH" 或 "" 空值则代表无法返回车间)
<code>Pause ()</code>	无返回值	函数	暂停游戏 (同时也暂停kOS运行) 一定要手动恢复才行 相当于按下Esc键
<code>CanQuickSave</code>	<code>Boolean</code>	只读	是否能快速存档
<code>QuickSave ()</code>	无返回值	函数	快速存档 (F5)
<code>QuickLoad ()</code>	无返回值	函数	快速赌坊 (F9)
<code>QuickSaveTo (name)</code>	<code>String</code>	函数	快速存档到name
<code>QuickLoadFrom (name)</code>	无返回值	函数	从name快速读档
<code>QuickSaveList</code>	<code>String</code> 列表	只读	存档的name列表
<code>HoursPerDay</code>	<code>Scalar</code>	只读	一天有几个小时 (6 or 24)
<code>DebugLog (message)</code>	无返回值	函数	在 KSP 的飞行日志中写字 用于插标测试
<code>DefaultLoadDistance</code>	<code>LoadDistance</code>	只读	下次载入时默认的加载距离
<code>TimeWarp</code>	<code>TimeWarp</code>	只读	时间加速
<code>ActiveVessel</code>	<code>Vessel</code>	读写	设置活动载具, 但自机在大气 中或时间加速时无效
<code>ForceSetActiveVessel (Vessel)</code>	无返回值	函数	强行设置活动载具, 自机在大 气中或者施加加速时也有效, 但可能有载具消失的Bug
<code>ForceActive (Vessel)</code>	无返回值	函数	同上
<code>GetCraft (name, editor)</code>	<code>CraftTemplate</code>	函数	从车间获取name名的载具存档
<code>LaunchCraft (template)</code>	无返回值	函数	发射template载具
<code>LaunchCraftFrom (template, site)</code>	无返回值	函数	在site发射场发射template载具
<code>CraftList ()</code>	<code>CraftTemplate</code> 列表	函数	SPH和VAB里所有载具存档的列表
<code>RealTime</code>	<code>Scalar (s)</code>	只读	真实世界时间, 1970年1月1日起
<hr/>			

(3) 实际例子

```
//DebugLog() 函数生成的日志文件在安装目录下的KSP.log
kuniverse:debuglog ("== abcdefg ==").

kuniverse:RevertToLaunch(). //返回到发射台刚发射前
kuniverse:RevertToEditor(). //返回到车间

kuniverse:QuickSave(). //即时存档
kuniverse:QuickLoad(). //即时读档

set kuniverse:ActiveVessel to vessel("q1"). //切换到名叫q1的飞船

Set c to kuniverse:GetCraft("ABC","VAB"). //在VAB中找名叫"ABC"的载具
kuniverse:LanchCraft(c). //发射这个载具
```

9.10.2

9.10.2 载具存档 CraftTemplate [结构体]

===== 点击以返回目录 =====



(1) 概述

载具存档 CraftTemplate 对应在游戏中车间里的各种载具设计。

玩家可以从 预设变量KUniverse 下的 CraftTemplate 成员来获取该结构体。

(2) 结构体成员

CraftTemplate 载具存档墙结构体 成员列表:

结构体成员	成员类型	读写性	说明
Name	String	只读	载具名称
FilePath	String	只读	载具的保存路径, 绝对路径
Description	String	只读	载具描述
Editor	String	只读	载具所属车间, "VAB" or "SPH"
LaunchSite	String	只读	载具的默认发射场, "LaunchPad" or "Runway"
Mass	Scalar	只读	载具的质量
Cost	Scalar	只读	载具的造价
PartCount	Scalar	只读	载具的部件数量
...			

9.10.3

9.10.3 载具载入距离 VesselLoadDistance [结构体]

===== 点击以返回目录 =====



(1) 概述

玩家可以从 载具Vessel结构体的LoadDistance 成员获取 载具载入距离 VesselLoadDistance。

载具的载入状态，由远及近有下面几种状态：

加载状态	距离	说明
on rails	很远	未载入物理引擎，不在屏幕上显示
loaded	远	未载入物理引擎，但在屏幕上显示方位和距离标记
packed	中	载具的整体外形被载入，在屏幕上可见载具，但不可完全操控
unpacked	近	载具的所有部件都被载入，在屏幕上可见载具，也可操控

(2) 结构体成员

KSP 中，载具分不同场景(场合)，有不同的载入距离设置。

VesselLoadDistance载具载入距离结构体 成员列表：

结构体成员	成员类型	读写性	说明
Escaping	SituationLoadDistance	只读	逃逸轨道载具的载入距离
Flying	SituationLoadDistance	只读	大气层中载具的载入距离
Landed	SituationLoadDistance	只读	地表运动载具的载入距离
Orbit	SituationLoadDistance	只读	在轨载具的载入距离
PreLaunch	SituationLoadDistance	只读	地表静止载具的载入距离
Splashed	SituationLoadDistance	只读	海中载具的载入距离
SubOrbital	SituationLoadDistance	只读	亚轨道载具的载入距离
.....			

9.10.4

9.10.4 场景载入距离 SituationLoadDistance [结构体]

===== 点击以返回目录 =====



(1) 概述

玩家可以从 载具载入距离 VesselLoadDistance 内

获取场景载入距离 SituationLoadDistance。

他对应载具在每一种场景（场合）下的载入距离设置

加载状态	距离	说明
on rails	很远	未载入物理引擎，不在屏幕上显示
loaded	远	未载入物理引擎，但在屏幕上显示方位和距离标记
packed	中	载具的整体外形被载入，在屏幕上可见载具，但不可操控
unpacked	近	载具的所有部件都被载入，在屏幕上可见载具，也可操控
Unloaded	----	远距离，远到大地图上一个点
Loaded	----	远距离
Loaded	UnPacked	中等距离，能看到样子，但还不能操控
Loaded	packed	近距离，能操控

(2) 结构体成员

SituationLoadDistance场景载入距离结构体 成员列表:

结构体成员	成员类型	读写性	说明
Load	Scalar (m)	读写	load距离, 要比Unload距离小
Unload	Scalar (m)	读写	取消load距离, 要比Load距离大
Pack	Scalar (m)	读写	pack距离, 要比Unpack距离小
Unpack	Scalar (m)	读写	取消pack距离, 要比Pack距离大
.....			

(3) 注意事项

考虑到已经Load但还没Unpack的状态是比较尴尬的,
因为 kOS 的自启动程序会在 Load 时开始工作,
但此时还是 Pack 转台, 还不能完全控制载具。

建议载入距离设置为: Pack > Unpack > Load & UnLoad

(4) 实际例子

通过 SituationLoadDistance 设置载具距离

```
set ship:LoadDistance:Landed:pack to 100.
set ship:LoadDistance:Landed:load to 200.
print ship:LoadDistance:Landed:pack.
print ship:LoadDistance:Landed:load.
```

9.10.5

9.10.5 时间 TimeSpan [结构体]

===== 点击以返回目录 =====



(1) 概述

时间 TimeSpan 对应游戏里的模拟时间, 用来返回时间值。

玩家可以通过 预设变量 Time 获取此结构体。

```
Print Time.
Print Time:Seconds.
```

在kOS中, 默认是6 hour= 1 day。

玩家还可以通过 Time 函数创建此类结构体。

```
Time(universal_time)
// 参数 universal_time 为 Scalar 类型, 为当前尤其开始后的秒数,
// 返回 TimeSpan类型
```

```
Print Time(). //如果参数为空则会返回当前时间
```

(2) 结构体成员

该结构体是可序列化（Serializable）的结构体。

TimeSpan时间结构体 成员列表：

结构体成员	成员类型	读写性	说明
Clock	String	只读	“HH : MM : SS” 时间格式
Calendar	String	只读	“Year YYYY, day DDD” 时间格式
Second	Scalar (int 0 - 59)	只读	秒数位的数字
Minute	Scalar (int 0 - 59)	只读	分钟数位的数字
Hour	Scalar (int 0 - 5)	只读	小时数位的数字
Day	Scalar (int 1 - 426)	只读	天数位的数字
Year	Scalar	只读	年数位的数字
Seconds	Scalar	只读	总秒数 (常用)
.....			

*注1：可序列化是指，

该类型变量能通过 WriteJSON 函数 和 ReadJSON 函数 被读写至 json 文件，
指该类型变量能用于僚机之间的通信数据互传。

9.10.6

9.10.6 时间加速 TimeWarp [结构体]

===== 点击以返回目录 =====



(1) 概述

时间加速TimeWarp 对应游戏里时间加速(Warp)的控制。

玩家可以从 预设变量KUniverse 下的 TimeWarp成员 来获取该结构体。

(2) 结构体成员

TimeWarp时间加速结构体 成员列表：

结构体成员	成员类型	读写性	说明
Mode	String	读写	时间加速模式，可设为 "Rails" 和 "Physics"
RailsRateList	Scalar列表	只读	轨道加速模式下的可设档位，取值见下文
PhysicsRateList	Scalar列表	只读	物理加速模式下的可设档位，取值见下文
RateList	Scalar列表	只读	为 RailsRateList 或 PhysicsRateList，由 Mode 值而定
Warp	Scalar	读写	当前设定为哪个档位
Rate	Scalar	读写	当前设定的档位时间流速
WarpTo (time)	无返回值	函数	时间加速到某个时间点
CancelWarp ()	无返回值	函数	取消时间加速
PhysicsDeltaT	Scalar (s)	只读	当前状态下物理演算的间隔时间 只有在物理加速模式下才会返回有意义的值
IsSettled	Boolean	只读	检查当前时间加速是否达到设定值 (KSP 会花时间逐步提高档位来响应设定命令)
.....			

以下是轨道加速模式和物理加速模式下加速档位对应的时间流速。

物理	时间流速	轨道	时间流速
0	x1	0	x1
1	x2	1	x5
2	x3	2	x10
3	x4	3	x50
		4	x100
		5	x1000
		6	x10000
		7	x100000

*注1：如果玩家为 Rate 设置了档位列表里没有的时间流速，Rate 会自动在档位列表里就近取值。

(3) 控制时间加速

```
//时间流速模式切换到物理加速
Set Kuniverse:TimrWarp:Mode to "PHYSICS".
//时间流速模式切换到轨道加速
Set Kuniverse:TimrWarp:Mode to "RAILS".
//将时间流速设定到当前模式下的对应和倍数（例如这里的5代表x1000），倍数可查下表
Set Kuniverse:TimrWarp:Warp to 5.
//准确设定时间加速倍率
Set Kuniverse:TimrWarp:Rate to 100.
//加速到设定时间，例如120秒后
Kuniverse:TimrWarp:WarpTo(TIme:Seconds+120).
//解除时间加速
Kuniverse:TimrWarp:CancelWarp().
```

(4) 控制时间加速（向下兼容）

以下是旧版本里的时间流速控制方式，也同样有效，但不建议用。

```
//时间流速模式切换到物理加速（大气层中的那种，原版游戏用不着这句）
Set WARPmode TO "PHYSICS".
//时间流速模式切换到轨道加速（宇宙里的那种，原版游戏用不着这句）
Set WARPmode TO "RAILS".
//将时间流速设定到当前模式下的对应和倍数（例如这里的5代表x1000），倍数可查下表
Set WARP to 5. //将时间流速设定到当前模式下的对应和倍数（例如这里的5代表x1000），倍数可查下表

WARPTO (TIME:SECONDS+60*10).
//时间快进到10分钟之后
```

9.10.7

9.10.7 指令窗口 Terminal [结构体]

===== 点击以返回目录 =====



(1) 概述

指令窗口 Terminal 对应当前 kOS 处理器的指令窗口设置，玩家可从 预设变量 Terminal 获取。

(2) 结构体成员

Terminal 指令窗口结构体 成员列表：

结构体成员	成员类型	读写性	说明
Width	Scalar	读写	显示屏宽度 (字符数)
Height	Scalar	读写	显示屏高度 (字符数)
Reverse	Boolean	读写	屏幕是否反白显示
VisualBeep	Boolean	读写	把蜂鸣音效换成屏幕闪动
BrightNess	Scalar	读写	字符亮度
CharWidth	Scalar	只读	字符宽度
CharHeight	Scalar	读写	字符高度
Input	TerminalInput	只读	用户的输入内容
.....			

9.10.8

9.10.8 指令窗口输入 TerminalInput [结构体]

===== 点击以返回目录 =====



(1) 概述

玩家可以从 指令窗口 Terminal 的 Input 成员 获取 指令窗口输入 TerminalInput。后者对应用户在指令窗口的输入内容。

(2) 结构体成员

TerminalInput 指令窗口输入结构体 成员列表：

结构体成员	成员类型	读写性	说明
Getchar ()	String	函数	获取指令窗口里玩家输入的下一个字符
HasChar	Boolean	只读	指令窗口里玩家是否有输入
Clear ()	无返回值	函数	清空玩家在指令窗口里的输入
BackSpace	String	只读	BackSpce键的对应字符，用于比对检查特殊字符
DeleteRight	String	只读	Delete键的对应字符，用于比对检查特殊字符
Return	String	只读	Return键的对应字符，用于比对检查特殊字符
Enter	String	只读	Enter键的对应字符，用于比对检查特殊字符
UpCursorOne	String	只读	上键的对应字符，用于比对检查特殊字符
DownCursorOne	String	只读	下键的对应字符，用于比对检查特殊字符
LeftCursorOne	String	只读	左键的对应字符，用于比对检查特殊字符
RightCursorOne	String	只读	右键的对应字符，用于比对检查特殊字符
HomeCursor	String	只读	Home键的对应字符，用于比对检查特殊字符
EndCursor	String	只读	End键的对应字符，用于比对检查特殊字符
PageUpCursor	String	只读	PageUp键的对应字符，用于比对检查特殊字符
PageDownCursor	String	只读	PageDown键的对应字符，用于比对检查特殊字符
.....			

(3) 实际例子

例子，下图载具，玩家输入 L 键被抓取并识别，然后程序打开载具的灯光

```
Until False { //一个死循环，本程序持续运行
    set ch to terminal:input:getchar().
    if ch = "L" {
        Lights on. //如果打的字是"L"，那么就开灯
    } else {
        Print "No Order". //如果不是，就报错
    }
    wait 0.001. //等到下一个物理帧
}
```

程序运行效果如下：



9.10.9

9.10.9 音轨 Voice [结构体]

===== 点击以返回目录 =====



(1) 概述

音轨 Voice 对应的是 音频演奏（SKID）里的音频轨道Voice。

玩家可以通过 GetVoice 函数 获取音轨，通过 StopAllVoices 函数清空音轨。

```
Set a to GetVoice(0).
StopAllVoices().
```

详见 音频演奏（SKID）

(2) 结构体成员

Voice 音轨结构体 成员列表：

结构体成员	成员类型	读写性	说明	默认值
Attack	Scalar (s)	读写	音轨内音符的淡入时间	0
Decay	Scalar (s)	读写	音轨内音符的衰退时间	0
Sustain	Scalar (s)	读写	音轨内音符的维持时间	1
Release	Scalar (s)	读写	音轨内音符的淡出时间	0.1
Volume	Scalar	读写	音轨的音量，范围 0~1	1
Wave	String	读写	音轨的波形	□
Play (song)	无返回值	函数	播放乐谱 song, song 是音符 Note 的列表	
Stop ()	无返回值	函数	停止播放乐谱	
Loop	Boolean	读写	是否循环播放	False
IsPlaying	Boolean	读写	是否正在播放	
Tempo	Scalar	读写	音轨的播放速度	1
.....				

*注1：:Wave 可以是这些值：“Square”、“Triangle”、“Sawtooth”、“Sine”、“Noise”

9.10.10

9.10.10 音符 Note [结构体]

===== 点击以返回目录 =====



(1) 概述

音符 Note 对应的是 音频演奏（SKID）里的单个音符。

完结可以通过 Note 函数构造音符。

```
//调用函数时如果不写全参数，没输入的参数会自动取默认值
//所以 Note函数 和 SlideNote函数 有以下几种用法：

Note(频率,音符时长,按键时长,音量) //纯音音符
Note(频率,音符时长,按键时长) //未赋值的参数会取默认值
Note(频率,音符时长) //未赋值的参数会取默认值

SlideNote(开始频率,结束频率,音符时长,按键时长,音量) //渐变音音符
SlideNote(开始频率,结束频率,音符时长,按键时长) //未赋值的参数会取默认值
SlideNote(开始频率,结束频率,音符时长) //未赋值的参数会取默认值
```

详见音频的代码实现

(2) 结构体成员

Note音符结构体 成员列表：

结构体成员	成员类型	读写性	说明	默认值
Frequency	Scalar (Hz)	只读	音符的起始频率	
EndFrequency	Scalar (Hz)	只读	音符的结束频率	
KeyDownLength	Scalar (s)	只读	音符的按键时长	1
Duration	Scalar (s)	只读	音符的音符时长	1
Volume	Scalar	只读	音符的音量	1
.....				

9.10.11

9.10.11 版本信息 VersionInfo [结构体]

===== 点击以返回目录 =====



(1) 概述

版本信息 VersionInfo 对应KSP的版本。玩家可从 Core:Version 获取。

(2) 结构体成员

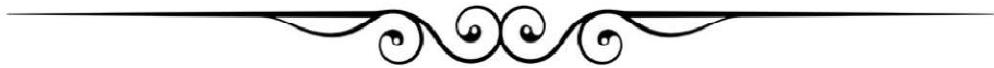
VersionInfo 版本信息 结构体 成员列表：

结构体成员	成员类型	读写性	说明
Major	Scalar	只读	[vxx].xx.xx.xx
Minor	Scalar	只读	vxx.[xx].xx.xx
Patch	Scalar	只读	vxx.xx.[xx].xx
Build	Scalar	只读	vxx.xx.xx.[xx]
.....			

9.10.12

9.10.12 路标 WayPoint [结构体]

===== 点击以返回目录 =====



(1) 概述

路标WayPoint就是KSP游戏里原有的位置标记。

(2) 创建结构体

WayPoint(name)	从已接受的合同里读取对应地点生成路标 name是字符串，调用时要加双引号
AllWayPoints()	从已接受的所有合同里读取所有地点 生成路标的列表

(3) 结构体成员

WayPoint路标结构体 成员列表：

结构体成员	成员类型	读写性	说明
Name	String	只读	路标名称，显示在地图上的那个文本
Body	Body	只读	路标所在天体
Geoposition	GeoCoordinates	只读	路标的经纬坐标
Position	Vector (m)	只读	路标的矢径 (Ship - raw坐标系)
Altitude	Scalar (m)	只读	路标点的地面海拔高度
AGL	Scalar (m)	只读	路标点的离地高度
NearSurface	Boolean	只读	是否是近地路标点
Grounded	Boolean	只读	是否是地表路标点
Index	Scalar	只读	所在路标序列中的编号
Clustered	Boolean	只读	是不是某个路标序列的一员
.....			

10.

10. MOD 相关

10.1

10.1 MOD支持 Addons [结构体]

===== 点击以返回目录 =====

KOS对以下六个MOD做了特殊支持。

全称	简称	说明
Action Groups Extended	AGE	动作组扩展
RemoteTech	RT	通讯限制
Kerbal Alarm Clock	KAC	坎巴拉闹钟
Infernal Robotics	IR	转轴
Trajectories	TR	落点预测

Addons结构体用于查询游戏中是否安装了相应MOD。

玩家可以用预设变量Addons来获取该结构体。

Addons MOD支持结构体 成员列表:

结构体成员	成员类型	读写性	说明
Available ("AGX")	Boolean	只读	是否安装了 动作组扩展MOD
Available ("RT")	Boolean	只读	是否安装了 通讯限制MOD
Available ("KAC")	Boolean	只读	是否安装了 坎巴拉闹钟MOD
Available ("IR")	Boolean	只读	是否安装了 转轴MOD
TR : Available	Boolean	只读	是否安装了 落点预测MOD
.....			

10.2

10.2 动作组扩展 Action Groups Extended (AGE)

===== 点击以返回目录 =====



动作组拓展MOD 允许玩家使用 250 个动作组，而不是原版的 10 个。

动作组拓展还支持发射后的动作组再编辑。

用法和之前的一样： AG11 on. ~ AG250 off.

```
declare cooldownTimeAG15 to 0.
on AG15 {
    if cooldownTimeAG15 + 10 < time:seconds {
        print "Solar Panel Toggled!".
        set cooldownTimeAG15 to time.
    }×
    preserve.
}
```

10.3

10.3 通讯MOD支持 RTAddon [结构体]

===== 点击以返回目录 =====



RTAddon结构体 是有关通讯MOD的结构体，用于检测 通讯MOD的工作状态。

玩家可以用 Addons:RT 来获取该结构体。

结构成员	成员类型	读写性	说明
Available	Boolean	只读	通讯MOD是否已安装并生效
Delay (vessel)	Scalar (s)	函数	给定飞船的通讯延迟
KSCDelay (vessel)	Scalar (s)	函数	给定飞船的从KSC通讯延迟
AntennaHasConnection (part)	Boolean	只读	是否有通讯连接
HasConnection (vessel)	Boolean	只读	是否受控
HasKSCConnection (vessel)	Boolean	只读	是否连接到KSC
HasLocalControl (vessel)	Boolean	只读	是否能控制
GroundStations ()	String列表	只读	所有地面站的列表

*注1：如果 RT 代替了 原版KSP 的通讯规则，
那么使用 RT 会影响 kOS 的报文Message 的传递。

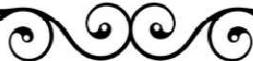
10.4

10.4 闹钟 Kerbal Alarm Clock (KAC)

10.4.1

10.4.1 闹钟 KACAddon [结构体]

===== 点击以返回目录 =====



Kerbal Alarm Clock 用于自动时间加速到设定时间点。与 kOS 的时间加速功能重复。

闹钟 KACAddon 则是对应坎巴拉闹钟的结构体。

玩家可以用 Addons:KAC 来获取该结构体。

KACAddon 闹钟结构体 成员列表：

结构体成员	成员类型	读写性	说明
Available	Boolean	只读	KAC是否已安装并生效
Alarms ()	KACAlarm 列表	函数	所有闹铃的列表
.....

10.4.2

10.4.2 闹铃 KACAlarm [结构体]

===== 点击以返回目录 =====



闹铃 KACAlarm 是对应坎巴拉闹钟里单个时间点闹铃的结构体。

玩家可以用 Addons:KAC:Alarms() 来获取该结构体。

KACAlarm 闹铃结构体 成员列表：

结构体成员	成员类型	读写性	说明
Id	String	只读	闹铃的UID
Name	String	读写	闹铃名称
Action	String	读写	闹铃行为
Type	String	只读	闹铃类型
Notes	String	读写	闹铃的详细说明
Remaining	Scalar (s)	只读	距离闹铃触发时间还有多久
Repeat	Boolean	读写	闹铃是否能重复触发
RepeatPeriod	Scalar (s)	读写	闹铃重复触发的CD间隔长度
OriginBody	String	读写	载具要离开的天体的名字
TargetBody	String	读写	载具要去往的天体的名字
.....

*注1：:ACTION 可以是：“MessageOnly”、“KillWarpOnly”、“KillWarp”、“PauseGame”、

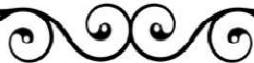
*注2：:Type 是创建闹铃时设定的，它可以是（不懂就别乱设）：

“Raw”(默认)、“Maneuver”、“ManeuverAuto”、“Apoapsis”、“Periapsis”、“AscendingNode”、“DescendingNode”、“LaunchRendevous”、“Closest”、“SOICChange”、“SOICChangeAuto”、“Transfer”、“TransferModelled”、“Distance”、“Crew”、“EarthTime”

10.4.3

10.4.3 使用坎巴拉闹钟

===== 点击以返回目录 =====



KAC闹钟操作函数

创建闹钟

AddAlarm(闹钟类型, 设定时间, 闹钟名称, 备注)

list闹钟

ListAlarms(闹钟类型)

删除闹钟

DeleteAlarm(闹钟UID)

代码示例

```
Set na to addAlarm("Raw",time:seconds+300, "Test", "Notes").
print na:NAME. //prints 'Test'
Set na:NOTES to "New one".
print na:NOTES. //prints 'New one'
```

下图为以上代码设定的闹钟



截图来自在旧版本KSP游戏中使用kOS的此命令

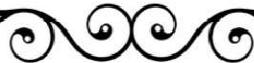
10.5

10.5 转轴 Infernal Robotics (IR)

10.5.1

10.5.1 转轴MOD支持 IRAddon [结构体]

===== 点击以返回目录 =====



IRAddon 支持 2.1.4 版本转轴 (Infernal Robotics)，不支持 3.0.0 版本转轴 (Infernal Robotics - Project Next)。

但 3.0.0 版本转轴仍可使用 PartModule 进行控制。

转轴MOD支持 IRAddon 对应转轴系统的控制。

玩家可以用 Addons:IR 来获取该结构体。

IRAddon 转轴MOD支持结构体 成员列表：

结构体成员	成员类型	读写性	说明
Available	Boolean	只读	IR是否已安装并生效
Gruops	IRControlGroup 列表	只读	转轴控制组的列表
AllServos	IRServo 列表	只读	转轴单元的列表
PartServos (part)	IRServo 列表	只读	获取某部件上所有的转轴
.....			

10.5.2

10.5.2 转轴控制组 IRControlGroup [结构体]

===== 点击以返回目录 =====



IRControlGroup 支持 2.1.4 版本转轴 (Infernal Robotics)，不支持 3.0.0 版本转轴 (Infernal Robotics - Project Next)。

但 3.0.0 版本转轴仍可使用 PartModule 进行控制。

(1) 转轴控制组

转轴MOD 不仅允许玩家单独操控每一个转轴单元，还允许玩家将几个转轴单元并成一组统一操控。如下图。



另外，转轴MOD还允许玩家事先设定数个预设位置，而后依靠切换预设位置（按键）来进行快捷准确的操作。如下图。



(2) 结构体成员

玩家可以在 Addons:IR:Groups 来获取该结构体的列表。

IRControlGroup 转轴控制组结构体 成员列表：

结构体成员	成员类型	读写性	说明
Name	String	读写	控制组名称
Speed	Scalar	读写	传动速度
Expanded	Boolean	读写	该转轴控制组是不是在转轴界面里打开了
ForwardKey	String	读写	正向传动的快捷键
ReverseKey	String	读写	逆向传动的快捷键
Servos	IRServo列表	只读	转轴单元的列表
Vessel	Vessel	只读	转轴控制组所在的载具
MoveRight ()	无返回值	函数	转轴控制组往右持续移动
MMoveLeft ()	无返回值	函数	转轴控制组往左持续移动
MoveCenter ()	无返回值	函数	转轴控制组往中心位置移动
MoveNextPreset ()	无返回值	函数	转轴控制组往下一个预设位置移动
MovePrevPreset ()	无返回值	函数	转轴控制组往上一个预设移动
Stop ()	无返回值	函数	转轴控制组停止移动
.....			

(3) 实际例子

```
ADDONS:IR:GROUPS[0]:MOVENEXTPRESET(). 第一个传动组的往下一个预设位置移动
ADDONS:IR:GROUPS[0]:MOVERIGHT(). 第一个传动组持续往右移动
ADDONS:IR:GROUPS[0]:STOP(). 第一个传动组停止移动
```

10.5.3

10.5.3 转轴单元 IRServo [结构体]

===== 点击以返回目录 =====



IRServo 支持 2.1.4 版本转轴 (Infernal Robotics)，不支持 3.0.0 版本转轴 (Infernal Robotics - Project Next)。
但 3.0.0 版本转轴仍可使用 PartModule 进行控制。

(1) 概述

转轴单元 IRServo 对应单个转轴部件的控制。

玩家可以使用 Addons:IR:PartServos(part) 函数获取某转轴部件对应的转轴单元。

```
//假设a是某部件
Addons:IR:PartServos(a)[0]
```

(2) 结构体成员

IRServo 转轴单元结构体 成员列表：

结构体成员	成员类型	读写性	说明
Name	String	读写	传动单元名称
Uid	Scalar	只读	UID编号
HighLight	Boolean	只写	是否高亮部件
Position	Scalar	只读	转轴单元的位置
MinCfgPosition	Scalar	只读	part.cfg里定义的最小传动位置
MaxCfgPosition	Scalar	只读	part.cfg里定义的最大传动位置
MinPosition	Scalar	读写	右键菜单中的最小位置
MaxPosition	Scalar	读写	右键菜单中的最大位置
ConfigSpeed	Scalar	只读	part.cfg定义的即时移动速度
Speed	Scalar	读写	右键菜单中的移动倍数，读写
CurrentSpeed	Scalar	只读	当前移动速度
Acceleration	Scalar	读写	右键菜单中的加速度倍数，读写
IsMoving	Boolean	只读	转轴单元是否在移动
IsFreeMoving	Boolean	只读	转轴单元是否是不可控的
Locked	Boolean	读写	锁定传动单元，读写
Inverted	Boolean	读写	反转控制方向，读写
Part	Part	只读	当前转轴单元对应的部件
MoveRight ()	无返回值	函数	转轴单元向右移动
MoveLeft ()	无返回值	函数	转轴单元向左移动
MoveCenter ()	无返回值	函数	转轴单元回到中心位置
MoveNextPreset ()	无返回值	函数	转轴单元到下一个位置
MovePrevPreset ()	无返回值	函数	转轴单元到上一个位置
Stop ()	无返回值	函数	转轴单元停止
MoveTo (position, speed)	无返回值	函数	转轴单元以设定的速度 移动到设定位置
.....			

(3) 实际例子

```
Addons:IR:Groups[0]:Servos[0]:MoveTo(180,5).
第一个传动组的第一个部件以速度5移动到180位置
```

10.6

10.6 落点预测 Trajectories (TR)

10.6.1

10.6.1 落点 TRAddon [结构体]

===== 点击以返回目录 =====



(1) 概述

Trajectories 是一个预测大气层内飞行时落点的 MOD。

落点TRAddon 对应 Trajectories 提供的信息。

玩家可以通过 Addons:TR 获取该结构体。



*注: Trajectories 提供的信息并不十分精准, 仅供参考。

(2) 结构体成员

TRAddon落点结构体 成员列表:

结构体成员	成员类型	读写性	说明
<code>Available</code>	<code>Boolean</code>	只读	是否Trajectories安装并生效
<code>GetVersion</code>	<code>String</code>	只读	Trajectories版本号
<code>IsVerTwo</code>	<code>Boolean</code>	只读	Trajectories版本号是否不低于2.0.0
<code>IsVerTwoTwo</code>	<code>Boolean</code>	只读	Trajectories版本号是否不低于2.2.0
<code>HasImpact</code>	<code>Boolean</code>	只读	当前载具是否有落点
<code>ImpactPos</code>	<code>GeoCoordinates</code>	只读	当前轨道的预测落点
<code>PlannedVec</code>	<code>Vector</code>	只读	想要维持当前的预测轨道载具应该摆放在的朝向矢量
<code>PlannedVector</code>	<code>Vector</code>	只读	同上
<code>SetTarget (GeoCoordinates)</code>	无返回值	函数	设置目标落点
<code>HasTarget</code>	<code>Boolean</code>	只读	是否有落点
<code>TimeTillImpact</code>	<code>Scalar (s)</code>	只读	多久后落地时间(秒)
<code>ProGrade</code>	<code>Boolean</code>	读写	设置降落方式为正向降落
<code>RetroGrade</code>	<code>Boolean</code>	读写	设置降落方式为逆向降落
<code>CorrectedVec</code>	<code>Vector</code>	只读	想要到达设定的目标落点载具应该摆放在的朝向矢量
<code>CorrectedVector</code>	<code>Vector</code>	只读	同上
.....			

11.

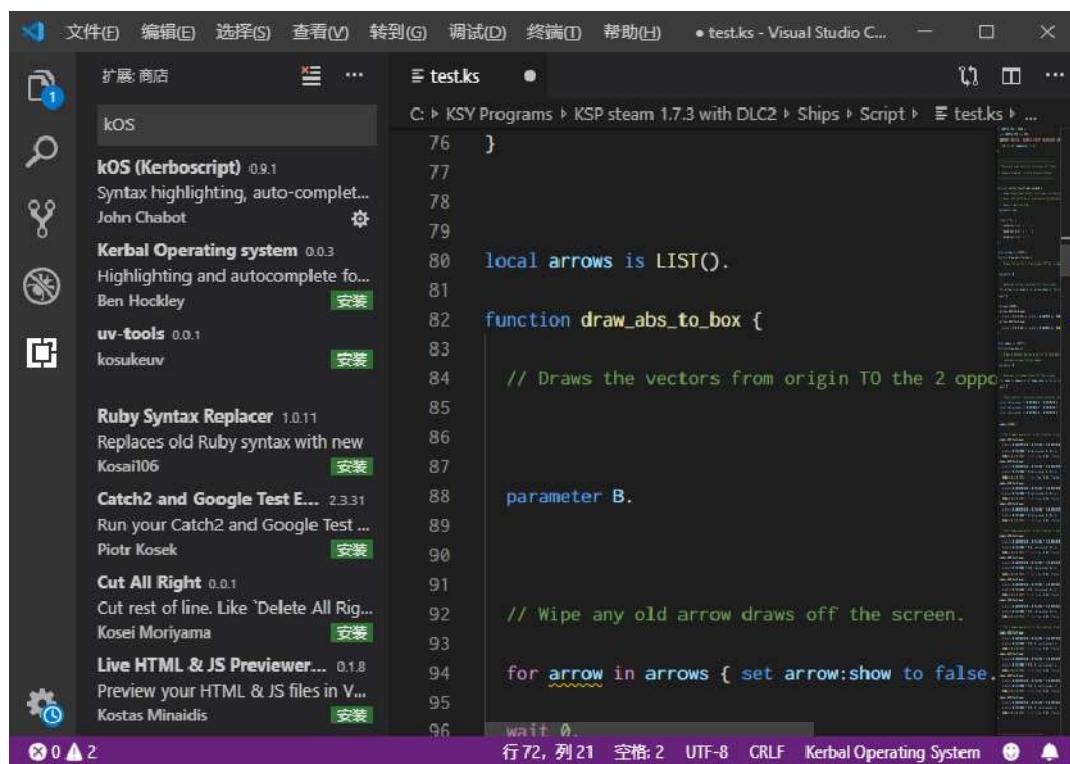
11. kOS 的编辑环境

===== 点击以返回目录 =====



Visual Studio Code (可在软件里下载 kOS 支持文件)

<https://code.visualstudio.com>



文件(E) 编辑(E) 选择(S) 查看(V) 转到(G) 调试(D) 终端(I) 帮助(H) • test.ks - Visual Studio C... ━ ×

扩展:商店

kOS

kOS (Kerboscript) 0.9.1
Syntax highlighting, auto-complet...
John Chabot

Kerbal Operating system 0.0.3
Highlighting and autocomplete fo...
Ben Hockley 安装

uv-tools 0.0.1
kosukeuv 安装

Ruby Syntax Replacer 1.0.11
Replaces old Ruby syntax with new
Kosai06 安装

Catch2 and Google Test E... 2.3.31
Run your Catch2 and Google Test ...
Piotr Kosek 安装

Cut All Right 0.0.1
Cut rest of line. Like 'Delete All Rig...
Kosei Moriyama 安装

Live HTML & JS Previewer... 0.1.8
Preview your HTML & JS files in V...
Kostas Minaidis 安装

test.ks

```
76 }
77
78
79
80 local arrows is LIST().
81
82 function draw_abs_to_box {
83
84 // Draws the vectors from origin TO the 2 oppo...
85
86
87
88 parameter B.
89
90
91
92 // Wipe any old arrow draws off the screen.
93
94 for arrow in arrows { set arrow:show to false.
95
96 wait 0.
```

行72, 列21 空格:2 UTF-8 CRLF Kerbal Operating System