蓝星际自动语音平台
Koodoo 语言
语法手册
Ver 1.8

BlueSpace

www.bluespace.com.cn
zhudn@bluespace.com.cn

0 目 录

1 概述

- Koodoo 语言是一种脚本语言，面向 CTI 方面的 IVR 和呼叫中心开发
- Koodoo 语言是一种动态语言
- Koodoo 语言类似 C 语言，是一种非常简单的结构化语言。
- 脚本文件就是使用 Koodoo 语言编写的源文件。
- 脚本文件缺省的扩展名为".bss"，bss 是 Blue Space Script(蓝星际脚本)的简称。
- 每行只允许一条语句，如果语句太长在一行内写不完，则可以在末尾增加续行符号'\'。
- 除了流程控制语句外每条语句必须以';'号结束。
- 为了增加可读性，可以有空行，可以并推荐采用缩进格式。
- 语句块使用 {} 大括号对，但 '{' 及 '}' 都必须单独一行。
- Koodoo 语言大小写敏感。

* Koodoo 音[kudu]，是一种南非条纹羚羊，象征着这种语言可以在多条线路上独立运行

2 运行环境

Koodoo 语言目前运行在蓝星际的开发平台或运行平台上，但实际上该语言并没有和硬件相关的元素，而且按通用的语言来设计，因而原则上它可以移植到各种平台或硬件之上。

脚本语言的运行单位是单一的线路（通道），一般在独立的线程上运行，它通常情况下只操作本线路的资源，这样每条线路互不干扰，我们甚至可以在各条线路上运行不同的脚本，实现不同的应用。

虚拟线路： 除了物理硬件线路外，系统增加了 1 线虚拟线路(在运行平台条数可配置多条)，虚拟线路可以编译所有的系统函数，但实际运行时有些语音操作类函数不会有实际的动作，有些则使用声卡麦克风等模拟，如 Play()等放音函数将自动在声卡上播放，Record()在麦克风上录音，GetKeys()用键盘按键模拟输入。虚拟线路可以在没有实际语音卡硬件的情况下调试程序,也可以运行不需要语音操作的如短信网关、数据库监控等脚本。虚拟线路的线路号排在所有的物理线路之后。

开发平台： BsTel.exe

在指定的一条线路上编译运行单个脚本，可以直观地了解运行流程以及所有变量的值，可以设置断点，单步执行，跟踪到子函数内部等等，是一个图形化的集成编程测试工具。

开发平台可以在蓝星际网站上免费下载。

开发平台也支持命令行参数，自动地执行指定的脚本，如：

```
BsTel.exe 脚本文件名 [线路号]
```

系统将自动在第 0 条线路或指定的线路编译并运行该脚本。这样用户免费下载开发平台后可以部署到最终用户，因为在很多情况下 1 线也是有实际价值的，如基于 TAPI 接口的语音 Modem 桌面应用，或短信应用等等。

运行平台： BsTelRun.exe

相应的配置文件为 BsTelRun.cfg，配置文件可以为线路指定相应的脚本文件，格式为:

```
0 = MainIvr.bss    // 指定 0 号线路的脚本文件为"MainIvr.bss"
```

或

```
1,3,4 = market.bss    // 指定 1 号、3 号及 4 号线路的脚本文件为"market.bss"
```

或

```
5-12 = group1.bss    // 指定 5 号至 12 号线路共 8 条线路都运行相同的脚本
                        文件"group1.bss"
```

配置文件还支持更多的运行参数，具体说明参见文档<运行平台配置说明>.

运行平台的界面比较简明，主要给用户提供一个系统运行的监视窗口.

系统还支持远程监控。

---

3 常量

常量有下面几种隐含的基本类型：

字符串：　"abcd"，"蓝星际公司"等，是用双引号引起来的内容。如果字符串中含有双引号"，则需用反斜杠转义，如：\"My name is \"bluesen\".\"，实际上表示字符串：My name is "bluesen". 与 C/C++一样，反斜杠要用'\\'表示
除了'\"'和'\\'以外，转义字符还有'\n'表示回车，即 ascii 码 13；'\r'表示换行，即 ascii 码 10； '\t'表示制表符号，即 ascii 码 5。其它特殊字符可以使用 Asc()函数生成.

整　型：　189，0，-2000 等

浮点型：　12.89，-0.01 等，目前不支持科学记数法.

空类型：　只有一个值就是 NULL

函数型：　用 function 语句定义的函数，函数名字相当于一个常量，可用来赋值给其它变量和进行函数调用

常量符号：　语法为：const 常量符号 = 常量;
常量不允许重新赋值。
习惯上常量符号全部大写。

例如：　const S_LXJ = "BlueSpace Co.";　// 定义了常量 S_LXJ，它的值为字符串"BlueSpace Co."
如果在后面的语句中出现：
S_LXJ = S_LXJ + " SHENZHEN";
将不起作用，因为常量不允许重新赋值，但系统也不会报错。

下面关于变量参与的表达式运算，对于常量也是一样。

系统预先定义了 4 个符号常量：

```
_lineNo    // 当前线路号，整型，从 0 开始；这个只读变量在编程中非常有用，
           比如读取本线路的消息队列等等
_lineType // 当前线路的类型，整型，0-虚拟线路，1-模拟外线，2-模拟内线，
           3-录音线路，4-数字线路
_linesCount    // 所有的线路总数，整型，包括虚拟线路
_bssFile  // 本线路当前脚本文件名，字符串类型
_bssDir    // 本线路当前脚本文件所在的路径，字符串类型
true       // 逻辑真，也就是整型 1
false      // 逻辑假，也就是整型 0
NULL       // 空值，通常用来判断未定义的数组成员，如 if( m[100]==NULL )，
           注意 NULL 既不是整型 0，也不是字符串""
```

---

4 变量

系统支持多种隐含的数据类型，分别是整型，浮点型，字符串型、数组类型以及函数类型，函数类型就是将函数名字赋值给变量，这样可以在运行时刻动态指定函数调用的方式，数组类型在下一章专门讨论。还有一种特殊的数据类型是空类型，就是 NULL。

布尔类型等于整型。

Koodoo 语言是动态类型的语言，变量的类型可以动态改变。

变量不必预先指定类型，具体的类型在赋值时自动确定，如 i = 0; 表示 i 是整型其值为 0. 在下一次赋值时可以变成其它的类型如 i="BlueSpace Co."是字符串类型，i = 19.68 则变成浮点类型，i=19 则成为整型。通过 Type(v, t)系统函数可以得到变量的当前类型。

变量名可以使用中文，也可以使用系统关键字(不推荐)，但变量名(以及自定义函数名)不允许数字开头，也不允许包含下列字符： +, -, *, /, \, ", !, %, 空格, (,), {, }, [,], '等等。

变量可以运算，目前支持下列操作:

---

1). 一个变量的赋值或变量的运算，如 v = 100; 或 a = b + c; 或 a = i / 12.23; 或 boolVal = a>b;, 系统支持的操作符有:

+	变量相加
-	变量相减 (对字符串无意义)
*	变量相乘 (如字符串乘
/	变量相除 (对字符串无意义)
%	变量取余 (对字符串无意义)
==	相等比较 (字符串比较时，大小写敏感)
>	大于比较
<	小于比较
>=	大于等于比较
<=	小于等于比较
!=	不等于比较
||	逻辑或
&&	逻辑与
!	逻辑取否
~	补
&	按位与
|	按位或
^	异或
<<	左移
>>	右移

注意: 系统支持复杂表达式,

例如:　　if(i>0 && color=="Blue")

或

v = 10 - 8*2 + a*(i1-10);

等等.

表达式的优先级和 C 语言相同，可以使用括号提高优先级.

对于两个不同类型的变量之间运算，如 v = a op b;，op 是指上面的运算符，归纳如下：

a).如果 a 为字符串，则 v 也为字符串，支持字符串相加（拼接）和字符串乘以整数（重复串），详见下面章节的讨论；

b).否则，如果 a 或 b 其中之一为浮点型，则 v 也浮点型，如：

```
v = 101.67 + "22.01";  // 结果为 123.68
v = 123.45 + 200;      // 结果为 323.45
```

c).否则，如果 a 或 b 其中之一为整型，则 v 也整型，如：

```
v = 23 + "22.01";  // 结果为 45
if( 23 > "22" )    // 结果为 true
```

d).否则，如果类型不匹配，则结果未知

---

2). 多个变量的相加，常用在字符串拼接。

例如　　sqlStr = "select * from " + tab1 + "where date > '" + sDate + "'";

直观而方便。

当然，其它类型相加也是可以的: 如 sum = 1 + 2 + 3 + 100;

注意 1：　如果第一个变量为字符串，则结果必然为字符串，而且后面的变量如果不是字符串类型将会自动转换为合适的字符串。

例如：　　num = 349.87;

s1 = "年份: " + 2003 + ", 总金额是: " + num;

则运算后 s1 的值为: "年份: 2003, 总金额是: 349.87", 这个特性可以运用在类型转换上:

s1 = "" + 256; // 将整型 256 转换为字符串"256"

注意 2：　在同一命令行，最多只能有 40 个变量相加

---

3). 变量自加或自减，变量必须为整型，

例如：　　i ++;

或 ++ loop;

j --;

或 -- x;

主要用在循环语句上，

注意:　　目前版本不允许将自加或自减用在赋值表达式中，所以 i++与++i 没有任何区别。

---

4). 字符串的重复复制，即字符串乘以一个整数，

例如：　　s = "abc"*4;

s 的值为: "abcabcabcabc"

系统预先已经定义了 2 个变量，分别是:

_tempVal　// 临时变量，系统内部使用，尽量不要在程序中使用它。

_retVal　// 操作返回值，整型，每个系统函数或自定义函数的返回值均放

在这里。

注意： _retVal 是动态变化的，每条语句执行完它就会被置新值，不管这条语句是赋值语句还是调用动态库函数还是什么别的语句。不推荐程序去写它。在程序中通常需要读取函数的返回值。

例如:
```
IsRings(1);   // 现在有来电吗?
if( _retVal > 0 )  // _retVal 就是调用上一条语句的返回值
{
   // 处理来电...
}
```
不过这样不太直观，通常应该这样写：
```
ring = IsRings(1); // 直接将函数返回值赋值给变量
if( ring )
{
   // 处理来电
}
```

变量分为全局变量和局部变量，在用户自定义函数里定义的变量为局部变量，在主模块里定义的变量为全局变量。其意义与 C 语言相同。

---

5 数组

数组实际上相当于字典或散列表，通过下标访问。

数组的使用是很灵活的，规则如下：

数组下标可以是任意类型，即使是整型也可以不连续；

数组等同于字典或散列表，不存在越界的问题；

数组可以用在复杂表达式中，其含义与其它语言如 C 一样；

数组可以有别名，就是不带下标的赋值，在函数调用中如果实参为数组名，函数内形式参数就是该数组的别名。所谓别名实际上就是原来数组名的引用，指向同一个数组。使用函数 Len(m)可以得到数组的成员个数，for(i in m)语句可以遍历数组的所有成员

使用 if(m[k]==NULL)可以判断 k 是否为数组 m 的下标成员

两个数组相加结果仍然是数组，以下标作为集合，C = A + B; C 等于 A 和 B 的并集

两个数组相减结果仍然是数组，以下标作为集合，C = A - B; C 等于 A 和 B 的差集

两个数组相乘结果仍然是数组，以下标作为集合，C = A * B; C 等于 A 和 B 的交集

两个数组相等比较，数组变量和它的别名变量相等，否则不相等

两个数组大于和小于比较，按数组的成员数量进行比较，即成员数多的数组更大

数组右移一个变量，表示移去该变量成员，A 为数组，B 是 A 的一个下标成员，则 C = A >> B; 将 B 从 A 中移去，C 是 A 的引用；如果只是需要移去数组 A 的成员，应该写成：A = A>>B;

例如：

```
m[0] = 12;  // m 为数组类型，下标 0 对应的值是 12
m[1] = "English";
m["123 "] = 78;  // 下标可以是字符串
i = m[0]*100 + 200;
p = m;  // 为数组 m 创建一个别名 p
p[8] = 90.23;  // 为数组增加一个成员
v = 0;
if( p[v]*20 < 10000 )  // 数组也可以用在条件判断表达式中
{
  p[v] = 10000;
}
```

在开发平台，双击变量列表中的数组型变量，可以浏览整个数组的全部元素。

当一个数组没有任何引用了，数组所占用的内存将自动释放。如：

```
m[0] = 12;  // m 为数组类型，下标 0 对应的值是 12
m[1] = "English";
p = m;
…
m = 0; // m 现在是整数 0
p = 0; // p 现在是整数 0，原来的数组没有任何引用，内存被释放
```

数组的成员可以指向另外一个数组，构成灵活的多维数组：

```
n[0] = 1;
n[1] = 2;

p[0] = 21;
p[1] = 22;

m["n"] = n;  // 指向数组 n
m["p"] = p;  // 指向数组 p

v1 = m["n"][0];  // 其值为 n[0]，等于 1
v2 = m["p"][1];  // 其值为 p[1]，等于 22
```

注意数组的下标如果是数组，下标值等于该数组的整型 id 值。

某些系统函数其输出参数为数组，该数组下标为从 0 开始的连续整数，如 AnlyStr(), FileReadLines(), GetFileList()等。

---

6 注释

与 C/C++相同，有两种注释方法：

单行注释，以 // 开始直到行尾；

多行注释，以 /* 开始，以 */ 结束.

7 语句

语句分为 3 种

1.变量赋值或表达式运算，在变量部分已经作了说明，其特点为 ： 变量 = 表达式;

2.操作类语句，表示对应一个系统操作、自定义函数调用或外部动态库调用，系统操作不需要预先定义，自定义函数和外部动态库就需要在调用前预先定义，具体参加后面相关章节。

表示为 ： 操作名([参数 1[,参数 2[,...参数 n]]]);

例如：Play(VocLangSele);

对应一个放音操作，参数是个字符串变量(或常量)，指出要放音的文件名

这类似 C 语言函数调用的写法

例如：ret = Play(VocLangSele);

操作的返回值放在系统变量_retVal 里，也可以写成:

```
Play((VocLangSele);
ret = _retVal;
```

如操作有更多的值需要返回，放置在规定的参数里，按引用方式返回

例如：GetKeys(keys，1，10)；按键数放在返回值或系统变量_retVal 里，用户按键返回在变量 keys 里

具体的操作功能及其参数列表，参见文档<蓝星际语音平台系统函数参考手册>，随着系统功能的扩展，'操作'会增加，这就像 C 语言的标准库函数.

注意：参数支持复杂表达式，但函数不能当成参数，如：

```
Play(dir+"welcome.wav");  // 参数支持表达式
if( GetKeys(k，10，40) )  // 不能直接使用函数当成 if 语句的参数
```

3.流程控制类语句，主要有 4 种

---

1).条件语句

```
if(表达式或布尔变量)
{
   语句块  // 表达式或布尔变量为真时执行
}
else
{
   语句块 // 表达式或布尔变量为假时执行
}
```

注意：　　else 语句块可以省略

和 C 语言一样，Koodoo 语言还支持 else if()语句:

```
if(表达式或布尔变量)
{
   语句块
}
```

```
else if(表达式或布尔变量)
{
   语句块
}
else
{
   语句块
}
```

对于整个 if 条件语句,可以有 0-n 个 else if()语句和 0-1 个 else 语句,else if()语句对于多分支判断非常有用，可以使逻辑更清晰，代码更简洁。

和 C 语言一样，如果语句块只有一句，则{和}可以省略。

注意: else if()语句必须在同一行。

---

2).循环语句

```
while(表达式或布尔变量)
{
   语句块    // 表达式或布尔变量为真时执行
}
```

语句块中可以包含 break 和 continue 语句，break 直接退出循环，而 continue 忽略下面的语句直接回到 while 继续

如果语句块只有一句，则{和}可以省略。

---

3).for 语句

i). 标准循环语句

```
for(初始表达式; 表达式或布尔变量; 步进表达式)
{
   语句块    // 表达式或布尔变量为真时执行
}
```

最常见的用法类似下面:

```
for(i=0; i<100; i++)
{
   语句块
}
```

ii). 字符串或数组遍历语句

```
for(变量 a in 数组或字符串变量 b)
{
   语句块
}
```

规则如下:

- 变量 a 可以不预先赋值
- 变量 b 如果为字符串，则变量 a 将顺序访问其每一个字符，直到字符串结束

- 变量 b 如果为数组，则变量 a 将顺序访问其每一个数组下标
- 变量 b 如果既不是字符串也不是数组，则语句块一次也不会被执行，当然也不会报错

注意：for in 语句是一种快速遍历字符串或数组的语法结构，不要在语句块内改变数组下标或字符串的内容，否则后果不可预测

例如：

```
// 遍历数组
s = "hello";
for(ch in s)
{
   x = ch; // 本条语句将执行 5 次,x 的值将顺序为:"h","e","l","l","o"
}

// 遍历字符串
m["CN"]="中国";
m["USA"]="美国";
for(i in m)
{
   x = i;
   y = m[i];
}
```

两种 for 语句的语句块内都支持 break 语句和 continue 语句：
与 while 语句相同，break 直接退出循环，而 continue 忽略下面的语句直接回到 for 继续
如果语句块只有一句，则{和}可以省略。

---

4).多分支条件语句

```
switch(主变量)
{
   case(变量 1 或常量 1) // 注意 case 后面要小括号，并且不用冒号
   {
      语句块  // 主变量=变量 1 时执行
   }
...
   case(变量 n 或常量 n)
   {
      语句块  // 主变量=变量 n 时执行
   }
   default()  // 看起来像一个函数调用
   {
```

```
            语句块   // 主变量不等于上述变量列表中的任何一个时执行
        }
    }
```

注意：　　与 C 语言相比不需要 break 语句，default() 语句块可以省略。

特别地，当某个 case()语句没有语句块时，表示穿透到下一条 case()或 default()语句，例如:

```
    switch(i)
    {
        case(1)
        {
            // i==1 时，执行的语句块
        }
        case(2)  // 穿透
        case(3)
        {
            // i==2 或 i==3 时，执行的语句块
        }
        default()
        {
            // 其它情况下执行的语句块
        }
    }
```

以上 4 类语句的语句块可以是空语句块，如

```
    if(i=0)
    {
        // 空语句块，即 i=0 时什么也不干
    }
    else
    {
        Play(VocLangSele);
    }
```

注意：　　除了 switch-case-default 语句外的其它流程控制语句，如 if-else if-else，for，while，如果语句块只有一句，则'{'和'}'可以省略。否则流程控制语句下面的语句块必须用大括号对括起来，而且 '{' 及 '}' 都必须单独一行

8 用户自定义函数

函数说明及其定义，以"function "关键字开头，

例如：
```
function CheckId(pGddm, pJymm, retNo, retInfo)
{
   // 函数实现
   return(retNo); // 返回值将自动放置在全局变量"_retVal"中
   //  当然也可以不使用返回语句
}
```

用户自定义函数在调用之前必需加以实现，但允许递归调用，递归调用注意不要形成无限循环。

所有的非常量参数都是按引用方式传递，即函数运行返回时将把参数的最终值返回给调用者。

技巧：　如果不需要某返回值，可以使用常量 0，如 CheckId(pGddm, pJymm, 0, 0);表示不要 retNo 和 retInfo 的返回值

注意：　最多只能有 40 个参数

平台有函数优化功能，对于从来没有被调用过的子函数，系统将不加载到内存中。

函数调用

与系统功能调用(操作类语句完全相同)，

例如：
```
ret = CheckId(pGddm, pJymm, retNo, retInfo);
```

调用参数可以是表达式，如：

```
function Add(a, b)
{
  return(a+b);
}
s = Add("Nun:", 8*2);  // s 的值为"Nun:16"
```

注意函数必须在调用之前加以定义（实现）。

动态函数调用

Koodo 语言支持动态函数调用，也就是支持用函数名做参数或赋值给变量。这样可以实现非常灵活的功能。

但要注意动态函数调用只有到运行时刻才知道参数个数是否相符，即调用是否合法。如果函数变量调用不合法，程序并不会终止或报错，但会在系统日志中记录一条信息。

举例如下：

```
// 定义两个变量相加的函数
function Add(v1, v2)
{
  return(v1+v2);
}
```

```
// 定义两个变量相减的函数
function Dec(v1, v2)
{
  return(v1-v2);
}

// 以函数名为参数
function Test(v1, v2, fun)
{
  ret = fun(v1, v2);  // 运行时刻才知道函数的名字
  return(ret);
}

x = 650;
y = 350;
z = Add(x, y);

f = Test;  // 函数名进行赋值
a = f(z, 2000, Add);  // 加法
b = f(z, 2000, Dec);  // 减法
return(0);
```

9 文件包含

#include <fileName> 或#include "fileName"

这个语句完全类似 C 语言，它只是在编译时使用

典型应用是将参数化变量放入配置文件；或将用户自定义函数集放在一个单独的文件，每次使用时包含它即可，这相当于用户可以构建自己的程序库。这样可以改善程序结构，增加主程序的可读性。

复杂的 IVR 应用通常由多个脚本文件组成，由主脚本文件包含其它文件。

注意：　不要形成循环包含。文件包含语句不能出现在自定义函数内部。同一个文件多次包含时，系统将给出编译警告。

10 外部扩展和动态脚本调用

外部扩展支持**外部程序调用和动态连接库调用**，具体内容参见<系统函数参考手册 七、文件及外部扩展>以及<外部动态连接库编写规范>

外部动态连接库调用如同自定义函数或系统操作函数，但必须预先说明

外部动态连接库调用对于参数有一定限制：对于返回字符串，参数必须预先分配足够的空间

对于一个函数调用，编译器按下列顺序解释：

1. 是系统操作函数吗?
2. 是用户自定义函数吗?
3. 是外部动态连接库调用函数吗?

如果上面 3 个都不是，编译期间将报告错误的操作名。

动态脚本调用

从 V1.63 版本开始增加了一个动态执行脚本的系统函数 RunBss(bss)，具体说明参见系统函数手册和其它相关文档。

动态脚本调用是个强大的机制，主脚本可以通过各种条件来动态调用子脚本，比如根据被叫号码来路由不同的流程，这些号码和脚本可以放置到配置文件里，也可以放到数据库里面，通过编写主脚本来灵活进行控制。

RunBss()执行时自动加载，这样可以实现子脚本的动态更新，动态装配。

动态脚本调用后系统会自动将它从内存中卸载，对于复杂的声讯应用，每个被叫号码对应一个不同的业务流程，自动卸载可以提高资源利用率。

动态脚本调用可以实现更高级的应用，如通过 TCP/IP 或 Http 传递一些动态脚本给主控程序，用实现动态菜单；又比如主控脚本动态解释 XML 文件（比如 VoiceXML 文件），将 XML 翻译成 Koodoo 脚本文件，再动态加载执行。

11 线路间通讯

除了采用外部数据库和 TCP/IP（包括 Http()函数）以外，最有效的方法是使用消息队列和共享内存变量，以及线路间函数调用

具体内容参见<系统函数参考手册.八、消息队列>、 <系统函数参考手册.九、共享内存变量>和<系统函数参考手册.十、线路间函数调用>。

12 Koodoo 语言和 C 语言的不同

暂时来说，除了上面讲到的，还有：

不支持 goto 语句，因为它带来的副作用远远大于给我们的好处；

不支持结构体，但可以用数组来模拟；

不支持指针，但数组别名和函数参数类似 C++的引用。

与 C 语言在程序结构上的不同：

C 语言的入口是 main()函数，而 Koodoo 脚本语言从遇到的第一条非函数说明的语句开始执行。

C 语言通常编译成二进制可执行文件，Koodoo 脚本语言则由语音平台直接编译到内存之中。