大脚蟹快速开发平台学习教程《二》：基础入门篇

上一节，我们学习了《环境搭建篇》，本篇将学习基础语法，默认所有的工具已经准备就绪中。

本文建议大家使用 **SublimeText 3.0**，不仅界面清爽美观，而且功能强大好用。

一、全世界通用的“**Hello World**”。

打开 **Demo\EX01_HelloWorld.carb**，将会显示如下图一样的程序代码。

上图中的代码，各部分解释如下：

1）用 **/* */**包起来的，是注释部分，注释是给人看的，机器编译的时候，会忽略这部分代码

2）**import system;** 这是让编译器导入库文件（注：库文件也是 **Crab** 格式的源代码文件）

3）**main {}** 这是主程序，开发者写的程序代码，将从这里开始执行。

4）**Print('Hello World');** 这一句，是在日志控制台上打印一句**'Hello World'**;

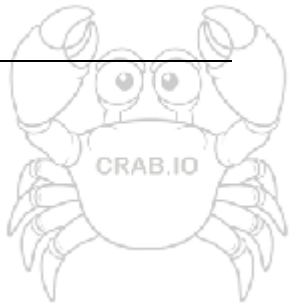
我们先不管代码是怎么样写的，而是先运行一下，看看效果。

打开 **CrabStudio**。

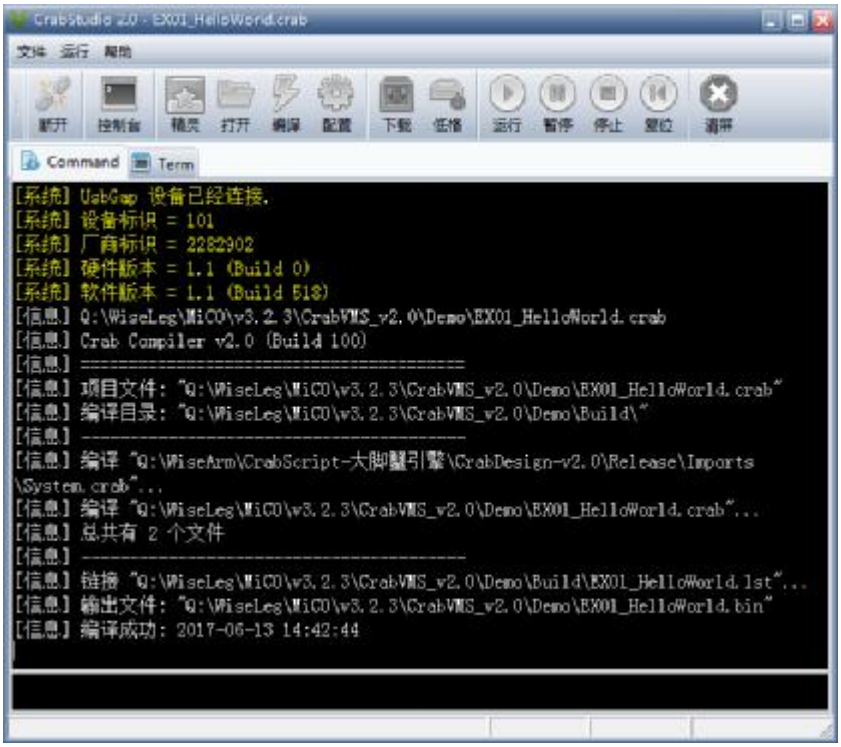
1）点击“连接”按钮，如果连接成功，下面窗口会出现连接后的设备信息。

2）点击“打开”，选择 **Demo\EX01_HelloWorld.carb** 这个文件。

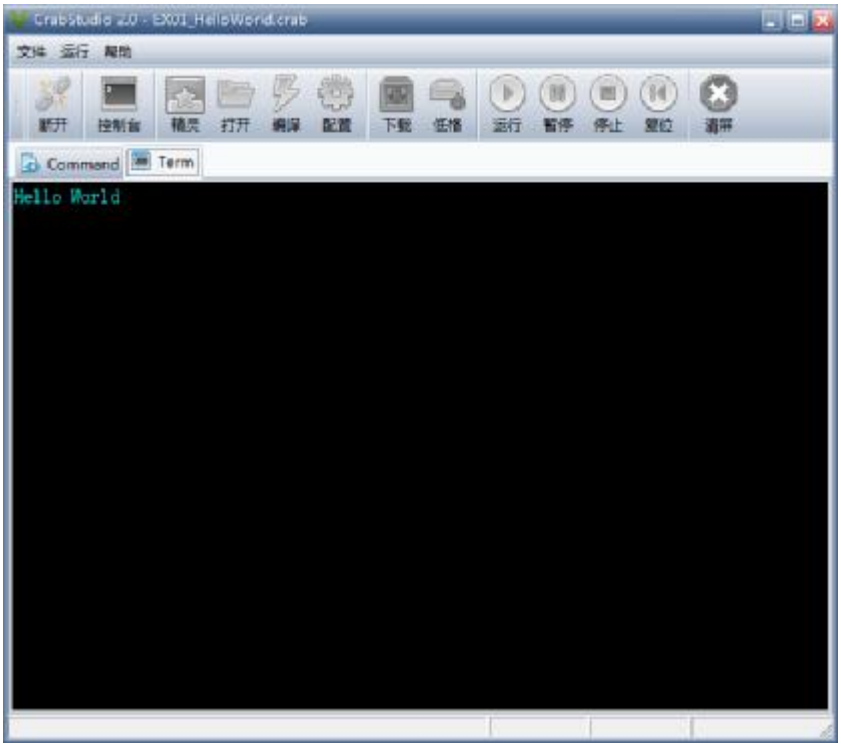
3）点击“编译”，将会出现如下图所示。



4）点击“下载”，将编译后的程序下载到开发板上。



5）点击“运行”，下方窗口将会切换到“**Term"**页面，并显示”**Hello World** “



这表示，我们的第一个程序，已经完美运行成功。

接下来，我们开始学习 **CRAB** 语言的语法部分。

二、基本语法

1）注释

注释是用于解释代码。编译器会忽略注释的条目。和大多数 **C** 系**(C/C#/C++)**语言相似，在 **CRAB** 程序中，多行注释以 **/*** 开始，并以字符 ***/** 终止，如下所示：

```
/*=======================================
  This is a simple DEMO program, it send
  a greeting of "Hello World" to
  the computer.

  这是一个最简单的DEMO程序，它向你发送一个
   "Hello World" 的问候。
=======================================*/
```

单行注释是用 **'//'** 符号表示，常用于行尾。例如：

```
Print('Hello World');  //发送 "Hello World"问候。
```

2）标识符

- 标识符是用来识别变量名、函数、类、接口或任何其它用户定义的项目。在 **CRAB** 中，标识符的命名必须遵循如下基本规则：
- 标识符必须以字母开头，后面可以跟一系列的字母、数字（ **0 - 9** ）或下划线（ _ ）。标识符中的第一个字符不能是数字。
- 标识符必须不包含任何嵌入的空格或符号，比如 **? - +! @ # % ^ & * ( ) [ ] { } . ; : " ' / **。但是，可以使用下划线（ _ ）。
- 标识符不区分大小写。大写字母和小写字母都认为是相同的字母。
- 标识符可以使用中文名称，比如：名称，我是中文变量
- 标识符不能是 **CRAB** 关键字。（注：请参阅文章最后的关键字表格）

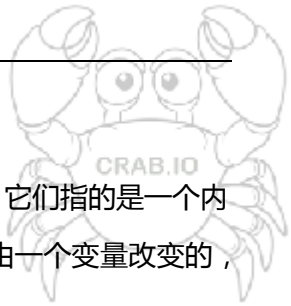
保留关键字						
and	as	boolean	break	case	catch	class
const	continue	crab	default	debug	do	double
else	enum	event	extern	false	finally	for
foreach	helper	if	import	in	interface	is
long	new	not	null	object	or	out
override	params	private	protected	public	property	refer
return	repeat	sizeof	static	string	struct	switch
this	throw	true	try	typeof	until	var
virtual	void	while	xor			

3）值类型

值类型变量可以直接分配给一个值。值类型直接包含数据。比如 **int**、**char**、**float**，它们分别存储数字、字母、浮点数。当您声明一个 **int** 类型时，系统分配内存来存储值。

下表列出了 **CRAB** 中可用的值类型：

类型	描述	范围	默认值
boolean	布尔值	True 或 False	False
byte	8 位无符号整数	0 到 255	0
char	8 位 单个字符	0 到 255	'\0'
tiny	8 位有符号整数	-127 到 127	0
short	16 位有符号整数类型	-32,768 到 32,767	0
ushort	16 位无符号整数类型	0 到 65,535	0
int	32 位有符号整数类型	-2,147,483,648 到 2,147,483,647	0
uint	32 位无符号整数类型	0 到 4,294,967,295	0L
long	64 位有符号整数类型	-923,372,036,854,775,808 到 9,223,372,036,854,775,807	0.0
ulong	64 位无符号整数类型	0 到 18,446,744,073,709,551,615	0.0
float	32 位单精度浮点型	-3.4 x 10^{38} 到 + 3.4 x 10^{38}	0
double	64 位双精度浮点型	(+/-)5.0 x 10^{-324} 到 (+/-)1.7 x 10^{308}	0
date	32 位日期型	从 0000-00-00 到 9999-12-31	0000-00-00
time	32 位时间型	从 00:00:00.000 到 23:59:59.999	00:00:00
datetime	64 位日期时间型	是 date 型和 time 型的合并体	0000-00-00 00:00:00



4）引用类型

引用类型不包含存储在变量中的实际数据，但它们包含对变量的引用。换句话说，它们指的是一个内存位置。使用多个变量时，引用类型可以指向一个内存位置。如果内存位置的数据是由一个变量改变的，其他变量会自动反映这种值的变化。内置的 引用类型有：**class**、**array** 和 **string**。

三、变量

1）变量类型

一个变量只不过是一个供程序操作的存储区的名字。在 **CRAB** 中，每个变量都有一个特定的类型，类型决定了变量的内存大小和布局。范围内的值可以存储在内存中，可以对变量进行一系列操作。

我们已经讨论了各种数据类型。**CRAB** 中提供的基本的值类型大致可以分为以下几类：

类型	举例
整数类型	tiny、byte、short、ushort、int、uint、long、ulong
浮点型	float 和 double
字符类型	char
布尔类型	true 或 false 值
日期时间型	date, time, datetime

2）变量的定义

变量定义的语法：

```
<data_type> <variable_list>;
```

在这里，**data_type** 必须是一个有效的 **CRAB** 数据类型，可以是 **char**、**int**、**float**、**double** 等等数据类型。**variable_list** 可以由一个或多个用逗号分隔的标识符名称组成。

一些有效的变量定义如下所示：

```
int i, j, k;
char c, ch;
float f, salary;
double d;
```

您可以在变量定义时进行初始化：

```
int i = 100;
```

四、常量

1）整数常量

整数常量可以是十进制、二进制或十六进制的常量。前缀指定基数：**0x** 或 **0X** 或 **$** 表示十六进制，**0b** 或 **0B** 表示二进制，没有前缀则表示十进制。

```
100     // 十进制
0x20    // 十六进制
$0c     // 十六进制
0b0101  //二进制
```

十进制整数常量也可以有后缀（不分大小写）。

后缀名	等值	示例
K	1,000	1k = 1000
M	1,000,000	1m = 1000k
G	1,000,000,000	1g = 1000m
T	1,000,000,000,000	1t = 1000g
W	10,000	1w =10k
Y	100,000,000	1y = 10000w

2）浮点常量

一个浮点常量是由整数部分、小数点、小数部分和指数部分组成。您可以使用小数形式或者指数形式来表示浮点常量。

例如：

```
3.14159      // 合法
314159E-5    // 合法
510E         // 非法：不完全指数
210f         // 非法：没有小数或指数
.e55         // 非法：缺少整数或小数
```

3）日期时间常量

一个日期常量，前缀是**"$"**，接着由单引号 **''** 或双引号 **""**包括，其值由年，月和日组成，中间有分隔符 **"-"**，或 **"/"**，一般用 **"-"**分隔符。

例如：

```
    $"2017-06-16" //常用格式
    $"2008/08/08" //兼容 DOS 格式
```

一个时间常量，前缀是**"$"**，接着由单引号 **''** 或双引号 **""**包括，其值由时，分，秒和毫秒组成，中间有分隔符 **":"**和 **"."**。秒和毫秒可以缺省不写。

例如：

```
    $"20:08:16"      //常用格式
    $"20:08:16.250" //包含毫秒格式
    $"20:08"         //仅有时和分，秒缺省为 0，毫秒也是为 0
```

一个日期时间常量，它是日期常量和时间常量的合并体，中间以空格分隔，可以相互转换。

例如：

```
    $"2017-06-16 20:08:16"      //常用格式
    $"2017-06-16 20:08:16.250" //完整格式，包含毫秒格式
    $"2017-06-16 20:08"         //仅有日期和时和分，其它缺省为 0
```

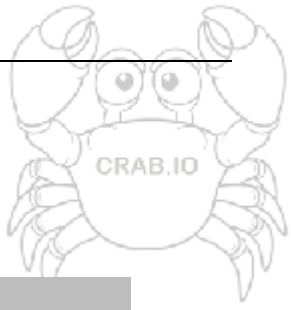
4）字符与字符串

CRAB 语言并不区分字符与字符串，这两者都统一识别为字符串。

字符串常量是括在单引号 **''** 或双引号 **""** 里，或是在引号外面使用转义符。字符串常量包含的字符，可以是：普通字符、转义序列。

例如：

```
    "Hello World"        //常用格式
    'I am aleyn.wu'     //完整格式，包含毫秒格式
    "I'm programer"     //当需要某个引号的时候，可以用另外的引号包括
    "Want return"\r\n   //在引号外连接转义符和转义序列
```



5）特殊常量

以下三个常量，属于系统原生自带的：

类型	举例
true	布尔值，字面意思是：真，是
false	布尔值，字面意思是：假，否
null	空值，仅用于引用类型变量，或判断。

6）常量的定义

常量是使用 **const** 关键字来定义的 。定义一个常量的语法如下：

```
const <constant name> = <value>;
```

- const：表达当前定义的是常量类型
- constant name：常量名称，它是一个唯一的标识符，大小写不敏感的。它不能与声明的其他标识符相同。
- value：一个常数值，它必须是一个值类型（包括字符串），或者是常量表达式。

常量的定义不需要指定常量类型，系统会自动识别常量类型。常量定义之后，也不占用内存，而且仅仅在编译期间有效。常量在定义的时候也可以使用常量表达式，也就是所有的未知数都必须是常量。

例如：

```
const PI            = 3.14159;
const LED_ON        = 1;
const KEY_PRESS     = 0x01000000;
const KEY_PRESS_F1  = KEY_PRESS + 0x11;
```

五、 运算符

运算符是一种告诉编译器执行特定的数学或逻辑操作的符号。**CRAB** 有丰富的内置运算符 ,分类如下：

1）算术运算符

下表显示了 **CRAB** 支持的所有算术运算符。假设变量 **A** 的值为 **10**，变量 **B** 的值为 **20**，则：

运算符	描述	实例
+	把两个操作数相加	A + B 将得到 30
-	从第一个操作数中减去第二个操作数	A - B 将得到 -10
*	把两个操作数相乘	A * B 将得到 200
/	分子除以分母	B / A 将得到 2
%	取模运算符，整除后的余数	B % A 将得到 0
++	自增运算符，整数值增加 1	A++ 将得到 11
--	自减运算符，整数值减少 1	A-- 将得到 9
#	字符串连接符，专用于字符串操作	"A:" # A将得到 "A:10"

2）关系运算符

下表显示了 **C#** 支持的所有关系运算符。假设变量 **A** 的值为 **10**，变量 **B** 的值为 **20**，则：

运算符	描述	实例
==	检查两个操作数的值是否相等，如果相等则条件为真。	(A == B) 为假。
!=	检查两个操作数的值是否相等，如果不相等则条件为真。	(A != B) 为真。
<>	<> 与 != 相同的功能，主要是兼容其它语言格式	(A <> B) 为真。
>	检查左操作数的值是否大于右操作数的值，如果是则条件为真。	(A > B) 为假。
<	检查左操作数的值是否小于右操作数的值，如果是则条件为真。	(A < B) 为真。
>=	检查左操作数的值是否大于或等于右操作数的值，如果是则条件为真。	(A >= B) 为假。
<=	检查左操作数的值是否小于或等于右操作数的值，如果是则条件为真。	(A <= B) 为真。

3）逻辑运算符

下表显示了 **CRAB** 支持的所有逻辑运算符。假设变量 **A** 为布尔值 **true**，变量 **B** 为布尔值 **false**，则：

运算符	描述	实例
&&	称为逻辑与运算符。如果两个操作数都非零，则条件为真。	(A && B) 为 false
and	and 和 $$ 是相同的功能，建议用 and 运算符，更直观。	(A and B) 为 false
||	称为逻辑或运算符。如果两个操作数中有任意一个非零，则条件为真。	(A || B) 为 true
or	or 和 || 是相同的功能，建议用 or 运算符，更直观。	(A or B) 为 true
!	称为逻辑非运算符。用来逆转操作数的逻辑状态。如果条件为真则逻辑非运算符将使其为假。	!B 为 true
not	not 和 ! 是相同的功能，建议用 not 运算符，更直观。	not B 为 true
^^	称为逻辑异或运算符。如果两个操作数中其中一个为真，另一个为假，则条件为真。	(A ^^ B) 为 true
xor	xor 和 ^^ 是相同的功能，建议用 xor 运算符，更直观。	(A xor B) 为 true

4）位运算符

位运算符作用于位，并逐位执行操作。

下表列出了 **CRAB** 支持的位运算符。假设变量 **A** 的值为 **60**，变量 **B** 的值为 **13**，则：

运算符	描述	实例
&	如果同时存在于两个操作数中，二进制 AND 运算符复制一位到结果中。	(A & B) 将得到 12，即为 0000 1100
|	如果存在于任一操作数中，二进制 OR 运算符复制一位到结果中。	(A | B) 将得到 61，即为 0011 1101
^	如果存在于其中一个操作数中但不同时存在于两个操作数中，二进制异或运算符复制一位到结果中。	(A ^ B) 将得到 49，即为 0011 0001
~	二进制补码运算符是一元运算符，具有"翻转"位效果，即 0 变成 1，1 变成 0。	(~A) 将得到 -61，即为 1100 0011，一个有符号二进制数的补码形式。
<<	二进制左移运算符。左操作数的值向左移动右操作数指定的位数。	A << 2 将得到 240，即为 1111 0000
>>	二进制右移运算符。左操作数的值向右移动右操作数指定的位数。	A >> 2 将得到 15，即为 0000 1111

5）赋值运算符

下表列出了 **CRAB** 支持的赋值运算符：

运算符	描述	实例
=	简单的赋值运算符，把右边操作数的值赋给左边操作数	C = A + B 将把 A + B 的值赋给 C
+=	加且赋值运算符，把右边操作数加上左边操作数的结果赋值给左边操作数	C += A 相当于 C = C + A
-=	减且赋值运算符，把左边操作数减去右边操作数的结果赋值给左边操作数	C -= A 相当于 C = C - A
*=	乘且赋值运算符，把右边操作数乘以左边操作数的结果赋值给左边操作数	C *= A 相当于 C = C * A
/=	除且赋值运算符，把左边操作数除以右边操作数的结果赋值给左边操作数	C /= A 相当于 C = C / A
%=	求模且赋值运算符，求两个操作数的模赋值给左边操作数	C %= A 相当于 C = C % A
<<=	左移且赋值运算符	C <<= 2 等同于 C = C << 2
>>=	右移且赋值运算符	C >>= 2 等同于 C = C >> 2
&=	按位与且赋值运算符	C &= 2 等同于 C = C & 2
^=	按位异或且赋值运算符	C ^= 2 等同于 C = C ^ 2
|=	按位或且赋值运算符	C |= 2 等同于 C = C | 2
#=	字符串连接且赋值运算符	C #= A 相当于 C = C # A

6）其他运算符

下表列出了 **C#** 支持的其他一些重要的运算符。

运算符	描述	实例
sizeof()	返回数据类型的大小。	sizeof(int)，将返回 4.
typeof()	返回 class 的类型。	typeof(StreamReader);
? :	条件表达式	如果条件为 true ? 则为 X : 否则为 Y
Is	判断对象是否为某一类型。	If(Ford is Car) // 检查 Ford 是否是 Car 类的一个对象。
in	判断变量是否在一个列表里	If (I in [1,2,3]) //检查 I 是否为 1,2,3 中任意一个

六、条件判断

判断结构要求程序员指定一个或多个要评估或测试的条件，以及条件为真时要执行的语句（必需的）和条件为假时要执行的语句（可选的）。

1）if 语句

一个 **if** 语句 由一个布尔表达式后跟一个或多个语句组成。语法如下：

```
if (boolean_expression)
{
   /* 如果布尔表达式为true将执行的语句 */
}
```

2）if...else...语句

一个 **if** 语句 后可跟一个可选的 **else** 语句，**else** 语句在布尔表达式为假时执行。

```
if (boolean_expression)
{
   /* 如果布尔表达式为true将执行的语句 */
}
else
{
   /* 如果布尔表达式为false将执行的语句 */
}
```

3）if 语句嵌套

if 语句可以嵌套使用，格式如下：

```
if (boolean_expression1)
{
}
else if (boolean_expression2)
{
}
else if ...
{
}
else
{
}
```

3）if (.. in..) 语句

if..in..语句，相当于把几个类似的情况，都集中一起判断，既能方便直观，又能节省代码空间。

此语句仅可单独使用，不可以嵌套使用，也不可以使用 **else** 语句。语法如下：

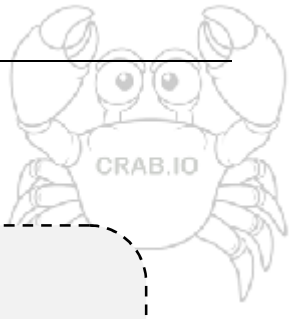
```
if (var_name in [case list])
{
   /* 如果布尔表达式为true将执行的语句 */
}
```

范例：

```
if (I in [1, 2, 3..5])
{
   /* 如果布尔表达式为true将执行的语句 */
}
```

以上语句相当以下语句的效果

```
if ((I == 1) || (I == 2) || ((I >=3) && (I <=5)))
{
   /* 如果布尔表达式为true将执行的语句 */
}
```



4）switch 语句

一个 **switch** 语句允许测试一个变量等于多个值时的情况。语法如下：

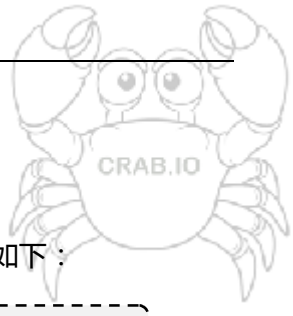
```
switch(expression)
{
    //单值的情况
    case constant-expression1 :
    {
       statement(s);
       break;  //可选项
    }
    //多值的情况
    case constant-expression2, constant-expression3 :
    {
       statement(s);
       break;  //可选项
    }
    //范围的情况
    case constant-expression4..constant-expression6 :
    {
       statement(s);
       break;  //可选项
    }

    /* 您可以有任意数量的 case 语句 */

    default : //可选项
    {
       statement(s);
       break;  //可选项
    }
}
```

- switch 语句中的 expression 必须是一个值类型或枚举类型。
- 一个 switch 中可以有任意数量的 case 语句。
- case 的 constant-expression 必须与 switch 中的变量具有相同的数据类型，且必须是常量类型。

- **case** 可以是这三种情况：单值，多值，范围，但这三种情况可以任何组合。
- 当被测试的变量等于 case 中的常量时，case 后跟的语句将被执行，直到遇到 break 语句为止，或是遇到下一个 case 开头为止。
- break 不是必须的，当前的 case 情况结束的时候，将会自动跳转到 switch 结束位置。
- 如果在 case 情况里任意位置加入 break，则运行到此位置的时候，会立刻跳转到结束位置。
- default 是以上 case 情况都不符合的时候才会运行的。它不是必须的。
- **CRAB** 不支持从一个 **case** 标签显式贯穿到另一个 **case** 标签。

范例：

```
switch(I)
{
    //单值的情况
    case 1 :
    {
       statement(s);
    }
    //多值的情况
    case 2, 3, 4 :
    {
       statement(s);
    }
    //范围的情况
    case 5..8 :
    {
       statement(s);
    }
    //多种情况混合使用
    case 9, 10, 12, 20..30 :
    {
       statement(s);
    }
    default :
    {
       statement(s);
    }
}
```



七、循环控制

1）for 语句

一个 **for** 循环是一个允许您编写一个执行特定次数的循环的重复控制结构。语法如下：

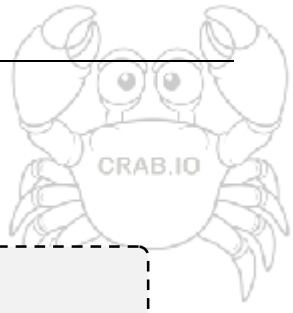
```
for ( init; condition; increment )
{
   statement(s);
}
```

下面是 **for** 循环的控制流：

- **init** 会首先被执行，且只会执行一次。这一步允许您声明并初始化任何循环控制变量。您也可以不在这里写任何语句，只要有一个分号出现即可。
- 接下来，会判断 **condition**。如果为真，则执行循环主体。如果为假，则不执行循环主体，且控制流会跳转到紧接着 **for** 循环的下一条语句。
- 在执行完 **for** 循环主体后，控制流会跳回上面的 **increment** 语句。该语句允许您更新循环控制变量。该语句可以留空，只要在条件后有一个分号出现即可。
- 条件再次被判断。如果为真，则执行循环，这个过程会不断重复（循环主体，然后增加步值，再然后重新判断条件）。在条件变为假时，**for** 循环终止。

范例：

```
for ( l = 0; l < 10; l++ )
{
   print(l);
}
```



2）while 语句

只要给定的条件为真，**while** 循环语句会重复执行一个目标语句。语法如下：

```
while(condition)
{
   statement(s);
}
```

condition 必须是布尔表达式。当条件为真时执行循环。当条件为假时，程序流将继续执行紧接着循环的下一条语句。所以在这里，**while** 循环的关键点是循环可能一次都不会执行。当条件被测试且结果为假时，会跳过循环主体，直接执行紧接着 **while** 循环的下一条语句。

范例：

```
I = 0;
while(I < 10)  //当 I<10时，继续上面的循环
{
   statement(s);
   I++;
}
```

3）do..while 语句

do...while 语句与 **while** 语句类似，不同的地方在于，**while** 语句是在循环头部测试循环条件，而 **do...while** 语句在循环的尾部检查它的条件。语法如下：

```
do
{
   statement(s);
}
while(condition);
```

do...while 语句会确保里面的语块至少会执行一次循环。

```
I = 0;
do
{
   statement(s);
   I++;
}
while(I < 10);  //当 I<10时，继续上面的循环
```

4）repeat..until 语句

repeat..until 语句与 **do..while** 语句类似， 不同的地方在于，**do..while** 语句是在测试循环条件为 **true** 时继续循环，而 **repeat..until** 语句在测试循环条件为 **true** 时结束循环。语法如下：

```
repeat
{
   statement(s);
}
until(condition);
```

同样的，**repeat..until** 语句也会确保里面的语块至少会执行一次循环。

范例：

```
repeat
{
   dosomething();
}
until error; //当error为true时，结束循环，否则继续上面的循环。
```

5）循环控制语句

循环控制语句将会更改执行的正常序列。**CRAB** 提供了 **break** 和 **continue** 两个控制语句：

break 语句有以下两种用法：

- 当 **break** 语句出现在一个循环内时，循环会立即终止，且程序流将继续执行紧接着循环的下一条语句。
- 它可用于终止 **switch** 语句中的一个 **case**。
- 如果您使用的是嵌套循环（即一个循环内嵌套另一个循环），**break** 语句会停止执行最内层的循环，然后开始执行该块之后的下一行代码。

continue 语句

- **continue** 会跳过当前循环中的代码，强迫开始下一次循环。
- 对于 **for** 循环，**continue** 语句会导致执行条件测试和循环增量部分。
- 对于 **while** 和 **do...while** 和 **repeat..until** 循环，**continue** 语句会导致程序控制回到条件测试上。

八、函数

1）常规函数

常规函数是指用户自己定义的，把一些相关的语句组织在一起，用来执行一个任务的语句块。

一个常规函数通常包括函数名称，参数，返回值，任务语句块，这四个定义内容。

2）事件函数

事件函数与普通函数很相似，不同的地方在于，一是事件函数不可以有参数，二是事件函数必须有事件标识 **ID**，三是事件函数仅提供给系统调用，用户不能直接调用。详情请参阅事件部分。

3）属性读写函数

属性读写函数是附加在属性上用于对应读和写的特殊函数。属性读写函数的名称是固定的，其中，属性读函数为 **get**，写函数为 **set**。当用户对属性进行读或写的时候，将会触发和调用相对应的读或写函数。详情请参阅属性部分。

4）接口函数

接口函数的定义部分与普通函数差不多，但前面需要加上 **extern** 指示符，而且函数头后面不可以有执行的任何语句块。接口函数是用来与宿主程序交互与沟通的特殊函数。详情请参阅接口部分。

5）类成员函数

类成员函数与普通函数一样，都有函数头，参数，任务语句块，等等。不同的地方在于，类成员函数在类结构定义里面，在调用的时候，同样需要与类实例一起配套使用。详情请参阅类部分。

6）内嵌函数

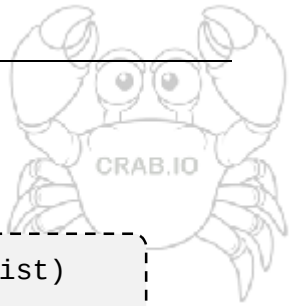
CRAB 语言有四个内嵌函数，分别是 **TypeOf**，**SizeOf**，**OpenEvent** 和 **CloseEvent** 函数。

- **TypeOf** 函数用于检测变量类型。
- **SizeOf** 函数用于检测变量大小。
- **OpenEvent** 函数用于打开全局事件驱动。
- **CloseEvent** 函数用于关闭全局事件驱动。

7）特殊函数

CRAB 语言有两个特殊函数，分别是 **main** 函数与 **setup** 函数。

- **main** 函数是应用程序的主入口，它不可缺失。
- **main** 函数没有任何参数，也不能返回任何类型的参数。
- 当 **main** 函数结束的时候，或是遇到 **return** 语句的时候，应用程序将结束它的生命周期。
- **setup** 函数是应用程序初始化函数，一般是由 **IDE** 自动生成，开发者不应自己编辑 **setup** 函数。
- **setup** 函数是可选项。如果有 **setup** 函数存在，则应用程序会先执行 **setup** 函数，然后再执行 **main** 函数。
- **setup** 函数和 **main** 函数一样，没有任何参数，也不能返回任何类型的参数。



九、常规函数

1）常规函数也是全局函数，只要定义了，后面任何地方都可以调用。语法如下：

```
<Access Specifier> <Return Type> <Method Name>(Parameter List)
{
   Method Body
}
```

- **Access Specifier**：访问修饰符，对于常规函数，此修饰符无效，所以可以忽略不写。
- **Return type**：返回类型，一个函数可以返回一个值。返回类型是函数返回的值的数据类型。如果函数不返回任何值，则返回类型为 **void**。
- **Method name**：函数名称，是一个唯一的标识符，大小写不敏感的。它不能与声明的其他标识符相同。
- **Parameter list**：参数列表，使用圆括号括起来，该参数是用来传递和接收函数的数据。参数列表是指函数的参数类型、顺序和数量。参数是可选的，也就是说，一个函数可能不包含参数。
- **Method body**：函数主体，包含了完成任务所需的指令集。

范例：

```
void MyFunc(int P1, int P2)  //这个位置，继续定义，并写函数主体。
{
   Method Body
}
```

2）常规函数可以先预定义，然后在后面的任意地方再继续写函数主体。

函数的预定义一般用于函数之间的相互调用，如果没有预定义，那前面定义的函数调用者，将无法找到后面才定义的另一个函数。

范例：

```
void MyFunc(int P1, int P2);  //函数的预定义必须在函数头的位置以; 结束

...  //这里可以是任意代码

void MyFunc(int P1, int P2)  //这个位置，继续定义，并写函数主体。
{
   Method Body
}
```

3）参数的定义

常规函数可以有任意个参数，也可以没有参数。

参数的完整定义方式和变量的定义方式差不多，惟一不同的地方就是，参数可以指定为输出方式。

```
<out> <data_type> <param_name>
```

- **out** 是可选项，它指示在函数结束时，需要把该参数的值输出给函数调用者。
- **data_type** 必须是一个有效的数据类型，可以是 **char**、**int**、**float**、**double** 等等数据类型。
- param_name 是一个唯一的标识符，大小写不敏感的。它不能与它的函数名称相同，也不能与其它已声明的参数名称相同。
- 参数与参数之间，需要用逗号来分隔。

范例：

```
void MyFunc(int P1, int P2, out int P3)
{
   Method Body
}
```

4）返回类型与返回方式

一个函数可以返回一个值，返回的类型是函数返回的值的数据类型。如果函数不返回任何值，则返回类型为 **void**。

函数的返回方式，可以使用 **return** 语句，后面带表达式，该表达式的运算结果必须与返回类型相同。还可以使用 **result** 变量，每一个函数，都有一个默认自带的与返回类型相同的变量 **result**，我们叫它返回值变量。在函数的任何地方，都可以使用 **result** 变量，而不会产生让函数立刻返回的效果。如果使用 **result** 变量，则在需要函数返回的时候，可以直接使用 **return** 语句，而不需要带一个表达式。

范例：

```
int Func1(int P1, int P2)
{
   return P1 + P2;   //直接使用return语句返回计算值
}
int Func2(int P1, int P2)
{
   result = P1 + P2;   //任何时候，都可以使用这个result变量
   return ;  //效果相当于 return result;
}
```

5）函数的调用

函数的调用是对指暂停和保存当前的任务位置，跳转到被调用的函数的位置，当执行完被调用的函数任务后，返回到当前任务的位置继续执行。函数调用的方式如下：

```
<Object Name Path><Call Method Name>(Parameter List);
```

- `Object Name Path` 是可选项，如果被调用的函数是接口函数或类成员函数，则在函数名前面需要加个接口名称或类名称或类实例名称。
- `Call Method Name` 被调用的函数名称。该名称必须是前面已定义的函数。
- `Parameter List` 需要传递给被调用的函数的参数列表，如果被调用的函数没有任何参数，则这里也不需要任何参数传入。
- 如果对应的参数有 `out` 指示符 ,则传入的参数也必须加上 `out` 指示符 ,而且该参数必须是变量，不可以是表达式或是属性。
- 如果被调用的函数有返回值，则它可以做为表达式的一份子，但它的返回值必须与表达式的类型相融合或兼容。
- 如果被调用的函数有返回值，但调用者并没有相对应的方式去保存它的返回值，则返回值将会被遗弃。

范例：

```
MyFunc(1, 2);          //调用MyFunc

int OutVar;
MyFuncWithOut(1, 2, out OutVar); //调用MyFuncWithOut,有输出参数

var R1;
R1 = Func1(1, 2);    //调用Func1，并获得函数返回值
R1 = 1 + Func2(3, 4) * 5;  //调用Func2，并将返回值当做表达式一部分

Func1(1, 2);  //调用Func1，并遗弃它的返回值。

//以下是错误行为。
R1 = MyFunc(1, 2);  //MyFunc函数并没有返回值，此调用将会失败。
```

十、数组

1）声明数组

数组是一个存储相同类型元素的固定大小的顺序集合。数组是用来存储数据的集合，通常认为数组是一个同一类型变量的集合。

声明数组变量并不是声明 **number0**、**number1**、...、**number99** 一个个单独的变量，而是声明一个就像 **numbers** 这样的变量，然后使用 **numbers[0]**、**numbers[1]**、...、**numbers[99]** 来表示一个个单独的变量。数组中某个指定的元素是通过索引来访问的。

所有的数组都是由连续的内存位置组成的。最低的地址对应第一个元素，最高的地址对应最后一个元素。

在 **CRAB** 中声明一个数组，您可以使用下面的语法：

```
datatype[] arrayName;
```

- **datatype** 用于指定被存储在数组中的元素的类型。
- **[ ]** 指定数组的大小（或长度）。如果数组有多个维度，则此处可以相应的增加，**CRAB** 语言最多支持 **3** 个维度的数组。
- **arrayName** 指定数组的名称。与变量的定义一样，它是一个唯一的标识符，大小写不敏感的，但不可以与其它已定义的标识符相同名称。

范例：

```
int[]     MyArray;   //声明一个1维数组
int[][]   MyRects;   //声明一个2维数组
```

2）初始化数组

声明一个数组不会在内存中初始化数组。当初始化数组变量时，您可以赋值给数组。

数组是一个引用类型，所以您需要使用 **new** 关键字来创建数组的实例。

范例：

```
int[]     MyArray;         //声明一个1维数组
MyArray = new int[10];   //初始化数组，并为之申请10个元素

int[]     MyArray2 = new int[20];   //声明数组，同时初始化数组。
int[][]   MyArray3 = new int[20][30]  //声明一个2维数组并初始化。
```

3）数组的访问与赋值

数组赋值需要通过使用索引号（数组下标）赋值给一个单独的数组元素。数组的索引号从 **0** 开始，最大值为初始化数组时所申请的元素个数。

范例：

```
int[]     MyArray;          //声明一个1维数组
MyArray = new int[10];   //初始化数组，并为之申请10个元素

MyArray[0] = 123;  //给数组的第一个元素（索引号为0）赋值。

int I;
for (I=0; I < 10; I++)
{
  MyArray[I] = I * 100;   //利用循环来给数组的每一个元素赋值
}
```

数组的访问规则与赋值类似。不仅如此，数组的访问还可以通过 **foreach** 语句来逐项访问。

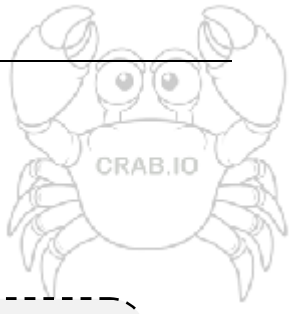
范例：

```
int[]     MyArray;          //声明一个1维数组
MyArray = new int[10];   //初始化数组，并为之申请10个元素

int Item;
foreach (Item in MyArray)
{
  Print(Item);   //逐项打印数组的每一个元素赋值

  //注意，不可以改变Item的值，因为Item的改变，并不会改变MyArray的内容。
  Item = 10;  //错误
}
```

未初始化的数组，它的内存类型为 **null**

范例：

```
int[]     MyArray;   //声明一个1维数组

if (MyArray == null) { Print("MyArray is null"); }
```



4）动态数组

数组也可以采用动态初始化的方式。

范例：

```
int[][]   MyArray;   //声明一个2维数组

//初始化数组，其中第2维组申请3个元素组，第1维不固定。
MyArray = new int[3][];

//为第2维的每一个元素组单独申请1维元素，每一组个数各自不同。
MyArray[0] = new int[20];
MyArray[1] = new int[30];
MyArray[2] = new int[40];

//同样采用索引号方式访问，多维数组索引号顺序为，高维数在前，低维数在后。
MyArray[0][1]   = 1;
MyArray[1][12]  = 50;
MyArray[2][20]  = 100;
```

5）多维数组的另一种索引号访问方式

范例：

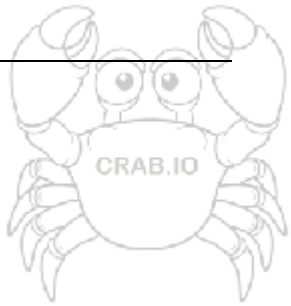
```
MyArray[0, 1]   = 1;     //  相当于MyArray[0][1]
MyArray[1, 12]  = 50;    //  相当于MyArray[1][12]
MyArray[2, 20]  = 100;   //  相当于MyArray[2][20]
```

6）函数的数组参数

在函数的参数定义里，数组参数仅需要声明数组类型和维数即可。

范例：

```
void MyArrayFunc(int[] ArrayParam)
{
   int Sum = ArrayParam[0] + ArrayParam[1];  //直接使用数组参数。
   Print("Sum = " # Sum);
}
```



十一、字符串

字符串是一个特殊的类，它在变量声明之后，无需申请内存即可直接使用。

字符串的最大长度是 **255** 个字符，最小长度是 **0**。

1）字符串变量声明

范例：

```
string MyStr;  //声明一个字符串变量
string MyStr2 = "Hello"; //声明一个字符串变量，并赋初值
```

2）字符串赋值

字符串在声明之后，还没赋值之前，他的值类型是 **null**。

可以使用任何明文的方式给字符串赋值，也可以使用字符串表达式，或转义序列。

范例：

```
string MyStr;  //声明一个字符串变量

MyStr = "Hello"; //明文字符串赋值方式。
MyStr = 10 + 20; //将普通的值类型表达式的结果赋值给字符串。
MyStr = 'GoodBye'\r\n;  //在字符串后面加上转义序列。
```

3）字符串的连接

字符串可以通过 # 运算符来连接。任何值类型的变量或表达式，只要通过 # 连接符连接到字符串表达式，那将会自动转化为字符串表达式。

范例：

```
string MyStr;  //声明一个字符串变量。

int R1 = 10 + 20;
MyStr = "Result = " # R1; //通过字符串连接符来连接字符串。

date D1 = $"2017-06-20"
MyStr = "Today is " # D1 # \r\n; //通过连接符来加入日期和转义序列。
```

3）函数定义里的字符串参数

如果函数参数里有字符串参数，则该参数将接受任何值类型参数和字符串，系统会自己根据类型自动转换。

范例：

```
void ShowStr(string Text);
{
  Print(Text);
}

string MyStr;  //声明一个字符串变量
MyStr = "Hello"; //明文字符串赋值方式。

ShowStr(MyStr);  //传入一个正规字符串参数
ShowStr(10+20);  //传入一个整型运算表达式
ShowStr(true);   //传入一个非字符串值
```