

Menthol 1.2 参考手册

Copyright @ltplayer.com Email:ltplayer@yeah.net



第一章：关于 Menthol

- 1.1. [什么是 Menthol](#)
- 1.2. [准备工作](#)
- 1.3. [开始写第一个程序](#)
- 1.4. [关键字](#)
- 1.5. [程序注释](#)

第二章：语言参考

2.1 数据类型

- 2.1.1. [基础](#)
- 2.1.2. [布尔类型](#)
- 2.1.3. [数值类型](#)
- 2.1.4. [字符串类型](#)
- 2.1.5. [数组类型](#)
- 2.1.6. [字典类型](#)
- 2.1.7. [object](#)
- 2.1.8. [null 类型](#)
- 2.1.9. [函数类型](#)
- 2.1.10. [模块类型](#)

2.2 变量

- 2.2.1. [变量基础](#)
- 2.2.2. [作用域](#)

2.3 表达式

2.4 运算符

- 2.4.1. [运算符基础](#)
- 2.4.2. [加\(+\)运算符](#)
- 2.4.3. [减\(-\)运算符](#)
- 2.4.4. [乘\(*\)运算符](#)
- 2.4.5. [除\(/\)运算符](#)
- 2.4.6. [三目\(?:\)运算符](#)
- 2.4.7. [按位“或”运算符 \(|\)](#)
- 2.4.8. [按位“与”运算符 \(&\)](#)
- 2.4.9. [逻辑“非”运算符 \(!\)](#)
- 2.4.10. [取余运算符 \(%\)](#)
- 2.4.11. [按位“异或”运算符 \(^\)](#)
- 2.4.12. [赋值运算符 \(=\)](#)
- 2.4.13. [小于运算符 \(<\)](#)
- 2.4.14. [大于运算符 \(>\)](#)
- 2.4.15. [比较运算符 \(!=\) \(<>\)](#)
- 2.4.16. [逻辑“或”运算符 \(||\)](#)
- 2.4.17. [逻辑“与”运算符 \(&&\)](#)
- 2.4.18. [逻辑“大于或等于”运算符 \(>=\)](#)
- 2.4.19. [逻辑“小于或等于”运算符 \(<=\)](#)
- 2.4.20. [逻辑“等于”运算符 \(==\)](#)
- 2.4.21. [加法赋值运算符 \(+=\)](#)
- 2.4.22. [减法赋值运算符 \(-=\)](#)
- 2.4.23. [除法赋值运算符 \(/=\)](#)
- 2.4.24. [乘法赋值运算符 \(*=\)](#)
- 2.4.25. [取余赋值运算符 \(%=\)](#)
- 2.4.26. [按位“与”赋值运算符 \(&=\)](#)
- 2.4.27. [按位“或”赋值运算符 \(|=\)](#)
- 2.4.28. [按位“异或”赋值运算符 \(^=\)](#)
- 2.4.29. [按位左移运算符 \(<<\)](#)
- 2.4.30. [按位右移运算符 \(>>\)](#)
- 2.4.31. [幂运算符 \(**\)](#)
- 2.4.32. [typeof](#)

2.5 流程控制

- 2.5.1. [基本介绍](#)
- 2.5.2. [if else](#)
- 2.5.3. [while](#)
- 2.5.4. [for ... in](#)
- 2.5.5. [try... except](#)

- 2.5.6. [throw](#)
- 2.5.7. [break](#)
- 2.5.8. [continue](#)
- 2.6 包、模块、函数
 - 2.6.1. [包的基本概念](#)
 - 2.6.2. [导入包](#)
 - 2.6.3. [模块的概念](#)
 - 2.6.4. [模块的使用](#)
 - 2.6.5. [函数基本知识](#)
 - 2.6.6. [函数定义和调用](#)
 - 2.6.7. [函数参数](#)
 - 2.6.8. [函数返回值](#)

第三章：API

- 3.1. [API 列表](#)

第四章：如何开发扩展库

- 4.1. [如何写一个扩展库](#)

第一章 关于 Menthol

1.1. 什么是 Menthol?

Menthol 是一种解释型的、面向函数的编程语言。它采用 C++ 开发而成，并且开放源码。配合简便的扩展方式，它可以满足你任何的开发需求

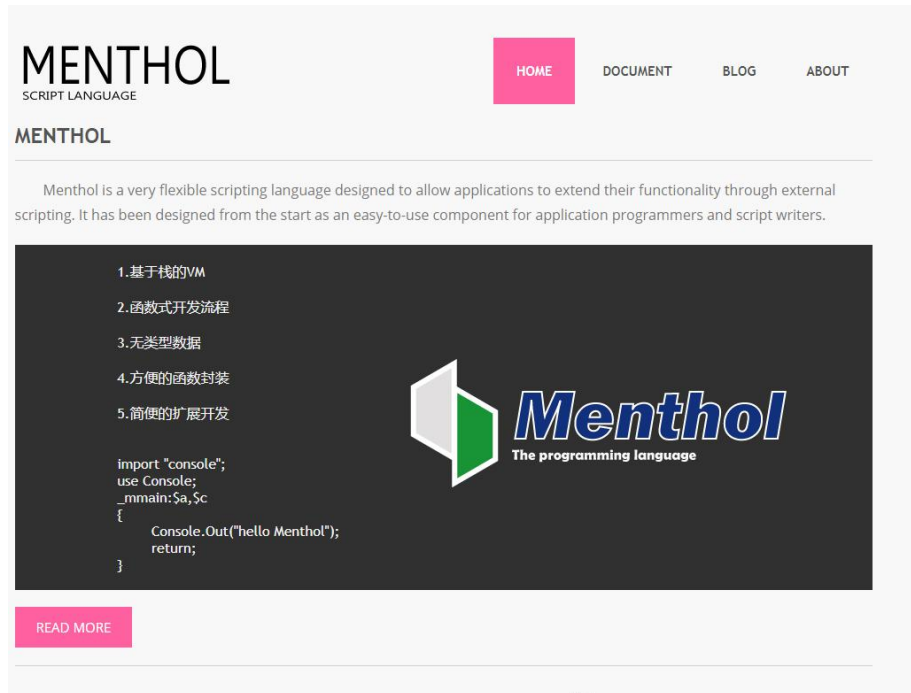
不过，Menthol 也有其局限性。你不能使用 Menthol 开发出独立的使用操作系统运行的程序，所有 Menthol 的程序都必须使用一个宿主进行解析后才可正常运行

Menthol 是一种无类型的语言。无类型意味着不必显式定义变量的数据类型。实际上你也无法上明确地定义数据类型。因为所有的类型都是在运行时才会确定的。为了开发方便，Menthol 还引入了包和模块的概念，包就是一堆模块，模块就是全局变量，函数的集合体，有点类似面向对象语言里的类的概念，但又不太相同。模块可以是你自己开发的变量函数集合体，也可以是用 C/C++ 开发出来的类库，不过在引用的时候它们没有差别

1.2. 准备工作

Menthol 是一个使用简单，易于学习的编程语言，如果你有其他编程语言，例如 C, javascript, Python 等编程语言的编写经验的话，本手册对你来说可能有点多余，你看看程序包或者源码包里的 example 中的实例就可以开始干了。其实我想说的是，Menthol 就是如此简单的一个编程语言，下面我将详细说明

首先，你要获得 menthol，访问 www.ltplayer.com/menthol.html



这个是 menthol 项目的页面，它部署于 github 上，如果你不想研究 menthol 的源码，直接点击 read more 会进入 github menthol 是项目页，在点击 release 就进入了下载页面



点第一个就可以下载 release 版本就可以了。

这个是个压缩包，解压后你会看到大概像如下的结构

example	2018/11/5 10:30
lib	2018/11/3 14:10
Compile.exe	2018/11/3 14:07
Menthol.dll	2018/11/3 14:07
Menthol.h	2018/11/1 11:02
Menthol.lib	2018/11/3 14:07
Run.exe	2018/11/3 14:07

Compile.exe 是编译程序，源码编写都要经过此程序编译后才能正常使用。Run.exe，为运行程序，编译后的源码使用此程序运行，便可以运行 menthol 程序。menthol.dll 是动态链接库，是支持 menthol 运行的核心库，menthol.lib 是静态库，如果你要开发 menthol 程序的话，需要使用这个静态库，menthol.h，头文件。

lib 文件夹下，是 menthol 自带的一些基本的类库。example 是一些实例，关于基本语法等，还包括一个使用 zplay 开发而成的简单音乐播放器

但是，如果你想研究或者自己编译 menthol 程序，就需要下载 menthol 源代码，同样进入 menthol 页面，点击 sourcecode 按钮，便会进入 menthol 在 github 上的项目仓库，不过前提需要你的电脑上安装过 vs2013 或者更高版本，因为 menthol 是在 vs2013 上开发而成，所以 vs2013 以下版本是否能正常编译，无法确定

libsrc 目录为系统自带包，在编译完 menthol 以后，需要用 menthol 编译程序编译它们，并将编译后的文件存入 lib 文件夹里，example1 为实例测试使用的库，如果你要运行测试实例，需要将它放在你运行的程序目录中，或者在程序中指定它的路径，具体可以参考 release 中的文件安排

Menthol is a very flexible scripting language

The screenshot shows a GitHub repository interface for 'lframe mod'. At the top, it displays '30 commits', '1 branch', '2 releases', and '1 contributor'. Below this, there are buttons for 'Branch: master', 'New pull request', 'Find file', and 'Clone or download'. A dropdown menu is open under 'Clone or download', showing options: 'Clone with HTTPS', 'Use Git or checkout with SVN using the web URL', a text input field containing 'https://github.com/lframe/menthol.git', 'Open in Desktop', and 'Download ZIP'. Below the dropdown, a table lists files and folders:

lframe mod	
CJson	menthol 1.0.0.1
Compile	menthol 1.0.0.1
Deelx	menthol 1.0.0.1
MArray	menthol 1.0.0.1
MDateTime	menthol 1.0.0.1
MDict	menthol 1.0.0.1
Mio	menthol 1.0.0.1
MMath	menthol 1.0.0.1
MNumber	menthol 1.0.0.1

可以点击 download zip 下载源码，这个最简单，如果有你要 clone 的话，需要 git 工具，这需要你有操作 git 工具的知识，本文档不讨论 git 文档的使用，不过 git 工具的使用并不复杂，如果使用 GUI 工具的话，使用会更加简便，读者只需简单学习便可以使用。

获得源码后，点击 menthol.sln 将会用打开。然后可以选择 release 或者 debug 编译，最后将在 debug 或者 release 文件夹中产生编译后的文件

1.3. 开始写第一个程序

一个完整的 Menthol 程序应该包括的是一个 `__mmain` 函数、导入的程序包、模块调用、模块定义。`__mmain` 函数很重要，这是启动函数，在程序被调用时，系统会自动执行这个函数，作为整个程序调用的入口，所以，它是必须要有的，否则程序无法正常启动。在 menthol 中，因为所有的函数定义必须要基于模块开发，但是，`__mmain` 函数是唯一一个不需要定义模块就能单独使用的函数，仅此一个函数。

一个最小的 Menthol 程序不要包含任何的包、模块、全局变量，便可以正常启动。但这似乎并没有什么意义。下面的代码就是一个最小的 menthol 程序

```
__mmain:$a,$c
{

}
```

但是他什么都不执行，仅仅是程序编写上的正常。

你可以把它复制到记事本或其他编辑器内，将文件保存为 `main.me` 类型。`me` 类型为 menthol 的可执行文件原文件名，取 menthol 单词的前两个字母。

现在打开控制台，并进入刚才文件所存储目录中，打开 `Compile.exe`，将刚才你存的文件作为参数跟在后面，回车执行，会生成一些调试信息，包括发生错误时，原文件的行数，以及源文件的完整路径

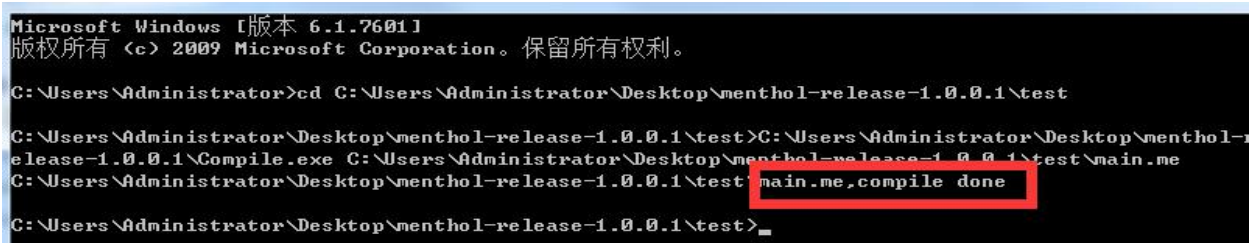


```
管理员: 命令提示符
Microsoft Windows [版本 6.1.7601]
版权所有 (c) 2009 Microsoft Corporation。保留所有权利。

C:\Users\Administrator>cd C:\Users\Administrator\Desktop\menthol-release-1.0.0.1\test

C:\Users\Administrator\Desktop\menthol-release-1.0.0.1\test>C:\Users\Administrator\Desktop\menthol-release-1.0.0.1\Compile.exe C:\Users\Administrator\Desktop\menthol-release-1.0.0.1\test\main.me_
```

好了，可以看到编译完成，你会发现在 `main.me` 下面多出来个 `main.mee`。这个便是编译以后生成的文件，`mee` 是 menthol 可执行文件的缩写 (`me execute`)



```
Microsoft Windows [版本 6.1.7601]
版权所有 (c) 2009 Microsoft Corporation。保留所有权利。

C:\Users\Administrator>cd C:\Users\Administrator\Desktop\menthol-release-1.0.0.1\test

C:\Users\Administrator\Desktop\menthol-release-1.0.0.1\test>C:\Users\Administrator\Desktop\menthol-release-1.0.0.1\Compile.exe C:\Users\Administrator\Desktop\menthol-release-1.0.0.1\test\main.me
C:\Users\Administrator\Desktop\menthol-release-1.0.0.1\test\main.me, compile done
C:\Users\Administrator\Desktop\menthol-release-1.0.0.1\test>_
```

然后我们就可以运行这个程序了，同样的操作方式，只不过要执行 `run.exe`，将刚才生成的 `main.mee` 作为参数跟在后面执行即可。因为程序内什么都没有执行，所以没有任何输出。

```
import "console";
use Console
module test
{
    def func:$str
    {
        Console.Out($str);
    }
}
__main:$a,$c
{
    test.func("!!!!!!");
}
```

然后重复上面的编译过程，然后运行

```
C:\Users\Administrator\Desktop\menthol-release-1.0.0.1\test>C:\Users\Administrator\Desktop\menthol-release-1.0.0.1\Compile.exe C:\Users\Administrator\Desktop\menthol-release-1.0.0.1\test\main.ne
C:\Users\Administrator\Desktop\menthol-release-1.0.0.1\test\main.ne, compile done

C:\Users\Administrator\Desktop\menthol-release-1.0.0.1\test>C:\Users\Administrator\Desktop\menthol-release-1.0.0.1\Run.exe C:\Users\Administrator\Desktop\menthol-release-1.0.0.1\test\main.nee
!!!!!!
```

可以看看到了输出文字。Console 是控制输入输出与文件读写的库，我们会在后面讲到。

1.4. 关键字

“关键字”是对 menthol 具有特殊含义的单词。标识符不能具有与 menthol 关键字相同的拼写和大小写,同时你也不能重新定义关键字,也不能将关键字用于定义函数名。Menthol 目前拥有 21 个关键字,下面是所有关键字

if	else	for	break	true	false	try	except
throw	continue	return	while	null	import	_mmain	def
var	in	typeof	module	use			

1.5. 程序注释

“注释”就是你不舍得删除又嫌它碍眼，或者你可能又不知道在啥时候会继续使用的一段代码的临时废除符号，使用注释符号，可以把你代码中被注释的代码在编译时候忽略掉，但你还能看见它，但编译器不会处理它，它的存在不会对程序运行起任何作用

“注释”是一个以正斜杠/星号组合（/*）开头的字符序列，到分隔符（*/）结束的一段表示。注释可以占用多行，但无法嵌套。如果你要注释单行也可以采用两个正斜杠//

```
_main:$a,$c
{
    /*var $i = 0;
    $i = 10;*/ (代码块注释)
    //$i = 10; (单行注释)
}
```

第二章：语言参考

2.1 数据类型

2.1.1 基础

Menthol 支持目前以下几种类型：

1. 数值
2. Null
3. 字符串
4. 数组
5. 字典
6. Object
7. 布尔类型
8. 函数类型
9. 模块类型

```
import "console";
use Console;

module test
{
    def func:
    {
        {
        }
    }
}

__main:$a, $c
{
    var $p = 123; //数值
    Console.Out(typeof($p)); //print M_NUMBER
    $p = null; //null
    Console.Out(typeof($p)); //print M_NULL
    $p = "string"; //字符串
    Console.Out(typeof($p)); //print M_STRING
    $p = {1, 2, 3}; //数组
    Console.Out(typeof($p)); //print M_ARRAY
    $p = (key1:"value1", key2:"value2"); //字典
    Console.Out(typeof($p)); //print M_DICT
    $p = true; //bool
    Console.Out(typeof($p)); //print M_BOOL
    $p = test.func; //函数
    Console.Out(typeof($p)); //print M_FUN
}
```

Menthol 作为无类型语言，变量类型可以随时转换类型，而不需其它任何操作

2.1.2 布尔类型

布尔类型是数据类型中最简单的类型了，就两个值，true, false，它区分大小写

声明一个布尔类型如下

```
_mmain: $a, $c
{
    Var $p = true;
}
```

布尔类型的值通常用在流程判断里，例如 while, if 语句中

```
_mmain: $a, $c
{
    Var $p = true;
    If ($p==true) {
    }
}
```

但是布尔值也是可以隐式转换的，比如：

1. 用在判断中的时候，0 表示为 false, 非 0 表位为 true
2. 用在加法时候，false 将转换为数值 0, true 将转换为数值 1
3. 用在字符串连接的时候, false 将转换为字符串 " false" , true 将转换为字符串 " true"

2.1.3 数值类型

数值可以使用十进制，十六进制，八进制表示，前面可以加上可选的符号（- 或者 +）。要使用八进制表达，数字前必须加上 0（零）。要使用十六进制表达，数字前必须加上 0x。

```
_mmain:$a, $c
{
    var $p = 066; //八进制
    $p = 0xFF; //十六进制
    $p = 123;
    $p = -123; //负数
    $p = 33.648;
}
```

数值类型的表示范围为 $-2^{1024} \sim +2^{1024}$ ，如果超过这个范围，程序将会报错。

在表示布尔判断的时候，0 将表示为 false，非 0 将表示为 true

```
import "console";
use Console;
_mmain:$a, $c
{
    if(-1223){
        Console.Out("true");
    }
    if(0){
        Console.Out("false");
    }
}
```

2.1.4 字符串类型

一个字符串 `string` 就是由一系列的字符组成，其中每个字符等同于一个字节

定义一个字符串的最简单的方法是用双引号（`“”`）把它包围起来。如果`””`中没有任何字符，则称之为空字符串，它代表`“”`中只有一个没有字符，从计算机本身的角度来讲空字符串中也有要给字符是`\0`，ascii 为 0

```
import "console";
use Console;
__main:$a, $c
{
    Console.Out("this is a string");
    var $p = "this is a string";
    Console.Out($p);
}
```

要表达一个单引号自身，需在它的前面加个反斜线（`\`）来转义。要表达一个反斜线自身，则用两个反斜线（`\\`）。其它任何方式的反斜线都会被当成反斜线本身：也就是说如果想使用其它转义序列例如 `\r` 或者 `\n`，并不代表任何特殊含义，就单纯是这两个字符本身。

转义符列表

转移符	含义
<code>\n</code>	换行
<code>\r</code>	回车
<code>\b</code>	退格
<code>\"</code>	双引号

字符串的连接通过加号（`+`）来实现，可以将两个或多个字符串连接在一起成为一个新的字符串

```
import "console";
use Console;
__main:$a, $c
{
    Console.Out("this is a string"+"1"+"2");//print this is a string12
}
```

事实上。字符串也是一个数组，关于数组的操作可以应用于字符串中，有关数组的定义及操作方式，下一节将会介绍

2.1.5. 数组类型

所谓数组，是有序的元素序列。若将有限个类型相同的变量的集合命名，那么这个名称为数组名。组成数组的各个变量称为数组的分量，也称为数组的元素

Menthol 定义数组使用括号定义([]), 在括号中，包含若干用逗号(,)分给的元素。如果括号中没有元素，则这个数组为 0 个元素的数组。数组元素可以是 Menthol 中的任何类型、表达式。

```
_mmain:$a,$c
{
    var $p = [];
    $p = [1,2,3,3+2,true];
}
```

如果要获取数组元素或设置数组元素的新值，可以使用[]来实现，[]是填写数组的索引值，索引值也称为下标，从 0 开始，0 表示数组的第一个元素

```
_mmain:$a,$c
{
    var $arr = ["a","b","c","d","e","f"];
    $a = $arr[1..]; // $a is ["b","c","d","e","f"]
    $a = $arr[..3]; // $a is ["a","b","c","d"]
    $a = $arr[2..4]; // $a is ["c","d","e"]
    $a = $arr[1]; $a is b
}
```

如果要设置也可以用[]方式来表示，如果设置一个大于当前数组索引的值，则这个索引以前的元素会被自动填充为 null

```
_mmain:$a,$c
{
    var $arr = [];
    $arr[0] = 1; // $arr is [1]
    $[3] = 2; // $arr is [1,null,null,2]
}
```

在上一节讲道字符串也是数组的时候说过，数组的操作方式也适用于字符串

```
_mmain:$a,$c
{
    var $arr = "abcdef";
    $a = $arr[1..]; // $a is "bcdef"
    $a = $arr[..3]; // $a is "abcd"
    $a = $arr[2..4]; // $a is "cde"
    $a = $arr[1]; $a is b
}
```

2.1.6. 字典类型

字典是另一种可变容器模型，且可存储任意类型对象。

字典的每个键值 `key, value` 对用双冒号 `::` 分割，每个键值对之间用逗号 `,` 分割，整个字典包括在括号 `()` 中，在字典中，键是唯一的，但值可以重复，键必须是字符串类型，但值可以是任意类型

```
_mmain:$a,$c
{
    var $str = "this is a string";
    var $dict = (key1::-111,key2::$str);
}
```

值可以设置任何数据类型，但键必须是不可变的。

字典的取值和设置方式同样也是通过 `key, value` 用双冒号分割 `::`

```
_mmain:$a,$c
{
    var $str = "this is a string";
    var $dict = (key1::-111,key2::$str);
    Var $v = $dict::key1; //print -111
    $dict::key1 = -222;
    $v = $dict::key1; //print -222
}
```

字典的便利可以通过 `for in` 方式

```
import "console";
use Console;
_mmain:$a,$c
{
    Var $arr = (key1::"value1",key2::"value2");
    for(var $key,$value in $arr)
    {
        Console.Out($key+":"+value);
    }
}
```

如果要设置一个不存在的 `key`，则会将新设置的 `key` 加入到字典中，如果取一个不存在的值则会返回 `null`

```
import "console";
use Console;
_mmain:$a,$c
{
    Var $arr = (key1::"value1",key2::"value2");
    Console.Out($arr::key3); //print null
    $arr::key3=333;
    Console.Out($arr::key3); //print 333
}
```


2.1.7. object

Object 通常用在和外部扩展库的交互上，例如 C/C++ 返回了指针类型，则在 menthol 程序中表示为 Object 类型，但在 menthol 源码中，不会使用这种类型

2.1.8. null 类型

null 是一个特殊的数据类型, null 值表示一个变量没有值。其实际含义就是部署于任何类型

在下列情况下一个变量被认为是 NULL:

1. 被赋值为 NULL。
2. 未被赋值。

```
_mmain:$a, $c
{
    Var $arr =null;
}
```

2.1.9. 函数类型

函数类型不是说定义一个函数，而是说把函数看做一个变量类型，例如把函数当做另一个函数的参数传入，通常被称为回调函数

```
import "console";
use Console;
module test
{
    def callback:$i
    {
        Console.Out($i);
    }

    def func:$fun,$i
    {
        $fun($i);
    }
}
__main:$a,$c
{
    test.func(test.callback,1000);
}
```

2.1.10. 模块类型

模块类型和函数类型比较相似，可以把一个模块定义为一个变量

```
import "console";
use Console;
module test
{
    def callback:$i
    {
        Console.Out($i);
    }

    def func:$fun, $i
    {
        $fun($i);
    }
}
__main:$a, $c
{
    var $m = test;
    $m.func($m.callback, 1000);
}
```

2.2 变量

2.2.1. 变量基础

menthol 中的变量用一个 var 关键字空格，变量名来表示一个变量。在 menthol 中变量分为全局变量和局部变量两种，全局变量为在一个模块内可以发生作用的变量，用@为前缀，局部变量为一个函数内或者一个作用域内发生作用的变量，用\$符号后面跟变量名来表示。变量名是区分大小写的。一个有效的变量名由字母或者下划线开头，后面跟上任意数量的字母，数字，或者下划线

```
module test
{
  var @global;
}
_mmain:$a,$c
{
  vaar $v1=123;
  var $_v2 = "aaaa";
  {
    Var $v3 = 666;
  }
}
```

变量默认总是传值赋值。那也就是说，当将一个表达式的值赋予一个变量时，整个原始表达式的值被赋值到目标变量。这意味着，例如，当一个变量的值赋予另外一个变量时，改变其中一个变量的值，将不会影响到另外一个变量

```
import "console";
use Console;
_mmain:$a,$c
{
  var $arr =(key1:"value1",key2:"value2");
  var $n = $arr;
  Console.Out($n::key1); //print value1
  $n = 4;
  Console.Out($arr::key1); //print value1
}
```

变量的声明，可以不要初始化，如果不初始化则变量被默认声明为 null，全局变量和局部变量都一样

```
import "console";
use Console;
_mmain:$a,$c
{
  var $n ;
  Console.Out($n); //print null
}
```

变量的声明可以是一行一个，也可以在一个 var 关键字后声明多个变量，并且遵循可以初始化也可以不初始化的原则

```
import "console";
use Console;
_mmain:$a,$c
{
  var $n=333,$f,$t = 666 ;
  Console.Out($n); //print 333
  Console.Out($f); //print NULL
  Console.Out($t); //print 666
}
```

一个重要原则，任何变量的使用(调用或给其赋值)，在使用前必须优先被声明，如果未声明时而使用或者在使用后再声明，则编译程序将会报错

```
_mmain:$a,$c  
{  
    $n = 666; //error  
    Var $n;  
}
```

2.2.2. 作用域

对于 menthol 来说作用域一般指一个模块、运行程序，以及花括号“{}”，变量的范围即它定义的上下文背景（也就是它的生效范围）。

```
import "console";
use Console;
__main:$a,$c
{
    var $f;
    {
        var $g;
    }
    {
        Console.Out($g); //错误，在本作用域内没有发现$g
        Console.Out($f); //ok
    }
}
```

对于全部变量，他的作用域在整个程序运行范围内，也就是说，在程序启动后，全部变量不论是不是在本包内，都是可见的，他的生存期是在整个程序启动到结束

对于函数参数，它的作用域就是本函数内，出了本函数，参数就失效了。

对于 for in 中的临时变量生命，它只在 for in 的循环体内

```
__main:$a,$c
{
    for(var $g in [1,2,3]){
        $g;
    }
    $g;//错误，$g 为临时变量
}
```

2.3 表达式

对大多数语言来说，程序就是一堆表达式的集合。在 menthol 中，任何的语句基本都叫做表达式，最简单的表达式就是语句的结束符“;”，这就是个表达式，没有任何意思。再比如随便一个数组，数字，字符串，声明、赋值语句等，都可以叫做表达式

```
__mmain: $a, $c
{
    123456;
    "asdfasdf";
    $a = 666;
    var $t = "fffff";
    1+2;
    1*2;
    "aaaaaaa"+"bbbbbbbbbb";
    $a+$c;
    [];
    [1, 2, 3];
}
```

但是，一定要注意，任何一个表达式，它的结尾都要有结束符“;”，否则会被编译器认为存在语法错误。表达式一行可以写多个，但需要在每个表达式后面写结束符

不过表达式本身是可以“,”分割的，例如

```
$a = 1, 2, 3;
```

类似这种的也是合法的表达式，但是上边的表达式，只会返回 1 赋值给 \$a。所以对于声明变量，一下声明都是正确的

```
module test
{
    var @g1, @g2=1, @g3;//ok
}
__mmain: $a, $c
{
    var $g1, $g2=1, $g3;//ok
    test.@g1=test.@g2=test.@g3=2;
}
```


2.4 运算符

2.4.1 运算符基础

运算符是可以通过给出的一或多个值（用编程行话来说，表达式）来产生另一个值（因而整个结构成为一个表达式）的东西。

运算符可按照其能接受几个值来分组。一元运算符只能接受一个值，例如！（逻辑取反运算符）二元运算符可接受两个值，例如熟悉的算术运算符+（加）和-（减），大多数 PHP 运算符都是这种。最后是唯一的三元运算符？：，可接受三个值；通常就简单称之为“三元运算符”（尽管称之为条件运算符可能更合适）。

Menthol 支持目前以下运算符：

- * / () ; , | & ? [] ! % ^ : :: .. = < > != <> || && >= <= == += -= /= *= %= &= |= ^= << >> **
typeof

2.4.2. 加法运算符

将数字表达式的值加到另一数字表达式上，或连接两个字符串，数组等

```
result = expression1 + expression2
```

```
import "console";
use Console;
__main:$a,$c
{
var $arr = [1,2,3,4,5,6];
$arr = 888+$arr;
$a = null;
$a = $a+true;
Console.Out($a);
}
```

expression1, expression2 为表达式。

1. expression1, expression2 同为数字时，相加
2. expression1, expression2 同为 bool 时，先转换 bool 值为数字，true 转换为 1，false 转换为 0，然后相加
3. expression1, expression2 同为 string 时，连接
4. expression1, expression2 有一个为 array 时，组合为新的 array
5. expression1, expression2 有一个为 string，另一个数字转字符串，bool 转 true, false, null 转空
6. expression1, expression2 有一个为 bool 时，先转换 bool 值为数字，true 转换为 1，false 转换为 0，然后相加
7. expression1, expression2 有一个为 null 时，另一个如果是字符，null 转换为空字符串，如果是数字时，true 转换为 1，false 转换为 0

2.4.3. 减法运算符

从一个表达式的值中减去另一个表达式的值，只有一个表达式时取其相反数。

```
result = expression1 - expression2
```

```
result = -expression1
```

```
import "console";
use Console;
__main:$a,$c
{
$a =-6;
Console.Out($a);
$a = null;
Console.Out($a-3);
}
```

expression1, expression2 为表达式。null 时, 转换为 0, true 转换为 1, false 转换为 0

除以上类型外, 其他类型进行详见操作, 则会报错

2.4.4. 乘(*)运算符

两个表达式的值相乘。

```
result = expression1 * expression2
```

```
import "console";
use Console;
_!main:$a, $c
{
    Console.Out(3*3);
}
```

expression1, expression2 为表达式。null 时, 转换为 0, true 转换为 1, false 转换为 0

除以上类型外, 其他类型进行详见操作, 则会报错

2.4.5. 除法(/)运算符

两个表达式的值相除。

```
result = expression1 / expression2
```

```
import "console";
use Console;

__main:$a,$c
{
$a = 3;
Console.Out($a/3);
}
```

expression1, expression2 为表达式。null 时, 转换为 0, true 转换为 1, false 转换为 0

除以上类型外, 其他类型进行详见操作, 则会报错

2.4.6. 三目(?:)运算符

根据条件执行两个语句中的其中一个。

```
test ? expression1 : expression2
```

test: bool 表达式语句 expression1, expression2 为表达式.

当 test 为 true 则执行 expression1, 否则执行 expression2

```
import "console";
use Console;
_mmain:$a,$c
{
    Console.Out(3>1?3:1); //print 3
    Console.Out(3<1?3:1); //print 1
}
```

2.4.7. 按位“或”运算符 (|)

对两个表达式执行按位“或”

```
result = expression1 | expression2
```

expression1, expression2 为表达式

运算符查看两个表达式的二进制表示法的值，并执行按位“或”操作。该操作的结果如下所示：

```
0101      (expression1)
```

```
1100      (expression2)
```

```
-----
```

```
1101      (结果)
```

任何时候，只要任一表达式的一位为 1，则结果的该位为 1。否则，结果的该位为 0。

2.4.8. 按位“与”运算符 (&)

对两个表达式执行按位“与”

```
result = expression1 & expression2
```

expression1, expression2 为表达式

运算符查看两个表达式的二进制表示法的值，并执行按位“与”操作。该操作的结果如下所示：

```
0101      (expression1)
```

```
1100      (expression2)
```

```
-----
```

```
0100      (结果)
```

任何时候，只要任一表达式的一位为 1，则结果的该位为 1。否则，结果的该位为 0。

2.4.9. 逻辑“非”运算符 (!)

对一个表达式执行逻辑非。

```
result = !expression
```

Expression 任何表达式。

```
import "console";
use Console;
__main:$a, $c
{
    Console.Out(!false); //print true
    Console.Out(!true); //print false
}
```

Expression 为 null 则转换为 false, 如果是数字, 则 0 为 false, 非 0 为 true, 其他类型则 true

2.4.10. 取余运算符 (%)

一个表达式的值除以另一个表达式的值，返回余数。

```
result = number1 % number2
```

expression1, expression2 为表达式。null 时, 转换为 0, true 转换为 1, false 转换为 0

除以上类型外，其他类型进行详见操作，则会报错

```
import "console";
use Console;

_mmain:$a,$c
{
    Console.Out(19.9%7); //print 5
}
```

取余（或余数）运算符用 number1 除以 number2，会将两个操作数做取整操作后，再取余数，然后只返回余数作为 result。

2.4.11. 按位“异或”运算符 (^)

对两个表达式执行按位“异或”

```
result = expression1 ^ expression2
```

expression1, expression2 为表达式

运算符查看两个表达式的二进制表示法的值，并执行按位“异或”操作。该操作的结果如下所示：

```
0101      (expression1)
```

```
1100      (expression2)
```

```
-----
```

```
1001      (结果)
```

任何时候，只要任一表达式的一位为 1，则结果的该位为 1。否则，结果的该位为 0。

2.4.12. 赋值运算符 (=)

给变量赋值

```
result = expression
```

expression 任何表达式。

= 运算符和其他运算符一样，除了把值赋给变量外，使用它的表达式还有一个值。这就意味着可以象下面这样把赋值操作连起来写：

```
j = k = l = 0;
```

执行完该例子语句后，j、k、和 l 的值都等于零。但上面的写法，仅限于赋值，初始化不可以

2.4.13. 小于运算符 (<)

两个表达式的小于比较。

result = expression1 < expression2

```
import "console";
use Console;
__main:$a, $c
{
    Console.Out(3<true);
}
```

expression1, expression2 为表达式。null 时, 转换为 0, true 转换为 1, false 转换为 0. 两个字符串比较, 遵循 C 的 strcmp 原则

除以上类型外, 其他类型进行详见操作, 则会报错

2.4.14. 大于运算符 (>)

两个表达式的大于比较。

```
result = expression1 > expression2
```

```
import "console";
use Console;
__main:$a,$c
{
    Console.Out(3>true);
}
```

expression1, expression2 为表达式。null 时, 转换为 0, true 转换为 1, false 转换为 0. 两个字符串比较, 遵循 C 的 strcmp 原则

除以上类型外, 其他类型进行详见操作, 则会报错

2. 4. 15. 比较运算符 (!=) (<>)

两个表达式的大于比较。

```
result = expression1 <>(!=) expression2
```

```
import "console";
use Console;
__main:$a, $c
{
    Console.Out(3!=true);
}
```

expression1, expression2 为表达式。null 时, 转换为 0, true 转换为 1, false 转换为 0. 两个字符串比较, 按照逐字符比较

除以上类型外, 其他类型进行详见操作, 则会报错

2.4.16. 逻辑“或”运算符 (||)

对两个表达式执行逻辑“或”。

```
result = expression1 || expression2
```

expression1, expression2 为表达式。null 时, 转换为 false, 数字 0 转换为 false, 非 0 为 true, 其他类型为 true

True	True	True
True	False	True
False	True	True
False	False	False

2.4.17. 逻辑“与”运算符 (&&)

对两个表达式执行逻辑“与”。

```
result = expression1 && expression2
```

```
import "console";
use Console;
__main: $a, $c
{
    Console.Out(3&&true);
}
```

expression1, expression2 为表达式。null 时, 转换为 false, 数字 0 转换为 false, 非 0 为 true, 其他类型为 true

当且仅当两个表达式的值都等于 True 时, result 才是 True。如果任一表达式的值等于 False, 则 result 为 False。

2.4.18. 逻辑“大于或等于”运算符 (>=)

result = expression1 >= expression2

```
import "console";
use Console;
__main: $a, $c
{
    Console.Out(3>=true);
}
```

expression1, expression2 为表达式。null 时, 转换为 0, true 转换为 1, false 转换为 0. 两个字符串比较, 按照逐字符比较

2.4.19. 逻辑“小于或等于”运算符 (<=)

result = expression1 <= expression2

```
import "console";
use Console;
__main:$a, $c
{
    Console.Out(3<=true);
}
```

expression1, expression2 为表达式。null 时, 转换为 0, true 转换为 1, false 转换为 0. 两个字符串比较, 按照逐字符比较

2.4.20. 逻辑“等于”运算符 (==)

result = expression1 == expression2

```
import "console";
use Console;
__main: $a, $c
{
    Console.Out(3==true);
}
```

expression1, expression2 为表达式。null 时, 转换为 0, true 转换为 1, false 转换为 0. 两个字符串比较, 按照逐字符比较

2. 4. 21. 加法赋值运算符 (+=)

将变量值与表达式值相加，并将和赋给该变量。

```
result += expression
```

```
import "console";
use Console;
__main: $a, $c
{
    Var $p = 1;
    $p+=1;
    Console.Out($p); //print 2
}
```

expression1, expression2 为表达式。null 时, 转换为 0, true 转换为 1, false 转换为 0. 两个字符串, 则进行连接操作

使用本运算符与这样指定完全相同:

```
result = result + expression
```

2. 4. 22. 减法赋值运算符 (-=)

将变量值与表达式值相减，并将和赋给该变量。

```
result -= expression
```

```
import "console";
use Console;
__main:$a, $c
{
    Var $p = 1;
    $p-=1;
    Console.Out($p);//print 0
}
```

expression1, expression2 为表达式。null 时, 转换为 0, true 转换为 1, false 转换为 0. 使用本运算符与这样指定完全相同:

```
result = result - expression
```

2. 4. 23. 除法赋值运算符 (/=)

将变量值与表达式值相除，并将和赋给该变量。

```
result /= expression
```

```
import "console";
use Console;
__main:$a, $c
{
    Var $p = 9;
    $p/=3;
    Console.Out($p);//print 3
}
```

expression1, expression2 为表达式。null 时, 转换为 0, true 转换为 1, false 转换为 0. 使用本运算符与这样指定完全相同:

```
result = result / expression
```

2. 4. 24. 乘法赋值运算符 (*=)

将变量值与表达式值相乘，并将和赋给该变量。

```
result *= expression
```

```
import "console";
use Console;
__main:$a, $c
{
    Var $p = 9;
    $p*=3;
    Console.Out($p);//print 27
}
```

expression1, expression2 为表达式。null 时, 转换为 0, true 转换为 1, false 转换为 0.

使用本运算符与这样指定完全相同:

```
result = result * expression
```


2.4.25. 取余赋值运算符 (%=)

result %= expression

```
import "console";
use Console;

__main:$a, $c
{
    Var $p = 9;
    $p%=3;
    Console.Out($p);//print 0
}
```

expression1, expression2 为表达式。null 时, 转换为 0, true 转换为 1, false 转换为 0.

使用本运算符与这样指定完全相同:

```
result = result % expression
```

2.4.26. 按位“与”赋值运算符 (&=)

result &= expression

```
import "console";
use Console;
__main:$a, $c
{
    var $p =1;
    $p&=1;
    Console.Out($p);//print 1
}
```

expression1, expression2 为表达式。null 时, 转换为 0, true 转换为 1, false 转换为 0.

使用本运算符与这样指定完全相同:

result = result & expression

2.4.27. 按位“或”赋值运算符 (|=)

result |= expression

expression1, expression2 为表达式。null 时, 转换为 0, true 转换为 1, false 转换为 0.

```
import "console";
use Console;
__main:$a, $c
{
    var $p = 1;
    $p |= 1;
    Console.Out($p); // print 1
}
```

使用本运算符与这样指定完全相同:

result = result | expression

2. 4. 28. 按位“异或”赋值运算符 (^=)

result ^= expression

```
import "console";
use Console;
__main:$a, $c
{
    var $p =1;
    $p ^=1;
    Console.Out($p);//print false
}
```

expression1, expression2 为表达式。null 时, 转换为 0, true 转换为 1, false 转换为 0. 其他类型则转换为 0

使用本运算符与这样指定完全相同:

result = result ^ expression

2.4.29. 按位左移运算符 (<<)

result = expression1 >> expression2

```
import "console";
use Console;
__main:$a,$c
{
  Console.Out(1<<1);//print 2
}
```

expression1, expression2 为表达式。null 时, 转换为 0, true 转换为 1, false 转换为 0. 其他类型则转换为 0

运算符把 expression1 的所有位向右移 expression2 指定的位数。expression1 的符号位被用来填充右移后左边空出来的位。向右移出的位被丢弃

2.4.30. 按位右移运算符 (>>)

result = expression1 >> expression2

```
import "console";
use Console;
__main:$a,$c
{
    Console.Out(1>>1); //print 0
}
```

expression1, expression2 为表达式。null 时, 转换为 0, true 转换为 1, false 转换为 0. 其他类型则转换为 0

<< 运算符把 expression1 的所有位向左移 expression2 指定的位数。

2.4.31. 幂运算符 (**)

result = expression1 ** expression2

```
import "console";
use Console;
__main:$a,$c
{
    Console.Out(3**3);//print 27
}
```

expression1, expression2 为表达式。null 时, 转换为 0, true 转换为 1, false 转换为 0.

除以上类型外, 其他类型进行详见操作, 则会报错

2. 4. 32. typeof

```
result = typeof(expression)
```

返回 expression 的类型

2.5 流程控制

2.5.1. 基础介绍

menthol 中的语句一般是按照写的顺序来运行的。这种运行称为顺序运行，是程序流的默认方向。

与顺序运行不同，另一种运行将程序流转换到脚本的另外的部分。也就是，不按顺序运行下一条语句，而是运行另外的语句。

要使脚本可用，该控制的转换必须以逻辑方式执行。程序控制的转换是基于一个“决定”，这个“决定”结果是真或假（返回 Boolean 型 true 或 false）。创建一个表达式，然后测试其是否为真。主要有两种程序结构实现本功能。

Menthol 使用三种流程控制方式

If ,if else ,while,for in

2.5.2. if else

根据一个表达式的值，有条件地执行一组语句。

```
if (condition) {
    statement1
}
[else {
    statement2}]
```

condition 必选项。一个 Boolean 表达式

statement1 可选项。condition 是 true 时要执行的语句。可以是复合语句。statement2 可选项。condition 是 false 时要被执行的语句。可以是复合语句。

```
import "console";
use Console;
__main:$a,$c
{
    if(1>2) {
        Console.Out(1);
    } else
    {
        Console.Out(2);
    }
}
```

If else 流程的 else 部分是可选的，可以不写，只写 if 也是可以的

```
import "console";
use Console;
__main:$a,$c
{
    if(1>2) {
        Console.Out(1);
    }
}
```

2.5.3. while

执行一个语句，直到指定的条件为 false。

```
while (expression){  
    statements  
  
}
```

expression 必选项。Boolean 表达式，在循环的每次迭代前被检查。如果 expression 是 true，则执行循环。如果 expression 是 false，则结束循环。

statements 可选项。expression 是 true 时要执行的一个或多个语句。

```
import "console";  
use Console;  
_mmain:$a,$c  
{  
    $a = 0;  
    while($a<10){  
        Console.Out($a);  
        $a = $a+1;  
    }  
}
```

表达式的值在每次开始循环时检查，所以即使这个值在循环语句中改变了，语句也不会停止执行，直到本次循环结束。有时候如果 while 表达式的值一开始就是 FALSE，则循环语句一次都不会执行。

2.5.4. for in

对应于一个对象的每个，或一个数组的每个元素，执行一个或多个语句。

```
for (var variable,variable.... in [dict | array]){  
  
}
```

variable 必选项。一个变量，它可以是 object 的任一属性或 array 的任一元素。

dict, array 要在其上遍历的字典或数组。

statement 可选项。相对于 object 的每个属性或 array 的每个元素，都要被执行的一个或多个语句。

```
import "console";  
use Console;  
_mmain:$a,$c  
{  
    for(var $t,$s in (key1::1,key2::2))  
    {  
        Console.Out($t + ":" + $s);  
    }  
    for(var $t in [0,1,2])  
    {  
        Console.Out($t);  
    }  
}
```

在循环的每次迭代前，variable 被赋予 object 的下一个属性或 array 的下一个元素。然后可以在循环内的任一语句中使用它，就好像正在使用 object 的该属性或 array 的该元素一样。

如果结合是 dict 则 variable 至少需要两个，一个是 key，一个是值

2.5.5. try except

错误处理

```
try {  
    statements1  
  
}except:$arg1... {  
  
    statements2  
  
}
```

statements 可选. 在执行 statements1 内容时, 如果有 throw 抛出, 则跳转至 statements2, \$arg1... 为抛出错误时的参数可有多个

```
import "console";  
use Console;  
_mmain:$a,$c  
{  
    Try{  
        Throw 111,222  
    }  
    Except:$arg1,$arg2  
    {  
        Console.Out($arg1) //print 111  
        Console.Out($arg1) //print 222  
    }  
}
```

2.5.6 throw

抛出错误

Throw \$arg, \$arg...

```
import "console";
use Console;
__main: $a, $c
{
    Try{
        Throw 111, 222
    }
    Except: $arg1, $arg2
    {
        Console.Out($arg1) //print 111
        Console.Out($arg1) //print 222
    }
}
```

2.5.7 break

跳出一个循环，用在 for in 和 while 中

Break 是无条件的，碰到 break 就会跳出循环

```
__main:$a,$c
{
    $a = 0;
    while($a<10){
        Break;
    }
}
```

2.5.8 continue

继续一个循环，用在 for in 和 while 中

continue 是无条件的，碰到 continue 就会跳会循环开始位置

```
_mmain:$a,$c
{
    $a = 0;
    while($a<10){
        continue
    }
}
```


2.6 包

2.6.1 包的基本概念

在 menthol 中，包分为两个类型，一个类型是用 menthol 写的包，另一个是用 C/C++写的扩展包

menthol 包的就是一个编译后的 menthol 文件或者是一个用 C/C++写的库，Menthol 包的扩展名是 .mep, 不过一个包从代码上来说和 .me 文件写法完全一样。包被编译好后，生成的文件扩展名为 .med. 用 C/C++写的包编译后也需要将扩展名改为 .med

2.6.2. 导入包

无论是 menthol 写的包还是 C/C++写的扩展包，包名就是文件名，例如有个包叫做 a.med 则在程序使用他的时候使用

```
Improt "a";
```

就可以导入这个包了，包名并不区分大小写，也就是说 `imprt "a"` 和 `improt "A"` 同样都是导入了 a.med 文件，在语法上，包是可以重复导入的，不过重复导入的包会被忽略掉。

2.6.3. 模块的概念

事实上，一个包中可以包含的只有另外的包，和模块了。模块相当于全局变量和函数的合集，全局变量和函数被封装在一个模块中，可以供其他的模块中的函数或者启动函数调用。

一个包中，可以包含无数个模块，每个模块中又能包含无数个全局变量和函数，而且不能模块中的全局变量和函数相互并不冲突，即使他们叫了相同的名字。

模块使用关键字 `module` 开始，后跟模块名 `{}`，`{}` 中就是全局变量和函数了

```
Module test
{
    Var @g;
    def func:
    {
    }
}
```

一个最简单的模块可以没有全局变量或者函数，只有模块名就够了

```
Module test
{
}
```

虽然在语法上是正确的，但是这样的模块实际上并没有什么意义

2.6.4. 模块的使用

在一个包被导入后，你就可以使用这个包中所包含的所有模块了，不过要使用一个模块，要使用关键字 `use` 模块名，来定义

`use` 关键字后面的模块名可以是一个，也可以是用逗号隔开的多个模块名，这些模块名可以是多个包内的，不要求完全从一个包内引入

```
Import "FileSystem" ;  
use File,Drives;  
use Directory
```

模块名是区分大小写的，在使用时一定要了解，被使用的模块名是大写还是小写，否则将找不到。

模块也可以被多次使用，但是如果存在的话，在此被使用的，将被忽略

```
Import "FileSystem" ;  
use File,Drives, Directory;  
use Directory
```

2.6.5. 函数的基本知识

函数执行操作，也可以返回值。某些时候是计算或比较的结果。

一个函数中包含有几个操作。这样可使得代码更合理化。可以写一组语句并给它命名，然后通过调用它并传递其需要的信息来运行整组语句。

给函数传递信息可以把信息放在函数名称后面的圆括号中。传递给函数的信息称作参数。某些函数根本不帶任何参数，而其他函数带一个或者多个参数。在某些函数中，参数的个数取决于如何使用该函数。

2.6.6. 函数定义和调用

函数采用关键字 `def` 函数名: 参数的形式定义, 参数名不是必须要的。函数名是区分大小写的, 并且函数名只能使用下划线, 字符, 数字构成, 并且函数名的第一个字符不能是数字. 函数可以有返回值, 也可以没有返回值

函数不能脱离模块使用, 也就是说任何函数 (除了 `_mmain`) 都不能在模块外部定义, 任何函数一定是属于一个模块的

```
module moduletest
{
    def funcn:$arg1,$arg2...
    {
        return
    }
}
```

函数必须在调用之前被定义, 否则编译程序将无法使用。

在同一模块内, 调用函数不需要加模块名, 只需要函数名 (参数列表) 就可以, 但是调用外部模块的函数则需要模块名. 函数名 (参数列表). 在本手册中多次使用的 `Console.Out` 就是 `Console` 包中的 `out` 函数

```
import "conosle";
use Console;
_mmmain:$a,$c
{
    funcn(1);
    Console.Out(1);
}
```

2.6.7. 函数参数

通过参数列表可以传递信息到函数，即以逗号作为分隔符的表达式列表。参数是从左向右求值的，并且在 `menthol` 中参数值传递

1. 函数的参数可能有多个，但在调用时候出入的参数比函数定义的参数要少，则默认传入 `NULL`

```
import "console" ;
use Console;
module test
{
    def fucn:$a,$s
    {
        Console.Out($a); //print 1
        Console.Out($s); //print null
    }
}
__mmain:$a,$c
{
    test.fucn(1);
}
```

2. 函数参数如果少于传入的参数，则多余的将被忽略

3. 函数的参数列表使用 C++风格的默认参数，不过默认参数都是在最后，即有个默认参数后，后面要么是没有参数，要么剩下的都是默认参数

```
import "console" ;
use Console;
module test
{
    def fucn:$a,$f,$s = 2,$t = 3
    {
        Console.Out($a); //print 1
        Console.Out($f); //print null
        Console.Out($s); //print 2
        Console.Out($t); //print 3
    }
}
__mmain:$a,$c
{
    test.fucn(1);
}
```

2.6.8. 函数返回值

值通过使用可选的返回语句返回。可以返回包括数组和对象的任意类型。返回语句会立即中止函数的运行，并且将控制权交回调用该函数的代码行

```
import "conosle" ;
use Console;
module test
{
    def fucn:
    {
        return 666;
    }
}
__main:$a,$c
{
    Console.Out(test.fucn(1));
}
```

如果省略了 return，则返回值为 NULL。

第三章 API 列表

3.1. API 列表

所有的 API 都在 menthol.h 中可以看到

1. enum ValueType: 表示所有 menthol 所使用的数据类型, 有些你可能永远不会使用到, 但你可以了解
M_NUMBER, M_LONG, M_DOUBLE, M_STRING, M_SSTRING, M_FUN, M_PFUN, M_BPMARK, M_BOOL, M_ARRAY, M_DICT, M_NULL, M_TRYMARK, M_FORMARK, M_MODULE, M_HASH, M_UNKONWN, M_OBJECT

```
2. struct StackState
{
    union{
        double d;//数值
        int i;
        hashValue hash;//hash 值
        pDict pdict;//字典值
        pArray parray;//数组值
        pString str;//字符串
        StackMark m;
        bool b;//布尔值
        ModuleState* ms;
    };
    pInst p;//表示一个 object 或者一个指针
    char* name;
    hashValue namehash;//变量或者函数名的 hash 值
    ValueType v; //数据类型
};
```

3. typedef int (*PrintErrorFunc)(char* str, char* cf, int line);
错误调用的函数指针, str 参数为错误信息, cf 为发生错误的文件, line 为错误的行数

4. typedef StackState (*funcallback)();
扩展函数函数的函数指针

5. MentholPackMethod void SetPrintCompileErrorFunc(PrintErrorFunc _pef);
指示编译时如果发生错, 将调用哪个函数显示错误信息, _per 为函数指针

6. MentholPackMethod void SetPrintRunTimeErrorFunc(PrintErrorFunc _pef);
指示运行时如果发生错, 将调用哪个函数显示错误信息, _per 为函数指针

7. MentholPackMethod int Compile(char* cfile);
编译一个文件, cfile 为原文件名

8. MentholPackMethod int Run(char* files, char* arg1, char* arg2);
执行可执行文件, files 为可执行文件.mee 的完整文件名, arg1, 为__mmain 的第一个参数, arg2, 为__mmain 的第二个参数

9. MentholPackMethod void RegisterModuleFunciton(RunTimeState* moduleinst, UserFunctionAtter* functionlist);
向系统注册需要扩展的函数名, moduleinst 为模块实例, functionlist 为函数定义数组

10. MentholPackMethod StackState GetParam(int index);
获取函数参数, index 为参数的索引位置, 从 1 开始

11. MentholPackMethod RunTimeState* CreateModuleRunTime(char* modulename);
创建一个运行模块

12. MentholPackMethod StackState Array_CreateArray();
创建一个数组

13. MentholPackMethod StackState Array_Get(pArray sk1, int index);
获取数组元素, sk1 为数组, index 为数组位置索引, 从 0 开始

14. MentholPackMethod void Array_Set(pArray sk1, StackState sk2, int index);
设置数组元素, sk1 为数组, index 为数组位置索引, 从 0 开始, sk2 为要这只的新值

15. MentholPackMethod int Array_Length(pArray p);
获取数组元素个数, p 为数组

16. MentholPackMethod void Array_Push(pArray sk1, StackState sk2);
增加数组元素, sk1 为数组, sk2 为要增加的元素

17. MentholPackMethod pString Array_Join(pArray a1, char* link);
用 link 将数组 a1, 连接在一起返回字符串

18. MentholPackMethod pArray Array_Reverse(pArray a1);
反转数组, 返回新的数组

19. MentholPackMethod StackState Dict_CreateDict();
创建一个字典

20. MentholPackMethod void Dict_Push(char* key, pDict sk1, StackState sk2);
增加字典元素, key 为键, sk1 为字典, sk2 为值

21. MentholPackMethod int Dict_Length(pDict p);
获取字典键值对个数, p 为字典

22. MentholPackMethod StackState Dict_Get(pDict sk1, hashValue key);
获取字典值对个数, sk1 为字典, key 为 key 的 hash 值

23. MentholPackMethod void Dict_Set(char* key, pDict sk1, StackState sk2);
设置字典值对个数, sk1 为字典, key 为键, sk2 为新值

24. MentholPackMethod pString Dict_Key(pDict pdict, hashValue sk2);
获取字典值对个数, sk1 为字典, key 为 key 的 hash 值

25. MentholPackMethod StackState String_CreateString(char* str);
创建字符串

26. MentholPackMethod void CreateFunctionCall(int pc);
创建回调函数的运行环境, pc 为压入函数的参数个数

27. MentholPackMethod void PushNumber(double d);
向栈内压入一个数值

28. MentholPackMethod void PushString(pString str);
向栈内压入一个字符串

29. MentholPackMethod void PushArray(pArray arr);
向栈内压入一个数组

30. MentholPackMethod void PushDict(pDict arr);
向栈内压入一个字典

31. MentholPackMethod StackState CallFunction(StackState fu);
执行函数

32. MentholPackMethod StackState Number_CreateNumber(double d);
创建一个数字

33. MentholPackMethod StackState Null_CreateNull();
创建一个 null

34. MentholPackMethod StackState Bool_CreateBool(bool b);
创建一个 bool

第四章 开发扩展程序

4.1. 如何写一个扩展库

如果要开发自定义的扩展类库，目前 menthol 的扩展开发仅可使用 C/C++ 进行开发，你需要了解的几个开发前提是：

1. 所有被扩展的函数必须要是静态的函数或者是全局的函数
2. 所有被扩展的函数都必须返回 StackState
3. 将所有定义的函数放在一个类型为 UserFunctionAtter 的数组当中，UserFunctionAtter 是一个结构，有 3 个属性

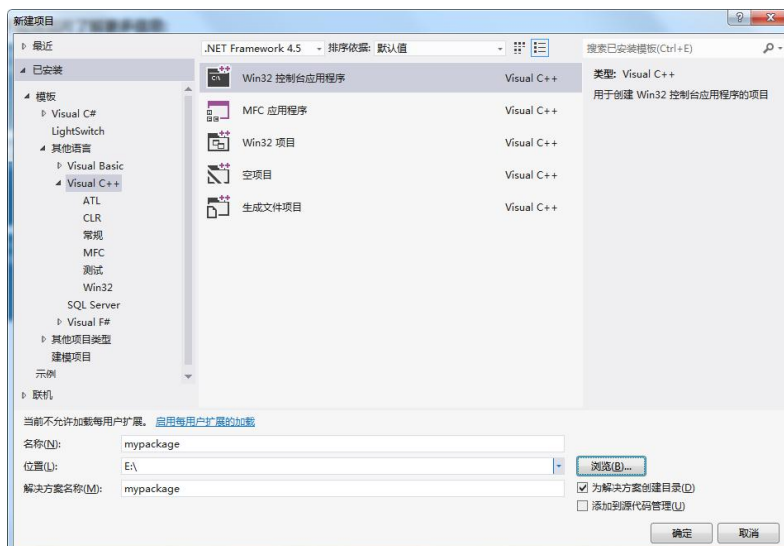
```
struct UserFunctionAtter  
{  
    char* name;  
    funcallback postion;  
    int paramcount;  
};
```

name 表示在 menthol 程序中调用的函数名称
postion 为函数指针，执行你刚才定义的函数
paramcount 表示你定义的函数的参数个数

4. 所有的扩展函数库必须有个函数 MentholModuleMethod void MP_Init(), 这个函数内会调用 CreateModuleRunTimehe 和 RegisterModuleFunciton 将你的函数增加至运行环境内
5. 会写或者会改 C/C++ 源码，并且能生成 DLL 库

menthol 扩展开发非常简单，你需要在你的 C/C++ 页面内引入 menthol.lib 静态库，并且将头文件 menthol.h 加入到你的源代码中。我下面将以 VS 中开发为范例开发一个最简单的扩展库

1. 在 VS 中创建了一个类库





引入 Menthol.h 或者引入 menthol 所在目录也可以，然后把 lib 文件包含在内，然后创建启动函数，启动函数的 MP_Init，大小写要注意

```
#include "stdafx.h"
#include "Menthol.h"
#pragma comment(lib, "Menthol.lib")
```

```
MentholModuleMethod void MP_Init()
{
}
}
```

2. 我们写一个函数，这个函数的功能是，我们输入任何字符串，他都会输出”hello”+ 刚才你输入的字符串

```
#include "stdafx.h"
#include "Menthol.h"
#pragma comment(lib, "Menthol.lib")

StackState function1()
{
    char str[1000];
    sprintf_s(str, "hello,%s", GetParam(1).str->string);
    return String_CreateString(str);
}

UserFunctionAtter funclist[] = {
    { "function1", function1, 1 },
    { NULL, NULL, 0 }
};

MentholModuleMethod void MP_Init()
{
    RunTimeState* Directoryprt = CreateModuleRunTime("mymodule");
    RegisterModuleFunciton(Directoryprt, funclist);
}
}
```

3. 编译，生成 mymoduletest.dll 然后改名为 mymoduletest.med, 然后在代码中调用

```
1 import "console";
2 import "mymoduletest";
3 use Console, mymodule;
4 _mmain: $a, $c
5 {
6     Console.Out (mymodule.function1 (Console.In()));
7 }
8 |
```

好了，现在到程序中测试一下，

```
menthol
hello,menthol
```