**Lua 5.1参考手册**

by Roberto Ierusalimschy, Luiz Henrique de Figueiredo, Waldemar Celes

版权(2006-2008)Lua.org，PUC-Rio。可根据[Lua许可证](http://www.lua.org/license.html#5).

[内容](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\contents.html#contents) · [指数](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\contents.html#index)

**1 -****导言**

Lua是一种扩展编程语言，旨在支持具有数据描述功能的一般过程编程。它还为面向对象编程、函数式编程和数据驱动编程提供了良好的支持.Lua是用来作为一个强大的，重量轻的脚本语言，任何需要一个程序。Lua是作为一个库实现的，它是在*打扫*C(即ANSI C和C++的公共子集)。

作为一种扩展语言，lua没有“主”程序的概念：它只起作用。*嵌入式*在主机客户端中，称为*嵌入程序*或者简单地说*寄主*。这个主机程序可以调用函数来执行一段Lua代码，可以编写和读取Lua变量，还可以注册由Lua代码调用的C函数。通过使用C函数，可以扩展Lua以处理广泛的不同领域，从而创建共享语法框架的定制编程语言。Lua发行版包括一个名为lua，它使用Lua库提供一个完整的、独立的Lua解释器。

Lua是免费软件，并按照其许可证的规定，照常提供，没有任何保证。本手册所述的实施情况可在Lua的官方网站上查阅，www.lua.org.

与任何其他参考手册一样，这份文件在某些地方是干燥的。关于Lua设计背后的决定，请参阅Lua网站上的技术文件。有关Lua编程的详细介绍，请参阅Roberto的书，*Lua编程(第二版)*.

**2 -****语言**

本节描述Lua的词汇、语法和语义。换句话说，本节描述哪些令牌是有效的，它们如何组合，以及它们的组合意味着什么。

语言结构将使用通常的扩展BNF表示法来解释，其中{*a*}表示0或更多*a*还有.*a*]指可选的*a*。非终端显示为非终端，关键字显示为**千字**，而其他终端符号则显示为`**=**“Lua的完整语法可在[§8](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#8)在这本手册的末尾。

**2.1 -****词汇约定**

*人名*(亦称*标识符*)在Lua中，可以是由字母、数字和下划线组成的任何字符串，而不是以数字开头。这与大多数语言中的名称定义是一致的。(字母的定义取决于当前区域设置：当前区域设置所认为的任何字符都可以在标识符中使用。)标识符用于命名变量和表字段。

以下内容*关键词*被保留，不能用作名称：

and break do else elseif

end false for function if

in local nil not or

repeat return then true until while

Lua是一种区分大小写的语言：and是个保留词，但是And和AND是两个不同的，有效的名字。作为惯例，名称以下划线开头，后面跟着大写字母(例如[\_VERSION](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#pdf-_VERSION))为Lua使用的内部全局变量保留。

以下字符串表示其他令牌：

+ - \* / % ^ #

== ~= <= >= < > =

( ) { } [ ]

; : , . .. ...

*文字字符串*可以通过匹配单引号或双引号分隔，并且可以包含以下类似C的转义序列：\a“(贝尔)”\b‘(后台)，’\f“(表格饲料)，”\n“(新线)”\r“(回程)，”\t“(水平选项卡)，”\v“(垂直选项卡)，”\\‘(反斜杠)，’\"‘(引号[双引号])，和\'‘(撇号[单引号])。此外，反斜杠后面跟着真正的换行符将导致字符串中的换行符。字符串中的字符也可以使用转义序列由其数值指定。\*ddd*，在哪里*DDD*是最多三个十进制数字的序列。(请注意，如果数字转义后面跟着一个数字，则必须使用精确的三位数字表示。)Lua中的字符串可以包含任何8位值，包括嵌入的零，可以指定为‘\0'.

还可以使用长格式定义文字字符串。*长括号*。我们定义*开长水平托架n*作为开头的方括号，后面跟着*n*同样的标志，后面跟着另一个开口的方括号。因此，0级的开头长括号编写为[[，第1级的开头长括号写为[=[等等。一个*闭合长托架*定义为类似的；例如，第4级的结束长括号被写为]====]。长字符串以任何级别的开始长括号开始，在同一级别的第一个结束长括号结束。这种括号中的文字可以运行几行，不解释任何转义序列，也可以忽略任何其他级别的长括号。它们可以包含任何东西，除了一个适当级别的结束括号。

为了方便起见，当开始长括号后面紧跟换行符时，换行符不包含在字符串中。例如，在使用ASCII的系统中(其中‘a“编码为97，换行符编码为10，”1‘编码为49)，下面的五个文字字符串表示相同的字符串：

a = 'alo\n123"'

a = "alo\n123\""

a = '\97lo\10\04923"'

a = [[alo

123"]]

a = [==[

alo

123"]==]

A *数值常数*可以用可选的小数部分和可选的十进制指数编写。Lua还接受整数十六进制常量，其前缀为0x。有效数值常数的示例如下

3 3.0 3.1416 314.16e-2 0.31416E1 0xff 0x56

A *评语*以双连字符开头(--)字符串以外的任何地方。如果后面的文字--不是开头的长括号，注释是*简短评论*它一直运行到线的末尾。否则，它就是*长话*，它将一直运行到相应的关闭长括号。长注释经常用于暂时禁用代码。

**2.2 -****价值和类型**

卢阿*动态类型语言*。这意味着变量没有类型；只有值有。语言中没有类型定义。所有的价值观都有自己的类型。

Lua中的所有值都是*一级值*。这意味着所有值都可以存储在变量中，作为参数传递给其他函数，并作为结果返回。

Lua有八种基本类型：*零*, *布尔型*, *数*, *弦*, *功能*, *用户数据*, *螺纹*，和*表*. *零*值的类型。**零**，其主要属性将不同于任何其他值；它通常表示没有有用的值。*布尔型*值的类型。**假的**和**千真万确**。双管齐下**零**和**假的**使条件为假；任何其他值使其为真。*数*表示实数(双精度浮点数)。(很容易构建使用其他内部表示的数字的lua解释器，例如单精度浮点数或长整数；请参阅文件。luaconf.h.) *弦*表示字符数组。Lua是8位干净的：字符串可以包含任何8位字符，包括嵌入的零(‘\0‘)(见[§2.1](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#2.1)).

Lua可以调用(和操作)用Lua编写的函数和用C编写的函数(参见[§2.5.8](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#2.5.8)).

类型*用户数据*以允许将任意C数据存储在Lua变量中。这种类型对应于原始内存块，除了赋值和标识测试之外，Lua中没有预定义的操作。但是，通过使用*Metatable*，程序员可以为userdata值定义操作(请参阅[§2.8](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#2.8))。用户数据值不能在Lua中创建或修改，只能通过CAPI。这保证了主机程序拥有的数据的完整性。

类型*螺纹*表示独立的执行线程，并用于实现协同(请参阅[§2.11](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#2.11))。不要混淆Lua线程和操作系统线程.Lua支持所有系统上的协同，甚至那些不支持线程的系统。

类型*表*实现关联数组，即不仅可以用数字索引的数组，还可以用任何值索引的数组(除**零**)。表可以是*异类*；也就是说，它们可以包含所有类型的值(除**零**)。表是Lua中唯一的数据结构机制，它们可以用来表示普通数组、符号表、集合、记录、图表、树等。为了表示记录，Lua使用字段名作为索引。该语言通过提供a.name作为句法糖a["name"]。有几种在Lua中创建表的方便方法(请参阅[§2.5.7](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#2.5.7)).

与索引一样，表字段的值可以是任意类型(除**零**)。特别是，由于函数是一流的值，所以表字段可以包含函数.因此，表也可以携带*方法*(见[§2.5.9](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#2.5.9)).

表、函数、线程和(完整)用户数据值是*对象*\*变量实际上并不*含*这些值*参考文献*敬他们。赋值、参数传递和函数返回总是操纵对这些值的引用；这些操作并不意味着任何类型的复制。

图书馆功能[type](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#pdf-type)返回一个字符串，该字符串描述给定值的类型。

**2.2.1 -****胁迫**

Lua在运行时提供字符串和数字值之间的自动转换。应用于字符串的任何算术操作都试图按照通常的转换规则将此字符串转换为数字。相反，只要在需要字符串的地方使用数字，就会以合理的格式将数字转换为字符串。若要完全控制数字如何转换为字符串，请使用format函数来自字符串库(请参见[string.format](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#pdf-string.format)).

**2.3 -****变量**

变量是存储值的位置。Lua中有三种变量：全局变量、局部变量和表字段。

单个名称可以表示全局变量或局部变量(或函数的形式参数，它是特定类型的局部变量)：

var ::= Name

名称表示标识符，如[§2.1](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#2.1).

除非显式声明为本地变量，否则任何变量都假定为全局变量(请参见[§2.4.7](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#2.4.7))。局部变量是*词汇范围*\*局部变量可以通过在其范围内定义的函数自由访问(请参见[§2.6](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#2.6)).

在第一个赋值给变量之前，它的值是**零**.

方括号用于索引表：

var ::= prefixexp `**[**´ exp `**]**´

访问全局变量和表字段的意义可以通过元表更改。对索引变量的访问t[i]等于呼叫。gettable\_event(t,i)。(见[§2.8](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#2.8)的完整描述。gettable\_event功能。在Lua中不能定义或调用此函数。我们在这里使用它只是为了解释。)

语法var.Name只是句法上的糖var["Name"]:

var ::= prefixexp `**.**´ Name

所有全局变量都以普通Lua表中的字段形式存在，称为*环境表*或者简单的*环境*(见[§2.9](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#2.9))。每个函数都有对环境的引用，因此该函数中的所有全局变量都将引用此环境表。创建函数时，它从创建函数的函数继承环境。要获取lua函数的环境表，您可以调用[getfenv](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#pdf-getfenv)。要替换它，你打电话给[setfenv](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#pdf-setfenv)。(只能通过调试库操作C函数的环境；(请参见[§5.9](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#5.9)).)

对全局变量的访问x等于\_env.x，这又相当于

gettable\_event(\_env, "x")

哪里\_env是运行函数的环境。(见[§2.8](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#2.8)的完整描述。gettable\_event功能。在Lua中不能定义或调用此函数。类似地，\_env变量未在Lua中定义。我们在这里使用它们只是为了解释。)

**2.4 -****陈述**

Lua支持几乎传统的语句集，类似于Pascal或C中的语句集。这组语句包括赋值、控制结构、函数调用和变量声明。

**2.4.1 -****块**

Lua的执行单位称为*块*。块只是一系列语句，这些语句是按顺序执行的。每个语句的后面都可以有一个分号：

chunk ::= {stat [`**;**´]}

没有空语句，因此‘;;“这是不合法的。

Lua将块处理为具有可变参数数的匿名函数的主体(请参见[§2.5.9](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#2.5.9))。因此，块可以定义局部变量、接收参数和返回值。

块可以存储在文件中，也可以存储在主机程序中的字符串中。要执行块，Lua首先将块预编译为虚拟机的指令，然后使用虚拟机的解释器执行编译后的代码。

块也可以预编译成二进制形式；请参阅程序。luac关于细节。源代码和编译后的表单中的程序是可互换的；Lua会自动检测文件类型并采取相应的行动。

**2.4.2 -****砌块**

块是语句的列表；从语法上讲，块与块相同：

block ::= chunk

可以显式地分隔块以生成一条语句：

stat ::= **do** block **end**

显式块对于控制变量声明的范围非常有用。显式块有时也用于添加**回归**或**破断**语句位于另一个块的中间(请参见[§2.4.4](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#2.4.4)).

**2.4.3 -****赋值**

Lua允许多个任务。因此，赋值语法定义左侧的变量列表和右侧的表达式列表。两个列表中的元素都用逗号分隔：

stat ::= varlist `**=**´ explist

varlist ::= var {`**,**´ var}

explist ::= exp {`**,**´ exp}

表达式将在[§2.5](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#2.5).

在分配之前，值的列表是*调整后*变量列表的长度。如果有超过需要的值，多余的值就会被丢弃。如果值比需要的少，则该列表将以相同的数量进行扩展。**零**如果表达式列表以函数调用结尾，则该调用返回的所有值在调整之前输入值列表(除非调用被括在括号中；请参见[§2.5](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#2.5)).

赋值语句首先计算其所有表达式，然后才执行赋值。因此，代码

i = 3

i, a[i] = i+1, 20

集a[3]至20，但不影响a[4]因为i在……里面a[i]在赋值4之前计算(到3)。

x, y = y, x

交换x和y，和

x, y, z = y, z, x

周期性地改变了x, y，和z.

分配给全局变量和表字段的意义可以通过元表来更改。索引变量的赋值t[i] = val等于settable\_event(t,i,val)。(见[§2.8](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#2.8)的完整描述。settable\_event功能。在Lua中不能定义或调用此函数。我们在这里使用它只是为了解释。)

全局变量的赋值x = val等于分配。\_env.x = val，这又相当于

settable\_event(\_env, "x", val)

哪里\_env是运行函数的环境。()\_env变量未在Lua中定义。我们在这里使用它只是为了解释。)

**2.4.4 -****控制结构**

控制结构**如果**, **当**，和**重复**具有通常的含义和熟悉的语法：

stat ::= **while** exp **do** block **end**

stat ::= **repeat** block **until** exp

stat ::= **if** exp **then** block {**elseif** exp **then** block} [**else** block] **end**

Lua也有**为**语句，有两种样式(请参见[§2.4.5](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#2.4.5)).

控制结构的条件表达式可以返回任何值。双管齐下**假的**和**零**都被认为是假的。所有不同的价值观**零**和**假的**被认为是真(特别是数字0和空字符串也是true)。

在**重复**–**直到**循环时，内部块不会以**直到**关键字，但只在条件之后。因此，条件可以引用在循环块中声明的局部变量。

这个**回归**语句用于从函数或块(这只是一个函数)返回值。函数和块可以返回多个值，因此**回归**语句是

stat ::= **return** [explist]

这个**破断**语句用于终止**当**, **重复**，或**为**循环，跳到循环之后的下一个语句：

stat ::= **break**

A **破断**结束最内部的封闭循环。

这个**回归**和**破断**语句只能作为*最后的*一个街区的陈述。如果真的有必要**回归**或**破断**在块的中间，就可以使用显式的内部块，就像成语中的那样。do return end和do break end，因为现在**回归**和**破断**是它们(内部)块中的最后语句。

**2.4.5 -****作陈述**

这个**为**语句有两种形式：一个是数字形式，另一个是通用形式。

数字**为**循环重复一段代码，而控制变量运行在算术过程中。它有以下语法：

stat ::= **for** Name `**=**´ exp `**,**´ exp [`**,**´ exp] **do** block **end**

这个*块*被重复*名字*从第一个值开始*exp*，直到它经过第二个*exp*按第三步*exp*。更准确地说，**为**像这样的语句

for v = *e1*, *e2*, *e3* do *block* end

等于代码：

do

local *var*, *limit*, *step* = tonumber(*e1*), tonumber(*e2*), tonumber(*e3*)

if not (*var* and *limit* and *step*) then error() end

while (*step* > 0 and *var* <= *limit*) or (*step* <= 0 and *var* >= *limit*) do

local v = *var*

*block*

*var* = *var* + *step*

end

end

注意：

* 在循环开始之前，只对所有三个控件表达式进行一次计算。它们都必须是数字的结果。
* *var*, *limit*，和*step*是看不见的变量。此处所示的名称仅供解释之用。
* 如果没有第三个表达式(步骤)，则使用步骤1。
* 你可以用**破断**退出a**为**循环。
* 循环变量v是循环的本地值；不能在**为**结束或破碎。如果需要此值，请在中断或退出循环之前将其赋值给另一个变量。

泛型**为**语句工作于函数，称为*迭代器*。在每次迭代中，迭代器函数被调用以生成一个新值，当这个新值为**零**。泛型**为**循环具有以下语法：

stat ::= **for** namelist **in** explist **do** block **end**

namelist ::= Name {`**,**´ Name}

A **为**像这样的语句

for *var\_1*, ···, *var\_n* in *explist* do *block* end

等于代码：

do

local *f*, *s*, *var* = *explist*

while true do

local *var\_1*, ···, *var\_n* = *f*(*s*, *var*)

*var* = *var\_1*

if *var* == nil then break end

*block*

end

end

注意：

* *explist*只计算一次。其结果是*迭代器*功能，a*国家*，以及第一个值的初始值。*迭代变量*.
* *f*, *s*，和*var*是看不见的变量。这里的名字只是为了解释。
* 你可以用**破断**退出a**为**循环。
* 循环变量*var\_i*是循环的本地值；您不能在**为**结束了。如果需要这些值，那么在中断或退出循环之前将它们分配给其他变量。

**2.4.6 -****函数调用为语句**

为了允许可能的副作用，函数调用可以作为语句执行：

stat ::= functioncall

在这种情况下，所有返回的值都会被抛出。函数调用在[§2.5.8](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#2.5.8).

**2.4.7 -****局部声明**

可以在块内的任何地方声明局部变量。声明可以包括初始转让：

stat ::= **local** namelist [`**=**´ explist]

如果存在，则初始赋值具有多个赋值的相同语义(请参见[§2.4.3](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#2.4.3))。否则，所有变量都将使用**零**.

块也是块(参见[§2.4.1](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#2.4.1))，因此可以在任何显式块之外的块中声明局部变量。这些局部变量的作用域一直延伸到块的末尾。

局部变量的可见性规则在[§2.6](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#2.6).

**2.5 -****表达**

Lua中的基本表达式如下：

exp ::= prefixexp

exp ::= **nil** | **false** | **true**

exp ::= Number

exp ::= String

exp ::= function

exp ::= tableconstructor

exp ::= `**...**´

exp ::= exp binop exp

exp ::= unop exp

prefixexp ::= var | functioncall | `**(**´ exp `**)**´

数字和文字字符串在[§2.1](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#2.1)变量在[§2.3](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#2.3)；函数定义在[§2.5.9](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#2.5.9)；函数调用在[§2.5.8](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#2.5.8)表构造函数在[§2.5.7](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#2.5.7)。Vararg表达式，用三个点表示(‘...)，只能在vararg函数内部直接使用；它们将在[§2.5.9](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#2.5.9).

二进制运算符包括算术运算符(请参阅[§2.5.1](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#2.5.1))，关系运算符(请参见[§2.5.2](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#2.5.2))，逻辑运算符(请参见[§2.5.3](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#2.5.3))，以及级联操作符(请参见[§2.5.4](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#2.5.4))。一元运算符构成一元减去(见[§2.5.1](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#2.5.1))，一元**不**(见[§2.5.3](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#2.5.3))，以及一元*长度算子*(见[§2.5.5](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#2.5.5)).

函数调用和vararg表达式都可能导致多个值。如果表达式用作语句(仅可用于函数调用)(请参见[§2.4.6](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#2.4.6))，则将其返回列表调整为零元素，从而丢弃所有返回的值。如果表达式用作表达式列表的最后(或唯一)元素，则不进行任何调整(除非调用被括在括号内)。在所有其他上下文中，Lua将结果列表调整为一个元素，丢弃除第一个以外的所有值。

以下是一些例子：

f() -- adjusted to 0 results

g(f(), x) -- f() is adjusted to 1 result

g(x, f()) -- g gets x plus all results from f()

a,b,c = f(), x -- f() is adjusted to 1 result (c gets nil)

a,b = ... -- a gets the first vararg parameter, b gets

-- the second (both a and b can get nil if there

-- is no corresponding vararg parameter)

a,b,c = x, f() -- f() is adjusted to 2 results

a,b,c = f() -- f() is adjusted to 3 results

return f() -- returns all results from f()

return ... -- returns all received vararg parameters

return x,y,f() -- returns x, y, and all results from f()

{f()} -- creates a list with all results from f()

{...} -- creates a list with all vararg parameters

{f(), nil} -- f() is adjusted to 1 result

括在括号中的任何表达式总是只产生一个值。因此，(f(x,y,z))总是一个值，即使f返回多个值。(价值)(f(x,y,z))返回的第一个值。f或**零**如果f不返回任何值。)

**2.5.1 -****算术算子**

Lua支持常用的算术运算符：二进制+(加法)，-(减法)\*(乘法)，/(分组)，%(模块)^(指数)；和一元-(否定)如果操作数是数字，或者可以转换为数字的字符串(请参见[§2.2.1](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#2.2.1))，则所有操作都具有通常的含义。指数对任何指数都有效。例如,x^(-0.5)的平方根的逆运算。x。模定义为

a % b == a - math.floor(a/b)\*b

也就是说，它是除法的剩余部分，它将商数圈到负无穷远。

**2.5.2 -****关系算子**

Lua中的关系运算符是

== ~= < > <= >=

这些运算符总是导致**假的**或**千真万确**.

平等(==)首先比较其操作数的类型。如果类型不同，则结果是**假的**。否则，比较操作数的值。数字和字符串按通常的方式进行比较。对象(表、用户数据、线程和函数)*参照系*\*只有当两个对象是*同*对象。每次创建新对象(表、用户数据、线程或函数)时，这个新对象都不同于以前存在的任何对象。

通过使用“eq”元方法，可以更改lua比较表和用户数据的方式(请参阅[§2.8](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#2.8)).

的转换规则[§2.2.1](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#2.2.1) *不要*适用于平等比较。因此，"0"==0评估为**假的**，和t[0]和t["0"]表示表中的不同项。

操作员~=就是对平等的否定(==).

Order操作符的工作方式如下。如果这两个参数都是数字，则将它们作为数字进行比较。否则，如果两个参数都是字符串，则根据当前区域设置比较它们的值。否则，lua尝试调用“lt”或“le”元方法(请参见[§2.8](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#2.8))。比较a > b翻译成b < a和a >= b翻译成b <= a.

**2.5.3 -****逻辑运算符**

Lua中的逻辑运算符是**和**, **或**，和**不**。类似于控制结构(参见[§2.4.4](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#2.4.4))，所有逻辑运算符都会同时考虑**假的**和**零**都是假的，其他的都是真的。

否定算子**不**总是回来**假的**或**千真万确**。连接运算符**和**如果此值为**假的**或**零**否则，**和**返回它的第二个参数。分离算子**或**如果此值与**零**和**假的**否则，**或**返回它的第二个参数。双管齐下**和**和**或**使用捷径计算；也就是说，只有在必要时才计算第二个操作数.以下是一些例子：

10 or 20 --> 10

10 or error() --> 10

nil or "a" --> "a"

nil and 10 --> nil

false and error() --> false

false and nil --> false

false or nil --> nil

10 and 20 --> 20

(在本手册中，-->指示前面表达式的结果。)

**2.5.4 -****级联**

Lua中的字符串级联操作符由两个点表示(‘..‘)。如果两个操作数都是字符串或数字，则按照[§2.2.1](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#2.2.1)。否则，调用“conat”元方法(请参见[§2.8](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#2.8)).

**2.5.5 -****长度算子**

长度运算符由一元运算符表示。#。字符串的长度是它的字节数(也就是说，当每个字符是一个字节时，字符串长度的通常含义)。

桌子的长度t定义为任何整数索引。n使.t[n]不是**零**和t[n+1]是**零**此外，如果t[1]是**零**, n可能是零。对于常规数组，其非零值从1到给定。n，它的长度正好是n，它的最后一个值的索引。如果数组有“孔”(也就是说，**零**其他非零值之间的值)，则#t可以是直接位于**零**价值(即，它可以考虑任何这样的情况)。**零**值作为数组的结尾)。

**2.5.6 -****优先**

在Lua中，运算符优先于下表，从低优先级到高优先级：

or

and

< > <= >= ~= ==

..

+ -

\* / %

not # - (unary)

^

与往常一样，您可以使用括号来更改表达式的先例。连接(‘)..)和指数(‘)^‘)运算符是正确的结合。所有其他二进制运算符都是左关联的。

**2.5.7 -****表构造函数**

表构造函数是创建表的表达式。每次计算构造函数时，都会创建一个新表。构造函数可用于创建空表或创建表并初始化其某些字段。构造函数的一般语法是

tableconstructor ::= `**{**´ [fieldlist] `**}**´

fieldlist ::= field {fieldsep field} [fieldsep]

field ::= `**[**´ exp `**]**´ `**=**´ exp | Name `**=**´ exp | exp

fieldsep ::= `**,**´ | `**;**´

表单的每个字段[exp1] = exp2在新表中添加带键的条目。exp1和价值exp2。表格中的字段name = exp等于["name"] = exp。最后，表单的字段exp等于[i] = exp，在哪里i是连续的数值整数，从1开始。其他格式的字段不影响此计数。例如,

a = { [f(1)] = g; "x", "y"; x = 1, f(x), [30] = 23; 45 }

等于

do

local t = {}

t[f(1)] = g

t[1] = "x" -- 1st exp

t[2] = "y" -- 2nd exp

t.x = 1 -- t["x"] = 1

t[3] = f(x) -- 3rd exp

t[30] = 23

t[4] = 45 -- 4th exp

a = t

end

如果列表中的最后一个字段具有窗体exp表达式是一个函数调用或vararg表达式，然后该表达式返回的所有值都将连续输入列表(请参见[§2.5.8](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#2.5.8))。为了避免这种情况，请将函数调用或vararg表达式括在括号中(请参见[§2.5](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#2.5)).

字段列表可以有一个可选的尾随分隔符，这为机器生成的代码提供了方便。

**2.5.8 -****函数调用**

Lua中的函数调用具有以下语法：

functioncall ::= prefixexp args

在函数调用中，首先计算前缀和args。如果前缀exp的值有类型*功能*，则使用给定的参数调用此函数。否则，调用前缀“Call”元方法，将前缀的值作为第一个参数，然后是原始的调用参数(请参见[§2.8](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#2.8)).

形式

functioncall ::= prefixexp `**:**´ Name args

可用于调用“方法”。打电话v:name(*args*)是句法糖v.name(v,*args*)，除了v只计算一次。

参数具有以下语法：

args ::= `**(**´ [explist] `**)**´

args ::= tableconstructor

args ::= String

所有参数表达式都在调用之前进行计算。形式的召唤f{*fields*}是句法糖f({*fields*})也就是说，参数列表是一个单一的新表。形式的召唤f'*string*'(或f"*string*"或f[[*string*]])是句法糖f('*string*')也就是说，参数列表是一个单一的文字字符串。

作为lua免费格式语法的一个例外，您不能在“(“在一个函数调用中。这种限制避免了语言中的一些歧义。如果你写

a = f

(g).x(a)

卢阿会把这看作是一个单一的陈述，a = f(g).x(a)。因此，如果您想要两个语句，您必须在它们之间添加一个分号。如果你真的想打电话f，您必须删除之前的中断行。(g).

形式的召唤return *功能调用*称为a*尾叫*。Lua工具*适当尾呼叫*(或*适当尾递归*)：在尾调用中，被调用函数重用调用函数的堆栈条目。因此，程序可以执行的嵌套尾调用的数量没有限制。但是，尾调用会擦除有关调用函数的任何调试信息。注意，尾调用只发生在特定的语法中，其中**回归**只有一个函数调用作为参数；此语法使调用函数完全返回被调用函数的返回。因此，以下示例中没有一个是尾调用：

return (f(x)) -- results adjusted to 1

return 2 \* f(x)

return x, f(x) -- additional results

f(x); return -- results discarded

return x or f(x) -- results adjusted to 1

**2.5.9 -****函数定义**

函数定义的语法是

function ::= **function** funcbody

funcbody ::= `**(**´ [parlist] `**)**´ block **end**

以下语法简化了函数定义：

stat ::= **function** funcname funcbody

stat ::= **local** **function** Name funcbody

funcname ::= Name {`**.**´ Name} [`**:**´ Name]

声明

function f () *body* end

翻译成

f = function () *body* end

声明

function t.a.b.c.f () *body* end

翻译成

t.a.b.c.f = function () *body* end

声明

local function f () *body* end

翻译成

local f; f = function () *body* end

*不*到

local f = function () *body* end

(只有当函数的主体包含对f.)

函数定义是可执行表达式，其值具有类型。*功能*。当Lua预编译一个块时，它的所有功能体也都是预编译的.然后，每当lua执行函数定义时，函数是*实例化*(或*关着的不营业的*)。此函数实例(或*封闭*)是表达式的最终值。同一函数的不同实例可以引用不同的外部局部变量，并且可以具有不同的环境表。

参数充当用参数值初始化的局部变量：

parlist ::= namelist [`**,**´ `**...**´] | `**...**´

调用函数时，参数列表将调整为参数列表的长度，除非该函数是变量或*vararg函数*，由三个点(‘)表示。...‘)在其参数列表的末尾。vararg函数不调整其参数列表；相反，它收集所有额外的参数并通过*vararg表达式*，它也被写成三个点。此表达式的值是所有实际额外参数的列表，类似于具有多个结果的函数。如果vararg表达式在另一个表达式中或在表达式列表的中间使用，则将其返回列表调整为一个元素。如果表达式用作表达式列表的最后一个元素，则不进行任何调整(除非最后一个表达式被括在括号中)。

例如，考虑以下定义：

function f(a, b) end

function g(a, b, ...) end

function r() return 1,2,3 end

然后，我们有从参数到参数和vararg表达式的以下映射：

CALL PARAMETERS

f(3) a=3, b=nil

f(3, 4) a=3, b=4

f(3, 4, 5) a=3, b=4

f(r(), 10) a=1, b=10

f(r()) a=1, b=2

g(3) a=3, b=nil, ... --> (nothing)

g(3, 4) a=3, b=4, ... --> (nothing)

g(3, 4, 5, 8) a=3, b=4, ... --> 5 8

g(5, r()) a=5, b=1, ... --> 2 3

结果将使用**回归**声明(见[§2.4.4](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#2.4.4))。如果控件到达函数的末尾而没有遇到**回归**语句，则函数返回时不带结果。

这个*结肠*语法用于定义*方法*，即具有隐式额外参数的函数。self。因此，声明

function t.a.b.c:f (*params*) *body* end

是句法糖

t.a.b.c.f = function (self, *params*) *body* end

**2.6 -****可见性规则**

Lua是一种在词汇范围内的语言。变量的范围从第一个语句开始。*后*它们的声明并持续到包含声明的最内部块的末尾。考虑以下示例：

x = 10 -- global variable

do -- new block

local x = x -- new 'x', with value 10

print(x) --> 10

x = x+1

do -- another block

local x = x+1 -- another 'x'

print(x) --> 12

end

print(x) --> 11

end

print(x) --> 10 (the global one)

注意，在这样的声明中local x = x，新的x被声明的范围还没有确定，因此第二个x引用外部变量。

由于词法作用域规则，局部变量可以通过在其作用域内定义的函数自由访问。内部函数使用的局部变量称为*上值*，或*外部局部变量*，内部功能。

注意，每次执行**本土化**语句定义新的局部变量。考虑以下示例：

a = {}

local x = 20

for i=1,10 do

local y = 0

a[i] = function () y=y+1; return x+y end

end

循环创建十个闭包(即匿名函数的十个实例)。这些闭包中的每一个都使用不同的y变量，而它们都共享相同的x.

**2.7 -****错误处理**

因为Lua是一种嵌入式扩展语言，所以所有Lua操作都从主机程序中的C代码开始，从Lua库调用函数(请参阅[lua\_pcall](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#lua_pcall))。每当在Lua编译或执行过程中发生错误时，控件都会返回给C，C可以采取适当的措施(例如打印错误消息)。

Lua代码可以通过调用[error](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#pdf-error)功能。如果需要捕获lua中的错误，可以使用[pcall](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#pdf-pcall)功能。

**2.8 -****Metatable**

Lua中的每个值都可以有一个*元可*。这*元可*是一个普通的Lua表，它定义了在某些特殊操作下原始值的行为。您可以通过在值元中设置特定字段来更改值上操作行为的几个方面。例如，当非数字值是加法的操作数时，lua将检查字段中的函数。"\_\_add"在它的代名词中。如果它找到了一个，Lua调用这个函数来执行加法。

我们称钥匙为可转换的*事件*和价值*元方法*。在前面的示例中，事件是"add"元方法是执行加法的函数。

可以通过[getmetatable](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#pdf-getmetatable)功能。

可以通过[setmetatable](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#pdf-setmetatable)功能。您不能从Lua中更改其他类型的元类型(除非使用调试库)；为此您必须使用CAPI。

表和完整的用户数据有单独的元数据(尽管多个表和用户数据可以共享它们的元数据)。所有其他类型的值在每种类型中共享一个元；也就是说，对于所有数字都有一个元值，对于所有字符串都有一个元值，等等。

元可控制对象在算术操作、顺序比较、连接、长度操作和索引中的行为。元数据还可以定义垃圾收集用户数据时调用的函数。对于这些操作中的每一个，Lua关联一个名为*事件*。当Lua对一个值执行这些操作之一时，它将检查该值是否与相应的事件具有元可选性。如果是这样，则与键(元方法)关联的值控制Lua将如何执行操作。

Metatables控制接下来列出的操作。每个操作都由其相应的名称标识。每个操作的键是一个字符串，其名称以两个下划线作为前缀，‘\_\_‘；例如，操作“Add”的键是字符串"\_\_add"。描述解释器如何执行操作的Lua函数更好地解释了这些操作的语义。

这里显示在Lua中的代码只是说明性的；实际的行为在解释器中是硬编码的，它比这个模拟要高效得多。这些描述中使用的所有函数([rawget](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#pdf-rawget), [tonumber](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#pdf-tonumber)等等)在[§5.1](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#5.1)。特别是，为了检索给定对象的元方法，我们使用以下表达式

metatable(obj)[event]

这应该被理解为

rawget(getmetatable(obj) or {}, event)

也就是说，对元方法的访问不会调用其他元方法，并且对没有元对象的对象的访问不会失败(它只会导致**零**).

* **“添加”：**这个+行动。

功能getbinhandler下面定义了Lua如何为二进制操作选择处理程序。首先，Lua尝试第一个操作数。如果它的类型没有为操作定义一个处理程序，那么Lua将尝试第二个操作数。

function getbinhandler (op1, op2, event)

return metatable(op1)[event] or metatable(op2)[event]

end

通过使用此函数，op1 + op2是

function add\_event (op1, op2)

local o1, o2 = tonumber(op1), tonumber(op2)

if o1 and o2 then -- both operands are numeric?

return o1 + o2 -- '+' here is the primitive 'add'

else -- at least one of the operands is not numeric

local h = getbinhandler(op1, op2, "\_\_add")

if h then

-- call the handler with both operands

return (h(op1, op2))

else -- no handler available: default behavior

error(···)

end

end

end

* **“分”：**这个-行动。行为类似于“添加”操作。
* **“mul”：**这个\*行动。行为类似于“添加”操作。
* **“DIV”：**这个/行动。行为类似于“添加”操作。
* **“国防部”：**这个%行动。类似于“Add”操作的行为，使用该操作o1 - floor(o1/o2)\*o2作为原始操作。
* **“POW”：**这个^(指数)运算。与函数类似的“添加”操作的行为pow(从C数学库)作为基本操作。
* **“UNM”：**一元-行动。
* function unm\_event (op)
* local o = tonumber(op)
* if o then -- operand is numeric?
* return -o -- '-' here is the primitive 'unm'
* else -- the operand is not numeric.
* -- Try to get a handler from the operand
* local h = metatable(op).\_\_unm
* if h then
* -- call the handler with the operand
* return (h(op))
* else -- no handler available: default behavior
* error(···)
* end
* end
* end
* **“优惠”：**这个..(串联)操作。
* function concat\_event (op1, op2)
* if (type(op1) == "string" or type(op1) == "number") and
* (type(op2) == "string" or type(op2) == "number") then
* return op1 .. op2 -- primitive string concatenation
* else
* local h = getbinhandler(op1, op2, "\_\_concat")
* if h then
* return (h(op1, op2))
* else
* error(···)
* end
* end
* end
* **“连”：**这个#行动。
* function len\_event (op)
* if type(op) == "string" then
* return strlen(op) -- primitive string length
* elseif type(op) == "table" then
* return #op -- primitive table length
* else
* local h = metatable(op).\_\_len
* if h then
* -- call the handler with the operand
* return (h(op))
* else -- no handler available: default behavior
* error(···)
* end
* end
* end

看见[§2.5.5](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#2.5.5)有关表长度的说明。

* **“eq”：**这个==行动。功能getcomphandler定义Lua如何为比较运算符选择元方法。只有当被比较的两个对象对所选操作具有相同的类型和相同的元方法时，才会选择元方法。
* function getcomphandler (op1, op2, event)
* if type(op1) ~= type(op2) then return nil end
* local mm1 = metatable(op1)[event]
* local mm2 = metatable(op2)[event]
* if mm1 == mm2 then return mm1 else return nil end
* end

“eq”事件的定义如下：

function eq\_event (op1, op2)

if type(op1) ~= type(op2) then -- different types?

return false -- different objects

end

if op1 == op2 then -- primitive equal?

return true -- objects are equal

end

-- try metamethod

local h = getcomphandler(op1, op2, "\_\_eq")

if h then

return (h(op1, op2))

else

return false

end

end

a ~= b等于not (a == b).

* **“它”：**这个<行动。
* function lt\_event (op1, op2)
* if type(op1) == "number" and type(op2) == "number" then
* return op1 < op2 -- numeric comparison
* elseif type(op1) == "string" and type(op2) == "string" then
* return op1 < op2 -- lexicographic comparison
* else
* local h = getcomphandler(op1, op2, "\_\_lt")
* if h then
* return (h(op1, op2))
* else
* error(···)
* end
* end
* end

a > b等于b < a.

* **“le”：**这个<=行动。
* function le\_event (op1, op2)
* if type(op1) == "number" and type(op2) == "number" then
* return op1 <= op2 -- numeric comparison
* elseif type(op1) == "string" and type(op2) == "string" then
* return op1 <= op2 -- lexicographic comparison
* else
* local h = getcomphandler(op1, op2, "\_\_le")
* if h then
* return (h(op1, op2))
* else
* h = getcomphandler(op1, op2, "\_\_lt")
* if h then
* return not h(op2, op1)
* else
* error(···)
* end
* end
* end
* end

a >= b等于b <= a。注意，在没有“le”元方法的情况下，Lua尝试“lt”，假设a <= b等于not (b < a).

* **“索引”：**索引访问table[key].
* function gettable\_event (table, key)
* local h
* if type(table) == "table" then
* local v = rawget(table, key)
* if v ~= nil then return v end
* h = metatable(table).\_\_index
* if h == nil then return nil end
* else
* h = metatable(table).\_\_index
* if h == nil then
* error(···)
* end
* end
* if type(h) == "function" then
* return (h(table, key)) -- call the handler
* else return h[key] -- or repeat operation on it
* end
* end
* **“newindex”：**索引分配table[key] = value.
* function settable\_event (table, key, value)
* local h
* if type(table) == "table" then
* local v = rawget(table, key)
* if v ~= nil then rawset(table, key, value); return end
* h = metatable(table).\_\_newindex
* if h == nil then rawset(table, key, value); return end
* else
* h = metatable(table).\_\_newindex
* if h == nil then
* error(···)
* end
* end
* if type(h) == "function" then
* h(table, key,value) -- call the handler
* else h[key] = value -- or repeat operation on it
* end
* end
* **“呼叫”：**当Lua调用一个值时调用。
* function function\_event (func, ...)
* if type(func) == "function" then
* return func(...) -- primitive call
* else
* local h = metatable(func).\_\_call
* if h then
* return h(func, ...)
* else
* error(···)
* end
* end
* end

**2.9 -****环境**

除了元类之外，类型为线程、函数和用户数据的对象还有另一个与它们相关联的表，称为*环境*。与元数据一样，环境是常规表，多个对象可以共享相同的环境。

线程是共享创建线程的环境而创建的。创建用户数据和C函数，共享创建C函数的环境。非嵌套Lua函数(由[loadfile](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#pdf-loadfile), [loadstring](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#pdf-loadstring)或[load](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#pdf-load))共享创建线程的环境。创建嵌套Lua函数，共享创建Lua函数的环境。

与用户数据相关的环境对Lua没有任何意义。对于程序员来说，将表与用户数据相关联只是一个方便的特性。

与线程关联的环境称为*全球环境*。它们被用作线程和线程创建的非嵌套Lua函数的默认环境，可以由C代码直接访问(请参见[§3.3](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#3.3)).

C代码可以直接访问与C函数关联的环境(请参阅[§3.3](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#3.3))。它用作其他C函数和该函数创建的用户数据的默认环境。

与lua函数关联的环境用于解析函数中对全局变量的所有访问(请参阅[§2.3](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#2.3))。它们用作函数创建的嵌套Lua函数的默认环境。

您可以通过调用lua函数或正在运行的线程来更改其环境。[setfenv](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#pdf-setfenv)。您可以通过调用lua函数或正在运行的线程获得环境。[getfenv](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#pdf-getfenv)。要操作其他对象(userdata、C函数、其他线程)的环境，必须使用CAPI。

**2.10 -****垃圾收集**

Lua执行自动内存管理。这意味着您不必担心为新对象分配内存，也不必担心在不再需要对象时释放内存。Lua通过运行*垃圾收集器*不时地收集所有*死物*(也就是说，不再可以从Lua访问的对象)。Lua使用的所有内存都受到自动管理：表、用户数据、函数、线程、字符串等。

Lua实现了一个增量标记和扫描收集器.它使用两个数字来控制垃圾回收周期：*垃圾收集器暂停*而*垃圾收集器步进乘法器*。两者都使用百分比作为单位(因此100的值意味着内部值1)。

垃圾收集器暂停控制收集器在开始新循环之前等待多长时间。较大的值会减少收集器的侵略性。小于100的值意味着收集器不会等待开始新的循环。值为200意味着收集器在开始新循环之前等待正在使用的总内存加倍。

步骤乘法器控制收集器相对于存储器分配的相对速度。较大的值使收集器更具侵略性，但也增加了每个增量步骤的大小。值小于100会使收集器太慢，并可能导致收集器永远不会完成循环。默认的200表示收集器以“两倍”的内存分配速度运行。

您可以通过调用[lua\_gc](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#lua_gc)以C或[collectgarbage](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#pdf-collectgarbage)在卢阿。使用这些函数，您还可以直接控制收集器(例如，停止并重新启动它)。

**2.10.1 -****垃圾收集元方法**

使用capi，您可以为userdata设置垃圾回收器元方法(请参阅[§2.8](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#2.8))。这些元方法也称为*终结器*。终结器允许您协调Lua的垃圾收集与外部资源管理(例如关闭文件、网络或数据库连接，或释放您自己的内存)。

带字段的垃圾用户数据\_\_gc在它们的元数据中，垃圾收集器不会立即收集。相反，Lua将他们列在一个列表中。在集合之后，Lua对该列表中的每个用户数据执行以下等效函数：

function gc\_event (udata)

local h = metatable(udata).\_\_gc

if h then

h(udata)

end

end

在每个垃圾回收周期的末尾，用户数据的终结器将在*倒转*他们的创造秩序，在那些收集在那个周期。也就是说，要调用的第一个终结器是与程序中最后创建的用户数据相关联的终结器。用户数据本身仅在下一个垃圾收集周期中释放。

**2.10.2 -****弱表**

A *弱表*是其元素为*弱引用*。垃圾收集器忽略弱引用。换句话说，如果对对象的唯一引用是弱引用，那么垃圾收集器将收集这个对象。

弱表可以有弱键、弱值，或者两者都有。带有弱键的表允许收集其键，但阻止其值的收集。具有弱键和弱值的表允许同时收集键和值。在任何情况下，如果收集了键或值，则整个对将从表中移除。表的弱点由\_\_mode它的元域。如果\_\_mode字段是包含字符‘’的字符串。k‘，桌子上的钥匙很弱。如果\_\_mode包含‘v表中的值很弱。

在将表用作可元表之后，不应更改其值。\_\_mode场。否则，由此元可操作控制的表的弱行为是未定义的。

**2.11 -****协同线**

Lua支持cooutines，也称为cooutines。*协同多线程*。Lua中的协同线代表一个独立的执行线程。但是，与多线程系统中的线程不同，Cooutine只通过显式调用产生函数来挂起它的执行。

创建一个协同线，并调用[coroutine.create](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#pdf-coroutine.create)。它的唯一论点是一个函数，它是协同线的主要功能。这个create函数只创建一个新的协同线并返回它的句柄(一个类型的对象)。*螺纹*)；它不启动协同执行。

当你第一次打电话的时候[coroutine.resume](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#pdf-coroutine.resume)作为它的第一个参数传递一个线程。[coroutine.create](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#pdf-coroutine.create)，协同线在其主要功能的第一行开始其执行。传递给[coroutine.resume](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#pdf-coroutine.resume)被传递到协同系统的主要功能。协同线开始运行后，它一直运行到终止或*产量*.

Cooutine可以通过两种方式终止它的执行：通常，当它的主函数返回时(显式或隐式地，在最后一条指令之后)；和异常，如果有一个无保护的错误。在第一种情况下，[coroutine.resume](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#pdf-coroutine.resume)回报**千真万确**，加上Cooutine主函数返回的任何值。如果有错误，[coroutine.resume](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#pdf-coroutine.resume)回报**假的**加上一条错误信息。

协同线通过调用[coroutine.yield](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#pdf-coroutine.yield)。当一个协同线产生时，相应的[coroutine.resume](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#pdf-coroutine.resume)立即返回，即使在嵌套函数调用中(即不是在主函数中，而是在由主函数直接或间接调用的函数中)。在收益的情况下，[coroutine.resume](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#pdf-coroutine.resume)还回来了**千真万确**，再加上传递给[coroutine.yield](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#pdf-coroutine.yield)。当您下一次恢复相同的协同线时，它将继续执行从其屈服点开始，并调用[coroutine.yield](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#pdf-coroutine.yield)返回传递给[coroutine.resume](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#pdf-coroutine.resume).

喜欢[coroutine.create](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#pdf-coroutine.create)，[coroutine.wrap](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#pdf-coroutine.wrap)函数还创建了一个协同线，但是它没有返回cooutine本身，而是返回一个函数，当调用该函数时，该函数将恢复cooutine。传递给此函数的任何参数都作为附加参数传递给[coroutine.resume](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#pdf-coroutine.resume). [coroutine.wrap](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#pdf-coroutine.wrap)返回由[coroutine.resume](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#pdf-coroutine.resume)，除了第一个(布尔错误代码)。不像[coroutine.resume](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#pdf-coroutine.resume), [coroutine.wrap](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#pdf-coroutine.wrap)不会捕获错误；任何错误都会传播到调用方。

例如，考虑以下代码：

function foo (a)

print("foo", a)

return coroutine.yield(2\*a)

end

co = coroutine.create(function (a,b)

print("co-body", a, b)

local r = foo(a+1)

print("co-body", r)

local r, s = coroutine.yield(a+b, a-b)

print("co-body", r, s)

return b, "end"

end)

print("main", coroutine.resume(co, 1, 10))

print("main", coroutine.resume(co, "r"))

print("main", coroutine.resume(co, "x", "y"))

print("main", coroutine.resume(co, "x", "y"))

运行它时，它会产生以下输出：

co-body 1 10

foo 2

main true 4

co-body r

main true 11 -9

co-body x y

main true 10 end

main false cannot resume dead coroutine

**3 -****应用程序接口**

本节描述用于Lua的C API，即主机程序可用于与Lua通信的C函数集。所有api函数及相关类型和常量都在头文件中声明。lua.h.

即使我们使用“函数”一词，API中的任何工具也可以作为宏提供。所有这些宏都只使用它们的每个参数一次(除了第一个参数，它始终是Lua状态)，因此不会产生任何隐藏的副作用。

与大多数C库一样，LuaAPI函数不检查它们的参数是否有效或是否一致。但是，您可以通过使用宏的适当定义编译lua来更改此行为。luai\_apicheck，存档luaconf.h.

**3.1 -****堆栈**

Lua使用*虚拟堆栈*若要向C传递值，此堆栈中的每个元素表示一个Lua值(**零**、数字、字符串等)。

每当Lua调用C时，被调用的函数就会得到一个新堆栈，它独立于以前的堆栈和仍然处于活动状态的C函数堆栈。这个堆栈最初包含C函数的任何参数，它是C函数将其结果推送给调用者的地方(参见[lua\_CFunction](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#lua_CFunction)).

为了方便起见，API中的大多数查询操作都不遵循严格的堆栈规则。相反，它们可以引用堆栈中的任何元素，方法是使用*指数*\*正指数表示*绝对的*堆栈位置(从1开始)；负索引表示*偏移量*相对于堆栈的顶部。更具体地说，如果堆栈具有*n*元素，然后是索引1表示第一个元素(即首先推到堆栈上的元素)和索引。*n*表示最后一个元素；index-1也表示最后一个元素(即顶部的元素)和索引。*-n*表示第一个元素。我们说索引是*有效*如果它位于1和堆栈顶部之间(也就是说，如果1 ≤ abs(index) ≤ top).

**3.2 -****堆栈大小**

当您与LuaAPI交互时，您负责确保一致性。特别是，*您负责控制堆栈溢出。*。您可以使用该函数[lua\_checkstack](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#lua_checkstack)来增加堆栈的大小。

每当Lua调用C时，它至少要确保LUA\_MINSTACK堆栈位置可用。LUA\_MINSTACK定义为20，因此通常不必担心堆栈空间，除非代码有循环将元素推到堆栈上。

大多数查询函数都接受可用堆栈空间中的任何值作为索引，也就是说，索引以所设置的最大堆栈大小为限。[lua\_checkstack](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#lua_checkstack)。这类索引称为*可接受指数*。更正式地说，我们定义了*可接受指数*详情如下：

(index < 0 && abs(index) <= top) ||

(index > 0 && index <= stackspace)

注意，0从来不是一个可接受的索引。

**3.3 -****伪指数**

除非另有说明，否则接受有效索引的任何函数也可以用*伪指数*，它们表示C代码可以访问但不在堆栈中的一些Lua值。伪索引用于访问线程环境、函数环境、注册表和C函数的上值(请参见[§3.4](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#3.4)).

线程环境(全局变量所在的地方)总是处于伪索引状态。LUA\_GLOBALSINDEX。运行C函数的环境总是处于伪索引状态。LUA\_ENVIRONINDEX.

若要访问和更改全局变量的值，可以在环境表上使用常规表操作。例如，要访问全局变量的值，请执行以下操作

lua\_getfield(L, LUA\_GLOBALSINDEX, varname);

**3.4 -****C闭包**

创建C函数时，可以将某些值与其关联，从而创建一个*C闭包*；这些值称为*向上值*，并且只要函数被调用，就可以访问它(请参阅[lua\_pushcclosure](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#lua_pushcclosure)).

每当调用C函数时，其上值都位于特定的伪索引处.这些伪索引是由宏生成的。lua\_upvalueindex。与函数关联的第一个值位于lua\_upvalueindex(1)等等。任何访问lua\_upvalueindex(*n*)，在哪里*n*大于当前函数的上值数(但不大于256)，则生成可接受的(但无效的)索引。

**3.5 -****登记处**

Lua提供*登记处*一个预定义的表，任何C代码都可以使用它来存储它需要存储的Lua值。此表始终位于伪索引处。LUA\_REGISTRYINDEX。任何C库都可以将数据存储到这个表中，但是它应该注意选择不同于其他库使用的键，以避免冲突。通常，您应该使用包含库名称的字符串或代码中带有C对象地址的轻型用户数据作为密钥。

注册表中的整数项由引用机制使用，由辅助库实现，因此不应用于其他目的。

**3.6 -****C中的错误处理**

在内部，Lua使用Clongjmp处理错误的工具。(如果使用C++，也可以选择使用异常；请参阅文件luaconf.h)当Lua面临任何错误(例如内存分配错误、类型错误、语法错误和运行时错误)时，*提高*一个错误，也就是跳远。一个*保护环境*使用setjmp若要设置恢复点，任何错误都会跳转到最近的活动恢复点。

API中的大多数函数都可能引发错误，例如，由于内存分配错误。每个函数的文档指示它是否可以抛出错误。

在C函数中，可以通过调用[lua\_error](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#lua_error).

**3.7 -****功能和类型**

在这里，我们按字母顺序列出了CAPI中的所有函数和类型。每个函数都有这样一个指示符：[-o+p，*x*]

第一个领域，o，是函数从堆栈中弹出的元素数。第二个领域，p，是函数推到堆栈上的元素数。(任何函数总是在弹出参数后推送其结果。)表格中的字段x|y意味着该函数可以推送(或POP)。x或y要素，视情况而定；审讯标记?“这意味着我们不能只看函数的参数(例如，它们可能取决于堆栈上的是什么)就知道函数POPS/推送了多少个元素。”第三个领域，x，告诉函数是否可能抛出错误：‘-“意味着函数永远不会抛出任何错误；”m“意味着该函数可能仅由于内存不足而引发错误；”e“意味着函数可能抛出其他类型的错误；”v表示函数可能故意抛出错误。

**lua\_Alloc**

typedef void \* (\*lua\_Alloc) (void \*ud,

void \*ptr,

size\_t osize,

size\_t nsize);

Lua使用的内存分配函数的类型。分配器函数必须提供类似于realloc，但不完全一样。它的论点是ud，一个不透明指针传递给[lua\_newstate](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#lua_newstate); ptr指向正在分配/重新分配/释放的块的指针；osize，原块的大小；nsize新的块体大小。ptr是NULL当且仅当osize是零。什么时候nsize为零，则分配器必须返回。NULL；如果osize不是零，它应该释放所指向的块。ptr。什么时候nsize不是零，分配器将返回NULL当且仅当它无法填写请求时。什么时候nsize不是零而且osize为零，分配器应表现为malloc。什么时候nsize和osize不是零，分配器的行为如下realloc。Lua假设分配器在osize >= nsize.

下面是分配器函数的一个简单实现。在辅助库中使用的[luaL\_newstate](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#luaL_newstate).

static void \*l\_alloc (void \*ud, void \*ptr, size\_t osize,

size\_t nsize) {

(void)ud; (void)osize; /\* not used \*/

if (nsize == 0) {

free(ptr);

return NULL;

}

else

return realloc(ptr, nsize);

}

此代码假定free(NULL)没有效果realloc(NULL, size)等于malloc(size)。Ansi C确保了这两种行为。

**lua\_atpanic**

[-0, +0, *-*]

lua\_CFunction lua\_atpanic (lua\_State \*L, lua\_CFunction panicf);

设置一个新的恐慌函数并返回旧的。

如果在任何受保护的环境之外发生错误，lua将调用*恐慌函数*然后打电话exit(EXIT\_FAILURE)从而退出主机应用程序。您的恐慌功能可以避免退出，因为永远不会返回(例如，跳远)。

恐慌函数可以访问堆栈顶部的错误消息。

**lua\_call**

[-(Nargs+1)，+nResuls，*e*]

void lua\_call (lua\_State \*L, int nargs, int nresults);

调用一个函数。

要调用函数，必须使用以下协议：首先，要调用的函数被推到堆栈上；然后，函数的参数按直接顺序推送；也就是说，首先推送第一个参数。最后你打电话[lua\_call](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#lua_call); nargs推到堆栈上的参数数。调用函数时，将从堆栈中弹出所有参数和函数值。当函数返回时，函数结果被推到堆栈上。结果的数目调整为nresults，除非nresults是LUA\_MULTRET。在这种情况下，*全*函数的结果被推送。Lua注意返回的值适合堆栈空间。函数结果按直接顺序推送到堆栈上(首先推送第一个结果)，因此调用后最后的结果位于堆栈的顶部。

调用函数中的任何错误都会向上传播(使用longjmp).

下面的示例演示了主机程序如何执行与此Lua代码等价的操作：

a = f("how", t.x, 14)

这里是C语言：

lua\_getfield(L, LUA\_GLOBALSINDEX, "f"); /\* function to be called \*/

lua\_pushstring(L, "how"); /\* 1st argument \*/

lua\_getfield(L, LUA\_GLOBALSINDEX, "t"); /\* table to be indexed \*/

lua\_getfield(L, -1, "x"); /\* push result of t.x (2nd arg) \*/

lua\_remove(L, -2); /\* remove 't' from the stack \*/

lua\_pushinteger(L, 14); /\* 3rd argument \*/

lua\_call(L, 3, 1); /\* call 'f' with 3 arguments and 1 result \*/

lua\_setfield(L, LUA\_GLOBALSINDEX, "a"); /\* set global 'a' \*/

注意，上面的代码是“平衡的”：在它的末尾，堆栈回到了原来的配置。这被认为是一个很好的编程实践。

**lua\_CFunction**

typedef int (\*lua\_CFunction) (lua\_State \*L);

C函数的类型。

为了与Lua进行适当的通信，C函数必须使用以下协议，该协议定义参数和结果的传递方式：C函数按照其堆栈的直接顺序接收来自Lua的参数(首先推送第一个参数)。所以，当函数开始时，lua\_gettop(L)返回函数接收的参数数。第一个参数(如果有的话)位于索引1，其最后一个参数位于索引处。lua\_gettop(L)。要将值返回给Lua，C函数只是按顺序将它们推送到堆栈上(首先推送第一个结果)，并返回结果的数量。结果下面的堆栈中的任何其他值都将被Lua正确地丢弃。与Lua函数一样，Lua调用的C函数也可以返回许多结果。

例如，以下函数接收可变数量的数值参数，并返回它们的平均值和和：

static int foo (lua\_State \*L) {

int n = lua\_gettop(L); /\* number of arguments \*/

lua\_Number sum = 0;

int i;

for (i = 1; i <= n; i++) {

if (!lua\_isnumber(L, i)) {

lua\_pushstring(L, "incorrect argument");

lua\_error(L);

}

sum += lua\_tonumber(L, i);

}

lua\_pushnumber(L, sum/n); /\* first result \*/

lua\_pushnumber(L, sum); /\* second result \*/

return 2; /\* number of results \*/

}

**lua\_checkstack**

[-0, +0, *m*]

int lua\_checkstack (lua\_State \*L, int extra);

确保至少有extra堆栈中的空闲堆栈槽。如果不能将堆栈扩展到该大小，则返回false。此函数从不缩小堆栈；如果堆栈已经大于新大小，则保持不变。

**lua\_close**

[-0, +0, *-*]

void lua\_close (lua\_State \*L);

销毁处于给定lua状态的所有对象(如果有调用相应的垃圾收集元方法)，并释放此状态所使用的所有动态内存。在几个平台上，您可能不需要调用此函数，因为当主机程序结束时，所有资源都会自然释放。另一方面，长时间运行的程序，如守护进程或web服务器，可能需要在不需要状态时立即发布状态，以避免增长过大。

**lua\_concat**

[-n，+1，*e*]

void lua\_concat (lua\_State \*L, int n);

Concatenatesn值位于堆栈顶部，弹出它们，并将结果留在顶部。如果n为1，则结果是堆栈上的单个值(即函数不执行任何操作)；如果n为0，结果是空字符串。连接按照Lua的通常语义执行(请参见[§2.5.4](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#2.5.4)).

**lua\_cpcall**

[-0, +(0|1), *-*]

int lua\_cpcall (lua\_State \*L, lua\_CFunction func, void \*ud);

调用C函数func在保护模式下。func开始时，它的堆栈中只有一个元素，即包含以下内容的轻型用户数据ud。如果有错误，[lua\_cpcall](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#lua_cpcall)返回与[lua\_pcall](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#lua_pcall)，加上堆栈顶部的Error对象；否则，它将返回零，并且不会更改堆栈。返回的所有值func都被丢弃了。

**lua\_createtable**

[-0, +1, *m*]

void lua\_createtable (lua\_State \*L, int narr, int nrec);

创建一个新的空表并将其推送到堆栈上。新表有预先分配的空间narr数组元素和nrec非数组元素。当您确切地知道该表将有多少个元素时，此预分配非常有用。否则，您可以使用该函数。[lua\_newtable](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#lua_newtable).

**lua\_dump**

[-0, +0, *m*]

int lua\_dump (lua\_State \*L, lua\_Writer writer, void \*data);

将函数转储为二进制块。在堆栈顶部接收一个Lua函数，并生成一个二进制块，如果再次加载，将产生一个相当于转储函数的函数。因为它产生了大块，[lua\_dump](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#lua_dump)调用函数writer(见[lua\_Writer](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#lua_Writer))与给定的data写出来。

返回的值是上次调用写入器时返回的错误代码；0表示没有错误。

此函数不会从堆栈中弹出Lua函数。

**lua\_equal**

[-0, +0, *e*]

int lua\_equal (lua\_State \*L, int index1, int index2);

如果在可接受的索引中返回两个值，则返回1。index1和index2是相等的，遵循Lua的语义。==运算符(即，可以调用元方法)。否则返回0。如果任何索引无效，也返回0。

**lua\_error**

[-1, +0, *v*]

int lua\_error (lua\_State \*L);

生成一个Lua错误。错误消息(实际上可以是任何类型的Lua值)必须位于堆栈顶部。这个函数进行跳远，因此永远不会返回。(见[luaL\_error](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#luaL_error)).

**lua\_gc**

[-0, +0, *e*]

int lua\_gc (lua\_State \*L, int what, int data);

控制垃圾收集器。

此函数根据参数的值执行多个任务。what:

* **LUA\_GCSTOP:**停止垃圾收集器。
* **LUA\_GCRESTART:**重新启动垃圾收集器。
* **LUA\_GCCOLLECT:**执行完整的垃圾回收周期。
* **LUA\_GCCOUNT:**返回Lua使用的当前内存量(以字节为单位)。
* **LUA\_GCCOUNTB:**返回将Lua使用的当前字节数除以1024的剩余部分。
* **LUA\_GCSTEP:**执行垃圾回收的增量步骤。步骤“大小”由data(更大的值意味着更多的步骤)以一种非指定的方式。如果要控制步骤大小，则必须通过实验调整data。如果步骤完成垃圾回收周期，则函数返回1。
* **LUA\_GCSETPAUSE:**集data的新值。*停顿*(见[§2.10](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#2.10))。函数返回暂停的前一个值。
* **LUA\_GCSETSTEPMUL:**集data的新值。*步进乘法器*(见[§2.10](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#2.10))。函数返回步骤乘法器的前一个值。

**lua\_getallocf**

[-0, +0, *-*]

lua\_Alloc lua\_getallocf (lua\_State \*L, void \*\*ud);

返回给定状态的内存分配函数。如果ud不是NULL，Lua商店\*ud传递给[lua\_newstate](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#lua_newstate).

**lua\_getfenv**

[-0, +1, *-*]

void lua\_getfenv (lua\_State \*L, int index);

将给定索引下值的环境表推送到堆栈上。

**lua\_getfield**

[-0, +1, *e*]

void lua\_getfield (lua\_State \*L, int index, const char \*k);

将值推到堆栈上。t[k]，在哪里t给定有效索引处的值。与Lua一样，此函数可能会触发“index”事件的元方法(请参见[§2.8](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#2.8)).

**lua\_getglobal**

[-0, +1, *e*]

void lua\_getglobal (lua\_State \*L, const char \*name);

将全局值推送到堆栈上。name。它被定义为一个宏：

#define lua\_getglobal(L,s) lua\_getfield(L, LUA\_GLOBALSINDEX, s)

**lua\_getmetatable**

[-0, +(0|1), *-*]

int lua\_getmetatable (lua\_State \*L, int index);

将给定可接受索引处的值推送到堆栈上。如果索引无效，或者如果该值没有元值，则函数返回0，并在堆栈上不推送任何内容。

**lua\_gettable**

[-1, +1, *e*]

void lua\_gettable (lua\_State \*L, int index);

将值推到堆栈上。t[k]，在哪里t是给定的有效索引处的值，并且k堆栈顶部的值。

此函数从堆栈中弹出键(将结果值放在其位置)。与Lua一样，此函数可能会触发“index”事件的元方法(请参见[§2.8](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#2.8)).

**lua\_gettop**

[-0, +0, *-*]

int lua\_gettop (lua\_State \*L);

返回堆栈中顶部元素的索引。因为索引从1开始，这个结果等于堆栈中的元素数(所以0表示一个空堆栈)。

**lua\_insert**

[-1, +1, *-*]

void lua\_insert (lua\_State \*L, int index);

将顶部元素移动到给定的有效索引中，将高于此索引的元素移到打开空间。不能用伪索引调用，因为伪索引不是实际的堆栈位置.

**lua\_Integer**

typedef ptrdiff\_t lua\_Integer;

LuaAPI用于表示整值的类型。

默认情况下，它是ptrdiff\_t，这通常是最大的符号整体式的机器处理“舒适”。

**lua\_isboolean**

[-0, +0, *-*]

int lua\_isboolean (lua\_State \*L, int index);

如果给定的可接受索引的值具有布尔类型，则返回1，否则返回0。

**lua\_iscfunction**

[-0, +0, *-*]

int lua\_iscfunction (lua\_State \*L, int index);

如果给定的可接受索引的值是C函数，则返回1，否则返回0。

**lua\_isfunction**

[-0, +0, *-*]

int lua\_isfunction (lua\_State \*L, int index);

如果给定的可接受索引的值是函数(C或Lua)，则返回1，否则返回0。

**lua\_islightuserdata**

[-0, +0, *-*]

int lua\_islightuserdata (lua\_State \*L, int index);

如果给定的可接受索引的值是轻型用户数据，则返回1，否则返回0。

**lua\_isnil**

[-0, +0, *-*]

int lua\_isnil (lua\_State \*L, int index);

如果给定可接受索引的值为1，则返回1。**零**，否则为0。

**lua\_isnone**

[-0, +0, *-*]

int lua\_isnone (lua\_State \*L, int index);

如果给定的可接受索引无效(即，它引用当前堆栈之外的元素)，则返回1，否则返回0。

**lua\_isnoneornil**

[-0, +0, *-*]

int lua\_isnoneornil (lua\_State \*L, int index);

如果给定的可接受索引无效(即，它引用当前堆栈之外的元素)，或者如果该索引的值为**零**，否则为0。

**lua\_isnumber**

[-0, +0, *-*]

int lua\_isnumber (lua\_State \*L, int index);

如果给定的可接受索引处的值是可转换为数字的数字或字符串，则返回1，否则返回0。

**lua\_isstring**

[-0, +0, *-*]

int lua\_isstring (lua\_State \*L, int index);

如果给定的可接受索引处的值是字符串或数字(该值总是可转换为字符串)，则返回1，否则返回0。

**lua\_istable**

[-0, +0, *-*]

int lua\_istable (lua\_State \*L, int index);

如果给定的可接受索引的值是表，则返回1，否则返回0。

**lua\_isthread**

[-0, +0, *-*]

int lua\_isthread (lua\_State \*L, int index);

如果给定的可接受索引的值是线程，则返回1，否则返回0。

**lua\_isuserdata**

[-0, +0, *-*]

int lua\_isuserdata (lua\_State \*L, int index);

如果给定的可接受索引处的值为userdata(完整的或轻型的)，则返回1，否则返回0。

**lua\_lessthan**

[-0, +0, *e*]

int lua\_lessthan (lua\_State \*L, int index1, int index2);

如果可接受索引处的值为1，则返回1。index1小于可接受索引处的值。index2，遵循Lua的语义<运算符(即，可以调用元方法)。否则返回0。如果任何索引无效，也返回0。

**lua\_load**

[-0, +1, *-*]

int lua\_load (lua\_State \*L,

lua\_Reader reader,

void \*data,

const char \*chunkname);

装载一块Lua块。如果没有错误，[lua\_load](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#lua_load)将编译后的块作为Lua函数推送到堆栈的顶部。否则，它会推送错误消息。的返回值。[lua\_load](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#lua_load)是：

* **0:**没有错误；
* **LUA\_ERRSYNTAX:**预编译过程中的语法错误；
* [**LUA\_ERRMEM**](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#pdf-LUA_ERRMEM)**:**内存分配错误

此函数只加载一个块；它不运行它。

[lua\_load](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#lua_load)自动检测块是文本还是二进制，并相应地加载(请参阅程序)。luac).

这个[lua\_load](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#lua_load)函数使用用户提供的reader函数来读取块(请参见[lua\_Reader](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#lua_Reader))。这个data参数是传递给读取器函数的不透明值。

这个chunkname参数为块提供一个名称，该块用于错误消息和调试信息(请参见[§3.8](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#3.8)).

**lua\_newstate**

[-0, +0, *-*]

lua\_State \*lua\_newstate (lua\_Alloc f, void \*ud);

创造一个新的、独立的国家。回报NULL如果无法创建状态(由于内存不足)。争论f是分配器函数；Lua通过此函数为此状态分配所有内存。第二个论点，ud，是一个不透明指针，Lua只是在每次调用中传递给分配程序。

**lua\_newtable**

[-0, +1, *m*]

void lua\_newtable (lua\_State \*L);

创建一个新的空表并将其推送到堆栈上。它相当于lua\_createtable(L, 0, 0).

**lua\_newthread**

[-0, +1, *m*]

lua\_State \*lua\_newthread (lua\_State \*L);

创建一个新线程，将其推入堆栈，并返回指向[lua\_State](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#lua_State)表示这个新线程的。此函数返回的新状态与原始状态共享所有全局对象(如表)，但具有独立的执行堆栈。

没有关闭或销毁线程的显式函数。线程受垃圾收集的限制，就像任何Lua对象一样。

**lua\_newuserdata**

[-0, +1, *m*]

void \*lua\_newuserdata (lua\_State \*L, size\_t size);

该函数按照给定的大小分配一个新的内存块，将一个新的完整用户数据推到堆栈上，并返回这个地址。

userdata表示Lua中的C值。一个*全用户数据*表示内存块。它是一个对象(就像一个表)：您必须创建它，它可以有它自己的元对象，并且您可以检测它何时被收集。完整的用户数据仅等于自身(在原始相等下)。

当lua收集完整的用户数据时，gc元方法，Lua调用元方法并将用户数据标记为已完成。当再次收集此用户数据时，Lua将释放其相应的内存。

**lua\_next**

[-1, +(2|0), *e*]

int lua\_next (lua\_State \*L, int index);

从堆栈中弹出一个键，并将给定索引处的表中的键值对推送(给定键后面的“下一个”键对)。如果表中没有其他元素，那么[lua\_next](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#lua_next)返回0(并没有推送)。

典型的遍历如下：

/\* table is in the stack at index 't' \*/

lua\_pushnil(L); /\* first key \*/

while (lua\_next(L, t) != 0) {

/\* uses 'key' (at index -2) and 'value' (at index -1) \*/

printf("%s - %s\n",

lua\_typename(L, lua\_type(L, -2)),

lua\_typename(L, lua\_type(L, -1)));

/\* removes 'value'; keeps 'key' for next iteration \*/

lua\_pop(L, 1);

}

在遍历表时，不要调用[lua\_tolstring](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#lua_tolstring)直接在键上，除非您知道键实际上是字符串。回想一下[lua\_tolstring](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#lua_tolstring) *变化*给定索引处的值；这将混淆下一个调用[lua\_next](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#lua_next).

**lua\_Number**

typedef double lua\_Number;

Lua中的数字类型。默认情况下，它是双倍的，但是可以在luaconf.h.

通过配置文件，您可以将Lua更改为使用另一种类型的数字(例如，浮动或长)。

**lua\_objlen**

[-0, +0, *-*]

size\_t lua\_objlen (lua\_State \*L, int index);

返回给定可接受索引处值的“长度”：对于字符串，这是字符串长度；对于表，这是Length运算符的结果(‘#)；对于userdata，这是分配给userdata的内存块的大小；对于其他值，它是0。

**lua\_pcall**

[-(Nargs+1)，+(NResulx 1)，*-*]

int lua\_pcall (lua\_State \*L, int nargs, int nresults, int errfunc);

以保护模式调用函数。

双管齐下nargs和nresults具有与.的含义相同的含义[lua\_call](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#lua_call)。如果调用过程中没有错误，[lua\_pcall](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#lua_pcall)行为就像[lua\_call](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#lua_call)。但是，如果有任何错误，[lua\_pcall](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#lua_pcall)捕获它，在堆栈上推送一个值(错误消息)，并返回一个错误代码。喜欢[lua\_call](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#lua_call), [lua\_pcall](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#lua_pcall)始终从堆栈中移除函数及其参数。

如果errfunc为0，则堆栈上返回的错误消息完全是原始错误消息。否则，errfunc的堆栈索引。*错误处理函数*。(在当前实现中，此索引不能是伪索引。)在出现运行时错误的情况下，将使用错误消息调用此函数，其返回值为[lua\_pcall](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#lua_pcall).

通常，错误处理程序函数用于向错误消息中添加更多调试信息，例如堆栈跟踪。返回后无法收集此类信息。[lua\_pcall](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#lua_pcall)，从那时起，堆栈就已经打开了。

这个[lua\_pcall](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#lua_pcall)函数在成功的情况下返回0或下列错误代码之一(定义为lua.h):

* **LUA\_ERRRUN:**运行时错误。
* **LUA\_ERRMEM:**内存分配错误对于这些错误，Lua不调用错误处理程序函数。
* **LUA\_ERRERR:**运行错误处理程序函数时出错。

**lua\_pop**

[-n，+0，*-*]

void lua\_pop (lua\_State \*L, int n);

持久性有机污染物n堆栈中的元素。

**lua\_pushboolean**

[-0, +1, *-*]

void lua\_pushboolean (lua\_State \*L, int b);

用值推送布尔值b堆上。

**lua\_pushcclosure**

[-n，+1，*m*]

void lua\_pushcclosure (lua\_State \*L, lua\_CFunction fn, int n);

将一个新的C闭包推到堆栈上。

创建C函数时，可以将某些值与其关联，从而创建C闭包(请参见[§3.4](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#3.4))；当函数被调用时，函数就可以访问这些值。若要将值与C函数关联，首先应将这些值推送到堆栈上(当有多个值时，首先推送第一个值)。然后[lua\_pushcclosure](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#lua_pushcclosure)调用来创建C函数，并使用参数将C函数推到堆栈上。n告诉函数应该关联多少个值。[lua\_pushcclosure](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#lua_pushcclosure)还会从堆栈中弹出这些值。

的最大值n是255

**lua\_pushcfunction**

[-0, +1, *m*]

void lua\_pushcfunction (lua\_State \*L, lua\_CFunction f);

将C函数推到堆栈上。此函数接收到指向C函数的指针，并将类型的Lua值推到堆栈上。function调用时，调用相应的C函数。

在lua中注册的任何函数都必须遵循正确的协议才能接收其参数并返回其结果(请参阅[lua\_CFunction](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#lua_CFunction)).

lua\_pushcfunction定义为宏：

#define lua\_pushcfunction(L,f) lua\_pushcclosure(L,f,0)

**lua\_pushfstring**

[-0, +1, *m*]

const char \*lua\_pushfstring (lua\_State \*L, const char \*fmt, ...);

将格式化字符串推到堆栈上，并返回指向此字符串的指针。它类似于C函数sprintf，但是有一些重要的区别：

* 您不必为结果分配空间：结果是一个Lua字符串，Lua负责内存分配(以及通过垃圾收集进行的去分配)。
* 转换说明符受到很大的限制。没有标志、宽度或精度。转换说明符只能是“%%‘(插入’%“在字符串中)，”%s‘(插入以零结尾的字符串，没有大小限制)，%f‘(插入[lua\_Number](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#lua_Number)), '%p“(将指针插入为十六进制数字)，”%d‘(插入int)，以及%c‘(插入int作为一个角色)。

**lua\_pushinteger**

[-0, +1, *-*]

void lua\_pushinteger (lua\_State \*L, lua\_Integer n);

用值推送一个数字n堆上。

**lua\_pushlightuserdata**

[-0, +1, *-*]

void lua\_pushlightuserdata (lua\_State \*L, void \*p);

将一个轻的用户数据推到堆栈上。

userdata表示Lua中的C值。一个*轻用户数据*表示指针。它是一个值(就像一个数字)：您不创建它，它没有单独的元化，也不被收集(因为它从未被创建)。轻型用户数据等于具有相同C地址的“任意”轻型用户数据。

**lua\_pushliteral**

[-0, +1, *m*]

void lua\_pushliteral (lua\_State \*L, const char \*s);

此宏等效于[lua\_pushlstring](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#lua_pushlstring)，但只能在s是一个文字字符串。在这些情况下，它自动提供字符串长度。

**lua\_pushlstring**

[-0, +1, *m*]

void lua\_pushlstring (lua\_State \*L, const char \*s, size\_t len);

推送所指向的字符串s有尺寸len堆上。Lua生成(或重用)给定字符串的内部副本，因此内存位于s可以在函数返回后立即释放或重用。字符串可以包含嵌入的零。

**lua\_pushnil**

[-0, +1, *-*]

void lua\_pushnil (lua\_State \*L);

将零值推送到堆栈上。

**lua\_pushnumber**

[-0, +1, *-*]

void lua\_pushnumber (lua\_State \*L, lua\_Number n);

用值推送一个数字n堆上。

**lua\_pushstring**

[-0, +1, *m*]

void lua\_pushstring (lua\_State \*L, const char \*s);

将指向的以零结尾的字符串推送。s堆上。Lua生成(或重用)给定字符串的内部副本，因此内存位于s可以在函数返回后立即释放或重用。字符串不能包含嵌入的零；假定它以第一个零结尾。

**lua\_pushthread**

[-0, +1, *-*]

int lua\_pushthread (lua\_State \*L);

所表示的线程。L堆上。如果此线程是其状态的主线程，则返回1。

**lua\_pushvalue**

[-0, +1, *-*]

void lua\_pushvalue (lua\_State \*L, int index);

将给定有效索引处的元素的副本推送到堆栈上。

**lua\_pushvfstring**

[-0, +1, *m*]

const char \*lua\_pushvfstring (lua\_State \*L,

const char \*fmt,

va\_list argp);

相当于[lua\_pushfstring](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#lua_pushfstring)，除非它收到一个va\_list而不是可变数量的参数。

**lua\_rawequal**

[-0, +0, *-*]

int lua\_rawequal (lua\_State \*L, int index1, int index2);

如果在可接受的索引中返回两个值，则返回1。index1和index2基本相等(也就是说，不调用元方法)。否则返回0。如果任何索引无效，也返回0。

**lua\_rawget**

[-1, +1, *-*]

void lua\_rawget (lua\_State \*L, int index);

类似于[lua\_gettable](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#lua_gettable)，但是进行原始访问(即没有元方法)。

**lua\_rawgeti**

[-0, +1, *-*]

void lua\_rawgeti (lua\_State \*L, int index, int n);

将值推到堆栈上。t[n]，在哪里t给定有效索引处的值。访问是原始的；也就是说，它不调用元方法。

**lua\_rawset**

[-2, +0, *m*]

void lua\_rawset (lua\_State \*L, int index);

类似于[lua\_settable](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#lua_settable)，但是执行原始赋值(即，没有元方法)。

**lua\_rawseti**

[-1, +0, *m*]

void lua\_rawseti (lua\_State \*L, int index, int n);

是否相当于t[n] = v，在哪里t是给定的有效索引处的值，并且v堆栈顶部的值。

此函数弹出堆栈中的值。赋值是原始的；也就是说，它不调用元方法。

**lua\_Reader**

typedef const char \* (\*lua\_Reader) (lua\_State \*L,

void \*data,

size\_t \*size);

所使用的读取器功能[lua\_load](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#lua_load)。每次它需要另一块的时候，[lua\_load](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#lua_load)打电话给读者data参数。读取器必须返回一个指向内存块的指针，其中包含一个新的块并进行设置。size整块大小。块必须存在，直到再次调用读取器函数。若要指示块的结尾，读取器必须返回。NULL或设定size降到零。读取器函数可以返回任何大小大于零的部件。

**lua\_register**

[-0, +0, *e*]

void lua\_register (lua\_State \*L,

const char \*name,

lua\_CFunction f);

设置C函数f作为全球的新价值name。它被定义为一个宏：

#define lua\_register(L,n,f) \

(lua\_pushcfunction(L, f), lua\_setglobal(L, n))

**lua\_remove**

[-1, +0, *-*]

void lua\_remove (lua\_State \*L, int index);

移除给定有效索引处的元素，向下移动此索引上的元素以填补空白。不能用伪索引调用，因为伪索引不是实际的堆栈位置.

**lua\_replace**

[-1, +0, *-*]

void lua\_replace (lua\_State \*L, int index);

将顶部元素移动到给定位置(并弹出它)，而不移动任何元素(因此替换给定位置的值)。

**lua\_resume**

[-?, +?, *-*]

int lua\_resume (lua\_State \*L, int narg);

启动并恢复给定线程中的协同线。

要启动协同线程，首先要创建一个新线程(请参见[lua\_newthread](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#lua_newthread))；然后将主函数和任何参数推到它的堆栈上，然后调用[lua\_resume](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#lua_resume)..narg是论点的数量。当协同线挂起或完成其执行时，此调用将返回。返回时，堆栈包含传递给[lua\_yield](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#lua_yield)，或Body函数返回的所有值。[lua\_resume](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#lua_resume)回报[LUA\_YIELD](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#pdf-LUA_YIELD)如果coroutine生成，则如果coroutine在没有错误的情况下完成其执行，则为0；如果出现错误，则为错误代码(请参见[lua\_pcall](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#lua_pcall))。如果出现错误，堆栈将不会打开，因此可以对其使用调试API。错误消息位于堆栈顶部。要重新启动cooutine，您只将要作为结果传递的值放在它的堆栈上。yield，然后打电话[lua\_resume](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#lua_resume).

**lua\_setallocf**

[-0, +0, *-*]

void lua\_setallocf (lua\_State \*L, lua\_Alloc f, void \*ud);

将给定状态的分配器函数更改为f用用户数据ud.

**lua\_setfenv**

[-1, +0, *-*]

int lua\_setfenv (lua\_State \*L, int index);

从堆栈中弹出一个表，并将其设置为给定索引值的新环境。如果给定索引处的值既不是函数，也不是线程，也不是用户数据，[lua\_setfenv](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#lua_setfenv)返回0。否则，它将返回1。

**lua\_setfield**

[-1, +0, *e*]

void lua\_setfield (lua\_State \*L, int index, const char \*k);

是否相当于t[k] = v，在哪里t是给定的有效索引处的值，并且v堆栈顶部的值。

此函数弹出堆栈中的值。与Lua一样，此函数可能会触发“newindex”事件的元方法(请参见[§2.8](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#2.8)).

**lua\_setglobal**

[-1, +0, *e*]

void lua\_setglobal (lua\_State \*L, const char \*name);

从堆栈中弹出一个值，并将其设置为全局值的新值。name。它被定义为一个宏：

#define lua\_setglobal(L,s) lua\_setfield(L, LUA\_GLOBALSINDEX, s)

**lua\_setmetatable**

[-1, +0, *-*]

int lua\_setmetatable (lua\_State \*L, int index);

从堆栈中弹出一个表，并将其设置为给定可接受索引下的值的新元值。

**lua\_settable**

[-2, +0, *e*]

void lua\_settable (lua\_State \*L, int index);

是否相当于t[k] = v，在哪里t是给定有效索引处的值，v堆栈顶部的值，并且k是顶部以下的值。

此函数同时弹出堆栈中的键和值。与Lua一样，此函数可能会触发“newindex”事件的元方法(请参见[§2.8](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#2.8)).

**lua\_settop**

[-?, +?, *-*]

void lua\_settop (lua\_State \*L, int index);

接受任何可接受的索引，或0，并将堆栈顶部设置为此索引。如果新的顶部大于旧的顶部，则新的元素将被填充。**零**。如果index为0，则删除所有堆栈元素。

**lua\_State**

typedef struct lua\_State lua\_State;

保持Lua解释器的整个状态的不透明结构。Lua库是完全可重入的：它没有全局变量。有关状态的所有信息都保存在此结构中。

指向此状态的指针必须作为库中每个函数的第一个参数传递，但[lua\_newstate](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#lua_newstate)，它从零开始创建一个Lua状态。

**lua\_status**

[-0, +0, *-*]

int lua\_status (lua\_State \*L);

返回线程的状态。L.

对于普通线程，状态可以为0，如果线程以错误完成其执行，则为错误代码，或LUA\_YIELD如果线程挂起。

**lua\_toboolean**

[-0, +0, *-*]

int lua\_toboolean (lua\_State \*L, int index);

将给定可接受索引处的Lua值转换为C布尔值(0或1)。就像卢阿的所有测试一样，[lua\_toboolean](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#lua_toboolean)中的任何lua值返回1。**假的**和**零**否则返回0。当使用非有效索引调用时，它还返回0。(如果只接受实际布尔值，请使用[lua\_isboolean](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#lua_isboolean)来测试值的类型。)

**lua\_tocfunction**

[-0, +0, *-*]

lua\_CFunction lua\_tocfunction (lua\_State \*L, int index);

将给定可接受索引处的值转换为C函数。该值必须是C函数；否则，返回NULL.

**lua\_tointeger**

[-0, +0, *-*]

lua\_Integer lua\_tointeger (lua\_State \*L, int index);

将给定可接受索引处的lua值转换为有符号的整型。[lua\_Integer](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#lua_Integer)。lua值必须是可转换为数字的数字或字符串(请参阅[§2.2.1](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#2.2.1))；否则，[lua\_tointeger](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#lua_tointeger)返回0。

如果该数字不是整数，则以某种非指定的方式截断它.

**lua\_tolstring**

[-0, +0, *m*]

const char \*lua\_tolstring (lua\_State \*L, int index, size\_t \*len);

将给定可接受索引处的Lua值转换为C字符串。如果len不是NULL，它还设置了\*len用绳子的长度。lua值必须是字符串或数字；否则，函数将返回NULL。如果值是一个数字，那么[lua\_tolstring](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#lua_tolstring)也*将堆栈中的实际值更改为字符串。*。(这一变化令人困惑[lua\_next](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#lua_next)什么时候[lua\_tolstring](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#lua_tolstring))应用于遍历表期间的键。)

[lua\_tolstring](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#lua_tolstring)返回指向Lua状态内字符串的完全对齐指针。这个字符串总是有一个零(‘\0)在它的最后一个字符之后(如C)，但可以在它的身体中包含其他的零。因为lua有垃圾收集，所以不能保证[lua\_tolstring](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#lua_tolstring)在从堆栈中移除相应的值后，将有效。

**lua\_tonumber**

[-0, +0, *-*]

lua\_Number lua\_tonumber (lua\_State \*L, int index);

将给定可接受索引处的Lua值转换为C类型[lua\_Number](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#lua_Number)(见[lua\_Number](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#lua_Number))。lua值必须是可转换为数字的数字或字符串(请参阅[§2.2.1](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#2.2.1))；否则，[lua\_tonumber](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#lua_tonumber)返回0。

**lua\_topointer**

[-0, +0, *-*]

const void \*lua\_topointer (lua\_State \*L, int index);

将给定可接受索引处的值转换为泛型C指针(void\*)。值可以是用户数据、表、线程或函数；否则，[lua\_topointer](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#lua_topointer)回报NULL。不同的对象会给出不同的指针。无法将指针转换回其原始值。

通常，此函数仅用于调试信息。

**lua\_tostring**

[-0, +0, *m*]

const char \*lua\_tostring (lua\_State \*L, int index);

相当于[lua\_tolstring](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#lua_tolstring)带着len等于NULL.

**lua\_tothread**

[-0, +0, *-*]

lua\_State \*lua\_tothread (lua\_State \*L, int index);

将给定可接受索引处的值转换为Lua线程(表示为lua\_State\*)。此值必须是一个线程；否则，该函数将返回NULL.

**lua\_touserdata**

[-0, +0, *-*]

void \*lua\_touserdata (lua\_State \*L, int index);

如果给定的可接受索引的值是一个完整的用户数据，则返回其块地址。如果该值是一个轻型用户数据，则返回其指针。否则，返回NULL.

**lua\_type**

[-0, +0, *-*]

int lua\_type (lua\_State \*L, int index);

返回给定可接受索引中值的类型，或LUA\_TNONE对于无效索引(即指向“空”堆栈位置的索引)。返回的类型。[lua\_type](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#lua_type)中定义的下列常量编码lua.h: LUA\_TNIL, LUA\_TNUMBER,LUA\_TBOOLEAN, LUA\_TSTRING, LUA\_TTABLE, LUA\_TFUNCTION, LUA\_TUSERDATA, LUA\_TTHREAD，和LUA\_TLIGHTUSERDATA.

**lua\_typename**

[-0, +0, *-*]

const char \*lua\_typename (lua\_State \*L, int tp);

返回由值编码的类型的名称。tp，该值必须是[lua\_type](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#lua_type).

**lua\_Writer**

typedef int (\*lua\_Writer) (lua\_State \*L,

const void\* p,

size\_t sz,

void\* ud);

使用的写入函数的类型。[lua\_dump](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#lua_dump)。每次它再产生一块，[lua\_dump](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#lua_dump)调用编写器，传递要写入的缓冲区(p)，它的大小(sz)，以及data提供给[lua\_dump](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#lua_dump).

作者返回一个错误代码：0表示没有错误；任何其他值表示错误并停止。[lua\_dump](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#lua_dump)再次给作家打电话。

**lua\_xmove**

[-?, +?, *-*]

void lua\_xmove (lua\_State \*from, lua\_State \*to, int n);

类的不同线程之间的交换值。*同*全球状态。

这一职能n堆栈中的值。from，并将它们推到堆栈上。to.

**lua\_yield**

[-?, +?, *-*]

int lua\_yield (lua\_State \*L, int nresults);

产生一种协同作用。

此函数只应作为C函数的返回表达式调用，如下所示：

return lua\_yield (L, nresults);

当C函数调用[lua\_yield](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#lua_yield)这样，运行协同器就会暂停它的执行，并且调用[lua\_resume](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#lua_resume)开始了这个协同线的返回。参数nresults作为结果传递给[lua\_resume](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#lua_resume).

**3.8 -****调试接口**

Lua没有内置的调试工具。相反，它通过函数和*钩钩*。该接口允许构造不同类型的调试器、分析器和其他需要解释器“内部信息”的工具。

**lua\_Debug**

typedef struct lua\_Debug {

int event;

const char \*name; /\* (n) \*/

const char \*namewhat; /\* (n) \*/

const char \*what; /\* (S) \*/

const char \*source; /\* (S) \*/

int currentline; /\* (l) \*/

int nups; /\* (u) number of upvalues \*/

int linedefined; /\* (S) \*/

int lastlinedefined; /\* (S) \*/

char short\_src[LUA\_IDSIZE]; /\* (S) \*/

/\* private part \*/

*other fields*

} lua\_Debug;

一种用于携带关于活动函数的不同信息的结构。[lua\_getstack](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#lua_getstack)只填充此结构的私有部分，供以后使用。的其他字段[lua\_Debug](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#lua_Debug)有有用的信息，打电话[lua\_getinfo](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#lua_getinfo).

田园[lua\_Debug](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#lua_Debug)有以下含义：

* **source:**如果函数是在字符串中定义的，则source是那根绳子。如果函数是在文件中定义的，则source以“@‘后面跟着文件名。
* **short\_src:**的“可打印”版本source，用于错误信息。
* **linedefined:**函数定义开始的行号。
* **lastlinedefined:**函数定义结束的行号。
* **what:**弦"Lua"如果这个函数是Lua函数，"C"如果是C函数，"main"如果它是块的主要部分，并且"tail"如果是函数做了尾叫的话。在后一种情况下，Lua没有关于该函数的其他信息。
* **currentline:**执行给定函数的当前行。当没有可用的线路信息时，currentline设置为-1。
* **name:**给定函数的合理名称。因为Lua中的函数是一流的值，所以它们没有固定的名称：有些函数可以是多个全局变量的值，而另一些函数只能存储在表字段中。这个lua\_getinfo函数检查如何调用该函数以找到合适的名称。如果它找不到名字，那么name设置为NULL.
* **namewhat:**解释name场。价值namewhat可以"global", "local", "method", "field", "upvalue"，或""(空字符串)，根据函数的调用方式。(Lua在其他选项似乎不适用时使用空字符串。)
* **nups:**函数的上值数。

**lua\_gethook**

[-0, +0, *-*]

lua\_Hook lua\_gethook (lua\_State \*L);

返回当前挂钩函数。

**lua\_gethookcount**

[-0, +0, *-*]

int lua\_gethookcount (lua\_State \*L);

返回当前挂钩计数。

**lua\_gethookmask**

[-0, +0, *-*]

int lua\_gethookmask (lua\_State \*L);

返回当前挂钩掩码。

**lua\_getinfo**

[-(0|1), +(0|1|2), *m*]

int lua\_getinfo (lua\_State \*L, const char \*what, lua\_Debug \*ar);

返回有关特定函数或函数调用的信息。

要获取有关函数调用的信息，参数ar必须是由上一次调用填充的有效激活记录。[lua\_getstack](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#lua_getstack)或用作钩子的参数(见[lua\_Hook](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#lua_Hook)).

若要获取有关函数的信息，请将其推入堆栈并启动what字符串与字符‘>“”(在这种情况下，lua\_getinfo)将函数弹出到堆栈的顶部。)例如，要知道函数在哪一行f定义后，可以编写以下代码：

lua\_Debug ar;

lua\_getfield(L, LUA\_GLOBALSINDEX, "f"); /\* get global 'f' \*/

lua\_getinfo(L, ">S", &ar);

printf("%d\n", ar.linedefined);

字符串中的每个字符what选择结构的某些字段ar要填充或要推送到堆栈上的值：

* **'n':**填充字段name和namewhat;
* **'S':**填充字段source, short\_src, linedefined, lastlinedefined，和what;
* **'l':**填充字段currentline;
* **'u':**填充字段nups;
* **'f':**将运行在给定级别上的函数推到堆栈上；
* **'L':**将一个表推到堆栈上，该表的索引是对函数有效的行数。(A)*有效行*是带有一些相关代码的行，也就是可以放置断点的行。无效行包括空行和注释。)

此函数在出错时返回0(例如，what).

**lua\_getlocal**

[-0, +(0|1), *-*]

const char \*lua\_getlocal (lua\_State \*L, lua\_Debug \*ar, int n);

获取有关给定激活记录的局部变量的信息。参数ar必须是由上一次调用填充的有效激活记录。[lua\_getstack](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#lua_getstack)或用作钩子的参数(见[lua\_Hook](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#lua_Hook))。指数n选择要检查的局部变量(1是第一个参数，还是活动局部变量，以此类推，直到最后一个活动局部变量)。[lua\_getlocal](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#lua_getlocal)将变量的值推入堆栈并返回其名称。

变量名以‘(‘(开括号)表示内部变量(循环控制变量、临时变量和C函数局部变量)。

回报NULL当索引大于活动局部变量的数量时(并且不推送)。

**lua\_getstack**

[-0, +0, *-*]

int lua\_getstack (lua\_State \*L, int level, lua\_Debug \*ar);

获取有关解释器运行时堆栈的信息。

此函数填充[lua\_Debug](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#lua_Debug)结构的标识。*激活记录*在给定级别上执行的函数。级别0是当前运行的函数，而级别是*n+1*是调用了level的函数。*n*。当没有错误的时候，[lua\_getstack](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#lua_getstack)返回1；当调用级别大于堆栈深度时，它返回0。

**lua\_getupvalue**

[-0, +(0|1), *-*]

const char \*lua\_getupvalue (lua\_State \*L, int funcindex, int n);

获取有关闭包的上值的信息。(对于Lua函数，upvalue是函数使用的外部局部变量，因此包含在其闭包中。)[lua\_getupvalue](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#lua_getupvalue)获取索引。n将上值的值推送到堆栈上，并返回其名称。funcindex指向堆栈中的闭包。(Upvalue没有特定的顺序，因为它们在整个函数中都是活动的。因此，它们是按任意顺序编号的。)

回报NULL当索引大于向上值的数量时(并且不推送)。对于C函数，此函数使用空字符串""作为所有向上值的名称。

**lua\_Hook**

typedef void (\*lua\_Hook) (lua\_State \*L, lua\_Debug \*ar);

用于调试钩子函数的类型。

每当调用钩子时，其ar争论有它的领域event设置为触发钩子的特定事件。Lua用以下常量标识这些事件：LUA\_HOOKCALL, LUA\_HOOKRET, LUA\_HOOKTAILRET, LUA\_HOOKLINE，和LUA\_HOOKCOUNT。此外，对于行事件，字段currentline也设置好了。中的任何其他字段的值。ar，钩子必须调用[lua\_getinfo](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#lua_getinfo)。对于返回事件，event可以LUA\_HOOKRET，正常值，或LUA\_HOOKTAILRET。在后一种情况下，lua模拟从执行尾调用的函数返回；在这种情况下，调用是无用的。[lua\_getinfo](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#lua_getinfo).

当Lua运行钩子时，它会禁用对钩子的其他调用。因此，如果钩子调用Lua以执行函数或块，则无需调用钩子即可执行此执行。

**lua\_sethook**

[-0, +0, *-*]

int lua\_sethook (lua\_State \*L, lua\_Hook f, int mask, int count);

设置调试挂钩函数。

论辩f是钩子函数。mask指定将在哪些事件上调用钩子：它由按位或常量组成。LUA\_MASKCALL, LUA\_MASKRET, LUA\_MASKLINE，和LUA\_MASKCOUNT。这个count参数只有在掩码包括以下内容时才有意义。LUA\_MASKCOUNT。对于每个事件，调用钩子如下所述：

* **呼叫钩：**在解释器调用函数时调用。在Lua进入新函数之后，在函数获得其参数之前调用钩子。
* **返回钩：**当解释器从函数返回时调用。在Lua离开函数之前调用钩子。您无法访问要由函数返回的值。
* **线钩：**当解释器即将开始执行新代码行时调用，或者当解释器跳回代码时(甚至跳回同一行代码)。(此事件仅在Lua执行Lua函数时发生。)
* **伯爵钩：**在解释器执行count指示。(此事件仅在Lua执行Lua函数时发生。)

通过设置一个钩子是禁用的。mask降到零。

**lua\_setlocal**

[-(0|1), +0, *-*]

const char \*lua\_setlocal (lua\_State \*L, lua\_Debug \*ar, int n);

设置给定激活记录的局部变量的值。参数ar和n就像在[lua\_getlocal](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#lua_getlocal)(见[lua\_getlocal](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#lua_getlocal)). [lua\_setlocal](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#lua_setlocal)将堆栈顶部的值分配给变量并返回其名称。它还弹出堆栈中的值。

回报NULL(如果索引大于活动局部变量的数量，则不会弹出任何值)。

**lua\_setupvalue**

[-(0|1), +0, *-*]

const char \*lua\_setupvalue (lua\_State \*L, int funcindex, int n);

设置闭包上值的值。它将堆栈顶部的值分配给上值，并返回其名称。它还弹出堆栈中的值。参数funcindex和n就像在[lua\_getupvalue](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#lua_getupvalue)(见[lua\_getupvalue](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#lua_getupvalue)).

回报NULL(当索引大于向上值的数目时，不会弹出任何值)。

**4 -****辅助图书馆**

这个*辅助图书馆*为C与Lua接口提供了几个方便的函数。虽然基本API为C和Lua之间的所有交互提供了基本函数，但辅助库为一些常见任务提供了更高级别的函数。

辅助库中的所有函数都在头文件中定义。lauxlib.h并且有一个前缀luaL\_.

辅助库中的所有函数都构建在基本API之上，因此它们提供了无法使用此API完成的任何操作。

辅助库中的几个函数用于检查C函数参数。他们的名字总是luaL\_check\*或luaL\_opt\*。如果检查不满意，所有这些函数都会抛出错误。因为错误消息是为参数格式化的(例如，“bad argument #1“)，不应将这些函数用于其他堆栈值。

**4.1 -****功能和类型**

在这里，我们按字母顺序列出了辅助库中的所有函数和类型。

**luaL\_addchar**

[-0, +0, *m*]

void luaL\_addchar (luaL\_Buffer \*B, char c);

添加字符c到缓冲区B(见[luaL\_Buffer](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#luaL_Buffer)).

**luaL\_addlstring**

[-0, +0, *m*]

void luaL\_addlstring (luaL\_Buffer \*B, const char \*s, size\_t l);

将指向的字符串添加到s有长l到缓冲区B(见[luaL\_Buffer](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#luaL_Buffer))。字符串可能包含嵌入的零。

**luaL\_addsize**

[-0, +0, *m*]

void luaL\_addsize (luaL\_Buffer \*B, size\_t n);

添加到缓冲区B(见[luaL\_Buffer](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#luaL_Buffer))一串长度n以前复制到缓冲区(请参阅[luaL\_prepbuffer](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#luaL_prepbuffer)).

**luaL\_addstring**

[-0, +0, *m*]

void luaL\_addstring (luaL\_Buffer \*B, const char \*s);

添加指向的以零结尾的字符串。s到缓冲区B(见[luaL\_Buffer](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#luaL_Buffer))。字符串可能不包含嵌入的零。

**luaL\_addvalue**

[-1, +0, *m*]

void luaL\_addvalue (luaL\_Buffer \*B);

将堆栈顶部的值添加到缓冲区。B(见[luaL\_Buffer](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#luaL_Buffer))。弹出值。

这是字符串缓冲区上唯一可以(而且必须)使用堆栈上的额外元素调用的函数，该元素是要添加到缓冲区的值。

**luaL\_argcheck**

[-0, +0, *v*]

void luaL\_argcheck (lua\_State \*L,

int cond,

int narg,

const char \*extramsg);

检查是否cond是真的。如果没有，则使用以下消息引发错误，其中func从调用堆栈检索：

bad argument #<narg> to <func> (<extramsg>)

**luaL\_argerror**

[-0, +0, *v*]

int luaL\_argerror (lua\_State \*L, int narg, const char \*extramsg);

使用以下消息引发错误，其中func从调用堆栈检索：

bad argument #<narg> to <func> (<extramsg>)

这个函数永远不会返回，但是在C函数中使用它是一个成语return luaL\_argerror(*args*).

**luaL\_Buffer**

typedef struct luaL\_Buffer luaL\_Buffer;

类型为a*串缓冲器*.

字符串缓冲区允许C代码分段构建Lua字符串。其使用方式如下：

* 首先声明一个变量b类型[luaL\_Buffer](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#luaL_Buffer).
* 然后通过调用初始化它。luaL\_buffinit(L, &b).
* 然后将字符串片段添加到缓冲区中，调用luaL\_add\*职能。
* 你以打电话结束luaL\_pushresult(&b)。此调用将最后的字符串保留在堆栈的顶部。

在正常操作期间，字符串缓冲区使用可变数量的堆栈槽。因此，在使用缓冲区时，不能假设您知道堆栈的顶部在哪里。只要使用平衡，就可以在对缓冲区操作的连续调用之间使用堆栈；也就是说，当调用缓冲区操作时，堆栈处于上一次缓冲区操作之后的同一级别。(这条规则的唯一例外是[luaL\_addvalue](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#luaL_addvalue))打电话后[luaL\_pushresult](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#luaL_pushresult)当缓冲区被初始化时，堆栈回到了它的级别，加上它顶部的最后一个字符串。

**luaL\_buffinit**

[-0, +0, *-*]

void luaL\_buffinit (lua\_State \*L, luaL\_Buffer \*B);

初始化缓冲区B。此函数不分配任何空间；必须将缓冲区声明为变量(请参阅[luaL\_Buffer](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#luaL_Buffer)).

**luaL\_callmeta**

[-0, +(0|1), *e*]

int luaL\_callmeta (lua\_State \*L, int obj, const char \*e);

调用元方法。

如果索引处的对象obj有一个可元的，而这个元的有一个字段e，此函数调用此字段并将对象作为其唯一参数传递。在本例中，此函数返回1并将调用返回的值推到堆栈上。如果没有元方法或元方法，则此函数返回0(而不推送堆栈上的任何值)。

**luaL\_checkany**

[-0, +0, *v*]

void luaL\_checkany (lua\_State \*L, int narg);

检查函数是否有任何类型的参数(包括**零**)就位narg.

**luaL\_checkint**

[-0, +0, *v*]

int luaL\_checkint (lua\_State \*L, int narg);

检查函数参数是否narg是一个数字，并将此数字转换为int.

**luaL\_checkinteger**

[-0, +0, *v*]

lua\_Integer luaL\_checkinteger (lua\_State \*L, int narg);

检查函数参数是否narg是一个数字，并将此数字转换为[lua\_Integer](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#lua_Integer).

**luaL\_checklong**

[-0, +0, *v*]

long luaL\_checklong (lua\_State \*L, int narg);

检查函数参数是否narg是一个数字，并将此数字转换为long.

**luaL\_checklstring**

[-0, +0, *v*]

const char \*luaL\_checklstring (lua\_State \*L, int narg, size\_t \*l);

检查函数参数是否narg是字符串，并返回此字符串；如果l不是NULL填充\*l用绳子的长度。

此函数使用[lua\_tolstring](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#lua_tolstring)为了得到它的结果，所以这个函数的所有转换和警告都适用于这里。

**luaL\_checknumber**

[-0, +0, *v*]

lua\_Number luaL\_checknumber (lua\_State \*L, int narg);

检查函数参数是否narg是一个数字，并返回这个数字。

**luaL\_checkoption**

[-0, +0, *v*]

int luaL\_checkoption (lua\_State \*L,

int narg,

const char \*def,

const char \*const lst[]);

检查函数参数是否narg是一个字符串，并在数组中搜索此字符串。lst(必须为空终止)。返回找到字符串的数组中的索引。如果参数不是字符串或找不到字符串，则引发错误。

如果def不是NULL，该函数使用def在没有参数的情况下作为默认值。narg或者如果这个论点是**零**.

这是一个将字符串映射到C枚举的有用函数。(Lua库中通常的惯例是使用字符串而不是数字来选择选项。)

**luaL\_checkstack**

[-0, +0, *v*]

void luaL\_checkstack (lua\_State \*L, int sz, const char \*msg);

将堆栈大小增加到top + sz元素，如果堆栈不能扩展到该大小，则会引发错误。msg是要进入错误消息的附加文本。

**luaL\_checkstring**

[-0, +0, *v*]

const char \*luaL\_checkstring (lua\_State \*L, int narg);

检查函数参数是否narg是一个字符串，并返回此字符串。

此函数使用[lua\_tolstring](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#lua_tolstring)为了得到它的结果，所以这个函数的所有转换和警告都适用于这里。

**luaL\_checktype**

[-0, +0, *v*]

void luaL\_checktype (lua\_State \*L, int narg, int t);

检查函数参数是否narg有型t。看见[lua\_type](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#lua_type)类型的编码t.

**luaL\_checkudata**

[-0, +0, *v*]

void \*luaL\_checkudata (lua\_State \*L, int narg, const char \*tname);

检查函数参数是否narg类型的用户数据。tname(见[luaL\_newmetatable](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#luaL_newmetatable)).

**luaL\_dofile**

[-0, +?, *m*]

int luaL\_dofile (lua\_State \*L, const char \*filename);

加载并运行给定的文件。它被定义为以下宏：

(luaL\_loadfile(L, filename) || lua\_pcall(L, 0, LUA\_MULTRET, 0))

如果没有错误，则返回0，如果出现错误，则返回1。

**luaL\_dostring**

[-0, +?, *m*]

int luaL\_dostring (lua\_State \*L, const char \*str);

加载并运行给定的字符串。它被定义为以下宏：

(luaL\_loadstring(L, str) || lua\_pcall(L, 0, LUA\_MULTRET, 0))

如果没有错误，则返回0，如果出现错误，则返回1。

**luaL\_error**

[-0, +0, *v*]

int luaL\_error (lua\_State \*L, const char \*fmt, ...);

引发错误。错误消息格式由fmt加上任何额外的参数，遵循相同的规则[lua\_pushfstring](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#lua_pushfstring)。它还会在消息的开头添加文件名和发生错误的行号(如果可用的话)。

这个函数永远不会返回，但是在C函数中使用它是一个成语return luaL\_error(*args*).

**luaL\_getmetafield**

[-0, +(0|1), *m*]

int luaL\_getmetafield (lua\_State \*L, int obj, const char \*e);

将字段推到堆栈上。e从对象在索引处的元数据obj。如果对象没有元对象，或者如果元对象没有此字段，则返回0并没有推送任何内容。

**luaL\_getmetatable**

[-0, +1, *-*]

void luaL\_getmetatable (lua\_State \*L, const char \*tname);

将与名称关联的元数据推到堆栈上。tname在注册表中(见[luaL\_newmetatable](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#luaL_newmetatable)).

**luaL\_gsub**

[-0, +1, *m*]

const char \*luaL\_gsub (lua\_State \*L,

const char \*s,

const char \*p,

const char \*r);

创建字符串的副本。s通过替换字符串的任何出现p用绳子r。在堆栈上推送结果字符串并返回它。

**luaL\_loadbuffer**

[-0, +1, *m*]

int luaL\_loadbuffer (lua\_State \*L,

const char \*buff,

size\_t sz,

const char \*name);

以Lua块的形式加载缓冲区。此函数使用[lua\_load](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#lua_load)若要加载缓冲区中指向的块，请执行以下操作buff有尺寸sz.

此函数返回与[lua\_load](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#lua_load). name块名，用于调试信息和错误消息。

**luaL\_loadfile**

[-0, +1, *m*]

int luaL\_loadfile (lua\_State \*L, const char \*filename);

以Lua块的形式加载文件。此函数使用[lua\_load](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#lua_load)来加载名为filename。如果filename是NULL，然后从标准输入加载。如果文件的第一行以#.

此函数返回与[lua\_load](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#lua_load)，但是它有一个额外的错误代码。LUA\_ERRFILE如果它无法打开/读取文件。

如[lua\_load](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#lua_load)，此函数只加载块；它不运行它。

**luaL\_loadstring**

[-0, +1, *m*]

int luaL\_loadstring (lua\_State \*L, const char \*s);

以Lua块的形式加载字符串。此函数使用[lua\_load](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#lua_load)若要在以零结尾的字符串中加载块，请执行以下操作：s.

此函数返回与[lua\_load](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#lua_load).

亦如[lua\_load](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#lua_load)，此函数只加载块；它不运行它。

**luaL\_newmetatable**

[-0, +1, *m*]

int luaL\_newmetatable (lua\_State \*L, const char \*tname);

如果注册表已经拥有该项tname，返回0。否则，创建一个新表作为用户数据的元数据，并使用键将其添加到注册表中。tname，并返回1。

在这两种情况下，将与tname在登记处。

**luaL\_newstate**

[-0, +0, *-*]

lua\_State \*luaL\_newstate (void);

创建一个新的Lua状态。它叫[lua\_newstate](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#lua_newstate)基于标准C的分配器realloc函数，然后设置一个恐慌函数(请参见[lua\_atpanic](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#lua_atpanic))，它在发生致命错误时将错误消息打印到标准错误输出。

返回新状态，或NULL如果存在内存分配错误。

**luaL\_openlibs**

[-0, +0, *m*]

void luaL\_openlibs (lua\_State \*L);

将所有标准Lua库打开到给定状态。

**luaL\_optint**

[-0, +0, *v*]

int luaL\_optint (lua\_State \*L, int narg, int d);

如果函数参数narg是一个数字，则将此数字转换为int。如果这个论点不存在或者是**零**、回报d。否则，引发错误。

**luaL\_optinteger**

[-0, +0, *v*]

lua\_Integer luaL\_optinteger (lua\_State \*L,

int narg,

lua\_Integer d);

如果函数参数narg是一个数字，则将此数字转换为[lua\_Integer](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#lua_Integer)。如果这个论点不存在或者是**零**、回报d。否则，引发错误。

**luaL\_optlong**

[-0, +0, *v*]

long luaL\_optlong (lua\_State \*L, int narg, long d);

如果函数参数narg是一个数字，则将此数字转换为long。如果这个论点不存在或者是**零**、回报d。否则，引发错误。

**luaL\_optlstring**

[-0, +0, *v*]

const char \*luaL\_optlstring (lua\_State \*L,

int narg,

const char \*d,

size\_t \*l);

如果函数参数narg是字符串，则返回此字符串。如果这个论点不存在或者是**零**、回报d。否则，引发错误。

如果l不是NULL，填补该职位\*l结果的长度。

**luaL\_optnumber**

[-0, +0, *v*]

lua\_Number luaL\_optnumber (lua\_State \*L, int narg, lua\_Number d);

如果函数参数narg是一个数字，返回这个数字。如果这个论点不存在或者是**零**、回报d。否则，引发错误。

**luaL\_optstring**

[-0, +0, *v*]

const char \*luaL\_optstring (lua\_State \*L,

int narg,

const char \*d);

如果函数参数narg是字符串，则返回此字符串。如果这个论点不存在或者是**零**、回报d。否则，引发错误。

**luaL\_prepbuffer**

[-0, +0, *-*]

char \*luaL\_prepbuffer (luaL\_Buffer \*B);

将地址返回到大小的空间。LUAL\_BUFFERSIZE中，可以复制要添加到缓冲区的字符串。B(见[luaL\_Buffer](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#luaL_Buffer))。将字符串复制到此空格后，必须调用[luaL\_addsize](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#luaL_addsize)使用字符串的大小将其实际添加到缓冲区中。

**luaL\_pushresult**

[-?, +1, *m*]

void luaL\_pushresult (luaL\_Buffer \*B);

完成缓冲区的使用B将最后的字符串留在堆栈的顶部。

**luaL\_ref**

[-1, +0, *m*]

int luaL\_ref (lua\_State \*L, int t);

创建并返回*参照系*，在索引表中t，用于堆栈顶部的对象(并弹出该对象)。

引用是唯一的整数键。只要不手动将整数键添加到表中t, [luaL\_ref](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#luaL_ref)确保它返回的键的唯一性。您可以检索引用所引用的对象。r打电话lua\_rawgeti(L, t, r)。功能[luaL\_unref](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#luaL_unref)释放引用及其关联对象。

如果堆栈顶部的对象是**零**, [luaL\_ref](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#luaL_ref)返回常量LUA\_REFNIL。常数LUA\_NOREF返回的引用不同。[luaL\_ref](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#luaL_ref).

**luaL\_Reg**

typedef struct luaL\_Reg {

const char \*name;

lua\_CFunction func;

} luaL\_Reg;

要注册的函数数组的类型[luaL\_register](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#luaL_register). name是函数名和func是指向函数的指针。任何数组[luaL\_Reg](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#luaL_Reg)必须以一个哨兵条目结尾，在该条目中，两者均为name和func是NULL.

**luaL\_register**

[-(0|1), +1, *m*]

void luaL\_register (lua\_State \*L,

const char \*libname,

const luaL\_Reg \*l);

打开一个图书馆。

当被调用时libname等于NULL，它只是在列表中注册所有函数。l(见[luaL\_Reg](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#luaL_Reg))放入堆栈顶部的表中。

当使用非空调用时libname, luaL\_register创建一个新表。t，将其设置为全局变量的值。libname，将其设置为package.loaded[libname]，并在其上注册列表中的所有函数。l。如果有一张桌子package.loaded[libname]或可变的libname重用此表，而不是创建新表。

无论如何，函数将表留在堆栈的顶部。

**luaL\_typename**

[-0, +0, *-*]

const char \*luaL\_typename (lua\_State \*L, int index);

返回给定索引处值的类型的名称。

**luaL\_typerror**

[-0, +0, *v*]

int luaL\_typerror (lua\_State \*L, int narg, const char \*tname);

使用如下消息生成一个错误：

*location*: bad argument *narg* to '*func*' (*tname* expected, got *rt*)

哪里*location*是由[luaL\_where](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#luaL_where), *func*当前函数的名称，以及*rt*实际参数的类型名称。

**luaL\_unref**

[-0, +0, *-*]

void luaL\_unref (lua\_State \*L, int t, int ref);

释放参考ref表中的索引t(见[luaL\_ref](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#luaL_ref))。从表中删除该项，以便可以收集引用的对象。参考文献ref也被释放用于再次使用。

如果ref是[LUA\_NOREF](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#pdf-LUA_NOREF)或[LUA\_REFNIL](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#pdf-LUA_REFNIL), [luaL\_unref](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#luaL_unref)什么都不做。

**luaL\_where**

[-0, +1, *m*]

void luaL\_where (lua\_State \*L, int lvl);

将标识控件当前位置的字符串推送到堆栈上。lvl在呼叫堆栈中。通常，此字符串具有以下格式：

*chunkname*:*currentline*:

级别0是运行函数，级别1是调用运行函数的函数，等等。

此函数用于为错误消息构建前缀。

**5 -****标准图书馆**

标准Lua库提供了直接通过CAPI实现的有用功能。其中一些职能为语言提供必要的服务(例如，[type](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#pdf-type)和[getmetatable](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#pdf-getmetatable))；其他服务提供“外部”服务(例如I/O)；其他服务可以在Lua本身实现，但非常有用，或者具有重要的性能要求，值得在C中实现(例如，[table.sort](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#pdf-table.sort)).

所有库都通过官方的CAPI实现，并作为单独的C模块提供。目前，Lua有以下标准库：

* 基础图书馆，

其中包括协同线子库；

* 包装库；
* 字符串操作；
* 表操作；
* 数学函数(SIN、LOG等)；
* 投入和产出；
* 操作系统设施；
* 调试工具。

除了基本库和包库之外，每个库都将其所有函数作为全局表的字段或其对象的方法提供。

要访问这些库，C宿主程序应该调用[luaL\_openlibs](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#luaL_openlibs)函数，它打开所有标准库。或者，它可以通过调用luaopen\_base(基本图书馆)，luaopen\_package(用于包库)，luaopen\_string(用于字符串库)，luaopen\_table(适用于表库)，luaopen\_math(数学图书馆)，luaopen\_io(供I/O图书馆使用)，luaopen\_os(适用于操作系统库)luaopen\_debug(用于调试库)。这些函数在lualib.h不应直接调用：您必须像调用任何其他Lua C函数一样调用它们，例如，使用[lua\_call](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#lua_call).

**5.1 -****基本功能**

基础库为Lua提供了一些核心功能。如果应用程序中没有包含此库，则应仔细检查是否需要为其某些功能提供实现。

**assert (v [, message])**

Issues an error when the value of its argument v is false (i.e., **零** or **假的**); otherwise, returns all its arguments. message is an error message; when absent, it defaults to "assertion failed!"

**collectgarbage (opt [, arg])**

此函数是垃圾收集器的通用接口。它根据第一个参数执行不同的功能，opt:

* **“停止”：**停止垃圾收集器。
* **“重新启动”：**重新启动垃圾收集器。
* **“收取”：**执行完整的垃圾回收周期。
* **“数”：**返回Lua使用的总内存(以字节为单位)。
* **“步骤”：**执行垃圾收集步骤。步骤“大小”由arg(更大的值意味着更多的步骤)以一种非指定的方式。如果要控制步骤大小，则必须通过实验调整arg。回报**千真万确**如果步骤完成了收集周期。
* **“设置暂停”：**集arg的新值。*停顿*(见[§2.10](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#2.10))。返回前一个值*停顿*.
* **“setStemul”：**集arg的新值。*步进乘法器*(见[§2.10](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#2.10))。返回前一个值*步进*.

**dofile (filename)**

Opens the named file and executes its contents as a Lua chunk. When called without arguments, dofile executes the contents of the standard input (stdin). Returns all values returned by the chunk. In case of errors, dofile propagates the error to its caller (that is, dofile does not run in protected mode).

**error (message [, level])**

Terminates the last protected function called and returns message as the error message. Function error never returns.

通常，error在邮件开头添加一些有关错误位置的信息。这个level参数指定如何获取错误位置。对于级别1(默认值)，错误位置位于error函数被调用。第2级将错误指向调用error被称为；以此类推。传递级别0可避免将错误位置信息添加到消息中。

**\_G**

A global variable (not a function) that holds the global environment (that is, \_G.\_G = \_G). Lua itself does not use this variable; changing its value does not affect any environment, nor vice-versa. (Use [setfenv](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#pdf-setfenv) to change environments.)

**getfenv ([f])**

Returns the current environment in use by the function. f can be a Lua function or a number that specifies the function at that stack level: Level 1 is the function calling getfenv. If the given function is not a Lua function, or if f is 0, getfenv returns the global environment. The default for f is 1.

**getmetatable (object)**

如果object没有元，返回**零**。否则，如果对象的元对象具有"\_\_metatable"字段返回关联的值。否则，返回给定对象的元对象。

**ipairs (t)**

返回三个值：迭代器函数，表t，以及0，所以这个建筑

for i,v in ipairs(t) do *body* end

将遍历这些对(1,t[1]), (2,t[2])，到表中缺少的第一个整数键为止。

**load (func [, chunkname])**

使用函数加载块func去拿它的碎片。每次呼叫func必须返回与先前结果连接的字符串。空字符串的返回，**零**，或者没有值表示块的结束。

如果没有错误，则将已编译的块作为函数返回；否则，返回**零**加上错误信息。返回函数的环境是全局环境。

chunkname用作错误消息和调试信息的块名。缺席时，默认为“=(load)".

**loadfile ([filename])**

类似于[load](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#pdf-load)，但是从文件中获取块。filename或者来自标准输入，如果没有指定文件名。

**loadstring (string [, chunkname])**

类似于[load](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#pdf-load)，但是从给定的字符串中获取块。

要加载并运行给定的字符串，请使用以下成语

assert(loadstring(s))()

不在的时候，chunkname默认为给定字符串。

**next (table [, index])**

允许程序遍历表的所有字段。它的第一个参数是一个表，第二个参数是这个表中的一个索引。next返回表的下一个索引及其关联值。当被调用时**零**作为第二个论点，next返回初始索引及其关联值。当用最后一个索引调用时，或者用**零**在空桌子上，next回报**零**。如果没有第二个参数，则解释为**零**。特别是，您可以使用next(t)若要检查表是否为空，请执行以下操作。

未指定枚举索引的顺序，*即使是数值指数*。(若要按数字顺序遍历表，请使用数字。**为**或者[ipairs](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#pdf-ipairs))

.的行为next是*未定*如果在遍历期间，将任何值赋值给表中不存在的字段.但是，您可以修改现有字段。特别是，您可以清除现有字段。

**pairs (t)**

返回三个值：[next](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#pdf-next)函数，表t，和**零**，让建筑

for k,v in pairs(t) do *body* end

将遍历表的所有键值对。t.

见函数[next](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#pdf-next)用于在表遍历期间修改表时的注意事项。

**pcall (f, arg1, ···)**

调用函数f中的给定参数*保护模式*。这意味着内部的任何错误f不是传播，而是，pcall捕获错误并返回状态代码。其第一个结果是状态代码(布尔值)，如果调用成功而没有错误，则为真。在这种情况下，pcall还返回调用的所有结果，在此第一个结果之后。如果有任何错误，pcall回报**假的**加上错误信息。

**print (···)**

Receives any number of arguments, and prints their values to stdout, using the [tostring](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#pdf-tostring) function to convert them to strings. print is not intended for formatted output, but only as a quick way to show a value, typically for debugging. For formatted output, use [string.format](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#pdf-string.format).

**rawequal (v1, v2)**

Checks whether v1 is equal to v2, without invoking any metamethod. Returns a boolean.

**rawget (table, index)**

Gets the real value of table[index], without invoking any metamethod. table must be a table; index may be any value.

**rawset (table, index, value)**

Sets the real value of table[index] to value, without invoking any metamethod. table must be a table, index any value different from **零**, and value any Lua value.

此函数返回table.

**select (index, ···)**

如果index是一个数字，在参数号之后返回所有参数。index。否则，index一定是字符串"#"，和select返回它收到的额外参数的总数。

**setfenv (f, table)**

设置要由给定函数使用的环境。f可以是Lua函数，也可以是指定堆栈级别函数的数字：第1级是调用函数。setfenv. setfenv返回给定的函数。

作为特例，当f是0setfenv更改正在运行的线程的环境。在这种情况下，setfenv不返回值。

**setmetatable (table, metatable)**

设置给定表的元可选。(您不能从Lua更改其他类型的元数据，而只能从C.)如果metatable是**零**，移除给定表的元值。如果原始元对象具有"\_\_metatable"字段引发一个错误。

此函数返回table.

**tonumber (e [, base])**

Tries to convert its argument to a number. If the argument is already a number or a string convertible to a number, then tonumber returns this number; otherwise, it returns **零**.

可选参数指定要解释数字的基。基可以是2到36之间的任意整数，包括。在10以上的碱基中，字母‘’A‘(大写或小写)表示10，’B‘表示11，以此类推，与’Z“代表35。在基数10(默认值)中，这个数字可以有一个十进制部分，也可以有一个可选的指数部分(参见[§2.1](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#2.1))。在其他基中，只接受无符号整数。

**tostring (e)**

Receives an argument of any type and converts it to a string in a reasonable format. For complete control of how numbers are converted, use [string.format](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#pdf-string.format).

如果e有一个"\_\_tostring"场，那么tostring调用相应的值e作为参数，并使用调用的结果作为其结果。

**type (v)**

Returns the type of its only argument, coded as a string. The possible results of this function are "nil" (a string, not the value **零**), "number", "string", "boolean", "table", "function", "thread", and "userdata".

**unpack (list [, i [, j]])**

Returns the elements from the given table. This function is equivalent to

return list[i], list[i+1], ···, list[j]

但上述代码只能为固定数量的元素编写。默认情况下，i是1和j列表的长度，由Length运算符定义(请参见[§2.5.5](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#2.5.5)).

**\_VERSION**

A global variable (not a function) that holds a string containing the current interpreter version. The current contents of this variable is "Lua 5.1".

**xpcall (f, err)**

此函数类似于[pcall](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#pdf-pcall)，但您可以设置一个新的错误处理程序。

xpcall调用函数f在保护模式下，使用err作为错误处理程序。任何内部错误f不是传播，而是，xpcall捕获错误，调用err函数，并返回状态代码。其第一个结果是状态代码(布尔值)，如果调用成功而没有错误，则为真。在这种情况下，xpcall还返回调用的所有结果，在此第一个结果之后。如果有任何错误，xpcall回报**假的**加上来自err.

**5.2 -****Cooutine操纵**

与协同器相关的操作由基本库的一个子库组成，并位于表中。coroutine。看见[§2.11](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#2.11)关于协同线的一般描述。

**coroutine.create (f)**

创建一个新的协同线，与身体f. f一定是Lua函数。返回这个新的协同线，它是一个具有"thread".

**coroutine.resume (co [, val1, ···])**

开始或继续执行协同线co。当你第一次恢复一条协同线时，它就开始运行它的身体。价值val1，···作为对身体功能的参数传递。如果合作线已经屈服了，resume重新启动；值val1，···作为产率的结果。

如果协同线运行时没有任何错误，resume回报**千真万确**加上传递给yield(如果coroutine产生)或Body函数返回的任何值(如果cooutine终止)。如果有任何错误，resume回报**假的**加上错误信息。

**coroutine.running ()**

返回运行的协同线，或**零**当主线程调用时。

**coroutine.status (co)**

返回coroutine的状态。co，作为字符串："running"，如果协同线正在运行(也就是说，它调用status); "suspended"，如果coroutine在调用yield，或者如果它还没有开始运行；"normal"如果协同线是活动的，但没有运行(也就是说，它已经恢复了另一个协同线)；以及"dead"如果协同线已经完成了它的身体功能，或者它已经停止了一个错误。

**coroutine.wrap (f)**

创建一个新的协同线，与身体f. f一定是Lua函数。返回一个函数，该函数每次调用协同线时都会恢复该函数。传递给该函数的任何参数都表现为resume。返回由resume，除了第一个布尔值。如果出现错误，则传播错误。

**coroutine.yield (···)**

暂停调用协同线的执行。Cooutine不能运行C函数、元方法或迭代器。任何论点yield作为附加结果传递给resume.

**5.3 -****模块**

包库为在Lua中加载和构建模块提供了基本设施。它在全球环境中直接输出其两项职能：[require](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#pdf-require)和[module](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#pdf-module)。其他的都在表中导出。package.

**module (name [, ···])**

创建一个模块。如果有一张桌子package.loaded[name]这张表就是模块。否则，如果存在全局表t使用给定的名称，这个表就是模块。否则会创建一个新表。t，并将其设置为全局name的价值package.loaded[name]。此函数还初始化t.\_NAME有了这个名字，t.\_M与模块(t)，以及t.\_PACKAGE使用包名(整个模块名减去最后一个组件；请参见下面)。最后，module集t作为当前功能的新环境和新的价值package.loaded[name]，所以[require](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#pdf-require)回报t.

如果name是一个复合名称(即由点分隔的组件)，module为每个组件创建(或重用，如果已经存在)表。例如，如果name是a.b.c，然后module将模块表存储在字段中。c场域b全球a.

此函数可以接收可选的*备选方案*在模块名称之后，其中每个选项都是要应用于模块上的函数。

**require (modname)**

加载给定模块。函数从查看[package.loaded](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#pdf-package.loaded)表以确定是否modname已经装好了。如果是，那么require返回存储在package.loaded[modname]。否则，它会试图找到一个*装载机*为了模块。

找到一个装载机，require是由[package.loaders](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#pdf-package.loaders)阵列。通过更改这个数组，我们可以更改require寻找一个模块。下面的解释是基于默认配置的[package.loaders](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#pdf-package.loaders).

第一require查询package.preload[modname]。如果它有一个值，这个值(应该是一个函数)是加载程序。不然的话require中存储的路径搜索Lua加载程序。[package.path](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#pdf-package.path)。如果同样失败，它将使用存储在[package.cpath](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#pdf-package.cpath)。如果同样失败，它将尝试*全合一*装载机(见[package.loaders](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#pdf-package.loaders)).

一旦找到装载机，require用一个参数调用加载程序，modname。如果加载程序返回任何值，require将返回的值赋值给package.loaded[modname]。如果加载程序没有返回任何值，并且没有将任何值分配给package.loaded[modname]，然后require亚细亚**千真万确**这个条目。无论如何,require返回package.loaded[modname].

如果加载或运行模块有任何错误，或者无法找到模块的任何加载程序，则require发出错误信号。

**package.cpath**

使用的路径[require](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#pdf-require)搜索C加载程序。

Lua初始化C路径[package.cpath](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#pdf-package.cpath)以同样的方式初始化lua路径。[package.path](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#pdf-package.path)，使用环境变量LUA\_CPATH中定义的默认路径。luaconf.h.

**package.loaded**

一张表[require](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#pdf-require)若要控制已加载的模块，请执行以下操作。当您需要一个模块时modname和package.loaded[modname]不是假的，[require](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#pdf-require)只需返回存储在那里的值。

**package.loaders**

一张表[require](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#pdf-require)以控制如何加载模块。

表中的每个条目都是*搜索函数*。在寻找模块时，[require](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#pdf-require)按升序调用这些搜索器中的每一个，并使用模块名(给出的参数为[require](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#pdf-require))作为它的唯一参数。该函数可以返回另一个函数(模块)。*装载机*)或者一个字符串来解释为什么它找不到那个模块(或者**零**如果它没有什么可说的)。Lua用四个函数初始化这个表。

第一个搜索器只是在[package.preload](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#pdf-package.preload)桌子。

第二个搜索器使用存储在[package.path](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#pdf-package.path)。路径是一系列的*模板*用分号隔开。对于每个模板，搜索程序将更改模板中的每个询问标记。filename，它是模块名，每个点替换为“目录分隔符”(例如“/“在unix中；然后它将尝试打开生成的文件名。

"./?.lua;./?.lc;/usr/local/?/init.lua"

查找模块的lua文件foo将尝试打开文件./foo.lua, ./foo.lc，和/usr/local/foo/init.lua，按顺序排列。

第三个搜索器使用变量给出的路径作为C库查找加载程序[package.cpath](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#pdf-package.cpath)。例如，如果C路径是字符串

"./?.so;./?.dll;/usr/local/?/init.so"

模块搜索器foo将尝试打开文件./foo.so, ./foo.dll，和/usr/local/foo/init.so，按顺序排列。一旦找到C库，这个搜索器首先使用动态链接工具将应用程序链接到库。然后，它试图在库中找到一个C函数作为加载程序。这个C函数的名称是字符串“luaopen\_“与模块名称的副本连接，其中每个点被下划线替换。此外，如果模块名有连字符，则它的前缀(并包括)第一个连字符被移除。例如，如果模块名是a.v1-b.c，函数名为luaopen\_b\_c.

第四个搜索者尝试*全合一装载机*。它在C路径中搜索库，以查找给定模块的根名。例如，当要求a.b.c，它将搜索C库a。如果找到，它将查找子模块的开放函数；在我们的示例中，这将是luaopen\_a\_b\_c。有了这个工具，一个包就可以将几个C子模块打包到一个库中，每个子模块保持原来的开放功能。

**package.loadlib (libname, funcname)**

将主机程序与C库动态链接libname。在这个库中，查找一个函数funcname并将此函数作为C函数返回。(所以，funcname必须遵循协议(请参阅[lua\_CFunction](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#lua_CFunction))).

这是一个低级函数。它完全绕过了包和模块系统。不像[require](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#pdf-require)，它不执行任何路径搜索，也不自动添加扩展。libname必须是C库的完整文件名，必要时包括路径和扩展名。funcname必须是C库导出的确切名称(它可能取决于所使用的C编译器和链接器)。

ANSI C.不支持此功能。因此，它只能在某些平台(Windows、Linux、MacOSX、Solaris、BSD以及其他支持dlfcn标准)。

**package.path**

使用的路径[require](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#pdf-require)寻找一个Lua装载机。

启动时，lua使用环境变量的值初始化此变量。LUA\_PATH中定义的默认路径。luaconf.h，如果没有定义环境变量。任何“;;“在环境变量的值中，将替换为默认路径。

**package.preload**

用于存储特定模块的加载器的表(请参见[require](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#pdf-require)).

**package.seeall (module)**

为module与其\_\_index字段引用全局环境，以便本模块从全局环境继承值。用作功能的选项[module](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#pdf-module).

**5.4 -****字符串操作**

该库为字符串操作提供了通用函数，例如查找和提取子字符串以及模式匹配。在Lua中索引字符串时，第一个字符位于位置1(而不是像C中的0)。索引被允许为负值，并被解释为从字符串末尾向后索引。因此，最后一个字符位于位置-1，依此类推。

字符串库提供表中的所有函数。string。它还为字符串设置了一个可转换的值，其中\_\_index字段指向string桌子。因此，您可以使用面向对象样式的字符串函数.例如,string.byte(s, i)可以写成s:byte(i).

字符串库采用一个字节字符编码。

**string.byte (s [, i [, j]])**

Returns the internal numerical codes of the characters s[i], s[i+1], ···, s[j]. The default value for i is 1; the default value for j is i.

请注意，数字代码不一定是跨平台可移植的。

**string.char (···)**

Receives zero or more integers. Returns a string with length equal to the number of arguments, in which each character has the internal numerical code equal to its corresponding argument.

请注意，数字代码不一定是跨平台可移植的。

**string.dump (function)**

返回包含给定函数的二进制表示形式的字符串，以便以后的[loadstring](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#pdf-loadstring)在此字符串上返回函数的副本。function必须是没有向上值的Lua函数。

**string.find (s, pattern [, init [, plain]])**

Looks for the first match of pattern in the string s. If it finds a match, then find returns the indices of s where this occurrence starts and ends; otherwise, it returns **零**. A third, optional numerical argument init specifies where to start the search; its default value is 1 and can be negative. A value of **千真万确** as a fourth, optional argument plain turns off the pattern matching facilities, so the function does a plain "find substring" operation, with no characters in patternbeing considered "magic". Note that if plain is given, then init must be given as well.

如果模式已经捕获，那么在成功匹配时，捕获的值也会在这两个索引之后返回。

**string.format (formatstring, ···)**

Returns a formatted version of its variable number of arguments following the description given in its first argument (which must be a string). The format string follows the same rules as the printf family of standard C functions. The only differences are that the options/modifiers \*, l, L, n, p, and h are not supported and that there is an extra option, q. The q option formats a string in a form suitable to be safely read back by the Lua interpreter: the string is written between double quotes, and all double quotes, newlines, embedded zeros, and backslashes in the string are correctly escaped when written. For instance, the call

string.format('%q', 'a string with "quotes" and \n new line')

将产生字符串：

"a string with \"quotes\" and \

new line"

选择c, d, E, e, f, g, G, i, o, u, X，和x所有的人都期望一个数字作为参数，而q和s期待一根绳子。

此函数不接受包含嵌入零的字符串值，除非作为q选择。

**string.gmatch (s, pattern)**

Returns an iterator function that, each time it is called, returns the next captures from pattern over string s. If pattern specifies no captures, then the whole match is produced in each call.

作为一个例子，下面的循环

s = "hello world from Lua"

for w in string.gmatch(s, "%a+") do

print(w)

end

将遍历字符串中的所有单词。s，每行打印一个。下一个示例收集所有对key=value从给定的字符串到表：

t = {}

s = "from=world, to=Lua"

for k, v in string.gmatch(s, "(%w+)=(%w+)") do

t[k] = v

end

对于这个函数，a‘^‘在模式开始时，它并不是一个锚，因为这会阻止迭代。

**string.gsub (s, pattern, repl [, n])**

Returns a copy of s in which all (or the first n, if given) occurrences of the pattern have been replaced by a replacement string specified by repl, which can be a string, a table, or a function. gsub also returns, as its second value, the total number of matches that occurred.

如果repl是字符串，则其值用于替换。人物%作为转义字符：repl表格%*n*..*n*在1到9之间，表示*n*-捕获的子字符串(见下文)。序列%0代表整个比赛。序列%%代表一个%.

如果repl是一个表，然后查询每个匹配项的表，使用第一个捕获作为键；如果模式指定没有捕获，则整个匹配被用作键。

如果repl是一个函数，那么这个函数在每次发生匹配时都会被调用，所有捕获的子字符串都按顺序传递为参数；如果模式指定没有捕获，那么整个匹配将作为唯一的参数传递。

如果表查询或函数调用返回的值是字符串或数字，则将其用作替换字符串；否则，则为**假的**或**零**，则没有替换(即原始匹配保留在字符串中)。

以下是一些例子：

x = string.gsub("hello world", "(%w+)", "%1 %1")

--> x="hello hello world world"

x = string.gsub("hello world", "%w+", "%0 %0", 1)

--> x="hello hello world"

x = string.gsub("hello world from Lua", "(%w+)%s\*(%w+)", "%2 %1")

--> x="world hello Lua from"

x = string.gsub("home = $HOME, user = $USER", "%$(%w+)", os.getenv)

--> x="home = /home/roberto, user = roberto"

x = string.gsub("4+5 = $return 4+5$", "%$(.-)%$", function (s)

return loadstring(s)()

end)

--> x="4+5 = 9"

local t = {name="lua", version="5.1"}

x = string.gsub("$name-$version.tar.gz", "%$(%w+)", t)

--> x="lua-5.1.tar.gz"

**string.len (s)**

Receives a string and returns its length. The empty string "" has length 0. Embedded zeros are counted, so "a\000bc\000" has length 5.

**string.lower (s)**

Receives a string and returns a copy of this string with all uppercase letters changed to lowercase. All other characters are left unchanged. The definition of what an uppercase letter is depends on the current locale.

**string.match (s, pattern [, init])**

Looks for the first *匹配* of pattern in the string s. If it finds one, then match returns the captures from the pattern; otherwise it returns **零**. If pattern specifies no captures, then the whole match is returned. A third, optional numerical argument init specifies where to start the search; its default value is 1 and can be negative.

**string.rep (s, n)**

Returns a string that is the concatenation of n copies of the string s.

**string.reverse (s)**

Returns a string that is the string s reversed.

**string.sub (s, i [, j])**

Returns the substring of s that starts at i and continues until j; i and j can be negative. If j is absent, then it is assumed to be equal to -1 (which is the same as the string length). In particular, the call string.sub(s,1,j) returns a prefix of s with length j, and string.sub(s, -i) returns a suffix of s with length i.

**string.upper (s)**

Receives a string and returns a copy of this string with all lowercase letters changed to uppercase. All other characters are left unchanged. The definition of what a lowercase letter is depends on the current locale.

**5.4.1 -****模式**

**人物类别：**

A *字符类*用于表示一组字符。在描述字符类时允许以下组合：

* ***x*:**(在哪里*x*不是*魔法人物* ^$()%.[]\*+-?)表示字符*x*本身。
* **.:**(点)代表所有字符。
* **%a:**代表所有字母。
* **%c:**表示所有控制字符。
* **%d:**表示所有数字。
* **%l:**表示所有小写字母。
* **%p:**表示所有标点符号。
* **%s:**表示所有空格字符。
* **%u:**表示所有大写字母。
* **%w:**表示所有字母数字字符。
* **%x:**表示所有十六进制数字。
* **%z:**表示表示为0的字符。
* **%*x*:**(在哪里*x*是任何非字母数字字符)表示该字符。*x*。这是逃避魔法人物的标准方法。任何标点符号(即使是非魔法字符)都可以在“%当用一个模式来表示自己时。
* **[*set*]:**表示类，该类是*集*。值分隔区域的结束字符，可以指定字符范围的范围。-“”所有课程%*x*上面描述的组件也可以用作*集*。所有其他字符*集*代表自己。例如,[%w\_](或[\_%w])表示所有字母数字字符加上下划线，[0-7]表示八进制数字，以及[0-7%l%-]表示八进制数字加上小写字母加上-性格。

未定义范围和类之间的交互。因此，像这样的模式[%a-z]或[a-%%]毫无意义。

* **[^*set*]:**表示…的补足*集*，在哪里*集*被解释为上述。

对于所有由单个字母表示的类(%a, %c)，相应的大写字母表示类的补码。例如,%S表示所有非空格字符。

字母、空格和其他字符组的定义取决于当前的区域设置。尤其是这个班[a-z]可能不等于%l.

**图案项目：**

A *模式项目*可以

* 单个字符类，与类中的任何单个字符匹配；
* 一个字符类，后面跟着“\*‘，它与类中的0或多个字符重复匹配。这些重复项目将永远匹配最长的可能序列；
* 一个字符类，后面跟着“+‘，它与类中的一个或多个字符重复匹配。这些重复项目将永远匹配最长的可能序列；
* 一个字符类，后面跟着“-‘，它还匹配类中的0或多个字符重复。不像‘\*‘，这些重复项将始终与*最短*可能的顺序；
* 一个字符类，后面跟着“?‘，它与类中字符的0或1匹配；
* %*n*，为*n*在1到9之间；这样的项匹配一个子字符串，该子字符串等于*n*-捕获的字符串(见下文)；
* %b*xy*，在哪里*x*和*y*是两个不同的字符；这样的项匹配以*x*，以*y*，在哪里*x*和*y*是*平衡*。这意味着，如果从左到右读取字符串，则计数。*+1*为了*x*和*-1*为了*y*，结局*y*是第一个*y*数到0的地方。例如，项目%b()将表达式与平衡括号匹配。

**模式：**

A *花纹*是模式项的序列。A‘^‘在模式的开头，在主题字符串的开头锚定匹配。A‘$‘在模式的末尾，将匹配锚定在主题字符串的末尾。在其他位置，‘^“和”$“没有什么特别的意思，代表自己。

**捕获：**

模式可以包含括在括号中的子模式；它们描述*捕捉*。当匹配成功时，将存储匹配捕获的主题字符串的子字符串(*被俘*)供将来使用。捕获按其左括号编号。例如，在模式中"(a\*(.)%w(%s\*))"，字符串匹配的部分。"a\*(.)%w(%s\*)"存储为第一次捕获(因此有数字1)；字符匹配“.“用数字2捕获，并与之匹配”%s\*“3号。

作为特例，空捕获()捕获当前字符串位置(一个数字)。例如，如果我们应用模式"()aa()"在绳子上"flaaap"，将有两个捕获：3和5。

模式不能包含嵌入的零。使用%z相反。

**5.5 -****表操作**

此库为表操作提供通用函数。它提供表中的所有功能。table.

表库中的大多数函数假定表表示数组或列表。对于这些函数，当我们谈论一个表的“长度”时，我们指的是长度运算符的结果。

**table.concat (table [, sep [, i [, j]]])**

Given an array where all elements are strings or numbers, returns table[i]..sep..table[i+1] ··· sep..table[j]. The default value for sep is the empty string, the default for i is 1, and the default for j is the length of the table. If i is greater than j, returns the empty string.

**table.insert (table, [pos,] value)**

插入元件value就位pos在……里面table，必要时，将其他元素移到开放空间。的默认值pos是n+1，在哪里n表的长度(请参见[§2.5.5](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#2.5.5))，所以一个电话table.insert(t,x)镶片x在桌子的末尾t.

**table.maxn (table)**

返回给定表的最大正数值索引，如果该表没有正数值索引，则返回零。(为了完成它的工作，这个函数对整个表进行线性遍历。)

**table.remove (table [, pos])**

移除table位置元素pos，如果有必要，移动其他元素以关闭空间。返回已删除元素的值。的默认值pos是n，在哪里n是表的长度，因此调用table.remove(t)移除表的最后一个元素。t.

**table.sort (table [, comp])**

Sorts table elements in a given order, *就地*, from table[1] to table[n], where n is the length of the table. If comp is given, then it must be a function that receives two table elements, and returns true when the first is less than the second (so that not comp(a[i+1],a[i]) will be true after the sort). If comp is not given, then the standard Lua operator < is used instead.

排序算法是不稳定的；也就是说，按给定顺序认为相等的元素可能会因排序而改变其相对位置。

**5.6 -****数学函数**

这个库是标准C数学库的接口。它提供表中的所有功能。math.

**math.abs (x)**

返回x.

**math.acos (x)**

返回的弧余弦x(以弧度表示)。

**math.asin (x)**

返回的弧正弦x(以弧度表示)。

**math.atan (x)**

返回的弧切线x(以弧度表示)。

**math.atan2 (y, x)**

返回的弧切线y/x(以弧度表示)，但使用两个参数的符号来查找结果的象限。(它还正确地处理了x为零。)

**math.ceil (x)**

返回大于或等于x.

**math.cos (x)**

返回余弦x(假定为弧形)。

**math.cosh (x)**

返回双曲余弦x.

**math.deg (x)**

返回角度x(以弧度表示)以度表示。

**math.exp (x)**

返回值*ex*.

**math.floor (x)**

返回小于或等于x.

**math.fmod (x, y)**

的除法的其余部分。x通过y使商数接近于零。

**math.frexp (x)**

回报m和e使.*X=m2e*, e为整数，其绝对值为m在范围内*[0.5, 1)*(或零时x是零)。

**math.huge**

价值HUGE\_VAL大于或等于任何其他数值的值。

**math.ldexp (m, e)**

回报*M2e* (e应该是整数)。

**math.log (x)**

返回自然对数x.

**math.log10 (x)**

返回基数-10的对数。x.

**math.max (x, ···)**

返回其参数中的最大值。

**math.min (x, ···)**

返回其参数中的最小值。

**math.modf (x)**

返回两个数字，即x的分数部分x.

**math.pi**

价值*派*.

**math.pow (x, y)**

回报*xy*。(您也可以使用表达式x^y若要计算此值，请执行以下操作。)

**math.rad (x)**

返回角度x(以度表示)以弧度表示。

**math.random ([m [, n]])**

此函数是简单伪随机生成器函数的接口。rand由ANSI C.(不能保证其统计性质。)

当不带参数调用时，返回范围内的一致伪随机实数。*[0,1)*。当用整数调用时m, math.random返回范围内的一致伪随机整数。*[1，m]*。用两个整数调用时m和n, math.random返回范围内的一致伪随机整数。*[男，女]*.

**math.randomseed (x)**

集x作为伪随机发生器的“种子”：相等的种子产生相等的数列。

**math.sin (x)**

返回x(假定为弧形)。

**math.sinh (x)**

返回双曲正弦x.

**math.sqrt (x)**

返回x。(您也可以使用表达式x^0.5若要计算此值，请执行以下操作。)

**math.tan (x)**

返回x(假定为弧形)。

**math.tanh (x)**

的双曲切线。x.

**5.7 -****输入和输出设施**

I/O库为文件操作提供了两种不同的样式。第一种方法使用隐式文件描述符；也就是说，存在设置默认输入文件和默认输出文件的操作，并且所有输入/输出操作都在这些默认文件之上。第二种样式使用显式文件描述符。

当使用隐式文件描述符时，所有操作都由表提供。io。使用显式文件描述符时，[io.open](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#pdf-io.open)返回文件描述符，然后所有操作都作为文件描述符的方法提供。

桌子io还提供了三个预定义的文件描述符，它们通常来自C：io.stdin, io.stdout，和io.stderr。I/O库从不关闭这些文件。

除非另有说明，否则所有I/O函数都会返回。**零**失败(加上第二个结果的错误消息和第三个结果与系统相关的错误代码)和一些不同于**零**关于成功。

**io.close ([file])**

相当于file:close()。无file关闭默认输出文件。

**io.flush ()**

相当于file:flush在默认输出文件上。

**io.input ([file])**

当使用文件名调用时，它将打开命名文件(在文本模式下)，并将其句柄设置为默认输入文件。当使用文件句柄调用时，它只是将此文件句柄设置为默认输入文件。当没有参数调用时，它返回当前的默认输入文件。

如果出现错误，此函数将引发错误，而不是返回错误代码。

**io.lines ([filename])**

以读模式打开给定的文件名，并返回一个迭代器函数，每次调用它时，该函数从文件中返回一个新行。因此，建设

for line in io.lines(filename) do *body* end

将遍历文件的所有行。当迭代器函数检测到文件结束时，它将返回**零**(完成循环)并自动关闭文件。

打电话io.lines()(没有文件名)等于io.input():lines()也就是说，它遍历默认输入文件的行。在这种情况下，循环结束时它不会关闭文件。

**io.open (filename [, mode])**

此函数以字符串中指定的模式打开文件。mode。它返回一个新的文件句柄，或者，如果出现错误，**零**加上一条错误信息。

这个mode字符串可以是以下任一内容：

* **“r”：**读取模式(默认)；
* **“W”：**写作模式；
* **“A”：**附加模式；
* **“R+”：**更新模式，保留所有以前的数据；
* **“w+”：**更新模式，删除所有以前的数据；
* **“a+”：**附加更新模式，保留以前的数据，只允许在文件末尾写入。

这个mode字符串也可以具有“b“最后，在某些系统中，需要在二进制模式下打开文件。这个字符串正是在标准C函数中使用的fopen.

**io.output ([file])**

类似于[io.input](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#pdf-io.input)，但是对默认输出文件进行操作。

**io.popen (prog [, mode])**

启动程序prog在分隔的进程中，并返回一个文件句柄，您可以使用该文件句柄从该程序读取数据(如果mode是"r"，或将数据写入此程序(如果mode是"w").

此功能依赖于系统，并非在所有平台上都可用。

**io.read (···)**

相当于io.input():read.

**io.tmpfile ()**

返回临时文件的句柄。此文件以更新模式打开，并在程序结束时自动删除。

**io.type (obj)**

检查是否obj是一个有效的文件句柄。返回字符串。"file"如果obj是一个打开的文件句柄，"closed file"如果obj是关闭的文件句柄，或**零**如果obj不是文件句柄。

**io.write (···)**

相当于io.output():write.

**file:close ()**

关闭file。注意，当文件句柄被垃圾收集时，文件会自动关闭，但这需要不可预测的时间。

**file:flush ()**

将任何书面数据保存到file.

**file:lines ()**

返回一个迭代器函数，每次调用它时，该函数从文件中返回一个新行。因此，建设

for line in file:lines() do *body* end

将遍历文件的所有行。(不像[io.lines](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#pdf-io.lines)，此函数在循环结束时不会关闭文件。)

**file:read (···)**

读取文件file，根据给定的格式，指定要读取的内容。对于每种格式，该函数返回一个字符串(或一个数字)，其中包含读取的字符，或**零**如果它不能读取指定格式的数据。当调用时没有格式，它使用默认格式读取整个下一行(见下文)。

可用格式如下

* **“\*n”：**读取数字；这是唯一返回数字而不是字符串的格式。
* **“\*a”：**从当前位置开始读取整个文件。在文件末尾，它返回空字符串。
* **“\*L”：**读取下一行(跳过行尾)，返回**零**在文件的末尾。这是默认格式。
* ***数*:**读取具有最多此字符数的字符串，然后返回**零**在文件的末尾。如果数字为零，则不读取任何内容，并返回空字符串，或**零**在文件的末尾。

**file:seek ([whence] [, offset])**

设置并获取文件位置，从文件的开头到offset加上字符串指定的基。whence，如下：

* **“设定”：**基是位置0(文件的开头)；
* **“CUR”：**基地为当前位置；
* **“结束”：**基础是文件的结尾；

在成功的情况下，功能seek返回文件的最后位置，从文件开始时以字节为单位。如果此函数失败，则返回**零**，加上一个描述错误的字符串。

的默认值whence是"cur"，以及为了offset是0。因此，呼叫file:seek()返回当前文件位置，而不更改它；调用file:seek("set")将位置设置为文件的开头(并返回0)；以及调用file:seek("end")将位置设置为文件的末尾，并返回其大小。

**file:setvbuf (mode [, size])**

设置输出文件的缓冲模式。有三种可用的模式：

* **“否”：**没有缓冲；任何输出操作的结果都会立即出现。
* **“全部”：**完全缓冲；只有当缓冲区已满时(或当您显式地执行时)，才会执行输出操作。flush文件(见[io.flush](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#pdf-io.flush))).
* **“行”：**行缓冲；输出被缓冲，直到换行符被输出，或者有来自某些特殊文件(例如终端设备)的任何输入。

在最后两起案件中，size指定缓冲区的大小(以字节为单位)。默认大小是适当的。

**file:write (···)**

将其每个参数的值写入file。参数必须是字符串或数字。若要编写其他值，请使用[tostring](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#pdf-tostring)或[string.format](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#pdf-string.format)以前write.

**5.8 -****操作系统设施**

这个库是通过表实现的。os.

**os.clock ()**

返回以秒为单位的程序所用CPU时间的近似值。

**os.date ([format [, time]])**

返回根据给定字符串格式化的包含日期和时间的字符串或表。format.

如果time参数存在，这是格式化的时间(请参阅[os.time](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#pdf-os.time)函数来描述此值)。否则，date格式化当前时间。

如果format以‘!‘，然后将日期格式化为协调的世界时间。在此可选字符之后，如果format是字符串“\*t“那么date返回具有以下字段的表：year(四位数)，month (1--12), day(1--31), hour (0--23), min (0--59), sec (0--61), wday(平日，星期日为1)，yday(年的日期)，及isdst(夏时制标志，布尔值)。

如果format不是“\*t“那么date将日期返回为字符串，按照与C函数相同的规则格式化strftime.

当没有争论的时候，date返回依赖于主机系统和当前区域设置的合理日期和时间表示(即，os.date()等于os.date("%c")).

**os.difftime (t2, t1)**

返回时间的秒数。t1到时间t2。在POSIX、Windows和其他一些系统中，这个值正是t2*-*t1.

**os.execute ([command])**

这个函数等价于C函数system。它过去了command由操作系统外壳执行。它返回一个状态代码，它依赖于系统.如果command不存在，则如果shell可用，则返回非零，否则返回零。

**os.exit ([code])**

调用C函数exit，有一个可选的code，以终止主机程序。的默认值code是成功的密码。

**os.getenv (varname)**

返回流程环境变量的值。varname，或**零**如果未定义变量。

**os.remove (filename)**

删除具有给定名称的文件或目录。目录必须为空才能被删除。如果此函数失败，则返回**零**，加上一个描述错误的字符串。

**os.rename (oldname, newname)**

重命名文件或目录oldname到newname。如果此函数失败，则返回**零**，加上一个描述错误的字符串。

**os.setlocale (locale [, category])**

设置程序的当前区域设置。locale是指定区域设置的字符串；category描述要更改哪个类别的可选字符串："all", "collate", "ctype", "monetary", "numeric"，或"time"；默认类别是"all"。函数返回新区域设置的名称，或**零**如果请求不能得到响应。

如果locale是空字符串，当前区域设置为实现定义的本机区域设置。如果locale是字符串“C“，当前区域设置为标准C语言环境。

当被调用时**零**作为第一个参数，此函数只返回给定类别的当前区域设置的名称。

**os.time ([table])**

返回不带参数的调用时的当前时间，或表示给定表指定的日期和时间的时间。这个表必须有字段year, month，和day，并且可能有字段hour, min, sec，和isdst(有关这些字段的说明，请参见[os.date](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#pdf-os.date)职能)。

返回的值是一个数字，其意义取决于您的系统。在POSIX、Windows和其他一些系统中，这个数字计算自某些给定的启动时间(“划时代”)以来的秒数。在其他系统中，没有指定含义，time只能用作date和difftime.

**os.tmpname ()**

返回一个具有文件名的字符串，该字符串可用于临时文件。文件必须在使用前显式打开，并在不再需要时显式删除。

在某些系统(POSIX)上，该函数还创建一个名为POSIX的文件，以避免安全风险。(其他人可能在获取名称到创建文件之间的时间内创建具有错误权限的文件。)您仍然必须打开文件才能使用它并删除它(即使您不使用它)。

如果可能，您可能更愿意使用[io.tmpfile](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#pdf-io.tmpfile)，它会在程序结束时自动删除文件。

**5.9 -****调试库**

此库为Lua程序提供调试接口的功能。在使用这个库时，你应该小心。这里提供的函数应该专门用于调试和类似的任务，例如分析。请抵制将它们用作常用编程工具的诱惑：它们可能非常慢。此外，这些函数中有几个违反了关于Lua代码的一些假设(例如，函数的局部变量不能从外部访问，或者用户数据元数据不能被Lua代码更改)，因此可能危及其他安全代码。

库中的所有函数都在debug桌子。在线程上操作的所有函数都有一个可选的第一个参数，即要操作的线程。默认值始终是当前线程。

**debug.debug ()**

与用户进入交互模式，运行用户输入的每个字符串。使用简单的命令和其他调试工具，用户可以检查全局变量和局部变量，更改它们的值，计算表达式，等等。只包含单词的一行cont完成此函数，以便调用者继续其执行。

请注意，用于debug.debug不是在任何函数中进行词汇嵌套，因此没有对局部变量的直接访问。

**debug.getfenv (o)**

Returns the environment of object o.

**debug.gethook ([thread])**

返回线程的当前钩子设置，作为三个值：当前钩子函数、当前挂钩掩码和当前钩子计数(由[debug.sethook](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#pdf-debug.sethook)职能)。

**debug.getinfo ([thread,] function [, what])**

返回包含函数信息的表。您可以直接给出函数，也可以给出一个数字作为function，这意味着函数在级别上运行。function给定线程的调用堆栈：级别0是当前函数(getinfo级别1是调用getinfo等等。如果function是一个大于活动函数数的数字，则getinfo回报**零**.

返回的表可以包含由[lua\_getinfo](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#lua_getinfo)，用绳子what描述要填写的字段。默认的what获取所有可用信息，但有效行表除外。如果有，选择‘f‘添加一个名为func功能本身。如果有，选择‘L‘添加一个名为activelines用有效行表。

例如，表达式debug.getinfo(1,"n").name如果可以找到合理的名称，则返回具有当前函数名称的表，并返回表达式。debug.getinfo(print)对象的所有可用信息返回一个表。[print](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#pdf-print)功能。

**debug.getlocal ([thread,] level, local)**

此函数返回带有索引的局部变量的名称和值。local职等的职能level堆里的东西。(第一个参数或局部变量具有索引1，等等，直到最后一个活动局部变量。)函数返回**零**如果给定索引没有局部变量，并且在使用level超出范围了。(你可以打电话[debug.getinfo](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#pdf-debug.getinfo)若要检查该级别是否有效，请执行以下操作。)

变量名以‘(‘(开括号)表示内部变量(循环控制变量、临时变量和C函数局部变量)。

**debug.getmetatable (object)**

返回给定的object或**零**如果它没有一个可元的。

**debug.getregistry ()**

返回注册表(请参阅[§3.5](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#3.5)).

**debug.getupvalue (func, up)**

此函数返回索引上值的名称和值。up职能func。函数返回**零**如果给定索引没有上值，则为。

**debug.setfenv (object, table)**

设置给定的object给予table。回报object.

**debug.sethook ([thread,] hook, mask [, count])**

将给定函数设置为钩子。弦mask以及数字count描述何时调用钩子。字符串掩码可以具有下列字符，并具有给定的含义：

* **"c":**每次Lua调用函数时都调用钩子；
* **"r":**每次Lua从函数返回时都会调用钩子；
* **"l":**每次Lua输入新代码行时都会调用钩子。

带着count与零不同，钩子是在每一次调用之后调用的。count指示。

当没有争论的时候，[debug.sethook](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#pdf-debug.sethook)关掉钩子。

调用钩子时，其第一个参数是一个字符串，描述触发其调用的事件："call", "return"(或"tail return"，当模拟尾部呼叫的返回时，"line"，和"count"。对于行事件，钩子还获取新的行号作为其第二个参数。在钩子里，你可以打电话给getinfo使用级别2获取有关运行函数的更多信息(级别0是getinfo函数，级别1是钩子函数)，除非事件是"tail return"。在这种情况下，lua只是模拟返回，并调用getinfo将返回无效数据。

**debug.setlocal ([thread,] level, local, value)**

此函数为value到带有索引的局部变量local职等的职能level堆里的东西。函数返回**零**如果给定索引没有局部变量，并且在使用level超出范围了。(你可以打电话getinfo若要检查该级别是否有效，请执行以下操作。)否则，它将返回局部变量的名称。

**debug.setmetatable (object, table)**

为给定的object给予table(这可以是**零**).

**debug.setupvalue (func, up, value)**

此函数为value到带索引的上值up职能func。函数返回**零**如果给定索引没有上值，则为。否则，它将返回上值的名称。

**debug.traceback ([thread,] [message] [, level])**

返回带有调用堆栈跟踪的字符串。任选message字符串附加在跟踪的开头。任选level数字指示在哪个级别启动跟踪(默认值为1，函数调用)。traceback).

**6 -****卢亚**

尽管Lua被设计为一种扩展语言，可以嵌入到主机C程序中，但它也经常被用作独立语言。作为独立语言的lua的解释器，称为lua，提供标准分布。独立解释器包括所有标准库，包括调试库.其用途如下：

lua [options] [script [args]]

备选方案如下：

* **-e *stat*:**执行字符串*统计*;
* **-l *mod*:**“要求”*国防部*;
* **-i:**运行后进入交互模式*剧本*;
* **-v:**打印版本信息；
* **--:**停止处理选项；
* **-:**执行stdin并停止处理选项。

在处理了它的选择之后，lua运行给定的*剧本*，传递给它*阿格斯*作为字符串参数。当没有争论的时候，lua表现为lua -v -i当标准输入(stdin)是终端，而lua -不然的话。

在运行任何参数之前，解释器将检查环境变量。LUA\_INIT。如果它的格式是@*filename*，然后lua执行文件。否则，lua执行字符串本身。

所有选项都按顺序处理，除-i。例如，类似于

$ lua -e'a=1' -e 'print(a)' script.lua

将第一组a到1，然后打印a(这是‘1)，并最终运行该文件。script.lua没有争论。(这里)$是shell提示符。你的提示符可能不一样。)

在开始运行脚本之前，lua收集全局表中命令行中的所有参数。arg。脚本名存储在索引0，脚本名称转到索引1之后的第一个参数，依此类推。脚本名称之前的任何参数(即解释器名称加上选项)都会转到负索引。例如，在调用中

$ lua -la b.lua t1 t2

解释器首先运行文件。a.lua，然后创建一个表。

arg = { [-2] = "lua", [-1] = "-la",

[0] = "b.lua",

[1] = "t1", [2] = "t2" }

并最终运行该文件。b.lua。调用该脚本的arg[1], arg[2]，··AS参数；它还可以使用vararg表达式访问这些参数...'.

在交互式模式下，如果您编写了不完整的语句，则解释器将通过发出不同的提示来等待它的完成。

如果全局变量\_PROMPT包含字符串，则其值用作提示符。类似地，如果全局变量\_PROMPT2包含字符串，其值用作辅助提示符(在不完整语句期间发出)。因此，两个提示都可以在命令行或任何lua程序中直接更改，方法是将\_PROMPT。参见下一个示例：

$ lua -e"\_PROMPT='myprompt> '" -i

(外对引号用于shell，内部对用于Lua。)注意使用-i若要进入交互模式，则程序将在分配到\_PROMPT.

为了允许在unix系统中使用lua作为脚本解释器，非重叠解释器跳过块的第一行(如果它以#。因此，可以使用chmod +x而#!表格，如

#!/usr/local/bin/lua

(当然，在您的机器中，Lua解释器的位置可能有所不同。如果lua在你的PATH，然后

#!/usr/bin/env lua

是一种更便携的解决方案。)

**7 -****与上一版本不兼容**

在这里，我们列出了在将程序从Lua5.0迁移到Lua5.1时可能发现的不兼容问题。您可以使用适当的选项来避免编译lua的大部分不兼容(请参阅文件)。luaconf.h)。然而，所有这些兼容性选项将在下一个版本的Lua中删除。

**7.1 -****语言的变化**

* vararg系统改变了伪参数。arg使用带有vararg表达式的额外参数的表。(见编译时选项)LUA\_COMPAT\_VARARG在……里面luaconf.h.)
* 的隐式变量的范围发生了微妙的变化。**为**声明和**重复**声明。
* 长字符串/长注释语法([[*string*]])不允许嵌套。您可以使用新的语法([=[*string*]=])在这些情况下。(见编译时选项)LUA\_COMPAT\_LSTR在……里面luaconf.h.)

**7.2 -****图书馆的变化**

* 功能string.gfind改名[string.gmatch](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#pdf-string.gmatch)。(见编译时选项)LUA\_COMPAT\_GFIND在……里面luaconf.h.)
* 什么时候[string.gsub](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#pdf-string.gsub)以函数作为第三个参数调用，只要该函数返回。**零**或**假的**替换字符串是整个匹配，而不是空字符串。
* 功能table.setn被反对了。功能table.getn对应于新的长度运算符(#)；使用操作符而不是函数。(见编译时选项)LUA\_COMPAT\_GETN在……里面luaconf.h.)
* 功能loadlib改名[package.loadlib](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#pdf-package.loadlib)。(见编译时选项)LUA\_COMPAT\_LOADLIB在……里面luaconf.h.)
* 功能math.mod改名[math.fmod](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#pdf-math.fmod)。(见编译时选项)LUA\_COMPAT\_MOD在……里面luaconf.h.)
* 功能table.foreach和table.foreachi不受欢迎。可以使用for循环pairs或ipairs相反。
* 职能发生了重大变化[require](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#pdf-require)由于新的模块系统。然而，新的行为大多与旧的行为兼容，但是require获取路径。[package.path](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#pdf-package.path)而不是从LUA\_PATH.
* 功能[collectgarbage](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#pdf-collectgarbage)有不同的论据。功能gcinfo被废弃；使用collectgarbage("count")相反。

**7.3 -****API中的更改**

* 这个luaopen\_\*函数(打开库)不能像普通C函数那样直接调用。它们必须通过Lua调用，就像Lua函数一样。
* 功能lua\_open代之以[lua\_newstate](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#lua_newstate)若要允许用户设置内存分配函数，请执行以下操作。你可以用[luaL\_newstate](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#luaL_newstate)从标准库中创建具有标准分配函数的状态(基于realloc).
* 功能luaL\_getn和luaL\_setn(从辅助库)被废弃。使用[lua\_objlen](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#lua_objlen)而不是luaL\_getn什么都没有，而是luaL\_setn.
* 功能luaL\_openlib代之以[luaL\_register](file:///E:\%E5%A4%87%E4%BB%BDmtk\mrp\%E6%97%A9%E6%9C%9F%E6%96%AF%E5%87%AFsdk\Mythroad%E8%AF%AD%E8%A8%80SDK\%E5%85%B6%E4%BB%96\manual.html#luaL_register).
* 功能luaL\_checkudata现在，当给定值不是预期类型的用户数据时，抛出一个错误。(在Lua5.0中它返回了NULL.)

**8 -****Lua的完全句法**

以下是扩展BNF中Lua的完整语法。(它不描述运算符的先例。)

chunk ::= {stat [`**;**´]} [laststat [`**;**´]]

block ::= chunk

stat ::= varlist `**=**´ explist |

functioncall |

**do** block **end** |

**while** exp **do** block **end** |

**repeat** block **until** exp |

**if** exp **then** block {**elseif** exp **then** block} [**else** block] **end** |

**for** Name `**=**´ exp `**,**´ exp [`**,**´ exp] **do** block **end** |

**for** namelist **in** explist **do** block **end** |

**function** funcname funcbody |

**local** **function** Name funcbody |

**local** namelist [`**=**´ explist]

laststat ::= **return** [explist] | **break**

funcname ::= Name {`**.**´ Name} [`**:**´ Name]

varlist ::= var {`**,**´ var}

var ::= Name | prefixexp `**[**´ exp `**]**´ | prefixexp `**.**´ Name

namelist ::= Name {`**,**´ Name}

explist ::= {exp `**,**´} exp

exp ::= **nil** | **false** | **true** | Number | String | `**...**´ | function |

prefixexp | tableconstructor | exp binop exp | unop exp

prefixexp ::= var | functioncall | `**(**´ exp `**)**´

functioncall ::= prefixexp args | prefixexp `**:**´ Name args

args ::= `**(**´ [explist] `**)**´ | tableconstructor | String

function ::= **function** funcbody

funcbody ::= `**(**´ [parlist] `**)**´ block **end**

parlist ::= namelist [`**,**´ `**...**´] | `**...**´

tableconstructor ::= `**{**´ [fieldlist] `**}**´

fieldlist ::= field {fieldsep field} [fieldsep]

field ::= `**[**´ exp `**]**´ `**=**´ exp | Name `**=**´ exp | exp

fieldsep ::= `**,**´ | `**;**´

binop ::= `**+**´ | `**-**´ | `**\***´ | `**/**´ | `**^**´ | `**%**´ | `**..**´ |

`**<**´ | `**<=**´ | `**>**´ | `**>=**´ | `**==**´ | `**~=**´ |

**and** | **or**

unop ::= `**-**´ | **not** | `**#**´

最新更新：2008年8月18日13：25：46