Lua 5.1 参考手册

by Roberto Ierusalimschy, Luiz Henrique de Figueiredo, Waldemar Celes

云风 译 www.codingnow.com

1 - 介绍

Lua 是一个扩展式程序设计语言，它被设计成支持通用的过程式编程，并有相关数据描述的设施。 Lua 也能对面向对象编程，函数式编程，数据驱动式编程提供很好的支持。 它可以作为一个强大、轻量的脚本语言，供任何需要的程序使用。 Lua 以一个用 *clean* C 写成的库形式提供。（所谓 Clean C ，指的 ANSI C 和 C++ 中共通的一个子集）

作为一个扩展式语言，Lua 没有 "main" 程序的概念：它只能 *嵌入* 一个宿主程序中工作，这个宿主程序被称作 *embedding program* 或简称为 *host* 。 宿主程序可以通过调用函数执行一小段 Lua 代码，可以读写 Lua 变量，可以注入 C 函数让 Lua 代码调用。 这些扩展的 C 函数，可以大大的扩展了 Lua 可以处理事务的领域，这样就可以订制出各种语言， 而它们共享一个统一的句法格式的框架。 Lua 的官方发布版就包含了一个叫做 `lua` 的简单的宿主程序，它用 Lua 库提供了一个保证独立的 Lua 解释器。

Lua 是一个自由软件，它的使用许可决定了对它的使用过程一般没有任何保证。 这份手册中描述的东西的实现，可以在 Lua 的官方网站 `www.lua.org` 找到，

跟其它的许多参考手册一样，这份文档有些地方比较枯燥。 关于 Lua 的设计想法的探讨，可以看看 Lua 网站上提供的技术论文。 有关用 Lua 编程的细节介绍，可以读一下 Roberto 的书，*Programming in Lua (Second Edition)* 。

2 - 语言

这一节从词法、语法、句法上描述 Lua 。 换句话说，这一节描述了哪些 token （符记）是有效的，它们如何被组合起来，这些组合方式有什么含义。

关于语言的构成概念将用常见的扩展 BNF 表达式写出。也就是这个样子： {*a*} 意思是 0 或多个 *a* ， [*a*] 意思是一个可选的 *a* 。 非最终的符号会保留原来的样子，关键字则看起来像这样 **kword** ，其它最终的符号则写成 `=´ 。 完整的 Lua 语法可以在本手册最后找到。

2.1 - 词法约定

Lua 中用到的 *名字*（也称作 *标识符*）可以是任何非数字开头的字母、数字、下划线组成的字符串。 这符合几乎所有编程语言中关于名字的定义。 （字母的定义依赖于当前环境：系统环境中定义的字母表中的字母都可以被用于标识符。） 标识符用来命名变量，或作为表的域名。

下面的关键字是保留的，不能用作名字：

```
and       break     do        else      elseif
end       false     for       function  if
in        local     nil       not       or
repeat    return    then      true      until     while
```

Lua 是一个大小写敏感的语言： `and` 是一个保留字，但是 `And` 和 `AND` 则是两个不同的合法的名字。 一般约定，以下划线开头连接一串大写字母的名字（比如 `_VERSION`）被保留用于 Lua 内部全局变量。

下面这些是其它的 token：

```
+     -     *     /     %     ^     #
==    ~=    <=    >=    <     >     =
(     )     {     }     [     ]
;     :     ,     .     ..    ...
```

字符串既可以用一对单引号引起，也可以是双引号，里面还可以包含类似 C 的转义符： `'\a'`（响铃）， `'\b'`（退格）， `'\f'`（表单）， `'\n'`（换行）， `'\r'`（回车）， `'\t'`（横向制表）， `'\v'`（纵向制表）， `'\\'`（反斜杠）， `'\"'`（双引号）， 以及 `'\''`（单引号)。 而且，如果在一个反斜杠后跟了一个真正的换行符，其结果就是在字符串中产生一个换行符。 我们还可以用反斜杠加数字的形式 `\`*ddd* 来描述一个字符。这里， *ddd* 是一串最多三位的十进制数字。（注意，如果需要在这种描述方法后接一个是数字的字符， 那么反斜杠后必须写满三个数字。）Lua 中的字符串可以包含任何 8 位的值。包括用 `'\0'` 表示的零。

只有在你需要把不同的引号、换行、反斜杠、或是零结束符这些字符置入字符串时， 你才必须使用转义符。别的任何字符都可以直接写在文本里。（一些控制符可以会影响文件系统造成某些问题， 但是不会引起 Lua 的任何问题。）

字符串还可以用一种长括号括起来的方式定义。 我们把两个正的方括号间插入 n 个等号定义为第 n 级正长括号。 就是说，0 级正的长括号写作 `[[` ， 一级正的长括号写作 `[=[` ，如此等等。 反的长扩展也作类似定义； 举个例子，4 级反的长括号写作 `]====]` 。 一个长字符串可以由任何一级的正的长括号开始，而由第一个碰到的同级反的长括号结束。 整个词法分析过程将不受分行限制，不处理任何转意符，并且忽略掉任何不同级别的长括号。 这种方式描述的字符串可以包含任何东西，当然特定级别的反长括号除外。

另一个约定是，当正的长括号后面立即跟了一个换行符， 这个换行符就不包含在这个字符串内。举个例子，假设一个系统使用 ASCII 码 （这时，`'a'` 编码为 97 ， 换行符编码为 10 ， `'1'` 编码为 49 ）， 下面五种方式描述了完全相同的字符串：

```
a = 'alo\n123"'
a = "alo\n123\""
a = '\97lo\10\04923"'
a = [[alo
123"]]
a = [==[
alo
123"]==]
```

数字常量可以分两部分写，十进制底数部分和十进制的指数部分。指数部分是可选的。 Lua 也支持十六进制整数常量，只需要在前面加上前缀 `0x` 。 下面是一些合法的数字常量的例子：

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

注释可以在除字符串内的任何地方是以两横 (--) 开始。 如果跟在两横后面的不是一个长括号，这就是一个短注释，它的作用范围直到行末； 否则就是一个长注释，其作用范围直到遇到反的长括号。 长注释通常被用来临时屏蔽代码块。

2.2 - 值与类型

Lua 是一种 *动态类型语言*。 这意味着变量没有类型，只有值才有类型。 语言中不存在类型定义。 而所有的值本身携带它们自己的类型信息。

Lua 中的所有值都是一致 (first-class) 的。 这意味着所有的值都可以被放在变量里，当作参数传递到另一个函数中，并被函数作为结果返回。

Lua 中有八种基本类型： *nil*, *boolean*, *number*, *string*, *function*, *userdata*, *thread*, and *table*. *Nil* 类型只有一种值 **nil** ，它的主要用途用于标表识和别的任何值的差异； 通常，当需要描述一个无意义的值时会用到它。 *Boolean* 类型只有两种值： **false** 和 **true**。 **nil** 和 **false** 都能导致条件为假；而另外所有的值都被当作真。 *Number* 表示实数（双精度浮点数）。 （编译一个其它内部数字类型的 Lua 解释器是件很容易的事；比如把内部数字类型改作 单精度浮点数或长整型。参见文件 `luaconf.h` 。） *String* 表示一串字符的数组。 Lua 是 8-bit clean 的： 字符串可以包含任何 8 位字符， 包括零结束符 (`'\0'`) （参见 §2.1）。

Lua 可以调用（和处理）用 Lua 写的函数以及用 C 写的函数（参见 §2.5.8）.

userdata 类型用来将任意 C 数据保存在 Lua 变量中。 这个类型相当于一块原生的内存，除了赋值和相同性判断，Lua 没有为之预定义任何操作。 然而，通过使用 *metatable* （*元表*），程序员可以为 userdata 自定义一组操作 （参见 §2.8）。 userdata 不能在 Lua 中创建出来，也不能在 Lua 中修改。这样的操作只能通过 C API。 这一点保证了宿主程序完全掌管其中的数据。

thread 类型用来区别独立的执行线程，它被用来实现 coroutine （协同例程）（参见 §2.11）。 不要把 Lua 线程跟操作系统的线程搞混。 Lua 可以在所有的系统上提供对 coroutine 的支持，即使系统并不支持线程。

table 类型实现了一个关联数组。也就是说， 数组可以用任何东西（除了**nil**）做索引，而不限于数

字。 table 可以以不同类型的值构成；它可以包含所有的类型的值（除 **nil** 外）。 table 是 lua 中唯一的一种数据结构；它可以用来描述原始的数组、符号表、集合、 记录、图、树、等等。 用于表述记录时，lua 使用域名作为索引。 语言本身采用一种语法糖，支持以 `a.name` 的形式表示 `a["name"]`。 有很多形式用于在 lua 中创建一个 table （参见 §2.5.7）。

跟索引一样， table 每个域中的值也可以是任何类型（除 **nil**外）。 特别的，因为函数本身也是值，所以 table 的域中也可以放函数。 这样 table 中就可以有一些 *methods* 了 （参见see §2.5.9）。

table， function ， thread ， 和 (full) userdata 这些类型的值是所谓的对象： 变量本身并不会真正的存放它们的值，而只是放了一个对对象的引用。 赋值，参数传递，函数返回，都是对这些对象的引用进行操作； 这些操作不会做暗地里做任何性质的拷贝。

库函数 `type` 可以返回一个描述给定值的类型的字符串。

2.2.1 - 强制转换

Lua 提供运行时字符串到数字的自动转换。 任何对字符串的数学运算操作都会尝试用一般的转换规则把这个字符串转换成一个数字。 相反，无论何时，一个数字需要作为字符串来使用时，数字都会以合理的格式转换为字符串。 需要完全控制数字怎样转换为字符串，可以使用字符串库中的 `format` 函数 （参见 `string.format`）。

2.3 - 变量

写上变量的地方意味着当以其保存的值来替代之。 Lua 中有三类变量：全局变量，局部变量，还有 table 的域。

一个单一的名字可以表示一个全局变量，也可以表示一个局部变量 （或者是一个函数的参数，这是一种特殊形式的局部变量）：

```
var ::= Name
```

Name 就是 §2.1 中所定义的标识符。

任何变量都被假定为全局变量，除非显式的以 local 修饰定义 （参见 §2.4.7）。 局部变量有其作用范围： 局部变量可以被定义在它作用范围中的函数自由使用 （参见 §2.6）。

在变量的首次赋值之前，变量的值均为 **nil**。

方括号被用来对 table 作索引：

```
var ::= prefixexp `[´ exp `]´
```

对全局变量以及 table 域之访问的含义可以通过 metatable 来改变。 以取一个变量下标指向的量 `t[i]` 等价于调用 `gettable_event(t,i)`。 （参见 §2.8 ，有一份完整的关于 `gettable_event` 函数的说明。 这个函数并没有在 lua 中定义出来，也不能在 lua 中调用。 这里我们把它列出来只是方便

说明。）

`var.Name` 这种语法只是一个语法糖，用来表示 `var["Name"]`：

```
var ::= prefixexp `.´ Name
```

所有的全局变量都是放在一个特定 lua table 的诸个域中，这个特定的 table 叫作 *environment （环境）table* 或者简称为 *环境* （参见 §2.9）。 每个函数都有对一个环境的引用， 所以一个函数中可见的所有全局变量都放在这个函数所引用的环境表（environment table）中。 当一个函数被创建出来，它会从创建它的函数中继承其环境，你可以调用 `getfenv` 取得其环境。 如果想改变环境，可以调用 `setfenv`。 （对于 C 函数，你只能通过 debug 库来改变其环境； 参见 §5.9）。

对一个全局变量 `x` 的访问 等价于 `_env.x`，而这又可以等价于

```
gettable_event(_env, "x")
```

这里，`_env` 是当前运行的函数的环境。 （函数 `gettable_event` 的完整说明参见 §2.8。 这个函数并没有在 lua 中定义出来，也不能调用。 当然，`_env` 这个变量也同样没有在 Lua 中定义出来。 我们在这里使用它们，仅仅只是方便解释而已。）

2.4 - 语句段（Statement）

Lua 支持惯例形式的语句段，它和 Pascal 或是 C 很相象。 这个集合包括赋值，控制结构，函数调用，还有变量声明。

2.4.1 - Chunk（语句组）

Lua 的一个执行单元被称作 *chunk*。 一个 chunk 就是一串语句段，它们会被循序的执行。 每个语句段可以以一个分号结束：

```
chunk ::= {stat [`;´]}
```

这儿不允许有空的语句段，所以 '`;;`' 是非法的。

lua 把一个 chunk 当作一个拥有不定参数的匿名函数 （参见 §2.5.9）处理。 正是这样，chunk 内可以定义局部变量，接收参数，并且返回值。

chunk 可以被保存在一个文件中，也可以保存在宿主程序的一个字符串中。 当一个 chunk 被执行，首先它会被预编译成虚拟机中的指令序列， 然后被虚拟机解释运行这些指令。

chunk 也可以被预编译成二进制形式；细节参考程序 `luac`。 用源码形式提供的程序和被编译过的二进制形式的程序是可以相互替换的； Lua 会自动识别文件类型并做正确的处理。

2.4.2 - 语句块

语句块是一列语句段；从语法上来说，一个语句块跟一个 chunk 相同：

```
block ::= chunk
```

一个语句块可以被显式的写成一个单独的语句段：

```
stat ::= do block end
```

显式的语句块对于控制变量的作用范围很有用。 有时候，显式的语句块被用来在另一个语句块中插入 **return** 或是 **break** （参见 §2.4.4）。

2.4.3 - 赋值

Lua 允许多重赋值。 因此，赋值的语法定义是等号左边放一系列变量， 而等号右边放一系列的表达式。 两边的元素都用逗号间开：

```
stat ::= varlist1 `=´ explist1
varlist1 ::= var {`,´ var}
explist1 ::= exp {`,´ exp}
```

表达式放在 §2.5 里讨论。

在作赋值操作之前， 那一系列的右值会被对齐到左边变量需要的个数。 如果右值比需要的更多的话，多余的值就被扔掉。 如果右值的数量不够需求， 将会按所需扩展若干个 **nil**。 如果表达式列表以一个函数调用结束， 这个函数所返回的所有值都会在对齐操作之前被置入右值序列中。 （除非这个函数调用被用括号括了起来；参见 §2.5）。

赋值段首先会做运算完所有的表达式，然后仅仅做赋值操作。 因此，下面这段代码

```
i = 3
i, a[i] = i+1, 20
```

会把 `a[3]` 设置为 20，而不会影响到 `a[4]`。 这是因为 `a[i]` 中的 `i` 在被赋值为 4 之前就被拿出来了（那时是 3 ）。 简单说 ， 这样一行

```
x, y = y, x
```

可以用来交换 `x` 和 `y` 中的值。

对全局变量以及 table 中的域的赋值操作的含义可以通过 metatable 来改变。 对变量下标指向的赋值，即 `t[i] = val` 等价于 `settable_event(t,i,val)`。 （关于函数 `settable_event` 的详细说明，参见 §2.8。 这个函数并没有在 Lua 中定义出来，也不可以被调用。 这里我们列出来，仅仅出于方便解释的目的）

对于全局变量的赋值 `x = val` 等价于 `_env.x = val`，这个又可以等价于

```
settable_event(_env, "x", val)
```

这里，`_env` 指的是正在运行中的函数的环境。 （变量 `_env` 并没有在 Lua 中定义出来。 我们仅仅

出于解释的目的在这里写出来。）

2.4.4 - 控制结构

if、 **while**、以及 **repeat** 这些控制结构符合通常的意义，而且也有类似的语法：

```
stat ::= while exp do block end
stat ::= repeat block until exp
stat ::= if exp then block {elseif exp then block} [else block] end
```

Lua 也有一个 **for** 语句，它有两种形式（参见 §2.4.5）。

控制结构中的条件表达式可以返回任何值。 **false** 和 **nil** 两者都被认为是假条件。 所有不同于 **nil** 和 **false** 的其它值都被认为是真 （特别需要注意的是，数字 0 和空字符串也被认为是真）。

在 **repeat–until** 循环中， 内部语句块的结束点不是在 **until** 这个关键字处， 它还包括了其后的条件表达式。 因此，条件表达式中可以使用循环内部语句块中的定义的局部变量。

return 被用于从函数或是 chunk（其实它就是一个函数）中 返回值。 函数和 chunk 可以返回不只一个值， 所以 **return** 的语法为

```
stat ::= return [explist1]
```

break 被用来结束 **while**、 **repeat**、或 **for** 循环， 它将忽略掉循环中下面的语句段的运行：

```
stat ::= break
```

break 跳出最内层的循环。

return 和 **break** 只能被写在一个语句块的最后一句。 如果你真的需要从语句块的中间 **return** 或是 **break** ， 你可以使用显式的声名一个内部语句块。 一般写作 `do return end` 或是 `do break end`，可以这样写是因为现在 **return** 或 **break** 都成了一个语句块的最后一句了。

2.4.5 - For 语句

for 有两种形式：一种是数字形式，另一种是一般形式。

数字形式的 **for** 循环，通过一个数学运算不断的运行内部的代码块。 下面是它的语法：

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

block 将把 *name* 作循环变量。从第一个 *exp* 开始起，直到第二个 *exp* 的值为止，其步长为 第三个 *exp* 。 更确切的说，一个 **for** 循环看起来是这个样子

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

这等价于代码：

```
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
```

注意下面这几点:

- 所有三个控制表达式都只被运算一次，表达式的计算在循环开始之前。 这些表达式的结果必须是数字。
- *var* 、 *limit* 、以及 *step* 都是一些不可见的变量。 这里给它们起的名字都仅仅用于解释方便。
- 如果第三个表达式（步长）没有给出，会把步长设为 1 。
- 你可以用 **break** 来退出 **for** 循环。
- 循环变量 v 是一个循环内部的局部变量； 当 **for** 循环结束后，你就不能在使用它。 如果你需要这个值，在退出循环前把它赋给另一个变量。

一般形式的 **for** 通过一个叫作迭代器（*iterators*）的函数工作。 每次迭代，迭代器函数都会被调用以产生一个新的值， 当这个值为 **nil** 时，循环停止。 一般形式的 **for** 循环的语法如下:

```
stat ::= for namelist in explist1 do block end
namelist ::= Name {`,´ Name}
```

for 语句好似这样

```
for var_1, ···, var_n in explist do block end
```

它等价于这样一段代码:

```
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
```

注意以下几点:

- *explist* 只会被计算一次。 它返回三个值， 一个迭代器函数，一个状态，一个迭代器的初始值。
- *f*、 *s*、 以及 *var* 都是不可见的变量。 这里给它们起的名字都只是为了解说方便。
- 你可以使用 **break** 来跳出 **for** 循环。
- 循环变量 *var_i* 对于循环来说是一个局部变量； 你不可以在 **for** 循环结束后继续使用。 如果你需要保留这些值，那么就在循环结束前赋值到别的变量里去。

2.4.6 - 把函数调用作为语句段

为了允许使用可能的副作用， 函数调用可以被作为一个语句段执行：

```
stat ::= functioncall
```

在这种情况下，所有的返回值都被舍弃。 函数调用在 §2.5.8 中解释。

2.4.7 - 局部变量声名

局部变量可以在语句块中任何地方声名。 声名可以包含一个初始化赋值操作：

```
stat ::= local namelist [`=´ explist1]
```

如果有的话，初始化赋值操作的行为等同于赋值操作（参见 §2.4.3）。 否则，所有的变量将被初始化为 **nil**。

一个 chunk 同时也是一个语句块（参见 §2.4.1）， 所以局部变量可以放在 chunk 中那些显式注明的语句块之外。 这些局部变量的作用范围从声明起一直延伸到 chunk 末尾。

局部变量的可见规则在 §2.6 中解释。

2.5 - 表达式

Lua 中有这些基本表达式：

```
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 `)´
```

数字和字符串在 §2.1 中解释； 变量在 §2.3 中解释； 函数定义在 §2.5.9 中解释； 函数调用在 §2.5.8 中解释； table 的构造在 §2.5.7 中解释； 可变参数的表达式写作三个点 ('...')，它只能被用在有可变参数的函数中； 这些在 §2.5.9 中解释。

二元操作符包含有数学运算操作符（参见 §2.5.1）， 比较操作符（参见 §2.5.2），逻辑操作符（参见 §2.5.3）， 以及连接操作符（参见 §2.5.4）。 一元操作符包括负号（参见see §2.5.1）， 取反 **not**（参见 §2.5.3）， 和取长度操作符（参见 §2.5.5）。

函数调用和可变参数表达式都可以放在多重返回值中。 如果表达式作为一个独立语句段出现（参见 §2.4.6）（这只能是一个函数调用）， 它们的返回列表将被对齐到零个元素，也就是忽略所有返回值。 如果表达式用于表达式列表的最后（或者是唯一）的元素， 就不会有任何的对齐操作

（除非函数调用用括号括起来）。 在任何其它的情况下，Lua 将把表达式结果看成单一元素， 忽略除第一个之外的任何值。

这里有一些例子:

```
f()                 -- 调整到 0 个结果
g(f(), x)           -- f() 被调整到一个结果
g(x, f())           -- g 被传入 x 加上所有 f() 的返回值
a,b,c = f(), x      -- f() 被调整到一个结果 （ c 在这里被赋为 nil ）
a,b = ...           -- a 被赋值为可变参数中的第一个,
                    -- b 被赋值为第二个 （如果可变参数中并没有对应的值,
                                          -- 这里 a 和 b 都有可能被赋为 nil）

a,b,c = x, f()      -- f() 被调整为两个结果
a,b,c = f()         -- f() 被调整为三个结果
return f()          -- 返回 f() 返回的所有结果
return ...          -- 返回所有从可变参数中接收来的值
return x,y,f()      -- 返回 x, y, 以及所有 f() 的返回值
{f()}               -- 用 f() 的所有返回值创建一个列表
{...}               -- 用可变参数中的所有值创建一个列表
{f(), nil}          -- f() 被调整为一个结果
```

被括号括起来的表达式永远被当作一个值。所以， `(f(x,y,z))` 即使 `f` 返回多个值，这个表达式永远是一个单一值。 （`(f(x,y,z))` 的值是 `f` 返回的第一个值。如果 `f` 不返回值的话，那么它的值就是 **nil** 。）

2.5.1 - 数学运算操作符

Lua 支持常见的数学运算操作符： 二元操作 `+` （加法）， `-` （减法），`*` （乘法）， `/` （除法）， `%` （取模），以及 `^` （幂）； 和一元操作 `-` （取负）。 如果对数字操作，或是可以转换为数字的字符串（参见 §2.2.1）， 所有这些操作都依赖它通常的含义。 幂操作可以对任何幂值都正常工作。比如， `x^(-0.5)` 将计算出 `x` 平方根的倒数。 取模操作被定义为

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

这就是说，其结果是商相对负无穷圆整后的余数。（译注：负数对正数取模的结果为正数）

2.5.2 - 比较操作符

Lua 中的比较操作符有

```
==    ~=    <     >     <=    >=
```

这些操作的结果不是 **false** 就是 **true**。

等于操作 (==) 首先比较操作数的类型。 如果类型不同，结果就是 **false**。 否则，继续比较值。 数字和字符串都用常规的方式比较。 对象 （table ， userdata ， thread ， 以及函数）以引用的形式比

较： 两个对象只有在它们指向同一个东西时才认为相等。 每次你创建一个新对象（一个 table 或是 userdata ， thread 函数）， 它们都各不相同，即不同于上次创建的东西。

你可以改变 Lua 比较 table 和 userdata 的方式，这需要使用 "eq" 这个原方法 （参见 §2.8）。

§2.2.1 中提及的转换规则并不作用于比较操作。 所以， `"0"==0` 等于 **false**， 而且 `t[0]` 和 `t["0"]` 描述的是 table 中不同的域。

操作符 ~= 完全等价于 (==) 操作的反值。

大小比较操作以以下方式进行。 如果参数都是数字，那么就直接做数字比较。 否则，如果参数都是字符串，就用字符串比较的方式进行。 再则，Lua 就试着调用 "lt" 或是 "le" 元方法 （参见 §2.8）。

2.5.3 - 逻辑操作符

Lua 中的逻辑操作符有 **and**, **or**, 以及 **not**。 和控制结构（参见 §2.4.4）一样， 所有的逻辑操作符把 **false** 和 **nil** 都作为假， 而其它的一切都当作真。

取反操作 **not** 总是返回 **false** 或 **true** 中的一个。 与操作符 **and** 在第一个参数为 **false** 或 **nil** 时 返回这第一个参数； 否则，**and** 返回第二个参数。 或操作符 **or** 在第一个参数不为 **nil** 也不为 **false** 时， 返回这第一个参数，否则返回第二个参数。 **and** 和 **or** 都遵循短路规则； 也就是说，第二个操作数只在需要的时候去求值。 这里有一些例子：

```
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
```

（在这本手册中， --> 指前面表达式的结果。）

2.5.4 - 连接符

Lua 中字符串的连接操作符写作两个点 ('`..`')。 如果两个操作数都是字符串或都是数字，连接操作将以 §2.2.1 中提到的规则把其转换为字符串。 否则，会取调用元方法 "concat" （参见 §2.8）。

2.5.5 - 取长度操作符

取长度操作符写作一元操作 `#`。 字符串的长度是它的字节数（就是以一个字符一个字节计算的字符串长度）。

table `t` 的长度被定义成一个整数下标 `n` 。 它满足 `t[n]` 不是 **nil** 而 `t[n+1]` 为 **nil**； 此外，如果 `t[1]`

为 **nil** ，n 就可能是零。 对于常规的数组，里面从 1 到 n 放着一些非空的值的时候， 它的长度就精确的为 n，即最后一个值的下标。 如果数组有一个“空洞” （就是说，**nil** 值被夹在非空值之间）， 那么 #t 可能是指向任何一个是 **nil** 值的前一个位置的下标 （就是说，任何一个 **nil** 值都有可能被当成数组的结束）。

2.5.6 - 优先级

Lua 中操作符的优先级写在下表中，从低到高优先级排序:

```
or
and
<     >     <=    >=    ~=    ==
..
+     -
*     /     %
not   #     - (unary)
^
```

通常，你可以用括号来改变运算次序。 连接操作符 ('..') 和幂操作 ('^') 是从右至左的。 其它所有的操作都是从左至右。

2.5.7 - Table 构造

table 构造子是一个构造 table 的表达式。 每次构造子被执行，都会构造出一个新的 table 。 构造子可以被用来构造一个空的 table， 也可以用来构造一个 table 并初始化其中的一些域。 一般的构造子的语法如下

```
tableconstructor ::= `{´ [fieldlist] `}´
fieldlist ::= field {fieldsep field} [fieldsep]
field ::= `[´ exp `]´ `=´ exp | Name `=´ exp | exp
fieldsep ::= `,´ | `;´
```

每个形如 [exp1] = exp2 的域向 table 中增加新的一项， 其键值为 exp1 而值为 exp2。 形如 name = exp 的域等价于 ["name"] = exp。 最后，形如 exp 的域等价于 [i] = exp ， 这里的 i 是一个从 1 开始不断增长的数字。 这这个格式中的其它域不会破坏其记数。 举个例子:

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

等价于

```
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
```

如果表单中最后一个域的形式是 `exp`， 而且其表达式是一个函数调用或者是一个可变参数， 那么这个表达式所有的返回值将连续的进入列表 （参见 §2.5.8）。 为了避免这一点，你可以用括号把函数调用（或是可变参数）括起来 （参见 §2.5）。

初始化域表可以在最后多一个分割符， 这样设计可以方便由机器生成代码。

2.5.8 - 函数调用

Lua 中的函数调用的语法如下:

```
functioncall ::= prefixexp args
```

函数调用时，第一步，prefixexp 和 args 先被求值。 如果 prefixexp 的值的类型是 *function*， 那么这个函数就被用给出的参数调用。 否则 prefixexp 的元方法 "call" 就被调用， 第一个参数就是 prefixexp 的值，跟下来的是原来的调用参数 （参见 §2.8）。

这样的形式

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

可以用来调用 "方法"。 这是 Lua 支持的一种语法糖。像 `v:name(`*`args`*`)` 这个样子，被解释成 `v.name(v,`*`args`*`)`， 这里 `v` 只会被求值一次。

参数的语法如下:

```
args ::= `(´ [explist1] `)´
args ::= tableconstructor
args ::= String
```

所有参数的表达式求值都在函数调用之前。 这样的调用形式 `f{`*`fields`*`}` 是一种语法糖用于表示 `f({`*`fields`*`})`； 这里指参数列表是一个单一的新创建出来的列表。 而这样的形式 `f'`*`string`*`'` （或是 `f"`*`string`*`"` 亦或是 `f[[`*`string`*`]]`） 也是一种语法糖，用于表示 `f('`*`string`*`')`； 这里指参数列表是一个单独的字符串。

因为表达式语法在 Lua 中比较自由， 所以你不能在函数调用的 '`(`' 前换行。 这个限制可以避免语言中的一些歧义。 比如你这样写

```
a = f
(g).x(a)
```

Lua 将把它当作一个单一语句段， `a = f(g).x(a)` 。 因此，如果你真的想作为成两个语句段，你必须在它们之间写上一个分号。 如果你真的想调用 `f`， 你必须从 `(g)` 前移去换行。

这样一种调用形式: `return` *functioncall* 将触发一个尾调用。 Lua 实现了适当的尾部调用（或是适当的尾递归）： 在尾调用中， 被调用的函数重用调用它的函数的堆栈项。 因此，对于程序执行的嵌套尾调用的层数是没有限制的。 然而，尾调用将删除调用它的函数的任何调试信息。 注意，尾调用只发生在特定的语法下， 这时， **return** 只有单一函数调用作为参数； 这种语法使得调用

函数的结果可以精确返回。 因此，下面这些例子都不是尾调用：

```
return (f(x))        -- 返回值被调整为一个
return 2 * f(x)
return x, f(x)       -- 最加若干返回值
f(x); return         -- 无返回值
return x or f(x)     -- 返回值被调整为一个
```

2.5.9 - 函数定义

函数定义的语法如下：

```
function ::= function funcbody
funcbody ::= `(´ [parlist1] `)´ block end
```

另外定义了一些语法糖简化函数定义的写法：

```
stat ::= function funcname funcbody
stat ::= local function Name funcbody
funcname ::= Name {`.´ Name} [`:´ Name]
```

这样的写法：

```
function f () body end
```

被转换成

```
f = function () body end
```

这样的写法：

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

被转换成

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

这样的写法：

```
local function f () body end
```

被转换成

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

注意，并不是转换成

```
local f = function () body end
```

（这个差别只在函数体内需要引用 `f` 时才有。）

一个函数定义是一个可执行的表达式， 执行结果是一个类型为 *function* 的值。 当 Lua 预编译一个

chunk 的时候， chunk 作为一个函数，整个函数体也就被预编译了。 那么，无论何时 Lua 执行了函数定义， 这个函数本身就被实例化了（或者说是关闭了）。 这个函数的实例（或者说是 *closure*（闭包）） 是表达式的最终值。 相同函数的不同实例有可能引用不同的外部局部变量，也可能拥有不同的环境表。

形参（函数定义需要的参数）是一些由实参（实际传入参数）的值初始化的局部变量:

```
parlist1 ::= namelist [`,´ `...´] | `...´
```

当一个函数被调用， 如果函数没有被定义为接收不定长参数，即在形参列表的末尾注明三个点 ('...')， 那么实参列表就会被调整到形参列表的长度， 变长参数函数不会调整实参列表； 取而代之的是，它将把所有额外的参数放在一起通过变长参数表达式传递给函数， 其写法依旧是三个点。 这个表达式的值是一串实参值的列表，看起来就跟一个可以返回多个结果的函数一样。 如果一个变长参数表达式放在另一个表达式中使用，或是放在另一串表达式的中间， 那么它的返回值就会被调整为单个值。 若这个表达式放在了一系列表达式的最后一个，就不会做调整了（除非用括号给括了起来）。

我们先做如下定义，然后再来看一个例子:

```
function f(a, b) end
function g(a, b, ...) end
function r() return 1,2,3 end
```

下面看看实参到形参数以及可变长参数的映射关系:

```
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
```

结果由 **return** 来返回（参见 §2.4.4）。 如果执行到函数末尾依旧没有遇到任何 **return** 语句， 函数就不会返回任何结果。

冒号语法可以用来定义方法， 就是说，函数可以有一个隐式的形参 `self`。 因此，如下写法:

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

是这样一种写法的语法糖:

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

2.6 - 可视规则

Lua 是一个有词法作用范围的语言。 变量的作用范围开始于声明它们之后的第一个语句段， 结束于包含这个声明的最内层语句块的结束点。 看下面这些例子:

```
x = 10                -- 全局变量
do                    -- 新的语句块
  local x = x         -- 新的一个 'x', 它的值现在是 10
  print(x)            --> 10
  x = x+1
  do                  -- 另一个语句块
    local x = x+1     -- 又一个 'x'
    print(x)          --> 12
  end
  print(x)            --> 11
end
print(x)              --> 10  (取到的是全局的那一个)
```

注意这里，类似 `local x = x` 这样的声明， 新的 `x` 正在被声明，但是还没有进入它的作用范围，所以第二个 `x` 指向的是外面一层的变量。

因为有这样一个词法作用范围的规则， 所以可以在函数内部自由的定义局部变量并使用它们。 当一个局部变量被更内层的函数中使用的时候， 它被内层函数称作 *upvalue*（上值），或是 *外部局部变量*。

注意，每次执行到一个 local 语句都会定义出一个新的局部变量。 看看这样一个例子:

```
a = {}
local x = 20
for i=1,10 do
  local y = 0
  a[i] = function () y=y+1; return x+y end
end
```

这个循环创建了十个 closure（这指十个匿名函数的实例）。 这些 closure 中的每一个都使用了不同的 `y` 变量， 而它们又共享了同一份 `x`。

2.7 - 错误处理

因为 Lua 是一个嵌入式的扩展语言， 所有的 Lua 动作都是从宿主程序的 C 代码调用 Lua 库 （参见 `lua_pcall`）中的一个函数开始的。 在 Lua 编译或运行的任何时候发生了错误，控制权都会交还给 C ， 而 C 可以来做一些恰当的措施（比如打印出一条错误信息）。

Lua 代码可以显式的调用 `error` 函数来产生一条错误。 如果你需要在 Lua 中捕获发生的错误， 你可以使用 `pcall` 函数。

2.8 - Metatable（元表）

Lua 中的每个值都可以用一个 *metatable*。 这个 *metatable* 就是一个原始的 Lua table ， 它用来定义

原始值在特定操作下的行为。 你可以通过在 metatable 中的特定域设一些值来改变拥有这个 metatable 的值 的指定操作之行为。 举例来说，当一个非数字的值作加法操作的时候， Lua 会检查它的 metatable 中 `"__add"` 域中的是否有一个函数。 如果有这么一个函数的话，Lua 调用这个函数来执行一次加法。

我们叫 metatable 中的键名为 *事件 (event)* ，把其中的值叫作 *元方法 (metamethod)*。 在上个例子中，事件是 `"add"` 而元方法就是那个执行加法操作的函数。

你可以通过 `getmetatable` 函数来查询到任何一个值的 metatable。

你可以通过 `setmetatable` 函数来替换掉 table 的 metatable 。 你不能从 Lua 中改变其它任何类型的值的 metatable （使用 debug 库例外）； 要这样做的话必须使用 C API 。

每个 table 和 userdata 拥有独立的 metatable （当然多个 table 和 userdata 可以共享一个相同的表作它们的 metatable）； 其它所有类型的值，每种类型都分别共享唯一的一个 metatable。 因此，所有的数字一起只有一个 metatable ，所有的字符串也是，等等。

一个 metatable 可以控制一个对象做数学运算操作、比较操作、连接操作、取长度操作、取下标操作时的行为， metatable 中还可以定义一个函数，让 userdata 作垃圾收集时调用它。 对于这些操作，Lua 都将其关联上一个被称作事件的指定健。 当 Lua 需要对一个值发起这些操作中的一个时， 它会去检查值中 metatable 中是否有对应事件。 如果有的话，键名对应的值（元方法）将控制 Lua 怎样做这个操作。

metatable 可以控制的操作已在下面列出来。 每个操作都用相应的名字区分。 每个操作的键名都是用操作名字加上两个下划线 '__' 前缀的字符串； 举例来说，"add" 操作的键名就是字符串 `"__add"`。 这些操作的语义用一个 Lua 函数来描述解释器如何执行更为恰当。

这里展示的用 Lua 写的代码仅作解说用； 实际的行为已经硬编码在解释器中，其执行效率要远高于这些模拟代码。 这些用于描述的的代码中用到的函数 （ `rawget` ， `tonumber` ，等等。） 都可以在 §5.1 中找到。 特别注意，我们使用这样一个表达式来从给定对象中提取元方法

```
metatable(obj)[event]
```

这个应该被解读作

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

这就是说，访问一个元方法不再会触发任何的元方法， 而且访问一个没有 metatable 的对象也不会失败（而只是简单返回 **nil**）。

- **"add":** + 操作。

  下面这个 `getbinhandler` 函数定义了 Lua 怎样选择一个处理器来作二元操作。 首先，Lua 尝试第一个操作数。 如果这个东西的类型没有定义这个操作的处理器，然后 Lua 会尝试第二个操作数。

```
function getbinhandler (op1, op2, event)
  return metatable(op1)[event] or metatable(op2)[event]
end
```

通过这个函数， `op1 + op2` 的行为就是

```
function add_event (op1, op2)
  local o1, o2 = tonumber(op1), tonumber(op2)
  if o1 and o2 then  -- 两个操作数都是数字?
    return o1 + o2   -- 这里的 '+' 是原生的 'add'
  else  -- 至少一个操作数不是数字时
    local h = getbinhandler(op1, op2, "__add")
    if h then
      -- 以两个操作数来调用处理器
      return h(op1, op2)
    else  -- 没有处理器：缺省行为
      error(···)
    end
  end
end
```

- **"sub":** `-` 操作。 其行为类似于 "add" 操作。
- **"mul":** `*` 操作。 其行为类似于 "add" 操作。
- **"div":** `/` 操作。 其行为类似于 "add" 操作。
- **"mod":** `%` 操作。 其行为类似于 "add" 操作， 它的原生操作是这样的 `o1 - floor(o1/o2)*o2`
- **"pow":** `^` （幂）操作。 其行为类似于 "add" 操作， 它的原生操作是调用 `pow` 函数（通过 C math 库）。
- **"unm":** 一元 `-` 操作。

```
function unm_event (op)
  local o = tonumber(op)
  if o then  -- 操作数是数字?
    return -o  -- 这里的 '-' 是一个原生的 'unm'
  else  -- 操作数不是数字。
    -- 尝试从操作数中得到处理器
    local h = metatable(op).__unm
    if h then
      -- 以操作数为参数调用处理器
      return h(op)
    else  -- 没有处理器：缺省行为
      error(···)
    end
  end
end
```

- **"concat":** `..` （连接）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  -- 原生字符串连接
  else
    local h = getbinhandler(op1, op2, "__concat")
    if h then
```

```
      return h(op1, op2)
    else
      error(···)
    end
  end
end
```

- **"len":** # 操作。

```
function len_event (op)
  if type(op) == "string" then
    return strlen(op)         -- 原生的取字符串长度
  elseif type(op) == "table" then
    return #op                -- 原生的取 table 长度
  else
    local h = metatable(op).__len
    if h then
      -- 调用操作数的处理器
      return h(op)
    else  -- 没有处理器：缺省行为
      error(···)
    end
  end
end
```

关于 table 的长度参见 §2.5.5 。

- **"eq":** == 操作。 函数 `getcomphandler` 定义了 Lua 怎样选择一个处理器来作比较操作。 元方法仅仅在参于比较的两个对象类型相同且有对应操作相同的元方法时才起效。

```
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" 事件按如下方式定义：

```
function eq_event (op1, op2)
  if type(op1) ~= type(op2) then  -- 不同的类型?
    return false   -- 不同的对象
  end
  if op1 == op2 then   -- 原生的相等比较结果?
    return true   -- 对象相等
  end
  -- 尝试使用元方法
  local h = getcomphandler(op1, op2, "__eq")
  if h then
    return h(op1, op2)
  else
    return false
  end
end
```

`a ~= b` 等价于 `not (a == b)` 。

- **"lt":** < 操作。

```
function lt_event (op1, op2)
  if type(op1) == "number" and type(op2) == "number" then
    return op1 < op2   -- 数字比较
  elseif type(op1) == "string" and type(op2) == "string" then
    return op1 < op2   -- 字符串按逐字符比较
  else
    local h = getcomphandler(op1, op2, "__lt")
    if h then
      return h(op1, op2)
    else
      error(···);
    end
  end
end
```

  `a > b` 等价于 `b < a`.

- **"le":** <= 操作。

```
function le_event (op1, op2)
  if type(op1) == "number" and type(op2) == "number" then
    return op1 <= op2   -- 数字比较
  elseif type(op1) == "string" and type(op2) == "string" then
    return op1 <= op2   -- 字符串按逐字符比较
  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` 等价于 `b <= a`。 注意，如果元方法 "le" 没有提供，Lua 就尝试 "lt" ， 它假定 `a <= b` 等价于 `not (b < a)` 。

- **"index":** 取下标操作用于访问 `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)      -- 调用处理器
  else return h[key]          -- 或是重复上述操作
  end
end
```

- **"newindex":** 赋值给指定下标 `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
    return h(table, key,value)    -- 调用处理器
  else h[key] = value             -- 或是重复上述操作
  end
end
```

- **"call":** 当 Lua 调用一个值时调用。

```
function function_event (func, ...)
  if type(func) == "function" then
    return func(...)   -- 原生的调用
  else
    local h = metatable(func).__call
    if h then
      return h(func, ...)
    else
      error(···)
    end
  end
end
```

2.9 - 环境

类型为 thread ， function ， 以及 userdata 的对象，除了 metatable 外还可以用另外一个与之关联的被称作 它们的环境的一个表， 像 metatable 一样，环境也是一个常规的 table ， 多个对象可以共享同一个环境。

userdata 的环境在 Lua 中没有意义。 这个东西只是为了在程序员想把一个表关联到一个 userdata 上时提供便利。

关联在线程上的环境被称作全局环境。 全局环境被用作它其中的线程以及线程创建的非嵌套函数

（通过 loadfile ， loadstring 或是 load ）的缺省环境。 而且它可以被 C 代码直接访问（参见 §3.3）。

关联在 C 函数上的环境可以直接被 C 代码访问（参见 §3.3）。 它们会作为这个 C 函数中创建的其它函数的缺省环境。

关联在 Lua 函数上的环境用来接管在函数内对全局变量（参见 §2.3）的所有访问。 它们也会作为这个函数内创建的其它函数的缺省环境。

你可以通过调用 setfenv 来改变一个 Lua 函数 或是正在运行中的线程的环境。 而想操控其它对象（userdata、C 函数、其它线程）的环境的话，就必须使用 C API 。

2.10 - 垃圾收集

Lua 提供了一个自动的内存管理。 这就是说你不需要关心创建新对象的分配内存操作，也不需要在这些对象不再需要时的主动释放内存。 Lua 通过运行一个垃圾收集器来自动管理内存，以此一遍又一遍的回收死掉的对象 （这是指 Lua 中不再访问的到的对象）占用的内存。 Lua 中所有对象都被自动管理，包括： table, userdata、 函数、线程、和字符串。

Lua 实现了一个增量标记清除的收集器。 它用两个数字来控制垃圾收集周期： *garbage-collector pause* 和 *garbage-collector step multiplier* 。

garbage-collector pause 控制了收集器在开始一个新的收集周期之前要等待多久。 随着数字的增大就导致收集器工作工作的不那么主动。 小于 1 的值意味着收集器在新的周期开始时不再等待。 当值为 2 的时候意味着在总使用内存数量达到原来的两倍时再开启新的周期。

step multiplier 控制了收集器相对内存分配的速度。 更大的数字将导致收集器工作的更主动的同时，也使每步收集的尺寸增加。 小于 1 的值会使收集器工作的非常慢，可能导致收集器永远都结束不了当前周期。 缺省值为 2 ，这意味着收集器将以内存分配器的两倍速运行。

你可以通过在 C 中调用 lua_gc 或是在 Lua 中调用 collectgarbage 来改变这些数字。 两者都接受百分比数值（因此传入参数 100 意味着实际值 1 ）。 通过这些函数，你也可以直接控制收集器（例如，停止或是重启）。

2.10.1 - 垃圾收集的元方法

使用 C API ， 你可以给 userdata （参见 §2.8）设置一个垃圾收集的元方法。 这个元方法也被称为结束子。 结束子允许你用额外的资源管理器和 Lua 的内存管理器协同工作 （比如关闭文件、网络连接、或是数据库连接，也可以说释放你自己的内存）。

一个 userdata 可被回收，若它的 metatable 中有 __gc 这个域 ， 垃圾收集器就不立即收回它。 取而代之的是，Lua 把它们放到一个列表中。 最收集结束后，Lua 针对列表中的每个 userdata 执行了下面这个函数的等价操作：

```
function gc_event (udata)
  local h = metatable(udata).__gc
  if h then
    h(udata)
  end
end
```

在每个垃圾收集周期的结尾，每个在当前周期被收集起来的 userdata 的结束子会以 它们构造时的逆序依次调用。 也就是说，收集列表中，最后一个在程序中被创建的 userdata 的 结束子会被第一个调用。

2.10.2 - Weak Table（弱表）

weak table 是一个这样的 table，它其中的元素都被弱引用。 弱引用将被垃圾收集器忽略掉， 换句话说， 如果对一个对象的引用只有弱引用， 垃圾收集器将回收这个对象。

weak table 的键和值都可以是 weak 的。 如果一个 table 只有键是 weak 的，那么将运行收集器回收它们的键， 但是会阻止回收器回收对应的值。 而一个 table 的键和值都是 weak 时，就即允许收集器回收键又允许收回值。 任何情况下，如果键和值中任一个被回收了，整个键值对就会从 table 中拿掉。 table 的 weak 特性可以通过在它的 metatable 中设置 `__mode` 域来改变。 如果 `__mode` 域中是一个包含有字符 'k' 的字符串时， table 的键就是 weak 的。 如果 `__mode` 域中是一个包含有字符 'v' 的字符串时， table 的值就是 weak 的。

在你把一个 table 当作一个 metatable 使用之后， 就不能再修改 `__mode` 域的值。 否则，受这个 metatable 控制的 table 的 weak 行为就成了未定义的。

2.11 - Coroutine （协同例程）

Lua 支持 coroutine ， 这个东西也被称为协同式多线程 (*collaborative multithreading*) 。 Lua 为每个 coroutine 提供一个独立的运行线路。 然而和多线程系统中的线程不同，coroutine 只在显式的调用了 yield 函数时才会挂起。

创建一个 coroutine 需要调用一次 `coroutine.create` 。 它只接收单个参数，这个参数是 coroutine 的主函数。 `create` 函数仅仅创建一个新的 coroutine 然后返回它的控制器 （一个类型为 *thread* 的对象）； 它并不会启动 coroutine 的运行。

当你第一次调用 `coroutine.resume` 时， 所需传入的第一个参数就是 `coroutine.create` 的返回值。 这时，coroutine 从主函数的第一行开始运行。 接下来传入 `coroutine.resume` 的参数将被传进 coroutine 的主函数。 在 coroutine 开始运行后，它讲运行到自身终止或是遇到一个 *yields* 。

coroutine 可以通过两种方式来终止运行： 一种是正常退出，指它的主函数返回（最后一条指令被运行后，无论有没有显式的返回指令）；另一种是非正常退出，它发生在未保护的错误发生的时候。 第一种情况中， `coroutine.resume` 返回 **true** ， 接下来会跟着 coroutine 主函数的一系列返回值。 第二种发生错误的情况下， `coroutine.resume` 返回 **false** ， 紧接着是一条错误信息。

coroutine 中切换出去，可以调用 coroutine.yield。 当 coroutine 切出，与之配合的 coroutine.resume 就立即返回， 甚至在 yield 发生在内层的函数调用中也可以（就是说， 这不限于发生在主函数中，也可以是主函数直接或间接调用的某个函数里）。 在 yield 的情况下，coroutine.resume 也是返回 **true**， 紧跟着那些被传入 coroutine.yield 的参数。 等到下次你在继续同样的 coroutine ，将从调用 yield 的断点处运行下去。 断点处 yield 的返回值将是 coroutine.resume 传入的参数。

类似 coroutine.create ， coroutine.wrap 这个函数也将创建一个 coroutine ， 但是它并不返回 coroutine 本身，而是返回一个函数取而代之。一旦你调用这个返回函数，就会切入 coroutine 运行。 所有传入这个函数的参数等同于传入 coroutine.resume 的参数。 coroutine.wrap 会返回所有应该由除第一个（错误代码的那个布尔量） 之外的由 coroutine.resume 返回的值。 和 coroutine.resume 不同， coroutine.wrap 不捕获任何错误； 所有的错误都应该由调用者自己传递。

看下面这段代码展示的一个例子：

```
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"))
```

当你运行它，将得到如下输出结果：

```
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 - 程序接口（API）

这个部分描述了 Lua 的 C API ， 也就是宿主程序跟 Lua 通讯用的一组 C 函数。 所有的 API 函数

按相关的类型以及常量都声明在头文件 `lua.h` 中。

虽然我们说的是“函数”，但一部分简单的 API 是以宏的形式提供的。 所有的这些宏都只使用它们的参数一次 （除了第一个参数，也就是 lua 状态机）， 因此你不需担心这些宏的展开会引起一些副作用。

在所有的 C 库中，Lua API 函数都不去检查参数的有效性和坚固性。 然而，你可以在编译 Lua 时加上打开一个宏开关来 开启 `luaconf.h` 文件中的宏 `luai_apicheck` 以改变这个行为。

3.1 - 堆栈

Lua 使用一个虚拟栈来和 C 传递值。 栈上的的每个元素都是一个 Lua 值 （**nil**，数字，字符串，等等）。

无论何时 Lua 调用 C，被调用的函数都得到一个新的栈， 这个栈独立于 C 函数本身的堆栈，也独立于以前的栈。 （译注：在 C 函数里，用 Lua API 不能访问到 Lua 状态机中本次调用之外的堆栈中的数据） 它里面包含了 Lua 传递给 C 函数的所有参数， 而 C 函数则把要返回的结果也放入堆栈以返回给调用者 （参见 `lua_CFunction`）。

方便起见，所有针对栈的 API 查询操作都不严格遵循栈的操作规则。 而是可以用一个索引来指向栈上的任何元素： 正的索引指的是栈上的绝对位置（从一开始）； 负的索引则指从栈顶开始的偏移量。 更详细的说明一下，如果堆栈有 n 个元素， 那么索引 1 表示第一个元素（也就是最先被压入堆栈的元素） 而索引 n 则指最后一个元素； 索引 -1 也是指最后一个元素（即栈顶的元素），索引 -n 是指第一个元素。 如果索引在 1 到栈顶之间（也就是，`1 ≤ abs(index) ≤ top`） 我们就说这是个有效的索引。

3.2 - 堆栈尺寸

当你使用 Lua API 时，就有责任保证其坚固性。 特别需要注意的是，你有责任控制不要堆栈溢出。 你可以使用 `lua_checkstack` 这个函数来扩大可用堆栈的尺寸。

无论何时 Lua 调用 C， 它都只保证 `LUA_MINSTACK` 这么多的堆栈空间可以使用。 `LUA_MINSTACK` 一般被定义为 20 ， 因此，只要你不是不断的把数据压栈，通常你不用关心堆栈大小。

所有的查询函数都可以接收一个索引，只要这个索引是任何栈提供的空间中的值。 栈能提供的最大空间是通过 `lua_checkstack` 来设置的。 这些索引被称作可接受的索引，通常我们把它定义为：

```
(index < 0 && abs(index) <= top) ||
(index > 0 && index <= stackspace)
```

注意，0 永远都不是一个可接受的索引。（译注：下文中凡提到的索引，没有特别注明的话，都指可接受的索引。）

3.3 - 伪索引

除了特别声明外，任何一个函数都可以接受另一种有效的索引，它们被称作“伪索引”。 这个可以帮助 C 代码访问一些并不在栈上的 Lua 值。 伪索引被用来访问线程的环境，函数的环境，注册表，还有 C 函数的 upvalue （参见 §3.4）。

线程的环境（也就是全局变量放的地方）通常在伪索引 `LUA_GLOBALSINDEX` 处。 正在运行的 C 函数的环境则放在伪索引 `LUA_ENVIRONINDEX` 之处。

你可以用常规的 table 操作来访问和改变全局变量的值，只需要指定环境表的位置。 举例而言，要访问全局变量的值，这样做：

```
lua_getfield(L, LUA_GLOBALSINDEX, varname);
```

3.4 - C Closure

当 C 函数被创建出来，我们有可能会把一些值关联在一起， 也就是创建一个 C closure ； 这些被关联起来的值被叫做 upvalue ， 它们可以在函数被调用的时候访问的到。 （参见 `lua_pushcclosure`）。

无论何时去调用 C 函数，函数的 upvalue 都被放在指定的伪索引处。 我们可以用 `lua_upvalueindex` 这个宏来生成这些伪索引。 第一个关联到函数的值放在 `lua_upvalueindex(1)` 位置处，依次类推。 任何情况下都可以用 `lua_upvalueindex(`*n*`)` 产生一个 upvalue 的索引， 即使 n 大于实际的 upvalue 数量也可以。它都可以产生一个可接受但不一定有效的索引。

3.5 - 注册表

Lua 提供了一个注册表，这是一个预定义出来的表，可以用来保存任何 C 代码想保存的 Lua 值。 这个表可以用伪索引 `LUA_REGISTRYINDEX` 来定位。 任何 C 库都可以在这张表里保存数据，为了防止冲突，你需要特别小心的选择键名。 一般的用法是，你可以用一个包含你的库名的字符串做为键名，或者可以取你自己 C 代码 中的一个地址，以 light userdata 的形式做键。

注册表里的整数健被用于补充库中实现的引用系统的工作，一般说来不要把它们用于别的用途。

3.6 - C 中的错误处理

在内部实现中，Lua 使用了 C 的 `longjmp` 机制来处理错误。 （如果你使用 C++ 的话，也可以选择换用异常；参见 `luaconf.h` 文件。） 当 Lua 碰到任何错误（比如内存分配错误、类型错误、语法错误、还有一些运行时错误） 它都会产生一个错误出去； 也就是调用一个 long jump 。 在保护环境下，Lua 使用 `setjmp` 来设置一个恢复点； 任何发生的错误都会激活最近的一个恢复点。

几乎所有的 API 函数都可能产生错误，例如内存分配错误。 但下面的一些函数运行在保护环境中

（也就是说它们创建了一个保护环境再在其中运行）， 因此它们不会产生错误出来: lua_newstate, lua_close, lua_load, lua_pcall, and lua_cpcall。

在 C 函数里，你也可以通过调用 lua_error 产生一个错误。

3.7 - 函数和类型

在这里我们按字母次序列出了所有 C API 中的函数和类型。

---

lua_Alloc

```
typedef void * (*lua_Alloc) (void *ud,
                             void *ptr,
                             size_t osize,
                             size_t nsize);
```

Lua 状态机中使用的内存分配器函数的类型。 内存分配函数必须提供一个功能类似于 realloc 但又不完全相同的函数。 它的参数有 ud ， 一个由 lua_newstate 传给它的指针； ptr ， 一个指向已分配出来或是将被重新分配或是要释放的内存块指针； osize ， 内存块原来的尺寸； nsize ， 新内存块的尺寸。 如果且只有 osize 是零时，ptr 为 NULL 。 当 nsize 是零，分配器必须返回 NULL； 如果 osize 不是零，分配器应当释放掉 ptr 指向的内存块。 当 nsize 不是零，若分配器不能满足请求时，分配器返回 NULL 。 当 nsize 不是零而 osize 是零时，分配器应该和 malloc 有相同的行为。 当 nsize 和 osize 都不是零时，分配器则应和 realloc 保持一样的行为。 Lua 假设分配器在 osize >= nsize 时永远不会失败。

这里有一个简单的分配器函数的实现。 这个实现被放在补充库中，由 luaL_newstate 提供。

```
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);
}
```

这段代码假设 free(NULL) 啥也不影响，而且 realloc(NULL, size) 等价于 malloc(size)。 这两点是 ANSI C 保证的行为。

---

lua_atpanic

```
lua_CFunction lua_atpanic (lua_State *L, lua_CFunction panicf);
```

设置一个新的 panic （恐慌） 函数，并返回前一个。

如果在保护环境之外发生了任何错误， Lua 就会调用一个 panic 函数，接着调用 `exit(EXIT_FAILURE)`， 这样就开始退出宿主程序。 你的 panic 函数可以永远不返回（例如作一次长跳转）来避免程序退出。

panic 函数可以从栈顶取到出错信息。

---

lua_call

```
void lua_call (lua_State *L, int nargs, int nresults);
```

调用一个函数。

要调用一个函数请遵循以下协议： 首先，要调用的函数应该被压入堆栈； 接着，把需要传递给这个函数的参数按正序压栈； 这是指第一个参数首先压栈。 最后调用一下 lua_call； `nargs` 是你压入堆栈的参数个数。 当函数调用完毕后，所有的参数以及函数本身都会出栈。 而函数的返回值这时则被压入堆栈。 返回值的个数将被调整为 `nresults` 个， 除非 `nresults` 被设置成 `LUA_MULTRET`。 在这种情况下，所有的返回值都被压入堆栈中。 Lua 会保证返回值都放入栈空间中。 函数返回值将按正序压栈（第一个返回值首先压栈）， 因此在调用结束后，最后一个返回值将被放在栈顶。

被调用函数内发生的错误将（通过 `longjmp`）一直上抛。

下面的例子中，这行 Lua 代码等价于在宿主程序用 C 代码做一些工作:

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

这里是 C 里的代码:

```
lua_getfield(L, LUA_GLOBALSINDEX, "f");          /* 将调用的函数 */
lua_pushstring(L, "how");                          /* 第一个参数 */
lua_getfield(L, LUA_GLOBALSINDEX, "t");          /* table 的索引 */
lua_getfield(L, -1, "x");         /* 压入 t.x 的值（第 2 个参数）*/
lua_remove(L, -2);                              /* 从堆栈中移去 't' */
lua_pushinteger(L, 14);                           /* 第 3 个参数 */
lua_call(L, 3, 1); /* 调用 'f'，传入 3 个参数，并索取 1 个返回值 */
lua_setfield(L, LUA_GLOBALSINDEX, "a");      /* 设置全局变量 'a' */
```

注意上面这段代码是“平衡”的： 到了最后，堆栈恢复成原由的配置。 这是一种良好的编程习惯。

---

lua_CFunction

```
typedef int (*lua_CFunction) (lua_State *L);
```

C 函数的类型。

为了正确的和 Lua 通讯，C 函数必须使用下列 定义了参数以及返回值传递方法的协议： C 函数通

过 Lua 中的堆栈来接受参数，参数以正序入栈（第一个参数首先入栈）。 因此，当函数开始的时候， `lua_gettop(L)` 可以返回函数收到的参数个数。 第一个参数（如果有的话）在索引 1 的地方，而最后一个参数在索引 `lua_gettop(L)` 处。 当需要向 Lua 返回值的时候，C 函数只需要把它们以正序压到堆栈上（第一个返回值最先压入）， 然后返回这些返回值的个数。 在这些返回值之下的，堆栈上的东西都会被 Lua 丢掉。 和 Lua 函数一样，从 Lua 中调用 C 函数也可以有很多返回值。

下面这个例子中的函数将接收若干数字参数，并返回它们的平均数与和：

```
static int foo (lua_State *L) {
  int n = lua_gettop(L);    /* 参数的个数 */
  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);   /* 第一个返回值 */
  lua_pushnumber(L, sum);     /* 第二个返回值 */
  return 2;                   /* 返回值的个数 */
}
```

---

lua_checkstack

```
int lua_checkstack (lua_State *L, int extra);
```

确保堆栈上至少有 `extra` 个空位。 如果不能把堆栈扩展到相应的尺寸，函数返回 false 。 这个函数永远不会缩小堆栈； 如果堆栈已经比需要的大了，那么就放在那里不会产生变化。

---

lua_close

```
void lua_close (lua_State *L);
```

销毁指定 Lua 状态机中的所有对象（如果有垃圾收集相关的元方法的话，会调用它们）， 并且释放状态机中使用的所有动态内存。 在一些平台上，你可以不必调用这个函数， 因为当宿主程序结束的时候，所有的资源就自然被释放掉了。 另一方面，长期运行的程序，比如一个后台程序或是一个 web 服务器， 当不再需要它们的时候就应该释放掉相关状态机。这样可以避免状态机扩张的过大。

---

lua_concat

```
void lua_concat (lua_State *L, int n);
```

连接栈顶的 `n` 个值， 然后将这些值出栈，并把结果放在栈顶。 如果 `n` 为 1 ，结果就是一个字符串

放在栈上（即，函数什么都不做）； 如果 n 为 0 ，结果是一个空串。 连接依照 Lua 中创建语义完成（参见 §2.5.4 ）。

---

lua_cpcall

```
int lua_cpcall (lua_State *L, lua_CFunction func, void *ud);
```

以保护模式调用 C 函数 func 。 func 只有能从堆栈上拿到一个参数，就是包含有 ud 的 light userdata。 当有错误时， lua_cpcall 返回和 lua_pcall 相同的错误代码， 并在栈顶留下错误对象； 否则它返回零，并不会修改堆栈。 所有从 func 内返回的值都会被扔掉。

---

lua_createtable

```
void lua_createtable (lua_State *L, int narr, int nrec);
```

创建一个新的空 table 压入堆栈。 这个新 table 将被预分配 narr 个元素的数组空间 以及 nrec 个元素的非数组空间。 当你明确知道表中需要多少个元素时，预分配就非常有用。 如果你不知道，可以使用函数 lua_newtable。

---

lua_dump

```
int lua_dump (lua_State *L, lua_Writer writer, void *data);
```

把函数 dump 成二进制 chunk 。 函数接收栈顶的 Lua 函数做参数，然后生成它的二进制 chunk 。 若被 dump 出来的东西被再次加载，加载的结果就相当于原来的函数。 当它在产生 chunk 的时候，lua_dump 通过调用函数 writer （参见 lua_Writer） 来写入数据，后面的 data 参数会被传入 writer 。

最后一次由写入器 (writer) 返回值将作为这个函数的返回值返回； 0 表示没有错误。

这个函数不会把 Lua 返回弹出堆栈。

---

lua_equal

```
int lua_equal (lua_State *L, int index1, int index2);
```

如果依照 Lua 中 == 操作符语义，索引 index1 和 index2 中的值相同的话，返回 1 。 否则返回 0 。 如果任何一个索引无效也会返回 0。

---

lua_error

```
int lua_error (lua_State *L);
```

产生一个 Lua 错误。 错误信息（实际上可以是任何类型的 Lua 值）必须被置入栈顶。 这个函数会做一次长跳转，因此它不会再返回。 （参见 luaL_error）。

---

lua_gc

```
int lua_gc (lua_State *L, int what, int data);
```

控制垃圾收集器。

这个函数根据其参数 what 发起几种不同的任务：

- **LUA_GCSTOP:** 停止垃圾收集器。
- **LUA_GCRESTART:** 重启垃圾收集器。
- **LUA_GCCOLLECT:** 发起一次完整的垃圾收集循环。
- **LUA_GCCOUNT:** 返回 Lua 使用的内存总量（以 K 字节为单位）。
- **LUA_GCCOUNTB:** 返回当前内存使用量除以 1024 的余数。
- **LUA_GCSTEP:** 发起一步增量垃圾收集。 步数由 data 控制（越大的值意味着越多步）， 而其具体含义（具体数字表示了多少）并未标准化。 如果你想控制这个步数，必须实验性的测试 data 的值。 如果这一步结束了一个垃圾收集周期，返回返回 1 。
- **LUA_GCSETPAUSE:** 把 data/100 设置为 *garbage-collector pause* 的新值（参见 §2.10）。 函数返回以前的值。
- **LUA_GCSETSTEPMUL:** 把 arg/100 设置成 *step multiplier* （参见 §2.10）。 函数返回以前的值。

---

lua_getallocf

```
lua_Alloc lua_getallocf (lua_State *L, void **ud);
```

返回给定状态机的内存分配器函数。 如果 ud 不是 NULL ， Lua 把调用 lua_newstate 时传入的那个指针放入 *ud 。

---

lua_getfenv

```
void lua_getfenv (lua_State *L, int index);
```

把索引处值的环境表压入堆栈。

---

lua_getfield

```
void lua_getfield (lua_State *L, int index, const char *k);
```

把 t[k] 值压入堆栈， 这里的 t 是指有效索引 index 指向的值。 在 Lua 中，这个函数可能触发对应 "index" 事件的元方法 （参见 §2.8）。

---

lua_getglobal

```
void lua_getglobal (lua_State *L, const char *name);
```

把全局变量 name 里的值压入堆栈。 这个是用一个宏定义出来的：

```
#define lua_getglobal(L,s)  lua_getfield(L, LUA_GLOBALSINDEX, s)
```

---

lua_getmetatable

```
int lua_getmetatable (lua_State *L, int index);
```

把给定索引指向的值的元表压入堆栈。 如果索引无效，或是这个值没有元表， 函数将返回 0 并且不会向栈上压任何东西。

---

lua_gettable

```
void lua_gettable (lua_State *L, int index);
```

把 t[k] 值压入堆栈， 这里的 t 是指有效索引 index 指向的值， 而 k 则是栈顶放的值。

这个函数会弹出堆栈上的 key （把结果放在栈上相同位置）。 在 Lua 中，这个函数可能触发对应 "index" 事件的元方法 （参见 §2.8）。

---

lua_gettop

```
int lua_gettop (lua_State *L);
```

返回栈顶元素的索引。 因为索引是从 1 开始编号的， 所以这个结果等于堆栈上的元素个数（因此返回 0 表示堆栈为空）。

---

lua_insert

```
void lua_insert (lua_State *L, int index);
```

把栈顶元素插入指定的有效索引处， 并依次移动这个索引之上的元素。 不要用伪索引来调用这个函数， 因为伪索引不是真正指向堆栈上的位置。

---

lua_Integer

```
typedef ptrdiff_t lua_Integer;
```

这个类型被用于 Lua API 接收整数值。

缺省时这个被定义为 ptrdiff_t ， 这个东西通常是机器能处理的最大整数类型。

---

lua_isboolean

```
int lua_isboolean (lua_State *L, int index);
```

当给定索引的值类型为 boolean 时，返回 1 ，否则返回 0 。

---

lua_iscfunction

```
int lua_iscfunction (lua_State *L, int index);
```

当给定索引的值是一个 C 函数时，返回 1 ，否则返回 0 。

---

lua_isfunction

```
int lua_isfunction (lua_State *L, int index);
```

当给定索引的值是一个函数（ C 或 Lua 函数均可）时，返回 1 ，否则返回 0 。

---

lua_islightuserdata

```
int lua_islightuserdata (lua_State *L, int index);
```

当给定索引的值是一个 light userdata 时，返回 1 ，否则返回 0 。

---

lua_isnil

```
int lua_isnil (lua_State *L, int index);
```

当给定索引的值是 **nil** 时，返回 1 ，否则返回 0 。

---

lua_isnumber

```
int lua_isnumber (lua_State *L, int index);
```

当给定索引的值是一个数字，或是一个可转换为数字的字符串时，返回 1 ，否则返回 0 。

---

lua_isstring

```
int lua_isstring (lua_State *L, int index);
```

当给定索引的值是一个字符串或是一个数字（数字总能转换成字符串）时，返回 1 ，否则返回 0 。

---

lua_istable

```
int lua_istable (lua_State *L, int index);
```

当给定索引的值是一个 table 时，返回 1 ，否则返回 0 。

---

lua_isthread

```
int lua_isthread (lua_State *L, int index);
```

当给定索引的值是一个 thread 时，返回 1 ，否则返回 0 。

---

lua_isuserdata

```
int lua_isuserdata (lua_State *L, int index);
```

当给定索引的值是一个 userdata （无论是完整的 userdata 还是 light userdata ）时，返回 1 ，否则返回 0 。

---

lua_lessthan

```
int lua_lessthan (lua_State *L, int index1, int index2);
```

如果索引 `index1` 处的值小于 索引 `index2` 处的值时，返回 1 ； 否则返回 0 。 其语义遵循 Lua 中的 < 操作符（就是说，有可能调用元方法）。 如果任何一个索引无效，也会返回 0 。

---

lua_load

```
int lua_load (lua_State *L,
              lua_Reader reader,
              void *data,
              const char *chunkname);
```

加载一个 Lua chunk 。 如果没有错误， lua_load 把一个编译好的 chunk 作为一个 Lua 函数压入堆栈。 否则，压入出错信息。 lua_load 的返回值可以是：

- **0:** 没有错误；
- **LUA_ERRSYNTAX:** 在预编译时碰到语法错误；
- **LUA_ERRMEM:** 内存分配错误。

这个函数仅仅加栽 chunk ； 而不会去运行它。

lua_load 会自动检测 chunk 是文本的还是二进制的， 然后做对应的加载操作（参见程序 `luac`）。

lua_load 函数使用一个用户提供的 `reader` 函数来 读取 chunk （参见 lua_Reader）。 `data` 参数会

被传入读取器函数。

chunkname 这个参数可以赋予 chunk 一个名字， 这个名字被用于出错信息和调试信息（参见 §3.8）。

---

lua_newstate

```
lua_State *lua_newstate (lua_Alloc f, void *ud);
```

创建的一个新的独立的状态机。 如果创建不了（因为内存问题）返回 NULL 。 参数 f 是一个分配器函数； Lua 将通过这个函数做状态机内所有的内存分配操作。 第二个参数 ud ， 这个指针将在每次调用分配器时被直接传入。

---

lua_newtable

```
void lua_newtable (lua_State *L);
```

创建一个空 table ， 并将之压入堆栈。 它等价于 lua_createtable(L, 0, 0) 。

---

lua_newthread

```
lua_State *lua_newthread (lua_State *L);
```

创建一个新线程，并将其压入堆栈， 并返回维护这个线程的 lua_State 指针。 这个函数返回的新状态机共享原有状态机中的所有对象（比如一些 table）， 但是它有独立的执行堆栈。

没有显式的函数可以用来关闭或销毁掉一个线程。 线程跟其它 Lua 对象一样是垃圾收集的条目之一。

---

lua_newuserdata

```
void *lua_newuserdata (lua_State *L, size_t size);
```

这个函数分配分配一块指定大小的内存块， 把内存块地址作为一个完整的 userdata 压入堆栈，并返回这个地址。

userdata 代表 Lua 中的 C 值。 完整的 userdata 代表一块内存。 它是一个对象（就像 table 那样的对象）： 你必须创建它，它有着自己的元表，而且它在被回收时，可以被监测到。 一个完整的 userdata 只和它自己相等（在等于的原生作用下）。

当 Lua 通过 gc 元方法回收一个完整的 userdata 时， Lua 调用这个元方法并把 userdata 标记为已终止。 等到这个 userdata 再次被收集的时候，Lua 会释放掉相关的内存。

---

lua_next

```
int lua_next (lua_State *L, int index);
```

从栈上弹出一个 key（键）， 然后把索引指定的表中 key-value（健值）对压入堆栈 （指定 key 后面的下一 (next) 对）。 如果表中以无更多元素， 那么 lua_next 将返回 0 （什么也不压入堆栈）。

典型的遍历方法是这样的:

```
/* table 放在索引 't' 处 */
lua_pushnil(L);  /* 第一个 key */
while (lua_next(L, t) != 0) {
  /* 用一下 'key' （在索引 -2 处） 和 'value' （在索引 -1 处） */
  printf("%s - %s\n",
         lua_typename(L, lua_type(L, -2)),
         lua_typename(L, lua_type(L, -1)));
  /* 移除 'value' ；保留 'key' 做下一次迭代 */
  lua_pop(L, 1);
}
```

在遍历一张表的时候， 不要直接对 key 调用 lua_tolstring， 除非你知道这个 key 一定是一个字符串。 调用 lua_tolstring 有可能改变给定索引位置的值； 这会对下一次调用 lua_next 造成影响。

---

lua_Number

```
typedef double lua_Number;
```

Lua 中数字的类型。 确省是 double ， 但是你可以在 luaconf.h 中修改它。

通过修改配置文件你可以改变 Lua 让它操作其它数字类型（例如： float 或是 long ）。

---

lua_objlen

```
size_t lua_objlen (lua_State *L, int index);
```

返回指定的索引处的值的长度。 对于 string ， 那就是字符串的长度； 对于 table ， 是取长度操作符 ('#') 的结果； 对于 userdata ， 就是为其分配的内存块的尺寸； 对于其它值， 为 0 。

---

lua_pcall

```
lua_pcall (lua_State *L, int nargs, int nresults, int errfunc);
```

以保护模式调用一个函数。

nargs 和 nresults 的含义与 lua_call 中的相同。 如果在调用过程中没有发生错误， lua_pcall

的行为和 lua_call 完全一致。 但是，如果有错误发生的话， lua_pcall 会捕获它， 然后把单一的值（错误信息）压入堆栈，然后返回错误码。 同 lua_call 一样， lua_pcall 总是把函数本身和它的参数从栈上移除。

如果 errfunc 是 0 ， 返回在栈顶的错误信息就和原始错误信息完全一致。 否则，errfunc 就被当成是错误处理函数在栈上的索引。 （在当前的实现里，这个索引不能是伪索引。） 在发生运行时错误时， 这个函数会被调用而参数就是错误信息。 错误处理函数的返回值将被 lua_pcall 作为出错信息返回在堆栈上。

典型的用法中，错误处理函数被用来在出错信息上加上更多的调试信息，比如栈跟踪信息 (stack traceback)。 这些信息在 lua_pcall 返回后，因为栈已经展开 (unwound)， 所以收集不到了。

lua_pcall 函数在调用成功时返回 0 ， 否则返回以下（定义在 lua.h 中的）错误代码中的一个:

- **LUA_ERRRUN:** 运行时错误。
- **LUA_ERRMEM:** 内存分配错误。 对于这种错，Lua 调用不了错误处理函数。
- **LUA_ERRERR:** 在运行错误处理函数时发生的错误。

---

lua_pop

```
void lua_pop (lua_State *L, int n);
```

从堆栈中弹出 n 个元素。

---

lua_pushboolean

```
void lua_pushboolean (lua_State *L, int b);
```

把 b 作为一个 boolean 值压入堆栈。

---

lua_pushcclosure

```
void lua_pushcclosure (lua_State *L, lua_CFunction fn, int n);
```

把一个新的 C closure 压入堆栈。

当创建了一个 C 函数后，你可以给它关联一些值，这样就是在创建一个 C closure （参见 §3.4）； 接下来无论函数何时被调用，这些值都可以被这个函数访问到。 为了将一些值关联到一个 C 函数上， 首先这些值需要先被压入堆栈（如果有多个值，第一个先压）。 接下来调用 lua_pushcclosure 来创建出 closure 并把这个 C 函数压到堆栈上。 参数 n 告之函数有多少个值需要关联到函数上。 lua_pushcclosure 也会把这些值从栈上弹出。

---

lua_pushcfunction

```
void lua_pushcfunction (lua_State *L, lua_CFunction f);
```

将一个 C 函数压入堆栈。 这个函数接收一个 C 函数指针，并将一个类型为 function 的 Lua 值 压入堆栈。当这个栈顶的值被调用时，将触发对应的 C 函数。

注册到 Lua 中的任何函数都必须遵循正确的协议来接收参数和返回值 （参见 lua_CFunction）。

lua_pushcfunction 是作为一个宏定义出现的：

```
#define lua_pushcfunction(L,f)  lua_pushcclosure(L,f,0)
```

---

lua_pushfstring

```
const char *lua_pushfstring (lua_State *L, const char *fmt, ...);
```

把一个格式化过的字符串压入堆栈，然后返回这个字符串的指针。 它和 C 函数 sprintf 比较像，不过有一些重要的区别：

- 摸你需要为结果分配空间： 其结果是一个 Lua 字符串，由 Lua 来关心其内存分配 （同时通过垃圾收集来释放内存）。
- 这个转换非常的受限。 不支持 flag ，宽度，或是指定精度。 它只支持下面这些： '%%' （插入一个 '%'）， '%s' （插入一个带零终止符的字符串，没有长度限制）， '%f' （插入一个 lua_Number）， '%p' （插入一个指针或是一个十六进制数）， '%d' （插入一个 int)， '%c' （把一个 int 作为一个字符插入）。

---

lua_pushinteger

```
void lua_pushinteger (lua_State *L, lua_Integer n);
```

把 n 作为一个数字压栈。

---

lua_pushlightuserdata

```
void lua_pushlightuserdata (lua_State *L, void *p);
```

把一个 light userdata 压栈。

userdata 在 Lua 中表示一个 C 值。 light userdata 表示一个指针。 它是一个像数字一样的值： 你不需要专门创建它，它也没有独立的 metatable ， 而且也不会被收集（因为从来不需要创建）。 只要表示的 C 地址相同，两个 light userdata 就相等。

---

lua_pushlstring

```
void lua_pushlstring (lua_State *L, const char *s, size_t len);
```

把指针 s 指向的长度为 len 的字符串压栈。 Lua 对这个字符串做一次内存拷贝（或是复用一个拷贝）， 因此 s 处的内存在函数返回后，可以释放掉或是重用于其它用途。 字符串内可以保存有零字符。

---

lua_pushnil

```
void lua_pushnil (lua_State *L);
```

把一个 nil 压栈。

---

lua_pushnumber

```
void lua_pushnumber (lua_State *L, lua_Number n);
```

把一个数字 n 压栈。

---

lua_pushstring

```
void lua_pushstring (lua_State *L, const char *s);
```

把指针 s 指向的以零结尾的字符串压栈。 Lua 对这个字符串做一次内存拷贝（或是复用一个拷贝）， 因此 s 处的内存在函数返回后，可以释放掉或是重用于其它用途。 字符串中不能包含有零字符；第一个碰到的零字符会认为是字符串的结束。

---

lua_pushthread

```
int lua_pushthread (lua_State *L);
```

把 L 中提供的线程压栈。 如果这个线程是当前状态机的主线程的话，返回 1 。

---

lua_pushvalue

```
void lua_pushvalue (lua_State *L, int index);
```

把堆栈上给定有效处索引处的元素作一个拷贝压栈。

---

lua_pushvfstring

```
const char *lua_pushvfstring (lua_State *L,
                              const char *fmt,
                              va_list argp);
```

等价于 lua_pushfstring， 不过是用 va_list 接收参数，而不是用可变数量的实际参数。

---

lua_rawequal

```
int lua_rawequal (lua_State *L, int index1, int index2);
```

如果两个索引 index1 和 index2 处的值简单地相等 （不调用元方法）则返回 1 。 否则返回 0 。 如果任何一个索引无效也返回 0 。

---

lua_rawget

```
void lua_rawget (lua_State *L, int index);
```

类似于 lua_gettable， 但是作一次直接访问（不触发元方法）。

---

lua_rawgeti

```
void lua_rawgeti (lua_State *L, int index, int n);
```

把 t[n] 的值压栈， 这里的 t 是指给定索引 index 处的一个值。 这是一个直接访问；就是说，它不会触发元方法。

---

lua_rawset

```
void lua_rawset (lua_State *L, int index);
```

类似于 lua_settable， 但是是作一个直接赋值（不触发元方法）。

---

lua_rawseti

```
void lua_rawseti (lua_State *L, int index, int n);
```

等价于 t[n] = v， 这里的 t 是指给定索引 index 处的一个值， 而 v 是栈顶的值。

函数将把这个值弹出栈。 赋值操作是直接的；就是说，不会触发元方法。

---

lua_Reader

```
typedef const char * (*lua_Reader) (lua_State *L,
                                    void *data,
                                    size_t *size);
```

lua_load 用到的读取器函数， 每次它需要一块新的 chunk 的时候， lua_load 就调用读取器， 每次都会传入一个参数 data 。 读取器需要返回含有新的 chunk 的一块内存的指针， 并把 size 设为这块内存的大小。 内存块必须在下一次函数被调用之前一直存在。 读取器可以通过返回一个 NULL 来指示 chunk 结束。 读取器可能返回多个块，每个块可以有任意的大于零的尺寸。

---

lua_register

```
void lua_register (lua_State *L,
                   const char *name,
                   lua_CFunction f);
```

把 C 函数 f 设到全局变量 name 中。 它通过一个宏定义：

```
    #define lua_register(L,n,f) \
           (lua_pushcfunction(L, f), lua_setglobal(L, n))
```

---

lua_remove

```
void lua_remove (lua_State *L, int index);
```

从给定有效索引处移除一个元素， 把这个索引之上的所有元素移下来填补上这个空隙。 不能用伪索引来调用这个函数， 因为伪索引并不指向真实的栈上的位置。

---

lua_replace

```
void lua_replace (lua_State *L, int index);
```

把栈顶元素移动到给定位置（并且把这个栈顶元素弹出）， 不移动任何元素（因此在那个位置处的值被覆盖掉）。

---

lua_resume

```
int lua_resume (lua_State *L, int narg);
```

在给定线程中启动或继续一个 coroutine 。

要启动一个 coroutine 的话，首先你要创建一个新线程 （参见 lua_newthread ）； 然后把主函数和若干参数压到新线程的堆栈上； 最后调用 lua_resume ， 把 narg 设为参数的个数。 这次调用会在 coroutine 挂起时或是结束运行后返回。 当函数返回时，堆栈中会有传给 lua_yield 的所有值， 或是主函数的所有返回值。 如果 coroutine 切换时，lua_resume 返回 LUA_YIELD ， 而当 coroutine 结束运行且没有任何错误时，返回 0 。 如果有错则返回错误代码（参见 lua_pcall）。 在发生错误的情况下， 堆栈没有展开， 因此你可以使用 debug API 来处理它。 出错信息放在栈顶。 要继续运行一个 coroutine 的话，你把需要传给 yield 作结果的返回值压入堆栈，然后调用 lua_resume 。

---

lua_setallocf

```
void lua_setallocf (lua_State *L, lua_Alloc f, void *ud);
```

把指定状态机的分配器函数换成带上指针 ud 的 f 。

---

lua_setfenv

```
int lua_setfenv (lua_State *L, int index);
```

从堆栈上弹出一个 table 并把它设为指定索引处值的新环境。 如果指定索引处的值即不是函数又不是线程或是 userdata ， lua_setfenv 会返回 0 ， 否则返回 1 。

---

lua_setfield

```
void lua_setfield (lua_State *L, int index, const char *k);
```

做一个等价于 t[k] = v 的操作， 这里 t 是给出的有效索引 index 处的值， 而 v 是栈顶的那个值。

这个函数将把这个值弹出堆栈。 跟在 Lua 中一样，这个函数可能触发一个 "newindex" 事件的元方法 （参见 §2.8）。

---

lua_setglobal

```
void lua_setglobal (lua_State *L, const char *name);
```

从堆栈上弹出一个值，并将其设到全局变量 name 中。 它由一个宏定义出来：

```
#define lua_setglobal(L,s)   lua_setfield(L, LUA_GLOBALSINDEX, s)
```

---

lua_setmetatable

```
int lua_setmetatable (lua_State *L, int index);
```

把一个 table 弹出堆栈，并将其设为给定索引处的值的 metatable 。

---

lua_settable

```
void lua_settable (lua_State *L, int index);
```

作一个等价于 t[k] = v 的操作， 这里 t 是一个给定有效索引 index 处的值， v 指栈顶的值， 而 k 是栈顶之下的那个值。

这个函数会把键和值都从堆栈中弹出。 和在 Lua 中一样，这个函数可能触发 "newindex" 事件的元方法 （参见 §2.8）。

---

lua_settop

```
void lua_settop (lua_State *L, int index);
```

参数允许传入任何可接受的索引以及 0 。 它将把堆栈的栈顶设为这个索引。 如果新的栈顶比原来的大，超出部分的新元素将被填为 **nil** 。 如果 index 为 0 ， 把栈上所有元素移除。

---

lua_State

```
typedef struct lua_State lua_State;
```

一个不透明的结构，它保存了整个 Lua 解释器的状态。 Lua 库是完全可重入的： 它没有任何全局变量。 （译注：从 C 语法上来说，也不尽然。例如，在 table 的实现中 用了一个静态全局变量 dummynode_ ，但这在正确使用时并不影响可重入性。 只是万一你错误链接了 lua 库，不小心在同一进程空间中存在两份 lua 库实现的代码的话， 多份 dummynode_ 不同的地址会导致一些问题。） 所有的信息都保存在这个结构中。

这个状态机的指针必须作为第一个参数传递给每一个库函数。 lua_newstate 是一个例外， 这个函数会从头创建一个 Lua 状态机。

---

lua_status

```
int lua_status (lua_State *L);
```

返回线程 L 的状态。

正常的线程状态是 0 。 当线程执行完毕或发生一个错误时，状态值是错误码。 如果线程被挂起，状态为 LUA_YIELD 。

---

lua_toboolean

```
int lua_toboolean (lua_State *L, int index);
```

把指定的索引处的的 Lua 值转换为一个 C 中的 boolean 值（ 0 或是 1 ）。 和 Lua 中做的所有测试一样， lua_toboolean 会把任何 不同于 **false** 和 **nil** 的值当作 1 返回； 否则就返回 0 。 如果用一个无效索引去调用也会返回 0 。 （如果你想只接收真正的 boolean 值，就需要使用 lua_isboolean 来测试值的类型。）

---

lua_tocfunction

```
lua_CFunction lua_tocfunction (lua_State *L, int index);
```

把给定索引处的 Lua 值转换为一个 C 函数。 这个值必须是一个 C 函数；如果不是就返回 NULL 。

---

lua_tointeger

```
lua_Integer lua_tointeger (lua_State *L, int idx);
```

把给定索引处的 Lua 值转换为 lua_Integer 这样一个有符号整数类型。 这个 Lua 值必须是一个数字或是一个可以转换为数字的字符串 （参见 §2.2.1）； 否则，lua_tointeger 返回 0 。

如果数字不是一个整数， 截断小数部分的方式没有被明确定义。

---

lua_tolstring

```
const char *lua_tolstring (lua_State *L, int index, size_t *len);
```

把给定索引处的 Lua 值转换为一个 C 字符串。 如果 len 不为 NULL ， 它还把字符串长度设到 *len 中。 这个 Lua 值必须是一个字符串或是一个数字； 否则返回返回 NULL 。 如果值是一个数字，lua_tolstring 还会把堆栈中的那个值的实际类型转换为一个字符串。 （当遍历一个表的时候，把 lua_tolstring 作用在键上，这个转换有可能导致 lua_next 弄错。）

lua_tolstring 返回 Lua 状态机中 字符串的以对齐指针。 这个字符串总能保证 （ C 要求的）最后一个字符为零 ('\0') ， 而且它允许在字符串内包含多个这样的零。 因为 Lua 中可能发生垃圾收集， 所以不保证 lua_tolstring 返回的指针， 在对应的值从堆栈中移除后依然有效。

---

lua_tonumber

```
lua_Number lua_tonumber (lua_State *L, int index);
```

把给定索引处的 Lua 值转换为 lua_Number 这样一个 C 类型（参见 lua_Number ）。 这个 Lua 值必须是一个数字或是一个可转换为数字的字符串 （参见 §2.2.1 ）； 否则，lua_tonumber 返回 0 。

---

lua_topointer

```
const void *lua_topointer (lua_State *L, int index);
```

把给定索引处的值转换为一般的 C 指针 (void*) 。 这个值可以是一个 userdata ， table ， thread 或是一个 function ； 否则，lua_topointer 返回 NULL 。 不同的对象有不同的指针。 不存在把指针再转回原有类型的方法。

这个函数通常只为产生 debug 信息用。

---

lua_tostring

```
const char *lua_tostring (lua_State *L, int index);
```

等价于 lua_tolstring ， 而参数 len 设为 NULL 。

---

lua_tothread

```
lua_State *lua_tothread (lua_State *L, int index);
```

把给定索引处的值转换为一个 Lua 线程（由 `lua_State*` 代表）。 这个值必须是一个线程；否则函数返回 `NULL` 。

---

lua_touserdata

```
void *lua_touserdata (lua_State *L, int index);
```

如果给定索引处的值是一个完整的 userdata ， 函数返回内存块的地址。 如果值是一个 light userdata ， 那么就返回它表示的指针。 否则， 返回 `NULL` 。

---

lua_type

```
int lua_type (lua_State *L, int index);
```

返回给定索引处的值的类型， 当索引无效时则返回 `LUA_TNONE` （那是指一个指向堆栈上的空位置的索引）。 lua_type 返回的类型是一些个在 `lua.h` 中定义的常量： `LUA_TNIL` ， `LUA_TNUMBER` ， `LUA_TBOOLEAN` ， `LUA_TSTRING` ， `LUA_TTABLE` ， `LUA_TFUNCTION` ， `LUA_TUSERDATA` ， `LUA_TTHREAD` ， `LUA_TLIGHTUSERDATA` 。

---

lua_typename

```
const char *lua_typename  (lua_State *L, int tp);
```

返回 `tp` 表示的类型名， 这个 `tp` 必须是 lua_type 可能返回的值中之一。

---

lua_Writer

```
typedef int (*lua_Writer) (lua_State *L,
                           const void* p,
                           size_t sz,
                           void* ud);
```

由 lua_dump 用到的写入器函数。 每次 lua_dump 产生了一块新的 chunk ， 它都会调用写入器。 传入要写入的缓存 (`p`) 和它的尺寸 (`sz`) ， 还有 lua_dump 的参数 `data` 。

写入器会返回一个错误码： 0 表示没有错误； 别的值均表示一个错误，并且会让 lua_dump 停止再次调用写入器。

---

lua_xmove

```
void lua_xmove (lua_State *from, lua_State *to, int n);
```

传递 *同一个* 全局状态机下不同线程中的值。

这个函数会从 from 的堆栈中弹出 n 个值， 然后把它们压入 to 的堆栈中。

---

lua_yield

```
int lua_yield  (lua_State *L, int nresults);
```

切出一个 coroutine 。

这个函数只能在一个 C 函数的返回表达式中调用。如下：

```
return lua_yield (L, nresults);
```

当一个 C 函数这样调用 lua_yield ， 正在运行中的 coroutine 将从运行中挂起， 然后启动这个 coroutine 用的那次对 lua_resume 的调用就返回了。 参数 nresults 指的是堆栈中需要返回的结果个数，这些返回值将被传递给 lua_resume 。

3.8 - 调试接口

Lua 没有内建的调试设施。 取而代之的是提供了一些函数接口和钩子。 利用这些接口，可以做出一些不同类型的调试器， 性能分析器，或是其它一些需要从解释器中取到“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) upvalue 个数 */
  int linedefined;            /* (S) */
  int lastlinedefined;        /* (S) */
  char short_src[LUA_IDSIZE]; /* (S) */
  /* 私有部分 */
  其它域
} lua_Debug;
```

一个用来携带活动中函数的各种信息的结构。 lua_getstack 仅填写这个结构中的私有部分， 这些部分以后会用到。 调用 lua_getinfo 则可以填上 lua_Debug 中有用信息的那些域。

lua_Debug 中的各个域有下列含义：

- **source:** 如果函数是定义在一个字符串中，source 就是这个字符串。 如果函数定义在一个文件中， source 是一个以 '@' 开头的文件名。

- **short_src:** 一个“可打印版本”的 source，用于出错信息。
- **linedefined:** 函数定义开始处的行号。
- **lastlinedefined:** 函数定义结束处的行号。
- **what:** 如果函数是一个 Lua 函数，则为一个字符串 "Lua"； 如果是一个 C 函数，则为 "C"；如果它是一个 chunk 的主体部分，则为 "main"； 如果是一个作了尾调用的函数，则为 "tail"。 别的情况下，Lua 没有关于函数的别的信息。
- **currentline:** 给定函数正在执行的那一行。 当提供不了行号信息的时候，currentline 被设为 -1 。
- **name:** 给定函数的一个合理的名字。 因为 Lua 中的函数也是一个值， 所以它们没有固定的名字： 一些函数可能是全局复合变量的值， 另一些可能仅仅只是被保存在一个 table 中。 lua_getinfo 函数会检查函数是这样被调用的，以此来找到一个适合的名字。 如果它找不到名字，name 就被设置为 NULL 。
- **namewhat:** 结实 name 域。 namewhat 的值可以是 "global", "local", "method", "field", "upvalue", 或是 "" （空串）。 这取决于函数怎样被调用。 （Lua 用空串表示其它选项都不符合）
- **nups:** 函数的 upvalue 的个数。

---

lua_gethook

```
lua_Hook lua_gethook (lua_State *L);
```

返回当前的钩子函数。

---

lua_gethookcount

```
int lua_gethookcount (lua_State *L);
```

返回当前钩子记数。

---

lua_gethookmask

```
int lua_gethookmask (lua_State *L);
```

返回当前的钩子掩码 (mask) 。

---

lua_getinfo

```
int lua_getinfo (lua_State *L, const char *what, lua_Debug *ar);
```

返回一个指定的函数或函数调用的信息。

当用于取得一次函数调用的信息时， 参数 ar 必须是一个有效的活动的记录。 这条记录可以是前一次调用 lua_getstack 得到的， 或是一个钩子 （参见 lua_Hook）得到的参数。

用于获取一个函数的信息时，可以把这个函数压入堆栈， 然后把 what 字符串以字符 '>' 起头。（这个情况下，lua_getinfo 从栈顶上弹出函数。） 例如，想知道函数 f 在哪一行定义的， 你可以下下列代码:

```
lua_Debug ar;
lua_getfield(L, LUA_GLOBALSINDEX, "f");  /* 取到全局变量 'f' */
lua_getinfo(L, ">S", &ar);
printf("%d\n", ar.linedefined);
```

what 字符串中的每个字符都筛选出结构 ar 结构中一些域用于填充，或是把一个值压入堆栈:

- **'n':** 填充 name 及 namewhat 域;
- **'S':** 填充 source， short_src， linedefined， lastlinedefined，以及 what 域;
- **'l':** 填充 currentline 域;
- **'u':** 填充 nups 域;
- **'f':** 把正在运行中指定级别处函数压入堆栈； （译注：一般用于获取函数调用中的信息，级别是由 ar 中的私有部分来提供。 如果用于获取静态函数，那么就直接把指定函数重新压回堆栈， 但这样做通常无甚意义。）
- **'L':** 压一个 table 入栈，这个 table 中的整数索引用于描述函数中哪些行是有效行。 （有效行指有实际代码的行， 即你可以置入断点的行。 无效行包括空行和只有注释的行。）

这个函数出错会返回 0 （例如，what 中有一个无效选项）。

---

lua_getlocal

```
const char *lua_getlocal (lua_State *L, lua_Debug *ar, int n);
```

从给定活动记录中获取一个局部变量的信息。 参数 ar 必须是一个有效的活动的记录。 这条记录可以是前一次调用 lua_getstack 得到的， 或是一个钩子 （参见 lua_Hook）得到的参数。 索引 n 用于选择要检阅哪个局部变量 （ 1 表示第一个参数或是激活的第一个局部变量，以此类推，直到最后一个局部变量）。 lua_getlocal 把变量的值压入堆栈并返回它的名字。

以 '(' （正小括号）开始的变量指内部变量 （循环控制变量，临时变量，C 函数局部变量）。

当索引大于局部变量的个数时，返回 NULL （什么也不压入）。

---

lua_getstack

```
int lua_getstack (lua_State *L, int level, lua_Debug *ar);
```

获取解释器的运行时栈的信息。

这个函数用正在运行中的给定级别处的函数的活动记录来填写 lua_Debug 结构的一部分。 0 级表示当前运行的函数， 而 n+1 级处的函数就是调用第 n 级函数的那一个。 如果没有错误，lua_getstack 返回 1； 当调用传入的级别大于堆栈深度的时候，返回 0 。

---

lua_getupvalue

```
const char *lua_getupvalue (lua_State *L, int funcindex, int n);
```

获取一个 closure 的 upvalue 信息。（对于 Lua 函数，upvalue 是函数需要使用的外部局部变量，因此这些变量被包含在 closure 中。） lua_getupvalue 获取第 n 个 upvalue， 把这个 upvalue 的值压入堆栈，并且返回它的名字。 funcindex 指向堆栈上 closure 的位置。（因为 upvalue 在整个函数中都有效，所以它们没有特别的次序。 因此，它们以字母次序来编号。）

当索引号比 upvalue 数量大的时候，返回 NULL （而且不会压入任何东西） 对于 C 函数，这个函数用空串 "" 表示所有 upvalue 的名字。

---

lua_Hook

```
typedef void (*lua_Hook) (lua_State *L, lua_Debug *ar);
```

用于调试的钩子函数类型。

无论何时钩子被调用，它的参数 ar 中的 event 域 都被设为触发钩子的事件。 Lua 把这些事件定义为以下常量： LUA_HOOKCALL， LUA_HOOKRET, LUA_HOOKTAILRET， LUA_HOOKLINE， and LUA_HOOKCOUNT。 除此之外，对于 line 事件，currentline 域也被设置。 要想获得 ar 中的其他域， 钩子必须调用 lua_getinfo。 对于返回事件，event 的正常值可能是 LUA_HOOKRET， 或者是 LUA_HOOKTAILRET 。 对于后一种情况，Lua 会对一个函数做的尾调用也模拟出一个返回事件出来；对于这个模拟的返回事件，调用 lua_getinfo 没有什么作用。

当 Lua 运行在一个钩子内部时，它将屏蔽掉其它对钩子的调用。 也就是说，如果一个钩子函数内再调回 Lua 来执行一个函数或是一个 chunk， 这个执行操作不会触发任何的钩子。

---

lua_sethook

```
int lua_sethook (lua_State *L, lua_Hook f, int mask, int count);
```

设置一个调试用钩子函数。

参数 f 是钩子函数。 mask 指定在哪些事件时会调用： 它由下列一组位常量构成 LUA_MASKCALL，LUA_MASKRET， LUA_MASKLINE， 以及 LUA_MASKCOUNT。 参数 count 只在 mask 中包含有 LUA_MASKCOUNT 才有意义。 对于每个事件，钩子被调用的情况解释如下：

- **call hook:** 在解释器调用一个函数时被调用。 钩子将于 Lua 进入一个新函数后，函数获取参数前被调用。
- **return hook:** 在解释器从一个函数中返回时调用。 钩子将于 Lua 离开函数之前的那一刻被调用。 你无权访问被函数返回出去的那些值。 （译注：原文 (You have no access to the values to be returned by the function) 如此。 但“无权访问”一词值得商榷。 某些情况下你可以访问到一些被命名为 (*temporary) 的局部变量， 那些索引被排在最后的 (*temporary) 变量指的就是返回值。 但是由于 Lua 对特殊情况做了一些优

化，比如直接返回一个被命名的局部变量， 那么就找不到对应的 (*temporary) 变量了。本质上，返回值一定存在于此刻的局部变量中， 并且可以访问它，只是无法确定是哪些罢了。至于这个时候函数体内的其它局部变量，是不保证有效的。进入 return hook 的那一刻起，实际已经退出函数内部的运行环节， 返回值占用的局部变量空间以后的部分，都有可能因 hook 本身复用它们而改变。）

- **line hook:** 在解释器准备开始执行新的一行代码时， 或是跳转到这行代码中时（即使在同一行内跳转）被调用。 （这个事件仅仅在 Lua 执行一个 Lua 函数时发生。）
- **count hook:** 在解释器每执行 `count` 条指令后被调用。 （这个事件仅仅在 Lua 执行一个 Lua 函数时发生。）

钩子可以通过设置 `mask` 为零屏蔽。

`lua_setlocal`

```
const char *lua_setlocal (lua_State *L, lua_Debug *ar, int n);
```

设置给定活动记录中的局部变量的值。 参数 `ar` 与 `n` 和 `lua_getlocal` 中的一样 （参见 `lua_getlocal`）。 `lua_setlocal` 把栈顶的值赋给变量然后返回变量的名字。 它会将值从栈顶弹出。

当索引大于局部变量的个数时，返回 `NULL` （什么也不弹出）。

`lua_setupvalue`

```
const char *lua_setupvalue (lua_State *L, int funcindex, int n);
```

设置 closure 的 upvalue 的值。 它把栈顶的值弹出并赋于 upvalue 并返回 upvalue 的名字。 参数 `funcindex` 与 `n` 和 `lua_getupvalue` 中的一样 （参见 `lua_getupvalue`）。

当索引大于 upvalue 的个数时，返回 `NULL` （什么也不弹出）。

4 - The Auxiliary Library

The *auxiliary library* provides several convenient functions to interface C with Lua. While the basic API provides the primitive functions for all interactions between C and Lua, the auxiliary library provides higher-level functions for some common tasks.

All functions from the auxiliary library are defined in header file `lauxlib.h` and have a prefix `luaL_`.

All functions in the auxiliary library are built on top of the basic API, and so they provide nothing that cannot be done with this API.

Several functions in the auxiliary library are used to check C function arguments. Their names are always `luaL_check*` or `luaL_opt*`. All of these functions raise an error if the check is not satisfied. Because the error message is formatted for arguments (e.g., "`bad argument #1`"), you should not use these functions for other stack values.

4.1 - Functions and Types

Here we list all functions and types from the auxiliary library in alphabetical order.

---

luaL_addchar

```
void luaL_addchar (luaL_Buffer *B, char c);
```

Adds the character `c` to the buffer `B` (see `luaL_Buffer`).

---

luaL_addlstring

```
void luaL_addlstring (luaL_Buffer *B, const char *s, size_t l);
```

Adds the string pointed to by `s` with length `l` to the buffer `B` (see `luaL_Buffer`). The string may contain embedded zeros.

---

luaL_addsize

```
void luaL_addsize (luaL_Buffer *B, size_t n);
```

Adds to the buffer `B` (see `luaL_Buffer`) a string of length `n` previously copied to the buffer area (see `luaL_prepbuffer`).

---

luaL_addstring

```
void luaL_addstring (luaL_Buffer *B, const char *s);
```

Adds the zero-terminated string pointed to by `s` to the buffer `B` (see `luaL_Buffer`). The string may not contain embedded zeros.

---

luaL_addvalue

```
void luaL_addvalue (luaL_Buffer *B);
```

Adds the value at the top of the stack to the buffer `B` (see `luaL_Buffer`). Pops the value.

This is the only function on string buffers that can (and must) be called with an extra element on the stack, which is the value to be added to the buffer.

---

luaL_argcheck

```
void luaL_argcheck (lua_State *L,
                    int cond,
                    int narg,
                    const char *extramsg);
```

Checks whether `cond` is true. If not, raises an error with the following message, where `func` is retrieved from the call stack:

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

---

luaL_argerror

```
int luaL_argerror (lua_State *L, int narg, const char *extramsg);
```

Raises an error with the following message, where `func` is retrieved from the call stack:

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

This function never returns, but it is an idiom to use it in C functions as `return luaL_argerror(`*`args`*`)`.

---

luaL_Buffer

```
typedef struct luaL_Buffer luaL_Buffer;
```

Type for a *string buffer*.

A string buffer allows C code to build Lua strings piecemeal. Its pattern of use is as follows:

- First you declare a variable `b` of type `luaL_Buffer`.
- Then you initialize it with a call `luaL_buffinit(L, &b)`.
- Then you add string pieces to the buffer calling any of the `luaL_add*` functions.
- You finish by calling `luaL_pushresult(&b)`. This call leaves the final string on the top of the stack.

During its normal operation, a string buffer uses a variable number of stack slots. So, while using a buffer, you cannot assume that you know where the top of the stack is. You can use the stack between successive calls to buffer operations as long as that use is balanced; that is, when you call a buffer operation, the stack is at the same level it was immediately after the previous buffer operation. (The only exception to this rule is `luaL_addvalue`.) After calling `luaL_pushresult` the stack is back to its level when the buffer was initialized, plus the final string on its top.

---

luaL_buffinit

```
void luaL_buffinit (lua_State *L, luaL_Buffer *B);
```

Initializes a buffer `B`. This function does not allocate any space; the buffer must be declared as a variable (see `luaL_Buffer`).

---

luaL_callmeta

```
int luaL_callmeta (lua_State *L, int obj, const char *e);
```

Calls a metamethod.

If the object at index `obj` has a metatable and this metatable has a field `e`, this function calls this field and

passes the object as its only argument. In this case this function returns 1 and pushes onto the stack the value returned by the call. If there is no metatable or no metamethod, this function returns 0 (without pushing any value on the stack).

---

luaL_checkany

```
void luaL_checkany (lua_State *L, int narg);
```

Checks whether the function has an argument of any type (including **nil**) at position `narg`.

---

luaL_checkint

```
int luaL_checkint (lua_State *L, int narg);
```

Checks whether the function argument `narg` is a number and returns this number cast to an `int`.

---

luaL_checkinteger

```
lua_Integer luaL_checkinteger (lua_State *L, int narg);
```

Checks whether the function argument `narg` is a number and returns this number cast to a `lua_Integer`.

---

luaL_checklong

```
long luaL_checklong (lua_State *L, int narg);
```

Checks whether the function argument `narg` is a number and returns this number cast to a `long`.

---

luaL_checklstring

```
const char *luaL_checklstring (lua_State *L, int narg, size_t *l);
```

Checks whether the function argument `narg` is a string and returns this string; if `l` is not `NULL` fills `*l` with the string's length.

---

luaL_checknumber

```
lua_Number luaL_checknumber (lua_State *L, int narg);
```

Checks whether the function argument `narg` is a number and returns this number.

---

luaL_checkoption

```
int luaL_checkoption (lua_State *L,
                      int narg,
```

```
                        const char *def,
                        const char *const lst[]);
```

Checks whether the function argument `narg` is a string and searches for this string in the array `lst` (which must be NULL-terminated). Returns the index in the array where the string was found. Raises an error if the argument is not a string or if the string cannot be found.

If `def` is not `NULL`, the function uses `def` as a default value when there is no argument `narg` or if this argument is **nil**.

This is a useful function for mapping strings to C enums. (The usual convention in Lua libraries is to use strings instead of numbers to select options.)

---

luaL_checkstack

```
void luaL_checkstack (lua_State *L, int sz, const char *msg);
```

Grows the stack size to `top + sz` elements, raising an error if the stack cannot grow to that size. `msg` is an additional text to go into the error message.

---

luaL_checkstring

```
const char *luaL_checkstring (lua_State *L, int narg);
```

Checks whether the function argument `narg` is a string and returns this string.

---

luaL_checktype

```
void luaL_checktype (lua_State *L, int narg, int t);
```

Checks whether the function argument `narg` has type `t`.

---

luaL_checkudata

```
void *luaL_checkudata (lua_State *L, int narg, const char *tname);
```

Checks whether the function argument `narg` is a userdata of the type `tname` (see `luaL_newmetatable`).

---

luaL_dofile

```
int luaL_dofile (lua_State *L, const char *filename);
```

Loads and runs the given file. It is defined as the following macro:

```
     (luaL_loadfile(L, filename) || lua_pcall(L, 0, LUA_MULTRET, 0))
```

It returns 0 if there are no errors or 1 in case of errors.

---

luaL_dostring

```
int luaL_dostring (lua_State *L, const char *str);
```

Loads and runs the given string. It is defined as the following macro:

```
(luaL_loadstring(L, str) || lua_pcall(L, 0, LUA_MULTRET, 0))
```

It returns 0 if there are no errors or 1 in case of errors.

---

luaL_error

```
int luaL_error (lua_State *L, const char *fmt, ...);
```

Raises an error. The error message format is given by `fmt` plus any extra arguments, following the same rules of `lua_pushfstring`. It also adds at the beginning of the message the file name and the line number where the error occurred, if this information is available.

This function never returns, but it is an idiom to use it in C functions as `return luaL_error(`*`args`*`)`.

---

luaL_getmetafield

```
int luaL_getmetafield (lua_State *L, int obj, const char *e);
```

Pushes onto the stack the field `e` from the metatable of the object at index `obj`. If the object does not have a metatable, or if the metatable does not have this field, returns 0 and pushes nothing.

---

luaL_getmetatable

```
void luaL_getmetatable (lua_State *L, const char *tname);
```

Pushes onto the stack the metatable associated with name `tname` in the registry (see `luaL_newmetatable`).

---

luaL_gsub

```
const char *luaL_gsub (lua_State *L,
                       const char *s,
                       const char *p,
                       const char *r);
```

Creates a copy of string `s` by replacing any occurrence of the string `p` with the string `r`. Pushes the resulting string on the stack and returns it.

---

luaL_loadbuffer

```
int luaL_loadbuffer (lua_State *L,
                     const char *buff,
                     size_t sz,
```

```
                        const char *name);
```

Loads a buffer as a Lua chunk. This function uses lua_load to load the chunk in the buffer pointed to by buff with size sz.

This function returns the same results as lua_load. name is the chunk name, used for debug information and error messages.

---

luaL_loadfile

```
int luaL_loadfile (lua_State *L, const char *filename);
```

Loads a file as a Lua chunk. This function uses lua_load to load the chunk in the file named filename. If filename is NULL, then it loads from the standard input. The first line in the file is ignored if it starts with a #.

This function returns the same results as lua_load, but it has an extra error code LUA_ERRFILE if it cannot open/read the file.

As lua_load, this function only loads the chunk; it does not run it.

---

luaL_loadstring

```
int luaL_loadstring (lua_State *L, const char *s);
```

Loads a string as a Lua chunk. This function uses lua_load to load the chunk in the zero-terminated string s.

This function returns the same results as lua_load.

Also as lua_load, this function only loads the chunk; it does not run it.

---

luaL_newmetatable

```
int luaL_newmetatable (lua_State *L, const char *tname);
```

If the registry already has the key tname, returns 0. Otherwise, creates a new table to be used as a metatable for userdata, adds it to the registry with key tname, and returns 1.

In both cases pushes onto the stack the final value associated with tname in the registry.

---

luaL_newstate

```
lua_State *luaL_newstate (void);
```

Creates a new Lua state. It calls lua_newstate with an allocator based on the standard C realloc function and then sets a panic function (see lua_atpanic) that prints an error message to the standard error output in case of fatal errors.

Returns the new state, or `NULL` if there is a memory allocation error.

---

luaL_openlibs

```
void luaL_openlibs (lua_State *L);
```

Opens all standard Lua libraries into the given state.

---

luaL_optint

```
int luaL_optint (lua_State *L, int narg, int d);
```

If the function argument `narg` is a number, returns this number cast to an `int`. If this argument is absent or is **nil**, returns `d`. Otherwise, raises an error.

---

luaL_optinteger

```
lua_Integer luaL_optinteger (lua_State *L,
                             int narg,
                             lua_Integer d);
```

If the function argument `narg` is a number, returns this number cast to a `lua_Integer`. If this argument is absent or is **nil**, returns `d`. Otherwise, raises an error.

---

luaL_optlong

```
long luaL_optlong (lua_State *L, int narg, long d);
```

If the function argument `narg` is a number, returns this number cast to a `long`. If this argument is absent or is **nil**, returns `d`. Otherwise, raises an error.

---

luaL_optlstring

```
const char *luaL_optlstring (lua_State *L,
                             int narg,
                             const char *d,
                             size_t *l);
```

If the function argument `narg` is a string, returns this string. If this argument is absent or is **nil**, returns `d`. Otherwise, raises an error.

If `l` is not `NULL`, fills the position `*l` with the results's length.

---

luaL_optnumber

```
lua_Number luaL_optnumber (lua_State *L, int narg, lua_Number d);
```

If the function argument `narg` is a number, returns this number. If this argument is absent or is **nil**, returns `d`. Otherwise, raises an error.

---

luaL_optstring

```
const char *luaL_optstring (lua_State *L,
                            int narg,
                            const char *d);
```

If the function argument `narg` is a string, returns this string. If this argument is absent or is **nil**, returns `d`. Otherwise, raises an error.

---

luaL_prepbuffer

```
char *luaL_prepbuffer (luaL_Buffer *B);
```

Returns an address to a space of size `LUAL_BUFFERSIZE` where you can copy a string to be added to buffer `B` (see `luaL_Buffer`). After copying the string into this space you must call `luaL_addsize` with the size of the string to actually add it to the buffer.

---

luaL_pushresult

```
void luaL_pushresult (luaL_Buffer *B);
```

Finishes the use of buffer `B` leaving the final string on the top of the stack.

---

luaL_ref

```
int luaL_ref (lua_State *L, int t);
```

Creates and returns a *reference*, in the table at index `t`, for the object at the top of the stack (and pops the object).

A reference is a unique integer key. As long as you do not manually add integer keys into table `t`, `luaL_ref` ensures the uniqueness of the key it returns. You can retrieve an object referred by reference `r` by calling `lua_rawgeti(L, t, r)`. Function `luaL_unref` frees a reference and its associated object.

If the object at the top of the stack is **nil**, `luaL_ref` returns the constant `LUA_REFNIL`. The constant `LUA_NOREF` is guaranteed to be different from any reference returned by `luaL_ref`.

---

luaL_Reg

```
typedef struct luaL_Reg {
  const char *name;
  lua_CFunction func;
} luaL_Reg;
```

Type for arrays of functions to be registered by `luaL_register`. `name` is the function name and `func` is a

pointer to the function. Any array of luaL_Reg must end with an sentinel entry in which both name and func are NULL.

---

luaL_register

```
void luaL_register (lua_State *L,
                    const char *libname,
                    const luaL_Reg *l);
```

Opens a library.

When called with libname equal to NULL, it simply registers all functions in the list l (see luaL_Reg) into the table on the top of the stack.

When called with a non-null libname, luaL_register creates a new table t, sets it as the value of the global variable libname, sets it as the value of package.loaded[libname], and registers on it all functions in the list l. If there is a table in package.loaded[libname] or in variable libname, reuses this table instead of creating a new one.

In any case the function leaves the table on the top of the stack.

---

luaL_typename

```
const char *luaL_typename (lua_State *L, int idx);
```

Returns the name of the type of the value at index idx.

---

luaL_typerror

```
int luaL_typerror (lua_State *L, int narg, const char *tname);
```

Generates an error with a message like the following:

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

where *location* is produced by luaL_where, *func* is the name of the current function, and *rt* is the type name of the actual argument.

---

luaL_unref

```
void luaL_unref (lua_State *L, int t, int ref);
```

Releases reference ref from the table at index t (see luaL_ref). The entry is removed from the table, so that the referred object can be collected. The reference ref is also freed to be used again.

If ref is LUA_NOREF or LUA_REFNIL, luaL_unref does nothing.

---

luaL_where

```
void luaL_where (lua_State *L, int lvl);
```

Pushes onto the stack a string identifying the current position of the control at level `lvl` in the call stack. Typically this string has the following format:

```
chunkname:currentline:
```

Level 0 is the running function, level 1 is the function that called the running function, etc.

This function is used to build a prefix for error messages.

5 - Standard Libraries

The standard Lua libraries provide useful functions that are implemented directly through the C API. Some of these functions provide essential services to the language (e.g., `type` and `getmetatable`); others provide access to "outside" services (e.g., I/O); and others could be implemented in Lua itself, but are quite useful or have critical performance requirements that deserve an implementation in C (e.g., `sort`).

All libraries are implemented through the official C API and are provided as separate C modules. Currently, Lua has the following standard libraries:

- basic library;
- package library;
- string manipulation;
- table manipulation;
- mathematical functions (sin, log, etc.);
- input and output;
- operating system facilities;
- debug facilities.

Except for the basic and package libraries, each library provides all its functions as fields of a global table or as methods of its objects.

To have access to these libraries, the C host program should call the `luaL_openlibs` function, which opens all standard libraries. Alternatively, it can open them individually by calling `luaopen_base` (for the basic library), `luaopen_package` (for the package library), `luaopen_string` (for the string library), `luaopen_table` (for the table library), `luaopen_math` (for the mathematical library), `luaopen_io` (for the I/O and the Operating System libraries), and `luaopen_debug` (for the debug library). These functions are declared in `lualib.h` and should not be called directly: you must call them like any other Lua C function, e.g., by using `lua_call`.

5.1 - Basic Functions

The basic library provides some core functions to Lua. If you do not include this library in your application, you should check carefully whether you need to provide implementations for some of its facilities.

`assert (v [, message])`

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

`collectgarbage (opt [, arg])`

This function is a generic interface to the garbage collector. It performs different functions according to its first argument, `opt`:

- **"stop":** stops the garbage collector.
- **"restart":** restarts the garbage collector.
- **"collect":** performs a full garbage-collection cycle.
- **"count":** returns the total memory in use by Lua (in Kbytes).
- **"step":** performs a garbage-collection step. The step "size" is controlled by `arg` (larger values mean more steps) in a non-specified way. If you want to control the step size you must experimentally tune the value of `arg`. Returns **true** if the step finished a collection cycle.
- **"setpause":** sets `arg`/100 as the new value for the *pause* of the collector (see §2.10).
- **"setstepmul":** sets `arg`/100 as the new value for the *step multiplier* of the collector (see §2.10).

`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.

Usually, `error` adds some information about the error position at the beginning of the message. The `level` argument specifies how to get the error position. With level 1 (the default), the error position is where the `error` function was called. Level 2 points the error to where the function that called `error` was called; and so on. Passing a level 0 avoids the addition of error position information to the message.

`_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` 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)

If `object` does not have a metatable, returns **nil**. Otherwise, if the object's metatable has a `"__metatable"` field, returns the associated value. Otherwise, returns the metatable of the given object.

---

ipairs (t)

Returns three values: an iterator function, the table `t`, and 0, so that the construction

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

will iterate over the pairs `(1,t[1])`, `(2,t[2])`, ···, up to the first integer key absent from the table.

---

load (func [, chunkname])

Loads a chunk using function `func` to get its pieces. Each call to `func` must return a string that concatenates with previous results. A return of **nil** (or no value) signals the end of the chunk.

If there are no errors, returns the compiled chunk as a function; otherwise, returns **nil** plus the error message. The environment of the returned function is the global environment.

`chunkname` is used as the chunk name for error messages and debug information.

---

loadfile ([filename])

Similar to `load`, but gets the chunk from file `filename` or from the standard input, if no file name is given.

---

loadstring (string [, chunkname])

Similar to `load`, but gets the chunk from the given string.

To load and run a given string, use the idiom

```
assert(loadstring(s))()
```

---

next (table [, index])

Allows a program to traverse all fields of a table. Its first argument is a table and its second argument is an index in this table. `next` returns the next index of the table and its associated value. When called with **nil** as its second argument, `next` returns an initial index and its associated value. When called with the last index, or with **nil** in an empty table, `next` returns **nil**. If the second argument is absent, then it is interpreted as **nil**. In particular, you can use `next(t)` to check whether a table is empty.

The order in which the indices are enumerated is not specified, *even for numeric indices*. (To traverse a table in numeric order, use a numerical **for** or the `ipairs` function.)

The behavior of `next` is *undefined* if, during the traversal, you assign any value to a non-existent field in

the table. You may however modify existing fields. In particular, you may clear existing fields.

---

pairs (t)

Returns three values: the `next` function, the table `t`, and **nil**, so that the construction

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

will iterate over all key–value pairs of table `t`.

See function `next` for the caveats of modifying the table during its traversal.

---

pcall (f, arg1, ···)

Calls function `f` with the given arguments in *protected mode*. This means that any error inside `f` is not propagated; instead, `pcall` catches the error and returns a status code. Its first result is the status code (a boolean), which is true if the call succeeds without errors. In such case, `pcall` also returns all results from the call, after this first result. In case of any error, `pcall` returns **false** plus the error message.

---

print (···)

Receives any number of arguments, and prints their values to `stdout`, using the `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`.

---

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 **nil**, and `value` any Lua value.

This function returns `table`.

---

select (index, ···)

If `index` is a number, returns all arguments after argument number `index`. Otherwise, `index` must be the

string `"#"`, and `select` returns the total number of extra arguments it received.

`setfenv (f, table)`

Sets the environment to be used by the given function. `f` can be a Lua function or a number that specifies the function at that stack level: Level 1 is the function calling `setfenv`. `setfenv` returns the given function.

As a special case, when `f` is 0 `setfenv` changes the environment of the running thread. In this case, `setfenv` returns no values.

`setmetatable (table, metatable)`

Sets the metatable for the given table. (You cannot change the metatable of other types from Lua, only from C.) If `metatable` is **nil**, removes the metatable of the given table. If the original metatable has a `"__metatable"` field, raises an error.

This function returns `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 **nil**.

An optional argument specifies the base to interpret the numeral. The base may be any integer between 2 and 36, inclusive. In bases above 10, the letter '`A`' (in either upper or lower case) represents 10, '`B`' represents 11, and so forth, with '`Z`' representing 35. In base 10 (the default), the number may have a decimal part, as well as an optional exponent part (see §2.1). In other bases, only unsigned integers are accepted.

`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`.

If the metatable of `e` has a `"__tostring"` field, then `tostring` calls the corresponding value with `e` as argument, and uses the result of the call as its result.

`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 **nil**), `"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]
```

except that the above code can be written only for a fixed number of elements. By default, `i` is 1 and `j` is the length of the list, as defined by the length operator (see §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)`

This function is similar to `pcall`, except that you can set a new error handler.

`xpcall` calls function `f` in protected mode, using `err` as the error handler. Any error inside `f` is not propagated; instead, `xpcall` catches the error, calls the `err` function with the original error object, and returns a status code. Its first result is the status code (a boolean), which is true if the call succeeds without errors. In this case, `xpcall` also returns all results from the call, after this first result. In case of any error, `xpcall` returns **false** plus the result from `err`.

5.2 - Coroutine Manipulation

The operations related to coroutines comprise a sub-library of the basic library and come inside the table `coroutine`. See §2.11 for a general description of coroutines.

`coroutine.create (f)`

Creates a new coroutine, with body `f`. `f` must be a Lua function. Returns this new coroutine, an object with type `"thread"`.

`coroutine.resume (co [, val1, ···])`

Starts or continues the execution of coroutine `co`. The first time you resume a coroutine, it starts running its body. The values `val1`, ··· are passed as the arguments to the body function. If the coroutine has yielded, `resume` restarts it; the values `val1`, ··· are passed as the results from the yield.

If the coroutine runs without any errors, `resume` returns **true** plus any values passed to `yield` (if the coroutine yields) or any values returned by the body function (if the coroutine terminates). If there is any error, `resume` returns **false** plus the error message.

`coroutine.running ()`

Returns the running coroutine, or **nil** when called by the main thread.

---

coroutine.status (co)

Returns the status of coroutine `co`, as a string: `"running"`, if the coroutine is running (that is, it called `status`); `"suspended"`, if the coroutine is suspended in a call to `yield`, or if it has not started running yet; `"normal"` if the coroutine is active but not running (that is, it has resumed another coroutine); and `"dead"` if the coroutine has finished its body function, or if it has stopped with an error.

---

coroutine.wrap (f)

Creates a new coroutine, with body `f`. `f` must be a Lua function. Returns a function that resumes the coroutine each time it is called. Any arguments passed to the function behave as the extra arguments to `resume`. Returns the same values returned by `resume`, except the first boolean. In case of error, propagates the error.

---

coroutine.yield (···)

Suspends the execution of the calling coroutine. The coroutine cannot be running a C function, a metamethod, or an iterator. Any arguments to `yield` are passed as extra results to `resume`.

5.3 - Modules

The package library provides basic facilities for loading and building modules in Lua. It exports two of its functions directly in the global environment: `require` and `module`. Everything else is exported in a table `package`.

---

module (name [, ···])

Creates a module. If there is a table in `package.loaded[name]`, this table is the module. Otherwise, if there is a global table `t` with the given name, this table is the module. Otherwise creates a new table `t` and sets it as the value of the global `name` and the value of `package.loaded[name]`. This function also initializes `t._NAME` with the given name, `t._M` with the module (`t` itself), and `t._PACKAGE` with the package name (the full module name minus last component; see below). Finally, `module` sets `t` as the new environment of the current function and the new value of `package.loaded[name]`, so that `require` returns `t`.

If `name` is a compound name (that is, one with components separated by dots), `module` creates (or reuses, if they already exist) tables for each component. For instance, if `name` is `a.b.c`, then `module` stores the module table in field `c` of field `b` of global `a`.

This function may receive optional *options* after the module name, where each option is a function to be applied over the module.

---

require (modname)

Loads the given module. The function starts by looking into the `package.loaded` table to determine

whether `modname` is already loaded. If it is, then `require` returns the value stored at `package.loaded[modname]`. Otherwise, it tries to find a *loader* for the module.

To find a loader, first `require` queries `package.preload[modname]`. If it has a value, this value (which should be a function) is the loader. Otherwise `require` searches for a Lua loader using the path stored in `package.path`. If that also fails, it searches for a C loader using the path stored in `package.cpath`. If that also fails, it tries an *all-in-one* loader (see below).

When loading a C library, `require` first uses a dynamic link facility to link the application with the library. Then it tries to find a C function inside this library to be used as the loader. The name of this C function is the string "`luaopen_`" concatenated with a copy of the module name where each dot is replaced by an underscore. Moreover, if the module name has a hyphen, its prefix up to (and including) the first hyphen is removed. For instance, if the module name is `a.v1-b.c`, the function name will be `luaopen_b_c`.

If `require` finds neither a Lua library nor a C library for a module, it calls the *all-in-one loader*. This loader searches the C path for a library for the root name of the given module. For instance, when requiring `a.b.c`, it will search for a C library for `a`. If found, it looks into it for an open function for the submodule; in our example, that would be `luaopen_a_b_c`. With this facility, a package can pack several C submodules into one single library, with each submodule keeping its original open function.

Once a loader is found, `require` calls the loader with a single argument, `modname`. If the loader returns any value, `require` assigns the returned value to `package.loaded[modname]`. If the loader returns no value and has not assigned any value to `package.loaded[modname]`, then `require` assigns **true** to this entry. In any case, `require` returns the final value of `package.loaded[modname]`.

If there is any error loading or running the module, or if it cannot find any loader for the module, then `require` signals an error.

---

package.cpath

The path used by `require` to search for a C loader.

Lua initializes the C path `package.cpath` in the same way it initializes the Lua path `package.path`, using the environment variable `LUA_CPATH` (plus another default path defined in `luaconf.h`).

---

package.loaded

A table used by `require` to control which modules are already loaded. When you require a module `modname` and `package.loaded[modname]` is not false, `require` simply returns the value stored there.

---

package.loadlib (libname, funcname)

Dynamically links the host program with the C library `libname`. Inside this library, looks for a function `funcname` and returns this function as a C function. (So, `funcname` must follow the protocol (see `lua_CFunction`)).

This is a low-level function. It completely bypasses the package and module system. Unlike `require`, it

does not perform any path searching and does not automatically adds extensions. `libname` must be the complete file name of the C library, including if necessary a path and extension. `funcname` must be the exact name exported by the C library (which may depend on the C compiler and linker used).

This function is not supported by ANSI C. As such, it is only available on some platforms (Windows, Linux, Mac OS X, Solaris, BSD, plus other Unix systems that support the `dlfcn` standard).

`package.path`

The path used by `require` to search for a Lua loader.

At start-up, Lua initializes this variable with the value of the environment variable `LUA_PATH` or with a default path defined in `luaconf.h`, if the environment variable is not defined. Any `";;"` in the value of the environment variable is replaced by the default path.

A path is a sequence of *templates* separated by semicolons. For each template, `require` will change each interrogation mark in the template by `filename`, which is `modname` with each dot replaced by a "directory separator" (such as `"/"` in Unix); then it will try to load the resulting file name. So, for instance, if the Lua path is

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

the search for a Lua loader for module `foo` will try to load the files `./foo.lua`, `./foo.lc`, and `/usr/local/foo/init.lua`, in that order.

`package.preload`

A table to store loaders for specific modules (see `require`).

`package.seeall (module)`

Sets a metatable for `module` with its `__index` field referring to the global environment, so that this module inherits values from the global environment. To be used as an option to function `module`.

5.4 - String Manipulation

This library provides generic functions for string manipulation, such as finding and extracting substrings, and pattern matching. When indexing a string in Lua, the first character is at position 1 (not at 0, as in C). Indices are allowed to be negative and are interpreted as indexing backwards, from the end of the string. Thus, the last character is at position -1, and so on.

The string library provides all its functions inside the table `string`. It also sets a metatable for strings where the `__index` field points to the `string` table. Therefore, you can use the string functions in object-oriented style. For instance, `string.byte(s, i)` can be written as `s:byte(i)`.

`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`.

Note that numerical codes are not necessarily portable across platforms.

---

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.

Note that numerical codes are not necessarily portable across platforms.

---

string.dump (function)

Returns a string containing a binary representation of the given function, so that a later `loadstring` on this string returns a copy of the function. `function` must be a Lua function without upvalues.

---

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 **nil**. A third, optional numerical argument `init` specifies where to start the search; its default value is 1 and may be negative. A value of **true** 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 `pattern` being considered "magic". Note that if `plain` is given, then `init` must be given as well.

If the pattern has captures, then in a successful match the captured values are also returned, after the two indices.

---

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')
```

will produce the string:

```
"a string with \"quotes\" and \
 new line"
```

The options `c`, `d`, `E`, `e`, `f`, `g`, `G`, `i`, `o`, `u`, `X`, and `x` all expect a number as argument, whereas `q` and `s` expect a string.

This function does not accept string values containing embedded zeros.

---

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.

As an example, the following loop

```
s = "hello world from Lua"
for w in string.gmatch(s, "%a+") do
  print(w)
end
```

will iterate over all the words from string `s`, printing one per line. The next example collects all pairs `key=value` from the given string into a table:

```
t = {}
s = "from=world, to=Lua"
for k, v in string.gmatch(s, "(%w+)=(%w+)") do
  t[k] = v
end
```

---

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

Returns a copy of `s` in which all occurrences of the `pattern` have been replaced by a replacement string specified by `repl`, which may be a string, a table, or a function. `gsub` also returns, as its second value, the total number of substitutions made.

If `repl` is a string, then its value is used for replacement. The character `%` works as an escape character: any sequence in `repl` of the form `%`*n*, with *n* between 1 and 9, stands for the value of the *n*-th captured substring (see below). The sequence `%0` stands for the whole match. The sequence `%%` stands for a single `%`.

If `repl` is a table, then the table is queried for every match, using the first capture as the key; if the pattern specifies no captures, then the whole match is used as the key.

If `repl` is a function, then this function is called every time a match occurs, with all captured substrings passed as arguments, in order; if the pattern specifies no captures, then the whole match is passed as a sole argument.

If the value returned by the table query or by the function call is a string or a number, then it is used as the replacement string; otherwise, if it is **false** or **nil**, then there is no replacement (that is, the original match is kept in the string).

The optional last parameter `n` limits the maximum number of substitutions to occur. For instance, when `n` is 1 only the first occurrence of `pattern` is replaced.

Here are some examples:

```
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 *match* of `pattern` in the string `s`. If it finds one, then `match` returns the captures from the pattern; otherwise it returns **nil**. 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 may 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` may 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 - Patterns

Character Class:

A *character class* is used to represent a set of characters. The following combinations are allowed in describing a character class:

- ***x*:** (where *x* is not one of the *magic characters* `^$()%.[]*+-?`) represents the character *x* itself.
- **`.`:** (a dot) represents all characters.
- **`%a`:** represents all letters.
- **`%c`:** represents all control characters.
- **`%d`:** represents all digits.
- **`%l`:** represents all lowercase letters.
- **`%p`:** represents all punctuation characters.
- **`%s`:** represents all space characters.
- **`%u`:** represents all uppercase letters.
- **`%w`:** represents all alphanumeric characters.
- **`%x`:** represents all hexadecimal digits.
- **`%z`:** represents the character with representation 0.
- **`%`*x*:** (where *x* is any non-alphanumeric character) represents the character *x*. This is the standard way to escape the magic characters. Any punctuation character (even the non magic) can be preceded by a '`%`' when used to represent itself in a pattern.
- **`[`*set*`]`:** represents the class which is the union of all characters in *set*. A range of characters may be specified by separating the end characters of the range with a '`-`'. All classes `%`*x* described above may also be used as components in *set*. All other characters in *set* represent themselves. For example, `[%w_]` (or `[_%w]`) represents all alphanumeric characters plus the underscore, `[0-7]` represents the octal digits, and `[0-7%l%-]` represents the octal digits plus the lowercase letters plus the '`-`' character.

  The interaction between ranges and classes is not defined. Therefore, patterns like `[%a-z]` or `[a-%%]` have no meaning.

- **`[^`*set*`]`:** represents the complement of *set*, where *set* is interpreted as above.

For all classes represented by single letters (`%a`, `%c`, etc.), the corresponding uppercase letter represents the complement of the class. For instance, `%S` represents all non-space characters.

The definitions of letter, space, and other character groups depend on the current locale. In particular, the

class `[a-z]` may not be equivalent to `%l`.

Pattern Item:

A *pattern item* may be

- a single character class, which matches any single character in the class;
- a single character class followed by '`*`', which matches 0 or more repetitions of characters in the class. These repetition items will always match the longest possible sequence;
- a single character class followed by '`+`', which matches 1 or more repetitions of characters in the class. These repetition items will always match the longest possible sequence;
- a single character class followed by '`-`', which also matches 0 or more repetitions of characters in the class. Unlike '`*`', these repetition items will always match the *shortest* possible sequence;
- a single character class followed by '`?`', which matches 0 or 1 occurrence of a character in the class;
- `%`*n*, for *n* between 1 and 9; such item matches a substring equal to the *n*-th captured string (see below);
- `%b`*xy*, where *x* and *y* are two distinct characters; such item matches strings that start with *x*, end with *y*, and where the *x* and *y* are *balanced*. This means that, if one reads the string from left to right, counting *+1* for an *x* and *-1* for a *y*, the ending *y* is the first *y* where the count reaches 0. For instance, the item `%b()` matches expressions with balanced parentheses.

Pattern:

A *pattern* is a sequence of pattern items. A '`^`' at the beginning of a pattern anchors the match at the beginning of the subject string. A '`$`' at the end of a pattern anchors the match at the end of the subject string. At other positions, '`^`' and '`$`' have no special meaning and represent themselves.

Captures:

A pattern may contain sub-patterns enclosed in parentheses; they describe *captures*. When a match succeeds, the substrings of the subject string that match captures are stored (*captured*) for future use. Captures are numbered according to their left parentheses. For instance, in the pattern `"(a*(.)%w(%s*))"`, the part of the string matching `"a*(.)%w(%s*)"` is stored as the first capture (and therefore has number 1); the character matching "`.`" is captured with number 2, and the part matching `"%s*"` has number 3.

As a special case, the empty capture `()` captures the current string position (a number). For instance, if we apply the pattern `"()aa()"` on the string `"flaaap"`, there will be two captures: 3 and 5.

A pattern cannot contain embedded zeros. Use `%z` instead.

5.5 - Table Manipulation

This library provides generic functions for table manipulation. It provides all its functions inside the table `table`.

Most functions in the table library assume that the table represents an array or a list. For these functions, when we talk about the "length" of a table we mean the result of the length operator.

---

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)

Inserts element `value` at position `pos` in `table`, shifting up other elements to open space, if necessary. The default value for `pos` is `n+1`, where `n` is the length of the table (see §2.5.5), so that a call `table.insert(t,x)` inserts `x` at the end of table `t`.

---

table.maxn (table)

Returns the largest positive numerical index of the given table, or zero if the table has no positive numerical indices. (To do its job this function does a linear traversal of the whole table.)

---

table.remove (table [, pos])

Removes from `table` the element at position `pos`, shifting down other elements to close the space, if necessary. Returns the value of the removed element. The default value for `pos` is `n`, where `n` is the length of the table, so that a call `table.remove(t)` removes the last element of table `t`.

---

table.sort (table [, comp])

Sorts table elements in a given order, *in-place*, 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.

The sort algorithm is not stable; that is, elements considered equal by the given order may have their relative positions changed by the sort.

5.6 - Mathematical Functions

This library is an interface to the standard C math library. It provides all its functions inside the table `math`.

---

math.abs (x)

Returns the absolute value of `x`.

---

math.acos (x)

Returns the arc cosine of `x` (in radians).

`math.asin (x)`

Returns the arc sine of `x` (in radians).

`math.atan (x)`

Returns the arc tangent of `x` (in radians).

`math.atan2 (x, y)`

Returns the arc tangent of `x/y` (in radians), but uses the signs of both parameters to find the quadrant of the result. (It also handles correctly the case of `y` being zero.)

`math.ceil (x)`

Returns the smallest integer larger than or equal to `x`.

`math.cos (x)`

Returns the cosine of `x` (assumed to be in radians).

`math.cosh (x)`

Returns the hyperbolic cosine of `x`.

`math.deg (x)`

Returns the angle `x` (given in radians) in degrees.

`math.exp (x)`

Returns the the value e^x.

`math.floor (x)`

Returns the largest integer smaller than or equal to `x`.

`math.fmod (x, y)`

Returns the remainder of the division of `x` by `y`.

`math.frexp (x)`

Returns `m` and `e` such that $x = m2^e$, `e` is an integer and the absolute value of `m` is in the range *[0.5, 1)* (or zero when `x` is zero).

`math.huge`

The value `HUGE_VAL`, a value larger than or equal to any other numerical value.

`math.ldexp (m, e)`

Returns $m2^e$ (`e` should be an integer).

`math.log (x)`

Returns the natural logarithm of `x`.

`math.log10 (x)`

Returns the base-10 logarithm of `x`.

`math.max (x, ···)`

Returns the maximum value among its arguments.

`math.min (x, ···)`

Returns the minimum value among its arguments.

`math.modf (x)`

Returns two numbers, the integral part of `x` and the fractional part of `x`.

`math.pi`

The value of *pi*.

`math.pow (x, y)`

Returns x^y. (You can also use the expression `x^y` to compute this value.)

`math.rad (x)`

Returns the angle `x` (given in degrees) in radians.

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

This function is an interface to the simple pseudo-random generator function `rand` provided by ANSI C. (No guarantees can be given for its statistical properties.)

When called without arguments, returns a pseudo-random real number in the range *[0,1)*. When called with a number `m`, `math.random` returns a pseudo-random integer in the range *[1, m]*. When called with two numbers `m` and `n`, `math.random` returns a pseudo-random integer in the range *[m, n]*.

`math.randomseed (x)`

Sets `x` as the "seed" for the pseudo-random generator: equal seeds produce equal sequences of numbers.

`math.sin (x)`

Returns the sine of `x` (assumed to be in radians).

`math.sinh (x)`

Returns the hyperbolic sine of `x`.

`math.sqrt (x)`

Returns the square root of `x`. (You can also use the expression `x^0.5` to compute this value.)

`math.tan (x)`

Returns the tangent of `x` (assumed to be in radians).

`math.tanh (x)`

Returns the hyperbolic tangent of `x`.

5.7 - Input and Output Facilities

The I/O library provides two different styles for file manipulation. The first one uses implicit file descriptors; that is, there are operations to set a default input file and a default output file, and all input/output operations are over these default files. The second style uses explicit file descriptors.

When using implicit file descriptors, all operations are supplied by table `io`. When using explicit file descriptors, the operation `io.open` returns a file descriptor and then all operations are supplied as methods of the file descriptor.

The table `io` also provides three predefined file descriptors with their usual meanings from C: `io.stdin`, `io.stdout`, and `io.stderr`.

Unless otherwise stated, all I/O functions return **nil** on failure (plus an error message as a second result) and some value different from **nil** on success.

`io.close ([file])`

Equivalent to `file:close()`. Without a `file`, closes the default output file.

`io.flush ()`

Equivalent to `file:flush` over the default output file.

`io.input ([file])`

When called with a file name, it opens the named file (in text mode), and sets its handle as the default input file. When called with a file handle, it simply sets this file handle as the default input file. When called without parameters, it returns the current default input file.

In case of errors this function raises the error, instead of returning an error code.

`io.lines ([filename])`

Opens the given file name in read mode and returns an iterator function that, each time it is called, returns a new line from the file. Therefore, the construction

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

will iterate over all lines of the file. When the iterator function detects the end of file, it returns **nil** (to finish the loop) and automatically closes the file.

The call `io.lines()` (with no file name) is equivalent to `io.input():lines()`; that is, it iterates over the lines of the default input file. In this case it does not close the file when the loop ends.

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

This function opens a file, in the mode specified in the string `mode`. It returns a new file handle, or, in case

of errors, **nil** plus an error message.

The `mode` string can be any of the following:

- "**r**": read mode (the default);
- "**w**": write mode;
- "**a**": append mode;
- "**r+**": update mode, all previous data is preserved;
- "**w+**": update mode, all previous data is erased;
- "**a+**": append update mode, previous data is preserved, writing is only allowed at the end of file.

The `mode` string may also have a 'b' at the end, which is needed in some systems to open the file in binary mode. This string is exactly what is used in the standard C function `fopen`.

---

io.output ([file])

Similar to `io.input`, but operates over the default output file.

---

io.popen (prog [, mode])

Starts program `prog` in a separated process and returns a file handle that you can use to read data from this program (if `mode` is `"r"`, the default) or to write data to this program (if `mode` is `"w"`).

This function is system dependent and is not available on all platforms.

---

io.read (···)

Equivalent to `io.input():read`.

---

io.tmpfile ()

Returns a handle for a temporary file. This file is opened in update mode and it is automatically removed when the program ends.

---

io.type (obj)

Checks whether `obj` is a valid file handle. Returns the string `"file"` if `obj` is an open file handle, `"closed file"` if `obj` is a closed file handle, or **nil** if `obj` is not a file handle.

---

io.write (···)

Equivalent to `io.output():write`.

---

file:close ()

Closes `file`. Note that files are automatically closed when their handles are garbage collected, but that takes an unpredictable amount of time to happen.

`file:flush ()`

Saves any written data to `file`.

`file:lines ()`

Returns an iterator function that, each time it is called, returns a new line from the file. Therefore, the construction

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

will iterate over all lines of the file. (Unlike `io.lines`, this function does not close the file when the loop ends.)

`file:read (···)`

Reads the file `file`, according to the given formats, which specify what to read. For each format, the function returns a string (or a number) with the characters read, or **nil** if it cannot read data with the specified format. When called without formats, it uses a default format that reads the entire next line (see below).

The available formats are

- **"*n":** reads a number; this is the only format that returns a number instead of a string.
- **"*a":** reads the whole file, starting at the current position. On end of file, it returns the empty string.
- **"*l":** reads the next line (skipping the end of line), returning **nil** on end of file. This is the default format.
- ***number*:** reads a string with up to this number of characters, returning **nil** on end of file. If number is zero, it reads nothing and returns an empty string, or **nil** on end of file.

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

Sets and gets the file position, measured from the beginning of the file, to the position given by `offset` plus a base specified by the string `whence`, as follows:

- **"set":** base is position 0 (beginning of the file);
- **"cur":** base is current position;
- **"end":** base is end of file;

In case of success, function `seek` returns the final file position, measured in bytes from the beginning of the file. If this function fails, it returns **nil**, plus a string describing the error.

The default value for `whence` is `"cur"`, and for `offset` is 0. Therefore, the call `file:seek()` returns the current file position, without changing it; the call `file:seek("set")` sets the position to the beginning of

the file (and returns 0); and the call `file:seek("end")` sets the position to the end of the file, and returns its size.

`file:setvbuf (mode [, size])`

Sets the buffering mode for an output file. There are three available modes:

- **"no":** no buffering; the result of any output operation appears immediately.
- **"full":** full buffering; output operation is performed only when the buffer is full (or when you explicitly `flush` the file (see `io.flush`)).
- **"line":** line buffering; output is buffered until a newline is output or there is any input from some special files (such as a terminal device).

For the last two cases, `sizes` specifies the size of the buffer, in bytes. The default is an appropriate size.

`file:write (···)`

Writes the value of each of its arguments to the `file`. The arguments must be strings or numbers. To write other values, use `tostring` or `string.format` before `write`.

5.8 - Operating System Facilities

This library is implemented through table `os`.

`os.clock ()`

Returns an approximation of the amount in seconds of CPU time used by the program.

`os.date ([format [, time]])`

Returns a string or a table containing date and time, formatted according to the given string `format`.

If the `time` argument is present, this is the time to be formatted (see the `os.time` function for a description of this value). Otherwise, `date` formats the current time.

If `format` starts with `'!'`, then the date is formatted in Coordinated Universal Time. After this optional character, if `format` is the string `"*t"`, then `date` returns a table with the following fields: `year` (four digits), `month` (1--12), `day` (1--31), `hour` (0--23), `min` (0--59), `sec` (0--61), `wday` (weekday, Sunday is 1), `yday` (day of the year), and `isdst` (daylight saving flag, a boolean).

If `format` is not `"*t"`, then `date` returns the date as a string, formatted according to the same rules as the C function `strftime`.

When called without arguments, `date` returns a reasonable date and time representation that depends on the host system and on the current locale (that is, `os.date()` is equivalent to `os.date("%c")`).

---

os.difftime (t2, t1)

Returns the number of seconds from time `t1` to time `t2`. In POSIX, Windows, and some other systems, this value is exactly `t2-t1`.

---

os.execute ([command])

This function is equivalent to the C function `system`. It passes `command` to be executed by an operating system shell. It returns a status code, which is system-dependent. If `command` is absent, then it returns nonzero if a shell is available and zero otherwise.

---

os.exit ([code])

Calls the C function `exit`, with an optional `code`, to terminate the host program. The default value for `code` is the success code.

---

os.getenv (varname)

Returns the value of the process environment variable `varname`, or **nil** if the variable is not defined.

---

os.remove (filename)

Deletes the file or directory with the given name. Directories must be empty to be removed. If this function fails, it returns **nil**, plus a string describing the error.

---

os.rename (oldname, newname)

Renames file or directory named `oldname` to `newname`. If this function fails, it returns **nil**, plus a string describing the error.

---

os.setlocale (locale [, category])

Sets the current locale of the program. `locale` is a string specifying a locale; `category` is an optional string describing which category to change: `"all"`, `"collate"`, `"ctype"`, `"monetary"`, `"numeric"`, or `"time"`; the default category is `"all"`. The function returns the name of the new locale, or **nil** if the request cannot be honored.

When called with **nil** as the first argument, this function only returns the name of the current locale for the given category.

---

os.time ([table])

Returns the current time when called without arguments, or a time representing the date and time specified

by the given table. This table must have fields `year`, `month`, and `day`, and may have fields `hour`, `min`, `sec`, and `isdst` (for a description of these fields, see the `os.date` function).

The returned value is a number, whose meaning depends on your system. In POSIX, Windows, and some other systems, this number counts the number of seconds since some given start time (the "epoch"). In other systems, the meaning is not specified, and the number returned by `time` can be used only as an argument to `date` and `difftime`.

`os.tmpname ()`

Returns a string with a file name that can be used for a temporary file. The file must be explicitly opened before its use and explicitly removed when no longer needed.

5.9 - The Debug Library

This library provides the functionality of the debug interface to Lua programs. You should exert care when using this library. The functions provided here should be used exclusively for debugging and similar tasks, such as profiling. Please resist the temptation to use them as a usual programming tool: they can be very slow. Moreover, several of its functions violate some assumptions about Lua code (e.g., that variables local to a function cannot be accessed from outside or that userdata metatables cannot be changed by Lua code) and therefore can compromise otherwise secure code.

All functions in this library are provided inside the `debug` table. All functions that operate over a thread have an optional first argument which is the thread to operate over. The default is always the current thread.

`debug.debug ()`

Enters an interactive mode with the user, running each string that the user enters. Using simple commands and other debug facilities, the user can inspect global and local variables, change their values, evaluate expressions, and so on. A line containing only the word `cont` finishes this function, so that the caller continues its execution.

Note that commands for `debug.debug` are not lexically nested within any function, and so have no direct access to local variables.

`debug.getfenv (o)`

Returns the environment of object `o`.

`debug.gethook ([thread])`

Returns the current hook settings of the thread, as three values: the current hook function, the current hook mask, and the current hook count (as set by the `debug.sethook` function).

---

debug.getinfo ([thread,] function [, what])

Returns a table with information about a function. You can give the function directly, or you can give a number as the value of `function`, which means the function running at level `function` of the call stack of the given thread: level 0 is the current function (`getinfo` itself); level 1 is the function that called `getinfo`; and so on. If `function` is a number larger than the number of active functions, then `getinfo` returns **nil**.

The returned table may contain all the fields returned by `lua_getinfo`, with the string `what` describing which fields to fill in. The default for `what` is to get all information available, except the table of valid lines. If present, the option '`f`' adds a field named `func` with the function itself. If present, the option '`L`' adds a field named `activelines` with the table of valid lines.

For instance, the expression `debug.getinfo(1,"n").name` returns a name of the current function, if a reasonable name can be found, and the expression `debug.getinfo(print)` returns a table with all available information about the `print` function.

---

debug.getlocal ([thread,] level, local)

This function returns the name and the value of the local variable with index `local` of the function at level `level` of the stack. (The first parameter or local variable has index 1, and so on, until the last active local variable.) The function returns **nil** if there is no local variable with the given index, and raises an error when called with a `level` out of range. (You can call `debug.getinfo` to check whether the level is valid.)

Variable names starting with '`(`' (open parentheses) represent internal variables (loop control variables, temporaries, and C function locals).

---

debug.getmetatable (object)

Returns the metatable of the given `object` or **nil** if it does not have a metatable.

---

debug.getregistry ()

Returns the registry table (see §3.5).

---

debug.getupvalue (func, up)

This function returns the name and the value of the upvalue with index `up` of the function `func`. The function returns **nil** if there is no upvalue with the given index.

---

debug.setfenv (object, table)

Sets the environment of the given `object` to the given `table`. Returns `object`.

---

debug.sethook ([thread,] hook, mask [, count])

Sets the given function as a hook. The string `mask` and the number `count` describe when the hook will be called. The string mask may have the following characters, with the given meaning:

- **`"c"`:** The hook is called every time Lua calls a function;
- **`"r"`:** The hook is called every time Lua returns from a function;
- **`"l"`:** The hook is called every time Lua enters a new line of code.

With a `count` different from zero, the hook is called after every `count` instructions.

When called without arguments, `debug.sethook` turns off the hook.

When the hook is called, its first parameter is a string describing the event that has triggered its call: `"call"`, `"return"` (or `"tail return"`), `"line"`, and `"count"`. For line events, the hook also gets the new line number as its second parameter. Inside a hook, you can call `getinfo` with level 2 to get more information about the running function (level 0 is the `getinfo` function, and level 1 is the hook function), unless the event is `"tail return"`. In this case, Lua is only simulating the return, and a call to `getinfo` will return invalid data.

`debug.setlocal ([thread,] level, local, value)`

This function assigns the value `value` to the local variable with index `local` of the function at level `level` of the stack. The function returns **nil** if there is no local variable with the given index, and raises an error when called with a `level` out of range. (You can call `getinfo` to check whether the level is valid.) Otherwise, it returns the name of the local variable.

`debug.setmetatable (object, table)`

Sets the metatable for the given `object` to the given `table` (which can be **nil**).

`debug.setupvalue (func, up, value)`

This function assigns the value `value` to the upvalue with index `up` of the function `func`. The function returns **nil** if there is no upvalue with the given index. Otherwise, it returns the name of the upvalue.

`debug.traceback ([thread,] [message] [, level])`

Returns a string with a traceback of the call stack. An optional `message` string is appended at the beginning of the traceback. An optional `level` number tells at which level to start the traceback (default is 1, the function calling `traceback`).

6 - Lua Stand-alone

Although Lua has been designed as an extension language, to be embedded in a host C program, it is also frequently used as a stand-alone language. An interpreter for Lua as a stand-alone language, called simply `lua`, is provided with the standard distribution. The stand-alone interpreter includes all standard libraries,

including the debug library. Its usage is:

```
lua [options] [script [args]]
```

The options are:

- **`-e stat`:** executes string *stat*;
- **`-l mod`:** "requires" *mod*;
- **`-i`:** enters interactive mode after running *script*;
- **`-v`:** prints version information;
- **`--`:** stops handling options;
- **`-`:** executes `stdin` as a file and stops handling options.

After handling its options, `lua` runs the given *script*, passing to it the given *args* as string arguments. When called without arguments, `lua` behaves as `lua -v -i` when the standard input (`stdin`) is a terminal, and as `lua -` otherwise.

Before running any argument, the interpreter checks for an environment variable `LUA_INIT`. If its format is `@filename`, then `lua` executes the file. Otherwise, `lua` executes the string itself.

All options are handled in order, except `-i`. For instance, an invocation like

```
$ lua -e'a=1' -e 'print(a)' script.lua
```

will first set `a` to 1, then print the value of `a` (which is '`1`'), and finally run the file `script.lua` with no arguments. (Here `$` is the shell prompt. Your prompt may be different.)

Before starting to run the script, `lua` collects all arguments in the command line in a global table called `arg`. The script name is stored at index 0, the first argument after the script name goes to index 1, and so on. Any arguments before the script name (that is, the interpreter name plus the options) go to negative indices. For instance, in the call

```
$ lua -la b.lua t1 t2
```

the interpreter first runs the file `a.lua`, then creates a table

```
arg = { [-2] = "lua", [-1] = "-la",
        [0] = "b.lua",
        [1] = "t1", [2] = "t2" }
```

and finally runs the file `b.lua`. The script is called with `arg[1]`, `arg[2]`, ··· as arguments; it can also access these arguments with the vararg expression '`...`'.

In interactive mode, if you write an incomplete statement, the interpreter waits for its completion by issuing a different prompt.

If the global variable `_PROMPT` contains a string, then its value is used as the prompt. Similarly, if the global variable `_PROMPT2` contains a string, its value is used as the secondary prompt (issued during incomplete statements). Therefore, both prompts can be changed directly on the command line. For instance,

```
$ lua -e"_PROMPT='myprompt> '" -i
```

(the outer pair of quotes is for the shell, the inner pair is for Lua), or in any Lua programs by assigning to

`_PROMPT`. Note the use of `-i` to enter interactive mode; otherwise, the program would just end silently right after the assignment to `_PROMPT`.

To allow the use of Lua as a script interpreter in Unix systems, the stand-alone interpreter skips the first line of a chunk if it starts with `#`. Therefore, Lua scripts can be made into executable programs by using `chmod +x` and the `#!` form, as in

```
#!/usr/local/bin/lua
```

(Of course, the location of the Lua interpreter may be different in your machine. If `lua` is in your `PATH`, then

```
#!/usr/bin/env lua
```

is a more portable solution.)

7 - Incompatibilities with the Previous Version

Here we list the incompatibilities that you may found when moving a program from Lua 5.0 to Lua 5.1. You can avoid most of the incompatibilities compiling Lua with appropriate options (see file `luaconf.h`). However, all these compatibility options will be removed in the next version of Lua.

7.1 - Changes in the Language

- The vararg system changed from the pseudo-argument `arg` with a table with the extra arguments to the vararg expression. (See compile-time option `LUA_COMPAT_VARARG` in `luaconf.h`.)
- There was a subtle change in the scope of the implicit variables of the **for** statement and for the **repeat** statement.
- The long string/long comment syntax (`[[`*string*`]]`) does not allow nesting. You can use the new syntax (`[=[`*string*`]=]`) in these cases. (See compile-time option `LUA_COMPAT_LSTR` in `luaconf.h`.)

7.2 - Changes in the Libraries

- Function `string.gfind` was renamed `string.gmatch`. (See compile-time option `LUA_COMPAT_GFIND` in `luaconf.h`.)
- When `string.gsub` is called with a function as its third argument, whenever this function returns **nil** or **false** the replacement string is the whole match, instead of the empty string.
- Function `table.setn` was deprecated. Function `table.getn` corresponds to the new length operator (`#`); use the operator instead of the function. (See compile-time option `LUA_COMPAT_GETN` in `luaconf.h`.)
- Function `loadlib` was renamed `package.loadlib`. (See compile-time option `LUA_COMPAT_LOADLIB` in `luaconf.h`.)
- Function `math.mod` was renamed `math.fmod`. (See compile-time option `LUA_COMPAT_MOD` in `luaconf.h`.)
- Functions `table.foreach` and `table.foreachi` are deprecated. You can use a for loop with `pairs` or `ipairs` instead.
- There were substantial changes in function `require` due to the new module system. However, the new behavior is mostly compatible with the old, but `require` gets the path from `package.path`

instead of from LUA_PATH.
- Function collectgarbage has different arguments. Function gcinfo is deprecated; use collectgarbage("count") instead.

7.3 - Changes in the API

- The luaopen_* functions (to open libraries) cannot be called directly, like a regular C function. They must be called through Lua, like a Lua function.
- Function lua_open was replaced by lua_newstate to allow the user to set a memory-allocation function. You can use luaL_newstate from the standard library to create a state with a standard allocation function (based on realloc).
- Functions luaL_getn and luaL_setn (from the auxiliary library) are deprecated. Use lua_objlen instead of luaL_getn and nothing instead of luaL_setn.
- Function luaL_openlib was replaced by luaL_register.
- Function luaL_checkudata now throws an error when the given value is not a userdata of the expected type. (In Lua 5.0 it returned NULL.)

8 - The Complete Syntax of Lua

Here is the complete syntax of Lua in extended BNF. (It does not describe operator precedences.)

```
chunk ::= {stat [`;´]} [laststat [`;´]]

block ::= chunk

stat ::=  varlist1 `=´ explist1 |
          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 explist1 do block end |
          function funcname funcbody |
          local function Name funcbody |
          local namelist [`=´ explist1]

laststat ::= return [explist1] | break

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

varlist1 ::= var {`,´ var}

var ::=  Name | prefixexp `[´ exp `]´ | prefixexp `.´ Name

namelist ::= Name {`,´ Name}

explist1 ::= {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 ::=  `(´ [explist1] `)´ | tableconstructor | String

function ::= function funcbody

funcbody ::= `(´ [parlist1] `)´ block end

parlist1 ::= namelist [`,´ `...´] | `...´

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

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

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

fieldsep ::= `,´ | `;´

binop ::= `+´ | `-´ | `*´ | `/´ | `^´ | `%´ | `..´ |
         `<´ | `<=´ | `>´ | `>=´ | `==´ | `~=´ |
         and | or

unop ::= `-´ | not | `#´
```

Last update: Tue Oct 3 21:27:28 BRT 2006
译文最后更新：修改几处别字 2009年4月7日