Lua 用户参考手册

0.前言

这是译者第一次翻译此类手册，本身英文水平不高，估计会有误译，欢迎来信讨论更正，我的邮箱：yanjun1982@126.com。

1. 介绍

Lua 是一种有很强数据描述能力且扩展性非常强的编程语言。它还为面向对象编程、函数式编程和数据驱动编程提供了很好的支持。Lua 的目标是要成为一种强大的、轻量的、可嵌入的脚本语言，以便任何项目都可以使用它。Lua 使用标准 C 和 C++两者的子集来编写的，以库的形式实现。

作为一种扩展性语言，Lua 没有“main”入口这个概念：它只作为一个嵌入模块来使用，它可以调用自身，也可以调用宿主语言（即使用 Lua 的语言）。宿主语言可以执行 Lua 的代码，可以读写 Lua 代码中的变量，可以将 C 函数注册给 Lua 代码调用。通过使用 C 函数，Lua 可以扩展为广泛适用于各种领域，比如创建自定义语法框架的编程语言。Lua 的发布版本包含有一个名字叫 lua 的程序，它是使用 Lua 库实现的一个独立解释器，可以用于交互式编程及批处理。

根据 Lua 的许可证声明，Lua 是免费的，不需要授权就可以使用。本手册中描述的相关实现可以在 Lua 的官方网站：www.lua.org 中找得到。

和其它的参考手册一种，这份文档比较枯燥无味。如果想要了解 Lua 设计背后做的一些决定，你可以在 lua 官网中找到相关的技术讨论页面。如果你想要了解 Lua 编程的更多细节，可以阅读 Roberto 的书：《Programming in Lua》。

2. 基本概念

本章说明 Lua 的一些基本概念。

2.1 - 值与类型

Lua 是一种动态类型的编程语言。这意味着 Lua 的变量没有类型，只有值才有类型。因此，Lua 没有类型定义这个行为。所有的值都能通过自身来表示自己的类型。

Lua 中的所有值都是“第一类（first-class）”值。这意味着所有的值都可以储存在变量中，可以当作参数传给函数，可以被函数当结果返回。

Lua 一共有 8 种基本数据类型：空值（nil），布尔（boolean），数字（number），字符串（string），函数（function），用户数据（userdata），线程（thread）和表（table）。其中值 **nil** 的类型是空值，它的主要作用就是用来区分别的值，通常用来表示其它类型的缺省值。值 **false** 和 **true** 的类型是布尔型。**nil** 和 **false** 在条件判断中都表示 false，其它任意值则表示 true。数字类型就是指一切的数字（双精度浮点数），操作数字类型值的时必须遵守 C 语言的规则，也即 IEEE754 标准。（你可以很容易去修改 Lua 数字类型的底层表示，比如你可以将数字类型的表示修改为单精度浮点型或长整型，具体可以看 luaconf.h 这个源代码文件）字符串类型是一种不可改变的字符序列。字符串中可以包含所有的 8 位的字符，包括终结符（‘\0’）。

Lua 可以调用（和处理）Lua 和 C 实现的函数。（参考 **3.4.9**）

用户数据类型（以下用 userdata 表示）可以保存任意的 C 数据类型。一个 userdata 实际上是一个指向某块内存空间的指针。一共有两种 userdata：full userdata 和 light userdata。前者指向的内存由 Lua 管理，后者指向的内存由宿主语言（C 或其它）管理。除了赋值和相等性比较操作，userdata 没有预定义的其它操作，但是通过使用元表（metatable），你可以定义 full userdata 的各种操作（参考 **2.4**）。在 Lua 代码中是不能创建 userdata 的值的，userdata 的值只能通过 C API 来创建和修改，这样可以保证宿主数据的安全性。

线程类型表示一个独立的执行过程，它可以用来实现协程（参考 **2.6**）。不要把 Lua 的线程和操作系统的线程混淆，Lua 在所有的操作系统上都支持协程，包括那些不支持线程的操作系统。

表类型（以下用 table 表示）是一种关联数组，除了可以用数字作为 table 的索引（也可以叫键）外，还可以使用除了 **nil** 和 NaN（一种未定义或不可表示的数字类型，比如 0/0）外的所有 Lua 值来作索引。table 中的值可以是任意类型的（除了 **nil**）。也就是说，如果一个键对应的值为 **nil**，则认为这个键不属于 table 的一部分。反过来说，table 中一个不存在的键，它关联的值就是一个 **nil**。

table 是 Lua 中唯一的数据结构；它可以用来表示数组，序列，符号表，集合，记录，图，树等。为了表示一个记录，Lua 使用带有名字的域来表示 table 的索引，比如 a.name，它是 a[“name”]的一种语法糖（syntactic sugar）。在 Lua 中，有好几种很方便的方法来创建 tables。（参考 **3.4.8**）

我们使用“序列”这个术语来描述索引为从 1 至 n 的 table，n 表示序列的长度。（参考 **3.4.6**）

和索引一样，table 中的值也可以是 Lua 中任意类型的值。因为函数是第一类值，所以 table 中的值可以是函数，即 table 是可以带方法的。（参考 **3.4.10**）

使用索引去获得 table 中的一个值时，它要遵守语言的“原始相等”规则。表达式 a[i] 和 a[j]表示同一个元素的前提是 i 和 j 是原始相等的（即 i 和 j 没有元方法）。

table，函数，线程和 full userdata 类型的值都是引用类型，即它们保存的值并不是实际的数据而是一个引用，因此，对这些类型的值进行赋值、参数传递、函数中返回等操作的操作对象都是它们的引用，这些操作都不产生数据的副本。

可以使用 Lua 标准库中的 **type** 函数来获得指定值的类型名字。（参考 **6.1**）

2.2 - 环境表与全局环境

任何的全局变量名 var 在语法分析的时候都会被翻译为_ENV.var 的形式，这个会在 **3.2** 和 **3.3.3** 时再讨论。每一个代码块在它的作用域内都有一个叫_ENV 的外部局部变量（参考

3.3.2），所以_ENV 在代码块里也不是一个全局变量。

虽然_ENV 看起来像是编译器生成的内容，但它确是实实在在存在的。你可以直接在 Lua 的代码中使用_ENV 来定义一个新的变量和参数。每一个通过_ENV 引用的全局名字，它在程序中是被全局可见的，它遵守 Lua 的可见性规则（参考 **3.5**）。

任何一个用法如_ENV 一样的 table 都称为环境。

Lua 中保留了一个特殊的全局环境。它的值被保留在 Lua 栈中，通过一个特殊的索引值来访问（参考 **4.5**）。在 Lua 的代码中，变量_G 指向这个值。

当编译一个代码块时，Lua 会把全局环境中的内容作为数据来初始化_ENV（参考 **load**）。因此，Lua 代码中的全局变量指的是全局环境中的内容。此外，所有标准库中的函数都会被加载进全局环境中，同时也提供了一些操作这个环境的函数。你可以使用 **load**（或者 **loadfile**）在加载一个代码块时指定不同的环境。（在 C 代码中，你必须在加载代码块后通过改变第一个上值（upvalue）来实现环境的改变。）

如果你改变了 Lua 注册表中的全局环境（通过 C 代码或调试库），那么之后加载的代码块将会得到一个新的环境。之前加载的代码块不受影响，它仍然使用之前的_ENV 变量。此外，变量 **_G**（储存在原先的全局环境中）从来不会被 Lua 更新。

2.3 - 错误处理

因为 Lua 是一种可扩展的嵌入式语言，它的所有行为都是通过主程序调用 Lua 库函数来开始的（参考 **lua_pcall**）。在编译或运行 Lua 代码时，只要有错误发生，那么控制权就会交回给主程序以采取合适的措施（比如打印一个错误信息）。

Lua 代码通过调用 **error** 函数可以自己产生一个错误信息。如果你想要捕获这些错误，那么你可以使用 **pcall** 或 **xpcall** 在保护模式下调用函数。

只要发生了错误，那么一个带有相关信息的错误对象（也可以叫错误信息）就会被抛出。Lua 代码只能产生携带字符串信息的错误对象，但主程序可以产生携带任何类型信息的错误对象。

当你使用 **xpcall** 或 **lua_pcall** 时，你可以指定一个消息处理函数，当错误发生时会调用该函数。这个函数处理原始的错误信息并返回处理后的错误信息。它在栈展开之前被调用，所以它可以收集到更多的与错误相关的信息，比如检查调用栈并创建栈回溯。这个处理函数也是在保护模式下调用的，所以当这个处理函数本身也有错误产生时，也会调用它自身。如果因此出现了死循环，那么 Lua 会在一定的循环次数后返回合适的错误消息。

2.4 - 元表与元方法

Lua 中的每一个值都可以绑定一个元表。这个元表是一个普通的 table，它可以定义与该值相关的某些操作。你可以通过设置元表中特定域的值来改变 Lua 值的行为。比如当一个非数字型的值作为加法操作的操作数时，Lua 会检查该值是否绑定了元表并且元表设置了域“__add”的值为一个函数，如果是，那么 Lua 就会调用这个函数来进行该值的加法操作。

元表中的键的名字由它对应的事件名得来；对应的值叫做元方法。在上面的例子中，事件是“add”，元方法就是进行加法操作时调用的函数。

你可以使用 **getmetatable** 函数来查询某个值是否绑定了元表。

你可以使用 **setmetatalbe** 函数来绑定 table 的元表。你不能在 Lua 代码中改变其它数据类型的元表（除非使用调试库）；你只能使用 C API 来实现这个目的。

每个 Table 和 full userdata 类型的值都可以有自己单独的元表（但多个 table 和 userdata 可以共享一个元表）。其它的每一种类型对应地只能绑定一个元表。也就是说，所有的数字类型只能绑定同一个元表，所有的字符串类型只能绑定同一个元表，等等。除了字符串类型的值默认有一个元表外（参考 **6.4**），其它的值默认是没有元表的。

一个元表控制了一个对象的算术、比较、连接，取长度操作和索引操作的行为。元表也可以定义一个在 userdata 或 table 被垃圾回收时调用的函数。当 Lua 在对一个值进行这些操作时，它会检查这个值是否有元表及元表中是否有对应的事件。如果有，那事件对应的元方法就控制了 Lua 如何处理这些操作。

元表可以控制接下来的这些操作。每个操作都被对应的名字所标识。每个操作在元表中对应的键名以两个下划线（“__”）作为前缀；举个例子，“add”操作对应的键名为“__add”。

Lua 函数可以很好地描述这些操作在 Lua 的解释器中是如何运行的。下面的 Lua 代码仅仅是用来演示说明的，真正的行为是在解释器中硬编码的，它比下面的模拟效率更高。在下面的描述中，所有的函数（**rawget**，**tonumber** 等等）都在 **6.1** 章节中有说明。特别说明一下，为了访问一个指定对象的元方法，我们使用下面的表达式：

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

这个可以解释为：

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

这意味着访问元方法并不会触发别的元方法，并且访问一个没有绑定元表的对象的元方法时不会出错（会返回 nil）。

对于一元操作符-和#，调用元方法时，函数的第二个参数是没有意义的，保留它仅仅是为了方便 Lua 的内部实现，在以后的版本中它有可能被删除，所以在下面的伪代码中不会出现这个参数。（大部分情况下，使用第二个参数是不适合的）

- "add": + （加法操作）

下面定义的函数 getbinhandler 模拟了 Lua 是如何选择二元操作符的元方法的。首先判断第一个操作数，如果它没有定义元方法再判断第二个操作数有没有定义元方法。

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

使用这个函数，两个操作数 op1和 op2相加的行为如下：

```
function add_event (op1, op2)
  local o1, o2 = tonumber(op1), tonumber(op2)
  if o1 and o2 then  -- op1 和 op2 都是数字类型？
    return o1 + o2    -- 这里的'+'表示原始的加法操作
  else  -- 至少有一个操作数不是数字类型
    local h = getbinhandler(op1, op2, "__add")
    if h then
      -- 调用它的元方法
```

```
        return (h(op1, op2))
      else   -- 找不到元方法，默认处理
        error(…)
      end
   end
end
```

- "sub": - （减法操作）行为和加法操作类似。
- "mul": * （乘法操作）行为和加法操作类似。
- "div": / （除法操作）行为和加法操作类似。
- "mod": % (求余操作)行为和加法操作类似。它的原始操作是 o1 - floor(o1/o2)*o2。
- "pow": ^ （求幂操作）行为和加法操作类似。 它的原始操作是 pow 函数（来自 C 的数学库）。
- "unm": - （一元操作符-）

```
function unm_event (op)
   local o = tonumber(op)
   if o then   -- 数字类型的操作数？
      return -o   -- '-' 原始的负号操作
   else   -- 操作数不是数字类型
      -- 尝试访问它的元方法
      local h = metatable(op).__unm
      if h then
         -- 调用元方法
         return (h(op))
      else   -- 找不到元方法，默认处理
         error(…)
      end
   end
end
```

- "concat": .. （连接操作符）

```
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   -- 原始的字符串连接操作
   else
      local h = getbinhandler(op1, op2, "__concat")
      if h then
         return (h(op1, op2))
      else
         error(…)
      end
```

```
    end
end
```

- "len": # （取长度操作符）

```
function len_event (op)
    if type(op) == "string" then
        return strlen(op)          -- 原始的字符串求长度操作
    else
        local h = metatable(op).__len
        if h then
            return (h(op))         -- 调用元方法
        elseif type(op) == "table" then
            return #op                      -- 原始的 table 求长度操作
        else    -- 找不到元方法，默认出错处理
            error(…)
        end
    end
end
```

table 的求长度操作参考 **3.4.6**。

- "eq": == （相等比较操作符） 函数 getequalhandler 模拟了 Lua 是怎么选择比较操作符的元方法的。只有当两个待比较的数值它们的类型相同，且类型为 table 或 full userdata，且__eq 元方法也相同时，getequalhandler 才返回__eq 的值作为比较操作符的元方法。

```
function getequalhandler (op1, op2)
    if type(op1) ~= type(op2) or
        (type(op1) ~= "table" and type(op1) ~= "userdata") then
        return nil          -- 不同的值
    end
    local mm1 = metatable(op1).__eq
    local mm2 = metatable(op2).__eq
    if mm1 == mm2 then return mm1 else return nil end
end
```

相等比较事件的实现如下：

```
function eq_event (op1, op2)
    if op1 == op2 then      -- 原始相等？（即 op1 和 op2 是同一个对象）
        return true      -- 值相等
    end
    -- 尝试使用元方法
    local h = getequalhandler(op1, op2)
```

```
    if h then
        return not not h(op1, op2)
    else
        return false
    end
end
```

注意返回的结果总是布尔型的。

- "lt": < （小于比较操作符）

```
function lt_event (op1, op2)
    if type(op1) == "number" and type(op2) == "number" then
        return op1 < op2      -- 数字类型比较
    elseif type(op1) == "string" and type(op2) == "string" then
        return op1 < op2      -- 按字典中的顺序比较
    else
        local h = getbinhandler(op1, op2, "__lt")
        if h then
            return not not h(op1, op2)
        else
            error(…)
        end
    end
end
```

注意返回的结果总是布尔型的。

- "le": <= （小于等于比较操作符）

```
function le_event (op1, op2)
    if type(op1) == "number" and type(op2) == "number" then
        return op1 <= op2      -- 数字类型比较
    elseif type(op1) == "string" and type(op2) == "string" then
        return op1 <= op2      -- 按字典中的顺序比较
    else
        local h = getbinhandler(op1, op2, "__le")
        if h then
            return not not h(op1, op2)
        else
            h = getbinhandler(op1, op2, "__lt")
            if h then
                return not h(op2, op1)
            else
                error(…)
            end
```

```
    end
  end
end
```

注意当没有定义“le”这个元方法时，Lua 假设 a <= b 等于 not(b < a)并使用“lt”来判断。

和其它比较操作一样，返回结果总是布尔型的。

- **"index":**索引操作。当使用一个不存于 table 中的键去索引 table 中的内容时会尝试调用此元方法。（当索引操作作用于的对象不是一个 table 时，那么所有键都是不存在的，所以元方法一定会被尝试调用。）

```
function gettable_event (table, key)
  local h
  if type(table) == "table" then
    local v = rawget(table, key)
    -- 如果键存在，返回原始的值
    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))        -- 调用元方法
  else return h[key]                  -- 或者把元方法当作一个 table 来使用
  end
end
```

- **"newindex":** table 赋值操作 table[key] = value。使用一个不存在于 table 中的键来给 table 中的域赋值时会尝试调用此元方法。

```
function settable_event (table, key, value)
  local h
  if type(table) == "table" then
    local v = rawget(table, key)
    -- 如果键存在，那就做原始赋值
    if v ~= nil then rawset(table, key, value); return end
    h = metatable(table).__newindex
    if h == nil then rawset(table, key, value); return end
  else
    h = metatable(table).__newindex
```

```
    if h == nil then
      error(…)
    end
  end
  if type(h) == "function" then
    h(table, key,value)                -- 调用元方法
  else h[key] = value                  -- 或者把元方法当作一个 table 来使用
  end
end
```

- "call":当 Lua 调用一个值的时候被调用。

```
function function_event (func, ...)
  if type(func) == "function" then
    return func(...)     -- 原始调用
  else
    local h = metatable(func).__call
    if h then
      return h(func, ...)
    else
      error(…)
    end
  end
end
```

2.5 - 垃圾回收

Lua 对内存进行自动管理。这意味着你既不需要担心内存分配的问题，也不需要担心内存释放的问题。Lua 通过垃圾回收器回收那些已经死了的数据对象（即 Lua 不可能再访问到的对象）从而实现内存的自动管理。Lua 用到的所有内存都会被自动管理，包括字符串，table，userdata，函数，线程，内部结构等等。

Lua 实现的是一个增量式的标记-清除垃圾回收器。它使用两个数值来控制整个垃圾回收循环的行为：“暂停”（garbage-collector pause）和“步长”（garbage-collector step multiplier）。它们的单位都是百分一（比如值 100 在内部表示为 1）。

“暂停”控制回收器在开始新一轮垃圾回收前要等待多久。数值越大表示回收器工作越不积极。数值小于 100 表示回收器从不等待，即结束一轮垃圾回收后立刻开始下一轮。数值等于 200 表示回收器会在内存占用量为之前的 2 倍时开始新一轮的垃圾回收。

“步长”控制了内存的回收速度，它与内存的分配速度有关。数值越大，回收器工作越积极，但同时也增加了每个步骤的工作量。数值小于 100 会使得回收器工作很慢，慢到回收器永远完成不了一轮垃圾回收。默认值是 200，这意味着内存回收的速度是分配速度的 2 倍。

如果你设置“步长”为一个很大的数值（大于程序最大可用内存的 10%），那回收器就会和“停止世界”（stop-the-world）式的回收器行为很像。如果把“暂停”设为 200，那回收器的行为就会和老版本的 Lua 垃圾回收器一样，在内存占用为之前的 2 倍时开始一轮完整的垃圾回收。

你可以使用 C 函数 **lua_gc** 或 Lua 函数 **collectgarbage** 来改变“暂停”和“步长”这两个数值。你也可以使用这两个函数来直接控制 Lua 的垃圾回收器（比如停止和重新开始）。

你可以将回收器由增量式改为世代式，这是 Lua5.2 版本中的一个实验性特色。世代式的回收器假设大部分对象都是在刚创建出来不久后就死掉了，这样的话，在回收的时候只需要遍历这些新的对象就行了。这样可以减少垃圾回收的时间，但会因此增加内存的使用（因为老的对象会被堆积起来）。为了缓解这个问题，世代式的回收器隔一段时间会进行一次全回收。记住这只是一个实验性质的特色，欢迎使用它，但自己要衡量好。

2.5.1 - 垃圾回收元方法

你可以设置 table 和 full userdata 的垃圾回收元方法（参考 **2.4**）。这些元方法也称 finalizers。Finalizers 可以让你协调管理外部资源的垃圾回收（比如文件关闭，网络或数据库链接，或释放你自己申请的内存）。

为了让一个对象（table 或 userdata）在被回收时能被终结处理，你必须给它做一个标记。方法是给这个对象绑定一个带有“__gc”域的元表。值得注意的是，如果你在绑定元表时元表没有“__gc”这个域，就算你之后给元表设了这个域，对象也不会被标记。不过，如果一个对象被标记后，你可以随便改变元表的“__gc”域。

当一个被标记的对象成为垃圾后，它不会立刻被回收器给回收。Lua 将它链入一个链表中。当完成一次回收后，Lua 会对链表中的每一个对象执行和下面函数功能等同的操作：

```
function gc_event (obj)
  local h = metatable(obj).__gc
  if type(h) == "function" then
    h(obj)
  end
end
```

在每一轮垃圾回收流程的最后，对象的终结处理会按照它们被标记时的逆序被调用。也就是说，第一个执行终结处理的对象是程序中最后一个被标记的对象。每一个终结处理都可能在任意的代码执行点中被调用。

因为被回收的对象在它的终结处理中会被用到，所以它必须被 Lua 复活。通常，这个复活的行为是非常短暂的，而且对象占用的内存空间会在下一轮的垃圾回收中被释放。不过，如果在终结处理中又将这个对象保存到了一个全局的地方（比如一个全局变量），那么这个对象就被永久复活了。不管怎样，当一个对象无法被访问到时，它占用的内存就会被回收掉，它的终结处理永远不会被调用两次或以上。

当你关闭 Lua 的虚拟机时（参考 **lua_close**），Lua 会按照对象被标记的逆序来调用所有被标记对象的终结处理。如果在这个阶段有一个终结处理标记了一个新的对象，那这个新标记的对象是不会被终结处理的。

2.5.2 - 弱表

如果一个表中有元素是弱引用，那这个表就叫弱表。弱引用是会被垃圾回收器忽略的。换一种说法，如果一个对象的唯一一个引用是弱引用，那垃圾回收器就会回收这个对象。

一个弱表可以有弱键，弱值，或者两者都有。一个有弱键的表允许它的键被回收，但阻止它的值被回收。一个有弱键和弱值的表允许键和值都被回收。不管怎样，如果键或值被回收了，那么键值对也就从表中被移除了。一个表的弱属性由它元表中的__mode 域控制。如果__mode 是一个带有字符“k”的字符串，那表中的键就是弱的。如果__mode 中带有字符“v”，那表中的值就是弱的。

一个有着弱键和强值的表有时也被称为“短命表”（ephemeron table）。在短命表中，一个值是否可以被访问主要是看它对应的键是否可以被访问。特殊地，如果一个键的唯一引用来自于它的值，那这个键值对相当于被移除了。

一个表的弱属性发生的任何改变都只能在下一轮的垃圾回收中起作用。尤其是当你把一个表的弱属性改成强一点的属性，Lua 还是会在属性起作用前回收一些内容。

只有那些需要显式构造的对象才会从弱表中移除。数值类型的，比如数字和 C 函数都不受垃圾回收系统管理，因此，他们不会从弱表中移除（除非与之关联的值被回收了）。虽然字符串受垃圾回收系统管理，但是它不需要显式的构造，因此也不会从弱表中移除。

复活的对象（也就是那些正在进行终结处理且只被其它正在进行终结处理的对象引用的对象）在弱表中有特殊的行为。当一个对象可以被释放时，它作为弱值，会在终结处理前从弱表中移除，但作为弱键，它会在终结处理完成后的下一轮垃圾回收周期中被移除。这样的机制使得终结处理可以通过弱表来访问与对象相关联的属性。

如果一个弱表是一个复活的对象，那么直到下一个回收周期前，它可能不会被正常清除。

2.6 - 协程

Lua 支持协程，也叫做协作式多线程（collaborative multithreading）。一个协程在 Lua 中表示一个独立执行的线程。与多线程操作系统中的线程概念不一样，一个协程只能被 yield 函数暂停执行。

使用 **coroutine.create** 来创建一个协程。该函数的唯一参数是一个函数，它表示协程的主函数。成功创建协程后，该函数返回一个对象（一个类型为 thread 的对象）来表示创建出来的协程；协程创建出来后不会自动开始执行。

调用 coroutine.resume 函数来执行一个协程。当你第一次调用 **coroutine.resume** 时，要把 **coroutine.create** 函数返回的对象作为第一个参数传给它，这样协程就开始执行它的主函数。传给 **coroutine.resume** 的其它参数会传递给协程的主函数。协程一旦开始执行，它就只能在执行完毕或让出执行权时才会停止。

这里有两种方法可以终止协程的执行，一种是主函数正常返回（显式调用 return 或执行完函数最后一行代码），另一种是发生了一个非保护的错误。在第一种情况，**coroutine.resume** 返回 **true** 以及主函数返回的值。另一种情况，当错误发生时，**coroutine.resume** 返回 **false** 以及错误信息。

调用 **coroutine.yield** 可以使协程让出它的执行权。当一个协程让出执行权时，就算 yield 发生在嵌套的函数中（也就是说，不仅仅是指主函数，还包括主函数直接或间接调用的函数），对应的 **coroutine.resume** 也会立刻返回。当协程让出执行权时，**coroutine.resume** 也会返回 **true** 及传递给 **coroutine.yield** 的任意值。当你恢复协程的执行时，它会在之前调用 **coroutine.yield** 的地方接着执行，并且 **coroutine.yield** 会返回传递给 **coroutine.resume** 的那些额外的参数。

和 **coroutine.create** 一样，**coroutine.wrap** 也可以创建一个协程，但不同的是它并不返回协程对象，而是返回一个函数，当调用这个函数时，协程就会执行。传递给这个函数的所有参数都会相当于 **coroutine.resume** 的额外参数。**coroutine.wrap** 的返回值和

coroutine.resume 的差不多，除了第一个返回值（布尔型的错误码）。与 **coroutine.resume** 不一样，**coroutine.wrap** 不捕获错误，任何的错误都会传递到上级的调用者。

下面的代码展示了协程是怎么工作的：

```
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"))
```

执行后的结果如下：

```
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
```

你也可以使用 C API 来创建和操作协程，请参考 **lua_newthread**，**lua_resume** 和 **lua_yield**。

3. 语言

这一部分说明了 Lua 的词法，语法及语义。换句话说，这一部分说明了哪些单词是关键字，它们之间是如何组合的，以及这些组合的含义。

Lua 使用扩展巴科斯范式（extened BNF）来解释语言的构造，{a}表示 0 或多个 a，[a]表示 a 是个可选项。非终结符用非粗体单词表示，关键字用粗体单词表示，终结符用两个单引号括住，比如'='。完整的 Lua 语法可以在本手册最后的第 **9** 章找到。

3.1 - 词法约定

Lua 是一种自由形式的语言。除了变量名与关键字之间的分隔符，其它的空格（包括换行符）和词法元素（tokens）之间的注释都会被忽略。

Lua 中的变量名（也叫标记符）可以由非数字开头的字母、数字和下划线组成。标记符用来命名变量，table 中的域（键）和标签。

下面的关键字被系统保留，它们不能被用作变量名和标签。

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

Lua 是大小写敏感的语言：and 是一个保留关键字，但 And 和 AND 是另外两个不同的合法名字。作为一个约定，由下划线开头，后面跟大写字母的名字（比如**_VERSION**）是 Lua 使用的内部名字。

下面的符号表示其它保留的标记：

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

一个字符串可以由一对单引号或一对双引号定义，它可以包含 C 风格的转义符：'\a'（响铃），'\b'（退格），'\f'（换页），'\n'（换行），'\r'（回车），'\t'（水平制表），'\v'（垂直制表），'\\'（反斜杠），'\"'（双号号），'\''（单引号）。一个反斜杠后面跟换行符在字符串中就表示为一个字面的换行符。'\z'可以忽略后面跟的空白字符，包括换行符，它可以在不加入换行符和空格符的情况下将一个很长的字符串分成多行。

字符串中字符可以用一个数值来表示。比如\xXX，XX 是对应字符的十六进制 ASCII 码，又比如\ddd，ddd 是对应字符的十进制 ASCII 码。（注意一定要用三位数字来表示。）Lua 的字符串可以包含任意的 8 位值，包括用'\0'表示的内嵌 0。

Lua 还可以用中括号来定义带有格式的字符串。我们使用左中括号和等号来表示字符串的定义开始符，等号的个数表示开始符的级别，比如[[表示 0 级的开始符，[=[表示 1 级的开始符，以此类推。字符串的定义结束符也是类似的，比如一个 4 级的结束符为]====]。一对同级的开始结束符定义了一个字符串，除了同等级的结束符外，该字符串中可以包含任意形式的文字。这种形式的字符串可以包含多行文字，会忽略其它等级的开始和结束符。任何形式的行结尾符（回车符，换行符，回车换行符，换行回车符）都会被转成换行符。

除了上面说的那些特殊字符外，其它字符在一个字符串中都表示自身。Lua 可以用文本方式来打开一个文件，某些操作文件的系统函数在处理一些控制符的时候有可能会出现问题。因此，用转义符的方式来表示非文本字符是比较安全的。

为了便利，如果一个字符串的定义开始符后后紧跟着一个换行符，那么这个换行符会被忽略掉。下面的 5 个例子在使用 ASCII 码（该种编码中，'a' 的编码是 97，换行符是 10，'1' 是 49）的系统中表示的是同一个字符串：

```
a = 'alo\n123"'
a = "alo\n123\""
```

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

一个数值常量可以带有小数及指数部分，其中指数部分用字符'e'或'E'来表示，后面跟一个数值表示乘以 10 的多少次方。Lua 也支持十六进制的数值常量，用 0x 或 0X 开头的数值表示。十六进制的常量同样也可以带有小数及指数部分，其中指数部分用字符'p'或'P'来表示，后面跟一个数值表示乘以 2 的多少次方。下面的例子都是合法的数值常量：

```
3       3.0       3.1416       314.16e-2       0.31416E1
0xff   0x0.1E   0xA23p-4     0X1.921FB54442D18P+1
```

不在字符串中的两个横杠（--）表示注释。如果紧跟--后面的不是一个字符串定义开始符，那么就表示这是一个短注释，它将注释掉从--开始至该行行尾的所有内容，否则就表示这是一个长注释，它将注释掉从注释开始符开始至匹配的注释结束符之间的所有内容。长注释经常用于注释一段代码。

3.2 - 变量

变量是存放值的地方。LUA 有三种类型的变量：全局变量，局部变量和表域。

一个单独的名字可以用来表示一个全局变量或局部变量（或者函数的形式参数，实际上也是一种局部变量）

var: : = Name

名字表示一个标识符，**3.1** 中有说明。

如果不用 local（参考 **3.3.7**）来声明，那么任何变量都是全局的。局部变量受作用域限制。局部变量可以被它作用域范围内定义的函数自由地访问（参考 **3.5**）。

变量在被赋值之前，它的值是 **nil**。

中括号用来索引 table 中的值：

var :: = prefixexp '[' exp ']'

访问表域操作的含义可以通过元表来改变。访问表中已存在的变量 t[i]的操作等同于调用函数 gettable_event(t,i)。（gettable_event 函数的完整描述可以参考 **2.4**，Lua 中是没有定义该函数的。我们在这里使用它只是出于解释的目标。）

语法 var.Name 其实是语法 var["Name"]的一种语法糖：

var :: = prefixexp '.' Name

对全局变量 x 的访问等同于_ENV.x。由于代码块编译方式的原因，_ENV 不是一个全局的名字（参考 **2.2**）。

3.3 - 语句

和 Pascal 或 C 一样，Lua 几乎支持所有的传统语句。包括有：赋值、控制、结构、函数调用和变量声明。

3.3.1 - 语句块

一个语句块为一系列按顺序执行的语句组成：

block ::= {stat}

Lua 有空的语句，它允许你使用分号来隔开语句，也允许你在一个语句块中用一个分号或连着的两个分号作为开始：

stat ::= ‘;’

函数调用与赋值都可以以小括号开始。这将有可能导致 Lua 的语法出现模棱两可的情况。思考一下下面的代码段：

a = b + c

(print or io.write)(‘done’)

根据语法它们可被解释成两种情况：

a = b + c(print or io.write)(‘done’)

a = b + c;(print or io.write)(‘done’)

分析器总是将它当作第一种情况来处理，将左小括号视为函数调用参数的开始。为了避免这种模棱两可的情况，在小括号前加上一个分号是一种良好的习惯。

;(print or io.write)(‘done’)

可以显式声明一个语句块：

state ::= **do** block **end**

显式声明的语句块在控制变量的作用域上很有用，有时它也用于在另外一个语句块的中间加入 **return** 语句（参考 **3.3.4**）。

3.3.2 - 代码块

Lua 编译的单元称之为代码块。从句法上面来看，一个代码块其实就是一个语句块：

chunk ::= block

Lua 将代码块当作一个有不定数量参数的匿名函数体来处理（参考 **3.4.10**）。因此，代码块可以定义局部变量，接收参数和返回值。进一步来说，这样的匿名函数其实是在一个叫 _ENV 的外部局部变量的作用域下编译的（参考 **2.2**），就算该函数没有使用到_ENV 这个变量，_ENV 也是它的唯一一个上值（upvalue）。

一个代码块可以存在于一个文件或主程序中的一个字符串中。为了执行一个代码块，Lua 首先将代码块预编译成虚拟机的指令，然后再使用虚拟机的解释器去执行编译后的代码。

代码块也可以被预编译成二进制的形式；详细的可以参考 luac 这个程序项目。源代码及编译后的代码是可以互换的；Lua 自动检测文件类型并作出相应的操作。

3.3.3 - 赋值

Lua 允许多重赋值。因此，赋值语句的语法左边是变量列表，右边是表达式列表。两边的元素都是使用逗号来分隔：

stat ::= varlist ‘=’ explist

varlist ::= var {‘,’var}

explist ::= exp {‘,’exp}

表达式将会在 **3.4** 节中讨论。

在赋值之前，值的数量会调整为与变量的数量一致。如果值的数量比变量的数量要多，那多出的值将被舍去。如果值的数量比变量的数量要少，那就会用 **nil** 来补齐。如果表达式列表中的最后一个元素是一个函数，那么该函数的所有返回值都会在赋值之前进入值列表（除非该调用被小括号括起来了。参考 **3.4**）。

赋值语句首先是计算所有表达式的值，然后再进行赋值操作。因此代码

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

设置 a[3]的值为 20，它不会影响到 a[4]，这是因为 a[i]中的 i 是在将 4 赋值给它前计算的（值为 3）。类似地，下面这行代码

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

交换了 x 与 y 的值，还有

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

循环地修改了 x,y 和 z 的值。

可以通过元表来改变对全局变量和表域的赋值操作的含义。对已建立索引的变量的赋值操作 t[i] = val 等同于函数 settable_event(t,i,val)。（settable_event 函数的完整描述可以参考 **2.4**。Lua 中是没有定义该函数的。我们在这里使用它只是出于解释的目标。）

对全局变量的赋值 x = val 等同于_ENV.x = val（参考 **2.2**）。

3.3.4 - 控制结构

控制结构 **if**，**while** 和 **repeat** 有着常见的含义和熟悉的语法：

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

Lua 也有 **for** 语句，有两种书写方式（参考 **3.3.5**）。

控制结构中的条件表达式可以返回任意的值。**false** 和 **nil** 值会被当作否。其余不同于 **nil** 和 **false** 的值被当作真（数字 0 与空字符串也是真）。

在 **repeat-until** 循环体中，语句块的结束位置并不是关键字 **until** 而是 **until** 后面的条件判断语句。因此，在条件判断语句中可以引用循环语句块中声明的局部变量。

goto 语句将程序的控制流程跳转到标签处。出于语法规则的原因，标签在 Lua 中也是一个语句：

```
stat ::= goto Name
state ::= label
label ::= ‘::’Name’::’
```

除了在嵌套语句块和嵌套函数中定义了相同名字标签的情况外，一个标签在它定义的整个语句块中都是可见的。只要不跳进一个局部变量作用域的内部，**goto** 语句可以跳到可见的任意标签处。标签和空语句被称为虚语句，因为它们什么都没做。

break 语句可以终止 **while**、**repeat**、**for** 这些循环语句的执行，直接跳到循环语句的下一个语句：

```
stat ::= break
```

break 语句只能终止它所在循环体的执行。

return 语句用来在函数或代码块（实际上也是函数）中返回值。函数可以返回多个值，

所以 return 语句的语法是：

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

return 语句只能作为语句块中的最后一条语句出现。如果有必要在语句块的中间返回，那么可以使用一个内部语句块来实现：**do return end**。这样，**return** 语句就是语句块（内部的）中的最后一条语句了。

3.3.5 - for 语句

for 语句有两种形式：一种是数值型，一种是泛化型。数值型的 for 循环通过一个不断变化的变量来重复执行语句块。它的语法如下：

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

语句块会重复执行，直到以第一个 exp 作为初始值的 Name 变量的值超过第二个 exp 的值为止，每执行一次语句块，Name 的值都会加上第三个 exp 的值。更准确一点，for 语句像下面这样：

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

它等于以下代码：

```
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
```

注意以下几点：

- 在开始执行循环体之前，三个控制表达式会被执行且只执行一次，它们的结果必须是数字类型。
- var，limit 和 step 是隐形变量。在这里显示只是为了方便解释的目的。
- 如果第三个表达式（step）没有写，那么它的默认值就是 1。
- 你可以使用 **break** 语句来退出 **for** 循环。
- 循环变量 v 是一个局部变量，**for** 循环结束后就不能再使用了。如果你需要使用这个值，可以在循环体退出之前将它赋值给另一个变量。

泛化型的 **for** 语句工作在函数的基础上，称之为迭代器。在每一次迭代中，迭代函数会被调用而产生一个新的值，当这个新的值为 **nil** 时，迭代结束。泛化型 **for** 的语法如下：

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

namelist ::= Name {',' Name}

下面这个 **for** 语句

for var_1, ..., var_n in explist do block end

等同于以下代码：

```
do
   local f, s, var = explist
   while true do
      local var_1, ···, var_n = f(s, var)
      if var_1 == nil then break end
      var = var_1
      block
   end
end
```

注意以下几点：

- explist（表达式列表）只会进行一次求值。它们的结果是迭代函数、状态和迭代变量初始值。
- f，s 和 var 是隐形变量。在这里显示只是为了方便解释的目的。
- 你可以使用 **break** 语句来退出 **for** 循环。
- 循环变量 var_i 是一个局部变量，**for** 循环结束后就不能再使用了。如果你需要使用这个值，可以在循环体退出之前将它赋值给另一个变量。

3.3.6 - 函数调用语句

考虑到潜在的边际效应（side-effects），函数调用会被当作下面的语句执行：

stat ::= functioncall

这种情况下，所有的返回值都会被抛弃。函数调用在 **3.4.9** 中有解释。

3.3.7 - 局部声明

局部变量可以在语句块中的任意地方声明。声明时可以给变量赋上一个初始值。

stat ::= **local** namelist [‘=’ explist]

如果指定了初始值，那么赋初值的行为和多重赋值是一样的（参考 **3.3.3**）。否则，所有变量都被初始化为 **nil** 值。

一个代码块也是一个语句块（参考 **3.3.2**），所以一个局部变量也可以在代码块中的任意语句块外声明。

局部变量的可视规则在 **3.5** 里有解释。

3.4 - 表达式

以下是 Lua 的基本表达式：

exp ::= prefixexp

exp ::= nil | false | true

exp ::= Number

exp ::= String

exp ::= functiondef
exp ::= tableconstructor
exp ::= '...'
exp ::= exp binop exp
exp ::= unop exp
prefixexp ::= var | functioncall | '(' exp ')'

数字和字符串在 **3.1** 中有解释。变量在 **3.2** 中有解释。函数定义在 **3.4.10** 中有解释。函数调用在 **3.4.9** 中有解释。表的构造在 **3.4.8** 中有解释。可变参数表达式使用三个点（...）来表示，它只能在不定参数的函数中使用，这个会在 **3.4.10** 中有解释。

二元操作符包括算术操作符（参考 **3.4.1**），关系操作符（参考 **3.4.3**），逻辑操作符（参考 **3.4.4**），以及连接操作符（参考 **3.4.5**）。一元操作符包括负号（参考 **3.4.1**），逻辑非操作符（参考 **3.4.4**）和长度操作符（参考 **3.4.6**）。

函数调用和变参表达式都可以返回多个值。如果一个函数调用只是作为一个语句（参考 **3.3.6**），那么它的返回值列表会被调整为 0 个元素，因此会丢弃掉所有的返回值。如果一个表达式是作为表达式列表中的最后一个（或者是唯一一个）元素的话，那么表达式返回值个数不作调整（除非表达式被小括号括住了）。其它的所有情况，Lua 都会将结果列表调整为 1 个元素。如果有多个返回结果，那么除了第一个值，其它值都会被丢弃。如果没有返回值，那就会将 **nil** 作为返回值。

下面是一些例子：

```
f()                 -- 调整为 0 个返回值
g(f(), x)           -- f() 被调整为只有 1 个返回值
g(x, f())           -- g 接收 x 及 f()返回的所有结果
a,b,c = f(), x      -- f() 被调整为只有 1 个返回值 (c 的值为 nil)
a,b = ...           -- a 被赋值为变参列表中的第一个值，b 被赋值
                    -- 为变参列表中的第二个值 （如果变参列表
                    -- 中没有对应的值，那 a 和 b 也会是 nil
a,b,c = x, f()      -- f() 被调整为 2 个返回值
a,b,c = f()         -- f() 被调整为 3 个返回值
return f()          -- 返回 f()的所有结果
return ...          -- 返回接收到的所有变参
return x,y,f()      -- 返回 x，y 和 f()返回的所有结果
{f()}               -- 使用 f()返回的所有结果创建一个列表
{...}               -- 使用所有变参创建一个列表
{f(), nil}          -- f() 被调整为 1 个返回值
```

任意的表达式只要被小括号括住，那么它只能返回 1 个值。因此，(f(x,y,z))总是代表 1 个值，就算 f 返回多个值。（(f(x,y,x))的值是 f 返回的第一个值，如果 f 没有返回值，那它的值就是 **nil**。）

3.4.1 - 算术操作符

Lua 支持常用的算术操作符：二元的+（加），-（减），*（乘），/（除），%（求模）和^（求幂）。一元的-（负号）。如果它们的操作数是数字或可以转为数字的字符串（参考 **3.4.2**），那么它们的含义就是普通的算术运算。求幂操作可以使用任意的指数，比如 x^(-0.5)表示 x 开方的倒数。求模操作被定义为：

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

也就是商向无穷小方向取整时，a 除以 b 的余数。

3.4.2 - 强制转换

Lua 在运行时支持字符串与数字之间的自动转换。对字符串进行任意的算术运算都会尝试按照词法分析器的规则（字符串可以有前缀、后缀和标记）将该字符串转成数字。相反地，数字也会根据需要转成合理格式的字符串。想要更好地控制数字转字符串，使用字符串库中的 format 函数。（参考 **string.format**）

3.4.3 - 关系操作符

Lua 中的关系操作有：

== ~= < > <= >=

这些操作符的运算结果总是 **false** 或 **true**。

等号（==）首先比较操作数的类型，如果类型不同，那么结果就是 **false**，否则比较操作数的值。数字与字符串按照普通规则进行比较。table，userdata 和协程比较的是它们的引用：如果它们引用的是同一个对象，那就认为它们是相等的。每当你创建一个新的对象时（一个 table，userdata 或协程），这个新的对象都与之前已存在的对象不同。含有相同引用的闭包总是相等的，只要有任何差别（不同的行为，不同的定义）就不相等。

你可以使用“eq”元方法来改变 table 和 userdata 的相等比较含义（参考 **2.4**）。

3.4.2 中的转换规则并不适用于相等比较操作。因此，”0” == 0 的结果是 **false**，t[0]和 t[“0”]代表的是 table 中不同的入口。

不等操作符~=和相等操作符（==）是完全相反的。

大小比较操作符的比较规则如下：如果两个操作数都是数字，那么它们依照数字的大小规则进行比较。否则，如果两个操作数都是字符串，那么它们依照字符串的本地化规则进行比较。否则，Lua 尝试调用“lt”和“le”元方法（参考 **2.4**）。a > b 会被翻译为 b < a，a >= b 会被翻译为 b <= a。

3.4.4 - 逻辑操作符

Lua 中的逻辑操作符是 **and**，**or** 和 **not**。和控制结构一样（参考 **3.3.4**），所有的逻辑操作符认为 **false** 和 **nil** 表示假，其它值表示真。

“非”操作符 **not** 总是返回 **false** 和 **true**。“与”操作符 **and** 判断第一个操作数是否为 **false** 或 **nil**，是则返回第一个操作数，否则返回第二个操作数。“或”操作符 **or** 判断第一个操作数如果不是 **nil** 和 **false** 则返回第一个操作数，否则返回第二个操作数。**and** 和 **or** 都是使用简捷（short-cut）求值的策略，即第二个操作数只有在必须的时候才会进行求值。下面是一些例子：

```
10 or 20                --> 10
10 or error()           --> 10
nil or "a"              --> "a"
nil and 10              --> nil
```

```
false and error()      --> false
false and nil          --> false
false or nil           --> nil
10 and 20                --> 20
```
(本参考手册中，-->指示出它左边表达式的结果。)

3.4.5 - 连接符

Lua 中的字符串连接符用两个点（..）来表示。如果两个操作数是字符串或数字，那么它们会按照 **3.4.2** 中提示的规则转换为字符串。否则会去调用__concat 元方法（参考 **2.4**）。

3.4.6 - 长度操作符

长度操作符为一元操作符#。字符串的长度是它的字节数（即当每个字符是一个字节时的传统长度含义）。

程序可以通过__len 元方法（参考 **2.4**）去修改除了字符串外的其它数值求长度操作的行为。

除非定义了__len 元方法，否则只有当表是一个顺序表时它的长度才能明确，也就是说，表的正整数键值是{1..n}，这种情况下，n 就是表的长度。如果一个表为

{10, 20, nil, 40}

这样的表不是一个顺序表，因为键 4 有值但键 3 没有值。（这个表没有像{1..n}这样的正整数键值。）注意，非数字类型的键不会影响到一个表是否是一个顺序表。

3.4.7 - 优先级

Lua 操作符的优先级如下表，优先级由低至高：

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

和别的语言一样，你可以使用小括号来改变表达式中的优先级。连接操作符（‘..’）和求幂操作符（‘^’）是右结合。其它的二元操作符都是左结合。

3.4.8 - table 构造

table 的构造器指的是创建 table 的表达式。每当对构造器进行求值，一个新的 table 就被创建了。构造器可以创建一个空表或带有初始域的表。构造器的一般语法是：

tableconstructor ::= ‘{‘ [fieldlist] ’}’

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

field ::= '[' exp ']' '=' exp | exp | Name '=' exp | exp

fieldsep ::= ',' | ';'

每一个形式如[exp1] = exp2 的域都会向新的表添加一个入口，它的键是 exp1，值是 exp2。形式 name = exp 的域等同于形式["name"] = exp 的域。最后，形式为 exp 的域等同于[i] = exp，i 是从 1 开始的连续的整数，其它形式的域不会影响到 i 的计数。比如：

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

等同于

```
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
```

如果表中最后一个域的形式是一个函数调用表达式或者是一个变参表达式，那么这个表达式返回的所有值都会紧接着前面的域而进入到表中（参考 **3.4.9**）。

在域列表的最后可以添加一个可选的分隔符，这样做是为了方便代码的机器生成。

3.4.9 - 函数调用

Lua 的函数调用语法如下：

functioncall ::= prefixexp args

函数调用时，首先会对 prefixexp 和 args 进行求值。如果 prefixexp 的值的类型是 function，那么就会用给定的参数来调用这个函数。否则就会调用 prefixexp 的“call”元方法，它的第一个参数是 prefixexp 的值，其它参数为原始调用的参数（参考 **2.4**）。

形式

functioncall ::= prefixexp ':' Name args

通常也被叫做“方法”。v:name(args)的调用形式是 v.name(v, args)调用形式的一个语法糖，不同之处在于 v 只进行一次求值。

参数的语法为：

args ::= '(' [explist] ')'

args ::= tableconstructor

args ::= String

所有的参数表达式都会在函数调用之前进行求值。调用形式 f{fields}是 f({fields})的一种语法糖，前提是函数的参数列表中只有一个新创建的 table；调用形式 f[[string]]是 f('string')的一种语法糖，前提是函数的参数列表中只有一个字符串。

return functioncall 这样的调用形式叫做尾调用。Lua 实现了真正的尾调用（或叫真正的尾递归）：在尾调用中，被调用函数重用了调用函数的栈的入口。因此，程序执行的尾调用不受嵌套数量限制。不过，尾调用也会清除调用函数的调试信息。要注意的是，只有当 **return**

语句返回的仅仅只有一个函数时才叫尾调用，这种语法使得调用函数返回的内容就是被调用函数返回的内容。所以，下面的例子都不是尾调用：

```
return (f(x))          -- 返回值数量被调整为 1，返回的不是被调用函数的所有结果
return 2 * f(x)        -- 返回值与被调用函数的返回值不一致
return x, f(x)         -- 有附加的返回值
f(x); return           -- 结果被丢弃了
return x or f(x)       -- 返回值数量被调整为 1，有可能与被调用函数的返回值不一致
```

3.4.10 - 函数定义

函数定义的语法是：

functiondef ::= **function** funcbody

funcbody ::= ‘(’ [parlist] ‘)’ block **end**

下面的语法糖简化了函数的定义：

stat ::= **function** funcname funcbody

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

funcname ::= Name {‘.’ Name} [‘:’ Name]

语句：

function f () body end

转化为：

f = function () body end

语句：

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

转化为

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

语句：

local function f () body end

转化为：

local f; f = function () body end

而不是：

local f = function () body end

（仅仅当函数体中包含有引用 f 时，这两种方式是不同的）

一个函数的定义是一个可执行的表达式，它的值的类型是 function。当 Lua 预编译一个代码块时，当中的所有函数体也都会被预编译。不管 Lua 在何时执行函数的定义，该函数被实例化（或闭合）。该函数的实例（或闭包）就是表达式最终的值。

函数的形参是作为局部变量被实参初始化的：

parlist ::= namelist [‘,’ ‘...’] | ‘...’

当函数被调用时，除了函数是一个变参函数（形参列表的最后是’...’）的情况外，实参的数量会被调整为与形参的数量一致。变参函数不会调整它的实参列表，而是将所有额外的参数通过变参表达式（也是用三个点来表示）来传递给函数。变参表达式的值是一个由所有额外参数组成的列表，和返回多个值的函数一样。如果变参表达式被用在另一个表达式中或处于表达式列表中间的位置，那么它的返回值将被调整为一个元素。如果变参表达式被用作表达式列表中的最后一个元素，那么它的返回列表不会被调整（除非它被小括号括住了）。

作为示例，请看下面的函数定义：

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

下面是实参到形参和变参表达式的映射：

```
调用                形参

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
```

使用 **return** 语句来返回结果（参考 **3.3.4**）。如果执行函数时一直到函数体的最后都没有遇到 **return** 语句，那么该函数将不返回任何值。

函数可以返回的结果个数与系统相关，但至少保证可以返回 1000 个值。

冒号语法用来定义方法，即带有隐藏的额外参数 self 的函数。因此，语句：

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

是下面语句的语法糖：

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

3.5 - 可视规则

Lua 是一种词法定界（lexical scoping）的语言。局部变量的作用域从声明它的语句后开始，一直到它所在语句块的最后一个非空语句结束。请看下面的例子：

```
x = 10                    -- 全局变量
do                        -- 新的语句块
  local x = x             -- 局部变量'x', 初始值为 10
  print(x)                --> 10
  x = x+1
  do                      -- 另一个语句块
    local x = x+1         -- 另一个'x'
    print(x)              --> 12
  end
  print(x)                --> 11
end
print(x)                  --> 10  (全局的那个)
```

注意，像 local x = x 这样的声明，新的 x 由于正在声明，它还没有产生作用域，因此第二个 x 引用的是外部的变量。

由于词法定界规则的原因，在局部变量作用域内定义的函数可以自由地访问这些局部变量。一个被内部函数使用的局部变量叫作上值（upvalue），或者叫内部函数的外部局部变量。

注意，每次执行 **local** 语句都会定义一个新的局部变量。请看下面的例子：

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

循环体一共创建了 10 个闭包（也就是 10 个匿名函数的实例）。每个闭包使用的是不同的局部变量 y，同时它们共享一个外部局部变量 x。

4. 应用程序接口（API）

本节说明 Lua 的 C API，即那些开放出来用于宿主程序与 Lua 程序交互的 C 函数。所有的 API 函数和相关的类型、常量都在 lua.h 这个头文件中声明。

虽然我们使用了“函数”这个术语，但其实任何便利的 API 都有可能只是个宏。除非另有说明，所有的这些宏都只使用它们的参数一次（除了第一个参数，这个参数常常是一个 Lua 状态机），因此不会产生隐藏的边际效应。

和大部分的 C 库一样，Lua 的 API 函数不会检查它们参数的合法性或一致性。不过你可以通过打开 LUA_USE_APICHECK 这个宏来编译 Lua，从而改变这个行为。

4.1 - 栈

Lua 使用一个虚拟栈来把值从 C 传出和把值传入 C。栈中的每个元素都是 Lua 的一种值（空值，数字，字符串等）。

每当 Lua 调用 C 时，被调用的 C 函数都会得到一个新的栈，这个栈独立于之前的栈和那些仍处于活动状态的 C 函数的栈。这个栈在一开始就包含了所有传递给 C 函数的那些参数，同时 C 函数也将那些要返回给调用函数的结果压到这个栈中（参考 **lua_CFunction**）。

为了方便，大部分查询操作的 API 不会严格遵守栈的规则，它们使用索引来访问栈中的任意元素。正的索引值表示栈中的绝对位置（从 1 开始）。负的索引值表示相对于栈项的位置。更具体地说，如果一个栈有 n 个元素，那么索引值 1 表示栈中的第一个元素（即第一个被压栈的元素），索引值 n 表示最后一个元素。索引-1 也表示最后一个元素（即栈顶的那个元素），索引-n 表示第一个元素。

4.2 - 栈大小

当你使用 Lua 的 API 来进行交互时，你要保证栈的一致性，尤其是要防止栈溢出。你可以使用 **lua_checkstack** 这个函数来确保栈中有足够的额外空间可以让新的元素进栈。

当 Lua 调用 C 的时候，Lua 会保证栈中至少有 LUA_MINSTACK 个额外的空间。LUA_MINSTACK 的值是 20，所以你不必担心栈的空间，除非你的代码循环地将元素压进

栈。

当你调用 Lua 函数时没有指定返回值的个数（参考 **lua_call**），Lua 会确保有足够的空间来保存所有的返回值，但不会有任何额外的空间，所以，在经过这样的调用之后想要压东西进栈，你都应该在压栈前使用 **lua_checkstack** 来作检查。

4.3 - 合法的和可接受的索引

任何以栈索引作为输入参数的 API 函数只有接收到合法的或可接受的索引时才能正确工作。

一个合法的索引是指那种指向栈中真实位置的索引，它的值在 1 至栈顶之间（1 ≤ abs(index) ≤ top）。那些可以修改索引对应的值的函数通常需要接收合法的索引。

除非另有说明，那些接受合法索引的函数也可以接受伪索引，伪索引指向那些不在栈中的，但是可以被 C 代码访问的值。伪索引用来获取 Lua 的注册表和 C 函数的上值。（参考 **4.4**）。

那些仅仅需要一个值而不管这个值的位置是否处于栈中的函数可以使用那些可接受的索引。一个可接受的索引可以是任意的合法索引，包括伪索引，同时也可以是大于栈顶值的正索引，即指向栈顶之上的位置。（注意 0 不是一个可接受的索引）除非另有说明，API 中的函数都是使用可接受的索引。

可接受的索引可以在查询栈时避免一些额外的检查。举个例子，一个 C 函数可以直接查询它的第三个参数而不需要先确定它是否有第三个参数，也就是说不用判断 3 是否是一个合法的索引。

因为调用函数时可以使用可接受的索引，所以任何非法的索引都可以看作是指向一种和空值很像的，类型为 LUA_TNONE 的虚拟值。

4.4 - C 闭包

当创建一个 C 函数时，它可以与一些值关联在一些，这样就创建了一个 C 闭包（参考 **lua_pushcclosure**）。这些值叫上值（upvalues），C 函数被调用时可以访问这些值。

当 C 函数被调用时，它的上值会被放在特殊的伪索引上。这些伪索引由宏 **lua_upvalueindex** 产生。第一个与函数关联的值在栈中的位置为 lua_upvalueindex(1)上，依此类推。如果 lua_upvalueindex(n)中的 n 大于当前函数的上值的数量（但不大于 256），那么它产生一个可接受的索引而不是合法的索引。

4.5 - 注册表

Lua 提供了一个注册表，这是一个预先定义好的 table，它可以被 C 代码用来存放任意的 Lua 值。这个注册表的位置在合法的伪索引 LUA_REGISTRYINDEX 所指的位置上。C 库中的任意函数都可以将数据存放在这个表上，但是要小心选择一个不与表中已存在的键冲突的键。通常，你应该使用包含有你的库名的字符串或你代码中保存 C 对象地址的 light userdata 或你代码中创建的任意 Lua 对象作为键。和全局变量名一样，以下划线开头，后接大写字母的键名是 Lua 的保留键名。

注册表中的整型键被用来实现引用机制，它由辅助库和一些预定义的值实现。因此，整型的键不能用作其它目的。

当你创建一个 Lua 状态机（Lua state）时，它的注册表中就有一些预定义的值。这些预定义的值可以被 lua.h 中定义的整型常量键索引。下面是这些常量：

- **LUA_RIDX_MAINTHREAD**：此索引指向 Lua 的主线程。（主线程在创建 Lua 状态机时一起被创建）
- **LUA_RIDX_GLOBALS**：此索引指向全局环境表。

4.6 - C 中的错误处理

Lua 本质是使用 C longjmp 的便利性来处理错误的。（如果把 Lua 当作 C++来编译，你也可以使用异常 exceptions。在源代码里搜索 LUAI_THROW）当 Lua 遇到错误（比如内存分配错误，类型错误，语法错误和运行时错误）时，它唤起一个错误，也就是做一个长跳转（long jump）。保护环境下使用 setjmp 来设置一个恢复点，任何错误都会跳转到离它最近的恢复点。

如果一个错误发生在保护环境之外，Lua 会调用恐慌函数（参考 **lua_atpanic**）然后调用 abort 中止，即退出主应用程序。不过恐慌函数可以不返回从而避免退出（比如执行一个长跳转到你自己的恢复点）。

恐慌函数就像消息处理机（参考 **2.3**），错误消息就在栈顶。不过，这里不保证栈中有足够的空间来存放错误。恐慌函数在将数据压入栈前应该先检查一下栈中是否有足够的空间（参考 **4.2**）。

大部分的 API 函数都可以抛出一个错误，比如内存分配出错时会抛出错误。每个函数的规格说明都会指出它是否可以抛出错误。

C 函数中可以调用 **lua_error** 来抛出错误。

4.7 - C 中的 yields 处理

Lua 本质是使用 C longjmp 的便利性来 yield（因为 yield 不好翻译为中文，所以下面直接使用 yield 这个词吧，能理解它的意思是退出协程就行了）协程的。因此，如果一个叫 foo 的 C 函数调用了一个 API 函数，并且这个 API 函数执行了 yield（直接地或通过调用另一个函数间接执行 yield），那么 Lua 就不能再返回到 foo 了，这是因为 longjmp 会破坏 C 的调用堆栈。为了避免这种问题，当 Lua 尝试在一个 API 中调用 yield 时会抛出一个错误，但下面三个 API 除外：**lua_yieldk**，**lua_callk** 和 **lua_pcallk**。这些函数接收一个叫“继续函数”的参数（参数名叫 k），在 yield 之后会继续执行这个“继续函数”。

我们需要一些专有的术语来解释什么是“继续”。有一个被 Lua 调用的 C 函数，我们叫它原始函数。这个原始函数调用上面提到的三个函数中的一个，我们叫它作被召函数（callee function），它会 yields 当前线程。（分两种情况，一，被召函数是 **lua_yieldk**。二，被召函数是 **lua_callk** 或 **lua_pcallk** 并且被他们自己调用。）

假设当前运行的线程在执行被召函数时 yields 了。当线程恢复时被召函数会结束运行。但是被召函数无法返回到原始函数，因为它在 C 堆栈中的结构被 yield 破坏了。这时 Lua 会调用“继续函数”，它是被召函数的一个参数。像它名字暗示的一样，“继续函数”会继续执行原始函数的任务。

Lua 像处理原始函数一样处理“继续函数”。在同一个状态机中，如果被召函数返回了，“继续函数”会把原始函数的栈继承过来。（举个例子，调用 **lua_callk** 后，函数及函数的参数会从栈中移除，它们会被调用的返回结果所替换。）它也有相同的 upvalue。Lua 处理它的

返回值就像处理原始函数的返回值一样。

在 Lua 状态机中，原始函数与“继续函数”唯一的不同是调用 **lua_getctx** 的返回结果。

4.8 - 函数与类型

下面我们按照字母表的顺序列出所有的 C API 函数和类型。每一个函数都会有一个类似这样的说明：[-o,+p,x]

其中第一部分 o 表示有多少个元素会被函数抛出栈。第二部分 p 表示有多少个元素被函数压进栈。（任何函数都总是先把它的参数抛出去再把它的结果压进来的。）x|y 这种形式表示函数可能抛出或压入 x 或 y 个元素，这是由特定情况决定的。’?’表示我们无法从函数的参数中知道有多少个元素会被抛出/压入栈（它们有可能由栈中的数据决定）。第三部分 x 表示函数是否会抛出错误：’-’表示函数不会抛出错误；’e’表示函数会抛出错误;’v’表示函数会抛出带有目的性的错误。

lua_absindex

[-0, +0, –]
int lua_absindex (lua_State *L, int idx);

将可接受的索引 idx 转换为绝对索引（即不由栈顶决定的索引）。

lua_Alloc

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

Lua 状态机使用的内存分配器的函数类型。分配器函数必须提供和 realloc 类似的功能，但不一定要完全一样。它的参数有：ud，一个传给 **lua_newstate** 的指针；ptr，一个指向被 allocated/reallocated/freed 的数据块的指针；osize，数据块的原始大小或分配信息数据。nsize，分配后的数据块的大小。

当 ptr 为非 NULL（空）时，osize 就是 ptr 指向的数据块的大小，即是说，当分配或重分配内存时，应该指明数据块的大小。

当 ptr 是 NULL 时，osize 代表着 Lua 分配的某种对象类型。当（仅当）Lua 创建 LUA_TSTRING, LUA_TTABLE, LUA_TFUNCTION, LUA_TUSERDATA, 或者 LUA_TTHREAD 这些类型的对象时，onsize 的值就是这些宏的值。当 osize 是其它值时，则 Lua 是在分配内存或者其它东西。

Lua 假设分配器函数有以下行为：

当 nsize 是 0 是，分配器的行为像 free 函数并且返回 NULL。

当 nsize 不是 0 时，分配器的行为应该像 realloc 函数。当且仅当分配器无法实现分配请求时返回 NULL。Lua 假设当 osize >= nsize 时永远不会分配失败。

下面有一个分配器函数的简单实现。它被辅助库中的 **luaL_newstate** 所使用。

```
static void *l_alloc (void *ud, void *ptr, size_t osize, size_t nsize) {
  (void)ud;  (void)osize;  /* 不使用 */
  if (nsize == 0) {
    free(ptr);
    return NULL;
  }
  else
    return realloc(ptr, nsize);
}
```

注意，标准 C 确保 free(NULL)不起任何作用以及 realloc(NULL, size)等同于 malloc(size)。上面的代码假设 realloc 在收缩数据块时不会失败。（虽然标准 C 没有保证该行为，它像一个安全的假设。）

lua_arith

[-(2|1), +1, e]
void lua_arith (lua_State *L, int op);

对栈顶的两个值（如果是做求反运算则是一个值）做数学运算，栈顶的那个值会作为第二个操作数。运算后这些值会被抛出栈，运算的结果会被压入栈。这个函数遵守的规则与对应的 Lua 运算符的规则一致（即有可能会触发元方法）。

op 的值必须是下面的一种：

- **LUA_OPADD**:加法(+)
- **LUA_OPSUB**:减法 (-)
- **LUA_OPMUL**:乘法 (*)
- **LUA_OPDIV**:除法 (/)
- **LUA_OPMOD**:求模 (%)
- **LUA_OPPOW**:求幂 (^)
- **LUA_OPUNM**:求反 (一元的 -)

lua_atpanic

[-0, +0, –]
lua_CFunction lua_atpanic (lua_State *L, lua_CFunction panicf);

设置一个新的恐慌函数并且返回旧的恐慌函数。（参考 **4.6**）

lua_call

[-(nargs+1), +nresults, e]
void lua_call (lua_State *L, int nargs, int nresults);

调用一个函数。

调用一个函数你必须遵守以下协议：首先，被调用函数被压入栈，接着按顺序将函数的将实参压入栈，即第一个参数先压栈。最后你调用 **lua_call**。nargs 指明压入栈的参数的个数。当函数返回时，所有的参数和函数都会从栈中抛出。返回的结果的个数会被调整为 nresults，除非 nresults 是 LUA_MULTRET，这种情况下，函数返回的所有结果都会被压入栈。Lua 会保证有足够的栈空间来存放这些返回值。函数的结果会按顺序压入栈（第一个返回的结果先入栈），因此，调用完毕后，最后一个结果在栈顶。

被调函数中的任何错误都会被向上传递（通过 longjmp）。

下面的例子展示了主程序如何等价实现 Lua 代码：

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

下面是 C 代码：

```
lua_getglobal(L, "f");                  /*  获得被调用的函数  */
lua_pushstring(L, "how");               /*  第一个参数  */
lua_getglobal(L, "t");                  /*  被索引的表  */
lua_getfield(L, -1, "x");               /*  将 t.x (第二个参数)  入栈*/
lua_remove(L, -2);                      /*  将 t 从栈中移除  */
lua_pushinteger(L, 14);                 /*  第三个参数*/
lua_call(L, 3, 1);                      /*  调用 f，指明有 3 个输入参数和 1 个返回值  */
lua_setglobal(L, "a");                  /*  设置全局变量'a'的值  */
```

注意上面的代码是“平衡”的：在最后，栈恢复到了它调用函数前的状态。这是一种好的编程习惯。

lua_callk

```
[-(nargs + 1), +nresults, e]
void lua_callk (lua_State *L, int nargs, int nresults, int ctx,
                lua_CFunction k);
```

这个函数的行为和 **lua_call** 完全一样，但是它允许被调用函数 yield（参考 **4.7**）。

lua_CFunction

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

C 函数的类型。

为了实现与 Lua 的正确沟通，C 函数必须遵守以下参数与返回值的协议：C 函数从 Lua 中按顺序接收参数（第一个参数首先压栈）。因此，当函数开始时，lua_gettop(L)返回函数接收到的参数的个数。第一个参数（如果有的话）在栈中的位置索引为 1，最后一个参数在栈中的位置索引为 lua_gettop(L)。为了将值返回给 Lua 代码，C 函数首先要将它们按顺序压入栈（第一个结果首先压栈），然后返回结果的个数。除了返回结果外的其它数值会被 Lua 正确的抛弃。和 Lua 函数一样，被 Lua 调用的 C 函数也可以返回多个值。

作为一个例子，下面的函数接收不定数量的数字作为参数并返回它们的平均值与总值：

```
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);          /* 第一个结果*/
    lua_pushnumber(L, sum);            /* 第二个结果 */
    return 2;                          /* 结果的个数*/
  }
```

lua_checkstack

[-0, +0, –]
int lua_checkstack (lua_State *L, int extra);

确保栈中有足够的空闲空间。如果它因为请求空间过大（通常最至少支持好几千个元素）或无法分配足够的内存，那么它就会返回 false。该函数不会缩小栈空间。如果栈的大小已经大于请求的大小，那么它会保持不变。

lua_close

[-0, +0, –]
void lua_close (lua_State *L);

摧毁指定 Lua 状态机的所有对象（如果对应的垃圾回收元方法就调用它）和释放该 Lua 状态机使用的所有动态内存。在多数平台上，你不必去调用这个函数，因为所有的资源都会在主程序结束的时候被释放。不过在另一方面，那些会创建多个状态机的长期运行的程序，比如守护进程和 web 服务，当不再需要某个 Lua 状态机的时候你就可以去关闭它。

lua_compare

[-0, +0, e]
int lua_compare (lua_State *L, int index1, int index2, int op);

比较两个 Lua 数值。当 index1 处的值与 index2 处的值满足 op 的比较规则（规则与对应的 Lua 运算符的规则，即有可能会触发元方法）时返回 1，否则返回 0。如果索引是非法的，那也会返回 0。

op 的值必须是下面的一种：

- **LUA_OPEQ**:相等比较 (==)
- **LUA_OPLT**:小于比较 (<)
- **LUA_OPLE**:小于等于比较 (<=)

lua_concat

[-n, +1, e]
void lua_concat (lua_State *L, int n);

连接栈顶处的 n 个值，从栈中抛出它们并且将结果压入栈。如果 n 是 1，那么结果就是栈顶的值（也就是函数什么都不做）；如果 n 是 0，那么结果就是空字符串。连接操作遵守 Lua 的一般语法规则（参考 **3.4.5**）。

lua_copy

[-0, +0, –]
void lua_copy (lua_State *L, int fromidx, int toidx);

将 fromidx 索引处的值复制到 toidx 索引处，此操作不会改变其它位置的元素（只替换掉那个位置的值）。

lua_createtable

[-0, +1, e]
void lua_createtable (lua_State *L, int narr, int nrec);

创建一个空的 table 并将其压入栈。参数 narr 指明这个 table 的数组部分有多少个元素；参数 nrec 指明这个 table 的其它部分有多少个元素。Lua 可以使用这两个参数来为一个新的 table 预分配内存。这种预分配在你预先知道 table 有多少个元素时很有用，不然的话你可以使用 **lua_newtable**。

lua_dump

[-0, +0, e]
int lua_dump (lua_State *L, lua_Writer writer, void *data);

将一个函数转化为一个二进制代码块。它根据栈顶的 Lua 函数产生一个二进制代码块，如果加载这个代码块，那么它作为一个函数是等同于转化前的那个函数的。由于它是代码块，**lua_dump** 可以调用函数 writer（参考 **lua_Writer**）来将它们写到 data 中。

返回值是一个错误码，它由最后一次调用的 writer 返回。0 表示没有错误。

这个函数不会将 Lua 函数抛出栈。

lua_error

[-1, +0, v]
int lua_error (lua_State *L);

生成一个 Lua 错误。这个错误消息（可以是 Lua 的值或者其它类型）一定要放在栈顶。此函数会执行一个长跳转，因此这里没有返回值（参考 **luaL_error**）。

lua_gc

[-0, +0, e]
int lua_gc (lua_State *L, int what, int data);

控制垃圾回收。
此函数可以执行多个任务，视参数 what 而定：

- **LUA_GCSTOP**:停止垃圾回收器的工作。
- **LUA_GCRESTART**:重启垃圾回收器。
- **LUA_GCCOLLECT**:执行一次垃圾全回收。
- **LUA_GCCOUNT**:返回当前被 Lua 使用的内存的大小（单位为千字节 Kbytes）。
- **LUA_GCCOUNTB**:返回当前被 Lua 使用的内存大小的不足千字节的那部分字节数。比如当前 Lua 使用的内存大小为 234567 字节，那么 LUA_GCCOUNT 返回的是 229（234567/1024），而 LUA_GCCOUNTB 返回的是 71（234567-1024*229）。
- **LUA_GCSTEP**:执行一步增量的垃圾回收流程。步长的大小由参数 data 决定（数值越大意味着更多的步数）。如果你想控制步长的大小，你必须自己调整 data 的大小以找到一个合适的数值。如果当前步执行后整个回收循环也结束了，那么函数返回 1。
- **LUA_GCSETPAUSE**:将 data 的值设为回收器的新的“暂停”值（参考 **2.5**）。函数返回之前的“暂停”值。
- **LUA_GCSETSTEPMUL**:将 data 的值设为步进式乘法器（step multiplier）的新的值（参考 **2.5**）。函数返回之前的步进式乘法器的值。
- **LUA_GCISRUNNING**:返回一个布尔型的值以表示回收器是否正在工作（也就是没有停止）。
- **LUA_GCGEN**:将回收器的工作方式转为世代式（参考 **2.5**）。
- **LUA_GCINC**:将回收器的工作方式转为增量式。这是默认方式。

更多关于这些选项的细节请参考 **collectgarbage**。

lua_getallocf

[-0, +0, –]
lua_Alloc lua_getallocf (lua_State *L, void **ud);

返回指定 Lua 状态机的内存分配函数。如果 ud 不是 NULL，那么 Lua 会将传给 lua_newstate 的指针存入*ud 中。

lua_getctx

[-0, +0, –]
int lua_getctx (lua_State *L, int *ctx);

此函数由“继续函数（continuation function）”（参考 **4.7**）调用以获得线程的状态和上下文信息。

当在原始函数中调用 **lua_getctx** 时，它的返回值为 LUA_OK 且不会修改参数 ctx 的值。当在继续函数中调用 **lua_getctx** 时，它的返回值为 LUA_YIELD 且将 ctx 设为上下文信息（与继续函数一同传给被召函数的那个 ctx 参数）。

当被召函数是 **lua_pcallk** 时，Lua 会调用它的继续函数来处理错误。也就是说，当调用 **lua_pcallk** 时发生了错误，Lua 不会返回到原始函数而是去调用继续函数。这种情况下调用 **lua_getctx** 会返回错误码（该值应该由 **lua_pcallk** 返回）。和 yield 的情况一样，ctx 的值会被设为上下文信息。

lua_getfield

[-0, +1, e]
void lua_getfield (lua_State *L, int index, const char *k);

将 t[k]的值压入栈，t 是由参数 index 指向的值。和在 Lua 代码中一样，此函数会触发“index”事件的元方法（参考 **2.4**）。

lua_getglobal

[-0, +1, e]
void lua_getglobal (lua_State *L, const char *name);

将名字为 name 的全局变量的值压入栈。

lua_getmetatable

[-0, +(0|1), –]
int lua_getmetatable (lua_State *L, int index);

将 index 指向的值的元表压入栈。如果该值没有元表，则此函数返回 0 且不压任何东西入栈。

lua_gettable

[-1, +1, e]
void lua_gettable (lua_State *L, int index);

将 t[k]的值压入栈，t 是由参数 index 指向的值，k 是栈顶的值。
此函数会将键出栈（把结果放在它的位置上）。和在 Lua 代码中一样，此函数会触发“index”事件的元方法（参考 **2.4**）。

lua_gettop

[-0, +0, –]
int lua_gettop (lua_State *L);

返回栈顶元素的索引。因此索引是从 1 开始的，所以返回的结果就是栈中元素的个数（因此返回 0 表示栈是个空栈）。

lua_getuservalue

[-0, +1, –]
void lua_getuservalue (lua_State *L, int index);

将与 index 处的 userdata 相关联的 Lua 值压入栈。Lua 的值必须是表或 **nil**。

lua_insert

[-1, +1, –]
void lua_insert (lua_State *L, int index);

将栈顶的元素移动到栈中的 index 处并且将 index 处及以上的元素向上移动一个位置。调用此函数时不能使用伪索引，因为伪索引所指示的位置不在栈中。

lua_Integer

typedef ptrdiff_t lua_Integer;

给 Lua 的 API 使用的表示有符号整形的类型。
默认情况下它是一个 ptrdiff_t，通常是机器能够正确处理的最大有符号整数。

lua_isboolean

[-0, +0, –]
int lua_isboolean (lua_State *L, int index);

如果 index 处的值是个布尔值则返回 1，否则返回 0。

lua_iscfunction

[-0, +0, –]
int lua_iscfunction (lua_State *L, int index);

如果 index 处的值是个 C 函数则返回 1，否则返回 0。

lua_isfunction

[-0, +0, –]
int lua_isfunction (lua_State *L, int index);

如果 index 处的值是个函数则返回 1（C 函数或 Lua 函数），否则返回 0。

lua_islightuserdata

[-0, +0, –]
int lua_islightuserdata (lua_State *L, int index);

如果 index 处的值是个 light userdata 则返回 1，否则返回 0。

lua_isnil

[-0, +0, –]
int lua_isnil (lua_State *L, int index);

如果 index 处的值是 **nil** 则返回 1，否则返回 0。

lua_isnone

[-0, +0, –]
int lua_isnone (lua_State *L, int index);

如果 index 处的值是个非法值则返回 1，否则返回 0。

lua_isnoneornil

[-0, +0, –]
int lua_isnoneornil (lua_State *L, int index);

如果 index 处的值是非法的或者是 **nil** 则返回 1，否则返回 0。

lua_isnumber

[-0, +0, –]
int lua_isnumber (lua_State *L, int index);

如果 index 处的值是个数字或是一个可转为数字的字符串则返回 1，否则返回 0。

lua_isstring

[-0, +0, –]
int lua_isstring (lua_State *L, int index);

如果 index 处的值是一个字符串或是一个数字（总是可转为字符串的）则返回 1，否则返回 0。

lua_istable

[-0, +0, –]
int lua_istable (lua_State *L, int index);

如果 index 处的值是一个表则返回 1，否则返回 0。

lua_isthread

[-0, +0, –]
int lua_isthread (lua_State *L, int index);

如果 index 处的值是线程则返回 1，否则返回 0。

lua_isuserdata

[-0, +0, –]
int lua_isuserdata (lua_State *L, int index);

如果 index 处的值是 userdata（full 或者 light）则返回 1，否则返回 0。

lua_len

[-0, +1, e]
void lua_len (lua_State *L, int index);

获得 index 处的值的“长度”。在 Lua 代码中等同于“#”操作符（参考 **3.4.6**）。结果会被压入栈。

lua_load

[-0, +1, –]
int lua_load (lua_State *L,
lua_Reader reader,
void *data,
const char *source,
const char *mode);

加载 Lua 代码块（不执行它）。如果没有错误，lua_load 将编译后的代码块作为一个函数放在栈顶，否则将错误信息压入栈。

以下是 lua_load 可能的返回值：

- **LUA_OK**:没有错误。
- **LUA_ERRSYNTAX**:有语法错误。
- **LUA_ERRMEM**:内存分配错误。
- **LUA_ERRGCMM**:执行__gc 元方法时的错误。（这个错误与加载的代码块无关。它由垃圾回收器产生。）

lua_load 函数使用用户提供的读函数来读取代码块（参考 **lua_Reader**）。参数 data 是个要传给读函数的指针。

参数 source 为代码块命名，这对显示错误信息和调试信息很有用（参考 **4.9**）。

lua_load 会自动检测代码块是文本形式还是二进制形式的而进行加载（参考 luac 项目）。参数 mode 的作用和函数 load 中的 mode 一样，如果它的值为 NULL，那就等同于“bt”字符串。

lua_load 的内部会使用到栈，因此读函数在返回时应该将栈恢复到原来未修改的状态。

如果加载后的结果函数有一个 upvalue，那么这个 upvalue 的值会被设到注册表中 LUA_RIDX_GLOBALS 索引处（参考 **4.5**）。如果加载的是主代码块，那么 upvalue 就是

变量_ENV（参考 **2.2**）。

lua_newstate

[-0, +0, –]
lua_State *lua_newstate (lua_Alloc f, void *ud);

创建一个在新的独立的状态机运行的新线程。如果不能创建线程或状态机（由于内存不足）就返回 NULL。参数 f 是分配器函数。该状态机中所有与内存分配相关的操作都通过这个分配器函数来完成。在每次调用分配器函数时都会将第二个参数 ud 传给分配器函数。

lua_newtable

[-0, +1, e]
void lua_newtable (lua_State *L);

创建一个新的空的表，并将其压入栈。它等同于 lua_createtable(L, 0, 0)。

lua_newthread

[-0, +1, e]
lua_State *lua_newthread (lua_State *L);

创建一个新的线程并将其压入栈，返回一个指向代表该新线程的 **lua_State** 指针。新的线程与创建它的原始线程共享一个全局环境，但是它有自己独立的运行栈。

没有函数可以关闭或摧毁一个线程。线程和其它 Lua 对象一样，由垃圾回收机制统一管理。

lua_newuserdata

[-0, +1, e]
void *lua_newuserdata (lua_State *L, size_t size);

此函数根据参数 size 指定的大小分配一块新的内存，将内存的地址与一个新的 full userdata 关联并将该 userdata 压入栈，然后返回内存块的地址。主程序可以自由地使用这一块内存。

lua_next

[-1, +(2|0), e]
int lua_next (lua_State *L, int index);

将一个键从栈中弹出并将参数 index 处的表中的一“键-值对”（指定键之后的“下一对”）压入栈。如果表中不再有“下一对”了，那么 **lua_next** 返回 0（不压任何东西入栈）。

一个典型的表遍历如下：

```
 /* 表在栈中的位置为“t” */
 lua_pushnil(L);  /* 第一个键 */
 while (lua_next(L, t) != 0) {
   /* 使用“键” (位于 index -2) 和 “值” (位于 index -1) */
   printf("%s - %s\n",
            lua_typename(L, lua_type(L, -2)),
            lua_typename(L, lua_type(L, -1)));
   /* 删除“值”，保留“键”以作下一次循环 */
   lua_pop(L, 1);
 }
```

当遍历一个表时不要直接对键调用 **lua_tolstring**，除非你知道该键确实是个字符串。调用 **lua_tolstring** 有可能会改变栈中 index 处的值，这会使下一次调用 **lua_next** 时产生混乱。

参考函数 next 以了解作表遍历时对表作修改应注意的事项。

lua_Number

```
typedef double lua_Number;
```

Lua 的数字类型。默认是 double 类型，但可以在 luaconf.h 中修改它。通过这个配置文件你可以把其它类型定义为 Lua 的数字类型（比如 float 或 long）。

lua_pcall

```
[-(nargs + 1), +(nresults|1), –]
int lua_pcall (lua_State *L, int nargs, int nresults, int msgh);
```

在保护模式下调用一个函数。

参数 nargs 和 nresults 的含义和 **lua_call** 中的一样。如果在调用的过程中没有发生错误，那么 **lua_pcall** 的行为和 **lua_call** 完全一致。不过如果发生了任何错误，**lua_pcall** 会捕捉到它，并把一个值压入栈（错误消息），还会返回一个错误码。和 **lua_call** 类似，**lua_pcall** 也会把函数和它的参数抛出栈。

如果 msgh 为 0，那么返回到栈中的错误消息就是原始的错误消息。否则 msgh 就是一个消息处理函数（message handler）的索引。（在当前版本的实现中，该索引不能为伪索引。）一旦发生运行时错误，该函数就会以错误消息作为参数被调用，它的返回值就是 **lua_pcall** 返回到栈中的消息。

通常这个消息处理函数是用来添加更多的与错误消息相关的调试信息的，比如栈的回溯。这些信息无法在 **lua_pcall** 返回后收集，因为这时栈已经展开（unwound）了。

lua_pcall 返回以下的值（在 lua.h 中定义）：

- **LUA_OK (0)**:成功.

- **LUA_ERRRUN**:运行时错误
- **LUA_ERRMEM**:内存分配错误。对于这种错误，Lua 不会调用消息处理函数。
- **LUA_ERRERR**:执行消息处理函数时的错误。.
- **LUA_ERRGCMM**:执行__gc 元方法时的错误。 (这个错误通常与被调用的函数无关。它由垃圾回收器产生。)

lua_pcallk

[-(nargs + 1), +(nresults|1), –]
int lua_pcallk (lua_State *L,
int nargs,
int nresults,
int errfunc,
int ctx,
lua_CFunction k);

此函数除了允许被调函数可以 yield（参考 **4.7**）外，其它行为和 **lua_pcall** 完全一致，。

lua_pop

[-n, +0, –]
void lua_pop (lua_State *L, int n);

从栈中弹出 n 个元素。

lua_pushboolean

[-0, +1, –]
void lua_pushboolean (lua_State *L, int b);

根据 b 的值将一个布尔类型的值压入栈。

lua_pushcclosure

[-n, +1, e]
void lua_pushcclosure (lua_State *L, lua_CFunction fn, int n);

将一个新的 C 闭包压入栈。

当一个 C 函数被创建时，它有可能会与一些值关联起来，这就叫创建 C 闭包（参考 **4.4**）。当函数被调用时，这些值都可以被函数访问到。为了将这些值与 C 函数关联起来，首先要先将这些值压栈（如果有多个值，第一个值先入栈）。然后调用 **lua_pushcclosure** 来创建 C 函数并将其压入栈，参数 n 指明有多少个值与 C 函数关联。**lua_pushcclosure** 会将这些值弹

出栈。

n 的最大值为 255。

当 n 为 0 时，此函数创建一个轻 C 函数（light C function），它仅仅是一个 C 函数的指针。这种情况下不会抛出内存错误。

lua_pushcfunction

[-0, +1, –]
void lua_pushcfunction (lua_State *L, lua_CFunction f);

将一个 C 函数压入栈。此函数接受一个 C 函数指针，并将一个 Lua 函数类型的值压入栈，当这个 Lua 函数被调用时会触发对应的 C 函数。

任何在 Lua 中注册的 C 函数都必须遵守正确的参数接收与返回值协议（参考 **lua_CFunction**）。

lua_pushcfunction 是一个宏定义：

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

注意 f 使用了两次（译者：哪有两次？）。

lua_pushfstring

[-0, +1, e]
const char *lua_pushfstring (lua_State *L, const char *fmt, ...);

将一个格式化的字符串压入栈并将该字符串的指针返回。此函数和 ANSI C 的 sprintf 函数类似，但也有一些重要的区别：

- 你不需要为字符串分配空间：因为这是一个 Lua 的字符串，Lua 会管理好内存的分配（和通过垃圾回收解除分配）。
- 转换说明符是非常受限制。没有标示，宽度和精度。转换说明符只能是“%%”（插入一个“%”），“%s”（插入一个 0 结尾的字符串，没有大小限制），“%f”（插入一个 **lua_Number** 类型的数字），“%p”（插入一个十六进制格式的指针），“%d”（插入一个整型）和“%c”（插入一个大小为一字节的整型）。

lua_pushglobaltable

[-0, +1, –]
void lua_pushglobaltable (lua_State *L);

将全局环境表压入栈。

lua_pushinteger

[-0, +1, –]

void lua_pushinteger (lua_State *L, lua_Integer n);

将一个整型数字 n 压入栈。

lua_pushlightuserdata

[-0, +1, –]
void lua_pushlightuserdata (lua_State *L, void *p);

将一个 light userdata 压入栈。

在 Lua 里，Userdata 代表 C 数据。一个 light userdata 代表一个指针，一个 void*指针。它是一个值（像一个数字）：你没有创建它，它没有自己的元表，不会被回收（因为它从没被创建）。如果两个 light userdata 它们的 C 地址一样，那就认为它们是相等的。

lua_pushliteral

[-0, +1, e]
const char *lua_pushliteral (lua_State *L, const char *s);

这个宏除了参数 s 只能是字符串字面值（literal string）外，它和 **lua_pushlstring** 是等同的。它会自动计算字符串的长度。

lua_pushlstring

[-0, +1, e]
const char *lua_pushlstring (lua_State *L, const char *s, size_t len);

将由参数 s 指定的字符串的长度为 len 的部分压入栈。Lua 会生成（或重用）这个字符串的一个内部的拷贝，因此，在此函数返回后，s 的内存可以被立刻释放或重用。字符串可以包含任意的二进制数据，包括内嵌的 0。

返回该字符串的内部拷贝的指针。

lua_pushnil

[-0, +1, –]
void lua_pushnil (lua_State *L);

将一个 nil 值压入栈。

lua_pushnumber

[-0, +1, –]
void lua_pushnumber (lua_State *L, lua_Number n);

将一个数字 n 压入栈。

lua_pushstring

[-0, +1, e]
const char *lua_pushstring (lua_State *L, const char *s);

将由参数 s 指定的以 0 结尾的字符串压入栈。Lua 会生成（或重用）这个字符串的一个内部的拷贝，因此，在此函数返回后，s 的内存可以被立刻释放或重用。

返回该字符串的内部拷贝的指针。

如果 s 是 NULL，那就会压入 **nil** 和返回 NULL。

lua_pushthread

[-0, +1, –]
int lua_pushthread (lua_State *L);

将参数 L 所代表的线程压入栈。如果该线程是主线程则返回 1。

lua_pushunsigned

[-0, +1, –]
void lua_pushunsigned (lua_State *L, lua_Unsigned n);

将无符号整型数字 n 压入栈。

lua_pushvalue

[-0, +1, –]
void lua_pushvalue (lua_State *L, int index);

将 index 位置的元素复制一份并将其压入栈。

lua_pushvfstring

[-0, +1, e]
const char *lua_pushvfstring (lua_State *L,
const char *fmt,
va_list argp);

除了用一个 va_list 类型的参数代替不定数量的参数外，此函数等同于 **lua_pushfstring**。

lua_rawequal

[-0, +0, –]
int lua_rawequal (lua_State *L, int index1, int index2);

如果索引 index1 和索引 index2 处的值是原始相等（即不调用元方法的情况下）的就返回 1。否则返回 0。如果有一个索引是非法的也会返回 0。

lua_rawget

[-1, +1, –]
void lua_rawget (lua_State *L, int index);

和 **lua_gettable** 类似，但这是一个原生的访问（即不触发元方法）。

lua_rawgeti

[-0, +1, –]
void lua_rawgeti (lua_State *L, int index, int n);

将 t[n]的值压入栈，t 是索引 index 处的表。这个访问是原生的，也就是说不会触发元方法。

lua_rawgetp

[-0, +1, –]
void lua_rawgetp (lua_State *L, int index, const void *p);

将 t[k]的值压入栈，t 是索引 index 处的表，k 是指针 p 代表的 light userdata。这个访问是原生的，也就是说不会触发元方法。

lua_rawlen

[-0, +0, –]
size_t lua_rawlen (lua_State *L, int index);

返回 index 位置的值的原生“长度”：对于字符串，它是字符串的长度；对于表，它是原生长度操作符（“#”）的运算结果；对于 userdata，它是 userdata 所占内存空间的大小；对于其它值，它是 0。

lua_rawset

[-2, +0, e]
void lua_rawset (lua_State *L, int index);

和 **lua_settable** 类似，但这是一个原生的赋值操作（即不触发元方法）。

lua_rawseti

[-1, +0, e]
void lua_rawseti (lua_State *L, int index, int n);

等同于 t[n] = v 的操作，t 是索引 index 处的表，v 是栈顶的值。
此函数会将栈顶的值弹出栈。此赋值操作是原生的，也就是说不会触发元方法。

lua_rawsetp

[-1, +0, e]
void lua_rawsetp (lua_State *L, int index, const void *p);

等同于 t[k] = v 的操作，t 是索引 index 处的表，k 是指针 p 代表的 light userdata，v 是栈顶的值。
此函数会将栈顶的值弹出栈。此赋值操作是原生的，也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);
```

给 **lua_load** 使用的读函数。每当需要读取代码块的一部分内容时，**lua_load** 都会调用

这个读函数并将参数 data 传给它。读函数必须返回一个指向该部分内容的指针并设置该部分的块大小。该部分在下一次调用读函数前都必须存在。为了标记已经读完了整个代码块，读函数必须返回 NULL 或将 size 设为 0。读函数可以返回任意大小大于 0 的内容。

lua_register

[-0, +0, e]
void lua_register (lua_State *L, const char *name, lua_CFunction f);

将 C 函数设到一个名字为 name 的全局变量中。此函数是一个宏：

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

lua_remove

[-1, +0, –]
void lua_remove (lua_State *L, int index);

将指定的合法索引处的元素删除，该索引之上的值都向下移动一个位置以填补空缺。调用此函数时不能使用伪索引，因为伪索引并不是真正的栈位置。

lua_replace

[-1, +0, –]
void lua_replace (lua_State *L, int index);

将 index 处的元素替换为栈顶的元素并将栈顶元素出栈。

lua_resume

[-?, +?, –]
int lua_resume (lua_State *L, lua_State *from, int nargs);

开启指定线程中的协程。

在开启协程前，你要将主函数及它的参数压栈，然后你调用 **lua_resume**，参数 nargs 是主函数参数的个数。当协程暂停或完成时调用返回。当它返回时，栈上包含所有要传给 **lua_yield** 的值或所有主函数返回的值。如果协程 yields 了，**lua_resume** 返回 LUA_YIELD，如果协程执行完毕且没有发生错误则返回 LUA_OK，如果发生错误则返回错误码（参考 **lua_pcall**）。

如果发生了错误，栈是不会展开的，所以你可以使用调试 API 来做些事情。错误信息放在栈顶。

开启一个协程时，你可以移除任意的由最后一次 **lua_yield** 返回的值，只保留那些你想

传递的值，然后调用 **lua_resume**。

参数 from 代表开启协程 L 的协程。如果没有这个协程，该参数可以填 NULL。

lua_setallocf

[-0, +0, –]
void lua_setallocf (lua_State *L, lua_Alloc f, void *ud);

改变指定状态机的内存分配函数为 f 及用户数据为 ud。

lua_setfield

[-1, +0, e]
void lua_setfield (lua_State *L, int index, const char *k);

等同于 t[k] = v 的操作，t 是索引 index 处的表，v 是栈顶的值。

此函数会将栈顶的值弹出栈。和 Lua 的代码一样，此函数会触发“newindex”事件的元方法（参考 **2.4**）。

lua_setglobal

[-1, +0, e]
void lua_setglobal (lua_State *L, const char *name);

将栈顶元素弹出并将它的值赋值给名字为 name 的全局变量中。

lua_setmetatable

[-1, +0, –]
void lua_setmetatable (lua_State *L, int index);

将栈顶的表弹出并将它设置为 index 处的值的元表。

lua_settable

[-2, +0, e]
void lua_settable (lua_State *L, int index);

等同于 t[k] = v 的操作，t 是索引 index 处的表，v 是栈顶的值，k 是栈顶下面的一个值。

此函数会将键和值都弹出栈。和 Lua 的代码一样，此函数会触发“newindex”事件的元方法（参考 **2.4**）。

lua_settop

[-?, +?, –]
void lua_settop (lua_State *L, int index);

接受任意的索引值或 0，将该索引设为栈顶的索引。如果新的栈顶大于旧的，那么新元素的值都会初始化为 nil。如果 index 为 0，那么栈中的所有元素都会被删除。

lua_setuservalue

[-1, +0, –]
void lua_setuservalue (lua_State *L, int index);

将栈顶的表或 **nil** 弹出栈并将其设为 index 处的 userdata 的关联值。

lua_State

typedef struct lua_State lua_State;

一个直接指向线程和间接（通过线程）指向整个 Lua 解释器状态的结构。The Lua library is fully reentrant: 它没有全局变量。状态机的所有信息都可以通过这个结构来获取。

库里面除了 **lua_newstate**（它创建一个 Lua 状态机）外，每一个函数的第一个参数都是一个指向此结构的指针。

lua_status

[-0, +0, –]
int lua_status (lua_State *L);

返回线程 L 的状态。

对于普通的线程，状态值可能是 0（LUA_OK），也可以是执行 **lua_resume** 时发生的错误对应的错误码，或者当线程暂停时是 LUA_YIELD。

你可以在线程状态为 LUA_OK 时在线程中调用函数。你可以在线程状态为 LUA_OK（要开启一个新的协程）或 LUA_YIELD（要恢复协程的执行）时恢复线程的执行。

lua_toboolean

[-0, +0, –]
int lua_toboolean (lua_State *L, int index);

将 index 处的值转为 C 的布尔类型的值（0 或 1）。

和在 Lua 中的测试结果一样，当要转换的值不是 false 也不是 nil，那 **lua_toboolean** 返回 true，否则返回 false。（如果你只想接受真正的布尔值，那可以使用 **lua_isboolean** 来判断值的类型。）

lua_tocfunction

[-0, +0, –]
lua_CFunction lua_tocfunction (lua_State *L, int index);

将 index 处的值转为 C 函数。该值必须为 C 函数，否则返回 NULL。

lua_tointeger

[-0, +0, –]
lua_Integer lua_tointeger (lua_State *L, int index);

等同于 isnum 为 NULL 时的 **lua_tointegerx**。

lua_tointegerx

[-0, +0, –]
lua_Integer lua_tointegerx (lua_State *L, int index, int *isnum);

将 index 处的值转为有符号整型 **lua_Integer**。该值必须为数字类型或可转为数字的字符串类型（参考 3.4.2）否则 **lua_tointegerx** 返回 0。

如果该数字不是一个整型值，那么它会以非特定形式被截断。

如果 isnum 不是 NULL，那么它会被赋值为布尔类型的值以表示操作是否成功。

lua_tolstring

[-0, +0, e]
const char *lua_tolstring (lua_State *L, int index, size_t *len);

将 index 处的值转为 C 字符串。如果参数 len 不是 NULL，那么会将字符串的长度设为 *len 的值。要转换的值必须为字符串或数字，否则函数返回 NULL。如果该值是个数字，那么 lua_tolstring 也会把栈中实际的值转为字符串。（在表遍历的过程中，如果使用 lua_tolstring 对键进行转换，那么它会使 **lua_next** 产生混乱。）

lua_tolstring 会返回一个指向 Lua 状态机内部字符串的完全对齐的指针（fully aligned pointer）。该字符串的最后一个字符的后面总是 0（“\0”），它的中间也可以包含其它的 0。因为 Lua 有垃圾回收机制，所以不能保证 lua_tolstring 返回的指针在对应值已从栈顶移除后还有效。

lua_tonumber

[-0, +0, –]
lua_Number lua_tonumber (lua_State *L, int index);

等同于 isnum 为 NULL 时的 **lua_tonumberx**。

lua_tonumberx

[-0, +0, –]
lua_Number lua_tonumberx (lua_State *L, int index, int *isnum);

将 index 处的值转为 C 类型的 **lua_Number**（参考 **lua_Number**），该值必须为数字类型或可转为数字的字符串类型（参考 **3.4.2**）否则 **lua_tonumberx** 返回 0。

如果 isnum 不是 NULL，那么它会被赋值为布尔类型的值以表示操作是否成功。

lua_topointer

[-0, +0, –]
const void *lua_topointer (lua_State *L, int index);

将 index 处的值转为普通的 C 指针（void*）。该值可以是 userdata，表，线程或函数，其它类型则 lua_topointer 返回 NULL。不同的对象会有不同的指针。没有办法将转换后的指针再转回原来的值。

通常这个函数用于处理调试信息。

lua_tostring

[-0, +0, e]
const char *lua_tostring (lua_State *L, int index);

等同于 len 为 NULL 时的 **lua_tolstring**。

lua_tothread

[-0, +0, –]
lua_State *lua_tothread (lua_State *L, int index);

将 index 处的值转为 Lua 线程（实际是一个 lua_State*）。该值必须为线程，否则此函数返回 NULL。

lua_tounsigned

[-0, +0, –]
lua_Unsigned lua_tounsigned (lua_State *L, int index);

等同于 isnum 为 NULL 时的 **lua_tounsignedx**。

lua_tounsignedx

[-0, +0, –]
lua_Unsigned lua_tounsignedx (lua_State *L, int index, int *isnum);

将 index 处的值转为无符号整型 **lua_Unsigned**。该值必须为数字类型或可转为数字的字符串类型（参考 3.4.2）否则 lua_tounsignedx 返回 0。

如果该数字不是一个整型值，那么它会以非特定形式被截断。如果一个数字超出了可表示的数值范围，it is normalized to the remainder of its division by one more than the maximum representable value.

如果 isnum 不是 NULL，那么它会被赋值为布尔类型的值以表示操作是否成功。

lua_touserdata

[-0, +0, –]
void *lua_touserdata (lua_State *L, int index);

如果 index 处的值是一个 full userdata 则返回它的块地址。如果该值是个 light userdata 则返回它的指针。否则返回 NULL。

lua_type

[-0, +0, –]
int lua_type (lua_State *L, int index);

返回 index 处的值的类型，如果索引是个非法索引（但可接受的）则返回 LUA_TNONE。lua_type 返回的类型在 lua.h 中定义，有以下一些类型值：LUA_TNIL，LUA_TNUMBER，LUA_BOOLEAN，LUA_TSTRING，LUA_TTABLE，LUA_TFUNCTION，LUA_TUSERDATA，LUA_TTHREAD 和 LUA_TLIGHTUSERDATA。

lua_typename

[-0, +0, –]
const char *lua_typename (lua_State *L, int tp);

返回类型为 tp 的类型名字，tp 必须是 **lua_type** 的返回值。

lua_Unsigned

typedef unsigned long lua_Unsigned;

给 Lua 的 API 使用的表示无符号整形的类型。它至少要求有 32 位。
默认情况下它是一个 unsigned int 或者 unsigned long，不论哪一种都可以保存 32 位的值。

lua_upvalueindex

[-0, +0, –]
int lua_upvalueindex (int i);

返回当前执行的函数的第 i 个 upvalue 的伪索引（参考 **4.4**）。

lua_version

[-0, +0, v]
const lua_Number *lua_version (lua_State *L);

返回保存在 Lua 核心中的版本号数据的地址。当 L 是个合法的 **lua_State** 时，返回创建状态机时使用的版本号数据地址。当 L 是 NULL 时，返回执行当前调用的状态机的版本号数据地址。

lua_Writer

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

lua_dump 使用的写函数的类型。每当 **lua_dump** 要处理一小块代码时，都会调用这个写函数，并将写缓冲区的指针（p）及缓冲区大小（sz）和 **lua_dump** 提供的 data 参数传给写函数。
写函数会返回一个错误码：0 表示没有错误，其它值表示错误并停止 **lua_dump** 对写函数的继续调用。

lua_xmove

[-?, +?, –]
void lua_xmove (lua_State *from, lua_State *to, int n);

在同一个状态机中的不同线程之间交换值。
此函数将线程 from 栈中的 n 个值弹出并将它们压入线程 to 的栈。

lua_yield

[-?, +?, –]
int lua_yield (lua_State *L, int nresults);

此函数等同于没有继续函数（参考 **4.7**）的 **lua_yieldk**。因此，当线程恢复时，它返回那个调用 lua_yield 的函数。

lua_yieldk

[-?, +?, –]
int lua_yieldk (lua_State *L, int nresults, int ctx, lua_CFunction k);

Yields 一个协程。
调用此函数时必须将其作为 C 函数的返回表达式，像下面这样：

```
return lua_yieldk(L, n, i, k);
```

当一个 C 函数像上面那样调用 **lua_yieldk** 时，正在执行的协程就会暂停，当调用 **lua_resume** 时协程开始返回。参数 nresults 是作为结果传给 **lua_resume** 的栈中的值的个数。
当协程恢复执行时，Lua 会调用继续函数 k 来实现 yield 住的 C 函数的继续执行（参考 **4.7**）。这个继续函数与之前的函数有相同的栈，栈中还有传给 **lua_resume** 的那些结果。此外，继续函数可以调用 **lua_getctx** 来访问 ctx 的值。

4.9 调试接口

Lua 没有内嵌的调试工具。它通过函数和钩子提供了一套特殊的接口。这套接口可以用来构建不同类型的调试器（debuggers），分析器（profilers）和其它需要获取解释器“内部信息”的工具。

lua_Debug

```
typedef struct lua_Debug {
  int event;
  const char *name;                /* (n) */
```

```
  const char *namewhat;        /* (n) */
  const char *what;            /* (S) */
  const char *source;          /* (S) */
  int currentline;             /* (l) */
  int linedefined;             /* (S) */
  int lastlinedefined;         /* (S) */
  unsigned char nups;          /* (u) upvalues 的数量 */
  unsigned char nparams;       /* (u) 参数的数量 */
  char isvararg;               /* (u) */
  char istailcall;             /* (t) */
  char short_src[LUA_IDSIZE]; /* (S) */
  /* private part */
  other fields
} lua_Debug;
```

这个结构是用来保存函数或活动记录的各种信息的。**lua_getstack** 只会填充该结构的私有部分给后面的函数使用。调用 **lua_getinfo** 会将有用的信息填充到 **lua_Debug** 的其它部分。

lua_Debug 的各个部分有以下含义：

- **source**:创建函数的源。如果源的名字以“@”开头，那么“@”后面的内容就是定义函数的文件名。如果源的名字以“=”开头，那么“=”后面的内容则是根据用户而定的方式来描述的。否则，函数就是在一个字符串中定义的，而源的内容就是该字符串。
- **short_src**:可打印版本的源描述。用于显示错误信息。
- **linedefined**:函数定义开始的行号。
- **lastlinedefined**:函数定义结束的行号。
- **what**:如果函数是个 Lua 函数那就是字符串“Lua”。如果是 C 函数则是字符串“C”。如果是一个代码块（编译后的代码块是作为函数存在的）函数则是字符串“main”。
- **currentline**:当前函数执行到的行号。如果无法获得行号信息，该值为-1。
- **name**:给定函数的一个合理的名字。因为函数在 Lua 中是第一类值，它们没有固定的名字：一些函数可以是多个全局变量的值，一些可能只保存在表中。函数 lua_getinfo 检测函数是如何被调用的，从而给它一个合适的名字。如果无法找到一个合适的名字，name 会被设为 NULL。
- **namewhat**:用来解释 name。根据函数是如何被调用的，namewhat 的值可以是“metamethod”，“for iterator”，“global”，“local”，“method”，“field”，“upvalue”或“”（空字符串。当没有对应的调用时用空字符串）
- **istailcall**:如果是通过尾调用的函数，则该值为真。这种情况下，这层的调用者不在调用栈中。
- **nups**:函数 upvalue 的数量。
- **nparams**:函数固定参数的数量（C 函数一定是 0）。
- **isvararg**:如果函数是个变参函数则为真（C 函数一定是真）。

lua_gethook

[-0, +0, –]

lua_Hook lua_gethook (lua_State *L);

返回当前的钩子函数。

lua_gethookcount

[-0, +0, –]
int lua_gethookcount (lua_State *L);

返回当前的钩子数量。

lua_gethookmask

[-0, +0, –]
int lua_gethookmask (lua_State *L);

返回当前钩子的掩码。

lua_getinfo

[-(0|1), +(0|1|2), e]
int lua_getinfo (lua_State *L, const char *what, lua_Debug *ar);

获得指定函数或函数调用（function invocation）的信息。

想要获得函数调用的信息，参数 ar 必须是一个合法的活动记录，它由 **lua_getstack** 或 **lua_Hook** 获得。

想要获得函数的信息，你首先要将函数压栈，且参数 what 是一个以字符“>”开头的字符串。（lua_getinfo 会将栈顶的函数弹出栈）举个例子，想要知道函数 f 定义的行号，你可以写以下代码：

```
lua_Debug ar;
lua_getglobal(L, "f");  /* 获得全局的 'f' */
lua_getinfo(L, ">S", &ar);
printf("%d\n", ar.linedefined);
```

what 中的每个字符都会选择结构 ar 的一些域来填充数据或者将一个值压入栈：

- 'n':填充域 name 和 namewhat。
- 'S':填充域 source,，short_src,，linedefined,，lastlinedefined,和 what。
- 'l':填充域 currentline。
- 't':填充域 istailcall。
- 'u':填充域 nups，nparams,，和 isvararg。
- 'f':将在指定层中运行的函数压入栈。
- 'L':将一个表压入栈。该表的索引是函数代码的合法行号。（合法的行号指的是与代码有关联的行号，也就是可以设置断点的行号。非法的行号包括空行和注释。）

发生错误时函数返回 0（比如 what 中有非法的字符）。

lua_getlocal

[-0, +(0|1), –]
const char *lua_getlocal (lua_State *L, lua_Debug *ar, int n);

获得指定活动记录或函数的局部变量信息。

第一种情况，参数 ar 必须是一个合法的活动记录，它由 **lua_getstack** 或 **lua_Hook** 获得。索引 n 指明要查看哪一个局部变量。想要了解变量索引和名字相关细节，请参考 **debug.getlocal**。

lua_getlocal 将变量的值压入栈，返回变量的名字。

第二种情况，参数 ar 应该是 NULL，目标函数必须在栈顶。这种情况下，只有 Lua 函数的参数是可见（因为这里没有活动变量的相关信息）且没有值会被压入栈。

当索引大于活动的局部变量的个数时，函数返回 NULL（没有东西被压入栈）。

lua_getstack

[-0, +0, –]
int lua_getstack (lua_State *L, int level, lua_Debug *ar);

获得解释器运行栈的相关信息。

此函数会将在指定层执行的活动记录的标识信息填充到 **lua_Debug** 这个数据结构。0 层指的是当前正在执行的函数，n+1 层指的是调用第 n 层函数（除了尾调用，因为它不在调用栈中）的函数。如果没有错误发生，**lua_getstack** 返回 1；如果层数大于调用栈的深度则返回 0。

lua_getupvalue

[-0, +(0|1), –]
const char *lua_getupvalue (lua_State *L, int funcindex, int n);

获得闭包的 upvalue 信息。（对于 Lua 的函数来说，upvalue 指的是函数使用的外部的局部变量。）lua_getupvalue 获得索引为 n 的 upvalue 的信息，将它的值压入栈并返回它的名字。参数 funcindex 指向栈中的闭包。（因为 upvalue 在整个函数中都是活动的，所以它没有什么特定的顺序。因此它们被随意地编号。）

当索引大于 upvalue 的个数时，函数返回 NULL（没有东西被压入栈）。对于 C 函数来说，用空字符串来给所有的 upvalue 命名。

lua_Hook

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

调试钩子函数的类型。

当一个钩子被调用时，它的 ar 参数中的 event 域会被设为特定的事件。Lua 通过以下常量来标记这些事件：LUA_HOOKCALL，LUA_HOOKRET，LUA_HOOKTAILCALL，LUA_HOOKLINE 和 LUA_HOOKCOUNT。对于 line 事件，currentline 域也会被设置。想要获得其它域的值，钩子必须调用 **lua_getinfo**。

对于 Call 事件，事件可能是 LUA_HOOKCALL 或 LUA_HOOKTAILCALL。后者没有对应的 Return 事件。

当 Lua 正在执行一个钩子时，它会阻止其它钩子的执行。因此，当一个钩子调用回 Lua 的函数或代码块时不会再次触发钩子的调用。

钩子函数不能有“继续函数”，也就是说，当 **lua_yieldk**，**lua_pcallk** 或 **lua_callk** 中的 k 为非空时，钩子函数中不能调用这三个函数。

钩子函数可以在下面的情况下 yield：只有 count 和 line 事件才可以 yield 并且它们不能 yield 其它的值。想要 yield 钩子函数，你必须使用 **lua_yield** 函数且它的参数 nresults 为 0 来结束钩子函数的执行。

lua_sethook

[-0, +0, –]
int lua_sethook (lua_State *L, lua_Hook f, int mask, int count);

设置调试钩子函数。

参数 f 是钩子函数。mask 指明了哪些事件会调用钩子函数：它用位来表示，用常量 LUA_MASKCALL，LUA_MASKRET，LUA_MASKLINE 和 LUA_MASKCOUNT 来组合。参数 count 只有在 mask 包含了 LUA_MASKCOUNT 的情况下才有意义。对于每一个事件，钩子被调用的解释如下：

- Call 钩子：当调用一个函数时会被调用。钩子是在进入新函数之后，获得函数参数之前被调用的。
- Return 钩子：当函数返回时会被调用。钩子是在离开函数后被调用的。没有一个标准的方法来访问函数返回的那些值。
- Line 钩子：当解释器正要开始执行一行新代码或跳转代码（包括跳转到相同的行）时会被调用。（此事件只会在 Lua 执行 Lua 函数时才发生。）
- Count 钩子：每当解释器执行完 count 条指令时会被调用。（此事件只会在 Lua 执行 Lua 函数时才发生。）

将 mask 设为 0 可以使钩子失效。

lua_setlocal

[-(0|1), +0, –]
const char *lua_setlocal (lua_State *L, lua_Debug *ar, int n);

设置指定活动记录中的局部变量的值。参数 ar 和 n 的含义和 **lua_getlocal** 中的一样（参考 **lua_getlocal**）。**lua_setlocal** 会将栈顶的值赋给指定的局部变量并返回局部变量的名字。

它也会将栈顶的值弹出栈。

当索引大于活动的局部变量的个数时，函数返回 NULL（没有东西被弹出栈）

lua_setupvalue

[-(0|1), +0, –]
const char *lua_setupvalue (lua_State *L, int funcindex, int n);

设置闭包的 upvalue 值。它将栈顶的值赋给指定的 upvalue 并返回 upvalue 的名字。同时它也会将栈顶的值弹出。参数 funcindex、n 的含义和 **lua_getupvalue** 中的一样（参考 **lua_getupvalue**）。

当索引大于 upvalue 的个数时，函数返回 NULL（没有东西被弹出栈）。

lua_upvalueid

[-0, +0, –]
void *lua_upvalueid (lua_State *L, int funcindex, int n);

返回 funcindex 处的闭包的第 n 个 upvalue 的唯一标记。参数 funcindex 和 n 的含义和 **lua_getupvalue** 中的一样（参考 **lua_getupvalue**）（n 不能大于 upvalue 的个数）。

程序使用这个唯一标记可以检查不同的闭包是否共享了 upvalue。共享同一个 upvalue（也就是访问了相同的外部的局部变量）的闭包会，它们返回的这个 upvalue 的标识是一样的。

lua_upvaluejoin

[-0, +0, –]
void lua_upvaluejoin (lua_State *L, int funcindex1, int n1,
int funcindex2, int n2);

将 funcindex1 处的闭包的第 n1 个 upvalue 引用为 funcindex2 处的闭包的第 n2 个 upvalue。（即前一个 upvalue 指向后一个 upvalue。）

5. 辅助库

辅助库提供了很多很有用的函数来帮助 Lua 与 C 的交互。基础 API 提供了 C 与 Lua 交互所需的所有函数，而辅助库提供了更高层次的函数以完成一些常见的操作。

辅助库中的所有函数和类型都在 lauxlib.h 头文件中定义，它们都有一个前缀：luaL_。

辅助库中的所有函数都是建立在基础 API 上的，因此离开基础 API，辅助库什么也做不了。尽管这样，使用辅助库中的函数可以使你写的代码有更好的一致性。

辅助库中的一些函数会在内部使用一些额外的栈空间。当辅助库中的函数使用的栈空间

数小于 5 时，它不会检查栈的大小，它简单地认为栈中有足够的空间。

辅助库中的一些函数是用来检查 C 函数的参数的。因为参数相关的错误消息都是有固定格式的（比如“bad argument #1”），所以你不能用这些函数来检查栈里面的其它值。

当函数调用 luaL_check*这样的函数时，如果检查不通过则反抛出一个错误。

5.1 - 函数与类型

下面我们按照字母表的顺序列出辅助库中所有的函数和类型

luaL_addchar

[-?, +?, e]
void luaL_addchar (luaL_Buffer *B, char c);

将字节数据 c 加入到 B 缓冲区（参考 **luaL_Buffer**）。

luaL_addlstring

[-?, +?, e]
void luaL_addlstring (luaL_Buffer *B, const char *s, size_t l);

将指针 s 指向的长度为 l 的字符串加入到 B 缓冲区（参考 **luaL_Buffer**）。字符串中可以包含嵌入式 0。

luaL_addsize

[-?, +?, e]
void luaL_addsize (luaL_Buffer *B, size_t n);

将之前已经拷贝到缓冲区（参考 **luaL_prepbuffer**）的长为 n 的字符串加入到缓冲区 B 中（参考 **luaL_Buffer**）。

luaL_addstring

[-?, +?, e]
void luaL_addstring (luaL_Buffer *B, const char *s);

将指针 s 指向的以 0 结尾的字符串加入到 B 缓冲区（参考 **luaL_Buffer**）。字符串中不能包含嵌入式的 0。

luaL_addvalue

[-1, +?, e]
void luaL_addvalue (luaL_Buffer *B);

将栈顶的值加入到 B 缓冲区（参考 **luaL_Buffer**）并将该值弹出。

这是唯一一个可以（必须）操作栈中额外元素（也就是要加入缓冲区的值）的字符串缓冲区相关的函数。

luaL_argcheck

[-0, +0, v]
void luaL_argcheck (lua_State *L,
int cond,
int arg,
const char *extramsg);

检查 cond 是否为真。如果为假则用标准消息唤起一个错误。

luaL_argerror

[-0, +0, v]
int luaL_argerror (lua_State *L, int arg, const char *extramsg);

用包含 extramsg（作为注解）的标准消息唤起一个错误。

虽然此函数永远不会返回，但作为一种习惯，在 C 函数中还是使用 return luaL_argerror(args)。

luaL_Buffer

typedef struct luaL_Buffer luaL_Buffer;

字符串缓冲区类型。

一个允许 C 代码创建 Lua 字符串的缓冲区。它的使用方法如下：

- 首先声明一个类型为 **luaL_Buffer** 的变量。
- 然后调用 luaL_buffinit(L, &b)来初始它。
- 接着调用 luaL_add*这些函数来一步步地构建缓冲区。
- 调用 luaL_pushresult(&b)来结束。此调用会将最终的字符串放在栈顶。

如果你事先知道最终字符串的大小，你可以像下面这样使用缓冲区：

- 首先声明一个类型为 **luaL_Buffer** 的变量。
- 然后调用 luaL_buffinitsize(L, &b, sz)来预分配大小为 sz 的空间。

- 接着将字符串拷贝到此空间。
- 调用 luaL_pushresultsize(&b, sz)来结束。sz 是拷贝到此空间的字符串的大小。

在操作缓冲区期间，字符串缓冲区会使用不定数量的栈空间。因此，在使用一个缓冲区的时候，你是不知道栈顶在哪里的。不过只要你能保证对栈的使用是平衡的，那么你可以在连续的缓冲区操作之间放心地使用栈，也就是说，当你调用一个缓冲区操作后，栈必须恢复到调用前的状态。（唯一的例外是 **luaL_addvalue**。）在调用 **luaL_pushresult** 后，栈恢复到它初始化时的状态，然后最后的结果会被放在栈顶。

luaL_buffinit

[-0, +0, –]
void luaL_buffinit (lua_State *L, luaL_Buffer *B);

初始化缓冲区 B。此函数并不分配任何空间。缓冲区必须声明为一个变量（参考 **luaL_Buffer**）。

luaL_buffinitsize

[-?, +?, e]
char *luaL_buffinitsize (lua_State *L, luaL_Buffer *B, size_t sz);

等同于执行完 **luaL_buffinit** 后再执行 **luaL_prepbuffsize**。

luaL_callmeta

[-0, +(0|1), e]
int luaL_callmeta (lua_State *L, int obj, const char *e);

调用一个元方法。

如果 index 处的对象有元表且元表中有 e 这个域，那么此函数会以 index 处的对象作为唯一参数传给元方法来调用它。此时函数返回真并将元方法返回的值压入栈。如果对象没有元表或元方法，函数返回假（不会压入任何值进栈）。

luaL_checkany

[-0, +0, v]
void luaL_checkany (lua_State *L, int arg);

检查函数的第 arg 个参数是否是任意类型的值（包括 nil）。

luaL_checkint

[-0, +0, v]
int luaL_checkint (lua_State *L, int arg);

检查函数的第 arg 个参数是否为数字，是的话就将该数字转为 int 后返回。

luaL_checkinteger

[-0, +0, v]
lua_Integer luaL_checkinteger (lua_State *L, int arg);

检查函数的第 arg 个参数是否为数字，是的话就将该数字转为 **lua_Integer** 后返回。

luaL_checklong

[-0, +0, v]
long luaL_checklong (lua_State *L, int arg);

检查函数的第 arg 个参数是否为数字，是的话就将该数字转为 long 后返回。

luaL_checklstring

[-0, +0, v]
const char *luaL_checklstring (lua_State *L, int arg, size_t *l);

检查函数的第 arg 个参数是否是一个字符串，是的话就返回该字符串。如果 l 非空，那么*l 会被设为字符串的长度。

此函数使用 **lua_tolstring** 来获得结果，因此此函数的转换规则与注意事项与 lua_tolstring 一样。

luaL_checknumber

[-0, +0, v]
lua_Number luaL_checknumber (lua_State *L, int arg);

检查函数的第 arg 个参数是否是一个数字，是的话就返回该数字。

luaL_checkoption

[-0, +0, v]
int luaL_checkoption (lua_State *L,
int arg,
const char *def,
const char *const lst[]);

检查函数的第 arg 个参数是否为字符串，是的话就在数组 lst（必须以 NULL 结尾）中搜索它。如果在 lst 数组中找到该字符串则返回该字符串在数组中的索引。如果参数不是一个字符串或无法在数组中找到就唤起一个错误。

如果参数 def 为非空，那么当参数不存在或参数为 **nil** 时使用 def 作为搜索的默认值。

此函数在将字符串映射为 C 枚举值时很有用。（Lua 的库中经常使用字符串而不是数字来做选项。）

luaL_checkstack

[-0, +0, v]
void luaL_checkstack (lua_State *L, int sz, const char *msg);

将栈的大小扩大到 top + sz 个元素，如果无法扩大则唤起一个错误。msg 是一个附加的消息，用于显示错误的（如果为 NULL 则没有附加消息）。

luaL_checkstring

[-0, +0, v]
const char *luaL_checkstring (lua_State *L, int arg);

检查函数的第 arg 个参数是否是一个字符串，是的话就返回该字符串。

此函数使用 **lua_tolstring** 来获得结果，因此此函数的转换规则与注意事项与 lua_tolstring 一样。

luaL_checktype

[-0, +0, v]
void luaL_checktype (lua_State *L, int arg, int t);

检查函数的第 arg 个函数的类型是否为 t。参考 **lua_type** 以了解类型 t 的编码值。

luaL_checkudata

[-0, +0, v]
void *luaL_checkudata (lua_State *L, int arg, const char *tname);

检查函数的第 arg 个函数是否是一个名字为 tname 的 userdata（参考 **luaL_newmetatable**）。是的话就返回 userdata 的地址（参考 **lua_touserdata**）。

luaL_checkunsigned

[-0, +0, v]
lua_Unsigned luaL_checkunsigned (lua_State *L, int arg);

检查函数的第 arg 个参数是否为数字，是的话就将该数字转为 **lua_Unsigned** 后返回。

luaL_checkversion

[-0, +0, –]
void luaL_checkversion (lua_State *L);

检查调用此函数的 core 和创建当前 Lua 状态机的 core 以及产生此调用的代码这三者使用的是否是同一个版本号的 Lua。同样也可以检查调用此函数的 core 和创建当前 Lua 状态机的 core 使用的是否是同一个地址空间。

luaL_dofile

[-0, +?, e]
int luaL_dofile (lua_State *L, const char *filename);

加载并运行指定的文件。它被定义为下面的宏：
(luaL_loadfile(L, filename) || lua_pcall(L, 0, LUA_MULTRET, 0))
没有发生错误时此函数返回 true（非 0），否则返回 false（0）。

luaL_dostring

[-0, +?, –]
int luaL_dostring (lua_State *L, const char *str);

加载并运行指定的字符串。它被定义为下面的宏：
(luaL_loadstring(L, str) || lua_pcall(L, 0, LUA_MULTRET, 0))
没有发生错误时此函数返回 false，否则返回 true。

luaL_error

[-0, +0, v]
int luaL_error (lua_State *L, const char *fmt, ...);

唤起一个错误。错误消息的格式由 fmt 和额外的参数指定，遵守和 **lua_pushfstring** 一样的规则。如果可以获得发生错误时的文件和行号信息，那么这些信息会被加到错误消息的前面。

虽然此函数永远不会返回，但作为一种习惯，在 C 函数中还是使用 return luaL_error(args)。

luaL_execresult

[-0, +3, e]
int luaL_execresult (lua_State *L, int stat);

此函数产生标准库中进程相关的函数（**os.execute** 和 **io.close**）的返回值。

luaL_fileresult

[-0, +(1|3), e]
int luaL_fileresult (lua_State *L, int stat, const char *fname);

此函数产生标准库中文件相关的函数（**io.open**，**os.rename**，**file:seek** 等等）的返回值。

luaL_getmetafield

[-0, +(0|1), e]
int luaL_getmetafield (lua_State *L, int obj, const char *e);

将 obj 索引处的对象的元表的域 e 的值压入栈。如果目标对象没有元表或元表中没有该域则返回 false 并不压任何东西入栈。

luaL_getmetatable

[-0, +1, –]
void luaL_getmetatable (lua_State *L, const char *tname);

将 Lua 注册表中与名字 tname 相关联的元表压入栈（参考 **luaL_newmetatable**）。

luaL_getsubtable

[-0, +1, e]
int luaL_getsubtable (lua_State *L, int idx, const char *fname);

确保 t[fname]是一个 table 并将其压入栈，其中 t 是索引 idx 处的值。如果存在这个 table 则返回 true，否则返回 false 并创建一个新的 table。

luaL_gsub

[-0, +1, e]
const char *luaL_gsub (lua_State *L,
const char *s,
const char *p,
const char *r);

创建字符串 s 的拷贝，其中字符串 s 中的字符串 p 会被替换为字符串 r。将拷贝结果压入栈并返回它。

luaL_len

[-0, +0, e]
int luaL_len (lua_State *L, int index);

返回 index 处的值的“长度”。它等同于 Lua 代码中的“#”操作符（参考 **3.4.6**）。如果操作的结果不是一个数字则唤起一个错误。（这种情况只能通过元方法发生。）

luaL_loadbuffer

[-0, +1, –]
int luaL_loadbuffer (lua_State *L,
const char *buff,
size_t sz,
const char *name);

等同于 mode 为 NULL 时的 **luaL_loadbufferx**。

luaL_loadbufferx

[-0, +1, –]
int luaL_loadbufferx (lua_State *L,

const char *buff,
size_t sz,
const char *name,
const char *mode);

将缓冲区的内容加载为为 Lua 代码块。此函数使用 **lua_load** 来将 buff 指向的长度为 sz 的的内容加载为代码块。

此函数的返回值和 **lua_load** 一样。name 是代码块的名字，用于调试信息和错误消息。字符串 mode 的作用和 **lua_load** 中的一样。

luaL_loadfile

[-0, +1, e]
int luaL_loadfile (lua_State *L, const char *filename);

等同于 mode 为 NULL 时的 **luaL_loadfilex**。

luaL_loadfilex

[-0, +1, e]
int luaL_loadfilex (lua_State *L, const char *filename,
const char *mode);

将文件的内容加载为 Lua 代码块。此函数使用 **lua_load** 来将文件名为 filename 的文件内容加载为代码块。如果 filename 为空，那么会从标准输入中加载。如果文件的第一行以# 开头，那么这行内容会被忽略。

字符串 mode 的作用和 **lua_load** 中的一样。

如果无法打开/读取文件或文件的模式有错误，此函数会返回 LUA_ERRFILE，此函数的其它返回值与 **lua_load** 的一样。

和 **lua_load** 一样，此函数只加载代码块并不执行它。

luaL_loadstring

[-0, +1, –]
int luaL_loadstring (lua_State *L, const char *s);

将字符串的内容加载为 Lua 代码块。此函数使用 lua_load 来加载以 0 结尾的字符串 s。

此函数的返回值和 lua_load 一样。

和 lua_load 一样，此函数只加载代码块并不执行它。

luaL_newlib

[-0, +1, e]
void luaL_newlib (lua_State *L, const luaL_Reg *l);

创建一个新的 table 并注册列表 l 中的函数。它由下面的宏实现：
(luaL_newlibtable(L,l), luaL_setfuncs(L,l,0))

luaL_newlibtable

[-0, +1, e]
void luaL_newlibtable (lua_State *L, const luaL_Reg l[]);

以固定的大小创建一个新 table，此 table 用来保存数组 l 中的入口（实际上并不保存它们）。它的目的是要配合 **luaL_setfuncs**（参考 **luaL_newlib**）来使用。

它以宏的形式实现。数组 l 必须是个真实的数组而不是数组指针。

luaL_newmetatable

[-0, +1, e]
int luaL_newmetatable (lua_State *L, const char *tname);

如果 Lua 的注册表中已经有一个键的名字叫 tname 则返回 0，否则新创建一个表作为 userdata 的元表，以 tname 作为键名，将新创建的表加入到 Lua 的注册表中，结果返回 1。

两种情况都会将注册表中与 tname 关联的值压入栈。

luaL_newstate

[-0, +0, –]
lua_State *luaL_newstate (void);

创建一个新的 Lua 状态机。它调用 **lua_newstate** 时使用了一个基于标准 C 的 realloc 函数作为分配函数，并设置了一个恐慌函数（参考 **4.6**）以在发生致命错误时将错误消息打印到标准错误输出中。

返回新的状态机，如果发生了内存分配错误则返回 NULL。

luaL_openlibs

[-0, +0, e]
void luaL_openlibs (lua_State *L);

在指定的状态机中开启所有的 Lua 标准库。

luaL_optint

[-0, +0, v]
int luaL_optint (lua_State *L, int arg, int d);

检查函数的第 arg 个参数是否为数字，是的话就将该数字转为 int 后返回。如果参数缺省或为 nil 则返回 d。其它情况唤起一个错误。

luaL_optinteger

[-0, +0, v]
lua_Integer luaL_optinteger (lua_State *L,
int arg,
lua_Integer d);

检查函数的第 arg 个参数是否为数字，是的话就将该数字转为 **lua_Integer** 后返回。如果参数缺省或为 **nil** 则返回 d。其它情况唤起一个错误。

luaL_optlong

[-0, +0, v]
long luaL_optlong (lua_State *L, int arg, long d);

检查函数的第 arg 个参数是否为数字，是的话就将该数字转为 long 后返回。如果参数缺省或为 **nil** 则返回 d。其它情况唤起一个错误。

luaL_optlstring

[-0, +0, v]
const char *luaL_optlstring (lua_State *L,
int arg,
const char *d,
size_t *l);

检查函数的第 arg 个参数是否为字符串，是的话就将返回该字符串。如果参数缺省或为 **nil** 则返回 d。其它情况唤起一个错误。

如果 l 为非空，则*l 会被设为该字符串的长度。

luaL_optnumber

[-0, +0, v]
lua_Number luaL_optnumber (lua_State *L, int arg, lua_Number d);

检查函数的第 arg 个参数是否为数字，是的话就返回该数字。如果参数缺省或为 **nil** 则返回 d。其它情况唤起一个错误。

luaL_optstring

[-0, +0, v]
const char *luaL_optstring (lua_State *L,
int arg,
const char *d);

检查函数的第 arg 个参数是否为字符串，是的话就将返回该字符串。如果参数缺省或为 **nil** 则返回 d。其它情况唤起一个错误。

luaL_optunsigned

[-0, +0, v]
lua_Unsigned luaL_optunsigned (lua_State *L,
int arg,
lua_Unsigned u);

检查函数的第 arg 个参数是否为数字，是的话就将该数字转为 **lua_Unsigned** 后返回。如果参数缺省或为 **nil** 则返回 u。其它情况唤起一个错误。

luaL_prepbuffer

[-?, +?, e]
char *luaL_prepbuffer (luaL_Buffer *B);

等同于预定义大小为 LUAL_BUFFERSIZE 的 **luaL_prebuffsize**。

luaL_prepbuffsize

[-?, +?, e]
char *luaL_prepbuffsize (luaL_Buffer *B, size_t sz);

返回一个指针，该指针指向的空间可以让你将一个长为 sz 的字符串拷贝到缓冲区 B（参

考 **luaL_Buffer**）。将字符串拷贝到缓冲区后，你必须以字符串的长度作为参数调用 **luaL_addsize**，这样才能真正地将字符串拷贝到缓冲区。

luaL_pushresult

[-?, +1, e]
void luaL_pushresult (luaL_Buffer *B);

结束使用缓冲区 B，将最终的字符串压入栈。

luaL_pushresultsize

[-?, +1, e]
void luaL_pushresultsize (luaL_Buffer *B, size_t sz);

等同于执行完 **luaL_addsize** 后再执行 **luaL_pushresult**。

luaL_ref

[-1, +0, e]
int luaL_ref (lua_State *L, int t);

在索引 t 处的 table 中为栈顶的对象创建并返回一个引用（将栈顶对象弹出栈）。

引用是一个唯一的整型键，只要你不手动往 t 处的 table 中加入整型键，那么 luaL_ref 会保证返回的键的唯一性。你可以使用引用 r 作为参数调用 lua_rawgeti(L, t, r)来获得对象的引用。函数 **luaL_unref** 释放一个引用及其关联的对象。

如果栈顶的对象是 **nil**，那么 luaL_ref 返回常量 LUA_REFNIL。常量 LUA_NOREF 用于区别 luaL_ref 返回的任何引用。

luaL_Reg

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

供注册函数 **luaL_setfuncs** 使用的函数数组类型。name 是函数的名字，func 是函数指针。任何 **luaL_Reg** 的数组必须在最后放置一个 name 和 func 都是 NULL 的元素作为哨兵。

luaL_requiref

[-0, +1, e]
void luaL_requiref (lua_State *L, const char *modname,
lua_CFunction openf, int glb);

以字符串 modname 作为参数调用函数 openf，并将其返回结果设置到 package.loaded[modname]中，就像该函数是通过 **require** 来调用的一样。

如果 glb 为 true，那么结果也会保存到名为 modname 的全局变量中。

调用结果的拷贝会压入栈。

luaL_setfuncs

[-nup, +0, e]
void luaL_setfuncs (lua_State *L, const luaL_Reg *l, int nup);

将数组 l（参考 **luaL_Reg**）中的所有函数注册到栈顶（如果有 upvalue 则为 upvalue 的下面，参考下面的解释。）的 table 中。

如果 nup 不是 0，那么所有函数都将共享 nup 个 upvalue。这些 upvalue 必须在压入库表（library table）后接着入栈。注册完成后，这些值都会被弹出栈。

luaL_setmetatable

[-0, +0, –]
void luaL_setmetatable (lua_State *L, const char *tname);

将注册表中与 tname 关联的表设为栈顶对象的元表（参考 **luaL_newmetatable**）。

luaL_testudata

[-0, +0, e]
void *luaL_testudata (lua_State *L, int arg, const char *tname);

此函数的作用和 **luaL_checkudata** 一样，只是当测试失败时函数返回 NULL 而不是抛出一个错误。

luaL_tolstring

[-0, +1, e]
const char *luaL_tolstring (lua_State *L, int idx, size_t *len);

用一个合适的格式将指定索引处的任何值转为 C 字符串。字符串会被压入栈并被函数返回。如果 len 为非空，那么函数将*len 的值设为字符串的长度。

如果指定的值有带“__tostring”域的元表，那么 luaL_tolstring 会以该值作为参数去调用对应的元方法，其结果将作为 luaL_tolstring 的结果返回。

luaL_traceback

[-0, +1, e]
void luaL_traceback (lua_State *L, lua_State *L1, const char *msg,
int level);

创建状态机 L1 的栈回溯并将其压栈。如果 msg 为非空，那么它会加到回溯信息的前面。参数 level 指定要从栈的哪一层开始回溯。

luaL_typename

[-0, +0, –]
const char *luaL_typename (lua_State *L, int index);

返回索引 index 处的值的类型名。

luaL_unref

[-0, +0, –]
void luaL_unref (lua_State *L, int t, int ref);

从索引 t 处的 table 中释放引用 ref（参考 **luaL_ref**）。该入口会从 table 中移除，因此引用的对象可以被回收掉。被释放的引用 ref 可以被再次使用。

如果 ref 是 LUA_NOREF 或 LUA_REFNIL，luaL_unref 不作任何操作。

luaL_where

[-0, +1, e]
void luaL_where (lua_State *L, int lvl);

将一个字符串压入栈。该字符串可以标识当前调用栈中第 lvl 层的控制位置。该字符串的格式为：

chunkname:currentline:

0 层是当前执行的函数，1 层是调用当前函数的函数，以此类推。

此函数用来构造错误消息的前缀。

6. 标准库

Lua 标准库提供通过一些很有用的函数，这些函数都是直接通过 C API 实现的。有些函数为语言提供了很基础的功能（比如 **type** 和 **getmetatable**）。有些提供了对“外部”服务的访问（比如 I/O）。还有一些很有用的函数是用 Lua 本身实现的，某些有高性能要求的函数则会用 C 实现（比如 **table.sort**）。

所有的库都是通过官方的 C API 实现的，它们是独立的 C 模块。Lua 目前有下面这些标准库：

- basic library (参考 **6.1**);
- coroutine library (参考 **6.2**);
- package library (参考 **6.3**);
- string manipulation (参考 **6.4**);
- table manipulation (参考 **6.5**);
- mathematical functions (参考 **6.6**) (sin, log, 等等.);
- bitwise operations (参考 **6.7**);
- input and output (参考 **6.8**);
- operating system facilities (参考 **6.9**);
- debug facilities (参考 **6.10**).

除了基础库和包相关的库，每个库中的函数都是作为某个全局 table 的域或某个对象的方法来提供的。

想要访问这些库，C 主程序可以调用 **luaL_openlibs** 来开启所有的库。如果你想分别打开标准库中的那些库，你可以使用函数 **luaL_requiref** 来调用 luaopen_base（基础库），luaopen_package(包库)，luaopen_coroutine(协程库)，luaopen_string(字符串库)，luaopen_table（表库），luaopen_math（数学库），luaopen_bit32（位操作库），luaopen_io（输入/输出库），luaopen_os（操作系统库），luaopen_debug（调试库）。这些函数都在头文件 lualib.h 中声明。

6.1 - 基础函数

基础库给 Lua 语言提供了核心的函数。如果你的应用程序中没有包含这个库，那么你应该认真考虑是否要自己实现一些函数。

assert (v [, message])

当参数 v 为假（比如 **nil** 或 **false**）时发出一个错误，否则返回函数的所有参数。参数 message 是一个错误消息，它的缺省值为“assertion failed!”。

collectgarbage ([opt [, arg]])

这个函数是垃圾回收器的接口。根据函数的第一个参数 opt，它实现了不同的功能：

- **"collect"**:执行一次完整的垃圾回收循环。这个是函数的默认选择。
- **"stop"**:停止垃圾回收器的自动执行。只有显式调用“重启”时，回收器才会再次

进行自动的回收工作。

- **"restart"**:重启垃圾回收器的自动执行。
- **"count"**:返回 Lua 使用的内存的总大小（单位为千字节）及内存大小（以字节为单位）模 1024 的值。第一个返回值带有小数部分，因此下面的等式永远为真：

```
k, b = collectgarbage("count")
assert(k*1024 == math.floor(k)*1024 + b)
```

(当 Lua 的数字类型不是浮点型时，第二个返回值就很有用了。)

- **"step"**:执行一次分步垃圾回收操作。步长由参数 arg 控制（值越大意味着回收步骤更多）。如果你想控制好步长，你必须通过实验的方法来调整 arg 的大小。如果这次分步执行完后，整个垃圾回收的循环也结束了，函数返回 true。
- **"setpause"**:将参数 arg 的值设为回收器的“暂停”（参考 **2.5**）值。返回之前的“暂停”值。
- **"setstepmul"**:将参数 arg 的值设为回收器的“步长”（参考 **2.5**）值。返回之前的“步长”值。
- **"isrunning"**:返回一个布尔值。如果垃圾回收器正在工作（没有 stop）就返回 true，否则返回 false。
- **"generational"**:将回收器的工作模式改为世代式。这是一个实验性质的特点（参考 **2.5**）。
- **"incremental"**:将回收器的工作模式改为增量式。这是默认的模式。

dofile ([filename])

打开指定名字的文件并将它的内容作为 Lua 的代码块来执行。如果不传参数，dofile 会执行标准输入（stdin）的内容。函数返回代码块的所有返回值。如果发生了错误，dofile 会将错误传给它的调用者（也就是说，dofile 不是在保护模式下运行的）。

error (message [, level])

终止最后一个保护的函数的调用并返回参数 message 作为错误消息。函数 error 从不会返回。

通常错误消息的开头都会带有错误发生位置的信息。参数 level 指定如果获得出错的位置信息。当 level 为 1 时（默认值），出错位置就是调用 error 函数的地方。当 level 为 2 时，出错位置就是调用 error 函数的函数被调用的地方，以此类推（调用栈上一级级往上）。如果不想将出错的位置信息加到错误消息中，将 0 传给 level 就可以了。

_G

一个保存了全局环境（参考 **2.2**）的全局变量（不是一个函数）。Lua 自己并不使用这个变量，改变它既不会影响任何的环境，也不会影响任何的 vice-versa（译者：不知道什么意思）。

getmetatable (object)

返回 object 的元表。如果对象没有元表就返回 **nil**。如果对象的元表中有“__metatable”这个域则返回该域所关联的值。

ipairs (t)

如果 t 有__ipairs 这个元方法，则将 t 作为参数去调用它并返回它的前三个返回值。否则的话，ipairs 返回三个值：迭代函数，表 t 和 0，所以语句：

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

会从表 t 中的第一对键值对(1,t[1])开始迭代，一直到出现第一个不连续的整型键时结束。

load (ld [, source [, mode [, env]]])

加载一个代码块。

如果 ld 是一个字符串，则代码块就是这个字符串。如果 ld 是一个函数，load 会不断地调用它来获得代码块的片断。每次调用 ld 都必须返回一个和之前结果连接起来的字符串。如果返回空字符串，**nil** 或没有返回都表示已到达代码块的尾部。

如果没有出现语法错误，那么 load 会将代码块作为一个函数返回，否则返回 **nil** 及错误消息。

如果返回的结果函数有 upvalues，那么它的第一个 upvalue 会被设为参数 env（如果有此参数）的值或全局环境的值。（当你加载主代码块的时候，结果函数一定且只有一个 upvalue，那就是_ENV（参考 **2.2**）。当你加载一个由函数（参考 **string.dump**）生成的二进制代码块时，结果函数可以有任意的 upvalues。）

参数 source 用来标识代码源的来源，它会用于显示错误消息和调试信息（参考 **4.9**）。当该参数的缺省时，如果 ld 是字符串，那缺省值就是 ld，否则就是字符串“=(load)”。

字符串参数 mode 限制了代码块的是文本形式还是二进制形式（也就是预编译的代码块）。它可能是字符串“b”（只能是二进制形式的代码块），“t”（只能是文本形式的代码块）或“bt”。默认值是“bt”。

loadfile ([filename [, mode [, env]]])

和 **load** 很相似，它从 filename 文件或标准输入加载代码块（如果 filename 参数缺省）。

next (table [, index])

用于遍历 table 中的所有元素。函数的第一个参数是一个 table，第二个参数是表中的一个索引。next 返回 table 中的下一下索引及与它关联的值。当 next 的第二个参数为 **nil** 时，如果 table 为空表，则 next 返回 **nil**，否则返回一个初始索引及与它关联的值。当 next 的第二个参数为最后一个索引时，next 返回 **nil**。第二个参数的缺省值为 **nil**。你可以使用 next(t) 来判断 table 是否是个空表。

遍历 table 时对元素的访问顺序是不定的，就算是数字的索引，也不是按数字顺序遍历的。（如果想按数字顺序遍历，可以使用数值形的 **for** 循环。）

如果在遍历的过程中对不存在的域进行赋值，next 的行为是未定义的。但是你可以在遍历的过程中修改已存在的那些域。特别地，你还可以删除已存在的域。

pairs (t)

如果 t 有__pairs 这个元方法，则将 t 作为参数去调用它并返回它的前三个返回值。否则的话，pairs 返回三个值：**next** 函数，表 t 和 **nil**，所以语句：

for i, v in pairs(t) do body end

会遍历表 t 中的所有键值对。

参考 next 以了解遍历过程中对 table 进行修改的警告说明。

pcall (f [, arg1, ···])

在保护模式下用指定的参数调用函数 f。这意味着在 f 中发生的任何错误都不会被传递，相反，pcall 捕获错误并返回一个状态码。它的第一个返回值是一个状态码（一个布尔值），如果为 true 则表示没有错误发生，调用成功。这种情况下， pcall 还会返回 f 函数的所有返回值，这些返回值在状态码之后被返回。如果发生了错误，pcall 返回 **false** 及错识消息。

print (···)

接收任意数量的参数并使用函数 **tostring** 将每一个参数转为字符串来输出到标准输出（stdout）。print 并不打算做格式化输出，它只是一种快速显示一个值的方法，比如在调 试 Lua 代码的时候。如果想完全地控制输出，请使用 **string.format** 和 **io.write**。

rawequal (v1, v2)

在不触发任何元方法的情况下比较 v1 和 v2 是否相等。返回一个布尔值。

rawget (table, index)

在不触发任何元方法的情况下获得 table[index]的值。参数 table 必须是一个表，参数 index 可以是任何值。

rawlen (v)

在不触发任何元方法的情况下获得对象 v 的长度，v 必须是个表或字符串。返回一个整型数字。

rawset (table, index, value)

在不触发任何元方法的情况下设置table[index]的值。参数table必须是一个表，参数index不能为 **nil** 和 NaN，参数 value 可以是任意的 Lua 值。

函数返回 table。

select (index, ···)

如果 index 是个正数，那么函数从第 index 个不定参数开始返回，如果 index 是个负数，则函数从倒数第 index 个（-1 表示最后一个参数）不定参数开始返回。否则，index 一定得是字符串“#”，这时 select 返回不定参数的个数。

setmetatable (table, metatable)

设置指定 table 的元表。（你不能在 Lua 中修改其它数据类型的元表，只有在 C 中才可以。）如果参数 metatable 为 nil 则表示删除指定 table 的元表。如果之前的元表中有“__metatable”这个域，调用 setmetatable 会报错。

函数返回 table。

tonumber (e [, base])

如果没有参数 base，tonumber 尝试将参数 e 转为一个数字。如果参数 e 是一个数字或一个可转为数字的字符串（参考 **3.4.2**），那么 tonumber 返回这个数字，否则返回 **nil**。

当填了 base（基数，如十进制的 10，二进制的 2）这个参数时，字符串参数 e 应根据这个基数来解释为一个整数。基数可以为 2 至 36 之间的任意整型，包括 2 和 36。当 base 大于 10 时，字母“A”（不管大小写）表示 10，“B”表示 11，依此类推，“Z”表示 35。如果字符串 e 中有不符合基数的字符，函数返回 **nil**。

tostring (v)

接受任意类型的值并将它转为一个合理的字符串。（想要完全地控制数字的转换，使用 **string.format**。）

如果 v 的元表中有“__tostring”这个域，那么 tostring 会以 v 作为参数调用对应值并将调用的返回值作为 tostring 的返回值。

type (v)

返回唯一参数 v 的类型名。可能的返回值有：“nil”（是一个字符串，不是值 **nil**），“number”，“string”，“boolean”，“table”，“function”，“thread”和“userdata”。

_VERSION

一个保存当前解释器版本名字的全局变量（不是一个函数）。此变量当前的值为“Lua 5.2”。

xpcall (f, msgh [, arg1, ···])

除了需要设置一个新的消息处理句柄 msgh 外，此函数和 **pcall** 一样。

6.2 - 协程操作

与协程相关的一些操作，它们作为基础库的一个子库封装在 coroutine 这个 table 中。关于协程，可以参考 **2.6** 中的介绍。

coroutine.create (f)

用函数体 f 来创建一个新的协程。f 必须是一个 Lua 函数。返回这个新的协程，它是一个类型为“thread”的对象。

coroutine.resume (co [, val1, ···])

开始或继续协程 co 的执行。当你第一次 resume 某个协程时，它就开始执行它的协程函数体。值 val1，……作为参数传给函数体。如果当前协程已经 yield 了，那么 resume 会使协程恢复执行，传递给 resume 的 val1，……会作为 yield 函数的返回结果。

如果执行协程的时候没有发生任何错误，那么 resume 会返回 **true** 和 yield 函数的参数（如果协程 yield 了）或函数体的返回值（如果协程执行完毕）。如果协程发生了错误，resume 返回 **false** 和错误消息。

coroutine.running ()

返回当前运行的协程及一个布尔值，如果布尔值为 true 则表示当前运行的协程为主协程。

coroutine.status (co)

返回协程 co 的状态。“running”表示协程正在执行（也就是说调用 status 的正是该协程）；“suspended”表示因调用了 yield 而暂停或还没开始；“normal”表示协程是激活的但并没有运行（也就是说，它 resume 了另一个协程）；“dead”表示协程已经执行完毕它的函数体或者因为某个错误而停止了。

coroutine.wrap (f)

用函数体 f 来创建一个新的协程。f 必须是一个 Lua 函数。返回一个封装函数，每次调用该封装函数时都会 resume 协程。传递给封装函数的所有参数都将作为 resume 函数的额外参数。除了没有第一个布尔值的返回值，封装函数的其它返回值和 resume 一样。如果协程发生了错误，那么错误将会被传递（propagate）。

coroutine.yield (···)

暂停当前协程的执行。传递给 yield 的所有参数都会作为 resume 的额外返回结果。

6.3 - 模块

包库提供了一些基本的工具用来在 Lua 中加载模块。库中的 require 函数在全局环境中，其它的函数都封装在 package 这个 table 中。

require (modname)

加载指定的模块。首先函数会在 **package.loaded** 这个表中查看是否已经加载了 modname 这个模块。如果是，那么 require 会返回保存在 package.loaded[modname]的值。否则它将尝试去查找一个加载该模块的加载器。

require 是由 **package.searchers** 引导来查找加载器的。修改 **package.searchers** 可以修改 require 查找模块的方式。下面的解释是基于默认配置的 **package.searchers**。

require 首先查询 package.preload[modname]是否有值，是则该值（应该是个函数）为加载器，否则 require 根据保存在 **package.path** 的路径去查找 Lua 加载器。如果找不到，那就根据保存在 **package.cpath** 的路径去查找，如果还是找不到，那么会尝试使用一个“集成”（all-in-one）的加载器（参考 **package.searchers**）。

一旦找到了加载器，require 会以两个参数去调用它，这两个参数为模块名及一个用来表示如何找到加载器的额外参数。（如果加载器来自某个文件，那额外参数就是该文件名）如果加载器返回一个非空的值，require 将该返回值赋值给 package.loaded[modname]，否则将 **true** 赋值给 package.loaded[modname]。不管哪种情况，require 都会返回 package.loaded[modname]的值。

如果在加载或执行模块时发生了错误，或者无法找到任何该模块的加载器，require 会唤起一个错误。

package.config

一个描述编译时包相关配置的字符串。此字符串是用行来分割的：

- 第一行配置目录分割符。Windows 系统下的默认值为“\”，其它系统的默认值为“/”。
- 第二行配置路径字符串中模板的分割符。默认值为“;”。
- 第三行配置模板中的替换符。默认值为“?”。

- 第四行是一个替换符，在 Windows 的路径中，它会被可执行目录（executable's directory）所替换。默认为“!”。
- 第五行是一个标记符。在构建 lua_open_函数名时，该标记符前面的字符都会被忽略。默认值为“-”。

package.cpath

require 查找 C 加载器的路径。

Lua 初始化 C 路径 **package.cpath** 的方式和初始化 Lua 路径 **package.path** 的方式是一样的，都是使用环境变量 LUA_CPATH_5_2 或环境变量 LUA_CPATH 或 luaconf.h 中定义的默认路径。

package.loaded

require 用来判断某个模块是否已经加载过的一个 table。当你加载一个名为 modname 的模块时且 package.loaded[modname] 不为假，**require** 仅简单地返回保存在 package.loaded[modname]的值。

该值仅仅是一个引用，对该值进行赋值并不会改变 require 使用的那个表。

package.loadlib (libname, funcname)

动态加载 libname 这个 C 库。

如果 funcname 是“*”，则只链接该库，使其它动态链接的库可以使用该库导出的符号。否则它在库中查找名为 funcname 的函数并返回这个函数。因此 funcname 这个函数必须遵守 **lua_CFunction** 的原型（参考 **lua_CFunction**）。

这是一个底层函数。它完全绕开了包和模块系统。和 **require** 不一样，它不会搜索路径，也不会自动做一些扩展操作。libname 必须是 C 库的完整路径，包括路径与扩展名。funcname 必须是 C 库导出的函数名（这个由所使用的编译器与链接器决定）。

此函数并不被标准 C 所支持。它只在一些平台（Windows，Linux，Mac OS X，Solaris，BSD 及其它支持 dlfcn 标准的 unix 平台）上有效果。

package.path

require 查找 Lua 加载器的路径。

在程序开始的时候，Lua 使用环境变量 LUA_PATH_5_2 或环境变量 LUA_PATH 或 luaconf.h 中定义的默认路径来初始化这个变量（如果前面两个环境变量没定义的话）。环境变量中的任何“;;”字符串都会被默认路径替换。

package.preload

一个保存指定模块加载器的 table（参考 **require**）。

该值仅仅是一个引用，对该值进行赋值并不会改变 **require** 使用的那个表。

package.searchers

一个给 require 控制如何加载模块的 table。

表中的每一个元素都是一个查找函数。当查找一个模块时，**require** 会按表索引的升序来调用这些查找函数，模块名（传给 **require** 的参数）是这些函数的唯一参数。函数会返回另一个函数（模块加载器）及一个会传给加载器的额外参数，函数也有可能返回一个解释为何找不到模块的字符串（如果无法解释则返回 **nil**）。

Lua 会用四个函数来初始化这个 table。

第一个查找函数只是简单地在 **package.preload** 中查找加载器。

第二个查找函数用保存在 **package.path** 中的路径去查找一个加载 Lua 库的加载器。查找方式在 **package.searchpath** 中说明。

第三个查找函数用保存在 **package.cpath** 中的路径去查找一个加载 C 库的加载器。同样，查找方式在 **package.searchpath** 中说明。举个例子，如果 C 路径是下面的字符串：

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

对模块 foo 的查找会按顺序尝试打开这些文件：./foo.so，./foo.dll 和/usr/local/foo/init.so。一旦找到一个 C 库，搜索函数首先会用动态链接工具将其与应用程序链接，然后它尝试在库中找到一个作为加载器的 C 函数。C 函数的名字为“lua_open_”与模块名组成的字符串，如果模块名有点号的话，所有的点号都会被替换为下划线。此外，如果模块名有连号（“-”），那么连号及连号之前的字符都会被移除。比如某个模块的名字为 a.v1-b.c，那么对应的函数名应该为 luaopen_b_c。

第四个查找函数尝试使用一个“集成”（all-in-one）的加载器。它使用模块名的根名字在 C 路径中查找库。比如 require(“a.b.c”)时，它会先查找名为“a”的 C 库，如果找到了就查找库中有没有对应的子模块的加载器函数，在我们这个例子中，加载器函数名为 lua_open_a_b_c。利用这个功能，我们可以将多个 C 的子模块打包进一个库文件，每个子模块只需定义自己的加载器函数就好。

除了第一个查找函数没有返回值外，其它的查找函数都会返回一个文件名（和 **package.searchpath** 返回的一样）以指示它在哪找到模块的。

package.searchpath (name, path [, sep [, rep]])

在指定路径中查找指定的模块名字。

路径是指一个包含多个由分号隔开的模板的字符串。对于每一个模板，函数会将其中的每一个问号（如果有的话）替换为参数 name，而参数 name 中的每一个 sep（默认是一个点）字符串也会被 rep（默认是操作系统的目录分割符）所替换。最后，函数尝试使用根据这个模板生成的文件名来打开这个文件。

举个例子，如果 path 是以下字符串：

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

查找模块 foo.a 时会按顺序尝试打开这些文件：./foo/a.lua，./foo/a.lc 和/usr/local/foo/a/init.lua。

函数返回第一个可以用读模式打开的结果文件名（在关闭文件后），或者在查找都不成功时返回 nil 及一个附加的错误消息。（错误消息会列出所有尝试去打开的文件名。）

6.4 - 字符串操作

此库提供了字符串相关操作的函数，比如查找和截取子串，模式匹配。索引 Lua 中的字符串时，第一个字符的位置是 1（不像 C 那样是 0）。索引可以是负数，表示从字符串后面往前的位置。因此，最后一个字符的位置为-1，其它的以此类推。

字符串库中的所有函数都封装在 string 这个 table 中。另外，每个字符串对象都有一个元表，元表中的__index 域指向 string 这个 table。因此，你可以用面向对象的方式来使用字符串库中的函数。比如 string.byte(s,i)也可以写成 s:byte(i)。

字符串库假设字符都是一字节编码的。

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

返回字符 s[i]，s[i+1],……，s[j]的内码。i 的默认值为 1，j 的默认值为 i。这些索引的规则和函数 **string.sub** 的是一样的。

字符的内码并不需要支持不同平台之间的移植。

string.char (…)

接受 0 或多个非负整数。返回一个长度为参数个数的字符串，字符串中每个字符的内码等于对应位置的参数值。

字符的内码并不需要支持不同平台之间的移植。

string.dump (function)

返回一个字符串，该字符串是给定函数的二进制表示，因此将此字符串传给 **load** 函数会返回该函数的副本（但函数的 upvalue 是新的）。

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

在字符串 s 中查找第一个与模式 pattern 相匹配的字符串。如果匹配成功则返回匹配字符串在字符串 s 的开始位置与结尾位置，否则返回 **nil**。第三个参数是一个可选的数字参数 init，它指定从字符串 s 的哪个位置开始查找，它的默认值为 1，可以为负数。第四个参数 plain 也是个可选的参数，如果为 **true** 则关闭模式匹配，此时函数是一个“查找子串”的操作，pattern 中没有字符会被看作为魔法字符。如果给定了 plain 参数，那么 init 也必须给定。

如果模式中有捕获，那么在匹配成功时，捕获的值会在返回的两个索引之后被返回。

string.format (formatstring, …)

根据第一个参数（必须是个字符串）指定的格式，将可变数量的参数格式化为一个字符串并返回。格式化字符串遵守与标准 C 函数 sprintf 一样的规则。唯一的区别是不支持*，h，

L，l，n 和 p 这些选项/修改符，另外增加了一个新的选项 q。选项 q 格式化双引号之间的字符串，在必要的时候使用转义符以保证 Lua 解释器可以安全地读回该字符串。举个例子，下面的调用：

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

会返回以下字符串：

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

选项 A 和 a（当可用时），E，e，f，G 和 g 期望参数是一个数字。选项 c，d，i，o，u，X 和 x 同样期望参数是一个字符，但数字的范围由 C 的实现所限制。选项 o，u，X 和 x，对应的数字可以为负数。选项 q 期望参数是一个字符串。选项 s 期望参数是一个不包含嵌入式 0 的字符串。如果选项 s 对应的参数不是一个字符串，那么它的转换规则与 **tostring** 的一样。

string.gmatch (s, pattern)

返回一个迭代函数，每次调用迭代函数的时候都会返回接下来根据模式 pattern 在字符串 s 中捕获的所有内容。如果 pattern 中没有指定捕获，则返回匹配的整个字符串。

作为一个例子，下面的循环会迭代字符串 s 中的所有单词，每行打印一个：

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

下一个例子将在给定的字符串中收集所有的键值对 key=value，并将它们放入一个 table：

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

对于此函数，补字符“^”无法作为模式的开始符，因为这会阻止迭代的执行。

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

返回字符串 s 的拷贝，其中 s 中出现的所有（或者前 n 个，如果给定了参数 n 的话）pattern 都将被 repl（或以是字符串，table 或函数）所指定的字符串所替换。gsub 还会返回匹配成功的次数作为它的第二个返回值。gsub 这个名字来源于 Global SUBstitution。

如果 repl 是个字符串，那么它的值会被用于替换。字符%是一种转换符：repl 中%d（d 在 1 至 9 之间）表示捕获的第 d 个子串。%0 表示完整的匹配字符串。%%表示字符%。

如果 repl 是一个 table，那么每次成功匹配的时候都会用第一个捕获的值作为键去查询 repl 中的值，然后替换 pattern。

如果 repl 是一个函数，那么每次成功匹配的时候都会按顺序将捕获的所有子串传递给这个函数来调用它。

如果 pattern 中没有指定捕获，那么整个 pattern 将作为一个捕获。

如果查询 table 得到的值或函数返回的值是一个字符串或数字，那么该值就会用作替换字符串，否则如果是 **false** 或 **nil** 则不作替换。

下面是一些例子：

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

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

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

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

x = string.gsub("4+5 = $return 4+5$", "%$(.-)%$", function (s)
      return load(s)()
    end)
--> x="4+5 = 9"

local t = {name="lua", version="5.2"}
x = string.gsub("$name-$version.tar.gz", "%$(%w+)", t)
--> x="lua-5.2.tar.gz"
```

string.len (s)

接受一个字符串作为参数并返回该字符串的长度。空字符串””的长度为 0。嵌入式 0 也会被计算，因此”a\000bc\000”的长度为 5（译者：这里\000 被当作为一个嵌入式 0，与\0 或\00 是一样的，但和\0000 却不一样）。

string.lower (s)

接受一个字符串作为参数并返回该字符串的一个拷贝，该拷贝中的所有大写字符都会转为小写字符，其它字符则保持不变。大写字符的定义与本地设置有关。

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

在字符串 s 中查找第一个与 pattern 匹配的字符串。如果匹配成功则返回捕获的所有字符串，否则返回 **nil**。第三个参数 init 是一个可选参数，它指定从字符串 s 的哪个位置开始查找，它的默认值为 1，可以为负数。

string.rep (s, n [, sep])

返回一个由 n 个 s 连接起来的字符串，s 与 s 之间由字符串 sep 隔开。sep 的默认值为空字符串（也就是没有分割符。）。

string.reverse (s)

返回 s 的反转字符串。

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

返回字符串 s 中位置 i 与位置 j 之间的子串。i 和 j 可以为负数。j 的缺省值可以认为是 -1（和字符串的长度一致）。string.sub(s,1,j)返回字符串 s 的长度为 j 的前缀字符串。string.sub(s, -i)返回字符串 s 的长度为 i 的后缀字符串。

如果负的索引 i 转换过后小于 1，那么它会纠正为 1；如果 j 大于字符串的长度，那么它会纠正为字符串的长度。如果转换及纠正过后的 i 大于 j，那么函数返回空字符串。

string.upper (s)

接受一个字符串作为参数并返回该字符串的一个拷贝，该拷贝中的所有小写字符都会转为大写字符，其它字符则保持不变。小写字符的定义与本地设置有关。

6.4.1 - 模式

字符类型：

一种字符类型用来表示某种字符的集合。下面的字符组合描述了各种字符类型：

- **x**:(x 不能为这些字符：^$()%.[]*+-?) 表示字符 x 本身。
- **.**:(点号) 表示所有字符。
- **%a**:表示所有字母。
- **%c**:表示所有的控制字符。
- **%d**:表示所有的数字。
- **%g**:表示所有可打印的字符，除了空白符（包括空格，换行，制表，回车符等）。
- **%l**:表示所有的小写字母。
- **%p**:表示所有的标点符号。
- **%s**:表示所有的空白符。
- **%u**:表示所有的大写字母。
- **%w**:表示所有的字母与数字。
- **%x**:表示所有的十六进制数字。
- **%x**:(x 为非字母非数字)表示字符 x。这是转义魔法字符的标准方法。任何的标点符

号（不是魔法字符也可以）在模式中可以在字符前面加上“%”来表示自身。

- **[set]:**表示一种集合类型，集合中包含了括号内的所有类型。可以用“-”按照升序的方式指定字符集合的范围。上面说明的所有类型也可以用在集合中。其它字符在集合中表示自身。比如[%w_]（或[_%w]）表示所有的字母、数字及下划线。[0-7]表示八进制数字，[0-7%l%-]表示八进制数字、小写字母及字符“-”。
  对类型应用范围说明是未定义的。像[%a-z]或[a-%%]这些是没有意义的。
- **[^set]**:表示集合类型的补集。

对于那些用一个字符表示的类型（%a，%c 等），对应的大写字母表示该类型的补集。比如%S 表示所有的非空白符。

字母、空白符和其它一些字符的定义都与本地设置有关。类型集合[a-z]与%l 并不一定是等价的。

模式项：

一个模式项可以是：

- 一个字符类型，匹配该类型的任意一个字符。
- 一个字符类型后面加“*”，匹配由 0 或多个该类型的字符组成的字符串。该匹配是最长匹配。
- 一个字符类型后面加“+”，匹配由 1 或多个该类型的字符组成的字符串。该匹配是最长匹配。
- 一个字符类型后面加“-”，匹配由 0 或多个该类型的字符组成的字符串。和“*”不一样的是，该匹配是最短匹配。
- 一个字符类型后面加“?”，匹配由 0 或 1 个该类型的字符组成的字符串。
- %n，n 为 1 至 9 的整数。匹配第 n 个捕获到的字符串。（请看下面的捕获）
- %bxy,，x 和 y 为两个可区分的字符。这种模式项匹配以字符 x 开头，字符 y 结尾的字符串，其中 x 与 y 是平衡的。也就是说，当从左至右读取字符串中的字符时，遇到 x 则计数加 1，遇到 y 则计数-1，结尾的 y 是当计数为 0 时的 y。举个例子，模式项%b()匹配平衡的左右圆括号。
- %f[set]，边界模式项。此模式项匹配一个空字符串，该空字符串后一个字符属于集合 set，前一个字符不属于集合 set。目标字符串的开头与结尾字符会被当作“\0”来处理。

模式：

一个模式是由一系列的模式项组成的。字符“^”放在模式的最前面限定了只能匹配以目标字符串的首字符开始的字符串。字符“$”放在模式的最后面限定了只能匹配以目标字符串的尾字符结束的字符串。“^”和“$”在其它的位置上表示它们自身。

捕获：

一个模式可以包含一个或多个由圆括号围起来的子模式，这些子模式称为捕获。当匹配成功时，匹配字符串中对应的捕获子串会被保存起来给接下来的操作使用。捕获会根据它们

的左圆括号来编号。比如模式"(a*(.)%w(%s*))"中匹配的字符串"a*(.)%w(%s*)"会保存在第一个捕获中（编号为 1）。与"."匹配的字符的捕获编号为 2。与"%s*"匹配的字符的捕获编号为 3。

一个特殊的例子是空的捕获()，它捕获的是当前字符串的位置（一个数字）。比如我们对字符串“flaaap”应用模式“()aa()”，这里捕获到两个数字：3 和 5。

6.5 - table 操作

此库提供了操作 table 的相关函数。所有的函数都封装在 table 这个表中。

记住，当一个操作需要用到 table 的长度时，该 table 必须是一个数列（sequence）或有__len 元方法（参考 **3.4.6**）。如果 table 作为函数的参数，那么非数值型键都会被库中的函数忽略掉。

基于性能的原因，函数对 table 的访问（获取/设置）都是原始的操作。

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

返回字符串 list[i]..sep..list[i+1] ··· sep..list[j]，其中 list 的全部元素为字符串或数字。sep 的默认值为空字符串。i 的默认值为 1，j 的默认值为#list。如果 i 大于 j 则返回空字符串。

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

将元素 value 插入到 list 的 pos 位置，元素 list[pos], list[pos+1], ···, list[#list]都向上移一个位置。pos 的默认值为#list+1，因此 table.insert(t, x)将 x 插入到 t 的最后面。

table.pack (···)

返回一个新的 table，其中 table[1]保存第一个参数，table[2]保存第二个参数，以此类推。同时 table 中还有一个键 n，它的值是函数参数的个数。注意，返回的 table 并不是一个数列。

table.remove (list [, pos])

删除 list 中位置为 pos 的元素并返回该元素。当 pos 是一个 1 至#list 之间的整数时，元素 list[pos+1], list[pos+2], ···, list[#list]都向下移一个位置并删除掉元素 list[#list]。当#list 为 0 时，pos 可以为 0，pos 也可以为#list+1，这种情况下，函数删除元素 list[pos]（译者：list[pos]有元素吗？）。

pos 的默认值为#list，因此 table.remove(t)删除 list 的最后一个元素。

table.sort (list [, comp])

将 list 中的元素排序。如果给定了参数 comp，那么它必须是一个函数，该函数接受两

个 list 中的元素，如果第一个元素在排序过后位置在第二个元素的后面则返回 true（排序后 not comp(list[i+1],list[i])为 true）。如果 comp 没给定，那么就会使用 Lua 的标准比较符“<”来代替。

排序算法是不稳定的，也就是说，被认为是相同的元素在经过排序后，在 list 中的位置有可能会发生改变。

table.unpack (list [, i [, j]])

返回 list 中的元素。此函数等价于下面的语句：

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

i 的默认值为 1，j 的默认值为#list。

6.6 - 数学函数

此库是标准 C 数学库的接口。库的所有函数都封装在 math 这个 table 中。

math.abs (x)

返回 x 的绝对值。

math.acos (x)

返回 x（单位为弧度）的反余弦值。

math.asin (x)

返回 x（单位为弧度）的反正弦值。

math.atan (x)

返回 x（单位为弧度）的反正切值。

math.atan2 (y, x)

利用两个参数的正负符号在对应象限中找到 y/x（单位都为弧度）的反正切值。（当 x 为 0 时也可以正确处理。）

math.ceil (x)

返回大于或等于 x 的最小整数。

math.cos (x)

返回 x（单位为弧度）的余弦值。

math.cosh (x)

返回 x 的双曲线余弦值。

math.deg (x)

返回弧度 x 对应的角度。

math.exp (x)

返回 e^x，即 e 的 x 次方。

math.floor (x)

返回小于或等于 x 的最大整数。

math.fmod (x, y)

返回 x 除以 y 的余数。

math.frexp (x)

返回两个值 m 与 e，这两个值使 $x = m2^e$ 成立。e 是一个整数，m 的绝对值在[0.5,1)之间（当 x 为 0 时，m 为 0）。

math.huge

一个大于或等于任意数字的值，即最大值。

math.ldexp (m, e)

返回 m2^{e}的值（e 应该是整数）。

math.log (x [, base])

返回以 base 为底数的 x 的对数。base 的默认值为 e（此时函数返回 x 的自然对数）。

math.max (x, ···)

返回参数列表中最大的值。

math.min (x, ···)

返回参数列表中最小的值。

math.modf (x)

返回 x 的整数部分与小数部分。

math.pi

π的值。

math.pow (x, y)

返回 x 的 y 次方。（你也可以用表达式 x^y 来计算该值。）

math.rad (x)

返回 x（单位为角度）对应的弧度。

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

此函数是标准 C 提供的简单伪随机函数 rand 的一个接口。(无法保证该函数的稳定值。)不传递任何参数去调用此函数时，返回一个范围为[0,1)的实数。当只传递了参数 m 时，math.random 返回范围为[1,m]的整数。当 m，n 参数都传递时，math.random 返回范围为[m,n]

的整数。

math.randomseed (x)

将 x 设为伪随机产生器的“种子”。相同的种子产生相同的数字序列。

math.sin (x)

返回 x（单位为弧度）的正弦值。

math.sinh (x)

返回 x 的双曲线正弦值。

math.sqrt (x)

返回 x 的开方值。（你也可以使用表达式 x^0.5 来计算该值。）

math.tan (x)

返回 x（单位为弧度）的正切值。

math.tanh (x)

返回 x 的双曲线正切值。

6.7 - 位操作

此库提供了位相关的操作。库中所有函数都封装在 bit32 这个 table 中。

除非另有说明，此库的所有函数接受一个范围在$(-2^{51}, +2^{51})$的数字参数。每个参数都会规格化为被 2^{32} 除的余数。因此最后值一定在$[0, 2^{32}-1]$这个区间。同样地，所有的结果也都在$[0, 2^{32}-1]$这个区间。注意：bit32.bnot(0)的结果是 0xFFFFFFFF，并不等于-1。

bit32.arshift (x, disp)

返回 x 右移 disp 位后的结果。disp 可以是任意可描述的整型。diap 为负数表示向左位移。

这种位移叫算术位移。左边新进的位为 x 的最高位，右边新进的位为 0。当 disp 的绝对值大于 31 时，结果一定为 0 或 0xFFFFFFFF（所有的位都移出去了）。

bit32.band (…)

将所有参数按位做与操作并返回最终的结果。

bit32.bnot (x)

对 x 按位取反并返回最终的结果。对于任何一个 x，下面都是一个恒等式：

```
assert(bit32.bnot(x) == (-1 - x) % 2^32)
```

bit32.bor (…)

将所有参数按位做或操作并返回最终的结果。

bit32.btest (…)

将所有参数按位做与操作，如果最终的结果不为 0 则返回 true，否则返回 false。

bit32.bxor (…)

将所有参数按位做异或操作并返回最终的结果。

bit32.extract (n, field [, width])

以无符号数字的形式返回数字 n 的第 field 位至第 field + width - 1 位所表示的值。位的编号为 0（最小标记）至 31（最大标记）。可读取的位的编号必须在范围[0,31]内。

width 的默认值为 1。

bit32.replace (n, v, field [, width])

将数字 n 的第 filed 位至第 field + width - 1 位的数据替换为数字 v 的二进制数据。field 和 width 的说明请参考 **bit32.extract**。

bit32.lrotate (x, disp)

返回 x 循环左位移（最高位会进入最低位）disp 位后的结果。disp 可以是任意可描述的整型。

对于任意的位移，下面都是一个恒等式：

```
assert(bit32.lrotate(x, disp) == bit32.lrotate(x, disp % 32))
```

如果 disp 为负数则表示向右位移。

bit32.lshift (x, disp)

返回 x 左移 disp 位后的结果。disp 可以是任意可描述的整型。diap 为负数表示向右位移。不管向哪个方向移动，新进的位都为 0。当 disp 的绝对值大于 31 时，结果一定为 0（所有的位都移出去了）。

当 disp 为正数时，下面是一个恒等式：

```
assert(bit32.lshift(b, disp) == (b * 2^disp) % 2^32)
```

bit32.rrotate (x, disp)

返回 x 循环右位移（最低位会进入最高位）disp 位后的结果。disp 可以是任意可描述的整型。

对于任意的位移，下面都是一个恒等式：

```
assert(bit32.rrotate(x, disp) == bit32.rrotate(x, disp % 32))
```

如果 disp 为负数则表示向左位移。

bit32.rshift (x, disp)

返回 x 右移 disp 位后的结果。disp 可以是任意可描述的整型。diap 为负数表示向左位移。不管向哪个方向移动，新进的位都为 0。当 disp 的绝对值大于 31 时，结果一定为 0（所有的位都移出去了）。

当 disp 为正数时，下面是一个恒等式：

```
assert(bit32.rshift(b, disp) == math.floor(b % 2^32 / 2^disp))
```

这种位移叫做逻辑位移。

6.8 - 输入与输出相关函数

I/O 库为文件操作提供了两种不同的方式。第一种方式使用隐式文件描述符，也就是说有些操作可以设置默认的输入文件和输出文件，所有的输入/输出操作都在这些默认文件上。第二种方式使用明确的文件描述符。

当使用隐式文件描述符时，所有的操作都有 io 这个 table 提供。当使用明确的文件描述符时，先使用 **io.open** 返回一个文件描述符，然后通过该文件描述符的元方法来操作文件。

io 这个 table 中也提供了三个预定义的文件描述符：io.stdin，io.stdout 和 io.stderr，它们的含义和 C 的一样。I/O 库永远不会关闭这三个文件。

除非另有说明，所有的 I/O 函数在失败的时候都返回 **nil**（第二个返回值为错误消息，第三个返回值是一个取决于系统的错误码），成功的时候都返回一个不同于 **nil** 的值。在非 Posix 的系统上，在发生错误时对错误消息和错误码的计算并不是线程安全的，因为它们依赖于 C 的全局变量 errno。

io.close ([file])

等同于 file:close()。如果不传 file 参数则关闭当前的默认输出文件。

io.flush ()

等同于 io.output():flush()。

io.input ([file])

当 file 是一个文件名时，它打开（以文本模式）file 这个文件并将它的句柄设为当前的默认输入文件。当 file 是一个文件句柄时，它只是简单地将这个句柄设为当前的默认输入文件。当不传递 file 参数时，函数返回当前的默认输入文件。

此函数在发生错误时会唤起一个错误而不是返回一个错误码。

io.lines ([filename …])

以只读模式打开给定文件名的文件并返回一个迭代函数，这个迭代函数的功能像基于该打开文件调用 file:lines(...)。当迭代函数检测到文件尾时会返回 **nil**（译者看代码后发现是没有返回值）（以结束循环）并自动关闭该文件。

io.lines()（没有参数）等同于 io.input():lines()。即迭代默认输入文件的每一行。这种情况下，当到达文件尾时并不会关闭文件。

此函数在发生错误时会唤起一个错误而不是返回一个错误码。

io.open (filename [, mode])

此函数以参数 mode 指定的方式打开一个文件。它返回一个新的文件句柄。如果发生了错误则返回 **nil** 及错误信息。

mode 可以为下面的任意字符串：

- **"r"**:读模式（默认值）。
- **"w"**:写模式。
- **"a"**:追加模式。
- **"r+"**:更新模式，文件之前的内容会被保留。
- **"w+"**:更新模式，文件之前的内容会被删除。
- **"a+"**:追加更新模式，文件之前的内容会被保留，只允许在文件的尾部写入数据。

mode 字符串的后面也可以加上“b”，在某些系统上需要用它来表示以二进制模式打开文件。

io.output ([file])

与 **io.input** 差不多，此函数是对默认输出文件的操作。

io.popen (prog [, mode])

此函数是依赖于系统的，并不是所有平台都可用。

在另一个进程中启动程序 prog 并返回一个文件句柄，你可以使用这个句柄来读取来自 prog 的数据（如果模式为“r”）或向 prog 写数据（如果模式为“w”）。

io.read (···)

等同于 io.input():read(...)。

io.tmpfile ()

返回一个临时文件的句柄。该文件以更新模式打开，在程序结束的时候自动关闭。

io.type (obj)

检查 obj 是否是一个合法的文件句柄。如果 obj 是一个打开的文件句柄则返回字符串“file”，如果 obj 是一个已关闭的文件句柄则返回“closed”，如果 obj 不是一个文件句柄则返回 nil。

io.write (···)

等同于 io.output():write(...)。

file:close ()

关闭文件。注意：当文件的句柄被垃圾回收器回收时，文件会自动关闭，但它的发生时间是不可预知的。

当关闭由 **io.popen** 打开的文件时，**file:close** 的返回值与 **os.execute** 的一样。

file:flush ()

将缓冲区的所有内容写入文件。

file:lines (···)

返回一个迭代函数，每次调用迭代函数时会根据给定的参数来读取文件的内容。当没有参数时全使用“*l”作默认值。下面的语句会从文件的当前位置开始遍历文件中的所有字符：

```
for c in file:lines(1) do body end
```

与 **io.lines** 不一样，当迭代结束时不会关闭文件。

此函数在发生错误时会唤起一个错误而不是返回一个错误码。

file:read (···)

根据给定的格式参数来读取文件的数据。对于每种格式，函数返回它读取到的字符串（或一个数字），如果无法按照给定的格式读取数据则返回 nil。当没有参数时，它的默认格式为“*l”。

有效的格式如下：

- **"*n"**:读一个数字。这是唯一一个返回一个数字的格式。
- **"*a"**:从当前位置开始，读取文件剩下的所有数据。如果当前位置为文件尾则返回空字符串。
- **"*l"**:读取下一行的数据，会忽略行尾（回车和换行符）。如果已达文件尾则返回 **nil**。这是函数默认使用的格式。
- **"*L"**:读取下一行的数据，包括行尾。如果已达文件尾则返回 **nil**。
- 数字:如果数字为 n，则从当前位置开始读取 n 个字节的数据，如果已达文件尾则返回 nil。如果数字为 0 则不读任何数据并返回空字符串或 **nil**（如果已达文件尾）。

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

以文件的开头作为计算基准，根据参数 whence（基于哪个位置）和 offset（偏移量）设置和获取文件的读写位置，whence 的值及含义如下：

- **"set"**:基于位置 0（文件的开头）。
- **"cur"**:基于当前位置。
- **"end"**:基于文件尾。

如果函数调用成功，seek 返回当前文件的读写位置（相对于文件开头），单位为字节。否则 seek 返回 **nil** 及一个描述错误的字符串。

whence 的默认值为“cur”，offset 的默认值为 0。因此 file:seek()在不改变文件当前读写位置的情况下返回它的读写位置。 file:seek(“set”)将当前读写位置设为文件头并返回 0。file:seek(“end”)将当前读写位置设为文件尾并返回文件的大小。

file:setvbuf (mode [, size])

设置输出文件的缓冲模式。有三种可用的模式：

- **"no"**:没有缓冲。任何输出都是即时写入文件的。
- **"full"**:全缓冲。只有当缓冲区满了或调用 flush（参考 **io.flush**）的时候才将缓冲区

的内容写入文件。

- **"line"**:行缓冲。只有当新的一行数据写入缓冲区或有来自别的文件(比如终端设备)的输入时才将缓冲区的内容写入文件。

对于最后两种模式，参数 size 以字节为单位指定了缓冲区的大小。默认值是一个合适的大小。

file:write (…)

将每个参数的值写入文件。参数必须是字符串或数字。
成功的话函数返回文件句柄，否则返回 **nil** 及一个描述错误的字符串。

6.9 - 操作系统相关函数

此库通过 os 这个 table 实现。

os.clock ()

返回程序占用 CPU 时间的近似值，单位为秒。

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

返回一个包含日期和时间信息的字符串或 table，格式由参数 format 指定。

如果传递了参数 time，则格式化 time 的时间（参考 **os.time** 以了解该值的说明）。否则，函数格式化当前的时间。

如果 format 参数以“!”开始，那么格式化后的时间为协调世界时。在“!”这个可选字符之后，如果格式字符串是“*t”，那么 date 函数返回一个 table，该 table 有下面这些域：year（4 位数），month（1-12），day（1-31），hour（0-23），min（0-59），sec（0-61），wday（星期几，1 为星期天），yday（一年当中第几天）以及 isdst（日光节省的标记，一个布尔值）。如果无法获得日光节省的相关信息的话就没有最后一个域。

如果 format 不是“*t”，那么 date 会根据标准 C 函数 strftime 的规则返回一个日期字符串。

当函数没有参数时，date 会根据主机系统及本地设置返回一个合理的日期与时间的字符串（也就是预，os.date()等同于 os.date(“%c”)）。

在非 Posix 的系统上，此函数并不是线程安全的，因为它依赖于 gmtime 和 localtime 这两个 C 函数。

os.difftime (t2, t1)

返回时间 t1 至时间 t2 的秒数差。在 POSIX、Windows 及其它一些系统上，返回值就是 t2-t1。

os.execute ([command])

此函数等同于标准 C 函数 system。它将 command 参数的值传给操作系统的壳（比如 Windows 的 DOS，linux 的 bash）。如果命令成功执行完毕，则函数的第一个返回值为 **true**，否则为 **nil**。在第一个返回值之后，函数还会返回一个字符串和一个数字，看下面说明：

- **"exit"**:命令正常结束。后面的数字表示命令的退出状态码。
- **"signal"**:命令被某个信号结束。后面的数字表示结束命令的信号。

不传递参数时，os.execute 返回一个布尔值，如果为 true 表示系统的壳是可用的。

os.exit ([code [, close])

调用标准 C 函数 exit 来结束主程序。如果 code 为 **true**，返回的状态码为 EXIT_SUCCESS，如果 code 为 **false**，返回的状态码为 EXIT_FAILURE。如果 code 是一个数字，则返回的状态码为该数字。参数 code 的默认值为 **true**。

如果第二个可选参数 close 为 true，则在退出主程序前先关闭 Lua 状态机。

os.getenv (varname)

返回名为 varname 的进程环境变量，如果该变量没有定义则返回 **nil**。

os.remove (filename)

删除名为 filename 的文件（或者是 POSIX 系统上的空文件夹）。如果失败则返回 **nil**，错误描述和错误码。

os.rename (oldname, newname)

将名为 oldname 的文件或文件夹改名为 newname。如果失败则返回 **nil**，错误描述和错误码。

os.setlocale (locale [, category])

设置程序的本地化信息。参数 locale 是一个系统相关的字符串，用于指定地域，比如“chs”表示简体中文的地区。参数 category 是一个可选的字符串，用于指定需要改变哪些类别：“all”（全部），“collate”（排序规则），“ctype”（字符类别及转换），“monetary”（货币格式），“numeric”（数字格式），或者“time”，默认值为“all”。函数返回新的本地化名字，如果请求不能满足则返回 **nil**。

如果 locale 是一个空字符串则将当前本地化信息设为当前机器的本地化信息。如果 locale 是“C”，则当前本地化信息会被设为标准 C 的本地化信息。

当第一个参数为 **nil** 时，函数返回指定类别当前的本地化信息。

此函数并不是线程安全的，因为它依赖于 setlocale 这个 C 函数。

os.time ([table])

如果没有参数，函数返回当前时间，如果有参数，则返回参数 table 指定的时间。参数 table 必须要有 year，month，day 这三个域，可选的域有 hour（默认值为 12），min（默认值为 0），sec（默认值为 0）和 isdst（默认值为 **nil**）。这些域的说明可以参考 **os.date** 函数。

函数的返回值是一个数字，它的含义取决于你的系统。在 POSIX，Windows 和其它一些系统上，该数字为从“epoch”（1970 年 1 月 1 日的 0 时 0 分 0 秒）到当前的秒数。在其它的一些系统上，该数字的含义是不确定的，它只能用作 **os.date** 和 **os.difftime** 的参数。

os.tmpname ()

返回一个可以用作临时文件名的字符串。该文件在使用前必须先明确地打开，并且在不需要时要明确地关闭。在 POSIX 系统上，此函数还会创建该名字的文件以避免安全性的风险。（有可能有人凭借错误的权限在生成名字与创建文件之间创建一个同名文件。）但你仍需要在使用它前打开它，同时要记得关闭它（就算你没有使用它）。

如果可能，你最好还是选择使用 **io.tmpfile** 这个函数，因此它会在程序结束时自动删除。

6.10 - 调试相关函数

此库提供了调试 Lua 程序的接口（参考 4.9）。你应该谨慎地使用这些接口。库中有好几个函数会破坏 Lua 代码的基本假设（比如无法从外部访问函数的局部变量；Lua 代码无法修改 userdata 的元表；Lua 程序不会崩溃等），因此会破坏 Lua 代码的安全性。此外，库中有些函数的执行速度可能会很慢。

库中的所有函数都封装在 debug 这个 table 中。那些基于线程的函数的第一个参数是一个可选的参数，该参数表示函数要操作的目标线程，它的默认值为当前线程。

debug.debug ()

进入与用户进行交互的模式，执行用户输入的每个字符串。用户可以使用简单的命令和其它调试工具观察和修改全局变量和局部变量的值，对表达式进行计算等等。输入单词“cont”可以结束此函数，从而继续原来执行。

注意，debug.debug 这个命令并不是语法嵌套在函数中的，因此它不能直接访问局部变量。

debug.gethook ([thread])

返回给定线程的当前钩子信息，有三个返回值：钩子函数，钩子掩码及钩子的计数（使用 **debug.sethook** 函数设置的）。

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

以 table 的形式返回指定线程中某个函数的信息。参数 f 可以是函数也可以是个数字，为函数时，则该函数为目标函数，为数字时表示运行在调用栈第 f 层的函数。0 层表示当前函数（即 getinfo 自身），1 层为调用 getinfo 的函数（除了尾调用，因为尾调用不在调用栈上），以此类推。如果层数大于活动函数的数量，getinfo 返回 **nil**。

返回的 table 可以包含 **lua_getinfo** 返回的所有域，参数 what 用来指定哪些域会被填上值。如果没有 what 参数则返回除了合法行号表外的所有的信息。

举个例子，如果能找到一个合理的名字，debug.getinfo(1, “n”).name 返回一个包含当前函数名字信息的 table。debug.getinfo(print)则返回一个包含 **print** 函数所有可得到的信息的 table。

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

此函数返回在调用栈 f 层的函数的第 local 个局部变量的名字和值。此函数不仅可以访问那些显式定义的局部变量，还可以访问函数参数临时变量等。

第一个参数或局部变量的索引为 1，以此类推直到最后一个活动的变量。负的索引表示变参，-1 表示变参中的第一个变量。当指定索引处没有局部变量时，函数返回 **nil**。当指定的层超出范围时，函数唤起一个错误。（你可以调用 **debug.getinfo** 来检查某层是否有效。）

以“(”（开圆括号）开头的变量名表示内部变量（循环的控制变量，临时变量，变参和 C function locals）。

参数 f 也可以是个函数。这种情况下，getlocal 只返回函数参数的名字（第一个参数的索引为 1）。

debug.getmetatable (value)

返回给定 value 的元表。如果 value 没有元表则返回 **nil**。

debug.getregistry ()

返回 Lua 的注册表（参考 **4.5**）。

debug.getupvalue (f, up)

返回指定函数 f 的第 up 个 upvalue 的名字和值。如果指定的索引处没有 upvalue 则返回 nil。

debug.getuservalue (u)

返回与 u 关联的 Lua 变量的值。如果 u 不是 userdata 则返回 **nil**。

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

将指定的函数设为一个钩子。字符串参数 mask 与数字参数 count 描述了钩子函数什么时候会被调用。mask 可以包含下面这些字符：

- **'c'**:每当 Lua 调用一个函数的时候会调用钩子函数。
- **'r'**:每当 Lua 从函数返回时会调用钩子函数。
- **'l'**:每当 Lua 执行新的一行代码时会调用钩子函数。

如果参数 count 的值不为 0，则每执行完 count 条指令时调用一次钩子函数。当函数没有参数时，**debug.sethook** 关闭钩子。

当钩子函数被调用时，它的第一个参数是一个描述触发事件的字符串：“call”（或“tail call”），“return”，“line”和“count”。对于 line 事件，钩子函数的第二个参数是新行的行号。在钩子函数里面，你可以调用 debug.getinfo(2)来获取更多的关于正在执行的函数的信息（0 层是 getinfo 函数，1 层是钩子函数）。

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

此函数将参数 value 的值赋值给调用栈第 level 层的函数的第 local 个局部变量。当指定索引处没有局部变量时，函数返回 **nil**。当指定的层超出范围时，函数唤起一个错误。（你可以调用 debug.getinfo 来检查某层是否有效。）否则它返回局部变量的名字。

参考 **debug.getlocal** 以了解更多关于局部变量索引与名字的信息。

debug.setmetatable (value, table)

将指定的参数 table（可以为 **nil**）设为指定参数 value 的元表。返回 value 的值。

debug.setupvalue (f, up, value)

此函数将参数 value 的值赋值给函数 f 的第 up 个 upvalue。当指定索引处没有 upvalue 时，函数返回 **nil**，否则函数返回 upvalue 的名字。

debug.setuservalue (udata, value)

将参数 value 设为与参数 udata 关联的 Lua 值。value 必须是一个 table 或 **nil**。udata 必须是 full userdata。

返回 udata。

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

如果 mesage 既不是字符串也不是 **nil**，那么此函数只返回 message，不做其它处理。否则返回调用栈的一个栈回溯信息。可选字符串参数 message 会被加到栈回溯信息的前面。可

选参数 level 指定了从栈的哪一层开始回溯(默认是 1，即从调用 traceback 的那个函数开始)。

debug.upvalueid (f, n)

返回函数 f 的第 n 个 upvalue 的唯一标识（是一个 light userdata)。

程序可以利用这个唯一标记来判断是否有不同的闭包共享了 upvalue。被不同闭包共享（即它们访问同一个外部局部变量）的 upvalue 它们的唯一标记是一样的。

debug.upvaluejoin (f1, n1, f2, n2)

将 Lua 闭包 f1 的第 n1 个 upvalue 引用为 Lua 闭包 f2 的第 n2 个 upvalue。

7. 独立使用 lua

虽然 Lua 被设计为一种扩展性语言以嵌入到 C 程序中，但它也经常作为一种独立的语言被使用。在 Lua 的标准发行包中有个叫 lua 的解释器可以独立执行 Lua 语言。这个标准的解释器包含所有的标准库和调试库。它的用法为：

lua [options] [script [args]]

options 可以为以下格式：

- **-e stat**:执行表达式字符串 stat。
- **-l mod**:"requires" mod 模块。
- **-i**:执行完脚本后进入交互模式。
- **-v**:打印版本信息。
- **-E**:忽略环境变量。
- **--**:停止处理 options。
- **-**:以标准输入（stdin）作为执行文件来执行并停止处理 options。

在处理完这些 options 后，Lua 将 args 作为字符串参数传给 script 并执行它。当没有任何参数时，如果标准输入（stdin）是一个终端，那么就相当于执行 lua -v -i，否则就相当于 lua -。

如果没有-E 选项，解释器会在执行任意参数前检查是否有 LUA_INIT_5_2（如果没有则检查 LUA_INIT）这个环境变量。如果环境变量的值为@filename 这种格式，则 Lua 会执行该文件，否则 Lua 执行环境变量这个字符串。

如果有-E 这个选项，除了会忽略 LUA_INIT 这个环境变量，lua 还会忽略 LUA_PATH 和 LUA_CAPTH 这两个环境变量。同时将 **package.path** 和 **package.cpath** 的值设为 luaconf.h 中定义的默认值。

除了-i 和-E，所有的选项都是按顺序处理的。举个例子：

```
$ lua -e'a=1' -e 'print(a)' script.lua
```

上面的命令会先将 1 赋值给变量 a，然后再打印 a，最后以没有参数的方式执行脚本文件 script.lua。（这里$为壳（shell）的提示符。你的提示符有可能不一样。）

在开始执行脚本之前，lua 将命令行中的所有参数都放入一个名为 arg 的全局 table 中。其中脚本名保存在索引 0 处，脚本名后面的第一个参数保存在索引 1 处，以此类推。脚本名之前

的参数（即解释器的名字和选项内容）放在负索引处。举个例子：

```
$ lua -la b.lua t1 t2
```

解释器首先执行脚本 a.lua，然后创建一个 table：

```
arg = { [-2] = "lua", [-1] = "-la",
        [0] = "b.lua",
        [1] = "t1", [2] = "t2" }
```

最后执行脚本 b.lua。解释器以参数 arg[1]，arg[2],……作为参数去调用脚本。这些参数也可以使用变参表达式“...”来访问。

在交互模式下，如果你写的语句不完整，解释器会用一个不同的提示符来提示你，直到你把语句写完整。

如果脚本中发生了非保护的错误，那么解释器会将该错误输出到标准错误流中。如果错误对象是一个字符串，那么解释器会加上一个栈回溯信息。否则，如果错误对象有__tostring 元方法，则解释器会调用该元方法来产生最终的错误消息。如果错误对象为 nil，则解释器不会输出错误信息。

当脚本正常结束的时候，解释器会关闭主 Lua 状态机（参考 **lua_close**）。脚本可以通过调用 **os.exit** 来避免这个操作。

为了允许 Unix 系统将 Lua 作为脚本解释器使用，如果代码块的第一行以#开始，则 Lua 的解释器会跳过该行。因此，使用 chmod +x 和#!这种格式可以使 Lua 脚本变为可执行文件，如下：

```
#!/usr/local/bin/lua
```

（当然，不同机器上 Lua 解释器所在的位置有可能不同，如果 lua 解释器在你的 PATH 环境变量指定的路径中，那么#!/usr/bin/env lua 具有更好的移植性。）

8. 与之前版本的不同

这里我们列出了从 Lua5.1 升级到 Lua5.2 时的一些不兼容的地方。你可以通过修改一些编译选项来避免这些不兼容的地方（参考 luaconf.h）。尽管如此，这些兼容选项会在 Lua 的下一个版本中被移除，Lua5.2 已经把 Lua5.1 中那些标记为“不推荐”的内容移除掉了。

8.1 - 语言上的变化

- 环境的概念变了。只有 Lua 的函数才有环境。使用变量_ENV 或函数 load 来设置 Lua 函数的环境。
- C 函数不再有环境。如果你想在不同的 C 函数中共享状态，可以使用一个 table 来作为 C 函数的共享 upvalue。（你可以使用 **luaL_setfuncs** 在开启 C 库时让所有函数都共享一个公共的 upvalue。
- 想要操作 userdata（现在叫 user value）的“环境”，你可以使用新的函数：**lua_getuservalue** 和 **lua_setuservalue**。
- Lua 的标识符不能使用依赖于地域（locale-dependent）的字母了。
- 执行分步回收或全回收时不会重新启动已停止工作的回收器。
- 拥有弱键的弱表现在表示得像短命（ephemeron）table。
- 调试钩子中的尾返回事件（tail return）被删除了，同时增加了尾调用事件（tail call）。

因此，调试器可以知道该种调用没有对应的返回事件。

- 函数的相等比较规则改变了。现在定义一个函数时不一定会创建一个新的值，如果它和之前的定义的一个函数没有任何区别，那么就会重用之前函数的值。

8.2 - 标准库的变化

- 不推荐使用函数 module。用常规的 Lua 代码可以很容易创建一个模块。 模块不需要设为全局变量了。
- 因为环境的改变，函数 setfenv 和 getfenv 已被删除了。
- 不推荐使用函数 math.log10。将 10 作为第 2 个参数来调用 **math.log** 可以代替 math.log10 这个函数。
- 不推荐使用函数 loadstring。可以使用函数 load 来代替。现在 load 可以接受一个字符串参数了，它的效果与 loadstring 完全一样。
- 不推荐使用函数 table.maxn。如果真需要，你可以自己用 Lua 实现一个。
- 当命令成功结束时，函数 os.execute 返回 **true**。否则返回 nil 和错误信息。
- 函数 unpack 被移到 table 库中了，因此必须用 **table.unpack** 这种形式调用。
- 不推荐使用模式中的字符类型%z，因为现在模式可以把'\0'当作常规字符包含进来。
- 表 package.loaders 改名为 package.searchers。
- Lua 不再有字节码的验证机制。因此，所有与加载代码有关的函数（**load** 和 **loadfile**）在加载不受信任的二进制数据时都有潜在的安全问题。（实际上，由于验证算法的漏洞，那些函数早就不安全了。）当你对某些代码有顾虑的时候，你可以使用那些函数的 mode 参数来限制它们只能加载文本形式的代码块。
- 不同版本的官方发行包之间，标准路径有可能会改变。

8.3 - API 的变化

- 伪索引 LUA_GLOBALSINDEX 被删除了。你必须从注册表中获得全局环境表（参考 **4.5**）。
- 伪索引 LUA_ENVIRONINDEX 和函数 lua_getfenv/lua_setfenv 被删除了，因为 C 函数不再有环境。
- 不推荐使用函数 luaL_register。使用 **luaL_setfuncs** 可以让你的模块不在全局中创建（不要求模块是一个全局变量）。
- 当创建一个新块时（也就是 ptr 为 NULL 时），分配函数的参数 osize 可以不为 0 了（参考 **lua_Alloc**）。只能通过 ptr == NULL 来判断分配的块是否是新的。
- userdata 按照它们被标记的反序而不是它们被创建的顺序来调用它们的__gc 元方法（参考 **2.5.1**）。（大部分 userdata 都是在它们被创建的时候就被标记了）同时，如果设置元表时元表中没有__gc 域，那么即使在之后设置了该域也不会被调用了。
- luaL_typerror 已被移除。有需要的话你可以自己实现一个。
- 不推荐使用函数 lua_cpcall。 你可以先用 **lua_pushcfunction** 将函数压栈再用 **lua_pcall** 执行函数。
- 不再推荐使用 lua_equal 和 lua_lessthan。可以使用新函数 **lua_compare** 来代替。
- 函数 lua_objlen 改名为 **lua_rawlen**。

- 函数 **lua_load** 多了一个额外的参数 mode。将 NULL 传给这个参数就可以实现旧的行为。
- 函数 **lua_resume** 多了一个额外的参数 from。可以传 NULL 或调用此函数的线程给这个参数。

9. 完整的 Lua 语法

下面是完整的 Lua 扩展巴科斯范式（extended BNF）语法描述。

chunk ::= block

block ::= {stat} [retstat]

stat ::= ‘;’ |
varlist ‘=’ explist |
functioncall |
label |
break |
goto Name |
do block **end** |
while exp **do** block **end** |
repeat block **until** exp |
if exp **then** block {**elseif** exp **then** block} [**else** block] **end** |
for Name ‘=’ exp ‘,’ exp [‘,’ exp] **do** block **end** |
for namelist **in** explist **do** block **end** |
function funcname funcbody |
local function Name funcbody |
local namelist [‘=’ explist]

retstat ::= **return** [explist] [‘;’]

label ::= ‘::’ Name ‘::’

funcname ::= Name {‘.’ Name} [‘:’ Name]

varlist ::= var {‘,’ var}

var ::= Name | prefixexp ‘[’ exp ‘]’ | prefixexp ‘.’ Name

namelist ::= Name {‘,’ Name}

explist ::= exp {‘,’ exp}

exp ::= **nil** | **false** | **true** | Number | String | ‘...’ | functiondef |

prefixexp | tableconstructor | exp binop exp | unop exp

prefixexp ::= var | functioncall | ‘(’ exp ‘)’

functioncall ::= prefixexp args | prefixexp ‘:’ Name args

args ::= ‘(’ [explist] ‘)’ | tableconstructor | String

functiondef ::= **function** funcbody

funcbody ::= ‘(’ [parlist] ‘)’ block **end**

parlist ::= namelist [‘,’ ‘...’] | ‘...’

tableconstructor ::= ‘{’ [fieldlist] ‘}’

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

field ::= ‘[’ exp ‘]’ ‘=’ exp | Name ‘=’ exp | exp

fieldsep ::= ‘,’ | ‘;’

binop ::= ‘+’ | ‘-’ | ‘*’ | ‘/’ | ‘^’ | ‘%’ | ‘..’ |
‘<’ | ‘<=’ | ‘>’ | ‘>=’ | ‘==’ | ‘~=’ |
and | **or**

unop ::= ‘-’ | **not** | ‘#’