[](http://www.lua.org/)Lua 5.3 参考手册

作者：罗伯托·伊鲁萨利姆施、路易斯·恩里克·德·菲格雷多、瓦尔德玛·塞莱斯

版权所有 © 2015–2020 Lua.org，PUC-Rio。根据[Lua许可证](http://www.lua.org/license.html)的条款免费提供。

[目录](http://www.lua.org/manual/5.3/contents.html#contents) · [索引](http://www.lua.org/manual/5.3/contents.html#index) ·[其他版本](http://www.lua.org/manual/)

1 – 简介

Lua 是一种强大、高效、轻量级、可嵌入的脚本语言。它支持过程编程、面向对象编程、函数式编程、数据驱动编程和数据描述。

Lua 将简单的过程语法与基于关联数组和可扩展语义的强大数据描述结构相结合。Lua 是动态类型的，通过使用基于寄存器的虚拟机解释字节码来运行，并具有具有增量垃圾回收的自动内存管理，使其成为配置、脚本编写和快速原型设计的理想选择。

Lua 被实现为一个库，用*干净的* C 编写，标准 C 和 C++ 的通用子集。Lua 发行版包括一个名为 lua 的主机程序，它使用 Lua 库提供完整的独立 Lua 解释器，用于交互式或批处理使用。Lua 既可以用作任何需要脚本的程序的强大、轻量级、可嵌入的脚本语言，也可以用作功能强大但轻量级且高效的独立语言。

作为一种扩展语言，Lua没有“主”程序的概念：它嵌入在主机客户端中，称为*嵌入程序*或简称为*主机*。（通常，此主机是独立的主机 路易计划。宿主程序可以调用函数来执行一段 Lua 代码，可以编写和读取 Lua 变量，并且可以注册 C 函数由 Lua 代码调用。通过使用C函数，Lua可以增强以应对各种不同的领域，从而创建共享语法框架的定制编程语言。

Lua是自由软件，如其许可证中所述，照常提供，没有任何保证。本手册中描述的实现可在Lua的官方网站 www.lua.org 上找到。

与任何其他参考手册一样，本文档在某些地方是干燥的。有关 Lua 设计背后的决策的讨论，请参阅 Lua 网站上的技术论文。有关 Lua 编程的详细介绍，请参阅 Roberto 的书《*Lua 中的编程》。*

2 – 基本概念

本节介绍该语言的基本概念。

2.1 – 值和类型

Lua 是一种*动态类型语言*。这意味着变量没有类型;只有值可以。该语言中没有类型定义。所有值都有自己的类型。

Lua 中的所有值都是*一等值*。这意味着所有值都可以存储在变量中，作为参数传递给其他函数，并作为结果返回。

Lua 中有八种基本类型：*nil*、*布尔值*、*数字*、*字符串*、*函数*、*用户数据* 、 *线程*和*表*。*nil* 类型有一个值 **nil**，其主要属性是不同于任何其他值;它通常表示缺少有用的值。*布尔*值类型有两个值：**false** 和 **true**。**nil** 和 **false** 都使条件为假;任何其他值都使条件为真。类型*数字*表示整数和实数（浮点数）。字符串类型 表示不可变的字节序列。Lua 是 8 位干净的：字符串可以包含任何 8 位值，包括嵌入的零 （'\0'）。Lua 也是与编码无关的;它不对字符串的内容做出任何假设。

*类型编号*使用两个内部表示形式或两个子类型，一个称为*整数*，另一个称为*浮点*数。Lua 对何时使用每个表示有明确的规则，但它也会根据需要在它们之间自动转换（参见 [§3.4.3](http://www.lua.org/manual/5.3/manual.html#3.4.3)）。因此，程序员可以选择主要忽略整数和浮点数之间的差异，或者完全控制每个数字的表示。标准 Lua 使用 64 位整数和双精度（64 位）浮点数，但您也可以编译 Lua，使其使用 32 位整数和/或单精度（32 位）浮点数。整数和浮点数的32位选项对于小型机器和嵌入式系统特别有吸引力。（请参阅宏 LUA\_32BITS在文件 luaconf.h.）

Lua 可以调用（和操作）用 Lua 编写的函数和用 C 编写的函数（参见 [§3.4.10](http://www.lua.org/manual/5.3/manual.html#3.4.10)）。两者都由类型*函数*表示。

*提供 userdata* 类型是为了允许将任意 C 数据存储在 Lua 变量中。用户数据值表示原始内存块。有两种类型的用户数据：*full userdata*，它是一个具有由Lua管理的内存块的对象，以及*轻用户数据*，它只是一个C指针值。用户数据在Lua中没有预定义的操作，除了分配和身份测试。通过使用*元表*，程序员可以定义完整用户数据值的操作（参见 [§2.4](http://www.lua.org/manual/5.3/manual.html#2.4)）。用户数据值不能在Lua中创建或修改，只能通过C API。这保证了主机程序拥有的数据的完整性。

线程类型表示独立的执行线程，它用于实现协程（参见 [§2.6](http://www.lua.org/manual/5.3/manual.html#2.6)）。Lua 线程与操作系统线程无关。Lua 支持所有系统上的协程，即使是那些本身不支持线程的系统。

类型*表*实现关联数组，即不仅可以将数字作为索引的数组，还可以将除 **nil** 和 NaN 之外的任何 Lua 值作为索引。（*Not a Number* 是用于表示未定义或无法表示的数值结果的特殊值，例如 0/0。表可以是*异类*的;也就是说，它们可以包含所有类型的值（**nil** 除外）。任何值为 **nil** 的键都不被视为表的一部分。相反，不属于表的任何键都具有关联的值 **nil**。

表是 Lua 中唯一的数据结构化机制;它们可用于表示普通数组、列表、符号表、集合、记录、图形、树等。为了表示记录，Lua 使用字段名称作为索引。该语言通过提供 a.name 作为 [“name”] 的语法糖来支持这种表示。在 Lua 中创建表有几种方便的方法（参见 [§3.4.9](http://www.lua.org/manual/5.3/manual.html#3.4.9)）。

与索引一样，表字段的值可以是任何类型的。特别是，由于函数是一等值，因此表字段可以包含函数。因此，表也可以携带*方法*（参见[§3.4.11](http://www.lua.org/manual/5.3/manual.html#3.4.11)）。

表的索引遵循语言中原始相等的定义。表达式 a[i] 和 a[j] 表示相同的表元素，当且仅当 i 和 j 原始相等 （即，在没有元方法的情况下相等）。特别是，具有整数值的浮点数等于它们各自的整数（例如，1.0 == 1）。为了避免歧义，任何将整数值用作键的浮点数都将转换为其各自的整数。例如，如果编写 a[2.0] = true，则插入到表中的实际键将是整数 2。（另一方面，2 和”2“ 是不同的 Lua 值，因此表示不同的表条目。

表、函数、线程和（完整）userdata 值都是*对象*：变量实际上不包含 这些值，只*包含*对它们的引用。赋值、参数传递和函数返回始终操作对此类值的引用;这些操作并不意味着任何类型的副本。

库函数[类型](http://www.lua.org/manual/5.3/manual.html#pdf-type)返回一个字符串，描述给定值的类型（参见 [§6.1](http://www.lua.org/manual/5.3/manual.html#6.1)）。

2.2 – 环境与全球环境

正如将在 §3.2 和 [§3.3.3](http://www.lua.org/manual/5.3/manual.html#3.3.3) 中讨论的那样，任何对自由名称（即不绑定到任何声明的名称）var 的引用 在语法上都转换为 \_ENV.var。此外，每个块都是在名为 \_ENV 的外部局部变量的范围内编译 的（参见 [§3.3.2](http://www.lua.org/manual/5.3/manual.html#3.3.2)），因此\_ENV本身永远不会是块中的自由名称。

尽管存在这个外部\_ENV变量和自由名称的翻译，\_ENV是一个完全常规的名称。特别是，您可以使用该名称定义新的变量和参数。每个对自由名称的引用都使用程序中该点可见的\_ENV，遵循Lua的通常可见性规则（参见[§3.5](http://www.lua.org/manual/5.3/manual.html#3.5)）。

用作\_ENV值的任何表都称为*环境*。

Lua保留了一个称为*全球环境的*杰出环境。该值保存在 C 注册表中的特殊索引中（参见 [§4.5](http://www.lua.org/manual/5.3/manual.html#4.5)）。在 Lua 中，全局变量 [\_G](http://www.lua.org/manual/5.3/manual.html#pdf-_G) 使用相同的值进行初始化。（[\_G](http://www.lua.org/manual/5.3/manual.html#pdf-_G)从不在内部使用。

当Lua加载一个块时，其\_ENV upvalue的默认值是全局环境（参见[加载](http://www.lua.org/manual/5.3/manual.html#pdf-load)）。因此，默认情况下，Lua 代码中的自由名称引用全局环境中的条目（因此，它们也称为全局*变量*）。此外，所有标准库都在全局环境中加载，并且其中的某些函数在该环境中运行。您可以使用 [load](http://www.lua.org/manual/5.3/manual.html#pdf-load)（或 [loadfile](http://www.lua.org/manual/5.3/manual.html#pdf-loadfile)）来加载具有不同环境的区块。（在 C 中，您必须加载块，然后更改其第一个上值的值。

2.3 – 错误处理

由于 Lua 是一种嵌入式扩展语言，因此所有 Lua 操作都从调用 Lua 库中函数的主机程序中的 C 代码开始。（当您独立使用 Lua 时，lua 应用程序是主机程序。每当在编译或执行 Lua 块期间发生错误时，控制权都会返回到主机，主机可以采取适当的措施（例如打印错误消息）。

Lua 代码可以通过调用错误函数显式生成[错误](http://www.lua.org/manual/5.3/manual.html#pdf-error)。如果需要在 Lua 中捕获错误，可以使用 [pcall](http://www.lua.org/manual/5.3/manual.html#pdf-pcall) 或 [xpcall](http://www.lua.org/manual/5.3/manual.html#pdf-xpcall) 在*保护模式下*调用给定函数。

每当出现错误时，都会传播*一个错误对象*（也称为*错误消息*），其中包含有关错误的信息。Lua 本身只生成错误对象为字符串的错误，但程序可能会生成任何值作为错误对象的错误。由Lua程序或其主机来处理此类错误对象。

当您使用 [xpcall](http://www.lua.org/manual/5.3/manual.html#pdf-xpcall) 或 [lua\_pcall](http://www.lua.org/manual/5.3/manual.html#lua_pcall) 时，您可以提供一个 在发生错误时要调用的消息处理程序。此函数使用原始错误对象调用，并返回新的错误对象。它在错误展开堆栈之前调用，以便它可以收集有关错误的更多信息，例如通过检查堆栈并创建堆栈回溯。此消息处理程序仍受受保护调用的保护;因此，消息处理程序中的错误将再次调用消息处理程序。如果此循环持续太久，Lua 会中断它并返回相应的消息。（仅针对常规运行时错误调用消息处理程序。对于内存分配错误或运行终结器时的错误，不会调用它。

2.4 – 元表和元方法

Lua 中的每个值都可以有一个*元表*。此*元表*是一个普通的 Lua 表，它定义了原始值在某些特殊操作下的行为。您可以通过在元表中设置特定字段来更改对值的操作行为的多个方面。例如，当非数值是加法的操作数时，Lua 会在值元表的字段“\_\_add”中检查函数。如果找到，Lua 将调用此函数来执行加法。

元表中每个事件的键是一个字符串，其事件名称以两个下划线为前缀;相应的值称为*元方法*。在前面的示例中，键是“\_\_add”，元方法是执行加法的函数。除非另有说明，否则元方法应该是函数值。

您可以使用 [getmetatable](http://www.lua.org/manual/5.3/manual.html#pdf-getmetatable) 函数查询任何值的元表。Lua 使用原始访问查询元表中的元方法（请参阅 [rawget](http://www.lua.org/manual/5.3/manual.html#pdf-rawget)）。因此，为了检索 对象 o 中事件 ev 的元方法，Lua 执行等效于以下代码的操作：

rawget（getmetatable（*o*） or {}， “\_\_*ev*”）

您可以使用 [setmetatable](http://www.lua.org/manual/5.3/manual.html#pdf-setmetatable) 函数替换表的元表。您无法从 Lua 代码更改其他类型的元表（除非使用调试库 （[§6.10](http://www.lua.org/manual/5.3/manual.html#6.10)））;您应该为此使用 C API。

表和完整用户数据具有单独的元表（尽管多个表和用户数据可以共享其元表）。所有其他类型的值共享每个类型的一个元表;也就是说，所有数字都有一个元表，所有字符串都有一个元表，依此类推。默认情况下，值没有元表，但字符串库为字符串类型设置元表（请参阅 [§6.4](http://www.lua.org/manual/5.3/manual.html#6.4)）。

元表控制对象在算术运算、按位运算、顺序比较、串联、长度运算、调用和索引中的行为方式。元表还可以定义在垃圾回收用户数据或表时要调用的函数 （[§2.5](http://www.lua.org/manual/5.3/manual.html#2.5)）。

对于一元运算符（否定、长度和按位 NOT），将使用与第一个操作数相等的虚拟第二个操作数计算和调用元方法。这个额外的操作数只是为了简化Lua的内部结构（通过使这些运算符的行为像二进制操作），并且可能会在未来的版本中删除。（对于大多数用法，这个额外的操作数是无关紧要的。

接下来给出了由元表控制的事件的详细列表。每个操作都由其相应的键标识。

* **\_\_add：**加法（+）操作。如果加法的任何操作数不是数字（也不是可强制到数字的字符串），Lua 将尝试调用元方法。首先，Lua 将检查第一个操作数（即使它是有效的）。如果该操作数未定义\_\_add的元方法，则 Lua 将检查第二个操作数。如果 Lua 可以找到元方法，它会以两个操作数作为参数调用元方法，调用的结果（调整为一个值）就是操作的结果。否则，它会引发错误。
* **\_\_sub：**减法（-）运算。类似于加法操作的行为。
* **\_\_mul：**乘法 （\*） 运算。类似于加法操作的行为。
* **\_\_div：**除法 （/） 操作。类似于加法操作的行为。
* **\_\_mod：**模 （%） 运算。类似于加法操作的行为。
* **\_\_pow：**幂 （^） 运算。类似于加法操作的行为。
* **\_\_unm：**否定（一元-）操作。类似于加法操作的行为。
* **\_\_idiv：**楼层划分 （//） 操作。类似于加法操作的行为。
* **\_\_band：**按位 AND （&） 操作。行为类似于加法运算，只是如果任何操作数既不是整数也不是可强制为整数的值，Lua 将尝试元方法（请参阅 [§3.4.3](http://www.lua.org/manual/5.3/manual.html#3.4.3)）。
* **\_\_bor：**按位 OR （|） 操作。行为类似于按位 AND 运算。
* **\_\_bxor：**按位独占 OR（二进制 ~）操作。行为类似于按位 AND 运算。
* **\_\_bnot：**按位 NOT（一元 ~）操作。行为类似于按位 AND 运算。
* **\_\_shl：**按位左移 （<<） 操作。行为类似于按位 AND 运算。
* **\_\_shr：**按位右移 （>>） 操作。行为类似于按位 AND 运算。
* **\_\_concat：**串联（.） 操作。行为类似于加法运算，只是如果任何操作数既不是字符串也不是数字（总是可以强制字符串），Lua 将尝试元方法。
* **\_\_len：**长度 （#） 操作。如果对象不是字符串，Lua 将尝试其元方法。如果存在元方法，Lua 会以对象作为参数调用它，调用的结果（始终调整为一个值）是操作的结果。如果没有元方法，但对象是一个表，则Lua使用表长度运算（参见[§3.4.7](http://www.lua.org/manual/5.3/manual.html#3.4.7)）。否则，Lua 会引发错误。
* **\_\_eq：**相等 （==） 操作。行为类似于加法操作，只是 Lua 仅在要比较的值都是表或都是完整的用户数据并且它们不原始相等时才会尝试元方法。调用的结果始终转换为布尔值。
* **\_\_lt：**小于 （<） 操作。行为类似于加法运算，不同之处在于 Lua 仅在要比较的值既不是数字也不是两个字符串时才尝试元方法。调用的结果始终转换为布尔值。
* **\_\_le：**不太相等的 （<=） 操作。与其他操作不同，不太相等的操作可以使用两个不同的事件。首先，Lua 在两个操作数中查找\_\_le元方法，就像在小于运算中一样。如果找不到这样的元方法，那么它将尝试\_\_lt元方法，假设 a <= b 等价于 not （b < a）。与其他比较运算符一样，结果始终是布尔值。（此使用 \_\_lt事件可以在将来的版本中删除;它也比实际的元方法\_\_le慢。
* **\_\_index：**索引访问操作表[键]。当表不是表或表中不存在键时，会发生此事件。元方法在表中查找。

尽管名称如此，但此事件的元方法可以是函数或表。如果是函数，则以表和键作为参数调用，调用的结果（调整为一个值）是操作的结果。如果是表，则最终结果是使用 key 索引此表的结果。（此索引是常规的，而不是原始的，因此可以触发另一个元方法。

* **\_\_newindex：**索引分配表[键] = 值。与索引事件一样，当表不是表或表中不存在键时，会发生此事件。元方法在表中查找。

与索引一样，此事件的元方法可以是函数或表。如果它是一个函数，则使用表、键和值作为参数调用它。如果是表，Lua 会使用相同的键和值对此表进行索引分配。（此赋值是常规的，而不是原始的，因此可以触发另一个元方法。

只要有\_\_newindex元方法，Lua 就不会执行基元赋值。（如有必要，元方法本身可以调用 用于执行作业的[原始集](http://www.lua.org/manual/5.3/manual.html#pdf-rawset)。

* **\_\_call：**调用操作函数（参数）。当 Lua 尝试调用非函数值（即 func 不是函数）时，会发生此事件 。元方法在函数中查找。如果存在，则调用元方法时，func 作为其第一个参数，后跟原始调用的参数 （args）。调用的所有结果都是操作的结果。（这是允许多个结果的唯一元方法。

在将表设置为某个对象的元表之前，将所有需要的元方法添加到表中是一种很好的做法。特别是，\_\_gc元方法仅在遵循此顺序时才有效（参见 [§2.5.1](http://www.lua.org/manual/5.3/manual.html#2.5.1)）。

由于元表是常规表，因此它们可以包含任意字段，而不仅仅是上面定义的事件名称。标准库中的某些函数（例如，[tostring](http://www.lua.org/manual/5.3/manual.html#pdf-tostring)）使用元表中的其他字段来实现自己的目的。

2.5 – 垃圾回收

Lua 执行自动内存管理。这意味着您不必担心为新对象分配内存或在不再需要对象时释放内存。Lua 通过运行*垃圾回收器*来收集所有*死对象*（即无法再从 Lua 访问的对象），从而自动管理内存。Lua 使用的所有内存都受到自动管理：字符串、表、用户数据、函数、线程、内部结构等。

Lua 实现了增量标记和扫描收集器。它使用两个数字来控制其垃圾收集周期：垃圾收集器暂停和*垃圾收集*器*步进乘数*。两者都使用百分点作为单位（例如，值 100 表示内部值 1）。

垃圾回收器暂停控制回收器在开始新循环之前等待的时间。值越大，收集器的攻击性越小。小于 100 的值表示收集器不会等待开始新的循环。值 200 表示收集器在开始新周期之前等待使用的总内存翻倍。

垃圾收集器步进乘数控制收集器相对于内存分配的相对速度。较大的值使收集器更具侵略性，但也会增加每个增量步骤的大小。不应使用小于 100 的值，因为它们会使收集器太慢，并可能导致收集器永远不会完成一个循环。默认值为 200，这意味着收集器以内存分配速度的“两倍”运行。

如果将步进乘法器设置为非常大的数字（大于程序可能使用的最大字节数的 10%），则收集器的行为类似于停止世界收集器。如果随后将暂停设置为 200，则收集器的行为与旧 Lua 版本中的行为相同，每次 Lua 的内存使用量翻倍时都会执行一个完整的收集。

您可以通过在 C 中调用 [lua\_gc](http://www.lua.org/manual/5.3/manual.html#lua_gc) 或在 Lua 中[收集垃圾](http://www.lua.org/manual/5.3/manual.html#pdf-collectgarbage)来更改这些号码。您还可以使用这些功能直接控制收集器（例如，停止并重新启动它）。

2.5.1 – 垃圾回收元方法

您可以为表设置垃圾收集器元方法，并使用 C API 为完整的用户数据设置垃圾收集器元方法（参见 [§2.4](http://www.lua.org/manual/5.3/manual.html#2.4)）。这些元方法也称为*终结器*。终结器允许您将 Lua 的垃圾回收与外部资源管理（例如关闭文件、网络或数据库连接或释放您自己的内存）协调起来。

对于要在收集时完成的对象（表或用户数据），必须*将其标记为*完成。当您设置对象的元表时，您可以标记要完成的对象，并且元表具有由字符串“\_\_gc”索引的字段。请注意，如果设置一个没有\_\_gc字段的元表 ，然后稍后在元表中创建该字段，则不会将该对象标记为完成。

当标记的对象成为垃圾时，垃圾回收器不会立即收集它。相反，Lua把它放在一个列表中。收集完毕后，Lua 浏览了该列表。对于列表中的每个对象，它会检查对象的\_\_gc元方法：如果它是一个函数，Lua 会以对象作为其单个参数调用它;如果元方法不是函数，Lua 干脆忽略它。

在每个垃圾回收周期结束时，对象的终结器以与在该循环中收集的对象中标记为定版的相反顺序调用对象;也就是说，要调用的第一个终结器是与程序中标记为最后的对象关联的终结器。每个终结器的执行可以在常规代码执行期间的任何时间点发生。

由于正在收集的对象仍必须由终结器使用，因此该对象（以及只能通过它访问的其他对象）必须由Lua复活。通常，这种复活是短暂的，对象内存在下一个垃圾收集周期中被释放。但是，如果终结器将对象存储在某个全局位置（例如，全局变量），则复活是永久性的。此外，如果终结器再次标记要完成的终结对象，则在无法访问该对象的下一个循环中将再次调用其终结器。在任何情况下，对象内存仅在 GC 循环中释放，在该循环中，对象无法访问且未标记为完成。

关闭状态时（请参阅[lua\_close](http://www.lua.org/manual/5.3/manual.html#lua_close)），Lua 将按照标记的顺序调用标记为完成的所有对象的终结器。如果任何终结器在该阶段标记要收集的对象，则这些标记不起作用。

2.5.2 – 弱表

弱表是其元素是*弱引用*的*表*。垃圾回收器将忽略弱引用。换句话说，如果对对象的唯一引用是弱引用，则垃圾回收器将收集该对象。

弱表可以具有弱键和/或弱值。具有弱值的表允许收集其值，但阻止收集其键。同时具有弱键和弱值的表允许同时收集键和值。在任何情况下，如果收集了键或值，则会从表中删除整个对。表的弱点由其元表的\_\_mode字段控制。如果\_\_mode字段是包含字符“k”的字符串，则表中的键较弱。如果\_\_mode包含“v”，则表中的值较弱。

具有弱键和强值的表也称为*星历表*。在星历表中，仅当值的键可访问时，该值才被视为可访问。特别是，如果对键的唯一引用来自其值，则会删除该对。

表弱点的任何更改可能仅在下一个收集周期生效。特别是，如果您将弱点更改为更强模式，Lua 可能仍会在更改生效之前从该表中收集一些项目。

只有具有显式构造的对象才会从弱表中删除。值（如数字和轻 C 函数）不受垃圾回收的影响，因此不会从弱表中删除（除非收集其关联值）。尽管字符串受垃圾回收的影响，但它们没有显式构造，因此不会从弱表中删除。

复活的对象（即，正在完成的对象和只能通过正在完成的对象访问的对象）在弱表中具有特殊行为。它们在运行终结器之前从弱值中删除，但只有在运行终结器后的下一个集合中，当这些对象实际被释放时，才会从弱键中删除。此行为允许终结器通过弱表访问与对象关联的属性。

如果弱表在收集周期中复活的对象中，则在下一个循环之前可能无法正确清除该表。

2.6 – 协程

Lua 支持协程，也称为*协作多线程*。Lua 中的协程表示独立的执行线程。但是，与多线程系统中的线程不同，协程仅通过显式调用 yield 函数来暂停其执行。

您可以通过调用 [coroutine.create](http://www.lua.org/manual/5.3/manual.html#pdf-coroutine.create) 来创建协程。它的唯一参数是一个函数，它是协程的主要函数。create 函数仅创建一个新的协程并返回该协程的句柄（线程类型的对象）;它不会启动协程。

通过调用 [coroutine.resume](http://www.lua.org/manual/5.3/manual.html#pdf-coroutine.resume) 来执行协程。当您第一次调用 [coroutine.resume](http://www.lua.org/manual/5.3/manual.html#pdf-coroutine.resume) 时，将 [coroutine.create](http://www.lua.org/manual/5.3/manual.html#pdf-coroutine.create) 返回的线程作为其第一个参数传递，coroutine 通过调用其 main 函数开始执行。传递给 [coroutine.resume](http://www.lua.org/manual/5.3/manual.html#pdf-coroutine.resume) 的额外参数将作为参数传递给该函数。协程开始运行后，它会一直运行，直到终止或*屈服*。

协程可以通过两种方式终止其执行：通常，当其主函数返回时（显式或隐式，在最后一条指令之后）;如果存在未受保护的错误，则异常。在正常终止的情况下，[coroutine.resume](http://www.lua.org/manual/5.3/manual.html#pdf-coroutine.resume) 返回 **true**，以及协程主函数返回的任何值。如果出现错误，[coroutine.resume](http://www.lua.org/manual/5.3/manual.html#pdf-coroutine.resume) 将返回 **false** 和一个错误对象。

协程通过调用[协程产生](http://www.lua.org/manual/5.3/manual.html#pdf-coroutine.yield)。当协程生成时，相应的 [coroutine.resume](http://www.lua.org/manual/5.3/manual.html#pdf-coroutine.resume) 会立即返回，即使生成发生在嵌套函数调用中（即，不是在主函数中，而是在主函数直接或间接调用的函数中）。在 yield 的情况下，coroutine.resume 也返回 **true**，以及传递给 [coroutine.yield](http://www.lua.org/manual/5.3/manual.html#pdf-coroutine.yield) 的任何值。下次恢复相同的协程时，它会从它生成的点继续执行，对 [coroutine.yield](http://www.lua.org/manual/5.3/manual.html#pdf-coroutine.yield) 的调用返回传递给 [coroutine.resume](http://www.lua.org/manual/5.3/manual.html#pdf-coroutine.resume) 的任何额外参数。

与 [coroutine.create](http://www.lua.org/manual/5.3/manual.html#pdf-coroutine.create) 一样，[coroutine.wrap](http://www.lua.org/manual/5.3/manual.html#pdf-coroutine.wrap) 函数也创建一个协程，但它不是返回协程本身，而是返回一个函数，该函数在调用时会恢复协程。传递给此函数的任何参数都作为额外的参数转到 [coroutine.resume](http://www.lua.org/manual/5.3/manual.html#pdf-coroutine.resume)。 [coroutine.wrap 返回 coroutine.resume](http://www.lua.org/manual/5.3/manual.html" \l "pdf-coroutine.wrap) 返回的所有值，除了第一个值（布尔错误代码）。与 [coroutine.resume](http://www.lua.org/manual/5.3/manual.html#pdf-coroutine.resume) 不同，[coroutine.wrap](http://www.lua.org/manual/5.3/manual.html#pdf-coroutine.wrap) 不会捕获错误;任何错误都会传播到调用方。

作为协程工作原理的示例，请考虑以下代码：

函数 foo （a）

打印（“foo”， a）

返回协程.yield（2\*a）

结束

co = coroutine.create（function （a，b）

打印（“共同主体”， a， b）

局部 r = foo（a+1）

打印（“共同体”，R）

局部 r， s = coroutine.yield（a+b， a-b）

打印（“共体”， r， s）

返回 B，“结束”

完）

print（“main”， coroutine.resume（co， 1， 10））

print（“main”， coroutine.resume（co， “r”））

print（“main”， coroutine.resume（co， “x”， “y”））

print（“main”， coroutine.resume（co， “x”， “y”））

运行它时，它会生成以下输出：

共同体 1 10

福 2

主要真 4

共体 R

主要真 11 -9

共体 x y

主真 10 端

主假 无法恢复死协程

您还可以通过 C API 创建和操作协程：请参阅[函数lua\_newthread](http://www.lua.org/manual/5.3/manual.html#lua_newthread)、[lua\_resume](http://www.lua.org/manual/5.3/manual.html#lua_resume) 和[lua\_yield](http://www.lua.org/manual/5.3/manual.html#lua_yield)。

3 – 语言

本节介绍 Lua 的词法、语法和语义。换句话说，本节介绍哪些令牌有效、如何组合它们以及它们的组合的含义。

语言结构将使用通常的扩展 BNF 表示法进行解释，其中 {a} 表示 0 或多个 a，[a] 表示可选的 *a*。非终端显示为非终端，关键字显示为**kword**，其他终端符号显示为“**=**”。Lua 的完整语法可以在 本手册末尾的 [§9](http://www.lua.org/manual/5.3/manual.html#9) 中找到。

3.1 – 词汇约定

Lua 是一种自由格式的语言。它忽略词法元素（标记）之间的空格（包括换行符）和注释，但作为名称和关键字之间的分隔符除外。

*Lua 中的名称*（也称为*标识符*）可以是任何字母、数字和下划线字符串，不以数字开头，也不是保留字。标识符用于命名变量、表字段和标签。

以下*关键字*是保留的，不能用作名称：

和休息做其他如果结束

如果函数在

本地 nil 不或重复返回

然后为真，直到

Lua 是一种区分大小写的语言：并且是一个保留字，但 And 和 AND 是两个不同的有效名称。按照惯例，程序应避免创建以下划线开头，后跟一个或多个大写字母（如 [\_VERSION](http://www.lua.org/manual/5.3/manual.html#pdf-_VERSION)）的名称。

以下字符串表示其他标记：

+ - \* / % ^ #

& ~ | << >> //

== ~= <= >= < > =

( ) { } [ ] ::

; : , . .. ...

*短文本字符串*可以通过匹配的单引号或双引号分隔，并且可以包含以下类似 C 的转义序列：“\a”（钟形）、“\b”（退格）、“\f”（表单进送）、“\n”（换行符）、“\r”（回车符）、“\t”（水平制表符）、“\v”（垂直制表符）、“\\”（反斜杠）、“\”' （引号 [双引号]） 和 '\'' （撇号 [单引号]）。反斜杠后跟换行符会导致字符串中出现换行符。转义序列 '\z' 跳过以下空格字符的范围，包括换行符;将长文本字符串分隔并缩进为多行而不将换行符和空格添加到字符串内容中特别有用。短文本字符串不能包含未转义的换行符，也不能包含未形成有效转义序列的转义。

我们可以通过其数值（包括嵌入的零）指定短文本字符串中的任何字节。这可以使用转义序列 \x XX（其中 *XX* 是正好包含两个十六进制数字的序列）或转义序列 \*ddd*（其中 *ddd* 是最多三个十进制数字的序列）来完成。（请注意，如果十进制转义序列后跟一个数字，则必须使用正好三个数字来表示。

Unicode 字符的 UTF-8 编码可以插入到带有转义序列 \u{XXX} 的文本字符串中（请注意必需的括括号），其中 *XXX* 是表示字符代码点的一个或多个十六进制数字序列。

文本字符串也可以使用*用长括号*括起来的长格式来定义。我们将*级别 n 的左长*括号定义为一个左方括号，后跟 *n* 个等号，后 跟另一个左方括号。因此，级别 0 的左长括号写为 [[，级别 1 的左长括号写为 [=[，依此类推。右长括号的定义类似;例如，级别 4 的右长括号写为 ]====]。长*文本*以任何级别的开始长括号开始，到同一级别的第一个结束长括号结束。它可以包含除同一级别的右括号之外的任何文本。这种括号形式的文本可以运行多行，不解释任何转义序列，并忽略任何其他级别的长括号。任何类型的行尾序列（回车符、换行符、回车后跟换行符或换行符后跟回车符）都将转换为简单换行符。

为方便起见，当左长括号紧跟换行符时，字符串中不包含换行符。例如，在使用 ASCII 的系统中（其中“a”编码为 97，换行符编码为 10，“1”编码为 49），下面的五个文字字符串表示相同的字符串：

a = 'alo\n123“'

a = “alo\n123\””

a = '\97Lo\10\04923“'

a = [[alo

123"]]

a = [==[

阿洛

123“]==]

文本字符串中不受先前规则明确影响的任何字节都表示自身。但是，Lua 在文本模式下打开文件进行分析，并且系统文件函数可能会出现某些控制字符的问题。因此，将非文本数据表示为带引号的文本更安全，并为非文本字符提供显式转义序列。

*数字常量*（或*数字*）可以用可选的小数部分和可选的小数指数来写，用字母“e”或“E”标记。Lua 还接受以 0x 或 0X 开头的十六进制常量。十六进制常量还接受可选的小数部分加上可选的二进制指数，由字母“p”或“P”标记。带有基数点或指数的数字常量表示浮点数;否则，如果其值适合整数，则表示整数。有效整数常量的示例包括

3 345 0xff 0xBEBADA

有效浮点常量的示例包括

3.0 3.1416 314.16e-2 0.31416e1 34e1

0x0.1E 0xA23p-4 0X1.921FB54442D18P+1

*注释*以字符串外任何位置的双连字符 （--） 开头。如果紧跟在 -- 后面的文本 不是左长括号，则注释是*短注释*，一直持续到行尾。否则，它是一个长*注释*，一直持续到相应的右长括号。长注释经常用于暂时禁用代码。

3.2 – 变量

变量是存储值的地方。Lua 中有三种变量：全局变量、局部变量和表字段。

单个名称可以表示全局变量或局部变量（或函数的形式参数，这是一种特定类型的局部变量）：

var ：：= 名称

名称表示标识符，如 [§3.1](http://www.lua.org/manual/5.3/manual.html#3.1) 中所定义。

除非明确声明为局部变量，否则任何变量名都假定为全局变量名（参见 [§3.3.7](http://www.lua.org/manual/5.3/manual.html#3.3.7)）。局部变量是*词法*作用域的：局部变量可以通过在其作用域内定义的函数自由访问（参见[§3.5](http://www.lua.org/manual/5.3/manual.html#3.5)）。

在第一次赋值变量之前，其值为 **nil**。

方括号用于为表编制索引：

var ：：= 前缀exp '**[**' exp '**]**'

访问表字段的含义可以通过元表更改（参见 [§2.4](http://www.lua.org/manual/5.3/manual.html#2.4)）。

语法变量。名称只是 var[“名称”] 的语法糖：

var ：：= 前缀exp '**.**' 名称

对全局变量 x 的访问等效于 \_ENV.x。由于块的编译方式，\_ENV从来都不是全局名称（参见 [§2.2](http://www.lua.org/manual/5.3/manual.html#2.2)）。

3.3 – 声明

Lua支持一组几乎传统的语句，类似于Pascal或C中的语句。此集合包括赋值、控制结构、函数调用和变量声明。

3.3.1 – 块

块是按顺序执行的语句列表：

块 ：：= {统计}

Lua 有*空*语句，允许您用分号分隔语句，用分号开始块或按顺序编写两个分号：

状态 ：：= '**;**'

函数调用和赋值可以以左括号开头。这种可能性导致Lua的语法含糊不清。请考虑以下片段：

a = b + c

（打印或 io.write）（“完成”）

语法可以通过两种方式看到它：

a = b + c（print or io.write）（'done'）

a = b + c;（打印或 io.write）（“完成”）

当前解析器总是以第一种方式看到此类构造，将左括号解释为调用参数的开头。为了避免这种歧义，最好始终在前面加上以括号开头的分号语句：

;(p rint or io.write）（'done'）

可以显式分隔块以生成单个语句：

统计 ：：= **做** 块 **结束**

显式块对于控制变量声明的范围很有用。显式块有时也用于在另一个块的中间添加 return 语句（参见 [§3.3.4](http://www.lua.org/manual/5.3/manual.html#3.3.4)）。

3.3.2 – 块

Lua的编译单位称为*块*。从语法上讲，块只是一个块：

块 ：：= 块

Lua 将块作为具有可变参数数量的匿名函数的主体进行处理（参见 [§3.4.11](http://www.lua.org/manual/5.3/manual.html#3.4.11)）。因此，块可以定义局部变量、接收参数和返回值。此外，这种匿名函数被编译为在称为\_ENV的外部局部变量的作用域中（参见§[2.2](http://www.lua.org/manual/5.3/manual.html#2.2)）。生成的函数始终将 \_ENV 作为其唯一的上行值，即使它不使用该变量也是如此。

块可以存储在文件或主机程序内的字符串中。要执行一个块，Lua 首先*加载*它，将块的代码预编译为虚拟机的指令，然后 Lua 使用虚拟机的解释器执行编译的代码。

块也可以预编译为二进制形式;有关详细信息，请参阅程序 luac 和函数 [string.dump](http://www.lua.org/manual/5.3/manual.html#pdf-string.dump)。源代码和编译形式的程序是可以互换的;Lua 会自动检测文件类型并采取相应措施（请参阅[加载](http://www.lua.org/manual/5.3/manual.html#pdf-load)）。

3.3.3 – 分配

Lua 允许多个分配。因此，赋值的语法在左侧定义变量列表，在右侧定义表达式列表。两个列表中的元素都用逗号分隔：

状态 ：：= 变量列表 '**=**' 表达式列表

varlist ：：= var {'**，**' var}

explist ：：= exp {'**，**' exp}

表达式在 [§3.4](http://www.lua.org/manual/5.3/manual.html#3.4) 中讨论。

在赋值之前，值列表将*调整*为变量列表的长度。如果值多于所需值，则会丢弃多余的值。如果值少于所需值，则会根据需要使用任意数量的 **nil** 扩展列表。如果表达式列表以函数调用结尾，则该调用返回的所有值在调整之前进入值列表（除非调用括在括号中;请参阅 [§3.4](http://www.lua.org/manual/5.3/manual.html#3.4)）。

赋值语句首先计算其所有表达式，然后才执行赋值。因此代码

i = 3

并且，a[i] = i+1， 20

将 A[3] 设置为 20，而不影响 A[4]，因为 A[i] 中的 i 在赋值 4 之前被计算（到 3）。同样，该行

x， y = y， x

交换 x 和 y 的值，以及

x， y， z = y， z， x

循环排列 x、y 和 z 的值。

对全局名称 x = val 的赋值等效于赋值 \_ENV.x = val（参见 [§2.2](http://www.lua.org/manual/5.3/manual.html#2.2)）。

表字段和全局变量（实际上也是表字段）赋值的含义可以通过元表更改（参见 [§2.4](http://www.lua.org/manual/5.3/manual.html#2.4)）。

3.3.4 – 控制结构

**if**、**while** 和 **repeat** 的控制结构具有通常的含义和熟悉的语法：

统计 ：：= **而** exp **做** 块 **结束**

统计 ：：= **重复** 阻止 **直到** exp

stat ：：= **如果** exp **则**阻止 {**elseif** exp **then** block } [**else** block] **end**

Lua 还有一个 **for** 语句，有两种风格（参见 [§3.3.5](http://www.lua.org/manual/5.3/manual.html#3.3.5)）。

控制结构的条件表达式可以返回任何值。假和**零**都被认为是假的。所有不同于 **nil** 和 **false** 的值都被视为 true（特别是，数字 0 和空字符串也为 true）。

在 **repeat**–till 循环中，内部块不会在 **till** 关键字处结束，而仅在条件之后结束。因此，条件可以引用在循环块内声明的局部变量。

**goto** 语句将程序控件转移到标签。出于语法原因，Lua 中的标签也被视为语句：

统计 ：：= **转到** 名称

统计 ：：= 标签

标签 ：：= “：：” 名称 “：**：**”

标签在定义它的整个块中可见，但定义了相同名称标签的嵌套块内部和嵌套函数内部除外。goto 可以跳转到任何可见标签，只要它不进入局部变量的范围。

标签和空语句称为 *void 语句*，因为它们不执行任何操作。

**break** 语句终止 **while**、**repeat**或 **for** 循环的执行 ，跳到循环后的下一条语句：

统计 ：：= **休息**

**中断**结束最里面的封闭循环。

**return** 语句用于从函数或块（匿名函数）返回值。函数可以返回多个值，因此 **return** 语句的语法为

stat ：：= **返回** [explist] ['**;**']

**return** 语句只能写为块的最后一个语句。如果确实需要在块的中间返回，则可以使用显式的内部块，就像成语中所做的返回结束一样，因为现在**返回**是其（内部）块中的最后一个语句。

3.3.5 – 对于语句

**for** 语句有两种形式：一种是数字形式，一种是泛型。

**数字 for** 循环重复代码块，而控制变量贯穿算术级数。它具有以下语法：

stat ：：= **对于**名称 '**=**' exp '，' exp ['**，**' exp] 做块**结束**

从第一个 exp 的值开始重复*该块*的名称，直到它通过第三个 *exp* 的步骤传递第二个 *exp*。更准确地说，**像这样的 for** 语句

对于 v = *e1*、*e2*、*e3* do *块*端

等效于代码：

的

本地*变量*， *限制*， *步长* = tonumber（e1）， tonumber（*e2*）， tonumber（*e3*）

如果不是（*var* 和 *limit* and *step*），则 error（） 结束

*var* = *var* - *步长*

虽然真的做

*其中* = *变量* + *步长*

如果（步骤 >= 0 且 var > 限制）或（*步骤* < 0 且 *var* < *限制*）则

破

结束

本地 v = *var*

*块*

结束

结束

请注意以下几点：

* 在循环开始之前，所有三个控件表达式仅计算一次。它们都必须产生数字。
* *var*、*limit* 和 *step* 是不可见的变量。此处显示的名称仅用于解释目的。
* 如果第三个表达式（步骤）不存在，则使用步骤 1。
* 您可以使用 **break** 和 **goto** 退出 **for** 循环。
* 循环变量 v 是循环主体的局部变量。如果在循环后需要其值，请在退出循环之前将其分配给另一个变量。

泛型 **for** 语句适用于称为*迭代器的*函数。在每次迭代中，都会调用迭代器函数以生成一个新值，当此新值为 **nil** 时停止。泛型 **for** 循环具有以下语法：

stat ：：= 对于 explist **中的**姓名列表**，执行**阻止**结束**

名称列表 ：：= 名称 {'**，**' 名称}

A **表示**语句，例如

对于 *var\_1*、···、*var\_n* 在 *explist* 中做*块*结束

等效于代码：

的

局部 *f*， *s*， *var* = *explist*

虽然真的做

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

如果 *var\_1* == nil 则中断结束

*其中* = *var\_1*

*块*

结束

结束

请注意以下几点：

* *explist* 只被评估一次。其结果是一个迭代器函数、一个*状态*和一个第一个*迭代器变量*的初始值。
* *f*、*s* 和 *var* 是不可见的变量。此处的名称仅用于解释目的。
* 您可以使用**中断**退出 **for** 循环。
* 循环变量*var\_i*是循环的局部变量;不能在 for 结束之后使用它们的值 。如果需要这些值，请在中断或退出循环之前将它们分配给其他变量。

3.3.6 – 作为语句调用的函数

为了允许可能的副作用，函数调用可以作为语句执行：

状态 ：：= 函数调用

在这种情况下，所有返回的值都将被丢弃。函数调用在 [§3.4.10](http://www.lua.org/manual/5.3/manual.html#3.4.10) 中进行了说明。

3.3.7 – 本地声明

局部变量可以在块内的任何位置声明。声明可以包含初始赋值：

统计 ：：= **本地**名称列表 ['**=**' explist]

如果存在，则初始赋值具有与多重赋值相同的语义（参见 [§3.3.3](http://www.lua.org/manual/5.3/manual.html#3.3.3)）。否则，所有变量都使用 **nil** 初始化。

块也是一个块（参见 [§3.3.2](http://www.lua.org/manual/5.3/manual.html#3.3.2)），因此局部变量可以在任何显式块之外的块中声明。

局部变量的可见性规则在 [§3.5](http://www.lua.org/manual/5.3/manual.html#3.5) 中进行了解释。

3.4 – 表达式

Lua 中的基本表达式如下：

exp ：：= 前缀exp

exp ：：= **无** | **假** | **真**

exp ：：= 数字

exp ：：= 文字字符串

exp ：：= 函数定义

exp ：：= 表构造函数

exp ：：= '**...**'

exp ：：= exp binop exp

exp ：：= 项目厅 exp

前缀exp ：：= var |函数调用|'**（**' exp '**）**'

数字和文字字符串在 §3.1 中解释;变量在 §3.2 中解释;函数定义在 §3.4.11 中解释;函数调用在 §3.4.10 中解释;表构造函数在 [§](http://www.lua.org/manual/5.3/manual.html#3.4.11) [3.4.9](http://www.lua.org/manual/5.3/manual.html#3.2) 中解释。瓦拉格表达式，用三个点表示（'...'），只能在直接在 vararg 函数中使用;它们在 [§3.4.11](http://www.lua.org/manual/5.3/manual.html#3.4.11) 中有解释。

二元运算符包括算术运算符（参见 §3.4.1）、按位运算符（参见 §3.4.2）、关系运算符（参见 §3.4.4）、逻辑运算符（参见 §3.4.5）和串联运算符（参见 [§3.4.6](http://www.lua.org/manual/5.3/manual.html#3.4.5)）。一元运算符包括一元减号（参见 [§3.4.1](http://www.lua.org/manual/5.3/manual.html#3.4.1)）、一元位 NOT（参见 §3.4.2）、一元逻辑**非**（参见 §3.4.5）和一元*长度运算符*（参见 [§3.4.7](http://www.lua.org/manual/5.3/manual.html#3.4.5)）。

函数调用和 vararg 表达式都可以生成多个值。如果函数调用用作语句（参见 [§3.3.6](http://www.lua.org/manual/5.3/manual.html#3.3.6)），则其返回列表将调整为零元素，从而丢弃所有返回值。如果表达式用作表达式列表的最后一个（或唯一）元素，则不进行任何调整（除非表达式括在括号中）。在所有其他上下文中，Lua 将结果列表调整为一个元素，要么丢弃除第一个值之外的所有值，要么在没有值时添加单个 **nil**。

以下是一些示例：

f（） -- 调整为 0 个结果

g（f（）， x） -- f（） 调整为 1 个结果

g（x， f（）） -- g 得到 x 加上 f（） 的所有结果

a，b，c = f（）， x -- f（） 调整为 1 个结果 （c 得到零）

a，b = ... -- a 得到第一个 vararg 参数，b 得到

-- 第二个（如果存在，则 A 和 B 都可以得到 nil

-- 没有相应的 vararg 参数）

a，b，c = x， f（） -- f（） 调整为 2 个结果

a，b，c = f（） -- f（） 调整为 3 个结果

返回 f（） -- 返回来自 f（） 的所有结果

返回 ... -- 返回所有收到的 vararg 参数

返回 x，y，f（） -- 返回 x、y 和来自 f（） 的所有结果

{f（）} -- 创建一个包含 f（） 的所有结果的列表

{...} -- 创建一个包含所有变量参数的列表

{f（）， nil} -- f（） 调整为 1 个结果

括在括号中的任何表达式始终只生成一个值。因此，（f（x，y，z）） 始终是单个值，即使 f 返回多个值也是如此。（的值 （f（x，y，z）） 是 f 或 **nil** 返回的第一个值，如果 f 不返回任何值。

3.4.1 – 算术运算符

Lua 支持以下算术运算符：

* **+：** 加法
* **-：**减法
* **\*：** 乘法
* **/：** 浮点分割
* **：** 地板划分
* **%：** 模块
* **^：** 幂
* **-：** 一元减号

除幂和浮点除法外，算术运算符的工作方式如下：如果两个操作数都是整数，则对整数执行运算，结果为整数。否则，如果两个操作数都是可以转换为数字的数字或字符串（参见 [§3.4.3](http://www.lua.org/manual/5.3/manual.html#3.4.3)），则它们将转换为浮点数，则按照浮点运算的常规规则（通常是 IEEE 754 标准）执行运算，结果为浮点数。

幂和浮点除法 （/） 总是将它们的操作数转换为浮点数，结果始终是浮点数。幂使用 ISO C 函数 pow，因此它也适用于非整数指数。

下限除法 （//） 是将商舍入为负无穷大（即其操作数除法的下限）的除法。

模被定义为将商四舍五入到负无穷大（下限除法）的除法的余数。

在整数算术中溢出的情况下，根据二进制补码算术的通常规则，所有操作都会环绕。（换句话说，它们返回相等模的唯一可表示整数 *264* 到数学结果。

3.4.2 – 按位运算符

Lua 支持以下按位运算符：

* **&：**按位和
* **|：**按位或
* **~：**按位独占 OR
* **>>：**右移
* **<<：**左移
* **~：**一元位 NOT

所有按位运算都将其操作数转换为整数（参见 [§3.4.3](http://www.lua.org/manual/5.3/manual.html#3.4.3)），对这些整数的所有位进行操作，并生成一个整数。

左右移位都用零填充空位。负位移向另一个方向;绝对值等于或大于整数中位数的位移将导致零（因为所有位都移出）。

3.4.3 – 胁迫和转换

Lua 在运行时提供了某些类型和表示之间的一些自动转换。按位运算符始终将浮点操作数转换为整数。幂和浮点除法始终将整数操作数转换为浮点数。应用于混合数（整数和浮点数）的所有其他算术运算将整数操作数转换为浮点数;这称为*通常规则*。C API 还会根据需要将整数转换为浮点数，将浮点数转换为整数。此外，除了字符串之外，字符串连接还接受数字作为参数。

Lua 还会将字符串转换为数字，只要需要数字。

在从整数到浮点数的转换中，如果整数值精确表示为浮点数，那就是结果。否则，转换将获得最接近的较高或最接近的较低可表示值。这种转换永远不会失败。

从浮点数到整数的转换检查浮点数是否具有作为整数的精确表示形式（即，浮点数具有整数值，并且它在整数表示范围内）。如果是这样，则表示就是结果。否则，转换将失败。

从字符串到数字的转换如下：首先，按照字符串的语法和Lua词法分析器的规则，将字符串转换为整数或浮点数。（字符串还可以具有前导空格和尾随空格以及符号。然后，将结果数字（浮点数或整数）转换为上下文所需的类型（浮点数或整数）（例如，强制转换的操作）。

从字符串到数字的所有转换都接受点和当前区域设置标记作为基数字符。（然而，Lua词法分析器只接受一个点。

从数字到字符串的转换使用非指定的人类可读格式。要完全控制如何将数字转换为字符串，请使用字符串库中的 format 函数（请参阅 [string.format](http://www.lua.org/manual/5.3/manual.html#pdf-string.format)）。

3.4.4 – 关系运算符

Lua 支持以下关系运算符：

* **==：** 相等
* **~=：** 不等式
* **<：**小于
* **>：**大于
* **<=：**小于或等于
* **>=：**大于或等于

这些运算符总是导致 **false** 或 **true**。

相等 （==） 首先比较其操作数的类型。如果类型不同，则结果为 **false**。否则，将比较操作数的值。字符串以明显的方式进行比较。如果数字表示相同的数学值，则它们相等。

表、用户数据和线程通过引用进行比较：只有当两个对象是同一对象时，它们才被视为相等。每次创建新对象（表、用户数据或线程）时，此新对象都不同于以前存在的任何对象。闭包总是等于自身。具有任何可检测差异（不同的行为，不同的定义）的闭包总是不同的。在不同时间创建但没有可检测差异的闭包可能被归类为相等或不相等（取决于内部缓存详细信息）。

您可以使用“eq”元方法更改Lua比较表和用户数据的方式（参见§[2.4](http://www.lua.org/manual/5.3/manual.html#2.4)）。

相等比较不会将字符串转换为数字，反之亦然。因此，“0”==0 的计算结果为 **false**， t[0] 和 t[“0”] 表示表中的不同条目。

运算符 ~= 正是相等 （==） 的否定。

订单运算符的工作方式如下。如果两个参数都是数字，则根据它们的数学值（无论它们的子类型如何）比较它们。否则，如果两个参数都是字符串，则根据当前区域设置比较它们的值。否则，Lua 会尝试调用 “lt” 或 “le” 元方法（参见 [§2.4](http://www.lua.org/manual/5.3/manual.html#2.4)）。比较 a > b 转换为 b < a ， a >= b 转换为 b <= a。

根据 IEEE 754 标准，NaN 既不小于、不等于也不大于任何值（包括其自身）。

3.4.5 – 逻辑运算符

Lua 中的逻辑运算符是 **and**、**or** 和 **not**。与控制结构（参见 [§3.3.4](http://www.lua.org/manual/5.3/manual.html#3.3.4)）一样，所有逻辑运算符都认为 **false** 和 **nil** 都是假的，其他任何内容都认为是真。

否定运算符**并不总是**返回 **false** 或 **true**。如果此值为 **false** 或 **nil**，则返回其第一个参数;否则，则返回其第二个参数。析取运算符 **或 如果**此值不同于 **nil** 和 **false**，则返回其第一个参数;否则，**或**返回其第二个参数。和 和 **或** 都使用短路计算;也就是说，仅在必要时计算第二个操作数。以下是一些示例：

10 或 20 --> 10

10 或错误 （） --> 10

nil 或 “a” --> “a”

零和 10 --> 零

false 和 error（） --> false

假和零 -->假

假或无 --> 无

10 和 20 --> 20

（在本手册中，--> 表示上述表达式的结果。

3.4.6 – 串联

Lua 中的字符串连接运算符由两个点 （'..').如果两个操作数都是字符串或数字，则根据 [§3.4.3](http://www.lua.org/manual/5.3/manual.html#3.4.3) 中描述的规则将它们转换为字符串。否则， 将调用\_\_concat元方法（参见 [§2.4](http://www.lua.org/manual/5.3/manual.html#2.4)）。

3.4.7 – 长度运算符

长度运算符由一元前缀运算符 # 表示。

字符串的长度是其字节数（即，当每个字符为一个字节时，字符串长度的通常含义）。

应用于表的长度运算符返回该表中的边框。 表 t 中的*边框*是满足以下条件的任何自然数：

（边框 == 0 或 t[边框] ~= nil） 和 t[边框 + 1] == nil

换句话说，边框是表格中的任何（自然）索引，其中非 nil 值后跟 nil 值（或零，当索引 1 为 nil 时）。

只有一个边框的表称为*序列*。例如，表 {10， 20， 30， 40， 50} 是一个序列，因为它只有一个边框 （5）。表 {10， 20， 30， nil， 50} 有两个边框（3 和 5），因此它不是一个序列。表 {nil， 20， 30， nil， nil， 60， nil} 有三个边框（0、3 和 6），因此它也不是一个序列。表 {} 是边框为 0 的序列。请注意，非自然键不会干扰表是否为序列。

当 t 是一个序列时，#t 返回其唯一的边界，这与序列长度的直观概念相对应。当 t 不是序列时，#t可以返回其任何边界。（确切的表示取决于表的内部表示形式的详细信息，而内部表示形式又取决于表的填充方式及其非数字键的内存地址。

表长度的计算保证最差时间为 *O（log n），*其中 *n* 是表中最大的自然键。

程序可以通过 \_\_len 元方法修改除字符串之外的任何值的长度运算符的行为（参见 [§2.4](http://www.lua.org/manual/5.3/manual.html#2.4)）。

3.4.8 – 优先级

Lua 中的运算符优先级如下表所示，从低优先级到高优先级：

或

和

< > <= >= ~= ==

|

~

&

<< >>

..

+ -

\* / // %

一元运算符（不是 # - ~）

^

像往常一样，您可以使用括号来更改表达式的优先级。串联 （'.'） 和幂 （'^'） 运算符是右关联运算符。所有其他二元运算符都保留关联。

3.4.9 – 表构造函数

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

表构造函数 ：：= '**{**' [字段列表] '**}**'

字段列表 ：：= 字段 {字段EP 字段} [字段EP]

字段 ：：= '**[**' exp '**]**' '**=**' exp |名称 '**=**' exp | exp

菲尔塞普 ：：= '**，**' |'**;**'

形式为 [exp1] = exp2 的每个字段都会向新表添加一个带有键 exp1 和值 exp2 的条目。表单名称 = exp 的字段等效于 [“名称”] = exp。最后，exp 形式的字段 等效于 [i] = exp，其中 i 是从 1 开始的连续整数。其他格式的字段不会影响此计数。例如

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

相当于

的

局部 t = {}

t[f（1）] = g

t[1] = “x” -- 第一个经验

t[2] = “y” -- 第二个经验

t.x = 1 -- t[“x”] = 1

t[3] = f（x） -- 3rd exp

t[30] = 23

t[4] = 45 -- 第 4 次经验

a = t

结束

构造函数中赋值的顺序未定义。（仅当存在重复键时，此顺序才相关。

如果列表中的最后一个字段具有 exp 形式，并且表达式是函数调用或 vararg 表达式，则此表达式返回的所有值将连续进入列表（请参阅 [§3.4.10](http://www.lua.org/manual/5.3/manual.html#3.4.10)）。

字段列表可以有一个可选的尾随分隔符，以方便机器生成的代码。

3.4.10 – 函数调用

Lua 中的函数调用具有以下语法：

函数调用 ：：= 前缀EXP 参数

在函数调用中，将计算第一个前缀 exp 和参数。如果前缀exp的值具有类型函数，则使用给定的参数调用此函数。否则，将调用前缀exp“call”元方法，其第一个参数是前缀exp的值，后跟原始调用参数（参见[§2.4](http://www.lua.org/manual/5.3/manual.html#2.4)）。

表格

函数调用 ：：= 前缀exp '**：**' 名称参数

可用于调用“方法”。调用 v：name（*args*） 是 v.name（v，*args*） 的语法糖，只是 v 只计算一次。

参数具有以下语法：

参数 ：：= '**（**' [explist] '**）**'

参数 ：：= 表构造函数

参数 ：：= 文字字符串

在调用之前计算所有参数表达式。f{fields} 形式的调用是 f（{*fields*}） 的语法糖;也就是说，参数列表是一个新表。形式为f'string'（或f“string”或 f[[string]]）的调用是f（'*string*'）的语法糖;也就是说，参数列表是单个文本字符串。

表单返回*函数*调用的调用称为*尾部调用*。Lua 实现了正确的尾部调用（或适当的尾部*递归*）：在尾部调用中，被调用的函数重用调用函数的堆栈条目。因此，程序可以执行的嵌套尾调用的数量没有限制。但是，尾部调用会擦除有关调用函数的任何调试信息。请注意，尾部调用仅发生在特定语法中，其中返回具有一个函数调用作为参数;此语法使调用函数完全返回被调用函数的返回。因此，以下示例都不是尾调用：

返回 （f（x）） -- 结果调整为 1

返回 2 \* f（x）

返回 x， f（x） -- 其他结果

f（x）;返回 -- 丢弃的结果

返回 x 或 f（x） -- 结果调整为 1

3.4.11 – 函数定义

函数定义的语法是

函数定义 ：：= **函数** 函数体

函数体 ：：= '**（**' [parlist] '**）**' 块**结束**

以下语法糖简化了函数定义：

stat ：：= 函数 函数名称函数体

stat ：：= **本地函数** Name funcbody

函数名称 ：：= 名称 {'**.**' 名称} ['**：**' 名称]

声明

函数 f （） *主体* 端

翻译为

f = 函数 （） *主体* 端

声明

函数 T.A.B.C.F （ *） 身体* 端

翻译为

T.A.B.C.F = FUNCTION （ *） 主体* 端

声明

局部函数 f （） *主体* 端

翻译为

本地 f;f = 函数 （） *主体* 端

不要

局部 f = 函数 （） *主体* 端

（仅当函数主体包含对 f 的引用时，这才会有所不同。

函数定义是一个可执行表达式，其值具有类型*函数*。当Lua预编译一个块时，它的所有函数体也会被预编译。然后，每当 Lua 执行函数定义时，函数就会*实例化*（或*关闭*）。此函数实例（或*闭包*）是表达式的最终值。

参数充当使用参数值初始化的局部变量：

帕里斯特 ：：= 名称列表 ['**，**' '**...**'] |'**......**'

调用函数时，参数列表将调整为参数列表的长度，除非该函数是 *vararg* 函数，该函数由三个点 （'...'） 在其参数列表的末尾。vararg 函数不调整其参数列表;相反，它收集所有额外的参数，并通过 *vararg* 表达式将它们提供给函数，该表达式也写为三个点。此表达式的值是所有实际额外参数的列表，类似于具有多个结果的函数。如果 vararg 表达式在另一个表达式内部或表达式列表中间使用，则其返回列表将调整为一个元素。如果将表达式用作表达式列表的最后一个元素，则不会进行任何调整（除非最后一个表达式括在括号中）。

例如，请考虑以下定义：

函数 f（a， b） 结束

函数 g（a， b， ...） 结束

函数 r（） 返回 1，2，3 结束

然后，我们有从参数到参数和 vararg 表达式的以下映射：

调用参数

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， ... --> （无）

g（3， 4） a=3， b=4， ... --> （无）

g（3， 4， 5， 8） a=3， b=4， ... --> 5 8

g（5， r（）） a=5， b=1， ... --> 2 3

结果使用 **return** 语句返回（参见 [§3.3.4](http://www.lua.org/manual/5.3/manual.html#3.3.4)）。如果控件到达函数的末尾而没有遇到 **return** 语句，则该函数返回时没有结果。

函数可能返回的值数存在系统相关限制。此限制保证大于 1000。

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

函数 T.A.B.C：F （*PARAMS*） *主体* 端

是句法糖

T.A.B.C.F = 函数（自身， *参数*） *主体* 端

3.5 – 可见性规则

Lua是一种词汇范围的语言。局部变量的作用域从其声明后的第一条语句开始，一直持续到包含该声明的最内层块的最后一个非 void 语句。请考虑以下示例：

x = 10 -- 全局变量

执行 -- 新建块

局部 x = x -- 新的“x”，值为 10

打印（X） --> 10

x = x+1

做 -- 另一个块

局部 x = x+1 -- 另一个“x”

打印（x） --> 12

结束

打印（x） --> 11

结束

打印（x） --> 10（全局的）

请注意，在像局部 x = x 这样的声明中，正在声明的新 x 尚未在范围内，因此第二个 x 引用外部变量。

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

请注意，**每次**执行局部语句都会定义新的局部变量。请考虑以下示例：

a = {}

局部 x = 20

对于 i=1，10 do

局部 y = 0

a[i] = 函数 （） y=y+1;返回 x+y 结束

结束

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

4 – 应用程序接口

本节介绍适用于 Lua 的 C API，即主机程序可用于与 Lua 通信的 C 函数集。所有 API 函数以及相关的类型和常量都在头文件 lua.h 中声明。

即使我们使用术语“函数”，API 中的任何工具也可能作为宏提供。除非另有说明，否则所有此类宏都只使用一次每个参数（第一个参数除外，它始终是 Lua 状态），因此不会产生任何隐藏的副作用。

与大多数 C 库一样，Lua API 函数不会检查其参数的有效性或一致性。但是，您可以通过使用定义的宏LUA\_USE\_APICHECK编译 Lua 来更改此行为。

Lua 库是完全可重入的：它没有全局变量。它将所需的所有信息保存在一个动态结构中，称为*Lua状态*。

每个 Lua 状态都有一个或多个线程，它们对应于独立的协作执行行。[类型 lua\_State](http://www.lua.org/manual/5.3/manual.html#lua_State)（尽管其名称）是指线程。（间接地，通过线程，它也引用与线程关联的Lua状态。

指向线程的指针必须作为第一个参数传递给库中的每个函数，但 [lua\_newstate](http://www.lua.org/manual/5.3/manual.html#lua_newstate) 除外，从头开始创建 Lua 状态并返回指向 处于新状态的主线程的指针。

4.1 – 堆栈

Lua 使用*虚拟堆栈*将值传入和传出 C。此堆栈中的每个元素都表示一个 Lua 值（**nil**、数字、字符串等）。API 中的函数可以通过它们接收的 Lua 状态参数访问此堆栈。

每当 Lua 调用 C 时，被调用的函数都会获得一个新的堆栈，该堆栈独立于以前的堆栈和仍然处于活动状态的 C 函数堆栈。此堆栈最初包含 C 函数的任何参数，C 函数可以存储临时 Lua 值，并且必须推送其结果以返回给调用方（请参阅[lua\_CFunction](http://www.lua.org/manual/5.3/manual.html#lua_CFunction)）。

为方便起见，API 中的大多数查询操作不遵循严格的堆栈规则。相反，它们可以使用索引引用堆栈中的任何元素：正索引表示绝对堆栈位置（从 1 开始）;负索引表示相对于堆栈顶部的偏移量。更具体地说，如果堆栈有 n 个元素，则索引 1 表示第一个元素（即首先推送到堆栈上的元素），索引 n 表示最后一个元素;索引 -1 也表示最后一个元素（即顶部的元素），索引 -*n* 表示第一个元素。

4.2 – 堆栈大小

当您与 Lua API 交互时，您有责任确保一致性。特别是，*您负责控制堆栈溢出*。您可以使用函数[lua\_checkstack](http://www.lua.org/manual/5.3/manual.html#lua_checkstack)来确保堆栈有足够的空间来推送新元素。

每当Lua调用C时，它都会确保堆栈至少有LUA\_MINSTACK个额外插槽的空间。 LUA\_MINSTACK定义为 20，因此通常您不必担心堆栈空间，除非您的代码具有将元素推送到堆栈上的循环。

当您调用没有固定数量结果的 Lua 函数时（请参阅 [lua\_call](http://www.lua.org/manual/5.3/manual.html#lua_call)），Lua 确保堆栈有足够的空间容纳所有结果，但它不能确保任何额外的空间。因此，在此类调用后推送堆栈中的任何内容之前，您应该使用 [lua\_checkstack](http://www.lua.org/manual/5.3/manual.html#lua_checkstack)。

4.3 – 有效和可接受的指数

API 中接收堆栈索引的任何函数仅适用于*有效*索引或*可接受的索引*。

*有效索引*是引用存储可修改 Lua 值的位置的索引。它包括介于 1 和堆栈顶部（1 ≤ abs（index） ≤ top）之间的堆栈索引以及伪索引，*伪索引*表示 C 代码可以访问但不在堆栈中的一些位置。伪索引用于访问注册表（参见 [§4.5](http://www.lua.org/manual/5.3/manual.html#4.5)）和 C 函数的上行值（参见 [§4.4](http://www.lua.org/manual/5.3/manual.html#4.4)）。

不需要特定的可变位置，而只需要一个值（例如，查询函数）的函数可以使用可接受的索引进行调用。*可接受的*索引可以是任何有效的索引，但也可以是为堆栈分配的空间内堆栈顶部之后的任何正索引，即最多到堆栈大小的索引。（请注意，0 永远不是可接受的索引。除非另有说明，否则 API 中的函数使用可接受的索引。

可接受的索引用于避免在查询堆栈时对堆栈顶部进行额外的测试。例如，C 函数可以查询其第三个参数，而无需首先检查是否存在第三个参数，也就是说，无需检查 3 是否为有效索引。

对于可以使用可接受的索引调用的函数，任何无效索引都被视为包含虚拟类型 LUA\_TNONE 的值，其行为类似于 nil 值。

4.4 – C 闭包

当创建一个C函数时，可以将一些值与它相关联，从而创建一个*C闭包*（参见[lua\_pushcclosure](http://www.lua.org/manual/5.3/manual.html#lua_pushcclosure)）;这些值称为*upvalue*，每当调用函数时都可以访问。

每当调用 C 函数时，其上加值都位于特定的伪索引处。这些伪索引由宏[lua\_upvalueindex](http://www.lua.org/manual/5.3/manual.html#lua_upvalueindex)生成。与函数关联的第一个上行值位于索引 lua\_upvalueindex（1） 处，依此类推。对 lua\_upvalueindex（*n*） 的任何访问，其中 *n* 大于当前函数的上行值数（但不大于 256，即闭包中最大上行值数的 1 加），都会生成一个可接受但无效的索引。

4.5 – 注册表

Lua 提供了一个*注册表*，一个预定义的表，任何 C 代码都可以使用它来存储它需要存储的任何 Lua 值。注册表表始终位于伪索引LUA\_REGISTRYINDEX。任何 C 库都可以将数据存储到此表中，但它必须注意选择与其他库使用的键不同的键，以避免冲突。通常，您应该使用包含库名称的字符串、代码中带有 C 对象地址的轻量级用户数据或代码创建的任何 Lua 对象作为键。与变量名称一样，以下划线开头，后跟大写字母的字符串键是为 Lua 保留的。

注册表中的整数键由引用机制（请参阅[luaL\_ref](http://www.lua.org/manual/5.3/manual.html#luaL_ref)）和一些预定义的值使用。因此，整数键不得用于其他目的。

创建新的 Lua 状态时，其注册表会附带一些预定义的值。这些预定义值使用在 lua.h 中定义为常量的整数键进行索引。定义了以下常量：

* **LUA\_RIDX\_MAINTHREAD：**在此索引中，注册表具有状态的主线程。（主线程是与状态一起创建的线程。
* **LUA\_RIDX\_GLOBALS：**在此索引中，注册表具有全局环境。

4.6 – C 语言中的错误处理

在内部，Lua 使用 C longjmp 工具来处理错误。（Lua 将使用异常，如果您将其编译为 C++;搜索 有关详细信息，LUAI\_THROW源代码中。当Lua遇到任何错误（例如内存分配错误或类型错误）时，它会*引发*错误;也就是说，它会跳远。*受保护的环境*使用 setjmp 设置恢复点;任何错误都会跳转到最新的活动恢复点。

在 C 函数中，您可以通过调用 [lua\_error](http://www.lua.org/manual/5.3/manual.html#lua_error) 引发错误。

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

如果错误发生在任何受保护环境之外，Lua 将调用 *panic 函数*（请参阅[lua\_atpanic](http://www.lua.org/manual/5.3/manual.html#lua_atpanic)），然后调用 abort，从而退出主机应用程序。您的紧急功能可以通过永不返回来避免此退出（例如，跳远到Lua以外的恢复点）。

顾名思义，恐慌功能是一种最后的手段。程序应避免它。作为一般规则，当 Lua 以 Lua 状态调用 C 函数时，它可以在该 Lua 状态上做任何它想做的事情，因为它应该已经受到保护。但是，当 C 代码对其他 Lua 状态（例如，函数的 Lua 参数、存储在注册表中的 Lua 状态或 [lua\_newthread](http://www.lua.org/manual/5.3/manual.html#lua_newthread) 的结果）进行操作时，它应该只在不会引发错误的 API 调用中使用它们。

panic 函数像消息处理程序一样运行（参见 [§2.3](http://www.lua.org/manual/5.3/manual.html#2.3)）;特别是，error 对象位于堆栈的顶部。但是，不能保证堆栈空间。要在堆栈上推送任何内容，panic 函数必须首先检查可用空间（参见 [§4.2](http://www.lua.org/manual/5.3/manual.html#4.2)）。

4.7 – 处理 C 语言中的产量

在内部，Lua 使用 C longjmp 工具来生成一个协程。因此，如果一个 C 函数 foo 调用一个 API 函数，并且这个 API 函数产生（直接或间接地通过调用另一个产生函数），Lua 不能再返回到 foo，因为 longjmp 从 C 堆栈中删除了它的帧。

为了避免此类问题，Lua 在尝试跨 API 调用让步时都会引发错误，但三个函数除外：[lua\_yieldk](http://www.lua.org/manual/5.3/manual.html#lua_yieldk)、[lua\_callk](http://www.lua.org/manual/5.3/manual.html#lua_callk) 和 [lua\_pcallk](http://www.lua.org/manual/5.3/manual.html#lua_pcallk)。所有这些函数都会接收一个*延续函数*（作为名为 k 的参数），以便在产生后继续执行。

我们需要设置一些术语来解释延续。我们有一个从Lua调用的C函数，我们将它称为*原始函数*。然后，这个原始函数调用 C API 中的这三个函数之一，我们将调用该函数，然后生成当前线程。（当被调用方函数 [lua\_yieldk](http://www.lua.org/manual/5.3/manual.html#lua_yieldk)，或者当被调用方函数lua\_callk 或[lua\_pcallk](http://www.lua.org/manual/5.3/manual.html#lua_pcallk)并且它们调用的函数生成时。

假设正在运行的线程在执行被调用方函数时产生。线程恢复后，它最终将完成调用方函数的运行。但是，被调用方函数无法返回到原始函数，因为它在 C 堆栈中的帧被 yield 破坏了。相反，Lua 调用*一个延续*函数，该函数是作为被调用方函数的参数给出的。顾名思义，延续函数应该继续原始函数的任务。

作为说明，请考虑以下函数：

int original\_function （lua\_State \*L） {

... /\* 代码 1 \*/

状态 = lua\_pcall（L， n， m， h）; /\* 呼叫 Lua \*/

... /\* 代码 2 \*/

}

现在我们想允许[lua\_pcall](http://www.lua.org/manual/5.3/manual.html#lua_pcall)运行的 Lua 代码产生。首先，我们可以像下面这样重写我们的函数：

int k （lua\_State \*L， int status， lua\_KContext ctx） {

... /\* 代码 2 \*/

}

int original\_function （lua\_State \*L） {

... /\* 代码 1 \*/

返回 k（L， lua\_pcall（L， n， m， h）， ctx）;

}

在上面的代码中，新函数 k 是一个*延续*函数（类型[为 lua\_KFunction](http://www.lua.org/manual/5.3/manual.html#lua_KFunction)），它应该完成调用 lua\_pcall 后原始函数正在做的所有工作。现在，我们必须通知 Lua，如果 [lua\_pcall](http://www.lua.org/manual/5.3/manual.html#lua_pcall) 正在执行的 Lua 代码以某种方式中断（错误或屈服），它必须调用 k，所以我们按这里重写代码，用 [lua\_pcallk](http://www.lua.org/manual/5.3/manual.html#lua_pcallk) 替换[lua\_pcall](http://www.lua.org/manual/5.3/manual.html#lua_pcall)：

int original\_function （lua\_State \*L） {

... /\* 代码 1 \*/

返回 k（L， lua\_pcallk（L， n， m， h， ctx2， k）， ctx1）;

}

请注意对延续的外部显式调用：Lua 将仅在需要时调用延续，也就是说，在发生错误或在产生后恢复的情况下。如果被调用的函数正常返回而不屈服，[lua\_pcallk](http://www.lua.org/manual/5.3/manual.html#lua_pcallk)（和[lua\_callk](http://www.lua.org/manual/5.3/manual.html#lua_callk)）也将正常返回。（当然，在这种情况下，您可以直接在原始函数中执行等效的工作，而不是调用延续。

除了 Lua 状态之外，继续函数还有另外两个参数：调用的最终状态加上最初传递给[lua\_pcallk](http://www.lua.org/manual/5.3/manual.html#lua_pcallk)的上下文值 （ctx）。（Lua 不使用此上下文值;它仅将此值从原始函数传递到延续函数。对于[lua\_pcallk](http://www.lua.org/manual/5.3/manual.html#lua_pcallk)，状态与 [lua\_pcallk](http://www.lua.org/manual/5.3/manual.html#lua_pcallk) 返回的值相同，只是在 收益后执行时[LUA\_YIELD](http://www.lua.org/manual/5.3/manual.html#pdf-LUA_YIELD)状态（而不是[LUA\_OK](http://www.lua.org/manual/5.3/manual.html#pdf-LUA_OK)）。对于[lua\_yieldk](http://www.lua.org/manual/5.3/manual.html#lua_yieldk)和[lua\_callk](http://www.lua.org/manual/5.3/manual.html#lua_callk)，当 Lua 调用延续时，状态始终LUA\_YIELD 。（对于这两个函数，Lua 在发生错误时不会调用延续，因为它们不处理错误。同样，使用 [lua\_callk](http://www.lua.org/manual/5.3/manual.html#lua_callk) 时，应调用以 [LUA\_OK](http://www.lua.org/manual/5.3/manual.html#pdf-LUA_OK) 作为状态的延续函数。（对于 [lua\_yieldk](http://www.lua.org/manual/5.3/manual.html#lua_yieldk)，直接调用延续函数没有多大意义，因为[lua\_yieldk](http://www.lua.org/manual/5.3/manual.html#lua_yieldk)通常不会返回。

Lua 将延续函数视为原始函数。继续函数从原始函数接收相同的 Lua 堆栈，其状态与被调用方函数返回时的状态相同。（例如，[lua\_callk](http://www.lua.org/manual/5.3/manual.html#lua_callk)后，函数及其参数将从堆栈中删除，并替换为调用的结果。它也具有相同的上升值。无论它返回什么，都由 Lua 处理，就好像它是原始函数的返回一样。

4.8 – 函数和类型

在这里，我们按字母顺序列出了 C API 中的所有函数和类型。每个函数都有一个这样的指标：[-o， +p， *x*]

第一个字段 o 是函数从堆栈中弹出的元素数。第二个字段 p 是函数推送到堆栈上的元素数。（任何函数总是在弹出其参数后推送其结果。x|y形式的字段表示函数可以根据情况推送（或弹出）x或y元素;询问标记“？' 意味着我们无法通过仅查看函数的参数来知道函数弹出/推送了多少元素（例如，它们可能取决于堆栈上的内容）。第三个字段 x 告诉函数是否可能引发错误：“-”表示函数永远不会引发任何错误;“m”表示函数可能会引发内存不足错误和运行\_\_gc元方法的错误;'e' 表示函数可能会引发任何错误（它可以直接或通过元方法运行任意 Lua 代码）;“v”表示函数可能会故意引发错误。

lua\_absindex

[-0, +0, –]

int lua\_absindex （lua\_State \*L， int idx）;

将可接受的索引 idx 转换为等效的绝对索引（即不依赖于堆栈顶部的索引）。

lua\_Alloc

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

空隙 \*PTR，

size\_t osize，

size\_t nsize）;

Lua 状态使用的内存分配函数的类型。分配器函数必须提供类似于 realloc 的功能，但不完全相同。它的参数是ud，一个传递给lua\_newstate的不透明指针; PTR，指向正在分配/重新分配/释放的块的指针; osize，块的原始大小或有关正在分配的内容的一些代码;和nsize，块的新大小。

当 ptr 不是 NULL 时，osize 是 ptr 指向的块的大小，即分配或重新分配时给定的大小。

当 ptr 为 NULL 时，osize 对 Lua 分配的对象类型进行编码。 osize 是 [Lua](http://www.lua.org/manual/5.3/manual.html#pdf-LUA_TSTRING) 创建该类型的新对象时（且仅当）LUA\_TSTRING、LUA\_TTABLE、LUA\_TFUNCTION、LUA\_TUSERDATA或LUA\_TTHREAD中的任何一个。当 osize 是其他值时，Lua 正在为其他内容分配内存。

Lua 假定分配器函数具有以下行为：

当 nsize 为零时，分配器的行为必须像 free 一样并 返回 NULL。

当 nsize 不为零时，分配器的行为必须像 realloc 一样。分配器返回 NULL 当且仅当它无法满足请求时。Lua 假设分配器在 osize >= nsize 时永远不会失败。

下面是分配器函数的简单实现。它由[luaL\_newstate](http://www.lua.org/manual/5.3/manual.html#luaL_newstate)在辅助库中使用。

静态空隙 \*l\_alloc （空隙 \*ud， 空隙 \*PTR， size\_t osize，

size\_t nsize） {

（空）乌德; （空）卵形; /\* 未使用 \*/

如果 （nsize == 0） {

免费（PTR）;

返回空;

}

还

返回 realloc（PTR， nsize ）;

}

请注意，标准 C 确保 free（NULL） 没有效果，并且 realloc（NULL，size） 等效于 malloc（size）。此代码假定 realloc 在收缩块时不会失败。（尽管标准 C 不能确保这种行为，但这似乎是一个安全的假设。

lua\_arith

[-（2|1）， +1， *e*]

void lua\_arith （lua\_State \*L， int op）;

对堆栈顶部的两个值（如果为否定，则为一个值）执行算术或按位运算，顶部的值是第二个操作数，弹出这些值，并推送运算结果。该函数遵循相应 Lua 运算符的语义（即，它可以调用元方法）。

op 的值必须是以下常量之一：

* **LUA\_OPADD：** 执行加法 （+）
* **LUA\_OPSUB：**执行减法 （-）
* **LUA\_OPMUL：**执行乘法 （\*）
* **LUA\_OPDIV：**执行浮点除法 （/）
* **LUA\_OPIDIV：**执行楼层划分 （//）
* **LUA\_OPMOD：**执行模 （%）
* **LUA\_OPPOW：**执行幂运算 （^）
* **LUA\_OPUNM：**执行数学否定（一元-）
* **LUA\_OPBNOT：**按位执行非 （~）
* **LUA\_OPBAND：**按位执行 AND （&）
* **LUA\_OPBOR：**执行按位 OR （|)
* **LUA\_OPBXOR：**执行按位独占 OR （~）
* **LUA\_OPSHL：**执行左移 （<<）
* **LUA\_OPSHR：**执行右移 （>>）

lua\_atpanic

[-0, +0, –]

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

设置一个新的 panic 函数并返回旧的 panic 函数（参见 [§4.6](http://www.lua.org/manual/5.3/manual.html#4.6)）。

lua\_call

[-（nargs+1）， +nresults， *e*]

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

调用函数。

要调用函数，必须使用以下协议：首先，将要调用的函数推送到堆栈上;然后，按直接顺序推送函数的参数;也就是说，第一个参数首先被推送。最后你打电话[给lua\_call](http://www.lua.org/manual/5.3/manual.html#lua_call); nargs 是你推送到堆栈上的参数数。调用函数时，所有参数和函数值都会从堆栈中弹出。函数返回时，函数结果将推送到堆栈上。结果数将调整为 n结果，除非 nresult 已LUA\_MULTRET。在这种情况下，将推送函数的所有结果;Lua 负责确保返回的值适合堆栈空间，但它不能确保堆栈中有任何额外的空间。函数结果按直接顺序推送到堆栈上（第一个结果首先推送），因此在调用后最后一个结果位于堆栈顶部。

被调用函数内的任何错误都会向上传播（使用 longjmp）。

以下示例显示了主机程序如何执行与此 Lua 代码等效的操作：

a = f（“how”， t.x， 14）

这是在 C 中：

lua\_getglobal（L， “f”）; 要调用的 /\* 函数 \*/

lua\_pushliteral（L， “如何”）; /\* 第一个参数 \*/

lua\_getglobal（L， “t”）; /\* 要编制索引的表 \*/

lua\_getfield（L， -1， “x”）; /\* T.x 的推送结果（第 2 个参数） \*/

lua\_remove（L， -2）; /\* 从堆栈中删除“t” \*/

lua\_pushinteger（L， 14）; /\* 第三个参数 \*/

lua\_call（L， 3， 1）; /\* 用 3 个参数和 1 个结果调用 'f' \*/

lua\_setglobal（L， “a”）; /\* 全局集 'a' \*/

请注意，上面的代码是*平衡的*：在其末尾，堆栈返回到其原始配置。这被认为是良好的编程实践。

lua\_callk

[-（nargs + 1）， +nresults， *e*]

空隙lua\_callk （lua\_State \*L，

国际纳尔格斯，

int nresults，

lua\_KContext ctx，

lua\_KFunction k）;

此函数的行为与[lua\_call](http://www.lua.org/manual/5.3/manual.html#lua_call)完全相同，但允许调用的函数产生（参见 [§4.7](http://www.lua.org/manual/5.3/manual.html#4.7)）。

lua\_CFunction

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

C 函数的类型。

为了与Lua正确通信，C函数必须使用以下协议，该协议定义了参数和结果的传递方式：C函数以直接顺序从其堆栈中的Lua接收其参数（首先推送第一个参数）。因此，当函数启动时，lua\_gettop（L） 返回函数接收的参数数。第一个参数（如果有）位于索引 1 处，最后一个参数位于索引 lua\_gettop（L） 处。为了将值返回给Lua，C函数只需按直接顺序将它们推送到堆栈上（首先推送第一个结果），并返回结果的数量。结果下方堆栈中的任何其他值都将被 Lua 正确丢弃。与Lua函数一样，Lua调用的C函数也可以返回许多结果。

例如，以下函数接收可变数量的数值参数，并返回其平均值和总和：

static int foo （lua\_State \*L） {

int n = lua\_gettop（L）; /\* 参数数 \*/

lua\_Number和 = 0.0;

国际 i;

for （i = 1; i <= n; i++） {

if （！lua\_isnumber（L， i）） {

lua\_pushliteral（L，“不正确的论点”）;

lua\_error（L）;

}

总和 += lua\_tonumber（L， i）;

}

lua\_pushnumber（升，总和/n）; /\* 第一个结果 \*/

lua\_pushnumber（L，总和）; /\* 第二个结果 \*/

返回 2; /\* 结果数 \*/

}

lua\_checkstack

[-0, +0, –]

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

确保堆栈至少有 n 个额外插槽的空间（也就是说，您可以安全地将最多 n 个值推送到其中）。如果它无法满足请求，则返回 false，因为它会导致堆栈大于固定的最大大小（通常至少几千个元素），或者因为它无法为额外空间分配内存。此函数从不缩小堆栈;如果堆栈已经有空间容纳额外的插槽，则保持不变。

lua\_close

[-0, +0, –]

无效lua\_close（lua\_State \*L）;

销毁给定 Lua 状态中的所有对象（调用相应的垃圾回收元方法，如果有），并释放此状态使用的所有动态内存。在一些平台中，您可能不需要调用此函数，因为所有资源在主机程序结束时自然释放。另一方面，创建多个状态的长时间运行的程序（如守护程序或 Web 服务器）可能需要在不需要状态时立即关闭状态。

lua\_compare

[-0， +0， *e*]

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

比较两个 Lua 值。如果索引索引 1 处的值与索引索引 2 处的值相比满足 op，则返回 1，遵循相应 Lua 运算符的语义（即，它可以调用元方法）。否则返回 0。如果任何索引无效，则返回 0。

op 的值必须是以下常量之一：

* **LUA\_OPEQ：**比较相等 （==）
* **LUA\_OPLT：**比较小于 （<）
* **LUA\_OPLE：**比较小于或相等 （<=）

lua\_concat

[-n， +1， *e*]

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

连接堆栈顶部的 n 个值，弹出它们，并将结果保留在顶部。如果 n 为 1，则结果是堆栈上的单个值（即，函数不执行任何操作）;如果 n 为 0，则结果为空字符串。串联是按照Lua的通常语义执行的（参见[§3.4.6](http://www.lua.org/manual/5.3/manual.html#3.4.6)）。

lua\_copy

[-0, +0, –]

void lua\_copy （lua\_State \*L， int fromidx， int toidx）;

将索引 fromidx 处的元素复制到有效的索引 toidx 中，替换该位置的值。其他位置的值不受影响。

lua\_createtable

[-0， +1， *米*]

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

创建一个新的空表并将其推送到堆栈上。参数 narr 是表将有多少个元素作为序列的提示;参数 nrec 是表 将有多少其他元素的提示。Lua 可以使用这些提示为新表预分配内存。当您事先知道表将具有多少个元素时，此预分配对于性能非常有用。否则，您可以使用函数lua\_newtable。

lua\_dump

[-0, +0, –]

int lua\_dump （lua\_State \*L，

lua\_Writer作家，

无效 \*数据，

int strip）;

将函数转储为二进制块。接收堆栈顶部的 Lua 函数并生成一个二进制块，如果再次加载，则会产生与转储的函数等效的函数。当它生成部分块时，[lua\_dump](http://www.lua.org/manual/5.3/manual.html#lua_dump)调用函数编写器（见[lua\_Writer](http://www.lua.org/manual/5.3/manual.html#lua_Writer)）与给定的数据来写入它们。

如果 strip 为 true，则二进制表示形式可能不包括有关函数的所有调试信息，以节省空间。

返回的值是上次调用编写器返回的错误代码;0 表示没有错误。

此函数不会从堆栈中弹出 Lua 函数。

lua\_error

[-1， +0， *v*]

int lua\_error （lua\_State \*L）;

生成 Lua 错误，使用堆栈顶部的值作为错误对象。此函数执行跳远，因此永远不会返回（见[luaL\_error](http://www.lua.org/manual/5.3/manual.html#luaL_error)）。

lua\_gc

[-0， +0， *米*]

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

控制垃圾回收器。

此函数根据参数的值执行多个任务：

* **LUA\_GCSTOP：**停止垃圾回收器。
* **LUA\_GCRESTART：**重新启动垃圾回收器。
* **LUA\_GCCOLLECT：**执行完整的垃圾收集周期。
* **LUA\_GCCOUNT：**返回 Lua 正在使用的当前内存量（以 KB 为单位）。
* **LUA\_GCCOUNTB：**返回 Lua 当前使用的内存字节数除以 1024 的余数。
* **LUA\_GCSTEP：**执行垃圾回收的增量步骤。
* **LUA\_GCSETPAUSE：**将数据设置为 收集器暂停 的新值（参见 [§2.5](http://www.lua.org/manual/5.3/manual.html#2.5)），并返回暂停的先前值。
* **LUA\_GCSETSTEPMUL：**将数据设置为收集器的*步进乘数*的新值 （参见 [§2.5](http://www.lua.org/manual/5.3/manual.html#2.5)），并返回步进乘数的先前值。
* **LUA\_GCISRUNNING：**返回一个布尔值，告知收集器是否正在运行（即未停止）。

有关这些选项的更多详细信息，请参阅[收集垃圾](http://www.lua.org/manual/5.3/manual.html#pdf-collectgarbage)。

lua\_getallocf

[-0, +0, –]

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

返回给定状态的内存分配函数。如果 ud 不是 NULL，Lua 在 \*ud 中存储设置内存分配器函数时给出的不透明指针。

lua\_getfield

[-0， +1， *e*]

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

将值 t[k] 推送到堆栈上，其中 t 是给定索引处的值。与 Lua 一样，此函数可能会触发“索引”事件的元方法（参见 [§2.4](http://www.lua.org/manual/5.3/manual.html#2.4)）。

返回推送值的类型。

lua\_getextraspace

[-0, +0, –]

无效 \*lua\_getextraspace （lua\_State \*L）;

返回指向与给定 Lua 状态关联的原始内存区域的指针。应用程序可以将此区域用于任何目的;Lua 不使用它做任何事情。

每个新线程都使用此线程区域的副本初始化此区域。

默认情况下，此区域的大小与指向 void 的指针相同，但您可以为此区域使用不同的大小重新编译 Lua。（见 luaconf.h 中的LUA\_EXTRASPACE）

lua\_getglobal

[-0， +1， *e*]

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

将全局名称的值推送到堆栈上。返回该值的类型。

lua\_geti

[-0， +1， *e*]

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

将值 t[i] 推送到堆栈上，其中 t 是给定索引处的值。与 Lua 一样，此函数可能会触发“索引”事件的元方法（参见 [§2.4](http://www.lua.org/manual/5.3/manual.html#2.4)）。

返回推送值的类型。

lua\_getmetatable

[-0， +（0|1）， –]

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

如果给定索引处的值具有元表，则该函数将该元表推送到堆栈上并返回 1。否则，该函数将返回 0 并在堆栈上推送任何内容。

lua\_gettable

[-1， +1， *e*]

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

将值 t[k] 推送到堆栈上，其中 t 是给定索引处的值，k 是堆栈顶部的值。

此函数从堆栈中弹出键，将结果值推到其位置。与 Lua 一样，此函数可能会触发“索引”事件的元方法（参见 [§2.4](http://www.lua.org/manual/5.3/manual.html#2.4)）。

返回推送值的类型。

lua\_gettop

[-0, +0, –]

int lua\_gettop （lua\_State \*L）;

返回堆栈中顶部元素的索引。由于索引从 1 开始，因此此结果等于堆栈中的元素数;特别是，0 表示空堆栈。

lua\_getuservalue

[-0, +1, –]

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

将与给定索引处的完整用户数据关联的 Lua 值推送到堆栈中。

返回推送值的类型。

lua\_insert

[-1, +1, –]

无效lua\_insert（lua\_State \*L，整数索引）;

将顶部元素移动到给定的有效索引中，向上移动此索引上方的元素以打开空间。此函数不能使用伪索引调用，因为伪索引不是实际的堆栈位置。

lua\_Integer

打字...lua\_Integer;

Lua 中的整数类型。

默认情况下，此类型为 long long（通常是 64 位双补码整数），但可以更改为 long 或 int（通常是 32 位双补码整数 ）。（见 LUA\_INT\_TYPE在路阿孔中。

Lua 还定义了常量 LUA\_MININTEGER 和 LUA\_MAXINTEGER，以及适合此类型的最小值和最大值。

lua\_isboolean

[-0, +0, –]

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

如果给定索引处的值是布尔值，则返回 1，否则返回 0。

lua\_iscfunction

[-0, +0, –]

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

如果给定索引处的值是 C 函数，则返回 1，否则返回 0。

lua\_isfunction

[-0, +0, –]

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

如果给定索引处的值是函数（C 或 Lua），则返回 1，否则返回 0。

lua\_isinteger

[-0, +0, –]

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

如果给定索引处的值是整数（即，该值是数字并表示为整数），则返回 1，否则返回 0。

lua\_islightuserdata

[-0, +0, –]

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

如果给定索引处的值是轻用户数据，则返回 1，否则返回 0。

lua\_isnil

[-0, +0, –]

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

如果给定索引处的值为 **nil**，则返回 1，否则返回 0。

lua\_isnone

[-0, +0, –]

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

如果给定索引无效，则返回 1，否则返回 0。

lua\_isnoneornil

[-0, +0, –]

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

如果给定索引无效或此索引处的值为 **nil**，则返回 1，否则返回 0。

lua\_isnumber

[-0, +0, –]

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

如果给定索引处的值是数字或可转换为数字的字符串，则返回 1，否则返回 0。

lua\_isstring

[-0, +0, –]

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

如果给定索引处的值是字符串或数字（始终可转换为字符串），则返回 1，否则返回 0。

lua\_istable

[-0, +0, –]

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

如果给定索引处的值是表，则返回 1，否则返回 0。

lua\_isthread

[-0, +0, –]

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

如果给定索引处的值是线程，则返回 1，否则返回 0。

lua\_isuserdata

[-0, +0, –]

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

如果给定索引处的值是用户数据（完整或浅色），则返回 1，否则返回 0。

lua\_isyieldable

[-0, +0, –]

国际lua\_isyieldable （lua\_State \*L）;

如果给定的协程可以生成，则返回 1，否则返回 0。

lua\_KContext

打字...lua\_KContext;

延续函数上下文的类型。它必须是数值类型。此类型定义为intptr\_t可用时intptr\_t，以便它也可以存储指针。否则，它被定义为ptrdiff\_t。

lua\_KFunction

typedef int （\*lua\_KFunction） （lua\_State \*L， int status， lua\_KContext ctx）;

延续函数的类型（参见 [§4.7](http://www.lua.org/manual/5.3/manual.html#4.7)）。

lua\_len

[-0， +1， *e*]

无效lua\_len（lua\_State \*L，整数指数）;

返回给定索引处值的长度。它等效于 Lua 中的 '#' 运算符（参见 §3.4.7），并且可以触发 “length” 事件的元方法（参见 [§2.4](http://www.lua.org/manual/5.3/manual.html#2.4)）。结果被推送到堆栈上。

lua\_load

[-0, +1, –]

int lua\_load （lua\_State \*L，

lua\_Reader读者，

无效 \*数据，

常量字符 \*块名，

常量字符 \*模式）;

加载 Lua 块而不运行它。如果没有错误，lua\_load将编译后的块作为 Lua 函数推送到堆栈顶部。否则，它会推送错误消息。

lua\_load的返回值为：

* [**LUA\_OK**](http://www.lua.org/manual/5.3/manual.html#pdf-LUA_OK)**：**没有错误;
* **LUA\_ERRSYNTAX：**预编译过程中的语法错误;
* [**LUA\_ERRMEM**](http://www.lua.org/manual/5.3/manual.html#pdf-LUA_ERRMEM)**：**内存分配（内存不足）错误;
* [**LUA\_ERRGCMM**](http://www.lua.org/manual/5.3/manual.html#pdf-LUA_ERRGCMM)**：**运行\_\_gc元方法时出错。（此错误与正在加载的块无关。它由垃圾回收器生成。

lua\_load函数使用用户提供的读取器函数来读取区块（请参阅[lua\_Reader](http://www.lua.org/manual/5.3/manual.html#lua_Reader)）。数据参数是传递给读取器函数的不透明值。

chunkname 参数为块提供一个名称，该名称用于错误消息和调试信息（参见 [§4.9](http://www.lua.org/manual/5.3/manual.html#4.9)）。

lua\_load 自动检测块是文本还是二进制，并相应地加载它（请参阅程序 luac）。字符串模式的工作方式与函数[加载](http://www.lua.org/manual/5.3/manual.html#pdf-load)一样，但 NULL 值等效于字符串“bt”。

lua\_load内部使用堆栈，因此 reader 函数在返回时必须始终保持堆栈不变。

如果生成的函数具有上行值，则其第一个上行值将设置为存储在注册表中索引LUA\_RIDX\_GLOBALS处的全局环境的值（参见 [§4.5](http://www.lua.org/manual/5.3/manual.html#4.5)）。加载主块时，此上值将是\_ENV变量（参见 [§2.2](http://www.lua.org/manual/5.3/manual.html#2.2)）。其他上行值初始化为 **nil**。

lua\_newstate

[-0, +0, –]

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

创建一个在新的独立状态下运行的新线程。 如果无法创建线程或状态（由于内存不足），则返回 NULL。参数 f 是分配器函数;Lua 通过此函数为此状态执行所有内存分配（请参阅[lua\_Alloc](http://www.lua.org/manual/5.3/manual.html#lua_Alloc)）。第二个参数 ud 是一个不透明的指针，Lua 在每次调用中都会传递给分配器。

lua\_newtable

[-0， +1， *米*]

无效lua\_newtable（lua\_State\*L）;

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

lua\_newthread

[-0， +1， *米*]

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

创建一个新线程，将其推送到堆栈上，并返回指向表示此新线程的lua\_State的指针。此函数返回的新线程与原始线程共享其全局环境，但具有独立的执行堆栈。

没有用于关闭或销毁线程的显式函数。线程受到垃圾回收的影响，就像任何Lua对象一样。

lua\_newuserdata

[-0， +1， *米*]

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

此函数分配一个具有给定大小的新内存块，将具有块地址的新的完整用户数据推送到堆栈上，并返回此地址。主机程序可以自由使用此内存。

lua\_next

[-1， +（2|0） *和*]

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

从堆栈中弹出一个键，并从表中的给定索引（给定键之后的“下一个”对）推送键值对。如果表中没有更多元素，则[lua\_next](http://www.lua.org/manual/5.3/manual.html#lua_next)返回 0（并且不推送任何内容）。

典型的遍历如下所示：

/\* 表位于索引 't' 的堆栈中 \*/

lua\_pushnil（L）; /\* 第一个键 \*/

而 （lua\_next（L， t） ！= 0） {

/\* 使用“键”（在索引 -2 处）和“值”（在索引 -1 处）\*/

printf（“%s - %s\n”，

lua\_typename（L， lua\_type（L， -2）），

lua\_typename（L， lua\_type（L， -1）））;

/\* 删除“值”;为下一次迭代保留“键” \*/

lua\_pop（L， 1）;

}

遍历表时，不要直接在键上调用lua\_tolstring，除非您知道键实际上是一个字符串。回想一下，[lua\_tolstring](http://www.lua.org/manual/5.3/manual.html#lua_tolstring)可能会更改给定索引的值;这会混淆下一次对lua\_next的调用。

有关在遍历表期间修改表的注意事项，请参阅[下面的](http://www.lua.org/manual/5.3/manual.html#pdf-next)函数。

lua\_Number

打字...lua\_Number;

Lua中的浮标类型。

默认情况下，此类型为双精度型，但可以更改为单浮点数或长双精度型。（见 luaconf.h 中的LUA\_FLOAT\_TYPE）

lua\_numbertointeger

int lua\_numbertointeger （lua\_Number n， lua\_Integer \*p）;

将 Lua 浮点数转换为 Lua 整数。此宏假定 n 具有整数值。如果该值在 Lua 整数范围内，则会将其转换为整数并分配给 \*p。宏将生成一个布尔值，指示转换是否成功。（请注意，由于四舍五入，如果没有此宏，此范围测试可能很难正确执行。

此宏可以多次计算其参数。

lua\_pcall

[-（nargs + 1）， +（nresults|1）， –]

你是lua\_pcall（lua\_State\*L，你这些纳格斯，你是nresults，你是msgh）;

在保护模式下调用函数。

nargs 和 nresult 的含义与 [lua\_call](http://www.lua.org/manual/5.3/manual.html#lua_call) 中的含义相同。如果在调用过程中没有错误，则[lua\_pcall](http://www.lua.org/manual/5.3/manual.html#lua_pcall)的行为与[lua\_call](http://www.lua.org/manual/5.3/manual.html#lua_call)完全相同。但是，如果有任何错误，lua\_pcall 会捕获它，在堆栈（错误对象）上推送单个值，并返回错误代码。与[lua\_call](http://www.lua.org/manual/5.3/manual.html#lua_call)一样，[lua\_pcall](http://www.lua.org/manual/5.3/manual.html#lua_pcall)总是从堆栈中删除函数及其参数。

如果 msgh 为 0，则堆栈上返回的错误对象正是原始错误对象。否则，msgh 是*消息处理程序*的堆栈索引。（此索引不能是伪索引。如果发生运行时错误，将使用错误对象调用此函数，其返回值将是 lua\_pcall 在堆栈上返回的对象。

通常，消息处理程序用于向错误对象添加更多调试信息，例如堆栈回溯。[lua\_pcall](http://www.lua.org/manual/5.3/manual.html#lua_pcall)返回后无法收集此类信息，因为到那时堆栈已经展开。

[lua\_pcall](http://www.lua.org/manual/5.3/manual.html#lua_pcall)函数返回以下常量之一（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。对于此类错误，Lua 不会调用消息处理程序。
* **LUA\_ERRERR：**运行消息处理程序时出错。
* **LUA\_ERRGCMM：**运行\_\_gc元方法时出错。对于此类错误，Lua 不会调用消息处理程序（因为此类错误通常与被调用的函数无关）。

lua\_pcallk

[-（nargs + 1）， +（nresults|1）， –]

int lua\_pcallk （lua\_State \*L，

国际纳尔格斯，

int nresults，

国际味精，

lua\_KContext ctx，

lua\_KFunction k）;

此函数的行为与[lua\_pcall](http://www.lua.org/manual/5.3/manual.html#lua_pcall)完全相同，但允许被调用的函数产生（参见 [§4.7](http://www.lua.org/manual/5.3/manual.html#4.7)）。

lua\_pop

[-n， +0， –]

无效lua\_pop （lua\_State \*L， int n）;

从堆栈中弹出 n 个元素。

lua\_pushboolean

[-0, +1, –]

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

将值为 b 的布尔值推送到堆栈上。

lua\_pushcclosure

[-n， +1， *m*]

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

将新的 C 闭包推送到堆栈上。

当创建一个C函数时，可以将一些值与它相关联，从而创建一个C闭包（参见[§4.4](http://www.lua.org/manual/5.3/manual.html#4.4)）;然后，每当调用函数时，都可以访问这些值。要将值与 C 函数相关联，首先必须将这些值推送到堆栈上（当有多个值时，首先推送第一个值）。然后调用 [lua\_pushcclosure](http://www.lua.org/manual/5.3/manual.html#lua_pushcclosure) 来创建 C 函数并将其推送到堆栈上，参数 n 告诉有多少值将与该函数关联。 [lua\_pushcclosure](http://www.lua.org/manual/5.3/manual.html#lua_pushcclosure)还会从堆栈中弹出这些值。

n 的最大值为 255。

当 n 为零时，此函数创建一个轻 C 函数，该函数只是一个指向 C 函数的指针。在这种情况下，它永远不会引发内存错误。

lua\_pushcfunction

[-0, +1, –]

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

将 C 函数推送到堆栈上。此函数接收指向 C 函数的指针，并将函数类型的 Lua 值推送到堆栈上，该函数在调用时调用相应的 C 函数。

Lua 可调用的任何函数都必须遵循正确的协议来接收其参数并返回其结果（参见 [lua\_CFunction](http://www.lua.org/manual/5.3/manual.html#lua_CFunction)）。

lua\_pushfstring

[-0， +1， *e*]

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

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

* 您不必为结果分配空间：结果是一个 Lua 字符串，Lua 负责内存分配（以及通过垃圾回收进行释放）。
* 转换说明符非常受限制。没有标志、宽度或精度。转换说明符只能是“%%”（插入字符“%”）、“%s”（插入以零结尾的字符串，没有大小限制）、“%f”（插入[lua\_Number](http://www.lua.org/manual/5.3/manual.html#lua_Number)）、“%I”（插入[lua\_Integer](http://www.lua.org/manual/5.3/manual.html#lua_Integer)）、“%p”（插入指针作为十六进制数字）、“%d”（插入整数）、“%c'（插入一个整数作为单字节字符）和“%U”（插入一个长整型作为 UTF-8 字节序列）。

与其他推送函数不同，此函数检查所需的堆栈空间，包括其结果的插槽。

lua\_pushglobaltable

[-0, +1, –]

无效lua\_pushglobaltable（lua\_State\*L）;

将全局环境推送到堆栈上。

lua\_pushinteger

[-0, +1, –]

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

将值为 n 的整数推送到堆栈上。

lua\_pushlightuserdata

[-0, +1, –]

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

将轻量级用户数据推送到堆栈上。

用户数据表示 Lua 中的 C 值。*浅用户数据*表示指针，空白\*。它是一个值（如数字）：你不创建它，它没有单独的元表，它不会被收集（因为它从未被创建过）。轻用户数据等于具有相同 C 地址的“任何”轻用户数据。

lua\_pushliteral

[-0， +1， *米*]

常量字符 \*lua\_pushliteral （lua\_State \*L， 常量字符 \*s）;

此宏等效于 [lua\_pushstring](http://www.lua.org/manual/5.3/manual.html#lua_pushstring)，但应仅在 s 是文本字符串时使用。

lua\_pushlstring

[-0， +1， *米*]

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

将 s 指向的字符串（大小为 len）推到堆栈上。Lua 创建（或重用）给定字符串的内部副本，因此 s 处的内存 可以在函数返回后立即释放或重用。字符串可以包含任何二进制数据，包括嵌入的零。

返回指向字符串的内部副本的指针。

lua\_pushnil

[-0, +1, –]

无效lua\_pushnil（lua\_State\*L）;

将 nil 值推送到堆栈上。

lua\_pushnumber

[-0, +1, –]

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

将值为 n 的浮点数推送到堆栈上。

lua\_pushstring

[-0， +1， *米*]

常量字符 \*lua\_pushstring （lua\_State \*L， 常量字符 \*s）;

将 s 指向的以零结尾的字符串推送到堆栈上。Lua 创建（或重用）给定字符串的内部副本，因此 s 处的内存 可以在函数返回后立即释放或重用。

返回指向字符串的内部副本的指针。

如果 s 为 NULL，则推送 **nil** 并返回 NULL。

lua\_pushthread

[-0, +1, –]

国际lua\_pushthread （lua\_State \*L）;

将 L 表示的线程推到堆栈上。如果此线程是其状态的主线程，则返回 1。

lua\_pushvalue

[-0, +1, –]

无效lua\_pushvalue（lua\_State \*L，整数指数）;

将给定索引处的元素副本推送到堆栈上。

lua\_pushvfstring

[-0， +1， *米*]

常量字符 \*lua\_pushvfstring （lua\_State \*L，

常量字符 \*FMT，

va\_list argp）;

等效于 [lua\_pushfstring](http://www.lua.org/manual/5.3/manual.html#lua_pushfstring)，不同之处在于它接收一个va\_list而不是可变数量的参数。

lua\_rawequal

[-0, +0, –]

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

如果索引 index1 和 index2 中的两个值原始相等（即，不调用 \_\_eq元方法），则返回 1。否则返回 0。如果任何索引无效，则返回 0。

lua\_rawget

[-1, +1, –]

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

与[lua\_gettable](http://www.lua.org/manual/5.3/manual.html#lua_gettable)类似，但执行原始访问（即，没有元方法）。

lua\_rawgeti

[-0, +1, –]

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

将值 t[n] 推送到堆栈上，其中 t 是给定索引处的表。访问是原始的，也就是说，它不调用\_\_index元方法。

返回推送值的类型。

lua\_rawgetp

[-0, +1, –]

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

将值 t[k] 推送到堆栈上，其中 t 是给定索引处的表，k 是表示 为轻用户数据的指针 p。访问权限是原始的;也就是说，它不调用\_\_index元方法。

返回推送值的类型。

lua\_rawlen

[-0, +0, –]

size\_t lua\_rawlen（lua\_State \*L，整数指数）;

返回给定索引处值的原始“长度”：对于字符串，这是字符串长度;对于表，这是没有元方法的长度运算符 （'#'） 的结果;对于 UserData，这是为 UserData 分配的内存块的大小;对于其他值，它是 0。

lua\_rawset

[-2， +0， *米*]

无效lua\_rawset（lua\_State \*L，整数指数）;

与[lua\_settable](http://www.lua.org/manual/5.3/manual.html#lua_settable)类似，但执行原始赋值（即没有元方法）。

lua\_rawseti

[-1， +0， *米*]

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

是否等效于 t[i] = v，其中 t 是给定索引处的表，v 是堆栈顶部的值。

此函数从堆栈中弹出值。赋值是原始的，也就是说，它不调用\_\_newindex元方法。

lua\_rawsetp

[-1， +0， *米*]

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

是否等效于 t[p] = v，其中 t 是给定索引处的表，p 编码为轻用户数据，v 是堆栈顶部的值。

此函数从堆栈中弹出值。赋值是原始的，也就是说，它不会调用\_\_newindex元方法。

lua\_Reader

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

无效 \*数据，

size\_t\*尺寸）;

[lua\_load](http://www.lua.org/manual/5.3/manual.html#lua_load)使用的读取器函数。每当它需要块的另一部分时，[lua\_load](http://www.lua.org/manual/5.3/manual.html#lua_load)调用读取器，传递其数据参数。读取器必须返回指向内存块的指针，其中包含新的块，并将大小设置为块大小。在再次调用读取器函数之前，该块必须存在。若要发出块结束的信号，读取器必须返回 NULL 或将大小设置为零。读取器函数可以返回大于零的任何大小的片段。

lua\_register

[-0， +0， *e*]

void lua\_register （lua\_State \*L， const char \*name， lua\_CFunction f）;

将 C 函数 f 设置为全局名称的新值。它被定义为一个宏：

#define lua\_register（L，n，f） \

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

lua\_remove

[-1, +0, –]

无效lua\_remove（lua\_State \*L，整数指数）;

删除给定有效索引处的元素，向下移动此索引上方的元素以填充空白。此函数不能使用伪索引调用，因为伪索引不是实际的堆栈位置。

lua\_replace

[-1, +0, –]

无效lua\_replace（lua\_State\*L，整数指数）;

将顶部元素移动到给定的有效索引中而不移动任何元素（因此替换该给定索引处的值），然后弹出顶部元素。

lua\_resume

[-?, +?, –]

int lua\_resume （lua\_State \*L， lua\_State \*from， int nargs）;

在给定线程 L 中启动和恢复协程。

要启动协程，请将主函数和任何参数推送到线程堆栈上;然后你调用[lua\_resume](http://www.lua.org/manual/5.3/manual.html#lua_resume)，nargs 是参数的数量。当协程挂起或完成其执行时，此调用将返回。当它返回时，堆栈包含传递给lua\_yield的所有值，或 body 函数返回的所有值。 [如果](http://www.lua.org/manual/5.3/manual.html#lua_resume)协程生成，[LUA\_OK](http://www.lua.org/manual/5.3/manual.html#pdf-LUA_OK)如果协程完成执行而没有错误，则返回 lua\_resume，如果出现错误，则返回 [LUA\_YIELD](http://www.lua.org/manual/5.3/manual.html#pdf-LUA_YIELD) 个错误代码（请参见 [lua\_pcall](http://www.lua.org/manual/5.3/manual.html#lua_pcall)）。

如果出现错误，堆栈不会展开，因此您可以在堆栈上使用调试 API。错误对象位于堆栈的顶部。

要恢复协程，请从最后一个[lua\_yield](http://www.lua.org/manual/5.3/manual.html#lua_yield)中删除所有结果，仅将作为 yield 结果传递的值放在其堆栈中，然后调用 [lua\_resume](http://www.lua.org/manual/5.3/manual.html#lua_resume)。

来自的参数表示正在恢复 L 的协程。如果没有这样的协程，则此参数可以为 NULL。

lua\_rotate

[-0, +0, –]

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

在有效索引 idx 和堆栈顶部之间旋转堆栈元素。元素沿 顶部方向旋转 n 个位置（对于正 n）或 -n 个位置沿底部方向旋转（对于负 n）。n 的绝对值不得大于要旋转的切片的大小。此函数不能使用伪索引调用，因为伪索引不是实际的堆栈位置。

lua\_setallocf

[-0, +0, –]

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

使用用户数据 ud 将给定状态的分配器函数更改为 f。

lua\_setfield

[-1， +0， *e*]

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

是否等效于 t[k] = v，其中 t 是给定索引处的值，v 是堆栈顶部的值。

此函数从堆栈中弹出值。与 Lua 一样，此函数可能会触发 “newindex” 事件的元方法（参见 [§2.4](http://www.lua.org/manual/5.3/manual.html#2.4)）。

lua\_setglobal

[-1， +0， *e*]

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

从堆栈中弹出一个值，并将其设置为全局名称的新值。

lua\_seti

[-1， +0， *e*]

无效lua\_seti（lua\_State \*L， int index， lua\_Integer n）;

等效于 t[n] = v，其中 t 是给定索引处的值，v 是堆栈顶部的值。

此函数从堆栈中弹出值。与 Lua 一样，此函数可能会触发 “newindex” 事件的元方法（参见 [§2.4](http://www.lua.org/manual/5.3/manual.html#2.4)）。

lua\_setmetatable

[-1, +0, –]

无效lua\_setmetatable（lua\_State \*L，整数索引）;

从堆栈中弹出一个表，并将其设置为给定索引处值的新元表。

lua\_settable

[-2， +0， *e*]

无效lua\_settable（lua\_State \*L，整数索引）;

是否等效于 t[k] = v，其中 t 是给定索引处的值，v 是堆栈顶部的值，k 是顶部正下方的值。

此函数从堆栈中弹出键和值。与 Lua 一样，此函数可能会触发 “newindex” 事件的元方法（参见 [§2.4](http://www.lua.org/manual/5.3/manual.html#2.4)）。

lua\_settop

[-?, +?, –]

无效lua\_settop（lua\_State \*L，整数索引）;

接受任何索引或 0，并将堆栈顶部设置为此索引。如果新顶部大于旧顶部，则新元素将填充 **nil**。如果索引为 0，则删除所有堆栈元素。

lua\_setuservalue

[-1, +0, –]

无效lua\_setuservalue（lua\_State \*L，整数指数）;

从堆栈中弹出一个值，并将其设置为与给定索引处的完整用户数据关联的新值。

lua\_State

typedef struct lua\_State lua\_State;

一个不透明的结构，指向线程并间接（通过线程）指向 Lua 解释器的整个状态。Lua 库是完全可重入的：它没有全局变量。有关状态的所有信息都可以通过此结构访问。

指向此结构的指针必须作为第一个参数传递给库中的每个函数，但 [lua\_newstate](http://www.lua.org/manual/5.3/manual.html#lua_newstate) 除外，从头开始创建 Lua 状态。

lua\_status

[-0, +0, –]

int lua\_status （lua\_State \*L）;

返回线程 L 的状态。

对于普通线程，状态可以是 0 （[LUA\_OK](http://www.lua.org/manual/5.3/manual.html#pdf-LUA_OK)），如果线程完成lua\_resume执行但出现错误，则状态可以是错误代码 ，如果线程挂起，则状态可以是 LUA\_YIELD。

只能在状态[LUA\_OK](http://www.lua.org/manual/5.3/manual.html#pdf-LUA_OK)的线程中调用函数。您可以恢复状态为 [LUA\_OK](http://www.lua.org/manual/5.3/manual.html#pdf-LUA_OK)（启动新的协程）或[LUA\_YIELD](http://www.lua.org/manual/5.3/manual.html#pdf-LUA_YIELD)（恢复协程）的线程。

lua\_stringtonumber

[-0, +1, –]

size\_t lua\_stringtonumber （lua\_State \*L， const char \*s）;

将以零结尾的字符串 s 转换为数字，将该数字推入堆栈，并返回字符串的总大小，即其长度加 1。根据Lua的词法约定，转换可以产生整数或浮点数（参见[§3.1](http://www.lua.org/manual/5.3/manual.html#3.1)）。字符串可以具有前导和尾随空格以及符号。如果字符串不是有效的数字，则返回 0 并且不推送任何内容。（请注意，如果转换成功，则结果可用作布尔值。

lua\_toboolean

[-0, +0, –]

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

将给定索引处的 Lua 值转换为 C 布尔值（0 或 1）。与Lua中的所有测试一样，[lua\_toboolean](http://www.lua.org/manual/5.3/manual.html#lua_toboolean)对于任何不同于**false**和**nil**的Lua值返回true;否则返回false。（如果只想接受实际的布尔值，请使用 [lua\_isboolean](http://www.lua.org/manual/5.3/manual.html#lua_isboolean)测试值的类型。

lua\_tocfunction

[-0, +0, –]

lua\_CFunction lua\_tocfunction（lua\_State\*L，整数索引）;

将给定索引处的值转换为 C 函数。该值必须是 C 函数;否则，返回 NULL。

lua\_tointeger

[-0, +0, –]

lua\_Integer lua\_tointeger（lua\_State \*L，整数索引）;

等效于[lua\_tointegerx](http://www.lua.org/manual/5.3/manual.html#lua_tointegerx)，则 isnum 等于 NULL。

lua\_tointegerx

[-0, +0, –]

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

将给定索引处的 Lua 值转换为有符号整数类型 [lua\_Integer](http://www.lua.org/manual/5.3/manual.html#lua_Integer)。Lua 值必须是整数，或者是可转换为整数的数字或字符串（请参阅 [§3.4.3](http://www.lua.org/manual/5.3/manual.html#3.4.3)）;否则，lua\_tointegerx返回 0。

如果 isnum 不是 NULL，则为其引用对象分配一个布尔值，该值指示操作是否成功。

lua\_tolstring

[-0， +0， *米*]

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

将给定索引处的 Lua 值转换为 C 字符串。如果 len 不是 NULL，它将 \*len 设置为字符串长度。Lua 值必须是字符串或数字;否则，该函数返回 NULL。如果值是数字，则lua\_tolstring还会*将堆栈中的实际值更改为字符串*。（此更改使人感到困惑 [lua\_next](http://www.lua.org/manual/5.3/manual.html#lua_next)在表遍历期间将lua\_tolstring应用于键时。

lua\_tolstring 返回指向 Lua 状态内的字符串的指针。此字符串在其最后一个字符（如 C 中）后始终有一个零 （'\0'），但在其正文中可以包含其他零。

由于 Lua 具有垃圾回收功能，因此无法保证从堆栈中删除相应的 Lua 值后，lua\_tolstring返回的指针将有效。

lua\_tonumber

[-0, +0, –]

lua\_Number lua\_tonumber（lua\_State \*L，整数指数）;

等效于[lua\_tonumberx](http://www.lua.org/manual/5.3/manual.html#lua_tonumberx)，其中 isnum 等于 NULL。

lua\_tonumberx

[-0, +0, –]

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

将给定索引处的 Lua 值转换为 C 类型[lua\_Number](http://www.lua.org/manual/5.3/manual.html#lua_Number)（请参阅[lua\_Number](http://www.lua.org/manual/5.3/manual.html#lua_Number)）。Lua 值必须是可转换为数字的数字或字符串（参见 [§3.4.3](http://www.lua.org/manual/5.3/manual.html#3.4.3)）;否则，[lua\_tonumberx](http://www.lua.org/manual/5.3/manual.html#lua_tonumberx)返回 0。

如果 isnum 不是 NULL，则为其引用对象分配一个布尔值，该值指示操作是否成功。

lua\_topointer

[-0, +0, –]

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

将给定索引处的值转换为泛型 C 指针 （void\*）。该值可以是用户数据、表、线程或函数;否则，lua\_topointer返回 NULL。不同的对象会给出不同的指针。无法将指针转换回其原始值。

通常，此函数仅用于哈希和调试信息。

lua\_tostring

[-0， +0， *米*]

常量字符 \*lua\_tostring （lua\_State \*L， int index）;

等效于 [lua\_tolstring](http://www.lua.org/manual/5.3/manual.html#lua_tolstring)，len 等于 NULL。

lua\_tothread

[-0, +0, –]

lua\_State \*lua\_tothread （lua\_State \*L， 整数指数）;

将给定索引处的值转换为 Lua 线程（表示为 lua\_State\*）。此值必须是线程;否则，该函数返回 NULL。

lua\_touserdata

[-0, +0, –]

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

如果给定索引处的值是完整的用户数据，则返回其块地址。如果值是轻用户数据，则返回其指针。否则，返回 NULL。

lua\_type

[-0, +0, –]

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

返回给定有效索引中的值类型，或返回无效（但可接受的）索引的LUA\_TNONE。lua\_type返回的类型由 lua.h 中定义的以下常量编码：LUA\_TNIL （0）、LUA\_TNUMBER、LUA\_TBOOLEAN、LUA\_TSTRING、LUA\_TTABLE、LUA\_TFUNCTION、LUA\_TUSERDATA  、LUA\_TTHREAD 和 LUA\_TLIGHTUSERDATA。

lua\_typename

[-0, +0, –]

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

返回由值 tp 编码的类型的名称，该值必须是 [lua\_type](http://www.lua.org/manual/5.3/manual.html#lua_type) 返回的值之一。

lua\_Unsigned

打字...lua\_Unsigned;

[lua\_Integer](http://www.lua.org/manual/5.3/manual.html#lua_Integer)的未签名版本。

lua\_upvalueindex

[-0, +0, –]

国际lua\_upvalueindex（国际一）;

返回表示正在运行的函数的第 i 个上行值的伪索引（参见 [§4.4](http://www.lua.org/manual/5.3/manual.html#4.4)）。

lua\_version

[-0, +0, –]

常量 lua\_Number \*lua\_version （lua\_State \*L）;

返回存储在 Lua 核心中的版本号（C 静态变量）的地址。使用有效[lua\_State](http://www.lua.org/manual/5.3/manual.html#lua_State)调用时，返回用于创建该状态的版本地址。使用 NULL 调用时，返回运行调用的版本地址。

lua\_Writer

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

常量无效\* p，

size\_t不，

无效\* UD）;

[lua\_dump](http://www.lua.org/manual/5.3/manual.html#lua_dump)使用的编写器函数的类型。每次生成另一段块时，[lua\_dump](http://www.lua.org/manual/5.3/manual.html#lua_dump)调用编写器，传递要写入的缓冲区 （p）、其大小 （sz） 以及 提供给[lua\_dump](http://www.lua.org/manual/5.3/manual.html#lua_dump)的数据参数。

编写器返回错误代码：0 表示没有错误;任何其他值都表示错误，并阻止[lua\_dump](http://www.lua.org/manual/5.3/manual.html#lua_dump)再次调用编写器。

lua\_xmove

[-?, +?, –]

无效lua\_xmove（lua\_State \*从、lua\_State \*到、int n）;

在具有相同状态的不同线程之间交换值。

此函数从堆栈中弹出 n 个值，并将它们推送到堆栈上。

lua\_yield

[-?, +？，*和*]

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

此函数等效于 [lua\_yieldk](http://www.lua.org/manual/5.3/manual.html#lua_yieldk)，但它没有延续（参见 [§4.7](http://www.lua.org/manual/5.3/manual.html#4.7)）。因此，当线程恢复时，它将继续调用调用函数的函数lua\_yield。

lua\_yieldk

[-?, +？，*和*]

int lua\_yieldk （lua\_State \*L，

int nresults，

lua\_KContext ctx，

lua\_KFunction k）;

生成协程（线程）。

当 C 函数调用 [lua\_yieldk](http://www.lua.org/manual/5.3/manual.html#lua_yieldk) 时，正在运行的协程将暂停其执行，并且将返回对启动此协程lua\_resume的调用。参数 nresults 是堆栈中将作为结果传递给[lua\_resume](http://www.lua.org/manual/5.3/manual.html#lua_resume)的值数。

当协程再次恢复时，Lua 调用给定的延续函数 k 以继续执行生成的 C 函数（参见 [§4.7](http://www.lua.org/manual/5.3/manual.html#4.7)）。此继续函数接收来自前一个函数的相同堆栈，其中 n 个结果被删除并替换为传递给 [lua\_resume 的](http://www.lua.org/manual/5.3/manual.html#lua_resume)参数。此外，延续函数接收 传递给[lua\_yieldk](http://www.lua.org/manual/5.3/manual.html#lua_yieldk)的值 ctx。

通常，此函数不返回;当协程最终恢复时，它将继续执行继续函数。但是，有一种特殊情况，即从行或计数钩子内部调用此函数（参见 [§4.9](http://www.lua.org/manual/5.3/manual.html#4.9)）。在这种情况下，应该调用lua\_yieldk而不进行延续（可能以 lua\_yield 的形式）并且没有结果，并且钩子应在调用后立即返回。Lua 将生成，当协程再次恢复时，它将继续正常执行触发钩子的 （Lua） 函数。

如果从具有没有继续函数的挂起 C 调用的线程调用此函数，或者从未在恢复中运行的线程（例如，主线程）调用此函数，则此函数可能会引发错误。

4.9 – 调试接口

Lua 没有内置的调试工具。相反，它通过函数和*钩子*提供了一个特殊的接口。此接口允许构造不同类型的调试器、探查器和其他需要解释器“内部信息”的工具。

lua\_Debug

typedef struct lua\_Debug {

国际事件;

常量字符 \*名称; /\* （n） \*/

常量字符 \*名称什么; /\* （n） \*/

常量字符 \*什么; /\* （S） \*/

常量字符 \*来源; /\* （S） \*/

整数电流线; /\* （l） \*/

int linedefined; /\* （S） \*/

int lastlinedefined; /\* （S） \*/

未签名的炭核; /\* （u） 上值数 \*/

未签名的字符 nparams; /\* （u） 参数数量 \*/

查尔·伊斯瓦拉格; /\* （u） \*/

夏亚·伊斯泰尔呼叫; /\* （t） \*/

夏亚short\_src[LUA\_IDSIZE];/\* （S） \*/

/\* 私人部分 \*/

*其他领域*

} lua\_Debug;

一种结构，用于携带有关函数或激活记录的不同信息。[lua\_getstack](http://www.lua.org/manual/5.3/manual.html#lua_getstack)仅填充此结构的私有部分，供以后使用。要用有用的信息填充lua\_Debug的其他字段 ，请拨打[lua\_getinfo](http://www.lua.org/manual/5.3/manual.html#lua_getinfo)。

[lua\_Debug](http://www.lua.org/manual/5.3/manual.html#lua_Debug)字段具有以下含义：

* **源：**创建函数的区块的名称。如果 source 以“@”开头，则表示该函数是在文件名跟在“@”后面的文件中定义的。如果源以“=”开头，则其其余内容以用户依赖的方式描述源。否则，该函数是在字符串中定义的，其中 source 是该字符串。
* **short\_src：**源的“可打印”版本，用于错误消息。
* **行定义：**函数定义的开始行号。
* **最后定义：**函数定义结束的行号。
* **what：**如果函数是Lua函数，则为字符串“Lua”，如果它是C函数，则为“C”，如果它是块的主要部分，则为“main”。
* **当前行：**执行给定函数的当前行。当没有可用的行信息时，当前行设置为 -1。
* **名称：**给定函数的合理名称。由于 Lua 中的函数是一等值，因此它们没有固定的名称：某些函数可以是多个全局变量的值，而其他函数只能存储在表字段中。lua\_getinfo函数检查如何调用函数以查找合适的名称。如果找不到名称，则名称 设置为 NULL。
* **名称内容：**解释名称字段。nameWhat 的值 可以是 “global”、“local”、“method”、“ field”、“upvalue” 或 “”（空字符串），具体取决于函数的调用方式。（Lua 在没有其他选项适用时使用空字符串。
* **istailcall：**如果此函数调用由尾部调用调用，则为 true。在这种情况下，此级别的调用方不在堆栈中。
* **NUPS：**函数的上值数。
* **nparams：**函数的固定参数数（对于 C 函数，始终为 0）。
* **isvararg：**如果函数是 vararg 函数，则为 true（对于 C 函数始终为 true）。

lua\_gethook

[-0, +0, –]

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

返回当前挂钩函数。

lua\_gethookcount

[-0, +0, –]

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

返回当前挂钩计数。

lua\_gethookmask

[-0, +0, –]

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

返回当前挂钩掩码。

lua\_getinfo

[-（0|1）、+（0|1|2） *和*]

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

获取有关特定函数或函数调用的信息。

要获取有关函数调用的信息，参数 ar 必须是有效的激活记录，该记录由先前对 lua\_getstack 的调用填充 或作为钩子的参数给出（请参阅[lua\_Hook](http://www.lua.org/manual/5.3/manual.html#lua_Hook)）。

要获取有关函数的信息，请将其推送到堆栈上，并以字符“>”开头 what 字符串。（在这种情况下， lua\_getinfo从堆栈顶部弹出函数。例如，要知道函数 f 是在哪一行中 定义的，您可以编写以下代码：

lua\_Debug会的;

lua\_getglobal（L， “f”）; /\* 获取全局“f” \*/

lua\_getinfo（L， “>S”， and ar）;

printf（“%d\n”， ar.linedefined）;

字符串中的每个字符 what 选择要填充的结构的某些字段 或要在堆栈上推送的值：

* **'n'：**填写字段名称，名称是什么;
* **“S”：**填写字段源、short\_src、行定义、最后一行定义和内容;
* **“l”：**填写字段电流线;
* **“t”：**填写字段 istailcall;
* **“u”：**填写字段 NUPS、nparams 和 isvararg;
* **'f'：**将给定级别运行的函数推送到堆栈上;
* **'L'：**将一个表推送到堆栈上，该表的索引是函数上有效的行数。（一 *有效*行是具有某些关联代码的行，即可以在其中放置断点的行。无效行包括空行和注释。

如果此选项与选项“f”一起给出，则在函数之后推送其表。

此函数在出错时返回 0（例如，what 中的无效选项）。

lua\_getlocal

[-0， +（0|1）， –]

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

获取有关给定激活记录或给定函数的局部变量的信息。

在第一种情况下，参数 ar 必须是有效的激活记录，该记录由先前对 lua\_getstack 的调用填充 或作为钩子的参数给出（请参阅[lua\_Hook](http://www.lua.org/manual/5.3/manual.html#lua_Hook)）。索引 n 选择要检查的局部变量;有关变量 索引和名称的详细信息，请参阅 [debug.getlocal](http://www.lua.org/manual/5.3/manual.html#pdf-debug.getlocal)。

[lua\_getlocal将](http://www.lua.org/manual/5.3/manual.html#lua_getlocal)变量的值推送到堆栈上并返回其名称。

在第二种情况下，ar 必须为 NULL，并且要检查的函数必须位于堆栈的顶部。在这种情况下，只有Lua函数的参数是可见的（因为没有关于哪些变量处于活动状态的信息），并且没有值被推送到堆栈上。

当索引大于活动局部变量的数量时返回 NULL（并且不推送任何内容）。

lua\_getstack

[-0, +0, –]

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

获取有关解释器运行时堆栈的信息。

此函数填充[lua\_Debug](http://www.lua.org/manual/5.3/manual.html#lua_Debug)结构的各个部分，标识在给定级别执行的函数的激活记录。级别 0 是当前正在运行的函数，而级别 *n+1* 是调用级别 n 的函数 （尾部调用除外，它不计算堆栈）。当没有错误时，[lua\_getstack](http://www.lua.org/manual/5.3/manual.html#lua_getstack)返回 1;当调用级别大于堆栈深度时，它返回 0。

lua\_getupvalue

[-0， +（0|1）， –]

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

获取有关索引函数索引处收盘的第 n 个上值的信息。它将上值的值推送到堆栈上并返回其名称。当索引 n 大于上行值的数量时返回 NULL（并且不推送任何内容）。

对于 C 函数，此函数使用空字符串 “” 作为所有上行值的名称。（对于 Lua 函数，upvalue 是函数使用的外部局部变量，因此包含在闭包中。

上值没有特定的顺序，因为它们在整个函数中处于活动状态。它们以任意顺序编号。

lua\_Hook

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

用于调试挂钩函数的类型。

每当调用钩子时，其 ar 参数都会将其字段事件设置为触发钩子的特定事件。Lua 使用以下常量标识这些事件：LUA\_HOOKCALL、LUA\_HOOKRET、LUA\_HOOKTAILCALL、LUA\_HOOKLINE 和LUA\_HOOKCOUNT。此外，对于线路事件，还设置了场电流线。要获取 ar 中任何其他字段的值，钩子必须调用 [lua\_getinfo](http://www.lua.org/manual/5.3/manual.html#lua_getinfo)。

对于呼叫事件，事件可以是LUA\_HOOKCALL（正常值）或LUA\_HOOKTAILCALL，对于尾部调用;在这种情况下，将没有相应的返回事件。

当 Lua 运行钩子时，它会禁用对钩子的其他调用。因此，如果钩子回调 Lua 来执行函数或块，则此执行无需调用钩子。

钩子函数不能有延续，也就是说，它们不能用非空 k 调用[lua\_yieldk](http://www.lua.org/manual/5.3/manual.html#lua_yieldk)、[lua\_pcallk](http://www.lua.org/manual/5.3/manual.html#lua_pcallk)或[lua\_callk](http://www.lua.org/manual/5.3/manual.html#lua_callk)。

钩子函数可以在以下条件下产生：只有计数和行事件可以产生;要产生收益，钩子函数必须在 nresult 等于零（即没有值）的情况下完成其执行调用[lua\_yield](http://www.lua.org/manual/5.3/manual.html#lua_yield)。

lua\_sethook

[-0, +0, –]

无效lua\_sethook（lua\_State \*L、lua\_Hook f、整数掩码、整数计数）;

设置调试挂钩函数。

参数 f 是钩子函数。 mask 指定将在哪些事件上调用钩子：它由常量 LUA\_MASKCALL、LUA\_MASKRET、LUA\_MASKLINE 和 LUA\_MASKCOUNT 的按位 OR 组成。count 参数仅在掩码包含 LUA\_MASKCOUNT 时才有意义。对于每个事件，调用钩子如下所述：

* **调用钩子：**在解释器调用函数时调用。钩子是在 Lua 进入新函数之后，在函数获取其参数之前调用的。
* **return hook： 在**解释器从函数返回时调用。钩子是在 Lua 离开函数之前调用的。没有标准方法来访问函数要返回的值。
* **行钩：**当解释器即将开始执行新代码行时调用，或者当它跳回代码中（甚至跳回到同一行）时调用。（此事件仅在 Lua 执行 Lua 函数时发生。
* **count hook：**在解释器执行每个计数指令后调用。（此事件仅在 Lua 执行 Lua 函数时发生。

通过将掩码设置为零来禁用钩子。

lua\_setlocal

[-（0|1）， +0， –]

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

设置给定激活记录的局部变量的值。它将堆栈顶部的值分配给变量并返回其名称。它还会从堆栈中弹出值。

当索引大于活动局部变量的数量时返回 NULL（并且不弹出任何内容）。

参数 ar 和 n 与函数 lua\_getlocal 中相同。

lua\_setupvalue

[-（0|1）， +0， –]

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

设置闭包的上值。它将堆栈顶部的值分配给 upvalue 并返回其名称。它还会从堆栈中弹出值。

当索引 n 大于上行值的数量时返回 NULL（并且不弹出任何内容）。

参数 funcindex 和 n 与函数lua\_getupvalue 中相同。

lua\_upvalueid

[-0, +0, –]

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

返回索引函数索引处闭包编号为 n 的上值的唯一标识符。

这些唯一标识符允许程序检查不同的闭包是否共享上值。共享上值（即访问相同的外部局部变量）的 Lua 闭包将为这些上值索引返回相同的 ID。

参数 funcindex 和 n 与函数 lua\_getupvalue 中相同，但 n 不能大于上行值的数量。

lua\_upvaluejoin

[-0, +0, –]

void lua\_upvaluejoin （lua\_State \*L， int funcindex1， int n1，

int funcindex2， int n2）;

使索引 funcindex1 处 Lua 闭包的第 n1 个上值是指索引 funcindex2 处 Lua 闭包的第 n2 个上值。

5 – 辅助图书馆

*辅助库*提供了几个方便的函数来连接 C 和 Lua。虽然基本 API 为 C 和 Lua 之间的所有交互提供了原始函数，但辅助库为一些常见任务提供了更高级别的功能。

辅助库中的所有函数和类型都在头文件 lauxlib.h 中定义，并具有前缀 luaL\_。

辅助库中的所有函数都构建在基本 API 之上，因此它们不提供该 API 无法完成的任何操作。尽管如此，使用辅助库可确保代码更加一致。

辅助库中的几个函数在内部使用一些额外的堆栈插槽。当辅助库中的函数使用少于五个插槽时，它不会检查堆栈大小;它只是假设有足够的插槽。

辅助库中的几个函数用于检查 C 函数参数。由于错误消息是针对参数格式化的（例如，“错误的参数 #1”），因此不应将这些函数用于其他堆栈值。

如果不满足检查，调用 luaL\_check\* 的函数总是引发错误。

5.1 – 函数和类型

在这里，我们按字母顺序列出了辅助库中的所有函数和类型。

luaL\_addchar

[-?, +？， *米*]

无效luaL\_addchar（luaL\_Buffer \*B，字符 c）;

将字节 c 添加到缓冲区 B（请参见[luaL\_Buffer](http://www.lua.org/manual/5.3/manual.html#luaL_Buffer)）。

luaL\_addlstring

[-?, +？， *米*]

无效luaL\_addlstring （luaL\_Buffer \*B， 常量字符 \*s， size\_t l）;

将长度为 l 的 s 指向的字符串添加到缓冲区 B（请参阅[luaL\_Buffer](http://www.lua.org/manual/5.3/manual.html#luaL_Buffer)）。字符串可以包含嵌入的零。

luaL\_addsize

[-?, +?, –]

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

将先前复制到缓冲区区域的长度为 n 的字符串添加到缓冲区 B（请参阅luaL\_Buffer）（请参阅[luaL\_prepbuffer](http://www.lua.org/manual/5.3/manual.html#luaL_prepbuffer)）。

luaL\_addstring

[-?, +？， *米*]

无效luaL\_addstring（luaL\_Buffer \*B，常量字符 \*s）;

将 s 指向的以零结尾的字符串添加到缓冲区 B（请参阅[luaL\_Buffer](http://www.lua.org/manual/5.3/manual.html#luaL_Buffer)）。

luaL\_addvalue

[-1， +？， *米*]

无效luaL\_addvalue（luaL\_Buffer \*B）;

将堆栈顶部的值添加到缓冲区 B（请参阅[luaL\_Buffer](http://www.lua.org/manual/5.3/manual.html#luaL_Buffer)）。弹出值。

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

luaL\_argcheck

[-0， +0， *v*]

空隙luaL\_argcheck （lua\_State \*L，

国际康德，

int arg，

常量字符 \*extramsg）;

检查 cond 是否为真。如果不是，则使用标准消息引发错误（请参阅[luaL\_argerror](http://www.lua.org/manual/5.3/manual.html#luaL_argerror)）。

luaL\_argerror

[-0， +0， *v*]

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

引发错误，报告调用它的 C 函数的参数参数 arg 出现问题，使用包含 extramsg 作为注释的标准消息：

糟糕的论点 #*arg* to '*funcname*' （*extramsg*）

此函数永远不会返回。

luaL\_Buffer

typedef struct luaL\_Buffer luaL\_Buffer;

*字符串缓冲区*的类型。

字符串缓冲区允许 C 代码逐段构建 Lua 字符串。其使用模式如下：

* 首先声明一个类型 [luaL\_Buffer](http://www.lua.org/manual/5.3/manual.html#luaL_Buffer) 的变量 b。
* 然后用调用 luaL\_buffinit（L， &b） 初始化它。
* 然后将字符串片段添加到调用任何 luaL\_add\* 函数的缓冲区。
* 最后调用luaL\_pushresult（&b）。此调用将最后一个字符串保留在堆栈顶部。

如果您事先知道结果字符串的总大小，则可以像这样使用缓冲区：

* 首先声明一个类型 [luaL\_Buffer](http://www.lua.org/manual/5.3/manual.html#luaL_Buffer) 的变量 b。
* 然后初始化它并使用调用luaL\_buffinitsize（L， &b， sz） 预分配大小为 sz 的空间。
* 然后将字符串复制到该空间中。
* 最后调用 luaL\_pushresultsize（&b， sz），其中 sz 是复制到该空间的结果字符串的总大小。

在正常操作期间，字符串缓冲区使用可变数量的堆栈槽。因此，在使用缓冲区时，您不能假设您知道堆栈的顶部在哪里。您可以在连续调用之间使用堆栈来缓冲操作，只要该使用是平衡的;也就是说，当您调用缓冲区操作时，堆栈处于与上一个缓冲区操作之后的同一级别。（此规则的唯一例外是[luaL\_addvalue](http://www.lua.org/manual/5.3/manual.html#luaL_addvalue)。调用luaL\_pushresult堆栈将返回到初始化缓冲区时的级别，以及其顶部的最后一个字符串。

luaL\_buffinit

[-0, +0, –]

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

初始化缓冲区 B。此函数不分配任何空间;缓冲区必须声明为变量（请参阅[luaL\_Buffer](http://www.lua.org/manual/5.3/manual.html#luaL_Buffer)）。

luaL\_buffinitsize

[-?, +？， *米*]

坦克 \*luaL\_buffinitsize （lua\_State \*L， luaL\_Buffer \*B， size\_t sz）;

相当于序列[luaL\_buffinit](http://www.lua.org/manual/5.3/manual.html#luaL_buffinit)，[luaL\_prepbuffsize](http://www.lua.org/manual/5.3/manual.html#luaL_prepbuffsize)。

luaL\_callmeta

[-0， +（0|1） *和*]

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

调用元方法。

如果索引 obj 处的对象有一个元表，并且这个元表有一个字段 e，则此函数调用此字段，将对象作为其唯一参数传递。在这种情况下，此函数返回 true 并将调用返回的值推送到堆栈上。如果没有元表或元方法，此函数返回 false（不推送堆栈上的任何值）。

luaL\_checkany

[-0， +0， *v*]

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

检查函数在位置参数处是否具有任何类型的参数（包括 **nil**）。

luaL\_checkinteger

[-0， +0， *v*]

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

检查函数参数 arg 是否为整数（或可以转换为整数），并将此整数强制转换为[lua\_Integer](http://www.lua.org/manual/5.3/manual.html#lua_Integer)。

luaL\_checklstring

[-0， +0， *v*]

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

检查函数参数 arg 是否为字符串并返回此字符串;如果 l 不是 NULL 则 用字符串的长度填充 \*l。

此函数使用[lua\_tolstring](http://www.lua.org/manual/5.3/manual.html#lua_tolstring)来获取其结果，因此该函数的所有转换和警告都适用于此处。

luaL\_checknumber

[-0， +0， *v*]

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

检查函数参数 arg 是否为数字并返回此数字。

luaL\_checkoption

[-0， +0， *v*]

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

int arg，

Const char \*def，

const char \*const lst[]）;

检查函数参数 arg 是否为字符串，并在数组 lst 中搜索此字符串 （必须以 NULL 结尾）。返回找到字符串的数组中的索引。如果参数不是字符串或找不到字符串，则引发错误。

如果 def 不是 NULL，则当没有参数 arg 或此参数为 **nil** 时，函数使用 def 作为默认值。

这是将字符串映射到 C 枚举的有用函数。（Lua 库中的通常约定是使用字符串而不是数字来选择选项。

luaL\_checkstack

[-0， +0， *v*]

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

将堆栈大小增加到顶部 + sz 元素，如果堆栈无法增长到该大小，则会引发错误。 msg 是进入错误消息的附加文本（或 NULL 表示没有其他文本）。

luaL\_checkstring

[-0， +0， *v*]

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

检查函数参数 arg 是否为字符串并返回此字符串。

此函数使用[lua\_tolstring](http://www.lua.org/manual/5.3/manual.html#lua_tolstring)来获取其结果，因此该函数的所有转换和警告都适用于此处。

luaL\_checktype

[-0， +0， *v*]

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

检查函数参数 arg 的类型是否为 t。有关 t 类型的编码，请参见[lua\_type](http://www.lua.org/manual/5.3/manual.html#lua_type)。

luaL\_checkudata

[-0， +0， *v*]

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

检查函数参数 arg 是否为 tname 类型的用户数据（请参阅 [luaL\_newmetatable](http://www.lua.org/manual/5.3/manual.html#luaL_newmetatable)），并返回用户数据地址（请参阅 [lua\_touserdata](http://www.lua.org/manual/5.3/manual.html#lua_touserdata)）。

luaL\_checkversion

[-0， +0， *v*]

无效luaL\_checkversion（lua\_State\*L）;

检查运行调用的核心、创建 Lua 状态的核心以及进行调用的代码是否都使用相同的 Lua 版本。还要检查运行调用的核心和创建 Lua 状态的核心是否使用相同的地址空间。

luaL\_dofile

[-0， +？*， 和*]

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

加载并运行给定的文件。它定义为以下宏：

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

如果没有错误，则返回 false，如果出现错误，则返回 true。

luaL\_dostring

[-0, +?, –]

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

加载并运行给定的字符串。它定义为以下宏：

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

如果没有错误，则返回 false，如果出现错误，则返回 true。

luaL\_error

[-0， +0， *v*]

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

引发错误。错误消息格式由 fmt 加上任何额外的参数给出，遵循相同的lua\_pushfstring规则。它还会在消息的开头添加文件名和发生错误的行号（如果此信息可用）。

此函数从不返回，但在 C 函数中将其用作返回luaL\_error（*args*）是一个习惯用法。

luaL\_execresult

[-0， +3， *米*]

你是luaL\_execresult（lua\_State\*L，你是一个国家）;

此函数为标准库（[os.execute](http://www.lua.org/manual/5.3/manual.html#pdf-os.execute) 和 [io.close](http://www.lua.org/manual/5.3/manual.html#pdf-io.close)）中与进程相关的函数生成返回值。

luaL\_fileresult

[-0， +（1|3）， *米*]

int luaL\_fileresult （lua\_State \*L， int stat， const char \*fname）;

此函数为标准库中与文件相关的函数（[io.open、](http://www.lua.org/manual/5.3/manual.html#pdf-io.open)[os.rename](http://www.lua.org/manual/5.3/manual.html#pdf-os.rename)、[file：seek](http://www.lua.org/manual/5.3/manual.html#pdf-file:seek) 等）生成返回值。

luaL\_getmetafield

[-0， +（0|1）， *米*]

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

将索引 obj 处对象的元表中的字段 e 推送到堆栈上，并返回推送值的类型。如果对象没有元表，或者元表没有此字段，则不推送任何内容并返回LUA\_TNIL。

luaL\_getmetatable

[-0， +1， *米*]

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

将与注册表中的名称 tname 关联的元表推送到堆栈上（请参阅[luaL\_newmetatable](http://www.lua.org/manual/5.3/manual.html#luaL_newmetatable)）（如果没有与该名称关联的元表，则**为 nil**）。返回推送值的类型。

luaL\_getsubtable

[-0， +1， *e*]

int luaL\_getsubtable （lua\_State \*L， int idx， const char \*fname）;

确保值 t[fname]（其中 t 是索引 idx 处的值）是一个表，并将该表推送到堆栈上。如果在那里找到以前的表，则返回 true;如果创建新表，则返回 false。

luaL\_gsub

[-0， +1， *米*]

常量字符 \*luaL\_gsub （lua\_State \*L，

常量字符 \*s，

常量字符 \*p，

常量字符 \*r）;

通过将字符串 p 的任何匹配项替换为字符串 r 来创建字符串 s 的副本。将生成的字符串推送到堆栈上并返回它。

luaL\_len

[-0， +0， *e*]

lua\_Integer luaL\_len（lua\_State\*L，整数索引）;

以数字形式返回给定索引处值的“长度”;它等效于 Lua 中的“#”运算符（参见 [§3.4.7](http://www.lua.org/manual/5.3/manual.html#3.4.7)）。如果操作的结果不是整数，则引发错误。（这种情况只能通过元方法发生。

luaL\_loadbuffer

[-0, +1, –]

int luaL\_loadbuffer （lua\_State \*L，

常量字符 \*buff，

size\_t不，

常量字符 \*名称）;

等效于[模式](http://www.lua.org/manual/5.3/manual.html#luaL_loadbufferx)等于 NULL 的luaL\_loadbufferx。

luaL\_loadbufferx

[-0, +1, –]

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

常量字符 \*buff，

size\_t不，

常量字符 \*名称，

常量字符 \*模式）;

将缓冲区加载为 Lua 块。此函数使用 [lua\_load](http://www.lua.org/manual/5.3/manual.html#lua_load) 将块加载到由大小为 sz 的 buff 指向的缓冲区中。

此函数返回与[lua\_load](http://www.lua.org/manual/5.3/manual.html#lua_load)相同的结果。 name 是区块名称，用于调试信息和错误消息。字符串模式的工作方式与函数[lua\_load](http://www.lua.org/manual/5.3/manual.html#lua_load)相同。

luaL\_loadfile

[-0， +1， *米*]

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

等效于[模式](http://www.lua.org/manual/5.3/manual.html#luaL_loadfilex)等于 NULL 的luaL\_loadfilex。

luaL\_loadfilex

[-0， +1， *米*]

int luaL\_loadfilex （lua\_State \*L， const char \*filename，

常量字符 \*模式）;

将文件加载为 Lua 块。此函数使用 [lua\_load](http://www.lua.org/manual/5.3/manual.html#lua_load) 在名为 filename 的文件中加载区块。如果文件名为 NULL，则它从标准输入加载。如果文件中的第一行以 # 开头，则忽略它。

字符串模式的工作方式与函数[lua\_load](http://www.lua.org/manual/5.3/manual.html#lua_load)相同。

此函数返回与 [lua\_load](http://www.lua.org/manual/5.3/manual.html#lua_load) 相同的结果，但它有一个额外的错误代码LUA\_ERRFILE用于与文件相关的错误（例如，它无法打开或读取文件）。

[如lua\_load](http://www.lua.org/manual/5.3/manual.html#lua_load)，此函数仅加载块;它不运行它。

luaL\_loadstring

[-0, +1, –]

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

将字符串加载为 Lua 块。此函数使用 [lua\_load](http://www.lua.org/manual/5.3/manual.html#lua_load) 在以零结尾的字符串 s 中加载块。

此函数返回与[lua\_load](http://www.lua.org/manual/5.3/manual.html#lua_load)相同的结果。

同样[lua\_load](http://www.lua.org/manual/5.3/manual.html#lua_load)，此函数仅加载块;它不运行它。

luaL\_newlib

[-0， +1， *米*]

无效luaL\_newlib （lua\_State \*L， 常量 luaL\_Reg l[]）;

创建一个新表，并在其中注册列表 l 中的函数。

它实现为以下宏：

（luaL\_newlibtable（L，l）， luaL\_setfuncs（L，l，0））

数组 l 必须是实际数组，而不是指向它的指针。

luaL\_newlibtable

[-0， +1， *米*]

无效luaL\_newlibtable （lua\_State \*L， 常量 luaL\_Reg l[]）;

创建一个新表，其大小经过优化，可存储数组 l 中的所有条目（但实际上并不存储它们）。它旨在与[luaL\_setfuncs](http://www.lua.org/manual/5.3/manual.html#luaL_setfuncs)一起使用（见[luaL\_newlib](http://www.lua.org/manual/5.3/manual.html#luaL_newlib)）。

它是作为宏实现的。数组 l 必须是实际数组，而不是指向它的指针。

luaL\_newmetatable

[-0， +1， *米*]

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

如果注册表已有密钥 tname，则返回 0。否则，创建一个要用作 userdata 元表的新表，将 \_\_name = tname 对添加到此新表中，将 [tname] = 新表对添加到注册表中，并返回 1。（条目 \_\_name 由某些错误报告函数使用。

在这两种情况下，都将注册表中与 tname 关联的最终值推送到堆栈上。

luaL\_newstate

[-0, +0, –]

lua\_State \*luaL\_newstate（无效）;

创建新的 Lua 状态。它使用基于标准 C realloc 函数的分配器调用 [lua\_newstate](http://www.lua.org/manual/5.3/manual.html#lua_newstate)，然后设置一个 panic 函数（参见 [§4.6](http://www.lua.org/manual/5.3/manual.html#4.6)），该函数在发生致命错误时将错误消息打印到标准错误输出。

返回新状态，如果存在内存分配错误，则返回 NULL。

luaL\_openlibs

[-0， +0， *e*]

无效luaL\_openlibs（lua\_State\*L）;

将所有标准 Lua 库打开到给定状态。

luaL\_opt

[-0， +0， *e*]

T luaL\_opt （L， func， arg， dflt）;

此宏定义如下：

（lua\_isnoneornil（L，（arg）） ？（dflt） ： func（L，（arg）））

换句话说，如果参数 arg 为 nil 或不存在，则宏将生成默认的 dflt。否则，它会导致调用 func 的结果，状态为 L 和参数索引 arg 作为参数。请注意，它仅在需要时计算表达式 dflt。

luaL\_optinteger

[-0， +0， *v*]

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

int arg，

lua\_Integer（d）;

如果函数参数 arg 是整数（或可转换为整数），则返回此整数。如果此参数不存在或为 **nil**，则返回 d。否则，将引发错误。

luaL\_optlstring

[-0， +0， *v*]

常量字符 \*luaL\_optlstring （lua\_State \*L，

int arg，

常量字符 \*d，

size\_t \*l）;

如果函数参数 arg 是字符串，则返回此字符串。如果此参数不存在或为 **nil**，则返回 d。否则，将引发错误。

如果 l 不是 NULL，则用结果的长度填充位置 \*l。如果结果为 NULL（仅在返回 d 和 d == NULL 时可能），则其长度被视为零。

此函数使用[lua\_tolstring](http://www.lua.org/manual/5.3/manual.html#lua_tolstring)来获取其结果，因此该函数的所有转换和警告都适用于此处。

luaL\_optnumber

[-0， +0， *v*]

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

如果函数参数 arg 是一个数字，则返回此数字。如果此参数不存在或为 **nil**，则返回 d。否则，将引发错误。

luaL\_optstring

[-0， +0， *v*]

常量字符 \*luaL\_optstring （lua\_State \*L，

int arg，

常量字符 \*d）;

如果函数参数 arg 是字符串，则返回此字符串。如果此参数不存在或为 **nil**，则返回 d。否则，将引发错误。

luaL\_prepbuffer

[-?, +？， *米*]

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

等效于具有预定义大小LUAL\_BUFFERSIZE的[luaL\_prepbuffsize](http://www.lua.org/manual/5.3/manual.html" \l "luaL_prepbuffsize)。

luaL\_prepbuffsize

[-?, +？， *米*]

字符 \*luaL\_prepbuffsize （luaL\_Buffer \*B， size\_t sz）;

将地址返回到大小为 sz 的空间，您可以在其中复制要添加到缓冲区 B 的字符串 （请参阅[luaL\_Buffer](http://www.lua.org/manual/5.3/manual.html#luaL_Buffer)）。将字符串复制到此空间后，必须 使用字符串的大小调用luaL\_addsize才能将其实际添加到缓冲区。

luaL\_pushresult

[-?, +1， *米*]

无效luaL\_pushresult（luaL\_Buffer \*B）;

完成缓冲区 B 的使用，将最后一个字符串保留在堆栈顶部。

luaL\_pushresultsize

[-?, +1， *米*]

无效luaL\_pushresultsize（luaL\_Buffer\*B，size\_t sz）;

等效于序列[luaL\_addsize](http://www.lua.org/manual/5.3/manual.html#luaL_addsize)，[luaL\_pushresult](http://www.lua.org/manual/5.3/manual.html#luaL_pushresult)。

luaL\_ref

[-1， +0， *米*]

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

在索引 t 处的表中为堆栈顶部的对象创建并返回引用（并弹出该对象）。

引用是唯一的整数键。只要您不手动将整数键添加到表 t 中，[luaL\_ref](http://www.lua.org/manual/5.3/manual.html#luaL_ref) 就可以确保它返回的键的唯一性。您可以通过 调用 lua\_rawgeti（L， t， r） 来检索引用 r 引用的对象。函数 [luaL\_unref](http://www.lua.org/manual/5.3/manual.html#luaL_unref) 释放引用及其关联的对象。

如果堆栈顶部的对象为 **nil**，[则luaL\_ref](http://www.lua.org/manual/5.3/manual.html#luaL_ref)返回常量LUA\_REFNIL。常量LUA\_NOREF保证与 luaL\_ref 返回的任何引用不同。

luaL\_Reg

typedef struct luaL\_Reg {

常量字符 \*名称;

lua\_CFunction功能;

} luaL\_Reg;

要由[luaL\_setfuncs](http://www.lua.org/manual/5.3/manual.html#luaL_setfuncs)注册的函数数组的类型。 name 是函数名称，func 是指向函数的指针。任何[luaL\_Reg](http://www.lua.org/manual/5.3/manual.html#luaL_Reg)数组都必须以 name 和 func 均为 NULL 的哨兵条目结尾。

luaL\_requiref

[-0， +1， *e*]

void luaL\_requiref （lua\_State \*L， const char \*modname，

lua\_CFunction openf， int glb）;

如果 modname 在 package.loaded 中不存在，则使用字符串 modname 作为参数调用函数 openf，并在 [package.loaded](http://www.lua.org/manual/5.3/manual.html#pdf-package.loaded)[modname] 中设置调用结果，就好像该函数已通过 [require](http://www.lua.org/manual/5.3/manual.html#pdf-require) 调用一样。

如果 glb 为 true，则还将模块存储到全局模组名中。

在堆栈上保留模块的副本。

luaL\_setfuncs

[-nup， +0， *m*]

void luaL\_setfuncs （lua\_State \*L， const luaL\_Reg \*l， int nup）;

将数组 l 中的所有函数（参见 [luaL\_Reg](http://www.lua.org/manual/5.3/manual.html#luaL_Reg)）注册到堆栈顶部的表中（低于可选的上行值，见 next）。

当 nup 不为零时，将创建共享 nup up值的所有函数，这些值必须事先推送到库表顶部的堆栈上。这些值在注册后从堆栈中弹出。

luaL\_setmetatable

[-0, +0, –]

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

将堆栈顶部对象的元表设置为与注册表中的名称 tname 关联的元表（请参阅[luaL\_newmetatable](http://www.lua.org/manual/5.3/manual.html#luaL_newmetatable)）。

luaL\_Stream

typedef struct luaL\_Stream {

文件 \*f;

lua\_CFunction关闭;

} luaL\_Stream;

文件句柄的标准表示形式，由标准 I/O 库使用。

文件句柄作为完整的用户数据实现，具有一个名为 LUA\_FILEHANDLE 的元表（其中LUA\_FILEHANDLE是具有实际元表名称的宏）。元表由 I/O 库创建（请参见[luaL\_newmetatable](http://www.lua.org/manual/5.3/manual.html#luaL_newmetatable)）。

此用户数据必须从结构luaL\_Stream开始;它可以包含此初始结构之后的其他数据。字段 f 指向相应的 C 流（也可以为 NULL 以指示未完全创建的句柄）。Field closef 指向一个 Lua 函数，当句柄关闭或收集时，将调用该函数来关闭流;此函数接收文件句柄作为其唯一参数，并且必须返回 **true**（如果成功）或 **nil** 加上错误消息（如果出现错误）。Lua 调用此字段后，会将字段值更改为 NULL，以指示句柄已关闭。

luaL\_testudata

[-0， +0， *米*]

void \*luaL\_testudata （lua\_State \*L， int arg， const char \*tname）;

此函数的工作方式与[luaL\_checkudata](http://www.lua.org/manual/5.3/manual.html#luaL_checkudata)类似，只是当测试失败时，它会返回 NULL 而不是引发错误。

luaL\_tolstring

[-0， +1， *e*]

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

以合理的格式将给定索引处的任何 Lua 值转换为 C 字符串。生成的字符串被推送到堆栈上，并由函数返回。如果 len 不是 NULL，则该函数还会使用 string 长度设置 \*len。

如果值具有具有\_\_tostring字段的元表，则luaL\_tolstring调用相应的元方法，并将值作为参数，并使用调用的结果作为其结果。

luaL\_traceback

[-0， +1， *米*]

void luaL\_traceback （lua\_State \*L， lua\_State \*L1， const char \*msg，

整数级）;

创建并推送堆栈 L1 的回溯。如果 msg 不是 NULL，则会追加在回溯的开头。level 参数指示在哪个级别开始回溯。

luaL\_typename

[-0, +0, –]

常量字符 \*luaL\_typename （lua\_State \*L， int 索引）;

返回给定索引处的值类型的名称。

luaL\_unref

[-0, +0, –]

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

从索引 t 处的表中释放引用引用（请参阅[luaL\_ref](http://www.lua.org/manual/5.3/manual.html#luaL_ref)）。该条目将从表中删除，以便可以收集引用的对象。引用引用也被释放出来，可以再次使用。

如果 ref [LUA\_NOREF](http://www.lua.org/manual/5.3/manual.html#pdf-LUA_NOREF)或[LUA\_REFNIL](http://www.lua.org/manual/5.3/manual.html#pdf-LUA_REFNIL)，[则luaL\_unref](http://www.lua.org/manual/5.3/manual.html#luaL_unref)不执行任何操作。

luaL\_where

[-0， +1， *米*]

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

将一个字符串推送到堆栈上，该字符串标识控件在调用堆栈中级别 lvl 的当前位置。通常，此字符串具有以下格式：

*区块名称*：*当前行*：

级别 0 是运行函数，级别 1 是调用运行函数的函数，依此类推。

此函数用于为错误消息构建前缀。

6 – 标准库

标准 Lua 库提供了直接通过 C API 实现的有用功能。其中一些函数为语言提供了基本服务（例如，[type](http://www.lua.org/manual/5.3/manual.html#pdf-type)和[getmetatable](http://www.lua.org/manual/5.3/manual.html#pdf-getmetatable)）;其他函数提供对“外部”服务的访问（例如，I / O）;其他函数可以在Lua本身中实现，但非常有用或具有关键的性能要求，值得在C中实现（例如，[table.sort](http://www.lua.org/manual/5.3/manual.html#pdf-table.sort)）。

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

* 基本库 （[§6.1](http://www.lua.org/manual/5.3/manual.html#6.1)）;
* 协程库 （[§6.2](http://www.lua.org/manual/5.3/manual.html#6.2)）;
* 软件包库 （[§6.3](http://www.lua.org/manual/5.3/manual.html#6.3)）;
* 字符串操作 （[§6.4](http://www.lua.org/manual/5.3/manual.html#6.4)）;
* 基本 UTF-8 支持 （[§6.5](http://www.lua.org/manual/5.3/manual.html#6.5)）;
* 表操作 （[§6.6](http://www.lua.org/manual/5.3/manual.html#6.6)）;
* 数学函数（[§6.7](http://www.lua.org/manual/5.3/manual.html#6.7)）（正弦、对数等）;
* 输入和输出（[§6.8](http://www.lua.org/manual/5.3/manual.html#6.8)）;
* 操作系统设施（[§6.9](http://www.lua.org/manual/5.3/manual.html#6.9)）;
* 调试工具 （[§6.10](http://www.lua.org/manual/5.3/manual.html#6.10)）。

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

要访问这些库，C 主机程序应调用 [luaL\_openlibs](http://www.lua.org/manual/5.3/manual.html#luaL_openlibs) 函数，该函数将打开所有标准库。或者，主机程序可以使用 [luaL\_requiref](http://www.lua.org/manual/5.3/manual.html#luaL_requiref) 调用luaopen\_base（对于基本库）、luaopen\_package（对于包库）、luaopen\_coroutine（对于协程库）、luaopen\_string（对于字符串库）、luaopen\_utf8（对于 UTF8 库）、luaopen\_table（对于表库）luaopen\_math（用于数学库）、luaopen\_io（用于 I/O 库）、luaopen\_os（用于操作系统库）和luaopen\_debug（用于调试库）。这些函数在 lualib.h 中声明。

6.1 – 基本功能

基本库为 Lua 提供核心功能。如果未在应用程序中包含此库，则应仔细检查是否需要为其某些工具提供实现。

断言 （v [， 消息]）

如果参数 v 的值为 false（即 **nil** 或 **false**），则调用 [error](http://www.lua.org/manual/5.3/manual.html#pdf-error);否则，返回其所有参数。如果出现错误，则消息是错误对象;如果不存在，则默认为“断言失败！"

收集垃圾（[选项 [， arg]]）

此函数是垃圾回收器的通用接口。它根据其第一个参数执行不同的功能，选择：

* **“**收集**”：**执行完整的垃圾收集周期。这是默认选项。
* **“stop”：**停止垃圾回收器的自动执行。收集器仅在显式调用时运行，直到调用重新启动它。
* **“restart”：**重启垃圾回收器的自动执行。
* **“count”：**返回 Lua 使用的总内存（以 KB 为单位）。该值有一个小数部分，因此乘以 1024 得到 Lua 使用的确切字节数（溢出除外）。
* **“**步骤**”：**执行垃圾收集步骤。步骤“大小”由 arg 控制。如果值为零，收集器将执行一个基本（不可分割）步骤。对于非零值，收集器将执行，就好像 Lua 分配了该内存量（以 KB 为单位）。 如果步骤完成了收集周期，则返回 **true**。
* **“setpause”：**将 arg 设置为收集器暂停 的新值（参见 [§2.5](http://www.lua.org/manual/5.3/manual.html#2.5)）。返回暂停的前一个值。
* **“setstepmul”：**将 arg 设置为收集器的*步进乘数*的新值 （参见 [§2.5](http://www.lua.org/manual/5.3/manual.html#2.5)）。返回步骤的上一个值。
* **“isrunning”：**返回一个布尔值，告知收集器是否正在运行（即未停止）。

DOFILE （[文件名]）

打开命名文件并将其内容作为 Lua 块执行。在没有参数的情况下调用时，dofile 执行标准输入 （stdin） 的内容。返回区块返回的所有值。如果发生错误，dofile 会将错误传播到其调用方（即 dofile 不会在保护模式下运行）。

错误（消息 [，级别]）

终止最后一个调用的受保护函数，并返回消息作为错误对象。函数错误永远不会返回。

通常，如果消息是字符串，则 error 会在消息开头添加有关错误位置的一些信息。level 参数指定如何获取错误位置。对于级别 1（默认值），错误位置是调用错误函数的位置。级别 2 将错误指向调用 error 的函数的调用位置 ;依此类推。传递级别 0 可避免向消息添加错误位置信息。

\_G

保存全局环境的全局变量（不是函数）（参见 [§2.2](http://www.lua.org/manual/5.3/manual.html#2.2)）。Lua 本身不使用此变量;更改其值不会影响任何环境，反之亦然。

获取元表（对象）

如果对象没有元表，则返回 **nil**。否则，如果对象的元表具有\_\_metatable字段，则返回关联的值。否则，返回给定对象的元表。

ipairs（t）

返回三个值（迭代器函数、表 t 和 0），以便构造

对于 i，v in ipairs（t） do *body* end

将迭代键值对 （1，t[1]）、（2，t[2]）、...，直到第一个 nil 值。

load （chunk [， chunkname [， mode [， env]]]）

加载一个区块。

如果块是一个字符串，则块就是这个字符串。如果 chunk 是一个函数，则 load 反复调用它以获取块块。每次调用块都必须返回一个与之前的结果连接的字符串。返回空字符串、**nil** 或无值表示块的结束。

如果没有语法错误，则以函数形式返回已编译的块;否则，返回 **nil** 加上错误消息。

如果生成的函数具有上行值，则第一个上行值将设置为 env 的值（如果给定该参数）或全局环境的值。其他上行值初始化为 **nil**。（当您加载主块时，生成的函数将始终只有一个上行值，即 \_ENV变量（参见 [§2.2](http://www.lua.org/manual/5.3/manual.html#2.2)）。但是，当您加载从函数创建的二进制块时（请参阅 [string.dump），](http://www.lua.org/manual/5.3/manual.html#pdf-string.dump)生成的函数可以具有任意数量的上行值。所有上值都是最新的，也就是说，它们不与任何其他函数共享。

块名用作错误消息和调试信息的块的名称（参见 [§4.9](http://www.lua.org/manual/5.3/manual.html#4.9)）。如果不存在，则默认为 chunk，如果 chunk 是字符串，否则默认为 “=（load）”。

字符串模式控制区块可以是文本还是二进制（即预编译区块）。它可以是字符串“b”（仅二进制块）、“t”（仅文本块）或“bt”（二进制和文本）。默认值为“bt”。

Lua 不检查二进制块的一致性。恶意制作的二进制块可能会使解释器崩溃。

加载文件 （[文件名 [， 模式 [， 环境]]]）

与[加载](http://www.lua.org/manual/5.3/manual.html#pdf-load)类似，但如果未给出文件名，则从文件名或标准输入中获取块。

下一个（表 [、索引]）

允许程序遍历表的所有字段。它的第一个参数是一个表，第二个参数是此表中的索引。next 返回表的下一个索引及其关联值。当使用 **nil** 作为其第二个参数调用时，next 返回初始索引及其关联值。当使用最后一个索引或 空表中的 **nil** 调用时，下一个返回 **nil**。如果第二个参数不存在，则将其解释为 **nil**。特别是，您可以使用 next（t） 检查表是否为空。

未指定枚举索引的顺序，*即使对于数字索引也是如此*。（要按数字顺序遍历表，请使用数字 **为**。

如果在遍历期间为表中不存在的字段分配任何值，则 next 的行为是未定义的。但是，您可以修改现有字段。特别是，您可以清除现有字段。

对 （t）

如果 t 具有元方法\_\_pairs，则使用 t 作为参数调用它， 并返回调用的前三个结果。

否则，返回三个值：[下一个](http://www.lua.org/manual/5.3/manual.html#pdf-next)函数、表 t 和 **nil**，以便构造

对于成对（t）中的k，v做 *身体* 端

将遍历表 T 的所有键值对。

有关在遍历表期间修改表的注意事项，请参阅[下面的](http://www.lua.org/manual/5.3/manual.html#pdf-next)函数。

pcall （f [， arg1， ···]）

在*保护模式下*使用给定参数调用函数 f。这意味着 f 中的任何错误都不会传播;相反，pcall 会捕获错误并返回状态代码。它的第一个结果是状态代码（布尔值），如果调用成功且没有错误，则为 true。在这种情况下，pcall 还会在第一个结果之后返回调用的所有结果。如果出现任何错误，pcall 将返回 **false** 加上错误消息。

打印 （···）

接收任意数量的参数并将其值打印到 stdout，使用 [tostring](http://www.lua.org/manual/5.3/manual.html#pdf-tostring) 函数将每个参数转换为字符串。 print 不用于格式化输出，而只是作为显示值的快速方法，例如用于调试。要完全控制输出，请使用 [string.format](http://www.lua.org/manual/5.3/manual.html#pdf-string.format) 和 [io.write](http://www.lua.org/manual/5.3/manual.html#pdf-io.write)。

原始平等 （v1， v2）

检查 v1 是否等于 v2，而不调用 \_\_eq元方法。返回布尔值。

Rawget （表，索引）

获取表[索引]的实际值，而不调用\_\_index元方法。 表必须是表; 索引可以是任何值。

罗伦 （V）

返回对象 v 的长度，该对象必须是表或字符串，而不调用 \_\_len 元方法。返回一个整数。

数据集（表、索引、值）

将表[索引]的实际值设置为值，而不调用\_\_newindex元方法。 table 必须是表，索引任何不同于 **nil** 和 NaN 的值，并值任何 Lua 值。

此函数返回表。

选择（索引、···）

如果 index 是一个数字，则返回参数编号 index 之后的所有参数;负数从末尾开始索引（-1 是最后一个参数）。否则，索引必须是字符串“#”，并且 select 返回它收到的额外参数的总数。

设置元表（表，元表）

设置给定表的元表。（要从 Lua 代码更改其他类型的元表，必须使用调试库 （[§6.10](http://www.lua.org/manual/5.3/manual.html#6.10)）。如果元表为 **nil**，则删除给定表的元表。如果原始元表具有\_\_metatable字段，则会引发错误。

此函数返回表。

tonumber（和 [， base]）

当调用时没有基数时，tonumber 会尝试将其参数转换为数字。如果参数已经是一个数字或可转换为数字的字符串，则 tonumber 返回此数字;否则，它返回 **nil**。

根据Lua的词汇约定，字符串的转换可以产生整数或浮点数（参见[§3.1](http://www.lua.org/manual/5.3/manual.html#3.1)）。（字符串可以包含前导空格和尾随空格以及符号。

当使用 base 调用时，e 必须是一个字符串，才能被解释为该基数中的整数数字。基数可以是介于 2 和 36 之间的任何整数（包括 2 和 36）。在 10 以上的基数中，字母“A”（大写或小写）表示 10，“B”表示 11，依此类推，“Z”表示 35。如果字符串 e 在给定的基数中不是有效的数字，则该函数返回 **nil**。

托字符串 （V）

接收任何类型的值，并将其转换为人类可读格式的字符串。（要完全控制数字的转换方式，请使用 [string.format](http://www.lua.org/manual/5.3/manual.html#pdf-string.format)。

如果 v 的元表有一个 \_\_tostring 字段，则 tostring 调用相应的值，以 v 作为参数，并使用调用的结果作为其结果。

类型 （V）

返回其唯一参数的类型，编码为字符串。此函数的可能结果是“nil”（字符串，而不是值 **nil**）、“数字”、“字符串”、“布尔值”、“表”、“函数”、“线程”和“用户数据”。

\_版本

一个全局变量（不是函数），它保存包含正在运行的 Lua 版本的字符串。此变量的当前值为“Lua 5.3”。

XPcall （f， msgh [， arg1， ···]）

此函数类似于 [pcall](http://www.lua.org/manual/5.3/manual.html#pdf-pcall)，不同之处在于它设置了新的消息处理程序 msgh。

6.2 – 协程操作

此库包含用于操作协程的操作，这些协程位于表协程内。有关 协程的一般描述，请参见 [§2.6](http://www.lua.org/manual/5.3/manual.html#2.6)。

协程创建 （f）

创建一个新的协程，主体为 f。 f 必须是函数。返回此新协程，即类型为“thread”的对象。

coroutine.isyieldable （）

当正在运行的协程可以生成时返回 true。

如果正在运行的协程不是主线程，并且不在不可生成的 C 函数内，则该协程是可生成的。

coroutine.resume （co [， val1， ···]）

启动或继续执行协程 co。首次恢复协程时，它将开始运行其正文。值 val1， ...作为参数传递给 body 函数。如果协程已生成，则恢复将重新启动它;值 val1， ...作为产量的结果传递。

如果协程运行没有任何错误，则 resume 将返回 **true** 加上传递给 yield 的任何值（当协程产生时）或主体函数返回的任何值（当协程终止时）。如果有任何错误，resume 将 返回 **false** 加上错误消息。

协程运行 （）

返回正在运行的协程加上一个布尔值，当正在运行的协程为主例程时，返回 true。

coroutine.status （co）

以字符串形式返回协程 co的状态：“正在运行”，如果协程正在运行（即称为状态）; “挂起”，如果协程在调用 yield 时挂起，或者如果它尚未开始运行; 如果协程处于活动状态但未运行（即，它已恢复另一个协程），则为“正常”;如果协程已完成其身体功能，或者因错误而停止，则为“死”。

coroutine.wrap （f）

创建一个新的协程，主体为 f。 f 必须是函数。返回一个函数，该函数在每次调用协程时恢复协程。传递给函数的任何参数都表现为要恢复的额外参数。返回 resume 返回的相同值，但第一个布尔值除外。如果出现错误，则传播错误。

协程产量 （···）

暂停调用协程的执行。任何要生成的参数都将作为额外的结果传递以恢复。

6.3 – 模块

包库提供了在 Lua 中加载模块的基本功能。它直接在全局环境中导出一个函数：[require](http://www.lua.org/manual/5.3/manual.html#pdf-require)。其他所有内容都导出到表包中。

需要（模组名称）

加载给定的模块。该函数首先查看 [package.loaded](http://www.lua.org/manual/5.3/manual.html#pdf-package.loaded) 表，以确定 modname 是否已加载。如果是，则 require 返回存储在 package.loaded[modname] 的值。否则，它会尝试为 模块查找*加载程序*。

要查找加载器，需要由 [package.searchers](http://www.lua.org/manual/5.3/manual.html#pdf-package.searchers) 序列引导。通过更改此顺序，我们可以更改 require 对模块的外观。以下说明基于 [package.searchers](http://www.lua.org/manual/5.3/manual.html#pdf-package.searchers) 的默认配置。

首先需要查询 package.preload[modname]。如果它有一个值，则此值（必须是函数）是加载器。否则需要使用存储在 [package.path](http://www.lua.org/manual/5.3/manual.html#pdf-package.path) 中的路径搜索 Lua 加载器。如果这也失败了，它会使用 package.cpath 中存储的路径搜索 C 加载器。如果这也失败了，它会尝试一个*多合一*的加载器（参见[package.searchers](http://www.lua.org/manual/5.3/manual.html#pdf-package.searchers)）。

找到加载器后，需要使用两个参数调用加载器：modname 和一个额外的值，具体取决于它如何获取加载器。（如果加载程序来自文件，则此额外值是文件名。如果加载器返回任何非 nil 值，则 require 将返回的值分配给 package.loaded[modname]。如果加载器没有返回非 nil 值，并且没有为 package.loaded[modname] 分配任何值，则 require 为此条目赋值 **true**。在任何情况下，require都返回package.loaded[modname]的最终值。

如果加载或运行模块时出现任何错误，或者找不到模块的任何加载程序，则 require 会引发错误。

package.config

描述包的一些编译时配置的字符串。此字符串是一系列行：

* 第一行是目录分隔符字符串。对于 Windows，默认值为“\”，对于所有其他系统，默认值为“/”。
* 第二行是分隔路径中模板的字符。默认值为“;'.
* 第三行是标记模板中的替换点的字符串。默认值为“？'.
* 第四行是一个字符串，在 Windows 中的路径中，该字符串被可执行文件的目录替换。默认值为“！'.
* 第五行是在构建luaopen\_函数名称时忽略其后所有文本的标记。默认值为“-”。

package.cpath

[用于](http://www.lua.org/manual/5.3/manual.html#pdf-require)搜索 C 加载程序的路径。

Lua 初始化 C path [package.cpath](http://www.lua.org/manual/5.3/manual.html#pdf-package.cpath) 的方式与初始化 Lua path package.path 的方式相同，使用 environment 变量 LUA\_CPATH\_5\_3、环境变量 LUA\_CPATH 或 luaconf.h 中定义的默认路径。

包.已加载

[由 require](http://www.lua.org/manual/5.3/manual.html#pdf-require) 用于控制已加载哪些模块的表。当你需要一个模块模组名并且package.loaded[modname]不为假时，[require](http://www.lua.org/manual/5.3/manual.html#pdf-require)只需返回存储在那里的值。

此变量只是对实际表的引用;对此变量的赋值不会更改 [require](http://www.lua.org/manual/5.3/manual.html#pdf-require) 使用的表。

package.loadlib （libname， funcname）

将主机程序与 C 库库库名称动态链接。

如果 funcname 为 “\*”，则它仅与库链接，使库导出的符号可用于其他动态链接的库。否则，它会在库中查找函数 funcname，并将此函数作为 C 函数返回。因此，funcname 必须遵循 lua\_CFunction 原型（见[lua\_CFunction](http://www.lua.org/manual/5.3/manual.html#lua_CFunction)）。

这是一个低级函数。它完全绕过了封装和模块系统。与 [require](http://www.lua.org/manual/5.3/manual.html#pdf-require) 不同，它不执行任何路径搜索，也不会自动添加扩展名。 libname 必须是 C 库的完整文件名，如有必要，包括路径和扩展名。 funcname 必须是 C 库导出的确切名称（这可能取决于所使用的 C 编译器和链接器）。

标准 C 不支持此功能。因此，它仅在某些平台上可用（Windows，Linux，Mac OS X，Solaris，BSD以及其他支持dlfcn标准的Unix系统）。

包.路径

[需要](http://www.lua.org/manual/5.3/manual.html#pdf-require)用于搜索 Lua 加载程序的路径。

在启动时，Lua 使用环境变量 LUA\_PATH\_5\_3 或环境变量的值初始化此变量 LUA\_PATH 或使用 luaconf.h 中定义的默认路径（如果未定义这些环境变量）。任何“;;“中的环境变量的值被默认路径替换。

package.preload

用于存储特定模块的加载程序的表（请参阅[要求](http://www.lua.org/manual/5.3/manual.html#pdf-require)）。

此变量只是对实际表的引用;对此变量的赋值不会更改 [require](http://www.lua.org/manual/5.3/manual.html#pdf-require) 使用的表。

package.searchers

由 [require](http://www.lua.org/manual/5.3/manual.html#pdf-require) 用于控制如何加载模块的表。

此表中的每个条目都是一个*搜索器函数*。查找模块时， require 按升序调用每个搜索器，模块名称（给定 [require](http://www.lua.org/manual/5.3/manual.html#pdf-require) 的参数）作为其唯一参数。该函数可以返回另一个函数（模块加载*器*）以及将传递给该加载器的额外值，或者一个解释为什么它没有找到该模块的字符串（如果它 无话可说，则为 **nil**）。

Lua 使用四个搜索器函数初始化此表。

第一个搜索器只是在[package.preload](http://www.lua.org/manual/5.3/manual.html#pdf-package.preload)表中查找加载器。

第二个搜索器使用存储在[package.path](http://www.lua.org/manual/5.3/manual.html#pdf-package.path)的路径查找加载器作为Lua库。搜索按照函数[包中所述](http://www.lua.org/manual/5.3/manual.html#pdf-package.searchpath)完成。

第三个搜索器使用变量 [package.cpath](http://www.lua.org/manual/5.3/manual.html#pdf-package.cpath) 给出的路径查找加载器作为 C 库。同样，搜索按照函数[包.搜索路径](http://www.lua.org/manual/5.3/manual.html#pdf-package.searchpath)中所述完成。例如，如果 C 路径是字符串

"./?.所以;。/?.dll;/usr/local/？/init.so”

模块 foo 的搜索者将尝试按该顺序打开文件 ./foo.so、./foo.dll 和 /usr/local/foo/init.so。找到 C 库后，此搜索器首先使用动态链接工具将应用程序与库链接。然后，它尝试在库内找到一个 C 函数以用作加载器。此 C 函数的名称是字符串“luaopen\_”与模块名称的副本连接，其中每个点都替换为下划线。此外，如果模块名称具有连字符，则删除第一个连字符之后（包括）的后缀。例如，如果模块名称为 a.b.c-v2.1，则函数名称将为luaopen\_a\_b\_c。

第四个搜索器尝试使用*多合一加载程序*。它在 C 路径中搜索库以查找给定模块的根名称。例如，当需要 a.b.c 时，它将搜索 a 的 C 库。如果找到，它会在它中找到子模块的开放函数;在我们的示例中，这将是luaopen\_a\_b\_c。使用此功能，包可以将多个 C 子模块打包到一个库中，每个子模块保持其原始的开放功能。

除第一个（预加载）之外的所有搜索器都返回找到模块的文件名作为额外值，由 [package.searchpath](http://www.lua.org/manual/5.3/manual.html#pdf-package.searchpath) 返回。第一个搜索器不返回任何额外值。

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

在给定路径中搜索给定名称。

路径是一个字符串，其中包含由分号分隔的模板序列。对于每个模板，该函数将模板中的每个询问标记（如果有）替换为名称副本，其中所有出现的 sep（默认情况下为点）都替换为 rep（默认情况下是系统的目录分隔符），然后尝试打开生成的文件名。

例如，如果路径是字符串

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

搜索名称 foo.a 将尝试按该顺序打开文件 ./foo/a.lua、./foo/a.lc 和 /usr/local/foo/a/init.lua。

返回可在读取模式下（关闭文件后）打开的第一个文件的结果名称，如果未成功，则返回 **nil** 加上错误消息。（此错误消息列出了它尝试打开的所有文件名。

6.4 – 字符串操作

此库提供用于字符串操作的通用函数，例如查找和提取子字符串以及模式匹配。在 Lua 中索引字符串时，第一个字符位于位置 1（而不是 C 中的 0）。索引允许为负数，并被解释为从字符串末尾向后索引。因此，最后一个字符位于位置 -1，依此类推。

字符串库在表字符串中提供其所有函数。它还为字符串设置元表，其中\_\_index字段指向字符串表。因此，可以在面向对象的样式中使用字符串函数。例如，string.byte（s，i） 可以写为 s：byte（i）。

字符串库采用单字节字符编码。

string.byte （s [， i [， j]]）

返回字符 s[i]、s[i+1]、...、 s[j] 的内部数字代码。 i 的默认值为 1;j 的默认值为 i。这些索引按照函数 [string.sub](http://www.lua.org/manual/5.3/manual.html#pdf-string.sub) 的相同规则进行更正。

数字代码不一定可以跨平台移植。

字符串.字符 （···）

接收零个或多个整数。返回长度等于参数数的字符串，其中每个字符的内部数字代码等于其相应的参数。

数字代码不一定可以跨平台移植。

string.dump （function [， strip]）

返回一个字符串，其中包含给定函数的二进制表示形式（*二进制块*），以便稍后 对此字符串的加载返回函数的副本（但具有新的上行值）。如果 strip 是 true 值，则二进制表示形式可能不包括有关函数的所有调试信息，以节省空间。

具有上值的函数仅保存其上值数。当（重新）加载时，这些上值会收到包含 **nil** 的新实例。（可以使用调试库以足以满足需求的方式序列化和重新加载函数的上行值。

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

在字符串 s 中查找模式的第一个匹配项（请参阅 [§6.4.1](http://www.lua.org/manual/5.3/manual.html#6.4.1)）。如果找到匹配项，则 find 返回此事件开始和结束的 s 的索引;否则，它返回 **nil**。第三个可选数字参数 init 指定从何处开始搜索;其默认值为 1，可以为负数。值 **true** 作为第四个可选参数 plain 会关闭模式匹配功能，因此该函数执行普通的“查找子字符串”操作，模式中的任何字符都被视为魔法。请注意，如果给出 plain，那么 也必须给出 init。

如果模式具有捕获，则在成功匹配中，还会在两个索引之后返回捕获的值。

string.format （formatstring， ···）

返回其变量参数数的格式化版本，其后跟在其第一个参数（必须是字符串）中给出的说明之后。格式字符串遵循与 ISO C 函数 sprintf 相同的规则。唯一的区别是不支持选项/修饰符 \*、h、L、l、n 和 p，并且还有一个额外的选项 q。

q 选项在双引号之间格式化字符串，必要时使用转义序列，以确保 Lua 解释器可以安全地读回它。例如，调用

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

可能会产生字符串：

“带有 \”引号“和 \ 的字符串

新行”

选项 A、A、E、E、F、 G 和 g 都需要一个数字作为参数。选项 c、d、i、o、u 、 X 和 x 需要一个整数。当使用 C89 编译器编译 Lua 时，选项 A 和 A（十六进制浮点数）不支持任何修饰符（标志、宽度、长度）。

选项 s 需要一个字符串;如果它的参数不是字符串，则按照相同的 tostring 规则将其转换为字符串。如果该选项有任何修饰符（标志、宽度、长度），则字符串参数不应包含嵌入的零。

string.gmatch （s， pattern）

返回一个迭代器函数，每次调用该函数时，都会通过字符串 s 返回来自模式的下一个捕获（参见 [§6.4.1](http://www.lua.org/manual/5.3/manual.html#6.4.1)）。如果模式未指定捕获，则在每次调用中生成整个匹配项。

例如，以下循环将遍历字符串 s 中的所有单词，每行打印一个：

s = “来自 Lua 的你好世界”

对于 String.gmatch（s， “%a+”） 中的 w，请执行

打印（W）

结束

下一个示例将给定字符串中的所有对 key=value 收集到一个表中：

t = {}

s = “from=world， to=Lua”

对于 k， v in string.gmatch（s， “（%w+）=（%w+）”） do

t[k] = v

结束

对于此函数，模式开头的插入符号“^”不能用作锚点，因为这会阻止迭代。

string.gsub （s， pattern， repl [， n]）

返回 s 的副本，其中模式的所有（或前 n，如果给定）出现（参见 [§6.4.1](http://www.lua.org/manual/5.3/manual.html#6.4.1)）已替换为 repl 指定的替换字符串，该字符串可以是字符串、表或函数。 gsub 还返回第二个值，作为其发生的匹配总数。gsub这个名字来自*Global SUBstitution*。

如果 repl 是一个字符串，则其值用于替换。字符 % 用作转义字符：任何形式为 %d 的 repl 序列，d 介于 1 和 9 之间，代表 捕获的第 *d* 个子字符串的值。序列 %0 代表整个匹配项。序列 %% 代表单个 %。

如果 repl 是一个表，则使用第一个捕获作为键来查询每个匹配项的表。

如果 repl 是一个函数，则每次发生匹配时都会调用此函数，所有捕获的子字符串都按顺序作为参数传递。

在任何情况下，如果模式未指定捕获，则其行为就像整个模式都在捕获中一样。

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

以下是一些示例：

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 = $return 4+5$”， “%$（.-）%$”， 函数 （s）

回程载荷（s）（）

完）

--> x=“4+5 = 9”

本地 t = {name=“lua”， version=“5.3”}

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

--> x=“lua-5.3.tar.gz”

string.len （s）

接收字符串并返回其长度。空字符串 “” 的长度为 0。嵌入的零被计算在内，因此“a\000bc\000”的长度为 5。

字符串.下 （s）

接收一个字符串并返回此字符串的副本，其中所有大写字母都更改为小写。所有其他字符保持不变。大写字母的定义取决于当前区域设置。

string.match （s， pattern [， init]）

在字符串 s 中查找模式的第一个*匹配*项（请参阅 [§6.4.1](http://www.lua.org/manual/5.3/manual.html#6.4.1)）。如果找到一个，则 match 返回模式中的捕获;否则返回 **nil**。如果模式未指定捕获，则返回整个匹配项。第三个可选数字参数 init 指定从何处开始搜索;其默认值为 1，可以为负数。

string.pack （fmt， v1， v2， ···）

返回一个二进制字符串，其中包含根据格式字符串 fmt 打包（即以二进制形式序列化）的值 v1、v2 等（请参阅 [§6.4.2](http://www.lua.org/manual/5.3/manual.html#6.4.2)）。

string.packsize （FMT）

返回由给定格式的 [string.pack](http://www.lua.org/manual/5.3/manual.html#pdf-string.pack) 生成的字符串的大小。格式字符串不能具有可变长度选项“s”或“z”（请参阅 [§6.4.2](http://www.lua.org/manual/5.3/manual.html#6.4.2)）。

string.rep （s， n [， sep]）

返回一个字符串，该字符串是由字符串 sep 分隔的字符串 s 的 n 个副本的串联。sep 的默认值为空字符串（即没有分隔符）。如果 n 不为正数，则返回空字符串。

（请注意，通过对此函数的单个调用，很容易耗尽计算机的内存。

string.reverse （s）

返回一个字符串，该字符串是字符串 s 反转。

string.sub （s， i [， j]）

返回从 i 开始 并一直持续到 j 的 s 子字符串; i 和 j 可以是负数。如果 j 不存在，则假定它等于 -1（与字符串长度相同）。特别是，调用 string.sub（s，1，j） 返回长度为 j 的前缀 s，而 string.sub（s， -i）（对于正 i）返回长度为 i 的后缀 s。

如果在负指数转换后，i 小于 1，则更正为 1。如果 j 大于字符串长度，则将其更正为该长度。如果在这些更正之后，i 大于 j，则该函数返回空字符串。

string.unpack （fmt， s [， pos]）

根据字符串 fmt 格式返回打包在字符串 s 中的值（请参阅 [string.pack](http://www.lua.org/manual/5.3/manual.html#pdf-string.pack)）（参见 [§6.4.2](http://www.lua.org/manual/5.3/manual.html#6.4.2)）。可选的 pos 标记在 s 中开始读取的位置 （默认值为 1）。在读取值之后，此函数还返回第一个未读字节的索引（以 s 为单位）。

字符串.上部 （s）

接收一个字符串并返回此字符串的副本，其中所有小写字母都更改为大写。所有其他字符保持不变。小写字母的定义取决于当前区域设置。

6.4.1 – 模式

Lua 中的模式由常规字符串描述，这些字符串由模式匹配函数 string.find、 string.gmatch、[string.gsub](http://www.lua.org/manual/5.3/manual.html#pdf-string.gsub) 和 [string.match](http://www.lua.org/manual/5.3/manual.html#pdf-string.match) 解释为模式。本节介绍这些字符串的语法和含义（即它们匹配的内容）。

角色类：

字符*类*用于表示一组字符。在描述字符类时，允许使用以下组合：

* ***x*：**（其中 *x* 不是*魔术字符*之一 ^$（）%.[]\*+-?） 表示字符 *x* 本身。
* **.：**（点）表示所有字符。
* **%a：**表示所有字母。
* **%c：**表示所有控制字符。
* **%d：**表示所有数字。
* **%g：**表示除空格之外的所有可打印字符。
* **%l：**表示所有小写字母。
* **%p：**表示所有标点字符。
* **%s：**表示所有空格字符。
* **%u：**表示所有大写字母。
* **%w：**表示所有字母数字字符。
* **%x：**表示所有十六进制数字。
* **%*x*：**（其中 x 是任何非字母数字字符）表示字符 *x*。这是逃离魔法字符的标准方法。任何非字母数字字符（包括所有标点符号，甚至是非魔法字符）在用于在模式中表示自身时都可以在前面加上“%”。
* **[*套*]：**表示作为*集合*中所有字符并集的类。可以通过用“-”按升序分隔范围的结束字符来指定字符范围。上面描述的所有类 %*x* 也可以用作集合中的组件。集合中的所有其他角色 都代表他们自己。例如，[%w\_]（或 [\_%w]）表示所有字母数字字符加上下划线，[0-7] 表示八进制数字，[0-7%l%-] 表示八进制数字加上小写字母加上“-”字符。

通过将右方括号定位为集合中的第一个字符，可以将右方括号放在集合中。您可以通过将连字符定位为集合中的第一个或最后一个字符来将其放入集合中。（您也可以对这两种情况使用转义。

未定义范围和类之间的交互。因此，像 [%a-z] 或 [a-%%] 这样的模式没有任何意义。

* **[^*集*]：**表示集合的补码，其中*集合*解释如上。

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

字母、空格和其他字符组的定义取决于当前区域设置。特别是，类 [a-z] 可能不等同于 %l。

图案项目：

*图案项*可以是

* 单个字符类，与类中的任何单个字符匹配;
* 后跟“\*”的单个字符类，用于匹配类中零个或多个重复字符。这些重复项目将始终匹配尽可能长的序列;
* 后跟“+”的单个字符类，它匹配类中一个或多个重复的字符。这些重复项目将始终匹配尽可能长的序列;
* 后跟“-”的单个字符类，该类还匹配类中零个或多个重复字符。与“\*”不同，这些重复项目将始终匹配尽可能短的序列;
* 单个字符类，后跟“？'，它匹配类中出现的零个或一个字符。如果可能，它始终匹配一个匹配项;
* %*n*，表示 1 到 9 之间的 n;此类项目匹配等于第 *n* 个捕获字符串的子字符串（见下文）;
* %b*xy*，其中 x 和 y 是两个不同的字符;此类项匹配以 x 开头、以 y 结尾以及 *x* 和 *y* 平衡的字符串。这意味着，如果从左到右读取字符串，对 *x* 计数 *+1*，对 y 计数 *-1* ，则结尾 y 是 计数达到 0 的第一个 *y*。例如，项 %b（） 匹配带有平衡括号的表达式。
* %f[set]，*边框模式*;此类项目在任何位置匹配空字符串，以便下一个字符属于 set ，而前一个字符不属于 *set*。集合 集按前面所述进行解释。主题的开头和结尾被处理为字符“\0”。

模式：

模式是一系列模式项目。模式开头的插入符号“^”将匹配项定位在主题字符串的开头。模式末尾的“$”将匹配项定位在主题字符串的末尾。在其他位置，“^”和“$”没有特殊含义，代表自己。

捕获：

模式可以包含括在括号中的子模式;他们描述*捕获*。匹配成功后，将存储（捕获）匹配捕获的主题字符串的子字符串以供将来使用。捕获根据其左括号进行编号。例如，在模式“（a\*（.）%w（%s\*））“，字符串中与”a\*（.） “匹配的部分%w（%s\*）“存储为第一个捕获（因此具有数字 1）;字符匹配 ”.“捕获的数字为 2，匹配 ”%s\*“ 的部分捕获的数字为 3。

作为特殊情况，空捕获 （） 捕获当前字符串位置（数字）。例如，如果我们在字符串“flaaap”上应用模式“（）aa（）”，将有两个捕获：3和5。

6.4.2 – 格式化打包和解包的字符串

string.pack、[string.packsize](http://www.lua.org/manual/5.3/manual.html#pdf-string.packsize) 和 [string.unpack](http://www.lua.org/manual/5.3/manual.html#pdf-string.unpack) 的第一个参数是一个格式字符串，它描述了正在创建或读取的结构的布局。

格式字符串是一系列转换选项。转换选项如下：

* **<：**设置小端序
* **>：**设置大端序
* **=：**设置本机字节序
* **![*n*]：**将最大对齐方式设置为 n（默认值为本机对齐方式）
* **B：**有符号字节（字符）
* **B：**无符号字节（字符）
* **H：**有符号的短裤（原始大小）
* **H：**无符号短短（原生大小）
* **L：**带符号的长整型（原生大小）
* **L：**无符号长整型（本机大小）
* **J：**lua\_Integer
* **J：**lua\_Unsigned
* **T：**size\_t （原始大小）
* **i[*n*]：**  具有 n 个字节的有符号整数（默认为本机大小）
* **I[*n*]：**  具有 n 个字节的无符号整数（默认为本机大小）
* **F：**浮点数（原生尺寸）
* **D：**  双人（原始尺寸）
* **N：**lua\_Number
* **c n：**具有 n 个字节的固定大小的字符串
* **z：**以零结尾的字符串
* **s[*n*]：**一个字符串，前面的长度编码为具有 n 个字节的无符号整数 （默认为 size\_t）
* **x：**一个字节的填充
* **X** op**：**根据选项 op 对齐的空项 （否则将被忽略）
* **' '：**（空白区域）忽略

（“[*n*]”表示可选的整数。除了填充、空格和配置（选项“xX <=>！“），每个选项对应于一个参数（在 [string.pack](http://www.lua.org/manual/5.3/manual.html#pdf-string.pack) 中）或一个结果（在 [string.unpack](http://www.lua.org/manual/5.3/manual.html#pdf-string.unpack) 中）。

对于选项“！*n*“、”s n“、”i n“和”I n“，*n* 可以是 1 到 16 之间的任何整数。所有积分选项检查溢出;[string.pack](http://www.lua.org/manual/5.3/manual.html#pdf-string.pack) 检查给定的值是否适合给定的大小; [string.unpack](http://www.lua.org/manual/5.3/manual.html#pdf-string.unpack) 检查读取值是否适合 Lua 整数。

任何格式字符串都以“！1=”开头，即最大对齐方式为 1（无对齐方式）和本机字节序。

对齐方式如下：对于每个选项，格式将获得额外的填充，直到数据从选项大小和最大对齐方式之间的最小值的倍数的偏移量开始;此最小值必须为 2 的幂。选项“c”和“z”未对齐;选项 “s” 遵循其起始整数的对齐方式。

所有填充都由string.pack填充零（并由[string.unpack](http://www.lua.org/manual/5.3/manual.html#pdf-string.unpack)忽略）。

6.5 – UTF-8 支持

此库提供对 UTF-8 编码的基本支持。它在表 utf8 中提供了所有功能。除了处理编码之外，此库不提供对 Unicode 的任何支持。任何需要字符含义的操作（如字符分类）都超出了其范围。

除非另有说明，否则所有期望字节位置作为参数的函数都假定给定位置是字节序列的开头或 1 加上主题字符串的长度。与字符串库一样，负索引从字符串末尾开始计数。

utf8.char （···）

接收零个或多个整数，将每个整数转换为其相应的 UTF-8 字节序列，并返回一个字符串，其中包含所有这些序列的串联。

utf8.charpattern

模式（字符串，而不是函数）“[\0-\x7F\xC2-\xF4][\x80-\xBF]\*”（参见 [§6.4.1](http://www.lua.org/manual/5.3/manual.html#6.4.1)），它正好匹配一个 UTF-8 字节序列，假设主题是有效的 UTF-8 字符串。

utf8.代码 （s）

返回值，以便构造

对于 utf8 中的 p、c，代码执行 *正文* 结束

将迭代字符串 S 中的所有字符，P 是位置（以字节为单位），C 是每个字符的代码点。如果它遇到任何无效的字节序列，则会引发错误。

utf8.codepoint （s [， i [， j]]）

返回 s 中从字节位置 i 和 j（都包括在内）开始的所有字符的代码点（整数形式）。i 的默认值为 1，j 的默认值为 i。如果它遇到任何无效的字节序列，则会引发错误。

utf8.len （s [， i [， j]]）

返回字符串 s 中从位置 i 和 j（包括两者）开始的 UTF-8 字符数。i 的默认值为 1，j 的默认值为 -1。如果找到任何无效的字节序列，则返回一个 false 值加上第一个无效字节的位置。

utf8.offset （s， n [， i]）

返回 s 的第 n 个字符的编码（从位置 i 开始计数）开始的位置（以字节为单位）。负 n 获取位置 i 之前的字符。当 n 为非负 数时，i 的默认值为 1，否则为 #s + 1，因此 utf8.offset（s， -n） 从字符串末尾获取第 n 个字符的偏移量。如果指定的字符既不在主语中，也不在其结尾之后，则该函数返回 **nil**。

作为特殊情况，当 n 为 0 时，该函数返回包含 s 的第 i 个字节的字符编码的开头。

此函数假定 s 是有效的 UTF-8 字符串。

6.6 – 表操作

此库提供用于表操作的通用函数。它在表内提供其所有功能。

请记住，每当操作需要表的长度时，有关长度运算符的所有警告都适用（参见 [§3.4.7](http://www.lua.org/manual/5.3/manual.html#3.4.7)）。所有函数都忽略表中作为参数给出的非数字键。

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

给定一个所有元素都是字符串或数字的列表，返回字符串列表[i]。九月。。列表[I+1] ···九月。。列表[J]。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）

在列表中的位置位置插入元素值，向上移动元素列表[pos]，列表[pos+1]，···，列表[#list]。pos 的默认值 为 #list+1，因此调用表.insert（t，x） 在列表 t 的末尾 插入 x。

table.move （a1， f， e， t [，a2]）

将元素从表 a1 移动到表 a2，执行等效于以下多重赋值：a2[t]，··· = a1[f]，···，a1[e]。a2 的默认值为 a1。目标范围可以与源范围重叠。要移动的元素数必须适合 Lua 整数。

返回目标表 a2。

表包 （···）

返回一个新表，其中包含所有参数存储在键 1、2 等中，字段“n”包含参数总数。请注意，生成的表可能不是一个序列。

table.remove （list [， pos]）

从列表中删除位置位置的元素，返回已删除元素的值。当 pos 是介于 1 和 #list 之间的整数时，它会向下移动元素列表[pos+1]、列表[pos+2]、·、列表[#list]并擦除元素列表[#list];当#list为 0 时，索引 pos 也可以为 0，或者 #list + 1;在这些情况下，函数会擦除元素列表[pos]。

pos 的默认值是 #list，因此调用 table.remove（l） 会删除列表 l 的最后一个元素。

table.sort （list [， comp]）

按给定顺序就*地*从列表 [1] 到列表 [#list] 对列表元素进行排序。如果给定 comp，那么它必须是一个函数，它接收两个列表元素，并在第一个元素必须在最终顺序中的第二个元素之前时返回 true（因此，在排序之后， i < j 表示不是 comp（list[j]，list[i]））。如果未给出 comp，则改用标准 Lua 运算符<。

请注意，comp 函数必须对列表中的元素定义严格的偏序;也就是说，它必须是非对称和传递的。否则，可能无法进行有效的排序。

排序算法不稳定：给定顺序认为相等的元素的相对位置可能会被排序改变。

table.unpack （list [， i [， j]]）

返回给定列表中的元素。此函数等效于

返回列表[i]， 列表[i+1]， ···， 列表[j]

默认情况下，i 为 1，j 为 #list。

6.7 – 数学函数

该库提供基本的数学函数。它在表数学中提供其所有函数和常量。带有注释“整数/浮点数”的函数为整数参数提供整数结果，为浮点（或混合）参数提供浮点结果。舍入函数（[math.ceil](http://www.lua.org/manual/5.3/manual.html#pdf-math.ceil)、[math.floor](http://www.lua.org/manual/5.3/manual.html#pdf-math.floor) 和 [math.modf](http://www.lua.org/manual/5.3/manual.html#pdf-math.modf)）在结果适合整数范围时返回整数，否则返回浮点数。

数学腹肌 （x）

返回 x 的绝对值。（整数/浮点数）

math.acos （x）

返回 x 的弧余弦（以弧度为单位）。

数学.asin （x）

返回 x 的弧正弦（以弧度为单位）。

math.atan （y [， x]）

返回 y/x 的反正切值（以弧度为单位），但使用两个参数的符号来查找结果的象限。（它还可以正确处理以下情况： x 为零。

x 的默认值为 1，因此调用 math.atan（y） 返回 y 的反正切。

数学.ceil （x）

返回大于或等于 x 的最小整数值。

数学.cos （x）

返回 x 的余弦值（假定以弧度为单位）。

数学度 （x）

将角度 x 从弧度转换为度。

math.exp （x）

返回值 e *x*（其中 e 是自然对数的底数）。

数学地板 （X）

返回小于或等于 x 的最大整数值。

math.fmod （x， y）

返回将商舍入到零的 x 除以 y 的余数。（整数/浮点数）

数学.巨大

浮点值HUGE\_VAL，该值大于任何其他数值。

数学.log （x [， 基数]）

返回给定底数中 x 的对数。base 的默认值 为 *e*（以便函数返回 x 的自然对数）。

数学.max （x， ···）

根据 Lua 运算符<返回具有最大值的参数。（整数/浮点数）

math.maxinteger

具有整数最大值的整数。

math.min （x， ···）

根据 Lua 运算符<返回具有最小值的参数。（整数/浮点数）

math.mininteger

具有整数的最小值的整数。

math.modf （x）

返回 x 的整数部分和 x 的小数部分。它的第二个结果始终是浮点数。

数学.pi

*π*的值。

math.rad （x）

将角度 x 从度转换为弧度。

math.random （[m [， n]]）

在没有参数的情况下调用时，返回一个在 *[0，1）* 范围内均匀分布的伪随机浮点数。当使用两个整数 m 和 n 调用时，math.random 返回一个伪随机整数，其均匀分布范围为 *[m， n]。* （值 *n-m* 不能为负数，并且必须适合 Lua 整数。调用 math.random（n） 等效于 math.random（1，n）。

此函数是 C 提供的底层伪随机生成器函数的接口。

数学随机种子 （X）

将 x 设置为伪随机生成器的“种子”：相等的种子产生相等的数字序列。

数学罪 （x）

返回 x 的正弦值（假定以弧度为单位）。

math.sqrt （x）

返回 x 的平方根。（您也可以使用表达式 x^0.5 来计算此值。

Math.tan （x）

返回 x 的正切值（假定以弧度为单位）。

math.tointeger （x）

如果值 x 可转换为整数，则返回该整数。否则，返回 **nil**。

数学类型 （x）

如果 x 是整数，则返回“整数”;如果它是浮点数，则返回“float”;如果 x 不是数字，则返回 **nil**。

math.ult （m， n）

返回布尔值 true，当且仅当整数 m 在 将它们作为无符号整数进行比较时低于整数 n。

6.8 – 输入和输出设施

I/O 库为文件操作提供了两种不同的样式。第一个使用隐式文件句柄;也就是说，存在设置默认输入文件和默认输出文件的操作，并且所有输入/输出操作都基于这些默认文件。第二种样式使用显式文件句柄。

使用隐式文件句柄时，所有操作都由表 io 提供。使用显式文件句柄时，操作 [io.open](http://www.lua.org/manual/5.3/manual.html#pdf-io.open) 返回文件句柄，然后所有操作都作为文件句柄的方法提供。

表 io 还提供了三个预定义的文件句柄，它们通常的含义来自 C： io.stdin、 io.stdout 和 io.stderr。I/O 库永远不会关闭这些文件。

除非另有说明，否则所有 I/O 函数在失败时返回 **nil**（加上作为第二个结果的错误消息和作为第三个结果的系统相关错误代码）和一些与成功时不同的值 。在非 POSIX 系统中，发生错误时错误消息和错误代码的计算可能不是线程安全的，因为它们依赖于全局 C 变量 errno。

io.close （[文件]）

等效于 file：close（）。如果没有文件，则关闭默认输出文件。

io.flush （）

等效于 io.output（）：flush（）。

IO.输入 （[文件]）

使用文件名调用时，它会打开命名文件（在文本模式下），并将其句柄设置为默认输入文件。使用文件句柄调用时，它只是将此文件句柄设置为默认输入文件。在没有参数的情况下调用时，它将返回当前默认输入文件。

如果出现错误，此函数将引发错误，而不是返回错误代码。

io.lines （[文件名， ···]）

在读取模式下打开给定的文件名，并返回一个迭代器函数，其工作原理类似于 file：lines（···） 在打开的文件上。当迭代器函数检测到文件末尾时，它不返回任何值（以完成循环）并自动关闭文件。

调用 io.lines（）（没有文件名）等效于 io.input（）：lines（“\*l”）;也就是说，它遍历默认输入文件的行。在这种情况下，迭代器不会在循环结束时关闭文件。

如果出现错误，此函数将引发错误，而不是返回错误代码。

io.open （文件名 [， mode]）

此函数以字符串模式中指定的模式打开文件。如果成功，它将返回一个新的文件句柄。

模式字符串可以是以下任何一种：

* **“r”：**读取模式（默认）;
* **“w”：**写入模式;
* **“a”：**追加模式;
* **“r+”：**更新模式，保留所有以前的数据;
* **“w+”：**更新模式，之前的所有数据都被擦除;
* **“A+”：**追加更新模式，保留以前的数据，只允许在文件末尾写入。

模式字符串的末尾也可以有一个“b”，在某些系统中需要以二进制模式打开文件。

IO.输出 （[文件]）

与 [io.input](http://www.lua.org/manual/5.3/manual.html#pdf-io.input) 类似，但对默认输出文件进行操作。

io.popen （prog [， mode]）

此功能取决于系统，并非在所有平台上都可用。

在单独的进程中启动程序程序，并返回可用于从此程序读取数据（如果模式为“r”，则为默认值）或将数据写入此程序（如果模式为 “w”）的文件句柄。

io.read （···）

等效于 io.input（）：read（···）.

io.tmpfile （）

如果成功，则返回临时文件的句柄。此文件在更新模式下打开，并在程序结束时自动删除。

io.type （obj）

检查 obj 是否为有效的文件句柄。如果 obj 是打开的文件句柄，则返回字符串“file”;如果 obj 是关闭的文件句柄，则返回字符串“关闭的文件”;如果 obj 不是文件句柄 ，则返回**字符串 nil**。

io.write （···）

等效于 io.output（）：write（···）.

文件：关闭 （）

关闭文件。请注意，当文件的句柄被垃圾回收时，文件会自动关闭，但这需要不可预测的时间才能发生。

关闭使用 [io.popen](http://www.lua.org/manual/5.3/manual.html#pdf-io.popen) 创建的文件句柄时，[file：close](http://www.lua.org/manual/5.3/manual.html#pdf-file:close) 返回与 [os.execute](http://www.lua.org/manual/5.3/manual.html#pdf-os.execute) 返回的相同值。

文件：同花顺 （）

将任何写入的数据保存到文件中。

文件：行 （···）

返回一个迭代器函数，每次调用该函数时，都会根据给定的格式读取文件。如果未给出格式，则使用“l”作为默认值。例如，建筑

对于文件中的 C：行（1） 做 *正文* 结束

将从当前位置开始遍历文件的所有字符。与 [io.lines](http://www.lua.org/manual/5.3/manual.html#pdf-io.lines) 不同，此函数在循环结束时不会关闭文件。

如果出现错误，此函数将引发错误，而不是返回错误代码。

文件：读取 （···）

根据指定要读取的内容的给定格式读取文件文件。对于每种格式，函数返回一个字符串或数字，其中读取的字符，如果它无法读取指定格式的数据，则返回 nil。（在后一种情况下，该函数不会读取后续格式。在没有格式的情况下调用时，它使用读取下一行的默认格式（见下文）。

可用的格式是

* **“n”：**读取数字并将其作为浮点数或整数返回，遵循 Lua 的词法约定。（数字可以有前导空格和符号。此格式始终读取作为数字的有效前缀的最长输入序列;如果该前缀未形成有效的数字（例如，空字符串、“0x”或“3.4e-”），则丢弃它，函数返回 **nil**。
* **“a”：**从当前位置开始读取整个文件。在文件末尾，它返回空字符串。
* **“l”：**读取下一行跳过行尾，在文件末尾返回 **nil**。这是默认格式。
* **“L”：**读取下一行，保留行尾字符（如果存在）， 在文件末尾返回 **nil**。
* ***number*：**读取最多包含此字节数的字符串， 在文件末尾返回 nil。如果 number 为零，则不读取任何内容，并在文件末尾返回空字符串或 **nil**。

格式“l”和“L”应仅用于文本文件。

文件：查找（[从哪里 [， 偏移量]]）

设置并获取文件位置（从文件开头开始测量）到偏移量加上字符串 whence 指定的基数给出的位置，如下所示：

* **“set”：**base 是位置 0（文件的开头）;
* **“cur”：**基数是当前位置;
* **“结束”：**基是文件的结尾;

如果成功，seek 将返回最终文件位置，以文件开头的字节为单位。如果查找失败，则返回 **nil**，以及描述错误的字符串。

其中的默认值为“cur”，偏移量的默认值为 0。因此，调用 file：seek（） 返回当前文件位置，而不更改它;调用 file：seek（“set”） 将位置设置为文件的开头（并返回 0）;调用 file：seek（“end”） 将位置设置为文件的末尾，并返回其大小。

文件：setvbuf （模式 [， 大小]）

设置输出文件的缓冲模式。有三种可用的模式：

* **“否”：**无缓冲;任何输出操作的结果都会立即显示。
* **“**完全**”：**完全缓冲;仅当缓冲区已满或显式刷新文件时，才会执行输出操作 （请参阅 [io.flush](http://www.lua.org/manual/5.3/manual.html#pdf-io.flush)）。
* **“**行**”：**行缓冲;输出被缓冲，直到输出换行符或某些特殊文件（如终端设备）有任何输入。

对于最后两种情况，size 指定缓冲区的大小（以字节为单位）。默认值为适当的大小。

文件：写入 （···）

将其每个参数的值写入文件。参数必须是字符串或数字。

如果成功，此函数将返回文件。否则，它将返回 **nil** 加上描述错误的字符串。

6.9 – 操作系统设施

此库是通过表 os 实现的。

操作系统时钟 （）

返回程序使用的 CPU 时间量的近似值（以秒为单位）。

os.date （[格式 [， time]]）

返回一个字符串或包含日期和时间的表，根据给定的字符串格式设置格式。

如果存在 time 参数，则这是要格式化的时间（有关此值的说明，请参阅 [os.time](http://www.lua.org/manual/5.3/manual.html#pdf-os.time) 函数）。否则，日期将设置当前时间的格式。

如果格式以“！'，则日期的格式为协调世界时。在此可选字符之后，如果格式为字符串“\*t”，则 date 返回一个包含以下字段的表：年、月 （1–12）、日 （1–31）、小时 （0–23）、分钟 （0–59）、秒 （0–61）、wday（工作日，1–7，星期日为 1）、yday（一年中的某天，1–366）和 ISDST（夏令时标志，布尔值）。如果信息不可用，则最后一个字段可能不存在。

如果格式不是“\*t”，则 date 以字符串形式返回日期，根据与 ISO C 函数 strftime 相同的规则进行格式化。

在没有参数的情况下调用时，date 返回合理的日期和时间表示形式，具体取决于主机系统和当前区域设置。（更具体地说， os.date（） 等效于 os.date（“%c”）.）

在非 POSIX 系统中，此函数可能不是线程安全的，因为它依赖于 C 函数 gmtime 和 C 函数本地时间。

os.difftime （t2， t1）

返回从时间 t1 到时间 t2（其中时间是 [os.time](http://www.lua.org/manual/5.3/manual.html#pdf-os.time) 返回的值）的差值（以秒为单位）。在 POSIX、Windows 和其他一些系统中，此值正好是 t2-t1。

os.execute （[命令]）

此功能等效于 ISO C 函数系统。它传递 要由操作系统外壳执行的命令。 如果命令成功终止，则其第一个结果为 **true**，否则为 **nil**。在第一个结果之后，函数返回一个字符串和一个数字，如下所示：

* **“退出”：**命令正常终止;以下数字是命令的退出状态。
* **“**信号**”：**命令被信号终止;以下数字是终止命令的信号。

在没有命令的情况下调用时，os.execute 返回一个布尔值，如果 shell 可用，则返回该布尔值。

os.exit （[代码 [， 关闭]]）

调用 ISO C 函数出口以终止主机程序。如果 code 为 **true**，则返回状态为 EXIT\_SUCCESS;如果 code 为 **false**，则返回状态为 EXIT\_FAILURE;如果 code 为 true，则返回状态 为 该数字。代码的默认值 为 **true**。

如果可选的第二个参数 close 为 true，则在退出之前关闭 Lua 状态。

os.getenv （varname）

返回进程环境变量 varname 的值，如果未定义变量，则返回 **nil**。

os.remove （文件名）

删除具有给定名称的文件（或 POSIX 系统上的空目录）。如果此函数失败，它将返回 **nil**，以及描述错误和错误代码的字符串。否则，它将返回 true。

os.rename （oldname， newname）

将名为 oldname 的文件或目录重命名为 newname。如果此函数失败，它将返回 **nil**，以及描述错误和错误代码的字符串。否则，它将返回 true。

os.setlocale （locale [， category]）

设置程序的当前区域设置。区域设置是指定区域设置的系统相关字符串; category 是描述要更改的类别的可选字符串：“ all”、“collate”、“ctype”、“货币”、“数字”或“time”;默认类别是“all”。 该函数返回新区域设置的名称，如果无法接受请求，则返回 **nil**。

如果区域设置为空字符串，则当前区域设置设置为实现定义的本机区域设置。如果区域设置是字符串“C”，则当前区域设置设置为标准 C 区域设置。

当使用 **nil** 作为第一个参数调用时，此函数仅返回给定类别的当前区域设置的名称。

此函数可能不是线程安全的，因为它依赖于 C 函数 setlocale。

操作系统时间 （[表]）

返回不带参数调用时的当前时间，或表示给定表指定的本地日期和时间的时间。此表必须具有字段年、月和日，并且可能具有字段小时（默认值为 12）、最小值（默认值为 0）、秒（默认值为 0）和 isdst（默认值为 **nil**）。其他字段将被忽略。有关这些字段的说明，请参阅 [os.date](http://www.lua.org/manual/5.3/manual.html#pdf-os.date) 函数。

这些字段中的值不需要在其有效范围内。例如，如果 sec 为 -10，则表示距离其他字段指定的时间 -10 秒;如果小时为 1000，则表示距离其他字段指定的时间 +1000 小时。

返回值是一个数字，其含义取决于您的系统。在 POSIX、Windows 和其他一些系统中，此数字计算自某个给定开始时间（“纪元”）以来的秒数。在其他系统中，没有指定含义，时间返回的数字只能用作 [os.date](http://www.lua.org/manual/5.3/manual.html#pdf-os.date) 和 [os.difftime](http://www.lua.org/manual/5.3/manual.html#pdf-os.difftime) 的参数。

（os.tmpname）

返回一个字符串，其中包含可用于临时文件的文件名。该文件必须在使用前显式打开，并在不再需要时显式删除。

在 POSIX 系统中，此函数还会创建具有该名称的文件，以避免安全风险。（其他人可能会在获取名称和创建文件之间的时间内使用错误的权限创建文件。您仍然必须打开文件才能使用它并删除它（即使您不使用它）。

如果可能，您可能更喜欢使用 [io.tmpfile](http://www.lua.org/manual/5.3/manual.html#pdf-io.tmpfile)，它会在程序结束时自动删除该文件。

6.10 – 调试库

该库为 Lua 程序提供调试接口 （[§4.9](http://www.lua.org/manual/5.3/manual.html#4.9)） 的功能。使用此库时应格外小心。它的几个函数违反了关于Lua代码的基本假设（例如，函数的本地变量不能从外部访问;用户数据元表不能被Lua代码更改;Lua程序不会崩溃），因此可能会损害其他安全代码。此外，此库中的某些函数可能会很慢。

此库中的所有函数都在调试表中提供。所有在线程上运行的函数都有一个可选的第一个参数，该参数是要操作的线程。默认值始终为当前线程。

debug.debug （）

与用户进入交互模式，运行用户输入的每个字符串。使用简单的命令和其他调试工具，用户可以检查全局变量和局部变量，更改其值，计算表达式等。仅包含单词 cont 的行完成此函数，以便调用方继续执行此函数。

请注意，debug.debug 的命令没有词法嵌套在任何函数中，因此无法直接访问局部变量。

debug.gethook （[thread]）

返回线程的当前挂钩设置，作为三个值：当前挂钩函数、当前挂钩掩码和当前挂钩计数（由 [debug.sethook](http://www.lua.org/manual/5.3/manual.html#pdf-debug.sethook) 函数设置）。

debug.getinfo （[thread，] f [， what]）

返回包含有关函数信息的表。你可以直接给出函数，也可以给出一个数字作为 f 的值，这意味着在给定线程的调用堆栈的级别 f 上运行的函数：级别 0 是当前函数（getinfo 本身）;级别 1 是调用 getinfo 的函数（尾部调用除外，它不依赖于堆栈）;等等。如果 f 是一个大于活动函数数的数字，则 getinfo 返回 **nil**。

返回的表可以包含[lua\_getinfo](http://www.lua.org/manual/5.3/manual.html#lua_getinfo)返回的所有字段，字符串 what 描述要填写哪些字段。获取除有效行表之外的所有可用信息的默认值。如果存在，选项“f”会添加一个名为 func 的字段以及函数本身。如果存在，选项“L”将添加一个名为活动行的字段，其中包含有效行的表。

例如，表达式debug.getinfo（1，“n”）.name返回当前函数的名称（如果可以找到合理的名称），表达式debug.getinfo（print）返回一个表，其中包含有关打印函数的所有可用信息 。

debug.getlocal （[thread，] f， local）

此函数返回局部变量的名称和值，并在堆栈的级别 f 处返回函数的局部索引。此函数不仅访问显式局部变量，还访问参数、临时变量等。

第一个参数或局部变量的索引为 1，依此类推，按照它们在代码中声明的顺序，仅计算在函数的当前范围内处于活动状态的变量。负指数是指变量参数;-1 是第一个 vararg 参数。 如果给定索引中没有变量，则该函数返回 nil，并在级别超出范围的情况下调用时引发错误。（您可以致电 [debug.getinfo](http://www.lua.org/manual/5.3/manual.html#pdf-debug.getinfo) 以检查级别是否有效。

以“（”（左括号）开头的变量名称表示没有已知名称的变量（内部变量，如循环控制变量，以及保存在没有调试信息的块中的变量）。

参数 f 也可以是一个函数。在这种情况下，getlocal 仅返回函数参数的名称。

debug.getmetatable （value）

返回给定值的元表，如果没有元表，则返回 **nil**。

debug.getregistry （）

返回注册表表（参见 [§4.5](http://www.lua.org/manual/5.3/manual.html#4.5)）。

debug.getupvalue （f， up）

此函数返回带有函数 f 索引的向上值的名称和值。 如果给定索引没有上升值，则该函数返回 **nil**。

以“（”（左括号）开头的变量名称表示没有已知名称的变量（保存在没有调试信息的块中的变量）。

debug.getuservalue （u）

返回与 u 关联的 Lua 值。如果 u 不是完整的用户数据，则返回 **nil**。

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

将给定函数设置为钩子。字符串掩码和数字计数描述何时调用钩子。字符串掩码可以具有以下字符的任意组合，具有给定的含义：

* **'c'：**每次 Lua 调用函数时都会调用钩子;
* **'r'：**每次 Lua 从函数返回时都会调用钩子;
* **'l'：**每次 Lua 输入新代码行时都会调用钩子。

此外，当计数不同于零时，钩子也会在每次计数指令之后调用。

当在没有参数的情况下调用时，[debug.sethook](http://www.lua.org/manual/5.3/manual.html#pdf-debug.sethook) 会关闭钩子。

当钩子被调用时，它的第一个参数是一个字符串，描述触发其调用的事件：“call ”（或“tail call”）、“return”、“line”和“count”。对于行事件，挂钩还会获取新的行号作为其第二个参数。在钩子内部，你可以调用 级别 2 的 getinfo 来获取有关运行函数的更多信息（级别 0 是 getinfo 函数，级别 1 是钩子函数）。

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

此函数将值值分配给局部变量，并在堆栈级别使用函数的 本地索引。 如果给定索引没有局部变量，则该函数返回 nil，并在级别超出范围的情况下调用时引发错误 。（您可以致电 获取信息以检查级别是否有效。否则，它将返回局部变量的名称。

有关变量索引和名称的更多信息，请参阅 [debug.getlocal](http://www.lua.org/manual/5.3/manual.html#pdf-debug.getlocal)。

debug.setmetatable （value， table）

将给定值的元表设置为给定表 （可以为 **nil**）。返回值。

debug.setupvalue （f， up， value）

此函数将值值分配给具有函数 f 索引的向上值。 如果给定索引没有上升值，则该函数返回 nil。否则，它将返回上行值的名称。

debug.setuservalue （udata， value）

将给定值设置为与给定 udata 关联的 Lua 值。 udata 必须是完整的用户数据。

返回 udata。

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

如果消息存在，但既不是字符串也不是 **nil**，则此函数返回消息而不进行进一步处理。否则，它将返回一个字符串，其中包含调用堆栈的回溯。可选消息字符串追加在回溯的开头。可选的级别号指示在哪个级别开始回溯（默认值为 1，调用回溯的函数）。

debug.upvalueid （f， n）

返回给定函数中编号为 n 的上值的唯一标识符（作为轻用户数据）。

这些唯一标识符允许程序检查不同的闭包是否共享上值。共享上值（即访问相同的外部局部变量）的 Lua 闭包将为这些上值索引返回相同的 ID。

debug.upvaluejoin （f1， n1， f2， n2）

使 Lua 闭包 f1 的第 n1 个上值是指 Lua 闭包 f2 的第 n2 个上值。

7 – Lua 独立

虽然Lua被设计为一种扩展语言，嵌入到宿主C程序中，但它也经常被用作一种独立的语言。Lua作为独立语言的解释器，简称lua，随标准发行版一起提供。独立解释器包括所有标准库，包括调试库。它的用法是：

Lua [选项] [脚本 [参数]]

选项包括：

* **-e *stat*：**执行字符串*统计*;
* **-l mod：“**requires” mod 并将结果分配给全局 @***mod***;
* **-i：**运行脚本后进入交互模式;
* **-v：**打印版本信息;
* **-E：**忽略环境变量;
* **--：**停止处理选项;
* **-：**将 stdin 作为文件执行并停止处理选项。

处理完其选项后，lua 运行给定的*脚本*。当在没有参数的情况下调用时，当 标准输入（stdin）是终端时，lua的行为为lua -v -i，否则表现为lua - 否则。

在没有选项 -E 的情况下调用时，解释器会在运行任何参数之前检查环境变量LUA\_INIT\_5\_3  （如果未  定义版本化名称，则LUA\_INIT）。如果变量内容的格式为 @*filename*，则 lua 将执行该文件。否则，lua 会执行字符串本身。

当使用选项 -E 调用时，除了忽略LUA\_INIT之外，Lua 还会忽略 LUA\_PATH 和 LUA\_CPATH 的值，使用 luaconf.h 中定义的默认路径设置 package.path 和 [package.cpath](http://www.lua.org/manual/5.3/manual.html#pdf-package.cpath) 的值。

所有选项都按顺序处理，但 -i 和 -E 除外。例如，像这样的调用

$ moon -e'a=1' -和 'print（a）' 脚本.lua

将首先将 a 设置为 1，然后打印 a 的值，最后运行文件脚本.lua没有参数。（这里 $ 是外壳提示符。您的提示可能会有所不同。

在运行任何代码之前，lua 会在名为 arg 的全局表中收集所有命令行参数。脚本名称转到索引 0，脚本名称后的第一个参数转到索引 1，依此类推。脚本名称（即解释器名称及其选项）之前的任何参数都将转到负索引。例如，在通话中

$ get -to b.lua t1 t2 t1

表格是这样的：

arg = { [-2] = “take”， [-1] = “-to”，

[0] = “b.lua”，

[1] = “t1”， [2] = “t2” }

如果调用中没有脚本，则解释器名称将转到索引 0，后跟其他参数。例如，调用

$ take -e “print（arg[1]]）”

将打印“-e”。如果有脚本，则使用参数 arg[1]， ···， arg[#arg] 调用脚本。（与Lua中的所有块一样，该脚本被编译为vararg函数。

在交互模式下，Lua 反复提示并等待一行。读完一行后，Lua首先尝试将这行解释为表达式。如果成功，它将打印其值。否则，它将该行解释为语句。如果您编写了不完整的语句，解释器会通过发出不同的提示来等待其完成。

如果全局变量\_PROMPT包含字符串，则其值用作提示。同样，如果全局变量\_PROMPT2包含字符串，则其值将用作辅助提示符（在不完整的语句期间发出）。

如果脚本中出现未受保护的错误，解释器会将错误报告给标准错误流。如果错误对象不是字符串，但具有元方法\_\_tostring，则解释器调用此元方法来生成最终消息。否则，解释器将错误对象转换为字符串，并向其添加堆栈回溯。

正常完成后，解释器关闭其主 Lua 状态（见[lua\_close](http://www.lua.org/manual/5.3/manual.html#lua_close)）。脚本可以通过调用 [os.exit](http://www.lua.org/manual/5.3/manual.html#pdf-os.exit) 来终止此步骤。

为了允许在 Unix 系统中使用 Lua 作为脚本解释器，如果块以 # 开头，则独立解释器会跳过块的第一行。因此，Lua 脚本可以通过使用 chmod +x 和 #！ 形式，如

#！/usr/local/bin/lua

（当然，Lua解释器的位置在您的机器中可能有所不同。如果 lua 在你的路径中，那么

#！/usr/bin/env lua

是一种更便携的解决方案。

8 – 与先前版本不兼容

在这里，我们列出了将程序从Lua 5.2移动到Lua 5.3时可能会发现的不兼容性。您可以通过使用适当的选项编译Lua来避免一些不兼容（请参阅文件luaconf.h）。但是，所有这些兼容性选项都将在将来删除。

Lua 版本始终可以以不暗示程序中源代码更改的方式更改 C API，例如常量的数值或函数作为宏的实现。因此，您不应假设二进制文件在不同的 Lua 版本之间兼容。使用新版本时，始终重新编译 Lua API 的客户端。

同样，Lua 版本总是可以更改预编译块的内部表示形式;预编译块在不同的Lua版本之间不兼容。

官方发行版中的标准路径可能会因版本而异。

8.1 – 语言更改

* Lua 5.2 和 Lua 5.3 之间的主要区别在于引入了数字的整数子类型。尽管此更改不应影响“正常”计算，但某些计算（主要是涉及某种溢出的计算）可能会给出不同的结果。

您可以通过强制数字为浮点数（在 Lua 5.2 中所有数字都是浮点数）来解决这些差异，特别是写入以 .0 结尾的常量或使用 x = x + 0.0 转换变量。（此建议仅用于快速修复偶尔的不兼容;它不是良好编程的一般准则。为了进行良好的编程，请在需要浮点数的地方使用浮点数，在需要整数的地方使用整数。

* 现在，如果结果看起来像整数，则将浮点数转换为字符串会在结果中添加 .0 后缀。（例如，浮点数 2.0 将打印为 2.0，而不是 2.）当您需要特定的数字格式时，应始终使用显式格式。

（从形式上讲，这不是不兼容的，因为Lua没有指定如何将数字格式化为字符串，但某些程序采用特定的格式。

* 已删除垃圾回收器的分代模式。（这是Lua 5.2中的一个实验性功能。

8.2 – 库中的更改

* bit32 库已被弃用。很容易需要一个兼容的外部库，或者更好的是，用适当的按位运算替换其函数。（请记住 bit32 对 32 位整数进行操作，而 Lua 5.3 中的按位运算符对 Lua 整数进行操作，默认情况下有 64 位。
* 表库现在遵循用于设置和获取元素的元方法。
* [ipair](http://www.lua.org/manual/5.3/manual.html#pdf-ipairs) 迭代器现在尊重元方法，其\_\_ipairs元方法已被弃用。
* [io.read](http://www.lua.org/manual/5.3/manual.html#pdf-io.read) 中的选项名称不再有开头的“\*”。为了兼容，Lua 将继续接受（并忽略）此字符。
* 以下函数在数学库中被弃用：atan2、cosh、sinh、tanh、pow 、frexp 和 ldexp。你可以用x^y 替换math.pow（x，y）;你可以用math.atan替换math.atan2，它现在接受一个或两个参数;你可以用 x \* 2.0^exp替换math.ldexp（x，exp）。 对于其他操作，您可以使用外部库或在 Lua 中实现它们。
* [require](http://www.lua.org/manual/5.3/manual.html#pdf-require)使用的C加载程序的搜索器更改了它处理版本化名称的方式。现在，版本应该在模块名称之后（就像大多数其他工具中通常一样）。为了兼容，如果该搜索者无法根据新样式找到打开函数，它仍然会尝试旧格式。（Lua 5.2 已经以这种方式工作，但它没有记录更改。
* 调用收集垃圾（“计数”）现在只返回一个结果。（您可以从第一个结果的小数部分计算第二个结果。

8.3 – API 中的更改

* 延续函数现在接收它们通过lua\_getctx所需的参数，因此lua\_getctx已被删除。相应地调整您的代码。
* 函数[lua\_dump](http://www.lua.org/manual/5.3/manual.html#lua_dump)有一个额外的参数 strip。使用 0 作为此参数的值以获取旧行为。
* 用于注入/投影无符号整数（lua\_pushunsigned、lua\_tounsigned、lua\_tounsignedx、 luaL\_checkunsigned luaL\_optunsigned）的函数已被弃用。将其已签名的等效项与类型强制转换一起使用。
* 投影非默认整数类型（luaL\_checkint、luaL\_optint、luaL\_checklong、luaL\_optlong）的宏已被弃用。在lua\_Integer上使用它们的等效项和 类型强制转换（或者，如果可能，在 代码中使用[lua\_Integer](http://www.lua.org/manual/5.3/manual.html#lua_Integer)）。

9 – Lua 的完整语法

以下是扩展 BNF 中 Lua 的完整语法。与扩展 BNF 中一样，{A} 表示 0 或更多 As，[A] 表示可选的 A.（有关运算符优先级，请参阅 §[3.4.8](http://www.lua.org/manual/5.3/manual.html#3.4.8);有关终端名称、数字和文本字符串的说明，请参阅 [§3.1](http://www.lua.org/manual/5.3/manual.html#3.1)。

块 ：：= 块

块 ：：= {统计} [retstat]

状态 ：：= '**;**' |

变量列表 '**=**' 表达式列表 |

函数调用|

标签|

**中断** |

**转到** 名称|

**做** 块 **端** |

**而** exp **确实阻止结束**|

**重复** 阻止 **直到** exp |

**如果** exp **则**阻止 {**elseif** exp **then** block } [**else** block] **结束**|

**对于**名称 '**=**' exp '，' exp ['**，**' exp] **做**块**结束**|

**对于**  explist **中的**姓名列表**，请**阻止**结束**|

**函数** funcname funcbody |

**本地函数** 名称函数体|

**本地** 名称列表 ['**=**' 表达式列表]

retstat ：：= **返回** [explist] ['**;**']

标签 ：：= “：：” 名称 “：**：**”

函数名称 ：：= 名称 {'**.**' 名称} ['**：**' 名称]

varlist ：：= var {'**，**' var}

var ：：= 名称|前缀EXP '**[**' exp '**]**' | prefixexp '**.**' 名称

名称列表 ：：= 名称 {'**，**' 名称}

explist ：：= exp {'**，**' exp}

exp ：：= **无** | **假** | **真** |数字|文字字符串|'**......**' | functiondef |

前缀EXP|表构造函数|exp binop exp |项目厅 exp

前缀exp ：：= var |函数调用|'**（**' exp '**）**'

函数调用 ：：= 前缀EXP 参数 |前缀exp '**：**' 名称参数

args ：：= '**（**' [explist] '**）**' | 表构造函数|文本字符串

函数定义 ：：= **函数** 函数体

函数体 ：：= '**（**' [parlist] '**）**' 块**结束**

帕里斯特 ：：= 名称列表 ['**，**' '**...**'] |'**......**'

表构造函数 ：：= '**{**' [字段列表] '**}**'

字段列表 ：：= 字段 {字段EP 字段} [字段EP]

字段 ：：= '**[**' exp '**]**' '**=**' exp |名称 '**=**' exp | exp

菲尔塞普 ：：= '**，**' |'**;**'

二进制 ：：= '**+**' |“**-**”|“**\***”|“**/**”|“/**/”**|“**^**” |“**%**” |

“**&**”|“**~**”|'**|**' |“**>>**”|“**<<**”|'.**.**' |

“**<**”|“**<=**”|“**>**”|“**>=**”|'**==**' |'**~=**' |

**和** | **或**

项目厅 ：：= '**-**' | **不**|“**#**”|'**~**'

最后更新： 周二 7月 14 10：32：39 UTC 2020