# Lua 5.3 中文指南

翻译:fangcun

2018 年 6 月 14 日

# 目录

**1 概述** **5**

**2 基础概念** **5**

2.1 值和类型 . . . . . 5
2.2 环境和全局环境 . . . . . 7
2.3 错误处理 . . . . . 7
2.4 元表和元方法 . . . . . 8
2.5 垃圾回收 . . . . . 11
2.5.1 垃圾回收元方法 . . . . . 12
2.5.2 弱表 . . . . . 13
2.6 协程 . . . . . 13

**3 Lua 语言** **15**

3.1 词法 . . . . . 15
3.2 变量 . . . . . 18
3.3 语句 . . . . . 18
3.3.1 语句块 . . . . . 18
3.3.2 程序块 . . . . . 19
3.3.3 赋值 . . . . . 19
3.3.4 控制结构 . . . . . 20
3.3.5 For 语句 . . . . . 21
3.3.6 函数调用作为语句 . . . . . 23
3.3.7 局部定义 . . . . . 23
3.4 表达式 . . . . . 23
3.4.1 算术操作符 . . . . . 25
3.4.2 位运算符 . . . . . 25
3.4.3 强制类型转换 . . . . . 26
3.4.4 关系运算符 . . . . . 26
3.4.5 逻辑运算符 . . . . . 27
3.4.6 连接符 . . . . . 28
3.4.7 求长度操作符 . . . . . 28
3.4.8 操作符优先级 . . . . . 28

3.4.9 表构造器 . . . . . 29
3.4.10 函数调用 . . . . . 30
3.4.11 函数定义 . . . . . 31
3.5 可见性 . . . . . 32

**4 应用程序接口 33**
4.1 栈 . . . . . 34
4.2 栈大小 . . . . . 34
4.3 有效和可接受的索引 . . . . . 35
4.4 C 闭包 . . . . . 35
4.5 注册表 . . . . . 35
4.6 在 C 语言中进行错误处理 . . . . . 36
4.7 在 C 语言中进行协程处理 . . . . . 36
4.8 函数和类型 . . . . . 36
4.9 调试接口 . . . . . 38

**5 辅助库 38**
5.1 函数和类型 . . . . . 38

**6 标准库 38**
6.1 基础函数 . . . . . 38
6.2 协程处理 . . . . . 38
6.3 模块 . . . . . 38
6.4 字符串处理 . . . . . 38
6.4.1 模式 . . . . . 38
6.4.2 Format Strings for Pack and Unpack . . . . . 38
6.5 UTF-8 支持 . . . . . 38
6.6 表处理 . . . . . 38
6.7 数学函数 . . . . . 38
6.8 Input and Output Facilities . . . . . 38
6.9 Operating System Facilities . . . . . 38
6.10 调试库 . . . . . 38

**7 Lua Standalone 38**

**8 向前兼容** **38**
8.1 语言变化 . . . . . 38
8.2 库变化 . . . . . 38
8.3 API 变化 . . . . . 38

**9 Lua 完整语法定义** **38**

# 1 概述

**Lua** 是一个强大、高效、轻量的嵌入式脚本语言。它不仅支持过程化、面向对象、函数式和数据驱动的编程方式，还拥有强大的数据描述能力。

**Lua** 不仅语法简单，还拥有基于**关联数组**和**可扩展语义**的强大数据描述能力。**Lua** 是动态类型的，它对代码的执行是依靠一个基于寄存器的虚拟机执行编译器生成的字节码来完成的。它还拥有增量内存垃圾自动回收的能力。这些能力使 **Lua** 成为 **配置文件**、**脚本语言**和**快速原型开发**的理想选择。

**Lua** 是以 *clean C*[1]的方式实现的。**Lua** 发布版包含一个叫做 *lua* 的程序，它可以以交互式或批处理的方式处理 **Lua 代码**。**Lua** 的设计倾向于嵌入到其它程序中去，为它们提供扩展的能力。但将 **Lua** 独立使用也非常不错。

作为一个提供扩展能力的语言，**Lua** 没有 *main* 函数的概念：**Lua** 通常被嵌入到其它程序，假设这个被嵌入的程序是 *host*。那么，*host* 程序就可以执行 **Lua** 代码，读写 **Lua** 变量，注册可以被 **Lua** 代码调用的 **C 函数**。这种和 **C 语言**交互的能力使得 **Lua** 具有广泛的应用领域。

**Lua** 是一个自由软件，它不提供任何担保。关于它的实现描述可以在 **www.lua.org** 找到。

像其它参考手册一样，这份文档是比较枯燥的。关于 **Lua** 在设计上的讨论可以在 **Lua** 网站上找到。对于使用 **Lua** 进行编程可以参考 *Roberto* 的 *Programming in Lua*。

# 2 基础概念

本章节描述了 **Lua** 语言的基础概念。

## 2.1 值和类型

**Lua** 是动态类型语言。它的变量没有类型的概念，而值有类型。

在 **Lua** 中所有不同类型的值都是一样的。它们都可以被保存在变量中，被作为参数传递函数，被函数返回作为结果。

[1] 标准 C 和 C++ 的共同部分。

在 **Lua** 中有 8 种基本类型：*nil*，*boolean*，*number*，*string*，*function*，*userdata*，*thread* 和 *table*。*nil* 类型只有一个值：**nil**，它通常用来表示什么都没有。*boolean* 有两个值：**false** 和 **true**。**nil** 和 **false** 使条件的结果为假，其它值使条件的结果为真。*number* 表示整数和浮点数。*string* 表示不可变的字节序列。**Lua** 的 *string* 可以包含任何 8 位的数据，包括 *\0*。此外，**Lua** 是编码无关的。

*number* 类型有两种内部表示方式，一种叫做 *integer*，另一种叫做 *float*。对于每种类型何时使用具有明确的规则**章节 3.4.3**。标准的 **Lua** 使用 **64 位整型**和**双精度浮点数**。我们可以通过 **LUA_32BITS 宏**来编译生成 **32 位**的 **Lua** [2]。

**Lua** 可以调用用 **Lua** 和 **C 语言**编写的代码 (章节 3.4.10)。并且它们都被视为 *function* 类型。

*userdata* 类型提供了 **C 语言**数据存储在 **Lua** 变量的方法。*userdata* 的值是一个**数据内存块**。有两种类型的 *userdata*:*full userdata* 和 *light userdata*。*full userdata* 是一个被 **Lua** 管理的数据内存块对象。*light userdata* 则是一个 **C 语言**指针。在 **Lua** 中 *userdata* 除了**赋值**和**判等**操作外，没有其它预定义的运算。通过**元表**，我们可以定义 *userdata* 的运算（章节 2.4）。为了保证**宿主程序**数据的完整性，我们只能通过 **C API** 创建和修改 *userdata* 类型的值。

*thread* 类型用来表示独立的 textbf 线程（章节 2.6）。**Lua** 的线程并不是基于操作系统。在所有操作系统上，都可以使用 **Lua** 的**协程**。

*table* 类型被用来实现**关联数组**。它可以使用除了 **nul** 和 **NaN**[3]的所有值作为 *key* 查询数据。*table* 可以存放除了 **nil** 外的所有类型的值。*key* 对应 **nil** 值被认为这个 *key* 在 *table* 中不存在。也就是说如果一个 *key* 在 *table* 中**存在**，它对应的值一定不是 **nil**。

*table* 在 **Lua** 中扮演这极为重要的角色，它可以用来表示数组，列表，记录，图，树等等。在 **Lua** 中有多种不同的方法来创建 *table*（章节 3.4.9）。

两个索引等价的判断依据是它们二进制表示是否相同。$a[i]$ 和 $a[j]$ 在 $i$ 和 $j$ 的二进制表示相同的情况下表示同一表元素。特别的，拥有整数数值的浮点数和相应的整数等价[4]。为了避免歧义，任何浮点数具有整数值在被作为 *key* 都会先被转换为整型表示。比如，$a[2.0] = true$，实际上在表中插

[2] **LUA_32BITS 宏**位于 *luaconf.h* 文件中。
[3] NaN 用来表示未定义或不可表示的数据，比如 0/0。
[4] 比如 1.0 == 1。

入了一个 *key* 为 2 的元素 [5]。

*table*、*function*、*thread* 和 *(full) userdata)* 的值都是对象。变量仅仅存放了它们的**引用**，而不是实际存放它们本身。所以在对这些值类型进行赋值，参数传递，函数返回时并没有进行对数据本身的复制。

**Lua** 的库函数 *type* 可以获取一个值的类型的字符串描述（章节 6.1）。

## 2.2 环境和全局环境

就像我们在章节 3.2 和章节 3.3.3 讨论的，标识符 *var* 被解释为 *_ENV.var*。每个程序块都被编译到一个叫做 *_ENV* 的外部局部变量作用域下。因此 *_ENV* 在一个程序块中不能作为标识符来使用。

尽管 *_ENV* 是一个外部变量，但我们仍然可以用这个标识符定义变量和参数。新定义的变量和参数作用域规则符合章节 3.5。

任何被 *_ENV* 作为值的 *table* 被叫做**环境**。

**Lua** 会维护一个**全局环境**。它被保持在 **C 注册表**中的一个特殊索引中（章节 4.5）。全局变量 *_G* 使用这个值进行初始化[6]。

当 **Lua** 载入程序块时，它的 *_ENV* 上值缺省是全局环境 [7]。也就是说，缺省情况下，标识符是在全局环境下的[8]。此外，所有标注库也被装载在全局环境下，并且一些函数依靠这个环境运作。我们可以使用 *load* 或 *loadfile* 使用不同环境载入程序块 [9]。

## 2.3 错误处理

**Lua** 作为嵌入式脚本语言，通常被宿主程序通过调用 **Lua** 库函数来启动执行 **Lua** 代码[10]。当有编译错误和运行错误发生时，返回错误信息给宿主程序。

**Lua** 代码可以通过调用 *error* 函数来显式地产生错误。如果需要在 **Lua** 中捕捉错误，可以在 *protected* 模式下调用 *pcall* 或 *xpcall* 传递进行错误处理的函数。

[5] 需要注意 2 和"2" 是不同的 *key*。
[6] _G 并没有在内部使用。
[7] 参考 *load* 函数
[8] 所以这些标识符也被叫做全局变量。
[9] 使用 C 语言，我们必须先载入程序块，然后改变它的第一个上值。
[10] 如果独立使用 Lua，那么 lua 就是这个宿主程序。

当错误产生时会生成一个 *error* 对象，这个对象包含了错误的一些信息。**Lua** 本身产生的错误生成的 *error* 对象是一个字符串，但具体的 **Lua** 程序可能使用任意值作为 *error* 对象。*error* 对象会被向上传递直到被 **Lua** 程序或宿主程序处理。

可以通过调用 *xpcall* 或 *lua_pcall* 来指定一个**错误处理函数**。当错误产生时，*error* 对象被作为参数传递给该函数，该函数被调用，结束后返回新的 *error* 对象。**错误处理函数**在栈帧未解开时就被调用，我们可以通过追踪调用栈获得更多的错误信息。**错误处理函数**内部产生的错误消息也会调用这个 **错误处理函数**，当长时间陷入这样的循环，**Lua** 会打破循环，然后返回一个消息[11]。

## 2.4 元表和元方法

**Lua** 允许任意值有一个**元表**。**元表**是一个普通的 **Lua** 表。它存放了针对这个值在特定情况下所进行的操作入口。我们可以通过改变**元表**中的特定元素，改变附加在这个值上的一些操作。比如，当一个非数字的值被作为加号的操作数时，**Lua** 会检查这个值 **元表**中的 *___add* 元素。找到后，**Lua** 调用这个函数执行加法操作。

在**元表**中以**双下划线**开始的字符串类型的 *key* 对应的值叫做**元方法**。在前面的例子中，*___add* 对应的值就是具体执行加法操作的函数。

可以通过调用 *getmetatable* 函数查询**元表**的数据。**Lua** 使用原始数据查询**元方法**。下面的代码显示了这一过程：

```
rawget(getmetatable(o) or {},"___ev")
```

可以通过 *setmetatable* 函数替换 *table* 的**元表**。但不可以使用 **Lua** 代码替换其它值类型的**元表** [12]。可以通过 **C API** 实现对其它值类型元表的替换。

每个 *table* 和 *userdata* 可以拥有独立的**元表** [13]。其它值类型每个类型共享一个**元表**。简单说，*number* 类型值共享一个**元表**，*string* 类型值共享一个**元表**，等等。缺省情况下，除了 *string* 值外，其它值是没有**元表**的（章节 6.4）。

[11]错误处理函数只有在发生运行时错误才被调用。它不会在内存分配和 *finalizer* 发生错误时调用。
[12]除了使用调试库（章节 6.10）。
[13]尽管多个 *table* 和 *userdata* 可能共享一个 **元表**

**元表**定义了附加在对象上的算术运算，位运算，比较运算，连接运算，长度运算，调用以及索引方法。**元表**可以为 *userdata* 或 *table* 指定一个在对象被垃圾回收前调用的函数（章节 2.5）。

对于一元运算（否定，求长度，和按位取反），**元方法**提供了一个伪的第二操作数，它的值等于第一操作数。这个设计是为了简化 **Lua** 的实现，在未来的版本可能会删除，因此应该不要使用这个伪的第二操作数。

下面给出了由**元表**控制的事件的详细列表。每个操作通过 *key* 来区分。

- ___add:(+) 加法操作。任何一个操作数不是 *number* 类型且不能被转换为 *number* 类型，**Lua** 将会调用元方法。首先，**Lua** 会检查第一个操作数，如果它没有定义 ___*add* 元方法，**Lua** 会检查第二个操作数。最后，如果 **Lua** 能找到一个**元方法**就会以这两个操作数为参数调用这个**元方法**。**元方法**返回的结果被作为运算符操作后的结果 [14]。如果找不到 **元方法**就会产生一个错误。
- ___sub:(-) 减法操作。执行过程类似加法操作。
- ___mul:(*) 乘法操作。执行过程类似加法操作。
- ___div:(/) 除法操作。执行过程类似加法操作。
- ___mod:(%) 模操作。执行过程类似加法操作。
- ___pow:( ^) 指数操作。执行过程类似加法操作。
- ___unm:(-) 求相反数操作。执行过程类似加法操作。
- ___idiv:(//) 整型除法。执行过程类似加法操作。
- ___band:(&) 按位与操作。如果有操作数既不是整型数也不能被强制转换为整型数，**Lua** 将会尝试调用元方法（章节 3.4.3）。其余执行过程类似加法操作。
- ___bor:(|) 按位或操作。执行过程类似按位与操作。
- ___bxor:( ~) 按位异或操作。执行过程类似按位与操作。
- ___bnot:( ~) 按位取反操作。执行过程类似按位与操作。

[14] 结果被调整为一个值。

- ___shl:(«) 按位左移操作。执行过程类似按位与操作。
- ___shr:(») 按位右移操作。执行过程类似按位与操作。
- ___concat:(..) 连接操作。如果任一操作数既不是字符串也不是数字[15]，**Lua** 将会尝试调用元方法。其余执行过程类似加法操作。
- ___len:(#) 求长度操作。如果这个对象不是一个字符床，**Lua** 将会尝试调用元方法。如果元方法存在，**Lua** 会把这个对象作为参数调用这个元方法，然后返回这个函数返回的值。如果元方法不存在，但这个对象是一个 *table*，那么 **Lua** 就会使用的 *table* 的求长度操作（章节 3.4.7）。其余情况，**Lua** 产生一个错误。
- ___eq:(==) 判等操作。它的执行过程和加法操作类似。除了比较的两个值都是 *table* 或 *full userdata*，并且它们引用相同的对象时，**Lua** 将会调用元方法。调用后的返回值被转换为一个布尔量。
- ___lt:(<) 判小于操作。它的执行过程和加法操作类似。除了比较的两个值不都是数字或不都是字符串时，**Lua** 将会调用元方法。调用后的返回值被转换为一个布尔量。
- ___le:(<=) 判小于等于操作。和其它操作不同，小于等于操作可以使用两个不同的事件。首先，就像小于操作一样，**Lua** 查看两个操作数的 ___*le* 元方法。如果找不同，**Lua** 尝试调用 ___*lt* 元方法，因为 $a <= b$ 等价于 $not(b < a)$。像其它比较操作一样，它的结果也是一个布尔量 [16]。
- ___index:([*key*]) 索引操作。这个事件在 *table* 不是一个表或 *key* 不在表中时发生。**Lua** 在表中查询这个**元方法**。

  这个**元方法**既可以是一个函数也可以是一个表。如果它是一个函数，*table* 和 *key* 会作为它的参数，然后这个函数调用的返回结果被调整为一个值后被作为操作的结果。如果它是一个表，结果就是用这个表索引 *key* 后得到的值 [17]。

[15] 数字会被强制转换为字符串。
[16] ___*lt* 事件在这里的使用可能在未来版本剔除。并且它要比直接使用 ___*le* 元方法慢。
[17] 这个索引过程是正常的索引，因此可能会触发其它的元方法。

- ___newindex:$table[key] = value$。和索引事件一样，这个事件当 *table* 不是一个 *table* 或 *key* 不在表中时触发。**Lua** 在表中查询这个**元方法**。

  和索引类似，这个元方法可以是一个函数或着一个表。如果它是一个函数，**Lua** 会把 *table* 和 *key* 作为它的参数调用它。如果它是一个表，**Lua** 会使用这个键对对这个表进行赋值 [18]。

  不管 ___*newindex* 元方法是否存在，**Lua** 都不会执行本原赋值[19]。

- ___call:*func(argc)*。当 **Lua** 尝试去调用一个非函数值时会触发这个事件。元方法在 $func$ 中查找。如果存在，这个元方法会以 $func$ 作为第一参数，后跟其它参数的方式进行调用。函数返回的结果直接作为操作的结果[20]。

在设置一个对象的**元表**前将所有需要使用的元方法添加其中是一个很好的习惯。特别的，___*gc* 元方法只在这种情况下才生效 (章节 2.5.1)。

元表同时也是一个正常的 *table*，可以包含任意的域，不仅仅只有上面描述的这些。一些标准库的函数[21]会使用元表的其它一些域来实现自己的目的。

## 2.5 垃圾回收

**Lua** 自动进行内存管理。我们不需要关心内存的申请和释放。当一个对象不能被 **Lua** 代码访问到时，它就可能被垃圾回收器释放掉。*string*, *table*,*userdata*,*function*, *thread*，内部结构等等被 **Lua** 使用的内存都被自动管理。

**Lua** 使用增量标记扫描的方式回收内存。它通过两个数字控制垃圾回收的周期：*garbage-collector pause* 和 *garbage-collector step multiplier*。它们都使用百分数作为作为单位[22]。

*garbage-collector pause* 控制等待多久之后开始新一轮的垃圾回收。它的数字越大，垃圾回收越不激进。它的值小于 100 将不会等待就开始新一轮的回收。它的值等于 200 意味着只有内存使用达到上一次回收后的两倍时开始新一轮的垃圾回收。

[18] 这个赋值过程是正常的赋值，因此可能会触发其它元方法。
[19] 如果必要，元方法可以调用 *rawset* 进行赋值。
[20] 这是唯一一个允许多个结果的元方法。
[21] 比如 *tostring* 函数。
[22] 100 相当与 1。

*garbage-collector step multiplier* 控制垃圾回收相对于内存分配的速度。它的值越大，垃圾回收越激进，但同时增加了步伐的增量。不应该给它小于 100 的值，因为这样会造成回收过慢以至于无法结束一个回收周期。它的缺省值是 200, 意味着垃圾回收的速度是内存分配的两倍。

如果把 *garbage-collector step multiplier* 设置为一个很大的值[23]，回收器的表现就像 *stop-the-world* 回收器一样。如果再设置 *garbage-collector pause* 为 200, 回收器表现的就像旧版本的 **Lua** 一样，当 **Lua** 使用双倍内存时进行垃圾回收。

可以通过 **C API** 函数 *lua_gc* 或使用 **Lua** 代码调用 *collectgarbage* 函数设置这些数字。我们也可以使用函数直接控制回收[24]。

### 2.5.1 垃圾回收元方法

我们可以通过 **Lua** 代码设置 *table* 的垃圾回收 **元方法**。对于 *userdata* 可以通过 **C API** 来设置（章节 2.4）。这些元方法也被叫做 *finalizer*。*finalizer* 使 **Lua** 的垃圾回收器拥有和外部资源协调的能力 [25]。

为了使一个对象再被回收时调用 *finalizer*，我们需要设置它的 **元表**时同时设置 *__gc* 域。如果没有同时设置这个域，或是之后在**元表**中创建这个域，对象的 *finalizer* 都不会被执行。

当一个被标记的对象变成垃圾，它并不会立即被回收。而是被 **Lua** 放入一个表中。**Lua** 完成所有收集操作后，会遍历这个表，检查每个对象的 *__gc* 元方法。然后将这个对象作为这个元方法的参数调用 *__gc* 元方法。

在每轮回收周期结尾，被标记的对象以相反的顺序调用 *finalizer*。*finalizer* 执行可能在正常代码执行的任一点发生。

被收集的对象可能需要被 *finalizer* 使用，**Lua** 可以恢复这些对象。通常，这种恢复是暂时的，在下一周期对象就会被释放。但是，如果 *finalizer* 把对象存放在一个全局可以访问的地方，这种恢复就是永久的。此外，如果一个 *finalizer* 标记一个正在结束的对象再次终结，它的 *finalizer* 会在下一周期再次被调用，但这时对象本身已经不可以被访问到。任何情况下，不可访问且未被标记的对象的内存在一个回收周期被释放。

当我们关闭状态时[26]，**Lua** 就会按照相反的顺序调用所有标记析构的

[23] 超过程序最大使用内存百分之十。
[24] 比如暂停和重启回收器。
[25] 比如关闭文件，网络和数据库连接。
[26] *lua_close*。

对象的 *finalizer*。在这时，任何 *finalizer* 标记回收对象都是无效的。

### 2.5.2 弱表

**弱表**是指表中元素是弱引用的 *table*。弱引用被垃圾回收器忽略。如果一个对象只被弱引用，垃圾回收器就会回收它。

**弱表**可以包含弱引用的 *key*，弱引用的值或者两者都有。值是弱引用的**表**允许回收它的值，但不能回收它的 *key*。*key* 和值都是弱引用的**表**允许回收它的 *key* 和值。在任何情况下，*key* 和值其中有一个被回收，整个键对就会被表中移除。表的 *___mode* 域控制了表的键值是否是弱引用的，如果 *___mode* 域包含了 *k* 字符，*key* 是弱引用的。如果 *___mode* 域包含了 *v* 字符，**值**是弱引用的。

如果一个 *table* 具有弱引用的 *key* 和强引用的**值**，那么它就是**蜉蝣表**，它的值只有在 *key* 可以被访问时才能被访问到。特别的，如果一个 *key* 的唯一引用是通过它的值，这个键对会被表移除。

表的强弱信息的改变在下一个回收周期生效。所以，尽管我们把表改为强引用模式，但是 **Lua** 在改变生效前仍会回收表中的元素。

只有显式构建的对象才会被弱表移除。数字和轻量的 **C** 函数并不被垃圾回收管理，它们自然也就不会被弱表移除 [27]。字符串接受垃圾回收器的管理，但它没有显式的构造，所以也不会被弱表移除。

弱表中的恢复对象有一些特殊的行为。它们在运行自己的 *finalizer* 前弱值会被删除，但只有在运行 *finalizer* 的下一回收周期，才删除 *key*。这样 *finalizer* 就可以通过弱表访问对象关联的信息。

一个弱表在一个回收周期是一个恢复对象，它可能需要在下一周期才能被清除。

## 2.6 协程

**Lua** 支持**协程**, 也叫做**协作多线程**。**协程**在 **Lua** 中代表独立的线程。和操作系统的线程不同，**协程**只有在显式调用 *yield* 函数后才会挂起执行。

可以通过调用 *coroutine.create* 来创建一个协程。它的参数是协程要执行的函数。*coroutine.create* 创建一个协程，被返回这个协程的句柄，但它不会开始**协程**的执行。

[27] 除非它们的值被回收

我们可以通过调用 *coroutine.resume* 来执行协程。当第一次调用 *coroutine.resume* 函数，协程开始运行它的执行函数。*coroutine.resume* 函数的可变参数被传递给协程的执行函数作为参数。**协程**开始执行之后，遇到 *yield* 或函数结束才会终止。

**协程**有两种方式终止运行，一种是从它的执行函数正常返回。另一种出现异常错误。在正常返回的情况下，*coroutine.resume* 函数会返回 **true**，加上其它被协程函数返回的值。在出错的情况下，*coroutine.resume* 函数返回 **false** ，加上一个错误对象。

可以通过调用 *coroutine.yield* 函数让出**协程**的执行。当一个**协程**被让出，*coroutine.resume* 函数会立即返回，即使这个 *yield* 函数是在执行函数内的一个嵌套函数中调用的。调用 *yield* 函数让出协程，*coroutine.resume* 同样返回 **true**，加上传递给 *coroutine.yield* 的值。下一次使用 *coroutine.resume* 唤醒同一个协程时，这个协程会从让出的地方开始执行。调用 *coroutine.yield* 返回的任何多余参数都会被传递给 *coroutine.resume*。

*coroutine.wrap* 函数被用来创建一个**协程**，但它不返回协程的句柄，而是返回一个函数，当这个函数被调用，协程就会被执行。传递给这个函数的参数就像传递给 *coroutine.resume* 函数的参数一样。*coroutine.wrap* 函数返回 *coroutine.resume* 函数除了第一个布尔错误代码外的所有返回值。*coroutine.wrap* 函数不会捕捉错误，调用者需要自己处理错误。

下面的代码演示了协程如何工作：

```
function foo (a)
        print("foo", a)
        return coroutine.yield(2*a)
end

co = coroutine.create(function (a,b)
                print("co-body", a, b)
                local r = foo(a+1)
                print("co-body", r)
                local r, s = coroutine.yield(
                                a+b,a-b)
                print("co-body", r, s)
                return b, "end"
```

```
end)

print("main", coroutine.resume(co, 1, 10))
print("main", coroutine.resume(co, "r"))
print("main", coroutine.resume(co, "x", "y"))
print("main", coroutine.resume(co, "x", "y"))
```

运行上面的代码，会产生下面这些输出：

```
co-body 1       10
foo     2
main    true    4
co-body r
main    true    11      -9
co-body x       y
main    true    10      end
main    false   cannot resume dead coroutine
```

我们也可以通过 **C API** 创建和处理协程，具体参考 *lua_newthread*，*lua_resume* 和 *lua_yield* 函数。

# 3 Lua 语言

本章描述了 **Lua 语言**的词法，语法和语义。

这里使用巴科斯范式描述 **Lua 语言**的构造。

## 3.1 词法

**Lua** 对于代码格式要求很自由。它忽略标记之间的空白符 (包括换行) 和注释，仅仅将它们作为标记之间的分割。

在 **Lua** 中，任何不以数字开始的字符串可以作为标识符。标识符可以用来给变量，表域和标签命名。

下面的关键字被 **Lua** 保留，不能作为标识符： *and break do else elseif end false for function goto if in local nil not or repeat return then true until while*

**Lua** 是一个大小写敏感的语言：*and* 是一个保留字，但 *And* 和 *AND* 不是。作为约定，我们应该避免使用以一个下划线开始后接一个或多个大写字母的标识符 [28]。

下面的符号有其特殊的意义：

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

可以使用单引号或双引号来定义短字符串字面量。字面量可以包含下面这些 **C 语言**转义字符：

'\a'（响铃）
'\b'（退个退格）
'\f'（换页）
'\n'（换行）
'\r'（回车）
'\t'（水平制表）
'\v'（垂直制表）
'\\'（反斜杠）
'\"'（双引号）
'\''（单引号）

反斜杠后跟换行会在字符串中插入新行。$\z$ 可以跳过紧接着的空白符，包括换行。可以利用 $\z$ 来更方便的组织字符串。短字符串字面量不能包括未转移的换行和非法转义字符。

我们可以精确地指定字符串的内容，甚至可以像字符串中插入 ASCII 为 0 的字符．要进行这样的操作只需要使用 $\xXX$ 转义即可，这里的 $XX$ 使用两个十六进制数字替换。还可以使用
*ddd* 的形式，这里的 *ddd* 是三个十进制的数字。

可以使用 $\u\{XXX\}$ 插入 UTF-8 编码的字符到字符串，$XXX$ 是字符代码的十六进制表示。

字符串字面量也可以使用长括号的格式。我们定义 $n$ 级开长括号是指

[28] 比如 **_VERSION**

一个 [括号后跟 $n$ 个 =，然后跟着另一个 [括号。0 级开长括号写作 [[，1 级开长括号写作 [=[，以此类推。闭长括号的定义与之类似。举个例子，4 级闭长括号写作]====]。长字符串字面量以一个级别的开长括号开始，然后结束于同级的闭长括号。开闭长括号之间可以包含任何文本，包括换行。

为了方便，[后紧跟新行，这个新行不会被包含在字符串中。举个例子，在一个使用 **ASCII** 编码的系统中，下面的 $a$ 变量被赋予完全相同的字符串常量：

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

任何在字符串字面量中的字符如果不受上面规则的影响，它们代表自己。**Lua** 使用文本模式打开文件。系统的文件函数可能会对一些控制字符进行处理。所以，对于一个非文本数据最好使用显式的转义表示。

数字常量可以用指数形式表示。也可以使用十六进制表示。数字常量带有小数点或幂被认为是一个浮点数。否则，如果它的值符合整型，它就是一个整型数。下面是合法的整型数的例子：

```
3   345   0xff   0xBEBADA
```

下面是合法的浮点数的例子：

```
3.0     3.1416   314.16e-2
0.31416E1        34e1
0x0.1E   0xA23p-4
0X1.921FB54442D18P+1
```

**Lua** 的注释以 -- 开始。如果 -- 后没有紧跟，这个注释就是一个行注释。否则，它是一个长注释，直到括号关闭，注释才结束。长注释通常被用来临时取消代码。

## 3.2 变量

**变量**是存放**值**的地方。**Lua** 有三种类型的变量：全局变量，局部变量和表域。

可以用**标识符**表示一个全局变量或局部变量。

```
var::=Name
```

默认情况下变量被假定为全局变量，除非显式地使用 *local* 定义。局部变量的作用范围由语义限定。

在变量未被赋值之前，它的值为 **nil**。

中括号被用来索引查询 *table*：

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

访问表域的行为可以通过**元表**更改。$t[i]$ 等价于调用 *gettable_event(t,i)* 函数 [29]。

*var.name* 等价于 *var*["*name*"]：

```
var ::= prefixexp '.' Name
```

对于全局变量 $x$ 的访问等价于 $\_ENV.x$。根据程序块被编译的方式，$\_ENV$ 不能作为一个全局标识符 (章节 2.2)。

## 3.3 语句

**Lua** 支持几乎所有的常用语句。形式类似于 Pascal 和 C。包括赋值，控制，函数调用和变量定义。

### 3.3.1 语句块

一个**语句块**是指一段连续的语句：

```
block ::= {stat}
```

**Lua** 支持空语句，可以用分号分割语句：

```
stat ::= ';'
```

[29] 章节 2.4 有关于 *gettable_event* 函数的完整描述。*gettable_event* 函数并没有被定义，使用它仅仅是出于解释的目的。

函数调用和赋值可以以括号开始。这可能会造成语法的歧义。考虑下面的代码：

```
a = b + c
(print or io.write)('done')
```

在语法上可以看作以下两种形式。

```
a = b + c(print or io.write)('done')
a = b + c; (print or io.write)('done')
```

目前，**Lua** 目前将其识别为第一种形式。、为了避免语法的二义性，最好是使用分号来分割语句。

```
;(print or io.write)('done')
```

一个语句块也是一条语句，语法形式如下：

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

显式语句块可以用来控制变量的作用域。除此之外，显式语句块也被用来在语句块的内部使用 *return* 返回结果。

### 3.3.2 程序块

**Lua** 的编译单位是**程序块**。语法构造上，程序块就是一个语句块。

```
chunk ::= block
```

**Lua** 将**程序块**作为一个带有可变参数的匿名函数进行处理。所以，程序块可以有自己的局部变量，接收参数，返回数据。

程序块可以被存放在文件中或者宿主程序的一个字符串中。**Lua** 把程序块编译成虚拟机指令代码，然后运行。

程序块也可以提前被编译为二进制的形式。关于这方面的内容可以了解 *luac* 程序和 *string.dump* 函数。**Lua** 可以自动判断程序块的形式进行执行（参看 *load* 函数）。

### 3.3.3 赋值

**Lua** 允许多重赋值。也就是等号左边的列表里的变量和等号右边列表中的表达式一对一对应赋值。

```
stat ::= varlist '=' explist
varlist ::= var { ',' var}
explist ::= exp { ',' exp}
```

**表达式**在章节 3.4 中被讨论。

### 3.3.4 控制结构

*if,while,repeat* 有相似的语法形式：

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

**Lua** 也有 *for* 语句（章节 3.3.5）。

控制结构允许任何值作为条件。**false** 和 **nil** 被认为是 **假**，其余情况被认为是**真**[30]。

*repeat until* 使用的循环条件可以是它包含的语句块内部定义的 **局部变量**。

*goto* 语句用来跳转到一个标签执行。标签的语法形式如下：

```
stat ::= goto Name
stat ::= label
label ::= '::' Name '::'
```

标签在定义它的整个语句块都是可见的，但如果语句块内嵌的语句块或函数定义了同名标签，那么它就会被覆盖。*goto* 可以跳到任何一个不进入局部变量作用域的可见标签。

标签和空语句被叫做 *void statements*，它们不执行任何动作。

*break* 语句被用来终止 *while*，*repeat* 和 *loop* 循环。

```
stat ::= break
```

*break* 仅仅终止包含它的那一层循环。

*return* 语句用来返回函数或程序块 [31]的结果。函数可以返回一个以上的返回值。*return* 语句的形式如下：

[30]特别注意，数字 0 和空字符串也被认为是**真**。

[31]程序块本身是一个匿名函数。

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

*return* 语句只能是语句块的最后一条语句。如果确实需要在语句块中间使用，可以显式地使用一个内部语句块包括 *return*，使它变成语句块的最后一条语句。就像 *do return end* 这样。

### 3.3.5 For 语句

*for* 语句有两种形式：基于数字和基于种类。
基于数字的 *for* 循环语法形式如下：

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

我们以下面这条语句对基于数字的 *for* 循环进行解释：

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

它等价于下面的代码：

```
do
        local var, limit, step = tonumber(e1),
        tonumber(e2), tonumber(e3)
        if not (var and limit and step) then error() end
        var = var - step
        while true do
                var = var + step
                if (step >= 0 and var > limit)
                or (step < 0 and var < limit) then
                        break
                end
                local v = var
                block
        end
end
```

需要注意下面这些细节：

- 三个控制表达式只会在循环开始前被计算一次，并且它们必须返回数字作为结果。

- *var*，*limit* 和 *step* 实际并不存在，这里只是出于解释的目的，使用它们。
- 如果不存在第三个表达式，step 会默认为 1。
- 可以使用 *break* 和 *goto* 语句跳出循环。
- 循环变量 *v* 的作用域是循环体，如果需要在循环后使用，应该在退出循环前将它赋值给其它变量。

基于种类的 *for* 语句通过叫做迭代器的函数工作。每一次迭代，迭代器产生一个新的值，当这个新值为 **nil** 时，迭代停止。这种形式 *for* 循环语法形式如下：

```
stat ::= for namelist in explist
        do block end
namelist ::= Name { ',' Name}
```

我们以下面的例子解释：

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

它等价于下面的代码：

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

需要注意下面这些细节：

- *explist* 仅在循环开始前计算一次，它们的结果是一个迭代函数，一个状态和第一个迭代器变量的初始值。
- 变量 *f*，*s* 和 *var* 实际并不存在，使用它们仅仅是出于解释的目的。

- 可以使用 *break* 跳出循环。
- 循环变量 *var_i* 的作用域在循环体内，如果我们需要在循环外使用它值，应该在循环结束前把它的值赋给其它变量。

### 3.3.6 函数调用作为语句

函数调用可以作为单一的一条语句，语法形式如下：

```
stat ::= functioncall
```

在这种情况下，函数的返回值被丢弃。函数调用的具体讨论在章节 3.4.10。

### 3.3.7 局部定义

局部变量可以在语句块的任何地方定义。局部变量定义的语法形式如下：

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

如果没有初始化赋值，变量会使用 **nil** 进行初始化。

一个程序块也是一个语句块（章节 3.3.2），所以局部变量可以定义程序块的任何地方。

局部变量的可见性规则在章节 3.5 讨论。

## 3.4 表达式

**Lua** 中的表达式语法形式如下：

```
exp ::= prefixexp
exp ::= nil | false | true
exp ::= Numeral
exp ::= LiteralString
exp ::= functiondef
exp ::= tableconstructor
exp ::= ‘...’
exp ::= exp binop exp
exp ::= unop exp
prefixexp ::= var | functioncall | ‘(’ exp ‘)’
```

*Numeral* 和 *LiteralString* 在章节 3.1 介绍。*var* 在章节 3.2 介绍。*functiondef* 在章节 3.4.11 介绍。*functioncall* 在章节 3.4.10 介绍。*tableconstructor* 在章节 3.4.9 介绍。可变参数表达式只可以在可变参数函数中使用，具体在章节 3.4.11 介绍。

二元算术运算符在章节 3.4.1 介绍，位运算符在章节 3.4.2 介绍，关系运算符在章节 3.4.4 介绍，逻辑运算符在 3.4.5 介绍，连接运算符在章节 3.4.6 介绍。一元减在章节 3.4.1 介绍，按位取反符在章节 3.4.2 介绍，逻辑否定运算符 (**not**) 在章节 3.4.5 介绍，求长度运算符在章节 3.4.7 介绍。

函数和可变参数表达式都能返回多个值作为结果。只有一条函数调用的语句 (章节 3.3.6)，返回值会被丢弃。表达式作为表达式列表的最后一个元素，表达式的结果不会被调整。其它情况，**Lua** 把表达式的结果调整为一个元素[32]。

下面是一些例子：

```
f()-- adjusted to 0 results
g(f(), x)-- f() is adjusted to 1 result
g(x, f())-- g gets x plus all results from f()
a,b,c = f(), x-- f() is adjusted to 1 result (c gets nil)
a,b = ...-- a gets the first vararg parameter, b gets
-- the second (both a and b can get nil if there
-- is no corresponding vararg parameter)

a,b,c = x, f()-- f() is adjusted to 2 results
a,b,c = f()-- f() is adjusted to 3 results
return f()-- returns all results from f()
return ...-- returns all received vararg parameters
return x,y,f()-- returns x, y, and all results from f()
{f()}-- creates a list with all results from f()
{...}-- creates a list with all vararg parameters
{f(), nil}-- f() is adjusted to 1 result
```

被括号包围的表达式只返回一个值。比如，$(f(x,y,z))$ 返回一个值，即使 $f$ 函数可能返回多个值也是这样，只返回它的第一个值，如果 $f$ 不返回值的话返回 **nil**。

[32]表达式不返回值的话，调整的结果是 **nil**。

### 3.4.1 算术操作符

**Lua** 支持下面这些算术运算符：

- +: 加法
- -: 减法
- *: 乘法
- /: 浮点除法
- //: 整型除法
- %: 模运算
- ^: 指数运算
- -: 一元减

除了指数和浮点除法，其余算术运算符按下面的方式工作：如果它的两个操作数都是整型，运算的结果也会是整型．如果它的两个操作数都是 *number* 或 emphstring，并且能被转换成 *number*，它们就会被转换为浮点数，然后遵循浮点数的运算规则进行计算 [33]，计算结果也会是一个浮点数。

指数和浮点除法总是把它们的操作数转换为浮点数然后进行计算，计算结果为浮点数。指数运算使用 *ISO C* 的库函数 *pow* 进行，所以非整数指数也是有效的。

整型除法 (//) 返回除法的商的部分。

**模**返回除法的余数部分。

整型的算术运算可能会产生溢出，根据补码的规则，这时得到的结果是经过环绕的。

### 3.4.2 位运算符

**Lua** 支持下面这些位运算操作符：

- &: 按位与
- |: 按位或

[33] **IEEE 754** 标准。

- ~: 按位异或
- »: 按位右移
- «: 按位左移
- ~: 按位取反

所有位操作会把它的操作数转换为整型（章节 3.4.3）。运算的结果也是整型。

右移和左移使用 0 来填充空位。

### 3.4.3 强制类型转换

**Lua** 在需要时会自动进行类型和表示方法的转换。使用位运算时，总是把浮点数转换为整型数。使用指数运算时，总是把整型数转换为浮点数。对于算数运算来说，如果它的两个参数中一个为整型数，一个为浮点数，那么整型数会被转换为浮点数，这被称为 *usual rule*。**C API** 也会根据需要转换参数类型。字符串连接符可以接受数字作为参数。

当需要 *number* 时，**Lua** 也会把 *string* 转换为 *number*。

整型数转换为浮点数时，如果这个数值有精确的浮点表示，就转为这个精确表示。否则的话，转为一个最接近的浮点表示。这种转换不会失败。

浮点数转换为整型数时，如果这个数值有精确的整型表示 [34]，就转为这个精确表示，否则，转换失败。

字符串可以根据语义转换为整型数或浮点数。

数字转换为字符串使用非特定的人类可读的格式。我们可以使用 *string.format* 来控制转换数字到字符串。

### 3.4.4 关系运算符

**Lua** 支持下面这些关系运算符：

- ==: 判等
- ~=: 判不等
- <: 判小于

[34] 浮点数的值为整数，并且在整型可表示的区间里。

- >: 判大于
- <=: 判小于或等于
- >=: 大于或等于

这些运算返回 **false** 或 **true**。

== 运算符首先比较它的两个操作数的类型，如果类型不同就会返回 **false**，相同的话，再比较它们的值返回结果。需要注意的是 *number* 的比较是基于数学，而不是二进制。

*table*，*userdata* 和 *thread* 通过引用比较：两个对象相同的条件是它们是同一个对象，也就是两个变量引用了同一个对象。具有相同引用的闭包是相等的。有任何差别的比包都是不等的。

我们可以通过 *eq* 元方法修改 **Lua** 比 *table* 和 *userdata* 的方式（章节 2.4）。

判等比较不会把字符串转换为数字，反之亦然。"0" == 0 的结果是 **false**，*t*[0] 和 *t*["0"] 表示表中不同的元素。

~= 是 == 的否定。

比较运算符的工作方式如下。如果两个操作数都是 *number*，则比较它们的算术大小 (忽略它们的子类型)。如果两个操作数都是 *string*，则根据当前语言环境比较。其余情况，**Lua** 会尝试调用操作数的 *lt* 或 *le* 元方法。$a > b$ 等价于 $b < a$ *and* $a >= b$ 等价于 $b <= a$。

根据 **IEEE 754 标准**，**NaN** 被认为不小于，不大于，不等于包括它自己在内的任何值。

### 3.4.5 逻辑运算符

**Lua** 中的逻辑运算符有 *and*，*or* 和 *not*。像所有控制结构一样（章节 3.3.4），**false** 和 **nil** 被认为是 **假**，其它都被认为**真**。

**not** 操作符返回 **false** 或 **true**。**and** 操作符返回操作数依赖于操作数本身，第一个操作数结果为**假**则返回地一个操作数，否则返回第二个操作数。**or** 操作符和 **and** 类似，它在第一个操作数为**真**时，返回第一个操作数，否则的话返回第二个操作数。**and** 和 **or** 都是**短路**的，当可以确定结果时，就不会计算剩余的表达式。下面是一些关于逻辑运算符运算结果的例子：

```
10 or 20            ---> 10
```

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

### 3.4.6 连接符

在 **Lua** 中可以使用.. 连接字符串。如果连接符的两个操作数都是字符串或数字，它们会根据在章节 3.4.3 中描述的规则进行转换。否则的话，*__concat* 元方法就会被调用（章节 2.4）。

### 3.4.7 求长度操作符

**Lua** 的求长度操作符是 #。*string* 的长度是它的字节数。[35]

对一个 *table* 使用 # 会返回它的边界。*table* 的边界是满足下面的条件的自然数：

```
(border == 0 or t[border] ~= nil) and t[border + 1] == nil
```

也就是说，*table* 的边界是一个后跟 **nil** 值的非 **nil** 值的位置。[36]。

可以通过 *__len* 元方法修改除了 *string* 外的任何类型值的求长度操作（章节 2.4）。

### 3.4.8 操作符优先级

**Lua** 中操作符的优先级从低到高的顺序如下：

```
or
and
< > <= >= ~= ==
|
~
```

[35] Lua 编码无关，所以依据字符个数判断长度是会出错的。

[36] 如果第一个元素为 **nil**，那么边界为 0。

```
&
<< >>
..
+ -
* / // %
一元操作符(not # - ~)
^
```

可以使用括号指定表达式的优先级。连接符.. 和指数符 ^ 是右结合的。所有二进制运算符是左结合的。

### 3.4.9 表构造器

表构造器是创建 *table* 的表达式。每一次构造器被计算，就会产生一个新的 *table*。构造器可以用来创建空表或是直接指定表的内容来创建表。构造器的语法形式如下：

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

形式 [*exp*1] = *exp*2 向表中添加一个 *key* 为 *exp*1，值为 *exp*2 的键对。形式 *name* = *exp* 等价于 ["*name*"] = *exp*。形式 *exp* 等价于 [*i*] = *exp*，这里的 i 是从 1 开始的连续整数。其它形式不影响域的计数。下面是一个例子：

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

它等价于：

```
do
        local t = {}
        t[f(1)] = g
        t[1] = "x"          -- 1st exp
        t[2] = "y"          -- 2nd exp
```

```
    t.x = 1                 -- t["x"] = 1
    t[3] = f(x)             -- 3rd exp
    t[30] = 23
    t[4] = 45               -- 4th exp
    a = t
end
```

表构造器赋值的顺序是不确定的[37]。

如果列表域的最后一个表达式是一个函数或可变参数表达式，它们返回的所有值会被加入表中（章节 3.4.10）。

为了方便生成代码，域列表的尾部有一个可选的分割符。

### 3.4.10 函数调用

**Lua** 的函数调用语法如下：

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

首先 *prefixexp* 和 *args* 被计算。如果 *prefixexp* 的值类型为 *function*。就会将 *args* 的计算结果作为参数调用 *prefixexp* 所指向的函数。如果 *prefixexp* 的值类型不为 *function* 的话，*prefixexp* 的值和 *args* 就会作为参数传给 *prefixexp* 的 *call* 元方法（章节 2.4）。

调用方法的形式为：

```
functioncall ::= prefixexp ':' Name args
```

*v:name(args)* 等价于 *v.name(v,args)*，除了 *v* 只被计算了一次。

参数的语法形式如下：

```
args ::= '(' [explist] ')'
args ::= tableconstructor
args ::= LiteralString
```

所有参数表达式在函数调用之前被计算。*f{fields}* 的调用形式等价于 *f({fields})*。参数列表实际上是一个 *table*。*f'string'* 和 *f"string"* 等价于 *f('string')*，也就是说，这时的参数表是一个字符串字面量。

在 *return* 语句中调用函数被称为**尾调用**。**Lau** 对尾调用有优化。尾调用会复用使用尾调用的函数的栈空间。因此，尾调用可以无限递归而不会

[37] 当 *key* 重复时，顺序才有一定规律。

溢栈。但是，尾调用覆盖了函数调用信息，这样就给调试带来了困难。触发 **Lua** 对尾调用的优化，需要使用特定的语法。只有 *return* 语句只有一个函数调用作为参数才能出发。下面的代码都不能出发尾调用：

```
return (f(x)) -- results adjusted to 1
return 2 * f(x)
return x, f(x) -- additional results
f(x); return -- results discarded
return x or f(x) -- results adjusted to 1
```

### 3.4.11 函数定义

函数定义的语法形式如下：

```
functiondef ::= function funcbody
funcbody ::= '(' [parlist] ')' block end
```

下面是函数定义的简化语法形式：

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

语句 *function f() body end* 等价于 *f=function() body end*
*function t.a.b.c.f() body end* 被解释为 *t.a.b.c.f=function() body end*
*local function f() body end* 被解释为 *local f;f=function() body end*
而不是 *local f=function() body end* [38]

函数定义是一个可以运行的值类型为 *function* 的表达式。**Lua** 编译程序块时同时也把其中所有的函数体编译了。之后，当 **Lua** 执行函数定义时，就会实例化一个 *function* 对象做为表达式的值。

函数变元被作为局部变量使用函数参数进行初始化。

```
parlist ::= namelist [ ',' '...' ] | '...'
```

考虑下面的函数定义：

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

[38] 当函数体需要使用 *f* 的引用时，就需要注意特别注意。

下面给出不同的调用下，变元接受参数的结果：

```
CALL              PARAMETERS

f(3)              a=3, b=nil
f(3, 4)           a=3, b=4
f(3, 4, 5)        a=3, b=4
f(r(), 10)        a=1, b=10
f(r())            a=1, b=2

g(3)              a=3, b=nil, ... -->  (nothing)
g(3, 4)           a=3, b=4,   ... -->  (nothing)
g(3, 4, 5, 8)     a=3, b=4,   ... -->  5  8
g(5, r())         a=5, b=1,   ... -->  2  3
```

函数结果可以使用 *return* 语句返回（章节 3.3.4）。如果没有 *return*，函数不返回结果。

函数可以返回的值的数量受到系统限制。这个限制值保证大于 1000。

分号被用来定义方法。这样定义的函数含有一个隐含的 *self* 参数。

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

## 3.5 可见性

**Lua** 是词法作用域语言。一个局部变量的作用域开始于它的定义之后，结束于

```
x = 10-- global variable
do-- new block
        local x = x-- new 'x', with value 10
        print(x)--> 10
        x = x+1
        do-- another block
                local x = x+1-- another 'x'
```

```
                print(x)--> 12
        end
        print(x)--> 11
end
print(x)--> 10  (the global one)
```

注意观察 *local x=x*，新定义的局部变量 $x$ 的作用域还有开始，所以等号右边的 $x$ 是外部的变量。

因为使用词法作用域，局部变量可以自由地被定义在作用域中的函数访问。一个被内部函数使用的在外部定义的局部变量叫做 *upvalue* 或者 **外部局部变量**。

**局部**语句的每次执行都会定义新的局部变量。考虑以下代码：

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

这段代码创建了 10 个闭包[39]。每个闭包都使用了一个不同的 $y$ 变量，但是都共用一个 $x$ 变量。

# 4 应用程序接口

本章节描述了 **Lua** 的 **C 语言 API**。也就是宿主程序用来和 **Lua** 进行交互的 **C 语言**函数。所有 API 函数和相关的类型，常量的定义可以在 *lua.h* 找到。

尽管我们使用**函数**这个术语，但实际上有一部分 API 是使用 **宏**的形式提供的。除非特别指明，这些宏只使用它们的参数一次 [40]，所以它们不会产生隐藏的副作用。

就像大多数 **C 库**一样，**Lua** 的 API 函数不对它的参数做合法性和一致性检查。当然，我们可以通过使用 *LUA_USE_APICHECK* 编译 **Lua** 来使它进行这个检查。

[39]十个匿名函数的实例。
[40]除了第一个参数 *Lua State*

**Lua** 库不维护全局信息，它通过 *Lua state* 参数来获取状态。

每个 *Lua State* 可以拥有一个或多个独立，并行的线程。*Lua State* 引用了一个线程 [41]。

除了 *lua_newstate* 函数外，线程指针必须被作为地一个参数传递给库中的函数。*lua_newstate* 函数返回一个新的状态指针。

## 4.1 栈

**Lua** 通过使用一个**虚拟栈**从 **C 语言**传递信息。栈中的每个元素代表一个 **Lua 值**[42]。API 函数可以通过 *Lua State* 参数访问这个栈。

**Lua** 调用 **C 函数**后，被调用的 **C 函数**会获得一个独立于原来的栈的新的栈。这个栈包含了 **C 函数**的所有参数，并且它可以被 **C 函数**用来存放临时的 **Lua** 值，以及返回结果给它的调用者[43]。

为了方便，大多数查询 API 并不严格遵循栈的规矩。它们可能通过索引来直接引用栈中的元素：一个正的索引可以用来表示元素在栈中的绝对位置[44]。一个负的索引用来表示从栈顶开始的相对偏移。更具体地说，如果一个栈中有 $n$ 个元素，索引 1 代表第一个元素 [45], 索引 $n$ 代表最后一个元素。索引 $-1$ 代表最后一个元素[46]。索引 $-n$ 代表第一个元素。

## 4.2 栈大小

当我们使用这 API 时，需要自己保证数据的一致性。特别的，我们需要自己保证栈没有溢出。我们可以使用 *lua_checkstack* 函数保证栈有足够的空间压入新元素。

从 **Lua** 中调用 **C**，能够保证栈中至少有 *LUA_MINSTACK* 个多余的槽。*LUA_MINSTACK* 被定义为 20，通常只要我们没有循环压入元素到栈中，是不要担心栈空间的问题。

当一个 **Lua** 函数没有固定的结果个数时[47]，**Lua** 保证有足够的栈空间存放这些结果，但它不保证有多余的空间。因此，在这样的一个函数调用后压入元素应该使用 *lua_checkstack* 函数。

[41] 线程也引用了一个和它绑定的 *Lua State*。
[42] **nil**，**number**，**string** 等等。
[43] 参考 *lua_CFunction* 函数。
[44] 从 1 开始。
[45] 地一个被压入栈中的元素。
[46] 也就是栈顶元素
[47] 参考 *lua_call* 函数

## 4.3 有效和可接受的索引

**API 函数**只接受有效或可接受的栈索引。

有效索引是指引用了一个存储了可以被修改的 **Lua** 值的位置。它的范围是 1 到栈顶加上伪索引[48]。伪索引被用来访问注册表 (章节 4.5) 和 **C 函数**的上值 (章节 4.4)。

## 4.4 C 闭包

**C** 函数创建时可以把一些值和它关联在一起创建 **C 闭包** [49]参考 *lua_pushcclosure* 函数。)。这些值被称为上值，可以在函数调用后被访问到。

**C** 函数被调用后，它的上值被存放在特定的伪索引。这些伪索引是通过 *lua_upvalueindex* 宏产生的。第一个和函数关联的上值在索引 *lua_upvalueindex(1)*，以此类推。调用 *lua_upvalueindex(n)*，如果 $n$ 大于当前函数的上值 [50]，它就会产生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**，它是一个预定义的可以被 **C** 代码使用存储 **Lua** 值的表。注册表所在的伪索引是 *LUA_REGISTRYINDEX*。任何 **C 库**可以在表中存储数据，但是需要注意不要和其它库使用相同的 *key*。习惯上，我们应该使用包含库名的字符串或是一个带有一个 C 对象地址 *light userdata*，或是被我们的代码创建的 **Lua** 对象作为 *key*。和变量名一样，字符串 *key* 以下划线开始后跟大写字母的名字是被 **Lua** 所保留的。

注册表中的整型 *key* 被引用系统和部分预定义值使用[51]。整型 *key* 不能用于其它目的。

当我们创建一个 **Lua State**，它的注册表就会出现一些预定义的值。这些预定义的值可以通过定义在 *lua.h* 中的常量索引。下面是被定义的常量：

- LUA_RIDX_MAINTHREAD: 对应状态的主线程[52]。
- LUA_RIDX_GLOBALS: 对应全局环境。

[48]用来表示可以被 **C 语言**代码访问到，但不在栈中的位置。
[49](
[50]但不大于 256。256 是一个闭包上值个数的最值 +1。
[51]参考 *luaL_ref* 函数。
[52]主线程是指和状态一同创建的线程。

## 4.6 在 C 语言中进行错误处理

**Lua** 在内部使用 **C 语言**的 *longjmp* 来处理错误 [53]。当 **Lua** 遇到错误[54] ，就会进行 *long jump*。在被保护的环境下可以使用 *setjmp* 来设置一个恢复点。任何错误都会被跳转到最近的恢复点。

在 **C** 函数里我们可以通过调用 *lua_error* 函数来产生一个错误。

大多数 **API 函数**都能引发错误[55]。本文档已经说明了每个函数是否会引发错误。

如果一个错误发生在未被包或的环境，**Lua** 会调用 *panic* 函数 [56]然后中断执行，退出宿主程序。我们可以通过不让 *panic* 函数返回来避免退出宿主程序 [57]。

*panic* 函数就像它的名字所暗示的那样，是在不得已的情况下才使用的。我们应该避免使用它。作为一个一般规则，**Lua** 以 *Lua State* 为参数调用一个 **C 函数**，这个函数可以对 *Lua State* 进行任何它想要的处理，就像它已经被保护一样。然而，**C** 语言操作 *Lua State* 应该使用不产生错误的 API 函数 [58]。

*panic* 函数就像一个消息处理器 (章节 2.3)。特别的，错误对象在栈顶。栈的空间并不被保证。在压入元素前，*panic* 函数应该先检查可用的空间 (章节 4.2)。

## 4.7 在 C 语言中进行协程处理

**Lua** 使用 *longjmp* 来进行协程的让出。如果 C 函数 $foo$ 调用了一个 API 函数 [59] ，然后这个 API 函数让出执行，**Lua** 就不能在返回 $foo$ 这个函数，因为 *longjmp* 已经移除了它的栈帧。

为了避免这类问题，在一个 API 调用中使用让出 **Lua** 会产生一个错误。但下面三个函数不会产生这个错误:、*lua_yieldk* 函数，*lua_callk* 函数和 *lua_pcallk* 函数。这些函数接受一个 *continuation* 函数 [60]，在让出后继

[53]如果使用 **C++** 编译 **Lua**，那它就会使用异常来处理错误，具体在 **Lua** 源代码中搜索 *LUAI_THROW* 查看相关代码。
[54]比如内存分配错误或类型错误。
[55]比如内存分配错误。
[56]参考 *lua_atpanic* 函数。
[57]通过 *long jump* 跳转到自己维护的恢复点。
[58]一个 **Lua** 函数的参数，一个被存放在注册表中的 *Lua State*，或是 *lua_newthread* 的结果。
[59]直接或间接调用执行让出的函数。
[60]作为一个名叫 $k$ 的参数。

续执行。

## 4.8 函数和类型

在这里我们按照字母顺序列出了所有 **C API** 中的**函数**和 **类型**。每个函数有一个标记，形式如下：

第一个域，*o*，代表函数从栈中弹出多少个元素。第二个域，*p*，代表函数压入栈中多少个元素 [61]。一个形式为 *x*|*y* 的域意味着函数可以根据情况压入或弹出 *x* 或 *y* 元素。一个 ? 标记意味着我们不能从它的参数看出函数弹出压入的元素个数 [62]。第三个域，*x* 指明函数是否会引起错误：– 意味着函数不会引起任何错误。*m* 意味着函数可能引起内存耗尽错误，并且这个错误会引起 *__gc* 元方法的执行。*e* 意味着这个函数可能会引起任意的错误 [63]。*v* 意味着函数可能故意产生一个错误。

[61] 函数总是在弹出它的参数后将结果压入栈中。
[62] 可能需要根据栈的内容而定。
[63] 它可以直接执行任意的 **Lua** 代码或间接通过元方法执行。

4.9 调试接口

# 5 辅助库

5.1 函数和类型

# 6 标准库

6.1 基础函数

6.2 协程处理

6.3 模块

6.4 字符串处理

6.4.1 模式

6.4.2 Format Strings for Pack and Unpack

6.5 UTF-8 支持

6.6 表处理

6.7 数学函数

6.8 Input and Output Facilities

6.9 Operating System Facilities

6.10 调试库

# 7 Lua Standalone

# 8 向前兼容

8.1 语言变化

8.2 库变化

8.3 API 变化

# 9 Lua 完整语法定义