Lua编程入门

书栈(BookStack.CN)

目　录

致谢

当前文档 《Lua编程入门》 由 进击的皇虫 使用 书栈(BookStack.CN) 进行构建，生成于 2019-03-02。

书栈(BookStack.CN) 仅提供文档编写、整理、归类等功能，以及对文档内容的生成和导出工具。

文档内容由网友们编写和整理，书栈(BookStack.CN) 难以确认文档内容知识点是否错漏。如果您在阅读文档获取知识的时候，发现文档内容有不恰当的地方，请向我们反馈，让我们共同携手，将知识准确、高效且有效地传递给每一个人。

同时，如果您在日常工作、生活和学习中遇到有价值有营养的知识文档，欢迎分享到 书栈(BookStack.CN) ，为知识的传承献上您的一份力量！

如果当前文档生成时间太久，请到 书栈(BookStack.CN) 获取最新的文档，以跟上知识更新换代的步伐。

文档地址: http://www.bookstack.cn/books/luaprimer

书栈官网: http://www.bookstack.cn

书栈开源: https://github.com/TruthHun

分享，让知识传承更久远！ 感谢知识的创造者，感谢知识的分享者，也感谢每一位阅读到此处的读者，因为我们都将成为知识的传承者。

《Lua编程入门》

学习Lua过程中的一些笔记整理

开始阅读

> 来源(书栈小编注): https://github.com/andycai/luaprimer

Lua 基础知识

(1) 变量

赋值

赋值是改变一个变量的值和改变表域的最基本的方法。Lua 中的变量没有类型，只管赋值即可。比如在 Lua 命令行下输入：

```
1.  end_of_world = "death"
2.  print(end_of_world)
3.  end_of_world = 2012
4.  print(end_of_world)
```

上面这四行代码 Lua 不会报错，而会输出：

```
1.  death
2.  2012
```

局部变量

使用 local 创建一个局部变量，与全局变量不同，局部变量只在被声明的那个代码块内有效

```
1.  x = 10
2.  local i = 1              -- 局部变量
3.
4.  while i<=x do
5.      local x = i*2        -- while 中的局部变量
6.      print(x)             --> 2, 4, 6, 8, ...
7.      i = i + 1
8.  end
```

应该尽可能的使用局部变量，有两个好处：

1. 避免命名冲突
2. 访问局部变量的速度比全局变量更快

代码块(block)

代码块指一个控制结构内，一个函数体，或者一个chunk（变量被声明的那个文件或者文本串）。

我们给block划定一个明确的界限：do..end内的部分。当你想更好的控制局部变量的作用范围的时候这是很有用的。

```
do
    local a2 = 2*a
    local d = sqrt(b^2 - 4*a*c)
    x1 = (-b + d)/a2
    x2 = (-b - d)/a2
end             -- scope of 'a2' and 'd' ends here
print(x1, x2)
```

（2）类型

虽说变量没有类型，但并不是说数据不分类型。Lua 基本数据类型共有八个：nil、boolean、number、string、function、userdata、thread、table。

- Nil Lua中特殊的类型，他只有一个值：nil；一个全局变量没有被赋值以前默认值为nil；给全局变量赋nil可以删除该变量。
- Booleans 两个取值false和true。但要注意Lua中所有的值都可以作为条件。在控制结构的条件中除了false和nil为假，其他值都为真。所以Lua认为0和空串都是真。
- Numbers 即实数，Lua 中的所有数都用双精度浮点数表示。
- Strings 字符串类型，指字符的序列，Lua中字符串是不可以修改的，你可以创建一个新的变量存放你要的字符串。
- Table 是很强大的数据结构，也是 Lua 中唯一的数据结构。可以看作是数组或者字典。
- Function 函数是第一类值（和其他变量相同），意味着函数可以存储在变量中，可以作为函数的参数，也可以作为函数的返回值。
- Userdata userdata可以将C数据存放在Lua变量中，userdata在Lua中除了赋值和相等比较外没有预定义的操作。userdata用来描述应用程序或者使用C实现的库创建的新类型。例如：用标准I/O库来描述文件。
- Thread 线程会在其它章节来介绍。

可以用 **type** 函数取得表达式的数据类型：

```
print(type(undefined_var))
print(type(true))
print(type(3.14))
print(type('Hello World'))
print(type(type))
print(type({}))
```

（3）表达式

操作符

1. 算术运算符: + - * / ^ (加减乘除幂)
2. 关系运算符: < > <= >= == ~=
3. 逻辑运算符: and or not
4. 连接运算符: ..

有几个操作符跟C语言不一样的:

- a ~= b 即 a 不等于 b
- a ^ b 即 a 的 b 次方
- a .. b 将 a 和 b 作为字符串连接

优先级:

1. ^
2. not -(负号)
3. * /
4. + -
5. ..
6. < > <= >= ~= ==
7. and
8. or

表的构造:

最简单的构造函数是{}，用来创建一个空表。可以直接初始化数组:

```
1.  days = {"Sunday", "Monday", "Tuesday", "Wednesday", "Thursday", "Friday",
    "Saturday"}
```

Lua将”Sunday”初始化days[1]（第一个元素索引为1），不推荐数组下标以0开始，否则很多标准库不能使用。

在同一个构造函数中可以数组风格和字典风格进行初始化:

```
1.  polyline = {color="blue", thickness=2, npoints=4,
2.                  {x=0,   y=0},
```

```
                {x=-10, y=0},
                {x=-10, y=1},
                {x=0,   y=1}
}
```

多重赋值和多返回值

另外 Lua 还支持多重赋值（还支持函数返回多个值）。也就是说：等号右边的值依次赋值给等号左边的变量。比如：

```
year, month, day = 2011, 3, 12
print(year, month, day)
reutrn year, month, day -- 多返回值
a, b = f()
```

于是，交换两个变量值的操作也变得非常简单：

```
a, b = b, a
```

(4) 控制流

if

```
name = "peach"
if name == "apple" then
    -- body
elseif name == "banana" then
    -- body
else
    -- body
end
```

for

```
-- 初始值, 终止值, 步长
for i=1, 10, 2 do
    print(i)
end

```

```
-- 数组
for k, v in ipairs(table) do
    print(k, v)
end

-- 字典
for k, v in pairs(table) do
    print(k, v)
end
```

反向表构造实例:

```
revDays = {}
for i,v in ipairs(days) do
    revDays[v] = i
end
```

while

```
while i<10 do
    print(i)
    i = i + 1
end
```

repeat-until

```
repeat
    print(i)
    i = i + 1
until i < 10
```

break 和 return

break 语句可用来退出当前循环(for, repeat, while)，循环外部不可以使用。

return 用来从函数返回结果，当一个函数自然结束，结尾会有一个默认的return。

Lua语法要求break和return只能出现在block的结尾一句（也就是说：作为chunk的最后一句，或者在end之前，或者else前，或者until前）：

```
local i = 1
```

```
while a[i] do
    if a[i] == v then break end
    i = i + 1
end
```

(5) C/C++ 中的 Lua

首先是最简单的 Lua 为 C/C++ 程序变量赋值，类似史前的 INI 配置文件。

```
width = 640
height = 480
```

这样的赋值即设置全局变量，本质上就是在全局表中添加字段。

在 C/C++ 中，Lua 其实并不是直接去改变变量的值，而是宿主程序通过「读取脚本中设置的全局变量到栈、类型检查、从栈上取值」几步去主动查询。

```
int w, h;
if (luaL_loadfile(L, fname) || // 读取文件，将内容作为一个函数压栈
    lua_pcall(L, 0, 0, 0))     // 执行栈顶函数，0个参数、0个返回值、无出错处理函数（出错时直接把错误信息压栈）
    error();

lua_getglobal(L, "width");     // 将全局变量 width 压栈
lua_getglobal(L, "height");    // 将全局变量 height 压栈
if (!lua_isnumber(L, -2))      // 自顶向下第二个元素是否为数字
    error();
if (!lua_isnumber(L, -1))      // 自顶向下第一个元素是否为数字
    error();
w = lua_tointeger(L, -2);      // 自顶向下第二个元素转为整型返回
h = lua_tointeger(L, -1);      // 自顶向下第一个元素转为整型返回
```

读取表的字段的操作也是类似，只不过细节上比较麻烦，有点让我想起在汇编里调戏各种寄存器：

```
score = { chinese=80, english=85 }

int chinese, english;
if (luaL_loadfile(L, fname) || lua_pcall(L, 0, 0, 0))
    error();

lua_getglobal(L, "score");       // 全局变量 score 压栈
```

```
8.
9.  lua_pushstring(L, "chinese");    // 字符串 math 压栈
10. lua_gettable(L, -2);             // 以自顶向下第二个元素为表、第一个元素为索引取值，弹
    栈，将该值压栈
11. if (!lua_isnumber(L, -1))        // 栈顶元素是否为数字
12.     error();
13. chinese = lua_tointeger(L, -2);
14. lua_pop(L, 1);                   // 弹出一个元素 (此时栈顶为 score 变量)
15.
16. lua_getfield(L, -1, "english");  // Lua5.1开始提供该函数简化七八两行
17. if (!lua_isnumber(L, -1))
18.     error();
19. english = lua_tointeger(L, -2);
20. lua_pop(L, 1);                   // 如果就此结束，这一行弹不弹都无所谓了
```

前面说过，设置全局变量本质就是在全局表中添加字段，所以 lua_getglobal 函数本质是从全局表中读取字段。没错，lua_getglobal 本身就是一个宏：

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

宏 LUA_GLOBALSINDEX 指明的就是全局表的索引。

导航

- 目录
- 下一章：环境与模块

环境与模块

(1) 全局变量与环境

lua 中真正存储全局变量的地方不是在 _G 里面，而是在setfenv(i,table)的table中，所有当前的全局变量都在这里面找，只不过在程序开始时lua会默认先设置一个变量 _G= 这个里面的table而已。所以在新设置环境后，如果还想找到之前的全局变量，通常需要附加上为新的table设置元表{_index=_G}

下面的几个例子:

```
1.  a=1
2.  print(a)
3.  print(_G.a)
4.  --正常情况，输出1,1
5.
6.  a=1
7.  setfenv(1,{})
8.  print(a)
9.  print(_G.a)
10. --这时会出错说找不到print，因为当前的全局变量表示空的，啥也找不到的
11.
12. a=1
13. setfenv(1,{_G=_G})
14. _G.print(_G.a)
15.
16. print(a)
17. --这时_G.print(_G.a)可以正常吗，因为可以在新的table中找到一个叫_G的表，这个_G有之前的奈尔全局变量，但是下面的print(a)则找不到print，因为当前的table{_G=_G}没有一个叫print的东西
18.
19. local mt={__index=_G}
20. local t={}
21. setmetatable(t,mt)
22. setfenv(1,t)
23. print(a)
24. print(_G.a)
25. --这是正确输出，因为新的全局表采用之前的表做找不到时的索引，原先的表里面存在print 、_G、 a这些东西
```

setfenv的第一个参数可以是当前的堆栈层次，如**1**代表当前代码块，**2**表调用当前的上一层，也可以是

具体的那个函数名，表示在那个函数里。

每个新创建的函数都将继承创建它的那个函数的全局环境。

从 Lua 5.1 开始，Lua 加入了标准的模块管理机制，可以把一些公用的代码放在一个文件里，以 API 接口的形式在其他地方调用，有利于代码的重用和降低代码耦合度。

(3) 创建模块

其实 Lua 的模块是由变量、函数等已知元素组成的 table，因此创建一个模块很简单，就是创建一个 table，然后把需要导出的常量、函数放入其中，最后返回这个 table 就行。格式如下：

```
1.  -- 定义一个名为 module 的模块
2.  module = {}
3.
4.  -- 定义一个常量
5.  module.constant = "this is a constant"
6.
7.  -- 定义一个函数
8.  function module.func1()
9.      io.write("this is a public function!\n")
10. end
11.
12. local function func2()
13.     print("this is a private function!")
14. end
15.
16. function module.func3()
17.     func2()
18. end
19.
20. return module
```

由上可知，模块的结构就是一个 table 的结构，因此可以像操作调用 table 里的元素那样来操作调用模块里的常量或函数。不过上面的 func2 声明为程序块的局部变量，即表示一个私有函数，因此是不能从外部访问模块里的这个私有函数，必须通过模块里的共有函数来调用。

最后，把上面的模块代码保存为跟模块名一样的 lua 文件里（例如上面是 module.lua），那么一个自定义的模块就创建成功。

(4) 加载模块

Lua 提供一个名为 require 的函数来加载模块，使用也很简单，它只有一个参数，这个参数就是要

指定加载的模块名，例如：

```
require("<模块名>")
-- 或者是
-- require "<模块名>"
```

然后会返回一个由模块常量或函数组成的 table，并且还会定义一个包含该 table 的全局变量。

或者给加载的模块定义一个别名变量，方便调用：

```
local m = require("module")

print(m.constant)

m.func3()
```

require 的意义就是导入一堆可用的名称，这些名称（非local的）都包含在一个table中，这个table再被包含在当前的全局表（“通常的那个_G”）中，这样访问一个模块中的变量就可以使用_G.table.**了，初学者可能会认为模块里的名称在导入后直接就是在 _G 中了。

```
m=require module
```

的 m 取决于这个导入的文件的返回值，没有返回值时true，所以在标准的情况下模块的结尾应该return 这个模块的名字，这样 m 就是这个模块的table了。

(5) 加载机制

对于自定义的模块，模块文件不是放在哪个文件目录都行，函数 require 有它自己的文件路径加载策略，它会尝试从 Lua 文件或 C 程序库中加载模块。

require 用于搜索 Lua 文件的路径是存放在全局变量 package.path 中，当 Lua 启动后，会以环境变量 LUA_PATH 的值来初始这个环境变量。如果没有找到该环境变量，则使用一个编译时定义的默认路径来初始化。

当然，如果没有 LUA_PATH 这个环境变量，也可以自定义设置，在当前用户根目录下打开 .profile 文件（没有则创建，打开 .bashrc 文件也可以），例如把 “~/lua/“ 路径加入 LUA_PATH 环境变量里：

```
#LUA_PATH
export LUA_PATH="~/lua/?.lua;;"
```

文件路径以 “;” 号分隔，最后的 2 个 “;;” 表示新加的路径后面加上原来的默认路径。

接着，更新环境变量参数，使之立即生效：

```
source ~/.profile
```

这时假设 package.path 的值是：

```
/Users/dengjoe/lua/?.lua;./?.lua;/usr/local/share/lua/5.1/?.lua;/usr/local/share/
```

那么调用 require(“module”) 时就会尝试打开以下文件目录去搜索目标

```
/Users/dengjoe/lua/module.lua;
./module.lua
/usr/local/share/lua/5.1/module.lua
/usr/local/share/lua/5.1/module/init.lua
/usr/local/lib/lua/5.1/module.lua
/usr/local/lib/lua/5.1/module/init.lua
```

如果找过目标文件，则会调用 package.loadfile 来加载模块。否则，就会去找 C 程序库。搜索的文件路径是从全局变量 package.cpath 获取，而这个变量则是通过环境变量 LUA_CPATH 来初始。搜索的策略跟上面的一样，只不过现在换成搜索的是 so 或 dll 类型的文件。如果找得到，那么 require 就会通过 package.loadlib 来加载它。

导航

- 目录
- 上一章：Lua 基础知识
- 下一章：函数与面向对象

函数与面向对象

变量声明与 C 语言的不同

Lua 中有一个常见的用法，不论变量、函数都可以用下面这种方法保存到局部变量中（同时加快访问速度）：

```
1.  local foo = foo
```

书里加了个括号来解释这种写法：

> The local foo becomes visible only after its declaration.

这一点需要瞎扯的是 C 语言里相应的东西。

```
1.  int foo = 12;
2.  int bar = 6;
3.
4.  void foobar(void)
5.  {
6.      int foo = foo;
7.      int bar[bar];
8.  }
```

与 Lua 不同，在 C 语言中初始赋值是声明之后的事情。所以这里函数 foobar 中的 foo 会被初始化为自己（而不是全局的 foo，所以值不确定），bar 却被合法地定义为一个含有 6 个元素的数组。

看似多余的限制

另一个有趣的现象是在 4.4 节中说到：

> For syntactic reasons, a break or return can appear only as the last statement of a block; in other words, as the last statement in your chunk or just before an end, an else, or an until.

乍一看觉得加上这个限制真是麻烦，但想想这不正是 break/return 的正确用法么？因为其后的语句都永远不会被执行到，所以如果不是在块的最后写 break/return 是毫无意义的（调试除外）。虽然看上去是挺多余的一段话，但也算是说出了事物的本源。

函数的本质

第六章 More About Functions 中说到我们平时在 Lua 中写的函数声明

```
1.  function foo (x) return 2*x end
```

其实是一种语法糖，本质上我们可以把它写成如下代码：

```
1.  foo = function (x) return 2*x end
```

于是也就可以说

- Lua 中的所有函数都是匿名函数，之前所谓「具名函数」只是保存了某个匿名函数的变量罢了。
- Lua 中的函数声明其实只是一个语句而已。

终于有用的知识

在第 47 页看到了一段令人泪流满面的代码和运行结果：

```
1.  function derivative (f, delta)
2.    delta = delta or 1e-4
3.    return function (x)
4.              return (f(x + delta) - f(x))/delta
5.            end
6.  end
7.
8.  c = derivative(math.sin)
9.  print(math.cos(10), c(10))
10. --> -0.83907152907645 -0.83904432662041
```

最初我并不知道 derivative 是什么意思，但看了示例代码和运行结果，顿时恍然大悟：这货不就是导数吗？

沙盒

背景知识

Lua 给我的感觉是：各种内置函数和标准库的存在感都是比较强的。如果执行这句：

```
1.  for name in pairs(_G) do print(name) end
```

就会把各种环境中已存在名称的打印出来：

- 全局变量：比如字符串 _VERSION。
- 内置函数：比如 print、tonumber、dofile 之类。
- 模块名称：比如 string、io、coroutine 之类。

这里的全局变量 _G 就是存放环境的表（于是会有 _G 中存在着 _G._G 的递归）。

于是，平时对于全局变量的访问就可以等同于对 _G 表进行索引：

```
value = _G[varname]  --> value = varname
_G[varname] = value  --> varname = value
```

改变函数的环境

函数的上下文环境可以通过 setfenv(f, table) 函数改变，其中 table 是新的环境表，f 表示需要被改变环境的函数。如果 f 是数字，则将其视为堆栈层级（Stack Level），从而指明函数（1 为当前函数，2 为上一级函数）：

```
a = 3           -- 全局变量 a
setfenv(1, {}) -- 将当前函数的环境表改为空表
print(a)        -- 出错，因为当前环境表中 print 已经不存在了
```

没错，不仅是 a 不存在，连 print 都一块儿不存在了。如果需要引用以前的 print 则需要在新的环境表中放入线索：

```
a = 3
setfenv(1, { g = _G })
g.print(a)              -- 输出 nil
g.print(g.a)            -- 输出 3
```

沙盒

于是，出于安全或者改变一些内置函数行为的目的，需要在执行 Lua 代码时改变其环境时便可以使用 setfenv 函数。仅将你认为安全的函数或者新的实现加入新环境表中：

```
local env = {}  -- 沙盒环境表，按需要添入允许的函数

function run_sandbox(code)
  local func, message = loadstring(code)
  if not func then return nil, message end  -- 传入代码本身错误
  setfenv(func, env)
  return pcall(func)
```

```
end
```

Lua 5.2 的 _ENV 变量

Lua 5.2 中所有对全局变量 var 的访问都会在语法上翻译为 _ENV.var。而 _ENV 本身被认为是处于当前块外的一个局部变量。（于是只要你自己定义一个名为 _ENV 的变量，就自动成为了其后代码所处的「环境」（enviroment）。另有一个「全局环境」（global enviroment）的概念，指初始的 _G 表。）

Lua 的作者之一 Roberto Ierusalimschy 同志在介绍 Lua 5.2 时说:

> the new scheme, with _ENV, allows the main benefit of setfenv with a little more than syntactic sugar.

就我的理解来说，优点就是原先虚无缥缈只能通过 setfenv、getfenv 访问的所谓「环境」终于实体化为一个始终存在的变量 _ENV 了。

于是以下两个函数内容大致是一样的:

```
-- Lua 5.1
function foobar()
  setfenv(1, {})
  -- code here
end

-- Lua 5.2
function foobar()
  local _ENV = {}
  -- code here
end
```

而更进一步的是，5.2 中对 load 函数作出了修改。（包括但不限于 :)）合并了 loadstring 功能，并可以在参数中指定所使用的环境表:

```
local func, message = load(code, nil, "t", env)
```

面向对象

没错，Lua 中只存在表（Table）这么唯一一种数据结构，但依旧可以玩出面向对象的概念。

添加成员函数

好吧，如果熟悉 C++ 还是很好理解类似的进化过程的：如果说 struct 里可以添加函数是从 C 过渡到 C++ 的第一认识的话，为 Table 添加函数也可以算是认识 Lua 是如何面向对象的第一步吧。

```
player = { health = 200 }  --> 一个普通的 player 表，这里看作是一个对象
function takeDamage(self, amount)
  self.health = self.health - amount
end

takeDamage(player, 20)  --> 调用
```

如何将独立的 takeDamage 塞进 player 中咧？答案是直接定义进去：

```
player = { health = 200 }
function player.takeDamage(self, amount)
  self.health = self.health - amount
end

player.takeDamage(player, 20)  --> 调用
```

这样就相当于在 player 表中添加了一个叫做 takeDamage 的字段，和下面的代码是一样的：

```
player = {
  health = 200,
  takeDamage = function(self, amount)  --> Lua 中的函数是 first-class value
    self.health = self.health - amount
  end
}

player.takeDamage(player, 20)  --> 调用
```

调用时的 player.takeDamage(player, 20) 稍显不和谐（据说用术语叫做 DRY），于是就要出动「冒号操作符」这个专门为此而生的语法糖了：

```
player:takeDamage(20)              --> 等同于 player.takeDamage(player, 20)
function player:takeDamage(amount) --> 等同于 function player.takeDamage(self,
amount)
```

从对象升华到类

类的意义在于提取一类对象的共同点从而实现量产（我瞎扯的 >_<）。同样木有 Class 概念的 Javascript 使用 prototype 实现面向对象，Lua 则通过 Metatable 实现与 prototype 类

似的功能。

```
Player = {}

function Player:create(o)     --> 参数 o 可以暂时不管
  o = o or { health = 200 }  --> Lua 的 or 与一般的 || 不同，如果非 nil 则返回该非 nil 值
  setmetatable(o, self)
  self.__index = self
  return o
end

function Player:takeDamage(amount)
  self.health = self.health - amount
end

playerA = Player:create()  --> 参数 o 为 nil
playerB = Player:create()

playerA:takeDamage(20)
playerB:takeDamage(40)
```

顾名思义 Metatable 也是一个 Table，可以通过在其中存放一些函数（称作 metamethod）从而修改一些默认的求值行为（如何显示为字符串、如何相加、如何连接、如何进行索引）。Metatable 的 **index** 域设置了「如何进行索引」的方法。例如调用 **foo.bar** 时，如果在 **foo** 中没有找到名为 **bar** 的域时，则会调用 **Metatable:** index(foo, bar)。于是：

```
playerA:takeDamage(20)
```

因为在 playerA 中并不存在 takeDamge 函数，于是求助于 Metatable:

```
getmetatable(playerA).__index.takeDamage(playerA, 20)
```

带入 Metatable 后：

```
Player.__index.takeDamage(playerA, 20)
```

因为 Player 的 __index 在 create 时被指定为 self，所以最终变为：

```
Player.takeDamage(playerA, 20)
```

于是 takeDamage 的 self 得到了正确的对象 playerA。

继承

继承是面向对象的一大特性，明白了如何创建「类」，那么继承也就比较明了了，还记得大明湖畔的参数 o 么？

```
RMBPlayer = Player:create()
function RMBPlayer:broadcast(message)  --> 为子类添加新的方法
  print(message)
end
function RMBPlayer:takeDamage(amount)  --> 子类重载父类方法
  self.health = self.health - amount / (self.money / 100)
end

vip = RMBPlayer:create { money = 200 } --> 子类添加新成员（单个 Table 作为参数可以省略括号）

vip:takeDamage(20)
vip:broadcast("F*ck")
```

以上便是 Lua 中实现面向对象的基本方法。

导航

- 目录
- 上一章：环境与模块
- 下一章：标准库

标准库

String

```
1.  string.byte
2.  string.char
3.  string.dump
4.  string.find
5.  string.format
6.  string.gmatch
7.  string.gsub
8.  string.len
9.  string.lower
10. string.match
11. string.rep
12. string.reverse
13. string.sub
14. string.upper
```

> 在string库中功能最强大的函数是: string.find（字符串查找），string.gsub（全局字符串替换），and string.gfind（全局字符串查找）。这些函数都是基于模式匹配的。
>
> 与其他脚本语言不同的是，Lua并不使用POSIX规范的正则表达式（也写作regexp）来进行模式匹配。主要的原因出于程序大小方面的考虑：实现一个典型的符合POSIX标准的regexp大概需要4000行代码，这比整个Lua标准库加在一起都大。权衡之下，Lua中的模式匹配的实现只用了500行代码，当然这意味着不可能实现POSIX所规范的所有更能。然而，Lua中的模式匹配功能是很强大的，并且包含了一些使用标准POSIX模式匹配不容易实现的功能。

(1) pattern 模式

下面的表列出了Lua支持的所有字符类:

```
1.  .      任意字符
2.  %a     字母
3.  %c     控制字符
4.  %d     数字
5.  %l     小写字母
6.  %p     标点字符
7.  %s     空白符
8.  %u     大写字母
```

```
%w      字母和数字
%x      十六进制数字
%z      代表0的字符
```

可以使用修饰符来修饰模式增强模式的表达能力，Lua中的模式修饰符有四个：

```
+       匹配前一字符1次或多次
*       匹配前一字符0次或多次
-       匹配前一字符0次或多次
?       匹配前一字符0次或1次
```

‘%b’ 用来匹配对称的字符。常写为 ‘%bxy’ ，x和y是任意两个不同的字符；x作为匹配的开始，y作为匹配的结束。比如，’%b()’ 匹配以 ‘(‘ 开始，以 ‘)’ 结束的字符串：

```
print(string.gsub("a (enclosed (in) parentheses) line", "%b()", "")) --> a
line
```

常用的这种模式有：’%b()’ ，’%b[]’，’%b%{%}’ 和 ‘%b<>’。你也可以使用任何字符作为分隔符。

(2) capture 捕获

Capture是这样一种机制：可以使用模式串的一部分匹配目标串的一部分。将你想捕获的模式用圆括号括起来，就指定了一个capture。

```
pair = "name = Anna"
_, _, key, value = string.find(pair, "(%a+)%s*=%s*(%a+)")
print(key, value)    --> name   Anna
```

(3) string.find 字符串查找

string.find 的基本应用就是用来在目标串（subject string）内搜索匹配指定的模式的串，函数返回两个值：匹配串开始索引和结束索引。

```
s = "hello world"
i, j = string.find(s, "hello")
print(i, j)                     --> 1    5
print(string.sub(s, i, j))      --> hello
print(string.find(s, "world"))  --> 7    11
i, j = string.find(s, "l")
print(i, j)                     --> 3    3
```

```
print(string.find(s, "lll"))        --> nil
```

string.find函数第三个参数是可选的：标示目标串中搜索的起始位置。

在string.find使用captures的时候，函数会返回捕获的值作为额外的结果：

```
pair = "name = Anna"
_, _, key, value = string.find(pair, "(%a+)%s*=%s*(%a+)")
print(key, value)    --> name   Anna
```

看个例子，假定你想查找一个字符串中单引号或者双引号引起来的子串，你可能使用模式 ‘[“‘].-[“‘]’，但是这个模式对处理类似字符串 “it’s all right” 会出问题。为了解决这个问题，可以使用向前引用，使用捕获的第一个引号来表示第二个引号：

```
s = [[then he said: "it's all right"!]]
a, b, c, quotedPart = string.find(s, "(["'])(.-)%1")
print(quotedPart)    --> it's all right
print(c)             --> "
```

(4) string.gmatch 全局字符串查找

string.gfind 函数比较适合用于范性 for 循环。他可以遍历一个字符串内所有匹配模式的子串。

```
words = {}
for w in string.gmatch("nick takes a stroll", "%a+") do
    table.insert(words, w)
end
```

URL解码

```
function unescape(s)
    s = string.gsub(s, "+", " ")
    s = string.gsub(s, "%%(%x%x)", function(h)
        return string.char(tonumber(h, 16))
    end)
    return s
end

print(unescape("a%2Bb+%3D+c")) -- a+b = c
```

对于name=value对，我们使用gfind解码，因为names和values都不能包含 ‘&’ 和 ‘=’我们可以

用模式 ‘[^&=]+’ 匹配他们:

```
cgi = {}
function decode (s)
    for name, value in string.gmatch(s, "([^&=]+)=([^&=]+)") do
        name = unescape(name)
        value = unescape(value)
        cgi[name] = value
    end
end
```

URL编码

这个函数将所有的特殊字符转换成 ‘%’ 后跟字符对应的ASCII码转换成两位的16进制数字（不足两位，前面补0），然后将空白转换为 ‘+’:

```
function escape(s)
    s = string.gsub(s, "([&=+%c])", function(c)
        return string.format("%%%02X", string.byte(c))
    end)
    s = string.gsub(s, " ", "+")
    return s
end

function encode(t)
    local s = ""
    for k, v in pairs(t) do
        s = s .. "&" .. escape(k) .. "=" .. escape(v)
    end
    return string.sub(s, 2) -- remove first '&'
end
t = {name = "al", query = "a+b = c", q = "yes or no"}

print(encode(t)) --> q=yes+or+no&query=a%2Bb+%3D+c&name=al
```

(5) string.gsub 全局字符串替换

string.gsub 函数有三个参数: 目标串，模式串，替换串，第四个参数是可选的，用来限制替换的数量。

```
print(string.gsub("nck eats fish", "fish", "chips")) --> nick eats chips 1
```

string.gsub 的第二个返回值表示他进行替换操作的次数：

```
print(string.gsub("fish eats fish", "fish", "chips")) --> chips eats chips 2
```

使用模式：

```
print(string.gsub("nick eats fish", "[AEIOUaeiou]", ".")) --> n.ck ..ts f.sh 4
```

使用捕获：

```
print(string.gsub("nick eats fish", "([AEIOUaeiou])", "(%1)")) --> n(i)ck (e)
(a)ts f(i)sh 4
```

使用替换函数：

```
function f(s)
    print("found " .. s)
end

string.gsub("Nick is taking a walk today", "%a+", f)

输出：
found Nick
found is
found taking
found a
found walk
found today
```

(6) string.sub, string.byte, string.format

```
s = "[in brackets]"
print(string.sub(s, 2, -2))     --> in brackets
```

string.char 函数和 string.byte 函数用来将字符在字符和数字之间转换，string.char 获取0个或多个整数，将每一个数字转换成字符，然后返回一个所有这些字符连接起来的字符串。
string.byte(s, i) 将字符串s的第i个字符的转换成整数。

```
print(string.char(97))                  --> a
i = 99; print(string.char(i, i+1, i+2))   --> cde
```

```
print(string.byte("abc"))                --> 97
print(string.byte("abc", 2))             --> 98
print(string.byte("abc", -1))            --> 99
```

string.format 和 C 语言的 printf 函数几乎一模一样，你完全可以照 C 语言的 printf 来使用这个函数，第一个参数为格式化串：由指示符和控制格式的字符组成。指示符后的控制格式的字符可以为：十进制'd'；十六进制'x'；八进制'o'；浮点数'f'；字符串's'。

```
print(string.format("pi = %.4f", PI)) --> pi = 3.1416
d = 5; m = 11; y = 1990
print(string.format("%02d/%02d/%04d", d, m, y)) --> 05/11/1990
tag, title = "h1", "a title"
print(string.format("<%s>%s</%s>", tag, title, tag)) --> <h1>a title</h1>
```

Table

```
table.concat
table.insert
table.maxn
table.remove
table.sort
```

(1) table.getn

```
print(table.getn{10,2,4})          --> 3
print(table.getn{10,2,nil})        --> 2
print(table.getn{10,2,nil; n=3})   --> 3
print(table.getn{n=1000})          --> 1000

a = {}
print(table.getn(a))               --> 0
table.setn(a, 10000)
print(table.getn(a))               --> 10000

a = {n=10}
print(table.getn(a))               --> 10
table.setn(a, 10000)
print(table.getn(a))               --> 10000
```

(2) table.insert, table.remove

```
table.isnert(table, value, position)
table.remove(table, position)
```

table库提供了从一个list的任意位置插入和删除元素的函数。table.insert函数在array指定位置插入一个元素，并将后面所有其他的元素后移。

```
a = {}
for line in io.lines() do
    table.insert(a, line)
end
print(table.getn(a))        --> (number of lines read)
```

table.remove 函数删除数组中指定位置的元素，并返回这个元素，所有后面的元素前移，并且数组的大小改变。不带位置参数调用的时候，他删除array的最后一个元素。

(3) table.sort

table.sort 有两个参数，存放元素的array和排序函数，排序函数有两个参数并且如果在array中排序后第一个参数在第二个参数前面，排序函数必须返回true。如果未提供排序函数，sort使用默认的小于操作符进行比较。

```
lines = {
    luaH_set = 10,
    luaH_get = 24,
    luaH_present = 48,
}

function pairsByKeys (t, f)
    local a = {}
    for n in pairs(t) do table.insert(a, n) end
    table.sort(a, f)
    local i = 0                 -- iterator variable
    local iter = function ()    -- iterator function
       i = i + 1
       if a[i] == nil then return nil
       else return a[i], t[a[i]]
       end
    end
    return iter
```

```
end

for name, line in pairsByKeys(lines) do
    print(name, line)
end
```

打印结果:

```
luaH_get          24
luaH_present      48
luaH_set          10
```

Coroutine

```
coroutine.create
coroutine.resume
coroutine.running
coroutine.status
coroutine.wrap
coroutine.yield
```

Math

```
math.abs
math.acos
math.asin
math.atan
math.atan2
math.ceil
math.cos
math.cosh
math.deg
math.exp
math.floor
math.fmod
math.frexp
math.huge
math.ldexp
math.log
```

```
math.log10
math.max
math.min
math.modf
math.pi
math.pow
math.rad
math.random
math.randomseed
math.sin
math.sinh
math.sqrt
math.tan
math.tanh
```

IO

```
io.close
io.flush
io.input
io.lines
io.open
io.output
io.popen
io.read
io.stderr
io.stdin
io.stdout
io.tmpfile
io.type
io.write
```

OS

```
os.clock
os.date
os.difftime
os.execute
os.exit
os.getenv
```

```
os.remove
os.rename
os.setlocale
os.time
os.tmpname
```

File

```
file:close
file:flush
file:lines
file:read
file:seek
file:setvbuf
file:write
```

Debug

```
debug.debug
debug.getfenv
debug.gethook
debug.getinfo
debug.getlocal
debug.getmetatable
debug.getregistry
debug.getupvalue
debug.setfenv
debug.sethook
debug.setlocal
debug.setmetatable
debug.setupvalue
debug.traceback
```

导航

- 目录
- 上一章: 函数与面向对象
- 下一章: 协程 Coroutine

协程 Coroutine

协程（coroutine）并不是 Lua 独有的概念，如果让我用一句话概括，那么大概就是：一种能够在运行途中主动中断，并且能够从中断处恢复运行的特殊函数。（嗯，其实不是函数。）

举个最原始的例子：

下面给出一个最简单的 Lua 中 coroutine 的用法演示：

```
1.  function greet()
2.      print "hello world"
3.  end
4.
5.  co = coroutine.create(greet) -- 创建 coroutine
6.
7.  print(coroutine.status(co))  -- 输出 suspended
8.  print(coroutine.resume(co))  -- 输出 hello world
9.                               -- 输出 true (resume 的返回值)
10. print(coroutine.status(co))  -- 输出 dead
11. print(coroutine.resume(co))  -- 输出 false    cannot resume dead coroutine
    (resume 的返回值)
12. print(type(co))              -- 输出 thread
```

协程在创建时，需要把协程体函数传递给创建函数 create。新创建的协程处于 suspended 状态，可以使用 resume 让其运行，全部执行完成后协程处于 dead 状态。如果尝试 resume 一个 dead 状态的，则可以从 resume 返回值上看出执行失败。另外你还可以注意到 Lua 中协程（coroutine）的变量类型其实叫做「thread」Orz…

乍一看可能感觉和线程没什么两样，但需要注意的是 resume 函数只有在 greet 函数「返回」后才会返回（所以说协程像函数）。

函数执行的中断与再开

单从上面这个例子，我们似乎可以得出结论：协程果然就是某种坑爹的函数调用方式啊。然而，协程的真正魅力来自于 resume 和 yield 这对好基友之间的羁绊。

函数 coroutine.resume(co[, val1, …])

开始或恢复执行协程 co。

如果是开始执行，val1 及之后的值都作为参数传递给协程体函数；如果是恢复执行，val1 及之后的

值都作为 yield 的返回值传递。

第一个返回值（还记得 Lua 可以返回多个值吗？）为表示执行成功与否的布尔值。如果成功，之后的返回值是 yield 的参数；如果失败，第二个返回值为失败的原因（Lua 的很多函数都采用这种错误处理方式）。

当然，如果是协程体函数执行完毕 return 而不是 yield，那么 resume 第一个返回值后跟着的就是其返回值。

函数 coroutine.yield(…)

中断协程的执行，使得开启该协程的 coroutine.resume 返回。再度调用 coroutine.resume 时，会从该 yield 处恢复执行。

当然，yield 的所有参数都会作为 resume 第一个返回值后的返回值返回。

OK，总结一下：当 co = coroutine.create(f) 时，yield 和 resume 的关系如下图：

How coroutine makes life easier

如果要求给某个怪写一个 AI：先向右走 30 帧，然后只要玩家进入视野就往反方向逃 15 帧。该怎么写？

传统做法

经典的纯状态机做法。

```
1.  -- 每帧的逻辑
2.  function Monster:frame()
3.      self:state_func()
4.      self.state_frame_count = self.state_frame_count + 1
5.  end
6.
7.  -- 切换状态
8.  function Monster:set_next_state(state)
9.      self.state_func = state
10.     self.state_frame_count = 0
11. end
12.
13. -- 首先向右走 30 帧
14. function Monster:state_walk_1()
15.     local frame = self.state_frame_count
16.     self:walk(DIRECTION_RIGHT)
```

```
    if frame > 30 then
        self:set_next_state(state_wait_for_player)
    end
end

-- 等待玩家进入视野
function Monster:state_wait_for_player()
    if self:get_distance(player) < self.range then
        self.direction = -self:get_direction_to(player)
        self:set_next_state(state_walk_2)
    end
end

-- 向反方向走 15 帧
function Monster:state_walk_2()
    local frame = self.state_frame_count;
    self:walk(self.direction)
    if frame > 15 then
        self:set_next_state(state_wait_for_player)
    end
end
```

协程做法

```
-- 每帧的逻辑
function Monster:frame()
    -- 首先向右走 30 帧
    for i = 1, 30 do
        self:walk(DIRECTION_RIGHT)
        self:wait()
    end

    while true do
        -- 等待玩家进入视野
        while self:get_distance(player) >= self.range do
            self:wait()
        end

        -- 向反方向走 15 帧
        self.direction = -self:get_direction_to(player)
        for i = 1, 15 do
            self:walk(self.direction)
```

```
            self:wait()
        end
    end
end

-- 该帧结束
function Monster:wait()
    coroutine.yield()
end
```

额外说一句，从 wait 函数可以看出，Lua 的协程并不要求一定要从协程体函数中调用 yield，这是和 Python 的一个区别。

协同程序（coroutine，这里简称协程）是一种类似于线程（thread）的东西，它拥有自己独立的栈、局部变量和指令指针，可以跟其他协程共享全局变量和其他一些数据，并且具有一种挂起（yield）中断协程主函数运行，下一次激活恢复协程会在上一次中断的地方继续执行（resume）协程主函数的控制机制。

Lua 把关于协程的所有函数放在一个名为 “coroutine” 的 table 里，coroutine 里具有以下几个内置函数：

```
-coroutine-yield [function: builtin#34]
|         -wrap [function: builtin#37]
|         -status [function: builtin#31]
|         -resume [function: builtin#35]
|         -running [function: builtin#32]
|         -create [function: builtin#33]
```

coroutine.create - 创建协程

函数 coroutine.create 用于创建一个新的协程，它只有一个以函数形式传入的参数，该函数是协程的主函数，它的代码是协程所需执行的内容

```
co = coroutine.create(function()
    io.write("coroutine create!\n")
end)
print(co)
```

当创建完一个协程后，会返回一个类型为 thread 的对象，但并不会马上启动运行协程主函数，协程的初始状态是处于挂起状态

coroutine.status - 查看协程状态

协程有 4 种状态，分别是：挂起（suspended）、运行（running）、死亡（dead）和正常（normal），可以通过 coroutine.status 来输出查看协程当前的状态。

```
print(coroutine.status(co))
```

coroutine.resume - 执行协程

函数 coroutine.resume 用于启动或再次启动一个协程的执行

```
coroutine.resume(co)
```

协程被调用执行后，其状态会由挂起（suspended）改为运行（running）。不过当协程主函数全部运行完之后，它就变为死亡（dead）状态。

传递给 resume 的额外参数都被看作是协程主函数的参数

```
co = coroutine.create(function(a, b, c)
    print("co", a, b, c)
end)
coroutine.resume(co, 1, 2, 3)
```

协程主函数执行完时，它的主函数所返回的值都将作为对应 resume 的返回值

```
co = coroutine.create(function()
    return 3, 4
end)
print(coroutine.resume(co))
```

coroutine.yield - 中断协程运行

coroutine.yield 函数可以让一个运行中的协程中断挂起

```
co = coroutine.create(function()
    for i = 1, 3 do
        print("before coroutine yield", i)
        coroutine.yield()
        print("after coroutine yield", i)
    end
end)
```

```
8.  coroutine.resume(co)
```

coroutine.resume(co)
上面第一个 resume 唤醒执行协程主函数代码，直到第一个 yield。第二个 resume 激活被挂起的协程，并从上一次协程被中断 yield 的位置继续执行协程主函数代码，直到再次遇到 yield 或程序结束。

resume 执行完协程主函数或者中途被挂起（yield）时，会有返回值返回，第一个值是 true，表示执行没有错误。如果是被 yield 挂起暂停，yield 函数有参数传入的话，这些参数会接着第一个值后面一并返回

```
1.  co = coroutine.create(function(a, b, c)
2.      coroutine.yield(a, b, c)
3.  end)
4.  print(coroutine.resume(co, 1, 2, 3))
```

以 coroutine.wrap 的方式创建协程

跟 coroutine.create 一样，函数 coroutine.wrap 也是创建一个协程，但是它并不返回一个类型为 thread 的对象，而是返回一个函数。每当调用这个返回函数，都会执行协程主函数运行。所有传入这个函数的参数等同于传入 coroutine.resume 的参数。 coroutine.wrap 会返回所有应该由除第一个（错误代码的那个布尔量） 之外的由 coroutine.resume 返回的值。 和 coroutine.resume 不同之处在于， coroutine.wrap 不会返回错误代码，无法检测出运行时的错误，也无法检查 wrap 所创建的协程的状态

```
1.  function wrap(param)
2.      print("Before yield", param)
3.      obtain = coroutine.yield()
4.      print("After yield", obtain)
5.      return 3
6.  end
7.  resumer = coroutine.wrap(wrap)
8.
9.  print(resumer(1))
10.
11. print(resumer(2))
```

coroutine.running - 返回正在运行中的协程

函数 coroutine.running 用于返回正在运行中的协程，如果没有协程运行，则返回 nil

```lua
print(coroutine.running())

co = coroutine.create(function()
    print(coroutine.running())
    print(coroutine.running() == co)
end)
coroutine.resume(co)

print(coroutine.running())
```

resume-yield 交互

下面代码放在一个 lua 文件里运行，随便输入一些字符后按回车，则会返回输出刚才输入的内容

```lua
function receive(prod)
    local status, value = coroutine.resume(prod)
    return value
end

function send(x)
    coroutine.yield(x)
end

function producer()
    return coroutine.create(function()
        while true do
            local x = io.read()
            send(x)
        end
    end)
end

function filter(prod)
    return coroutine.create(function()
--      for line = 1, math.huge do
        for line = 1, 5 do
            local x = receive(prod)
            x = string.format("%5d Enter is %s", line, x)
            send(x)
        end
    end)
```

```lua
end

function consumer(prod)
--  repeat
--      local x = receive(prod)
--      print(type(x))
--      if x then
--          io.write(x, "\n")
--      end
--  until x == nil
    while true do
        local obtain = receive(prod)
        if obtain then
            io.write(obtain, "\n\n")
        else
            break
        end
    end
end

p = producer()
f = filter(p)
consumer(f)
```

导航

- 目录
- 上一章: 标准库
- 下一章: Table 数据结构

Table 数据结构

Lua中的table不是一种简单的数据结构，它可以作为其它数据结构的基础。如数组、记录、线性表、队列和集合等，在Lua中都可以通过table来表示。

(1) 数组:

使用整数来索引table即可在Lua中实现数组。因此，Lua中的数组没有固定的大小，如:

```
1.  a = {}
2.  for i = 1, 1000 do
3.      a[i] = 0
4.  end
5.  print("The length of array 'a' is " .. #a)
6.  --The length of array 'a' is 1000
```

在Lua中，可以让任何数作为数组的起始索引，但通常而言，都会使用1作为其起始索引值。而且很多Lua的内置功能和函数都依赖这一特征，因此在没有充分理由的前提下，尽量保证这一规则。下面的方法是通过table的构造器来创建并初始化一个数组的，如:

```
1.  squares = {1, 4, 9, 16, 25}
```

(2) 二维数组:

在Lua中我们可以通过两种方式来利用table构造多维数组。其中，第一种方式通过“数组的数组”的方式来实现多维数组的，即在一维数组上的每个元素也同样为table对象，如:

```
1.  mt = {}
2.  for i = 1, N do
3.      mt[i] = {}
4.      for j = 1, M do
5.          mt[i][j] = i * j
6.      end
7.  end
```

第二种方式是将二维数组的索引展开，并以固定的常量作为第二维度的步长，如:

```
1.  mt = {}
2.  for i = 1, N do
3.      for j = 1, M do
```

```
        mt[(i - 1) * M + j] = i * j
    end
end
```

(3) 链表:

由于table是动态的实体，所以在Lua中实现链表是很方便的。其中，每个结点均以table来表示，一个“链接”只是结点中的一个字段，该字段包含对其它table的引用，如:

```
list = nil
for i = 1, 10 do
    list = { next = list, value = i}
end

local l = list
while l do
    print(l.value)
    l = l.next
end
```

(4) 队列与双向队列:

在Lua中实现队列的简单方法是使用table库函数insert和remove。但是由于这种方法会导致后续元素的移动，因此当队列的数据量较大时，不建议使用该方法。下面的代码是一种更高效的实现方式，如:

```
List = {}

function List.new()
    return {first = 0, last = -1}
end

function List.pushFront(list, value)
    local first = list.first - 1
    list.first = first
    list[first] = value
end

function List.pushBack(list, value)
    local last = list.last + 1
    list.last = last
```

```
    list[last] = value
end

function List.popFront(list)
    local first = list.first
    if first > list.last then
        error("List is empty")
    end
    local value = list[first]
    list[first] = nil
    list.first = first + 1
    return value
end

function List.popBack(list)
    local last = list.last
    if list.first > last then
        error("List is empty")
    end
    local value = list[last]
    list[last] = nil
    list.last = last - 1
    return value
end
```

(5) 集合和包(Bag):

在Lua中用table实现集合是非常简单的，见如下代码:

```
reserved = { ["while"] = true, ["end"] = true, ["function"] = true, }
if not reserved["while"] then
    --do something
end
```

在Lua中我们可以将包(Bag)看成MultiSet，与普通集合不同的是该容器中允许key相同的元素在容器中多次出现。下面的代码通过为table中的元素添加计数器的方式来模拟实现该数据结构，如:

```
function insert(bag, element)
    bag[element] = (bag[element] or 0) + 1
end

function remove(bag, element)
```

```
    local count = bag[element]
    bag[element] = (count and count > 1) and count - 1 or nil
end
```

(6) StringBuilder:

如果想在Lua中将多个字符串连接成为一个大字符串的话，可以通过如下方式实现，如:

```
local buff = ""
for line in io.lines() do
    buff = buff .. line .. "\n"
end
```

上面的代码确实可以正常的完成工作，然而当行数较多时，这种方法将会导致大量的内存重新分配和内存间的数据拷贝，由此而带来的性能开销也是相当可观的。事实上，在很多编程语言中String都是不可变对象，如Java，因此如果通过该方式多次连接较大字符串时，均会导致同样的性能问题。为了解决该问题，Java中提供了StringBuilder类，而Lua中则可以利用table的concat方法来解决这一问题，见如下代码:

```
local t = {}
for line in io.lines() do
    t[#t + 1] = line .. "\n"
end
local s = table.concat(t)

--concat方法可以接受两个参数，因此上面的方式还可以改为：
local t = {}
for line in io.lines() do
    t[#t + 1] = line
end
local s = table.concat(t,"\n")
```

导航

- 目录
- 上一章: 协程 Coroutine
- 下一章: 常用的 C API

常用的 C API

基础概念

states

Lua连接库是完全可重入的，因为它没有全局变量。Lua解释器的整个state（如全局变量、堆栈等）都存储在一个结构类型为Lua_State动态分配的对象里。指向这一对象的指针必须作为第一个参数传递给所有连接库的API，除了用来生成一个Lua state的函数—lua_open。在调用所有的API函数之前，你必须先用lua_open以生成一个state:

```
1.  lua_State* lua_open(void);
```

可以通过调用lua_close来释放一个通过lua_open生成的state:

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

这一函数销毁给定的Lua_State中的所有对象并释放state所占用的动态内存（如果有必要的话将通过调用对应的垃圾收集元方法来完成），在某些平台上，你不必调用这个函数，因为当宿主程序退出时会释放所有的资源，换句话说，长期运行的程序，如守护进程或web服务器，应尽快释放state所占的资源，以避免其过于庞大。

堆栈与索引

Lua使用虚拟堆栈机制和C程序互相传值，所有的堆栈中的元素都可以看作一个Lua值（如nil, number, string等）。

当Lua调用C函数时，被调用的C函数将得到一个新的堆栈。这一堆栈与之前调用此函数的堆栈无关，也有其它C函数的堆栈无关。这一新的堆栈用调用C函数要用到的参数初始化，同时，这一堆栈也被用以返回函数调用结果。

为了便于操作，在API的中大量操作都并不依从堆栈只能操作栈顶元素的严格规则。而通过索引引用堆栈的任一元素。一个正整数索引可以看作某一元素在堆栈中的绝对位置（从1开始计数），一个负整数索引可以看作某一元素相对于栈顶的偏移量。

特别地，如果堆栈中有n个元素，那么索引1指向第一个元素（即第一个压入栈的元素）索引n指向最后一个元素；反过来，索引-1指向最后一个元素（即栈顶元素）索引-n指向第一个元素。当一个索引大于1并小于n时我们称其为一个有效索引（即1 <= abs(index) <= top）。

接口解析

lua_newstate

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

创建一个新的独立 state，不能创建返回 NULL。形参 f 是 allocator 函数，Lua 通过这个函数来为这个 state 分配内存。第二个形参 ud，是一个透明指针，每次调用时，Lua简单地传给 allocator 函数。

lua_open/lua_close

lua_open 被 lua_newstate 替换，可以使用luaL_newstate从标准库中创建一个标准配置的state，如: lua_State *L = luaL_newstate(); 。

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

销毁指定的 state 中所有的对象，并释放指定的 state 中使用的所有动态内存。

lua_load/lua_call/lua_pcall/lua_cpcall

这些函数的目的就是让我们能够执行压入栈中的函数，该函数可能是lua中定义的函数,可能是C++重定义的函数，当然我们一般是用来执行lua中执行的函数,C++中定义的基本上可以直接调用的。

```
1.  int lua_load (lua_State *L,
2.                lua_Reader reader,
3.                void *data,
4.                const char *chunkname);
5.
6.  void lua_call(lua_State *L, int nargs, int nresults);
7.  void lua_pcall(lua_State *L, int nargs, int nresults, int errfunc);
8.  void lua_cpcall(lua_State *L, int nargs, int nresults, int errfunc, void *ud);
```

L是执行环境，可以理解为当前栈，nargs参数个数,nresults返回值个数。lua_pcall和该函数区别是多一个参数，用于发生错误处理时的代码返回。lua_cpcall则又多一个用于传递用户自定义的数据结构的指针。

lua_call的运行是无保护的，他与lua_pcall相似，但是在错误发生的时候她抛出错误而不是返回错误代码。当你在应用程序中写主流程的代码时，不应该使用 lua_call，因为你应该捕捉任何可能发生的错误。当你写一个函数的代码时，使用lua_call是比较好的想法，如果有错误发生，把错误留给关

心她的人去处理。所以，写应用程序主流程代码用lua_pcall，写C Native Function代码时用lua_call。

示例**1:**

Lua 代码:

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

C 代码:

```
1.  lua_getfield(L, LUA_GLOBALSINDEX, "f"); /* function to be called */
2.  lua_pushstring(L, "how");                        /* 1st argument */
3.  lua_getfield(L, LUA_GLOBALSINDEX, "t");   /* table to be indexed */
4.  lua_getfield(L, -1, "x");        /* push result of t.x (2nd arg) */
5.  lua_remove(L, -2);                  /* remove 't' from the stack */
6.  lua_pushinteger(L, 14);                          /* 3rd argument */
7.  lua_call(L, 3, 1);     /* call 'f' with 3 arguments and 1 result */
8.  lua_setfield(L, LUA_GLOBALSINDEX, "a");        /* set global 'a' */
```

在上面的例子除了描述了lua_call的使用外，还对lua_getfield的使用有一定的参考价值。特别是学习如何在一个表中获取他的值。

在上面的例子中，可能再调用lua_getfield时就会忘记调用lua_remove,当然这是我想象自己使用时会犯下的错。lua_getfield函数功能是从指定表中取出指定元素的值并压栈。上面获取t.x的值的过程就是先调用:

```
1.  lua_getfield(L, LUA_GLOBALSINDEX, "t");
```

从全局表中获取t的值，然而t本身是一个表，现在栈顶的值是t表。于是再一次调用:

```
1.  lua_getfield(L, -1, "x");
```

从t中取出x的值放到栈上，-1表示栈顶。那该函数执行完成后t的位置由-1就变成-2了，所以下面一句lua_remove 索引的是-2，必须把t给remove掉，否则栈中就是4个参数了。上面的最后一句lua_setfield 的目的是把返回值取回赋给全局变量a，因为在**lua_call**执行完成后，栈顶的就是返回值了。

示例**2:**

```
1.  //test.lua
2.  function printmsg()
```

```
    print("hello world")
end
x = 10

//test.c
#include <stdio.h>
#include <unistd.h>

#include <lua.h>
#include <lauxlib.h>
#include <lualib.h>

int main(int argc, const char *argv[]) {
    lua_State *L;
    if(NULL == (L = luaL_newstate())) {
        perror("luaL_newstate failed");
        return -1;
    }
    luaL_openlibs(L);
    if(luaL_loadfile(L, "./test.lua")) {
        perror("loadfile failed");
        return -1;
    }
    lua_pcall(L, 0, 0, 0);

    lua_getglobal(L, "printmsg");
    lua_pcall(L, 0, 0, 0);

    lua_close(L);
    return 0;
}
```

上面的代码就是在test.c中调用test.lua的函数printmsg函数。

对于上面的C代码，我想大家都知道几个函数的大概作用：

- luaL_newstate(): 创建一个新的Lua虚拟机
- luaL_openlibs(): 打开一些必要的库，比如print等
- luaL_loadfile(): 手册上写的是”This function uses lua_load to load the chunk in the filenamed filename.” 而lua_load就是把编译过的chunk放在stack的顶部。理解chunk很重要，后面会具体讲到
- lua_pcall: 执行栈上的函数调用

一开始我一直认为既然 luaL_loadfile 执行以后，就可以直接用 lua_getglobal 获得test.lua中的函数，其实不然。手册中明确提到，**lua_load**把一个**lua**文件当作一个**chunk**编译后放到**stack**的栈顶，而什么是**chunk**呢？**chunk**就是一个可执行语句的组合，可以是一个文件也可以是一个**string**，“Lua handles a chunk as the body of an anonymous function with a variable number of arguments”这是Lua对chunk也就是lua文件的处理方式，就是认为是一个可变参数的匿名函数。也就是说，调用后栈上有一个匿名函数，这个函数的body就是文件中所有的内容。

在 luaL_loadfile 后，调用 lua_gettop 以及 lua_type 可以知道栈的大小为1，放在栈上的是一个 function 类型的value。为什么 loadfile 后我们不能直接获取到 printmsg 这个函数呢，那是因为刚才提到的，loadfile仅仅视编译lua文件，并不执行这个文件，也就是说只是在栈上形成了一个匿名函数。只有执行这个函数一次，才会使得printmsg可以通过 lua_getglobal 获取，否则，全局变量是空的。我在手册上看到这样一句话：Lua在执行函数的时候，函数会实例化，获得的 closure 也是这个函数的最终值。其实不管是函数，还是其他类型，如果不执行的话，它们只是被编译，并不能在进程的空间种获取到他们，感觉就像c的库一样，他们的编译文件.so已经存在，但是如果你不调用它，那么库中所有的变量不能被实例化，调用者也就无法访问。其实pringmsg和x本质是一样的，只是他们类型不同而已。

lua_getfield/lua_setfield

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

把值 t[k] 压入堆栈，t 是给定有效的索引 index 的值，和在 Lua 中一样，这个函数可能会触发元方法 index 事件。

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

相当于 t[k] = v，t 是给定的有效索引 index 的值，v 是堆栈顶部的值，这个函数会弹出这个值，和在 Lua 中一样，这个函数可能会触发 newindex 元方法事件。

lua_getglobal/lua_setglobal

lua_getglobal

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

把全局 name 的值压入栈顶，它被定义为宏(macro):

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

lua_setglobal

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

从栈中弹出一个值并赋值给全局 name，它被定义成宏(macro):

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

lua_gettop/lua_settop/lua_pop

在任何时候，你都可以通过调用lua_gettop函数取得栈顶元素的索引:

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

因为索引从1开始计数，lua_gettop的返回值等于这个堆栈的元素个数（当堆栈为空时返回值为0）

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

lua_settop用于把堆栈的栈顶索引设置为指定的数值，它可以接受所有可接受索引。如果新的栈顶索引比原来的大，则新的位置用nil填充。如果index为0，则将删除堆栈中的所有元素。在lua.h中定义了如下一个宏:

```
1.  #define lua_pop(L,n) lua_settop(L,-(n)-1)
```

用以把堆栈上部的n个元素删除。

lua_pushvalue/lua_insert/lua_remove/lua_replace

```
1.   void lua_pushvalue (lua_State* L, int index);
2.   void lua_remove (lua_State* L, int index);
3.   void lua_insert (lua_State* L, int index);
4.   void lua_replace (lua_State* L, int index);
```

lua_pushvalue压入一个元素的值拷贝到指定的索引处，相反地，lua_remove删除给定索引的元素，并将之一索引之上的元素来填补空缺。同样地，lua_insert在上移给定索引之上的所有元素后再在指定位置插入新元素。Lua_replace将栈顶元素压入指定位置而不移动任何元素（因此指定位置的元素的值被替换）。这些函数都仅接受有效索引（你不应当使用假索引调用lua_remove或lua_insert，因为它不能解析为一个堆栈位置）。下面是一个例子，栈的初始状态为10 20 30 40 50 （从栈底到栈顶，“”标识为栈顶，有:

```
lua_pushvalue(L, 3)    --> 10 20 30 40 50 30*
lua_pushvalue(L, -1)   --> 10 20 30 40 50 30 30*
lua_remove(L, -3)      --> 10 20 30 40 30 30*
lua_remove(L,  6)      --> 10 20 30 40 30*
lua_insert(L,  1)      --> 30 10 20 30 40*
lua_insert(L, -1)      --> 30 10 20 30 40*  (没影响)
lua_replace(L, 2)      --> 30 40 20 30*
lua_settop(L, -3)      --> 30 40*
lua_settop(L,  6)      --> 30 40 nil nil nil nil*
```

lua_gettable/lua_settable

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

把 t[k] 压入堆栈，t 是给出的有效的索引 index 的值，k 是栈顶的值，这个函数会从堆栈中弹出 key，并将结果值放到它的位置，和在 Lua 一样，函数可能会触发一个元方法 index 事件。

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

相当于 t[k]=v，t 是给出的有效的索引 index 的值，v 是堆栈顶部的值，k 是堆栈顶部下面的值。这个函数会从堆栈中弹出 key 和 value 的值，和在 Lua 中一样，函数可能会触发元方法 newindex 事件。

lua_concat

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

用来连接字符串，等价于Lua中的..操作符：自动将数字转换成字符串，如果有必要的时候还会自动调用metamethods。另外，她可以同时连接多个字符串。调用lua_concat(L,n)将连接(同时会出栈)栈顶的n个值，并将最终结果放到栈顶。

lua_type/lua_typename

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

lua_type返回堆栈元素的值类型，当使用无效索引时返回LUA_TNONE（如当堆栈为空的时候），lua_type返回的类型代码为如下在lua.h中定义的常量：LUA_TNIL，LUA_TNUMBER，LUA_TBOOLEAN，LUA_TSTRING，LUA_TTABLE，LUA_TFUNCTION，LUA_USERDATA，LUA_TTHEARD，LUA_TLIGHTUSERDATA。下面的函数可以将这些常量转换为字符串：

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

lua_checkstack

当你使用Lua API的时候，你有责任控制堆栈溢出。函数

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

将把堆栈的尺寸扩大到可以容纳top+extra个元素；当不能扩大堆栈尺寸到这一尺寸时返回假。这一函数从不减小堆栈的尺寸；当前堆栈的尺寸大于新的尺寸时，它将保留原来的尺寸，并不变化。

lua_is*

```
int lua_isnumber(lua_State *L, int index);
int lua_isboolean(lua_State *L, int index);
int lua_isfunction(lua_State *L, int index);
int lua_istable(lua_State *L, int index);
int lua_isstring(lua_State *L, int index);
int lua_isnil(lua_State *L, int index);
int lua_iscfunction(lua_State *L, int index);
```

带lua_is*前辍的函数在当堆栈元素对象与给定的类型兼容时返回1，否则返回0。Lua_isboolean是个例外，它仅在元素类型为布尔型时成功（否则没有意思，因为任何值都可看作布尔型）。当使用无效索引时，它们总返回0。Lua_isnumber接受数字或者全部为数字的字符串；lua_isstring打接受字符串和数值，lua_isfunction接受lua函数和C函数；lua_isuserdata也可接受完全和轻量级两种userdata。如果想区分C函数和lua函数，可以使用lua_iscfunction函数；同样地，想区分完全和轻量级userdata可以使用lua_islightuserdata；区分数字和数字组成的字符串可以使用lua_type。

API函数中还有比较堆栈中的两个值 的大小的函数：

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

lua_equal和lua_lessthan与相对应的lua操作符等价（参考2.5.2）。lua_rawequal直接判断两个值的原始值，而非通过调用元方法来比较。以上的函数当索引无效时返回0。

lua_to*

```
int lua_toboolean(lua_State *L, int index);
lua_CFunction lua_tocfunction(lua_State *L, int index);
lua_Integer lua_tointeger(lua_State *L, int index);
const char *lua_tolstring(lua_State *L, int index);
lua_Number lua_tonumber(lua_State *L, int index);
void *lua_topointer(lua_State *L, int index);
lua_State *lua_tothread(lua_State *L, int index);
const char *lua_tostring(lua_State *L, int index);
```

这些函数可通过任意可接受索引调用，如果用无效索引为参数，则和给定值并不匹配类型一样。lua_toboolean转换指定索引lua值为C“布尔型”值（0或1）。当lua值仅为false或nil时返回0（如果你仅想接受一个真正的布尔值，可以先使用lua_isboolean去测试这个值的类型。

lua_tonumber转换指定索引的值为数字（lua_Number默认为double）。这一lua值必须数字或可转换为数字的字符串（参考2.2.1），否则lua_tonumber返回0。

lua_tostring将指定索引的值转换为字符串（const char*）。lua值必须为字符串或数字，否则返回NULL。当值为数字，lua_tostring将会把堆栈的原值转换为字符串（当lua_tostring应用到键值上时会使lua_next出现难以找出原因的错误）。lua_tostring返回一个完全对齐的字符串指针，这一字符串总是’/0’结尾（和C一样），但可能含有其它的0。如果你不知道一个字符串有多少个0，你可以使用lua_strlen取得真实长度。因为lua有垃圾收集机制，因此不保证返回的字符串指针在对应的值从堆栈中删除后仍然有效。如果你以后还要用到当前函数返回的字符串，你应当备份它或者将它放到registry中（参考3.18）。

lua_tofunction将堆栈中的值转换为C函数指针，这个值必须为C函数指针，否则返回NULL。数据类型lua_CFunction将在3.16节讲述。

lua_tothread转换堆栈中的值为lua线程（以lua_State*为表现形式），此值必须是一个线程，否则返回NULL。

lua_topointer转换堆栈中的值为通用C指针（void*）。这个值必须为userdata、表、线程或函数，否则返回NULL。lua保证同一类型的不同对象返回不同指针。没有直接方法将指针转换为原值，这一函数通常用以获取调试信息。

lua_push*

```
void lua_pushboolean(lua_State *L, int b);
void lua_pushcclosure (lua_State *L, lua_CFunction fn, int n);
void lua_pushcfunction(lua_State *L, lua_CFunction f);
const char *lua_pushfstring (lua_State *L, const char *fmt, ...);
void lua_pushinteger (lua_State *L, lua_Integer n);
void lua_pushliteral
```

```
7.  void lua_pushlstring(lua_State *L, const char *s, size_t len);
8.  void lua_pushnil(lua_State *L);
9.  void lua_pushnumber(lua_State *L, lua_Number n);
10. void lua_pushstring(lua_State *L, const char *s);
11. const char *lua_pushvfstring (lua_State *L,
12.                               const char *fmt,
13.                               va_list argp);
```

这些函数接受一个C值，并将其转换为对应的lua值，然后将其压入堆栈。lua_pushlstring和lua_pushstring对给定的字符串生成一个可以互转的拷贝，这是个例外。lua_pushstring能压C字符串（即以0结尾并且内部没有0），否则建议使用更通用的lua_pushlstring，它能指定长度。

你同样可以压入“格式化”字符串：

```
1.  const char *lua_pushfstring  (lua_State *L, const char *fmt, ...);
2.  const char *lua_pushvfstring (lua_State *L, const char *fmt, va_list argp);
```

这两个函数向堆栈压入格式化字符串并返回指向字符串的指针。它们跟sprintf和vsprintf很象但有如下的重要不同：

- 你不用申请内存去保存格式化结果，这结果是一个lua字符串并且lua自己会小心管理内存（并通过垃圾收集机制释放）。
- 使用转义字符受限。它们没有标志量、宽度和精确度。转义字符能够是’%%’（插入一个”%”）、’%s’（插入一个以0结尾的字符串）、’%f’（插入一个lua_Number）、’%d’（插入一个int）和’%c’（插入一个用int表示的字符）。

lua_register

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

设置 C 函数 f 为新的全局变量 name 的值，它被定义为宏(macro):

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

完整示例

```
1.  #include <stdio.h>
2.  #include <string.h>
3.  #include <lua.hpp>
4.  #include <lauxlib.h>
5.  #include <lualib.h>
```

```
6.
7.  void
8.  load(lua_State *L, const char *fname, int *w, int *h) {
9.      if (luaL_loadfile(L, fname) || lua_pcall(L, 0, 0 ,0)) {
10.         printf("Error Msg is %s.\n", lua_tostring(L, -1));
11.         return;
12.     }
13.     lua_getglobal(L, "width");    // #define lua_getglobal(L,s) lua_getfield(L,
    LUA_GLOBALSINDEX, (s))
14.     lua_getglobal(L, "height");
15.     if (!lua_isnumber(L, -2)) {
16.         printf("'width' should be a number\n");
17.         return;
18.     }
19.     if (!lua_isnumber(L, -1)) {
20.         printf("'height' should be a number\n", );
21.         return;
22.     }
23.     *w = lua_tointeger(L, -2);
24.     *h = lua_tointeger(L, -1);
25. }
26.
27. int
28. main() {
29.     lua_State *L = luaL_newstate();
30.     int w, h;
31.     load(L, "D:/test.lua", &w, &h);
32.     printf("width = %d, height = %d\n", w, h);
33.     lua_close(L);
34.     return 0;
35. }
```

导航

- 目录
- 上一章: Table 数据结构
- 下一章: Lua 与 C/C++ 交互

Lua 与 C/C++ 交互

绑定Lua和C/C++的库

- CPPlua
- tolua
- tolua++
- luawrapper
- luabind
- luaplus

Lua调用C/C++

简介

Lua（念“鲁啊”）作为一门发展成熟的脚本语言,正在变得越来越流行。它也可以作为和C/C++执行脚本交互的语言。并且Lua的整个库很小，Lua 5.1版本整个静态链接的lua.dll才164KB，所以Lua很轻量，特别适合轻量级脚本嵌入。

这节要讲Lua和C/C++的交互—Lua通过C/C++导出的dll来调用。

LUA调用C文件中的函数方法

- C中注册函数

```
1.    lua_pushcfunction(l, l_sin); //注册在lua中使用的c函数l_sin
2.    lua_setglobal(l, "mysin");   //设定绑定到lua中的名字为mysin
```

- C中提供的函数其定义要符合:

  ```
  typedef int function(lua_State *L)
  ```

准备工作

安装完Lua，需要在Visual Studio中配置Lua路径，使得你的编译器能搜寻到。关于VS2010的配置，见我的博文《VS2010 C++目录配置》一文。完成后新建一个Dll工程便可以了。

我们用一个在Lua中显示Windows对话框的程序来简要介绍一下，程序虽小，但五脏俱全。程序如下:

```
1.  // 将一些有用的Win32特性导出
```

```
// 以便在Lua中使用
extern "C"
{
#include <lua.h>
#include <lualib.h>
#include <lauxlib.h>
#pragma comment(lib, "lua.lib")
};

#include <Windows.h>
#include <iostream>
using namespace std;

static const char* const ERROR_ARGUMENT_COUNT = "参数数目错误！";
static const char* const ERROR_ARGUMENT_TYPE  = "参数类型错误！";

// 发生错误,报告错误
void ErrorMsg(lua_State* luaEnv, const char* const pszErrorInfo)
{
    lua_pushstring(luaEnv, pszErrorInfo);
    lua_error(luaEnv);
}

// 检测函数调用参数个数是否正常
void CheckParamCount(lua_State* luaEnv, int paramCount)
{
    // lua_gettop获取栈中元素个数.
    if (lua_gettop(luaEnv) != paramCount)
    {
        ErrorMsg(luaEnv, ERROR_ARGUMENT_COUNT);
    }
}

// 显示Windows对话框.
// @param [in] pszMessage string 1
// @param [in] pszCaption string 2
extern "C" int ShowMsgBox(lua_State* luaEnv)
{
    const char* pszMessage = 0;
    const char* pszCaption = 0;

    // 检测参数个数是否正确.
```

```
    CheckParamCount(luaEnv, 2);

    // 提取参数.
    pszMessage = luaL_checkstring(luaEnv, 1);
    pszCaption = luaL_checkstring(luaEnv, 2);

    if (pszCaption && pszMessage)
    {
        ::MessageBox(
            NULL,
            pszMessage,
            pszCaption,
            MB_OK | MB_ICONINFORMATION
            );
    }
    else
    {
        ErrorMsg(luaEnv, ERROR_ARGUMENT_TYPE);
    }

    // 返回值个数为0个.
    return 0;
}

// 导出函数列表.
static luaL_Reg luaLibs[] =
{
    {"ShowMsgBox", ShowMsgBox},
    {NULL, NULL}
};

// Dll入口函数, Lua调用此Dll的入口函数.
extern "C" __declspec(dllexport)
int luaopen_WinFeature(lua_State* luaEnv)
{
    const char* const LIBRARY_NAME = "WinFeature";
    luaL_register(luaEnv, LIBRARY_NAME, luaLibs);

    return 1;
}
```

程序解析

首先我们包含Lua的头文件，并链入库文件。注意：Lua的头文件为C风格，所以用external “C”来含入。在此例中，我们最终的导出函数为“ShowMsgBox”。

每一个导出函数的格式都为：

```
1.  extern “C”int Export_Proc_Name(luaState* luaEnv);
```

其中，luaState*所指的结构中包含了Lua调用此Dll时必备的Lua环境。那么Lua向函数传递参数该怎么办呢？实际上是用luaL_check[type]函数来完成的。如下：

```
1.  const char* pHelloStr = luaL_checkstring(luaEnv, 1);
2.  double value = luaL_checknumber(luaEnv, 2);
3.  int ivalue = luaL_checkint(luaEnv, 3);
```

luaL_check系列函数的第二个参数是Lua调用该函数时传递参数从坐到右的顺序（从1开始）。

然后我们看到，static的一个luaL_Reg的结构数组中包含了所有要导出的函数列表。最后通过luaopen_YourDllName的一个导出函数来完成一系列操作。YourDllName就是你最终的Dll的名字（不含扩展名）。因为你在Lua中调用此Dll时，Lua会根据此Dll名字找luaopen_YourDllName对应的函数，然后从此函数加载该Dll。

Dll入口函数格式如下：

```
1.  extern "C" __declspec(dllexport)
2.  int luaopen_WinFeature(lua_State* luaEnv)
3.  {
4.      const char* const LIBRARY_NAME = "WinFeature";
5.      luaL_register(luaEnv, LIBRARY_NAME, luaLibs);
6.
7.      return 1;
8.  }
```

我们通过luaL_register将LIBRARY_NAME对应的库名，以及luaL_Reg数组对应的导出列表来注册到lua_State*对应的Lua环境中。

Lua调用

那么我们要如何调用该Dll呢？首先，把该Dll放到你的Lua能搜寻到的目录——当前目录、Lua安装目录下的clibs目录，然后通过require函数导入。

因为Lua中如果你的函数调用参数只有一个，并且该参数为字符串的话，函数调用时的括号是可以省略的，所以：require(“YourLibName”)和requir“YourLibName”都是合法的。我们把刚刚生成的

WinFeature.dll文件拷贝到C盘下，然后在C盘启动Lua。示例如下：

```
> require "WinFeature"
> for k, v in pairs(WinFeature) do
>>        print(k, v)
>> end
ShowMsgBox functon:0028AB90
>
```

可以看到，函数调用方式都是“包名.函数名”，而包名就是你的Dll的名字。我们可以用下面的方式查看一个包中的所有函数：

```
for k, v in pairs(PackageName) do
    print(k, v)
end
```

然后我们调用WinFeature.ShowMsgBox函数：

```
> WinFeature.ShowMsgBox("Hello, this is a msgBox", "Tip")
```

会弹出对话框显示内容为”Hello, this is a msgBox”和标题为”Tip”。

Lua堆栈详解

唔，那么lua_State结构如何管理Lua运行环境的呢？Lua又是如何将参数传递到C/C++函数的呢？C/C++函数又如何返回值给Lua呢？……这一切，都得从Lua堆栈讲起。

Lua在和C/C++交互时，Lua运行环境维护着一份堆栈——不是传统意义上的堆栈，而是Lua模拟出来的。Lua与C/C++的数据传递都通过这份堆栈来完成，这份堆栈的代表就是lua_State*所指的那个结构。

堆栈结构解析

堆栈通过lua_push系列函数向堆栈中压入值，通过luaL_check系列从堆栈中获取值。而用luaL_check系列函数时传递的参数索引，比如我们调用WinFeature.ShowMsgBox(“Hello”, “Tip”)函数时，栈结构如下：

栈顶
“Tip” 2或者-1
“Hello” 1或者-2
栈底

其中，参数在栈中的索引为参数从左到右的索引（从1开始），栈顶元素索引也可以从-1记起。栈中元素个数可以用lua_gettop来获得，如果lua_gettop返回0，表示此栈为空。（lua_gettop这个函数名取得不怎么样！）

提取参数

luaL_check系列函数在获取值的同时，检测这个值是不是符合我们所期望的类型，如果不是，则抛出异常。所有这个系列函数如下：

```
1. luaL_checkany          —— 检测任何值（可以为nil）
2. luaL_checkint          —— 检测一个值是否为number（double），并转换成int
3. luaL_checkinteger      —— 检测一个值是否为number（double），并转换
   成lua_Integer（prtdiff_t），在我的机子上，ptrdiff_t被定义为int
4. luaL_checklong         —— 检测一个值是否为number（double），并转换成long
5. luaL_checklstring      —— 检测一个值是否为string，并将字符串长度传递在[out]参数中返回
6. luaL_checknumber       —— 检测一个值是否为number（double）
7. luaL_checkstring       —— 检测一个值是否为string并返回
8. luaL_checkudata        —— 检测自定义类型
```

传递返回值

当我们要传递返回值给Lua时，可以用lua_push系列函数来完成。每一个导出函数都要返回一个int型整数，这个整数是你的导出函数的返回值的个数。而返回值通过lua_push系列函数压入栈中。比如一个Add函数：

```
1. extern “C” int Add(lua_State* luaEnv)
2. {
3.     CheckParamCount(luaEnv, 2);
4.
5.     double left = luaL_checknumber(luaEnv, 1);
6.     double right = luaL_checknumber(luaEnv, 2);
7.
8.     double result = left + right;
9.     lua_pushnumber(luaEnv, result);
10.
11.    return 1;
12. }
```

可以看出，我们用lua_pushnumber把返回值压入栈，最后返回1——1代表返回值的个数。lua_push系列函数如下：

```
1. lua_pushboolean        —— 压入一个bool值
```

```
lua_pushcfunction      — 压入一个lua_CFunction类型的C函数指针
lua_pushfstring        — 格式化一个string并返回，类似于sprintf
lua_pushinteger        — 压入一个int
lua_pushlightuserdata  — 压入自定义数据类型
lua_pushliteral        — 压入一个字面值字符串
lua_pushlstring        — 压入一个规定长度内的string
lua_pushnil            — 压入nil值
lua_pushnumber         — 压入lua_Number（double）值
lua_pushstring         — 压入一个string
lua_pushthread         — 压入一个所传递lua_State所对应的线程，如果此线程是主线程，则返回1
lua_pushvalue          — 将所传递索引处的值复制一份压入栈顶
lua_pushvfstring       — 类似lua_pushfstring
```

通过这些函数，我们可以灵活的使用C/C++的高性能特性，来导出函数供Lua调用。

C/C++调用Lua脚本

简介

C调用LUA文件中的函数方法

```
lua_getglobal(L, <function name>) //获取lua中的函数
lua_push*() //调用lua_push系列函数，把输入参数压栈。例如lua_pushnumber(L, x)
lua_pcall(L, <arguments nums>, <return nums>, <错误处理函数地址>)
```

例如：

```
lua_settop(m_pLua,0);
lua_getglobal(m_pLua,"mainlogic");
lua_pushlstring(m_pLua,(char*)msg.getBuf(),msg.size());
int ret = 0;
ret = lua_pcall(m_pLua,1,4,0);
```

上一节介绍了如何在Lua中调用C/C++代码，本节介绍如何在C/C++中调用Lua脚本。本节介绍一个例子，通过Lua来生成一个XML格式的便签。便签格式如下：

```
<?xml version="1.0" encoding="utf-8" ?>
<note>
    <fromName>发送方姓名</fromName>
    <toName>接收方姓名</toName>
```

```
    <sendTime>发送时间</sendTime>
    <msgContent>便签内容</msgContent>
</note>
```

我们通过C/C++来输入这些信息，然后让Lua来生成这样一个便签文件。

Lua代码

```lua
xmlHead = '<?xml version="1.0" encoding="utf-8" ?>\n'

-- Open note file to wriet.
function openNoteFile(fileName)
    return io.open(fileName, "w")
end

-- Close writed note file.
function closeNoteFile(noteFile)
    noteFile:close()
end

function writeNestedLabel(ioChanel, label, nestCnt)
    if nestCnt == 0 then
        ioChanel:write(label)
        return
    end

    for i = 1, nestCnt do
        ioChanel:write("\t")
    end

    ioChanel:write(label)
end

function generateNoteXML(fromName, toName, msgContent)
    local noteFile = openNoteFile(fromName .. "_" .. toName .. ".xml")
    if not noteFile then
        return false
    end

    noteFile:write(xmlHead)
    noteFile:write("<note>\n")

```

```
    local currNestCnt = 1
    writeNestedLabel(noteFile, "<fromName>", currNestCnt)
    noteFile:write(fromName)
    writeNestedLabel(noteFile, "</fromName>\n", 0)

    writeNestedLabel(noteFile, "<toName>", currNestCnt)
    noteFile:write(toName)
    writeNestedLabel(noteFile, "</toName>\n", 0)

    local sendTime = os.time()
    writeNestedLabel(noteFile, "<sendTime>", currNestCnt)
    noteFile:write(sendTime)
    writeNestedLabel(noteFile, "</sendTime>\n", 0)

    writeNestedLabel(noteFile, "<msgContent>", currNestCnt)
    noteFile:write(msgContent)
    writeNestedLabel(noteFile, "</msgContent>\n", 0)

    noteFile:write("</note>\n")
    closeNoteFile(noteFile)

    return true
end
```

我们通过openNoteFile和closeNoteFile来打开/关闭XML文件。generateNoteXML全局函数接受发送方姓名、接收方姓名、便签内容，生成一个XML便签文件。便签发送时间通过Lua标准库os.time()函数来获取。writeNestedLabel函数根据当前XML的缩进数目来规范XML输出格式。此文件很好理解，不再赘述。

C++调用Lua脚本

```
extern "C"
{
#include <lua.h>
#include <lualib.h>
#include <lauxlib.h>
#pragma comment(lib, "lua.lib")
};

#include <iostream>
#include <string>
using namespace std;
```

```
12.
13. // 初始化Lua环境.
14. lua_State* initLuaEnv()
15. {
16.     lua_State* luaEnv = lua_open();
17.     luaopen_base(luaEnv);
18.     luaL_openlibs(luaEnv);
19.
20.     return luaEnv;
21. }
22.
23. // 加载Lua文件到Lua运行时环境中
24. bool loadLuaFile(lua_State* luaEnv, const string& fileName)
25. {
26.     int result = luaL_loadfile(luaEnv, fileName.c_str());
27.     if (result)
28.     {
29.         return false;
30.     }
31.
32.     result = lua_pcall(luaEnv, 0, 0, 0);
33.     return result == 0;
34. }
35.
36. // 获取全局函数
37. lua_CFunction getGlobalProc(lua_State* luaEnv, const string& procName)
38. {
39.     lua_getglobal(luaEnv, procName.c_str());
40.     if (!lua_iscfunction(luaEnv, 1))
41.     {
42.         return 0;
43.     }
44.
45.     return lua_tocfunction(luaEnv, 1);
46. }
47.
48. int main()
49. {
50.     // 初始化Lua运行时环境.
51.     lua_State* luaEnv = initLuaEnv();
52.     if (!luaEnv)
53.     {
```

```
54.         return -1;
55.     }
56.
57.     // 加载脚本到Lua环境中.
58.     if (!loadLuaFile(luaEnv, ".\\GenerateNoteXML.lua"))
59.     {
60.         cout<<"Load Lua File FAILED!"<<endl;
61.         return -1;
62.     }
63.
64.     // 获取Note信息.
65.     string fromName;
66.     string toName;
67.     string msgContent;
68.
69.     cout<<"Enter your name:"<<endl;
70.     cin>>fromName;
71.
72.     cout<<"\nEnter destination name:"<<endl;
73.     cin>>toName;
74.
75.     cout<<"\nEnter message content:"<<endl;
76.     getline(cin, msgContent);
77.     getline(cin, msgContent);
78.
79.     // 将要调用的函数和函数调用参数入栈.
80.     lua_getglobal(luaEnv, "generateNoteXML");
81.     lua_pushstring(luaEnv, fromName.c_str());
82.     lua_pushstring(luaEnv, toName.c_str());
83.     lua_pushstring(luaEnv, msgContent.c_str());
84.
85.     // 调用Lua函数（3个参数,一个返回值）.
86.     lua_pcall(luaEnv, 3, 1, 0);
87.
88.     // 获取返回值.
89.     if (lua_isboolean(luaEnv, -1))
90.     {
91.         int success = lua_toboolean(luaEnv, -1);
92.         if (success)
93.         {
94.             cout<<"\nGenerate Note File Successful!"<<endl;
95.         }
```

```
96.     }
97.
98.     // 将返回值出栈.
99.     lua_pop(luaEnv, 1);
100.
101.    // 释放Lua运行时环境.
102.    lua_close(luaEnv);
103.
104.    system("pause");
105.    return 0;
106. }
```

代码解析

初始化Lua运行时环境

lua_State*所指向的结构中封装了Lua的运行时环境。我们用initLuaEnv这个函数来初始化。lua_open函数用来获取一个新环境，然后我们用luaopen_base打开Lua的基础库，然后用luaL_openlibs打开Lua的io、string、math、table等高级库。

加载Lua文件

然后我们用luaL_loadfile和lua_pcall来将一个Lua脚本加载到对应的Lua运行时环境中—注意：并不自动执行，只是加载。这两个函数如果返回非0，表示加载失败—你的Lua脚本文件可能未找到或Lua脚本有语法错误……

加载Lua函数

我们用lua_getglobal函数将Lua脚本中的全局函数、全局变量等入栈，放在栈顶。

压入Lua函数调用参数

我们用lua_push系列函数来将要调用函数所需参数全部入栈，入栈顺序为Lua函数对应参数从左到右的顺序。

调用Lua函数

最后用lua_pcall来调用函数。Lua_pcall函数原型如下：

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

其中，L为此函数执行的Lua环境，nargs为此函数所需的参数个数，nresults为此函数的返回值个数，errfunc为发生错误时错误处理函数在堆栈中的索引。

获取Lua函数返回值

然后，我们可以通过检测栈顶的位置（从-1开始），来获取返回值。获取返回值后，用lua_pop将栈顶元素出栈—出栈个数为返回值个数。

释放Lua环境

最后，通过lua_close函数来关闭Lua环境并释放资源。

运行结果

我们将GenerateNoteXML.lua脚本和最终的C++生成的GenerateNoteXML.exe放在同一路径下，并运行GenerateNoteXML.exe，在此目录下会生成一个XML文件。如下：

```
Enter your name:
Jack

Enter destnation name:
Joe

Enter message content:
Hello, Can you help me?

Generate Note File Successful!
```

生成的arnozhang_YaFengZhang.xml文件如下：

```
<?xml version="1.0" encoding="utf-8" ?>
<note>
  <fromName>Jack</fromName>
  <toName>Joe</toName>
  <sendTime>1317971623</sendTime>
  <msgContent>Hello, Can you help me?</msgContent>
</note>
```

C 作为动态库文件被 Lua 调用

C/C++中的入口函数定义

一定是要定义成：
luaopen_(dll或so文件的文件名称)，(dll或so文件的文件名称)必须和dll或so文件名称保持一

致。

例如(C++ windows情况):

```
#ifdef _WIN32
#define _EXPORT extern "C" __declspec(dllexport)
#else //unix/linux
#define _EXPORT extern "C"
#endif
_EXPORT int luaopen_capi_mytestlib(lua_State *L)
{
    struct luaL_reg driver[] = {
        {"average", average1},
        {NULL, NULL},};
    luaL_register(L, "mylib", driver);
    //luaL_openlib(L, "mylib", driver, 0);
    return 1;
}
```

动态库要供LUA调用的function

其定义要符合:

```
typedef int function(lua_State *L)
```

在动态库调用LUA注册

将要调用的函数放到这个结构体里:

```
struct luaL_Reg lib[] ={}
```

在动态库的入口函数里调用luaL_register将这个结构体注册，在这个入口函数注册结构体时，要注册成:

```
luaL_register(L,"XXX",lib);
```

在写脚本的时候，使用require(“XXX”)

就是入口函数的luaopen_后面的XXX，注意大小写敏感

编译生成的动态库命令成XXX.so或XXX.dll(win)

同入口函数的luaopen_后面的XXX一致

示例：

C文件如下：

```
#include <stdio.h>
#include "lua/lua.h"
#include "lua/lualib.h"
#include "lua/lauxlib.h"
static int add(lua_State *L)
{
    int a,b,c;
    a = lua_tonumber(L,1);
    b = lua_tonumber(L,2);
    c = a+b;
    lua_pushnumber(L,c);
    printf("test hello!!!\r\n");
    return 1;
}
static const struct luaL_Reg lib[] =
{
    {"testadd",add},
    {NULL,NULL}
};

int luaopen_testlib(lua_State *L)
{
    luaL_register(L,"testlib",lib);
    return 1;
}
```

编译：gcc test.c -fPIC -shared -o testlib.so

lua脚本编写：

```
require("testlib")
c = testlib.testadd(15,25)
print("The result is ",c);
```

示例：

```
int lua_createmeta (lua_State *L, const char *name, const luaL_reg *methods) {
```

```
    if (!luaL_newmetatable (L, name))
        return 0;

    luaL_openlib (L, NULL, methods, 0);

    lua_pushliteral (L, "__gc");
    lua_pushcfunction (L, methods->func);
    lua_settable (L, -3);
    lua_pushliteral (L, "__index");
    lua_pushvalue (L, -2);
    lua_settable (L, -3);
    lua_pushliteral (L, "__metatable");
    lua_pushliteral (L, "you're not allowed to get this metatable");
    lua_settable (L, -3);
    return 1;
}
```

示例中的luaopen_testlib函数替换为:

```
int luaopen_testlib(lua_State *L)
{
    lua_createmeta(L,"testlib",lib);
    return 1;
}
```

导航

- 目录
- 上一章: 常用的 C API
- 下一章: 编译 Lua 字节码

编译 Lua 字节码

导航

- 目录
- 上一章: Lua 与 C/C++ 交互
- 下一章: LuaJIT 介绍

LuaJIT 介绍

导航

- 目录
- 上一章: 编译 Lua 字节码
- 下一章: Lua 5.1 程序接口

附录一 Lua 5.1 程序接口

Lua functions

```
_G
_VERSION

assert
collectgarbage
dofile
error
getfenv
getmetatable
ipairs
load
loadfile
loadstring
module
next
pairs
pcall
print
rawequal
rawget
rawset
require
select
setfenv
setmetatable
tonumber
tostring
type
unpack
xpcall

coroutine.create
coroutine.resume
coroutine.running
coroutine.status
coroutine.wrap
```

```
coroutine.yield

debug.debug
debug.getfenv
debug.gethook
debug.getinfo
debug.getlocal
debug.getmetatable
debug.getregistry
debug.getupvalue
debug.setfenv
debug.sethook
debug.setlocal
debug.setmetatable
debug.setupvalue
debug.traceback

file:close
file:flush
file:lines
file:read
file:seek
file:setvbuf
file:write

io.close
io.flush
io.input
io.lines
io.open
io.output
io.popen
io.read
io.stderr
io.stdin
io.stdout
io.tmpfile
io.type
io.write

math.abs
math.acos
```

```
math.asin
math.atan
math.atan2
math.ceil
math.cos
math.cosh
math.deg
math.exp
math.floor
math.fmod
math.frexp
math.huge
math.ldexp
math.log
math.log10
math.max
math.min
math.modf
math.pi
math.pow
math.rad
math.random
math.randomseed
math.sin
math.sinh
math.sqrt
math.tan
math.tanh

os.clock
os.date
os.difftime
os.execute
os.exit
os.getenv
os.remove
os.rename
os.setlocale
os.time
os.tmpname

package.cpath
```

```
package.loaded
package.loaders
package.loadlib
package.path
package.preload
package.seeall

string.byte
string.char
string.dump
string.find
string.format
string.gmatch
string.gsub
string.len
string.lower
string.match
string.rep
string.reverse
string.sub
string.upper

table.concat
table.insert
table.maxn
table.remove
table.sort
```

C API

```
lua_Alloc
lua_CFunction
lua_Debug
lua_Hook
lua_Integer
lua_Number
lua_Reader
lua_State
lua_Writer

lua_atpanic
```

```
lua_call
lua_checkstack
lua_close
lua_concat
lua_cpcall
lua_createtable
lua_dump
lua_equal
lua_error
lua_gc
lua_getallocf
lua_getfenv
lua_getfield
lua_getglobal
lua_gethook
lua_gethookcount
lua_gethookmask
lua_getinfo
lua_getlocal
lua_getmetatable
lua_getstack
lua_gettable
lua_gettop
lua_getupvalue
lua_insert
lua_isboolean
lua_iscfunction
lua_isfunction
lua_islightuserdata
lua_isnil
lua_isnone
lua_isnoneornil
lua_isnumber
lua_isstring
lua_istable
lua_isthread
lua_isuserdata
lua_lessthan
lua_load
lua_newstate
lua_newtable
lua_newthread
```

```
54. lua_newuserdata
55. lua_next
56. lua_objlen
57. lua_pcall
58. lua_pop
59. lua_pushboolean
60. lua_pushcclosure
61. lua_pushcfunction
62. lua_pushfstring
63. lua_pushinteger
64. lua_pushlightuserdata
65. lua_pushliteral
66. lua_pushlstring
67. lua_pushnil
68. lua_pushnumber
69. lua_pushstring
70. lua_pushthread
71. lua_pushvalue
72. lua_pushvfstring
73. lua_rawequal
74. lua_rawget
75. lua_rawgeti
76. lua_rawset
77. lua_rawseti
78. lua_register
79. lua_remove
80. lua_replace
81. lua_resume
82. lua_setallocf
83. lua_setfenv
84. lua_setfield
85. lua_setglobal
86. lua_sethook
87. lua_setlocal
88. lua_setmetatable
89. lua_settable
90. lua_settop
91. lua_setupvalue
92. lua_status
93. lua_toboolean
94. lua_tocfunction
95. lua_tointeger
```

```
lua_tolstring
lua_tonumber
lua_topointer
lua_tostring
lua_tothread
lua_touserdata
lua_type
lua_typename
lua_upvalueindex
lua_xmove
lua_yield
```

auxiliary library

```
luaL_Buffer
luaL_Reg

luaL_addchar
luaL_addlstring
luaL_addsize
luaL_addstring
luaL_addvalue
luaL_argcheck
luaL_argerror
luaL_buffinit
luaL_callmeta
luaL_checkany
luaL_checkint
luaL_checkinteger
luaL_checklong
luaL_checklstring
luaL_checknumber
luaL_checkoption
luaL_checkstack
luaL_checkstring
luaL_checktype
luaL_checkudata
luaL_dofile
luaL_dostring
luaL_error
luaL_getmetafield
```

```
luaL_getmetatable
luaL_gsub
luaL_loadbuffer
luaL_loadfile
luaL_loadstring
luaL_newmetatable
luaL_newstate
luaL_openlibs
luaL_optint
luaL_optinteger
luaL_optlong
luaL_optlstring
luaL_optnumber
luaL_optstring
luaL_prepbuffer
luaL_pushresult
luaL_ref
luaL_register
luaL_typename
luaL_typerror
luaL_unref
luaL_where
```

导航

- 目录
- 上一章: LuaJIT 介绍
- 下一章: 附录二 Lua 5.2 程序接口

附录二 Lua 5.2 程序接口

Lua functions

```
_G
_VERSION

assert
collectgarbage
dofile
error
getmetatable
ipairs
loadfile
load
next
pairs
pcall
print
rawequal
rawget
rawlen
rawset
require
select
setmetatable
tonumber
tostring
type
xpcall

bit32.arshift
bit32.band
bit32.bnot
bit32.bor
bit32.btest
bit32.bxor
bit32.extract
bit32.lrotate
bit32.lshift
```

```
bit32.replace
bit32.rrotate
bit32.rshift

coroutine.create
coroutine.resume
coroutine.running
coroutine.status
coroutine.wrap
coroutine.yield

debug.debug
debug.getuservalue
debug.gethook
debug.getinfo
debug.getlocal
debug.getmetatable
debug.getregistry
debug.getupvalue
debug.setuservalue
debug.sethook
debug.setlocal
debug.setmetatable
debug.setupvalue
debug.traceback
debug.upvalueid
debug.upvaluejoin

file:close
file:flush
file:lines
file:read
file:seek
file:setvbuf
file:write

io.close
io.flush
io.input
io.lines
io.open
io.output
```

```
io.popen
io.read
io.stderr
io.stdin
io.stdout
io.tmpfile
io.type
io.write

math.abs
math.acos
math.asin
math.atan
math.atan2
math.ceil
math.cos
math.cosh
math.deg
math.exp
math.floor
math.fmod
math.frexp
math.huge
math.ldexp
math.log
math.max
math.min
math.modf
math.pi
math.pow
math.rad
math.random
math.randomseed
math.sin
math.sinh
math.sqrt
math.tan
math.tanh

os.clock
os.date
os.difftime
```

```
os.execute
os.exit
os.getenv
os.remove
os.rename
os.setlocale
os.time
os.tmpname

package.config
package.cpath
package.loaded
package.loadlib
package.path
package.preload
package.searchers
package.searchpath

string.byte
string.char
string.dump
string.find
string.format
string.gmatch
string.gsub
string.len
string.lower
string.match
string.rep
string.reverse
string.sub
string.upper

table.concat
table.insert
table.pack
table.remove
table.sort
table.unpack
```

C API

```
lua_Alloc
lua_CFunction
lua_Debug
lua_Hook
lua_Integer
lua_Number
lua_Reader
lua_State
lua_Unsigned
lua_Writer

lua_absindex
lua_arith
lua_atpanic
lua_call
lua_callk
lua_checkstack
lua_close
lua_compare
lua_concat
lua_copy
lua_createtable
lua_dump
lua_error
lua_gc
lua_getallocf
lua_getctx
lua_getfield
lua_getglobal
lua_gethook
lua_gethookcount
lua_gethookmask
lua_getinfo
lua_getlocal
lua_getmetatable
lua_getstack
lua_gettable
lua_gettop
lua_getupvalue
lua_getuservalue
lua_insert
lua_isboolean
```

```
lua_iscfunction
lua_isfunction
lua_islightuserdata
lua_isnil
lua_isnone
lua_isnoneornil
lua_isnumber
lua_isstring
lua_istable
lua_isthread
lua_isuserdata
lua_len
lua_load
lua_newstate
lua_newtable
lua_newthread
lua_newuserdata
lua_next
lua_pcall
lua_pcallk
lua_pop
lua_pushboolean
lua_pushcclosure
lua_pushcfunction
lua_pushfstring
lua_pushinteger
lua_pushlightuserdata
lua_pushliteral
lua_pushlstring
lua_pushnil
lua_pushnumber
lua_pushstring
lua_pushthread
lua_pushvalue
lua_pushvfstring
lua_rawequal
lua_rawget
lua_rawgeti
lua_rawlen
lua_rawset
lua_rawseti
lua_rawgetp
```

```
lua_rawsetp
lua_register
lua_remove
lua_replace
lua_resume
lua_setallocf
lua_setfield
lua_setglobal
lua_sethook
lua_setlocal
lua_setmetatable
lua_settable
lua_settop
lua_setupvalue
lua_setuservalue
lua_status
lua_toboolean
lua_tocfunction
lua_tointeger
lua_tointegerx
lua_tolstring
lua_tonumber
lua_tonumberx
lua_topointer
lua_tostring
lua_tothread
lua_tounsigned
lua_tounsignedx
lua_touserdata
lua_type
lua_typename
lua_upvalueid
lua_upvalueindex
lua_upvaluejoin
lua_version
lua_xmove
lua_yield
lua_yieldk
```

auxiliary library

```
luaL_Buffer
luaL_Reg

luaL_addchar
luaL_addlstring
luaL_addsize
luaL_addstring
luaL_addvalue
luaL_argcheck
luaL_argerror
luaL_buffinit
luaL_buffinitsize
luaL_callmeta
luaL_checkany
luaL_checkinteger
luaL_checkint
luaL_checklong
luaL_checklstring
luaL_checknumber
luaL_checkoption
luaL_checkstack
luaL_checkstring
luaL_checktype
luaL_checkudata
luaL_checkunsigned
luaL_checkversion
luaL_dofile
luaL_dostring
luaL_error
luaL_execresult
luaL_fileresult
luaL_getmetafield
luaL_getmetatable
luaL_getsubtable
luaL_gsub
luaL_len
luaL_loadbuffer
luaL_loadbufferx
luaL_loadfile
luaL_loadfilex
luaL_loadstring
luaL_newlib
```

```
43. luaL_newlibtable
44. luaL_newmetatable
45. luaL_newstate
46. luaL_openlibs
47. luaL_optinteger
48. luaL_optint
49. luaL_optlong
50. luaL_optlstring
51. luaL_optnumber
52. luaL_optstring
53. luaL_optunsigned
54. luaL_prepbuffer
55. luaL_prepbuffsize
56. luaL_pushresult
57. luaL_pushresultsize
58. luaL_ref
59. luaL_requiref
60. luaL_setfuncs
61. luaL_setmetatable
62. luaL_testudata
63. luaL_tolstring
64. luaL_traceback
65. luaL_typename
66. luaL_unref
67. luaL_where
```

导航

- 目录
- 上一章: 附录一 Lua 5.1 程序接口