Lua/Programming in Lua

@于龙

目录

[1 开始 3](#_Toc418249478)

[1.1 程序块 3](#_Toc418249479)

[1.2 词法规范 3](#_Toc418249480)

[1.3 全局变量 3](#_Toc418249481)

[2 类型与值 4](#_Toc418249482)

[2.1 nil 4](#_Toc418249483)

[2.2 boolean 4](#_Toc418249484)

[2.3 number 4](#_Toc418249485)

[2.4 string 4](#_Toc418249486)

[2.5 table 5](#_Toc418249487)

[2.6 function 6](#_Toc418249488)

[2.7 userdata和thread 6](#_Toc418249489)

[3 表达式 6](#_Toc418249490)

[3.1 算术表达式 6](#_Toc418249491)

[3.2 关系操作符 6](#_Toc418249492)

[3.3 逻辑操作符 7](#_Toc418249493)

[3.4 字符串连接 7](#_Toc418249494)

[3.5 table构造式 7](#_Toc418249495)

[4 语句 7](#_Toc418249496)

[4.1 赋值 7](#_Toc418249497)

[4.2 局部变量与块 8](#_Toc418249498)

[4.3 控制结构 8](#_Toc418249499)

[4.4 break与return 9](#_Toc418249500)

[5 函数 9](#_Toc418249501)

[5.1 多重返回值 10](#_Toc418249502)

[5.2 变长参数 10](#_Toc418249503)

[5.3 具名实参 11](#_Toc418249504)

[6 深入函数 11](#_Toc418249505)

[6.1 closure闭合函数 11](#_Toc418249506)

[6.2 非全局的函数 12](#_Toc418249507)

[6.3 正确的尾调用 13](#_Toc418249508)

[7 迭代器与泛型for 13](#_Toc418249509)

[7.1 泛型for的语义 14](#_Toc418249510)

[7.2 无状态的迭代器 14](#_Toc418249511)

[7.3 具有复杂状态的迭代器 15](#_Toc418249512)

[8 编译、执行与错误 15](#_Toc418249513)

[8.1 编译 15](#_Toc418249514)

[8.2 C代码 15](#_Toc418249515)

[8.3 错误 15](#_Toc418249516)

[8.4 错误处理与异常 16](#_Toc418249517)

[8.5 错误消息与追溯 16](#_Toc418249518)

[9 数据文件与持久性 16](#_Toc418249519)

[9.1 数据文件 16](#_Toc418249520)

[10 元表与元方法 16](#_Toc418249521)

[10.1 算术类元方法 17](#_Toc418249522)

[10.2 关系类元方法 17](#_Toc418249523)

[10.3 库定义的元方法 17](#_Toc418249524)

[10.4 table访问的元方法 17](#_Toc418249525)

[11 环境 18](#_Toc418249526)

[11.1 具有动态名字的全局变量 18](#_Toc418249527)

[11.2 全局变量声明 18](#_Toc418249528)

[11.3 非全局的环境 18](#_Toc418249529)

[12 面向对象编程 18](#_Toc418249530)

[12.1 类 19](#_Toc418249531)

[12.2 继承 19](#_Toc418249532)

[12.3 多重继承 19](#_Toc418249533)

[12.4 私密性 20](#_Toc418249534)

[13 弱引用table 21](#_Toc418249535)

[13.1 备忘录函数 21](#_Toc418249536)

[13.2 对象属性 21](#_Toc418249537)

[14 调试库 22](#_Toc418249538)

[14.1 自省机制 22](#_Toc418249539)

[14.1.1 访问局部变量 22](#_Toc418249540)

[14.1.2 访问非局部变量 22](#_Toc418249541)

[14.2 钩子 22](#_Toc418249542)

[14.3 性能剖析 22](#_Toc418249543)

# 开始

## 程序块

一个程序块就是一连串的语句或命令.

可以使用也可以不使用;分隔语句：

|  |
| --- |
| a = 1  b = 2  等价  a = 1 b = 2 |

动态加载文件：

|  |
| --- |
| dofile(“file.lua”) |

## 词法规范

标示符：任意字母、数字、下划线，不能以数字开头.

避免使用下划线跟多个大写字母标示符，如“\_VERSION”，Lua保留变量.

Lua标示符区分大小写.

注释：

|  |
| --- |
| 行注释：  --print(a)  块注释  --[[  print(a)  --]]  取消块注释  ---[[ |

## 全局变量

访问未声明全局变量为nil；

设置全局变量为nil删除；

# 类型与值

Lua没有类型定义的语法，每个值都携带了类型信息.

Lua中的8种基本类型：

|  |
| --- |
| nil  boolean  number  string  userdata  function  thread  table |

type返回类型名，返回值为string，与上述类型名对应：

|  |
| --- |
| print(type(nil)) -- nil  Print(type(type)) -- function |

## nil

nil是一种类型，只有nil一个值，用于区别其他任何值.

变量未赋值前为nil，赋值nil删除变量.

## boolean

可选值：false和true.

Lua视false和nil为假，而其他值视为真.

数字0和空字符串也视为真.

## number

表示实数，可以表示任何32位整数，不会产生四舍五入错误.

## string

Lua中的字符串通常表示为一个字符序列，完全采用8位编码，字符串中的字符可以具有任何编码，包括数值0.

可以将任意二进制数据存储到一个字符串中.

Lua中字符串是不可变值.

支持双引号、单引号界定，良好的编码习惯应坚持使用一种.

ASCII数值制定转移字符：

|  |
| --- |
| \<ddd>  \97 - a  \49 - 1 |

双括号界定字符串，并不会处理其中的转义：

|  |
| --- |
| a = [[  a  b  c  ]]  print(a) |

Lua提供运行时字符串与数字自动转换：

|  |
| --- |
| print(“10” + 1) -- 11  print(“hello” + 1) -- 错误  print(“10” + “10”) -- 20  print(10 .. 20) -- 1020  10 == “10” -- false |

直接在数字后使用..需要加空格，否则视为小数点.

转换函数：

|  |
| --- |
| tonumber -- 失败返回nil  tostring tostring(“10”) == “10” -- true |

获取字符串长度#：

|  |
| --- |
| a = “hello”  print(#a) -- 5 |

## table

Lua通过table实现了关联数组，除了nil都可以作为索引.

table没有固定大小，Lua中唯一的数据结构机制.

Lua也用table来表示模块、包、对象.

table构造：

|  |
| --- |
| a = {} |

语法糖：

|  |
| --- |
| a[“x”]  a.x |

Lua习惯以1作为索引的起始位置.

#长度操作符返回一个数组或线性表的最后一个索引值.

习惯写法：

|  |
| --- |
| a[#a] --最后一个值  a[#a] = nil --删除最后一个值  a[#a+1] = v --添加到列表末尾 |

Lua将nil作为界定数组结尾的标志，当数组有空隙时，#操作符会认为这就是结尾.

|  |
| --- |
| a = {}  a[1000] = 1  #a --0 table.maxn(a) --1000 |

## function

Lua中函数是作为第一类值.

函数可以存储在变量中，可以通过参数传递给其他函数.

## userdata和thread

userdata可以将任意C语言类型存入Lua变量.只能进行赋值和相等性测试.

# 表达式

## 算术表达式

^计算指数：

|  |
| --- |
| x^0.5 -- x的平方根 |

%取模运算：

|  |
| --- |
| x%1 -- x为实数，则取小数  x – x%1 -- 取整数  x – x%0.01 -- 取精确到小数点后两位 |

## 关系操作符

不等于~=，等于==.

先比较类型，类型不同，则视为不同.

nil只与其自身相等.

对于talbe、userdata、function，比较的是引用.

只能对数字和字符串做大小比较.

## 逻辑操作符

and or not.

false和nil视为假，其他为真.

and和or都使用短路求值.

not永远只返回true or false：

|  |
| --- |
| not nil -- true  not false -- true  not 0 -- false  not not nil -- false |

## 字符串连接

..连接字符串，遇到数字会转为字符串.

## table构造式

构造：

|  |
| --- |
| a = {}  a = {“sunday”, “monday”}  a = {x = 1,y = 1}  a = {x = 1, {y = 1}}  a = {[“x”] = 1} |

构造式中还可以用;代替,

# 语句

## 赋值

Lua允许多重赋值：

|  |
| --- |
| a,b = 10,20  错误：  a,b,c = 0 |

多重赋值中，先对等号右边所有元素求值，然后才执行赋值，可以用来交换变量：

|  |
| --- |
| x,y = y,x |

等号右边值不足，则为nil，多余则忽略.

## 局部变量与块

局部变量的作用域限制与控制结构的执行体、函数的执行体或一个程序块

|  |
| --- |
| a = 1 全局  local b = a 局部 |

do … end 可用于控制局部变量：

|  |
| --- |
| do  local x = 1  end |

访问局部变量比全局变量更快.

一种技巧将全局变量保存在局部变量，加速其访问：

|  |
| --- |
| local foo = foo |

## 控制结构

if then else

|  |
| --- |
| if a == 1 then  elseif a == 2 then  else  end |

Lua不支持switch.

while

|  |
| --- |
| while a == 1 do  end |

repeat

|  |
| --- |
| repeat  until a == 1 |

重复执行，直到条件为真，至少执行一次.

循环体内的局部变量作用域包括了条件测试：

|  |
| --- |
| repeat  local x = 0  until x > 0 |

数字型for

|  |
| --- |
| for var=exp1,exp2,exp3 do  end |

var从exp1变化到exp2（小于等于），每次以exp3为递增，exp3默认为1.

exp1~exp2在开始时一次求值，声明为局部变量.

不要在循环中修改控制变量的值，否则导致不可预知的后果.

泛型for

通过循环一个迭代器函数来遍历所有值：

|  |
| --- |
| for I,v in ipairs(a) do  end  for k in pairs(a) do  end |

Lua基础库提供ipairs、pairs，用于遍历数组的迭代器函数.

## break与return

break只能跳出包含它的内部循环.

break与return只能是一个块的最后一条语句.

|  |
| --- |
| function foo()  do return end  ….被注释  end |

# 函数

Lua为面向对象式的调用提供了一种特殊的语法，o.foo(o,x)的冒号操作符写法：o:foo(x).

冒号操作符使o作为函数隐含的第一个参数.

对于Lua函数与C函数，在调用上没有区别.

实参多于形参，舍弃多余实参；实参不足，则多余的形参初始化为nil.

|  |
| --- |
| n = n or 1  n为nil时，n = 1 |

## 多重返回值

Lua允许函数返回多个结果.

只有当一个函数调用是一系列表达式中的最后一个元素时，才能获得所有值：

|  |
| --- |
| function foo()  return 1,2  end  x,y = foo(),20 -- x = 1, y = 20  t = {foo()} -- t = {1,2} |

一系列表达式的四种情况：

多重赋值、函数调用的实参列表、table构造式、return.

unpack函数：接受一个数组作为参数，并从下标1开始返回所有元素：

|  |
| --- |
| lua实现：  function unpack (t, i)  I = I or 1  if t[i] then  return [i]; unpack(t, i+1)  end  end |

## 变长参数

Lua中的函数还可以接受不同数量的实参.

|  |
| --- |
| function add(…)  local s = 0  for I,v in ipairs(…) do  s = s + v  end  return s  end  local a,b,c = … |

selector访问变长参数，数字为n返回第n个参数；#返回变长参数的总和，包括参数里的nil.

|  |
| --- |
| for I = 1,selector(“#”,…) do  local a = selector(I,…)  end |

## 具名实参

将所有的实参组织到一个table中，将table作为参数传递给函数.

# 深入函数

Lua函数特殊性：函数是第一类值，具有自己的词法域。

函数可以存储到变量中或table中，可以作为实参传递给其他函数，还可以作为返回值。

Lua函数可以嵌套在另一个函数里，内部的函数可以访问外部函数中的变量。

Lua函数与其他所有值一样都是匿名的，当讨论一个函数名时（如print），实际上是讨论一个持有某个函数的变量.

一个函数定义实际就是一条赋值语句：

|  |
| --- |
| function foo(x)  end  等价  foo = function(x)  end |

## closure闭合函数

将一个函数写在另一个函数之内，位于内部的函数便可以访问外部函数中的局部变量，这项特征称为词法域。

|  |
| --- |
| function newCounter()  local I = 0  return function ()  I = I + 1  return i  end  end  c1 = newCounter()  print(c1()) -- 1  print(c1()) -- 2 |

一个closure就是一个函数加上该函数所需访问的所有“非局部的变量”，如果再次调用newCounter，那么它会创建一个新的局部变量i，从而得到一个新的closure.

|  |
| --- |
| c2 = newCounter()  print(c2()) -- 1  print(c2()) -- 2 |

c1和c2是同一个函数所创建的两个不同的closure，它们拥有各自的局部变量i的独立实例.

从技术上讲，Lua中只有closure，而不存在函数.

closure的另一种常用情况，重新定义某些函数或预定义的函数，构建安全的沙盒环境：

|  |
| --- |
| do  local oldOpen = io.open  io.open = function ()  end  end |

## 非全局的函数

存储在table中：

|  |
| --- |
| Lib = {}  Lib.foo = function() end  Lib = { foo = function() end}  function Lib.foo ()  end |

存储在局部变量中：

|  |
| --- |
| local f = function ()  end  local function f()  end |

定义递归的局部函数时，需要先声明：

|  |
| --- |
| local fact  fact = function (n)  if n == 0 then  return 1  else  return n \* fact(n – 1\_  end  end |

## 正确的尾调用

Lua支持尾调用消除.

当一个函数调用是另一个函数的最后一个动作时，该调用才算是尾调用.

当尾调用后，程序不需要保存任何关于该函数的栈信息了，使得尾调用不消耗任何栈空间.

使用尾调用一个程序可以拥有无数嵌套的尾调用，不会造成栈溢出.

尾调用形式：

|  |
| --- |
| function f(x)  return g(x)  end  错误：  function f(x)  g(x)  end  错误：  return g(x) + 1  return x or g(x)  return (g(x)) |

# 迭代器与泛型for

迭代器就是一种可以遍历一种集合中所有元素的机制.

迭代器需要在每次成功调用之间保持一些状态，以存储位置及如何进入下一个位置.

|  |
| --- |
| function values(t)  local I = 0  return function () I = I + 1  return t[i]  end  end  for e in values()  end |

泛型for在内部保存了迭代器函数，在每次更新迭代时调用迭代器，并在迭代器返回nil结束循环.

## 泛型for的语义

泛型for语法：

|  |
| --- |
| for <var-list> in <exp-list> do  end |

变量列表第一个值为nil时循环结束，使用的是最后一个表达式的返回值.

for做的第一件事是对in后面的表达式求值.

此表达式可返回迭代器函数、恒定状态、控制变量三个值.

for之后使用恒定状态、控制变量来调用迭代器函数.

等价代码：

|  |
| --- |
| do  local \_f,\_s,\_var = <exp-list>  while true do  local var1,…varn = \_f(s, \_var)  \_var = var1  if \_var == nil then break end  end  end |

## 无状态的迭代器

迭代器自身不保存任何状态，可以在多个循环中使用一个迭代器，避免创建新的closure的开销.

ipairs的实现：

|  |
| --- |
| local function iter(a,i)  I = I + 1  local v = a[i]  if v then  return I,v  end  end  function ipairs(a)  return iter,a,0  end |

## 具有复杂状态的迭代器

除了恒定状态和控制变量外，还有其他需要保存的状态.

两种解决方案：使用closure、使用table存储恒定状态.

基于clousure的方式比table方式更为高效.

# 编译、执行与错误

## 编译

loadfile会从一个文件加载Lua代码块，但它不会运行，只是编译代码，然后将编译结果作为一个函数返回.

loadfile不会引发错误，只返回错误值，不处理.

dofile的伪实现：

|  |
| --- |
| function dofile(filename)  local f = assert(loadfile(filename))  return f()  end |

loadfile多次运行同一个代码比dofile性能要高，并且具有错误处理.

loadstring，总是在全局环境中运行.

## C代码

Lua通常不会包含任何无法通过ANSI C来实现的机制，但为动态链接在所有平台提供了实现.

package.loadlib

|  |
| --- |
| local f = package.loadlib(path, “luaopen\_socket”) |

## 错误

|  |
| --- |
| n = assert(io.read(“xxx”), “invalid”) |

assert函数检查第一个参数是否为true，为true则返回该参数，否则就引发一个错误.

第二个参数为可选字符串.

error(“”)函数用于引发一个错误.

对于返回nil还是引发一个错误的原则：易于避免的异常应引发一个错误.

## 错误处理与异常

要在Lua中处理错误，则必须使用pcall来执行：

|  |
| --- |
| function foo()  end  if pcall(foo) then  正常，返回true和函数返回值  else  错误，返回false及错误消息  end |

错误消息可以是任何类型的，传递给error函数.

## 错误消息与追溯

只要错误消息是一个字符串，Lua就会附加一些关于错误发生位置的信息：错误文件及行号.

error第二个函数可用于指明层级.

xpcall可提供错误发生时的调用栈.

# 数据文件与持久性

## 数据文件

通过使用table构造式来定义一种文件格式.将数据作为Lua代码来输出.

实现方法：

|  |
| --- |
| Entry{“xxxxx”}  Entry{“xxxxx”}  Entry{}与Entry({})等价 |

# 元表与元方法

可以通过修改元表来修改一个值的行为.

Lua中的每个值都有一个元表.table和userdata可以有各自独立的元表，而其他类型的值则共享类型所属的单一元表.

Lua在创建新的table时不会创建元表.

|  |
| --- |
| t = {}  print(getmetatable(t)) - nil  setmetatable(xx,xx) |

任何table都可以作为任何值的元表.

在Lua代码中，只能设置table的元表，若要设置其他类型的值的元表，必须通过C代码来完成.

标准的字符串库为所有的字符串都设置了一个元表.其他类型默认都没有元表.

## 算术类元方法

Lua按以下步骤查找元表：如果第一个值有元表且存在元方法，则调用；反之，则查找第二个值的元方法，如果都没有则引发一个错误.

## 关系类元方法

与算术类不同，关系类元方法不能应用于混合的类型,

## 库定义的元方法

print 🡪 \_\_tostring

\_\_metatable 禁止访问和修改元方法

## table访问的元方法

当访问一个table中不存在的字段时，得到的结果为nil.

这会促使解释器去查找一个叫\_\_index的元方法.

函数实现方式：

|  |
| --- |
| mt = {}  mt.\_\_index = function (table, key)  end |

\_\_index元方法不一定是函数，还可以是一个table来实现.

rawget会绕过元表，进行原始的访问.

\_\_newindex元方法用于table的更新.当对一个table中不存在的索引赋值时，解释器就会查找\_\_newindex元方法，如果存在元方法就会调用，而不是执行赋值.

如果\_\_newindex是个table，解释器就在此table中执行赋值，而不是对原来的table.

rawset会绕过这个元方法.

# 环境

Lua将其所有的全局变量保存在一个常规的table中，这个table称为环境.

这个table存储在全局变量\_G中.

## 具有动态名字的全局变量

|  |
| --- |
| \_G[varname]  代替  value = loadstring(“return ”.. varname)() |

## 全局变量声明

通过对\_G设置元表，来限制全局变量的声明.

检测对不存在的全局变量的访问.

在元表中使用debug.getinfo来确定是主程序块还是Lua函数的声明.

不允许全局变量设置为nil.

## 非全局的环境

可以通过函数setfenv来改变一个函数的环境，该函数的参数是一个函数和一个新的环境table.第一个参数还可以通过数字来指定调用栈的层次，1为本函数，2为上一级函数.

两种保留\_G的方式：

|  |
| --- |
| setfenv(1, {\_G = \_G})  local newgt = {}  setmetatable(newgt, {\_\_index = \_G})  setfenv(1,newgt) |

# 面向对象编程

Lua中的table就是一种对象.

|  |
| --- |
| function Account.withdraw(self, v)  self.balance = self.balance - v  end  a1 = Account;  a1.withdraw(a1, 100) |

Lua的冒号写法：

|  |
| --- |
| function Account:withdraw(v)  self.balance = self.balance - v  end  a1 = Account;  a1:withdraw(100) |

冒号的作用是在一个方法定义中添加一个额外的隐藏参数，以及在一个方法调用中添加一个额外的实参，并没有引入新东西.

## 类

原型也是一种常规的对象，当其他对象遇到一个未知操作时，原型会先查找它.

类和原型都是一种组织对象间共享行为的方式.

Lua的实现方式：

|  |
| --- |
| function Account:new(o)  o = o or {}  setmetatable(o, self)  self.\_\_index = self  return o  end |

## 继承

一行实现继承操作：

|  |
| --- |
| SpecialAccount = Account:new()  a = SpecialAccount:new() |

a的元表是SpecialAccount，SpecialAccount的元表时Account.

Lua中的对象可以无需为指定一种新行为而创建一个新类，只需要在该对象中直接定义这个行为.

## 多重继承

通过设置\_\_index为元方法来实现：

|  |
| --- |
| local function search(k,plist)  for I=1,#plist do  local v = plist[i][k]  if v then return v end  end  end  function createClass(…)  local c = {}  local parents = {…}  setmetatable(c, {\_\_index = function (t,k)  return search(k,parents)  end})  c.\_\_index = c  function c:new(o)  o = o or {}  setmatatable(o, c)  return o  end  return c  end  NamedAccount = createClass(Account, Named) |

改进多重继承性能的方法：

|  |
| --- |
| setmetatable(c, {\_\_index = function(t,k)  local v = search(k, parents)  t[k] = v  return v  end}} |

## 私密性

创建两个table，一个存储数据，一个存储接口，存储数据的table保存在closure里.

使用这种方法创建的对象不需要使用冒号：来访问.

创建私有方法，在closure中：

|  |
| --- |
| local func = function ()  end |

# 弱引用table

Lua采用了自动内存管理.

Lua的垃圾收集器与一些其他的收集器有所不同，它没有环形引用的问题.当用到环形数据结构时，无须做出任何特殊的处理，它们也可以像其他数据一样正常回收.

弱引用table就是用它来告诉Lua一个引用不应该阻碍一个对象的回收.

所谓弱引用就是有一种会被垃圾收集器忽视的对象引用，Lua就可以回收这个对象.

Lua用弱引用table来实现弱引用，一个弱引用table就是一个具有弱引用条目的table.

如果一个对象只被一个弱引用table所持有，那么最终Lua是会回收这个对象.

有3种弱引用table：具有弱引用key的table、弱引用value的table、同时具有两种的table.

不论是哪种类型，只要有一个key或value被回收了，那么它所在的整个条目都会从table中删除.

弱引用类型通过元表中的\_\_mode字段来决定的，是个字符串类型，包含k就是key弱引用，包含v就是value弱引用.

|  |
| --- |
| a = {}  b = {\_\_mode = “k”}  setmetatable(a, b)  key = {}  a[key] = 1  key = {}  a[key] = 2  collectgarbage()  for k,v in pairs(a) do print(v) end  ->>2 |

Lua只会回收弱引用table中的对象，而数字和布尔这样的值是不可回收的.

字符串是值，不会从弱引用table中删除.

但如果key是数字，对应的值是对象被删除，也会被删除.

## 备忘录函数

空间换时间，记录函数计算的结果，再次调用复用结果.

如可记录loadstring的结果.

## 对象属性

关于弱引用table，另一项重要应用是将属性与对象关联起来.

思路是单独一个table存储属性，将对象作为key，属性作为value.

这种情况必须使用弱引用，否则由于table持有了对象作为key，对象将无法释放.

# 调试库

调试库的功能由C API给出，Lua调试库提供了访问这些API的接口.

调试库由两类函数组成：自省函数和钩子.

自省函数允许检查一个正在运行中的程序的数据.

钩子则允许跟踪一个程序的执行.

栈层：调用调试库的函数是层1，上一层是2，以此类推.

## 自省机制

debug.getinfo函数，参数可以为函数名或栈层.

debug.traceback返回调用栈.

### 访问局部变量

debug.getlocal来检查任意活动函数的局部变量.

debug.setlocal 改变局部变量的值.

### 访问非局部变量

debug.getupvalue，获取closure持有变量的信息.

debug.setupvalue改变非局部变量.

## 钩子

调试库中的钩子机制使用户可以注册一个钩子函数，在程序运行中某个特定的事件发生时被调用.

4种事件会触发钩子：调用函数、函数返回、执行一行新代码、执行完指定数量的代码.

debug.sethook

## 性能剖析

基于钩子和自省机制实现的性能剖析.