Lua快速入门

Robert Ray
2011-5更新

前言

本文针对的读者是希望了解Lua或者迅速抓住Lua的关键概念和编程模式的有经验的程序员。因此本文并不打算教给读者诸如条件语句的语法或者函数定义的方式等等显而易见的东西，以及一些诸如变量、函数等现代编程语言普遍的基本概念。本文只打算告诉读者Lua那些与众不同的特性以及它们实际上带来了怎样颠覆以往的、独特的编程思维方式。

本文共分初、中、高阶三大部分，每部分又有若干章节。读者应当从头至尾循序渐进阅读，但标有“*”号的章节（主要讨论OO在Lua中的实现方式）可以略去而不影响对后面内容的理解。读者只要把前两部分完成就可以胜任Lua开发的绝大部分任务。高阶部分可作为选择。

本文不能取代Lua参考手册或者一本全面的Lua教科书，对一些重要的Lua函数也未做足够的说明。在阅读的同时或者之后，读者应当多多参考Lua的正式文档或者其他相关材料（附录里列出了一些常用的Lua参考资料）。

初阶话题

- 数据类型
- 函数
- 表
- 简单对象的实现*

数据类型

八种类型：

- 数值（number）
  内部以double表示的数值
- 字符串(string)
  由任意字符(包括零)组成的、以零结尾的字符序列
- 布尔（boolean）

只有“true”或者“false”两个值的逻辑类型

- 函数(function)

基本的Lua对象，不同于C语言的函数或函数指针，Lua的关键概念之一

- 表(table)
  “异构”的Hash表，Lua的另一关键概念
- userdata

C语言用户定义的数据结构，用于扩展Lua语言，本文不会涉及

- 线程（thread）

协作线程（coroutine），不同于一般操作系统的抢占式线程

- nil

代表“空”，类似于C语言的NULL，但实际含义要深刻得多

函数

```
function foo(a, b, c)
  local sum = a + b
  return sum, c  --函数可以返回多个值
end

r1, r2 = foo(1, '123', 'hello')  --平行赋值
print(r1, r2)
```

输出结果：

```
124 hello
```

函数(续)

- 函数定义

用关键字function定义函数，以关键字end结束

- 函数可以返回多个值

return a, b, c, ...

- 平行赋值

a, b = c, d

- 局部变量

  用关键字local定义。如果没有用local定义，即使在函数内部定义的变量也是全局变量！

- 全局变量

没有用local定义的变量都是全局变量。前面的代码定义了三个全局变量：foo、r1和r2

表

```
--定义一张空表
a = { }
--定义一张有初始内容的表
b = { n = 1,  str = 'abc', 100, 'hello' }

--访问表的成员
a.n = 1
a.str = 'abc'
a[1] = 100
a[2] = 'hello'
a["a table"] = b

--任何类型的值都可以做表项的key
--(除了nil)
function func1() end
function func2() end
a[func1] = func2
```

```
--穷举表a
for k, v in pairs(a) do
  print(k, "=>", v)
end
```

输出：

```
1       =>      100
2       =>      hello
str     =>      abc
function:
0027C988        =>      function:
0027C9A8
a table =>      table: 0027B138
n       =>      1
```

表

- 表

Lua的表既是Hash表，也是数组。实际上，可以把数组看作键为数值的Hash表。

- 访问表的成员

通过“.”或者“[]”运算符来访问表的成员。
注意：表达式a.b等价于a[“b”]，但不等价于a[b]

- 表项的键和值

任何类型的变量，除了nil，都可以做为表项的键或值。
给一个表项的值赋nil意味着从表中删除这一项：比如令a.b = nil，则把表a中键为“b”的项删除。
访问一个不存在的表项，会得到nil：比如有c = a.b，但表a中没有键为“b”的项，则c得到nil。

简单对象的实现*

```
function createFoo(name)
  local obj = { name = name}
  function obj:SetName(name)
    self.name = name
  end
  function obj:GetName()
    return self.name
  end
  return obj
end
```

```
o = createFoo("Sam")
print("name:", o:GetName())

o:SetName("Lucy")
print("name:", o:GetName())
```

输出结果：

```
name: Sam
name: Lucy
```

简单对象的实现*

- 对象工厂模式

如前面代码的create函数

- 对象的表示方法

用表来表示对象，把对象的数据和方法都放在一张表内，虽然没有隐藏私有成员，但在实践中完全可行。

- 成员方法的定义

function obj:method(a1, a2, ...) … end 等价于

function obj.method(self, a1, a2, ...) … end 等价于

obj.method = function (self, a1, a2, ...) … end

- 成员方法的调用

obj:method(a1, a2, …) 等价于

obj.method(obj, a1, a2, ...)

进阶话题

- 函数闭包（function closure）
- 基于对象的编程（object based programming）*
- 元表(metatable)
- 基于原型的继承（prototype based inheritance）*
- 函数环境（function envrionment）
- 模块（module）

函数闭包

```
function createCountdownTimer
(second)
  local ms = second * 1000
  local function countDown()
    ms = ms - 1
    return ms
  end
  return countDown
end

timer1 = createCountdownTimer(1)
for i = 1, 3 do
  print(timer1())
end
```

```
print("------------")
timer2 = createCountdownTimer(1)
for i = 1, 3 do
  print(timer2())
end
```

输出结果：

```
999
998
997
------------
999
998
997
```

函数闭包

- Upvalue

一个函数所使用的定义在它的函数体之外的局部变量（external local variable）称为这个函数的upvalue。

在前面的代码中，函数countDown使用的定义在函数createCountdownTimer中的局部变量ms就是countDown的upvalue，但ms对createCountdownTimer而言只是一个局部变量，不是upvalue。

- 函数闭包

一个函数和它的所有upvalue构成了一个函数闭包。函数闭包是Lua这一类“函数式”语言的核心概念，建议读者结合示例和相应文档仔细体会。

- Lua函数闭包与C函数的比较

Lua函数闭包使函数在几次调用间具有保持自身状态的能力，从此角度看，与带静态局部变量的C函数相似。但二者其实截然不同：前者是一个运行时对象，后者只是一个静态地址；前者可以有“同一类型”的若干实例，每个实例都有自己的状态（如前面的例子），而后者只是一个静态地址，谈不上实例化。

基于对象的编程*

```
function createFoo(name)
  local data = { name = name }
  local obj = {}
  function obj.SetName(name)
    data.name = name
  end
  function obj.GetName()
    return data.name
  end
  return obj
end
```

```
o = createFoo("Sam")
print("name:", o.GetName())
o.SetName("Lucy")
print("name:", o.GetName())
```

输出结果：

```
name:  Sam
name: Lucy
```

基于对象的编程*

- 实现要点

把需要隐藏的成员放在一张表里，把该表作为公有成员函数的upvalue；再把所有的共有成员放在另一张表里，把这张表作为对象。

- 局限性

考虑到对象继承的情况，这种方法的适用性有所限制。但另一方面，是否需要对象继承要视情况而定。

元表

```
t = {}
t2 = { a = " and ", b = "Li Lei", c = "Han Meimei" }
m = { __index = t2}
setmetatable(t, m)  --设表m为表t的元表
for k, v in pairs(t) do  --穷举表t
  print(k, v)
end
print("------------")
print(t.b, t.a, t.c)
```

输出结果：

```
------------
Li Lei and Han Meimei
```

元表

```
function add(t1, t2)
  --‘#’运算符取表长度
  local length = #t1
  for i = 1, length do
    t1[i] = t1[i] + t2[i]
  end
  return t1
end

t1 = { 1, 2, 3 }
t2 = { 10, 20, 30 }
setmetatable(t1, { __add = add })
setmetatable(t2, { __add = add })

t1 = t1 + t2
```

```
--穷举表t1
for i = 1, #t1 do
  print(t1[i])
end
```

输出结果：

```
11
22
33
```

元表

- 元表

元表本身只是一张普通的表，一般带有一些特殊的事件回调函数，通过 setmetatable被设置到某个对象上进而影响这个对象的行为。回调事件（如__index和__add）由Lua定义，而事件回调函数由脚本用户定义并在相应事件发生时被Lua解释器调用。以前面的代码为例，表的加法运算在缺省状态下将产生异常，但是设置了适当元表的表就可以进行加法运算了——Lua解释器将在表做加法运算时调用用户定义的__add回调函数。

- 重载运算符

  从前面的例子里读者可能已经意识到在Lua里运算符可以被重载。确实是这样，不仅是“+”运算，几乎所有的对象的运算符都可以被重载。

- 更多内容

元表Lua的内容十分丰富，建议读者参考Lua手册获得更多了解。

基于原型的继承*

```
Robot = { name = "Sam", id = 001 }
function Robot:New(extension)
  local t = setmetatable(extension or { }, self)
  self.__index = self
  return t
end
function Robot:SetName(name)
  self.name = name
end
function Robot:GetName()
  return self.name
end
function Robot:SetId(id)
  self.id = id
end
function Robot:GetId()
  return self.id
end
robot = Robot:New()
print("robot's name:", robot:GetName())
print("robot's id:", robot:GetId())
print("----------------")
FootballRobot = Robot:New(
  {position = "right back"})
function FootballRobot:SetPosition(p)
  self.position = p
end
function FootballRobot:GetPosition()
  return self.position
end
fr = FootballRobot:New()
print("fr's position:", fr:GetPosition())
print("fr's name:", fr:GetName())
print("fr's id:", fr:GetId())
print("----------------")
fr:SetName("Bob")
print("fr's name:", fr:GetName())
print("robot's name:", robot:GetName())
```

输出结果：

```
robot's name: Sam
robot's id: 1
----------------
fr's position: right back
fr's name: Sam
fr's id: 1
----------------
fr's name: Bob
robot's name: Sam
```

基于原型的继承*

- prototype模式

一个对象既是一个普通的对象，同时也可以作为创建其他对象的原型的对象（即类对象，class object）；动态的改变原型对象的属性就可以动态的影响所有基于此原型的对象；另外，基于一个原型被创建出来的对象可以重载任何属于这个原型对象的方法、属性而不影响原型对象；同时，基于原型被创建出来的对象还可以作为原型来创建其他对象。

函数环境

```
function foo()
  print(g or "'g' is not defined!")
end

foo()

env = { g = 100, print = print }
setfenv(foo, env)  --设置foo的环境为表env
foo()
print(g or "'g' is not defined!")
```

输出结果：

```
'g' is not defined!
100
'g' is not defined!
```

函数环境

- 函数环境

函数环境就是一个函数在运行时所能访问的“全局”变量的集合，装在一个表中。在缺省状态下，一个函数与定义它的函数共享同一个环境；但是每个函数都可以有自己独立的环境，通过setfenv来设定。

在前面的代码中，函数foo的缺省环境里没有定义变量g，因此第一次执行foo，g为nil。随后，foo被指定了一个环境 { g = 100, print = print }。这个环境定义了（全局）变量g，以及打印函数print，因此第二次执行foo，g的值就是100。但是在定义函数foo的函数的环境下，g仍然是一个未定义的变量。

函数环境是一个比较复杂的概念，建议读者结合示例与文档仔细体会。

- 应用

  利用它可以实现“安全沙箱”执行不受信任的代码；另外Lua的模块（module）的实现也依赖它。

模块

```
--hello.lua:
--定义名为hello的模块
--并使全局变量在此模块中可见
module(‘hello’, package.seeall)

ver = "version 0.1"

function hello()
    print("Hello!")
end
```

```
--test_hello.lua:
--使用模块
require "hello"

print(hello.ver)
hello.hello()
```

执行test_hello.lua的输出结果：

```
version 0.1
Hello!
```

模块

- 模块
  模块是一种代码的组织方式。
- 定义模块
  一般在一个Lua文件内以module调用开始定义一个模块。module调用同时为这个Lua文件定义了一个新的函数环境（初始为空表）。这里要注意：Lua解释器把一个Lua文件的内容当作一个匿名函数体来处理。设定了新的函数环境后，该文件内所定义的所有全局变量都保存在这个环境（表）里。package.seeall的意思是使全局变量在此模块中“可见”（如果没有package.seeall，在模块里就不可访问print函数，因为新的环境里没有定义它）。
- 使用方式
  一般用require函数来导入一个模块，要导入的模块必须被置于包路径（package path）上。包路径可以通过package.path或者环境变量来设定。一般来说，当前工作路径总是在包路径中。
- 更多
  请参考Lua手册进一步了解更多有关模块的说明。

高阶话题

- 迭代(iteration)
- 协作线程（coroutine）

迭代

```
function enum(array)
    local index = 1
    return function() --返回迭代函数
        local ret = array[index]
        index = index + 1
        return ret
    end
end

function foreach(array, action)
    for element in enum(array) do
        action(element)
    end
end

foreach({1, 2, 3}, print)
```

输出结果：

```
1
2
3
```

迭代

- 迭代
  迭代是for语句的一种特殊形式，for语句可以驱动迭代函数对一个给定集合进行遍历。正式、完备的语法说明较复杂，请参考Lua手册。
- 实现
  如前面代码所示：enum函数返回一个匿名的迭代函数，for语句每次调用该迭代函数都得到一个值（通过element变量引用），若该值为nil，则for循环结束。

协作线程

```
function producer()
    return coroutine.create(
        function (salt)
            local t = { 1, 2, 3 }
            for i = 1, #t do
                salt =
                    coroutine.yield(t[i] + salt)
            end
        end
    )
end
```

输出结果：

```
11
102
10003
END!
```

```
function consumer(prod)
    local salt = 10
    while true do
        local running, product =
            coroutine.resume(prod, salt)
        salt = salt * salt
        if running then
            print(product or "END!")
        else
            break
        end
    end
end

consumer(producer())
```

协作线程

- 协作线程
  Lua的线程对象。不同于一般操作系统所采取的抢占式线程，Lua采取了一种合作式线程。也就是说，只有在一个线程主动放弃处理器时，另一个线程才能执行。
- 创建协作线程
  通过coroutine.create可以创建一个协作线程，该函数接收一个函数类型的参数作为线程的执行体，返回一个线程对象。
- 启动或继续线程
  通过coroutine.resume可以启动一个线程或者继续一个挂起的线程。该函数接收一个线程对象以及其他需要传递给该线程的参数。线程可以通过线程函数的参数或者coroutine.yield调用的返回值来获取这些参数。当线程初次执行时，resume传递的参数通过线程函数的参数传递给线程，线程从线程函数开始执行；当线程由挂起转为执行时，resume传递的参数以yield调用返回值的形式传递给线程，线程从yield调用后继续执行。
- 线程放弃调度
  线程调用coroutine.yield暂停自己的执行，并把执行权返回给启动/继续它的线程；线程还可利用yield返回一些值给后者，这些值以resume调用的返回值的形式返回。

协作线程

```
function instream()
    return coroutine.wrap(function()
        while true do
            local line = io.read("*l")
            if line then
                coroutine.yield(line)
            else
                break
            end
        end
    end)
end

function filter(ins)
    return coroutine.wrap(function()
        while true do
            local line = ins()
            if line then
                line = "** " .. line .. " **"
                coroutine.yield(line)
            else
                break
            end
        end
    end)
end
```

```
function outstream(ins)
    while true do
        local line = ins()
        if line then
            print(line)
        else
            break
        end
    end
end

outstream(filter(instream()))
```

输入/输出结果：

```
abc
** abc **
123
** 123 **
^Z
```

协作线程

- Unix管道与Stream IO
  利用协作线程可以方便地设计出类似Unix管道或者Stream IO的结构。

协作线程

```
function enum(array)
    return coroutine.wrap(function()
        local len = #array
        for i = 1, len do
            coroutine.yield(array[i])
        end
    end)
end

function foreach(array, action)
    for element in enum(array) do
        action(element)
    end
end

foreach({1, 2, 3}, print)
```

输出结果：

```
1
2
3
```

协作线程

- 另一种迭代方式
  协作线程可以作为for循环迭代器的另一种实现方式。虽然对于简单的数组遍历来说，没有必要这么做，但是考虑一下，如果需要遍历的数据集合是一个复杂数据结构，比如一棵树，那么协作线程在简化实现上就大有用武之地了。

附录 常用的Lua参考资料

- Lua参考手册（最正式、权威的Lua文档）
- Lua编程（在线版，同样具权威性的Lua教科书）
- Lua正式网站的文档页面（包含很多有价值的文档资料链接）
- Lua维基（最全面的Lua维基百科）
- LuaForge（最丰富的Lua开源代码基地）