C/C++程序员的Lua快速入门指南

Robert Z

2010-1

前言

本文针对的读者是有经验的C/C++程序员，希望了解Lua或者迅速抓住Lua的关键概念和模式进行开发的。因此本文并不打算教给读者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、函数等编程语言的基本概念。本文只打算告诉读者Lua那些与C/C++显著不同的东西以及它们实际上带来了怎样不同于C/C++的思考方式。不要小看它们，它们即将颠覆你传统的C/C++的世界观！

本文一共分初阶、进阶和高阶三大部分，每个部分又有若干章节。读者应当从头至尾循序渐进的阅读，但是标有“*”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。

本文不打算取代Lua参考手册，因此对一些重要的Lua函数也未做足够的说明。在阅读的同时或者之后，读者应当在实践中多多参考Lua的正式文档（附录里列出了一些常用的Lua参考资料）。

请访问本文的在线版本获得最新更新。

另外，作者还有一个开源的Lua调试器——RLdb以及一个讨论Lua的站点，欢迎访问。

欢迎读者来信反馈意见。

初阶话题

- 数据类型
- 函数
- 表
- 简单对象的实现*
- 简单继承*

数据类型

八种基本类型：

- 数值（number）
  内部以double表示
- 字符串(string)
  总是以零结尾，但可以包含任意字符（包括零），因此并不等价于C字符串，而是其超集。
- 布尔（boolean）
  只有“true”或者“false”两个值。
- 函数(function)
  Lua的关键概念之一。不简单等同于C的函数或函数指针。
- 表(table)
  异构的Hash表。Lua的关键概念之一。
- userdata
  用户（非脚本用户）定义的C数据结构。脚本用户只能使用它，不能定义。
- 线程(thread)
  Lua协作线程（coroutine），与一般操作系统的抢占式线程不一样。
- nil
  代表什么也没有，可以与C的NULL作类比，但它不是空指针。

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
- 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！
- 函数可以返回多个值
  return a, b, c, ...
- 平行赋值
  a, b = c, d
- 全局变量
  前面的代码定义了三个全局变量：foo、r1和r2

表

```
a = { }
b = { x = 1, ["hello, "] = "world!" }
a.astring = "ni, hao!"
a[1] = 100
a["a table"] = b

function foo()
end
function bar()
end
a[foo] = bar

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

输出结果：

```
1         =>      100
a table =>      table: 003D7238
astring =>      ni, hao!
function:
003DBCE0      =>      function:
003DBD00
--------------------------
hello,  =>      world!
x         =>      1
```

表

- 定义表（Table）的方式
  a = {}, b = {...}
- 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。从简单的数值、字符串到复杂的函数、表等等都可以；同样，任何类型的变量，除了nil，都可以作为表项的值。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。如果访问一个不存在的表项，其值也是nil，比如有c = a.b，但表a中没有键为“b”的项，则c等于nil。

一种简单的对象实现方式*

```
function create(name, id)
  local obj = { name = name, id = id }
  function obj:SetName(name)
    self.name = name
  end
  function obj:GetName()
    return self.name
  end
  function obj:SetId(id)
    self.id = id
  end
  function obj:GetId()
    return self.id
  end
  return obj
end
```

```
o1 = create("Sam", 001)

print("o1's name:", o1:GetName(),
"o1's id:", o1:GetId())

o1:SetId(100)
o1:SetName("Lucy")

print("o1's name:", o1:GetName(),
"o1's id:", o1:GetId())
```

输出结果：

```
o1's name: Sam o1's id: 1
o1's name: Lucy o1's id: 100
```

一种简单的对象实现方式*（续）

- 对象工厂模式
  如前面代码的create函数
- 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，但对于简单脚本来说完全可以接受。
- 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, ...)

简单继承*

```
function createRobot(name, id)
  local obj = { name = name, id = id }

  function obj:SetName(name)
    self.name = name
  end

  function obj:GetName()
    return self.name
  end

  function obj:GetId()
    return self.id
  end

  return obj
end
```

```
function createFootballRobot(name,
id, position)

  local obj = createRobot(name, id)
  obj.position = "right back"

  function obj:SetPosition(p)
    self.position = p
  end

  function obj:GetPosition()
    return self.position
  end

  return obj
end
```

简单继承*（续）

- 优点：
  简单、直观

- 缺点：
  传统、不够动态

进阶话题

- 函数闭包（function closure）
- 基于对象的实现方式（object based programming）*
- 元表(metatable)
- 基于原型的继承（prototype based inheritance）*
- 函数环境（function envrionment）
- 包(package)

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（续）

- 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。
  Upvalue是Lua不同于C/C++的特有属性，需要结合代码仔细体会。
- 函数闭包
  一个函数和它所使用的所有upvalue构成了一个函数闭包。
- Lua函数闭包与C函数的比较
  Lua函数闭包使函数具有保持它自己的状态的能力，从这个意义上说，可以与带静态局部变量的C函数相类比。但二者有显著的不同：对Lua来说，函数是一种基本数据类型——代表一种（可执行）对象，可以有自己的状态；但是对带静态局部变量的C函数来说，它并不是C的一种数据类型，更不会产生什么对象实例，它只是一个静态地址的符号名称。

基于对象的实现方式*

```
function create(name, id)
  local data = { name = name, id = id
}
  local obj = {}
  function obj.SetName(name)
    data.name = name
  end
  function obj.GetName()
    return data.name
  end
  function obj.SetId(id)
    data.id = id
  end
  function obj.GetId()
    return data.id
  end
  return obj
end
```

```
o1 = create("Sam", 001)
o2 = create("Bob", 007)
o1.SetId(100)

print("o1's id:", o1.GetId(), "o2's id:",
o2.GetId())

o2.SetName("Lucy")

print("o1's name:", o1.GetName(),
"o2's name:", o2.GetName())
```

输出结果：

```
o1's id: 100 o2's id: 7
o1's name: Sam o2's name: Lucy
```

基于对象的实现方式*（续）

- 实现方式
  把需要隐藏的成员放在一张表里，把该表作为成员函数的upvalue。
- 局限性
  基于对象的实现不涉及继承及多态。但另一方面，脚本编程是否需要继承和多态要视情况而定。

元表

```
t = {}
m = { a = " and ", b = "Li Lei", c = "Han Meimei" }

setmetatable(t, { __index = m})  --表{ _index=m }作为表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(续)

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

--setmetatable返回被设置的表
t1 = setmetatable({ 1, 2, 3}, { __add
= add })
t2 = setmetatable({ 10, 20, 30 }, {
__add = add })
```

```
t1 = t1 + t2
for i = 1, #t1 do
  print(t1[i])
end
```

输出结果：

```
11
22
33
```

元表(续)

- 定义
  元表本身只是一个普通的表，通过特定的方法（比如setmetatable）设置到某个对象上，进而影响这个对象的行为；一个对象有哪些行为受到元表影响以及这些行为按照何种方式受到影响是受Lua语言约束的。比如在前面的代码里，两个表对象的加法运算，如果没有元表的干预，就是一种错误；但是Lua规定了元表可以“重载”对象的加法运算符，因此若把定义了加法运算的元表设置到那两个表上，它们就可以做加法了。元表是Lua最关键的概念之一，内容也很丰富，请参考Lua文档了解详情。
- 元表与C++虚表的比较
  如果把表比作对象，元表就是可以改变对象行为的“元”对象。在某种程度上，元表可以与C++的虚表做一类比。但二者还是迥然不同的：元表可以动态的改变，C++虚表是静态不变的；元表可以影响表（以及其他类型的对象）的很多方面的行为，虚表主要是为了定位对象的虚方法（最多再带上一点点RTTI）。

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*（续）

- 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 "No g defined!")
end

foo()

setfenv(foo, { g = 100, print = print })  --设置foo的环境为表{ g=100, ...}

foo()

print(g or "No g defined!")
```

输出结果：

```
No g defined!
100
No g defined!
```

函数环境（续）

- 定义
  函数环境就是函数在执行时所见的全局变量的集合，以一个表来承载。
- 说明
  每个函数都可以有自己的环境，可以通过setfenv来显示的指定一个函数的环境。如果不显示的指定，函数的环境缺省为定义该函数的函数的环境。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，表达式g or "No g defined!"的值就是"No g defined!"。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。
- 应用
  函数环境的作用很多，利用它可以实现函数执行的“安全沙箱”；另外Lua的包的实现也依赖它。

包

```
--testP.lua:
pack = require "mypack" --导入包

print(ver or "No ver defined!")
print(pack.ver)

print(aFunInMyPack or
    "No aFunInMyPack defined!")
pack.aFunInMyPack()

print(aFuncFromMyPack or
    "No aFuncFromMyPack defined!")
aFuncFromMyPack()
```

```
--mypack.lua:
module(..., package.seeall) --定义包

ver = "0.1 alpha"

function aFunInMyPack()
    print("Hello!")
end

_G.aFuncFromMyPack =
aFunInMyPack
```

包(续)

执行testP.lua的输出结果：

```
No ver defined!
0.1 alpha
No aFunInMyPack defined!
Hello!
function: 003CBFC0
Hello!
```

包(续)

- 定义
  包是一种组织代码的方式。
- 实现方式
  一般在一个Lua文件内以module函数开始定义一个包。module同时定义了一个新的包的函数环境，以使在此包中定义的全局变量都在这个环境中，而非使用包的函数的环境中。理解这一点非常关键。
  以前面的代码为例，“module(..., package.seeall)”的意思是定义一个包，包的名字与定义包的文件的名字相同（除去文件名后缀，在前面的代码中，就是“mypack”），并且在包的函数环境里可以访问使用包的函数环境（比如，包的实现使用了print，这个变量没有在包里定义，而是定义在使用包的外部环境中）。
- 使用方式
  一般用require函数来导入一个包，要导入的包必须被置于包路径（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。
- 其他
  请参考Lua手册进一步了解包的详细说明。

高阶话题

- 迭代(iteration)
- 协作线程（coroutine）

迭代

```
function enum(array)
    local index = 1
    return function()
        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(续)

- 定义
  迭代是for语句的一种特殊形式，可以通过for语句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（续）

- 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（续）

```
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（续）

- 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（续）

```
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（续）

- 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）