Lua脚本语言简明入门教程_Lua_脚本之家

这篇文章主要介绍了Lua脚本语言简明入门教程,本文简洁干练,可以让一个有编程基础的快速的学会Lua脚本语言,需要的朋友可以参考下
这几天系统地学习了一下Lua这个脚本语言，Lua脚本是一个很轻量级的脚本，也是号称性能最高的脚本，用在很多需要性能的地方，比如：游戏脚本，nginx，wireshark的脚本，当你把他的源码下下来编译后，你会发现解释器居然不到200k，这是多么地变态啊（/bin/sh都要1M，MacOS平台），而且能和C语言非常好的互动。我很好奇得浏览了一下Lua解释器的源码，这可能是我看过最干净的C的源码了。

我不想写一篇大而全的语言手册，一方面是因为已经有了（见本文后面的链接），重要的原因是，因为大篇幅的文章会挫败人的学习热情，我始终觉得好的文章读起来就像拉大便一样，能一口气很流畅地搞完，才会让人爽（这也是我为什么不想写书的原因）。所以，这必然又是一篇“入厕文章”，还是那句话，我希望本文能够让大家利用上下班，上厕所大便的时间学习一个技术。呵呵。

相信你现在已经在厕所里脱掉裤子露出屁股已经准备好大便了，那就让我们畅快地排泄吧……

运行
首先，我们需要知道，Lua是类C的，所以，他是大小写字符敏感的。

下面是Lua的Hello World。注意：Lua脚本的语句的分号是可选的，这个和GO语言很类似。

复制代码 代码如下:

```
print("Hello World")
```

你可以像python一样，在命令行上运行lua命令后进入lua的shell中执行语句。
复制代码 代码如下:

```
chenhao-air:lua chenhao$ lua
Lua 5.2.2  Copyright (C) 1994-2013 Lua.org, PUC-Rio
> print("Hello, World")
Hello, World
>
```

也可以把脚本存成一个文件，用如下命令行来运行。

复制代码 代码如下:

```
>lua  file.lua
```

或是像shell一样运行：
复制代码 代码如下:

```
chenhao-air:lua chenhao$ cat hello.lua
#!/usr/local/bin/lua
print("Hello, World")
chenhao-air:lua chenhao$ chmod +x hello.lua
chenhao-air:test chenhao$ ./hello.lua
Hello, World
```

语法注释
复制代码 代码如下:

```
-- 两个减号是行注释
--[[
 这是块注释
 这是块注释
 --]]
```

变量
Lua的数字只有double型，64bits，你不必担心Lua处理浮点数会慢（除非大于100,000,000,000,000），或是会有精度问题。

你可以以如下的方式表示数字，0x开头的16进制和C是很像的。

复制代码 代码如下:

```
num = 1024
num = 3.0
num = 3.1416
num = 314.16e-2
num = 0.31416E1
```

num = 0xff
num = 0x56

字符串你可以用单引号，也可以用双引号，还支持C类型的转义，比如：‘\a'（响铃），‘\b'（退格），‘\f'（表单），‘\n'（换行），‘\r'（回车），‘\t'（横向制表），‘\v'（纵向制表），‘\\'（反斜杠），‘\”‘（双引号），以及‘\”（单引号)
下面的四种方式定义了完全相同的字符串（其中的两个中括号可以用于定义有换行的字符串）

复制代码 代码如下:

a = 'alo\n123"'
a = "alo\n123\""
a = '\97lo\10\04923"'
a = [[alo
123"]]
C语言中的NULL在Lua中是nil，比如你访问一个没有声明过的变量，就是nil，比如下面的v的值就是nil

复制代码 代码如下:

v = UndefinedVariable
布尔类型只有nil和false是 false，数字0啊，‘’空字符串（'\0’）都是true！

另外，需要注意的是：lua中的变量如果没有特殊说明，全是全局变量，那怕是语句块或是函数里。变量前加local关键字的是局部变量。

复制代码 代码如下:

theGlobalVar = 50
local theLocalVar = "local variable"
控制语句
不多说了，直接看代码吧（注意：Lua没有++或是+=这样的操作）

while循环
复制代码 代码如下:

```
sum = 0
num = 1
while num <= 100 do
    sum = sum + num
    num = num + 1
end
print("sum =",sum)
```

if-else分支

复制代码 代码如下:

```
if age == 40 and sex =="Male" then
    print("男人四十一枝花")
elseif age > 60 and sex ~="Female" then
    print("old man without country!")
elseif age < 20 then
    io.write("too young, too naive!\n")
else
    local age = io.read()
    print("Your age is "..age)
end
```

上面的语句不但展示了if-else语句，也展示了

1）“ ~ =”是不等于，而不是!=

2）io库的分别从stdin和stdout读写的read和write函数

3）字符串的拼接操作符“..”

另外，条件表达式中的与或非为分是：and, or, not关键字。

for 循环

从1加到100

复制代码 代码如下:

```
sum = 0
for i = 1, 100 do
    sum = sum + i
end
```

从1到100的奇数和

复制代码 代码如下:

```
sum = 0
for i = 1, 100, 2 do
    sum = sum + i
end
```

从100到1的偶数和

复制代码 代码如下:

```
sum = 0
for i = 100, 1, -2 do
    sum = sum + i
end
```

until循环

复制代码 代码如下:

```
sum = 2
repeat
   sum = sum ^ 2 --幂操作
   print(sum)
until sum >1000
```

函数

Lua的函数和Javascript的很像

递归

复制代码 代码如下:

```
function fib(n)
  if n < 2 then return 1 end
  return fib(n - 2) + fib(n - 1)
end
```

闭包

同样，Javascript附体！

示例一

复制代码 代码如下:

```
function newCounter()
    local i = 0
    return function()     -- anonymous function
       i = i + 1
        return i
    end
end

c1 = newCounter()
print(c1())  --> 1
print(c1())  --> 2
```

示例二

复制代码 代码如下:

```
function myPower(x)
    return function(y) return y^x end
end

power2 = myPower(2)
power3 = myPower(3)

print(power2(4)) --4的2次方
print(power3(5)) --5的3次方
```

函数的返回值

和Go语言一样，可以一条语句上赋多个值，如：

复制代码 代码如下:

```
name, age, bGay = "haoel", 37, false, "haoel@hotmail.com"
```

上面的代码中，因为只有3个变量，所以第四个值被丢弃。

函数也可以返回多个值：

复制代码 代码如下:

```
function getUserInfo(id)
    print(id)
    return "haoel", 37, "haoel@hotmail.com", "http://jb51.net"
end

name, age, email, website, bGay = getUserInfo()
```

注意：上面的示例中，因为没有传id，所以函数中的id输出为nil，因为没有返回bGay，所以bGay也是nil。

局部函数

函数前面加上local就是局部函数，其实，Lua中的函数和Javascript中的一个德行。

比如：下面的两个函数是一样的：

复制代码 代码如下:

```
function foo(x) return x^2 end
foo = function(x) return x^2 end
```

Table

所谓Table其实就是一个Key Value的数据结构，它很像Javascript中的Object，或是PHP中的数组，在别的语言里叫Dict或Map，Table长成这个样子：

复制代码 代码如下:

```
haoel = {name="ChenHao", age=37, handsome=True}
```

下面是table的CRUD操作：

复制代码 代码如下:

```
haoel.website="http://jb51.net/"
local age = haoel.age
haoel.handsome = false
haoel.name=nil
```

上面看上去像C/C++中的结构体，但是name,age, handsome, website都是key。你还可以像下面这样写义Table：

复制代码 代码如下:

```
t = {[20]=100, ['name']="ChenHao", [3.14]="PI"}
```

这样就更像Key Value了。于是你可以这样访问：t[20]，t["name"], t[3.14]。

我们再来看看数组：

复制代码 代码如下:

```
arr = {10,20,30,40,50}
```

这样看上去就像数组了。但其实其等价于：
复制代码 代码如下:

```
arr = {[1]=10, [2]=20, [3]=30, [4]=40, [5]=50}
```

所以，你也可以定义成不同的类型的数组，比如：
复制代码 代码如下:

```
arr = {"string", 100, "haoel", function() print("coolshell.cn") end}
```

注：其中的函数可以这样调用：arr[4]()。

我们可以看到Lua的下标不是从0开始的，是从1开始的。

复制代码 代码如下:

```
for i=1, #arr do
    print(arr[i])
end
```

注：上面的程序中：#arr的意思就是arr的长度。

注：前面说过，Lua中的变量，如果没有local关键字，全都是全局变量，Lua也是用Table来管理全局变量的，Lua把这些全局变量放在了一个叫“_G”的Table里。

我们可以用如下的方式来访问一个全局变量（假设我们这个全局变量名叫globalVar）：

复制代码 代码如下:

_G.globalVar
_G["globalVar"]
我们可以通过下面的方式来遍历一个Table。

复制代码 代码如下:

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

MetaTable 和 MetaMethod

MetaTable和MetaMethod是Lua中的重要的语法，MetaTable主要是用来做一些类似于C++重载操作符式的功能。

比如，我们有两个分数：

复制代码 代码如下:

```
fraction_a = {numerator=2, denominator=3}
fraction_b = {numerator=4, denominator=7}
```

我们想实现分数间的相加：2/3 + 4/7，我们如果要执行： fraction_a + fraction_b，会报错的。

所以，我们可以动用MetaTable，如下所示：

复制代码 代码如下:

```
fraction_op={}
function fraction_op.__add(f1, f2)
    ret = {}
    ret.numerator = f1.numerator * f2.denominator + f2.numerator * f1.denominator
    ret.denominator = f1.denominator * f2.denominator
    return ret
end
```

为之前定义的两个table设置MetaTable：（其中的setmetatble是库函数）

复制代码 代码如下:

```
setmetatable(fraction_a, fraction_op)
```

setmetatable(fraction_b, fraction_op)
于是你就可以这样干了：（调用的是fraction_op.__add()函数）

复制代码 代码如下:

```
fraction_s = fraction_a + fraction_b
```

至于__add这是MetaMethod，这是Lua内建约定的，其它的还有如下的MetaMethod：
复制代码 代码如下:

```
__add(a, b)                   对应表达式 a + b
__sub(a, b)                   对应表达式 a - b
__mul(a, b)                   对应表达式 a * b
__div(a, b)                   对应表达式 a / b
__mod(a, b)                   对应表达式 a % b
__pow(a, b)                   对应表达式 a ^ b
__unm(a)                      对应表达式 -a
__concat(a, b)                对应表达式 a .. b
__len(a)                      对应表达式 #a
__eq(a, b)                    对应表达式 a == b
__lt(a, b)                    对应表达式 a < b
__le(a, b)                    对应表达式 a <= b
__index(a, b)                 对应表达式 a.b
__newindex(a, b, c)           对应表达式 a.b = c
__call(a, ...)                对应表达式 a(...)
```

“面向对象”
上面我们看到有__index这个重载，这个东西主要是重载了find key的操作。这操作可以让Lua变得有点面向对象的感觉，让其有点像Javascript的prototype。

所谓__index，说得明确一点，如果我们有两个对象a和b，我们想让b作为a的prototype只需要：

复制代码 代码如下:

```
setmetatable(a, {__index = b})
```

例如下面的示例：你可以用一个Window_Prototype的模板加上__index的MetaMethod来创

建另一个实例：
复制代码 代码如下:

```
Window_Prototype = {x=0, y=0, width=100, height=100}
MyWin = {title="Hello"}
setmetatable(MyWin, {__index = Window_Prototype})
```

于是：MyWin中就可以访问x, y, width, height的东东了。（注：当表要索引一个值时如table[key], Lua会首先在table本身中查找key的值, 如果没有并且这个table存在一个带有__index属性的Metatable, 则Lua会按照__index所定义的函数逻辑查找）

有了以上的基础，我们可以来说说所谓的Lua的面向对象。

复制代码 代码如下:

```
Person={}

function Person:new(p)
    local obj = p
    if (obj == nil) then
        obj = {name="ChenHao", age=37, handsome=true}
    end
    self.__index = self
    return setmetatable(obj, self)
end

function Person:toString()
    return self.name .." : ".. self.age .." : ".. (self.handsome and "handsome" or "ugly")
end
```

上面我们可以看到有一个new方法和一个toString的方法。其中：

1）self 就是 Person，Person:new(p)，相当于Person.new(self, p)
2）new方法的self.__index = self 的意图是怕self被扩展后改写，所以，让其保持原样
3）setmetatable这个函数返回的是第一个参数的值。

于是：我们可以这样调用：

复制代码 代码如下:

```
me = Person:new()
print(me:toString())

kf = Person:new{name="King's fucking", age=70, handsome=false}
print(kf:toString())
```

继承如下，我就不多说了，Lua和Javascript很相似，都是在Prototype的实例上改过来改过去的。

复制代码 代码如下:

```
Student = Person:new()

function Student:new()
    newObj = {year = 2013}
    self.__index = self
    return setmetatable(newObj, self)
end

function Student:toString()
    return "Student : ".. self.year.." : " .. self.name
end
```

模块

我们可以直接使用require(“model_name”)来载入别的lua文件，文件的后缀是.lua。载入的时候就直接执行那个文件了。比如：

我们有一个hello.lua的文件：

复制代码 代码如下:

```
print("Hello, World!")
```

如果我们：require(“hello”)，那么就直接输出Hello, World！了。

注意：

1）require函数，载入同样的lua文件时，只有第一次的时候会去执行，后面的相同的都不执行了。

2）如果你要让每一次文件都会执行的话，你可以使用dofile(“hello”)函数
3）如果你要玩载入后不执行，等你需要的时候执行时，你可以使用 loadfile()函数，如下所示：

复制代码 代码如下:

```
local hello = loadfile("hello")
... ...
... ...
hello()
```

loadfile(“hello”)后，文件并不执行，我们把文件赋给一个变量hello，当hello()时，才真的执行。（我们多希望JavaScript也有这样的功能

当然，更为标准的玩法如下所示。

假设我们有一个文件叫mymod.lua，内容如下：

文件名：mymod.lua

复制代码 代码如下:

```
local HaosModel = {}

local function getname()
    return "Hao Chen"
end

function HaosModel.Greeting()
    print("Hello, My name is "..getname())
end

return HaosModel
```

于是我们可以这样使用：

复制代码 代码如下:

```
local hao_model = require("mymod")
hao_model.Greeting()
```

其实，require干的事就如下：（所以你知道为什么我们的模块文件要写成那样了）
复制代码 代码如下:

```
local hao_model = (function ()
  --mymod.lua文件的内容--
end)()
```

参考

我估计你差不多到擦屁股的时间了，所以，如果你还比较喜欢Lua的话，下面是几个在线文章你可以继续学习之：

manual.luaer.cn lua在线手册
book.luaer.cn lua在线lua学习教程
lua参考手册Lua参考手册的中文翻译（云风翻译版本）

关于Lua的标库，你可以看看官方文档：string， table， math，io， os。

- *1*