

lua介绍

1993年巴西里约热内卢天主教大学诞生了一门编程语言，他们取名lua---在葡萄牙语中代表美丽的月亮，简洁优雅，lua从一开始就是作为一门方便嵌入（其他应用程序）并可扩展的轻量级脚本语言来设计，因此他一直遵循简单、小巧、可移植原则，能以C程序库的形式嵌入到宿主程序中。LuaJIT2和标准Lua 5.1 解释器采用著名的MIT许可协议。

Lua和LuaJIT的区别

lua非常高效，它运行得比其他脚本（Perl、Python、Ruby）都快，这点在第三方的独立测试中得以证实，尽管如此，仍然会有人不满足，他们总觉得“还不够快”。LuaJIT就是一个为了榨出一些速度的尝试，它利用即时编译（Just-in Time）技术把Lua代码编译成本机机器码反交给cpu直接执行。

Lua编写HelloWord脚本

为了方便以后查找，我们在/usr/local/下创建lua目录，用以存放lua脚本，创建一个helloworld.lua脚本，再使用notepad进行编辑

```
[root@MyHost local]# mkdir lua
[root@MyHost local]# cd lua/
[root@MyHost lua]# touch helloworld.lua
[root@MyHost lua]# ls
helloworld.lua
```

- 编写helloworld代码

```
print("hello word")

function sayHello()
    print("function Hello")
end

sayHello();
```

- 编译运行

```
[root@MyHost profile.d]# cd /usr/local/openresty/
[root@MyHost openresty]# ls
bin  luajit  lualib  nginx
[root@MyHost openresty]# cd luajit/
[root@MyHost luajit]# ls
bin  include  lib  share
[root@MyHost luajit]# cd bin/
[root@MyHost bin]# ls
luajit  luajit-2.1.0-beta1
[root@MyHost bin]# ./luajit /usr/local/lua/helloworld.lua    # 编译运行
hello word
function Hello
```

Lua基本语法

- 注释

```
单行注释: 两个减号  --注释内容
多行注释:
--[[
    多行注释
]]
```

- 基本类型 (lua中有8个基本类型分别为)

```
nil (空) -- python的None
boolean (布尔)
number (数字) -- python的int, 双精度的浮点数
string (字符串)
table (表) -- 类似于python的map
function (函数)
userdata (自定义类型)
thread (线程/协程)

-- 使用type函数测试给定变量的值类型
```

- 变量

命名规则: 大小写区分、其他的python命名规则类似

- Lua 标识符用于定义变量, 函数获取其他用户定义的项, 标识符以一个字母或'_'开头后。
- **注意:** 一般约定, 以下划线开头连接的一串大写字母的名字 (如: _VERSION) 被保留用于lua的全局变量
- 不能使用关键字

```
and 、 break、 do、 else、 elseif、 end、 false、 true、 for、 function、 if
in、 local、 nil、 not、 or、 return、 then、 repeat、 until、 while
```

全局变量: 在默认情况下, 变量总是认为是全局的, 除非, 你在前面加了"local", 这一点要特别注意, 因为可能想在函数中使用局部变量, 却忘了加local。

全局变量不需要声明，给一个变量复制后即创建了一个全局变量，访问一个没有初始化的全局变量也不会报错，只不过得到：nil。

```
b = 1  -- 可加分号，也可不加
print("变量B: ",b)
print("变量B的类型: ",type(b))  --number, 和Python一样自动类型赋值
print(a,c)  -- nil nil

--如果你想删除一个全局变量，只需要将变量赋值为nil。
b = nil
print(b)  -- nil, 当且仅当一个变量不等于nil时，这个变量即存在

function syaVar()
    local d = 2;
    e = 4;
    print("局部变量: ",d)
end

syaVar();
print(d,e);  -- nil 4
```

局部变量：在变量名前加local

- nil类型

```
print(type(a))  --nil
```

对于全局变量和table，nil还有一个删除的作用，给全局变量或者table表里的变量赋值一个nil值，等同于把他们删除

```
tab = {key1 = "val1",key2="val2"}
for k,v in pairs(tab) do
    print(k .. "--" .. v)  -- .. 表示相加的意思，python中字符串相加使用+，这里使用..
end
print('-----')

tab.key1 = nil  -- 访问table的值
for k,v in pairs(tab) do
    print(k,"---",v)
end

-- 判断nil类型 作比较时应该加上双引号
type(x)
print(type(x)==nil)  -- false
print(type(x)=="nil")  -- true
```

- 布尔类型，可选值 true/false

lua中nil和false为假，其他的所有值都为真，比如：0、空字符

```

if a then
    print(a)
else
    print("not a")
end

if b then
    print(b)
else
    print("not b")
end

```

- number类型，用于表示实数，和C/C++里面的都变了类型很类似。可以使用数学函数 math.floor(向下取整)和 math.ceil(向上取整)进行取整操作

```

-- number类型
local count = 10
local order = 3.99
local score = 98.01

print(math.floor(order)) -- 向下取整 3
print(math.ceil(score)) -- 向上取整 99

```

- string类型

lua中有三种方式表达字符串：

- 1、使用一对匹配的单引号，如：'hello'
- 2、使用一对匹配的双引号，如："hello"
- 3、字符串可以用一种长括号 ([[]])括起来的方式定义，整个词法分析的时候不会受到分行限制，类似于多行字符串，如写诗之类的；注意：lua中的字符串在创建时会插入lua虚拟机中，相对而言是独立的，所以不能修改，也不能通过下标访问。

```

-- string类型
local str1 = "hello world"
local str2 = 'hello lua'
local str3 = [[
    窗前明月光，
    疑似地上霜。
]]

local change = "wang \'Li\' " -- 支持转义字符
local change = "wang\n \'Li\' " -- 换行
print("a" .. "b") -- 字符串的拼接，使用..

print(str1)
print(str2)
print(str3)
print(change)

print(tonumber("10") == 10) -- 字符串转number

```

```

print(tostring(10) == "10") -- number转字符串

print("#This is string") -- 使用#来计算字符串长度 14

s3 = '1234455sd'
local s4 = string.sub(s,2,5) -- 字符串的截断，但是必须赋值给新变量，s并没有变化
print(s3,s4) -- 1234455sd      2344  截取的是开区间，不骨头不顾尾

```

注意：string类型只要不允许修改，只能重新赋值。

```

s = '12313'
s1 = s
s = 'dadfafd'

print(s) -- dadfafd
print(s1) -- 12313  s被重新赋值了，但是s1指向没有变，和python一样

```

function 函数

有名函数：

```

optional_function_scope function function_name(argument1, argument2, argument3...)
    function_body
    return result_params_comma_separated
end

```

--解析：

optional_function_scope: 该参数是可选的指定函数是全局函数还是局部函数，未设置该参数时默认为全局函数，如果你需要设置函数为局部函数，使用local。

function: 函数定义关键字

function_name: 指定函数名称

argument1, argument2...: 函数参数

function_body: 函数体

result_params_comma_separated: 函数返回值，lua语言函数可以返回多个值，用,隔开。

end: 函数结束关键字

--函数返回两个值的最大值

```

local function max(v1,v2)
    if (v1>v2) then
        return v1;
    else
        return v2;
    end
end

-- 调用函数
print("两个比较最大值",max(2,5));

```

匿名函数：本质是将函数赋值给变量

```

optional_function_scope function_name = function (argument1, argument2, argument3...)
    function_body
    return result_params_comma_separated
end

function foo() end 等价于 foo = function() end

```

全局函数和局部函数的区别：

- 使用function声明的函数为全局函数，在引用时可以不会因为声明顺序而找不到
- 使用local function的函数为局部函数，在引用时必须声明在声明函数的后面

```

local function test1()
    print("hello test1")
end

function test()
    test2()
    test1()
end

function test2()
    print("hello test2")
end

test() -- 会报错

```

函数参数：

- 将函数作为参数传递给函数

```

local myprint = function(param)
    print("这是打印函数 -  ##",param,"##")
end

local function add(num1,num2,functionPrint)
    result = num1 + num2
    functionPrint(result)
end

add(1,4,myprint)

```

- 传递参数，lua参数可变

```

local function foo(a,b,c,d)
    print(a,b,e,r)
end

foo(1,3) -- 打印1, 3, nil, 不会报错

```

- 若参数个数大于形参个数，从左向右，多余的实参被忽略
- 若参数个数小于形参个数，从左向右，没有被初始化的形参被初始化为nil
- lua还支持变长参数（不定长参数），用...表示。此时访问参数要用你... 如

```
-- 算平均值
function average(...)
    result = 0
    print("...类型",type(...),...) -- ...类型          number  1      2      4
5    6      3
    local arg = {...} -- arg 就是table类型,将每个number做成一个table
    for i, v in ipairs(arg) do
        result = result + v
    end
    print("总共传入" .. #arg .."个数")
    return result/#arg
end

print("平均值: ",average(1,2,4,5,6,3))
```

• 返回值

lua函数允许返回多个值，返回多个值时用逗号隔开

```
local function init()
    return 1,"lua"
end

local x, y = init();
print(x,y)

local h, z, g = init(),2 -- 注意当lua中赋值时函数后面还有表达式时不会解构，也就是说h会取init()返回的
                           第一个值
print(h,z,g) -- 1,2,nil

local h1, z1, g1 = 2,init() -- 当函数后没有表达式时是可以解构的
print(h1,z1,g1) -- 2, 1, lua

local h1, z1, g1 = 2,(init()) -- 使用括号取第一个返回值，不解构
print(h1,z1,g1) -- 2, 1, nil
```

函数返回值的规则：

- 若返回值个数大于接受变量个数，多余的返回值将会被忽略
- 若小于接受的个数，默认赋值为nil
- 如果你确保只取函数返回值的第一个值，可以使用括号运算符

table 表

table类型实现了一种抽象的“关联数组”。即可用作数组，也可用作map。（lua中没有数组和map，都是table这个类型实现）

- 初始化表

```
mytable = {}
```

- 指定值

```
mytable[1] = "lua"  
mytable["k1"] = "v1"
```

- 移除引用


```
mytable = nil -- lua垃圾回收会释放内存，其实是移除了内存的引用，lua会自动回收没有引用的数据
```

- 实例

```
-- 先初始化，后赋值  
mytable = {}  
mytable[1] = "lua"  
mytable["k1"] = "v1"  
print(mytable,mytable[2],mytable[1],mytable['k1']) -- table: 0x41f6cc78 nil      lua  
v1  
  
-----  
-- 初始化时赋值数组  
mytable1 = {1,2,4,5,6} -- 类似于数组  
print(mytable1,mytable1[1]) -- 注意: lua中的数组索引从1开始不是0; table: 0x416edd28 1  
mytable2 = {"a","b","c"}  
print(mytable2[2]) -- b  
-- 修改数组  
mytable2[3]='x'  
print(mytable2[3])  
  
-----  
-- 初始化时赋值字典  
mytable3 = {k1="wang",k2="Li"} -- 在定义map时，key不用引号  
print(mytable3["k2"]) -- 在访问key时需要引号  
-- 修改值  
mytable3["k2"] = "xing"  
print(mytable3["k2"])  
  
-----  
-- 数组和map结合状态下  
mytable4 = {"a",key1="v2","b",k2="v3"}  
--注意，我们访问b时，是数组的下标2  
print(mytable4[1],mytable4["key1"],mytable4[2]) -- a, v2, b
```

table是内存的引用


```
-- map内存引用
mytable6 = {k1="wang",k2="Li"} -- 在定义map时, key不用引号
mytable7 = mytable6
mytable6["k1"] = '666'
print(mytable7['k1']) --666
```

1583160431683

Lua的运算符

- 计算运算符实例

```
print(1+3)
print(5/10)
print(2^10)

local num = 1247
print(num%2)
print((num%2)==1)
```

- 关系运算符, 返回true或者false

```
1、 == 等于, 检测两端值是否相等
2、 ~= 不等于
3、 > 大于
4、 < 小于
5、 >= 大于等于
6、 <= 小于等于
```

注意: table、userdata和函数进行判断是判断引用地址是否相等

```
local a = {x=1,y=0}
local b = {x=1,y=0}

if a == b then
    print("a==b")
else
    print("a~=b")
end
```

- 逻辑运算符

```
and 逻辑与运算符 (A and B) A 真则B, A假则A
or 逻辑或运算符 (A or B) A 真则A, A假则B
not 逻辑非运算符 取反
```

```
local c = nil
local d = 0
local e = 100

print(c and d) -- nil
print(c and e) -- nil
```

```
print(d and e)    -- 100

print(c or d) -- 0
print(c or e) -- 100
```

lua控制结构

- 条件-控制结构：if-else; 是我们熟知的一种控制结构，lua跟其他语言一样，提供了if-else
 - 单个if分支型

```
x = 20
if x > 0 then
    print("ok")
end

if (x > 0) then
    print("也 ok")
end
```

- 两个条件分支

```
x = 10

if x > 0 then
    print("分支1")
else
    print("分支2")
end
```

- 多个条件分支

```
score = 90

if score == 100 then
    print("分支1")
elseif score >= 60 then
    print("分支2")
-- 此处可添加多个elseif
else
    print("分支3")
end
```

- 注意：elseif 是连在一起的，如果分开要注意以下状态

```
score = 0
if score == 100 then
    print("分支1")
elseif score >= 60 then
    print("分支2")
else
    if score > 0 then
        print("分支3")
    else
        print("分支4")
    end
end
end
```

循环语句

lua和其他常见语言一样，提供了while控制语句，语法上也美哟特答的区别，但是没有提供 do-while型控制结构，但是提供了功能先当的repeat。

while 型控制结构语句语法如下，当表达式值为假（即false/nil）时循环结束，也可以使用break语句跳出循环

- while语法

```
while 表达式 do
    --body
end

--实例：求1+2+3...+5的和
x = 1
sum = 0

while x<=5 do
    sum = sum + x
    x = x +1
end
print(sum)
```

值得一提的是：lua并没有提供continue这样的控制语句用来立即进入下一个循环迭代，因此我们需要仔细安排循环里的分支，以免这样的需求（没有continue，但有break）

```
--实例： 遍历table, 查找值为11的下标

local t = {1,4,5,6,98,11,21,15}
local i = 1

while i < #t do
    if 11 == t[i] then
        print("index["..i.."] have right value[11]")
        break
    end
    i = i + 1
end
```

- repeat 控制结构

lua中的repeat控制结构类似于其他语言中的 do-while，但是控制方式是刚好相反的，简单说执行repeat循环体后，直到until的条件为真时才结束，而其他语言的do-while则是当条件为假时才结束

```
-- 以下代码会死循环
x = 10

repeat
    print(x)
until false -- 条件为真则结束循环
--该代码会导致死循环，因为until的条件一直为假，循环不为真不结束（repeat也可使用break跳出）。

repeat
    print(x)
    x = x - 1
until x < 0
```

注意点：

- repeat until 控制结构，它至少会执行一遍； while控制的更精确些

- for循环控制

for语句有两种形式： 数字 for 和 范型 for

- 数字型 for 的语法如下：

```
for var = begin, finish, step do
    -- body
end

-- 关于数字 for 需要注意几点：
--1、 var 从begin变化到finish，每次变化都已step作为步长递增 var
--2、 begin、 finish、 step 三个表达式只会在循环开始时执行一次
--3、 step 可选，默认为1
--4、 控制var的作用域仅在for循环内部，需要在外边控制，则需将赋值给一个新变量
--5、 循环过程中不要改变循环变量的值

for i=1,5 do
    print(i)
```

```

end

for i=1,10,2 do
    print(i)
end

for i=10,1,-2 do
    print(i)
end

-- 无穷大
for i=1, math.huge do
    if(0.3*i^3 - 20*i^2 -500 >= 0) then
        print(i)
        break
    end
end
end

```

- for 范型

对lua的table类型的遍历；范型for循环通过一个迭代器（iterator）函数来遍历所有值：

```

-- 打印数组所有值
local a = {"a","b","c"}

for i, v in ipairs(a) do
    print(i, v)
end

--lua的基本库提供了ipairs，这是一个用于遍历数组迭代器的函数，每次循环中，i会被赋值一个索引值，同时v会获得对于的元素值

local a = {"a","b","c",k="k"}

for i, v in ipairs(a) do
    print(i, v)
end

[[ -- 道理很简单ipairs函数只遍历数组，字典的值是不会被遍历的
1      a
2      b
3      c
]]

```

下面是另一个类型的实例，演示了如何遍历一个table中的**所有key**

```
-- 打印table中的所有key
local t = {"a","b","c",k="k"}

for k,v in pairs(t) do -- ipairs() table中的数组, pairs() table中的所有
    print(k)
end

[[
1      a
2      b
3      c
k      k
]]
```

- 案例：假设有一个table，它的内容是一周的每天的名称，现在要将一个名称转换成它在一周的位置

```
local days = {
    "sunday","monday","tuesday","wednesday","thursday","friday","saturday"
}
revDays = {}

-- 实现数组逆转
for i, v in ipairs(days) do
    revDays[v] = i
end

for k, v in pairs(revDays) do
    print(k.."="..v)
end
```

break、return关键字

- break

语句break用于终止while、repeat和for循环，跳出循环

```
-- 计算最小x，使得从1到x的所有数相加大于100
sum = 0
i = 1
while true do
    sum = sum + i
    if sum > 100 then
        break
    end
    i = i + 1
end
print("the result is"..i)
```

- return 主要用于从函数中返回结果，或者用于简单的结束一个函数的执行

```
local function add(x, y)
    return x+y
end
```

lua正则表达式

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（regex）来进行模式匹配，主要原有出于程序大小方面考虑：实现一个典型的符合POSIX标准的regex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有500行代码，当然意味着不可能实现POSIX所规范的所有功能，然而，lua中匹配功能很强大，并且包含了一些使用标准POSIX模式不容易匹配的功能。

- lua正则中的特殊字符（元字符）包括如下几种：() . % + - * ? [] ^ \$

元字符	描述	表达式实例	匹配的字符串
普通字符	除去特殊的字符 (() . % ..)	kana	kana
.	匹配任意字符	ka.a	kana
%	转义字符，改变后一个字符的原意思 当后面接的是特殊字符时，将还原有特殊字符的原意。%和一些特殊字符组合构成lua预定义字符集。%和数字1~9组合表示之前捕获的分组	K%wna %%na%% (a)na%1(第三位是前面1的分组)	Kana %na% ana
[...]	字符集（字符类）。匹配一个包含于集合内的字符。[...] 中的特殊字符将还原其原意，但有下面几种特殊情况：1、%, %, %, %..... 作为整体表示字符',','\','^'; 2、预定义字符集作为以以一个整体表示对应的字符集	[a%%]na [a%]na [%%a]na	%na %na wna
[...-...]	表示ascii码在它一个字符到它后一个字符之间的所有字符	[a-z]a	na
[^...]	b不在...中的字符集合	[^0-9]na	

```
test = "n2,nn45,n678,n34n"
print(string.gsub(test,"n?[0-9]+",""))  -- ,n,,n      4
```

- 重复字符，量词

重复 (数量词)			
*	表示前一个字符出现0次或多次	[0-9]*	2020
+	表示一个字符出现至少一次	n+[0-9]+	n2009
-	匹配前一个字符0次或多次		
?	表示前一个字符出现0次或1次	n?[0-9]+	2009

注意:

- 元字符+和*是贪婪的, 总是进行最长的匹配, 而-则是吝啬的, 总是最短匹配

```
例子:
local test = "<font>a</font><font>b</font>"
print(string.gsub(test,"<font>.+</font>", "6")) -- 6 1 这是会将所有的字符替换为6, 从开始的<font>到最后的</font>

local test = "<font>a</font><font>b</font>"
print(string.gsub(test,"<font>.-</font>", "6")) -- 66 2 匹配到两次, 两次替换
```

• 预定义字符集

预定义字符集			
%s	空白字符集 (比如空格,多个空格)	an[%s]?9	an 9
%p	标点符号	an[%p]9	an.9
%c	控制字符, 例如\n		
%w	字母数字[a-Z0-9]	[%w]+	kana9
%a	字母		
%u	大写字母		
%l	小写字母		
%d	数字		
%x	16进制数		

• 分组

(...) 表达式中用小括号包围的字符串为一个分组, 分组从左到右 (以左括号的位置), 组序号从1开始递增


```
local test1 = "123,234 again1 123,234,test"
print(string.gsub(test1,"([%d]+),([%d]+) again1 %1,%2","ok")) -- ok,test 1
```

- 边界符号

- ^: 匹配字符开头, 如: `^(%a)%w* abc123`
- \$: 匹配字符串结尾, 如: `%w*(%d)$ abc123`