[Lua 环境安装 3](#_Toc19141)

[Linux 系统上安装 3](#_Toc31044)

[Window 系统上安装 3](#_Toc4818)

[Lua 基本语法 3](#_Toc20474)

[Lua 数据类型 3](#_Toc20535)

[nil 3](#_Toc26834)

[boolean 3](#_Toc20929)

[number 4](#_Toc17058)

[string 4](#_Toc24867)

[table 4](#_Toc5545)

[function 4](#_Toc23978)

[thread 5](#_Toc21469)

[userdata 5](#_Toc18894)

[Lua 变量 5](#_Toc21708)

[赋值语句 5](#_Toc18962)

[索引 5](#_Toc20845)

[Lua循环 6](#_Toc18444)

[while循环 6](#_Toc14223)

[for循环 6](#_Toc17041)

[repeat循环 6](#_Toc22999)

[循环控制语句 6](#_Toc23638)

[Lua 流程控制 6](#_Toc26711)

[Lua函数 7](#_Toc1158)

[函数定义 7](#_Toc16209)

[可变参数 7](#_Toc12573)

[Lua 运算符 7](#_Toc17570)

[Lua 字符串 7](#_Toc8457)

[字符串操作 7](#_Toc4577)

[Lua 数组 8](#_Toc3166)

[Lua 迭代器 8](#_Toc22599)

[Lua table(表) 8](#_Toc6945)

[table(表)的构造 8](#_Toc5316)

[Table 操作 8](#_Toc32631)

[Lua 模块与包 8](#_Toc7678)

[require 函数 9](#_Toc13879)

[require加载机制 9](#_Toc8973)

[C 包 9](#_Toc18745)

[Lua 元表(Metatable) 9](#_Toc10530)

[\_\_index 元方法 9](#_Toc2251)

[\_\_newindex 元方法 10](#_Toc18318)

[为表添加操作符 10](#_Toc7742)

[\_\_call 元方法 10](#_Toc7350)

[\_\_tostring 元方法 11](#_Toc10086)

[Lua 协同程序(coroutine) 11](#_Toc3067)

[线程和协同程序区别 11](#_Toc19324)

[基本语法 11](#_Toc5031)

[实例 11](#_Toc31696)

[协程状态 12](#_Toc17229)

[Lua 文件 I/O 12](#_Toc26395)

[简单模式（simple model） 13](#_Toc24144)

[完全模式（complete model） 13](#_Toc7298)

[Lua 错误处理 13](#_Toc21349)

[Lua 调试(Debug) 13](#_Toc900)

[Lua 垃圾回收 14](#_Toc14659)

[Lua 面向对象 14](#_Toc14430)

[Lua 数据库访问 15](#_Toc16882)

## Lua 环境安装

### Linux 系统上安装

curl -R -O http://www.lua.org/ftp/lua-5.3.0.tar.gz

tar zxf lua-5.3.0.tar.gz

cd lua-5.3.0

make linux test

make install

### Window 系统上安装

菜鸟教程下载地址：[LuaForWindows\_v5.1.4-46.exe](http://static.runoob.com/download/LuaForWindows_v5.1.4-46.exe)

Github 下载地址：<https://github.com/rjpcomputing/luaforwindows/releases>

## Lua 基本语法

#!E:\javasoftware\Lua5.1，在文件头指定解释器，解释器忽略#标记内容

--，单行注释

--[[...--]]，--[=[...]=]，多行注释

变量标示符，规则与java一致。

变量默认为全局，给变量赋值后即创建全局变量；访问未初始化的全局变量结果为nil

删除全局变量，将其赋值为nil。

## Lua 数据类型

Lua是动态类型语言，变量不要类型定义,只需要为变量赋值。

变量声明时必须赋值。

8个基本类型：nil、boolean、number、string、userdata、function、thread、table。

### nil

空值，赋值给变量时表示删除变量。

可直接比较，x == nil

做类型比较时需使用双引号””，type(x )== “nil”

### boolean

true、false

nil为false，其他都为true。

### number

双精度数字，double。

### string

双引号””、单引号””、两个方括号[[]](一块字符串)括起来的字符序列。

html = [[

<html>

<head></head>

<body>

<a href="http://www.runoob.com/">菜鸟教程</a>

</body>

</html>

]]

**运算符**

.. 连接字符串，数字后写..时加空格防止解释错；’a’..’b’ ab

# 计算字符串长度，#’abc’ 3

### table

表；实质为”关联数组”，数组的索引为数字(.)、字符串(.，[])。

table数字索引从1开始。

table长度不固定，加入数据时自动增长。

未初始化的table为nil。

table = {};

table = {‘hello’,’tian’};

### function

函数，在 Lua 中，函数是被看作是"第一类值（First-Class Value）"，可存在变量里。

格式：function fName(args) fBody end；args可为函数、匿名函数

function factorial1(n)

if n == 0 then

return 1

else

return n \* factorial1(n - 1)

end

end

print(factorial1(5)) 120

factorial2 = factorial1

print(factorial2(5)) 120

匿名函数：function f() print(‘hi’); end

### thread

在 Lua 里，最主要的线程是协同程序（coroutine）。它跟线程（thread）差不多，拥有自己独立的栈、局部变量和指令指针，可以跟其他协同程序共享全局变量和其他大部分东西。

线程跟协程的区别：线程可以同时多个运行，而协程任意时刻只能运行一个，并且处于运行状态的协程只有被挂起（suspend）时才会暂停。

### userdata

用户自定义数据，用于表示一种由应用程序或 C/C++ 语言库所创建的类型，可以将任意 C/C++ 的任意数据类型的数据（通常是 struct 和 指针）存储到 Lua 变量中调用。

## Lua 变量

变量在使用前，必须在代码中进行声明，即创建该变量。

Lua 变量有三种类型：全局变量、局部变量、表中的域。

Lua 中的变量全是全局变量，那怕是语句块或是函数里，除非用 local 显式声明为局部变量。

局部变量的作用域为从声明位置开始到所在语句块结束。

变量的默认值均为 nil。

尽可能的使用局部变量，避免命名冲突，访问局部变量的速度比全局变量更快。

### 赋值语句

a = "hello" .. "world"

多个变量同时赋值，变量列表和值列表的各个元素用逗号分开，赋值语句右边的值会依次赋给左边的变量。

a, b = 10, 2\*x <--> a=10; b=2\*x

遇到赋值语句Lua会先计算右边所有的值然后再执行赋值操作，交换变量的值：

x, y = y, x -- swap 'x' for 'y'

当变量个数和值的个数不一致时，变量多则赋值为nil，变量少则忽略多余值。

### 索引

对 table 的索引使用方括号 [](字符串、数字)。Lua 也提供了 . (字符串)操作。

t[i]

t.i -- 当索引为字符串类型时的简化写法

## Lua循环

### while循环

while(exp) do ... end

exp为false跳出循环。

### for循环

**数值for循环**

格式：

for var=exp1,exp2,exp3 do ... end

exp3可选，默认递增1。

for的三个表达式在循环开始前一次性求值，以后不再进行求值。

**泛型for循环**

泛型for循环通过一个迭代器函数来遍历所有值，类似java中的foreach语句。

格式:

for i,v in ipairs(table) do ... end ipairs()为迭代器函数，用来迭代数组。

days = {"Sunday","Monday","Tuesday","Wednesday","Thursday","Friday","Saturday"}

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

### repeat循环

repeat ... util(exp)

exp为true跳出循环。

### 循环控制语句

break

## Lua 流程控制

if exp then ... end

if exp then ... else ... end

if exp then ... esle if ... end

## Lua函数

### 函数定义

格式：

optional\_function\_scope function fName( args)

function\_body

return result\_params\_comma\_separated

end

optional\_function\_scope，函数类型，可选；默认为全局函数，local设为局部函数

result\_params\_comma\_separated，返回值；可有多个返回值，逗号隔开

### 可变参数

在函数参数列表中使用三点 ... 表示函数有可变的参数。

函数体中可通过 {...} 访问可变参数，也可将其赋值给变量。

select(‘#’, ...)，可变参数长度

select(‘n’, ...)，获取第n个可变参数

## Lua 运算符

算术运算符：+，-(也可为负号)，\*，/，%，^(乘幂)

关系运算符：==，~=(不等)，>，<，>=，<=

逻辑运算符：and，or，not

其他运算符：..(连接字符串)，#(获取字符串、表的长度)

## Lua 字符串

双引号””、单引号””、两个方括号[[]](一块字符串)括起来的字符序列。

特殊字符用\转义。

### 字符串操作

string.upper(arg);转大写

string.lower(arg);转小写

string.gsub(str,find,replace,times);替换指定内容

string.find(str,substr,[begin,[end]]);返回指定字符串在目标字符串中开始、结束位置

string.reverse(arg);反转字符串

string.format(...);格式化

string.char(args);将整数转为字符，并拼接为字符串

string.byte(str,[index]);字符串中指定位置字符转为整数，默认位置1

string.len(str);字符串长度

string.rep(str,times);返回字符串的times个拷贝

string.gmatch(str,pattern);返回一个迭代函数，调用迭代函数返回str中满足pattern的字符串

string.match(str,pattern,init);

## Lua 数组

通过for循环遍历数组，赋值、取值。

## Lua 迭代器

pairs: 迭代 table，可以遍历表中所有的 key 可以返回 nil

ipairs: 迭代数组，不能返回 nil,如果遇到 nil 则退出

## Lua table(表)

table 是 Lua 的一种数据结构用来帮助我们创建不同的数据类型，如：数组、字典等。

Lua table 使用关联型数组，你可以用任意类型的值来作数组的索引，但这个值不能是 nil。

Lua table 是不固定大小的，你可以根据自己需要进行扩容。

### table(表)的构造

最简单的构造函数是{}，用来创建一个空表。可以直接初始化数组。

### Table 操作

table.concat(table,[sep,[start,[end]]]);列出元素，以sep隔开

table.insert(table,[index,],value);插入

table.remove(table[,index]);删除

table.sort(table[,comp]);升序排序

## Lua 模块与包

Lua 的模块是由变量、函数等已知元素组成的 table，因此创建一个模块很简单，就是创建一个 table，然后把需要导出的常量、函数放入其中，最后返回这个 table 就行。

module = {}

module.constant = "这是一个常量"

local function func2()

print("这是一个私有函数！")

end

function module.func3()

func2() --通过公有函数调用私有函数

end

return module

### require 函数

加载模块，格式：

require(“<模块名>”);

require”<模块名>”;

### require加载机制

require 用于搜索 Lua 文件的路径是存放在全局变量 package.path 中，当 Lua 启动后，会以环境变量 LUA\_PATH 的值来初始这个环境变量。

如果找到目标文件，则会调用 package.loadfile 来加载模块。否则，就会去找 C 程序库。

### C 包

loadlib(path,”functionName”);加载动态库

## Lua 元表(Metatable)

Lua 提供了元表(Metatable)，允许我们改变table的行为，每个行为关联了对应的元方法。

setmetatable(table,metatable);设置元表，并返回设置元表后的表；如果已设置则会失败

getmetatable(table);返回元表

类似es6 prototype。

### \_\_index 元方法

mytable = setmetatable({},{\_index={‘hi’});

Lua查找一个表元素3个步骤:

1.在表中查找，如果找到，返回该元素，找不到则继续

2.判断该表是否有元表，如果没有元表，返回nil，有元表则继续。

3.判断元表有没有\_\_index方法，如果\_\_index方法为nil，则返回nil；如果\_\_index方法是一个表，则重复1、2、3；如果\_\_index方法是一个函数，则返回该函数的返回值。

### \_\_newindex 元方法

\_\_newindex 元方法用来对表更新，\_\_index则用来对表访问 。

当你给表的一个缺少的索引赋值，解释器就会查找\_\_newindex 元方法：如果存在则调用这个函数而不进行赋值操作。对已有索引元素进行更新时直接更新。

mytable = setmetatable({key1 = "value1"}, {

\_\_newindex = function(mytable, key, value)

rawset(mytable, key, "\""..value.."\"")

end

})

mytable.key1 = "new value" new value

mytable.key2 = 4 “4”

### 为表添加操作符

\_\_add 对应的运算符 '+'.

\_\_sub 对应的运算符 '-'.

\_\_mul 对应的运算符 '\*'.

\_\_div 对应的运算符 '/'.

\_\_mod 对应的运算符 '%'.

\_\_unm 对应的运算符 '-'.

\_\_concat 对应的运算符 '..'.

\_\_eq 对应的运算符 '=='.

\_\_lt 对应的运算符 '<'.

\_\_le 对应的运算符 '<='.

对应方法赋值函数后，进行运算操作则调用对应函数处理。

### \_\_call 元方法

\_\_call 元方法在 Lua 调用一个值时调用。当调用值与函数参数不匹配时报错。

mytable = setmetatable({10}, {

\_\_call = function(mytable, newtable)

sum = 0

for i = 1, table\_maxn(mytable) do

sum = sum + mytable[i]

end

for i = 1, table\_maxn(newtable) do

sum = sum + newtable[i]

end

return sum

end

})

newtable = {10,20,30}

print(mytable(newtable)) 70

### \_\_tostring 元方法

\_\_tostring 元方法用于修改表的输出行为。

## Lua 协同程序(coroutine)

Lua 协同程序(coroutine)与线程比较类似：拥有独立的堆栈，独立的局部变量，独立的指令指针，同时又与其它协同程序共享全局变量和其它大部分东西。

### 线程和协同程序区别

一个具有多个线程的程序可以同时运行几个线程，而协同程序却需要彼此协作的运行。

在任一指定时刻只有一个协同程序在运行，并且这个正在运行的协同程序只有在明确的被要求挂起的时候才会被挂起。

协同程序有点类似同步的多线程，在等待同一个线程锁的几个线程有点类似协同。

### 基本语法

coroutine.create();创建coroutine，参数为函数，返回coroutine，配合resume()唤醒函数

coroutine.wrap();创建coroutine；参数为函数，返回一个函数，调用函数时进入cortoutine

coroutine.resume();重启coroutine，配合create()；返回boolean，yield参数/失败信息coroutine.yield();挂起

coroutine.status();查看状态，dead、suspend、running

coroutine.running();返回正在运行的coroutine

### 实例

co = coroutine.create(

function(i)

print(i);

end

)

coroutine.resume(co, 1) -- 1

print(coroutine.status(co)) -- dead

print("----------")

co = coroutine.wrap(

function(i)

print(i);

end

)

co(1)

print("----------")

co2 = coroutine.create(

function()

for i=1,10 do

print(i)

if i == 3 then

print(coroutine.status(co2)) --running

print(coroutine.running()) --thread:XXXXXX

end

coroutine.yield()

end

end

)

coroutine.resume(co2) --1

coroutine.resume(co2) --2

coroutine.resume(co2) --3

print(coroutine.status(co2)) -- suspended

print(coroutine.running())

### 协程状态

coroutine.create();挂起suspend

coroutine.resume();运行running

coroutine.yield();挂起suspend

协程执行完毕后，dead。

coroutine.resume();只能唤醒suspend状态的协程。

## Lua 文件 I/O

file = io.open (filename [, mode])

### 简单模式（simple model）

使用标准的 I/O 或使用一个当前输入文件和一个当前输出文件。

io.open(path);打开文件

io.input(file);设为默认输入文件

io.read();读取一行

io.close(file);关闭文件

io.write(str);写文件

io.output(file);设为默认输出文件

io.temfile();返回临时文件句柄

io.type(file);检测file是否为可用文件句柄

io.flush();写入所有缓冲数据

io.lines();返回一个迭代函数，每次返回一行内容

### 完全模式（complete model）

使用外部的文件句柄来实现。它以一种面对对象的形式，将所有的文件操作定义为文件句柄的方法。

使用 file:function\_name 来代替 io.function\_name 方法。

file = io.open(‘test.lua’,’r’);

file:read();读

file:close();关闭

file:write();写

file:seek();设置和获取当前文件位置

file:flush();写入缓冲数据

io.lines(fileName);返回指定文件迭代器

## Lua 错误处理

assert(exp,info); exp为false，抛出错误信息info

error(info[,level]); 终止函数，抛出错误信息info

if pcall(fName/function,args) then noERR else ERR end; 以args参数执行fName函数，

xpcall(fName/function,handler,args); 出错时调用handler，可通过debug.debug，debug.traceback获取错误信息

## Lua 调试(Debug)

debug 库包含以下函数：

debug.debug();进入一个用户交互模式，运行用户输入的每个字符串。输入’cont’终止

debug.traceback ([thread,] [message [, level]]);错误跟踪信息

debug.getfenv(object);返回对象环境变量

debug.gethook(optional thread);返回线程：当前钩子函数，当前钩子掩码，当前钩子数

debug.getinfo([thread,] f [,what]);返回关于一个函数信息的表。

debug.getlocal([thread,] f ,local);返回在栈的 f 层处函数的索引为 local 的局部变量 debug.getmetatable(value);把给定索引指向的值的元表压入堆栈。

debug.getregistry();返回注册表表

debug.getupvalue(f,up);返回函数 f 的第 up 个上值的名字和值。

debug.sethook([thread,] hook,mask [,count]);将一个函数作为钩子函数设入。

debug.setlocal ([thread,] level, local, value);将 value 赋给 栈上第 level 层函数的第 local 个局部变量。

debug.setmetatable (value, table);将 value 的元表设为 table

debug.setupvalue (f, up, value);将 value 设为函数 f 的第 up 个上值。

## Lua 垃圾回收

基本函数：

collectgarbage ([opt [, arg]]);

collectgarbage("collect"): 做一次完整的垃圾收集循环。通过参数 opt 它提供了一组不同的功能：

collectgarbage("count"): 以 K 字节数为单位返回 Lua 使用的总内存数。 这个值有小数部分，所以只需要乘上 1024 就能得到 Lua 使用的准确字节数（除非溢出）。

collectgarbage("restart"): 重启垃圾收集器的自动运行。

collectgarbage("setpause"): 将 arg 设为收集器的 间歇率 （参见 §2.5）。 返回 间歇率 的前一个值。

collectgarbage("setstepmul"): 返回 步进倍率 的前一个值。

collectgarbage("step"): 单步运行垃圾收集器。 步长"大小"由 arg 控制。 传入 0 时，收集器步进（不可分割的）一步。 传入非 0 值， 收集器收集相当于 Lua 分配这些多（K 字节）内存的工作。 如果收集器结束一个循环将返回 true 。

collectgarbage("stop"): 停止垃圾收集器的运行。 在调用重启前，收集器只会因显式的调用运行。

## Lua 面向对象

Account = {balance = 0} --成员变量

function Account.withdraw (v) --成员函数

Account.balance = Account.balance - v

end

function Account:add(v) --派生方法，：连接；调用派生方法.，:

v = v or 0 --必须为local变量v赋值，否则为nil；右边v为传入参数

Account.balance = Account.balance - v

end

function Rectangle:new (o,length,breadth) --获取对象

o = o or {}

setmetatable(o, self) --self==this？

self.\_\_index = self

self.length = length or 0

self.breadth = breadth or 0

self.area = length\*breadth;

return o

end

## Lua 数据库访问

require "luasql.mysql"

--创建环境对象

env = luasql.mysql()

--连接数据库

conn = env:connect("数据库名","用户名","密码","IP地址",端口)

--设置数据库的编码格式

conn:execute"SET NAMES UTF8"

--执行数据库操作

cur = conn:execute("select \* from role")

row = cur:fetch({},"a")

--文件对象的创建

file = io.open("role.txt","w+");

while row do

var = string.format("%d %s\n", row.id, row.name)

print(var)

file:write(var)

row = cur:fetch(row,"a")

end

file:close() --关闭文件对象

conn:close() --关闭数据库连接

env:close() --关闭数据库环境