chunk

A chunk is simply a sequence of commands (or statements)

lua 被作为数据描述语言

调用操作系统库函数os.exit()推出交互式模式，或者 ctl C/D退出

交互模式： lua –i progname，执行程序并进入交互模式，利于调试和测试，即可以访问可见的变量

运行函数块之dofile

dofile加载编译后的chunk到当前环境中，在当前环境中即可调用chunk中的函数或者变量

dofile function：immediately executes a file，testing a piece of code can be useful with dofile，dofile也可以做测试用

标识符命令规范

以下划线或者字母开头

标识符的第一个字母不可作为数字

驼峰式命名

\*下划线开头的命名应尽量避免

\_ : for a dummy variable(虚拟的变量)

lua的关键(保留)字

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

Lua is case-sensitive(大小写敏感)

--单行注释

--[[ 多行注释--]]

取消多行注释：在多行注释前加一个 ---[[ 多行注释--]]

Global variables do not need declarations

lua脚本参数的使用

usage of lua : lua [options] [script [args]]

lua中全局变量无需申明，直接使用；存取一个未初始化的变量会得到一个nil值，为变量赋nil值取消变量的引用

The -e option allows us to enter code directly into the command line(在命令行中直接运行lua脚本)

The -l option loads a library(加载库)

lua -lmylib 加载mylib库，去除后缀名(.lua)

The -i enters interactive mode after running the other arguments

交互式模式下，= varname ，取变量的值

预定义全局变量arg，脚本可以使用arg接收命令行参数

脚本的位置是0，之后的顺序加1，之前的顺序减1，一遍使用arg取得参数

the interpreter creates the table arg with all the command-line arguments, before running the script

Lua is a dynamically typed language，each value carries its own type

lua脚本开头： #!/usr/bin/lua

基本数据类型

lua是一个动态类型语言，lua中没有类型定义

eight basic types in Lua: nil, boolean, number, string, userdata,function, thread, and table

the result of type is always a string.

Variables have no predefined types,any variable can contain values of any type

(任一变量都是已知的八种类型之一)

functions are first-class values in Lua

nil在lua中是具有单一值的类型

Lua uses nil as a kind of non-value， you can assign nil to a global variable to delete it.

未赋值的全局变量具有一个nil值，nil用来表示缺省值的有效值

Lua中除false和nil为假，其余都为真，lua将0和空字符串是为真(true)

number:lua中没有整型数字，是实型数字(双精度浮点型)

十进制的数字能被二进制数有限表示出来的，则计算的时候是正常的值

integers do have exact representations and therefore do not have rounding errors

Strings

Strings：a sequence of characters，immutable values

(序列字符，不可变的值)

字符串是自动内存管理的对象，表和函数也是如此

程序中处理1M以上的字符串并不常见

取长操作符:字符串的长度 = #”字符串”

单引号和双引号并不区分字符串: “string“或者’string’

Literal string matching double square brackets

双方括号中的字符串可以多行输出(io.write(variable))

注释的小技巧：--[==[comment]==]

闭合代码块的小技巧：[=====[code]=====]

lua5.2中\z：跳过所有的字符，直到遇到非空白字符

Coercions 强制转换

Lua provides automatic conversions between numbers and strings at run time

lua在运行的时候自动字符串和数字转换，在字符串上面使用数学运算符，会使字符串向数字转换

lua会在使用字符串的地方将数字转换为字符串

.. is lua string concatenation operator

\*\* 数字后面跟 .. 需要用一个空格隔开，不然lua会将前一个数字误认为浮点数

tonumber a function convert a string to a number explicitly

tostring a function convert a number to a string

表类型是由关联数组实现的，关联数组可以由数字或者字符串来检索，空值(nil)除外

Tables are the main (in fact, the only) data structuring mechanism in Lua, and a powerful one

io.read():使用read关键字符串作为表io的索引

Lua uses tables to represent packages and objects as well

Tables in Lua are neither values nor variables; they are objects(表是对象，既不是值也不是变量)

table as a dynamically allocated object，program manipulates only references (or pointers) to them

A table is always anonymous(表作为一个动态分配的对象，程序使用引用来操作表的值)

lua中的表无需申明，使用 {} 来创建表

表是匿名的，在表的变量名和表之间无确定的关系

一个表可以被多个变量来引用，并且在一个变量改变表的时候，另一个变量获得改变后的值。

当表没有被引用时，lua的内存回收器会删除表，并重用内存

每张表能以不同类型的索引来存储值

表字段未被初始化的时候是空值(nil)，也可以为表字段赋nil来删除字段

lua使用普通的表来存储全局变量

可以使用点选的方式为表字段赋值，也可以使用[index]来表字段赋值；点选清楚的表示我们把表作为记录使用，而[index]表示可以把任意字符串作为关键字

a.x 以”x”作为索引，即a[“x”]

a[x] 以变量x作为索引，即a[x]，索引值即为x的值

index 检索（查找）

对于lua中的表来说，不需要申明大小，lua中表的下标以1开始

sequence：无空值的序列，#返回最后一个元素的下标

Functions are first-class values in Lua(和基本数据类型一类的值)

可以把函数传给一个变量，也可以将函数作为参数传给另一个函数，可以把函数作为返回值，lua提供了函数嵌套，函数在面向对象编程中扮演了一个很重要的角色

Lua中，函数可以以变量存储，当作参数传递，作为结果返回

Lua中可以调用lua函数和C函数，借助C函数中的便利性和性能

The userdata type allows arbitrary C data to be stored in Lua variables(允许任意的C数据存储到lua变量中，用户数据表示C语言中库或函数中的新类型)

Userdata are used to represent new types created by an application program or a library written in C;

Lua中可以用 == 比较数字和字符串

Lua中逻辑判断有 and or not

Everything is right but for false and nil

not returns a bool always.

Concatenation in lua is ..(作用于表和字符串上)

# 长度运算符：字符串返回字符的个数，表返回元素的个数

#atable -- atable的长度

btable[#btable] -- btable的最后一个元素

btable[#btable]=nil – 删除btable的最后一个元素

btable[#btable + 1]=value -- 增加一个元素

Lua的表中，位置i上元素为nil，则位置i的值被剔除

Lua中除 .. 和 ^外(右结合)，都是左结合，必要时，显示用小括号

构造器是初始化和创建表的表达式({}--构造器，创建了一个空表)

序列：数字作为表的关键字的集合

a = {1,2,3,4,nil,nil,nil,5,6,7}

print(#a) – 10

a = {1,2,3,4,5,6,7,nil,nil,nil}

print(#a) -- 7

分号分隔不同的数据类型部分

表的三种初始化：

t = {“1”，“2”，“3”}

t = {x = 1, y = 2}

t = {}; t.x = 1, t.y = 2

不管那种情况，表时可以随时添加和移除的，而且我们还可以在构造器中混合记录式或者列表式风格，不允许使用负数或者不合适的标识符来初始化构造

abc = {x=10, y=45; "one", "two", "three"}

abc.x →10 abc[1] →one

lua表中 [ ] 索引键后面的变量会替换

{x=10, y = 12} == {[“x”] = 10, [“y”] = 12}

statements : assignment, control structures, and procedure calls

lua support multiple assignments and local variable declarations

表达式

表达式计算出值，表达式包括数字常量、变量、字符串常量、一元和二元操作符和函数调用

|  |  |
| --- | --- |
| 二元操作符 | +、-、\*、/、%、^ |
| 一元操作符 | - |
| 关系运算符 | <= >= ~= == |
| 逻辑运算符 | and not or |

两者都为假则为假，或者nil为假，其余都为真

and：如果第一个参数为假(false)则返回第一个参数，否则返回第二个参数

or：第一个参数为非假(true)则返回第一个参数，否则返回第二个参数(第一个参数为false)

x = x or v(不是x就是v，当x为空值时，则x默认值为v)

a and b or c(a不是b就是c)

not not true(结果时true)，not总是返回相反的结果

取模操作的定义：

a % b == a – math.floor(a/b)\*b

对于整数，结果总是与右边的计算后的值的符号相同

x % 1是分数的分子部分(小数部分)，x – x%1 是整数部分

lua中类型不同，则不相等；特别的，nil == nil -> true

即nil只和它自己相等

lua中以引用来比较表和用户数据的相等性，引用的是同一个对象则相等

排序运算符仅用于数字和数字、字符串和字符串之间

连接符： ..

create local variables with the local statement

Statement

lua支持常用的语句块，赋值、控制、循环和函数(过程)调用，还有多重赋值和局部变量申明

lua中的多重赋值，先计算所有的值，然后再赋值，lua会自动舍去多余的值和补全缺少的值(nil)，因而再使用多重赋值的时候要两边对齐，多重赋值并不比单行赋值快

局部变量被限制在它自己的申明块中

A block is the body of a control structure, the body of a function, or a chunk

In interactive mode, each line is a chunk by itself

交互式模块中，一行就是一块(chunk)

用do-end显式声明块

局部变量避免全局变量的命名混乱，局部变量存取也比全局变量更快，在函数运行结束后自动消亡

使用局部变量编程是一种好的风格

Local variables help you avoid cluttering the global environment with unnecessary names.

可以在任意写语句的地方写局部变量，它的范围从声明处到结束

Lua中的惯例local foo = foo，将全局的foo的值赋予局部的foo，局部变量在它申明之后可见，作用是保护全局变量的值和加快对局部变量的存取，仅在你需要的时候申明变量

by declaring a variable only when you need it which increases readabilityt

Control Structures： if ... then ... end、 while ... do ... end、for ... do ... end、repeat ... until(条件执行到为真的时候结束循环，在循环中申明的变量在until处仍然可见)

while ...do ... end

If ... then ... end

If ... then ... elseif ...then ... end

1 递增的for

for var = exp1, exp2, exp3 do

<something>

end

This loop will execute something for each value of var from exp1 to exp2, using exp3 as the step to increment var.

for的特性：每个表达式部分只被计算一次，控制变量只在循环体内可见(局部变量)，不要改变控制变量的值

2 一般(通用)for

迭代器的for：控制变量在循环体内可见，不应再赋值

The generic for loop traverses all values returned by an iterator function

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

the loop variables are local to the loop body and you should never assign any value to them.

(本地循环变量在循环体内时局部的)

The break and return statements allow us to jump out of a block，The goto statement allows us to jump to almost any point in a function

This statement breaks the inner loop (for, repeat, or while) that contains it,break cannot be used outside a loop

A return statement returns occasional results from a function or simply finishes the function(return返回函数临时的结果)

若不写return，函数隐式返回一个空值。一般情况下，return位于函数末尾，也可以在函数中的任意位置作为错误处理用

label name

小心的使用goto会提高程序的质量

label has two colons followed by the label name followed by more two colons like this ::name:: .think twice before using a goto,goto只应该影响局部范围

标签应遵循可见性原则，避免跳出函数，标签应在局部变量以外，标签被认为是空语句

labels are considered void statements

goto is also useful to write state machines

goto遵循常用的可见性规则，其次标签不能调出函数，再次标签不能跳进局部范围内

do ... end包含语句块

局部变量的范围在非空语句块之前结束，标签被视为空语句

goto对写状态机很有帮助，数据驱动程序是更好设计的

Functions

Functions are the main mechanism for abstraction of statements and expressions in Lua

函数是抽象语句或者表达式的主要机制

小括号里面的参数列表引起函数调用，若无参数，需要用一个空的参数列表引起调用，字符串或者表除外，即他们无需小括号也能调用

能完成某项特殊的任务（过程或子例程），有返回值

语句改变状态，表达式有返回值

一般情况下，函数调用要使用 ( ),但若参数是一个字面量或一个表时，函数( )可以省略

面向对象的调用符号“： ”

lua中的程序可以使用lua或者C中定义的函数(或者其他任意一种语言定义的函数)

参数作为局部变量使用，调用函数的时候，可以使用任意数量的参数，lua会忽略无效的参数

函数定义：function functionname(paralist) ...(body) end

参数列表只在函数体内起作用（局部变量），lua会调整传进来的列表参数，多余的忽略，少的赋予nil

多值：lua中函数可以返回多值，用多变量接收，返回的时候也要返回多个变量

Lua也会调整函数的返回值，多余舍弃，不足为nil

函数表达式不是最为最后一个语句调用时，只会产生一个值

函数的返回值作为唯一参数传递时，所有的返回值都被返回传入

表会把函数调用的返回值作为元素集合起来（作为最后一个表达式调用）

函数调用在其他位置仅会返回一个值

return functionname() 会返回函数的所有值，多加一个(functionname()) 强制返回一个值

lua中会调整多重赋值：多重赋值，return 多个值的语句，多参数函数调用，表构造

函数无返回值或者没有期望的结果, lua为缺失的值赋予nil

函数调用不是在元素列表的最后一个位置，则多返回值的函数只会返回一个值

当函数调用不是在参数列表的最后一个位置时，lua会调整多返回值的函数只会返回第一个返回值

当多重返回值函数调用是作为最后或者唯一的参数，多重返回值函数的结果全部返回给调用函数

总的来说，在最后一个位置的多重返回值函数会返回多个值，其余位置只返回一个值

return f() 返回f的返回值，无论f是单值还是多值，goes as follow function f

在多重函数调用的时候多加一个小括号，强制多重返回值函数返回单个值

函数的返回语句处不需要额外的小括号，若加了小括号，则函数返回多值的地方只会返回单值

lua中的table.unpack({ ... })返回表中所有的元素，对unpack的一个重要应用是通用调用机制(以任意参数动态的调用任意函数)，就像这样f(table.unpack(a))

unpack(atable,startindex,limitnumber)

可变参数函数：function functionname( ... ) ... end

可变参数函数：函数可以接收任意数量的可变参数

参数列表中的 ... 暗示着函数是可变的，乱的内部会收集所有的参数，... 称为变量参数表达式

function foo (a, b, c)

function foo (...)

local a, b, c = ...

格式文本：string.format

文本写入：io.write

Variadic functions can have any number of fixed parameters before the variadic part.(在可变参数部分前面可以有任意数量固定的参数)

使用{ ... }把可变参数收集起来

table.pack()接收任意数量的参数，并返回一个新表，且有字段n，表示参数的个数

{ ... } 比 table.pack() 简洁和快速

具名参数：有名字的参数，在lua中，使用表的机制来完成居民参数的传递

lua在函数参数传递的过程中，按照位置来获取参数值

参数传递机制是按照参数传递的位置来获取值的

函数进深

first-class values：a function is a value with the same rights as conventional values like numbers and strings.

lexical scoping：functions can access variables of their enclosing functions.

(局部函数可以存取包含它的函数的变量)

返回函数调用的所有参数：

function all(...) return ... end

在lua中，可以把函数当作一个值来使用，函数可以用变量来存储，可以当参数传递，也可以位于表中

函数是匿名的，并没有名字，可以用变量名来调用函数

函数定义是一个语句块(创建了一个函数类型的值，并赋给一个变量)， 在lua中，函数定义更适合说成是赋值语句，调用函数时会返回函数类型的值

abc = function() ... end or function abc() ... end

function constructor：function (x) body end

anonymous function(匿名函数)

table constructor：{ }

table sort：receives a table and sorts its elements.

order function: a function that receives two elements and returns whether the first must come before the second in the sorted list.

闭包

闭包：在函数内定义的函数可以访问本函数外的局部变量

a closure is a function plus all it needs to access non-local variables correctly

Lexical scoping, plus first-class functions, is a powerful concept in a programming language

闭包提供了一种有效的参数传递方式，闭包对函数也很有用处

闭包在不同的上下文中也很有用处

闭包还能营造安全的环境simplicity and flexibility

在lua中，除非显式申明变量为局部的，否则是全局的

非全局函数：将函数位于表的字段中，或者是把函数存于局部变量中, 即限制于特定的范围

Lib = {}

Lib.foo = function (x,y) return x + y end

Lib.goo = function (x,y) return x - y end

构造器：

Lib = {

foo = function (x,y) return x + y end,

goo = function (x,y) return x - y end

}

不常见的定义

Lib = {}

function Lib.foo (x,y) return x + y end

function Lib.goo (x,y) return x - y end

Lua handles each chunk as a function，a chunk can declare local functions, Lexical scoping ensures that other functions in the package can use these local functions.

Example:

local f = function (<params>)

<body>

end

local g = function (<params>)

<some code>

f() -- ’f’ is visible here

<some code>

end

另一种定义局部函数的示例

local function f (<params>)

<body>

end

局部递归函数定义的模式：

local function foo (<params>) <body> end

lua的前向申明：

local f, g -- ’forward’ declarations

function g ()

<some code> f() <some code>

end

function f ()

<some code> g() <some code>

end

lua的尾调用真是牛(尾递归消除就是尾调用所指)

形式作用上像 goto

尾调用：直接在调用函数的位置返回尾函数的返回值，不保存调用函数的栈空间信息，当return 尾函数后，调用点直接到调用函数的那个位置，程序接着往下执行

function f (x) return g(x) end

tail calls use no stack space

It will never overflow the stack

尾调用的一个标准是在调用函数后什么也不做，更为严格的格式是return func(args)

tail call：calling function has nothing else to do after the call(有点毛病)，调用函数在函数被调用后什么也不做

tail-call elimination：尾调用在调用函数结束后无需返回到其调用位置，即跳到函数的下一个位置

在lua中，用一个闭包和函数组成，每次调用一个这个函数，闭包保持其原来的状态和闭包变量，每次调用的时候，返回下一个值

closure is a function that accesses one or more local variables from its enclosing environment

closure：what is a value in Lua is the closure, not the function.

lua中可以调用原来的库函数来定义新的函数

闭包：存取非局部变量及其相关的东西

迭代器

迭代器：允许你反复的构造出一个集合中的元素，也就是反复的取出集合中的元素

lua中以函数来表示迭代器，即每次调用函数，它返回下一个元素

迭代器需要在连续的调用中保持一些状态，以便下次调用处理。而闭包提供了很好的这种机制

a closure is a function that accesses one or more local variables from its enclosing environment

闭包是一个能存取本地变量的函数，这个本地变量在包含它的函数里，通过闭包的连续调用，这些变量能保持它的值

闭包包含两个函数：闭包本身和一个构造工厂(创建闭包和封闭的变量，也称为非局部变量)

泛型for：内部自动保持迭代函数，每一次新的调用，会自动调用迭代函数

泛型for的一般语法：

for <var-list> in <exp-list> do

<body>

end

for迭代：

1 计算表达式，得出迭代函数，不变状态，控制变量的初始值

2 以不变状态和控制变量为参数调用迭代函数(实际上只会传递控制变量的值给迭代函数)

3 迭代函数返回的值赋给变量列表

注：若迭代变量的值为nil，迭代结束

迭代函数为f，不变状态为s，控制变量为a0，则有：

a0=f(s,a2)

a1=f(s,a0)

a2=f(s,a1)

直到an为nil结束循环

每迭代一次，状态更新一次，并在下一次状态中更新

控制变量的值为a0，则迭代器是这样的：

a1=f(s,a0) a2=f(s,a1) a3=f(s,a2) a4=f(s,a3)

无状态迭代器：迭代的过程中，迭代器不保持任何状态

迭代器的三要素：循环不变式(状态量)，迭代器，控制变量

what iterates is the for loop. Iterators only provide the successive values for the iteration.

迭代器接受函数内部的返回值

循环不变量：也就是对象本身正在发生着的事所产生的一种状态

lua 内置原始函数：next

for循环调整表达式列表为三个结果：迭代器函数、循环不变量、控制变量

复杂状态的迭代器：把一个迭代器打包在一个表中，并将表作为循环不变量

迭代器仅为迭代提供连续不断的值，也可以称为产生器

真正的迭代器：将函数作为参数传递，并在循环中调用这个参数

编译、执行和错误

lua在运行之前会把源代码编译成中间代码的形式，解释型代码明显的特征是飞一般的执行代码(运行速度超快)

编译：运行lua 代码块的原始操作之dofile，dofile是loadfile的一个辅助函数

loadfile 从文件中加载代码块，却并不运行，它只编译块，把编译后的块作为函数返回，也不引发错误，而是返回错误码

assert 断言会引发错误，如果错误发生，loadfile 会返回nil和错误信息

如果程序多次运行，则loadfile加载一次，多次调用结果，loadfile只编译一次

load函数从一个字符串中加载块，load后的块会变为一个函数

load(s)()：加载块，并调用这个函数，若s 中有语法错误，会返回nil和错误信息,load是没有词法界定的，总是把块编译到全局环境中.通常来说，使用load加载字符串是没有必要的,典型的load用法是加载外部的代码

Note that load expects a chunk, statements

如果想要执行表达式，在表达式前加return，load返回的是一个常规的函数，load连续的调用reader直到返回nil

(nil是块的结束标志)

注：load 加载一个字符串，并自行并执行

Lua把单独的chunk视为匿名带有可变参数的函数的主体

function (...) a = 1 end

chunk中也可以申明局部变量

f = load("local a = 10; print(a + 20)")

f()称为函数的调用

load函数从不引发错误，有错误的时候只是返回nil和错误信息

函数定义是赋值操作，发生在运行时，不在编译时，作为高质量的代码，应该处理加载块时报出的错误

f = loadfile("foo.lua")

要定义foo.lua 中的函数，需要运行f()后才会定义foo.lua 中的函数

预编译的代码

Lua在运行之前会预编译源代码

预编译格式：luac –o prog.lc prog.lua

执行预编译代码(二进制)：lua prog.lc

Lua accepts precompiled code mostly anywhere it accepts source code.(只要在接收源代码的地方，lua也接收预编译代码)

load和loadfile也接受预编译后的代码（binary chunk 二进制代码）

arg[index]:获取命令行参数，程序名的位置时0，向后加1，向前减1

将代码编译成二进制代码，加载更快

预编译后的代码并不总是比源码小，但是加载更快，其次，保护代码不会意外的改变

避免不确信的预编译的代码被执行

string.dump:接收lua函数，作为字符串返回预编译代码

C 代码

加载库：loadlib 函数加载给定的库，并把lua链接进去。

package.loadlib()需要的参数是库的路径名和库中的函数名

查看是否可以使用动态连接：

print(package.loadlib("a","b"))，若提示No such file or directory：可以使用动态链接，否则不行。

lua的loadlib只加载库，并不调用函数，将C中的函数返回当作Lua函数使用，若初始化函数或加载函数出错的时候，返回nil和错误信息

通常使用require函数加载C库，require搜索库，并用loadlib为库初始化函数，当调用的时候，这个初始化函数从库中创建并返回带有那个函数的表

错误

Lua是一门可扩张的语言，常嵌入到应用中，当发生错误的时候，lua结束当前块并返回到应用程序中

对错误的处理常有捕获错误和error函数抛出（引发）

断言(assert):如果条件为false，则引发错误。当函数发现异常，会返回错误码并引发错误

an exception that is easily avoided should raise an error; otherwise, it should return an error code.

简单的异常应该引发错误，除此应返回错误代码

读数字：io.read("\*n")

在调用函数之前，lua总是会计算参数值

文件打开：io.open

防御式编程：保证函数参数的正确性，停止计算和提示一个错误信息。简单的操作是使用断言操作来确保操作安全

错误和异常处理:通常你不需要处理lua中的错误，由应用程序来处理，lua中的调用始于应用程序，lua返回错误代码一遍应用程序处理错误

lua中是可以用pcall（protected call）来封装你的代码,即把会出现错误的代码放在pcall()函数中,pcall()函数使用保护模式来调用第一个参数，若函数调用没有错误，则返回true并携带调用的返回值，否则返回false和错误信息

错误信息和回溯

通常错误信息时描述错误的字符串，此外，错误信息是传递给error函数的值

定位信息：文件名和行号

error 的第二个参数level，表示错误的级别，定位错误信息

debug.debug which gives you a Lua prompt so that you can inspect by yourself what was going on when the error happened

debug.traceback which builds an extended error message with a traceback.

traceback：引发错误的完整的栈调用

xpcall(function,[debug.debug/debug.traceback 消息处理器])

协程

有自己的栈变量、局部变量和结构体指针，但是协程之间共享全局变量，一个可执行的流水线

协程和线程的区别是协程同一时刻只能运行一个，线程可并发运行

协程使用coroutine table包装协程相关的函数

create创建一个新的协程，有一个函数作为参数，并且coroutine .create的返回值是thread，代表一个新的协程

通常create的参数是一个匿名的函数

协程的四个状态：挂起，运行，死亡和正常态，用coroutine .status来检查协程的状态

当协程创建时，处于挂起状态，并不会自动运行，使用coroutine.resume唤醒协程(挂起->运行->结束)

yields 将运行着的协程挂起，以便以后唤醒

all activity that happens while it is suspended is happening inside its call to yield. When we resume the coroutine, this call to yield finally returns and the coroutine continues its execution until the next yield or until its end

从协程的角度看：当在协程内部调用yield时，协程就会挂起

唤醒协程时，调用yield的地方会最终返回协程的执行，再次遇到yield时挂起或者运行结束(协程会挂起直到到下一次执行遇到yield时)

当协程结束时再次唤醒，则会引发错误，此时的协程已经结束

resume以保护模式来运行，协程中的出错信息会返回到resume的调用处

当一个协程调用另一个协程的时候，这个协程不是挂起状态亦非运行状态，称为正常状态，但是却不能被唤醒

协程的一个很有用的地方是resume和yield配对出现以交换数据使用

resume调用返回true则表示没问题，传递给相关的yield

语法上来说， resume传进来的多余的参数，yield会返回

当协程调用结束时，相关的resume调用主函数的返回其他值

管道和过滤器

协程正是多线程中的一种

管道在上下文环境中是很重要的，进程的切换代价很大，而协程切换则很便捷

当一个协程调用yield，它不进入新的函数，并处于挂起状态，等待resume调用；当调用resume的时候，也不开始一个新的函数，而是等待yield调用

使用管道，每个人物可以分开处理，使用协程，每个人物可以以分开的协程处理

协程作为迭代器

协程提供的特性：调用者和被调用者里外相互切换

lua的通用模式：在函数的内部包装一个给协程的resume调用函数

coroutine.wrap()创建一个协程，但并不返回协程，而是返回一个函数，这个函数被调用的时候会唤醒协程，当发生错误的时候函数会引起错误

coroutine.wrap()：使用函数唤醒协程调用，却缺少灵活性。wrap创建的协程不能检查协程创建时候的状态，亦不能检查运行时候的错误。

每一个协程等价于一个线程，协程的resume和yield配对使用交换线程的控制权。

与常规的多线程不同，协程是非抢占式的，协程在运行的时候，不能从外部停止，只有显示的调用yield才会停止执行。

没有抢占式的话，编程会容易些，不用担心同步的bugs。

只要在外面的边界确定协程yileds。

非抢占式(不可抢占):whenever any thread calls a blocking operation, the whole program blocks until the operation completes.(当线程操作块的时候，整个块要等到操作完成)

To download a file, we must open a connection to its site, send a request to the file, receive the file (in blocks), and close the connection.

表与对象

数据结构：Tables in Lua are not a data structure; they are the data structure.

列表 = 记录 + 指针

Lua中实现数组：使用整数来检索表，表在lua中是对象

# lua中是长度操作符，可以用循环表达式来初始化，也可以用单行表达式来初始化，通常以下标1开始

可以使用任意数值来作为数组的索引

矩阵：

1表里面嵌套表

2组合两个分段到一个分段中

(即二维数组用一维数组来表示)

pairs 遍历非空元素(即空值会跳过)，键在表中没有严格的顺序

链表：节点用表来表示，以字段作为表的接性，即链接。

lua中的表是动态的实体，表的字段包含其(指向)他表的引用

链表使用表实现：

list=nil

list={next = list, value = v}

遍历链表：

local l = list

while l do

<visit l.value>

l = l.next

end

栈：使用数组（无边界）来表示栈

队列：使用表里的lib函数即可，insert和remove. 更为有效的方式是用两个下标来指示表的开头和结尾。为了避免污染全局空间，将操作定义到表中。lua使用双精度来表示数字

集合与包

集合：元素唯一，范围有限。将集合元素作为表中的索引

示例：

reserved = {

["while"] = true, ["end"] = true,

["function"] = true, ["local"] = true,

}

for w in allwords() do

if not reserved[w] then

<do something with ’w’> -- 'w' is not a reserved word

end

end

辅助函数：

function Set (list)

local set = {}

for \_, l in ipairs(list) do set[l] = true end

return set

end

reserved = Set{"while", "end", "function", "local", }

包(多维集合)：集合中的每个元素会出现多次，即为每一个key元素计数；在添加集合时，计数加；移除集合时，计数减

字符串缓存：使用表来代替字符串缓存，table.concat()

table.concat()：返回指定字符串list的连接

lua 一次性读取整个文件：io.read("\*a")

.. 字符串连接符，先分配足够大小的空间，再拷贝字符串

示例：

local t = {}

for line in io.lines() do

t[#t + 1] = line .. "\n"

end

local s = table.concat(t, "\n") 此处的回车作为分隔符使用

数据文件与持久性

处理数据文件的时候，写数据比读数据跟简单

一个健壮的程序应该处理错误情况

表构造器为文件格式提供了可变性，用lua的表来构建我们自己要使用的格式

因为表是一个独立的单元体，因而Entry{code} is the same as Entry({code})， {code}每加载一次，Entry自动执行一次

表作为单一参数传函数，即可以写作：Entry{code}

自描述的数据格式，每一条数据都有一条简短的说明，更具可读性，也便于修改

数据文件，即以自描述的格式来编写数据，如name-value

Lua不仅运行的很快，编译的也很快

序列化数据

将数据转为字节流或字符流，便于存进文件或者通过网络连接来传送

表达式会返回一个值

a = 'I can say: "nothing" \\string'

a = string.format("%q",a)

print(a)

长字符串的声明使用：[=[...(strings)]=] \*两边的等号相等

手写代码注释提供一种安全的方式：[=[...]=]

序列化：以指定的想要的格式格式化字符串

在lua中，可以不用全局变量名而保存一个值

元表与元方法

lua中的值通常有一个可预测的操作集合，使用元表可以添加表，比较函数，调用字符串

面对一个未定义的操作，元表允许我们改变值的行为

元表对应的方法称为元方法，lua中的每一个值都有一个相关联的元表，元表和用户数据有单独的元表，其它类型的值共享一个单一的元表

lua总是创建无元表的新表，使用setmetatable 改变其他表的元表

t = {}

print(getmetatable(t)) --> nil

t1 = {}

setmetatable(t, t1)

print(getmetatable(t) == t1) --> true

\*lua的元表该怎么理解

lua中只对表设置元表，使用C代码(会导致不可重用的代码)来操作元表其他类型的值，默认的其他类型无元表

print(getmetatable("hi")) --> table: 0x80772e0

print(getmetatable("xuxu")) --> table: 0x80772e0

print(getmetatable(10)) --> nil

print(getmetatable(print)) --> nil

元表常用到的两个方法：getmetatable 和setmetatable

Any table can be the metatable of any value;a group of related tables can share a common metatable,a table can be its own metatable

元表任意的值可以是任意的表，一组相关的表也可以共享一个公共的表（描述公共的行为），一个表也可以本身的元表(描述自己特有的行为)；任意的配置都是合理有效的

为使namespace不受命名的影响，通常在表中申明字段

元表定义了自己的操作

Lua查找元方法的时候，会检查第一个值的元表是否有元方法，若有，则使用这个元方法，否则检查第二个值的元表是否有元方法，并使用这个值的元方法，在使用之前先检查操作数的合法性

相关联的元方法：

算术元方法：

|  |  |
| --- | --- |
| \_\_add | add |
| \_\_mul | multiplication |
| \_\_sub | subtraction |
| \_\_div | division |
| \_\_unm | negation |
| \_\_mod | modulo |
| \_\_pow | exponentiation |
| \_\_concat | concatenation |

关系元方法：

|  |  |
| --- | --- |
| \_\_eq | equal to |
| \_\_lt | less than |
| \_\_le | less than or equal to |

以上元方法是虚拟机检测元表的元方法操作所涉及的值用到的core元方法

调用元方法的时候：

1查看元表的第一个值是否有元字段(lua使用这个字段做为元方法)

2 查看第二个值是否有相同的元方法

此外，lua会引发错误

IEEE 754标准规定NaN(Not a Number) 代表一个未定义的值，任何涉及到NaN的比较都会返回一个false

对两个比较的对象：若元方法和对象的基本类型不同，则对象不等

通常的实践是为库的元表定义自己的字段

库定义的元方法

元表也是一个普通的表，任何人都可以使用它，所以可以在库中的元表定义自己的字段

tostring 以一种更简单的格式来表示表，print总是调用tostring来格式化它的输出，tostring在格式化的时候会检查这个值是否有 \_\_tostring 的元方法，tostring 调用元方法来完成格式化，把对象当作参数来传递，然后元方法返回tostring的结果

函数 setmetatable 和 getmetatable 也被使用做元字段，以此来保护元表。若为元表设置\_\_metatable字段，getmetatable会返回这个字段的值

保护元表，为元表设置\_\_metatable字段，这样用户既不能看到元表，也不能更改元表，getmetatable会返回元表的\_\_metatable字段的值，然而使用setmatatable会引起错误

表存取元方法

存取本表中没有的字段的时候，lua会去寻找元表中\_\_index的元字段，并返回\_\_index的值，前提是要为表设置元表

算术或逻辑元方法为未知的方法定义了新的处理情况，但却并未改变语言的行为。

lua为查询和修改表中没有的字段提供了一种方式来改变表的行为

\_\_index元方法

当存取一个表中没有的字段，会得到一个nil值

当存取元表中的字段，会引发解释器寻找一个\_\_index的元方法，若没有index元方法，则返回空，否则返回元表中\_\_index字段预定的值

描述属性的时候有两种方法：

1 可以用构造函数来为属性赋予初始值；

2 用表来继承上一表的字段

在lua中使用\_\_index 作为继承是很常见的方式

\_\_index 可以是元方法，也可以是一个表；

\_\_index作为函数的时候，lua以表来调用\_\_index，并将表中没有的键作为参数传递；

\_\_index作为表的时候，lua重新存取表中的字段

将表当作\_\_index元方法使用提供了一种更快和更简单的继承的实现方法，\_\_index是函数的时候提供了灵活性，但花费更高

不调用\_\_index而获取一张表的时候，使用rawget函数

当不调用\_\_index 元方法时，会调用rawget(t,i)，即先获取表，在取得值

\_\_newindex 元方法

为表存取表更新\_\_index的的元方法，当为表中没有的字段赋初始值，解释器会查找\_\_newindex元方法，若有\_\_newindex ，解释器调用\_\_newindex而非赋值。

和\_\_index类似，\_\_newindex是表的话，则赋值给这个表，取代原始的表。

rawset(t,k,v)为表t设置k、v键值对

为表设置默认值

function setDefault (t, d)

local mt = {\_\_index = function () return d end}

setmetatable(t, mt)

end

当表存取其中没有的字段的时候会引发表对\_\_index的调用

闭包，表自己的内部环境

使用单个元表为不同的表提供不同的初始值：

不担心命名冲突："\_\_\_" 作为表特有的字段

local mt = {\_\_index = function (t) return t.\_\_\_ end}

function setDefault (t, d)

t.\_\_\_ = d

setmetatable(t, mt)

end

担心命名冲突：

local key = {} -- unique key

local mt = {\_\_index = function (t) return t[key] end}

function setDefault (t, d)

t[key] = d

setmetatable(t, mt)

end

关联表自己的默认值是使用分离表(切片[分隔]是表，值是表的默认值)，如弱表；另一种是记住元表，为表重用相同的元表

追溯表的存取

当检索在表中的字段不存在的时候，\_\_index 和 \_\_newindex才会相关联

存取一张表的时候是让其为空

遍历一张表，定义\_\_pairs作为入口：

mt.\_\_pairs = function ()

return function (\_, k)

return next(\_t, k)

end

end

检测管理多个表：将每个代理表与原始表先关联，并未所有的代理表共享一个原始表

只读表

当尝试更新表的时候引起错误

\_\_index：使用原始表，对每个只读的元表，\_\_index指向原始的表

lua环境

lua将全局变量保存在一张普通的表中(global environment)

一方面表的结构简化lua内部的实现，其次操作全局变量的表和普通的表一样

lua维护全局变量，即使在表中也会维护，在多个环境中也会维护全局的变量，将全局环境存在 \_G中

lua将环境存储到全局变量\_G中

输出全局环境：for n in pairs(\_G) do print(n) end

动态命名的全局变量

使用赋值足够存取和设置全局变量了

获取变量的值(常规的写法)：

value = loadstring("return " .. varname)()

上面的做法会创建和编译一个新的块，代价昂贵

动态获取全局变量的值：value = \_G[varname]

动态设置全局变量的值：\_G[varname] = value

getfield("io.read") 获得表中的字段

为全局变量申请新的环境

申明新的全局变量：

1 rawset(\_G, name, initval or false)

检测赋值是否在主块中，使用debug库，调用debug.getinfo(2,"S").what，返回表的what字段,这个字段告诉我们调用元方法的函数是主块，还是常规的lua函数或者是C函数

\*erro函数会中止函数的运行

测试变量是否存在：rawget(\_G,varname)

不允许全局变量为空值，若为空值的话认为是未申明的全局变量

使用辅助表来存储已申明变量的名字，无论元方法被调用与否，会检查表中的变量是否被申明

strict.lua模块实现了全局变量检测

全局变量申明

获取全局动态的变量名的值：value = \_G.variablename

修改全局动态的变量名的值：\_G.variablename = value

非全局环境

全局环境的一个影响是，一旦更改，会影响到所有的部分程序

lua中，全局变量并非是全局的，甚至可以说lua没有全局变量

自由命名(free names)，自由变量

未绑定到显示申明的名称，换句话说，在局部变量范围内没有出现以这个名称命名的变量，自由变量并未涉及到全局变量

lua将自由变量编译转换成\_ENV.var，即var这个自由变量被编译成\_ENV.var，lua将预定义的upvalue(非局部变量)的块编译成\_ENV

local \_ENV = <the global environment>

return function (...)

\_ENV.var1 = \_ENV.var2 + 3

end

注：the var1 field of the global environment gets the value of the var2 field plus 3

全局环境的var1字段获得了全局变量的var2字段加上3的值

Lua treats any chunk as an anonymous function

lua将任何的块视作匿名函数

1. Lua compiles any chunk in the scope of an upvalue called \_ENV.(非局部变量范围的块编译做\_ENV)
2. The compiler translates any free name var to \_ENV.var.(编译器将自由变量var转换作\_ENV.var)
3. The load (or loadfile) function initializes the first upvalue of a chunk with the global environment.(初始化一个块中的第一个upvalue到全局环境中 )

\*前两条是编译器做的

使用\_ENV

在lua的交互式模块中，每一行都是一个不同的块

单一块：将代码置于单独的文件中或者放在do...end之间

\_ENV是一个普通变量，可以像其他变量一样进行赋值和存取操作。\_ENV=nil使得直接存取全局变量无效，控制代码使用

显式调用全局的变量：

a = 13 -- global

local a = 12

print(a) --> 12 (local)

print(\_ENV.a) --> 13 (global)

\_ENV的通常用法是通过医端代码来改变环境，移植环境的时候，使用\_G来取得原先的环境，以便在新的环境中使用

\_G和普通变量一样的变量名，这个特别的状态发生在lua创建初始化全局表并把这个表赋值给全局变量\_G的时候

使用继承来获得原先的环境，然而，通过此种方式，任何的赋值都会赋给新表；如此，即使在全局环境中改变了一个变量也不会影响，尽管可以使用\_G来改变全局环境中的变量

a = 1

local newgt = {} -- create new environment

setmetatable(newgt, {\_\_index = \_G})

\_ENV = newgt -- set it

print(a) --> 1

-- continuing previous code

a = 10

print(a) --> 10

print(\_G.a) --> 1

\_G.a = 20

print(\_G.a) --> 20

\_ENV作为普通变量，遵循变量的范围原则。特别的，函数内部定义的块存取\_ENV好像函数内部存取外部变量一样：

\_ENV = {\_G = \_G}

local function foo ()

\_G.print(a) -- compiled as '\_ENV.\_G.print(\_ENV.a)'

end

a = 10 -- \_ENV.a

foo() --> 10

\_ENV = {\_G = \_G, a = 20}

foo() --> 20

如果定义一个局部变量\_ENV,则引用的自由变量会绑定到\_ENV新变量上

a = 2

do

local \_ENV = {print = print, a = 14}

print(a) --> 14

end

print(a) --> 2 (back to the original \_ENV)

function factory (\_ENV)

return function ()

return a -- "global" a

end

end

f1 = factory{a = 6}

f2 = factory{a = 7}

print(f1()) --> 6

print(f2()) --> 7

范围原则的使用：将多个函数共享到公共环境中，或将改变改变环境的函数共享到其它函数中

load 初始化已加载的 \_ENV非局部变量的块到全局环境中

\_ENV 和 load

env = {}

f = loadfile("config.lua", "t", env)

f()

配置文件将会运行到空的环境env中，更重要的是配置文件的定义会加载到env环境中，因此配置文件不会产生影响

拒绝服务式攻击，仅仅是浪费了CPU的时间和内存

多次运行一个块，每次都在不同的环境表中:debug.setupvalue 允许改变一个函数的任意非局部变量(upvalue)的值

f = loadfile(filename)

...

env = {}

debug.setupvalue(f, 1, env)

参数说明：函数名，upvalue的下标,upvalue的新值

下表通常是1，load 或者loadfile的结果是一个函数，lua确保只有一个upvalue，且upvalue是\_ENV

lua将块编译作可变的函数

模块与包机制

为不同的人共享代码提供方便，包是模块的集合

tables, functions, metatables, and environments

可以通过表，函数，元表，环境来实现包(模块)

从用户的角度出发，模块是可以被require加载的一些代码，且require创建返回一张表；所有从模块返回的东西(函数、常量)，都定义在这张表中，这张表好像一个namespace一样

所有的库都是模块

模块：通过require加载的代码，并且返回一个表，此表相当于一个namespace

所有的标准库都是模块，导入模块：require “modulenam”

标准解释器预先加载所有的标准库：比如这样

math = require "math"

string = require "string"

使用表来实现模块就可以操作模块像使用表一样方便，在lua中，模块是first-class，可以当作变量、传递给函数的参数一样使用

调用模块中的函数

local moduleName = require “modulename”

1 moduleName.fun()

2 local f = moduleName.fun()

f()

require 避免多次加载一个模块，显式为加载的模块传递参数

一旦模块被加载，它就会被再次需要的程序重用

mod.init(0, 0) init函数返回模块本身

mod(0,0) 返回函数本身

模块仅被加载一次，模块解决初始化冲突

require函数

对于require，模块是一些定义值(函数or包含函数的表)的代码，这些代码返回包含模块函数的表，返回表的动作是由木块代码来做的，而非require；有些模块会返回一些其它的值

require 函数总是最小化模块的数量，对于require来说，模块是定义值的一些代码，并返回一个函数组合的表

package.loaded是一个包含模块的表

require"module" 的步骤：

1 检测模块是否在package.loaded 表中，若在，直接调用运行，若没在，requirez在lua 文件中搜索模块名，找到则用loadfile加载，loadfile(加载器)结果是函数；还没有找到的话，lua在C库中搜索模块名，若找到，则用package.loadlib加载，加载器(luaopen\_modname)结果是loadlib

要加载模块，lua使用两个参数：模块名和加载器中传来的文件名，若加载器返回一些值，require返回并存储这些值到package.loaded表中，下次调用相同的模块的时候返回相同的值

强制require运行两次：package.loaded.<modname> = nil

备注：require 和C的include一样，却又不完全一样

加载模块: require “modulename”

1. 检查package.loaded，看模块是否被加载，若已经加载，则从模块中直接取值，否则，从lua文件中搜索一遍，找到使用loadfile加载进来；若在lua文件中未找到，则去C库中查找，找到使用package.loadlib加载，并使用luaopen\_modulename函数加载进来
2. require加载需要两个参数，模块名和找到loader的文件名(加载器中传来的文件名)；

若加载器返回值，require返回这个值，并存在package.loaded表中，等待将来在同一个模块中取用

1. 为了重复加载模块，需要把前一次的模块清除

package.loaded.<modulename> = nil

重命名模块

重命名模块，避免命名冲突

模块搜索路径：会在一个标准的模板路径中去搜索，例如：

require搜索的路径是template(模板)列表，模板提供了一种可供选择的方式将模块名(require的参数)转化为文件名

路径中的摸吧是一个包含可选？的文件名，每个模板，require将模块置换为？，并检测路径中的文件是否有相同的模块名

路径中的模板被分号隔开

模块名 <--> 文件名

如路径为：?;?.lua;c:\windows\?;/sur/local/lua/?/?.lua

则require"sql"会寻找以下路径：

sql

sql.lua

c:\windows\sql

/usr/local/lua/sql/sql.lua

？代表模块名

require函数只认问号(?) 和 分号(;),其它的如目录分隔符和文件扩展符是由路径自己定义的

require搜索lua文件的路径总是在当前变量package.path的值中

lua运行的时候，使用环境变量LUA\_PATH\_5\_2初始化路径变量(package.path),若环境变量LUA\_PATH\_5\_2未定义，则lua定义环境变量LUA\_PATH，若都没有定义，则lua使用编译定义后的默认路径

使用环境变量的值的时候，lua会使用子串";;"来替换默认路径；如：LUA\_PATH\_5\_2 =“mydir/?.lua;;”，最终template的mydir/?.lua会放在默认路径后

搜索C模块以相同的方式，搜索路径在环境变量package.cpath中

类似的，环境变量package.cpath从LUA\_CPATH\_5\_2 or LUA\_CPATH 获得初始值

Unix 例子：./?.so;/usr/local/lib/lua/5.2/?.so

Note:路径定义了文件扩展名，.so 作为文件模板

Windows例子：.\?.dll;C:\Program Files\Lua502\dll\?.dll

函数package.searchpath 编码以上的规则来搜索库，接受模块名和路径，并在package.searchpath描述的规则搜索文件

若在，则返回第一个同名的文件，否则返回nil和错误信息，打开错误

> path = ".\\?.dll;C:\\Program Files\\Lua502\\dll\\?.dll"

> print(package.searchpath("X", path))

Searchers

搜索器是一个接收模块名和返回nil或者模块加载器的函数

package.searchers列出了require使用的搜索器，当查询以一个模块的时候，require传进模块名来调用列表中的搜索器，知道搜索器为模块找到加载器，若找不到，require引起错误

使用列表驱动搜索对require由很大的灵活性：如存储压缩后的模块，只需要提供搜索器函数并加到列表中

package.searchersmore默认配置，搜索器搜索lua文件和搜索C库，预加载搜索器允许任意函数的定义加载模块，package.preload是一张表，映射模块名到加载函数

搜索模块的时候，搜索器仅仅搜索在表package.preload中搜索给定的名字，找到函数的话，将其作为模块加载器返回，否则返回nil

package.preload搜索器提供了通用的方式来处理不便的情况

Lua中写模块的基本方法：

一：创建一张表，把想要导出的函数写在这个表中，最后在文件末尾返回表(返回这个表)

table = {}

table.fun1 = function() ... end

table.fun0 = function() ... end

return table

另一种是把模块直接赋值给package.loaded

local table = {}

package.loaded[...] = table

require将模块名作为第一参数传递给加载器，并调用

可变参数 ... 产生模块名

table.fun0 = function() ... end

table.fun1 = function() ... end

注：更推荐前一种，清楚明了

若模块没有返回value，require将会返回package.loaded[modname]当前的值

在模块的表中定义局部变量或局部函数，称为私有变量或函数

模块导出列表

return {

new = new,

i = i,

add = add,

sub = sub,

mul = mul,

div = div,

tostring = tostring,

}

使用环境

使用基本的方法很容易污染全局空间，如在私有申明的时候忘记加local关键字

环境提供了一种有趣的技巧来创建模块，以解决污染全局空间的问题

一旦环境的主块有单独的环境，不仅仅是模块自己的函数分享到返回的表中，而且模块的全局变量也会分享到返回的表中。

将共有函数作为全局变量申明，且函数会自动的把表分开

模块所需要做的是要返回的表赋值给环境变量\_ENV中

local M = {}

\_ENV = M

function add (c1, c2)

return new(c1.r + c2.r, c1.i + c2.i)

end

在相同的模块中无需前缀即可调用模块中的其它函数

这种方法如果忘记加local，也不会污染全局命名空间

避免错误的创建一个全局的：\_ENV = nil,一旦改变了\_ENV,我们就不能存取先前所有的全局变量；

解决办法：

1. 使用继承：

local M = {}

setmetatable(M, {\_\_index = \_G})

\_ENV = M

\*注：若\_ENV = M在setmetatable之前，则setmetatable会设置一个空的元表

1. 使用局部变量来接受全局的环境：

local M = {}

local \_G = \_G

\_ENV = M -- or \_ENV = nil

\*调用的时候加载前缀\_G.，这种存取很快

1. 仅为你需要的函数申明为局部：

-- module setup

local M = {}

-- Import Section:

-- declare everything this module needs from outside

local sqrt = math.sqrt

local io = io

-- no more external access after this point

\_ENV = nil -- or \_ENV = M

\*由于使用局部变量，运行得更快

引用模块：

local local\_name = require "module\_name"

子模块和包

包是完整得模块树，lua允许层级模块命名，用点来分隔

如require调用mod.sub模块，require 先查询package.loaded，表package.preload使用为原始的模块名mod.sub作为关键字

当搜索定义的子模块文件时，require将点(.)转化为系统目录分隔符，再去搜索

如路路是

./?.lua;/usr/local/lua/?.lua;/usr/local/lua/?/init.lua

require"a.b" 加载模块时，会

./a/b.lua

/usr/local/lua/a/b.lua

/usr/local/lua/a/b/init.lua

这种行为允许所有的模块在单一目录种存在(存活)

\*\*包时目录，模块是文件

目录分隔符在编译的时候被lua配置

C中的模块名不包含点，require将C中模块名中的点转化转化为下划线 a.b --> luaopen\_a\_b

连字符转化技巧：

模块名 mod/v-a a包下的子模块mod

require "mod.v-a"，如此require也会找到mod/v-a文件

require在加载C子模块的时候，它会有超过多余一个以上的搜索器，当require既不能找到lua文件，也不能找到C中的子模块时，最终会有一个搜索器在C路径中去搜索，此时搜索的是包名

conglua的观点来看，同意包下面的子模块没有显示的关系，请求模块a并不是自动加载它的子模块，类似的a.b也不会自动加载a

OOP(面向对象编程)

表在lua中是一个对象，不止有一种含义。和对象一样，表有状态、ID(和其它值分开，保持独立)和生命周期(在哪里创建和被谁创建时孤立开的)；两个表有相同的值，却不是相同的对象，对象在不同的时候有不同的值

对象有自己的操作，表也有自己的操作：

Account = {balance = 0}

function Account.withdraw (v)

Account.balance = Account.balance - v

end

method(方式):即一个对象特有的函数，表现为一个动词

函数只为特殊的对象服务，其二函数只为存储到特定的全局变量的对象服务

a, Account = Account, nil

a.withdraw(100.00) -- ERROR!

对象有自己单独的生存周期

操作接收器，其值为self或者this

function Account.withdraw (self, v)

self.balance = self.balance - v

end

使用self参数，可以为许多对象使用相同的方法；大部分的OO编程语言对程序员隐藏了self和this参数，但是可以在方法内部中使用冒号(:)操作符来隐藏：

Account = {balance = 0}

function Account:withdraw (v)

self.balance = self.balance - v

end

方法调用：Account:withdraw(100.00)

可以使用点来定义函数，以冒号来调用，或者相反

Account = { balance=0,

withdraw = function (self, v)

self.balance = self.balance - v

end

}

function Account:deposit (v)

self.balance = self.balance + v

end

Account.deposit(Account, 200.00)

Account:withdraw(100.00)

点定义函数的的副作用时要添加额外的隐藏的参数到方法定义中，且调用的时候要加额外的参数来调用；冒号则不需要

对象有标识符，状态，在这个状态下的操作

类系统，继承、私有

类

类，即创建对象的模型，每个对象是特定类的实例

原型：第一个对象查找一些未知操作的普通对象

为其它对象创建一个对象作为唯一的原型，类和原型都是将被多个对象共享的行为放在一起的地方

使用继承来实现原型

setmetatable(a, {\_\_index = b})

使得b成为a的原型。之后，a遇到未知的操作会去b中查询

继承不仅继承了原方法，还继承了新表中没有的字段

类不仅可以提供方法，也可以为实例的字段设置默认值，继承后的字段的默认值被修改了之后，存取新实例字段的时候，不会去检索index元方法。

b.balance = b.balance + v

此时b已经有自己的balance字段了

继承

类是对象，其不能从其它类中获得方法

如基类Account

Account = {balance = 0}

function Account:new (o)

o = o or {}

setmetatable(o, self)

self.\_\_index = self

return o

end

至此，SpecialAccount是Account的一个实例

SpecialAccount = Account:new()

s是从SpecialAccount继承而来，SpecialAccount是从Account继承而来

s = SpecialAccount:new{limit=1000.00}

new执行的时候，self参数引用到SpecialAccount，因此s的元表是SpecialAccount，s在字段\_\_index的值也是SpecialAccount

s:deposit(100.00)

lua在s中找不到deposit，因而转到SpecialAccount中查找，SpecialAccount中没有deposit，再转到Account中查找，有deposit方法，因而调用这个deposit方法

对SpecialAccount，重定义其超类中继承下来的方法

function SpecialAccount:withdraw (v)

if v - self.balance >= self:getLimit() then

error"insufficient funds"

end

self.balance = self.balance - v

end

function SpecialAccount:getLimit ()

return self.limit or 0

end

现在再调用withdraw方法，因为lua在SpecialAccount中找到了withdraw，故而不会到Account中

s:withdraw(200.00)

如此会去调用限额的withdraw方法

lua中关于对象有趣的一个方面是：你无需创建一个新类并指定behavior，如果一个对象需要一个特定的行为，可以直接在那个对象中实现

function s:getLimit ()

return self.balance \* 0.10

end

这样便不会调用SpecialAccount中的getLimit方法，而是会去调用最新定义的方法

多重继承

lua中的对象并不是原始数据类型

\_\_index方法实现OOP编程是集简单，灵活和性能的特点来的

\_\_index元字段用作函数，元表的元字段\_\_index是一个函数的时候，在原始表中找不到key的时候，会去调用\_\_index对应的元方法，\_\_index可以在多个父类中查找缺失的键。

多重继承指一个类可以有不止一个子类，因此不可以用类方法来创建一个子类。

尽管是多重继承，每个对象实例仍然属于单一类的实例，在单一类中查找所有父类的方法

类和子类的关系不同于类和实例的关系，一个类不能同时是实例和子类的元表

Named = {}

function Named:getname ()

return self.name

end

function Named:setname (n)

self.name = n

end

私有性

每个对象的状态保持自己的内部关系，控制对象的方法和字段对外部的可见性

lua中的对象设计并没有提供私有性机制，这是因为lua使用通用的表结构来表示对象的结构

如果不想再表中存取任何东西，那就别做

lua的另一个目标是灵活，为程序员提供元机制来模拟其它机制

用两张表来表示表示对象，一个表用来表示状态，一个表用来表示操作或者对外的接口。通过第二张表组合的接口来存取对象自己。

为避免非法存取表示对象状态的表不在另一张表的字段中，状态只保存在方法的闭包中

function newAccount (initialBalance)

local self = {balance = initialBalance}

local withdraw = function (v)

self.balance = self.balance - v

end

local deposit = function (v)

self.balance = self.balance + v

end

local getBalance = function () return self.balance end

return {

withdraw = withdraw,

deposit = deposit,

getBalance = getBalance

}

end

1 创建一张保持内部对象状态的表，存储到self变量中

2 函数创建对象的方法

3函数创建并返回一个外部对象，将方法名映射到实际的方法实现

关键是方法不会将self作为外部参数，而是直接对self存取

以上方式对存取到self表中的任何对象完全满足私有性

可以将对象的私有属性都存取到这个表中，可以定义私有的方法

Single-Method Approach

(单一方法设计)

将表示对象的单一方法返回，迭代器保存状态的是单一方法对象；此外单一方法出现在基于不同参数来分派不同任务的方法

单一对象的实现举例如下：

function newObject (value)

return function (action, v)

if action == "get" then return value

elseif action == "set" then value = v

else error("invalid action")

end

end

end

d = newObject(0)

print(d("get")) --> 0

d("set", 10)

print(d("get")) --> 10

使用单一闭包的代价比使用一个表的代价低，如此便有了完全的私有性：通过对象单独的方法来存取一个对象的状态

弱表和终结器

lua自动管理内存，使用垃圾(空间)回收器删除垃圾对象。

lua使用弱表和终结器来协助垃圾回收器

弱表允许收集程序仍可存取的lua对象；终结器允许未在垃圾回收器控制下直接存取的外部对象的收集

弱表

垃圾回收器仅仅收集确定是垃圾的东西，它不确保你所认为的垃圾被收集

存取到全局变量的对象对lua来说并非垃圾，尽管全局变量只会用到一次或者不被用到，为变量赋值nil，解除引用

一旦对象成为垃圾回收器的一部分，垃圾回收器便不会再去收集

弱表机制：告诉lua防止对象引用再次引用

(弱引用)weak reference：

不被lua垃圾回收器认为的对象引用

如果所有的引用指向的对象都是弱的，那么对象会被收集，引用会被删除

弱表：弱表中的进入都是weak的，如果一个对象仅在弱表中存在，lua会收集弱表中的对象

表有键和值，两者都可以是任意的对象；正常情况下，不会收集键和值是对象的表，且键和值对表是可存取的

键和值是强引用类型，防止引用的对象再次引用；在弱表中，键和值也是弱的，分三种情况：表中的键是弱的，表中的值是弱的，表中的键和值都是弱的

若键和值都被收集了，从表中的进入将会无效(消失)

表的虚弱性是由元表字段\_\_mode而赋予的，这个字段的值是一个字符串

|  |  |
| --- | --- |
| \_\_mode="k" | 表中的键是弱的 |
| \_\_mode="v" | 表中的值是弱的 |
| \_\_mode="kv" | 表中的键和值都是弱的 |

a = {}

b = {\_\_mode = "k"}

setmetatable(a, b) -- now 'a' has weak keys

key = {} -- creates first key

a[key] = 1

key = {} -- creates second key

a[key] = 2

collectgarbage() -- forces a garbage collection cycle

for k, v in pairs(a) do print(v) end

--> 2

以上代码：第二个key的赋值重写(覆盖)了第一个赋值，collectgarbage强制垃圾回收器进行一次完整的收集，故第一个key的引用没有了，表中相关的进入也移除了，而第二个key，仍被变量key引用，因此没有回收

只有对象可以再弱表中回收，值(数字和布尔)不被回收

如果在表中插入一个数字键，回收器键不会移除；如果表中的弱值的键的值是数字的话，仍会被回收，整个进入也会从表中移除。

表和线程是显示的创建的,{}系统创建一个值；从程序员的角度上看，字符串是值，非对象；和数字，布尔一样，字符串不会被移除

备忘录函数

编程的通用技巧是时空互换，存储函数的计算的结果以提速程序运行

local results = {}

function mem\_loadstring (s)

local res = results[s]

if res == nil then -- result not available?

res = assert(load(s)) -- compute new result

results[s] = res -- save for later reuse

end

return res

end

弱表解除内存占用的问题：

setmetatable(results, {\_\_mode = "v"}) -- make values weak

setmetatable(results, {\_\_mode = "kv"})-- make fully weak

对象属性

弱表的另一个重要的地方是关联对象的属性；将属性关联到对象的情况：

函数=>函数名、数组=>数组大小、表=>表的默认值

对象是一个表的时候，可以用唯一的键值把属性存储到表自身当中，小技巧是创建一个表，做为唯一键值

如果对象不是表的时候，将无法为其保存自身的属性；有时需要用原始对象来存储属性

属性关联到对象

外部的表为属性关联到表提供了一种理想的方式，表有时候也叫关联数组；将对象作为键，将属性作为值。外部表可以为任意的类型的对象保存属性，只要lua允许将任意的对象作为键

保存在外部表中的属性不与其它对象的接口，而且与表一样的私有

一旦将对象用作表中的键，即锁住了对象，lua不回收用做键的对象；如果使用一张普通的表来关联函数到函数名，这些函数将不会被回收，使用弱表可以解决这个缺点。弱键不阻止key回收，只要没有其它引用到key，key是可以回收的。

另一方面，表不能有弱值，存活对象的属性可以回收

以默认值访问表

设置默认值的技巧：对象属性和存储化记忆

使用弱表来关联每个表的默认值：

local defaults = {}

setmetatable(defaults, {\_\_mode = "k"})

local mt = {\_\_index = function (t) return defaults[t] end}

function setDefault (t, d)

defaults[t] = d

setmetatable(t, mt)

end

用不同的元表来设置不同的默认值，重用相同的元表存储化记忆：

local metas = {}

setmetatable(metas, {\_\_mode = "v"})

function setDefault (t, d)

local mt = metas[d]

if mt == nil then

mt = {\_\_index = function () return d end}

metas[d] = mt -- memorize

end

setmetatable(t, mt)

end

使用弱值lua允许回收不再使用的弱表

以上两种实现默认值的比较：第一种实现需要为每张表的默认值花费存储空间，第二种的实现需要较多的内存空间(表、闭包、元进入等)。如果有很多的表，要设置较少的默认值，使用第二种，如果较少的表要共享公共的默认值，则使用第一种。

临时表

存储化记忆的例子：

function factory (o)

return function () return o end

end

do

local mem = {}

setmetatable(mem, {\_\_mode = "k"})

function factory (o)

local res = mem[o]

if not res then

res = function () return o end

mem[o] = res

end

return res

end

end

value (the constant function) associated with an object in mem refers back to its own key (the object itself)

从弱表标准的编译器来说，记忆化表种的任何懂西都不会移除，值对函数是强引用，每个函数和相关对象关联，因而总是对每个对象强引用，对象也不会被回收，尽管key是弱的。

闭包和对象关联，对象又返回闭包

a table with weak keys and strong values is an ephemeron table(弱键强值的表是ephemeron table)

在ephemeron table中，键的可存取性控制着相关值的可存取性

在ephemeron table 中的 entry (k,v)

只有对k是强引用的，对v才能是强引用的，此外，entry将会从表中移除，尽管v和k是相关(直接或者间接)的。

终结器

lua垃圾回收器是回收lua对象的，也能帮助程序释放外部资源

终结器是一个与要回收的对象关联的函数

lua使用元方法 \_\_gc 来实现终结器：

o = {x = "hi"}

setmetatable(o, {\_\_gc = function (o) print(o.x) end})

o = nil --擦除连接，调用collectgarbage强制垃圾回收

collectgarbage() --> hi

在回收的时候，lua检测表不再可存取，因而调用终结器方法

lua终结器的细微之处在于标记要终结对象的概念，标记要终结的对象，为其元表设置非空的\_\_gc元方法；若不标记对象，则对象不会被终结。

如果不设置\_\_gc元方法，会出现一些奇怪的现象：

o = {x = "hi"}

mt = {}

setmetatable(o, mt)

mt.\_\_gc = function (o) print(o.x) end

o = nil

collectgarbage() --> (prints nothing)

尽管为元表设置了\_\_gc元方法，但lua并不检测赋值，所以lua不标记这个对象。在使用元表之后改变元方法并不常见。

若要在之后改变\_\_gc元方法，先为\_\_gc元方法设置一个任意的初始值：

o = {x = "hi"}

mt = {\_\_gc = true}

setmetatable(o, mt)

mt.\_\_gc = function (o) print(o.x) end

o = nil

collectgarbage() --> hi

lua仅在终结器是一个函数的时候，才会调用\_\_gc元方法

如果回收器在一段周期内终结了几个对象，回收器会以相反的顺序收集标记为终结的对象

mt = {\_\_gc = function (o) print(o[1]) end}

list = nil

for i = 1, 3 do

list = setmetatable({i, link = list}, mt)

end

list = nil

collectgarbage()

--> 3

--> 2

--> 1

第一个终结的对象是最后一个被标记的，一个常见的误解是连接的顺序影响终结的顺序，然而连接会形成环，因此不影响终结的顺序。

但终结器调用的时候，终结器将被终结的对象作为参数获得，因此，对象在终结的时候激活，称为transient resurrection

(瞬时存在)。

终结器运行的时候，没有什么能阻止终结器存储全局变量中的对象，所以到终结器返回的时候，对象仍可存取

Resurrection must be transitive：

A = {x = "this is A"}

B = {f = A}

setmetatable(B, {\_\_gc = function (o) print(o.f.x) end})

A, B = nil

collectgarbage() --> this is A

B获取A的终结器，在B终结之前，A是不会被回收的。lua在运行终结器之前必须保证A和B是存活的(resurrect-复活)

带有终结器的对象回收分两个阶段：

1 回收器检测带有终结器的对象不可到达(未绑定)，回收器激活对象，并将其排到终结队列中，一旦终结器运行，lua将队列中的对象标记未终结对象。到下一次回收器检测到对象不可到达，然后将其删除。如果要确保整个程序中的垃圾全被释放，调用collectgarbage两次，第二次调用collectgarbage将会删除第一次调用终结的对象

终结器对每个对象只运行一次，因为lua将终结的对象放在一起标记。如果在程序运行结束的时候对象仍未被删除，但整个lua状态是闭合的时候，lua将会调用其终结器来删除对象。

atexit函数：在程序结束之前函数即刻运行，我们所做的是在表中创建一个终结器，并固定终结器到某个地方，比如说全局变量。

\_G.AA = {\_\_gc = function ()

-- your 'atexit' code comes here

print("finishing Lua program")

end}

setmetatable(\_G.AA, \_G.AA)

另外一个有趣的技巧是，每当lua完成一个回收周期的时候允许程序调用一个指定的函数。终结器只运行一次，小技巧是使终结器创建一个新的对象来运行下一次终结器：

do

local mt = {\_\_gc = function (o)

-- whatever you want to do

print("new cycle")

-- creates new object for next cycle

setmetatable({}, getmetatable(o))

end}

-- creates first object

setmetatable({}, mt)

end

collectgarbage() --> new cycle

collectgarbage() --> new cycle

collectgarbage() --> new cycle

带有终结器的对象和和弱表也有一点细微的差别，收集器再弱表中的值唤醒之前清除弱表中的值，键在唤醒之后再清除：

-- a table with weak keys

wk = setmetatable({}, {\_\_mode = "k"})

-- a table with weak values

wv = setmetatable({}, {\_\_mode = "v"})

o = {} -- an object

wv[1] = o; wk[o] = 10 -- add it to both tables

setmetatable(o, {\_\_gc = function (o)

print(wk[o], wv[1])

end})

o = nil; collectgarbage() --> 10 nil

对象的终结器运行的时候，它找到弱表wk中的对象，但却不在wv，原理是使用带有weak keys的表来保持对象的属性，终结器可能会需要存取这些属性。然而，我们使用weak value 的表来重用存活的对象，如此，对象终结后便不再使用

The Standard Libraries(标准库)

一、数学库

math 库组合了一些标准数学函数：如三角函数、指数函数、取舍函数，max、min函数等，还有数学变量pi、huge(无穷大)。

度和弧度的转换：degrees(deg) and radians(rad)

随机数函数：math.random

1 无参数的调用，返回0到1之间的实型数字

2 单个数字调用，返回1到n之间的整型数字

3 两个参数嗲用，返回l到u之间的整型数字

种子函数：randomseed，使用当前时间作为种子，以便产生不同的随机数，math.randomseed(os.time())，返回当前时间的数字，通常是秒数os.time。

math.random 使用标准C库中的rand函数

二、bitwise库

将移位操作换成算数操作，左移位为乘2的n次，右移位为除2的n次

位库：bit32

bit32.band(0xDF, 0xFD)

bit32.bor(0xDF, 0xFD)

bit32.bnot(0x0)

bit32.bxor(0xDF, 0xFD)

band, bor, 和 bxor可以接受任意数量的参数，bitwise可以操作无符号整数

左移位：lshift()

右移位：rshift()和arshift()

循环左移位：lrotate()

循环右移位：rrotate()

出了算术上的移位，所有的移位后会以0来填充空白位，算术上的移位拷贝符号位上的位来填充

示例代码1：

printx(bit32.rshift(0xDF, 4)) --> 0xD

printx(bit32.lshift(0xDF, 4)) --> 0xDF0

printx(bit32.rshift(-1, 28)) --> 0xF

printx(bit32.arshift(-1, 28)) --> 0xFFFFFFFF

printx(bit32.lrotate(0xABCDEF01, 4)) --> 0xBCDEF01A

printx(bit32.rrotate(0xABCDEF01, 4)) --> 0x1ABCDEF0

示例代码2：

printx(bit32.lrotate(0xABCDEF01, -4)) --> 0x1ABCDEF0

printx(bit32.lrotate(0xABCDEF01, -36)) --> 0x1ABCDEF0

printx(bit32.lshift(0xABCDEF01, -36)) --> 0x0

printx(bit32.rshift(-1, 34)) --> 0x0

printx(bit32.arshift(-1, 34)) --> 0xFFFFFFFF

示例代码3：btest band

print(bit32.btest(12, 1)) --> false

print(bit32.btest(13, 1)) --> true

示例代码4：从数字中提取一个字节，若最后一个参数未提供，ze默认提取一位

bit32.replace，它的参数和extract一样的意思

note：bit32.extract(x,f,w) returns w bits from x, starting at bit f

bit32.extract(0xABCDEF01, 4, 8) --> 0xF0

bit32.extract(0xABCDEF01, 20, 12) --> 0xABC

bit32.extract(0xABCDEF01, 0, 12) --> 0xF01

等价性:

(bit32.replace(x, bit32.extract(x, f, w), f, w) == x)

三、表库

表库将操作函数的表当作数组使用，未数组提供删除、插入、排序、链接字符串登函数。

table.insert: 将元素擦汗如指定位置，若未指定位置，则插入到数组的最后一个位置。

常用小技巧：t[#t+1]=line

table.remove：

返回给定位置的元素并删除，默认移除数组最后一个位置的元素

table.sort：排序表中的元素

table.concat：链接表中的元素

四、字符串库

字符串调用方法：string.upper(s) or s:upper()

取长度：string.len(s) #s

重复字符串：string.rep(s,n) or s:rep(n)

小写字母：string.lower(s)，其他字符不变

取子串：string.sub(s,i,j)，取i-j的字符串

负数从末尾开始取子串，字符串是免疫的(不可更改)，因而lua返回的是一个新的字符串而非改变了的字符串。

若想更改变量的值，使用赋值来覆盖。

string.char--将数字转换未字符

string.byte--将指定的字符转换为数字，默认是第一个，也可以是负数

lua 5.1 以后string.byte(s,i,j)，返回i到j之间的字符所转换的数字

{s:byte(1,-1)} 返回字符串开始到结尾的字符所对应的字节

string.char(table.unpack(t))，还原字节对应的字符

string.format格式化字符串：

|  |  |
| --- | --- |
| %d | 数字(decimal) |
| %x | 十六进制(haxadecimal) |
| %o | 十进制(octal) |
| %f | 浮点数(float) |
| %s | 字符串 |
|  |  |

模式匹配函数：

find, match, gsub(全局替换),gmatch(全局匹配)

string.find：搜索指定对象的字符串，返回匹配到的字符串下标的起始位置和终止位置；可选参数指定从哪里开始搜索字符串子串

string.match：返回匹配到的子串

string.gsub：返回替换后的字符串

s = string.gsub("Lua is cute", "cute", "great")

print(s) --> Lua is great

s = string.gsub("all lii", "l", "x")

print(s) --> axx xii

s = string.gsub("Lua is great", "Sol", "Sun")

print(s) --> Lua is great

限制替换次数：

s = string.gsub("Lua is cute", "cute", "great")

print(s) --> Lua is great

s = string.gsub("all lii", "l", "x")

print(s) --> axx xii

s = string.gsub("Lua is great", "Sol", "Sun")

print(s) --> Lua is great

string.gmatch 返回一个在字符串中出现的模式的迭代函数

words = {}

for w in string.gmatch(s, "%a+") do

words[#words + 1] = w

end

%a+ alphabetic(字母序列)

package.searchpath

function search (modname, path)

modname = string.gsub(modname, "%.", "/")

for c in string.gmatch(path, "[^;]+") do

local fname = string.gsub(c, "?", modname)

local f = io.open(fname)

if f then

f:close()

return fname

end

end

return nil -- not found

end

模式：character classes：模式中的项，匹配特定集合中的字符

. all characters

%a letters

%c control characters

%d digits

%g printable characters except spaces

%l lower-case letters

%p punctuation characters

%s space characters

%u upper-case letters

%w alphanumeric characters

%x hexadecimal digits

s = "Deadline is 30/05/1999, firm"

date = "%d%d/%d%d/%d%d%d%d"

print(string.sub(s, string.find(s, date))) --> 30/05/1999

%A 所有非字符

magic characters：用于模式匹配具有特殊含义的字符

% 和\ 是转义字符

对于lua解析器来说，patterns are regular strings，模式是常规的字符串，只是模式函数将字符串作为模式解释

char-set：字符集合，再方括号中允许你创建字符类

[字符集]

[^0-7] ^——取集合的补集

通配符：

+ 1 or more repetitions 获得匹配模式的最长序列

\* 0 or more repetitions 获得匹配模式的最短序列

- 0 or more lazy repetitions

? optional (0 or 1 occurrence)

%(%s\*%)：%s\*匹配零个或更多个空格

[\_%a][\_%w]\*：数字或下划线开头的序列，后面跟着零个或更多的下划线和字母的标识符

|  |  |
| --- | --- |
| + | 匹配一个或多个字符，获得匹配最初序列 |
| \* | 和+类似，接受零次字符的出现 |
| - | 匹配零个或多个字符，匹配最短序列 |
| ？ | 匹配可选字符 |
| ^ | 匹配开头 |
| $ | 匹配结尾 |

例1：

test = "int x; /\* x \*/ int y; /\* y \*/"

print(string.match(test, "/%\*.\*%\*/"))

--> /\* x \*/ int y; /\* y \*/

例2：

test = "int x; /\* x \*/ int y; /\* y \*/"

print(string.gsub(test, "/%\*.-%\*/", ""))

--> int x; int y;

[+-]?%d+：?使得 +- 号可选

string.find(s, "^%d") 匹配数字

%b：匹配平衡字符串，%bxy，匹配以x之前的开始字符，匹配以y之后的结束字符，%b()’, ‘%b[]’, ‘%b{}’, or ‘%b<>’等

s = "a (enclosed (in) parentheses) line"

print(string.gsub(s, "%b()", "")) --> a line

%f[char-set] 边界模式：匹配空串，前一个字符不在字符集合中，后一个字符在字符集合中，然后返回边界字符串之前的字符

s = "the anthem is the theme"

print(s:gsub("%f[%w]the%f[%W]", "one"))

--> one anthem is one theme

%f[%w]：非字母数字和字母数字之间的字符

%f[%W]：字母数字和非字母数字之间的字符

捕捉机制：保存匹配到的部分模式字符串以便将来使用，在括号中写出你想要捕捉的字符串模式。

pair = "name = Anna"

key, value = string.match(pair, "(%a+)%s\*=%s\*(%a+)")

print(key, value) --> name Anna

|  |  |
| --- | --- |
| %a+ | 非空字符串 |
| %s\* | 空的空白字符串，如" " |
| %d | 单个数字 |
| %0 | 改变整个匹配 |

date = "Today is 17/7/1990"

d, m, y = string.match(date, "(%d+)/(%d+)/(%d+)")

print(d, m, y) --> 17 7 1990

\*仅仅保存第几次捕捉

["'].-["']：匹配单引号或双引号之间的字符

s = [[then he said: "it's all right"!]]

q, quotedPart = string.match(s, "([\"'])(.-)%1")

print(quotedPart) --> it's all right

print(q) --> "

%[(=\*)%[(.-)%]%1%]：匹配长字符串，左右相同的方括号之间的字符串

p = "%[(=\*)%[(.-)%]%1%]"

s = "a = [=[[[ something ]] ]==] ]=]; print(a)"

print(string.match(s, p)) --> = [[ something ]] ]==]

字符替换gsub：

%d在替换字符串中，当替换发生的时候

%作为字符的时候需要使用%来转义，即%%

字符串切剪

function trim (s)

return (string.gsub(s, "^%s\*(.-)%s\*$", "%1"))

end

替换

将函数或者表作为参数传递给string.gsub

调用函数：每当string.gsub匹配到的时候调用函数，对参数的调用就是捕捉，并且函数返回的值被用作替换字符串

调用表：string.gsub 使用第一个捕捉作为键值来查找表，键相关联的值用作替换字符串。

若函数调用的结果或者表中查找的结果为空(nil)，gsub不做匹配。

如：

function expand (s)

return (string.gsub(s, "$(%w+)", \_G))

end

name = "Lua"; status = "great"

print(expand("$name is $status, isn't it?"))

--> Lua is great, isn't it?

对于每一次匹配$(%w)，gsub在全局表\_G中查找捕捉名，然后将查找到的结果替换匹配到的字符。

print(expand("$othername is $status, isn't it?"))

--> $othername is great, isn't it?

当全局表\_G中没有查找到键的时候，则不会发生替换。

如果不确定给定的变量是否有字符串值，使用tostring来返回替换后的值。

function expand (s)

return (string.gsub(s, "$(%w+)", function (n)

return tostring(\_G[n])

end))

end

print(expand("print = $print; a = $a"))

--> print = function: 0x8050ce0; a = nil

URL编码

HTTP协议发送URL参数，将特殊的字符编码成%xx的格式，(xx是十六进制编码)之后会将空格转换为+号，之后会将键值之间用 = 隔开，在键值对之间用&号连接。

如：name = "al"; query = "a+b = c"; q="yes or no"

编码为：name=al&query=a%2Bb+%3D+c&q=yes+or+no

解码URL路径，将值存在表中，使用相关的值来索引：

解码函数：

function unescape (s)

s = string.gsub(s, "+", " ") -- 将字符+ -> 空格

s = string.gsub(s, "%%(%x%x)", function (h)

return string.char(tonumber(h, 16))

--每匹配带一次，匿名函数将字符以16进制转换为十进制，然后返回相关字符

end)

return s

end

print(unescape("a%2Bb+%3D+c")) --> a+b = c

对于键值对(name=value)的解码，使用gmatch，键和值中不能包含 &和= ，故使用模式[^&=]+ 来匹配。

cgi = {}

function decode (s)

for name, value in string.gmatch(s, "([^&=]+)=([^&=]+)") do

name = unescape(name)

value = unescape(value)

cgi[name] = value

end

end

captures(捕捉):小括号包含的匹配字符串

function encode (t)

local b = {}

for k,v in pairs(t) do

b[#b + 1] = (escape(k) .. "=" .. escape(v))

end

return table.concat(b, "&")

end

t = {name = "al", query = "a+b = c", q = "yes or no"}

print(encode(t)) --> q=yes+or+no&query=a%2Bb+%3D+c&name=al

Table扩展

()：空捕捉，不是什么都不捕捉，而是捕捉()所在子串的位置

print(string.match("hello", "()ll()")) --> 3 5

以字符串扩展表的位置捕捉：

function expandTabs (s, tab)

tab = tab or 8 -- tab "size" (default is 8)

local corr = 0

s = string.gsub(s, "()\t", function (p)

local sp = tab - (p - 1 + corr)%tab

corr = corr - 1 + sp

return string.rep(" ", sp)

end)

return s

end

function unexpandTabs (s, tab)

tab = tab or 8

s = expandTabs(s)

local pat = string.rep(".", tab)

s = string.gsub(s, pat, "%0\1")

s = string.gsub(s, " +\1", "\t")

s = string.gsub(s, "\1", "")

return s

end

模式匹配是操作字符的强有力的工具，可以将任何复杂的操作转换为string.gsub()，需要小心使用；模式匹配不仅仅是一个解析器的替换。

^(.-)%$：

获得第一个$字符串中的文本，^如果在第一个位置没有找到匹配，^告知算法停止搜索。

empty pattern(空模式匹配)：匹配空字符串

在开头或结尾以模式限定符 - 的模式是无意义的，总是会匹配空串。

包含 '.\*'的模式会匹配到超出你预期的匹配，扩展字符匹配

match a long line：a sequence that less than 70 character

pattern = string.rep("[^\n]", 70) .. "[^\n]\*"

编码：

UTF-8:the dominant encoding for Unicode on the Web,ASCII 的可移植性

UTF-8包含了所有ASCII范围内的字符编码，ASCII是小于128的单字节

lua是8比特的，可以像普通字符串一样读写，存储UTF-8字符。

文字字符串可以包含UTF-8的数据。字符串连接符对UTF-8字符同样起作用，字符串比较操作符对UTF-8的比较遵循Unicode的编码。

操作系统库和IO库是底层系统的主要接口，对UTF-8的支持依赖于底层系统；Linux系统下使用UTF-8作为文件名，而Windows使用UTF-16来命名；所以为了操作Windows中的Unicode编码文件名，需要一些外部的库，或者在Lua标准库中做一些变化。

字符串函数：

string.reverse, string.byte, string.char, string.upper,

string.lower。所有这些函数都是视一个字符为一个字节

string.format 和string.rep，除了 %c(一个字符一个字节)选项外，对UTF-8字符串处理没问题。

string.len 和 string.sub 对UTF-8的字符串处理也没问题

对于模式匹配的函数，对UTF-8的适用性依赖于模式；文字匹配模式没问题，UTF-8对任意字符的编码不会在其他字符的编码中出现(编码的唯一性)，字符类和字符集合仅对ASCII码字符。如 %s对UTF-8是字符串，当时它会匹配ASCII码的空白字符，当时它不会匹配特别的Unicode空白字符。

#(string.gsub(s, "[\128-\191]", ""))

gsub从字符串中移除连续的字节，故留下来的是一个字节和多行序列的开始字符，一个字符一个字节。

输出UTF-8字符：

for c in string.gmatch(s, ".[\128-\191]\*") do

print(c)

end

联合编码有许多特性；尽管字符的概念是广泛的，在联合编码字符和字母之间没有一一对应的关系。可区别的字符标记着"完全可忽略"的字符。其他的基本概念在在不通的语言中也会有所区别，比如字母

IO库

IO库为文件操作提供了两种不同的模型：输入文件和输出文件，简单的模型对于简单的操作是方便的。对于高级的文件操作，比如同时读取或写入多个文件，就需要一个完整的模块。

简单模型

简单模型将所有操作都放在两个当前文件，库会初始化当前输入文件(stdin)和当前输出文件(stdout)

io.read():从标准输入读取一行

io.input()和io.output():改变当前输入输出文件

io.input(filename):以读取的模式打开给定的文件，并将起设置为当前文件，从这一点来说，所有的输入都会来自这个文件。

知道其他的来调用io.input()函数，单出现错误的时候两个函数都会引发错误。

write比read更简单，io.write()获得任意大小的字符串参数，并将其写入当前输出文件，它会使用基本转换规则将数字转换为字符串。使用string.format来控制转换后的格式。

io.write("sin (3) = ", math.sin(3), "\n")

--> sin (3) = 0.14112000805987

io.write(string.format("sin (3) = %.4f\n", math.sin(3)))

--> sin (3) = 0.1411

io.write(a,b,c)比io.write(a..b..c)消耗更少的资源来完成同样的效果

使用print来调试运行快速，格式较差的程序；使用write来完全控制输出的程序，

print("hello", "Lua"); print("Hi")

--> hello Lua

--> Hi

io.write("hello", "Lua"); io.write("Hi", "\n")

--> helloLuaHi

和print不一样，write不会输出多余的字符，比如换行符和制表符；write可以重定向输出，print总是用于标准输出。最后，print总是对它的参数使用tostring函数，这对于调试时便捷的。

io.read从当前输入文件读取字符串，参数控制符：

|  |  |
| --- | --- |
| \*a | 读取整个文件 |
| \*l | 读取下一行，不包括换行符 |
| \*L | 读取下一行，包含换行符 |
| \*n | 读取数字 |
| num | 读取n个字节的字符 |

io.read("\*a")读取整个当前输入的文件：

t = io.read("\*a") -- read the whole file

t = string.gsub(t, ...) -- do the job

io.write(t) -- write the file

读取当前输入到当前输出：

for count = 1, math.huge do

local line = io.read()

if line == nil then break end

io.write(string.format("%6d ", count), line, "\n")

end

一行一行的迭代整个文件，使用io.lines()迭代器

读取n个字节

while true do

local block = io.read(2^13) -- buffer size is 8K

if not block then break end

io.write(block)

end

io.read(0)测试文件结束符

Complete I/O Model

file handle：文件句柄，表示打开文件的位置

确保文件打开正确：

local f = assert(io.open(filename, mode))

io库提供了三种C语言预定义的流：io.stdin、io.stdout 和 io.stderr.

直接输出错误信息：io.stderr:write(message)

我们可以混合简单模型和完整模型，调用io.input()获得当前输入。调用io.input(handle)设置当前的句柄。

io.lines参数可以是文件名，也可以是文件句柄

若以文件名为参数，io.lines会以读取模式打开，并在到达文件尾的时候关闭。若以文件句柄打开文件，则读取完后不会打开关闭文件；若是无参，则从当前文件输入读取。

lua整个读取文件比一行一行读物文件快。

local lines, rest = f:read(BUFSIZE, "\*l"):

读取被块阶段的剩余的字符

二进制文件

io.input 和 io.output总是以文本模式打开

在Unix中二进制文件和文本文件没有区别；在Windows中打开二进制文件必须以特殊的标志位('b')打开,lua处理二进制文件和文本文件一样，lua中的字符串可以包含任意字节的，几乎所有库中的函数都可以处理任意字节，只要模式不包含0个字节，可以使用 %z来匹配0个字节。

可以以 \*a 模式来读取整个二进制文件。

将Windows格式的 \r\n 转化为 Unix格式的 \n ：

local inp = assert(io.open(arg[1], "rb"))

-- 以文本或者二进制模式打开

local out = assert(io.open(arg[2], "wb"))

local data = inp:read("\*a")

data = string.gsub(data, "\r\n", "\n")

out:write(data)

assert(out:close())

输出二进制文件中的所有字符串：

local f = assert(io.open(arg[1], "rb"))

local data = f:read("\*a")

local validchars = "[%g%s]"

local pattern = "(" .. string.rep(validchars, 6) .. "+)\0"

for w in string.gmatch(data, pattern) do

print(w)

end

输出二进制文件：

local f = assert(io.open(arg[1], "rb"))

local block = 16

for bytes in f:lines(block) do

for c in string.gmatch(bytes, ".") do

io.write(string.format("%02X ", string.byte(c)))

end

io.write(string.rep(" ", block - string.len(bytes)))

io.write(" ", string.gsub(bytes, "%c", "."), "\n")

end

文件的其他选项

tmpfile函数以读写的模式返回临时文件的句柄，程序结束的时候自动删除。

flush函数刷新缓冲区中的东西，调用io.flash()刷新当前输出文件；f:flash()刷新特定的文件。

setvbuf()设置缓冲流的模式：

|  |  |
| --- | --- |
| no | 禁止缓存 |
| full | 全缓存(缓存区满后写出)，可设置缓存大小 |
| line | 行缓存(遇到newline输出)，可设置缓存大小 |

标准错误是不缓存的(io.stderr)，标准输出是行缓存的。

seek设置和得到文件当前的位置。

f:seek(whence,offset)，whence(起始位置)

|  |  |
| --- | --- |
| set | 文件开始位置 |
| cur | 文件当前位置 |
| end | 文件结束位置 |

whence的默认值是cur，offset的默认值是0，file:seek()返回当前文件位置；file:seek("set")返回文件起始位置；file:seek("end")返回文件结束位置。

获得文件大小：

function fsize (file)

local current = file:seek() -- get current position

local size = file:seek("end") -- get file size

file:seek("set", current) -- restore position

return size

end

local f = io.open(arg[1], "rb")

local s = fsize(f)

print(s)

OS库

操作系统库包含了一些文件操作，系统当前时间和一些其他的便利性，定义再os表中。

由于lua是用标准C写的，它只使用标准C提供的函数posix库提供了POSIX.1的所有标准库；luasocket提供了网络支持；LuaFileSystem提供了目录和文件属性的操作。

对于文件操作，OS提供的库是os.rename(更名操作)和os.remove(移除文件)。

时间和日期

time：无参数调用，返回当前日期和时间，返回一个代表日期和时间的数值

data tables字段：

|  |  |
| --- | --- |
| year | 一全年 |
| month | 01-12 |
| day | 01-31 |
| hour | 00-23 |
| min | 00-59 |
| sec | 00-59 |
| isdst | 布尔值，如果是白天返回true |

前三个字段是必须的，其他字段未提供则是默认未中午(12:00)

date函数是和time相反和函数，将代表日期和时间的函数转换未高级表示。第一个参数是格式化字符串，描述我们想代表的东西，第二个是数字时间

\*t 产生date表。

其他格式串

示例：

print(os.date("a %A in %B")) --> a Thursday in April

print(os.date("%x", 906000490)) --> 04/12/18

%a abbreviated weekday name (e.g., Wed)

%A full weekday name (e.g., Wednesday)

%b abbreviated month name (e.g., Sep)

%B full month name (e.g., September)

%c date and time (e.g., 09/16/98 23:48:10)

%d day of the month (16) [01–31]

%H hour, using a 24-hour clock (23) [00–23]

%I hour, using a 12-hour clock (11) [01–12]

%j day of the year (259) [001–366]

%M minute (48) [00–59]

%m month (09) [01–12]

%p either “am” or “pm” (pm)

%S second (10) [00–60]

%w weekday (3) [0–6 = Sunday–Saturday]

%x date (e.g., 09/16/98)

%X time (e.g., 23:48:10)

%y two-digit year (98) [00–99]

%Y full year (1998)

%% the character ‘%’

%x, %X, and %c根据本地系统时间显示

os.clock返回CPU处理程序的秒数

其它系统调用

os.exit()终结程序执行，第一个可选参数是返回程序的状态(0 or true 表示执行无误)

os.getenv获得环境变量的值，接收变量名，返回字符串和它的值

print(os.getenv("HOME"))

os.execute执行系统命令，第一个返回值返回true表示程序执行没问题，第二个人返回值是exit，第三个值是返回程序执行状态。

os.setlocale，设置lua当前本地的环境，locale name和category(区分本地特性)

六种category

|  |  |
| --- | --- |
| collate | 控制字符串顺序 |
| ctyp | 字符类型 |
| monetary | 对lua程序没有影响 |
| numeric | 控制数字格式化 |
| time | 控制时间和日期的显示 |
| all | 控制以上所有函数 |

默认为all，setlocal 返回locale name或者nil

调试库

调试库并没有为Lua提供调试器，当时调试库提供了你写调试器的原始基本单元。出于性能考虑，官方对于原始基本单元的实现是通过C API，Lua中的调试库在lua内部代码直接存取C API

必须小心的使用调试库，库中的一些函数并不是在性能上很出名，其次，调试库破坏了语言的一些特性，例如不能从外部存取局部变量。还有在最终产品版本发布的时候，你可能想擦除或者关闭调试库中的函数。

调试库中分为两种函数：introspective functions 和hooks

Hooks：跟踪程序执行

introspective functions：

检测运行中的程序的各个方面(函数栈、程序执行当前行，局部变量名和值)

调试库中一个重要的概念是stack level，是引用当前激活的函数的数字：函数调用调试库有1级别，被调用的函数有2级别。

main introspective function is debug.getinfo()

para1: 函数名或者栈的级别

调用debug.getinfo(foo)得到foo的表(表中有数据)，表的一些字段：

source:函数定义的地方，若函数使用字符串定义的，则source是字符串；若函数实在文件中定义的，则source是文件名加上前缀@

short\_src:是source的简短版本

linedefined:函数第一次定义的行数

lastlinedefined:函数最后一次定义的行数

what:函数的类别，Lua --> 常规函数、C --> C函数、

main --> 主要的部分是lua块

name:规范的函数名称

namewhat:前一个字段的意思，可以是"global"、"local"、"filed"、"method" 或者 " "

nups:函数的upvalue的个数

activelines:表示函数的激活行的集合的表，active line是有代码的行，和空行和包含注释的行相反。

func:函数本身

如果是C函数的话，表中只有what, name, and namewhat这些字段

debug.getinfo(n)：获得对应的栈级别的激活函数(n=0,获得getinfo函数本身)，如果n大于激活函数的栈级别，getinfo返回nil

使用debug.getinfo(n)查询激活函数的时候，返回的结果表会有额外的字段：

currentline：函数运行时候的行

函数在Lua中是第一类值，一个函数或者没有值，或者有多个值，Lua通过查找调用函数的代码来寻找函数名，这种方法只在调用getinfo(n)的时候有效

getinfo的可选参数，如此，getinfo就不会花费额外的时间来收集我们不需要的数据。第二个参数是一个字符串，每一个字符串选择一组字段。

|  |  |
| --- | --- |
| 'n' | name,namewhat |
| 'f' | func |
| 'S' | source,short\_src,what,linedefined,lastlinedefined |
| 'l' | currentline |
| 'L' | activelines |
| 'u' | nup |

输出激活的原始栈：

function traceback ()

for level = 1, math.huge do

local info = debug.getinfo(level, "Sl")

if not info then break end

if info.what == "C" then -- is a C function?

print(level, "C function")

else -- a Lua function

print(string.format("[%s]:%d", info.short\_src,

info.currentline))

end

end

end

debug.traceback()

存取局部变量

debug.getlocal存取激活函数的局部变量

para1:查询函数的栈级别

para2:索引变量

返回当前变量名和值，如果变量的索引大于活动变量的数量，getlocal返回nil；如果栈的级别不合法，getlocal引起错误。

Lua以函数中变量出现的顺序计数(从传进来的参数开始计数)，计数依赖函数当前活动变量范围。

示例：

function foo (a, b)

local x

do local c = a - b end

local a = 1

while true do

local name, value = debug.getlocal(1, a)

if not name then break end

print(name, value)

a = a + 1

end

end

调用foo(10,20)结果输出：

a 10

b 20

x nil

a 4

局部变量是在代码初始化后可见的变量。负数检索得到函数额外参数的信息：-1得到第一个额外参数的，这种情况是可变参数

debug.setlocal更改局部变量的值，debug.setlocal(stacklevel,variableindex,value)

返回变量名，若索引超过范围，则返回nil

\*debug.getinfo检索栈的级别

存取非局部变量

getupvalue():存取非局部变量，非局部变量和函数相关，不管函数是否活动(运行中)。

getupvalue(function,variableindex) function-->closure

Lua计数非局部变量是按照第一相关的函数中出现的顺序，一个函数不能以相同的name来存取两个非局部变量。

debug.setupvalue：更新非局部变量的值，返回变量的名字

示例：

function getvarvalue (name, level)

local value

local found = false

level = (level or 1) + 1

-- try local variables

for i = 1, math.huge do

local n, v = debug.getlocal(level, i)

if not n then break end

if n == name then

value = v

found = true

end

end

if found then return value end

-- try non-local variables

local func = debug.getinfo(level, "f").func

for i = 1, math.huge do

local n, v = debug.getupvalue(func, i)

if not n then break end

if n == name then return v end

end

-- not found; get value from the environment

local env = getvarvalue("\_ENV", level)

return env[name]

end

存取其它的协程

debug库的introspective functions接收可选的协程作为第一参数，因而我们可以从外部观察协程。

示例：

co = coroutine.create(function ()

local x = 10

coroutine.yield()

error("some error")

end)

coroutine.resume(co)

print(debug.traceback(co))

print(coroutine.resume(co)) --> false temp:4: some error

-->

stack traceback:

[C]: in function 'yield'

temp:3: in function <temp:1>

回溯并没有经过调用，因为协程和主要的代码块运行在不通发栈上面。

协程引发错误的时候并不展开栈，这移位着我们可以在发生错误之后来观察协程

Hooks

Hook机制允许我们在程序运行的时候将函数注册到特定的事件中以备调用。

四种类型的事件引发hook机制：

call：每当Lua调用函数

return：每当函数返回

line：当Lua开始执行新一行的代码的时候

count：执行完给定数字的指令后发生

Lua用单个参数来调用：

para1：“call” (or “tail call”), “return”, “line”, or “count”.

debug.getinfo获取hook内部更多的信息。

注册hook，调用debug.sethook(hookfunction,maskstring(--描述监听的事件),number(描述想要计数的频率))

关闭hooks，debug.sethook()，即无参数调用。

输出解释器每一行的执行示例：

debug.sethook(print, "l")

\* l-line、c-call、r-return

将print注册到hook函数，且仅在行事件发生的时候执行调用(有点回调的意思)

与hook一起使用的debug.debug，输出执行任意Lua命令的提示：

function debug1 ()

while true do

io.write("debug> ")

local line = io.read()

if line == "cont" then break end

assert(load(line))()

end

end

配置文件(Profile)

local Counters = {}

local Names = {}

local function hook ()

local f = debug.getinfo(2, "f").func

local count = Counters[f]

if count == nil then -- first time 'f' is called?

Counters[f] = 1

Names[f] = debug.getinfo(2, "Sn")

else -- only increment the counter

Counters[f] = count + 1

end

end

获得函数名

function getname (func)

local n = Names[func]

if n.what == "C" then

return n.name

end

local lc = string.format("[%s]:%d", n.short\_src, n.linedefined)

if n.what ~= "main" and n.namewhat ~= "" then

return string.format("%s (%s)", lc, n.name)

else

return lc

end

end

C API

Lua是一门嵌入式语言，它不是一个单独的包，而是一个连接到其它应用的库，将lua的便利性嵌入到应用中。

Lua解释器是一个小型的应用，使用lua库来实现独立的解释器。这个程序是用户交互的接口，将文件和字符串传到Lua库。

用来扩展程序应用的库的能力使得Lua是可扩展单的语言。同时使用Lua的程序可以将新函数注册到Lua环境中。这个函数可以是用C来实现的，如此可以添加Lua不可以直接实现的便利性。这也使得Lua是可扩展的。

Lua 和 C的两种交互模型

1、C有控制权，Lua作为库；这种交互模式下，C是应用代码

2、Lua有控制权，C作为库；这种交互模式下，C是库代码

应用代码和库代码都是使用相同的API来与Lua交互的(C API)

C API是允许C代码和Lua交互的函数集合，将读写Lua的全局变量组合起来

为了调用Lua函数，运行部分Lua代码，注册C函数以便稍后被Lua代码调用。几乎所有Lua可以做的事可以通过C API的C代码来完成。

C API遵循C的模块操作，与Lua模块操作不同；API强调灵活性和简单性，在Lua引用手册中可以找到更多的细节和具体的函数。在Lua发布版本中也可以找到使用API的例子，Lua独立的解释器(lua.c)中也有应用代码的例子，标准库(lstrlib.clmathlib.c)提供了库代码的例子.

C和lua交互的主要组件是虚拟栈，几乎所有的API调用并操作这个栈中的值。所有的数据更改都是通过这个栈发生的(C to Lua or Lua to C)，也可以通过这个栈来保存中间结果。

这个栈解决了两个Lua和C之间不匹配的问题：

1 Lua的垃圾回收，C是要求显式分配

2 Lua的动态类型编程和C的静态类型编程

===============================================

Linux下安装Lua：

1 下载lua-5.3.4.tar.gz，指定文件夹

2 tar -zxf lua-5.3.4.tar.gz

3 cd lua-5.3.4

4 make linux install

5 make test --查看版本号

Lua 5.3.4 Copyright (C) 1994-2017 Lua.org, PUC-Rio

===============================================

静态库和动态库

1 编译成对象文件(.o)

g++ -c StaticMath.cpp

2 ar工具打包文件

ar -crv libstaticmath.a StaticMath.o

使用静态库：

指定搜索路径(-L)和库名(-l，不需要lib前缀和.a后缀)

动态库：

动态库在程序编译时并不会被连接到目标代码中，而是在程序运行是才被载入

Linux下动态库命名规则：libxxx.so

1 生成目标文件：g++ -fPIC -c DynamicMath.cpp

2 生成动态库：g++ -shared -o libmath.so DynamicMath.o

合并：g++ -fPIC -shared -o libmath.so DynamicMath.cpp

使用动态库：

指定搜索路径(-L)和库名(-l，不需要lib前缀和.so 后缀)

-I 在头文件的搜索路径中加入要引用的目录

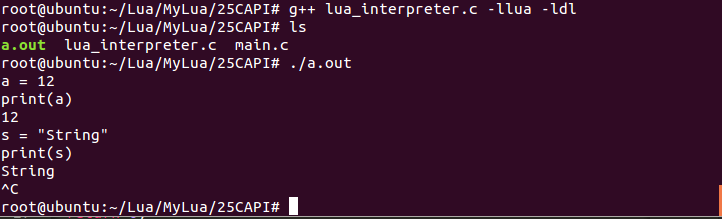
-L 在库的搜素路径中加入要引用的目录

-l 链接库的名称

====================================================

在以上步骤过后，终于编译通过并调用执行

g++ lua\_interpreter.c -llua -ldl



独立的Lua解释器

lua.h 提供了Lua定义的基本函数，包括创建Lua环境的函数和

调用Lua的函数(lua\_pcall)，以及读写Lua环境全局变量的函数，注册新函数以便为Lua调用。lua.h有lua\_ 的前缀。

lauxlib.h定义了辅助库(auxiliary library (auxlib))

提供的函数，auxlib使用lua.h提供的高级抽象层的基本API，特别是对标准库的抽象。基本API是为经济和正交而有的，辅助库是为了共同的库而有的，为你的程序创建其它需要的抽象是简单的，注意的是辅助库不存取Lua的内部，通过官方提供的基本API来完成job的。

Lua库并不定义全局变量，再动态结构lua\_State保持变量的状态。lua中所有内部函数接收一个指向lua\_State结构体的指针作为参数，这种实现使得lua可重入并在多线程中使用。

luaL\_newstate函数创建一个新的lua状态，当luaL\_newstate创建一个新的环境时，环境中并不包含预定义的函数甚至是print。为了使得Lua小巧，所有的标准库提供分离的包，因而你不需要的时候不必使用他们。lualib.h定义了打开库的函数，luaL\_openlibs打开所有的标准库。

创建了一个新的状态并将标准库放在一起后，接下来解释用户的输入，对于用户输入的每一行，程序调用luaL\_loadstring来编译代码，如果没有错误，调用返回0并将产生的函数入栈。接着程序调用lua\_pcall，弹出栈中的函数，并以保护模式运行。和luaL\_loadstring一样，lua\_pcall没有错误的话返回0。

若发生错误，两个函数都会将错误信息入栈，以lua\_tostring得到错误信息，然后以lua\_pop弹出消息并移除。

注意，发生错误的时候，程序仅仅是将错误信息输出到标准错误流。C中的错误处理是非常复杂的，lua并不直接将任何东西写入输出流。

错误处理：

#include <stdarg.h>

#include <stdio.h>

#include <stdlib.h>

void error (lua\_State \*L, const char \*fmt, ...) {

va\_list argp;

va\_start(argp, fmt);

vfprintf(stderr, fmt, argp);

va\_end(argp);

lua\_close(L);

exit(EXIT\_FAILURE);

}

你可以将lua编译成C代码或这C++代码，lua.h并没有包含典型的代码调整，其它多个C库中

#ifdef \_\_cplusplus

extern "C" {

#endif

...

#ifdef \_\_cplusplus

}

#endif

若将lua编译为C代码并用于C++，可以包含lua.hpp而不是lua.h，如下定义：

extern "C" {

#include "lua.h"

}

栈

lua和C交换值的时候遇到的两个问题：

1 错误匹配静态和动态系统

2 错误匹配自动和手动的内存管理

Lua中，如果写a[k]=v，由于元表的原因，a、k和v都可以有多种不同的类型。

不仅把lua设计为C/C++的接口，也为Java、Fortran、C#和其它类似语言的接口；其次Lua的垃圾回收机制，如果把Lua表保存到C变量中Lua engine不知道这个表的使用情况。会错误的假设这个表是垃圾并回收。

因此LuaAPI定义了一个抽象的栈来为C和Lua之间交换值，栈中的每一个slot(插缝、槽)可以保存Lua任意Lua值。无论何时你想从Lua中调取一个值，你调用Lua，Lua将请求的值入栈；无论何时你想传递一个值给Lua，首先入栈，然后调用Lua弹出这个值。我们需要不同的函数将C类型入栈，也需要不同的函数从栈中获取C类型。

几乎所有API的函数都使用栈。例如，LuaL\_loadstring将结果留在栈上(错误信息或者编译后的块)；lua\_pcall获得从栈中被调用的函数，并且也留下特殊的错误信息。

Lua以严格的LIFO原则操作栈(Last In， First Out)。当你调用Lua的 时候，它仅仅改变栈顶的一部分，C代码更加自由，它可以检测栈中的任一元素，即使是在任一位置插入和删除元素。

元素入栈

API有一个入栈函数，对于每一个可以用Lua来表示的C类型，lua\_pushnil 入栈常量NULL，lua\_pushboolean 入栈布尔值，

lua\_pushnumber 入栈双精度，lua\_pushinteger 入栈整数，

lua\_pushunsigned 入栈无符号整数，lua\_pushlstring 入栈任意字符串(字符指针加长度)，lua\_pushstring 入栈0 字符中止的字符串：

void lua\_pushnil (lua\_State \*L);

void lua\_pushboolean (lua\_State \*L, int bool);

void lua\_pushnumber (lua\_State \*L, lua\_Number n);

void lua\_pushinteger (lua\_State \*L, lua\_Integer n);

void lua\_pushunsigned (lua\_State \*L, lua\_Unsigned n);

void lua\_pushlstring (lua\_State \*L, const char \*s, size\_t len);

void lua\_pushstring (lua\_State \*L, const char \*s);

也可以有入栈C函数和入栈用户数据值的函数。

类型lua\_Number是Lua中的数字类型，默认是double，有时候也可以是float或者长整型，以适应Lua严格的机器类型。

类型lua\_Integer是有符号整型，足够存储大字符串的大小；、

类型lua\_Unsigned 是C中的32无符号整型，被用于位库和相关的函数。

Lua中的字符串不是0终结的，可以包含任意的二进制数据。因而需要显式的依赖长度，入栈字符串的基本函数式lua\_pushlstring，需要显示的长度；对于0终结的字符串，使用lua\_pushstring，使用strlen来计算字符串长度(C中的函数式static的)，Lua从不将指针保存到额外字符串。对于Lua保存的任意字符串，Lua使用内部拷贝或者重用。因而你可以自由的更改函数返回的缓存(buffer)。

入栈一个元素的时候，需要确保栈有足够的空间；当Lua开始或者在Lua调用C的时候，确保栈有20个slot，lua.h有定义常量LUA\_MINSTACK，这片空间对大多数使用已经足够了，通常无需考虑。然而有些任务需要更多的栈空间，特别式需要循环入栈元素的时候，这样，你可以调用lua\_checkstack(检测栈是否有足够的空间供你使用)。

int lua\_checkstack (lua\_State \*L, int sz);

查询元素

关联栈中的元素，API使用切片(indices)，第一个入栈的元素下标为1，接下来为2，直到栈满；可以使用栈顶作为存取元素的引用，使用负数索引，-1存取栈顶元素；

lua\_tostring(L,-1)将栈顶元素的值作为字符串返回。检查元素的类型，API 提供了lua\_is\* 函数族，\*可以是任意lua类型。

原型：int lua\_is\* (lua\_State \*L, int index);

如：lua\_isnumber,lua\_isstring, lua\_istable

lua\_isnumber并不检查值的具体类型，而是检查值可以转换的类型

lua\_type，返回栈中元素的类型，每一种类型都在头文件lua.h

中有常量表示。LUA\_TNIL,LUA\_TBOOLEAN,LUA\_TNUMBER,LUA\_TSTRING, LUA\_TTABLE, LUA\_TTHREAD,LUA\_TUSERDATA, 和LUA\_TFUNCTION.

获取栈中元素的值：

int lua\_toboolean (lua\_State \*L, int index);

const char \*lua\_tolstring (lua\_State \*L, int index,

size\_t \*len);

lua\_Number lua\_tonumber (lua\_State \*L, int index);

lua\_Integer lua\_tointeger (lua\_State \*L, int index);

lua\_Unsigned lua\_tounsigned (lua\_State \*L, int idx);

lua\_toboolean将任意Lua值转换到C的布尔值(0 和 1)。

Lua条件判断：nil和false为真，其余为假。

对于给定的元素没有合适的类型也可以调用lua\_to\*函数组。

lua\_toboolean对任意类型都可以调用；

lua\_tolstring对非字符串值返回空(NULL)；

数字函数对于错误类型的符号数返回0；

lua\_Number lua\_tonumberx (lua\_State \*L, int idx, int \*isnum);

lua\_Integer lua\_tointegerx (lua\_State \*L, int idx, int \*isnum);

lua\_Unsigned lua\_tounsignedx (lua\_State \*L, int idx, int \*isnum);

输出参数isnum返回表示Lua值是否是是一个数字的布尔值。

lua\_tolstring函数返回指向内部字符串的拷贝和字符串的长度的指针。不可以更改内部拷贝，Lua确保这个指针是合理的，只要相关字符串的值在栈中。当一个C函数被Lua返回调用时，Lua会清理栈。你无需在获取字符串的函数外面存储指向Lua字符串的指针。

任何lua\_tolstring返回的字符串在末尾都会有一个0字符，返回的大小要通过第三个参数len来表示字符串实际的字符串长度大小。

也可以以NULL来作为第三个参数调用lua\_tolstring，宏lua\_tostring默认使用NULL来调用lua\_tolstring

输出栈中的全部内容：

static void stackDump (lua\_State \*L) {

int i;

int top = lua\_gettop(L); /\* depth of the stack \*/

for (i = 1; i <= top; i++) { /\* repeat for each level \*/

int t = lua\_type(L, i);

switch (t) {

case LUA\_TSTRING: { /\* strings \*/

printf("'%s'", lua\_tostring(L, i));

break;

}

case LUA\_TBOOLEAN: { /\* booleans \*/

printf(lua\_toboolean(L, i) ? "true" : "false");

break;

}

case LUA\_TNUMBER: { /\* numbers \*/

printf("%g", lua\_tonumber(L, i));

break;

}

default: { /\* other values \*/

printf("%s", lua\_typename(L, t));

break;

}

}

printf(" "); /\* put a separator \*/

}

printf("\n"); /\* end the listing \*/

}

栈的其它操作

除了之前的栈操作，API还提供了一下的栈操作函数：

int lua\_gettop (lua\_State \*L);

void lua\_settop (lua\_State \*L, int index);

void lua\_pushvalue (lua\_State \*L, int index);

void lua\_remove (lua\_State \*L, int index);

void lua\_insert (lua\_State \*L, int index);

void lua\_replace (lua\_State \*L, int index);

void lua\_copy (lua\_State \*L, int fromidx, int toidx);

特别的，lua\_settop(L,0)是清空栈，宏定义中的弹出n葛元素，

#define lua\_pop(L,n) lua\_settop(L, -(n) - 1)；

lua\_copy拷贝索引处的值。

错误处理

lua中的所有结构都是动态的：随着需要增长，必要的时候会收缩，着表明内存管理失败的可能性在Lua中式常见的，几乎所有的操作最终都会遇到这种可能性。

此外很多操作都会引起其它的错误，最后，分配内存的操作会触发垃圾回收器(调用终结器，这也会引发错误)。换句话说，API中的大多数函数会造成错误。

Lua API不用错误码来标记错误操作，而是使用异常信号来表示错误。通过调用C中的setjmp来模拟异常处理的机制，Lua中调用longjmp来抛出错误。

应用程序中的错误处理

当程序调用Lua API 函数的时候，会引发错误。需要设置相关的setjmp，若未设置setjmp，解释器不会跳转的。因而Lua会调用panic 函数，函数返回的时候会退出应用。可以使用lua\_atpanic来设置自己的panic函数。

应用代码中处理错误，必须要通过Lua来调用你的代码，以便设置合适的环境来捕捉错误(在setjmp的环境中来运行代码)。同样，可以使用pcall以保护模式来运行Lua代码，使用lua\_pcall来运行C代码。将C代码打包在一个函数中，并通过Lua使用lua\_pcall来调用这个函数。使用这种设置，你的C代码会以保护模式来运行，即使内存分配失败，lua\_pcall会返回合适的错误代码，并以一致的状态离开解释器。

栈操作示例：

int main (void) {

lua\_State \*L = luaL\_newstate();

lua\_pushboolean(L, 1);

lua\_pushnumber(L, 10);

lua\_pushnil(L);

lua\_pushstring(L, "hello");

stackDump(L);

/\* true 10 nil 'hello' \*/

lua\_pushvalue(L, -4); stackDump(L);

/\* true 10 nil 'hello' true \*/

lua\_replace(L, 3); stackDump(L);

/\* true 10 true 'hello' \*/

lua\_settop(L, 6); stackDump(L);

/\* true 10 true 'hello' nil nil \*/

lua\_remove(L, -3); stackDump(L);

/\* true 10 true nil nil \*/

lua\_settop(L, -5); stackDump(L);

/\* true \*/

lua\_close(L);

return 0;

}

库代码中错误处理

Lua 使安全的语言，不论你用Lua写了什么，不论错误使什么，你可以就Lua本身来理解程序的行为，就Lua本身可以检测和显示出来。可以对比C，错误程序的行为在硬件的底部表示出来。

当你为Lua添加新的C函数时，这就打破了Lua的安全性，如poke:在任意内存地址存储任意字节，会导致内存越界。必须确保添加的函数是安全的并提供良好的错误处理。

正如之前所说，C程序必须通过lua\_pcall来设置错误处理。当为Lua写库函数的时候，通常并不需要错误处理。通过Lua中的pcall或者应用代码中的lua\_pcall调用库函数会将捕捉到的错误抛出。因此C库中的函数检测到错误会调用lua\_error(luaL\_error，格式化错误信息并调用lua\_error)。

函数lua\_error清除Lua中需要清除的东西，返回原始执行调用处，传递错误信息。

扩展你的程序

使用Lua重要的一点是lua作为配置性语言，本章使用Lua配置一个程序

Basic

用一个表来配置C程序的窗口初始化程序(配置文件转化为Lua程序)

配置文件：

-- define window size

width = 200

height = 300

使用Lua API 指导Lua解析文件，获得全局变量width和height的值，即使用load函数来做这个事情，这个函数在Lua环境中使用。

luaL\_loadfile从文件fname中加载块，调用lua\_pcall来编译块，若发生错误，则这些函数将错误信息放在栈上面，返回非零错误代码，使用lua\_tostring(L, -1)来检索栈顶的信息。

从配置文件中获得用户信息：

void load (lua\_State \*L, const char \*fname, int \*w, int \*h) {

if (luaL\_loadfile(L, fname) || lua\_pcall(L, 0, 0, 0))

error(L, "cannot run config. file: %s", lua\_tostring(L, -1));

lua\_getglobal(L, "width");

lua\_getglobal(L, "height");

if (!lua\_isnumber(L, -2))

error(L, "'width' should be a number\n");

if (!lua\_isnumber(L, -1))

error(L, "'height' should be a number\n");

\*w = lua\_tointeger(L, -2);

\*h = lua\_tointeger(L, -1);

}

lua\_getglobal在Lua环境中获得全局变量，每次调用都会将相应的全局变量入栈。

Lua配置文件的优点：

1 处理语法细节

2 完成复杂的配置

3 使得程序更加灵活

表操作

使用表来配置颜色

使用BLUE = {r=0, g=0, b=1.0}

<other color definitions>

而不是：

background = BLUEbackground = {r=0.30, g=0.10, b=0}

来配置颜色

lua\_gettable：接受栈中表的位置，弹出栈中的键，将相关联的值入栈。Lua5.1提供了lua\_getfield

调用Lua函数

Lua的一个长出是可以定义函数以供应用程序来调用。

API拟定调用函数是简单的：

1 入栈调用函数

2 将调用的参数入栈，lua\_pcall来实际调用

3 最后获得结果

lua\_pcall会调整实际的结果数量，如果需要的话会入栈空值，或将额外的值忽略。入栈结果的时候，lua\_pcall会将栈中的函数和参数移除。

通用LUA函数调用(包装器)

使用C中的可变参数vararg创建一个调用lua的包装器，包装函数call\_va，接受被调用的函数名，描述参数类型和结果的字符串，参数列表和接受结果的指针。有了这个函数就可以将之前的函数调用写成：call\_va(L, "f", "dd>d", x, y, &z);

call\_va(L, functionname, argtype, arglist, result)

"dd>d" --> 两个参数类型是 double，结果类型是double。

> ：分隔结果参数，若无返回结果，则 > 可选。

|  |  |
| --- | --- |
| i | integer |
| d | double |
| s | strin |

以下是call\_va的实现，遵循以下步骤：入栈函数，入栈参数，调用，获取结果。

三个不同点：

1无需检测func是否是函数类型，lua\_pcall会引发错误。

2因为要入栈任意数量的参数，必须要检查栈的空间

3函数返回的是字符串，call\_va不能从栈中返回结果，取决于调用器来弹出。

Lua 调用C

Lua并不能调用任意的C函数，C调用Lua函数的时候，要遵循协议来传递参数和获取结果，Lua调用C函数也是如此。Lua调用C函数必须要注册函数，以适当的方式将地址传递给Lua。调用的时候，使用C调用Lua相同的栈，C函数从栈中获取参数并将结果入栈。

栈并不是全局结构，每个函数都有它自己私有的局部栈。Lua调用C函数的时候，第一个参数在局部栈的第一个位置。

注册C函数到Lua中：lua\_CFunction

从C的角度来看，C函数从Lua环境中获得一个参数，并返回一个整数值到栈中，因此，函数在入栈结果之前不需要清除栈。

返回后，Lua自动保存结果并清除整个栈。

在Lua中使用C函数之前，必须要注册C函数，使用lua\_pushcfunction来注册C函数，它获得一个指向C函数的指针，并创建函数类型的值，以便在Lua中表示这个函数，一旦注册，C函数就和Lua中的其它函数表现得一样了。

快速测试C函数，在以打开的Lua环境中加入一下两行(luaL\_openlibs):

lua\_pushcfunction(L, l\_sin);--入栈函数类型

lua\_setglobal(L, "mysin");--赋值到全局函数

之后你就可以在Lua脚本中使用mysin函数了。

附录

通过lua\_pcall和lua\_call，Lua调用的C函数可以回调Lua。标准库里面的许多函数都是那样做的：table.sort可以调用排序函数；string.gsub可以调用替换函数；

pcall 和 xpcall以保护模式来调用函数。

若我们知道主要的代码是从C中调用的，将会得到一个像C调用Lua脚本的序列(C库调用Lua)。

通常Lua处理这些调用序列是没问题的，毕竟与C集成是Lua的语言特点，这种集成引起了协程的问题。

Lua的每个协程都有自己的栈，保存挂起的协程信息。特别的栈保存地址、参数和每次调用的局部变量。对于调用Lua函数，解释器用合适的数据结构来实现栈，叫软栈；对于调用C函数，解释器必须使用C栈，毕竟C函数的局部变量和返回地址存活于C栈中。

对于解释器来说，有多个软栈(softstack)是容易的，可是ANSIC在运行期间只有一个内部栈，因此Lua的协程不能挂起执行的C函数。如果调用路径的C函数从唤醒到相关(各自)的输出，Lua不能保存C函数的状态还原到下一次的唤醒。

co = coroutine.wrap(function (a)

return pcall(function (x)

coroutine.yield(x[1])

return x[3]

end, a)

end)

print(co({10, 3, -8, 15}))

--> false attempt to yield across metamethod/C-call boundary

---------------------读取目录--------------------------

#include <dirent.h>

#include <errno.h>

#include <string.h>

#include "lua.h"

#include "lauxlib.h"

static int l\_dir (lua\_State \*L) {

DIR \*dir;

struct dirent \*entry;

int i;

const char \*path = luaL\_checkstring(L, 1);

/\* open directory \*/

dir = opendir(path);

if (dir == NULL) { /\* error opening the directory? \*/

lua\_pushnil(L); /\* return nil... \*/

lua\_pushstring(L, strerror(errno)); /\* and error message \*/

return 2; /\* number of results \*/

}

/\* create result table \*/

lua\_newtable(L);

i = 1;

while ((entry = readdir(dir)) != NULL) {

lua\_pushnumber(L, i++); /\* push key \*/

lua\_pushstring(L, entry->d\_name); /\* push value \*/

lua\_settable(L, -3);

}

closedir(dir);

return 1; /\* table is already on top \*/

}

pcall是一个C函数，Lua不能将其挂起。因为ANSI C没法将C函数挂起并稍后唤醒。Lua5.2使用continuations改善了这种情况。

Lua5.2使用long jump实现yields，以同样的方式实现了errors，long jump 抛除任何的C栈中的C函数的信息，因而可能唤醒C函数。然而C函数foo可以具体化为continuation 函数foo-c，当foo再次唤醒的时候，解释器会去调用foo-c这个C函数，而不是foo这个函数。

C模块

Lua模块是一个定义了Lua函数并保存在合适位置的chunk，一般作为表的入口，为Lua使用的C模块模仿了这种行为，除了C函数的定义，还定义了一个特殊的函数，扮演Lua库中的主块的角色。这个函数将所有的C函数注册到模块中并保存到一个合适的地方，在次使用表作为入口。和Lua主块一样，它应该初始化模块中需要初始化的的东西。

Lua通过这个注册过程接受C函数。一旦C函数被表示和存储到Lua中，Lua可以直接通过地址来调用。换句话说，C函数一旦注册后，Lua不需要函数名、包位置或可见性规则来调用函数。一个C模块有一个单一公共的函数(打开库的函数)，所有其它的函数是私有的，C中以static申明。

当你以C函数来扩展Lua时，将你的C代码设计为一个C模块，即使你指向注册一个C函数。迟早你会需要其它函数(复用性原则)。辅助库提供了一个有用的函数来做这件事。宏luaL\_newlib接受C函数名列表，并将他们注册到一个新表中。

1 模块中定义库函数

2 申明一个luaL\_Reg类型的数组

3 申明一个主函数，并用luaL\_newlib打开库函数

调用luaL\_newlib创建了一个新表，并以name-function来填充，返回时，luaL\_newlib在栈中留下了一个打开库的新表

4 链接库到解释器，最便捷的方式是动态链接，创建动态库，然后将动态库放到C路径中，之后在Lua中直接使用require加载库

local mylib = require "mylib"

动态连接器必须知道luaopen\_mylib函数的名字以便找到，动态连接器会找到luaopen\_开头的模块名

如果不支持动态链接的话需要将Lua重编译到库中。一个简单的方法是将库luaopen\_mylib添加到标准库中以便luaL\_openlibs打开，luaL\_openlibs在linit.c文件中。

C API

<https://blog.csdn.net/u014086857/article/details/51944163>

书写C函数的技巧(数组、字符串、C中存储Lua值)

数组操作

数组，是Lua中表的特殊使用的一个名字，可以使用操作表的函数来操作数组。lua\_settable和lua\_gettable，为了便捷和性能考虑，API 提供了两个函数来操作数组：

void lua\_rawgeti (lua\_State \*L, int index, int key);

void lua\_rawseti (lua\_State \*L, int index, int key);

index涉及栈中的表，key涉及表中的元素；表用作数组，却很少用作元方法。

luaL\_checktype, luaL\_len, and lua\_pcall

luaL\_checktype函数(lauxlib.h)确保给定的参数是指定的类型

lua\_len等同于#操作符，由于是元方法，操作符会再任意对象上产生结果，而不仅仅是数字l，因此ua\_len返回栈中的结果。

luaL\_len(辅助库中的函数)如果长度不是数字的话会引发错误，返回C中的integer类型的长度。

lua\_call函数不以保护模式运行，和lua\_pcall类似，但是不传播错误，而是返回错误代码。写函数的时候，用lua\_call是一个好的想法。

字符串操作

C函数接受Lua字符串参数，需要注意两点：存取字符串的时候，不从栈中弹出；从不去修改字符串。当C函数需要返回字符串给Lua时事情变得迫切起来了。由C代码来关注缓存的分配和销毁以及溢出等，Lua API也提供了一些函数来处理这些任务。

标准的API提供了两个字符串的操作函数：字符串提取和连接，lua\_pushlstring获得字符串的长度。Lua提供了lua\_concat来连接字符串，和 Lua中..连接操作符一样，必要的时候将数字转换为

字符串，它可以一次性连接多个字符串，lua\_concat(L,n)会连接并弹出栈顶的n个值，并将结果入栈到栈顶。

const char \*lua\_pushfstring (lua\_State \*L, const char \*fmt, ...);

和C函数的sprintf类似，根据额外的参数和格式化字符串产生字符串，和sprintf不同的是，无需使用buffer，Lua为你动态的创建字符串，函数将结果字符串入栈并返回一个指向其的指针，不必担心缓存溢出。

lua\_concat 和 lua\_pushfstring在连接字符串的时候都很

有用。

%s inserts a zero-terminated string

%d inserts an integer

%f inserts a Lua number, that is, a double

%p inserts a pointer

%c inserts an integer as a character

%% inserts the character ‘%’

然而要连接任意字符串的时候，一个一个的连接很没有效率，相反，我们使用辅助库的缓存的便利性来连接字符串。缓存与两个函数一起作用，一个是给出缓存任意大小以便你存取字符串，一个是将缓存的内容转换为Lua字符串。

函数luaL\_pushresultsize不需要Lua状态作为它的第一个参数，初始化后，缓存保持对状态的引用，所以当调用其它操作缓存的函数的时候，并不需要将Lua环境状态作为参数传递。我们也可以使用辅助库里的缓存而不需直到最终结果的大小。

辅助库里提供了多个函数将东西添加到缓存中：luaL\_addvalue将栈顶的Lua字符串添加到缓存中；luaL\_addlstring以显式长度添加字符串到缓存中；

luaL\_addstring添加0终结的字符串到缓存中；

luaL\_addchar添加大哥字符到缓存中；

void luaL\_buffinit (lua\_State \*L, luaL\_Buffer \*B);

void luaL\_addvalue (luaL\_Buffer \*B);

void luaL\_addlstring (luaL\_Buffer \*B, const char \*s, size\_t l);

void luaL\_addstring (luaL\_Buffer \*B, const char \*s);

void luaL\_addchar (luaL\_Buffer \*B, char c);

void luaL\_pushresult (luaL\_Buffer \*B);

当使用辅助库的时候，需要担心一个细节，一旦初始化缓存后，在Lua的栈中会保持一些中间结果。因而你不能假设栈顶中仍然是你之前的使用缓存的结果。然而，尽管在使用栈的时候，你可以在使用缓存的同事使用栈来做其它任务。每当你存取缓存的时候，需要保持pop/push平衡。luaL\_addvalue会打破这个规则，它假定字符串被添加到缓存中的时候就被入到栈顶中。

C函数中存储状态

C函数需要保存一些非局部变量(调用之外的数据)，C中使用全局的或者静态的变量来满足这种需求。在Lua中，其一不能将Lua通用值存储到C中；其二使用这种变量的库不能与Lua状态协作。

一个Lua函数有两个基本的地方来存储非局部数据：全局变量和非局部变量。CAPI也提供了两个基本的地方来存储非局部数据的地方：registry和upvalue。

registry是一张仅可以被C代码存取的全局表，使用registry来存储多个模块共享的数据；如果想将数据存储到私有模块中或者单一函数中，则使用upvalues。

registry位于伪索引，其值是LUA\_REGISTRYINDEX定义的。伪索引是像栈中的索引一样，然而伪索引关联的值却不在栈中。

获得registry中的键的字段：

lua\_getfield(L, LUA\_REGISTRYINDEX, "Key");

registry是一个常规的Lua表，因而你可以使用任意的Lua值来索引除了nil。然而所有的C模块代码都共享相同的registry，必须小心的选择键的值。当你想要用独立的库来存取数据的时候，字符串做为键值是非常有用的。因为所有的C代码都需要直到键的值。

在registry中避免使用数字作为键，因为这种键在引用系统中被保留了。函数luaL\_ref(L, LUA\_REGISTRYINDEX);创建了一个新的引用。前面的调用弹出栈中的值，以一个新的整数作为键值来引用这个值并存到registry中，返回这个键，称这个键为引用。

使用引用主要是将需要引用到Lua的值存储到C结构中。

不应将Lua字符串的指针存储到C函数之外并使用Lua字符串，更多的是Lua没有提供指向其它对象的指针：如表或函数。因此不能通过指针来引用Lua对象，相反当我们需要这种指针的时候，创建一个引用并存储到C中。

为了入栈一个值，并将其关联到栈中的一个引用r：

lua\_rawgeti(L, LUA\_REGISTRYINDEX, r);

释放值和引用：

luaL\_unref(L, LUA\_REGISTRYINDEX, r);

以上调用后，luaL\_ref会再次返回r这个引用，引用系统将nil视为一种特殊的情况，当调用luaL\_ref得到一个nil时，并不会创建一个新的引用，而是返回一个常量引用LUA\_REFNIL。

luaL\_unref(L, LUA\_REGISTRYINDEX, LUA\_REFNIL);

下一次入栈一个空值：

lua\_rawgeti(L, LUA\_REGISTRYINDEX, LUA\_REFNIL);

引用系统也定义了常量：LUA\_NOREF，是一个不同于合理引用常量的整数，对标记非法引用也是有用的。

另外一种方法是创建一个key到registry中：使用静态变量的地址作为key，C链接编辑器确保在所有的库中是唯一的。为了使用这个选项，需要使用函数lua\_pushlightuserdata，入栈Lua栈一个值代表C指针。

一下方法是从registry中存储与索引一个字符串：

/\* variable with a unique address \*/

static char Key = 'k';

/\* store a string \*/

lua\_pushlightuserdata(L, (void \*)&Key); /\* push address \*/

lua\_pushstring(L, myStr); /\* push value \*/

lua\_settable(L, LUA\_REGISTRYINDEX); /\* registry[&Key] = myStr \*/

/\* retrieve a string \*/

lua\_pushlightuserdata(L, (void \*)&Key); /\* push address \*/

lua\_gettable(L, LUA\_REGISTRYINDEX); /\* retrieve value 轻量级用户数据\*/

myStr = lua\_tostring(L, -1); /\* convert to string \*/

为了使变量地址简化，Lua5.2 提供了两个函数：lua\_rawgetp和lua\_rawsetp，使用指针作为key。重写上面的例子：

static char Key = 'k';

/\* store a string \*/

lua\_pushstring(L, myStr);

lua\_rawsetp(L, LUA\_REGISTRYINDEX, (void \*)&Key);

/\* retrieve a string \*/

lua\_rawgetp(L, LUA\_REGISTRYINDEX, (void \*)&Key);

myStr = lua\_tostring(L, -1);

Upvalues

尽管registry提供了全局变量，upvalue(非局部变量)机制实现了一种等价于C的静态变量(只在特殊函数里面可见)。

每当在Lua中创建一个新的函数，可以使用任意数量的upvalue来关联，每个upvalue可以保留一个Lua值。当函数调用的时候，使用pseudo-indices(伪索引)来存取任意的upvalue。

把C函数和一个upvalue的关联性称之为闭包，一个C的闭包近似于一个Lua闭包。你可以使用相同的函数代码来创建不同的闭包，可upvalue却不相同。

例子：

static int counter (lua\_State \*L); /\* forward declaration \*/

int newCounter (lua\_State \*L) {

lua\_pushinteger(L, 0);

lua\_pushcclosure(L, &counter, 1);

return 1;

}

lua\_pushcclosure创建一个新的闭包，counter使一个基底函数，1是upvalue的数量。在创建一个新的闭包之前，必须为栈中的upvalue初始化值。

counter 的定义：

static int counter (lua\_State \*L) {

int val = lua\_tointeger(L, lua\_upvalueindex(1));

lua\_pushinteger(L, ++val); /\* new value \*/

lua\_pushvalue(L, -1); /\* duplicate it \*/

lua\_replace(L, lua\_upvalueindex(1)); /\* update upvalue \*/

return 1; /\* return new value \*/

}

这里的key元素是宏lua\_upvalueindex，产生一个upvalue的伪索引；特别的，表达式lua\_upvalueindex(1)产生运行函数的的第一个upvalue的伪索引。这个伪索引不同于其它的栈索引，不在栈中存活。因而调用lua\_tointeger将当前值的第一个upvalue作为数字返回。接着，函数counter入栈一个新值++val，并拷贝一份，用其中的拷贝来替换upvalue的值；最后返回其它的拷贝来作为counter的返回值。

元组是一种以匿名字段的常量记录(也就是说，值关联的字段是匿名的)，你可以以数字索引来检索字段。或者一次性检索所有字段。使用在upvalue中存储值的函数来表示元组；当使用数字作为索引的时候，返回特定字段；也可以无参调用，返回所有的字段：

x = tuple.new(10, "hi", {}, 3)

print(x(1)) --> 10

print(x(2)) --> hi

print(x()) --> 10 hi table: 0x8087878 3

检索一个不存在的upvalue时，结果会返回一个LUA\_TNONE类型的伪值，当存取一个栈顶的索引的时候，也会返回一个LUA\_TNONE类型的伪值。使用lua\_isnone来测试给定的值是否是upvalue。然而，不应以负数索引来调用lua\_upvalueindex，所以要检测这个索引值。函数luaL\_argcheck检测任意给定的条件，必要的时候会引起错误。

共享的upvalue(非局部变量)

有时需要为库中的所有函数共享一些变量或者值，尽管可以使用registry来完成这项任务，也可以用upvalue来完成这项任务。不同于Lua的闭包，C闭包不能共享upvalue，每个闭包都有独立的upvalues，然而可以为不同函数的upvalue值引用到一张共同的表，所以这张表变成函数共享数据的公共环境。

函数luaL\_newlib打开C库，宏luaL\_newlibtable为库创建了一张新的表。

函数luaL\_setfuncs将列表l中的函数添加到新表中，它的第三个参数告知库中的新函数的upvalue的数量，upvalue的初始值应该在栈中。

为库中的函数创建一个公用的表作为单一upvalue值：

/\* create library table ('lib' is its list of functions) \*/

luaL\_newlibtable(L, lib);

/\* create shared upvalue \*/

lua\_newtable(L);

/\* add functions in list 'lib' to the new library, sharing

previous table as upvalue \*/

luaL\_setfuncs(L, lib, 1);

最后一次调用从栈中移除了共享的表，仅留下新库。

C中的用户定义类型

使用C中的新类型来扩展Lua，使用元方法或者其它的方法。

用户数据

用户数据提供一片原始的内存区域，而没有Lua预定义的操作，这片空间可以存取任意的东西。

lua\_newuserdata分配给定大小的内存块，将相关的用户数据入栈，然后返回块的地址。

void \*lua\_newuserdata (lua\_State \*L, size\_t size);

辅助库里的luaL\_newlib创建一张表，并以数组提供的name-function来填充。

将一个数据类型从另一个用户数据类型区别出来是为那种类型创建唯一的一个元表。每当创建一个用户数据类型，就用一个相关的元表关联，Lua不能改变用户数据的元表。

也需要将元表存储到一个新的地方，以便存取创建新的用户数据并检测给定的用户数据的类型是否正确。

两种方式来存取元表：在registry和作为库中的函数的upvalue，在Lua中注册一种新的C类型到registry是常规的，使用类型名作为索引，元表作为值。

为避免冲突，需小心的选择类型名称。

int luaL\_newmetatable (lua\_State \*L, const char \*tname);

void luaL\_getmetatable (lua\_State \*L, const char \*tname);

void \*luaL\_checkudata (lua\_State \*L, int index,

const char \*tname);

luaL\_newmetatable创建一个新表，当作元表使用，将新表置于栈顶，将表关联到registry中给定的名字。

luaL\_getmetatable从registry中指定的名字检索元表。

luaL\_checkudata检查给定栈位置的对象的元表数据是否匹配registry中给定名字的数据。若指定位置的对象没有正确的元表或者不是用户数据，则引发错误，否则返回用户数据地址。

面向对象的存取

将新类型转换为对象，以便使用面向对象的语法来操作实例，obj:attr()和obj.attr()等价。

\_\_index元方法，对于表来说，Lua找不到给定键的值的时候，Lua会调用\_\_index元方法；对于用户数据，每次存取的时候都会调用\_\_index元方法，因为用户数据没有键(key)。

local metaarray = getmetatable(array.new(1))

metaarray.\_\_index = metaarray

metaarray.set = array.set

metaarray.get = array.get

metaarray.size = array.size

不能从Lua中设置用户数据的元表，但可以获得。

数组存取

对于面向对象符号存取数组的一种可选的方法是使用常规符号数组符号来存取数组。可以写a[i]来代替a:get(i)。

函数setarray和getarray接收参数，以便和相关元方法关联。

local metaarray = getmetatable(array.new(1))

metaarray.\_\_index = array.get

metaarray.\_\_newindex = array.set

metaarray.\_\_len = array.size

在C代码中注册元方法

static const struct luaL\_Reg arraylib\_f [] = {

{"new", newarray},

{NULL, NULL}

};

static const struct luaL\_Reg arraylib\_m [] = {

{"\_\_newindex", setarray},

{"\_\_index", getarray},

{"\_\_len", getsize},

{"\_\_tostring", array2string},

{NULL, NULL}

};

int luaopen\_array (lua\_State \*L) {

luaL\_newmetatable(L, "LuaBook.array");

luaL\_setfuncs(L, arraylib\_m, 0);

luaL\_newlib(L, arraylib\_f);

return 1;

}

轻量级用户数据

full userdata(满用户数据)，还有轻量级用户数据，轻量级用户数据是一个表示C指针的值(void\* 的值)，一个轻量级用户数据是一个值，不是一个对象，并不创建轻量级用户数据，lua\_pushlightuserdata将轻量级用户数据入栈。

void lua\_pushlightuserdata (lua\_State \*L, void \*p);

轻量级数据和满用户数据是完全不通的东西，轻量级数据不是缓存，而是一个单一的指针，没有元数据，也不被垃圾回收器管理回收。

有时候使用轻量级数据作为一种可选的方案来替代满用户数据。一、轻量级用户数据并没有元表，故而无法知道他们的类型

二、除了名字，满用户数据花销并不昂贵，和malloc相比会有点花销。

轻量级用户数据的真正使用源于相等性，满用户数据是一个对象，它之和自己相等；轻量级用户数据表示一个C指针的值，因此和任意表示相同指针值的用户数据相等。因而我们可以使用轻量级用户数据来找到Lua中的C对象。轻量级用户数据的使用，作为registry中的键来使用，轻量级用户数据的基本要素是相等性。每当将相同地址值入栈(lua\_pushlightuserdata)，会得到相同的Lua值，因而在registry中的相同的入口。

另一种设想是给定C的地址值来检索满用户数据

管理资源

生活不易，有时候对象不仅需要内存，还需要其它的资源，如文件描述符、窗口句柄，这种情况下，对象变成垃圾会被回收，而一些其它资源也必须被释放，Lua提供了元方法\_\_gc终结器。

线程和状态

Lua并不支持真正意义上的多线程，具有控制权的线程共享内存。缺少这种支持的原因有两个：第一个原因是ANSI C 没有提供，所以在Lua中没有可移植的方式来实现这种机制；第二个强有力的原因是我们并不认为多线程对Lua来说是一个好主意。

多线程是为了低级编程开发的，同步机制像信号量或者监听器置于操作系统而不是在应用程序中，涉及到多线程的bugs很难被找出或者发现，有些bugs会导致安全性；在部分严格的程序中，涉及到同步的地方会导致表现性能差，例如内存分配器。

线程的优先级结合和共享内存，多线程会引起许多问题。我们可以使用非优先级线程或者非共享内存来避免这些问题。

Lua提供协程，避免了线程转换的时候引起的不可预测的问题。Lua的状态无需内存共享，因而形成Lua并发性很好的基础。

多线程

从C API得角度来看，将线程视作栈是很有用的，而且那也是线程的实际所是。从实现的点来看，栈保持挂起线程的调用和每次调用的参数和局部变量。换句话说，栈有一个线程持续运行的所有信息。

当我们从Lua-C API调用大多数函数的时候，哪个函数是在操作一个具体的栈；

如：lua\_pushnumb必须入栈一个数字到具体的栈中；lua\_pcall需要调用栈。Lua怎么知道哪个栈是使用的呢？如果将一个数字入到不通的栈该怎么办？诀窍在于lua\_State类型，是这些函数的第一个参数，不仅表示Lua状态，也表示再状态中的线程。

无论何时你创建了一个Lua State，Lua自动再这个State中创建一个新的线程，并且返回lua\_State来表示这个线程，主线程从不回收，当使用lua\_close关闭state时，线程和state一起释放。

创建线程：

lua\_State \*lua\_newthread (lua\_State \*L);

返回的lua\_State类型的指针表示一个新的线程，将新的线程入栈到栈顶。作为"线程"的值。

如：

L1 = lua\_newthread(L);

两个线程L1和L引用到相同的Lua state。每个线程都有自己的栈。新线程L1以空栈开始，原线程L在它的栈顶上有新的线程。

printf("%d\n", lua\_gettop(L1)); --> 0

printf("%s\n", luaL\_typename(L, -1)); --> thread

除了主线程，线程是垃圾回收装置的对象，和其它Lua对象类似。当你创建一个新的线程时，线程的值会入栈到栈中，以确保线程不是垃圾。不要在不固定的状态中使用线程。

调用lua\_pushstring会引发垃圾回收器并回收L1，尽管L1在使用着。为避免这种情况，保持对你使用的线程的应用，如在registry中或者栈中固定的线程。

一旦有了新的线程，可以像使用主线程一样使用它。可以入栈或者弹出栈中的元素，可以使用线程来调用函数。

lua\_getglobal(L1, "f"); /\* assume a global function 'f' \*/

lua\_pushinteger(L1, 5);

lua\_call(L1, 1, 1);

lua\_xmove(L1, L, 1);

函数lua\_xmove移动在相同state下的两个栈之间的值。

如lua\_xmove(F,T,n)，弹出栈F上的n个元素，并入栈T上。

主要的点是使用多线程来实现协程，以便我们可以在执行的时候挂起，之后再次唤醒，为此需要lua\_resume函数。

和使用lua\_pcall一样使用lua\_resume来唤醒协程：入栈将被调用的函数，入栈元素，并且调用lua\_resume传进narg(参数的个数)，线程调用参数。这种行为更像lua\_pcall，区别点有三。

其一：lua\_resume从调用函数中返回所有结果，没有记录想要结果的数量的参数。

其二：没有消息句柄的参数，错误发生后不会释放栈，所以可以在发生错误后检测栈。

其三：如果运行函数输出，lua\_resume返回一个特殊码LUA\_YIELD，并将线程留在state中，以便稍后唤醒。

当lua\_resume返回LUA\_YIELD，线程栈中可见的部分仅仅包含传递给yield的值。调用lua\_gettop返回产生的值的数量，使用lua\_xmove将值传送到其它的线程。

唤醒挂起线程，需再次调用lua\_resume，Lua假设掌中所有的值通过yield调用来返回。你无需在lua\_resume和下一次resume之间接触线程栈，yield会返回栈中产生的确切值。

通常以Lua函数作为协程的主体。这个Lua函数可以调用其他函数，其中的任意函数都会间或的产生值，如此结束lua\_resume的调用。

function foo (x) coroutine.yield(10, x) end

function foo1 (x) foo(x + 1); return 3 end

C代码：

lua\_State \*L1 = lua\_newthread(L);

lua\_getglobal(L1, "foo1");

lua\_pushinteger(L1, 20);

lua\_resume(L1, L, 1);

调用lua\_resume会返回LUA\_YIELD(线程输出的标志)，至此L1栈中的可输出值有：

printf("%d\n", lua\_gettop(L1)); --> 2

printf("%d\n", lua\_tointeger(L1, 1)); --> 10

printf("%d\n", lua\_tointeger(L1, 2)); --> 21

当再次唤醒线程的时候，会从上次停止(yield)的地方继续运行。

lua\_resume(L1, L, 0);

printf("%d\n", lua\_gettop(L1)); --> 1

printf("%d\n", lua\_tointeger(L1, 1)); --> 3

再次调用的时候，会返回LUA\_OK,表示正常返回，协程结束。

协程也能调用C函数，是从其他Lua函数回调。C函数本身也能输出，这样的话，当协程唤醒的时候，需要提供持续的函数(continuation function)以便调用。为了输出值，C函数必须调用以下函数

lua\_yieldk，将这个函数作为返回语句使用。

int lua\_yieldk (lua\_State \*L, int nresults, int ctx,

lua\_CFunction k);

static inf myCfunction (lua\_State \*L) {

...

return lua\_yieldk(L, nresults, ctx, k);

}

\*连续，即接着上一次的调用处接着运行。

以上调用立刻挂起运行着的协程，参数nresults是返回栈中值的数量，这些值是相关的lua\_resume再次调用返回。ctx是传入到连续的环境信息。k是连续函数。当协程唤醒的时候，控制权直接转到连续函数中，输出后，myCfunction 不能做其他的事情。它必须为连续代理更多的工作(work)。

int prim\_read (lua\_State \*L) {

if (nothing\_to\_read())

return lua\_yieldk(L, 0, 0, &prim\_read);

--lua\_yield(L, nres);

lua\_pushstring(L, read\_some\_data());

return 1;

}

return lua\_yield(L, nres);

经过以上调用，当线程再次唤醒控制权返回给调用myCfunction的函数

Lua State

每次调用luaL\_newstate会创建一个新的Lua State，不同的Lua state是完全彼此独立的，state之间是不共享数据的，不管Lua State内部发生什么，都不会影响另外的State。这也意味着LuaState之间不会交互，因而需要一些C的中间代码。

如给定两个Lua State L1和L2以下命令将L1栈顶的字符串入栈到L2中

lua\_pushstring(L2, lua\_tostring(L1, -1));

因为数据必须要通过C来传递，Lua State仅能交换C中表示的数据类型，其他的类型必须要序列化后才能被传递。

系统提供了多线程，为每个线程创建了独立的Lua State

这种架构导致了线程和UNIX的进程类似，即无需共享内存，即可有并发操作。

lproc库

在多线程环境中使用多个Lua State：

lproc.start(chunk)：开启进程，运行指定块

lproc.send(channel,val1,val2,...)：发送值

lproc.receive(channel)：接受值

lproc.exit()：退出

同步进行的：进程发送字符串消息到通道中，阻塞进程，直到有一个进程从通道中接受；一个从通道中接受信息阻塞，直到有一个进程发送。

总是打开基本库是一个好的习惯；你也需要打包库。在打开新的states时我们可以写自己的打开库的函数而非调用luaL\_openlibs。当一个进程需要库中的某些库时，显式的加载调用。require会调用相关的luaopen\_\*函数。

内存管理

Lua动态的分配数据结构，这些结构在需要的时候增长，最终收缩或者消失，Lua对内存的使用控制得很严格，我们关闭Lua State得时候，Lua显式释放State的所有内存，甚至于所有的Lua对象都是垃圾回收器回收的东西：不仅式表和字符串，还有函数、线程、模块(实际上是表)。

对大多数应用来说，Lua管理内存是很便捷的，一些特殊的应用需要适配，如在内存限制的环境中或者精简垃圾回收的最少数量。Lua允许两个层面的适配：底级层面，设置Lua使用的分配函数；高级层面。设置控制垃圾回收器的参数，或者直接控制垃圾回收器。

分配函数

Lua核心不对分配内存做任何假设，不调用malloc 或者realloc来分配内存，它调用一个单一的allocation function来分配或者取消分配内存。allocation function在用户创建Lua State的时候必须提供。

函数luaL\_newstate(创建State)是一个以默认分配函数来创建LuaState的辅助函数。默认的分配函数使用C库中标准的malloc–realloc–free函数，对于常规应用很好了。这些函数对于常规的应用已经差不多了。然而，很容易获得完全的Lua分配控制。

--创建Lua state

lua\_newstate

lua\_State \*lua\_newstate (lua\_Alloc f, void \*ud);

这个函数接受两个参数：一个分配函数和一个用户数据，以这种方式创建的state，其内存分配通过调用函数f来分配或者取消分配。(即使lua\_State结构也是通过f分配的)

分配函数lua\_Alloc类型如下：

typedef void \* (\*lua\_Alloc) (void \*ud,void \*ptr,

size\_t osize,size\_t nsize);

第一个参数是user data，为创建的state准备的

第二个参数是重分配或者释放的块的地址

第三个参数是原始块的大小

若ptr不为空(NULL)，Lua确保块分配osize大小的存储

Lua将0大小的块视作NULL，nsize是0 的时候，分配函数释放ptr所指向的块，并返回NULL，NULL和需要分配的块的大小(0)相关。指针ptr为NULL的时候，分配函数必须分配并返回指定块的大小；如果不能分配指定块的时候，会返回NULL。若ptr为NULL，nsize为0，分配函数不做什么，并返回NULL。

最后若两者(ptr不为NULL和nsize不为0)都不为空的时候，分配函数会重新分配块，并返回地址。若发生错误，则返回NULL。当新分配的大小小于或等于原来的，Lua的分配函数会成功分配。垃圾回收期间，Lua会收缩数据结构，错误发生处不可恢复已分配的块。

void \*l\_alloc (void \*ud, void \*ptr, size\_t osize, size\_t nsize) {

if (nsize == 0) {

free(ptr);

return NULL;

}

else

return realloc(ptr, nsize);

}

通过调用lua\_getallocf获得Lua state内存分配器

lua\_Alloc lua\_getallocf (lua\_State \*L, void \*\*ud);

若ud不为空，函数为分配器设置\*ud的(user date)值。

可以调用lua\_setallocf来改变Lua state的内存分配器。

新的分配器会为之前分配的块的释放。新函数是包含原来函数的包装器(wrapper)，如追踪分配的栈或者同步存取栈。

Lua内部不缓存释放的内存块以便再次使用，Lua假设分配函数做了这个缓存，也不最小化碎片，好的分配器不会创建许多碎片。碎片的产生是由于分配器的策略而非程序的行为。

一个好的分配器是很难完成并实现的，Lua给你一个再分配或者释放的块，一个特俗的分配器不会保存块的大小、每个块精简的内存信息。

另一种情况是再多线程系统中改善内存分配，这种系统要求内存分布函数同步，以便系统好使用全局资源。然而要存取Lua state也必须要同步，严格限制到单线程中。如果每个Lua state 从私有的线程池中分派内存，分派器会避免额外的同步花费。

垃圾回收器

Lua5.0版本，Lua使用的标记清除的垃圾回收器(GC)，这个回收器也称stop-the-world，也就是说随着时间的推移，Lua停止解释主程序，以便不影响整个垃圾回收器的周期。整个周期包含三个阶段：标记、清洁、打扫。

Lua标记阶段：将活动着的(alive)标记为根集合，包含Lua直接存取的对象：registry和主线程，存储在活动对象的任意对象都是可被程序获取的。因此也被标记为alive，当所有可获取的对象标记为alive的时候，标记阶段结束。

Lua清洁阶段：涉及到相关的终结器和弱表。终结器遍历所有标记为终结的对象，并查找非标记的对象，这些对象被标记为alive并放在一张分离的列表中。这个列表在终结的时候使用；其次，Lua遍历弱表，并将其从入口中移除，因此键和值未被标记。

Lua打扫阶段：遍历所有Lua对象，将所有创建的对象保存在一个链接列表中。若对象未被标记未alive，Lua回收这个对象。除此而外，Lua清除标记，等待下一个回收周期，在清扫阶段，Lua调用在清洁阶段分离的终结的对象。

Lua5.1版本使用增量式回收器，增量式回收器和原先的回收器的步骤一样，在运行期间无需暂停整个运行程序。相反增量式回收器和解释器交互式运行。每当解释器分派一些内存的时候，回收器运行一小步。增量式回收器运行的时候，解释器把对象隔离开来。为确保回收器的正确性，解释器会带来一些障碍，包括检测危险的操作并纠正对象涉及的标记。

垃圾回收器的API

Lua提供了一些API允许我们控制垃圾回收器：

C的角度：

int lua\_gc (lua\_State \*L, int what, int data);

Lua的角度：collectgarbage(what [, data])

what参数在C中是枚举值，在Lua中是字符串：

|  |  |
| --- | --- |
| 停止垃圾回收器 | LUA\_GCSTOP(“stop”) |
| 重启垃圾回收器 | LUA\_GCRESTART(“restart”) |
| 垃圾回收 | LUA\_GCCOLLECT(“collect”) |
| 返回千字节数 | LUA\_GCCOUNT(“count”) |
| 部分垃圾回收工作 | LUA\_GCSTEP(“step”) |
| Lua使用的碎片数量 | LUA\_GCCOUNTB(not available) |

计算字节数：(lua\_gc(L, LUA\_GCCOUNT, 0) \* 1024)

+ lua\_gc(L, LUA\_GCCOUNTB, 0)

Lua中collectgarbage("count")返回一个浮点数，

计算字节数：collectgarbage("count") \* 1024

LUA\_GCSETPAUSE (“setpause”)：设置垃圾回收器停止。值以data的百分数形式。

LUA\_GCSETSTEPMUL (“setstepmul”)：设置垃圾回收器的step multiplier参数。

任何垃圾回收器是以CPU的时间来争取使用CPU的。极端的时候，垃圾回收器不会运行，在大内存消耗的时候不会花费CPU的时间。另一些极端情况下，垃圾回收器每当可存取graph改变的时候就会做一次完整的回收。在大量CPU消耗的时候，程序会使用需要的确切的最小化内存。在两种极端的情况下，好的垃圾回收器会找到一个好的均衡的。

和分配器函数一样，Lua的垃圾回收器对大多数应用时足够的。在一些场景，需要去优化回收器。两个参数pause和stepmul允许控制回收器的字符。

pause参数控制在完成回收和下一次性的回收之间的等待时间。pause zero使得Lua在上一次的回收一结束就看是新的回收。

pause 200% 要2倍的内存使用才开始新的回收(内存要有两倍大)。

如果想要内存使用的情况降低，设置更低的pause以便争得更多的CPU时间。pause值应该在0-200%的范围内。

stepmul：控制回收器的每千字节的内存分配，stepmul的值越大，回收器越是慢增长。一个大的值100000000%使得回收器像不增长的回收器，默认值是200%。值小于100%使collector回收缓慢，以至于不能完成回收。

lua\_gc的其他选项让你在collector运行的时候，可以控制回收器。游戏是这种例子的经典客户端。例如你在一些时期不想垃圾回收器工作，调用collectgarbage("stop")停止，再调用collectgarbage("restart")重启。系统中有周期性的空闲时期，你可以让回收器stopped，并再空闲时期调用collectgarbage("step",n)，为了在空闲时期设置更多的任务。你可以实验性的设置n的值或者在循环中调用collectgarbage(),把n设置为0，直到回收结束结束。

经过了漫长的时间，终于把这本书啃完了（Programing in Lua）。还好是把时间设定在六月底，六月底，我完成了！！！

回顾一下整本书，Lua的学习包含基础语法知识、协程、表、C的相互调用、函数等。

多用吧，在用的过程中才会有经验。

\*全局的虚拟表\_G,