1、-i //进入交互模式

-e //传入命令参数给lua

doFile（文件名） //导入文件名文件

-l //同doFile功能，导入相关文件

lua -i -e "\_PROMPT='DingYong:'" //进入交互模式，并以DingYong作为提示符

2、

3、type函数可以测试给定变量或者值的类型

4、a是一个元包，a.x与a[x]是不同的，它实际上是a中的”x”键，等价于a[“x”]

userdata可以将C数据存置入Lua变量中

5、java中的do…while…语句在lua不在存在，而是采用repeat…until…模式

如果一个函数有多个返回值，并且函数不是最后的或唯一的右值表达式，它只返回一个返回值。

6、可以强制一个函数只返回一个返回值，而只需要将返回值列表用圆括号括起来



7、理解闭包的概念，理解按钮回调和重写math.sin函数功能方法

do

localoldOpen = io.open

io.open = function(filename, mode)

ifaccess\_OK(filename, mode) then

returnoldOpen(filename, mode)

else

return nil, "access denied"

end

end

end

8、理解局部函数在回调上面的机制。

理解尾调用机制

9、掌握用闭包实现迭代器原理，学习下列文件单词迭代原理。



10、学习掌握如下迭代器原理

function iter(a, i)

print("test i = " .. i);

i = i + 1

local v = a[i]

if v then

return i, v

end

end

function mypairs(a)

return iter, a, 0

end

--测试如下

a = {"one", "two", "three"}

for i, v in mypairs(a) do

print(i, v)

end

11、执行f = loadfile("foo.lua")后，foo函数已被编译但还没有被定义，必须采用f()方式定义，方能被识别为函数，之后的代码才可调用foo(10)这样的例子

12、loadstring总是在全局环境中编译它的字符串，即首先在全局环境中查询相关变量信息。

当一个协同程序正在运行时，是无法在外部终止它的，唯一能够将其挂起的方式就是它本身调用yield将自己挂起

13、Lua使用了真正的垃圾收集算法，当发现程序使用了太多的内存，它就会遍历所有的数据并释放那些不再被需要的数据（即垃圾数据）。这个算法通常都有很好的性能（这也说明了Lua的快速并非偶然），但是下述代码（逐行读取一个文件中的内容）中的循环令垃圾收集算法失去了其应有的高效。更为有效的实现是通过io.raed(\*all)实现。

localbuff = ""

for line in io.lines() do

buff = buff .. line .. "\n"

end

14、有如此规律：定义一个函数FUN(可有参数)，其后声明FUN的表，则会自动调用函数FUN，即创建同名表时，会回调其同名之函数，可以当做回调函数理解。调用原则是首先检查有参数的函数，如果不存在有参数函数，再调用无参数函数。

元表的作用，指定一个表的元表之后，可以定义一些共有的行为模式，如对表的加减乘除等。重写元表的某些指定方法后，执行相关操作就会回调相应函数实现。

15、\_\_newindex元方法用于更新表，而\_\_index元方法则用于访问表

16、学习对表中元素变动的监听：

t = {} -- original table (created somewhere)

local \_t = t

t = {}

local mt = {

\_\_index = function(t, k)

print("\*access to element ".. tostring(k))

return \_t[k] -- access the original table

end,

\_\_newindex = function(t, k, v)

print("\*update of element ".. tostring(k) .. " to ".. tostring(v))

\_t[k] = v -- update original table

end

}

setmetatable(t, mt)

--测试

t[2] = 'hello'

print(t[2])

17、学习创建只读表的例子

function readOnly(t)

local proxy = {}

local mt = { -- create metatable

\_\_index = t,

\_\_newindex = function(t, k, v)

error("attempt to update a read-only table",2)

--参数2表示将错误信息重定向到试图执行更新操作的代码

end

}

setmetatable(proxy, mt)

return proxy

end

--测试

days = readOnly{"Sunday", "Monday", "Tuesday", "Wednesday",

"Thursday", "Friday", "Saturday"}

print(days[1]) --> Sunday

days[2] = "Noday"

18、Lua将运行环境本身存储在全局变量\_G中，可以通过如下方式查看具体值：

for n in pairs(\_G) do

print(n)

end

19、rawget或rawset方法可以绕过元方法直接对\_G的访问，了解具体做法。

20、实现一个简单类的方法：

实现方法1.

Account = {balance = 0}

Function Account.withdraw(v)

Account.balance = Account.balance - v

End

上例表示声明一个类Account，并定义一个类中的函数withdraw

实现方法2.

Account = {

balance=0,

withdraw = function(self, v)

self.balance = self.balance - v

end

}

21、类中函数的申明方式：

1.以“类名.函数名（self…）”形式声明的函数，在调用的时候，可以以“对象.函数名（对象…）”或者“对象：函数名（…）”的形式调用函数。

2. 以“类名：函数名（…）”形式声明的函数（在函数内部还是可以使用self），在调用的时候，只能以“对象：函数名（…）”的形式调用函数。

22、类继承之后的搜索机制需要理解。

子类继承父类之后，子类调用某个方法会首先在自身查找该方法，如果该方法存在则调用，如果查找不到则向上搜索，直到搜索到为止，即如果Lua调用一个原始表中不存在的函数，\_\_index域对应的函数将被直接调用。实现子类继承的发放如下：

Function Account:new(o)

o = o or{}

setmetatable(o, self)

self.\_\_index = self

retur o

end

23、掌握多重继承原理，理解多重继承的实现方法。

24、理解如下私有函数原理（self不可由外部调用，balance也不可以通过外部显示的赋值等操作），不能显示地调用对象，只能通过其提供的方法调用。

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

--测试

acc1 = newAccount(100.00)

acc1.withdraw(40.00)

print(acc1.getBalance())

25、掌握单方法对象的实现，理解如下单方法对象的写法：

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

--测试

d = newObject(0)

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

d("set", 10)

print(d("get"))

26、一个常见的错误是试图对表的索引进行排序。在一个表中，所有索引可以被视为一个集合，因此无序的。如果想对它们进行排序，就必须将它们复制到一个数组，然后对这个数组进行排序。

27、ipairs和pairs的区别是：前者利用诸如1、2、……的索引顺序进行遍历，后者使用自然的随机顺序。

28、理解如下排序函数的原理（通过对“键”排序的迭代输出）

Function pairsByKeys(t, f)

Local a = {}

For n in pairs(t) do

table.insert(a, n)

end

table.sort(a, f)

local i = 0 -- iterator variable

local iter = function() -- iterator function

i = i + 1

if a[i] == nil then

return nil

else

return a[i], t[a[i]]

end

end

return iter

end

--测试

For name, line in pairsByKeys(lines) do

print(name, line)

end

29、匹配字符表：

. 任意字符

%a 字母

%c 控制字符

%d 数字

%l 小写字母

%p 标点字符

%s 空白符

%u 大写字母

%w 字母和数字

%x 十六进制数字

%z 字符“\0”

上述字符类的大写形式表示小写形式所代表的集合的补集。例如，“%A”表示非字母的字符

30、你可以通过使用方括号将字符类或字符括起来的方式创建自己的字符集（Char-Set），比如，“[%w\_]”可以匹配字母、数字和下划线、“[01]”可以匹配二进制数字、而“[%[%]]”能够匹配一对方括号（%表示转意字符）。你也可以在字符集中使用字符范围，比如，“%d”等价于“[0-9]”、“%x”等价于“[0-9a-fA-F]”。可以在模式的开始处使用“^”来表示其补集：“[^0-7]”匹配任何不是八进制数字的字符“[^\n]”匹配任何非换行符的字符。注意，你可以使用大写的字符类表示其补集，因为“%S”比“[^%s]”要简短些。对于以上的匹配以及捕获特性，在需要的时候可以专门学习。

未解决问题：

P32中描述的table.sort函数的具体用法是怎么样的？

项目中的未理解问题：

* lua中的类和java中的类有什么区别，使用class定义类，并不是lua规范的，设计是如何实现的？
* 如果我想实现一个界面，需要继承哪个类？Node、State类是用作什么功能的？

local scene=new(SceneEx, XXlayout, self)表示将一个layout生成一个场景是如何实现的。

* 在layout中声明一个event=”test”之后，再在实现界面的文件中复写onEvent函数，就能实现按钮的监听，具体是如何实现的？

Lua学习笔记

* 格式化打印输出

function printf(fmt, ...)

io.write(string.format(fmt, ...))

end

printf("Hello %s from %s on %s\n",

os.getenv"USER" or "there", \_VERSION, os.date())

-------- Output ------

Hello there from Lua 5.1 on 05/02/13 11:32:57

* 排序

a={2}

table.insert(a,3);

table.insert(a,4);

table.sort(a,function(v1,v2) return v1 > v2 end)

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

-------- Output ------

1 4

2 3

3 2

3、一种对象实现方式

function create(name, id)

local obj = { name = name, id = id }

function obj:SetName(name)

self.name = name

end

function obj:GetName()

return self.name

end

function obj:SetId(id)

self.id = id

end

function obj:GetId()

return self.id

end

return obj

end

o1 = create("Sam", 001)

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

"o1's id:", o1:GetId())

o1:SetId(100)

o1:SetName("Lucy")

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

"o1's id:", o1:GetId())

输出结果：

o1's name: Sam o1's id: 1

o1's name: Lucy o1's id: 100

* 简单继承

function createRobot(name, id)

local obj = { name = name, id = id }

function obj:SetName(name)

self.name = name

end

function obj:GetName()

return self.name

end

function obj:GetId()

return self.id

end

return obj

end

function createFootballRobot(name,

id, position)

local obj = createRobot(name, id)

obj.position = "right back"

function obj:SetPosition(p)

self.position = p

end

function obj:GetPosition()

return self.position

end

return obj

end

* 基于对象的实现方式

function create(name, id)

local data= { name = name, id = id

}

local obj = {}

function obj.SetName(name)

data.name = name

end

function obj.GetName()

return data.name

end

function obj.SetId(id)

data.id = id

end

function obj.GetId()

return data.id

end

return obj

end

o1 = create("Sam", 001)

o2 = create("Bob", 007)

o1.SetId(100)

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

o2.GetId())

o2.SetName("Lucy")

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

"o2's name:", o2.GetName())

输出结果：

o1's id: 100 o2's id: 7

o1's name: Sam o2's name: Lucy

* 基于原型的继承



prototype模式

一个对象既是一个普通的对象，同时也可以作为创建其他对象的原型的对象（即类对象，class object）；动态的改变原型对象的属性就可以动态的影响所有基于此原型的对象；另外，基于一个原型被创建出来的对象可以重载任何属于这个原型对象的方法、属性而不影响原型对象；同时，基于原型被创建出来的对象还可以作为原型来创建其他对象。

* 函数环境

function foo()

print(g or "No g defined!")

end

foo()

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

foo()

print(g or "No g defined!")

输出结果：

No g defined!

100

No g defined

函数环境定义了函数执行时的环境情况（即只针对函数本身的一些变量值等），可以实现函数的“安全沙箱”。

* 关于包的问题

--testP.lua:

pack = require"mypack" --导入包

print(ver or "No ver defined!")

print(pack.ver)

print(aFunInMyPack or

"No aFunInMyPack defined!")

pack.aFunInMyPack()

print(aFuncFromMyPack or

"No aFuncFromMyPack defined!")

aFuncFromMyPack()

--mypack.lua:

module(..., package.seeall) --定义包

ver = "0.1 alpha"

function aFunInMyPack()

print("Hello!")

end

\_G.aFuncFromMyPack =

aFunInMyPack

执行testP.lua的输出结果：

No ver defined!

0.1 alpha

No aFunInMyPack defined!

Hello!

function: 003CBFC0

Hello!

定义

包是一种组织代码的方式。

实现方式

一般在一个Lua文件内以module函数开始定义一个包。module同时定义了一

个新的包的函数环境，以使在此包中定义的全局变量都在这个环境中，而非

使用包的函数的环境中。理解这一点非常关键。

以前面的代码为例， “module(..., package.seeall)”的意思是定义一个

包，包的名字与定义包的文件的名字相同（除去文件名后缀，在前面的代码

中，就是“mypack”），并且在包的函数环境里可以访问使用包的函数环境

（比如，包的实现使用了print，这个变量没有在包里定义，而是定义在使

用包的外部环境中）。

使用方式

一般用require函数来导入一个包，要导入的包必须被置于包路径（package

path）上。包路径可以通过package.path或者环境变量来设定。一般来说，

当前工作路径总是在包路径中。

其他

8、Lua中的堆栈

lua\_pushvalue(L, 3)    --> 10 20 30 40 50 30\*  
lua\_pushvalue(L, -1)   --> 10 20 30 40 50 30 30\*  
lua\_remove(L, -3)      --> 10 20 30 40 30 30\*  
lua\_remove(L, 6)      --> 10 20 30 40 30\*  
lua\_insert(L, 1)      --> 30 10 20 30 40\*  
lua\_insert(L, -1)      --> 30 10 20 30 40\* (no effect)  
lua\_replace(L, 2)      --> 30 40 20 30\*  
lua\_settop(L, -3)      --> 30 40\*  
lua\_settop(L, 6)      --> 30 40 nil nil nil nil\*

**断定堆栈中是什么类型**

int lua\_type            (lua\_State \*L, int index);  
int lua\_isnil           (lua\_State \*L, int index);  
int lua\_isboolean       (lua\_State \*L, int index);  
int lua\_isnumber        (lua\_State \*L, int index);  
int lua\_isstring        (lua\_State \*L, int index);  
int lua\_istable         (lua\_State \*L, int index);  
int lua\_isfunction      (lua\_State \*L, int index);  
int lua\_iscfunction     (lua\_State \*L, int index);  
int lua\_isuserdata      (lua\_State \*L, int index);  
int lua\_islightuserdata (lua\_State \*L, int index);

**从堆栈中取出值来**

int            lua\_toboolean   (lua\_State \*L, int index);  
lua\_Number     lua\_tonumber    (lua\_State \*L, int index);  
const char    \*lua\_tostring    (lua\_State \*L, int index);  
size\_t         lua\_strlen      (lua\_State \*L, int index);  
lua\_CFunction lua\_tocfunction (lua\_State \*L, int index);  
void          \*lua\_touserdata (lua\_State \*L, int index);  
lua\_State     \*lua\_tothread    (lua\_State \*L, int index);  
void          \*lua\_topointer   (lua\_State \*L, int index);

**将值写入堆栈中**

void lua\_pushboolean       (lua\_State \*L, int b);  
void lua\_pushnumber        (lua\_State \*L, lua\_Number n);  
void lua\_pushlstring       (lua\_State \*L, const char \*s, size\_t len);  
void lua\_pushstring        (lua\_State \*L, const char \*s);  
void lua\_pushnil           (lua\_State \*L);  
void lua\_pushcfunction     (lua\_State \*L, lua\_CFunction f);  
void lua\_pushlightuserdata (lua\_State \*L, void \*p);1. string库中所有的字符索引从前往后是1,2,...;从后往前是-1,-2,...

2. string库中所有的function都不会直接操作字符串，而是返回一个结果

s = "[abc]"

string.len(s) <==返回5

string.rep("abc", 2) <==返回"abcabc"

string.lower("ABC") <==返回"abc"

string.upper("abc") <==返回"ABC"

string.sub(s, 2) <==返回"abc]"

string.sub(s, -2) <==返回"c]"

string.sub(s, 2, -2) <==返回"abc"

string.format(fmt, ...)返回一个类似printf的格式化字符串

string.find(s, pattern, pos)

第1个参数：源字符串

第2个参数：待搜索之模式串

第3个参数：A hint, 从pos位置开始搜索

找到匹配返回：匹配串开始和结束的位置，否则返回nil

简单的模式串

s = "hello world"

i, j = string.find(s, "hello")

print(i, j) --> 1 5

print(string.sub(s, i, j)) --> hello

print(string.find(s, "world")) --> 7 11

i, j = string.find(s, "l")

print(i, j) --> 3 3

print(string.find(s, "lll")) --> nil

格式化的模式串

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

下面的表列出了Lua支持的所有字符类：

. 任意字符

%s 空白符

%p 标点字符

%c 控制字符

%d 数字

%x 十六进制数字

%z 代表0的字符

%a 字母

%l 小写字母

%u 大写字母

%w 字母和数字

上面字符类的大写形式表示小写所代表的集合的补集。例如，'%A'非字母的字符：

模式串中的特殊字符

( ) . % + - \* ? [ ^ $

'%' 用作特殊字符的转义字符

'%.' 匹配点；

'%%' 匹配字符 '%'。

转义字符 '%'不仅可以用来转义特殊字符，还可以用于所有的非字母的字符。当对一个字符有疑问的时候，为安全起见请使用转义字符转义他。

用'[]'创建字符集

'[%w\_]' 匹配字母数字和下划线

'[01]' 匹配二进制数字

'[%[%]]'匹配一对方括号

在'[]'中使用连字符'-'

'%d' 表示 '[0-9]'；

'%x' 表示 '[0-9a-fA-F]'

'[0-7]' 表示 '[01234567]'

在'[]'开始处使用 '^' 表示其补集：

'[^0-7]' 匹配任何不是八进制数字的字符；

'[^\n]' 匹配任何非换行符户的字符。

'[^%s]' == '%S'

模式修饰符

+ 匹配前一字符1次或多次

\* 匹配前一字符0次或多次;最长匹配

- 匹配前一字符0次或多次;最短匹配

? 匹配前一字符0次或1次

^ 匹配字符串开头

$ 匹配字符串结尾

捕获：用()将要捕获的部分包围起来

pair = "name = Anna"

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

print(key, value) <== name Anna

拷贝捕获(%1-%9)

s = "abc \"it\'s a cat\""

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

print(q) <== it's a cat 如果%d代表第几个捕获的拷贝。

string.gsub(s, pattern, reps)

第1个参数：源字符串

第2个参数：待替换之模式串

第3个参数：替换为reps

将s中所有符合pattern的字串替换为reps，返回结果串+匹配数

print(string.gsub("hello, world", "o", "a")) <== hella, warld 2

gsub也可以用拷贝捕获技巧

print(string.gsub("hello, world", "(o)", "%1-%1")) <== hello-o, wo-orld 2

print(string.gsub("hello Lua", "(.)(.)", "%2%1")) <== ehll ouLa 4

function trim (s) return (string.gsub(s, "^%s\*(.-)%s\*$", "%1")) end <== 注意匹配数用括号丢弃

string.gsub(s, pattern, func)

第3个参数：自定义函数，对找到的匹配操作，并传出替换值

s, n = string.gsub("hello world", "l+", function(s) return "xxx" end)

print(s, n) <== hexxxo worxxxd 2

string.gfind(s, pattern)

返回一个迭代器，迭代器每执行一次，返回下一个匹配串；

iter = string.gfind("a=b c=d", "[^%s+]=[^%s+]")

print(iter()) <== a=b

print(iter()) <== c=d

通常用于泛性for循环,下面的例子结果同上

for s in string.gfind("a=b c=d", "[^%s+]=[^%s+]") do

print(s)

end

string.char函数和string.byte函数用来将字符在字符和数字之间转换。string.char获取0个或多个整数，将每一个数字转换成字符，然后返回一个所有这些字符连接起来的字符串。string.byte(s, i)将字符串s的第i个字符的转换成整数；第二个参数是可选的，缺省情况下i=1。下面的例子中，我们假定字符用ASCII表示：

print(string.char(97)) --> a

i = 99; print(string.char(i, i+1, i+2)) --> cde

print(string.byte("abc")) --> 97

print(string.byte("abc", 2)) --> 98

print(string.byte("abc", -1)) --> 99

上面最后一行，我们使用负数索引访问字符串的最后一个字符。

Lua提供了string.format()函数来生成具有特定格式的字符串, 函数的第一个参数是格式(formatstring), 之后是对应格式中每个代号的各种数据. 由于格式字符串的存在, 使得产生的长字符串可读性大大提高了. 这个函数的格式很像C语言中的printf().函数string.format在用来对字符串进行格式化的时候，特别是字符串输出，是功能强大的工具。这个函数有两个参数，你完全可以照C语言的printf来使用这个函数。第一个参数为格式化串：由指示符和控制格式的字符组成。指示符后的控制格式的字符可以为：十进制'd'；十六进制'x'；八进制'o'；浮点数'f'；字符串's'。在指示符'%'和控制格式字符之间还可以有其他的选项：

格式字符串可能包含以下的转义码:

%c - 接受一个数字, 并将其转化为ASCII码表中对应的字符

%d, %i - 接受一个数字并将其转化为有符号的整数格式

%o - 接受一个数字并将其转化为八进制数格式

%u - 接受一个数字并将其转化为无符号整数格式

%x - 接受一个数字并将其转化为十六进制数格式, 使用小写字母

%X - 接受一个数字并将其转化为十六进制数格式, 使用大写字母

%e - 接受一个数字并将其转化为科学记数法格式, 使用小写字母e

%E - 接受一个数字并将其转化为科学记数法格式, 使用大写字母E

%f - 接受一个数字并将其转化为浮点数格式

%g(%G) - 接受一个数字并将其转化为%e(%E, 对应%G)及%f中较短的一种格式

%q - 接受一个字符串并将其转化为可安全被Lua编译器读入的格式

%s - 接受一个字符串并按照给定的参数格式化该字符串

为进一步细化格式, 可以在%号后添加参数. 参数将以如下的顺序读入:

(1) 符号: 一个+号表示其后的数字转义符将让正数显示正号. 默认情况下只有负数显示符号.

(2) 占位符: 一个0, 在后面指定了字串宽度时占位用. 不填时的默认占位符是空格.

(3) 对齐标识: 在指定了字串宽度时, 默认为右对齐, 增加-号可以改为左对齐.

(4) 宽度数值

(5) 小数位数/字串裁切: 在宽度数值后增加的小数部分n, 若后接f(浮点数转义符, 如%6.3f)则设定该浮点数的小数只保留n位, 若后接s(字符串转义符, 如%5.3s)则设定该字符串只显示前n位.

在这些参数的后面则是上述所列的转义码类型(c, d, i, f, ...).

print(string.format("pi = %.4f", PI))

--> pi = 3.1416

d = 5; m = 11; y = 1990

print(string.format("%02d/%02d/%04d", d, m, y))

--> 05/11/1990

tag, title = "h1", "a title"

print(string.format("<%s>%s</%s>", tag, title, tag))

--> <h1>a title</h1>

第一个例子，%.4f代表小数点后面有4位小数的浮点数。第二个例子%02d代表以固定的两位显示十进制数，不足的前面补0。而%2d前面没有指定0，不足两位时会以空白补足。对于格式串部分指示符得详细描述清参考lua手册，或者参考C手册，因为Lua调用标准C的printf函数来实现最终的功能。

以下是一些例子:

string.format("%%c: %c", 83) 输出S

string.format("%+d", 17.0) 输出+17

string.format("%05d", 17) 输出00017

string.format("%o", 17) 输出21

string.format("%u", 3.14) 输出3

string.format("%x", 13) 输出d

string.format("%X", 13) 输出D

string.format("%e", 1000) 输出1.000000e+03

string.format("%E", 1000) 输出1.000000E+03

string.format("%6.3f", 13) 输出13.000

string.format("%q", "One\nTwo") 输出"One\

　　Two"

string.format("%s", "monkey") 输出monkey

string.format("%10s", "monkey") 输出 monkey

string.format("%5.3s", "monkey") 输出 mon

Lua 的模式匹配不使用Posix规范的正则表达式（也写做regexp）来进行模式匹配。主要的原因出于程序大小方面的考虑：实现一个典型的符合POSIX标准的regexp大概需要4000行代码，这比整个Lua标准库加在一起都大。权衡之下，Lua中的模式匹配的实现只用了500行代码，当然这意味着不可能实现POSIX所规范的所有更能。然而，Lua中的模式匹配功能是很强大的，并且包含了一些使用标准POSIX模式匹配不容易实现的功能。

Lua支持的所有字符类：

. 任意字符

%a 字母

%c 控制字符

%d 数字

%l 小写字母

%p 标点字符

% s 空白符

%u 大写字符

%w 字母和数字

%x 十六进制数字

%z 代表0的字符

上面字符类的大写形式表示小写所代表的集合的补集。例如，'%A'非字母的字符：

print(string.gsub("hello, up-down!", "%A", "."))

--> hello..up.down. 4

例子：

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

在模式匹配中有一些特殊字符，他们有特殊的意义，Lua中的特殊字符如下：

( ) . % + - \* ? [ ^ $

'%' 用作特殊字符的转义字符，因此 '%.' 匹配点；'%%' 匹配字符 '%'。转义字符 '%'不仅可以用来转义特殊字符，还可以用于所有的非字母的字符。当对一个字符有疑问的时候，为安全起见请使用转义字符转义他。

对Lua而言，模式串就是普通的字符串。他们和其他的字符串没有区别，也不会受到特殊对待。只有他们被用作模式串用于函数的时候，'%' 才作为转义字符。所以，如果你需要在一个模式串内放置引号的话，你必须使用在其他的字符串中放置引号的方法来处理，使用 '/' 转义引号，'/' 是Lua的转义符。

你可以使用方括号将字符类或者字符括起来创建自己的字符类（译者：Lua称之为char-set，就是指传统正则表达式概念中的括号表达式）。比如，'[%w\_]' 将匹配字母数字和下划线，'[01]' 匹配二进制数字，'[%[%]]' 匹配一对方括号。

下面的例子统计文本中元音字母出现的次数：

\_, nvow = string.gsub(text, "[AEIOUaeiou]", "")

在char-set中可以使用范围表示字符的集合，第一个字符和最后一个字符之间用连字符连接表示这两个字符之间范围内的字符集合。大部分的常用字符范围都已经预定义好了，所以一般你不需要自己定义字符的集合。比如，'%d' 表示 '[0-9]'；'%x' 表示 '[0-9a-fA-F]'。然而，如果你想查找八进制数，你可能更喜欢使用 '[0-7]' 而不是 '[01234567]'。你可以在字符集(char-set)的开始处使用 '^' 表示其补集：'[^0-7]' 匹配任何不是八进制数字的字符；'[^/n]' 匹配任何非换行符户的字符。记住，可以使用大写的字符类表示其补集：'%S' 比 '[^%s]' 要简短些。

可以使用修饰符来修饰模式增强模式的表达能力，Lua中的模式修饰符有四个：

+ 匹配前一字符1次或多次

\* 匹配前一字符0次或多次

- 匹配前一字符0次或多次

? 匹配前一字符0次或1次

'+'，匹配一个或多个字符，总是进行最长的匹配。比如，模式串 '%a+' 匹配一个或多个字母或者一个单词：

print(string.gsub("one, and two; and three", "%a+", "word"))

--> word, word word; word word

'%d+' 匹配一个或多个数字（整数）：

i, j = string.find("the number 1298 is even", "%d+")

print(i,j) --> 12 15

'\*' 与 '+' 类似，但是他匹配一个字符0次或多次出现.一个典型的应用是匹配空白。

比如，为了匹配一对圆括号()或者括号之间的空白，可以使用 '%(%s\*%)'。（ '%s\*' 用来匹配0个或多个空白。由于圆括号在模式中有特殊的含义，所以我们必须使用 '%' 转义他。）再看一个例子，'[\_%a][\_%w]\*' 匹配Lua程序中的标示符：字母或者下划线开头的字母下划线数字序列。

'-' 与 '\*' 一样，都匹配一个字符的0次或多次出现，但是他进行的是最短匹配。某些时候这两个用起来没有区别，但有些时候结果将截然不同。比如，如果你使用模式 '[\_%a][\_%w]-' 来查找标示符，你将只能找到第一个字母，因为 '[\_%w]-' 永远匹配空。另一方面，假定你想查找C程序中的注释，很多人可能使用 '/%\*.\*%\*/'（也就是说 "/\*" 后面跟着任意多个字符，然后跟着 "\*/" ）。然而，由于 '.\*' 进行的是最长匹配，这个模式将匹配程序中第一个 "/\*" 和最后一个 "\*/" 之间所有部分：

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

print(string.gsub(test, "/%\*.\*%\*/", "<COMMENT>"))

--> int x; <COMMENT>

然而模式 '.-' 进行的是最短匹配，她会匹配 "/\*" 开始到第一个 "\*/" 之前的部分：

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

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

--> int x; <COMMENT> int y; <COMMENT>

'?' 匹配一个字符0次或1次。举个例子，假定我们想在一段文本内查找一个整数，整数可能带有正负号。模式 '[+-]?%d+' 符合我们的要求，它可以匹配像 "-12"、"23" 和 "+1009" 等数字。'[+-]' 是一个匹配 '+' 或者 '-' 的字符类；接下来的 '?' 意思是匹配前面的字符类0次或者1次。

与其他系统的模式不同的是，Lua中的修饰符不能用字符类；不能将模式分组然后使用修饰符作用这个分组。比如，没有一个模式可以匹配一个可选的单词（除非这个单词只有一个字母）。下面我将看到，通常你可以使用一些高级技术绕开这个限制。

以 '^' 开头的模式只匹配目标串的开始部分，相似的，以 '$' 结尾的模式只匹配目标串的结尾部分。这不仅可以用来限制你要查找的模式，还可以定位（anchor）模式。比如：

if string.find(s, "^%d") then ...

检查字符串s是否以数字开头，而

if string.find(s, "^[+-]?%d+$") then ...

检查字符串s是否是一个整数。

'%b' 用来匹配对称的字符。常写为 '%bxy' ，x和y是任意两个不同的字符；x作为匹配的开始，y作为匹配的结束。比如，'%b()' 匹配以 '(' 开始，以 ')' 结束的字符串：

print(string.gsub("a (enclosed (in) parentheses) line",

"%b()", ""))

--> a line

常用的这种模式有：'%b()' ，'%b[]'，'%b%{%}' 和 '%b<>'。你也可以使用任何字符作为分隔符。

正则测试代码：

----------------以下是模式匹配功能-------------------

--[[s = "abc \"it\'s a cat\""

print("s = "..s)

\_,\_,\_,q = string.find(s, ("([\"'])(.-)%1")) --这个没搞懂

print(q)

print(string.gsub("hello Lua", "(.)(.)", "%2%1"))

s, n = string.gsub("hello world", "l+", function(s) return "xxx" end)

print(s, n)

iter = string.gfind("a=b c=d", "[^%s+]=[^%s+]");

print(iter());

print(iter());

i = 99; print(string.char(i, i+1, i+2,i+3,i+4));

d = 5; m = 11; y = 1990;

print(string.format("%02d/%02d/%04d", d, m, y))

print(string.format("%%c%%d: %c", 83))

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

print("One\nTwo")

print(string.format("%q", "One\nTwo"));

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

s = "hello world from Lua"

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

print(w);

end

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

t = {}

s = "from=world, to=Lua"

for k, v in string.gmatch(s, "(%w+)=(%w+)") do

t[k]=v

end

for k, v in pairs(t) do

print(k.."---->", v)

end

print("--------------------------hello%")

print(string.gsub("hello world", "[%w+]", "test"))

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

print(string.gsub("hello world", "(%w+)%s\*(%w+)", "%2 %2"))

string.gsub("hello world", "%w+", print)

lookupTable = {["hello"] = "hola", ["world"] = "mundo"}

print(string.gsub("hello world", "(%w+)", lookupTable))

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

print(string.gsub("home = $home, user = $USER", "$(%w+)", os.getenv));]]--

--[]内的模式是或的形式匹配

print(string.gsub("[this is a test]", "[%[%]]", "v"));

print(string.gsub("[this is(),( ) a test]", "%(%s\*%)", "v"));

print(string.gsub("[824 \_\_ok \_test \_542 a test]", "[\_%a][\_%w]\*", "v"));

print(string.gsub("[824 \_\_ok \_test \_542 a test]", "[\_%a][\_%w]-", "v"));

print(string.gsub("[824 \_\_ok/\* \_tes/\*t \_5\*/42 a \*/test]", "/%\*.-%\*/", "v"));

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

local s="this 6is a stest";

if string.find(s, "^%d") then

print("------------yes");

end

print(string.gsub("hello world", "%a\*", "k"))

-- '%b' 用来匹配对称的字符。常写为 '%bxy' ，x和y是任意两个不同的字符；x作为匹配的开始，y作为匹配的结束。比如，'%b()' 匹配以 '(' 开始，以 ')' 结束的字符串：

print(string.gsub("a (enclosed (in) parentheses) line","%b()", "k")) --换成%(.\*%)思考

s = "abc \' \"its a cat\" he\'he"

print("s = "..s)

print(string.gsub(s,"([\"%'])(.-)%1", "k")) --%1表示第一次匹配到的