Lua程序设计语言简述

Lua语言入门

Lua是一门小巧的脚本语言。其设计的初衷是能够嵌入其他宿主应用程序中（Lua语言代码和C/C++语言代码是很容易互相调用的），从能够保证程序执行效率依然比较高，并且其开发速度大大提升。Lua在游戏领域使用非常广泛，无论历史比较悠久的国内外客户端游戏（比如《大话西游》《魔兽世界》），还是一些比较火的移动端游戏（比如《我叫MT》《刀塔传奇》），无论是客户端还是服务端，都采用了Lua语言作为脚本语言甚至业务层主力语言。

Hello, world!

要想运行Lua程序，首先需要安装Lua解释器。Lua解释器会在官网上面以源代码的方式发布，用户可以下载源代码以后，自行使用C语言编译器和makefile工具进行编译。因为cocos引擎、Openresty等项目支持的Lua版本是5.1（或者是以其为蓝本的LuaJIT），所以本课程会介绍Lua的5.1版本。

```
$ curl -R -O http://www.lua.org/ftp/lua-5.1.5.tar.gz
#也可以在其他途径下载
$ tar xfvz lua-5.1.5.tar.gz
$ cd lua-5.1.5
$ make linux
#如果make过程提示链接错误，则可以使用apt命令安装对应的共享库
$ sudo make install
# 当安装完成以后，如果输入lua命令，能够进入交互式环境，就说明安装成功了
$ lua
```

使用lua命令可以进入Lua交互式环境，按下`ctrl+d`或者输入`os.exit()`语句，就能退出交互式环境。

Lua解释器安装成功以后，接下可以使用文本编译器编辑一个名为`hello.lua`文件，文件内容如下：

```
-- hello.lua
print("Hello, world!")
```

```
#随后在shell当中输入如下命令
$lua hello.lua
```

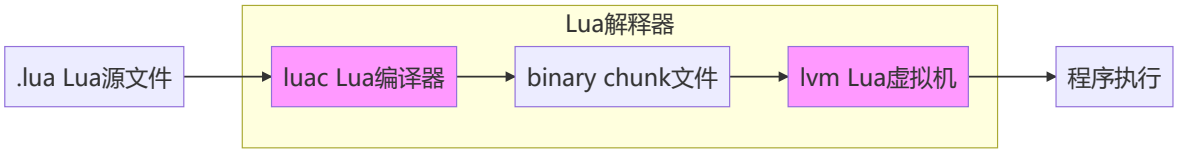
除此以外，还可以使用Lua语言脚本来执行Lua语言程序，将下列文件保存，然后增加执行权限，就可以直接执行。

```
#!/usr/local/bin/lua
print("Hello World!")
```

Lua的编译、解释和命令行参数

Lua是一门脚本语言，也是一门解释型语言（可以根据源码来动态地执行程序）。相较于其他解释性语言，Lua的运行效率确非常突出，其高效的秘诀在于：Lua在语言中内置了虚拟机，当Lua脚本运行时，并不是解释器一段段地直接解释运行Lua，而是先将Lua脚本使用Lua编译器(`luac`)编译成字节码，再交给虚拟机(`lvm`)来执行。

Lua字节码运行的载体称为程序段(`chunk`)。程序段就是一段可以被Lua解释器解释执行的代码。程序段可以小到一条语句，也可以大到包含成千上万语句和各种复杂的函数定义。通常一个程序段中的代码都会保存在一个以`.lua`为后缀的文件当中(比如之前所使用的`hello.lua`)。



将程序段经过Lua编译器编译以后得到了由中间代码构成的二进制文件，称为二进制程序段，也叫预编译程序段。在Lua代码中，可以调用函数`loadfile/dofile`来将另一个Lua源文件/二进制程序段加载运行。

使用`lua`命令可以启动解释器，如果`lua`命令后面以一个代码文件作为参数，则会将这个代码文件加载进Lua解释器执行。而使用`luac`命令可以将Lua源文件直接编译成二进制chunk文件，使用二进制chunk文件可以加快程序的加载速度，也可以在一定程度上保护源代码；`luac`命令也可以反编译二进制chunk文件。

在使用`lua`命令启动解释器时，解释器会首先查找`LUA_INIT`的环境变量（这个环境变量可以控制首先执行的文件），然后解释器会解析传入的参数，将所有的命令行参数存储在一个名为`arg`的列表中。

```lua
print(-1,arg[-1])
print(0,arg[0])
-- 从此处可以看出，列表从1开始，键并不限制为正整数
for i = 1,#arg do -- #是求列表长度运算符
    print(i,arg[i])
end
print(-2,arg[-2])
```

```
$ lua 4_arg.lua a b c d
-1  lua
0   4_arg.lua
1   a
2   b
3   c
4   d
-2  nil
```

标识符、关键字、注释和语法规范

Lua当中标识符的规则和C一致，开头是字母或者是下划线，其余部分为字母、数字或者下划线，对大小写敏感。

Lua的关键字如下：

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

使用`--`可以实现单行注释，使用`--[[内容]]`可以实现多行注释

```
-- 单行注释
--[[
注意-和[之间不能有空格
多行注释
]]
```

多个Lua语句之间可以不需要任何符号进行分隔，也可以使用分号进行分隔。对于空白字符和空行，Lua编译器无视。

变量、值和数据类型

在Lua语言中，变量无需声明即可使用。变量默认为全局变量，如果变量使用时还没有初始化，那么它的数值为`nil`。显然，删除一个全局变量的方法就是将其赋值为`nil`。

```
print(a)
a = 1
print(a)
```

Lua是动态类型语言，变量不携带类型信息，值携带类型信息。因此，任何Lua变量可以被赋予任何类型的值。Lua的值有8种数据类型，分别为nil、布尔、数字、字符串、表、函数、线程和用户数据。其中nil、布尔、数值、字符串为基础数据类型。

使用type函数可以根据一个值返回其类型的名称

```
t = type(nil) --> nil
print(t)
t = type(true) --> boolean
print(t)
t = type(false) --> boolean
print(t)
t = type(1234) --> number
print(t)
t = type("Hello world") --> string
print(t)
t = type(io.stdin) --> userdata
print(t)
t = type(print) --> function
print(t)
t = type(type) --> function
print(t)
t = type(type(123)) --> string
print(t)
```

nil和布尔类型

nil表示无效值，没有赋值的变量默认值为`nil`，将一个变量赋值`nil`相当于删除变量。

布尔类型有两个：`true`和`false`。在Lua中，条件测试将除了`false`和`nil`的所有其他值视为真，特别地数值0和空字符串也是真。

逻辑运算符：与、或、非

逻辑运算符`and`的结果为：如果第一个操作数为`false`或者`nil`，返回第一个操作数，否则返回第二个操作数

逻辑运算符`or`的结果为：如果第一个操作数不为`false`和`nil`，返回第一个操作数，否则返回第二个操作数

```
print (4 and 5)
-- 5
print (nil and 13)
-- nil
print (false and 13)
-- false
print (0 or 5)
-- 0
print (false or "hi")
-- "hi"
print (nil or false)
-- false
```

`and`和`or`遵循短路求值原则，并且`and`的优先级高于`or`。下面是两个示例：

```
x = x or v
a and b or c
-- 如果b为真，那么当a为真时，返回b；当a为假时，返回c
```

逻辑运算符`not`返回一个布尔类型的值。

```
not 0
not not nil
-- not只能返回true或者false
```

数值

Lua5.3以前，数值只有浮点数(cocos-Lua引擎采用的Lua版本是5.1)，之后的版本数值有浮点数和整数两种。需要明确的一点是，Lua的浮点是双精度的，所以它可以精确地表示32位的任何整数，并不需要担忧由于采用浮点数存储导致无法判断整数是否相等的问题。

```
print(4) -- 整数形式
print(0.4) -- 小数形式
print(4.0e1) --指数形式
print(40 == 4.0e1) --整数和浮点数只要实际数值一致 数值就是相同的 true
print(3/2) -- 除法时，按照浮点数规则运算1.5
print(3%2) -- 1
print(3.2%2.1) -- 1.1 取余运算可以使用小数i
print(1 ~= 2) -- Lua的不相等使用~= 其余关系运算符和C一致
print(2^(0.5)) -- ^是指数运算符
```

数值的字面值表示方法、加、减、乘、除、取余以及各种关系运算符和C几乎一致。其中，有区别的地方有：**Lua当中使用整数形式和浮点数形式表示的值只要实际数值一致，数值就相同。除法运算会自动转换成浮点数。取余运算可以使用浮点数作为操作数。不相等运算符使用的符号是~=。**

字符串

Lua中的字符串是字符的序列。字符使用8位来进行存储。同时，Lua也支持使用utf-8等编码方法来存储Unicode编码字符串。Lua的字符串是常量，其内容无法修改。

```
-- 在lua中单双引号表现一致
a = 'a line'
b = "another line"
-- lua支持C风格的转义字符(如果采用'作为字符串边界，那么转义字符\'在串内表示单引号的含义)、
"one line\nnext line\n\"in quotes\",'in quotes'"
-- 使用[[ ]]可以界定多行字符串字面值（忽略第一个换行符）
[[
<head>
</head>
]]
-- 使用[=n个=[]=n个=]可以自定义界定符号
[===[
a = b[c[i]]
]===]
-- ..运算符可以实现字符串连接
print(#"1234")
-- #运算符可以获取字符串的长度
print("123".."234")
```

字符串和数值之间的转换

当使用加、减、乘、除等运算符时，如果左右操作数当中有字符串值，Lua会将字符串自动转换成数值。类似地，..运算符是字符串连接运算符，它的左右操作数当中如果有数值，会强制转换成字符串。当然，也可以使用tonumber函数将字符串转换成数值。如果字符串的形式无法转换成数值，那些表达式的结果和函数的返回值为nil。

```
-- print函数的参数必须是字符串
print("10" + 1)
-- +运算符的操作数是数值，所以首先将"10"转换成10，运算后的结果再转换成字符串
print("-5.3e-10"*"2")
-- 两个操作数都转换成数值
print(1 .. 2)
-- 两个操作数转换成字符串，注意中间必须要有空格，否则会触发语法错误
line = io.read() --从stdin中读取一行
n = tonumber(line) --将这一行强制转换成数值
if n == nil then --判断是否转换成功
```

```
    error(line.." is not a valid number") -- error函数可以终止程序，并输出报错
else
    print(n*2)
end
```

字符串基本标准库

使用标准库可以创建、连接、比较字符串、获取长度等等操作，比较基本的操作有这些：

库函数	效果
string.len	获取字符串长度
string.upper	将字母转换成大写
string.lower	将字母转换成小写
string.rep	重复字符串，构造新的字符串
string.reverse	翻转字符串
string.sub	子串
string.byte	获取串中字符的ASCII码
string.char	根据ASCII码，转成字符

下面是代码示例：

```
string.len("Hello") -- 获取长度
string.lower("A Long Line") -- 转换成小写
string.upper("A Long Line") -- 转换成大写
string.lower(a) < string.lower(b) -- 比较字符串
string.rep("abc",3) -- 重复3次
string.reverse("A Long Line!") -- 翻转
-- 提取子串
s = "[in brackets]"
string.sub(s,2,-2) -- 第2个字符到倒数第2个字符
string.sub(s,1,1) -- 第1个字符
string.sub(s,-1,-1) -- 最后一个字符
-- 字符和ASCII转换
string.char(97) -- a
string.byte("abc") -- 97
string.byte("abc",2) -- 98
string.byte("abc",-1) -- 99
string.byte("abc",1,2) -- 97 98
```

使用`string.format`可以实现格式化输出。`format`函数的效果与C语言的`printf`函数几乎一致，只有一些稍微的区别：format函数不支持`*`，`l`和`p`等控制符。除此以外，因为`format`函数的本质是调用了C当中的`printf`函数，而C语言的字符串是以'\0'结尾的，所以如果Lua风格的字符串当中有'\0'字符，就不能使用`s`控制符，为此，`format`函数引入了`q`控制符来显示。

总之，`string.format`函数支持`c`，`d`，`e`，`f`，`g`，`i`，`o`，`u`和`x`等运算符。其中`s`和`q`可以用来控制字符串的显示。

```lua
a = 3.1416
str = string.format("%f",a)
-- 不支持l str = string.format("%lf",a)
b = "a\0b"
str = string.format("%s",b) -- 打印a
str = string.format("%q",b) -- 打印"a\000b"
print(str)
```

字符串的内存实现和性能优化

在Lua当中，所有的字符串数据都存储在一个全局的区域。每个存放字符串的变量并没有存放字符串的副本，而是对字符串数据的引用。每次新建字符串变量的时候，如果这个变量的字符串数据已经使用过，那些这个变量就会引用之前已经创建好的字符串数据。

字符串数据是不可以修改的，所以如果修改字符串变量的内容，实际上是新建了一个字符串数据，并且修改了变量的引用，原来的字符串数据会根据使用情况，由垃圾回收机制回收存储空间。

从上述可以知，字符串的使用过程中需要频繁地判断字符串数据是否重复出现过，所以Lua在全局的区域创建了一个散列桶（就是使用链表法处理冲突的哈希表）数据结构来组织字符串。

因此，当使用字符串的时候应当尽可能减少新字符串数据的创建。大量的临时字符串数据会往散列桶中插入大量数据，并且由于绝大多数数据没有使用，所以在垃圾回收阶段也会消耗很多时间。

```lua
-- 代码1 使用大量临时字符串
a = os.clock()
local s = ''
for i = 1,300000 do
    s = s .. 'a'
end
b = os.clock()
print(b - a)
-- 代码2 使用列表来存储字符
a = os.clock()
local s = ''
local t = {}
for i = 1,300000 do
    t[#t + 1] = a
end
s = table.concat(t,'')
b = os.clock()
print(b - a)
```

赋值语句

Lua支持使用 = 将值赋值给变量。除此以外Lua还支持多重赋值，其中值和变量使用 , 进行分隔。

```lua
x = 1
print(x)
x,y = 2,3
print(x,y)
```

在多重赋值当中，Lua先对等号右边的所有元素求值，然后才执行赋值。所以可以使用多重赋值来交换变量数值。

```
x,y = y,x -- 交换xy
```

如果 = 左右两边值的数量和变量的数量不相等时，多余的值会被抛弃，多余的变量会赋值为nil

```
a,b,c = 1,2
print(a,b,c)
a,b = 1,2,3
print(a,b)
```

通常来说，多重赋值的主要应用场景是交换变量的值和和获取函数的多个返回值。其余场景下如无必要，不推荐使用。

变量作用域

除了全局变量以外，使用local关键字还可以定义局部变量。局部变量的作用域局限于定义语句所在的块。在Lua当中，使用do/end或者是条件/循环结构可以显式地指定块的范围，否则的话，所有在同一个程序段中的语句都属于同一个块。

```
x = 1
do
    local y = 1
    print("x = ",x)
    print("y = ",y)
end
print("x = ",x)
print("y = ",y)
for i = 1,3 do
    print("i = ", i)
end
print("i = ", i)
```

在允许的条件下，程序中应该要尽量可能多地使用局部变量。局部变量拥有更好的可读性，避免使全局环境混乱。另外，局部变量的效率更高，因为一旦作用域结束时，垃圾回收器就会回收局部变量的内存。

Lua中有一种编程技巧，就是使用和全局变量重名的局部变量，并且使用全局变量的值来对其初始化。这样可以保证在作用域当中不会修改全局变量的值，同时提升了程序的执行速度。

```
x = 2
do
    local x = x -- 使用全局变量给同名局部变量赋值
    if x == 2 then
        print("x == 2")
    end
end
```

条件结构和循环结构

条件结构

使用if/then/else/end关键字可以实现条件结构，如之前所述，条件测试语句中，把nil和false当做假，其余都是真（包括0和空字符串）。如果要使用嵌套的if结构，可以使用elseif关键字

```
a = io.read("*n") -- 从标准输入中读取一个数字
if a == nil then
    print("a is not a number")
end
if a < 160 then
    print("short")
elseif a < 184 then
    print("middle")
else
    print("high")
end
```

循环结构

while

使用while/do/end可以实现循环的效果，程序会先测试while的条件是否为真，然后决定进入循环或者终止循环。

```
local i = 1
local cnt = 0
while i <= 100 do
    cnt = cnt + i
    i = i + 1
end
print("cnt = ", cnt)
```

repeat

使用repeat/until也可以实现循环的结果，不过程序会先执行循环体，之后再检查条件是否为真。

```
local cnt = 0
local i = 1
repeat
    cnt = cnt + i
    i = i + 1
until i > 100
print("cnt =",cnt)
```

值得注意的是，在循环体里面定义的局部变量，其作用域包括until后面的条件测试语句。

```
x = 20
local sqr = x/2
repeat
    sqr = (sqr + x/sqr)/2
    local error = math.abs(sqr^2 - x)
until error < 0.0001 -- 此处error依然有效
print(sqr)
```

数字for

使用数字for可以实现固定次数的循环，其中exp1表示初始值，exp2表示终止值，可选的exp3表示每次的步长，如果不写exp3，则默认为1。

```
--[[
数字for的语法如下:
    for var = exp1,exp2,exp3 do
        body
    end
]]
local cnt = 0
for i = 1,100 do
    cnt = cnt + i
end
print("cnt =",cnt)
```

如果不想限制循环的次数，可以使用常量math.huge，通常可以配合break关键字一起使用。

```
local cnt = 0
for i = 1,math.huge do
    cnt = cnt + i
    if i >= 100 then
        break
    end
end
print("cnt =",cnt)
```

Lua的数字for和其他语言的for结构会有一些区别。for的3个表达式都只会执行一次，所以如果exp2当中使用了函数的话，只根据其第一次运算结果来确定执行次数。

```
function f(x)
    return 5*x
end
x = 1
for i = 1,f(x) do
    print("Hello")
    x = 2 -- 不影响总运行次数
end
```

此外，循环变量是局部变量，循环体外无法使用。如果需要在循环结构以外获取循环变量的数值，可以使用另一个作用域有效的变量进行存储。

```
str = "awesome"
for i = 1,#str do
    ch = string.sub(str,i,i)
    if ch == "s" then
        pos = i
        break
    end
end
print("pos =",pos) --不能直接输出i
```

表

Lua提供了唯一一种数据结构，就是**表**。Lua表的功能很强大，可以直接当做数组和列表使用，也可以用来实现其他数据结构。

表的本质是一个**关联数组**。表中存储了两两关联的键值对，其中**键**（索引）可以是除了nil或者NaN以外的所有值，**值**（元素）可以是任何Lua值。

使用{}运算符可以实现**构造表达式**从而创建新的表，并且可以将表的**引用**赋值给变量。表是动态分配的，内存的分配和回收是采用引用计数的方法来管理的。

通过`[]`和`.`运算符可以访问表内的元素，表内元素的使用方法和普通变量一致。表的对键的类型没有任何要求，如果表元素没有初始化，那么它的值为nil（实际上Lua中所有的全局变量都集中在一个表内管理的）。`.`运算符的效果是采用其右操作数的字符串形式来访问表，比如`t["name"]`和`t.name`就是等价的，注意由于语法分析器的设计问题，`.`后必须使用符合标识符规范的单词，不过按照普遍的阅读习惯，`.`通常是表示访问成员的含义，`[]`运算符通常是表示根据索引访问元素的含义。

```
a = {}
k = "x"
a[k] = 10
print(a["x"]) -- 10
a[20] = "great"
k = 20
print(a[k]) -- great
a["x"] = a["x"] + 1
print(a["x"]) -- 11

x = "y"
a[x] = 10
print(a[x]) -- 10
print(a.x) -- 等价于a["x"] nil
print(a.y) -- 等价于a["y"] 10
```

当使用数值作为键时，只要值的实际值是一致的，那么不同的写法都会访问到同一个元素；如果采用字符串值作为键，那么必须要字符串值的内容完全一致，对应的元素才是同一个（例如`a["0"]`和`a["+0"]`没有访问同一个元素）。数值`0`和字符串`"0"`作为键的时候，所对应的元素不一致。

```
a = {}
a[0] = 1
a["0"] = 2
print(a[0], a["0"], a['0'], a['+0'], a[tonumber('0')])
-- 1 2 2 nil 1
print(a[0], a[0.0])
-- 1 1
```

表的构造器

使用构造器可以初始化表。`{}`是最简单的构造器，称为空构造器。除此以外，还可以往括号中添加多个值，从而初始化表。采用这种**列表式构造器**初始化的表，其索引为从1开始递增的整数。

```
days = {"Sunday", "Monday", "Tuesday", "Wednesday", "Thursday", "Friday",
"Saturday"}
for i = 1,7 do
    print(days[i]) -- 使用从1开始的整数来访问列表
end
```

除此以外，还可以采用**记录式构造器**。这样的写法类似于初始化结构体变量的成员。

```
a = {x = 10, y = 20}
-- 采用记录式构造器 其索引为"x"和"y"
print(a.x,a.y)
print(a["x"], a["y"])
```

使用构造器的时候，可以混用列表式和记录式：

```
polyline = {
    color = "blue",
    thickness = 2,
    npoints = 4,
    {x = 0, y = 0}, -- polyline[1]
    --这里采用了列表式，列表中元素也是另一个列表
    {x = -10, y = 0}, -- polyline[2]
    {x = -10, y = 1}, -- polyline[3]
    {x = 0, y = 1} -- polyline[4]
}
for i = 1,4 do
    print(polyline[i].x, polyline[i]["y"])
end
```

列表式构造器和记录式构造器也有一些局限性：列表式的索引必须是从1开始递增，不能修改；记录式的`=`左边必须是符合标识符规范的单词。可以对构造器做一些修改，使其支持各种数值的索引和所有字符串值索引。

```
t = {
    -- -1 = 1
    -- 1 = 1 这里必须符合标识符的规范
    ["1"] = 1 -- 使用[""]可以使用不符合标识符的索引
}
print(t["1"])
i = 20
a = {
    [-1] = 5,
    [i - 16] = 7,
    8 -- a[1]
}
print(a[-1],a[i-16],a[1])
```

列表当中的最后一个逗号是可选的，写不写不会影响结果。

记录、数组和序列

如果表的键全都是字符串类型，那么这个表称为**记录**；如果表的键全部都是正整数(不包含0)，那么这个表称为**数组**。如果数组中某个值是nil，则称为**空洞**；如果一个数组当中没有空洞，则该数组称为**序列**。使用`#`可以获取序列的长度，不要将`#`用在有空洞的数组里面。

```
a = {'a','b','c','d','e'}
for i = 1,#a do
    print(a[i])
end
```

泛型for

在编程实践当中，往往需要需要采用迭代器模式来访问表。迭代器是一种顺序访问列表元素而又不暴露表的内部结构的设计模式。对于表，Lua内置了pairs迭代器，而对于序列，Lua提供了ipairs迭代器。迭代器可以配合泛型for来遍历表中的所有元素，而关于迭代器具体设计，会等到函数阶段再详细讲解，我们先熟悉迭代器的使用。

```
giant = {health = 30, magic = 20, damage = 10, armor = 5}
for k,v in pairs(giant) do
    print(k,v)
end
t = {10, print ,12 , "hi"}
for k,v in ipairs(t) do
    print(k,v)
end
```

表的安全访问

有些时候，用户想要访问表中的某个函数，但是用户并不知道这个函数是否存在，所以通常需要一个条件结构来检查函数的存在性。而在实际的工程项目当中，表的层级是多级的，所以这种条件结构处理起来会非常繁琐，所以可以采用一些简单的方式来处理问题。

```
-- t = (table or {}).member
-- 当table为nil时，table or {}返回nil， t也为nil
-- 采用这种做法可以减少列表的访问次数
zip = (((company or {}).director or {}).address or {}).zipcode
```

表标准库

Lua提供了一系列标准库函数来操作表和表中元素。

```
t = {}
for line in io.lines() do
-- io.lines()是一个迭代器，每次从文件中读取一行
    table.insert(t,line) --尾部插入
end
print(#t)
for line in io.lines() do
    num = tonumber(line)
    table.remove(t,num) -- 删除下标为num的元素，并填上空洞
    for k,v in pairs(t) do
        print(num,k,v)
    end
end
```

使用表标准库来实现栈

```
function push(stack,val)
    table.insert(stack,val) -- 入栈
end
function pop(stack)
    table.remove(stack) -- 出栈
end
stack = {} --创建一个空栈
```

```
for line in io.lines() do
    push(stack,line)
    for k,v in ipairs(stack) do
        print(line,k,v)
    end
end
for i = 1,#stack do
    pop(stack)
    print("--------pop--------")
    for k,v in ipairs(stack) do
        print(k,v)
    end
end
```

表的底层实现

Lua表的本质上是关联数组(字典)，里面存储的是两两关联的键值对。所以Lua虚拟机底层采用了一种由哈希表和数组混合的数据结构来实现。当传入键时，如果键是整数或者是能够转换为整数的浮点数时，直接按索引访问数组部分，否则就传入哈希表进行查找。在表中，哈希表是同时采用了链表法和开放地址法来解决冲突问题。哈希表的每一项都是一个头结点，而头结点还存储一个元素，称为主位置元素。

> 查找的流程：
>
> 如果key是一个正整数，并且小于数组大小，则在数组中查找
>
> 否则，在哈希表中查找，先求出哈希值，然后根据哈希值访问哈希表，如果发生冲突，采用链表法来解决

> 新增的流程：
>
> 首先，哈希表中数组部分和哈希表部分的大小都为0，当插入key的时候，无论key是否为整数，都需要计算其哈希值。
>
> 通过哈希值，可以获取其主位置元素，除此以外，还需要查找一个哈希表的空位置，如果没有空位置，则需要重新建立哈希表，而如果有空位置，就使用这个空位置来存储元素。再根据主位置元素的哈希值，决定主位置元素是否移动。
>
> > 在重建哈希表的过程中，会遍历所有索引为正整数的元素，根据索引的分布情况，来分配数组部分的大小。哈希表和数组部分的大小都是2的整数次幂，而数组部分大小应该是在每个2次方位置容纳的元素数量都超过了一半，其余的数据存储在哈希表中。比如`{1,2,3,[20] = 4}`这个列表，它的数组部分大小为4，索引20存储在哈希表当中。

使用列表应当注意：

- 同一个元素，在列表的规模不同的时候，可能分别存储在表的数组部分或者是哈希表部分。
- 避免向空表逐渐插入元素来扩大表的规模，因为表的各个部分初始大小从0、1、2、4增大的过程中会发生多次重建哈希表的过程。
- 在编程实践当中，尽量避免混用记录和数组。
- 不要往数组中主动插入nil值，这样会使#运算符失效

```
a = os.clock()
for i = 1,1000000 do
    t = {}
    t[1] = 1
    t[2] = 2
    t[3] = 3
```

```
    t[4] = 4
    t[5] = 5
    t[8] = 10
end
b = os.clock()
print(b-a)
a = os.clock()
for i = 1,1000000 do
    t = {1,2,3,4,5,[8] = 10}
end
b = os.clock()
print(b-a)
```

函数

函数调用

Lua当中函数调用的方法和C是一致。在书写了函数名以后，再使用一对圆括号将所有需要传递的实参括起来，如果不需要传递任何参数，也需要一个空的圆括号。当函数的参数只有一个而且参数是字符串值或者是表构造器的时候，可以忽略圆括号。如果一个表当中有类型为函数的元素，并且这个函数的参数第一个为列表本身，那么就可以使用`:`运算符（类似于通过对象访问方法）来访问函数。

```
print("Hello")
print "Hello"
t = type {}
-- print t 字符串必须是常量，才能忽略()
print(t)
```

```
o = {
    para = 1
}
function o.func1(self,arg1)
-- 等价于 o.func1 = function (self,arg1)
    print(self.para,arg1)
end
o:func1(2) -- 等价于o.func1(o,2)
```

函数定义

函数定义由**函数名**、（形式）**参数**列表和**函数体**组成。形式参数的行为和局部变量一致，在函数调用的时候，相当于用传入的实际参数的值来初始化一个局部变量（表现和C函数的值传递一致）。当调用的实参和定义的形参的个数不一致的时候，多余的实参会被丢弃，多余形参会被初始化为`nil`，从而可以实现类似多态的效果 。

```lua
function func(arg1,arg2,arg3)
    print(arg1,arg2,arg3)
end
func(1,2,3,4)
func(1,2,3)
func(1,2)
func(1)
func()
```

在函数定义中使用`or`运算符，可以实现默认参数的效果。

```lua
function default(arg1)
    arg1 = arg1 or 1 --默认参数1，如果arg1是nil，则被赋值为1
    print(arg1)
end
default()
default(2)
```

返回值

使用`return`关键字可以指定函数的返回值，特别地Lua当中的函数支持多返回值，只需在返回语句中把多个值使用`,`隔开即可。在不同的情况下，函数调用返回值数量会有所调整：如果函数被单独调用，则返回值无效；如果函数作为表达式的中间某一部分，那么，就只获取第一个返回值；只有当函数调用是**赋值**、**传入参数**、**表构造器**和`return`**表达式**的**最后一项/唯一一项**的时候，才会返回所有的返回值。

```lua
function func0() end
function func1() return 1 end
function func2() return 1,2 end
x,y = func2()
print(x,y)
x,y,z = 3,func2()
print(x,y,z)
x,y = func1()
print(x,y)
x,y = func0()
print(x,y)
x,y,z = func2(),3 -- 如果要返回多个值，不能是中间部分
print(x,y,z)

t = {func2()} -- t = {1,2}
for i = 1,#t do
    print(i,t[i])
end

function func(i) -- return 语句也可以获取所有的返回值
    if i == 0 then
        return func0()
    elseif i == 1 then
        return func1()
    elseif i == 2 then
        return func2()
    end
end
print(func(0))
print(func(1))
```

```
print(func(2))
print((func(2))) -- 加上圆括号就只有一个返回值
print(func(2)+3)
```

使用Lua实现快速排序

Lua当中也可以使用函数递归，具体的使用方法和C语言非常相似。下面是使用Lua实现快速排序的例子。

```
function swap(a,b) -- 典型的返回多返回值的函数
    return b,a
end
function sort(arr,left,right)
    if left < right then
        k = partition(arr,left,right)
        sort(arr,left,k-1)
        sort(arr,k+1,right)
    end
end
function partition(arr,left,right)
    local j = left
    local pivot = arr[right]
    for i = left,right do
        if(arr[i] <  pivot) then
            arr[i], arr[j] = swap(arr[i], arr[j])
            j = j + 1
        end
    end
    arr[j], arr[right] = swap(arr[j], arr[right])
    return j
end
a = {1,3,5,7,9,2,4,6,8,10}
sort(a,1,#a)
for k,v in ipairs(a) do
    print(k,v)
end
```

八皇后问题

在一个8*8大小的国际象棋的棋盘当中，希望能够将八个皇后都放置在棋盘上，并且互相的位置不会冲突（任意两个皇后不在同一个条横线、纵线或者是对角线上），这就是经典的八皇后问题。

解决八皇后问题，首先需要设计数据结构。使用大小为8的序列可以存储皇后们的位置，其中索引为横坐标，元素为纵坐标。接下来就是算法，可以采用深度优先遍历的方法解决问题。

```
N = 8
num = 1
function isplaceok(a,n,c)
    for i = 1,n-1 do -- a[i] = j 表示第i行第j列上已经有棋子了
        if (a[i] == c) or (a[i] - i == c - n) or (a[i] + i == c + n) then
            return false
        end
    end
    return true
end
function printsolution(a)
```

```
    print("case ",num,":")
    num = num +1
    for i = 1,N do
        for j = 1,N do
            io.write(a[i] == j and "X" or "-", " ")
        end
        io.write("\n")
    end
    io.write("\n")
end
function addqueen(a,n)
    if n > N then
        printsolution(a)
    else
        for c = 1,N do
            if isplaceok(a,n,c) then
                a[n] = c
                addqueen(a,n+1)
            end
        end
    end
end
addqueen({},1)
```

string.find函数

`string.find`就是一个典型的拥有多个返回值的函数，它会从字符串当中查找模板，然后返回第一个匹配的起始索引和结尾索引。使用多重赋值可以获取函数的所有返回值。

```
function out(x,y)
    print(x,y)
end
out(string.find("Hello world", "world"))
```

可变长函数参数

在参数列表的末尾（或者是唯一）的`...`表示该函数参数是可变长的。当这个函数被调用的时候，多余的参数会被收集起来，称为函数的**额外参数**。表达式`...`是函数收集到的所有额外参数，称为可变长参数表达式，那么，表达式`{...}`是指所有收集到的额外参数构成的列表。

```
function add (x, ...)
    local cnt = x
    for k,v in ipairs({...}) do
        cnt = cnt + v
    end
    return cnt
end
print(add(1))
print(add(1,2))
print(add(1,2,3))
```

调试

在Lua当中，可以在函数的内部定义另一个函数。所以通常可以配合上可变参数来在程序运行过程中输出函数传入的参数。

```
function func(x,y,z)
    return (x or 0) + (y or 0) + (z or 0)
end
function test(...)
    print("call func", ...)
    return func(...)
end
print(test(1,2,3,4))
print(test(1,2))
```

格式化输出

`string.format`不能直接将结果打印到屏幕上，所以利用可变参数可以比较容易地将格式化和输出的功能合并。

```
function luaprintf(fmt, ...)
    return io.write(string.format(fmt,...))
end
name = "Liming"
age = 22
luaprintf("I am %s. I am %d\n", name, age)
```

unpack函数

`unpack`可以将序列的所有元素转换成多个值，从而可以作为函数的多个返回值。利用`unpack`函数，可以使用给任意函数传入任意多个参数。

```
print(unpack({1,2,3}))
a,b = unpack({1,2,3})
--
f = string.find
a = {"hello", "ll"}
print(f(unpack(a)))
```

在Lua中，使用递归可以自己实现一个unpack函数。

```
function myunpack(t,i,n)
    i = i or 1
    n = n or #t
    if i <= n then
        return t[i], myunpack(t,i+1,n)
    end
end
a,b = myunpack({1,2,3})
print(a,b)
```

select函数

`select`函数可以用来获取`...`当中部分参数。`select`的第一个参数要么是数字i，表示第i个以后的参数；要么是字符串`"#"`，表示参数的总数量。

```
function add(...)
    local cnt = 0
    for i = 1, select("#", ...) do
        cnt = cnt + select(i, ...)
    end
    return cnt
end
```

闭包

在Lua中，函数是**第一类值**，也就是所谓的“一等公民”。函数可以像数值一样赋值给变量，在函数的内部也可以定义另一个函数，使用起来和其他值无异。如果一个函数以其他函数为参数，或者返回其他函数，这个函数就称为**高阶函数**，否则就是**一阶函数**。在函数内部定义其他函数的时候，内部定义的函数称为**嵌套函数**，外部的函数称为**外围函数**。

在Lua当中，所有的函数类型的值都是匿名的。函数定义的本质就是在内存当中分配空间，创建一个函数类型的值（称为**闭包**），然后将闭包的引用赋值给变量。而在函数调用的过程中，Lua虚拟机会找到闭包的内存位置，然后执行函数内部的指令。

```
function a (x,y)
    print(x,y)
end
--等价于
a = function (x,y) print(x,y) end
```

变量的静态作用域

如果在函数内部定义了函数，Lua的外围函数的局部变量是可以被嵌套函数访问的，这种变量的作用域称为**静态作用域**或者**词法定界**。

```
function a()
    function b()
        print(x)
        x = 2
        print(x)
    end
    x = 1 -- 交换 x=1 和 b()会导致函数运行不一样的结果
    b()
end
a()
```

使用闭包捕获upvalue

当函数调用的时候，函数可以访问本函数内部定义的局部变量，也可以访问全局变量，还可以访问外围函数的局部变量。对嵌套函数的闭包而言，如果访问的是外围函数的局部变量，就称这个闭包捕获了`upvalue`。

```
local u = 1
local v = 2
local function f()
    u = v
end
print(u,v)
f()
print(u,v)
```

从上述例子可以看出，`f`函数也捕获了`u`和`v`这两个`upvalue`，而`u`和`v`实际上是`main`函数的局部变量，所以被`f`的闭包捕获也不足为奇了。 在函数`f`的前面加上`local`关键字，表示这个闭包的作用域限制在`main`函数内部（实际上的好处是减少了访问全局变量的指令，运行速度更快一点）。

闭包除了可以访问外围函数的局部变量以外，还可以从外围函数的外围函数中访问`upvalue`。实际上，无论其作用层级，只要处于静态作用域中，所有的局部变量都可以被其内部的嵌套函数访问。

```
local u = 1
local v = 2
local function f()
    local function g()
        u = v
    end
    g()
end
print(u,v)
f()
print(u,v)
```

利用闭包，可以实现简单的计数器。不过这里要注意闭包的释放情况。

```
function newCounter()
    local count = 0
    return function()
        count = count + 1
        return count
    end
end
print(newCounter()()) -- 执行完成以后，闭包被释放
print(newCounter()())
print(newCounter()())
c1 = newCounter()
print(c1()) -- c1保存了闭包的引用，不会释放
print(c1())
print(c1())
print(c1())
```

利用闭包实现回调函数

闭包也经常用于**回调函数**的使用。当程序执行时，应用程序经常会调用一些接口，但是部分接口在被调用时还需要传入一个函数，接口在实现时，首先会登记这个传入的函数，以便后面的某些情况下可以去真正调用它，从而完成目标任务。这个被传入并且会在之后被调用的函数就称为回调函数。在GUI的设计中，经常性地需要用到回调函数，比如在UI上面创建一个按钮就需要传入回调函数，但是要等到用户真正按下按钮时，合适的处理动作的回调函数才会被调用。

```
function Button(paralist)
    -- 这里只是一个例子，演示创建button的效果
    for k,v in pairs(paralist) do
        print(k,v)
    end
    return paralist
    -- 实际上的gui程序中，这里会调用图形库创建按钮
end
function messageButton(label)
    return Button({
        label = label,
        action = function () print("message", label) end
    })
    -- 把label和回调函数作为Button闭包的upvalue
end
newbutton = messageButton("hello")
-- 在创建完成按钮以后，也可以在稍后的时机调用它的回调函数
newbutton.action()
-- 也可以重新注册一个回调函数
newbutton.action = function () print("new messgae") end
newbutton.action()
```

在上述例子中，Button是一个创建按钮的函数，label是按钮的标签，action是回调函数。当messageButton函数执行时，会创建messageButton和Button的闭包，其中label和action会作为Button闭包的upvalue，只要闭包没有释放（也就是例子的newbutton没有销毁），我们在messageButton函数执行完成以后，依然可以重新注册另一个回调函数。

利用闭包重定义函数

即使是预定义函数，也可以改变它的定义。而利用闭包，可以在实现新的定义函数中使用原来函数的定义。下面是重新定义sin函数的例子。假设要重新定义sin函数，使其参数以角度为单位而不是以弧度为单位。

```
do
    local oldSin = math.sin
    local k = math.pi/180
    math.sin = function(x) return oldSin(x*k) end
    print(math.sin(30))
end
```

简单迭代器的实现

从前面所述可知，闭包可以保存外围函数的某个局部变量的变化情况，利用闭包的这个特性，我们可以实现迭代器来记录容器的访问位置并且顺序向后访问。实现迭代器通常需要两个函数，一个是迭代器本身，另一个是创建迭代器的**工厂**。工厂通常用于创建迭代器，它也会记录容器的各种信息以及迭代器的当前访问位置，而迭代器的闭包就会捕获这些值作为它的upvalue。

```lua
function factory(t) -- 工厂
    local i = 0
    return function () -- 迭代器的闭包
        i = i + 1
        return t[i]
    end
end
t = {10,20,30}
for element in factory(t) do
    print(element)
end
```

Lua函数调用的原理

使用`luac`命令的`-l`选项可以查看源代码经过预编译以后的字节码情况。我们会以一个简单的函数的例子来理解`Lua`函数的调用流程。

`lua`虚拟机由3个主要部分构成：`lua_state`、`lua`栈和函数调用栈。`lua_state`里面存储了虚拟机的所有全局状态，包括`lua`栈的基址、当前位置和栈顶，也包括了函数调用栈的相关信息。`lua`栈存储了所有的数据，包括全局变量、`upvalue`和局部变量。函数调用栈则与C语言有比较大的不同，函数调用栈是一个线性表，表当中的元素是各个函数栈帧，在`lua`当中，所有的栈帧实际上是闭包的引用。

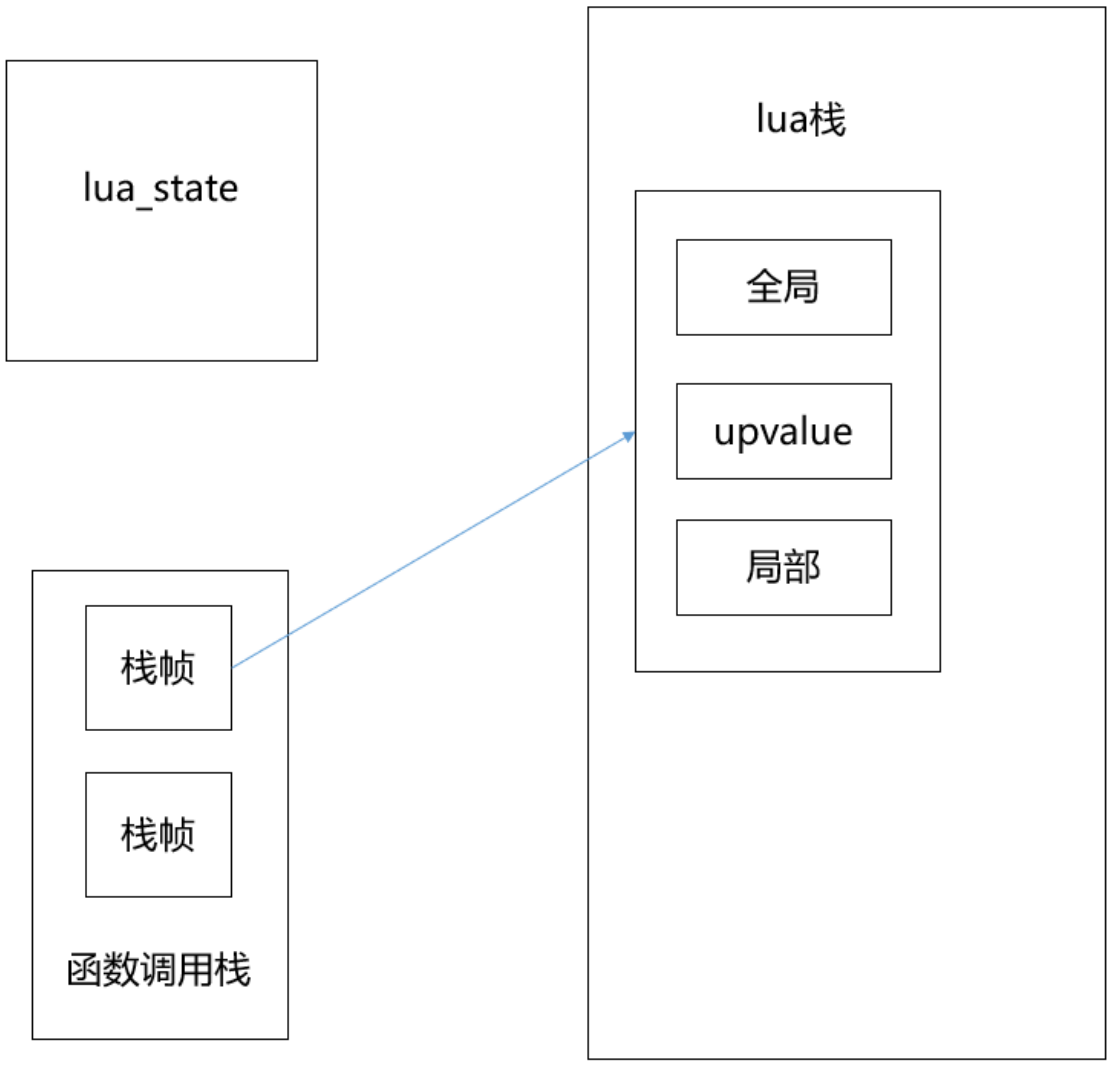
```lua
local function test()
    local cnt = 0
    return function () cnt = cnt + 1 return cnt end
end
test()()
test()()
test()()
c = test()
c()
c()
c()
--[[
main <test.lua:0,0> (20 instructions, 80 bytes at 0x55851c8f3870)
0+ params, 2 slots, 0 upvalues, 1 local, 1 constant, 1 function
    1   [4] CLOSURE     0 0 ; 0x55851c8f3ad0
    2   [5] MOVE        1 0
    3   [5] CALL        1 1 2
    4   [5] CALL        1 1 1
    5   [6] MOVE        1 0
    6   [6] CALL        1 1 2
    7   [6] CALL        1 1 1
    8   [7] MOVE        1 0
    9   [7] CALL        1 1 2
    10  [7] CALL        1 1 1
    11  [8] MOVE        1 0
    12  [8] CALL        1 1 2
    13  [8] SETGLOBAL   1 -1    ; c
    14  [9] GETGLOBAL   1 -1    ; c
    15  [9] CALL        1 1 1
    16  [10]    GETGLOBAL   1 -1    ; c
    17  [10]    CALL        1 1 1
    18  [11]    GETGLOBAL   1 -1    ; c
    19  [11]    CALL        1 1 1
    20  [11]    RETURN      0 1
```

```
constants (1) for 0x55851c8f3870:
    1   "c"
locals (1) for 0x55851c8f3870:
    0   test    2   20
upvalues (0) for 0x55851c8f3870:

function <test.lua:1,4> (5 instructions, 20 bytes at 0x55851c8f3ad0)
0 params, 2 slots, 0 upvalues, 1 local, 1 constant, 1 function
    1   [2] LOADK       0 -1    ; 0
    2   [3] CLOSURE     1 0 ; 0x55851c8f3ee0
    3   [3] MOVE        0 0
    4   [3] RETURN      1 2
    5   [4] RETURN      0 1
constants (1) for 0x55851c8f3ad0:
    1   0
locals (1) for 0x55851c8f3ad0:
    0   cnt 2   5
upvalues (0) for 0x55851c8f3ad0:

function <test.lua:3,3> (6 instructions, 24 bytes at 0x55851c8f3ee0)
0 params, 2 slots, 1 upvalue, 0 locals, 1 constant, 0 functions
    1   [3] GETUPVAL    0 0 ; cnt
    2   [3] ADD         0 0 -1  ; - 1
    3   [3] SETUPVAL    0 0 ; cnt
    4   [3] GETUPVAL    0 0 ; cnt
    5   [3] RETURN      0 2
    6   [3] RETURN      0 1
constants (1) for 0x55851c8f3ee0:
    1   1
locals (0) for 0x55851c8f3ee0:
upvalues (1) for 0x55851c8f3ee0:
    0   cnt
]]
```

当定义函数的时候，`lua`栈上面会分配空间来存储该函数的闭包。闭包中存储的内容主要是该闭包所用的指令、全局变量、`upvalue`和局部变量列表。当函数调用的时候，虚拟机会执行加载闭包的指令，同时将这个闭包的引用作为函数栈帧加入到函数调用栈当中。新的函数栈帧一旦加入函数调用栈以后，会采用数据结构栈的方式来管理。当函数调用结束以后，函数栈帧会释放。但是闭包的内存释放是由垃圾回收系统控制的。



Lua的尾调用优化

如果某个函数的最后一步是调用另一个函数，那么就称为**尾调用**，如果这是一个递归调用，就称为**尾递归**。在Lua当中，函数的调用会导致函数调用栈当中分配栈帧，这样就限制函数调用的最大深度。但是如果采用了尾调用，主调函数执行完成除最后一步以外所有指令以后才会执行被调函数，那么当执行被调函数时，主调函数的栈帧就毫无价值了，此时就可以采用尾调用优化，就是在函数调用栈当中，直接用被调函数的栈帧取代主调函数的栈帧。如果函数调用都是尾调用的话，那么其深度的大小不受限制。

```
function f(i)
    print(i)
    return f(i+1)
    -- return (f(i+1)) 这种写法不是尾调用，所以调用深度会有限制
end
f(1)
```

模式匹配

`lua`当中的模式匹配规则和`POSIX`正则表达式并不一致。`lua`的模式匹配实现了绝大多数匹配规则，同时它的实现代码的规模也非常精简。和模式匹配相关的函数有4个，分别是：`string.gsub`、`string.find`、`string.match`和`string.gmatch`。

模式规则之基本单位

`lua`的**模式**就是一个普通的字符串，其功能和正则表达式差不多，但是规则上稍微有一些区别。在模式中，匹配的基本单位依然是单个字符，如果单字符不是控制字符的话，就可以匹配该字符本身。`lua`使用`%`作为转义字符（和C语言当中的`printf/scanf`一致）。转义字符组合上字母可以匹配某个集合中的字符，而转义字符组合上控制字符可以匹配该控制字符本身。基本单位连写时，就是将基本单位进行连接运算。

模式字符分类	对应含义
.	任一字符
%a	任一字母
%c	任一控制字符
%d	任一数字
%l	任一小写字符
%p	任一标点符号
%s	任一空白字符
%u	任一大写字符
%w	任一数字和数字
%x	任一16进制字符
[若干模式字符或是范围]	对各个模式字符/范围集合求并集，然后匹配任一集合内字符
[^集合]	匹配任一上述集合的补集内的字符

模式相关的函数

string.find

函数`string.find`用于在目标字符串中搜索指定的模式。如果查找成功，就返回两个值，分别是起始位置的索引和结束位置的索引，如果查找失败，则返回`nil`。

```
s = "hello world"
i, j = string.find(s, "hello")
print(i,j)
print(string.sub(s,i,j))
print(string.find(s,"world"))
i, j = string.find(s, "l")
print(i,j)
print(string.find(s, "lll"))
```

string.match

函数`string.match`也用于在一个字符串中搜索模式。如果查找成功，就返回与模式相匹配的那部分字串。

```
s = "12345678"
```

```
pat = "..78"
print(string.match(s,pat))
s = "hello\n\t123HELLO,'''123''''"
pat = "%c123"
print(string.match(s,pat))
pat = "%d"
print(string.match(s,pat))
pat = "%l"
print(string.match(s,pat))
pat = "%u"
print(string.match(s,pat))
pat = "%p%p"
print(string.match(s,pat))
pat = "%x"
print(string.match(s,pat))
pat = "%s"
print(string.match(s,pat))
pat = "[a-z]"
print(string.match(s,pat))
pat = "[^a-z%s]"
print(string.match(s,pat))
```

string.gsub

函数`string.gsub`至少有3个参数：目标字符串、模式和替换字符串，其第一个返回值是将目标字符串的所有符合模式的部分使用替换字符串替代后的字符串，第二个返回值是发生替换的次数。
`string.gsub`也可以传入第4个参数，用来限制替换的次数。

```
s = string.gsub("Luna is pretty","pretty","cute")
print(s)
s = string.gsub("all lii", "l", "x")
print(s)
s,n = string.gsub("Luna is pretty", "x", "b")
print(s,n)
s = string.gsub("all lii", "l", "x", "1")
print(s)
```

string.gmatch

函数`string.gmatch`返回一个函数，通过返回的函数可以遍历一个字符串中所有出现的指定模式。所以，可以将其作为泛型for的迭代器使用。

```
s = "how are you"
words = {}
for w in string.gmatch(s, "%a+") do
    words[#words + 1] = w
end
for k,v in ipairs(words) do
    print(v)
end
```

模式规则之魔法字符

`lua`把模式中用来控制基本单位的重复情况称为魔法字符。魔法字符可以组合上基本单位，从而控制基本单位的重复情况或者是其他规则。魔法字符有这几个:: `( ) . % + - * ? [ ] ^ $`

魔法字符	含义
+	重复1到多次
*	重复0到多次
-	重复0到多次(最小匹配)
?	可选(出现0次或者1次)
^	出现在开头
$	出现在结尾
%	转义字符

下面是示例:

```
-- +表示1到多次
print(string.gsub("one, and two; and three", "%a+", "word"))
print(string.match("the number 1298 is even", "%d+"))
-- */-表示0到多次
-- %表示转义字符
print(string.match("f((    ))", "%(%s*%)"))
print(string.match("/* hello*//* world */", "/%*.*%*/")) -- *匹配最长序列
print(string.match("/* hello*//* world */", "/%*.-%*/")) -- -匹配最短序列
-- ?表示0到1次
print(string.match("+128", "[+-]?%d+"))
print(string.match("128", "[+-]?%d+"))
print(string.match("-128", "[+-]?%d+"))
-- ^表示开头
print(string.match("abc", "^%d%a+"))
print(string.match("1abc", "^%d%a+"))
-- $表示结尾
print(string.match("abc", "%a+%d"))
print(string.match("abc1", "%a+%d"))
```

%b匹配成对字符串

使用`%bxy`，这里`x`和`y`是两个不同的字符，模式可以匹配从`x`到`y`当中的所有字符。这种匹配是平衡的，它并不是匹配遇到的第一个y，而是匹配第一个满足x的个数和y的个数相等的情况，这样可以保证匹配嵌套括号时的准确性。

```
print(string.match("(a+b)(a-b) = a^2 - b^2", "%b()"))
```

模式规则之捕获机制

捕获机制允许将模式匹配的内容用于后续继续使用。使用圆括号可以进行捕获。对于具有捕获的模式，`string.match`会将所有捕获到的值作为单独的结果返回。

```
pair = "name = Anna"
key, value = string.match(pair,"(%a+)%s*=%s*(%a+)") -- match的返回值是各个捕获的模式
print(key,value)
date = "Today is 2020/9/25"
d,m,y = string.match(date,"(%d+)/(%d+)/(%d+)")
print(d,m,y)
```

在`%`后面添加数字可以用来指代捕获的结果（比如`%1`就表示第一个捕获的结果）。这种写法可实现比较复杂的匹配，也可以将其利用在`string.gsub`当中。特别地，`%0`表示整个匹配。

```
str = [[ then he said: "It's all right!" ]]
q,quotedPart = string.match(str,"(['\"])(.-)%1") -- 如果第一个捕获是", 那%1就也是"
print(q)
print(quotedPart)
print(string.gsub("hello Lua", "%a", "%0-%0")) -- 注意gsub会替换多次
print(string.gsub("hello Lua", "(%a+)(%s*)(%a+)", "%3%2%1"))
-- 使用gsub可将latex代码转换成xml
-- \command{some text} --> <command>some text</command>
s = [[ the \quote{task} is to \em{change} that. ]]
s = string.gsub(s, "\\(%a+){(.-)}","<%1>%2</%1>")
print(s)
```

gsub的高级用法

函数`string.gsub`的第3个参数可以是函数或者是列表。当第3个参数是函数时，`string.gsub`会在每次找到匹配的时候，将匹配的内容作为参数传入函数，并将返回值作为替换字符串。当第3个参数是表时，会将匹配项作为键，将该键对应的值作为替换字符串。如果函数返回值为nil或者是键没有对应的值或者为nil，那么gsub就不会改变匹配。

```
function expand(s)
    return (string.gsub(s, "$(%w+)", _G)) -- _G表示全局变量表
end
name = "Lua"
status = "great"
for k,v in pairs(_G) do
    print(k,v)
end
print(expand("$name is $status, isn't it"))
print(expand("$nodefine is $status, isn't it"))
```

```
function expand(s)
    --return (string.gsub(s, "$(%w+)", _G)) -- _G表示全局变量表
    return (string.gsub(s,"$(%w+)", function(n) return tostring(_G[n]) end))
end
for k,v in pairs(_G) do
    print(k,v)
end
print(expand("print = $print; a = $a"))
```

```
function toxml(s)
    s = string.gsub(s,"\\(%a+)(%b{})", function (tag, body) -- %1传入tag %2传入
body
        body = string.sub(body,2,-2) -- 去除括号
        body = toxml(body) -- 处理嵌套的命令
        return string.format("<%s>%s</%s>",tag,body,tag)
        end
    )
    return s
end

print(toxml("\\title{The \\bold{\\test{big}} example}"))
```

元表和元方法

在lua当中，某种类型的值，它所能支持的操作是有限的：比如数值类型的值可以使用加法，但是表类型的值就不能使用加法。使用**元表**可以修改某个值在遇到未知操作时的行为，这个行为就称为**元方法**。元表也是一个表，当值遇到未知操作时，比如对列表使用加法时，Lua会首先根据操作的类型获取键(加法对应的是`__add`)，然后根据键访问表中的元素，而这个元素就是一个函数，描述了列表所将要执行的操作，即元方法。

对于表类型和用户数据类型的值，每个值会拥有自己独立的元表；而对于其他类型的值，同一类型的值会共享一个元表。为了安全，Lua语言当中只能修改表类型的元表，其他类型的元表只能通过C代码或者某些库函数来修改。

使用`getmetatable`函数可以获取值的元表，默认情况下，字符串类型的元表是非nil值，而其他类型的元表都为nil。

```
var = 123
t = getmetatable(var)
print(t)
var = "123"
t = getmetatable(var)
print(t)
for k,v in pairs(t) do
    print(k,v)
end
```

使用`setmetable`函数可以修改一个表类型的值的元表。lua对元表使用没有任何限制，一个表可以是任意值的元表，多个表值可以共享一个元表，表值的元表可以是它本身。

通过元表，用户可以自定义元表的算术运算操作、比较操作、连接操作、取长度操作以及按键访问操作，当然，元表也可以定义垃圾回收相关操作。

算术运算的元方法

在Lua中可以使用模块机制在多个代码文件之间共享代码。模块的本质就是由函数和常量构成的列表。使用`require`函数可以导入模块，而且模块本身就是一个列表，那么自然就可以对它的元表进行一些操作。

首先，在设计模块的代码当中就设置好模块的元表，这样所有导入模块的数据结构都会共享一个元表。如果希望让模块支持加法运算，就需要将其元表的 `__add` 键修改为一个合适的函数（既然要支持加法运算，这个函数的参数就必须是两个）。

```
-- 一个用于集合的模块 set.lua
-- 集合用列表描述
-- 其中键代表元素
-- 值为true表示元素属于集合
local set = {}
local mt = {}
-- 从序列中将值存入集合
function set.new (l)
    local set = {}
    setmetatable(set,mt)
    for k,v in ipairs(l) do
        set[v] = true
    end
    return set
end
-- 并集
function set.union(a,b)
    local res = set.new({})
    for k in pairs(a) do
        res[k] = true
    end
    for k in pairs(b) do
        res[k] = true
    end
    return res
end
--  交集
function set.intersection(a,b)
    local res = {}
    for k in pairs(a) do
        if b[k] then
            res[k] = true
        end
    end
    return res
end
-- tostring 方便输入
function set.tostring(a)
    local l = {}
    for k in pairs(a) do
        l[#l+1] = tostring(k)
    end
    return "{" .. table.concat(l,", ") .. "}"
end

return set
```

```
local set = require("set")
local a = set.new({1,2,3})
local b = set.new({1,3,5})
print(set.tostring(a))
```

```lua
print(set.tostring(b))
print(set.tostring(set.intersection(a,b)))
local mt = getmetatable(a)
-- print(set.tostring(a+b))
mt.__add = set.union -- 类似C++重载+运算符
print("a+b",set.tostring(a+b))
mt.__mul = set.intersection
print("a*b",set.tostring(a*b))
-- 除了module中的函数，也可以使用自定义函数
mt.__unm = function (a)
    local res = set.new({})
    for k in pairs(a) do
        res[-k] = true
    end
    return res
end
print("-a",set.tostring(-a))
-- 因为a和8的元表不同，当遇见a+8时，首先查找左操作数的元表的对应键，当找不到左操作数元表的对应键时，才会找到右操作数元表
-- a的元表有__add键，但是对应的函数参数不支持数值，函数报错
print("a+8", set.tostring(a+8)) -- 这个语句会访问a的元表
print("8+a", set.tostring(8+a)) -- 8没有元表，所以这个语句依然访问a的元表
print("'hello'+a", set.tostring('hello'+a)) -- 这个语句同样会访问a的元表
for k,v in pairs(mt) do
    print(k,v)
end
```

常见元方法和对应的运算符

下面是一些常见的元方法：

键/元方法	对应的运算符
__add	+
__sub	-
__mul	*
__div	/
__mod	%
__pow	^
__unm	- 单目运算符，取相反数
__concat	..
__len	#
__eq	== ~=会被转换成为==取反
__lt	< >=会被转换成为<
__le	<= >会被转换成为<=
__index	table[key]
__newindex	table[key] = value
__call	()

关系运算符相关的元方法

在Lua5.1版本中，程序会将`a >= b`转换为`b <= a`，`a > b`转换成`b <=a`，`a ~= b`转换成`not(a == b)`。

```
local mt = {}
local function new(a)
    return setmetatable(a,mt)
end
-- 使用元表的技巧：创建一个new函数，使其返回setmetatable。这样使用new构造的列表会共享一个元表
mt.__lt = function (a,b)
    a_cnt = 0
    b_cnt = 0
    print("lt")
    for k,v in ipairs(a) do
        a_cnt = a_cnt + v
    end
    for k,v in ipairs(b) do
        b_cnt = b_cnt + v
    end
    return a_cnt < b_cnt
end
mt.__le = function (a,b)
    a_cnt = 0
    b_cnt = 0
    print("le")
```

```
    for k,v in ipairs(a) do
        a_cnt = a_cnt + v
    end
    for k,v in ipairs(b) do
        b_cnt = b_cnt + v
    end
    return a_cnt <= b_cnt
end
a = new {1,2,3}
b = new {3,2,1}
print(a>=b) -- b <= a
print(b>a) -- a > b
```

库函数的元方法

除了之前在表格中涉及的元方法，许多的库函数也使用了一些独特的元方法，例如库函数`tostring`就和`__tostring`元方法有关。在调用`tostring`时，如果这个对象的元表中存在`__tostring`元方法，则会优先执行该元方法。

```
local mt = {}
local function new(a)
    return setmetatable(a,mt)
end
mt.__tostring = function (a)
    str  = {}
    for k,v in pairs(a) do
        str[#str + 1] = tostring(v)
    end
    return table.concat(str," ")
end
a = new {"Harry", "Ron", "Hermione"}
print(a)
```

类似地，如果元表当中存在`__metatable`元方法，那么执行函数`getmetatable`和`setmetatable`时就会优先调用该元方法，这种操作经常用来创建一个无法被修改元表的列表。

index和newindex的示例

通过键来访问和修改表中的元素是列表最基本的操作。其中`__index`元方法和访问新的键有关，而`__newindex`元方法和修改新的键对应的值有关。下面的例子演示了只读表格的例子。

其中`__index`和`__newindex`的元方法中可以是表名或者函数名，具体实现的大致代码流程如下：

```
-- __index
function gettable_event (table, key)
    local h
    if type(table) == "table" then --当table是表
        local v = rawget(table, key) -- rawget不使用元方法获取值
        if v ~= nil then return v end -- table[key]存在则直接返回
        h = metatable(table).__index -- table[key]不存在，访问table元表的__index元方
法
        if h == nil then return nil end
    else
        h = metatable(table).__index --当table非表时，获取其元方法
```

```
        if h == nil then
            error(···)
        end
    end
    if type(h) == "function" then -- 检查元方法是否是函数
        return (h(table, key))      -- 为函数则直接调用
    else return h[key]            -- 否则就是按表访问
    end
end
-- __newindex
function settable_event (table, key, value) --逻辑和__index一致
    local h
    if type(table) == "table" then
        local v = rawget(table, key)
        if v ~= nil then rawset(table, key, value); return end -- 不使用元方法设置值
        h = metatable(table).__newindex
        if h == nil then rawset(table, key, value); return end
    else
        h = metatable(table).__newindex
        if h == nil then
            error(···)
        end
    end
    if type(h) == "function" then
        h(table, key,value)
    else h[key] = value
    end
end
```

所以如果`getmetatable(table1).__index == getmetatable(table2).__index`，那么就会出现访问`table1[key]`等价于访问`table2[key]`。

```
function readonlytable(table)
    return setmetatable({}, {
        __index = table, -- 元方法这里是一个表名，可以看成是返回表的__index元方法和table的__index元方法一致
        -- table[1] --> getmetatable(table).__index[1]
        __newindex = function(table, key, value)
            error("Attempt to modify read-only table")
        end,
        __metatable = false
    });
end
local t = readonlytable({1,2,3,4})
print(t[1])
t[1] = 2 -- 报错
```

（自学内容）元表综合应用之参数类型检查

接下来是一段比较复杂的代码，它实现了类似C++函数重载的功能。比如，`foo.string.number = string.rep`就声明了`foo(string arg1, number arg1)`，且其函数实现为`string.rep`，从而说明了函数对象`foo`的参数列表，然后就可以传入合适类型的参数给`foo`进行调用。

```
function overloaded()
```

```
    local fns = {} -- 键为参数类型列表，值为对应的函数
    local mt = {} -- 元表
    local function oerror()
        return error("Invalid argument types to overloaded function")
    end
    function mt.__call(oname, ...) -- call元方法使得列表可以像函数一样调用
        local arg = {...}
        local signature = {}
        for k,v in ipairs {...} do
            signature[k] = type(v)
        end
        signature = table.concat(signature, ",") -- 生成参数类型列表
        return (fns[signature] or oerror)(...)
    end

    -- foo.string.number = func 的访问过程
    -- 先使用foo.string的__index元方法访问foo.string对应的值tmp
    -- 再使用tmp的__newindex元方法来修改tmp.number对应的值
    -- 所以__index元方法返回值也会对应一个元表
    function mt.__index(oname,key) -- oname[key]
        local signature = {}
        local submt = {} -- 这个元表是所有的__index返回值所对应的元表
        function submt.__newindex (oname, key, value)
            signature[#signature+1] = key
            fns[table.concat(signature, ",")] = value
        end
        -- index返回一个表，这个表以submt作为元表
        function submt.__index (oname, key)
            signature[#signature+1] = key
            return setmetatable({}, submt)
        end

        return submt.__index(oname, key) --mt的__index就是调用submt的__index
    end

    function mt.__newindex(oname,key, value)
        fns[key] = value
    end

    return setmetatable({}, mt)
end
foo = overloaded()
function foo.number(n) -- 定义1
    return n^2
end
function foo.string(str) -- 定义1
    return foo(tonumber(str))
end
foo.string.number = string.rep -- 定义2

print(foo(6))
print(foo('4'))
print(foo('abc',4))
print(foo('123','abc'))
```

模块和包

模块

在之前的元表演示中，我们曾经使用过模块。模块的本质是由常量和函数构成的列表，通过`require`函数导入以后，可以实现代码复用和共同开发。使用模块的行为就和访问列表元素的行为一致，也和其他语言的命名空间的使用方法非常相似。

require函数

使用函数`require`可以加载模块。`lua`当中会有一个名为`package`的列表来管理模块加载信息。

当调用`require`函数的时候，首先，函数`require`会在表`package.loaded`中检查该模块是否已经加载过，如果已经加载过的，会直接使用上次加载的结果；否则，`require`函数会尝试加载模块：首先根据`package.path`获得搜索路径，再根据`require`函数的参数查找文件。如果能找到对应的`lua`文件，就会调用`loadfile`函数加载文件，并且返回一个函数或者列表，被称为**加载器**，其作用是在被调用时加载模块；如果查找不到对应的`lua`文件的话，`require`函数会搜索C标准库文件，然后返回一个C程序加载器。一旦获取了模块的加载器，`require`就可以使用模块名来加载模块。最后，`require`函数还会修改环境变量`package.loaded`。

```
-- 查看已经加载的模块
for k,v in pairs(package.loaded) do
    print(k,v)
end
-- 搜索路径
print(package.path)
-- C搜索路径
print(package.cpath)
```

编写lua模块

第一种创建模块的方法是，创建一个表，把所有的需要导出的函数和变量放入表中，最后导出这个表。

```
-- complex.lua
local M = {}
complex = M -- 使用全局变量来记录模块的名字
-- 也可以根据require函数的传入参数来确定模块名
--[[
    local modname = ...
    _G[modname] = M
]]
local function new(r,i)
    return {r=r,i=i}
end
M.new = new
M.i = new(0,1)
function M.tostring(c)
    return string.format("%g+%gi",c.r,c.i)
end
return M
-- 也可以写成
--[[
    package.loaded[modname] = M
]]
-- test.lua
local m = require("complex")
```

```
a = m.i
print(m.tostring(a))
```

第二种创建模块的方法是，把所有的函数和变量定义为局部变量，在最后构造返回的表。

```
local function new(r,i) return {r=r,i=i} end
local i = new(0,1)
local function tostring(c)
    return string.format("%g+%gi",c.r,c.i)
end
return {
    new = new,
    i = i,
    tostring = tostring
}
```

以上的两种方法存在一个问题，就是程序员会经常性地忘记`local`关键字。为此，`lua`当中引入了**函数环境**的概念。模块的代码块中会创建一个独占的表，这样所有的模块内函数和变量都可以放入表，此外，表的名字可以命令为模块名，并将其加入`package.loaded`里面，最后，使用`setfenv(1,table)`可以将表设置为本模块的环境。这样的话，模块的变量和函数的作用域默认就不是全局的而是在模块内的。

```
local modname = ...
local M = {}
_G[modname] = M
local _G = _G --保存之前的全局环境
package.loaded[modname] = M
setfenv(1,M) -- 将模块的环境设置为M
function new (r,i)
    return {r = r, i = i}
end
i = new(0,1)
function tostring(c)
    return _G.string.format("%d + %d i", c.r, c.i)
end
```

使用`module`函数可以将简化上述步骤：

```
module(..., package.seeall) -- package.seeall可以保存之前全局环境，如果不需要可以不添加
local function new(r,i) return {r=r,i=i} end
i = new(0,1)
function tostring(c)
    return string.format("%g+%gi",c.r,c.i)
end
```

子模块和包

lua支持具有层次结构的模块名。子模块使用`.`运算符来表示，比如mod.sub就表示模块mod.sub是模块mod的一个子模块。所谓的包就是一颗由模块和它的所有孩子所构成的树。如果需要加载mod.sub，require函数的使用并没有什么不同，但是在搜索文件路径时，.运算符会被替换为系统的目录分隔符，这样的话，如果需要加载一个包，那么包中所有模块需要放在同一个目录下。比如具有模块p、p.a、p.b可以存储在p目录下面的p/init.lua、p/a.lua和p/b.lua，而p目录只要放在一个合理的搜索路径下即可。

热更新的原理

从require的机制可以知道，require首先判断模块是否已经加载过；然后调用加载器来加载模块；最后修改环境变量。如果要实现Lua的热更新，就是需要让Lua重新加载某个模块。根据require的原理，就是要让Lua虚拟机认为这个模块重来没有加载过。

```
local m = require("mod")
for k,v in pairs(m._M) do
    print(k,v)
end
print(m.tostring(m.i))
for k,v in pairs(package.loaded) do
    print(k,v)
end
function require_ex( name )
    print(string.format("require_ex = %s", name))
    if package.loaded[name] then
        print(string.format("require_ex module[%s] reload", name))
    end
    package.loaded[name] = nil
    require(name)
end
io.read()
require_ex("mod")
print(m.tostring(m.i))
```

热更新注意事项

当启用热更新代码的时候，最好将热更新代码单独存储为一个文件，这样可以保证热更新代码不会和项目中其他程序混杂。当程序可以热更新的时候，通常可以采用信号的方式来通知进程执行热更新代码。

在热更新的时候，由于函数的更新可能会修改数据，所以对于全局变量数据，热更新的代码一定要进行保护，通常可以采用类似a = a or 0的方式来保护数据，这样可以避免热更新导致代码数据异常。

Lua中的面向对象

Lua中表的行为和面向对象编程模式中对象的概念很像：首先，表和对象一样，可以存储一些状态和行为；另外，每个表都可以通过`self`来访问表本身；还有就是，两个存储完全一致的键-值的表可以是两个表；最后，表的生命周期和创建函数无关。

对象、原型对象和构造函数

列表的成员函数可以通过`self`来访问列表本身，只需要把成员函数的第一个参数设置为`self`即可。另外Lua当中提供了冒号运算符，可以简化这种使用`self`的成员函数的定义操作（`table.func(self,..)`等价于`table:func(...)`）。

Lua原生是没有面向对象相关的语法结构和关键字的，所以要实现面向对象需要用户使用元表的方式来实现。由于Lua当中只有表一种数据结构，所以类和对象都用表来实现，因为Lua当中的类又被称为原型对象。那么在Lua当中，类和对象的关系本质上就是多个表（对象）共享一个表（原型对象/类）的相关代码。

为了使表的构造方法和一般的对象一致，首先需要将表的构造修改成常见的初始化写法。这里的解决方案是修改`__call`元方法。

```
local MyClass = {}
MyClass.__index = MyClass -- 定义MyClass.__index 实例化相关的语句
setmetatable(MyClass, {
    -- a(x) 等价于 a.new(x)
    __call = function (cls, ...) return cls.new(...) end
})
function MyClass.new(init) -- 实例化
    local self = setmetatable({}, MyClass) --根据MyClass.__index的定义 self.func -
-> MyClass.func 从而实现代码复用
    -- 也可以不直接写 local self = setmetatable({}, {__index = MyClass})
    self.value = init
    return self -- 返回值就是一个对象
end
-- function MyClass.set_value(self,newval)
function MyClass:set_value(newval)
    self.value = newval
end
function MyClass:get_value()
    return self.value
end
function MyClass:print() -- > MyClass.print(self)
    print(self)
    print(self:get_value())
end
local instance = MyClass(5)
instance:print()
-- instance:print() --> instance.print(instance) -->
getmetatable(instance).__index.print(instance)
instance:set_value(6)
instance:print()
```

总之，在Lua中如果需要实现从类到对象的实例化过程，一种典型的方法就是：在类中添加`__index`键，其对应的值就是类本身，然后在具体的构造函数当中，将类作为返回的对象的元表。

继承和重写

在Lua当中，一种继承的实现方案就是让子类的`__index`和父类保持一致，如果需要重写某个方法，直接在子类内部去修改该函数的定义就好了。

```
local BaseClass = {}
BaseClass.__index = BaseClass
setmetatable(BaseClass, {
```

```
    __call = function (cls, ...) --cls就是BaseClass
        local self = setmetatable({}, cls)
        self:new(...)
        return self
    end
})
function BaseClass:new(init)
    self.value = init
end
function BaseClass:set_value(newval)
    self.value = newval
end
function BaseClass:get_value()
    return self.value
end
function BaseClass:print()
    print(self:get_value())
end
local DerivedClass = {}
DerivedClass.__index = DerivedClass
setmetatable(DerivedClass, {
    __index = BaseClass, -- 继承的关键，让子类的__index和父类一致
    __call = function (cls, ...)
        local self = setmetatable({}, cls)
        self:new(...)
        return self
    end
})
function DerivedClass:new(init1, init2) --子类重写父类的方法
    BaseClass:new(init1) -- 显示指定使用父类的new方法
    self.value2 = init2
end
function DerivedClass:get_value()
    return self.value + self.value2
end
local i = DerivedClass(1, 2)
i:print()
```

按照上面这种实现方法，子类对象在调用其方法时，会首先在子类中查找实现，然后再到父类中查找实现，期间通过`getmetatable`的`__index`方法不断跳转（如果继承层次非常复杂，可能会导致函数调用栈溢出）。提升跳转速度的一种解决方案就是参考C++的实现，在子类中将父类的所有成员拷贝一份。

继承的接口设计

在工程实践当中，通常会把继承的过程封装成一个函数，以下是cocos的部分代码：

```
local function class(classname, super) -- classname是子类名 super是父类/超类
    local superType = type(super)
    local cls

    -- ... 省略的部分是处理C++继承的代码
        -- inherited from Lua Object
        if super then
            cls = {}
            setmetatable(cls, {__index = super})
            cls.super = super
        else
```

```
            cls = {ctor = function() end}
        end

        cls.__cname = classname
        cls.__ctype = 2 -- lua
        cls.__index = cls

        function cls.new(...)
            local instance = setmetatable({}, cls)
            instance.class = cls
            instance:ctor(...)
            return instance
        end
    end

    return cls
end
```

使用闭包实现继承

除了使用元表以外，利用闭包的特性也可以实现继承和重写，并且还可以区分private成员和public成员。

```
local function MyClass(init)
    -- self当中存储了public成员
    local self = {
        public_field = 0
    }
    -- private成员就是局部变量
    local private_field = init
    function self.get_value() -- 方法放置
        return self.public_field + private_field
    end
    function self.change_value()
        private_field = private_field + 1
    end
    -- 返回所有的public成员构成的列表
    return self
end
local i = MyClass(5) -- 所有的私有成员都是upvalue，公共成员是返回列表的元素
print(i.get_value()) --> 5
i.public_field = 3
i.change_value()
print(i.get_value()) --> 9
-- 继承的例子
local function BaseClass(init)
    local self = {}
    local private_field = init
    function self.get_value()
        return private_field
    end
    function self.change_value()
        private_field = private_field + 1
    end
    return self
end
local function DerivedClass(init, init2)
```

```
    local self = BaseClass(init)
    self.public_field = init2
    local private_field = init2 --这个private_field和父类的同名成员无关
    local base_get_value = self.get_value
    function self.get_value()
        return private_field + self.public_field + base_get_value()
    end
    return self
end

local i = DerivedClass(1, 2)
print(i.get_value()) --> 5
i.change_value() -- 子类对象使用父类的方法
print(i.get_value()) --> 6
```

使用闭包继承的优势：方便实现封装特性；访问upvalue比访问表成员更快；更快的调用方法速度。

使用表继承的优势：创建对象更快；代码共享程度高，节约内存；直接从表中获取方法更符合用户习惯。

多重继承

多重继承表示某个类是多个互不相同的类同时派生生成的。为了让一个子类继承自多个父类，当子类调用一个自身未定义的方法时，需要去从所有的父类当中查找该方法的定义。

```
local function search(k, plist) -- 从所有的父类当中找到合适的键
    for i = 1,#plist do
        local v = plist[i][k]
        if v then
            return v
        end
    end
end
function createClass(...)
    local child = {}
    local parents = {...}

    setmetatable(child, {
        __index = function(table, key)
            return search(key, parents)
        end
    })

    child.__index = child -- 实现实例化
    function child:new(a)
        local object = {}
        object.a = a
        setmetatable(object,child)
        return object
    end
    return child
end
local classA = {}
classA.a = 0
function classA:print()
    print(self.a)
```

```
end
-- 如果不通过classA来实例化对象，下面内容可以注释掉
-- classA.__index = classA
--[[function classA:new(a)
    local instance = {a = a}
    setmetatable(instance,classA)
    return instance
end]]

local classB = {}
classB.a = 0
function classB:change()
    self.a = self.a + 1
end
-- classB.__index = classB
--[[function classB:new(a)
    local instance = {a = a}
    setmetatable(instance,classB)
    return instance
end]]
local classC = createClass(classA, classB)
local objc = classC:new(1)
objc:change()
objc:change()
objc:change()
objc:print()
```

当`objc:change()`执行时，由于objc的`__index`元方法是是classC，而classC的`__index`元方法就是search函数，所以就会从所有的父类中找到名为change的键，然后执行对应的方法。当然，采用这样的方法时，每次访问方法都需要遍历所有的父类，因此访问效率会比较低，一种解决方案是在第一次访问方法以后，就将这个方法保存起来，从而减少下次访问所需的时间。

Lua和C的交互

Lua和C有两种交互方法，第一种方式中，C语言拥有控制权，Lua代码被当成是库，由C来调用。在这种情况下，C部分代码称为**应用代码**；第二种方式则相反，Lua语言拥有控制权，而C代码被当成是库，C部分代码称为**库代码**。无论是应用代码还是库代码，都应该使用**C API**和Lua进行交互。

`C API`是一个由C函数、常量和类型组成的集合。包括读写Lua全局变量的函数、调用Lua函数的函数、运行Lua程序段的函数以及注册C函数以供等。使用C API要按照C语言的编写规范来写。由于Lua的源代码使用是“干净的C”，也是就是C++和C的交集部分，所以和C++交互的方式和C是一致的。

Lua和C语言通信依赖于数据结构**虚拟栈**。`C API`调用会操作栈上的值；C和Lua之间相互的数据交换通过栈完成；栈可以保存中间结果。栈可以兼容C和Lua的两大差异：静态和动态类型的区别，垃圾回收和手动释放内存的区别。

从第一个例子说起

```
// 编译命令 gcc -o test  test.c ~/lua-5.1.5/src/liblua.a -lm
// -lm表示链接数学库
#include <func.h>
#include "lua.h" //基础函数 lua_*
#include "lauxlib.h" //辅助库 luaL_*
#include "lualib.h" //lua标准库
int main()
```

```
{
    char buff[256];
    int error;
    lua_State *L = luaL_newstate();
    luaL_openlibs(L);
    puts("hello this is test.c");
    while(fgets(buff,sizeof(buff), stdin) != NULL)
    {
        error = luaL_loadbuffer(L,buff,strlen(buff), "line")
            ||lua_pcall(L,0,0,0);
        if(error)
        {
            fprintf(stderr,"%s", lua_tostring(L,-1));
            lua_pop(L,1);
        }
    }
    lua_close(L);
    return 0;
}
```

`lua_State`类型的数据结构保存一个lua虚拟机运行的状态，使用`luaL_newstate`可以创建新状态，而使用`lua_close`可以关闭掉它。默认情况下，`luaL_newstate`使用了C库函数`realloc`来分配存储空间。

```
lua_State *lua_newstate (lua_Alloc f, void *ud);
//f是分配内存函数 ud是和内存分配有关的指针
lua_State *luaL_newstate (void);
//如果无需修改默认内存分配函数，一般都是用辅助函数来创建lua虚拟机
```

当创建一个新状态以后，状态中并没有包含任何一个预定义函数，所以如果需要使用库函数，需要调用`luaL_openlibs`来打开所有的标准库。

```
void luaL_openlibs (lua_State *L);
```

使用`luaL_loadbuffer`可以将字符串作为一个程序段进行编译。`luaL_load*`还有很多其他函数。

```
int lua_load (lua_State *L,
              lua_Reader reader,
              void *data,
              const char *chunkname);
//加载lua代码段（不会运行）
//reader函数来读取代码段 data是reader函数的参数
//chunkname是为代码段命名的名字
//lua_load函数会将代码段整体看成一个函数，再将这个函数压入栈顶
int luaL_loadbuffer (lua_State *L,
                     const char *buff,
                     size_t sz,
                     const char *name);
//将buff的内容作为代码加载
//sz是buff大小
//name是chunkname
int luaL_loadfile (lua_State *L, const char *filename);
//将filename的文件内容作为代码加载
//如果filename是NULL，则从stdin读取内容
int luaL_loadstring (lua_State *L, const char *s);
```

```
//将s的内容作为代码加载，和luaL_loadbuffer等价
```

然后使用`lua_pcall`来将lua代码切换到保护模式并调用代码。当出错时，会将报错存入虚拟栈当中。`lua_tostring`可以从虚拟栈当中读取Lua字符串，并且将其转换成C字符串。`lua_pop`可以删除虚拟栈当中的数据。

```
int lua_pcall (lua_State *L, int nargs, int nresults, int errfunc);
//在保护模式下调用函数
//这个函数的基本使用流程
//1 将被调用的函数压入栈顶
//2 将函数的参数按顺序压入栈顶（调用时参数会被弹出）
//3 调用函数结束以后，返回值压入栈顶
//4 如果调用过程发生报错，lua_pcall会捕获报错，然后将错误信息压入栈顶
//nargs是参数个数
//nresults是返回值个数 LUA_MULTIRET表示不限制个数
//errfunc是错误处理函数在栈中的下标 0表示默认错误处理函数
void lua_pop (lua_State *L, int n);
//弹栈，n表示弹出元素的个数
```

如果需要在C++代码当中使用Lua代码，可以使用`lua.hpp`来取代`lua.h`，其实`lua.hpp`的内容就是如下所示：

```
extern "C" {
#include "lua.h"
#include "lualib.h"
#include "lauxlib.h"
}
```

虚拟栈及其结构

如前所述，C和Lua通信主要面临两个问题：首先是动态类型和静态类型的之间的区别，然后是垃圾回收和手动管理内存的区别。Lua采用了**虚拟栈**来解决交换数据的问题。虚拟栈是Lua中的数据结构，它的内存分配是由Lua的垃圾回收机制管理的，栈中的元素是Lua值。

当需要将C中的值传入Lua时，就调用C函数将值压入栈当中，然后Lua会栈当中的数据；而需要在C代码里面获取Lua值时，Lua首先将值压入栈中，然后就调用C函数获取这个值。因为C是静态类型的，所以传值入栈和获取栈中值的函数会根据值的类型有多个。

虚拟栈遵循后进先出的法则，使用Lua函数永远只能修改操作栈的顶部，但是可以访问栈中的所有元素。虚拟栈的大小默认和LUA_MINSTACK有关（默认是20），使用`lua_checkstack`可以增加栈的大小。由于栈大小由用户自己控制，故溢出的风险也由用户承担，因为在代码当中应当注意主动将不需要的数据弹栈。

虚拟栈并没有严格限制元素的读取行为，`C API`可以使用下标来访问栈当中的元素。从1开始的正数下标表示了栈的**绝对下标**，负数的下标表示了**偏移下标**。当栈中有n个元素时，下标`1`或者是下标`-n`代表了第一个入栈的元素，下标`n`或者是`-1`代表了最后一个入栈的元素。因此，从`-n`到`n`的之间的下标是有效下标。使用**伪下标**可以访问一些栈外的元素，`LUA_GLOBALSINDEX`是用来指示全局变量的伪下标，`LUA_ENVIRONINDEX`是用来指示环境的伪下标。

下面是操作虚拟栈有关的函数：

```
int lua_gettop (lua_State *L);//获取栈大小
void lua_settop (lua_State *L, int index);//修改栈顶位置，销毁多余的元素
void lua_pushvalue (lua_State *L, int index);//拷贝栈中某个元素，压入栈顶
void lua_remove (lua_State *L, int index);//删除栈中某个元素，原位置上方的元素会向下平移
void lua_insert (lua_State *L, int index);//取出栈顶元素，插入栈中，原位置和其上方元素会向上平移
void lua_replace (lua_State *L, int index);//取出栈顶元素，插入栈中，覆盖原位置元素
void lua_pop (lua_State *L, int n);//从栈中弹出n个元素
```

使用`lua_push*`系列函数可以将lua值压入栈中。

```
void lua_pushboolean (lua_State *L, int b);
const char *lua_pushfstring (lua_State *L, const char *fmt, ...);
void lua_pushinteger (lua_State *L, lua_Integer n);
void lua_pushliteral (lua_State *L, const char *s);
void lua_pushnil (lua_State *L);
void lua_pushnumber (lua_State *L, lua_Number n);
int lua_pushthread (lua_State *L);
void lua_pushlstring(lua_State *L, const char *s, size_t len);//定长字符数组，不以'\0'结尾
void lua_pushstring(lua_State *L, const char *s);//C风格字符串
```

使用`lua_is*`系列函数可以检查lua值的数据类型，使用`lua_type`可以获取lua值的数据类型，通常和switch结构连用。

```
int lua_isboolean (lua_State *L, int index);
int lua_iscfunction (lua_State *L, int index);
int lua_isfunction (lua_State *L, int index);
int lua_isnil (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_isthread (lua_State *L, int index);
int lua_isuserdata (lua_State *L, int index);
int lua_type (lua_State *L, int index);
//返回值为下面的几种常量之一：LUA_TNIL, LUA_TNUMBER, LUA_TBOOLEAN, LUA_TSTRING, LUA_TTABLE, LUA_TFUNCTION, LUA_TUSERDATA, LUA_TTHREAD, LUA_TLIGHTUSERDATA.
```

使用`lua_to*`系列函数可以从栈当中获取值并转换成C的数据。

```
int lua_toboolean (lua_State *L, int index);
//nil和false返回0，其余返回1
lua_CFunction lua_tocfunction (lua_State *L, int index);
lua_Integer lua_tointeger (lua_State *L, int index);
const char *lua_tolstring (lua_State *L, int index, size_t *len);
//末尾额外添加'\0'，中间可能有'\0'
lua_Number lua_tonumber (lua_State *L, int index);
const void *lua_topointer (lua_State *L, int index);
const char *lua_tostring (lua_State *L, int index);
lua_State *lua_tothread (lua_State *L, int index);
void *lua_touserdata (lua_State *L, int index);
```

使用`lua_get*`可以获取虚拟机的运行环境和其他信息。

```
lua_Alloc lua_getallocf (lua_State *L, void **ud);
//获取内存分配函数的信息，ud是分配函数的参数
void lua_getfenv (lua_State *L, int index);
//将index所指的元素的环境列表压入栈顶
void lua_getfield (lua_State *L, int index, const char *k);
//把t[k]值压入栈顶，其中t是index所指元素，可能触发index相关事件
void lua_getglobal (lua_State *L, const char *name);
//把name对应的全局变量的值压入栈顶
//这个它的宏定义 #define lua_getglobal(L,s)  lua_getfield(L, LUA_GLOBALSINDEX, s)
int lua_getmetatable (lua_State *L, int index);
//把index所指元素的值的元表压入栈中。
void lua_gettable (lua_State *L, int index);
//k是当前栈顶元素，把k弹出栈，然后把t[k]值压入栈顶，其中t是index所指元素，可能触发index相关事件
```

通过上述的接口，可以实现检查栈信息的函数

```
void stackDump(lua_State *L)
{
    int size = lua_gettop(L);
    for(int i = 1; i <= size; ++i)
    {
        int type = lua_type(L,i);
        switch(type)
        {
        case LUA_TNIL:
            printf("idx:%d type:nil\n",i);
            break;
        case LUA_TNUMBER:
            printf("idx:%d type:number value:%g\n",
                   i, lua_tonumber(L,i));
            break;
        case LUA_TBOOLEAN:
            printf("idx:%d type:boolean value:%s\n",
                   i, lua_toboolean(L,i)?"true":"false");
            break;
        case LUA_TSTRING:
            printf("idx:%d type:string value:%s\n",
                   i, lua_tostring(L,i));
            break;
        case LUA_TTABLE:
            printf("idx:%d type:table\n",i);
            break;
        case LUA_TFUNCTION:
            printf("idx:%d type:function\n",i);
            break;
        }
    }
}
```

使用C语言运行Lua代码

如果使用lua代码中保存一些数据，C语言代码可以通过一定的方法读取这些数据。C代码首先需要创建一个虚拟机，然后通过`luaL_loadfile`接口将lua代码整体编译，并且将其main函数压入虚拟栈中，然后可以执行代码，然后可以利用`lua_get*`系列函数获取全局值。下面是一个使用C获取全局数值信息的示例：

```
//C代码
#include <func.h>
#include <lua.h>
#include <lauxlib.h>
#include <lualib.h>
lua_Integer getInteger(lua_State *L, const char *var);
int main(int argc, char **argv)
{
    ARGS_CHECK(argc,2);
    lua_State *L = luaL_newstate();
    int ret = luaL_loadfile(L,argv[1])||lua_pcall(L,0,0,0);
    //加载文件，然后执行lua代码
    if(ret)
    {
        fprintf(stderr,"%s\n", lua_tostring(L,-1));
        lua_pop(L,1);
    }
    lua_Integer year = getInteger(L,"year");
    lua_Integer mon = getInteger(L,"mon");
    lua_Integer day = getInteger(L,"day");
    printf("Today is  %ld/%ld/%ld\n", year,mon,day);
    lua_close(L);
}
lua_Integer getInteger(lua_State *L, const char *var)
{
   lua_getglobal(L,var);
   //将lua代码中var变量对应的值存入栈中
   lua_Integer res = lua_tointeger(L,-1);
   //从栈顶取出一个整数
   lua_pop(L, 1);
   return res;
}
```

```
year = 2020
mon = 10
day = 26
```

在使用C语言操作lua数据时，要特别注意的是lua的表值：

```
#include <func.h>
#include <lua.h>
#include <lauxlib.h>
#include <lualib.h>
lua_Integer getTable(lua_State *L, const char *var);
int main(int argc, char **argv)
{
    ARGS_CHECK(argc,2);
```

```
    lua_State *L = luaL_newstate();
    int ret = luaL_loadfile(L,argv[1])||lua_pcall(L,0,0,0);
    if(ret)
    {
        fprintf(stderr,"%s\n", lua_tostring(L,-1));
        lua_pop(L,1);
    }
    lua_getglobal(L,"date");
    //把表压入栈顶
    lua_Integer year = getTable(L,"year");
    lua_Integer mon = getTable(L,"month");
    lua_Integer day = getTable(L,"day");
    printf("Today is  %ld/%ld/%ld\n", year,mon,day);
    lua_close(L);
}
lua_Integer getTable(lua_State *L, const char *var)
{
    lua_pushstring(L,var);
    //把var压入栈顶
    lua_gettable(L,-2);
    //date在-2 var在栈顶 弹出var 把date[var]压入栈顶
    lua_Integer res = lua_tointeger(L,-1);
    lua_pop(L,1);
    //把date[var]弹栈
    return res;
}
```

```
-- lua代码
date = {
    year = 2020,
    month = 10,
    day = 27
}
```

除了直接运行lua文件中代码以外，还可以单独运行lua代码中的函数：

```
#include <func.h>
#include <lua.h>
#include <lauxlib.h>
#include <lualib.h>
void stackDump(lua_State *L);
void runfunc(lua_State *L,const char *funcName);
int main(int argc, char **argv)
{
    ARGS_CHECK(argc,2);
    lua_State *L = luaL_newstate();
    luaL_openlibs(L);
    int ret = luaL_loadfile(L,argv[1])||lua_pcall(L,0,0,0);
    if(ret)
    {
        fprintf(stderr,"%s\n", lua_tostring(L,-1));
        lua_pop(L,1);
    }
    runfunc(L,"func");
    lua_close(L);
}
```

```c
void runfunc(lua_State *L,const char *funcName)
{
    lua_getglobal(L,funcName);
    lua_pushnumber(L,1);
    lua_pushnumber(L,3);
    lua_pushnumber(L,5);
    //stackDump(L);
    //将函数压入栈底 再依次压入参数
    int ret = lua_pcall(L,3,2,0);
    if(ret)
    {
        fprintf(stderr,"%s\n", lua_tostring(L,-1));
        lua_pop(L,1);
    }
    printf("res1 = %g, res2 = %g\n", lua_tonumber(L,1),lua_tonumber(L,2));
    lua_pop(L,2);
}
```

```lua
-- lua代码
function func(...)
    cnt = 0
    for k,v in ipairs({...}) do
        --print(k,v)
        cnt = cnt + v
    end
    return cnt, #{...}
end
```

在Lua当中调用C代码

lua代码当中也可以调用C语言函数。C函数在调用之前，必须先**注册**，并将其地址传递给Lua代码。在调用的时候，lua代码会维护一个虚拟栈，这个虚拟栈的结构和之前C代码中使用lua的虚拟栈在结构上是一致的，其功能是传递参数和返回值。值得注意的是，虚拟栈是局部的，每次调用C函数时，所使用的虚拟栈是本次调用**独有**的，**不能共享**，也就是说，如果lua调用C函数，这个过程中使用了虚拟栈，而这个C函数也使用了Lua代码，那么，对于lua调用C和C调用lua的过程中，各自的虚拟栈是无关的。

如果想要C函数能被lua代码调用，其设计应按照如下规范：

- 在lua传递参数给C函数时，会将参数按次序将参数压入栈中，第一个参数会栈底，而最后一个参数会在栈底。
- 在函数调用的过程中，栈中存储的参数会首先被清空。
- 如果要返回值给lua，C函数只需要依次将要返回的值压入栈中即可，依旧是第一个返回值在栈底，最后一个返回值在栈顶。
- C函数的`return`应当返回参数的个数。

C函数的定义如下：

```c
typedef int (*lua_CFunction)(lua_State *L);
//显然C函数的参数只能是lua_State*类型，返回值必须是int类型
```

使用`lua_pushcfunction`或者是`lua_pushcclosure`可以注册一个C函数：

```
void lua_pushcclosure (lua_State *L, lua_CFunction fn, int n);
//n表示闭包中数据的个数
void lua_pushcfunction (lua_State *L, lua_CFunction f);
//本质是#define lua_pushcfunction(L,f)  lua_pushcclosure(L,f,0)
```

在注册C函数的时候，会创建一个闭包，从而C函数可以保存一些数据，无论在什么时候C函数被调用，这些数据都能被访问。如果想要和C函数交互，可以首先先将数据压入栈中，再调用`lua_pushcclosure`来创建闭包，然后清空栈中参数，并将闭包压入栈中。

一旦注册完成以后，lua代码就可以像调用普通函数来调用C函数。

修改虚拟机属性调用C代码

在创建虚拟机以后，首先需要使用`lua_pushcfunction`注册C函数，并将C函数压入虚拟栈当中，然后使用`lua_setglobal`来将函数名设置为虚拟机当中的全局变量。

```
#include <func.h>
#include <lua.h>
#include <lauxlib.h>
#include <lualib.h>
static int C_read(lua_State *L);//所有的cfunction参数和返回值皆是如此
void runfunc(lua_State *L, const char *funcname, const char *pipename);
int main(int argc, char *argv[])
{
    ARGS_CHECK(argc,3);
    lua_State *L = luaL_newstate();
    luaL_openlibs(L);
    //将C函数压入虚拟栈，传递给lua代码
    lua_pushcfunction(L,C_read);
    //将
    lua_setglobal(L,"C_read");
    int ret = luaL_loadfile(L,argv[1])||lua_pcall(L,0,0,0);
    if(ret)
    {
        fprintf(stderr,"%s\n", lua_tostring(L,-1));
        lua_pop(L,1);
    }
    runfunc(L,"lua_read",argv[2]);
    lua_close(L);
    return 0;
}
void runfunc(lua_State *L, const char *funcname, const char *pipename)
{
    lua_getglobal(L,funcname);
    lua_pushstring(L,pipename);
    int ret = lua_pcall(L,1,0,0);
    if(ret)
    {
        fprintf(stderr,"%s\n", lua_tostring(L,-1));
        lua_pop(L,1);
    }
}
static int C_read(lua_State *L)
{
    const char * filename = lua_tostring(L,1);
    int fd = open(filename,O_RDONLY);
```

```
    char str[128] = {0};
    read(fd,str,sizeof(str));
    lua_pushstring(L,str);
    return 1;
}
```

将C代码打包成库再调用

采用上一种方法就不可避免地需要去修改虚拟机启动时的代码。并且还牵扯了至少两次跨语言的调用：首先是启动虚拟机相关，此时宿主语言是C，脚本语言是lua，然后脚本语言lua在运行的过程中又调用C语言的库代码。另一种方法实现lua中调用C函数就更加自然一些，将C代码打包成库文件，再由lua代码调用这些C库函数。

库函数的定义第一种方案并无区别：

```
static int C_read(lua_State *L)
{
    const char * filename = lua_tostring(L,1);
    int fd = open(filename,O_RDONLY);
    char str[128] = {0};
    read(fd,str,sizeof(str));
    lua_pushstring(L,str);
    return 1;
}
```

然后需要定义一个数组，元素类型是`luaL_Reg`结构体，`luaL_reg`结构体的成员分别是函数名和函数指针。

```
typedef struct luaL_Reg {
  const char *name;
  lua_CFunction func;
} luaL_Reg;
```

然后可以定义一个主函数，主函数中会调用`luaL_register`函数来注册C函数。

```
void luaL_register (lua_State *L,
                    const char *libname,
                    const luaL_Reg *l);
//当libname为NULL时，所有l中的函数会被注册，然后存入一个列表中，最后，此列表会被压入栈顶
//当libname为非NULL时，luaL_register会创建一个新表t，使t为全局变量libname和
package.loaded[libname]的值，然后注册所有l中的函数。如果表t已经存在，则会重用表t。最后表t会
被压入栈顶
```

以下是一个库函数编写的示例：

```
#include <func.h>
#include <lua.h>
#include <lauxlib.h>
#include <lualib.h>
static int C_read(lua_State *L)
{
    const char * filename = lua_tostring(L,1);
    int fd = open(filename,O_RDONLY);
    char str[128] = {0};
```

```
    read(fd,str,sizeof(str));
    lua_pushstring(L,str);
    return 1;
}
luaL_Reg myread[] ={
    {"C_read", C_read},//第一个成员是函数名字符串，第二个成员是函数名指针
    {NULL,NULL}//结尾元素
};
int luaopen_myread(lua_State *L)
//函数的名称必须是 luaopen_模块名
//这样require函数会搜索对应的库文件
{
    luaL_register(L,"myread",myread);
    return 1;
    //1表示列表所在的下标
}
```

将库函数打包成动态库，不过注意动态库名字不需要以lib开头：

```
$gcc -c myread.c -fpic
$gcc -shared myread.o -o myread.so
```

lua代码当中应该使用`require`函数来加载模块：

```
local mod = require("myread") -- 寻找luaopen_myread函数
local str = mod.C_read(arg[1])
print(str)
```