Lua 语言

一. Lua 基本语法

1. 单行注释

在 Lua 中使用两个减号来表示单行注释，即" --"。

PS：--这是一个单行注释

2. 多行注释

在 Lua 中使用减号与大括号的组合来实现，在使用多行注释时需要注意，多行注释是一个有始有终的组合" —[[]]--"，即有开始就一定有结束。（就像 HTML 的 一样）

PS：

--[[

这

是

一

个

多

行

注

释

--]]

3. 标示符

Lua 标示符用于定义一个变量，函数可以获取其他用户定义的项。标示符以一个字母 A 到 Z 或 a 到 z 或下划线_开头后加上 0 个或多个字母，下划线，数字（0 到 9）。

最好不要使用下划线加大写字母的标示符，因为 Lua 的保留字也是这样的。

Lua 不允许使用特殊字符如 @, $, 和 % 来定义标示符。Lua 是一个区分大小写的编程语言。因此在 Lua 中 Runoob 与 runoob 是两个不同的标示符。

以下列出了一些正确的标示符：

mohd zara abc move_name a_123

myname50 _temp j a23b9 retVal

4. 关键字

下面列出了 Lua 的保留关键字。保留关键字不能作为常量或变量或其他用户自定义标示符。

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

5. 全局变量

Lua 在默认的情况下，其变量总被认为是全局的（只要不加 local 关键字修饰的变量，都是全局变量）。

全局变量不需要声明，给一个变量赋值后即创建了这个全局变量，访问一个没有初始化

的全局变量也不会出错，只不过得到的结果是：nil。

PS：print(b)　结果为　nil

PS：

b=10　--这里定义了一个全局变量 b

print(b)　--输出所定义的全局变量 b

结果为 10

如果想删除一个全局变量，只需要将变量赋值为 nil。

PS：

b = nil

print(b)

结果为 nil

当我们将全局变量赋值为 nil 后，这个全局变量就好像从没被使用过一样。换句话说，当且仅当一个变量不等于 nil 时，这个变量即存在。

二. 数据类型

Lua 是动态类型语言，其变量并不需要定义数据类型，只需要为变量赋值。值可以存储在变量中，作为参数传递或结果返回。

Lua 语言中具有 8 个基本的数据类型，它们分别为：nil、boolean、number、string、userdata、function、thread 和 table。

数据类型	描述
nil	只有值 nil 属于该类，表示一个无效值（在条件表达式中相当于 false）。
boolean	包含两个值：false 和 true。
number	表示双精度类型的实浮点数
string	字符串由一对双引号或单引号来表示
userdata	表示任意存储在变量中的 C 数据结构
function	由 C 或 Lua 编写的函数
thread	表示执行的独立线路，用于执行协同程序
table	Lua 中的表（table）其实是一个"关联数组"（associative arrays），数组的索引可以是数字、字符串或表类型。在 Lua 里，table 的创建是通过"构造表达式"来完成，最简单的构造表达式是{}，用来创建一个空表。

我们可以使用 type 函数测试给定变量或者值的类型：

PS：

```
print(type("Hello world"))      --> string 类型
print(type(10.4*3))             --> number 类型
print(type(print))              --> function 类型
print(type(type))               --> function 类型
print(type(true))               --> boolean 类型
print(type(nil))                --> nil 类型
print(type(type(X)))            --> string 类型
```

1. 空-nil

nil 类型表示没有任何有效值，它只有一个值就是 nil，如果打印一个没有赋值的变量，

便会输出一个 nil 值：

PS：

```
print(type(a))
```

输出结果为 nil

对于全局变量和 table，nil 还有一个"删除"作用，给全局变量或者 table 表里的变量赋一个 nil 值，等同于把它们删掉。

PS：

```
tab1 = { key1 = "value_1", key2 = "value_2", "value_3" }
for key, element in pairs(tab1) do
    print(key .. " - " .. element)
end

tab1.key1 = nil
for key, element in pairs(tab1) do
    print(key .. " - " .. element)
end
```

1)nil 作比较时应该加上双引号"

PS：

```
type(x)     -- 输出 nil
type(x) == nil    -- 输出 false
type(x) == "nil"     --输出 true
```

type(x) == nil 的结果为 false 的原因是 type(x)实质是返回的 "nil" 字符串，是一个 string 类型。

print(type(type(x)))输出的结果为 string

1. 布尔-boolean

boolean 类型只有两个可选值：true（真）和 false（假），Lua 把 false 和 nil 看作是 false，其他的都为 true，数字 0 也是 true。

PS：

```
print(type(true))
print(type(false))
print(type(nil))

if false or nil
then
    print("至少有一个是 true")
else
    print("false 和 nil 都为 false")
end

if 0
then
    print("数字 0 是 true")
else
    print("数字 0 为 false")
```

end
上面的结果为
boolean
boolean
nil
false 和 nil 都为 false
数字 0 是 true

2. 数字-number

Lua 默认只有一种 number 类型，即 double（双精度）类型（默认类型可以修改 luaconf.h 里的定义），以下几种写法都被看作是 number 类型。

PS：

```
print(type(2))
print(type(2.2))
print(type(0.2))
print(type(2e+1))
print(type(0.2e-1))
print(type(7.8263692594256e-06))
```

3. 字符串-string

字符串由一对双引号或单引号来表示。

PS：

```
string1 = "this is string1"
string2 = 'this is string2'
```

也可以用两个方括号 "[[]]" 来表示"一块"字符串。

PS：

```
html = [[
<html>
<head></head>
<body>
    <a href="https://space.bilibili.com/7399222?spm_id_from=333.1007.0.0">教程
</a>
</body>
</html>
]]
print(html)
```

上面的内容输出的结果就为：

```
<html>
<head></head>
<body>
    <a href="https://space.bilibili.com/7399222?spm_id_from=333.1007.0.0">教程
</a>
</body>
</html>
```

1) 对数字字符串进行算术操作

在对一个数字字符串上进行算术操作时，Lua 会尝试将这个数字字符串转换成一个数字。

PS：

```
print("2" + 6)     --结果为 8.0
print("2" + "6")   --结果为 8.0
print("2 + 6")    --结果为 2 + 6，因为这就是一个完整的字符串
print("-2e2" * "6")   --结果为-1200.0
```

2)字符串连接

要将字符串连接起来，需要使用的是 ".."。

PS：print("a" .. 'b')   --结果为 ab

```
print(157 .. 428)   --结果为 157428
```

3)计算字符串长度

使用#来计算字符串的长度，#需要放在字符串的前面。

PS：

```
len = "www.runoob.com"
print(#len)
print(#"www.runoob.com")
```

上面的输出结果为 14，即字符的个数。

4. 表-table

在 Lua 里，table 的创建是通过"构造表达式"来完成，最简单构造表达式是{}，用来创建一个空表。也可以在表里添加一些数据，直接初始化表。

1) 创建一个空的 table

```
local tbl1 = {}
```

2)直接初始表

```
local tbl2 = {"apple", "pear", "orange", "grape"}
```

3)关联数组

Lua 中的表（table）其实是一个"关联数组"（associative arrays），数组的索引可以是数字或者是字符串，就相当于数据字典(dictionary)。

PS：

```
a = {}
a["key"] = "value"
key = 10
a[key] = 22
a[key] = a[key] + 11
for k, value in pairs(a)
do
    print(k .. " : " .. value)
end
```

上面的结果为 key : value   10 : 33

不同于其他语言的数组，会把 0 作为数组的初始索引，在 Lua 里表的默认初始索引一般以 1 开始。

```
local tbl = {"apple", "pear", "orange", "grape"}
```

```
for key, value in pairs(tbl)
do
    print("Key", key)
end
```

输出结果为

```
Key     1
Key     2
Key     3
Key     4
```

table 不会固定长度大小，有新数据添加时 table 长度会自动增长，没初始的 table 都是 nil。

PS：

```
a3 = {}
for i = 1, 10
do
    a3[i] = i
end
a3["key"] = "val"
print(a3["key"])
print(a3["none"])
```

从上面的代码可以知道，字符串"key"在 table 中所对应的值为"val"，而字符串"none"在 table 中并没有对应的值，因此它的输出结果为 nil。

5. 函数-function

在 Lua 中，函数是被看作是"第一类值（First-Class Value）"，函数可以存放在变量里。

PS：

```
function factorial1(n)
    if n == 0
    then
        return 1
    else
        return n * factorial1(n - 1)    --递归，最终结果为 n!（数 n 的阶乘）
    end
end
print(factorial1(5))
factorial2 = factorial1    --用全局变量 factorial2 去存储 factorial1 函数
print(factorial2(5))
```

上面代码的输出都是为 120。

1) 匿名函数

函数(function)可以以匿名函数（anonymous function）的方式通过参数传递。

PS：

```
function testFun(tab,fun)
    for key ,value in pairs(tab)
    do
        print(fun(key,value));
```

```
    end
end

tab={key1="val1",key2="val2"};
testFun(tab,
        function(key,value)   --这是一个匿名函数
            return key.."="..value;     -- ".."表示连接
        end
);
```

上面的执行结果为

```
key1 = val1
key2 = val2
```

6. 线程-thread

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

1) 线程跟协程的区别

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

7. 自定义类型-userdata

userdata 是一种用户自定义数据，用于表示一种由应用程序或 C/C++语言库所创建的类型，可以将任意 C/C++的任意数据类型的数据（通常是 struct 和指针）存储到 Lua 变量中调用。

三. Lua 的变量、赋值、索引

1. 变量

变量在使用前，需要在代码中进行声明，即创建该变量。

编译程序执行代码之前，编译器需要知道如何给语句变量开辟存储区，用于存储变量的值。

Lua 变量有三种类型：全局变量、局部变量、表中的域。

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

局部变量的作用域为从声明位置开始到所在语句块结束。

变量的默认值均为 nil。

PS：

```
a = 5                   -- 这是一个默认的全局变量
local b = 5             -- 这是我们显示声明的局部变量

function joke()
    c = 5               -- 这是全局变量
    local d = 6         -- 这是函数中的局部变量
end

joke()
print(c,d)              -- 输出结果为 5 nil，因为变量 d 是一个局部变量
```

```
do
    local a = 6        -- 局部变量
    b = 6              -- 对局部变量重新赋值
    print(a,b);        -- 这里的 a 使用的是这个域中的局部变量，输出结果为 6 6
end

print(a,b)         --这里使用的 a 是最初声明的全局变量 a ，输出结果为 5 6
```

2. 赋值语句

赋值是改变一个变量的值和改变表域的最基本的方法。

PS：

```
a = "hello" .. "world"
t.n = t.n + 1
```

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

PS： a, b = 10, 2*x -- 这条语句相当于 a=10; b=2*x

遇到赋值语句 Lua 会先计算右边所有的值然后再执行赋值操作，利用这一特性我们就可以交换变量的值。

PS：

```
x,y = 1,2   --为全局变量 x 和 y 赋值
x, y = y, x    --利用 lua 赋值特性来交换 x 和 y 的值
print(x,y)
```

在上面的代码中，我们利用 lua 的赋值特性来交换了 x 和 y 的值，因此最后的输出结果为 2 1。

当变量个数和值的个数不一致时，Lua 会一直以变量个数为基础采取不同的策略。

当 变量个数 > 值的个数　　按变量个数补足 nil

当 变量个数 < 值的个数　　多余的值会被忽略

PS：

```
a, b, c = 0, 1--因为值的数量小于参数的个数，因此后续的参数值用 nil 补足
print(a,b,c) --输出结果为 0   1   nil

a, b = a+1, b+1, b+2 --因为值的数量大于参数的个数，多余的值被忽略
print(a,b)   --输出结果为 1    2

a, b, c = 0--因为值的数量小于参数的个数，因此后续的参数值用 nil 补足
print(a,b,c)   --输出结果为 0    nil    nil
```

注意：如果要对多个变量赋值必须依次对每个变量赋值。

多值赋值经常用来交换变量，或将函数调用返回给变量。

3. 索引

对 table 的索引使用方括号[]。Lua 也提供了 . 操作。

```
t[i]
t.i                    -- 当索引为字符串类型时的一种简化写法
gettable_event(t,i) -- 采用索引访问本质上是一个类似这样的函数调用
```

PS：

```
site = {}  --创建一个空的 table，其相当于一个数据字典
site["key"] = "this" --为这个表赋值
print(site["key"])
print(site.key)
```

上面代码的最终输出都是 this，只是采用的索引方式不同。

四. 循环

很多情况下我们需要做一些有规律性的重复操作，因此在程序中就需要重复执行某些语句。一组被重复执行的语句称之为循环体，终止条件决定能否继续重复。

循环结构是在一定条件下反复执行某段程序的流程结构，被反复执行的程序被称为循环体。循环语句是由循环体及循环的终止条件两部分组成的。

1. 循环类型

1) while

Lua 编程语言中 while 循环语句在判断条件为 true 时会重复执行循环体语句，这和 C、C#、C++中的 while 循环一致。

```
while(condition)
do
   statements
end
```

statements(循环体语句) 可以是一条或多条语句，condition(条件) 可以是任意表达式，在 condition(条件) 为 true 时执行循环体语句。

PS：

```
a=10
while( a < 20 )
do
    print("a 的值为:", a)
    a = a+1
end
```

2)for

Lua 编程语言中 for 循环语句可以重复执行指定语句，重复次数可在 for 语句中控制。

a. 数值 for 循环

```
for var=exp1,exp2,exp3
do
    <执行体>
end
```

上述表示 var 从 exp1 变化到 exp2，每次变化以 exp3 为步长递增 var，并执行一次 "执行体"。exp3 是可选的，如果不指定，默认为 1。

PS：

```
for i=1,3--没有设置可选参数 3，将默认每次执行完成后加 1
do
    print(i)
end
```

```
for i=10,1,-1--设置了可选参数，每次执行完成后数值减 1
do
    print(i)
end
```

b. 泛型 for 循环

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

Lua 编程语言中泛型 for 循环语法格式为：

```
a = {"one", "two", "three"}--这是一个直接初始表
for index, value in ipairs(a)
do
    print(index, value)
end
```

index 是数组索引值，value 是对应索引的数组元素值。ipairs 是 Lua 提供的一个迭代器函数，用来迭代数组。

PS：

```
days = {"Sunday","Monday","Tuesday","Wednesday","Thursday","Friday","Saturday"}
for index,value in ipairs(days)
do
    print(value)
end
```

上面代码，通过迭代器函数遍历数组的索引，来输出它所对应的 value 值。

3)repeat…until

Lua 编程语言中 repeat...until 循环语句不同于 for 和 while 循环，for 和 while 循环的条件语句在当前循环执行开始时判断，而 repeat...until 循环的条件语句在当前循环结束后判断。（repeat...until 相当于 do while）

Lua 编程语言中 repeat...until 循环语法格式:

```
repeat
   statements
until( condition )
```

我们注意到循环条件判断语句（condition）在循环体末尾部分，所以在条件进行判断前循环体都会执行一次。如果条件判断语句（condition）为 false，循环会重新开始执行，直到条件判断语句（condition）为 true 才会停止执行。

PS：

```
a = 10--定义一个全局变量
repeat
    print("a 的值为:", a)
    a = a + 1
until( a > 15 )--当 a>15 时，跳出循环
```

4)循环嵌套

Lua 编程语言中允许循环中嵌入循环。

a. for 循环嵌套语法格式

```
for init,max/min value, increment
do
    for init,max/min value, increment
    do
    statements
    end
statements
end
```

b. while 循环嵌套语法格式

```
while(condition)--condition 为 true 执行循环体
do
    while(condition)--condition 为 true 执行循环体
    do
        statements--循环体
    end
    statements--循环体
end
```

c. repeat...until 循环嵌套语法格式

```
repeat
    statements--循环体
    repeat
        statements--循环体
    until( condition )
until( condition )--当 condition 为 false 时执行循环体，为 true 时跳出循环
```

除了以上同类型循环嵌套外，我们还可以使用不同的循环类型来嵌套，如 for 循环体中嵌套 while 循环。

PS：

```
j =2   --这是一个全局变量
for i=2,10 --for 循环未使用第三个参数，默认每次加 1
do
    for j=2,(i/j) , 2
    do
        if(not(i%j))--求余
        then
```

```
            break
        end
        if(j > (i/j))
        then
            print("i 的值为：",i)
        end
    end
end
```

2. 循环控制语句

循环控制语句用于控制程序的流程， 以实现程序的各种结构方式。

1) break 语句

Lua 编程语言 break 语句插入在循环体中，用于退出当前循环或语句，并开始脚本执行紧接着的语句。

如果你使用循环嵌套，break 语句将停止最内层循环的执行，并开始执行的外层的循环语句。

PS：

```
a = 10--定义一个全局变量
while( a < 20 )--当 a<20 时，执行循环体内容
do
    print("a 的值为:", a)
    a=a+1;
    if( a > 15)
    then
        break--使用 break 语句终止循环
    end
end
```

上述代码执行 while 循环，在变量 a 小于 20 时输出 a 的值，并在 a 大于 15 时终止执行循环。

2)goto 语句

goto 语句允许将控制流程无条件地转到被标记的语句处。

语法格式如下：

```
goto Label
```

Label 的格式为：

```
:: Label ::
```

PS：

```
local a = 1   --这是一个局部变量
::label:: print("--- goto label ---")

a = a+1   --这是一个全局变量
if(a < 3)
then
    goto label     --当 a < 3 的时候跳转到标签 label
end
```

上述代码的最终输出会有两个--- goto label ---。

我们可以在 lable 中设置多个语句：

```
i = 0   --这是一个全局变量
::s1::
do
    print(i)
    i = i+1
end
if(i>3)
then
    os.exit()    -- i 大于 3 时退出
end
goto s1   --跳转到标签 s1 的位置
```

有了 goto 语句，我们就可以实现 continue 的功能。

PS：

```
for i=1, 3   --for 循环未使用第三个参数，将默认每次执行循环体后 i 加 1
do
    if (i <= 2)
    then
        print(i, "yes continue")
        goto continue   --跳转到 continue 标签
    end
    print(i, "no continue")
    ::continue::
    print([[i'm end]])
end
```

3. 无限循环

在循环体中如果终止条件永远为 true，循环语句就会永远执行下去。

PS：

```
while( true )
do
   print("循环将永远执行下去")
end
```

五. 流程控制

Lua 编程语言流程控制语句通过程序设定一个或多个条件语句来设定。在条件为 true 时执行指定程序代码，在条件为 false 时执行其他指定代码。

控制结构的条件表达式结果可以是任何值，Lua 认为 false 和 nil 为假，true 和非 nil 为真。

注意：Lua 中 0 为 true，而 C#中的 0 为假(false)。

PS：

```
if(0) --在 Lua 中，0 代表 true，这一点和 c 系列语言不一样
then
    print("0 为 true")
end
```

1. if 语句

if 语句由一个布尔表达式作为条件判断，其后紧跟其他语句组成。

语句语法格式如下：

```
if(布尔表达式)  --当布尔表达式结果为 true 时才执行条件语句
then
   --[ 在布尔表达式为 true 时执行的语句 --]
End
```

PS：

```
a = 10;  --定义了一个全局变量

if( a < 20 )  --使用 if 语句判断全局变量 a
then
     --[ if 条件为 true 时打印以下信息 --]
     print("a 小于 20" );
end
print("a 的值为:", a);
```

上述代码用于判断变量 a 的值是否小于 20，根据条件判断执行不同的输出。

2. if…else…语句

if 语句可以与 else 语句搭配使用，在 if 条件表达式为 false 时执行 else 语句代码块。

语句语法格式如下：

```
if(布尔表达式)
then
   --[ 布尔表达式为 true 时执行该语句块 --]
else
   --[ 布尔表达式为 false 时执行该语句块 --]
end
```

当布尔表达式为 true 时会 if 中的代码块会被执行，当布尔表达式为 false 时，else 的代码块会被执行。

Lua 认为 false 和 nil 为假，true 和非 nil 为真。

注意：Lua 中 0 为 true，而 C#中的 0 为假(false)。

PS：

```
a = 100;  --定义了一个全局变量
if( a < 20 )  --条件判断语句
then
     --[ if 条件为 true 时执行该语句块 --]
     print("a 小于 20" );
else
     --[ if 条件为 false 时执行该语句块 --]
     print("a 大于 20" );
end
print("a 的值为 :", a);
```

3. if...elseif...else 语句

if 语句可以与 elseif...else 语句搭配使用，在 if 条件表达式为 false 时执行

elseif...else 语句代码块，用于检测多个条件语句。

语句语法格式如下：

```
if( 布尔表达式 1)
then
--[ 在布尔表达式 1 为 true 时执行该语句块 --]

    elseif( 布尔表达式 2)
    then
    --[ 在布尔表达式 2 为 true 时执行该语句块 --]

        elseif( 布尔表达式 3)
        then
        --[ 在布尔表达式 3 为 true 时执行该语句块 --]
                else
                --[ 如果以上布尔表达式都不为 true 则执行该语句块 --]
end
```

PS：

```
a = 100   --这是一个全局变量
if( a == 10 )  --这是条件判断语句
then
    --[ 如果条件为 true 打印以下信息 --]
    print("a 的值为 10" )
    elseif( a == 20 )
    then
        --[ if else if 条件为 true 时打印以下信息 --]
        print("a 的值为 20" )
        elseif( a == 30 )
        then
            --[ if else if condition 条件为 true 时打印以下信息 --]
            print("a 的值为 30" )
            else
                --[ 以上条件语句没有一个为 true 时打印以下信息 --]
                print("没有匹配 a 的值" )
end
print("a 的真实值为: ", a )
```

上述代码对对变量 a 的值进行判断。

4. if 嵌套语句

if 语句允许嵌套，这就意味着你可以在一个 if 或 else if 语句中插入其他的 if 或 else if 语句。

语句语法格式如下：

```
if( 布尔表达式 1)
then
   --[ 布尔表达式 1 为 true 时执行该语句块 --]
   if(布尔表达式 2)
```

```
    then
        --[ 布尔表达式 2 为 true 时执行该语句块 --]
    end
end
```

我们可以用同样的方式嵌套 else if...else 语句。

PS：

```
--下面定义了两个全局变量
a = 100;
b = 200;

if( a == 100 )--条件判断语句
then
    --[ if 条件为 true 时执行以下 if 条件判断 --]
    if( b == 200 )
    then
        --[ if 条件为 true 时执行该语句块 --]
        print("a 的值为 100 b 的值为 200" );
    end
end
print("a 的值为 :", a );
print("b 的值为 :", b );
```

上述代码判断变量 a 和 b 的值，根据条件判断语句的结果输出不同的值。

六. 函数

在 Lua 中，函数是对语句和表达式进行抽象的主要方法。既可以用来处理一些特殊的工作，也可以用来计算一些值。

Lua 提供了许多的内建函数，你可以很方便的在程序中调用它们，如 print()函数可以将传入的参数打印在控制台上。

函数可以完成指定的任务，这种情况下函数作为调用语句使用；也可以计算并返回值，这种情况下函数作为赋值语句的表达式使用。

1. 函数定义

定义格式如下：

```
optional_function_scope function function_name( argument1, argument2, argument3..., argumentn)
    function_body   --函数体
    return result_params_comma_separated   --返回值
end
```

① optional_function_scope

该参数是可选的制定函数是全局函数还是局部函数，未设置该参数默认为全局函数，如果你需要设置函数为局部函数需要使用关键字 local。

② function_name

指定函数名称。

③ argument1, argument2, argument3..., argumentn
函数参数，多个参数以逗号隔开，函数也可以不带参数。

④ function_bod
函数体，函数中需要执行的代码语句块。

⑤ result_params_comma_separated
函数返回值，Lua 语言函数可以返回多个值，每个值以逗号隔开。

PS：

```
function max(num1, num2)--函数返回两个值的最大值
    if (num1 > num2)
    then
        result = num1;
    else
        result = num2;
    end

    return result;
end
-- 调用函数
print("两值比较最大值为 ",max(10,4))
print("两值比较最大值为 ",max(5,6))
```

上面代码定义了函数 max()，参数为 num1, num2，用于比较两值的大小，并返回最大值。

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

function add(num1,num2,functionPrint)
    result = num1 + num2
    functionPrint(result)-- 调用传递的函数参数
end
myprint(10)
-- myprint 函数作为参数传递
add(2,5,myprint)
```

在上述代码中，我们可以将函数作为参数传递给函数。

2. 多返回值函数

Lua 函数可以返回多个结果值，比如 string.find，其返回匹配串"开始和结束的下标"（如果不存在匹配串返回 nil）。

PS：

```
function maximum (a)--最大值
    local maxIndex = 1                  -- 最大值索引
    local maxValue = a[maxIndex]                -- 最大值
    for index,value in ipairs(a) do
```

```
        if value > maxValue
        then
            maxIndex = index
            maxValue = value
        end
    end
    return m, maxIndex
end

print(maximum({8,10,23,12,5}))
```

在上述代码中，我们的函数 maximum 返回了多个结果，即最大值和它所对应的索引。

3. 可变参数

Lua 函数可以接受可变数目的参数，和 C 语言类似，在函数参数列表中使用三点 ... 表示函数有可变的参数。

PS：

```
function add(...)
    local s = 0   --这是一个局部变量
    for i, v in ipairs{...} -- {...} 表示一个由所有变长参数构成的数组
    do
        s = s + v
    end
    return s
end
print(add(3,4,5,6,7))   --输出结果为 25
```

PS：

```
function average(...)--计算平均数
    result = 0   --定义一个全局变量 result
    local arg={...}      -- arg 为一个表，是局部变量
    for index,value in ipairs(arg)
    do
        result = result + value
    end
    print("总共传入 " .. #arg .. " 个数")
    return result/#arg
end

print("平均值为",average(10,5,3,4,5,6))
```

1) 获取可变参数的数量

可以通过 select("#",...) 来获取可变参数的数量。

PS：

```
function average(...)
    result = 0
    local arg={...}
    for index,value in ipairs(arg)
```

```
    do
        result = result + value
    end
    --使用 select("#",...) 来获取可变参数的数量
    print("总共传入 " .. select("#",...) .. " 个数")
    return result/select("#",...)
end

print("平均值为",average(10,5,3,4,5,6))
```

4. 固定参数加上可变参数

有时候我们可能需要几个固定参数加上可变参数，固定参数必须放在变长参数之前。

```
function fwrite(fmt, ...)  ---> 固定的参数 fmt
return io.write(string.format(fmt, ...))
end

fwrite("runoob\n")        --->fmt = "runoob", 没有变长参数。
fwrite("%d%d\n", 1, 2)    --->fmt = "%d%d", 变长参数为 1 和 2
```

通常在遍历变长参数的时候只需要使用 {···}，然而变长参数可能会包含一些 nil，那么就可以用 select 函数来访问变长参数了：select('#', ···) 或者 select(n, ···)。

select('#', ···) ：返回可变参数的长度。

select(n, ···) ：用于返回从起点 n 开始到结束位置的所有参数列表。

调用 select 时，必须传入一个固定实参 selector(选择开关) 和一系列变长参数。如果 selector 为数字 n，那么 select 返回参数列表中从索引 n 开始到结束位置的所有参数列表，否则只能为字符串#，这样 select 返回变长参数的总数。

PS：

```
function f(...)
    a = select(3,...)  -->从第三个位置开始，变量 a 对应右边变量列表的第一个参数
    print (a)
    print (select(3,...)) -->打印第 3 个位置后的所有列表参数
end
f(0,1,2,3,4,5) –调用函数 f()
```

PS：

```
do
    function foo(...)
        for i = 1, select('#', ...) do   -->获取参数总数
            local arg = select(i, ...); -->读取参数，arg 对应的是右边变量列表的第一个参数
            print("arg", arg);
        end
    end
    foo(1, 2, 3, 4);
end
```

5. 命名参数

Lua 的函数参数是和位置相关的，调用时实参会按顺序依次传给形参。有时候用名字指

定参数是很有用的，比如 rename 函数用来给一个文件重命名，有时候我们我们记不清命名前后两个参数的顺序了：

```
rename(old="temp.lua", new="temp1.lua")
```

上面这段代码是无效的，Lua 可以通过将所有的参数放在一个表中，把表作为函数的唯一参数来实现上面这段伪代码的功能。因为 Lua 语法支持函数调用时实参可以是表的构造。

根据这个想法我们重定义了 rename：

```
function rename (arg)
    return os.rename(arg.old, arg.new)
end
```

当函数的参数很多的时候，这种函数参数的传递方式很方便的。例如 GUI 库中创建窗体的函数有很多参数并且大部分参数是可选的，可以用下面这种方式：

```
w = Window
{
    x=0, y=0, width=300, height=200,
    title = "Lua", background="blue",
    border = true
}
function Window (options)
    -- check mandatory options
    if type(options.title) ~= "string"
    then
        error("no title")
    elseif type(options.width) ~= "number"
    then
        error("no width")
    elseif type(options.height) ~= "number"
    then
        error("no height")
    end
    -- everything else is optional
    _Window(options.title,
            options.x or 0, -- default value
            options.y or 0, -- default value
            options.width, options.height,
            options.background or "white", -- default
            options.border -- default is false (nil)
    )
end
```

七. 再论函数

1. 闭包

当一个函数内部嵌套另一个函数定义时，内部的函数体可以访问外部的函数的局部变量，这种特征我们称作词法定界。虽然这看起来很清楚，事实并非如此，词法定界加上第一类函数在编程语言里是一个功能强大的概念，很少语言提供这种支持。

下面看一个简单的例子，假定有一个学生姓名的列表和一个学生名和成绩对应的表；现

在想根据学生的成绩从高到低对学生进行排序。

PS：

```
names = {"Peter", "Paul", "Mary"}
grades = {Mary = 10, Paul = 7, Peter = 8}
table.sort(names, function (n1, n2)
    return grades[n1] > grades[n2]    -- compare the grades
end)
```

假定创建一个函数实现此功能。

PS：

```
function sortbygrade (names, grades)
    table.sort(names, function (n1, n2)
        return grades[n1]  >  grades[n2]    -- compare the grades
    end)
end
```

上述例子中，包含在 sortbygrade 函数内部的 sort，其匿名函数可以访问 sortbygrade 的参数 grades，在匿名函数内部 grades 不是全局变量也不是局部变量，我们称作外部的局部变量（external local variable）或者 upvalue。（upvalue 意思有些误导，然而在 Lua 中他的存在有历史的根源，还有他比起 external local variable 简短）。

PS：

```
function newCounter()
    local i = 0
    return function()   -- anonymous function
        i = i + 1
        return i
    end
end
c1 = newCounter()
print(c1())        --> 1
print(c1())        --> 2
```

匿名函数使用 upvalue i 保存他的计数，当我们调用匿名函数的时候 i 已经超出了作用范围，因为创建 i 的函数 newCounter 已经返回了。然而 Lua 用闭包的思想正确处理了这种情况。简单的说，闭包是一个函数以及它的 upvalues。如果我们再次调用 newCounter，将创建一个新的局部变量 i，因此我们得到了一个作用在新的变量 i 上的新闭包。

PS：

```
c2 = newCounter()
print(c2())        --> 1
print(c1())      --> 3
print(c2())      --> 2
```

c1、c2 是建立在同一个函数上，但作用在同一个局部变量的不同实例上的两个不同的闭包。

技术上来讲，闭包指值而不是指函数，函数仅仅是闭包的一个原型声明；尽管如此，在不会导致混淆的情况下我们继续使用术语函数代指闭包。

闭包在上下文环境中提供很有用的功能，如前面我们见到的可以作为高级函数（sort）的参数；作为函数嵌套的函数（newCounter）。这一机制使得我们可以在 Lua 的函数世界里

组合出奇幻的编程技术。闭包也可用在回调函数中，比如在 GUI 环境中你需要创建一系列 button，但用户按下 button 时回调函数被调用，可能不同的按钮被按下时需要处理的任务有点区别。具体来讲，一个十进制计算器需要 10 个相似的按钮，每个按钮对应一个数字，可以使用下面的函数创建他们。

PS：

```
function digitButton (digit)
    return Button{ label = digit,
        action = function ()
            add_to_display(digit)
        end
    }
end
```

上面的例子我们假定 Button 是一个用来创建新按钮的工具，label 是按钮的标签，action 是按钮被按下时调用的回调函数。（实际上是一个闭包，因为他访问 upvalue digit）。digitButton 完成任务返回后，局部变量 digit 超出范围，回调函数仍然可以被调用并且可以访问局部变量 digit。

闭包在完全不同的上下文中也是很有用途的。因为函数被存储在普通的变量内我们可以很方便的重定义或者预定义函数。通常当你需要原始函数有一个新的实现时可以重定义函数。例如你可以重定义 sin 使其接受一个度数而不是弧度作为参数。

PS：

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

这样我们把原始版本放在一个局部变量内，访问 sin 的唯一方式是通过新版本的函数。

利用同样的特征我们可以创建一个安全的环境（也称作沙箱，和 java 里的沙箱一样），当我们运行一段不信任的代码（比如我们运行网络服务器上获取的代码）时安全的环境是需要的，比如我们可以使用闭包重定义 io 库的 open 函数来限制程序打开的文件。

PS：

```
do
    local oldOpen = io.open
    io.open = function (filename, mode)
        if access_OK(filename, mode) then
            return oldOpen(filename, mode)
        else
            return nil, "access denied"
        end
```

```
    end
end
```

2. 非全局变量

Lua 中函数可以作为全局变量也可以作为局部变量，我们已经看到一些例子，比如函数作为 table 的域（大部分 Lua 标准库使用这种机制来实现的比如 io.read、math.sin）。这种情况下，必须注意函数和表语法。

1) 表和函数放在一起

PS：

```
Lib = {}
Lib.foo = function (x,y) return x + y end
Lib.goo = function (x,y) return x - y end
```

2)使用表构造函数

PS：

```
Lib = {
    foo = function (x,y) return x + y end,
    goo = function (x,y) return x - y end
}
```

3)Lua 提供另一种语法方式

PS：

```
Lib = {}
function Lib.foo (x,y)
    return x + y
end
function Lib.goo (x,y)
    return x - y
end
```

当我们将函数保存在一个局部变量内时，我们得到一个局部函数，也就是说局部函数像局部变量一样在一定范围内有效。这种定义在包中是非常有用的，因为 Lua 把 chunk 当作函数处理，在 chunk 内可以声明局部函数（仅仅在 chunk 内可见），词法定界保证了包内的其他函数可以调用此函数。（下面是声明局部函数的两种方式）

a. 方式一

PS：

```
local f = function (...)
    ...
end
local g = function (...)
    ...
    f() -- external local `f' is visible here
    ...
end
```

b. **方式二**

PS：

```
local function f (...)
    ...
End
```

有一点需要注意的是声明递归局部函数的方式。

PS：

```
local fact = function (n)
    if n == 0 then
        return 1
    else
        return n*fact(n-1) -- buggy
    end
end
```

上面这种方式导致 Lua 编译时遇到 fact(n-1)并不知道他是局部函数 fact，Lua 会去查找是否有这样的全局函数 fact。为了解决这个问题我们必须在定义函数以前先声明。

PS：

```
local fact
fact = function (n)
    if n == 0 then
        return 1
    else
        return n*fact(n-1)
    end
end
```

这样在 fact 内部 fact(n-1)调用是一个局部函数调用，运行时 fact 就可以获取正确的值了。

但是 Lua 扩展了他的语法，使得其可以直接在递归函数定义时使用两种方式。在定义非直接递归局部函数时要先声明然后定义才可以。

PS：

```
local f, g        -- `forward' declarations
function g ()
    ... f() ...
end
function f ()
    ... g() ...
end
```

3. 正确的尾调用

Lua 中函数的另一个有趣的特征是可以正确的处理尾调用（proper tail recursion，一些书使用术语“尾递归”，虽然并未涉及到递归的概念）。

尾调用是一种类似在函数结尾的 goto 调用，当函数最后一个动作是调用另外一个函数时，我们称这种调用尾调用。

PS：

```
function f(x)
    return g(x)
end
```

g 的调用是尾调用。

例子中 f 调用 g 后不会再做任何事情，这种情况下当被调用函数 g 结束时程序不需要返回到调用者 f；所以尾调用之后程序不需要在栈中保留关于调用者的任何信息。一些编译器比如 Lua 解释器利用这种特性在处理尾调用时不使用额外的栈，我们称这种语言支持正确的尾调用。

由于尾调用不需要使用栈空间，那么尾调用递归的层次可以无限制的。例如下面调用不论 n 为何值不会导致栈溢出。

PS：

```
function foo (n)
if n > 0 then return foo(n - 1) end
end
```

注意：必须明确什么是尾调用，一些调用者函数调用其他函数后，没有做其他的事情这不属于尾调用。

PS：

```
function f (x)
    g(x)
    return
end
```

上面这个例子中 f 在调用 g 后，不得不丢弃 g 地返回值，所以不是尾调用，同样的下面几个例子也不时尾调用。

PS：

```
return g(x) + 1      -- must do the addition
return x or g(x)      -- must adjust to 1 result
return (g(x))     -- must adjust to 1 result
```

Lua 中类似 return g(...)这种格式的调用是尾调用。但是 g 和 g 的参数都可以是复杂表达式，因为 Lua 会在调用之前计算表达式的值。

```
return x[i].foo(x[j] + a*b, i + j)
```

可以将尾调用理解成一种 goto，在状态机的编程领域尾调用是非常有用的。状态机的应用要求函数记住每一个状态，改变状态只需要 goto(or call)一个特定的函数。我们考虑一个迷宫游戏作为例子，迷宫有很多个房间，每个房间有东西南北四个门，每一步输入一个移动的方向，如果该方向存在即到达该方向对应的房间，否则程序打印警告信息。目标是：从开始的房间到达目的房间。

这个迷宫游戏是典型的状态机，每个当前的房间是一个状态。我们可以对每个房间写一个函数实现这个迷宫游戏，我们使用尾调用从一个房间移动到另外一个房间。一个四个房间的迷宫代码如下：

PS：

```
function room1 ()
    local move = io.read()
    if move == "south" then
        return room3()
```

```
        elseif move == "east" then
            return room2()
        else
            print("invalid move")
            return room1() -- stay in the same room
        end
    end

    function room2 ()
        local move = io.read()
        if move == "south" then
            return room4()
        elseif move == "west" then
            return room1()
        else
            print("invalid move")
            return room2()
        end
    end

    function room3 ()
        local move = io.read()
        if move == "north" then
            return room1()
        elseif move == "east" then
            return room4()
        else
            print("invalid move")
            return room3()
        end
    end

    function room4 ()
        print("congratilations!")
    end
```

我们可以调用 room1()开始这个游戏。

如果没有正确的尾调用，每次移动都要创建一个栈，多次移动后可能导致栈溢出。但正确的尾调用可以无限制的尾调用，因为每次尾调用只是一个 goto 到另外一个函数并不是传统的函数调用。

八. 运算符

1. 算术运算符

下面表格列出了 Lua 语言常用算术运算符，设定 A 的值为 10，B 的值为 20。

操作符	描述	示例
+	加法	A + B 输出结果 30

-	减法	A - B 输出结果 -10
*	乘法	A * B 输出结果 200
/	除法	B / A 输出结果 2
%	取余	B % A 输出结果 0
^	乘幂	A^2 输出结果 100
-	负号	-A 输出结果 -10
//	整除运算符（Lua 版本大于 5.3 可用）	5//2 输出结果 2

在 Lua 中，/ 用作除法运算，计算结果包含小数部分，// 用作整除运算，计算结果不包含小数部分。

```
a = 5
b = 2
print("除法运算 - a/b 的值为 ", a / b )
print("整除运算 - a//b 的值为 ", a // b )
```

上述代码运行结果为

```
除法运算 - a/b 的值为    2.5
整除运算 - a//b 的值为   2
```

在 Lua 里，按位异或不是^，而是~。

2. 关系运算符

下表列出了 Lua 语言中的常用关系运算符，设定 A 的值为 10，B 的值为 20。

操作符	描述	示例
==	等于，检测两个值是否相等，相等返回 true，否则返回 false	(A == B) 为 false。
~=	不等于，检测两个值是否相等，不相等返回 true，否则返回 false	(A ~= B) 为 true。
>	大于，如果左边的值大于右边的值，返回 true，否则返回 false	(A > B) 为 false。
<	小于，如果左边的值大于右边的值，返回 false，否则返回 true	(A < B) 为 true。
>=	大于等于，如果左边的值大于等于右边的值，返回 true，否则返回 false	(A >= B) 返回 false。
<=	小于等于， 如果左边的值小于等于右边的值，返回 true，否则返回 false	(A <= B) 返回 true。

这些操作符返回结果为 false 或者 true；==和~=比较两个值，如果两个值类型不同,Lua 认为两者不同；nil 只和自己相等。Lua 通过引用比较 tables、userdata、functions。

也就是说当且仅当两者表示同一个对象时相等。

3. 逻辑运算符

操作符	描述	示例
and	逻辑与操作符。 若 A 为 false，则返回 A，否则返回 B。	(A and B) 为 false。
or	逻辑或操作符。 若 A 为 true，则返回 A，否则返回 B。	(A or B) 为 true。
not	逻辑非操作符。与逻辑运算结果相反，如果条件为 true，逻辑非为 false。	not(A and B) 为 true。

4. 其他运算符

操作符	描述	示例
..	连接两个字符串	a..b ，其中 a 为 "Hello " ， b 为 "World", 输出结果为 "Hello World"。
#	一元运算符，返回字符串或表的长度	#"Hello" 返回 5

5. 运算符优先级

下表是优先级从高到低的顺序

^					
not	-				
*	/	%			
+	-				
..					
<	>	<=	>=	~=	==
And					
or					

除了^和..外所有的二元运算符都是左连接的

a+i < b/2+1	<——>	(a+i) < ((b/2)+1)
5+x^2*8	<——>	5+((x^2)*8)
a < y and y <= z	<——>	(a < y) and (y <= z)
-x^2	<——>	-(x^2)
x^y^z	<——>	x^(y^z)

九. Lua 字符串

字符串或串(String)是由数字、字母、下划线组成的一串字符。

Lua 语言中字符串可以使用以下三种方式来表示：

① 单引号间的一串字符。

② 双引号间的一串字符。

③ [[与]] 间的一串字符。

PS：

```
string1 = "Lua"
print("\"字符串 1 是\"",string1)
```

```
string2 = '666'
print("字符串 2 是",string2)

string3 = [["Lua 语言"]]
print("字符串 3 是",string3)
```

1. 转义字符

转义字符用于表示不能直接显示的字符，比如后退键，回车键，等。如在字符串转换双引号可以使用 "\""。

转义字符	意义	ASCII 码值（十进制）
\a	响铃(BEL)	007
\b	退格(BS) ，将当前位置移到前一列	008
\f	换页(FF)，将当前位置移到下页开头	012
\n	换行(LF) ，将当前位置移到下一行开头	010
\r	回车(CR) ，将当前位置移到本行开头	013
\t	水平制表(HT) （跳到下一个 TAB 位置）	009
\v	垂直制表(VT)	011
\\	代表一个反斜线字符"\'	092
\'	代表一个单引号（撇号）字符	039
\"	代表一个双引号字符	034
\0	空字符(NULL)	000
\ddd	1 到 3 位八进制数所代表的任意字符	三位八进制
\xhh	1 到 2 位十六进制所代表的任意字符	二位十六进制

2. 字符串操作

序号	方法&用途	示例
1	string.upper(argument): 字符串全部转为大写字母。	string1 = "Lua"; print(string.upper(string1)) 结果为：LUA
2	string.lower(argument): 字符串全部转为小写字母。	string1 = "Lua"; print(string.lower(string1)) 结果为：lua
3	string.gsub(mainString,findString,replaceString,num) 在字符串中替换。 mainString 为要操作的字符串， findString 为被替	string.gsub("aaaa","a","z",3);

	换的字符，replaceString 要替换的字符，num 替换次数（可以忽略，则全部替换）。	
4	string.find (str, substr, [init, [end]]) 在一个指定的目标字符串 str 中搜索指定的内容 substr，如果找到了一个匹配的子串，就会返回这个子串的起始索引和结束索引，不存在则返回 nil。	查找"Lua"的起始索引和结束索引位置 string.find("Hello Lua user", "Lua", 1) 结果为：7 9
5	string.reverse(arg) 将字符串反转	
6	string.format(...) 返回一个类似 printf 的格式化字符串	string.format("the value is:%d",4) 结果：the value is:4
7	string.char(arg) 和 string.byte(arg[,int]) char 将整型数字转成字符并连接，byte 转换字符为整数值(可以指定某个字符，默认第一个字符)。	
8	string.len(arg) 计算字符串长度。	string.len("abc") 结果为：3
9	string.rep(string, n) 返回字符串 string 的 n 个拷贝	string.rep("abcd",2) 结果为：abcdabcd
10	.. 链接两个字符串	print("www.runoob.."com") 结果为：www.runoob.com
11	string.gmatch(str, pattern) 返回一个迭代器函数，每一次调用这个函数，返回一个在字符串 str 找到的下一个符合 pattern 描述的子串。如果参数 pattern 描述的字符串没有找到，迭代函数返回 nil。	for word in string.gmatch("Hello Lua user", "%a+") do print(word) end 结果：Hello Lua user
12	string.match(str, pattern, init) string.match()只寻找源字串 str 中的第一个配对。参数 init 可选，指定搜寻过程的起点，默认为 1。 在成功配对时，函数将返回配对表达式中的所有捕获结果；如果没有设置捕获标记，则返回整个配对字符串．当没有成功的配对时，返回 nil。	print(string.match("I have 2 questions for you.", "%d+ %a+")) 结果为：2 questions print(string.format("%d, %q", string.match("I have 2 questions for you.", "(%d+) (%a+)"))) 结果为：, "questions"

3. 字符串截取

字符串截取使用 sub()方法，string.sub() 用于截取字符串。

string.sub(s, i [, j])

① s：要截取的字符串。

② i：截取开始位置。
③ j：截取结束位置，默认为 -1，最后一个字符。

```
-- 字符串
local sourcestr = "prefix--runoobgoogletaobao--suffix"
print("\n 原始字符串", string.format("%q", sourcestr))

-- 截取部分，第 4 个到第 15 个
local first_sub = string.sub(sourcestr, 4, 15)
print("\n 第一次截取", string.format("%q", first_sub))

-- 取字符串前缀，第 1 个到第 8 个
local second_sub = string.sub(sourcestr, 1, 8)
print("\n 第二次截取", string.format("%q", second_sub))

-- 截取最后 10 个
local third_sub = string.sub(sourcestr, -10)
print("\n 第三次截取", string.format("%q", third_sub))

-- 索引越界，输出原始字符串
local fourth_sub = string.sub(sourcestr, -100)
print("\n 第四次截取", string.format("%q", fourth_sub))
```

上述代码执行结果为：

```
原始字符串      "prefix--runoobgoogletaobao--suffix"
第一次截取      "fix--runoobg"
第二次截取      "prefix--"
第三次截取      "ao--suffix"
第四次截取      "prefix--runoobgoogletaobao--suffix"
```

4. 字符串格式化

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，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()。

格式字符串可能包含以下的转义码:

转义码	描述
%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
%x	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
%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。

PS：

```
--下面是定义的全局变量，分别是 string 和 number 类型
string1 = "Lua"
string2 = "Tutorial"
number1 = 10
number2 = 20
-- 基本字符串格式化
print(string.format("基本格式化 %s %s",string1,string2))--%s 接受一个字符串并按照给定的参数格式化该字符串
-- 日期格式化
date = 2; month = 1; year = 2014
print(string.format("日期格式化 %02d/%02d/%03d", date, month, year))--%d 接受一个数字并将其转化为有符号的整数格式
-- 十进制格式化
print(string.format("%.4f",1/3))--接受一个数字并将其转化为浮点数格式
```

5. 字符与整数相互转换

```
--byte 转换字符为整数值(可以指定某个字符，默认第一个字符)
print(string.byte("Lua"))          --结果为 76
-- 转换第三个字符
print(string.byte("Lua",3))          --结果为 97
-- 转换末尾第一个字符
print(string.byte("Lua",-1))          --结果为 97
-- 第二个字符
```

print(string.byte("Lua",2))　　--结果为 117

-- 转换末尾第二个字符

print(string.byte("Lua",-2))　　--结果为 117

-- 整数 ASCII 码转换为字符

print(string.char(97))--char 将整型数字转成字符并连接　　--结果为 a

6. 匹配模式

Lua 中的匹配模式直接用常规的字符串来描述。 它用于模式匹配函数 string.find, string.gmatch, string.gsub, string.match。

你还可以在模式串中使用字符类。

字符类指可以匹配一个特定字符集合内任何字符的模式项。比如，字符类 %d 匹配任意数字。所以你可以使用模式串 %d%d/%d%d/%d%d%d%d 搜索 dd/mm/yyyy 格式的日期：

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

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

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

下面列表列出了 Lua 支持的所有字符类。

单个字符(除 ^$()%.[]*+-? 外): 与该字符自身配对。

.（点）	与任何字符配对
%a	与任何字符配对
%c	与任何控制符配对(例如\n)
%d	与任何数字配对
%l	与任何小写字母配对
%p	与任何标点(punctuation)配对
%s	与空白字符配对
%u	与任何大写字母配对
%w	与任何字母/数字配对
%x	与任何十六进制数配对
%z	与任何代表 0 的字符配对
%x(此处 x 是非字母非数字字符)	与字符 x 配对. 主要用来处理表达式中有功能的字符(^$()%.[]*+-?)的配对问题, 例如%%与%配对
[数个字符类]	与任何[]中包含的字符类配对. 例如[%w_]与任何字母/数字, 或下划线符号(_)配对
[^数个字符类]	与任何不包含在[]中的字符类配对. 例如[^%s]与任何非空白字符配对

当上述的字符类用大写书写时, 表示与非此字符类的任何字符配对。例如, %S 表示与任何非空白字符配对。例如，'%A'非字母的字符。

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

结果为 hello..up.down.　　4；数字 4 不是字符串结果的一部分，他是 gsub 返回的第二个结果，代表发生替换的次数。

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。

十. Lua 数组

数组，就是相同数据类型的元素按一定顺序排列的集合，可以是一维数组和多维数组。

Lua 数组的索引键值可以使用整数表示，数组的大小不是固定的。

1. 一维数组

一维数组是最简单的数组，其逻辑结构是线性表。一维数组可以用 for 循环出数组中的元素。

PS：

```
array = {"Lua", "Tutorial"}     —这是一个直接初始表
for i= 0, 2
do
    print(array[i])
end
```

上述代码的输出结果为，nil；Lua；Tutorial。

正如我们所看到的，我们可以使用整数索引来访问数组元素，如果知道的索引没有值则返回 nil。

在 Lua 索引值是以 1 为起始，但你也可以指定 0 开始。

除此外我们还可以以负数为数组索引值。

PS：

```
array = {}--这是一个空表
for i= -2, 2    --该 for 循环没有设置第三个参数，因此将采用默认值 1
do
      array[i] = i *2
end
for i = -2,2 do
      print(array[i])
end
```

上述代码结果为-4，-2，0，2，4。

2. 多维数组

多维数组即数组中包含数组或一维数组的索引键对应一个数组。

1) 数组的数组

以下是一个三行三列的阵列多维数组（矩阵）：

PS：

```
-- 初始化数组
array = {}   --这是一个空表
for i=1,3    --该 for 循环没有设置第三个参数，因此将采用默认值 1
do
      array[i] = {}
      for j=1,3    --该 for 循环没有设置第三个参数，因此将采用默认值 1
      do
            array[i][j] = i*j
      end
end
```

```
-- 访问数组
for i=1,3   --该 for 循环没有设置第三个参数，因此将采用默认值 1
do
    for j=1,3
    do
        print(array[i][j])
    end
end
```

2)不同索引键的一维数组

表示矩阵的另一方法，是将行和列组合起来。如果索引下标都是整数，通过第一个索引乘于一个常量（列）再加上第二个索引。

PS：

```
-- 初始化数组
array = {}
maxRows = 3   --最大的行数
maxColumns = 3   --最大的列数
for row=1,maxRows
do
    for col=1,maxColumns
    do
        array[row*maxColumns +col] = row*col
    end
end

-- 访问数组
for row=1,maxRows
do
    for col=1,maxColumns
    do
        print(array[row*maxColumns +col])
    end
end
```

如果索引是字符串，可用一个单字符将两个字符串索引连接起来构成一个单一的索引下标，例如一个矩阵 m，索引下标为 s 和 t，假定 s 和 t 都不包含冒号，代码为 m[s..':'..t]，

如果 s 或者 t 包含冒号将导致混淆，比如("a:", "b") 和("a", ":b")，当对这种情况有疑问的时候可以使用控制字符来连接两个索引字符串，比如'\0'。

实际应用中常常使用稀疏矩阵，稀疏矩阵指矩阵的大部分元素都为空或者 0 的矩阵。

例如，我们通过图的邻接矩阵来存储图，也就是说：当 m,n 两个节点有连接时，矩阵的 m,n 值为对应的 x，否则为 nil。如果一个图有 10000 个节点，平均每个节点大约有 5 条边，为了存储这个图需要一个行列分别为 10000 的矩阵，总计 10000*10000 个元素，实际上大约只有 50000 个元素非空（每行有五列非空，与每个节点有五条边对应）。很多数据结构的书上讨论采用何种方式才能节省空间，但是在 Lua 中你不需要这些技术，因为用 table 实现的数据本身天生的就具有稀疏的特性。如果用我们上面说的第一种多维数组来表

示，需要 10000 个 table，每个 table 大约需要五个元素（table）；如果用第二种表示方法来表示，只需要一张大约 50000 个元素的表，不管用那种方式，你只需要存储那些非 nil 的元素。

十一. Lua 迭代器

迭代器（iterator）是一种对象，它能够用来遍历标准模板库容器中的部分或全部元素，每个迭代器对象代表容器中的确定的地址。

在 Lua 中迭代器是一种支持指针类型的结构，它可以遍历集合的每一个元素。

1. 泛型 for 迭代器

泛型 for 在自己内部保存迭代函数，实际上它保存三个值：迭代函数、状态常量、控制变量。

泛型 for 迭代器提供了集合的 key/value 对，语法格式如下：

```
for key, value in pairs(t)
do
    print(key, value)
end
```

上面代码中，key, value 为变量列表；pairs(t)为表达式列表。

PS：

```
array = {"Google", "Runoob"}   --这是一个直接初始 table
for key,value in ipairs(array)
do
    print(key, value)
end
```

以上示例中我们使用了 Lua 默认提供的迭代函数 ipairs。

下面我们看看泛型 for 的执行过程：

① 首先

初始化，计算 in 后面表达式的值，表达式应该返回泛型 for 需要的三个值：迭代函数、状态常量、控制变量；与多值赋值一样，如果表达式返回的结果个数不足三个会自动用 nil 补足，多出部分会被忽略。

② 第二

将状态常量和控制变量作为参数调用迭代函数（注意：对于 for 结构来说，状态常量没有用处，仅仅在初始化时获取他的值并传递给迭代函数）。

③ 第三

将迭代函数返回的值赋给变量列表。

④ 第四

如果返回的第一个值为 nil 循环结束，否则执行循环体。

⑤ 第五

回到第二步再次调用迭代函数

2. 无状态的迭代器

无状态的迭代器是指不保留任何状态的迭代器，因此在循环中我们可以利用无状态迭代器避免创建闭包花费额外的代价。

每一次迭代，迭代函数都是用两个变量（状态常量和控制变量）的值作为参数被调用，一个无状态的迭代器只利用这两个值可以获取下一个元素。

这种无状态迭代器的典型的简单的例子是 ipairs，它遍历数组的每一个元素，元素的索

引需要是数值。

PS：

```
function square(iteratorMaxCount,currentNumber)--iteratorMaxCount 表示迭代的总次数
    if (currentNumber<iteratorMaxCount)
    then
        currentNumber = currentNumber+1;
        return currentNumber, currentNumber*currentNumber
    end
end

for index,number in square,3,0--总的迭代次数为 3，从 0 开始
do
    print(index,number);
end
```

上述代码我们使用了一个简单的函数来实现迭代器，实现了数字 n 的平方。

迭代的状态包括被遍历的表（循环过程中不会改变的状态常量）和当前的索引下标（控制变量），ipairs 和迭代函数都很简单。

PS：

```
function iter (a, i)
    i = i + 1
    local value = a[i]
    if value then
        return i, value
    end
end

function ipairs (a)
    return iter, a, 0
end
```

当 Lua 调用 ipairs(a)开始循环时，他获取三个值：迭代函数 iter、状态常量 a、控制变量初始值 0；然后 Lua 调用 iter(a,0)返回 1, a[1]（除非 a[1]=nil）；第二次迭代调用 iter(a,1) 返回 2, a[2]……直到第一个 nil 元素。

3. 多状态的迭代器

很多情况下，迭代器需要保存多个状态信息而不是简单的状态常量和控制变量，最简单的方法是使用闭包，还有一种方法就是将所有的状态信息封装到 table 内，将 table 作为迭代器的状态常量，因为这种情况下可以将所有的信息存放在 table 内，所以迭代函数通常不需要第二个参数。

```
array = {"Google", "Runoob"}

function elementIterator (collection)
    local index = 0     --这是一个局部变量
    local count = #collection    --使用#来计算字符串的长度
    --  闭包函数
```

```
        return function ()
            index = index + 1
            if (index <= count)
            then
                --  返回迭代器的当前元素
                return collection[index]
            end
        end
    end

    for element in elementIterator(array)
    do
        print(element)
    end
```

上述代码，我们将所有的信息都存放在了 table 内，所以迭代函数不需要第二个参数。

十二. 表-table

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

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

Lua table 是不固定大小的，你可以根据自己需要进行扩容。

Lua 也是通过 table 来解决模块（module）、包（package）和对象（Object）的。 例如 string.format 表示使用"format"来索引 table string。

1. 表的构造

构造器是创建和初始化表的表达式。表是 Lua 特有的功能强大的东西。最简单的构造函数是{}，用来创建一个空表。

PS：

```
-- 初始化表
mytable = {}

-- 指定值
mytable[1]= "Lua"

-- 移除引用
mytable = nil
-- lua 垃圾回收会释放内存
```

对于 table，数据类型 nil 还有一个"删除"作用，给 table 表里的变量赋一个 nil 值，等同于把它们删掉。

当我们为 table a 设置元素，然后将 a 赋值给 b，则 a 与 b 都指向同一个内存（即引用的地址）。如果将 a 设置为 nil ，则 b 依然能访问 table 的元素。如果没有指定的变量指向 a，Lua 的垃圾回收机制会清理相对应的内存。

PS：

```
mytable = {}  -- 一个简单的 table
print("mytable 的类型是 ",type(mytable))  --使用 type 函数测试给定变量或者值的类型
```

```
mytable[1]= "Lua"
mytable["wow"] = "修改前"
print("mytable 索引为 1 的元素是 ", mytable[1])
print("mytable 索引为 wow 的元素是 ", mytable["wow"])

alternatetable = mytable  --指向同一个内存（即引用的地址）

print("alternatetable 索引为 1 的元素是 ", alternatetable[1])
print("mytable 索引为 wow 的元素是 ", alternatetable["wow"])

alternatetable["wow"] = "修改后"

print("mytable 索引为 wow 的元素是 ", mytable["wow"])

alternatetable = nil  -- 给 table 表里的变量赋一个 nil 值，等同于把它们删掉
print("alternatetable 是 ", alternatetable)

-- mytable 仍然可以访问
print("mytable 索引为 wow 的元素是 ", mytable["wow"])

mytable = nil
print("mytable 是 ", mytable)
```

2. Table 操作

以下列出了 Table 操作常用的方法：

序号	方法&用途
1	table.concat (table [, sep [, start [, end]]]): concat 是 concatenate(连锁，连接)的缩写。 table.concat()函数列出参数中指定 table 的数组部分从 start 位置到 end 位置的所有元素，元素间以指定的分隔符(sep)隔开。
2	table.insert (table, [pos,] value): 在 table 的数组部分指定位置(pos)插入值为 value 的一个元素，其他元素向后移动。pos 参数可选，默认为数组部分末尾。
3	table.maxn (table) 指定 table 中所有正数 key 值中最大的 key 值。如果不存在 key 值为正数的元素，则返回 0。 (Lua5.2 之后该方法已经不存在了,本文使用了自定义函数实现)
4	table.remove (table [, pos]) 返回 table 数组部分位于 pos 位置的元素。其后的元素会被前移。pos 参数可选，默认为 table 长度，即从最后一个元素删起。
5	table.sort (table [, comp])

	对给定的 table 进行升序排序。

1) 连接

我们可以使用 concat() ，将一个列表中的元素连接成一个字符串，然后将其输出。

PS：

```
fruits = {"banana","orange","apple"}
-- 返回将 table 中的元素连接后的字符串
print("连接后的字符串 ",table.concat(fruits))
-- 指定 table 中的各元素以什么字符进行连接
print("连接后的字符串 ",table.concat(fruits,"： "))
-- 指定从 table 中的第几个元素开始，连接到第几个元素结束
print("连接后的字符串 ",table.concat(fruits,", ", 2,3))
```

2)插入和移除

PS：

```
fruits = {"banana","orange","apple"}

-- 未指定插入位置，默认在 table 的末尾进行插入
table.insert(fruits,"mango")
print("索引为 4 的元素为 ",fruits[4])

-- 指定了插入位置，在索引为 2 的键处插入，其他元素后移
table.insert(fruits,2,"grapes")
print("索引为 2 的元素为 ",fruits[2])

print("最后一个元素为 ",fruits[5])
table.remove(fruits)--未指定位置，将从最后一个元素开始移除
print("移除后最后一个元素为 ",fruits[5])
```

3)排序

PS：

```
fruits = {"banana","orange","apple","grapes"}
print("排序前")
for key,value in ipairs(fruits)
do
    print(key,value)
end

table.sort(fruits)--将 table 表按照顺序进行排序
print("排序后")
for key,value in ipairs(fruits)
do
    print(key,value)
end
```

4)最大值

table.maxn 在 Lua5.2 之后该方法已经不存在了，我们定义了 table_maxn 方法来实现。

PS：

```
function table_maxn(t)
    local mn=nil;
    for key, value in pairs(t)
    do
        if(mn==nil)
        then
            mn=value
        end
        if mn < value
        then
            mn = value
        end
    end
    return mn
end
tbl = {[1] = 2, [2] = 6, [3] = 34, [26] =5}--关联型数组
print("tbl 最大值：", table_maxn(tbl))
print("tbl 长度 ", #tbl)
```

注意：

当我们获取 table 的长度的时候无论是使用 # 还是 table.getn 其都会在索引中断的地方停止计数，而导致无法正确取得 table 的长度。

可以使用以下方法来代替：

PS：

```
function table_leng(t)
    local leng=0
    for key, value in pairs(t)
    do
        leng=leng+1
    end
    return leng;
end
```

十三. 数据文件与持久化

当我们处理数据文件的，一般来说，写文件比读取文件内容来的容易。因为我们可以很好的控制文件的写操作，而从文件读取数据常常碰到不可预知的情况。一个健壮的程序不仅应该可以读取存有正确格式的数据还应该能够处理坏文件（译者注：对数据内容和格式进行校验，对异常情况能够做出恰当处理）。正因为如此，实现一个健壮的读取数据文件的程序是很困难的。

文件格式可以通过使用 Lua 中的 table 构造器来描述。我们只需要在写数据的稍微做一些做一点额外的工作，读取数据将变得容易很多。方法是：将我们的数据文件内容作为 Lua 代码写到 Lua 程序中去。通过使用 table 构造器，这些存放在 Lua 代码中的数据可以像其他普通的文件一样看起来引人注目。

为了更清楚地描述问题，下面我们看看例子。如果我们的数据是预先确定的格式，比如 CSV（逗号分割值），我们几乎没得选择。但是，如果我们打算创建一个文件为了将来使用，

除了 CSV，我们可以使用 Lua 构造器来我们表述我们数据，这种情况下，我们将每一个数据记录描述为一个 Lua 构造器。将下面的代码

Donald E. Knuth,Literate Programming,CSLI,1992

Jon Bentley,More Programming Pearls,Addison-Wesley,1990

写成

PS：

```
Entry{"Donald E. Knuth",
"Literate Programming",
"CSLI",
1992}
Entry{"Jon Bentley",
"More Programming Pearls",
"Addison-Wesley",
1990}
```

记住 Entry{...}与 Entry({...})等价，他是一个以表作为唯一参数的函数调用。所以，前面那段数据在 Lua 程序中表示如上。如果要读取这个段数据，我们只需要运行我们的 Lua 代码。

PS：

```
local count = 0
function Entry (b) count = count + 1 end
dofile("data")
print("number of entries: " .. count)
```

下面这段程序收集一个作者名列表中的名字是否在数据文件中出现，如果在文件中出现则打印出来。（作者名字是 Entry 的第一个域；所以，如果 b 是一个 entry 的值，b[1]则代表作者名）

PS：

```
local authors = {}     -- a set to collect authors
function Entry (b) authors[b[1]] = true end
dofile("data")
for name in pairs(authors) do print(name) end
```

注意，在这些程序段中使用事件驱动的方法：Entry 函数作为回调函数，dofile 处理数据文件中的每一记录都回调用它。当数据文件的大小不是太大的情况下，我们可以使用 name-value 对来描述数据。

PS：

```
Entry{
author = "Donald E. Knuth",
title = "Literate Programming",
publisher = "CSLI",
year = 1992
}

Entry{
author = "Jon Bentley",
title = "More Programming Pearls",
publisher = "Addison-Wesley",
```

year = 1990
}

（如果这种格式让你想起 BibTeX，这并不奇怪。Lua 中构造器正是根据来自 BibTeX 的灵感实现的）这种格式我们称之为自描述数据格式，因为每一个数据段都根据他的意思简短的描述为一种数据格式。相对 CSV 和其他紧缩格式，自描述数据格式更容易阅读和理解，当需要修改的时候可以容易的手工编辑，而且不需要改动数据文件。例如，如果我们想增加一个域，只需要对读取程序稍作修改即可，当指定的域不存在时，也可以赋予默认值。使用 name-value 描述的情况下，上面收集作者名的代码可以改写为。

PS：

```
local authors = {}     -- a set to collect authors
function Entry (b) authors[b.author] = true end
dofile("data")
for name in pairs(authors) do print(name) end
```

现在，记录域的顺序无关紧要了，甚至某些记录即使不存在 author 这个域，我们也只需要稍微改动一下代码即可。

```
function Entry (b)
if b.author then authors[b.author] = true end
end
```

Lua 不仅运行速度快，编译速度也快。例如，上面这段搜集作者名的代码处理一个 2MB 的数据文件时间不会超过 1 秒。另外，这不是偶然的，数据描述是 Lua 的主要应用之一，从 Lua 发明以来，我们花了很多心血使他能够更快的编译和运行大的 chunks。

1. 序列化

我们经常需要序列化一些数据，为了将数据转换为字节流或者字符流，这样我们就可以保存到文件或者通过网络发送出去。我们可以在 Lua 代码中描述序列化的数据，在这种方式下，我们运行读取程序即可从代码中构造出保存的值。

通常，我们使用这样的方式 varname = <exp>来保存一个全局变量的值。varname 部分比较容易理解，下面我们来看看如何写一个产生值的代码。

对于一个数值，PS：

```
function serialize (o)
    if type(o) == "number" then
        io.write(o)
    else ...
end
```

对于字符串值，PS：

```
if type(o) == "string" then
  io.write("""", o, """)
```

然而，如果字符串包含特殊字符（比如引号或者换行符），产生的代码将不是有效的 Lua 程序，这时候你可能用下面方法解决特殊字符的问题。

PS：

```
if type(o) == "string" then
  io.write("[[", o, "]]")
```

千万不要这样做！双引号是针对手写的字符串的而不是针对自动产生的字符串。如果有人恶意的引导你的程序去使用"]]..os.execute('rm *')..[["这样的方式去保存某些东西（比如它可能提供字符串作为地址）你最终的 chunk 将是这个样子。

varname = [[ ]]..os.execute('rm *')..[[ ]]

如果你 load 这个数据，运行结果可想而知的。为了以安全的方式引用任意的字符串，string 标准库提供了格式化函数专门提供"%q"选项。它可以使用双引号表示字符串并且可以正确的处理包含引号和换行等特殊字符的字符串。

这样一来，我们的序列化函数可以写为。

PS：

```
function serialize (o)
    if type(o) == "number" then
        io.write(o)
    elseif type(o) == "string" then
        io.write(string.format("%q", o))
    else ...
end
```

1) 保存不带循环的 table

我们下一个艰巨的任务是保存表。根据表的结构不同，采取的方法也有很多。没有一种单一的算法对所有情况都能很好地解决问题。简单的表不仅需要简单的算法而且输出文件也需要看起来美观。

PS：

```
function serialize (o)
    if type(o) == "number" then
        io.write(o)
    elseif type(o) == "string" then
        io.write(string.format("%q", o))
    elseif type(o) == "table" then
        io.write("{\n")
        for k,v in pairs(o) do
            io.write(" ", k, " = ")
            serialize(v)
            io.write(",\n")
        end
    io.write("}\n")
    else
        error("cannot serialize a " .. type(o))
    end
end
```

尽管代码很简单，但很好地解决了问题。只要表结构是一个树型结构（也就是说，没有共享的子表并且没有循环），上面代码甚至可以处理嵌套表（表中表）。对于所进不整齐的表我们可以少作改进使结果更美观，这可以作为一个练习尝试一下。（提示：增加一个参数表示缩进的字符串，来进行序列化）。前面的函数假定表中出现的所有关键字都是合法的标示符，如果表中有不符合 Lua 语法的数字关键字或者字符串关键字，上面的代码将碰到麻烦。

一个简单的解决这个难题的方法是将 io.write(" ", k, " = ")改为：

PS：

```
io.write(" [")
serialize(k)
```

```
io.write("] = ")
```

这样一来，我们改善了我们的函数的健壮性，比较一下两次的结果。

PS：

```
-- result of serialize{a=12, b='Lua', key='another "one"'}
-- 第一个版本
{
a = 12,
b = "Lua",
key = "another \"one\"",
}
-- 第二个版本
{
["a"] = 12,
["b"] = "Lua",
["key"] = "another \"one\"",
}
```

我们可以通过测试每一种情况，看是否需要方括号，另外，我们将这个问题留作一个练习给大家。

2)保存带有循环的 table

针对普通拓扑概念上的带有循环表和共享子表的 table，我们需要另外一种不同的方法来处理。构造器不能很好地解决这种情况，我们不使用。为了表示循环我们需要将表名记录下来，下面我们的函数有两个参数，table 和对应的名字。另外，我们还必须记录已经保存过的 table 以防止由于循环而被重复保存。我们使用一个额外的 table 来记录保存过的表的轨迹，这个表的下表索引为 table，而值为对应的表名。

我们做一个限制，要保存的 table 只有一个字符串或者数字关键字。下面的这个函数序列化基本类型并返回结果。

PS：

```
function basicSerialize (o)
    if type(o) == "number" then
        return tostring(o)
    else -- assume it is a string
        return string.format("%q", o)
    end
end
```

关键内容在接下来的这个函数，saved 这个参数是上面提到的记录已经保存的表的踪迹的 table。

PS：

```
function save (name, value, saved)
    saved = saved or {} -- initial value
    io.write(name, " = ")
    if type(value) == "number" or type(value) == "string" then
        io.write(basicSerialize(value), "\n")
    elseif type(value) == "table" then
        if saved[value] then -- value already saved?
```

```
                -- use its previous name
                io.write(saved[value], "\n")
            else
                saved[value] = name -- save name for next time
                io.write("{}\n") -- create a new table
                for k,v in pairs(value) do -- save its fields
                    local fieldname = string.format("%s[%s]", name,
                                                    basicSerialize(k))
                    save(fieldname, v, saved)
                end
            end
        else
            error("cannot save a " .. type(value))
        end
    end
```

如果我们将要保存的 table 为。

PS：

```
a = {x=1, y=2; {3,4,5}}
a[2] = a    -- cycle
a.z = a[1]    -- shared sub-table
```

调用 save('a', a)之后结果为。

PS：

```
a = {}
a[1] = {}
a[1][1] = 3
a[1][2] = 4
a[1][3] = 5
a[2] = a
a["y"] = 2
a["x"] = 1
a["z"] = a[1]
```

（实际的顺序可能有所变化，它依赖于 table 遍历的顺序，不过，这个算法保证了一个新的定义中需要的前面的节点都已经被定义过）

如果我们想保存带有共享部分的表，我们可以使用同样 table 的 saved 参数调用 save 函数，例如我们创建下面两个表。

PS：

```
a = {{"one", "two"}, 3}
b = {k = a[1]}
```

接着保存它们。

PS：

```
save('a', a)
save('b', b)
```

结果将分别包含相同部分：

PS：

```
a = {}
a[1] = {}
a[1][1] = "one"
a[1][2] = "two"
a[2] = 3
b = {}
b["k"] = {}
b["k"][1] = "one"
b["k"][2] = "two"
```

然而如果我们使用同一个 saved 表来调用 save 函数。

PS：

```
local t = {}
save('a', a, t)
save('b', b, t)
```

结果将共享相同部分。

PS：

```
a = {}
a[1] = {}
a[1][1] = "one"
a[1][2] = "two"
a[2] = 3
b = {}
b["k"] = a[1]
```

上面这种方法是 Lua 中常用的方法，当然也有其他一些方法可以解决问题。比如，我们可以不使用全局变量名来保存，即使用封包，用 chunk 构造一个 local 值然后返回之；通过构造一张表，每张表名与其对应的函数对应起来等。Lua 给予你权力，由你决定如何实现。

十四. 环境

Lua 用一个名为 environment 普通的表来保存所有的全局变量。（更精确的说，Lua 在一系列的 environment 中保存他的“global”变量，但是我们有时候可以忽略这种多样性）这种结果的优点之一是他简化了 Lua 的内部实现，因为对于所有的全局变量没有必要非要有不同的数据结构。另一个(主要的)优点是我们可以像其他表一样操作这个保存全局变量的表。为了简化操作，Lua 将环境本身存储在一个全局变量_G 中，（_G._G 等于_G）。例如，下面代码打印在当前环境中所有的全局变量的名字。

```
for n in pairs(_G) do print(n) end
```

这一章我们将讨论一些如何操纵环境的有用的技术。

1. 使用动态名字访问全局变量

通常，赋值操作对于访问和修改全局变量已经足够。然而，我们经常需要一些元编程（meta-programming）的方式，比如当我们需要操纵一个名字被存储在另一个变量中的全局变量，或者需要在运行时才能知道的全局变量。为了获取这种全局变量的值，有的程序员可能写出下面类似的代码。

PS：

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

or

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

如果 varname 是 x，上面连接操作的结果为："return x"（第一种形式为 "value = x"），当运行时才会产生最终的结果。然而这段代码涉及到一个新的 chunk 的创建和编译以及其他很多额外的问题。你可以换种方式更高效更简洁的完成同样的功能，代码如下。

PS：

```
value = _G[varname]
```

因为环境是一个普通的表，所以你可以使用你需要获取的变量（变量名）索引表即可。

也可以用相似的方式对一个全局变量赋值，_G[varname] = value。小心：一些程序员对这些函数很兴奋，并且可能写出这样的代码：_G["a"] = _G["var1"]，这只是 a = var1 的复杂的写法而已。

对前面的问题概括一下，表域可以是型如"io.read" or "a.b.c.d"的动态名字。我们用循环解决这个问题，从_G 开始，一个域一个域的遍历。

PS：

```
function getfield (f)
    local v = _G    -- start with the table of globals
    for w in string.gfind(f, "[%w_]+") do
    v = v[w]
    end
    return v
end
```

我们使用 string 库的 gfind 函数来迭代 f 中的所有单词（单词指一个或多个子母下线的序列）。相对应的，设置一个域的函数稍微复杂些。

PS：

```
a.b.c.d.e = v
```

实际等价于。

```
local temp = a.b.c.d
temp.e = v
```

也就是说，我们必须记住最后一个名字，必须独立的处理最后一个域。新的 setfield 函数，当其中的域不存在的时候还需要创建中间表。

PS：

```
function setfield (f, v)
    local t = _G -- start with the table of globals
    for w, d in string.gfind(f, "([%w_]+)(.?)") do
        if d == "." then -- not last field?
            t[w] = t[w] or {} -- create table if absent
            t = t[w] -- get the table
        else -- last field
            t[w] = v -- do the assignment
        end
    end
end
```

这个新的模式匹配以变量 w 加上一个可选的点（保存在变量 d 中）的域。如果一个域名后面不允许跟上点，表明它是最后一个名字。

PS：

```
setfield("t.x.y", 10)
```

创建一个全局变量表 t，另一个表 t.x，并且对 t.x.y 赋值为 10。

```
print(t.x.y)     --> 10
print(getfield("t.x.y"))     --> 10
```

2. 声明全局变量

全局变量不需要声明，虽然这对一些小程序来说很方便，但程序很大时，一个简单的拼写错误可能引起 bug 并且很难发现。然而，如果我们喜欢，我们可以改变这种行为。

因为 Lua 所有的全局变量都保存在一个普通的表中，我们可以使用 metatables 来改变访问全局变量的行为。

第一个方法代码如下所示。

PS：

```
setmetatable(_G, {
    __newindex = function (_, n)
    error("attempt to write to undeclared variable "..n, 2)
    end,

    __index = function (_, n)
        error("attempt to read undeclared variable "..n, 2)
    end,
})
```

这样一来，任何企图访问一个不存在的全局变量的操作都会引起错误。

```
> a = 1
stdin:1: attempt to write to undeclared variable a
```

但是我们如何声明一个新的变量呢？使用 rawset，可以绕过 metamethod。

```
function declare (name, initval)
    rawset(_G, name, initval or false)
end
```

or 带有 false 是为了保证新的全局变量不会为 nil。

注意：你应该在安装访问控制以前（before installing the access control）定义这个函数，否则将得到错误信息，毕竟你是在企图创建一个新的全局声明。只要刚才那个函数在正确的地方，你就可以控制你的全局变量了。

PS：

```
a = 1
stdin:1: attempt to write to undeclared variable a
declare "a"
a = 1        -- OK
```

但是现在，为了测试一个变量是否存在，我们不能简单的比较他是否为 nil。如果他是 nil 访问将抛出错误。所以，我们使用 rawget 绕过 metamethod。

```
if rawget(_G, var) == nil then
    -- 'var' is undeclared
    ...
End
```

改变控制允许全局变量可以为 nil 也不难，所以我们需要的是创建一个辅助表用来保存所有已经声明的变量的名字。不管什么时候 metamethod 被调用的时候，他会检查这张辅助表看变量是否已经存在。

PS：

```
local declaredNames = {}
function declare (name, initval)
    rawset(_G, name, initval)
    declaredNames[name] = true
end
setmetatable(_G, {
    __newindex = function (t, n, v)
    if not declaredNames[n] then
        error("attempt to write to undeclared var. "..n, 2)
    else
        rawset(t, n, v) -- do the actual set
    end
end,
    __index = function (_, n)
    if not declaredNames[n] then
        error("attempt to read undeclared var. "..n, 2)
    else
        return nil
    end
end,
})
```

两种实现方式，代价都很小可以忽略不计的。第一种解决方法，metamethods 在平常操作中不会被调用。第二种解决方法，他们可能被调用，不过当且仅当访问一个值为 nil 的变量时。

3. 非全局的环境

全局环境的一个问题是，任何修改都会影响你的程序的所有部分。例如，当你安装一个 metatable 去控制全局访问时，你的整个程序都必须遵循同一个指导方针。如果你想使用标准库，标准库中可能使用到没有声明的全局变量，你将碰到坏的结果。

Lua 5.0 允许每个函数可以有自己的环境来改善这个问题，听起来这很奇怪；毕竟，全局变量表的目的就是为了全局性使用。然而在 Section 15.4 我们将看到这个机制带来很多有趣的结构，全局的值依然是随处可以获取的。

可以使用 setfenv 函数来改变一个函数的环境。Setfenv 接受函数和新的环境作为参数。除了使用函数本身，还可以指定一个数字表示栈顶的活动函数。数字 1 代表当前函数，数字 2 代表调用当前函数的函数（这对写一个辅助函数来改变他们调用者的环境是很方便的）依此类推。

应用 setfenv 失败的例子。

PS：

```
a = 1 -- create a global variable
-- change current environment to a new empty table
setfenv(1, {})
print(a)
```

导致：stdin:5: attempt to call global `print' (a nil value)

（你必须在单独的 chunk 内运行这段代码，如果你在交互模式逐行运行他，每一行都

是一个不同的函数，调用 setfenv 只会影响他自己的那一行。）一旦你改变了你的环境，所有全局访问都使用这个新的表，如果她为空，你就丢失所有你的全局变量，甚至_G，所以，你应该首先使用一些有用的值封装（populate）她，比如老的环境。

PS：

```
a = 1 -- create a global variable
-- change current environment
setfenv(1, {_G = _G})
_G.print(a) --> nil
_G.print(_G.a)     --> 1
```

现在，当你访问"global" _G，他的值为旧的环境，其中你可以使用 print 函数，也可以使用继承封装(populate)你的新的环境。

PS：

```
a = 1
local newgt = {}     -- create new environment
setmetatable(newgt, {__index = _G})
setfenv(1, newgt)     -- set it
print(a)     -->1
```

在这段代码新的环境从旧的环境中继承了 print 和 a；然而，任何赋值操作都对新表进行，不用担心误操作修改了全局变量表。另外，你仍然可以通过_G 修改全局变量。

PS：

```
-- continuing previous code
a = 10
print(a)     --> 10
print(_G.a)     --> 1
_G.a = 20
print(_G.a)     --> 20
```

当你创建一个新的函数时，他从创建他的函数中继承了环境变量。所以，如果一个 chunk 改变了他自己的环境，这个 chunk 所有在改变之后定义的函数都共享相同的环境，都会受到影响。

十五.　模块与包

1. 模块

模块类似于一个封装库，从 Lua 5.1 开始，Lua 加入了标准的模块管理机制，可以把一些公用的代码放在一个文件里，以 API 接口的形式在其他地方调用，有利于代码的重用和降低代码耦合度。

Lua 的模块是由变量、函数等已知元素组成的 table，因此创建一个模块很简单，就是创建一个 table，然后把需要导出的常量、函数放入其中，最后返回这个 table 就行。以下为创建自定义模块 module.lua，文件代码格式如下：

PS：

```
-- 文件名为 module.lua
-- 定义一个名为 module 的模块
module = {}

-- 定义一个常量
module.constant = "这是一个常量"
```

```
-- 定义一个函数
function module.func1()
    io.write("这是一个公有函数！\n")
end

local function func2()  --定义一个私有函数
    print("这是一个私有函数！")
end

function module.func3()
    func2()
end

return module
```

由上面的代码可知，模块(module)的结构就是一个 table 的结构，因此可以像操作调用 table 里的元素那样来操作调用模块里的常量或函数。

上面的 func2 声明为程序块的局部变量，即表示一个私有函数，因此是不能从外部访问模块里的这个私有函数，必须通过模块里的公有函数来调用。

2. require 函数

Lua 提供了一个名为 require 的函数用来加载模块。要加载一个模块，只需要简单地调用就可以了。

PS：

require("<模块名>")或者 require "<模块名>"

执行 require 后会返回一个由模块常量或函数组成的 table，并且还会定义一个包含该 table 的全局变量。

1) 直接调用模块

PS：

```
-- module  模块为上面提到的 module.lua
require("module")
print(module.constant)
module.func3()
```

执行结果为：

```
这是一个常量
这是一个私有函数！
```

2)通过别名调用

上面展示了如何直接调用模块，我们也可以给加载的模块定义一个别名变量，方便后续的调用。

PS：

```
-- module  模块为上文提到到  module.lua
-- 别名变量  m
local m = require("module")  --定义一个局部变量去存储模块
print(m.constant)
```

m.func3()

其最终输出结果，和上面直接调用一致。

3)加载机制

对于自定义的模块，模块文件不是放在哪个文件目录都行，函数 require 有它自己的文件路径加载策略，它会尝试从 Lua 文件或 C 程序库中加载模块。

require 用于搜索 Lua 文件的路径是存放在全局变量 package.path 中，当 Lua 启动后，会以环境变量 LUA_PATH 的值来初始这个环境变量。如果没有找到该环境变量，则使用一个编译时定义的默认路径来初始化。

当然，如果没有 LUA_PATH 这个环境变量，也可以自定义设置，在当前用户根目录下打开.profile 文件（没有则创建，打开.bashrc 文件也可以），例如把 "~/lua/" 路径加入 LUA_PATH 环境变量里：

#LUA_PATH

export LUA_PATH="~/lua/?.lua;;"

文件路径以 ";" 号分隔，最后的 2 个 ";;" 表示新加的路径后面加上原来的默认路径。

接着，更新环境变量参数，使之立即生效。

source ~/.profile

这时假设 package.path 的值是：

/Users/dengjoe/lua/?.lua;./?.lua;/usr/local/share/lua/5.1/?.lua;/usr/local/share/lua/5.1/?/init.lua;/usr/local/lib/lua/5.1/?.lua;/usr/local/lib/lua/5.1/?/init.lua

那么调用 require("module") 时就会尝试打开以下文件目录去搜索目标。

/Users/dengjoe/lua/module.lua;

./module.lua

/usr/local/share/lua/5.1/module.lua

/usr/local/share/lua/5.1/module/init.lua

/usr/local/lib/lua/5.1/module.lua

/usr/local/lib/lua/5.1/module/init.lua

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

搜索的文件路径是从全局变量 package.cpath 获取，而这个变量则是通过环境变量 LUA_CPATH 来初始。

搜索的策略跟上面的一样，只不过现在换成搜索的是 so 或 dll 类型的文件。如果找得到，那么 require 就会通过 package.loadlib 来加载它。

3. C 包

Lua 和 C 是很容易结合的，使用 C 为 Lua 写包。

与 Lua 中写包不同，C 包在使用以前必须首先加载并连接，在大多数系统中最容易的实现方式是通过动态连接库机制。

Lua 在一个叫 loadlib 的函数内提供了所有的动态连接的功能。这个函数有两个参数，库的绝对路径和初始化函数。所以典型的调用的例子如下：

PS：

local path = "/usr/local/lua/lib/libluasocket.so"

local f = loadlib(path, "luaopen_socket")

loadlib 函数加载指定的库并且连接到 Lua，然而它并不打开库（也就是说没有调用初始化函数），反之他返回初始化函数作为 Lua 的一个函数，这样我们就可以直接在 Lua 中调用他。

如果加载动态库或者查找初始化函数时出错，loadlib 将返回 nil 和错误信息。我们可

以修改前面一段代码，使其检测错误然后调用初始化函数：

```
local path = "/usr/local/lua/lib/libluasocket.so"
-- 或者 path = "C:\\windows\\luasocket.dll"，这是 Window 平台下
local f = assert(loadlib(path, "luaopen_socket"))
f()  -- 真正打开库
```

一般情况下我们期望二进制的发布库包含一个与前面代码段相似的 stub 文件，安装二进制库的时候可以随便放在某个目录，只需要修改 stub 文件对应二进制库的实际路径即可。

将 stub 文件所在的目录加入到 LUA_PATH，这样设定后就可以使用 require 函数加载 C 库了。

十六. Package

很多语言专门提供了某种机制组织全局变量的命名，比如 Modula 的 modules，Java 和 Perl 的 packages，C++的 namespaces。每一种机制对在 package 中声明的元素的可见性以及其他一些细节的使用都有不同的规则。但是他们都提供了一种避免不同库中命名冲突的问题的机制。每一个程序库创建自己的命名空间，在这个命名空间中定义的名字和其他命名空间中定义的名字互不干涉。

Lua 并没有提供明确的机制来实现 packages。然而，我们通过语言提供的基本的机制很容易实现他。主要的思想是：像标准库一样，使用表来描述 package。

使用表实现 packages 的明显的好处是：我们可以像其他表一样使用 packages，并且可以使用语言提供的所有的功能，带来很多便利。大多数语言中，packages 不是第一类值(first-class values)（也就是说，他们不能存储在变量里，不能作为函数参数。。。）因此，这些语言需要特殊的方法和技巧才能实现类似的功能。

Lua 中，虽然我们一直都用表来实现 packages，但也有其他不同的方法可以实现 package。

1. 基本方法

包的简单方法，是在包内的每一个对象前都加包名作为前缀。例如，假定我们正在写一个操作复数的库，我们使用表来表示复数，表有两个域 r（实数部分）和 i（虚数部分），我们在另一张表中声明我们所有的操作来实现一个包。

PS：

```
complex = {}
function complex.new (r, i) return {r=r, i=i} end
-- defines a constant `i'
complex.i = complex.new(0, 1)
function complex.add (c1, c2)
    return complex.new(c1.r + c2.r, c1.i + c2.i)
end
function complex.sub (c1, c2)
    return complex.new(c1.r - c2.r, c1.i - c2.i)
end
function complex.mul (c1, c2)
    return complex.new(c1.r*c2.r - c1.i*c2.i,
        c1.r*c2.i + c1.i*c2.r)
end
function complex.inv (c)
    local n = c.r^2 + c.i^2
    return complex.new(c.r/n, -c.i/n)
```

```
end
return complex
```

这个库定义了一个全局名：complex，其他的定义都是放在这个表内。有了上面的定义，我们就可以使用符合规范的任何复数操作了。

PS：

```
c = complex.add(complex.i, complex.new(10, 20))
```

这种使用表来实现的包和真正的包的功能并不完全相同。首先，我们对每一个函数定义都必须显示的在前面加上包的名称。第二，同一包内的函数相互调用必须在被调用函数前指定包名。我们可以使用固定的局部变量名，来改善这个问题，然后，将这个局部变量赋值给最终的包。依据这个原则，我们重写上面的代码。

PS：

```
local P = {}
complex = P     -- package name
P.i = {r=0, i=1}
function P.new (r, i) return {r=r, i=i} end
function P.add (c1, c2)
    return P.new(c1.r + c2.r, c1.i + c2.i)
end
```

当在同一个包内的一个函数调用另一个函数的时候（或者调用自身），他仍然需要加上前缀名。至少，它不再依赖于固定的包名。另外，只有一个地方需要包名。可能你注意到包中最后一个语句 return complex。

这个 return 语句并非必需的，因为 package 已经赋值给全局变量 complex 了。但是，我们认为 package 打开的时候返回本身是一个很好的习惯。额外的返回语句并不会花费什么代价，并且提供了另一种操作 package 的可选方式。

2. 私有成员-Privacy

有时候，一个 package 会公开他的所有内容，也就是说，任何 package 的客户端都可以访问他。然而，一个 package 也拥有自己的私有部分（也就是只有 package 本身才能访问）也是很有用的。在 Lua 中一个传统的方法是将私有部分定义为局部变量来实现。例如，我们修改上面的例子增加私有函数来检查一个值是否为有效的复数。

PS：

```
local P = {}
complex = P
local function checkComplex (c)
    if not ((type(c) == "table") and
    tonumber(c.r) and tonumber(c.i)) then
        error("bad complex number", 3)
    end
end
function P.add (c1, c2)
    checkComplex(c1);
    checkComplex(c2);
    return P.new(c1.r + c2.r, c1.i + c2.i)
end
...
```

return P

1) 优点

package 中所有的名字都在一个独立的命名空间中。Package 中的每一个实体（entity）都清楚地标记为公有还是私有。另外，我们实现了一个真正的私有（privacy），私有实体在 package 外部是不可访问的。

2)缺点

访问同一个 package 内的其他公有的实体写法冗余，必须加上前缀 P.。还有一个大的问题是，当我们修改函数的状态(公有变成私有或者私有变成公有)我们必须修改函数的调用方式。

有一个有趣的方法可以立刻解决上述缺点，我们可以将 package 内的所有函数都声明为局部的，最后将他们放在最终的表中。按照这种方法，上面的 complex package 将可以修改成下面的代码。

PS：

```
local function checkComplex (c)
    if not ((type(c) == "table")
    and tonumber(c.r) and tonumber(c.i)) then
        error("bad complex number", 3)
    end
end
local function new (r, i) return {r=r, i=i} end
local function add (c1, c2)
    checkComplex(c1);
    checkComplex(c2);
    return new(c1.r + c2.r, c1.i + c2.i)
end
...
complex = {
    new = new,
    add = add,
    sub = sub,
    mul = mul,
    div = div,
}
```

现在我们调用函数的时候就不再需要加上前缀了，公有的和私有的函数调用方法相同。在 package 的结尾处，有一个简单的列表列出所有公有的函数。可能大多数人觉得这个列表放在 package 的开始处更自然，但我们不能这样做，因为我们必须首先定义局部函数。

3. 包与文件

我们经常写一个 package 然后将所有的代码放到一个单独的文件中。然后我们只需要执行这个文件即加载 package。例如，如果我们将上面我们的复数的 package 代码放到一个文件 complex.lua 中，命令“require complex”将打开这个 package。记住 require 命令不会将相同的 package 加载多次。

需要注意的问题是，搞清楚保存 package 的文件名和 package 名的关系。当然，将他们联系起来是一个好的想法，因为 require 命令使用文件而不是 packages。一种解决方法是

在 package 的后面加上后缀（比如.lua）来命名文件。Lua 并不需要固定的扩展名，而是由你的路径设置决定。例如，如果你的路径包含："/usr/local/lualibs/?.lua"，那么复数 package 可能保存在一个 complex.lua 文件中。

有些人喜欢先命名文件后命名 package。也就是说，如果你重命名文件，package 也会被重命名。这个解决方法提供了很大的灵活性。例如，如果你有两个有相同名称 package，你不需要修改任何一个，只需要重命名一下文件。在 Lua 中我们使用_REQUIREDNAME 变量来重命名。记住，当 require 加载一个文件的时候，它定义了一个变量来表示虚拟的文件名。

PS：

```
local P = {} -- package
if _REQUIREDNAME == nil then
    complex = P
else
    _G[_REQUIREDNAME] = P
End
```

代码中的 if 测试使得我们可以不需要 require 就可以使用 package。如果 _REQUIREDNAME 没有定义，我们用固定的名字表示 package（例子中 complex）。另外，package 使用虚拟文件名注册他自己。如果使用者将库放到文件 cpx.lua 中并且运行 require cpx，那么 package 将本身加载到表 cpx 中。如果其他的使用者将库改名为 cpx_v1.lua 并且运行 require cpx_v1，那么 package 将自动将本身加载到表 cpx_v1 当中。

4. 使用全局表

上面这些创建 package 的方法的缺点是，他们要求程序员注意很多东西。比如，在声明的时候也很容易忘掉 local 关键字。全局变量表的 Metamethods 提供了一些有趣的技术，也可以用来实现 package。这些技术中共同之处在于，package 使用独占的环境。这很容易实现，如果我们改变了 package 主 chunk 的环境，那么由 package 创建的所有函数都共享这个新的环境。

最简单的技术实现，一旦 package 有一个独占的环境，不仅她的所有函数共享环境，并且它的所有全局变量也共享这个环境。所以，我们可以将所有的公有函数声明为全局变量，然后他们会自动作为独立的表（表指 package 的名字）存在，所有 package 必须要做的是将这个表注册为 package 的名字。下面这段代码阐述了复数库使用这种技术的结果。

PS：

```
local P = {}
complex = P
setfenv(1, P)
```

现在，当我们声明函数 add，她会自动变成 complex.add。

```
function add (c1, c2)
    return new(c1.r + c2.r, c1.i + c2.i)
end
```

另外，我们可以在这个 package 中不需要前缀调用其他的函数。例如，add 函数调用 new 函数，环境会自动转换为 complex.new。这种方法提供了对 package 很好的支持，程序员几乎不需要做什么额外的工作，调用同一个 package 内的函数不需要前缀，调用公有和私有函数也没什么区别。如果程序员忘记了 local 关键字，也不会污染全局命名空间，只不过使得私有函数变成公有函数而已。另外，我们可以将这种技术和前一节我们使用的 package 名的方法组合起来。

PS：

```
local P = {}     -- package
if _REQUIREDNAME == nil then
      complex = P
else
      _G[_REQUIREDNAME] = P
end
setfenv(1, P)
```

向上面这样就不能访问其他的 packages 了，一旦我们将一个空表 P 作为我们的环境，我们就不能访问以前的所有全局变量了。下面有好几种方法可以解决这个问题，但都各有利弊。

最简单的解决方法是使用继承，像前面我们看到的一样。

PS：

```
local P = {}     -- package
setmetatable(P, {__index = _G})
setfenv(1, P)
```

（你必须在调用 setfenv 之前调用 setmetatable，你能说出原因么？）使用这种结构，package 就可以直接访问所有的全局标示符，但必须为每一个访问付出一小点代价。理论上来讲，这种解决方法带来一个有趣的结果，你的 package 现在包含了所有的全局变量。例如，使用你的 package 也可以调用标准库的 sin 函数：complex.math.sin(x)。

另外一种快速的访问其他 packages 的方法是声明一个局部变量来保存老的环境。

PS：

```
local P = {}
pack = P
local _G = _G
setfenv(1, P)
```

现在，你必须对外部的访问加上前缀_G.，但是访问速度更快，因为这不涉及到 metamethod。与继承不同的是这种方法，使得你可以访问老的环境；这种方法的好与坏是有争议的，但是有时候你可能需要这种灵活性。

一个更加正规的方法是，只把你需要的函数或者 packages 声明为 local。

PS：

```
local P = {}
pack = P
local sqrt = math.sqrt
local io = io
-- no more external access after this point
setfenv(1, P)
```

这一技术要求稍多，但他使你的 package 的独立性比较好，他的速度也比前面那几种方法快。

十七.　元表-Metatable

在 Lua table 中我们可以访问对应的 key 来得到 value 值，但是却无法对两个 table 进行操作(比如相加)。

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

例如，使用元表我们可以定义 Lua 如何计算两个 table 的相加操作 a+b。

当 Lua 试图对两个表进行相加时，先检查两者之一是否有元表，之后检查是否有一个叫 __add 的字段，若找到，则调用对应的值。 __add 等即时字段，其对应的值（往往是一个函数或是 table）就是"元方法"。

有两个很重要的函数来处理元表：

1. setmetatable(table, metatable)

对指定 table 设置元表(metatable)，如果元表(metatable)中存在 __metatable 键值，setmetatable 会失败。

2. getmetatable(table)

返回对象的元表(metatable)。

PS：

```
mytable = {}                          -- 普通表
mymetatable = {}                      -- 元表
setmetatable(mytable,mymetatable)     -- 把 mymetatable 设为 mytable 的元表
```

以上代码也可以直接写成一行：

```
mytable = setmetatable({},{})
```

下面的代码为返回对象元表：

```
getmetatable(mytable)                 -- 这会返回 mymetatable
```

3. __index 元方法

这是元表(metatable)最常用的键。

当你通过键来访问 table 的时候，如果这个键没有值，那么 Lua 就会寻找该 table 的 metatable（假定有 metatable）中的__index 键。如果__index 包含一个表格，Lua 会在表格中查找相应的键。

我们可以在使用 lua 命令进入交互模式查看：

PS：

```
other = { foo = 3 }
t = setmetatable({}, { __index = other })
t.foo
```

如果__index 包含一个函数的话，Lua 就会调用那个函数，table 和键会作为参数传递给函数。

__index 元方法查看表中元素是否存在，如果不存在，返回结果为 nil；如果存在则由 __index 返回结果。

PS：

```
mytable = setmetatable({key1 = "value1"},
        {
    __index = function(mytable, key)
        If( key == "key2")
        then
            return "metatablevalue"
        else
            return nil
        end
    end
```

```
})
print(mytable.key1,mytable.key2)
```

上述代码，将 mytable 表赋值为 {key1 = "value1"}。

mytable 设置了元表，元方法为 __index。

在 mytable 表中查找 key1，如果找到，返回该元素，找不到则继续。

在 mytable 表中查找 key2，如果找到，返回 metatablevalue，找不到则继续。

判断元表有没有__index 方法，如果__index 方法是一个函数，则调用该函数。

元方法中查看是否传入 "key2" 键的参数（mytable.key2 已设置），如果传入 "key2" 参数返回 "metatablevalue"，否则返回 mytable 对应的键值。

我们可以将以上代码写得更为简洁。

PS：

```
mytable = setmetatable({key1 = "value1"}, { __index = { key2 = "metatablevalue" } })
print(mytable.key1,mytable.key2)
```

1) 总结

Lua 查找一个表元素时的规则，其实就是如下 3 个步骤:

① 在表中查找，如果找到，返回该元素，找不到则继续。

② 判断该表是否有元表，如果没有元表，返回 nil，有元表则继续。

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

4. __newindex 元方法

__newindex 元方法用来对表进行更新，__index 则用来对表进行访问 。

当表中的一个索引赋值，解释器就会查找__newindex 元方法：如果存在则调用这个函数而不进行赋值操作。

下面的示例演示了 __newindex 元方法的应用：

PS：

```
mymetatable = {}
--使用 setmetatable 设置元表
mytable = setmetatable({key1 = "value1"}, { __newindex = mymetatable })

print(mytable.key1)
--在对新索引键（newkey）赋值时，会调用元方法，而不进行赋值。
mytable.newkey = "新值 2"
print(mytable.newkey,mymetatable.newkey)
--而如果对已存在的索引键（key1），则会进行赋值，而不调用元方法 __newindex。
mytable.key1 = "新值 1"
print(mytable.key1,mymetatable.key1)
```

上述示例中为表设置了元方法 __newindex，在对新索引键（newkey）赋值时（mytable.newkey = "新值 2"），会调用元方法，而不进行赋值。而如果对已存在的索引键（key1），则会进行赋值，而不调用元方法 __newindex。

下面的示例使用了 rawset 函数来更新表：

PS：

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

```
        {
    __newindex = function(mytable, key, value)    --这是元方法__newindex
        rawset(mytable, key, "\""..value.."\"")
    end
})

mytable.key1 = "new value"
mytable.key2 = 4

print(mytable.key1,mytable.key2)
```

5. 为表添加操作符

下面的代码演示了两表的相加操作。

PS：

```
-- 计算表中最大值，table.maxn 在 Lua5.2 以上版本中已无法使用
-- 自定义计算表中最大键值函数 table_maxn，即计算表的元素个数
function table_maxn(table)
    local maxnumber = 0
    for key, value in pairs(table)
    do
        if (maxnumber < key)
        then
            maxnumber = key
        end
    end
    return maxnumber
end

-- 两表相加操作
mytable = setmetatable({ 1, 2, 3 },
        {
    __add = function(mytable, newtable)
        for i = 1, table_maxn(newtable)
        do
            --在 table 的数组部分指定位置(pos)插入值为 newtable[i]的一个元素，其他元素向后移动。
            table.insert(mytable, table_maxn(mytable)+1,newtable[i])
        end
        return mytable
    end
})

secondtable = {4,5,6}

mytable = mytable + secondtable--对表执行相加操作
```

```
for key,value in ipairs(mytable)
do
    print(key,value)
end
```

模式	描述
__add	对应的运算符 '+'
__sub	对应的运算符 '-'
__mul	对应的运算符 '*'
__div	对应的运算符 '/'
__mod	对应的运算符 '%'
__unm	对应的运算符 '-'
__concat	对应的运算符 '..'
__eq	对应的运算符 '=='
__lt	对应的运算符 '<'
__le	对应的运算符 '<='

6. __call 元方法

__call 元方法在 Lua 调用一个值时调用。

下面的示例演示了如何计算表中元素的和。

PS：

```
-- 计算表中最大值，table.maxn 在 Lua5.2 以上版本中已无法使用
-- 自定义计算表中最大键值函数 table_maxn，即计算表的元素个数
function table_maxn(t)
    local maxNumber = 0
    for key, value in pairs(t)
    do
        if maxNumber < key
            then
            maxNumber = key
        end
    end
    return maxNumber
end
--setmetatable 就是对指定的 table 设置元表
mytable = setmetatable({10},
        {
    __call = function(mytable, newtable)--定义元方法__call，会在调用一个值时调用。
        sum = 0
        for i = 1, table_maxn(mytable)    --table_maxn 计算表中的元素个数
        do
            sum = sum + mytable[i]
        end
        for i = 1, table_maxn(newtable)    --table_maxn 计算表中的元素个数
        do
```

```
            sum = sum + newtable[i]
        end
        return sum
    end
})
newtable = {10,20,30}
print(mytable(newtable))
```

7. __tostring 元方法

__tostring 元方法用于修改表的输出行为。

下面的示例自定义了表的输出内容。

PS：

```
--setmetatable 就是对指定的 table 设置元表
mytable = setmetatable ({ 10, 20, 30 },
        {
    __tostring = function(mytable)
        sum = 0
        for key, value in pairs(mytable)
        do
            sum = sum + value
        end
        return "表所有元素的和为 " .. sum
    end
})
print(mytable)
```

十八. 协同程序-coroutine

1. 什么是协同(coroutine)

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

协同是非常强大的功能，但是用起来也很复杂。

2. 线程和协同程序区别

线程与协同程序的主要区别在于，一个具有多个线程的程序可以同时运行几个线程，而协同程序却需要彼此协作的运行。

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

协同程序有点类似同步的多线程，在等待同一个线程锁的几个线程有点类似协同。

3. 基本语法

方法	描述
coroutine.create()	创建 coroutine，返回 coroutine，参数是一个函数，当和 resume 配合使用的时候就唤醒函数调用。
coroutine.resume()	重启 coroutine，和 create 配合使用。 操作成功返回 true，否则返回 false。
coroutine.yield()	挂起 coroutine，将 coroutine 设置为挂起状态，

	这个和 resume 配合使用能有很多有用的效果。
coroutine.status()	查看 coroutine 的状态 注：coroutine 的状态有三种：dead，suspended，running，具体什么时候有这样的状态请参考下面的程序。
coroutine.wrap()	创建 coroutine，返回一个函数，一旦你调用这个函数，就进入 coroutine，和 create 功能重复。
coroutine.running()	返回正在跑的 coroutine，一个 coroutine 就是一个线程，当使用 running 的时候，就是返回一个 corouting 的线程号。

下面的示例演示了上表中各个方法的用法。

PS：

```
--创建一个 coroutine
co = coroutine.create(
        function(i)
            print(i);
        end
)

coroutine.resume(co, 1)     --  输出结果为 1
print(coroutine.status(co))   --  输出结果为 dead

print("----------")
--创建 coroutine 和 creat 功能一样
co = coroutine.wrap(
        function(i)
            print(i);
        end
)

co(1)--使用协同程序

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

co2 = coroutine.create(
        function()
            for i=1,10
            do
                print(i)
                if i == 3
                then
                    print(coroutine.status(co2))    --running
```

```
                print(coroutine.running()) --thread:XXXXXX
            end
            coroutine.yield()--挂起协程
        end
    end
)

coroutine.resume(co2) --1
coroutine.resume(co2) --2
coroutine.resume(co2) --3

print(coroutine.status(co2))    -- suspended
print(coroutine.running())--因为协程已被挂起，因此线程号为 nil

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

从 coroutine.running 就可以看出来,coroutine 在底层实现就是一个线程。

当 create 一个 coroutine 的时候就是在新线程中注册了一个事件。

当使用 resume 触发事件的时候，create 的 coroutine 函数就被执行了，当遇到 yield 的时候就代表挂起当前线程，等候再次 resume 触发事件。

接下来我们分析一个更详细的实例。

PS：

```
function foo (a)
    print("foo 函数输出", a)
    return coroutine.yield(2 * a) -- 返回  2*a 的值
end
--创建一个协程
co = coroutine.create(function (a , b)
    print("第一次协同程序执行输出", a, b) -- co-body 1 10
    local r = foo(a + 1)
    --因为在 foo 函数中挂起了协程，因此就不会继续执行后面的内容，除非重新启动协程
    print("第二次协同程序执行输出", r)
    local r, s = coroutine.yield(a + b, a - b)  -- a，b 的值为第一次调用协同程序时传入

    print("第三次协同程序执行输出", r, s)
    return b, "结束协同程序"                    -- b 的值为第二次调用协同程序时传入
end)

print("main", coroutine.resume(co, 1, 10)) -- true, 4
print("--分割线----")
print("main", coroutine.resume(co, "r")) -- true 11 -9
print("---分割线---")
```

```
print("main", coroutine.resume(co, "x", "y")) -- true 10 end
print("---分割线---")
print("main", coroutine.resume(co, "x", "y")) -- cannot resume dead coroutine
print("---分割线---")
```

首先我们调用 resume，将协同程序唤醒，resume 操作成功返回 true，否则返回 false；接着协同程序开始运行，运行到 yield 语句时，yield 会挂起协同程序，第一次 resume 返回；（注意：此处 yield 返回，参数是 resume 的参数）

第二次 resume，再次唤醒协同程序；（注意：此处 resume 的参数中，除了第一个参数，剩下的参数将作为 yield 的参数），yield 返回。

如果使用的协同程序继续运行完成后继续调用 resume 方法则输出：cannot resume dead coroutine。

resume 和 yield 的配合强大之处在于，resume 处于主程中，它将外部状态（数据）传入到协同程序内部；而 yield 则将内部的状态（数据）返回到主程中。

十九. 文件 I/O

Lua I/O 库用于读取和处理文件。分为简单模式（和 C 一样）、完全模式。

简单模式（simple model）拥有一个当前输入文件和一个当前输出文件，并且提供针对这些文件相关的操作。

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

简单模式在做一些简单的文件操作时较为合适。但是在进行一些高级的文件操作的时候，简单模式就显得力不从心。例如同时读取多个文件这样的操作，使用完全模式则较为合适。

PS：

```
file = io.open (filename [, mode])     -- 打开文件的操作语句
```

1. 模式值

模式	描述
r	以只读方式打开文件，该文件必须存在。
w	打开只写文件，若文件存在则文件长度清为0，即该文件内容会消失。若文件不存在则建立该文件。
a	以附加的方式打开只写文件。若文件不存在，则会建立该文件，如果文件存在，写入的数据会被加到文件尾，即文件原先的内容会被保留。（EOF 符保留）
r+	以可读写方式打开文件，该文件必须存在。
w+	打开可读写文件，若文件存在则文件长度清为零，即该文件内容会消失。若文件不存在则建立该文件。
a+	与 a 类似，但此文件可读可写
b	二进制模式，如果文件是二进制文件，可以加上 b
+	加号表示对文件既可以读也可以写

2. 简单模式

简单模式使用标准的 I/O 或使用一个当前输入文件和一个当前输出文件。

以下为 file.lua 文件代码，操作的文件为 test.lua(如果没有你需要创建该文件)，代码如下。

PS：

```
-- 以只读方式打开文件
file = io.open("test.lua", "r")
-- 设置默认输入文件为 test.lua
io.input(file)
-- 输出文件第一行
print(io.read())
-- 关闭打开的文件
io.close(file)
-- 以附加的方式打开只写文件
file = io.open("test.lua", "a")
-- 设置默认输出文件为 test.lua
io.output(file)
-- 在文件最后一行添加 Lua 注释
io.write("--  test.lua 文件末尾注释")
-- 关闭打开的文件
io.close(file)
```

模式	描述
"*n"	读取一个数字并返回它。例：file.read("*n")
"*a"	从当前位置读取整个文件。例：file.read("*a")
"*l"（默认）	读取下一行，在文件尾 (EOF) 处返回 nil。例：file.read("*l")
number	返回一个指定字符个数的字符串，或在 EOF 时返回 nil。例：file.read(5)

1) **其他 IO 方法有：**

a. io.tmpfile()

返回一个临时文件句柄，该文件以更新模式打开，程序结束时自动删除。

b. io.type(file)

检测 obj 是否一个可用的文件句柄。

c. io.flush()

向文件写入缓冲中的所有数据。

d. io.lines(optional file name)

返回一个迭代函数,每次调用将获得文件中的一行内容,当到文件尾时，将返回 nil,但不关闭文件。

3. 完全模式

通常我们需要在同一时间处理多个文件。我们需要使用 file:function_name 来代替 io.function_name 方法。以下实例演示了如何同时处理同一个文件。

PS：

```
-- 以只读方式打开文件
file = io.open("test.lua", "r")
-- 输出文件第一行
print(file:read())
-- 关闭打开的文件
file:close()
-- 以附加的方式打开只写文件
file = io.open("test.lua", "a")
-- 在文件最后一行添加 Lua 注释
file:write("--test")
-- 关闭打开的文件
file:close()
```

上述代码的 read 的参数与简单模式一致。

1) file:seek(optional whence, optional offset)

设置和获取当前文件位置,成功则返回最终的文件位置(按字节),失败则返回 nil 加错误信息。参数 whence 值可以是：

"set"：从文件头开始。

"cur"：从当前位置开始[默认]。

"end"：从文件尾开始。

offset：默认为 0。

不带参数 file:seek()就会返回当前位置，file:seek("set")则是定位到文件头，file:seek("end")则定位到文件尾并返回文件大小。

2)file:flush()

向文件写入缓冲中的所有数据

3)io.lines(optional file name)

打开指定的文件 filename 为读模式并返回一个迭代函数,每次调用将获得文件中的一行内容,当到文件尾时，将返回 nil,并自动关闭文件。

若不带参数时 io.lines() <=> io.input():lines(); 读取默认输入设备的内容，但结束时不关闭文件。

```
for line in io.lines("main.lua")
    do
    print(line)
    end
```

以下实例使用了 seek 方法，定位到文件倒数第 25 个位置并使用 read 方法的*a 参数，即从当期位置(倒数第 25 个位置)读取整个文件。

PS：

```
-- 以只读方式打开文件
file = io.open("test.lua", "r")
file:seek("end",-25)
print(file:read("*a"))
-- 关闭打开的文件
file:close()
```

二十. 错误处理

程序运行中错误处理是必要的，在我们进行文件操作，数据转移及 web service 调用过程中都会出现不可预期的错误。如果不注重错误信息的处理，就会造成信息泄露，程序无法运行等情况。

1. 语法错误

语法错误通常是由于对程序的组件（如运算符、表达式）使用不当引起的。

PS：

```
a  ==  2
```

以上代码执行结果为：

```
lua: test.lua:2: syntax error near '=='。
```

正如你所看到的，以上出现了语法错误，一个"="号跟两个"="号是有区别的。一个 "=" 是赋值表达式，而两个 "=" 是比较运算。

PS：

```
for a= 1,10
    print(a)
end
```

上述代码会报如下错误：

```
lua: test2.lua:2: 'do' expected near 'print'
```

语法错误比程序运行错误更简单，运行错误无法定位具体错误，而语法错误我们可以很快的解决，上面的示例我们只要在 for 语句下添加 do 即可。

2. 运行错误

运行错误是程序可以正常执行，但是会输出报错信息。

下面的示例由于参数输入错误，程序在执行时会报错。

PS：

```
function add(a,b)
    return a+b
end

add(10)
```

上述代码在编译时是可以成功的，但是在运行的时候会产生下面的错误。

PS：

```
lua: test2.lua:2: attempt to perform arithmetic on local 'b' (a nil value)
stack traceback:
      test2.lua:2: in function 'add'
      test2.lua:5: in main chunk
      [C]: ?
```

lua 里调用函数时，即使实参列表和形参列表不一致也能成功调用，多余的参数会被舍弃，缺少的参数会被补为 nil。

以上报错信息是由于参数 b 被补为 nil 后，nil 参与了+运算。

假如 add 函数内不是 "return a+b" 而是 "print(a,b)" 的话，结果会变成 "10 nil" 不会报错。

3. 错误处理

我们可以使用两个函数，即 assert 和 error 来处理错误。

1) assert

PS：

```
local function add(a,b)
    assert(type(a) == "number", "a 不是一个数字")
    assert(type(b) == "number", "b 不是一个数字")
    return a+b
end
add(10)
```

当我们执行上述代码时，会出现下面的错误。

```
lua: test.lua:3: b 不是一个数字
stack traceback:
    [C]: in function 'assert'
    test.lua:3: in local 'add'
    test.lua:6: in main chunk
    [C]: in ?
```

实例中 assert 首先检查第一个参数，若没问题，assert 不做任何事情；否则，assert 以第二个参数作为错误信息抛出。

2)error

语法格式如下。

PS：error (message [, level])

error 的功能是终止正在执行的函数，并返回 message 的内容作为错误信息(error 函数永远都不会返回)

通常情况下，error 会附加一些错误位置的信息到 message 头部。

Level 参数指示获得错误的位置:

Level=1：这是默认的，为调用 error 位置(文件+行号)

Level=2：指出哪个调用 error 的函数的函数

Level=0：不添加错误位置信息

3)pcall 和 xpcall、debug

Lua 中处理错误，可以使用函数 pcall（protected call）来包装需要执行的代码。

pcall 接收一个函数和要传递给后者的参数，并执行，执行结果：有错误、无错误；返回值 true 或者为 false， errorinfo。

语法格式如下。

PS：

```
If  pcall(function_name, ….)
then
-- 没有错误
```

else
-- 一些错误
end
下面是 pcall 函数的一个实际案例。
PS：
print(pcall(function(i) print(i) end, 33))
输出结果为 33 true。
pcall 以一种"保护模式"来调用第一个参数，因此 pcall 可以捕获函数执行中的任何错误。
通常在错误发生时，希望落得更多的调试信息，而不只是发生错误的位置。但 pcall 返回时，它已经销毁了调用栈的部分内容。
Lua 提供了 xpcall 函数，xpcall 接收第二个参数——一个错误处理函数，当错误发生时，Lua 会在调用栈展开（unwind）前调用错误处理函数，于是就可以在这个函数中使用 debug 库来获取关于错误的额外信息了。
debug 库提供了两个通用的错误处理函数：
① debug.debug
提供一个 Lua 提示符，让用户来检查错误的原因。
② debug.traceback
根据调用栈来构建一个扩展的错误消息。

二十一. 调试-Debug

Lua 提供了 debug 库用于提供创建我们自定义调试器的功能。Lua 本身并未有内置的调试器，但很多开发者共享了他们的 Lua 调试器代码。

序号	方法	用途
1	debug():	进入一个用户交互模式，运行用户输入的每个字符串。 使用简单的命令以及其它调试设置，用户可以检阅全局变量和局部变量， 改变变量的值，计算一些表达式，等等。 输入一行仅包含 cont 的字符串将结束这个函数， 这样调用者就可以继续向下运行。
2	getfenv(object):	返回对象的环境变量。
3	gethook(optional thread):	返回三个表示线程钩子设置的值：当前钩子函数，当前钩子掩码，当前钩子计数
4	getinfo ([thread,] f [, what]):	返回关于一个函数信息的表。你可以直接提供该函数，也可以用一个数字 f 表示该函数。数字 f 表示运行在指定线程的调用栈对应层次上的函数：0 层表示当前函数（getinfo 自身）；1 层表示调用 getinfo 的函数（除非是尾调用，这种情况不计入栈）；等等。如果

		f 是一个比活动函数数量还大的数字，getinfo 返回 nil。
5	debug.getlocal ([thread,] f, local):	此函数返回在栈的 f 层处函数的索引为 local 的局部变量的名字和值。这个函数不仅用于访问显式定义的局部变量，也包括形参、临时变量等。
6	getmetatable(value):	把给定索引指向的值的元表压入堆栈。如果索引无效，或是这个值没有元表，函数将返回 0 并且不会向栈上压任何东西。
7	getregistry():	返回注册表表，这是一个预定义出来的表，可以用来保存任何 C 代码想保存的 Lua 值。
8	getupvalue (f, up)	此函数返回函数 f 的第 up 个上值的名字和值。如果该函数没有那个上值，返回 nil。 以'('（开括号）打头的变量名表示没有名字的变量 （去除了调试信息的代码块）。
9	sethook ([thread,] hook, mask [, count]):	将一个函数作为钩子函数设入。字符串 mask 以及数字 count 决定了钩子将在何时调用。掩码是由下列字符组合成的字符串，每个字符有其含义： 'c': 每当 Lua 调用一个函数时，调用钩子； 'r': 每当 Lua 从一个函数内返回时，调用钩子； 'l': 每当 Lua 进入新的一行时，调用钩子。
10	setlocal ([thread,] level, local, value):	这个函数将 value 赋给栈上第 level 层函数的第 local 个局部变量。如果没有那个变量，函数返回 nil。如果 level 越界，抛出一个错误。
11	setmetatable (value, table):	将 value 的元表设为 table（可以是 nil）。返回 value。
12	setupvalue (f, up, value):	这个函数将 value 设为函数 f 的第 up 个上值。如果函数没有那个上值，返回 nil 否则，返回该上值的名字。

13	traceback ([thread,] [message [, level]]):	如果 message 有，且不是字符串或 nil， 函数不做任何处理直接返回 message。否则，它返回调用栈的栈回溯信息。字符串可选项 message 被添加在栈回溯信息的开头。数字可选项 level 指明从栈的哪一层开始回溯（默认为 1，即调用 traceback 的那里）。

二十二. 垃圾回收

Lua 采用了自动内存管理。 这意味着你不用操心新创建的对象需要的内存如何分配出来，也不用考虑在对象不再被使用后怎样释放它们所占用的内存。

Lua 运行了一个垃圾收集器来收集所有死对象（即在 Lua 中不可能再访问到的对象）来完成自动内存管理的工作。Lua 中所有用到的内存，如字符串、表、用户数据、函数、线程、内部结构等，都服从自动管理。

Lua 实现了一个增量标记-扫描收集器。它使用这两个数字来控制垃圾收集循环：垃圾收集器间歇率和垃圾收集器步进倍率。这两个数字都使用百分数为单位（例如：值 100 在内部表示 1）。

垃圾收集器间歇率控制着收集器需要在开启新的循环前要等待多久。增大这个值会减少收集器的积极性。当这个值比 100 小的时候，收集器在开启新的循环前不会有等待。设置这个值为 200 就会让收集器等到总内存使用量达到之前的两倍时才开始新的循环。

垃圾收集器步进倍率控制着收集器运作速度相对于内存分配速度的倍率。增大这个值不仅会让收集器更加积极，还会增加每个增量步骤的长度。不要把这个值设得小于 100， 那样的话收集器就工作的太慢了以至于永远都干不完一个循环。默认值是 200，这表示收集器以内存分配的"两倍"速工作。

如果你把步进倍率设为一个非常大的数字（比你的程序可能用到的字节数还大 10% ），收集器的行为就像一个 stop-the-world 收集器。接着你若把间歇率设为 200，收集器的行为就和过去的 Lua 版本一样了：每次 Lua 使用的内存翻倍时，就做一次完整的收集。

1. 垃圾回收器函数

Lua 提供了以下函数 collectgarbage ([opt [, arg]])用来控制自动内存管理:

1) collectgarbage("collect")

做一次完整的垃圾收集循环。通过参数 opt 它提供了一组不同的功能：

2)collectgarbage("count")

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

3)collectgarbage("restart")

重启垃圾收集器的自动运行。

4)collectgarbage("setpause")

将 arg 设为收集器的间歇率。返回间歇率的前一个值。

collectgarbage("setpause", 200)，表示内存增大 2 倍（200/100）时自动释放一次内存（200 是默认值）。

5)collectgarbage("setstepmul")

返回步进倍率的前一个值。

collectgarbage("setstepmul", 200)，收集器单步收集的速度相对于内存分配速度的倍率，设置 200 时倍率等于 2 倍（200/100）。（200 是默认值）

6)collectgarbage("step")

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

7)collectgarbage("stop")

停止垃圾收集器的运行。在调用重启前，收集器只会因显式的调用运行。

下面的代码演示了一个简单的垃圾回收示例。

PS：

mytable = {"apple", "orange", "banana"}—这是一个直接初始表

print(collectgarbage("count"))—输出结果为 20.9560546875

mytable = nil --删除表

print(collectgarbage("count"))—输出结果为 20.9853515625

print(collectgarbage("collect"))—输出结果为 0

print(collectgarbage("count"))—输出结果为 19.4111328125

2. GC 的原理及其算法设计

不同的语言，对 GC 算法的设计不同，常见的 GC 算法是引用计数和 Mark-Sweep 算法，c#采用的是 Mark-sweep && compact 算法，Lua 采用的是 Mark-Sweep 算法，分开说一下：

引用计数算法：在一个对象被引用的情况下，将其引用计数加 1，反之则减 1，如果计数值为 0，则在 GC 的时候回收，这个算法有个问题就是循环引用。

Mark-Sweep 算法：每次 GC 的时候，对所有对象进行一次扫描，如果该对象不存在引用，则被回收，反之则保存。

在 Lua5.0 及其更早的版本中，Lua 的 GC 是一次性不可被打断的过程，使用的 Mark 算法是双色标记算法(Two color mark)，这样系统中对象的非黑即白，要么被引用，要么不被引用，这会带来一个问题：在 GC 的过程中如果新加入对象，这时候新加入的对象无论怎么设置都会带来问题，如果设置为白色，则如果处于回收阶段，则该对象会在没有遍历其关联对象的情况下被回收；如果标记为黑色，那么没有被扫描就被标记为不可回收，是不正确的。

为了降低一次性回收带来的性能问题以及双色算法的问题，在 Lua5.1 后，Lua 都采用分布回收以及三色增量标记清除算法（Tri-color incremental mark and sweep）。

PS：

每个新创建的对象颜色设置为白色

//初始化阶段

遍历 root 节点中引用的对象，从白色置为灰色，并且放入到灰色节点列表中

//标记阶段

while(灰色链表中还有未扫描的元素)：

从中取出一个对象，将其置为黑色

遍历这个对象关联的其他所有对象：

if 为白色

标记为灰色，加入到灰色链表中(insert to the head)

//回收阶段

遍历所有对象：
if 为白色，
没有被引用的对象，执行回收
else
重新塞入到对象链表中，等待下一轮 GC

二十三. 面向对象程序设计

Lua 中的表不仅在某种意义上是一种对象。像对象一样，表也有状态（成员变量）；也有与对象的值独立的本性，特别是拥有两个不同值的对象（table）代表两个不同的对象；一个对象在不同的时候也可以有不同的值，但他始终是一个对象；与对象类似，表的生命周期与其由什么创建、在哪创建没有关系。对象有他们的成员函数，表(Table)也有：

PS：

```
Account = {balance = 0}
function Account.withdraw (v)
  Account.balance = Account.balance - v
End
```

上面代码定义并创建了一个新的函数，并且保存在 Account 对象的 withdraw 域内，下面我们可以这样调用：

PS：

```
Account.withdraw(100.00)
```

这种函数就是我们所谓的方法，然而，在一个函数内部使用全局变量名 Account 是一个不好的习惯。首先，这个函数只能在这个特殊的对象（这里指 Account）中使用；第二，即使对这个特殊的对象而言，这个函数也只有在对象被存储在特殊的变量（指 Account）中才可以使用。如果我们改变了这个对象的名字，函数 withdraw 将不能工作：

PS：

```
a = Account;
Account = nil
a.withdraw(100.00) -- ERROR!
```

这种行为违背了前面的对象应该有独立的生命周期的原则。

一个灵活的方法是：定义方法的时候带上一个额外的参数，来表示方法作用的对象。这个参数经常为 self 或者 this。

PS：

```
function Account.withdraw (self, v)
  self.balance = self.balance - v
end
```

现在，当我们调用这个方法的时候不需要指定他操作的对象了。

PS：

```
a1 = Account; Account = nil
...
a1.withdraw(a1, 100.00) --  这是可行的
```

使用 self 参数定义函数后，我们可以将这个函数用于多个对象上。

PS：

```
a2 = {balance=0, withdraw = Account.withdraw}
...
```

a2.withdraw(a2, 260.00)

self 参数的使用是很多面向对象语言的要点。大多数语言将这种机制隐藏起来，这样程序员不必声明这个参数（虽然仍然可以在方法内使用这个参数）。Lua 也提供了通过使用冒号操作符来隐藏这个参数的声明。我们可以重写上面的代码：

PS：

```
function Account:withdraw (v)
  self.balance = self.balance - v
end
a:withdraw(100.00)
```

冒号的效果相当于在函数定义和函数调用的时候，增加一个额外的隐藏参数。这种方式只是提供了一种方便的语法，实际上并没有什么新的内容。我们可以使用 dot 语法定义函数而用冒号语法调用函数，反之亦然，只要我们正确的处理好额外的参数。

PS：

```
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)
```

现在我们的对象拥有一个标示符，一个状态和操作这个状态的方法。但他们依然缺少一个 class 系统，继承和隐藏。先解决第一个问题：我们如何才能创建拥有相似行为的多个对象呢？明确地说，我们怎样才能创建多个 accounts？（针对上面的对象 Account 而言）

1. 类

一些面向对象的语言中提供了类的概念，作为创建对象的模板。在这些语言里，对象是类的实例。Lua 不存在类的概念，每个对象定义他自己的行为并拥有自己的形状（shape）。然而，依据基于原型（prototype）的语言比如 Self 和 NewtonScript，在 Lua 中仿效类的概念并不难。在这些语言中，对象没有类。相反，每个对象都有一个 prototype（原型），当调用不属于对象的某些操作时，会最先会到 prototype 中查找这些操作。在这类语言中实现类（class）的机制，我们创建一个对象，作为其它对象的原型即可（原型对象为类，其它对象为类的 instance）。类与 prototype 的工作机制相同，都是定义了特定对象的行为。

在 Lua 中，使用继承的思想，很容易实现 prototypes。更明确的来说，如果我们有两个对象 a 和 b，我们想让 b 作为 a 的 prototype 只需要：

PS：

```
setmetatable(a, {__index = b})
```

这样，对象 a 调用任何不存在的成员都会到对象 b 中查找。术语上，可以将 b 看作类，a 看作对象。回到前面银行账号的例子上。为了使得新创建的对象拥有和 Account 相似的行为，我们使用__index metamethod，使新的对象继承 Account。注意一个小的优化：我们不需要创建一个额外的表作为 account 对象的 metatable；我们可以用 Account 表本身作为

metatable。

PS：

```
function Account:new (o)
    o = o or {} -- create object if user does not provide one
    setmetatable(o, self)
    self.__index = self
    return o
end
```

（当我们调用 Account:new 时，self 等于 Account；因此我们可以直接使用 Account 取代 self。然而，使用 self 在我们下一节介绍类继承时更合适）。有了这段代码之后，当我们创建一个新的账号并且调用一个方法的时候，有什么发生呢？

PS：

```
a = Account:new{balance = 0}
a:deposit(100.00)
```

当我们创建这个新的账号 a 的时候，a 将 Account 作为他的 metatable（调用 Account:new 时，self 即 Account）。当我们调用 a:deposit(100.00)，我们实际上调用的是 a.deposit(a,100.00)（冒号仅仅是语法上的便利）。然而，Lua 在表 a 中找不到 deposit，因此他回到 metatable 的__index 对应的表中查找，情况大致如下：

PS：

```
getmetatable(a).__index.deposit(a, 100.00)
```

a 的 metatable（元表）是 Account，Account.__index 也是 Account（因为 new 函数中 self.__index = self）。所以我们可以重写上面的代码为。

PS：

```
Account.deposit(a, 100.00)
```

也就是说，Lua 传递 a 作为 self 参数，然后调用原始的 deposit 函数。所以，新的账号对象从 Account 继承了 deposit 方法。使用同样的机制，可以从 Account 继承所有的域。继承机制不仅对方法有效，对表中所有的域都有效。所以，一个类不仅提供方法，也提供了他的实例的成员的默认值。

注意：

在我们第一个 Account 定义中，我们提供了成员 balance 默认值为 0，所以，如果我们创建一个新的账号而没有提供 balance 的初始值，他将继承默认值。

PS：

```
b = Account:new()
print(b.balance)        --> 0
```

当我们调用 b 的 deposit 方法时，实际等价于 b.balance = b.balance + v（因为 self 就是 b）。表达式 b.balance 等于 0 并且初始的存款（b.balance）被赋予 b.balance。下一次我们访问这个值的时候，不会在涉及到 index metamethod，因为 b 已经存在他自己的 balance 域。

2. 继承

通常的面向对象语言中，继承是使得类可以访问其他类的方法，这在 Lua 中也很容易实现，下面我们就假定有一个基类 Account。

PS：

```
Account = {balance = 0} –这是一个直接初始表
function Account:new (o)
```

```
  o = o or {}
  setmetatable(o, self)
  self.__index = self
return o
end
function Account:deposit (v)
  self.balance = self.balance + v
end
function Account:withdraw (v)
      if v > self.balance
            then
            error"insufficient funds"
      end
      self.balance = self.balance - v
end
```

我们打算从基类派生出一个子类 SpecialAccount，这个子类允许客户取款超过它的存款余额限制，我们从一个空类开始，从基类继承所有操作。

PS：

```
SpecialAccount = Account:new()
s = SpecialAccount:new{limit=1000.00}
```

SpecialAccount 从 Account 继承了 new 方法，当 new 执行的时候，self 参数指向 SpecialAccount。所以，s 的 metatable 是 SpecialAccount，__index 也是 SpecialAccount。这样，s 继承了 SpecialAccount，后者继承了 Account。

PS：

```
s:deposit(100.00)
```

当我们执行上述代码，Lua 在 s 中找不到 deposit 域，他会到 SpecialAccount 中查找，在 SpecialAccount 中找不到，会到 Account 中查找。使得 SpecialAccount 的特殊之处在于，它可以重定义从父类中继承来的方法。

PS：

```
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，当我们调用方法 s:withdraw(200.00)，Lua 不会到 Account 中查找，因为它第一次就在 SpecialAccount 中发现了新的 withdraw 方法，由于 s.limit 等于 1000.00（记住，我们创建 s 的时候初始化了这个值）程序执行了取款操作，s 的 balance 变成了负值。

在 Lua 中面向对象有趣的一个方面是你不需要创建一个新类去指定一个新的行为。如果仅仅一个对象需要特殊的行为，你可以直接在对象中实现，例如，如果账号 s 表示一些特殊

的客户：取款限制是他的存款的 10%，你只需要修改这个单独的账号：

PS：

```
function s:getLimit ()
    return self.balance * 0.10
end
```

这样声明之后，调用 s:withdraw(200.00)将运行 SpecialAccount 的 withdraw 方法，但是当方法调用 self:getLimit 时，我们新定义的 getLimit 被触发。

3. 多重继承

由于 Lua 中的对象不是原生(primitive)的，所以在 Lua 中有很多方法可以实现面向对象的程序设计。我们前面所见到的使用 index metamethod 的方法可能是简洁、性能、灵活各方面综合最好的。然而，针对一些特殊情况也有更适合的实现方式，下面我们在 Lua 中实现多重继承。

实现的关键在于：将函数用作__index。记住，当一个表的 metatable 存在一个__index 函数时，如果 Lua 调用一个原始表中不存在的函数，Lua 将调用这个__index 指定的函数。这样可以用__index 实现在多个父类中查找子类不存在的域。

多重继承意味着一个类拥有多个父类，所以，我们不能用创建一个类的方法去创建子类。取而代之的是，我们定义一个特殊的函数 createClass 来完成这个功能，将被创建的新类的父类作为这个函数的参数。这个函数创建一个表来表示新类，并且将它的 metatable 设定为一个可以实现多继承的__index metamethod。尽管是多重继承，每一个实例依然属于一个在其中能找得到它需要的方法的单独的类。所以，这种类和父类之间的关系与传统的类与实例的关系是有区别的。特别是，一个类不能同时是其实例的 metatable 又是自己的 metatable。在下面的实现中，我们将一个类作为他的实例的 metatable，创建另一个表作为类的 metatable。

PS：

```
-- look up for `k' in list of tables 'plist'
local function search (k, plist)
    for i=1, table.getn(plist)
    do
    local v = plist[i][k] -- try 'i'-th superclass
    if v
        then
        return v
        end
    end
end
function createClass (...)
local c = {} -- new class
-- class will search for each method in the list of its
-- parents (`arg' is the list of parents)
 setmetatable(c, {__index = function (t, k)
return search(k, arg)
end})
-- prepare `c' to be the metatable of its instances
c.__index = c
```

```
-- define a new constructor for this new class
function c:new (o)
 o = o or {}
 setmetatable(o, c)
return o
end
-- return new class
return c
end
```

让我们用一个小例子阐明一下 createClass 的使用，假定我们前面的类 Account 和另一个类 Named，Named 只有两个方法 setname and getname。

PS：

```
Named = {} --这是一个空表
function Named:getname ()
      return self.name
end
function Named:setname (n)
      self.name = n
end
```

为了创建一个继承于这两个类的新类，我们将调用 createClass。

PS：

```
NamedAccount = createClass(Account, Named)
```

为了创建和使用实例，我们将像往常一样设置。

PS：

```
account = NamedAccount:new{name = "Paul"}
print(account:getname())       --> Paul
```

现在我们看看上面最后一句发生了什么，Lua 在 account 中找不到 getname，因此他查找 account 的 metatable 的__index，即 NamedAccount。但是，NamedAccount 也没有 getname，因此 Lua 查找 NamedAccount 的 metatable 的__index，因为这个域包含一个函数，Lua 调用这个函数并首先到 Account 中查找 getname，没有找到，然后到 Named 查找，找到并返回最终的结果。当然，由于搜索的复杂性，多重继承的效率比起单继承要低。一个简单的改善性能的方法是将继承方法拷贝到子类，使用这种技术，index 方法如下。

PS：

```
setmetatable(c, {__index = function (t, k)
      local v = search(k, arg)
      t[k] = v -- save for next access
      return v
end})
```

应用这个技巧，访问继承的方法和访问局部方法一样快（特别是第一次访问）。缺点是系统运行之后，很难改变方法的定义，因为这种改变不能影响继承链的下端。

4. 私有性-privacy

很多人认为私有性是面向对象语言的应有的一部分。每个对象的状态应该是这个对象自己的事情。在一些面向对象的语言中，比如 C++和 Java 你可以控制对象成员变量或者成员方法是否私有。其他一些语言比如 Smalltalk 中，所有的成员变量都是私有，所有的成员方

法都是公有的。第一个面向对象语言 Simula 不提供任何保护成员机制。

如前面我们所看到的，Lua 中的主要对象设计不提供私有性访问机制。部分原因是因为，我们使用通用数据结构 tables 来表示对象的结果。但是，这也反映了 Lua 的设计思想，Lua 没有打算被用来进行大型的程序设计，相反，Lua 目标定于小型到中型的程序设计，通常是作为大型系统的一部分。典型的，被一个或者很少几个程序员开发，甚至被非程序员使用。所以，Lua 避免太冗余和太多的人为限制，如果你不想访问一个对象内的一些东西就不要访问。

然而，Lua 的另一个目标是灵活性，提供程序员元机制（meta-mechanisms），通过它你可以实现很多不同的机制。虽然 Lua 中基本的面向对象设计并不提供私有性访问的机制，我们可以用不同的方式来实现他。虽然这种实现并不常用，但知道他也是有益的，不仅是因为它展示了 Lua 的一些有趣的角落，也因为它可能是某些问题的很好地解决方案。

设计的基本思想是，每个对象用两个表来表示：一个描述状态；另一个描述操作（或者叫接口）。对象本身通过第二个表来访问，也就是说，通过接口来访问对象。为了避免未授权的访问，表示状态的表中不涉及到操作；表示操作的表也不涉及到状态，取而代之的是，状态被保存在方法的闭包内。例如，用这种设计表述我们的银行账号，我们使用下面的函数工厂创建新的对象。

PS：

```
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 [1]= deposit,
        getBalance = getBalance
    }
end
```

首先，函数创建一个表用来描述对象的内部状态，并保存在局部变量 self 内。然后，函数为对象的每一个方法创建闭包（也就是说，嵌套的函数实例）。最后，函数创建并返回外部对象，外部对象中将局部方法名指向最终要实现的方法。这儿的关键点在于，这些方法没有使用额外的参数 self，代替的是直接访问 self。因为没有这个额外的参数，我们不能使用冒号语法来访问这些对象，函数只能像其他函数一样调用。

PS：

```
acc1 = newAccount(100.00)
acc1.withdraw(40.00)
print(acc1.getBalance())   --输出结果为 60
```

这种设计，使得任何存储在 self 表中的部分都是私有的，newAccount 返回之后，没有什么方法可以直接访问对象，我们只能通过 newAccount 中定义的函数来访问他。虽然我们

[1] v. 放下，放置；储蓄；存放，寄存；沉积，沉淀；n. 沉积物，沉积层；订金；押金；存款；

的例子中仅仅将一个变量放到私有表中，但是我们可以将对象的任何的部分放到私有表中。

我们也可以定义私有方法，他们看起来像是公有的(Public)，但我们并不将其放到接口中。例如，我们的账号可以给某些用户取款享有额外的 10%的存款上限，但是我们不想用户直接访问这种计算的详细信息，我们实现如下。

PS：

```
function newAccount (initialBalance)
    local self = {
        balance = initialBalance,
        LIM = 10000.00,
    }
    local extra = function ()
        if self.balance > self.LIM then
            return self.balance*0.10
        else
            return 0
        end
    end
    local getBalance = function ()
        return self.balance + self.extra()
    end
```

像上面这样，用户就没有办法直接访问 extra 函数了。

5. Single-Method 的对象实现方法

前面的程序设计的方法有一种特殊情况：对象只有一个单一的方法。这种情况下，我们不需要创建一个接口表，取而代之的是，我们将这个单一的方法作为对象返回。这听起来有些不可思议，其实，一个保存状态的迭代子函数就是一个 single-method 对象。

PS：

```
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
```

single-method 对象使用起来很简单。

PS：

```
d = newObject(0)
print(d("get"))     --输出结果为 0
d("set", 10)
print(d("get"))     --输出结果为 10
```

这种非传统的对象实现是非常有效的，语法 d("set",10)虽然很罕见，但也只不过比传统的 d:set(10)长两个字符而已。每一个对象是用一个单独的闭包，代价比起表来小的多。这种方式没有继承但有私有性，访问对象状态的唯一方式是通过它的内部方法。

Tcl/Tk 的窗口部件（widgets）使用了相似的方法，在 Tk 中一个窗口部件的名字表示一

个在窗口部件上执行各种可能操作的函数（a widget command）。

二十四. Lua 热更新

1. 原理

Lua 的 require(modelname) 把一个 lua 文件加载存放到 ackage.loaded[modelname]。

当我们加载一个模块的时候，会先判断是否在 package.loaded 中已存在，若存在则返回改模块，不存在才会加载(loadfile)，防止重复加载。

package.loaded 是一个 Table，其中包含了全局表_G、默认加载的模块(string, debug, package, io, os, table, math, coroutine)和用户加载的模块。

2. 热更新

最简单粗暴的热更新就是将 package.loaded[modelname]的值置为 nil，强制重新加载：

PS：

```
function reload_module(module_name)
    package.loaded[modulename] = nil
    require(modulename)
end
```

这样做虽然能完成热更，但问题是已经引用了该模块的地方不会得到更新， 因此我们需要将引用该模块的地方的值也做对应的更新。

PS：

```
function reload_module(module_name)
    local old_module = _G[module_name]

    package.loaded[module_name] = nil
    require (module_name)

    local new_module = _G[module_name]
    for k, v in pairs(new_module) do
        old_module[k] = v
    end

    package.loaded[module_name] = old_module
end
```

二十五. C#与 Lua 交互

1. 从代码文件方面解释

1) Lua 调用 C#过程

Lua→wrap→C#

先生成 Wrap 文件，Wrap 文件把字段方法注册到 lua 虚拟机，然后 lua 通过 wrap 就可以调 C#了。

2) C#调用 Lua 过程

C#→Bridge→dll→Lua OR C#→dll→Lua

C#生成 Bridge 文件，Bridge 调用 dll 文件，先调用 lua 中 dll 文件，由 dll 文件执行 lua 代码。

二十六. 优化

1. 尽可能使用局部变量

应该尽可能的使用局部变量，这可以避免命名冲突；访问局部变量的速度比全局变量更快。

二十七. 技巧

对于 lua 中的 Table，我们可以直接使用连接字符去循环获取对应的字段。

```
for i=0 , j do
table["GameObject"..i]
end
```

1. ipairs 和 pairs 的区别

简单来说，pairs 会遍历表中全部的键值对（key，value），其中键可以是字符串也可以是整数（例如：array['wjr']，array[2]）。而 ipairs 则从下标为 1 开始遍历，然后下标累加 1（例如：array[1]，array[2]，...），如果某个下标元素不存在就终止遍历。

这就导致如果下标不连续或者不是从 1 开始的表，如果使用 ipairs 遍历就会中断或者遍历不到所有元素。

① ipairs 遇到 nil 会停止，pairs 会输出 nil 值然后继续下去。

② ipairs 并不会输出 table 中存储的键值对,会跳过键值对，然后顺序输出 table 中的值。而 pairs 会输出 table 的键值对，先顺序输出值，再乱序(键的哈希值)输出键值对。

③ pairs 用于迭代 table；ipairs 用于迭代数组。

1) 解决既要顺序遍历，也要遇到 nil 也能遍历完全

PS：

```
for idx=1, maxSize do
        if pTable[idx] ~= nil then --不等于
                -- 做相应的处理...
        end
end
```

2. 热更新方案有哪些？

1) 整包：存放在上 SteamingAssets 里

策略：完整更新资源放在包里

优点：首次更新少

缺点：安装包下载时间长，首次安装久

2) 分包

策略：少部分资源放在包里，大部分更新资源存放在更新资源器中。

优点：安装包小，安装时间短，下载快。

缺点：首次更新下载解压缩包时间九。

3) 适用性

海外游戏大部分是使用分包策略，平台规定

国内游戏大部分是使用整包策略

3. 资源如何打包？依赖项列表如何生成？

1) 查找指定文件夹 ABResource 里的资源文件

① Directory.GetFile(资源路径)

② 新建 AssetBundleBuild 对象
③ 获取资源名称，并赋值对应 AB 名称
④ 获取各个资源的依赖项：通过 UnityEditor.AssetDataBase 类获取各个资源的依赖项

2) 使用 Unity 自带的 BuildPipeline 进行构建 AB 包
① BuildPipeLine.BuildAssetBundles(输出 AB 包路径)
② File.WriteAllLines(将依赖项写入文件里)

4. lua 深拷贝和浅拷贝的区别？如何实现深拷贝？

1) 实现浅拷贝

使用 = 运算符进行浅拷贝。

① 拷贝的对象是 string、number、bool 基本类型。拷贝的过程就是复制黏贴！修改新拷贝出来的对象，不会影响原先对象的值，两者互不干涉。
② 拷贝对象是 table 表，拷贝出来的对象和原先对象是同一个对象，占用同一个对象，只是一个人两个名字，类似 C#引用地址，指向同一个堆里的数据~，两者任意改变都会影响对方。

2) 深拷贝

复制对象的基本类型，也复制源对象中的对象。赋值一个全新的一模一样的对象，但不是同一个表。

Lua 没有实现深拷贝，需封装一个函数，递归拷贝 table 中所有元素，以及设置 metetable 元表。

如果 key 和 value 都不包含 table 属性，那么每次在泛型 for 内调用的 Func 就直接由 if 判断返回具体的 key 和 value。

如果有包含多重 table 属性，那么这段 if 判断就是用来解开下一层 table 的，最后层层递归返回。

PS：

```
t={name="asd",hp=100,table1={table={na="aaaaaaaa"}}};
--实现深拷贝的函数
function copy_Table(obj)
        function copy(obj)
            if type(obj)  ~= "table" then                    --对应代码梳理“1”  （代码梳理在下面）
                return obj;
            end
            local newTable={};                               --对应代码梳理“2”
            for k,v in pairs(obj) do
                newTable[copy(k)]=copy(v);                   --对应代码梳理“3”
            end
            return setmetatable(newTable,getmetatable(obj));
        end
    return copy(obj)
end
a=copy_Table(t);
```

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

① 先看 copy 方法中的代码，如果这个类型不是表的话，就没有遍历的必要，可以直接作为返回值赋值。

② 当前传入的变量是表，就新建一个表来存储老表中的数据，下面就是遍历老表，并分别将 k,v 赋值给新建的这个表，完成赋值后，将老表的元表赋值给新表。

③ 在对 k,v 进行赋值时，同样要调用 copy 方法来判断一下是不是表，如果是表就要创建一个新表来接收表中的数据，以此类推并接近无限递归。

5. __index 和__newindex 元方法的区别

__newindex 用于表的更新，__index 用于表的查询。

如果访问不存在的数据，由__index 提供最终结果；如果对不存在的数据赋值，由__newindex 对数据进行赋值。

__index 元方法可以是一个函数，Lua 语言就会以【表】和【不存在键】为参数调用该函数；__index 元方法也可以是一个表，Lua 语言就访问这个元表，对表中不存在的值进行赋值的时候，解释器会查找__newindex；__newindex 元方法如果是一个表，Lua 语言就对这个元表的字段进行赋值。

6. 函数冒号(:)与点(.)的区别

冒号隐式传递 self，点的话需要显示传递。

7. 获取系统时间及格式化方法

1) 系统当前时间对应的时间戳

PS：

```
local ntime = os.time
```

2) 格式化时间显示

使用 os.date ([format [, time]])

由原型可以知道，我们可以省略第二个参数也可以省略两个参数。只省略第二个参数函数会使用当前时间作为第二个参数，如果两个参数都省略则按当前系统的设置返回格式化的字符串，做以下等价替换 os.date() <=> os.date("%c")。

如果 format 以“!”开头，则按格林尼治时间进行格式化。

如果 format 是一个“*t”，将返一个带 year(4 位)，month(1-12)，day (1--31)，hour (0-23)，min (0-59)，sec (0-61)，wday (星期几， 星期天为 1)， yday (年内天数)和 isdst (是否为日光节约时间 true/false)的带键名的表。

如果 format 不是“*t”，os.date 会将日期格式化为一个字符串，表格如下。

代码	描述	示例
%a	一星期中天数的简写	星期三就是 Wed
%A	星期几的全称	星期三就是 Wednesday
%b	月的简缩写	九月就是 Sep
%B	月的全称	九月就是 September
%c	日期和时间	09/16/98 23:48:10
%d	一个月中的第几天	范围是 01-31
%H	小时（使用 24 小时格式）	范围是 00-23
%I	小时（使用 12 小时格式）	范围是 01-12

%M	分钟	范围是 00-59
%m	月	范围是 01-12
%p	要么是 am（上午）要么是 pm（下午）	
%S	秒	范围是 00-61
%j	一年中的第几天	01 - 366
%w	星期几	范围是 0-6 对应 Sunday-Saturday
%W	一年中的第几个星期	范围是 0 - 52
%x	日期	09/16/98
%X	时间	23:48:10
%Y	某年的全称	1998
%y	某年的后两位	98（范围是 00-99）
%%	字符%	

PS：

os.date("1060004s is %x", 1060004)

os.date("today is %c, in %A") --表示今天的日期，在星期几

3) 总结

注意 format "!" 的用法，因为我们的北京时间处于东 8 区，所以两次的结果会差 8 个小时。

注意使用 format "*t"返回的 table 中 wday 如果是 1 表示星期天，而使用通用格式时%w 用 0 表示星期天。

Lua 标准库中提供了时间函数 os.time()和 os.date()，这两个函数需要注意的地方就是加入了时区的概念。

a. os.time()

不传参的话返回当前时间转化成秒数的结果。

PS：

local time2 = os.time()

传参则返回指定时间转化秒数的结果

PS：

local time = os.time({year =2016, month = 11, day =23, hour =17, min =17, sec = 00})

任何一个时区，在相同的时间，同时调用 os.time() 返回的结果都是一样的。

b. os.date()

os.date() 把时间戳转化成可显示的时间字符串。

如果服务器返回一个时间戳，客户端想要转换成可读样式，需要考虑到时区问题，因为是按照本地时区转换的，如果本地时区跟服务器所在时区不一致，就会导致时间显示错误，一般解决办法就是 加上时区差即可。

8. 简述 Lua 实现面向对象的原理

① 表 table 就是一个对象，对象具有了标识 self，状态等相关操作。
② 使用参数 self 表示方法的该接受者是对象本身，是面向对象的核心点，冒号操作符可以隐藏该 self 参数。
③ 类（Class）：每个对象都有一个原型，原型(lua 类体系)可以组织多个对象间共享行为。
④ setmetatable(A,{__index=B}) 把 B 设为 A 的原型。
⑤ 继承（Inheritance）：Lua 中类也是对象，可以从其他类（对象）中获取方法和没有的字段。
⑥ 继承特性：可以重新定义（修改实现）在基类继承的任意方法。
⑦ 多重继承：一个函数 function 用作__Index 元方法，实现多重继承，还需要对父类列表进行查找方法，但多继承复杂性，性能不如单继承，优化，将继承的方法赋值到子类当中。
⑧ 私有性（很少用）基本思想：两个表表示一个对象，第一个表保存对象的状态在方法的闭包中，第二个表用来保存对象的操作（或接口），用来访问对象本身。使第一个表完成内容私有性。

9. table 的一些重要知识点

① table 是 Lua 的一种数据结构，用于帮助我们创建不同的数据类型，如：数组、字典等；
② table 是一个关联型数组，你可以用任意类型的值来作数组的索引，但这个值不能是 nil，所有索引值都需要用[和]括起来；如果是字符串，还可以去掉引号和中括号；即如果没有[]括起，则认为是字符串索引，Lua table 是不固定大小的，你可以根据自己需要进行扩容。
③ table 的默认初始索引一般以 1 开始，如果不写索引，则索引就会被认为是数字，并按顺序自动从 1 往后编。
④ table 的变量只是一个地址引用，对 table 的操作不会产生数据影响。
⑤ table 不会固定长度大小，有新数据插入时长度会自动增长。
⑥ table 里保存数据可以是任何类型，包括 function 和 table。
⑦ table 所有元素之间，总是用逗号“，”隔开。

10. lua 中的闭包

闭包=函数+引用环境 子函数可以使用父函数中的局部变量，这种行为可以理解为闭包！

① 闭包的数据隔离，不同实例上的两个不同闭包，闭包中的 upvalue 变量各自独立，从而实现数据隔离。
② 闭包的数据共享，两个闭包共享一份变量 upvalue，引用的是更外部函数的局部变量（即 Upvlaue），变量是同一个，引用也指向同一个地方，从而实现对共享数据进行访问和修改。
③ 利用闭包实现简单的迭代器 迭代器只是一个生成器，他自己本身不带循环。我们还需要在循环里面去调用它才行。

11. Lua Table 数组与字典区分

table 是字典和数组的混合体，当没有 key 时，table 为数组，当指定具体的 key 时，table 为字典。

二十八. 工具

1. IntelliJ IDEA

1) 快捷键

快捷键	介绍
Ctrl+ F	在当前文件进行文本查找
Ctrl + R	在当前文件进行文本替换
Ctrl + Z	撤销
Ctrl + Y	删除光标所在行 或 删除选中的行
Ctrl + X	剪切光标所在行 或 剪切选择内容
Ctrl + /	注释光标所在行代码，会根据当前不同文件类型使用不同的注释符号
Ctrl + C	复制光标所在行 或 复制选择内容
Ctrl + D	复制光标所在行 或 复制选择内容，并把复制内容插入光标位置下面
Ctrl + W	递进式选择代码块。可选中光标所在的单词或段落，连续按会在原有选中的基础上再扩展选中范围
Ctrl + O	选择可重写的方法
Ctrl + I	选择可继承的方法
Ctrl + +	展开代码
Ctrl + -	折叠代码
Ctrl + [	移动光标到当前所在代码的花括号开始位置
Ctrl +]	移动光标到当前所在代码的花括号结束位置
Ctrl + Enter	智能分隔行
Ctrl + End	跳到文件尾
Ctrl + Home	跳到文件头
Ctrl + Alt + L	格式化代码，可以对当前文件和整个包目录使用
Ctrl + Alt + O	优化导入的类，可以对当前文件和整个包目录使用
Ctrl + Alt + 左方向键	退回到上一个操作的地方
Ctrl + Alt + 右方向键	前进到上一个操作的地方
Ctrl + Shift + F	根据输入内容查找整个项目 或 指定目录内文件
Ctrl + Shift + R	根据输入内容替换对应内容，范围为整个项目 或 指定目录内文件
Ctrl + Shift + J	自动将下一行合并到当前行末尾
Ctrl + Shift + Z	取消撤销
Ctrl + Shift + W	递进式取消选择代码块。可选中光标所在的单词或段落，连续按会在原有选中的基础上再扩展取消选中范围
Ctrl + Shift + N	通过文件名定位 / 打开文件 / 目录，打开目

	录需要在输入的内容后面多加一个 正斜杠
Ctrl + Shift + U	对选中的代码进行大 / 小写轮流转换
Ctrl + Shift + Enter	自动结束代码，行末自动添加分号
F2	跳转到下一个高亮错误 或 警告位置
F3	在查找模式下，定位到下一个匹配处

2) 插件

Chinese：中文语言包，使用后界面文字将变为中文。

EmmyLua：用于 Lua 开发。

Save Action：自动格式化，安装后在 Save Action 的 General 和 Formatting Actions 中勾选。

Rainbow Brackets：使用 ctrl+鼠标右键单击，使最近的括号内容获得高亮效果；alt+鼠标右键单击，使最近的括号内容外获得暗淡效果。

CodeGlance：代码迷你缩放图。

Translation：内置翻译。

CamelCase：命名风格转换插件，可以在 kebab-case，SNAKE_CASE，PascalCase，camelCase，snake_case 和空格风格之间切换。苹果快捷键为⇧+⌥+ U，windows 为 Shift + Alt +U。

3) 常规设置

a. IDEA 设置智能提示忽略大小写

b. IDEA 设置鼠标滚轮操作

c. IDEA 以新窗口的形式打开多个项目

d. IDEA 忽略某个文件或者文件夹

e. IDEA 统一编辑文件编码

f. **去掉 IntelliJ IDEA 的拼写检查**

2. EmmyLua

1) 注解功能

a. @class

EmmyLua 利用 @class 注解来模拟面向对象中的类，可以继承，可以定义字段/属性。
完整格式：--@class MY_TYPE[:PARENT_TYPE] [@comment]

b. @type

利用 @type 注解来标记目标变量的类型，以增强代码提示以及其它功能。
完整格式：---@type MY_TYPE[|OTHER_TYPE] [@comment]
应用目标

① Local 变量

PS：

```
---@type Car @instance of car
local car1 = {}

---@type Car|Ship @transport tools, car or ship. Since lua is dynamic-typed, a variable may be of different types
---use | to list all possible types
local transport = {}
```

② Global 变量

PS：

```
---@type Car @global variable type
global_car = {}
```

③ Property 属性

PS：

```
local obj = {}
---@type Car @property type
obj.car = getCar()
```

c. @alias

可以使用 @alias 将一些复杂不容易输入的类型注册为一个新的别名。
完整格式：---@alias NEW_NAME TYPE
PS：

```
---@alias Handler fun(type: string, data: any):void
```

```
---@param handler Handler
function addHandler(handler)
end
```

d. @param

利用 @param 注解来标记函数定义参数的类型，以增强代码提示以及其它功能。
完整格式：---@param param_name MY_TYPE[|other_type] [@comment]
应用目标
① 函数参数

PS：

```
---@param car Car
local function setCar(car)
    ...
end
```

PS：

```
---@param car Car
setCallback(function(car)
    ...
end)
```

② For 循环参数

PS：

```
---@param car Car
for k, car in ipairs(list) do
end
```

e. @return

利用 @return 注解来标记函数的返回值类型。
完整格式：---@return MY_TYPE[|OTHER_TYPE] [@comment]
应用目标
① 函数

PS：

```
---@return Car|Ship
local function create()
    ...
end

---Here car_or_ship doesn't need @type annotation, EmmyLua has already inferred the
type via "create" function
local car_or_ship = create()
```

PS：

```
---@return Car
```

```
function factory:create()
    ...
end
```

f. @field

利用 @field 注解来标记某个类的额外的属性（即使这个属性没有出现在代码里）。

完整格式：---@field [public|protected|private] field_name FIELD_TYPE[|OTHER_TYPE] [@comment]

应用目标

① 在 @class 注解之后

PS：

```
---@class Car
---@field public name string @add name field to class Car, you'll see it in code completion
local cls = class()
```

g. @generic

利用 @generic 注解来模拟高级语言中的 泛型

完整格式：--@generic T1 [: PARENT_TYPE] [, T2 [: PARENT_TYPE]]

应用目标

① 函数

PS：

```
---@generic T : Transport, K
---@param param1 T
---@param param2 K
---@return T
local function test(param1, param2)
    -- todo
end

---@type Car
local car = ...

local value = test(car)
```

h. @vararg

使用 @vararg 注解一个函数的不定参数部分的类型。

完整格式：---@vararg TYPE

PS：

```
---@vararg string
```

```
---@return string
local function format(...)
    local tbl = { ... } -- inferred as string[]
end
```

i. @language

可以利用 @language 的方式来标注一段文本为某种代码格式，从而可以显示高亮。

完整格式：---@language LANGUAGE_ID

PS：

```
---@language JSON
local jsonText = [[{
    "name":"Emmy"
}]]
```