Lua 学习笔记

环境安装

在 MacOS / Linux 系统上安装：

```
# download lua-5.3.0.tar.gz file from remote
# -O/--remote-name: write output to a file named as the remote file
# -R/--remote time: set the remote file's time on the local output
# tip: for all manual infos by `curl --help all`
curl -R -O http://www.lua.org/ftp/lua-5.4.6.tar.gz
# -z/--gzip/--ungzip: compress/uncompress archive
# -x/--extract: extract the file from archive file
# -f/--file: -f <filename>  Location of archive
# tip: for more manual infos by `tar --help`
tar zxf lua-5.4.6.tar.gz
cd lua-5.4.6
make macox test
make install
```

curl

`curl` 是一个非常流行的命令行工具，用于在命令行界面（CLI）中发送网络请求。它支持多种不同的协议，包括HTTP、HTTPS、FTP、SFTP等。`curl` 广泛用于网页抓取、自动化任务、系统管理和网络测试，用法详见：

```
hwte@sw demo code % curl --help
Usage: curl [options...] <url>
 -d, --data <data>          HTTP POST data
 -f, --fail                 Fail fast with no output on HTTP errors
 -h, --help <category>      Get help for commands
```

```
 -i, --include             Include protocol response headers in the
output
 -o, --output <file>       Write to file instead of stdout
 -O, --remote-name         Write output to a file named as the remote
file
 -s, --silent              Silent mode
 -T, --upload-file <file>  Transfer local FILE to destination
 -u, --user <user:password> Server user and password
 -A, --user-agent <name>   Send User-Agent <name> to server
 -v, --verbose             Make the operation more talkative
 -V, --version             Show version number and quit

This is not the full help, this menu is stripped into categories.
Use "--help category" to get an overview of all categories.
For all options use the manual or "--help all".
```

tar

`tar` 是一个非常强大的命令行工具，用于创建和管理 tar 文件，这是一种常见的文件打包和压缩格式， `tar` 命令的全称是 " `tape archive` "，它最初设计用于磁带备份，但现在广泛用于文件归档和传输：

基本用法：

1. 创建一个 tar 文件（打包文件）：
   - `-c` 表示创建一个新的 tar 文件
   - `-v` 表示显示详细的输出，包括被处理的文件列表
   - `-f` 后面紧跟的是 tar 文件的名称
   - 注：`directory_or_files` 是要被打包的目录或文件

   ```
   tar -cvf archive.tar directory_or_files
   ```

2. 查看 tar 文件的内容：
   - `-t` 表示列出 tar 文件的内容

```
tar -tvf archive.tar
```

3. 解压（提取）一个 tar 文件：
   - `-x` 表示从 tar 文件中提取文件
   - 注：解压后的文件将放在当前目录下

```
tar -xvf archive.tar
```

4. 结合 `gzip` 使用，创建一个压缩的 tar 文件（.tar.gz 或 .tgz 文件）：
   - `-z` 表示在打包的过程中使用 `gzip` 进行压缩

```
tar -cvzf archive.tar.gz directory_or_files
```

5. 解压一个压缩的 tar 文件：
   - `-z` 表示在解压的过程中使用 `gzip` 来解压缩

```
tar -xvzf archive.tar.gz
```

更多用法详见：

```
tar --help
```

make

`make test` 和 `make install` 是两个常见的命令，它们通常在 `Makefile` 中定义。`Makefile` 是一个描述如何编译和安装一个软件项目的文件，它是由 `make` 命令解释和执行的，下面分别解释这两个命令的含义和作用：

1. `make test`：
   - 含义：这个命令指示 `make` 程序运行测试套件，以确保软件在编译后能够正常工作
   - 作用：它通常会执行一组自动化测试，这些测试旨在覆盖软件的主要功能，以确保代码在编译过程中没有引入错误。测试套件可能包括单元测试、集成测试、性能测试等

- 通常情况下，`make test` 会编译测试代码（如果它们是单独的），然后运行它们。测试的结果可能会输出到标准输出或错误，或者被记录在一个日志文件中

2. `make install`：
   - 含义：这个命令指示 `make` 程序将编译好的目标文件（通常是可执行文件或库文件）安装到系统的指定位置
   - 作用：它负责将编译好的软件从构建目录复制到系统中的安装目录，这通常包括将可执行文件安装到 `/usr/bin`（或其他bin目录），将库文件安装到 `/usr/lib`（或其他lib目录），并将文档和配置文件安装到相应的位置
   - `make install` 通常需要管理员权限（root权限），因为它是将文件写入系统中的全局位置

在 `Makefile` 中，这些命令通常与特定的目标和规则相关联，例如，一个 `Makefile` 可能包含以下内容：

代码解释：在下面的例子中，`all` 目标是编译 `src/*.c` 文件并生成可执行文件 `myprogram`，`test` 目标依赖于 `all` 目标，这意味着在运行测试之前，`myprogram` 必须已经编译完成，`install` 目标则负责将编译好的 `myprogram` 安装到 `/usr/bin`，`clean` 目标是清理编译过程中产生的临时文件.

开发者或用户可以通过在终端中输入 `make test` 来运行测试，或者输入 `make install` 来安装编译好的软件，`make` 命令会根据 `Makefile` 中的规则来执行相应的任务.

```
all:
    # 编译源代码
    gcc -o myprogram src/*.c

test: all
    # 运行测试套件
    ./myprogram test/testcases.txt

install: all
    # 将可执行文件安装到指定位置
    install -D myprogram /usr/bin
    chmod +x /usr/bin/myprogram
```

```
clean:
    # 删除编译生成的文件
    rm -f myprogram
```

基本语法

在终端中启动 `交互式编程` ：

```
lua -i
```

`脚本式编程` ：

```
echo 'print('hello Lua')' >> hello.lua
print('hello Lua')
lua run hello.lua
```

或者：

```
-- write the code into file of demo.lua
# !/usr/local/bin/lua
print('hello demo.lua')
-- run it
./demo.lua
```

分析 Lua 的编译过程：

```
-- compile hello.lua and ouput result into comiledResult.out
-- tip: the file suffix of .out is binary file,
-- it's generated by comilper and linker,
-- and it's a executable file on os.
luac -o compiledResult hello.lua
-- run it
sudo ./compiledResult
```

> ❗注意：为什么在 MacOS 中无法运行 compiledResult.out ？？？

注释规则：

```
--- 这是单行注释
--[[
这是多行注释
]]
print('hello Lua')
```

全局变量：

`在 Lua 中，默认情况下，变量总是认为是全局的.`

全局变量不需要声明，给一个变量赋值后就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

```lua
-- output: nil
print(num)
num = 1
-- output: 1
print(num)
-- you just need modify the value,
-- if yo want delete the global variable
num = nil
-- ouput: nil
print(num)
```

数据类型

Lua 是动态类型语言，`变量不需要定义类型`，只需要为变量赋值即可 .

- nil : 一个无效值
- boolean : 布尔值，true or false
- number : 双精度类型的实浮点数
- string : 字符串，由单引号或双引号表示
- function : 由 C 或 Lua 编写的函数
- userdata : 任意存储在变量中的C数据结构
- thread : 执行的独立线程，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，最简单构造表达式是{}，用来创建一个空表.

使用 type 函数输出指定变量的类型 :

```
> print(type(nil))
nil
> print(type(true))
boolean
> print(type(1))
number
> print(type('hello'))
string
> print(type(print))
function
> print(type({1}))
table
```

boolean

`Lua 把 false 和 nil 看作是 false，其他的都为 true，数字 0 也为 true .`

```
-- output: all is true
if false or nil or 0 then print(' all is true') else print('false') end
```

number

Lua 默认只有一种 number 类型，即 double 双精度类型 .

注 : e ≈ 2.71828

```
> print(type(1),type(0.1),type(1e+1),type(0.1e+1),type(0.1e-01))
number  number  number  number  number
```

❗双精度类型

在编程语言中，`双精度类型（Double Precision）`是一种表示`浮点数`的类型，它比`单精度类型（Single Precision）`提供了更高的精度，**浮点数是一种用来表示小数点位置不固定的数字的格式， 通常用于科学计算、财务计算和其他需要精确表示小数的场合 .**

双精度类型的数字通常使用**64位（8个字节）**来存储，它能够表示的数值范围比单精度类型更广，并且**在处理可能非常大或非常小的数字时，或者在需要较高计算精度时，双精度类型是首选**.

在Lua中，双精度浮点数由 `number` 类型表示，不需要单独的双精度类型，Lua的 `number` 类型在内部实际上是使用双精度浮点数来表示的.

以下是一个使用Lua编程中的双精度类型的示例：

```lua
local pi = 3.14159265358979323846
local radius = 1.0
local area = pi * radius * radius

print("Area = " .. area)
```

代码解释：在这个例子中， `pi` 是一个近似的圆周率值， `radius` 是一个半径值， `area` 是通过将 `pi` 乘以 `radius` 的平方得到的面积，由于 Lua 的 `number` 类型是双精度的，我们可以准确地计算和存储这个面积值 .

在实际使用中，当需要处理财务数据、科学计算（如天文、物理学）或者任何需要精确表示小数的场合时，双精度类型都是非常适合的，例如，在游戏中，使用双精度类型可以更准确地表示角色位置和方向，在数据分析中，双精度类型可以更精确地处理统计计算.

需要注意的是，虽然 **Lua 的 `number` 类型是双精度**的，但在进行复杂的数学运算时，可能会因为浮点运算的误差而导致不准确的计算结果。因此，在进行关键的计算时，可能需要结合其他方法来确保计算的准确性 .

❗单精度类型

在编程语言中， `单精度类型（Single Precision）` 是指一种数据类型，它通常用来表示 `浮点数` ，并且具有一个较小的存储大小，通常为 **32 位或 4 个字节**，与单精度类型相对应的是 `双精度类型（Double Precision）` ，后者通常为 **64 位或 8 个字节**，**单精度类型的浮点数精度较低，但计算速度较快，因为它占用的内存较少，适合于对精度要求不高但对速度有要求的计算场景 .**

单精度类型在科学计算、图形处理、游戏开发等领域非常常见，因为这些领域通常需要大量的浮点运算，而单精度类型可以提供足够的精度同时保持较高的计算效率 .

下面是一个使用Python编程语言来演示如何处理单精度浮点数的示例 :

```
import numpy as np

# 创建一个包含单精度浮点数的数组
single_precision_array = np.array([1.2, 3.4, 5.6, 7.8], dtype=np.single)

# 计算数组中的最大值
max_single_precision_value = np.max(single_precision_array)

# 打印结果
print("The maximum value in the array is:", max_single_precision_value)
```

代码解释 : 在这个示例中，我们使用了 NumPy 库来创建一个包含单精度浮点数的数组，NumPy提供了 np.single 数据类型，它是单精度浮点数的代表，我们使用 np.max 函数来计算数组中的最大值，并打印结果 .

请注意，**Python中的标准 float 类型通常是双精度浮点数**，如果你想要使用单精度浮点数，你需要使用 NumPy 的 np.single 类型或者其他支持单精度类型的库 .

string

可用两个方括号吗 `[[]]` 来表示一块字符串 .

```
local noteAddress = [[HackOrg.com/lua/lua-learning-note.html]]
```

对一个数字字符串进行算数操作时，Lua 会尝试将这个数字字符串转换成一个数字 :

```
print(("1" + 1) * "2") -- ouput: 4
```

可用 `. .` 符号连接字符串 :

```lua
print("a".."b".."c".."d") -- ouput: abcd
```

可用 `#` 来计算字符串的长度：

```lua
print(#'Welcome Join Us: HackOrg.com') -- output: 28
```

table

在 Lua 中 table 的创建是通过 "构造表达式" 来完成的，最简单的构造表达式是 `{}` ，可用其来创建一个空表，也可以在表里面添加一些数据，直接初始化表：

❗注意：不同于其他语言的数组把 0 作为数组的初始索引，`在 Lua 里表的默认初始索引一般从 1 开始 .`

```lua
table1 = { 'name', 'age' }
table1['domain(key)'] = 'hackorg.com(value)'
for k, v in pairs(table1) do
    print(k .. ' : ' .. v)
end
```

function

递归算法示例：

```lua
function algorithm(num)
    if num == 0 then
        return 1
    else
        return num * algorithm(num - 1)
    end
end
-- output: 3 * 2 * 1 = 6
print(algorithm(3))
```

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

```
function outputTableFun(tab, formatOuputFun)
    for k, v in pairs(tab) do
        print(formatOuputFun(k, v))
    end
end

tab = { key1 = 'value1', key2 = 'value2' }

outputTableFun(tab,
    function(k, v)
        return k .. ' : ' .. v
    end
)
-- output result as following:
-- key1 : value1
-- key2 : value2
```

上述功能实现若不用匿名函数，则普通写法为：

```
function outputTableFun(tab)
    for k, v in pairs(tab) do
        formatOuputFun(k, v)
    end
end

function formatOuputFun(key, value)
    print(key .. ' : ' .. value)
end

tab = { key1 = 'value1', key2 = 'value2' }

outputTableFun(tab)
-- ouput result as following:
```

```
    -- key2 : value2
    -- key1 : value1
```

thread

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

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

userdata

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 .

变量

变量类型

Lua 变量有三种类型 :

- 全局变量 : Lua 中的变量全是全局变量，哪怕是语句块或是函数里，除非用 `local` 关键字显式声明为局部变量 .
- 局部变量 : 用 `local` 关键字显式声明的变量
- 表中的域 : . . . ? ? ?

```lua
global_var1 = 1
local local_var1 = 1

function demo()
    global_var2 = 1
    local local_var2 = 1
    global_var1 = global_var1 + 1
    print(global_var1 + global_var2) -- ouput: 3
    print(local_var1) -- ouput: 1
end

demo()
print(local_var2) --output: nil
```

赋值语句

❗注：上述代码中最后一个例子是一个常见的错误，`在 Lua 中如果对多个变量赋值必须依次对每个变量赋值 .`

```lua
a, b, c = 0
-- output: 0 nil nil
print(a, b, c)
a, b, c = 0, 1
-- ouput: 0 1 nil
print(a, b, c)
```

索引

操作 table 中索引值使用括号 `[index]` ，当索引为字符串类型时也可使用 `.index` ：

```
domain = {}
domain['join_us'] = 'hackorg.com'
domain.personal_website = 'goog.tech'
-- output: hackorg.com
print(domain['join_us'])
-- output: goog.tech
print(domain.personal_website)
```

循环

循环类型

while 循环

```
local count = 1

while count <= 2 do
    print("count: " .. count)
    count = count + 1
end
-- output result as following:
-- count: 1
-- count: 2
```

for 循环

```
for i = 1, 2 do
    print("count: " .. i)
end
-- output result as following:
-- count: 1
-- count: 2
```

```lua
local tab = { 'a', 'b', }
for key, value in pairs(tab) do
    print(key .. ' : ' .. value)
end
-- output result as following:
-- 1 : a
-- 2 : b
```

`repeat . . . until : 重复执行循环，直到指定的条件为真为止 .`

```lua
local i = 1
repeat
    print(i)
    i = i + 1
until i > 2
-- output result as following:
-- 1
-- 2
```

循环控制语句

break : 退出当前循环或语句，并开始脚本执行紧接着的语句 .

```lua
for i = 1, 5 do
    print('count: ' .. i)
    if i == 2 then
        break
    end
end
-- output result as following:
-- count: 1
-- count: 2
```

`goto : 将程序的控制点转移到一个标签处 .`

```lua

```

```lua
-- Define a label
::start::

-- Prompt the user for input
io.write("Enter 'continue' to continue or 'exit' to exit: ")
local input = io.read()

-- Check the user input
if input == 'continue' then
    -- If the input is "continue",
    -- print a message and go to the start label
    print('Continuing. . .')
    goto start
elseif input == 'exit' then
    -- If the input is "exit",
    -- print a message and exit the program
    print('Exiting. . .')
else
    -- If the input is neither "continue" nor "exit",
    -- print an error message and go to the start label
    print('Invalid input, pls try again.')
    goto start
end
```

> ❗ On Windows : F:\Lua\5.1\lua.exe: .\demo.lua:59: unexpected symbol near ':' ? ? ?

流程控制

控制结构的条件表达式结果可以是任意值，Lua 认为 false 和 `nil 为假`，true 为真 .

Lua 提供了以下控制结构语句 :

- If 语句
- if else 语句

```
local var

if var then
    print('the var is not nil which means nil is ture')
else
    print('the var is nil which means nil is false')
end

if 0 then
    print('the 0 means true')
end

-- ouput result as following:
-- the var is nil which means nil is false
-- the 0 means true
```

函数

函数定义

Lua 编程语言函数定义格式如下：

```
optional_function_scope function function_name( argument1, argument2,
argument3..., argumentn )
    function_body
    return result_params_comma_separated
end
```

- `optional_function_scope`：未设置该参数默认为全局函数，局部函数需要使用关键字 `local`

- `function_name`：函数名称
- `argument1, argument2, argument3..., argumentn`：函数参数，多个参数用逗号隔开，也可不带参数
- `fuction_body`：函数体，函数中需要执行的代码语句块
- `result_params_comma_separated`：函数返回值，多个返回值用逗号隔开，也可没有返回值

示例：

```
function max(num1, num2)
    if (num1 > num2) then
        result = num1
    else
        result = num2
    end
    return result
end
-- output: max number: 2
print('max number: ' .. max(1, 2))
```

函数作为参数

示例：

```lua
function add(num1, num2, printResult)
    result = num1 + num2
    printResult(result)
end

function printResult(result)
    print('result is: ' .. result)
end

-- output: result is: 3
add(1, 2, printResult)
```

上述代码也可以写成：

```lua
function add(num1, num2, printResult)
    result = num1 + num2
    printResult(result)
end

printResult = function(result)
    print('result is: ' .. result)
end

-- output: result is: 3
add(1, 2, printResult)
```

当然，我们也可以使用匿名函数来实现两数相加，且自定义输出函数这个功能：

```
function add(num1, num2, printResult)
    result = num1 + num2
    printResult(result)
end

-- output: result is: 3
add(1, 2,
    function(result)
        -- or: print('result is: ' .. result)
        return print('result is: ' .. result)
    end
)
```

⭐匿名函数

❗❗❗值得注意的点：`在 Lua 语言中，所有的函数都是匿名的（anonymous）`，像其他所有的值一样，函数并没有名字，当讨论函数名时，例如 print，实际上是指保存该函数的变量.
程序示例如下所示：

```
Lua 5.4.6  Copyright (C) 1994-2023 Lua.org, PUC-Rio

> a = {p = print} -- a.p 指向 print 函数
> a.p('hello lua')
hello lua

> print = math.max -- print 指向 math.max 函数
> a.p(print(1,2))
2
```

.....以上内容摘自 Lua 程序设计（第四版）.

在Lua中，匿名函数（也称为**闭包**）是一种不具名的函数，它通常作为另一个函数的返回值或者作为函数的参数，匿名函数在某些情况下非常有用，尤其是在需要创建一个函数来执行特定任务，但是这个函数只在很小的范围内使用或者不需要被显式调用的时候，以下是一些匿名函数的常用场景：

作为函数参数

作为函数参数：当一个函数接受另一个函数作为参数时，你可以在调用时创建一个匿名函数来传递给这个函数。

```
-- 一个接受函数作为参数的函数
local function apply_function(fn, arg)
    return fn(arg)
end

-- 使用匿名函数作为参数的例子
local res = apply_function(function(arg)
    -- 这里可以是一个复杂的表达式或函数体
    return arg * arg
end, 5)

-- 输出结果: 25
print(res)
```

回调函数

回调函数：当你需要提供一个函数作为回调，并且在将来某个时候执行它时，匿名函数可以很方便地实现这一点。

```
-- 一个使用回调的函数
local function do_something(callback)
    -- 做一些事情
    local result = 'some result'
    -- 执行回调
    callback(result)
end

-- 使用匿名函数作为回调的例子
do_something(function(arg)
    print('Callback received: ', arg)
end)
```

闭包

`闭包`： 当需要在函数内部创建一个闭包时，通常会使用匿名函数，闭包可以捕获外部函数的作用域中的变量，并在外部函数返回后继续使用它们 .

```
local function creates_counter()
    local count = 0
    return function()
        count = count + 1
        return count
    end
end

local counter = creates_counter()
print(counter()) -- 1
print(counter()) -- 2
```

迭代器

`迭代器` ：匿名函数可以用来创建迭代器，例如在 `for` 循环中 .

❗注意：其中用到了 `闭包` 的概念 .

```
local myTable = { 'h', 'e', 'l', 'l', 'o' }

-- 一个使用迭代器的函数
local function process_items(items, iterator)
    for index, item in iterator(items) do
        -- 处理每个item
        print(index .. ' : ' .. item)
    end
end

-- 使用匿名函数作为迭代器的例子
process_items(myTable, function(tab)
```

```
    local index = 1
    -- ❗当需要创建一个迭代器来遍历一个集合时,闭包可以用来保存迭代状态
    return function()
        local item = tab[index]
        index = index + 1
        if item == nil then
            return
        end
        -- 返回item及其索引值
        return index - 1, item
    end
end)
```

函数式编程

`函数式编程（Functional Programming）`：在 Lua 中，匿名函数支持函数式编程风格，允许创建小的、特定用途的函数，并将它们组合在一起以实现复杂的逻辑 .

```
local add = function(x, y) return x + y end

print(add(1, 2)) -- 3
```

多返回值

示例：

```
local tab = { 1, 2, 3, 4, 5, 10 }
local max_value_index = 1
local max_value = tab[max_value_index]
local function maxMun(tab)
    for index, value in ipairs(tab) do
        if value > max_value then
            max_value_index = index
            max_value = value
        end
    end
```

```
    return max_value, max_value_index
end

maxMun(tab)

-- ouput: max value is: 10 and its index is: 6
print('max value is: ' .. max_value ..
    ' and its index is: ' .. max_value_index)
```

可变参数

Lua 函数可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 `...` 表示函数有可变的参数 .

```
local function average(...)
    local result = 0
    local arg = { ... }
    for index, value in ipairs(arg) do
        result = result + value
    end
    -- or print('the length of arg is: ' .. #arg)
    print('the length of arg is: ' .. select('#', ...))
    return result / #arg
end

print('the average is: ' .. average(1, 2, 3))

-- the output result be shown as following:
-- the length of arg is: 3
-- the average is: 2.0
```

有时候我们可能需要几个固定参数加上可变参数，固定参数必须放在可变参数之前：

```lua
local function intro(name, ...)
    print(name, ...)
end

intro('googtech', 'about me: goog.tech')
intro('googtech', 'join us: hackorg.com', 2024)

-- the output result be shown as following:
-- googtech   join us: hackorg.com
-- googtech   about me: goog.tech   2024
```

通常在遍历变长参数的时候只需要使用 `{...}` ，但是变长参数可能会包含一些 `nil` ，我们常用 `select` 函数来访问变长参数：

- `select('#', ...)` ：返回可变参数的长度
- `select(n, ...)` ：返回从起点索引 n 开始到结束位置的所有索引值（value），即参数列表
- 注意：`var = select(n, ...)` 其作用为将下标为 n 所对应的 value 赋值给变量 var
- 注意：`var1, var2 = select(n, ...)` 其作用为将下标为 n 及 n + 1 所对应的 value1，value2 分别赋值给 var1，var2

```lua
local function printAll(...)
    for index = 1, select('#', ...) do
        local value = select(index, ...)
        print(index .. ' : ' .. value)
    end
end

printAll('goog.tech', 'hackorg.com')
-- ouput result be shown as below :
-- 1 : goog.tech
-- 2 : hackorg.com

local function demo(...)
```

```
    local index1_value, index2_value = select(1, ...)
    print(index1_value .. ' and ' .. index2_value)
end

demo('goog.tech', 'hackorg.com', 'aihack.com.cn')
-- output result be shown as below :
-- goog.tech and hackorg.com
```

运算符

算术运算符

相比 Java / Python，有一个值得注意的点，即 `整除运算符(lua version >= 5.3)` .

在 Lua 中 `/` 用作除法运算，计算结果包含小数部分，`//` 用作整除法运算，计算结果不包含小数部分（不会四舍五入）：

```
print(3 / 2) -- 1.5
print(3 // 2) -- 1
```

关系运算符

相比 Java / Python，有一个值得注意的点，即 `~=` ，其用于检测两个数值是否相等，若相等则返回 true .

❗注意：`~=` 仅用于检测数值是否相等

```
if 1 ~= 2 then
    -- output: 1 is not equal with 2
    print('1 is not equal with 2')
end
```

逻辑运算符

Lua 语言中常用的逻辑运算符为：and, or, not

```
local flagA = true
local falgB = false
if flagA or falgB then
    print('flagA or flagB is true.')
end
if flagA and falgB then
    print('flagA and flagB is all true.')
end
if flagA and not falgB then
    print('flagA and `not flagB` is all true.')
end

-- output be shown as following :
-- flagA or flagB is true.
-- flagA and `not flagB` is all true.
```

其他运算符

- `..`：连接两个字符串
- `#`：一元运算符，返回字符串或表的长度

```
local str = 'Lua'
print('hello' .. ' ' .. str) -- hello Lua
print(#str) -- 3
print('hello' .. ' ' .. str, #'2024') -- hello Lua   4
```

运算符优先级

运算符优先级从高到低的顺序如下：

```
^
not    - (unary)
*      /       %
+      -
..
<      >      <=     >=     ~=     ==
and
or
```

字符串

在 Lua 种字符串可以使用三种方式表示：单引号、双引号、`[[ ]]`。

```
local str1 = 'hello Lua'
local str2 = 'Join us: hackorg.com'
local str3 = [[ About me: goog.tech ]]
```

字符串常用操作

字符串长度的计算

一般若包含中文用 `utf-8.len()`，而仅包含 ASCII 字符串则用 `string.len()`

```
print(string.len('hackorg.com')) -- output: 11
print(utf8.len(' 我的青春才刚刚开始！ ')) -- output:
```

> ❗ On Windows : F:\Lua\5.1\lua.exe: .\demo.lua:87: attempt to index global 'utf8' (a nil value) ? ? ?

字符串截取

`string.sub(s, i [, j])` 方法用于截取字符串 .

- s : 要截取的字符串
- i : 截取开始的位置
- j : 截取结束位置，默认为 -1，既为最后一个字符

```
local begin_len = #'join us: '
local tail_len = #'join us: hackorg.com'
-- hackorg.com
print(string.sub('join us: hackorg.com', begin_len, tail_len))

local len = #'hackorg.com'
-- hackorg.com
print(string.sub('join us: hackorg.com', -len))
```

字符串大小写转换

```
-- ABOUT ME: GOOG.TECH
print(string.upper('about me: goog.tech'))
-- about me: goog>tech
print(string.lower('ABOUT ME: GOOG>TECH'))
```

字符串查找与反转

```
local str = 'about me: goog.tech'
local begin_index, end_index = string.find(str, 'goog.tech')
-- goog.tech
print(string.sub(str, begin_index, end_index))
-- hcet.goog :em tuoba
print(string.reverse(str))
```

字符串格式化

Lua 中可使用 `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，剩余参数是对应格式中每个代号的各种数据 . 常用格式如下：

- %c： 接受一个数字并将其转化为 ASCII 码表中对应的字符
- %d : 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十六进制数格式
- %f : 接受一个数字并将其转化为浮点数格式
- %s : 接受一个字符串并按照给定的参数格式化该字符串

```
-- a
print(string.format('%c', 97))

-- +1 -2
print(string.format("%+d %d", 1.0, -2.0))

-- 12
print(string.format("%o", 10))

-- a
print(string.format("%x", 10))

-- 5.2
-- 5.210
-- 5.2100
-- 5.2100
--  5.2100
--   5.2100
--    5.2100
print(string.format("%.1f", 5.21))
print(string.format("%5.3f", 5.21))
print(string.format("%5.4f", 5.21))
```

```lua
print(string.format("%6.4f", 5.21))
print(string.format("%7.4f", 5.21))
print(string.format("%8.4f", 5.21))
print(string.format("%9.4f", 5.21))

-- g
-- go
-- goo
print(string.format('%.1s', 'goog.tech'))
print(string.format("%.2s", 'goog.tech'))
print(string.format('%.3s', 'goog.tech'))

-- goog.tech
-- goog.tech
--  goog.tech
--   goog.tech
print(string.format('%8s', 'goog.tech'))
print(string.format('%9s', 'goog.tech'))
print(string.format('%10s', 'goog.tech'))
print(string.format('%11s', 'goog.tech'))

-- name: GoogTech and about me: https://goog.tech
print(string.format('name: %s and about me: %s',
'GoogTech', 'https://goog.tech'))

-- 2024 / 04 / 29
local year, month, day = 2024, 4, 29
print(string.format('%04d / %02d / %02d', year, month, day))
```

字符串与整数相互转换

```lua
-- convert first char into byte
-- output: 103
print(string.byte('googtech'))
-- convert second char into byte
-- output: 111
print(string.byte('googtech', 2))
-- convert last char into byte
-- output: 104
print(string.byte('googtech', -1))
-- convert ASCII into char
-- output: goh
print(string.char(103, 111, 104))
```

⭐匹配模式

在Lua中，模式匹配（Pattern Matching）通常是通过使用**正则表达式（Regular Expressions）**来实现的，Lua提供了对正则表达式的支持，你可以使用正则表达式来匹配字符串并根据**模式**提取子字符串 .

在Lua中，正则表达式的使用涉及到两个函数：`string.match` 和 `string.gmatch` .

1. `string.match` :
    - 这个函数用于在给定的字符串中查找**第一个匹配**给定模式的位置
    - 如果找到匹配，它返回一个包含所有捕获组（captured groups）的`表`，如果没有找到匹配，则返回 `nil`
    - **它通常用于查找单一匹配**

      在下面这个例子中， `^` 表示匹配字符串的开始， `%a+` 匹配一个或多个字母，所以，"Hello:World" 中的 "Hello" 被成功匹配并返回 :

```lua
local pattern = "^%a+"
local string = "Hello:World"
local matches = string.match(string, pattern)
print(matches)
-- 输出结果:
-- Hello
```

2. `string.gmatch` :
   - 这个函数用于在给定的字符串中查找**所有匹配**给定模式的子字符串
   - 它返回一个 `迭代器` ，每次调用迭代器都会返回一个匹配的子字符串
   - **它通常用于查找多个匹配**

     在这个例子中， `%a+` 匹配一个或多个字母，所以 "Hello World" 中的每个单词都被成功匹配并打印出来 :

     ```lua
     local pattern = "%a+"
     local string = "Hello World"
     for word in string.gmatch(string, pattern) do
       print(word)
     end
     -- 输出结果:
     -- Hello
     -- World
     ```

更多用法详解 : https://www.runoob.com/lua/lua-strings.html

其他常用函数

例如计算字符串长度，字符串连接，字符串复制. . .

```lua
local prefix = 'https://'
local domain_name = 'goog.tech'
-- output: about me:  https://goog.tech
print('about me: ', prefix .. domain_name)
-- ouput: 9
print(string.len(domain_name))
-- output: iii
print(string.rep('i', 3))
```

数组

Lua 中并没有专门的数组类型，而是使用一种被称为 "table" 的数据结构来实现数组的功能 .

索引键值可以使用整数表示，数组的大小不是固定的 .

值得注意的是：`Lua 中索引值默认为 1 开始.`

一维数组

```lua
-- define a array
local arr = { 2024, 1 / 4, 'hackorg.com', 'goog.tech' }

-- get the value by index
-- 0.25
print(arr[2])

-- get the array length
-- 4
print(#arr)

-- modify the value by index
arr[3] = 'join us: hackrog.com'
-- join us: hackrog.com
```

```lua
print(arr[3])

-- add a new value into array tail
arr[#arr + 1] = 'aihack.com.cn'
-- aihack.com.cn
print(arr[#arr])

-- delete the last value of array
table.remove(arr, #arr)
-- goog.tech
print(arr[#arr])

-- get all values from begin to end
-- note: the default first index is 1 in Lua
for index = 1, #arr do
    -- 2024
    -- 0.25
    -- join us: hackrog.com
    -- goog.tech
    print(arr[index])
end

-- define a array and init the first index you want
local arr2 = {}
local begin_index, end_index = -2, 2
for index = begin_index, end_index do
    arr2[index] = index * 2
    -- -4
    -- -2
    -- 0
    -- 2
    -- 4
    print(arr2[index])
end
```

多维数组

多维数组即为数组中包含数组 / 一维数组的索引键对应一个数组 .

❗注意：`io.write()` 函数的行为和 `print()` 函数类似，但它不会在输出的末尾添加换行符 .

```
local arr = {}
local rowLength, colLength = 3, 3
print('{')
for row = 1, rowLength do
    arr[row] = {}
    for col = 1, colLength do
        arr[row][col] = row * col
        io.write('{' .. arr[row][col] ..
        '=' .. row .. '*' .. col .. '}')
    end
    if row ~= rowLength then
        print('\n')
    end
end
print('\n}')
-- {
-- {1=1*1}{2=1*2}{3=1*3}

-- {2=2*1}{4=2*2}{6=2*3}

-- {3=3*1}{6=3*2}{9=3*3}
-- }
```

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 .

泛型 for 迭代器

下面这个代码示例使用了 Lua 默认提供的迭代函数 `ipairs()` .

```
local arr = { 'goog.tech', 'hackorg.com', 'aihack.com.cn' }
for key, value in ipairs(arr) do
    print(key .. ' : ' .. value)
end
-- 1 : goog.tech
-- 2 : hackorg.com
-- 3 : aihack.com.cn
```

迭代器的分类

在 Lua 中我们常常使用函数来描述迭代器，每次调用该函数就返回集合的下一个元素 .

Lua 的迭代器包含以下两种类型 :

- 无状态迭代器
- 有状态迭代器

无状态迭代器

无状态迭代器的典型例子是 `ipairs()` ，它遍历数组的每一个元素，元素的索引值需要是数值 .

这里我们来实现实现一下 `ipairs()` 函数 :

核心代码解释 :

- 当 Lua 调用 `ipairs(arr)`，他获取三个值：`迭代函数 iter`、`状态常量 arr`、`控制变量初始值 0`.
- 然后 Lua 第一次调用 iter(arr, 0) 返回 1 与 arr[1] （除非 a[1] = nil），
- 第二次迭代调用 iter(arr, 1) 返回 2 与 arr[2],
- . . .直到 arr[index] = nil，结束迭代

```
local function my_ipairs(arr)
    -- same as iter(arr, 0) in Java
    return iter, arr, 0
end

function iter(arr, index)
    index = index + 1
    local value = arr[index]
    if value then
        return index, value
    end
end

local arr = { 'goog.tech', 'hackrog.com', 'aihack.com.cn' }
for key, value in my_ipairs(arr) do
    print(key, value)
end

-- 1  goog.tech
-- 2  hackrog.com
-- 3  aihack.com.cn
```

❗区分pair()与ipair()

在Lua中，`pairs()` 和 `ipairs()` 是两个用于迭代表格（table）的方法，它们都返回一个迭代器，但是它们的使用场景和行为有一些显著的区别：

1. `pairs()`：
    - 这个方法用于迭代一个表格的所有键值对，包括**数值键**和**字符串键**

- 它返回三个值：第一个是表格的一个键，第二个是对应的值，第三个是一个布尔值，指示是否还有更多的元素可以迭代
- 它的行为是随机的，因为它是通过哈希表的内部哈希函数来决定下一个要迭代的键
- 它通常用于**不需要保持迭代顺序**的场景，或者当**表格的键不全是数值**时

```
for key, value in pairs(table) do
  -- 这里可以处理key和value
end
```

2. `ipairs()`：
   - 这个方法专门用于迭代一个表格的**数值键**和对应的值，**从 1 开始顺序迭代**
   - 它返回两个值：第一个是表格的数值键，第二个是对应的值
   - **它只适用于键从小到大 `连续排列` 的表格，并且键必须是 `数值`**
   - 如果表格中没有数值键，或者数值键不是从小到大连续的，`ipairs` 会返回一个**空迭代器**

```
for index, value in ipairs(table) do
  -- 这里可以处理index和value
end
```

总结来说，**`pairs()` 适用于任何表格的键值对迭代，而 `ipairs()` 则专门用于迭代数值键的表格**，选择使用哪一个取决于表格的结构和你的迭代需求，如果你需要保持**数值键的顺序**，并且表格的**数值键是从小到大连续**的，那么应该使用 `ipairs()`，否则，可以使用 `pairs()`.

在实际编程中，**如果你不确定表格的键类型或者它们的顺序，通常会使用 `pairs()`，因为它是通用的**，如果你知道表格的键都是**数值**，并且它们是**连续**的，那么 `ipairs()` 可以提供 `更高效` 的迭代.

有状态迭代器

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. 示例程序如下：

```
local function eleInterator(array)
    local index = 0
    local arrLength = #array
    -- 闭包匿名函数
    return function()
        index = index + 1
        if index <= arrLength then
            return array[index]
        end
    end
end

local arr = { 'goog.tech', 'hackorg.com' }

for element in eleInterator(arr) do
    -- goog.tech
    -- hackorg.com
    print(element)
end
```

⭐表

1. 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`，例如：数组、字典等 .
2. Lua table 使用关联数组，即可以用 `任意类型的值来作数组的索引`，但这个数值不能是 `nil` .
3. Lua table 是 `不固定大小` 的，即可以根据自己需要进行扩容 .
4. Lua 也是通过 table 来解决模块（module）、包（package）和对象（Object）的 .

> 例如 string.format() :
>
> - 其中 string 表示名为 string 的 table，即 table string .
> - 而 format() 为存储在 table string 中的方法 .
>
> 相关知识点详见 "模块与包" .

分类

在Lua中，数组通常称为 "表"（Table），并且可以根据**索引类型**分为两大类 : 数值数组（Numerical Table）和字符串数组（String Table）.

数值数组

```
local numbers = {10, 20, 30, 40}
print(numbers[1])  --> 10
print(numbers[4])  --> 40
```

字符串数组

```
local strings = {
  ["first_name"] = "John",
  ["last_name"] = "Doe",
  ["age"] = 30
}
print(strings["first_name"])  --> John
print(strings["age"])  --> 30
```

混合数组

注意 : 在Lua中，表的索引可以是任意的，这意味着你可以混合使用数值和字符串索引来表示不同类型的数据结构，示例程序如下所示 :

```lua
local mixed = {
  10, 20, 30,
  ["first_name"] = "John",
  [4] = "special value"
}
print(mixed[1])  --> 10
print(mixed["first_name"])  --> John
print(mixed[4])  --> special value
```

索引类型

在 Lua 中，数组可以有三种不同的索引类型，即整型索引，字符串索引以及用户自定义索引 .

整形索引

- 使用连续的整数作为索引，从1开始
- 通常用于表示一个有序的集合，类似于其他编程语言中的数组

```lua
local arr = { 1, 3, 5 }
for index = 1, #arr do
    -- 1 : 1
    -- 2 : 3
    -- 3 : 5
    print(index .. ' : ' .. arr[index])
end
```

字符串索引

❗注意：这种类型的数组通常称为 `哈希表 / 字典`

- 使用字符串作为索引
- 通常用于表示一个映射，其中每个字符串键都与一个值相关联

```lua
local arr = { ['name'] = 'googtech', ['website'] = 'goog.tech' }
for key, value in pairs(arr) do
    -- name : googtech
    -- website : goog.tech
    print(key .. ' : ' .. value)
end
```

用户自定义索引

表索引（表嵌套）的示例程序如下所示：

```lua
local tab = {
    ['key1'] = {
        ['key1-1'] = { 'key1-1-value1', 'key1-1-value2' },
        ['key1-2'] = { 'key1-2-value1', 'key1-2-value2' }
    },
    ['key2'] = {
        ['key2-1'] = { 'key2-1-value1', 'key2-1-value2' },
        ['key2-2'] = { 'key2-2-value1', 'key2-2-value2' }
    }
}

local len = 0
for index, value in pairs(tab) do
    print(index)
    for index2, value2 in pairs(value) do
        print('\t'..index2)
        for index3, value3 in pairs(value2) do
            print('\t\t'..index3..' : '..value3)
        end
    end
    len = len + 1
end

print('\nthe tab length is: ' .. len)
```

```lua
local len2 = #tab['key1']['key1-1']
print('\nthe len2 is: ' .. len2)

-- The output result be shown as following :

-- key2
--  key2-2
--    1 : key2-2-value1
--    2 : key2-2-value2
--  key2-1
--    1 : key2-1-value1
--    2 : key2-1-value2
-- key1
--  key1-1
--    1 : key1-1-value1
--    2 : key1-1-value2
--  key1-2
--    1 : key1-2-value1
--    2 : key1-2-value2

-- the tab length is: 2

-- the len2 is: 2
```

关键字索引的示例程序如下所示：

```lua
local data = {
    name = 'googtech',
    aboutMe = 'https://goog.tech',
    joinUs = 'https://hackorg.com'
}

-- name: googtech
-- about me: https://goog.tech
-- join us: https://hackorg.com
print('name: ' .. data.name .. '\n'
    .. 'about me: ' .. data.aboutMe .. '\n'
    .. 'join us: ' .. data.joinUs)
```

函数索引的示例如下所示：

```lua
local function getIndex(key)
    return 'index_' .. key
end

local arr = {
    [getIndex(1)] = 'value1',
    [getIndex(2)] = 'value2',
    [getIndex(3)] = 'value3'
}

for index, value in pairs(arr) do
    -- index_2 : value2
    -- index_1 : value1
    -- index_3 : value3
    print(index .. ' : ' .. value)
end
```

使用 table 来表示一个**二叉树**，每个表的索引对应于树的节点，而表的值则可以表示节点的值或其他相关信息，示例程序如下所示：

```
--[[
    1
   /  \
   2   3
      /
      4
]]
local root = {
    value = 1,
    left = {
        value = 2,
        left = nil,
        righ = nil
    },
    right = {
        value = 3,
        left = {
            value = 4,
            left = nil,
            right = nil
        }
    }
}

-- ouput: 4
print(root.right.left.value)
```

❗table的构造

值得注意的是：当我们初始化 tab，并将 tab 赋值给 tab_copy 后，tab 与 tab_copy 都指向同一个内存地址，如果将其中一个例如 tab_copy 设置为 nil，即删除 tab_copy，而 tab 不受其影响. 示例程序如下：

```
-- init the table and check its type
local tab = { 'goog.tech', 'hackorg.com' }
print('the type of table is: ' .. type(tab))
```

```lua
-- point to the same memory address with tab
local tab_copy = tab
print(string.format('tab addr: %s, and tab_copy addr: %s', tab,
tab_copy))

-- ouput all keys and values of tab_copy
for key, value in ipairs(tab_copy) do
    print(key .. ' : ' .. value)
end

-- update table of tab
-- NOTE: the default first index number is 1 in Lua
tab[1] = 'about me: goog.tech'
tab[2] = 'join us: hackorg.com'

-- ouput all keys and values from tab_copy
-- it's difference with above ouput infos,
-- because the tab was updated before.
for key, value in ipairs(tab_copy) do
    print(key .. ' : ' .. value)
end

-- delete the tab_copy
tab_copy = nil

-- ouput the memory address of tab_copy
-- nil will be outputed, because the tab was deleted.
print('the tab_copy addr is: ', tab_copy)

-- ouput all keys and values of tab
-- the action of deleting tab_copy isn't affect tab
for key, value in ipairs(tab) do
    print(key .. ' : ' .. value)
end
```

```
-- the output result be shown as following :
-- the type of table is: table

-- tab addr: table: 0x6000030a8800, and tab_copy addr: table:
0x6000030a8800

-- 1 : goog.tech
-- 2 : hackorg.com

-- 1 : about me: goog.tech
-- 2 : join us: hackorg.com

-- the tab_copy addr is:  nil

-- 1 : about me: goog.tech
-- 2 : join us: hackorg.com
```

❗table（不同数据类型的索引 or 不连续的整型索引）中利用 `ipairs()` 进行遍历 table 以及利用 `#` 获取 table 长度时会遇到一些类似于 bug 的问题：

```
-- init table
local tab = {}

-- nothing will be output
-- because the defeault first index number is 1 in Lua
-- and it's same design in ipairs().
tab[0] = 2024
tab['about-me'] = 'goog.tech'
for index, value in ipairs(tab) do
    print(index .. ' : ' .. value)
end

-- just output: "1:2024" and tab['about-me'] not be output
-- the reason is ipairs() just return the number type of index,
-- and it isn't return the string type of index.
-- the source code of my_ipairs() be shown as following:
```

```lua
--[[
local function my_ipairs(arr)
    -- same as iter(arr, 0) in Java
    return iter, arr, 0
end

function iter(arr, index)
    index = index + 1
    local value = arr[index]
    if value then
        return index, value
    end
end
--]]
tab[1] = 2024
tab['about-me'] = 'goog.tech'
for index, value in ipairs(tab) do
    print(index .. ' : ' .. value)
end

-- just output: "1:2024" and tab[3] not be ouput.
-- the reason pls refer the source code of my_ipairs().
tab[1] = 2024
tab[3] = 'hello lua'
tab['about-me'] = 'goog.tech'
for index, value in ipairs(tab) do
    print(index .. ' : ' .. value)
end

-- get the length of tab2 and it's will ouput: 1
-- because the default first index number is 1 in Lua,
-- and if the index number once isn't consecutive,
-- the length variable will be stop count! it's terriable all right?!
local tab2 = {}
tab2[0] = 0
tab2[1] = 1
```

```
tab2[3] = 3
print(#tab2)

-- get the length of tab2 and it's will ouput: 1
-- if the index type isn't Int, the length variable will be stop count.
local tab3 = {}
tab3[1] = 1
tab3['two'] = 2
tab3[3] = 3
print(#tab2)
```

❗注意1：

- 在Lua中，若数组只能使用从 `1` 开始的连续整数作为索引，你可以使用 table 模块中的 `ipairs` 函数来迭代这样的数组. 但是，即使是 `ipairs` 函数，也只适用于那些 `键(index)是连续整数` 的数组 .

  如果你想要一个能够处理任何索引类型的数组，那么你可能需要编写自己的迭代函数，或者使用一些第三方库，比如LuaTableLibrary，它提供了一些高级的数组处理函数，包括能够遍历 `不同索引类型` 的数组的函数 .

  总之，由于Lua数组的灵活性，遍历它们可能需要根据数组的索引类型选择不同的方法 .

❗注意2：

- Lua 中的表可以包含任何类型的键（index），而 `#` 操作符只适用于那些使用 `整数` 作为键的 `有序(不允许中断)` 表，而且其仅会返回表中 `从 1 开始` 的有效元素的个数 .

  对于哈希表或者包含非整数键的表，例如 table 中的 index 既有纯整型也有纯字符型，应该使用其他方法来获取键的个数 .

table的常用操作

连接

我们可以使用 `concat()` 函数将列表中的元素连接成字符串：

```lua
local tab = { 'a', 'b', 'c', 'd' }
print(table.concat(tab)) -- abcd
print(table.concat(tab, '-')) -- a-b-c-d
print(table.concat(tab, '-', 2, 3)) -- b-c
```

插入与移除

```lua
-- init a table
local arr = { 'goog.tech' }
-- insert new element into tail
table.insert(arr, 'aihack.com.cn')
-- insert new element by index
-- NOTE: the default first index number is 1 in Lua
table.insert(arr, 2, 'hackorg.com')
-- output the last element
print(arr[3])
-- remove the element by index
print(table.remove(arr, 2))

-- output result be shown as following :
-- aihack.com.cn
-- hackorg.com
```

排序

使用 `sort()` 中默认的比较函数，即对表中的元素进行升序排序：

```lua
local arr = { 'goog.tech', 'hackorg.com', 'aihack.com.cn' }
for key, value in pairs(arr) do
    -- 1:goog.tech
    -- 2:hackorg.com
    -- 3:aihack.com.cn
    print(key .. ':' .. value)
end
```

```
table.sort(arr)

for key, value in pairs(arr) do
    -- 1:aihack.com.cn
    -- 2:goog.tech
    -- 3:hackorg.com
    print(key .. ':' .. value)
end
```

不使用 `sort()` 中默认的比较函数，而使用自定义的比较函数，示例程序如下所示：

```
local arr = { 'goog.tech', 'hackorg.com', 'aihack.com.cn' }

-- 正向排序
table.sort(arr)
for key, value in pairs(arr) do
    -- 1:aihack.com.cn
    -- 2:goog.tech
    -- 3:hackorg.com
    print(key .. ':' .. value)
end

-- 逆向排序
table.sort(arr, function(a, b)
    return a > b
end)
for key, value in pairs(arr) do
    -- 1:hackorg.com
    -- 2:goog.tech
    -- 3:aihack.com.cn
    print(key .. ':' .. value)
end


local scores = {
    {
```

```lua
        name = 'xiaoming',
        score = 99
    },
    {
        name = 'zhangsan',
        score = 98
    },
    {
        name = 'lisi',
        score = 100
    }
}

-- 自定义比较函数
local function sortFunction(a, b)
    -- 按照表中 score 属性值进行降序排列
    if a.score > b.score then
        return true
    end
    -- 按照表中 score 属性值进行升序排列
    -- if a.score < b.score then
    --     return true
    -- end
end

-- 按照表中 score 属性值进行降序排列，故需自定义比较函数
table.sort(scores, sortFunction)

-- 输出排序后的结果
for key, value in pairs(scores) do
    -- lisi : 100
    -- xiaoming : 99
    -- zhangsan : 98
    print(value.name .. ' : ' .. value.score)
end
```

最大值

`table.maxn()` 方法在 Lua5.2 之后被废除了，接下来我们使用自定义函数来实现获取整型数组中的最大元素值及其索引，示例程序如下所示：

❗ Note : `table.maxn()` : Returns the largest positive numerical index of the given table, or zero if the table has no positive numerical indices.

```
local intArr = { [1] = 2, [2] = 4, [4] = 8 }
local max_ind = 1
local max_ele = intArr[max_ind]
local function table_max_ele()
    for index, value in pairs(intArr) do
        if value > max_ele then
            max_ind = index
            max_ele = value
        end
    end
end

table_max_ele()
-- max element is: 8 and its index is: 4
print(string.format('max element is: %s and its index is: %s',
    max_ele, max_ind))
```

长度

❗ 当我们获取 table 的长度时无论是使用 `#` 还是 `table.getn(arr)` 都会在索引中断的地方停止计数，从而导致无法正确地获得 table 的长度，其解决方案之一是我们可以通过自定义函数来代替上述方法：

```lua
arr = { [1] = 2, [3] = 4, [5] = 6, ['str_ind'] = 'str_val' }

local function get_table_len(arr)
    local length = 0
    for key, value in pairs(arr) do
        length = length + 1
    end
    return length
end

-- output : 4
print(get_table_len(arr))
```

模块与包

require 函数

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. 下面我们创建一个自定义模块 myModule.lua，文件代码内容如下 :

```lua
-- file : myModule.lua

myModule = {}

myModule.intro = "a custom module created by googtech"

-- private func
-- it's can't be call by exterior code
```

```lua
local function printResult(res)
    print('the add result is: ' .. res)
end

-- public func
function myModule.add(var1, var2)
    local result = var1 + var2
    printResult(result)
end

return myModule
```

Lua 中通过 `require` 函数来加载模块，接下来我们我在 myModuleTest.lua 中调用上述自定义模块 myModule.lua 中的变量及函数， myModuleTest.lua 中代码如下所示：

```lua
-- file : myModuleTest.lua

-- require 'myModule'
-- print(myModule.intro)
-- myModule.add(1, 2)

-- or:
-- local mm = require 'myModule'
local mm = require('myModule')
print(mm.intro)
print(mm.add(1, 2))

-- output result be shown as following :
-- a custom module created by googtech
-- the add result is: 3
```

⭐元表（Metatable）

在Lua编程中，元表（metatable）是一个与普通表类似的数据结构，但是它具有特殊的用途和行为. 元表的主要作用是提供了一个机制，允许你自定义表（或者其他值类型）的操作，当一个表被设置为一个元表后，它为这个表定义了一系列的 **"元方法"**（metamethods），这些元方法是在执行特定操作时调用的特殊函数.

元表通过Lua的 `setmetatable` 函数来设置，它接受两个参数：第一个是普通的表，第二个是元表，例如：

```lua
local myTable = { ... } -- 一个普通的表
local metaTable = { ... } -- 元表，定义了一些元方法
setmetatable(myTable, metaTable) -- 把 metaTable 设为 myTable 的元表
```

一旦设置了元表，当对原始表 myTable 进行某些操作时，Lua会检查元表 metaTable 中是否有对应的 `元方法` 来执行该操作，以下是一些常见的元方法和它们对应的操作：

- `__index`：当表的一个键被访问，但是表中没有对应的值时调用
- `__newindex`：当表的一个键被设置一个新值时调用
- `__call`：当表被当作函数调用时调用
- `__len`：当 `#` 操作符被用于获取表的长度时调用
- `__eq`：当用 `==` 操作符比较两个值时调用
- `__tostring`：当用 `tostring` 函数将值转换为字符串时调用

除了这些预定义的元方法，你还可以创建自定义的元方法来处理其他操作，元表中的元方法通常是一个函数，它接受一个参数，这个参数是表中的一个键或者值，元方法可以返回任何你想要的结果，这些结果将作为执行操作时的实际结果.

❗ **元表在 Lua 中是一个强大的工具，它允许你实现许多高级编程概念，比如 `委托(delegation)` 、 `多态(polymorphism)` 、 `自定义数据类型` 等. 通过元表，你可以有效地模拟 `类` 和 `继承` 的特性，尽管 Lua 本身是一种非常轻量级和动态的脚本语言， `不支持` 传统的面向对象编程 .**

__index程序示例

在Lua的元表（metatable）中，`__index`元方法是用来定制当对一个表进行索引（即访问不存在的键）时发生的行为，当表的一个键被访问，而表中没有对应的值时，`__index`元方法会被调用，这样，即使表中没有直接存储某个键的值，你也可以通过定义`__index`元方法来提供这个值 .

下面是一个简单的例子，展示了如何使用`__index`元方法来为表定义一个默认的值 :

代码解释 : 在下面的这个例子中，当访问`myTable`中的`a`或`b`键时，由于`myTable`中没有直接存储这些键的值，Lua会查找元表中的`__index`函数，`__index`函数检查了键是否为`'a'`或`'b'`，如果是，则返回`myTable.defaultValue`，即我们定义的默认值 0，如果键不是`'a'`或`'b'`，`__index`函数会抛出一个异常，因为在这种情况下，我们希望程序能够立即停止，并指出访问了一个不存在的键 .

请注意，`__index`元方法只有在表中没有直接存储相应的键时才会被调用，如果表中已经存储了某个键的值，元方法就不会被触发，直接返回表中存储的值 .

❗ **此外，如果你定义的表是一个用户数据（userdata）或者是一个函数，`__index`元方法也可以用来模拟`类的继承`行为，这时它通常会查询一个名为`__class`的表来获取方法和属性。这种技术被称为“模拟类继承”，是Lua中常见的设计模式之一 .**

```
local myTable = {
    ['key1'] = 'value1',
    defaultValue = 0
}

local MetaTable = {
    __index = function(table, key)
        -- 如果表中没有键 'a' 或者 'b'，返回默认值
        if key == 'a' or key == 'b' then
            return myTable.defaultValue
        end
         -- 如果表中没有这个键，抛出一个异常
        error('the key of ' .. key .. ' not found!')
    end
```

```lua
}

-- 把 MetaTable 设为 myTable 的元表
setmetatable(myTable, MetaTable)

-- 输出结果: key1's value is: value1
print("key1's value is: ", myTable['key1'])
-- 输出结果: a =  0
print('a = ', myTable.a)
-- 输出结果: b =  0
print('b = ', myTable.b)
-- 输出结果: lua: meta.lua:13: the key of c not found!
print(myTable.c)
```

__index模拟继承

在Lua中，通过使用元表（metatable）和 `__index` 元方法，可以模拟类的继承行为，这种机制允许你定义一个基础类（或称原型），并让其他对象继承这个基础类的属性和方法，下面是一个简单的例子，展示了如何使用 `__index` 元方法来模拟类的继承：

代码解释：在这个例子中，`Base` 表是基础类，它定义了一些属性和方法，**`BaseMetatable` 元表将 `Base` 作为 `__index` 元方法，这意味着如果访问 `derived` 中不存在的方法或属性，Lua会查找 `Base` 表中是否有定义**，这样，`derived` 就继承了 `Base` 的所有属性和方法。

请注意，当继承发生时，子对象（`derived`）会覆盖父对象（`Base`）中的同名属性，例如在例子中，`derived.age` 被设置为 1，这会覆盖 `Base` 中的 `age` 属性。

此外，你可以在继承的基础上添加自己的属性和方法，在例子中，`derived` 添加了 `newVariable` 属性和 `newMethod` 方法，这些新增的属性和方法不会影响基础类，也不会被基础类的方法所覆盖。

一. 创建一个 Base.lua 文件，存放基类（父类）：

```lua
-- 定义一个基础类（父类）
local Base = {}

-- 定义一些属性和方法
Base.age = 0
function Base.intro()
    print('hello I am Base')
end

-- 返回此table，以供其它模块调用
return Base
```

二. 创建一个 Derived.lua 文件，存放继承父类的子类：

```lua
-- 调用 Base 模块中的属性或方法
local Base = require('Base')

-- 定义一个继承自基础类的元表
local BaseMetaTable = {
    __index = Base
}
-- 创建一个继承自基础类的对象
local derived = setmetatable({}, BaseMetaTable)

-- 调用继承的属性和方法
print(derived.age) -- 输出结果: 0
derived:intro()    -- 输出结果: hello I am Base

-- 重写父类的属性后再次调用
derived.age = 1
print(derived.age) -- 输出结果: 1

-- 重写父类的方法后再次调用
derived.intro = function()
    print('hello I am Derived') -- 输出结果: hello I am Derived
```

```lua
end
derived:intro()

-- 定义自己的属性和方法
local newVariable = 2
local function newMethod()
    print('hello I am a new method')
end

-- 调用自己的属性和方法
print(newVariable) -- 输出结果: 2
newMethod()        -- 输出结果: hello I am a new method
```

❗注意点：上面示例程序中代码是可以简化的：

```lua
-- local BaseMetaTable = {
--     __index = Base
-- }

-- local derived = setmetatable({}, BaseMetaTable)

-- 创建一个继承自基础类的对象(下面这行代码 == 上面的两段代码)：
--[[
setmetatable 对象返回一个空表 {},
而 {} 的元表(metatable)的 __index = {age=0, intro=./Base.lua:6}
注: 6是指 intro() 这个方法在 Base.lua 文件中的位置(行号)
]]
local derived = setmetatable({}, {__index = Base})
```

__newindex程序示例

在Lua中，元方法（metamethod）是一种特殊的方法，它可以在对象上定义额外的操作，`__newindex` 就是这样一个元方法，它会在对象的一个新索引被设置时被调用，`__newindex` 元方法可以用来控制当尝试向一个 table 中添加新元素或者修改已有元素时发生的事情。

下面是一个简单的例子，展示了如何使用 `__newindex` 元方法：

代码解释：在这个例子中，当尝试将 `"new_field"` 设置为 `"new_value"` 时，`__newindex` 元方法被调用，它首先打印了一条消息，然后将值 value 赋给 table 的指定 key，**这种机制允许你在值被设置之前执行一些额外的逻辑，比如验证输入、自动转换类型或者记录日志等。**

❗请注意，当使用 `__newindex` 时，**元方法中的第一个参数总是table本身（即上述程序中的myTable），第二个参数是新增或修改的键key，第三个参数是试图设置的值value，元方法需要返回被设置的值，这样Lua才能完成正常的赋值操作。**

```lua
-- 创建一个空表
local myTable = {}

-- 设置元表，并定义 __newindex 元方法
local metaTable = {
  __newindex = function(table, key, value)
    -- 在这里处理将要设置的值
    -- lua: metaTest.lua:8: C stack overflow
    -- table[key] = value
    rawset(table, key, value .. '_tail')
    print("Setting ", key, " to ", value)
  end
}

-- 把 MetaTable 设为 myTable 的元表
setmetatable(myTable, metaTable)

-- 因 new_field 是表中的新索引
-- 故会调用 __newindex 方法来控制如何在表中设置 value
myTable.new_field = "new_value"

-- 验证 value 是否设置成功
print(myTable.new_field)

-- 程序输出结果如下所示：
-- Setting new_field  to new_value
```

```
-- new_value_tail
```

此外，如果你想要在元方法中阻止对 table 的直接赋值（比如上面的例子中，阻止 `object.new_field = "new_value"` 这样的操作），你可以写一些逻辑处理代码，下面是一个阻止赋值的例子，在这个例子中，如果尝试设置的键是 `"valid_key"`，则允许赋值，否则赋值会被阻止：

```
-- 创建一个空表
local myTable = {}

-- 设置元表，并定义 __newindex 元方法
local metaTable = {
  __newindex = function(table, key, value)
    -- 这里可以添加一些额外的逻辑
    -- 例如验证 key 是否有效
    if key == "valid_key" then
      -- table[key] = value
      rawset(table, key, value .. '_tail')
    else
      -- 阻止设置其他键
      print('the key is invalid')
    end
  end
}

-- 把 MetaTable 设为 myTable 的元表
setmetatable(myTable, metaTable)

-- 因 valid_key 是表中的新索引
-- 故会调用 __newindex 方法来控制如何在表中设置 value
myTable.valid_key = "valid_value"
-- 验证 value 是否设置成功
print(myTable.valid_key)


-- 因 others_field 是表中的新索引
```

```lua
-- 故会调用 __newindex 方法来控制如何在表中设置 value
myTable.others_field = "others_value"
-- 验证 value 是否设置成功
print(myTable.others_field)
```

❗注意：在上述程序中，如果使用 `table[key] = value` 代替 `rawset(table, key, value)`，则会抛出如下异常信息：

```
lua: metaTest.lua:7: C stack overflow
```

❗扩展：rawset()

在Lua中，`rawset()` 是一个全局函数，它用于直接修改数组的元素或表的值，**而不调用任何元方法（metamethods）**，通常，当你设置一个表的值时，**如果这个表有一个元表，并且元表中定义了相应的元方法，那么在设置值之前，元方法会被调用，但是，`rawset()` 绕过了这些元方法的执行。**

`rawset()` 函数有三个参数：需要修改的表、键（可以是数字或字符串）和新的值，下面是一个简单的例子：

```lua
local t = { 10, 20, 30 }

-- 使用 rawset 直接设置数组元素
rawset(t, 2, 100)
-- 输出结果: 100
print(t[2])
```

如果你想要设置一个表的值，而表的键是一个字符串，你也可以使用 `rawset()`：

```lua
local tab = { hello = 2023 }

-- 使用 rawset 直接设置表的值
rawset(tab, 'hello', 2024)
-- 输出结果: 2024
print(tab['hello'])
```

在Lua中，**使用 `rawset()` 通常是为了避免元方法的影响**，或者是在处理需要直接访问底层数据的库时. 然而，直接使用 `rawset()` 可能会导致代码难以维护，**因为它绕过了Lua的常规操作流程，通常，调用元方法的行为是期望的**，也是Lua设计的一部分.

请注意， `rawset()` 对于数组类型的表（**即下标从1开始连续增加的表**）和普通表都是有效的:

- 对于数组类型的表，键可以是数字
- 对于普通表，键可以是任何可以哈希的值，包括字符串和数值

__call程序示例

在Lua中，元表（metatable）的 `__call` 方法是在对一个表格执行函数调用操作时被调用的，通常，当使用 `()` 操作符来调用一个函数时，如果该函数实际上是一个table，并且这个table有一个元表，那么Lua会尝试调用元表中的 `__call` 方法。下面是一个简单的程序示例，展示了如何使用 `__call` 方法来自定义表格的行为 .

代码解释 : 在这个例子中，当调用 `function_table()` 时，实际上是调用了元表中的 `__call` 方法，在 `__call` 方法内部，你可以执行任何你想要的逻辑，例如打印信息、返回值或者执行一些业务逻辑 .

请注意， **`__call` 方法通常用于将一个table转换为一个函数对象**，这样你就可以像调用普通函数一样调用这个table，这在Lua中提供了一种动态创建函数的方式 .

```lua
-- 定义一个元表,其中包含一个__call方法
local meta = {
  __call = function(table, ...)
    -- 这里可以执行任何你想要的逻辑
```

```
    -- 例如,打印传入的参数
    print("Function called with arguments:", ...)
    -- 或者,返回一个特定的值
    return "Custom message"
  end
}

-- 创建一个具有元表的函数
local function_table = setmetatable({}, meta)

-- 调用该函数,就像调用一个普通函数一样
-- 这将调用元表中的__call方法
local msg1 = function_table()
print(msg1)

-- 调用该函数并传入参数,就像调用一个普通函数一样
-- 这将调用元表中的__call方法
local msg2 = function_table(1, 2, 3)
print(msg2)
```

__call常用情景

1. **创建函数对象**：你可以创建一个具有特定行为或状态的函数对象，并在 `__call` 方法中实现你想要的逻辑
2. **模拟类的行为**：如果你想要在Lua中模拟类的行为，可以使用 `__call` 来模拟类的构造函数。当你创建一个类的新实例时，`__call` 会被调用
3. **装饰器模式**：`__call` 可以用于装饰器模式，允许你添加额外的行为或逻辑到已有的函数上
4. **代理**：`__call` 可以用于创建代理对象，这些代理可以记录函数的调用或者在调用前执行一些前置逻辑
5. **闭包**：`__call` 可以与闭包结合使用，创建具有闭包绑定的函数对象

⭐协同程序（Coroutine）

- `一个进程由多个线程组成`，一般来说一个应用程序就是一个进程
- 协同程序和线程差不多，也是处于线程这一级别的`执行流最小单元`，并且都是由一些列代码指令组成 .
- 协同程序和线程的最大区别是：`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` .

> 注：多线程所谓的 "同时" 是指同一时间而不是同一时刻，因为实际上 CPU 在某一时刻只能执行某一条指令，所以多线程其实是指在极短时间内，多个线程来回切换以获得 CPU 使用权 .
>
> 因此多线程是人的一种宏观感受，从微观角度看，某一时刻其实还是只有一个线程的某一条指令在执行 .

Lua 将所以有关协同程序的函数放置在一个名为 "coroutine" 的 table 中 .

基本语法

- `coroutine.create()`：创建 coroutine，返回一个 thread 类型的值（可视为`协同程序的 id`），参数是一个函数
- `coroutine.status()`：查看 coroutine 的状态
- `coroutine.resume()`：启动 / 重启 coroutine
- `coroutine.yield()`：将 coroutine 设置为挂起状态
- `coroutine.wrap()`：和 create 功能重复，创建 coroutine，返回一个函数，一旦你调用这个函数，就进入 coroutine.
- `coroutine.running()`：返回当前正在运行的 coroutine 的线程号，一个 coroutine 就是一个线程

create函数

函数 `create()` 用于创建协同程序，`它只有一个参数，就是一个函数`，该函数的代码就是协同程序所需执行的内容 .

create() 会返回一个 thread 类型的值（可以将其视为 `协同程序的 id`），用于表示该协同程序，通常 create() 的参数是一个 `匿名函数` .

```
local co = coroutine.create(function()
    print('hello coroutine')
end)

-- ouput: thread: 0x600000e44008
print(co)
```

疑问点：为什么协同程序中的 `print('hello coroutine')` 代码没有被执行呢？？？

答：因为一个协同程序可以处于 4 种不同的状态，即 `挂起（suspended）` 、 `运行（running）` 、 `死亡（dead）` 和 `正常（no rmal）` ，当创建一个协同程序时，它处于挂起状态，也就是说，协同程序不会在创建它时自动执行其内容 .

status函数

我们可以通过 `status()` 方法来检查协同程序的状态：通过下面的程序示例我们可以看到，刚刚创建好的协同程序的状态为 suspended 挂起状态 .

```
local co = coroutine.create(function()
    print('hello coroutine')
end)

-- thread: 0x6000000ec008 : suspended
print(co, ' : ' .. coroutine.status(co))
```

疑问点：那么我们怎么能让我们的协同程序运行起来呢？也即怎么才能运行协同程序中的内容呢，例如上述协同程序中的 `print('hello coroutine')...`

答：详见下面新知识点...

resume函数

我们可以通过调用 `resume()` 函数来 `启动或重启` 一个协同程序，并将其状态由 `挂起态` `->` `运行态` ，

当协同程序中的内容执行完毕后，其状态会由 `运行态` `->` `死亡态` .

```
local co = coroutine.create(function()
    print('hello coroutine')
end)

print('whether the coroutine be resumed successfully: ',
    coroutine.resume(co))
print(co, ' : ' .. coroutine.status(co))

-- output result be shown as following :
-- hello coroutine
-- whether the coroutine be resumed successfully: true
-- thread: 0x60000228c008  : dead
```

❗注意点

resume() 是在 `保护模式` 中运行的，因此，**如果一个协同程序在执行中发生任何错误，Lua 是不会显示错误消息的**，而是将执行权返回给 resume() 调用 .

通过运行下面示例程序，我们发现 Lua 并没有输出错误信息，因此这一点也在提醒着我们在实际开发中使用协同程序一定要慎重，因为一旦协同程序中出现错误并不好定位 .

```lua
-- output: attempt to concatenate a nil value
-- print('hello' .. nil)

-- ouput: nothing...(no error info)
local co = coroutine.create(function()
    print('hello' .. nil)
end)
coroutine.resume(co)
```

yield函数

到目前为止，协同程序看上去还只是像一种复杂的函数调用方法，**其实协同程序真正强大之处在于函数 yield() 的使用上**，该函数可以让一个运行中（running）的协同程序挂起（suspended），而之后可以再恢复它的运行：

```lua
local co = coroutine.create(function()
    for i = 1, 3 do
        print('hello coroutine : ' .. i)
        coroutine.yield()
    end
end)

-- hello coroutine : 1
-- resume successfully ? :  true
-- thread: 0x600002958008 :suspended
print('resume successfully ? :', coroutine.resume(co))
print(co, ':' .. coroutine.status(co))

-- hello coroutine : 2
-- resume successfully ? :  true
-- thread: 0x600002958008 :suspended
print('resume successfully ? :', coroutine.resume(co))
print(co, ':' .. coroutine.status(co))

-- hello coroutine : 3
```

```
-- resume successfully ? :  true
-- thread: 0x600002958008 :suspended
print('resume successfully ? :', coroutine.resume(co))
print(co, ':' .. coroutine.status(co))

-- resume successfully ? :  true
-- thread: 0x600002958008 :dead
print('resume successfully ? :', coroutine.resume(co))
print(co, ':' .. coroutine.status(co))

-- resume successfully ? :  false cannot resume dead coroutine
-- thread: 0x600002958008 :dead
print('resume successfully ? :', coroutine.resume(co))
print(co, ':' .. coroutine.status(co))
```

从上面程序的运行结果可以得到：

- 第 4 次调用 resume() 函数来启动 / 重启协同程序时，协同程序中的打印函数并没有被执行，这是因为此时 i = 4，也即协同程序中的 for 循环在进行最后一次逻辑判断，然后发现不满足执行打印函数的条件（i <= 3）.

  但是调用 coroutine.resume(co) 所返回的结果依旧是 true，这是因为毕竟协同程序还没有彻底执行完毕，即协同程序还未处于死亡态（dead）.
- 第 5 次调用 resume() 函数来启动 / 重启协同程序时，其返回结果是一行错误信息，即 "false cannot resume dead coroutine"，因为此时协同程序已彻底执行完毕，即协同程序处于死亡态（dead），故无法被唤醒

wrap函数

```lua
local co = coroutine.wrap(function(a, b) -- a='Lua', b=nil
    print('hello ' .. a)
end)

-- ouput: function: 0x600000e2cd20
print(co)

-- ouput: hello Lua
co('Lua') -- a='Lua'
```

running函数

```lua
co = coroutine.create(function()
    print('the status of coroutine is: ',
        coroutine.status(co))
end)

-- output: the status of coroutine is: running
coroutine.resume(co)
```

数据传递

- 一：Lua 的协同程序还有一项有用的机制，即**可以通过一对 `resume-yield` 来交换数据**.

  一般调用一个 yield() 挂起协同程序后都会有一个 resume() 恢复协同程序与之对应，

  但在第一次调用 resume() 时，并没有对应的 yield 在等待它，因此所有传递给 resume() 的额外参数都将视为协同程序主函数的参数，示例程序如下所示：

  ```lua
  local co = coroutine.create(function(a, b, c) -- a=1, b=2, c=3
      print('hello coroutine: ' .. a .. b .. c) -- a=1, b=2, c=3
  end)

  -- output: hello coroutine: 123
  coroutine.resume(co, 1, 2, 3, 4) -- a=1, b=2, c=3, d=4
  ```

- 二：在 resume() 返回的内容中，第一个返回值为布尔值 true / false，表示启动 / 重启协同程序成功 / 失败，

  而后面的所有返回值都是对应 yield() 传入的参数（当第一次调用 resume 时，并没有对应的 yield 在等待它，此时所有传递给 resume 的额外参数都将视为协同程序主函数的参数，详见上面那个示例）. 示例程序如下所示：

```
local co = coroutine.create(function(a, b, c, d) -- a=1, b=2, c=nil, d=nil.
    coroutine.yield(a + b, a - b) -- a+b=1+2=3, a-b=1-2=-1
end)

-- output: true 3 -1
print(coroutine.resume(co, 1, 2)) -- a=1, b=2
```

- 三：在 二 中，我们说 **resume() 的返回值就是对应 yield() 额外传入的参数**，

  那么相反也成立：**yield() 的返回值就是对应 resume() 额外传入的参数**. 程序示例如下所示：

```
local co1 = coroutine.create(function(a, b, c) -- a=nil, b=nil, c=nil
    print('hello coroutine: ' .. coroutine.yield())
end)
-- ouput: nothing...
coroutine.resume(co1)

local co2 = coroutine.create(function(a, b, c) -- a=1, b=nil, c=nil
    print('hello coroutine: ' .. coroutine.yield())
end)
-- output: hello coroutine: 1
coroutine.resume(co2, 1)
```

- 四：**当一个协同程序结束时， resume() 的返回值均为主函数所返回的值**.

  示例程序如下所示：

```
local co = coroutine.create(function()
    return 1, 2
end)

-- ouput: true  1 2
print(coroutine.resume(co))
```

- 五：区分 三 与 四，通过运行下面的示例程序，我们发现，第 8 条和第 10 条并不矛盾：
  1. 当协调程序里面有 yield() 时，调用 resume() 会在 yield() 处挂起，并且 yield() 传入的参数会作为此次 resume() 的返回值 .
  2. 当协同程序中调用 return 执行完毕后，此次恢复该协同程序的 resume() 会把 return 的返回值作为自己的返回值 .
  3. 当第三次调用 resume() 时，其返回值为 false 及一条错误信息，因为此时协同程序已彻底运行完毕，即协同程序处于死亡态（dead），故无法再次重启协同程序 .

> 三：**resume() 的返回值就是对应 yield() 额外传入的参数**
>
> 四：**当一个协同程序结束时， resume() 的返回值均为主函数所返回的值**

```
local co = coroutine.create(function(a, b, c) -- a=1, b=2, c=nil
    coroutine.yield(a + b, a - b) -- a+b=3, a-b=-1
    return a * b, a // b -- a*b=2, a//b=0
end)

-- output: true 3 -1
print(coroutine.resume(co, 1, 2)) -- a=1, b=2
-- output: true 2 0
print(coroutine.resume(co))
-- ouput: false cannot resume dead coroutine
print(coroutine.resume(co))
```

4种状态总结

注：当一个协同程序 A 唤醒另一个协同程序 B 时，协同程序 A 就处于一个特殊状态，即不是 `挂起状态（suspended）`（无法继续 A 的执行），也不是 `运行状态（running）`（此时是 B 在运行），所以将这个状态称为 `正常态（normal）`.

```
-- output coroutine status
local function getCorStatus()
    print('the status of co1 is: ',
        coroutine.status(co1))
end

-- create the first coroutine by create() method
co1 = coroutine.create(function()
    -- co1 status: running
    getCorStatus()
    -- resume the second coroutine
    co2()
end)

-- create the second coroutine by wrap() method
co2 = coroutine.wrap(function()
    -- co1 status: normal
    getCorStatus()
end)

-- co1 status: suspended
getCorStatus()

-- resume the first coroutine
-- co1 status: running
coroutine.resume(co1)

-- co1 status: dead
getCorStatus()

-- the output result be shown as following :
-- the status of co1 is:  suspended
```

```
-- the status of co1 is:  running
-- the status of co1 is:  normal
-- the status of co1 is:  dead
```

生产者-消费者问题

```
-- the coroutine's name
local productor_consumer

function productor()
    local product_id = 0
    while true do
        product_id = product_id + 1
        -- send the product_id to consumer
        send(product_id)
    end
end

function send(pid)
    print('Productor sent the product_id: ' .. pid)
    -- suspend the coroutine named productor_consumer after
    -- sent the pid(product_id) to resume()
    -- NOTE: the param of pid(product_id) will return to resume()
    coroutine.yield(pid)
end

function consumer()
    while true do
        -- get product_id from productor
        local product_id = receive()
        print('Consumer received the product_id: ' .. product_id ..
'\n')
        -- sleep one second, because the running speed is so fast.
        -- NOTE: the method can be run on linux/macos.
        os.execute('sleep 2')
    end
```

```
    end

    function receive()
        -- resume the coroutine named productor_consumer
        -- status: whether the coroutine be resumed successfully
        -- product_id_from_yield: this param come from coroutine.yield(pid)
        local status, product_id_from_yield =
    coroutine.resume(productor_consumer)
        if status == true then
            return product_id_from_yield
        end
    end

    -- create a coroutine named productor_consumer.
    -- productor is a method name, and this method will be executed when
    -- the method of coroutine.resume(productor_consumer) be called.
    productor_consumer = coroutine.create(productor)
    -- run consumer() method
    consumer()
```

上述程序运行效果如下所示（有彩蛋的🥚哟(•ˇ∀ˇ•)~）：

```
    hwte@sw demo code % lua demo.lua
    Productor sent the product_id: 1
    Consumer received the product_id: 1

    Productor sent the product_id: 2
    Consumer received the product_id: 2

    Productor sent the product_id: 3
    Consumer received the product_id: 3

    . . . . . .
```

线程与协同程序的区别

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 .

主要区别如下 :

- 调度方式 : 线程通常由操作系统的调度器进行抢占式调度，操作系统会在不同线程之间切换执行权 .

  而**协同程序是非抢占式调度的，它们由程序员显式地控制执行权的转移 .**
- 并发性 : 线程是并发执行的，多个线程可以同时运行在多个处理器核心上，或通过时间片轮转在单个核心上切换执行 .

  协同程序则是协作式的，只有一个协同程序处于运行状态，其他协同程序必须等待当前协同程序主动放弃执行权 .
- 内存占用 : 线程通常需要独立的堆栈和上下文环境，因此线程的创建和销毁会带来额外的开销.

  而**协同程序可以共享相同的堆栈和上下文，因此创建和销毁协同程序的开销较小 .**
- 数据共享 : 线程之间可以共享内存空间，但需要注意线程安全性和同步问题 .

  协同程序通常通过 参数传递和返回值(详见数据传递知识点中的一二三四五) 来进行数据共享，不同协同程序之间的数据隔离性较好 .
- 调度和错误处理 : 线程通常在调试和错误处理方面更复杂，因为多个线程之间的交互和并发执行可能导致难以调试的问题. 协同程序则在调试和错误处理方面相对简单，因为它们是由程序员显式地控制执行流程的 .

总体而言，线程适用于需要并发执行的场景，例如在多核处理器上利用并行性加快任务的执行速度 .

而协同程序适用于需要协作和协调的场景，例如状态机、事件驱动编程或协作式任务处理 . . .

文件处理

Lua I/O 库用于读取和处理文件，分为简单模式和完全模式 .

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

  > 注 : 在简单模式中读文件时，需调用 `io.input(fileObj)`，写文件时需调用 `io.output(fileObj)`

- 完全模式（complete model）使用外部的文件句柄实现，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 .

mode值

- `r` : 以**只读**方式打开文件，该文件必须存在
- `r+` : 与 `r` 类似，但此文件可读可写
- `w` : 打开**只写**文件，若文件存在则清空内容，**若文件不存在则建立该文件**
- `w+` : 与 `w` 类似，但此文件可读可写
- `a` : 以 附加 的方式打开**只写**文件，若文件不在则新建，**若文件存在，写入的数据会被加到文件尾**（EOF符保留）
- `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
- `+` : + 号表示对文件既可以读也可以写

简单模式

io.read()中的参数

- `"*n"`：读取一个数字并返回它
- `"*a"`：从当前位置读取整个文件
- `"*l"`：读取下一行（默认）
- `"number"`：返回一个指定字符串个数的字符串，或者 EOF 时返回 nil

从文件中读数据

一. 读取文件中的第一行信息并输出：

```
local file = io.open('gt.md', 'r')
io.input(file)
print(io.read())
io.close(file)
```

二. 读取文件中的前 3 行并输出：

```
local file = io.open('gt.md', 'r')
io.input(file)
for n = 1, 3 do
    print(io.read("*l"))
end
io.close(file)

-- or :

local lcount = 0
for line in io.lines('gt.md') do
    lcount = lcount + 1
    if lcount <= 3 then
        print(line)
    else
        break
    end
```

```
end
```

三. 读取文件中的所有信息并输出：

```
local file = io.open('gt.md', 'r')
io.input(file)
print(io.read("*a"))
io.close(file)
```

四. 逐行读取文件中的所有信息并输出：

```
for line in io.lines('gt.md') do
    print(line)
end
```

五. 从终端读取用户输入的每行信息并输出：

```
for line in io.lines() do
    -- action you want..eg.print it.
    print(line)
end
```

向文件中写数据

一. 将数据写入到文件的最后一行：

```
file = io.open('gt.md', 'a')
io.output(file)
io.write('\nthis is last line')
io.close(file)
```

二. 将用户从终端输入的每行数据追加写入到文件中：

```
local file = io.open('gt.md', 'a')
io.output(file)
for line in io.lines() do
    if line == 'exit' then
        break
    end
    io.write('\nfrom terminal: ' .. line)
end
io.close(file)
```

io中的其它方法

- `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
- `io.type(file)`：检测 file 是否是一个可用的文件句柄
- `io.flush()`：向文件写入缓冲中的所有数据
- `io.lines(optional file name)`：返回一个迭代函数，每次调用将获得文件中的第一行内容 .

复杂模式

file:read()中的参数

源码文档如下所示（摘自 io.lua）：

```
--- Reads the file `file`, according to the given formats, which specify
--- what to read. For each format, the function returns a string or a
number
--- with the characters read, or **nil** if it cannot read data with the
--- specified format. (In this latter case, the function does not read
--- subsequent formats.) When called without parameters, it uses a
default
--- format that reads the next line (see below).
----
--- The available formats are:
```

```
--- **"n"**: reads a numeral and returns it as a float or an integer,
following
--- the lexical conventions of Lua. (The numeral may have leading spaces
and a
--- sign.) This format always reads the longest input sequence that is a
valid
--- prefix for a numeral; if that prefix does not form a valid numeral
(e.g., an
--- empty string, "`0x`", or "`3.4e-`"), it is discarded and the format
returns
--- **nil**;
--- **"a"**: reads the whole file, starting at the current position. On
end of
--- file, it returns the empty string;
--- **"l"**: reads the next line skipping the end of line, returning
**nil** on
--- end of file. This is the default format.
--- **"L"**: reads the next line keeping the end-of-line character (if
present),
--- returning **nil** on end of file;
--- *number*: reads a string with up to this number of bytes, returning
**nil**
--- on end of file. If `number` is zero, it reads nothing and returns an
--- empty string, or **nil** on end of file.
---@return string|number
-- [`View online doc`](https://www.lua.org/manual/5.4/manual.html#pdf-
file:read)  |  [`View local doc`](command:extension.luahelper.doc?["en-
us/54/manual.html/pdf-file:read"])
function file:read(...) end
```

从文件中读数据

一. 读取文件的第一行数据并输出：

```
file = io.open('gt.md', 'r')
print(file:read())
file:close()
```

二. 读取文件中的所有信息并输出：

```
file = io.open('gt.md', 'r')
print(file:read("*a"))
file:close()
```

三. 逐行读取文件中的所有信息并输出：

```
file = io.open('gt.md', 'r')
-- file:lines() :
-- Returns an iterator function that,
-- each time it is called, reads the file
-- according to the given formats.
-- When no format is given, uses "l" as a default.
for line in file:lines() do
    print(line)
end
-- output: whether the file be closed: closed file
file:close()
```

四. 读取文件中的前 3 行并输出：

```
file = io.open('gt.md', 'r')
for n = 1, 3 do
    print(file:read("*l"))
end
file:close()

-- or :
```

```
file = io.open('gt.md', 'r')
local lcount = 0
for line in file:lines() do
    lcount = lcount + 1
    if lcount <= 3 then
        print(line)
    else
        break
    end
end
file:close()
```

五. 从当前位置（即倒数第 20 个位置）开始读取文件内容直至末尾，并输出：

```
file = io.open('gt.md', 'r')
file:seek("end", -20)
print(file:read("*a"))
file:close()
```

向文件中写数据

一. 将数据追加写到文件的尾部：

```
file = io.open('gt.md', 'a')
file:write('\n\nhello 2024')
file:close
```

二. 将用户从终端输入的每行数据追加写入到文件中（**依旧需要用到 `io.lines()` 方法**）：

```
file = io.open('gt.md', 'a')
for line in io.lines() do
    if line == 'exit' then
        break
    end
    file:write('\nfrom terminal: ' .. line)
end
file:close()
```

file中的其它方法

- `file:seek(optional whence, optional offset)`：按字节设置当前文件指针位置，成功则返回文件指针位置，失败则返回 nil 加错误信息 . offset 默认为 0，whence 的值如下所示 :
  - "set" : 从文件头开始
  - "cur" : 从当前位置开始（默认）
  - "end" : 从文件尾开始，如 file:seek("end") 定位到文件尾部并返回文件大小 .
- `file:flush()`：向文件写入缓冲中的所有数据
- `file:lines(optional file name)`：打开指定文件（ file name）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. ❗ **Unlike `io.lines`, this function does not close the file when the loop ends .**

错误处理

错误类型

语法错误

语法错误通常是对程序的运行符，表达式使用不当引起的.（现在VSCode这么强大，几乎不会遇到此类错误吧. . .）

```
-- hwte@sw demo code % lua demo.lua
-- lua: demo.lua:1169: syntax error near '=='
syntax_error == 1
```

运行错误

运行错误是指可以成功编译，但程序运行时会输出报错信息 .

```
-- hwte@sw demo code % lua demo.lua
-- lua: demo.lua:1174: attempt to perform arithmetic on a nil value
-- stack traceback:
-- demo.lua:1174: in main chunk
-- [C]: in ?
print(1 + nil)
```

错误处理

assert函数

在Lua中， `assert` 函数是一个用于检查条件是否为真的实用函数。如果条件不为真， `assert` 函数会抛出一个错误，并给出一个错误信息。 `assert` 函数的语法如下：

```
assert(condition, message)
```

- `condition` 是一个布尔表达式，表示要检查的条件 .
- `message` （可选）是一个字符串，表示当条件不成立时输出的错误信息。如果省略了 `message` ， `assert` 会使用一个默认的错误信息 .

注意点：在实际应用中， `assert` 函数通常用于**调试和测试阶段**，以确保程序中的条件是预期的，**并且在生产环境中可能会被移除**，以避免潜在的运行时错误。这是因为 `assert` 函数会导致程序在错误发生时终止，这在生产环境中通常不是理想的 .

下面是一个简单的示例，展示了如何使用 `assert` 来确保一个函数参数是非空的：

```lua
local function process_data(data)
    assert(data ~= nil, "data must be provided")
    print('Process the data...\n')
end

process_data("some data")
process_data()

-- output result be shown as following :
-- hwte@sw demo code % lua demo.lua
-- Process the data...

-- lua: demo.lua:1184: data must be provided
```

error函数

在Lua中，`error`函数是一个内置函数，用于在运行时抛出错误. 它的基本语法如下：

```lua
error(message [, level])
```

其中，`message`是错误信息，`level`是错误级别，默认值是 1.

错误级别可以用来区分不同的错误类型. 当错误发生时，Lua的解释器会停止执行，并显示错误信息.

> - Level=1 [默认] : 为调用error位置（文件+行号）
> - Level=2 : 指出哪个调用error函数的函数
> - Level=0 : 不添加错误位置信息

注意点：实际应用中，`error`函数通常与条件语句结合使用，以确保在特定条件下抛出错误，以防止程序进入无效状态 .

下面是一个简单的示例，展示了如何使用`error`函数：

```
```

```lua
local function add(num1, num2)
    if not num1 or not num2 then
        error("Please provide two numbers to add!")
    end
    return num1 + num2
end

print(add(1, 2))
print(add(1))

-- output resule be shown as following :
-- hwte@sw demo code % lua demo.lua
-- 3
-- lua: demo.lua:1194: Please provide two numbers to add!
```

pcall函数

在Lua中，`pcall`（Protected Call）函数用于执行一段代码，并**在执行期间保护程序不受错误的影响**. 当一个错误发生时，`pcall`会捕获这个错误，并返回一个错误代码。这使得程序员可以在错误发生时选择如何处理错误，而不是让程序崩溃 .

`pcall`函数的基本语法如下：

```lua
pcall(f, ...)
```

其中，`f`是将被执行的函数，`...`是传递给`f`的参数. 如果`f`执行成功，`pcall`将返回一个值为 true 的布尔值，如果`f`执行时发生错误，`pcall`将返回一个错误代码 .

注意点：在实际应用中，**`pcall`常用于保护程序免受恶意代码的影响**，或者在不需要错误传播的场合下处理错误 .

在下面的示例中，我们有一个`divide`函数，它执行除法操作. 如果除以零，函数会引发一个错误. 但是我们使用`pcall`来调用`divide`函数，这样即使在除以零时，程序也不会崩溃，而是会捕获错误并打印出错误信息 .

```lua

```

```lua
local function divide(a, b)
    if b == 0 then
        error("division by zero")
    end
    return a / b
end

-- Result: 2.0
local success, result = pcall(divide, 10, 5)
if success then
    print("Result: ", result)
else
    print("Error: ", result)
end

-- Error: demo.lua:1206: division by zero
local success, result = pcall(divide, 1, 0)
if success then
    print("Result: ", result)
else
    print("Error: ", result)
end
```

xpcall函数

在Lua中，`xpcall`函数是一个异常处理函数，它允许在运行时捕获错误并执行清理操作. `xpcall`函数有两个参数：第一个参数是一个函数，即你想要保护的代码块. 第二个参数是一个函数，用于处理任何抛出的错误. 在Lua 5.1及更高版本中，`xpcall`函数是异常处理机制的一部分.

xpcall 函数的基本语法如下：

```lua
xpcall(f, errorhandler, arg1, arg2, ...)
```

- `f` 是一个函数，它将被调用，并且它的错误将传递给 `errhandler` .

- `errhandler` 是一个函数，它将在 `f` 抛出错误时被调用 .
- `arg1, arg2, ...` : 传递给 `f` 的参数 .

❗注意点 : 这个函数类似于pcall，不同之处在于它设置了一个新的消息处理程序（即 errhandler 函数）.

在这下面这个例子中 :

- 如果 `f` 执行时抛出错误， `errhandler` 将会被调用，并打印错误信息，其中 `err` 变量将会包含错误信息 .

  而 `xpcall` 的返回值 `success` 将会是 `false` .
- 如果 `f` 执行成功， `err` 将会是 `nil` ，而 xpcall 的返回值 `success` 将会是 `true` .

```
local function f()
    error("This is an error")
end

local function errhandler(err)
    print("An error occurred:", err)
end

local success, err = xpcall(f, errhandler)

if success then
    print("f() executed successfully")
else
    print("An error was caught by errhandler")
end

-- hwte@sw demo code % lua demo.lua
-- An error occurred: demo.lua:1253: This is an error
-- An error was caught by errhandler
```

```
local function f(num1, num2)
```

```
    print('add result: ' .. num1 + num2)
    -- error("This is an error")
end

local function errhandler(err)
    print("An error occurred:", err)
end

local success, err = xpcall(f, errhandler, 1, 2)

if success then
    print("f() executed successfully")
else
    print("An error was caught by errhandler")
end

-- the output result be shown as following :
-- hwte@sw demo code % lua demo.lua
-- add result: 3
-- f() executed successfully
```

知识点总结

- `assert`：用于在条件为假时引发错误（通常用于调试和测试阶段）.
- `error`：直接引发错误并停止执行 .
- `pcall`：用于包装另一个函数的调用，并尝试处理任何抛出的错误 .
- `xpcall`：类似于 `pcall`，不同之处在于它设置了一个新的消息处理程序（即 errhandler 函数）.

调试(Debug)

姐妹，VSCode 宇宙第一好用的编码工具了解一下. . .

推荐一个 vscode lua-debug extension : `Extension ID : actboy168.lua-debug`

垃圾回收

Lua 是一种轻量级、可嵌入的脚本语言，它的垃圾回收机制是一种基于对象的引用计数（Reference Counting）机制，即每个对象都包含一个与之关联的计数器，这个计数器记录了该对象被引用的次数，当一个对象被创建时，它的引用计数器被设置为 1，当另一个变量引用该对象时，它的引用计数器加 1，当一个变量停止引用一个对象时，该对象的引用计数器减 1，当一个对象的引用计数器达到 0 时，说明没有其他变量引用这个对象，那么这个对象就可以被垃圾回收了 .

垃圾回收函数

直接看源码文档吧（摘自 basic.lua ）：

注意点 : 如果没有提供参数， `collectgarbage()` 会使用其默认值，这个默认值在Lua 5.3及其之前的版本中是 `"stop"` ，表示停止正在执行的垃圾回收过程，在Lua 5.3之后，这个默认值被改为 `"collect"` ，表示开始一次垃圾回收循环 .

```
---@alias gcoptions
---|>"collect"      # Performs a full garbage-collection cycle.
---| "stop"         # Stops automatic execution.
---| "restart"      # Restarts automatic execution.
---| "count"        # Returns the total memory in
Kbytes(KB,1MB=1024KB,1KB=1024B).
---| "step"         # Performs a garbage-collection step.
---| "isrunning"    # Returns whether the collector is running.
---| "incremental"  # Change the collector mode to incremental.
---| "generational" # Change the collector mode to generational.
```

```
---
---This function is a generic interface to the garbage collector. It
performs different functions according to its first argument, `opt`.
---
---[View documents](command:extension.lua.doc?["en-
us/54/manual.html/pdf-collectgarbage"])
---
---@param opt? gcoptions
---@param ... any
---@return any
function collectgarbage(opt, ...) end
```

程序示例如下所示：

```
local a = { x = 10 }
print("Memory usage: ",
    collectgarbage("count") .. ' KB\n')

local b = a.x
print("Memory usage: ",
    collectgarbage("count") .. ' KB\n')

a.x = nil
print("Memory usage: ",
    collectgarbage("count") .. ' KB\n')

collectgarbage()
print("Memory usage after garbage collection:",
    collectgarbage("count") .. ' KB')
print("Memory usage after garbage collection:",
    (collectgarbage("count") / 1024) .. ' MB')


-- the output result be shown as following :
-- Memory usage:  21.7421875 KB
```

```
-- Memory usage:  21.814453125 KB

-- Memory usage:  21.890625 KB

-- Memory usage after garbage collection: 21.3125 KB
-- Memory usage after garbage collection: 0.020876884460449 MB
```

对比Py的内存使用

让我们来康康相同的程序， Lua 与 Python 的内存使用事情吧！

Python 的示例程序如下所示：

```python
# Python 3.12.3
import resource


def algorithm(num):
    if num <= 1:
        return 1
    else:
        return num * algorithm(num - 1)

print(algorithm(3))

bytes = resource.getrusage(resource.RUSAGE_SELF).ru_maxrss

print(f'{bytes / 1024} KB')
print(f'{bytes / (1024 * 1024)} MB')

# 6
# 9680.0 KB
# 9.453125 MB
```

Lua 的示例程序如下所示：

```lua
-- Lua 5.4.6  Copyright (C) 1994-2023 Lua.org, PUC-Rio
function algorithm(num)
    if num <= 1 then
        return 1
    else
        return num * algorithm(num - 1)
    end
end

print(algorithm(3))

kbytes = collectgarbage('count')

print(kbytes .. ' KB')
print((kbytes / 1024) .. ' MB')

-- 6
-- 21.7607421875 KB
-- 0.02125072479248 MB
```

通过上述两个程序示例的运行结果可知，两者内存占用情况相差 400 多倍！

这也正解释了为什么 Lua 非常适合嵌入式开发的原因之一吧，而且：

- python-3.12.3-macos11.pkg 为 45.7 MB
- lua-5.4.6.tar.gz 只有 363 KB

面向对象

面向对象（Object Oriented Programming，OOP）是一种非常流行的计算机编程架构 .

Lua 并不直接支持面向对象编程 (OOP)，但通过一些模式和库，可以实现一些面向对象的特性.

面向对象特征

- 封装：指能够把一个实体的信息、功能、响应都装入一个单独的对象中的特性
- 继承：继承的方法允许在不改动原程序的基础上对其进行扩充，这样使得原功能得以保存，而新功能也得以扩展
- 多态：同一操作作用于不同的对象，可以有不同的解释，产生不同的执行结果
- 抽象：是简化复杂的现实问题的途径，它可为具体问题找到最恰当的类定义，并且可以在最恰当的继承级别解释问题

类与实例

有参构造

❗注：下例程序中的 `new` 函数是用来创建一个新的 `Rectangle` 实例的，下面是其中代码的详细解释：

1. `local instance = { width = width, height = height }`：
   这一行创建了一个新的表 `instance`，并将其作为将要返回的新矩形实例. 在这个表中，我们设置了两个属性 `width` 和 `height`，它们的值分别来自于传递给 `new` 函数的参数 `width` 和 `height`.
2. `setmetatable(instance, self)`：
   这一行将 `self` 作为 `instance` 表的元表（metatable）. ‼️**元表是 Lua 中的一个核心概念，它提供了一种机制来控制对表的访问和操作.** 在这个例子中，我们将 `self` 设置为 `instance` 的元表，这意味着当尝试访问 `instance` 表中不存在的属性或方法时，Lua 会查找 `self` 中的对应项.
3. `self.__index = self`：
   这一行设置 `self` 上的 `__index` 字段为 `self` 本身. **当通过 `instance` 表访问一个不存在的属性时，Lua 会查找元表的 `__index` 字段，并尝试在该字段中找到对应的值.** 由于我们设置了 `self.__index = self`，这意味着对 `instance` 表中不存在的属性的访问会被转发到 `self` 上去查找.
4. `return instance`：
   这一行返回 `instance` 表，这个表现在是一个新的 `Rectangle` 实例，包含了通过 `new` 函数传递的 `width` 和 `height` 参数，**并且拥有一个元表，可以提供对实例属性和方法的支持.**

通过这样的设计，我们就可以创建出多个独立的 `Rectangle` 实例，每个实例都有自己的 `width` 和 `height` 属性，同时还可以通过这些实例调用定义在 `Rectangle` 模块中的方法 .

```
local Rectangle = {}

function Rectangle:new(width, height)
    -- 创建一个新实例,并初始化实例属性
    local instance = {
        width = width,
        height = height
    }
    -- 将 self 设置为 instance 的元表
    setmetatable(instance, self)
    -- 设置 self 上的 __index 字段为 self 本身
    -- 这意味着对 instance 表中不存在的属性/方法的访问会被转发到 self 上去查找
    self.__index = self
    return instance
end

function Rectangle:area()
    return self.width * self.height
end
```

创建对象 :

```
local cal = Rectangle:new(1, 2)
```

访问属性 :

```
print(cal.width..' , '..cal.height) -- 1 , 2
```

访问成员函数 :

```
print(cal:area()) -- 2
```

注：上述示例程序常被写成下述形式（代码更加简洁）：

```lua
local Rectangle = {}

function Rectangle:new(obj)
    -- 如果 obj 未定义，创建一个空表
    obj = obj or {}
    -- 将 self 设置为 obj 的元表
    setmetatable(obj, self)
    -- 设置 self 上的 __index 字段为 self 本身
    -- 这意味着对 obj 表中不存在的属性/方法的访问会被转发到 self 上去查找
    self.__index = self
    return obj
end

function Rectangle:area()
    return self.width * self.height
end

local cal = Rectangle:new({ width = 1, height = 2 })

print(cal.width .. ' , ' .. cal.height) -- 1 , 2

print(cal:area()) -- 2
```

无参构造

上述示例程序的无参构造情况：

```lua
local Rectangle = {}

function Rectangle:new()
    ------------------------------
    local instance = {}
    setmetatable(instance, self)
    self.__index = self
```

```
    ------------------------------
    return instance
end

function Rectangle:area()
    return self.width * self.height
end

local rec = Rectangle:new()
rec.height = 2
rec.width = 1

print(rec.width .. ' , ' .. rec.height) -- 1 , 2

print(rec:area()) -- 2
```

继承

在Lua中，由于其**动态类型**的特性，传统的类继承概念并不适用，然而，我们可以使用**元表**和**闭包**来模拟继承的行为.

下面是一个简单的示例，展示了如何在Lua中模拟继承：

```
-- 定义一个基类（父类）
local Parent = {}

-- 创建一个新实例的方法
function Parent:new(o)
    -- 如果 o 未定义，创建一个空表
    o = o or {}
    setmetatable(o, self)
    -- 将 self 作为 o 的 metatable
    self.__index = self
    -- 确保通过 o 访问属性时查找 self
    return o
end
```

```lua
-- 定义一个基类的方法
function Parent:sayHello()
    print("Hello from Parent!")
end

-- 定义一个子类
local Child = {}

-- 子类通过闭包来继承基类的方法
function Child:new(o)
    -- 调用基类的 new 方法来创建基础实例
    o = Parent:new(o)
    -- 将 self 作为 o 的 metatable
    setmetatable(o, self)
    -- 确保通过 o 访问属性时查找 self
    self.__index = self
    return o
end

-- 确保子类可以访问基类的方法
function Child:sayHello()
    -- 调用基类的 sayHello 方法
    Parent.sayHello(self)
end

-- 创建一个子类实例
local my_child = Child:new({ name = "I'm A Child" })

-- 调用子类实例的方法，它实际上是继承自基类的
-- 输出: Hello from Parent!
my_child:sayHello()

-- 尝试访问子类实例中的属性
-- 输出: I'm A Child
print(my_child.name)
```

多态

Lua并没有动态绑定机制，如果我们想要实现更接近于传统面向对象编程中多态的行为，我们可以使用**元表**.

首先，我们创建一个元表（metatable），它将存储我们的 "重载" 方法，**元表是一种特殊类型的表，它关联到一个普通表，并定义了如何处理对该普通表的操作.**

在下面这个例子中，我们定义了一个元表 `M`，它包含一个名为 `__call` 的元方法，这个元方法会在我们尝试调用一个没有方法名的函数时被调用，这种函数也称为 "无名函数" 或 "匿名函数"，我们根据传递给 `__call` 的参数类型来决定调用哪个问候方法.

```lua
-- 定义元表，它将存储我们的重载方法
local M = {}

-- 定义一个用于打印问候语的元方法
function M.__call(self, who)
    -- 根据who的类型，调用不同的问候方法
    if type(who) == "string" then
        print("Hello, " .. who)
    elseif type(who) == "number" then
        print("Hello, you are " .. who .. " years old.")
    else
        error("Invalid argument type. Expecting a string or a number.")
    end
end

-- 创建一个空对象，并设置元表
local person = {}
setmetatable(person, M)

-- 调用问候方法
-- 输出: Hello, John
person("John")
-- 输出: Hello, you are 20 years old.
person(20)
```

抽象

定义一个抽象类的基类 AbstractInterface.lua :

```
local AbstractInterface = {}

function AbstractInterface.abstractMethod(self, arg1, arg2)
    error('Abstract method is not implemented.')
end

function AbstractInterface.abstractMethod2() end

return AbstractInterface
```

创建一个具体的子类 ConcreteClass.lua，它将实现抽象接口中的方法 :

```
require('AbstractInterface')

local ConcreteClass = {}

--[[
❗__index元方法用于当一个对象的方法或属性访问失败（即在对象本身中找不到对应的方法或属性）时，Lua应该去哪里寻找．在这个例子中，我们设置了一个元表，将AbstractInterface模块作为__index的值。这意味着，如果我们在ConcreteClass对象中访问一个未定义的方法或属性，Lua会查找AbstractInterface模块中是否有对应的定义．
]]
setmetatable(ConcreteClass, { __index = AbstractInterface })

function ConcreteClass.abstractMethod(self, arg1, arg2)
    print('ConcreteClass: abstractMethod called with args: ', arg1, arg2)
end

function ConcreteClass.abstractMethod2()
    print('ConcreteClass: abstractMethod2 called without args')
```

```
    end

    return ConcreteClass
```

创建一个测试类 Test.lua，测试抽象类中的方法：

```
    local ConcreteClass = require('ConcreteClass')

    local concreteObject = setmetatable({}, { __index = ConcreteClass })

    concreteObject:abstractMethod(1, 2)
    concreteObject:abstractMethod2()

    -- the output result be shown as following :
    -- ConcreteClass: abstractMethod called with args:  1 2
    -- ConcreteClass: abstractMethod2 called without args
```

数据库访问

`LuaSQL` 是 Lua 语言中一个流行的库，它为 Lua 提供了对 SQL 数据库的访问支持. `LuaSQL` 并不是一个单独的库，而是一个包含多个驱动的集合，这些驱动允许 Lua 程序与不同的 SQL 数据库管理系统（DBMS）进行交互，例如 MySQL、PostgreSQL、SQLite 等.

`LuaSQL` 的设计使得每个 SQL 数据库的驱动都是一个独立的库，这些库遵循 `LuaSQL` 的标准接口，因此开发者可以使用相同的 API 来操作不同的数据库，这种设计使得 `LuaSQL` 非常灵活且易于使用.

`LuaSQL` 的核心功能包括：

1. **数据库连接**：`LuaSQL` 提供了一个接口来创建到指定数据库的连接
2. **执行查询**：开发者可以通过 `LuaSQL` 执行SELECT、INSERT、UPDATE 和 DELETE 等 SQL 语句
3. **结果集处理**：`LuaSQL` 提供了处理查询结果集的机制，通常以表格形式返回结果

4. **事务处理**：支持事务的开始、提交和回滚
5. **绑定参数**：允许开发者使用预处理语句和绑定参数来提高 SQL 语句的执行效率和安全性
6. **错误处理**：`LuaSQL` 提供了统一的错误处理机制，方便开发者处理数据库操作中的错误

每个 `LuaSQL` 的驱动都是独立的，因此开发者需要根据自己使用的数据库系统选择对应的驱动，例如，对于 MySQL，可以使用 `LuaSQL-MySQL` 驱动；对于 PostgreSQL，可以使用 `LuaSQL-PostgreSQL` 驱动.

使用 `LuaSQL`，开发者可以在 Lua 程序中轻松地操作数据库，而无需直接处理复杂的 SQL 数据库接口，这使得 `LuaSQL` 非常适合在需要与数据库交互的大型项目中使用，尤其是在游戏开发、网络应用和服务端脚本等领域.

请注意，`LuaSQL` 并不是唯一一个提供数据库访问的 Lua 库，还有其他库如 `Loky` 和 `luaffie`，它们也提供了类似的功能，选择哪个库取决于项目的具体需求、数据库的兼容性以及开发者个人的偏好.

LuaRocks包管理器

`LuaRocks` 是 Lua 的一个包管理器，它允许开发者轻松地安装、管理、搜索和共享 Lua 库. `LuaRocks` 提供了一个命令行工具，可以用来在 Lua 环境中安装各种库（称为 "rocks"）。这些库可能是第三方库，也可能是由 Lua 社区开发的.

`LuaRocks` 的主要特点包括：

1. **自动化安装**：`LuaRocks` 可以自动处理库的安装，包括依赖项的解析和安装
2. **版本管理**：它支持对不同版本的库进行安装和管理，允许开发者指定特定的版本进行安装
3. **搜索和发现**：`LuaRocks` 提供了一个在线的 rock 仓库（https://luarocks.org/），开发者可以在其中搜索和发现新的库
4. **系统集成**：`LuaRocks` 可以与各种操作系统和 Lua 版本集成，确保库的正确安装和配置
5. **构建系统**：`LuaRocks` 内置了一个构建系统，用于编译和安装 C 语言编写的 Lua 扩展模块

使用 `LuaRocks`，开发者可以通过简单的命令来安装 Lua 库，例如：

```
luarocks install <rockname>
```

这里的 `<rockname>` 是想要安装的库的名字。`luarocks` 会自动下载、编译（如果需要）并安装该库.

除了安装，`LuaRocks` 还允许开发者进行其他操作，如：

- 更新已安装的库：`luarocks update <rockname>`
- 列出已安装的库：`luarocks list`
- 搜索特定的库：`luarocks search <query>`
- 移除已安装的库：`luarocks remove <rockname>`

`LuaRocks` 是 Lua 生态系统中一个非常有用的工具，它简化了依赖管理，并使得在不同的项目中共享和重用 Lua 库变得更加容易.

安装luasql-sqlite

一. 首先我们在 Linux / Mac 中安装一下 LuaRocks：

> 注：Window 下安装 LuaRocks

```
$ wget https://luarocks.org/releases/luarocks-3.11.0.tar.gz
$ tar zxpf luarocks-3.11.0.tar.gz
$ cd luarocks-3.11.0
$ ./configure && make && sudo make install
$ sudo luarocks install luasocket
$ lua
Lua 5.3.5 Copyright (C) 1994-2018 Lua.org, PUC-Rio
> require "socket"
```

二. 利用 LuaRocks 安装 SQLite：

```
$ luarocks install luasql-sqlite3
```

> 注 : 运行此命令的前提是你的操作系统里已经安装好 sqlite .
>
> Download sqlite : https://www.sqlite.org/download.html
>
> How to install sqlite on diff os : https://www.tutorialspoint.com/sqlite/sqlite_installation.htm

三. . .好吧，卡住了！

我滴天啊啊啊，在 windows 中下载个 luasql-sqlite3 好绕好麻烦啊. . .暂停此模块知识点的学习. . .

```
> luarocks install luasql-sqlite3
Installing http://luarocks.org/repositories/rocks/luasql-sqlite3-2.6.1-3.rockspec...

Error: Could not find expected file sqlite3.h for SQLITE -- you may have to install SQLITE in your system and/or set the SQLITE_DIR variable
```