Lua

1. iterator

arr = {"jflaj","a","b"}

for key,value in ipairs(arr)--ipairs为无状态迭代器

do

print(key,value)

end

注：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。

分类：①无状态

②多状态

## pairs 和 ipairs异同

同：都是能遍历集合（表、数组）

异：ipairs 仅仅遍历值，每次从索引1开始，按照索引升序遍历，索引中断停止遍历。即不能返回 nil,只能返回数字 0，如果遇到 nil 则退出。它只能遍历到集合中出现的第一个不是整数的 key，换句话说如果索引1就不是整数key遍历将停止，无法获取到任何元素。

pairs 能遍历集合的所有元素，并且遍历顺序随意。即 pairs 可以遍历集合中所有的 key，并且除了迭代器本身以及遍历表本身还可以返回 nil。

1. table

mytable = {}

mytable = nil

mytable[1] = “fjaljf”

mytable[“wor”] = 12

newtable = mytable

newtable = nil

//此时mytable仍然可以用，不为nil

--10.27

table 中也可以key来存储 value

tt = { [“client”] = func1}

这样就可以 tt.client 来访问相应value值

1. 模块与包

模块的结构就是一个 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。

module = {}

module.constant = "这是一个常量"

function module.func1()

io.write("这是一个公有函数\n");

end

local function func2()

print("this is a private function")

end

function module.func3()

func2()

end

return module

注：将上述代码保存为module.lua 即该文件就是一个模块，可在其他代码中加载

通过require（“模块名”）加载模块

require("module")

-- 加载模块的另外一种方式 require "module"

--模块内的局部函数只能通过该模块的全局函数调用

print(module.constant)

print(module.anotherVar)

module.func3()

--某种程度上来说，模块局就是可以动态调用的table

1. 元表

元表的目的，类似于 C++ 的运算符重载，当我们需要对table进行新的操作（比如两个Table相加），我们给table设置元表，并给元表添加相应方法。

srcT = {}

mT = {}

setmetatable(srcT,mT)

具体：

mT.\_\_add \_\_sub \_\_div \_\_unm(相反数) \_\_mod（取模） \_\_pow(幂)

\_\_mul

当我们操作table时就会去调用相应元表里面的方法

与C++重载运算符相比

同：都是让table或对象能进行新的操作

异：C++需要双方在进行新的操作时，原先都有相关运算符的重载。（即 string + newClass 若newClass类中没有对 + 运算符的重载，则会出错）

Lua进行操作操作数只要有一个相关元表(\_\_add)则操作就可进行下去

\_\_index 用于访问

当用索引运算符[ ]对table某元素进行访问时，若当前table内没有该索引（或key）对应的值，回去寻找该table所定义的\_\_index方法，若还是无法找到。那么才返回nil

1. IO 文件

6.lua面向对象

Table + function 实现类

: 作用域即继承运算符

CellEntity = {}

CellEntity.\_\_index = CellEntity --表明该类只支持派生出一个子类，若在该派生类再派生出一个新的子类则无法继承方法

Lua与C++ 交互

luaL\_newmetatable（l,name） --创建一个新的元表放于栈中

value

key

luaL\_setTable(l,index)

Python

列表 [ ]

元组 ( )

字典 { }

Lua end

C++ { } ；

Python 缩进