**- 元表的理解，元表里面的关键字的使用？举例+解释**

元操作或者行为的集合

setmetatable:寻找\_metatable

getmetatable:

\_\_add、\_\_sub、\_\_mod等

索引访问table[key]，自动寻取\_\_index元方法（往上回溯，用于模拟继承）

索引赋值，\_\_newindex,赋值时在元表中自动创建key值 =function(table.key,value) end

面向对象——利用第一点，将自身作为回溯对象

rawset/rawget：对“原始的”表进行直接的赋值/取值操作（忽略\_\_index和\_\_newindex）。格式：rawset(table, key, value)；rawget(table, key)

继承举例：

local a = {}

a.asd = "ddd"

function a:new(b)

b = b or {}

return setmetatable(b, {

\_\_index = self --将\_\_index原方法设为自己，则自己可以被看成对象

})

end

function a.fuck()

print("hehe,just a joke")

end

--c = a.new(a)

local c = a:new()--两种写法都行，: 调用相当于传入了自己

c.fuck()

c.fuck = "jjjj"

print(c.fuck)

a.fuck()

**- 类继承的理解（参考Babe内部的class/new/delete, ctor 和 dtor 的实现）**

用于定义一个类.class：setmetatable(classType, { \_\_index = super, \_\_newindex = super, \_\_call = new, \_\_class=1 });或者setmetatable(classType, { \_\_call = new });

new:

创建一个类的实例.etmetatable(obj, { \_\_index = classType, \_\_newindex = classType, \_\_object=1});

删除某个实例.destory(obj);

require ("object")

local A = class()

A.m\_value = 1

A.test = function ( self )

print("A->",self)

print("aaaa"..self.m\_value)

end

local B = class(A)

B.m\_value = 2

B.test = function ( self )

print("B->",self)

A.test(self)

print("bbbb"..self.m\_value)

end

local C = class(B, true)

C.m\_value = 3

C.test = function ( self )

print("C->",self)

B.test(self)

print("ccc"..self.m\_value)

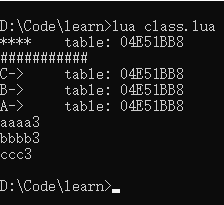
end

local c = new(C)

print("\*\*\*\*",c)

print("###########")

c:test()



**- 对扩展的理解**

将需要拓展的类exClass,基类，同名方法是否同时被调用bBoth传入ClassEx.classExtra()中，如果基类中有同名函数，调用基类的，如果没有，则将拓展类rawset加到基类（table）中。

**- pair和ipair的区别**

pairs会遍历table的所有键值对。table是键值对的数据结构。

而ipairs就是固定地从key值1开始，下次key累加1进行遍历，如果key对应的value不存在，就停止遍历。顺便说下，记忆也很简单，带i的就是根据integer key值从1开始遍历的。

**- 移除表{1,2,3,4,5,6}的偶数? 写代码**

不能直接删的原因：

在table表中使用for迭代时，将符合条件的元素删除时，后面元素前移，然后产生跳跃

而且使用for i=1,#mytable 这样的循环时，由于在for时就已经确定了循环的终点是table的总长度，在整个for运行过程中，终点不会改变。所以当你删除元素时，在循环到i = #mytable 时，会报错nil。因为此时table长度已经缩短了。

------删偶方法一------

local a = {1,2,3,4,5,6}

local b = {}

for k,v in pairs(a) do

if v%2 == 0 then

else

table.insert(b,v)

end

end

for k,v in pairs(b) do

print(k,v)

end

------删偶方法二------

local c = {1,2,3,4,5,6,2}

for i= #c-1,1,-1 do

if c[i]%2 == 0 then

--此处删除键或者值都可以

table.remove(c,i)

--table.remove(c,c[i])

end

end

for k,v in pairs(c) do

print(k,v)

end

**- 对\_G的理解**

lua中所有没有local修饰的变量和函数都是全局的，这些全局变量都放在\_G这个变量中，这个变量是环境全局，也就是说整个lua运行环境都认识这个变量。有\_G.\_G == \_G，在默认情况，Lua在全局环境\_G中添加了标准库比如math、函数比如pairs等

**- 对require的理解？如何清理require进来的文件 ?**

1.加载指定的模块,相当与#include作用类似，加载了该模块，那么就可已使用模块中的全局函数和全局数据（如表等等）

注：实际上require “xxx”后，会将xxx中的全局函数和数据放到表\_G中，所以也就能访问了。

2.如果该模块自身有返回值，且模块加载成功，那么require 的返回值就是该模块的返回值

3.如果模块没有返回值，如果require加载模块成功，就返回ture

4.require 内部将返回值储存在：package.loaded表中。

5.require使用的路径和普通我们看到的路径还有些区别，我们一般见到的路径都是一个目录列表。require的路径是一个模式列表，每一个模式指明一种由虚文件名（require的参数）转成实文件名的方法。更明确地说，每一个模式是一个包含可选的问号的文件名。匹配的时候Lua会首先将问号用虚文件名替换，然后看是否有这样的文件存在。如果不存在继续用同样的方法用第二个模式匹配。例如，路径如下：?;?.lua;c:\windows\?;/usr/local/lua/?/?.lua

调用require "test"时会试着打开这些文件：

test

test.lua

c:\windows\test

/usr/local/lua/test/test.lua

require关注的问题只有分号（模式之间的分隔符）和问号，其他的信息（目录分隔符，文件扩展名）在路径中定义。

6.一个模块被加载后会被缓存到 pacakge.loaded[modulename] 中，清理：package.loaded[modulename] = nil

**- .和:的区别？**

1.定义的时候：Class:test()与 Class.test(self)是等价的，点号(.)要达到冒号(:)的效果要加一个self参数到第一个参数；

2.调用的时候：object:test() 与object.test(object)等价，点号(.)要添加对象自身到第一个参数。

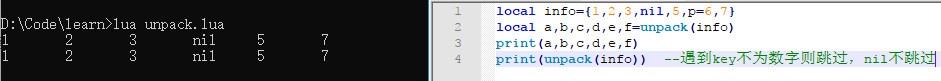
总结：可以把点号(.)作为静态方法来看待，冒号(:)作为成员方法来看待。

<https://blog.csdn.net/daimou123/article/details/50689623>

**- unpack理解，实现unpack?(隐式显式下标都有，优先按隐式)**

unpack它接受一个数组(table)作为参数,并默认从下标1开始返回数组的所有元素。

unpack(t,2) --从下表2开始输出t



local info={1,2,3,nil,"w",p=6,q=8,7}

local a,b,c,d,e,f=unpack(info)

print(a,b,c,d,e,f)

--print(unpack(info,3)) --遇到key不为数字则跳过，nil不跳过

------------unpack具体实现-------------

print(#info) -- key不为number的不计算在里面

for i =1,#info do

print(info[i])

end

**- local a = 10/3, a的值是多少？**

3.33333333333

**- 对lua协程的理解？阻塞？线程？进程**？

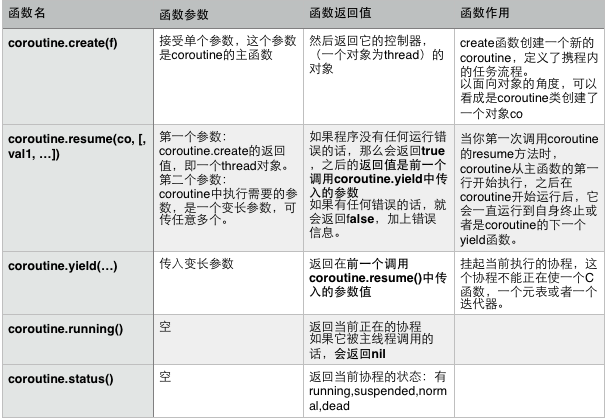
协程和多线程

相同之处：拥有自己独立的桟、局部变量和PC计数器，同时又与其他协程共享全局变量和其他大部分东西

不同之处：一个多线程程序可以同时运行几个线程（并发执行、抢占），而协程却需要彼此协作地运行，并非真正的多线程，即一个多协程程序在同一时间只能运行一个协程，并且正在执行的协程只会在其显式地要求挂起（suspend）时，它的执行才会暂停（无抢占、无并发）。注意：由于Lua中的协程无法在外部将其停止，而且有可能导致程序阻塞。

协程和多线程是两种解决“多任务”编程的技术。多线程使得同一时刻可以有多个线程在执行，不过需要在多个线程间协调资源，因为多个线程的执行进度是“不可控”的。而协程则避免了多线程的问题，同一时刻实质上只有一个“线程”在执行，所以不会存在资源“抢占”的问题。

coroutine 有4个不同的状态：suspended, running, dead, normal。







**- pcall/xpcall的理解？使用情景**？

如果需要在Lua中处理错误，必须使用函数pcall（protected call）来包装需要执行的代码。

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

xpcall接收第二个参数——一个错误处理函数，当错误发生时，Lua会在调用桟展看（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:

debug.debug：提供一个Lua提示符，让用户来价差错误的原因

debug.traceback：根据调用桟来构建一个扩展的错误消息

f=function(...) error(...) end -- 被包装函数抛出错误信息

print(pcall(f, 123)) -- pcall包装f()函数,返回bool，错误信息

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

xpcall(f,

function(e) --错误处理函数，打印调用栈信息

print(debug.traceback())

return e

end,

123)