Lua迭代器

iterator迭代器是一种对象 能够用来遍历标准模板容器中的部分或者全部元素

每个迭代器对象代表容器的确定的地址

支持指针类型的结构 可以遍历集合元素

泛型for迭代器

泛型for迭代器内部保存迭代函数 实际上保存的三个值

1 迭代函数 重复执行一段指令的算法逻辑 用来遍历的代码

2 状态常量 循环或迭代过程中不会改变的值

3 控制变量 当前的索引下标(key)

泛型迭代 key&value的格式

```lua
mytable={10,20,30,40} --定义一个表

for k,v in ipairs(mytable) do
    print(k,v)
end

for x,y in pairs(mytable)do
    print(x,y)
end
--输出结果完全一样 pairs和ipairs都是用来遍历表
```

区别:

```lua
local tab={
[3]="www",
[6]="zzz",
[4]="yyy"
}--结合初始索引不从1开始 顺序打乱

```

```lua
--一次输出结果都没有
--因为Ipairs迭代 不能返回nil 遇到nil会退出 key=1 遇到nil
--只适用于从默认key为1开始 连续的数组结构
for k,v in ipairs(tab)do
    print(k,v)
end

--是能使用pairs来进行遍历  pairs可以遍历表中所有的key 也可以返回nil
for k,v in pairs(tab) do
    print(k,v)
end
```

ipairs一般用于数组迭代 key为number且连续的情况使用 遇到nil退出

pairs一般用于元素迭代 可迭代任意的泛型集合 可以返回nil

迭代器的核心在于迭代函数的实现

所以一般用函数来描述迭代器

除了系统提供的迭代方法以外 开发者可以通过定义迭代函数来实现迭代

Lua迭代器的两种类型

无状态的迭代器

多状态的迭代器

无状态的迭代器

不保留任何状态的迭代器

迭代函数每次都通过两个变量（状态常量和控制变量）的值作为参数进行调用 然后获取下一个数据

ipairs迭代就是无状态的迭代(当前索引和最大索引)

例

迭代实现1-5的两倍数

```lua
--迭代函数
--最大迭代次数 状态常量 iteratorCount
--当前次数     控制变量 current


function double(iteratorCount,current)
    if current<iteratorCount then
        current=current+1
        return current,current*2
    end
end

--i,n=double(5,i)

--迭代的调用 i是从0开始调用 得到结果赋值给n
for i,n in double,5,0 do
    print(i,n)
end
```

实现遍历数组的ipairs迭代

```lua

mytable={1,231,123,1123,111,1}

--迭代函数 参数1个表 1个控制变量
function iter(a,i)
    i=i+1    --更新控制变量
    local v=a[i] --通过索引得到元素值
    if v then
        return i,v  --返回
    end
end

--a参数类型: table
function ipairsSelf(a)
    return iter,a,0
end

for k,v in ipairsSelf(mytable) do
    print(k,v)
end
```

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仅仅是简单的状态常量和控制变量

1 使用闭包

闭包(closure) 一般适用于匿名函数或者嵌套函数

由一个函数和该函数能访问到的非局部变量(不是全局变量)组成

非局部变量(UpValue)：不在局部作用域范围内定义 且 不是全局变量 内部嵌套函数使用外部函数定义的局部变量

```
function fun() --闭包
    local b=200
    return function()
        local a=b
        return a,b --b是非局部变量 Upvalue
    end
end
```

2 将所有的信息都封装到table内

将table作为迭代器的状态常量 所有信息都在table内 所以迭代函数一般不需要定义多个参数

```
mytable={"xxx","yyy"}

function iterator(array)
    local index=0    --控制变量
    local count=#array  --状态常量
    --闭包函数 会使用到外部函数的局部变量
    return function()
        index=index+1 --控制变量的更新
        if index<=count then
            return index,array[index] --返回元素
        end
    end
end

--多状态的迭代调用 更简单直接
```

```lua
for k,v in iterator(mytable) do
    print(k,v)
end
```

Lua 协同程序（thread coroutine）

Lua协同程序与操作系统线程的概念类似

拥有独立的堆栈 独立的局部变量 独立的指令指针

同时又与其他协同程序共享全局变量和公共资源

线程和协同程序的区别

线程和协程的主要区别在于 一个拥有多个线程的程序可以同时运行几个线程

而协同程序同一时间段只会有一个协程在运行

并且这个正在运行的协同程序只有在明确的要求被挂起的时候才会挂起

协同程序有点类似同步的多线程 在等待同一个线程互斥锁的几个线程类似

Lua的协程和Unity的协程一样 都是采用时间分割策略 解决游戏引擎多线程支持不足的问题

基本语法

方法

coroutine.create()

创建coroutine 返回coroutine 参数是一个函数 当和resume配合使用时会唤醒函数调用

coroutine.resume()

1 重启或开启coroutine 和create配合使用时让协程开始运行

2 唤醒yield 使被挂起的协程接着上次运行的地方继续执行

coroutine.yield() 挂起协程

coroutine.status()状态

三种状态:

dead: 函数走完后的状态 不能再次被resume

suspended: 挂起 协程刚创建完成或者 被 yield以后

running:运行中

coroutine.wrap()

创建coroutine 返回一个function 调用这个函数 可以进入coroutine 和create功能类似 返回结果不同

coroutine.running()

返回正运行的coroutine 一个coroutine就是一个线程 所以 当调用running时 返回一个线程号

协程的创建和执行状态

```
--使用create方法创建thread

--函数 thread内部 是通过函数实现的 所以参数会是一个函数
function fun()
    print(100)
end

--把函数当做参数传递给create方法 得到一个thread
co=coroutine.create(fun)
--协程被唤醒 fun函数会调用
coroutine.resume(co)

--Lambda写法用的较多
co=coroutine.create(
    function(x)
        print(x)
    end
)

print(type(co))--thread
print(coroutine.status(co)) --suspended 当thread被创建但是没有执行
coroutine.resume(co,1)   --1 当一个协程被创建必须通过resume来执行
print(coroutine.status(co)) --dead 因为function执行一次结束

--使用Wrap和create功能类似 都可以创建协程 但是wrap返回fucntion
co1=coroutine.wrap(
        function(y)
```

```
            print(y)
        end
        )

print(type(co1)) --function  co1是一个方法
--print(coroutine.status(co1)) --调用报错 因为co1是一个fucntion无法获得状态
co1(2) --function通过wrap创建 所以调用方法就是调用thread
```

协程的yield语法和resume语法

```
co=coroutine.create(
    function()
        for x=1,5 do
            print("第"..x.."次循环")

            if x==3 then
                print(coroutine.status(co)) --状态running
                print(coroutine.running())  --正在运行的线程号 thread: 0024D5A8
            end
            coroutine.yield() --将协程设置为挂起状态 等待下一个resume命令
        end
    end
)

coroutine.resume(co) --第1次循环
coroutine.resume(co) --第2次循环
coroutine.resume(co) --第3次循环
print(coroutine.status(co)) --状态 suspended
print(coroutine.running())  --nil 得到nil说明没有thread在running
coroutine.resume(co) --第4次循环
coroutine.resume(co) --第5次循环
print(coroutine.status(co)) --状态 suspended
coroutine.resume(co) --最后一次调用

print(coroutine.status(co)) --dead
```

协同程序的返回值

```
function fun(a)
    print("fun的参数",a)
    return coroutine.yield(1000*a) --返回1000*a的值
end

--fun(2) 不能直接调用fun 因为内部有yield语句 必须在协程内部调用

co=coroutine.create(
    function(a,b)
        print("第一次输出",a,b) --1 10
        local r=fun(a+1)         --返回a*1000的结果 定义一个r用来接受下一次resume的参数

        print("第二次输出",r)    --第二次resume传递的数据666被赋值给了r
        local r,s=coroutine.yield(a+b,a-b) --a=1 b=10 返回结果 11 -9

        print("第三次输出",r,s) --第三次 resume传递来的数据 www  xxx
        return b,"gameOver"    --10 gameover
    end
    )

--resume接受返回值 重启协程 当重启成功返回true 当重启失败返回false
--resume不能重启状态是dead的thread
print("thread",coroutine.resume(co,1,10)) --true   2000
print("thread",coroutine.resume(co,"666")) --true  11 -9
print("thread",coroutine.resume(co,"www","xxx")) --true 10 gameover
print("thread",coroutine.resume(co)) --false cannot resume dead coroutine
```

Lua协程resume和yield

1 调用resume 将协同程序唤醒 操作成功返回true 操作失败返回false

2 resume可以把参数传递给协同程序

3 数据可以通过yield传递给resume

4 resume的参数可以传递给yield

5 协同程序代码结束时 如果有返回值 也会return给resume

6 如果协程在完成后继续调用reusme 会提示 cannot resume dead coroutine

7 resume和yield的配合 实现了主线程和分线程二者相互的调用和获取

8 resume把外部装塔斯传递到协程内部 而yield则把内部的状态返回到了主线程中

生产者和消费者问题

设定 一共消费者需要100个物品 然而生产者生产力有限也没有库存 必须卖出去一个就生产一个

所以 生产者永远处于挂起状态 每当生产 出一个商品后 yield挂起并返回商品

当消费者购买商品是 需要调用resume把货款传递给生产者 得到商品

```
--生产者的协同程序 初始为挂起状态
local p=coroutine.create(
    function()
        for x=0,99 do
            set(x)
        end
    end
)

--消费者方法 购买100杯奶茶
function consumer()
    for x=0,99 do
        --每次需要购买商品 通知生产者生产一个商品
        local state,item=get()
        print(state,item)
    end
end

--消费者的接受方法 每次购买就调用一次
function get()
    --通知生产者开始生产 并返回生产者生产的商品
    return coroutine.resume(p)
end

--生产者的生产方法 每次返回一个商品 然后设置为挂起状态 等待下一个消费者
function set(x)
```

```
    --冲 包 加
    coroutine.yield("奶茶"..x)
end
consumer()
```

Lua元表 metatable

Lua每一个值都可以设定一个元表

用于定义原始值在特性情况下的操作

元表中的键值

键对应 事件名 事件必须以"__"前缀 例如 __index __add

值对应 元方法 完成事件的操作函数

案例1:

在元表中__index指定函数

```
--普通表

table1={"C#","Lua","C++"}

--元表
metatable1={
    --__index事件名  =  元方法
    __index=function(tab,key)
                print(key)
                return '2019'
            end
}

--设定元表 把metatable1设定为table1的元表
table1=setmetatable(table1,metatable1)

print(table1[2]) --Lua 正常访问

```

```
print(table1['x']) --访问表中没有的键 2019
--当普通表table1没有对应的键值对时 回去调用metatable1的__index指定的方法返回2019字符串
```

案例2:

在元表中指定另一张表

```
--普通表
table1={"C#","Lua","C++"} --3
table2={"a","b",'c','d'}  --4

--元表
metatable1={
    --__index事件名
    --当普通表搜索不到键值对时 会去到__index指定的table2表中进行索引
    __index=table2
}

--设定元表 把metatable1设定为table1的元表
table1=setmetatable(table1,metatable1)

print(table1[1]) --C# 正常访问
print(table1[4]) --d  访问元表中指定的table2
print(table1[6]) --nil 元表中设定的table2也无法找到 返回nil
```