Lua元表

__newindex事件指定元方法增加新的键值 在用新索引赋值时触发

```lua
table1={"x","y","z"}
metatable1={
    __newindex=function(tab,key,value)
        print(tab[2])
        print(key)
        print(value)
        --插入数据
        table.insert(tab,key,value)
    end
}
table1=setmetatable(table1,metatable1)
table1[2]="6"    --修改原有key值为2的数据
table1[5]="w"    --触发元表中newindex事件
print(table1[5]) --w
```

__newindex指定一张表

```lua
table1={"x"} --普通表
table2={}

metatable1={
    __newindex=table2
}
table1=setmetatable(table1,metatable1)
table1[1]="6"  --改变table1本身普通表
table1[5]="www" --改变元表指向的新表
print(table1[1]) --输出 6
print(table1[5]) --输出 nil
print(table2[1]) --输出 nil
print(table2[5]) --输出 www
--当对普通表新增键值对时 只作用于元表中执行的新表 而不作用于普通表
```

Lua中其他的事件名

常用的__add __call __tostring

```lua
mytable={"t1","t2","t3"} --普通表

mymetatable={
    __add=function(tab1,tab2)

          --获得tab1的最大索引值
          local max_index=#tab1

          for k,v in pairs(tab2)do
             max_index=max_index+1
             table.insert(tab1,max_index,v)
          end
         return tab1
         end,--多个事件操作 逗号隔开

    __call=function(tab,num1,num2)
             print(tab[1])
             print(num1)
             print(num2)
             return num1+num2,num1-num2
         end,

    __tostring=function(tab)
             for k,v in pairs(tab)do
                 print(k..":"..v)
             end
             return "把每个元素都打印出来" --tostring事件必须返回一个string
             end
}
mytable=setmetatable(mytable,mymetatable)

newtable={"t4","t5"}
newtable=mytable+newtable --触发__add

print(mytable(5,6)) --触发__call
print(mytable)--触发 __tostring
```

其他事件操作名:

```
__sub 对应运算符 -
__mul 对应运算符 *
__div 对应运算符 /
__mod 对应运算符 %
__unm 对应运算符 - 取负
__pow 对应运算符 ^ 乘幂
__Concat 对应运算符 ..
__eq 对应运算符 ==
__lt 对应运算符 <
__le 对应运算符 <=
__band 对应运算符 &
__bor 对应运算符 |
__shl 对应运算符 <<
__shr 对应运算符 >>
```

Lua标准库

next(table,key=nil)

第一个参数 需要操作的表 第二个参数 表中的某个键

```lua
t={"table",["a"]=5,["c"]=6}

print(next(t)) --不传递第二个参数 得第一个元素

--print(next(t,"d")) --index不存在 没有d的key 报错

print(next(t,1)) --a 5
print(next(t,"a")) --c 6
print(next(t,"c")) ---nil

--判断一个表是不是空表 不能直接判定t2~=nil

t2={}
if next(t2)~=nil then
    print(t2,"不为空")
```

```lua
else
    print(t2,"为空")
end
```

assert(bool,message=nil)

断点调试信息Lua的一种报错机制 方便Debug使用

如果第一个参数为假(nil or false) 调用error 打印message 断点 否则返回所有参数

第二个参数message为空 空则默认为：“assertion failed”

```lua
t={x="1",y="2"}

local a,b=assert(t~=nil,"我错了") --t不为空 直接执行
print(a) --true
print(b) --message 我错了

--assert(s~=nil) --打印默认message:assertion failed! 断点

assert(s~=nil,"我真的错了") --打印 我真的错了 断点

assert(www) --打印默认message:assertion failed! 断点
```

loadstring(str)

加载运行一个字符串str 当做是lua的语法去执行

```lua
loadstring("print('100')")() --100

--一般配合assert使用

assert(loadstring("for index=1,5 do print(index) end"))() --1 2 3 4 5

--接受string中定义的函数 并且调用

z=assert(loadstring("a=function() return 2019 end return a"))()

print(z())--2019
```

dofile(filename)

加载文件 执行文件中的lua代码

和加载模块的require不同 这里是加载文件并且把文件中的代码看成是一段lua语句去使用

```
return 100
```

```
x=dofile("text2.lua") --加载Lua文件
print(x)
--执行逻辑 在dofile(text2.lua)时 相当于定义了一个function
--函数体就是text2.lua中的语法
```

rawequal(v1,v2)

判断是否相等 类型一样且值一样

```
t={"123"}
s="123"
s2="123"
print(rawequal(t,s)) --false
print(rawequal(s,s2)) --true
print(rawequal(t[1],s)) --true
```

rawget(table,index)

获取table中index的值 table参数必须是一个表 找不到返回nil

rawset(table,index,value)

table[index]=value

```
t={"value",x=5}
t2={"sub value"}
```

```lua
rawset(t,1,"new value")
rawset(t,"y",6)
rawset(t,t2,"sub table") --t2也是一个索引

print(t[1]) --new value
print(t.y)  --6
print(t[t2]) --sub table
```

tonumber(e,base)

把e转换为10进制的number base代表e的进制数

```lua
print(tonumber(100)) --100

print(tonumber("100")) --100

print(tonumber("abc")) --nil

print(tonumber("110",2)) --6  把二进制转化为10进制

print(tonumber("f",16)) --15 把十六进制转化为10进制
```

tostring(e)

将任意数据转化成string类型

```lua
print(tostring(123)) --"123"

print(tostring("abc")) --"abc"

print(tostring(type)) --"function:0x函数地址"

table1={"axd"}

print(tostring(table1)) --"table:0x表地址"
```

string 标准库

```
--3种字符串的表示方式
x="sss"                --type string
y='xxx'                --type string
z=[["dsfasdfadasfa"]]  --type string 表示一段字符串

print(string.upper("abc")) --ABC 转大写
print(string.lower("ABC")) --abc 转小写


print(string.find("www.baidu.com","baidu")) --5 9 查找子字符串的起始索引和结束索引

print(string.reverse("ABC")) --字符串反转

print(string.len("1234567890")) --字符串长度 和#同理

print(string.rep("xyz",3)) --重复拼接字符串 xyzxyzxyz

print("ss".."xx") --拼接字符串 ssxx

print(string.byte("a")) --97 返回ascii码

print(string.sub("gamestart",4,7)) --截取字符串 esta

print(string.gsub("abcb","b","x")) --axcx 2 b出现2次

print(string.format("%.4f",3.1415926)) --3.1416 保留4位浮点数

print(string.format("%d %x %o",31,31,31)) --格式化字符串 十进制 十六进制 八进制
```

IO库 输入输出操作

io:open(file,mode)

mode文件打开的方式

```
"r":只读模式打开 默认打开模式
```

```
"w":只写模式打开 允许修改已经存在的文件和创建新文件
"a":追加模式打开 对于已经存在的文件追加新内容 不允许修改原有内容 可以创建新文件
"r+": 读写模式打开已有文件
"w+": 如果文件存在删除文件中数据 如果文件不存在创建新文件 读写模式打开
"a+":以可读的追加模式打开已有文件 若文件不存在则新建文件
```

模式 r r+不会创建新文件 如果文件不存在会返回nil

模式 a a+ w w+ 都会创建新文件

```lua
local file=io.open("test.txt","a+") --文件没有会创建
file:write("abc\n") -- 写入数据
file:close()

local file2=io.open("test.txt","r+") --文件不存在会返回nil
print(file2:read())   --读一行
print(file2:read())
print(file2:read())
print(file2:read())
file2:close()
```

Lua自动内存管理和垃圾回收

Lua采用自动内存管理机制 内存的分配和释放都需要开发者处理

不被任何对象或者全局变量引用的数据 被被标记成回收

Lua的内存泄露问题 主要是由于代码执行所装载的资源没有被彻底释放（卸载）

比如 局部变量忘记加local关键字

Lua解决办法

Collectgarbage

Collectgarbage("collect") 显式的回收函数

Collectgarbage("count") Lua的内存情况

```lua
function text()
```

```lua
    --返回Lua所使用的内存值 单位是k *1024得到字节数
    local c=collectgarbage("count")*1024
    print(c)

    table1={}  --局部变量
    for i=0,2000 do
        table1[i]=i
    end

    local c1=collectgarbage("count")*1024
    print(c1)
end
end

text()

collectgarbage("collect") --执行一次 完整的垃圾回收循环操作
local c2=collectgarbage("count")*1024
print(c2)
```

1 针对大量的代码逻辑块 先调用collectgarbage("count")获取最初的内存

2 代码执行后 在调用collectgarbage("collect")进行一次垃圾收集

3 最后通过collecgarbage("count")获取内存 比较内存差

Lua面向对象

对象是指class的实例 class由方法和属性组成

Lua的基本机构 是table 所以需要实现Class就必须使用table来描述对象的属性

Lua中的function 可以用来表示class的方法 也就是类的行为

像对象一样 表也有成员变量和成员方法 很多功能需要用到metatable来实现

```lua
--表 实际上描述的是类的成员
class1={x=0,y=0,z=0}

--new方法 类似构造函数
function class1:new(x,y)
    local t={} --创建一个table t用来生成新对象
```

```
    setmetatable(t,self) --设置t的元表是class1 self和C#的this 同理
    self.__index=self
    self.x=x or 0
    self.y=y or 0
    self.z=x+y
    return t;
end

--class1的成员函数 printZ
function class1:printZ()
    print("x和y的和是",self.z) --因为用到了self 所以方法必须定义成:调用
end
r=class1:new(4,3) --实例化对象

print(r.x)  --访问成员变量
print(r.y)
r:prinZ()
--.和:都可以访问类的成员 但是如果内部使用到了self 就必须使用:
```

继承和重写(多态)

```
ball={name="",price=0,size=0} --球类 名字 价格 大小(基类)

function ball:new(name,price,size) --基类的new函数
    o={}
    setmetatable(o,self)
    self.__index=self
    self.name=name or ""
    self.price=price or 0
    self.size=sieze or 0
    return o;
end

function ball:play()
    print("玩球")
end


--实例化一个父类对象
```

```lua
b1=ball:new("球",50,0)
b1:play()


--子类 basketball
basketball={}

function basketball:new(name,price,size,color)
    o=ball:new(name,price,size)
    setmetatable(self,o) --将父类设置为子类的元表
    o.__index=o;        --当方法或属性在子类无法查找时 再去调用父类的方法
    self.color=color --子类比父类多定义的属性
    return self
end

--对基类方法的重写
function basketball:play()
    print("打篮球",self.color,self.name)
end

--实例化一个篮球子类对象
b2=basketball:new("篮球",100,10,"红色")
print(b2.price)
b2:play() --调用子类的打篮球方法

--实例化第二个篮球子类对象
b3=basketball:new("篮球",510,0,"红色")
print(b3.price)
b3:play() --调用子类的打篮球方法
```

Lua的单例模式

```lua

--利用一个全局的table实现

manager={} --新建一个类
manager.num=1
manager.__index=manager --定义__index索引 使其作为元表使用

function manager:new()
```

```lua
    self={}
    setmetatable(self,manager) --设置元表为manager
    return self --返回实例对象
end

function manager:getInstance()
    if self.instance==nil then
        self.instance=manager:new()
    end
    return self.instance
end

function manager:func1()
    print("func1")
end

manager1=manager:getInstance()
manager1.num=100
manager2=manager:getInstance()
print(manager2.num) --100 全局元表 数据互通 唯一对象 实现单例
```