Lua 元表(Metatable) Lua 文件 I/O

# Lua 协同程序(coroutine)

## 什么是协同(coroutine)？

Lua 协同程序(coroutine)与线程比较类似：拥有独立的堆栈，独立的局部变量，独立的指令指针，同时又与其它协同程序共享全局变量和其它大部分东西。

协同是非常强大的功能，但是用起来也很复杂。

### 线程和协同程序区别

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。

### 基本语法

| 方法 | 描述 |
| --- | --- |
| coroutine.create() | 创建coroutine，返回coroutine， 参数是一个函数，当和resume配合使用的时候就唤醒函数调用 |
| coroutine.resume() | 重启coroutine，和create配合使用 |
| coroutine.yield() | 挂起coroutine，将coroutine设置为挂起状态，这个和resume配合使用能有很多有用的效果 |
| coroutine.status() | 查看coroutine的状态<br>注：coroutine的状态有三种：dead，suspend，running，具体什么时候有这样的状态请参考下面的程序 |
| coroutine.wrap（ ） | 创建coroutine，返回一个函数，一旦你调用这个函数，就进入coroutine，和create功能重复 |
| coroutine.running() | 返回正在跑的coroutine，一个coroutine就是一个线程，当使用running的时候，就是返回一个corouting的线程号 |

### 以下实例演示了以上各个方法的用法：

```
-- coroutine_test.lua 文件
co = coroutine.create(
    function(i)
        print(i);
    end
)

coroutine.resume(co, 1)   -- 1
print(coroutine.status(co))  -- dead
```

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

co = coroutine.wrap(
    function(i)
        print(i);
    end
)

co(1)

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

co2 = coroutine.create(
    function()
        for i=1,10 do
            print(i)
            if i == 3 then
                print(coroutine.status(co2))  --running
                print(coroutine.running()) --thread:XXXXXX
            end
            coroutine.yield()
        end
    end
)

coroutine.resume(co2) --1
coroutine.resume(co2) --2
coroutine.resume(co2) --3

print(coroutine.status(co2))   -- suspended
print(coroutine.running())

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

以上实例执行输出结果为：

```
1
dead
----------
1
----------
1
2
3
running
thread: 0x7fb801c05868    false
suspended
```

```
thread: 0x7fb801c04c88    true
----------
```

coroutine.running就可以看出来,coroutine在底层实现就是一个线程。

当create一个coroutine的时候就是在新线程中注册了一个事件。

当使用resume触发事件的时候，create的coroutine函数就被执行了，当遇到yield的时候就代表挂起当前线程，等候再次resume触发事件。

接下来我们分析一个更详细的实例：

```
function foo (a)
    print("foo 函数输出", a)
    return coroutine.yield(2 * a) -- 返回  2*a 的值
end

co = coroutine.create(function (a , b)
    print("第一次协同程序执行输出", a, b) -- co-body 1 10
    local r = foo(a + 1)

    print("第二次协同程序执行输出", r)
    local r, s = coroutine.yield(a + b, a - b)  -- a，b的值为第一次调用协同程序时传入

    print("第三次协同程序执行输出", r, s)
    return b, "结束协同程序"                   -- b的值为第二次调用协同程序时传入
end)

print("main", coroutine.resume(co, 1, 10)) -- true, 4
print("--分割线----")
print("main", coroutine.resume(co, "r")) -- true 11 -9
print("---分割线---")
print("main", coroutine.resume(co, "x", "y")) -- true 10 end
print("---分割线---")
print("main", coroutine.resume(co, "x", "y")) -- cannot resume dead coroutine
print("---分割线---")
```

以上实例执行输出结果为：

```
第一次协同程序执行输出    1    10
foo 函数输出    2
main    true    4
--分割线----
第二次协同程序执行输出    r
main    true    11    -9
---分割线---
第三次协同程序执行输出    x    y
main    true    10    结束协同程序
---分割线---
```

```
main    false    cannot resume dead coroutine
---分割线---
```

以上实例接下如下：

- 调用resume，将协同程序唤醒,resume操作成功返回true，否则返回false；
- 协同程序运行；
- 运行到yield语句；
- yield挂起协同程序，第一次resume返回；（注意：此处yield返回，参数是resume的参数）
- 第二次resume，再次唤醒协同程序；（注意：此处resume的参数中，除了第一个参数，剩下的参数将作为yield的参数）
- yield返回；
- 协同程序继续运行；
- 如果使用的协同程序继续运行完成后继续调用 resume方法则输出：cannot resume dead coroutine

resume和yield的配合强大之处在于，resume处于主程中，它将外部状态（数据）传入到协同程序内部；而yield则将内部的状态（数据）返回到主程中。

---

# 生产者-消费者问题

现在我就使用Lua的协同程序来完成生产者-消费者这一经典问题。

```
local newProductor

function productor()
     local i = 0
     while true do
          i = i + 1
          send(i)     -- 将生产的物品发送给消费者
     end
end

function consumer()
     while true do
          local i = receive()     -- 从生产者那里得到物品
          print(i)
     end
end

function receive()
     local status, value = coroutine.resume(newProductor)
     return value
end

function send(x)
```

```
    coroutine.yield(x)     -- x表示需要发送的值，值返回以后，就挂起该协同程序
end

-- 启动程序
newProductor = coroutine.create(productor)
consumer()
```

以上实例执行输出结果为：

```
1
2
3
4
5
6
7
8
9
10
11
12
13
.....
```

Lua 元表(Metatable) Lua 文件 I/O