

Lua 基础 coroutine —— Lua 的多线程编程

1. Coroutine 基础

Lua 将 coroutine 相关的所有函数封装在表 `coroutine` 中。`create` 函数，创建一个 coroutine，以该 coroutine 将要运行的函数作为参数，返回类型为 `thread`。

```
> co = coroutine.create(function () print("hi") end)
> print(co, type(co))
thread: 0x2429f70      thread
```

coroutine 有 4 个不同的状态：*suspended*, *running*, *dead*, *normal*。当新 *create* 一个 coroutine 的时候，它的状态为 *suspended*，意味着在 *create* 完成后，该 *coroutine* 并没有立即运行。我们可以用函数 `status` 来看该 coroutine 的状态：

```
> co = coroutine.create(function () print("hi") end)
> print(co, type(co))
thread: 0x2429f70      thread
> print(coroutine.status(co))
suspended
```

函数 `coroutine.resume`（恢复）运行该 coroutine，将其状态从 *suspended* 变为 *running*：

```
> co = coroutine.create(function () print("hi") end)
> print(co, type(co))
thread: 0x2429f70      thread
> print(coroutine.status(co))
suspended
> coroutine.resume(co)
hi
```

在该示例中，该 coroutine 运行，简单地输出一个“hi”就结束了，该 coroutine 变为 dead 状态：

```
> co = coroutine.create(function () print("hi") end)
> print(co, type(co))
thread: 0x2429f70      thread
> print(coroutine.status(co))
suspended
> coroutine.resume(co)
hi
> print(coroutine.status(co))
dead
```

到目前为止，coroutine 看起来好像也就这么回事，类似函数调用，但是更复杂的函数调用。但是，coroutine 的真正强大之处在于它的 *yield* 函数，它可以将正在运行的 coroutine 挂起，并可以在适当的时候再重新被唤醒，然后继续运行。下面，我们先看一个简单的示例：

```
> co = coroutine.create(function () for i=1,10 do print("co", i) coroutine.yield() end end)
> coroutine.resume(co)
co      1
> coroutine.resume(co)
co      2
> coroutine.resume(co)
co      3
> coroutine.resume(co)
co      4
> coroutine.resume(co)
co      5
> coroutine.resume(co)
co      6
> coroutine.resume(co)
co      7
> coroutine.resume(co)
co      8
> coroutine.resume(co)
co      9
> print(coroutine.status(co))
suspended
> coroutine.resume(co)
co     10
> print(coroutine.status(co))
suspended
> coroutine.resume(co)
> print(coroutine.status(co))
dead
> print(coroutine.resume(co))
false    cannot resume dead coroutine
```

我们一步一步来讲，该 coroutine 每打印一行，都会被挂起，看起来是不是在运行 *yield* 函数的时候被挂起了呢？当我们用 *resume* 唤醒该 coroutine 时，该 coroutine 继续运行，打印出下一行。直到最后没有东西打印出来的时候，该 coroutine 退出循环，变为 dead 状态（注意最后那里的状态变化）。如果对一个 dead 状态的 coroutine 进行 *resume* 操作，那么 *resume* 会返回 false+err_msg，如上面最后两行所示。

注意，`resume` 是运行在 `protected mode` 下。当 `coroutine` 内部发生错误时，Lua 会将错误信息返回给 `resume` 调用。

当一个 `coroutine A` 在 `resume` 另一个 `coroutine B` 时，`A` 的状态没有变为 `suspended`，我们不能去 `resume` 它；但是它也不是 `running` 状态，因为当前正在 `running` 的是 `B`。这时 `A` 的状态其实就是 `normal` 状态了。

Lua 的一个很有用的功能，`resume-yield` 对，可以用来交换数据。下面是 4 个小示例：

1) `main` 函数中没有 `yield`，调用 `resume` 时，多余的参数，都被传递给 `main` 函数作为参数，下面的示例，1 2 3 分别就是 `a b c` 的值了：

```
> co = coroutine.create(function (a,b,c)
>> print("co", a,b,c)
>> end)
> coroutine.resume(co, 1, 2, 3)
co      1      2      3
```

2) `main` 函数中有 `yield`，所有被传递给 `yield` 的参数，都被返回。因此 `resume` 的返回值，除了标志正确运行的 `true` 外，还有传递给 `yield` 的参数值：

```
> co = coroutine.create(function (a,b)
>> coroutine.yield(a+b, a-b) end)
> print(coroutine.resume(co, 20, 10))
true    30    10
```

3) yield 也会把多余的参数返回给对应的 resume，如下：

```
> co = coroutine.create(function ()
>> print("co", coroutine.yield()) end)
> coroutine.resume(co)
> coroutine.resume(co, 4, 5)
co      4      5
```

为啥第一个 resume 没有任何输出呢？我的答案是，yield 没有返回，print 就根本还没运行。

4) 当一个 coroutine 结束的时候，main 函数的所有返回值都被返回给 resume：

```
> co = coroutine.create(function () return 6,7 end)
> print(coroutine.resume(co))
true    6    7
```

我们在同一个 coroutine 中，很少会将上面介绍的这些功能全都用上，但是所有这些功能都是很 useful 的。

目前为止，我们已经了解了 Lua 中 coroutine 的一些知识了。下面我们需要明确几个概念。Lua 提供的是 asymmetric coroutine，意思是说，它需要一个函数 (yield) 来挂起一个 coroutine，但需要另一个函数 (resume) 来唤醒这个被挂起的 coroutine。对应的，一些语言提供了 symmetric coroutine，用来切换当前 coroutine 的函数只有一个。

有人想把 Lua 的 coroutine 称为 semi-coroutine，但是这个词已经被用作别的意思了，用来表示一个被限制了一些功能来实现出来的 coroutine，这样的 coroutine，只有在一个 coroutine 的调用堆栈中，没有剩余任何挂起的调用时，才会被挂起，换句话说，就是只有 main 可以挂起。Python 中的 generator 好像就是这样一个类似的 semi-coroutine。

跟 asymmetric coroutine 和 symmetric coroutine 的区别不同，coroutine 和 generator (Python 中的) 的不同在于，generator 并没有 coroutine 的功能强大，一些用 coroutine 可实现的有趣的功能，用 generator 是实现不了的。Lua 提供了一个功能完整的 coroutine，如果有人喜欢 symmetric coroutine，可以自己简单的进行一下封装。

2. pipes 和 filters

coroutine 的一个典型的例子就是 producer-consumer 问题。我们来假设有这样两个函数，一个不停的 produce 一些值出来（例如从一个 file 中不停地读），另一个不断地 consume 这些值（例如，写入到另一个 file 中）。这两个函数的样子应该如下：

[\[plain\]](#) [view](#) [plain copy](#)

1. function producer ()

```

2.     while true do

3.         local x = io.read() -- produce new value

4.         send(x) -- send to consumer

5.     end

6. end

7. function consumer ()

8.     while true do

9.         local x = receive() -- receive from producer

10.        io.write(x, "\n") -- consume new value

11.    end

12. end

```

这两个函数都不停的在执行，那么问题来了，怎么来匹配 send 和 receive 呢？究竟谁先谁后呢？

coroutine 提供了解决上面问题的一个比较理想的工具 resume-yield。我们还是不说废话，先看看代码再说说我自己的理解：

[\[plain\]](#) [view](#) [plain copy](#)

```

1. function receive (prod)

2.     local status, value = coroutine.resume(prod)

```

```
3.     return value
4. end
5.
6.     function send (x)
7.         coroutine.yield(x)
8.     end
9.
10.    function producer()
11.        return coroutine.create(function ()
12.            while true do
13.                local x = io.read() -- produce new value
14.                send(x)
15.            end
16.        end)
17.    end
18.
19.    function consumer (prod)
20.        while true do
21.            local x = receive(prod) -- receive from producer
```



```
22.      io.write(x, "\n") -- consume new value
```

```
23.      end
```

```
24.  end
```

```
25.
```

```
26.  p = producer()
```

```
27.  consumer(p)
```

程序先调用 consumer，然后 recv 函数去 resume 唤醒 producer，produce 一个值，send 给 consumer，然后继续等待下一次 resume 唤醒。看下下面的这个示例应该就很明白了：

```
> function send(x) coroutine.yield(x) end
> co = coroutine.create(function () while true do local x = io.read() send(x) end end)
> print(coroutine.status(co))
suspended
> print(coroutine.resume(co))
1
true    1
> print(coroutine.status(co))
suspended
> print(coroutine.resume(co))
2
true    2
> print(coroutine.status(co))
suspended
> print(coroutine.resume(co))
3
true    3
> print(coroutine.status(co))
suspended
```

我们可以继续扩展一下上面的例子，增加一个 filter，在 producer 和 consumer 之间做一些数据转换啥的。那么 filter 里都做些什么呢？我们先看一下没加 filter 之前的逻辑，基本就是 producer 去 send，send to consumer，consumer 去 recv，recv from producer，可以这么理解吧。加了 filter 之后呢，因为 filter 需要对 data 做一些转换操作，因此这时的逻辑为，producer 去 send，send to filter，filter 去 recv，recv from producer，filter 去 send，send to consumer，consumer 去 recv，recv from filter。红色的部分是跟原来不同的。此时的代码如下：

[\[plain\]](#) [view](#) [plain copy](#)

```
1.  function send(x)
2.      coroutine.yield(x)
3.  end
4.
5.  function producer()
6.      return coroutine.create(function ()
7.          while true do
8.              local x = io.read()
9.              send(x)
10.          end
11.      end)
```

12. end

13.

14. function consumer(prod)

15. while true do

16. local x = receive(prod)

17. if x then

18. io.write(x, '\n')

19. else

20. break

21. end

22. end

23. end

24.

25. function filter(pro

d)

26. return coroutine.create(function ()

27. for line = 1, math.huge do

28. local x = receive(prod)

29. x = string.format('%5d %s', line, x)

```
30.         send(x)
```

```
31.         end
```

```
32.     end)
```

```
33. end
```

```
34.
```

```
35. p = producer()
```

```
36. f = filter(p)
```

```
37. consumer(f)
```

看完上面的例子,你是否想起了 unix 中的 pipe? coroutine 怎么说也是 multithreading 的一种。使用 pipe, 每个 task 得以在各自的 process 里执行, 而是用 coroutine, 每个 task 在各自的 coroutine 中执行。pipe 在 writer (producer) 和 reader (consumer) 之间提供了一个 buffer, 因此相对的运行速度还是相当可以的。这个是 pipe 很重要的一个特性, 因为 process 间通信, 代价还是有点大的。使用 coroutine, 不同 task 之间的切换成本更小, 基本上也就是一个函数调用, 因此, writer 和 reader 几乎可以说是齐头并进了啊。

3. 用 coroutine 实现迭代器

我们可以把迭代器 循环看成是一个特殊的 producer-consumer 例子：
迭代器 produce，循环体 consume。下面我们就看一下 coroutine 为我们提供的强大的功能，用 coroutine 来实现迭代器。

我们来遍历一个数组的全排列。先看一下普通的 loop 实现，代码如下：

[\[plain\]](#) [view](#) [plain copy](#)

```
1.  function printResult(a)
2.      for i = 1, #a do
3.          io.write(a[i], ' ')
4.      end
5.      io.write("\n")
6.  end
7.
8.  function permgen
    (a, n)
9.      n = n or #a
10.     if n <= 1 then
11.         printResult(a)
12.     else
```

```
13.     for i = 1, n do
14.         a[n], a[i] = a[i], a[n]
15.         permgen(a, n-1)
16.         a[n], a[i] = a[i], a[n]
17.     end
18. end
19. end
20.
21. permgen({1,2,3})
```

运行结果如下：

```
[carl@localhost lua]$ lua permgen_test01.lua
2 3 1
3 2 1
3 1 2
1 3 2
2 1 3
1 2 3
```

再看一下迭代器实现，注意比较下代码的改变的部分：

[\[plain\]](#) [view](#) [plain copy](#)

```
1.  function printResult(a)
2.      for i = 1, #a do
3.          io.write(a[i], ' ')
4.      end
```

5. io.write("\n")

6. end

7.

8. function permgen(a, n)

9. n = n or #a

10. if n <= 1 then

11. coroutine.yield(a)

12. else

13. for i = 1, n do

14. a[n], a[i] = a[i], a[n]

15. permgen(a, n-1)

16. a[n], a[i] = a[i], a[n]

17. end

18. end

19. end

20.

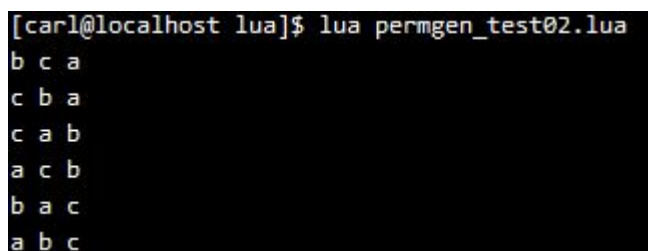
21. function permutations(a)

22. local co = coroutine.create(function () permgen(a) en

 d)

```
23.     return function ()  
  
24.         local code, res = coroutine.resume(co)  
  
25.         return res  
  
26.     end  
  
27. end  
  
28.  
  
29. for p in permutations({"a", "b", "c"}) do  
  
30.     printResult(p)  
  
31. end
```

运行结果如下：



```
[carl@localhost lua]$ lua permgen_test02.lua  
b c a  
c b a  
c a b  
a c b  
b a c  
a b c
```

permutations 函数使用了一个 Lua 中的常规模式，将在函数中去 resume 一个对应的 coroutine 进行封装。Lua 对这种模式提供了一个函数 coroutine.wrap 。跟 create 一样，wrap 创建一个新的 coroutine ，但是并不返回给 coroutine，而是返回一个函数，调用这个函数，对应的 coroutine 就被唤醒去运行。跟原来的 resume 不同的是，该函数不会返回 errcode 作为第一个返回值，一旦有 error 发生，就退出了（类似 C 语言的 assert）。使用 wrap， permutations 可以如下实现：


```
1.  function permutations (a)
2.      return coroutine.wrap(function () permgen(a) end)
3.  end
```

`wrap` 比 `create` 简单，它实在的返回了我们最需要的东西：一个可以唤醒对应 `coroutine` 的函数。但是不够灵活。没有办法去检查 `wrap` 创建的 `coroutine` 的 `status`，也不能检查 `runtime-error`（没有返回 `errcode`，而是直接 `assert`）。

4. 非抢占式多线程

从我们前面所写的可以看到，`coroutine` 运行一系列的协作的多线程。每个 `coroutine` 相当于一个 `thread`。一个 `yield-resume` 对可以在不同的 `thread` 之间切换控制权。但是，跟常规的 `multithr` 不同，`coroutine` 是非抢占式的。一个 `coroutine` 在运行的时候，不可能被其他的 `coroutine` 从外部将其挂起，只有由其本身显式地调用 `yield` 才会挂起，并交出控制权。对一些程序来说，这没有任何问题，相反，因为非抢占式的缘故，程序变得更加简单。我们不需要担心同步问题的 `bug`，因为在 `threads` 之间的同步都是显式的。我们只需要保证在对的时刻调用 `yield` 就可以了。

但是，使用非抢占式 multithreading，不管哪个 thread 调用了一个阻塞的操作，那么整个程序都会被阻塞，这是不能容忍的。由于这个原因，很多程序员并不认为 coroutine 可以替代传统的 multithreading。但是，下面我们可以看到一个有趣的解决办法。

一个很典型的 multithreading 场景：通过 http 下载多个 remote files。我们先来看下如何下载一个文件，这需要使用 LuaSocket 库，如果你的开发环境没有这个库的话，可以看下博主的另一篇文章 [Lua 基础安装 LuaSocket](#)，了解下如何在 Linux 上安装 LuaSocket. 下载一个 file 的 lua 代码如下：

[\[plain\]](#) [view](#) [plain copy](#)

```
1.  require("socket")
2.
3.  host = "www.w3.org"
4.  file = "/standards/xml/schema"
5.
6.  c = assert(socket.connect(host, 80))
7.  c:send("GET " .. file .. " HTTP/1.0\r\n\r\n") -- 注意 GET 后和 HTTP 前面的空格
8.
9.  while true do
```

```
10.    local s, status, partial = c:receive(2^10)

11.    io.write(s or partial)

12.    if status == "closed" then

13.        break

14.    end

15. end

16.

17. c:close()
```

运行结果有点长，不方便截图，就不贴了。

现在我们就知道怎么下载一个文件了。现在回到前面说的下载多个 remote files 的问题。当我们接收一个 remote file 的时候，程序花费了大多数时间去等待数据的到来，也就是在 receive 函数的调用是阻塞。因此，如果能够同时下载所有的 files，那么程序的运行速度会快很多。下面我们看一下如何用 coroutine 来模拟这个实现。我们为每一个下载任务创建一个 thread，在一个 thread 没有数据可用的时候，就调用 yield 将程序控制权交给一个简单的 dispatcher，由 dispatcher 来唤醒另一个 thread。下面我们先把之前的代码写成一个函数，但是有少许改动，不再将 file 的内容输出到 stdout 了，而只是间的输出 filesize。

```
1.  function download(host, file)
2.      local c = assert(socket.connect(host, 80))
3.      local count = 0 -- counts number of bytes read
4.      c:send("GET " .. file .. " HTTP/1.0\r\n\r\n")
5.      while true do
6.          local s, status, partial = receive(c)
7.          count = count + #(s or partial)
8.          if status == "closed" then
9.              break
10.         end
11.     end
12.     c:close()
13.     print(file, count)
14. end
```

上面代码中有个函数 `receive`，相当于下载单个文件中的实现如下：

[\[plain\]](#) [view](#) [plain copy](#)

```
1.  function receive (connection)
2.      return connection:receive(2^10)
3.  end
```

但是,如果要同时下载多文件的话,这个函数必须非阻塞地接收数据。在没有数据接收的时候,就调用 `yield` 挂起,交出控制权。实现应该如下:

[\[plain\]](#) [view](#) [plain copy](#)

```
1.  function receive(connection)
2.
3.      connection:settimeout(0) -- do not block
4.
5.      local s, status, partial = connection:receive(2^10)
6.
7.      if status == "timeout" then
8.
9.          coroutine.yield(connection)
10.
11.      end
12.
13.      return s or partial, status
14.
15.  end
```

`settimeout(0)`将这个连接设为非阻塞模式。当 `status` 变为“timeout”时,意味着该操作还没完成就返回了,这种情况下,该 `thread` 就 `yield`。传递给 `yield` 的 `non-false` 参数,告诉 `dispatcher` 该线程仍然在运行。注意,即使 `timeout` 了,该连接还是会返回它已经收到的东西,存在 `partial` 变量中。

下面的代码展示了一个简单的 `dispatcher`。表 `threads` 保存了一系列的运行中的 `thread`。函数 `get` 确保每个下载任务都单独一个 `thread`。`dis`

patcher 本身是一个循环，不断的遍历所有的 thread，一个一个的去 resume。如果一个下载任务已经完成，一定要将该 thread 从表 thread 中删除。当没有 thread 在运行的时候，循环就停止了。

最后，程序创建它需要的 threads，并调用 dispatcher。例如，从 w3c 网站下载四个文档，程序如下所示：

[\[plain\]](#) [view](#) [plain copy](#)

```
1.  require "socket"
2.
3.  function receive(connection)
4.      connection:settimeout(0) -- do not block
5.      local s, status, partial = connection:receive(2^10)
6.      if status == "timeout" then
7.          coroutine.yield(connection)
8.      end
9.      return s or partial, status
10. end
11.
12. function download(host, file)
```

```
13.    local c = assert(socket.connect(host, 80))

14.    local count = 0 -- counts number of bytes read

15.    c:send("GET " .. file .. " HTTP/1.0\r\n\r\n")

16.    while true do

17.        local s, status, partial = receive(c)

18.        count = count + #(s or partial)

19.        if status == "closed" then

20.            break

21.        end

22.    end

23.    c:close()

24.    print(file, count)

25. end

26.

27. threads = {} -- list of all live threads

28.

29. function get(host, file)

30.     -- create coroutine

31.     local co = coroutine.create(function ()
```

```
32.     download(host, file)

33. end)

34.     -- insert it in the list

35.     table.insert(threads, co)

36. end

37.

38. function dispatch()

39.     local i = 1

40.     while true do

41.         if threads[i] == nil then -- no more threads?

42.             if threads[1] == nil then -- list is empty?

43.                 break

44.             end

45.             i = 1 -- restart the loop

46.         end

47.         local status, res = coroutine.resume(threads[i])

48.         if not res then -- thread finished its task?

49.             table.remove(threads, i)

50.         else
```



```

51.         i = i + 1

52.     end

53. end

54. end

55.

56. host = "www.w3.org"

57. get(host, "/TR/html401/html40.txt")

58. get(host, "/TR/2002/REC-xhtml1-20020801/xhtml1.pdf")

59. get(host, "/TR/REC-html32.html")

60. get(host, "/TR/2000/REC-DOM-Level-2-Core-20001113/DOM2-Core.txt")

61. dispatch() -- main loop

```

我的程序运行了 10s 左右，4 个文件已经下载完成，运行结果如下：

```

[carl@localhost lua]$ lua download_files.lua
/TR/2002/REC-xhtml1-20020801/xhtml1.pdf 115785
/TR/REC-html32.html 125638
/TR/2000/REC-DOM-Level-2-Core-20001113/DOM2-Core.txt 229691
/TR/html401/html40.txt 792682

```

我又重新用阻塞式的顺序下载重试了一下，需要时间 12s 多一点，可能文件比较小，也不够多，对比不是很明显，阻塞的多文件下载代码如下，其实就是上面几段代码放在一块了

[\[plain\]](#) [view](#) [plain copy](#)

```

1. function receive (connection)

```

```
2.     return connection.receive(2^10)

3. end

4.

5. function download(host, file)

6.     local c = assert(socket.connect(host, 80))

7.     local count = 0 -- counts number of bytes read

8.     c:send("GET " .. file .. " HTTP/1.0\r\n\r\n")

9.     while true do

10.        local s, status, partial = receive(c)

11.        count = count + #(s or partial)

12.        if status == "closed" then

13.            break

14.        end

15.    end

16.    c:close()

17.    print(file, count)

18. end

19.

20. require "socket"
```

21.

22. `host = "www.w3.org"`

23.

24. `download(host, "/TR/html401/html40.txt")`

25. `download(host, "/TR/2002/REC-xhtml1-20020801/xhtml1.pdf")`

26. `download(host, "/TR/REC-html32.html")`

27. `download(host, "/TR/2000/REC-DOM-Level-2-Core-20001113/DOM2-Core.txt")`

运行结果如下，跟上面的非阻塞式有点不同，下载完成的顺序，就是代码中写的顺序：

```
[carl@localhost lua]$ lua download_files_block.lua
/TR/html401/html40.txt 792682
/TR/2002/REC-xhtml1-20020801/xhtml1.pdf 115785
/TR/REC-html32.html 125638
/TR/2000/REC-DOM-Level-2-Core-20001113/DOM2-Core.txt 229691
```

既然速度没有明显的更快，那么有没有优化空间呢，答案是，有。当没有 thread 有数据接收时，dispatcher 遍历了每一个 thread 去看它有没有数据过来，结果这个过程比阻塞式的版本多耗费了 30 倍的 cpu。

为了避免这个情况，我们使用 LuaSocket 提供的 select 函数。它运行程序在等待一组 sockets 状态改变时阻塞。代码改动比较少，在循环

中，收集 timeout 的连接到表 connections 中，当所有的连接都 timeout 了，dispatcher 调用 select 来等待这些连接改变状态。该版本的程序，在博主开发环境测试，只需 7s 不到，就下载完成 4 个文件，除此之外，对 cpu 的消耗也小了很多，只比阻塞版本多一点点而已。新的 dispatcher 代码如下：

[\[plain\]](#) [view](#) [plain copy](#)

```
1.  function dispatch()
2.      local i = 1
3.      local connections = {}
4.      while true do
5.          if threads[i] == nil then -- no more threads?
6.              if threads[1] == nil then -- list is empty?
7.                  break
8.              end
9.              i = 1 -- restart the loop
10.             connections = {}
11.         end
12.         local status, res = coroutine.resume(threads[i])
13.         if not res then -- thread finished its task?
14.             table.remove(threads, i)
```

```
15.     else

16.         i = i + 1

17.         connections[#connections + 1] = res

18.         if #connections == #threads then -- all threads blocked?

19.             socket.select(connections)

20.         end

21.     end

22. end

23. end
```

运行结果如下：

```
[carl@localhost lua]$ lua download_files_optimal.lua
/TR/2002/REC-xhtml1-20020801/xhtml1.pdf 115785
/TR/REC-html32.html 125638
/TR/2000/REC-DOM-Level-2-Core-20001113/DOM2-Core.txt 229691
/TR/html401/html40.txt 792682
```

这边文章又是断断续续写了几天，文章的每个例子都是