1. for循环的形式：
2. for i = 1, 100 do

函数体

end

这种循环形式执行100次，i每次加1

1. for i= 1,100,2 do

函数体

end

这种循环形式执行100次，i每次加2

3) for i=1,#t do

函数体

End

#t为表的长度，执行表的长度次循环

4） for k,v in pairs(t) do

函数体

End

遍历获取表中key，value值，执行表的长度次循环

1. 加载lua文件：  
   1）require函数，载入同样的lua文件时，只有第一次的时候会去执行，后面的相同的都不执行了。  
   2）如果你要让每一次文件都会执行的话，你可以使用dofile(“hello”)函数  
   3）如果你要玩载入后不执行，等你需要的时候执行时，你可以使用 loadfile()函数，如下所示：

local hello = loadfile("hello")

... ...

... ...

hello()

1. 类型转换
   1. 当字符串使用算术运算符（+、-、\*、\）进行操作时，字符串会自动的转换为数字
   2. 当数字使用连接符(“..”)进行操作时，数字会自动的转换为字符串,当在一个数字后面写..时，必须加上空格以防止被解释错。

Print(10 .. 20)

1. 逻辑运算符（or、and、not）
   1. a or b,如果a为true，返回a,否则返回b
   2. a and b,如果a为false，返回a,否则返回b
   3. and的优先级比or高
   4. not的结果只返回true和false
2. break和return
   1. Lua语法要求break和return只能出现在block的结尾一句（也就是说：作为chunk的最后一句，或者在end之前，或者else前，或者until前）

**2016.2.17**

1.函数多重返回值的作为参数的实例：

f() -- 调整到 0 个结果

g(f(), x) -- f() 被调整到一个结果

g(x, f()) -- g 被传入 x 加上所有 f() 的返回值

a,b,c = f(), x -- f() 被调整到一个结果 （ c 在这里被赋为 nil ）

a,b = ... -- a 被赋值为可变参数中的第一个，

-- b 被赋值为第二个 （如果可变参数中并没有对应的值，这里 a 和 b 都有可能被赋为 nil）

a,b,c = x, f() -- f() 被调整为两个结果

a,b,c = f() -- f() 被调整为三个结果

return f() -- 返回 f() 返回的所有结果

return ... -- 返回所有从可变参数中接收来的值

return x,y,f() -- 返回 x, y, 以及所有 f() 的返回值

{f()} -- 用 f() 的所有返回值创建一个列表

{...} -- 用可变参数中的所有值创建一个列表

{f(), nil} -- f() 被调整为一个结果

被括号括起来的表达式永远被当作一个值。所以， (f(x,y,z)) 即使 f 返回多个值，这个表达式永远是一个单一值。 （(f(x,y,z)) 的值是 f 返回的第一个值。如果 f 不返回值的话，那么它的值就是 **nil** 。）

2.for循环通过迭代器（iterators）工作

**for** 语句好似这样

for *var\_1*, ···, *var\_n* in *explist* do *block* end

它等价于这样一段代码：

do

local *f*, *s*, *var* = *explist*

while true do

local *var\_1*, ···, *var\_n* = *f*(*s*, *var*)

*var* = *var\_1*

if *var* == nil then break end

*block*

end

end

注意以下几点：

* *explist* 只会被计算一次。 它返回三个值， 一个迭代器函数，一个状态，一个迭代器的初始值。
* *f*、 *s*、 以及 *var* 都是不可见的变量。 这里给它们起的名字都只是为了解说方便。
* 你可以使用 **break** 来跳出 **for** 循环。
* 循环变量 *var\_i* 对于循环来说是一个局部变量； 你不可以在 **for** 循环结束后继续使用。 如果你需要保留这些值，那么就在循环结束前赋值到别的变量里去。

**2016.2.18**

1.coroutine的使用

Coroutine.create()，需要传入单个参数（且只能为函数，我们称为coroutine主函数），返回coroutine对象，不会启动coroutine。

Coroutine.resume(),启动coroutine，需要将coroutine对象作为第一个参数传入该函数，接下来传入的参数作为coroutine主函数的参数

Coroutine.yield(…),可以不传入任何参数，传入的参数作为上一个Coroutine.resume()的返回值（resume()返回值固定格式：true or false，…；第一个参数代表resume函数是否为正常结束，正常结束为true，否则为false，接下来的不定参数为yield函数传入的参数）,yield()的返回值就是下一个resume()传入的参数

看下面这段代码展示的一个例子：

function foo (a)

print("foo", a)

return coroutine.yield(2\*a)

end

co = coroutine.create(function (a,b)

print("co-body", a, b)

local r = foo(a+1)

print("co-body", r)

local r, s = coroutine.yield(a+b, a-b)

print("co-body", r, s)

return b, "end"

end)

print("main", coroutine.resume(co, 1, 10))

print("main", coroutine.resume(co, "r"))

print("main", coroutine.resume(co, "x", "y"))

print("main", coroutine.resume(co, "x", "y"))

当你运行它，将得到如下输出结果：

co-body 1 10

foo 2

main true 4

co-body r

main true 11 -9

co-body x y

main true 10 end

main false cannot resume dead coroutine

2.lua堆栈的理解



1. 正数索引，我们无需知道堆栈的大小，栈底元素的索引永远为1
2. 负数索引，我们无需知道堆栈的大小，栈顶元素的索引永远为-1

# Lua优化注意事项

1. 尽可能的使用Local，因为Local变量是存放在寄存器中，并且直接索引访问，Global变量是存放在内存中，图1是lua开发者Roberto Ierusalimschy在他的《Lua Programming Gems》书中的原话；《===》图2是测试结果，几乎没有差别，（1000W次函数调用），测试都是基于Window平台，没有在真机上测试过



图1

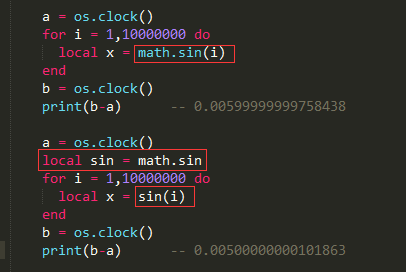
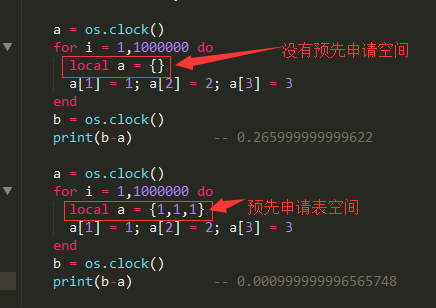


图2

1. 当你有很多很小的表需要创建的时候，你需要预填充这些表，否则当表的空间不够的时候，就会重新的申请空间，并且将原来的数据拷贝过去，所以会产生时间消耗，表的空间都是申请2^N次方。下面是测试100W次的数据，差距很明显：



1. 在大字符串连接时，我们应该尽量避免使用“..”连接方式，使用table模拟buffer，最后使用table.concat来连接字符串，测试数据图下图，性能差别还是很大的，不过一般情况用不到。

