Programming in LUA中文版

转自csdn的BLOG（译者应该是ouyangbuxiu）

序(Preface):

当前很多程序语言致力于帮你编写成千上万行的代码，所以这种类型的语言提供包、命名空间、复杂的类型系统、无数的结构、上千页的文档需要学习。

Lua并不帮你编写大量的代码的程序，相反，Lua帮你用少量的代码解决问题。为实现这个目标，像其他语言一样Lua依赖于其可扩展性。然而与其他语言不同的是，不仅用Lua编写的软件易于扩展，其他语言比如c/c++编写的软件也很容易扩展。

一开始，Lua就被设计成很容易和传统的c/c++整合的语言。这种语言的二元性带来很大的好处。Lua是一个小巧简单的语言，因为Lua不致力于做c语言已经做得很好的领域，比如：性能，底层操作以及与第三方软件的接口。Lua依赖于c去做完成这些任务。Lua所提供的机制是c不善于的：高级语言、动态结构、简洁、易于测试和调试。正因为如此，Lua具有很好的安全保证，自动内存管理，容易处理字符串以及其他大小动态改变的数据。

Lua不仅是一种易于扩展的语言，也是一种易整合语言(glue language);Lua支持基于组件的软件开发方法，我们可以将一些已经存在的高级组件整合在一起实现一个应用。一般情况下，组件使用像c/c++的静态的语言编写。Lua是我们整合各个组件的粘合剂。通常情况下，组件(或对象)表现为具体的在程序开发过程中很少变化的、占用大量CPU时间的决定性的程序，比如窗口部件和数据结构。对那种在产品的生命周期内变化很多的应用使用Lua可以方便的适用变化。除了作为整合语言外，Lua自身也是一个功能强大的语言。Lua不仅可以整合组件，还可以编辑组件甚至完全使用Lua创建组件。

除了Lua外，还有很多类似的脚本语言，比如： Perl, Tcl, Ruby, Forth, Python;下面这些特征是Lua特有的：

可扩展性：Lua的扩展性非常卓越，以至于很多人把Lua当作一个搭建领域语言的工具(注：比如游戏脚本)。Lua被设计为易于扩展的，可以通过Lua代码或者c代码扩展，Lua的很多功能都是通过外部库来扩展的。Lua很容易与c/c++, java,fortran,Smalltalk,Ada,以及其他脚本语言接口。

简单：简单，小巧；内容少但功能强大，这使得Lua易于学习，很容易实现一些小的应用。他的完全发布版(代码、手册以及某些平台的二进制文件)一张软盘就可以装得下。

高效率：Lua有很高的执行效率，统计表明Lua是目前平均效率最高的脚本语言。

平台无关：Lua几乎可以运行在所有我们听过的系统上，NextStep, OS/2, PlayStation II (Sony), Mac OS-9 and OS X, BeOS, MS-DOS, IBM mainframes, EPOC, PalmOS, MCF5206eLITE Evaluation Board, RISC OS,所有的windows和Unix；Lua不是使用条件编译实现平台无关，而是完全使用ANSI (ISO) C，这意味着只要你ANSI C编译器你就可以编译并使用Lua。

Lua大部分强大的功能来自于他的类库，这并非偶然。Lua的长处之一是可以通过新类型和函数扩展。动态类型检查最大限度允许多态，自动内存管理简化调用接口，因为这样不需要关心谁来分配内存谁来释放内存，也不需要担心溢出。高级函数和匿名函数可以接受高级参数，使函数更通用。

Lua安装可以指定标准库的一个子集。当在一个限制严格的环境下安装Lua时，认真选择你需要的类库是明智的。如果环境限制非常严格，可以很容易的到类库源代码目录下，修改源代码，保留你需要的函数即可。记住：Lua是很小的(即使加上全部的标准库)并且在大部分系统下你可以不用担心的使用全部的功能。

Lua的使用者：

Lua使用者分为三大类：使用Lua嵌入到其他应用中的、独立使用Lua的、Lua和C混合使用的。

第一：很多人使用Lua嵌入在应用中，比如CGILua (搭建动态网页) 、 LuaOrb (访问 CORBA 对象).

这些类型的应用使用Lua-API注册新函数，创建新类型，通过配置Lua就可以改变应用宿主语言的行为。通常，这种应用的使用者并不知道Lua是一种独立的语言。例如：CGILua用户一般会认为Lua是一中用于Web的语言。

第二：作为一种独立运行的语言，Lua也是很有用的，主要用于文本处理或者只运行一次的小程序。这种应用Lua主要使用它的标准库实现，标准库提供模式匹配和其他一些字串处理的功能。我们可以这样认为：Lua是对文本处理领域的嵌入式语言。

第三：还有一些使用者使用其他语言开发，把Lua当作库使用。这些人大多使用c语言开发，但使用Lua建立简单灵活的易于使用的接口。

本书面向以上三类读者。书的第一部分阐述了语言的本身，展示语言的潜在功能。我们讲述了不同的语言结构，并用一些例子展示如何解决实际问题。这部分既包括基本的语言的控制结构，也包括高级的迭代子和协同。

第二部分重点放在Lua特有的数据结构-tables,讨论了数据结构、持久性、包、面向对象编程，这里我们将看到Lua的真正强大之处。

第三部分介绍标准库。每个标准库一章：数学库，table库，string库，I/O库，OS库，Debug库。

最后一部分介绍了Lua和c接口的API，这部分介绍在c语言中开发应用而不是Lua中，对于那些打算将Lua嵌入到c/c++中的读者可能会对此部分更感兴趣。

其他资源：

参考手册是必备的，如果你真得想学一门语言，本书和Lua手册互为补充，手册描述语言本身，他不会告诉你语言的数据结构也不会举例说明，但手册是Lua的权威文档，http://www.lua.org可以得到手册的内容。

http://lua-users.org ---Lua用户社区，提供了一些第三方包和文档。

http://www.inf.puc-rio.br/~roberto/book/ --本书的更新勘误表，代码和例子。

另外本书针对Lua 5.0，如果你的版本不同，请查阅Lua手册或者版本间的差异。

约定：

1.字符串使用双引号，比如"literal strings"；单字符使用单引号，比如'a';模式串也是用单引号，比如'[%w_]*'。

2.符号 --> 表示语句的输出或者表达式的结果：

```
print(10)   --> 10
13 + 3     --> 16
```

3.符号<-->表示等价，即对于Lua来说，this 写和that 写没有区别。

```
this   <-->   that
```

关于本书：

开始打算写这本书是1998年冬天(南半球)，那时候Lua版本是3.1；2000年v4.0;2003年v5.0；Lua的变化给本书带来很大的冲击，很多章节被改写或重写，另外一些章节倍增加进来。

本书的完成必须服从语言的变化，本书在这个时候完成的原因：1.Lua 5.0是一个成熟的版本。2.语言变得越来越大，超出了最初本书的目标。另外一个原因是我想将Lua介绍给大家让更多的人了解Lua。

感谢：

在完成本书的过程中，很多人给了我极大的帮助，人名列表：xxxxxx

谢谢他们所有人。

2.Types and Values

Lua是动态类型语言,变量不要类型定义.Lua中有8个基本类型:nil, boolean, number, string, userdata,
function, thread, and table.

```
print(type("Hello world")) --> string
  print(type(10.4*3))       --> number
  print(type(print))       --> function
  print(type(type))       --> function
  print(type(true))       --> boolean
  print(type(nil))         --> nil
  print(type(type(X)))     --> string
```

变量没有预定义的类型,每一个变量都可能包含任一种类型的值.

```
print(type(a))   --> nil   (`a' is not initialized)
  a = 10
  print(type(a))   --> number
  a = "a string!!"
  print(type(a))   --> string
  a = print     -- yes, this is valid!
  a(type(a))     --> function
```

注意上面最后两行,我们可以使用function像使用其他值一样使用.一般情况下同一变量代表不同类型的值会造成混乱,最好不要用,特殊情况下可以带来便利,比如nil.

2.1 nil:Lua中特殊的类型,给全局变量负nil可以删除该变量.

2.2

booleans:两个取值false和true.但要注意Lua中所有的值都可以作为条件.在控制结构的条件中除了false和nil为假,其他值都为真.所以Lua认为0和空串都是真.

2.3

numbers:表示实数,Lua中没有整数.一般有个错误的看法CPU运算符点数比整数慢.事实不是如此,用实数代替整数不会有什么误差(除非数字大于100,000,000,000,000).Lua的numbers可以处理任何长整数不用担心误差.你也可以

在编译Lua的时候使用长整型或者单精度符点型代替numbers.数字常量的例子:

```
4   0.4   4.57e-3   0.3e12   5e+20
```

2.4 lua 是8位字节，所以字符串可以包含任何数值字符，包括嵌入的0。

这意味着你可以存储任意的2进制数据在一个字符串里.Lua中字符串是不可以修改的,你可以创建一个新的变量存放你要的字符串,如下:

```
a = "one string"
  b = string.gsub(a, "one", "another") -- change string parts
  print(a)     --> one string
  print(b)     --> another string
```

string和其他对象一样,Lua自动进行内存分配和释放,一个string可以只包含一个字母也可以包含一本书,Lua可以高效的处理长字符串,1M的string在Lua中是很常见的.

可以使用单引号或者双引号表示字符串

```
a = "a line"
  b = 'another line'
```

为了风格统一,最好使用一种,除非两种引号嵌套情况.对于字符串中含有引号的情况还可以使用转义符来表示.Lua中的转义序列有:

```
  a bell
 back space
f form feed

newline
carriage return
horizontal tab
v vertical tab
\ backslash
" double quote
' single quote
[ left square bracket
] right square bracket
```

例子:

```
> print("one line
next line
"in quotes", 'in quotes'")
  one line
  next line
  "in quotes", 'in quotes'
  > print('a backslash inside quotes: '\'')
  a backslash inside quotes: ''
  > print("a simpler way: '\'")
  a simpler way: ''
```

还可以在字符串中使用ddd(ddd为三位十进制数字)方式表示字母.

```
"alo
123"" 和 '97lo104923"'是相同的
```

还可以使用[[...]]表示字符串.这种形式的字符串可以包含多行,可以嵌套,不会解释转义序列,如果第一个字符是换行符会被自动忽略掉.这种形式的字符串用来包含一段代码是非常方便的.

```
page = [[
  <HTML>
  <HEAD>
  <TITLE>An HTML Page</TITLE>
  </HEAD>
  <BODY>
  <A HREF="http://www.lua.org">Lua</A>
  [[a text between double brackets]]
  </BODY>
  </HTML>
  ]]

  io.write(page)
```

运行时,Lua会在string和numbers之间自动进行类型转换,当一个字符串使用算术操作符时,string会被转成数字.

```
print("10" + 1)       --> 11
  print("10 + 1")       --> 10 + 1
  print("-5.3e - 10"*"2")   --> -1.06e-09
  print("hello" + 1)     -- ERROR (cannot convert "hello")
```

反过来,当Lua期望一个string而碰到数字时,会将数字转成string .

```
print(10 .. 20)     --> 1020
```

..在Lua中是字符串连接符,当在一个数字后面写..时必须加上空格防止被解释错.

尽管字符串和数字可以自动转换,两者是不同的,像 10 ==

"10"这样的比较永远都是错的.如果需要显式将string转成数字可以使用函数tonumber(),如果string不是正确的数字该函数返回nil.

```
line = io.read()   -- read a line
  n = tonumber(line)   -- try to convert it to a number
  if n == nil then
    error(line .. " is not a valid number")
  else
    print(n*2)
  end
```

反之,可以调用tostring()将数字转成字符串,这种转换一直有效:

```
print(tostring(10) == "10")   --> true
  print(10 .. "" == "10")     --> true
```

2.5 tables

Lua的tables实现了关联数组,关联数组指不仅可以通过数字检索数据,还可以通过别的类型的值检索数据.Lua中除了nil外的类型都可以作为tables的索引下标.tables是Lua主要的也是唯一的数据结构,我们可以通过他实现传统数组,

符号表, 集合, 记录(pascal), 队列,

以及其他的数据结构.Lua的包也是使用tables来描述的,io.read意味着调用io包中的read函数,对Lua而言意味着使用字符串read作为key访问io表.

Lua中tables不是变量也不是值而是对象.你可以把tables当作自动分配的对象,程序中只需要操纵表的引用(指针)即可.Lua中不需要声明表,使用最简单的{}表达式语句即可创建表.

```
a = {}   -- create a table and store its reference in `a'
  k = "x"
  a[k] = 10     -- new entry, with key="x" and value=10
  a[20] = "great" -- new entry, with key=20 and value="great"
  print(a["x"])   --> 10
  k = 20
  print(a[k])     --> "great"
  a["x"] = a["x"] + 1   -- increments entry "x"
  print(a["x"])   --> 11
```

表是匿名的,意味着表和持有表的变量没有必然的关系.

```
a = {}
  a["x"] = 10
  b = a     -- `b' refers to the same table as `a'
  print(b["x"]) --> 10
  b["x"] = 20
  print(a["x"]) --> 20
  a = nil   -- now only `b' still refers to the table
  b = nil   -- now there are no references left to the table
```

当程序中不再引用表时,这个表将被删除,内存可以重新被利用.表可以使用不同的索引类型存储值.索引大小随着表中元素个数增加而增加.

```
a = {}   -- empty table
  -- create 1000 new entries
  for i=1,1000 do a = i*2 end
  print(a[9])   --> 18
  a["x"] = 10
  print(a["x"]) --> 10
  print(a["y"]) --> nil
```

最后一行,表对应的域没有被初始化所以为nil,和全局变量一样,Lua的全局变量存储正是使用表来存储的.

可以使用域名作为索引下表访问表中元素,Lua也支持a.name代替a["name"],所以我们可以用更清晰的方式重写上面的例子:

```
a.x = 10              -- same as a["x"] = 10
  print(a.x)              -- same as print(a["x"])
  print(a.y)              -- same as print(a["y"])
```

两种方式可以混合使用,对于Lua来说,两种方式相同,但对于读者来说单一的风格更易理解.

常见的错误:混淆a.x 和a[x];第一种表示a["x"],即访问域为字符串"x"的表中元素,第二种表示使用变量x作为索引下标访问表中元素.

```
a = {}
  x = "y"
  a[x] = 10           -- put 10 in field "y"
  print(a[x])   --> 10     -- value of field "y"
  print(a.x)   --> nil   -- value of field "x" (undefined)
  print(a.y)   --> 10     -- value of field "y"
```

只要使用整数作为索引下标就可以表示传统的数组了,不需要指定数组大小:

```
-- read 10 lines storing them in a table
  a = {}
  for i=1,10 do
    a = io.read()
  end
```

当遍历数组元素时,第一个没有初始化的元素返回nil,可以用这个当作数组下标的边界标志.可以用下面的代码打印出上个例子读入的行:

```
-- print the lines
  for i,line in ipairs(a) do
    print(line)
  end
```

既然可以使用任意值作为表的下标,你可以以任何数字作为数组下标的开始,但是Lua中一般以1开始而不是0(c语言).Lua标准库也是以这个设计的.

有一点需要特别注意,否则你的代码中可能引入很多难以发现的bug.因为我们可以使用任意类型的值作为索引下标,要注意:number 0 和string

"0"是不同的,同样strings "+1", "01", and "1"也是不同的.

```
i = 10; j = "10"; k = "+10"
  a = {}
  a = "one value"
  a[j] = "another value"
  a[k] = "yet another value"
  print(a[j])         --> another value
  print(a[k])         --> yet another value
  print(a[tonumber(j)]) --> one value
  print(a[tonumber(k)]) --> one value
```

当对你检索的类型有疑问时,请使用显示类型转换.

2.6 Functions

函数是第一类值(和其他变量相同),意味着函数可以存储在变量中,可以作为函数的参数,也可以作为函数的返回值.这个特性给了语言很大的灵活性:一个程序可以重新定义函数增加新的功能或者为了避免运行不可靠代码创建安全运行环境而隐藏函数,另外这个特性在Lua实现面向对象中也起了重要作用.

Lua可以调用lua或者C实现的函数,Lua所有标准库都是用C实现的.标准库包括string库,table库,I/O库,OS库,算术库,debug库.

2.7 Userdata and Threads

userdata 可以将C数据存放在Lua变量中,userdata 在Lua中除了负值和相等比较外没有预定义的操作.userdata

用来描述应用程序或者使用C实现的库创建的新类型.例如:标准I/O库用来描述文件.下面在C API章节中我们将详细讨论.

在第九章讨论协同操作的时候,我们介绍线程.

3.Expressions

Lua中的表达式包括数字常量,字符串常量,变量,一元和二元运算符,函数调用.还可以是非传统的函数定义和表构造.

3.1 算术运算符

二元运算符:+ - * / ^ (加减乘除幂)

一元运算符:- (负值)

这些运算符的操作数都是实数.

3.2 关系运算符

< > <= >= == ~=

这些操作符返回结果为false或者true;==和~=比较两个值,如果两个值类型不同,Lua认为两者不同;nil只和自己相等.Lua通过引用比较tables,userdata,functions.也就是说当且仅当两者表示同一个对象时相等.

```
a = {}; a.x = 1; a.y = 0
  b = {}; b.x = 1; b.y = 0
  c = a

  a==c but a~=b
```

Lua比较数字按传统的数字大小进行,比较字符串按字母的顺序进行,但是字母顺序依赖于本地环境.

当比较不同类型的值的时候要特别注意:

```
"0"==0 is false
2<15 is obviously true
"2"<"15" is false (alphabetical order!).
```

为了避免不一致的结果,混合比较数字和字符串,Lua会报错,比如:2<"15

3.3 逻辑运算符

and or not

逻辑运算符认为false和nil是假(false),其他为真,0也是true.

and 和 or的运算结果不是true和false, 而是和它的两个操作数相关。

a and b:如果a为false,则返回a,否则返回b

a or b:如果a为true,则返回a,否则返回b

```
print(4 and 5)       --> 5
  print(nil and 13)     --> nil
  print(false and 13)   --> false
  print(4 or 5)       --> 4
  print(false or 5)     --> 5
```

一个很实用的技巧:如果x为false或者nil则给x赋初始值v

```
x = x or v 等价于 if not x then x = v end
```

and的优先级比or高;

C语言中的三元运算符a ? b : c 在Lua中可以这样实现:(a and b) or c

not的结果一直返回false或者true

```
print(not nil)     --> true
  print(not false)   --> true
  print(not 0)     --> false
  print(not not nil) --> false
```

3.4 连接运算符

..字符串连接,如果操作数为数字,Lua将数字转成字符串.

```
print("Hello " .. "World") --> Hello World
  print(0 .. 1)           --> 01
```

3.5 优先级

从高到低的顺序:

```
^
    not - (unary)
    *   /
    +   -
    ..
    <   >   <= >= ~= ==
    and
    or
```

除了^和..外所有的二元运算符都是左连接的.

```
a+i < b/2+1       <-->     (a+i) < ((b/2)+1)
```

```
  5+x^2*8         <-->     5+((x^2)*8)
  a < y and y <= z   <-->     (a < y) and (y <= z)
  -x^2           <-->     -(x^2)
  x^y^z           <-->     x^(y^z)
```

3.6 表的构造

构造器是创建和初始化表的表达式.表是Lua特有的功能强大的东西.最简单的构造函数是{},用来创建一个空表.可以直接初始化数组:

```
days = {"Sunday", "Monday", "Tuesday", "Wednesday", "Thursday", "Friday",
"Saturday"}
```

Lua将用string "Sunday"初始化days[1](第一个元素索引为1),用"Monday"初始化days[2]...

```
print(days[4]) --> Wednesday
```

构造函数可以使用任何表达式初始化:

```
tab = {sin(1), sin(2), sin(3), sin(4),
        sin(5), sin(6), sin(7), sin(8)}
```

如果想初始化一个表作为record使用可以这样:

```
a = {x=0, y=0} 和 a = {}; a.x=0; a.y=0等价
```

不管用何种方式创建table,我们都可以向表中添加或者删除任何类型的域,构造函数仅仅影响表的初始化.

```
w = {x=0, y=0, label="console"}
  x = {sin(0), sin(1), sin(2)}
  w[1] = "another field"
  x.f = w
  print(w["x"])   --> 0
  print(w[1])   --> another field
  print(x.f[1])   --> another field
  w.x = nil     -- remove field "x"
```

每次调用构造函数,Lua都会创建一个新的table,可以使用table构造一个list:

```
list = nil
  for line in io.lines() do
    list = {next=list, value=line}
  end
```

这段代码从标准输入读进每行,然后反序形成链表.下面的代码打印链表的内容:

```
l = list
  while l do
    print(l.value)
    l = l.next
  end
```

在同一个构造函数中可以混合列表风格和record风格进行初始化,如:

```
polyline = {color="blue", thickness=2, npoints=4,
          {x=0,   y=0},
          {x=-10, y=0},
          {x=-10, y=1},
          {x=0,   y=1}
          }
```

这个例子也表明我们可以嵌套构造函数来表示复杂的数据结构.

```
print(polyline[2].x)   --> -10
```

上面两种构造函数的初始化方式还有限制,比如你不能使用负索引初始化一个表中元素,字符串索引也不能被恰当的表示,下面介绍一种更一般的初始化方式,我们用[expression]显示的表示将被初始化的索引:

```
opnames = {["+"] = "add", ["-"] = "sub",
          ["*"] = "mul", ["/"] = "div"}

  i = 20; s = "-"
  a = {[i+0] = s, [i+1] = s..s, [i+2] = s..s..s}

  print(opnames[s])   --> sub
  print(a[22])       --> ---
```

list风格初始化和record风格初始化是这种一般初始化的特例:

```
{x=0, y=0} <-->   {["x"]=0, ["y"]=0}
  {"red", "green", "blue"} <-->   {[1]="red", [2]="green", [3]="blue"}
```

如果真的想要数组下标从0开始:

```
days = {[0]="Sunday", "Monday", "Tuesday", "Wednesday",
        "Thursday", "Friday", "Saturday"}
```

注意:不推荐数组下标从0开始,否则很多标准库不能使用.

在构造函数的最后的","是可选的,可以方便以后的扩展.

```
a = {[1]="red", [2]="green", [3]="blue",}
```

在构造函数中域分隔符逗号(",")可以用分号(";")替代,通常我们使用分号用来分割不同类型的表元素.

```
{x=10, y=45; "one", "two", "three"}
```

4.Statements

Lua像C和PASCAL几乎支持传统的语句:赋值语句,控制结构语句,函数调用等,同时也支持非传统的多变量赋值,局部变量声明.

4.1 赋值语句

赋值是改变一个变量的值和改变表域的最基本的方法.

```
a = "hello" .. "world"
  t.n = t.n + 1
```

Lua可以对多个变量同时赋值,变量列表和值列表的各个元素用逗号分开,赋值语句右边的值会依次赋给左边的变量. a, b = 10, 2*x

表示,a=10 and b=2*x

遇到赋值语句Lua会先计算右边所有的值然后再执行赋值操作,所以我们可以这样进行交换变量的值:

```
x, y = y, x           -- swap 'x' for 'y'
  a, a[j] = a[j], a   -- swap 'a' for 'a'
```

当变量个数和值的个数不一致时,Lua会一直以变量个数为基础采取以下策略,:

a.变量个数>值的个数 按变量个数补足nil

b.变量个数<值的个数 多余的值会被忽略

```
a, b, c = 0, 1
    print(a,b,c)       --> 0   1   nil
    a, b = a+1, b+1, b+2   -- value of b+2 is ignored
    print(a,b)         --> 1   2
    a, b, c = 0
    print(a,b,c)       --> 0   nil   nil
```

上面最后一个例子是一个常见的错误情况,注意:如果要对多个变量赋值必须依次对每个变量赋值 .

```
a, b, c = 0, 0, 0
print(a,b,c)       --> 0   0   0
```

多值赋值经常用来交换变量或者将函数调用返回给变量:

```
a, b = f()
```

f()返回两个值,第一个赋给a,第二个赋给b.

4.2 局部变量与代码块(block)

使用local创建一个局部变量,与全局变量不同,局部变量只在被声明的那个代码块内有效.代码块指:一个控制结构,一个函数体,或者一个chunk(变量被声明的那个文件或者文本串).

```
x = 10
  local i = 1     -- local to the chunk

  while i<=x do
    local x = i*2   -- local to the while body
    print(x)       --> 2, 4, 6, 8, ...
    i = i + 1
  end

  if i > 20 then
    local x       -- local to the "then" body
    x = 20
    print(x + 2)
  else
    print(x)       --> 10 (the global one)
  end
```

```
  print(x)       --> 10 (the global one)
```

注意,如果在交互模式下上面的例子可能不能输出期望的结果,因为第二句local i=1是一个完整的chunk,在交互模式下执行完这一句后,Lua将开始一个新的chunk,这样第二句的i已经超出了他的有效范围.可以将这段代码放在do..end(相当于c/c++的{})块中.

应该尽可能的使用局部变量,有两个好处:1.避免命名冲突2.访问局部变量的速度比全局变量更快.

我们给block划定一个明确的界限:do..end内的部分.当你想更好的控制局部变量的作用范围的时候这是很有用的.

```
do
    local a2 = 2*a
    local d = sqrt(b^2 - 4*a*c)
    x1 = (-b + d)/a2
    x2 = (-b - d)/a2
  end       -- scope of 'a2' and 'd' ends here
  print(x1, x2)
```

4.3 控制结构语句

控制结构的条件表达式结果可以是任何值,Lua认为false和nil为假,其他值为真.

if语句,有三种形式:

```
if conditions then
  then-part
end;

if conditions then
  then-part
else
  else-part
end;

if conditions then
  then-part
elseif conditions then
  elseif-part
..   --->多个elseif
else
  else-part
end;
```

while语句:

```
while condition do
  statements;
end;
```

repeat-until语句:

```
repeat
  statements;
until conditions;
```

for语句有两大类:

第一,数值for循环:

```
  for var=exp1,exp2,exp3 do
    loop-part
  end
```

for将用exp3作为step从exp1(初始值)到exp2(终止值),执行loop-part;exp3可以省略,默认step=1

有几点需要注意:

1.三个表达式只会被计算一次,并且是在循环开始前.

```
for i=1,f(x) do print(i) end
  for i=10,1,-1 do print(i) end
```

第一个例子f(x)只会在循环前被调用一次

2.控制变量var是局部变量自动被声明,并且只在循环内有效.

```
for i=1,10 do print(i) end
  max = i     -- probably wrong! 'i' here is global
```

如果需要保留控制变量的值,需要在循环中将其保存

```
-- find a value in a list
  local found = nil
  for i=1,a.n do
    if a == value then
      found = i     -- save value of 'i'
      break
    end
  end
  print(found)
```

3.循环过程中不要改变控制变量的值,那样做的结果是不可预知的.

如果要退出循环,使用break语句.

第二,范型for循环:

前面已经见过一个例子:

```
-- print all values of array 'a'
  for i,v in ipairs(a) do print(v) end
```

范型for遍历迭代子函数返回的每一个值.

再看一个遍历表key的例子:

```
-- print all keys of table 't'
  for k in pairs(t) do print(k) end
```

范型for和数值for有两点相同:1.控制变量是局部变量,2.不要修改控制变量的值

再看一个例子,假定有一个表:

```
days = {"Sunday", "Monday", "Tuesday", "Wednesday",
          "Thursday", "Friday", "Saturday"}
```

现在想把对应的名字转换成星期几,一个有效地解决问题的方式是构造一个反向表:

```
revDays = {["Sunday"] = 1, ["Monday"] = 2,
            ["Tuesday"] = 3, ["Wednesday"] = 4,
            ["Thursday"] = 5, ["Friday"] = 6,
            ["Saturday"] = 7}
```

下面就可以很容易获取问题的答案了:

```
x = "Tuesday"
    print(revDays[x])   --> 3
```

我们不需要手工,可以自动构造反向表

```
revDays = {}
    for i,v in ipairs(days) do
    revDays[v] = i
    end
```

如果你对范型for还有些不清楚在后面的章节我们会继续来学习.

4.4 break和return语句

break语句用来退出当前循环(for,repeat,while).在循环外部不可以使用.

return用来从函数返回结果,当一个函数自然结束结尾会有一个默认的return.(这种函数类似pascal的过程)

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),例如:

```
local i = 1
  while a do
    if a == v then break end
    i = i + 1
  end
```

有时候为了调试或者其他目的需要在block的中间使用return或者break,可以显式的使用do..end来实现:

```
function foo ()
      return       --<< SYNTAX ERROR
      -- 'return' is the last statement in the next block
      do return end   -- OK
      ...         -- statements not reached
    end
```

5.Functions

函数有两种用途:1.完成指定的任务,这种情况下函数作为调用语句使用2.计算并返回值,这种情况下函数作为赋值语句的表达式使用.

语法:

```
function func_name (arguments-list)
statements-list;
end;
```

调用函数的时候,如果参数列表为空,必须使用()表明是函数调用.

```
print(8*9, 9/8)
  a = math.sin(3) + math.cos(10)
  print(os.date())
```

上述规则有一个例外,当函数只有一个参数并且这个参数是字符串或者表构造的时候,()是可选的:

```
print "Hello World"   <-->   print("Hello World")
  dofile 'a.lua'       <-->   dofile ('a.lua')
  print [[a multi-line   <-->   print([[a multi-line
  message]]                  message]])
  f{x=10, y=20}       <-->   f({x=10, y=20})
  type{}              <-->   type({})
```

Lua也提供了面向对象方式调用函数的语法,比如o:foo(x)与o.foo(o, x)是等价的,后面的章节会详细介绍面向对象内容.

Lua使用的函数可以是Lua编写也可以是其他语言编写,对于Lua程序员来说用什么语言实现的函数使用起来都一样.

Lua函数实参和形参的匹配与赋值语句类似,多余部分被忽略,缺少部分用nil补足.

```
function f(a, b) return a or b end

CALL         PARAMETERS

  f(3)         a=3, b=nil
  f(3, 4)       a=3, b=4
  f(3, 4, 5)     a=3, b=4   (5 is discarded)
```

5.1 返回多个结果值

Lua函数可以返回多个结果值,一些预定义的函数返回多值比如string.find,他返回匹配串的开始和结束下标(如果不存在匹配串返回nil).

```
s, e = string.find("hello Lua users", "Lua")

  print(s, e)   --> 7     9
```

Lua函数中,在return后列出要返回的值得列表即可返回多值,如:

```
function maximum (a)
    local mi = 1       -- maximum index
    local m = a[mi]     -- maximum value
    for i,val in ipairs(a) do
    if val > m then
      mi = i
      m = val
    end
    end
    return m, mi
  end

  print(maximum({8,10,23,12,5}))   --> 23   3
```

Lua总是调整函数返回值的个数去适用调用环境,当作为一个语句调用函数时,所有返回值被忽略.

```
function foo0 () end             -- returns no results
  function foo1 () return 'a' end     -- returns 1 result
  function foo2 () return 'a','b' end   -- returns 2 results
```

第一,当作为表达式调用函数时:有以下几种情况:

1.当调用作为表达式最后一个参数或者仅有一个参数时,根据变量个数函数尽可能多地返回多个值,不足补nil,超出舍去.

2.其他情况下,函数调用仅返回第一个值(如果没有返回值为nil)

```
x,y = foo2()     -- x='a', y='b'
  x = foo2()       -- x='a', 'b' is discarded
  x,y,z = 10,foo2()   -- x=10, y='a', z='b'

x,y = foo0()     -- x=nil, y=nil
  x,y = foo1()     -- x='a', y=nil
  x,y,z = foo2()   -- x='a', y='b', z=nil

x,y = foo2(), 20     -- x='a', y=20
  x,y = foo0(), 20, 30 -- x='nil', y=20, 30 is discarded
```

第二,函数调用作为函数参数被调用时,和多值赋值是相同.

```
print(foo0())       -->
  print(foo1())       --> a
  print(foo2())       --> a   b
  print(foo2(), 1)     --> a   1
  print(foo2() .. "x")   --> ax
```

第三,函数调用在表构造函数中初始化时,和多值赋值时相同.

```
a = {foo0()}       -- a = {} (an empty table)
  a = {foo1()}       -- a = {'a'}
  a = {foo2()}       -- a = {'a', 'b'}

a = {foo0(), foo2(), 4}   -- a[1] = nil, a[2] = 'a', a[3] = 4
```

另外,return f()这种类型的返回f()返回的所有值

```
function foo (i)
    if i == 0 then return foo0()
    elseif i == 1 then return foo1()
    elseif i == 2 then return foo2()
    end
  end

  print(foo(1))   --> a
  print(foo(2))   --> a b
  print(foo(0))   -- (no results)
  print(foo(3))   -- (no results)
```

可以使用圆括号强制使调用返回一个值.

```
print((foo0()))     --> nil
  print((foo1()))     --> a
  print((foo2()))     --> a
```

一个return语句如果使用圆括号将返回值括起来也将导致返回一个值.

函数多值返回的特殊函数unpack,接受一个数组作为输入参数,返回数组的所有元素.unpack被用来实现范型调用机制,在C语言中可以使用函数指针调用可变的函数,可以声明参数可变的函数,但不能两者同时可变.在Lua中如果你想调用可变参数的可变函数只需要这样:

```
f(unpack(a))
```

unpack返回a所有的元素作为f()的参数

```
f = string.find
  a = {"hello", "ll"}
print(f(unpack(a)))   -->3 4
```

预定义的unpack函数是用C语言实现的,我们也可以用Lua来完成:

```
function unpack (t, i)
    i = i or 1
    if t then
    return t, unpack(t, i + 1)
    end
  end
```

5.2 可变参数

Lua函数可以接受可变数目的参数,和C语言类似在函数参数列表中使用三点(...)表示函数有可变的参数.Lua将函

数的参数放在一个叫arg的表中,除了参数以外,arg表中还有一个域n表示参数的个数.

例如,我们可以重写print函数:

```
printResult = ""

  function print (...)
    for i,v in ipairs(arg) do
    printResult = printResult .. tostring(v) .. " "
    end
    printResult = printResult .. "
"
  end
```

有时候我们可能需要几个固定参数加上可变参数

```
function g (a, b, ...) end

CALL         PARAMETERS

  g(3)         a=3, b=nil, arg={n=0}
  g(3, 4)       a=3, b=4, arg={n=0}
  g(3, 4, 5, 8)   a=3, b=4, arg={5, 8; n=2}
```

如上面所示,Lua会将前面的实参传给函数的固定参数,后面的实参放在arg表中.

举个具体的例子,如果我们只想要string.find返回的第二个值:

一个典型的方法是使用虚变量(下划线)

```
local _, x = string.find(s, p)
  -- now use `x'
  ...
```

还可以利用可变参数声明一个select函数:

```
function select (n, ...)
    return arg[n]
  end

print(string.find("hello hello", " hel"))       --> 6 9
  print(select(1, string.find("hello hello", " hel"))) --> 6
  print(select(2, string.find("hello hello", " hel"))) --> 9
```

有时候需要将函数的可变参数传递给另外的函数调用,可以使用前面我们说过的unpack(arg)返回arg表所有的可变参数,Lua提供了一个文本格式化的函数string.format(类似C语言的sprintf函数):

```
function fwrite (fmt, ...)
    return io.write(string.format(fmt, unpack(arg)))
  end
```

这个例子将文本格式化操作和写操作组合为一个函数.

5.3 命名参数

Lua的函数参数是和位置相关的,调用时实参会按顺序依次传给形参.有时候用名字指定参数是很有用的,比如rename函数用来给一个文件重命名,有时候我们我们记不清明名前后两个参数的顺序了:

```
-- invalid code
  rename(old="temp.lua", new="temp1.lua")
```

上面这段代码是无效的,Lua可以通过将所有的参数放在一个表中,把表作为函数的唯一参数来实现上面这段伪代码的功能.因为Lua语法支持函数调用时实参可以是表的构造.

```
rename{old="temp.lua", new="temp1.lua"}
```

根据这个想法我们重定义了rename:

```
function rename (arg)
    return os.rename(arg.old, arg.new)
  end
```

当函数的参数很多的时候,这种函数参数的传递方式很方便的.例如GUI库中创建窗体的函数有很多参数并且大部分参数是可选的,可以用下面这种方式:

```
w = Window{ x=0, y=0, width=300, height=200,
           title = "Lua", background="blue",
           border = true
         }
```

```
function Window (options)
    -- check mandatory options
    if type(options.title) ~= "string" then
    error("no title")
    elseif type(options.width) ~= "number" then
    error("no width")
    elseif type(options.height) ~= "number" then
    error("no height")
    end

    -- everything else is optional
    _Window(options.title,
        options.x or 0,   -- default value
        options.y or 0,   -- default value
        options.width, options.height,
        options.background or "white",   -- default
        options.border     -- default is false (nil)
        )
  end
```

```
    1999_StarCraft
    2002_Diablo II
    2003_WarCraft III
    2004_World of Warcraft
    2007_HellGate:London
```

Posted: 2005-03-12 18:04 | 4 楼

6.More about Functions

Lua中的函数是带有词法定界(lexical scoping)的第一类值(first-class values).

第一类值指:在Lua中函数和其他值(数值,字符串)一样,函数可以被存放在变量中,也可以存放在表中,可以作为函数的参数,还可以作为函数的返回值.

词法定界指:函数可以访问他内部嵌套的函数中的变量.这一特性给Lua提供了强大的编程能力.

Lua中关于函数稍微难以理解的是函数也可以没有名字,匿名的.当我们提到函数名(比如print),实际上是说一个指向函数的变量,像其持有其他类型值的变量一样

```
a = {p = print}
  a.p("Hello World") --> Hello World
  print = math.sin -- `print' now refers to the sine function
  a.p(print(1))   --> 0.841470
  sin = a.p       -- `sin' now refers to the print function
  sin(10, 20)     --> 10     20
```

既然函数是值,那么表达式也可以创建函数了,Lua中我们经常这样写:

```
function foo (x) return 2*x end
```

这实际上是利用Lua提供的"语法上的甜头"(syntactic sugar)的结果,下面是原本的函数:

```
foo = function (x) return 2*x end
```

函数定义实际上是一个赋值语句,将类型为function的变量赋给一个变量.我们使用function (x) ... end来定义一个函数和使用{}创建一个表一样.

table标准库提供一个排序函数,接受一个表作为输入参数并且排序表中的元素.这个函数必须能够对不同类型的值(字符串或者数值)按升序或者降序进行排序.Lua不是尽可能多地提供参数来满足这些情况的需要,而是接受一个排序函数作为参数(类似C++的函数对象),排序函数接受两个排序元素作为输入参数,并且返回两者的大小关系,例如:

```
network = {
    {name = "grauna", IP = "210.26.30.34"},
    {name = "arraial", IP = "210.26.30.23"},
    {name = "lua",   IP = "210.26.23.12"},
    {name = "derain", IP = "210.26.23.20"},
    }
```

如果我们想通过表的name域排序:

```
table.sort(network, function (a,b)
    return (a.name > b.name)
  end)
```

以其他函数作为参数的函数在Lua中被称作高级函数,高级函数在Lua中并没有特权,只是Lua把函数当作第一类函数处理的一个简单的结果.

下面给出一个绘图函数的例子:

```
function eraseTerminal ()
        io.write("27[2J")
        end

        -- writes an `*' at column `x' , row `y'
        function mark (x,y)
        io.write(string.format("27[%d;%dH*", y, x))
        end

        -- Terminal size
        TermSize = {w = 80, h = 24}

        -- plot a function
        -- (assume that domain and image are in the range [-1,1])
        function plot (f)
        eraseTerminal()
        for i=1,TermSize.w do
          local x = (i/TermSize.w)*2 - 1
          local y = (f(x) + 1)/2 * TermSize.h
          mark(i, y)
        end
        io.read() -- wait before spoiling the screen
        end
```

要想让这个例子正确的运行,你必须调整你的终端类型和代码中的控制符一致

```
plot(function (x) return math.sin(x*2*math.pi) end)
```

将在屏幕上输出一个正弦曲线.

将第一类值函数应用在表中是Lua实现面向对象和包机制的关键,这部分内容在后面章节介绍

6.1 闭包

当一个函数内部嵌套以一个函数定义时,内部的函数体可以访问外部的函数的局部变量,这种特征我们称作词法定界.虽然这看起来很清楚,事实并非如此,词法定界加上第一类函数在编程语言里是一个功能强大的概念,很少语言提供这种支持.

下面看一个简单的例子,假定有一个学生姓名的列表和一个学生名和成绩对应的表;现在想根据学生的成绩从高到低对学生进行排序,可以这样做:

```
names = {"Peter", "Paul", "Mary"}
  grades = {Mary = 10, Paul = 7, Peter = 8}
  table.sort(names, function (n1, n2)
    return grades[n1] > grades[n2]   -- compare the grades
  end)
```

假定创建一个函数实现此功能:

```
function sortbygrade (names, grades)
    table.sort(names, function (n1, n2)
    return grades[n1] > grades[n2]   -- compare the grades
    end)
  end
```

例子中包含在sortbygrade函数内部的sort中的匿名函数可以访问sortbygrade的参数grades,在匿名函数内部grades不是全局变量也不是局部变量,我们称作外部的局部变量(external

local variable)或者upvalue.(upvalue意思有些误导,然而在Lua中他的存在有历史的根源,还有他比起external

local variable简短).

看下面的代码 :

```
function newCounter ()
    local i = 0
```

```
    return function ()   -- anonymous function
          i = i + 1
          return i
        end
  end

  c1 = newCounter()
  print(c1()) --> 1
  print(c1()) --> 2
```

匿名函数使用upvalue

i保存他的计数,当我们调用匿名函数的时候i已经超出了作用范围,因为创建i的函数newCounter已经返回了.然而Lua用闭包的思想正确处理了这种情况.简单的说闭包是一个函数加上它可以正确访问的upvalues.如果我们再次调用newCounter,将创建一个新的局部变量i,因此我们得到了一个作用在新的变量i上的新闭包.

```
c2 = newCounter()
  print(c2()) --> 1
  print(c1()) --> 3
  print(c2()) --> 2
```

c1,c2是建立在同一个函数上,但作用在同一个局部变量的不同实例上的两个不同的闭包.

技术上来讲,闭包指值而不是指函数,函数仅仅是闭包的一个原型声明;尽管如此,在不会导致混淆的情况下我们继续使用术语函数代指闭包.

闭包在上下文环境中提供很有用的功能,如前面我们见到的可以作为高级函数(sort)的参数;作为函数嵌套的函数(newCounter).这一机制使得我们可以在Lua的函数世界里组合出奇幻的编程技术.闭包也可用在回调函数中,比如在GUI环境中你需要创建一系列button,但用户按下button时回调函数被调用,可能不同的按钮被按下时需要处理的任务有点区别.具体来讲,一个十进制计算器需要10个相似的按钮,每个按钮对应一个数字,可以使用下面的函数创建他们:

```
function digitButton (digit)
    return Button{ label = digit,
              action = function ()
                    add_to_display(digit)
                    end
             }
  end
```

这个例子中我们假定Button是一个用来创建新按钮的工具,label是按钮的标签,action是按钮被按下时调用的回调函数.(实际上是一个闭包,因为他访问upvalue

digit).digitButton完成任务返回后,局部变量digit超出范围,回调函数仍然可以被调用并且可以访问局部变量digit.

闭包在完全不同的上下文中也是很有用途的.因为函数被存储在普通的变量内我们可以很方便的重定义或者预定义函数.通常当你需要原始函数有一个新的实现时可以重定义函数.例如你可以重定义sin使其接受一个度数而不是弧度作为参数:

```
oldSin = math.sin
  math.sin = function (x)
    return oldSin(x*math.pi/180)
  end
```

更清楚的方式:

```
do
    local oldSin = math.sin
    local k = math.pi/180
    math.sin = function (x)
    return oldSin(x*k)
    end
  end
```

这样我们把原始版本放在一个局部变量内,访问sin的唯一方式是通过新版本.

利用同样的特征我们可以创建一个安全的环境(也称作沙箱,和java里的沙箱一样),当我们运行一段不信任的代码(比如我们运行网络服务器上获取的代码)时安全的环境是需要的,比如我们可以使用闭包重定义io库的open函数来限制程序打开的文件.

```
do
    local oldOpen = io.open
    io.open = function (filename, mode)
```

```
    if access_OK(filename, mode) then
      return oldOpen(filename, mode)
    else
      return nil, "access denied"
    end
    end
  end
```

6.2 非全局函数

Lua中函数可以作为全局变量也可以作为局部变量,我们已经看到一些例子:函数作为table的域(大部分Lua标准库使用这种机制来实现的比如io.read;math.sin).这种情况下,必须注意函数和表语法:

1.表和函数放在一起

```
Lib = {}
  Lib.foo = function (x,y) return x + y end
  Lib.goo = function (x,y) return x - y end
```

2.使用表构造函数

```
Lib = {
    foo = function (x,y) return x + y end,
    goo = function (x,y) return x - y end
  }
```

3.Lua提供另一种语法方式

```
Lib = {}
  function Lib.foo (x,y)
    return x + y
  end
  function Lib.goo (x,y)
    return x - y
  end
```

当我们将函数保存在一个局部变量内时,我们得到一个局部函数,也就是说局部函数像局部变量一样在一定范围内有效.这种定义在包中是非常有用的:因为Lua把chunk当作函数处理,在chunk内可以声明局部函数(仅仅在chunk内可见),词法定界保证了包内的其他函数可以调用此函数.下面是声明局部函数的两种方式:

1.方式一

```
local f = function (...)
    ...
  end

  local g = function (...)
    ...
    f()   -- external local `f' is visible here
    ...
  end
```

2.方式二

```
local function f (...)
    ...
  end
```

有一点需要注意的是在声明递归局部函数的方式:

```
  local fact = function (n)
    if n == 0 then return 1
    else return n*fact(n-1)   -- buggy
    end
  end
```

上面这种方式导致Lua编译时遇到fact(n-1)并不知道他是局部函数fact,Lua会去查找是否有这样的全局函数fact.为了解决这个问题我们必须在定义函数以前先声明:

```
  local fact
  fact = function (n)
    if n == 0 then return 1
    else return n*fact(n-1)
    end
```

```
  end
```

这样在fact内部fact(n-1)调用是一个局部函数调用,运行时fact就可以获取正确的值了.

但是Lua扩展了他的语法使得可以在直接递归函数定义时使用两种方式都可以.

在定义非直接递归局部函数时要先声明然后定义才可以:

```
local f, g   -- `forward' declarations

    function g ()
      ... f() ...
    end

    function f ()
      ... g() ...
    end
```

6.3 正确的尾调用(原文:Proper Tail Calls)

Lua中函数的另一个有趣的特征是可以正确的处理尾调用.(一些作者使用术语尾递归[原文:proper tail recursion],虽然并未涉及到递归的概念) .

尾调用是一种类似在函数结尾的goto调用,当函数最后一个动作是调用另外一个函数时,我们称这种调用尾调用.例如:

```
function f (x)
    return g(x)
  end
```

g的调用是尾调用.

例子中f调用g后不会再做任何事情,这种情况下当被调用函数g结束时程序不需要返回到调用者f;所以尾调用之后程序不需要在栈中保留关于调用者的任何信息.一些编译器比如Lua解释器利用这种特性在处理尾调用时不使用额外的栈,我们称这种语言支持正确的尾调用.

由于尾调用不需要使用栈空间,那么尾调用递归的层次可以无限制的.例如下面调用不论n为何值不会导致栈溢出.

```
  function foo (n)
    if n > 0 then return foo(n - 1) end
  end
```

需要注意的是:明确什么是尾调用.

一些调用者函数调用其他函数后也没有做其他的事情但不属于尾调用.比如:

```
function f (x)
        g(x)
        return
    end
```

上面这个例子中f在调用g后,不得不丢弃g地返回值,所以不是尾调用,同样的下面几个例子也不时尾调用:

```
  return g(x) + 1   -- must do the addition
    return x or g(x)   -- must adjust to 1 result
    return (g(x))     -- must adjust to 1 result
```

Lua中类似return g(...)

这种格式的调用是尾调用.但是g和g的参数都可以是复杂表达式,因为Lua会在调用之前计算表达式的值.例如下面的调用是尾调用:

```
  return x.foo(x[j] + a*b, i + j)
```

可以将为调用理解成一种goto,Lua中尾调用的用处在于有关状态机的编程.状态机的应用要求函数记住每一个状态,改变状态只需要goto(or call)一个特定的函数.我们考虑一个迷宫游戏作为例子:迷宫有很多个房间,每个房间有东西南北四个门,每一步输入一个移动的方向,如果该方向存在即到达该方向对应的房间,否则程序打印警告信息.目标是:从开始的房间到达目的房间.

这个迷宫游戏是典型的状态机,每个当前的房间是一个状态.我们可以对每个房间写一个函数实现这个迷宫游戏,我们使用尾调用从一个房间移动到另外一个房间.一个四个房间的迷宫代码如下:

```
function room1 ()
    local move = io.read()
    if move == "south" then return room3()
    elseif move == "east" then return room2()
    else print("invalid move")
      return room1()   -- stay in the same room
```

```
  end
end

function room2 ()
  local move = io.read()
  if move == "south" then return room4()
  elseif move == "west" then return room1()
  else print("invalid move")
    return room2()
  end
end

function room3 ()
  local move = io.read()
  if move == "north" then return room1()
  elseif move == "east" then return room4()
  else print("invalid move")
    return room3()
  end
end

function room4 ()
  print("congratilations!")
end
```

我们可以调用room1()开始这个游戏.

如果没有正确的尾调用,每次移动都要创建一个栈,多次移动后可能导致栈溢出.但正确的尾调用可以无限制的尾调用,因为每次尾调用只是一个goto到另外一个函数并不是传统的函数调用.

7.Iterators and the Generic for

在这一章我们讨论为范性for写迭代器, 我们从一个简单的迭代器开始,然后我们学习如何通过利用范性for的强大之处写出更高效的迭代器.

7.1 迭代器与闭包

迭代器是一种支持指针类型的结构,它可以使遍历集合的每一个元素.在Lua中我们常常使用函数来描述迭代器,每次调用该函数就返回集合的下一个元素.

迭代器需要保留上一次成功调用的状态和下一次成功调用的状态,也就是他知道来自于哪里和将要前往哪里.闭包提供的机制可以很容易实现这个任务.记住:闭包是一个内部函数,它可以访问一个或者多个外部函数的外部局部变量.每次闭包的成功调用后这些外部局部变量都保存他们的值(状态).当然如果要创建一个闭包必须要创建其外部局部变量.所以一个典型的闭包的结构包含两个函数:一个是闭包自己;另一个是工厂(创建闭包的函数).

举一个简单的例子,我们为一个list写一个简单的迭代器,与ipairs()不同的是我们实现的这个迭代器返回元素的值而不是索引下标:

```
function list_iter (t)
    local i = 0
    local n = table.getn(t)
    return function ()
          i = i + 1
          if i <= n then return t end
        end
  end
```

这个例子中list_iter

是一个工厂,每次调用他都会创建一个新的闭包(迭代器本身).闭包包村内部局部变量(t,i,n),因此每次调用他返回list中的下一个元素值,当list中没有值时,返回nil.我们可以在while语句中使用这个迭代器:

```
t = {10, 20, 30}
  iter = list_iter(t)   -- creates the iterator
  while true do
    local element = iter()   -- calls the iterator
    if element == nil then break end
    print(element)
```

```
  end
```

我们设计的这个迭代器也很容易用于范性for语句

```
t = {10, 20, 30}
  for element in list_iter(t) do
    print(element)
  end
```

范性for为迭代循环处理所有的薄记(bookkeeping):首先调用迭代工厂;内部保留迭代函数,因此我们不需要iter变量;然后在每一个新的迭代处调用迭代器函数;当迭代器返回nil时循环结束(后面我们将看到范性for能胜任更多的任务).

下面看一个稍微高级一点的例子:我们写一个迭代器遍历一个文件内的所有匹配的单词.为了实现目的,我们需要保留两个值:当前行和在当前行的偏移量,我们使用两个外部局部变量line,pos保存这两个值.

```
function allwords ()
    local line = io.read() -- current line
    local pos = 1       -- current position in the line
    return function ()     -- iterator function
      while line do       -- repeat while there are lines
      local s, e = string.find(line, "%w+", pos)
      if s then       -- found a word?
        pos = e + 1     -- next position is after this word
        return string.sub(line, s, e)   -- return the word
      else
        line = io.read() -- word not found; try next line
        pos = 1       -- restart from first position
      end
      end
      return nil         -- no more lines: end of traversal
    end
  end
```

迭代函数的主体部分调用了string.find函数,string.find在当前行从当前位置开始查找匹配的单词,例子中匹配的单词使用模式'%w+'描述的;如果查找到一个单词,迭代函数更新当前位置pos为单词后的第一个位置,并且返回这个单词(string.sub函数从line中提取两个位置参数之间的子串).否则迭代函数读取新的一行并重新搜索.如果没有line可读返回nil结束.

尽管迭代函数有些复杂,但使用起来是很直观的:

```
for word in allwords() do
    print(word)
  end
```

通常情况下,迭代函数都难写易用.这不是一个大问题:一般Lua编程不需要自己定义迭代函数,而是使用语言提供的,除非确实需要自己定义.

7.2 范性for的语义

前面我们看到的迭代器有一个缺点:每次调用都需要创建一个闭包,大多数情况下这种做法都没什么问题,例如在allwords迭代器中创建一个闭包的代价比起读整个文件来说微不足道,然后在有些情况下创建闭包的代价是不能忍受的.在这些情况下我们可以使用范性for本身来保存迭代的状态.

前面我们看到在循环过程中范性for在自己内部保存迭代函数,实际上它保存三个值:迭代函数,状态常量和控制变量.下面详细说明.

范性for的文法如下:

```
for <var-list> in <exp-list> do
    <body>
  end
```

<var-list>是一个或多个以逗号分割变量名的列表,<exp-list>是一个或多个以逗号分割的表达式列表,通常情况下exp-list只有一个值:迭代工厂的调用.

```
for k, v in pairs(t) do
    print(k, v)
    end
```

变量列表k,v;表达式列表pair(t),在很多情况下变量列表也只有一个变量,比如:

```
for line in io.lines() do
    io.write(line, '
')
```

```
  end
```
我们称变量列表中第一个变量为控制变量,其值为nil时循环结束.

下面我们看看范性for的执行过程:

首先,初始化,计算in后面表达式的值,表达式应该返回范性for需要的三个值:迭代函数,状态常量和控制变量;与多值赋值一样,如果表达式返回的结果个数不足三个会自动用nil补足，多出部分会被忽略.

第二,将状态常量和控制变量作为参数调用迭代函数(注意:对于for结构来说,状态常量没有用处,仅仅在初始化时获取他的值并传递给迭代函数).

第三,将迭代函数返回的值赋给变量列表.

第四,如果返回的第一个值为nil循环结束,否则执行循环体.

第五,回到第二步再次调用迭代函数.

更精确的来说:

```
for var_1, ..., var_n in explist do block end
```
等价于
```
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
```
如果我们的迭代函数是f，状态常量是s,控制变量的初始值是a0,那么控制变量将循环:a1=f(s,a0);a2=f(s,a1);...直到ai=nil

7.3 无状态的迭代器

无状态的迭代器是指不保留任何状态的迭代器,因此在循环中我们可以利用无状态迭代器避免创建闭包花费额外的代价.

每一次迭代,迭代函数都是用两个变量(状态常量和控制变量)的值作为参数被调用,一个无状态的迭代器只利用这两个值可以获取下一个元素.这种无状态迭代器的典型的简单的例子是ipairs,他遍历数组的每一个元素.
```
  a = {"one", "two", "three"}
  for i, v in ipairs(a) do
    print(i, v)
  end
```
迭代的状态包括被遍历的表(循环过程中不会改变的状态常量)和当前的索引下标(控制变量),ipairs和迭代函数都很简单,我们在Lua中可以这样实现:
```
function iter (a, i)
    i = i + 1
    local v = a
    if v then
      return i, v
    end
  end

  function ipairs (a)
    return iter, a, 0
  end
```
当Lua调用ipairs(a)开始循环时,他获取三个值:迭代函数iter,状态常量a和控制变量初始值0;然后Lua调用iter(a,0)返回1,a[1](除非a[1]=nil);第二次迭代调用iter(a,1)返回2,a[2]...直到第一个非nil元素.

Lua库中实现的pairs是一个用next实现的原始方法:
```
function pairs (t)
    return next, t, nil
  end
```
还可以不使用ipairs直接使用next
```
for k, v in next, t do
    ...
```

```
  end
```

记住:exp-list返回结果会被调整为三个,所以Lua获取next,t,nil;确切地说当他调用pairs时获取.

7.4 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是简单的状态常量和控制变量,最简单的方法是使用闭包,还有一种方法就是将所有的状态信息封装到table内,将table作为迭代器的状态常量,因为这种情况下可以将所有的信息存放在table内,所以迭代函数通常不需要第二个参数.

下面我们重写allwords迭代器,这一次我们不是使用闭包而是使用带有两个域(line,pos)的table.

开始迭代的函数是很简单的,他必须返回迭代函数和初始状态:

```
local iterator   -- to be defined later

  function allwords ()
    local state = {line = io.read(), pos = 1}
    return iterator, state
  end
```

真正的处理工作是在迭代函数内完成:

```
function iterator (state)
    while state.line do     -- repeat while there are lines
      -- search for next word
      local s, e = string.find(state.line, "%w+", state.pos)
      if s then           -- found a word?
      -- update next position (after this word)
      state.pos = e + 1
      return string.sub(state.line, s, e)
      else   -- word not found
      state.line = io.read() -- try next line...
      state.pos = 1       -- ... from first position
      end
    end
    return nil           -- no more lines: end loop
  end
```

我们应该尽可能的写无状态的迭代器,因为这样循环的时候由for来保存状态,不需要创建对象花费的代价小;如果不能用无状态的迭代器实现,应尽可能使用闭包;尽可能不要使用table这种方式,因为创建闭包的代价要比创建table小,另外Lua处理闭包要比处理table速度快些.后面我们还将看到另一种使用协同来创建迭代器的方式,这种方式功能更强但更复杂.

7.4 真正的迭代器

迭代器的名字有一些误导,因为它并没有迭代,完成迭代功能的是for语句,也许更好的叫法应该是'生成器';但是在其他语言比如java,C++迭代器的说法已经很普遍了,我们也将沿用这种术语.

有一种方式创建一个在内部完成迭代的迭代器.这样当我们使用迭代器的时候就不需要使用循环了;我们仅仅使用每一次迭代需要处理的任务作为参数调用迭代器即可,具体地说,迭代器接受一个函数作为参数,并且这个函数在迭代器内部被调用.

作为一个具体的例子,我们使用上述方式重写allwords迭代器:

```
function allwords (f)
    -- repeat for each line in the file
    for l in io.lines() do
      -- repeat for each word in the line
      for w in string.gfind(l, "%w+") do
      -- call the function
      f(w)
      end
    end
  end
```

如果我们想要打印出单词,只需要

```
allwords(print)
```

更一般的做法是我们使用匿名函数作为作为参数,下面的例子打印出单词'hello'出现的次数:

```
local count = 0
  allwords(function (w)
    if w == "hello" then count = count + 1 end
```

```
  end)
  print(count)
```

用for结构完成同样的任务:

```
local count = 0
  for w in allwords() do
    if w == "hello" then count = count + 1 end
  end
  print(count)
```

真正的迭代器风格的写法在Lua老版本中很流行,那时还没有for循环.

两种风格的写法相差不大,但也有区别:一方面,第二种风格更容易书写和理解;另一方面,for结构更灵活,可以使用break和continue语句

;在真正的迭代器风格写法中return语句只是从匿名函数中返回而不是退出循环.

8.Compilation, Execution, and Errors

虽然我们把Lua当作解释型语言,但是Lua会首先把代码预编译成中间码然后再执行(很多解释型语言都是这么做的).在解释型语言中存在编译阶段听起来不合适,然而,解释型语言的特征不在于他们是否被编译，而是编译器是语言运行时的一部分,所以，执行编译产生的中间码速度会更快.我们可以说函数dofile的存在就是说明可以将Lua作为一种解释型语言被调用.

前面我们介绍过dofile,把它当作Lua运行代码的chunk的一种原始的操作.dofile实际上是一个辅助的函数,真正完成功能的函数是loadfile;与dofile不同的是loadfile编译代码成中间码并且返回编译后的chunk作为一个函数,而不执行代码;另外loadfile不会抛出错误信息而是返回错误代码.我们可以这样定义dofile:

```
function dofile (filename)
    local f = assert(loadfile(filename))
    return f()
  end
```

如果loadfile失败assert会抛出错误.

完成简单的功能dofile比较方便,他读入文件编译并且执行.然而loadfile更加灵活.在发生错误的情况下,loadfile返回nil和错误信息,这样我们就可以自定义错误处理.另外,如果我们运行一个文件多次的话,loadfile只需要编译一次,但可多次运行.dofile却每次都要编译.

loadstring与loadfile相似,只不过它不是从文件里读入chunk,而是从一个串中读入.例如:

```
f = loadstring("i = i + 1")
```

f将是一个函数,调用时执行:i=i+1:

```
i = 0
  f(); print(i)   --> 1
  f(); print(i)   --> 2
```

loadstring函数功能强大,但使用时需多加小心.确认没有其它简单的解决问题的方法再使用.

Lua把每一个chunk都作为一个匿名函数处理.例如:chunk "a = 1",loadstring返回与其等价的function () a =

```
1 end
```

与其他函数一样,chunks可以定义局部变量也可以返回值:

```
f = loadstring("local a = 10; return a + 20")
  print(f())       --> 30
```

loadfile和loadstring都不会抛出错误,如果发生错误他们将返回nil加上错误信息:

```
print(loadstring("i i"))
    --> nil   [string "i i"]:1: `=' expected near `i'
```

另外,loadfile和loadstring都不会有边界效应产生,他们仅仅编译chunk成为自己内部实现的一个匿名函数.通常对他们的误解是他们定义了函数.Lua中的函数定义是发生在运行时的赋值而不是发生在编译时.假如我们有一个文件foo.lua:

```
-- file `foo.lua'
  function foo (x)
    print(x)
  end
```

当我们执行命令f = loadfile("foo.lua")后,foo被编译了但还没有被定义,如果要定义他必须运行chunk:

```
f()       -- defines `foo'
  foo("ok")   --> ok
```

如果你想快捷的调用dostring (比如加载并运行)可以调用loadstring

返回的结果loadstring(s)(),然而如果加载的内容存在语法错误的话,loadstring返回nil和错误信息(attempt to
call a nil value);为了返回更清楚的错误信息可以使用assert:

```
assert(loadstring(s))()
```

通常使用loadstring加载一个字串没什么意义,例如:

```
f = loadstring("i = i + 1")
```

大概与f = function () i = i + 1
end等价,但是第二段代码速度更快因为它只需要编译一次,第一段代码每次调用loadstring都会重新编译,还有一个重要区别:loadstring编译的时候不关心词法范围:

```
local i = 0
  f = loadstring("i = i + 1")
  g = function () i = i + 1 end
```

这个例子中,和想象的一样g使用局部变量i,然而f使用全局变量i;loadstring总是在全局环境中编译他的串.
loadstring通常用于运行程序外部的代码,比如运行用户自定义的代码.注意:loadstring期望一个chunk,即语句.如果想要加载表达式,需要在表达式前加return,那样将返回表达式的值.看例子:

```
print "enter your expression:"
  local l = io.read()
  local func = assert(loadstring("return " .. l))
  print("the value of your expression is " .. func())
```

loadstring返回的函数和普通函数一样,可以多次被调用

```
print "enter function to be plotted (with variable `x'):"
  local l = io.read()
  local f = assert(loadstring("return " .. l))
  for i=1,20 do
    x = i   -- global `x' (to be visible from the chunk)
    print(string.rep("*", f()))
  end
```

8.1 require函数
Lua提供高级的require函数来加载运行库.粗略的说require和dofile完成同样的功能但有两点不同:1.require会搜索目录加载文件2.require会判断是否文件已经加载避免重复加载同一文件.由于上述特征,require在Lua中是加载库的更好的函数.
require使用的路径和普通我们看到的路径还有些区别,我们一般见到的路径都是一个目录列表.require的路径是一个模式列表,每一个模式指明一种由虚文件名(require的参数)转成实文件名的方法.更明确地说,每一个模式是一个包含可选的问号的文件名.匹配的时候Lua会首先将问号用虚文件名替换,然后看是否有这样的文件存在.如果不存在继续用同样的方法用第二个模式匹配.例如,路径如下:

```
?;?.lua;c:windows?;/usr/local/lua/?/?.lua
```

调用require"lili"时会试着打开这些文件:

```
lili
  lili.lua
  c:windowslili
  /usr/local/lua/lili/lili.lua
```

require关注的问题只有分号(模式之间的分隔符)和问号,其他的信息(目录分隔符,文件扩展名)在路径中定义.
为了确定路径,Lua首先检查全局变量LUA_PATH是否为一个字符串,如果是则认为这个串就是路径;否则require检查环境变量LUA_PATH的值,如果两个都失败require使用固定的路径(典型的
"?;?.lua")
require的另一个功能是避免重复加载同一个文件两次.Lua保留一张所有已经加载的文件的列表(使用table保存).如果一个加载的文件在表中存在require简单的返回;表中保留加载的文件的虚名,而不是实文件名.所以如果你使用不同的虚文件名require同一个文件两次,将会加载两次该文件.比如require
"foo"和require
"foo.lua",路径为"?;?.lua"将会加载foo.lua两次.我们也可以通过全局变量_LOADED访问文件名列表,这样我们就可以判断文件是否被加载过;同样我们也可以使用一点小技巧让require加载一个文件两次.比如,require
"foo"之后_LOADED["foo"]将不为nil,我们可以将其赋值为nil,require "foo.lua"将会再次加载该文件.
一个路径中的模式也可以不包含问号而只是一个固定的路径,比如:

```
?;?.lua;/usr/local/default.lua
```

这种情况下,require没有匹配的时候就会使用这个固定的文件(当然这个固定的路径必须放在模式列表的最后才有

意义).在require运行一个chunk以前,它定义了一个全局变量_REQUIREDNAME用来保存被required的虚文件的文件名.我们可以通过使用这个技巧扩展require的功能.举个极端的例子,我们可以把路径设为"/usr/local/lua/newrequire.lua",这样以后每次调用require都会运行newrequire.lua,这种情况下可以通过使用_REQUIREDNAME的值去实际加载required的文件.

8.2 C-包

Lua和C是很容易结合的,使用C为Lua写包.与Lua中写包不同,C包在使用以前必须首先加载并连接.在大多数系统中最容易的实现方式是通过动态连接库机制,然而动态连接库不是ANSI

C的一部分,也就是说在标准C中实现动态连接是很困难的.

通常Lua不包含任何不能用标准C实现的机制,动态连接库是一个特例.我们可以将动态连接库机制视为其他机制之母:一旦我们拥有了动态连接机制,我们就可以动态的加载Lua中不存在的机制.所以,在这种特殊情况下,Lua打破了他平台兼容的原则而通过条件编译的方式为一些平台实现了动态连接机制.标准的Lua为windows,Linux,FreeBSD,Solaris和其他一些Unix平台实现了这种机制,扩展其它平台支持这种机制也是不难的.在Lua提示符下运行print(loadlib())看返回的结果如果显示bad

arguments则说明你的发布版支持动态连接机制,否则说明动态连接机制不支持或者没有安装.

Lua在一个叫loadlib的函数内提供了所有的动态连接的功能.这个函数有两个参数:库的绝对路径和初始化函数.所以典型的调用的例子如下:

```
local path = "/usr/local/lua/lib/libluasocket.so"
  local f = loadlib(path, "luaopen_socket")
```

loadlib函数加载指定的库并且连接到Lua,然而它并不打开库(也就是说没有调用初始化函数),反之他返回初始化函数作为Lua的一个函数,这样我们就可以直接在Lua中调用他.如果加载动态库或者查找初始化函数时出错,loadlib将返回nil和错误信息.我们可以修改前面一段代码,使其检测错误然后调用初始化函数:

```
local path = "/usr/local/lua/lib/libluasocket.so"
  -- or path = "C:\windows\luasocket.dll"
  local f = assert(loadlib(path, "luaopen_socket"))
  f() -- actually open the library
```

一般情况下我们期望二进制的发布库包含一个与前面代码段相似的stub文件,安装二进制库的时候可以随便放在某个目录,只需要修改stub文件对应二进制库的实际路径即可.将stub文件所在的目录加入到LUA_PATH,这样设定后就可以使用require函数加载C库了.

8.3 错误

Errare humanum est(拉丁谚语:犯错是人的本性).

所以我们要尽可能的处理错误,Lua经常作为扩展语言嵌入在别的应用中,所以不能当错误发生时简单的崩溃或者退出.相反,当错误发生时Lua结束当前的chunk并返回到应用中.

当Lua遇到不期望的情况时就会抛出错误,比如:两个非数字进行相加;调用一个非函数的变量;访问表中不存在的值等(可以通过metatables修改这种行为,后面介绍).你也可以通过调用error函数显示的抛出错误,error的参数是要抛出的错误信息.

```
print "enter a number:"
  n = io.read("*number")
  if not n then error("invalid input") end
```

Lua提供了专门的内置函数assert来完成上面类似的功能:

```
print "enter a number:"
  n = assert(io.read("*number"), "invalid input")
```

assert首先检查第一个参数是否返回错误,如果不返回错误assert简单的返回,否则assert以第二个参数抛出错误信息.第二个参数是可选的.注意assert是普通的函数,他会首先计算两个参数然后再调用函数,所以以下代码:

```
n = io.read()
  assert(tonumber(n),
         "invalid input: " .. n .. " is not a number")
```

将会总是进行连接操作,使用显示的test可以避免这种情况.

当函数遇到异常有两个基本的动作:返回错误代码或者抛出错误.这两种方式选择哪一种没有固定的规则,但有一般的原则:容易避免的异常应该抛出错误否则返回错误代码.

例如我们考虑sin函数,如果以一个table作为参数,假定我们返回错误代码,我们需要检查错误的发生,代码可能如下:

```
local res = math.sin(x)
  if not res then   -- error
    ...
```

然而我们可以在调用函数以前很容易的判断是否有异常:

```
if not tonumber(x) then   -- error: x is not a number
    ...
```

然而通常情况下我们既不是检查参数也不是检查返回结果,因为参数错误可能意味着我们的程序某个地方存在问题,这种情况下,处理异常最简单最实际的方式是抛出错误并且终止代码的运行.

再来看一个例子io.open函数用来打开一个文件,如果文件不存在结果会怎么样呢?很多系统中,通过试着去打开文件来判断是否文件存在.所以如果io.open不能打开文件(由于文件不存在或者没有权限),函数返回nil和错误信息.以这种方式我们可以通过与用户交互(比如:是否要打开另一个文件)合理的处理问题:

```
local file, msg
  repeat
    print "enter a file name:"
    local name = io.read()
    if not name then return end   -- no input
    file, msg = io.open(name, "r")
    if not file then print(msg) end
  until file
```

如果你想偷懒不想处理这些情况,又想代码安全的运行,可以简单的使用assert:

```
file = assert(io.open(name, "r"))
```

Lua中有一个习惯:如果io.open失败,assert将抛出错误.

```
file = assert(io.open("no-file", "r"))
    --> stdin:1: no-file: No such file or directory
```

注意:io.open返回的第二个结果(错误信息)作为assert的第二个参数.

8.4 异常和错误处理

很多应用中,不需要在Lua进行错误处理,一般有应用来完成.通常应用要求Lua运行一段chunk,如果发生异常,应用根据Lua返回的错误代码进行处理.在控制台模式下的Lua解释器如果遇到异常,打印出错误然后继续显示提示符等待下一个命令.

如果在Lua中需要处理错误,需要使用pcall函数封装你的代码.

假定你想运行一段Lua代码,这段代码运行过程中可以捕捉所有的异常和错误,第一步:将这段代码封装在一个函数内,

```
function foo ()
      ...
    if unexpected_condition then error() end
      ...
    print(a)   -- potential error: `a' may not be a table
      ...
  end
```

第二步:使用pcall调用这个函数

```
if pcall(foo) then
    -- no errors while running `foo'
    ...
  else
    -- `foo' raised an error: take appropriate actions
    ...
  end
```

当然也可以用匿名函数的方式调用pcall：

```
if pcall(function () ... end) then ...
  else ...
```

pcall在保护模式下调用他的第一个参数并运行,因此可以捕获所有的异常和错误.如果没有异常和错误,pcall返回true和调用返回的任何值;否则返回nil加错误信息.

错误信息不一定非要是一个字符串(下面的例子是一个table),传递给error的任何信息都会被pcall返回:

```
local status, err = pcall(function () error({code=121}) end)
  print(err.code) --> 121
```

这种机制提供了我们在Lua中处理异常和错误的所需要的全部内容.我们通过error抛出异常,然后通过pcall捕获他.

8.5 错误信息和回跟踪(Tracebacks)

虽然你可以使用任何类型的值作为错误信息,通常情况下,我们使用字符串来描述遇到的错误信息.如果遇到内部错误(比如对一个非table的值使用索引下表访问)Lua将自己产生错误信息,否则Lua使用传递给error函数的参数作为错误

信息.不管在什么情况下,Lua都尽可能清楚的描述发生的错误.

```
local status, err = pcall(function () a = 'a'+1 end)
  print(err)
    --> stdin:1: attempt to perform arithmetic on a string value

  local status, err = pcall(function () error("my error") end)
  print(err)
    --> stdin:1: my error
```

例子中错误信息给出了文件名(stdin)加上行号.

函数error还可以有第二个参数,表示错误的运行级别.有了这个参数你就无法抵赖错误是别人的了,比如,加入你写了一个函数用来检查error是否被正确的调用:

```
function foo (str)
    if type(str) ~= "string" then
      error("string expected")
    end
    ...
  end
```

可能有人这样调用这个函数:

```
foo({x=1})
```

Lua会指出发生错误的是foo而不是error,实际的错误是调用error时产生的,为了纠正这个问题修改前面的代码让error报告错误发生在第二级(你自己的函数是第一级)如下:

```
function foo (str)
    if type(str) ~= "string" then
      error("string expected", 2)
    end
    ...
  end
```

当错误发生的时候我们常常需要更多的错误发生相关的信息,而不单单是错误发生的位置.至少期望有一个完整的显示导致错误发生的调用栈的tracebacks,当pcall返回错误信息的时候他已经释放了保存错误发生情况的栈的信息.因此,如果我们想得到tracebacks我们必须在pcall返回以前获取.Lua提供了xpcall

来实现这个功能,xpcall接受两个参数:调用函数和错误处理函数.当错误发生时,Lua会在栈释放以前调用错误处理函数,因此可以使用debug库收集错误相关的信息.有两个常用的debug处理函数:debug.debug和debug.traceback,前者给出Lua的提示符,你可以自己动手察看错误发生时的情况;后者通过traceback创建更多的错误信息,后者是控制台解释器用来构建错误信息的函数.你可以在任何时候调用debug.traceback获取当前运行的traceback

信息:

```
print(debug.traceback())
```

9.Coroutines

协同程序与多线程情况下的线程比较类似:有自己的堆栈,自己的局部变量,有自己的指令指针,但是和其他协同程序共享全局变量等很多信息.线程和协同程序的主要不同在于:在多处理器情况下,从概念上来讲多线程程序同时运行多个线程;而协同程序是通过协作来完成,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.

协同是非常强大的功能,但是用起来也很复杂.如果你第一次阅读本章时不理解本章中的例子请不要担心,你可以继续阅读本书的其他部分然后再回过头来阅读本章.

9.1 协同的基础

Lua通过table提供了所有的协同函数,create函数创建一个新的协同程序,create只有一个参数:协同程序将要运行的代码封装而成的函数,返回值为thread类型的值表示创建了一个新的协同程序.通常情况下,create的参数是一个匿名函数:

```
co = coroutine.create(function ()
        print("hi")
      end)

  print(co)   --> thread: 0x8071d98
```

协同有三个状态:挂起态,运行态,停止态.当我们创建一个协同程序时他开始的状态为挂起态,也就是说我们创建协同程序的时候不会自动运行,可以使用status函数检查协同的状态:

```
print(coroutine.status(co))   --> suspended
```

函数coroutine.resume可以使程序由挂起状态变为运行态:

```
coroutine.resume(co)   --> hi
```

这个例子中,协同体仅仅打印出"hi"之后便进入终止状态

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

当目前为止,协同看起来只是一种复杂的调用函数的方式,真正的强大之处体现在yield函数,它可以将正在运行的代码挂起,看一个例子:

```
co = coroutine.create(function ()
      for i=1,10 do
        print("co", i)
        coroutine.yield()
      end
      end)
```

现在重新执行这个协同程序,程序将在第一个yield处被挂起:

```
coroutine.resume(co)   --> co   1
print(coroutine.status(co))   --> suspended
```

从协同的观点看:使用函数yield可以使程序挂起,当我们激活被挂起的程序时,yield返回并继续程序的执行直到再次遇到yield或者程序结束.

```
coroutine.resume(co)   --> co   2
  coroutine.resume(co)   --> co   3
  ...
  coroutine.resume(co)   --> co   10
  coroutine.resume(co)   -- prints nothing
```

上面最后一次调用的时候,协同体已经结束,因此协同程序处于终止状态.如果我们仍然企图激活他,resume将返回false和错误信息.

```
print(coroutine.resume(co))
  --> false   cannot resume dead coroutine
```

注意:resume运行在保护模式下,因此如果协同内部存在错误Lua并不会抛出错误而是将错误返回给resume函数.

Lua中一对resume-yield可以相互交换数据.

下面第一个例子resume,没有相应的yield,resume把额外的参数传递给协同的主程序.

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

第二个例子,resume返回除了true以外的其他部分将作为参数传递给相应的yield

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

对称性,yield返回的额外的参数也将会传递给resume.

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

最后一个例子,当协同代码结束时主函数返回的值都会传给相应的resume:

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

我们很少在同一个协同程序中使用这几种特性,但每一种都有其用处.

现在已经了解了一些协同的内容,在我们继续学习以前,先要澄清两个概念:Lua提供的这种协同我们称为不对称的协同,就是说挂起一个正在执行的协同的函数与使一个被挂起的协同再次执行的函数是不同的,有些语言提供对称的协同,这种情况下,由执行到挂起之间状态转换的函数是相同的.

有人称不对称的协同为半协同,另一些人使用同样的术语表示真正的协同,严格意义上的协同不论在什么地方只要它不是在其他的辅助代码内部的时候都可以并且只能使执行挂起,不论什么时候在其控制栈内都不会有不可决定的调用.(However,

```
other people use the same term semi-coroutine to denote a restricted
```

implementation of coroutines, where a coroutine can only suspend its execution when it is not inside any auxiliary function, that is, when it has no pending calls in its control stack.).只有半协同程序的主体中才可以yield,python中的产生器(generator)就是这种类型的半协同的例子.

与对称的协同和不对称协同的区别不同的是,协同与产生器的区别更大.产生器相对比较简单,他不能完成真正的协同所能完成的一些任务.我们熟练使用不对称的协同之后,可以利用不对称的协同实现比较优越的对称协同.

9.2 管道和过滤器

协同最有代表性的作用是用来描述生产者-消费者问题.我们假定有一个函数在不断的生产值(比如从文件中读取),另一个函数不断的消费这些值(比如写到另一文件中),这两个函数如下:

```
function producer ()
    while true do
      local x = io.read()   -- produce new value
      send(x)           -- send to consumer
    end
  end

  function consumer ()
    while true do
      local x = receive()     -- receive from producer
      io.write(x, "
")       -- consume new value
    end
  end
```

(例子中生产者和消费者都在不停的循环,修改一下使得没有数据的时候他们停下来并不困难),问题在于如何使得receive和send协同工作.只是一个典型的谁拥有住循环的情况,生产者和消费者都处在活动状态,都有自己的主循环,都认为另一方是可调用的服务.对于这种特殊的情况,可以改变一个函数的结构解除循环,使其作为被动的接受.然而这种改变在某些特定的实际情况下可能并不简单.

协同为解决这种问题提供了理想的方法,因为调用者与被调用者之间的resume-yield关系会不断颠倒.当一个协同调用yield时并不会进入一个新的函数,取而代之的是返回一个未决的resume的调用.相似的,调用resume时也不会开始一个新的函数而是返回yield的调用.这种性质正是我们所需要的,与使得send-receive协同工作的方式是一致的.receive唤醒生产者生产新值,send把产生的值送给消费者消费.

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

  function send (x)
    coroutine.yield(x)
  end

producer = coroutine.create(
    function ()
      while true do
      local x = io.read()   -- produce new value
      send(x)
      end
    end)
```

这种设计下,开始时调用消费者,当消费者需要值时他唤起生产者生产值,生产者生产值后停止直到消费者再次请求.我们称这种设计为消费者驱动的设计.

我们可以使用过滤器扩展这个涉及,过滤器指在生产者与消费者之间,可以对数据进行某些转换处理.过滤器在同一时间既是生产者又是消费者,他请求生产者生产值并且转换格式后传给消费者,我们修改上面的代码加入过滤器(每一行前面加上行号).完整的代码如下:

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

```
  function send (x)
    coroutine.yield(x)
  end

  function producer ()
    return coroutine.create(function ()
      while true do
      local x = io.read()   -- produce new value
      send(x)
      end
    end)
  end

  function filter (prod)
    return coroutine.create(function ()
      local line = 1
      while true do
      local x = receive(prod)   -- get new value
      x = string.format("%5d %s", line, x)
      send(x)     -- send it to consumer
      line = line + 1
      end
    end)
  end

  function consumer (prod)
    while true do
      local x = receive(prod)   -- get new value
      io.write(x, "
")       -- consume new value
    end
  end
```

可以调用:

```
p = producer()
  f = filter(p)
  consumer(f)
```

或者:

```
consumer(filter(producer()))
```

看完上面这个例子你可能很自然的想到UNIX的管道,协同是一种非抢占式的多线程.管道的方式下,每一个任务在独立的进程中运行,而协同方式下,每个任务运行在独立的协同代码中.管道在读(consumer)与写(producer)之间提供了一个缓冲,因此两者相关的的速度没有什么限制,在上下文管道中这是非常重要的,因为在进程间的切换代价是很高的.协同模式下,任务间的切换代价较小,与函数调用相当,因此读写可以很好的协同处理.

9.3 用作迭代器的协同

我们可以将循环的迭代器看作生产者-消费者模式的特殊的例子.迭代函数产生值给循环体消费.所以可以使用协同来实现迭代器.协同的一个关键特征是它可以不断颠倒调用者与被调用者之间的关系,这样我们毫无顾虑的使用它实现一个迭代器,而不用保存迭代函数返回的状态.

我们来完成一个打印一个数组元素的所有的排列来阐明这种应用.直接写这样一个迭代函数来完成这个任务并不容易,但是写一个生成所有排列的递归函数并不难.思路是这样的:将数组中的每一个元素放到最后,依次递归生成所有剩余元素的排列.代码如下:

```
function permgen (a, n)
    if n == 0 then
    printResult(a)
    else
    for i=1,n do

      -- put i-th element as the last one
      a[n], a = a, a[n]
```

```
      -- generate all permutations of the other elements
      permgen(a, n - 1)

      -- restore i-th element
      a[n], a = a, a[n]

    end
    end
  end

function printResult (a)
    for i,v in ipairs(a) do
    io.write(v, " ")
    end
    io.write("
")
  end

  permgen ({1,2,3,4}, 4)
```

有了上面的生成器后,下面我们将这个例子修改一下使其转换成一个迭代函数:

1.第一步printResult 改为 yield

```
function permgen (a, n)
    if n == 0 then
    coroutine.yield(a)
    else
    ...
```

2.第二步,我们定义一个迭代工厂,修改生成器在生成器内创建迭代函数,并使生成器运行在一个协同程序内.迭代函数负责请求协同产生下一个可能的排列.

```
function perm (a)
    local n = table.getn(a)
    local co = coroutine.create(function () permgen(a, n) end)
    return function ()   -- iterator
    local code, res = coroutine.resume(co)
    return res
    end
  end
```

这样我们就可以使用for循环来打印出一个数组的所有排列情况了:

```
for p in perm{"a", "b", "c"} do
    printResult(p)
  end
    --> b c a
    --> c b a
    --> c a b
    --> a c b
    --> b a c
    --> a b c
```

perm函数使用了Lua中常用的模式:将一个对协同的resume的调用封装在一个函数内部,这种方式在Lua非常常见,所以Lua专门为此专门提供了一个函数coroutine.wrap.与create相同的是,wrap创建一个协同程序;不同的是wrap不返回协同本身,而是返回一个函数,当这个函数被调用时将resume协同.wrap中resume协同的时候不会返回错误代码作为第一个返回结果,一旦有错误发生,将抛出错误.我们可以使用wrap重写perm:

```
function perm (a)
    local n = table.getn(a)
    return coroutine.wrap(function () permgen(a, n) end)
  end
```

一般情况下,coroutine.wrap比coroutine.create使用起来简单直观,前者更确切的提供了我们所需要的:一个可以resume协同的函数,然而缺少灵活性,没有办法知道wrap所创建的协同的状态,也没有办法检查错误的发生.

9.4 非抢占式多线程

如前面所见,Lua中的协同是一协作的多线程,每一个协同等同于一个线程,yield-resume可以实现在线程中切换.

然而与真正的多线程不同的是,协同是非抢占式的.当一个协同正在运行时,不能在外部终止他.只能通过显示的调用yield挂起他的执行.对于某些应用来说这个不存在问题,但有些应用对此是不能忍受的.不存在抢占式调用的程序是容易编写的.不需要考虑同步带来的bugs,因为程序中的所有线程间的同步都是显示的.你仅仅需要在协同代码超出临界区时调用yield即可.

对非抢占式多线程来说,不管什么时候只要有一个线程调用一个阻塞操作(blocking operation),整个程序在阻塞操作完成之前都将停止.对大部分应用程序而言,只是无法忍受的,这使得很多程序员离协同而去.下面我们将看到这个问题可以被有趣的解决.

看一个多线程的例子:我们想通过http协议从远程主机上下在一些文件.我们使用Diego Nehab开发的LuaSocket库来完成.我们先看下在一个文件的实现,大概步骤是打开一个到远程主机的连接,发送下载文件的请求,开始下载文件,下载完毕后关闭连接.

第一,加载LuaSocket库

```
require "luasocket"
```

第二,定义远程主机和需要下载的文件名

```
host = "www.w3.org"
  file = "/TR/REC-html32.html"
```

第三,打开一个TCP连接到远程主机的80端口(http服务的标准端口)

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

上面这句返回一个连接对象,我们可以使用这个连接对象请求发送文件

```
c:send("GET " .. file .. " HTTP/1.0

")
```

receive函数返回他送接收到的数据加上一个表示操作状态的字符串.当主机断开连接时,我们退出循环.

第四,关闭连接

```
c:close()
```

现在我们知道了如何下载一个文件,下面我们来看看如何下载多个文件.一种方法是我们在一个时刻只下载一个文件,这种顺序下载的方式必须等前一个文件下载完成后一个文件才能开始下载.实际上是,当我们发送一个请求之后有很多时间是在等待数据的到达,也就是说大部分时间浪费在调用receive上.如果同时可以下载多个文件,效率将会有很大提高.当一个连接没有数据到达时,可以从另一个连接读取数据.很显然,协同为这种同时下载提供了很方便的支持,我们为每一个下载任务创建一个线程,当一个线程没有数据到达时,他将控制权交给一个分配器,由分配器唤起另外的线程读取数据.

使用协同机制重写上面的代码,在一个函数内:

```
function download (host, file)
    local c = assert(socket.connect(host, 80))
    local count = 0   -- counts number of bytes read
    c:send("GET " .. file .. " HTTP/1.0

")
    while true do
    local s, status = receive(c)
    count = count + string.len(s)
    if status == "closed" then break end
    end
    c:close()
    print(file, count)
  end
```

由于我们不关心文件的内容,上面的代码只是计算文件的大小而不是将文件内容输出.(当有多个线程下载多个文件时,输出会混杂在一起),在新的函数代码中,我们使用receive从远程连接接收数据,在顺序接收数据的方式下代码如下:

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

在同步接受数据的方式下,函数接收数据时不能被阻塞,而是在没有数据可取时yield,代码如下:

```
function receive (connection)
    connection:timeout(0)   -- do not block
    local s, status = connection:receive(2^10)
    if status == "timeout" then
    coroutine.yield(connection)
    end
```

```
    return s, status
  end
```

调用函数timeout(0)使得对连接的任何操作都不会阻塞.当操作返回的状态为timeout时意味着操作未完成就返回了.在这种情况下,线程yield.非false的数值作为yield的参数告诉分配器线程仍在执行它的任务.(后面我们将看到分配器需要timeout连接的情况),注意:即使在timeout模式下,连接依然返回他接受到直到timeout为止,因此receive会一直返回s给她的调用者.

下面的函数保证每一个下载运行在自己独立的线程内:

```
threads = {}   -- list of all live threads
  function get (host, file)
    -- create coroutine
    local co = coroutine.create(function ()
    download(host, file)
    end)
    -- insert it in the list
    table.insert(threads, co)
  end
```

代码中table中为分配器保存了所有活动的线程.

分配器代码是很简单的,它是一个循环,逐个调用每一个线程.并且从线程列表中移除已经完成任务的线程.当没有线程可以运行时退出循环.

```
function dispatcher ()
    while true do
    local n = table.getn(threads)
    if n == 0 then break end   -- no more threads to run
    for i=1,n do
      local status, res = coroutine.resume(threads)
      if not res then   -- thread finished its task?
        table.remove(threads, i)
        break
      end
    end
    end
  end
```

最后,在主程序中创建需要的线程调用分配器,例如:从W3C站点上下载4个文件：

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

  get(host, "/TR/html401/html40.txt")
  get(host,"/TR/2002/REC-xhtml1-20020801/xhtml1.pdf")
  get(host,"/TR/REC-html32.html")
  get(host,
    "/TR/2000/REC-DOM-Level-2-Core-20001113/DOM2-Core.txt")

  dispatcher()   -- main loop
```

使用协同方式下,我的机器花了6s下载完这几个文件;顺序方式下用了15s,大概2倍的时间.

尽管效率提高了,但距离理想的实现还相差甚远,当至少有一个线程有数据可读取的时候,这段代码可以很好的运行.否则,分配器将进入忙等待状态,从一个线程到另一个线程不停的循环判断是否有数据可获取.结果协同实现的代码比顺序读取将花费30倍的CPU时间.

为了避免这种情况出现,我们可以使用LuaSocket库中的select函数.当程序在一组socket中不断的循环等待状态改变时,它可以使程序被阻塞.我们只需要修改分配器,使用select函数修改后的代码如下:

```
function dispatcher ()
    while true do
    local n = table.getn(threads)
    if n == 0 then break end   -- no more threads to run
    local connections = {}
    for i=1,n do
      local status, res = coroutine.resume(threads)
      if not res then   -- thread finished its task?
        table.remove(threads, i)
        break
```

```
      else   -- timeout
        table.insert(connections, res)
      end
    end
    if table.getn(connections) == n then
      socket.select(connections)
    end
    end
  end
```

在内层的循环分配器收集连接表中timeout地连接,注意:receive将连接传递给yield,因此resume返回他们.当所有的连接都timeout分配器调用select等待任一连接状态的改变.最终的实现效率和上一个协同实现的方式相当,另外,他不会发生忙等待,比起顺序实现的方式消耗CPU的时间仅仅多一点点.

10.Complete Examples

我们看两个完整的例子来阐明Lua语言的使用.第一个例子来自于Lua网站,他展示了Lua作为数据描述语言的使用.第二个例子讲解了马尔可夫链算法的实现,这个算法在Kernighan

& Pike著作的Practice of

Programming书中也有描述.这两个完整的例子之后我们将结束Lua语言方面的介绍,后面将继续介绍table和面向对象的内容以及标准库,C-API等.

10.1 Lua作为数据描述语言使用

Lua网站保留一个包含世界各地使用Lua创建的工程的例子的数据库.在数据库中我们用一个构造器以自动归档的方式表示每一个工程入口点,代码如下:

```
entry{
    title = "Tecgraf",
    org = "Computer Graphics Technology Group, PUC-Rio",
    url = "http://www.tecgraf.puc-rio.br/",
    contact = "Waldemar Celes",
    description = [[
      TeCGraf is the result of a partnership between PUC-Rio,
      the Pontifical Catholic University of Rio de Janeiro,
      and <A HREF="http://www.petrobras.com.br/">PETROBRAS</A>,
      the Brazilian Oil Company.
      TeCGraf is Lua's birthplace,
      and the language has been used there since 1993.
      Currently, more than thirty programmers in TeCGraf use
      Lua regularly; they have written more than two hundred
      thousand lines of code, distributed among dozens of
      final products.]]
    }
```

有趣的是,工程入口的列表是存放在一个Lua文件中的,每个工程入口以table的形式作为参数去调用entry函数.我们的目的是写一个程序将这些数据以html格式展示出来.由于工程太多,我们首先列出工程的标题,然后显示每个工程的明细.结果如下:

```
<HTML>
  <HEAD><TITLE>Projects using Lua</TITLE></HEAD>
  <BODY BGCOLOR="#FFFFFF">
  Here are brief descriptions of some projects around the
  world that use <A HREF="home.html">Lua</A>.
  <BR>
  <UL>
  <LI><A HREF="#1">TeCGraf</A>
  <LI> ...
  </UL>

  <H3>
  <A NAME="1" HREF="http://www.tecgraf.puc-rio.br/">TeCGraf</A>
  <BR>
  <SMALL><EM>Computer Graphics Technology Group,
```

```
          PUC-Rio</EM></SMALL>
  </H3>

      TeCGraf is the result of a partnership between
      ...
      distributed among dozens of final products.<P>
  Contact: Waldemar Celes

  <A NAME="2"></A><HR>
  ...

  </BODY></HTML>
```

为了读取数据,我们需要做的是正确的定义函数entry,然后使用dofile直接运行数据文件即可(db.lua).注意,我们需要遍历入口列表两次,第一次为了获取标题,第二次为了获取每个工程的表述.一种方法是:使用相同的entry函数运行数据文件一次将所有的入口放在一个数组内;另一种方法:使用不同的entry函数运行数据文件两次.因为Lua编译文件是很快的,这里我们选用第二种方法.

首先,我们定义一个辅助函数用来格式化文本的输出(参见5.2函数部分内容)

```
function fwrite (fmt, ...)
    return io.write(string.format(fmt, unpack(arg)))
  end
```

第二,我们定义一个BEGIN函数用来写html页面的头部

```
function BEGIN()
    io.write([[
      <HTML>
      <HEAD><TITLE>Projects using Lua</TITLE></HEAD>
      <BODY BGCOLOR="#FFFFFF">
      Here are brief descriptions of some projects around the
      world that use <A HREF="home.html">Lua</A>.
      <BR>
    ]])
  end
```

第三,定义entry函数

a.第一个entry函数,将每个工程一列表方式写出,entry的参数o是描述工程的table.

```
function entry0 (o)
    N=N + 1
    local title = o.title or '(no title)'
    fwrite('<LI><A HREF="#%d">%s</A>
', N, title)
  end
```

如果o.title为 nil表明table中的域title没有提供,我们用固定的"no title"替换.

b.第二个entry函数,写出工程所有的相关信息,稍微有些复杂,因为所有项都是可选的.

```
function entry1 (o)
    N=N + 1
    local title = o.title or o.org or 'org'
    fwrite('<HR>
<H3>
')
    local href = ''

    if o.url then
      href = string.format(' HREF="%s"', o.url)
    end
    fwrite('<A NAME="%d"%s>%s</A>
', N, href, title)

    if o.title and o.org then
      fwrite('<BR>
<SMALL><EM>%s</EM></SMALL>', o.org)
    end
```

```
    fwrite('
</H3>
')

    if o.description then
      fwrite('%s', string.gsub(o.description,
                       '


*', '<P>
'))
      fwrite('<P>
')
    end

    if o.email then
      fwrite('Contact: <A HREF="mailto:%s">%s</A>
',
           o.email, o.contact or o.email)
    elseif o.contact then
      fwrite('Contact: %s
', o.contact)
    end
  end
```

由于html中使用双引号,为了避免冲突我们这里使用单引号表示串.

第四,定义END函数,写html的尾部

```
function END()
    fwrite('</BODY></HTML>
')
  end
```

在主程序中,我们首先使用第一个entry运行数据文件输出工程名称的列表,然后再以第二个entry运行数据文件输出工程相关信息.

```
  BEGIN()

  N = 0
  entry = entry0
  fwrite('<UL>
')
  dofile('db.lua')
  fwrite('</UL>
')

  N = 0
  entry = entry1
  dofile('db.lua')

  END()
```

10.2 马尔可夫链算法

我们第二个例子是马尔可夫链算法的实现,我们的程序以前n(n=2)个单词串为基础随机产生一个文本串.

程序的第一部分读出原文,并且对没两个单词的前缀建立一个表,这个表给出了具有那些前缀的单词的一个顺序.建表完成后,这个程序利用这张表生成一个随机的文本.在此文本中,每个单词都跟随着它的的前两个单词,这两个单词在文本中有相同的概率.这样,我们就产生了一个非常随机,但并不完全随机的文本.例如,当应用.....这个程序的输出结果会出现“构造器也可以通过表构造器,那么一下几行的插入语对于整个文件来说，不是来存储每个功能的内容,而是来展示它的结构.”如果你想在队列里找到最大元素并返回最大值,接着显示提示和运行代码.下面的单词是保留单词,不能用在度和弧度之间转换.

我们编写一个函数用来将两个单词中间加上空个连接起来:

```
function prefix (w1, w2)
    return w1 .. ' ' .. w2
  end
```

我们用NOWORD(即
)表示文件的结尾并且初始化前缀单词,例如,下面的文本:

```
the more we try the more we do
```

初始化构造的表为:

```
{ ["

"] = {"the"},
    ["
the"] = {"more"},
    ["the more"] = {"we", "we"},
    ["more we"] = {"try", "do"},
    ["we try"] = {"the"},
    ["try the"] = {"more"},
    ["we do"] = {"
"},
  }
```

我们使用全局变量statetab来保存这个表,下面我们完成一个插入函数用来在这个statetab中插入新的单词.

```
function insert (index, value)
    if not statetab[index] then
      statetab[index] = {value}
    else
      table.insert(statetab[index], value)
    end
  end
```

这个函数中首先检查指定的前缀是否存在,如果不存在则创建一个新的并赋上新值.如果已经存在则调用table.insert将新值插入到列表尾部.

我们使用两个变量w1和w2来保存最后读入的两个单词的值,对于每一个前缀,我们保存紧跟其后的单词的列表.例如上面例子中初始化构造的表.

初始化表之后,下面来看看如何生成一个MAXGEN(=1000)个单词的文本.首先,重新初始化w1和w2,然后对于每一个前缀,在其next单词的列表中随机选择一个,打印此单词并更新w1和w2,完整的代码如下:

```
-- Markov Chain Program in Lua

  function allwords ()
    local line = io.read()   -- current line
    local pos = 1         -- current position in the line
    return function ()     -- iterator function
      while line do       -- repeat while there are lines
      local s, e = string.find(line, "%w+", pos)
      if s then     -- found a word?
        pos = e + 1 -- update next position
        return string.sub(line, s, e)   -- return the word
      else
        line = io.read()   -- word not found; try next line
        pos = 1         -- restart from first position
      end
      end
      return nil         -- no more lines: end of traversal
    end
  end

  function prefix (w1, w2)
    return w1 .. ' ' .. w2
  end

  local statetab

  function insert (index, value)
    if not statetab[index] then
      statetab[index] = {n=0}
```

```
    end
    table.insert(statetab[index], value)
  end

  local N = 2
  local MAXGEN = 10000
  local NOWORD = "
"

  -- build table
  statetab = {}
  local w1, w2 = NOWORD, NOWORD
  for w in allwords() do
    insert(prefix(w1, w2), w)
    w1 = w2; w2 = w;
  end
  insert(prefix(w1, w2), NOWORD)

  -- generate text
  w1 = NOWORD; w2 = NOWORD   -- reinitialize
  for i=1,MAXGEN do
    local list = statetab[prefix(w1, w2)]
    -- choose a random item from list
    local r = math.random(table.getn(list))
    local nextword = list[r]
    if nextword == NOWORD then return end
    io.write(nextword, " ")
    w1 = w2; w2 = nextword
  end
```