**1.1 chunks（块）**

Chunk是一系列语句，lua执行的每一块语句，比如一个文件或者交互模式下的每一行都是一个chunk

每个语句结尾的分号是可选的，但如果是同一行有多个语句最好用分号分开

一个chunk可以是一个语句，也可以是一系列语句的组合，还可以是函数，chunk可以很大，在lua中几个MByte的Chunk是很常见的

**1.2 全局变量**

全局变量不需要声明，给一个变量赋值后即创建了这个全局变量，访问一个没有初始化的全局变量也不会出错，只不过得到的结果是：nil。如果想删除一个全局变量，值需要将变量赋值为nil，这样的变量就好像从没被使用过一样，换句话说，当且仅当一个变量不等于nil时，这个变量存在。





**1.3 词法约定**

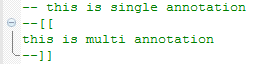
保留字：



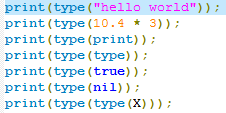
注意：lua是大小写敏感的

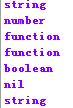
单行注释：--

多行注释：--[[ --]]

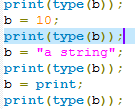


Lua是动态类型语言，变量不要类型定义。Lua中有8个基本类型分别为：nil、boolean、number、string、userdata、function、thread和table，函数type可以测试给定变量或者值的类型：





变量没有预定义的类型，每个变量都可能包含任一种类型的值





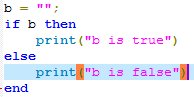
一般情况下同一变量代表不同类型的值会造成混乱，最好不要使用，但是特殊情况下可以带来便利，比如nil

**2.1 Nil**

Lua中的特殊类型，它只有一个值：nil，一个全局变量没有被赋值以前默认值为nil；给全局变量赋nil可以删除该变量

**2.2 Booleans**

两个取值false和true。但是要注意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，其他的都为真，所有lua认为0和空串都是真（已验证）





**2.3 Numbers**

表示实数，lua中没有证书，一般有个错误的看法cpu运算浮点数比整数慢。是是不是如此，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可以处理任何长证书不用担心误差。你也可以在编译lua的时候使用长整型或者单精度浮点型代替numbers，在一些平台硬件不支持浮点数的情况下这个特性是非常有用的，

**2.3 Strings**

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二进制数据在一个字符串中。Lua中的字符串是不可以修改的，我们可以创建一个新的变量存放所要的字符串



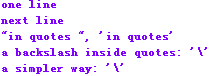


注意：如果没有找到“one”这个字符串，则b的值也是one string（已验证）

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可以高效的处理字符串，1M的string在lua中是很常见的，可以使用单引号或双引号表示字符串。

转义字符：



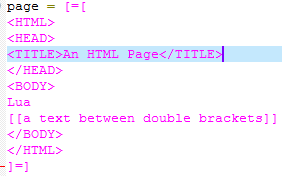


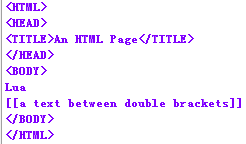
还可以在字符串中使用\ddd(ddd为三位十进制数字)方式表示字母

"alo\n123\""与'\97lo\10\04923"'是一样的

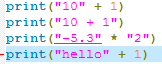
还可以用[[…]]表示字符串。这种形式的字符串可以包含多行，可以嵌套也不会解释转义序列，如果这一个字符是换行符会被自动忽略掉，这种形式的字符串用来包含一段代码是非常方便的。

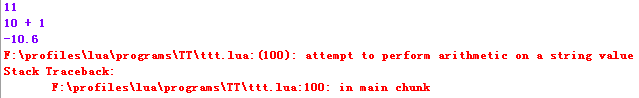
在这种情况下如果要进行嵌套，则必须改为[=[…]=]





运行时，lua会自动在string与numbers之间自动进行类型转换，当一个字符串使用段数操作符时，string就会被转成数字





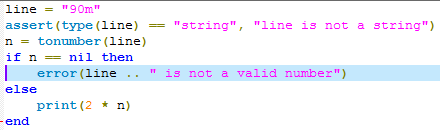
反过来，当lua期望一个string而碰到数字时，会将数字转成string

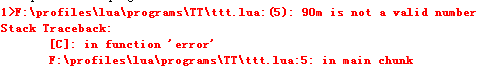




..在lua中是字符串连接符，当在一个数字后面写..时，必须加上空格以防止被解释错误

尽管字符串和数字可以自动转换，但两者是不同的，像10==’10’这样的饿比较永远都是错的。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将返回nil





如果改为line = “90”



反之，可以调用tostring将数字转换成字符串，这种转换一直有效





**2.4 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，此外，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、io库、os库、算术库、debug库

**2.5 UserData and Threads**

Userdata可以将c数据存放在lua变量中，userdata在lua中除了赋值和相等比较外没有预定义的操作

**3.1 算数运算符**

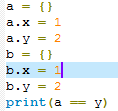
二元运算符：+ - \* / ^ （加减乘除幂）

一元运算符：-（负号）

**3.2 关系运算符**

< > >= <= == ~=

这些操作符返回结果为false或true；==与~=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





**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

一个很实用的技巧：

X = x or v 如果x为false或者nil则给x赋初始值v

等价于：

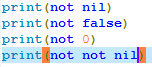
if not x then

x = v

end

注意：and的优先级比or高

not的结果一直返回false或者true





**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

**3.5 表的构造**

构造器是创建和初始化表的表达式，表示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





等价于



注意：第一个元素索引为1

还可以在初始化表的死后为元素提供一个索引





等价于：

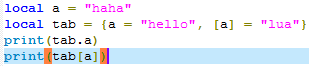


也等价于：



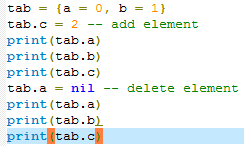
不管用何种方式创建table，我们都可以向表中添加或者删除任何类型的域，构造函数仅仅影响表的初始化

注意区分tab[“a”]和tab[a]这两种形式，第一种代表的索引是字符串a，可以使用等价的tab.a来访问；第二种代表的是使用a这个变量的值作为索引来访问



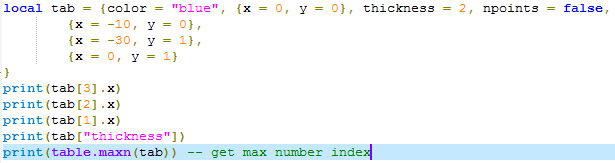


添加和删除元素



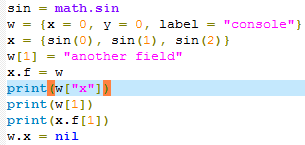


一个表可以混合各种类型的值，可以是波哦了按、字符串、表、函数等等



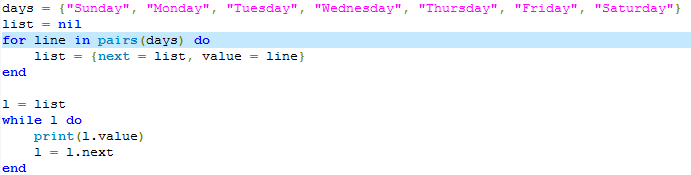


由上面的结果可以看出，上面如果是字符串索引的话就直接引用就可以了，如果用下标获取值的话都是要除掉字符串索引之后进行获取





每次调用构造函数，lua都会创建一个新的table，可以用table构造一个list





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。

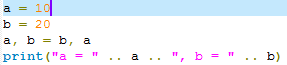
**4.1 赋值语句**

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一次赋给左边的变量





遇到赋值语句lua会先计算右边所有的值后在执行赋值操作，所有我们可以这样进行交换变量的值：

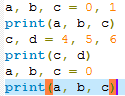




当变量个数和值的个数不一致时，lua会一直以变量个数为基础采取以下策略：

1）变量个数>值个数 按变量个数不足nil

2) 变量个数<值个数 多余的值会被忽略





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

多次赋值经常用来交换变量，或将函数调用返回给变量

a, b = f()

f()返回两个值，第一个赋为a，第二个赋给b

**4.2 局部变量与代码块（block）**

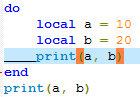
使用local创建一个局部变量，与全局变量不同，局部变量只在被声明的那个代码块内有效。代码块：只一个控制结构内，一个函数体，或者一个chunk（变量被声明的那个文件或者文本串）

应该尽量使用局部变量，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

我们可以给block划定一个明确的界限：do..end内的部分。当你想更好地控制局部变量的作用范围时，这是很有用的：





**4.3 控制结构语句**

（1）if语句

if conditions then

then-part

end

if conditions then

then-part

else

else-part

end

if conditions then

then-part

elseif conditions then

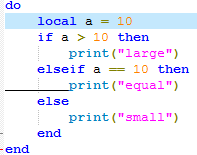
elseif-part

..

else

else-part

end



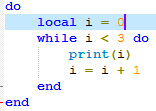


（2）while语句

While conditions do

Statements

end



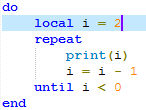


（3）repeat-until语句

repeat

statements；

until conditions





（4）for语句

for语句有两大类：

1）数值for循环

for var=exp1, exp2, exp3 do

Loop-part

end

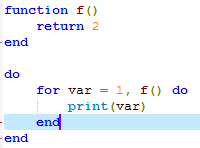
for将用exp3作为step从exp1（初始值）到exp2（终止值），执行loop-part。其中exp3可以省略，默认step=1

有几点需要注意：

**三个表达式只会被计算一次，并且是循环开始前**

**控制变量var是局部变量自动被声明，并且只在循环内有效**

**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，如果要退出使用break**



注意上面的f()只会在循环前被调用一次

2）泛型for循环

-- print all values of array ‘a’

for i, v in ipairs(a) do print(v) end

i是数组索引值，v是对应索引的数组元素值，ipairs是lua提供的一个迭代器函数，用来迭代数组

泛型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：

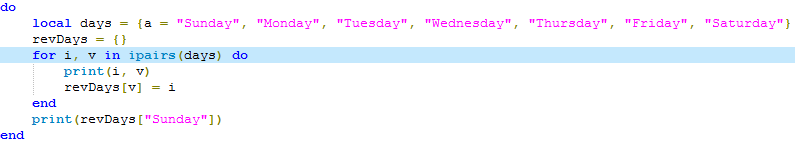
-- print all keys of table ‘t’

For k in paris(t) do print(k) end

泛型for和数值for有两点相同：

**控制变量是局部变量**

**不要修改控制变量的值**





**5.1 函数**

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





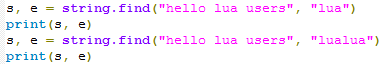
上述规则有一个例外，当函数只有一个参数并且这个参数是字符串或者表构造的时候，()是可选的



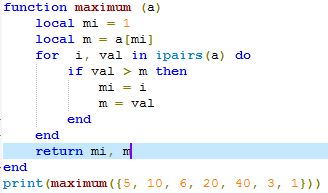


Lua也提供了面向对象方式调用函数的语法，比如o:foo(x)与o.foo(q,x)是等价的。Lua使用的函数可以是lua编写的也可以是其他语言编写，对lua程序猿来说用什么语言实现的函数使用起来都一样

Lua函数可以返回多个结果值，比如string.find，其返回匹配穿”开始和结束的下标”(如果不存在匹配串则返回nil)









Lua总是调整函数返回值的分数去使用调用环境，当作为一个语句调用函数时，所有返回值被忽略，假设有如下三个函数：



第一，当作为表达式调用函数时，有以下几种情况：

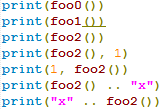
1 当调用作为表达式最后一个参数或者仅有一个参数时，根据变量个数函数尽可能多地返回多个值，不足补nil，超出舍去

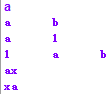
2 其他情况下，函数调用仅返回第一个值（如果没有返回值为nil）





第二，函数调用作为函数参数被调用时，和多值赋值是相同的



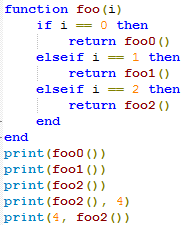


第三，函数调用在表构造函数中初始化时，和多值赋值时相同





另外，return f()这种类型的返回f()返回的所有值





一个return语句如果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被用来实现泛型调用机制，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：

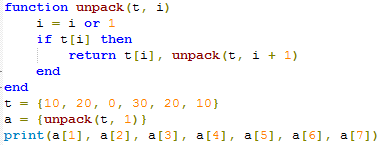
f(unpack(a))

unpack返回a所有的元素作为f()的参数





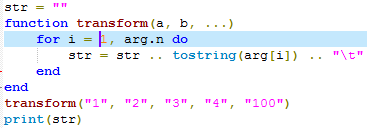
预定义的unpack函数是用C语言实现的，我们也可以用lua来完成





**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(…)，表示函数有可变参数。Lua将函数的参数放在一个叫做arg的表中，除了参数以外，arg表中海有一个域n表示参数的个数，lus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





如果我们只需要某个函数返回的某一个值，可以使用**虚变量（下划线）**



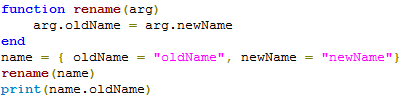


有时候需要将函数的可变参数传递给另外的函数调用，可以使用前面我们说过的unpack(arg)返回arg表所有的可变参数，lua提供了一个文本格式化的函数string.format（类似于c语言的sprintf函数）

**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。

Lua可以通过将所有参数放在一个表中，把表作为函数的唯一参数来实现上面的功能，因为lua语法支持函数调用时实参可以是表的构造



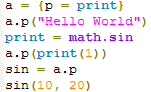


Lua中的函数是带有词法定界的第一类值。

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

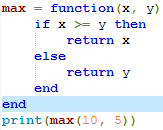
词法定界指：被嵌套的函数可以访问它外部函数中的变量。这一特性给lua提供了强大的编程能力

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，像持有其他类型值的变量一样



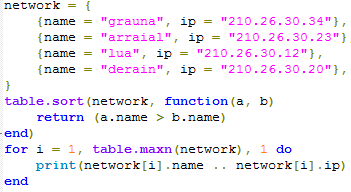


函数定义实际上是一个赋值语句，将类型为function的变量赋给一个变量，我们使用function(x) … end来定义一个函数和使用{}创建一个表一样





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。





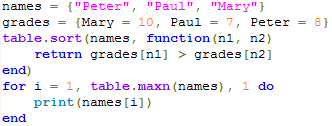
以其他函数作为参数的函数在lua中被称为高级函数，高级函数在lua并没有特权，只是lua把函数当做第一类函数处理的一个简单的结果。

**6.1 闭包**

当一个函数内部嵌套另一个函数定义时，内部的函数体可以访问外部的函数的局部变量，这种特征我们成为词法定界

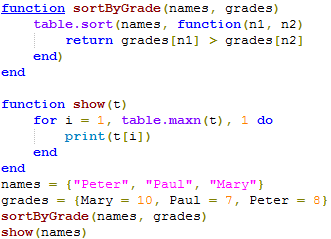
看看下面的例子：

假设有一个学生姓名和成绩的表单，现在想从高到低进行排序，可以这样做：



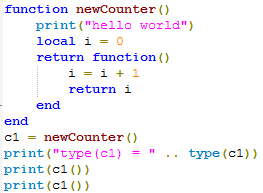


我们现在可以创建一个函数来实现此功能：



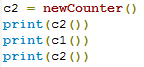


在上面的例子中，包含得sortByGrade函数内部的sort中的匿名函数可以访问sortByGrades的参数grades，在匿名函数内部grades不是全局变量也不是局部变量，我们称为外部的局部变量（external local variable）或者upvalue





匿名函数使用upvalue i保存它的计数，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。简单地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。





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

**6.2 非全局函数**

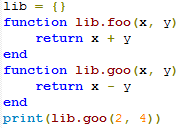
Lua中函数可以作为全局变量也可以作为局部变量，我们已经看到了一些例子：函数作为table的域（大部分lua标准库使用这种机制来实现的比如io.read、math.sin）。这种情况下，必须注意函数和表语法：

1）表和函数放在一起



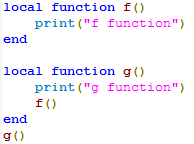
2）使用表构造函数

3）lua提供了另一种语法方式

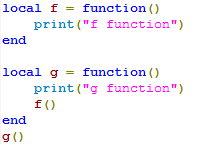


当我们将函数保存在一个局部变量内时，我们得到一个局部函数，也就是说局部函数像局部变量一样在一定范围内有效。这种定义在包中非常有用：因为lua把chunk当做函数处理，在chunk内可以声明局部函数（仅仅在chunk内可见）。词法定界保证了包内的其他函数可以调用此函数。下面是声明局部函数的两种方式：

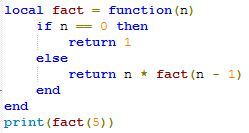
1）方式一



2）方式二

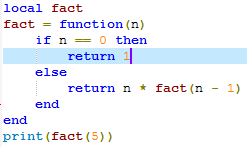


有一点需要注意的是在声明递归局部函数的方式：





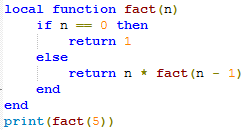
上面这种方式导致lua编译时遇到fact(n-1)并不知道它是局部函数，lua会查找是否有这样的全局函数fact，为了解决这个问题我们必须在定义函数以前先声明：





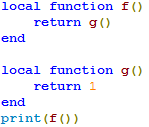
这样在fact内部fact(n-1)调用是一个局部函数调用，运行时fact就可以获取正确的值了。

但是可以这样

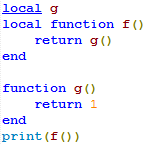




在定义非直接递归局部函数时要先声明然后定义才可以：









**6.3 正确的尾调用**

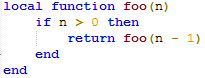
尾调用时一种类似在函数结尾的goto调用，当函数最后一个动作时调用另外一个函数时，我们称这种调用为尾调用，例如：



g的调用时尾调用

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，例如下面的调用不论n为何值不会导致栈溢出



需要注意的是：必须明确什么是尾递归

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：



上面这个例子总f在调用g后，不得不对齐g的返回值，所以不是尾调用，同样的下面几个例子也不是尾调用：



Lua中类似return g(…)这种格式的调用时尾调用，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：

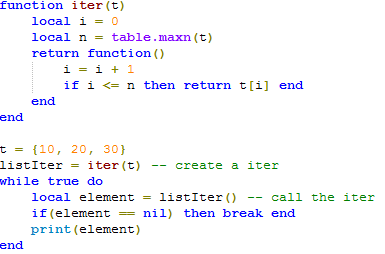


可以将尾递归理解成一种goto，并不是一个传统的函数调用。

**7.1 迭代器与闭包**

迭代器是一种支持指针类型的机构，它可以遍历几何的每一个元素，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，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他局部变量。所以一个典型的闭包的机构包含两个函数：一个是闭包自己，一个是工厂（创建闭包的函数）





在这个例子中iter是一个工厂，每次调用都会创建一个新的闭包（迭代器本身），闭包保存内部局部变量（t，i，n），因此每次调用它返回list中的下一个元素值，当list中没有值时，返回nil

我们设计的这个迭代器也很容易用于泛型for语句：





泛型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

**7.2 泛型for的语义**

前面我们看到迭代器有一个缺点：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有什么问题，但是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泛型for本身来保存迭代的状态

前面我们看到在循环过程中泛型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

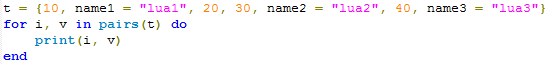
泛型for的语法如下：

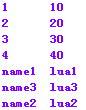
for <var-list> in <exp-list> do

<body>

end

<var-list>是一个或多个以逗号分割的变量名列表，<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





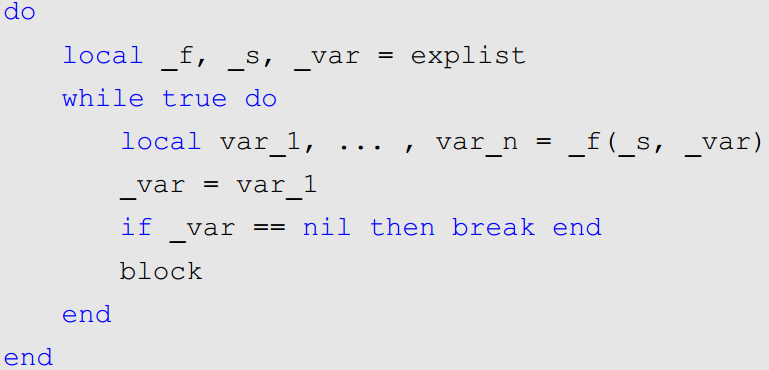
我们称变量列表中第一个变量为控制变量，其值为nil时循环结束，下面我们看看泛型for的执行过程：

1. 初始化，计算in后面表达式的值，表达式应该返回泛型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
2. 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它的值并传递给迭代函数）
3. 将迭代函数返回的值赋给变量列表
4. 如果返回的第一个值为nil循环结束，否则执行循环体
5. 回到第二步再次调用迭代函数

更精确的说：



等价于：



**7.3 无状态的迭代器**

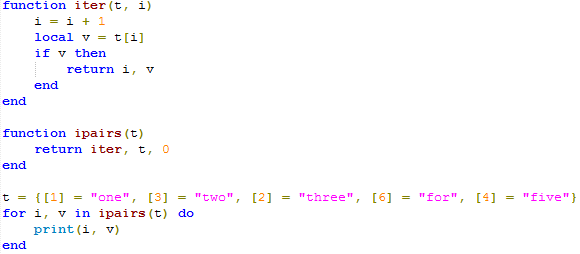
无状态的迭代器是指不保留任何状态的迭代器，因此在循环中我们可以利用无状态迭代器避免创建闭包花费的额外代价

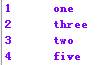
每一次迭代，迭代函数都是用两个变量（状态常量和控制变量）的值作为参数被调用，一个无状态的迭代器只利用这两个值可以获取下一个元素。这种无状态迭代器的典型的粒子是ipairs，它遍历数组的每一个元素





我们可以使用lua这么实现iparis：





当lua调用ipairs(t)开始循环时，它获取三个值：迭代函数iter，状态变量t和控制变量初始化0，然后lua调用iter(t,0) 返回1，t[1](除非t[1]==nul)，知道第一个非nil元素

我们可以使用如下方法实现pairs：（？？？）

Lua中的next函数：

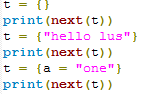
Allow a program to traverse all fields of a table. Its first argument is a table and its second argument is an index in this table. Next returns the next index of the table and its associated value. When called with nil as its second argument, next returns an initial index and its associated value. When called with the last index, or with nil in an empty table, next returns nil. If the second argument is absent, then it is interpreted ad nil. In particular, you can use next(t) to check whether a table is empty.

The behavior of next is undefined if during the traversal, you assign any value to a non-exitent field in the table. You may however modify existing fields. In particular, you may clear existing fields



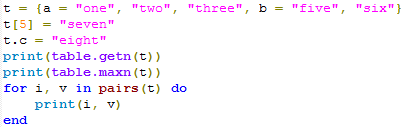


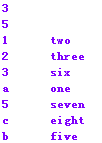
判断一个表是否为空

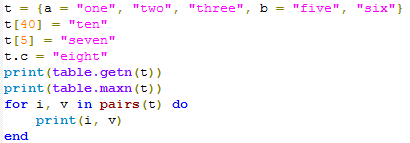


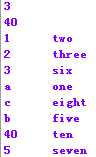
也就是说如果next(t)==nil则说明表为空，否则不为空

关于lua中的getn与maxn









**7.4 多状态的迭代器**

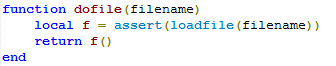
在很多情况下，迭代器需要保存多个状态信息而不是简单的状态常量和控制变量，最简单的方式就是使用闭包，还有一种方法就是将所有的状态信息封装到table内，将table作为迭代器的状态常量，因为这种情况下可以将所有的信息存放在table中，所以迭代函数通常不需要第二个函数

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，不需要创建对象话费的代价；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比要创建table小，另外lua处理闭包要比处理table速度快些，后面我们将会介绍另一种使用协同来创建迭代器的方式。

**8.1 编译、运行、调试**

虽然我们将lua当做解释型语言，但是lua会首先把代码预编译成中间代码然后再执行（很多解释型语言都是这么做的）。再解释型语言中存在编译阶段听起来不合适，然而，解释型语言的特性不在于他们是否被编译，而且天性是语言运行时的一部分，所以，执行中间代码的速度会更快，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。

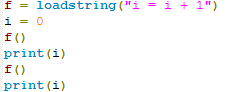
Dofile实际上是一个辅助函数，真正完成功能的函数是loadfile；与dofile不同的是loadfile编译代码成中间码并且返回编译后的chunk作为一个函数，而不执行代码；另外，loadfile不会抛出错误信息而是返回错误代码，我们可以这样定义dofile：



如果loadfile失败，assert会抛出异常

完成简单的功能dofile比较方便，它读入文件编译并执行。然而loadfile更加灵活，再发生错误的情况下，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，但可以多次运行，dofile却每次都要编译。

Loadstring与loadfile相似，只不过它不是从文件读入chunk，二是从一个串中读入，比如：





Lua将每个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：





Loadstring和loadfile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





另外，loadstring和loadfile都不会有边界效应产生，它们仅仅编译chunk成为自己内部实现的一个匿名函数。通常对它们的误解是它们定义了函数。Lua中函数定义是发生在运行时的复制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，如果要定义它必须运行chunk

f() -- define foo

foo(“ok”) --> ok

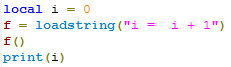


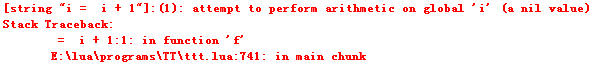


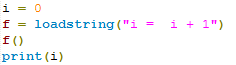
通常使用loadstring加载一个字符串没有什么意义，比如：

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。还有一个重要的区别：loading编译的时候不关心词法范围：









在上面的这个例子中，loadstring中的i使用的是全局变量i，loadstring总是在全局环境中编译它的串。

**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，每个模式指明一种由虚文件（require的参数）转成实文件名的方法。更确切的说，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。如果不存在继续同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:\windows\lili

/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，其它信息（目录分隔符，文件扩展名）在路径中定义。

比如在文件夹T中定义了tt.lua







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使用固定路径（典型的”?;?.lua”）

Require的另一个功能是避免重加载同一个文件两次，lua保留一张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，则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。

**8.3 错误**



Errir的参数值要抛出的错误信息。Lua提供了专门的内置函数assert来完成上面类似的功能：



Assert首先检查第一个参数是否返回错误，如果不返回错误assert简单地返回，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，它会首先计算两个参数然后再调用函数，所以一下代码：

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。

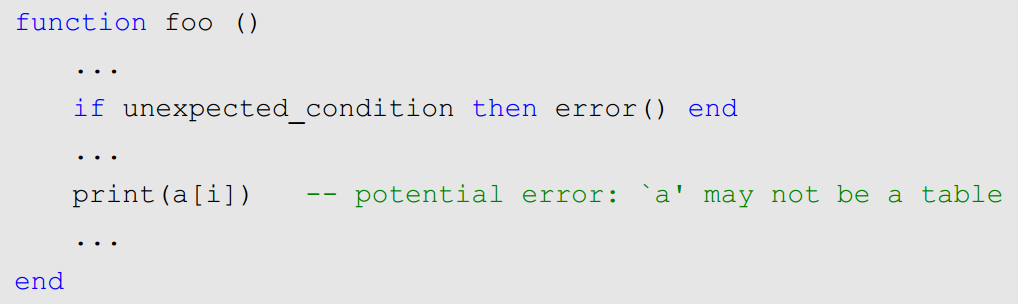
**8.4 异常和错误处理**

很多应用中，不需要在lua进行错误处理，一般由应用来完成。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。

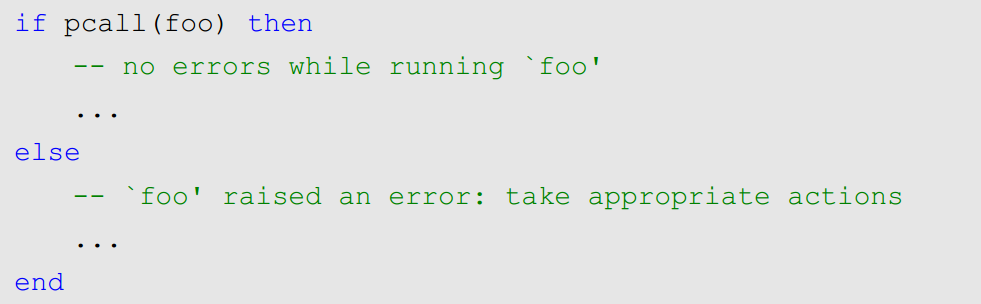
如果在lua中需要处理错误，需要使用pcall函数封装你的代码

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中



第二步：使用pcall调用这个函数



Pcall在保护模式下调用它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和调用返回的任何职；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的任何信息都会白pcall返回：





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捕获它。

**8.5 错误信息和回跟踪**

当错误发生时，我们常常需要更多的错误发生相关信息，而不仅仅是错误发生的位置。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它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会在栈释放前调用错误处理函数，因此可以使用debug库收集错误相关信息。有两个常用的debug处理函数：debug和debug.traceback，前者给出了lua的提示符，你可以自动动手查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。可以在任何时候调用debug.traceback获取当前运行的trackback信息：

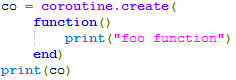




**9.1 协同基础**

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的，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要求挂起的时候才会被挂起。其实主要的区别就是：一个协同程序运行的时候，其他协同程序是无法获得执行的机会的，只有正在运行的协同程序主动挂起时，其他协同程序才有机会执行。而线程即使不主动休眠，也很有可能因为轮片时间到达而把执行机会让给其他线程

Lua通过coroutine提供了所有的协同函数，lua将它的协程函数都放进了coroutine这个表中。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：





协同有三个状态：挂起状态，运行状态，停止状态。当我们创建一个协同程序时，它开始的状态为挂起状态，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函数检查协同状态：





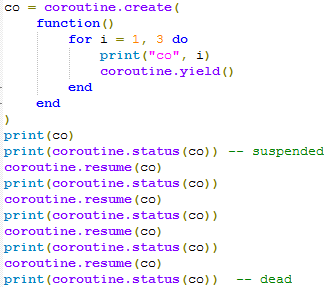
函数coroutine.resume可以使程序由挂起状态变为运行状态：

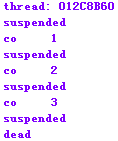




在上面这个例子中，协同体仅仅打印出“foo function”之后便进入终止状态

写成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：

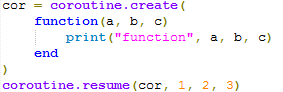




上面最后一次调用的时候，协同体已经结束，因此协同程序处于终止状态，

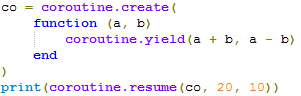
Lua中一对resume-yield可以相互交换数据：

（1）没有相应的yield，resume把额外的参数传递给协同的主程序



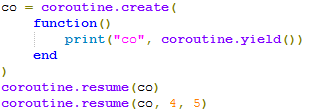


（2）resume返回除了true以外的其他部分将作为参数传递给响应的yield



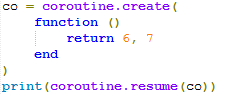


对称性，yield返回的额外的参数也将会传递给resume



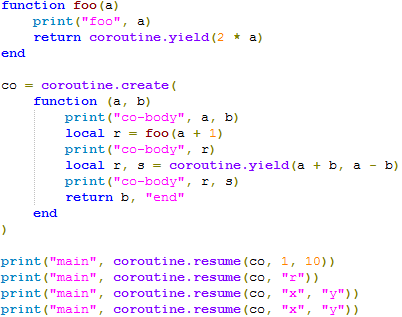


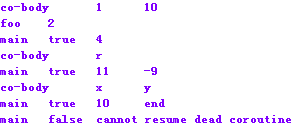
（3）当协同代码结束时主函数返回的值都会传给相应的resume





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，由执行到挂起之间的状态转换的函数是相同的。





**管道和过滤器**

协同最具有代表性的作用是用来描述生产者-消费者问题。

云风的一些建议：

[Lua](http://blog.codingnow.com/cloud/Lua5) 的 5.1 版本已经正式发布。现在，我们应该把全部讨论放在这个版本上。  
  
Hot! **应该尽量使用 local 变量而非 global 变量**。这是 [Lua](http://blog.codingnow.com/cloud/Lua5) 初学者最容易犯的错误。global 变量实际上是放在一张全局的 table 里的。global 变量实际上是利用一个 string (变量名作 key) 去访问这个 table 。虽然[[InterWiki]Lua5 的 table 效率很高](http://blog.codingnow.com/2005/10/lua_table.html) ，但是相对于 local 变量，依然有很大的效率损失。local 变量是直接通过 [Lua](http://blog.codingnow.com/cloud/Lua5) 的堆栈访问的。有些 global 变量的访问是不经意的，比如我们有双重循环操作一个迭代的 table：

for k1,v1 in pairs(tbl) do

for k2,v2 in pairs(v1) do

...

end

end

这里，pairs 其实是一个全局变量应用的函数。如果我们这样做：

do

local pairs=pairs

for k1,v1 in pairs(tbl) do

for k2,v2 in pairs(v1) do

...

end

end

end  
效率会稍微提高一些。如果是单层循环，这样做就没有意义。因为 for ... in 循环中的 pairs 这个函数只会被调用一次，而不是每次循环都去调。我们的原则其实是，被多次读取的 global 变量，都应该提取出来放到 local 变量中。  
  
Hot! **警惕临时变量** 字符串的连接操作，会产生新的对象。这是由 lua 本身的 string 管理机制导致的。lua 在 VM 内对相同的 string 永远只保留一份唯一 copy ，这样，所有字符串比较就可以简化为地址比较。这也是 lua 的 table 工作很快的原因之一。这种 string 管理的策略，跟 java 等一样，所以跟 java 一样，应该尽量避免在循环内不断的连接字符串，比如 a = a..x 这样。每次运行，都很可能会生成一份新的 copy 。  
  
同样，记住，每次构造一份 table 都会多一份 table 的 copy 。比如在 lua 里，把平面坐标封装成 { x, y } 用于参数传递，就需要考虑这个问题。每次你想构造一个坐标对象传递给一个函数，{ 10,20 }  这样明确的写出，都会构造一个新的 table 出来。要么，我们想办法考虑 table 的重用；要么，干脆用 x,y 两个参数传递坐标。  
  
同样需要注意的是以 function foo (...) 这种方式定义函数， ... 这种不定参数，每次调用的时候都会被定义出一个 table 存放不定数量的参数。  
  
这些临时构造的对象往往要到 gc 的时候才被回收，过于频繁的 gc 有时候正是效率瓶颈。  
  
Hot! **使用 closure 代替 table** 上面提到封装坐标的问题。诚然，我们可以用 { x=1,y=2 } 这样封装一个坐标。不过还有一个方法可供选择。它稍微轻量一点。

function point (x,y)

return function () return x,y end

end

*-- 使用范例*

p=point(1,2)

print(p())

*-- 输出 1 2*

如果你愿意，还可以做的复杂一点：

function point (x,y)

return function (idx)

if idx=="x" then return x

elseif idx=="y" then return y

else return x,y end

end

end

*-- 使用范例*

p=point(1,2)

print(p("x")) *-- 1*

print(p("y")) *-- 2*

x,y 实际被存放在 closure 里，每次调用 function point 都有一份独立的 closure。当然，function 的 code 只有一份。  
  
Hot! **设法减少从 C 向 Lua 传递字符串** 字符串常量在 Lua VM 内部工作的非常快，但是一个从 C 向 lua vm 通过 lua\_pushstring 之类的 api 传递进 VM 时，就需要掂量一下了。这至少包含一个再 hash 和匹配的过程。[[InterWiki]我的 Blog 上的一篇文章讨论了这个问题](http://blog.codingnow.com/2006/01/_lua.html)。  
  
Hot! **lua 中的继承** lua 中实现 OO ，虚表往往设置一个 metatable 并设置 \_\_index ，而继承则用 metatable 的 \_\_index 把虚表串起来。当类继承层次过多的时候，效率比较低，那么就可以用下面这个技巧。

function inherit(sub,super)

setmetatable(sub,

{ \_\_index=function(t,k)

local ret=super[k]

sub[k]=ret

return ret

end } )

end

Hot! **利用逻辑运算的短路效应** lua 编程中，and or 跟 C 一样是有短路效应的，不过他们的返回值并非 bool 类型，而是表达式中的左值或者右值。我们常常利用这个特性来简化代码。

function foo(arg)

arg=arg or "default"

...

end

利用 or 运算赋缺省值是最常用的技巧。上例中，如果 arg 为 nil ，arg 就会被赋值为 "default" 。但是这个技巧有个缺陷，当缺省值是 true 的时候会有点问题。

a=a or true *-- 错误的写法，当 a 明确写为 false 的时候，也会被改变成 true 。*

a= a ~= false *-- 正确的写法，当 a 为 nil 的时候，被赋值为 true ；而 false 则不变。*

另外，巧妙使用 and or 还可以实现类似 C 语言中的 ?: 三元操作：

function max(a,b)

return a>b and a or b

end

上面这个函数可以返回 a 和 b 中较大的一个，其逻辑类似 C 语言中的 return (a>b) ? a : b ;  
  
Hot! **模拟 pascal 中的 with**pascal(delphi) vb 中都有一个 with 的关键字，可以帮助程序员减少大量的输入，并能提高效率。lua 中我们可以利用以下方式模拟。

a={}

setfenv(function()

a=1

b=2

end,a)()

*-- 以上代码等价于*

a.a=1

a.b=2