注意：本文档翻译自书籍《Programming in Lua, Fourth Edition》，为快速入门，仅选取了原书最重要的一些章节并进行了删改。深入学习请购买电子工业出版社的《Lua程序设计 第4版》。

## 13.1 进入LUA的世界

### 13.1.1 Hello world!

依据传统，第一个Lua程序也是打印“Hello World”：

print("Hello World")

如果使用独立解释器，只需要执行lua + 源码的文件名即可（也可能是lua53 + 文件名）。比如如果文件是hello.lua，那么用以下命令执行：

% lua hello.lua

复杂一些的例子，下面的程序定义了一个函数，用来计算一个数的阶乘。然后让用户输入一个数字并打印阶乘的结果：

--defines a factorial

--定义一个函数function

function fact (n)

if n == 0 then return 1

else

return n \* fact(n - 1) end

end

print("enter a number:")

a = io.read("\*n") -- reads a number print(fact(a))

### 13.1.2 代码块

每一段可以单独执行的代码称为一个代码块（Chunk），例如一个lua文件或是交互模式下单独的一行代码。一个代码块是一串命令的序列（或语句）。

最简的代码块可以是一个单独的语句，比如“Hello World”的例子；或者也可以混合很多表达式和函数定义（函数定义实际上是一种赋值操作，后面会讲到），比如求阶乘的例子。代码块可以有任意的大小。由于Lua也可以用来作为数据描述语言，因此数兆字节的代码块也并不罕见。Lua解释器完全可以运行这么大的代码块。

可以不用把程序写在文件里，也可以以交互方式运行解释器。不加任何参数执行lua解释器，会有如下效果：

% lua

Lua 5.3 Copyright (C) 1994-2016 Lua.org, PUC-Rio

>

在这种模式下，每一条命令会立即执行。输入EOF符号（POSIX系统下为ctrl+D，Windows系统下为ctrl+Z）、或执行os.exit()命令，就可以退出解释器。

在Lua 5.3以后的版本中，如果输入一个表达式，会立即打印该表达式的值：

% lua

Lua 5.3 Copyright (C) 1994-2016 Lua.org, PUC-Rio

>math.pi / 4--> 0.78539816339745

>a = 15

> a^2 --> 225

> a + 2 --> 17

而在较老的版本中，可以使用前缀等号来实现同样的效果：

% lua5.2

Lua 5.2.3 Copyright (C) 1994-2013 Lua.org, PUC-Rio

> a = 15

> = a^2 --> 225

为保证兼容性，Lua5.3也支持前缀等号的用法。

在整体执行lua文件的情况下（整体运行代码块），要打印表达式的值，必须使用print函数并将参数包裹在括号中：

print(math.pi / 4)

a = 15

print(a^2)

print(a + 2)

在交互模式下，Lua通常认为每一句都是独立的代码块或语句。但是，如果能识别出一行代码并未结束，Lua就会试着等待更多输入直到代码块完整。因此在定义函数时就可以依次输入多行代码，直到函数输入完毕。但是，如果写较长的定义，还是将代码写在文件中更为合适。

可以使用-i选项来让Lua先执行一个代码块，然后再进入交互模式：

% lua -i prog

这样的命令会让Lua先运行prog文件，再进入交互模式。这种方式在调试和手动测试时非常有用。本章的末尾会介绍其他的解释器选项。

另一种执行代码块的方式是使用函数dofile，它会立即执行一个文件。例如有一个源文件lib1.lua，内容如下：

function norm (x, y)

return math.sqrt(x^2 + y^2)

end

function twice (x)

return 2.0 \* x

end

然后，在交互模式下输入：

> dofile("lib1.lua") -- 执行lib1.lua，即加载这个lib

> n = norm(3.4, 1.0)

> twice(n) --> 7.0880180586677

dofile函数在测试一段代码时也非常有用。打开两个窗口，一边编写代码一边在交互窗口中执行测试，是一种非常实用的技巧。

在源代码保存之后，只要在解释器中执行dofile(“prog.lua”)，就会重新执行新的代码，然后就可以重新调用函数、打印结果了。

### 13.1.3 一些字符的约定

Lua的标识符（或名称）可以是任意不以数字开头的字母、数字、下划线的组合，例如：

i j i10 \_ij

aSomewhatLongName \_INPUT

应当避免在命名时使用下划线加一个或多个大写字母的命名，例如\_VERSION。在lua中这种命名方式会用来表达特殊的含义。通常单独的\_（下划线）用来表示不需要保留的变量（占位用）。

以下是所有的保留字，不能用作标识符：

and break do else elseif

end false for function goto

if in local nil not

or repeat return then true

until while

Lua是大小写敏感的：例如and是一个保留字，而And和AND是两个不同的标识符。

注释以两个减号（--）开头，直到一行的结尾。

Lua还支持多行注释，符号为：两个减号加两个左方括号作为注释的开头，两个右方括号作为注释的结尾：

--[[A multi-line

long comment

]]

有一种常见的技巧，使用--[[和--]]配对：

--[[

print(10) -- no action (commented out)

--]]

这时，只要在开头加一个减号，就可以重新激活被注释掉的部分：

---[[

print(10) --> 10

--]]

因为一开始，--[[ 表示一个多行注释的开头，而结尾的--]]依然可以用来表示注释结尾。而加上减号以后，---[[就变成了一个普通的单行注释，结尾的--]]也变成了普通的单行注释，这样中间的print部分就不属于注释之内了。

两个中括号实际上用于表示“长字符串”。

Lua不需要使用分号区分多个表达式，但是如果必要仍然可以使用分号。行中间的分号并不影响Lua的语法，例如以下四个代码块是等价的：

a = 1

b = a \* 2

a = 1;

b = a \* 2;

a = 1; b = a \* 2

a = 1 b = a \* 2 -- 很丑，但是是合法的

我自己的规则是仅在一行中有多个语句时使用分号（很少有这种情况）。

### 13.1.4 全局变量

全局变量不需要特别声明，只需要直接使用即可。直接使用未初始化的变量并非错误，只会得到nil的结果：

> b --> nil

> b = 10

> b --> 10

将一个全局变量赋值为nil，结果就好像从未使用过它一样：

> b = nil

> b --> nil

对Lua来说，一个未初始化过的变量与一个被赋值为nil的变量没有任何不同。在赋值nil后，Lua会稍后释放变量可能占用的内存。

### 13.1.5 类型和值

Lua是一种动态类型的语言。使用时不需要定义变量类型，每个值都拥有自己的类型。

Lua有8种基本数据类型：nil，Boolean（布尔类型），number（数字），string（字符串），userdata，function（函数），thread（线程）和table（表）。使用内置函数type可以获得任意值的类型：

> type(nil) --> nil

> type(true) --> boolean

> type(10.4 \* 3) --> number

> type("Hello world") --> string

> type(io.stdin) --> userdata

> type(print) --> function

> type(type) --> function

> type({}) --> table

> type(type(X)) --> string

最后一行里，无论X是什么，结果最后都是string，因为type的返回值就是字符串。

userdata类型允许我们将C语言的数据保存在Lua变量中。Lua未对userdata类型的值定义任何操作，只包含基本的赋值和相等性判断。userdata类型可以用来表示用C语言创建的新的类型，例如标准I/O库使用它来表示打开的文件。之后讨论C API时再来讨论userdata。

变量没有预先定义的类型，变量可以保存任意类型的值：

> type(a) --> nil ('a'未初始化)

> a = 10

> type(a) --> number

> a = "a string!!"

> type(a) --> string

> a = nil

> type(a) --> nil

通常，当我们用一个变量来指代很多不同类型的数据时，总会让代码变得很乱。但是有时正确使用这种特性也很有用，例如，使用nil来表示异常的函数返回值，以区别于正常的函数返回值。

接下来讨论简单的类型nil与Boolean（布尔）。在之后的章节中，会详细讨论数字类型（第3章），字符串类型（第4章），表（第5章），函数（第6章）。第24章协程（Coroutines）中会讨论到thread（线程）类型。

### 13.1.6 空（nil）

Nil类型只有唯一的一个值——nil，它与其它类型均不同。Lua中使用nil来表示“不是一个值”，用来代表不存在有效的值。前面看到了，全局变量默认为nil直到它被赋予了新的值，且可以通过赋值nil来删除一个变量的值。

### 13.1.7 布尔类型

布尔类型（Boolean）具有两个值：false和true，与传统的布尔类型完全相同。但是Lua中布尔类型并不是条件判断的唯一依据：任何值都可以用来作为真假判断的条件。条件判断时，false和nil都代表“假”，其它任何值都代表“真”。特别的，空字符串和0也是真值。

本书中所有的“false”意思都是指任何为假的值，也就是布尔值false或nil。当明确指示是布尔的false时，会以黑体表示。同样的规则适用于true和“true”。

Lua支持常用的逻辑运算符：and，or和not。同上面说的条件判断相同，所有逻辑运算符将false和nil作为假，其它所有值都为真。and运算符的规则是：若第一个表达式为假，那么返回第一个表达式；否则返回第二个表达式。or运算符规则：若第一个表达式为真，那么返回第一个表达式；否则返回第二个表达式。

> 4 and 5 --> 5

> nil and 13 --> nil

> false and 13 --> false

> 0 or 5 --> 0

> false or "hi" --> "hi"

> nil or false --> false

and和or都采用“最短路径求值”规则，即只在必要时才会求第二个表达式的值。最短路径求值规则保证了如(i ~= 0 and a/i > b)这样的表达式不会导致运行时错误：这样的代码保证i为0时不会去计算a / i。

有一种常见技巧：x = x or v，等价于：

if not x then x = v end

上面的代码意思是，如果x未被赋值那么将其赋值为默认值v（假设不考虑x被赋值为false的情况）。

另一个常见技巧是 ((a and b) or c) 或更简化的 (a and b or c) ，由于and的优先级高于or所以可以这样写。这个写法等价于C语言中的 a ? b : c，假设b不为假。例如这样可以求x和y中的较大值：(x > y) and x or y。当x > y时，第一个表达式的结果为真，所以and的结果为第二个表达式x，由于x是数字所以肯定是真值，接着进行or运算，由于x为真直接返回x。当x > y为假，那么and返回假，or运算返回它的第二个表达式，即y。

not操作符对真假值取反，结果总是一个布尔类型的表达式：

> not nil --> true

> not false --> true

> not 0 --> false

> not not 1 --> true

> not not nil --> false

### 13.1.8 独立解释器

独立解释器（有时称为lua .c，有时称为lua，分别是它的源码文件名和可执行文件名），是一个小程序，允许直接使用lua。本节介绍常见的用法。

当解释器加载一个文件时，它会忽略以#号开始的第一行。这个特性是为了保证脚本文件与POSIX系统的兼容性。如果脚本第一行写为：

#!/usr/local/bin/lua

或写为（依lua执行文件的路径而定）：

#!/usr/bin/env lua

那么就可以直接执行这个脚本，而不需要明确执行lua。

lua解释器的用法如下：

lua [options] [script [args]]

所有的参数都是可选的。如我们所见，当不带参数执行lua时，会进入交互模式。

参数-e允许我们直接在命令行中输入lua代码，像这样：

% lua -e "print(math.sin(12))" --> -0.53657291800043

（POSIX系统要求将不需要解释执行的部分用双引号括起来）

-l选项可以加载一个库。如前面所说，-i参数让lua在进入交互模式之前先运行其它脚本。因此下面的写法会先加载lib这个库，然后再执行赋值语句x=10，最后显示执行结果。

% lua -i -llib -e "x = 10"

当我们在交互模式中写出下面的表达式时，Lua会打印它的值：

> math.sin(3) --> 0.14112000805987

> a = 30

> a --> 30

（注意，这个功能是Lua 5.3版新增的，较老的版本中要使用前缀等号来实现。）要关闭这个打印表达式的值的功能，可以在行位加一个分号：

> io.flush() --> true

> io.flush();

分号让这一行不构成完整的表达式，但依然是一个有效的指令。

在运行参数之前，解释器会查找一个类似LUA\_INIT\_5\_3的环境变量，如果不存在则查找LUA\_INIT。如果找到了类似的环境变量且其内容为“@文件名”，那么解释器就会先运行那个文件。如果存在该环境变量但其内容不以@开头，那么解释器会认为该内容是Lua代码，并执行这段内容。使用LUA\_INIT来配置独立解释器是一种很强大的特性，因为在配置时可以用到所有的Lua语言功能。例如可以预先加载一些库，修改系统路径，定义特殊的函数，删除或重命名已有的函数，等等。

在Lua脚本中，可以通过访问全局变量arg来获得命令行参数。比如执行 % lua script a b c，解释器预先会创建一个arg表包含所有的命令行参数。脚本名称放在下标0的位置，第一个参数（即“a”）放在下标1的位置，等等。其它在脚本名称之前的参数，会放在下标为负数的位置上，因为它们是在脚本名称之前出现的。比如如下命令：

% lua -e "sin=math.sin" script a b

命令行参数内容如下：

arg[-3] = "lua"

arg[-2] = "-e"

arg[-1] = "sin=math.sin"

arg[0] = "script"

arg[1] = "a"

arg[2] = "b"

通常脚本会使用正数部分的下标（本例中是arg[1]和arg[2]）。

脚本也可以通过变参表达式来访问这些命令行参数。在脚本的主体部分使用表达式“...”（三个点）就可以访问到命令行参数（后面的“函数变参”章节会讨论变参表达式）。

### 13.1.9 练习

练习1.1 安装lua运行环境，编写Hello world进行测试。

练习1.2 执行求阶乘的例子，当输入负数时会怎样呢？试着修正这个问题。

练习1.3 执行“twice”的例子，尝试-l参数和dofile两种方式。你更喜欢哪种？

练习1.4 下列哪些字符串是合法的标识符？

\_\_\_ \_end End end until ? nil NULL one-step

练习1.5 表达式type(nil) == nil的值是什么？（可以实际试一下）。你能解释原因吗？

练习1.6 如何检查一个值是否是布尔类型，且不适用type函数？

练习1.7 考虑下面的表达式：

(x and y and (not z)) or ((not y) and x)

括号是必要的吗？这里括号的使用是不是值得推荐的？

练习1.8 编写一个简单的脚本，功能为输出脚本自身的名称。

## 13.2 前奏：八皇后问题

本节我们插入一个短小的章节，展示一个简单但完整的Lua程序，解决八皇后问题：在国际象棋的棋盘上放置八个皇后，条件是让它们不能互相攻击到。

这里的代码没有用到非常特殊的Lua特性，应当可以仅通过语法改动将它翻译为其他的编程语言。这里讨论这些代码，目的是让读者对Lua有一个大体的认识，比如Lua语法看起来是什么样的，暂时不需要深入细节。一些不清楚的地方会在后面的章节中陆续谈到。

### 13.2.1 思路

解决八皇后问题的第一步，是意识到每一行只能有一个皇后棋子。这样就意味着，我们可以用一个简单的8个元素的数组来表示这8个皇后的位置，每个下标代表一行，每个元素代表该棋子的列数。例如数组{3, 7, 2, 1, 8, 6, 5, 4} 表示八个皇后位于(1,3), (2, 7), (3, 2), (4, 1), (5, 8), (6, 6), (7, 5)和(8, 4)（顺便说一下，这个例子并非一个有效的解法，因为(3, 2)的棋子可以攻击到(4, 1)的棋子）。注意到所有皇后的行坐标、列坐标都一定是从1到8，因此正确的解法中，8个棋子的列数也不能重复。

### 13.2.2 范例代码

解决八皇后问题，完整代码如下：

-- The eight-queen program

N = 8 -- board size 棋盘大小

-- 检查位置(n,c)是否是一个安全的位置，a是数组

function isplaceok (a, n, c)

for i = 1, n - 1 do -- 对每一个已经摆放好的列

if (a[i] == c) or -- 同一列

(a[i] - i == c - n) or -- 同斜线

(a[i] + i == c + n) then -- 同斜线

return false -- 位置是可被攻击到的

end

end

return true -- 位置是安全的

end

-- 打印棋盘

function printsolution (a)

for i = 1, N do -- 循环每一行

for j = 1, N do -- 循环每一列

--打印X和横线

io.write(a[i] == j and "X" or "-", " ")

end

io.write("\n")

end

io.write("\n")

end

-- 添加一个棋子，n代表第几个棋子

function addqueen (a, n)

if n > N then -- 已经完成了最后一列

printsolution(a)

else -- 尝试放第N个皇后

for c = 1, N do

if isplaceok(a, n, c) then

a[n] = c -- 将第N个皇后放在第c行

addqueen(a, n + 1)

end

end

end

end

-- 启动程序

addqueen({}, 1)

第一个函数是isplaceok，用来检查一个指定的位置是否是安全位置，即是否能被之前放置过的棋子攻击到。进一步的，还会检查同一列上是否已经有其它皇后。记住，之前已经提到过，一个正确的解法中，每一列也只能有一个皇后。因此isplaceok会检查是否有同一列、同斜线的皇后，以判定某一格是否是可以放置的。

下一个函数是printsolution，用来打印整个棋盘。它只是简单地遍历了整个棋盘，在有皇后的位置打印X，其它位置打印减号，而没有过多的修饰（这里使用了and-or技巧来选择合适的字符）。打印效果像这样：

X - - - - - - -

- - - - X - - -

- - - - - - - X

- - - - - X - -

- - X - - - - -

- - - - - - X -

- X - - - - - -

- - - X - - - -

最后一个函数addqueen，是程序的核心。它的作用是放置所有大于等于n（序号）的皇后到棋盘上。它使用了递归调用来搜索解法。首先，它会检查求解过程是否已经完成，如果已经完成，就打印结果。否则，就继续为第n个皇后循环遍历所有的列，在找到安全的列后，就把第n个皇后放在该位置并继续求解下一个皇后的位置。

最后，在程序的主体部分（函数之外）调用addqueen函数。

### 13.2.3 练习

练习2.1 修改八皇后问题的程序，让它在打印出第一个解之后就结束。

练习2.2 思考和尝试八皇后问题的非递归解法。如果发现非递归解法比较困难，那么具体难点在哪？

**参考：《8皇后问题，一种非递归的极简写法》**<https://zhuanlan.zhihu.com/p/80066564>

## 13.3 数值

直到5.2版本，Lua都一直使用双精度浮点数来表示所有的数字。从5.3版本开始，Lua使用两种方式来表示数字：64位整数，简称为整数，以及双精度浮点数，简称为浮点数。（注意本书中提到的浮点数或float，都是指双精度浮点数，而不是通常的单精度的）。对某些特定平台来说，可以在编译Lua5.3时设置为“Small Lua”，也就是使用32位整型以及单精度浮点数。

对整型的介绍对Lua 5.3来说是标志性的，因为它是与之前版本的主要区别之一。不过，这种改动带来的不兼容性极小，因为双精度浮点数也可以精确地表示到253这么大的整数。大部分关于数字类型的说明，其实对于5.2以及之前的版本也是适用的。在本章结尾我会详细讨论更多不兼容的细节。

### 13.3.1 数值类型

数字常量可以直接写出，可以包含十进制数与十进制指数两部分（指数可以没有），如下所示：

> 4 --> 4

> 0.4 --> 0.4

> 4.57e-3 --> 0.00457

> 0.3e12 --> 300000000000.0

> 5E+20 --> 5e+20

带有小数点或指数的数字，会被认为是浮点数；否则就作为整数。

无论整数或浮点数，类型都是“number”：

> type(3) --> number

> type(3.5) --> number

> type(3.0) --> number

它们具有同样的类型，因为很多时候它们是会互相转换的。另外，具有同样值的整数与小数，会被认为是相等的：

> 1 == 1.0 --> true

> -3 == -3.0 --> true

> 0.2e3 == 200 --> true

少数情况下，需要确切地知道一个数字是整数还是小数，这是可以用math.type函数得出：

> math.type(3) --> integer

> math.type(3.0) --> float

另外，整数和小数的显示结果也略有区别：

> 3 --> 3

> 3.0 --> 3.0

> 1000 --> 1000

> 1e3 --> 1000.0

与大部分语言一样，Lua也支持16进制表示的常数，只需要在数字前面加上0x即可。而且与其它语言不同的是，Lua也支持浮点的16进制常数，它具有一个小数部分和一个以2为底的指数部分，指数部分以p或P为前缀。下面的例子展示了这种格式：

> 0xff --> 255

> 0x1A3 --> 419

> 0x0.2 --> 0.125

> 0x1p-1 --> 0.5

> 0xa.bp2 --> 42.75

使用string.format函数的”%a”参数，可以显示出数字的16进制表示：

> string.format("%a", 419) --> 0x1.a3p+8

> string.format("%a", 0.1) --> 0x1.999999999999ap-4

尽管不太容易阅读，这种格式可以保留浮点数的所有精度信息，而且转换也比十进制的要快。

### 13.3.2 算数运算符

Lua支持常用的算数操作符：加法、减法、乘法、除法和负号。还支持整除、模运算以及指数。

之前在对Lua 5.3中整数的介绍中，最重要的信息之一是“请程序编写者尽可能忽略整数和浮点数的差异，或者说不要试图完全掌控数字的具体类型是整数还是浮点数”。尽管如此，所有的算数运算应当对于相同的数字给出完全一样的结果。

两个整数的和一定是一个整数，这一点对于减法、乘法、负号来说也是一样的。对于这些操作，操作数是整型还是浮点型（与对应整数相等的），对结果是没有影响的（溢出的情况除外，稍后在“表示范围”的章节中会谈到）；也就是说无论结果是浮点数还是整数，都是相等的。

> 13 + 15 --> 28

> 13.0 + 15.0 --> 28.0

如果两个操作数都是整型，结果也是整型；否则结果是浮点型。对类型不同的两个操作数，Lua会把整型转换为浮点型再进行运算：

> 13.0 + 25 --> 38.0

> -(3 \* 6.0) --> -18.0

而除法并不符合这个规则，因为两个整数的商不一定是一个整数（在数学中表达为——整数对于除法运算不封闭）。为了避免整数与浮点数除法的差异，除法总是返回浮点型的结果：

> 3.0 / 2.0 --> 1.5

> 3 / 2 --> 1.5

为了方便整数的除法，Lua 5.3提供了一个新的运算符，称为整除（floor division），写作“//”。如它的名字所示，整除总是将结果向负无穷大的方向取整，以保证结果总是一个整数值。根据这个定义，两个操作数的结果和加法遵循同样的规则：两个整型操作数整除结果是整型；否则结果是浮点型（与对应整数相等）。

> 3 // 2 --> 1

> 3.0 // 2 --> 1.0

> 6 // 2 --> 3

> 6.0 // 2.0 --> 3.0

> -9 // 2 --> -5

> 1.5 // 0.5 --> 3.0

以下等式定义了模运算：

a % b == a - ((a // b) \* b)

对整型操作数会保证得到整形的结果，所以模运算也符合算数操作的规则：如果两个操作数都是整型，那么结果也是整型；否则结果是浮点型。

对整型操作数来说，结果与除数的正负号相同。特别的，对于任意正整数K，x % K的结果总是在[0,K-1]的范围内，就算x是负数也是如此。例如，i % 2对任意i的结果总是0或1。

对浮点操作数来说，模运算会有点意外。例如，x - x % 0.01的结果是x保留两位十进制小数，而x - x % 0.001则是保留三位小数。

> x = math.pi

> x - x%0.01 --> 3.14

> x - x%0.001 --> 3.141

举例一个模运算的具体应用，假设我们想要检查一辆车的转向角度是否需要反转。如果方向以角度表示，可以采用下面的算法：

local tolerance = 10

function isturnback (angle)

angle = angle % 360

return (math.abs(angle - 180) < tolerance)

end

这个方法对于负的角度也适用：

print(isturnback(-180)) --> true

如果想要以弧度计算而不是角度，只需要简单修改为：

local tolerance = 0.17

function isturnback(angle)

angle = angle % (2\*math.pi)

return (math.abs(angle - math.pi) < tolerance)

end

只需要使用表达式angle % (2 \* math.pi)，就可以将任意角度限定在[0, 2π)的范围之中。

Lua也提供了指数运算符，用“^”表示。与除法类似，它的结果总是浮点数。（整数对幂运算不闭合，比如2-2就不是一个整数）。可以写x^0.5来求x的平方根，x^(1/3)来求x的立方根。

### 13.3.3 关系运算符

Lua提供了一下关系运算符：

< > <= >= == ~=

以上这些运算符的结果总是一个布尔值。

运算符==用于测试相等性；~=测试不等性。可以将这二者用于任何类型的值。如果两个值的类型不同，那么就肯定不相等。否则，Lua再根据类型进行具体的判断。

number类型是同一种类型，而不论具体是浮点型还是整型。影响相等性的只有它们具体的值（但是对同种子类型的数字进行比较，还是会快一些）。

### 13.3.4 数学库

Lua提供给了标准库math，内含一组常用的数学函数，包括三角函数（sin, cos, tan, asin等等）、对数函数、取整函数、最大值函数max和最小值函数min，还有伪随机函数random，以及一些常数比如π、huge（无穷大，在大部分平台上是一个特殊值inf）。

> math.sin(math.pi / 2) --> 1.0

> math.max(10.4, 7, -3, 20) --> 20

> math.huge --> inf

所有的三角函数都以弧度为准。可以使用deg和rad函数来转换角度和弧度。

### 13.3.5 随机数发生器

math.random函数用于生成伪随机数。可以用三种方式调用它。如果不填写参数，则返回一个均匀分布在[0,1)之间的随机数。如果只填写一个整数参数n，则它返回一个[1,n]范围的随机数。例如，可以用random(6)模拟投骰子的结果。最后，还可以用两个整数l和u作为random的参数，得到[l,u]范围的随机数。

可以通过调用函数randomseed为随机数发生器设置一个种子；它接受一个数字参数作为种子。当程序开始运行时，系统会初始化固定的随机数种子1。如果不设置种子，那么每次程序都会生成同样的随机数序列。在调试时这样会比较有用，但在实际游戏中就会每次都得到同样的结果。为了产生不同的随机数，一种常用的方法是使用当前时间作为随机数种子，也就是调用math.randomseed(os.time())。之后的章节会介绍os.time函数。

### 13.3.6 取整函数

数学库提供了三种取整函数：floor、ceil以及modf。floor函数会向负无穷大方向取整，ceil函数会向无穷大方向取整，modf则向零点方向取整。当返回的结果可以用整型表示，则返回整型，否则返回浮点型（与对应整数相等）。函数modf比较特殊，不仅返回取整结果，还会返回小数部分作为第二个参数。

> math.floor(3.3) --> 3

> math.floor(-3.3) --> -4

之后会讲到，Lua中的函数可以有多个返回值。

> math.ceil(3.3) --> 4

> math.ceil(-3.3) --> -3

> math.modf(3.3) --> 3 0.3

> math.modf(-3.3) --> -3 -0.3

> math.floor(2^70) --> 1.1805916207174e+21

如果参数已经是整型了，那么就原封不动地返回。

如果想要将一个数x向最近的整数取整，可以用floor(x+0.5)得出。但是，当x非常大时，可能会得到错误的结果。例如考虑下面的数字：

x = 2^52 + 1

print(string.format("%d %d", x, math.floor(x + 0.5)))

--> 4503599627370497 4503599627370498

这个问题出现的原因是252 + 1.5 没有浮点数能够精确表示它，所以最终以一种不可控的方式被取整了。要避免这个问题，可以对整数做一个特殊处理：

function round (x)

local f = math.floor(x)

if x == f then return f

else return math.floor(x + 0.5)

end

end

上面的函数总是会将超过0.5的小数提升到下一个整数（例如2.5进位为3）。如果我们想要无偏差取整（进位到下一个最近的偶数），那么当x+0.5是奇数时，之前的算法就不对了：

> math.floor(3.5 + 0.5) --> 4 (ok)

> math.floor(2.5 + 0.5) --> 3 (wrong)

模运算在这里再次派上了用场：（x % 2.0 == 0.5）只有当x+0.5是奇数时才会为真，利用这点就可以进行判断了。采用这个方法定义一个无偏差取整函数：

function round (x)

local f = math.floor(x)

if (x == f) or (x % 2.0 == 0.5) then

return f

else

return math.floor(x + 0.5)

end

end

print(round(2.5)) --> 2

print(round(3.5)) --> 4

print(round(-2.5)) --> -2

print(round(-1.5)) --> -2

### 13.3.7 数字的范围限制

大部分编程语言中的数字表示都采用了固定的二进制位数，这种表示意味着数值表示范围是有限的，包括总的范围和精度。

标准Lua采用了64位整数。64位整数最大表示的整数为263 - 1，大约是1019。（Small Lua采用32位整数，最大值约21亿）。math库定义了常数math.maxinteger为最大整数，math.mininteger为最小整数。

64位整数的最大值是一个非常大的数字，它可以达到世界财富总量（以美分计）的数千倍，或是世界总人口的一百万倍。只有超出这个范围才会引起溢出。如果在计算时结果超过了最大值或者低于最小值的时候，数值会发生回滚。

在数学角度来看，“回滚”意思是运算的结果必须在最小整数和最大整数之间，也就是该结果模264的结果。也就是在具体计算时，会抛弃最后一个进位（最后一个进位位于第65位上，因此忽略这一位不会影响模264的结果）。在Lua中，对整数进行这种操作的结果是明确而稳定的：

> math.maxinteger + 1 == math.mininteger --> true

> math.mininteger - 1 == math.maxinteger --> true

> -math.mininteger == math.mininteger --> true

> math.mininteger // -1 == math.mininteger --> true

最大的整数是0x7ff...fff，也就是除二进制最高位为0外其它位都是1，最高位正是整数的符号位（1表示负，0表示非负）。如果给这个最大整数加1，就得到了0x800...000，正好是最小整数的表示。在模264时，最小整数比最大整数大1，如下面所示：

> math.maxinteger --> 9223372036854775807

> 0x7fffffffffffffff --> 9223372036854775807

> math.mininteger --> -9223372036854775808

> 0x8000000000000000 --> -9223372036854775808

标准Lua使用双精度浮点数。它也用64个bit表示一个数字，其实11位是指数部分。双精度浮点数可以表示约16个十进制小数位，范围从 -10308 到 10308。（Small Lua使用单精度浮点数，32个bit，范围从-1038 到 1038，约7个十进制小数位）。

双精度浮点数的范围对大多数程序来说都足够用了，但是我们还是要了解其限制的精度。计算机中的精度问题，与用笔和纸计算并没有什么不同。比如如果用10个数字表示1/7，就要写为0.142857142。如果用它来计算1/7 \* 7，那么就得到了0.999999994，而不是1。甚至在十进制中有限长度的小数，在二进制中也可能是无限长的。例如，12.7 - 20 + 7.3在用双精度型计算时，它的结果就不是精确的0。因为12.7和7.3在二进制下没有精确的表示（练习3.5中会提到）。

由于整数和浮点数有着不同的限制范围，所以当运算超出范围时，两种类型的数字会有完全不同的结果：

> math.maxinteger + 2 --> -9223372036854775807

> math.maxinteger + 2.0 --> 9.2233720368548e+18

在上面两个例子中，无论哪个从数学角度看都是错误的。第一行是整数加法越界，导致了回滚。第二行则是浮点数加法，虽然没有“越界”，但是会被降低精度，结果是一个近似值。对于这种情况再举一个例子：

> math.maxinteger + 2.0 == math.maxinteger + 1.0 --> true

每种表达方式都有其优点。当然，浮点数可以表示小数部分，且浮点数的范围比整数大很多，但是浮点数能够精确表示数字的范围只有[-253,253]。除了这些限制以外，很多时候都可以忽略整数和浮点数的区别。而如果接近了越界范围的时候，就要对数字的类型多加小心了。

### 13.3.8 数字的转换

要强制转换一个数字为浮点数，只要简单地加上0.0即可。整数会被自动转换为浮点数：

> -3 + 0.0 --> -3.0

> 0x7fffffffffffffff + 0.0 --> 9.2233720368548e+18

用双精度浮点数表示整数时，只能精确地表示到 253 (即9007199254740992)的大小。如果整数的值大于这个范围，在转换浮点数时就会丢失一部分精度：

> 9007199254740991 + 0.0 == 9007199254740991 --> true

> 9007199254740992 + 0.0 == 9007199254740992 --> true

> 9007199254740993 + 0.0 == 9007199254740993 --> false

上面最后一行中， 253 +1被取整为253，因此被判断为不相等。要向保证一个数字为整型，可以考虑将它和0取“按位或（符号为 |）”。

> 2^53 --> 9.007199254741e+15 (float)

> 2^53 | 0 --> 9007199254740992 (integer)

使用“按位或”时，Lua要求操作数有精确的二进制表示，也就是说没有小数部分，且在整型表示的范围之内。否则就会产生一个错误：

> 3.2 | 0 -- fractional part

stdin:1: number has no integer representation

> 2^64 | 0 -- out of range

stdin:1: number has no integer representation

> math.random(1, 3.5)

stdin:1: bad argument #2 to 'random'

(number has no integer representation)

要对一个数字取整，要用到取整相关的函数。

另一种强制数字为整型的方法是使用math.tointeger函数，当数字无法被转换时，结果返回nil。

> math.tointeger(-258.0) --> -258

> math.tointeger(2^30) --> 1073741824

> math.tointeger(5.01) --> nil (not an integral value)

> math.tointeger(2^64) --> nil (out of range)

这个函数用于检查数字是否能被转换时，非常有用。例如，下面的例子会在参数可以转换为整数时，返回转换后的整数，否则返回原数字：

function cond2int (x)

return math.tointeger(x) or x

End

### 13.3.9 运算符优先级

Lua中，运算符的优先级如下表，优先级从高到低：

^

单目运算符 (- # ~ not)

\* / // %

+ -

.. (concatentation)

<< >> (bitwise shifts)

& (bitwise AND)

~ (bitwise exclusive OR)

| (bitwise OR)

< > <= >= ~= ==

and

or

所有的二进制运算符都是左结合的，除了指数和连接运算符是右结合。以下写法中，左边的写法和右边等价：

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)

如果在写程序时拿不准，可以使用小括号明确指定。

### 13.3.10 旧版本兼容性

并非偶然，Lua 5.3版中的整型数字与之前的版本极少有不兼容的地方。如前面所说，程序编写者大部分情况下都可以忽略整型与浮点型的差异。如果忽略了整型与浮点型的差异，那么也就可以忽略Lua 5.3与Lua5.2在数字类型方面的差异，Lua5.2中所有数字都是浮点型的（在这一点上Lua5.0、5.1与5.2版是相同的）。

当然，Lua 5.3与5.2的主要区别是整数的表示范围。Lua 5.2只能精确表示最大为253的数字，而Lua5.3则扩大到了263。在表示物体数量时，这基本不会带来问题。但是当数字用在一些二进制操作时（比如将三个20bit的数字压缩为一个数字表示），就会带来问题了。

虽然Lua 5.2并不支持整型数字，但是这个问题被语言的设计所化解了。例如，C语言实现的库函数通常就接受整型参数。Lua 5.2并不严格规定浮点型如何被转换为整型，规范中只说了“数字以非特定方式截取（[the number] is truncated in some non-specified way）”。这个问题实际上确实存在，Lua 5.2有可能将-3.2转换为-3或者-4，取决于特定的平台。Lua 5.3则准确定义了转换问题，确保一定是确定的整数。

Lua 5.2并不提供math.type函数，因为所有的数字子类型也一样。Lua 5.2也不提供math.maxinteger和math.mininteger函数，因为没有整型。Lua 5.2不提供整除操作符，尽管可以做到（毕竟取模运算符可以实现整除操作）。

让人意外的是，Lua最终引入整型的主要原因，是为了解决数字如何转为字符串的问题。Lua 5.2将所有不带小数的整数以整数表示，不带小数点。而Lua 5.3将浮点数以带小数的方式表示，有可能是以小数方式或者带有指数的方式。也就是说，Lua 5.2会将3.0转为字符串”3” ，而Lua 5.3则会转为”3.0”。尽管Lua标准没有定义如何将数字转为字符串，但是这个小瑕疵引发的问题却比较深远——整数会无理由地被转为浮点数。（实际上，这就是Lua 5.3规定新的格式化规则的主要动机。整数被以浮点数表示，通常是一种坏的味道（bad smell）。新的格式化规则将这种坏味道暴露了出来。）

### 13.3.11 练习

**练习3.1** 以下哪些是合法的数字？它们代表什么数？

.0e12 .e12 0.0e 0x12 0xABFG 0xA FFFF 0xFFFFFFFF

0x 0x1P10 0.1e1 0x0.1p1

**练习3.2** 解释以下结果：

> math.maxinteger \* 2 --> -2

> math.mininteger \* 2 --> 0

> math.maxinteger \* math.maxinteger --> 1

> math.mininteger \* math.mininteger --> 0

（记住，整数越界时总是会“回滚”）

**练习3.3** 以下程序会打印什么？

for i = -10, 10 do

print(i, i % 3)

end

**练习3.4** 2^3^4的执行结果是什么？2^-3^4呢？

**练习3.5** 数字12.7等于分数127/10，其中分母为10的幂。你能将12.7转化为分母为2的幂的分数吗？数字5.5呢？

**练习3.6** 写一个函数，计算正圆锥体的体积，参数为高度、母线与轴的夹角。

**练习3.7** 利用math.random函数，写一个函数生成符合标准正态分布（高斯分布）的随机数。

参考：Box-Muller方法。

<https://blog.csdn.net/doublehhcc/article/details/81166502>（数学算法介绍）

<https://blog.csdn.net/thrt520asd/article/details/81778654>（Lua参考）

## 13.4 字符串

### 13.4.1 字符串

字符串用来表示文本。Lua中的字符串可以包含一个字母也可以是一整本书。用一个字符串表示100K甚至1M的内容在Lua中并不罕见。

在Lua中，字符串是字节的序列。Lua内核中不关心这些字节是什么编码格式。Lua只是单纯地以8bit为单位保存它们，且每个字节可以是任意数字，包括0。这就意味着我们可以用字符串保存任意二进制数据。也可以将Unicode字符串保存为任意表示方式（UTF-8，UTF-16等等）；但是，下面会谈到，尽可能使用UTF-8与有很多好处。Lua的标准字符串库假定了字符最少占一个字节，这与UTF-8有良好的兼容性。进一步说，从5.3版本开始，Lua内置了一个小的处理UTF-8编码的库。

Lua中的字符串是不可变的（immutable）。不可能单独修改字符串中的一个字符，像C语言那样。而是需要创建一个新的字符串，且将它创建为你想要的样子：

a = "one string"

b = string.gsub(a, "one", "another") -- 修改部分字符

print(a) --> one string

print(b) --> another string

Lua中的字符串是自动内存管理的对象，与其它Lua对象相同（表、函数等等）。这意味着不需要关心字符串空间的申请和释放；Lua已经为我们处理好了。

可以使用取长度运算符获取字符串的长度（写作“#”）。

a = "hello"

print(#a) --> 5

print(#"good bye") --> 8

取长度的结果总是以字节数表示，在不同的编码下可能与实际表示的字符数不同。

可以通过字符串连接运算符“..”（两个点）来连接两个字符串。如果其中一个操作数是数字，那么就会将数字转为字符串并连接：

> "Hello " .. "World" --> Hello World

> "result is " .. 3 --> result is 3

某些语言使用加号连接字符串，但是3+5 和 3..5是不同的。）

记住，Lua中的字符串是不可变对象。连接操作总是产生一个新的字符串，而不会修改原有的字符串：

> a = "Hello"

> a .. " World" --> Hello World

> a --> Hello

### 13.4.2 字符串字面量

使用单引号或双引号，可以创建一个字符串，其内容就是引号内的部分：

a = "a line"

b = 'another line'

以上两种写法完全等价；唯一的区别是，在单引号表示的字符串内部，可以放心地使用双引号而不需要转义字符；反之也是同样的。

为了统一风格，大部分程序编写者会对同种类的字符串使用同样的引号，这种字符串的“种类”取决于具体的程序。例如，一个处理XML的程序会为表示XML字段的字符串使用单引号，因为这些字段内往往会包含双引号。

Lua中的字符串，支持C风格的转义字符：

\a bell 响铃

\b back space 退格

\f form feed 换页

\n newline 换行

\r carriage return 回车

\t horizontal tab 制表符

\v vertical tab 纵向制表符

\\ backslash 反斜杠

\" double quote 双引号

\' single quote 单引号

转义字符的示例如下：

>print("one line\nnext line\n\"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: '\'

我们同样可以用数字的方式来表示字符串中的一个字符，形式为\ddd和\xhh，其中ddd三位数字，而hh是两个16进制数字。举一个刻意的例子：字符串”ALO\n123\””与字符串’\x41LO\10\04923’具有同样的值，在符合ASCII规范的系统中，0x41(即十进制的65)是字符A的编码，10是换行符的编码，而49是数字字符1的编码（这里注意必须写作\049，因为后面跟着的还是一个数字，会造成歧义，被误认为是\492）。我们还可以把以上字符串写作’\x41\x4c\x4f\x0a\x31\x32\x33\x22’，将每个字符都用十六进制编码表示。

从Lua 5.3版本开始，还可以用\u{h...h}的方式指定UTF-8编码的字符串；在大括号内可以写上任意数量的十六进制数字以表示一串字符：

> "\u{3b1} \u{3b2} \u{3b3}" --> # # #

（以上例子假设是在一个UTF-8终端中执行的。）

### 13.4.3 长字符串

我们还可以使用两个方括号来定义一个字符串，类似多行注释那样。这种方式定义的字符串可以跨多行且不需要转义字符。而且如果第一个字符是一个换行符，还会忽略第一个字符。这种定义字符串的方式对于编写很长的字符串非常方便，举例如下：

page = [[

<html>

<head>

<title>An HTML Page</title>

</head>

<body>

<a href="http://www.lua.org">Lua</a>

</body>

</html>

]]

write(page)

有时候，其它代码也可能会用到两个连续的中括号，比如a = b[c[i]]（注意结尾是]]），或是去包含已经被多行注释过的内容时也会有这个问题。要处理这种情况，只需要在两个左方括号之间加上任意数量的等号：例如[===[。这样一来，就必须要用同样等号数量的方括号来配对：]===]，编译器会忽略掉等号数量不同的中括号。通过选择不同数量的等号，就可以去包含任意一段代码而不需要对它做出修改。这就解决了中括号冲突问题。

同样的方法也可以用于多行注释。例如如果一个注释以--[=[开始，那么就要以]=]结束。这种方法适用于多行注释发生嵌套的情形。

长字符串非常适用于表示字面字符串的情况，但是不应当用它来表示非文本类的字符串。尽管长字符串可以包含不可打印的二进制字节，但是使用长字符串直接包含二进制数据是不合适的（比如说你的编辑器可能会有显示问题）；甚至，回车符例如“\r\n”会被转换成“\n”。更好的方式是在表示二进制数据时，使用数字的转义字符，例如”\x13\x01\xA1\xBB”。但是，在字符串较长时，会导致一行源码内有一个非常长的字符串。对于这种情况，Lua 5.2版本提供了一个转义字符\z，它会跳过之后的所有空白类字符，直到第一个飞空白字符。如下面的例子：

data = "\x00\x01\x02\x03\x04\x05\x06\x07\z \x08\x09\x0A\x0B\x0C\x0D\x0E\x0F"

第一行结尾的\z的作用是，跳过第一行结尾的换行符和第二行开头的空白字符，直到下一个\x08，也就是说\x08紧接在\x07的后面。这种写法方便我们把一个很长的字符串写在多行里。

### 13.4.4 隐式转换

Lua在运行时，会在字符串和数字之间做自动的转换。在使用数字操作符时，会尝试将字符串转为数字。Lua只有在遇到数字操作符时才会进行这种转换，但是一些期望得到数字的情况下，也会进行这种转换，例如math.sin函数的参数。

相对的，当使用字符串操作符时，Lua会自动将数字转化为字符串：

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

（注意，如果在数字之后使用一个连接符，那么应当在连接符之前空一格；否则连接符可能被当作小数点而造成错误。）

很多人认为Lua的这种自动转换不是一个良好的设计。作为一种规则，不应当依赖这种转换编写代码。这种转换只在某些情况下比较有用，但是增大了语言与程序的复杂度。

因此Lua5.3没有对隐式转换做出完整的实现，而是采用了更为简单高效的实现方式。规则为：对数字运算来说，只有当两个操作数均为整数时，结果才是一个整数。字符串不是整数，所以任何对字符串的数字运算，都会尝试将字符串转为浮点数：

> "10" + 1 --> 11.0

要明确地将字符串转为数字，可以使用函数tonumber，当字符串不能转为数字时返回nil；否则返回一个整数或浮点数，与编译器的规则保持一致：

> tonumber(" -3 ") --> -3

> tonumber(" 10e4 ") --> 100000.0

> tonumber("10e") --> nil (not a valid number)

> tonumber("0x1.3p-4") --> 0.07421875

默认的，tonumber假定数字为十进制的，但是可以通过参数指定从2到36的任意进制：

> tonumber("100101", 2) --> 37

> tonumber("fff", 16) --> 4095

> tonumber("-ZZ", 36) --> -1295

> tonumber("987", 8) --> nil

上面最后一行，由于“987”对于八进制来说不是一个合法的数字，所以返回值为nil。

要将一个字符串转为数字，可以明确调用函数tostring：

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

转换为字符串的操作总是有效的。但是要注意我们对于转换的结果是没有控制力的（例如小数的位数）。要更细节地控制转换的格式，应当使用string.format函数，后面会讲到。

与数字操作符不同，比较操作符不会对参数进行转换。记住，”0”与0不同，进一步的，2<15结果显然是true，但”2”<”15”就是false了（按字典顺序比较）。为了避免不一致的结果，Lua会在对一个字符串和一个数字进行比较操作时抛出异常，例如2 < “15”。

### 13.4.5 字符串库

Lua解释器能够对string进行的操作是极为有限的。可以创建、连接和比较字符串，也可以获得字符串的长度。但是在截取字符串、检查内容时就不行了。Lua中，操作字符串的强大功能来自于字符串库（string library）。

前面提到过，字符串库假设字符串符合1字节的编码。这个假设对于某些编码来说是正确的（例如ASCII或ISO-8859-1），但是它在用于Unicode时就不适用了。但是没关系，后面会看到，很多功能对于UTF-8编码的字符串也是很有用的。

字符串库中的某些函数非常简单：比如调用string.len(s)会返回字符串s的长度，等价于#s。而string.rep(s, n)则返回s重复n次的结果。可以创建一个长度为1MB的字符串，使用string.rep(“a”, 2^20)即可。函数string.reverse可以反向一个字符串。而string.lower(s)将s中的所有大写字母转为小写，其它字符不变。类似的，string.upper函数会将小写字母转为大写。

>string.rep("abc", 3)

>string.reverse("A Long Line!")

>string.lower("A Long Line!")

>string.upper("A Long Line!")

--> abcabcabc

--> !eniL gnoL A

--> a long line!

--> A LONG LINE!

有一种很常见的用法，比较两个字符串且忽略大小写时，就可以这样写：

string.lower(a) < string.lower(b)

调用string.sub(s, i, j)会截取字符串s的一部分并返回，截取范围是从第i个字符到第j个字符，包含i和j（注意下标从1开始的）。同样可以使用负数的i和j，负数代表从后往前数，比如-1就是最后一个字符，-2是倒数第二个字符等等。举例来说，string.sub(s, 1, j)截取s的前j个字符；string.sub(s, j, -1)会截取从j开始一直到结束的字符；string.sub(s, 2, -2)截取的是除了第一个和最后一个字符的部分。

> s = "[in brackets]"

> string.sub(s, 2, -2) --> in brackets

> string.sub(s, 1, 1) --> [

> string.sub(s, -1, -1) --> ]

记住，Lua中的字符串是不可修改的。因此string.lua也不会修改字符串的值，只会返回一个新的字符串。常见的错误是使用string.sub(s, 2, -2)，以为s的值被修改了。如果要修改s的值，必须进行明确的赋值操作：

s = string.sub(s, 2, -2)

使用string.char和string.byte可以在字符和内部编码数字之间转换。string.char接收0到多个参数，将每个数字转换为对应的字符，然后再连接成一个字符串。而string.byte(s, i)会返回s的第i个字符对应的数字，第二个参数i是可以省略的，默认值是1。以下例子都假设使用ASCII编码：

print(string.char(97)) --> a

i = 99; print(string.char(i, i+1, i+2)) --> cde

print(string.byte("abc")) --> 97

print(string.byte("abc", 2)) --> 98

print(string.byte("abc", -1)) --> 99

最后一行代码使用了负数下标来获取最后一个字符。

string.byte(s, i, j)这种用法会返回多个值，分别是字符串s从第i个到第j个字符的数字表示。

print(string.byte("abc", 1, 2)) --> 97 98

有一种很好的用法是string.byte(s, 1, -1)，会将字符串s转化为一个数字序列。（注意这个技巧只对小于1MB的字符串有效。因为返回值的栈空间是有限的，默认为一百万个位置。）

函数string.format也具有强大的功能，用于格式化字符串或将数字转为字符串。他会返回调用的第一个参数的副本，第一个参数也就是“格式字符串”，其中有一些“指示符”用来指定接下来的参数以什么样的格式替换到字符串之中。指示符的设计类似与C语言中的printf。指示符以百分号开始，后跟一个字符来说明如何格式化参数：d表示十进制整数，x是十六进制整数，f是浮点数，s是字符串，还有其它几种。

> string.format("x = %d y = %d", 10, 20) --> x = 10 y = 20

> string.format("x = %x", 200) --> x = c8

> string.format("x = 0x%X", 200) --> x = 0xC8

> string.format("x = %f", 200) --> x = 200.000000

> tag, title = "h1", "a title"

> string.format("<%s>%s</%s>", tag, title, tag) --> <h1>a title</h1>

在百分号和字符之间，还可以添加一些选项，来控制格式化的细节，例如对数字来说可以控制整数和小数的位数：

print(string.format("pi = %.4f", math.pi)) --> pi = 3.1416

d = 5; m = 11; y = 1990

print(string.format("%02d/%02d/%04d", d, m, y)) --> 05/11/1990

第一个例子中，%.4f意思是保留四位小数。第二个例子，%02d意思是用2位数表示数字，不足的位数用0填充；如果是%2d的写法，就是用空格填充。具体的完整说明请参考C语言标准的printf函数，因为Lua底层实际上是调用C语言的标准库来实现这个复杂的功能。

所有string库函数的调用，都可以改用字符串对象加冒号的形式代替。例如，string.sub(s, i, j)可以改写为s:sub(i, j)；string.upper(s)可以改为s:upper()。（第21章面向对象编程中会详细介绍冒号的用法，）

字符串库还包含几种基于模式匹配的函数。比如string.find函数会在字符串中查找指定的模式：

> string.find("hello world", "wor") --> 7 9

> string.find("hello world", "war") --> nil

它返回模式在字符串中开始和结束的位置，如果没有找到则返回nil。函数string.gsub（Global SUBstitution，全局替换）会替换所有查找到的符合模式的部分，替换为另一个字符串。

> string.gsub("hello world", "l", ".") --> he..o wor.d 3

> string.gsub("hello world", "ll", "..") --> he..o world 1

> string.gsub("hello world", "a", ".") --> hello world 0

它的第二个返回值，指的是替换的次数。

在本书第10章模式匹配中，会详细介绍更多的关于模式匹配的知识。

### 13.4.6 Unicode

从Lua 5.3版本开始，Lua新增了一个小的库以支持UTF-8编码的字符串。其实在之前的版本中，Lua也已经提供了对UTF-8有效的支持。

在Unicode的网页应用中，UTF-8已经成为了终极解决方案。由于UTF-8编码对ASCII的兼容性，Lua也适合使用它。由于良好的兼容性，很多操作ASCII编码的字符串的技术也同样适用UTF-8编码，而不需要任何改动。

UTF-8编码使用不定数量的字节来表示一个Unicode字符。例如，字母A使用一个字节表示，内容为65，而希伯来文字符Aleph（Unicode码为1488），则使用两个字节表示，内容为215-144。在ASCII范围内的所有字符（即编码小于128的字符），UTF-8编码与ASCII相同。而对其它的字符，首字节范围为[194, 244]，后续字节的范围是[128,191]。进一步的，双字节字符的首字节范围是[194, 223]；三字节字符的首字节范围是[224,239]，四字节则是[240,244]。互不冲突。这种设计保证了任意字符不会产生多种可能的解释。比如任何小于128的字节都不会出现在一个多字节字符中，也就是说一个小于128的字节，一定是一个ASCII字符。

Lua字符串的操作对于UTF-8编码字符串来说“刚好”可以工作。因为Lua以8比特为准，所以它也可以正常读、写和存储任意UTF-8编码的字符串，与ASCII编码的字符串一样。字面量可以包含UTF-8的字符串（当然，如果你的编辑器支持UTF-8，你肯定也希望用UTF-8编码来编写代码）。连接操作符对于UTF-8字符串可以正常工作，字符串的比较操作（小于、小于等于等等）也可以应用于UTF-8字符串，大小顺序以它们的Unicode码为准。

由于Lua中和操作系统相关的库，以及I/O库主要是对操作系统接口的封装，所以它们是否支持UTF-8取决于具体的操作系统。在Linux平台上，可以使用UTF-8编码作为文件名，但是Windows就要使用UTF-16。要处理Windows上的Unicode文件名，需要借助其他库的帮助。

现在看看字符串库如何处理UTF-8编码的字符串。函数reserve，upper，lower，byte与char对UTF-8字符串不兼容，因为它们总是假设一个字符长度为一个字节。string.format与string.rep函数绝大部分情况可以正常工作，标识符’%c’除外，因为它也会假设一个字符是一个字节。string.len和string.sub可以正确处理UTF-8字符串，但是是以字节数为准（而不是字符数），很多时候这样处理恰好是我们想要的。

接下来看看新的utf8库。函数utf8.len会返回字符串中UTF-8字符的个数。而且它还会对字符串是否符合UTF-8标准进行检查，如果发现非法的字节，则返回false和发生错误的字节位置。

> utf8.len("résumé") --> 6

> utf8.len("ação") --> 4

> utf8.len("Månen") --> 5

> utf8.len("ab\x93") --> nil 3

（当然，要运行以上代码，需要一个支持UTF-8的终端。）

函数utf8.char和utf8.codepoint相当于是string.char和string.byte的UTF-8版本：

> utf8.char(114, 233, 115, 117, 109, 233) --> résumé

> utf8.codepoint("résumé", 6, 7) --> 109 233

注意最后一行代码的下标。大部分UTF-8的库函数是以字节数为准的。例如，调用string.codepoint(s, i, j) 中的i和j，都是以字节为准的下标。如果我们想要使用字符下标，函数utf8.offset可以将字符位置转换为字节位置：

> s = "Nähdään"

> utf8.codepoint(s, utf8.offset(s, 5)) --> 228

> utf8.char(228) --> ä

上面的例子使用utf8.offset获得了第5个字符的字节位置，并将它传给了codepoint函数。

与string库相同，utf8.offset的参数可以是负数，同样也是从后往前数：

> s = "ÃøÆËÐ"

> string.sub(s, utf8.offset(s, -2)) --> ËÐ

最后一个要介绍的函数是utf8.codes，用来方便地遍历utf8字符串的每一个字符：

for i, c in utf8.codes("Ação") do

print(i, c)

end

--> 1 65

--> 2 231

--> 4 227

--> 6 111

这种写法会根据字符来遍历字符串，每次可以得到字节位置和该字节位置的数值。在上面的例子中，只打印了这两个数据（在第18章，迭代器与通用for中会继续探讨迭代操作的细节）。

不幸的是，Lua无法再提供更多功能了。Unicode有很多古怪的细节问题，很难实际总结出一个抽象的方法适用于所有的语言。甚至“字符”这个概念的定义都是模糊的，因为存在“修饰字符”的情况。例如，字符é可以用两种方法表示，可以用单字符("\u{E9}") ，也可以用两个字符 ("e\u{301}")，即字母e接一个修饰字符。甚至基本概念例如“字母”，也会因语言不同而不同。由于这种复杂性，完整支持Unicode需要借助于很大的编码表，这与小巧的Lua是不兼容的。所以，如果要解决更复杂的编码问题，最好的方式是借助第三方库来解决。

### 13.4.7 练习

练习4.1 如何将这段XML代码嵌入到Lua代码中？

<![CDATA[

Hello world

]]>

练习4.2 写一个函数，功能是将一个字符串插入到另一个字符串的指定位置：

> insert("hello world", 1, "start: ") --> start: hello world

> insert("hello world", 7, "small ") --> hello small world

练习4.3 再写一个上面的函数，让它能够处理UTF-8编码的字符串：

> insert("ação", 5, "!") --> ação!

（注意位置的计算方式）

练习4.4 写一个函数，删除字符串中的一个片段；片段由起始位置和长度指定。

> remove("hello world", 7, 4) --> hello d

练习4.6 再写一个上面的函数，让它能够处理UTF-8编码的字符串：

> remove("ação", 2, 2) --> ao

（以上位置和长度都以codepoints计算）

练习4.7 写一个函数，判断给定字符是否是一个“回文”（对称的字符串）：

> ispali("step on no pets") --> true

> ispali("banana") --> false

练习4.8 在写一个上面的函数，让他能忽略标点符号和空格的不同。

练习4.9 再写一个上面的函数，让它能够处理UTF-8编码的字符串。

## 13.5 表

### 13.5.1 Lua表

在Lua中，表（Table）是主要的（实际上是唯一的）数据结构，表的机制非常强大。表用来表示数组、集合、记录等等各种数据结构，并以这种简易、通用、高效的方式统一起来。Lua使用表来表示程序包（Packages）与对象（Objects）。例如我们写math.sin，意思是math库中的sin函数，对Lua来说，其实就是取出math表的键“sin”对应的值。

Lua中的表是一种混合的数组。表不仅可以用数字下标访问，还可以用字符串和其它类型的数据作为下标（nil除外）。

Lua中的表既不是值也不是变量，它们是对象。如果你熟悉Java或Scheme中的数组，你应该能理解到我的意思——它会动态创建，仅能使用“引用”或者说“指针”访问它们。Lua永远不会隐式地拷贝或者创建表格对象。

创建表，也就是一个构造表达式，最简单的方式用{}即可：

> a = {} --> 创建表并将引用交给a

> k = "x"

> a[k] = 10 --> 添加新的入口（entry），key为”x”，value为10

> a[20] = "great" --> 新的入口，key为20，value为”great”

> a["x"] --> 10

> k = 20

> a[k] --> "great"

> a["x"] = a["x"] + 1 -- "x"对应的值自增1

> a["x"] --> 11

表总是“匿名”的。引用表的变量和表本身之间没有固定的关系：

> a = {}

> a["x"] = 10

> b = a --> b指向了与a相同的表

> b["x"] --> 10

> b["x"] = 20

> a["x"] --> 20

> a = nil -- 现在只有b指向那个表了

> b = nil -- 该表现在没有变量指向它了

当程序中不再有变量指向（引用）一个表，垃圾回收器（garbage collector）会在稍后删除该表并释放它占用的内存。

### 13.5.2 表的下标

每个表都可以保存不同类型的数据，且这些数据的下标也可以是不同类型。在添加新的入口时，表格容量也会在必要时自动增长。

> a = {} -- 空表

> -- 新增1000个入口

> for i = 1, 1000 do a[i] = i\*2 end

> a[9] --> 18

> a["x"] = 10

> a["x"] --> 10

> a["y"] --> nil

注意最后一行：与全局变量一样，当表格的入口没有初始化时，值为nil；且将某个入口赋值为nil就可以删除它。这不是一个巧合：Lua的全局变量也是保存在一个表中。（在第22章中我们会继续讨论这个问题——环境（The Environment）。）

要表示结构体时，我们使用名称来作为下标。Lua支持a.name的写法，作为a[“name”]的语法糖。我们用这种简单的方式重写之前的代码：

> a = {} -- 空表

> a.x = 10 -- 相当于a["x"] = 10

> a.x --> 10 -- 相当于 a["x"]

> a.y --> nil -- 相当于 a["y"]

在Lua中，以上两种形式是完全等价的，可以任意交换。在人们阅读代码时，两种方式通常代表不同的意图。用“点”的方式清晰地表明是把表格当作结构体在用，一般结构体具有固定的、实现定义好的一些字段。而用中括号加字符串的方式表示表格用字符串作为键，且这个键可能也是程序运行时才生成的。

初学者常犯的错误是混淆了a.x与a[x]。其实前者代表的是a[“x”]，也就是用字符串”x”作为下标。而后者是以x变量的值作为下标，举例来看：

> a = {}

> x = "y"

> a[x] = 10 -- 键”y”的值为10

> a[x] --> 10 -- 取下标为10的值

> a.x --> nil -- 取下标为字符串”x”的值

> a.y --> 10 -- 去下标为字符串”y”的值

由于我们可以用任意类型作为表的下标，所以当使用完全相等的下标时才会访问到同一个位置。尽管0和”0”都可以用来作为下标，这二者却是代表完全不同的入口。类似的，”+1”、”01”和”1”也是不同的字符串，代表不同的㐰。当你不确定下标的实际类型时，最好使用类型转换来确保这一点：

> i = 10; j = "10"; k = "+10"

> a = {}

> a[i] = "number key"

> a[j] = "string key"

> a[k] = "another string key"

> a[i] --> 这里i是number

> a[j] --> 字符串key

> a[k] --> 字符串key

> a[tonumber(j)] --> number key

> a[tonumber(k)] --> number key

如果不注意这一点，可能会在程序中引起相关的BUG。

整数与浮点数没有以上问题，由于2等于2.0，所以二者都可以指向表的同一个入口：

> a = {}

> a[2.0] = 10

> a[2.1] = 20

> a[2] --> 10

> a[2.1] --> 20

更特别的是，当浮点数作为下标使用时，如果浮点数可以转换为整数且不损失精度，那么就会被转换为整数。举个例子，当执行a[2.0] = 10的时候，会自动转换2.0为2。如果浮点数不能直接转换为整数（而不损失精度），那么就保持不变。

### 13.5.3 表的构造式

表的构造式就是创建并初始化表的表达式。它是Lua中非常独特的特性，也是Lua中最有用且灵活的机制之一。

最简单的构造式就是空的大括号。另外还可以用来初始化列表（数字下标数组）。例如，下面的表达式就将days[1]初始化为”Sunday”（注意Lua中数字下标从1开始而不是0），days[2]为”Monday”，以此类推：

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

print(days[4]) --> Wednesday

Lua还提供一种特殊的语法来初始化记录（字典）类型的表，例如：

a = {x = 10, y = 20}

以上写法等价于：

a = {}; a.x = 10; a.y = 20

第一种写法执行速度更快，因为在创建表时表的大小已经确定了。

无论采用哪种构造式来创建表，都可以在之后添加或者删除字段：

w = {x = 0, y = 0, label = "console"}

x = {math.sin(0), math.sin(1), math.sin(2)}

w[1] = "another field" -- 为表w添加入口1

x.f = w -- 为表x添加入口f

print(w["x"]) --> 0

print(w[1]) --> another field

print(x.f[1]) --> another field

w.x = nil -- 删除w的入口x

但是，如上面提到的，用合适的构造式来创建表更加高效，而且更简洁。

可以在同一个构造式中，混用字典形式和列表形式的初始化方法：

polyline = {color="blue",

thickness=2,

npoints=4,

{x=0, y=0}, -- polyline[1]

{x=-10, y=0}, -- polyline[2]

{x=-10, y=1}, -- polyline[3]

{x=0, y=1} -- polyline[4]

}

上面的例子同样表明了如何嵌套表来表示复杂的数据结构。polyline[i]的每个元素都又指向了一个字典形式的表：

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

print(polyline[4].y) --> 1

这两种构造式有一些问题。例如，无法构造具有负数下标的字段，也不能构造不符合变量命名规则的字符串下标。如果有这种需求，就要用到更普遍性的格式。也就是要明确写出字符串Key，并用中括号包围起来：

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]) --> ---

这种写法感觉更繁琐，但是也很好理解：列表形式与字典形式的初始化都是它的特例，下面的代码展示了等价的写法：

{x = 0, y = 0} <--> {["x"] = 0, ["y"] = 0}

{"r", "g", "b"} <--> {[1] = "r", [2] = "g", [3] = "b"}

为方便起见，最后一个元素后面可以带着逗号，也可以不带。带着逗号方便以后添加元素：

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

而且允许在最后元素之后带着逗号的写法，就不需要在很多情况下对最后一个元素进行特殊处理。

最后提一下，在构造式中总是可以用分号代替逗号。这个特性是Lua历史版本遗留下来的，我猜现在已经很少用到这个特性了。

### 13.5.4 数组，列表与序列

要表示一个典型的列表或数组，只要保证只使用数字下标即可。不必要也没有办法指定列表的长度，只需要按需要初始化即可：

-- 读取10行输入，放入表a

for i = 1, 10 do a[i] = io.read()

end

由于Lua的表可以用任意的下标，所以我们可以从任意整数开始（0、1甚至-1都可以）。但是，在Lua中约定俗成使用1作为下标的起点，Lua中很多库和函数都遵循这一约定。

通常，对列表进行计算时要知道它的长度。这个长度可能是一个常数或保存在某个地方。通常将长度保存在非数字Key的字段中；由于历史原因，有些程序使用Key ”n”来保存长度。但是通常长度是不记录的。记住没有初始化的字段一律为nil，所以可以用nil来代表列表的结尾。例如，很容易知道一个列表的长度是10，因为它的数字Key是1, 2, ..., 10。这种设计的前提是在列表中不能有“洞”，也就是说在数组中间出现了值为nil元素。这里称没有“洞”的列表为序列（Sequence）。

Lua对序列提供了取长度的运算符“#”。之前看到了，对字符串使用#运算符会得到字符串中的字节数。对表来说，它会计算序列的长度。例如，利用它可以打印前面的多行数据：

--print the lines, from 1 to #a

for i = 1, #a do

print(a[i]) end

使用长度运算符也可以方便修改序列：

a[#a + 1] = v --在末尾处添加v

长度运算符对有“洞”的列表来说是不可靠的。只有对没有“洞”的序列来说才能保证正确的结果。更准确地说，一个序列就是下标是正整数从1到n的表（注意任何值为nil的键，都相当于不存在）。特别的，一个不具有数字键的表相当于一个长度为0的序列。

Lua中，对带有“洞”的列表使用长度运算符，是争议最大的特性之一。多年以来已经有过很多提议，比如在对有洞列表取长度时抛出错误信息，或是让带洞的列表也可以取长度等等。但是，说起来容易做起来难。这个问题的本质是，由于列表实际上是一个字典，那么“长度”的概念就很模糊。例如以下代码生成的列表：

a = {}

a[1] = 1

a[2] = nil -- 这句实际不产生效果，因为下标2本来就是nil

a[3] = 1

a[4] = 1

从上面的代码很容易说列表长度为4，在下标2的地方有一个洞。但是下面这个例子呢？

a = {}

a[1] = 1

a[10000] = 1

应当把它当做长度为10000，有9998个洞吗？如果那样的话，试着这么做：

a[10000] = nil

那么这时列表长度是多少呢？因为仅删除了一个元素，长度应该是9999吗？或者仍然是10000，仅是多了一个洞呢？还是说长度应当变为1呢？

另一种常见的意见是让#运算符返回表中实际的元素个数。这种设计语义是明确的，但是不太有用，也不太符合直觉。仔细考虑上面讨论过的例子，想想如果这么设计是不是真的好用。

还有一种情况也会带来麻烦，就是nil出现在列表的末尾。下面的列表长度应该是多少呢？

a = {10, 20, 30, nil, nil}

记住，对Lua来说，值为nil的字段相当于不存在的字段。因此，上面的列表长度为3。

你可能会觉得nil位于列表的末尾是一种特殊的情形。其实有很多列表都是逐个加入元素形成的，所以在加入的过程中，它的末尾一直存在多个nil。

除开以上所讨论的内容，平时我们使用的大部分列表都是序列（例如文件中的一行不会是nil），且这种情况下使用取长度运算符都是安全的。如果真的需要处理带有洞的列表，那么就应该指定另一个地方保存列表的长度。

### 13.5.5 表的遍历方法

使用pairs迭代器，可以遍历表的所有键值对：

t = {10, print, x = 12, k = "hi"}

for k, v in pairs(t) do

print(k, v)

end

--> 1 10

--> k hi

--> 2 function: 0x420610

--> x 12

由于Lua实现表的机制原因，遍历表的顺序是未定义的。同样的程序有可能在遍历表时采用了不同的顺序。唯一能够确定的是，在遍历时每个元素出现且仅出现一次。

对列表来说，我们使用ipairs迭代器：

t = {10, print, 12, "hi"}

for k, v in ipairs(t) do

print(k, v)

end

--> 1 10

--> 2 function: 0x420610

--> 3 12

--> 4 hi

这种迭代方式中，Lua会保证遍历的顺序。

另一种确保顺序的写法，是采用数值的for循环：

t = {10, print, 12, "hi"}

for k = 1, #t do

print(k, t[k])

end

--> 1 10

--> 2 function: 0x420610

--> 3 12

--> 4 hi

### 13.5.6 安全访问字段

考虑以下情况：我们需要知道某个库中的某个函数是否存在。如果我们确定知道该库存在，那么就可以写if lib.foo then ...。否则，必须先判断库本身是否存在，if lib and lib.foo then ...。

当表格嵌套层次较深时，这种写法会很有问题，如下面的例子所示：

zip = company and company.director and

company.director.address and

company.director.address.zipcode

这种写法不仅繁琐，而且也很低效。它做了6次表格访问才得到了一次成功访问，而不是应有的3次。

一些编程语言，例如C#，提供了一种安全访问运算符（在C#中写作?.）专门解决这个问题。当我们写a?.b时，如果a是nil，则结果也是nil，而不会产生错误。使用这种操作符，我们可以把之前的例子改为下面的：

zip = company?.director?.address?.zipcode

如果路径中的任何一部分存在nil，则安全访问运算符会逐级产生nil直到最后返回nil。

Lua并不支持这种安全访问运算符，且它并不是必要的。Lua是非常精简的。而且这种运算符非常有争议，很多人争论——不是没有理由的——这种写法会导致粗心的编码。但是，我们可以稍微多写一点代码来实现这种安全访问。

如果执行a or {}，当a为nil时，结果是一个空的表。所以当执行(a or {}).b时，如果a是nil，则最终结果也是nil。采用这种方法，我们可以像这样重写前面的表达式：

zip = (((company or {}).director or {}).address or {}).zipcode

稍加修改，还能让它更简洁、稍微更快一些：

E = {} -- 这个表以后可被重复使用

...

zip = (((company or E).director or E).address or E).zipcode

确实，这种写法会比安全访问运算符繁琐得多。然而，每个名称依然只写了一次，在执行时它的访问次数依然能保持最小（在这个例子中是3次），且这种写法也不需要在语言中引入新的运算符。以个人观点看来，这是一种足够好的代替方案。

### 13.5.7 Table库

table库提供了一些有用的函数，用于操作列表和序列（作者注：你可以将它理解为“序列库”或“列表库”。为了兼容历史版本，我们保持了“table库”的名称）。

table.insert函数在指定的位置插入元素，之后的元素会向后移动。例如，t为列表{10, 20, 30}，在调用table.insert(t, 1, 15)之后，t的内容变为{15, 10, 20, 30}。另一种特殊用法非常常用——如果调用时不指定位置参数，那么元素会被添加到末尾，而不会造成元素移动。例如，下面的代码会从输入流一行一行读取并保存在列表的末尾。

t = {}

for line in io.lines() do

table.insert(t, line)

end

print(#t) --> (number of lines read)

函数table.remove删除并返回指定位置的元素，后面的元素会向前移动来补上空位。当不提供位置参数时，就会删除序列的最后一个元素。

使用这两个函数，就可以直接实现栈、队列与双端队列等数据结构了。可以通过t={}初始化这些数据结构。push操作等价于table.insert(t, x)，pop操作等价于table.remove(t)。table.insert(t, 1, x)就是在头部插入元素，table.remove(t, 1)也是删除头部的元素。最后两种操作并不高效，因为必须将所有元素前移或者后移。但是，由于table库是在C语言中实现这些函数的，所以这些操作的开销并不会过于太大，对于不大的数组来说，这个操作是足够好用的（例如几百个元素的数组）。

Lua5.3提供了一种更通用的函数来移动表中的元素，即table.move(a, f, e, t)，它会将表a中从f到e（包含f和e）的元素移动到t的位置。例如要在列表a的开头插入元素，方法如下：

table.move(a, 1, #a, 2)

a[1] = newElement

下面的代码会删除第一个元素：

table.move(a, 2, #a, 1)

a[#a] = nil

注意，在计算机领域，数组的移动操作通常都是将数据从一个地方拷贝到另一个地方。在最后的例子中，必须显式地删除最后一个元素。

table.move函数还有一个附加的可选参数：一个表。如果填写了这个参数，将会把第一个列表中的一段移动到第二个表中。例如调用table.move(a, 1, #a, 1, {})会得到一份a的拷贝（会拷贝每个元素到新的列表），而table.move(a, 1, #a, #b+1, b)会将a的所有元素加到b的最后。

### 13.5.8 练习

练习5.1 以下代码输出是什么？请给出解释。

sunday = "monday"; monday = "sunday"

t = {sunday = "monday", [sunday] = monday} print(t.sunday, t[sunday], t[t.sunday])

练习5.2 假设有如下代码：

a = {}; a.a = a

那么，a.a.a.a的值是什么？连续访问a得到的值一直是一样的吗？

现在，加上以下这行代码：

a.a.a.a = 3

现在a.a.a.a的值是什么？

练习5.3 假设你想要创建一个表，该表对于字符串映射了所有escape sequence到实际的字符。对于这个表，如何编写构造式？

练习5.4 可以将一个多项式anxn + an-1xn-1 + ... + a1x1 + a0 的系数表示为一个列表，形式如{a0, a1, ..., an}。

编写一个函数，参数为多项式系数（用一个表表示）以及x的值，返回多项式的计算结果。

练习5.5 重写上面的多项式计算函数，让它最多只用n次乘法和n次加法（没有幂运算），你能做到吗？

练习5.6 编写一个函数，测试一个表是否是一个合法的序列。

练习5.7 编写一个函数，将一个列表的所有元素，插入到另一个列表的指定位置。

练习5.8 table库还提供一个函数叫做table.concat，它接收一个字符串列表作为参数，并返回它们连接以后的结果：

print(table.concat({"hello", " ", "world"})) --> hello world

请用自己的方法编写一个同样功能的函数。

比较一下你自己编写的函数与table.concat的性能差异，可以用一个非常大的列表做测试，比如成百上千个字符串的列表（可以使用for循环来创建大的列表）。

注：该问题与C#的字符串连接问题本质相同。在C#中，我们用StringBuilder代替字符串加法，是基于同样的考虑。

## 13.6 面向对象编程

### 13.6.1 实现思路

Lua中的表具有多种功能。与“对象”相似，表可以保存状态，也具有独立的标识（self）；特别的，两个具有相同内容的对象（表）是不同的个体，而同一个对象在不同的时刻也可以有不同的数据。与对象相同，表也具有独立的生命周期，与谁创建它以及在什么地方创建它无关。

对象具有针对它自己的操作，表格也可以做到，如下所示：

Account = {balance = 0}

function Account.withdraw(v)

--Account.withdraw=function(v) ..... end

Account.balance = Account.balance - v

end

上面的代码定义了一个新的函数，并将函数保存在Account对象的withdraw字段中。然后可以像这样调用：

Account.withdraw(100.00)

这样的函数（function）已经几乎算是对象方法（method）了。但是，在函数中显式使用全局变量Account在实践中是一种非常糟糕的写法。首先，这个函数只能为这个特定的对象工作；其次，仅就此对象而言，该函数也只能在该对象保存于这个特定的Account变量时才能正常工作。如果我们修改了这个对象的名称，withdraw函数也就无法工作了：

a, Account = Account, nil

a.withdraw(100.00) -- ERROR!

这样的设计违背了“对象应具有独立的生命周期”的原则。

一种更加实用的做法，是针对接受操作的对象进行处理。因此，我们需要一个额外的参数用来指定被操作的对象。一般将它命名为self：

function Account.withdraw (self, v)

self.balance = self.balance - v

end

现在，当我们调用该方法时，需要指定操作的对象：

a1 = Account; Account = nil

...

a1.withdraw(a1, 100.00) -- OK

有了self参数，就可以将该方法用在不同的对象上：

a2 = {balance=0, withdraw = Account.withdraw}

...

a2.withdraw(a2, 260.00)

Self参数的使用是很多面向对象语言的核心概念。大部分面向对象（OO）的语言都会采用在语法上隐藏self的方式，因此不需要每次都显式写出self参数（但是也依然可以在对象方法中使用）。Lua也提供了隐藏self参数的写法——使用冒号。有了冒号，我们就可以重写前面的方法，写作a2:withdraw(260.00)，函数定义也可以修改如下：

function Account:withdraw (v)

self.balance = self.balance - v

end

冒号的作用就是添加一个额外的参数，并将它隐含的作为函数的第一个参数。冒号只是一种语法糖，尽管用起来比较方便，但是本质上并没有提供新的语法机制。我们可以不用冒号而用“点”来定义对象方法，并用冒号语法调用，也可以反过来。只要正确处理额外的self参数，就没有问题：

Account = {

balance=0,

withdraw = function (self, v)

self.balance = self.balance - v

end

}

function Account:deposit (v)

self.balance = self.balance + v

end

Account.deposit(Account, 200.00)

Account:withdraw(100.00)

### 13.6.2 类

至此，对象已经具有了独立标识符、独立的状态，可以定义针对它的操作。但是依然缺乏类型系统、继承与访问级别控制。让我们先聚焦于第一个问题：如何方便地创建一些同类的对象？比如说，创建一些账户对象。

大部分面向对象语言都提供了class的语法，用它来作为创建对象的模板。在这类语言中，每个对象都是class的实例。而Lua不提供class的概念，仅提供有一点类似的元表（metatables）概念，但元表并不能直接作为class来使用。作为替代，Lua提供了“基于原型”的语法，（JavaScript也采用了这种思路）。在“基于原型”的语言里，对象不属于“类（class）”，而相对的，每个对象都具有一个“原型”对象，原型也是一个普通的对象，在访问到其它对象的字段时，如果找不到，就去原型对象中查找。为了提供类似“class”的概念，我们只需要简单地创建一个对象作为其它对象的原型即可。其实，class或原型都是在为其它对象提供公共的字段和方法。

在Lua中使用原型用到了“继承”的思想，具体要用到\_\_index元方法。更实际一些，要让B成为A的原型，只需要像这样：

setmetatable(A, {\_\_index = B})

这样操作后，只要是A不具有的字段，都会到B里面去查。甚至可以把B当作A的class来看待。

让我们回到银行账户的例子。要创建其它类似Account的账户，只需要让其它账户继承Account，使用上面的\_\_index元方法。

local mt = {\_\_index = Account}

function Account.new(o)

o = o or {} -- 如果没有传参数则创建新表

setmetatable(o, mt)

return o

end

这样操作之后，我们访问新对象的方法会如何呢？

a = Account.new{balance = 0}

a:deposit(100.00)

当我们新建账户对象a，a就具有mt作为自己的元表（metatable）。当我们调用a:deposit(100.00)时，实际上是在执行a.deposit(a, 100.00)，记住冒号只是一个语法糖而已。但是这时a中并没有”deposit”，于是Lua就继续查找元表中的\_\_index。整个过程类似这样：

getmetatable(a).\_\_index.deposit(a, 100.00)

a的元表是mt，而mt.\_\_index就是Account。所以上面这句话等价于下面这句：

Account.deposit(a, 100.00)

于是，Lua调用了原始的deposit函数，将a作为self参数传递。所以，新的账户对象a继承了原来的Account对象中的deposit函数。同样，a也继承了所有Account中的其它字段和方法。

另外可以做两项小改进。首先，不需要为新的对象显式创建元表，作为替代，可以用Account表本身来作为元表。其次，可以直接将冒号用在new方法上。因此，new方法可以像这样改写：

function Account:new (o)

o = o or {}

self.\_\_index = self

setmetatable(o, self)

return o

end

现在，当我们调用Account:new()时，隐藏参数self就是Account表本身，然后我们让Account.\_\_index等于Account本身，并把Account设置为新对象o的元表。看起来我们并没有从冒号上得到太多好处，使用self的好处在接下来的“继承”用法中会更明显一些。

继承不仅用于方法，也对任何字段起作用。就像class不仅能定义方法，也可以定义常数、字段等等。比如Account中的字段balance默认值为0。所以新的account对象本身不包含balance，但会继承原型中的初始值：

b = Account:new()

print(b.balance) --> 0

调用b的deposit方法，其结果与下面的代码等价，因为self换成了b：

b.balance = b.balance + v

b.balance的初始值为0，以上代码还为b.balance赋了值。之后再访问b.balance就不会再触发\_\_index元方法，因为b已经有了自己的balance字段。

### 13.6.3 继承

由于“类”本身也是对象，所以类也能从其它类那里“借用”方法。这一想法让“继承”在Lua中很容易实现。

借用前面的Account类作为基类。

-- the Account class --

Account = {balance = 0}

function Account:new (o)

o = o or {}

self.\_\_index = self

setmetatable(o, self)

return o

end

function Account:deposit (v)

self.balance = self.balance + v

end

function Account:withdraw (v)

if v > self.balance then error "insufficient funds" end

self.balance = self.balance - v

end

从Account类继承一个派生类，名为SpecialAccount，允许用户提取超出余额（balance）的金额。首先，从基类创建一个新的、空的类，继承所有方法和字段：

SpecialAccount = Account:new()

到此为止SpecialAccount只是Account的一个实例。魔法开始了：

s = SpecialAccount:new{limit=1000.00}

SpecialAccount也继承了基类的new方法，与其它方法没有区别。但是这次，当new方法执行时，它的self参数引用的是SpecialAccount。也就是说，对象s的元表是SpecialAccount，元表的\_\_index字段也是SpecialAccount。然后，当执行s:deposit(100.00)时，Lua找不到s中的该方法名，就会去SpecialAccount中找，依然找不到。最终会找到基类Account中的方法deposit。

要想让SpecialAccount变得特别，只需要重写必要的方法即可。也就是说，只需要重写下面的方法：

function SpecialAccount:withdraw (v)

if v - self.balance >= self:getLimit() then error"insufficient funds"

end

self.balance = self.balance - v

end

function SpecialAccount:getLimit ()

return self.limit or 0

end

这样，当我们调用s:withdraw(200.00)时，Lua不会去Account中查找该方法，因为从SpecialAccount已经能找到了。由于s.limit是1000.00（透支额度，在创建s时已经设置过），取款操作会成功，账户s的余额变为了负数。

Lua的面向对象有着有趣的一面——不需要创建新的类，就能给对象定义特殊的行为。假如单独某个对象需要修改自身的逻辑，只需要针对该对象重写方法即可。例如，账户s希望自身的透支额度总是余额的10%，我们可以仅修改s这一个对象：

function s:getLimit ()

return self.balance \* 0.10

end

这样做以后，当调用s:withdraw(200.00)时，会调用SpecialAccount的withdraw方法，但withdraw方法会调用getLimit方法，这时s的getLimit方法将被调用。

### 13.6.4 练习

练习6.1 编写本节的Account实例，理解Lua是如何用简单的表实现对象的。

练习6.2 实现一个“栈”类，它应具有push、pop、top和isempty方法。

练习6.3 编写一个简单的回合制对战程序，理解面向对象的使用方法。

1. 先写一个不使用原型继承的版本
2. 适当使用继承，实现多态特性

## 13.7 模块与包

### 13.7.1 Lua用表实现模块

通常，Lua语言不会硬性设置规则。相反，Lua语言提供的是足够灵活的机制供不同的开发者实现适合自己的规则。然而，这种方法对于模块（module）而言并不是特别适用。模块系统（module system）的主要目标之一就是允许不同的人共享代码，缺乏公共规则就无法实现这样的共享。

Lua语言从5.1版本开始为模块和包（package：模块的集合）定义了一系列的规则。这些规则不需要从语言中引人额外的功能，程序员可以使用目前为止我们学习到的机制实现这些规则。程序员也可以自由地使用不同的策略。当然，不同的实现可能会导致程序不能使用外部模块，或者模块不能被外部程序使用。

从用户观点来看，一个模块（module）就是一些代码（用Lua或C编写的），这些代码可以通过函数require加载，然后创建并返回一个表。这个表就像是某种命名空间，内容是模块中导出的东西，比如函数和常量。

例如，所有的标准库都是模块。我们可以按照如下的方法使用数学库：

local m = require "math" --注：参数为一个字符串时可省略括号

print(m.sin(3.14)) - - > 0.0015926529164868

独立解释器会使用跟如下代码等价的方式提前加载所有标准库 ：

math = require “math”

string = require “string”

这种提前加载使得我们可以不用费劲地编写代码来加载模块math就可以直接使用函数math.sin。

使用表来实现模块的显著优点之一，是可以像操作普通表那样操作模块，并且能利用Lua表的特性实现额外的功能。在大多数语言中，模块不是第一类值（即它们不能被保存在变量中，也不能被当作参数传递给函数等），所以那些语言需要为模块实现一套专门的机制。而在Lua语言中，我们则可以轻易地实现这些功能。

例如，用户调用模块中的函数就有几种方法。其中常见的方法是：

local mod = require "mod ”

mod . foo ()

用户可以为模块设置一个局部名称：

local m = require “mod”

m. foo ()

也可以为个别函数提供不同的名称：

local m = require “mod”

local f = m.foo

f()

还可以只引入特定的函数：

local f = require “mod”.foo

--这里省略了括号实际是：(require(“mod”)).foo

f()

上述这些方法的好处是无须语言的特别支持，它们使用的都是语言已经提供的功能。

### 13.7.2 require函数

尽管函数require也只是一个没什么特殊之处的普通函数，但在Lua语言的模块实现中扮演着核心角色。要加载模块时，只需要简单地调用这个函数，然后传入模块名作为参数。请记住，当函数的参数只有一个字符串常量时括号是可以省略的，而且一般在使用require时按照惯例也会省略括号。不过尽管如此，下面的这些用法也都是正确的：

local m = require “math”

local modname = “math”

local m = require(modname)

函数require尝试对模块的定义做最小的假设。对于该函数来说，一个模块可以是定义了一些变量（比如函数或者包含函数的表）的代码。典型地，这些代码返回一个由模块中函数组成的表。不过，由于这个动作是由模块的代码而不是由函数require完成的，所以某些模块可能会选择返回其他的值或者甚至引发副作用（例如，模块创建了全局变量）。

首先，函数require在表package.loaded中检查模块是否已被加载。如果模块已经被加载，函数require就返回相应的值。因此，一旦一个模块被加载过，后续的对于同一模块的所有require调用都将返回同一个值，而不会再运行任何代码。

如果模块尚未加载，那么函数require则搜索具有指定模块名的Lua文件（搜索路径由变量package.path指定，我们会在后续对其进行讨论）。如果函数require找到了相应的文件，那么就用函数loadfile将其进行加载，结果是一个我们称之为加载器（loader）的函数（加载器就是一个被调用时加载模块的函数）。

如果函数require找不到指定模块名的Lua文件，那么它就搜索相应名称的C标准库（在这种情况下，搜索路径由变量package.cpath指定）。如果找到了一个C标准库，则使用底层函数package.loadlib进行加载，这个底层函数会查找名为luaopen\_*modname*的函数。在这种情况下，加载函数就是loadlib的执行结果，也就是一个被表示为Lua函数的C语言函数luaopen\_*modname*。

不管模块是在Lua文件还是C标准库中找到的，函数require此时都具有了用于加载

它的加载函数。为了最终加载模块，函数require带着两个参数调用加载函数：模块名和加

载函数所在文件的名称（大多数模块会忽略这两个参数）。如果加载函数有返回值，那么函

数require会返回这个值，然后将其保存在表package.loaded中，以便于将来在加载同一

模块时返回相同的值。如果加载函数没有返回值且表中的package.loaded[@rep{modname}] 为空，函数require就假设模块的返回值是true。如果没有这种补偿，那么后续调用函数require时将会重复加载模块。

要强制函数require加载同一模块两次，可以先将模块从package.loaded中删除：

package.loaded.*modname* = nil

下一次再加载这个模块时，函数require就会重新加载模块。

对于函数require来说，一个常见的抱怨是它不能给待加载的模块传递参数。例如，数

学模块可以对角度和弧度的选择增加一个选项：

-- 错误的代码

localmath = require (”math”, ”degree")

这里的问题在于，函数require的主要目的之一就是避免重复加载模块。一旦一个模块被加载，该模块就会在后续所有调用require的程序部分被复用。这样，不同参数的同名模块之间就会产生冲突。如果读者真的需要具有参数的模块，那么最好使用一个显式的函数来设置参数，比如：

localmod = require ”mod”

mod.init(0,0)

如果加载函数返回的是模块本身，那么还可以写成：

local mod = require ”mod”.init(0, 0)

请记住，模块在任何情况下只加载一次；至于如何处理冲突的加载，取决于模块自己。

**模块重命名**

通常，我们通过模块本来的名称来使用它们，但有时，我们也需要将一个模块改名以避

免命名冲突。一种典型的情况就是，出于测试的目的而需要加载同一模块的不同版本。对于

一个Lua语言模块来说，其内部的名称并不要求是固定的，因此通常修改.lua文件的文件名

就够了。不过，我们却无法修改C标准库的二进制目标代码中luaopen\_\* 函数的名称。为了进行这种重命名，函数require运用了一个连字符（即减号“-”）的技巧：如果一个模块名中包含连字符，那么函数require就会用连字符之前的内容来创建luaopen\_\* 函数的名称。 例如，如果一个模块的名称为mod-v3.4，那么函数require会认为该模块的加载函数应该是luaopen\_mod而不是luaopen\_mod-v3.4（这也不是有效的C语言函数名）。因此，如果需要使用两个名称均为mod的模块（或相同模块的两个不同版本），那么可以对其中的一个进行重命名，如mod-v1。当调用m1=require"mod-v1”时，函数require会找到改名后的文件mod-v1并将其中原名为luaopen\_mod的函数作为加载函数。

**搜索路径**

在搜索一个Lua文件时，函数require使用的路径与典型的路径略有不同。典型的路径

是很多目录组成的列表，并在其中搜索指定的文件。不过，ISO C（Lua语言依赖的抽象平台）并没有目录的概念。所以，函数require使用的路径是一组模板（template），其中的每项都指定了将模块名（函数require的参数）转换为文件名的方式。更准确地说，这种路径中的每一个模板都是一个包含可选问号的文件名。对于每个模板，函数require会用模块名来替换每一个问号，然后检查结果是否存在对应的文件；如果不存在，则尝试下一个模板。路径中的模板以在大多数操作系统中很少被用于文件名的分号隔开。例如，考虑如下路径：

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

在使用这个路径时，调用require “sql” 将尝试打开如下的Lua文件：

sql

sql.lua

c:\windows\sql

/usr/local/lua/sql/sql.lua

函数require只处理分号（作为分隔符）和问号，所有其他的部分（包括目录分隔符和文件扩展名）则由路径自己定义。

函数require用于搜索Lua文件的路径是变量package.path的当前值。当package模块被初始化后，它就把变量package.path设置成环境变量LUA\_PATH\_5\_3的值。如果这个环境变量没有被定义，那么Lua语言则尝试另一个环境变量LUA\_PATH。如果这两个环境变量都没有被定义，那么Lua语言则使用一个编译时定义的默认路径。在使用一个环境变量的值时，Lua语言会将其中所有的“;;”替换成默认路径。例如，如果将LUA\_PATH\_5\_3设为“mydir/?.lua;;”，那么最终路径就会是模板“mydir/?.lua;;”后跟默认路径。

搜索C标准库的路径的逻辑与此相同，只不过C标准库的路径来自变量package.cpath而不是package.path。类似地，这个变量的初始值也来自环境变量LUA\_CPATH\_5\_3或LUA\_CPATH。在POSIX系统中这个路径的典型值形如：

./?.so;/usr/local/lib/lua/5.21?.so

请注意定义文件扩展名的路径。在上例中，所有模板使用的都是.so，而在Windows操作系统中一般是：

.\?.dll;C:\ProgramFiles\Lua502\dll\?.dll

函数package.searchpath中实现了搜索库的所有规则，该函数的参数包括模块名和路径，然后遵循上述规则来搜索文件。函数package.searchpath要么返回第一个存在的文件的文件名，要么返回nil外加描述所有文件都无法成功打开的错误信息，如下：

> path=”.\\?.dll;C:\\Program Files\\Lua502\\dll\\?.dll”

> print(package.searchpath（”X”，path))

nil

nofile '.\X.dll'

nofile 'C:\Program Files\Lua502\dll\X.dll'

作为一个有趣的练习，我们在示例8.1中实现了一个与函数package.searchpath类似的函数。

示例7.1，实验版的package.searchpath：

function search(modname, path)

modname = string.gsub(modname, “”）

localmsg = {}

for c in string.gmatch(path, ”[^;]+”) do

local fname = string.gsub(c, ”?”, modname)

local f = io.open(fname)

if f then

f:close()

return fname

else

msg[#msg+1] = string.farmat(”\n\tno file ’%s’”, fname);

end

end

return nil, table.concat(msg) -- 没找到

end

上述函数首先替换目录分隔符，在本例中即把所有的点换成斜杠（我们会在后续看到模块名中的点具有特殊含义）。之后，该函数遍历路径中的所有组成部分，也就是每一个不含分号的最长匹配。对于每一个组成部分，该函数使用模块名来替换问号得到最终的文件名，然后检查相应的文件是否存在。如果存在，该函数关闭这个文件，然后返回文件的名称；否则，该函数保存失败的文件名用于可能的错误提示（请注意字符串缓冲区在避免创建无用的长字符串时的作用）。如果一个文件都找不到，该函数则返回nil及最终的错误信息。

**搜索器**

在现实中，函数require比此前描述过的稍微复杂一点。搜索Lua文件和C标准库的方式只是更加通用的搜索器（searcher）的两个实例。一个搜索器是一个以模块名为参数，以对应模块的加载器或nil（如果找不到加载器）为返回值的简单函数。

数组package.searchers列出了函数require使用的所有搜索器。在寻找模块时，函数require传入模块名并调用列表中的每一个搜索器直到它们其中的一个找到了指定模块的加载器。如果所有搜索器都被调用完后还找不到，那么函数require就抛出一个异常。

用一个列表来驱动对一个模块的搜索给函数require提供了极大的灵活性。例如，如果想保存被压缩在zip文件中的模块，只需要提供一个合适的搜索器（函数），然后把它增加到该列表中。在默认配置中，我们此前学习过的用于搜索Lua文件和C标准库的搜索器排在列表的第二、三位，在它们之前是预加载搜索器。

预加载（preload）搜索器使得我们能够为要加载的模块定义任意的加载函数。预加载搜索器使用一个名为package.preload的表来映射模块名称和加载函数。当搜索指定的模块名时，该搜索器只是简单地在表中搜索指定的名称。如果它找到了对应的函数，那么就将该函数作为相应模块的加载函数返回；否则，则返回nil。预加载搜索器为处理非标场景提供了一种通用的方式。例如，一个静态链接到Lua中的C标准库可以将其luaopen\_ 函数注册到表preload中，这样luaopen\_ 函数只有当用户加载这个模块时才会被调用。用这种方式，程序不会为没有用到的模块浪费资源。

默认的package.searchers中的第4个函数只与子模块有关，我们会在13.7.4节对其进行介绍。

### 13.7.3 编写模块的示范

在Lua语言中创建模块的最简单方法是，创建一个表并将所有需要导出的函数放入其

中，最后返回这个表。下面的示例演示了这种方法。

示例7.2：一个用于复数的简单模块

local M = {} -- 模块名称M

--创建一个新的复数

local function new(r, i)

return {r=r,i=i}

end

M.new = new --把函数”new”加到模块中

-- 常数 i

M.i = new(0, 1) -- M.i 是一个表 {r=0, i=1}

function M.add(c1,c2)

return new(c1.r+c2.r, c1.i+c2.i)

end

function M.sub(cl,c2)

return new(cl.r-c2.r, cl.i-c2.i)

end

function M.mul(c1,c2)

return new(cl.r\*c2.r-c1.i\*c2.i, cl.r\*c2.i + c1.i\*c2.r)

end

local function inv(c) -- 模块的私有函数

local n = c.r^2 + c.i^2

return new(c.r/n, - c.i/n)

end

function M.div(c1, c2)

return M.mul(c1, inv(c2))

end

function M.tostring(c)

return string.format("(%g, %g)", c.r, c.i)

end

return M --返回M，重要

请注意我们是如何通过简单地把new和inv声明为局部变量而使它们成为代码段的私

有函数（private function）的。

有些人不喜欢最后的返回语句。一种将其省略的方式是直接把模块对应的表放到package.loaded中：

local M = {}

package.loaded[...] = M -- 与之前相同，但没有return语句

请注意，函数require会把模块的名称作为第一个参数传给加载函数。因此，表索引中的可变长参数表达式，...其实就是模块名。在这一赋值语句后，我们就不再需要在模块的最后返回了：如果一个模块没有返回值，那么函数require会返回package.loaded[modname]的当前值（如果不是nil的话）。不过，笔者认为在模块的最后加上return语句更清晰。如果我们忘了return语句，那么在测试模块的时候很容易就会发现问题。

另一种编写模块的方法是把所有的函数定义为局部变量，然后在最后构造返回的表。

示例7.3，使用导出表的模块：

local function new(r, i) return {r＝r，i=i} end

local i = complex.new（白，1)

-- 与之前一样的函数定义，略……

return{

new = new, i = i, add = add, sub = sub,

mul = mul, div = div, tostring = tostring,

}

这种方式的优点在于，无须在每一个标识符前增加前缀M.或类似的东西。通过显式的

导出表，我们能够以与在模块中相同的方式定义和使用导出和内部函数。这种方式的缺点在

于，导出表位于模块最后而不是最前面（把前面的话当作简略文档的话更有用），而且由于

必须把每个名字都写两遍，所以导出表有点冗余（这一缺点其实可能会变成优点，因为这允

许函数在模块内和模块外具有不同的名称，不过程序很少会用到）。

不管怎样，无论怎样定义模块，用户都能用标准的方法使用模块：

local cpx = require "complex”

print(cpx.tostring(cpx.add(cpx.new(3,4),cpx.i)))

-- 结果为(3, 5)

后续，我们会看到如何使用诸如元表和环境之类的高级Lua语言功能来编写模块。不过，

除了发现由于失误而定义的全局变量时有一个技巧外，笔者在编写模块时都是用基本功能。

### 13.7.4 子模块与包

Lua支持具有层次结构的模块名，通过点来分隔名称中的层次。例如，一个名为mod.sub的模块是模块mod的一个子模块（submodule）。一个包（package）是一棵由模块组成的完整的树，它是Lua语言中用于发行程序的单位。

当加载一个名为mod.sub的模块时，函数require依次使用原始的模块名”mod.sub”作为键来查询表package.loaded和表package.preload。这里，模块名中的点像模块名中的其他字符一样，没有特殊含义。

然而，当搜索一个定义子模块的文件时，函数require会将点转换为另一个字符，通常就是操作系统的目录分隔符（例如，POSIX操作系统的斜杠或Windows操作系统的反斜杠）。转换之后，函数require会像搜索其他名称一样搜索这个名称。例如，假设目录分隔符是斜杠并且有如下路径，调用require”a.b”会尝试打开以下文件：

./a/b.lua

/usr/local/lua/a/b.lua

/usr/local/lua/a/b/init.lua

这种行为使得一个包中的所有模块能够放到一个目录中。例如，一个具有模块p、p.a和p.b的包对应的文件可以分别是p/init.lua、p/a.lua和p/b.lua，目录p又位于其他合适的目录中。Lua语言使用的目录分隔符是编译时配置的，可以是任意的字符串（请记住，Lua并不知道目录的存在）。例如，没有目录层次的系统可以使用下画线作为“目录分隔符”，因此调用require "a.b”会搜索’文件a\_b.lua。

C语言中的名称不能包含点，因此一个用C语言编写的子模块a.b无法导出函数luaopen\_a.b。这时，函数require会将点转换为其他字符，即下画线。因此，一个名为a.b的C标准库应将其加载函数命名为luaopen\_a\_b。

作为一种额外的机制，函数require在加载C语言编写的子模块时还有另外一个搜索器。当该函数找不到子模块对应的Lua文件或C文件时，它会再次搜索C文件所在的路径，不过这次将搜索包的名称。例如，如果一个程序要加载子模块a.b.c，搜索器会搜索文件a。如果找到了C标准库a，那么函数require就会在该库中搜索对应的加载函数luaopen\_a\_b\_c。这种机制允许一个发行包将几个子模块组织为一个C标准库，每个子模块有各自的加载函数。从Lua语言的视角看，同一个包中的子模块没有显式的关联。加载一个模块并不会自动加载它的任何子模块。同样，加载子模块也不会自动地加载其父模块。当然，只要包的实现者愿意，也可以创造这种关联。例如，一个特定的模块可能一开始就显式地加载它的一个或全部子模块。

### 13.7.5 练习

练习7-1：将列表或栈重写为恰当的模块。

练习7-2：将面向对象的回合制对战程序分模块重写，例如把角色原型的定义单独放在一个模块中。

练习7-3：总结Lua表的各种功能。

提示：列表，字典，面向对象，模块和包，作为元表扩展表的功能……

## 13.8 闭包

### 13.8.1 函数与闭包

Lua中的函数，是第一类对象（first-class value），且具有合理设计的词法域。

“第一类对象”指的是什么呢？它指的是，在Lua中，函数也是一种值，具有和普通的数字、字符串同等的地位。程序可以讲函数对象保存在变量中（全局变量或局部变量都可以）、保存到表中、用参数传递函数对象或者让函数返回一个函数，都是可以的。

而具有“词法域”是什么意思呢？它的意思是，一个函数可以访问包含了该函数的函数中的变量（也就是嵌套了该函数的外层函数的变量），同时这也说明了Lua支持lambda表达式。

这两个设计加在一起，为Lua语言提供了极大的便利性。例如，一个程序可以重新定义一个函数以添加新的功能，或是在运行不可信代码时清除一个函数（比如从网络上接收的代码，具有一定风险）。更重要的是，这些功能允许我们在Lua中使用强大的函数式编程技术。即便你对函数式编程没有很大兴趣，学习一点函数式编程的技术依然是十分值得的，因为这些技术确实能让你的程序变得更简洁。

### 13.8.2 Lua函数是第一类对象

前面说过，Lua中的函数是第一类对象。下面的例子是很好的解释：

a = {p = print} --> 'a.p' refers to the 'print' function

a.p("Hello World") --> Hello World

print = math.sin --> 'print' now refers to the sine function

a.p(print(1)) --> 0.8414709848079

math.sin = a.p --> 'sin' now refers to the print function

math.sin(10, 20) --> 10 20

如果函数也是一种值，那么用什么语法创建函数对象呢？实际上，通常定义函数是像这样写：

function foo (x) return 2\*x end

其实它是一种语法糖，等价于以下代码：

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

右侧的 (function (x) 函数体 end)实际上是函数的构造式，就像{}是表的构造式一样。也就是说，我们常用的定义函数的语法，其实是创建了一个函数对象，并将它赋给一个变量。

注意到，Lua中的所有函数都是匿名的。就和所有其它的值一样，它们都没有名字。我们平时说“函数名称”的时候，例如print，其实本质上是指向了该函数的变量的名字。尽管一般来说我们都会将函数对象赋值给全局变量，相当于给它们起了名字，仍然有很多情况下会使用匿名函数，接下来看一些例子。

Table库函数中有一个常用函数table.sort，它接收一个表格并对它进行排序。这样一个函数必须兼容无数种排序的情况：升序、降序、按数值比较、按字母顺序比较、按Key比较等等。不可能预先提供好所有排序的要求，相对的，sort函数提供了一个特殊的可选参数：一个比较函数，该函数接受两个元素，并返回前者是否应当排在后者前面。例如，假设有这样一个表：

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"},

}

如果我们要按照name排序，按字母顺序反向排列，只需要这样写：

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

可以看出，在这里使用匿名函数多么方便。

一个函数使用另一个函数作为参数，比如sort，被称为“高阶函数”（higher-order function）。高阶函数是一种很强大的语言机制，匿名函数的使用带来了巨大的便利性。要注意的是，高阶函数并没有高于一般函数的特权，它是Lua将函数作为第一类对象的自然结果。

为更好地展示高阶函数的用法，我们可以定义一个高阶函数derivative用来求导数。这里用一种非正式的方法计算导数，函数f的导数是 f'(x) = (f(x + d) - f(x)) / d ，其中d是一个很小的值。代码实现如下：

function derivative (f, delta)

delta = delta or 1e-4

return function (x)

return (f(x + delta) - f(x))/delta

end

end

给定一个函数f，调用derivative(f)就会返回它的导数函数（近似的）：

c = derivative(math.sin)

print(math.cos(5.2), c(5.2))

--> 0.46851667130038 0.46856084325086

print(math.cos(10), c(10))

--> -0.83907152907645 -0.83904432662041

注：sin(x)的导数是cos(x)，以上代码验证了derivative函数的正确性。

### 13.8.3 非全局函数

函数是第一类对象这一设计，自然产生了一个结果：不仅可以将函数保存在全局变量中，也可以保存在表的字段中或是局部变量中。

我们已经见过了很多将函数保存在表中的例子，大部分Lua的库函数都使用了这种方式（比如io.read，math.sin等）。要创建类似的函数，只需要灵活运用我们已经学过的语法即可：

Lib = {}

Lib.foo = function (x,y) return x + y end

Lib.goo = function (x,y) return x - y end

print(Lib.foo(2, 3), Lib.goo(2, 3)) --> 5 -1

也可以使用表的构造式来写：

Lib = {

foo = function (x,y) return x + y end,

goo = function (x,y) return x - y end

}

甚至还可以用定义function的语法，达到同样的效果：

Lib = {}

function Lib.foo (x,y) return x + y end

function Lib.goo (x,y) return x - y end

如21章说过的，Lua的面向对象设计中，将函数放在表格中是一个非常关键的思路。

当我们用局部变量指向一个函数的时候，就得到了一个局部函数，也就是一个限定在特定范围内的函数。这种函数的定义对设计程序包（package）时非常有用，因为Lua将每个代码块当作函数看待，每个代码块都可以声明局部的函数，而这个局部函数只在代码块内可见。词法域确保了本代码块内的函数可以使用这些局部函数。

Lua也支持定义局部函数的语法糖：

local function f (params)

body

end

在定义递归的局部函数时会遇到一个小问题，考虑如下代码：

local fact = function(n)

if n == 0 then return 1

else return n\*fact(n-1) -- buggy

end

end

当Lua编译到fact(n-1)时，局部变量fact还未被赋值。因此这时会去查找全局名称fact，而不是这里的局部变量fact。只要稍加改动，提前定义local fact就可以解决这个问题：

local fact

fact = function (n)

if n == 0 then return 1

else return n\*fact(n-1)

end

end

现在函数体里面的fact已经正确指代了局部变量。函数定义时fact的值是否已经存在不重要，因为当函数初次执行的时候，fact已经被正确赋值了。

所以在使用函数定义的语法糖时，也会考虑到上述问题。所以以下代码：

local function foo (params) body end

等价为如下代码：

local foo; foo = function (params) body end

所以，在用这种语法定义递归函数时大可放心。

这种技巧在定义间接递归的函数时又会遇到问题。在定义间接递归的函数时，就必须显式地提前定义局部变量：

local f -- 前置变量声明

local function g()

-- some code...

f()

-- some code...

end

function f()

-- some code...

g()

-- some code...

end

要注意的是，最后定义函数f时，千万不要加local，如果加上local又会重新定义变量f覆盖前面的f，导致g函数中的f还是nil。

### 13.8.4 词法域

当我们编写一个嵌套了另一个函数的函数时，内层函数可以访问外层函数的局部变量。这种特性称之为“词法域（lexical scoping）”。这种对外层的可见性看似是理所应当的，但实际上并不是。词法域再加上嵌套的一类函数对象，会为语言带来非常强大的功能，但是很多语言并不支持这二者的结合。

我们从一个简单的例子开始。假设我们有一个学生名称的列表，以及一个从名称到分数的查询表，我们希望以分数为准给名称列表排序，分数高的名称排在前面。可以编写如下代码：

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)

将上面的代码改写为函数：

function sortbygrade (names, grades)

table.sort(names, function (n1, n2)

return grades[n1] > grades[n2] --> compare the grades

end)

end

有趣的是，最内层的匿名函数用到了grades变量，而grades变量是外层函数sortbygrade的参数。在这个匿名函数中，grades变量既不是全局变量，又不是局部变量，而是被称之为非局部的变量（non-local variable）。由于历史原因，非局部变量在Lua中称之为upvalue。

这一点为什么能带来有趣的结果呢？因为函数作为第一类对象，可以“逃出”原本的作用域。考虑如下代码：

function newCounter()

local count = 0

return function () -- 匿名函数

count = count + 1

return count

end

End

c1 = newCounter()

print(c1()) --> 1

print(c1()) --> 2

上述代码中，匿名函数引用了外层非局部变量count，用于保存计数。但是当我们调用从newCounter返回的匿名函数时，看起来count变量逃出了newCounter的内部，因为newCounter的调用已经结束了，而count依然在计数。Lua可以正确处理这种情况，因为Lua实现了闭包（closure）的概念。简单来说，闭包就是一个函数，且带着所有相关的非局部变量。如果再次调用newCounter，就会产生一个新的变量count以及一个新的闭包，这个闭包内置了新的变量：

c2 = newCounter()

print(c2()) --> 1

print(c1()) --> 3

print(c2()) --> 2

所以，c1和c2是不同的闭包。二者都是用同样的函数生成的，但是分别具有独立的变量count。

从技术实现的角度看，Lua中真正的对象是闭包，而不是函数。函数本身只是闭包的原型。尽管如此，我们还是继续使用术语“函数”来指代闭包，一般不会带来歧义。

闭包在很多情况下都提供了一种有效的工具。如前所见，在使用sort函数参数时就会用到。闭包在用函数创建函数的情况下也很有用，比如前面newCounter的例子和导数函数的例子。闭包机制允许Lua使用很多函数式编程特有的理念和技术。闭包用于回调函数也是很有效的，一个典型的例子是用于在GUI程序中创建按钮。每个按钮都保存了一个回调函数，当用户按下按钮时就会触发该函数，每个按钮在按下时的处理都略有不同。

比方说，模拟计算器的程序有10个数字键，每个键对应一个数字。可以用函数创建这些按钮的回调函数：

function digitButton(digit)

return Button{ label = tostring(digit),

action = function ()

add\_to\_display(digit)

end

}

end

上面的例子中，Button函数是一个工具函数，用来创建新的按钮。label是按钮上面的标签，action是当按钮按下的回调函数。回调函数可以在按钮按下之后很长时间才执行，但依然可以随时访问digit变量。

闭包还可以用于更多不同的场合。因为函数只是保存在通常的变量之中，所以可以随时重新定义Lua中的函数，清除或者改写都可以。这也是说Lua非常灵活的原因之一。当我们重新定义一个函数时，经常需要借用原本的函数。比如我们想要重新定义sin函数，改为角度单位而不是弧度，那么新的函数会先把角度参数转为弧度，再调用原来的sin函数，代码可以像这样写：

local oldSin = math.sin

math.sin = function (x)

return oldSin(x \* (math.pi / 180))

end

稍微写得更清晰一些：

do

local oldSin = math.sin

local k = math.pi / 180

math.sin = function (x)

return oldSin(x \* k)

end

end

下面的写法使用do代码块以限制oldSin变量的作用域，因此oldSin变量不会与外面的变量冲突，oldSin仅在本代码块中可见，也只能在调用新的sin函数时用到。

我们可以使用这种技术来创建安全的私有环境，也称之为沙盒。沙盒环境在运行不可信代码时非常有用，比如运行从不可信的服务器上下载的代码时。例如为了防止一段代码任意修改本地的文件，我们可以在局部改写io.open函数：

do

local oldOpen = io.open

local access\_OK = function (filename, mode)

check access

end

io.open = function (filename, mode)

if access\_OK(filename, mode) then

return oldOpen(filename, mode)

else

return nil, "access denied"

end

end

end

这段代码的意义在于，经过重新定义之后，其它代码再也无法调用原始版本的io.open函数，只能通过新版函数实现，而新版函数已经加上了权限限制。原始的不安全版本的io.open函数以非局部变量的形式保存在闭包之中，从外部不可访问。

Lua采取了这种巧妙的机制，提供多种元方法以便用户根据安全需要定制解决方案，而不是试图提供一个万能的解决方案。（真实的沙盒机制不仅仅要保护文件系统，在未来的“沙盒化”章节中会继续讨论这个问题。）

### 13.8.5 浅尝函数式编程

为了给出一个更加实际的函数式编程的例子，本节我们将开发一个简单的几何图形系统。设计目标是实现一个能够代表几何图形的系统，一个图形由一系列顶点组成。我们希望能用它表示所有类型的形状，且能用多种方式修改形状（旋转、平移、取并集等等）。

要实现这套系统，首先要找到合适的数据结构来表示图形。可以采用面向对象的方式来创建一系列形状。或者我们可以用更高级的抽象方式，直接用一个特征函数来表示一种形状（集合A的特征函数fA是这样一种函数：对任意的x，仅当x属于A时，fA(x)返回true）。也就是说，我们用一个函数来代表一类图形，仅当一个点位于该图形内部时，返回true。

比如，下面的函数定义了一个圆盘（圆形区域），圆心位于（1.0, 3.0），半径为4.5：

function disk1 (x, y)

return (x - 1.0)^2 + (y - 3.0)^2 <= 4.5^2

end

这里可以采用高阶函数与词法域，定义一个“圆盘工厂”，只要给定圆心和半径，就可以创建出对应的圆盘函数：

function disk (cx, cy, r)

return function (x, y)

return (x - cx)^2 + (y - cy)^2 <= r^2

end

end

只要调用disk(1.0, 3.0, 4.5)，就得到了等价于前面的disk1的函数。

下面的函数创建了一个延坐标轴方向摆放的矩形，参数为四条边的坐标：

function rect(left, right, bottom, up)

return function (x, y)

return left <= x and x <= right and

bottom <= x and x <= up

end

end

用类似的方法，我们可以用函数来定义其它基本形状，例如三角形、任意角度的矩形。每种形状都具有独立的实现，需要的仅仅是一个合适的特征函数。

现在来看看如何修改与合并区域。要创建任意形状的补集：

function complement (r)

return function (x, y)

return not r(x, y)

end

end

并集、交集与差集也是类似的，如下面的代码所示：

-- 图形的并集、交集与差集

function union (r1, r2)

return function (x, y)

return r1(x, y) or r2(x, y)

end

end

function intersection (r1, r2)

return function (x, y)

return r1(x, y) and r2(x, y)

end

end

function difference (r1, r2)

return function (x, y)

return r1(x, y) and not r2(x, y)

end

end

以下函数会平移形状：

function translate (r, dx, dy)

return function (x, y)

return r(x - dx, y - dy)

end

end

如果要让图形区域可视化，可以遍历整个区域的每个像素。在图形内部的像素用黑色表示，外部像素用白色表示。为简便起见，我们写一个函数生成PBM（portable bitmap）文件，用给定的区域画出图形。

PBM文件的结构很简单（但这种结构比较低效，不过我们的主要目的还是为了简便）。PBM文件是基于文本描述的变体，先以一行字符串“P1”为开头，然后下一行是以像素表示的宽度、高度。最后是一个较长的数字序列，每个数字代表一个像素（1是黑色，0是白色），以空格或换行符分割。下面的代码创建了一个PBM文件，以给定的形状和图片大小绘制图形，将虚拟的图形区域（横、纵坐标范围是从[-1,1]，[-1,1]）映射到可视范围[1,M]，[1,N]。

-- 以PBM格式绘制图形

function plot (r, M, N)

io.write("P1\n", M, " ", N, "\n") -- 表头

for i = 1, N do -- for each line

local y = (N - i\*2)/N

for j = 1, M do -- j代表每一列

local x = (j\*2 - M)/M

io.write(r(x, y) and "1" or "0")

end

io.write("\n")

end

end

最后，下面的指令将绘制一轮从南半球看到的弯月：

c1 = disk(0, 0, 1)

plot(difference(c1, translate(c1, 0.3, 0)), 500, 500)

### 13.8.6 练习

练习8.1：写一个取整函数integral，接受一个函数参数f，返回对f结果取整的函数。

练习8.2：下面的代码运行结果是什么？

function F (x)

return {

set = function (y) x = y end,

get = function () return x end

}

End

o1 = F(10)

o2 = F(20)

print(o1.get(), o2.get())

o2.set(100)

o1.set(300)

print(o1.get(), o2.get())

练习8.3：练习5.4要我们编写一个函数，接受一组多项式参数（用table表示）以及变量的值，然后计算多项式的值。现在请编写一个改进版本——你编写的函数将接受一组多项式参数，然后返回一个用来计算多项式的值的函数，该函数接受变量的值，然后计算出多项式的值。举例：

f = newpoly({3, 0, 1})

print(f(0)) --> 3

print(f(5)) --> 28

print(f(10)) --> 103

练习8.4：借用本章中的图形区域程序，绘制一个从北半球看到的弯月。

练习8.5：为图形系统添加一个函数，该函数接受一个角度，以该角度旋转图形。