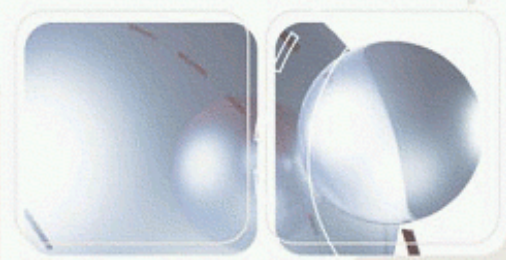
ROBERTO IERUSALIMSCHY

Programming in Lua



Lua.org

Programming in Lua

作者：Roberto Ierusalimschy

翻译：www.luachina.net

Simple is beautiful

版权声明

《Programming in Lua》的翻译由 www.luachina.net完成。本站已经征得作者Mr. Roberto Ierusalimschy的同意，可以翻译他的著作并在本站发布，本书的版权归Mr. Roberto Ierusalimschy 所有，有关版权请参考下面引自官方网站的声明，未经许可不得擅自转贴或者以任何形式发布本书，否则后果自负。

译序

“袁承志知道若再谦逊，那就是瞧人不起，展开五行拳，发拳当胸打去。荣彩和旁观三人本来都以为他武功有独到之秘，哪知使出来的竟是武林中最寻常不过的五行拳。敌对三人登时意存轻视，温青脸上不自禁露出失望的神色。

“荣彩心中暗喜，双拳如风，连抢三下攻势，满拟自己的大力魔爪手江南独步，三四招之间就可破去对方五行拳，那知袁承志轻描淡写的一一化解。再拆数招，荣彩暗暗吃惊，原来对方所使虽是极寻常的拳术，但每一招均是含劲不吐，意在拳先，举手抬足之间隐含极浑厚的内力。”

——金庸《碧血剑》

编程语言之于程序员，若武功招式之于习武之人，招式虽重要，却更在于使用之人。胜者之道，武功只行于表，高手用剑，片草只叶亦威力无穷。

当今武林，派别林立，语言繁杂，林林总总不计其数。主流文化的C/C++、Java、C#、VB[1]；偏安一隅的Fortran；动态语言中的Perl、Tcl、Ruby、Forth、Python，以及本书介绍的Lua；……，等等等等。再加上世界上那些不知道躲在哪的旮旯的奇奇怪怪的hacker捣鼓出来的异想天开的语言，要想各类语言样样精通，不异于痴人说梦。不信可欣赏一下BrainFuck语言[2]的Hello World程序，语言本身依如其名。-☺-

```
>+++++++++[<++++++++>-]<.>+++++++[<++++>-]<+.+++++++..+++.[
-]>++++++++[<++++>-]<.#>++++++++++[<+++++>-]<.>++++++++[<+
++>-]<.+++.------.--------.[-]>++++++++[<++++>-]<+.[-]+++++
+++++.
```

虽说语言的威力依使用者本身的修为高低而定，但不同语言本身的设计又有不同。若让用 Java 写写操作系统内核、Perl 写写驱动程序、C/C++写写 web 应用，都无异于舍近求远，好刀只用上了刀背。

Lua 本身是以简单优雅为本，着眼于处理那些 C 不擅长的任务。借助 C/C++为其扩展，Lua 可闪现无穷魅力。Lua 本身完全遵循 ANSI C 而写成，只要有 C 编译器的地方，Lua 便可发挥她的力量。Lua 不需要追求 Python 那样的大而全的库，太多累赘，反而破坏了她的优美。

语言的优美，来自于使用者自己的感悟。Lua 的优雅，也只有使用后才会明白。

---

[1] http://www.contextfree.net/wangyg/b/tech_his/basic_history.html，VB 虽非正统之 Basic，也不能算纯粹的语言，但其使用广泛，姑且忝列其中吧。

[2] http://www.muppetlabs.com/~breadbox/bf/，有趣的 Brain Fuck 语言。

扬起帆，让我们一同踏上 Lua 的学习之旅……

本书的翻译，是 www.luachina.net中朋友们共同努力的结果。下面是参与翻译与校对的朋友：

```
-- file: 'thanks.lua'
-- desc: to print the list of the contributing guys

helpful_guys = {
    "----参与翻译----",
    "buxiu", "凤舞影天", "zhang3",
    "morler", "lambda", "sunlight",
    "\n",

    "----参与校对----",
    "凤舞影天", "doyle", "flicker",
    "花生魔人", "zhang3", "kasicass",
    "\n"
}

for _,e in ipairs(helpful_guys) do
    print(e)
end
```

www.luachina.net翻译组

2005 年 7 月 26 日

注：本 pdf 为翻译稿，校对工作正在进行。

目录

第一篇 语言

第 0 章 序言

本章包括作者的序言、文章的体例（convention）以及其它一些“每本书开头都会的内容”。

0.1 序言

目前很多程序语言都专注于帮你编写成千上万行的代码，所以此类型的语言所提供的包、命名空间、复杂的类型系统及无数的结构，有上千页的文档需要操作者学习。

而 Lua 并不帮你编写大量的代码的程序，相反的，Lua 仅让你用少量的代码解决关键问题。为实现这个目标，像其他语言一样 Lua 依赖于其可扩展性。但是与其他语言不同的是，不仅用 Lua 编写的软件易于扩展，而且用其他语言比如 C/C++编写的软件也很容易使用 Lua 扩展其功能。

一开始，Lua 就被设计成很容易和传统的 C/C++整合的语言。这种语言的二元性带来了极大的好处。Lua 是一个小巧而简单的语言，因为 Lua 不致力于做 C 语言已经做得很好的领域，比如：性能、底层操作以及与第三方软件的接口。Lua 依赖于 C 去做完成这些任务。Lua 所提供的机制是 C 不善于的：高级语言、动态结构、简洁、易于测试和调试等。正因为如此，Lua 具有良好的安全保证，自动内存管理，简便的字符串处理功能及其他动态数据的改变。

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

除了 Lua 外，还有很多类似的脚本语言，例如：Perl、Tcl、Ruby、Forth、Python。虽然其他语言在某些方面与 Lua 有着共同的特色，但下面这些特征是 Lua 特有的：

① **可扩展性**。Lua 的扩展性非常卓越，以至于很多人把 Lua 用作搭建领域语言的工具（注：比如游戏脚本）。Lua 被设计为易于扩展的，可以通过 Lua 代码或者 C 代码扩展，Lua 的很多功能都是通过外部库来扩展的。Lua 很容易与 C/C++、java、fortran、Smalltalk、Ada，以及其他语言接口。

② **简单**。Lua 本身简单，小巧；内容少但功能强大，这使得 Lua 易于学习，很容易实现一些小的应用。他的完全发布版（代码、手册以及某些平台的二进制文件）

仅用一张软盘就可以装得下。
③ **高效率**。Lua 有很高的执行效率，统计表明 Lua 是目前平均效率最高的脚本语言。
④ **与平台无关**。Lua 几乎可以运行在所有我们听说过的系统上，如 NextStep、OS/2、PlayStation II (Sony)、Mac OS-9、OS X、BeOS、MS-DOS、IBM mainframes、EPOC、PalmOS、MCF5206eLITE Evaluation Board、RISC OS，及所有的 Windows 和 Unix。Lua 不是通过使用条件编译实现平台无关，而是完全使用 ANSI (ISO) C，这意味着只要你有 ANSI C 编译器你就可以编译并使用 Lua。

Lua 大部分强大的功能来自于他的类库，这并非偶然。Lua 的长处之一就是可以通过新类型和函数来扩展其功能。动态类型检查最大限度允许多态出现，并自动简化调用内存管理的接口，因为这样不需要关心谁来分配内存谁来释放内存，也不必担心数据溢出。高级函数和匿名函数均可以接受高级参数，使函数更为通用。

Lua 自带一个小规模的类库。在受限系统中使用 Lua，如嵌入式系统，我们可以有选择地安装这些类库。若运行环境十分严格，我们甚至可以直接修改类库源代码，仅保留需要的函数。记住：Lua 是很小的（即使加上全部的标准库）并且在大部分系统下你仍可以不用担心的使用全部的功能。

0.2 Lua 的使用者

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

第一：很多人使用 Lua 嵌入在应用程序，比如 CGILua（搭建动态网页）、LuaOrb（访问 CORBA 对象。这些类型用 Lua-API 注册新函数，创建新类型，通过配置 Lua 就可以改变应用宿主语言的行为。通常，这种应用的使用者并不知道 Lua 是一种独立的语言。例如：CGILua 用户一般会认为 Lua 是一种用于 Web 的语言。

第二：作为一种独立运行的语言，Lua 也是很有用的，主要用于文本处理或者只运行一次的小程序。这种应用 Lua 主要使用它的标准库来实现，标准库提供模式匹配和其它一些字串处理的功能。我们可以这样认为：Lua 是文本处理领域的嵌入式语言。

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

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

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

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

库、Debug 库。

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

0.3 Lua 的相关资源

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

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

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

另外本书仅针对 Lua 5.0，如果你的版本不同，请查阅 Lua 手册或者比较版本间的差异。

0.4 本书的体例

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

<2> 符号-->表示语句的输出或者表达式的结果:

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

<3> 符号<-->表示等价，即对于 Lua 来说，用 this 与 that 没有区别。

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

0.5 关于本书

开始打算写这本书是 1998 年冬天(南半球)，那时候 Lua 版本是 3.1；2000 年 v4.0；2003 年 v5.0。

很明显的是，这些变化给本书带来很大的冲击，有些内容失去了它存在理由，比如关于超值（upvalues）的复杂的解释。一些章节被重写，比如 C API，另外一些章节被增加进来，比如协同处理。

不太明显的是，Lua 语言本身的发展对本书的完成也产生了很大的影响。一些语言

的变化在本书中并没有被涵盖进来，这并非偶然的。在本书的创作过程中，有的时候在某个章节我会突然感觉很困惑，因为我不知道该从何开始或者怎样去讲问题阐述清楚。当你想尽力去解释清楚如何使用的前提是你应该觉得使用这个东西很容易，这表明 Lua 某些地方需要被改进。还有的时候，我顺利的写完某个章节，结果却是没有人能看得懂我写的或者没有人对我在这个章节内表达的观点达成一致。大部分情况下，这是我的过错因为我是个作家，偶尔我也会因此发现语言本身的一些需要改进的缺陷（举例来说，从 upvalues 到 lexical scoping 的转变是由无意义的尝试所带来的抱怨所引发的，在此书的先前的草稿里，把 upvalues 形容成是 lexical scoping 的一种）。

本书的完成必须服从语言的变化，本书在这个时候完成的原因：

<1> Lua 5.0 是一个成熟的版本
<2> 语言变得越来越大，超出了最初本书的目标。此外一个原因是我迫切的想将 Lua 介绍给大家让更多的人了解 Lua。

0.6 感谢

在完成本书的过程中，很多人给了我极大的帮助：

Luiz Henrique de Figueiredo 和 Waldemar Celes 给了我很大的帮助，使得本书能够更好完成，Luiz Henrique 也帮助设计了本书的内部。

Noemi Rodriguez, André Carregal, Diego Nehab, 以及 Gavin Wraith 阅读了本书的草稿提出了很多有价值的建议。

Renato Cerqueira, Carlos Cassino, Tomás Guisasola, Joe Myers 和 Ed Ferguson 也提出了很多重要的建议。

Alexandre Nakonechnyj 负责本书的封面和内部设计。

Rosane Teles 负责 CIP 数据的准备。

谢谢他们所有人。

第 1 章 起点

写一个最最简单的程序——Hello World。

```
print("Hello World")
```

假定你把上面这句保存在 hello.lua 文件中，你在命令行只需要：

```
prompt> lua hello.lua
```

看到结果了吗?

让我们来看一个稍微复杂点的例子：

```
-- defines a factorial 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("*number")        -- read a number
print(fact(a))
```

这个例子定义了一个函数，计算输入参数 n 的阶乘；本例要求用户输入一个数字 n，然后打印 n 的阶乘。

1.1 Chunks

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，但如果同一行有多个语句最好用；分开

```
a = 1   b = a*2    -- ugly, but valid
```

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。

你还可以以交互模式运行 Lua，不带参数运行 Lua:

```
Lua 5.0  Copyright © 1994-2003 Tecgraf, PUC-Rio
>
```

你键入的每个命令（比如："Hello World"）在你键入回车之后立即被执行，键入文件结束符可以退出交互模式（Ctrl-D in Unix, Ctrl-Z in DOS/Windows），或者调用 OS 库的 os.exit()函数也可以退出。

在交互模式下，Lua 通常把每一个行当作一个 Chunk，但如果 Lua 一行不是一个完整的 Chunk 时，他会等待继续输入直到得到一个完整的 Chunk.在 Lua 等待续行时，显示不同的提示符（一般是>>）.

可以通过指定参数让 Lua 执行一系列 Chunk。例如：假定一个文件 a 内有单个语句 x=1；另一个文件 b 有语句 print(x)

```
prompt> lua -la -lb
```

命令首先在一个 Chunk 内先运行 a 然后运行 b。（注意：-l 选项会调用 require，将会在指定的目录下搜索文件，如果环境变量没有设好，上面的命令可能不能正确运行。我们将在 8.1 节详细更详细的讨论 the require function）

-i 选项要求 Lua 运行指定 Chunk 后进入交互模式.

```
prompt> lua -i -la -lb
```

将在一个 Chunk 内先运行 a 然后运行 b，最后直接进入交互模式。

另一个连接外部 Chunk 的方式是使用 dofile 函数，dofile 函数加载文件并执行它.假设有一个文件:

```
-- file 'lib1.lua'

function norm (x, y)
    local n2 = x^2 + y^2
    return math.sqrt(n2)
end

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

在交互模式下：

```
> dofile("lib1.lua")     -- load your library
> n = norm(3.4, 1.0)
> print(twice(n))        --> 7.0880180586677
```

-i 和 dofile 在调试或者测试 Lua 代码时是很方便的。

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.

```
print(b)      --> nil
b = 10
print(b)      --> 10
```

如果你想删除一个全局变量，只需要将变量负值为 nil

```
b = nil
print(b)      --> nil
```

这样变量 b 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

标示符：字母(letter)或者下划线开头的字母、下划线、数字序列.最好不要使用下划线加大写字母的标示符，因为 Lua 的保留字也是这样的。Lua 中，letter 的含义是依赖于本地环境的。

保留字：以下字符为 Lua 的保留字，不能当作标识符。

```
and       break     do        else      elseif
end       false     for       function  if
in        local     nil       not       or
repeat    return    then      true      until
while
```

注意：Lua 是大小写敏感的.

注释：单行注释:--

多行注释：--[[--]]

```
--[[
print(10)         -- no action (comment)
--]]
```

1.4 命令行方式

lua [options] [script [args]]

-e：直接将命令传入 Lua

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

-l：加载一个文件.

-i：进入交互模式.

_PROMPT 内置变量作为交互模式的提示符

```
prompt> lua -i -e "_PROMPT=' lua> '"
lua>
```

Lua 的运行过程，在运行参数之前，Lua 会查找环境变量 LUA_INIT 的值，如果变量存在并且值为@filename，Lua 将加载指定文件。如果变量存在但不是以@开头，Lua 假定 filename 为 Lua 代码文件并且运行他。利用这个特性，我们可以通过配置，灵活的设置交互模式的环境。可以加载包，修改提示符和路径，定义自己的函数，修改或者重命名函数等。

全局变量 arg 存放 Lua 的命令行参数。

```
prompt> lua script a b c
```

在运行以前，Lua 使用所有参数构造 arg 表。脚本名索引为 0，脚本的参数从 1 开始增加。脚本前面的参数从-1 开始减少。

```
prompt> lua -e "sin=math.sin" script a b
```

arg 表如下：

```
arg[-3] = "lua"
arg[-2] = "-e"
arg[-1] = "sin=math.sin"
arg[0] = "script"
arg[1] = "a"
arg[2] = "b"
```

第 2 章 类型和值

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。

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

变量没有预定义的类型，每一个变量都可能包含任一种类型的值。

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

注意上面最后两行，我们可以使用 function 像使用其他值一样使用（更多的介绍参考第六章）。一般情况下同一变量代表不同类型的值会造成混乱，最好不要用，但是特殊情况下可以带来便利，比如 nil。

2.1 Nil

Lua 中特殊的类型，他只有一个值：nil；一个全局变量没有被赋值以前默认值为 nil；给全局变量负 nil 可以删除该变量。

2.2 Booleans

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

2.3 Numbers

表示实数，Lua 中没有整数。一般有个错误的看法 CPU 运算浮点数比整数慢。事实不是如此，用实数代替整数不会有什么误差（除非数字大于 100,000,000,000,000）。Lua 的 numbers 可以处理任何长整数不用担心误差。你也可以在编译 Lua 的时候使用长整型或者单精度浮点型代替 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，具体的情况请参考 Lua 发布版所附的详细说明。和其他语言类似，数字常量的小数部分和指数部分都是可选的，数字常量的例子：

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

2.4 Strings

指字符的序列。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里。Lua 中字符串是不可以修改的，你可以创建一个新的变量存放你要的字符串，如下：

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

string 和其他对象一样，Lua 自动进行内存分配和释放，一个 string 可以只包含一个字母也可以包含一本书，Lua 可以高效的处理长字符串，1M 的 string 在 Lua 中是很常见的。可以使用单引号或者双引号表示字符串

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

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

```
\a bell
\b back space                   -- 后退
\f form feed                    -- 换页
\n newline                      -- 换行
\r carriage return              -- 回车
\t horizontal tab               -- 制表
\v vertical tab
\\ backslash                    -- "\"
\" double quote                 -- 双引号
\' single quote                 -- 单引号
```

```
\[ left square bracket      -- 左中括号
\] right square bracket     -- 右中括号
```

例子：

```
> 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: '\'
```

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\""和"\97lo\10\04923"'是相同的。

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

```
page = [[
<HTML>
<HEAD>
<TITLE>An HTML Page</TITLE>
</HEAD>
<BODY>
Lua
[[a text between double brackets]]
</BODY>
</HTML>
]]

io.write(page)
```

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

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

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

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

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

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。

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

反之,可以调用 tostring()将数字转成字符串，这种转换一直有效：

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

2.5 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，此外这特性在 Lua 实现面向对象中也起了重要作用（在第 16 章详细讲述）。

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

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

在第九章讨论协同操作的时候，我们介绍线程。

第 3 章 表达式

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

3.1 算术运算符

二元运算符：+ - * / ^ (加减乘除幂)

一元运算符：- (负值)

这些运算符的操作数都是实数。

3.2 关系运算符

```
<       >       <=      >=      ==      ~=
```

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

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

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

当比较不同类型的值的时候要特别注意：

```
"0" == 0      -- false
2 < 15        -- true
"2" < "15"    -- false (alphabetical order!)
```

为了避免不一致的结果，混合比较数字和字符串，Lua 会报错，比如：2 < "15"

3.3 逻辑运算符

```
and    or     not
```

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

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

```
a and b       -- 如果 a 为 false，则返回 a，否则返回 b
a or  b       -- 如果 a 为 true，则返回 a，否则返回 b
```

例如：

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

一个很实用的技巧：如果 x 为 false 或者 nil 则给 x 赋初始值 v

```
x = x or v
```

等价于

```
if not x then
    x = v
end
```

and 的优先级比 or 高。

C 语言中的三元运算符

```
a ? b : c
```

在 Lua 中可以这样实现：

```
(a and b) or c
```

not 的结果只返回 false 或者 true

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

3.4 连接运算符

```
..          --两个点
```

字符串连接，如果操作数为数字，Lua 将数字转成字符串。

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

3.5 优先级

从高到低的顺序：

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

除了^和..外所有的二元运算符都是左连接的。

```
a+i < b/2+1          <-->       (a+i) < ((b/2)+1)
5+x^2*8              <-->       5+((x^2)*8)
a < y and y <= z     <-->       (a < y) and (y <= z)
-x^2                 <-->       -(x^2)
x^y^z                <-->       x^(y^z)
```

3.6 表的构造

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

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

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

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

构造函数可以使用任何表达式初始化：

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

如果想初始化一个表作为 record 使用可以这样：

```
a = {x=0, y=0}        <-->       a = {}; a.x=0; a.y=0
```

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

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

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

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

这段代码从标准输入读进每行，然后反序形成链表。下面的代码打印链表的内容：

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

在同一个构造函数中可以混合列表风格和 record 风格进行初始化，如：

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

这个例子也表明我们可以嵌套构造函数来表示复杂的数据结构.

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

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

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

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

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

list 风格初始化和 record 风格初始化是这种一般初始化的特例:

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

如果真的想要数组下标从 0 开始：

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

注意：不推荐数组下标从 0 开始，否则很多标准库不能使用。

在构造函数的最后的","是可选的，可以方便以后的扩展。

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

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

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

第 4 章 基本语法

Lua 像 C 和 PASCAL 几乎支持所有的传统语句：赋值语句、控制结构语句、函数调用等，同时也支持非传统的多变量赋值、局部变量声明。

4.1 赋值语句

赋值是改变一个变量的值和改变表域的最基本的方法。

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

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

```
a, b = 10, 2*x       <-->       a=10; b=2*x
```

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

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

当变量个数和值的个数不一致时，Lua 会一直以变量个数为基础采取以下策略：

```
a. 变量个数 > 值的个数              按变量个数补足 nil
b. 变量个数 < 值的个数              多余的值会被忽略
```

例如：

```
a, b, c = 0, 1
print(a,b,c)             --> 0   1   nil

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

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

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

```
a, b, c = 0, 0, 0
```

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

多值赋值经常用来交换变量，或将函数调用返回给变量：

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

f()返回两个值，第一个赋给 a，第二个赋给 b。

4.2 局部变量与代码块（block）

使用 local 创建一个局部变量，与全局变量不同，局部变量只在被声明的那个代码块内有效。代码块：指一个控制结构内，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）。

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

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

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

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

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

应该尽可能的使用局部变量，有两个好处：

1. 避免命名冲突
2. 访问局部变量的速度比全局变量更快.

我们给 block 划定一个明确的界限：do..end 内的部分。当你想更好的控制局部变量

的作用范围的时候这是很有用的。

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

print(x1, x2)
```

4.3 控制结构语句

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

if 语句，有三种形式：

```
if conditions then
    then-part
end;

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

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

while 语句：

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

repeat-until 语句：

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

for 语句有两大类：

第一，数值 for 循环：

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

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

有几点需要注意：

1. 三个表达式只会被计算一次，并且是在循环开始前。

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

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

第一个例子 f(x)只会在循环前被调用一次。

2. 控制变量 var 是局部变量自动被声明,并且只在循环内有效.

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

如果需要保留控制变量的值，需要在循环中将其保存

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

3. 循环过程中不要改变控制变量的值，那样做的结果是不可预知的。如果要退出循环，使用 break 语句。

第二，范型 for 循环：

前面已经见过一个例子：

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

范型 for 遍历迭代子函数返回的每一个值。

再看一个遍历表 key 的例子：

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

范型 for 和数值 for 有两点相同：

1. 控制变量是局部变量

2. 不要修改控制变量的值

再看一个例子，假定有一个表：

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

现在想把对应的名字转换成星期几，一个有效地解决问题的方式是构造一个反向表：

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

下面就可以很容易获取问题的答案了:

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

我们不需要手工，可以自动构造反向表

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

如果你对范型 for 还有些不清楚在后面的章节我们会继续来学习。

4.4 break 和 return 语句

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

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

Lua 语法要求 break 和 return 只能出现在 block 的结尾一句（也就是说：作为 chunk 的最后一句，或者在 end 之前，或者 else 前，或者 until 前），例如：

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

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

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

第 5 章 函数

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

语法：

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

调用函数的时候，如果参数列表为空，必须使用()表明是函数调用。

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

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

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

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

Lua 使用的函数，既可是 Lua 编写的，也可以是其他语言编写的，对于 Lua 程序员，用什么语言实现的函数使用起来都一样。

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

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

CALL              PARAMETERS

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

5.1 多返回值

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

```
s, e = string.find("hello Lua users", "Lua")
print(s, e)       --> 7  9
```

Lua 函数中，在 return 后列出要返回的值得列表即可返回多值，如：

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

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

Lua 总是调整函数返回值的个数以适用调用环境，当作为独立的语句调用函数时，所有返回值将被忽略。假设有如下三个函数：

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

第一，当作为表达式调用函数时，有以下几种情况：

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

2. 其他情况下，函数调用仅返回第一个值（如果没有返回值为 nil）

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

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

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

第二，函数调用作为函数参数被调用时，和多值赋值是相同。

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

第三，函数调用在表构造函数中初始化时，和多值赋值时相同。

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

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

另外，return f()这种形式，则返回“f()的返回值”：

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

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

可以使用圆括号强制使调用返回一个值。

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

一个 return 语句如果使用圆括号将返回值括起来也将导致返回一个值。

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

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

unpack 返回 a 所有的元素作为 f()的参数

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

预定义的 unpack 函数是用 C 语言实现的，我们也可以用 Lua 来完成：

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

5.2 可变参数

Lua 函数可以接受可变数目的参数，和 C 语言类似在函数参数列表中使用三点（...）表示函数有可变的参数。Lua 将函数的参数放在一个叫 arg 的表中，除了参数以外，arg 表中还有一个域 n 表示参数的个数。

例如，我们可以重写 print 函数：

```
printResult = ""

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

有时候我们可能需要几个固定参数加上可变参数

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

CALL              PARAMETERS

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

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

举个具体的例子，如果我们只想要 string.find 返回的第二个值。一个典型的方法是使用哑元（dummy variable，下划线）：

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

还可以利用可变参数声明一个 select 函数：

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

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

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

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

这个例子将文本格式化操作和写操作组合为一个函数。

5.3 命名参数

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

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

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

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

根据这个想法我们重定义了 rename：

```
function rename (arg)
```

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

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

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

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

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

第 6 章 再论函数

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

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

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

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

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

既然函数是值，那么表达式也可以创建函数了，Lua 中我们经常这样写：

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

这实际上是 Lua 语法的特例，下面是原本的函数：

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

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

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

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

如果我们想通过表的 name 域排序：

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

以其他函数作为参数的函数在 Lua 中被称作高级函数（higher-order function），如上面的 sort。在 Lua 中，高级函数与普通函数没有区别，它们只是把“作为参数的函数”当作第一类值（first-class value）处理而已。

下面给出一个绘图函数的例子：

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

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

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

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

要想让这个例子正确的运行，你必须调整你的终端类型和代码中的控制符[3]一致：

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

将在屏幕上输出一个正弦曲线。

[3] 终端类型、控制符，参见附录 A。

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

6.1 闭包

当一个函数内部嵌套另一个函数定义时，内部的函数体可以访问外部的函数的局部变量，这种特征我们称作词法定界。虽然这看起来很清楚，事实并非如此，词法定界加上第一类函数在编程语言里是一个功能强大的概念，很少语言提供这种支持。

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

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

假定创建一个函数实现此功能：

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

例子中包含在 sortbygrade 函数内部的 sort 中的匿名函数可以访问 sortbygrade 的参数 grades，在匿名函数内部 grades 不是全局变量也不是局部变量，我们称作外部的局部变量（external local variable）或者 upvalue。（upvalue 意思有些误导，然而在 Lua 中他的存在有历史的根源，还有他比起 external local variable 简短）。

看下面的代码：

```
function newCounter()
    local i = 0
    return function()     -- anonymous function
        i = i + 1
        return i
    end
end

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

```
print(c1())  --> 2
```

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

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

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

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

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

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

这个例子中我们假定 Button 是一个用来创建新按钮的工具， label 是按钮的标签，action 是按钮被按下时调用的回调函数。（实际上是一个闭包，因为他访问 upvalue digit）。digitButton 完成任务返回后，局部变量 digit 超出范围，回调函数仍然可以被调用并且可以访问局部变量 digit。

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

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

```
end
```

更清楚的方式：

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

这样我们把原始版本放在一个局部变量内，访问 sin 的唯一方式是通过新版本的函数。

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

```
do
    local oldOpen = io.open
    io.open = function (filename, mode)
        if access_OK(filename, mode) then
            return oldOpen(filename, mode)
        else
            return nil, "access denied"
        end
    end
end
```

6.2 非全局函数

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

1. 表和函数放在一起

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

2. 使用表构造函数

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

3. Lua 提供另一种语法方式

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

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

1. 方式一

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

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

2. 方式二

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

有一点需要注意的是在声明递归局部函数的方式：

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

```
    end
end
```

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

```
local fact

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

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

但是 Lua 扩展了他的语法使得可以在直接递归函数定义时使用两种方式都可以。

在定义非直接递归局部函数时要先声明然后定义才可以：

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

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

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

6.3 正确的尾调用（Proper Tail Calls）

Lua 中函数的另一个有趣的特征是可以正确的处理尾调用（proper tail recursion，一些书使用术语“尾递归”，虽然并未涉及到递归的概念）。

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

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

g 的调用是尾调用。

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

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

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

需要注意的是：必须明确什么是尾调用。

一些调用者函数调用其他函数后也没有做其他的事情但不属于尾调用。比如：

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

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

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

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

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

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

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

```
function room1 ()
    local move = io.read()
    if move == "south" then
```

```
        return room3()
    elseif move == "east" then
        return room2()
    else
        print("invalid move")
        return room1()   -- stay in the same room
    end
end

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

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

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

我们可以调用 room1()开始这个游戏。

如果没有正确的尾调用，每次移动都要创建一个栈，多次移动后可能导致栈溢出。

但正确的尾调用可以无限制的尾调用，因为每次尾调用只是一个 goto 到另外一个函数并不是传统的函数调用。

第 7 章 迭代器与泛型 for

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

7.1 迭代器与闭包

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

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

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

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

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

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

```
end
```

我们设计的这个迭代器也很容易用于范性 for 语句

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

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

下面看一个稍微复杂一点的例子：我们写一个迭代器遍历一个文件内的所有匹配的单词。为了实现目的，我们需要保留两个值：当前行和在当前行的偏移量，我们使用两个外部局部变量 line、pos 保存这两个值。

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

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

尽管迭代函数有些复杂，但使用起来是很直观的：

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

```
    print(word)
end
```

通常情况下，迭代函数大都难写易用。这不是大问题，一般 Lua 编程不需要自己写迭代函数，语言本身提供了许多。当然，必要时，自己动手构造一二亦可。

7.2 范性 for 的语义

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

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

范性 for 的文法如下：

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

<var-list>是以一个或多个逗号分隔的变量名列表，<exp-list>是以一个或多个逗号分隔的表达式列表，通常情况下 exp-list 只有一个值：迭代工厂的调用。

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

上面代码中，k, v 为变量列表；pair(t)为表达式列表。

在很多情况下变量列表也只有一个变量，比如：

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

我们称变量列表中第一个变量为控制变量，其值为 nil 时循环结束。

下面我们看看范性 for 的执行过程：

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

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

第三，将迭代函数返回的值赋给变量列表。

第四，如果返回的第一个值为 nil 循环结束，否则执行循环体。

第五，回到第二步再次调用迭代函数。

更具体地说：

```
for var_1, ..., var_n in explist do block end
```

等价于

```
do
    local _f, _s, _var = explist
    while true do
        local var_1, ... , var_n = _f(_s, _var)
        _var = var_1
        if _var == nil then break end
        block
    end
end
```

如果我们的迭代函数是 f，状态常量是 s，控制变量的初始值是 a0，那么控制变量将循环：a1=f(s,a0)、a2=f(s,a1)、……，直到 ai=nil。

7.3 无状态的迭代器

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

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

```
a = {"one", "two", "three"}
for i, v in ipairs(a) do
    print(i, v)
end
```

迭代的状态包括被遍历的表（循环过程中不会改变的状态常量）和当前的索引下标（控制变量），ipairs 和迭代函数都很简单，我们在 Lua 中可以这样实现：

```
function iter (a, i)
    i = i + 1
    local v = a[i]
    if v then
```

```
        return i, v
    end
end

function ipairs (a)
    return iter, a, 0
end
```

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

Lua 库中实现的 pairs 是一个用 next 实现的原始方法：

```
function pairs (t)
    return next, t, nil
end
```

还可以不使用 ipairs 直接使用 next

```
for k, v in next, t do
    ...
end
```

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

7.4 多状态的迭代器

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

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

开始迭代的函数是很简单的，他必须返回迭代函数和初始状态：

```
local iterator       -- to be defined later

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

真正的处理工作是在迭代函数内完成：

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

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

7.5 真正的迭代器

迭代器的名字有一些误导，因为它并没有迭代，完成迭代功能的是 for 语句，也许更好的叫法应该是生成器（generator）；但是在其他语言比如 java、C++迭代器的说法已经很普遍了，我们也就沿用这个术语。

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

作为一个具体的例子，我们使用上述方式重写 allwords 迭代器：

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

```
            f(w)
        end
    end
end
```

如果我们想要打印出单词，只需要

```
allwords(print)
```

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

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

用 for 结构完成同样的任务：

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

真正的迭代器风格的写法在 Lua 老版本中很流行，那时还没有 for 循环。

两种风格的写法相差不大，但也有区别：一方面，第二种风格更容易书写和理解；另一方面，for 结构更灵活，可以使用 break 和 continue 语句；在真正的迭代器风格写法中 return 语句只是从匿名函数中返回而不是退出循环。

第 8 章 编译・运行・错误信息

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

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

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

如果 loadfile 失败 assert 会抛出错误。

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

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

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

f 将是一个函数，调用时执行 i=i+1。

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

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

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

与其他函数一样，chunks 可以定义局部变量也可以返回值：

```
f = loadstring("local a = 10; return a + 20")
```

```
print(f())        --> 30
```

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

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

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

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

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

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

如果你想快捷的调用 dostring（比如加载并运行），可以这样

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

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

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

通常使用 loadstring 加载一个字串没什么意义，例如：

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

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

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

这个例子中，和想象的一样 g 使用局部变量 i，然而 f 使用全局变量 i；loadstring 总是在全局环境中编译他的串。

loadstring 通常用于运行程序外部的代码，比如运行用户自定义的代码。注意：loadstring 期望一个 chunk，即语句。如果想要加载表达式，需要在表达式前加 return，那样将返回表达式的值。看例子：

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

loadstring 返回的函数和普通函数一样，可以多次被调用：

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

8.1 require 函数

Lua 提供高级的 require 函数来加载运行库。粗略的说 require 和 dofile 完成同样的功能但有两点不同：

1. require 会搜索目录加载文件
2. require 会判断是否文件已经加载避免重复加载同一文件。由于上述特征，require 在 Lua 中是加载库的更好的函数。

require 使用的路径和普通我们看到的路径还有些区别，我们一般见到的路径都是一个目录列表。require 的路径是一个模式列表，每一个模式指明一种由虚文件名（require 的参数）转成实文件名的方法。更明确地说，每一个模式是一个包含可选的问号的文件名。匹配的时候 Lua 会首先将问号用虚文件名替换，然后看是否有这样的文件存在。如果不存在继续用同样的方法用第二个模式匹配。例如，路径如下：

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

调用 require "lili"时会试着打开这些文件：

```
lili
lili.lua
c:\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，其他的信息（目录分隔符，文件扩展名）在路径中定义。

为了确定路径，Lua 首先检查全局变量 LUA_PATH 是否为一个字符串，如果是则认为这个串就是路径；否则 require 检查环境变量 LUA_PATH 的值，如果两个都失败 require 使用固定的路径（典型的"?;?.lua"）

require 的另一个功能是避免重复加载同一个文件两次。Lua 保留一张所有已经加载的文件的列表（使用 table 保存）。如果一个加载的文件在表中存在 require 简单的返回；表中保留加载的文件的虚名，而不是实文件名。所以如果你使用不同的虚文件名 require 同一个文件两次，将会加载两次该文件。比如 require "foo"和 require "foo.lua"，路径为 "?;?.lua"将会加载 foo.lua 两次。我们也可以通过全局变量_LOADED 访问文件名列表，这样我们就可以判断文件是否被加载过；同样我们也可以使用一点小技巧让 require 加载一个文件两次。比如，require "foo"之后_LOADED["foo"]将不为 nil，我们可以将其赋值为 nil，require "foo.lua"将会再次加载该文件。

一个路径中的模式也可以不包含问号而只是一个固定的路径，比如：

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

这种情况下，require 没有匹配的时候就会使用这个固定的文件（当然这个固定的路径必须放在模式列表的最后才有意义）。在 require 运行一个 chunk 以前，它定义了一个全局变量_REQUIREDNAME 用来保存被 required 的虚文件的文件名。我们可以通过使用这个技巧扩展 require 的功能。举个极端的例子，我们可以把路径设为 "/usr/local/lua/newrequire.lua"，这样以后每次调用 require 都会运行 newrequire.lua，这种情况下可以通过使用_REQUIREDNAME 的值去实际加载 required 的文件。

8.2 C Packages

Lua 和 C 是很容易结合的，使用 C 为 Lua 写包。与 Lua 中写包不同，C 包在使用以前必须首先加载并连接，在大多数系统中最容易的实现方式是通过动态连接库机制，然而动态连接库不是 ANSI C 的一部分，也就是说在标准 C 中实现动态连接是很困难的。

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

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

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

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

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

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

8.3 错误

Errare humanum est（拉丁谚语：犯错是人的本性）。所以我们要尽可能的防止错误的发生，Lua 经常作为扩展语言嵌入在别的应用中，所以不能当错误发生时简单的崩溃或者退出。相反，当错误发生时 Lua 结束当前的 chunk 并返回到应用中。

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

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

Lua 提供了专门的内置函数 assert 来完成上面类似的功能：

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

assert 首先检查第一个参数，若没问题，assert 不做任何事情；否则，assert 以第二个参数作为错误信息抛出。第二个参数是可选的。注意，assert 会首先处理两个参数，然后才调用函数，所以下面代码，无论 n 是否为数字，字符串连接操作总会执行：

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

当函数遇到异常有两个基本的动作：返回错误代码或者抛出错误。选择哪一种方式，没有固定的规则，不过基本的原则是：对于程序逻辑上能够避免的异常，以抛出错误的方式处理之，否则返回错误代码。

例如 sin 函数，假定我们让 sin 碰到错误时返回错误代码，则使用 sin 的代码可能变为：

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

```
    ...
```

当然，我们也可以在调用 sin 前检查 x 是否为数字：

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

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

再来看一个例子。io.open 函数用于打开文件，如果文件不存在，结果会如何？很多系统中，我们通过“试着去打开文件”来判断文件是否存在。所以如果 io.open 不能打开文件（由于文件不存在或者没有权限），函数返回 nil 和错误信息。依据这种方式，我们可以通过与用户交互（比如：是否要打开另一个文件）合理地处理问题：

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

如果你想偷懒不想处理这些情况，又想代码安全的运行，可以使用 assert：

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

Lua 中有一个习惯：如果 io.open 失败，assert 将抛出错误。

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

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

8.4 异常和错误处理

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

如果在 Lua 中需要处理错误，需要使用 pcall 函数封装你的代码。

假定你想运行一段 Lua 代码，这段代码运行过程中可以捕捉所有的异常和错误。

第一步：将这段代码封装在一个函数内

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

第二步：使用 pcall 调用这个函数

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

当然也可以用匿名函数的方式调用 pcall：

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

pcall 在保护模式（protected mode）下执行函数内容，同时捕获所有的异常和错误。若一切正常，pcall 返回 true 以及“被执行函数”的返回值；否则返回 nil 和错误信息。

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

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

这种机制提供了强大的能力，足以应付 Lua 中的各种异常和错误情况。我们通过 error 抛出异常，然后通过 pcall 捕获之。

8.5 错误信息和回跟踪（Tracebacks）

虽然你可以使用任何类型的值作为错误信息，通常情况下，我们使用字符串来描述遇到的错误。如果遇到内部错误（比如对一个非 table 的值使用索引下标访问）Lua 将自己产生错误信息，否则 Lua 使用传递给 error 函数的参数作为错误信息。不管在什么情况下，Lua 都尽可能清楚的描述问题发生的缘由。

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

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

例子中错误信息给出了文件名（stdin）与行号。

函数 error 还可以有第二个参数，表示错误发生的层级。比如，你写了一个函数用来检查“error 是否被正确调用”：

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

可有人这样调用此函数：

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

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

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

当错误发生的时候，我们常常希望了解详细的信息，而不仅是错误发生的位置。若能了解到“错误发生时的栈信息”就好了，但 pcall 返回错误信息时，已经释放了保存错误发生情况的栈信息。因此，若想得到 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、错误处理函数。当错误发生时，Lua 会在栈释放以前调用错误处理函数，因此可以使用 debug 库收集错误相关信息。有两个常用的 debug 处理函数：debug.debug 和 debug.traceback，前者给出 Lua 的提示符，你可以自己动手察看错误发生时的情况；后者通过 traceback 创建更多的错误信息，也是控制台解释器用来构建错误信息的函数。你可以在任何时候调用 debug.traceback 获取当前运行的 traceback 信息：

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

第 9 章 协同程序

协同程序（coroutine）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（IP，instruction pointer），但与其它协同程序共享全局变量等很多信息。线程和协同程序的主要不同在于：在多处理器情况下，从概念上来讲多线程程序同时运行多个线程；而协同程序是通过协作来完成，在任一指定时刻只有一个协同程序在运行，并且这个正在运行的协同程序只在必要时才会被挂起。

协同是非常强大的功能，但是用起来也很复杂。如果你是第一次阅读本章，某些例子可能会不大理解，不必担心，可先继续阅读后面的章节，再回头琢磨本章内容。

9.1 协同的基础

Lua 的所有协同函数存放于 coroutine table 中。create 函数用于创建新的协同程序，其只有一个参数：一个函数，即协同程序将要运行的代码。若一切顺利，返回值为 thread 类型，表示创建成功。通常情况下，create 的参数是匿名函数：

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

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

协同有三个状态：挂起态（suspended）、运行态（running）、停止态（dead）。当我们创建协同程序成功时，其为挂起态，即此时协同程序并未运行。我们可用 status 函数检查协同的状态：

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

函数 coroutine.resume 使协同程序由挂起状态变为运行态：

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

本例中，协同程序打印出"hi"后，任务完成，便进入终止态：

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

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

```
co = coroutine.create(function ()
    for i=1,10 do
```

```
        print("co", i)
        coroutine.yield()
    end
end)
```

执行这个协同程序，程序将在第一个 yield 处被挂起：

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

从协同的观点看：使用函数 yield 可以使程序挂起，当我们激活被挂起的程序时，将从函数 yield 的位置继续执行程序，直到再次遇到 yield 或程序结束。

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

上面最后一次调用时，协同体已结束，因此协同程序处于终止态。如果我们仍然希望激活它，resume 将返回 false 和错误信息。

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

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

Lua 中协同的强大能力，还在于通过 resume-yield 来交换数据。

第一个例子中只有 resume，没有 yield，resume 把参数传递给协同的主程序。

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

第二个例子，数据由 yield 传给 resume。true 表明调用成功，true 之后的部分，即是 yield 的参数。

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

相应地，resume 的参数，会被传递给 yield。

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

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

最后一个例子，协同代码结束时的返回值，也会传给 resume：

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

我们很少在一个协同程序中同时使用多个特性，但每一种都有用处。

现在已大体了解了协同的基础内容，在我们继续学习之前，先澄清两个概念：Lua 的协同称为不对称协同（asymmetric coroutines），指“挂起一个正在执行的协同函数”与“使一个被挂起的协同再次执行的函数”是不同的，有些语言提供对称协同（symmetric coroutines），即使用同一个函数负责“执行与挂起间的状态切换”。

有人称不对称的协同为半协同，另一些人使用同样的术语表示真正的协同，严格意义上的协同不论在什么地方只要它不是在其他的辅助代码内部的时候都可以并且只能使执行挂起，不论什么时候在其控制栈内都不会有不可决定的调用。(However, other people use the same term semi-coroutine to denote a restricted implementation of coroutines, where a coroutine can only suspend its execution when it is not inside any auxiliary function, that is, when it has no pending calls in its control stack.)。只有半协同程序内部可以使用 yield，python 中的产生器（generator）就是这种类型的半协同。

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

9.2 管道和过滤器

协同最具代表性的例子是用来解决生产者-消费者问题。假定有一个函数不断地生产数据（比如从文件中读取），另一个函数不断的处理这些数据（比如写到另一文件中），函数如下：

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

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

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

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

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

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

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

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

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

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

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

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

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

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

可以调用：

```
p = producer()
```

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

或者：

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

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

9.3 用作迭代器的协同

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

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

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

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

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

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

        end
```

```
    end
end

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

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

有了上面的生成器后，下面我们将这个例子修改一下使其转换成一个迭代函数：

1. 第一步 printResult 改为 yield

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

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

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

这样我们就可以使用 for 循环来打印出一个数组的所有排列情况了：

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

```
--> a b c
```

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

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

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

9.4 非抢占式多线程

如前面所见，Lua 中的协同是一协作的多线程，每一个协同等同于一个线程，yield-resume 可以实现在线程中切换。然而与真正的多线程不同的是，协同是非抢占式的。当一个协同正在运行时，不能在外部终止他。只能通过显示的调用 yield 挂起他的执行。对于某些应用来说这个不存在问题，但有些应用对此是不能忍受的。不存在抢占式调用的程序是容易编写的。不需要考虑同步带来的 bugs，因为程序中的所有线程间的同步都是显示的。你仅仅需要在协同代码超出临界区时调用 yield 即可。

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

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

第一，加载 LuaSocket 库

```
require "luasocket"
```

第二，定义远程主机和需要下载的文件名

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

第三，打开一个 TCP 连接到远程主机的 80 端口（http 服务的标准端口）

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

上面这句返回一个连接对象，我们可以使用这个连接对象请求发送文件

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

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

第四，关闭连接

```
c:close()
```

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

使用协同机制重写上面的代码，在一个函数内：

```
function download (host, file)
    local c = assert(socket.connect(host, 80))
    local count = 0       -- counts number of bytes read
    c:send("GET " .. file .. " HTTP/1.0\r\n\r\n")
    while true do
        local s, status = receive©
        count = count + string.len(s)
        if status == "closed" then break end
    end
    c:close()
    print(file, count)
end
```

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

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

在同步接受数据的方式下，函数接收数据时不能被阻塞，而是在没有数据可取时

yield，代码如下：

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

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

下面的函数保证每一个下载运行在自己独立的线程内：

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

代码中 table 中为分配器保存了所有活动的线程。

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

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

```
        end
    end
end
```

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

```
host = "www.w3c.org"

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

dispatcher()      -- main loop
```

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

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

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

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

```
        if table.getn(connections) == n then
            socket.select(connections)
        end
    end
end
```

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

第 10 章 完整示例

本章通过两个完整的例子，来展现 Lua 的实际应用。第一个例子来自于 Lua 官方网站，其展示了 Lua 作为数据描述语言的应用。第二个例子为马尔可夫链算法的实现，算法在 Kernighan & Pike 著作的 Practice of Programming 书中有描述。本章结束后，Lua 语言方面的介绍便到此结束。后续章节将分别介绍 table 与面向对象（object-orient）、标准库以及 C-API 等内容。

10.1 Lua 作为数据描述语言使用

慢慢地，Lua 正被世界上越来越多的人使用。Lua 官方网站的数据库中保存着一些“使用了 Lua”的项目的信息。在数据库中，我们用一个构造器以自动归档的方式表示每个工程入口，代码如下：

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

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

```
<HTML>
<HEAD><TITLE>Projects using Lua</TITLE></HEAD>
```

```
<BODY BGCOLOR="#FFFFFF">
Here are brief descriptions of some projects around the
world that use <A HREF="home.html">Lua</A>.


<UL>
<LI><A HREF="#1">TeCGraf</A>
<LI> ...
</UL>

<H3>
<A NAME="1"
    HREF="http://www.tecgraf.puc-rio.br/">TeCGraf</A>


<SMALL><EM>Computer Graphics Technology Group,
PUC-Rio</EM></SMALL>
</H3>

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

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

</BODY></HTML>
```

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

首先，我们定义一个辅助函数用来格式化文本的输出（参见 5.2 函数部分内容）

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

第二，我们定义一个 BEGIN 函数用来写 html 页面的头部

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

第三，定义 entry 函数

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

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

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

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

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

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

    if o.title and o.org then
        fwrite('\n<SMALL><EM>%s</EM></SMALL>', o.org)
    end
    fwrite('\n</H3>\n')

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

```
                        '\n\n\n*', '<P>\n'))
        fwrite('<P>\n')
    end

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

由于 html 中使用双引号，为了避免冲突我们这里使用单引号表示串。

第四，定义 END 函数，写 html 的尾部

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

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

```
BEGIN()

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

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

END()
```

10.2 马尔可夫链算法

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

程序的第一部分读出原文，并且对没两个单词的前缀建立一个表，这个表给出了具

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

我们编写一个函数用来将两个单词中间加上空个连接起来：

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

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

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

初始化构造的表为：

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

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

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

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

我们使用两个变量 w1 和 w2 来保存最后读入的两个单词的值，对于每一个前缀，我

们保存紧跟其后的单词的列表。例如上面例子中初始化构造的表。

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

```
-- Markov Chain Program in Lua

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

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

local statetab

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

local N = 2
local MAXGEN = 10000
```

```
local NOWORD = "\n"

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

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

第二篇 tables 与 objects

第 11 章 数据结构

table 是 Lua 中唯一的数据结构，其他语言所提供的数据结构，如：arrays、records、lists、queues、sets 等，Lua 都是通过 table 来实现，并且在 lua 中 table 很好的实现了这些数据结构。

在传统的 C 语言或者 Pascal 语言中我们经常使用 arrays 和 lists（record+pointer）来实现大部分的数据结构，在 Lua 中不仅可以用 table 完成同样的功能，而且 table 的功能更加强大。通过使用 table 很多算法的实现都简化了，比如你在 lua 中很少需要自己去实现一个搜索算法，因为 table 本身就提供了这样的功能。

我们需要花一些时间去学习如何有效的使用 table，下面通过一些例子，我们来看看如果通过 table 来实现一些常用的数据结构。首先，我们从 arrays 和 lists 开始，因为两者是其他数据结构的基础，大家也比较熟悉。前面章节，我们已接触了 table 的一些内容，本章，我们将彻底了解它。

11.1 数组

在 lua 中通过整数下标访问 table 中元素，即是数组。并且数组大小不固定，可动态增长。

通常我们初始化数组时，就间接地定义了数组的大小，例如：

```
a = {}     -- new array
for i=1, 1000 do
    a[i] = 0
end
```

数组 a 的大小为 1000，访问 1-1000 范围外的值，将返回 nil。数组下标可以根据需要，从任意值开始，比如：

```
-- creates an array with indices from -5 to 5
a = {}
for i=-5, 5 do
    a[i] = 0
end
```

然而习惯上，Lua 的下标从 1 开始。Lua 的标准库遵循此惯例，因此你的数组下标必须也是从 1 开始，才可以使用标准库的函数。

我们可以用构造器在创建数组的同时初始化数组：

```
squares = {1, 4, 9, 16, 25, 36, 49, 64, 81}
```

这样的语句中，数组的大小可以任意的大。

11.2 矩阵和多维数组

Lua 中有两种表示矩阵的方法，一是“数组的数组”。也就是说，table 的每个元素是另一个 table。例如，可以使用下面代码创建一个 n 行 m 列的矩阵：

```
mt = {}           -- create the matrix
for i=1,N do
    mt[i] = {}    -- create a new row
    for j=1,M do
        mt[i][j] = 0
    end
end
```

由于 Lua 中 table 是对象，所以每一行我们必须显式地创建一个 table，比起 c 或 pascal，这显得冗余，但另一方面也提供了更多的灵活性，例如可修改前面的例子创建一个三角矩阵：

```
for j=1,M do
```

改成

```
for j=1,i do
```

这样实现的三角矩阵比起整个矩阵，仅使用一半的内存空间。

表示矩阵的另一方法，是将行和列组合起来。如果索引下标都是整数，通过第一个索引乘于一个常量（列）再加上第二个索引，看下面的例子实现创建 n 行 m 列的矩阵：

```
mt = {}           -- create the matrix
for i=1,N do
    for j=1,M do
        mt[i*M + j] = 0
    end
end
```

如果索引是字符串，可用一个单字符将两个字符串索引连接起来构成一个单一的索引下标，例如一个矩阵 m，索引下标为 s 和 t，假定 s 和 t 都不包含冒号，代码为：m[s..':'..t]，如果 s 或者 t 包含冒号将导致混淆，比如("a:", "b") 和("a", ":b")，当对这种情况有疑问的时候可以使用控制字符来连接两个索引字符串，比如'\0'。

实际应用中常常使用稀疏矩阵，稀疏矩阵指矩阵的大部分元素都为空或者 0 的矩阵。例如，我们通过图的邻接矩阵来存储图，也就是说：当 m,n 两个节点有连接时，矩阵的

m,n 值为对应的 x，否则为 nil。如果一个图有 10000 个节点，平均每个节点大约有 5 条边，为了存储这个图需要一个行列分别为 10000 的矩阵，总计 10000*10000 个元素，实际上大约只有 50000 个元素非空（每行有五列非空，与每个节点有五条边对应）。很多数据结构的书上讨论采用何种方式才能节省空间，但是在 Lua 中你不需要这些技术，因为用 table 实现的数据本身天生的就具有稀疏的特性。如果用我们上面说的第一种多维数组来表示，需要 10000 个 table，每个 table 大约需要五个元素（table）；如果用第二种表示方法来表示，只需要一张大约 50000 个元素的表，不管用那种方式，你只需要存储那些非 nil 的元素。

11.3 链表

Lua 中用 tables 很容易实现链表，每一个节点是一个 table，指针是这个表的一个域（field），并且指向另一个节点（table）。例如，要实现一个只有两个域：值和指针的基本链表，代码如下：

根节点：

```
list = nil
```

在链表开头插入一个值为 v 的节点：

```
list = {next = list, value = v}
```

要遍历这个链表只需要：

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

其他类型的链表，像双向链表和循环链表类似的也是很容易实现的。然后在 Lua 中在很少情况下才需要这些数据结构，因为通常情况下有更简单的方式来替换链表。比如，我们可以用一个非常大的数组来表示栈，其中一个域 n 指向栈顶。

11.4 队列和双向队列

虽然可以使用 Lua 的 table 库提供的 insert 和 remove 操作来实现队列，但这种方式实现的队列针对大数据量时效率太低，有效的方式是使用两个索引下标，一个表示第一个元素，另一个表示最后一个元素。

```
function ListNew ()
    return {first = 0, last = -1}
end
```

为了避免污染全局命名空间，我们重写上面的代码，将其放在一个名为 list 的 table 中：

```
List = {}
function List.new ()
    return {first = 0, last = -1}
end
```

下面，我们可以在常量时间内，完成在队列的两端进行插入和删除操作了。

```
function List.pushleft (list, value)
    local first = list.first - 1
    list.first = first
    list[first] = value
end

function List.pushright (list, value)
    local last = list.last + 1
    list.last = last
    list[last] = value
end

function List.popleft (list)
    local first = list.first
    if first > list.last then error("list is empty") end
    local value = list[first]
    list[first] = nil    -- to allow garbage collection
    list.first = first + 1
    return value
end

function List.popright (list)
    local last = list.last
    if list.first > last then error("list is empty") end
    local value = list[last]
    list[last] = nil     -- to allow garbage collection
    list.last = last - 1
    return value
end
```

对严格意义上的队列来讲，我们只能调用 pushright 和 popleft，这样以来，first 和 last 的索引值都随之增加，幸运的是我们使用的是 Lua 的 table 实现的，你可以访问数组的元

素，通过使用下标从 1 到 20，也可以 16,777,216 到 16,777,236。另外，Lua 使用双精度表示数字，假定你每秒钟执行 100 万次插入操作，在数值溢出以前你的程序可以运行 200 年。

11.5 集合和包

假定你想列出在一段源代码中出现的所有标示符，某种程度上，你需要过滤掉那些语言本身的保留字。一些 C 程序员喜欢用一个字符串数组来表示，将所有的保留字放在数组中，对每一个标示符到这个数组中查找看是否为保留字，有时候为了提高查询效率，对数组存储的时候使用二分查找或者 hash 算法。

Lua 中表示这个集合有一个简单有效的方法，将所有集合中的元素作为下标存放在一个 table 里，下面不需要查找 table，只需要测试看对于给定的元素，表的对应下标的元素值是否为 nil。比如：

```
reserved = {
    ["while"] = true,     ["end"] = true,
    ["function"] = true, ["local"] = true,
}

for w in allwords() do
    if reserved[w] then
    -- `w' is a reserved word
    ...
```

还可以使用辅助函数更加清晰的构造集合：

```
function Set (list)
    local set = {}
    for _, l in ipairs(list) do set[l] = true end
    return set
end

reserved = Set{"while", "end", "function", "local", }
```

11.6 字符串缓冲

假定你要拼接很多个小的字符串为一个大的字符串，比如，从一个文件中逐行读入字符串。你可能写出下面这样的代码：

```
-- WARNING: bad code ahead!!
```

```
local buff = ""
for line in io.lines() do
    buff = buff .. line .. "\n"
end
```

尽管这段代码看上去很正常，但在 Lua 中他的效率极低，在处理大文件的时候，你会明显看到很慢，例如，需要花大概 1 分钟读取 350KB 的文件。（这就是为什么 Lua 专门提供了 io.read(*all)选项，她读取同样的文件只需要 0.02s）

为什么这样呢？Lua 使用真正的垃圾收集算法，但他发现程序使用太多的内存他就会遍历他所有的数据结构去释放垃圾数据，一般情况下，这个算法有很好的性能（Lua 的快并非偶然的），但是上面那段代码 loop 使得算法的效率极其低下。

为了理解现象的本质，假定我们身在 loop 中间，buff 已经是一个 50KB 的字符串，每一行的大小为 20bytes，当 Lua 执行 buff..line.."\n"时，她创建了一个新的字符串大小为 50,020 bytes，并且从 buff 中将 50KB 的字符串拷贝到新串中。也就是说，对于每一行，都要移动 50KB 的内存，并且越来越多。读取 100 行的时候（仅仅 2KB），Lua 已经移动了 5MB 的内存，使情况变遭的是下面的赋值语句：

```
buff = buff .. line .. "\n"
```

老的字符串变成了垃圾数据，两轮循环之后，将有两个老串包含超过 100KB 的垃圾数据。这个时候 Lua 会做出正确的决定，进行他的垃圾收集并释放 100KB 的内存。问题在于每两次循环 Lua 就要进行一次垃圾收集，读取整个文件需要进行 200 次垃圾收集。并且它的内存使用是整个文件大小的三倍。

这个问题并不是 Lua 特有的：其它的采用垃圾收集算法的并且字符串不可变的语言也都存在这个问题。Java 是最著名的例子，Java 专门提供 StringBuffer 来改善这种情况。

在继续进行之前，我们应该做个注释的是，在一般情况下，这个问题并不存在。对于小字符串，上面的那个循环没有任何问题。为了读取整个文件我们可以使用 io.read(*all)，可以很快的将这个文件读入内存。但是在某些时候，没有解决问题的简单的办法，所以下面我们将介绍更加高效的算法来解决这个问题。

我们最初的算法通过将循环每一行的字符串连接到老串上来解决问题，新的算法避免如此：它连接两个小串成为一个稍微大的串，然后连接稍微大的串成更大的串。。。算法的核心是：用一个栈，在栈的底部用来保存已经生成的大的字符串，而小的串从栈定入栈。栈的状态变化和经典的汉诺塔问题类似：位于栈下面的串肯定比上面的长，只要一个较长的串入栈后比它下面的串长，就将两个串合并成一个新的更大的串，新生成的串继续与相邻的串比较如果长于底部的将继续进行合并，循环进行到没有串可以合并或者到达栈底。

```
function newStack ()
    return {""}   -- starts with an empty string
end
```

```
function addString (stack, s)
    table.insert(stack, s)   -- push 's' into the the stack
    for i=table.getn(stack)-1, 1, -1 do
        if string.len(stack[i]) > string.len(stack[i+1]) then
            break
        end
        stack[i] = stack[i] .. table.remove(stack)
    end
end
```

要想获取最终的字符串，我们只需要从上向下一次合并所有的字符串即可。table.concat 函数可以将一个列表的所有串合并。

使用这个新的数据结构，我们重写我们的代码：

```
local s = newStack()
for line in io.lines() do
    addString(s, line .. "\n")
end
s = toString(s)
```

最终的程序读取 350KB 的文件只需要 0.5s，当然调用 io.read("*all")仍然是最快的只需要 0.02s。

实际上，我们调用 io.read("*all")的时候，io.read 就是使用我们上面的数据结构，只不过是用 C 实现的，在 Lua 标准库中，有些其他函数也是用 C 实现的，比如 table.concat，使用 table.concat 我们可以很容易的将一个 table 的中的字符串连接起来，因为它使用 C 实现的，所以即使字符串很大它处理起来速度还是很快的。

Concat 接受第二个可选的参数，代表插入的字符串之间的分隔符。通过使用这个参数，我们不需要在每一行之后插入一个新行：

```
local t = {}
for line in io.lines() do
    table.insert(t, line)
end
s = table.concat(t, "\n") .. "\n"
```

io.lines 迭代子返回不带换行符的一行，concat 在字符串之间插入分隔符，但是最后一字符串之后不会插入分隔符，因此我们需要在最后加上一个分隔符。最后一个连接操作复制了整个字符串，这个时候整个字符串可能是很大的。我们可以使用一点小技巧，插入一个空串：

```
table.insert(t, "")
s = table.concat(t, "\n")
```

第 12 章 数据文件与持久化

当我们处理数据文件的，一般来说，写文件比读取文件内容来的容易。因为我们可以很好的控制文件的写操作，而从文件读取数据常常碰到不可预知的情况。一个健壮的程序不仅应该可以读取存有正确格式的数据还应该能够处理坏文件（译者注：对数据内容和格式进行校验，对异常情况能够做出恰当处理）。正因为如此，实现一个健壮的读取数据文件的程序是很困难的。

正如我们在 Section 10.1（译者：第 10 章 Complete Examples）中看到的例子，文件格式可以通过使用 Lua 中的 table 构造器来描述。我们只需要在写数据的稍微做一些做一点额外的工作，读取数据将变得容易很多。方法是：将我们的数据文件内容作为 Lua 代码写到 Lua 程序中去。通过使用 table 构造器，这些存放在 Lua 代码中的数据可以像其他普通的文件一样看起来引人注目。

为了更清楚地描述问题，下面我们看看例子。如果我们的数据是预先确定的格式，比如 CSV（逗号分割值），我们几乎没得选择。（在第 20 章，我们介绍如何在 Lua 中处理 CSV 文件）。但是如果我们打算创建一个文件为了将来使用，除了 CSV，我们可以使用 Lua 构造器来我们表述我们数据，这种情况下，我们将每一个数据记录描述为一个 Lua 构造器。将下面的代码

```
Donald E. Knuth,Literate Programming,CSLI,1992
Jon Bentley,More Programming Pearls,Addison-Wesley,1990
```

写成

```
Entry{"Donald E. Knuth",
"Literate Programming",
"CSLI",
1992}

Entry{"Jon Bentley",
"More Programming Pearls",
"Addison-Wesley",
1990}
```

记住 Entry{...}与 Entry({...})等价，他是一个以表作为唯一参数的函数调用。所以，前面那段数据在 Lua 程序中表示如上。如果要读取这个段数据，我们只需要运行我们的 Lua 代码。例如下面这段代码计算数据文件中记录数：

```
local count = 0
function Entry (b) count = count + 1 end
```

```
dofile("data")
print("number of entries: " .. count)
```

下面这段程序收集一个作者名列表中的名字是否在数据文件中出现，如果在文件中出现则打印出来。（作者名字是 Entry 的第一个域；所以，如果 b 是一个 entry 的值，b[1] 则代表作者名）

```
local authors = {}       -- a set to collect authors
function Entry (b) authors[b[1]] = true end
dofile("data")
for name in pairs(authors) do print(name) end
```

注意，在这些程序段中使用事件驱动的方法：Entry 函数作为回调函数，dofile 处理数据文件中的每一记录都回调用它。当数据文件的大小不是太大的情况下，我们可以使用 name-value 对来描述数据：

```
Entry{
author = "Donald E. Knuth",
title = "Literate Programming",
publisher = "CSLI",
year = 1992
}

Entry{
author = "Jon Bentley",
title = "More Programming Pearls",
publisher = "Addison-Wesley",
year = 1990
}
```

（如果这种格式让你想起 BibTeX，这并不奇怪。Lua 中构造器正是根据来自 BibTeX 的灵感实现的）这种格式我们称之为自描述数据格式，因为每一个数据段都根据他的意思简短的描述为一种数据格式。相对 CSV 和其他紧缩格式，自描述数据格式更容易阅读和理解，当需要修改的时候可以容易的手工编辑，而且不需要改动数据文件。例如，如果我们想增加一个域，只需要对读取程序稍作修改即可，当指定的域不存在时，也可以赋予默认值。使用 name-value 对描述的情况下，上面收集作者名的代码可以改写为：

```
local authors = {} -- a set to collect authors
function Entry (b) authors[b.author] = true end
dofile("data")
for name in pairs(authors) do print(name) end
```

现在，记录域的顺序无关紧要了，甚至某些记录即使不存在 author 这个域，我们也只需要稍微改动一下代码即可：

```
function Entry (b)
if b.author then authors[b.author] = true end
end
```

Lua 不仅运行速度快，编译速度也快。例如，上面这段搜集作者名的代码处理一个2MB 的数据文件时间不会超过 1 秒。另外，这不是偶然的，数据描述是 Lua 的主要应用之一，从 Lua 发明以来，我们花了很多心血使他能够更快的编译和运行大的 chunks。

12.1 序列化

我们经常需要序列化一些数据，为了将数据转换为字节流或者字符流，这样我们就可以保存到文件或者通过网络发送出去。我们可以在 Lua 代码中描述序列化的数据，在这种方式下，我们运行读取程序即可从代码中构造出保存的值。

通常，我们使用这样的方式 varname = <exp>来保存一个全局变量的值。varname 部分比较容易理解，下面我们来看看如何写一个产生值的代码。对于一个数值来说：

```
function serialize (o)
    if type(o) == "number" then
        io.write(o)
    else ...
end
```

对于字符串值而言，原始的写法应该是：

```
if type(o) == "string" then
    io.write("'", o, "'")
```

然而，如果字符串包含特殊字符（比如引号或者换行符），产生的代码将不是有效的 Lua 程序。这时候你可能用下面方法解决特殊字符的问题：

```
if type(o) == "string" then
    io.write("[[", o, "]]")
```

千万不要这样做！双引号是针对手写的字符串的而不是针对自动产生的字符串。如果有人恶意的引导你的程序去使用"]]..os.execute('rm *')..[["这样的方式去保存某些东西（比如它可能提供字符串作为地址）你最终的 chunk 将是这个样子：

```
varname = [[ ]]..os.execute('rm *')..[[ ]]
```

如果你 load 这个数据，运行结果可想而知的。为了以安全的方式引用任意的字符串，string 标准库提供了格式化函数专门提供"%q"选项。它可以使用双引号表示字符串并且可以正确的处理包含引号和换行等特殊字符的字符串。这样一来，我们的序列化函数可以写为：

```
function serialize (o)
```

```
    if type(o) == "number" then
        io.write(o)
    elseif type(o) == "string" then
        io.write(string.format("%q", o))
    else ...
end
```

12.1.1 保存不带循环的 table

我们下一个艰巨的任务是保存表。根据表的结构不同，采取的方法也有很多。没有一种单一的算法对所有情况都能很好地解决问题。简单的表不仅需要简单的算法而且输出文件也需要看起来美观。

我们第一次尝试如下：

```
function serialize (o)
    if type(o) == "number" then
        io.write(o)
    elseif type(o) == "string" then
        io.write(string.format("%q", o))
    elseif type(o) == "table" then
        io.write("{\n")
        for k,v in pairs(o) do
            io.write(" ", k, " = ")
            serialize(v)
            io.write(",\n")
        end
        io.write("}\n")
    else
        error("cannot serialize a " .. type(o))
    end
end
```

尽管代码很简单，但很好地解决了问题。只要表结构是一个树型结构（也就是说，没有共享的子表并且没有循环），上面代码甚至可以处理嵌套表（表中表）。对于所进不整齐的表我们可以少作改进使结果更美观，这可以作为一个练习尝试一下。（提示：增加一个参数表示缩进的字符串，来进行序列化）。前面的函数假定表中出现的所有关键字都是合法的标示符。如果表中有不符合 Lua 语法的数字关键字或者字符串关键字，上面的代码将碰到麻烦。一个简单的解决这个难题的方法是将：

```
io.write(" ", k, " = ")
```

改为

```
io.write(" [")
serialize(k)
io.write("] = ")
```

这样一来，我们改善了我们的函数的健壮性，比较一下两次的结果：

```
-- result of serialize{a=12, b='Lua', key='another "one"'}
-- 第一个版本
{
a = 12,
b = "Lua",
key = "another \"one\"",
}

-- 第二个版本
{
["a"] = 12,
["b"] = "Lua",
["key"] = "another \"one\"",
}
```

我们可以通过测试每一种情况，看是否需要方括号，另外，我们将这个问题留作一个练习给大家。

12.1.2 保存带有循环的 table

针对普通拓扑概念上的带有循环表和共享子表的 table，我们需要另外一种不同的方法来处理。构造器不能很好地解决这种情况，我们不使用。为了表示循环我们需要将表名记录下来，下面我们的函数有两个参数：table 和对应的名字。另外，我们还必须记录已经保存过的 table 以防止由于循环而被重复保存。我们使用一个额外的 table 来记录保存过的表的轨迹，这个表的下表索引为 table，而值为对应的表名。

我们做一个限制：要保存的 table 只有一个字符串或者数字关键字。下面的这个函数序列化基本类型并返回结果。

```
function basicSerialize (o)
    if type(o) == "number" then
        return tostring(o)
    else       -- assume it is a string
        return string.format("%q", o)
    end
```

```
end
```

关键内容在接下来的这个函数，saved 这个参数是上面提到的记录已经保存的表的踪迹的 table。

```
function save (name, value, saved)
    saved = saved or {}         -- initial value
    io.write(name, " = ")
    if type(value) == "number" or type(value) == "string" then
        io.write(basicSerialize(value), "\n")
    elseif type(value) == "table" then
        if saved[value] then    -- value already saved?
            -- use its previous name
            io.write(saved[value], "\n")
        else
            saved[value] = name  -- save name for next time
            io.write("{}\n")     -- create a new table
            for k,v in pairs(value) do  -- save its fields
                local fieldname = string.format("%s[%s]", name,
                                                basicSerialize(k))
                save(fieldname, v, saved)
            end
        end
    else
        error("cannot save a " .. type(value))
    end
end
```

举个例子：

我们将要保存的 table 为：

```
a = {x=1, y=2; {3,4,5}}
a[2] = a       -- cycle
a.z = a[1]     -- shared sub-table
```

调用 save('a', a)之后结果为：

```
a = {}
a[1] = {}
a[1][1] = 3
a[1][2] = 4
a[1][3] = 5
```

```
a[2] = a
a["y"] = 2
a["x"] = 1
a["z"] = a[1]
```

（实际的顺序可能有所变化，它依赖于 table 遍历的顺序，不过，这个算法保证了一个新的定义中需要的前面的节点都已经被定义过）

如果我们想保存带有共享部分的表，我们可以使用同样 table 的 saved 参数调用 save 函数，例如我们创建下面两个表：

```
a = {{"one", "two"}, 3}
b = {k = a[1]}
```

保存它们：

```
save('a', a)
save('b', b)
```

结果将分别包含相同部分：

```
a = {}
a[1] = {}
a[1][1] = "one"
a[1][2] = "two"
a[2] = 3
b = {}
b["k"] = {}
b["k"][1] = "one"
b["k"][2] = "two"
```

然而如果我们使用同一个 saved 表来调用 save 函数：

```
local t = {}
save('a', a, t)
save('b', b, t)
```

结果将共享相同部分：

```
a = {}
a[1] = {}
a[1][1] = "one"
a[1][2] = "two"
a[2] = 3
b = {}
b["k"] = a[1]
```

上面这种方法是 Lua 中常用的方法，当然也有其他一些方法可以解决问题。比如，我们可以不使用全局变量名来保存，即使用封包，用 chunk 构造一个 local 值然后返回之；通过构造一张表，每张表名与其对应的函数对应起来等。Lua 给予你权力，由你决定如何实现。

第 13 章 Metatables and Metamethods

Lua 中的 table 由于定义的行为，我们可以对 key-value 对执行加操作，访问 key 对应的 value，遍历所有的 key-value。但是我们不可以对两个 table 执行加操作，也不可以比较两个表的大小。

Metatables 允许我们改变 table 的行为，例如，使用 Metatables 我们可以定义 Lua 如何计算两个 table 的相加操作 a+b。当 Lua 试图对两个表进行相加时，他会检查两个表是否有一个表有 Metatable，并且检查 Metatable 是否有__add 域。如果找到则调用这个__add 函数（所谓的 Metamethod）去计算结果。

Lua 中的每一个表都有其 Metatable。(后面我们将看到 userdata 也有 Metatable)，Lua 默认创建一个不带 metatable 的新表

```
t = {}
print(getmetatable(t))       --> nil
```

可以使用 setmetatable 函数设置或者改变一个表的 metatable

```
t1 = {}
setmetatable(t, t1)
assert(getmetatable(t) == t1)
```

任何一个表都可以是其他一个表的 metatable，一组相关的表可以共享一个 metatable（描述他们共同的行为）。一个表也可以是自身的 metatable（描述其私有行为）。

13.1 算术运算的 Metamethods

这一部分我们通过一个简单的例子介绍如何使用 metamethods。假定我们使用 table 来描述结合，使用函数来描述集合的并操作，交集操作，like 操作。我们在一个表内定义这些函数，然后使用构造函数创建一个集合：

```
Set = {}

function Set.new (t)
    local set = {}
    for _, l in ipairs(t) do set[l] = true end
    return set
```

```
end

function Set.union (a,b)
    local res = Set.new{}
    for k in pairs(a) do res[k] = true end
    for k in pairs(b) do res[k] = true end
    return res
end

function Set.intersection (a,b)
    local res = Set.new{}
    for k in pairs(a) do
        res[k] = b[k]
    end
    return res
end
```

为了帮助理解程序运行结果，我们也定义了打印函数输出结果：

```
function Set.tostring (set)
    local s = "{"
    local sep = ""
    for e in pairs(set) do
        s = s .. sep .. e
        sep = ", "
    end
    return s .. "}"
end

function Set.print (s)
    print(Set.tostring(s))
end
```

现在我们想加号运算符(+)执行两个集合的并操作，我们将所有集合共享一个 metatable，并且为这个 metatable 添加如何处理相加操作。

第一步，我们定义一个普通的表，用来作为 metatable。为避免污染命名空间，我们将其放在 set 内部。

```
Set.mt = {}       -- metatable for sets
```

第二步，修改 set.new 函数，增加一行，创建表的时候同时指定对应的 metatable。

```
function Set.new (t)     -- 2nd version
```

```
    local set = {}
    setmetatable(set, Set.mt)
    for _, l in ipairs(t) do set[l] = true end
    return set
end
```

这样一来，set.new 创建的所有的集合都有相同的 metatable 了：

```
s1 = Set.new{10, 20, 30, 50}
s2 = Set.new{30, 1}
print(getmetatable(s1))      --> table: 00672B60
print(getmetatable(s2))      --> table: 00672B60
```

第三步，给 metatable 增加__add 函数。

```
Set.mt.__add = Set.union
```

当 Lua 试图对两个集合相加时，将调用这个函数，以两个相加的表作为参数。

通过 metamethod，我们可以对两个集合进行相加：

```
s3 = s1 + s2
Set.print(s3)     --> {1, 10, 20, 30, 50}
```

同样的我们可以使用相乘运算符来定义集合的交集操作

```
Set.mt.__mul = Set.intersection

Set.print((s1 + s2)*s1)      --> {10, 20, 30, 50}
```

对于每一个算术运算符，metatable 都有对应的域名与其对应，除了__add、__mul 外，还有__sub(减)、__div(除)、__unm(负)、__pow(幂)，我们也可以定义__concat 定义连接行为。

当我们对两个表进行加没有问题，但如果两个操作数有不同的 metatable 例如：

```
s = Set.new{1,2,3}
s = s + 8
```

Lua 选择 metamethod 的原则：如果第一个参数存在带有__add 域的 metatable，Lua 使用它作为 metamethod，和第二个参数无关；

否则第二个参数存在带有__add 域的 metatable，Lua 使用它作为 metamethod 否则报错。

Lua 不关心这种混合类型的，如果我们运行上面的 s=s+8 的例子在 Set.union 发生错误：

```
bad argument #1 to `pairs' (table expected, got number)
```

如果我们想得到更加清楚地错误信息，我们需要自己显式的检查操作数的类型：

```
function Set.union (a,b)
    if getmetatable(a) ~= Set.mt or
                      getmetatable(b) ~= Set.mt then
        error("attempt to `add' a set with a non-set value", 2)
    end
        ...  -- same as before
```

13.2 关系运算的 Metamethods

Metatables 也允许我们使用 metamethods：__eq（等于），__lt（小于），和__le（小于等于）给关系运算符赋予特殊的含义。对剩下的三个关系运算符没有专门的 metamethod，因为 Lua 将 a ~= b 转换为 not (a == b)；a > b 转换为 b < a；a >= b 转换为 b <= a。

（直到 Lua 4.0 为止，所有的比较运算符被转换成一个，a <= b 转为 not (b < a)。然而这种转换并不一致正确。当我们遇到偏序（partial order）情况，也就是说，并不是所有的元素都可以正确的被排序情况。例如，在大多数机器上浮点数不能被排序，因为他的值不是一个数字（Not a Number 即 NaN）。根据 IEEE 754 的标准，NaN 表示一个未定义的值，比如 0/0 的结果。该标准指出任何涉及到 NaN 比较的结果都应为 false。也就是说，NaN <= x 总是 false，x < NaN 也总是 false。这样一来，在这种情况下 a <= b 转换为 not (b < a)就不再正确了。）

在我们关于基和操作的例子中，有类似的问题存在。<=代表集合的包含：a <= b 表示集合 a 是集合 b 的子集。这种意义下，可能 a <= b 和 b < a 都是 false；因此，我们需要将__le 和__lt 的实现分开：

```
Set.mt.__le = function (a,b)    -- set containment
    for k in pairs(a) do
        if not b[k] then return false end
    end
    return true
end

Set.mt.__lt = function (a,b)
    return a <= b and not (b <= a)
end
```

最后，我们通过集合的包含来定义集合相等：

```
Set.mt.__eq = function (a,b)
    return a <= b and b <= a
end
```

有了上面的定义之后，现在我们就可以来比较集合了：

```
s1 = Set.new{2, 4}
s2 = Set.new{4, 10, 2}
print(s1 <= s2)          --> true
print(s1 < s2)           --> true
print(s1 >= s1)          --> true
print(s1 > s1)           --> false
print(s1 == s2 * s1)     --> true
```

与算术运算的 metamethods 不同，关系元算的 metamethods 不支持混合类型运算。对于混合类型比较运算的处理方法和 Lua 的公共行为类似。如果你试图比较一个字符串和一个数字，Lua 将抛出错误。相似的，如果你试图比较两个带有不同 metamethods 的对象，Lua 也将抛出错误。

但相等比较从来不会抛出错误，如果两个对象有不同的 metamethod，比较的结果为 false，甚至可能不会调用 metamethod。这也是模仿了 Lua 的公共的行为，因为 Lua 总是认为字符串和数字是不等的，而不去判断它们的值。仅当两个有共同的 metamethod 的对象进行相等比较的时候，Lua 才会调用对应的 metamethod。

13.3 库定义的 Metamethods

在一些库中，在自己的 metatables 中定义自己的域是很普遍的情况。到目前为止，我们看到的所有 metamethods 都是 Lua 核心部分的。有虚拟机负责处理运算符涉及到的 metatables 和为运算符定义操作的 metamethods。但是，metatable 是一个普通的表，任何人都可以使用。

tostring 是一个典型的例子。如前面我们所见，tostring 以简单的格式表示出 table：

```
print({})      --> table: 0x8062ac0
```

（注意：print 函数总是调用 tostring 来格式化它的输出）。然而当格式化一个对象的时候，tostring 会首先检查对象是否存在一个带有__tostring 域的 metatable。如果存在则以对象作为参数调用对应的函数来完成格式化，返回的结果即为 tostring 的结果。

在我们集合的例子中我们已经定义了一个函数来将集合转换成字符串打印出来。因此，我们只需要将集合的 metatable 的__tostring 域调用我们定义的打印函数：

```
Set.mt.__tostring = Set.tostring
```

这样，不管什么时候我们调用 print 打印一个集合，print 都会自动调用 tostring，而 tostring 则会调用 Set.tostring：

```
s1 = Set.new{10, 4, 5}
print(s1)      --> {4, 5, 10}
```

setmetatable/getmetatable 函数也会使用 metafield，在这种情况下，可以保护

metatables。假定你想保护你的集合使其使用者既看不到也不能修改 metatables。如果你对 metatable 设置了__metatable 的值，getmetatable 将返回这个域的值，而调用 setmetatable 将会出错：

```
Set.mt.__metatable = "not your business"

s1 = Set.new{}
print(getmetatable(s1))      --> not your business
setmetatable(s1, {})
stdin:1: cannot change protected metatable
```

13.4 表相关的 Metamethods

关于算术运算和关系元运算的 metamethods 都定义了错误状态的行为，他们并不改变语言本身的行为。针对在两种正常状态：表的不存在的域的查询和修改，Lua 也提供了改变 tables 的行为的方法。

13.4.1 The __index Metamethod

前面说过，当我们访问一个表的不存在的域，返回结果为 nil，这是正确的，但并不一定正确。实际上，这种访问触发 lua 解释器去查找__index metamethod：如果不存在，返回结果为 nil；如果存在则由__index metamethod 返回结果。

这个例子的原型是一种继承。假设我们想创建一些表来描述窗口。每一个表必须描述窗口的一些参数，比如：位置，大小，颜色风格等等。所有的这些参数都有默认的值，当我们想要创建窗口的时候只需要给出非默认值的参数即可创建我们需要的窗口。第一种方法是，实现一个表的构造器，对这个表内的每一个缺少域都填上默认值。第二种方法是，创建一个新的窗口去继承一个原型窗口的缺少域。首先，我们实现一个原型和一个构造函数，他们共享一个 metatable：

```
-- create a namespace
Window = {}
-- create the prototype with default values
Window.prototype = {x=0, y=0, width=100, height=100, }
-- create a metatable
Window.mt = {}
-- declare the constructor function
function Window.new (o)
    setmetatable(o, Window.mt)
    return o
end
```

现在我们定义__index metamethod：

```
Window.mt.__index = function (table, key)
    return Window.prototype[key]
end
```

这样一来，我们创建一个新的窗口，然后访问他缺少的域结果如下：

```
w = Window.new{x=10, y=20}
print(w.width)        --> 100
```

当 Lua 发现 w 不存在域 width 时，但是有一个 metatable 带有__index 域，Lua 使用 w（the table）和 width（缺少的值）来调用__index metamethod，metamethod 则通过访问原型表（prototype）获取缺少的域的结果。

__index metamethod 在继承中的使用非常常见，所以 Lua 提供了一个更简洁的使用方式。__index metamethod 不需要非是一个函数，他也可以是一个表。但它是一个函数的时候，Lua 将 table 和缺少的域作为参数调用这个函数；当他是一个表的时候，Lua 将在这个表中看是否有缺少的域。所以，上面的那个例子可以使用第二种方式简单的改写为：

```
Window.mt.__index = Window.prototype
```

现在，当 Lua 查找 metatable 的__index 域时，他发现 window.prototype 的值，它是一个表，所以 Lua 将访问这个表来获取缺少的值，也就是说它相当于执行：

```
Window.prototype["width"]
```

将一个表作为__index metamethod 使用，提供了一种廉价而简单的实现单继承的方法。一个函数的代价虽然稍微高点，但提供了更多的灵活性：我们可以实现多继承，隐藏，和其他一些变异的机制。我们将在第 16 章详细的讨论继承的方式。

当我们想不通过调用__index metamethod 来访问一个表，我们可以使用 rawget 函数。Rawget(t,i)的调用以 raw access 方式访问表。这种访问方式不会使你的代码变快（the overhead of a function call kills any gain you could have），但有些时候我们需要他，在后面我们将会看到。

13.4.2 The __newindex Metamethod

__newindex metamethod 用来对表更新，__index 则用来对表访问。当你给表的一个缺少的域赋值，解释器就会查找__newindex metamethod：如果存在则调用这个函数而不进行赋值操作。像__index 一样，如果 metamethod 是一个表，解释器对指定的那个表，而不是原始的表进行赋值操作。另外，有一个 raw 函数可以绕过 metamethod：调用 rawset(t,k,v)不掉用任何 metamethod 对表 t 的 k 域赋值为 v。__index 和__newindex metamethods 的混合使用提供了强大的结构：从只读表到面向对象编程的带有继承默认值的表。在这一张的剩余部分我们看一些这些应用的例子，面向对象的编程在另外的章

节介绍。

13.4.3 有默认值的表

在一个普通的表中任何域的默认值都是 nil。很容易通过 metatables 来改变默认值：

```
function setDefault (t, d)
    local mt = {__index = function () return d end}
    setmetatable(t, mt)
end

tab = {x=10, y=20}
print(tab.x, tab.z)      --> 10   nil
setDefault(tab, 0)
print(tab.x, tab.z)      --> 10   0
```

现在，不管什么时候我们访问表的缺少的域，他的__index metamethod 被调用并返回 0。setDefault 函数为每一个需要默认值的表创建了一个新的 metatable。在有很多的表需要默认值的情况下，这可能使得花费的代价变大。然而 metatable 有一个默认值 d 和它本身关联，所以函数不能为所有表使用单一的一个 metatable。为了避免带有不同默认值的所有的表使用单一的 metatable，我们将每个表的默认值，使用一个唯一的域存储在表本身里面。如果我们不担心命名的混乱，我可使用像"___"作为我们的唯一的域：

```
local mt = {__index = function (t) return t.___ end}
function setDefault (t, d)
    t.___ = d
    setmetatable(t, mt)
end
```

如果我们担心命名混乱，也很容易保证这个特殊的键值唯一性。我们要做的只是创建一个新表用作键值：

```
local key = {}    -- unique key
local mt = {__index = function (t) return t[key] end}
function setDefault (t, d)
    t[key] = d
    setmetatable(t, mt)
end
```

另外一种解决表和默认值关联的方法是使用一个分开的表来处理，在这个特殊的表中索引是表，对应的值为默认值。然而这种方法的正确实现我们需要一种特殊的表：weak table，到目前为止我们还没有介绍这部分内容，将在第 17 章讨论。

为了带有不同默认值的表可以重用相同的原表，还有一种解决方法是使用 memoize

metatables，然而这种方法也需要 weak tables，所以我们再次不得不等到第 17 章。

13.4.4 监控表

__index 和__newindex 都是只有当表中访问的域不存在时候才起作用。捕获对一个表的所有访问情况的唯一方法就是保持表为空。因此，如果我们想监控一个表的所有访问情况，我们应该为真实的表创建一个代理。这个代理是一个空表，并且带有__index 和__newindex metamethods，由这两个方法负责跟踪表的所有访问情况并将其指向原始的表。假定，t 是我们想要跟踪的原始表，我们可以：

```
t = {}     -- original table (created somewhere)

-- keep a private access to original table
local _t = t

-- create proxy
t = {}

-- create metatable
local mt = {
    __index = function (t,k)
    print("*access to element " .. tostring(k))
    return _t[k]  -- access the original table
    end,

    __newindex = function (t,k,v)
    print("*update of element " .. tostring(k) ..
                " to " .. tostring(v))
    _t[k] = v     -- update original table
    end
}
setmetatable(t, mt)
```

这段代码将跟踪所有对 t 的访问情况：

```
> t[2] = 'hello'
*update of element 2 to hello
> print(t[2])
*access to element 2
hello
```

（注意：不幸的是，这个设计不允许我们遍历表。Pairs 函数将对 proxy 进行操作，

而不是原始的表。）如果我们想监控多张表，我们不需要为每一张表都建立一个不同的 metatable。我们只要将每一个 proxy 和他原始的表关联，所有的 proxy 共享一个公用的 metatable 即可。将表和对应的 proxy 关联的一个简单的方法是将原始的表作为 proxy 的域，只要我们保证这个域不用作其他用途。一个简单的保证它不被作他用的方法是创建一个私有的没有他人可以访问的 key。将上面的思想汇总，最终的结果如下：

```
-- create private index
local index = {}

-- create metatable
local mt = {
    __index = function (t,k)
        print("*access to element " .. tostring(k))
        return t[index][k]   -- access the original table
    end

    __newindex = function (t,k,v)
    print("*update of element " .. tostring(k) .. " to "
                    .. tostring(v))
    t[index][k] = v          -- update original table
    end
}

function track (t)
    local proxy = {}
    proxy[index] = t
    setmetatable(proxy, mt)
    return proxy
end
```

现在，不管什么时候我们想监控表 t，我们要做得只是 t=track(t)。

13.4.5 只读表

采用代理的思想很容易实现一个只读表。我们需要做得只是当我们监控到企图修改表时候抛出错误。通过__index metamethod，我们可以不使用函数而是用原始表本身来使用表，因为我们不需要监控查寻。这是比较简单并且高效的重定向所有查询到原始表的方法。但是，这种用法要求每一个只读代理有一个单独的新的 metatable，使用__index 指向原始表：

```
function readOnly (t)
```

```
    local proxy = {}
    local mt = {          -- create metatable
        __index = t,
        __newindex = function (t,k,v)
            error("attempt to update a read-only table", 2)
        end
    }

    setmetatable(proxy, mt)
    return proxy
end
```

（记住：error 的第二个参数 2，将错误信息返回给企图执行 update 的地方）作为一个简单的例子，我们对工作日建立一个只读表：

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

print(days[1])        --> Sunday
days[2] = "Noday"
stdin:1: attempt to update a read-only table
```

第 14 章 环境

Lua 用一个名为 environment 普通的表来保存所有的全局变量。（更精确的说，Lua 在一系列的 environment 中保存他的“global”变量，但是我们有时候可以忽略这种多样性）这种结果的优点之一是他简化了 Lua 的内部实现，因为对于所有的全局变量没有必要非要有不同的数据结构。另一个(主要的)优点是我们可以像其他表一样操作这个保存全局变量的表。为了简化操作，Lua 将环境本身存储在一个全局变量_G 中，（_G._G 等于_G）。例如，下面代码打印在当前环境中所有的全局变量的名字：

```
for n in pairs(_G) do print(n) end
```

这一章我们将讨论一些如何操纵环境的有用的技术。

14.1 使用动态名字访问全局变量

通常，赋值操作对于访问和修改全局变量已经足够。然而，我们经常需要一些原编程（meta-programming）的方式，比如当我们需要操纵一个名字被存储在另一个变量中的全局变量，或者需要在运行时才能知道的全局变量。为了获取这种全局变量的值，有的程序员可能写出下面类似的代码：

```
    loadstring("value = " .. varname)()
or
    value = loadstring("return " .. varname)()
```

如果 varname 是 x，上面连接操作的结果为："return x"（第一种形式为 "value = x"），当运行时才会产生最终的结果。然而这段代码涉及到一个新的 chunk 的创建和编译以及其他很多额外的问题。你可以换种方式更高效更简洁的完成同样的功能，代码如下：

```
value = _G[varname]
```

因为环境是一个普通的表，所以你可以使用你需要获取的变量（变量名）索引表即可。

也可以用相似的方式对一个全局变量赋值：_G[varname] = value。小心：一些程序员对这些函数很兴奋，并且可能写出这样的代码：_G["a"] = _G["var1"]，这只是 a = var1 的复杂的写法而已。

对前面的问题概括一下，表域可以是型如"io.read" or "a.b.c.d"的动态名字。我们用循环解决这个问题，从_G 开始，一个域一个域的遍历：

```
function getfield (f)
    local v = _G      -- start with the table of globals
```

```
    for w in string.gfind(f, "[%w_]+") do
        v = v[w]
    end
    return v
end
```

我们使用 string 库的 gfind 函数来迭代 f 中的所有单词（单词指一个或多个子母下划线的序列）。相对应的，设置一个域的函数稍微复杂些。赋值如：

```
a.b.c.d.e = v
```

实际等价于：

```
local temp = a.b.c.d
temp.e = v
```

也就是说，我们必须记住最后一个名字，必须独立的处理最后一个域。新的 setfield 函数当其中的域（译者注：中间的域肯定是表）不存在的时候还需要创建中间表。

```
function setfield (f, v)
    local t = _G         -- start with the table of globals
    for w, d in string.gfind(f, "([%w_]+)(.?)") do
        if d == "." then  -- not last field?
            t[w] = t[w] or {}     -- create table if absent
            t = t[w]          -- get the table
        else                  -- last field
            t[w] = v          -- do the assignment
        end
    end
end
```

这个新的模式匹配以变量 w 加上一个可选的点（保存在变量 d 中）的域。如果一个域名后面不允许跟上点，表明它是最后一个名字。（我们将在第 20 章讨论模式匹配问题）。使用上面的函数

```
setfield("t.x.y", 10)
```

创建一个全局变量表 t，另一个表 t.x，并且对 t.x.y 赋值为 10：

```
print(t.x.y)                  --> 10
print(getfield("t.x.y"))      --> 10
```

14.2 声明全局变量

全局变量不需要声明，虽然这对一些小程序来说很方便，但程序很大时，一个简单

的拼写错误可能引起 bug 并且很难发现。然而，如果我们喜欢，我们可以改变这种行为。因为 Lua 所有的全局变量都保存在一个普通的表中，我们可以使用 metatables 来改变访问全局变量的行为。

第一个方法如下：

```
setmetatable(_G, {
    __newindex = function (_, n)
        error("attempt to write to undeclared variable "..n, 2)
    end,

    __index = function (_, n)
        error("attempt to read undeclared variable "..n, 2)
    end,
})
```

这样一来，任何企图访问一个不存在的全局变量的操作都会引起错误：

```
> a = 1
stdin:1: attempt to write to undeclared variable a
```

但是我们如何声明一个新的变量呢？使用 rawset，可以绕过 metamethod：

```
function declare (name, initval)
    rawset(_G, name, initval or false)
end
```

or 带有 false 是为了保证新的全局变量不会为 nil。注意：你应该在安装访问控制以前（before installing the access control）定义这个函数，否则将得到错误信息：毕竟你是在企图创建一个新的全局声明。只要刚才那个函数在正确的地方，你就可以控制你的全局变量了：

```
> a = 1
stdin:1: attempt to write to undeclared variable a
> declare "a"
> a = 1       -- OK
```

但是现在，为了测试一个变量是否存在，我们不能简单的比较他是否为 nil。如果他是 nil 访问将抛出错误。所以，我们使用 rawget 绕过 metamethod：

```
if rawget(_G, var) == nil then
    -- 'var' is undeclared
    ...
end
```

改变控制允许全局变量可以为 nil 也不难，所有我们需要的是创建一个辅助表用来

保存所有已经声明的变量的名字。不管什么时候 metamethod 被调用的时候，他会检查这张辅助表看变量是否已经存在。代码如下：

```
local declaredNames = {}
function declare (name, initval)
    rawset(_G, name, initval)
    declaredNames[name] = true
end
setmetatable(_G, {
    __newindex = function (t, n, v)
    if not declaredNames[n] then
        error("attempt to write to undeclared var. "..n, 2)
    else
        rawset(t, n, v)   -- do the actual set
    end
end,
    __index = function (_, n)
    if not declaredNames[n] then
        error("attempt to read undeclared var. "..n, 2)
    else
        return nil
    end
end,
})
```

两种实现方式，代价都很小可以忽略不计的。第一种解决方法：metamethods 在平常操作中不会被调用。第二种解决方法：他们可能被调用，不过当且仅当访问一个值为 nil 的变量时。

14.3 非全局的环境

全局环境的一个问题是，任何修改都会影响你的程序的所有部分。例如，当你安装一个 metatable 去控制全局访问时，你的整个程序都必须遵循同一个指导方针。如果你想使用标准库，标准库中可能使用到没有声明的全局变量，你将碰到坏运。

Lua 5.0 允许每个函数可以有自己的环境来改善这个问题，听起来这很奇怪；毕竟，全局变量表的目的就是为了全局性使用。然而在 Section 15.4 我们将看到这个机制带来很多有趣的结构，全局的值依然是随处可以获取的。

可以使用 setfenv 函数来改变一个函数的环境。Setfenv 接受函数和新的环境作为参数。除了使用函数本身，还可以指定一个数字表示栈顶的活动函数。数字 1 代表当前函数，数字 2 代表调用当前函数的函数（这对写一个辅助函数来改变他们调用者的环境是

很方便的）依此类推。下面这段代码是企图应用 setfenv 失败的例子：

```
a = 1      -- create a global variable
-- change current environment to a new empty table
setfenv(1, {})
print(a)
```

导致：

```
stdin:5: attempt to call global `print' (a nil value)
```

（你必须在单独的 chunk 内运行这段代码，如果你在交互模式逐行运行他，每一行都是一个不同的函数，调用 setfenv 只会影响他自己的那一行。）一旦你改变了你的环境，所有全局访问都使用这个新的表，如果她为空，你就丢失所有你的全局变量，甚至_G，所以，你应该首先使用一些有用的值封装（populate）她，比如老的环境：

```
a = 1  -- create a global variable
-- change current environment
setfenv(1, {_G = _G})
_G.print(a)       --> nil
_G.print(_G.a)    --> 1
```

现在，当你访问"global" _G，他的值为旧的环境，其中你可以使用 print 函数。

你也可以使用继承封装(populate)你的新的环境：

```
a = 1
local newgt = {}      -- create new environment
setmetatable(newgt, {__index = _G})
setfenv(1, newgt)     -- set it
print(a)              --> 1
```

在这段代码新的环境从旧的环境中继承了 print 和 a；然而，任何赋值操作都对新表进行，不用担心误操作修改了全局变量表。另外，你仍然可以通过_G 修改全局变量：

```
-- continuing previous code
a = 10
print(a)       --> 10
print(_G.a)    --> 1
_G.a = 20
print(_G.a)    --> 20
```

当你创建一个新的函数时，他从创建他的函数继承了环境变量。所以，如果一个 chunk 改变了他自己的环境，这个 chunk 所有在改变之后定义的函数都共享相同的环境，都会受到影响。这对创建命名空间是非常有用的机制，我们下一章将会看到。

第 15 章 Packages

很多语言专门提供了某种机制组织全局变量的命名，比如 Modula 的 modules，Java 和 Perl 的 packages，C++的 namespaces。每一种机制对在 package 中声明的元素的可见性以及其他一些细节的使用都有不同的规则。但是他们都提供了一种避免不同库中命名冲突的问题的机制。每一个程序库创建自己的命名空间，在这个命名空间中定义的名字和其他命名空间中定义的名字互不干涉。

Lua 并没有提供明确的机制来实现 packages。然而，我们通过语言提供的基本的机制很容易实现他。主要的思想是：像标准库一样，使用表来描述 package。

使用表实现 packages 的明显的好处是：我们可以像其他表一样使用 packages，并且可以使用语言提供的所有的功能，带来很多便利。大多数语言中，packages 不是第一类值(first-class values)（也就是说，他们不能存储在变量里，不能作为函数参数。。。）因此，这些语言需要特殊的方法和技巧才能实现类似的功能。

Lua 中，虽然我们一直都用表来实现 pachages，但也有其他不同的方法可以实现 package，在这一章，我们将介绍这些方法。

15.1 基本方法

第一包的简单的方法是对包内的每一个对象前都加包名作为前缀。例如，假定我们正在写一个操作复数的库：我们使用表来表示复数，表有两个域 r（实数部分）和 i（虚数部分）。我们在另一张表中声明我们所有的操作来实现一个包：

```
complex = {}

function complex.new (r, i) return {r=r, i=i} end
-- defines a constant `i'
complex.i = complex.new(0, 1)

function complex.add (c1, c2)
    return complex.new(c1.r + c2.r, c1.i + c2.i)
end

function complex.sub (c1, c2)
    return complex.new(c1.r - c2.r, c1.i - c2.i)
end
```

```
function complex.mul (c1, c2)
    return complex.new(c1.r*c2.r - c1.i*c2.i,
           c1.r*c2.i + c1.i*c2.r)
end

function complex.inv (c)
    local n = c.r^2 + c.i^2
    return complex.new(c.r/n, -c.i/n)
end

return complex
```

这个库定义了一个全局名：coplex。其他的定义都是放在这个表内。

有了上面的定义，我们就可以使用符合规范的任何复数操作了，如：

```
c = complex.add(complex.i, complex.new(10, 20))
```

这种使用表来实现的包和真正的包的功能并不完全相同。首先，我们对每一个函数定义都必须显示的在前面加上包的名称。第二，同一包内的函数相互调用必须在被调用函数前指定包名。我们可以使用固定的局部变量名，来改善这个问题，然后，将这个局部变量赋值给最终的包。依据这个原则，我们重写上面的代码：

```
local P = {}
complex = P       -- package name
P.i = {r=0, i=1}
function P.new (r, i) return {r=r, i=i} end
function P.add (c1, c2)
    return P.new(c1.r + c2.r, c1.i + c2.i)
end
...
```

当在同一个包内的一个函数调用另一个函数的时候（或者她调用自身），他仍然需要加上前缀名。至少，它不再依赖于固定的包名。另外，只有一个地方需要包名。可能你注意到包中最后一个语句：

```
return complex
```

这个 return 语句并非必需的，因为 package 已经赋值给全局变量 complex 了。但是，我们认为 package 打开的时候返回本身是一个很好的习惯。额外的返回语句并不会花费什么代价，并且提供了另一种操作 package 的可选方式。

15.2 私有成员（Privacy）

有时候，一个 package 公开他的所有内容，也就是说，任何 package 的客户端都可以访问他。然而，一个 package 拥有自己的私有部分（也就是只有 package 本身才能访问）也是很有用的。在 Lua 中一个传统的方法是将私有部分定义为局部变量来实现。例如，我们修改上面的例子增加私有函数来检查一个值是否为有效的复数：

```
local P = {}
complex = P
local function checkComplex (c)
    if not ((type(c) == "table") and
    tonumber(c.r) and tonumber(c.i)) then
        error("bad complex number", 3)
    end
end

function P.add (c1, c2)
    checkComplex(c1);
    checkComplex(c2);
    return P.new(c1.r + c2.r, c1.i + c2.i)
end

...

return P
```

这种方式各有什么优点和缺点呢？package 中所有的名字都在一个独立的命名空间中。Package 中的每一个实体（entity）都清楚地标记为公有还是私有。另外，我们实现一个真正的隐私（privacy）：私有实体在 package 外部是不可访问的。缺点是访问同一个 package 内的其他公有的实体写法冗余，必须加上前缀 P.。还有一个大的问题是，当我们修改函数的状态(公有变成私有或者私有变成公有)我们必须修改函数得调用方式。

有一个有趣的方法可以立刻解决这两个问题。我们可以将 package 内的所有函数都声明为局部的，最后将他们放在最终的表中。按照这种方法，上面的 complex package 修改如下：

```
local function checkComplex (c)
    if not ((type(c) == "table")
    and tonumber(c.r) and tonumber(c.i)) then
        error("bad complex number", 3)
    end
```

```
end

local function new (r, i) return {r=r, i=i} end
local function add (c1, c2)
    checkComplex(c1);
    checkComplex(c2);
    return new(c1.r + c2.r, c1.i + c2.i)
end

...

complex = {
    new = new,
    add = add,
    sub = sub,
    mul = mul,
    div = div,
}
```

现在我们不再需要调用函数的时候在前面加上前缀，公有的和私有的函数调用方法相同。在 package 的结尾处，有一个简单的列表列出所有公有的函数。可能大多数人觉得这个列表放在 package 的开始处更自然，但我们不能这样做，因为我们必须首先定义局部函数。

15.3 包与文件

我们经常写一个 package 然后将所有的代码放到一个单独的文件中。然后我们只需要执行这个文件即加载 package。例如，如果我们将上面我们的复数的 package 代码放到一个文件 complex.lua 中，命令“require complex”将打开这个 package。记住 require 命令不会将相同的 package 加载多次。

需要注意的问题是，搞清楚保存 package 的文件名和 package 名的关系。当然，将他们联系起来是一个好的想法，因为 require 命令使用文件而不是 packages。一种解决方法是在 package 的后面加上后缀（比如.lua）来命名文件。Lua 并不需要固定的扩展名，而是由你的路径设置决定。例如，如果你的路径包含："/usr/local/lualibs/?.lua"，那么复数 package 可能保存在一个 complex.lua 文件中。

有些人喜欢先命名文件后命名 package。也就是说，如果你重命名文件，package 也会被重命名。这个解决方法提供了很大的灵活性。例如，如果你有两个有相同名称的 package，你不需要修改任何一个，只需要重命名一下文件。在 Lua 中我们使用 _REQUIREDNAME 变量来重命名。记住，当 require 加载一个文件的时候，它定义了一

个变量来表示虚拟的文件名。因此，在你的 package 中可以这样写：

```
local P = {}      -- package
if _REQUIREDNAME == nil then
    complex = P
else
    _G[_REQUIREDNAME] = P
end
```

代码中的 if 测试使得我们可以不需要 require 就可以使用 package。如果 _REQUIREDNAME 没有定义，我们用固定的名字表示 package（例子中 complex）。另外，package 使用虚拟文件名注册他自己。如果以使用者将库放到文件 cpx.lua 中并且运行 require cpx，那么 package 将本身加载到表 cpx 中。如果其他的使用者将库改名为 cpx_v1.lua 并且运行 require cpx_v1，那么 package 将自动将本身加载到表 cpx_v1 当中。

15.4 使用全局表

上面这些创建 package 的方法的缺点是：他们要求程序员注意很多东西，比如，在声明的时候也很容易忘掉 local 关键字。全局变量表的 Metamethods 提供了一些有趣的技术，也可以用来实现 package。这些技术中共同之处在于：package 使用独占的环境。这很容易实现：如果我们改变了 package 主 chunk 的环境，那么由 package 创建的所有函数都共享这个新的环境。

最简单的技术实现。一旦 package 有一个独占的环境，不仅所有她的函数共享环境，而且它的所有全局变量也共享这个环境。所以，我们可以将所有的公有函数声明为全局变量，然后他们会自动作为独立的表（表指 package 的名字）存在，所有 package 必须要做的是将这个表注册为 package 的名字。下面这段代码阐述了复数库使用这种技术的结果：

```
local P = {}
complex = P
setfenv(1, P)
```

现在，当我们声明函数 add,她会自动变成 complex.add:

```
function add (c1, c2)
    return new(c1.r + c2.r, c1.i + c2.i)
end
```

另外，我们可以在这个 package 中不需要前缀调用其他的函数。例如，add 函数调用 new 函数，环境会自动转换为 complex.new。这种方法提供了对 package 很好的支持：程序员几乎不需要做什么额外的工作，调用同一个 package 内的函数不需要前缀，调用公有和私有函数也没什么区别。如果程序员忘记了 local 关键字，也不会污染全局命名空间，

只不过使得私有函数变成公有函数而已。另外，我们可以将这种技术和前一节我们使用的 package 名的方法组合起来：

```
local P = {}      -- package
if _REQUIREDNAME == nil then
    complex = P
else
    _G[_REQUIREDNAME] = P
end
setfenv(1, P)
```

这样就不能访问其他的 packages 了。一旦我们将一个空表 P 作为我们的环境，我们就失去了访问所有以前的全局变量。下面有好几种方法可以解决这个问题，但都各有利弊。

最简单的解决方法是使用继承，像前面我们看到的一样：

```
local P = {}      -- package
setmetatable(P, {__index = _G})
setfenv(1, P)
```

（你必须在调用 setfenv 之前调用 setmetatable，你能说出原因么？）使用这种结构，package 就可以直接访问所有的全局标示符，但必须为每一个访问付出一小点代价。理论上来讲，这种解决方法带来一个有趣的结果：你的 package 现在包含了所有的全局变量。例如，使用你的 package 人也可以调用标准库的 sin 函数：complex.math.sin(x)。（Perl's package 系统也有这种特性）

另外一种快速的访问其他 packages 的方法是声明一个局部变量来保存老的环境：

```
local P = {}
pack = P
local _G = _G
setfenv(1, P)
```

现在，你必须对外部的访问加上前缀_G.，但是访问速度更快，因为这不涉及到 metamethod。与继承不同的是这种方法，使得你可以访问老的环境；这种方法的好与坏是有争议的，但是有时候你可能需要这种灵活性。

一个更加正规的方法是：只把你需要的函数或者 packages 声明为 local：

```
local P = {}
pack = P

-- Import Section:

-- declare everything this package needs from outside
```

```
local sqrt = math.sqrt
local io = io

-- no more external access after this point
setfenv(1, P)
```

这一技术要求稍多，但他使你的 package 的独立性比较好。他的速度也比前面那几种方法快。

15.5 其他一些技巧（Other Facilities）

正如前面我所说的，用表来实现 packages 过程中可以使用 Lua 的所有强大的功能。这里面有无限的可能性。在这里，我只给出一些建议。

我们不需要将 package 的所有公有成员的定义放在一起，例如，我们可以在一个独立分开的 chunk 中给我们的复数 package 增加一个新的函数：

```
function complex.div (c1, c2)
    return complex.mul(c1, complex.inv(c2))
end
```

（但是注意：私有成员必须限制在一个文件之内，我认为这是一件好事）反过来，我们可以在同一个文件之内定义多个 packages，我们需要做的只是将每一个 package 放在一个 do 代码块内，这样 local 变量才能被限制在那个代码块中。

在 package 外部，如果我们需要经常使用某个函数，我们可以给他们定义一个局部变量名：

```
local add, i = complex.add, complex.i
c1 = add(complex.new(10, 20), i)
```

如果我们不想一遍又一遍的重写 package 名，我们用一个短的局部变量表示 package：

```
local C = complex
c1 = C.add(C.new(10, 20), C.i)
```

写一个函数拆开 package 也是很容易的，将 package 中所有的名字放到全局命名空间即可：

```
function openpackage (ns)
    for n,v in pairs(ns) do _G[n] = v end
end

openpackage(complex)
c1 = mul(new(10, 20), i)
```

如果你担心打开 package 的时候会有命名冲突，可以在赋值以前检查一下名字是否存在：

```
function openpackage (ns)
    for n,v in pairs(ns) do
        if _G[n] ~= nil then
            error("name clash: " .. n .. " is already defined")
        end
        _G[n] = v
    end
end
```

由于 packages 本身也是表，我们甚至可以在 packages 中嵌套 packages；也就是说我们在一个 package 内还可以创建 package，然后很少有必要这么做。

另一个有趣之处是自动加载：函数只有被实际使用的时候才会自动加载。当我们加载一个自动加载的 package，会自动创建一个新的空表来表示 package 并且设置表的 __index metamethod 来完成自动加载。当我们调用任何一个没有被加载的函数的时候，__index metamethod 将被触发去加载着个函数。当调用发现函数已经被加载，__index 将不会被触发。

下面有一个简单的实现自动加载的方法。每一个函数定义在一个辅助文件中。(也可能一个文件内有多个函数)这些文件中的每一个都以标准的方式定义函数，例如：

```
function pack1.foo ()
    ...
end

function pack1.goo ()
    ...
end
```

然而，文件并不会创建 package，因为当函数被加载的时候 package 已经存在了。

在主 package 中我们定义一个辅助表来记录函数存放的位置：

```
local location = {
    foo = "/usr/local/lua/lib/pack1_1.lua",
    goo = "/usr/local/lua/lib/pack1_1.lua",
    foo1 = "/usr/local/lua/lib/pack1_2.lua",
    goo1 = "/usr/local/lua/lib/pack1_3.lua",
}
```

下面我们创建 package 并且定义她的 metamethod：

```
pack1 = {}
```

```
setmetatable(pack1, {__index = function (t, funcname)
local file = location[funcname]
if not file then
   error("package pack1 does not define " .. funcname)
end
assert(loadfile(file))()     -- load and run definition
return t[funcname]           -- return the function
end})

return pack1
```

加载这个 package 之后，第一次程序执行 pack1.foo()将触发__index metamethod，接着发现函数有一个相应的文件，并加载这个文件。微妙之处在于：加载了文件，同时返回函数作为访问的结果。

因为整个系统（译者：这里可能指复数吧？）都使用 Lua 写的，所以很容易改变系统的行为。例如，函数可以是用 C 写的，在 metamethod 中用 loadlib 加载他。或者我们我们可以在全局表中设定一个 metamethod 来自动加载整个 packages.这里有无限的可能等着你去发掘。

第 16 章 面向对象程序设计

Lua 中的表不仅在某种意义上是一种对象。像对象一样，表也有状态（成员变量）；也有与对象的值独立的本性，特别是拥有两个不同值的对象（table）代表两个不同的对象；一个对象在不同的时候也可以有不同的值，但他始终是一个对象；与对象类似，表的生命周期与其由什么创建、在哪创建没有关系。对象有他们的成员函数，表也有：

```
Account = {balance = 0}
function Account.withdraw (v)
    Account.balance = Account.balance - v
end
```

这个定义创建了一个新的函数，并且保存在 Account 对象的 withdraw 域内，下面我们可以这样调用：

```
Account.withdraw(100.00)
```

这种函数就是我们所谓的方法，然而，在一个函数内部使用全局变量名 Account 是一个不好的习惯。首先，这个函数只能在这个特殊的对象（译者：指 Account）中使用；第二，即使对这个特殊的对象而言，这个函数也只有在对象被存储在特殊的变量（译者：指 Account）中才可以使用。如果我们改变了这个对象的名字，函数 withdraw 将不能工作：

```
a = Account; Account = nil
a.withdraw(100.00)    -- ERROR!
```

这种行为违背了前面的对象应该有独立的生命周期的原则。

一个灵活的方法是：定义方法的时候带上一个额外的参数，来表示方法作用的对象。这个参数经常为 self 或者 this：

```
function Account.withdraw (self, v)
    self.balance = self.balance - v
end
```

现在，当我们调用这个方法的时候不需要指定他操作的对象了：

```
a1 = Account; Account = nil
...
a1.withdraw(a1, 100.00)      -- OK
```

使用 self 参数定义函数后，我们可以将这个函数用于多个对象上：

```
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要点。大多数 OO 语言将这种机制隐藏起来，这样程序员不必声明这个参数（虽然仍然可以在方法内使用这个参数）。Lua 也提供了通过使用冒号操作符来隐藏这个参数的声明。我们可以重写上面的代码：

```
function Account:withdraw (v)
    self.balance = self.balance - v
end
```

调用方法如下：

```
a:withdraw(100.00)
```

冒号的效果相当于在函数定义和函数调用的时候，增加一个额外的隐藏参数。这种方式只是提供了一种方便的语法，实际上并没有什么新的内容。我们可以使用 dot 语法定义函数而用冒号语法调用函数，反之亦然，只要我们正确的处理好额外的参数：

```
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)
```

现在我们的对象拥有一个标示符，一个状态和操作这个状态的方法。但他们依然缺少一个 class 系统，继承和隐藏。先解决第一个问题：我们如何才能创建拥有相似行为的多个对象呢？明确地说，我们怎样才能创建多个 accounts？（译者：针对上面的对象 Account 而言）

16.1 类

一些面向对象的语言中提供了类的概念，作为创建对象的模板。在这些语言里，对象是类的实例。Lua 不存在类的概念，每个对象定义他自己的行为并拥有自己的形状（shape）。然而，依据基于原型（prototype）的语言比如 Self 和 NewtonScript，在 Lua

中仿效类的概念并不难。在这些语言中，对象没有类。相反，每个对象都有一个 prototype（原型），当调用不属于对象的某些操作时，会最先会到 prototype 中查找这些操作。在这类语言中实现类（class）的机制，我们创建一个对象，作为其它对象的原型即可（原型对象为类，其它对象为类的 instance）。类与 prototype 的工作机制相同，都是定义了特定对象的行为。

在 Lua 中，使用前面章节我们介绍过的继承的思想，很容易实现 prototypes.更明确的来说，如果我们有两个对象 a 和 b，我们想让 b 作为 a 的 prototype 只需要：

```
setmetatable(a, {__index = b})
```

这样，对象 a 调用任何不存在的成员都会到对象 b 中查找。术语上，可以将 b 看作类，a 看作对象。回到前面银行账号的例子上。为了使得新创建的对象拥有和 Account 相似的行为，我们使用__index metamethod，使新的对象继承 Account。注意一个小的优化：我们不需要创建一个额外的表作为 account 对象的 metatable；我们可以用 Account 表本身作为 metatable：

```
function Account:new (o)
    o = o or {}   -- create object if user does not provide one
    setmetatable(o, self)
    self.__index = self
    return o
end
```

（当我们调用 Account:new 时，self 等于 Account；因此我们可以直接使用 Account 取代 self。然而，使用 self 在我们下一节介绍类继承时更合适）。有了这段代码之后，当我们创建一个新的账号并且掉用一个方法的时候，有什么发生呢？

```
a = Account:new{balance = 0}
a:deposit(100.00)
```

当我们创建这个新的账号 a 的时候，a 将 Account 作为他的 metatable（调用 Account:new 时，self 即 Account）。当我们调用 a:deposit(100.00)，我们实际上调用的是 a.deposit(a,100.00)（冒号仅仅是语法上的便利）。然而，Lua 在表 a 中找不到 deposit，因此他回到 metatable 的__index 对应的表中查找，情况大致如下：

```
getmetatable(a).__index.deposit(a, 100.00)
```

a 的 metatable 是 Account，Account.__index 也是 Account（因为 new 函数中 self.__index = self）。所以我们可以重写上面的代码为：

```
Account.deposit(a, 100.00)
```

也就是说，Lua 传递 a 作为 self 参数调用原始的 deposit 函数。所以，新的账号对象从 Account 继承了 deposit 方法。使用同样的机制，可以从 Account 继承所有的域。继承机制不仅对方法有效，对表中所有的域都有效。所以，一个类不仅提供方法，也提供了

他的实例的成员的默认值。记住：在我们第一个 Account 定义中，我们提供了成员 balance 默认值为 0，所以，如果我们创建一个新的账号而没有提供 balance 的初始值，他将继承默认值：

```
b = Account:new()
print(b.balance)      --> 0
```

当我们调用 b 的 deposit 方法时，实际等价于：

```
b.balance = b.balance + v
```

（因为 self 就是 b）。表达式 b.balance 等于 0 并且初始的存款（b.balance）被赋予 b.balance。下一次我们访问这个值的时候，不会在涉及到 index metamethod，因为 b 已经存在他自己的 balance 域。

16.2 继承

通常面向对象语言中，继承使得类可以访问其他类的方法，这在 Lua 中也很容易现实：

假定我们有一个基类 Account：

```
Account = {balance = 0}

function Account:new (o)
    o = o or {}
    setmetatable(o, self)
    self.__index = 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
```

我们打算从基类派生出一个子类 SpecialAccount，这个子类允许客户取款超过它的存款余额限制，我们从一个空类开始，从基类继承所有操作：

```
SpecialAccount = Account:new()
```

到现在为止，SpecialAccount 仅仅是 Account 的一个实例。现在奇妙的事情发生了：

```
s = SpecialAccount:new{limit=1000.00}
```

SpecialAccount 从 Account 继承了 new 方法，当 new 执行的时候，self 参数指向 SpecialAccount。所以，s 的 metatable 是 SpecialAccount，__index 也是 SpecialAccount。这样，s 继承了 SpecialAccount，后者继承了 Account。当我们执行：

```
s:deposit(100.00)
```

Lua 在 s 中找不到 deposit 域，他会到 SpecialAccount 中查找，在 SpecialAccount 中找不到，会到 Account 中查找。使得 SpecialAccount 特殊之处在于，它可以重定义从父类中继承来的方法：

```
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
```

现在，当我们调用方法 s:withdraw(200.00)，Lua 不会到 Account 中查找，因为它第一次救在 SpecialAccount 中发现了新的 withdraw 方法，由于 s.limit 等于 1000.00（记住我们创建 s 的时候初始化了这个值）程序执行了取款操作，s 的 balance 变成了负值。

在 Lua 中面向对象有趣的一个方面是你不需要创建一个新类去指定一个新的行为。如果仅仅一个对象需要特殊的行为，你可以直接在对象中实现，例如，如果账号 s 表示一些特殊的客户：取款限制是他的存款的 10%，你只需要修改这个单独的账号：

```
function s:getLimit ()
    return self.balance * 0.10
end
```

这样声明之后，调用 s:withdraw(200.00)将运行 SpecialAccount 的 withdraw 方法，但是当方法调用 self:getLimit 时，最后的定义被触发。

16.3 多重继承

由于 Lua 中的对象不是元生(primitive)的，所以在 Lua 中有很多方法可以实现面向对象的程序设计。我们前面所见到的使用 index metamethod 的方法可能是简洁、性能、灵活各方面综合最好的。然而，针对一些特殊情况也有更适合的实现方式。下面我们在 Lua

中多重继承的实现。

实现的关键在于：将函数用作__index。记住，当一个表的 metatable 存在一个__index 函数时，如果 Lua 调用一个原始表中不存在的函数，Lua 将调用这个__index 指定的函数。这样可以用__index 实现在多个父类中查找子类不存在的域。

多重继承意味着一个类拥有多个父类，所以，我们不能用创建一个类的方法去创建子类。取而代之的是，我们定义一个特殊的函数 createClass 来完成这个功能，将被创建的新类的父类作为这个函数的参数。这个函数创建一个表来表示新类，并且将它的 metatable 设定为一个可以实现多继承的__index metamethod。尽管是多重继承，每一个实例依然属于一个在其中能找得到它需要的方法的单独的类。所以，这种类和父类之间的关系与传统的类与实例的关系是有区别的。特别是，一个类不能同时是其实例的 metatable 又是自己的 metatable。在下面的实现中，我们将一个类作为他的实例的 metatable，创建另一个表作为类的 metatable：

```
-- look up for `k' in list of tables 'plist'
local function search (k, plist)
    for i=1, table.getn(plist) do
        local v = plist[i][k]    -- try 'i'-th superclass
        if v then return v end
    end
end

function createClass (...)
    local c = {}      -- new class

    -- class will search for each method in the list of its
    -- parents (`arg' is the list of parents)
    setmetatable(c, {__index = function (t, k)

    return search(k, arg)
end})

-- prepare `c' to be the metatable of its instances
c.__index = c

-- define a new constructor for this new class
function c:new (o)
    o = o or {}
    setmetatable(o, c)
    return o
```

```
end

-- return new class
    return c
end
```

让我们用一个小例子阐明一下 createClass 的使用，假定我们前面的类 Account 和另一个类 Named，Named 只有两个方法 setname and getname：

```
Named = {}
function Named:getname ()
    return self.name
end

function Named:setname (n)
    self.name = n
end
```

为了创建一个继承于这两个类的新类，我们调用 createClass：

```
NamedAccount = createClass(Account, Named)
```

为了创建和使用实例，我们像通常一样：

```
account = NamedAccount:new{name = "Paul"}
print(account:getname())     --> Paul
```

现在我们看看上面最后一句发生了什么，Lua 在 account 中找不到 getname，因此他查找 account 的 metatable 的__index，即 NamedAccount。但是，NamedAccount 也没有 getname，因此 Lua 查找 NamedAccount 的 metatable 的__index，因为这个域包含一个函数，Lua 调用这个函数并首先到 Account 中查找 getname，没有找到，然后到 Named 中查找，找到并返回最终的结果。当然，由于搜索的复杂性，多重继承的效率比起单继承要低。一个简单的改善性能的方法是将继承方法拷贝到子类。使用这种技术，index 方法如下：

```
...

setmetatable(c, {__index = function (t, k)
    local v = search(k, arg)
    t[k] = v      -- save for next access
    return v
end})

...
```

应用这个技巧，访问继承的方法和访问局部方法一样快（特别是第一次访问）。缺点是系统运行之后，很难改变方法的定义，因为这种改变不能影响继承链的下端。

16.4 私有性（privacy）

很多人认为私有性是面向对象语言的应有的一部分。每个对象的状态应该是这个对象自己的事情。在一些面向对象的语言中，比如 C++和 Java 你可以控制对象成员变量或者成员方法是否私有。其他一些语言比如 Smalltalk 中，所有的成员变量都是私有，所有的成员方法都是公有的。第一个面向对象语言 Simula 不提供任何保护成员机制。

如前面我们所看到的 Lua 中的主要对象设计不提供私有性访问机制。部分原因因为这是我们使用通用数据结构 tables 来表示对象的结果。但是这也反映了后来的 Lua 的设计思想。Lua 没有打算被用来进行大型的程序设计，相反，Lua 目标定于小型到中型的程序设计，通常是作为大型系统的一部分。典型的，被一个或者很少几个程序员开发，甚至被非程序员使用。所以，Lua 避免太冗余和太多的人为限制。如果你不想访问一个对象内的一些东西就不要访问（If you do not want to access something inside an object, just do not do it.）。

然而，Lua 的另一个目标是灵活性，提供程序员元机制（meta-mechanisms），通过他你可以实现很多不同的机制。虽然 Lua 中基本的面向对象设计并不提供私有性访问的机制，我们可以用不同的方式来实现他。虽然这种实现并不常用，但知道他也是有益的，不仅因为它展示了 Lua 的一些有趣的角落，也因为它可能是某些问题的很好地解决方案。设计的基本思想是，每个对象用两个表来表示：一个描述状态；另一个描述操作（或者叫接口）。对象本身通过第二个表来访问，也就是说，通过接口来访问对象。为了避免未授权的访问，表示状态的表中不涉及到操作；表示操作的表也不涉及到状态，取而代之的是，状态被保存在方法的闭包内。例如，用这种设计表述我们的银行账号，我们使用下面的函数工厂创建新的对象：

```
function newAccount (initialBalance)
    local self = {balance = initialBalance}
    local withdraw = function (v)
        self.balance = self.balance - v
    end

    local deposit = function (v)
        self.balance = self.balance + v
    end

    local getBalance = function () return self.balance end

    return {
```

```
        withdraw = withdraw,
        deposit = deposit,
        getBalance = getBalance
    }
end
```

首先，函数创建一个表用来描述对象的内部状态，并保存在局部变量 self 内。然后，函数为对象的每一个方法创建闭包（也就是说，嵌套的函数实例）。最后，函数创建并返回外部对象，外部对象中将局部方法名指向最终要实现的方法。这儿的关键点在于：这些方法没有使用额外的参数 self，代替的是直接访问 self。因为没有这个额外的参数，我们不能使用冒号语法来访问这些对象。函数只能像其他函数一样调用：

```
acc1 = newAccount(100.00)
acc1.withdraw(40.00)
print(acc1.getBalance())     --> 60
```

这种设计实现了任何存储在 self 表中的部分都是私有的，newAccount 返回之后，没有什么方法可以直接访问对象，我们只能通过 newAccount 中定义的函数来访问他。虽然我们的例子中仅仅将一个变量放到私有表中，但是我们可以将对象的任何的部分放到私有表中。我们也可以定义私有方法，他们看起来象公有的，但我们并不将其放到接口中。例如，我们的账号可以给某些用户取款享有额外的 10%的存款上限，但是我们不想用户直接访问这种计算的详细信息，我们实现如下：

```
function newAccount (initialBalance)
    local self = {
        balance = initialBalance,
        LIM = 10000.00,
    }

    local extra = function ()
        if self.balance > self.LIM then
            return self.balance*0.10
        else
            return 0
        end
    end

    local getBalance = function ()
        return self.balance + self.extra()
    end

    ...
```

这样，对于用户而言就没有办法直接访问 extra 函数了。

16.5 Single-Method 的对象实现方法

前面的 OO 程序设计的方法有一种特殊情况：对象只有一个单一的方法。这种情况下，我们不需要创建一个接口表，取而代之的是，我们将这个单一的方法作为对象返回。这听起来有些不可思议，如果需要可以复习一下 7.1 节，那里我们介绍了如何构造迭代子函数来保存闭包的状态。其实，一个保存状态的迭代子函数就是一个 single-method 对象。

关于 single-method 的对象一个有趣的情况是：当这个 single-method 实际是一个基于重要的参数而执行不同的任务的分派（dispatch）方法时。针对这种对象：

```
function newObject (value)
    return function (action, v)
        if action == "get" then return value
        elseif action == "set" then value = v
        else error("invalid action")
        end
    end
end
```

使用起来很简单：

```
d = newObject(0)
print(d("get"))       --> 0
d("set", 10)
print(d("get"))       --> 10
```

这种非传统的对象实现是非常有效的，语法 d("set",10)虽然很罕见，但也只不过比传统的 d:set(10)长两个字符而已。每一个对象是用一个单独的闭包，代价比起表来小的多。这种方式没有继承但有私有性：访问对象状态的唯一方式是通过它的内部方法。

Tcl/Tk 的窗口部件（widgets）使用了相似的方法，在 Tk 中一个窗口部件的名字表示一个在窗口部件上执行各种可能操作的函数（a widget command）。

第 17 章 Weak 表

Lua 自动进行内存的管理。程序只能创建对象（表，函数等），而没有执行删除对象的函数。通过使用垃圾收集技术，Lua 会自动删除那些失效的对象。这可以使你从内存管理的负担中解脱出来。更重要的，可以让你从那些由此引发的大部分 BUG 中解脱出来，比如指针挂起（dangling pointers）和内存溢出。

和其他的不同，Lua 的垃圾收集器不存在循环的问题。在使用循环性的数据结构的时候，你无须加入特殊的操作；他们会像其他数据一样被收集。当然，有些时候即使更智能化的收集器也需要你的帮助。没有任何的垃圾收集器可以让你忽略掉内存管理的所有问题。

垃圾收集器只能在确认对象失效之后才会进行收集；它是不会知道你对垃圾的定义的。一个典型的例子就是堆栈：有一个数组和指向栈顶的索引构成。你知道这个数组中有效的只是在顶端的那一部分，但 Lua 不那么认为。如果你通过简单的出栈操作提取一个数组元素，那么数组对象的其他部分对 Lua 来说仍然是有效的。同样的，任何在全局变量中声明的对象，都不是 Lua 认为的垃圾，即使你的程序中根本没有用到他们。这两种情况下，你应当自己处理它（你的程序），为这种对象赋 nil 值，防止他们锁住其他的空闲对象。

然而，简单的清理你的声明并不总是足够的。有些语句需要你和收集器进行额外的合作。一个典型的例子发生在当你想在你的程序中对活动的对象（比如文件）进行收集的时候。那看起来是个简单的任务：你需要做的是在收集器中插入每一个新的对象。然而，一旦对象被插入了收集器，它就不会再被收集！即使没有其他的指针指向它，收集器也不会做什么的。Lua 会认为这个引用是为了阻止对象被回收的，除非你告诉 Lua 怎么做。

Weak 表是一种用来告诉 Lua 一个引用不应该防止对象被回收的机制。一个 weak 引用是指一个不被 Lua 认为是垃圾的对象的引用。如果一个对象所有的引用指向都是 weak，对象将被收集，而那些 weak 引用将会被删除。Lua 通过 weak tables 来实现 weak 引用：一个 weak tables 是指所有引用都是 weak 的 table。这意味着，如果一个对象只存在于 weak tables 中，Lua 将会最终将它收集。

表有 keys 和 values，而这两者都可能包含任何类型的对象。在一般情况下，垃圾收集器并不会收集作为 keys 和 values 属性的对象。也就是说，keys 和 values 都属于强引用，他们可以防止他们指向的对象被回收。在一个 weak tables 中，keys 和 vaules 也可能是 weak 的。那意味着这里存在三种类型的 weak tables：weak keys 组成的 tables；weak values 组成的 tables；以及纯 weak tables 类型，他们的 keys 和 values 都是 weak 的。与 table 本身的类型无关，当一个 keys 或者 vaule 被收集时，整个的入口（entry）都将从这个 table 中消失。

表的 weak 性由他的 metatable 的__mode 域来指定的。在这个域存在的时候，必须是个字符串：如果这个字符串包含小写字母‘k’，这个 table 中的 keys 就是 weak 的；如果这个字符串包含小写字母‘v’，这个 table 中的 vaules 就是 weak 的。下面是一个例子，虽然是人造的，但是可以阐明 weak tables 的基本应用：

```
a = {}
b = {}
setmetatable(a, b)
b.__mode = \"k\"      -- now 'a' has weak keys

key = {}              -- creates first key
a[key] = 1

key = {}              -- creates second key
a[key] = 2

collectgarbage()      -- forces a garbage collection cycle

for k, v in pairs(a) do print(v) end
--> 2
```

在这个例子中，第二个赋值语句 key={}覆盖了第一个 key 的值。当垃圾收集器工作时，在其他地方没有指向第一个 key 的引用，所以它被收集了，因此相对应的 table 中的入口也同时被移除了。可是，第二个 key，仍然是占用活动的变量 key，所以它不会被收集。

要注意，只有对象才可以从一个 weak table 中被收集。比如数字和布尔值类型的值，都是不会被收集的。例如，如果我们在 table 中插入了一个数值型的 key（在前面那个例子中），它将永远不会被收集器从 table 中移除。当然，如果对应于这个数值型 key 的 vaule 被收集，那么它的整个入口将会从 weak table 中被移除。

关于字符串的一些细微差别：从上面的实现来看，尽管字符串是可以被收集的，他们仍然跟其他可收集对象有所区别。 其他对象，比如 tables 和函数，他们都是显示的被创建。比如，不管什么时候当 Lua 遇到{}时，它建立了一个新的 table。任何时候这个 function（）。。。end 建立了一个新的函数（实际上是一个闭包）。然而，Lua 见到“a”.. “b”的时候会创建一个新的字符串？如果系统中已经有一个字符串“ab”的话怎么办？Lua 会重新建立一个新的？编译器可以在程序运行之前创建字符串么？这无关紧要：这些是实现的细节。因此，从程序员的角度来看，字符串是值而不是对象。所以，就像数值或布尔值，一个字符串不会从 weak tables 中被移除（除非它所关联的 vaule 被收集）。

17.1 记忆函数

一个相当普遍的编程技术是用空间来换取时间。你可以通过记忆函数结果来进行优化，当你用同样的参数再次调用函数时，它可以自动返回记忆的结果。

想像一下一个通用的服务器，接收包含 Lua 代码的字符串请求。每当它收到一个请求，它调用 loadstring 加载字符串，然后调用函数进行处理。然而，loadstring 是一个“巨大”的函数，一些命令在服务器中会频繁地使用。不需要反复调用 loadstring 和后面接着的 closeconnection（），服务器可以通过使用一个辅助 table 来记忆 loadstring 的结果。在调用 loadstring 之前，服务器会在这个 table 中寻找这个字符串是否已经有了翻译好的结果。如果没有找到，那么（而且只是这个情况）服务器会调用 loadstring 并把这次的结果存入辅助 table。我们可以将这个操作包装为一个函数：

```
local results = {}
function mem_loadstring (s)
    if results[s] then       -- result available?
        return results[s]    -- reuse it
    else
        local res = loadstring(s)    -- compute new result
        results[s] = res             -- save for later reuse
        return res
    end
end
```

这个方案的存储消耗可能是巨大的。尽管如此，它仍然可能会导致意料之外的数据冗余。尽管一些命令一遍遍的重复执行，但有些命令可能只运行一次。渐渐地，这个 table 积累了服务器所有命令被调用处理后的结果；早晚有一天，它会挤爆服务器的内存。一个 weak table 提供了对于这个问题的简单解决方案。如果这个结果表中有 weak 值，每次的垃圾收集循环都会移除当前时间内所有未被使用的结果（通常是差不多全部）：

```
local results = {}
setmetatable(results, {__mode = \"v\"})    -- make values weak
function mem_loadstring (s)
    ...     -- as before
```

事实上，因为 table 的索引下标经常是字符串式的，如果愿意，我们可以将 table 全部置 weak：

```
setmetatable(results, {__mode = \"kv\"})
```

最终结果是完全一样的。

记忆技术在保持一些类型对象的唯一性上同样有用。例如，假如一个系统将通过 tables 表达颜色，通过有一定组合方式的红色，绿色，蓝色。一个自然颜色调色器通过

每一次新的请求产生新的颜色：

```
function createRGB (r, g, b)
    return {red = r, green = g, blue = b}
end
```

使用记忆技术，我们可以将同样的颜色结果存储在同一个 table 中。为了建立每一种颜色唯一的 key，我们简单的使用一个分隔符连接颜色索引下标：

```
local results = {}
setmetatable(results, {__mode = \"v\"})    -- make values weak
function createRGB (r, g, b)
    local key = r .. \"-\" .. g .. \"-\" .. b
    if results[key] then return results[key]
    else
        local newcolor = {red = r, green = g, blue = b}
        results[key] = newcolor
        return newcolor
    end
end
```

一个有趣的后果就是，用户可以使用这个原始的等号运算符比对操作来辨别颜色，因为两个同时存在的颜色通过同一个的 table 来表达。要注意，同样的颜色可能在不同的时间通过不同的 tales 来表达，因为垃圾收集器一次次的在清理结果 table。然而，只要给定的颜色正在被使用，它就不会从结果中被移除。所以，任何时候一个颜色在同其他颜色进行比较的时候存活的够久，它的结果镜像也同样存活。

17.2 关联对象属性

weak tables 的另一个重要的应用就是和对象的属性关联。在一个对象上加入更多的属性是无时无刻都会发生的： 函数名称，tables 的缺省值，数组的大小，等等。

当对象是表的时候，我们可以使用一个合适的唯一 key 来将属性保存在表中。就像我们在前面说的那样，一个很简单并且可以防止错误的方法是建立一个新的对象（典型的比如 table）然后把它当成 key 使用。然而，如果对象不是 table，它就不能自己保存自身的属性。即使是 tables，有些时候我们可能也不想把属性保存在原来的对象中去。例如，我们可能希望将属性作为私有的，或者我们不想在访问 table 中元素的时候受到这个额外的属性的干扰。在上述这些情况下，我们需要一个替代的方法来将属性和对象联系起来。当然，一个外部的 table 提供了一种理想化的方式来联系属性和对象（tables 有时被称作联合数组并不偶然）。我们把这个对象当作 key 来使用，他们的属性作为 vaule。一个外部的 table 可以保存任何类型对象的属性（就像 Lua 允许我们将任何对象看作 key）。此外，保存在一个外部 table 的属性不会妨碍到其他的对象，并且可以像这个 table

本身一样私有化。

然而，这个看起来完美的解决方案有一个巨大的缺点：一旦我们在一个 table 中将一个对象使用为 key，我们就将这个对象锁定为永久存在。Lua 不能收集一个正在被当作 key 使用的对象。如果我们使用一个普通的 table 来关联函数和名字，那么所有的这些函数将永远不会被收集。正如你所想的那样，我们可以通过使用 weak table 来解决这个问题。这一次，我们需要 weak keys。一旦没有其他地方的引用，weak keys 并不会阻止任何的 key 被收集。从另一方面说，这个 table 不会存在 weak vaules；否则，活动对象的属性就可能被收集了。

Lua 本身使用这种技术来保存数组的大小。像我们下面即将看到的那样，table 库提供了一个函数来设定数组的大小，另一个函数来读取数组的大小。当你设定了一个数组的大小，Lua 将这个尺寸保存在一个私有的 weak table，索引就是数组本身，而 value 就是它的尺寸。

17.3 重述带有默认值的表

在章节 13.4.3，我们讨论了怎样使用非 nil 的默认值来实现表。我们提到一种特殊的技术并注释说另外两种技术需要使用 weak tables，所以我们推迟在这里介绍他们。现在，介绍她们的时候了。就像我们说的那样，这两种默认值的技术实际上来源于我们前面提到的两种通用的技术的特殊应用：对象属性和记忆。

在第一种解决方案中，我们使用 weak table 来将默认 vaules 和每一个 table 相联系：

```
local defaults = {}
setmetatable(defaults, {__mode = \"k\"})
local mt = {__index = function (t) return defaults[t] end}

function setDefault (t, d)
    defaults[t] = d
    setmetatable(t, mt)
end
```

如果默认值没有 weak 的 keys，它就会将所有的带有默认值的 tables 设定为永久存在。在第二种方法中，我们使用不同的 metatables 来保存不同的默认值，但当我们重复使用一个默认值的时候，重用同一个相同的 metatable。这是一个典型的记忆技术的应用：

```
local metas = {}
setmetatable(metas, {__mode = \"v\"})

function setDefault (t, d)
    local mt = metas[d]
    if mt == nil then
```

```
        mt = {__index = function () return d end}
        metas[d] = mt     -- memoize
    end
    setmetatable(t, mt)
end
```

这种情况下，我们使用 weak vaules，允许将不会被使用的 metatables 可以被回收。

把这两种方法放在一起，哪个更好？通常，取决于具体情况。它们都有相似的复杂性和相似的性能。第一种方法需要在每个默认值的 tables 中添加一些文字（一个默认的入口）。第二种方法需要在每个不同的默认值加入一些文字（一个新的表，一个新的闭包，metas 中新增入口）。所以，如果你的程序有数千个 tables，而这些表只有很少数带有不同默认值的，第二种方法显然更优秀。另一方面，如果只有很少的 tabels 可以共享相同的默认 vaules，那么你还是用第一种方法吧。

第三篇 标准库

第 18 章 数学库

在这一章中（下面关于标准库的几章中同样）我的主要目的不是对每一个函数给出完整地说明，而是告诉你标准库能够提供什么功能。为了能够清楚地说明问题，我可能会忽略一些小的选项或者行为。主要的思想是激发你的好奇心，这些好奇之处可能在参考手册中找到答案。

数学库由算术函数的标准集合组成，比如三角函数库（sin, cos, tan, asin, acos, etc.），幂指函数（exp, log, log10），舍入函数（floor, ceil）、max、min，加上一个变量 pi。数学库也定义了一个幂操作符（^）。

所有的三角函数都在弧度单位下工作。（Lua4.0 以前在度数下工作。）你可以使用 deg 和 rad 函数在度和弧度之间转换。如果你想在 degree 情况下使用三角函数，你可以重定义三角函数：

```
local sin, asin, ... = math.sin, math.asin, ...
local deg, rad = math.deg, math.rad
math.sin = function (x) return sin(rad(x)) end
math.asin = function (x) return deg(asin(x)) end
...
```

math.random 用来产生伪随机数，有三种调用方式：

第一：不带参数，将产生 [0,1)范围内的随机数.

第二：带一个参数 n，将产生 1 <= x <= n 范围内的随机数 x.

第三：带两个参数 a 和 b,将产生 a <= x <= b 范围内的随机数 x.

你可以使用 randomseed 设置随机数发生器的种子，只能接受一个数字参数。通常在程序开始时，使用固定的种子初始化随机数发生器，意味着每次运行程序，将产生相同的随机数序列。为了调试方便，这很有好处，但是在游戏中就意味着每次运行都拥有相同的关卡。解决这个问题的一个通常的技巧是使用当前系统时间作为种子：

```
math.randomseed(os.time())
```

（os.time 函数返回一个表示当前系统时间的数字，通常是自新纪元以来的一个整数。）

第 19 章 Table 库

table 库由一些操作 table 的辅助函数组成。他的主要作用之一是对 Lua 中 array 的大小给出一个合理的解释。另外还提供了一些从 list 中插入删除元素的函数，以及对 array 元素排序函数。

19.1 数组大小

Lua 中我们经常假定 array 在最后一个非 nil 元素处结束。这个传统的约定有一个弊端：我们的 array 中不能拥有 nil 元素。对大部分应用来说这个限制不是什么问题，比如当所有的 array 有固定的类型的时候。但有些时候我们的 array 需要拥有 nil 元素，这种情况下，我们需要一种方法来明确的表明 array 的大小。

Table 库定义了两个函数操纵 array 的大小：getn，返回 array 的大小；setn，设置 array 的大小。如前面我们所见到的，这两个方法和 table 的一个属性相关：要么我们在 table 的一个域中保存这个属性，要么我们使用一个独立的（weak）table 来关联 table 和这个属性。两种方法各有利弊，所以 table 库使用了这两个方法。

通常，调用 table.setn(t, n)使得 t 和 n 在内部（weak）table 关联，调用 table.getn(t)将得到内部 table 中和 t 关联的那个值。然而，如果表 t 有一个带有数字值 n 的域，setn 将修改这个值，而 getn 返回这个值。Getn 函数还有一个选择：如果他不能使用上述方法返回 array 的大小，就会使用原始的方法：遍历 array 直到找到第一个 nil 元素。因此，你可以在 array 中一直使用 table.getn(t)获得正确的结果。看例子：

```
print(table.getn{10,2,4})          --> 3
print(table.getn{10,2,nil})        --> 2
print(table.getn{10,2,nil; n=3})   --> 3
print(table.getn{n=1000})          --> 1000

a = {}
print(table.getn(a))               --> 0
table.setn(a, 10000)
print(table.getn(a))               --> 10000

a = {n=10}
print(table.getn(a))               --> 10
table.setn(a, 10000)
print(table.getn(a))               --> 10000
```

默认的，setn 和 getn 使用内部表存储表的大小。这是最干净的选择，因为它不会使用额外的元素污染 array。然而，使用 n 域的方法也有一些优点。在带有可变参数的函数种，Lua 内核使用这种方法设置 arg 数组的大小，因为内核不依赖于库，他不能使用 setn。另外一个好处在于：我们可以在 array 创建的时候直接初始化他的大小，如我们在上面例子中看到的。

使用 setn 和 getn 操纵 array 的大小是个好的习惯，即使你知道大小在域 n 中。table 库中的所有函数（sort、concat、insert 等等）都遵循这个习惯。实际上，提供 setn 用来改变域 n 的值可能只是为了与旧的 lua 版本兼容，这个特性可能在将来的版本中改变，为了安全起见，不要假定依赖于这个特性。请一直使用 getn 获取数组大小，使用 setn 设置数组大小。

19.2 插入/删除

table 库提供了从一个 list 的任意位置插入和删除元素的函数。table.insert 函数在 array 指定位置插入一个元素，并将后面所有其他的元素后移。另外，insert 改变 array 的大小（using setn）。例如，如果 a 是一个数组{10,20,30}，调用 table.insert(a,1,15)后，a 变为{15,10,20,30}。经常使用的一个特殊情况是，我们不带位置参数调用 insert，将会在 array 最后位置插入元素（所以不需要元素移动）。下面的代码逐行独入程序，并将所有行保存在一个 array 内：

```
a = {}
for line in io.lines() do
    table.insert(a, line)
end
print(table.getn(a))        --> (number of lines read)
```

table.remove 函数删除数组中指定位置的元素，并返回这个元素，所有后面的元素前移，并且数组的大小改变。不带位置参数调用的时候，他删除 array 的最后一个元素。

使用这两个函数，很容易实现栈、队列和双端队列。我们可以初始化结构为 a={}。一个 push 操作等价于 table.insert(a,x)；一个 pop 操作等价于 table.remove(a)。要在结构的另一端结尾插入元素我们使用 table.insert(a,1,x)；删除元素用 table.remove(a,1)。最后两个操作不是特别有效的，因为他们必须来回移动元素。然而，因为 table 库这些函数使用 C 实现，对于小的数组(几百个元素)来说效率都不会有什么问题。

19.3 排序

另一个有用的函数是 table.sort。他有两个参数：存放元素的 array 和排序函数。排序函数有两个参数并且如果在 array 中排序后第一个参数在第二个参数前面，排序函数必须返回 true。如果未提供排序函数，sort 使用默认的小于操作符进行比较。

一个常见的错误是企图对表的下标域进行排序。在一个表中，所有下标组成一个集合，但是无序的。如果你想对他们排序，必须将他们复制到一个 array 然后对这个 array 排序。我们看个例子，假定上面的读取源文件并创建了一个表，这个表给出了源文件中每一个函数被定义的地方的行号：

```
lines = {
    luaH_set = 10,
    luaH_get = 24,
    luaH_present = 48,
}
```

现在你想以字母顺序打印出这些函数名，如果你使用 pairs 遍历这个表，函数名出现的顺序将是随机的。然而，你不能直接排序他们，因为这些名字是表的 key。当你将这些函数名放到一个数组内，就可以对这个数组进行排序。首先，必须创建一个数组来保存这些函数名，然后排序他们，最后打印出结果：

```
a = {}
for n in pairs(lines) do table.insert(a, n) end
table.sort(a)
for i,n in ipairs(a) do print(n) end
```

注意，对于 Lua 来说，数组也是无序的。但是我们知道怎样去计数，因此只要我们使用排序好的下标访问数组就可以得到排好序的函数名。这就是为什么我们一直使用 ipairs 而不是 pairs 遍历数组的原因。前者使用 key 的顺序 1、2、……，后者表的自然存储顺序。

有一个更好的解决方法，我们可以写一个迭代子来根据 key 值遍历这个表。一个可选的参数 f 可以指定排序的方式。首先，将排序的 keys 放到数组内，然后遍历这个数组，每一步从原始表中返回 key 和 value：

```
function pairsByKeys (t, f)
    local a = {}
    for n in pairs(t) do table.insert(a, n) end
    table.sort(a, f)
    local i = 0                 -- iterator variable
    local iter = function ()    -- iterator function
        i = i + 1
        if a[i] == nil then return nil
        else return a[i], t[a[i]]
        end
    end
    return iter
end
```

有了这个函数，很容易以字母顺序打印这些函数名，循环：

```
for name, line in pairsByKeys(lines) do
   print(name, line)
end
```

打印结果：

```
luaH_get          24
luaH_present      48
luaH_set          10
```

第 20 章 String 库

Lua 解释器对字符串的支持很有限。一个程序可以创建字符串并连接字符串，但不能截取子串，检查字符串的大小，检测字符串的内容。在 Lua 中操纵字符串的功能基本来自于 string 库。

String 库中的一些函数是非常简单的：string.len(s)返回字符串 s 的长度；string.rep(s, n)返回重复 n 次字符串 s 的串；你使用 string.rep("a", 2^20)可以创建一个 1M bytes 的字符串（比如，为了测试需要）；string.lower(s)将 s 中的大写字母转换成小写（string.upper 将小写转换成大写）。如果你想不关心大小写对一个数组进行排序的话，你可以这样：

```
table.sort(a, function (a, b)
    return string.lower(a) < string.lower(b)
end)
```

string.upper 和 string.lower 都依赖于本地环境变量。所以，如果你在 European Latin-1 环境下，表达式：

```
string.upper("a??o")
          --> "A??O".
```

调用 string.sub(s,i,j)函数截取字符串 s 的从第 i 个字符到第 j 个字符之间的串。Lua 中，字符串的第一个字符索引从 1 开始。你也可以使用负索引，负索引从字符串的结尾向前计数：-1 指向最后一个字符，-2 指向倒数第二个，以此类推。所以， string.sub(s, 1, j)返回字符串 s 的长度为 j 的前缀；string.sub(s, j, -1)返回从第 j 个字符开始的后缀。如果不提供第 3 个参数，默认为-1，因此我们将最后一个调用写为 string.sub(s, j)；string.sub(s, 2, -2)返回去除第一个和最后一个字符后的子串。

```
s = "[in brackets]"
print(string.sub(s, 2, -2))     --> in brackets
```

记住：Lua 中的字符串是恒定不变的。String.sub 函数以及 Lua 中其他的字符串操作函数都不会改变字符串的值，而是返回一个新的字符串。一个常见的错误是：

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

认为上面的这个函数会改变字符串 s 的值。如果你想修改一个字符串变量的值，你必须将变量赋给一个新的字符串：

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

string.char 函数和 string.byte 函数用来将字符在字符和数字之间转换。string.char 获取 0 个或多个整数，将每一个数字转换成字符，然后返回一个所有这些字符连接起来的字符串。string.byte(s, i)将字符串 s 的第 i 个字符的转换成整数；第二个参数是可选的，

缺省情况下 i=1。下面的例子中，我们假定字符用 ASCII 表示：

```
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
```

上面最后一行，我们使用负数索引访问字符串的最后一个字符。

函数 string.format 在用来对字符串进行格式化的时候，特别是字符串输出，是功能强大的工具。这个函数有两个参数，使用和 C 语言的 printf 函数几乎一模一样，你完全可以照 C 语言的 printf 来使用这个函数。第一个参数为格式化串：由指示符和控制格式的字符组成。指示符后的控制格式的字符可以为：十进制'd'；十六进制'x'；八进制'o'；浮点数'f'；字符串's'。在指示符'%'和控制格式字符之间还可以有其他的选项：用来控制更详细的格式，比如一个浮点数的小数的位数：

```
print(string.format("pi = %.4f", PI))
       --> pi = 3.1416
d = 5; m = 11; y = 1990
print(string.format("%02d/%02d/%04d", d, m, y))
       --> 05/11/1990
tag, title = "h1", "a title"
print(string.format("<%s>%s</%s>", tag, title, tag))
       --> <h1>a title</h1>
```

第一个例子，%.4f 代表小数点后面有 4 位小数的浮点数。第二个例子%02d 代表以固定的两位显示十进制数，不足的前面补 0。而%2d 前面没有指定 0，不足两位时会以空白补足。对于格式串部分指示符得详细描述清参考 lua 手册，或者参考 C 手册，因为 Lua 调用标准 C 的 printf 函数来实现最终的功能。

20.1 模式匹配函数

在 string 库中功能最强大的函数是：string.find（字符串查找），string.gsub（全局字符串替换），and string.gfind（全局字符串查找）。这些函数都是基于模式匹配的。

与其他脚本语言不同的是，Lua并不使用POSIX规范的正则表达式[4]（也写作regexp）来进行模式匹配。主要的原因出于程序大小方面的考虑：实现一个典型的符合POSIX标准的regexp大概需要 4000 行代码，这比整个Lua标准库加在一起都大。权衡之下，Lua中的模式匹配的实现只用了 500 行代码，当然这意味着不可能实现POSIX所规范的所有更能。然而，Lua中的模式匹配功能是很强大的，并且包含了一些使用标准POSIX模式匹

[4] 译注：POSIX 是 unix 的工业标准，regexp 最初来源于 unix，POSIX 对 regexp 也作了规范。

配不容易实现的功能。

string.find 的基本应用就是用来在目标串（subject string）内搜索匹配指定的模式的串。函数如果找到匹配的串返回他的位置，否则返回 nil.最简单的模式就是一个单词，仅仅匹配单词本身。比如，模式'hello'仅仅匹配目标串中的"hello"。当查找到模式的时候，函数返回两个值：匹配串开始索引和结束索引。

```
s = "hello world"
i, j = string.find(s, "hello")
print(i, j)                           --> 1    5
print(string.sub(s, i, j))            --> hello
print(string.find(s, "world"))        --> 7    11
i, j = string.find(s, "l")
print(i, j)                           --> 3    3
print(string.find(s, "lll"))          --> nil
```

例子中，匹配成功的时候，string.sub 利用 string.find 返回的值截取匹配的子串。（对简单模式而言，匹配的就是其本身）

string.find 函数第三个参数是可选的：标示目标串中搜索的起始位置。当我们想查找目标串中所有匹配的子串的时候，这个选项非常有用。我们可以不断的循环搜索，每一次从前一次匹配的结束位置开始。下面看一个例子，下面的代码用一个字符串中所有的新行构造一个表：

```
local t = {}      -- table to store the indices
local i = 0
while true do
    i = string.find(s, "\n", i+1)   -- find 'next' newline
    if i == nil then break end
    table.insert(t, i)
end
```

后面我们还会看到可以使用 string.gfind 迭代子来简化上面这个循环。

string.gsub 函数有三个参数：目标串，模式串，替换串。他基本作用是用来查找匹配模式的串，并将使用替换串其替换掉：

```
s = string.gsub("Lua is cute", "cute", "great")
print(s)       --> Lua is great
s = string.gsub("all lii", "l", "x")
print(s)       --> axx xii
s = string.gsub("Lua is great", "perl", "tcl")
print(s)       --> Lua is great
```

第四个参数是可选的，用来限制替换的范围：

```
s = string.gsub("all lii", "l", "x", 1)
print(s)          --> axl lii
s = string.gsub("all lii", "l", "x", 2)
print(s)          --> axx lii
```

string.gsub 的第二个返回值表示他进行替换操作的次数。例如，下面代码涌来计算一个字符串中空格出现的次数：

```
_, count = string.gsub(str, " ", " ")
```

（注意，_ 只是一个哑元变量）

20.2 模式

你还可以在模式串中使用字符类。字符类指可以匹配一个特定字符集合内任何字符的模式项。比如，字符类%d 匹配任意数字。所以你可以使用模式串'%d%d/%d%d/%d%d%d%d'搜索 dd/mm/yyyy 格式的日期：

```
s = "Deadline is 30/05/1999, firm"
date = "%d%d/%d%d/%d%d%d%d"
print(string.sub(s, string.find(s, date)))    --> 30/05/1999
```

下面的表列出了 Lua 支持的所有字符类：

```
.       任意字符
%a      字母
%c      控制字符
%d      数字
%l      小写字母
%p      标点字符
%s      空白符
%u      大写字母
%w      字母和数字
%x      十六进制数字
%z      代表 0 的字符
```

上面字符类的大写形式表示小写所代表的集合的补集。例如，'%A'非字母的字符：

```
print(string.gsub("hello, up-down!", "%A", "."))
    --> hello..up.down. 4
```

（数字 4 不是字符串结果的一部分，他是 gsub 返回的第二个结果，代表发生替换的次数。下面其他的关于打印 gsub 结果的例子中将会忽略这个数值。）在模式匹配中有一些特殊字符，他们有特殊的意义，Lua 中的特殊字符如下：

```
( ) . % + - * ? [ ^ $
```

'%' 用作特殊字符的转义字符，因此 '%.' 匹配点；'%%' 匹配字符 '%'。转义字符 '%' 不仅可以用来转义特殊字符，还可以用于所有的非字母的字符。当对一个字符有疑问的时候，为安全起见请使用转义字符转义他。

对 Lua 而言，模式串就是普通的字符串。他们和其他的字符串没有区别，也不会受到特殊对待。只有他们被用作模式串用于函数的时候，'%' 才作为转义字符。所以，如果你需要在一个模式串内放置引号的话，你必须使用在其他的字符串中放置引号的方法来处理，使用 '\' 转义引号，'\' 是 Lua 的转义符。你可以使用方括号将字符类或者字符括起来创建自己的字符类（译者：Lua 称之为 char-set，就是指传统正则表达式概念中的括号表达式）。比如，'[%w_]' 将匹配字母数字和下划线，'[01]' 匹配二进制数字，'[%[%]]' 匹配一对方括号。下面的例子统计文本中元音字母出现的次数：

```
_, nvow = string.gsub(text, "[AEIOUaeiou]", "")
```

在 char-set 中可以使用范围表示字符的集合，第一个字符和最后一个字符之间用连字符连接表示这两个字符之间范围内的字符集合。大部分的常用字符范围都已经预定义好了，所以一般你不需要自己定义字符的集合。比如，'%d' 表示 '[0-9]'；'%x' 表示 '[0-9a-fA-F]'。然而，如果你想查找八进制数，你可能更喜欢使用 '[0-7]' 而不是 '[01234567]'。你可以在字符集(char-set)的开始处使用 '^' 表示其补集：'[^0-7]' 匹配任何不是八进制数字的字符；'[^\n]' 匹配任何非换行符户的字符。记住，可以使用大写的字符类表示其补集：'%S' 比 '[^%s]' 要简短些。

Lua 的字符类依赖于本地环境，所以 '[a-z]' 可能与 '%l' 表示的字符集不同。在一般情况下，后者包括 'ç' 和 'ã'，而前者没有。应该尽可能的使用后者来表示字母，除非出于某些特殊考虑，因为后者更简单、方便、更高效。

可以使用修饰符来修饰模式增强模式的表达能力，Lua 中的模式修饰符有四个：

```
+       匹配前一字符 1 次或多次
*       匹配前一字符 0 次或多次
-       匹配前一字符 0 次或多次
?       匹配前一字符 0 次或 1 次
```

'+'，匹配一个或多个字符，总是进行最长的匹配。比如，模式串 '%a+' 匹配一个或多个字母或者一个单词：

```
print(string.gsub("one, and two; and three", "%a+", "word"))
    --> word, word word; word word
```

'%d+' 匹配一个或多个数字（整数）：

```
i, j = string.find("the number 1298 is even", "%d+")
print(i,j)     --> 12   15
```

'*' 与 '+' 类似，但是他匹配一个字符 0 次或多次出现.一个典型的应用是匹配空白。

比如，为了匹配一对圆括号()或者括号之间的空白，可以使用 '%(%s*%)'。（ '%s*' 用来匹配 0 个或多个空白。由于圆括号在模式中有特殊的含义，所以我们必须使用 '%' 转义他。）再看一个例子，'[_%a][_%w]*' 匹配 Lua 程序中的标示符：字母或者下划线开头的字母下划线数字序列。

'-' 与 '*' 一样，都匹配一个字符的 0 次或多次出现，但是他进行的是最短匹配。某些时候这两个用起来没有区别，但有些时候结果将截然不同。比如，如果你使用模式 '[_%a][_%w]-' 来查找标示符，你将只能找到第一个字母，因为 '[_%w]-' 永远匹配空。另一方面，假定你想查找 C 程序中的注释，很多人可能使用 '/%*.*%*/'（也就是说 "/*" 后面跟着任意多个字符，然后跟着 "*/" ）。然而，由于 '.*' 进行的是最长匹配，这个模式将匹配程序中第一个 "/*" 和最后一个 "*/" 之间所有部分：

```
test = "int x; /* x */  int y; /* y */"
print(string.gsub(test, "/%*.*%*/", "<COMMENT>"))
    --> int x; <COMMENT>
```

然而模式 '.-' 进行的是最短匹配，她会匹配 "/*" 开始到第一个 "*/" 之前的部分：

```
test = "int x; /* x */  int y; /* y */"
print(string.gsub(test, "/%*.-%*/", "<COMMENT>"))
    --> int x; <COMMENT>  int y; <COMMENT>
```

'?' 匹配一个字符 0 次或 1 次。举个例子，假定我们想在一段文本内查找一个整数，整数可能带有正负号。模式 '[+-]?%d+' 符合我们的要求，它可以匹配像 "-12"、"23" 和 "+1009" 等数字。'[+-]' 是一个匹配 '+' 或者 '-' 的字符类；接下来的 '?' 意思是匹配前面的字符类 0 次或者 1 次。

与其他系统的模式不同的是，Lua 中的修饰符不能用字符类；不能将模式分组然后使用修饰符作用这个分组。比如，没有一个模式可以匹配一个可选的单词（除非这个单词只有一个字母）。下面我将看到，通常你可以使用一些高级技术绕开这个限制。

以 '^' 开头的模式只匹配目标串的开始部分，相似的，以 '$' 结尾的模式只匹配目标串的结尾部分。这不仅可以用来限制你要查找的模式，还可以定位（anchor）模式。比如：

```
if string.find(s, "^%d") then ...
```

检查字符串 s 是否以数字开头，而

```
if string.find(s, "^[+-]?%d+$") then ...
```

检查字符串 s 是否是一个整数。

'%b' 用来匹配对称的字符。常写为 '%bxy' ，x 和 y 是任意两个不同的字符；x 作为匹配的开始，y 作为匹配的结束。比如，'%b()' 匹配以 '(' 开始，以 ')' 结束的字符串：

```
print(string.gsub("a (enclosed (in) parentheses) line",
                  "%b()", ""))
```

```
--> a  line
```

常用的这种模式有：'%b()' ，'%b[]'，'%b%{%}' 和 '%b<>'。你也可以使用任何字符作为分隔符。

20.3 捕获（Captures）

Capture[5]是这样一种机制：可以使用模式串的一部分匹配目标串的一部分。将你想捕获的模式用圆括号括起来，就指定了一个capture。

在 string.find 使用 captures 的时候，函数会返回捕获的值作为额外的结果。这常被用来将一个目标串拆分成多个：

```
pair = "name = Anna"
_, _, key, value = string.find(pair, "(%a+)%s*=%s*(%a+)")
print(key, value)     --> name   Anna
```

'%a+' 表示菲空的字母序列；'%s*' 表示 0 个或多个空白。在上面的例子中，整个模式代表：一个字母序列，后面是任意多个空白，然后是 '=' 再后面是任意多个空白，然后是一个字母序列。两个字母序列都是使用圆括号括起来的子模式，当他们被匹配的时候，他们就会被捕获。当匹配发生的时候，find 函数总是先返回匹配串的索引下标（上面例子中我们存储哑元变量 _ 中），然后返回子模式匹配的捕获部分。下面的例子情况类似：

```
date = "17/7/1990"
_, _, d, m, y = string.find(date, "(%d+)/(%d+)/(%d+)")
print(d, m, y)        --> 17  7  1990
```

我们可以在模式中使用向前引用，'%d'（d 代表 1-9 的数字）表示第 d 个捕获的拷贝。看个例子，假定你想查找一个字符串中单引号或者双引号引起来的子串，你可能使用模式 '["'].-["']'，但是这个模式对处理类似字符串 "it's all right" 会出问题。为了解决这个问题，可以使用向前引用，使用捕获的第一个引号来表示第二个引号：

```
s = [[then he said: "it's all right"!]]
a, b, c, quotedPart = string.find(s, "(["'])(.-)%1")
print(quotedPart)     --> it's all right
print(c)              --> "
```

第一个捕获是引号字符本身，第二个捕获是引号中间的内容（'.-' 匹配引号中间的子串）。

捕获值的第三个应用是用在函数 gsub 中。与其他模式一样，gsub 的替换串可以包

[5]译注：下面译为捕获或者 capture，模式中捕获的概念指，使用临时变量来保存匹配的子模式，常用于向前引用。

含 '%d'，当替换发生时他被转换为对应的捕获值。（顺便说一下，由于存在这些情况，替换串中的字符 '%' 必须用 "%%" 表示）。下面例子中，对一个字符串中的每一个字母进行复制，并用连字符将复制的字母和原字母连接起来：

```
print(string.gsub("hello Lua!", "(%a)", "%1-%1"))
    --> h-he-el-ll-lo-o L-Lu-ua-a!
```

下面代码互换相邻的字符:

```
print(string.gsub("hello Lua", "(.)(.)", "%2%1"))
    -->  ehll ouLa
```

让我们看一个更有用的例子，写一个格式转换器：从命令行获取 LaTeX 风格的字符串，形如：

```
\command{some text}
```

将它们转换为 XML 风格的字符串：

```
<command>some text</command>
```

对于这种情况,下面的代码可以实现这个功能：

```
s = string.gsub(s, "\\(%a+){(.-)}", "<%1>%2</%1>")
```

比如，如果字符串 s 为：

```
the \quote{task} is to \em{change} that.
```

调用 gsub 之后，转换为：

```
the <quote>task</quote> is to change that.
```

另一个有用的例子是去除字符串首尾的空格：

```
function trim (s)
    return (string.gsub(s, "^%s*(.-)%s*$", "%1"))
end
```

注意模式串的用法，两个定位符（'^' 和 '$'）保证我们获取的是整个字符串。因为，两个 '%s*' 匹配首尾的所有空格，'.-' 匹配剩余部分。还有一点需要注意的是 gsub 返回两个值，我们使用额外的圆括号丢弃多余的结果（替换发生的次数）。

最后一个捕获值应用之处可能是功能最强大的。我们可以使用一个函数作为 string.gsub 的第三个参数调用 gsub。在这种情况下，string.gsub 每次发现一个匹配的时候就会调用给定的作为参数的函数，捕获值可以作为被调用的这个函数的参数，而这个函数的返回值作为 gsub 的替换串。先看一个简单的例子，下面的代码将一个字符串中全局变量$varname 出现的地方替换为变量 varname 的值：

```
function expand (s)
    s = string.gsub(s, "$(%w+)", function (n)
```

```
        return _G[n]
    end)
    return s
end

name = "Lua"; status = "great"
print(expand("$name is $status, isn't it?"))

--> Lua is great, isn't it?
```

如果你不能确定给定的变量是否为 string 类型，可以使用 tostring 进行转换：

```
function expand (s)
    return (string.gsub(s, "$(%w+)", function (n)
        return tostring(_G[n])
    end))
end

print(expand("print = $print; a = $a"))

--> print = function: 0x8050ce0; a = nil
```

下面是一个稍微复杂点的例子，使用 loadstring 来计算一段文本内$后面跟着一对方括号内表达式的值：

```
s = "sin(3) = $[math.sin(3)]; 2^5 = $[2^5]"
print((string.gsub(s, "$(%b[])", function (x)
    x = "return " .. string.sub(x, 2, -2)
    local f = loadstring(x)
    return f()
end)))

-->  sin(3) = 0.1411200080598672; 2^5 = 32
```

第一次匹配是 "$[math.sin(3)]"，对应的捕获为 "$[math.sin(3)]"，调用 string.sub 去掉首尾的方括号，所以被加载执行的字符串是 "return math.sin(3)"，"$[2^5]" 的匹配情况类似。

我们常常需要使用 string.gsub 遍历字符串，而对返回结果不感兴趣。比如，我们收集一个字符串中所有的单词，然后插入到一个表中：

```
words = {}
string.gsub(s, "(%a+)", function (w)
    table.insert(words, w)
```

```
end)
```

如果字符串 s 为 "hello hi, again!"，上面代码的结果将是：

```
{"hello", "hi", "again"}
```

使用 string.gfind 函数可以简化上面的代码：

```
words = {}
for w in string.gfind(s, "(%a)") do
    table.insert(words, w)
end
```

gfind 函数比较适合用于范性 for 循环。他可以遍历一个字符串内所有匹配模式的子串。我们可以进一步的简化上面的代码，调用 gfind 函数的时候，如果不显示的指定捕获，函数将捕获整个匹配模式。所以，上面代码可以简化为：

```
words = {}
for w in string.gfind(s, "%a") do
    table.insert(words, w)
end
```

下面的例子我们使用 URL 编码，URL 编码是 HTTP 协议来用发送 URL 中的参数进行的编码。这种编码将一些特殊字符（比如 '='、'&'、'+'）转换为 "%XX" 形式的编码，其中 XX 是字符的 16 进制表示，然后将空白转换成 '+'。比如，将字符串 "a+b = c" 编码为 "a%2Bb+%3D+c"。最后，将参数名和参数值之间加一个 '='；在 name=value 对之间加一个 "&"。比如字符串：

```
name = "al";  query = "a+b = c";  q="yes or no"
```

被编码为：

```
name=al&query=a%2Bb+%3D+c&q=yes+or+no
```

现在，假如我们想讲这 URL 解码并把每个值存储到表中，下标为对应的名字。下面的函数实现了解码功能：

```
function unescape (s)
    s = string.gsub(s, "+", " ")
    s = string.gsub(s, "%%(%x%x)", function (h)
        return string.char(tonumber(h, 16))
    end)
    return s
end
```

第一个语句将 '+' 转换成空白，第二个 gsub 匹配所有的 '%' 后跟两个数字的 16 进制数，然后调用一个匿名函数，匿名函数将 16 进制数转换成一个数字（tonumber 在 16 进制情况下使用的）然后再转化为对应的字符。比如：

```
print(unescape("a%2Bb+%3D+c"))      --> a+b = c
```

对于 name=value 对，我们使用 gfind 解码，因为 names 和 values 都不能包含 '&' 和 '=' 我们可以用模式 '[^&=]+' 匹配他们：

```
cgi = {}
function decode (s)
    for name, value in string.gfind(s, "([^&=]+)=([^&=]+)") do
        name = unescape(name)
        value = unescape(value)
        cgi[name] = value
    end
end
```

调用 gfind 函数匹配所有的 name=value 对，对于每一个 name=value 对，迭代子将其相对应的捕获的值返回给变量 name 和 value。循环体内调用 unescape 函数解码 name 和 value 部分，并将其存储到 cgi 表中。

与解码对应的编码也很容易实现。首先，我们写一个 escape 函数，这个函数将所有的特殊字符转换成 '%' 后跟字符对应的 ASCII 码转换成两位的 16 进制数字（不足两位，前面补 0），然后将空白转换为 '+'：

```
function escape (s)
    s = string.gsub(s, "([&=+%c])", function (c)
        return string.format("%%%02X", string.byte(c))
    end)
    s = string.gsub(s, " ", "+")
    return s
end
```

编码函数遍历要被编码的表，构造最终的结果串：

```
function encode (t)
    local s = ""
    for k,v in pairs(t) do
        s = s .. "&" .. escape(k) .. "=" .. escape(v)
    end
    return string.sub(s, 2)      -- remove first `&'
end
t = {name = "al",  query = "a+b = c", q="yes or no"}

print(encode(t)) --> q=yes+or+no&query=a%2Bb+%3D+c&name=al
```

20.4 转换的技巧（Tricks of the Trade）

模式匹配对于字符串操纵来说是强大的工具，你可能只需要简单的调用 string.gsub 和 find 就可以完成复杂的操作，然而，因为它功能强大你必须谨慎的使用它，否则会带来意想不到的结果。

对正常的解析器而言，模式匹配不是一个替代品。对于一个 quick-and-dirty 程序，你可以在源代码上进行一些有用的操作，但很难完成一个高质量的产品。前面提到的匹配 C 程序中注释的模式是个很好的例子：'/%*.-%*/'。如果你的程序有一个字符串包含了"/*"，最终你将得到错误的结果：

```
test = [[char s[] = "a /* here";  /* a tricky string */]]
print(string.gsub(test, "/%*.-%*/", "<COMMENT>"))
    --> char s[] = "a <COMMENT>
```

虽然这样内容的字符串很罕见，如果是你自己使用的话上面的模式可能还凑活。但你不能将一个带有这种毛病的程序作为产品出售。

一般情况下，Lua 中的模式匹配效率是不错的：一个奔腾 333MHz 机器在一个有 200K 字符的文本内匹配所有的单词(30K 的单词)只需要 1/10 秒。但是你不能掉以轻心，应该一直对不同的情况特殊对待，尽可能的更明确的模式描述。一个限制宽松的模式比限制严格的模式可能慢很多。一个极端的例子是模式 '(.-)%$' 用来获取一个字符串内$符号以前所有的字符，如果目标串中存在$符号，没有什么问题；但是如果目标串中不存在$符号。上面的算法会首先从目标串的第一个字符开始进行匹配，遍历整个字符串之后没有找到$符号，然后从目标串的第二个字符开始进行匹配，……这将花费原来平方次幂的时间，导致在一个奔腾 333MHz 的机器中需要 3 个多小时来处理一个 200K 的文本串。可以使用下面这个模式避免上面的问题 '^(.-)%$'。定位符^告诉算法如果在第一个位置没有没找到匹配的子串就停止查找。使用这个定位符之后，同样的环境也只需要不到 1/10 秒的时间。

也需要小心空模式：匹配空串的模式。比如，如果你打算用模式 '%a*' 匹配名字，你会发现到处都是名字：

```
i, j = string.find(";$%  **#$hello13", "%a*")
print(i,j)     --> 1  0
```

这个例子中调用 string.find 正确的在目标串的开始处匹配了空字符。永远不要写一个以 '-' 开头或者结尾的模式，因为它将匹配空串。这个修饰符得周围总是需要一些东西来定位他的扩展。相似的，一个包含 '.*' 的模式是一个需要注意的，因为这个结构可能会比你预算的扩展的要多。

有时候，使用 Lua 本身构造模式是很有用的。看一个例子，我们查找一个文本中行字符大于 70 个的行，也就是匹配一个非换行符之前有 70 个字符的行。我们使用字符类'[^\n]'表示非换行符的字符。所以，我们可以使用这样一个模式来满足我们的需要：重复

匹配单个字符的模式 70 次，后面跟着一个匹配一个字符 0 次或多次的模式。我们不手工来写这个最终的模式，而使用函数 string.rep：

```
pattern = string.rep("[^\n]", 70) .. "[^\n]*"
```

另一个例子，假如你想进行一个大小写无关的查找。方法之一是将任何一个字符 x 变为字符类 '[xX]'。我们也可以使用一个函数进行自动转换：

```
function nocase (s)
    s = string.gsub(s, "%a", function (c)
        return string.format("[%s%s]", string.lower(c),
                                        string.upper(c))
    end)
    return s
end

print(nocase("Hi there!"))
    -->  [hH][iI] [tT][hH][eE][rR][eE]!
```

有时候你可能想要将字符串 s1 转化为 s2，而不关心其中的特殊字符。如果字符串 s1 和 s2 都是字符串序列，你可以给其中的特殊字符加上转义字符来实现。但是如果这些字符串是变量呢，你可以使用 gsub 来完成这种转义：

```
s1 = string.gsub(s1, "(%W)", "%%%1")
s2 = string.gsub(s2, "%%", "%%%%")
```

在查找串中，我们转义了所有的非字母的字符。在替换串中，我们只转义了 '%' 。另一个对模式匹配而言有用的技术是在进行真正处理之前，对目标串先进行预处理。一个预处理的简单例子是，将一段文本内的双引号内的字符串转换为大写，但是要注意双引号之间可以包含转义的引号（"""）：

这是一个典型的字符串例子：

```
"This is "great"!".
```

我们处理这种情况的方法是，预处理文本把有问题的字符序列转换成其他的格式。比如，我们可以将 """ 编码为 "\1"，但是如果原始的文本中包含 "\1"，我们又陷入麻烦之中。一个避免这个问题的简单的方法是将所有 "\x" 类型的编码为 "\ddd"，其中 ddd 是字符 x 的十进制表示：

```
function code (s)
    return (string.gsub(s, "\\(.)", function (x)
        return string.format("\\%03d", string.byte(x))
    end))
end
```

注意，原始串中的 "\ddd" 也会被编码，解码是很容易的：

```
function decode (s)
    return (string.gsub(s, "\\(%d%d%d)", function (d)
        return "\" .. string.char(d)
    end))
end
```

如果被编码的串不包含任何转义符，我们可以简单的使用 '".-"' 来查找双引号字符串：

```
s = [[follows a typical string: "This is "great"!".]]
s = code(s)
s = string.gsub(s, '(".-")', string.upper)
s = decode(s)
print(s)
    --> follows a typical string: "THIS IS "GREAT"!".
```

更紧缩的形式：

```
print(decode(string.gsub(code(s), '(".-")', string.upper)))
```

我们回到前面的一个例子，转换\command{string}这种格式的命令为 XML 风格：

```
<command>string</command>
```

但是这一次我们原始的格式中可以包含反斜杠作为转义符，这样就可以使用"\"、"\{" 和 "\}"，分别表示 '\'、'{' 和 '}'。为了避免命令和转义的字符混合在一起，我们应该首先将原始串中的这些特殊序列重新编码，然而，与上面的一个例子不同的是，我们不能转义所有的 \x，因为这样会将我们的命令（\command）也转换掉。这里，我们仅当 x 不是字符的时候才对 \x 进行编码：

```
function code (s)
    return (string.gsub(s, '\\(%A)', function (x)
        return string.format("\\%03d", string.byte(x))
    end))
end
```

解码部分和上面那个例子类似，但是在最终的字符串中不包含反斜杠，所以我们可直接调用 string.char：

```
function decode (s)
    return (string.gsub(s, '\\(%d%d%d)', string.char))
end

s = [[a \emph{command} is written as \\command\{text\}.]]
s = code(s)
s = string.gsub(s, "\\(%a+){(.-)}", "<%1>%2</%1>")
```

```
print(decode(s))
-->  a <emph>command</emph> is written as \command{text}.
```

我们最后一个例子是处理 CSV（逗号分割）的文件，很多程序都使用这种格式的文本，比如 Microsoft Excel。CSV 文件十多条记录的列表，每一条记录一行，一行内值与值之间逗号分割，如果一个值内也包含逗号这个值必须用双引号引起来，如果值内还包含双引号，需使用双引号转义双引号（就是两个双引号表示一个），看例子，下面的数组：

```
{'a b', 'a,b', 'a,"b"c', 'hello "world"!', }
```

可以看作为：

```
a b,"a,b"," a,""b""c", hello "world"!,
```

将一个字符串数组转换为 CSV 格式的文件是非常容易的。我们要做的只是使用逗号将所有的字符串连接起来：

```
function toCSV (t)
    local s = ""
    for _,p in pairs(t) do
        s = s .. "," .. escapeCSV(p)
    end
    return string.sub(s, 2)      -- remove first comma
end
```

如果一个字符串包含逗号活着引号在里面，我们需要使用引号将这个字符串引起来，并转义原始的引号：

```
function escapeCSV (s)
    if string.find(s, '[,"]') then
        s = '"' .. string.gsub(s, '"', '""') .. '"'
    end
    return s
end
```

将 CSV 文件内容存放到一个数组中稍微有点难度，因为我们必须区分出位于引号中间的逗号和分割域的逗号。我们可以设法转义位于引号中间的逗号，然而并不是所有的引号都是作为引号存在，只有在逗号之后的引号才是一对引号的开始的那一个。只有不在引号中间的逗号才是真正的逗号。这里面有太多的细节需要注意，比如，两个引号可能表示单个引号，可能表示两个引号，还有可能表示空：

```
"hello""hello", "",""
```

这个例子中，第一个域是字符串 "hello"hello",第二个域是字符串 " """（也就是一个空白加两个引号），最后一个域是一个空串。

我们可以多次调用 gsub 来处理这些情况，但是对于这个任务使用传统的循环（在每个域上循环）来处理更有效。循环体的主要任务是查找下一个逗号；并将域的内容存放到一个表中。对于每一个域，我们循环查找封闭的引号。循环内使用模式 '"("?)' 来查找一个域的封闭的引号：如果一个引号后跟着一个引号，第二个引号将被捕获并赋给一个变量 c，意味着这仍然不是一个封闭的引号

```
function fromCSV (s)
    s = s .. ','        -- ending comma
    local t = {}        -- table to collect fields
    local fieldstart = 1
    repeat
        -- next field is quoted? (start with `"'?)
        if string.find(s, '^"', fieldstart) then
        local a, c
        local i  = fieldstart
        repeat
            -- find closing quote
            a, i, c = string.find(s, '"("?)', i+1)
        until c ~= '"'    -- quote not followed by quote?
        if not i then error('unmatched "') end
            local f = string.sub(s, fieldstart+1, i-1)
            table.insert(t, (string.gsub(f, '""', '"')))
            fieldstart = string.find(s, ',', i) + 1
        else                -- unquoted; find next comma
            local nexti = string.find(s, ',', fieldstart)
            table.insert(t, string.sub(s, fieldstart,
                                       nexti-1))
            fieldstart = nexti + 1
        end
    until fieldstart > string.len(s)
    return t
end

t = fromCSV('"hello "" hello", "",""')
for i, s in ipairs(t) do print(i, s) end
    --> 1      hello " hello
    --> 2       ""
    --> 3
```

第 21 章 IO 库

I/O 库为文件操作提供两种模式。简单模式（simple model）拥有一个当前输入文件和一个当前输出文件，并且提供针对这些文件相关的操作。完全模式（complete model）使用外部的文件句柄来实现。它以一种面对对象的形式，将所有的文件操作定义为文件句柄的方法。简单模式在做一些简单的文件操作时较为合适。在本书的前面部分我们一直都在使用它。但是在进行一些高级的文件操作的时候，简单模式就显得力不从心。例如同时读取多个文件这样的操作，使用完全模式则较为合适。I/O 库的所有函数都放在表（table）io 中。

21.1 简单 I/O 模式

简单模式的所有操作都是在两个当前文件之上。I/O 库将当前输入文件作为标准输入（stdin），将当前输出文件作为标准输出（stdout）。这样当我们执行 io.read，就是在标准输入中读取一行。我们可以使用 io.input 和 io.output 函数来改变当前文件。例如 io.input(filename)就是打开给定文件（以读模式），并将其设置为当前输入文件。接下来所有的输入都来自于该文，直到再次使用 io.input。io.output 函数。类似于 io.input。一旦产生错误两个函数都会产生错误。如果你想直接控制错误必须使用完全模式中 io.read 函数。写操作较读操作简单，我们先从写操作入手。下面这个例子里函数 io.write 获取任意数目的字符串参数，接着将它们写到当前的输出文件。通常数字转换为字符串是按照通常的规则，如果要控制这一转换，可以使用 string 库中的 format 函数：

```
> io.write("sin (3) = ", math.sin(3), "\n")
  --> sin (3) = 0.1411200080598672
> io.write(string.format("sin (3) = %.4f\n", math.sin(3)))
  --> sin (3) = 0.1411
```

在编写代码时应当避免像 io.write(a..b..c)；这样的书写，这同 io.write(a,b,c)的效果是一样的。但是后者因为避免了串联操作，而消耗较少的资源。原则上当你进行粗略（quick and dirty）编程，或者进行排错时常使用 print 函数。当需要完全控制输出时使用 write。

```
> print("hello", "Lua"); print("Hi")
  --> hello   Lua
  --> Hi

> io.write("hello", "Lua"); io.write("Hi", "\n")
  --> helloLuaHi
```

Write 函数与 print 函数不同在于，write 不附加任何额外的字符到输出中去，例如制

表符，换行符等等。还有 write 函数是使用当前输出文件，而 print 始终使用标准输出。另外 print 函数会自动调用参数的 tostring 方法，所以可以显示出表（tables）函数（functions）和 nil。

read 函数从当前输入文件读取串，由它的参数控制读取的内容：

"*all"	读取整个文件
"*line"	读取下一行
"*number"	从串中转换出一个数值
num	读取 num 个字符到串

io.read("*all")函数从当前位置读取整个输入文件。如果当前位置在文件末尾，或者文件为空，函数将返回空串。由于 Lua 对长串类型值的有效管理，在 Lua 中使用过滤器的简单方法就是读取整个文件到串中去，处理完之后（例如使用函数 gsub），接着写到输出中去：

```
t = io.read("*all")        -- read the whole file
t = string.gsub(t, ...)     -- do the job
io.write(t)                -- write the file
```

以下代码是一个完整的处理字符串的例子。文件的内容要使用 MIME（多用途的网际邮件扩充协议）中的 quoted-printable 码进行编码。以这种形式编码，非 ASCII 字符将被编码为“=XX”，其中 XX 是该字符值的十六进制表示，为表示一致性“=”字符同样要求被改写。在 gsub 函数中的“模式”参数的作用就是得到所有值在 128 到 255 之间的字符，给它们加上等号标志。

```
t = io.read("*all")
t = string.gsub(t, "([\128-\255=])", function (c)
    return string.format("=%02X", string.byte(c))
end)
io.write(t)
```

该程序在奔腾 333MHz 环境下转换 200k 字符需要 0.2 秒。

io.read("*line")函数返回当前输入文件的下一行（不包含最后的换行符）。当到达文件末尾，返回值为 nil（表示没有下一行可返回）。该读取方式是 read 函数的默认方式，所以可以简写为 io.read()。通常使用这种方式读取文件是由于对文件的操作是自然逐行进行的，否则更倾向于使用*all 一次读取整个文件，或者稍后见到的逐块的读取文件。下面的程序演示了应如何使用该模式读取文件。此程序复制当前输入文件到输出文件，并记录行数。

```
local count = 1
while true do
    local line = io.read()
    if line == nil then break end
```

```
    io.write(string.format("%6d  ", count), line, "\n")
    count = count + 1
end
```

然而为了在整个文件中逐行迭代。我们最好使用 io.lines 迭代器。例如对文件的行进行排序的程序如下：

```
local lines = {}
-- read the lines in table 'lines'
for line in io.lines() do
    table.insert(lines, line)
end
-- sort
table.sort(lines)
-- write all the lines
for i, l in ipairs(lines) do io.write(l, "\n") end
```

在奔腾 333MHz 上该程序处理处理 4.5MB 大小，32K 行的文件耗时 1.8 秒，比使用高度优化的 C 语言系统排序程序快 0.6 秒。io.read("*number")函数从当前输入文件中读取出一个数值。只有在该参数下 read 函数才返回数值，而不是字符串。当需要从一个文件中读取大量数字时，数字间的字符串为空白可以显著的提高执行性能。*number 选项会跳过两个可被识别数字之间的任意空格。这些可识别的字符串可以是-3、+5.2、1000，和 -3.4e-23。如果在当前位置找不到一个数字（由于格式不对，或者是到了文件的结尾），则返回 nil 可以对每个参数设置选项，函数将返回各自的结果。假如有一个文件每行包含三个数字：

```
6.0         -3.23          15e12
4.3         234            1000001
...
```

现在要打印出每行最大的一个数，就可以使用一次 read 函数调用来读取出每行的全部三个数字：

```
while true do
    local n1, n2, n3 = io.read("*number", "*number", "*number")
    if not n1 then break end
    print(math.max(n1, n2, n3))
end
```

在任何情况下，都应该考虑选择使用 io.read 函数的 "*.all" 选项读取整个文件，然后使用 gfind 函数来分解：

```
local pat = "(%S+)%s+(%S+)%s+(%S+)%s+"
for n1, n2, n3 in string.gfind(io.read("*all"), pat) do
```

```
    print(math.max(n1, n2, n3))
end
```

除了基本读取方式外，还可以将数值 n 作为 read 函数的参数。在这样的情况下 read 函数将尝试从输入文件中读取 n 个字符。如果无法读取到任何字符(已经到了文件末尾)，函数返回 nil。否则返回一个最多包含 n 个字符的串。以下是关于该 read 函数参数的一个进行高效文件复制的例子程序（当然是指在 Lua 中）

```
local size = 2^13        -- good buffer size (8K)
while true do
    local block = io.read(size)
    if not block then break end
    io.write(block)
end
```

特别的，io.read(0)函数的可以用来测试是否到达了文件末尾。如果不是返回一个空串，如果已是文件末尾返回 nil。

21.2 完全 I/O 模式

为了对输入输出的更全面的控制，可以使用完全模式。完全模式的核心在于文件句柄（file handle）。该结构类似于 C 语言中的文件流（FILE*），其呈现了一个打开的文件以及当前存取位置。打开一个文件的函数是 io.open。它模仿 C 语言中的 fopen 函数，同样需要打开文件的文件名参数，打开模式的字符串参数。模式字符串可以是 "r"（读模式），"w"（写模式，对数据进行覆盖），或者是 "a"（附加模式）。并且字符 "b" 可附加在后面表示以二进制形式打开文件。正常情况下 open 函数返回一个文件的句柄。如果发生错误，则返回 nil，以及一个错误信息和错误代码。

```
print(io.open("non-existent file", "r"))
    --> nil    No such file or directory      2

print(io.open("/etc/passwd", "w"))
    --> nil    Permission denied              13
```

错误代码的定义由系统决定。

以下是一段典型的检查错误的代码：

```
local f = assert(io.open(filename, mode))
```

如果 open 函数失败，错误信息作为 assert 的参数，由 assert 显示出信息。文件打开后就可以用 read 和 write 方法对他们进行读写操作。它们和 io 表的 read/write 函数类似，但是调用方法上不同，必须使用冒号字符，作为文件句柄的方法来调用。例如打开一个文件并全部读取。可以使用如下代码。

```
local f = assert(io.open(filename, "r"))
local t = f:read("*all")
f:close()
```

同 C 语言中的流（stream）设定类似，I/O 库提供三种预定义的句柄：io.stdin、io.stdout 和 io.stderr。因此可以用如下代码直接发送信息到错误流（error stream）。

```
io.stderr:write(message)
```

我们还可以将完全模式和简单模式混合使用。使用没有任何参数的 io.input()函数得到当前的输入文件句柄；使用带有参数的 io.input(handle)函数设置当前的输入文件为 handle 句柄代表的输入文件。（同样的用法对于 io.output 函数也适用）例如要实现暂时的改变当前输入文件，可以使用如下代码：

```
local temp = io.input()      -- save current file
io.input("newinput")         -- open a new current file
...                          -- do something with new input
io.input():close()           -- close current file
io.input(temp)               -- restore previous current file
```

21.2.1 I/O 优化的一个小技巧

由于通常 Lua 中读取整个文件要比一行一行的读取一个文件快的多。尽管我们有时候针对较大的文件（几十，几百兆），不可能把一次把它们读取出来。要处理这样的文件我们仍然可以一段一段（例如 8kb 一段）的读取它们。同时为了避免切割文件中的行，还要在每段后加上一行：

```
local lines, rest = f:read(BUFSIZE, "*line")
```

以上代码中的 rest 就保存了任何可能被段划分切断的行。然后再将段（chunk）和行接起来。这样每个段就是以一个完整的行结尾的了。以下代码就较为典型的使用了这一技巧。该段程序实现对输入文件的字符，单词，行数的计数。

```
local BUFSIZE = 2^13         -- 8K
local f = io.input(arg[1])   -- open input file
local cc, lc, wc = 0, 0, 0   -- char, line, and word counts

while true do
    local lines, rest = f:read(BUFSIZE, "*line")
    if not lines then break end
    if rest then lines = lines .. rest .. '\n' end
    cc = cc + string.len(lines)
    -- count words in the chunk
```

```
    local _,t = string.gsub(lines, "%S+", "")
    wc = wc + t
    -- count newlines in the chunk
    _,t = string.gsub(lines, "\n", "\n")
    lc = lc + t
end

print(lc, wc, cc)
```

21.2.2 二进制文件

默认的简单模式总是以文本模式打开。在 Unix 中二进制文件和文本文件并没有区别，但是在如 Windows 这样的系统中，二进制文件必须以显式的标记来打开文件。控制这样的二进制文件，你必须将“b”标记添加在 io.open 函数的格式字符串参数中。在 Lua 中二进制文件的控制和文本类似。一个串可以包含任何字节值，库中几乎所有的函数都可以用来处理任意字节值。（你甚至可以对二进制的“串”进行模式比较，只要串中不存在 0 值。如果想要进行 0 值字节的匹配，你可以使用%z 代替）这样使用*all 模式就是读取整个文件的值，使用数字 n 就是读取 n 个字节的值。以下是一个将文本文件从 DOS 模式转换到 Unix 模式的简单程序。（这样转换过程就是将“回车换行字符”替换成“换行字符”。）因为是以二进制形式（原稿是 Text Mode!!？？）打开这些文件的，这里无法使用标准输入输入文件（stdin/stdout）。所以使用程序中提供的参数来得到输入、输出文件名。

```
local inp = assert(io.open(arg[1], "rb"))
local out = assert(io.open(arg[2], "wb"))

local data = inp:read("*all")
data = string.gsub(data, "\r\n", "\n")
out:write(data)

assert(out:close())
```

可以使用如下的命令行来调用该程序。

```
> lua prog.lua file.dos file.unix
```

第二个例子程序：打印在二进制文件中找到的所有特定字符串。该程序定义了一种最少拥有六个“有效字符”，以零字节值结尾的特定串。（本程序中“有效字符”定义为文本数字、标点符号和空格符，由变量 validchars 定义。）在程序中我们使用连接和 string.rep 函数创建 validchars，以%z 结尾来匹配串的零结尾。

```
local f = assert(io.open(arg[1], "rb"))
```

```
local data = f:read("*all")
local validchars = "[%w%p%s]"
local pattern = string.rep(validchars, 6) .. "+%z"
for w in string.gfind(data, pattern) do
    print(w)
end
```

最后一个例子：该程序对二进制文件进行一次值分析 [6]（Dump）。程序的第一个参数是输入文件名，输出为标准输出。其按照 10 字节为一段读取文件，将每一段各字节的十六进制表示显示出来。接着再以文本的形式写出该段，并将控制字符转换为点号。

```
local f = assert(io.open(arg[1], "rb"))
local block = 10
while true do
    local bytes = f:read(block)
    if not bytes then break end
    for b in string.gfind(bytes, ".") do
        io.write(string.format("%02X ", string.byte(b)))
    end
    io.write(string.rep("   ", block - string.len(bytes) + 1))
    io.write(string.gsub(bytes, "%c", "."), "\n")
end
```

如果以 vip 来命名该程序脚本文件。可以使用如下命令来执行该程序处理其自身：

```
prompt> lua vip vip
```

在 Unix 系统中它将会会产生一个如下的输出样式：

```
6C 6F 63 61 6C 20 66 20 3D 20      local f =
61 73 73 65 72 74 28 69 6F 2E      assert(io.
6F 70 65 6E 28 61 72 67 5B 31      open(arg[1
5D 2C 20 22 72 62 22 29 29 0A      ], "rb")).
               ...
22 25 63 22 2C 20 22 2E 22 29      "%c", ".")
2C 20 22 5C 6E 22 29 0A 65 6E      , "\n").en
64 0A                              d.
```

21.3 关于文件的其它操作

函数 tmpfile 函数用来返回零时文件的句柄，并且其打开模式为 read/write 模式。该

[6] 译注：得到类似于十六进制编辑器的一个界面显示

零时文件在程序执行完后会自动进行清除。函数 flush 用来应用针对文件的所有修改。同 write 函数一样，该函数的调用既可以按函数调用的方法使用 io.flush()来应用当前输出文件；也可以按文件句柄方法的样式 f:flush()来应用文件 f。函数 seek 用来得到和设置一个文件的当前存取位置。它的一般形式为 filehandle:seek(whence,offset)。Whence 参数是一个表示偏移方式的字符串。它可以是 "set"，偏移值是从文件头开始；"cur"，偏移值从当前位置开始；"end"，偏移值从文件尾往前计数。offset 即为偏移的数值，由 whence 的值和 offset 相结合得到新的文件读取位置。该位置是实际从文件开头计数的字节数。whence 的默认值为 "cur"，offset 的默认值为 0。这样调用 file:seek()得到的返回值就是文件当前的存取位置，且保持不变。file:seek("set")就是将文件的存取位置重设到文件开头。（返回值当然就是 0）。而 file:seek("end")就是将位置设为文件尾，同时就可以得到文件的大小。如下的代码实现了得到文件的大小而不改变存取位置。

```
function fsize (file)
    local current = file:seek()     -- get current position
    local size = file:seek("end")   -- get file size
    file:seek("set", current)       -- restore position
    return size
end
```

以上的几个函数在出错时都将返回一个包含了错误信息的 nil 值。

第 22 章 操作系统库

操作系统库包含了文件管理，系统时钟等等与操作系统相关信息。这些函数定义在表（table）os 中。定义该库时考虑到 Lua 的可移植性，因为 Lua 是以 ANSI C 写成的，所以只能使用 ANSI 定义的一些标准函数。许多的系统属性并不包含在 ANSI 定义中，例如目录管理，套接字等等。所以在系统库里并没有提供这些功能。另外有一些没有包含在主体发行版中的 Lua 库提供了操作系统扩展属性的访问。例如 posix 库，提供了对 POSIX 1 标准的完全支持；在比如 luasocket 库，提供了网络支持。

在文件管理方面操作系统库就提供了 os.rename 函数（修改文件名）和 os.remove 函数（删除文件）。

22.1 Date 和 Time

time 和 date 两个函数在 Lua 中实现所有的时钟查询功能。函数 time 在没有参数时返回当前时钟的数值。（在许多系统中该数值是当前距离某个特定时间的秒数。）当为函数调用附加一个特殊的时间表时，该函数就是返回距该表描述的时间的数值。这样的时间表有如下的区间：

year	a full year
month	01-12
day	01-31
hour	01-31
min	00-59
sec	00-59
isdst	a boolean, **true** if daylight saving

前三项是必需的，如果未定义后几项，默认时间为正午（12:00:00）。如果是在里约热内卢（格林威治向西三个时区）的一台 Unix 计算机上（相对时间为 1970 年 1 月 1 日，00:00:00）执行如下代码，其结果将如下。

```
-- obs: 10800 = 3*60*60 (3 hours)
print(os.time{year=1970, month=1, day=1, hour=0})
    --> 10800
print(os.time{year=1970, month=1, day=1, hour=0, sec=1})
    --> 10801
print(os.time{year=1970, month=1, day=1})
    --> 54000   (obs: 54000 = 10800 + 12*60*60)
```

函数 data，不管它的名字是什么，其实是 time 函数的一种“反函数”。它将一个表示日期和时间的数值，转换成更高级的表现形式。其第一个参数是一个格式化字符串，描述了要返回的时间形式。第二个参数就是时间的数字表示，默认为当前的时间。使用格式字符 "*t"，创建一个时间表。例如下面这段代码：

```
temp = os.date("*t", 906000490)
```

则会产生表

```
{year = 1998, month = 9, day = 16, yday = 259, wday = 4,
 hour = 23, min = 48, sec = 10, isdst = false}
```

不难发现该表中除了使用到了在上述时间表中的区域以外，这个表还提供了星期（wday，星期天为 1）和一年中的第几天（yday，一月一日为 1）除了使用 "*t" 格式字符串外，如果使用带标记（见下表）的特殊字符串，os.data 函数会将相应的标记位以时间信息进行填充，得到一个包含时间的字符串。（这些特殊标记都是以 "%" 和一个字母的形式出现）如下：

```
print(os.date("today is %A, in %B"))
    --> today is Tuesday, in May
print(os.date("%x", 906000490))
    --> 09/16/1998
```

这些时间输出的字符串表示是经过本地化的。所以如果是在巴西（葡萄牙语系），"%B" 得到的就是 "setembro"（译者按：大概是葡萄牙语九月？），"%x" 得到的就是 "16/09/98"（月日次序不同）。标记的意义和显示实例总结如下表。实例的时间是在 1998 年九月 16 日，星期三，23:48:10。返回值为数字形式的还列出了它们的范围。（都是按照英语系的显示描述的，也比较简单，就不烦了）

%a	abbreviated weekday name (e.g., Wed)
%A	full weekday name (e.g., Wednesday)
%b	abbreviated month name (e.g., Sep)
%B	full month name (e.g., September)
%c	date and time (e.g., 09/16/98 23:48:10)
%d	day of the month (16) [01-31]
%H	hour, using a 24-hour clock (23) [00-23]
%I	hour, using a 12-hour clock (11) [01-12]
%M	minute (48) [00-59]
%m	month (09) [01-12]
%p	either "am" or "pm" (pm)
%S	second (10) [00-61]
%w	weekday (3) [0-6 = Sunday-Saturday]
%x	date (e.g., 09/16/98)
%X	time (e.g., 23:48:10)

%Y	full year (1998)
%y	two-digit year (98) [00-99]
%%	the character '%'

事实上如果不使用任何参数就调用 date，就是以%c 的形式输出。这样就是得到经过格式化的完整时间信息。还要注意%x、%X 和%c 由所在地区和计算机系统的改变会发生变化。如果该字符串要确定下来（例如确定为 mm/dd/yyyy），可以使用明确的字符串格式方式（例如"%m/%d/%Y"）。

函数 os.clock 返回执行该程序 CPU 花去的时钟秒数。该函数常用来测试一段代码。

```
local x = os.clock()
local s = 0
for i=1,100000 do s = s + i end
print(string.format("elapsed time: %.2f\n", os.clock() - x))
```

22.2 其它的系统调用

函数 os.exit 终止一个程序的执行。函数 os.getenv 得到“环境变量”的值。以“变量名”作为参数，返回该变量值的字符串：

```
print(os.getenv("HOME"))     --> /home/lua
```

如果没有该环境变量则返回 nil。函数 os.execute 执行一个系统命令（和 C 中的 system 函数等价）。该函数获取一个命令字符串，返回一个错误代码。例如在 Unix 和 DOS-Windows 系统里都可以执行如下代码创建一个新目录：

```
function createDir (dirname)
    os.execute("mkdir " .. dirname)
end
```

os.execute 函数较为强大，同时也更加倚赖于计算机系统。函数 os.setlocale 设定 Lua 程序所使用的区域（locale）。区域定义的变化对于文化和语言是相当敏感的。setlocale 有两个字符串参数：区域名和特性（category，用来表示区域的各项特性）。在区域中包含六项特性：“collate”（排序）控制字符的排列顺序；"ctype" controls the types of individual characters (e.g., what is a letter) and the conversion between lower and upper cases; "monetary"（货币）对 Lua 程序没有影响；"numeric"（数字）控制数字的格式；"time" 控制时间的格式（也就是 os.date 函数）；和“all”包含以上所以特性。函数默认的特性就是“all”，所以如果你只包含地域名就调用函数 setlocale 那么所有的特性都会被改变为新的区域特性。如果运行成功函数返回地域名，否则返回 nil（通常因为系统不支持给定的区域）。

```
print(os.setlocale("ISO-8859-1", "collate"))  --> ISO-8859-1
```

关于“numeric”特性有一点难处理的地方。尽管葡萄牙语和其它的一些拉丁文语言

使用逗号代替点号来表示十进制数，但是区域设置并不会改变Lua划分数字的方式。（除了其它一些原因之外，由于print（3,4）还有其它的函数意义。）因此设置之后得到的系统也许既不能识别带逗号的数值，又不能理解带点号的数值[7]：

```
-- 设置区域为葡萄牙语系巴西
print(os.setlocale('pt_BR'))    --> pt_BR
print(3,4)          --> 3    4
print(3.4)          --> stdin:1: malformed number near `3.4'
```

The category "numeric" is a little tricky. Although Portuguese and other Latin languages use a comma instead of a point to represent decimal numbers, the locale does not change the way that Lua parses numbers (among other reasons because expressions like print(3,4) already have a meaning in Lua). Therefore, you may end with a system that cannot recognize numbers with commas, but cannot understand numbers with points either:

```
-- set locale for Portuguese-Brazil
print(os.setlocale('pt_BR'))    --> pt_BR
print(3,4)          --> 3    4
print(3.4)          --> stdin:1: malformed number near '3.4'
```

[7] 译者按：好像是巴西人的烦恼，不甚解。附原文。

第 23 章 Debug 库

debug 库并不给你一个可用的 Lua 调试器，而是给你提供一些为 Lua 写一个调试器的方便。出于性能方面的考虑，关于这方面官方的接口是通过 C API 实现的。Lua 中的 debug 库就是一种在 Lua 代码中直接访问这些 C 函数的方法。Debug 库在一个 debug 表内声明了他所有的函数。

与其他的标准库不同的是，你应该尽可能少的是有 debug 库。首先，debug 库中的一些函数性能比较低；第二，它破坏了语言的一些真理(sacred truths)，比如你不能在定义一个局部变量的函数外部，访问这个变量。通常，在你的最终产品中，你不想打开这个 debug 库，或者你可能想删除这个库：

```
debug = nil
```

debug 库由两种函数组成：自省(introspective)函数和 hooks。自省函数使得我们可以检查运行程序的某些方面，比如活动函数栈、当前执行代码的行号、本地变量的名和值。Hooks 可以跟踪程序的执行情况。

Debug 库中的一个重要的思想是栈级别(stack level)。一个栈级别就是一个指向在当前时刻正在活动的特殊函数的数字，也就是说，这个函数正在被调用但还没有返回。调用 debug 库的函数级别为 1，调用他(他指调用 debug 库的函数)的函数级别为 2，以此类推。

23.1 自省（Introspective）

在 debug 库中主要的自省函数是 debug.getinfo。他的第一个参数可以是一个函数或者栈级别。对于函数 foo 调用 debug.getinfo(foo)，将返回关于这个函数信息的一个表。这个表有下列一些域：

- ✓ source，标明函数被定义的地方。如果函数在一个字符串内被定义（通过 loadstring），source 就是那个字符串。如果函数在一个文件中定义，source 是@加上文件名。
- ✓ short_src，source 的简短版本（最多 60 个字符），记录一些有用的错误信息。
- ✓ linedefined，source 中函数被定义之处的行号。
- ✓ what，标明函数类型。如果 foo 是一个普通得 Lua 函数，结果为 "Lua"；如果是一个 C 函数，结果为 "C"；如果是一个 Lua 的主 chunk，结果为 "main"。
- ✓ name，函数的合理名称。
- ✓ namewhat，上一个字段代表的含义。这个字段的取值可能为：W"global"、"local"、

"method"、"field"，或者 ""（空字符串）。空字符串意味着 Lua 没有找到这个函数名。

- ✓ nups，函数的 upvalues 的个数。
- ✓ func，函数本身；详细情况看后面。

当 foo 是一个 C 函数的时候，Lua 无法知道很多相关的信息，所以对这种函数，只有 what、name、namewhat 这几个域的值可用。

以数字 n 调用 debug.getinfo(n)时，返回在 n 级栈的活动函数的信息数据。比如，如果 n=1，返回的是正在进行调用的那个函数的信息。（n=0 表示 C 函数 getinfo 本身）如果 n 比栈中活动函数的个数大的话，debug.getinfo 返回 nil。当你使用数字 n 调用 debug.getinfo 查询活动函数的信息的时候，返回的结果 table 中有一个额外的域：currentline，即在那个时刻函数所在的行号。另外，func 表示指定 n 级的活动函数。

字段名的写法有些技巧。记住：因为在 Lua 中函数是第一类值，所以一个函数可能有多个函数名。查找指定值的函数的时候，Lua 会首先在全局变量中查找，如果没找到才会到调用这个函数的代码中看它是如何被调用的。后面这种情况只有在我们使用数字调用 getinfo 的时候才会起作用，也就是这个时候我们能够获取调用相关的详细信息。

函数 getinfo 的效率并不高。Lua 以不消弱程序执行的方式保存 debug 信息（Lua keeps debug information in a form that does not impair program execution），效率被放在第二位。为了获取比较好地执行性能，getinfo 可选的第二个参数可以用来指定选取哪些信息。指定了这个参数之后，程序不会浪费时间去收集那些用户不关心的信息。这个参数的格式是一个字符串，每一个字母代表一种类型的信息，可用的字母的含义如下：

'n'	selects fields name and namewhat
'f'	selects field func
'S'	selects fields source, short_src, what, and linedefined
'l'	selects field currentline
'u'	selects field nup

下面的函数阐明了 debug.getinfo 的使用，函数打印一个活动栈的原始跟踪信息（traceback）：

```
function traceback ()
  local level = 1
  while true do
    local info = debug.getinfo(level, "Sl")
    if not info then break end
    if info.what == "C" then    -- is a C function?
      print(level, "C function")
    else   -- a Lua function
      print(string.format("[%s]:%d",
            info.short_src, info.currentline))
```

```
        end
        level = level + 1
    end
end
```

不难改进这个函数，使得 getinfo 获取更多的数据，实际上 debug 库提供了一个改善的版本 debug.traceback，与我们上面的函数不同的是，debug.traceback 并不打印结果，而是返回一个字符串。

23.1.1 访问局部变量

调用 debug 库的 getlocal 函数可以访问任何活动状态的局部变量。这个函数由两个参数：将要查询的函数的栈级别和变量的索引。函数有两个返回值：变量名和变量当前值。如果指定的变量的索引大于活动变量个数，getlocal 返回 nil。如果指定的栈级别无效，函数会抛出错误。（你可以使用 debug.getinfo 检查栈级别的有效性）

Lua 对函数中所出现的所有局部变量依次计数，只有在当前函数的范围内是有效的局部变量才会被计数。比如，下面的代码

```
function foo (a,b)
    local x
    do local c = a - b end
    local a = 1
    while true do
        local name, value = debug.getlocal(1, a)
        if not name then break end
        print(name, value)
        a = a + 1
    end
end

foo(10, 20)
```

结果为：

```
a       10
b       20
x       nil
a       4
```

索引为 1 的变量是 a，2 是 b，3 是 x，4 是另一个 a。在 getlocal 被调用的那一点，c 已经超出了范围，name 和 value 都不在范围内。（记住：局部变量仅仅在他们被初始化之后才可见）也可以使用 debug.setlocal 修改一个局部变量的值，他的前两个参数是栈级

别和变量索引，第三个参数是变量的新值。这个函数返回一个变量名或者 nil（如果变量索引超出范围）

23.1.2 访问 Upvalues

我们也可以通过 debug 库的 getupvalue 函数访问 Lua 函数的 upvalues。和局部变量不同的是，即使函数不在活动状态他依然有 upvalues（这也就是闭包的意义所在）。所以，getupvalue 的第一个参数不是栈级别而是一个函数（精确的说应该是一个闭包），第二个参数是 upvalue 的索引。Lua 按照 upvalue 在一个函数中被引用(refer)的顺序依次编号，因为一个函数不能有两个相同名字的 upvalues，所以这个顺序和 upvalue 并没什么关联（relevant）。

可以使用函数 ebug.setupvalue 修改 upvalues。也许你已经猜到，他有三个参数：一个闭包，一个 upvalues 索引和一个新的 upvalue 值。和 setlocal 类似，这个函数返回 upvalue 的名字，或者 nil（如果 upvalue 索引超出索引范围）。

下面的代码显示了，在给定变量名的情况下，如何访问一个正在调用的函数的任意的给定变量的值：

```
function getvarvalue (name)
    local value, found

    -- try local variables
    local i = 1
    while true do
        local n, v = debug.getlocal(2, i)
        if not n then break end
        if n == name then
            value = v
            found = true
        end
        i = i + 1
    end
    if found then return value end

    -- try upvalues
    local func = debug.getinfo(2).func
    i = 1
    while true do
        local n, v = debug.getupvalue(func, i)
        if not n then break end
```

```
        if n == name then return v end
        i = i + 1
    end

    -- not found; get global
    return getfenv(func)[name]
end
```

首先，我们尝试这个变量是否为局部变量：如果对于给定名字的变量有多个变量，我们必须访问具有最高索引的那一个，所以我们总是需要遍历整个循环。如果在局部变量中找不到指定名字的变量，我们尝试这个变量是否为 upvalues：首先，我们使用 debug.getinfo(2).func 获取调用的函数，然后遍历这个函数的 upvalues，最后如果我们找到给定名字的变量，我们在全局变量中查找。注意调用 debug.getlocal 和 debug.getinfo 的参数 2（用来访问正在调用的函数）的用法。

23.2 Hooks

debug 库的 hook 是这样一种机制：注册一个函数，用来在程序运行中某一事件到达时被调用。有四种可以触发一个 hook 的事件：当 Lua 调用一个函数的时候 call 事件发生；每次函数返回的时候，return 事件发生；Lua 开始执行代码的新行时候，line 事件发生；运行指定数目的指令之后，count 事件发生。Lua 使用单个参数调用 hooks，参数为一个描述产生调用的事件："call"、"return"、"line" 或 "count"。另外，对于 line 事件，还可以传递第二个参数：新行号。我们在一个 hook 内总是可以使用 debug.getinfo 获取更多的信息。

使用带有两个或者三个参数的 debug.sethook 函数来注册一个 hook：第一个参数是 hook 函数；第二个参数是一个描述我们打算监控的事件的字符串；可选的第三个参数是一个数字，描述我们打算获取 count 事件的频率。为了监控 call、return 和 line 事件，可以将他们的第一个字母（'c'、'r' 或 'l'）组合成一个 mask 字符串即可。要想关掉 hooks，只需要不带参数地调用 sethook 即可。

下面的简单代码，是一个安装原始的跟踪器：打印解释器执行的每一个新行的行号：

```
debug.sethook(print, "l")
```

上面这一行代码，简单的将 print 函数作为 hook 函数，并指示 Lua 当 line 事件发生时调用 print 函数。可以使用 getinfo 将当前正在执行的文件名信息加上去，使得跟踪器稍微精致点的：

```
function trace (event, line)
    local s = debug.getinfo(2).short_src
    print(s .. ":" .. line)
end
```

```
debug.sethook(trace, "l")
```

23.3 Profiles

尽管 debug 库名字上看来是一个调式库，除了用于调式以外，还可以用于完成其他任务。这种常见的任务就是 profiling。对于一个实时的 profile 来说（For a profile with timing），最好使用 C 接口来完成：对于每一个 hook 过多的 Lua 调用代价太大并且通常会导致测量的结果不准确。然而，对于计数的 profiles 而言，Lua 代码可以很好的胜任。下面这部分我们将实现一个简单的 profiler：列出在程序运行过程中，每一个函数被调用的次数。

我们程序的主要数据结构是两张表，一张关联函数和他们调用次数的表，一张关联函数和函数名的表。这两个表的索引下标是函数本身。

```
local Counters = {}
local Names = {}
```

在 profiling 之后，我们可以访问函数名数据，但是记住：在函数在活动状态的情况下，可以得到比较好的结果，因为那时候 Lua 会察看正在运行的函数的代码来查找指定的函数名。

现在我们定义 hook 函数，他的任务就是获取正在执行的函数并将对应的计数器加 1；同时这个 hook 函数也收集函数名信息：

```
local function hook ()
    local f = debug.getinfo(2, "f").func
    if Counters[f] == nil then  -- first time `f' is called?
        Counters[f] = 1
        Names[f] = debug.getinfo(2, "Sn")
    else   -- only increment the counter
        Counters[f] = Counters[f] + 1
    end
end
```

下一步就是使用这个 hook 运行程序，我们假设程序的主 chunk 在一个文件内，并且用户将这个文件名作为 profiler 的参数：

```
prompt> lua profiler main-prog
```

这种情况下，我们的文件名保存在 arg[1]，打开 hook 并运行文件：

```
local f = assert(loadfile(arg[1]))
debug.sethook(hook, "c")    -- turn on the hook
```

```
f()     -- run the main program
debug.sethook()   -- turn off the hook
```

最后一步是显示结果，下一个函数为一个函数产生名称，因为在 Lua 中的函数名不确定，所以我们对每一个函数加上他的位置信息，型如 file:line 。如果一个函数没有名字，那么我们只用它的位置表示。如果一个函数是 C 函数，我们只是用它的名字表示（他没有位置信息）。

```
function getname (func)
    local n = Names[func]
    if n.what == "C" then
        return n.name
    end
    local loc = string.format("[%s]:%s",
           n.short_src, n.linedefined)
    if n.namewhat ~= "" then
        return string.format("%s (%s)", loc, n.name)
    else
        return string.format("%s", loc)
    end
end
```

最后，我们打印每一个函数和他的计数器：

```
for func, count in pairs(Counters) do
    print(getname(func), count)
end
```

如果我们将我们的 profiler 应用到 Section 10.2 的马尔科夫链的例子上，我们得到如下结果：

```
[markov.lua]:4 884723
write  10000
[markov.lua]:0 (f)       1
read   31103
sub    884722
[markov.lua]:1 (allwords)       1
[markov.lua]:20 (prefix)        894723
find   915824
[markov.lua]:26 (insert)        884723
random  10000
sethook 1
insert  884723
```

那意味着第四行的匿名函数（在 allwords 内定义的迭代函数）被调用 884,723 次，write(io.write)被调用 10,000 次。

你可以对这个 profiler 进行一些改进，比如对输出排序、打印出比较好的函数名、改善输出格式。不过，这个基本的 profiler 已经很有用，并且可以作为很多高级工具的基础。

第四篇 C API

第 24 章 C API 纵览

Lua 是一个嵌入式的语言，意味着 Lua 不仅可以是一个独立运行的程序包也可以是一个用来嵌入其他应用的程序库。你可能觉得奇怪：如果 Lua 不只是独立的程序，为什么到目前为止贯穿整本书我们都是在使用 Lua 独立程序呢？这个问题的答案在于 Lua 解释器（可执行的 lua）。Lua 解释器是一个使用 Lua 标准库实现的独立的解释器，她是一个很小的应用（总共不超过 500 行的代码）。解释器负责程序和使用者的接口：从使用者那里获取文件或者字符串，并传给 Lua 标准库，Lua 标准库负责最终的代码运行。

Lua 可以作为程序库用来扩展应用的功能，也就是 Lua 可以作为扩展性语言的原因所在。同时，Lua 程序中可以注册有其他语言实现的函数，这些函数可能由 C 语言(或其他语言)实现，可以增加一些不容易由 Lua 实现的功能。这使得 Lua 是可扩展的。与上面两种观点(Lua 作为扩展性语言和可扩展的语言)对应的 C 和 Lua 中间有两种交互方式。第一种，C 作为应用程序语言，Lua 作为一个库使用；第二种，反过来，Lua 作为程序语言，C 作为库使用。这两种方式，C 语言都使用相同的 API 与 Lua 通信，因此 C 和 Lua 交互这部分称为 C API。

C API 是一个 C 代码与 Lua 进行交互的函数集。他有以下部分组成：读写 Lua 全局变量的函数，调用 Lua 函数的函数，运行 Lua 代码片断的函数，注册 C 函数然后可以在 Lua 中被调用的函数，等等。（本书中，术语函数实际上指函数或者宏，API 有些函数为了方便以宏的方式实现）

C API 遵循 C 语言的语法形式，这 Lua 有所不同。当使用 C 进行程序设计的时候，我们必须注意，类型检查，错误处理，内存分配都很多问题。API 中的大部分函数并不检查他们参数的正确性；你需要在调用函数之前负责确保参数是有效的。如果你传递了错误的参数，可能得到 \"segmentation fault\" 这样或者类似的错误信息，而没有很明确的错误信息可以获得。另外，API 重点放在了灵活性和简洁性方面，有时候以牺牲方便实用为代价的。一般的任务可能需要涉及很多个 API 调用，这可能令人烦恼，但是他给你提供了对细节的全部控制的能力，比如错误处理，缓冲大小，和类似的问题。如本章的标题所示，这一章的目标是对当你从 C 调用 Lua 时将涉及到哪些内容的预览。如果不能理解某些细节不要着急，后面我们会一一详细介绍。不过，在 Lua 参考手册中有对指定函数的详细描述。另外，在 Lua 发布版中你可以看到 API 的应用的例子，Lua 独立的解释器（lua.c）提供了应用代码的例子，而标准库（lmathlib.c、lstrlib.c 等等）提供了程序库代码的例子。

从现在开始，你戴上了 C 程序员的帽子，当我们谈到“你/你们”，我们意思是指当你使用 C 编程的时候。在 C 和 Lua 之间通信关键内容在于一个虚拟的栈。几乎所有的 API 调用都是对栈上的值进行操作，所有 C 与 Lua 之间的数据交换也都通过这个栈来完成。另外，你也可以使用栈来保存临时变量。栈的使用解决了 C 和 Lua 之间两个不协调

的问题：第一，Lua 会自动进行垃圾收集，而 C 要求显示的分配存储单元，两者引起的矛盾。第二，Lua 中的动态类型和 C 中的静态类型不一致引起的混乱。我们将在 24.2 节详细地介绍栈的相关内容。

24.1 第一个示例程序

通过一个简单的应用程序让我们开始这个预览：一个独立的 Lua 解释器的实现。我们写一个简单的解释器，代码如下：

```
#include <stdio.h>
#include <lua.h>
#include <lauxlib.h>
#include <lualib.h>

int main (void)
{
    char buff[256];
    int error;
    lua_State *L = lua_open();  /* opens Lua */
    luaopen_base(L);         /* opens the basic library */
    luaopen_table(L);        /* opens the table library */
    luaopen_io(L);           /* opens the I/O library */
    luaopen_string(L);       /* opens the string lib. */
    luaopen_math(L);         /* opens the math lib. */

    while (fgets(buff, sizeof(buff), stdin) != NULL) {
        error = luaL_loadbuffer(L, buff, strlen(buff),
                    "line") || lua_pcall(L, 0, 0, 0);
        if (error) {
            fprintf(stderr, "%s", lua_tostring(L, -1));
            lua_pop(L, 1);/* pop error message from the stack */
        }
    }

    lua_close(L);
    return 0;
}
```

头文件 lua.h 定义了 Lua 提供的基础函数。其中包括创建一个新的 Lua 环境的函数（如 lua_open），调用 Lua 函数（如 lua_pcall）的函数，读取/写入 Lua 环境的全局变量

的函数，注册可以被 Lua 代码调用的新函数的函数，等等。所有在 lua.h 中被定义的都有一个 lua_前缀。

头文件 lauxlib.h 定义了辅助库（auxlib）提供的函数。同样，所有在其中定义的函数等都以 luaL_打头（例如，luaL_loadbuffer）。辅助库利用 lua.h 中提供的基础函数提供了更高层次上的抽象；所有 Lua 标准库都使用了 auxlib。基础 API 致力于 economy and orthogonality，相反 auxlib 致力于实现一般任务的实用性。当然，基于你的程序的需要而创建其它的抽象也是非常容易的。需要铭记在心的是，auxlib 没有存取 Lua 内部的权限。它完成它所有的工作都是通过正式的基本 API。

Lua 库没有定义任何全局变量。它所有的状态保存在动态结构 lua_State 中，而且指向这个结构的指针作为所有 Lua 函数的一个参数。这样的实现方式使得 Lua 能够重入（reentrant）且为在多线程中的使用作好准备。

函数 lua_open 创建一个新环境（或 state）。lua_open 创建一个新的环境时，这个环境并不包括预定义的函数，甚至是 print。为了保持 Lua 的苗条，所有的标准库以单独的包提供，所以如果你不需要就不会强求你使用它们。头文件 lualib.h 定义了打开这些库的函数。例如，调用 luaopen_io，以创建 io table 并注册 I/O 函数（io.read,io.write 等等）到 Lua 环境中。

创建一个 state 并将标准库载入之后，就可以着手解释用户的输入了。对于用户输入的每一行，C 程序首先调用 luaL_loadbuffer 编译这些 Lua 代码。如果没有错误，这个调用返回零并把编译之后的 chunk 压入栈。（记住，我们将在下一节中讨论魔法般的栈）之后，C 程序调用 lua_pcall，它将会把 chunk 从栈中弹出并在保护模式下运行它。和 luaL_laodbuffer 一样，lua_pcall 在没有错误的情况下返回零。在有错误的情况下，这两个函数都将一条错误消息压入栈；我们可以用 lua_tostring 来得到这条信息、输出它，用 lua_pop 将它从栈中删除。

注意，在有错误发生的情况下，这个程序简单的输出错误信息到标准错误流。在 C 中，实际的错误处理可能是非常复杂的而且如何处理依赖于应用程序本身。Lua 核心决不会直接输出任何东西到任务输出流上；它通过返回错误代码和错误信息来发出错误信号。每一个应用程序都可以用最适合它们自己的方式来处理这些错误。为了讨论的简单，现在我们假想一个简单的错误处理方式，就象下面代码一样，它只是输出一条错误信息、关闭 Lua state、退出整个应用程序。

```
#include <stdarg.h>
#include <stdio.h>
#include <stdlib.h>

void error (lua_State *L, const char *fmt, ...) {
    va_list argp;
    va_start(argp, fmt);
    vfprintf(stderr, argp);
    va_end(argp);
```

```
    lua_close(L);
    exit(EXIT_FAILURE);
}
```

稍候我们再详细的讨论关于在应用代码中如何处理错误.因为你可以将 Lua 和 C/C++代码一起编译，lua.h 并不包含这些典型的在其他 C 库中出现的整合代码：

```
#ifdef __cplusplus
extern "C" {
#endif
    ...
#ifdef __cplusplus
}
#endif
```

因此，如果你用 C 方式来编译它，但用在 C++中，那么你需要象下面这样来包含 lua.h 头文件。

```
extern "C" {
#include <lua.h>
}
```

一个常用的技巧是建立一个包含上面代码的 lua.hpp 头文件，并将这个新的头文件包含进你的 C++程序。

24.2 堆栈

当在 Lua 和 C 之间交换数据时我们面临着两个问题：动态与静态类型系统的不匹配和自动与手动内存管理的不一致。

在 Lua 中，我们写下 a[k]=v 时，k 和 v 可以有几种不同的类型（由于 metatables 的存在，a 也可能有不同的类型）。如果我们想在 C 中提供类似的操作，无论怎样，操作表的函数(settable)必定有一个固定的类型。我们将需要几十个不同的函数来完成这一个的操作（三个参数的类型的每一种组合都需要一个函数）。

我们可以在 C 中声明一些 union 类型来解决这个问题，我们称之为 lua_Value，它能够描述所有类型的 Lua 值。然后，我们就可以这样声明 settable

```
void lua_settable (lua_Value a, lua_Value k, lua_Value v);
```

这个解决方案有两个缺点。第一，要将如此复杂的类型映射到其它语言可能很困难；Lua 不仅被设计为与 C/C++易于交互，Java,Fortran 以及类似的语言也一样。第二，Lua 负责垃圾回收：如果我们将 Lua 值保存在 C 变量中，Lua 引擎没有办法了解这种用法；它可能错误地认为某个值为垃圾并收集他。

因此，Lua API 没有定义任何类似 lua_Value 的类型。替代的方案，它用一个抽象的栈在 Lua 与 C 之间交换值。栈中的每一条记录都可以保存任何 Lua 值。无论你何时想要从 Lua 请求一个值（比如一个全局变量的值），调用 Lua，被请求的值将会被压入栈。无论你何时想要传递一个值给 Lua，首先将这个值压入栈，然后调用 Lua（这个值将被弹出）。我们仍然需要一个不同的函数将每种 C 类型压入栈和一个不同函数从栈上取值（译注：只是取出不是弹出），但是我们避免了组合式的爆炸（combinatorial explosion）。另外，因为栈是由 Lua 来管理的，垃圾回收器知道那个值正在被 C 使用。 几乎所有的 API 函数都用到了栈。正如我们在第一个例子中所看到的，luaL_loadbuffer 把它的结果留在了栈上（被编译的 chunk 或一条错误信息）；lua_pcall 从栈上获取要被调用的函数并把任何临时的错误信息放在这里。

Lua 以一个严格的 LIFO 规则（后进先出；也就是说，始终存取栈顶）来操作栈。当你调用 Lua 时，它只会改变栈顶部分。你的 C 代码却有更多的自由；更明确的来讲，你可以查询栈上的任何元素，甚至是在任何一个位置插入和删除元素。

24.2.1 压入元素

API 有一系列压栈的函数，它将每种可以用 C 来描述的 Lua 类型压栈：空值（nil）用 lua_pushnil，数值型（double）用 lua_pushnumber，布尔型（在 C 中用整数表示）用 lua_pushboolean，任意的字符串（char*类型，允许包含'\0'字符）用 lua_pushlstring，C 语言风格（以'\0'结束）的字符串（const char*）用 lua_pushstring：

```
void lua_pushnil (lua_State *L);
void lua_pushboolean (lua_State *L, int bool);
void lua_pushnumber (lua_State *L, double n);
void lua_pushlstring (lua_State *L, const char *s,
                                    size_t length);
void lua_pushstring (lua_State *L, const char *s);
```

同样也有将 C 函数和 userdata 值压入栈的函数，稍后会讨论到它们。

Lua 中的字符串不是以零为结束符的；它们依赖于一个明确的长度，因此可以包含任意的二进制数据。将字符串压入串的正式函数是 lua_pushlstring，它要求一个明确的长度作为参数。对于以零结束的字符串，你可以用 lua_pushstring（它用 strlen 来计算字符串长度）。Lua 从来不保持一个指向外部字符串（或任何其它对象，除了 C 函数——它总是静态指针）的指针。对于它保持的所有字符串，Lua 要么做一份内部的拷贝要么重新利用已经存在的字符串。因此，一旦这些函数返回之后你可以自由的修改或是释放你的缓冲区。

无论你何时压入一个元素到栈上，你有责任确保在栈上有空间来做这件事情。记住，你现在是 C 程序员；Lua 不会宠着你。当 Lua 在起始以及在 Lua 调用 C 的时候，栈上至少有 20 个空闲的记录（lua.h 中的 LUA_MINSTACK 宏定义了这个常量）。对于多数普通的用法栈是足够的，所以通常我们不必去考虑它。无论如何，有些任务或许需要更多的

栈空间（如，调用一个不定参数数目的函数）。在这种情况下，或许你需要调用下面这个函数：

```
int lua_checkstack (lua_State *L, int sz);
```

它检测栈上是否有足够你需要的空间（稍后会有关于它更多的信息）。

24.2.2 查询元素

API 用索引来访问栈中的元素。在栈中的第一个元素（也就是第一个被压入栈的）有索引 1，下一个有索引 2，以此类推。我们也可以用栈顶作为参照来存取元素，利用负索引。在这种情况下，-1 指出栈顶元素（也就是最后被压入的），-2 指出它的前一个元素，以此类推。例如，调用 lua_tostring(L, -1)以字符串的形式返回栈顶的值。我们下面将看到，在某些场合使用正索引访问栈比较方便，另外一些情况下，使用负索引访问栈更方便。

API 提供了一套 lua_is*函数来检查一个元素是否是一个指定的类型，*可以是任何 Lua 类型。因此有 lua_isnumber,lua_isstring,lua_istable 以及类似的函数。所有这些函数都有同样的原型：

```
int lua_is... (lua_State *L, int index);
```

lua_isnumber 和 lua_isstring 函数不检查这个值是否是指定的类型，而是看它是否能被转换成指定的那种类型。例如，任何数字类型都满足 lua_isstring。

还有一个 lua_type 函数，它返回栈中元素的类型。（lua_is*中的有些函数实际上是用了这个函数定义的宏）在 lua.h 头文件中，每种类型都被定义为一个常量：LUA_TNIL、LUA_TBOOLEAN 、 LUA_TNUMBER 、 LUA_TSTRING 、 LUA_TTABLE 、LUA_TFUNCTION、LUA_TUSERDATA 以及 LUA_TTHREAD。这个函数主要被用在与一个 switch 语句联合使用。当我们需要真正的检查字符串和数字类型时它也是有用的

为了从栈中获得值，这里有 lua_to*函数：

```
int             lua_toboolean (lua_State *L, int index);
double          lua_tonumber (lua_State *L, int index);
const char *    lua_tostring (lua_State *L, int index);
size_t          lua_strlen (lua_State *L, int index);
```

即使给定的元素的类型不正确，调用上面这些函数也没有什么问题。在这种情况下，lua_toboolean、lua_tonumber 和 lua_strlen 返回 0，其他函数返回 NULL。由于 ANSI C 没有提供有效的可以用来判断错误发生数字值，所以返回的 0 是没有什么用处的。对于其他函数而言，我们一般不需要使用对应的 lua_is*函数：我们只需要调用 lua_is*，测试返回结果是否为 NULL 即可。

Lua_tostring 函数返回一个指向字符串的内部拷贝的指针。你不能修改它（使你想起那里有一个 const）。只要这个指针对应的值还在栈内，Lua 会保证这个指针一直有效。

当一个 C 函数返回后，Lua 会清理他的栈，所以，有一个原则：永远不要将指向 Lua 字符串的指针保存到访问他们的外部函数中。

Lua_string 返回的字符串结尾总会有一个字符结束标志 0，但是字符串中间也可能包含 0，lua_strlen 返回字符串的实际长度。特殊情况下，假定栈顶的值是一个字符串，下面的断言(assert)总是有效的：

```
const char *s = lua_tostring(L, -1);   /* any Lua string */
size_t l = lua_strlen(L, -1);          /* its length */
assert(s[l] == '\0');
assert(strlen(s) <= l);
```

24.2.3 其他堆栈操作

除开上面所提及的 C 与堆栈交换值的函数外，API 也提供了下列函数来完成通常的堆栈维护工作：

```
int  lua_gettop (lua_State *L);
void lua_settop (lua_State *L, int index);
void lua_pushvalue (lua_State *L, int index);
void lua_remove (lua_State *L, int index);
void lua_insert (lua_State *L, int index);
void lua_replace (lua_State *L, int index);
```

函数 lua_gettop 返回堆栈中的元素个数，它也是栈顶元素的索引。注意一个负数索引-x 对应于正数索引 gettop-x+1。lua_settop 设置栈顶（也就是堆栈中的元素个数）为一个指定的值。如果开始的栈顶高于新的栈顶，顶部的值被丢弃。否则，为了得到指定的大小这个函数压入相应个数的空值（nil）到栈上。特别的，lua_settop(L,0)清空堆栈。你也可以用负数索引作为调用 lua_settop 的参数；那将会设置栈顶到指定的索引。利用这种技巧，API 提供了下面这个宏，它从堆栈中弹出 n 个元素：

```
#define lua_pop(L,n)  lua_settop(L, -(n)-1)
```

函数 lua_pushvalue 压入堆栈上指定索引的一个拷贝到栈顶；lua_remove 移除指定索引位置的元素，并将其上面所有的元素下移来填补这个位置的空白；lua_insert 移动栈顶元素到指定索引的位置，并将这个索引位置上面的元素全部上移至栈顶被移动留下的空隔；最后，lua_replace 从栈顶弹出元素值并将其设置到指定索引位置，没有任何移动操作。注意到下面的操作对堆栈没有任何影响：

```
lua_settop(L, -1);   /* set top to its current value */
lua_insert(L, -1);   /* move top element to the top */
```

为了说明这些函数的用法，这里有一个有用的帮助函数，它 dump 整个堆栈的内容：

```
static void stackDump (lua_State *L) {
```

```
    int i;
    int top = lua_gettop(L);
    for (i = 1; i <= top; i++) {  /* repeat for each level */
        int t = lua_type(L, i);
        switch (t) {

        case LUA_TSTRING:  /* strings */
            printf("`%s'", lua_tostring(L, i));
            break;

        case LUA_TBOOLEAN:  /* booleans */
            printf(lua_toboolean(L, i) ? "true" : "false");
            break;

        case LUA_TNUMBER:  /* numbers */
            printf("%g", lua_tonumber(L, i));
            break;

        default:  /* other values */
            printf("%s", lua_typename(L, t));
            break;

        }
        printf("  ");  /* put a separator */
    }
    printf("\n");     /* end the listing */
}
```

这个函数从栈底到栈顶遍历了整个堆栈，依照每个元素自己的类型打印出其值。它用引号输出字符串；以%g 的格式输出数字；对于其它值（table，函数，等等）它仅仅输出它们的类型（lua_typename 转换一个类型码到类型名）。

下面的函数利用 stackDump 更进一步的说明了 API 堆栈的操作。

```
#include <stdio.h>
#include <lua.h>

static void stackDump (lua_State *L) {
    ...
}
```

```
int main (void) {
    lua_State *L = lua_open();
    lua_pushboolean(L, 1); lua_pushnumber(L, 10);
    lua_pushnil(L); lua_pushstring(L, "hello");
    stackDump(L);
           /* true  10  nil  `hello'  */

    lua_pushvalue(L, -4); stackDump(L);
           /* true  10  nil  `hello'  true  */

    lua_replace(L, 3); stackDump(L);
           /* true  10  true  `hello'  */

    lua_settop(L, 6); stackDump(L);
           /* true  10  true  `hello'  nil  nil  */

    lua_remove(L, -3); stackDump(L);
           /* true  10  true  nil  nil  */

    lua_settop(L, -5); stackDump(L);
           /* true  */

    lua_close(L);
    return 0;
}
```

24.3 C API 的错误处理

不象 C++或者 JAVA 一样，C 语言没有提供一种异常处理机制。为了改善这个难处，Lua 利用 C 的 setjmp 技巧构造了一个类似异常处理的机制。（如果你用 C++来编译 Lua，那么修改代码以使用真正的异常并不困难。）

Lua 中的所有结构都是动态的：它们按需增长，最终当可能时又会缩减。意味着内存分配失败的可能性在 Lua 中是普遍的。几乎任意操作都会面对这种意外。Lua 的 API 中用异常发出这些错误而不是为每步操作产生错误码。这意味着所有的 API 函数可能抛出一个错误（也就是调用 longjmp）来代替返回。

当我们写一个库代码时（也就是被 Lua 调用的 C 函数）长跳转（long jump）的用处几乎和一个真正的异常处理一样的方便，因为 Lua 抓取了任务偶然的错误。当我们写应用程序代码时（也就是调用 Lua 的 C 代码），无论如何，我们必须提供一种方法来抓取

这些错误。

24.3.1 应用程序中的错误处理

典型的情况是应用的代码运行在非保护模式下。由于应用的代码不是被 Lua 调用的，Lua 根据上下文情况来捕捉错误的发生（也就是说，Lua 不能调用 setjmp）。在这些情况下，当 Lua 遇到像 "not enough memory" 的错误，他不知道如何处理。他只能调用一个 panic 函数退出应用。（你可以使用 lua_atpanic 函数设置你自己的 panic 函数）

不是所有的 API 函数都会抛出异常，lua_open、lua_close、lua_pcall 和 lua_load 都是安全的，另外，大多数其他函数只能在内存分配失败的情况下抛出异常：比如，luaL_loadfile 如果没有足够内存来拷贝指定的文件将会失败。有些程序当碰到内存不足时，他们可能需要忽略异常不做任何处理。对这些程序而言，如果 Lua 导致内存不足，panic 是没有问题的。

如果你不想你的应用退出，即使在内存分配失败的情况下，你必须在保护模式下运行你的代码。大部分或者所有你的 Lua 代码通过调用 lua_pcall 来运行，所以，它运行在保护模式下。即使在内存分配失败的情况下，lua_pcall 也返回一个错误代码，使得 lua 解释器处于和谐的（consistent）状态。如果你也想保护所有你的与 Lua 交互的 C 代码，你可以使用 lua_cpcall。（请看参考手册，有对这个函数更深的描述，在 Lua 的发布版的 lua.c 文件中有它应用的例子）

24.3.2 类库中的错误处理

Lua 是安全的语言，也就是说，不管你些什么样的代码，也不管代码如何错误，你都可以根据 Lua 本身知道程序的行为。另外，错误也会根据 Lua 被发现和解释。你可以与 C 比较一下，C 语言中很多错误的程序的行为只能依据硬件或者由程序计数器给出的错误出现的位置被解释。

不论什么时候你向 Lua 中添加一个新的 C 函数，你都可能打破原来的安全性。比如，一个类似 poke 的函数，在任意的内存地址存放任意的字节，可能使得内存瘫痪。你必须想法设法保证你的插件（add-ons）对于 Lua 来讲是安全的，并且提高比较好的错误处理。

正如我们前面所讨论的，每一个 C 程序都有他自己的错勿处理方式，当你打算为 Lua 写一个库函数的时候，这里有一些标准的处理错误的方法可以参考。不论什么时候，C 函数发现错误只要简单的调用 lua_error（或者 luaL_error，后者更好，因为她调用了前者并格式化了错误信息）。Lua_error 函数会清理所有在 Lua 中需要被清理的，然后和错误信息一起回到最初的执行 lua_pcall 的地方。

第 25 章 扩展你的程序

作为配置语言是 LUA 的一个重要应用。在这个章节里,我们举例说明如何用 LUA 设置一个程序。让我们用一个简单的例子开始然后展开到更复杂的应用中。

首先，让我们想象一下一个简单的配置情节：你的 C 程序（程序名为 PP）有一个窗口界面并且可以让用户指定窗口的初始大小。显然，类似这样简单的应用，有多种解决方法比使用 LUA 更简单，比如环境变量或者存有变量值的文件。但，即使是用一个简单的文本文件，你也不知道如何去解析。所以，最后决定采用一个 LUA 配置文件（这就是 LUA 程序中的纯文本文件）。在这种简单的文本形式中通常包含类似如下的信息行：

```
-- configuration file for program `pp'
-- define window size
width = 200
height = 300
```

现在，你得调用 LUA API 函数去解析这个文件，取得 width 和 height 这两个全局变量的值。下面这个取值函数就起这样的作用：

```
#include <lua.h>
#include <lauxlib.h>
#include <lualib.h>

void load (char *filename, int *width, int *height) {
    lua_State *L = lua_open();
    luaopen_base(L);
    luaopen_io(L);
    luaopen_string(L);
    luaopen_math(L);

    if (luaL_loadfile(L, filename) || lua_pcall(L, 0, 0, 0))
        error(L, "cannot run configuration file: %s",
            lua_tostring(L, -1));

    lua_getglobal(L, "width");
    lua_getglobal(L, "height");
    if (!lua_isnumber(L, -2))
        error(L, "`width' should be a number\n");
    if (!lua_isnumber(L, -1))
```

```
        error(L, "`height' should be a number\n");
    *width = (int)lua_tonumber(L, -2);
    *height = (int)lua_tonumber(L, -1);

    lua_close(L);
}
```

首先，程序打开 LUA 包并加载了标准函数库（虽然这是可选的，但通常包含这些库是比较好的编程思想）。然后程序使用 luaL_loadfile 方法根据参数 filename 加载此文件中的信息块并调用 lua_pcall 函数运行，这些函数运行时若发生错误（例如配置文件中有语法错误），将返回非零的错误代码并将此错误信息压入栈中。通常，我们用带参数 index 值为-1 的 lua_tostring 函数取得栈顶元素（error 函数我们已经在 24.1 章节中定义）。

解析完取得的信息块后，程序会取得全局变量值。为此，程序调用了两次 lua_getglobal 函数，其中一参数为变量名称。每调用一次就把相应的变量值压入栈顶，所以变量 width 的 index 值是-2 而变量 height 的 index 值是-1（在栈顶）。（因为先前的栈是空的，需要从栈底重新索引，1 表示第一个元素 2 表示第二个元素。由于从栈顶索引，不管栈是否为空，你的代码也能运行）。接着，程序用 lua_isnumber 函数判断每个值是否为数字。lua_tonumber 函数将得到的数值转换成 double 类型并用(int)强制转换成整型。最后，关闭数据流并返回值。

Lua 是否值得一用?正如我前面提到的，在这个简单的例子中，相比较于 lua 用一个只包含有两个数字的文件会更简单。即使如此，使用 lua 也带来了一些优势。首先，它为你处理所有的语法细节（包括错误）；你的配置文件甚至可以包含注释！其次，用可以用 lua 做更多复杂的配置。例如，脚本可以向用户提示相关信息，或者也可以查询环境变量以选择合适的大小：

```
-- configuration file for program 'pp'
if getenv("DISPLAY") == ":0.0" then
    width = 300; height = 300
else
    width = 200; height = 200
end
```

在这样简单的配置情节中，很难预料用户想要什么；不过只要脚本定义了这两个变量，你的 C 程序无需改变就可运行。

最后一个使用 lua 的理由：在你的程序中很容易的加入新的配置单元。方便的属性添加使程序更具有扩展性。

25.1 表操作

现在，我们打算使用 Lua 作为配置文件，配置窗口的背景颜色。我们假定最终的颜

色有三个数字（RGB）描述，每一个数字代表颜色的一部分。通常，在 C 语言中，这些数字使用[0,255]范围内的整数表示，由于在 Lua 中所有数字都是实数，我们可以使用更自然的范围[0,1]来表示。

一个粗糙的解决方法是，对每一个颜色组件使用一个全局变量表示，让用户来配置这些变量：

```
-- configuration file for program 'pp'
width = 200
height = 300
background_red = 0.30
background_green = 0.10
background_blue = 0
```

这个方法有两个缺点：第一，太冗余（为了表示窗口的背景，窗口的前景，菜单的背景等，一个实际的应用程序可能需要几十个不同的颜色）；第二，没有办法预定义共同部分的颜色，比如，假如我们事先定义了 WHITE，用户可以简单的写 background = WHITE 来表示所有的背景色为白色。为了避免这些缺点，我们使用一个 table 来表示颜色：

```
background = {r=0.30, g=0.10, b=0}
```

表的使用给脚本的结构带来很多灵活性，现在对于用户（或者应用程序）很容易预定义一些颜色，以便将来在配置中使用：

```
BLUE = {r=0, g=0, b=1}
...
background = BLUE
```

为了在 C 中获取这些值，我们这样做：

```
lua_getglobal(L, "background");
if (!lua_istable(L, -1))
    error(L, "`background' is not a valid color table");

red = getfield("r");
green = getfield("g");
blue = getfield("b");
```

一般来说，我们首先获取全局变量 backgroud 的值，并保证它是一个 table。然后，我们使用 getfield 函数获取每一个颜色组件。这个函数不是 API 的一部分，我们需要自己定义他：

```
#define MAX_COLOR      255

/* assume that table is on the stack top */
int getfield (const char *key) {
```

```
    int result;
    lua_pushstring(L, key);
    lua_gettable(L, -2);  /* get background[key] */
    if (!lua_isnumber(L, -1))
        error(L, "invalid component in background color");
    result = (int)lua_tonumber(L, -1) * MAX_COLOR;
    lua_pop(L, 1);  /* remove number */
    return result;
}
```

这里我们再次面对多态的问题：可能存在很多个 getfield 的版本，key 的类型，value 的类型，错误处理等都不尽相同。Lua API 只提供了一个 lua_gettable 函数，他接受 table 在栈中的位置为参数，将对应 key 值出栈，返回与 key 对应的 value。我们上面的 getfield 函数假定 table 在栈顶，因此，lua_pushstring 将 key 入栈之后，table 在-2 的位置。返回之前，getfield 会将栈恢复到调用前的状态。

我们对上面的例子稍作延伸，加入颜色名。用户仍然可以使用颜色 table，但是也可以为共同部分的颜色预定义名字，为了实现这个功能，我们在 C 代码中需要一个颜色 table：

```
struct ColorTable {
    char *name;
    unsigned char red, green, blue;
} colortable[] = {
    {"WHITE",  MAX_COLOR, MAX_COLOR, MAX_COLOR},
    {"RED",    MAX_COLOR, 0,         0},
    {"GREEN",  0,         MAX_COLOR, 0},
    {"BLUE",   0,         0,         MAX_COLOR},
    {"BLACK",  0,         0,         0},
    ...
    {NULL,     0,         0,         0}      /* sentinel */
};
```

我们的这个实现会使用颜色名创建一个全局变量，然后使用颜色 table 初始化这些全局变量。结果和用户在脚本中使用下面这几行代码是一样的：

```
WHITE   = {r=1, g=1, b=1}
RED     = {r=1, g=0, b=0}
...
```

脚本中用户定义的颜色和应用中（C 代码）定义的颜色不同之处在于：应用在脚本之前运行。

为了可以设置 table 域的值，我们定义个辅助函数 setfield；这个函数将 field 的索引

和 field 的值入栈，然后调用 lua_settable：

```
/* assume that table is at the top */
void setfield (const char *index, int value) {
    lua_pushstring(L, index);
    lua_pushnumber(L, (double)value/MAX_COLOR);
    lua_settable(L, -3);
}
```

与其他的 API 函数一样，lua_settable 在不同的参数类型情况下都可以使用，他从栈中获取所有的参数。lua_settable 以 table 在栈中的索引作为参数，并将栈中的 key 和 value 出栈，用这两个值修改 table。Setfield 函数假定调用之前 table 是在栈顶位置(索引为-1)。将 index 和 value 入栈之后，table 索引变为-3。

Setcolor 函数定义一个单一的颜色，首先创建一个 table，然后设置对应的域，然后将这个 table 赋值给对应的全局变量：

```
void setcolor (struct ColorTable *ct) {
    lua_newtable(L);              /* creates a table */
    setfield("r", ct->red);       /* table.r = ct->r */
    setfield("g", ct->green);     /* table.g = ct->g */
    setfield("b", ct->blue);      /* table.b = ct->b */
    lua_setglobal(ct->name);      /* 'name' = table */
}
```

lua_newtable 函数创建一个新的空 table 然后将其入栈，调用 setfield 设置 table 的域，最后 lua_setglobal 将 table 出栈并将其赋给一个全局变量名。

有了前面这些函数，下面的循环注册所有的颜色到应用程序中的全局变量：

```
int i = 0;
while (colortable[i].name != NULL)
    setcolor(&colortable[i++]);
```

记住：应用程序必须在运行用户脚本之前，执行这个循环。

对于上面的命名颜色的实现有另外一个可选的方法。用一个字符串来表示颜色名，而不是上面使用全局变量表示，比如用户可以这样设置 background = "BLUE"。所以，background 可以是 table 也可以是 string。对于这种实现，应用程序在运行用户脚本之前不需要做任何特殊处理。但是需要额外的工作来获取颜色。当他得到变量 background 的值之后，必须判断这个值的类型，是 table 还是 string：

```
lua_getglobal(L, "background");
if (lua_isstring(L, -1)) {
    const char *name = lua_tostring(L, -1);
    int i = 0;
```

```
    while (colortable[i].name != NULL &&
        strcmp(colorname, colortable[i].name) != 0)
        i++;
    if (colortable[i].name == NULL)  /* string not found? */
        error(L, "invalid color name (%s)", colorname);
    else {  /* use colortable[i] */
        red = colortable[i].red;
        green = colortable[i].green;
        blue = colortable[i].blue;
    }
} else if (lua_istable(L, -1)) {
    red = getfield("r");
    green = getfield("g");
    blue = getfield("b");
} else
    error(L, "invalid value for `background'");
```

哪个是最好的选择呢？在 C 程序中，使用字符串表示不是一个好的习惯，因为编译器不会对字符串进行错误检查。然而在 Lua 中，全局变量不需要声明，因此当用户将颜色名字拼写错误的时候，Lua 不会发出任何错误信息。比如，用户将 WHITE 误写成 WITE，background 变量将为 nil(WITE 的值没有初始化)，然后应用程序就认为 background 的值为 nil。没有其他关于这个错误的信息可以获得。另一方面，使用字符串表示，background 的值也可能是拼写错了的字符串。因此，应用程序可以在发生错误的时候，定制输出的错误信息。应用可以不区分大小写比较字符串，因此，用户可以写"white"，"WHITE"，甚至"White"。但是，如果用户脚本很小，并且颜色种类比较多，注册成百上千个颜色(需要创建成百上千个 table 和全局变量)，最终用户可能只是用其中几个，这会让人觉得很怪异。在使用字符串表示的时候，应避免这种情况出现。

25.2 调用 Lua 函数

Lua 作为配置文件的一个最大的长处在于它可以定义个被应用调用的函数。比如，你可以写一个应用程序来绘制一个函数的图像，使用 Lua 来定义这个函数。

使用 API 调用函数的方法是很简单的：首先，将被调用的函数入栈；第二，依次将所有参数入栈；第三，使用 lua_pcall 调用函数；最后，从栈中获取函数执行返回的结果。

看一个例子，假定我们的配置文件有下面这个函数：

```
function f (x, y)
    return (x^2 * math.sin(y))/(1 - x)
end
```

并且我们想在 C 中对于给定的 x,y 计算 z=f(x,y)的值。假如你已经打开了 lua 库并且运行了配置文件，你可以将这个调用封装成下面的 C 函数：

```
/* call a function `f' defined in Lua */
double f (double x, double y) {
    double z;

    /* push functions and arguments */
    lua_getglobal(L, "f");   /* function to be called */
    lua_pushnumber(L, x);    /* push 1st argument */
    lua_pushnumber(L, y);    /* push 2nd argument */

    /* do the call (2 arguments, 1 result) */
    if (lua_pcall(L, 2, 1, 0) != 0)
        error(L, "error running function `f': %s",
            lua_tostring(L, -1));

    /* retrieve result */
    if (!lua_isnumber(L, -1))
        error(L, "function `f' must return a number");
    z = lua_tonumber(L, -1);
    lua_pop(L, 1);  /* pop returned value */
    return z;
}
```

可以调用 lua_pcall 时指定参数的个数和返回结果的个数。第四个参数可以指定一个错误处理函数，我们下面再讨论它。和 Lua 中赋值操作一样，lua_pcall 会根据你的要求调整返回结果的个数，多余的丢弃，少的用 nil 补足。在将结果入栈之前，lua_pcall 会将栈内的函数和参数移除。如果函数返回多个结果，第一个结果被第一个入栈，因此如果有 n 个返回结果，第一个返回结果在栈中的位置为-n，最后一个返回结果在栈中的位置为-1。

如果 lua_pcall 运行时出现错误，lua_pcall 会返回一个非 0 的结果。另外，他将错误信息入栈（仍然会先将函数和参数从栈中移除）。在将错误信息入栈之前，如果指定了错误处理函数，lua_pcall 毁掉用错误处理函数。使用 lua_pcall 的最后一个参数来指定错误处理函数，0 代表没有错误处理函数，也就是说最终的错误信息就是原始的错误信息。否则，那个参数应该是一个错误函数被加载的时候在栈中的索引，注意，在这种情况下，错误处理函数必须要在被调用函数和其参数入栈之前入栈。

对于一般错误，lua_pcall 返回错误代码 LUA_ERRRUN。有两种特殊情况，会返回特殊的错误代码，因为他们从来不会调用错误处理函数。第一种情况是，内存分配错误，对于这种错误，lua_pcall 总是返回 LUA_ERRMEM。第二种情况是，当 Lua 正在运行错

误处理函数时发生错误，这种情况下，再次调用错误处理函数没有意义，所以 lua_pcall 立即返回错误代码 LUA_ERRERR。

25.3 通用的函数调用

看一个稍微高级的例子，我们使用 C 的 vararg 来封装对 Lua 函数的调用。我们的封装后的函数（call_va）接受被调用的函数明作为第一个参数，第二参数是一个描述参数和结果类型的字符串，最后是一个保存返回结果的变量指针的列表。使用这个函数，我们可以将前面的例子改写为：

```
call_va("f", "dd>d", x, y, &z);
```

字符串 "dd>d" 表示函数有两个 double 类型的参数，一个 double 类型的返回结果。我们使用字母 'd' 表示 double；'i' 表示 integer，'s' 表示 strings；'>' 作为参数和结果的分隔符。如果函数没有返回结果，'>' 是可选的。

```
#include <stdarg.h>

void call_va (const char *func, const char *sig, ...) {
    va_list vl;
    int narg, nres;   /* number of arguments and results */

    va_start(vl, sig);
    lua_getglobal(L, func);  /* get function */

    /* push arguments */
    narg = 0;
    while (*sig) {    /* push arguments */
        switch (*sig++) {

        case 'd':  /* double argument */
            lua_pushnumber(L, va_arg(vl, double));
            break;

        case 'i':  /* int argument */
            lua_pushnumber(L, va_arg(vl, int));
            break;

        case 's':  /* string argument */
            lua_pushstring(L, va_arg(vl, char *));
            break;
```

```
        case '>':
            goto endwhile;

        default:
            error(L, "invalid option (%c)", *(sig - 1));
        }
        narg++;
        luaL_checkstack(L, 1, "too many arguments");
    } endwhile:

    /* do the call */
    nres = strlen(sig);  /* number of expected results */
    if (lua_pcall(L, narg, nres, 0) != 0)  /* do the call */
        error(L, "error running function `%s': %s",
            func, lua_tostring(L, -1));

    /* retrieve results */
    nres = -nres;     /* stack index of first result */
    while (*sig) {    /* get results */
        switch (*sig++) {

        case 'd':  /* double result */
            if (!lua_isnumber(L, nres))
                error(L, "wrong result type");
            *va_arg(vl, double *) = lua_tonumber(L, nres);
            break;

        case 'i':  /* int result */
            if (!lua_isnumber(L, nres))
                error(L, "wrong result type");
            *va_arg(vl, int *) = (int)lua_tonumber(L, nres);
            break;

        case 's':  /* string result */
            if (!lua_isstring(L, nres))
                error(L, "wrong result type");
            *va_arg(vl, const char **) = lua_tostring(L, nres);
            break;
```

```
        default:
            error(L, "invalid option (%c)", *(sig - 1));
        }
        nres++;
    }
    va_end(vl);
}
```

尽管这段代码具有一般性，这个函数和前面我们的例子有相同的步骤：将函数入栈，参数入栈，调用函数，获取返回结果。大部分代码都很直观，但也有一点技巧。首先，不需要检查 func 是否是一个函数，lua_pcall 可以捕捉这个错误。第二，可以接受任意多个参数，所以必须检查栈的空间。第三，因为函数可能返回字符串，call_va 不能从栈中弹出结果，在调用者获取临时字符串的结果之后（拷贝到其他的变量中），由调用者负责弹出结果。

第 26 章 调用 C 函数

扩展 Lua 的基本方法之一就是为应用程序注册新的 C 函数到 Lua 中去。

当我们提到 Lua 可以调用 C 函数，不是指 Lua 可以调用任何类型的 C 函数（有一些包可以让 Lua 调用任意的 C 函数，但缺乏便捷和健壮性）。正如我们前面所看到的，当 C 调用 Lua 函数的时候，必须遵循一些简单的协议来传递参数和获取返回结果。相似的，从 Lua 中调用 C 函数，也必须遵循一些协议来传递参数和获得返回结果。另外，从 Lua 调用 C 函数我们必须注册函数，也就是说，我们必须把 C 函数的地址以一个适当的方式传递给 Lua 解释器。

当 Lua 调用 C 函数的时候，使用和 C 调用 Lua 相同类型的栈来交互。C 函数从栈中获取她的参数，调用结束后将返回结果放到栈中。为了区分返回结果和栈中的其他的值，每个 C 函数还会返回结果的个数（the function returns (in C) the number of results it is leaving on the stack.）。这儿有一个重要的概念：用来交互的栈不是全局变量，每一个函数都有他自己的私有栈。当 Lua 调用 C 函数的时候，第一个参数总是在这个私有栈的 index=1 的位置。甚至当一个 C 函数调用 Lua 代码（Lua 代码调用同一个 C 函数或者其他的 C 函数），每一个 C 函数都有自己的独立的私有栈，并且第一个参数在 index=1 的位置。

26.1 C 函数

先看一个简单的例子，如何实现一个简单的函数返回给定数值的 sin 值（更专业的实现应该检查他的参数是否为一个数字）：

```
static int l_sin (lua_State *L) {
    double d = lua_tonumber(L, 1);  /* get argument */
    lua_pushnumber(L, sin(d));      /* push result */
    return 1;                       /* number of results */
}
```

任何在 Lua 中注册的函数必须有同样的原型，这个原型声明定义就是 lua.h 中的 lua_CFunction：

```
typedef int (*lua_CFunction) (lua_State *L);
```

从 C 的角度来看，一个 C 函数接受单一的参数 Lua state，返回一个表示返回值个数的数字。所以，函数在将返回值入栈之前不需要清理栈，函数返回之后，Lua 自动的清除栈中返回结果下面的所有内容。

我们要想在 Lua 使用这个函数，还必须首先注册这个函数。我们使用

lua_pushcfunction 来完成这个任务：他获取指向 C 函数的指针，并在 Lua 中创建一个 function 类型的值来表示这个函数。一个 quick-and-dirty 的解决方案是将这段代码直接放到 lua.c 文件中，并在调用 lua_open 后面适当的位置加上下面两行：

```
lua_pushcfunction(l, l_sin);
lua_setglobal(l, "mysin");
```

第一行将类型为 function 的值入栈，第二行将 function 赋值给全局变量 mysin。这样修改之后，重新编译 Lua，你就可以在你的 Lua 程序中使用新的 mysin 函数了。在下面一节，我们将讨论以比较好的方法将新的 C 函数添加到 Lua 中去。

对于稍微专业点的 sin 函数，我们必须检查 sin 的参数的类型。有一个辅助库中的 luaL_checknumber 函数可以检查给定的参数是否为数字：当有错误发生的时候，将抛出一个错误信息；否则返回作为参数的那个数字。将上面我们的函数稍作修改：

```
static int l_sin (lua_State *L) {
    double d = luaL_checknumber(L, 1);
    lua_pushnumber(L, sin(d));
    return 1;  /* number of results */
}
```

根据上面的定义，如果你调用 mysin('a'),会得到如下信息：

```
bad argument #1 to 'mysin' (number expected, got string)
```

注意看看 luaL_checknumber 是如何自动使用：参数 number（1），函数名（"mysin"），期望的参数类型（"number"），实际的参数类型（"string"）来拼接最终的错误信息的。

下面看一个稍微复杂的例子：写一个返回给定目录内容的函数。Lua 的标准库并没有提供这个函数，因为 ANSI C 没有可以实现这个功能的函数。在这儿，我们假定我们的系统符合 POSIX 标准。我们的 dir 函数接受一个代表目录路径的字符串作为参数，以数组的形式返回目录的内容。比如，调用 dir("/home/lua")可能返回{".", "..", "src", "bin", "lib"}。当有错误发生的时候，函数返回 nil 加上一个描述错误信息的字符串。

```
#include <dirent.h>
#include <errno.h>

static int l_dir (lua_State *L) {
    DIR *dir;
    struct dirent *entry;
    int i;
    const char *path = luaL_checkstring(L, 1);

    /* open directory */
    dir = opendir(path);
```

```
    if (dir == NULL) {   /* error opening the directory? */
        lua_pushnil(L);   /* return nil and ... */
        lua_pushstring(L, strerror(errno)); /* error message */
        return 2;  /* number of results */
    }

    /* create result table */
    lua_newtable(L);
    i = 1;
    while ((entry = readdir(dir)) != NULL) {
        lua_pushnumber(L, i++);            /* push key */
        lua_pushstring(L, entry->d_name);  /* push value */
        lua_settable(L, -3);
    }

    closedir(dir);
    return 1;         /* table is already on top */
}
```

辅助库的 luaL_checkstring 函数用来检测参数是否为字符串，与 luaL_checknumber 类似。（在极端情况下，上面的 l_dir 的实现可能会导致小的内存泄漏。调用的三个 Lua 函数 lua_newtable、lua_pushstring 和 lua_settable 可能由于没有足够的内存而失败。其中任何一个调用失败都会抛出错误并且终止 l_dir，这种情况下，不会调用 closedir。正如前面我们所讨论过的，对于大多数程序来说这不算个问题：如果程序导致内存不足，最好的处理方式是立即终止程序。另外，在 29 章我们将看到另外一种解决方案可以避免这个问题的发生）

26.2 C 函数库

一个 Lua 库实际上是一个定义了一系列 Lua 函数的 chunk，并将这些函数保存在适当的地方，通常作为 table 的域来保存。Lua 的 C 库就是这样实现的。除了定义 C 函数之外，还必须定义一个特殊的用来和 Lua 库的主 chunk 通信的特殊函数。一旦调用，这个函数就会注册库中所有的 C 函数，并将他们保存到适当的位置。像一个 Lua 主 chunk 一样，她也会初始化其他一些在库中需要初始化的东西。

Lua 通过这个注册过程，就可以看到库中的 C 函数。一旦一个 C 函数被注册之后并保存到 Lua 中，在 Lua 程序中就可以直接引用他的地址（当我们注册这个函数的时候传递给 Lua 的地址）来访问这个函数了。换句话说，一旦 C 函数被注册之后，Lua 调用这个函数并不依赖于函数名，包的位置，或者调用函数的可见的规则。通常 C 库都有一个外部（public/extern）的用来打开库的函数。其他的函数可能都是私有的，在 C 中被声明

为 static。

当你打算使用 C 函数来扩展 Lua 的时候，即使你仅仅只想注册一个 C 函数，将你的 C 代码设计为一个库是个比较好的思想：不久的将来你就会发现你需要其他的函数。一般情况下，辅助库对这种实现提供了帮助。luaL_openlib 函数接受一个 C 函数的列表和他们对应的函数名，并且作为一个库在一个 table 中注册所有这些函数。看一个例子，假定我们想用一个我们前面提过的 l_dir 函数创建一个库。首先，我们必须定义库函数：

```
static int l_dir (lua_State *L) {
    ...    /* as before */
}
```

第二步，我们声明一个数组，保存所有的函数和他们对应的名字。这个数组的元素类型为 luaL_reg：是一个带有两个域的结构体，一个字符串和一个函数指针。

```
static const struct luaL_reg mylib [] = {
    {"dir", l_dir},
    {NULL, NULL}  /* sentinel */
};
```

在我们的例子中，只有一个函数 l_dir 需要声明。注意数组中最后一对必须是{NULL, NULL}，用来表示结束。第三步，我们使用 luaL_openlib 声明主函数：

```
int luaopen_mylib (lua_State *L) {
    luaL_openlib(L, "mylib", mylib, 0);
    return 1;
}
```

luaL_openlib 的第二个参数是库的名称。这个函数按照指定的名字创建(或者 reuse)一个表，并使用数组 mylib 中的 name-function 对填充这个表。luaL_openlib 还允许我们为库中所有的函数注册公共的 upvalues。例子中不需要使用 upvalues，所以最后一个参数为 0。luaL_openlib 返回的时候，将保存库的表放到栈内。luaL_openlib 函数返回 1，返回这个值给 Lua。（The luaopen_mylib function returns 1 to return this value to Lua）（和 Lua 库一样，这个返回值是可选的，因为库本身已经赋给了一个全局变量。另外，像在 Lua 标准库中的一样，这个返回不会有额外的花费，在有时候可能是有用的。）

完成库的代码编写之后，我们必须将它链接到 Lua 解释器。最常用的方式使用动态连接库，如果你的 Lua 解释器支持这个特性的话（我们在 8.2 节已经讨论过了动态连接库）。在这种情况下，你必须用你的代码创建动态连接库（windows 下.dll 文件，linux 下.so 文件）。到这一步，你就可以在 Lua 中直接使用 loadlib 加载你刚才定义的函数库了，下面这个调用：

```
mylib = loadlib("fullname-of-your-library", "luaopen_mylib")
```

将 luaopen_mylib 函数转换成 Lua 中的一个 C 函数，并将这个函数赋值给 mylib（那就是为什么 luaopen_mylib 必须和其他的 C 函数有相同的原型的原因所在）。然后，调用

mylib()，将运行 luaopen_mylib 打开你定义的函数库。

如果你的解释器不支持动态链接库，你必须将你的新的函数库重新编译到你的 Lua 中去。除了这以外，还不要一些方式告诉独立运行的 Lua 解释器，当他打开一个新的状态的时候必须打开这个新定义的函数库。宏定义可以很容易实现这个功能。第一，你必须使用下面的内容创建一个头文件（我们可以称之为 mylib.h）：

```
int luaopen_mylib (lua_State *L);

#define LUA_EXTRALIBS { "mylib", luaopen_mylib },
```

第一行声明了打开库的函数。第二行定义了一个宏 LUA_EXTRALIBS 作为函数数组的新的入口，当解释器创建新的状态的时候会调用这个宏。（这个函数数组的类型为 struct luaL_reg[]，因此我们需要将名字也放进去）

为了在解释器中包含这个头文件，你可以在你的编译选项中定义一个宏 LUA_USERCONFIG。对于命令行的编译器，你只需添加一个下面这样的选项即可：

```
-DLUA_USERCONFIG=\"mylib.h\"
```

（反斜线防止双引号被 shell 解释，当我们在 C 中指定一个头文件时，这些引号是必需的。）在一个整合的开发环境中，你必须在工程设置中添加类似的东西。然后当你重新编译 lua.c 的时候，它包含 mylib.h，并且因此在函数库的列表中可以用新定义的 LUA_EXTRALIBS 来打开函数库。

第 27 章 撰写 C 函数的技巧

官方的 API 和辅助函数库 都提供了一些帮助程序员如何写好 C 函数的机制。在这一章我们将讨论数组操纵、string 处理、在 C 中存储 Lua 值等一些特殊的机制。

27.1 数组操作

Lua 中数组实际上就是以特殊方式使用的 table 的别名。我们可以使用任何操纵 table 的函数来对数组操作，即 lua_settable 和 lua_gettable。然而，与 Lua 常规简洁思想（economy and simplicity）相反的是，API 为数组操作提供了一些特殊的函数。这样做的原因出于性能的考虑：因为我们经常在一个算法（比如排序）的循环的内层访问数组，所以这种内层操作的性能的提高会对整体的性能的改善有很大的影响。

API 提供了下面两个数组操作函数：

```
void lua_rawgeti (lua_State *L, int index, int key);
void lua_rawseti (lua_State *L, int index, int key);
```

关于的 lua_rawgeti 和 lua_rawseti 的描述有些使人糊涂，因为它涉及到两个索引：index 指向 table 在栈中的位置；key 指向元素在 table 中的位置。当 t 使用负索引的时候（otherwise，you must compensate for the new item in the stack），调用 lua_rawgeti(L,t,key) 等价于：

```
lua_pushnumber(L, key);
lua_rawget(L, t);
```

调用 lua_rawseti(L, t, key)（也要求 t 使用负索引）等价于：

```
lua_pushnumber(L, key);
lua_insert(L, -2);   /* put 'key' below previous value */
lua_rawset(L, t);
```

注意这两个函数都是用 raw 操作，他们的速度较快，总之，用作数组的 table 很少使用 metamethods。

下面看如何使用这些函数的具体的例子，我们将前面的 l_dir 函数的循环体：

```
lua_pushnumber(L, i++);             /* key */
lua_pushstring(L, entry->d_name);   /* value */
lua_settable(L, -3);
```

改写为：

```
lua_pushstring(L, entry->d_name);  /* value */
lua_rawseti(L, -2, i++);           /* set table at key 'i' */
```

下面是一个更完整的例子，下面的代码实现了 map 函数：以数组的每一个元素为参数调用一个指定的函数，并将数组的该元素替换为调用函数返回的结果。

```
int l_map (lua_State *L) {
    int i, n;

    /* 1st argument must be a table (t) */
    luaL_checktype(L, 1, LUA_TTABLE);

    /* 2nd argument must be a function (f) */
    luaL_checktype(L, 2, LUA_TFUNCTION);

    n = luaL_getn(L, 1);  /* get size of table */

    for (i=1; i<=n; i++) {
        lua_pushvalue(L, 2);     /* push f */
        lua_rawgeti(L, 1, i);    /* push t[i] */
        lua_call(L, 1, 1);       /* call f(t[i]) */
        lua_rawseti(L, 1, i);    /* t[i] = result */
    }

    return 0;  /* no results */
}
```

这里面引入了三个新的函数。luaL_checktype（在 lauxlib.h 中定义）用来检查给定的参数有指定的类型；否则抛出错误。luaL_getn 函数栈中指定位置的数组的大小（table.getn 是调用 luaL_getn 来完成工作的）。lua_call 的运行是无保护的，他与 lua_pcall 相似，但是在错误发生的时候她抛出错误而不是返回错误代码。当你在应用程序中写主流程的代码时，不应该使用 lua_call，因为你应该捕捉任何可能发生的错误。当你写一个函数的代码时，使用 lua_call 是比较好的想法，如果有错误发生，把错误留给关心她的人去处理。

27.2 字符串处理

当 C 函数接受一个来自 lua 的字符串作为参数时，有两个规则必须遵守：当字符串正在被访问的时候不要将其出栈；永远不要修改字符串。

当 C 函数需要创建一个字符串返回给 lua 的时候，情况变得更加复杂。这样需要由 C 代码来负责缓冲区的分配和释放，负责处理缓冲溢出等情况。然而，Lua API 提供了

一些函数来帮助我们处理这些问题。

标准 API 提供了对两种基本字符串操作的支持：子串截取和字符串连接。记住，lua_pushlstring 可以接受一个额外的参数，字符串的长度来实现字符串的截取，所以，如果你想将字符串 s 从 i 到 j 位置（包含 i 和 j）的子串传递给 lua，只需要：

```
lua_pushlstring(L, s+i, j-i+1);
```

下面这个例子，假如你想写一个函数来根据指定的分隔符分割一个字符串，并返回一个保存所有子串的 table，比如调用：

```
split("hi,,there", ",")
```

应该返回表{"hi", "", "there"}。我们可以简单的实现如下，下面这个函数不需要额外的缓冲区，可以处理字符串的长度也没有限制。

```
static int l_split (lua_State *L) {
    const char *s = luaL_checkstring(L, 1);
    const char *sep = luaL_checkstring(L, 2);
    const char *e;
    int i = 1;

    lua_newtable(L);  /* result */

    /* repeat for each separator */
    while ((e = strchr(s, *sep)) != NULL) {
        lua_pushlstring(L, s, e-s);  /* push substring */
        lua_rawseti(L, -2, i++);
        s = e + 1;  /* skip separator */
    }

    /* push last substring */
    lua_pushstring(L, s);
    lua_rawseti(L, -2, i);

    return 1;  /* return the table */
}
```

在 Lua API 中提供了专门的用来连接字符串的函数 lua_concat。等价于 Lua 中的..操作符：自动将数字转换成字符串，如果有必要的时候还会自动调用 metamethods。另外，她可以同时连接多个字符串。调用 lua_concat(L,n)将连接(同时会出栈)栈顶的 n 个值，并将最终结果放到栈顶。

另一个有用的函数是 lua_pushfstring：

```
const char *lua_pushfstring (lua_State *L,
                             const char *fmt, ...);
```

这个函数某种程度上类似于 C 语言中的 sprintf，根据格式串 fmt 的要求创建一个新的字符串。与 sprintf 不同的是，你不需要提供一个字符串缓冲数组，Lua 为你动态的创建新的字符串，按他实际需要的大小。也不需要担心缓冲区溢出等问题。这个函数会将结果字符串放到栈内，并返回一个指向这个结果串的指针。当前，这个函数只支持下列几个指示符： %%（表示字符 '%'）、%s（用来格式化字符串）、%d（格式化整数）、%f（格式化 Lua 数字，即 doubles）和 %c（接受一个数字并将其作为字符），不支持宽度和精度等选项。

当我们打算连接少量的字符串的时候，lua_concat 和 lua_pushfstring 是很有用的，然而，如果我们需要连接大量的字符串（或者字符），这种一个一个的连接方式效率是很低的，正如我们在 11.6 节看到的那样。我们可以使用辅助库提供的 buffer 相关函数来解决这个问题。Auxlib 在两个层次上实现了这些 buffer。第一个层次类似于 I/O 操作的 buffers：集中所有的字符串（或者但个字符）放到一个本地 buffer 中，当本地 buffer 满的时候将其传递给 Lua（使用 lua_pushlstring）。第二个层次使用 lua_concat 和我们在 11.6 节中看到的那个栈算法的变体，来连接多个 buffer 的结果。

为了更详细地描述 Auxlib 中的 buffer 的使用，我们来看一个简单的应用。下面这段代码显示了 string.upper 的实现（来自文件 lstrlib.c）：

```
static int str_upper (lua_State *L) {
    size_t l;
    size_t i;
    luaL_Buffer b;
    const char *s = luaL_checklstr(L, 1, &l);
    luaL_buffinit(L, &b);
    for (i=0; i<l; i++)
        luaL_putchar(&b, toupper((unsigned char)(s[i])));
    luaL_pushresult(&b);
    return 1;
}
```

使用 Auxlib 中 buffer 的第一步是使用类型 luaL_Buffer 声明一个变量，然后调用 luaL_buffinit 初始化这个变量。初始化之后，buffer 保留了一份状态 L 的拷贝，因此当我们调用其他操作 buffer 的函数的时候不需要传递 L。宏 luaL_putchar 将一个单个字符放入 buffer。Auxlib 也提供了 luaL_addlstring 以一个显示的长度将一个字符串放入 buffer，而 luaL_addstring 将一个以 0 结尾的字符串放入 buffer。最后，luaL_pushresult 刷新 buffer 并将最终字符串放到栈顶。这些函数的原型如下：

```
void luaL_buffinit (lua_State *L, luaL_Buffer *B);
void luaL_putchar (luaL_Buffer *B, char c);
```

```
void luaL_addlstring (luaL_Buffer *B, const char *s, size_t l);
void luaL_addstring (luaL_Buffer *B, const char *s);
void luaL_pushresult (luaL_Buffer *B);
```

使用这些函数，我们不需要担心 buffer 的分配，溢出等详细信息。正如我们所看到的，连接算法是有效的。函数 str_upper 可以毫无问题的处理大字符串（大于 1MB）。

当你使用auxlib中的buffer时，不必担心一点细节问题。你只要将东西放入buffer，程序会自动在Lua栈中保存中间结果。所以，你不要认为栈顶会保持你开始使用buffer的那个状态。另外，虽然你可以在使用buffer的时候，将栈用作其他用途，但每次你访问buffer的时候，这些其他用途的操作进行的push/pop操作必须保持平衡[8]。有一种情况，即你打算将从Lua返回的字符串放入buffer时，这种情况下，这些限制有些过于严格。这种情况下，在将字符串放入buffer之前，不能将字符串出栈，因为一旦你从栈中将来自于Lua的字符串移出，你就永远不能使用这个字符串。同时，在将一个字符串出栈之前，你也不能够将其放入buffer，因为那样会将栈置于错误的层次（because then the stack would be in the wrong level）。换句话说你不能做类似下面的事情：

```
luaL_addstring(&b, lua_tostring(L, 1));   /* BAD CODE */
```

（译者：上面正好构成了一对矛盾），由于这种情况是很常见的，auxlib 提供了特殊的函数来将位于栈顶的值放入 buffer：

```
void luaL_addvalue (luaL_Buffer *B);
```

当然，如果位于栈顶的值不是字符串或者数字的话，调用这个函数将会出错。

27.3 在 C 函数中保存状态

通常来说，C 函数需要保留一些非局部的数据，也就是指那些超过他们作用范围的数据。C 语言中我们使用全局变量或者 static 变量来满足这种需要。然而当你为 Lua 设计一个程序库的时候，全局变量和 static 变量不是一个好的方法。首先，不能将所有的（一般意义的，原文 generic）Lua 值保存到一个 C 变量中。第二，使用这种变量的库不能在多个 Lua 状态的情况下使用。

一个替代的解决方案是将这些值保存到一个 Lua 全局变量中，这种方法解决了前面的两个问题。Lua 全局变量可以存放任何类型的 Lua 值，并且每一个独立的状态都有他自己独立的全局变量集。然而，并不是在所有情况下，这种方法都是令人满意地解决方案，因为 Lua 代码可能会修改这些全局变量，危及 C 数据的完整性。为了避免这个问题，Lua 提供了一个独立的被称为 registry 的表，C 代码可以自由使用，但 Lua 代码不能访问他。

[8] 译注：即有多少次 push 就要有多少次 pop。

27.3.1 The Registry

registry 一直位于一个由 LUA_REGISTRYINDEX 定义的值所对应的假索引(pseudo-index)的位置。一个假索引除了他对应的值不在栈中之外，其他都类似于栈中的索引。Lua API 中大部分接受索引作为参数的函数，也都可以接受假索引作为参数—除了那些操作栈本身的函数，比如 lua_remove，lua_insert。例如，为了获取以键值 "Key" 保存在 registry 中的值，使用下面的代码：

```
lua_pushstring(L, "Key");
lua_gettable(L, LUA_REGISTRYINDEX);
```

registry 就是普通的 Lua 表，因此，你可以使用任何非 nil 的 Lua 值来访问她的元素。然而，由于所有的 C 库共享相同的 registry ，你必须注意使用什么样的值作为 key，否则会导致命名冲突。一个防止命名冲突的方法是使用 static 变量的地址作为 key：C 链接器保证在所有的库中这个 key 是唯一的。函数 lua_pushlightuserdata 将一个代表 C 指针的值放到栈内，下面的代码展示了使用上面这个方法，如何从 registry 中获取变量和向 registry 存储变量：

```
/* variable with an unique address */
static const char Key = 'k';

/* store a number */
lua_pushlightuserdata(L, (void *)&Key);  /* push address */
lua_pushnumber(L, myNumber);  /* push value */
/* registry[&Key] = myNumber */
lua_settable(L, LUA_REGISTRYINDEX);

/* retrieve a number */
lua_pushlightuserdata(L, (void *)&Key);   /* push address */
lua_gettable(L, LUA_REGISTRYINDEX);  /* retrieve value */
myNumber = lua_tonumber(L, -1);  /* convert to number */
```

我们会在 28.5 节中更详细的讨论 light userdata。

当然，你也可以使用字符串作为 registry 的 key，只要你保证这些字符串唯一。当你打算允许其他的独立库房问你的数据的时候，字符串型的 key 是非常有用的，因为他们需要知道 key 的名字。对这种情况，没有什么方法可以绝对防止名称冲突，但有一些好的习惯可以采用，比如使用库的名称作为字符串的前缀等类似的方法。类似 lua 或者 lualib 的前缀不是一个好的选择。另一个可选的方法是使用 universal unique identifier（uuid），很多系统都有专门的程序来产生这种标示符（比如 linux 下的 uuidgen）。一个 uuid 是一个由本机 IP 地址、时间戳、和一个随机内容组合起来的 128 位的数字（以 16 进制的方式书写，用来形成一个字符串），因此它与其他的 uuid 不同是可以保证的。

27.3.2 References

你应该记住，永远不要使用数字作为 registry 的 key，因为这种类型的 key 是保留给 reference 系统使用。Reference 系统是由辅助库中的一对函数组成，这对函数用来不需要担心名称冲突的将值保存到 registry 中去。（实际上，这些函数可以用于任何一个表，但他们典型的被用于 registry）

调用

```
int r = luaL_ref(L, LUA_REGISTRYINDEX);
```

从栈中弹出一个值，以一个新的数字作为 key 将其保存到 registry 中，并返回这个 key。我们将这个 key 称之为 reference。

顾名思义，我们使用 references 主要用于：将一个指向 Lua 值的 reference 存储到一个 C 结构体中。正如前面我们所见到的，我们永远不要将一个指向 Lua 字符串的指针保存到获取这个字符串的外部的 C 函数中。另外，Lua 甚至不提供指向其他对象的指针，比如 table 或者函数。因此，我们不能通过指针指向 Lua 对象。当我们需要这种指针的时候，我们创建一个 reference 并将其保存在 C 中。

要想将一个 reference 的对应的值入栈，只需要：

```
lua_rawgeti(L, LUA_REGISTRYINDEX, r);
```

最后，我们调用下面的函数释放值和 reference：

```
luaL_unref(L, LUA_REGISTRYINDEX, r);
```

调用这个之后，luaL_ref 可以再次返回 r 作为一个新的 reference。

reference 系统将 nil 作为特殊情况对待，不管什么时候，你以 nil 调用 luaL_ref 的话，不会创建一新的 reference ，而是返回一个常量 reference LUA_REFNIL。下面的调用没有效果：

```
luaL_unref(L, LUA_REGISTRYINDEX, LUA_REFNIL);
```

然而

```
lua_rawgeti(L, LUA_REGISTRYINDEX, LUA_REFNIL);
```

像预期的一样，将一个 nil 入栈。

reference 系统也定义了常量 LUA_NOREF，她是一个表示任何非有效的 reference 的整数值，用来标记无效的 reference。任何企图获取 LUA_NOREF 返回 nil，任何释放他的操作都没有效果。

27.3.3 Upvalues

registry 实现了全局的值，upvalue 机制实现了与 C static 变量等价的东东，这种变量

只能在特定的函数内可见。每当你在 Lua 中创建一个新的 C 函数，你可以将这个函数与任意多个 upvalues 联系起来，每一个 upvalue 可以持有一个单独的 Lua 值。下面当函数被调用的时候，可以通过假索引自由的访问任何一个 upvalues。

我们称这种一个 C 函数和她的 upvalues 的组合为闭包（closure）。记住：在 Lua 代码中，一个闭包是一个从外部函数访问局部变量的函数。一个 C 闭包与一个 Lua 闭包相近。关于闭包的一个有趣的事实是，你可以使用相同的函数代码创建不同的闭包，带有不同的 upvalues。

看一个简单的例子，我们在 C 中创建一个 newCounter 函数。（我们已经在 6.1 节部分在 Lua 中定义过同样的函数）。这个函数是个函数工厂：每次调用他都返回一个新的 counter 函数。尽管所有的 counters 共享相同的 C 代码，但是每个都保留独立的 counter 变量，工厂函数如下：

```
/* forward declaration */
static int counter (lua_State *L);

int newCounter (lua_State *L) {
    lua_pushnumber(L, 0);
    lua_pushcclosure(L, &counter, 1);
    return 1;
}
```

这里的关键函数是 lua_pushcclosure，她的第二个参数是一个基本函数（例子中卫 counter），第三个参数是 upvalues 的个数（例子中为 1）。在创建新的闭包之前，我们必须将 upvalues 的初始值入栈，在我们的例子中，我们将数字 0 作为唯一的 upvalue 的初始值入栈。如预期的一样，lua_pushcclosure 将新的闭包放到栈内，因此闭包已经作为 newCounter 的结果被返回。

现在，我们看看 counter 的定义：

```
static int counter (lua_State *L) {
    double val = lua_tonumber(L, lua_upvalueindex(1));
    lua_pushnumber(L, ++val);    /* new value */
    lua_pushvalue(L, -1);        /* duplicate it */
    lua_replace(L, lua_upvalueindex(1));  /* update upvalue */
    return 1;  /* return new value */
}
```

这里的关键函数是 lua_upvalueindex（实际是一个宏），用来产生一个 upvalue 的假索引。这个假索引除了不在栈中之外，和其他的索引一样。表达式 lua_upvalueindex(1) 函数第一个 upvalue 的索引。因此，在函数 counter 中的 lua_tonumber 获取第一个(仅有的)upvalue 的当前值，转换为数字型。然后，函数 counter 将新的值++val 入栈，并将这个值的一个拷贝使用新的值替换 upvalue。最后，返回其他的拷贝。

与 Lua 闭包不同的是，C 闭包不能共享 upvalues：每一个闭包都有自己独立的变量集。然而，我们可以设置不同函数的 upvalues 指向同一个表，这样这个表就变成了一个所有函数共享数据的地方。

第 28 章 User-Defined Types in C

在面的一章，我们讨论了如何使用 C 函数扩展 Lua 的功能，现在我们讨论如何使用 C 中新创建的类型来扩展 Lua。我们从一个小例子开始，本章后续部分将以这个小例子为基础逐步加入 metamethods 等其他内容来介绍如何使用 C 中新类型扩展 Lua。

我们的例子涉及的类型非常简单，数字数组。这个例子的目的在于将目光集中到 API 问题上，所以不涉及复杂的算法。尽管例子中的类型很简单，但很多应用中都会用到这种类型。一般情况下，Lua 中并不需要外部的数组，因为哈希表很好的实现了数组。但是对于非常大的数组而言，哈希表可能导致内存不足，因为对于每一个元素必须保存一个范性的（generic）值，一个链接地址，加上一些以备将来增长的额外空间。在 C 中的直接存储数字值不需要额外的空间，将比哈希表的实现方式节省 50%的内存空间。

我们使用下面的结构表示我们的数组：

```
typedef struct NumArray {
    int size;
    double values[1];  /* variable part */
} NumArray;
```

我们使用大小 1 声明数组的 values，由于 C 语言不允许大小为 0 的数组，这个 1 只是一个占位符；我们在后面定义数组分配空间的实际大小。对于一个有 n 个元素的数组来说，我们需要

```
sizeof(NumArray) + (n-1)*sizeof(double) bytes
```

（由于原始的结构中已经包含了一个元素的空间，所以我们从 n 中减去 1）

28.1 Userdata

我们首先关心的是如何在 Lua 中表示数组的值。Lua 为这种情况提供专门提供一个基本的类型：userdata。一个 userdatum 提供了一个在 Lua 中没有预定义操作的 raw 内存区域。

Lua API 提供了下面的函数用来创建一个 userdatum：

```
void *lua_newuserdata (lua_State *L, size_t size);
```

lua_newuserdata 函数按照指定的大小分配一块内存，将对应的 userdatum 放到栈内，并返回内存块的地址。如果出于某些原因你需要通过其他的方法分配内存的话，很容易创建一个指针大小的 userdatum，然后将指向实际内存块的指针保存到 userdatum 里。我们将在下一章看到这种技术的例子。

使用 lua_newuserdata 函数，创建新数组的函数实现如下：

```
static int newarray (lua_State *L) {
    int n = luaL_checkint(L, 1);
    size_t nbytes = sizeof(NumArray) + (n - 1)*sizeof(double);
    NumArray *a = (NumArray *)lua_newuserdata(L, nbytes);
    a->size = n;
    return 1;  /* new userdatum is already on the stack */
}
```

（函数 luaL_checkint 是用来检查整数的 luaL_checknumber 的变体）一旦 newarray 在 Lua 中被注册之后，你就可以使用类似 a = array.new(1000)的语句创建一个新的数组了。

为了存储元素，我们使用类似 array.set(array, index, value)调用，后面我们将看到如何使用 metatables 来支持常规的写法 array[index] = value。对于这两种写法，下面的函数是一样的，数组下标从 1 开始：

```
static int setarray (lua_State *L) {
    NumArray *a = (NumArray *)lua_touserdata(L, 1);
    int index = luaL_checkint(L, 2);
    double value = luaL_checknumber(L, 3);

    luaL_argcheck(L, a != NULL, 1, "`array' expected");

    luaL_argcheck(L, 1 <= index && index <= a->size, 2,
                  "index out of range");

    a->values[index-1] = value;
    return 0;
}
```

luaL_argcheck 函数检查给定的条件，如果有必要的话抛出错误。因此，如果我们使用错误的参数调用 setarray，我们将得到一个错误信息：

```
array.set(a, 11, 0)
--> stdin:1: bad argument #1 to 'set' ('array' expected)
```

下面的函数获取一个数组元素：

```
static int getarray (lua_State *L) {
    NumArray *a = (NumArray *)lua_touserdata(L, 1);
    int index = luaL_checkint(L, 2);
```

```
    luaL_argcheck(L, a != NULL, 1, "'array' expected");

    luaL_argcheck(L, 1 <= index && index <= a->size, 2,
                          "index out of range");

    lua_pushnumber(L, a->values[index-1]);
    return 1;
}
```

我们定义另一个函数来获取数组的大小：

```
static int getsize (lua_State *L) {
    NumArray *a = (NumArray *)lua_touserdata(L, 1);
    luaL_argcheck(L, a != NULL, 1, "`array' expected");
    lua_pushnumber(L, a->size);
    return 1;
}
```

最后，我们需要一些额外的代码来初始化我们的库：

```
static const struct luaL_reg arraylib [] = {
    {"new", newarray},
    {"set", setarray},
    {"get", getarray},
    {"size", getsize},
    {NULL, NULL}
};

int luaopen_array (lua_State *L) {
    luaL_openlib(L, "array", arraylib, 0);
    return 1;
}
```

这儿我们再次使用了辅助库的 luaL_openlib 函数，他根据给定的名字创建一个表，并使用 arraylib 数组中的 name-function 对填充这个表。

打开上面定义的库之后，我们就可以在 Lua 中使用我们新定义的类型了：

```
a = array.new(1000)
print(a)                --> userdata: 0x8064d48
print(array.size(a))    --> 1000
for i=1,1000 do
    array.set(a, i, 1/i)
end
```

```
print(array.get(a, 10))  --> 0.1
```

在一个 Pentium/Linux 环境中运行这个程序，一个有 100K 元素的数组大概占用 800KB 的内存，同样的条件由 Lua 表实现的数组需要 1.5MB 的内存。

28.2 Metatables

我们上面的实现有一个很大的安全漏洞。假如使用者写了如下类似的代码：array.set(io.stdin, 1, 0)。io.stdin 中的值是一个带有指向流(FILE*)的指针的 userdatum。因为它是一个 userdatum，所以 array.set 很乐意接受它作为参数，程序运行的结果可能导致内存 core dump（如果你够幸运的话，你可能得到一个访问越界（index-out-of-range）错误）。这样的错误对于任何一个 Lua 库来说都是不能忍受的。不论你如何使用一个 C 库，都不应该破坏 C 数据或者从 Lua 产生 core dump。

为了区分数组和其他的 userdata，我们单独为数组创建了一个 metatable（记住 userdata 也可以拥有 metatables）。下面，我们每次创建一个新的数组的时候，我们将这个单独的 metatable 标记为数组的 metatable。每次我们访问数组的时候，我们都要检查他是否有一个正确的 metatable。因为 Lua 代码不能改变 userdatum 的 metatable，所以他不会伪造我们的代码。

我们还需要一个地方来保存这个新的 metatable，这样我们才能够当创建新数组和检查一个给定的 userdatum 是否是一个数组的时候，可以访问这个 metatable。正如我们前面介绍过的，有两种方法可以保存 metatable：在 registry 中，或者在库中作为函数的 upvalue。在 Lua 中一般习惯于在 registry 中注册新的 C 类型，使用类型名作为索引，metatable 作为值。和其他的 registry 中的索引一样，我们必须选择一个唯一的类型名，避免冲突。我们将这个新的类型称为 "LuaBook.array"。

辅助库提供了一些函数来帮助我们解决问题，我们这儿将用到的前面未提到的辅助函数有：

```
int   luaL_newmetatable (lua_State *L, const char *tname);
void  luaL_getmetatable (lua_State *L, const char *tname);
void *luaL_checkudata (lua_State *L, int index,
                                     const char *tname);
```

luaL_newmetatable 函数创建一个新表（将用作 metatable），将新表放到栈顶并建立表和 registry 中类型名的联系。这个关联是双向的：使用类型名作为表的 key；同时使用表作为类型名的 key（这种双向的关联，使得其他的两个函数的实现效率更高）。luaL_getmetatable 函数获取 registry 中的 tname 对应的 metatable。最后，luaL_checkudata 检查在栈中指定位置的对象是否为带有给定名字的 metatable 的 usertatum。如果对象不存在正确的 metatable，返回 NULL（或者它不是一个 userdata）；否则，返回 userdata 的地址。

下面来看具体的实现。第一步修改打开库的函数，新版本必须创建一个用作数组

metatable 的表：

```
int luaopen_array (lua_State *L) {
    luaL_newmetatable(L, "LuaBook.array");
    luaL_openlib(L, "array", arraylib, 0);
    return 1;
}
```

第二步，修改 newarray，使得在创建数组的时候设置数组的 metatable：

```
static int newarray (lua_State *L) {
    int n = luaL_checkint(L, 1);
    size_t nbytes = sizeof(NumArray) + (n - 1)*sizeof(double);
    NumArray *a = (NumArray *)lua_newuserdata(L, nbytes);

    luaL_getmetatable(L, "LuaBook.array");
    lua_setmetatable(L, -2);

    a->size = n;
    return 1;  /* new userdatum is already on the stack */
}
```

lua_setmetatable 函数将表出栈，并将其设置为给定位置的对象的 metatable。在我们的例子中，这个对象就是新的 userdatum。

最后一步，setarray、getarray 和 getsize 检查他们的第一个参数是否是一个有效的数组。因为我们打算在参数错误的情况下抛出一个错误信息，我们定义了下面的辅助函数：

```
static NumArray *checkarray (lua_State *L) {
    void *ud = luaL_checkudata(L, 1, "LuaBook.array");
    luaL_argcheck(L, ud != NULL, 1, "`array' expected");
    return (NumArray *)ud;
}
```

使用 checkarray，新定义的 getsize 是更直观、更清楚：

```
static int getsize (lua_State *L) {
    NumArray *a = checkarray(L);
    lua_pushnumber(L, a->size);
    return 1;
}
```

由于 setarray 和 getarray 检查第二个参数 index 的代码相同，我们抽象出他们的共同部分，在一个单独的函数中完成：

```
static double *getelem (lua_State *L) {
```

```
    NumArray *a = checkarray(L);
    int index = luaL_checkint(L, 2);

    luaL_argcheck(L, 1 <= index && index <= a->size, 2,
                  "index out of range");

    /* return element address */
    return &a->values[index - 1];
}
```

使用这个 getelem，函数 setarray 和 getarray 更加直观易懂：

```
static int setarray (lua_State *L) {
    double newvalue = luaL_checknumber(L, 3);
    *getelem(L) = newvalue;
    return 0;
}

static int getarray (lua_State *L) {
    lua_pushnumber(L, *getelem(L));
    return 1;
}
```

现在，假如你尝试类似 array.get(io.stdin, 10)的代码，你将会得到正确的错误信息：

```
error: bad argument #1 to 'getarray' ('array' expected)
```

28.3 访问面向对象的数据

下面我们来看看如何定义类型为对象的 userdata，以致我们就可以使用面向对象的语法来操作对象的实例，比如：

```
a = array.new(1000)
print(a:size())       --> 1000
a:set(10, 3.4)
print(a:get(10))      --> 3.4
```

记住 a:size()等价于 a.size(a)。所以，我们必须使得表达式 a.size 调用我们的 getsize 函数。这儿的关键在于__index 元方法（metamethod）的使用。对于表来说，不管什么时候只要找不到给定的 key，这个元方法就会被调用。对于 userdata 来讲，每次被访问的时候元方法都会被调用，因为 userdata 根本就没有任何 key。

假如我们运行下面的代码：

```
local metaarray = getmetatable(array.new(1))
metaarray.__index = metaarray
metaarray.set = array.set
metaarray.get = array.get
metaarray.size = array.size
```

第一行，我们仅仅创建一个数组并获取他的 metatable，metatable 被赋值给 metaarray（我们不能从 Lua 中设置 userdata 的 metatable，但是我们在 Lua 中无限制的访问 metatable）。接下来，我们设置 metaarray.__index 为 metaarray。当我们计算 a.size 的时候，Lua 在对象 a 中找不到 size 这个键值，因为对象是一个 userdatum。所以，Lua 试着从对象 a 的 metatable 的__index 域获取这个值，正好__index 就是 metaarray。但是 metaarray.size 就是 array.size,因此 a.size(a)如我们预期的返回 array.size(a)。

当然，我们可以在 C 中完成同样的事情，甚至可以做得更好：现在数组是对象，他有自己的操作，我们在表数组中不需要这些操作。我们实现的库唯一需要对外提供的函数就是 new，用来创建一个新的数组。所有其他的操作作为方法实现。C 代码可以直接注册他们。

getsize、getarray 和 setarray 与我们前面的实现一样，不需要改变。我们需要改变的只是如何注册他们。也就是说，我们必须改变打开库的函数。首先，我们需要分离函数列表，一个作为普通函数，一个作为方法：

```
static const struct luaL_reg arraylib_f [] = {
    {"new", newarray},
    {NULL, NULL}
};

static const struct luaL_reg arraylib_m [] = {
    {"set", setarray},
    {"get", getarray},
    {"size", getsize},
    {NULL, NULL}
};
```

新版本打开库的函数 luaopen_array，必须创建一个 metatable，并将其赋值给自己的__index 域，在那儿注册所有的方法，创建并填充数组表：

```
int luaopen_array (lua_State *L) {
    luaL_newmetatable(L, "LuaBook.array");

    lua_pushstring(L, "__index");
    lua_pushvalue(L, -2);    /* pushes the metatable */
    lua_settable(L, -3); /* metatable.__index = metatable */
```

```
    luaL_openlib(L, NULL, arraylib_m, 0);

    luaL_openlib(L, "array", arraylib_f, 0);
    return 1;
}
```

这里我们使用了 luaL_openlib 的另一个特征，第一次调用，当我们传递一个 NULL 作为库名时，luaL_openlib 并没有创建任何包含函数的表；相反，他认为封装函数的表在栈内，位于临时的 upvalues 的下面。在这个例子中，封装函数的表是 metatable 本身，也就是 luaL_openlib 放置方法的地方。第二次调用 luaL_openlib 正常工作：根据给定的数组名创建一个新表，并在表中注册指定的函数（例子中只有一个函数 new）。

下面的代码，我们为我们的新类型添加一个__tostring 方法，这样一来 print(a)将打印数组加上数组的大小，大小两边带有圆括号（比如，array(1000)）：

```
int array2string (lua_State *L) {
    NumArray *a = checkarray(L);
    lua_pushfstring(L, "array(%d)", a->size);
    return 1;
}
```

函数 lua_pushfstring 格式化字符串，并将其放到栈顶。为了在数组对象的 metatable 中包含 array2string，我们还必须在 arraylib_m 列表中添加 array2string：

```
static const struct luaL_reg arraylib_m [] = {
    {"__tostring", array2string},
    {"set", setarray},
    ...
};
```

28.4 访问数组

除了上面介绍的使用面向对象的写法来访问数组以外，还可以使用传统的写法来访问数组元素，不是 a:get(i)，而是 a[i]。对于我们上面的例子，很容易实现这个，因为我们的 setarray 和 getarray 函数已经依次接受了与他们的元方法对应的参数。一个快速的解决方法是在我们的 Lua 代码中正确的定义这些元方法：

```
local metaarray = getmetatable(newarray(1))
metaarray.__index = array.get
metaarray.__newindex = array.set
```

（这段代码必须运行在前面的最初的数组实现基础上，不能使用为了面向对象访问

的修改的那段代码）

我们要做的只是使用传统的语法：

```
a = array.new(1000)
a[10] = 3.4       -- setarray
print(a[10])      -- getarray   --> 3.4
```

如果我们喜欢的话，我们可以在我们的 C 代码中注册这些元方法。我们只需要修改我们的初始化函数：

```
int luaopen_array (lua_State *L) {
    luaL_newmetatable(L, "LuaBook.array");
    luaL_openlib(L, "array", arraylib, 0);

    /* now the stack has the metatable at index 1 and
        'array' at index 2 */
    lua_pushstring(L, "__index");
    lua_pushstring(L, "get");
    lua_gettable(L, 2); /* get array.get */
    lua_settable(L, 1); /* metatable.__index = array.get */

    lua_pushstring(L, "__newindex");
    lua_pushstring(L, "set");
    lua_gettable(L, 2); /* get array.set */
    lua_settable(L, 1); /* metatable.__newindex = array.set */

    return 0;
}
```

28.5 Light Userdata

到目前为止我们使用的 userdata 称为 full userdata。Lua 还提供了另一种 userdata: light userdata。

一个 light userdatum 是一个表示 C 指针的值（也就是一个 void *类型的值）。由于它是一个值，我们不能创建他们（同样的，我们也不能创建一个数字）。可以使用函数 lua_pushlightuserdata 将一个 light userdatum 入栈：

```
void lua_pushlightuserdata (lua_State *L, void *p);
```

尽管都是 userdata，light userdata 和 full userdata 有很大不同。Light userdata 不是一个缓冲区，仅仅是一个指针，没有 metatables。像数字一样，light userdata 不需要垃圾收

集器来管理她。

有些人把 light userdata 作为一个低代价的替代实现，来代替 full userdata，但是这不是 light userdata 的典型应用。首先，使用 light userdata 你必须自己管理内存，因为他们和垃圾收集器无关。第二，尽管从名字上看有轻重之分，但 full userdata 实现的代价也并不大，比较而言，他只是在分配给定大小的内存时候，有一点点额外的代价。

Light userdata 真正的用处在于可以表示不同类型的对象。当 full userdata 是一个对象的时候，它等于对象自身；另一方面，light userdata 表示的是一个指向对象的指针，同样的，它等于指针指向的任何类型的 userdata。所以，我们在 Lua 中使用 light userdata 表示 C 对象。

看一个典型的例子，假定我们要实现：Lua 和窗口系统的绑定。这种情况下，我们使用 full userdata 表示窗口（每一个 userdatum 可以包含整个窗口结构或者一个有系统创建的指向单个窗口的指针）。当在窗口有一个事件发生（比如按下鼠标），系统会根据窗口的地址调用专门的回调函数。为了将这个回调函数传递给 Lua，我们必须找到表示指定窗口的 userdata。为了找到这个 userdata,我们可以使用一个表：索引为表示窗口地址的 light userdata，值为在 Lua 中表示窗口的 full userdata。一旦我们有了窗口的地址，我们将窗口地址作为 light userdata 放到栈内，并且将 userdata 作为表的索引存到表内。（注意这个表应该有一个 weak 值，否则，这些 full userdata 永远不会被回收掉。）

第 29 章 资源管理

在前面一章介绍的数组实现方法，我们不必担心如何管理资源，只需要分配内存。每一个表示数组的 userdatum 都有自己的内存，这个内存由 Lua 管理。当数组变为垃圾（也就是说，当程序不需要）的时候，Lua 会自动收集并释放内存。

生活总是不那么如意。有时候，一个对象除了需要物理内存以外，还需要文件描述符、窗口句柄等类似的资源。（通常这些资源也是内存，但由系统的其他部分来管理）。在这种情况下，当一个对象成为垃圾并被收集的时候，这些相关的资源也应该被释放。一些面向对象的语言为了这种需要提供了一种特殊的机制（称为 finalizer 或者析构器）。Lua 以__gc 元方法的方式提供了 finalizers。这个元方法只对 userdata 类型的值有效。当一个 userdatum 将被收集的时候，并且 usedatum 有一个__gc 域，Lua 会调用这个域的值（应该是一个函数）：以 userdatum 作为这个函数的参数调用。这个函数负责释放与 userdatum 相关的所有资源。

为了阐明如何将这个元方法和 API 作为一个整体使用，这一章我们将使用 Lua 扩展应用的方式，介绍两个例子。第一个例子是前面已经介绍的遍历一个目录的函数的另一种实现。第二个例子是一个绑定 Expat（Expat 开源的 XML 解析器）实现的 XML 解析器。

29.1 目录迭代器

前面我们实现了一个 dir 函数，给定一个目录作为参数，这个函数以一个 table 的方式返回目录下所有文件。我们新版本的 dir 函数将返回一个迭代子，每次调用这个迭代子的时候会返回目录中的一个入口（entry）。按新版本的实现方式，我们可以使用循环来遍历整个目录：

```
for fname in dir(".") do  print(fname)  end
```

在 C 语言中，我们需要 DIR 这种结构才能够迭代一个目录。通过 opendir 才能创建一个 DIR 的实例，并且必须显式的调用 closedir 来释放资源。我们以前实现的 dir 用一个本地变量保存 DIR 的实例，并且在获取目录中最后一个文件名之后关闭实例。但我们新实现的 dir 中不能在本地变量中保存 DIR 的实例，因为有很多个调用都要访问这个值，另外，也不能仅仅在获取目录中最后一个文件名之后关闭目录。如果程序循环过程中中断退出，迭代子根本就不会取得最后一个文件名，所以，为了保证 DIR 的实例一定能够被释放掉，我们将它的地址保存在一个 userdatum 中，并使用这个 userdatum 的__gc 的元方法来释放目录结构。

尽管我们实现中userdatum的作用很重要，但这个用来表示一个目录的userdatum，并

不需要在Lua可见范围之内。Dir函数返回一个迭代子函数，迭代子函数需要在Lua的可见范围之内。目录可能是迭代子函数的一个upvalue。这样一来，迭代子函数就可以直接访问这个结构[9]，但是Lua不可以（也不需要）访问这个结构。

总的来说，我们需要三个 C 函数。第一，dir 函数，一个 Lua 调用他产生迭代器的工厂，这个函数必须打开 DIR 结构并将他作为迭代函数的 upvalue。第二，我们需要一个迭代函数。第三，__gc 元方法，负责关闭 DIR 结构。一般来说，我们还需要一个额外的函数来进行一些初始的操作，比如为目录创建 metatable，并初始化这个 metatable。

首先看我们的 dir 函数：

```
#include <dirent.h>
#include <errno.h>

/* forward declaration for the iterator function */
static int dir_iter (lua_State *L);

static int l_dir (lua_State *L) {
    const char *path = luaL_checkstring(L, 1);

    /* create a userdatum to store a DIR address */
    DIR **d = (DIR **)lua_newuserdata(L, sizeof(DIR *));

    /* set its metatable */
    luaL_getmetatable(L, "LuaBook.dir");
    lua_setmetatable(L, -2);

    /* try to open the given directory */
    *d = opendir(path);
    if (*d == NULL)  /* error opening the directory? */
        luaL_error(L, "cannot open %s: %s", path,
                   strerror(errno));

    /* creates and returns the iterator function
       (its sole upvalue, the directory userdatum,
       is already on the stack top */
    lua_pushcclosure(L, dir_iter, 1);
    return 1;
}
```

[9] 译注：指目录结构，即 userdatum

这儿有一点需要注意的，我们必须在打开目录之前创建 userdatum。如果我们先打开目录，然后调用 lua_newuserdata 会抛出错误，这样我们就无法获取 DIR 结构。按照正确的顺序，DIR 结构一旦被创建，就会立刻和 userdatum 关联起来；之后不管发生什么，__gc 元方法都会自动的释放这个结构。

第二个函数是迭代器：

```
static int dir_iter (lua_State *L) {
    DIR *d = *(DIR **)lua_touserdata(L, lua_upvalueindex(1));
    struct dirent *entry;
    if ((entry = readdir(d)) != NULL) {
        lua_pushstring(L, entry->d_name);
        return 1;
    }
    else return 0;  /* no more values to return */
}
```

__gc 元方法用来关闭目录，但有一点需要小心：因为我们在打开目录之前创建 userdatum，所以不管 opendir 的结果是什么，userdatum 将来都会被收集。如果 opendir 失败，将来就没有什么可以关闭的了：

```
static int dir_gc (lua_State *L) {
    DIR *d = *(DIR **)lua_touserdata(L, 1);
    if (d) closedir(d);
    return 0;
}
```

最后一个函数打开这个只有一个函数的库：

```
int luaopen_dir (lua_State *L) {
    luaL_newmetatable(L, "LuaBook.dir");

    /* set its __gc field */
    lua_pushstring(L, "__gc");
    lua_pushcfunction(L, dir_gc);
    lua_settable(L, -3);

    /* register the `dir' function */
    lua_pushcfunction(L, l_dir);
    lua_setglobal(L, "dir");

    return 0;
}
```

整个例子有一个注意点。开始的时候，dir_gc 看起来应该检查他的参数是否是一个目录。否则，一个恶意的使用者可能用其他类型的参数（比如，文件）调用这个函数导致严重的后果。然而，在 Lua 程序中无法访问这个函数：他被存放在目录的 metatable 中，Lua 程序从来不会访问这些目录。

29.2 XML 解析

现在，我们将要看到一个xml解析器的简单实现，称为lxp[10]，它包括了Lua和Expat（http://www.libexpat.org/）。Expat是一个开源的C语言写成的XML 1.0 的解析器。它实现了SAX（http://www.saxproject.org/），SAX是XML简单的API，是基于事件的API，这意味着一个SAX解析器读取有一个XML文档，然后反馈给应用程序他所发现的。举个例子，我们要通知Expat解析这样一个字符串：

```
<tag cap="5">hi</tag>
```

它将会产生三个事件：当它读取子字符串 "<tag cap="5">hi</tag>"，产生一个读取到开始元素的事件；当它解析 "hi" 时，产生一个读取文本事件（有时也称为字符数据事件）；当解析 "end" 时，产生一个读取结束元素的事件。而每个事件，都会调用应用程序适当的句柄。

这里我们不会涉及到整个 Expat 库，我们只会集中精力关注那些能够阐明和 Lua 相互作用的新技术的部分。当我们实现了核心功能后，在上面进行扩展将会变得很容易。虽然 Expat 解析 XML 文档时会有很多事件，我们将会关心的仅仅是上面例子提到的三个事件（开始元素，结束元素，文本数据），我们需要调用的 API 是 Expat 众多 API 中很少的几个。首先，我们需要创建和析构 Expat 解析器的函数：

```
#include <xmlparse.h>

XML_Parser XML_ParserCreate (const char *encoding);
void XML_ParserFree (XML_Parser p);
```

这里函数参数是可选的；在我们的使用中，我们直接选用 NULL 作为参数。当我们有了一个解析器的时候，我们必须注册回调的句柄：

```
XML_SetElementHandler(XML_Parser p,
                      XML_StartElementHandler start,
                      XML_EndElementHandler end);

XML_SetCharacterDataHandler(XML_Parser p,
                            XML_CharacterDataHandler hndl);
```

[10] 译注：估计是 lua xml parser 的简写。

第一个函数登记了开始元素和结束元素的句柄。第二个函数登记了文本数据（在XML 语法中的字符数据）的句柄。所有回掉的句柄通过第一个参数接收用户数据。开始元素的句柄同样接收到标签的名称和它的属性作为参数：

```
typedef void (*XML_StartElementHandler)(void *uData,
                                        const char *name,
                                        const char **atts);
```

这些属性来自于以 '\0' 结束的字符串组成的数组，这些字符串分别对应了一对以属性名和属性值组成的属性。结束元素的句柄只有一个参数，就是标签名。

```
typedef void (*XML_EndElementHandler)(void *uData,
                                      const char *name)
```

最终，一个文本句柄仅仅以字符串作为额外的参数。该文本字符串不能是以'\0'结束的字符串，而是显式指明长度的字符串：

```
typedef void
(*XML_CharacterDataHandler)(void *uData,
                            const char *s,
                            int len);
```

我们用下面的函数将这些文本传给 Expat：

```
int XML_Parse (XML_Parser p,
               const char *s, int len, int isFinal);
```

Expat 通过成功调用 XML_Parse 一段一段的解析它接收到的文本。XML_Parse 最后一个参数为 isFinal，他表示这部分是不是 XML 文档的最后一个部分了。需要注意的是不是每段文本都需要通过 0 来表示结束，我们也可以通过显实的长度来判定。XML_Parse 函数如果发现解析错误就会返回一个 0（expat 也提供了辅助的函数来显示错误信息，但是因为简单的缘故，我们这里都将之忽略掉）。我们需要 Expat 的最后一个函数是允许我们设置将要传给句柄的用户数据的函数：

```
void XML_SetUserData (XML_Parser p, void *uData);
```

好了，现在我们来看一下如何在 Lua 中使用 Expat 库。第一种方法也是最直接的一种方法：简单的在 Lua 中导入这些函数。比较好的方法是对 Lua 调整这些函数。比如 Lua 是没有类型的，我们不需要用不同的函数来设置不同的调用。但是我们怎么样避免一起调用那些注册了的函数呢。替代的是，当我们创建了一个解析器，我们同时给出了一个包含所有回调句柄以及相应的键值的回调表。举个例子来说，如果我们要打印一个文档的布局，我们可以用下面的回调表：

```
local count = 0

callbacks = {
```

```
    StartElement = function (parser, tagname)
        io.write("+ ", string.rep("  ", count), tagname, "\n")
        count = count + 1
    end,

    EndElement = function (parser, tagname)
        count = count - 1
        io.write("- ", string.rep("  ", count), tagname, "\n")
    end,
}
```

输入"<to> <yes/> </to>"，这些句柄将会打印出：

```
+ to
+   yes
-   yes
- to
```

通过这个 API，我们不需要维护这些函数的调用。我们直接在回调表中维回他们。因此，整个 API 需要三个函数：一个创建解析器，一个解析一段段文本，最后一个关闭解析器。（实际上，我们用解析器对象的方法，实现了最后两个功能）。对这些 API 函数的典型使用如下：

```
p = lxp.new(callbacks)       -- create new parser
for l in io.lines() do       -- iterate over input lines
    assert(p:parse(l))       -- parse the line
    assert(p:parse("\n"))    -- add a newline
end
assert(p:parse())            -- finish document
p:close()
```

现在，让我们把注意力集中到实现中来。首先，考虑如何在 Lua 中实现解析器。很自然的会想到使用 userdatum，但是我们将什么内容放在 userdata 里呢？至少，我们必须保留实际的 Expat 解析器和一个回调表。我们不能将一个 Lua 表保存在一个 userdatum（或者在任何的 C 结构中），然而，我们可以创建一个指向表的引用，并将这个引用保存在 userdatum 中。（我们在 27.3.2 节已经说过，一个引用就是 Lua 自动产生的在 registry 中的一个整数）最后，我们还必须能够将 Lua 的状态保存到一个解析器对象中，因为这些解析器对象就是 Expat 回调从我们程序中接受的所有内容，并且这些回调需要调用 Lua。一个解析器的对象的定义如下：

```
#include <xmlparse.h>

typedef struct lxp_userdata {
```

```
    lua_State *L;
    XML_Parser *parser;  /* associated expat parser */
    int tableref; /* table with callbacks for this parser */
} lxp_userdata;
```

下面是创建解析器对象的函数：

```
static int lxp_make_parser (lua_State *L) {
    XML_Parser p;
    lxp_userdata *xpu;

    /* (1) create a parser object */
    xpu = (lxp_userdata *)lua_newuserdata(L,
                  sizeof(lxp_userdata));

    /* pre-initialize it, in case of errors */
    xpu->tableref = LUA_REFNIL;
    xpu->parser = NULL;

    /* set its metatable */
    luaL_getmetatable(L, "Expat");
    lua_setmetatable(L, -2);

    /* (2) create the Expat parser */
    p = xpu->parser = XML_ParserCreate(NULL);
    if (!p)
        luaL_error(L, "XML_ParserCreate failed");

    /* (3) create and store reference to callback table */
    luaL_checktype(L, 1, LUA_TTABLE);
    lua_pushvalue(L, 1);  /* put table on the stack top */
    xpu->tableref = luaL_ref(L, LUA_REGISTRYINDEX);

    /* (4) configure Expat parser */
    XML_SetUserData(p, xpu);
    XML_SetElementHandler(p, f_StartElement, f_EndElement);
    XML_SetCharacterDataHandler(p, f_CharData);
    return 1;
}
```

函数 lxp_make_parser 有四个主要步骤：

第一步遵循共同的模式：首先创建一个 userdatum，然后使用 consistent 的值预初始化 userdatum，最后设置 userdatum 的 metatable。预初始化的原因在于：如果在初始化的时候有任何错误的话，我们必须保证析构器（__gc 元方法）能够发现在可靠状态下发现 userdata 并释放资源。

第二步，函数创建一个 Expat 解析器，将它保存在 userdatum 中，并检测错误。

第三步，保证函数的第一个参数是一个表（回调表），创建一个指向表的引用，并将这个引用保存到新的 userdatum 中。

第四步，初始化 Expat 解析器，将 userdatum 设置为将要传递给回调函数的对象，并设置这些回调函数。注意，对于所有的解析器来说这些回调函数都一样。毕竟，在 C 中不可能动态的创建新的函数，取代的方法是，这些固定的 C 函数使用回调表来决定每次应该调用哪个 Lua 函数。

下一步是解析方法，负责解析一段 XML 数据。他有两个参数：解析器对象(方法自己)和一个可选的一段 XML 数据。当没有数据调用这个方法时，他通知 Expat 文档已经解析结束：

```
static int lxp_parse (lua_State *L) {
    int status;
    size_t len;
    const char *s;
    lxp_userdata *xpu;

    /* get and check first argument (should be a parser) */
    xpu = (lxp_userdata *)luaL_checkudata(L, 1, "Expat");
    luaL_argcheck(L, xpu, 1, "expat parser expected");

    /* get second argument (a string) */
    s = luaL_optlstring(L, 2, NULL, &len);

    /* prepare environment for handlers: */
    /* put callback table at stack index 3 */
    lua_settop(L, 2);
    lua_getref(L, xpu->tableref);
    xpu->L = L;  /* set Lua state */

    /* call Expat to parse string */
    status = XML_Parse(xpu->parser, s, (int)len, s == NULL);

    /* return error code */
    lua_pushboolean(L, status);
```

```
    return 1;
}
```

当 lxp_parse 调用 XML_Parse 的时候，后一个函数将会对在给定的一段 XML 数据中找到的所有元素，分别调用这些元素对应的句柄。所以，lxp_parse 会首先为这些句柄准备环境，在调用 XML_Parse 的时候有一些细节：记住这个函数的最后一个参数告诉 Expat 给定的文本段是否是最后一段。当我们不带参数调用他时，s 将使用缺省的 NULL，因此这时候最后一个参数将为 true。现在让我们注意力集中到回调函数 f_StartElement、f_EndElement 和 f_CharData 上，这三个函数有相似的结构：每一个都会针对他的指定事件检查 callback 表是否定义了 Lua 句柄，如果有，预处理参数然后调用这个 Lua 句柄。

我们首先来看 f_CharData 句柄，他的代码非常简单。她调用他对应的 Lua 中的句柄（当存在的时候），带有两个参数：解析器 parser 和字符数据（一个字符串）

```
static void f_CharData (void *ud, const char *s, int len) {
    lxp_userdata *xpu = (lxp_userdata *)ud;
    lua_State *L = xpu->L;

    /* get handler */
    lua_pushstring(L, "CharacterData");
    lua_gettable(L, 3);
    if (lua_isnil(L, -1)) {  /* no handler? */
        lua_pop(L, 1);
        return;
    }

    lua_pushvalue(L, 1); /* push the parser (`self') */
    lua_pushlstring(L, s, len);  /* push Char data */
    lua_call(L, 2, 0);   /* call the handler */
}
```

注意，由于当我们创建解析器的时候调用了 XML_SetUserData，所以，所有的 C 句柄都接受 lxp_userdata 数据结构作为第一个参数。还要注意程序是如何使用由 lxp_parse 设置的环境的。首先，他假定 callback 表在栈中的索引为 3；第二，假定解析器 parser 在栈中索引为 1（parser 的位置肯定是这样的，因为她应该是 lxp_parse 的第一个参数）。

f_EndElement 句柄和 f_CharData 类似，也很简单。他也是用两个参数调用相应的 Lua 句柄：一个解析器 parser 和一个标签名（也是一个字符串，但现在是以 '\0' 结尾）：

```
static void f_EndElement (void *ud, const char *name) {
    lxp_userdata *xpu = (lxp_userdata *)ud;
    lua_State *L = xpu->L;
```

```
    lua_pushstring(L, "EndElement");
    lua_gettable(L, 3);
    if (lua_isnil(L, -1)) {  /* no handler? */
        lua_pop(L, 1);
        return;
    }

    lua_pushvalue(L, 1); /* push the parser (`self') */
    lua_pushstring(L, name);  /* push tag name */
    lua_call(L, 2, 0);   /* call the handler */
}
```

最后一个句柄 f_StartElement 带有三个参数：解析器 parser，标签名，和一个属性列表。这个句柄比上面两个稍微复杂点，因为它需要将属性的标签列表翻译成 Lua 识别的内容。我们是用自然的翻译方式，比如，类似下面的开始标签：

```
<to method="post" priority="high">
```

产生下面的属性表：

```
{ method = "post", priority = "high" }
```

f_StartElement 的实现如下：

```
static void f_StartElement (void *ud,
                            const char *name,
                            const char **atts) {
    lxp_userdata *xpu = (lxp_userdata *)ud;
    lua_State *L = xpu->L;

    lua_pushstring(L, "StartElement");
    lua_gettable(L, 3);
    if (lua_isnil(L, -1)) {  /* no handler? */
        lua_pop(L, 1);
        return;
    }

    lua_pushvalue(L, 1);  /* push the parser (`self') */
    lua_pushstring(L, name);  /* push tag name */

    /* create and fill the attribute table */
    lua_newtable(L);
    while (*atts) {
```

```
        lua_pushstring(L, *atts++);
        lua_pushstring(L, *atts++);
        lua_settable(L, -3);
    }

    lua_call(L, 3, 0);  /* call the handler */
}
```

解析器的最后一个方法是 close。当我们关闭一个解析器的时候，我们必须释放解析器对应的所有资源，即 Expat 结构和 callback 表。记住，在解析器创建的过程中如果发生错误，解析器并不拥有这些资源：

```
static int lxp_close (lua_State *L) {
    lxp_userdata *xpu;

    xpu = (lxp_userdata *)luaL_checkudata(L, 1, "Expat");
    luaL_argcheck(L, xpu, 1, "expat parser expected");

    /* free (unref) callback table */
    luaL_unref(L, LUA_REGISTRYINDEX, xpu->tableref);
    xpu->tableref = LUA_REFNIL;

    /* free Expat parser (if there is one) */
    if (xpu->parser)
        XML_ParserFree(xpu->parser);
    xpu->parser = NULL;
    return 0;
}
```

注意我们在关闭解析器的时候，是如何保证它处于一致的（consistent）状态的,当我们对一个已经关闭的解析器或者垃圾收集器已经收集这个解析器之后，再次关闭这个解析器是没有问题的。实际上，我们使用这个函数作为我们的析构函数。他负责保证每一个解析器自动得释放他所有的资源，即使程序员没有关闭解析器。

最后一步是打开库，将上面各个部分放在一起。这儿我们使用和面向对象的数组例子（28.3 节）一样的方案：创建一个 metatable，将所有的方法放在这个表内，表的__index 域指向自己。这样，我们还需要一个解析器方法的列表：

```
static const struct luaL_reg lxp_meths[] = {
    {"parse", lxp_parse},
    {"close", lxp_close},
    {"__gc", lxp_close},
```

```
    {NULL, NULL}
};
```

我们也需要一个关于这个库中所有函数的列表。和 OO 库相同的是，这个库只有一个函数，这个函数负责创建一个新的解析器：

```
static const struct luaL_reg lxp_funcs[] = {
    {"new", lxp_make_parser},
    {NULL, NULL}
};
```

最终，open 函数必须要创建 metatable，并通过__index 指向表本身，并且注册方法和函数：

```
int luaopen_lxp (lua_State *L) {
    /* create metatable */
    luaL_newmetatable(L, "Expat");

    /* metatable.__index = metatable */
    lua_pushliteral(L, "__index");
    lua_pushvalue(L, -2);
    lua_rawset(L, -3);

    /* register methods */
    luaL_openlib (L, NULL, lxp_meths, 0);

    /* register functions (only lxp.new) */
    luaL_openlib (L, "lxp", lxp_funcs, 0);
    return 1;
}
```

第四篇 附录

A. 终端机控制符

在几十年前，流行的是各种终端机（terminal），它们都遵守 ANSI X3.64 控制字符序列标准（还有一些公司比如 IBM、DEC、HP 制定了自己的扩展标准），这些控制字符序列能帮助终端对显示的内容作一些处理，比如光标定位，字符色彩，背景色，窗口等等。

随着 PC 的流行，终端机被淘汰，但是原先的终端显示方式以及这些控制字符序列都被保留。人们开发出了虚拟的终端仿真程序来获得和从前一样的终端体验。自然，各种原先的终端机都有被仿真，较为流行的比如有 DEC-VT100，简称 VT100。在 linux 下可以用：

```
echo $TERM
```

来查看当前终端类型，也可以在/etc/termcap 或者/etc/terminfo/中查到完整的终端类型。

Windows 终端窗口的很多显示功能，都是通过 win32 API 来实现的。对于古老的 DOS 和 Win95/98，可通过加载 ansi.sys 来支持终端机控制符。下面是各版本 Windows 对其的支持：

WIN95：使用 ANSI terminal control，config.sys 中加入 device=ansi.sys

WIN98：使用 ANSI terminal control

WINNT：使用 console mode API

WIN2K：使用 console mode API

所以，对于目前的 Win2k/XP/2003 的用户，不能使用终端机控制符。

注：在 google 上查 ansi.sys，可找到许多相关的资料。