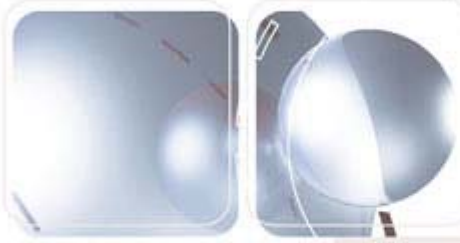
ROBERTO IERUSALIMSCHY

Programming in Lua



Lua.org

Programming in Lua

First Edition

作者：Roberto Ierusalimschy

翻译：Peter Pan

Simple is beautiful

简洁之美

版权声明

本书的版权归 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

作为脚本语言，Lua 以其简洁优雅著称，对 ANSI C 标准的遵循令其具有很好的可移植性，并能高效地运行于各操作系统平台。与其他脚本语言不同，Lua 自诞生起就致力于追求轻小便捷，精简的标准库易学易用，比起 Python 大而全的库，Lua 的优美是不言而喻的。

本翻译取材于http://www.lua.org/pil/，即《Programming In Lua》之英文原版，在尽可能依照原文内容的前提下对部分艰涩的叙述进行了斟酌，以提高本书的可理解性。

作为中文版的《Programming In Lua》，本书遵循其第二版的书名规则，命名为《Lua 程序设计》。本书主要针对的 Lua 版本号为 5.0，到目前为止尚无出版信息可查。正在发行中的《Lua 程序设计》第二版主要针对 Lua 5.1，因此与本书会有部分内容上的差异。尽管如此，本书（或称为《Lua 程序设计》第一版）的绝大部分内容仍然是可靠的，且本书来自于网络，服务于网络，相对于第二版，它是免费网络读物。能够给予有意了解学习 Lua 的人以帮助，是本翻译的最大宗旨。

另，在本书的翻译过程中，作为翻译原文的材料来自于www.lua.org，期间参考了www.luachina.net的译文草稿。

Peter Pan

2008 年 12 月 15 日

目录

第一篇 语言参考

第 0 章 前言

0.1 序言

目前，很多程序语言都致力于帮你编写成千上万行的代码，为此，使用者必须学习这些语言所提供的包、名字空间、复杂的类型系统、无数的结构以及成千上万的文档。

Lua 并不帮你编写成百上千行的程序代码，恰恰相反，Lua 让你使用数百行或者更少的代码来就能解决问题。为实现这个目标，Lua 依赖于其可扩展性，正如某些其他语言一样。与其他语言不同的是，Lua 不仅可以凭借自身编写的软件进行扩展，还能依靠其他语言编写的软件进行扩展，比如 C 和 C++。

从一开始，Lua 就被设计成容易和 C 语言或其他传统语言整合在一起，语言的二元性将带来诸多好处。Lua 之所以简洁，是因为 Lua 并不试图重复 C 语言已经做得很好的领域，比如，高性能、底层操作以及用于第三方软件的接口，而是通过 C 语言去完成这些任务。Lua 提供的机制是 C 所不善于的：高级语言、动态结构、简洁、易于测试和调试等。正因为如此，Lua 具有良好的安全性、自动内存管理、便捷的字符串处理功能及其他具有动态大小的数据。

Lua 不仅是易于扩展的语言，同时也是一种易整合的语言（Glue Language），Lua 支持基于组件的软件开发，在这种开发模式下，仅通过将高级组件整合在一起就能实现一个应用程序。一般情况下，组件由 C 或 C++之类的编译型静态语言编写，而 Lua 就类似于将各个组建联系起来的粘合剂。通常，组件或对象是一个具体的低层次的概念（如，部件或数据结构等），它们在程序开发过程中很少发生变化，且需要大量 CPU 的资源。而在软件生命周期中经常变化的应用程序整体部署，则通常由 Lua 负责。与其他整合技术不同的是，Lua 是一个完整的语言，因此，它不仅可以用于整合组件，也能够重塑组件，甚至创建全新的组件。

当然，Lua 并不是唯一的脚本语言，除了 Lua 以外，还有很多能够提供类似功能的脚本语言，例如，Perl、Tcl、Ruby、Forth、Python 等。虽然这些语言在某些方面有着同样的特色，但是下列特征是 Lua 特有的：

1、可扩展性。Lua 的扩展性非常卓越，很多人甚至将 Lua 视为用于构造特殊应用领域的工具。Lua 从一开始就被设计成易于扩展的语言，Lua 代码或 C 代码都能对其进行扩展，作为佐证，Lua 中的很多功能都是通过外部库来实现的。Lua 很容易与 C/C++、Java、Fortran、Smalltalk、Ada 以及其他语言进行交互。

2、简洁。Lua 本身十分简洁，但功能却很强大。这使得 Lua 易于学习，很适合小规模的应用，其完整的发布版（源代码、参考手册以及某些平台的二进制文件）仅用一张软盘就能装下。

3、高效。Lua 有很高的执行效率，相关测试表明 Lua 是最快的脚本语言之一。

4、可移植性。当论及可移植性，我们并不指仅在 Windows 和 Unix 平台上运行 Lua，而是指 Lua 几乎可以运行在任何现有的系统上，比如，NextStep、OS/2、PlayStation II（Sony）、Mac OS-9、OS X、BeOS、MS-DOS、IBM 主机、EPOC、PalmOS、MCF5206eLITE Evaluation Board、RISC OS 以及所有的 Windows 和 Unix 系统。用于这些平台的 Lua 源代码几乎是一样的，Lua 并不通过条件编译来实现可移植性，而是完全遵循 ANSI（ISO）C 的标准，这意味着只需 ANSI C 编译器，就能编译 Lua。

Lua 的大部分功能来自于它的库，这并非偶然。Lua 的强项之一就是可以通过新的类型和函数进行扩展，很多语言特性促成了该能力。动态类型检查极大地支持了多态性；自动内存管理简化了语言，因为不再需要考虑内存的分配与释放，也无需考虑内存溢出的问题；高级函数和匿名函数推动了参数化机制的产生，并使函数更为通用。

Lua 仅包含少量标准库。在严格受限的环境下使用 Lua，比如，嵌入型处理器，我们可以有选择地安装这些标准库。若运行环境近乎苛刻，我们甚至可以直接修改标准库的源代码，仅保留所需的函数。注意，Lua 的短小精悍（即使包含全部标准库）令它可以附带所有功能运行于大部分系统中。

0.2 Lua使用者

Lua 的使用者主要分为三大类：以嵌入方式在其他应用程序中使用 Lua 者、独立使用 Lua 者、整合使用 Lua 与 C 者。

很多人以嵌入方式在其他应用程序中使用 Lua，比如，CGILua（用于搭建动态网页）、LuaOrb（用于访问 CORBA 对象）。这些应用程序使用 Lua 的 C API 注册新函数、创建新类型、通过配置 Lua 来改变某些语言的行为模式。通常，这类应用程序的使用者甚至不知道 Lua 是一种被用于特定领域的独立语言，例如，CGILua 的用户倾向于认为 Lua 是一种专用的 Web 语言。

作为一种独立运行的语言，Lua 也是很有用的，在此，Lua 主要用于文本处理领域或只运行一次的小程序。对这类使用者而言，Lua 的主要功能来自其标准库，标准库提供了模式匹配和其他字符串处理的功能。我们可以认为，独立使用的 Lua 是文本处理应用领域的嵌入型语言。

还有一些使用者在其他语言平台上工作，他们将 Lua 作为库来使用。这些人用 C 语言编程的频率高于 Lua，尽管他们需要透彻了解 Lua，以便创建简单易用且与该语言紧密结合的接口。

本书主要面向以上三类使用者。本书的第一部分覆盖了语言本身的知识，并展示了语言的潜在能力。在关注语言结构的同时，还使用了众多的例子来介绍 Lua 的实际应用。第一部分既涵盖了 Lua 的基本概念，如，控制结构，也包含了一些高级内容，比如迭代器（Iterator）和协同程序（Coroutine）。

本书的第二部分将重点放在 Lua 的唯一数据结构（即，Lua 表）上。我们在该部分中将讨论数据结构、持久性、包及面向对象的编程，在此你将看到 Lua 的真正强大之处。

本书的第三部分介绍了 Lua 的标准库，这将有助于那些独立使用 Lua 的读者。该部分的每一章对应一个标准库：数学库、表库、字符串库、输入输出库、操作系统库、调试库。

本书的最后一部分介绍了 Lua 和 C 之间的 API，这对于那些希望完全发挥 Lua 能力的 C 程序员而言将很有帮助。在这部分中，将使用 C 语言作为程序语言，而不是 Lua，我们可能需要即时转换角色。因此，对某些读者而言，这部分是无关紧要的，而对另一部分读者而言，这部分的内容才是真正至关重要的。

0.3 相关资源

如果你想真的学习某一门语言，那么该语言的参考手册是必备的，本书并不能取代 Lua 参考手册的位置，刚好相反，两者本书与 Lua 参考手册互为补充。Lua 参考手册仅描述语言本身，因此它既不会给出参考例子，也不会讨论语言结构方面的基本原理，不过 Lua 参考手册将描述整个 Lua 语言，而本书将忽略语言中极少用到的部分。另外参考手册是 Lua 的权威性文档，任何本书与参考手册相悖的地方都以参考手册为准。Lua 参考手册以及其他更多信息请访问 Lua 站点：http://www.lua.org。

在Lua用户社区也有不少有用的信息，社区的网址为http://lua-users.org，在众多可用资源中，它提供了学习Lua的导引资料、第三方的Lua包和文档，以及官方邮件列表的存档。你也可以访问本书的网站http://www.inf.puc-rio.br/~roberto/book/，在那你可以找到最新的勘误表、本书中某些例子的代码，以及一些额外的材料。

本书的描述对象是 Lua 5.0。如果你正在使用最新版本的 Lua，请查阅 Lua 参考手册来了解不同版本之间的差异。如果你还在使用旧版本的 Lua，那么现在是更新的时候了。

0.4 本书的体例

在阅读本书之前，需要了解某些符号的约定使用方式。

标记“-->”用于表示 Lua 语句的输出，或表达式的结果:

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

由于一对连接号“--”在 Lua 中表示注释语句的开始，因此在程序中使用上述标记不会带来任何问题。

标记“<-->”用于表示等价：

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

即对 Lua 而言，上例中的 this 与 that 没有任何区别。

0.5 关于本书

开始写本书是 1998 年冬天（南半球，那意味着年中，所以“冬天”更像是一年的盛秋），那时，Lua 的版本号码仍然为 3.1，从那以后，Lua 经历了两次大改动，第一次是 2000 年 4.0 版，之后是 2003 年 5.0 版。

很明显，这些变化对本书有着重大的影响，部分内容失去了其存在理由，例如，有关超值（Upvalue）复杂性的解释。部分章节被重写，比如，C-API，另外一些章节被新增进来，比如，协同程序（Coroutine）。

不太明显的是，Lua 语言本身的发展对本书所产生的重大影响，语言中的某些巨变尚未在本书中被涵盖，这并非偶然。在编写本书各章的过程中，经常会有犹豫不决的时候，经常地，我不知道该从何开始叙述，或如何将一个问题提出来，有时，你还会发现将一个自己认为非常容易的东西解释清楚有多么的困难。这些在写作中遇到的困难暗示着 Lua 仍然需要继续改进。还有些时候，当我顺利地完成了某个章节时，却发现没人能够理解或认同它。大多情况下，作为本书的写作者，我发现问题出自我自己，不过偶尔也会因此而看到语言本身的一些缺陷。比如，从超值到词法定界（Lexical Scoping）的转变受启发于对一次无意义尝试的抱怨，在那次尝试中，曾试图在本书先前的草稿中将超值描述为词法定界）。

语言中的某些变化推迟了本书的完成（而为了完成本书可能又推迟了 Lua 完善的进程），原因至少有两点：较早期版本而言，Lua 5.0 是一个更为简洁更为成熟的语言（部分归因于本书），其次，本书更加重视语言的整体性（语言本身及其外围的内容），因此语言变得越来越复杂，这是本书的第一目标，另外一个目标便是希望 Lua 能够得到更广的普及。

0.6 感谢

非常感谢那些在我完成本书过程中给予帮助和支持的人们。

协助我完成 Lua 的 Luiz Henrique de Figueiredo 和 Waldemar Celes 在改进 Lua 的过程中给予了极大的支持，因此使我有了更多的精力投入本书的写作，Luiz Henrique 还在本书的内部设计中提供了极大的帮助。Noemi Rodriguez、André Carregal、Diego Nehab 以及 Gavin Wraith 复审了本书的草稿，并提出了很多有价值的建议。Renato Cerqueira、Carlos Cassino、Tomás Guisasola、Joe Myers 和 Ed Ferguson 也给出了多项重要的建议。本书的封面由 Alexandre Nakonechnyj 设计，他在本书的内部设计中也给予了支持。Rosane Teles 负责 CIP 数据的准备。

感谢他们所有人。

第 1 章 起点

为了保持程序语言的传统开场白，我们的第一个 Lua 程序将打印出：Hello World。

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

假定你运行独立的 Lua 解释器，你只需要以包含上述程序的文件名为参数执行 Lua 解释器，就能运行上述程序代码。如果上述代码被保存在名为 hello.lua 的文件中，你只需在命令行执行下列操作：

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

看到结果了吗？

让我们再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))
```

上述例子定义了一个计算阶乘的函数，函数要求用户输入一个数字 n，然后打印 n 的阶乘。

如果你正在将 Lua 嵌入其他应用程序（比如 CGI-Lua 或 IUP-Lua），你可能需要参考该项应用的参考手册以便执行你的 Lua 程序。尽管如此，Lua 并没有发生任何变化，不管你如何使用 Lua，在此有关 Lua 的介绍都是有效的。但是作为初学者，我们推荐你使用独立的 Lua 解释器（即可执行的 Lua）来运行这里的例子。

1.1 代码段（Chunks）

代码段是一系列语句，Lua 执行的每一块语句，比如一个文件或交互模式下的每一行都是一个代码段。

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号隔开：

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

一个代码段可以是一个语句，也可以是一系列语句的组合，还可以是函数，代码段可以很大，在 Lua 中几兆字节大的代码段是很常见的。

你还可以以交互模式下运行 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：print "Hello World"），在你键入回车之后会被立即执行。如果想要退出交互模式，你可以键入文件结束符（在 Unix 下是 Ctrl+D，在 DOS 或 Windows 下是 Ctrl+Z），调用 OS 库的 os.exit()函数也可以退出交互模式。

在交互模式下，Lua 通常把每一个行当作一个代码段，但如果 Lua 检测到某一行无法构成一个完整的代码段时，他会等待继续输入直到得到一个完整的代码段。在 Lua 等待续行时，将显示不同的提示符（一般是>>提示符）。

可以通过指定参数让 Lua 执行一系列代码段。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
```

命令首先在一个代码段内先运行 a 然后运行 b。注意：-l 选项会调用 require 函数，它将会在指定的目录下搜索文件，如果环境变量没有设好，上面的命令可能无法正确运行。我们将在 8.1 节详细更详细的讨论 require 函数。

-i 选项要求 Lua 运行指定代码段后进入交互模式：

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

上述将在一个代码段内先运行 a 然后运行 b，最后直接进入交互模式。

另一个连接外部代码段的方式是使用 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空值 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

标识符：以字母或下划线开头的字母、下划线和数字序列。最好不要使用以下划线开头的大写字母标示符，因为这是 Lua 保留字的书写风格。Lua 中，字母的含义是依赖于本地区域环境的，在某区域环境中你可以定义诸如índice 或 ação 的变量名称，但是使用此类字母作为标识符的程序将很难移植到其他环境中。

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：and 是保留字，而 And 或 AND 却不是。

Lua 的单行注释语句以双连接符“--”开头，结束于该注释所在地行尾。与其他语言一样，Lua 也提供了注释语句块或多行注释，注释语句块以“--[[”开头，在遇到“]]”时结束。在使用注释语句块的时候，可以像下面这样：

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

如果我们在第一行行首加上一个连接符：

```
---[[
print(10)         --> 10
--]]
```

在第一个例子中，位于最后一行的双连接符位于注释语句块中，而在第二个例子中，“---[[”并不能作为注释语句块的开头，因此 print 语句没有被注释掉，与第一个例子不同的是，最后一行成了一个独立的注释语句。

1.4 独立的Lua解释器

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 在开始运行之前，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，如加载包，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，定义变量不需要声明类型。Lua 中的 8 个基本类型分别为：nil、boolean、number、string、userdata、function、thread 和 table。type 函数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 空值类型

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

2.2 布尔类型

布尔类型只有两个取值 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 数值类型

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 的数值类型可以处理任何长整数而不用担心误差。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数值类型，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，数值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 字符串类型

字符串类型表示字符串的序列。Lua 的字符为 8 位字节长度，所以字符串可以包含任何数值字符，包括嵌入的“\0”。这意味着你可以在一个字符串里存储任意的二进制数据。你不能像在 C 语言中那样修改 Lua 字符串，但你可以创建一个新的变量存放你修改过的字符串，比如：

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

字符串和其他对象一样，Lua 自动对其进行内存分配和释放，一个字符串可以只包含一个字母也可以包含一本书，Lua 可以高效地处理长字符串，一兆大小的字符串在 Lua 中是很常见的。字符串可以用单引号或者双引号表示：

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

为了风格统一，最好使用一种，除非遇到两种引号嵌套使用情况。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）的方式表示字母。

"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，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 会自动在字符串类型和数值类型之间自动进行类型转换，当一个字符串使用算术操作符时，字符串就会被转成数值类型。

```
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 期望一个字符串类型但却碰到数值类型时，会将数值转成字符串类型。

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

“..”在 Lua 中是字符串连接符，当在一个数值类型后使用“..”时，必须加上空格以防止被误解释。

尽管字符串和数值可以自动转换，但两者是不同的，像 10 == "10"这样的比较永远都是错的。如果需要显式地将字符串转成数字可以使用 tonumber()函数，如果字符串不是正确的数值，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()将数值转成字符串类型，这种转换总是有效的：

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

2.5 表类型

表类型实现了关联数组。关联数组是这样一类数组，它可以用数值、字符串或者其他被语言许可得值作为索引，除了空值以外。另外，表没有固定尺寸，你可以动态增加元素。表是主要的，实际上，也是唯一的 Lua 数据结构，其功能非常强大。对于表，我们可以用一种简单、统一且有效的方式来表示普通的数组、符号表、集合、记录、队列以及其他数据结构。Lua 也使用表来表示 Lua 包，当我们写 io.read，它的意思是 io 库里的 read 项。在 Lua 中，意为“io 表中索引为 read 的项”。

Lua 中的表既非值也不是变量，而是对象。如果你熟悉 Java 或者 Scheme 中的数组，那么你应该明白它的确切意义。但如果你只知道 C 或者 Pascal 中的数组，那么你可能还需要更进一步地了解它。你可以认为 Lua 表是一个动态的对象，而你只需要在程序中只是通过引用或者指针的方式来操纵它。而实际上，这个对象的背后并没不存在任何拷贝或新创建的表。此外，你不需要在使用之前声明一个表，实际上，你也无法声明一个表。表是通过构造表达式来创建的，最简单的创建方式即使用“{}”：

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

Lua 表总是“匿名”的：一个表变量和表本身之间不存在固定的联系：

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

当程序中的表不再被引用，Lua 的内存管理机制最终会删除表并释放表所占用的内存空间。

表可以存储不同类型索引的值，并根据需要在尺寸上增长，以便适应新添加的项。

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

注意上述最后一行：就像全局变量那样，未经初始化的表域，其值为空值，同样，你也可以将一个表域赋成空值来删除这个域。这并不是一个巧合，因为对于全局变量，Lua 也同样使用一个普通的表来存储它们。更多相关信息请参考第十四章。

你可以用表域的名称作为一个索引，来表示记录。在 Lua 中可以用 a.name 来便利地表示 a["name"]，因此我们可以更简洁地重写上例中的最后几行：

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

在 Lua 中，上述两种方式是等价的，因此可以自由地混合使用。但作为程序读者的普通人而言，每

种写法可能代表着不同的意义。

初学者常犯的一种错误是将 a.x 和 a[x]弄混淆了。这里的第一种写法代表了 a["x"]，也即，表 a 中由"x"索引的项，而第二种写法表示表 a 中由变量 x 所对应的值索引得项。看下例中的区别：

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

你可以简单地使用整数索引来表示一个普通的数组。你无法声明该数组的尺寸，而仅仅是初始化你所需要的元素。

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

遍历数组中的元素时，首个未初始化的元素其值为空值，你可以使用此特性来判定数组的末尾。比如你可以用下列代码打印上个例子中读取的数据：

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

Lua 的基本库提供了 ipairs 函数，它允许你方便地遍历数组中的各元素，它默认上述有关数组末尾的约定进行工作。

既然你可以用任何值来索引数组，那么你也可以用任何数值来索引数组，但在 Lua 中一般用 1 来索引数组的第一个元素（在 C 语言中是 0），因为 Lua 标准库遵循此约定。

对于表的索引而言，尽管数值 0 和字符串"0"都能胜任，但是两者是不同的，因此它们索引得位置也是不同的。同样地，字符串"+1"、"01"和"1"各自索引不同的位置。如果不确定具体的索引类型，可以使用显式的转换：

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

如果不加注意，你可能会在你的程序中引入难以发现的错误。

2.6 函数类型

函数是第一类值（和其他变量相同），这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，此外该特性在 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、math 库、debug 库。

2.7 Userdata和Threads

userdata 可以将 C 数据存置入 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、一元运算符、二元运算符和函数调用。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，因此 nil 只和自己相等。Lua 通过引用比较表、userdata、函数，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
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，其他为真，0 也为真.

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 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，最简单的构造器是“{}”，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 用"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记录使用可以这样做：

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

不管用何种方式创建表，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 都会创建一个新的表，可以使用表构造一个链表：

```
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链表风格和记录风格进行初始化，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“[表达式]”显式地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])         --> ---
```

链表风格初始化和记录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逗号是可选的，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和局部变量声明。

4.1 赋值

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()
```

如果函数 f()返回两个值，则第一个返回值赋给 a，第二个返回值赋给 b。

4.2 局部变量与代码块（Blocks）

除了全局变量，Lua 还支持局部变量。使用 local 关键字可以创建一个局部变量，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。代码块是指一个控制结构体，一个函数体，或者一个代码段（Chunk，即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 是一个完整的代码段，在交互模式下执行完这一句后，Lua 将开始一个新的代码段，这样第二句的 i 已经超出了它的有效范围。可以将这段代码放在 do..end 语句块中（相当于 C/C++中用{}标识的语句块）。

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

我们给代码块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，使用 do..end 语法是很有用的。

```
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 控制语句

Lua 中只提供了简洁且方便的控制结构，其中，if 语句用来表示条件控制，而 while、repeat 和 for

来表示循环控制。所有的控制结构都有一个显式的终结符：if、for 和 while 结构以 end 结束，repeat 结构以 until 结束。

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

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
..                          ---> more elseif-blocks
else
    else-part
end
```

while 语句。Lua 首先测试 while 条件，如果条件为假，那么程序跳出 while 循环，如果条件为真，那么程序执行 while 循环内部的代码块，并重复这一过程。

```
while condition do
    statements
end
```

repeat-until 语句。只要 until 条件为真，repeat-until 语句便会重复执行其内部的代码块。因为条件测试语句位于代码块末尾，所以代码块总是至少被执行一次。

```
repeat
    statements
until conditions
```

for 语句有两大类：数值 for 循环和范型 for 循环。

数值 for 循环：

```
for var = initial, final, step do
    loop-part
end
```

for 将用 step 作为步进从 initial（初始值）到 final（终止值），执行 loop-part。其中 step 可以省略，默认步进值为 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、控制变量是由 for 语句自动声明的局部变量，它只在 for 循环体内有效。

```
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。

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。

再看一个遍历表索引的例子：

```
-- 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
```

如果不想进行手工的操作，那么可以利用下列代码自动构造反向表：

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

如果你对范型 for 还有不清楚，那么在后面的章节我们还会继续学习到。

4.4 break和return语句

break 和 return 允许我们从内部语句块跳出。

break 语句用来跳出当前循环（for、repeat、while），因此在循环外部不可以使用。使用 break 跳出之后，程序从紧接着中断点处继续运行。

return 用来从函数返回结果或者只是无值返回。当一个函数自然结束时，结尾会有一个默认的 return，在这种情况下，可以不用 return 语句返回。

Lua 语法要求 break 和 return 只能出现在代码块的结尾一句，也就是说，作为整个代码段的最后一句、用在 end 之前、else 之前或者 until 之前。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，可以显式地使用 do..end 来实现：

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

第 5 章 函数

函数通常由若干语句或者表达式组成，在 Lua 中，函数是抽象化语句或者表达式的一种重要机制。是对函数有两种用途：

1、完成指定的任务，这种情况下函数被作为调用语句。在某些其他语言中，这类函数也被成为过程或子程序；

2、计算并返回值，这种情况下函数被作为表达式。

函数语法：

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

无论是上述哪种情况，我们都将函数的参数列表写在一对圆括号内。如果函数的参数列表为空，则必须使用一对空圆括号表明是函数调用。

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

上述规则有一个例外情况，当函数只有一个参数，且这个参数是字符串或者表构造器的时候，圆括号可以省略：

```
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 也为面向对象方式的函数调用提供了特殊的语法，表达式 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 或者 C 语言编写的，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。不管是用 Lua 还是用 C 语言定义的函数，调用它们的时候没有任何差别。

Lua 函数中的实参形参匹配与赋值语句中的变量值匹配类似，多余部分被忽略，缺少部分用空值补足：

```
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)
```

尽管这种匹配模式会导致程序错误，但是它同样也是非常有用的，尤其是对于参数的默认值而言。参看下列函数，它将一个全局计数器累加：

```
function incCount(n)
    n = n or 1
    count = count + n
end
```

该函数指定 1 为其参数的默认值，也就是说，无参调用 incCount()将会令 count 加 1。当你调用 incCount()函数，Lua 首先将参数 n 初始化为空值，然后将其与 or 的第二个操作数，也就是 1 的计算结果作为默认值赋给 n。

5.1 多返回值

一个非传统性但却很方便的 Lua 特性是，Lua 函数可以返回多个结果。不少 Lua 预定义函数都有多个返回值，比如 string.find 函数，它在给定字符串中定位匹配子串，该函数有两个返回值：第一个返回值指出子串在给定字符串中的首位置，第二个返回值指出子串在给定字符串中的结束位置。如果没有找到子串，则返回空值 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。下面的函数返回数组中的最大值及其在数组中的位置：

```
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。通常情况下，当函数作为表达式，Lua 只保留第一个返回值。当函数为表达式列表中最后的或者唯一的表达式时，我们才能得到所有的返回值。这类表达式列表在 Lua 中分成四类：多重赋值、函数的参数列表、表构造器和 return 语句。为了说明这几类用法，我们首先定义下列三个函数供后面的例子使用：

```
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，多重赋值。有以下几种情况：

1、当函数作为最后的或唯一的右值表达式，它将返回尽量多的返回值：

```
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'
```

2、如果函数的返回值不够，它将返回空值来补足：

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

3、如果函数不是最后的或唯一的右值表达式，它只返回一个返回值：

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

第二，函数的参数列表。当函数调用为最后的或唯一的表达式，它所有的返回值都将成为参数。这种情况早在我们开始使用 print 函数时就有接触：

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

上例中最后一行，因为 foo2()并不是最后的或唯一的表达式（它出现在表达式列表当中），Lua 自动调整其返回值，只返回一个，因此只有“a”在字符串连接运算中被使用。

上述 print 函数可以接受可变数量的参数（在下一节我们将介绍参数数量可变的函数）。如果有 f(g())这样一个函数，函数 f 的参数数量固定，那么 Lua 会调整函数 g 的返回值个数，以满足 f 的参数需求。

第三，表构造器。与多重赋值一样，当函数为构造器中最后的或唯一的元素时，构造器接受函数所有的返回值：

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

否则，Lua 会调整返回值，只返回一个：

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

第四，return 语句。return f()将返回 f 函数所有的返回值：

```
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)
```

我们可以强制一个函数只返回一个返回值，而只需要将返回值列表用圆括号括起来：

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

但是注意 return 语句并不需要圆括号来将其返回值括起来。一个诸如 return (f())的语句总是返回一个返回值，不管函数 f 原先返回多少个返回值。也许这是你想要的，如果不是，请注意这种用法。

Lua 有一个特殊函数 unpack，它接受一个数组作为参数，从索引 1 开始，返回该数组中的所有元素：

```
print(unpack{10, 20, 30})       --> 10    20    30
a, b = unpack{10, 20, 30}       -- a = 10, b = 20, 30 is discarded
```

函数 unpack 主要被用来实现范型调用机制，范型调用机制允许你使用任意的参数动态地调用任何函数。在 ANSI C（标准 C）中，是无法做到的。在 C 语言中你可以使用函数指针调用函数变量，也可以声明参数可变的函数，但不能兼具两者。在 Lua 中如果你想调用可变参数的函数变量只需要这样：

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

unpack 返回数组 a 中所有的元素作为 f()的参数。见下例：

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

上述例子中的最后一句等价于 string.find("hello", "ll")。

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

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

当我们用一个数组参数调用上述函数，第二个参数 i 会自动取得默认值 1。

5.2 可变参数

某些 Lua 函数可以接受可变数目的参数，比如我们用过的 print 函数，我们可以带一个参数、两个参数或者更多参数来调用它。

假设我们现在想要重新用 Lua 定义 print 函数。我们可能无法打印，因为系统里可能没有 stdout 之类的对象，因此我们可以不进行打印输出，而是将需要打印的内容保存到一个全局变量里，以便后续之用。下面就是用 Lua 重定义过的 print 函数：

```
printResult = ""

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

```
    printResult = printResult .. "\n"
end
```

函数参数列表中的三点（...）表示该函数有可变数量的参数。当该函数被调用的时候，它所有的参数会被存入一个表变量，该变量是一个名为 arg 的隐藏变量，表中除了存储所有的参数之外，还附带一个额外的域 n 用来存储参数的个数。

有时候，一个函数的参数列表可以由若干固定参数外加可变数量的参数构成。让我们先看一个例子，假设我们定义了一个返回多个返回值的函数，当我们需要将这个函数写入一个表达式，显然只有第一个返回值会被采用。但是，有时候，我们可能需要另一个返回值，一个典型的解决方案是使用哑元变量（Dummy Variable，下划线），比如，我们只需要 string.find 的第二个返回值，我们可以这么做：

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

另一个解决方案是定义一个名为 select 的函数，它选择并返回指定的返回值：

```
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
```

注意调用 select 函数的时候总是有一个固定的参数。该函数列表由一个固定的“变量选择符”参数和可变数量的参数组成，这里的可变数量的参数可以是某一个函数的返回值。为了适应固定参数，函数必须在可变数量的参数（三点）前声明固定参数。之后调用该函数的时候，Lua 将前面的若干实参赋给固定参数，而将剩余的实参（如果还有剩余的话）赋给可变数量的参数。为了更好地理解，我们定义了如下函数，并列出了函数参数列表的赋值情况：

```
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}
```

看过上面的例子之后，我们便可以很容易地定义 select 函数了：

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

有时候需要将函数的可变数量的参数传递给另一个函数作为参数，这时候，可以使用我们前面介绍过的 unpack(arg)返回 arg 表所有的可变参数。一个极佳的例子是一个负责输出格式化文本的函数，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）和一个输出函数 io.write()。当然，我们可以将这两个函数合并成一个新函数，只是这个新的函数必须将其可变数量的参数传递给 string.format，该函数如下所示：

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

5.3 具名参数

Lua 的函数参数传递机制是“位置相关的”：当我们调用一个函数，实参会按顺序依次传给形参，第一个实参赋值给第一个形参，第二个实参赋值给第二个形参，以此类推。但是有时候，具名的参数显得非常有用，比如 OS 库中的 rename 函数可以重命名文件，问题是我们经常忘记新文件名和旧文件名在参数列表中的顺序，因此，我们可能需要重定义该函数，令其接受两个具名参数：

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

上面这段代码是无效的，因为 Lua 并不显式地支持这一语法，但是我们可以在语法上稍加修改就能产生所期望的效果。具体的做法是：将所有的参数放在一个表中，并将该表作为函数的唯一参数。Lua 语法支持函数实参是表构造器的情况，因此我们的问题也就解决了：

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

根据这个想法，我们重定义 rename，它只有一个参数，通过解析该参数得到真正需要的参数：

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

当函数的参数为数众多的时候，这种函数参数的传递方式是很有效的，并且该种传递方式令所有的实际参数都是可选的。例如 GUI 库中创建窗体的函数有很多参数，它们都以具名方式传递，其中的大部分参数是可选的：

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

上述 Window 函数可以自由地对强制参数进行检查、设置参数的默认值等等。假设我们已经有一个名为_Window 的函数，它负责实际创建新的窗口（因此需要得到所有可能的参数）。那么 Window 函数可以如下定义：

```
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 函数高级应用

Lua 中的函数是带有词法定界（lexical scoping）的第一类值（First-Class Values）。

第一类值意为，Lua 中的函数是一个值，如同传统的数值或字符串值一样。函数可以被存入局部或全局变量中，可以存入 Lua 表中，可以作为其他函数的参数被传递，还可以作为其他函数的返回值。

词法定界意为，Lua 函数可以存取其函数体包含的范围内的变量，同时也意为着 Lua 正确地包含了 λ-Calculus（嵌套的函数可以访问它外部函数中的变量）。在稍后的章节我们将看到，这显然无害的特性给 Lua 带来强大的功能，因为它允许我们在 Lua 中应用来自于函数型语言的多种强大的编程技术。就算你对函数型编程毫无兴趣，了解一下如何开发这些技术是值得的，因为它们将令你的程序看起来更加简洁。

在 Lua 中，一个有关函数的难解的概念是，与其他类型的值一样，函数也是匿名的，它们并没有名字。当我们提到函数名（比如 print），实际上是指一个持有函数功能的变量，因此我们可以像操纵其他变量那样操纵函数变量。下面一个看起来有点傻的例子展示了这一概念：

```
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
```

稍后我们还将看到更有用的相关应用。

既然函数是值，那么使用表达式也应该可以创建函数。实际上，我们在 Lua 中经常这样写：

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

这实际上是 Lua 中语法的特例，换句话说，下面的写法相对来说更为漂亮：

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

实际上，函数定义是一个语句，更确切地说，是一个将类型为 Function 的值赋给一个变量得赋值语句。我们可以将表达式 function(x) ... end 视为一个函数构造器，就像使用{}作为一个表构造器一样。我们称这种函数构造器的结果为匿名函数，尽管我们经常用一个全局的名字来命名函数，但是在很多情况下我们使用函数会选择匿名的方式。让我们先看几个例子。

Lua 的 table 标准库提供一个名为 table.sort 的排序函数，它接受一个表作为参数并排序其中的元素。类似这种函数必须能够提供多种不同的排序方式：升序还是降序、数值顺序或者字母顺序、是否按照表的索引来排序，等等。Lua 在 table.sort 中提供了一个可选的参数（它接受一个排序函数作为参数值），而不是直接提供所有可能的排序函数。这个作为参数存在地排序函数接受两个参数，并返回一个布尔值来指示在排序时两个参数中到底哪个应该排在另一个的前面。假设存在一个如下所示的记录表：

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

如果我们想按照字母降序的方式，对该表的 name 域进行排序，那么我们只需要像下面这样做：

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

你可以发现上例中使用匿名函数是如此的便利。

以其他函数作为参数的函数被称作高级函数（Higher-Order Function），如上面的提及的 table.sort。高级函数是一种强有力的编程机制，同时使用匿名函数作为它们的参数提供了很大的便利。但在 Lua 中，高级函数与普通函数没有特别的不同之处，它们的出现只不过是 Lua 将函数作为第一类值来对待的自然结果。

为了展示参数化函数，我们将给出一个有关高级函数的简单实现：plot 函数。这个函数负责绘制数学图形。下面我们将利用一些转义序列以实现在标准终端上的绘图，当然你有可能需要修改某些控制符或序列来适应你的终端类型：

```
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
```

函数定义完成后，你可以通过它来绘制正弦函数的图形：

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

当我们调用 plot 函数，匿名函数将被赋值给它的参数 f，之后 f 参数对应的函数将在 for 循环中被重复调用，以完成绘图的工作。

因为函数在 Lua 中是第一类值，我们可以将其存入全局变量、局部变量或表域中。之后我们会看到，将函数存入表域是某些高级应用的前提，比如 Lua 包以及面相对象的编程。

6.1 闭包（Closures）

当一个函数内部嵌套另一个函数时，位于内部的函数可以访问位于外部地函数的局部变量，这种特性被称作词法定界。尽管看起来很简单，事实却并非如此，词法定界（Lexical Scoping）与第一类值函数（First-Class Function）一起，在编程语言里是一个强有力的概念，但很少语言能够提供这种支持。

让我们从一个简单的例子开始，假定有一个学生姓名的列表和一个学生姓名与其成绩相关联的 Lua 表，如果现在想根据学生的成绩高低的降序方式对该 Lua 表进行排序，那么可以这样做：

```
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)
```

现在，更进一步假定我们需要在一个函数内完成此功能：

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

上述例子中的有趣之处在于，传递给 sort 函数的匿名函数可以存取 grades 参数，而该参数是 sortbygrade 函数的布局变量。在匿名函数内部，grades 既不是全局变量也不是局部变量，因此我们可以称它为外部的局部变量（External Local Variable）或者超值（Upvalue）。Upvalue 这个术语在意思上有些许误导，因为 grades 显然是一个变量，而非值。然而，在 Lua 中该术语有其历史的根源，另外，比起 External Local Variable，Upvalue 显得更加简短。

为什么外部的局部变量会如此有趣？那是因为函数是第一类值。请考虑下述代码：

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

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

上例中，匿名函数使用作为 Upvalue 的 i 来保存它的计数器。当我们调用匿名函数的时候，i 已经超出了作用范围，因为创建变量 i 的函数 newCounter 已经返回了。然而 Lua 按照闭包的概念正确地处理了这种情况，简单地说，闭包的概念包含了一个函数，以及该函数正确地存取其 Upvalue 所需的一切。如果我们再次调用 newCounter，它将创建一个全新的局部变量 i，也就是说，我们得到了一个作用在新变量 i 上的新闭包。紧接上一个例子，我们可以得到如下结果：

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

```
print(c2())              --> 2
```

c1 和 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。技术地讲，闭包本质上是指变量值，而不是指函数本身，因为函数仅仅是闭包的一个原型声明。尽管如此，在不至于混淆的前提下，我们将继续使用函数这个术语来指代闭包。

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，闭包可以作为高级函数的参数（如前面提及的 sort 函数），同时也为构成多其他函数的嵌套函数提供了支持（如前面提及的 newCounter 函数）。这一机制使得我们可以在 Lua 的函数型编程中组合出奇幻的编程技术。闭包也常被用在回调函数中，比如在 GUI 环境中创建一系列的按钮控件，但用户按下按钮时，回调函数即被调用，而不同的按钮被按下时需要处理的任务有些许区别。举个例子，一个计算器需要十个相似的按钮，每个按钮对应一个数字，你可以使用下面的函数创建它们：

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

在这个例子中，我们假定 Button 是一个用来创建新按钮的函数，label 代表按钮上的标签，action 是按钮被按下时将被调用的回调函数（实际上这个函数是一个闭包，因为它可以存取作为 Upvalue 的 digit 变量）。digitButton 返回之后，局部变量 digit 超出了其作用的范围，但回调函数仍然可以被调用且能够存取 digit 局部变量。

闭包在完全不同的上下文中也是很有用的。因为函数被存储在普通的变量内，因此我们可以很方便地重新定义函数，或者预定义函数。这个能力是 Lua 如此灵活的原因之一。但是经常性地，当你重新定义函数的时候，你往往需要原始的函数来完成新函数的定义。假如你想重定义 sin 函数，使其接受一个度数作为参数，而不是一个弧度参数，新函数得将弧度参数转化为度数参数，并调用原始的 sin 函数来完成这个工作，它的代码类似这样：

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

一个更易懂的方案是这样：

```
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的函数放在一个局部变量内，从而，访问原始函数的唯一方式就是通过新版本的函数。

利用同样的特性，我们可以创建一个安全的运行环境（也称作沙箱，就像 Java 里的沙箱一样），当我们运行一段不可靠的代码（比如我们运行一段从网络上下载的代码）时安全的环境是必要的。比如，我们可能需要限制这个程序可以访问的文件，那么我们可以用闭包的方式来重新定义 IO 标准库里的 open 函数。

```
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
```

上述程序之所以有效，是因为当完成上述定义之后，程序将无法再通过未经限制的 open 函数来存取文件，除非又再次重新定义了一个限制版本的 open 函数。新函数通过将不安全版本的函数存入闭包的一个局部变量，来限制外部对其的访问。在这种功能的帮助下，你可以仅通过 Lua 自身建立 Lua 的沙箱，而依然保留 Lua 的灵活性。当然，这不是唯一的解决方案，Lua 还提供了元机制（Meta-Mechanism）让你修剪自己的运行环境以适应特殊的安全需要。

6.2 非全局函数

一个由第一类值函数直接导致的结果是，我们不仅可以将函数存储于全局变量，也可以将其存储于表域或局部变量。

我们已经看过一些函数作为表域的例子：大部分 Lua 标准库是使用这种机制来实现的，比如 io.read、math.sin 等。为了创建此类函数，我们只需要将函数语法和表语法整合在一起：

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

当然，我们也可以使用表构造器来完成同样的定义：

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

另外，Lua 也提供了第三种定义此类函数的语法：

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

当我们将函数存储于一个局部变量时，我们得到了一个局部函数，所谓局部函数是指，它就像局部变量一样只在一定范围内有效。这种应用常见于 Lua 包：因为 Lua 把代码段当作函数处理，在代码段内部可以声明局部函数，它仅仅在该代码段内部有效。词法定界保证了 Lua 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
```

需要注意的是声明递归局部函数的情况。下列方式并不像想像中那样奏效：

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

当 Lua 编译上述代码中的 fact(n-1)时，会发现局部函数变量 fact 还没有被定义过，因此 Lua 会假设这里的 fact 函数是一个全局函数，而不是一个局部函数。为了解决这一问题，我们必须在定义函数之前定义该局部变量：

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

这时，函数内部的 fact 变量会指向局部变量。在定义函数时，fact 的值是什么无关紧要，因为当函数运行的时候，fact 已经被正确地赋值。这就是 Lua 为局部函数而扩充的特殊语法结构，利用它你可以轻松地使用递归函数：

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

对于直接递归函数，扩展后的 Lua 语法支持上述例子中介绍过的两种定义方式。但是如果在定义非直接递归局部函数时，则还是需要先显式地声明过才能使用：

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

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

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

6.3 正确的函数尾调用

Lua 函数的另一个有趣特性是它们会执行正确尾调用（不少作者选择 Proper Tail Recursion 这个术语，即正确的尾递归，尽管概念本身并不涉及递归）。

所谓尾调用是一种类似在函数末尾执行地址跳转（goto）的调用方式，当函数最后一个动作是调用另外一个函数时，除了调用其他函数外没有执行任何多余的操作，我们称这种调用为尾调用。下列代码中，对于函数 g 的调用就是尾调用：

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

上述例子中，函数 f 调用函数 g 之后，不会执行任何多余的操作，在这种情况下，当被调用的函数 g 运行结束时不需要返回到调用者函数 f。因此，执行尾调用时不需要在栈中保留有关调用者的任何信息。某些语言的实现，比如 Lua 解释器，充分利用了这种特性，在处理尾调用时不使用额外的栈空间，通常我们称这种语言的实现支持了正确的尾调用。

由于尾调用不需要使用栈空间，因此尾调用的嵌套层次是没有限制的。比如，当我们调用下列函时，不管 n 为何值都不会导致栈溢出。

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

在使用正确的尾调用前，我们必须首先明白什么是尾调用。一些调用者函数调用其他函数后，虽然没有做任何操作，但却不属于尾调用。如同下列函数 g 的调用就不属于尾调用：

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

之所以不属于尾调用，是因为调用 g 返回之后，函数 f 还必须丢弃 g 可能有的返回值（将返回值从栈中弹出）。同样地，下面几例中的调用也不是尾调用：

```
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
```

在 Lua 中，只有类似 return g(...)这种格式的调用属于尾调用，其中，g 和 g 的参数都可以是复杂的表达式，因为 Lua 会在调用之前计算表达式的值。例如下面的调用属于尾调用：

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

就像一开始讲的，尾调用是一种地址跳转指令。正确的尾调用在 Lua 状态机编程应用领域显得相当有益，此类应用要求函数与状态对应，在需要改变状态的时候只需要调用或者跳转到一个特定的函数。迷宫游戏是一个很好的例子，一个迷宫内有多个房间，每个房间有东西南北四道门。为了移动，用户必须在每一步输入一个移动的方向，如果该方向存在对应的门，则移动到对应的房间内，否则程序打印警告信息。游戏的目标是，从一个最初的房间到达一个最终的房间。

经由上述分析可以了解到，迷宫游戏恰恰是一个典型的状态机，当前所在的房间就是状态。我们可以为每一个房间定义一个函数来实现这个迷宫游戏，之后，我们使用尾调用从一个房间移动到另外一个房间。一个只有四个房间的迷宫游戏代码如下：

```
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()，从初始的房间即房间一，来开始这个游戏。

如果没有正确的尾调用，用户每移动一次都需要在栈中增加一个层次，可想而知，经过一定数量的移动之后，栈将会溢出。而使用正确的尾调用可以无限次数地进行尾调用，每次尾调用只是执行一个跳转指令。

就此简单的游戏，你会发现使用数据驱动的编程方式，即用 Lua 表来表述房间和移动信息，将会是一个更好的方案。但是，如果并不是每个房间的状况都一致，那么上述状态机的方式将是相当合适的选择。

第 7 章 迭代器与泛型for

本章我们将讨论与范性 for 的迭代器（Iterator）相关的内容，我们从一些简单的迭代器开始，然后我们学习到如何利用范性 for 的强大能力来写出更高效的迭代器。

7.1 迭代器与闭包

迭代器是一种让你在一个集合的元素上自由移动的结构，因此你可以借助它来遍历集合中的每一个元素。在 Lua 中，我们使用函数来描述迭代器，每次调用该函数就返回集合中的“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，根据这个状态信息它可以知道它当前所处的位置以及如何跳转至下一个状态。闭包为迭代器的实现提供了很好的机制，记住，闭包是一个能够访问其外部函数局部变量的内部函数。每次闭包的成功调用后，这些外部的局部变量都能保存其值（状态），闭包凭此状态得知当前它在遍历过程中的位置。因为创建一个闭包也必须同时创建其外部的局部变量，所以创建一个闭包的过程中需要设计两个函数：一个是闭包函数本身，另一个是工厂函数，后者负责创建闭包。

作为一个简单的例子，我们为链表编制一个简单的迭代器，与 ipairs()不同的是，这个迭代器只返回元素的值，而不会反索引值：

```
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
```

这个例子中的 list_iter 是一个工厂函数，每次调用它，它都会创建一个新的闭包，也即迭代器本身。闭包记录着其外部的局部变量（t、i、n），因此每次调用它，它都返回链表中的下一个元素值，当链表中没有下一个值时，它将返回空值 nil。

我们可以在 while 语句中测试这个迭代器：

```
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
```

同时，这个迭代器也很容易被应用于范性 for 循环，因为它就是被设计成适合迭代操作：

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

范性 for 在循环过程中为迭代循环记录所有的状态：调用函数工厂生成迭代器、在循环内部保存迭代器（因此我们不需要像刚刚的 while 循环例子那样声明 iter 变量）、在需要取得下一个元素的时候调用迭代器函数、当迭代器返回空值时结束循环（后面我们将看到范性 for 实际上能完成更多的工作）。

下面介绍一个高级应用的例子：设计一个迭代器来遍历一个文件内所有的单词。我们需要保存当前所在的行以及在当前行的偏移量这两个值，以便实现遍历操作，有了这两个值，我们才能从中产生下一个单词，我们使用两个外部的局部变量 line、pos 来保存这两个值：

```
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
```

上述迭代函数的主体部分是对 string.find 函数的调用，该函数从当前行地当前位置开始查找一个完整的单词，例子中用模式“'%w+'”来匹配一个单词，意为“一个或多个连续的字母组合”。如果找到一个匹配单词，迭代函数更新当前位置 pos 为该单词后第一个字符的位置，并且返回这个单词，这一工作由于 string.sub 函数完成，它能够从一个字符串中提取两个位置之间的子串。若没有找到匹配的单词，迭代函数继续读取新行并重新搜索，如果没有新行可读则返回空值结束。

尽管迭代函数本身有点复杂，但使用起来却相当直观：

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

这就是迭代器的普遍状况：设计困难，而使用简便。这不是什么大问题，因为通常情况下，Lua 程序的最终用户不需要自己设计迭代函数，他们使用程序提供的迭代器。

7.2 范性for的语义

前面定义的迭代器有一个缺点，即每次循环都需要创建一个闭包。但大多数情况下，这不是真正的问题所在，例如在上述 allwords 函数中创建一个新闭包的代价比起读整个文件来说，简直微不足道。但有时候创建闭包付出的代价是不能忍受的，这时我们可以使用范性 for 本身来保存迭代信息。

前面我们看到在循环过程中，范性 for 会在内部保存迭代函数。实际上它保存了三个值：迭代函数、状态常量、控制变量。下面我们将研究一下细节的内容。

范性 for 的语法如下：

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

<var-list>是一个或多个以逗号分隔的变量的列表，<exp-list>是一个或多个以逗号分隔的表达式的列表。通常情况下 exp-list 只有一个表达式，即对迭代器工厂的调用，见下例：

```
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。经常地，变量列表中也只含有一个变量：

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

变量列表中的第一个变量就是所谓的控制变量，在循环过程中，它总是不为空值，因为其值一旦为空值，循环也就结束了。

我们分析一下范性 for 的执行过程：

第一，计算 in 后面的表达式的值，该表达式应该返回范性 for 所需要的三个值：迭代函数、状态常量和控制变量。类似多重赋值操作：只有最后的或唯一的表达式可以返回多个值；最终的返回值将被调整为三个，也就是说，多处的值会被舍弃，而不足的用空值补足。当我们使用比较简单的迭代器时，工厂函数只返回迭代器本身，因此状态常量和控制变量被赋予空值。

第二，经过第一步的初始化之后，for 语句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，状态常量没有任何用处，它仅仅在初始化时被赋值，并在调用迭代函数时作为参数被传递）。

第三，将迭代函数的返回值赋给变量列表。

第四，如果返回的第一个值为空值，则循环结束，否则继续执行循环体。

第五，回到第二步继续调用迭代函数。

更精确地说：

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

等价于下列代码：

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

```
end
```

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，那么控制变量将在循环中依次取得这些值：a1 = f(s, a0)、a2 = f(s, a1)、……（直到 ai 为空值）。如果 for 语句的参数列表还有其他变量，那么它们将取得 f 函数的额外返回值。

7.3 无状态迭代器

顾名思义，无状态迭代器是指那些自身不保留任何状态信息的迭代器，因此我们可以在循环中使用无状态迭代器，以避免创建闭包所花费的额外代价。

我们已经知道，每一次迭代，都是以迭代函数都是以状态常量和控制变量的值作为参数调用迭代函数。一个无状态迭代器仅仅利用这两个值获取下一个元素。一个典型的例子是就 ipairs 函数，它遍历数组的所有元素：

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

迭代过程中的状态信息包括：被遍历的表（也就是在循环过程中不会改变的状态常量）和当前索引（即控制变量），ipairs 函数和它返回迭代函数都很简单，可以用 Lua 这样实现它：

```
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 在 for 循环中调用 ipairs(a)时，它将获取三个值：作为迭代器的迭代函数 iter、状态常量 a、初始值为 0 的控制变量。之后 Lua 调用 iter(a, 0)返回 1 和 a[1]（除非 a[1]为空值）；第二次的迭代将调用 iter(a, 1)返回 2 和 a[2]……直到遇上第一个为空值得元素。

Lua 库中的 pairs 函数同样可以遍历一个表内的所有元素，除了其迭代函数为 next：

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

在调用 next(t, k)中，k 是表 t 的索引，该调用以随机的排序方式返回表中的下一个索引以及与此索引相关联的值。因此 next(t, nil)将返回第一个“索引-值”对，如果表中没有元素，该调用将返回空值。

你还可以选择不使用 pairs 函数，而是直接使用 next 函数：

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

注意，返回给 for 循环参数列表的结果会被调整为三个，因此 Lua 将获取 next、t 和 nil，与直接调用 pair(t)时的返回值一致。

7.4 多状态的迭代器

一个经常发生的情况是，迭代器需要保存更多的状态信息，而不只是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需的状态信息都封装到表内，之后将表作为迭代器的状态常量，使用表来保存数据可以让迭代器在循环过程中保存尽可能多的信息，并按照需要修改表内的数据，因此尽管状态信息一直都在同一个表内（也就是所谓的控制变量，或者不变量），但是表的内容却可以根据循环的需要进行修改。一个表足以让迭代器保存其所有状态信息，因此迭代器通常忽略由范型 for 语句传递的第二个参数（也就是所谓的控制变量，或者可变量）。

为了展示这种技术的应用，下面我们将重写 allwords 迭代器（它负责遍历一个文件中的所有单词），但是这次我们将使用带有两个域（分别是 line 和 pos 两个域）的表来保存状态信息。

用来启动迭代的函数非常简单，它的返回值是一个迭代函数以及迭代的初始状态：

```
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的：

```
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
```

只要条件允许，我们都应尽可能地使用无状态迭代器，以便循环的时候所有状态信息都有 for 语句来保存，而无需要每次进入循环都不得不创建新的对象。如果程序无法由无状态迭代器实现，那么应尽可能地使用闭包，一个典型的闭包代码除了看起来更加优雅之外，其执行效率也比使用表的迭代器高：首先，创建闭包的代价比创建表小，其次，存取 Upvalue 的速度也比存取表域快。稍后我们将看到另一种使用协同程序的迭代器，这是一种更强大，但代价也不小的解决方案。

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，迭代器只是为迭代陆续地提供值。因此一个更好的叫法可能应该是生成器（Generator），尽管如此，迭代器的称呼依然被沿用，是因为它早就在诸如 Java 之类的其他语言中确立了它的地位。

有一种方式可以创建真正完成迭代工作的迭代器。使用这类迭代器并不需要使用循环语句，而是调用迭代器，并将一个指示每次迭代需要处理的任务的参数传递给它即可。更具体地说，这类迭代器接受一个函数作为参数，而该函数会在迭代器内部被调用。

为了更具体的说明这一点，我们使用刚刚介绍过的技术来重写 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
```

上述 f 函数指明了每次迭代需要执行的工作，如果我们想要打印出每个单词，只需要以 print 函数为参数调用迭代器：

```
allwords(print)
```

就上述例子，我们可能更经常地使用一个匿名函数作为参数，下面的例子打印出“'hello'”这个单词在文件中出现的次数：

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

同样的工作，用先前介绍的方式（也就是 for 循环）完成也没有什么太大的不同：

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

真正的迭代器的写法在老版本的 Lua 中很常用，因为那时还没有 for 循环语句。它与“生成器”的方式有何不同？两种方式都有差不多的代价：每一步迭代都需要调用一个函数。一方面用真正的迭代器的方式更容易书写和理解（尽管用协同程序同样也有此优点），但另一方面，用生成器的方式更加灵活（在

真正的迭代器中，return 语句是从匿名函数中返回的，而不是从负责迭代的函数中返回）。

第 8 章 编译、运行和错误处理

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（在实践中，大部分解释型语言都这么做的）。在类似 Lua 这样的解释型语言中存在编译阶段听起来有点不合适，然而，解释型语言的特征在于编译器是语言运行时的一部分，而不在于代码是否被编译，正因为如此，执行经过编译的中间码速度会相当的快。可以说 dofile 函数的存在恰好说明了 Lua 是一种解释型的语言。

正如我们前面介绍过的，dofile 函数是运行 Lua 代码段的一种简单方式。dofile 函数实际上是一个辅助函数，而提供重要功能是 loadfile 函数，该函数与 dofile 一样将 Lua 代码段从文件载入，但是它并不执行代码，而是将源代码编译成中间码并将其作为一个函数返回。与 dofile 不同的另外一点是，loadfile 不会抛出错误信息，取而代之的是返回错误码以便处理错误。我们可以这样定义 dofile：

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

如果 loadfile 函数失败将导致 assert 函数抛出错误。

如果只是完成简单的功能，使用 dofile 函数会比较方便，只需要调用该函数就能完成载入和运行的工作。但 loadfile 更为灵活，在发生错误的情况下，loadfile 返回空值以及错误信息，这样我们可以自定义错误的处理方式。另外，如果我们多次运行一个文件的话，只需要调用 loadfile 一次，因此只需要编译一次就可多次运行，而 dofile 却需要调用多次，且每次都要重新编译。

loadstring 与 loadfile 相似，只不过它不是从文件中读入代码段，而是从一个字符串中读入。例如：

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

执行上述代码后，f 将成为一个函数，调用 f()时会执行 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。如其他函数相比它也是一个代价高昂的函数，并且会降低程序的可读性。因此在使用前，务必确认没有其他更简单的解决问题的方式。

Lua 把每一个独立的代码段都视为匿名函数。例如：对于代码段“a = 1”，可以使用 loadstring 返回与其等价的代码，即：

```
function()
    a = 1
end
```

与其他函数一样，代码段内部也可以定义局部变量和返回值：

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

loadfile 和 loadstring 都不会抛出错误，如果发生错误它们将返回空值和错误信息：

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

另外，loadfile 和 loadstring 都没有任何副作用，它们仅仅将代码段编译成为内部可识别的代码并返回结果，这个结果就是一个匿名函数。一个常见的误解是 loadfile 和 loadstring 定义了函数，但是在 Lua 中，函数定义是发生在运行时的赋值操作，而不是发生在编译时。假如我们有一个名为 foo.lua 的文件：

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

执行 f = loadfile("foo.lua")后，foo 函数已被编译但还没有被定义，因此必须运行该代码段来定义它：

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

如果你想粗略地实现载入并运行代码段，你可以通过直接调用 dostring 的返回值：

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

但是如果代码段内存在语法错误，那么 loadstring 函数将返回空值和诸如“attempt to call a nil value”的错误信息；为了返回更明确的错误信息，我们可以使用断言函数 assert：

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

通常使用 loadstring 加载一个字符串没有什么意义，比如下面的代码：

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

该代码粗略地等同于 f = function () i = i + 1 end，但是后者的速度更快，因为它只需要被编译一次，对于前者，每调用 loadstring 一次都会导致重新编译。虽然是等价，但是却不完全一样，因为 loadstring 编译的时候并不关心语法定界的问题，让我们对上述例子稍加修改：

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

就如所期望的那样，函数 g 存取得是局部变量。而函数 f 访问的却是全局变量 i，这是因为 loadstring 总是在全局环境中编译它的字符串。

loadstring 通常用于运行程序外部的代码，也就是说，它要运行的代码并不在程序内部。比如，你想在自己的程序里使用第三方提供的函数来进行绘图操作，这时候用户可以输入该函数地代码，而 loadstring 函数负责将其载入你的程序。注意，loadstring 载入的是一个代码段，即语句的集合，如果你想载入表达式，那么你需要使用 return 将其返回，这样你就得到了一个能够返回表达式值得语句。例子如下：

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

loadstring 返回的函数是一个普通地函数，因此可以多次被调用：

```
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
```

假设在一个产品级的程序内运行外部代码，你需要处理 loadstring 函数可能返回的所有错误。另外，如果外部代码不安全，你可能会想要在一个受保护的运行环境内运行它以避免不必要的副作用。

8.1 require函数

Lua 提供高级的 require 函数来加载和运行 Lua 库。粗略地说，require 和 dofile 这两个函数完成几乎一样的功能，除了两个重要的区别：

1、require 会在目录中搜索文件

2、require 会判断文件是否已经加载以避免重复的工作。

基于对上述特征的考虑，在 Lua 中使用 require 来加载库是一个比较明智的做法。

require 使用的路径和普通意义上的路径有些区别，大多数程序使用路径列表来搜索指定文件，但是 ANSI C 中并没有路径的概念。因此，require 所使用的路径是一个模式列表，每一个模式匹配提供一种可选的方式将虚拟文件名（即 require 函数的参数）转换成真正的文件名，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，然后检查是否存在这样的文件。如果不存在，就继续匹配第二个模式。路径列表中的多个模式由分号（这是一个在绝大多数操作系统中都很少用于文件名的字符）隔开。假设路径列表如下：

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

调用 require "lili"时，Lua 会试着打开下列可能存在的文件：

```
lili
lili.lua
c:\windows\lili
/usr/local/lua/lili/lili.lua
```

require 仅仅负责分号（模式之间的分隔符）和问号的处理，其他的信息（如目录分隔符、文件扩展名等）都由路径列表直接定义。

为了确定路径列表，require 函数首先检查 LUA_PATH 全局变量。如果它是一个字符串，则认为它就是路径列表，否则，require 继而检查 LUA_PATH 环境变量的值。如果两个都失败，require 将使用默认路径（这个默认值一般是“?;?.lua”，如果你自定义了 Lua 的编译，那么很容易就能修改这个默认值）。

require 的另一个功能是避免重复加载同一个文件，为此 Lua 保留一个包含所有已经加载的文件的列表，如果一个需要加载的文件在表中存在，则 require 只是简单地返回。该表中保留的是已加载文件的虚拟文件名，而不是真实的文件名，因此如果使用不同的虚拟文件名加载同一个文件，那么该文件将会被加载多次。比如 require "foo"和 require "foo.lua"（假设路径为列表为“?;?.lua”），那么 require 函数将会加载 foo.lua 两次。我们可以通过_LOADED 全局变量来访问上述表，查询该表可以判断某个文件是否已经被加载，利用该表我们可以“愚弄”require 函数以便多次运行同一个文件：执行 require "foo"之后_LOADED["foo"]必定不为空值，如果你将其设为空值，再次执行 require "foo.lua"将会再次加载该文件。

一个路径列表中的模式不一定包含问号，它可以包含一个指定的文件名，比如下列路径列表中最后一个模式：

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

在这种情况下，当 require 没有找到任何匹配的时候，就会加载此文件（当然这个指定的文件名必须放在路径列表的最后才有意义）。在 require 运行一个代码段之前，它将定义一个名为

_REQUIREDNAME 的全局变量用来保存被加载的文件的虚拟文件名。我们可以通过使用这个特性来扩展 require 的功能，举个极端的例子，我们可以把路径列表设为“/usr/local/lua/newrequire.lua”，这样以后每次调用 require 都会运行 newrequire.lua，这种情况下将使用_REQUIREDNAME 值去实际加载所需的文件。

8.2 用C编写的Lua包

Lua 和 C 语言之间很容易交互，同时使用 C 语言设计 Lua 包也很容易。与由 Lua 写的包不同，用 C 编写的 Lua 包在使用以前必须首先加载并与执行 Lua 程序的应用程序连接，在大多数操作系统中，最简单的方式就是通过动态连接库机制，然而动态连接库并没有包含在 ANSI C 的规范中，也就是说，这种实现方式缺乏可移植性。

通常 Lua 不包含任何不能用 ANSI C 实现的特性，动态连接是一个特例。我们可以将动态连接机制是所有其他机制的基础：一旦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，只在这一特例中，Lua 打破了其平台兼容的原则，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 和其他类 Unix 的平台实现了这种机制，扩展其他平台支持这种机制并不是很难。在 Lua 提示符下执行 print(loadlib())，如果提示诸如“bad arguments”，则说明你的发行版支持动态连接机制，否则说明动态连接机制未被支持或没有安装。

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

```
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将其连接到 Lua，然而它并不打开库（也就是说没有调用初始化函数），与之相反，它将初始化函数作为一个 Lua 函数返回，这样我们就可以直接在 Lua 中调用它。如果加载动态库或者查找初始化函数时出错，loadlib 将返回空值以及相应的错误信息。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
```

通常我们可以期望发行版的库中包含着一个与前面代码段相似的存根程序（Stub），这里存根程序就是指，负责测试或者调用库的程序。安装实际二进制共享库的时候，首先将其置于某个目录下、修改存根程序以对应库的实际路径、并将存根程序的路径列入 LUA_APTH。这样设置完毕后，我们就可以随时使用 require 函数将存根程序打开，以便载入 C 语言编写的 Lua 库。

8.3 错误

Errare humanum est（拉丁谚语：人非圣贤，孰能无过）。因此我们必须尽可能地处理好可能会发生的种种错误。作为扩展性语言的 Lua，经常被嵌入在其他应用程序中，因此当错误发生时不可能只是简单地让程序崩溃或者退出，相反地，Lua 退出当前的代码段并返回到应用程序中。

当 Lua 遇到不期望的情况时就会抛出错误，比如：两个非数字的相加、调用一个非函数的变量、通过索引访问非表类型的变量等等（可以通过 metatables 改变这种行为模式，后面介绍）。调用 error 函数也可以显式地抛出错误，该函数的参数是字符串型的错误信息。通常，此函数能很好地在你的代码中处理错误：

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

类似上例的条件错误语法是如此的常用，因此 Lua 提供了专门预定义函数 assert 来完成上述功能：

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

assert 首先检查第一个参数，如果为真，则返回第一个参数；否则（该参数为 false 或 nil），assert 抛出错误。作为错误信息的第二个参数是可选的，如果你不需要自定义错误信息，你可以忽略此参数。但是注意请 assert 函数是一个普通的函数，在 Lua 中会首先计算两个参数的值，之后才调用函数。因此下列代码中的字符串连接运算总是会被执行，不管 n 是否为数字：

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

当一个函数发现异常时，它可以假定两种基本的处理方式：返回错误代码，或者调用 error 函数抛出错误。选择哪一种处理方式，没有固定的规则，不过有一条可供参考的基本原则：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，使用一个表作为参数来调用它，该如何处理错误？假设用返回错误代码的方式，如果我们需要检查该错误，那么代码类似这样：

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

但是，如果在调用 sin 函数前判断此异常也同样容易：

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

通常我们既不是检查参数，也不是检查返回结果，因为如果参数错误，那可能意味着我们的程序某个地方有问题，这种情况下，处理异常最简单有效的方式终止代码的运行并抛出错误。

再来考察一下 io.open 函数，它用于打开一个文件。如果文件不存在，它应该如何应对此错误呢？在此情况下，在调用函数前判断异常是否发生并不那么容易，因此在很多系统中，我们通过尝试打开文件来判断文件是否存在。如果由于文件不存在或用户没有访问权限而导致 io.open 无法打开文件，函数将返回空值和字符串型的错误信息。因此在此方式下，我们可以通过询问用户是否打开另外一个文件，来合理地处理该问题：

```
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
```

如果你并不想处理这些错误，但又希望程序正常运行，那么你可以借助 assert 函数来监督代码：

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

这是一个经典的 Lua 习语：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 的第二个返回值（错误信息）将成为 assert 函数的第二个参数。

8.4 错误处理和异常

对多数应用程序来说，错误处理并不需要在 Lua 中进行，应用程序本身会处理各类错误。通常应用程序调用 Lua，并要求 Lua 运行某个代码段，如果此时发生错误，应用会根据 Lua 返回的错误代码采取适当地措施。如果是在独立的 Lua 解释器中出现错误，解释器的主循环只是打印一串错误信息、显示 Lua 提示符并等待后续的命令。

如果你需要在 Lua 中处理错误，你应该使用 pcall（Protected Call）函数来封装你的代码。

假定你想运行一段 Lua 代码，并且捕获代码运行中任何可能的错误，首先需要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
```

然后，使用 pcall 调用上述 foo 函数：

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

当然，也可以使用 pcall 调用匿名函数：

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

pcall 函数在保护模式（Protected Mode）调用它的第一个参数（这个参数是一个函数），同时在运行中捕获所有的错误。如果运行中没有出错，pcall 返回 true 以及被调用函数的返回值，否则，返回空值和相关错误信息。

不要被“错误信息”这个名字所迷惑，错误信息不一定非得是字符串类型，传递给 error 的任何参数值都会被 pcall 返回：

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

```
print(err.code)          --> 121
```

这种机制提供的能力足以应付 Lua 中的各种异常处理。我们通过 error 抛出异常，并使用 pcall 捕获将其捕获。其中的错误信息表明了错误的类型。

8.5 错误信息和错误跟踪（Traceback）

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错误的细节。如果遇到内部错误（例如访问一个非表类型的变量的索引），错误信息将由 Lua 自动生成，否则错误信息由传递给 error 函数的参数决定。不管是何种情况，Lua 都会尝试在错误信息中加入有关出错位置的信息。

```
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程序的文件名（例子中为 stdin）与行号（例子中为 1）。

函数 error 还可以有第二个可选的参数，用来表示报告错误的层次，这个层次指明了错误的来源：到底是程序本身，还是调用此程序的程序，或者更高的调用层次。比如，你写了一个函数，该函数的第一个任务就是检查其参数类型是否正确：

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

如果有人调用上述函数的时候使用了一个错误的参数：

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

Lua 会指出错误来源的是 foo（因为出错的时候调用 error 的就是 foo 函数），而不是真正导致错误的凶手。为了纠正这个问题，你可以通知 error 函数，错误发生在第二层，也就是第一个调用者的位置（被调用的函数是第一层）：

```
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调试信息，而不仅仅是错误发生的位置。至少，我们想要错误跟踪的信息，该信息能显示错误发生时栈内的全面信息，但当 pcall 返回错误信息时，已经释放了部分栈信息，因此我们必须在 pcall 返回建立错误跟踪信息。Lua 提供了 xpcall 来满足此需要，xpcall 接受两个参数：调用函数以及错误处理函数。当错误发生时，Lua 会在栈被释放以前调用错误处理函数，

好让错误处理函数能够利用 Debug 库收集所需的错误信息。有两个常用的错误处理函数：debug.debug 和 debug.traceback，前者给出 Lua 的提示符，你可以自己动手检查错误发生的原因；后者可以创建更细致的错误跟踪信息。独立的 Lua 解释器用来构建错误信息的函数就是 debug.traceback。你可以在任何时候调用 debug.traceback 获取当前运行状况的跟踪信息：

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

第 9 章 协同程序

协同程序（Coroutine）类似于多线程情况下的线程：是一种程序的运行绪，它有自己的栈，自己的局部变量，自己的当前指令指针（Instruction Pointer），但与其他协同程序共享全局变量以几乎所有其他资源。线程和协同程序的主要区别在于：在概念上讲，在多处理器情况下，拥有多线程的程序并发运行多个线程；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，一个正在运行的协同程序只在它显式要求被挂起时才会被挂起。

协同程序是一个功能强大的概念，但是其应用却有点复杂。如果第一次阅读本章的时候你没有完全理解所有的例子，可以先进入后续章节的阅读，之后再回到本章的内容。

9.1 协同的基础

Lua 所有与协同程序有关的函数都在 coroutine 表中。用于创建新的协同程序的 create 函数，只有一个参数：一个函数，即协同程序将要执行的代码。若一切顺利，create 函数将返回一个 thread 类型的值，这就是一个新创建的协同程序。通常情况下，create 的参数是一个匿名函数：

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

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

协同程序有三个状态：挂起（Suspended）、运行（Running）、停止（Dead）。当我们成功创建协同程序时，其初始状态为挂起，即此时协同程序并不自动运行其代码。可以用 status 函数测试协同程序所处的状态：

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

函数 coroutine.resume 通过使协同程序的状态由挂起变为运行，可以启动或重新启动一个协同程序，或者说唤醒一个协同程序：

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

本例中，协同程序打印出“hi”并终止程序，之后便进入无法再次被唤醒的终止状态：

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

当目前为止，协同看起来只是一种复杂的调用函数的方式，真正的强大之处体现在 yield 函数，它可以将正在运行的代码挂起以便再次被唤醒。先看一个简单的例子：

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

当我们唤醒这个协同程序，它将开始执行代码，直至遇到第一个 yield 处，这时它被挂起：

```
coroutine.resume(co)            --> co    1
```

如果这时检查其状态，我们可以看到这个协同程序已经被挂起，因此它能够再次被唤醒：

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

站在协同程序的角度来看：当协同程序被挂起的时候，它将进入 yield 函数的内部循环，当我们唤醒被挂起的协同程序时，一开始对 yield 的调用指令将返回（即 yield 函数退出其内部循环），继而执行 yield 调用之后的代码，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
```

上例中最后一次调用 resume 函数时，协同程序已完成循环并返回，因此协同程序处于终止状态。如果仍然试图唤醒它，那么它将返回空值和相应的错误信息：

```
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 并不会抛出错误信息，而是将将其返回给 resume 函数。

一个有用的 Lua 应用是利用成对出现的 resume 和 yield 函数来相互（双方分别为调用 resume 和 yield 的代码段）交换数据。第一个例子中只有 resume，没有 yield，resume 函数接受额外的信息作为协同程序的参数：

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

第二个例子中，调用 resume 后得到的返回值包括：用以指示协同程序无错运行的 true 值，和传递给对应的 yield 函数的所有参数：

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

同样地，传递给对应的 resume 函数的参数，也会被传递给 yield 函数：

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

最终，当协同程序结束时，其所有的返回值，也会传给对应的 resume 函数：

```
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，但每一种都有其用处。

现在已大体了解了协同程序的基础内容，在我们继续学习之前，先澄清某些概念。Lua 的协同被称为非对称协同程序（Asymmetric Coroutines），指挂起一个正在运行的协同程序的函数与唤醒一个被挂起的协同程序的函数是不同的，而有些语言提供了对称协同程序（Symmetric Coroutines），它们使用一个函数负责分配协同程序的程序运行控制权。

有人称非对称的协同程序为“半协同程序”（因为它们并非对称，也并不是真正的合作关系），而另一些人使用同样的术语表示“受限制的协同程序”，这意味着：一个受限制的协同程序只有运行在任何非其他函数内部的时候才能将自己挂起（放弃程序运行控制权），也就是说，在该协同程序的运行栈内部没有后续调用的时候才能放弃控制权。换句话说，只有在半协同程序的内部才能使用 yield 函数将自己挂起，Python 中的生成器（Generator）就是这种类型的半协同程序。

与对称协同和非对称协同的区别不同的是，协同程序与生成器的区别更大：生成器相对比较简单，它不能完成真正的协同程序所能完成的一些任务。Lua 提供了真正的非对称的协同程序，而且以非对称的协同机制为基础可以很容易地实现对称的协同程序：每一次程序运行控制权的转移都在执行 resume 之后接着执行 yield 函数将自己挂起。

9.2 管道和过滤器

协同程序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，从一个文件中读取），另一个函数不断地“消耗”掉这些数据（比如，将其写入另一文件中）。下面是典型的生产者以及消费者函数：

```
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
```

例子中生产者在不断地循环生产，而消费者在不断地循环消费，稍加修改令它们在没有数据的时候停下来并不困难。这里的问题在于如何协调 receive 和 send，这是一个典型的“谁该拥有主循环”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，都认为随时可以使用另一方提供的服务。对于这种特殊的情况，可以改变一个函数的结构来解除其循环，使其作处于被动接受的状态。然而在实际情况下可能很难调整这种结构。

协同程序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“唤醒-挂起”（resume-yield）关系会不断颠倒。当一个协同程序调用 yield 时并不会进入一个新的函数，而是进入等待状态直到一个随后而来的 resume 调用（yield 将其返回值返回给一个将其唤醒的 resume 函数），与之相似，调用 resume 时也不会开始一个新的函数，而是将协同程序的运行控制权返回给下一个由协同程序发出的 yield 调用。这种性质正是我们所需要的，与使得 send 与 receive 之间的关系如同主仆关系。receive 将唤醒生产者以便生产一个可供消耗的新值，而 send 将其所在地生产者挂起，并把生产的新值交给消费者。

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

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

当然，现在生产者必须是一个协同程序：

```
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，当消费者需要值时它将唤醒生产者，生产者生产并交出所生产的新值之后停止生产直到消费者再次将其唤醒。我们称这种运行方式为“由消费者驱动的模式”。

我们可以使用过滤器扩展这个设计，过滤器位于在生产者与消费者之间，对数据进行某些转换处理。过滤器既是生产者又是消费者，它唤醒生产者生产新，将新值进行转换处理后交给消费者。接下来我们将修改上述代码加入一个过滤器（该过滤器给每个新值前加上行号），下列代码可供参考：

```
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
```

最后需要做的仅仅是，创建所需要的组件（生产者）、将其与过滤器连接并启动消费者：

```
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地想到 UNIX 的管道，协同程序是一种非抢占式的多线程。管道方式下，每个任务在独立的进程中运行，而在协同程序方式下，每个任务运行于独立的协同程序中。管道在写（生产者）与读（消费者）之间提供了一个缓冲，因此两者之间的相对速度并不影响正常的读写这在管道的上下文中这是非常重要的，因为在进程间的进行切换（由于读写速度不一样，经常需要从速度快的进程切换到速度慢的进程以防止慢速的进程被淹没）的代价是很高的。对于协同程序的方式，任务间的切换代价要小很多，其代价几乎等同于函数调用的代价，因此程序可以在读写之间进行无间的切换。

9.3 用作迭代器的协同程序

用于循环结构的迭代器可以看作是“生产者-消费者”模式的一个具体例子，迭代器为循环体不断地生产新值，因此使用协同程序来实现迭代器显得十分合适。实际上，协同程序的确很适合用于此类应用，其关键特性是允许协同程序在调用者和被调用者两种角色之间不断地交替，正是由于这种特性，我们可以无需再考虑如何为迭代器保存其状态信息。

为了说明协同程序的这类应用，我们将设计一个迭代器，它能够列举一个数组中元素的所有排列方式。直接写出这样一个迭代器并不是那么容易，但是设计一能够产生所有排列方式的个递归函数并不难，思路如下：遍历数组元素，每遍历一个元素便将其放到最后的位置；以此方式递归产生剩余元素的所有排列方式。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
```

为了执行代码，还需要定义一个合适的 printResult 函数并使用恰当的参数来调用 permgen：

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

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

有了上述生成器后，就可以将其转换成一个迭代器了。首先，用 yield 替换 printResult：

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

然后，我们定义一个函数工厂，使生成器可以运行在一个协同程序内，并创建迭代器。迭代器负责唤醒协同程序来生成下一个排列方式：

```
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
```

有了上述结构，就很容易使用 for 语句遍历一个数组的所有排列了：

```
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
```

函数 perm 使用了 Lua 中常用的模式：将对 resume 的调用与协同程序一起封装在一个函数内。这种模式在 Lua 中非常常见，因此 Lua 为此提供了一个专用的 coroutine.wrap 函数。与 create 相同的是，

wrap 创建一个协同程序；与之不同的是，wrap 并不返回协同程序本身，而是返回一个函数（通过调用此函数可以唤醒协同程序）。另一个与 resume 不同的地方是，wrap 并不会返回错误代码作为第一返回值，而当错误发生时便抛出错误。可以使用 wrap 重写 perm 函数：

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

一般情况下，使用 coroutine.wrap 比使用 coroutine.create 更加简单直观，因为它刚好为我们提供了所需要的功能：一个可以唤醒协同程序的函数。但是相比而言，它缺乏一定的灵活性：我们无法查询一个由 wrap 创建的协同程序的状态，更重要的是，我们无法通过错误代码检查错误的发生。

9.4 非抢占式的多线程

如前面所见，Lua 中的协同程序是一合作式的多线程，每一个协同程序等同于一个线程，线程的“挂起-唤醒”状态可以在彼此之间切换。然而与真正的多线程不同的是，协同程序是非抢占式的，也就是说，当一个协同程序正在运行时，是无法在外部终止它的，唯一能够将其挂起的方式就是它本身调用 yield 将自己挂起，与想像中的不同，这对于很多应用程序根本不是问题，不使用抢占式状态切换的程序设计更加容易。因此，根本不需要线程间同步的问题，因为线程间的同步都是显而易见的，你只需要保证协同程序仅在临界区外将自己挂起即可。

然而，对非抢占式多线程来说，任何时候只要有一个线程调用一个阻塞操作（比如将自己挂起），整个程序在阻塞操作完成之前将处于停止状态。对大部分应用程序而言，这是无法忍受的，很多程序员因此而放弃了使用协同程序来实现多线程。下面我们将看到这个问题的一个有趣的解决方案。

看一个典型的多线程例子：通过 HTTP 协议从远程主机上下载一些文件。在下载多个文件之前，首选需要了解如何下载一个远程文件。在此例子中，我们将使用 Diego Nehab 开发的 LuaSocket 库来完成。为了下载一个文件，首先必须建立一个到文件所在服务器的连接，向服务器发送一个下载请求，在请求得到许可之后接收服务器上的文件，最后关闭网络连接。在 Lua 中，我们可以用下面的方式来完成这个下载任务。首先，必须加载 LuaSocket 库：

```
require "luasocket"
```

接着还要确定服务器地址以及需要下载的文件。在这个例子中，我们将从 W3C 的站点下载 HTML 3.2 的规格参考书：

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

然后，打开一个到指定站点 80（HTTP 服务的标准端口）端口的 TCP 连接：

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

上述指令返回一个连接对象，我们可以使用这个连接对象向服务器发送文件请求：

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

receive 函数总是返回它接收到的数据以及一个表示操作状态的字符串。当主机断开连接时，我们也就退出了数据接收的循环。

最后，我们需要关闭上述 TCP 连接：

```
c:close()
```

既然我们已经了解如何下载单个文件，下面回到如何下载多个文件这个问题上来。一种简单的方法

是每次下载一个文件直到所有文件都被下载，但是这种依次下载的方式速度太慢：必须在一个文件下载完之后才能开始下载后续的文件，而在一个请求发送之后绝大部分时间都是在等待数据的到达，也就是说，大部分时间都花费在 receive 函数的运行上（此时 receive 处于等待数据接收的阻塞状态）。如果可以同时下载多个文件，速度将得到很大的提高：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，协同程序为这种并发下载提供了便利的支持，我们为每一个下载任务创建一个线程，当某一线程没有数据到达时，它将控制权交给一个分配器，分配器将唤醒其他线程尝试读取数据。

为了使用协同程序的方式，首先需要将上述下载文件的代码封装到一个函数内：

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

文件内容为何无关大体，因此上述函数只是计算文件尺寸，而没有将文件内容显示到标准输出上（多个线程读取多个文件时，输出会混杂在一起）。在新的代码中，我们使用一个辅助 receive 通过远程连接来接收数据，采用依次接收数据的方式时，代码如下：

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

如果采用并发接收数据的方式，函数在接收数据时不能被阻塞，在没有数据可取应该自行挂起，其代码如下：

```
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
```

执行 timeout(0)使得该连接上的任何操作都不会被阻塞，当返回的操作状态为“timeout”时意味着操作超时返回，此时，线程自行挂起。使用非假值作为 yield 的参数将通知分配器仍有线程在执行任务（稍后还将看到需要超时连接的分配器）。注意即使是超时返回，该连接依然返回它已经接收的数据，因此 receive 总是返回 s 给它的调用者。

下面的函数保证每一个下载任务都运行于独立的线程内：

```
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
```

代码中的 threads 表为分配器保存着所有处于活动状态的线程。

分配器的代码很简单，它遍历并调用每一个线程。当然，它也必须在线程完成其任务后将其从任务列表内移除，当任务列表为空时它将终止循环：

```
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，需要在主程序中创建所需的线程并调用分配器，假设需要从 W3C 下载四个文件，主程序如下：

```
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
```

采用协同程序的方式，在我的机器上花费了 6 秒下载完这四个文件，而依次下载的方式花费了 15 秒，超过两倍的时间。

尽管新的方式速度提高了，但还不是很理想，这种方式在至少有一个线程有数据可读取时，运行得还不错，如果所有线程都处于等待数据的状态时，分配器将进入忙等待状态，逐个检查各个线程，结果发现每一个线程都没有任何数据可读，由此导致了，协同程序实现的代码比依次下载的方式多花费了近 30 倍的 CPU 资源。

为了避免这种情况，我们可以使用 LuaSocket 库中的 select 函数，在程序能够从阻塞状态向唤醒状态切换前，一直处于被阻塞状态，也就是说，在程序无法摆脱阻塞状态时，不会花费 CPU 时间来查询其状态。为了实现这种能力，只需要稍稍修改已有的分配器即可，新版本的分配器代码如下：

```
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
```

在内层循环中，分配器将所有超时的连接记录到 connections 表，记住 receive 将连接传递给 yield，因此 resume 才能够将其返回。当所有的连接都超时的时候，分配器调用 select 函数，等待其中任何一个连接的状态改变。修改版本的运行速度与最初的版本相当，另外，因为它并不忙等待处于阻塞状态的连接，因此使用的 CPU 的也只是比依次下载的方式多了少许。

第 10 章 完整的示例

本章将通过两个完整的例子来结束 Lua 语言方面的介绍，期间你会看到 Lua 的多种应用方式。第一个例子来自于 Lua 官方站，它展示了 Lua 在数据描述方面的应用。第二个例子为马尔可夫链算法的一个实现，该算法的描述可以参考由 Kernighan 和 Pike 合著的《Practice of Programming》（由 Addison-Wesley 出版社于 1999 年出版）一书。

10.1 数据描述

Lua 官方站的数据库中保存着一些 Lua 项目的信息，数据库中的每个项目记录都由一个构造器以自动归档的方式表示，就如下面的例子：

```
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.]]
}
```

实际上，上述项目记录在 Lua 中是一个函数调用，函数名为 entry，参数是用一对括起来的 Lua 表，因为此调用此函数时只使用了一个参数且该参数类型为表，所以参数两侧的花括号可以省略。如果一个文件内包含了若干上述项目记录，就等于是包含了一系列的 entry 函数调用。

程序最终需要将项目记录写入一个 HTML 文档，就像 http://www.lua.org/uses.html 这静态 HTML 页面一样。由于项目数量过众，因此最终的 HTML 文档只显式项目记录的标题，之后再显示每个项目的明细。程序的输出结果如下所示：

```
<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 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.<P>
Contact: Waldemar Celes

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

</BODY></HTML>
```

要读取数据到 HTML 文档，程序需要做的就是正确地定义 entry 函数，然后使用 dofile 函数载入并执行包含项目记录的数据文件。需要注意的是，我们需要遍历所有的项目记录两次，第一次获取标题，第二次获取每个项目的描述信息。程序的第一种实现方式是：将所有的项目记录放在一个数组内，之后只需要遍历数组即可。因为 Lua 的编译速度很快，所以可以采用另一种更吸引人的实现方式：定义两个不同的 entry 函数，使用这两个不同的函数运行数据文件两次。接下来的讨论将遵循第二种实现方式。

首先需要定义一个辅助函数用来输出格式化的文本（在 5.2 章节中我们已经介绍过这个函数）：

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

因为最终输出的 HTML 文档头部是不变得，因此定义 BEGIN 函数来输出 HTML 文档的头部：

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

```
end
```

函数 entry 的第一个定义将在 HTML 文档的列表标记中输出项目标题。参数 o 即用以描述项目信息的表：

```
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 为空值（即表的 title 域不存在），就用的“no title”表示。

函数 entry 的第二个定义将输出项目记录中的所有相关信息。这可能有点复杂，因为记录中的各项信息都是可选的（即记录中的某些信息可能没有提供）。

```
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
```

HTML 文档中使用了双引号，为了避免冲突，我们在程序中使用单引号来表示字符串。下面的函数输出 HTML 文档的尾部：

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

在最终的程序中，BEGIN 函数首先输出 HTML 文档的头部，entry 函数的第一个定义输出项目标题，entry 函数的第二个定义输出项目的其他信息，最后，END 函数输出文档的尾部（假设包含项目记录的数据文件其文件名为 db.lua）：。

```
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

本章的第二个例子是马尔可夫链算法（Markov Chain Algorithm）的实现，我们的程序以前 n 个单词串为基础随机产生一个文本串，在此，n 取值为 2。

程序的第一部分读出原文，并依此建立一个表，这个表依次在原文中取出两个单词作为索引，而将这两个单词的后续单词所组成的表作为与该索引对应的值。建表完成后，这个程序利用这张表生成一个随机的文本，该随机文本中的每个单词都跟随着它的前两个单词，每个单词在原文中有相同的出现几率。这样，就产生了一个非常随机，但又不完全随机的文本。假如该程序将本书英文内容作为原文输入，则输出的结果有可能是“Constructors can also traverse a table constructor, then the parentheses in the following line does the whole file in a field n to store the contents of each function, but to show its only argument. If you want to find the maximum element in an array can return both the maximum value and continues showing the prompt and running the code. The following words are reserved and cannot be used to convert between degrees and radians.”

我们编写如下的函数，用于生成任意一个单词的前缀，这个前缀由该单词的前两个单词组成，两个单词之间用空格连接：

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

我们用 NOWORD（即“\n”）初始化原文的前缀，同时用其作为原文末尾的标记。例如：

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

用上述原文可以生成下面的表，等号后表中的值对应的索引值（即等号前的索引）就是它在原文中紧随其后的两个单词，因此它在表中出现的次数即它在原文中出现的次数（也就是上文所说的：它在原文中出现的几率）：

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

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

程序使用全局变量 statetab 来保存上述表。为了在该表中插入索引及其对应的值，我们需要用到下面的函数：

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

上述函数首先检查指定的前缀索引是否已生成对应的表，如果没有，则创建一个新前缀索引并赋予新值，如果已有，则调用 table.insert 函数将新值插入到列表尾部。

为了创建 statetab 表，我们需要使用 w1 和 w2 两个变量来保存最后读取的两个单词。对于每一个前缀，我们像上述由原文生成的表那样，保存一个紧随其后的单词的列表。

当表生成之后，程序将着手生成一个含有 MAXGEN 个单词的文本串。首先，程序需要重新初始化 w1 和 w2 变量，然后，对于每一个前缀，在其紧随其后的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 Lua表与对象

第 11 章 数据结构

表是 Lua 中唯一的数据结构。其他语言所提供的数据结构，如：数组、记录、链表、队列、集合等，在 Lua 都可以通过表来实现，不仅能够实现，而且实现的效率相当高。

在传统语言（C 或 Pascal 语言）中，我们使用数组和链表（链表表示为由指针连接的记录）来实现绝大部分数据结构。尽管可以用 Lua 表来实现数组和链表（有时我们需要这种实现），但是 Lua 表比数组或链表更加强大，很多算法使用了 Lua 表之后都变得异常简单。比如，你几乎不需要在 Lua 中编写搜索算法的代码，因为 Lua 表可以让你直接访问任何类型。

我们需要花一些时间去学习如何有效地使用表，下面将通过一些例子来说明如何利用表来实现一些典型的数据结构，以及它们的用法。首先，从数组和链表开始，之所以这样，并不是因为需要它们来实现其他数据结构，而是因为大家对它们比较熟悉。虽然在前面有关语言的部分，已接触到表的一些内容，在此，我们将对其重复阐述以便给出一个完整的体系。

11.1 数组

使用整数索引 Lua 表就实现了数组。数组大小并不固定，但可以根据需要增长，通常当我们初始化数组时，就间接地定义了数组的大小，下面的代码能够说明这一点：

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

访问由上述代码建立的数组，一旦访问 1-1000 以外下标的值，将得到空值，而不是 0。数组下标可以从 0、1 或任意值开始：

```
-- 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 中数组的下标从 1 开始。Lua 的标准库都遵循此惯例，因此只有数组下标从 1 开始，才可以使用标准库中与数组相关的函数。

我们可以在表构造器中创建并初始化数组：

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

这种构造器可以随心所欲地扩充，甚至多到容纳几百万的元素。

11.2 矩阵和多维数组

在 Lua 中表示矩阵的方法有两种。第一，数组的数组，也就是说，表的每个元素都是表。下面的例子创建了一个 N 行 M 列的 0 值矩阵：

```
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
```

Lua 表是对象，因此矩阵中的每一行都必须显式地创建。当然，对于 C 或 Pascal 中只需要简单地声明整个矩阵来说，这显得有点繁琐，但另一方面，这也提供了更大的灵活性。前面的例子稍加修改就可以得到一个三角矩阵，将下面这一行：

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

替换成：

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

以这种方式实现的三角矩阵比起常规做法，可以节省一半的内存消耗。

第二种表示矩阵的方法是，将上述两个下标合而为一。如果两个下标都是整数，可以将第一个下乘以一个常数再加上第二个下标得到合而为一的下标。下面是以这种方式创建 N 行 M 列矩阵的例子：

```
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，可用一个字符将两个字符串索引连接成一个单一的索引下标，例如，你可以用 m[s..':'..t]来索引一个矩阵 m 的元素，其中 s 和 t 为不包含冒号的字符串，如果 s 或 t 包含冒号将导致冲突，比如("a:", "b")和("a", ":b")将变成单个索引“a::b”。如果对此抱怀疑态度，你可以使用控制字符（转移字符）“\0”来连接两个索引。

实际应用中常常会用到稀疏矩阵，稀疏矩阵是指大部分元素都为 0 或空值的矩阵。例如，我们通过图的邻接矩阵来存储图，当且仅当 m、n 两个节点有 x 长度的连接时，矩阵的(m, n)位置才有 x 值，如果 m、n 节点点没有连接，则矩阵地(m, n)位置为空值。如果一个图有 10000 个节点，大致每个节点有 5 条边（从自身到四邻以及自身），为了存储这个图需要一个行列分别为 10000 的矩阵，总计 100000000 个元素，实际上大约只有 50000 个元素为非空值（每行有五列非空，对应于每个节点到四邻以及自身的五个连接）。很多数据结构的书上都详细讨论如何才能用少量的内存实现稀疏矩阵，但是在 Lua 中几乎用不到这些技术，因为以表实现的数组天生的就具有稀疏特性。如果用第一种方式来表示矩阵，那么总共需要 10000 个表，每个表大约五个元素，共需要 50000 个元素；如果用第二种方式来表示矩阵，那么只需要一张大约 50000 个元素的表。不管用那种方式，只需要存储那些非空值的元素。

11.3 链表

由于 Lua 表中的元素是动态的，因此使用表很容易实现链表，其中每一个节点是一个表，表域则代表指向另一个节点的指针。要实现一个只有两个域（值和指针）的链表，我们需要一个变量作为链表的根节点：

```
list = nil
```

想要在链表的头部插入一个值为 v 的节点，我们可以这么做：

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

如果想遍历这个链表，可以这样做：

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

其他类型的链表，如双向链表或循环链表，其实现方式也相当简单。但是，你会发现在 Lua 中很少用到这类数据结构，因为经常都存在一个更简单的方式来表示你的数据结构，而不是用链表。例如，可以用一个无边界限制的数组来表示栈，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 table 库提供的 insert 和 remove 函数来实现队列，但这种方式实现的队列针在需要处理大量数据时效率太低。更有效的方式是使用两个索引，一个用于第一个元素，另一个用于最后一个元素：

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

为了避免全局名字空间的污染，我们将所有操作置于一个表中，并称其为一个链表。因此新的代码看起来像下面这样：

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

之后，不管是在队列两端插入还是删除元素，操作都可以在常量时间内完成：

```
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
```

如果我们遵循严格意义上的队列的规范，只调用 pushright 和 popleft 操作，那么 first 和 last 都将持续地增长。但是实际上，我们用 Lua 表来表示数组，因此可以用 1 到 20 来索引数组，也可以用 16777216 到 16777236 来索引数组。另外，La 使用了双精度数来表示数字，因此假定每秒钟可以执行 100 万次插入操作，那么在数值溢出之前你的程序足可以运行 200 年。

11.5 集合和包

假定要列出一段源代码中所有的标识符，你得通过某种方法过滤掉语言本身的保留字，一些 C 程序员喜欢用一个字符串数组来表示这个保留字的集合，然后搜索这个数组来决定某个给定的单词是否在这个集合内。为了优化查询速度，C 程序员们甚至用二叉树或者哈希表来表示这个集合。

在 Lua 中表示这个集合有一个简单有效的方法：将集合中所有的元素作为索引存放在一个 Lua 表中，这样在查询给定元素是否在表中的时候，只需要测试该以该元素为索引的元素值是否为空值。在我们的例子中，我们可以这样写：

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

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

由于 while 是 Lua 的保留字，无法作为普通标识符，因此，我们不能写作 while = 1，而应该写成 ["while"] = 1。

还可以使用一个清晰明了的辅助函数来建立这个集合：

```
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 带缓冲区的字符串

如果你需要处理众多字符串的连接运算，比如从一个文件中逐行读入字符串，并将其连接成一个单个字符串。你的代码可能会像下面这样：

```
-- WARNING: bad code ahead!!
local buff = ""
for line in io.lines() do
    buff = buff .. line .. "\n"
end
```

尽管这段代码看起来毫无问题，然而在在 Lua 中处理大文件时，它将显露出其性能上的缺陷。它可能会在一个 350KB 的文件上花费掉近一分钟的时候（这就是为什么 Lua 专门提供了 io.read(*all)这个选择，它读取同样的文件只需要 0.02 秒）。

是什么原因导致了上述程序的无效率呢？实际上 Lua 使用了真正的垃圾收集算法，当发现程序使用了太多的内存，它就会遍历所有的数据并释放那些不再被需要的数据（即垃圾数据）。这个算法通常都有很好的性能（这也说明了 Lua 的快速并非偶然），但是上述代码中的循环令垃圾收集算法失去了其应有的高效。

为了弄清楚到底发生了什么，让我们先假定程序目前仍然在执行循环，变量 buff 已经含有一个 50KB 字节的字符串，且文件的每一行大小为 20 字节，当 Lua 试图将 buff 与新读取的行进行连接时（即执行 buff..line.."\n"时），她将创建一个大小为 50,020 字节的新字符串，并且将 buff 中的 50KB 字节的字符串拷贝到新字符串中。也就是说，每读取一行，Lua 就需要移动 50KB 字节的内存，并且需要移动的内存逐渐累加变得越来越大。当读完 100 个新行时（每行仅仅 2KB 字节），Lua 已经移动了 5MB 字节的内存，正是下面的赋值语句令情况变得更糟：

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

执行上述语句的时候，旧的字符串变成了垃圾数据。仅仅是执行两轮循环，就会有两个大小超过 100KB 字节的旧字符串成了垃圾数据。这个时候正如预期的，Lua 将启动垃圾收集器来释放不再有用的内存。问题在于每一次循环都会导致 Lua 进行垃圾收集，因此在读取整个文件之前，垃圾收集器已经运行了 2000 次。即使垃圾收集期间一切正常，它所使用的内存大小也已近该文件的三倍之多。

然而这并不是 Lua 特有的问题：其他的拥有垃圾收集器且无法修改字符串的语言都存在这个问题。Java 就是其中最著名的例子，为此，Java 专门提供了 StringBuffer 类来改善这种情况。

在继续下文之前，需要提醒一下，不管前面介绍了什么，这并不是一个经常会遇到的问题。对于短小的字符串，上面的循环算法完全可以胜任，但如果是读取整个文件，那么我们得使用 io.read(*all)，它能够一次性读取文件的全部内容。但在某些时候，并不存在简单的解决办法，因此唯一的答案就是最好的答案。下面我们将介绍一个解决方法。

上面介绍的程序采用了线性复杂度的解法，通过逐个连接小字符串来生成最终的字符串。下面将要介绍的新算法采用了二分法（也就是 Log 复杂度的解法）来避免这一弊端，它将若干小字符串连接起来，并在适时将这些大字符串合并成一个更大的字符串，这个算法的核心在于一个用于将大字符串保存在栈

底的栈空间，而后续的小字符串从栈顶入栈。这个栈所遵循的原则就如同广为人知（至少在程序员之间）的汉诺塔（Tower of Hanoi）一样：位于栈下层的字符串总是比上层的长。只要一个更长的字符串入栈，那么这两个串势必要连接成一个更大的串，这个更大的新串可能比它下层的串更大，如果真的是这样，那么新串与其下的串也必须连接成更新的串。之后将重复前面的步骤，直到遇上更大的串或到达栈的底部。

```
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
```

为了取得最终的字符串，我们只需要从上而下依次合并所有的字符串即可，因此例子中用到了函数 table.concat，它可以将一系列的字符串合并在一起。

有了这个新的数据结构，我们总段可以重写上述代码了：

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

还是读取 350KB 字节的文件，最终的程序将运行时间减少从一开始的 40 秒减少到现在地 0.5 秒，当然，调用 io.read("*all")仍然是最好的选择，因为它只需要 0.02 秒。

实际上，当我们调用 io.read("*all")时，io.read 所使用的数据结构就是我们刚刚介绍过的那类，只不过它的代码是用 C 实现的。Lua 标准库中的不少函数也采用了这种 C 的实现方式，函数 table.concat 就是其中之一。使用 table.conca 能够可以很快地将一个表中的所有字符串连接起来，由于它使用了 C 的实现方式，因此即使是遇到很大的字符串它也能非常有效率地工作。

函数 concat 的第二个参数是可选的，它代表将要被插入字符串之间的分隔符。有了这个参数，我们可以不必在每一行之后都插入一个新行：

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

迭代器 io.lines 返回不带换行符的行，concat 函数在字符串之间插入指定的分隔符，除了最后一个字符串的后面之外，因此我们需要在最后的字符串后面加上一个分隔符，但这最后连接末尾的分隔符操作将复制整个字符串，这个字符串可能体积巨大，而函数本身并不提供任何选项帮助我们有效地连接最后的分隔符。不过我们可以使用一个小伎俩来蒙骗 concat 函数，那就是在文件内所有的行被读取完毕之

后，再“读取”一个空串：

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

这样，新行将被连接在结果串之后、空串之前，而这正是我们想要的结果。

第 12 章 数据文件与持久化

一般来说，处理数据文件的，写文件比读取文件内容更容易一些。向文件写入数据时我们有所有的控制权，而从文件读取数据时常常碰到不可预知的情况。除了正常的文件包含的各类数据，一个健壮的程序还应该能够读取错误的文件，正因为如此，编写健壮的输入程序总是很困难的。

正如我们在 10.1 章节中看到的例子，表构造器为文件格式提供了一个有趣的解决方案，只要在写入数据时完成一些额外的工作，读取数据时将变得容易很多。这种技术的实现方法是：将文件内容作为 Lua 代码写到 Lua 程序中去，当它被执行的时候，数据也就建立了。通过使用表构造器，这些 Lua 代码中的数据就像普通的平文本文件。

为了更清楚地描述问题，让我们看一些例子。如果数据文件使用了预定义格式，比如 CSV（逗号分隔值文件），那么我们只有很少的选择（在第 20 章，我们会介绍如何用 Lua 读取 CSV）。但是，如果是创建文件的话，除了 CSV，还可以使用 Lua 表构造器来格式化数据，在这种数据格式中，每一个数据记录都用一个 Lua 表构造器来描述。如果有以下内容：

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

那么可以用我们的格式来定义成：

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

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

记住 Entry{...}与 Entry({...})等价，意为，以表作为唯一参数调用 Entry 函数。因此，上述数据同样也是一段 Lua 程序代码。要读取这些数据，只需要定义一个合适的 Entry 函数，并运行这段 Lua 代码。下面的代码计算数据文件中的记录数：

```
local count = 0
function Entry(b)
    count = count + 1
end
dofile("data")
print("number of entries: " .. count)
```

下面的程序取得数据文件中所有的作者名字的集合，并将其打印出来。作者名字是记录的第一个域，所以，如果 b 是一个记录，b[1]则代表作者名字。

```
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，上述程序段使用了事件驱动的方法：Entry 函数作为回调函数，在 dofile 处理数据文件中的记录时调用它。

在不考虑数据文件大小的情况下，我们可以使用“名字-值”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
}
```

这种格式是否让你想起了 BibTeX，这并非巧合，Lua 的表构造器正是受 BibTeX 的启发而产生的。这种格式被称为自描述数据格式，因为每一个数据值都与一个能简要描述其意义的描述信息相关联。相对 CSV 或其他紧缩格式，自描述数据格式更容易被人类阅读和理解；如果需要，还进行手工编辑；且要做出微小变动时不需要改动数据文件。例如需要增加一个域，就只需要对读取数据的程序，甚至为不存在的域提供了默认值。

在使用“名字-值”对的前提下，上述程序可以改成：

```
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
```

这样一来，记录中域的顺序就变得无关紧要了。即使某些记录缺少 author 信息，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，编译速度也很快，例如，上述代码在处理一个 2MB 的数据文件所花费的时间不会超过 1 秒，这再次证明 Lua 的快并不是偶然。自 Lua 诞生以来，数据描述就是 Lua 的主要应用之一，我们花费了大量的心血令它能够更快地编译和运行大块的代码段。

12.1 序列化

我们经常需要对数据进行序列化处理，也就是说将数据转换成字节流或字符流，以便将其保存到文件或便于通过网络传送数据。Lua 代码能够对数据进行序列化处理，即运行 Lua 代码，程序就读取重新构造出来的保存值。

通常，要想恢复一个全局变量的值，相应的 Lua 代码看起来就会像 varname = <exp>这样，其中的<exp>就需要恢复的保存值。varname 部分比较容易理解，因此让我们来看一下如何用代码恢复保存值。对于一个数值类型的值而言，其代码相对比较简单：

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

而对于字符串类型的值，一个初步的实现方式如下：

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

然而，这显然是不够的，字符串可能包含特殊字符（诸如引号或者换行符等），因此最终产生的将不是有效的 Lua 程序代码。这时候你可能会试图改变引用定界符：

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

但是，请不要这么做。因为双方括号是用于手写字符串的，而不是被用于自动产生的字符串。如果有人恶意地利用你的程序去保存类似“]]..os.execute('rm *')..[[”的字符串（比如在需要填入其地址的时候填入了该字符串），那么最终的代码将是这个样子：

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

如果你在程序中载入这段“数据”，其运行结果就可想而知了。

Lua 的 string 库中有个名为 format 的函数，它提供了一个选项“%q”来保证任意字符串都能安全地被引用，它使用双引号来引用字符串，并且正确地将双引号、换行符以及其他特殊字符转义成相应的代码。利用这种特性，我们可以像下面这样重写 serialize 函数：

```
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 保存不带环状结构的表

接下来，我们将尝试一个更难的任务：保存 Lua 表。根据表结构的不同，保存它的方法也有所不同，在此，并没有一种单一的解决方法。简单的表不仅需要简单的算法，而且最终的输出文件也要漂亮。

初步代码如下：

```
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
```

代码很简单，但合理地解决了问题。只要表是树型结构（也就是说，不存在被共享的子表，不存在环状结构），上述代码甚至能够处理嵌套的表（表中还含有类型为表的数据）。该代码尚可改进之处就是对嵌套表的缩进，读者可以将其作为一个练习（提示：为 serialize 函数增加一个参数，用以提示缩进字符串）。

上述函数假定表中的索引为有效的标识符。如果一个表用不符合 Lua 标识符语法的数值或者字符串作为索引，情况就大有不同了。一个简单的解决方法是将上述代码中的：

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

替换成：

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

经过上述修改之后，函数的健壮性已经得到了很好的巩固，比较一下两次的结果：

```
-- result of serialize{a = 12, b = 'Lua', key = 'another "one"'}
-- first version
{
    a = 12,
    b = "Lua",
    key = "another \"one\"",
}
```

```
-- second version
{
    ["a"] = 12,
    ["b"] = "Lua",
    ["key"] = "another \"one\"",
}
```

我们可以尽可能多地测试每一种情况，看是否需要方括号。同样地，这个问题将作为一个练习机会留给大家。

12.1.2 保存带有环状结构的表

针对普遍意义上的拓扑结构的表（即，内部具有被共享的子表，具有环状结构的表），需要一个不同方法将其序列化。表构造器表示这种表，因此接下来将不再使用它。表示环状结构需要知道子表的名字，以便后面将介绍的函数能够将其与保存值作为参数。另外，还需要记录被保存过的子表名，因为当检测到环状结构时还需要重用它（而不是重复保存它），为此，我们需要一个额外的表来保存这些子表名，该表将子表变量作为索引，而将子表名作为值。

首先假定我们需要保存的表仅使用数值或字符串作为索引，下面的函数负责序列化基本类型（数值和字符串），并返回经过序列化的值：

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

关键部分在于下面的函数，其中 saved 参数指用于记录已经被保存过的子表的表：

```
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
```

就上述代码给出一个例子，如果我们有下列表：

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

当调用 save('a', a)序列化表 a，将得到下列结果：

```
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]
```

实际的赋值顺序可能有所不同，因为它依赖于表遍历，尽管如此，在需要重新定义生成该表的时候，此算法保证了表中的所有节点都会被定义。

如果要保存带有被共享子表的表时，我们可以使用同一个 saved 表作为参数调用 save 函数。例如存在下列二表：

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

使用下面的调用保存它们：

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

输出结果将包含同样的数据：

```
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"
```

但是如果我们使用同样的 saved 表作为参数调用 save 函数：

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

则最终的输出结果将共享同样的数据：

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

与以前例子一样，为了保存带有环状结构的表，还有其他的方法可以采用。比如，可以不使用全局标量来保存值，即使用闭包机制，利用代码段构造一个局部变量并将其返回；还比如，通过构造一张表，并为所有的子表建立一个函数。方法有很多，Lua 给予你控制权，而由你决定实现机制。

第 13 章 元表和元方法

在 Lua 中有关表的所有操作都是预先定义好的，我们可以为表添加“索引-值”对，可以访问索引对应的值，可以遍历所有的“索引-值”对。但是，我们不能对两个表执行加操作，不能将两个表进行比较，也不能想调用函数一样调用一个表。

元表（Metatable）允许我们改变表的行为模式，比如我们可以使用元表来定义如何计算如何计算表达式 a + b 的值，而 a 和 b 都是表。当 Lua 试图对两个表进行加操作时，它会依次检查两个表，看两表中是否至少有一个表拥有元表，并且检查该元表中是否存在一个名为“__add”的域，如果存在这个域，则调用这个域所对应的值来进行加操作，这个值即所谓的元方法（Metamethod），它是一个函数类型的值。

每一个 Lua 表都有其元表（后面我们还将看到 userdata 也有元表），但是，Lua 总是默认地创建不带元表的新表：

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

可以使用 setmetatable 函数来设置或改变一个表的元表：

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

任何一个表都可能是任何其他表的元表，一组相关的表可以共享一个元表（这个元表将描述它们共有的行为模式），当然，一个表也可以是它自身的元表（这个元表将描述其独特的行为模式）。表与其元表之间的搭配是没有限制的。

13.1 算术运算的元方法

在这一部分，我们将通过一个简单的例子来说明如何使用元表。我们将使用 Lua 表来描述集合这个数据结构，并使用函数来描述该集合的并集、交集等集合上的操作。就像前面介绍链表一样，我们将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
```

为了便于监视程序的运行，我们也定义了能够打印集合内容的函数：

```
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
```

为了使用加号运算符来执行并集的操作，我们将令所有的集合共享一个元表，而这个元表将定义集合在加号运算符上的操作。首先，定义一个普通的表，并让它成为所有集合的元表。为避免污染名字空间，我们将在 Set 表内部定义它。

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

接着，需要修改用于创建集合的 Set.new 函数。修改过的函数与原函数相比多了一行额外的代码，在创建新集合时候，这额外的代码将 mt 设为新集合的元表：

```
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
```

之后，用函数 Set.new 创建的新集合都有了相同的元表：

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

最后，我们给这个元表添加元方法，元表的域__add 将用于描述并集的操作：

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

当 Lua 试图对两个集合进行加操作时候，用于__add 的函数将会被调用，加号操作符两边的集合将作为该函数的参数。

通过刚才定义的元方法，现在可以用加号操作符对两个集合进行并集的操作：

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

同样地，我们可以使用乘号运算符对集合进行并集的操作：

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

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

每一个算术运算符都有一个元方法与之对应，除了已经介绍过的__add、__mul，还有__sub（减号操作符）、__div（除号操作符）、__unm（负数操作符）以及__pow（指数操作符）。另外，我们也可以定义__concat 的行为模式，__concat 对应于连接操作符。

定义了加号操作符之后，将两个集合相加已经没有任何问题。但是，我们可能需要对两个拥有不同元表的值进行加号操作，例如：

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

这里是 Lua 选择元方法的原则：

1、如果第一操作数拥有附带__add 域的元表，Lua 都将使用该元方法，而不管第二操作数为何；

2、如果第一操作符不满足上述条件，而第二操作数却满足该条件，Lua 使用第二操作数的元方法；

3、除此之外，Lua 将抛出错误。

虽然我们在编写代码的时候会关心此类问题，但是 Lua 并不关心这种混合类型的操作符出错。因此，不管上例最后一个语句也好，诸如 10 + s 或者"hy" + s 也好，都将在 Set.union 函数内抛出一个错误：

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

如果想得到更具体的错误报告，我们必须在运算之前显式地检查操作数的类型：

```
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 关系运算的元方法

元表同样允许我们使用元方法定义下列关系运算符：__eq（等于）、__lt（小于）和__le（小于等于）。对于余下的三种关系运算符（即，不等于、大于以及大于等于），Lua 并没有给出独立的元方法，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。

在 Lua 4.0 以前，所有的次序（Order）运算符都被统一成一个，比如，a <= b 转换为 not (b < a)。然而这种转换在偏序（Partial Order）的情况下并不正确，所谓偏序，意为着在同一类型中，并不是所有元素都能被正确地排序。比如在大多数机器上，浮点数不能被完全排序，因为浮点类型中存在一个名为 NaN 的数（Not a Number），它的值并不是一个数字，根据现行的 IEEE 754 标准，NaN 表示一个未经定义的值，比如 0/0，其结果就是 NaN。该标准指出，任何涉及 NaN 比较的结果都应为假（false），也就是说，NaN <= x 总是为假，而 x < NaN 也总是为假。于是，将 a <= b 转换为 not (b < a)就不再正确了。

在我们有关集合的例子中，存在着类似的问题。在集合关系运算中“<=”代表着集合的“包含于”操作：a <= b 表示集合 a 是集合 b 的子集。在此，a <= b 和 b < a 可能同时为假，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，我们将通过集合的“包含于”关系运算来定义集合的相等关系运算：

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

有了上述定义之后，我们就可以对集合进行关系运算了：

```
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
```

与算术运算的元方法不同，关系元算的元方法不支持混合类型的运算，对于混合类型的关系运算的处理方法就如同 Lua 的普通行为模式：如果试图比较一个字符串和一个数值，Lua 将抛出错误。与之类似，如果试图比较两个带有不同元方法的对象，Lua 也将抛出错误。

但关系运算中的相等运算绝不会抛出错误，如果两个对象拥有不同的元方法，相等运算的结果必定为假，甚至不需要调用任何一方的元方法。这种行为模式也与 Lua 的普通行为模式相仿： Lua 总是认为字符串和数值是不相等的，不管它们为何值。只有相等运算的两个操作对象拥有同一个相等运算的元方法时，Lua 才会调用相应的元方法进行相等运算。

13.3 库预定义的元方法

一些库在其元表中定义自己的域是一种很常见的做法。到目前为止，我们看到的所有元方法都是替 Lua 的核心设计的：如果参与特殊运算符操作的对象包含了元表，并且在元表上定义了与该运算符对应的元方法，那么 Lua 的核心，也就是 Lua 虚拟机，会负责处理这些参与运算的对象。不过元表只是普通的表，任何人都能使用它。

函数 tostring 就是一个典型的例子。如前文所见，tostring 函数以相当简单的方式来表示 Lua 表：

```
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 函数会首先检查该对象是否存在一个附带__tostring 域的元表，加入果真如此，则以对象作为参数调用对应的函数来完成格式化任务，该函数的返回结果也就是 tostring 函数最终返回的结果。

在集合的例子中，我们已经定义过一个函数将集合转换成字符串并打印输出，因此，只需要简单地用这个函数设定集合元表的__tostring 域：

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

在此之后，不管什么时候调用 print 函数打印集合，print 都会自动调用系统预定义的 tostring 函数，而该函数则会调用 Set.tostring 函数：

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

函数 setmetatable 和 getmetatable 也使用类似与__tostring 这样的域来保护元表。假定你想保护集合的元表，使其使用者既无法访问也无法修改，那么你可以设定元表的__metatable 域，这样，调用 getmetatable 只能得到该域的设定值，而调用 setmetatable 将会抛出错误：

```
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 表存取的元方法

算术运算和关系运算的元方法定义了必要条件下表的行为模式，但它们并不改变语言本身的行为模式。不过 Lua 提供了两类普遍情况下对表的行为模式的修改：对表中不存在的域的查询和修改。

13.4.1 __index元方法

前文已经介绍过，如果访问表中一个不存在的域，则其返回结果为空值，这是正确的，但又不完全正确。实际上，访问表中不存在的域将触发 Lua 解释器去寻找__index 元方法：如果元方法不存在，则其返回结果与预料中的一样，为空值；如果元方法存在，则其返回结果由该元方法给出。

下面将要给出的这个例子，其的原理就是继承。假设我们想创建一些表来描述屏幕上的窗口，则每一个表都必须描述窗口的某些参数，比如：位置，大小，颜色方案等等。所有的这些参数都有其默认值，这样，当我们创建窗口的时候只需给出非默认值的参数即可。第一种方法是，提供一个构造器，并填上所有缺失的域。第二种方法是，创建一个新窗口来继承一个原型窗口中缺少的域。首先，我们需要声明

一个原型和一个构造函数，而这个构造函数将创建共享同一个元表的新窗口：

```
-- 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
```

现在我们来定义__index 元方法：

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

有了上述代码，当我们创建一个新窗口并访问它缺少的域，其结果如下：

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

当 Lua 发现 w 不存在域 width，但拥有一个附带__index 域的元表时，Lua 将以 w（table 参数）和 width（缺少的域）作为参数调用__index 元方法，而该元方法将在原型中查找缺少的域并返回其对应的值。

__index 元方法在继承中非常常见，因此 Lua 提供了一套更便捷的方式。__index 不必非得是一个函数，它也可以是一个表。当它是一个函数时，Lua 将以表和缺少的域作为参数调用这个函数；而当它是一个表的时候，Lua 将在这个表中查找是否含有缺少的域。所以上述例子可以使用第二种方式进行修改：

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

现在，当 Lua 查找元表的__index 域时，它将发现 Window.prototype 这个表就是它的值，于是 Lua 将在该表中查询缺少的域，也就是说，它相当于执行了：

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

正如所期望的，这个等价的方式同样返回正确的结果。

将一个表作为__index 元方法来使用，为单继承提供了一种简洁的实现方法。尽管使用一个函数的代价有点高，但却提供了更多的灵活性：我们可以实现多重继承、隐藏以及一些其他的特性。我们将在第 16 章详细讨论更多继承的方式。

想要绕过__index 元方法来访问一个表，可以使用 rawget 函数。对 rawget(t, i)的调用将以最直接的方式访问表 t。这种访问方式不会加速代码的执行（通常，调用函数的代价将会抵消你在其他方面取得的好处），但有些时候我们的确需要他，之后我们将看到这一点。

13.4.2 __newindex元方法

__newindex 元方法用于更新表，而__index 元方法则用于访问表。当你给表的一个缺失的域赋值，解释器就会查找__newindex 元方法：如果存在，则调用它，而不进行赋值操作。像__index 一样，如果

元方法是一个表，解释器将对该表进行赋值操作，而不是原来的表。另外，有一个能直接存取的函数可以绕过元方法：执行 rawset(t, k, v)将不会调用任何元方法，而是直接赋值 v 给 t 表的 k 域。

__index 和__newindex 元方法的组合提供了若干强有力的语言结构：只读表、带有默认值的表、面向对象的编程中的继承。在本章的剩余部分我们将看到这些应用的例子，而面向对象的编程将另立章节。

13.4.3 附带默认值的表

一个普通表中任何域的默认值都是空值，然而，通过元表可以很容易地改变其默认值：

```
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上述表中缺少的域，它的__index 元方法都将被调用并返回 0。但如果需要为其设定默认值的表为数众多，使用这种方法的代价将会很高。元表将默认值 d 与其本身关联，因此 setDefault 函数不能为所有表使用单一的元表。为了避免为不同的默认值使用同一个元表，我们可以将默认值存储在每个表中的某个特殊的域。如果不必命名冲突带来的混乱，我们可使用诸如“___”这样的值作为特殊域：

```
local mt = {__index = function(t)
    return t.___
end}

function setDefault(t, d)
    t.___ = d
    setmetatable(t, mt)
end
```

如果命名冲突是必须避免的，为了保证这个特殊域的唯一性其实也并不困难，只需要创建一个新表作为其特殊域：

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

function setDefault(t, d)
    t[key] = d
    setmetatable(t, mt)
end
```

另外一种方法是使用独立的表，在这个独立的表中，索引是普通表，索引对应的值为默认值。然而

这种方法的正确实现还需要一种特殊的表：Weak 表。我们决定在第 17 章讨论这种表，因此暂时不会在此使用到它。

还有一种方法是保留元表的信息，以便能重用同一个元表，然而这种方法也需要 Weak 表的支持，因此我们不得不等到第 17 章。

13.4.4 监控表的访问

__index 和__newindex 都是只有当表中访问的域不存在时候才起作用。要监视某个表的访问情况的唯一方法就是保持该表为空，要监视该表的所有访问情况，我们还应该为该表创建一个代理表，这个代理表是一个空表，它通过__index 和__newindex 元方法监视访问的情况并将访问重定向到原来的表。假定，t 是需要监视的表，我们可以设计类似下面这样的代码：

```
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)
```

这段代码将监视所有针对 t 的访问：

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

上述设计的遗憾之处是，不允许遍历表，因为 pairs 函数将对代理表进行操作，而不是原始的表。

要想监视多张表，并不需要为每个表都建立不同的元表，这时可以将每个代理表与其原始表关联，而所有的代理表共享同一个元表即可。将表与其代理表关联的一个简单的方法是，将表作为代理表的域，只要能够保证这个域不会被移作他用。为保证它不被移作他用，可以创建一个私有的域令他人无法对其进行访问。综上所述，有了下面的代码：

```
-- 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
```

这样，不管何时想要监视表 t，只需要执行 t = track(t)就能得到与上例同样的结果：

```
t = {}
t = track(t)
t[2] = "Hello"
print(t[2])
```

13.4.5 只读表

采用代理表的思想能够很容易地实现只读表，需要做得只是当监控到企图修改表的操作时抛出错误。对于__index 元方法，可以只使用原始表自身特性，而不使用任何函数，因为对原始表的查询并不是我们需要关注的。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。不过这种方法要求每个代理表拥有独立的新元表，且该元表的__index 域指向原始表本身：

```
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
```

注意 error 函数的第二个参数值 2，它能够将错误信息重定向到试图执行更新操作的代码。作为一个

应用的例子，我们将创建一个与工作日相关的只读表：

```
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 Lua的运行环境

Lua 用一个普通的表来保存所有的全局变量，这个表被成为 Lua 的运行环境（更精确地说，Lua 在多个不同的运行环境中保存它的“全局”变量，但是有时候我们会选择忽视这种重复的运行环境）。这种结构的优点之一是，它简化了 Lua 的内部实现，因为这样一来，Lua 就不必为全局变量设计不同的数据结构。另一个优点，也是最大的优点是，我们可以像操作其他表一样操作那些用于保存全局变量的表。为了提供这种操纵方式， Lua 将运行环境本身存储在全局变量_G 中（也就是说，_G._G 等同于_G）。下面代码将打印出当前运行环境中所有全局变量的名字：

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

这一章我们将讨论一些用于操纵运行环境的有用技术。

14.1 使用动态名字访问全局变量

通常情况下，赋值操作对于访问和修改全局变量已经足够。然而，我们还时常需要某些形式的元编程（Meta-Programming），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，或者名字需要在运行时才能知道的全局变量。为了获取这类全局变量的值，很多程序员会这样做：

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

或者

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

如果 varname 的值为“x”，则上述字符串连接操作的结果为：“return x”（第一种形式为“value = x”），之后在运行的时候才会产生最终的结果。然而，这段代码涉及到一个新的代码段的创建及其编译，以及其他大量额外的负担。你可以换种方式以更高效更简洁的地方式完成同样的功能：

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

因为运行环境只是一个普通的表，所以只需要使用对应的变量名作为索引，就能取得变量的值。

同样地，也可以以此方式为一个全局变量赋值：_G[varname] = value。但是请注意，一些程序员对此功能很是兴奋，并且可能写出这样的代码：_G["a"] = _G["var1"]，然而，这只是 a = var1 的复杂的写法而已。

综上所述，表域可以是诸如“io.read”或“a.b.c.d”之类的动态名字。我们将利用循环语句来解决这个问题，从_G 开始，逐个遍历其中的域：

```
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 函数来遍历 f 中的所有单词（单词指若干个个字母和下划线的序列）。

与之对应的用来设置域的函数稍显复杂。下列赋值语句：

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

完全等价于下列语句：

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

也就是说，我们必须取得最后一个名字之前的所有名字，且必须独立地处理位于最后的这个域（名字）。新的 setfield 函数在位于中间的域不存在的时候，还需要创建额外的临时表：

```
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
```

函数 gfind 中的新匹配模式将捕获一个变量名（并保存在 w 中）和一个可能存在的点（并保存在 d 中）。如果该变量名之后不存在点，则代表它就是最后的名字（我们将在第 20 章详细讨论模式匹配问题）。

当使用上述函数，并执行：

```
setfield("t.x.y", 10)
```

上述语句将创建一个全局表 t 和另一个表 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，对于小型程序来说这的确很方便，但在大型程序中，一个简单的拼写问题就有可能引发难以排除程序错误。然而，我们可以更许需要来改变这种行为模式。Lua 将全局变量都保存在一个普通的表中，因此我们可以利用元表机制来改变全局变量访问的行为模式。

下面是第一种方案：

```
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，任何试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
```

但是又该如何声明新的变量呢？我们可以使用 rawset 函数，它能够绕过元方法进行赋值操作：

```
function declare(name, initval)
    rawset(_G, name, initval or false)
end
```

使用“or false”是为了保证新声明的全局变量不为空值。注意，必须在设定存取全局变量的控制机制之前定义好该函数，否则你将得到一个错误信息：因为定义该函数意味着你在创建一个新的全局变量 declare。只要设定了上述函数，你就可以完全控制全局变量的存取了：

```
> 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已经存在，我们不能简单将其与空值进行比较，因为如果它未经声明，即它的值为空值，在比较操作中访问它就必然会抛出错误。因此，我们需要使用 rawget 函数来绕过元方法：

```
if rawget(_G, var) == nil then
    -- 'var' is undeclared
    ...
end
```

改变全局标量的存取控制以便允许全局变量的值为空值并不困那，我们需要的仅仅是一个用于记录已声明的变量名的辅助表，当元方法被调用的时候，它将检查这个辅助表，该变量是否已经被声明过看。其代码如下：

```
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,
})
```

这两种实现方式的代价几乎微不足道。在第一种方式下，元方法在通常操作中不会被调用，而在第二种方式下，元方法仅在在程序访问一个值为空值的变量时才会被调用。

14.3 非全局的运行环境

运行环境的一个问题是它是全局的，任何修改都会直接影响程序的所有部分。例如，你设置了一个元表来控制全局变量的存取，那么你的整个程序都必须遵循这一指导方针，如果你想使用某个 Lua 库，但是该库可能使用了未经声明的全局变量，这是你就无法在你的程序中使用该库。

Lua 5.0 改善了这一问题，它允许每个函数拥有属于自己的运行环境。这听起来这很奇怪，毕竟，全局变量表的目的就是为了使用变脸全局化。然而，你将会在 15.4 章节内看到这一机制带来的很多有趣的结构，而全局变量依然保持其全局性。

要改变一个函数的运行环境，可以使用 setfenv（Set Function Environment）函数来进行设定，该函数以函数和新运行环境作为参数。除了使用函数作为参数，你还可以用一个数字来表示指定栈层次中的活动函数（该栈中正在被执行的函数）。数字 1 代表当前函数（即当前栈的当前函数），数字 2 代表调用当前函数的函数（对于编写辅助函数来改变其调用者的运行环境，这的确是很方便的），以此类推。

下面是一个使用 setfenv 函数设定运行环境而导致访问失败的例子：

```
a = 1                -- create a global variable
setfenv(1, {})       -- change current environment to a new empty table
print(a)
```

上述代码将导致下面的错误：

```
stdin:5: attempt to call global 'print' (a nil value)
```

你必须在单独的代码段内运行这段代码，如果你在交互模式下逐行运行它，那么每一行都是一个不同的函数，调用 setfenv 只会影响它自己那一行。一旦你改变了运行环境，所有全局事务都参考这个新的表，如果它为空，便无法访问任何全局变量，甚至是_G，所以，你应该首先使用某些有意义的值来对其进行赋值，比如旧的运行环境：

```
a = 1                    -- create a global variable
setfenv(1, {_G = _G})    -- change current environment
_G.print(a)              --> nil
_G.print(_G.a)           --> 1
```

现在，当访问“全局的”_G，它将提供旧的运行环境，在此运行环境中你仍然能够使用 print 函数。

你也可以使用继承机制来生成新的运行环境：

```
a = 1
local newgt = {}         -- create new environment
setmetatable(newgt, {__index = _G})
setfenv(1, newgt)        -- set it
print(a)                 --> 1
```

在上述代码中，新的运行环境从旧的运行环境处继承了 print 和 a，尽管如此，任何赋值操作都是针对新表进行，不必担心由于误操作而修改了原来的全局变量表，当然，你仍然可以通过_G 来对它们进行修改：

```
-- 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，它从创建它的函数处继承了运行环境。因此，如果一个代码段改变了它自己的运行环境，则所有在此之后定义的函数都将受新运行环境的影响。对于创建名字空间而言，这是非常有用的机制，关于这一点，我们将在下一章看到。

第 15 章 Lua包

很多语言都提供了某种机制来组织全局的名字空间，比如 Modula 的模块（Modula），Java 和 Perl 的包（Package），C++的的名字空间（Namespace）。对于包内已声明元素的使用、可见性以及其他细节，每一种机制都有自己的规范，但是它们都不约而同地都提供了一种避免不同库中名字冲突的机制。每一个库都有用自己的名字空间，在这个名字空间内定义的名字并与其他名字空间中定义的名字互不干扰。

Lua 并没有提供明确的机制来实现包，然而，我们可以通过语言提供的基本机制来实现它。实现包的主要的思想是：像标准库一样，使用表来描述包。

使用表来实现包的一个明显的好处是：我们可以像操纵普通表一样使用操纵包，并使用语言提供的全部能力来实现额外的功能。在大多数语言中，包不是第一类值（First-Class Values），也就是说，它们不能被存储在变量里，不能作为函数的参数等等。因此，Lua 包的某些额外功能，在这些语言中就需要特殊的方法和技巧才能实现。

在 Lua 中，虽然我们总是用表来实现包，其实还有其他不同的方法来实现这一机制。在这一章，我们将介绍其中的一些方法。

15.1 最基本的方法

定义包的一个简单方法是，将包名作为包内每个对象名的前缀。假定需要编写一个用于操作复数的库，我们将每个复述表示为一个表，该表有 r（实数部分）和 i（虚数部分）两个域。我们将在另一个表中声明所有的操作，这个表的行为模式就如同一个包：

```
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的名字 complex，其他所有定义都在这个表内。

有了上述的定义，我们就可以根据操作的名字来进行任何有关复数操作了，比如：

```
c = complex.add(complex.i, complex.new(10, 20))
```

这种使用表来实现的包和真正的包在功能上并不完全相同。首先，对每一个函数的定义都必须显式地加上包名。第二，同一个包内的函数相互调用必须在被调用函数的前面指定包名。我们可以为包指定一个局部的名称来改善这个问题，这个局部变量将被赋值给最终的包。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（或者它调用自身进行递归），它仍然需要加上包名作为前缀。但至少目前，它不再依赖于包名，因为，在整个包的内部，只有一个地方需要填写包名。

你可能已经注意到包中的最后一个语句：

```
return complex
```

这个返回语句并非必需，因为包本身已经赋值给全局变量 complex 了。但是，我们认为包在打开的时候返回本身是一个好习惯，额外的返回语句并没有什么代价，同时提供了另一种操作包的方式。

15.2 私有化

有些包选择公开它所有的内容，也就是说，任何使用该包的人都能访问它内部的所有名字。然而，有时候一个包拥有自己的私有部分是很有用的，这意味着只有包本身才能访问这些私有的数据或函数。Lua 的传统做法是将私有部分定义为局部变量。举个例子，修改前一个例子，为其增加一个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
```

这种做法有何种优缺点呢？包中所有的名字都存在于一个独立的名字空间内，其中的每一个实体（Entity）都清楚地被划分为公有或者私有，而且，存在真正的隐私：私有实体在包的外部是不可访问的。其中的缺点之一是，访问同一个包内的其他公有的实体的写法显得冗余，因为访问内部实体都必须加上前缀“P.”。还有一个大问题是，当修改函数的状态（公有变私有，或私有变公有）时，也必须修改函数的调用方式。

有一个有趣的方法可以同时解决这两个问题。我们可以将包内的所有函数都声明为局部的，之后将其在最终的表中设为公有。按照这种方法，complex 包将编程下面这样：

```
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,
}
```

有了上述代码之后，我们不再需要为函数调用加上前缀，对公有和私有函数的调用方法完全相同。在包的结尾部分，使用一个简明的表列出所有的公有函数。可能大多数人觉得将这个表放在包的开头部分更为自然，但这是行不通的，因为表中的那些局部函数必须首先经过定义。

15.3 包与文件

通常，当编写一个包时，我们将所有的代码写在一个单独的文件中，之后如果想要打开或载入这个包（令其处于可用状态），只需要执行这个文件即可。举个例子，如果 complex.lua 这个文件包含了我们对复数包的定义，只要执行 require "complex"就可以载入这个包。记住 require 命令会自动避免同一个包的多次加载。

一个只得注意的问题是，包名与该包所在文件的文件名之间的关系，当然，在两者之间建立某种关联是一个不错的想法，因为 require 命令针对的是文件而不是包。一种方案根据包名建立对应的文件名，并加上文件扩展名。对于文件的全名，Lua 并不附加任何扩展名，它完全由指定的路径名决定的。例如，指定路径包含了“/usr/local/lualibs/?.lua”，那么 Lua 会认为 complex 包可能可能保存在一个名为 complex.lua 的文件中。

在命名规则上，有些人选择相反的做法，即根据文件名动态地来确定包名，也就是说，一旦文件被重命名，则包也会被重命名。这个命名方案提供了更大的灵活性。例如，假设存在两个同名的不同包，你不必修改其中任何一个包的名称，只需要重命名文件即可。我们使用_REQUIREDNAME 变量来实现这种命名方案，注意当 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，你可以在包代码中这么做：

```
local P = {}          -- package
if _REQUIREDNAME == nil then
    complex = P
else
    _G[_REQUIREDNAME] = P
end
```

上述例子中的测试代码允许我们无需使用 require 就能使用该包。如果_REQUIREDNAME 未经定义，我们可以用某个固定的名字表示包（例子中为 complex），否则，包将使用虚拟文件名进行注册，而不管虚拟文件名是什么。如果用户将包放在文件 cpx.lua 中，并运行 require "cpx"，那么包将自身加载到 cpx 表中，如果有另外的用户将包放在文件 cpx_v1.lua 中，并运行 require "cpx_v1"，那么包会将自身加载到 cpx_v1 表中。

15.4 使用全局表

上文中创建包的方法有其缺点：很多细节问题要求程序员自己关注。比如，在声明的时候 local 关键字是很容易被忘记的。全局变量表的元方法提供了一些有趣的技术来实现包，这些技术的共同之处在于，每个包有其专用的运行环境。这是很容易实现的：只需要改变包所在主代码段的运行环境，就能使所有由包创建的函数共享这个新的运行环境。

一个最简单的实现技术能够做得更好。一旦包拥有了专用的运行环境（运行环境是用表来表示的），不仅会被包内的所有函数共享，而且也能被包内的所有全局变量共享（全局变量即在这个表内）。因此，我们尽可以将所有的公有函数声明为全局变量，它们将自动进入这个独立的表（运行环境），为实现这种技术，包只需要以包的名字注册这个表。下面的代码介绍了这种技术在 complex 包的应用：

```
local P = {}
complex = P
setfenv(1, P)
```

现在，当我们声明函数 add 时，它将自动被注册为 complex.add：

```
function add(c1, c2)
```

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

不仅如此，在这个包内部调用其他函数再也不需要使用任何前缀了。例如，add 函数调用 new 函数，前者会自动在运行环境中取得 complex.new。

这种方法为包的实现提供了很好的支持：程序员只需要不需要做很少的额外工作；在同一个包内调用函数不需要使用前缀；调用公有函数和私有函数没有任何区别；程序员忘记 local 关键字不会污染全局名字空间，只不过原来的私有函数变成了公有函数而已。另外，还可以将这种技术与前一节有关包名的技术结合起来：

```
local P = {}         -- package
if _REQUIREDNAME == nil then
    complex = P
else
    _G[_REQUIREDNAME] = P
end
setfenv(1, P)
```

当然，这样就不能访问其他的包了，一旦空表 P 成为新的运行环境，我们就无法再访问之前的所有全局变量。下面将介绍几种解决方案，它们各有利弊。

最简单的解决方法是使用继承，就像前面看到的一样：

```
local P = {}         -- package
setmetatable(P, {__index = _G})
setfenv(1, P)
```

函数 setfenv 必须在调用 setmetatable 之前被调用，你一定能够解释这个原因。将原来的运行环境纳入包之后，包就能直接访问所有的全局标示符了，虽然必须付出一点小小的代价。这种方案在概念方面附带了一个有趣的结果：这个包涵盖了所有的全局变量，因此，如果某人正在使用这个包，他可以通过调用 complex.math.sin 来完成对标准库中 sin 函数的调用（Perl 语言的包系统也有这种特性）。

另外一种支持快速访问其他包的方法是，声明一个局部变量来保存旧的运行环境：

```
local P = {}
pack = P
local _G = _G
setfenv(1, P)
```

现在，所有对包外部的访问都必须加上前缀“_G.”，但是访问速度比前一种方案更快，因为这里并没有涉及到元方法。与基于继承的方案不同的是，这种方法允许你修改旧的运行环境；这种方法的好坏与否颇具争议，但是有时候你可能会需要这种灵活性。

一个更加正规的方案是，仅将那些需要的函数或包声明为局部变量：

```
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)
```

这一技术要求额外的工作，但能够使包的独立性比较好。另外，它的运行速度也比前者更更胜一筹。

15.5 其他技巧

正如前面所说的，用表来实现包可以让我们发挥 Lua 的全部能力。这里面有无限的可能性，因此我只给出一些建议供读者参考。

包中的所有公用成员并不需要定义在同一处，例如，可以在 complex 包之外为其添加一个新的成员函数：

```
function complex.div(c1, c2)
    return complex.mul(c1, complex.inv(c2))
end
```

但是请注意，私有成员必须被限定一个文件内，我认为这是一件好事。反过来，我们可以在同一个文件内定义多个包，需要做的只是将每一个包放在独立的 do...end 代码块内，这样以来，局部变量才能被限制在那个代码块中。

在包的外部，如果需要经常进行某个操作，我们可以为它们声明局部变量名：

```
local add, i = complex.add, complex.i
c1 = add(complex.new(10, 20), i)
```

如果不想像上述代码那样做，而又不想一遍又一遍重复输入包名，那么可以为包声明一个短小的局部变量名：

```
local C = complex
c1 = C.add(C.new(10, 20), C.i)
```

把一个包拆开并将包内的所有名字存入全局名字空间是很容易的：

```
function openpackage(ns)
    for n, v in pairs(ns) do
        _G[n] = v
    end
end

openpackage(complex)
c1 = mul(new(10, 20), i)
```

如果你担心拆开包并将内部名字全局化的时候会有名字冲突，那么可以在赋值以前检查一下包内名字是否已经在全局表内存在：

```
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
```

由于包本身就是表，所以甚至可以在包内嵌套包，也就是说，可以在一个包内创建另一个包。不过，几乎没有必要用到这种技巧。

另一个有趣的技巧被称为自动加载（Autoload）：函数只有被使用的时候才会被自动加载。当我们加载一个拥有自动加载能力的包，它将创建一个空表来表示这个包，并设置 v__index 元方法来完成自动加载的功能。之后，当我们调用任何一个尚未被加载的函数时，__index 元方法将被调用，继而加载所需的函数，后续调用发现该函数已经被加载过，因此它们将直接使用该函数，而不去在此调用__index 元方法。

下面是一个简单用以实现自动加载功能的方法。每一个函数都被定义在一个辅助文件中（当然，每个文件内可以包含多个函数），每个文件中的函数都以普通的方式进行定义，例如：

```
function pack1.foo()
    ...
end

function pack1.goo()
    ...
end
```

然而，该文件并不需要创建包，因为当函数被加载时包早已存在。

在包的主文件中我们定义一个辅助的表来指明各函数所在的位置：

```
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",
}
```

紧接着，创建包并定义__index 元方法：

```
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
```

加载这个包之后，当程序第一次执行 pack1.foo()将会触发__index 元方法，该元方法检查函数所在的文件并加载该文件，唯一精妙之处在于，不仅需要加载文件，还必须返回该函数作为结果。

因为整个程序都是依靠 Lua 提供的功能完成的，所以很容易改变其行为模式。例如，函数可能是 C 语言编写的，不过仍然可以在元方法中使用 loadlib 函数加载它，或者我们可以在全局表中设定一个元方法来自动加载整个包。这意味着无限的可能性。

第 16 章 面向对象的程序设计

Lua 表在很多层面上都具有对象的特征：表具有状态；表有独立与其值的标识，具体地说，两个拥有一样值的表代表着两个不同的对象（一个对象在不同的时候有不同的值，但它始终是同一个对象）；表具有独立的生命周期，与表由谁创建、在哪里创建都没有关系。

对象具有成员函数，而表也具备此特性：

```
Account = {balance = 0}
function Account.withdraw(v)
    Account.balance = Account.balance - v
end
```

上述代码创建了一个新函数，并将其保存于 Account 对象的 withdraw 域，之后便可以这样调用：

```
Account.withdraw(100.00)
```

这种类型的函数几乎就是我们所谓的对象方法，然而，在一个函数内部使用全局变量名 Account 并不是一个好习惯：首先，这个函数只能对特定的对象起作用；其次，即使只作用于特定的对象，也只有在该对象被存储在特定的全局变量中时才起作用，如果我们改变了对象的名字，函数 withdraw 将无法正常工作：

```
a = Account;
Account = nil
a.withdraw(100.00)          -- ERROR!
```

这种行为模式违背了前面提到的，对象应该具有独立的生命周期这一原则。

一个灵活的解决方法是为被操作的对象指定一个“接受者”：定义对象方法时带上一个额外的参数，这个参数用以表示被操作的对象。经常使用 self 或 this 作为这个参数的名称：

```
function Account.withdraw(self, v)
    self.balance = self.balance - v
end
```

现在，当我们调用这个方法时就需要指定方法操作的对象了：

```
a1 = Account;
Account = nil
...
a1.withdraw(a1, 100.00)     -- OK
```

在函数中使用 self 参数之后，我们就可以在不止一个对象上使用该函数了：

```
a2 = {balance=0, withdraw = Account.withdraw}
...
a2.withdraw(a2, 260.00)
```

使用 self 参数是所有面向对象的语言的要点。大多数面向对象的语言都将这种机制隐藏起来，这样程序员就不必再声明这个参数了（虽然语言本身仍然可以在方法内部使用这个参数）。Lua 提供了通过使用冒号操作符来隐藏这个参数，这里我们将重写上述代码：

```
function Account:withdraw(v)
```

```
    self.balance = self.balance - v
end
```

其调用方法如下：

```
a:withdraw(100.00)
```

冒号语法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，这只是语法上的便利，实际上并没有什么新的内容。我们可以使用点语法定义函数而用冒号语法调用函数，或者反过来，用冒号语法定义函数而用点语法调用函数，只要我们能够正确地处理额外的参数：

```
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)
```

到目前位置，我们的对象已经拥有独立的标识、对象状态以及作用于状态的方法，不过它们依然缺少一个类型系统、继承机制和私有机制。先解决第一个问题：如何才能创建多个具有相似行为模式的对象？或者更具体地，怎样才能创建多个帐号（Account）对象？

16.1 类

类为对象的创建提供了一个模板，面向对象的语言提供了类的概念。在这些语言里，对象是类的一个实例。在 Lua 中没有类的概念，每个对象都必须定义自己的行为，并拥有自己的形态，然而，依据基于原型（Prototype）的语言（诸如 Self、NewtonScript 等）来模拟类的机制并不困难。在这些基于原型的语言中，对象并不属于某种类，相反，每个对象都可以拥有一个原型，而这个原型只不过是一个普通的对象，当执行不属于对象的操作时，会首先在原型查找这些操作。为了在这些语言中实现类的机制，我们可以创建一个对象并将其作为其他对象的原型（其他对象可以被视为该原型的实例）。类与原型的工作原理并无差异，它们都定义了特定对象的行为模式。

使用前面介绍过的继承机制，很容易就能在 Lua 中建立原型。更具体地说，如果我们有 a 和 b 两个对象，要想让 b 成为 a 的原型，只需要像下面这样：

```
setmetatable(a, {__index = b})
```

这样，调用 a 中任何不存在的成员都将在对象 b 中查找。将 b 视为对象 a 的类只不过是术语上的改变而已。

回到前面银行账号的例子上，为了使新创建的帐号对象都拥有与 Account 相似的行为，我们将利用 __index 元方法，使新对象继承 Account 的特性。这里有一个小的优化：我们不需要创建一个额外的表作为帐号对象的元表，我们可以用 Account 表本身作为元表：

```
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，内部的 self 等同于 Account 自身，因此我们可以直接使用 Account 取代 self。然而，使用 self 对于我们将在下一节介绍的类继承更为合适。有了上述代码之后，如果创建一个新的账号并调用它的一个方法时，会发生什么呢？

```
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 作为它的元表（调用 Account:new 时，self 即 Account）。而当我们调用 a:deposit(100.00)，实际上我们调用的是 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 方法，因此它将在元表的__index 域中查找，情况大致如下：

```
getmetatable(a).__index.deposit(a, 100.00)
```

a 的元表是 Account，Account.__index 也是 Account（因为在 new 函数中执行了 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 参数并调用了原始的 deposit 函数。所以，新对象 a 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，而且对新帐号对象中所有缺失的域都有效。因此，一个类不仅可以给它的实例提供方法，也为实例的域提供了默认值。别忘记在第一个 Account 定义中，我们提供了默认值为 0 的 balance 域，所以，如果创建了一个新的账号而没有提供 balance 的初始值，它将继承这个默认值：

```
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 即 b 本身，表达式 b.balance 等于 0，且存款的初始值，也即 v 被赋给了 b.balance。下一次我们访问这个值的时候，不会再涉及到__index 元方法，因为 b 已经拥有了自己的 balance 域。

16.2 继承

在 Lua 中，类是用对象来描述的，它们也是对象，因此也能够从其他类获得方法。这使 Lua 类的继承（这里的继承指面向对象普遍意义上的继承概念）变得非常简单。

假设存在一个基本的 Account 类：

```
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
```

从这个基本类出发，我们将派生一个子类 SpecialAccount，这个子类允许帐号的所有者能够从帐户内不受余额限制地提取存款。我们将从一个空类开始，它只是简单地从基类继承所有的操作：

```
SpecialAccount = Account:new()
```

到现在为止，SpecialAccount 仅仅是 Account 的一个实例。接下来将发生奇妙的事：

```
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，SpecialAccount 成为 s 的元表，SpecialAccount 的__index 域也指向了 SpecialAccount。这样，s 继承了 SpecialAccount，而 SpecailAccount 继承了 Account。当我们执行：

```
s:deposit(100.00)
```

Lua 在 s 中找不到 deposit 域，因此它会在 SpecialAccount 中查找，然而在 SpecialAccount 中同样找不到 deposit，最终，它在 Account 中查找并在 Account 中找到 deposit 的原始定义。

SpecialAccount 特殊之处在于，它可以重新定义任何从父类 Account 中继承的方法。需要做的仅仅是在 SpecialAccount 内部编写一个新的方法：

```
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 s:withdraw(200.00)，Lua 再也无须于 Account 中查找该方法的定义，因为它马上就在 SpecialAccount 内发现了新定义的 withdraw，由于 s.limit 的值为 1000.00（注意在创建 s 的时候初始化过这个值），程序执行了取款操作，而 s.balance 变成了负值。

Lua 中面向对象的编程存在着一个有趣的特点：你不需要创建一个新的类来指定新的行为模式。如果一个对象需要具备某种特殊的行为模式，你可以直接在该对象中实现它。例如，如果账号 s 需要表示一些取款额度为存款的 10%的特殊的客户，你只需要修改 s 这个单独的账号：

```
function s:getLimit()
```

```
    return self.balance * 0.10
end
```

经过这个声明之后，s:withdraw(200.00)被调用后将执行 SpecialAccount 的 withdraw 方法，但当 withdraw 调用 self:getLimit 时，上述新定义的 getLimit 方法将被执行。

16.3 多重继承

由于 Lua 中的对象不是元生（Primitive）的，即它不是语言自带基本类型，所以在 Lua 中有很多可以实现面向对象程序设计的方法。前面介绍的使用__index 元方法的方案，在综合考虑简洁性、性能以及灵活性等各方面的因素，可能是最好的方案之一。不过，在某些特殊情况下可能需要另外一些更为合适的实现方式，下面我们将看到在 Lua 中实现多重继承的方法。

实现的关键在于__index 元方法对应的函数。注意，当一个表的元表的__index 域存在对应的函数时，如果 Lua 调用一个原始表中不存在的函数，__index 域对应的函数将被直接调用。如此一来，也可一令__index 对应的函数在多个父类中不存在的域。

多重继承意味着一个子类拥有多个父类，所以，不能像前面那样用父类的方法去创建子类，而是定义一个以父类为参数的特殊函数来完成此功能。该函数将为新类创建一个表，并设定该表的元表的__index 元方法令其负责多重继承。尽管是多重继承，每一个实例仍属于特定的类，所以，子类和父类之间的关系与类和实例之间的关系是不同的。特别是一个类不能既是其实例的元表又是自身的元表。在下面的实现中，我们将一个类作为其实例的元表，并创建另一个表作为该类的元表：

```
-- 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
```

下面通过一个简单的例子说明 createClass 函数的使用方法，假定前面定义过的 Account 类以及含有两个方法的另一个类 Named，这两个方法分别为 setname 和 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 Account 和 Named 两个类的子类，我们只需调用 createClass：

```
NamedAccount = createClass(Account, Named)
```

要创建和使用 NamedAccount 的实例，只需要像往常那样：

```
account = NamedAccount:new{name = "Paul"}
print(account:getname())           --> Paul
```

让我们分析一下上例中的最后一个语句。显然，Lua 在 account 对象中找不到 getname 方法，因此它将查询 account 对象的元表（即 NamedAccount）的__index 域，不幸的是 NamedAccount 也没有提供 getname 方法，Lua 继续查询 NamedAccount 的元表的__index 域，并发现该域对应的函数可用，于是 Lua 调用这个函数。这个函数首先在 Account 类中查找 getname 方法无果，最终在 Named 类中找到该方法并作为 search 函数的返回值被返回。

当然，由于搜索函数 search 的复杂性，多重继承的运行效率相对于单继承要低。一个简单的改善性能的方法是，将子类从父类继承的方法拷贝到子类。使用这种技术，__index 元方法将变成：

```
...
setmetatable(c, {__index = function(t, k)
    local v = search(k, arg)
    t[k] = v                       -- save for next access
    return v
end})
...
```

应用这个技巧之后，访问继承的方法就如同访问局部方法一样快（除了第一次访问会比较慢之外）。这么做的缺点是系统运行之后将很难改变方法的定义，因为这种改变无法影响继承链的下端。

16.4 私有化

很多人认为成员私有化是面向对象语言整个体系的一部分，每个对象的状态只是该对象自己的事情。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成员方法在外部是否可见，其

他一些语言，比如 Smalltalk，将所有的成员变量私有化，并将所有的成员方法公有化，而第一个面向对象的语言 Simula 并不提供任何保护机制。

从前面有关面向对象设计的例子中可以看到，Lua 并没有提供私有化机制。其部分原因是，Lua 使用通用数据结构（Lua 表）来表示对象，但这也反映了隐藏在 Lua 背后的设计思想，Lua 没有打算被用来进行大型的程序设计，相反，Lua 定位于中小型程序设计，通常是作为大型系统的一部分，常用于一个或少数程序员开发的情况，甚至是由非程序员进行的开发。所以，Lua 避免了过分的冗余和过多的人为限制。如果你不希望一个对象的内部结构被访问，那么就不要去访问它们。

然而，Lua 的另一个目标是灵活性，它为程序员提供了元机制（Meta-Mechanisms），通过它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，但我们可以用不同的方式来实现对象，从而令它能够限制访问。虽然这种实现并不常用，但知道它也是有益的，不仅是因为它展示了 Lua 中一些有趣的方面，也因为它可能是某些其他问题很好的解决方案。

这里的基本设计思想是，每个对象用两个表来表示：一个描述状态，另一个描述操作（或称为接口），对象本身通过第二个表来访问，也就是说，通过接口（包括所有的操作）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，该函数创建一个表用来描述对象的内部状态，并保存于局部变量 self。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，在外部对象中将其方法映射到函数内已经实现的方法。这儿的关键点在于：这些方法没有使用额外的参数 self，而是直接访问 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，这个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
```

这种设计令任何存储在 self 表中的成员都是私有的，函数 newAccount 返回之后，没有任何方法可以直接访问对象，我们只能通过 newAccount 函数内部定义的函数来访问对象。虽然我们的例子中仅仅将一个成员变量放到私有表中，但是实际上我们可以将对象的任何的成员放到该私有表中。我们也可以定义私有方法，它们看起来像公有的，但我们并不将其放到公有的接口中。例如，我们的账号可以给某些用户享有额外 10%的信用额度，但是我们又不想用户直接访问这种计算功能，那么实现方式如下：

```
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再直接访问 extra 函数了。

16.5 单方法对象的实现

上述用于实现面向对象的程序设计的方法有一种特殊情况，那就是，当对象只有唯一的方法时。这种情况下，我们不需要创建一个接口表，而是将这个方法作为对象返回。这听起来有些不可思议，如果需要可以复习一下 7.1 节，在那里我们介绍了如何构造迭代函数来保存闭包的状态。实际上，一个能够保存状态的迭代函数正是单方法（Single-Method）对象。

关于单方法对象的一个有趣的情况，也就是当这个唯一方法的功能是基于不同的参数而执行不同任务时。针对这种单方法对象的一个可能的实现方法如下：

```
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
```

它的使用很简单：

```
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)长几个字符而已。每一个对象都使用一个单独的闭包，代价比起表来要小得多。这种方式实现的对象无

法继承但有完全的私有特性：访问对象状态的唯一方式是通过它的内部方法。

Tcl/Tk 的部件（Widget）使用了相似的方法，在 Tk 中，一个部件的名字表示一个能在该部件上执行各种操作的函数（部件的指令）。

第 17 章 Weak表

Lua 的内存管理是自动进行的。程序只能创建对象（表、函数等），而没有删除对象的功能。Lua 通过垃圾收集（Garbage Collection）机制自动删除那些失效的对象，这将使你从绝大部分内存管理的负担中解脱出来，而且更重要的，它可以让你从那些由此引发的程序错误中解脱出来，比如无效的指针（Dangling Pointer）和内存溢出（Memory Leak）等。

与其他的收集器不同，Lua 的垃圾收集器不存在循环的问题。使用循环性的数据结构时，你无须执行特殊的操作；它们会像其他数据一样被收集。不过，有时候即使更智能的收集器也需要你的帮助，没有一种垃圾收集器能够帮你处理所有的内存管理问题。

垃圾收集器不知道你对垃圾的定义，它只能收集它自己认为的垃圾数据。典型的例子就是栈：由一个数组和指向栈顶的索引构成。只有你自己知道数组中的有效部分位于栈顶索引以下，而 Lua 对此全不知情，如果你只是令栈顶索引减 1 来完成“栈内元素出栈”的操作，那么“已经出栈的” 数组元素对 Lua 来说并不是垃圾数据。同样地，对于 Lua 而言，任何储存于全局变量的对象都不是垃圾数据，即使你在程序再也不使用它们。在以上两种情况中，你都应当为含有垃圾数据的变量赋空值，否则无用的数据将无法释放其占用的内存空间。

不过，只是简单地用赋空值的方式来清除引用有时是不够的，对于有些数据结构的回收，你还需要与垃圾收集器进行额外的交流。一种典型的情况是，当你需要在一个集合内保存某种对象（比如，文件等）时。看起来可能很简单：你只需要将新对象插入集合内。但是，该对象一旦进入了集合内部，它将再也不会被垃圾收集器回收！因为就算没有任何指针指向这类对象，实际上集合本身引用了该对象，Lua 会认为这个引用是为了阻止该对象被回收，除非你告诉 Lua 该对象是可以被回收的。

Weak 表（Weak Table）是一种用来告诉 Lua 一个引用不应该防止被其引用的对象被回收的机制。一个 Weak 引用（Weak Reference）是指一个不被 Lua 考虑的引用（对象的引用计数不会因为它而改变）。如果指向对象的所有引用都是 Weak 引用，则该对象将被垃圾收集器收集，而那些 Weak 引用也将通过某种方式被删除。Lua 通过 Weak 表来实现 Weak 引用：一个 Weak 表是指表内所有的引用都是 Weak 引用的表。这意味着，如果一个对象只被 Weak 表引用，那么它最终将被 Lua 收集。

表由索引和值组成，而这两者都可能包含任何类型的对象。在普通情况下，垃圾收集器并不会回收作为索引和值的对象，也就是说，索引和值都属于强引用（Strong Reference），因为它们可以防止由其引用的对象被回收。在一个 Weak 表中，索引和值可以是 Weak 性质的，这意味着存在着三种类型的 Weak 表：由 Weak 索引组成的表；由 Weak 值组成的表；以及纯 Weak 表类型，这种表的索引和值都是 Weak 性质的。不管表为何种类型，当一个索引或值被回收时，整个记录（Entry）都将从表中消失。

表的 Weak 性质由它元表的__mode 域来指定。这个域的值必须以字符串的形式存在：如果其值为小写字母“k”，则该表的索引具有 Weak 属性；如果其值为小写字母“v”，则该表的值具有 Weak 属性。下面的例子，虽然是虚构的，但是可以阐明 Weak 表的基本特性：

```
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 = {}覆盖了表 a 中的第一个索引。当垃圾收集器工作时，会发现第一个索引不再被引用，所以该索引将被回收，同时表中与之对应的记录也将被回收。然而，第二个索引，仍然被变量 key 占用着，所以它不会被回收。

要注意，只有对象才可以从一个 Weak 表中被收集，类似数值和布尔类型的值，是无法被回收的。例如，在表中插入一个数值型的索引（参考前面的例子），那么它将永远不会被垃圾收集器从表中移除。当然，如果对应于这个数值型索引的值被收集，那么整个记录将从 Weak 表被移除。

字符串存在一些细微的差别：尽管字符串是可以被回收的，但是从实现角度看，它们与其他可回收的对象又有所不同。其他对象，比如表和函数，它们是显式地被创建的，当 Lua 遇到“{}”时，它将建立一个新表，当遇到 function()...end 时则建立一个新函数（实际上，是一个闭包）。然而，当 Lua 遇到"a".."b"的时候会创建一个新的字符串吗？如果系统中已经存在一个字符串“ab”的话怎么办？Lua 会重新创建一个新的字符串吗？编译器可以在程序运行之前创建字符串么？实际上这些都无关紧要：它们只是语言实现的细节。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 表中被移除（除非它所关联的值被回收）。

17.1 记忆函数

一个惯用的编程技术就是用空间换取时间。你可以通过记忆函数的结果来加速函数的执行速度，之后当你用同样的参数再次调用函数时，它可以重用上次计算的结果。

假设有一个通用服务器接收一个包含 Lua 代码的字符串请求，每收到一个请求，它就调用 loadstring 函数来加载字符串，并执行该字符串中的代码。然而，一方面，loadstring 是一个开销巨大的函数，另一方面，某些命令在服务端经常性地被调用，因此，可以令服务端通过一个辅助表来“记忆”上次调用 loadstring 函数的结果，而不是反复用 loadstring 调用某些常用的命令（诸如 closeconnection()等）。之后，每次在调用 loadstring 之前，服务端都将通过辅助表检查该字符串是否已经被加载，如果没有发现该字符串，服务端调用 loadstring 加载字符串并将其存入辅助表。我们可以使用下列函数来表述上面的功能：

```
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非常巨大，而且它可能会导致无法预料的存储浪费，尽管某些命令被一遍又以便地重复执行，但另外一些命令可能仅仅运行了一次。这样服辅助表内逐渐积累了服务端接收到的

所有命令以及与命令相对应的字符串代码，最终，服务端的内存将被消耗殆尽。Weak 表为这个问题提供了一个简单的解决方案，如果辅助表内的值为 Weak 类型，那么它们在每一个垃圾收集周期都将从辅助表中被移除：

```
local results = {}
setmetatable(results, {__mode = "v"})     -- make values weak
function mem_loadstring (s)
    ...                                   -- as before
```

事实上，表的索引经常是字符串类型的，如果有必要，我们可以令该表为纯 Weak 类型：

```
setmetatable(results, {__mode = "kv"})
```

这样做的最终结果与前者并无二样。

记忆技术在确保某些对象的唯一性上同样非常有用。例如，假如一个系统用表描述将颜色值，其中，颜色值由 RGB 颜色体系的红、绿、蓝三色来表示。一个简单的能够按请求产生颜色值的函数如下：

```
function createRGB(r, g, b)
    return {red = r, green = g, blue = b}
end
```

使用记忆技术，我们可以重用已经产生过的颜色值，因此需要替每一种颜色产生唯一的索引，为此，可以使用一个分隔符将红、绿、蓝三色简单地组合成为索引：

```
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
```

使用该技术后，产生一个有趣的结果：可以使用普通的等号比较符来比较颜色值相等与否，因为同种颜色在同一时期必定对应与同一个代表该颜色值的表。需要注意的是，同种颜色在在不同的时期可能对应于不同代表该颜色值的表，该种颜色可能是在被垃圾收集器回收之后重新产生的（因此该表并不是之前的表）。然而，只要给定的颜色值正在被使用，它就不会被移除。因此，只要某种颜色值一直处于使用中，那么用以表示该颜色值的表也将一直处于可用状态。

17.2 对象属性

Weak 表的另一重要应用就是用以关联对象及其属性。很多情况下我们都需要将某些属性与对象关联起来：名字与函数关联、默认值与表关联、数组大小与数组关联等。

当对象是表的时候，我们可以使用一个合适的唯一索引将某个属性保存在该表中，在上文介绍过，创建一个唯一索引的有一个简单且不会出错的方法，那就是创建一个新对象，并将其作为索引。但是，如果对象不是表，那么它就无法保存其自身的属性，有时候即便是对于表，我们也不想将属性存储于自身内部。比如，我们可能希望该属性是私有的（无法从外部对其进行访问），或者我们不想该属性在遍历

表的时候产生干扰效果，在这类情况下，需要一个能够将属性与对象关联起来的替代方案。当然，一个外部表为属性与对象的关联提供了一个理想的解决方法（正因为如此，Lua 表才时常被称为关联数组），在这个外部表种，我们将对象作为索引，而将其属性作为值。一个外部表可以将任何类型的对象作为属性（因为 Lua 允许任何类型的对象作为索引），另外，保存于外部表的属性并不会干扰对象，同时也能像这个外部表一样具有私有性质。

然而，这个看似完美的方案有一个巨大的缺点：一旦我们使用一个对象作为索引，该对象将永久存在而不会被回收。因为 Lua 不会回收一个正被用于索引的对象。例如，假设我们使用一个普通的表来关联函数名字，那么这些函数永远都不会被回收。正如你所预料的，我们可以使用 Weak 表来避免这个问题，而这一次，我们需要 Weak 索引，一旦对象不被引用，Weak 索引可以防止对象无法回收的问题。另一方面，该表不能使用 Weak 值，否则，正在被使用的对象将有可能被回收。。

Lua 语言自身就使用这种技术来记录用以保存数组的表的大小。后面我们还将看到，Lua 的 table 库提供了可以设置数组大小的函数和可以取得数组大小的函数。当你为一个数组设定了大小，Lua 将这个大小信息存储于一个私有的 Weak 表，其中，索引就是数组本身，而值就是该数组的大小。

17.3 重述带有默认值的表

在 13.4.3 章节，我们探讨过如何实现拥有非空值的表，介绍了一种具体的实现技术并提到另外两种需要借助 Weak 表来实现的技术，在介绍过 Weak 表之后，现在是该介绍它们的时候了。我们将看到，这两种用以实现默认值的技术，实际上是前面介绍的两种通用技术的特殊应用：对象属性与记忆函数。

在第一种解决方案中，我们使用 Weak 表来关联一个表及其默认值：

```
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 defaults 表没有 Weak 索引，它就会令所有带有默认值的表成为永久活动的对象（因此无法被垃圾收集器回收）。

在第二种方案中，我们使用不同的元表来保存不同的默认值，而当某一个默认值需要重复使用的时候，其对应的元表也将被重复使用。这是记忆函数的一个典型应用：

```
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
```

上述方案中，我们使用了 Weak 值，这将将导致不再需要的元表被回收。

上述哪种方案更好？这通常取决于具体情况，因为两者都具有类似的复杂度和性能。在第一种方案中，每个默认值对应的表都需要占用若干字节的空间（defaults 表中的记录）。在第二种方案中，每个默认值需要占用更多字节的空间（一个新表、一个新闭包、metas 中的新增记录）。因此，如果你的程序中有成百上千个不同的默认值，则第二种方案将成为首选，而如果只有少数几个表共享相同的默认值，则还是应该选择第一种方案。

第三篇 标准库

第 18 章 数学库

在这一章（以及标准库这一部分的其他章节），我的主要目的不是对每一个函数给出完整地说明，而是告诉你相应的 Lua 库能够提供什么样的功能。为了能够清楚地说明问题，我可能会忽略一些无关紧要的函数选项或者函数行为模式的介绍，因此下文的主要之处在于激发你的好奇心，如果想要了解更多细节，请参阅 Lua 的参考手册。

数学库（Math 库）是一个标准的算术函数的集合，比如，三角函数库（sin、cos、tan、asin、acos 等），指数函数和对数函数（exp、log、log10），舍入函数（floor、ceil），此外还有 max、min 和变量 pi 等。数学库也定义了一个指数操作符“^”。

所有的三角函数都以弧度为单位标准（Lua 4.0 以前，三角函数以度数为标准）。你可以使用 deg 和 rad 函数在度数和弧度之间转换。如果你想在度数标准下使用三角函数，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 函数用来产生伪随机数，对其有三种调用方式：

1、不带参数，函数返回一个在[0,1]范围内实数类型的随机数

2、带一个整型参数 n，函数返回一个满足 1 <= x <= n 条件的整型随机数 x

3、带两个整型参数 a 和 b，函数返回一个满足 a <= x <= b 条件的整型随机数 x

你可以使用 randomseed 函数设置随机数发生器的种子，其唯一的数值型参数就是该种子。通常在启动的时候，程序会使用固定的种子来初始化随机数发生器，这意味着每次你运行程序，都将产生相同的随机数序列。就调试而言，这当然很有用，不过如果用于游戏，那么每次运行游戏都会面对一模一样的场景，一个通用的解决方案是使用系统当前时间来作为种子：

```
math.randomseed(os.time())
```

函数 os.time 返回一个表示当前系统时间的数值，通常是从新纪元开始到当前时间的秒数。

第 19 章 表库

表库（Table 库）由一些将 Lua 表作为数组来操作的辅助函数组成，其主要作用之一就是为数组的 size 属性给出一个合适的解释。同时它也提供了在链表中插入和移除元素的函数，以及对数组元素排序的函数。

19.1 数组大小

在 Lua 中，我们需要经常假设数组的结束位置位于：从数组头部开始搜寻，第一个空值之前的位置。这个约定有一个弊端：数组无法拥有空值元素。对大部分应用程序而言，这个限制并无妨碍，有其是当所有的数组元素都属于一个固定类型的时候。但有些时候我们需要能够支持空值元素的数组，在这种情况下，我们需要一种方法显式地保存数组的大小信息。

表库定义了两个用于操纵数组大小的函数：getn 用于返回数组的大小；setn 用于设置数组的大小。在上文介绍过，有这两个方法可以将属性与表关联起来：要么在该表的域中保存属性，要么使用一个独立的 Weak 表来完成关联的功能。两种方法各有利弊，因此表库同时采用了两者。

通常，调用 table.setn(t, n)会在内部的 Weak 表将 t 和 n 相关联，而调用 table.getn(t)将从这个内部表中取得与 t 相关联的值。然而，如果表 t 拥有一个值为数值类型且名为 n 的域，则 setn 将修改这个值，而 getn 将返回这个值。getn 函数还有一个选项：如果不能使用上述选项返回数组的大小，它就会遍历数组直到找到第一个空值元素。因此，你总能够在数组中使用 table.getn(t)来获得一个有意义的结果：

```
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
```

默认情况下，setn 和 getn 使用内部表来保存数组大小的信息。这是最简洁的选择，因为它不会使用额外的元素污染该数组。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中，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，它无法使用 setn 函数。另外一个好处在于，我们可以在数组创建的时候直接设置它的大小，正如上例所示。

成对地使用 setn 和 getn 函数来操纵数组的大小信息是个好习惯，即使你知道数组的大小信息在域 n 中。表库中的所有函数（sort、concat、insert 等）都遵循这个习惯。实际上，提供 setn 函数来改变域 n 的值只是为了与旧版的 Lua 兼容，这个特性可能在未来的版本中改变，因此为了安全起见，不要依赖于这个特性，而是总是用 getn 函数来获取由 setn 设置过的数组的大小。

19.2 插入和删除元素

表库提供了在链表的任意位置插入和移除元素的函数。table.insert 函数可以在数组中的指定位置插入一个元素，并将其后的所有元素后移。另外，insert 函数会改变数组的大小（通过使用 setn 函数）。例如，如果 a 是一个数组{10, 20, 30}，调用 table.insert(a, 1, 15)后，a 将变成{15, 10, 20, 30}。一个经常性的特殊用法是：如果不带位置参数调用 insert 函数，则该函数会在数组的最后位置插入元素（因此不需要移动任何元素）。下面的代码逐行读入程序，并将所有行保存在一个数组内：

```
a = {}
for line in io.lines() do
    table.insert(a, line)
end
print(table.getn(a))        --> (number of lines read)
```

table.remove 函数移除数组中指定位置的元素，返回这个元素，被移除的元素之后的所有元素前移，并改变数组的大小。如果不带位置参数调用 remove 函数，则它将删除数组的最后一个元素。

使用这两个函数，很容易实现栈、队列和双向队列。我们可以这样初始化这类结构：a = {}。一个压栈操作等价于 table.insert(a, x)，一个出栈操作等价于 table.remove(a)。从另一端插入元素可以使用 table.insert(a, 1, x)，从该端删除元素可以使用 table.remove(a, 1)。最后两个操作并不是特别有效率，因为它们必须移动元素。不过，因为表库的这些函数使用 C 语言实现，对于那些含有几百个元素的小数组而言，效率并不是太大的问题。

19.3 排序

另一个有用的函数是 table.sort，它他有两个参数：需要排序的数组以及排序函数。排序函数有两个参数，如果第一个参数排在第二个参数之前，则它必须返回真值。如果未提供排序函数，sort 使用默认的小于操作符“<”进行比较。

一个常见的错误是试图对表的索引进行排序。在一个表中，所有索引可以被视为一个集合，因此无序的。如果想对它们进行排序，就必须将它们复制到一个数组，然后对这个数组进行排序。下面是一个例子，我们将读取一个源文件并创建一个表，改表给出每个函数的名字以及该函数定义处的行号：

```
lines = {
    luaH_set = 10,
    luaH_get = 24,
    luaH_present = 48,
}
```

现在想以字母顺序打印这些函数名，如果你使用 pairs 函数遍历这个表，函数名出现的顺序将是随机的。不过，你不能直接对这些函数名进行排序，因为它们是表的索引。如果你将这些函数名存入一个数组，则就能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，数组同样是无序的，但是我们知道如何计数，因此只要使用排序好的索引访问数组就可以得到经过排序的值。这就是为什么我们一直使用 ipairs 函数遍历数组而不是 pairs 函数，因为前者利用诸如 1、2、……的索引顺序进行遍历，而后者使用自然的随机顺序。

除上述方法之外，还有一个更好的方案，我们可以编写一个根据索引顺序遍历表的迭代器。一个可选的参数 f 可以指定排序的方式。首先，它将已排序的索引存入一个数组，然后遍历这个数组，每一步都从原来的表中返回索引和值：

```
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
```

用这个函数，可以很容易地按照字母顺序打印这些函数名。执行下列循环：

```
for name, line in pairsByKeys(lines) do
    print(name, line)
end
```

其打印结果为：

```
luaH_get          24
luaH_present      48
luaH_set          10
```

第 20 章 字符串库

在纯 Lua 解释器中，对字符串操作的支持很有限，一个程序可以创建字符串以及连接字符串，但无法截取子串，检查字符串的长度，检测字符串的内容。而 Lua 的字符串库（String 库）将给予你完全操纵字符串的能力。

字符串库中的某些函数相当简单：string.len(s)用于返回字符串 s 的长度；string.rep(s, n)用于返回由 n 个子串 s 组成的字符串（基于测试的目的，你可能会使用 string.rep("a", 2 ^ 20)创建一个 1MB 大小的字符串）；string.lower(s)用于返回字符串 s 的小写版本，而其中的非大写字母将不会发生改变；而 string.upper 用于返回字符串的大写版本。就如典型的应用那样，如果你想排序一个由字符串组成的数组，而不想关心大小写字母的问题，那么你可以这么做：

```
table.sort(a, function(a, b)
    return string.lower(a) < string.lower(b)
end)
```

函数 string.upper 和 string.lower 都依赖于本地环境。所以，如果你的程序运行于 European Latin-1 字符集环境下，下列表达式将会变成这样：

```
string.upper("ação")            --> "AÇÃO"
```

调用 string.sub(s, i, j)可以截取字符串 s 中包含第 i 个字符到第 j 个字符的子串。在 Lua 中，字符串的第一个索引为 1，当然你也可以使用负索引，负索引从字符串的结尾向前计数：索引-1 指向字符串的最后一个字符，-2 指向字符串的倒数第二个字符，以此类推。所以，string.sub(s, 1, j)返回一个从字符串 s 头部开始直到第 j 个字符的子串，string.sub(s, j, -1)返回一个从字符串 s 第 j 个字符开始直到尾部的子串（如果不提供第三个参数，则默认为-1，因此 string.sub(s, j, -1)可以简写为 string.sub(s, j)的形式），而执行 string.sub(s, 2, -2)将返回一个去除字符串 s 的首字符和尾字符后的子串。

```
s = "[in brackets]"
print(string.sub(s, 2, -2))     --> in brackets
```

注意 Lua 中的字符串是常量。string.sub 函数其他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。一个常见的错误是想利用下面的代码来改变字符串：

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

上面的函数调用不会改变字符串 s 的值，如果你想修改一个字符串变量，则必须将新值重新赋值给这个字符串变量：

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

函数 string.char 和 string.byte 用于字符与字符的内部表示(一般为该字符的数值编码)之间的转换。函数 string.char 获取 0 个或多个整数，将每一个数值转换成字符，并返回一个由这些字符连接而成的字符串。函数 string.byte(s, i)可以将字符串 s 的第 i 个字符的转换成整数，其中第二个参数是可选的，缺省值为 1，也就是将字符串 s 的首字符转换成该字符的内部表示。在下面的例子中，我们假定字符使用的 ASCII 编码：

```
print(string.char(97))                          --> a
i = 99; print(string.char(i, i + 1, i + 2))     --> cde
print(string.byte("abc"))                       --> 97
print(string.byte("abc", 2))                    --> 98
```

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

上例的最后一行使用了负索引来指定字符串的最后一个字符。

函数 string.format 在字符串格式化方面，有其是用于格式化字符串输出的时候，称得上是一个强大的工具。该函数为可变参数函数，能够返回由其第一个参数指定格式的字符串，其中第一个参数就是所谓的格式化字符串。格式化字符串的规则与 C 语言中的 printf 函数类似：它包含普通的文本以及格式化指令，格式化指令用于控制后续参数在格式化字符串中的位置以及格式。最简单的格式化指令由控制符“%”和一个格式字符组成：其中 d 代表十进制数、x 代表十六进制数、o 代表八进制数、f 代表浮点数、s 代表字符串。在控制符“%”和格式字符之间可以包含其他选项，以便控制格式化的细节问题。拿一个十进制的浮点数作例子：

```
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>
```

上例中的%.4f 表示保留浮点数小数点后面 4 位小数； %02d 代表以固定的两位显示十进制数，不足的数位用 0 补足，而%2d 未指定填充的字符，因此位数不足时将以空格补足。要了解完整的格式化指令请参考 Lua 参考手册，或者直接参考 C 语言的参考手册，因为 Lua 通过直接调用标准 C 语言库来完成格式化的功能。

20.1 模式匹配函数

字符串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）以及 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（Regular Expression，也作 regexp）来进行模式匹配。其主要原因是代码大小：实现典型的符合 POSIX 标准的正则表达式需要编写超过 4000 行的代码。这比整个 Lua 标准库之和还大，相比而言，Lua 中模式匹配的实现只用了不足 500 行代码，当然，模式匹配也就无法包含 POSIX 的所能功能，不过，Lua 中的模式匹配功能还是足够强大的，它包含了一些使用标准 POSIX 规范不容易匹配的特性。

函数 string.find 的基本功能就是在目标串（Subject String）内搜索与指定模式匹配的串，如果找到匹配的串则返回它的位置，否则返回空值。最简单的模式就是一个单词，它仅仅匹配该单词本身。比如，模式“hello”仅仅匹配目标串中的“hello”，当找到与模式匹配的串时，函数将返回两个值：匹配串开始位置的索引和匹配串结束位置的索引。

```
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
```

如果匹配成功，string.sub 以 string.find 的返回值为参数返回已成功匹配的字符串（对简单模式而言，成功匹配的字符串的就是其本身）。

函数 string.find 的第三个参数是可选的，它表示函数在目标串中搜索的起始位置。当我们需要得到

目标串中所有匹配的子串的索引时，这个选项将非常有用，我们可以重复搜索新的匹配，每一次从前一次匹配的结束位置开始。下面的示例代码用一个字符串中所有换行符的位置信息构造一个表：

```
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
```

稍后我们还将利用 string.gfind 迭代器来简化上述循环。

函数 string.gsub 有三个参数：目标串、模式串、替换串。它基本作用是替换目标串中所有匹配的串：

```
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
```

该函数的第四个参数是可选的，它用来限制替换的次数：

```
s = string.gsub("all lii", "l", "x", 1)
print(s)                                          --> axl lii
s = string.gsub("all lii", "l", "x", 2)
print(s)                                          --> axx lii
```

函数 string.gsub 的第二个返回值表示替换操作的次数。例如，下面代码可以方便地计算出一个字符串中空格出现的次数：

```
_, count = string.gsub(str, " ", " ")
```

注意，此处的“_”只是一个哑元变量。

20.2 模式

你可以在模式串中使用更有用的字符类（Character Class），字符类指模式串中一个能够匹配特定字符集的模式项。比如，字符类%d 可以匹配任意数字，因此你可以使用“%d%d/%d%d/%d%d%d%d”模式来匹配“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
```

下表列出了所有的字符类：

```
.           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          字符“\0”
```

上述字符类的大写形式表示小写形式所代表的集合的补集。例如，“%A”表示非字母的字符：

```
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，而是 gsub 函数的第二个返回值，它代表替换发生的次数。其他的打印 gsub 结果的例子可能会忽略这个返回值。

在模式匹配中有一些具有特殊意义的字符，被称作魔术字符，它们分别是：

```
(   )   .   %   +   -   *   ?   [   ^   $
```

字符“%”是魔术字符的转义字符，因此，“%.”匹配一个点、“%%”匹配字符“%”本身。转义字符“%”不仅可以用来转义魔术字符，还可以用于所有的非字母类型的字符，如果对某个字符有疑问，你可以使用转义字符确保该字符的意义。

对 Lua 而言，模式只是普通的字符串，它们和其他的字符串别无二致，因此也没有任何特殊待遇。只有当它们被相关函数视为模式时，“%”才成为转义字符。因此，如果想要在一个模式中使用引号，你必须使用在其他的字符串中使用引号的相同方式进行处理，也就是 Lua 中的转义字符“\”来转义引号。

你可以通过使用方括号将字符类或字符括起来的方式创建自己的字符集（Char-Set），比如，“[%w_]”可以匹配字母、数字和下划线、“[01]”可以匹配二进制数字、而“[%[%]]”能够匹配一对方括号。下面的代码能够统计文本中元音字母的个数：

```
_, nvow = string.gsub(text, "[AEIOUaeiou]", "")
```

你也可以在字符集中使用字符范围，也就是将字符范围内的第一个字符与最后一个字符用连字符连接起来，这样就可以表示由处于这两个字符之间的字符组成字符集。不过你将几乎用不上这个功能，因为大部分常用的字符范围都已经预定义过，比如，“%d”等价于“[0-9]”、“%x”等价于“[0-9a-fA-F]”，不过，如果查找八进制数字，你可能更喜欢使用“[0-7]”而不是 “[01234567]”。你可以在模式的开始处使用“^”来表示其补集：“[^0-7]”匹配任何不是八进制数字的字符、“[^\n]”匹配任何非换行符的字符。注意，你可以使用大写的字符类表示其补集，因为“%S”比“[^%s]”要简短些。

Lua 的字符类依赖于本地字符集环境，所以“[a-z]”可能与“%l”表示的字符集不同。在某些情况下，后者可能包括“ç”和“ã”等字符，而前者没有。因此，应该尽可能地使用第二种方式，除非你有明确理由不这么做：因为后者更简洁、更具可移植性、也更为高效。

可以使用修饰符令模式具备重复以及可选的能力，Lua 中的模式修饰符有以下四个：

```
+           匹配前一字符 1 次或多次
*           匹配前一字符 0 次或多次（最长匹配）
-           匹配前一字符 0 次或多次（最短匹配）
?           匹配前一字符 0 次或 1 次
```

修饰符“+”可以匹配一个或多个字符，总是获取能够匹配的最长的字符串。比如，模式“%a+”能够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
```

模式“%d+”能够匹配一个或多个数字：

```
i, j = string.find("the number 1298 is even", "%d+")
print(i, j)          --> 12    15
```

修饰符“*”与“+”类似，不过它还能匹配一个字符的 0 次重复，其典型的应用是匹配模式之间空白。比如，为了匹配一对圆括号之间的空白，可以使用“%(%s*%)”模式，其中“%s*”用来匹配 0 个或多个空白，圆括号在模式中有特殊的含义，所以我们必须使用“%”对其进行转义。再看一个例子，模式“[_%a][_%w]*”匹配 Lua 程序中的标识符：以字母或下划线开头的字母、下划线或数字序列。

修饰符“-”与“*”一样，都匹配一个字符类的 0 次或多次重复，但是它总是匹配最短的字符串。某些时候这两个修饰符没有太大区别，但是通常而言，使用它们将带来迥然不同的结果。比如，假设你使用模式“[_%a][_%w]-”来查找标示符，你将只能找到第一个字母，因为“[_%w]-”将永远匹配空，另一方面，假设你需要查找 C 程序中的注释，很多人可能会使用“/%*.*%*/”来进行匹配（也就是说以“/*”开头，后面跟着任意多个字符，而这任意多个字符以“*/”结尾），但是，由于“.*”进行总是匹配最长的字符串，因此，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，它会匹配从“/*”开始到第一个“*/”之间的部分：

```
test = "int x; /* x */  int y; /* y */"
print(string.gsub(test, "/%*.-%*/", "<COMMENT>"))
    --> int x; <COMMENT>  int y; <COMMENT>
```

修饰符“?”匹配一个可选的字符。举个例子，假设需要在一段文本中查找一个整数，而该整数可能带有正负号。那么模式“[+-]?%d+”能够符合我们的要求，它可以匹配像“-12”、“23”或“+1009”等字符串。“[+-]”是一个匹配“+”或“-”的字符类，其后续的“?”修饰符意为：匹配前面的字符类 0 次或 1 次。

与其他系统不同的是，Lua 中的修饰符不能作用于字符类；不能将模式分组然后使用修饰符作用于这个分组。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）。通常你可以使用一些高级技术绕开这个限制，在下文将对此有所介绍。

以“^”开头的模式只能匹配目标串的开始部分，与此类似，以“$”结尾的模式只能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 ...
```

上述代码将检查字符串 s 是否以数字开头，而下列代码：

```
if string.find(s, "^[+-]?%d+$") then ...
```

将检查字符串 s 是否是一个整数。

另一个模式项是“%b”，它可以匹配对称的字符串。这类模式项通常为“%bxy”，其中的 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()”可以匹配以“(”开始，以“)”结束的字符串：

```
print(string.gsub("a (enclosed (in) parentheses) line", "%b()", ""))
    --> a  line
```

使用这种模式项的模式有：“%b()”，“%b[]”，“%b%{%}”和“%b<>”。当然，你也可以使用任何字符作为前后的分隔符。

20.3 捕获（Capture）

捕获是这样一种机制：它可以使用模式的一部分匹配目标串的一部分，并将匹配的部分留做后用。将你想捕获的模式用圆括号括起来，就指定了一个捕获。

在 string.find 函数中使用捕获的时候，函数会返回捕获的值作为额外的结果。这常被用于拆分一个目标串：

```
pair = "name = Anna"
_, _, key, value = string.find(pair, "(%a+)%s*=%s*(%a+)")
print(key, value)           --> name    Anna
```

模式项“%a+”表示非空的字母序列，“%s*”表示 0 个或多个空白。在上述例子中，整个模式代表着：一个字母序列之后，紧跟着任意多个空白、一个“=”字符、任意多个空白和另一个字母序列。两个字母序列都是使用圆括号括起来的子模式，成功匹配的时候，它们就会被捕获。当匹配发生的时候，find 函数总是先返回匹配串的索引（在例子中被存储于哑元变量中），然后再返回子模式匹配的捕获部分。下面的例子情况非常类似：

```
date = "17/7/1990"
_, _, d, m, y = string.find(date, "(%d+)/(%d+)/(%d+)")
print(d, m, y)       --> 17    7    1990
```

我们也可以在模式本身中使用捕获，在一个模式中“%d”（d 代表数字）表示第 d 个捕获的拷贝。在通常的应用中，假设你想查找一个字符串中单引号对或者双引号对之间的子串，你可能会使用模式“['"].-['"]”，但这个模式在处理类似“it's all right”这样的字符串时会出问题。为了解决这个问题，可以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在函数 gsub 的替换串参数中。与其他模式一样，gsub 的替换串参数可以包含“%d”，当替换发生时它被转换为对应的捕获值（值得一提的是，正是由于捕获，替换串中的字符“%”必须用“%%”表示）。下面的代码示例，将复制字符串中的每一个字母，并用连字符连接被复制的字母：

```
print(string.gsub("hello Lua!", "(%a)", "%1-%1"))
    --> h-he-el-ll-lo-o L-Lu-ua-a!
```

下面代码将相邻字符的位置互换：

```
print(string.gsub("hello Lua", "(.)(.)", "%2%1"))
    --> ehll ouLa
```

作为一个更为有用的例子，我们将编写一个格式转换器，它以 LaTeX 风格的命令作为目标串，命令格式形如：

```
\command{some text}
```

然后将其转换为 XML 风格的字符串：

```
<command>some text</command>
```

就上述说明，下面的代码可以实现这个功能：

```
s = string.gsub(s, "\\(%a+){(.-)}", "<%1>%2</%1>")
```

如果字符串 s 为：

```
the \quote{task} is to \em{change} that.
```

则调用 gsub 之后，将得到：

```
the <quote>task</quote> is to <em>change</em> 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
```

注意模式的精确用法，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，而“.-”匹配剩余部分。需要注意的是，gsub 将返回两个值，我们使用额外的圆括号丢弃多余的结果（即替换发生的次数）。

最后一个有关捕获值的应用可能是最为强大的。我们使用一个函数作为 string.gsub 的第三个参数调用 gsub，在这种情况下，string.gsub 每发现一个匹配时都会调用这个指定的函数，而捕获值将成为该函数的参数，这个函数最终的返回值将成为 gsub 函数的替换串参数。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 出现的地方替换为该变量的值：

```
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?
```

如果不能确定指定的变量为字符串类型，你可以使用 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
```

下面这个更为强大的例子使用 loadstring 函数来计算一段文本中以“$”为前缀并包含在一对方括号内的表达式的值：

```
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
```

第一次的成功匹配是“$[math.sin(3)]”，对应的捕获为“[math.sin(3)]”，调用 string.sub 移除该捕获首尾的方括号，因此最终被加载并执行的字符串是“return math.sin(3)”，另一个匹配“$[2^5]”的情况与此类似。

我们常常需要使用 string.gsub 遍历字符串，而对返回的字符串结果不感兴趣。比如，我们收集一个字符串中所有的单词并将其存入一个 Lua 表：

```
words = {}
string.gsub(s, "(%a+)", function(w)
    table.insert(words, w)
end)
```

如果字符串 s 为“hello hi, again!”，则上面代码的结果将是：

```
{"hello", "hi", "again"}
```

使用 string.gfind 函数可以简化上述代码：

```
words = {}
for w in string.gfind(s, "(%a)") do
    table.insert(words, w)
end
```

gfind 函数比较适合被用于范性 for 循环。它返回一个能够遍历所有模式对应的字符串的函数。

我们可以进一步的简化上述代码，调用 gfind 函数的时候，如果不显式地指定捕获，函数将捕获整个匹配模式。所以，我们可以重写上述代码为：

```
words = {}
for w in string.gfind(s, "%a") do
    table.insert(words, w)
end
```

在下一个例子中，我们将使用 URL 编码，URL 编码是 HTTP 协议用来发送参数的编码。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，而空白将被转换成“+”字符。比如，字符串“a+b = c”被编码后将成为“a%2Bb+%3D+c”。最后，将参数名和参数值之间“=”字符连接，并在“参数名-参数值”对之间加一个“&”。比如下列字符串：

```
name = "al"; query = "a+b = c"; q = "yes or no"
```

将被编码为：

```
name=al&query=a%2Bb+%3D+c&q=yes+or+no
```

现在，假如我们想要解码此 URL 编码，将每个参数值存入 Lua 表，并将参数名作为对应的索引。下面的函数实现了基本的解码功能：

```
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 匹配所有以“%”开头后跟两个 16 进制数的串，并调用一个匿名函数，该匿名函数可以将 16 进制数转换成一个数字（tonumber 函数，由于 16 进制情况下）,并返回对应的字符（string.char 函数）。比如：

```
print(unescape("a%2Bb+%3D+c"))         --> a+b = c
```

对于“名字-值”对，可以使用 gfind 进行解码，因为名字和值都不能包含“&”和“=”，因此我们可以用模式“[^&=]+”进行匹配：

```
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 函数可以匹配所有的“名字-值”对，对于每一个“名字-值”对，迭代器将返回对应的捕获值给变量 name 和 value。在循环体内，调用 unescape 函数对 name 和 value 进行解码，并将结果存入 cgi 表中。

与此解码相对应的编码也很容易实现。首先，我们编写一个 escape 函数，这个函数将所有的特殊字符转换成以“%”开头后跟该字符对应的两位 16 进制 ASCII 码（格式化指令“%02X”将使用 0 在不足两位的编码前补足），以及将空白转换为“+”字符：

```
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
```

函数 encode 遍历需要编码的表，并构造最终的结果串：

```
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 专业技巧

模式匹配是一个操纵字符串的强大工具，你可能只需要调用 string.gsub 和 find 函数就能完成复杂的操作，然而，你必须像使用任何强大的力量一样，谨慎地使用它。

对于某些解析程序而言，模式匹配并不能完全替代它们。在一个粗糙的程序中，你可以使用模式匹配在源代码上进行一些实用的操作，但是你很难利用它产生一个高质量的软件产品。前面介绍过的用于的匹配 C 程序中注释的模式就是个很好的例子：“/%*.-%*/”。如果目标串中包含了“/*”，则模式匹配将会遇到问题：

```
test = [[char s[] = "a /* here"; /* a tricky string */]]
print(string.gsub(test, "/%*.-%*/", "<COMMENT>"))
    --> char s[] = "a <COMMENT>
```

含有这类内容的字符串很罕见，因此如果只是供自己使用的话，上述模式尚能胜任，但是你不能将带有此类瑕疵的程序作为产品进行出售。

通常情况下，Lua 中模式匹配的效率是相当不错的：一台奔腾 333MHz 的计算机在 200K 字符的文本（约 30K 单词）内匹配所有的单词时只需要 1/10 秒。但你仍需小心，并总是尽可能具体地描述模式，一个含糊的模式比一个具体的模式要慢很多。一个极端的例子就是模式“(.-)%$”，它用于获取一个以“$”符号结尾的字符串，如果目标串中存在“$”符号，则程序还能正常运行。但如果目标串中不存在“$”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“$”符号，然后从目标串的第二个字符开始进行匹配依然没有找到匹配，以此类推，这将在时间方面产生巨大的开销，导致在一台奔腾 333MHz 的计算机上需要 3 个多小时来处理一个 200K 的文本串。可以使用下面的模式来避免上面的问题：“^(.-)%$”。定位符“^”通知该算法遇到第一个位置上的失败匹配就停止进行后续的匹配。使用定位符之后，同样的环境也只需要不到 1/10 秒的时间：

另外也需要注意空模式，即匹配空串的模式。比如，如果用模式“%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
```

上述例子中，函数 string.find 在目标串的开始处匹配了空字符串。

设计模式的时候，以修饰符“-”开头或结尾将毫无意义，因为它将只匹配空串，这个修饰符一般用于模式的中间部分，以两边的模式项来限制其扩展的方式。一个包含“.*”的模式同样容易被误解，因为它所能够匹配的范围远比你预想的要广。

有时候，可以利用 Lua 自身的功能来创建模式。在接下来的例子中，我们将匹配字符数大于 70 的文本行，也就是匹配一个含有多于 70 个非换行符字符的行。我们使用字符类“[^\n]”表示非换行符的字符，所以，可以使用这样模式：单个非换行符字符的模式重复 70 次，后面紧跟一个匹配上述字符 0 次或多次的模式。我们可以使用 string.rep 函数来代替手工输入：

```
pattern = string.rep("[^\n]", 70) .. "[^\n]*"
```

还有另外一个例子，假如你想进行字母大小写无关的查找，一个方法就是用字符类“[xX]”来替代字符“x”，即一个包含大写于小写字母的字符类。我们可以使用 Lua 函数进行自动的大小写转换：

```
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，而不关心其中的魔术字符。如果字符串 s1 和 s2 都是字符串，你可以为其中的魔术字符加上转义字符。但是如果这些字符串是变量的话，你可以再次使用 gsub 函数来转义转义字符本身：

```
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，我们只转义了转义符“%”本身。

另一个基于模式匹配的实用技术是在真正使用之前，对目标串进行预处理。一个预处理的简单例子是，将一段文本中的双引号内的字符串转换为大写，该字符串的两端各有一个双引号，且可能含有经过转义的双引号“\"”：

```
follows a typical string: "This is \"great\"!".
```

为了处理含有经过转义的双引号，我们需要对其进行编码预处理。比如，我们可以将“"”编码为“\1”，但如果原始的文本中已包含“\1”，则这种方案将无法奏效。这里有一个简单的方法，既可以对其进行编码又能避免此类问题，那就是将所有“\x”序列编码为“\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
```

现在，编码后的字符串中所有的“\ddd”序列必然是经过编码的，因为原始字符串中任何形为“\ddd”的序列已经被编码。因此，对已编码的串进行解码是很容易的：

```
function decode(s)
    return (string.gsub(s, "\\(%d%d%d)", function(d)
        return "\" .. string.char(d)
    end))
end
```

现在可以完成既定的任务了，因为经过编码预处理的字符串已经不再包含经过转义的双引号“\"”，我们可以直接用“".-"”来匹配用双引号引用的字符串：

```
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"!".
```

更紧缩的书写形式为：

```
print(decode(string.gsub(code(s), '(".-")', string.upper)))
```

接下来让我们回到前面的一个例子，来完成一个更为复杂的任务。那就是将\command{string}这种命令串转换为 XML 风格：

```
<command>string</command>
```

但这一次允许原始的命令串中包含反斜杠作为通用的转义符，这样我们就可以使用“\”、“\{”和“\}”，来分别表示“\”、“{”和“}”。为了避免模式匹配混淆命令串与经过转义的字符，首先需要将原始串中经过转义的序列进行重新编码，然而，在此我们不能像上例一样转义所有的“\x”序列，因为这

样会导致命令序列“\command”也被重新编码。因此，仅当 x 不是字母的时候才对“\x”进行重新编码：

```
function code(s)
    return (string.gsub(s, '\\(%A)', function(x)
        return string.format("\\%03d", string.byte(x))
    end))
end
```

用于解码的 decode 函数与上例类似，但是最终的字符串中将不包含反斜杆字符，因此可直接调用 string.char 函数：

```
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}.
```

作为最后一个例子，我们将接触到逗号分隔值（Comma-Sparated Values，CVS）格式，很多应用程序都使用这种格式，比如 Microsoft Excel 用这种格式来表示其单元格的值。CSV 文件是由多条记录组成的列表，每条记录写在一行，每行由以逗号分隔的字符串值组成，如果一个字符串值包含了逗号字符，则该字符串值必须在首尾使用双引号引用，如果该字符串值中还包含了双引号，则该双引号必须用一对双引号替代。下列数组：

```
{'a b', 'a,b', 'a,"b"c', 'hello "world"!', ''}
```

如表示成 CVS 格式，则为：

```
a b,"a,b"," a,""b""c",hello "world"!,
```

将一个类似的字符串数组转换为 CSV 格式相当容易，只需要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
```

如果其中的字符串包含逗号或引号，那么我们需要将其写入一对双引号之间，并转义其中原有的引号：

```
function escapeCSV(s)
    if string.find(s, '[,"]') then
        s = '"' .. string.gsub(s, '"', '""') .. '"'
    end
    return s
end
```

对 CSV 格式的文本进行解码并存入一个数组则稍微有点难度，因为我们必须区分位于引号中间的逗号和用于分隔字符串值的逗号。我们可以尝试转义位于引号对之间的逗号，然而并不是所有的引号都用于引用字符串值，只有逗号之后的引号才是引号对的头部（这里的逗号是用于分隔字符串值的逗号，它并不是字符串值的一部分）。有太多的细节需要注意，比如，两个引号可以表示单个引号，可以表示两个引号，还可以表示空串：

```
"hello""hello", "",""
```

上述例子中，第一部分表示字符串“hello"hello”，第二部分表示字符串“ ""”（一个空白加两个引号），而最后一部分表示一个空串。

我们可以多次调用 gsub 来进行处理，但可以通过一个遍历每个域的循环来更方便有效地解决问题。循环体的主要任务是查找下一个逗号，并将域的内容存入一个表中。对于每一个域，我们显式地检查其是否以引号开头，如果是，则用循环的方式查找末尾的引号，在该循环内可以使用模式“"("?)”来查找这对引号：如果一个引号后跟着另一个引号，则第二个引号将被捕获并赋给变量 c，这意味着该引号还一个引号对的尾部。

```
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 输入输出库

输入输出库（I/O 库）为文件操作提供两种模式。简单模式（Simple Model）的前提是一个当前输入文件和一个当前输出文件，且所有操作都是作用于这两个文件上的。完全模式（Complete Model）使用显式的文件句柄，且采用了面向对象的方式来定义作用于文件句柄的方法。

简单模式较适合于简单的文件操作，而且在本书的前面部分我们一直都在使用这种模式。不过对于高级的文件操作（例如，同时从多个文件读取数据）的时候，简单模式就显得力不从心了，因此，这时候使用完全模式较为合适。

输入输出库的所有函数都放在表 io 中。

21.1 简单模式

简单模式的所有操作都是作用于两个当前文件之上，输入输出库将当前输入文件初始化为标准输入（stdin），而将当前输出文件初始化为标准输出（stdout）。因此，当调用 io.read 这样的函数，我们就能够从标准输入中读取一行。

我们可以使用 io.input 函数和 io.output 函数来改变当前文件。调用 io.input(filename)就能以读模式打开指定的文件并将其设置为当前输入文件，之后所有的输入都将来自于该文件，直到再次调用函数 io.input 来改变当前输入文件。io.output 函数的原理与前者相同，只不过它影响的是当前输出文件。一旦发生错误，这两个函数都会抛出错误，如果你想直接控制错误，则必须使用完全模式下的 io.read 函数。

写操作比起读操作相对简单，因此我们先从写操作入手。io.write 函数能够获取任意个数的字符串参数，并将其写入当前输出文件，这时，数值参数将按照通用的转换规则转换成字符串，如果需要完全控制转换过程，那么可以使用字符串库中的 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
```

注意避免类似 io.write(a .. b .. c)这样的代码，io.write(a, b, c)以较小的代价完成了同样的效果，因为后者避免了字符串的连接操作。

按照惯例，你应该在初级程序或程序调试中使用 print 函数，而在需要完全控制输出的情况下使用 write 函数：

```
> print("hello", "Lua"); print("Hi")
    --> hello   Lua
    --> Hi

> io.write("hello", "Lua"); io.write("Hi", "\n")
    --> helloLuaHi
```

与 print 函数不同，write 函数不会在输出中附加额外的字符，例如制表符、换行符等。此外，write 函数使用当前输出文件，而 print 函数总是使用标准输出。最后，print 函数会自动为参数调用 tostring 函数，因此它可以打印 Lua 表、函数以及空值。

函数 read 从当前输入文件读取字符串，由它的参数来控制读取的内容：

"*all"	读取整个文件
"*line"	读取下一行
"*number"	读取一个数值
num	读取 num 个字符的串

调用 io.read("*all")函数将从当前位置读取整个输入文件，如果当前位置在文件末尾，或文件为空，函数将返回空串。

由于 Lua 可以对长字符串进行有效的管理，一个编写过滤器的简单技巧是：将整个文件读入到一个字符串中，处理该字符串（比如使用 gsub 函数进行类似处理），并将最终的字符串写入到输出：

```
t = io.read("*all")          -- read the whole file
t = string.gsub(t, ...)      -- do the job
io.write(t)                  -- write the file
```

下面是一个使用 MIME（多用途的网际邮件扩充协议）的 Quoted-Printable 编码来编码文件内容的完整程序。在 QP 编码中，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，为了保持一致性，“=”字符同样被要求进行编码。函数 gsub 中的模式捕获所有值在 128 到 255 之间的字符，并加上等号：

```
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 环境下转换一个包含 200K 字符的文件需要 0.2 秒。

调用 io.read("*line")可以返回当前输入文件中的下一行（不包含行末的换行符），如果已经到达文件末尾，该函数返回空值（因为不存在可以返回的下一行），这种读取方式是 read 函数的默认方式，所以 io.read()的效果与 io.read("*line")完全一样。通常，只有当程序需要逐行处理时才会采用这种方式读取文件，否则，我们更倾向于一次性读取整个文件，或逐块读取文件（稍后将会介绍）。下面的例子演示了这种读取模式的应用，该程序复制当前输入文件到当前输出文件，并标记各行的行号。

```
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
```

但是，如果要在整个文件中逐行迭代，最好还是使用 io.lines 迭代器。下面是一个对文件各行进行排序的完整程序：

```
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 环境下，该程序处理处理一个含有 32K 行大小为 4.5MB 的文件共耗时 1.8 秒，相比之下，高度优化的 C 语言 sort 程序需要 0.6 秒。

调用 io.read("*number")可以从当前输入文件中读取一个数值，这是唯一能够令 read 函数返回数值而不是字符串的情况。当需要从一个文件中读取多个数值时，数值间的空白字符串可以显著的提高执行性能。“*number”选项会忽略数值前的空白字符，可接受的数值形如-3、+5.2、1000 或-3.4e-23。如果在当前位置找不到可接受的数值（格式不对，或已经到达文件结尾），则返回空值。

你可以设置多个读取模式，对于每个参数，read 函数返回对应的结果。假设有个每行包含三个数值的文件：

```
6.0     -3.23      15e12
4.3     234        1000001
...
```

现在要打印出每行三个数值中的最大值，你可以调用 read 函数一次性读取每行的全部数值：

```
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，你都应该首先考虑选择使用 io.read 函数的“*.all”选项读取整个文件，然后使用 gfind 函数来对其进行分解：

```
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
```

除了基本读取模式之外，你还可以使用数值 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，如果无法读取任何字符（已经到达文件末尾），函数返回空值，否则，函数将返回一个最多包含 n 个字符的字符串。下面的程序可以有效地（当然是指在 Lua 中）将文件从标准输入拷贝到标准输出，该程序使用了这种读取模式：

```
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
```

这里有一个特例，io.read(0)可以用来测试是否已经到达了文件末尾：如果不是，则返回一个空串，否则，返回空值。

21.2 完全模式

想要得到全面的输入输出控制，你可以使用完全模式。文件句柄（File Handle）是完全模式的核心概念，文件句柄类似于 C 语言中的文件流（FILE *）：它表示一个已打开的文件及其当前存取位置。

你可以使用 io.open 函数打开一个文件，它类似于 C 语言中的 fopen 函数，它接受文件名以及打开模式作为参数。模式字符串可以是“r”（读模式）、“w”（写模式，将删除文件内的原有数据）或“a”（附加模式），可选的“b”选项用于以二进制方式打开文件。函数 open 返回指定文件的一个句柄，如果发生错误，则返回空值、错误信息以及错误代码：

```
print(io.open("non-existent file", "r"))
    --> nil    No such file or directory    2

print(io.open("/etc/passwd", "w"))
    --> nil    Permission denied    13
```

错误代码的具体意义由系统定义。

下面是一段典型的用于检查错误的代码：

```
local f = assert(io.open(filename, mode))
```

如果 open 函数失败，错误信息将作为 assert 的第二个参数，并由 assert 显示该错误信息。

文件打开后，就可以使用 read 和 write 方法对其进行读写操作，这两个函数与 io 表的 read 和 write 函数类似，但是调用方法上有所不同，前者必须使用冒号操作符，作为文件句柄的方法来调用。下面是一段打开文件并读取其内容的代码：

```
local f = assert(io.open(filename, "r"))
local t = f:read("*all")
f:close()
```

输入输出库同样也为 C 中的三种流定义了相应的句柄，它们对应于 C 中的：io.stdin、io.stdout 和 io.stderr。因此你可以利用下面的代码直接向错误流发送信息：

```
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()调用得到当前的输入文件句柄，使用带有参数的 io.input(handle)调用设置当前的输入文件句柄（同样的用法同样适用于 io.output 函数）。如果想要暂时地改变当前输入文件，可以使用下列代码：

```
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 一个优化技巧

在 Lua 中，比起逐行读取文件，一次性读取整个文件通常要快得多，但是，有时候会遇到几十几百兆的大文件，一次性读取整个文件显然并不明智。要在处理这类文件时获得最大的效率，最快的方法就

是逐次读取固定大小的文件块（比如，每次读取 8KB），与此同时，为了避免打断文件中的行，你可以在每次读取时再额外读取一行：

```
local lines, rest = f:read(BUFSIZE, "*line")
```

上例中的 rest 变量保存着被打断的行的剩余内容，之后我们可以将读取的文件块与 rest 连接起来，这样一来，最终得到的文件块将包含完整的行。

上述技术的一个典型应用就是对 wc 程序的实现，该段程序可以对输入文件中的字符、单词和行数进行计数：

```
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

简单模式下的 io.input 和 io.output 函数总是以默认的文本模式打开一个文件。在 Unix 系统中，二进制文件和文本文件并没有区别，但是在诸如 Windows 这样的系统中，二进制文件必须以一个特殊的标记来打开。处理这样的二进制文件，你必须将“b”选项附加到 io.open 函数的打开模式参数中。

在 Lua 中，二进制数据的处理类似于文本处理。一个 Lua 字符串可以包含任何字符，而 Lua 库中的绝大部分函数都能处理任意字符（你甚至可以对二进制数据进行模式匹配，只要模式中含有“\0”字符。如果要匹配数据中的“\0”字符，你可以使用字符类%z）。

通常，你会使用“*all”模式读取整个文件，或使用 n 模式从文件中读取 n 字节。下面是一个将文本文件从 DOS 模式转换到 Unix 模式的简单程序（即，将“回车换行符”替换成“换行符”），该程序不使用标准输入输出文件（stdin 和 stdout），因为这类文件是以文本模式打开的。程序假设输入文件和输出文件的文件名将以参数的形式传递进来：

```
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
```

下面来看另外一个例子。下列程序将打印二进制文件中的所有字符串，该程序假设一个字符串是以“\0”字符结尾的至少含有 6 个有效字符的序列，在这个例子中，有效字符由字母数字、标点符号以及空格组成。我们将使用字符串连接操作符和 string.rep 函数创建一个模式以便捕获所有含有至少 6 个有效字符（即 validchars 中的定义）的序列，模式中位于末尾的%z 将匹配该字符串末尾的“\0”字符。

```
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
```

接下来是最后一个例子。该程序对二进制文件进行倾倒（Dump）操作，同样地，程序的第一个参数是输入文件名，程序输出在标准输出上。程序每次读取 10 个字节作为一段，并输出该段中每个字节的十六进制表示，然后再以文本的形式输出该段，其中的控制字符将被转换为点号：

```
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
```

如果将该程序的脚本文件命名为 vip，那么可以使用下面的命令来倾倒该脚本本身：

```
prompt> lua vip vip
```

在 Unix 系统中（即 vip 为 Unix 格式的文件），它将会产生如下的输出：

```
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")).
...
```

```
74 65 73 2C 20 22 25 63 22 2C    tes, "%c",
20 22 2E 22 29 2C 20 22 5C 6E     "."), "\n
22 29 0A 65 6E 64                 ").end
```

21.3 文件的其他操作

函数 tmpfile 返回一个以读写模式打开的临时文件句柄，该临时文件在程序执行完毕后会被自动删除。函数 flush 即时刷新一个文件句柄的缓存，令所有待写入的信息立刻写入相应的文件，该函数的用法类似于 write 函数，你可以像调用函数一样调用它：io.flush()，也可以像调用对象方法那样：f:flush()，前者刷新当前的输出文件，而后者刷新句柄 f 所对应的文件。

函数 seek 可取得或和设置一个文件的当前存取位置，其一般形式为 filehandle:seek(whence, offset)。其中 whence 参数指明位置偏移的模式：如果是“set”，则位置偏移从文件头部开始，如果是“cur”，则位置偏移值从当前位置开始，如果是“end”，则位置偏移值从文件尾部开始向前计数。另外一个 offset 参数即偏移量。该函数最终返回从文件头部开始计算的当前存取位置。

参数 whence 的默认值为“cur”，而 offset 的默认值为 0。因此，调用 file:seek()将得到当前的存取位置，调用 file:seek("set")将令文件的存取位置定位到文件头部（返回值当然就是 0），而调用 file:seek("end")将令文件的存取位置定位到文件末尾，同时也得到了作为返回值的文件大小信息。下列代码在不改变当前存取位置的前提下取得了文件的大小信息：

```lua
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
```

在出错的情况下，上述函数将都返回空值和错误信息。

第 22 章 操作系统库

操作系统库（OS 库）包含了用于文件管理、取得系统日期时间以及其他牵涉到操作系统的函数，这些函数定义在表 os 中。定义该库时充分考虑了 Lua 的可移植性，因为 Lua 是以 ANSI C 编写的，因此只使用了 ANSI 定义的标准函数。而其他未包含在 ANSI 定义中的操作系统功能，例如目录管理、网络套接字等，都没有被应用到 Lua 的操作系统库里。另外一些没有被包含 Lua 主要发行版中库提供了扩展的操作系统功能，例如，posix 库提供了对 POSIX 1 标准的全面支持，luasocket 库提供了网络支持。

在文件管理方面，操作系统库只提供了 os.rename 函数，它可以对文件进行重命名操作，此外还有 os.remove 函数，它能够从磁盘上删除指定文件。

22.1 日期和时间

函数 time 和 date 在 Lua 中提供了日期时间的查询功能。

不带参数调用 time 函数时，函数以数值形式返回当前日期和时间（在大多数操作系统中，该数值即当前时间距离新纪元的秒数）。如果以一个 Lua 表作为参数调用该函数时，将返回一个用于表示改表所提供的日期时间的数值，这种表应该具有下列域：

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

上表列出的前三个域是必须的（因此具有上述特性的表也被成为日期表），如果后面的几个域未经定义，则默认为正午的时间 12:00:00。在里约热内卢（格林威治西向三个时区），假设在一个 Unix 系统（新纪元开始的时间为 1970 年 1 月 1 日，00:00:00 UTC）中执行如下代码，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 date 类似于 time 函数的反函数：它将一个表示日期时间的数值转换成更为高级的表现形式。date 函数的第一个参数是一个格式化字符串，该参数日期时间的描述方式，第二个参数就是用于表示日期时间的数值，其默认值为当前日期时间。

我们可以使用格式化字符串“*t”来创建一个日期表，如下面的代码所示：

```
temp = os.date("*t", 906000490)
```

上述代码将产生下列结果：

```
{year = 1998, month = 9, day = 16, yday = 259, wday = 4,
```

```
    hour = 23, min = 48, sec = 10, isdst = false}
```

注意上述表中的域，除 os.time 函数所需要的域之外，由 os.date 函数产生的日期表内还包含了工作日（wday，星期天为 1）以及年日（yday，一月一日为 1）。

除了“*t”之外，还可以在格式化字符串中使用其他特殊标记，os.data 函数将用日期时间信息对这些标记进行填充，从而得到一个表示日期时间的字符串。所有的特殊标以都是以“%”字符开头后跟一个字母的形式出现，看下面的例子：

```
print(os.date("today is %A, in %B"))
    --> today is Tuesday, in May
print(os.date("%x", 906000490))
    --> 09/16/1998
```

所有有关日期时间的表示形式都是以本地的区域设置为准，因此，如果本地操作系统的区域设置为 Brazil-Portuguese（巴西-葡萄牙语系），则“%B”将对应于“setembro”，而“%x”将对应于 “16/09/98”。

下表列出了所有的可用标记和它们的意义，以及它们对应于“1998 年 9 月 16 日，星期三，23:48:10”的值。对于数值形式的值，还列出了它们的取值范围：

%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 '%'

如果不带任何参数调用 date 函数，该函数就以“%c”格式输出，即完整的日期时间信息。需要注意的是，“%x”、“%X”和“%c”的表示由本地操作系统及其区域设置决定，如果需要某种固定的表示方式，例如，mm/dd/yyyy，你可以使用明确的格式化字符串，例如，“%m/%d/%Y”。

函数 os.clock 返回该程序执行时所花费的 CPU 时间（单位，秒），该函数常用于代码的性能评估：

```
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 其他系统调用

函数 os.exit 用于终止当前程序的执行。函数 os.getenv 用于取得环境变量的值，它接受环境变量名作为参数，以字符串形式返回该变量的值：

```
print(os.getenv("HOME"))                    --> /home/lua
```

如果该环境变量未经定义，则函数返回空值。函数 os.execute 执行一个系统命令，它等价于 C 语言中的 system 函数，该函数获取一个字符串形式的命令作为参数，并返回一个错误代码。例如，在 Unix 以及 DOS-Windows 操作系统中，可以使用下列代码创建一个新的目录：

```
function createDir(dirname)
   os.execute("mkdir " .. dirname)
end
```

函数 os.execute 功能强大，但是它相当依赖于操作系统。

函数 os.setlocale 设定 Lua 程序所使用的区域设置，区域设置定义了文化和语言的不同特性。函数 setlocale 有两个字符串类型的参数：区域名称和特性列表（后者指定了该区域设置的特性）。区域设置中含有六项特性：“collate”用于控制字母和字符串的排序方式、“ctype”定义了该区域设置中的字符集及其大小写的转换规则、“monetary”即货币设置，对 Lua 程序没有影响、“numeric”定义了数字的表示格式、“time”用于控制日期时间的格式（换言之，它将影响到 os.date 函数）、“all”包含了以上的所有特性。setlocale 函数使用的默认特性就是“all”，因此如果只使用区域名称作为参数来调用该函数，那么它设置所有的特性。该函数返回区域名称，但如果设置失败，它将返回空值（通常是因为操作系统不支持指定的区域）。

```
print(os.setlocale("ISO-8859-1", "collate")) --> ISO-8859-1
```

特性“numeric”有点令人混淆。尽管葡萄牙语和其他拉丁文语使用逗号代替点号来表示小数，但是区域设置并不会影响 Lua 解析数字的方式（作为众多原因中的一个：类似于 print（3,4）这样的表达式的意义在 Lua 中早有定义）。因此改变区域设置后的系统，可能既无法识别使用逗号的数值，也无法理解使用点号的数值：

```
-- 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'
```

上例中，我们的期望是使用逗号表示小数点，然而 Lua 却将 print(3,4)理解成执行两个数字的打印；而原来的小数打印 print(3.4)也无法正确执行。

第 23 章 调试库

调试库（Debug 库）并没有提供任何可用的 Lua 调试器，它给你提供了编写 Lua 调试器所需的一切基础函数。出于性能方面的考虑，这些函数的正式接口是由 C API 实现的，调试库提供了在 Lua 代码中直接使用这些功能的便利。调试库中的所有函数都在 debug 表中被声明。

与其他 Lua 库不同，你应该尽可能少地使用调试库。首先，调试库中的一些函数性能并不高，其次，它打破了语言本身的某些原则，比如，你不能函数外部访问函数内部的局部变量。通常，你可能并不想在产品的最终版中使用它，或者你可能向删除该库：

```
debug = nil
```

调试库由两类函数组成：自检函数（Introspective Function）和钩子函数（Hook）。自检函数可以让我们从多方面监视运行中的程序，比如程序栈中的当前函数、当前指令代码、局部变量名及其值。钩子函数允许你跟踪程序的执行情况。

栈层次（Stack Level）是调试库的一个重要概念。栈层次是一个用以标识当前活动函数的数值，即，这是一个被调用但还未返回的函数。例如，用于调用调试库的函数其栈层次为 1，调用该函数的函数其栈层次为 2，以此类推。

23.1 自检函数

调试库中最主要的自检函数是 debug.getinfo，它的第一个参数是一个函数或栈层次，假设存在一个一个名为 foo 的函数，并调用 debug.getinfo(foo)，你将得到一个含有该函数信息的表，这个表可能含有下列域：

source	函数的定义位置。如果函数在字符串内被定义（通过 loadstring 函数），source 就是该字符串，如果函数在文件中被定义，source 就是带“@”前缀的文件名。
short_src	source 的简短版本（60 个字符以内），对错误信息很有用。
linedefined	source 中函数被定义处的行号。
what	函数类型。如果 foo 是普通的 Lua 函数，结果为“Lua”；如果是 C 函数，结果为“C”；如果是 Lua 的主代码段，结果为“main”。
name	函数的名称。
namewhat	name 域的含义。可能的取值为：“global”、“local”、“method”、“field”，或者空字符串。空字符串意味着 Lua 无法找到这个函数名。
nups	函数中的 Upvalues 的个数。
func	函数本身。稍后介绍。

如果 foo 是一个 C 函数，Lua 无法给出所有信息，对于 C 函数来说，只有 what、name、namewhat 这几个域可用。

以数值 n 作为参数调用 debug.getinfo(n)时，该函数返回在栈层次 n 处的函数的信息。例如，如果 n 的取值为 1，你将得到调用该函数（getinfo 函数）的函数的信息；如果 n 的取值为 0，你将得到 getinfo 这个 C 函数本身的信息。如果 n 的取值大于栈中活动函数的总个数，debug.getinfo 返回空值。当使用数值 n 调用 debug.getinfo 查询某活动函数的信息时，返回表中有一个额外的域 currentline，即在该函数当前所在的行号，另外，func 即指定栈层次中的函数。

域 name 可能会造成混淆。由于 Lua 中的函数是第一类值，所以一个函数可能是匿名的或拥有多个名字。此时，Lua 将尝试查找一个含有该函数的全局变量，或者检查调用该函数的代码，看该函数是如

何被调用的。第二种情况只有使用数值参数调用 getinfo 时才会发生，这时，我们将得到具体的调用信息。

函数 getinfo 的效率并不高，Lua 保存调试信息的第一原则是不妨碍程序的执行，其次才考虑效率因素。为了获得更好的效率，函数 getinfo 提供了第二个可选的参数，该参数决定哪些调试信息将被返回，指定了该参数之后，getinfo 函数将不会浪费时间去收集那些用户并不关心的信息。这个参数以字符串形式给出，其中的每个字母代表一类信息：

字母选项	返回的域
'n'	name、namewhat
'f'	返 func
'S'	source、short_src、what、linedefined
'l'	currentline
'u'	nup

下面的函数介绍了 debug.getinfo 的使用方法，该函数打印了活动栈的基本跟踪信息：

```
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
```

改进该函数从 getinfo 函数获得更多的数据并不困难。实际上，调试库提供了一个上述函数的改进版，debug.traceback。与上述函数不同的是，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

你可以调用调试库的 getlocal 函数来访问当前正运行的函数中的局部变量。函数 getlocal 有两个参数：待查询函数的栈层次和变量的索引。该函数有两个返回值：变量名以及变量的当前值。如果指定的变量索引大于活动变量的个数，getlocal 将返回空值，如果指定的栈层次无效，则函数会抛出错误（可以调用 debug.getinfo 函数来检查栈层次的有效性）。

Lua 以出现的次序对函数中的局部变量进行计数，只有在当前函数作用域内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（函数 foo 的第一个参数），2 是 b，3 是 x，4 是另一个 a。函数 getlocal 被调用时，c 已经超出了作用域，而变量 name 和 value 都还未进入作用域（因为局部变量仅在初始化后才可见）。

你可以使用 debug.setlocal 函数修改局部变量的值，该函数的第一第二参数与 getlocal 函数一样，分别是栈层次和变量索引，第三个参数是变量的新值，函数 setlocal 返回被赋值的变量名或空值（如果该变量并不在作用域内）。

23.1.2 访问Upvalue

调试库的 getupvalue 函数允许我们访问 Lua 函数的 Upvalue。和局部变量不同的是，即使函数不处于运行状态也仍然具有 Upvalue（毕竟，这就是闭包的意义所在）。因此，函数 getupvalue 的第一个参数不是栈层次而是一个函数（精确地说，是一个闭包），第二个参数是 Upvalue 的索引。Lua 按照 Upvalue 在函数中被引用的顺序进行编号，因为一个函数不可能有两个同名的 Upvalue，所以这个顺序并不重要。

可以使用函数 debug.setupvalue 函数修改 Upvalue 的值，正如你预计的，该函数有三个参数：闭包、Upvalue 索引和 Upvalue 的新值。与 setlocal 函数类似，这个函数返回 Upvalue 的名字或空值（如果 Upvalue 并不在作用域内）。

下面的代码演示了如何在给定变量名的情况下获得该变量的值：

```
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，我们尝试从局部变量中查找指定的变量：如果存在多个同名的变量，则必须返回索引值最大的那一个，因此，我们必须执行所有可能的循环（而不是在找到第一个满足条件的变量时，使用 break 跳出循环）。如果在局部变量中找不到指定的变量，将继续尝试 Upvalue 类型的变量：首先，我们调用 debug.getinfo(2).func 取得调用上述函数的函数，然后遍历该函数中的 Upvalue。最后如果没有找到一个具有指定变量名的 Upvalue，则返回一个拥有指定变量名的全局变量（当然，如果不存在该全局变量，则其值为空值）。注意调用 debug.getlocal 和 debug.getinfo 函数时候使用了 2 作为参数，它表示调用上述函数的函数。

23.2 钩子程序（Hook）

调试库的钩子机制允许我们注册一个函数，在运行程序中发生某一事件时，该函数将被调用。能够触发钩子程序的事件有四类：当 Lua 调用一个函数的时候将发生 Call 事件；当函数返回的时候将发生 Return 事件；当 Lua 执行某一行代码的时候将发生 Line 事件；当执行完指定数目的指令之后将发生 Count 事件。Lua 使用单个参数调用钩子程序，该参数是一个用于描述事件的字符串：“call”、“return”、“line”以及“count”。此外，对于 Line 事件，还需传递第二个参数：新行号。我们总是可以使用函数 debug.getinfo 在钩子程序内部获取更多的信息。

要注册一个钩子程序，我们将传递两个或三个参数给 debug.sethook 函数：第一个参数是钩子函数，第二个参数是一个用于描述被监视事件类型的字符串；可选的第三个参数是一个用于描述 Count 事件的

频率的数值。为了监视 Call、Return 和 Line 事件，可以将它们的首字母（“c”、“r”或“l”）嵌入第二个字符串参数内。要关闭钩子程序，只需调用无参的 sethook 函数即可。

下面是一个简单的例子，该代码将安装一个最基本的跟踪程序，该跟踪程序将打印被执行的每一行的行号：

```
debug.sethook(print, "l")
```

上述代码只是简单地将 print 函数作为钩子函数，并指示 Lua 仅在 Line 事件发生时调用它。下面是一个稍复杂的跟踪器，它将使用 getinfo 函数在跟踪信息中添加当前文件名的信息：

```
function trace(event, line)
    local s = debug.getinfo(2).short_src
    print(s .. ":" .. line)
end

debug.sethook(trace, "l")
```

23.3 程序分析（Profile）

尽管有着“调试库”的称呼，调试库在执行任务方面绝不比程序调试逊色。一个常见的任务就是程序分析。对于实时分析而言，最好借助于 C 程序来完成：因为在 Lua 中调用钩子程序的代价太高，且容易影响分析结果的准确性。不过，对于计数类的分析而言，Lua 代码还是能够胜任的。在这一节中，我们将设计一个简单的分析程序，它可以列出每个函数在程序运行过程中被调用的次数。

分析程序中的主要数据结构是两个 Lua 表：一个表用于关联函数及其调用次数，另一个表用于关联函数及其函数名。这两个表的索引就是函数本身。

```
local Counters = {}
local Names = {}
```

虽然我们可以在分析完成之后取得函数名，但是请注意，在函数处于活动状态的情况下取得函数名可以提高程序的效率，因为此时 Lua 可以检视调用该函数的代码并取得该函数的名字。

下面我们将定义钩子函数，它的任务是获取正在执行的函数并将其计数器加 1，同时取得该函数的名字：

```
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
```

下一步就是在程序中使用上述钩子函数。我们假设程序的主代码段位于一个文件内，且用户将向分析程序传递该文件的文件名作为参数：

```
prompt> lua profiler main-prog
```

按照上述前提，我们将取得保存在 arg[1]中的文件名，打开钩子程序，并运行该文件：

```
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
```

最后一个步骤是显示程序运行的结果，下面的函数将为函数产生一个名字。由于 Lua 中的函数名具有很大的可变动性，所以我们给每一个函数加上它的行号信息，例如，file:line。如果一个函数是匿名的，那么我们就使用它的位置信息来表示函数名，如果一个函数是 C 函数，那么我们仅使用其函数名（它没有位置信息）。

```
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的函数名和它的计数器：

```
for func, count in pairs(Counters) do
    print(getname(func), count)
end
```

如果我们将上述分析程序应用到 10.2 章节中马尔科夫链的例子上，我们将得到如下结果：

```
[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
```

从上述结果可以看出，位于第四行的匿名函数（在 allwords 内定义的迭代函数）被调用了 884723 次，而函数 write（即 io.write）被调用了 10000 次。

你可以对这个分析程序进行一些改进，比如，对输出结果进行排序、以更好的形式打印函数名、改进输出格式等。不过，这个基本的分析程序已经很管用了，同时还可以作为编写更高级工具的基础。

第四篇 C-API

第 24 章 C-API概要

Lua 是一种嵌入式的语言，这意味着 Lua 不是一个独立的软件包，而是一个可以被连接到其他应用程序并为其提供 Lua 功能的库。

你可能会疑惑：如果 Lua 不是独立的程序，那为什么到目前为止我们都是独立地运行 Lua 程序呢？这个问题的答案在于 Lua 解释器（可执行的 Lua）。Lua 解释器是一个很小的应用程序（不超过 500 行代码），它是一个借助于 Lua 标准库来实现的独立解释器。解释器处理程序与用户之间的交互：从用户那里获取文件或字符串，然后将其传给 Lua 标准库，Lua 标准库负责所有的工作（比如，运行最终的代码）。

能够作为程序库来扩展其他应用程序的功能，也就是 Lua 被称为扩展性的语言的原因所在。同时，一个使用 Lua 的应用程序能够在 Lua 环境中注册新的函数，这些函数可以由 C 语言或其他语言实现，并能够增加一些不容易由 Lua 实现的功能，这使 Lua 具有可扩展性，从而被称为可扩展的语言。

上述两种观点（Lua 是扩展性的语言和 Lua 是可扩展的语言）对应于 C 语言和 Lua 之间的两种交互方式。第一种，C 语言具有控制权而 Lua 为其提供库，这种方式下的 C 代码通常被称为应用代码。第二种，Lua 具有控制权而 C 为其提供库，这种方式下的 C 代码通常被称为库代码。在这两种方式下，作为应用代码和库代码的 C 语言都使用了相同的 API 与 Lua 通信，也就是所谓的 C-API。

C-API 是一个允许 C 代码与 Lua 进行交互的函数集。它有以下几类函数组成：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 函数以便 Lua 代码调用的函数等。在本书中，函数这个术语实际上指函数或宏，API 提供的部分功能是以宏的方式实现的。

C-API 遵循 C 语言的语法形式，这与 Lua 相当不同。使用 C 进行程序设计的时候，我们必须注意类型检查（还有类型错误）、错误处理、内存分配错误等诸多问题。API 中的大部分函数都不检查其参数的正确性，因此你必须在调用函数前确保其参数是有效的。如果传递了错误的参数，你将有可能得到类似“segmentation fault”这样的错误，而不会得到更明确的错误信息。另外，API 将重点放在了灵活性和简洁性上，有时候甚至在易用性上作出了妥协。一个普通的任务可能需要涉及多个 API 调用，这可能相当令人乏味，但它在细节上为你提供了全面的控制能力，比如，错误处理、缓冲大小的设置等。

正如本章的标题所示，这一章的目标是对 C 代码调用 Lua 时涉及的内容做一个概要的介绍。如果在某些细节方面有所不解请先不要着急，因为稍后还会对细节部分进行详细的介绍。不过，请记住你可以在 Lua 参考手册中找到有关具体函数的更详细的介绍。另外，你还可以在 Lua 发行版中找到有关 API 应用方面的例子，Lua 解释器（lua.c）提供了应用代码的例子，而标准库（lmathlib.c、lstrlib.c 等）提供了库代码的例子。

从现在开始，你戴上了 C 程序员的帽子，当下文中提到“你”或“你们”的时候，其意思就是指你将使用 C 语言进行程序设计。

C 语言与 Lua 进行通信所依赖的重要组成部分就是一个虚拟的栈。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。这个虚拟的栈弥合了存在于 C 和 Lua 之间的两个不协调之处：第一，Lua 会自动进行垃圾收集，而 C 要求显式地释放内存空间；第二，Lua 中的动态类型和 C 中的静态类型不一致。我们将在 24.2 章节详细地介绍与栈相关的内容。

24.1 第一个示例程序

我们将通过一个简单的应用程序开始介绍 Lua 的 C-API：一个独立的 Lua 解释器。我们将编写一个

简单的 Lua 解释器，代码如下：

注意，下列代码只适用于 Lua 5.0，如要运行于 Lua 5.1，则必须要将前 5 个名为 luaopen_*的调用替换成 luaL_openlibs(L)。

```
#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 状态的函数（如 lua_open）、调用 Lua 函数的函数（如 lua_pcall）、读写 Lua 状态中的全局变量的函数、注册可以被 Lua 代码调用的新函数的函数等。所有在 lua.h 中被定义的函数都有一个 lua_前缀。

头文件 lauxlib.h 定义了由辅助库（auxlib）提供的函数，同样，所有在其中定义的函数等都以 luaL_作为前缀（如 luaL_loadbuffer）。辅助库利用 lua.h 中提供的基础函数推出了更高层次上的抽象；所有 Lua 标准库都使用了辅助库。基础 API 致力于简洁精练，而辅助库则致力于实现针对一般任务的实用性，当然，就你的应用程序的需要来创建其他的抽象也是非常容易的。需要注意的是，辅助库没有存取 Lua 内部结构的权限，它的工作都是通过调用最基本 API 来完成的。

Lua 库没有定义任何全局变量，它将其状态信息都保存在动态结构 lua_State 中，而指向这个结构的指针总是被作为参数传递给所有的 Lua 函数。这样的实现方式使得 Lua 能够被多个调用者读取，而且也为多线程的应用作好了准备。

函数 lua_open 用于创建一个 Lua 环境或状态，当一个新的状态被 lua_open 创建时，该状态并不包含任何预定义的函数，甚至是 print 函数。为了保持 Lua 的精简，所有的标准库都是以单独的包提供的，所以如果没有需要也不会强求程序员使用它们。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 表，并将输入输出函数（如 io.read、io.write 等）注册到该表中。

在创建一个 Lua 状态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，该 C 程序首先调用 luaL_loadbuffer 编译这些 Lua 代码，如果没有错误，该函数返回零并将编译好的代码段压栈（我们将在下一节中讨论“魔法”栈）。之后，在 C 程序中调用 lua_pcall，该函数将代码段出栈并在保护模式下运行它。和 luaL_laodbuffer 函数一样，函数 lua_pcall 在没有错误的情况下返回零，在出错的情况下，这两个函数都将错误消息压栈，之后便可以用 lua_tostring 来获取该错误信息，在将其输出之后，我们可以使用 lua_pop 函数将其从栈中删除。

注意，在出错的情况下，这个程序只是简单地在标准错误流中输出错误信息。在 C 语言中，实际的错误处理是非常复杂的，如何处理这些错误则依赖于应用程序本身的特点。Lua 核心绝不会在输出流上写入任何数据，而是通过返回错误代码和错误信息来通知错误的发生，每一个应用程序都可以用最适合它自己的方式来处理这些错误。为了简化讨论，我们将考虑一个简单的错误处理程序，就像下面代码一样，它只是输出一条错误信息、关闭 Lua 状态并退出应用程序：

```
#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);
}
```

稍候我们再详细地讨论如何在应用程序中处理错误。

为了让 Lua 可以在 C 或 C++任何一个环境中编译，lua.h 并不包含下列经常出现在其他 C 库中的调整代码：

```
#ifdef __cplusplus
extern "C" {
#endif
    ...
#ifdef __cplusplus
}
#endif
```

因此，如果你以 C 方式编译得到 Lua 库，但需要用在 C++中，那么你需要像下面这样来包含 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到你的 C++程序中。

24.2 虚拟栈

在 Lua 和 C 之间交换数据时将面临两个问题：动态类型与静态类型系统的不匹配，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（由于元表的存在，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，不管怎样，任何操作表的函数（比如，settable）必定只有一个固定的类型，因此我们将需要几十个不同的函数来完成这一个的操作（三个参数类型的每一种组合都需要一个函数）。

我们可以在 C 中声明某种联合（Union）类型来解决这个问题，假设这种类型为 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用于操作表的函数：

```
void lua_settable(lua_Value a, lua_Value k, lua_Value v);
```

这种解决方案有两个缺点。第一，要将如此复杂的类型映射到其他语言很困难，Lua 不仅被设计成易于与 C/C++交互，与 Java、Fortran 之类语言之间的交互也必须一样容易。第二，Lua 负责垃圾回收，如果我们将一个 Lua 值保存在一个 C 变量中，由于 Lua 无法了解这种用法，它可能会错误地视这个值为垃圾而对其进行回收。

因此，在 API 中没有定义任何类似 lua_Value 的类型。取而代之的方案是，利用一个抽象的栈在 Lua 与 C 之间交换数据。栈中的每一条记录都可以保存任何 Lua 值，无论何时想要从 Lua 请求一个值（如一个全局变量的值），你只需要调用 Lua，Lua 将这个被请求的值压栈。无论何时想要传递一个值给 Lua，首先将这个值压栈，然后调用 Lua 将这个值出栈。尽管我们仍然需要不同的函数将每种 C 类型的值压栈以及将其从栈上取出（译注：是取出而不是弹出），但是我们避免了不同类型间过多的组合。另外，因为栈是由 Lua 管理的，因此垃圾回收器知道哪个值正在被 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（被编译的代码段，或者一条错误信息）留在了栈上，而 lua_pcall 从栈上获取要被调用的函数并把任何可能的错误信息压栈。

Lua 以一个严格的后进先出（LIFO，Last In First Out）规则来操作栈，也就是说，始终存取栈顶。当你调用 Lua 时，它只会改变栈顶部分。但是你的 C 代码却有更多的自由，更明确地讲，你可以查询栈上的任何元素，甚至在任何一个位置插入和删除元素。

24.2.1 元素压栈

对于每一种可以用 C 类型描述的 Lua 类型，API 中都有一个压栈函数与之对应：函数 lua_pushnil 可以将空值压栈；函数 lua_pushnumber 可以将数值型的值压栈，该类型可以用 C 语言中的双精度类型 double 表示；函数 lua_pushboolean 可以将布尔型的值压栈，该类型可以用 C 语言中的整数类型 int 来表示；函数 lua_pushlstring 可以将含有任意字符的字符串压栈，该类型可以用 C 语言中的字符指针 char *来表示；而函数 lua_pushstring 可以将 C 风格（以“\0”结束）的字符串压栈：

```
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);
```

同样也有可将 C 函数和 Userdata 值压栈的函数，稍后将会讨论到它们。

Lua 中的字符串不是以“\0”为结束符的，它的长度需要明确指定，因此可以包含任意的二进制数据，将字符串压栈的正式函数是 lua_pushlstring，它要求一个明确的长度作为参数。对于以“\0”结束

的字符串，你可以用 lua_pushstring，它用 C 函数 strlen 来计算字符串的长度。Lua 不会保存一个指向外部字符串或任何其他对象的指针，除了 C 函数——它总是静态指针。对于需要保持的字符串，Lua 要么做一份内部的拷贝，要么重用已经存在的字符串。因此，一旦这些函数返回之后，你可以自由地或是释放你的缓冲区。

一旦你需要将元素压栈，你就有责任确保在栈上有足够的空间来容纳该元素，记住，你现在是 C 程序员，Lua 不会宠着你。当 Lua 被启动以及 Lua 调用 C 的时候，栈上至少有 20 个空闲的位置（lua.h 中的 LUA_MINSTACK 宏定义了这个常量），这对多数常规用法是足够的，所以通常我们不必去考虑它。但是，有些任务可能需要更多的栈空间（如调用一个不定参数数目的函数），在这种情况下，你可能要调用下面这个函数：

```
int lua_checkstack(lua_State * L, int sz);
```

该函数检测栈上是否有足够的空间（稍后会有关于它的更多信息）。

24.2.2 元素查询

API 使用索引来访问栈中的元素。栈中第一个元素（也就是第一个被压栈的元素）的索引为 1，下一个元素的索引为 2，以此类推。我们也可以使用栈顶作为索引元素的参照，那就是使用负索引，此时，索引-1 指向栈顶元素（也就是最后压栈的函数），索引-2 指向前一个元素，以此类推。例如，如果调用 lua_tostring(L, -1)，将从栈顶得到字符串类型的值。我们下面将看到，在某些场合下使用正索引访问栈比较方便，而在另外一些情况下，使用负索引则更为方便。

API 提供了一套名为 lua_is*的函数来检查元素是否属于指定的类型，“*”代表任何 Lua 类型，因此包含了 lua_isnumber、lua_isstring、lua_istable 等函数。所有这些函数都有相同的原型：

```
int lua_is...(lua_State * L, int index);
```

lua_isnumber 和 lua_isstring 函数不检查该元素的值是否是指定的类型，而是判断能够将其被转换成指定的类型。例如，任何数值类型都满足 lua_isstring 函数。

还有一个 lua_type 函数，该函数返回栈中元素的类型（lua_is*中的某些函数实际上是使用这个函数定义的宏）。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 switch 语句中。当我们需要真正地检查字符串和数值类型时它将是非常有用的。

为了从栈中获得值，API 定义了一系列名为 lua_to*的函数：

```
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);
```

即使指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，此时，lua_toboolean、lua_tonumber 和 lua_strlen 返回 0，其他函数返回 NULL。由于 ANSI C 没有提供可以用来判断错误的数值，所以返回的 0 是没有什么用处的。对于这些函数的其他功用而言，我们一般并不真正需要对应的 lua_is*函数：我们只需要调用 lua_is*，并测试返回结果是否为 NULL 即可。

lua_tostring 函数返回一个指向该字符串内部拷贝的指针，但你不能修改该字符串，因为 const 限定词规定了该字符串是无法被修改的常量。只要这个指针对应的值还在栈内，Lua 就会保证这个指针总是有效的。当一个 C 函数返回后，Lua 会清理该函数的栈，所以这里有一个原则：永远不要将指向 Lua 字符串的指针保存到访问它们的函数的外部。

由 lua_string 返回的字符串的结尾总会有一个字符串结束标志“\0”，但是字符串中间也有可能包

含该字符，此时可以使用 lua_strlen 返回该字符串的实际长度。具体而言，假定栈顶的值是一个字符串，下面的断言总是有效的：

```
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 其他栈操作

除了上面所提及的用于支持 C 与虚拟栈交换值的函数外，API 也提供了下列函数来完成常规的栈维护工作：

```
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 返回栈中的元素个数，元素个数也即栈顶元素的索引。注意一个负数索引-X 对应的正数索引值为 lua_gettop - X + 1。

函数 lua_settop 用于设置栈顶索引值，即栈中元素的个数，如果原来的栈顶高于新的栈顶，多出来的元素将被丢弃，如果新的栈顶高于原来的栈顶，为了设置指定的栈大小，该函数将相应个数的空值压栈。特别需要注意的是，调用 lua_settop(L, 0)将清空整个栈。你也可以用负数索引作为调用 lua_settop 的参数，来设置栈顶的索引值。在该函数的基础上，API 提供了下面这个宏，它从将栈中的 n 个元素出栈：

```
#define lua_pop(L, n)  lua_settop(L, -(n)-1)
```

函数 lua_pushvalue 将指定索引对应的元素的副本压入栈顶；函数 lua_remove 移除指定索引对应的元素，并将位于该元素之上的所有元素下移，来填补该元素的空白；函数 lua_insert 将栈顶元素移动到指定索引的位置，并将位于该索引位置之上的所有元素上移；函数 lua_replace 将栈顶元素出栈并将指定索引处的元素设成其值，该函数无须移动任何元素。注意下列操作对栈没有任何影响：

```
lua_settop(L, -1);       /* set top to its current value */
lua_insert(L, -1);       /* move top element to the top */
```

为了演示这些函数的用法，下面是一个有用的辅助函数，它能够倾倒（Dump）整个栈的内容：

```
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整个栈，参照每个元素的类型打印其值。它用引号输出字符串；以%g格式输出数值；对于其他值（Lua 表、函数等），它仅输出其所属的类型（函数 lua_typename 可以把类型代码转换成类型名）。

下面的程序利用上述 stackDump 函数进一步说明虚拟栈的操作：

```
#include <stdio.h>
#include <lua.h>

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 */
}

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中的错误处理

不像 C++或 Java，C 语言没有提供任何异常处理机制。为了改善这种处境，Lua 利用 C 的 setjmp 函数构造了一个类似异常处理的机制（如果用 C++来编译 Lua，那么你可以修改代码来使用真正的异常处理机制）。

Lua 中所有的结构都是动态的：它们按需增长，必要时又可以缩减。这意味着内存分配失败的可能性在 Lua 中是普遍的，几乎任何操作都会引发这种意外。Lua 的 API 用异常来通知这些错误的发生，而不是为每一步操作产生错误码，也就是说，几乎所有的 API 函数都会抛出错误（也就是调用 longjmp 函数）来代替返回。

当我们编写库代码（也就是能够被 Lua 调用的 C 函数）时，使用长跳转（Long Jump）几乎和真正的异常处理一样的方便，因为 Lua 能够捕获任何偶发的错误。但是，当我们编写应用代码时（也就是能够调用 Lua 的 C 代码），我们必须自行提供一种方法来捕获这些错误。

24.3.1 应用代码中的错误处理

通常你的应用代码运行在非保护模式下。由于应用代码不是由 Lua 调用的，因此 Lua 无法根据正确的上下文情况来捕捉错误（也就是说，Lua 无法调用 setjmp 函数进行跳转）。在此环境下，当 Lua 遇到类似“内存不足”的错误，能够由 Lua 处理的确实不多，Lua 将调用一个 panic 函数，当该函数返回时退出应用程序（你可以使用 lua_atpanic 函数设置你自己的 panic 函数）。

并不不是所有的 API 函数都会抛出异常，函数 lua_open、lua_close、lua_pcall 和 lua_load 都不会

抛出异常，另外，大多数其他函数都只能在内存分配失败时抛出异常：比如，如果函数 luaL_loadfile 没有足够内存来容纳指定的文件，该函数调用将以失败告终。有些程序在遇到内存不足时，可能会忽略该异常，对这些程序而言，如果没有足够的内存让 Lua 继续运行，使用 panic 是完全可以胜任的。

如果不希望应用程序在内存分配失败的情况下退出，那么你必须在保护模式下运行 Lua 代码。通常你可以调用 lua_pcall 来运行 Lua 代码，这时候，该 Lua 代码就运行于保护模式下，此时，即使内存分配失败，lua_pcall 也返回一个错误代码，以保持 Lua 解释器的一致性。如果想要保护所有与 Lua 交互的 C 代码，那么你可以使用 lua_cpcall 函数（关于这个函数更具体的说明，请看 Lua 参考手册，在 lua.c 文件中有关于该函数应用的例子）。

24.3.2 库代码中的错误处理

Lua 是一种安全的语言，也就是说，不管是什么样的代码，也不管代码导致何种错误，你都可以根据 Lua 本身的规定来预测程序的行为模式。同时，错误也被发现并根据 Lua 的规定被解释。你可以与 C 语言对比一下，C 语言中很多错误代码的行为模式只能依据底层的硬件进行解释，同时错误发生的地点也只能由指令计数器指出。

在 Lua 中添加一个新的 C 函数时，原有的安全性可能会被打破。比如，一个能够在任意内存地址存放任意字节的 poke 函数，就极有可能导致各种各样的内存错误。你应该把重心放在确保新加函数的安全性上，并切实处理好潜在的错误。

正如我们前面所讨论的，每一个 C 程序都有它自己处理错误的方式，当你为 Lua 编写库函数的时候，有一个处理错误的标准方法可供参考：不管何时，当 C 函数发现错误时，只需要简单地调用 lua_error 函数（或 luaL_error 函数，该函数能够格式化错误信息并调用 lua_error 函数）。lua_error 函数会清理 Lua 中所有无用的数据，然后附带错误信息，并跳转到导致异常的 lua_pcall 函数。

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，我们将通过一个简单的例子来说明如何借助 Lua 来配置应用程序，之后还会逐步改进这个例子来完成更为复杂的任务。

首先，让我们想象一下简单的配置方案：你的 C 程序（让我们称呼其 pp）有一个窗口界面并允许用户自定义窗口的初始大小。显然，就如此简单的任务而言，有多种解决方案都比使用 Lua 更简单，比如，使用环境变量或读取含有“名字-值”对的文件。但即便是采用一个简单的文本文件，你仍然需要解析该文件的内容。所以，最后决定采用 Lua 配置文件（也就是一个包含 Lua 程序的文本文件），该文件可能只是简单地包含如下信息：

```
-- configuration file for program 'pp'
-- define window size
width = 200
height = 300
```

现在，你必须调用 Lua API 去解析这个文件，并取得 width 和 height 这两个全局变量的值。下面的函数将完成这一工作：

```
#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，该函数打开 Lua 包并载入标准函数库（虽然这是可选的，但通常包含这些库是一个比较好的做法）。然后使用 luaL_loadfile 函数加载 filename 指定的文件中的代码段，同时调用 lua_pcall 函数运行该代码段，如

果这两个函数发生错误（例如，配置文件中含有语法错误），它们将返回非零的错误代码并将相应的错误信息压栈。与往常一样，程序使用-1 作为索引参数调用 lua_tostring 函数来取得位于栈顶的错误信息（上述代码中的 error 函数已经在 24.1 章节中定义过）。

执行完上述代码段之后，程序将要取得全局变量的值，为此，程序两次调用 lua_getglobal 函数，除了必须的 lua_State 类型的参数之外，另一个参数即全局变量的名称。每次调用都将相应的全局变量值压入栈顶，所以变量 width 的索引值为-2，而变量 height 的索引值为-1，即位于栈顶（因为初始的栈为空，因此你可以参照栈底进行索引，1 表示第一个元素，2 表示第二个元素。不过，如果参照栈顶进行索引，则不管初始的栈是否为空，你的代码都能正确地运行）。接着，程序调用 lua_isnumber 函数判断每个变量的值是否为数值类型，后又调用 lua_tonumber 函数将变量值转换成 double 类型，而 C 程序得到该值后又强制转换为 int 类型。最后，程序关闭 Lua 状态并返回。

Lua 是否值得一用？正如我在前面所说，在这个简单的例子中，使用一个仅包含两个数值的文件将比使用 Lua 更为容易。尽管如此，使用 Lua 将带来了一系列的优势。首先，Lua 将为你处理所有语法问题（包括程序错误）；配置文件甚至可以包含注释信息！其次，用户可以利用 Lua 作出更为复杂的配置。例如，脚本可以向用户提示某些信息，或取得环境变量的值：

```
-- configuration file for program 'pp'
if getenv("DISPLAY") == ":0.0" then
    width = 300; height = 300
else
    width = 200; height = 200
end
```

即使是如此简单的配置方案，也很难预料用户的意图，但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功能，这将令你的程序更具灵活性。

25.1 Lua表的操作

现在，我们将使用 Lua 来配置程序，除了窗口的尺寸，还需要配置窗口的背景颜色。我们假定最终的颜规格含有三个数值，每个数值代表 RGB 颜色的一个分量。通常，在 C 语言中，这些数值为[0, 255]范围内的整数，由于 Lua 中的所有数值都是实数，因此我们可以使用更自然的范围[0, 1]来表示。

一个初步方案是，每一个 RGB 颜色的分量使用一个全局变量来表示：

```
-- configuration file for program 'pp'
width = 200
height = 300
background_red = 0.30
background_green = 0.10
background_blue = 0
```

这个方案有两大缺点：第一，太过冗余（为了表示窗口的背景色，窗口的前景色，菜单的背景色等，一个实际的应用程序可能需要众多不同的颜色）；第二，无法预定义颜色的共同部分，比如，事先定义了 WHITE 颜色，那么用户只需要简单地指定 background = WHITE 来表示背景色为白色。为了避免这些缺点，我们将使用 Lua 表来表示颜色：

```
background = {r = 0.30, g = 0.10, b = 0}
```

采用 Lua 表的方式使教本更具有结构性。现在用户和应用程序都很容易进行颜色的预定义，之后便能够在配置文件中使用预定的颜色：

```
BLUE = {r = 0, g = 0, b = 1}
...
background = BLUE
```

接着便可以在 C 程序中获取这些颜色值：

```
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");
```

与往常一样，我们首先获取全局变量 backgroud 的值，并确保它的类型是 Lua 表。然后，我们使用 getfield 函数获取颜色分量，但该函数不是 API 的一部分，因此我们需要自行定义：

```
#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;
}
```

这里我们再次遇到多种类型组合的问题：鉴于 key 的类型、value 的类型以及错误处理方式的不同，可能需要不同版本的 getfield 函数。不过，Lua API 只提供了一个 lua_gettable 函数，该函数接受 Lua 表在栈中的位置作为参数，将位于栈顶的 key 出栈，并将与 key 对应的 value 压栈。我们自定义的 getfield 函数假定 Lua 表位于栈顶，因此，在 lua_pushstring 函数将 key 压栈之后，Lua 表的索引变成了-2。在返回以前，getfield 函数将 value 值出栈，将栈恢复成调用 getfield 函数前的状态。

我们将对上述例子稍作延伸：为用户加入颜色名。用户仍然可以使用颜色表，但是也可以继续使用预定义的常用颜色。为了实现这个特性，在 C 代码中需要建立一个颜色表：

```
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 */
```

```
};
```

这个实现方案将使用颜色名称创建 Lua 全局变量，并使用上述颜色表中的颜色对其进行初始化。程序的运行效果就如同用户在教本中使用了下面的代码一样：

```
WHITE = {r = 1, g = 1, b = 1}
RED  = {r = 1, g = 0, b = 0}
...
```

上述方案（在 C 代码中定义颜色）与用户自定义颜色的方案的不同之处在于：应用程序在教本运行之前就已经将颜色定义好了。

为了设置 Lua 颜色表的域（颜色分量），我们将定义一个辅助函数 setfield，该函数将索引及其对应的值压栈后，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 API 函数一样，函数 lua_settable 支持不同的数据类型，它从栈中获取所有的操作数。lua_settable 以 Lua 表在栈中的索引作为参数，并将栈中 Lua 表的域和该域对应的值出栈。函数 setfield 假定在调用之前 Lua 表位于栈顶（其索引为-1），将 Lua 表的域和该域对应的值入栈之后，Lua 表的在栈中的索引将变成-3。

函数 setcolor 用于定义一个颜色，首先它必须创建一个 Lua 表，设置相关的域，最后将这个 Lua 表赋给对应的全局变量：

```
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 */
}
```

在上述例子中，函数 lua_newtable 创建一个空的 Lua 表并将其入栈，函数 setfield 用于设置该 Lua 表的域，最后函数 lua_setglobal 将该 Lua 表出栈并将其赋给一个全局变量。

有了前面这些函数，下面的循环语句可以在程序的全局变量中注册所有的颜色：

```
int i = 0;
while(colortable[i].name != NULL)
    setcolor(&colortable[i++]);
```

注意，应用程序必须在运行用户脚本之前，执行这个循环。

上述方案（用名称指定颜色值）有另外一个可选的方法。用一个字符串来表示颜色名，而不是像上述例子那样使用全局变量表示，编写配置文件时可以指定 background = "BLUE"。因此，background 既可以是 Lua 表也可以是字符串。有了这种实现方案，应用程序在运行用户脚本之前就不需要再注册颜色表了。不过，却因此需要一些额外的步骤来获取颜色。当程序得到变量 background 的值之后，必须判断这个值的类型，是 Lua 表还是字符串：

```
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'");
```

哪个方案最好呢？在 C 程序中，使用字符串来提供选项不是一个好的做法，因为编译器不会对字符串进行拼写检查，但在 Lua 中，全局变量不需要声明，因此当用户错误地拼写了颜色名时，Lua 不会给出任何错误信息。如果用户将“WHITE”误写成“WITE”，background 变量将为 nil（也就是 WITE 的值，该变量并没有初始化过），然后应用程序会认为 background 的值为 nil，而没有其他关于这个错误的信息可以获得。另一方面，background 的值可能是错误拼写的字符串，因此，在发生错误的时候，应用程序可以将此信息加到错误信息中去。应用程序可以忽略大小写来比较字符串，因此，用户可以使用“white”、“WHITE”，甚至“White”。另外，如果用户脚本很小并且颜色数量众多，注册了百上千个颜色（需要创建成百上千个 Lua 表和全局变量）而最终用户只用到了其中的几个，这会让人觉得很怪异。但在使用字符串的时候，你将能够避免这种局面。

25.2 调用Lua函数

使用 Lua 的一个好处是可以在配置文件中定义函数，而该函数最终可以被应用程序调用。比如，编写一个应用程序来绘制函数的图像，而该绘图函数由 Lua 配置文件来定义。

使用 API 调用 Lua 函数的方法很简单：首先，将被调用函数入栈，其次，依次将所需参数入栈，然后，使用 lua_pcall 来调用该函数，最后，从栈中弹出函数执行的结果（函数的返回值）。

在下面的例子中，我们假定 Lua 配置文件定义了下面的函数：

```
function f(x, y)
    return (x ^ 2 * math.sin(y)) / (1 - x)
end
```

对于给定 x、y，我们想在 C 代码计算 z = f(x, y)的值。假如你已经打开了 Lua 库并且运行了 Lua 配置文件，然后你就可以将 Lua 函数的调用封装成下面的 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;
}
```

调用 lua_pcall 时需要指定参数的个数（第二个参数）和返回结果的个数（第三个参数），第四个参数指定了错误处理函数，稍后我们再来讨论它。与 Lua 的赋值操作一样，lua_pcall 会根据你的要求调整返回结果的个数，多余的结果将被丢弃，不足的使用空值来补足。在将返回值入栈之前，lua_pcall 会先将栈内的函数及其参数移除。如果函数返回多个结果，第一个返回值将被头一个入栈，因此如果有 n 个返回值，第一个返回值在栈中的索引为-n，最后一个返回值在栈中的索引为-1。

如果 lua_pcall 运行时出现了错误，lua_pcall 将返回一个非 0 的结果，同时，它还将错误信息入栈（当然，在此之前先将函数及其参数从栈中移除）。如果在调用 lua_pcall 时指定了错误处理函数，那么在将错误信息入栈之前，lua_pcall 会调用该错误处理函数。要指定错误处理函数，我们可以使用 lua_pcall 的最后一个参数，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，如果需要指定错误处理函数，那么最后一个参数应该是该错误处理函数在栈中的索引。注意，在这种情况下，错误处理函数必须要在被调用函数及其参数入栈之前入栈。

对于一般的错误，lua_pcall 返回错误代码 LUA_ERRRUN。在两种特殊情况下，该函数会返回特殊的错误代码，因为此时将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，错误处理函数运行时发生错误，在这种情况下，再次调用错误处理函数显然没有任何意义，所以 lua_pcall 立即返回错误代码 LUA_ERRERR。

25.3 通用的函数调用

下面将讨论一个更高级的例子，我们将借助于 C 语言的 va_arg 函数来封装 Lua 函数的调用。封装后的函数（让我们使用 call_va 来命名它）接受被调用的 Lua 函数作为第一参数，第二参数是一个描述参数和返回值的类型的字符串，然后一个可变的参数列表，最后是一个用于保存返回值的指针列表。有了这个函数，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”表示该 Lua 函数有两个 double 类型的参数和一个 double 类型的返回值。我们使用字母“d”表示双精度类型、“i”表示整数类型、“s”表示字符串类型，“>”字符作为参数和返回值的分隔符。如果函数没有返回值，则“>”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);
}
```

尽管上述函数具有通用性，但该函数与先前的例子在设置步骤上一致：函数入栈、参数入栈、调用函数、获取返回值。大部分代码都很直观，但还是需要注意某些细节。首先，不需要检查 func 参数是否是一个函数，lua_pcall 可以捕捉到这个错误。第二，因为例子中的函数可以接受任意多个参数，因此必须检查栈空间是否足够。第三，因为可能返回字符串，call_va 不能从栈中弹出结果，需要在获取临时字符串的结果之后（拷贝到其他的缓冲区），由调用者负责弹出该字符串类型结果。

第 26 章 Lua代码调用C函数

扩展 Lua 的基本方法之一就是在 Lua 中注册新的 C 函数。

当我们提到 Lua 可以调用 C 函数，不是指 Lua 可以调用任何 C 函数（有一些包可以让 Lua 调用任何 C 函数，但缺乏可移植性和健壮性）。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值。相似的，C 函数被 LuaLua 调用时，C 函数也必须遵循一些协议来取得参数和给出函数的结果。另外，想要用 Lua 调用 C 函数，我们必须先注册 C 函数，也就是说，我们必须将 C 函数的地址以一个适当的方式传递给 Lua。

当 Lua 调用 C 函数的时候，它使用一个与 C 调用 Lua 时相同类型的栈来交互，C 函数从栈中获取参数并将函数的结果压栈。为了区分函数的结果与栈中的其他的值，每个 C 函数还会返回结果的个数。这儿有一个重要的概念：用来交互的栈不是全局的，每一个函数都有它自己的私有局部栈。当 Lua 调用 C 函数时，第一个参数总是在这个局部栈索引为 1 的位置。甚至当一个 C 函数调用一段 Lua 代码，而 Lua 代码反过来又调用该 C 函数或者其他 C 函数时，每一次调用都有它自己独立的私有栈，且它的第一个参数在索引为 1 的位置。

26.1 C函数

先看一个简单的例子，我们将实现一个能够返回给定数值的正弦值的函数（更专业的做法应该在函数中检查它的参数是否为数值类型）：

```
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 中注册的函数必须拥有相同的原型，该原型 lua_CFunction 在 lua.h 中声明：

```
typedef int (* lua_CFunction)(lua_State * L);
```

从 C 的角度来看，一个 C 函数接受单个参数 lua_State（Lua 状态），并返回一个表示结果（在 Lua 中的返回值）个数的数值。因此，函数在将返回值入栈之前不需要清理栈，在函数返回之后，Lua 会自动清除栈中位于返回结果下面的所有内容。

在 Lua 中使用该函数之前，必须注册这个函数。我们使用 lua_pushcfunction 函数来完成这个任务：它获取指向 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。一个粗略的用于测试这段代码的方案是将这段代码直接放到 lua.c 文件中，并在调用 lua_open 之后添加两行：

```
lua_pushcfunction(l, l_sin);
lua_setglobal(l, "mysin");
```

第一行将类型为 function 的值入栈，第二行将该给 Lua 量 mysin。只要在修改之后重新编译 Lua 解释器，你就可以在 Lua 程序中使用新的 mysin 函数了。在下一节中，我们将讨论一个能够将 C 函数添加到 Lua 的更好的方法。

对于更专业的正弦函数，我们必须检查函数的参数类型。辅助库中的 luaL_checknumber 函数可以检查给定的参数是否为数值类型：如果该参数不是数值类型，则抛出一个错误信息，否则，返回作为参数的数值。将上面的函数稍作修改后我们得到了：

```
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')将会得到如下信息：

```
bad argument #1 to 'mysin' (number expected, got string)
```

注意 luaL_checknumber 是如何产生上述错误信息的：参数编号为 1，函数名为“mysin”，期望的参数类型为数值型，实际得到的参数类型为字符串型。

下面看一个稍微复杂的例子：编写一个能够返回给定目录内容的函数。Lua 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。在此，将假定我们的系统符合 POSIX 标准。我们的 l_dir 函数接受一个代表路径的字符串作为参数，并以数组的形式返回该路径的内容。比如，当我们调用 dir("/home/lua")时，其返回值可能类似于{".", "..", "src", "bin", "lib"}。如果发生错误，该函数将返回空值和一个用于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 */
}
```

与辅助库的 luaL_checknumber 函数类似，luaL_checkstring 函数用于检测参数是否为字符串类型。

在极端情况下，上例中的 l_dir 函数可能会导致小规模的内存泄漏。上例中调用的三个 Lua 函数可能会由于内存不足而失败：lua_newtable、lua_pushstring、lua_settable，因此只要其中之一运行失败就会抛出错误并终止 l_dir，这种情况下，closedir 不会被调用。正如前面我们所讨论过的，对于大多数程序来说这不算问题：如果程序内存不足，最好的处理方式就是立即终止程序。然而，在第 29 章我们将看到另外一种实现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 函数的代码段，这些函数需要被保存在适当的地方，通常使用 Lua 表来保存它们。Lua 的 C 函数库就是这样实现的，除了定义相应的 C 函数之外，还必须定义一个特殊的用来和 Lua 库的主代码段通信的特殊函数。一旦调用这个函数，C 函数库中的所有函数都将被注册，并保存到适当的位置。像 Lua 的主代码段一样，该函数也会进行函数库的初始化。

通过注册的过程，Lua 就可以看到库中的 C 函数了。一旦 C 函数被注册并保存到 Lua 中， Lua 程序中就可以直接引用它的地址（注册这个函数的时候传递给 Lua 的地址）来调用这个函数了。换句话说，一旦 C 函数被注册之后，Lua 要调用这个函数就不需要依赖于函数名、包的位置或函数的可见规则。通常 C 库都有一个外部（public/extern）的用来打开库的函数，而其他的函数可能都是私有的，它们在 C 中被声明为 static 静态类型。

当你打算用 C 函数来扩展 Lua 时，即使仅需注册一个 C 函数，将你的 C 代码设计为一个库是个比较好的做法：迟早你就会发现你还需要注册其他函数。正如所期望的，辅助库为此提供了一个辅助函数，函数 luaL_openlib 接受一系列的 C 函数及其函数名作为参数，并用指定的库名称在一个 Lua 表中注册所有的函数。在下面的例子中，我们将使用前面介绍过的 l_dir 函数来创建一个库。首先，我们必须定义库函数：

```
static int l_dir(lua_State * L) {
    ...             /* as before */
}
```

第二步，我们将声明一个数组来保存函数中所有的函数以及与之对应的名字。这个数组的元素类型为 luaL_reg，这是一个带有两个域的结构：一个字符串和一个函数指针。

```
static const struct luaL_reg mylib[] = {
    {"dir", l_dir},
    {NULL, NULL}  /* sentinel */
};
```

在这个例子中，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}，它用来表示需要注册的库函数到此结束。最后，我们将使用 luaL_openlib 函数来定义主函数：

```
int luaopen_mylib(lua_State * L) {
    luaL_openlib(L, "mylib", mylib, 0);
    return 1;
}
```

函数 luaL_openlib 的第二个参数是库名称。这个函数按照指定的库名称创建（或重用）一个 Lua 表，并使用数组 mylib 中的“名字-函数”对来填充这个表。函数 luaL_openlib 还允许我们为库中的所有函数注册公共的 Upvalue，因为例子中没有用到 Upvalue，所以最后一个参数为 0。函数 luaL_openlib 返回的时候，将用于保存库的 Lua 表放到栈内，并返回 1 来通知 C 程序栈内有 1 个结果将返回给 Lua。对于 Lua 库来说，这个返回值是可选的，因为库本身已经赋给了一个 Lua 全局变量。同样地，像在 Lua 标准库中一样，这个返回不会有额外的开销，在有的时候可能还是很有用的。

在 C 函数库建立之后，还必须将它连接到 Lua 解释器。最方便的方式是使用动态链接机制，如果你的 Lua 解释器支持这个特性的话（我们在 8.2 章节已经讨论过动态链接）。在这种情况下，你必须利用代码创建动态链接库（Windows 下为.dll 文件，Linux 下为.so 文件）。之后，你就可以在 Lua 中直接使用函数 loadlib 来加载函数库了：

```
mylib = loadlib("fullname-of-your-library", "luaopen_mylib")
```

上述调用将 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 变量（这就是为什么 luaopen_mylib 必须和其他的 C 函数有相同的原型的原因所在）。然后就可以调用 mylib() 来运行 luaopen_mylib 以便打开函数库。

如果你的解释器不支持动态链接，你必须使用函数库重新编译 Lua 解释器。除此以外，还需要告诉 Lua 解释器，当它打开一个新的状态后，必须打开这个新定义的函数库。宏定义可以很容易完成此任务。首先，你必须使用下面的内容创建一个头文件（我们将其命名为 mylib.h）：

```
int luaopen_mylib(lua_State * L);

#define LUA_EXTRALIBS { "mylib", luaopen_mylib },
```

第一行声明了用于打开库的函数。第二行定义了一个宏 LUA_EXTRALIBS 作为函数列表中的一项，当解释器创建新的状态的时候会调用包含这个宏的函数列表（包含了这个函数列表的数组的类型为 struct luaL_reg[]，因此我们需要给出名字）。

要在解释器中包含这个头文件，你可以在编译选项中定义一个宏 LUA_USERCONFIG，而对于命令行的编译器，你只需添加下面的选项即可：

```
-DLUA_USERCONFIG=\"mylib.h\"
```

反斜杆防止双引号被操作系统外壳程序（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，它已经包含了 mylib.h 头文件，因此，在需要由解释器打开的函数库列表中，就能使用新定义的 LUA_EXTRALIBS 了。

第 27 章 编写C函数的技巧

正式的 API 和辅助函数库都提供了一些能够帮助程序员编写 C 函数的机制。在这一章中，我们将讨论有关数组操作、字符串处理、在 C 中存储 Lua 值等一些特殊的机制。

27.1 数组操作

Lua 中的“数组”实际上就是以特殊方式使用的 Lua 表的别名。我们可以使用任何能够操纵 Lua 表的函数来操纵数组，即 lua_settable 和 lua_gettable。然而，与 Lua 推崇的简洁思想相反的是，API 为数组操作提供了特殊的函数。这样做是出于性能的考虑：因为我们经常在一个算法（比如，排序）的内层循环中访问数组，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 的描述有点令人糊涂，因为它涉及到两个索引：index 指向表在栈中的位置，key 指向元素在表中的位置。调用 lua_rawgeti(L, t, key)等价于：

```
lua_pushnumber(L, key);
lua_rawget(L, t);
```

如果 t 使用正索引，调用 lua_rawseti(L, t, key)等价于（如果 t 使用负索引，在新元素压栈之后你必须对 t 的值进行调整）：

```
lua_pushnumber(L, key);
lua_insert(L, -2);       /* put 'key' below previous value */
lua_rawset(L, t);
```

注意这两个函数都使用 raw 操作，它们的速度较快，总之，用作数组的表很少使用元方法。

下面是一个有关如何使用这些函数的具体例子，我们将重写 l_dir 函数的循环体，原来的代码为：

```
lua_pushnumber(L, i++);                /* key */
lua_pushstring(L, entry->d_name);      /* value */
lua_settable(L, -3);
```

改写后为：

```
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，并将该元素替换为指定函数的结果。

```
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 */
}
```

在上例中引入了三个新的函数。函数 luaL_checktype（在 lauxlib.h 中声明）用来检查给定的参数是否为指定的类型，如果不是，则抛出错误。函数 luaL_getn 取栈中指定位置的数组的大小（table.getn 是调用 luaL_getn 来实现的）。函数 lua_call 在非保护模式下运行，它与 lua_pcall 相似，但在发生错误时抛出错误而不是返回错误代码。当你在应用程序中编写主线程代码时，不应该使用 lua_call 函数，因为你需要捕捉任何可能发生的错误。当你编写一个函数代码时，使用 lua_call 函数通常是比较好的做法，如果有错误发生，把错误留给关心它的人处理。

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时不要将其出栈，并且绝不要修改字符串。

当 C 函数需要创建一个字符串并返回给 Lua 的时候，情况变得更加复杂。这需要由 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。上文介绍过的字符串压栈函数 lua_pushlstring 需要接受字符串的长度作为参数，所以，如果想要将字符串 s 第 i 个字符到第 j 个字符间的子串传递给 Lua，你只需要调用：

```
lua_pushlstring(L, s + i, j - i + 1);
```

下面例子中的函数可以根据给定的分隔符对一个字符串进行分割，并返回一个保存所有子串的表，如下列调用：

```
split("hi,,there", ",")
```

将返回表{"hi", "", "there"}。我们以下列方式实现这个函数，该函数不需要额外的缓冲区，且能够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 */
}
```

Lua API 提供了专门的用来连接字符串的函数 lua_concat，该函数等价于 Lua 中的“..”操作符：自动将数字转换成字符串，必要时自动调用待连接对象的元方法。另外，它可以同时连接多个字符串，调用 lua_concat(L, n)将连接栈顶的 n 个值（同时将其出栈），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, ...);
```

这个函数在某种程度上类似于 C 语言中的 sprintf，它根据格式串 fmt 和某些额外的参数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区，Lua 将根据所需的长度动态创建新的字符串，因此无需担心缓冲区溢出等问题。这个函数会将作为结果的字符串放到栈内，并返回一个指向该字符串的指针。就目前而言，该函数只支持下列几个控制字符：%%表示字符“%”、%s 由于格式化字符串、%d 用于格式化整数、%f 用于格式化 Lua 数值，即 doubles 类型、%c 用于接受一个整数并将其格式化为字符。该函数不支持输出宽度和精度控制等选项。

连接少量的字符串时，函数 lua_concat 和 lua_pushfstring 是很实用用的，然而，如果需要连接大量的字符串（或字符），这种逐个连接的方式效率是很低的，正如我们在第 11.6 章节看到的那样。我们可以使用辅助库提供的缓冲函数来解决这个问题，辅助库为此提供了两种方案。第一种类似于 I/O 操作的缓冲区：获取所有字符串（或单个字符），将其存入一个本地缓存区中，当本地缓冲区满的时候再将其传递给 Lua（使用 lua_pushlstring）。第二种使用 lua_concat 和第 11.6 章节中栈算法的变体，来连接多个缓冲结果。

为了更详细地描述辅助库中缓冲函数的用法，让我们先来看一个简单的应用。下列代码演示了函数 string.upper 的实现方法（摘自 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;
```

```
}
```

使用辅助库中缓冲函数的第一步是声明一个类型为 luaL_Buffer 的变量，然后调用 luaL_buffinit 初始化该变量，即初始化缓冲区。初始化之后，缓冲区保留了一份 Lua 状态 L 的拷贝，因此当我们调用其他操作该缓冲区的函数时就不再需要传递 L 参数。宏 luaL_putchar 将单个字符放入缓冲区。同时，辅助库也提供了 luaL_addlstring 缓冲函数，该函数将一个指定长度的字符串放入缓冲区，而另一个类似的缓冲函数 luaL_addstring 将一个以“\0”结尾的字符串放入缓冲区。最后，luaL_pushresult 刷新缓冲区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);
```

有了这些函数，我们就不需要担心缓冲区的分配、溢出等细节问题。正如我们所看到的，上述连接算法是十分高效的，函数 str_upper 可以快速地处理大字符串（大于 1MB）。

使用辅助库中的缓冲函数的时候，你必须关注一个细节问题。当你将数据放入缓冲区的时候，程序需要在栈中保存某些中间结果。所以，不要认为栈顶会保持一开始使用缓冲区的状态。另外，尽管在使用缓冲区的时候，你仍然可以使用栈，但每次在访问。

只在一种情况下，这个限制显得过于苛求：将一个由 Lua 返回的结果放入缓冲区（即，此时该结果正位于栈顶）。此时，在将字符串值放入缓冲区之前，你不能将其出栈，因为你无法使用一个已经出栈的字符串值；另一方面，在将字符串值出栈之前，你不能够将其放入缓冲区，因为在出栈之前更新缓冲区将导致栈处于错误的状态（根据前面所说的：在每次访问缓冲区之前，你必须保证压栈与出栈的次数一致），换句话说你执行类似下面的操作：

```
luaL_addstring(&b, lua_tostring(L, 1));      /* BAD CODE */
```

基于上述情况的普遍性，辅助提供了一个特殊函数，用来将位于栈顶的值放入缓冲区：

```
void luaL_addvalue(luaL_Buffer * B);
```

当然，如果栈顶的值不是字符串或者数值的话，调用这个函数将会出错。

27.3 在C函数中保存状态

经常地，C 函数需要保存一些非局部的数据，也就是指那些超超出其作用域的数据。在 C 语言中我们常使用全局变量或静态变量来满足这种需要。然而，在为 Lua 设计函数库的时候，使用全局变量和静态变量不是一个好方法。首先，你无法将 Lua 值保存到一个 C 变量中，因为 Lua 变量是动态类型的。第二，使用这种变量的函数库不能在多个 Lua 状态的情况下使用，因为多个 Lua 状态将只能共享这些变量。

一个替代的解决方案是使用 Lua 全局变量来保存这些值，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 Lua 状态都有其独立的全局变量。然而，这并不是一个十全十美的解决方案，因为 Lua 代码可能会篡改这些全局变量，从而危及 C 程序的数据完整性。为了避免这个问题，Lua 提供了一个被称为注册表（Registry）的表，C 代码可以自由使用该表，但 Lua 代码却无法访问它。

27.3.1 注册表

Lua 注册表（Registry）所在的位置总是由一个假索引（Pseudo-Index）决定的，该假索引的值由

LUA_REGISTRYINDEX 定义。一个假索引类似栈中的普通索引，只是假索引对应的值并不在栈中。Lua API 中大部分接受索引作为参数的函数也都能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 和 lua_insert。为了从注册表中获取一个索引为“Key”的值，可以使用下列代码：

```
lua_pushstring(L, "Key");
lua_gettable(L, LUA_REGISTRYINDEX);
```

注册表是一个普通的 Lua 表，因此，可以使用任何非空值来索引其中的元素。然而，由于所有的 C 函数库共享同一个注册表，你必须选择合适的值作为索引，以避免发生命名冲突。一个防止命名冲突的方法是使用静态变量的地址作为作为索引值：C 链接器将保证该索引值在所有函数库中都是唯一的。函数 lua_pushlightuserdata 可以将一个 C 指针作为值放入栈内，下面的代码演示了如何使用该方法在注册表中存储以及获取一个数值：

```
/* 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 */
```

我们会在第 28.5 章节中更详细地讨论轻量级的 Userdata（Light Userdata）。

当然，你也可以使用字符串作为注册表的索引值，只要保证这些字符串的唯一性。如果允许其他函数库访问你的数据时，字符串类型的索引值是非常有用的，因为它们需要知道索引的名字。在这种情况下，没有什么方法可以绝对防止命名冲突，但有一些好的做法可供参考，比如，使用函数库的名称作为字符串的前缀，使用“lua”或者“lualib”作为前缀不是一个好的选择。另一个方法是使用通用唯一识别码（Universal Unique Identifier，uuid），很多操作系统都有专门的程序来产生这种识别码（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产生的 128 位的数值（表示为一个 16 进制编码的字符串），由此保证了它与其他 uuid 的不同。

27.3.2 引用

需要注意的是，绝不要使用数值作为注册表的索引值，因为数值类型的索引是专门为引用系统（Reference System）所保留的。引用系统是由辅助库中的一些函数组成的，这对函数允许你在注册表中存储值而不用担心命名冲突，实际上，这些函数可以用于任何 Lua 表，但它们通常被用于注册表。

下列调用语句：

```
int r = luaL_ref(L, LUA_REGISTRYINDEX);
```

将从栈中弹出一个值，以一个独一无二的整数索引值将该弹出值存储到注册表中，并返回这个索引值。我们称这个索引值为引用（Reference）。

顾名思义，Lua 中的引用主要用于：在 C 数据结构中存储一个指向 Lua 值的引用。正如上文所述，绝不应该将一个指向 Lua 字符串的指针保存到获取该指针的 C 函数的外部，而且，Lua 也不提供指向其

他对象的指针，比如 Lua 表或函数。因此，我们不能通过指针来引用 Lua 中的对象，当需要这种类似指针的功能时，我们可以创建一个 Lua 引用并将其保存在 C 数据结构中。

要将一个引用 r 对应的值入栈，只需要执行：

```
lua_rawgeti(L, LUA_REGISTRYINDEX, r);
```

要删除引用及其对应的值，只需要执行：

```
luaL_unref(L, LUA_REGISTRYINDEX, r);
```

执行上述调用后，r 便可以被用作新引用的值。

引用系统对空值进行特殊处理，当你使用 luaL_ref 存储空值的时候，该函数不会创建一个新的引用，而是返回一个引用常量 LUA_REFNIL。因此下面的调用没有任何效果：

```
luaL_unref(L, LUA_REGISTRYINDEX, LUA_REFNIL);
```

而下列调用：

```
lua_rawgeti(L, LUA_REGISTRYINDEX, LUA_REFNIL);
```

像预期的一样，将一个空值入栈。

引用系统也定义了另一个常量 LUA_NOREF，它是一个不同于任何有效引用值的整数值，可以使用该常量来表示无效的引用，任何试图获取 LUA_NOREF 所引用的值的操作都将返回空值，任何试图删除该引用的操作都是无效的。

27.3.3 Upvalue

注册表为函数库全局变量的实现提供了帮助，而 Upvalue 机制则实现了 C 语言中的静态变量的等价物，这种变量只能在特定的函数内可见。当你在 Lua 中创建新的 C 函数时，你可以将该函数与任意多个 Upvalue 关联起来，每一个 Upvalue 可以含有有一个 Lua 值。之后，当函数被调用的时候，可以通过假索引自由地访问这些 Upvalue。

这种 C 函数及其 Upvalue 的关联被称作闭包（Closure）。注意在 Lua 代码中，闭包是一个能够访问外部函数的局部变量的函数。C 闭包是用 C 语言实现的准 Lua 闭包。有关闭包的一个有趣之处是，你可以使用相同的函数代码创建不同的闭包，每个闭包带有不同的 Upvalue。

在下面的例子中，我们将用 C 语言创建一个 newCounter 函数（在第 6.1 章节已经定义过同样的 Lua 函数），该函数是一个函数工厂：每次调用该函数都将返回一个新的 counter 函数。尽管所有的 counter 函数共享相同的 C 代码，但是每个 counter 函数都保有独立的计数器。该工厂函数代码如下：

```
/* forward declaration */
static int counter(lua_State * L);

int newCounter(lua_State * L) {
    lua_pushnumber(L, 0);
    lua_pushcclosure(L, &counter, 1);
    return 1;
}
```

上列中的关键在于 lua_pushcclosure 函数，该函数将创建一个新的闭包。它的第二个参数是一个普通的函数（例子中为 counter 函数），第三个参数指明了 Upvalue 的个数（例子中为 1）。在创建新闭包之前，我们必须将 Upvalue 的初始值入栈，在上述例子中，我们将数值 0 作为唯一的 Upvalue 的初始值入栈。如预期的一样，函数 lua_pushcclosure 将新闭包放入栈内，此时该闭包已经作为 newCounter 函数的结果被留在栈内。

现在，让我们来完成 counter 函数的定义：

```
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（实际上是一个宏），它被用来产生一个 Upvalue 的假索引，该与其他栈索引一样，只是它不在栈中。表达式 lua_upvalueindex(1)代表闭包函数第一个 Upvalue 的索引，因此，函数 lua_tonumber 将获取仅有的 Upvalue 的当前值并将其转换为数值，之后，counter 函数将新值++val 压栈、拷贝该新值，并用该拷贝的值替换 Upvalue 的值，最后，压栈的新值将被作为结果返回。

与 Lua 闭包不同的是，C 闭包不能共享 Upvalue：每一个闭包都拥有独立的变量集。然而，可以使用同一个 Lua 表为不同闭包函数的 Upvalues 赋值，这样该 Lua 表就变成了一个所有闭包函数的数据共享区。

第 28 章 C语言中的自定义类型

在上一章中，我们使用 C 函数来扩展 Lua，现在我们将使用 C 语言中的自定义类型来扩展 Lua。我们从一个简单的例子开始，本章后续部分将以这个例子为基础逐步加入元方法（Metamethods）等其他内容。

该例子涉及一个简单的类型：数值数组。选择这个例子是因为它不涉及复杂的算法，因此我们能够将注意力集中到 API 问题上。尽管例子中的自定义类型很简单，但在某些应用程序中都会用到这种自定义类型。一般情况下，Lua 中并不需要引入外部的数组类型，因为哈希表很好地实现了数组的功能。但是对于非常大的数组而言，哈希表可能会导致内存不足，因为哈希表必须为每一个元素保存一个范性的（Generic）值、一个链接地址（指针）和一些以备将来增长的额外空间，而在 C 中直接存储数值不需要额外的空间，这比使用哈希表节省了约 50%的内存空间。

我们使用下面的结构表示自定义类型的数值数组：

```
typedef struct NumArray {
    int size;
    double values[1];    /* variable part */
} NumArray;
```

我们声明数组的 values 字段大小为 1 作为占位符，是因为 C 语言不允许大小为 0 的数组，之后我们会定义实际的数组大小。对于一个拥有 n 个 values 元素的数组来说，我们共需要

```
sizeof(NumArray) + (n - 1) * sizeof(double)
```

字节。由于原始的 NumArray 结构中已经包含了 1 个 values 元素的空间，所以我们从 n 中减去 1。

28.1 Userdata

接下来我们首先要关心的是如何在 Lua 中表示新定义的数值数组。Lua 为此提供了一个专用的类型：Userdata（用户自定义数据）。这种 Userdata 类型提供了一块没有预定义任何操作的内存空间。

Lua API 提供了下列函数来创建 Userdata：

```
void * lua_newuserdata(lua_State * L, size_t size);
```

函数 lua_newuserdata 按照指定大小分配一块内存空间，将对应的 Userdata 放入栈内，并返回该内存块的地址。如果出于某些原因，需要通过其他方式分配内存，你会发现很容易创建一个指针大小的 Userdata，然后将指向实际内存空间的指针保存到该 Userdata 里。在下一章我们将看到该技术的例子。

使用 lua_newuserdata 函数创建自定义类型的数值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 userdata is already on the stack */
}
```

函数 luaL_checkint 是数值检查函数 luaL_checknumber 的变体。一旦上述 newarray 函数被注册，

你就可以在 Lua 中使用类似 a = array.new(1000)的语句创建自定义类型的数值数组了。

要在该数组中存储元素，我们使用类似 array.set(array, index, value)的调用语句，后面我们将看到如何使用元表来支持类似 array[index] = value 的常规写法。无论采用何种写法，下面的函数将实现该功能，与以往一样，我们假定该数值数组的索引从 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;
}
```

函数 luaL_argcheck 用于检查给定的条件，如果条件为假则抛出错误，该函数的原型为：

```
void luaL_argcheck(lua_State * L, int cond, int narg, const char * extramsg);
```

其中，参数 narg 用于在错误信息中提示参数的编号。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)
```

下面的函数可以从上述数值数组中获取一个元素：

```
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;
}
```

而下列函数可以获取该数值数组的大小：

```
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，我们还需要一些额外的代码来初始化库：

```
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;
}
```

在上述代码中，我们再次使用了辅助库的 luaL_openlib 函数，它根据给定的名字（例子中为“array”）创建一个表，并使用变量 arraylib 对应的数组中的“名字-函数”对来填充这个表。

当打开上述库之后，我们就可以在 Lua 中使用新定义的数值数组类型了：

```
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
```

针对上述自定义类型的数值数组，在某 Pentium/Linux 环境下，利用 C 代码直接产生一个含有 100K 元素的该数值数组大概需要 800KB 的内存，而在同样的运行环境下使用 Lua 表来产生含有相同元素个数的数值数组需要 1.5MB 的内存。

28.2 元表

上一节中数值数组的实现存在重大的安全漏洞。假设用户执行这样的调用：array.set(io.stdin, 1, 0)，其中，io.stdin 是一个含有文件流指针（FILE *）的 Userdata，正因为它是一个 Userdata，array.set 将很乐意接受它作为参数，运行该代码很有可能导致内存错误（幸运的话，你可能只是得到一个访问越界的错误）。这样的错误对于任何一个 Lua 库来说都是不能接受的，不论你如何使用一个 C 函数库，都不应该破坏 C 数据或在 Lua 代码中发生核心内存转储（Core Dump，在系统崩溃时，将内存中的数据转储于文件中，供给有关人员进行排错分析）。

为了区分自定义的数值数组和其他类型的 Userdata，我们将为该数值数组创建一个唯一的元表（注意 Userdata 也可以拥有元表），之后，每当我们创建一个新的数值数组时，我们将使用这个元表来标记该数值数组，而每次我们访问该数值数组时，我们都要检查它是否拥有正确的元表。因为 Lua 代码不能改变 Userdata 的元表，所以它无法仿造用于设置元表的 C 代码。

我们需要一个地方来保存这个新的元表，之后才能够在创建数值数值时取得该元表，也才能够在检查某个给定的 Userdata 是否是数值数组时访问该元表。正如我们前面介绍过的，有两种方法可以保存该元表：在注册表中，或在库中作为函数的 Upvalue。Lua 的惯常做法是，将类型名作为索引，将元表作为值，在注册表中注册新的 C 类型。就像在注册表中注册其他索引一样，我们必须尽量选择一个唯一的类型名以避免冲突。在此，我们将使用“LuaBook.array”作为新类型的名称。

与往常一样，辅助库提供了一些相关的函数，我们将要用到的函数有：

```
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);
```

函数 luaL_newmetatable 可以创建一个新表（该表将被用作元表），将该表放到栈顶，并建立表和注册表中给定类型名的关联。这个关联是双向的：使用类型名作为表的索引，同时使用表作为类型名的索引（这种双向的关联，使得其他两个函数的实现效率更高）。函数 luaL_getmetatable 获取注册表中与类型名 tname 对应的元表。而函数 luaL_checkudata 检查栈中指定位置的对象是否为带有给定名字的元表的 Userdata，如果该对象不带有这样的元表或该对象不是一个 Userdata，函数返回 NULL，否则，返回 Userdata 的地址。

下面来看具体的实现。第一步我们将修改打开库的函数，新版本的该函数必须创建一个用作数值数组元表的表：

```
int luaopen_array(lua_State * L) {
    luaL_newmetatable(L, "LuaBook.array");
    luaL_openlib(L, "array", arraylib, 0);
    return 1;
}
```

下一步，我们需要修改 newarray 函数，以便将上述元表设置为新创建的数值数组的元表：

```
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 userdata is already on the stack */
}
```

函数 lua_setmetatable 将表出栈，并将其设置为给定位置的对象的元表。在我们的例子中，这个对象就是新的 Userdata。

最后，setarray、getarray 和 getsize 检查它们的第一个参数是否为一个有效的数值数组，在参数错误的情况下我们将令这些函数抛出一个错误，首先我们需要定义一个辅助函数：

```
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;
}
```

使用上述 checkarray 函数，将令新定义的 getsize 是更为直观和清楚：

```
static int getsize(lua_State * L) {
    NumArray * a = checkarray(L);
    lua_pushnumber(L, a->size);
    return 1;
}
```

由于函数 setarray 和 getarray 检查第二个参数 index 的代码相同，因此可以将其共同的部分抽出来定义一个独立的函数：

```
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];
}
```

使用上述新定义的 getelem 函数，可以令函数 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，如果执行类似 array.get(io.stdin, 10)这样的代码，你将会得到正确的错误信息：

```
error: bad argument #1 to 'getarray' ('array' expected)
```

28.3 面向对象的访问方式

接下来我们将把上文创建的数值数组类型转换为一个对象，这样就可以使用面向对象的语法来操作对象的实例：

```
a = array.new(1000)
print(a:size())      --> 1000
a:set(10, 3.4)
print(a:get(10))     --> 3.4
```

注意 a:size()等价于 a.size(a)，因此，我们必须令表达式 a.size 能够调用 getsize 函数，这里的关键在于__index 元方法的使用。对于 Lua 表来说，不论何时只要找不到指定的索引对应的值，该元方法就会被调用。对于 Userdata 而言，每次访问 Userdata 时该元方法都会被调用，因为 Userdata 根本没有任何索引。

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
```

上述代码中，第一行，我们创建一个数组只是为了获取它的元表，元表被赋值给 metaarray 变量（我们不能在 Lua 中设置 Userdata 的元表，但却可以在 Lua 中无限制地取得 Userdata 的元表）。接下来，我们将 metaarray.__index 赋值为 metaarray。当我们计算 a.size 的时候，Lua 在对象 a 中无法找到名为“size”索引，因为该对象是一个 Userdata。因此，Lua 试着从对象 a 的元表的__index 域获取这个值，正好__index 就是 metaarray，而 metaarray.size 就是 array.size，因此 a.size(a)如我们预期的那样返回 array.size(a)。

当然，我们可以用 C 代码完成同样的事情，甚至可以做得更好：既然数值数组是拥有自定义操作的对象，那么在表 array 中就不再需要这些操作了。函数库中唯一需要对外公布的函数是 new，该函数用于创建新的数值数组。所有其他的操作将被作为对象方法来实现，而 C 代码可以直接注册它们。

对象方法 getsize、getarray 和 setarray 与前面的实现一样，不需要改变，需要改变的只是如何注册它们。也就是说，我们必须修改打开库的函数，首先，我们需要两个独立的函数列表，其中的一个作为普通函数列表，一个作为对象方法列表：

```
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}
};
```

新版本的用于打开库的函数 luaopen_array，必须创建一个元表，并将其赋值给自身的__index 域、在该域注册所有的方法、创建并初始化 array 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;
}
```

在此，我们使用了函数 luaL_openlib 的另一个特性。第一次调用该函数时，当我们传递 NULL 值作为库名时，luaL_openlib 并没有创建任何包含函数的表，相反，它假定这个用于封装函数的表在栈内，位于任何可能的 Upvalues 之下。在这个例子中，用于封装函数的表是元表本身，也就是 luaL_openlib 封装方法的地方。第二次对函数 luaL_openlib 的调用以正常方式工作：根据给定的名称（例子中为“array”）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）。

最后，我们将在新类型中添加一个__tostring 方法，这样一来执行 print(a)将打印数组“array(size)”，其中 size 为数组 a 的大小：

```
int array2string (lua_State * L) {
    NumArray *a = checkarray(L);
    lua_pushfstring(L, "array(%d)", a->size);
    return 1;
}
```

上例子中，函数 lua_pushfstring 将产生格式化字符串，并将其放到栈顶。为了在数值数组对象的元表中包含 array2string 方法，我们还必须在 arraylib_m 列表中添加它：

```
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，即采用 a[i]的写法，而不是 a:get(i)，就上述例子而言，这是很容易实现的，因为函数 setarray 和 getarray 已经依次接受了与的元方法对应的参数。一个快速的解决方案是在 Lua 代码中定义这些元方法：

```
local metaarray = getmetatable(newarray(1))
metaarray.__index = array.get
metaarray.__newindex = array.set
```

这段代码必须在最初的数值数组的实现基础上运行，而不是修改后的以面相对象方式访问的基础上。我们要做的只是使用下列通用的语法：

```
a = array.new(1000)
a[10] = 3.4              -- setarray
print(a[10])             -- getarray   --> 3.4
```

只要愿意，我们还可以在 C 代码中注册这些元方法。为此，只需要修改初始化函数即可：

```
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 轻量级的Userdata

到目前为止，一直在使用的这类 Userdata 被称为完整的 Userdata（Full Userdata）。Lua 还提供了另一种 Userdata，被称为轻量级的 Userdata（Light Userdata）。

轻量级的 Userdata 是一个用于表示 C 语言指针的值（即 void *类型的值）。正因为它是一个值，所以我们并不创建这样的值（正如我们不会去创建一个数值那样）。可以使用函数 lua_pushlightuserdata 将一个轻量级的 Userdata 入栈：

```
void lua_pushlightuserdata(lua_State * L, void * p);
```

尽管同为 Userdata，轻量级的 Userdata 和完整的 Userdata 有很大不同：轻量级的 Userdata 不是缓冲区，而是指针，它不具有元表。像数值一样，轻量级的 Userdata 不需要垃圾收集器对其进行回收。

有些人把轻量级的 Userdata 作为一个低成本的完整的 Userdata 来使用，但是这不是轻量级的 Userdata 的典型应用。首先，使用轻量级的 Userdata 时你必须自行管理内存，因为垃圾收集器不会对其进行管理。第二，尽管从名字上看有轻重之分，但完整的 Userdata 的开销也不大，只是在分配内存时有少许额外的开销。

轻量级的 Userdata 的主要应用在于可以对其进行比较操作。因为完整的 Userdata 是对象，它只与其自身相等，而轻量级的 Userdata 表示指向对象的指针，它等于任何表示同一指针的 Userdata。因此，可以在 Lua 中使用轻量级的 Userdata 查找 C 对象。

作为一个典型的例子，假定我们要绑定 Lua 和某窗口系统。这种情况下，我们使用完整的 Userdata 来表示窗口（每一个 Userdata 可以包含整个窗口结构或只是一个由系统创建的指向该窗口的指针）。如果遇到窗口事件（比如，鼠标单击），系统会根据窗口地址调用专门的回调函数。为了将这个回调函数传递给 Lua，我们必须找到表示该窗口的 Userdata，为此，可以使用一个特殊表：该表的索引为表示窗口地址的轻量级的 Userdata，值为表示窗口的完整的 Userdata，一旦取得了窗口地址，可将其作为轻量级的 Userdata 放入栈内，并且将该 Userdata 作为该特殊表的索引（注意这个特殊表的值为 Weak 性质，否则，那些完整的 Userdata 将永远不会被回收）。

第 29 章 资源管理

在前一章的数值数组的实现方法中，除了内存之外，我们不必担心资源管理问题，每一个表示数值数组的 Userdata 都有自己的内存，这些内存由 Lua 管理。当数值数组成为垃圾数据时（即，无法为程序所用），最终会由 Lua 自动回收并将其释放。

通常，情况并不总是那么简单，除了物理内存之外，对象还可能需要文件描述符、窗口句柄等资源（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回收的时候，相关的资源应该被释放掉，一些面向对象的语言为此提供了一种特殊的机制（被称为 Finalizer 或 Destructor，析构器），而 Lua 以__gc 元方法的方式提供了相应的机制，该元方法只能应用于 Userdata 类型的值。当 Userdata 即将被收集时，并且该 Userdata 有一个__gc 域，Lua 会以该 Userdata 作为参数来调用这个域的值（该域必须是一个函数）。这个函数将负责相关资源的释放。

为了说明该元方法和 API 的整体应用，我们将在这一章通过两个例子来展示 Lua 的扩展能力。第一个例子对上文已介绍过的目录遍历程序的另一种实现。第二个更具实际意义的例子将通过 Expat（Expat 是一个开源的 XML 解析器）来扩展 Lua。

29.1 目录迭代器

上文我们已经实现了 dir 函数，该函数以指定的目录作为参数，返回一个包含该目录下所有文件的 Lua 表。下面将要实现的新版本的 dir 函数将返回一个迭代器，每次调用该迭代器都会返回目录中的一条记录（Entry）。根据新的实现方式，我们可以使用下列循环语句来遍历整个目录：

```
for fname in dir(".") do
    print(fname)
end
```

在 C 语言中，需要 DIR 结构才能够迭代一个目录，调用 opendir 函数能够创建 DIR 结构的实例，之后必须显式地调用 closedir 函数才能被释放。旧版的 dir 函数使用局部变量保存 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。在下文新版的 dir 函数中不能使用局部变量来保存 DIR 的实例，因为在之后多次调用中都需要访问这个值，此外，也不能在获取目录中最后一个文件名之后才关闭目录，因为如果循环过程中发生中断，那么迭代器将无法取得最后一个文件名，因此，为了保证 DIR 的实例一定能够被释放，可以将 DIR 结构的地址保存在 Userdata 中，并使用该 Userdata 的__gc 的元方法来释放目录结构。

尽管 Userdata 在下面的实现中很重要，但用以表示目录的 Userdata 并不需要在 Lua 中可见，函数 dir 将返回一个迭代器，而该迭代器需要在 Lua 中可见，含有目录信息的变量可能是迭代器的 Upvalue。这样一来，迭代器就可以直接访问这个目录结构，但 Lua 却不可以（也不需要）访问该结构。

总的来说，我们需要三个 C 函数。第一，dir 工厂函数，它打开 DIR 结构并将其作为迭代器的 Upvalue，Lua 调用该函数产生迭代器。第二，迭代函数。第三，__gc 元方法，它负责关闭 DIR 结构。与以往一样，还需要一个额外的函数来初始化参数，比如，为目录创建元表，并初始化该元表。

首先定义 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 userdata 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 userdata,
       is already on the stack top */
    lua_pushcclosure(L, dir_iter, 1);
    return 1;
}
```

需要注意的是，必须在打开目录之前创建 Userdata。如果先打开目录，然后调用 lua_newuserdata 函数将会抛出错误，从而无法获取 DIR 结构。按照正确的顺序，DIR 结构一旦被创建，便立刻与 Userdata 建立关联，之后不管发生什么，__gc 元方法都将最终释放该结构。

第二个函数即迭代器本身：

```
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 */
}
```

元方法__gc 用于关闭目录结构，但必须注意：因为 Userdata 需要在打开目录之前被创建，所以不管 opendir 函数的结果是什么，Userdata 都将被回收。如果 opendir 函数失败，那么也就不存在任何需要关闭的结构：

```
static int dir_gc(lua_State * L) {
    DIR * d = *(DIR **)lua_touserdata(L, 1);
    if(d)
        closedir(d);
    return 0;
}
```

最后一个所需的函数用于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;
}
```

上述例子中需要引起注意的是，或许，函数 dir_gc 需要检查它的参数是否为目录，不然，一个恶意的使用者可能会使用其他类型的 Userdata（比如，文件）来调用该函数，这将导致严重的后果，但实际上 Lua 程序并不能访问该函数：该函数被存放在目录 Userdata 的元表中，Lua 程序无法访问该 Userdata。

29.2 XML解析器

在这一章节，我们将介绍 lxp（Lua XML Parser）的一个简单的实现，它该实现方案结合了 Lua 和 Expat，Expat 是一个由 C 语言编写的开源的 XML 1.0 解析器。它实现了 SAX（Simple API for XML），SAX 是基于事件的 API，这意味着在 SAX 解析器读取 XML 文档时，它将通过回调函数向应用程序反馈它所读取的内容。举个例子，如果通过 Expat 解析下列字符串：

```
<tag cap="5">hi</tag>
```

它将会产生三个事件：读取“<tag cap="5">”时，将产生“start-element”事件；读取“hi”时，将产生“text”事件（有时也被称为“character data”事件）；读取“</tag>”时，将产生“end-element”事件。每个事件都将调用相应的回调函数的句柄（Callback Handler）。

在此，我们不会涵盖整个 Expat 库的内容，而只是集中精力关注 Expat 中与 Lua 进行交互的技术。在实现核心功能之后，再往其中添加特色功能并不困难。尽管 Expat 能够处理众多的事件，但我们将只考虑上述三种事件（start-element、text 以及 end-element），在 Expat API 中我们所需要的只是一小部分，首先，我们需要能够创建和析构 Expat 解析器的函数：

```
#include <xmlparse.h>

XML_Parser XML_ParserCreate(const char * encoding);
void XML_ParserFree(XML_Parser p);
```

参数 encoding 是可选的，在本文的实现中，我们将直接选用 NULL 作为参数。

有了解析器之后，我们还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);
```

第一个函数注册了“start-element”和“end-element”事件的句柄，第二个函数注册了“text”（即XML 术语中的 character data）事件的的句柄。

所有回调函数的句柄接受某些用户数据作为第一参数。start-element 事件的句柄还接受 XML 标签名以及该标签对应的若干属性作为参数：

```
typedef void (* XML_StartElementHandler)(void * uData, const char * name,
                                         const char ** atts);
```

这些属性表示为一个 C 风格字符串（以“\0”字符结尾）的数组，该数组中，每两个连续的字符串分别对应了一组属性名及其属性值。end-element 事件的句柄只有一个参数，即标签名。

```
typedef void (* XML_EndElementHandler)(void * uData, const char * name)
```

最终，text 事件的句柄只接受文本作为额外的参数。该文本字符串并不是以“\0”字符作为字符串的结束符，而是必须显式地指定字符串的长度：

```
typedef void (* XML_CharacterDataHandler)(void * uData, const char * s,
                                          int len);
```

下列函数可以将文本数据传给 Expat：

```
int XML_Parse(XML_Parser p, const char * s, int len, int isFinal);
```

Expat 通过陆续地调用 XML_Parse 函数，逐段分析整个文档。函数 XML_Parse 的最后一个参数，即 isFinal，用于通知 Expat 当前文本段是否为该文档的最后一个部分。需要注意的是每个文本段并不需要以“\0”字符作为结束符，因为该文本段通过明确地指定长度来表明其结束的位置。如果 XML_Parse 在解析中遇到错误，它将返回 0（Expat 也提供了辅助函数来获取错误信息，出于简洁，在此我们将其忽略）。

我们所需的最后一个 Expat 函数用于设置将要传给事件句柄的用户数据：

```
void XML_SetUserData(XML_Parser p, void * uData);
```

现在让我们来看一下如何在 Lua 中使用 Expat 库。第一种方案是最直接的方案：将这些函数导入 Lua。更好的方案是将这些功能纳入 Lua，比如，Lua 是没有类型语言，因此无需为不同的回调函数设置不同的函数，实际上，我们可以完全避免回调函数的注册函数。这样，创建解析器时，我们将给出一个包含所有回调函数的句柄回调表，在该表中每个回调函数的句柄对应一个适当的索引。比如，如果只需要打印一个文档的布局，我们可以采用下列回调表：

```
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：

```
+ to
+   yes
-   yes
- to
```

通过这个 API，我们不再需要用于管理回调函数的函数，因为可以直接使用回调表对其进行管理。因此，整个 API 需要三个函数：创建解析器的函数、解析文本段的函数、关闭解析器的函数（实际上，最后两个函数将被作为解析器的方法来实现）。这些 API 的使用方法如下：

```
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如何实现上述功能。首先，考虑如何在 Lua 中表示解析器，我们很自然会想到 Userdata，但是如何定义该 Userdata 呢？至少，必须在其中定义 Expat 解析器和一个回调表。我们无法将 Lua 表保存于 Userdata（或任何 C 结构中），不过，可以创建一个指向 Lua 表的引用，并将其保存于 Userdata（在第 27.3.2 章节已有定义，引用是由 Lua 自动产生的用作注册表索引中的一个整数）。最后，还必须将 Lua 状态保存于解析器对象，因为 Expat 回调函数能够从程序中接收的信息只有解析器对象，并且这些回调函数需要调用 Lua。因此，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;
```

下面是用于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 的主体可以分成四个主要步骤：

第一步骤遵循一个常用的模式：首先创建一个 Userdata，然后使用通用的值预初始化 Userdata，最后设置 Userdata 的元表。预初始化的原因在于：如果初始化时发生任何错误，我们必须保证析构器，即__gc 元方法，能够通过 Userdata 的预初始化值发现是否发生了错误（如，预初始化 parser 为 NULL，如果之后初始化 parser 失败，那么 parser 仍然为 NULL，反过来，如果析构器发现 parser 为 NULL，则代表初始化发生了错误）。

第二步骤，创建一个 Expat 解析器，将器保存在 Userdata 中，并检测可能的错误。

第三步，保证函数的第一参数是表（回调表），创建指向该表的引用，并将其保存于 Userdata 中。

第四步，初始化 Expat 解析器，在此过程中，Userdata 将被设置为传递给回调函数的对象，并设置三类事件对应的回调函数。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 中不可能动态地创建新函数，取而代之的是，这些预定义的 C 函数将使用回调表来决定每次应该调用的 Lua 函数。

下一步是解析方法，该方法负责解析一段 XML 数据，它将接受两个参数：解析器对象（即方法的 self 属性所对应的对象）和一段可选的 XML 数据。如果忽略第可选的数据，调用该方法时，它将通知 Expat 该文档已经解析完毕：

```
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 时，后者会为它在 XML 文本段中找到相关元素调用对应的事件句柄，因此，lxp_parse 会首先为这些事件句柄准备运行环境。调用 XML_Parse 时需要注意：通过该函数的最后一个参数可以让 Expat 了解给定的文本段是否是最后一段。如果不带该参数调用 XML_Parse 时，s 的值为 NULL，因此最后一个参数将为真。

现在把注意力转移到回调函数 f_StartElement、f_EndElement 和 f_CharData 上，这三个函数有相似的结构：检查回调表中是否为该函数对应的事件定义了回调函数的句柄，如果已有定义，则生成参数并调用该句柄。

我们首先看一下 f_CharData 句柄，它的代码非常简单。它调用 Lua 中对应的句柄（如果存在的话），该句柄带有两个参数：解析器和字符数据（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，由于创建解析器时需调用 XML_SetUserData，所以，因此，此类由 C 语言编写的句柄都接受 lxp_userdata 数据结构作为第一参数。同时也要注意该句柄如何使用由 lxp_parse 设置的环境：首先，它假定回调表在栈中的索引为 3，其次，它假定解析器在栈中的索引为 1（必然如此，因为解析器是函数 lxp_parse 的第一参数）。

句柄 f_EndElement 和 f_CharData 类似，也很简单。它同样附带两个参数调用相应的 Lua 句柄：解析器和标签名（以“\0”字符结尾的字符串）：

```
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 调用 Lua 句柄时附带三个参数：解析器、标签名和一个属性列表。该句柄比其他两个稍显复杂，因为它需要将属性列表翻译成 Lua 能够识别的内容。我们采用一种相当自然的翻译方式，比如，类似下面的开始标签：

```
<to method="post" priority="high">
```

将被翻译成下列由属性列表组成的 Lua 表：

```
{method = "post", priority = "high"}
```

句柄 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。关闭解析器时，必须释放该解析器所占有的所有资源，即 Expat 结构和回调表。注意，如果早先在创建过程中发生了错误，那么解析器可能并不拥有这些资源：

```
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;
}
```

注意在关闭解析器时，需要保证它处于一致性状态。因此，重复关闭该解析器或垃圾收集器回收该解析器时，不会导致任何问题。实际上，该函数正是我们指定的析构函数，即使程序员没有关闭解析器，它也能确保每个解析器最终都能释放其占用的资源。

最后一步是打开库，将上述各部分组成整体。我们将使用与面向对象的数值数组（第 28.3 章节）一样的方案：创建一个包含所有方法的元表，并将自身赋值给该表的__index 域。为此，我们需要一个包含解析器方法的列表：

```
static const struct luaL_reg lxp_meths[] = {
    {"parse", lxp_parse},
    {"close", lxp_close},
    {"__gc", lxp_close},
    {NULL, NULL}
};
```

同时我们还需要一个包含库所有库函数的列表。和面向对象的库相同的是，该库只含有一个函数，该函数负责创建解析器：

```
static const struct luaL_reg lxp_funcs[] = {
    {"new", lxp_make_parser},
    {NULL, NULL}
};
```

最终，用于打开库的函数必须创建一个元表，通过__index 域指向元表自身，并注册方法和函数：

```
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;
}
```